

Python Fluente, 2ª edição
volume 3: Controle e Metaprogramação

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 3: Fluxo e Metaprogramação

1ª edição: 2015

2ª edição: 2022

Revisão: pyfl2-vol3-pb-2026-03-03.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 Ascidoctor

Ficha catalográfica: Edison Luís dos Santos

Publisher: Heinar Maracy @ Z•Edições

R135p Ramalho, Luciano.

Python Fluente, 2ª edição, volume 3: Fluxo e Metaprogramação /
Luciano Ramalho - São Paulo, SP - Z.Edições, 2025.

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

ISBN: 978-65-989778-1-8

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

4.Metaprogramação.

I.Título II.Fluxo e Metaprogramação III.RAMALHO, Luciano.

CDU: 004.438

CDD: 005.133

Para Marta, com todo o meu amor.

Sumário

Parte IV: Classes e Protocolos	1
17. Iteradores, geradores e corrotinas clássicas	3
17.1. Novidades neste capítulo	4
17.2. Uma sequência de palavras	4
17.3. Como a função <code>iter</code> devolve iteradores	7
17.4. Iteráveis versus iteradores	10
17.5. Classes <code>Sentence</code> com <code>__iter__</code>	15
17.6. Sentenças preguiçosas	23
17.7. Quando usar expressões geradoras	26
17.8. Um gerador de progressão aritmética	29
17.9. Funções geradoras na biblioteca padrão	33
17.10. Funções de redução de iteráveis	48
17.11. Subgeradores com <code>yield from</code>	50
17.12. Tipos iteráveis genéricos	60
17.13. Corrotinas clássicas	62
17.14. Resumo do capítulo	75
17.15. Para saber mais	76
18. Instruções <code>with</code> , <code>match</code> , e blocos <code>else</code>	83
18.1. Novidades neste capítulo	84
18.2. Instrução <code>with</code> e gerenciadores de contexto	84
18.3. Estudo de caso: <code>match/case</code> no <i>lis.py</i>	97
18.4. Faça isso, então aquilo: blocos <code>else</code> além do <code>if</code>	119
18.5. Resumo do capítulo	122
18.6. Para saber mais	123
19. Modelos de concorrência em Python	131
19.1. Novidades neste capítulo	132
19.2. A visão geral	133
19.3. Um pouco de jargão	134
19.4. Um "Olá mundo" concorrente	139
19.5. O verdadeiro impacto da GIL	152

19.6. Um pool de processos caseiro	156
19.7. Python no mundo multi-núcleo.	167
19.8. Resumo do capítulo	177
19.9. Para saber mais	178
20. Executores concorrentes	191
20.1. Novidades neste capítulo	191
20.2. Downloads concorrentes da Web	192
20.3. Processos com <code>concurrent.futures</code>	204
20.4. Experimento com <code>Executor.map</code>	209
20.5. Barra de progresso e tratamento de erros	212
20.6. Resumo do capítulo	225
20.7. Para saber mais	225
21. Programação assíncrona	229
21.1. Novidades neste capítulo	230
21.2. Algumas definições.	230
21.3. Sondando domínios com <code>asyncio</code>	232
21.4. Novo conceito: esperável (<i>awaitable</i>).	236
21.5. Downloads com <code>asyncio</code> e <i>HTTPX</i>	237
21.6. Gerenciadores de contexto assíncronos	242
21.7. Melhorando o download de bandeiras assíncrono.	244
21.8. Delegando tarefas a executores	256
21.9. Programando servidores assíncronos	258
21.10. Iteráveis assíncronos	271
21.11. Sondando domínios com <i>Curio</i>	284
21.12. Dicas de tipo para objetos assíncronos	288
21.13. Como a programação assíncrona funciona e como não funciona ..	290
21.14. Resumo do capítulo	292
21.15. Para saber mais	293
22. Atributos dinâmicos e propriedades	301
22.1. Novidades neste capítulo	302
22.2. Explorando dados com atributos dinâmicos	302
22.3. Propriedades computadas	313

22.4. Propriedades para validação de atributos	327
22.5. Propriedades em profundidade.....	330
22.6. Uma fábrica de propriedades.....	336
22.7. Tratando a exclusão de atributos	340
22.8. Atributos e funções essenciais para tratamento de atributos	342
22.9. Resumo do capítulo	346
22.10. Leitura Complementar	347
23. Descritores de Atributos	355
23.1. Novidades neste capítulo	356
23.2. Descritor para validação de atributos	356
23.3. Descritores dominantes ou não dominantes	369
23.4. Métodos são descritores	377
23.5. Dicas para usar descritores.....	380
23.6. Docstrings e exclusão de descritores	382
23.7. Resumo do capítulo	383
23.8. Para saber mais	384
24. Metaprogramação de classes	389
24.1. Novidades neste capítulo	390
24.2. Classes como objetos	390
24.3. type: a fábrica de classes embutida	392
24.4. Uma função fábrica de classes.....	394
24.5. Apresentando <code>__init_subclass__</code>	397
24.6. Um decorador de classes.....	406
24.7. O que acontece quando: importação versus execução	410
24.8. Introdução às metaclasses	416
24.9. Checked, agora com metaclasses.....	430
24.10. Metaclasses no mundo real	436
24.11. Um <i>hack</i> de metaclasses com <code>__prepare__</code>	440
24.12. Para encerrar.....	442
24.13. Resumo do capítulo	444
24.14. Para saber mais	445

Parte IV: Classes e Protocolos

Capítulo 17. Iteradores, geradores e corrotinas clássicas

Quando vejo padrões em meus programas, considero isso um mau sinal. A forma de um programa deve refletir apenas o problema que ele precisa resolver. Qualquer outra regularidade no código é, pelo menos para mim, um sinal de que estou usando abstrações que não são poderosas o suficiente— muitas vezes estou gerando à mão as expansões de alguma macro que preciso escrever.^[1]

— Paul Graham, hacker de Lisp e investidor

A iteração é fundamental para o processamento de dados: programas aplicam computações sobre séries de dados, de pixels a nucleotídeos. Se os dados não cabem na memória, precisamos buscar esses itens de forma *preguiçosa*—um de cada vez e sob demanda. É isso que um iterador faz. Este capítulo mostra como o padrão de projeto *Iterator* (Iterador) está embutido na linguagem Python, de modo que nunca será necessário programá-lo manualmente.

Todas as coleções padrão de Python são *iteráveis*. Um *iterável* é um objeto que fornece um *iterador*, que Python usa para suportar operações como:

- O laço `for`
- Compreensões de lista, `dict` e `set`
- Atribuições com desempacotamento de tuplas
- Criação de instâncias de coleções

Este capítulo cobre os seguintes tópicos:

- Como Python usa a função embutida `iter()` para lidar com objetos iteráveis
- Como é o padrão *Iterator* clássico escrito em Python
- Porque podemos substituir o padrão *Iterator* clássico por uma função geradora ou por uma expressão geradora
- Como funciona uma função geradora, em detalhes, linha a linha

- Como aproveitar o poder das funções geradoras de uso geral da biblioteca padrão
- Usando expressões `yield from` para combinar geradores
- Porque geradores e corrotinas clássicas se parecem, mas são usados de formas muito diferentes e não devem ser misturadas

17.1. Novidades neste capítulo

A Seção 17.11 agora inclui experimentos simples demonstrando o comportamento de geradores com `yield from`, e um exemplo de código para percorrer uma estrutura de dados em árvore, desenvolvido passo a passo.

Novas seções explicam as dicas de tipo para os tipos `Iterable`, `Iterator` e `Generator`.

A última grande seção do capítulo, Seção 17.13, é agora uma introdução de 9 páginas a um tópico que ocupava um capítulo de 40 páginas na primeira edição. Atualizei e publiquei «no site» [fpy.li/oldcoro] que acompanha o livro o capítulo *Classic Coroutines* da primeira edição (em inglês). Era o capítulo mais difícil do livro, mas ficou menos relevante após a introdução das corrotinas nativas no Python 3.5 (estudaremos as corrotinas nativas no Capítulo 21).

Vamos começar examinando como a função embutida `iter()` torna as sequências iteráveis.

17.2. Uma sequência de palavras

Vamos começar nossa exploração de iteráveis implementando uma classe `Sentence`: seu construtor recebe uma string de texto e daí podemos iterar sobre a "sentença" palavra por palavra. A primeira versão vai implementar o protocolo de sequência e será iterável, pois todas as sequências são iteráveis—como sabemos desde o «Capítulo 1» [fpy.li/1] (vol.1). Agora veremos exatamente por que isso acontece.

O Exemplo 1 mostra uma classe `Sentence` que permite ler as palavras de um texto por índice.

Exemplo 1. sentence.py: Sentence como uma sequência de palavras

```
import re
import reprlib

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

class Sentence:

    def __init__(self, text):
        self.text = text
        self.words = RE_WORD.findall(text) ①

    def __getitem__(self, index):
        return self.words[index] ②

    def __len__(self): ③
        return len(self.words)

    def __repr__(self):
        return 'Sentence(%)' % reprlib.repr(self.text) ④
```

- ① `.findall` devolve a lista com todos os trechos não sobrepostos correspondentes à expressão regular, como uma lista de strings.
- ② `self.words` preserva o resultado de `.findall`, então basta devolver a palavra em um dado índice.
- ③ Para completar o protocolo de sequência, implementamos `__len__`, apesar dele não ser necessário para criar um iterável.
- ④ `reprlib.repr` devolve representações abreviadas, como vimos ao implementar `__repr__` na classe `Vector` da «Seção 12.3» [fpy.li/cb] (vol.2).

Por default, `reprlib.repr` limita a string gerada a 30 caracteres. Veja como `Sentence` é usada na sessão de console do Exemplo 2.

Exemplo 2. Testando a iteração em uma instância de Sentence

```
>>> s = Sentence("The time has come," the Walrus said,') ①
>>> s
Sentence('The time ha... Walrus said,') ②
>>> for word in s: ③
...     print(word)
The
time
has
come
the
Walrus
said
>>> list(s) ④
['The', 'time', 'has', 'come', 'the', 'Walrus', 'said']
```

- ① Uma sentença criada a partir de uma string.
- ② Observe a saída de `__repr__` gerada por `reprlib.repr`, usando `'...'`.
- ③ Instâncias de `Sentence` são iteráveis; veremos a razão em seguida.
- ④ Sendo iteráveis, objetos `Sentence` podem ser usados como entrada para criar listas e outros tipos iteráveis.

Nas próximas páginas vamos desenvolver outras classes `Sentence` que passam nos testes do Exemplo 2. Entretanto, a implementação no Exemplo 1 difere das outras por ser também uma sequência, e então é possível obter palavras usando um índice:

```
>>> s[0]
'The'
>>> s[5]
'Walrus'
>>> s[-1]
'said'
```

Programadores Python sabem que sequências são iteráveis. Agora vamos descobrir exatamente o porquê disso.

17.3. Como a função `iter` devolve iteradores

Para iterar sobre um objeto `x`, o interpretador Python invoca `iter(x)`. A função embutida `iter`:

1. Verifica se o objeto implementa o método `__iter__`, e o invoca para obter um iterador.
2. Se `__iter__` não for implementado, mas `__getitem__` sim, então `iter` cria um iterador que tenta buscar itens pelo índice, a partir de 0 (zero).
3. Se isso falhar, Python gera um `TypeError`, com a mensagem `'C' object is not iterable` ("objeto 'C' não é iterável"), onde `C` é a classe do objeto alvo.

Por isso todas as sequências de Python são iteráveis: por definição, todas implementam `__getitem__`. Na verdade, todas as sequências padrão também implementam `__iter__`, e as classes de sequências que você criar também devem implementar este método. A iteração automática via `__getitem__` existe para manter a compatibilidade retroativa, e pode desaparecer em algum momento—mas eu duvido que será removida no futuro.

Como mencionado na «Seção 13.4.1» [fpy.li/cc] (vol.2), esta é uma forma extrema de tipagem pato: um objeto é considerado iterável não apenas quando implementa o método especial `__iter__`, mas também quando implementa `__getitem__`. Confira:

```
>>> class Spam:
...     def __getitem__(self, i):
...         print('->', i)
...         raise IndexError()
...
>>> spam_can = Spam()
>>> iter(spam_can)
<iterator object at 0x10a878f70>
>>> list(spam_can)
-> 0
[]
>>> from collections import abc
>>> isinstance(spam_can, abc.Iterable)
False
```

Se uma classe fornece `__getitem__`, a função embutida `iter()` aceita uma instância daquela classe como iterável e cria um iterador a partir da instância. A maquinaria de iteração de Python chamará `__getitem__` com índices, começando de 0, e entenderá um `IndexError` como sinal de que não há mais itens.

Observe que, apesar de `spam_can` ser iterável (seu método `__getitem__` poderia fornecer itens), ela não é reconhecida assim por uma chamada a `isinstance` contra `abc.Iterable`.

Na tipagem ganso (*goose typing*), a definição de um iterável é mais simples, mas não tão flexível: um objeto é considerado iterável se implementa o método `__iter__`. Não é necessário ser subclasse ou se registrar como subclasse virtual, pois `abc.Iterable` implementa o `__subclasshook__`, como visto na «Seção 13.5.8» [fpy.li/cd] (vol.2). Demonstração:

```
>>> class GooseSpam:
...     def __iter__(self):
...         pass
...
>>> from collections import abc
>>> isinstance(GooseSpam, abc.Iterable)
True
>>> goose_spam_can = GooseSpam()
>>> isinstance(goose_spam_can, abc.Iterable)
True
```



Desde o Python 3.10, a forma mais precisa de checar se um objeto `x` é iterável é invocar `iter(x)` e tratar a exceção `TypeError` se ele não for. Isso é mais preciso que usar `isinstance(x, abc.Iterable)`, porque `iter(x)` também leva em consideração o método legado `__getitem__`, enquanto a ABC `Iterable` não considera tal método.

Verificar explicitamente se um objeto é iterável pode não valer a pena, se você for iterar sobre o objeto logo após a checagem. Afinal, quando se tenta iterar sobre um não-iterável, a exceção gerada pelo Python é bem explícita: `TypeError: 'C' object is not iterable` (o objeto 'C' não é iterável). Se você quiser fazer algo além de gerar um `TypeError`, então faça isso em um bloco `try/except` ao invés de

realizar uma checagem explícita. A checagem explícita pode fazer sentido se você estiver guardando o objeto para iterar sobre ele mais tarde; neste caso, falhar logo facilita o diagnóstico de erros.

A função embutida `iter()` é usada mais pelo próprio Python do que em código que nós escrevemos. Vejamos outra forma de usá-la, menos conhecida.

17.3.1. Usando `iter` com um invocável

Podemos chamar `iter()` com dois argumentos, para criar um iterador a partir de uma função ou de qualquer objeto invocável. Nesta forma de uso, o primeiro argumento deve ser um invocável que será invocado sem argumentos repetidamente para produzir valores, e o segundo argumento é um «sentinel value» [fpy.li/17-2] (valor sentinela): um valor que, quando devolvido pelo invocável, faz o iterador gerar um `StopIteration` ao invés de produzir o valor sentinela.

O exemplo a seguir mostra como usar `iter` para rolar um dado de seis faces enquanto o valor 1 não é sorteado:

```
>>> def d6():
...     return randint(1, 6)
...
>>> d6_iter = iter(d6, 1)
>>> d6_iter
<callable_iterator object at 0x10a245270>
>>> for roll in d6_iter:
...     print(roll)
...
4
3
6
3
```

Observe que a função `iter` devolve um `callable_iterator`. O laço `for` no exemplo pode rodar por um longo tempo, mas nunca vai devolver 1, pois esse é o valor sentinela. Como é comum com iteradores, o objeto `d6_iter` se torna inútil após ser esgotado. Para recomençar, é necessário reconstruir o iterador, invocando novamente `iter()`.

A «documentação de `iter`» [fpy.li/97] inclui a seguinte explicação e código de exemplo:^[2]

Uma aplicação útil da segunda forma de `iter()` é construir um leitor bloco-a-bloco (block reader). Por exemplo, ler blocos de 64 bytes de um arquivo binário de banco de dados até que o final do arquivo seja atingido:

```
from functools import partial

with open('mydata.db', 'rb') as f:
    read_block = partial(f.read, 64)
    for block in iter(read_block, b''):
        process_block(block)
```

Para deixar o código mais fácil de ler, adicionei a atribuição `read_block`, que não está no «exemplo original» [fpy.li/97]. A função `partial()` é necessária porque o invocável passado a `iter()` não pode requerer argumentos. No exemplo, um objeto bytes vazio é a sentinela, pois é isso que `f.read` devolve quando não há mais bytes para ler. A variável `block` pode receber menos de 64 bytes uma vez no final do arquivo, mas nunca receberá 0 bytes, porque `b''` é o valor sentinela.

A próxima seção detalha a relação entre iteráveis e iteradores.

17.4. Iteráveis versus iteradores

Da explicação na Seção 17.3 podemos extrapolar a seguinte definição:

iterável

Qualquer objeto a partir do qual a função embutida `iter` consegue obter um iterador. Objetos que implementam um método `__iter__` devolvendo um iterador são iteráveis. Sequências são sempre iteráveis, bem como objetos que implementam um método `__getitem__` que aceite índices iniciando em 0.

É importante deixar clara a relação entre iteráveis e iteradores: Python obtém um iterador a partir de um iterável.

Aqui está um simples laço `for` iterando sobre uma `str`. A `str` `'ABC'` é o iterável aqui. Você não vê, mas há um iterador por trás das cortinas:

```
>>> s = 'ABC'
>>> for char in s:
...     print(char)
...
A
B
C
```

Se não existisse uma instrução `for` e fosse preciso emular o mecanismo do `for` à mão com um laço `while`, isso é o que teríamos que escrever:

```
>>> s = 'ABC'
>>> it = iter(s) ①
>>> while True:
...     try:
...         char = next(it) ②
...     except StopIteration: ③
...         del it ④
...         break ⑤
...     print(char) ⑥
A
B
C
```

- ① Cria um iterador `it` a partir de um iterável.
- ② Chama `next` repetidamente com o iterador, para obter o item seguinte.
- ③ O iterador gera `StopIteration` quando não há mais itens.
- ④ Libera a referência a `it`—o objeto iterador é descartado.
- ⑤ Sai do laço.
- ⑥ Exibe `char`. Esta variável continua existindo depois do laço.

`StopIteration` sinaliza que o iterador esgotou. Esta exceção é tratada internamente pelo Python, dentro da lógica dos laços `for` e de outros contextos de iteração, como compreensões de lista, desempacotamento de iteráveis, etc.

A interface padrão de um iterador em Python tem dois métodos:

`__next__`

Devolve o próximo item da série, gerando `StopIteration` se não há mais nenhum.

`__iter__`

Devolve `self`; assim o iterador pode ser usado quando um iterável é esperado. Por exemplo, em um laço `for`.

Esta interface está formalizada na ABC `collections.abc.Iterator`, que declara o método abstrato `__next__`, e é uma subclasse de `Iterable`—onde o método abstrato `__iter__` é declarado. Veja a Figura 1.

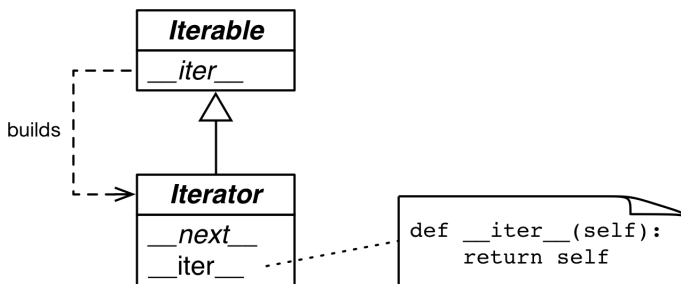


Figura 1. As ABCs `Iterable` e `Iterator`. Métodos em itálico são abstratos. Um `Iterable.__iter__` concreto deve devolver uma nova instância de `Iterator`. Um `Iterator` concreto deve implementar `__next__`. O método `Iterator.__iter__` apenas devolve a própria instância.

O código-fonte de `collections.abc.Iterator` aparece no Exemplo 3.

Exemplo 3. Classe `abc.Iterator`; extraído de `Lib/_collections_abc.py` [fpy.li/17-5]

```
class Iterator(Iterable):

    __slots__ = ()

    @abstractmethod
    def __next__(self):
        """Return the next item from the iterator.
        When exhausted, raise StopIteration"""
        raise StopIteration

    def __iter__(self):
        return self
```



```
@classmethod
def __subclasshook__(cls, C): ①
    if cls is Iterator:
        return _check_methods(C, '__iter__', '__next__') ②
    return NotImplemented
```

- ① `__subclasshook__` suporta a checagem de tipos estrutural com `isinstance` e `issubclass`. Vimos isso na «Seção 13.5.8» [fpy.li/cd] (vol.2).
- ② `_check_methods` percorre o atributo `__mro__` da classe, para checar se os métodos estão implementados em alguma superclasse. Ele está definido no mesmo módulo, *Lib/collections_abc.py*. Se os métodos estiverem implementados, a classe `C` será reconhecida como uma subclasse virtual de `Iterator`. Em outras palavras, `issubclass(C, Iterable)` devolverá `True`.



O método abstrato da ABC `Iterator` é `it.__next__()` no Python 3 e `it.next()` no Python 2. Como sempre, você deve evitar invocar métodos especiais diretamente. Use apenas `next(it)`: essa função embutida faz a coisa certa no Python 2 e no 3—algo útil para quem está migrando bases de código do 2 para o 3.

O código-fonte do módulo *Lib/types.py* [fpy.li/17-6] no Python 3.9 tem um comentário dizendo:

Iteradores no Python não são uma questão de tipo, mas sim de protocolo. Um número grande e variável de tipos embutidos implementa *alguma* forma de iterador. Não verifique o tipo! Em vez disso, use `'hasattr'` para detectar os atributos `"__iter__"` e `"__next__"`.

Isto é exatamente o que o método `__subclasshook__` da ABC `abc.Iterator` faz.



Dado o conselho de *Lib/types.py* e a lógica implementada em *Lib/collections_abc.py*, a melhor forma de checar se um objeto `x` é um iterador é invocar `isinstance(x, abc.Iterator)`. Graças ao `Iterator.__subclasshook__`, este teste funciona mesmo quando a classe de `x` não é uma subclasse real ou virtual de `Iterator`.

Voltando à nossa classe `Sentence` no Exemplo 1, usando o console de Python podemos ver claramente como o iterador é criado por `iter()` e consumido por `next()`:

```
>>> s3 = Sentence('Life of Brian') ①
>>> it = iter(s3) ②
>>> it # doctest: +ELLIPSIS
<iterator object at 0x...>
>>> next(it) ③
'Life'
>>> next(it)
'of'
>>> next(it)
'Brian'
>>> next(it) ④
Traceback (most recent call last):
...
StopIteration
>>> list(it) ⑤
[]
>>> list(iter(s3)) ⑥
['Life', 'of', 'Brian']
```

- ① Cria uma sentença `s3` com três palavras.
- ② Obtém um iterador a partir de `s3`.
- ③ `next(it)` devolve a próxima palavra.
- ④ Não há mais palavras, então o iterador gera uma exceção `StopIteration`.
- ⑤ Uma vez esgotado, um iterador vai sempre lançar `StopIteration`, indicando que não há mais itens.
- ⑥ Para percorrer a sentença novamente, precisamos criar um novo iterador.

Como os únicos métodos exigidos de um iterador são `__next__` e `__iter__`, não há como checar se há itens restantes, exceto invocando `next()` e capturando `StopIteration`. Além disso, não é possível "reiniciar" um iterador. Se precisar começar de novo, invoque `iter()` novamente no iterável que criou o iterador original. Invocar `iter()` no próprio iterador esgotado não funciona, pois—como já mencionado—a implementação de `Iterator.__iter__` apenas devolve `self`, e isso não reinicia o iterador.

Esta interface minimalista faz sentido porque, na realidade, nem todos os iteradores são reiniciáveis. Por exemplo, se um iterador está lendo pacotes da rede, não há como "rebobiná-lo".^[3]

A primeira versão de *Sentence*, no Exemplo 1, era iterável graças ao tratamento especial dispensado pela função `iter` às sequências. A seguir, vamos codar variações de *Sentence* que implementam `__iter__` para devolver iteradores.

17.5. Classes *Sentence* com `__iter__`

As próximas variantes de *Sentence* implementam o protocolo iterável padrão, primeiro implementando o padrão de projeto *Iterable* e depois com funções geradoras.

17.5.1. *Sentence* versão #2: um iterador clássico

A próxima implementação de *Sentence* segue a forma do padrão de projeto *Iterator* clássico, do livro *Padrões de Projeto*. Observe que isso não é Python idiomático, como as refatorações seguintes deixarão claro. Mas é útil para mostrar a distinção entre uma coleção iterável e um iterador que trabalha com ela.

A classe *Sentence* no Exemplo 4 é iterável por implementar o método especial `__iter__`, que cria e devolve um *SentenceIterator*. É assim que um iterável e um iterador se relacionam.

Exemplo 4. sentence_iter.py: Sentence implementada usando o padrão Iterator

```
import re
import replib

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

class Sentence:

    def __init__(self, text):
        self.text = text
        self.words = RE_WORD.findall(text)
```

```

def __repr__(self):
    return f'Sentence({reprlib.repr(self.text)})'

def __iter__(self): ①
    return SentenceIterator(self.words) ②

class SentenceIterator:

    def __init__(self, words):
        self.words = words ③
        self.index = 0 ④

    def __next__(self):
        try:
            word = self.words[self.index] ⑤
        except IndexError:
            raise StopIteration() ⑥
        self.index += 1 ⑦
        return word ⑧

    def __iter__(self): ⑨
        return self

```

- ① O método `__iter__` é o único acréscimo à implementação anterior de `Sentence`. Esta versão não tem `__getitem__`, para deixar claro que a classe é iterável por implementar `__iter__`.
- ② `__iter__` atende ao protocolo iterável instanciando e devolvendo um iterador.
- ③ `SentenceIterator` preserva uma referência para a lista de palavras.
- ④ `self.index` determina a próxima palavra a ser recuperada.
- ⑤ Obtém a palavra em `self.index`.
- ⑥ Se não há palavra em `self.index`, levanta `StopIteration`.
- ⑦ Incrementa `self.index`.
- ⑧ Devolve a palavra.
- ⑨ Implementa `self.__iter__` para suportar `iter(self)`.

O código do Exemplo 4 passa nos testes do Exemplo 2.

Veja que não é de fato necessário implementar `__iter__` em `SentenceIterator` para este exemplo funcionar, mas é recomendado: supõe-se que iteradores implementem tanto `__next__` quanto `__iter__`, e fazer isso permite ao nosso iterador passar no teste `issubclass(SentenceIterator, abc.Iterator)`. Se tivéssemos tornado `SentenceIterator` uma subclasse de `abc.Iterator`, teríamos herdado o método concreto `abc.Iterator.__iter__`.

É bastante trabalho (pelo menos para nós, programadores mimados pelo Python). Observe que a maior parte do código em `SentenceIterator` serve para gerenciar o estado interno do iterador. Logo veremos como evitar essa burocracia. Mas antes, um pequeno desvio para tratar de um atalho de implementação que pode parecer tentador, mas é apenas errado.

17.5.2. Não torne o iterável também um iterador

Uma causa comum de erros na criação de iteráveis e iteradores é confundir os dois. Para deixar claro: um iterável tem um método `__iter__` que instancia um novo iterador a cada invocação. Um iterador implementa um método `__next__`, que devolve itens individuais, e um método `__iter__`, que devolve `self`.

Assim, iteradores também são iteráveis, mas iteráveis não são iteradores.

Pode ser tentador implementar `__next__` além de `__iter__` na classe `Sentence`, tornando cada instância de `Sentence` ao mesmo tempo um iterável e um iterador de si mesma. Mas raramente isso é uma boa ideia. Também é um anti-padrão comum, de acordo com Alex Martelli, que tem vasta experiência revisando código no Google.

A seção "Aplicabilidade" do padrão de projeto *Iterator* no livro *Padrões de Projeto* diz:

Use o padrão Iterator para:

- *Acessar o conteúdo de um objeto agregado sem expor sua representação interna.*
- *Suportar travessias múltiplas de objetos agregados.*
- *Fornecer uma interface uniforme para atravessar diferentes estruturas agregadas (isto é, para suportar iteração polimórfica).*

Para "suportar travessias múltiplas", deve ser possível obter múltiplos iteradores independentes a partir de um mesmo objeto iterável, e cada iterador deve preservar seu próprio estado interno. Assim, uma implementação adequada do padrão exige que cada invocação de `iter(meu_iterável)` crie um novo iterador independente. Por isto precisamos da classe `SentenceIterator` neste exemplo.

Agora que sabemos como funciona o padrão *Iterator* clássico, veremos que não precisamos "escrever à mão" nenhuma classe de iterador em Python, graças à instrução `yield`, que foi inspirada pela linguagem *CLU* [fpy.li/17-7], criada por um time liderado por Barbara Liskov.

As próximas seções apresentam versões mais idiomáticas de `Sentence`.

17.5.3. `Sentence` versão #3: uma função geradora

Uma implementação pythônica da mesma funcionalidade usa um gerador, evitando todo o trabalho para implementar a classe `SentenceIterator`. A explicação completa do gerador está logo após o Exemplo 5.

Exemplo 5. `sentence_gen.py`: `Sentence` implementada usando um gerador

```
import re
import reprlib

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

class Sentence:

    def __init__(self, text):
        self.text = text
        self.words = RE_WORD.findall(text)

    def __repr__(self):
        return 'Sentence(%s)' % reprlib.repr(self.text)

    def __iter__(self):
        for word in self.words: ①
            yield word ② ③

# done! ④
```

- ① Itera sobre `self.words`.
- ② Produz a `word` atual.
- ③ Um `return` explícito não é necessário. Uma função geradora não gera `StopIteration`: ela simplesmente termina quando acaba de produzir valores.^[4]
- ④ Não precisamos escrever uma classe iteradora!

Temos aqui mais uma implementação de `Sentence` que passa nos testes do Exemplo 2. No código de `Sentence` do Exemplo 4, `__iter__` chamava o construtor `SentenceIterator` para criar e devolver um iterador. Agora o iterador do Exemplo 5 é um objeto gerador, criado automaticamente quando o método `__iter__` é invocado, porque neste exemplo `__iter__` é uma função geradora.

A seguir: uma explicação bem completa sobre geradores.

17.5.4. Como funciona um gerador

Qualquer função de Python contendo a instrução `yield` em seu corpo é uma função geradora: uma função que, quando invocada, devolve um objeto gerador. Em outras palavras, uma função geradora é uma fábrica de geradores.



O único elemento sintático que identifica uma função geradora é a presença da instrução `yield` em algum lugar de seu corpo. Alguns defenderam que uma nova palavra reservada (como `gen`), deveria ser usada no lugar de `def` para declarar funções geradoras, mas Guido não concordou. Seus argumentos estão na *PEP 255—Simple Generators* [fpy.li/pep255] (Geradoras Simples).^[5]

O Exemplo 6 mostra o comportamento de uma função geradora simples.^[6]

Exemplo 6. Uma função geradora que produz três números

```
>>> def gen_123():  
...     yield 1 ①  
...     yield 2  
...     yield 3  
...
```

```

>>> gen_123 # doctest: +ELLIPSIS
<function gen_123 at 0x...> ②
>>> gen_123() # doctest: +ELLIPSIS
<generator object gen_123 at 0x...> ③
>>> for i in gen_123(): ④
...     print(i)
1
2
3
>>> g = gen_123() ⑤
>>> next(g) ⑥
1
>>> next(g)
2
>>> next(g)
3
>>> next(g) ⑦
Traceback (most recent call last):
...
StopIteration

```

- ① O corpo de uma função geradora muitas vezes contém `yield` dentro de um laço, mas não necessariamente; aqui eu apenas repeti `yield` três vezes.
- ② Olhando mais de perto, vemos que `gen_123` é um objeto função.
- ③ Mas quando invocado, `gen_123()` devolve um objeto gerador.
- ④ Objetos geradores implementam a interface `Iterator`, então são também iteráveis.
- ⑤ Atribuímos esse novo objeto gerador a `g`, para podermos testar seu funcionamento.
- ⑥ Como `g` é um iterador, chamar `next(g)` obtém o próximo item produzido por `yield`.
- ⑦ Quando a função geradora termina, o objeto gerador levanta uma `StopIteration`.

Uma função geradora cria um objeto gerador que encapsula o corpo da função. Quando invocamos `next()` no objeto gerador, a execução avança para o próximo `yield` no corpo da função, e a chamada a `next()` resulta no valor produzido quando o corpo da função é suspenso. Por fim, o objeto gerador externo criado

pelo Python levanta `StopIteration` quando a função retorna, de acordo com o protocolo `Iterator`.



Acho útil ser rigoroso ao falar sobre valores obtidos a partir de um gerador. É confuso dizer que um gerador "devolve" ou "retorna" valores. Funções devolvem valores. A chamada a uma função geradora devolve um gerador. Um gerador produz (*yields*) valores. Um gerador não "devolve" valores no sentido comum do termo: a instrução `return` no corpo de uma função geradora faz o objeto gerador levantar `StopIteration`. Se você escrever `return x` na função geradora, é possível obter o valor de `x` embrulhado na exceção `StopIteration`, mas normalmente isso é feito usando a sintaxe `yield from`, como veremos na Seção 17.13.2.

O Exemplo 7 torna mais explícita a interação entre um laço `for` e o corpo da função geradora.

Exemplo 7. Uma função geradora que exibe mensagens quando roda

```
>>> def gen_AB():
...     print('start')
...     yield 'A'           ①
...     print('continue')
...     yield 'B'           ②
...     print('end.')       ③
...
>>> for c in gen_AB():     ④
...     print('-->', c)    ⑤
...
start                        ⑥
--> A                        ⑦
continue                     ⑧
--> B                        ⑨
end.                         ⑩
>>>                         ⑪
```

- ① A primeira chamada implícita a `next()` no laço `for` em ④ vai exibir 'start' e parar no primeiro `yield`, produzindo o valor 'A'.

- ② A segunda chamada implícita a `next()` no laço `for` vai exibir `'continue'` e parar no segundo `yield`, produzindo o valor `'B'`.
- ③ A terceira chamada a `next()` vai exibir `'end.'` e continuar até o final do corpo da função, fazendo com que o objeto gerador levante uma `StopIteration`.
- ④ Para iterar, o mecanismo do `for` faz o equivalente a `g = iter(gen_AB())` para obter um objeto gerador, e daí `next(g)` a cada iteração.
- ⑤ O laço exibe `-->` e o valor devolvido por `next(g)`. Esse resultado só aparece após a saída das chamadas `print` dentro da função geradora.
- ⑥ O texto `start` vem de `print('start')` no corpo do gerador.
- ⑦ `yield 'A'` no corpo do gerador produz o valor `'A'` consumido pelo laço `for`, que é atribuído à variável `c` e resulta na saída `--> A`.
- ⑧ A iteração continua com a segunda chamada a `next(g)`, avançando no corpo do gerador de `yield 'A'` para `yield 'B'`. O texto `continue` é gerado pelo segundo `print` no corpo do gerador.
- ⑨ `yield 'B'` produz o valor `'B'` consumido pelo laço `for`, que é atribuído à variável `c` do laço, que então exibe `--> B`.
- ⑩ A iteração continua com uma terceira chamada a `next(it)`, avançando para o final do corpo da função. O texto `end.` é exibido por causa do terceiro `print` no corpo do gerador.
- ⑪ Quando a função geradora chega ao final, o objeto gerador levanta uma `StopIteration`. O mecanismo do laço `for` captura essa exceção, e o laço encerra naturalmente.

Espero agora ter deixado claro como `Sentence.__iter__` no Exemplo 5 funciona: `__iter__` é uma função geradora que, quando invocada, cria um objeto gerador que implementa a interface `Iterator`, então a classe `SentenceIterator` não é mais necessária.

A segunda versão de `Sentence` é mais concisa que a primeira, mas não é tão preguiçosa quanto poderia ser. Atualmente, a *preguiça* é considerada uma virtude, pelo menos em linguagens de programação e APIs. Uma implementação preguiçosa adia a produção de valores até o último momento possível. Isso economiza memória e também pode evitar o desperdício de ciclos de CPU.

A seguir, criaremos classes `Sentence` preguiçosas.

17.6. Sentenças preguiçosas

As últimas variações de `Sentence` são preguiçosas, valendo-se de uma função geradora do módulo `re`.

17.6.1. `Sentence` versão #4: um gerador preguiçoso

A interface `Iterator` foi projetada para ser preguiçosa: `next(my_iterator)` produz um item por vez. O oposto de preguiçosa é ávida: avaliação preguiçosa (*lazy evaluation*) e avaliação ávida (*eager evaluation*) são termos técnicos da teoria das linguagens de programação.^[7]

Até aqui, nossas implementações de `Sentence` não são preguiçosas, pois o `__init__` cria avidamente uma lista com todas as palavras no texto, vinculando-as ao atributo `self.words`. Isso exige o processamento do texto inteiro, e a lista pode acabar usando tanta memória quanto o próprio texto (provavelmente mais: vai depender de quantos caracteres que não fazem parte de palavras existirem no texto). A maior parte deste trabalho será inútil se o usuário iterar apenas sobre as primeiras palavras. Se você está se perguntando se "Existiria uma forma preguiçosa de fazer isso em Python?", a resposta muitas vezes é "Sim".

A função `re.finditer` é uma versão preguiçosa de `re.findall`. Em vez de uma lista, `re.finditer` devolve um gerador que produz instâncias de `re.MatchObject` sob demanda. Se existirem muitos itens, `re.finditer` economiza muita memória. Com ela, nossa terceira versão de `Sentence` agora é preguiçosa: ela só lê a próxima palavra do texto quando necessário. O código está no Exemplo 8.

Exemplo 8. `sentence_gen2.py`: `Sentence` implementada usando uma função geradora que invoca a função geradora `re.finditer`

```
import re
import replib

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

class Sentence:

    def __init__(self, text):
```

```

        self.text = text ①

    def __repr__(self):
        return f'Sentence({reprlib.repr(self.text)})'

    def __iter__(self):
        for match in RE_WORD.finditer(self.text): ②
            yield match.group() ③

```

① Não é necessário manter uma lista `words`.

② `finditer` cria um iterador sobre os termos encontrados com `RE_WORD` em `self.text`, produzindo instâncias de `MatchObject`.

③ `match.group()` extrai o texto da instância de `MatchObject`.

Geradores são um ótimo atalho, mas o código pode ser ainda mais conciso com uma expressão geradora.

17.6.2. Sentence versão #5: Expressão geradora preguiçosa

Podemos substituir funções geradoras simples—como aquela na última classe `Sentence` (no Exemplo 8)—por uma expressão geradora. Assim como uma compreensão de lista cria listas, uma expressão geradora cria objetos geradores. O Exemplo 9 compara o comportamento nos dois casos.

Exemplo 9. A função geradora `gen_AB` é usada primeiro por uma compreensão de lista, depois por uma expressão geradora

```

>>> def gen_AB(): ①
...     print('start')
...     yield 'A'
...     print('continue')
...     yield 'B'
...     print('end.')
...
>>> res1 = [x*3 for x in gen_AB()] ②
start
continue
end.
>>> for i in res1: ③
...     print('-->', i)

```

```

...
--> AAA
--> BBB
>>> res2 = (x*3 for x in gen_AB()) ④
>>> res2
<generator object <genexpr> at 0x10063c240>
>>> for i in res2: ⑤
...     print('-->', i)
...
start      ⑥
--> AAA
continue
--> BBB
end.

```

- ① Esta é a mesma função `gen_AB` do Exemplo 7.
- ② A compreensão de lista itera avidamente sobre os itens produzidos pelo objeto gerador devolvido por `gen_AB()`: 'A' e 'B'. Observe a saída nas linhas seguintes: `start`, `continue`, `end`.
- ③ Este laço `for` itera sobre a lista `res1` criada pela compreensão de lista.
- ④ A expressão geradora devolve `res2`, um objeto gerador. O gerador não é consumido aqui.
- ⑤ Este gerador obtém itens de `gen_AB` apenas quando o laço `for` itera sobre `res2`. Cada iteração do laço `for` invoca, implicitamente, `next(res2)`, que por sua vez invoca `next()` sobre o objeto gerador devolvido por `gen_AB()`, fazendo este último avançar até o próximo `yield`.
- ⑥ Observe como a saída de `gen_AB()` se intercala com a saída do `print` no laço `for`.

Podemos usar uma expressão geradora para reduzir ainda mais o código na classe `Sentence`. Veja o Exemplo 10.

Exemplo 10. `sentence_genexp.py`: `Sentence` implementada usando uma expressão geradora

```

import re
import replib

```

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

```
class Sentence:
```

```
    def __init__(self, text):  
        self.text = text
```

```
    def __repr__(self):  
        return f'Sentence({reprlib.repr(self.text)})'
```

```
    def __iter__(self):  
        return (match.group() for match in RE_WORD.finditer(self.text))
```

A única diferença com o Exemplo 8 é o método `__iter__`, que aqui não é uma função geradora (ela não contém uma instrução `yield`) mas usa uma expressão geradora para criar um gerador e devolvê-lo. O resultado final é o mesmo: quem invoca `__iter__` recebe um objeto gerador.

Expressões geradoras são "açúcar sintático": sempre podem ser substituídas por funções geradoras, mas às vezes são mais convenientes. A próxima seção trata do uso de expressões geradoras.

17.7. Quando usar expressões geradoras

Usei expressões geradoras quando implementamos a classe `Vector` no Exemplo 16 do «Capítulo 12» [fpy.li/12] (vol.2). Estes métodos contêm expressões geradoras: `__eq__`, `__hash__`, `__abs__`, `angle`, `angles`, `format`, `__add__`, e `__mul__`. Em todos eles, uma compreensão de lista também funcionaria, usando mais memória para armazenar os valores da lista intermediária.

No Exemplo 10, vimos que uma expressão geradora é um atalho sintático para criar um gerador sem definir e invocar uma função. Por outro lado, funções geradoras são mais flexíveis: podemos programar uma lógica complexa, com várias instruções, e podemos até usá-las como *corrotinas*, como veremos na Seção 17.13.

Nos casos mais simples, uma expressão geradora é mais fácil de ler de relance, como mostra o exemplo de `Vector`.

Minha regra básica para escolher qual sintaxe usar é simples: se a expressão geradora exige mais que um par de linhas, prefiro escrever uma função geradora, em nome da legibilidade.



Dica de sintaxe

Quando uma expressão geradora é passada como único argumento a uma função ou construtor, não é necessário escrever um par de parênteses para invocar função e outro par ao redor da expressão geradora. Um único par é suficiente, como na invocação do construtor `Vector` no método `__mul__` do Exemplo 16 do «Capítulo 12» [fpy.li/12] (vol.2), reproduzido abaixo:

```
def __mul__(self, scalar):
    if isinstance(scalar, numbers.Real):
        return Vector(n * scalar for n in self)
    else:
        return NotImplemented
```

Entretanto, se a invocação exigir mais argumentos após a expressão geradora, é preciso cercá-la com parênteses para evitar um `SyntaxError`.

Os exemplos de `Sentence` vistos até aqui mostram geradores fazendo o papel do padrão `Iterator` clássico: obter itens de uma coleção. Mas podemos também usar geradores para produzir valores sem acessar uma estrutura de dados. A próxima seção mostra um exemplo.

Mas antes, uma pequena discussão sobre os conceitos sobrepostos de *iterador* e *gerador*.

Comparando iteradores e geradores

Na documentação e na base de código oficiais de Python, a terminologia em torno de iteradores e geradores é inconsistente e está em evolução. Adotei as seguintes definições:

iterador

Termo geral para qualquer objeto que implementa um método `__next__`. Iteradores são projetados para produzir dados a serem consumidos pelo código cliente, isto é, o código que controla o iterador através de um laço `for` ou outro mecanismo de iteração, ou chamando `next(it)` explicitamente no iterador—apesar desse uso explícito incomum. Na prática, a maioria dos iteradores que usamos no Python são *geradores*.

gerador

Um iterador criado pelo compilador do Python. Para criar um gerador, não implementamos uma classe com `__next__`. Em vez disso, usamos a palavra reservada `yield` para criar uma *função geradora*, que é uma fábrica de *objetos geradores*. Uma *expressão geradora* é outra maneira de criar um objeto gerador. Objetos geradores fornecem `__next__`, portanto são iteradores. Desde o Python 3.5, também temos *geradores assíncronos*, declarados com `async def`. Vamos estudá-los no Capítulo 21.

O *Glossário de Python* [fpy.li/99] introduziu recentemente o termo «iterador gerador» [fpy.li/9a] para se referir a objetos geradores criados por funções geradoras, enquanto o verbete para «expressão geradora» [fpy.li/9b] diz que ela devolve um "iterador".

Mas, para o interpretador Python, os objetos devolvidos em ambos os casos são objetos geradores:

```
>>> def g():
...     yield 0
...
>>> g()
<generator object g at 0x10e6fb290>
>>> ge = (c for c in 'XYZ')
>>> ge
<generator object <genexpr> at 0x10e936ce0>
>>> type(g()), type(ge)
(<class 'generator'>, <class 'generator'>)
```


17.8. Um gerador de progressão aritmética

O padrão *Iterator* clássico trata de navegar por uma estrutura de dados. Sua interface padrão é um método que fornece o próximo item de uma série. Tal interface também é útil quando os itens são produzidos sob demanda, ao invés de serem lidos de uma coleção. Por exemplo, a função embutida `range` gera uma progressão aritmética (PA) de inteiros. E se precisarmos gerar uma PA com números de qualquer tipo, não apenas inteiros?

O Exemplo 11 mostra alguns testes no console com uma classe `ArithmeticProgression`, que veremos em breve. A assinatura do construtor no Exemplo 11 é `ArithmeticProgression(begin, step[, end])`. A assinatura completa da função embutida `range` é `range(start, stop[, step])`. Escolhi implementar uma assinatura diferente porque o `step` é obrigatório, mas o `end` é opcional. Também mudei os nomes dos argumentos `start/stop` para `begin/end`, para deixar claro que optei por uma assinatura diferente. Para cada teste no Exemplo 11, chamo `list()` com o resultado para exibir os valores gerados.

Exemplo 11. Demonstração de uma classe `ArithmeticProgression`

```
>>> ap = ArithmeticProgression(0, 1, 3)
>>> list(ap)
[0, 1, 2]
>>> ap = ArithmeticProgression(1, .5, 3)
>>> list(ap)
[1.0, 1.5, 2.0, 2.5]
>>> ap = ArithmeticProgression(0, 1/3, 1)
>>> list(ap)
[0.0, 0.3333333333333333, 0.6666666666666666]
>>> from fractions import Fraction
>>> ap = ArithmeticProgression(0, Fraction(1, 3), 1)
>>> list(ap)
[Fraction(0, 1), Fraction(1, 3), Fraction(2, 3)]
>>> from decimal import Decimal
>>> ap = ArithmeticProgression(0, Decimal('.1'), .3)
>>> list(ap)
[Decimal('0'), Decimal('0.1'), Decimal('0.2')]
```

Observe que o tipo dos números na progressão aritmética resultante segue o tipo de `begin + step`, de acordo com as regras de coerção numérica da aritmética de

Python. No Exemplo 11, você pode ver listas de números `int`, `float`, `Fraction`, e `Decimal`. O Exemplo 12 mostra a implementação da classe `ArithmeticProgression`.

Exemplo 12. A classe `ArithmeticProgression`

```
class ArithmeticProgression:

    def __init__(self, begin, step, end=None):           ①
        self.begin = begin
        self.step = step
        self.end = end # None -> "infinite" series

    def __iter__(self):
        result_type = type(self.begin + self.step)      ②
        result = result_type(self.begin)                 ③
        forever = self.end is None                       ④
        index = 0
        while forever or result < self.end:              ⑤
            yield result                                  ⑥
            index += 1
            result = self.begin + self.step * index      ⑦
```

- ① `__init__` exige dois argumentos: `begin` e `step`; `end` é opcional, se for `None`, a série será ilimitada.
- ② Obtém o tipo somando `self.begin` e `self.step`. Por exemplo, se um for `int` e o outro `float`, o `result_type` será `float`.
- ③ Esta linha cria um `result` com o mesmo valor numérico de `self.begin`, mas coagido para o tipo das somas subsequentes.^[8]
- ④ Para melhorar a legibilidade, o sinalizador `forever` será `True` se o atributo `self.end` for `None`, resultando em uma série ilimitada.
- ⑤ Este laço roda `forever` (para sempre) ou até o resultado ser igual ou maior que `self.end`. Quando este laço termina, a função retorna.
- ⑥ O `result` atual é produzido.
- ⑦ O próximo resultado em potencial é calculado. Ele pode nunca ser produzido, se o laço `while` terminar.

Na última linha do Exemplo 12, em vez de somar `self.step` ao `result` anterior a cada volta do laço, optei por ignorar o `result` existente: cada novo `result` é criado somando `self.begin` a `self.step` multiplicado por `index`. Isso evita o efeito cumulativo de erros após a adição sucessiva de números de ponto flutuante. Alguns experimentos simples revelam a diferença:

```
>>> 100 * 1.1
110.00000000000001
>>> sum(1.1 for _ in range(100))
109.99999999999982
>>> 1000 * 1.1
1100.0
>>> sum(1.1 for _ in range(1000))
1100.00000000000086
```

A classe `ArithmeticProgression` do Exemplo 12 é outro exemplo do uso de uma função geradora para implementar o método especial `__iter__`. Agora, se o único objetivo de uma classe é criar um gerador pela implementação de `__iter__`, podemos substituir a classe inteira por uma função geradora. Afinal, uma função geradora é uma fábrica de geradores.

O Exemplo 13 mostra uma função geradora chamada `aritprog_gen`, que realiza a mesma tarefa da `ArithmeticProgression`, com menos código. Se, em vez de chamar `ArithmeticProgression`, você chamar `aritprog_gen`, os testes no Exemplo 11 são todos bem-sucedidos.^[9]

Exemplo 13. a função geradora `aritprog_gen`

```
def aritprog_gen(begin, step, end=None):
    result = type(begin + step)(begin)
    forever = end is None
    index = 0
    while forever or result < end:
        yield result
        index += 1
        result = begin + step * index
```

O Exemplo 13 é elegante, mas lembre-se: há muitos geradores prontos para uso na biblioteca padrão, e a próxima seção vai mostrar uma implementação mais curta, usando o módulo `itertools`.

17.8.1. Progressão aritmética com `itertools`

O módulo `itertools` no Python 3.10 contém 20 funções geradoras, que podem ser combinadas de várias maneiras interessantes.

Por exemplo, a função `itertools.count` devolve um gerador que produz números. Sem argumentos, ele produz uma série de inteiros começando de 0. Mas você pode fornecer os valores opcionais `start` e `step`, para obter um resultado similar ao das nossas funções `aritprog_gen`:

```
>>> import itertools
>>> gen = itertools.count(1, .5)
>>> next(gen)
1
>>> next(gen)
1.5
>>> next(gen)
2.0
>>> next(gen)
2.5
```



`itertools.count` nunca para, então se você chamar `list(count())`, Python vai tentar criar uma `list` que preencheria todos os chips de memória já fabricados. Na prática, sua máquina vai ficar muito mal-humorada bem antes da chamada fracassar.

Por outro lado, temos também a função `itertools.takewhile`: ela devolve um gerador que consome outro gerador e para quando um predicado resulta falso. Então podemos combinar os dois e escrever o seguinte:

```
>>> gen = itertools.takewhile(lambda n: n < 3, itertools.count(1, .5))
>>> list(gen)
[1, 1.5, 2.0, 2.5]
```

Aproveitando `takewhile` e `count`, o Exemplo 14 fica mais conciso.

Exemplo 14. `aritprog_v3.py`: funciona como as funções `aritprog_gen` anteriores

```
import itertools

def aritprog_gen(begin, step, end=None):
    first = type(begin + step)(begin)
    ap_gen = itertools.count(first, step)
    if end is None:
        return ap_gen
    return itertools.takewhile(lambda n: n < end, ap_gen)
```

Observe que `aritprog_gen` no Exemplo 14 não é uma função geradora: não há um `yield` em seu corpo. Mas ela devolve um gerador, exatamente como faz uma função geradora.

A lição do Exemplo 14 é: ao implementar geradores, veja o que já está disponível na biblioteca padrão, caso contrário você tem uma boa chance de reinventar a roda. Por isso a próxima seção trata de várias funções geradoras prontas para usar.

17.9. Funções geradoras na biblioteca padrão

A biblioteca padrão oferece muitos geradores, desde objetos de arquivo de texto fornecendo iteração linha por linha até a incrível função `os.walk` [fpy.li/17-12], que produz nomes de arquivos percorrendo uma árvore de diretórios, reduzindo uma busca recursiva no sistema de arquivos a um simples laço `for`.

A função geradora `os.walk` é impressionante, mas nesta seção quero me concentrar em funções genéricas que recebem iteráveis arbitrários como argumento e devolvem geradores que produzem itens selecionados, agregados ou reordenados.

17.9.1. Funções geradoras de seleção

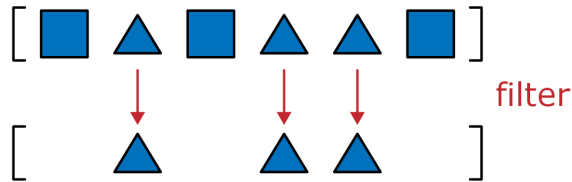


Figura 2. Funções geradoras de seleção criam geradores que selecionam itens conforme algum critério.

O primeiro grupo são funções geradoras de seleção: elas devolvem um gerador que produz um subconjunto dos itens do iterável de entrada, sem mudar os itens em si.

O exemplo mais conhecido é a função embutida `filter`:

`filter(predicate, it)`

Aplica `predicate` para cada item de `iterable`, produzindo o item se `predicate(item)` resulta verdadeiro. Quando `predicate` é `None`, apenas itens verdadeiros serão produzidos.

Como `filter`, a maioria das funções listadas na Tabela 1 recebe uma `predicate` (uma função booleana de um argumento) que será aplicada a cada item no iterável de entrada, para determinar se aquele item será incluído na saída. A exceção é `itertools.compress`, que consome dois iteráveis, sendo que o segundo iterável fornece valores para decidir quais itens do primeiro iterável serão produzidos.

A seção de console no Exemplo 15 demonstra o uso dos geradores de seleção.

Exemplo 15. Exemplos de funções geradoras de seleção

```
>>> def vowel(c):
...     return c.lower() in 'aeiou'
...
>>> list(filter(vowel, 'Aardvark'))
['A', 'a', 'a']
>>> import itertools
>>> list(itertools.filterfalse(vowel, 'Aardvark'))
['r', 'd', 'v', 'r', 'k']
>>> list(itertools.dropwhile(vowel, 'Aardvark'))
```

```
['r', 'd', 'v', 'a', 'r', 'k']
>>> list(itertools.takewhile(vowel, 'Aardvark'))
['A', 'a']
>>> list(itertools.compress('Aardvark', (1, 0, 1, 1, 0, 1)))
['A', 'r', 'd', 'a']
>>> list(itertools.islice('Aardvark', 4))
['A', 'a', 'r', 'd']
>>> list(itertools.islice('Aardvark', 4, 7))
['v', 'a', 'r']
>>> list(itertools.islice('Aardvark', 1, 7, 2))
['a', 'd', 'a']
```

Tabela 1. *itertools: funções geradoras de seleção*

Função	Descrição
<code>compress(it, selector_it)</code>	Consome dois iteráveis em paralelo; produz itens de <code>it</code> sempre que o item correspondente em <code>selector_it</code> é verdadeiro
<code>dropwhile(predicate, it)</code>	Consome <code>it</code> , pulando itens enquanto <code>predicate</code> resultar verdadeiro, e daí produz todos os elementos restantes (nenhuma verificação adicional é realizada)
<code>filterfalse(predicate, it)</code>	Como a <code>filter</code> , mas invertendo a lógica: produz itens quando <code>predicate</code> resulta falso
<code>islice(it, stop)</code> <code>islice(it, start, stop, step=1)</code>	Produz itens de uma fatia de <code>it</code> , similar a <code>s[:stop]</code> ou <code>s[start:stop:step]</code> , mas <code>it</code> é qualquer iterável e a operação é preguiçosa
<code>takewhile(predicate, it)</code>	Produz itens enquanto <code>predicate</code> resultar verdadeiro, e daí para (nenhuma verificação adicional é realizada).

17.9.2. Funções geradoras de transformação

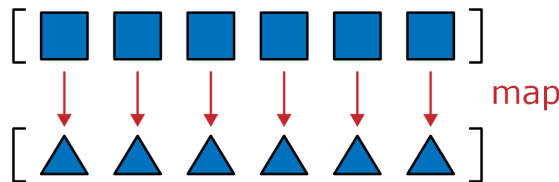


Figura 3. Funções geradoras de transformação criam geradores que aplicam uma função aos itens da entrada, produzindo itens transformados.

O grupo seguinte contém os geradores de transformação. Estes geradores produzem itens computados a partir de cada item individual no iterável de entrada—ou iteráveis, nos casos de `map` e `starmap`. Se a entrada vier de mais de um iterável, a saída para assim que o primeiro iterável de entrada for esgotado.

Tabela 2. Funções geradoras de transformação (embutidas)

Função	Descrição
<code>enumerate(iterable, start=0)</code>	Produz tuplas de dois itens na forma (index, item), onde index é contado a partir de start, e item é obtido do iterable
<code>map(func, it1, [it2, ..., itN])</code>	Aplica func a cada item de it, produzindo o resultado; se forem fornecidos N iteráveis, func deve aceitar N argumentos, e os iteráveis serão consumidos em paralelo

O módulo `itertools` oferece mais duas funções geradoras de transformação:

Tabela 3. `itertools`: funções geradoras de transformação

Função	Descrição
<code>starmap(func, it)</code>	Aplica func a cada item de it, produzindo o resultado; o iterável de entrada deve produzir itens iteráveis iit, e func é aplicada na forma <code>func(*iit)</code>
<code>accumulate(it, [func])</code>	Produz somas cumulativas; se func for fornecida, produz o resultado da aplicação de func ao primeiro par de itens, depois ao primeiro resultado e ao próximo item, etc.

O Exemplo 16 demonstra alguns usos de `itertools.accumulate`.

Exemplo 16. Exemplos das funções geradoras de `itertools.accumulate`

```
>>> import itertools
>>> sample = [5, 4, 2, 8, 7, 6, 3, 0, 9, 1]
>>> list(itertools.accumulate(sample)) ①
[5, 9, 11, 19, 26, 32, 35, 35, 44, 45]
>>> list(itertools.accumulate(sample, min)) ②
[5, 4, 2, 2, 2, 2, 2, 0, 0, 0]
>>> list(itertools.accumulate(sample, max)) ③
[5, 5, 5, 8, 8, 8, 8, 8, 9, 9]
>>> import operator
>>> list(itertools.accumulate(sample, operator.mul)) ④
[5, 20, 40, 320, 2240, 13440, 40320, 0, 0, 0]
>>> list(itertools.accumulate(range(1, 11), operator.mul))
[1, 2, 6, 24, 120, 720, 5040, 40320, 362880, 3628800] ⑤
```

- ① Soma acumulada.
- ② Mínimo corrente.
- ③ Máximo corrente.
- ④ Produto acumulado.
- ⑤ Fatoriais de 1! a 10!.

As demais funções de transformação são demonstradas no Exemplo 17.

Exemplo 17. Exemplos de funções geradoras de transformação

```
>>> list(enumerate('albatroz', 1)) ①
[(1, 'a'), (2, 'l'), (3, 'b'), (4, 'a'), (5, 't'), (6, 'r'), (7, 'o'), (8, 'z')]
>>> import operator
>>> list(map(operator.mul, range(11), range(11))) ②
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
>>> list(map(operator.mul, range(11), [2, 4, 8])) ③
[0, 4, 16]
>>> list(map(lambda a, b: (a, b), range(11), [2, 4, 8])) ④
[(0, 2), (1, 4), (2, 8)]
>>> import itertools
```

```
>>> list(itertools.starmap(operator.mul, enumerate('albatroz', 1))) ⑤
['a', 'll', 'bbb', 'aaaa', 'ttttt', 'rrrrrr', 'ooooooo', 'zzzzzzzz']
>>> sample = [5, 4, 2, 8, 7, 6, 3, 0, 9, 1]
>>> list(itertools.starmap(lambda a, b: b / a,
...     enumerate(itertools.accumulate(sample), 1))) ⑥
[5.0, 4.5, 3.6666666666666665, 4.75, 5.2, 5.333333333333333,
5.0, 4.375, 4.888888888888889, 4.5]
```

- ① Número de letras na palavra, começando por 1.
- ② Os quadrados dos inteiros de 0 a 10.
- ③ Multiplicando os números de dois iteráveis em paralelo; os resultados cessam quando o iterável menor termina.
- ④ Isso é o que faz a função embutida `zip`.
- ⑤ Repete cada letra na palavra de acordo com a posição da letra na palavra, começando por 1.
- ⑥ Média corrente.

17.9.3. Funções geradoras de mesclagem

A seguir temos o grupo de geradores de mesclagem (*merge*). Elas mesclam os itens de vários iteráveis de entrada.

O exemplo mais conhecido é a função embutida `zip`:

`zip(it1, ..., itN, strict=False)`

Produz tuplas de N elementos criadas a partir de itens obtidos dos iteráveis em paralelo, terminando silenciosamente quando o menor iterável é esgotado, a menos que `strict=True` for passado (a partir do Python 3.10). Quando `strict=True`, um `ValueError` é gerado se algum iterável esgotar antes dos outros. O default é `False`, para manter a compatibilidade retroativa.

Na Tabela 4, vale notar que `chain` e `chain.from_iterable` consomem os iteráveis de entrada um após o outro, enquanto `product`, e `zip_longest` consomem os iteráveis de entrada em paralelo, como faz a função `zip`.

Tabela 4. itertools: funções geradoras que mesclam os iteráveis de entrada

Função	Descrição
<code>chain(it1, ..., itN)</code>	Produz todos os itens de it1, a seguir de it2, etc., continuamente.
<code>chain.from_iterable(it)</code>	Produz todos os itens de cada iterável produzido por it, um após o outro, continuamente; it é um iterável cujos itens também são iteráveis, uma lista de tuplas, por exemplo
<code>product(it1, ..., itN, repeat=1)</code>	Produto cartesiano: produz tuplas de N elementos criadas combinando itens de cada iterável de entrada, como laços for aninhados produziriam; repeat permite que os iteráveis de entrada sejam consumidos mais de uma vez
<code>zip_longest(it1, ..., itN, fillvalue=None)</code>	Produz tuplas de N elementos criadas a partir de itens obtidos dos iteráveis em paralelo, terminando apenas quando o último iterável for esgotado, preenchendo os itens ausentes com o fillvalue

O Exemplo 18 demonstra o uso destas funções geradoras. Lembre-se de que o nome zip refere-se ao zíper (ou fecho-éclair) e não tem relação com a compressão de dados.

Exemplo 18. Exemplos de funções geradoras de fusão

```

>>> list(itertools.chain('ABC', range(2))) ①
['A', 'B', 'C', 0, 1]
>>> list(itertools.chain(enumerate('ABC'))) ②
[(0, 'A'), (1, 'B'), (2, 'C')]
>>> list(itertools.chain.from_iterable(enumerate('ABC'))) ③
[0, 'A', 1, 'B', 2, 'C']
>>> list(zip('ABC', range(5), [10, 20, 30, 40])) ④
[('A', 0, 10), ('B', 1, 20), ('C', 2, 30)]
>>> list(itertools.zip_longest('ABC', range(5))) ⑤
[('A', 0), ('B', 1), ('C', 2), (None, 3), (None, 4)]
>>> list(itertools.zip_longest('ABC', range(5), fillvalue='?')) ⑥
[('A', 0), ('B', 1), ('C', 2), ('?', 3), ('?', 4)]

```

- ① `chain` é normalmente invocada com dois ou mais iteráveis.
- ② `chain` não faz nada de útil se invocada com um único iterável.
- ③ Mas `chain.from_iterable` pega cada item do iterável e os encadeia em sequência, desde que cada item seja também iterável.
- ④ Qualquer número de iteráveis pode ser consumido em paralelo por `zip`, mas o gerador sempre para assim que o primeiro iterável acaba. No Python ≥ 3.10 , se o argumento `strict=True` for passado e um iterável terminar antes dos outros, um `ValueError` é gerado.
- ⑤ `itertools.zip_longest` funciona como `zip`, exceto por consumir todos os iteráveis de entrada até o fim, preenchendo as tuplas de saída com `None` nas posições correspondentes aos iteráveis esgotados antes do iterável mais longo.
- ⑥ O argumento nomeado `fillvalue` especifica um valor de preenchimento customizado.

O gerador `itertools.product` é uma forma preguiçosa de gerar produtos cartesianos. Na «Seção 2.3.3» [fpy.li/ce] (vol.1), criamos produtos cartesianos de modo ávido usando compreensões de lista com mais de uma instrução `for`. Expressões geradoras com várias instruções `for` também podem ser usadas para produzir produtos cartesianos de forma preguiçosa. O Exemplo 19 demonstra `itertools.product`.

Exemplo 19. Exemplo da função geradora `itertools.product`

```
>>> list(itertools.product('ABC', range(2))) ①
[('A', 0), ('A', 1), ('B', 0), ('B', 1), ('C', 0), ('C', 1)]
>>> suits = 'spades hearts diamonds clubs'.split()
>>> list(itertools.product('AK', suits)) ②
[('A', 'spades'), ('A', 'hearts'), ('A', 'diamonds'), ('A', 'clubs'),
 ('K', 'spades'), ('K', 'hearts'), ('K', 'diamonds'), ('K', 'clubs')]
>>> list(itertools.product('ABC')) ③
[('A',), ('B',), ('C',)]
>>> list(itertools.product('ABC', repeat=2)) ④
[('A', 'A'), ('A', 'B'), ('A', 'C'), ('B', 'A'), ('B', 'B'),
 ('B', 'C'), ('C', 'A'), ('C', 'B'), ('C', 'C')]
>>> list(itertools.product(range(2), repeat=3))
[(0, 0, 0), (0, 0, 1), (0, 1, 0), (0, 1, 1), (1, 0, 0),
 (1, 0, 1), (1, 1, 0), (1, 1, 1)]
>>> rows = itertools.product('AB', range(2), repeat=2)
```

```
>>> for row in rows: print(row)
```

```
...  
( 'A', 0, 'A', 0)  
( 'A', 0, 'A', 1)  
( 'A', 0, 'B', 0)  
( 'A', 0, 'B', 1)  
( 'A', 1, 'A', 0)  
( 'A', 1, 'A', 1)  
( 'A', 1, 'B', 0)  
( 'A', 1, 'B', 1)  
( 'B', 0, 'A', 0)  
( 'B', 0, 'A', 1)  
( 'B', 0, 'B', 0)  
( 'B', 0, 'B', 1)  
( 'B', 1, 'A', 0)  
( 'B', 1, 'A', 1)  
( 'B', 1, 'B', 0)  
( 'B', 1, 'B', 1)
```

- ① O produto cartesiano de uma str com três caracteres e um range com dois inteiros produz seis tuplas (porque $3 * 2$ é 6).
- ② O produto de duas cartas altas ('AK') e quatro naipes é uma série de oito tuplas.
- ③ Dado um único iterável, product produz uma série de tuplas de um elemento—muito pouco útil.
- ④ O argumento nomeado repeat=N diz à função para consumir cada iterável de entrada N vezes.

Outras funções geradoras expandem a entrada, produzindo mais de um valor por item de entrada. Elas estão listadas na Tabela 5.

Tabela 5. *itertools: funções geradoras que expandem cada item de entrada em múltiplos itens de saída*

Função	Descrição
<code>combinations(it, out_len)</code>	Produz combinações com <code>out_len</code> itens a partir dos itens produzidos por <code>it</code>
<code>combinations_with_replacement(it, out_len)</code>	Produz combinações com <code>out_len</code> itens a partir dos itens produzidos por <code>it</code> , incluindo combinações com itens repetidos
<code>count(start=0, step=1)</code>	Produz números começando em <code>start</code> , somando <code>step</code> para obter o número seguinte, indefinidamente
<code>cycle(it)</code>	Produz itens de <code>it</code> , armazenando uma cópia de cada, e então produz a sequência inteira repetida, indefinidamente
<code>pairwise(it)</code>	Produz pares sobrepostos sucessivos, obtidos do iterável de entrada (novidade do Python 3.10)
<code>permutations(it, out_len=None)</code>	Produz permutações de <code>out_len</code> itens a partir dos itens produzidos por <code>it</code> ; por default, <code>out_len</code> é <code>len(list(it))</code>
<code>repeat(item, [times])</code>	Produz um item repetidamente, indefinidamente, a menos que um número de <code>times</code> (vezes) seja especificado

As funções `count` e `repeat` de `itertools` devolvem geradores que conjuram itens do nada: elas não consomem itens de um iterável de entrada. Já vimos `itertools.count` na Seção 17.8.1.

O gerador `cycle` faz uma cópia do iterável de entrada e produz seus itens repetidamente. O Exemplo 20 ilustra o uso de `count`, `cycle`, `pairwise` e `repeat`.

Exemplo 20. `count`, `cycle`, `pairwise`, e `repeat`

```
>>> ct = itertools.count() ①
>>> next(ct) ②
0
```

```

>>> next(ct), next(ct), next(ct) ③
(1, 2, 3)
>>> list(itertools.islice(itertools.count(1, .3), 3)) ④
[1, 1.3, 1.6]
>>> cy = itertools.cycle('ABC') ⑤
>>> next(cy)
'A'
>>> list(itertools.islice(cy, 7)) ⑥
['B', 'C', 'A', 'B', 'C', 'A', 'B']
>>> list(itertools.pairwise(range(7))) ⑦
[(0, 1), (1, 2), (2, 3), (3, 4), (4, 5), (5, 6)]
>>> rp = itertools.repeat(7) ⑧
>>> next(rp), next(rp)
(7, 7)
>>> list(itertools.repeat(8, 4)) ⑨
[8, 8, 8, 8]
>>> list(map(operator.mul, range(11), itertools.repeat(5))) ⑩
[0, 5, 10, 15, 20, 25, 30, 35, 40, 45, 50]

```

- ① Cria `ct`, um gerador `count`.
- ② Obtém o primeiro item de `ct`.
- ③ Não posso criar uma `list` a partir de `ct`, pois `ct` nunca para. Então pego os próximos três itens.
- ④ Posso criar uma `list` de um gerador `count` se ele for limitado por `islice` ou `takewhile`.
- ⑤ Cria um gerador `cycle` a partir de `'ABC'`, e obtém seu primeiro item, `'A'`.
- ⑥ Uma `list` só pode ser criada se limitada por `islice`; os próximos sete itens são obtidos aqui.
- ⑦ Para cada item na entrada, `pairwise` produz uma tupla de dois elementos com aquele item e o próximo—se existir um próximo item. Disponível no Python ≥ 3.10.
- ⑧ Cria um gerador `repeat` que produzirá o número 7 para sempre.
- ⑨ Um gerador `repeat` pode ser limitado passando o argumento `times`: aqui o número 8 será produzido 4 vezes.
- ⑩ Um uso comum de `repeat`: fornecer um argumento fixo em `map`; aqui ele fornece o multiplicador 5.

As funções geradoras `combinations`, `combinations_with_replacement` e `permutations` —juntamente com `product`—são chamadas *geradoras combinatórias* na «documentação do `itertools`» [fpy.li/9c]. Também há uma relação muito próxima entre `itertools.product` e o restante das funções *combinatórias*, como mostra o Exemplo 21.

Exemplo 21. Funções geradoras combinatórias produzem múltiplos valores para cada item de entrada

```
>>> list(itertools.combinations('ABC', 2)) ①
[('A', 'B'), ('A', 'C'), ('B', 'C')]
>>> list(itertools.combinations_with_replacement('ABC', 2)) ②
[('A', 'A'), ('A', 'B'), ('A', 'C'), ('B', 'B'), ('B', 'C'), ('C', 'C')]
>>> list(itertools.permutations('ABC', 2)) ③
[('A', 'B'), ('A', 'C'), ('B', 'A'), ('B', 'C'), ('C', 'A'), ('C', 'B')]
>>> list(itertools.product('ABC', repeat=2)) ④
[('A', 'A'), ('A', 'B'), ('A', 'C'), ('B', 'A'), ('B', 'B'), ('B', 'C'),
 ('C', 'A'), ('C', 'B'), ('C', 'C')]
```

- ① Todas as combinações com `len()==2` a partir dos itens em `'ABC'`; a ordem dos itens nas tuplas geradas é irrelevante (elas poderiam ser conjuntos).
- ② Todas as combinações com `len()==2` a partir dos itens em `'ABC'`, incluindo combinações com itens repetidos.
- ③ Todas as permutações com `len()==2` a partir dos itens em `'ABC'`; a ordem dos itens nas tuplas geradas é relevante.
- ④ Produto cartesiano de `'ABC'` e `'ABC'` (esse é o efeito de `repeat=2`).

17.9.4. Funções geradoras de rearranjo

As últimas funções geradoras que vamos examinar nessa seção produzem todos os itens dos iteráveis de entrada, mas rearranjados de alguma forma. Aqui estão duas funções que devolvem múltiplos geradores: `itertools.groupby` e `itertools.tee`.

A função embutida `reversed` é o único gerador tratado neste capítulo que não aceita qualquer iterável como entrada, apenas sequências. Faz sentido: como `reversed` produzirá os itens do último para o primeiro, só funciona com uma sequência porque seu tamanho é conhecido, então é possível acessar o último

item diretamente. Ao produzir cada item sob demanda, `reversed` evita o custo de criar uma cópia invertida da sequência.

Tabela 6. Funções geradoras de rearranjo

Módulo	Função	Descrição
<code>itertools</code>	<code>groupby(it, key=None)</code>	Produz tuplas de 2 elementos na forma (key, group), onde key é o critério de agrupamento e group é um gerador que produz os itens no grupo
(embutida)	<code>reversed(seq)</code>	Produz os itens de seq na ordem inversa, do último para o primeiro; seq deve ser uma sequência ou implementar o método especial <code>__reversed__</code>
<code>itertools</code>	<code>tee(it, n=2)</code>	Produz uma tupla de N geradores, cada um produzindo os itens do iterável de entrada de forma independente

O Exemplo 22 demonstra o uso de `itertools.groupby` e da função embutida `reversed`. Observe que `itertools.groupby` assume que o iterável de entrada está ordenado pelo critério de agrupamento, ou que pelo menos os itens estejam agrupados por aquele critério—mesmo que não estejam completamente ordenados.

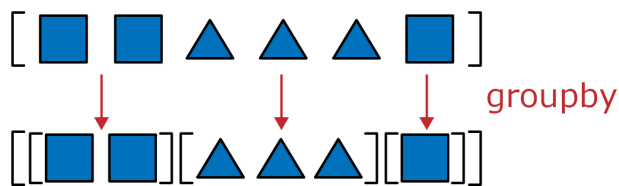


Figura 4. A função geradora `groupby` produz vários geradores, agrupando os itens da entrada segundo algum critério.

O revisor técnico Miroslav Šedivý sugeriu esse caso de uso: você pode ordenar objetos `datetime` em ordem cronológica, e então `groupby` por dia da semana, para obter o grupo com os dados de segunda-feira, seguidos pelos dados de terça, etc., e então da segunda (da semana seguinte) novamente, e assim por diante.

Exemplo 22. itertools.groupby

```
>>> list(itertools.groupby('LLLLAAGGG')) ①
[('L', <itertools._grouper object at 0x102227cc0>),
 ('A', <itertools._grouper object at 0x102227b38>),
 ('G', <itertools._grouper object at 0x102227b70>)]
>>> for char, group in itertools.groupby('LLLLAAAGG'): ②
...     print(char, '->', list(group))
...
L -> ['L', 'L', 'L', 'L']
A -> ['A', 'A',]
G -> ['G', 'G', 'G']
>>> animals = ['duck', 'eagle', 'rat', 'giraffe', 'bear',
...            'bat', 'dolphin', 'shark', 'lion']
>>> animals.sort(key=len) ③
>>> animals
['rat', 'bat', 'duck', 'bear', 'lion', 'eagle', 'shark',
 'giraffe', 'dolphin']
>>> for length, group in itertools.groupby(animals, len): ④
...     print(length, '->', list(group))
...
3 -> ['rat', 'bat']
4 -> ['duck', 'bear', 'lion']
5 -> ['eagle', 'shark']
7 -> ['giraffe', 'dolphin']
>>> for length, group in itertools.groupby(reversed(animals), len): ⑤
...     print(length, '->', list(group))
...
7 -> ['dolphin', 'giraffe']
5 -> ['shark', 'eagle']
4 -> ['lion', 'bear', 'duck']
3 -> ['bat', 'rat']
>>>
```

- ① groupby produz tuplas de (key, group_generator).
- ② Tratar geradores groupby envolve iteração aninhada: neste caso, o laço for externo e o construtor de list interno.
- ③ Ordena animals pelo tamanho de cada string.
- ④ Novamente, um laço sobre o par key e group, para exibir key e expandir o group em uma list.

⑤ Aqui o gerador `reverse` itera sobre `animals` da direita para a esquerda.

A última das funções geradoras nesse grupo é `iterator.tee`, que apresenta um comportamento singular: ela produz múltiplos geradores a partir de um único iterável de entrada, cada um deles produzindo todos os itens daquele iterável. Estes geradores podem ser consumidos de forma independente, como mostra o Exemplo 23.

Exemplo 23. `itertools.tee` produz múltiplos geradores, cada um produzindo todos os itens do gerador de entrada

```
>>> list(itertools.tee('ABC'))
[<itertools._tee object at 0x10222abc8>, <itertools._tee object at
0x10222ac08>]
>>> g1, g2 = itertools.tee('ABC')
>>> next(g1)
'A'
>>> next(g2)
'A'
>>> next(g2)
'B'
>>> list(g1)
['B', 'C']
>>> list(g2)
['C']
>>> list(zip(*itertools.tee('ABC')))
[('A', 'A'), ('B', 'B'), ('C', 'C')]
```

Observe que vários exemplos nesta seção usam combinações de funções geradoras. Esta é uma característica poderosa destas funções: como recebem geradores como argumentos e devolvem geradores como resultado, elas podem ser combinadas de muitas formas diferentes.

Vamos agora revisar outro grupo de funções da biblioteca padrão que lidam com iteráveis.

17.10. Funções de redução de iteráveis

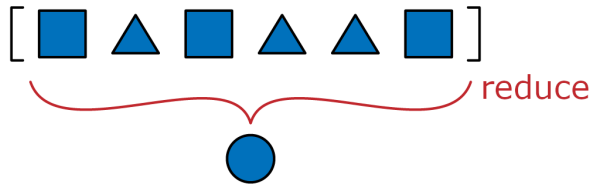


Figura 5. A função `reduce` aplica uma função aos sucessivos itens da entrada, gerando um único resultado acumulado.

Todas as funções na Tabela 7 recebem um iterável e devolvem um resultado único. Elas são conhecidas como funções de "redução", "dobra" (*folding*) ou "acumulação". A função de redução mais flexível é `functools.reduce`:

`functools.reduce(func, it, [initial])`

Devolve o resultado da aplicação de `func` ao primeiro par de itens, depois aplica `func` ao resultado anterior e ao terceiro item, e assim por diante. Se `initial` for passado, esse argumento formará o par inicial com o primeiro item do iterável `it`.

Podemos implementar cada uma das funções embutidas listadas a seguir com `functools.reduce`, mas elas estão embutidas no Python por simplificarem os casos de uso mais comuns de `functools.reduce`. Vimos uma explicação mais aprofundada sobre `functools.reduce` na «Seção 12.7» [fpy.li/59] (vol.2).

Nos casos de `all` e `any`, há uma importante otimização não suportada por `functools.reduce`: `all` e `any` implementam o retorno por curto-circuito—isto é, elas param de consumir o iterador assim que o resultado é determinado. Veja o último teste com `any` no Exemplo 24.

Tabela 7. Funções embutidas que leem iteráveis e devolvem um único valor

Função	Descrição
<code>all(it)</code>	Devolve <code>True</code> se todos os itens em <code>it</code> forem verdadeiros, <code>False</code> em caso contrário; <code>all([])</code> devolve <code>True</code>
<code>any(it)</code>	Devolve <code>True</code> se qualquer item em <code>it</code> for verdadeiro, <code>False</code> em caso contrário; <code>any([])</code> devolve <code>False</code>

Função	Descrição
<code>max(it, [key=,] [default=])</code>	Devolve o valor máximo entre os itens de <code>it</code> ; ^[10] <code>key</code> é uma função de ordenação, como em <code>sorted</code> ; <code>default</code> é devolvido se o iterável estiver vazio
<code>min(it, [key=,] [default=])</code>	Devolve o valor mínimo entre os itens de <code>it</code> . ^[11] <code>key</code> é uma função de ordenação, como em <code>sorted</code> ; <code>default</code> é devolvido se o iterável estiver vazio
<code>sum(it, start=0)</code>	A soma de todos os itens em <code>it</code> , acrescida do valor opcional <code>start</code> (para uma precisão melhor na adição de números de ponto flutuante, use <code>math.fsum</code>)

O Exemplo 24 exemplifica a operação de `all` e de `any`.

Exemplo 24. Resultados de `all` e `any` para algumas sequências

```
>>> all([1, 2, 3])
True
>>> all([1, 0, 3])
False
>>> all([])
True
>>> any([1, 2, 3])
True
>>> any([1, 0, 3])
True
>>> any([0, 0.0])
False
>>> any([])
False
>>> g = (n for n in [0, 0.0, 7, 8])
>>> any(g) ①
True
>>> next(g) ②
8
```

① `any` iterou sobre `g` até `g` produzir 7; neste momento `any` parou e devolveu `True`.

② É por isso que 8 ainda restava.

Outra função embutida que recebe um iterável e devolve outra coisa é `sorted`. Diferente de `reversed`, que é uma função geradora, `sorted` cria e devolve uma nova `list`. Afinal, cada um dos itens no iterável de entrada precisa ser lido para que todos possam ser ordenados, e a ordenação acontece em uma `list`; `sorted` então apenas devolve aquela `list` após terminar seu processamento. Menciono `sorted` aqui porque ela consome um iterável arbitrário.

Claro, `sorted` e as funções de redução só funcionam com iteráveis que terminam. Caso contrário, eles seguirão consumindo itens e nunca devolverão um resultado.



Se você chegou até aqui, já viu o conteúdo mais importante e útil deste capítulo. As seções restantes tratam de recursos avançados de geradores, que a maioria de nós não vê ou precisa com muita frequência, tal como a instrução `yield from` e as corrotinas clássicas.

Há também seções sobre dicas de tipo para iteráveis, iteradores e corrotinas clássicas.

A sintaxe `yield from` fornece uma nova forma de combinar geradores. É nosso próximo assunto.

17.11. Subgeradores com `yield from`

A sintaxe da expressão `yield from` foi introduzida no Python 3.3, para permitir que um gerador delegue tarefas a um subgerador.

Antes da introdução de `yield from`, usávamos um laço `for` quando um gerador precisava produzir valores de outro gerador:

```
>>> def sub_gen():
...     yield 1.1
...     yield 1.2
...
>>> def gen():
...     yield 1
...     for i in sub_gen():
...         yield i
...     yield 2
```

```
...
>>> for x in gen():
...     print(x)
...
1
1.1
1.2
2
```

Podemos obter o mesmo resultado usando `yield from`, como se vê no Exemplo 25.

Exemplo 25. Experimentando `yield from`

```
>>> def sub_gen():
...     yield 1.1
...     yield 1.2
...
>>> def gen():
...     yield 1
...     yield from sub_gen()
...     yield 2
...
>>> for x in gen():
...     print(x)
...
1
1.1
1.2
2
```

No Exemplo 25, o laço `for` é o *código cliente*, `gen` é o *gerador delegante* e `sub_gen` é o *subgerador*. Observe que `yield from` suspende `gen`, e `sub_gen` toma o controle até se esgotar. Os valores produzidos por `sub_gen` passam através de `gen` diretamente para o laço `for` do cliente. Enquanto isso, `gen` está suspenso e não pode ver os valores que passam por ele. `gen` continua apenas quando `sub_gen` termina.

Quando o subgerador contém uma instrução `return` com um valor, aquele valor pode ser capturado pelo gerador delegante, com o uso de `yield from` como parte de uma expressão. Veja a demonstração no Exemplo 26.

Exemplo 26. yield from recebe o valor devolvido pelo subgerador

```
>>> def sub_gen():
...     yield 1.1
...     yield 1.2
...     return 'Done!'
...
>>> def gen():
...     yield 1
...     result = yield from sub_gen()
...     print('<--', result)
...     yield 2
...
>>> for x in gen():
...     print(x)
...
1
1.1
1.2
<-- Done!
2
```

Agora que já vimos o básico sobre `yield from`, vamos estudar alguns exemplos simples mas práticos de sua utilização.

17.11.1. Reinventando `chain`

Vimos na Tabela 4 que `itertools` fornece um gerador `chain`, que produz itens a partir de vários iteráveis, iterando sobre o primeiro, depois sobre o segundo, e assim por diante, até o último. A maioria das funções de `itertools` são escritas em C, incluindo `chain`. Abaixo está uma implementação caseira de `chain`, com laços `for` aninhados, em Python:

```
>>> def chain(*iterables):
...     for it in iterables:
...         for i in it:
...             yield i
...
>>> s = 'ABC'
>>> r = range(3)
```



```
>>> list(chain(s, r))
['A', 'B', 'C', 0, 1, 2]
```

O gerador `chain`, no código acima, está delegando para cada iterável `it`, controlando cada `it` no laço `for` interno. Aquele laço interno pode ser substituído por uma expressão `yield from`, como mostra a seção de console a seguir:

```
>>> def chain(*iterables):
...     for i in iterables:
...         yield from i
...
>>> list(chain(s, t))
['A', 'B', 'C', 0, 1, 2]
```

O uso de `yield from` neste exemplo está correto, e o código é mais legível, mas parece açúcar sintático, com pouco ganho real. Vamos então desenvolver um exemplo mais interessante.

17.11.2. Percorrendo uma árvore

Nessa seção, veremos `yield from` em um script para percorrer uma estrutura de árvore. Vou desenvolvê-lo passo a passo, em *baby steps* (passinhos de bebê).

A estrutura de árvore nesse exemplo é a «hierarquia das exceções» [fpy.li/9d] de Python. Mas o padrão pode ser adaptado para exibir uma árvore de diretórios ou qualquer outra estrutura de árvore.

Começando de `BaseException` no nível zero, a hierarquia de exceções tem cinco níveis de profundidade no Python 3.10. Nosso primeiro passinho será exibir o nível zero.

Dada uma classe raiz, o gerador `tree` no Exemplo 27 produz o nome dessa classe e para.

Exemplo 27. tree/step0/tree.py: produz o nome da classe raiz e para

```
def tree(cls):
    yield cls.__name__

def display(cls):
    for cls_name in tree(cls):
        print(cls_name)

if __name__ == '__main__':
    display(BaseException)
```

A saída do Exemplo 27 tem apenas uma linha:

```
BaseException
```

O próximo pequeno passo nos leva ao nível 1. O gerador `tree` produzirá o nome da classe raiz e os nomes de cada subclasse direta. Os nomes das subclasses são indentados para explicitar a hierarquia. Esta é a saída que queremos:

```
$ python3 tree.py
BaseException
    Exception
    GeneratorExit
    SystemExit
    KeyboardInterrupt
```

O Exemplo 28 produz a saída acima.

Exemplo 28. tree/step1/tree.py: produz o nome da classe raiz e das subclasses diretas

```
def tree(cls):
    yield cls.__name__, 0
    for sub_cls in cls.__subclasses__():
        yield sub_cls.__name__, 1
```

```
def display(cls):
    for cls_name, level in tree(cls):
        indent = ' ' * 4 * level
        print(f'{indent}{cls_name}')

if __name__ == '__main__':
    display(BaseException)
```

- ① Para suportar a saída indentada, produz o nome da classe e seu nível na hierarquia.
- ② Usa o método especial `__subclasses__` para obter uma lista de subclasses.
- ③ Produz o nome da subclasse e o nível (1).
- ④ Cria a string de indentação de 4 espaços vezes o `level`. No nível zero, isso será uma string vazia.

No Exemplo 29, refatorei `tree` para separar o caso especial da classe raiz, processando as subclasses no gerador `sub_tree`. Em `yield from`, o gerador `tree` é suspenso, e `sub_tree` passa a produzir valores.

Exemplo 29. `tree/step2/tree.py`: `tree` produz o nome da classe raiz, e então delega para `sub_tree`

```
def tree(cls):
    yield cls.__name__, 0
    yield from sub_tree(cls)

def sub_tree(cls):
    for sub_cls in cls.__subclasses__():
        yield sub_cls.__name__, 1

def display(cls):
    for cls_name, level in tree(cls):
        indent = ' ' * 4 * level
        print(f'{indent}{cls_name}')
```

```
if __name__ == '__main__':  
    display(BaseException)
```

- ① Delega para `sub_tree`, para produzir os nomes das subclasses.
- ② Produz o nome de cada subclasse e o nível (1). Por causa do `yield from sub_tree(cls)` dentro de `tree`, esses valores escapam completamente ao gerador `tree` ...
- ③ ... e são recebidos aqui diretamente.

Seguindo o método de *baby steps*, apresento o código mais simples que consigo imaginar para chegar ao nível 2. Para percorrer uma árvore primeiro em profundidade (*depth-first*) [fpy.li/9e], após produzir cada nó do nível 1, quero produzir os filhotes daquele nó no nível 2 antes de voltar ao nível 1. Um laço `for` aninhado cuida disso, como no Exemplo 30.

Exemplo 30. tree/step3/tree.py: sub_tree percorre os níveis 1 e 2, primeiro em profundidade

```
def tree(cls):  
    yield cls.__name__, 0  
    yield from sub_tree(cls)  
  
def sub_tree(cls):  
    for sub_cls in cls.__subclasses__():  
        yield sub_cls.__name__, 1  
        for sub_sub_cls in sub_cls.__subclasses__():  
            yield sub_sub_cls.__name__, 2  
  
def display(cls):  
    for cls_name, level in tree(cls):  
        indent = ' ' * 4 * level  
        print(f'{indent}{cls_name}')  
  
if __name__ == '__main__':  
    display(BaseException)
```

Este é o resultado da execução de *step3/tree.py*, do Exemplo 30:

```
$ python3 tree.py
BaseException
  Exception
    TypeError
    StopAsyncIteration
    StopIteration
    ImportError
    OSError
    EOFError
    RuntimeError
    NameError
    AttributeError
    SyntaxError
    LookupError
    ValueError
    AssertionError
    ArithmeticError
    SystemError
    ReferenceError
    MemoryError
    BufferError
    Warning
  GeneratorExit
  SystemExit
  KeyboardInterrupt
```

Você pode já ter percebido para onde isso segue, mas vou insistir mais uma vez nos pequenos passos: vamos atingir o nível 3, acrescentando ainda outro laço *for* aninhado.

Não há novidades no resto do programa, então o Exemplo 31 mostra apenas o gerador *sub_tree*.

Exemplo 31. O gerador sub_tree de tree/step4/tree.py

```
def sub_tree(cls):
    for sub_cls in cls.__subclasses__():
        yield sub_cls.__name__, 1
        for sub_sub_cls in sub_cls.__subclasses__():
            yield sub_sub_cls.__name__, 2
            for sub_sub_sub_cls in sub_sub_cls.__subclasses__():
                yield sub_sub_sub_cls.__name__, 3
```

Há um padrão claro no Exemplo 31. Entramos em um laço for para obter as subclasses do nível N. A cada volta do laço, produzimos uma subclasse do nível N, e então iniciamos outro laço for para visitar o nível N+1.

Na Seção 17.11.1, vimos como é possível substituir um laço for aninhado controlando um gerador com yield from sobre o mesmo gerador. Podemos aplicar aquela ideia aqui, se fizermos sub_tree aceitar um parâmetro level, usando yield from recursivamente e passando a subclasse atual como nova classe raiz com o número do nível seguinte. Veja o Exemplo 32.

Exemplo 32. tree/step5/tree.py: a sub_tree recursiva vai tão longe quanto a memória permitir

```
def tree(cls):
    yield cls.__name__, 0
    yield from sub_tree(cls, 1)

def sub_tree(cls, level):
    for sub_cls in cls.__subclasses__():
        yield sub_cls.__name__, level
        yield from sub_tree(sub_cls, level+1)

def display(cls):
    for cls_name, level in tree(cls):
        indent = ' ' * 4 * level
        print(f'{indent}{cls_name}')

if __name__ == '__main__':
    display(BaseException)
```

O Exemplo 32 pode percorrer árvores de qualquer profundidade, limitado apenas pelo limite de recursão de Python. O limite default permite 1.000 funções pendentes.

Qualquer bom tutorial sobre recursão enfatizará a importância de ter um caso base, para evitar uma recursão infinita. Um caso base é um ramo condicional que retorna sem fazer uma chamada recursiva. O caso base é frequentemente implementado com uma instrução `if`. No Exemplo 32, `sub_tree` não tem um `if`, mas há uma condicional implícita no laço `for`: se `cls.subclasses()` devolver uma lista vazia, o corpo do laço não é executado, e assim a chamada recursiva não ocorre. O caso base ocorre quando a classe `cls` não tem subclasses. Nesse caso, `sub_tree` não produz nada, apenas retorna.

O Exemplo 32 funciona como planejado, mas podemos fazê-lo mais conciso recordando o padrão que observamos quando alcançamos o nível 3 (no Exemplo 31): produzimos uma subclasse de nível `N`, e então iniciamos um laço `for` aninhado para visitar o nível `N+1`. No Exemplo 32, substituímos o laço aninhado por `yield from`. Agora podemos fundir `tree` e `sub_tree` em um único gerador. O Exemplo 33 é o último passo deste exemplo.

Exemplo 33. `tree/step6/tree.py`: chamadas recursivas de `tree` passam um argumento `level` incrementado

```
def tree(cls, level=0):
    yield cls.__name__, level
    for sub_cls in cls.__subclasses__():
        yield from tree(sub_cls, level+1)

def display(cls):
    for cls_name, level in tree(cls):
        indent = ' ' * 4 * level
        print(f'{indent}{cls_name}')

if __name__ == '__main__':
    display(BaseException)
```

No início da Seção 17.11, vimos como `yield from` conecta a subgeradora diretamente ao código cliente, escapando do gerador delegante. Aquela conexão se torna realmente importante quando geradores são usados como corrotinas, e não apenas produzem mas também consomem valores do código cliente, como veremos na Seção 17.13.

Após esse primeiro encontro com `yield from`, vamos olhar as dicas de tipo para iteráveis e iteradores.

17.12. Tipos iteráveis genéricos

A biblioteca padrão de Python contém muitas funções que aceitam argumentos iteráveis. Em seu código, tais funções podem ser anotadas como a função `zip_replace` no Exemplo 15 do «Capítulo 8» [fpy.li/8] (vol.1), usando `collections.abc.Iterable`. Veja o Exemplo 34.

Exemplo 34. `replacer.py` devolve um iterador de tuplas de strings

```
from collections.abc import Iterable

FromTo = tuple[str, str] ①

def zip_replace(text: str, changes: Iterable[FromTo]) -> str: ②
    for from_, to in changes:
        text = text.replace(from_, to)
    return text
```

① Define um apelido (*alias*) de tipo; isso não é obrigatório, mas torna a próxima dica de tipo mais legível. Desde o Python 3.10, a variável `FromTo` deve ter uma dica de tipo `typing.TypeAlias`, para esclarecer a razão para esta atribuição:
`FromTo: TypeAlias = tuple[str, str]`

② Anota `changes` para aceitar um `Iterable` de tuplas `FromTo`.

Tipos `Iterator` não aparecem com a mesma frequência de tipos `Iterable`, mas eles também são simples de escrever. O Exemplo 35 mostra o famoso gerador de Fibonacci, anotado.

Exemplo 35. fibo_gen.py: fibonacci devolve um gerador de inteiros

```
from collections.abc import Iterator

def fibonacci() -> Iterator[int]:
    a, b = 0, 1
    while True:
        yield a
        a, b = b, a + b
```

Observe que o tipo `Iterator` é usado para geradores criados como funções com `yield`, bem como para iteradores escritos "à mão", como classes que implementam `__next__`. Há também o tipo `collections.abc.Generator` (e o decontinuado `typing.Generator` correspondente) que podemos usar para anotar objetos geradores, mas ele é verboso e redundante para geradores usados como iteradores, que não recebem valores via `.send()`.

O Exemplo 36, quando checado com o `Mypy`, revela que o tipo `Iterator` é, na verdade, um caso especial simplificado do tipo `Generator`.

Exemplo 36. itergentype.py: duas formas de anotar iteradores

```
from collections.abc import Iterator
from keyword import kwlist
from typing import TYPE_CHECKING

short_kw = (k for k in kwlist if len(k) < 5) ①

if TYPE_CHECKING:
    reveal_type(short_kw) ②

long_kw: Iterator[str] = (k for k in kwlist if len(k) >= 4) ③

if TYPE_CHECKING: ④
    reveal_type(long_kw)
```

① Uma expressão geradora que produz palavras reservadas de Python com menos de 5 caracteres.

② O `Mypy` infere: `typing.Generator[builtins.str*, None, None]`.^[12]

③ Isso também produz strings, mas acrescentei uma dica de tipo explícita.

④ Tipo revelado: `typing.Iterator[builtins.str]`.

`abc.Iterator[str]` é *consistente-com* `abc.Generator[str, None, None]`, assim o Mypy não reporta erros na checagem de tipos no Exemplo 36.

`Iterator[T]` é um atalho para `Generator[T, None, None]`. Ambas as anotações significam "um gerador que produz itens do tipo `T`, mas não consome ou devolve valores." Geradores capazes de consumir e devolver valores são corrotinas, nosso próximo tópico.

17.13. Corrotinas clássicas



A PEP 342—*Coroutines via Enhanced Generators* [[fpy.li/pep342](https://www.python.org/dev/peps/pep-342/)] (Corrotinas via geradores aprimorados) introduziu o método `.send()` e outros recursos que tornaram possível usar geradores como corrotinas. A PEP 342 usa a palavra "corrotina" (*coroutine*) no mesmo sentido que estou usando aqui. É lamentável que a documentação oficial de Python e da biblioteca padrão agora use uma terminologia inconsistente para se referir a geradores usados como corrotinas, obrigando-me a adotar o qualificador "corrotina clássica", para diferenciá-las das novas "corrotinas nativas".

Após o lançamento de Python 3.5, a tendência é usar "corrotina" como sinônimo de "corrotina nativa". Mas a PEP 342 não está descontinuada, e as corrotinas clássicas ainda funcionam como originalmente projetadas, apesar de não serem mais suportadas pela biblioteca `asyncio`.

Entender as corrotinas clássicas no Python é mais confuso porque elas são, na verdade, geradores usados de uma forma diferente. Vamos então dar um passo atrás e examinar outro recurso de Python que pode ser usado de duas maneiras.

Vimos na «Seção 2.4» [[fpy.li/cf](https://www.python.org/dev/peps/pep-24/)] (vol.1) que é possível usar instâncias de `tuple` como registros ou como sequências imutáveis. Quando usadas como um registro, espera-se que uma tupla tenha um número fixo de itens, e cada item pode ter um tipo diferente. Quando usadas como listas imutáveis, uma tupla pode ter

qualquer tamanho, e espera-se que todos os itens sejam do mesmo tipo. Por isso, há duas formas de anotar tuplas com dicas de tipo:

```
# Um registro de cidade, como nome, país e população:
city: tuple[str, str, int]

# Uma sequência imutável de nomes de domínios:
domains: tuple[str, ...]
```

Algo similar ocorre com geradores. Eles são normalmente usadas como iteradores, mas podem também ser usados como corrotinas. Na verdade, *corrotina* é uma função geradora, criada com a palavra-chave `yield` em seu corpo. E um *objeto corrotina* é um objeto gerador, fisicamente. Apesar de compartilharem a mesma implementação subjacente em C, os casos de uso de geradores e corrotinas em Python são tão diferentes que há duas formas de escrever dicas de tipo para elas:

```
# A variável 'readings' pode ser vinculada a um iterador
# ou a um objeto gerador que produz itens 'float':
readings: Iterator[float]

# A variável 'sim_taxi' pode ser vinculada a uma corrotina
# representando um táxi em uma simulação de eventos discretos.
# Ela produz eventos, recebe um 'float' de data/hora, e devolve
# o número de viagens realizadas durante a simulação:
sim_taxi: Generator[Event, float, int]
```

Para aumentar a confusão, os autores do módulo `typing` decidiram nomear aquele tipo `Generator`, quando ele de fato descreve a API de um objeto gerador projetado para ser usado como uma corrotina, enquanto geradores são mais frequentemente usados como iteradores.

A «documentação do módulo `typing`» [fpy.li/9f] descreve assim os parâmetros de tipo formais de `Generator`:

```
Generator[YieldType, SendType, ReturnType]
```

O `SendType` só é relevante quando o gerador é usado como uma corrotina. Aquele parâmetro é o tipo de `x` na chamada `gen.send(x)`. É um erro invocar `.send()` em um gerador escrito para se comportar como um iterador em vez de uma corrotina. Da mesma forma, `ReturnType` só faz sentido para anotar uma corrotina, pois iteradores não devolvem valores como fazem as funções comuns. A única operação razoável em um gerador usado como um iterador é invocar `next(it)` direta ou indiretamente, via laços `for` e outras formas de iteração. O `YieldType` é o tipo do valor devolvido em uma chamada a `next(it)`.

O tipo `Generator` tem os mesmos parâmetros de tipo de `typing.Coroutine` [fpy.li/typepecoro]:

```
Coroutine[YieldType, SendType, ReturnType]
```

A documentação de `typing.Coroutine` [fpy.li/typepecoro] diz literalmente: "A variância e a ordem das variáveis de tipo correspondem às de `Generator`." Mas `typing.Coroutine` (descontinuada) e `collections.abc.Coroutine` (genérica a partir de Python 3.9) foram projetadas para anotar apenas corrotinas nativas, e não corrotinas clássicas. Se você quiser usar dicas de tipo em corrotinas clássicas, terá que anotá-las como `Generator[YieldType, SendType, ReturnType]`.

David Beazley criou algumas das melhores palestras e algumas das oficinas mais abrangentes sobre corrotinas clássicas. No material de seu curso na PyCon 2009 [fpy.li/17-18] há um slide chamado *Keeping It Straight* ("Cada coisa em seu lugar"), onde se lê:

- *Geradoras produzem dados para iteração*
- *Corrotinas são consumidoras de dados*
- *Para não explodir seu cérebro, não misture os dois conceitos*
- *Corrotinas não têm relação com iteração*
- *Nota: Há uma forma de fazer `yield` produzir um valor em uma corrotina, mas isso não está ligado à iteração.*^[13]

Vamos ver agora como as corrotinas clássicas funcionam.

17.13.1. Exemplo: corrotina para computar uma média móvel

Quando discutimos clausuras no «Capítulo 9» [fpy.li/9] (vol.2), estudamos objetos para computar uma média móvel. Na «Seção 9.6» [fpy.li/c8], o Exemplo 7 mostra uma classe e o Exemplo 8 apresenta uma função de ordem superior devolvendo uma função que preserve em sua clausura as variáveis `total` e `count` entre uma invocação e a próxima. O Exemplo 37 mostra como fazer o mesmo com uma corrotina.^[14]

Exemplo 37. coroaverager.py: corrotina para computar uma média móvel

```
from collections.abc import Generator

def averager() -> Generator[float, float, None]: ①
    total = 0.0
    count = 0
    average = 0.0
    while True: ②
        term = yield average ③
        total += term
        count += 1
        average = total/count
```

- ① Esta função devolve um gerador que produz valores `float`, aceita valores `float` via `.send()`, e não devolve um valor útil ao retornar.^[15]
- ② Este laço infinito significa que a corrotina continuará produzindo médias enquanto o código cliente enviar valores.
- ③ A instrução `yield` aqui suspende a corrotina, produz um resultado para o cliente e—mais tarde—recebe um valor enviado pelo código de invocação para a corrotina, iniciando outra iteração do laço infinito.

Em uma corrotina, `total` e `count` podem ser variáveis locais: não precisamos usar atributos de uma instância ou uma clausura para preservar o contexto enquanto a corrotina está suspensa, esperando pelo próximo `.send()`. Por isso as corrotinas são substitutas atraentes para *callbacks* em programação assíncrona—elas preservam o estado local entre ativações.

O Exemplo 38 executa doctests mostrando a corrotina `averager` em operação.

Exemplo 38. coroaverager.py: doctest para a corrotina de média móvel do Exemplo 37

```
>>> coro_avg = averager() ①
>>> next(coro_avg) ②
0.0
>>> coro_avg.send(10) ③
10.0
>>> coro_avg.send(30)
20.0
>>> coro_avg.send(5)
15.0
```

- ① Cria o objeto corrotina.
- ② Inicializa a corrotina. Isso produz o valor inicial de average: 0.0.
- ③ Agora estamos conversando: cada chamada a `.send()` produz a média atual.

No Exemplo 38, a chamada `next(coro_avg)` faz a corrotina avançar até o `yield`, produzindo o valor inicial de average. Também é possível inicializar a corrotina chamando `coro_avg.send(None)`—na verdade é isso que a função embutida `next()` faz. Mas você não pode enviar qualquer valor diferente de `None`, pois a corrotina só pode aceitar um valor enviado quando está suspensa, em uma linha de `yield`. Invocar `next()` ou `.send(None)` para avançar até o primeiro `yield` é conhecido como *priming the coroutine* (preparando a corrotina).

Após cada ativação, a corrotina é suspensa exatamente na palavra-chave `yield`, e espera que um valor seja enviado. A linha `coro_avg.send(10)` fornece aquele valor, ativando a corrotina. A expressão `yield` se resolve para o valor 10, que é atribuído à variável `term`. O restante do laço atualiza as variáveis `total`, `count`, e `average`. A próxima volta do laço `while` produz `average`, e a corrotina é novamente suspensa na palavra-chave `yield`.

O leitor atento pode estar ansioso para saber como a execução de uma instância de `averager` (por exemplo, `coro_avg`) pode ser encerrada, pois seu corpo é um laço infinito. Em geral, não precisamos encerrar um gerador, pois será coletado como lixo assim que não existirem mais referências válidas para ele. Se for necessário encerrá-la explicitamente, use o método `.close()`, como mostra o Exemplo 39.

Exemplo 39. coroaverager.py: continuando de Exemplo 38

```
>>> coro_avg.send(20) ①
16.25
>>> coro_avg.close() ②
>>> coro_avg.close() ③
>>> coro_avg.send(5) ④
Traceback (most recent call last):
...
StopIteration
```

- ① `coro_avg` é a instância criada no Exemplo 38.
- ② O método `.close()` gera uma exceção `GeneratorExit` na expressão `yield` suspensa. Se não for tratada na função corrotina, a exceção a encerra. `GeneratorExit` é capturada pelo objeto gerador que encapsula a corrotina—por isso não a vemos.
- ③ Invocar `.close()` em uma corrotina previamente encerrada não tem efeito.
- ④ Tentar usar `.send()` em uma corrotina encerrada levanta `StopIteration`.

Além do método `.send()`, a *PEP 342—Coroutines via Enhanced Generators* [fpy.li/pep342] (Corrotinas via geradores aprimorados) também criou uma forma de uma corrotina devolver um valor. A próxima seção mostra como fazer isso.

17.13.2. Devolvendo um valor a partir de uma corrotina

Vamos agora estudar outra corrotina para computar uma média. Esta versão não produzirá resultados parciais. Em vez disso, ela devolve uma tupla com o número de termos e a média. Dividi a listagem em duas partes, no Exemplo 40 e no Exemplo 41.

Exemplo 40. coroaverager2.py: a primeira parte do arquivo

```
from collections.abc import Generator
from typing import Union, NamedTuple

class Result(NamedTuple): ①
    count: int # type: ignore ②
    average: float
```

```
class Sentinel: ③
    def __repr__(self):
        return f'<Sentinel>'

STOP = Sentinel() ④

SendType = Union[float, Sentinel] ⑤
```

- ① A corrotina `averager2` no Exemplo 41 vai devolver uma instância de `Result`.
- ② `Result` é, na verdade, uma subclasse de `tuple`, que tem um método `.count()`, que não preciso neste exemplo. O comentário `# type: ignore` evita que o `Mypy` reclame sobre a redeclaração do atributo `count`.^[16]
- ③ Uma classe para criar um valor sentinela com um `__repr__` legível.
- ④ O valor sentinela que vou usar para fazer a corrotina parar de coletar dados e devolver um resultado.
- ⑤ Vou usar esse apelido de tipo para o segundo parâmetro de tipo devolvido pela corrotina `Generator`, o parâmetro `SendType`.

A definição de `SendType` também funciona no Python 3.10 mas, se não for necessário suportar versões mais antigas, é melhor escrever a anotação assim, após importar `TypeAlias` de `typing`:

```
SendType: TypeAlias = float | Sentinel
```

Usar `|` em vez de `typing.Union` é tão conciso e legível que eu provavelmente não criaria aquele apelido de tipo. Em vez disso, escreveria a assinatura de `averager2` assim:

```
def averager2(verbose:bool=False) -> Generator[None, float|Sentinel, Result]:
```

Vamos agora estudar o código da corrotina em si no Exemplo 41.

Exemplo 41. coroaverager2.py: corrotina que devolve um resultado final

```
def averager2(verbose:bool=False) -> Generator[None, SendType, Result]:  
    ①  
    total = 0.0  
    count = 0  
    average = 0.0  
    while True:  
        term = yield ②  
        if verbose:  
            print('received:', term)  
        if isinstance(term, Sentinel): ③  
            break  
        total += term ④  
        count += 1  
        average = total / count  
    return Result(count, average) ⑤
```

- ① Para essa corrotina, o tipo produzido ('YieldType') é None, porque ela não produz dados. Ela recebe dados do tipo SendType e devolve uma tupla Result quando termina o processamento.
- ② Usar yield assim só faz sentido em corrotinas, pois elas consomem dados. Esta linha produz None, mas recebe um term na próxima invocação de .send(term).
- ③ Se term é um Sentinel, sai do laço. Graças a essa checagem com isinstance...
- ④ ...Mypy me permite somar term a total sem sinalizar um erro (eu não poderia somar um float a um objeto que pode ser um float ou um Sentinel).
- ⑤ Esta linha só será alcançada se um Sentinel for enviado para a corrotina.

Vamos ver agora como podemos usar essa corrotina, começando por um exemplo simples, que não devolve um resultado (no Exemplo 42).

Exemplo 42. coroaverager2.py: doctest mostrando .cancel()

```
>>> coro_avg = averager2()
>>> next(coro_avg)
>>> coro_avg.send(10) ①
>>> coro_avg.send(30)
>>> coro_avg.send(6.5)
>>> coro_avg.close() ②
```

- ① Lembre-se de que `averager2` não produz resultados parciais. Ela produz `None`, que o console de Python omite.
- ② Invocar `.close()` nessa corrotina a faz parar, mas não devolve um resultado, pois a exceção `GeneratorExit` é gerada na linha `yield` da corrotina, então a instrução `return` nunca é alcançada.

Vamos então fazê-la funcionar, no Exemplo 43.

Exemplo 43. coroaverager2.py: doctest mostrando StopIteration com um Result

```
>>> coro_avg = averager2()
>>> next(coro_avg)
>>> coro_avg.send(10)
>>> coro_avg.send(30)
>>> coro_avg.send(6.5)
>>> try:
...     coro_avg.send(Sentinel()) ①
... except StopIteration as exc:
...     result = exc.value ②
...
>>> result ③
Result(count=3, average=15.5)
```

- ① Enviar um valor `Sentinel()` faz a corrotina sair do laço e devolver um `Result`. O objeto gerador que encapsula a corrotina gera então uma `StopIteration`.
- ② A instância de `StopIteration` tem um atributo `value` vinculado ao valor do comando `return` que encerrou a corrotina.
- ③ Acredite se quiser!

Esta ideia de "contrabandear" o valor devolvido para fora de uma corrotina dentro de uma exceção `StopIteration` é um *hack*. Entretanto, este *hack* é parte da *PEP 342—Coroutines via Enhanced Generators* [fpy.li/pep342], e está documentada com a «exceção `StopIteration`» [fpy.li/9m] e na seção *Expressões yield* [fpy.li/9n] do capítulo 6 de *A Referência da Linguagem Python*.

Um gerador delegante pode obter o valor devolvido por uma corrotina diretamente, usando a sintaxe `yield from`, como demonstrado no Exemplo 44.

Exemplo 44. coroaverager2.py: doctest mostrando `StopIteration` com um `Result`

```
>>> def compute():
...     res = yield from averager2(True) ①
...     print('computed:', res) ②
...     return res ③
...
>>> comp = compute() ④
>>> for v in [None, 10, 20, 30, Sentinel()]: ⑤
...     try:
...         comp.send(v) ⑥
...     except StopIteration as exc: ⑦
...         result = exc.value
received: 10
received: 20
received: 30
received: <Sentinel>
computed: Result(count=3, average=20.0)
>>> result ⑧
Result(count=3, average=20.0)
```

- ① `res` vai coletar o valor devolvido por `averager2`; o mecanismo de `yield from` recupera o valor devolvido ao tratar a exceção `StopIteration`, que sinaliza o encerramento da corrotina. Quando `True`, o parâmetro `verbose` faz a corrotina exibir o valor recebido, tornando sua operação visível.
- ② Preste atenção na saída desta linha quando o gerador for acionado.
- ③ Devolve o resultado. Ele também estará encapsulado em `StopIteration`.
- ④ Cria o objeto corrotina delegante.
- ⑤ Este laço vai controlar a corrotina delegante.

- ⑥ O primeiro valor enviado é `None`, para preparar a corrotina; o último é a sentinela, para pará-la.
- ⑦ Captura `StopIteration` para obter o valor devolvido por `compute`.
- ⑧ Após as linhas exibidas por `averager2` e `compute`, recebemos a instância de `Result`.

Mesmo com esses exemplos aqui, que não fazem muita coisa, o código é difícil de entender. Controlar a corrotina com chamadas `.send()` e recuperar os resultados é complicado, exceto com `yield from`—mas só podemos usar essa sintaxe dentro de um gerador/corrotina, que no fim precisa ser controlada por algum código não-trivial, como mostra o Exemplo 44.

Os exemplos anteriores mostram que o uso direto de corrotinas é incômodo e confuso. Acrescente o tratamento de exceções e o método de corrotina `.throw()`, e os exemplos ficam ainda mais complicados. Não vou tratar de `.throw()` nesse livro porque—como `.send()`—ele só é útil para controlar corrotinas "manualmente", e não recomendo fazer isso, a menos que você esteja criando um novo framework baseado em corrotinas do zero .



Se tiver interesse em uma discussão mais profunda sobre corrotinas clássicas—incluindo o método `.throw()`—por favor veja *Classic Coroutines* [fpy.li/oldcoro] no site que acompanha o livro, *fluentpython.com*. Aquele texto inclui pseudo-código similar ao Python detalhando como `yield from` controla geradores e corrotinas, bem como uma pequena simulação de eventos discretos, demonstrando uma forma de concorrência usando corrotinas sem um framework de programação assíncrona.

Na prática, qualquer trabalho produtivo com corrotinas exige o apoio de um framework especializado. É isso que `asyncio` oferecia para corrotinas clássicas lá atrás, no Python 3.3. Com o advento das corrotinas nativas no Python 3.5, os mantenedores de Python estão gradualmente eliminando o suporte a corrotinas clássicas no `asyncio`. Mas os mecanismos subjacentes são muito similares. A sintaxe `async def` torna as corrotinas nativas mais visíveis no código, um grande benefício por si só. Internamente, as corrotinas nativas usam `await` em vez de `yield from` para delegar a outras corrotinas. O Capítulo 21 é todo sobre este assunto.

Vamos agora encerrar o capítulo com uma seção alucinante sobre co-variância e contravariância em dicas de tipo para corrotinas.

17.13.3. Dicas de tipo genéricas para corrotinas clássicas

Na «Seção 15.7.4.3» [fpy.li/cg] (vol.2), mencionei `typing.Generator` como um dos poucos tipos da biblioteca padrão com um parâmetro de tipo contravariante. Agora que estudamos as corrotinas clássicas, estamos prontos para entender este tipo genérico.

É assim que `typing.Generator` era declarado [fpy.li/17-25] no módulo `typing.py` de Python 3.6:^[17]

```
T_co = TypeVar('T_co', covariant=True)
V_co = TypeVar('V_co', covariant=True)
T_contra = TypeVar('T_contra', contravariant=True)

# muitas linhas omitidas

class Generator(Iterator[T_co], Generic[T_co, T_contra, V_co],
                extra=_G_base):
```

Esta declaração de tipo genérico significa que uma dica de tipo de `Generator` requer aqueles três parâmetros de tipo que vimos antes:

```
my_coro : Generator[YieldType, SendType, ReturnType]
```

Nas declarações das variáveis de tipo nos parâmetros formais, vemos que `YieldType` e `ReturnType` são covariantes, mas `SendType` é contravariante. Para entender a razão disso, considere que `YieldType` e `ReturnType` são tipos de "saída". Ambos descrevem dados que saem do objeto corrotina—isto é, o objeto gerador quando usado como um objeto corrotina..

Faz sentido que esses parâmetros sejam covariantes, pois qualquer código esperando uma corrotina que produz números de ponto flutuante pode usar uma corrotina que produz inteiros. Por isso `Generator` é covariante em seu parâmetro `YieldType`. O mesmo raciocínio se aplica ao parâmetro `ReturnType`—também covariante.

Usando a notação introduzida na «Seção 15.7.4.2» [fpy.li/ch] (vol.2), a covariância do primeiro e do terceiro parâmetros pode ser expressa pelos símbolos `>` apontando para a mesma direção:

```
float > int
Generator[float, Any, float] > Generator[int, Any, int]
```

`YieldType` e `ReturnType` são exemplos da primeira regra apresentada na «Seção 15.7.4.4» [fpy.li/cj] (vol.2):

1. *Se um parâmetro de tipo formal define um tipo para dados que saem de um objeto, ele pode ser covariante.*

Por outro lado, `SendType` é um parâmetro de "entrada": ele é o tipo do argumento `value` do método `.send(value)` do objeto corrotina. Código cliente que precisa enviar números de ponto flutuante para uma corrotina não pode usar uma corrotina que receba `int` como o `SendType`, porque `float` não é um subtipo de `int`. Em outras palavras, `float` não é *consistente-com* `int`. Mas o cliente pode usar uma corrotina que tenha `complex` como `SendType`, pois `float` é um subtipo de `complex`, e portanto `float` é *consistente-com* `complex`.

A notação `>` torna visível a contravariância do segundo parâmetro:

```
float > int
Generator[Any, float, Any] <: Generator[Any, int, Any]
```

Este é um exemplo da segunda regra geral da variância:

2. *Se um parâmetro de tipo formal define um tipo para dados que entram em um objeto após sua construção inicial, ele pode ser contravariante.*

Esta alegre discussão sobre variância encerra o capítulo mais longo do livro.

17.14. Resumo do capítulo

A iteração está integrada tão profundamente à linguagem que gosto de dizer que Python *groks* iteráveis.^[18] A integração do padrão *Iterator* na semântica de Python é um exemplo perfeito de como padrões de projeto não são aplicáveis a todas as linguagens de programação. No Python, um *Iterator* clássico, implementado "à mão", como no Exemplo 4, não tem qualquer função prática, exceto como exemplo didático.

Neste capítulo, criamos algumas versões de uma classe para iterar sobre palavras individuais em arquivos de texto (que podem ser muito grandes). Vimos como Python usa a função embutida `iter()` para criar iteradores a partir de objetos similares a sequências. Criamos um iterador clássico como uma classe com `__next__()`, e então usamos geradores, para tornar a classe `Sentence` mais concisa e legível a cada nova refatoração.

Daí criamos um gerador de progressões aritméticas, e mostramos como usar o módulo `itertools` para simplificar o código. A isso se seguiu uma revisão da maioria das funções geradoras de uso geral na biblioteca padrão.

A seguir estudamos expressões `yield from` no contexto de geradores simples, com os exemplos `chain` e `tree`.

A última seção foi sobre corrotinas clássicas, um tópico de importância decrescente após a introdução das corrotinas nativas, no Python 3.5. Apesar de difíceis de usar na prática, corrotinas clássicas são os alicerces das corrotinas nativas, e a expressão `yield from` é precursora direta de `await`.

Também abordamos dicas de tipo para os tipos `Iterable`, `Iterator`, e `Generator`—com esse último oferecendo um raro exemplo concreto de um parâmetro de tipo contravariante.

17.15. Para saber mais

Uma explicação técnica detalhada sobre geradores aparece na *A Referência da Linguagem Python*, em *Expressões yield* [fpy.li/9g]. A PEP onde as funções geradoras foram definidas é a *PEP 255—Simple Generators* [fpy.li/pep255].

A documentação do módulo *itertools* [fpy.li/9c] é excelente, especialmente por todos os exemplos incluídos. Apesar das funções daquele módulo serem implementadas em C, a documentação mostra como algumas delas poderiam ser escritas em Python, frequentemente se valendo de outras funções no módulo. Os exemplos de uso também são ótimos; por exemplo, há um trecho mostrando como usar a função *accumulate* para amortizar um empréstimo com juros, dada uma lista de pagamentos ao longo do tempo. Há também a seção *Receitas com itertools* [fpy.li/9h], com funções adicionais de alto desempenho, usando as funções de *itertools* como base.

Além da biblioteca padrão de Python, recomendo o pacote *More Itertools* [fpy.li/17-30], que continua a bela tradição do *itertools*, oferecendo geradores poderosos, acompanhados de muitos exemplos e receitas úteis.

Iterators and Generators, o capítulo 4 de *Python Cookbook*, 3ª ed., de David Beazley e Brian K. Jones (O'Reilly), traz 16 receitas sobre o assunto, de muitos ângulos diferentes, concentradas em aplicações práticas. O capítulo contém algumas receitas esclarecedoras com *yield from*.

Sebastian Rittau—atualmente um dos principais colaboradores do *typeshed*—explica por que iteradores devem ser iteráveis. Ele observou, em 2006, que *Java: Iterators are not Iterable* [fpy.li/17-31] (Java: Iteradores não são Iteráveis).

A sintaxe de *yield from* é explicada, com exemplos, na seção *What's New in Python 3.3* (Novidades no Python 3.3) da *PEP 380—Syntax for Delegating to a Subgenerator* [fpy.li/17-32] (Sintaxe para Delegar para um Subgerador). Meu artigo *Classic Coroutines* [fpy.li/oldcoro] *fluentpython.com* explica *yield from* em profundidade, incluindo pseudo-código em Python de sua implementação em C.

David Beazley é a autoridade máxima sobre geradores e corrotinas clássicas no Python. O *Python Cookbook*, 3rd. ed. [fpy.li/pycook3], (O'Reilly), que ele escreveu com Brian Jones, traz inúmeras receitas com corrotinas. Os tutoriais de Beazley sobre esse tópico nas PyCon são famosos por sua profundidade e abrangência. O primeiro foi na PyCon US 2008: *Generator Tricks for Systems Programmers* [fpy.li/17-33] (Truques com Geradoras para Programadores de Sistemas). Na PyCon US 2009 aconteceu o lendário *A Curious Course on Coroutines and Concurrency* [fpy.li/17-34] (Um Curioso Curso sobre Corrotinas e Concorrência) (links de vídeo difíceis de encontrar para as três partes: parte 1 [fpy.li/17-35], parte 2 [fpy.li/17-36], e parte 3 [fpy.li/17-37]). Seu tutorial na PyCon 2014 em Montreal foi *Generators: The Final Frontier* [fpy.li/17-38] (Geradores: A Fronteira Final), onde ele apresenta mais exemplos de concorrência—então é, na verdade, mais relacionado aos tópicos do Capítulo 21. Dave não consegue deixar de explodir cérebros em suas aulas, então, na última parte de *A Fronteira Final*, corrotinas substituem o padrão clássico *Visitor* em um analisador de expressões aritméticas.

Corrotinas permitem organizar o código de novas maneiras e, assim como a recursão e o polimorfismo (despacho dinâmico), leva um tempo para a gente se dar conta de suas possibilidades. Um exemplo interessante de um algoritmo clássico reescrito com corrotinas aparece no post *Greedy algorithm with coroutines* [fpy.li/17-39] (Algoritmo guloso com corrotinas), de James Powell.

O *Effective Python*, 1ª ed. [fpy.li/17-40] (Addison-Wesley), de Brett Slatkin, tem um excelente capítulo curto chamado *Consider Coroutines to Run Many Functions Concurrently* (Considere as Corrotinas para Executar Muitas Funções de Forma Concorrente). Esse capítulo não aparece na segunda edição de *Effective Python*, mas ainda está «disponível online» [fpy.li/17-41] como um capítulo de amostra. Slatkin apresenta o melhor exemplo que já vi do controle de corrotinas com `yield from`: uma implementação do *Jogo da Vida* [fpy.li/9j], de John Conway, no qual corrotinas gerenciam o estado de cada célula conforme o jogo avança. Refatorei o código do exemplo do Jogo da Vida—separando funções e classes que implementam o jogo dos trechos de teste no código original de Slatkin. Também reescrevi os testes como doctests, então você pode ver o resultados de várias corrotinas e classes sem executar o script. Publiquei o «exemplo refatorado» [fpy.li/17-43] como um *GitHub gist* [fpy.li/17-44].

Ponto de Vista

A interface Iterador minimalista de Python

Na seção *Implementação* do padrão *Iterator*,^[19] a Gangue dos Quatro escreveu:

A interface mínima de Iterator consiste das operações First, Next, IsDone, e CurrentItem.

Entretanto, esta mesma frase tem uma nota de rodapé, onde se lê:

Podemos tornar esta interface ainda menor, fundindo Next, IsDone, e CurrentItem em uma única operação que avança para o próximo objeto e o devolve. Se a travessia estiver encerrada, esta operação daí devolve um valor especial (0, por exemplo), que marca o final da iteração.

Isto é próximo do que temos em Python: um único método `__next__`, faz o serviço. Mas em vez de uma sentinela, que poderia passar despercebida por engano ou distração, a exceção `StopIteration` sinaliza o final da iteração. Simples e correto: esse é o jeito de Python.

Geradoras plugáveis

Qualquer um que gerencie grandes conjuntos de dados encontra muitos usos para geradores. Esta é a história da primeira vez que criei uma solução prática baseada em geradores.

Muitos anos atrás, eu trabalhava na BIREME, uma biblioteca digital operada pela OPAS/OMS (Organização Pan-Americana da Saúde/Organização Mundial da Saúde) em São Paulo, Brasil. Entre os conjuntos de dados bibliográficos criados pela BIREME estão a LILACS (Literatura Latino-Americana e do Caribe em Ciências da Saúde) e a SciELO (Scientific Electronic Library Online), duas bases de dados abrangentes, indexando a literatura de pesquisa em ciências da saúde produzida na região.

Desde o final dos anos 1980, o sistema de banco de dados usado para gerenciar a LILACS é o CDS/ISIS, um banco de dados não-relacional orientado a documentos, criado pela UNESCO. Uma de minhas tarefas era pesquisar alternativas para uma possível migração da LILACS—e eventualmente também a SciELO—para um banco de dados documental moderno de código aberto, tal como o CouchDB ou o MongoDB. Naquela época publiquei um artigo científico explicando o modelo de dados semi-estruturado e as diferentes formas de representar dados CDS/ISIS com registros do tipo JSON: *From ISIS to CouchDB: Databases and Data Models for Bibliographic Records* [fpy.li/17-45] (Do ISIS ao CouchDB: Bancos de Dados e Modelos de Dados para Registros Bibliográficos).

Como parte daquela pesquisa, escrevi um script Python para ler um arquivo CDS/ISIS e escrever um arquivo JSON adequado para importação pelo CouchDB ou pelo MongoDB. Inicialmente, o arquivo lia arquivos no formato ISO-2709, exportados pelo CDS/ISIS. A leitura e a escrita tinham de ser feitas de forma incremental, pois os conjuntos de dados completos eram muito maiores que a memória principal. Isso era fácil: cada iteração do laço for principal lia um registro do arquivo *.iso*, o manipulava e escrevia no arquivo de saída *.json*.

Entretanto, por razões operacionais, foi considerado necessário que o *isis2json.py* suportasse outro formato de dados do CDS/ISIS: os arquivos binários *.mst*, usados em produção na BIREME—para evitar uma exportação dispendiosa para ISO-2709. Agora eu tinha um problema: as bibliotecas usadas para ler arquivos ISO-2709 e *.mst* tinham APIs muito diferentes. E o laço de escrita JSON já era complicado, pois o script aceitava várias opções na linha de comando para reestruturar cada registro de saída. Seria ainda mais complicado ler dados usando duas APIs diferentes no mesmo laço for onde o JSON era produzido.

A solução foi isolar a lógica de leitura em um par de funções geradoras: uma para cada formato de entrada suportado. No fim, dividi o script *isis2json.py* em quatro funções. Você pode ver o código-fonte em Python 2, com suas dependências, no repositório *fluentpython/isis2json* [fpy.li/17-46] no GitHub.^[20]

Aqui está uma visão geral em alto nível de como o script está estruturado:

`main`

A função `main` usa `argparse` para ler opções de linha de comando que configuram a estrutura dos registros de saída. Baseado na extensão do nome do arquivo de entrada, uma função geradora é selecionada para ler os dados e produzir os registros, um por vez.

`iter_iso_records`

Função geradora que lê arquivos `.iso` (que se presume estarem no formato ISO-2709). Ela aceita dois argumentos: o nome do arquivo e `isis_json_type`, uma das opções relacionadas à estrutura do registro. Cada iteração de seu laço `for` lê um registro, cria um `dict` vazio, o preenche com dados dos campos, e produz o `dict`.

`iter_mst_records`

Função geradora que lê arquivos `.mst`.^[21] Se você examinar o código-fonte de `isis2json.py`, vai notar que ela não é tão simples quanto `iter_iso_records`, mas sua interface e estrutura geral é a mesma: a função recebe como argumentos um nome de arquivo e um `isis_json_type`, e entra em um laço `for`, que cria e produz por iteração um `dict`, representando um único registro.

`write_json`

Função que escreve os registros JSON, um por vez. Ela recebe numerosos argumentos, mas o primeiro—`input_gen`—é uma referência para uma função geradora: `iter_iso_records` ou `iter_mst_records`. O laço `for` principal itera sobre os dicionários produzidos pelo gerador `input_gen` escolhido, os reestrutura de diferentes formas, conforme as opções de linha de comando, e anexa o registro JSON ao arquivo de saída.

Aproveitando as funções geradoras, consegui isolar completamente a leitura da escrita. Claro, a maneira mais simples de isolar as duas operações seria ler todos os registros para a memória e então escrevê-los no disco. Mas essa não era uma opção viável, pelo tamanho dos conjuntos de dados. Usando geradores, a leitura e a escrita são intercaladas, então o script pode processar arquivos de qualquer tamanho. Além disso, a lógica

especial para ler um registro em formatos de entrada diferentes está isolada da lógica de reestruturação de cada registro para escrita.

Agora, se precisarmos que *isis2json.py* suporte um formato de entrada adicional—digamos, MARCXML, da Biblioteca do Congresso dos EUA—será fácil acrescentar uma terceira função geradora para implementar a lógica de leitura, sem mudar nada na complicada função `write_json`.

Não foi uma grande inovação, mas é um exemplo real onde os geradores permitiram uma solução eficiente e flexível para processar bancos de dados como um fluxo de registros, mantendo o uso de memória baixo e independente do tamanho do conjunto de dados.

[1] De *Revenge of the Nerds* [fpy.li/17-1] (A Revanche dos Nerds), um post de blog.

[2] NT: Mudei um pouco a tradução, e «sugeri a melhoria» [fpy.li/98] no repositório oficial da tradução PT-BR da documentação do Python.

[3] Agradeço ao revisor técnico Leonardo Rochael por este ótimo exemplo.

[4] Ao revisar esse código, Alex Martelli sugeriu que o corpo deste método poderia ser simplesmente `return iter(self.words)`. Ele está certo: o resultado da invocação de `self.words.__iter__()` também seria um iterador, como deve ser. Entretanto, usei um laço `for` com `yield` aqui para introduzir a sintaxe de uma função geradora, que exige a instrução `yield`, como veremos na próxima seção. Durante a revisão da segunda edição deste livro, Leonardo Rochael sugeriu ainda outro atalho para o corpo de `__iter__`: `yield from self.words`. Também vamos falar de `yield from` mais adiante neste mesmo capítulo.

[5] Eu algumas vezes acrescento um prefixo ou sufixo `gen` ao nomear funções geradoras, mas essa não é uma prática comum. E claro que não é possível fazer isso ao implementar um iterável: o método especial obrigatório deve se chamar `__iter__`.

[6] Agradeço a David Kwast por sugerir esse exemplo.

[7] NT: Em português a literatura usa também avaliação *estrita* e *não estrita*. Optamos pelos termos "preguiçosa" e "ávida", que são mais descritivos.

[8] No Python 2, havia uma função embutida `coerce()`, mas ela não existe mais no Python 3. Foi considerada desnecessária, pois as regras de coerção numérica estão implícitas nos métodos dos operadores aritméticos. Então, a melhor forma que pude imaginar para forçar o valor inicial para o mesmo tipo do restante da série foi realizar a adição e usar seu tipo para converter o resultado. Perguntei sobre isso na Python-list e recebi uma excelente «resposta de Steven D'Aprano» [fpy.li/17-11].

[9] O diretório *17-it-generator/* no «repositório de código» [fpy.li/code] do *Python Fluente* inclui doctests e um script, *aritprog_runner.py*, que roda os testes contra todas as variações dos scripts *aritprog*.py*.

[10] Pode também ser invocado na forma `max(arg1, arg2, ..., [key=?])`, devolvendo então o valor máximo entre os argumentos passados.

[11] Pode também ser invocado na forma `min(arg1, arg2, ..., [key=?])`, devolvendo então o valor mínimo entre os argumentos passados.

[12] Na versão 0.910, a versão mais recente disponível quando escrevi este capítulo), o Mypy ainda utiliza os tipos descontinuados de `typing`.

[13] Slide 33, *Keeping It Straight* em *A Curious Course on Coroutines and Concurrency* [fpy.li/17-18] (Um Curioso Curso sobre Corrotinas e Concorrência).

[14] Este exemplo foi inspirado por um trecho enviado por Jacob Holm à lista Python-ideas, em uma mensagem intitulada *Yield-From: Finalization guarantees* [fpy.li/17-20] (Yield-From: Garantias de finalização). Algumas variantes aparecem mais tarde na mesma thread, e Holm dá mais explicações na «mensagem 003912» [fpy.li/17-21].

[15] Na verdade, ela nunca retorna, a menos que uma exceção interrompa o laço. O Mypy 0.910 aceita tanto `None` quanto `typing.NoReturn` como parâmetro de tipo devolvido pelo gerador—mas ele também aceita `str` naquela posição, então aparentemente o Mypy não consegue analisar completamente o código da corrotina, pelo menos em sua versão 0.910.

[16] Considerei renomear o atributo, mas `count` é o melhor nome para a variável local na corrotina, e é o nome que usei para essa variável em exemplos similares ao longo do livro, então faz sentido usar o mesmo nome no atributo de `Result`. Não hesito em usar `# type: ignore` para evitar as limitações e os aborrecimentos de checadores de tipos estáticos, quando se submeter à ferramenta tornaria o código pior ou desnecessariamente complicado.

[17] Desde o Python 3.7, `typing.Generator` e outros tipos que correspondem a ABCs em `collections.abc` foram refatorados e encapsulados nas ABCs correspondentes, então seus parâmetros genéricos não são visíveis no código-fonte de `typing.py`. Por isso estou fazendo referência ao código-fonte do Python 3.6 aqui.

[18] De acordo com o *Jargon file* [fpy.li/17-26], "to grok" não é meramente entender algo, mas absorver de uma forma que "aquilo se torna parte de você, parte de sua identidade".

[19] Gamma et. al., *Design Patterns: Elements of Reusable Object-Oriented Software*, p. 261.

[20] O código está em Python 2 porque uma de suas dependências opcionais é uma biblioteca Java chamada *Bruma*, que podemos importar quando executamos o script com o Jython—que ainda não suporta Python 3.

[21] A biblioteca usada para ler o complicado formato binário *.mst* é escrita em Java, então esta funcionalidade só está disponível quando *isis2json.py* é executado com o interpretador Jython, versão 2.5 ou superior. Para mais detalhes, veja o arquivo *README.rst* [fpy.li/17-47] no repositório. As dependências são importadas dentro das funções geradoras que precisam delas, então o script pode rodar mesmo quando apenas uma das bibliotecas externas está instalada.

Capítulo 18. Instruções with, match, e blocos else

Gerenciadores de contexto podem vir a ser quase tão importantes quanto a própria sub-rotina. Só arranhamos a superfície das possibilidades. [...] Basic tem uma instrução with, há instruções with em várias linguagens. Mas elas não fazem a mesma coisa, todas fazem algo muito raso, economizam consultas a atributos com o operador ponto (.), elas não configuram e desfazem ambientes. Não pense que é a mesma coisa só porque o nome é igual. A instrução with é mais que isso.^[1]

— Raymond Hettinger, um eloquente evangelista de Python

Este capítulo é sobre mecanismos de controle de fluxo não muito comuns em outras linguagens e que, por essa razão, podem ser ignorados ou subutilizados em Python. São eles:

- A instrução with e o protocolo de gerenciamento de contexto
- A instrução match/case para casamento de padrões (*pattern matching*)
- A cláusula else nas instruções for, while, e try

A instrução with cria um contexto temporário e o desfaz com segurança, sob o controle de um objeto gerenciador de contexto. Isso previne erros e reduz código repetitivo, tornando as APIs ao mesmo tempo mais seguras e mais fáceis de usar. Programadores Python estão encontrando muitos usos para blocos with além do fechamento automático de arquivos.

Já estudamos casamento de padrões em capítulos anteriores, mas aqui veremos como a gramática de uma linguagem de programação pode ser expressa como padrões de sequências. Por isso match/case é uma ferramenta eficiente para criar processadores de linguagem fáceis de entender e de estender. Vamos examinar um interpretador completo de subconjunto pequeno mas funcional da linguagem Scheme. As mesmas ideias poderiam ser aplicadas no desenvolvimento de uma linguagem de templates ou uma DSL (*Domain-Specific Language*, Linguagem de Domínio Específico) para representar regras de negócio em um sistema maior.

A cláusula `else` não é grande coisa, mas ajuda a transmitir a intenção por trás do código quando usada corretamente com as instruções `for`, `while` e `try`.

18.1. Novidades neste capítulo

A Seção 18.3 é nova: um exemplo maior de casamento de padrões.

Também atualizei a Seção 18.2.1 para incluir alguns recursos do módulo `contextlib` adicionados desde o Python 3.6, e os novos gerenciadores de contexto agrupados entre parênteses, desde o Python 3.10.

Vamos começar com a poderosa instrução `with`.

18.2. Instrução `with` e gerenciadores de contexto

Objetos gerenciadores de contexto servem para controlar uma instrução `with`, da mesma forma que iteradores servem para controlar uma instrução `for`.

A instrução `with` foi projetada para simplificar alguns usos comuns de `try/finally`, que garantem que alguma operação seja realizada após um bloco de código, mesmo que o bloco termine com um `return`, uma exceção, ou uma chamada `sys.exit()`. O código no bloco `finally` normalmente libera um recurso crítico ou restaura um estado anterior que havia sido temporariamente modificado.

A comunidade Python está encontrando novos usos criativos para gerenciadores de contexto. Alguns exemplos, da biblioteca padrão, são:

- Gerenciar transações no módulo `sqlite3`—veja «Usando a conexão como gerenciador de contexto» [fpy.li/a3].
- Manipular travas, condições e semáforos de forma segura—como descrito na «documentação do módulo `threading`» [fpy.li/9q].
- Configurar ambientes customizados para operações aritméticas com objetos `Decimal`—veja a «documentação de `decimal.localcontext`» [fpy.li/9r].
- Remendar (*patch*) objetos para testes—veja a função «`unittest.mock.patch`» [fpy.li/9s].

A interface gerenciador de contexto consiste dos métodos `__enter__` and `__exit__`. No topo do `with`, Python chama o método `__enter__` do objeto gerenciador de contexto. Quando o bloco `with` encerra ou termina por qualquer razão, Python chama `__exit__` no objeto gerenciador de contexto.

O exemplo mais comum é garantir que um arquivo seja fechado. O Exemplo 1 é uma demonstração detalhada do uso do `with` para fechar um arquivo.

Exemplo 1. Demonstração do uso de um objeto arquivo como gerenciador de contexto

```
>>> with open('mirror.py') as fp: ①
...     src = fp.read(60) ②
...
>>> len(src)
60
>>> fp ③
<_io.TextIOWrapper name='mirror.py' mode='r' encoding='UTF-8'>
>>> fp.closed, fp.encoding ④
(True, 'UTF-8')
>>> fp.read(60) ⑤
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: I/O operation on closed file.
```

- ① `fp` está vinculado ao arquivo de texto aberto, pois o método `__enter__` do arquivo devolve `self`.
- ② Lê 60 caracteres Unicode de `fp`.
- ③ A variável `fp` continua disponível após o bloco; uma instrução `with` não define um novo escopo, como faz a instrução `def`.
- ④ Podemos acessar atributos do objeto `fp`.
- ⑤ Mas não podemos mais ler texto de `fp` pois, no final do bloco `with`, o método `TextIOWrapper.__exit__` foi invocado, e este método fecha o arquivo.

A nota ① no Exemplo 1 é sutil mas muito importante: o objeto gerenciador de contexto é o resultado da avaliação da expressão após o `with`, mas o valor vinculado à variável alvo (na cláusula `as`) é o resultado devolvido pelo método `__enter__` do objeto gerenciador de contexto.

Acontece que a função `open()` devolve uma instância de `TextIOWrapper`, e o método `__enter__` dessa classe devolve `self`. Mas em uma classe diferente, o método `__enter__` pode devolver algum outro objeto em vez do próprio gerenciador de contexto.

Quando o fluxo de controle sai do bloco `with` de qualquer forma, o método `__exit__` é invocado no objeto gerenciador de contexto, e não no que quer que `__enter__` tenha devolvido.

A cláusula `as` da instrução `with` é opcional. No caso de `open`, sempre precisamos obter uma referência para o arquivo, para podermos invocar seus métodos. Mas alguns gerenciadores de contexto devolvem `None`, pois não têm um objeto útil para entregar ao usuário.

O Exemplo 2 mostra o funcionamento de um gerenciador de contexto perfeitamente frívolo, projetado para ressaltar a diferença entre o gerenciador de contexto e o objeto devolvido por seu método `__enter__`.

Exemplo 2. Testando a classe gerenciadora de contexto LookingGlass

```
>>> from mirror import LookingGlass
>>> with LookingGlass() as what: ①
...     print('Alice, Kitty and Snowdrop') ②
...     print(what)
...
pordwonS dna yttiK ,ecilA
YKCOWREBBAJ
>>> what ③
'JABBERWOCKY'
>>> print('Back to normal.') ④
Back to normal.
```

- ① O gerenciador de contexto é uma instância de `LookingGlass`; Python chama `__enter__` no gerenciador de contexto e o resultado é vinculado a `what`.
- ② Exibe uma `str`, depois o valor da variável alvo `what`. A saída de cada `print` é invertida.
- ③ Agora o bloco `with` terminou. Podemos ver que o valor devolvido por `__enter__`, armazenado em `what`, é a string `'JABBERWOCKY'`.

④ A saída do programa não está mais invertida.

O Exemplo 3 mostra a implementação de LookingGlass.

Exemplo 3. mirror.py: código da classe gerenciadora de contexto LookingGlass

```
import sys

class LookingGlass:

    def __enter__(self): ①
        self.original_write = sys.stdout.write ②
        sys.stdout.write = self.reverse_write ③
        return 'JABBERWOCKY' ④

    def reverse_write(self, text): ⑤
        self.original_write(text[::-1])

    def __exit__(self, exc_type, exc_value, traceback): ⑥
        sys.stdout.write = self.original_write ⑦
        if exc_type is ZeroDivisionError: ⑧
            print('Please DO NOT divide by zero!')
            return True ⑨
```

⑩

- ① Python invoca `__enter__` sem argumentos além de `self`.
- ② Armazena o método `sys.stdout.write` original, para podermos restaurá-lo mais tarde.
- ③ Faz um *monkey-patch* em `sys.stdout.write`, substituindo-o com nosso próprio método.
- ④ Devolve a string `'JABBERWOCKY'`, apenas para termos algo para colocar na variável alvo `what`.
- ⑤ Nosso substituto de `sys.stdout.write` inverte o argumento `text` e chama a implementação original.
- ⑥ Se tudo correu bem, Python chama `__exit__` com `None, None, None`; se ocorreu uma exceção, os três argumentos recebem dados da exceção, como descrito logo após este exemplo.

- ⑦ Restaura o método original em `sys.stdout.write`.
- ⑧ Se a exceção não é `None` e seu tipo é `ZeroDivisionError`, exibe uma mensagem...
- ⑨ ...e devolve `True`, para informar o interpretador que a exceção foi tratada.
- ⑩ Se `__exit__` devolve `None` ou qualquer valor *falso*, qualquer exceção levantada dentro do bloco `with` será propagada.



Quando aplicações reais tomam o controle da saída padrão, elas frequentemente desejam substituir `sys.stdout` com outro objeto similar a um arquivo por algum tempo, depois voltar ao original. O gerenciador de contexto `contextlib.redirect_stdout` [fpy.li/18-6] faz exatamente isso: passe a ele seu objeto similar a um arquivo que substituirá `sys.stdout`.

O interpretador chama o método `__enter__` sem qualquer argumento—além do `self` implícito. Os três argumentos passados a `__exit__` são:

`exc_type`

A classe da exceção (por exemplo, `ZeroDivisionError`).

`exc_value`

A instância da exceção. Algumas vezes, parâmetros passados para o construtor da exceção—tal como a mensagem de erro—podem ser encontrados em `exc_value.args`.

`traceback`

Um objeto `traceback`.^[2]

Para uma visão detalhada de como funciona um gerenciador de contexto, vejamos o Exemplo 4, onde `LookingGlass` é usado fora de um bloco `with`, de forma que podemos invocar manualmente seus métodos `__enter__` e `__exit__`.

Exemplo 4. Exercitando o `LookingGlass` sem um bloco `with`

```
>>> from mirror import LookingGlass
>>> manager = LookingGlass() ①
>>> manager # doctest: +ELLIPSIS
<mirror.LookingGlass object at 0x...>
>>> monster = manager.__enter__() ②
```

```

>>> monster == 'JABBERWOCKY' ③
eurT
>>> monster
'YKCOWREBBAJ'
>>> manager # doctest: +ELLIPSIS
>... ta tcejbo ssalGgnikool.rorrim<
>>> manager.__exit__(None, None, None) ④
>>> monster
'JABBERWOCKY'

```

- ① Instancia e inspeciona a instância de `manager`.
- ② Chama o método `__enter__` do `manager` e guarda o resultado em `monster`.
- ③ `monster` é a string `'JABBERWOCKY'`. O identificador `True` aparece invertido, porque toda a saída via `stdout` passa pelo método `write`, que modificamos em `__enter__`.
- ④ Chama `manager.__exit__` para restaurar o `stdout.write` original.

Gerenciadores de contexto entre parênteses

Python 3.10 adotou um novo parser [fpy.li/pep617] (analisador sintático), mais poderoso que o antigo parser LL(1) [fpy.li/18-8]. Isso permitiu introduzir novas sintaxes que não eram viáveis anteriormente. Uma melhoria na sintaxe foi permitir gerenciadores de contexto agrupados entre parênteses, assim:



```

with (
    CtxManager1() as example1,
    CtxManager2() as example2,
    CtxManager3() as example3,
):
    ...

```

Antes do 3.10, as linhas acima teriam que ser escritas como blocos `with` aninhados.

A biblioteca padrão inclui o pacote `contextlib`, com funções, classes e decoradores convenientes para desenvolver, combinar e usar gerenciadores de contexto.

18.2.1. Utilitários da `contextlib`

Antes de desenvolver suas próprias classes gerenciadoras de contexto, dê uma olhada em `contextlib` [fpy.li/9t] (Utilitários para contextos da instrução `with`), na documentação de Python. Pode ser que você esteja prestes a escrever algo que já existe, ou talvez exista uma classe ou algum invocável que tornará seu trabalho mais fácil.

Além do gerenciador de contexto `redirect_stdout` mencionado logo após o Exemplo 3, o `redirect_stderr` foi acrescentado no Python 3.5—ele faz o mesmo que seu par mais antigo, mas com as saídas direcionadas para `stderr`.

O pacote `contextlib` também inclui:

`closing`

Uma função para criar gerenciadores de contexto a partir de objetos que forneçam um método `close()` mas não implementam a interface `__enter__/_exit__`.

`suppress`

Um gerenciador de contexto para ignorar temporariamente exceções passadas como parâmetros.

`nullcontext`

Um gerenciador de contexto que não faz nada, para simplificar a lógica condicional em torno de objetos que podem não implementar um gerenciador de contexto adequado. Ele serve como um substituto quando o código condicional antes do bloco `with` pode ou não fornecer um gerenciador de contexto para a instrução `with`. Adicionado no Python 3.7.

O módulo `contextlib` fornece classes e um decorador que são mais largamente aplicáveis que os decoradores mencionados acima:

`@contextmanager`

Um decorador que permite construir um gerenciador de contexto a partir de uma simples função geradora, em vez de criar uma classe e implementar a interface. Veja a Seção 18.2.2.

AbstractContextManager

Uma ABC que formaliza a interface gerenciador de contexto, e torna um pouco mais fácil criar classes gerenciadoras de contexto, através de subclasses—adicionada no Python 3.6.

ContextDecorator

Uma classe base para definir gerenciadores de contexto baseados em classes que podem também ser usadas como decoradores de função, rodando a função inteira dentro de um contexto gerenciado.

ExitStack

Um gerenciador de contexto que permite entrar em um número variável de gerenciadores de contexto. Quando o bloco `with` termina, `ExitStack` chama os métodos `__exit__` dos gerenciadores de contexto empilhados na ordem LIFO (Last In, First Out, *Último a Entrar, Primeiro a Sair*). Use essa classe quando você não sabe de antemão em quantos gerenciadores de contexto será necessário entrar no bloco `with`; por exemplo, ao abrir ao mesmo tempo todos os arquivos de uma lista arbitrária de arquivos.

Com Python 3.7, `contextlib` acrescentou `AbstractAsyncContextManager`, `@asynccontextmanager`, e `AsyncExitStack`. Eles são similares aos utilitários equivalentes sem a parte `async` no nome, mas projetados para uso com a nova instrução `async with`, tratada no Capítulo 21.

Entre estas ferramentas, a mais fácil de usar é o decorador `@contextmanager`, então ele merece mais atenção. Este decorador também é interessante por mostrar um uso da instrução `yield` não relacionado a iteração.

18.2.2. Usando o `@contextmanager`

O decorador `@contextmanager` é uma ferramenta elegante e prática, que une três recursos distintos de Python: um decorador de função, um gerador, e a instrução `with`.

Usar o `@contextmanager` reduz o código repetitivo na criação de um gerenciador de contexto: em vez de escrever toda uma classe com métodos `__enter__`/`__exit__`, você só precisa implementar um gerador com uma única instrução `yield`, que deve produzir o que o método `__enter__` deveria devolver.

Em um gerador decorado com `@contextmanager`, o `yield` divide o corpo da função em duas partes: tudo que vem antes do `yield` será executado no início do bloco `with`, quando o interpretador chama `__enter__`; o código após o `yield` será executado quando `__exit__` é invocado, no final do bloco.

O Exemplo 5 substitui a classe `LookingGlass` do Exemplo 3 por uma função geradora.

Exemplo 5. `mirror_gen.py`: um gerenciador de contexto implementado com um gerador

```
import contextlib
import sys

@contextlib.contextmanager ①
def looking_glass():
    original_write = sys.stdout.write ②

    def reverse_write(text): ③
        original_write(text[::-1])

    sys.stdout.write = reverse_write ④
    yield 'JABBERWOCKY' ⑤
    sys.stdout.write = original_write ⑥
```

- ① Aplica o decorador `contextmanager`.
- ② Preserva o método `sys.stdout.write` original.
- ③ `reverse_write` pode invocar `original_write` mais tarde, pois ele está disponível em sua clausura (closure).
- ④ Substitui `sys.stdout.write` por `reverse_write`.
- ⑤ Produz o valor que será vinculado à variável alvo na cláusula `as` da instrução `with`. O gerador é suspenso neste ponto, enquanto o corpo do `with` é executado.
- ⑥ Quando o fluxo de controle sai do bloco `with`, a execução continua após o `yield`; neste ponto o `sys.stdout.write` original é restaurado.

O Exemplo 6 mostra a função `looking_glass` em operação.

Exemplo 6. Testando a função gerenciadora de contexto `looking_glass`

```
>>> from mirror_gen import looking_glass
>>> with looking_glass() as what: ①
...     print('Alice, Kitty and Snowdrop')
...     print(what)
...
pordwonS dna yttiK ,ecilA
YKCOWREBBAJ
>>> what
'JABBERWOCKY'
>>> print('back to normal')
back to normal
```

- ① A única diferença do Exemplo 2 é o nome do gerenciador de contexto: `looking_glass` em vez de `LookingGlass`.

O decorador `contextlib.contextmanager` envolve a função em uma classe que implementa os métodos `__enter__` e `__exit__`.^[3]

O método `__enter__` daquela classe:

1. Invoca a função geradora para obter um objeto gerador—vamos chamá-lo de `gen`.
2. Invoca `next(gen)` para executar o gerador até o `yield`.
3. Devolve o valor produzido por `next(gen)`, para permitir que o usuário o vincule a uma variável usando o cláusula `as` da instrução `with`.

Quando o bloco `with` termina, o método `__exit__`:

1. Verifica se uma exceção foi passada no argumento `exc_type`; em caso afirmativo, `gen.throw(exception)` é invocado, fazendo com que a exceção seja levantada na posição do `yield`, dentro do corpo da função geradora.
2. Caso contrário, `next(gen)` é invocado, retomando a execução do corpo da função geradora após o `yield`.

O Exemplo 5 tem um defeito: se uma exceção for levantada no corpo do bloco `with`, o interpretador Python vai capturá-la e levantá-la novamente na expressão `yield` dentro de `looking_glass`. Mas não há tratamento de erro ali, então o gerador `looking_glass` vai terminar sem nunca restaurar o método `sys.stdout.write` original, deixando o sistema em um estado inconsistente.

O Exemplo 7 agora trata a exceção `ZeroDivisionError`, tornando-o funcionalmente equivalente ao Exemplo 3, que é uma classe.

Exemplo 7. `mirror_gen_exc.py`: gerenciador de contexto baseado em gerador com tratamento de erro—mesmo comportamento de Exemplo 3

```
import contextlib
import sys

@contextlib.contextmanager
def looking_glass():
    original_write = sys.stdout.write

    def reverse_write(text):
        original_write(text[::-1])

    sys.stdout.write = reverse_write
    msg = '' ①
    try:
        yield 'JABBERWOCKY'
    except ZeroDivisionError: ②
        msg = 'Please DO NOT divide by zero!'
    finally:
        sys.stdout.write = original_write ③
        if msg:
            print(msg) ④
```

- ① Cria uma variável para uma possível mensagem de erro; essa é a primeira mudança em relação a Exemplo 5.
- ② Trata `ZeroDivisionError`, fixando uma mensagem de erro.
- ③ Desfaz o *monkey-patching* de `sys.stdout.write`.
- ④ Mostra a mensagem de erro, se ela foi determinada.

Lembre-se de que o método `__exit__` diz ao interpretador que ele tratou a exceção ao devolver um valor *verdadeiro*; nesse caso, o interpretador suprime a exceção.

Por outro lado, se `__exit__` não devolver explicitamente um valor, o interpretador recebe o habitual `None`, e propaga a exceção. Com o `@contextmanager`, o comportamento default é invertido: o método `__exit__` fornecido pelo decorador assume que qualquer exceção enviada para o gerador está tratada e deve ser suprimida.



Ter um `try/finally` (ou um bloco `with`) em torno do `yield` é o preço inescapável do uso de `@contextmanager`, porque você nunca sabe o que os usuários do seu gerenciador de contexto vão fazer dentro do bloco `with`.^[4]

Um recurso pouco conhecido do `@contextmanager` é que os geradores decorados com ele também podem ser usados como decoradores.^[5] Isso ocorre porque `@contextmanager` é implementado com a classe `contextlib.ContextDecorator`.

O Exemplo 8 mostra o gerenciador de contexto `looking_glass` do Exemplo 5 sendo usado como um decorador.

Exemplo 8. O gerenciador de contexto `looking_glass` também funciona como um decorador.

```
>>> @looking_glass()
... def verse():
...     print('The time has come')
...
>>> verse() ①
emoc sah emit ehT
>>> print('back to normal') ②
back to normal
```

① `looking_glass` faz seu trabalho antes e depois do corpo de `verse` rodar.

② Isso confirma que o `sys.write` original foi restaurado.

Compare o Exemplo 8 com o Exemplo 6, onde `looking_glass` é usado como um gerenciador de contexto.

Um exemplo real interessante de `@contextmanager` fora da biblioteca padrão é descrito em *Easy in-place file rewriting* [fpy.li/18-11] (Reescrita de arquivo no mesmo lugar] de Martijn Pieters. O Exemplo 9 mostra como usar a função `inplace` apresentada naquele artigo.

Exemplo 9. Um gerenciador de contexto para reescrever arquivos no lugar

```
import csv

with inplace(csvfilename, 'r', newline='') as (inhf, outfh):
    reader = csv.reader(inhf)
    writer = csv.writer(outfh)

    for row in reader:
        row += ['new', 'columns']
        writer.writerow(row)
```

A função `inplace` constrói um gerenciador de contexto que fornece a você duas referências para o mesmo arquivo (`inhf` e `outfh` no exemplo), permitindo que seu código leia e escreva nele ao mesmo tempo. Isto é mais fácil de usar que a «função `fileinput.input`» [fpy.li/9v] da biblioteca padrão (que, por sinal, também fornece um gerenciador de contexto).

Se você quiser estudar o código-fonte do `inplace` de Martijn (listado no «post» [fpy.li/18-11]), encontre a palavra reservada `yield`: tudo antes dela lida com configurar o contexto, que implica criar um arquivo de backup, então abrir e produzir referências para os objetos de leitura e escrita que serão devolvidos pela chamada a `__enter__`. O processamento do `__exit__` após o `yield` fecha os objetos vinculados ao arquivo e, se algo deu errado, restaura o arquivo do backup.

Isto conclui nossa revisão da instrução `with` e dos gerenciadores de contexto.

Vamos agora estudar a instrução `match/case` em um exemplo maior do que aqueles que vimos quando falamos de casamenteo de padrões nos capítulos anteriores.

18.3. Estudo de caso: match/case no *lis.py*

Na «Seção 2.6.1» [fpy.li/ck] (vol.1), vimos exemplos de sequências de padrões extraídos da função `evaluate` do interpretador *lis.py* de Peter Norvig, portado para Python 3.10. Nesta seção quero apresentar uma visão geral do *lis.py*, e também explorar todas as cláusulas `case` de `evaluate`, explicando não apenas os padrões, mas também o que o interpretador faz em cada `case`.

Além de mostrar mais casamento de padrões, escrevi essa seção por três razões:

1. O *lis.py* de Norvig é um lindo exemplo de código Python idiomático.
2. A simplicidade do Scheme é uma aula magna de design de linguagens.
3. Aprender como um interpretador funciona me deu um entendimento mais profundo sobre Python e sobre linguagens de programação em geral—interpretadas ou compiladas.

Antes de olhar o código Python, vamos ver um pouquinho de Scheme, para você poder entender este estudo de caso—pensando em quem nunca viu Scheme e Lisp antes.

18.3.1. A sintaxe do Scheme

No Scheme não há diferença sintática entre expressões e instruções (*statements*), como temos em Python. Também não existem operadores infixos. Todas as expressões usam a notação prefixa, como `(+ x 13)` em vez de `x + 13`. A mesma notação prefixa é usada para chamadas de função—por exemplo, `(gcd x 13)`—e formas especiais—por exemplo, `(define x 13)`, que em Python escreveríamos como uma instrução de atribuição: `x = 13`.

A notação usada no Scheme e na maioria dos dialetos de Lisp é conhecida como *S-expression* (expressão-S).^[6]

O Exemplo 10 mostra um exemplo simples em Scheme.

Exemplo 10. Maior divisor comum em Scheme

```
(define (mod m n)
  (- m (* n (quotient m n))))

(define (gcd m n)
  (if (= n 0)
      m
      (gcd n (mod m n))))

(display (gcd 18 45))
```

O Exemplo 10 mostra três expressões em Scheme: duas definições de função—`mod` e `gcd`—e uma chamada a `display`, que vai devolver 9, o resultado de `(gcd 18 45)`. O Exemplo 11 é o mesmo código em Python (mais curto que a explicação em português do «algoritmo recursivo de Euclides» [fpy.li/9w]).

Exemplo 11. Igual ao Exemplo 10, mas escrito em Python

```
def mod(m, n):
    return m - (m // n * n)

def gcd(m, n):
    if n == 0:
        return m
    else:
        return gcd(n, mod(m, n))

print(gcd(18, 45))
```

Em Python idiomático, eu usaria o operador `%` em vez de reinventar `mod`, e seria mais eficiente usar um laço `while` em vez de recursão. Minha intenção foi mostrar duas definições de funções, e fazer os exemplos o mais similares possível, para ajudar você a ler o código Scheme.

O Scheme não tem instruções de laço como `while` ou `for`. Toda iteração é feita com recursão. Observe que não há atribuições nos exemplos em Python e Scheme. O uso intensivo de recursão e o uso mínimo de atribuição são marcas registradas do estilo funcional de programação.^[7]

Agora vamos revisar o código da versão Python 3.10 do *lis.py*. O código-fonte completo, com testes, está no diretório *18-with-match/lispy/py3.10/* [fpy.li/18-15], do repositório *fluentpython/example-code-2e* [fpy.li/code].

18.3.2. Importações e tipos

O Exemplo 12 mostra as primeiras linhas do *lis.py*. O uso do `TypeAlias` e do operador de união de tipos `|` exige Python 3.10.

Exemplo 12. lis.py: início do arquivo

```
import math
import operator as op
from collections import ChainMap
from itertools import chain
from typing import Any, TypeAlias, NoReturn

Symbol: TypeAlias = str
Atom: TypeAlias = float | int | Symbol
Expression: TypeAlias = Atom | list
```

Os tipos definidos são:

Symbol

Só um alias para `str`. Em *lis.py*, `Symbol` é usado para identificadores; não há um tipo de dados string, com operações como fatiamento (*slicing*), partição (*splitting*), etc.^[8]

Atom

Um elemento sintático simples, tal como um número ou um `Symbol`—ao contrário de uma estrutura composta, formada por vários elementos distintos, como uma lista.

Expression

Os componentes básicos de programas Scheme são expressões feitas de átomos e listas, possivelmente aninhadas.

18.3.3. O parser

O parser (*analisador sintático*) de Norvig tem 36 linhas de código que exibem o poder de Python aplicado ao tratamento da sintaxe recursiva simples das expressões-S—sem strings, comentários, macros e outros recursos que tornam a análise sintática do Scheme padrão mais complicada (Exemplo 13).

Exemplo 13. lis.py: as principais funções do analisador

```
def parse(program: str) -> Expression:
    "Read a Scheme expression from a string."
    return read_from_tokens(tokenize(program))

def tokenize(s: str) -> list[str]:
    "Convert a string into a list of tokens."
    return s.replace('(', ' ( ').replace(')', ' ) ').split()

def read_from_tokens(tokens: list[str]) -> Expression:
    "Read an expression from a sequence of tokens."
    # mais código do analisador omitido na listagem do livro
```

A principal função deste grupo é `parse`, que recebe uma expressão-S em forma de `str` e devolve um objeto `Expression`, como definido no Exemplo 12: um `Atom` ou uma `list` que pode conter mais átomos e listas aninhadas.

Norvig usa um truque elegante em `tokenize`: ele acrescenta espaços antes e depois de cada parêntese na entrada, e então a particiona com `split`, resultando em uma lista de *tokens* (símbolos sintáticos) com `'('` e `)'` como itens separados. Este atalho funciona porque não há um tipo `string` no pequeno Scheme de *lis.py*, então todo `'('` ou `)'` é um delimitador de expressão. O código recursivo do analisador está em `read_from_tokens`, uma função de 14 linhas que você pode ler no repositório *fluentpython/example-code-2e* [fpy.li/18-17]. Vou pular isso, pois quero me concentrar em outras partes do interpretador.

Aqui estão alguns doctests extraídos do *lispy/py3.10/examples_test.py* [fpy.li/18-18]:

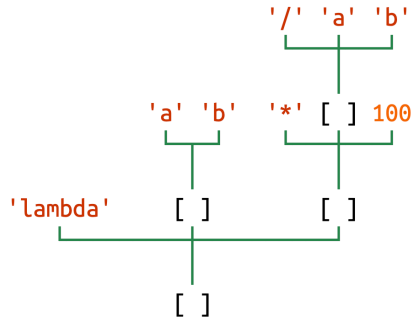
```
>>> from lis import parse
>>> parse('1.5')
1.5
>>> parse('ni!')
'ni!'
>>> parse('(gcd 18 45)')
['gcd', 18, 45]
>>> parse(''
... (define double
...   (lambda (n)
...     (* n 2)))
... ''')
['define', 'double', ['lambda', ['n'], ['*', 'n', 2]]]
```

As regras de avaliação deste subconjunto do Scheme são simples:

1. Um símbolo sintático que se pareça com um número é tratado como um float ou um int.
2. Todo o resto que não seja um '(' ou um ')' é considerado um Symbol—uma str, a ser usado como um identificador. Isso inclui texto no código-fonte como +, set!, e make-counter, que são três identificadores válidos em Scheme, mas não em Python.
3. Expressões dentro de '(' e ')' são avaliadas recursivamente como listas contendo átomos ou listas aninhadas que podem conter átomos ou mais listas aninhadas.

Usando a terminologia do interpretador Python, a saída de parse é uma AST (*Abstract Syntax Tree*—Árvore Sintática Abstrata): uma representação conveniente de um programa Scheme como listas aninhadas formando uma estrutura similar a uma árvore, onde a lista mais externa é o tronco, listas internas são os galhos, e os átomos são as folhas (Figura 1).

```
(lambda (a b) (* (/ a b) 100))
```



```
['lambda', ['a', 'b'], ['*', ['/', 'a', 'b'], 100]]
```

Figura 1. Uma expressão lambda de Scheme, representada como código-fonte (sintaxe concreta de expressões-S), como uma árvore, e como uma sequência de objetos Python (sintaxe abstrata).

Agora veremos como é construído o ambiente (*environment*) que fornece as definições usadas pelos programas dos usuários, semelhante ao módulo `builtins` do Python.

18.3.4. O ambiente

A classe `Environment` estende `collections.ChainMap`, acrescentando o método `change`, para atualizar um valor dentro de um dos dicts encadeados que as instâncias de `ChainMap` guardam no atributo `self.maps`, que é lista de mapeamentos. O método `change` é necessário para suportar a instrução (`set! ...`) do Scheme, descrita mais tarde.

Exemplo 14. lis.py: a classe Environment

```
class Environment(ChainMap[Symbol, Any]):
    "A ChainMap that allows changing an item in-place."

    def change(self, key: Symbol, value: Any) -> None:
        "Find where key is defined and change the value there."
        for map in self.maps:
            if key in map:
                map[key] = value # type: ignore[index]
                return
        raise KeyError(key)
```

Observe que o método `change` só atualiza chaves existentes.^[9] Tentar mudar uma chave não encontrada causa um `KeyError`.

Esse doctest mostra como `Environment` funciona:

```
>>> from lis import Environment
>>> inner_env = {'a': 2}
>>> outer_env = {'a': 0, 'b': 1}
>>> env = Environment(inner_env, outer_env)
>>> env['a'] ①
2
>>> env['a'] = 111 ②
>>> env['c'] = 222
>>> env
Environment({'a': 111, 'c': 222}, {'a': 0, 'b': 1})
>>> env.change('b', 333) ③
>>> env
Environment({'a': 111, 'c': 222}, {'a': 0, 'b': 333})
```

- ① Ao ler os valores, `Environment` funciona como `ChainMap`: as chaves são procuradas nos mapeamentos aninhados da esquerda para a direita. Por isso o valor de `a` no `outer_env` é encoberto pelo valor em `inner_env`.
- ② Atribuir com `[]` sobrescreve ou insere novos itens, mas sempre no primeiro mapeamento, `inner_env` nesse exemplo.
- ③ `env.change('b', 333)` busca a chave `b` e atribui a ela um novo valor no mesmo lugar, no `outer_env`

A seguir temos a função `standard_env()`, que constrói e devolve um `Environment` carregado com funções pré-definidas, similar ao módulo `__builtins__` de Python, que está sempre disponível (Exemplo 15).

Exemplo 15. lis.py: `standard_env()` constrói e devolve o ambiente global

```
def standard_env() -> Environment:
    "An environment with some Scheme standard procedures."
    env = Environment()
    env.update(vars(math))    # sin, cos, sqrt, pi, ...
    env.update({
        '+': op.add,
        '-': op.sub,
        '*': op.mul,
        '/': op.truediv,
        # omitidas: várias definições de funções
        'abs': abs,
        'append': lambda *args: list(chain(*args)),
        'apply': lambda proc, args: proc(*args),
        'begin': lambda *x: x[-1],
        'car': lambda x: x[0],
        'cdr': lambda x: x[1:],
        # omitidas: várias definições de funções
        'number?': lambda x: isinstance(x, (int, float)),
        'procedure?': callable,
        'round': round,
        'symbol?': lambda x: isinstance(x, Symbol),
    })
    return env
```

Resumindo, o mapeamento `env` é carregado com:

- Todas as funções do módulo `math` de Python;
- Operadores selecionados do módulo `op` de Python;
- Funções simples porém poderosas construídas com o `lambda` de Python;
- Estruturas e entidades embutidas de Python, algumas renomeadas, como `callable` para `procedure?`, ou mapeadas diretamente, como `round`.

18.3.5. O REPL

O código do REPL (read-eval-print-loop, *laço-lê-calcula-imprime*) de Norvig é fácil de entender, mas não é amigável para o usuário (veja o Exemplo 16). Se nenhum argumento de linha de comando é passado a *lis.py*, a função `repl()` é invocada por `main()`—definida no final do módulo. No prompt de *lis.py*>, devemos digitar expressões corretas e completas; se esquecermos de fechar um parêntese, *lis.py* se encerra.^[10]

Exemplo 16. As funções do REPL

```
def repl(prompt: str = 'lis.py> ') -> NoReturn:
    "A prompt-read-eval-print loop."
    global_env = Environment({}, standard_env())
    while True:
        ast = parse(input(prompt))
        val = evaluate(ast, global_env)
        if val is not None:
            print(lispstr(val))

def lispstr(exp: object) -> str:
    "Convert a Python object back into a Lisp-readable string."
    if isinstance(exp, list):
        return '(' + ' '.join(map(lispstr, exp)) + ')'
    else:
        return str(exp)
```

Segue uma breve explicação sobre essas duas funções:

`repl(prompt: str = 'lis.py> ') → NoReturn`

Chama `standard_env()` para provisionar as funções embutidas para o ambiente global, então entra em um laço infinito, lendo e avaliando cada linha de entrada, calculando-a no ambiente global, e exibindo o resultado—a menos que seja `None`. O `global_env` pode ser modificado por `evaluate`. Por exemplo, quando o usuário define uma nova variável global ou uma função nomeada, ela é armazenada no primeiro mapeamento do ambiente—o dict vazio na chamada ao construtor de `Environment` na primeira linha de `repl`.

`lispstr(exp: object) → str`

A função inversa de parse: dado um objeto Python representando a AST de uma expressão, `lispstr` devolve o código-fonte correspondente. Por exemplo, dado `['+', 2, 3]`, o resultado é `'(+ 2 3)'`.

18.3.6. O avaliador de expressões

Agora podemos apreciar a beleza do avaliador de expressões de Norvig—ainda mais elegante com `match/case`. A função `evaluate` no Exemplo 17 recebe uma `Expression` (construída por `parse`) e um `Environment`.

O corpo de `evaluate` é composto por uma única instrução `match` com uma expressão `exp` como sujeito. Os padrões de `case` expressam a sintaxe e a semântica do Scheme com uma clareza impressionante.

Exemplo 17. `evaluate` recebe uma expressão e calcula seu valor

```
KEYWORDS = ['quote', 'if', 'lambda', 'define', 'set!']

def evaluate(exp: Expression, env: Environment) -> Any:
    "Evaluate an expression in an environment."
    match exp:
        case int(x) | float(x):
            return x
        case Symbol(var):
            return env[var]
        case ['quote', x]:
            return x
        case ['if', test, consequence, alternative]:
            if evaluate(test, env):
                return evaluate(consequence, env)
            else:
                return evaluate(alternative, env)
        case ['lambda', [*parms], *body] if body:
            return Procedure(parms, body, env)
        case ['define', Symbol(name), value_exp]:
            env[name] = evaluate(value_exp, env)
        case ['define', [Symbol(name), *parms], *body] if body:
            env[name] = Procedure(parms, body, env)
        case ['set!', Symbol(name), value_exp]:
            env.change(name, evaluate(value_exp, env))
```

```

case [func_exp, *args] if func_exp not in KEYWORDS:
    proc = evaluate(func_exp, env)
    values = [evaluate(arg, env) for arg in args]
    return proc(*values)
case _:
    raise SyntaxError(lispstr(exp))

```

Vamos estudar cada cláusula case e o que cada uma faz. Em algumas casos coloquei comentários mostrando uma expressão-S que casaria com o padrão quando transformada em uma lista de Python. Os doctests extraídos de *examples_test.py* [fpy.li/18-21] demonstram cada case.

18.3.6.1. avaliando números

```

case int(x) | float(x):
    return x

```

Padrão:

Instância de int ou float.

Ação:

Devolve o próprio valor.

Exemplo:

```

>>> from lis import parse, evaluate, standard_env
>>> evaluate(parse('1.5'), {})
1.5

```

18.3.6.2. avaliando símbolos

```

case Symbol(var):
    return env[var]

```

Padrão:

Instância de Symbol, isto é, uma str usada como identificador.

Ação:

Consulta var em env e devolve seu valor.

Exemplos:

```
>>> evaluate(parse('+'), standard_env())
<built-in function add>
>>> evaluate(parse('ni!'), standard_env())
Traceback (most recent call last):
...
KeyError: 'ni!'
```

18.3.6.3. (quote ...)

A forma especial quote trata átomos e listas como dados em vez de expressões a serem avaliadas.

```
# (quote (99 bottles of beer))
case ['quote', x]:
    return x
```

Padrão:

Lista começando com o símbolo 'quote', seguido de uma expressão x.

Ação:

Devolve x sem avaliá-la.

Exemplos:

```
>>> evaluate(parse('(quote no-such-name)'), standard_env())
'no-such-name'
>>> evaluate(parse('(quote (99 bottles of beer))'), standard_env())
[99, 'bottles', 'of', 'beer']
>>> evaluate(parse('(quote (/ 10 0))'), standard_env())
['/', 10, 0]
```

Sem quote, cada expressão no teste geraria um erro:

- no-such-name seria buscado no ambiente, gerando um `KeyError`
- `(99 bottles of beer)` não pode ser avaliado, pois o número 99 não é um `Symbol` nomeando uma forma especial, um operador ou uma função
- `(/ 10 0)` geraria um `ZeroDivisionError`

Por que linguagens têm palavras reservadas?

Apesar de ser simples, `quote` não pode ser implementada como uma função em Scheme. Seu poder especial é impedir que o interpretador avalie `(f 10)` na expressão `(quote (f 10))`: o resultado é apenas uma lista com um `Symbol` e um `int`. Por outro lado, em uma chamada de função como `(abs (f 10))`, o interpretador primeiro calcula o resultado de `(f 10)` antes de invocar `abs`. Por isso `quote` é uma palavra reservada: ela precisa ser tratada de uma forma especial.

De modo geral, palavras reservadas são necessárias para:

- Introduzir regras especiais de avaliação, como `quote` e `lambda`—que não avaliam nenhuma de suas sub-expressões
- Mudar o fluxo de controle, como em `if` e chamadas de função—que também têm regras especiais de avaliação
- Para gerenciar o ambiente, como em `define` e `set`

Por isso também Python, e linguagens de programação em geral, precisam de palavras reservadas. Pense em `def`, `if`, `yield`, `import`, `del`, e o que elas fazem em Python.

18.3.6.4. (if ...)

```
# (if (< x 0) 0 x)
case ['if', test, consequence, alternative]:
    if evaluate(test, env):
        return evaluate(consequence, env)
    else:
        return evaluate(alternative, env)
```

Padrão:

Lista começando com 'if' seguida de três expressões: test, consequence, e alternative.

Ação:

Avalia test:

- Se verdadeira, avalia consequence e devolve seu valor.
- Caso contrário, avalia alternative e devolve seu valor.

Exemplos:

```
>>> evaluate(parse('(if (= 3 3) 1 0)'), standard_env())
1
>>> evaluate(parse('(if (= 3 4) 1 0)'), standard_env())
0
```

Os ramos consequence e alternative devem ser expressões simples. Se mais de uma expressão for necessária em um ramo, você pode combiná-las com (begin exp1 exp2...), fornecida como uma função em *lis.py*—veja o Exemplo 15.

18.3.6.5. (lambda ...)

A forma lambda do Scheme define funções anônimas. Ela não sofre das limitações da lambda de Python: qualquer função que pode ser escrita em Scheme pode ser escrita usando a sintaxe (lambda ...).

```
# (lambda (a b) (/ (+ a b) 2))
case ['lambda' [*parms], *body] if body:
    return Procedure(parms, body, env)
```

Padrão:

Lista começando com 'lambda', seguida de:

- Lista de zero ou mais nomes de parâmetros
- Uma ou mais expressões coletadas em body (a expressão guarda if body garante que body não pode ser vazio).

Ação:

Cria e devolve uma nova instância de `Procedure` com os nomes de parâmetros, a lista de expressões como o corpo da função, e o ambiente atual.

Exemplo:

```
>>> expr = '(lambda (a b) (* (/ a b) 100))'
>>> f = evaluate(parse(expr), standard_env())
>>> f # doctest: +ELLIPSIS
<lis.Procedure object at 0x...>
>>> f(15, 20)
75.0
```

A classe `Procedure` implementa o uma clausura (*closure*): um objeto invocável contendo nomes de parâmetros, um corpo de função, e uma referência ao ambiente no qual a `Procedure` está sendo declarada. Vamos estudar o código de `Procedure` daqui a pouco.

18.3.6.6. (define ...)

A palavra reservada `define` é usada em duas formas sintáticas diferentes. A mais simples é:

```
# (define half (/ 1 2))
case ['define', Symbol(name), value_exp]:
    env[name] = evaluate(value_exp, env)
```

Padrão:

Lista começando com `'define'`, seguido de um `Symbol` e uma expressão.

Ação:

Avalia a expressão e coloca o valor resultante em `env`, usando `name` como chave.

Exemplo:

```
>>> global_env = standard_env()
>>> evaluate(parse('(define answer (* 7 6))'), global_env)
>>> global_env['answer']
42
```

O doctest para este case cria um `global_env`, para podermos verificar que `evaluate` coloca `answer` dentro daquele `Environment`.

Podemos usar a primeira forma de `define` para criar variáveis ou para vincular nomes a funções anônimas, usando `(lambda ...)` como o `value_exp`.

A segunda forma de `define` é um atalho para definir funções nomeadas.

```
# (define (average a b) (/ (+ a b) 2))
case ['define', [Symbol(name), *parms], *body] if body:
    env[name] = Procedure(parms, body, env)
```

Padrão:

Lista começando com `'define'`, seguida de:

- Uma lista começando com um `Symbol(name)`, seguida de zero ou mais itens agrupados em uma lista chamada `parms`.
- Uma ou mais expressões agrupadas em `body` (a expressão guarda garante que `body` não esteja vazio)

Ação:

- Cria uma nova instância de `Procedure` com os nomes dos parâmetros, o corpo como uma lista de expressões, e o ambiente atual.
- Insere a `Procedure` em `env`, usando `name` como chave.

O doctest no Exemplo 18 define e coloca no `global_env` uma função chamada `%`, que calcula uma porcentagem.

Exemplo 18. Definindo uma função chamada %, que calcula uma porcentagem

```
>>> global_env = standard_env()
>>> percent = '(define (% a b) (* (/ a b) 100))'
>>> evaluate(parse(percent), global_env)
>>> global_env['%'] # doctest: +ELLIPSIS
<lis.Procedure object at 0x...>
>>> global_env['%'](170, 200)
85.0
```

Após invocar `evaluate`, verificamos que `%` está vinculada a uma `Procedure` que recebe dois argumentos numéricos e devolve uma porcentagem.

O padrão para o segundo `define` não obriga os itens em `parms` a serem todos instâncias de `Symbol`. Eu teria que verificar isso antes de criar a `Procedure`, mas não o fiz—para manter o código aqui tão fácil de acompanhar quanto o de Norvig.

18.3.6.7. (set! ...)

A forma `set!` muda o valor de uma variável previamente definida.^[11]

```
# (set! n (+ n 1))
case ['set!', Symbol(name), value_exp]:
    env.change(name, evaluate(value_exp, env))
```

Padrão:

Lista começando com `'set!'`, seguida de um `Symbol` e de uma expressão.

Ação:

Atualiza o valor de `name` em `env` com o resultado da avaliação da expressão.

O método `Environment.change` atravessa os ambientes encadeados de local para global, e atualiza a primeira ocorrência de `name` com o novo valor. Se não estivéssemos implementando a palavra reservada `'set!'`, esse interpretador poderia usar apenas o `ChainMap` de Python para implementar `env`, sem precisar da nossa classe `Environment`.

O nonlocal de Python e o set! do Scheme tratam da mesma questão

O uso da forma `set!` tem relação com a instrução `nonlocal` em Python: declarar `nonlocal x` permite a `x = 10` atualizar uma variável `x` anteriormente definida fora do escopo local. Sem a declaração `nonlocal x`, `x = 10` vai sempre criar uma variável local em Python, como vimos na «Seção 9.7» [fpy.li/cm] (vol.2).

De forma similar, `(set! x 10)` atualiza um `x` anteriormente definido que pode estar fora do ambiente local da função. Por outro lado, a variável `x` em `(define x 10)` será sempre uma variável local, criada ou atualizada no ambiente local.

Ambos, `nonlocal` e `(set! ...)`, são necessários para atualizar o estado do programa mantido em variáveis dentro de uma clausura. O Exemplo 13 do «Capítulo 9» [fpy.li/9] (vol.2) demonstrou o uso de `nonlocal` para implementar uma função que calcula uma média contínua, preservando os itens `count` e `total` em uma clausura. Aqui está a mesma ideia, escrita no subconjunto de Scheme de *lis.py*:

```
(define (make-averager)
  (define count 0)
  (define total 0)
  (lambda (new-value)
    (set! count (+ count 1))
    (set! total (+ total new-value))
    (/ total count))
)
(define avg (make-averager)) ①
(avg 10) ②
(avg 11) ③
(avg 15) ④
```

① Cria uma nova clausura com a função interna definida por `lambda` e as variáveis `count` e `total`, inicializadas com 0; vincula a clausura a `avg`.

② Devolve 10.0.

③ Devolve 10.5.

④ Devolve 12.0.

O código acima é um dos testes em *lispy/py3.10/examples_test.py* [fpy.li/18-18].

Agora chegamos a uma chamada de função.

18.3.6.8. Chamada de função

```
# (gcd (* 2 105) 84)
case [func_exp, *args] if func_exp not in KEYWORDS:
    proc = evaluate(func_exp, env)
    values = [evaluate(arg, env) for arg in args]
    return proc(*values)
```

Padrão:

Lista com um ou mais itens.

A expressão guarda garante que *func_exp* não é um de ['quote', 'if', 'define', 'lambda', 'set!']—listados logo antes de *evaluate* no Exemplo 17.

O padrão casa com qualquer lista com uma ou mais expressões, vinculando a primeira expressão a *func_exp* e o restante a *args* como uma lista, que pode ser vazia.

Ação:

- Avaliar *func_exp* para obter um Procedure que representa a função.
- Avaliar cada item em *args* para criar uma lista de valores dos argumentos.
- Invocar *proc* com os valores como argumentos separados, devolvendo o resultado.

Exemplo:

```
>>> evaluate(parse('( (* 12 14) (- 500 100)')), global_env)
42.0
```

Esse doctest continua do Exemplo 18: ele assume que `global_env` contém uma função chamada `%`. Os argumentos passados a `%` são expressões aritméticas, para enfatizar que eles são avaliados antes da função ser chamada.

A expressão guarda nesse `case` é necessária porque `[func_exp, *args]` casa com qualquer sujeito que seja uma sequência de um ou mais itens. Entretanto, se `func_exp` é uma palavra reservada e o sujeito não casou com nenhum dos `case` anteriores, então temos um erro de sintaxe, como `(define x)`.

18.3.6.9. Capturar erros de sintaxe

Se o sujeito `exp` não casa com nenhum dos `case` anteriores, o `case` "coringa" gera um `SyntaxError`:

```
case _:  
    raise SyntaxError(lispstr(exp))
```

Aqui está um exemplo de um `(lambda ...)` malformatado, identificado como um `SyntaxError`:

```
>>> evaluate(parse('(lambda is not like this)'), standard_env())  
Traceback (most recent call last):  
...  
SyntaxError: (lambda is not like this)
```

Se o `case` para chamada de função não tivesse aquela expressão guarda rejeitando palavras reservadas, a expressão `(lambda is not like this)` teria sido tratada como uma chamada de função, que geraria um `KeyError`, pois `'lambda'` faz parte do ambiente—assim como `lambda` em Python não é o nome de uma função embutida.

18.3.7. Procedure: uma classe que implementa uma clausura

A classe `Procedure` poderia muito bem se chamar `Closure`, porque é isso que ela representa: uma definição de função no contexto de um ambiente. A definição de função inclui o nome dos parâmetros e as expressões que formam o corpo da função. O ambiente é usado quando a função é chamada, para fornecer os

valores das *variáveis livres*: variáveis que aparecem no corpo da função, mas não são parâmetros, variáveis locais ou variáveis globais. Vimos os conceitos de *clausura* e de *variáveis livres* na «Seção 9.6» [fpy.li/c8] (vol.2).

Aprendemos como usar clausuras em Python, mas agora podemos mergulhar mais fundo e ver como uma clausura é implementada em *lis.py*:

```
class Procedure:
    "A user-defined Scheme procedure."

    def __init__( ①
        self,
        parms: list[Symbol],
        body: list[Expression],
        env: Environment
    ):
        self.parms = parms ②
        self.body = body
        self.env = env

    def __call__(self, *args: Expression) -> Any: ③
        local_env = dict(zip(self.parms, args)) ④
        env = Environment(local_env, self.env) ⑤
        for exp in self.body: ⑥
            result = evaluate(exp, env)
        return result ⑦
```

- ① `__init__` é invocado quando uma função é definida pelas instruções `lambda` ou `define`.
- ② Salva os nomes dos parâmetros, as expressões no corpo e o ambiente, para uso posterior.
- ③ `__call__` é invocado por `proc(*values)` na última linha da cláusula `case` [`func_exp`, `*args`].
- ④ Cria `local_env`, mapeando `self.parms` como nomes de variáveis locais e os `args` passados como valores.
- ⑤ Cria um novo `env` combinado, colocando `local_env` primeiro e então `self.env`—o ambiente salvo quando a função foi definida.

- ⑥ Itera sobre cada expressão em `self.body`, avaliando-as no `env` combinado.
- ⑦ Devolve o resultado da última expressão avaliada.

Há um par de funções simples após `evaluate` em `lis.py` [fpy.li/18-24]: `run` lê um programa Scheme completo e o executa, e `main` chama `run` ou `repl`, dependendo da linha de comando—parecido com o modo como Python faz. Não vou descrever essas funções, pois não há nada novo ali. Meus objetivos aqui eram compartilhar com vocês a beleza do pequeno interpretador de Norvig, explicar melhor como as clausuras funcionam, e mostrar como `match/case` foi uma ótima adição ao Python.

Para fechar essa seção estendida sobre casamento de padrões, vamos formalizar o conceito de um padrão-OU (*OR-pattern*).

18.3.8. Usando padrões-OU

Uma série de padrões separados por `|` formam um padrão-OU (documentado em *OR-patterns* [fpy.li/18-25]): ele casa se qualquer dos sub-padrões casar. Este é um padrão-OU que já vimos na Seção 18.3.6.1:

```
case int(x) | float(x):  
    return x
```

Todos os sub-padrões em um padrão-OU devem usar as mesmas variáveis. Esta restrição é necessária para garantir que as variáveis estejam disponíveis na expressão de guarda e no corpo do `case`, independentemente de qual sub-padrão tenha casado.



No contexto de uma cláusula `case`, o operador `|` tem um significado especial. Ele não aciona o método especial `__or__`, que manipula expressões como `a | b` em outros contextos, onde ele é sobrecarregado para realizar operações como união de conjuntos ou disjunção binária com inteiros (o "ou binário"), dependendo dos operandos.

Um padrão-OU não está limitado a aparecer no nível superior de um padrão. O símbolo `|` pode também ser usado em sub-padrões. Por exemplo, se quiséssemos

que o *lis.py* aceitasse a letra grega λ (lambda)^[12] além da palavra reservada `lambda`, poderíamos reescrever o padrão assim:

```
# (λ (a b) (/ (+ a b) 2) )
case ['lambda' | 'λ', [*parms], *body] if body:
    return Procedure(parms, body, env)
```

Agora podemos passar para o terceiro e último assunto deste capítulo: lugares incomuns onde a cláusula `else` pode aparecer no Python.

18.4. Faça isso, então aquilo: blocos `else` além do `if`

Não é segredo, mas é um recurso pouco conhecido em Python: a cláusula `else` pode ser usada não apenas com instruções `if`, mas também com as instruções `for`, `while`, e `try`.

A semântica para `for/else`, `while/else`, e `try/else` é semelhante, mas é muito diferente do `if/else`. No início, a palavra `else` atrapalhou meu entendimento destas cláusulas, mas acabei me acostumando.

Aqui estão as regras:

`for`

O bloco `else` será executado somente se e quando o laço `for` rodar até o iterável terminar (isto é, não rodará se o `for` for interrompido com um `break`).

`while`

O bloco `else` será executado somente se e quando o laço `while` terminar pela condição se tornar *falsa* (novamente, não rodará se o `while` for interrompido por um `break`)

`try`

O bloco `else` será executado somente se nenhuma exceção for gerada no bloco `try`. A «documentação oficial» [fpy.li/9x] também afirma: "Exceções na cláusula `else` não são tratadas pela cláusula `except` precedente."

Em todos os casos, a cláusula `else` também será ignorada se uma exceção ou uma instrução `return`, `break` ou `continue` fizer com que o fluxo de controle saia do bloco principal da instrução composta. No caso do `try`, esta é a diferença importante entre `else` e `finally`: o bloco `finally` será executado sempre, ocorrendo ou não uma exceção, e até mesmo se o fluxo de execução sair do bloco `try` por uma instrução como `return`.



Não tenho nada contra o funcionamento destas cláusulas `else`, mas do ponto de vista do design da linguagem, a palavra `else` foi uma escolha infeliz, porque `else` implica em uma alternativa excludente, como em "Execute esse laço, caso contrário faça aquilo." Mas o significado do `else` em laços é o oposto: "Execute esse laço, então faça aquilo." Isso sugere que `then` ("então") seria uma escolha melhor. Também faria sentido no contexto de um `try`: "Tente isso, então faça aquilo." Entretanto, acrescentar uma nova palavra reservada é uma ruptura séria em uma linguagem—uma decisão difícil. Guido sempre foi econômico com palavras reservadas.

Usar `else` com essas instruções muitas vezes torna o código mais fácil de ler e evita o transtorno de criar variáveis de controle ou codar instruções `if` adicionais.

O uso de `else` em laços em geral segue o padrão desse trecho:

```
for item in my_list:
    if item.flavor == 'banana':
        break
else:
    raise ValueError('No banana flavor found!')
```

No caso de blocos `try/except`, o `else` pode parecer redundante à primeira vista. Afinal, a `after_call()` no trecho a seguir só será invocada se a `dangerous_call()` não gerar uma exceção, correto?

```
try:
    dangerous_call()
after_call()
```

```
except OSError:
    log('OSError...')
```

Entretanto, não há um bom motivo para colocar a `after_call()` dentro do bloco `try`. Por clareza e correção, o corpo de um bloco `try` deveria conter apenas instruções que podem gerar as exceções esperadas. Assim fica melhor:

```
try:
    dangerous_call()
except OSError:
    log('OSError...')
else:
    after_call()
```

Agora fica explícito que o bloco `try` está de guarda contra possíveis erros na `dangerous_call()`, e não em `after_call()`. Também fica explícito que `after_call()` só será executada se nenhuma exceção for gerada no bloco `try`.

Em Python, `try/except` é bastante usado para controle de fluxo, não apenas para tratamento de erro. Há inclusive uma sigla/slogan para isso, documentado no «glossário oficial» [fpy.li/9y] do Python:

EAFP

Iniciais da expressão em inglês “easier to ask for forgiveness than permission” que significa “é mais fácil pedir perdão que permissão”. Este estilo de codificação comum em Python assume a existência de chaves ou atributos válidos e captura exceções caso essa premissa se prove falsa. Este estilo limpo e rápido se caracteriza pela presença de várias instruções `try` e `except`. A técnica diverge do estilo LBYL, comum em outras linguagens como C, por exemplo.

O glossário então define LBYL:

LBYL

Iniciais da expressão em inglês “look before you leap”, que significa algo como “olhe antes de pisar”. Este estilo de codificação testa as pré-condições explicitamente antes de fazer chamadas ou buscas. Este estilo

contrasta com a abordagem EAFP e é caracterizado pela presença de muitas instruções if. Em um ambiente multithread, a abordagem LBYL pode arriscar a introdução de uma condição de corrida entre “o olhar” e “o pisar”. Por exemplo, o código `if key in mapping: return mapping[key]` pode falhar se outra thread remover key do mapping após o teste (olhar), mas antes de acessar a chave (pisar). Este problema pode ser resolvido com bloqueios [travas] ou usando a abordagem EAFP.

Dado o estilo EAFP, faz mais sentido conhecer e usar os blocos `else` corretamente nas instruções `try/except`.



Quando a instrução `match` foi proposta, algumas pessoas (inclusive eu) acharam que ela também devia ter uma cláusula `else`. Afinal ficou decidido que isso não era necessário, pois `case _`: tem o mesmo efeito.^[13]

Agora vamos resumir o capítulo.

18.5. Resumo do capítulo

Este capítulo começou com o significado da instrução `with` e os gerenciadores de contexto, indo além do uso mais comum: fechar arquivos automaticamente. Implementamos um gerenciador de contexto customizado, a classe `LookingGlass`, usando os métodos `__enter__`/`__exit__`, e vimos como tratar exceções no método `__exit__`. Uma ideia fundamental apontada por Raymond Hettinger, na palestra de abertura da Pycon US 2013, é que `with` não serve apenas para gerenciamento de recursos; ele é uma ferramenta para fatorar código comum de configuração e de finalização, ou qualquer par de operações que precisem ser executadas antes e após outro procedimento.^[14]

Revisamos funções no módulo `contextlib` da biblioteca padrão. Uma delas, o decorador `@contextmanager`, permite implementar um gerenciador de contexto usando apenas um mero gerador com um `yield`—uma solução menos trabalhosa que criar uma classe com pelo menos dois métodos. Reimplementamos a `LookingGlass` como uma função geradora `looking_glass`, e discutimos como fazer tratamento de exceções usando o `@contextmanager`.

Então estudamos o elegante interpretador Scheme de Peter Norvig, o *lis.py*, escrito em Python idiomático e refatorado para usar `match/case` em `evaluate`—a função central de qualquer interpretador. Entender o funcionamento de `evaluate` exigiu revisar um pouco de Scheme, um parser para expressões-S, um REPL simples e a construção de escopos aninhados através de `Environment`, uma subclasse de `collection.ChainMap`. No fim, *lys.py* foi um instrumento para explorarmos mais que casamento de padrões. Ele mostra como diferentes partes de um interpretador trabalham juntas, ilustrando conceitos fundamentais do próprio Python: por que palavras reservadas são necessárias, como as regras de escopo funcionam, e como clausuras são criadas e usadas.

18.6. Para saber mais

O «Capítulo 8, Instruções Compostas» [fpy.li/9x] na *Referência da Linguagem Python* diz praticamente tudo que há para dizer sobre cláusulas `else` em instruções `if`, `for`, `while` e `try`. Sobre o uso pythônico de `try/except`, com ou sem `else`, Raymond Hettinger deu uma resposta brilhante para a pergunta *Is it a good practice to use try-except-else in Python?* (É uma boa prática usar `try-except-else` em Python?) [fpy.li/18-31] no StackOverflow. O *Python in a Nutshell*, 3rd ed. [fpy.li/pynut3], de Martelli et.al., tem um capítulo sobre exceções com uma excelente discussão sobre o estilo EAFP, atribuindo à pioneira da computação Grace Hopper a frase "É mais fácil pedir perdão que pedir permissão."

O capítulo 4 de *A Biblioteca Padrão de Python*, "Tipos Embutidos", tem uma seção dedicada a «Tipos de Gerenciador de Contexto» [fpy.li/9z]. Os métodos especiais `__enter__`/`__exit__` também estão documentados em *A Referência da Linguagem Python*, em «Gerenciadores de Contexto da Instrução `with`» [fpy.li/a4]. Os gerenciadores de contexto foram propostos na *PEP 343—The "with" Statement* [fpy.li/pep343].

Raymond Hettinger apontou a instrução `with` como um "recurso maravilhoso da linguagem" em sua «palestra de abertura» [fpy.li/18-29] da PyCon US 2013. Ele também mostrou alguns usos interessantes de gerenciadores de contexto em sua apresentação *_Transforming Code into Beautiful, Idiomatic Python+* [fpy.li/18-35] (Transformando Código em Python Lindo e Idiomático), na mesma conferência.

O post de Jeff Presling em seu blog, *The Python 'with' Statement by Example* [fpy.li/18-36] (A Instrução 'with' de Python através de exemplos) é interessante pelos exemplos de uso de gerenciadores de contexto com a biblioteca gráfica *pycairo*.

A classe `contextlib.ExitStack` foi baseada em uma ideia original de Nikolaus Rath, que escreveu um post curto explicando por que ela é útil: *On the Beauty of Python's ExitStack* [fpy.li/18-37] (Sobre a Beleza do ExitStack de Python). No texto, Rath argumenta que `ExitStack` é similar, mas mais flexível que a instrução `defer` em Go—que acho uma das melhores ideias naquela linguagem.

Beazley e Jones desenvolveram gerenciadores de contexto para propósitos muito diferentes em seu livro, *Python Cookbook, 3rd. ed.* [fpy.li/pycook3]. A *Recipe 8.3. Making Objects Support the Context-Management Protocol* (Fazendo Objetos Suportarem o Protocolo Gerenciador de Contexto) implementa uma classe `LazyConnection`, cujas instâncias são gerenciadores de contexto que abrem e fecham conexões de rede automaticamente, em blocos `with`. A *Recipe 9.22. Defining Context Managers the Easy Way* (O Jeito Fácil de Definir Gerenciadores de Contexto) apresenta um gerenciador de contexto para código de cronometragem, e outro para realizar mudanças transacionais em um objeto `list`: dentro do bloco `with` é criada uma cópia de trabalho da instância de `list`, e todas as mudanças são aplicadas àquela cópia de trabalho. Apenas quando o bloco `with` termina sem uma exceção a cópia de trabalho substitui a original. Simples e genial.

Peter Norvig descreve seu pequeno interpretador Scheme nos posts (*How to Write a (Lisp) Interpreter (in Python)*) [fpy.li/18-38] (Como Escrever um Interpretador (Lisp) (em Python)) e (*An Even Better (Lisp) Interpreter (in Python)*) [fpy.li/18-39] (Um Interpretador (Lisp) (Ainda Melhor)) (em Python)). O código-fonte de `lis.py` e `lispy.py` está no repositório *norvig/pytudes* [fpy.li/18-40]. Meu repositório, *fluentpython/lispy* [fpy.li/18-41], inclui a versão *mylis* do `lis.py`, atualizado para Python 3.10, com um REPL melhor, integração com a linha de comando, exemplos, mais testes e referências para aprender mais sobre Scheme. O melhor ambiente e dialeto de Scheme para explorar é o *Racket* [fpy.li/18-42].

Ponto de vista

Fatorando o pão

Em sua palestra de abertura na PyCon US 2013, *What Makes Python Awesome* [fpy.li/18-1] (O que torna Python incrível), Raymond Hettinger diz que quando viu a proposta da instrução `with`, pensou que era "um pouquinho misteriosa." Eu tive uma reação similar, inicialmente. As PEPs são muitas vezes difíceis de ler, e a PEP 343 é típica nesse sentido.

Mas aí—nos contou Hettinger—ele teve uma ideia: as sub-rotinas são a invenção mais importante na história das linguagens de computador. Se você tem sequências de operações, como `A;B;C` e `P;B;Q`, você pode fatorar `B` em uma sub-rotina. É como fatorar o recheio de um sanduíche: usar atum com tipos de diferentes de pão. Mas e se você quiser fatorar o pão, para fazer sanduíches com pão de trigo integral usando recheios diferentes a cada vez? É isso que a instrução `with` oferece. Ela é o complemento da sub-rotina. Hettinger continuou:

A instrução with é algo muito importante. Encorajo vocês a irem lá e olharem para a ponta desse iceberg, e daí cavarem mais fundo. Provavelmente é possível fazer coisas muito profundas com a instrução with. Seus melhores usos ainda estão por ser descobertos. Acredito que, se fizerem bom uso dela, ela será copiada em outras linguagens, e todas as linguagens futuras vão incluí-la. Vocês podem fazer parte da descoberta de algo quase tão profundo quanto a invenção da própria sub-rotina.

Hettinger admite que está forçando a "venda" da instrução `with`. Mesmo assim, é um recurso bem útil. Quando ele usou a analogia do sanduíche para explicar como `with` é o complemento da sub-rotina, muitas possibilidades se abriram na minha mente.

Se você precisa convencer alguém que de Python é sensacional, assista à palestra de abertura de Hettinger. A parte sobre gerenciadores de contexto fica entre 23:00 e 26:15. Mas a palestra inteira é excelente.

Recursão eficiente com chamadas de cauda apropriadas

As implementações padrão de Scheme são obrigadas a oferecer *chamadas de cauda apropriadas* (PTC, sigla de *Proper Tail Calls*), para tornar a iteração por recursão uma alternativa prática aos laços `while` e `for` das linguagens imperativas. Alguns autores se referem às PTC como *otimização de chamadas de cauda* (TCO, sigla de *Tail Call Optimization*); para outros, TCO é uma coisa diferente. Para mais detalhes, leia «Chamadas recursivas de cauda» [fpy.li/a2] na Wikipedia em português e «Tail call» [fpy.li/18-44], mais aprofundado, na Wikipedia em inglês, e «Tail call optimization in ECMAScript 6» [fpy.li/18-45].

Uma *chamada de cauda* é quando uma função devolve o resultado de uma chamada de função, que pode ou não ser a ela mesma (a função que está devolvendo o resultado). Os exemplos `gcd` no Exemplo 10 e no Exemplo 11 fazem chamadas de cauda (recursivas) no desvio *falso* do `if`.

Por outro lado, esta `factorial` não faz uma chamada de cauda:

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

A chamada para `factorial` na última linha não é uma chamada de cauda, pois o valor de `return` não é somente o resultado de uma chamada recursiva: o resultado é multiplicado por `n` antes de ser devolvido.

Aqui está uma alternativa que usa uma chamada de cauda, e é portanto recursiva de cauda (*tail recursive*):

```
def factorial_tc(n, product=1):  
    if n < 1:  
        return product  
    return factorial_tc(n - 1, product * n)
```

Python não tem PTC então não há vantagem em escrever funções recursivas de cauda. Neste caso, versão `factorial_tc`. Para usos na vida real, não se esqueça de que Python tem o `math.factorial`, escrito em C sem recursão. O ponto é que, mesmo em linguagens que implementam PTC, isso não beneficia toda função recursiva, apenas aquelas cuidadosamente escritas para fazer chamadas de cauda.

Se a linguagem implementa PTC, quando o interpretador vê uma chamada de cauda, ele pula para dentro do corpo da função chamada sem criar um novo stack frame, economizando memória. Há também linguagens compiladas que implementam PTC, por vezes como uma otimização que pode ser ligada e desligada.

Não existe um consenso universal sobre a definição de TCO ou sobre o valor das PTC em linguagens que não foram projetadas como linguagens funcionais desde o início, como Python e JavaScript. Em linguagens funcionais, PTC não é apenas uma otimização desejável. Se a linguagem não tem outro mecanismo de iteração além da recursão, então PTC é necessário para viabilizar o uso da linguagem na prática. O `lis.py` [fpy.li/18-46] de Norvig não implementa PTC, mas seu interpretador mais elaborado, o `lisp.py` [fpy.li/18-16], implementa.

Argumentos contra chamadas de cauda apropriadas em Python e JavaScript

O CPython não implementa PTC, e provavelmente nunca o fará. Guido van Rossum escreveu *Final Words on Tail Calls* [fpy.li/18-48] (Últimas Palavras sobre Chamadas de Cauda) para explicar o motivo. Resumindo, aqui está uma passagem fundamental de seu post:

Pessoalmente, acho que é um bom recurso para algumas linguagens, mas não acho que se encaixe no Python: a eliminação dos registros do stack para algumas chamadas mas não para outras certamente confundiria muitos usuários, que não foram criados na religião das chamadas de cauda, mas podem ter aprendido sobre a semântica das chamadas rastreando algumas chamadas em um depurador.

Em 2015, PTC foram incluídas no padrão ECMAScript 6 para JavaScript. Em outubro de 2021 o interpretador do «WebKit as implementa» [fpy.li/18-49]. O WebKit é usado pelo Safari. Os interpretadores JS em todos os outros navegadores populares não têm PTC, assim como o Node.js, que depende do interpretador V8 que o Google mantém para o Chrome. Transpiladores e *polyfills* (injetores de código) voltados para o JS, como o TypeScript, o ClojureScript e o Babel, também não suportam PTC, de acordo com uma «tabela de compatibilidade com ECMAScript 6» [fpy.li/18-50].

Já vi várias explicações para a rejeição das PTC por parte dos implementadores, mas a mais comum é a mesma que Guido van Rossum mencionou: PTC tornam a depuração mais difícil para todo mundo, e beneficiam apenas uma minoria que prefere usar recursão para fazer iteração. Para mais detalhes, veja *What happened to proper tail calls in JavaScript?* [fpy.li/18-51] (O que aconteceu com as chamadas de cauda apropriadas em JavaScript?) de Graham Marlow.

Há casos em que a recursão é a melhor solução, mesmo no Python sem PTC. Em um post anterior [fpy.li/18-52] sobre o assunto, Guido escreveu:

[...] uma implementação típica de Python permite 1000 recursões, o que é bastante para código não-recursivo e para código que usa recursão para percorrer, por exemplo, uma árvore de parsing típica, mas não o bastante para um laço escrito de forma recursiva sobre uma lista grande.

Concordo com Guido e com a maioria dos implementadores de JavaScript.

A falta de PTC é a maior restrição ao desenvolvimento de programas Python em um estilo funcional—mais que a sintaxe limitada de `lambda`.

Se você estiver curioso em ver como PTC funciona em um interpretador com menos recursos (e menos código) que o *lisper.py* de Norvig, veja o *mylis_2* [fpy.li/18-53]. O truque começa com o laço infinito em `evaluate` e o código no `case` que faz chamadas de função: essa combinação faz o interpretador pular para dentro do corpo da próxima `Procedure` sem invocar `evaluate` recursivamente durante a chamada de cauda.

Estes pequenos interpretadores demonstram o poder da abstração: apesar de Python não implementar PTC, é possível e não muito difícil escrever um interpretador, em Python, que implementa PTC. Aprendi a fazer isso lendo o código de Peter Norvig. Obrigado por compartilhar, professor!

A opinião de Norvig sobre `evaluate()` com casamento de padrões

Mostrei o código da versão Python 3.10 de *lis.py* para Peter Norvig. Ele gostou do exemplo usando casamento de padrões, mas sugeriu uma solução diferente: em vez de usar as guardas que escrevi, ele teria exatamente um `case` por palavra reservada, e teria testes dentro de cada `case`, para fornecer mensagens de `SyntaxError` mais específicas—por exemplo, quando o corpo estiver vazio.

Se eu fizesse assim, a expressão guarda nesta linha seria desnecessária:

```
case [func_exp, *args] if func_exp not in KEYWORDS:
```

A guarda seria redundante pois todas as palavras reservadas já teriam sido tratadas antes do `case` para chamadas de função.

Provavelmente seguirei o conselho do professor Norvig quando acrescentar funcionalidades ao *mylis* [fpy.li/18-54].

Mas a forma como estruturei `evaluate` no Exemplo 17 tem algumas vantagens didáticas neste livro:

- o exemplo é paralelo à implementação original de Norvig com `if/elif/...` no Exemplo 11 do «Capítulo 2» [fpy.li/2] (vol.1);
- as cláusulas `case` demonstram mais recursos de casamento de padrões;
- o código ficou mais legível, na minha opinião.

[1] Palestra de abertura da PyCon US 2013: "What Makes Python Awesome" ("O que torna Python incrível") [fpy.li/18-1]; a parte sobre `with` começa em 23:00 e termina em 26:15.

[2] Os três argumentos recebidos por `self` são exatamente o que você obtém se chama `sys.exc_info()` [fpy.li/18-7] no bloco `finally` de uma instrução `try/finally`. Isso faz sentido, considerando que a instrução `with` tem por objetivo substituir a maioria dos usos de `try/finally`, e invocar `sys.exc_info()` é muitas vezes necessário para determinar que ação de limpeza é necessária.

[3] A classe real se chama `_GeneratorContextManager`. Se você quiser saber exatamente como ela funciona, leia seu código-fonte [fpy.li/18-10] na `Lib/contextlib.py` de Python 3.10.

[4] Essa dica é uma citação literal de um comentário de Leonardo Rochael, um dos revisores técnicos desse livro. Muito bem colocado, Leo!

[5] "Pouco conhecido" porque pelo menos eu e os outros revisores técnicos não sabíamos disso até Caleb Hattingh nos contar. Obrigado, Caleb!

[6] As pessoas reclamam sobre o excesso de parênteses no Lisp, mas um bom editor e a indentação consistente do código praticamente resolvem essa questão. O maior problema de legibilidade é o uso da mesma notação (`f ...`) para invocar funções e aplicar formas especiais como (`define ...`), (`if ...`) e (`quote ...`), que têm comportamentos muito diferentes de qualquer função.

[7] Para que a iteração por recursão seja prática e eficiente, o Scheme e outras linguagens funcionais otimizam certas chamadas recursivas. Para ler mais sobre isso, veja o Ponto de vista.

[8] Mas o segundo interpretador de Norvig, `lisp.py` [fpy.li/18-16], suporta strings como um tipo de dado, e também traz recursos avançados como macros sintáticas, continuações, e chamadas de cauda otimizadas. Entretanto, o `lisp.py` é quase três vezes maior que o `lis.py`—é mais difícil de entender.

[9] O comentário `# type: ignore[index]` está ali por causa do issue #6042 [fpy.li/18-19] no `typeshed`, que segue sem resolução quando esse capítulo está sendo revisado. `ChainMap` é anotado como `MutableMapping`, mas a dica de tipo no atributo `maps` diz que ele é uma lista de `Mapping`, indiretamente tornando todo o `ChainMap` imutável até onde o Mypy entende.

[10] Enquanto estudava o `lis.py` e o `lisp.py` de Norvig, comecei uma versão chamada `mylis` [fpy.li/18-20], que acrescenta alguns recursos, incluindo um REPL que aceita expressões-S parciais e espera a continuação, como o REPL do Python que sabe quando não terminamos uma instrução e apresenta um prompt secundário (...) até entrarmos uma instrução completa, que possa ser analisada e avaliada. O `mylis` também trata alguns erros de forma graciosa, mas ele ainda é fácil de quebrar. Não é nem de longe tão robusto quanto o REPL do Python.

[11] A atribuição é um dos primeiros recursos ensinados em muitos tutoriais de programação, mas set! só aparece na página 220 do mais conhecido livro de Scheme, *Structure and Interpretation of Computer Programs* (A Estrutura e a Interpretação de Programas de Computador), 2nd ed., [fpy.li/18-22] de Abelson et al. (MIT Press), também conhecido como SICP ou *Wizard Book* (Livro do Mago). Programas em estilo funcional podem nos levar muito longe sem as mudanças de estado típicas da programação imperativa e da programação orientada a objetos.

[12] O nome para λ (U+03BB) no Unicode é GREEK SMALL LETTER LAMDA. Isso não é um erro ortográfico: o caractere é chamado "lamda" sem o "b" no banco de dados do Unicode. De acordo com o artigo "Lambda" [fpy.li/18-26] da Wikipédia em inglês, o Unicode Consortium adotou essa ortografia em função de "preferências expressas pela Autoridade Nacional Grega."

[13] Acompanhando a discussão na lista python-dev, achei que um motivo para a rejeição do `else` foi a falta de consenso sobre como indentá-lo dentro do `match`: o `else` deveria ser indentado no mesmo nível do `match` ou no mesmo nível do `case`?

[14] Veja o slide 21 em *Python is Awesome* (Python é Incrível) [fpy.li/18-29].

Capítulo 19. Modelos de concorrência em Python

Concorrência é lidar com muitas coisas ao mesmo tempo.

Paralelismo é fazer muitas coisas ao mesmo tempo.

Não são iguais, mas têm relação.

[Concorrência] é sobre estrutura, [paralelismo] é sobre execução.

A concorrência fornece uma maneira de estruturar uma solução para resolver um problema que pode (mas não necessariamente) ser paralelizado.^[1]

— Rob Pike, Co-criador da linguagem Go

Este capítulo é sobre como fazer Python "lidar com muitas coisas ao mesmo tempo." Isso pode envolver programação concorrente ou paralela. Até mesmo acadêmicos discordam sobre o uso destas palavras. Vou adotar as definições informais de Rob Pike, na epígrafe acima, mas encontrei artigos e livros que dizem ser sobre computação paralela, mas são quase que inteiramente sobre concorrência.^[2]

Na perspectiva de Pike, o paralelismo é, um caso especial de concorrência. Todo sistema paralelo é concorrente, mas nem todo sistema concorrente é paralelo. No início dos anos 2000, usávamos laptops GNU Linux de um único núcleo, que rodavam 100 processos ao mesmo tempo. Um laptop moderno com quatro núcleos de CPU rotineiramente está executando mais de 200 processos a qualquer momento, sob uso normal, casual. Para executar 200 tarefas em paralelo, você precisaria de 200 núcleos. Portanto, na prática, a maior parte da computação em nosso cotidiano é concorrente e não paralela. O SO administra centenas de processos, assegurando que cada um tenha a oportunidade de progredir, mesmo quando a CPU em si não roda mais que quatro tarefas em paralelo.

Este capítulo não assume que você tenha conhecimento prévio de programação concorrente ou paralela. Após uma breve introdução conceitual, vamos estudar exemplos simples, para apresentar e comparar os principais pacotes da biblioteca padrão de Python dedicados à programação concorrente: `threading`, `multiprocessing`, e `asyncio`.

O último terço do capítulo é uma revisão geral de ferramentas, servidores de aplicação e filas de tarefas distribuídas (*distributed task queues*) de vários fornecedores, capazes de melhorar o desempenho e a escalabilidade de aplicações Python. Todos esses são tópicos importantes, mas fogem do escopo de um livro focado nos recursos fundamentais da linguagem Python. Mesmo assim, achei importante mencionar estes temas nesta segunda edição do *Python Fluente*, porque a aptidão de Python para computação concorrente e paralela não está limitada ao que a biblioteca padrão oferece. Por isso YouTube, DropBox, Instagram, Reddit e outros foram capazes de atingir alta escalabilidade quando começaram, usando Python como sua linguagem primária—apesar das persistentes alegações de que "Python não escala."

19.1. Novidades neste capítulo

Este capítulo é novo, escrito para a segunda edição do *Python Fluente*. Os exemplos com os caracteres giratórios na Seção 19.4 antes estavam no capítulo sobre `asyncio`. Aqui eles foram revisados, e apresentam uma primeira ilustração das três abordagens de Python à concorrência: threads, processos e corrotinas nativas.

O resto do conteúdo é novo, exceto por alguns parágrafos, que apareciam originalmente nos capítulos sobre `concurrent.futures` e `asyncio`.

A Seção 19.7 é diferente do resto do livro: não há código exemplo. O objetivo ali é apresentar brevemente ferramentas importantes, que você pode querer estudar para conseguir concorrência e paralelismo de alto desempenho, para além do que é possível com a biblioteca padrão de Python.

Nota sobre o cenário em 2026

Contratualmente, esta tradução precisa seguir o conteúdo do *Fluent Python, Second Edition*, que publiquei pela O'Reilly em 2022.



Pesquisei e escrevi este capítulo em 2021, quando a versão mais recente do Python era a 3.10. Desde então, novas versões do Python têm trazido melhorias importantes para a programação concorrente, inclusive novas formas de contornar a GIL.

Além disso, o ecossistema de desenvolvimento para novas aplicações Web hoje é dominado pelas soluções de provedores de nuvem, como AWS, que oferecem substitutos para gerenciadores de fila como *Celery* e novas arquiteturas para execução concorrente diferentes dos servidores de aplicação como *uWSGI* e *Gunicorn*.

Mais do que qualquer outro capítulo no livro, este precisaria de muitas atualizações para refletir o cenário em 2026, mas os princípios e conceitos fundamentais continuam válidos, especialmente para o desenvolvimento de sistemas *on premise*, independentes de um provedor de nuvem.

19.2. A visão geral

Há muitos fatores que tornam a programação concorrente difícil, mas quero tocar no mais básico deles: iniciar threads ou processos é fácil, mas como administrá-los?^[3]

Quando você invoca uma função, o código que faz a chamada aguarda até que a função retorne. Então você sabe que a função terminou, e pode facilmente acessar o valor devolvido por ela. Se a função lançar uma exceção, o código cliente pode cercar aquela chamada com um bloco `try/except` para tratar o erro.

Tais opções não existem quando você inicia threads ou um processo: você não sabe automaticamente quando eles terminaram, e obter os resultados ou os erros requer algum canal de comunicação que você precisa fornecer, como uma fila de mensagens (*message queue*).

Além disso, criar uma thread ou um processo tem um custo, você não quer iniciar um deles apenas para executar uma única computação e encerrar. Muitas vezes queremos amortizar o custo de inicialização transformando cada thread ou processo em um *worker* ou "unidade de trabalho", que entra em um laço e espera por dados para processar. Isso complica ainda mais a comunicação e introduz mais questões. Como terminar um "worker" quando ele não é mais necessário? E como fazer para encerrá-lo sem interromper uma tarefa inacabada, deixando dados inconsistentes e recursos não liberados—tal como arquivos abertos? A resposta envolve novamente filas e mensagens.

Uma corrotina é fácil de iniciar. Se você inicia uma corrotina usando a palavra-chave `await`, é fácil obter o valor de retorno e há um local óbvio para tratar exceções. Mas corrotinas muitas vezes são iniciadas pelo framework assíncrono, e isso pode torná-las tão difíceis de monitorar quanto `threads` ou processos.

Por fim, as corrotinas e `threads` de Python não são adequadas para tarefas de uso intensivo da CPU, como veremos.

Programação concorrente envolve conceitos e modelos de programação que podem ser novidade para você. Então vamos primeiro garantir que estamos na mesma página em relação a alguns conceitos centrais.

19.3. Um pouco de jargão

Aqui estão alguns termos que usaremos pelo restante deste capítulo e nos dois seguintes:

Concorrência

A capacidade de lidar com múltiplas tarefas pendentes, fazendo progredir uma por vez ou várias em paralelo (se possível), de forma que cada uma delas avance até terminar com sucesso ou falhar. Uma CPU de um núcleo é capaz de concorrência se rodar um *scheduler* (escalonador) do sistema operacional, que intercale a execução das tarefas pendentes. Esta capacidade também é conhecida como multitarefa (*multitasking*).

Paralelismo

A habilidade de executar múltiplas operações computacionais ao mesmo tempo. Isso requer uma CPU com múltiplos núcleos, múltiplas CPUs, uma GPU [fpy.li/19-2], ou múltiplos computadores em um *cluster* (agrupamento).

Unidades de execução

Termo genérico para objetos que executam código de forma concorrente, cada um com um estado e uma pilha de chamada independentes. Python suporta de forma nativa três tipos de unidade de execução: *processos*, *threads*, e *corrotinas*.

Processo

Uma instância de um programa de computador em execução, usando parte da memória e uma fatia do tempo da CPU. Os sistemas operacionais modernos em nossos computadores e celulares rodam rotineiramente centenas de processos de forma concorrente, cada um deles isolado em seu próprio espaço de memória privado. Processos se comunicam via *pipes*, soquetes ou arquivos mapeados na memória (*memory mapped files*). Todos esses métodos só comportam bytes em estado bruto. Objetos Python precisam ser serializados (convertidos em sequências de bytes) para passarem de um processo a outro. Isto é caro, e nem todos os objetos Python podem ser serializados. Um processo pode gerar subprocessos, chamados "processos filhos". Estes também rodam isolados entre si e do processo original. Os processos permitem *multitarefa preemptiva*: o agendador do sistema operacional exerce *preempção*—isto é, suspende cada processo em execução periodicamente, para permitir que outros processos sejam executados. Isto significa que um processo travado não pode travar todo o sistema—em teoria.

Thread

Uma unidade de execução dentro de um processo. Quando um processo se inicia, ele tem uma única thread: a thread principal. Um processo pode chamar APIs do sistema operacional para criar mais threads para operar de forma concorrente. Threads dentro de um processo compartilham o mesmo espaço de memória, onde são mantidos objetos Python "vivos" (não serializados). Isso facilita o compartilhamento de informações entre threads, mas pode também levar à corrupção de dados, se uma thread está lendo um objeto enquanto ele está sendo modificado por outra thread. Como os processos, as threads também possibilitam a *multitarefa preemptiva* sob a supervisão do agendador do SO. Uma thread consome menos recursos que um processo para realizar a mesma tarefa.

Corrotina

Uma função que pode suspender sua própria execução e continuar depois. Em Python, corrotinas clássicas são criadas a partir de funções geradoras, e corrotinas nativas são definidas com `async def`. A Seção 17.13 introduziu o conceito, e o Capítulo 21 trata do uso de corrotinas nativas. As corrotinas de Python normalmente rodam dentro de uma única thread, sob a supervisão de um laço de eventos (*event loop*), também na mesma thread. Frameworks de programação assíncrona como *asyncio*, *Curio*, ou *Trio* fornecem um laço de

eventos e bibliotecas de apoio para E/S não-bloqueante baseado em corrotinas. Corrotinas permitem *multitarefa cooperativa*: cada corrotina deve ceder explicitamente o controle com as palavras-chave `yield` ou `await`, para que outra possa continuar de forma concorrente (mas não em paralelo). Isso significa que qualquer código bloqueante em uma corrotina bloqueia a execução do laço de eventos e de todas as outras corrotinas—ao contrário da *multitarefa preemptiva* suportada por processos e threads. Por outro lado, cada corrotina consome menos recursos para executar o mesmo trabalho que uma thread ou processo.

Fila (*queue*)

Uma estrutura de dados que nos permite adicionar e retirar itens, normalmente na ordem FIFO: o primeiro que entra é o primeiro que sai.^[4] Filas permitem que unidades de execução separadas troquem dados da aplicação e mensagens de controle, como códigos de erro e sinais de término. A implementação de uma fila varia de acordo com o modelo de concorrência subjacente: o pacote `queue` na biblioteca padrão de Python fornece classes de fila para suportar threads, já os pacotes `multiprocessing` e `asyncio` implementam suas próprias classes de fila. Os pacotes `queue` e `asyncio` também incluem filas não FIFO: `LifoQueue` e `PriorityQueue`.

Trava (*lock*)

Um objeto que as unidades de execução podem usar para sincronizar suas ações e evitar corrupção de dados. Ao atualizar uma estrutura de dados compartilhada, o código em execução deve invocar uma função para obter uma trava associada a tal estrutura. Isso sinaliza a outras partes do programa que elas devem aguardar até que a trava seja liberada, antes de acessar a mesma estrutura de dados. A variante mais simples de trava é conhecida também como mutex (de *mutual exclusion*, exclusão mútua). O mecanismo para implementar uma trava depende do modelo de concorrência subjacente.

Contenda (*contention*)

Disputa por um recurso limitado. Contenda por recursos ocorre quando múltiplas unidades de execução tentam acessar um recurso compartilhado—tal como uma trava ou unidade de armazenamento. Há também contenda pela CPU, quando processos ou threads de computação intensiva precisam aguardar até que o agendador do SO dê a eles uma quota do tempo da CPU.

Agora vamos usar um pouco desse jargão para entender o suporte à concorrência no Python.

19.3.1. Processos, threads, e a infame GIL de Python

Veja como os conceitos que acabamos de tratar se aplicam ao Python, em dez pontos:

1. Cada instância do interpretador Python é um processo. Você pode iniciar processos Python adicionais usando as bibliotecas `multiprocessing` ou `concurrent.futures`. A biblioteca `subprocess` de Python foi projetada para rodar programas externos escritos em qualquer linguagem.
2. O interpretador Python usa uma única thread para rodar o programa do usuário e o coletor de lixo da memória. Você pode iniciar threads Python adicionais usando as bibliotecas `threading` ou `concurrent.futures`.
3. O acesso à contagem de referências a objetos e outros estados internos do interpretador é controlado por uma trava, a Global Interpreter Lock (GIL) ou *Trava Global do Interpretador*. A qualquer dado momento, apenas uma thread de Python pode reter a trava. Isso significa que apenas uma thread pode executar código Python a cada momento, mesmo que a CPU tenha vários núcleos.
4. Para evitar que uma thread de Python segure a GIL indefinidamente, o interpretador de bytecode de Python pausa a thread Python corrente a cada 5ms por default, liberando a GIL.^[5] A thread pode então tentar readquirir a GIL, mas se existirem outras threads esperando, o agendador do SO pode escolher uma delas para continuar.
5. Quando escrevemos código Python, não temos controle sobre a GIL. Mas uma função embutida ou uma extensão escrita em C—ou qualquer linguagem que trabalhe no nível da API Python/C—pode liberar a GIL enquanto estiver rodando alguma tarefa demorada.
6. Toda função na biblioteca padrão de Python que executa uma `syscall` libera a GIL^[6]. Isto inclui todas as funções que executam operações de escrita e leitura de arquivos, escrita e leitura na rede, e `time.sleep()`. Muitas funções de uso intensivo da CPU nas bibliotecas NumPy/SciPy, bem como as funções de compressão e descompressão dos módulos `zlib` e `bz2`, também liberam a GIL.^[7]

7. Extensões binárias que se comunicam via API Python/C também podem iniciar outras threads não-Python, que não são afetadas pela GIL. Essas threads fora do controle da GIL normalmente não podem modificar objetos Python, mas podem ler e escrever na memória usada por objetos que suportam o buffer protocol [fpy.li/pep3118], como `bytearray`, `array.array`, e arrays do *NumPy*.
8. O efeito da GIL sobre a programação de redes com threads Python é relativamente pequeno, porque as funções de E/S liberam a GIL, e ler e escrever na rede sempre implica em alta latência—comparado a ler e escrever na memória. Consequentemente, cada thread individual já passa muito tempo esperando mesmo, então sua execução pode ser intercalada sem maiores impactos no desempenho geral. Por isso David Beazley diz: "As threads de Python são ótimas em fazer nada."^[8]
9. As contendas pela GIL desaceleram as threads Python que fazem processamento intensivo. Código sequencial de uma única thread é mais simples e mais rápido para este tipo de tarefa.
10. Para rodar código Python de uso intensivo da CPU em múltiplos núcleos, você precisa usar múltiplos processos Python.

Aqui está um bom resumo, parte da documentação do módulo `threading` [fpy.li/a7]:

Detalhe de implementação do CPython: *Em CPython, devido à Trava Global do Interpretador, apenas uma thread pode executar código Python de cada vez (mas certas bibliotecas de alto desempenho podem contornar esta limitação). Se você quer que sua aplicação faça melhor uso dos recursos computacionais de máquinas com CPUs de múltiplos núcleos, aconselha-se usar `multiprocessing` ou `concurrent.futures.ProcessPoolExecutor`.*

Entretanto, threads ainda são o modelo adequado se você deseja rodar múltiplas tarefas ligadas a E/S simultaneamente.

O parágrafo anterior começa com "Detalhe de implementação do CPython" porque a GIL não é parte da definição da linguagem Python. As implementações Jython e o IronPython não têm uma GIL. Infelizmente, ambas estão ficando para trás, ainda compatíveis apenas com Python 2.7 e 3.4, respectivamente. O interpretador de alto desempenho PyPy [fpy.li/19-9] também tem uma GIL em suas versões 2.7, 3.8 e 3.9 (a mais recente em março de 2021).



Esta seção não mencionou corrotinas, por que por default elas compartilham a mesma thread Python entre si e com o laço de eventos supervisor fornecido por um framework assíncrono—então não são afetadas pela GIL. É possível usar múltiplas threads em um programa assíncrono, mas a melhor prática é ter uma thread rodando o laço de eventos e todas as corrotinas, enquanto as threads adicionais executam tarefas específicas. Isso será explicado na Seção 21.8.

Mas chega de conceitos por agora. Vamos ver algum código.

19.4. Um "Olá mundo" concorrente

Durante uma discussão sobre threads e sobre como evitar a GIL, o contribuidor do Python Michele Simionato postou um exemplo [fpy.li/19-10] que é praticamente um "Olá Mundo" concorrente: o programa mais simples possível mostrando como o Python pode "assobiar e chupar cana ao mesmo tempo".

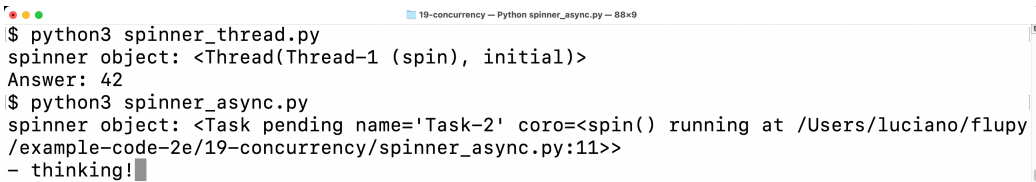
O programa de Simionato usa multiprocessing, mas eu o adaptei para apresentar também threading e asyncio. Vamos começar com a versão threading, que pode parecer familiar se você já estudou threads em Java ou C.

19.4.1. Caracteres animados com threads

A ideia dos próximos exemplos é simples: iniciar uma função que pausa por 3 segundos enquanto anima caracteres no terminal, para deixar o usuário saber que o programa está "pensando" e não congelado.

O script cria uma animação giratória mostrando em sequência cada caractere da string '\|/-' na mesma posição da tela.^[9] Quando a computação lenta termina, a linha com a animação é apagada e o resultado é apresentado: Answer: 42.

Figura 1 mostra a saída de duas versões do exemplo: primeiro com threads, depois com corrotinas. Se você estiver longe do computador, imagine que o hífen (-) na última linha está girando.



```
$ python3 spinner_thread.py
spinner object: <Thread(Thread-1 (spin), initial)>
Answer: 42
$ python3 spinner_async.py
spinner object: <Task pending name='Task-2' coro=<spin() running at /Users/luciano/flupy
/example-code-2e/19-concurrency/spinner_async.py:11>>
- thinking!
```

Figura 1. Os scripts `spinner_thread.py` e `spinner_async.py` produzem um resultado similar: o repr do objeto `spinner` e o texto "Answer: 42". Na captura de tela, `spinner_async.py` ainda está rodando, e a mensagem animada "- thinking!" é apresentada; aquela linha será substituída por "Answer: 42" após 3 segundos.

Vamos estudar o script `spinner_thread.py` primeiro. O Exemplo 1 lista as duas primeiras funções no script, e o Exemplo 2 mostra o restante.

Exemplo 1. `spinner_thread.py`: as funções `spin` e `slow`

```
import itertools
import time
from threading import Thread, Event

def spin(msg: str, done: Event) -> None: ①
    for char in itertools.cycle(r'\|/-'): ②
        status = f'\r{char} {msg}' ③
        print(status, end='', flush=True)
        if done.wait(.1): ④
            break ⑤
    blanks = ' ' * len(status)
    print(f'\r{blanks}\r', end='') ⑥

def slow() -> int:
    time.sleep(3) ⑦
    return 42
```

- ① Esta função vai rodar em uma thread separada. O argumento `done` é uma instância de `threading.Event`, um objeto simples para sincronizar threads.
- ② Isto é um laço infinito, porque `itertools.cycle` produz um caractere por vez, circulando pela string para sempre.
- ③ O truque para animação em modo texto: mova o cursor de volta para o início da linha com o caractere ASCII *carriage return*: `'\r'`.
- ④ O método `Event.wait(timeout=None)` retorna `True` quando o evento é sinalizado

por outra thread; se o `timeout` passou, ele retorna `False`. O tempo de 0,1s estabelece a velocidade da animação em 10 FPS (quadros por segundo). Se quiser uma animação mais rápida, use um tempo menor aqui.

- ⑤ Sai do laço infinito.
- ⑥ Sobrescreve a linha de status com espaços para limpá-la e move o cursor de volta para o início.
- ⑦ `slow()` será chamada pela thread principal. Imagine que isso é uma chamada de API lenta, através da rede. Chamar `sleep` bloqueia a thread principal, mas a GIL é liberada e a thread da animação pode continuar.



O primeiro detalhe importante deste exemplo é que `time.sleep()` bloqueia a thread que a chama, mas libera a GIL, permitindo que outras threads Python rodem.

As funções `spin` e `slow` serão executadas de forma concorrente. A thread principal—a única thread quando o programa é iniciado—vai iniciar uma nova thread para rodar `spin` e então chamará `slow`. propositalmente, não existe API para terminar uma thread em Python. É preciso enviar algum sinal para encerrar uma thread.

A classe `threading.Event` é o mecanismo de sinalização para coordenar threads mais simples no Python. Uma instância de `Event` tem um atributo booleano interno que começa como `False`. Uma chamada a `Event.set()` muda o atributo para `True`. Enquanto o atributo for falso, se uma thread chamar `Event.wait()`, ela será bloqueada até que outra thread chame `Event.set()`. Então a próxima invocação de `Event.wait()` retornará `True`, sem esperar. Se um tempo de espera (*timeout*) em segundos é passado para `Event.wait(s)`, essa chamada retorna `False` quando aquele tempo tiver passado, ou retorna `True` assim que `Event.set()` é chamado por outra thread.

A função `supervisor`, que aparece no Exemplo 2, usa um `Event` para sinalizar para a função `spin` que ela deve encerrar.

Exemplo 2. spinner_thread.py: as funções supervisor e main

```
def supervisor() -> int: ①
    done = Event() ②
    spinner = Thread(target=spin, args=('thinking!', done)) ③
    print(f'spinner object: {spinner}') ④
    spinner.start() ⑤
    result = slow() ⑥
    done.set() ⑦
    spinner.join() ⑧
    return result

def main() -> None:
    result = supervisor() ⑨
    print(f'Answer: {result}')

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

- ① supervisor retornará o resultado de slow.
- ② A instância de `threading.Event` é a chave para coordenar as atividades das threads `main` e `spinner`, como explicado abaixo.
- ③ Para criar uma nova `Thread`, forneça uma função como nomeado `target`, e argumentos posicionais para a `target` como uma tupla passada via `args`.
- ④ Mostra o objeto `spinner`. A saída é `<Thread(Thread-1, initial)>`, onde `initial` é o estado da thread—significando aqui que ela ainda não foi iniciada.
- ⑤ Inicia a thread `spinner`.
- ⑥ Chama `slow`, que bloqueia a thread principal. Enquanto isso, a thread secundária está rodando a animação.
- ⑦ Muda o estado de `Event` para `True`; isto vai encerrar o laço `for` em `spin`.
- ⑧ Espera até que a thread `spinner` termine.
- ⑨ Roda a função `supervisor`. Escrevi `main` e `supervisor` como funções separadas para deixar esse exemplo mais parecido com a versão `asyncio` no Exemplo 4.

Quando a thread `main` sinaliza o evento `done`, a thread `spinner` acabará notando e encerrará corretamente.

Agora vamos ver um exemplo similar usando o pacote `multiprocessing`.

19.4.2. Animação com processos

O pacote `multiprocessing` permite executar tarefas concorrentes em processos Python separados em vez de threads. Quando você cria uma instância de `multiprocessing.Process`, todo um novo interpretador Python é iniciado como um processo filho, em segundo plano. Como cada processo Python tem sua própria GIL, isto permite que seu programa use todos os núcleos de CPU disponíveis—mas isso depende, em última instância, do agendador do sistema operacional. Veremos os efeitos práticos na Seção 19.6, mas para este programa simples não faz grande diferença.

O objetivo dessa seção é apresentar o `multiprocessing` e mostrar como sua API emula a API de `threading`, facilitando a conversão de programas simples de threads para processos, como mostra o *spinner_proc.py* (Exemplo 3).

Exemplo 3. spinner_proc.py: apenas as partes modificadas são mostradas; todo o resto é idêntico a spinner_thread.py

```
import itertools
import time
from multiprocessing import Process, Event ①
from multiprocessing import synchronize ②

def spin(msg: str, done: synchronize.Event) -> None: ③

# as funções spin, slow e main são iguais a spinner_thread.py

def supervisor() -> int:
    done = Event()
    spinner = Process(target=spin, ④
                      args=('thinking!', done))
    print(f'spinner object: {spinner}') ⑤
    spinner.start()
    result = slow()
    done.set()
    spinner.join()
    return result
```

- ① A API básica de multiprocessing imita a API de threading, mas as dicas de tipo e o Mypy revelam esta diferença: `multiprocessing.Event` é uma função (e não uma classe como `threading.Event`) que retorna uma instância de `synchronize.Event`...
- ② ...nos obrigando a importar `multiprocessing.synchronize`...
- ③ ...para escrever essa dica de tipo.
- ④ O uso básico da classe `Process` é similar ao da classe `Thread`.
- ⑤ O objeto `spinner` aparece como `<Process name='Process-1' parent=14868 initial>`, onde 14868 é o id do processo filho: a outra instância de Python que está executando o `spinner_proc.py`.

As APIs básicas de threading e multiprocessing são similares, mas sua implementação é muito diferente, e multiprocessing tem uma API muito maior, para dar conta da complexidade adicional da programação multiprocessos. Por exemplo, um dos desafios ao converter um programa de threads para processos é a comunicação entre processos, que são isolados pelo sistema operacional e não podem compartilhar objetos Python. Isso significa que objetos cruzando fronteiras entre processos precisam ser serializados e deserializados, criando custos adicionais. No Exemplo 3, o único dado que cruza a fronteira entre os processos é o estado de Event, implementado com um semáforo de baixo nível do SO, no código em C sob o módulo multiprocessing.^[10]



Desde o Python 3.8, existe o pacote `multiprocessing.shared_memory` [fpy.li/a8] na biblioteca padrão, mas ele não suporta instâncias de classes definidas pelo usuário. Além de bytes puros, o pacote permite que processos compartilhem uma `ShareableList`, uma sequência mutável que pode manter um número fixo de itens dos tipos `int`, `float`, `bool`, e `None`, bem como `str` e `bytes`, até o limite de 10 MB por item. Veja a documentação de `ShareableList` [fpy.li/a9] para mais detalhes.

Agora vamos ver como o mesmo comportamento pode ser obtido com corrotinas em vez de threads ou processos.

19.4.3. Animação com corrotinas



O Capítulo 21 é inteiramente dedicado à programação assíncrona com corrotinas. Esta seção é apenas uma introdução rápida, para contrastar esta abordagem com as threads e os processos. Por isso, vamos passar por cima de alguns detalhes.

Alocar tempo da CPU para a execução de threads e processos é trabalho dos agendadores do SO. As corrotinas, por outro lado, são controladas por um laço de evento no nível da aplicação, que gerencia uma fila de corrotinas pendentes, as executa uma por vez, monitora eventos disparados por operações de E/S iniciadas pelas corrotinas, e passa o controle de volta para a corrotina correspondente quando cada evento acontece. O laço de eventos, as corrotinas da biblioteca, e as corrotinas do usuário rodam todas em uma única thread. Assim, o tempo gasto em uma corrotina bloqueia o laço de eventos e todas as outras corrotinas.

A versão com corrotinas do programa de animação é mais fácil de entender se começarmos por uma função `main`, e depois olharmos a supervisor. É isso que o Exemplo 4 mostra.

Exemplo 4. `spinner_async.py`: a função `main` e a corrotina `supervisor`

```
def main() -> None: ①
    result = asyncio.run(supervisor()) ②
    print(f'Answer: {result}')

async def supervisor() -> int: ③
    spinner = asyncio.create_task(spin('thinking!')) ④
    print(f'spinner object: {spinner}') ⑤
    result = await slow() ⑥
    spinner.cancel() ⑦
    return result

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

① `main` é a única função normal definida nesse programa—as outras são corrotinas.

- ② A função `asyncio.run` inicia o laço de eventos para acionar a corrotina que em algum momento colocará as outras corrotinas em movimento. A função `main` ficará bloqueada até que `supervisor` retorne. O valor devolvido por `supervisor` será o valor devolvido por `asyncio.run`.
- ③ Corrotinas nativas são definidas com `async def`.
- ④ `asyncio.create_task` agenda a execução futura de `spin`, retornando imediatamente uma instância de `asyncio.Task`.
- ⑤ O repr do objeto `spinner` se parece com `<Task pending name='Task-2' coro=<spin() running at /path/to/spinner_async.py:11>>`.
- ⑥ A palavra-chave `await` chama `slow`, bloqueando `supervisor` até que `slow` retorne. O devolvido por `slow` é atribuído a `result`.
- ⑦ O método `Task.cancel` lança uma exceção `CancelledError` dentro da corrotina, como veremos no Exemplo 5.

O Exemplo 4 demonstra as três principais formas de rodar uma corrotina:

`asyncio.run(coro())`

É invocada a partir de uma função normal, para acionar o objeto corrotina, que é o ponto de entrada para todo o código assíncrono no programa, como a `supervisor` neste exemplo. Esta chamada bloqueia até que `coro` retorne. O resultado de `coro` será o resultado de `run`.

`asyncio.create_task(coro())`

É invocada dentro de uma corrotina para agendar a execução futura de outra corrotina. Essa chamada não suspende a corrotina atual. Ela retorna imediatamente uma instância de `Task`, um objeto que contém o objeto corrotina e fornece métodos para controlar e consultar seu estado.

`await coro()`

É invocada dentro de uma corrotina para transferir o controle para o objeto corrotina retornado por `coro()`. Isto suspende a corrotina atual até que `coro` retorne. O valor da expressão `await` será o que quer que `coro` devolva como resultado.



Lembre-se: invocar uma corrotina como `coro()` retorna imediatamente um objeto corrotina, mas não executa o corpo da função `coro`. Acionar o corpo de corrotinas é a função do laço de eventos.

Vamos estudar agora as corrotinas `spin` e `slow` no Exemplo 5.

Exemplo 5. `spinner_async.py`: as corrotinas `spin` e `slow`

```
import asyncio
import itertools

async def spin(msg: str) -> None: ①
    for char in itertools.cycle(r'\|/-'):
        status = f'\r{char} {msg}'
        print(status, flush=True, end='')
        try:
            await asyncio.sleep(.1) ②
        except asyncio.CancelledError: ③
            break
    blanks = ' ' * len(status)
    print(f'\r{blanks}\r', end='')

async def slow() -> int:
    await asyncio.sleep(3) ④
    return 42
```

- ① Não precisamos do argumento `Event`, que era usado para sinalizar que `slow` havia terminado de rodar no `spinner_thread.py` (Exemplo 1).
- ② Use `await asyncio.sleep(.1)` em vez de `time.sleep(.1)`, para pausar sem bloquear outras corrotinas. Veja o experimento após o exemplo.
- ③ `asyncio.CancelledError` é lançada quando o método `cancel` é chamado na `Task` que controla essa corrotina. É hora de sair do laço.
- ④ A corrotina `slow` também usa `await asyncio.sleep` em vez de `time.sleep`.

19.4.3.1. Experimento: quebrar a animação para revelar um fato

Aqui está um experimento que recomendo para entender como *spinner_async.py* funciona. Importe o módulo `time`, daí vá até a corrotina `slow` e substitua a linha `await asyncio.sleep(3)` por uma chamada a `time.sleep(3)`, como no Exemplo 6.

Exemplo 6. spinner_async.py: substituindo `await asyncio.sleep(3)` por `time.sleep(3)`

```
async def slow() -> int:
    time.sleep(3)
    return 42
```

Observar o comportamento é mais memorável que ler sobre ele. Vai lá, eu espero.

Ao rodar o experimento, você vê o seguinte:

1. O objeto `spinner` aparece: `<Task pending name='Task-2' coro=<spin() running at .../spinner_async.py:12>>`.
2. A animação nunca aparece. O programa trava por 3 segundos.
3. Answer: 42 aparece e o programa termina.

Para entender o que está acontecendo, lembre-se de que o código Python que está usando `asyncio` tem apenas uma unidade de execução, a menos que você inicie explicitamente threads ou processos adicionais. Isso significa que apenas uma corrotina é executada a qualquer dado momento. A concorrência é obtida controlando a passagem de uma corrotina a outra. No Exemplo 7, vamos nos concentrar no que ocorre nas corrotinas `supervisor` e `slow` durante o experimento proposto.

Exemplo 7. spinner_async_experiment.py: as corrotinas `supervisor` e `slow`

```
async def slow() -> int:
    time.sleep(3) ④
    return 42

async def supervisor() -> int:
    spinner = asyncio.create_task(spin('thinking!')) ①
```



```
print(f'spinner object: {spinner}') ②
result = await slow() ③
spinner.cancel() ⑤
return result
```

- ① A tarefa `spinner` é criada para, no futuro, acionar a corrotina `spin`.
- ② O display mostra que Task está *pending* (pendente, em espera).
- ③ A expressão `await` transfere o controle para a corrotina `slow`.
- ④ `time.sleep(3)` bloqueia tudo por 3 segundos; nada pode acontecer no programa, porque a thread principal está bloqueada—e ela é a única thread. O sistema operacional vai seguir com outras atividades. Após 3 segundos, `sleep` desbloqueia, e `slow` retorna.
- ⑤ Logo após `slow` retornar, a tarefa `spinner` é cancelada. O corpo da corrotina `spin` nunca foi acionado.

O `spinner_async_experiment.py` ensina uma lição importante, como explicado no box abaixo.



Nunca use `time.sleep(...)` em corrotinas assíncronas, a menos que você queira pausar o programa inteiro. Se uma corrotina precisa passar algum tempo sem fazer nada, use `await asyncio.sleep(DELAY)`. Isto devolve o controle para o laço de eventos do `asyncio`, que pode acionar outras corrotinas pendentes.

Greenlet e gevent

Ao discutir concorrência com corrotinas, vale mencionar o pacote *greenlet* [fpy.li/19-14], que já existe há muitos anos e é muito usado.^[11] O pacote suporta multitarefa cooperativa através de corrotinas leves—chamadas *greenlets*—que não exigem qualquer sintaxe especial tal como `yield` ou `await`, e assim são mais fáceis de integrar a bases de código sequencial existentes. O SQLAlchemy 1.4 ORM [fpy.li/19-15] usa *greenlets* internamente para implementar sua nova API assíncrona [fpy.li/19-16] compatível com `asyncio`.

A biblioteca de programação de redes *gevent* [fpy.li/19-17] modifica o módulo `socket` padrão de Python via *monkey patching*, tornando-o não-bloqueante ao substituir parte do código por `greenlets`. Na maior parte dos casos, *gevent* é transparente para o código em seu entorno, tornando mais fácil adaptar aplicações e bibliotecas sequenciais—tal como drivers de bancos de dados—para executar E/S de rede de forma concorrente. Inúmeros projetos open source [fpy.li/19-18] usam *gevent*, incluindo o muito usado *Gunicorn* [fpy.li/gunicorn]—mencionado na Seção 19.7.4.

19.4.4. Supervisores lado a lado

O número de linhas de *spinner_thread.py* e *spinner_async.py* é quase o mesmo. As funções supervisor são a parte mais importante destes exemplos. Vamos compará-las em detalhes.

Exemplo 8. spinner_thread.py: a função supervisor com threads

```
def supervisor() -> int:
    done = Event()
    spinner = Thread(target=spin,
                     args=('thinking!', done))
    print('spinner object:', spinner)
    spinner.start()
    result = slow()
    done.set()
    spinner.join()
    return result
```

Para comparar, o Exemplo 9 mostra a corrotina supervisor do Exemplo 4.

Exemplo 9. spinner_async.py: a corrotina assíncrona supervisor

```
async def supervisor() -> int:
    spinner = asyncio.create_task(spin('thinking!'))
    print('spinner object:', spinner)
    result = await slow()
    spinner.cancel()
    return result
```

Aqui está um resumo das diferenças e semelhanças notáveis entre as duas implementações de supervisor:

- Uma `asyncio.Task` é aproximadamente equivalente a `threading.Thread`.
- Uma `Task` aciona um objeto corrotina, e uma `Thread` invoca um *callable*.
- Uma corrotina passa o controle explicitamente com a palavra-chave `await`.
- Você não instancia objetos `Task` diretamente, eles são obtidos passando uma corrotina para `asyncio.create_task(...)`.
- Quando `asyncio.create_task(...)` devolve um objeto `Task`, ele já está agendado para rodar, mas uma instância de `Thread` precisa ser iniciada explicitamente através de uma chamada a seu método `start`.
- Na supervisor da versão com threads, `slow` é uma função comum e é invocada diretamente pela thread principal. Na versão assíncrona da supervisor, `slow` é uma corrotina acionada por `await`.
- Não existe um método para terminar uma thread externamente; em vez disso, é preciso enviar um sinal—como invocar `set` no objeto `Event`. Objetos `Task` oferecem o método `.cancel()`, que levantará um `CancelledError` na expressão `await` onde a corrotina está suspensa naquele momento.
- A corrotina supervisor é acionada com `asyncio.run` na função `main`.

Esta comparação ajuda a entender como a concorrência é orquestrada com `asyncio`, em contraste com como isso é feito com o módulo `threading`, que pode ser mais familiar para quem já usou threads em qualquer linguagem.

Um último ponto relativo a threads versus corrotinas: quem já escreveu qualquer programa não-trivial com threads sabe quão desafiador é estruturar o programa, porque o agendador pode interromper uma thread a qualquer momento. É preciso lembrar de manter travas para proteger seções críticas do programa, para evitar ser interrompido no meio de uma operação de muitas etapas—algo que poderia deixar dados em um estado inválido.

Com corrotinas, seu código está protegido de interrupções arbitrárias. É preciso chamar `await` explicitamente para deixar o resto do programa rodar. Em vez de manter travas para sincronizar as operações de múltiplas threads, corrotinas são "sincronizadas" por definição: apenas uma delas está rodando em qualquer momento. Para entregar o controle, você usa `await` para passar o controle de

volta ao agendador. Por isso é possível cancelar uma corrotina de forma segura: por definição, uma corrotina só pode ser cancelada quando está suspensa em uma expressão `await`, então é possível realizar qualquer limpeza necessária capturando a exceção `CancelledError` naquele ponto da corrotina.

A chamada `time.sleep()` bloqueia mas não faz nada. Vamos agora experimentar com uma função intensiva em CPU, para entender melhor a GIL, bem como o efeito de funções de processamento intensivo sobre código assíncrono.

19.5. O verdadeiro impacto da GIL

Na versão com `threads` (Exemplo 1), você pode trocar a chamada `time.sleep(3)` na função `slow` por uma requisição de cliente HTTP de sua biblioteca favorita, e a animação continuará girando. Isso acontece porque qualquer boa biblioteca de programação para rede vai liberar a GIL enquanto estiver esperando uma resposta. Por padrão, toda operação de E/S em Python libera a GIL.

Você também pode trocar a expressão `asyncio.sleep(3)` na corrotina `slow` para fazer `await` esperar a resposta de uma corrotina de biblioteca bem desenhada de acesso assíncrono à rede. Tais bibliotecas implementam corrotinas para devolver o controle para o laço de eventos enquanto esperam por uma resposta da rede. Enquanto isso, a animação seguirá girando.

Com código de uso intensivo da CPU, a história é outra. Considere a função `is_prime` no Exemplo 10, que retorna `True` se o argumento for um número primo, `False` se não for.

Exemplo 10. `primes.py`: uma checagem de números primos fácil de entender, do exemplo em `ProcessPoolExecutor` [fpy.li/aa] na documentação de Python]

```
def is_prime(n: int) -> bool:
    if n < 2:
        return False
    if n == 2:
        return True
    if n % 2 == 0:
        return False

    root = math.isqrt(n)
    for i in range(3, root + 1, 2):
```

```
if n % i == 0:
    return False
return True
```

A chamada `is_prime(5_000_111_000_222_021)` leva cerca de 3s no laptop da empresa que estou usando agora.^[12]

19.5.1. Teste Rápido

Dado o que vimos até aqui, pare um instante para pensar sobre a seguinte questão, de três partes. Uma das partes da resposta é um pouco mais complicada (pelo menos para mim foi).

O que aconteceria à animação após as seguintes modificações, presumindo que `n = 5_000_111_000_222_021`—aquele número primo que minha máquina levou 3s para checar:

1. Em `spinner_proc.py`, substitua `time.sleep(3)` por uma chamada a `is_prime(n)`?
2. Em `spinner_thread.py`, substitua `time.sleep(3)` por uma chamada a `is_prime(n)`?
3. Em `spinner_async.py`, substitua `await asyncio.sleep(3)` por uma chamada a `is_prime(n)`?

Antes de executar o código ou continuar lendo, recomendo dedicar um tempo para considerar as perguntas e formular suas respostas. Depois, copie, modifique e rode os exemplos *spinner* como sugerido.

Agora as respostas, da mais fácil para a mais difícil.

Resposta para multiprocessamento (*spinner_proc.py*)

A animação é controlada por um processo filho, então continua girando enquanto o teste de números primos é computado no processo raiz.^[13]

Resposta para *asyncio* (*spinner_async.py*)

Se você chamar `is_prime(5_000_111_000_222_021)` na corrotina `slow` do exemplo *spinner_async.py*, a animação nunca vai aparecer. O efeito é o que

vimos no Exemplo 6, quando substituímos `await asyncio.sleep(3)` por `time.sleep(3)`: nenhuma animação. O fluxo de controle vai passar da supervisor para `slow`, e então para `is_prime`. Quando `is_prime` retornar, `slow` vai retornar também, e supervisor retomará a execução, cancelando a tarefa `spinner` antes dela ser executada sequer uma vez. O programa parecerá congelado por aproximadamente 3s, e então mostrará a resposta.

Resposta para threads (*spinner_thread.py*)

A animação é controlada por uma thread secundária, então continua girando enquanto o teste de número primo é computado na thread principal.

Não acertei essa resposta inicialmente: eu esperava que a animação congelasse, porque superestimei o impacto da GIL.

Neste exemplo em particular, a animação segue girando porque Python suspende a thread em execução a cada 5ms (por default), tornando a GIL disponível para outras threads pendentes. Assim, a thread principal executando `is_prime` é interrompida a cada 5ms, permitindo à thread secundária acordar e executar uma vez o laço `for`, até chamar o método `wait` do evento `done`, quando então ela liberará a GIL. A thread principal então pegará a GIL, e o cálculo de `is_prime` continuará por mais 5 ms.

Isso não tem um impacto visível no tempo de execução deste exemplo específico, porque a função `spin` rapidamente realiza uma iteração e libera a GIL, enquanto espera pelo evento `done`, então não há muita contenda pela GIL. A thread principal executando `is_prime` terá a GIL na maior parte do tempo.

Conseguimos nos safar usando threads para uma tarefa de processamento intensivo nesse experimento simples porque só temos duas threads: uma ocupando a CPU, e a outra acordando apenas 10 vezes por segundo para atualizar a animação.

Mas se você tiver duas ou mais threads disputando mais tempo da CPU, seu programa será mais lento que um programa sequencial.

Power nap (soneca rápida) com `sleep(0)`

Um jeito de manter a animação funcionando é reescrever `is_prime` como uma corrotina, e periodicamente chamar `asyncio.sleep(0)` em uma expressão `await`, para passar o controle de volta para o laço de eventos, como no Exemplo 11.

Exemplo 11. `spinner_async_nap.py`: `is_prime` agora é uma corrotina

```
async def is_prime(n):
    if n < 2:
        return False
    if n == 2:
        return True
    if n % 2 == 0:
        return False

    root = math.isqrt(n)
    for i in range(3, root + 1, 2):
        if n % i == 0:
            return False
        if i % 100_000 == 1:
            await asyncio.sleep(0) ①
    return True
```

① Dormir a cada 50.000 iterações (porque o argumento `step` em `range` é 2).

O *Issue #284* [[fpy.li/19-20](https://github.com/pytest-dev/pytest/issues/284)] no repositório do `asyncio` trata do uso de `asyncio.sleep(0)`.

Entretanto, observe que isso vai tornar `is_prime` mais lento, e—mais importante—vai também atrasar o laço de eventos e tornar o programa inteiro mais lento. Quando usei `await asyncio.sleep(0)` a cada 100.000 iterações, a animação foi suave mas o programa rodou por 4,9s na minha máquina, quase 50% a mais que a função `primes.is_prime` rodando sozinha com o mesmo argumento (`5_000_111_000_222_021`).

Usar `await asyncio.sleep(0)` pode ser uma medida paliativa até o código assíncrono ser refatorado para delegar computações de uso intensivo da CPU para outro processo. Veremos como fazer isso com o `asyncio.loop.run_in_executor` [fpy.li/19-21], abordado no Capítulo 21. Outra opção seria uma fila de tarefas, que vamos discutir brevemente na Seção 19.7.5.

Até aqui experimentamos com uma única chamada para uma função de uso intensivo de CPU. A próxima seção apresenta a execução concorrente de múltiplas chamadas de uso intensivo da CPU.

19.6. Um pool de processos caseiro

Não conheço boas traduções para *process pool*. Uma alternativa seria "um banco de processos". A ideia é que vários processos são iniciados e ficam aguardando tarefas. Este padrão amortiza o custo de subir um processo para cada tarefa.



Escrevi esta seção para mostrar o uso de um banco de processos em cenários de uso intensivo de CPU, com filas para distribuir tarefas para os processos, e coletar os resultados. O Capítulo 20 apresenta uma forma mais simples de distribuir tarefas para processos: o `ProcessPoolExecutor` do pacote `concurrent.futures`, que também usa filas, mas elas não são visíveis para o usuário.

Nesta seção vamos escrever programas para checar se os números dentro de uma amostra de 20 inteiros são primos. Os números variam de 2 até 9.999.999.999.999.999—isto é, $10^{16} - 1$, ou mais de 2^{53} . A amostra inclui números primos pequenos e grandes, bem como números compostos com fatores primos pequenos e grandes.

O programa *sequential.py* fornece a linha base de desempenho. Aqui está o resultado de uma execução de teste:


```
$ python3 sequential.py
      2 P 0.000001s
142702110479723 P 0.568328s
299593572317531 P 0.796773s
333333333333301 P 2.648625s
333333333333333 0.000007s
3333335652092209 2.672323s
444444444444423 P 3.052667s
444444444444444 0.000001s
444444448888889 3.061083s
5555553133149889 3.451833s
555555555555503 P 3.556867s
555555555555555 0.000007s
666666666666666 0.000001s
6666666666666719 P 3.781064s
6666667141414921 3.778166s
7777777536340681 4.120069s
777777777777753 P 4.141530s
777777777777777 0.000007s
999999999999917 P 4.678164s
999999999999999 0.000007s
Total time: 40.31
```

Os resultados aparecem em três colunas:

- O número a ser checado.
- P se é um número primo, vazia se é um número composto.
- Tempo decorrido para checar se aquele número é primo.

Neste exemplo, o tempo total é aproximadamente a soma do tempo de cada checagem, mas está computado separadamente, como se vê no Exemplo 12.

Exemplo 12. sequential.py: checagem de números primos em um pequeno conjunto de dados

```
#!/usr/bin/env python3

"""
sequential.py: baseline for comparing sequential, multiprocessing,
and threading code for CPU-intensive work.
```

```

"""

from time import perf_counter
from typing import NamedTuple

from primes import is_prime, NUMBERS

class Result(NamedTuple): ①
    prime: bool
    elapsed: float

def check(n: int) -> Result: ②
    t0 = perf_counter()
    prime = is_prime(n)
    return Result(prime, perf_counter() - t0)

def main() -> None:
    print(f'Checking {len(NUMBERS)} numbers sequentially:')
    t0 = perf_counter()
    for n in NUMBERS: ③
        prime, elapsed = check(n)
        label = 'P' if prime else ' '
        print(f'{n:16} {label} {elapsed:9.6f}s')

    elapsed = perf_counter() - t0 ④
    print(f'Total time: {elapsed:.2f}s')

if __name__ == '__main__':
    main()

```

- ① A função check (logo abaixo) devolve uma tupla Result com o valor booleano da chamada a is_prime e o tempo decorrido.
- ② check(n) chama is_prime(n) e calcula o tempo decorrido para retornar um Result.
- ③ Para cada número na amostra, chamamos check e apresentamos o resultado.
- ④ Calcula e mostra o tempo total decorrido.

19.6.1. Solução baseada em processos

O próximo exemplo, *procs.py*, mostra o uso de múltiplos processos para distribuir a checagem de números primos por muitos núcleos da CPU. Esses são os tempos obtidos com *procs.py*:

```
$ python3 procs.py
Checking 20 numbers with 12 processes:
      2 P 0.000002s
33333333333333333333 0.000021s
44444444444444444444 0.000002s
55555555555555555555 0.000018s
66666666666666666666 0.000002s
142702110479723 P 1.350982s
77777777777777777777 0.000009s
299593572317531 P 1.981411s
99999999999999999999 0.000008s
33333333333333301 P 6.328173s
3333335652092209 6.419249s
4444444488888889 7.051267s
44444444444444423 P 7.122004s
5555553133149889 7.412735s
55555555555555503 P 7.603327s
66666666666666719 P 7.934670s
6666667141414921 8.017599s
777777536340681 8.339623s
7777777777777753 P 8.38859s
9999999999999917 P 8.117313s
20 checks in 9.58s
```

A última linha dos resultados mostra que *procs.py* foi 4,2 vezes mais rápido que *sequential.py*.

19.6.2. Entendendo os tempos decorridos

Observe que o tempo decorrido na primeira coluna é o tempo para checar aquele número específico. Por exemplo, `is_prime(777777777777753)` demorou quase 8,4s para retornar `True`. Enquanto isso, outros processos estavam checando outros números em paralelo.

Há 20 números para serem checados. Escrevi *procs.py* para iniciar um número de processos de trabalho igual ao número de núcleos na CPU, como determinado por `multiprocessing.cpu_count()`.

O tempo total neste caso é muito menor que a soma dos tempos decorridos para cada checagem individual. Há algum tempo gasto em iniciar processos e na comunicação entre processos, então o resultado final é que a versão multiprocessos é apenas cerca de 4,2 vezes mais rápida que a sequencial. Isso é bom, mas um pouco desapontador, considerando que o código inicia 12 processos, para usar todos os núcleos desse laptop.



A função `multiprocessing.cpu_count()` retorna 12 no MacBook Pro que usei para escrever este capítulo. Ele é um i7 com uma CPU de 6 núcleos, mas o SO informa 12 CPUs devido ao *hyperthreading*, uma tecnologia da Intel que executa duas threads por núcleo. Entretanto, *hyperthreading* funciona melhor quando uma das threads não está trabalhando tão pesado quanto a outra thread no mesmo núcleo—talvez a primeira esteja parada, esperando por dados após a invalidação do cache, enquanto a outra está mastigando números. De qualquer forma, não existe almoço grátis: este laptop tem o desempenho de uma máquina com 6 CPUs para atividades de processamento intensivo com pouco uso de memória, como este exemplo.

19.6.3. Código do chegador de primos usando múltiplos núcleos

Quando delegamos processamento para threads e processos, nosso código não invoca diretamente a função que realiza o trabalho, então não conseguimos simplesmente devolver um resultado com `return`. Em vez disso, a função de trabalho é acionada pela biblioteca de threads ou processos, e produz um resultado que precisa ser armazenado em algum lugar. Coordenar threads ou processos de trabalho e coletar resultados são usos comuns de filas em programação concorrente, e também em sistemas distribuídos.

Muito do código novo em *procs.py* se refere a configurar e usar filas. O início do arquivo está no Exemplo 13.

Exemplo 13. procs.py: checagem de primos com múltiplos processos; importações, tipos, e funções

```
import sys
from time import perf_counter
from typing import NamedTuple
from multiprocessing import Process, SimpleQueue, cpu_count ①
from multiprocessing import queues ②

from primes import is_prime, NUMBERS

class PrimeResult(NamedTuple): ③
    n: int
    prime: bool
    elapsed: float

JobQueue = queues.SimpleQueue[int] ④
ResultQueue = queues.SimpleQueue[PrimeResult] ⑤

def check(n: int) -> PrimeResult: ⑥
    t0 = perf_counter()
    res = is_prime(n)
    return PrimeResult(n, res, perf_counter() - t0)

def worker(jobs: JobQueue, results: ResultQueue) -> None: ⑦
    while n := jobs.get(): ⑧
        results.put(check(n)) ⑨
        results.put(PrimeResult(0, False, 0.0)) ⑩

def start_jobs(
    procs: int, jobs: JobQueue, results: ResultQueue ⑪
) -> None:
    for n in NUMBERS:
        jobs.put(n) ⑫
    for _ in range(procs):
        proc = Process(target=worker, args=(jobs, results)) ⑬
        proc.start() ⑭
        jobs.put(0) ⑮
```

- ① Na tentativa de emular threading, multiprocessing fornece multiprocessing.SimpleQueue, mas esse é um método vinculado a uma

instância pré-definida de uma classe de nível mais baixo, `BaseContext`. Temos que chamar essa `SimpleQueue` para criar uma fila. Mas não podemos usá-la em dicas de tipo.

- ② `multiprocessing.queues` contém a classe `SimpleQueue` que precisamos para dicas de tipo.
- ③ `PrimeResult` inclui o número checado. Preservar `n` com os outros campos do resultado simplifica a exibição mais tarde.
- ④ Isso é um apelido de tipo para uma `SimpleQueue` que a função `main` (Exemplo 14) vai usar para enviar os números para os processos que farão a checagem.
- ⑤ Apelido de tipo para uma segunda `SimpleQueue` que vai coletar os resultados em `main`. Os valores na fila serão tuplas contendo o número a ser testado e uma tupla `Result`.
- ⑥ Isso é similar a *`sequential.py`*.
- ⑦ `worker` recebe uma fila com os números a serem checados, e outra para colocar os resultados.
- ⑧ Nesse código, usei o número 0 como um sinal para que o processo encerre. Se `n` não é 0, o laço continua.^[14]
- ⑨ Invoca a checagem de número primo e coloca o `PrimeResult` na fila.
- ⑩ Devolve um `PrimeResult(0, False, 0.0)`, para informar ao laço principal que esse processo terminou seu trabalho.
- ⑪ `procs` é o número de processos que executarão a checagem de números primos em paralelo.
- ⑫ Coloca na fila ``jobs`` todos os números a serem checados.
- ⑬ Cria um processo filho para cada `worker`. Cada um desses processos executará o laço dentro de sua própria instância da função `worker`, até encontrar um 0 na fila `jobs`.
- ⑭ Inicia cada processo filho.
- ⑮ Coloca um 0 na fila para cada processo, para encerrá-los.

Laços, sentinelas e pílulas venenosas

A função `worker` no Exemplo 13 segue um modelo comum em programação concorrente: rodar um laço continuamente, pegando itens de uma fila e processando cada um com uma função que realiza o trabalho real. O laço termina quando `worker` retira da fila um valor sentinela. Neste modelo, a sentinela que encerra o processo é muitas vezes chamada de *poison pill* (pílula venenosa).

`None` é bastante usado como valor sentinela, mas pode não ser adequado se for também um valor válido na série de dados. Invocar `object()` é uma forma comum de obter um objeto único para usar como sentinela.

Entretanto, isto não funciona entre processos, pois os objetos Python precisam ser serializados para comunicação entre processos. Quando você serializa um objeto com `pickle.dump` e desserializa com `pickle.load`, a instância recuperada tem uma identidade diferente do original. Uma boa alternativa a `None` é o objeto embutido `Ellipsis` (também conhecido como `...`), que sobrevive à serialização sem perder sua identidade.^[15]

A biblioteca padrão de Python usa «muitos valores diferentes» [fpy.li/19-22] como sentinelas. A *PEP 661—Sentinel Values* [fpy.li/pep661] propõe um tipo sentinela padrão.

Agora vamos estudar a função `main` de *procs.py* no Exemplo 14.

Exemplo 14. procs.py: checagem de números primos com múltiplos processos; função main

```
def main() -> None:
    if len(sys.argv) < 2: ①
        procs = cpu_count()
    else:
        procs = int(sys.argv[1])

    print(f'Checking {len(NUMBERS)} numbers with {procs} processes:')
    t0 = perf_counter()
    jobs: JobQueue = SimpleQueue() ②
    results: ResultQueue = SimpleQueue()
    start_jobs(procs, jobs, results) ③
```

```

checked = report(procs, results) ④
elapsed = perf_counter() - t0
print(f'{checked} checks in {elapsed:.2f}s') ⑤

def report(procs: int, results: ResultQueue) -> int: ⑥
    checked = 0
    procs_done = 0
    while procs_done < procs: ⑦
        n, prime, elapsed = results.get() ⑧
        if n == 0: ⑨
            procs_done += 1
        else:
            checked += 1 ⑩
            label = 'P' if prime else ' '
            print(f'{n:16} {label} {elapsed:9.6f}s')
    return checked

if __name__ == '__main__':
    main()

```

- ① Se nenhum argumento é dado na linha de comando, define o número de processos como o número de núcleos na CPU; caso contrário, cria a quantidade de processos indicada no primeiro argumento.
- ② jobs e results são as filas descritas no Exemplo 13.
- ③ Inicia proc processos para consumir jobs e computar results.
- ④ Recupera e exibe os resultados; report está definido em ⑥.
- ⑤ Mostra quantos números foram checados e o tempo total decorrido.
- ⑥ Os argumentos são o número de procs e a fila para armazenar os resultados.
- ⑦ Percorre o laço até que todos os processos terminem.
- ⑧ Obtém um PrimeResult. Chamar .get() em uma fila deixa o processamento bloqueado até que haja um item na fila. Também é possível fazer isso de forma não-bloqueante ou estabelecer um timeout. Veja os detalhes na documentação de SimpleQueue.get [fpy.li/ab].
- ⑨ Se n é zero, então um processo terminou; incrementa o contador procs_done.
- ⑩ Senão, incrementa o contador checked (para acompanhar os números checados) e mostra os resultados.

Os resultados não vão retornar na mesma ordem em que as tarefas foram submetidas. Por isso incluí `n` em cada tupla `PrimeResult`. De outra forma não teríamos como saber qual resultado corresponde a cada número.

Se o processo principal terminar antes que todos os subprocessos finalizem, podemos ter *tracebacks* difíceis de analisar, com referências a exceções de `FileNotFoundError` levantadas por uma trava interna em `multiprocessing`. Depurar código concorrente é sempre difícil, e depurar código baseado no `multiprocessing` é ainda mais difícil devido a toda a complexidade por trás da fachada que imita `threads`. Felizmente, o `ProcessPoolExecutor` que veremos no Capítulo 20 é mais fácil de usar e mais robusto.



Agradeço ao leitor Michael Albert, que notou que o código que publiquei durante o pré-lançamento tinha uma *race condition* (condição de corrida [fpy.li/ac]) no Exemplo 14. Uma condição de corrida é um bug que pode ou não ocorrer, dependendo da ordem das ações realizadas pelas unidades de execução concorrentes. Se "A" acontecer antes de "B", tudo segue normal; mas se "B" acontecer antes, acontece um erro. Esta é a corrida.

Se tiver curiosidade, veja o «diff que mostra o bug e a correção» [fpy.li/19-25] (mas saiba que depois eu refatorei o exemplo para delegar partes de `main` para as funções `start_jobs` e `report`). Escrevi um *README.md* [fpy.li/19-26] no mesmo diretório explicando o problema e a solução.

19.6.4. Experimentando com mais ou menos processos

Você pode experimentar rodar *procs.py*, passando argumentos que modifiquem o número de processos filhos. Por exemplo, este comando...

```
$ python3 procs.py 2
```

...vai iniciar dois subprocessos, produzindo os resultados quase duas vezes mais rápido que *sequential.py*—se a sua máquina tiver uma CPU com pelo menos dois núcleos e não estiver muito ocupada rodando outros programas.

Rodei *procs.py* 12 vezes, usando de 1 a 20 subprocessos, totalizando 240 execuções. Então calculei a mediana do tempo para todas as execuções com o mesmo número de subprocessos, e desenhei a Figura 2.

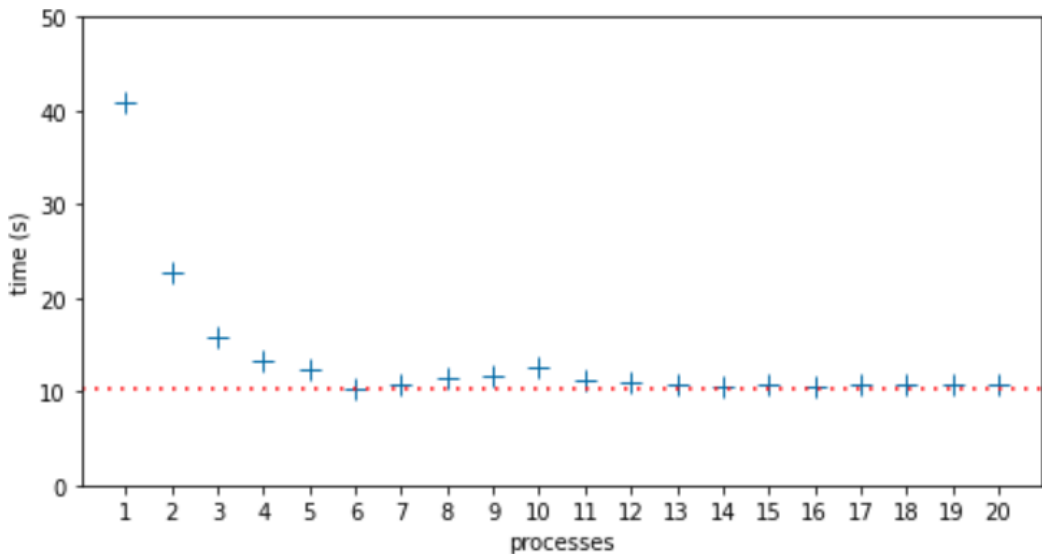


Figura 2. Mediana dos tempos de execução para cada número de subprocessos de 1 a 20. O maior tempo mediano foi 40,81s, com 1 processo. O tempo mediano mais baixo foi 10,39s, com 6 processos, indicado pela linha pontilhada.

Neste laptop de 6 núcleos, o menor tempo mediano ocorreu com 6 processos: 10.39s—marcado pela linha pontilhada na Figura 2. Seria de se esperar que o tempo de execução aumentasse após 6 processos, devido à disputa pela CPU, e ele atingiu um máximo local de 12.51s, com 10 processos. Eu não esperava e não sei explicar por que o desempenho melhorou com 11 processos e permaneceu praticamente igual com 13 a 20 processos, com tempos medianos apenas ligeiramente maiores que o menor tempo mediano com 6 processos.

19.6.5. Solução equivocada baseada em threads

Também escrevi *threads.py*, uma versão de *procs.py* usando *threading* em vez de *multiprocessing*. O código é muito similar quando convertemos exemplo simples entre as duas APIs.^[16] Devido à GIL e à natureza de processamento intensivo de *is_prime*, a versão com threads é mais lenta que a versão sequencial do Exemplo 12, e fica mais lenta conforme aumenta o número de threads, por causa da disputa pela CPU e o custo da mudança de contexto. Para passar de uma thread para outra, o SO precisa salvar os registradores da CPU e atualizar o contador de

programas e o ponteiro do stack, disparando efeitos colaterais custosos, como invalidar os caches da CPU e talvez até trocar páginas de memória.^[17]

Os dois próximos capítulos tratam de mais temas ligados à programação concorrente em Python, usando a biblioteca de alto nível *concurrent.futures* para gerenciar threads e processos (Capítulo 20) e a biblioteca *asyncio* para programação assíncrona (Capítulo 21).

As demais seções nesse capítulo procuram responder à questão:

Dadas as limitações discutidas até aqui, como é possível que Python seja tão bem-sucedido em um mundo de CPUs com múltiplos núcleos?

19.7. Python no mundo multi-núcleo.

Considere o seguinte trecho do artigo *The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software* [fpy.li/19-29] (O Almoço Grátis Acabou: Uma Virada Fundamental do Software em Direção à Concorrência) de Herb Sutter, publicado em 2005 e muito citado desde então:

Os fabricantes e arquiteturas de processadores mais importantes, desde Intel e AMD até Sparc e PowerPC, esgotaram o potencial da maioria das abordagens tradicionais de aumento do desempenho das CPUs. Ao invés de aumentar a frequência do clock [dos processadores] e acelerar o processamento das instruções sequenciais, eles estão se voltando em massa para o hyper-threading e para arquiteturas multi-núcleo.

O que Sutter chama de "almoço grátis" era a tendência do software ficar mais rápido sem qualquer esforço adicional por parte dos desenvolvedores, porque as CPUs executavam código sequencial cada vez mais rápido, com avanços exponenciais a cada nova geração. Desde 2004 isto não é mais verdade: a frequência dos *clocks* das CPUs e as otimizações de execução atingiram um platô, e agora qualquer melhoria significativa no desempenho precisa vir do aproveitamento de múltiplos núcleos ou do *hyperthreading*, avanços que só beneficiam código escrito para execução concorrente.

A história de Python começa no início dos anos 1990, quando as CPUs ainda estavam ficando exponencialmente mais rápidas na execução de código sequencial. Naquele tempo não se falava de CPUs com múltiplos núcleos, exceto para supercomputadores. Assim, a decisão de ter uma GIL era óbvia. A GIL torna mais leve e rápido o interpretador rodando em um único núcleo, e simplifica sua implementação.^[18] A GIL também torna mais fácil escrever extensões simples com a API Python/C.



Escrevi "extensões simples" porque uma extensão não é obrigada a lidar com a GIL. Uma função escrita em C ou Fortran pode ser centenas de vezes mais rápida que sua equivalente em Python.^[19] Assim, a complexidade adicional de liberar a GIL para tirar proveito de CPUs multi-núcleo pode ser desnecessária em muitos casos. Então podemos agradecer à GIL por muitas das extensões disponíveis em Python—e isso é certamente uma das razões fundamentais da popularidade da linguagem hoje.

Apesar da GIL, Python está cada vez mais popular entre aplicações que exigem execução concorrente ou paralela, graças a bibliotecas e arquiteturas de software que contornam as limitações do CPython.

Agora vamos discutir como Python é usado em administração de sistemas, ciência de dados, e desenvolvimento de aplicações para servidores no mundo do processamento distribuído e dos multi-núcleos de 2023.

19.7.1. Administração de sistemas

O Python é largamente utilizado para gerenciar grandes frotas de servidores, roteadores, balanceadores de carga e armazenamento conectado à rede (*network-attached storage* ou NAS). Ele é também a opção preferencial para redes definidas por software (SDN, *software-defined networking*) e hacking ético. Os maiores provedores de serviços na nuvem suportam Python através de bibliotecas e tutoriais de sua própria autoria, ou criados pela grande comunidade de usuários da linguagem.

Nesse campo, scripts Python automatizam tarefas de configuração, emitindo comandos a serem executados pelas máquinas remotas, então raramente há

operações limitadas pela CPU da máquina do administrador de sistemas. Threads ou corrotinas são bastante adequadas para tais atividades. Em particular, o pacote `concurrent.futures`, que veremos no Capítulo 20, pode ser usado para realizar as mesmas operações em muitas máquinas remotas ao mesmo tempo, sem grande complexidade.

Além da biblioteca padrão, há muitos projetos populares baseados em Python para gerenciar clusters (*agrupamentos*) de servidores: ferramentas como o *Ansible* [fpy.li/19-30] e o *Salt* [fpy.li/19-31], bem como bibliotecas como a *Fabric* [fpy.li/19-32].

Há também um número crescente de bibliotecas para administração de sistemas que suportam corrotinas e `asyncio`. Em 2016, a «equipe de Engenharia de Produção» [fpy.li/19-33] do Facebook relatou: "Estamos cada vez mais confiantes no `AsyncIO`, introduzido no Python 3.4, e vendo ganhos de desempenho imensos conforme migramos as bases de código de Python 2."

19.7.2. Ciência de dados

A ciência de dados—incluindo a inteligência artificial—e a computação científica estão muito bem servidas pelo Python.

Aplicações nesses campos são de processamento intensivo, mas os usuários de Python se beneficiam de um vasto ecossistema de bibliotecas de computação numérica, escritas em C, C++, Fortran, Cython, etc.—muitas delas capazes de aproveitar os benefícios de máquinas multi-núcleo, GPUs, e/ou computação paralela distribuída em clusters heterogêneos.

Em 2021, o ecossistema de ciência de dados de Python já incluía algumas ferramentas impressionantes:

Project Jupyter [fpy.li/19-34]

Duas interfaces para navegadores—Jupyter Notebook e JupyterLab—que permitem aos usuários rodar e documentar código analítico, que pode ser executado através da rede em máquinas remotas. Ambas são aplicações híbridas Python/JavaScript, suportando servidores de processamento (chamados *kernel*) escritos em diferentes linguagens, todos integrados via ZeroMQ—uma biblioteca de comunicação por mensagens assíncrona para aplicações distribuídas. O nome *Jupyter* remete a Julia, Python, e R, as três

primeiras linguagens suportadas pelo Notebook. O rico ecossistema construído sobre as ferramentas Jupyter inclui o *Bokeh* [fpy.li/19-35], uma poderosa biblioteca de visualização iterativa que permite aos usuários navegarem e interagirem com grandes conjuntos de dados ou um fluxo de dados continuamente atualizado, graças ao desempenho dos navegadores modernos e seus interpretadores JavaScript.

TensorFlow [fpy.li/19-36] e PyTorch [fpy.li/19-37]

Estes são os principais frameworks de aprendizagem profunda (*deep learning*), de acordo com o «relatório de Janeiro de 2021 da O'Reilly» [fpy.li/19-38] medido pela utilização em 2020. Os dois projetos são escritos em C++, e conseguem se beneficiar de múltiplos núcleos, GPUs e clusters. Eles também suportam outras linguagens, mas Python é seu maior foco e é usado pela maioria de seus usuários. O *TensorFlow* foi criado e é usado internamente pelo Google; o *PyTorch* pelo Facebook.

Dask [fpy.li/dask]

Uma biblioteca de computação paralela que delega tarefas para processos locais ou um cluster de máquinas, "testado em alguns dos maiores supercomputadores do mundo"—como seu «site afirma» [fpy.li/dask]. O Dask oferece APIs que emulam muito bem a NumPy, a pandas, e a scikit-learn—hoje as mais populares bibliotecas em ciência de dados e aprendizagem de máquina. O Dask pode ser usado a partir do JupyterLab ou do Jupyter Notebook, e usa o *Bokeh* não apenas para visualização de dados mas também para um painel interativo (*dashboard*) que mostra o fluxo de dados e a carga de processamento entre processos/máquinas quase em tempo real. O Dask é tão impressionante que recomendo assistir o «vídeo de demonstração» [fpy.li/19-39], onde o mantenedor Matthew Rocklin apresenta o Dask mastigando dados em 64 núcleos distribuídos por 8 máquinas EC2 na AWS.

Estes são apenas alguns exemplos para ilustrar como a comunidade de ciência de dados está criando soluções que aproveitam o melhor de Python e superam as limitações do runtime do CPython.

19.7.3. Servidores para Web/Computação Móvel

O Python é largamente utilizado em aplicações Web e em APIs de apoio a aplicações para computação móvel no servidor. Como o Google, o YouTube, o Dropbox, o Instagram, o Quora, e o Reddit—entre outros—conseguiram desenvolver aplicações de servidor em Python que atendem centenas de milhões de usuários todo dia? Novamente, a resposta vai bem além do que Python fornece em sua biblioteca padrão. Antes de discutir as ferramentas necessárias para usar Python em larga escala, preciso citar uma advertência do relatório *Technology Radar* da consultoria Thoughtworks:

Inveja de alto desempenho/inveja de escala da Web

Vemos muitas equipes se metendo em apuros por escolher ferramentas, frameworks ou arquiteturas complexas, porque eles "talvez precisem de escalabilidade". Empresas como Twitter e Netflix precisam suportar cargas extremas, então precisam dessas arquiteturas, mas elas também têm equipes de desenvolvimento numerosas, com anos de experiência, capazes de lidar com a complexidade. A maioria das situações não exige estas façanhas de engenharia; as equipes devem manter sua inveja da escalabilidade na web sob controle, e preferir soluções simples que fazem o que precisa ser feito.^[20]

Na *escala da Web*, a chave é uma arquitetura que permita escalabilidade horizontal. Neste cenário, todos os sistemas são sistemas distribuídos, e possivelmente nenhuma linguagem de programação será a alternativa ideal para todas as partes da solução.

Sistemas distribuídos são um campo da pesquisa acadêmica, mas felizmente alguns profissionais da área escreveram livros acessíveis, baseados em pesquisas sólidas e experiência prática. Um deles é Martin Kleppmann, o autor de *Designing Data-Intensive Applications* (Projetando Aplicações de Uso Intensivo de Dados) (O'Reilly).

Observe a Figura 3, o primeiro de muitos diagramas de arquitetura que adaptamos do livro de Kleppmann. Aqui há alguns componentes que vi em muitos ambientes Python onde trabalhei ou que conheci pessoalmente:

- Caches de aplicação:^[21] *memcached, Redis, Varnish*
- Bancos de dados relacionais: *PostgreSQL, MySQL*
- Bancos de documentos: *Apache CouchDB, MongoDB*
- Full-text indexes (índices de texto integral): *Elasticsearch, Apache Solr*
- Filas de mensagens: *RabbitMQ, Redis*

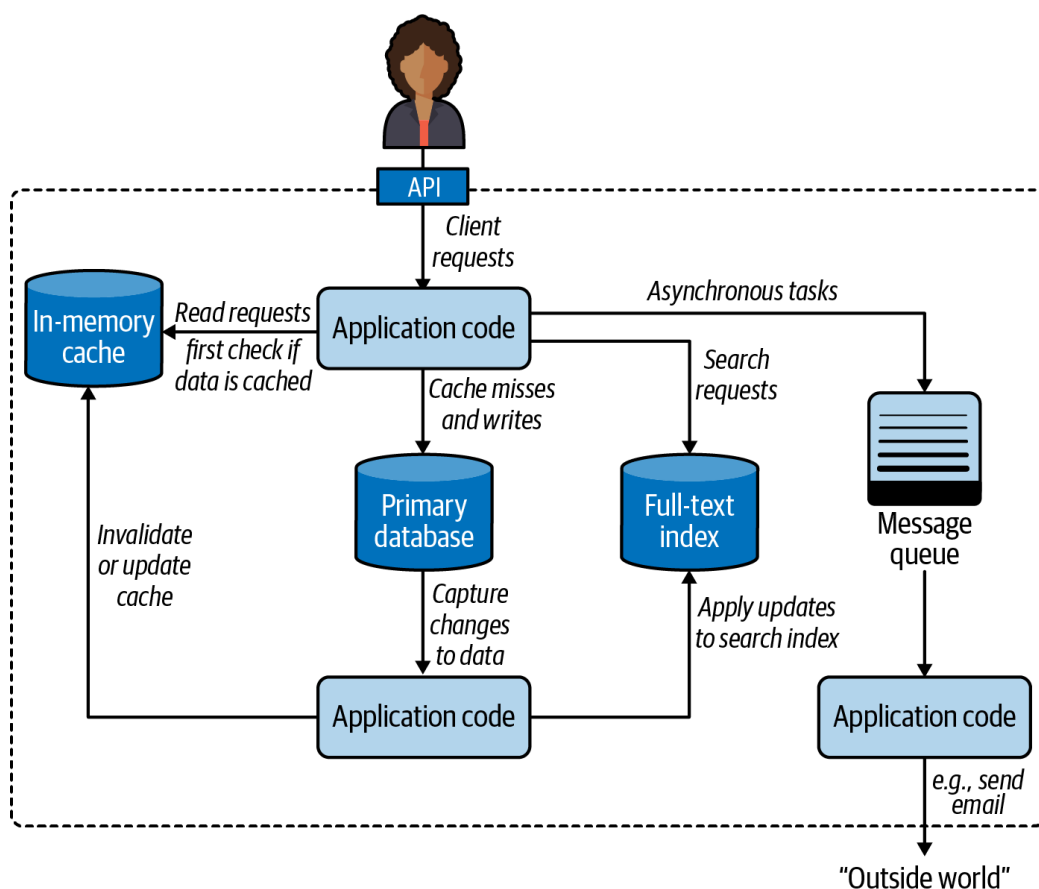


Figura 3. Uma arquitetura possível combinando diversos componentes.^[22]

Há outros produtos de código aberto extremamente robustos em cada uma dessas categorias. Os grandes fornecedores de serviços na nuvem também oferecem suas próprias alternativas proprietárias

O diagrama de Kleppmann é genérico e independente da linguagem—como seu livro. Para aplicações de servidor em Python, dois componentes específicos são comumente utilizados:

- Um servidor de aplicação, para distribuir a carga entre várias instâncias da aplicação Python. O servidor de aplicação apareceria perto do topo na Figura 3, processando as requisições dos clientes antes delas chegarem ao código da aplicação.
- Uma fila de tarefas construída em torno da fila de mensagens no lado direito da Figura 3, oferecendo uma API de alto nível e mais fácil de usar, para distribuir tarefas para processos rodando em outras máquinas.

As duas próximas seções exploram esses componentes, recomendados pelas boas práticas de implementações de aplicações Python de servidor.

19.7.4. Servidores de aplicação WSGI

A WSGI, *Web Server Gateway Interface* [fpy.li/pep3333] (Interface de Integração de Servidores Web), é a API padrão para uma aplicação ou um framework Python receber requisições de um servidor HTTP e enviar para ele as respostas.^[23]

Servidores de aplicação WSGI gerenciam um ou mais processos rodando a sua aplicação, maximizando o uso das CPUs disponíveis.

A Figura 4 ilustra uma instalação WSGI típica.



Se quiséssemos fundir os dois diagramas, o conteúdo do retângulo tracejado na Figura 4 substituiria o retângulo sólido "Application code"(*código da aplicação*) no topo da Figura 3.

Os servidores de aplicação mais conhecidos em projetos Web com Python são:

- *mod_wsgi* [fpy.li/19-41]
- *uWSGI* [fpy.li/19-42]^[24]
- *Gunicorn* [fpy.li/gunicorn]
- *NGINX Unit* [fpy.li/19-43]

Para usuários do servidor HTTP Apache, *mod_wsgi* é a melhor opção. Ele é tão antigo quanto a própria WSGI, mas é ativamente mantido, e agora pode ser iniciado via linha de comando com o *mod_wsgi-express*, que o torna mais fácil de configurar e mais apropriado para uso com containers Docker.

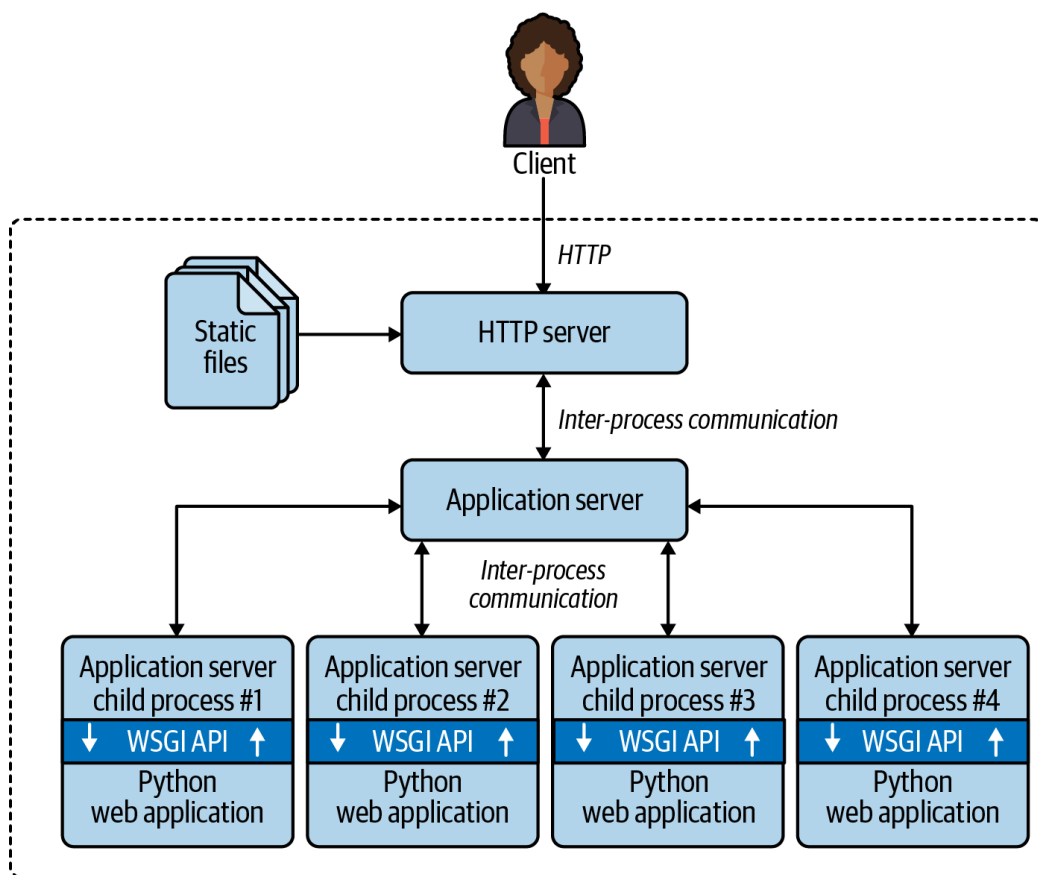


Figura 4. Clientes se conectam a um servidor HTTP que entrega arquivos estáticos e roteia outras requisições para o servidor de aplicação, que gerencia processos filhos para executar o código da aplicação, utilizando múltiplos núcleos de CPU. A API WSGI integra o servidor de aplicação ao código da aplicação Python.

O *uWSGI* e o *Gunicorn* são as escolhas mais populares entre os projetos recentes que conheço. Ambos são frequentemente combinados com o servidor HTTP *NGINX*. *uWSGI* oferece muita funcionalidade adicional, incluindo um cache de aplicação, uma fila de tarefas, tarefas periódicas estilo cron, e muitas outras. Por outro lado, o *uWSGI* é mais difícil de configurar corretamente que o *Gunicorn*.^[25]

Lançado em 2018, o *NGINX Unit* é um novo produto dos desenvolvedores do conhecido servidor HTTP e proxy reverso *NGINX*. O *mod_wsgi* e o *Gunicorn* só suportam aplicações Web Python, enquanto o *uWSGI* e o *NGINX Unit* funcionam também com outras linguagens. Para saber mais, consulte a documentação de cada um deles.

O ponto principal: todos esses servidores de aplicação podem, potencialmente, utilizar todos os núcleos de CPU no servidor, criando múltiplos processos Python para executar aplicações Web tradicionais escritas no bom e velho código sequencial em *Django*, *Flask*, *Pyramid*, etc. Isto explica como é possível ganhar a vida como desenvolvedor Python em aplicações *server side* sem nunca ter estudado os módulos *threading*, *multiprocessing*, ou *asyncio*: o servidor de aplicação lida de forma transparente com a concorrência.



ASGI—Asynchronous Server Gateway Interface

A WSGI é uma API síncrona. Ela não suporta corrotinas com *async/await*, que são a forma mais eficiente de implementar WebSockets em Python. A especificação da ASGI [fpy.li/19-46] é a sucessora assíncrona da WSGI, projetada para frameworks Python assíncronos para programação Web, como *aiohttp*, *Sanic*, *FastAPI*, etc., bem como *Django* e *Flask*, que estão gradualmente incorporando mais funcionalidades assíncronas.

Agora vamos examinar outra forma de evitar a GIL para obter um melhor desempenho em aplicações Python de servidor.

19.7.5. Filas de tarefas distribuídas

Quando o servidor de aplicação entrega uma requisição a um dos processos Python rodando sua aplicação, seu código precisa responder rápido: você quer que o processo esteja disponível para processar a requisição seguinte assim que possível. Entretanto, algumas requisições exigem ações que podem demorar—por exemplo, enviar um e-mail ou gerar um PDF. As filas de tarefas distribuídas foram projetadas para resolver este problema.

A *Celery* [fpy.li/19-47] e a *RQ* [fpy.li/19-48] são as filas de tarefas *Open Source* mais conhecidas com uma API para Python. Provedores de serviços na nuvem também oferecem suas filas de tarefas proprietárias.

Esses produtos encapsulam filas de mensagens e oferecem uma API de alto nível para delegar tarefas a processos executores, possivelmente rodando em máquinas diferentes.



No contexto de filas de tarefas, as palavras *produtor* e *consumidor* são usadas no lugar da terminologia tradicional de cliente/servidor. Por exemplo, para gerar documentos, uma view do Django *produz* requisições de serviço, colocadas em uma fila para serem *consumidas* por um ou mais processos renderizadores de PDFs.

Citando diretamente o «FAQ do Celery» [fpy.li/19-49], eis alguns casos de uso:

- *Executar algo em segundo plano. Por exemplo, para encerrar uma requisição Web o mais rápido possível, e então atualizar a página do usuário de forma incremental. Isso dá ao usuário a impressão de um bom desempenho e de "vivacidade", ainda que o trabalho real possa na verdade demorar um pouco mais.*
- *Executar algo após a requisição Web ter terminado.*
- *Assegurar-se de que algo seja feito, através de execução assíncrona, repetindo tentativas quando necessário.*
- *Agendar tarefas periódicas.*

Além de resolver esses problemas imediatos, as filas de tarefas suportam escalabilidade horizontal. Produtores e consumidores são desacoplados: um produtor não precisa chamar um consumidor, ele coloca uma requisição em uma fila. Consumidores não precisam saber nada sobre os produtores (mas a requisição pode incluir informações sobre o produtor, se uma confirmação for necessária). Pode-se adicionar mais unidades de execução para consumir tarefas à medida que a demanda cresce. Por isso a *Celery* e a *RQ* são chamadas de filas de tarefas distribuídas.

Lembre-se de que nosso simples *procs.py* (Exemplo 13) usava duas filas: uma para requisitar tarefas, outra para coletar resultados. A arquitetura distribuída do *Celery* e do *RQ* usa um esquema similar. Ambos suportam o uso do banco de dados NoSQL *Redis* [fpy.li/19-50] para armazenar as filas de mensagens e resultados. O *Celery* também suporta outras filas de mensagens, como a *RabbitMQ* ou a *Amazon SQS*, e também alguns bancos de dados para armazenar resultados.

Isto encerra nossa introdução à concorrência em Python. Os dois próximos capítulos continuam nesse tema com mais código, demonstrando os pacotes `concurrent.futures` e `asyncio` da biblioteca padrão.

19.8. Resumo do capítulo

Após um pouco de teoria, esse capítulo apresentou scripts da animação giratória, implementados em cada um dos três modelos de programação de concorrência nativos de Python:

- Threads, com o pacote `threading`
- Processo, com `multiprocessing`
- Corrotinas assíncronas com `asyncio`

Então exploramos o impacto real da GIL com um experimento: mudar os exemplos de animação para computar se um inteiro grande era primo e observar o comportamento resultante. Assim comprovamos que funções que usam a CPU intensivamente devem ser evitadas em `asyncio`, pois elas bloqueiam o laço de eventos. A versão com threads do experimento funcionou—apesar da GIL—porque Python interrompe periodicamente as threads, e o exemplo usou apenas duas threads: uma fazendo um trabalho de computação intensiva, a outra controlando a animação apenas 10 vezes por segundo. A variante com `multiprocessing` contornou a GIL, iniciando um novo processo só para a animação, enquanto o processo principal calculava se o número era primo.

O exemplo seguinte, computando a primalidade de uma série de números, destacou a diferença entre `multiprocessing` e `threading`, provando que apenas processos permitem ao Python se beneficiar de CPUs com múltiplos núcleos. A GIL de Python faz com que as threads sejam mais lentas que o código sequencial para processamento pesado.

A GIL domina as discussões sobre computação concorrente e paralela em Python, mas não devemos superestimar seu impacto. Este foi o tema da Seção 19.7. Por exemplo, a GIL não afeta muitos dos casos de uso de Python em administração de sistemas. Por outro lado, as comunidades de ciência de dados e de desenvolvimento para servidores evitaram os problemas com a GIL usando soluções robustas, criadas sob medida para suas necessidades específicas.

As últimas duas seções mencionaram os dois elementos comuns que sustentam o uso de Python em aplicações de servidor escaláveis: servidores de aplicação WSGI e filas de tarefas distribuídas.

19.9. Para saber mais

Este capítulo tem uma extensa lista de referências, então a dividi em subseções.

19.9.1. Concorrência com threads e processos

A biblioteca `concurrent.futures`, tratada no Capítulo 20, usa threads, processos, travas e filas debaixo dos panos, mas você não verá as instâncias individuais destes componentes; eles são encapsulados e gerenciados por abstrações de nível mais alto: `ThreadPoolExecutor` ou `ProcessPoolExecutor`. Para aprender mais sobre a prática da programação concorrente com aqueles objetos de baixo nível, *An Intro to Threading in Python* [fpy.li/19-51] (Uma introdução a *threading* em Python) de Jim Anderson é uma boa primeira leitura. Doug Hellmann tem um capítulo chamado *Concurrency with Processes, Threads, and Coroutines* em seus site [fpy.li/19-52] e livro, *The Python 3 Standard Library by Example* [fpy.li/19-53] (Addison-Wesley).

O *Effective Python* [fpy.li/effectpy], 2nd ed. (Addison-Wesley), de Brett Slatkin, *Python Essential Reference*, 4th ed. (Addison-Wesley), de David Beazley, e *Python in a Nutshell*, 3rd ed. (O'Reilly) de Martelli et.al. são outras referências gerais de Python com uma cobertura significativa de *threading* e *multiprocessing*. A vasta documentação oficial de *multiprocessing* inclui conselhos úteis em sua seção *Programming guidelines* [fpy.li/ae] (Diretrizes de programação).

Jesse Noller e Richard Oudkerk contribuíram para o pacote *multiprocessing*, proposto na *PEP 371—Addition of the multiprocessing package to the standard library* [fpy.li/pep371]. A documentação oficial do pacote é um arquivo `.rst` [fpy.li/ah] de 93 KB (cerca de 63 páginas), um dos capítulos mais longos da biblioteca padrão de Python.

Em *High Performance Python* [fpy.li/19-56] (O'Reilly), os autores Micha Gorelick e Ian Ozsvald incluem um capítulo sobre *multiprocessing* com um exemplo sobre checagem de números primos usando uma estratégia diferente do nosso exemplo *procs.py*. Para cada número, eles dividem a faixa de fatores possíveis de 2 a

`sqrt(n)`—em subfaixas, e fazem cada unidade de execução iterar sobre uma das subfaixas. Sua abordagem de dividir para conquistar é típica de aplicações de computação científica, onde os conjuntos de dados são enormes, e as estações de trabalho (ou clusters) têm mais núcleos de CPU que usuários. Em um sistema servidor, processando requisições de muitos usuários, é mais simples e mais eficiente deixar cada processo realizar uma tarefa computacional do início ao fim—reduzindo a sobrecarga de comunicação e coordenação entre processos. Além de multiprocessing, Gorelick e Ozsvald apresentam muitas outras formas de desenvolver e implantar aplicações de ciência de dados de alto desempenho, aproveitando múltiplos núcleos de CPU, GPUs, clusters, analisadores e compiladores como *Cython* e *Numba*. Seu capítulo final, *Lessons from the Field* (Lições da Vida Real) é uma valiosa coleção de estudos de caso curtos, contribuição de outros praticantes de computação de alto desempenho em Python.

O *Advanced Python Development* [fpy.li/19-57], de Matthew Wilkes (Apress), mostra como desenvolver uma aplicação realista pronta para implantação em produção: um agregador de dados, similar aos sistemas de monitoramento DevOps ou aos coletores de dados para sensores distribuídos IoT. Dois capítulos no *Advanced Python Development* tratam de programação concorrente com `threading` e `asyncio`.

O *Parallel Programming with Python* [fpy.li/19-58] (Packt, 2014), de Jan Palach, explica os principais conceitos por trás da concorrência e do paralelismo, abarcando a biblioteca padrão de Python bem como a *Celery*.

The Truth About Threads (A Verdade Sobre as Threads) é o título do capítulo 2 de *Using Asyncio in Python* [fpy.li/hattingh], de Caleb Hattingh (O'Reilly).^[26] O capítulo trata dos benefícios e das desvantagens das threads—com citações convincentes de várias fontes confiáveis—deixando claro que os desafios fundamentais das threads não têm relação com Python ou a GIL. Citando literalmente a página 14 de *Using Asyncio in Python* (nossa tradução):

Estes temas se repetem com frequência:

- Programação com threads torna o código difícil de analisar.
- Programação com threads é um modelo ineficiente para concorrência em larga escala (milhares de tarefas concorrentes).

Se você quiser aprender do jeito difícil como é complicado raciocinar sobre threads e travas—sem colocar seu emprego em risco—tente resolver os problemas no livro de Allen Downey *The Little Book of Semaphores* [fpy.li/19-59] (Green Tea Press). O livro inclui exercícios muito difíceis e até sem solução conhecida, mas até os fáceis são desafiadores.

19.9.2. A GIL

Se você ficou curioso sobre a GIL, lembre-se de que não temos controle sobre ela a partir do código em Python, então a referência canônica é a documentação da C-API: *Thread State and the Global Interpreter Lock* [fpy.li/19-60] (O Estado das Threads e a Trava Global do Interpretador). A resposta no FAQ *Python Library and Extension* (A Biblioteca e as Extensões de Python): *Can't we get rid of the Global Interpreter Lock?* [fpy.li/af] (Não podemos remover o Bloqueio Global do interpretador?). Também vale a pena ler os posts de Guido van Rossum e Jesse Noller (contribuidor do pacote `multiprocessing`), respectivamente: *It isn't Easy to Remove the GIL* [fpy.li/19-62] (Não é Fácil Remover a GIL) e *Python Threads and the Global Interpreter Lock* [fpy.li/19-63] (As Threads de Python e a Trava Global do Interpretador).

CPython Internals [fpy.li/19-64], de Anthony Shaw (Real Python) explica a implementação do interpretador CPython 3 no nível da programação em C. O capítulo mais longo do livro é "Parallelism and Concurrency" (*Paralelismo e Concorrência*): um mergulho profundo no suporte nativo de Python a threads e processos, incluindo o gerenciamento da GIL por extensões usando a API C/Python.

David Beazley apresentou uma exploração detalhada em *Understanding the Python GIL* [fpy.li/19-65] (*Entendendo a GIL de Python*).^[27] No slide 54 da apresentação [fpy.li/19-66], Beazley relata um aumento no tempo de processamento de um benchmark específico com o novo algoritmo da GIL, introduzido no Python 3.2. O problema não tem importância com cargas de trabalho reais, de acordo com um «comentário» [fpy.li/19-67] de Antoine Pitrou (que implementou um novo algoritmo para a GIL) no relatório de bug submetido por Beazley: *Python issue #7946* [fpy.li/19-68].

19.9.3. Concorrência além da biblioteca padrão

O *Python Fluente* se concentra nos recursos fundamentais da linguagem e nas partes centrais da biblioteca padrão. *Full Stack Python* [fpy.li/19-69] é um ótimo complemento para esse livro: é sobre o ecossistema de Python, com seções chamadas "Development Environments (*Ambientes de Desenvolvimento*)," "Data (*Dados*)," "Web Development (*Desenvolvimento Web*)," e "DevOps," entre outros.

Já mencionei dois livros que abordam a concorrência usando a biblioteca padrão de Python e também incluem conteúdo significativo sobre bibliotecas externas e ferramentas: *High Performance Python, 2nd ed.* [fpy.li/19-56] e *Parallel Programming with Python* [fpy.li/19-58]. O *Distributed Computing with Python* [fpy.li/19-72] de Francesco Pierfederici (Packt) cobre a biblioteca padrão e também provedores de infraestrutura de nuvem e clusters HPC (*High-Performance Computing*, computação de alto desempenho).

O *Python, Performance, and GPUs* [fpy.li/19-73] de Matthew Rocklin é uma atualização do status do uso de aceleradores GPU com Python, publicado em junho de 2019.

"O Instagram hoje representa a maior instalação do mundo do framework Web *Django*, escrito inteiramente em Python." Essa é a linha de abertura do post *Web Service Efficiency at Instagram with Python* [fpy.li/19-74], escrito por Min Ni, da engenharia do Instagram. O post descreve as métricas e ferramentas usadas pelo Instagram para otimizar a eficiência de sua base de código Python, bem como para detectar e diagnosticar regressões de desempenho a cada uma das "30 a 50 vezes diárias" que o back-end é atualizado.

Architecture Patterns with Python: Enabling Test-Driven Development, Domain-Driven Design, and Event-Driven Microservices [fpy.li/19-75], de Harry Percival e Bob Gregory (O'Reilly) apresenta modelos de arquitetura para aplicações de servidor em Python. Os autores publicaram o livro gratuitamente online em *cosmicpython.com* [fpy.li/19-76].

Duas bibliotecas elegantes e fáceis de usar para a paralelização de processos são a *lelo* [fpy.li/19-77] de João S. O. Bueno e a *python-parallelize* [fpy.li/19-78] de Nat Pryce. O pacote *lelo* define um decorador `@parallel` que você pode aplicar a qualquer função para torná-la magicamente não-bloqueante: quando você chama uma função decorada, sua execução é iniciada em outro processo. O

pacote *python-parallelize* de Nat Pryce fornece um gerador *parallelize*, que distribui a execução de um laço *for* por múltiplas CPUs. Ambos os pacotes são baseados na biblioteca *multiprocessing*.

Eric Snow, um dos mantenedores do Python, mantém um wiki chamado *Multicore Python* [fpy.li/19-79], com observações sobre os esforços dele e de outros para melhorar o suporte de Python para execução em paralelo. Snow é o autor da *PEP 554-Multiple Interpreters in the Stdlib* [fpy.li/pep554] (Múltiplos interpretadores na biblioteca padrão). A PEP 554 assenta as bases para melhorias futuras, que podem um dia permitir que Python use múltiplos núcleos sem pagar os altos custos de inicialização, memória, e comunicação do *multiprocessing*. Um dos grandes empecilhos é a iteração entre múltiplos subinterpretadores ativos e extensões que assumem a existência de um único interpretador.

Mark Shannon—também um mantenedor do Python—criou uma «tabela» [fpy.li/19-80] bem útil comparando os modelos de concorrência em Python, referida em uma discussão sobre subinterpretadores entre ele, Eric Snow e outros desenvolvedores na lista de discussão *python-dev* [fpy.li/19-81]. Na tabela de Shannon, a coluna "Ideal CSP" se refere ao modelo teórico *Communicating Sequential Processes* [fpy.li/19-82] (processos sequenciais comunicantes), proposto por Tony Hoare em 1978, que influenciou o projeto da linguagem Go. Go também permite objetos compartilhados, violando uma das restrições essenciais do CSP: as unidades de execução devem se comunicar somente por mensagens enviadas por canais.

O *Stackless Python* [fpy.li/19-83] (também conhecido como *Stackless*) é um fork do CPython que implementa *microthreads*, que são *threads* leves no nível da aplicação—ao contrário das *threads* do SO. O jogo online multijogador massivo *EVE Online* [fpy.li/19-84] foi desenvolvido com *Stackless*, e os engenheiros da desenvolvedora de jogos CCP [fpy.li/19-85] foram «mantenedores do *Stackless*» [fpy.li/19-86] por algum tempo. Alguns recursos do *Stackless* foram reimplementados no interpretador *Pypy* [fpy.li/19-87] e no pacote *greenlet* [fpy.li/19-14], a tecnologia central da biblioteca de programação em rede *gevent* [fpy.li/19-17], que por sua vez é a fundação do servidor de aplicação *Gunicorn* [fpy.li/gunicorn].

O modelo de atores (*actor model*) de programação concorrente está no centro das linguagens altamente escaláveis Erlang e Elixir, e é também o modelo do framework Akka para Scala e Java. Se você quiser experimentar o modelo de atores em Python, veja as bibliotecas *Thespian* [fpy.li/19-90] e *Pykka* [fpy.li/19-91].

Minhas recomendações restantes fazem pouca ou nenhuma menção direta ao Python, mas são importantes para aprofundar o tema das arquiteturas escaláveis com suporte à concorrência.

19.9.4. Concorrência e escalabilidade para além de Python

RabbitMQ in Action [fpy.li/19-92] (Manning), de Alvaro Videla e Jason J. W. Williams, é uma introdução muito bem escrita ao *RabbitMQ* e ao padrão AMQP (*Advanced Message Queuing Protocol*, Protocolo Avançado de Enfileiramento de Mensagens), com exemplos em Python, PHP, e Ruby. Independente do resto de seu stack tecnológico, e mesmo se você planeja usar *Celery* com *RabbitMQ* debaixo dos panos, recomendo esse livro por sua abordagem dos conceitos, da motivação e dos modelos das filas de mensagem distribuídas, bem como a operação e configuração do *RabbitMQ* em larga escala.

Aprendi muito lendo *Seven Concurrency Models in Seven Weeks* [fpy.li/19-93], de Paul Butcher (Pragmatic Bookshelf), que traz o eloquente subtítulo *When Threads Unravel*.^[28] O capítulo 1 do livro apresenta os conceitos centrais e os desafios da programação com threads e travas em Java.^[29] Os outros seis capítulos do livro são dedicados ao que o autor considera as melhores alternativas para programação concorrente e paralela, e como funcionam com diferentes linguagens, ferramentas e bibliotecas. Os exemplos usam Java, Clojure, Elixir, e C (no capítulo sobre programação paralela com o framework OpenCL [fpy.li/19-94]). O modelo CSP é exemplificado com código Clojure, e não Go—que popularizou esta abordagem. Elixir é a linguagem dos exemplos que ilustram o modelo de atores. Um «capítulo bonus» [fpy.li/19-95] (disponível online gratuitamente) sobre atores usa Scala e o framework Akka. A menos que você já saiba Scala, Elixir é uma linguagem mais acessível para aprender e experimentar o modelo de atores e plataforma de sistemas distribuídos Erlang/OTP.

Unmesh Joshi, da Thoughtworks criou uma série de artigos documentando padrões de sistemas distribuídos no «blog de Martin Fowler» [fpy.li/19-96]. A «página de abertura da série» [fpy.li/19-97] é uma ótima introdução ao assunto, com links para vários padrões. Joshi está acrescentando modelos gradualmente, mas o que já está publicado reflete anos de experiência adquirida a duras penas em sistema de missão crítica.

O Designing Data-Intensive Applications [fpy.li/19-98] (Projetando aplicações intensivas em dados), de Martin Kleppmann (O'Reilly), é um dos raros livros

escritos por um profissional com vasta experiência prática e também conhecimento acadêmico avançado sobre sistemas distribuídos. O autor trabalhou com infraestrutura de dados em larga escala no LinkedIn e em duas startups, antes de se tornar um pesquisador de sistemas distribuídos na Universidade de Cambridge. Cada capítulo do livro termina com uma extensa lista de referências, incluindo resultados de pesquisas recentes. O livro também inclui vários diagramas esclarecedores e lindos mapas conceituais.

Tive a sorte de assistir ao excelente workshop de Francesco Cesarini sobre a arquitetura de sistemas distribuídos confiáveis, na OSCON 2016: *Designing and architecting for scalability with Erlang/OTP* (Projetando e estruturando para a escalabilidade com Erlang/OTP) («video só para assinantes» [fpy.li/19-99] na O'Reilly Learning Platform). Apesar do título, aos 9:35 no vídeo, Cesarini explica:

Muito pouco do que vou dizer será específico de Erlang [...]. Resta o fato de que o Erlang remove muitas dificuldades acidentais no desenvolvimento de sistemas resilientes que nunca falham, além de serem escaláveis. Então será mais fácil se vocês usarem Erlang ou uma linguagem rodando na máquina virtual Erlang.

Aquele workshop foi baseado nos últimos quatro capítulos do *Designing for Scalability with Erlang/OTP* [fpy.li/19-100] de Francesco Cesarini e Steve Vinoski (O'Reilly).

Desenvolver sistemas distribuídos é desafiador e empolgante, mas cuidado com a «inveja da escalabilidade na web» [fpy.li/19-40]. O «princípio KISS» [fpy.li/19-102] (KISS é a sigla de *Keep It Simple, Stupid*: "Deixe Simples, Idiota") continua sendo uma recomendação sábia de engenharia.

Veja também o artigo *Scalability! But at what COST?* [fpy.li/19-103], de Frank McSherry, Michael Isard, e Derek G. Murray. Os autores identificaram sistemas paralelos de processamento de grafos apresentados em simpósios acadêmicos que precisavam de centenas de núcleos para superar "uma implementação competente com uma única thread." Eles também encontraram sistemas que "têm desempenho pior que uma thread em todas as configurações documentadas." Estes resultados me lembram uma piada clássica entre *hackers*:

Meu script Perl é mais rápido que seu cluster Hadoop.

Ponto de vista

Para gerenciar a complexidade, precisamos de restrições

Aprendi a programar em uma calculadora TI-58. Sua "linguagem" era similar ao assembler. Naquele nível, todas as "variáveis" eram globais, e não havia o conforto dos comandos estruturados de controle de fluxo. Existiam saltos condicionais: instruções que transferiam a execução diretamente para uma localização arbitrária—à frente ou atrás do local atual—dependendo do valor de um registrador na CPU.

É possível fazer basicamente qualquer coisa em assembler, e esse é o desafio: há muito poucas restrições para evitar que você cometa erros, e para ajudar mantenedores a entender o código quando mudanças são necessárias.

A segunda linguagem que aprendi foi o BASIC desestruturado que vinha nos computadores de 8 bits—nada comparável ao Visual Basic, que surgiu mais tarde. BASIC tinha as instruções FOR, GOSUB e RETURN, mas não variáveis locais! Todas as linhas de código eram explicitamente numeradas no código-fonte. O GOSUB não permitia passagem de parâmetros: era apenas um GOTO mais chique, que guardava um número de linha de retorno em uma pilha, fornecendo um local para a instrução RETURN saltar de volta. Subrotinas só podiam ler e escrever dados globais. Era preciso improvisar outras formas de controle de fluxo, com combinações de IF e GOTO—que permitia saltar para qualquer linha do programa.

Após alguns anos programando com saltos e variáveis globais, lembro da batalha para reestruturar meu cérebro para a "programação estruturada", quando aprendi Pascal. Agora era obrigatório usar instruções de controle de fluxo em torno de blocos de código que tinham um único ponto de entrada. Não podia mais saltar para qualquer instrução que desejasse. Variáveis globais eram inevitáveis em BASIC, mas agora se tornaram tabu. Eu precisava repensar o fluxo de dados e passar argumentos para funções explicitamente.

Meu próximo desafio foi aprender programação orientada a objetos. No fundo, programação orientada a objetos é programação estruturada com

mais restrições e polimorfismo. O ocultamento de informações (*information hiding*) força uma nova perspectiva sobre onde os dados residem. Lembro de mais de uma vez ficar frustrado por ter que refatorar meu código, para que um método que estava escrevendo pudesse obter informações que estavam encapsuladas em um objeto que aquele método não conseguia acessar.

Linguagens de programação funcionais acrescentam outras restrições, mas a imutabilidade é a mais difícil de engolir, após décadas de programação imperativa e orientada a objetos. Após nos acostumarmos a tais restrições, as vemos como bênçãos. Elas facilitam pensar sobre o código.

A falta de restrições é o maior problema com o modelo de threads-e-travas de programação concorrente. Ao resumir o capítulo 1 de *Seven Concurrency Models in Seven Weeks*, Paul Butcher escreveu:

A maior fraqueza da abordagem, entretanto, é que programação com threads-e-travas é difícil. Pode ser fácil para um projetista de linguagens acrescentá-las a uma linguagem, mas elas oferecem muito pouca ajuda a nós, pobres programadores.

Alguns exemplos de comportamento sem restrições naquele modelo:

- Threads podem compartilhar estruturas de dados mutáveis arbitrárias.
- O *scheduler* pode interromper uma thread praticamente em qualquer ponto, incluindo no meio de uma operação simples, como `a += 1`. Muito poucas operações são atômicas no nível das expressões do código-fonte.
- Travas são, em geral, recomendações (*advisory*). Esse é um termo técnico, dizendo que você precisa lembrar de obter explicitamente uma trava antes de atualizar uma estrutura de dados compartilhada. Se você esquecer de obter a trava, nada impede seu código de bagunçar os dados enquanto outra thread, que obedientemente detém a trava, está atualizando os mesmos dados.

Em comparação, considere algumas restrições impostas pelo modelo de atores, no qual a unidade de execução é chamada *actor* (ator):^[30]

- Um ator pode ter um estado interno, mas não pode compartilhar este estado com outros atores.
- Atores só podem se comunicar enviando e recebendo mensagens.
- Mensagens contém só cópias de dados, e não referências para dados mutáveis.
- Um ator só processa uma mensagem de cada vez. Não há execução concorrente dentro de um único ator.

Claro, é possível adotar uma forma de programação no estilo de atores em qualquer linguagem, seguindo essas regras. Você também pode usar padrões de programação orientada a objetos em C, e até padrões de programação estruturada em assembler. Mas fazer isso requer muita consistência e disciplina da parte de qualquer um que mexa no código.

Gerenciar travas é desnecessário no modelo de atores, como implementado em Erlang e Elixir, onde todos os tipos de dados são imutáveis.

Threads-e-travas não vão desaparecer. Eu só acho que lidar com tais mecanismos de baixo nível não é um bom uso do meu tempo quando escrevo aplicações—e não módulos do kernel, drivers de hardware, ou servidores de bancos de dados.

Sempre me reservo o direito de mudar de opinião. Mas neste momento, estou convencido de que o modelo de atores é o modelo de programação concorrente mais sensato que existe. CSP (Communicating Sequential Processes) também é sensato, mas sua implementação em Go deixa de fora algumas restrições importantes. A ideia em CSP é que corrotinas (ou *goroutines* em Go) trocam dados e se sincronizam somente usando filas, chamadas *channels* (canais) em Go. Mas Go também permite compartilhamento de memória e travas. Vi um livro sobre Go defender o uso de memória compartilhada e travas em vez de canais—para maximizar o desempenho. É difícil abandonar velhos hábitos.

[1] Slide 8 da palestra *Concurrency Is Not Parallelism* [fpy.li/19-1] (Concorrência não é paralelismo).

[2] Estudei e trabalhei com o Prof. Imre Simon, que gostava de dizer que há dois grandes pecados na ciência: usar palavras diferentes para significar a mesma coisa e usar uma palavra para significar coisas diferentes. Imre Simon (1942-2009) foi um pioneiro da ciência da computação no Brasil, com

contribuições seminais para a Teoria dos Autômatos. Ele fundou o campo da Matemática Tropical e foi também um defensor do software livre, da cultura livre, e da Wikipédia.

[3] Essa seção foi sugerida por meu amigo Bruce Eckel—autor de livros sobre Kotlin, Scala, Java, e C++.

[4] NT: "FIFO" é a sigla em inglês para "first in, first out".

[5] Invoque `sys.getswitchinterval()` [fpy.li/a5] para obter o intervalo; ele pode ser modificado com `sys.setswitchinterval(s)` [fpy.li/ag].

[6] Uma `syscall` é uma chamada a partir do código do usuário para uma função do núcleo (*kernel*) do sistema operacional. E/S, temporizadores e travas são alguns dos serviços do núcleo do SO disponíveis através de `syscalls`. Para aprender mais sobre esse tópico, leia o artigo «Chamada de sistema» [fpy.li/a6] na Wikipédia.

[7] Os módulos `zlib` e `bz2` são mencionados nominalmente em uma mensagem de Antoine Pitrou na `python-dev` [fpy.li/19-6]. Pitrou contribuiu para a lógica da divisão de tempo da GIL no Python 3.2.

[8] Fonte: slide 106 do tutorial de Beazley, "Generators: The Final Frontier" [fpy.li/19-7].

[9] O Unicode tem muitos caracteres úteis para animações simples, como por exemplo os padrões Braille [fpy.li/19-11]. Usei os caracteres ASCII `'\|/-'` para simplificar os exemplos do livro.

[10] O semáforo é um bloco fundamental que pode ser usado para implementar outros mecanismos de sincronização. Python fornece diferentes classes de semáforos para uso com threads, processos e corrotinas. Veremos o `asyncio.Semaphore` na Seção 21.7.1 (Capítulo 21).

[11] Agradeço aos revisores técnicos Caleb Hattingh e Jürgen Gmach, que não me deixaram esquecer de *greenlet* e *gevent*.

[12] É um MacBook Pro 15" de 2018, com uma CPU Intel Core i7 2.2 GHz de 6 núcleos.

[13] Isso é verdade hoje porque você provavelmente está usando um SO moderno, com *multitarefa preemptiva*. O Windows antes da era NT e o MacOS antes da era OSX não eram "preemptivos", então qualquer processo podia tomar 100% da CPU e paralisar o sistema inteiro. Não estamos inteiramente livres desse tipo de problema hoje, mas confie na minha barba branca: esse tipo de coisa assombrava todos os usuários nos anos 1990, e a única cura era um reset de hardware.

[14] Nesse exemplo, `0` é um sinal conveniente. `None` também é bastante usado para este fim, mas o `0` simplifica a dica de tipo para `PrimeResult` e a implementação de `worker`.

[15] Sobreviver à serialização sem perder nossa identidade é um ótimo objetivo de vida.

[16] Veja `19-concurrency/primes/threads.py` [fpy.li/19-27] no «repositório de código» [fpy.li/code] do *Fluent Python*.

[17] Para saber mais, consulte "Troca de contexto" [fpy.li/ad] na Wikipédia.

[18] Provavelmente foram estas razões que levaram o criador de Ruby, Yukihiro Matsumoto, a também usar uma GIL no seu interpretador.

[19] Na faculdade, como exercício, tive que implementar o algoritmo de compressão LZW em C. Mas antes escrevi o código em Python, para verificar meu entendimento da especificação. A versão C foi cerca de 900 vezes mais rápida.

[20] Fonte: Thoughtworks Technology Advisory Board, *Technology Radar—November 2015* [fpy.li/19-40].

[21] Compare os caches de aplicação—usados diretamente pelo código de sua aplicação—com caches HTTP, que estariam no limite superior da Figura 3, servindo recursos estáticos como imagens e arquivos CSS ou JS. Redes de Fornecimento de Conteúdo (CDNs de *Content Delivery Networks*) oferecem outro tipo de cache HTTP, instalados em datacenters próximos aos usuários finais de sua

aplicação.

[22] Diagrama adaptado da Figura 1-1, *Designing Data-Intensive Applications* de Martin Kleppmann (O'Reilly).

[23] Alguns palestrantes soletram a sigla WSGI, enquanto outros a pronunciam como uma palavra rimando com "whisky."

[24] *uWSGI* é escrito com um "u" minúsculo, mas pronunciado como a letra grega "μ," então o nome completo soa como "micro-whisky", mas com um "g" no lugar do "k."

[25] Os engenheiros da Bloomberg Peter Sperl e Ben Green escreveram *Configuring uWSGI for Production Deployment* [fpy.li/19-44] (Configurando o uWSGI para Implantação em Produção), explicando como muitas das configurações default do *uWSGI* não são adequadas para cenários comuns de implantação. Sperl apresentou um resumo de suas recomendações na *EuroPython 2019* [fpy.li/19-45]. Muito recomendado para usuários de *uWSGI*.

[26] Caleb é um dos revisores técnicos da segunda edição de *Python Fluente*.

[27] Agradeço a Lucas Brunialti por me enviar um link para essa palestra.

[28] NT: Trocadilho intraduzível com thread no sentido de "fio" ou "linha", algo como "Quando as linhas desfiam."

[29] As APIs Python `threading` e `concurrent.futures` foram fortemente influenciadas pela biblioteca padrão de Java.

[30] A comunidade Erlang usa o termo "processo" para se referir a um ator. Em Erlang, cada processo é apenas uma função em seu próprio laço, então são muito leves, possibilitando ter milhões deles ativos ao mesmo tempo em uma única máquina—nenhuma relação com os pesados processos do SO, dos quais falamos em outros pontos deste capítulo. Então temos aqui os dois pecados descritos pelo Prof. Simon: usar palavras diferentes para se referir à mesma coisa, e usar uma palavra para se referir a coisas diferentes.

Capítulo 20. Executores concorrentes

Quem fala mal de threads são tipicamente programadoras de sistemas, que têm em mente casos de uso que a programadora de aplicações típica nunca encontrará na vida.[...] Em 99% dos casos de uso que a programadora de aplicações poderá encontrar, o modelo simples de disparar um monte de threads independentes e coletar os resultados em uma fila é tudo que se precisa saber.^[1]

— Michele Simionato, profundo pensador de Python.

Este capítulo se concentra nas subclasses de `concurrent.futures.Executor`, que incorporam o modelo descrito por Michele Simionato: "disparar um monte de threads independentes e coletar os resultados em uma fila". Executores concorrentes implementam internamente este modelo, não apenas com threads mas também com processos—que oferecem melhor desempenho em tarefas de processamento intensivas no uso de CPUs.

Também introduzo aqui o conceito de *futures*—objetos que representam a execução assíncrona de uma operação, similares aos *promises* de JavaScript. Esta ideia é a fundação de `concurrent.futures` bem como do pacote `asyncio`, assunto do Capítulo 21.

20.1. Novidades neste capítulo

Mudei o título deste capítulo de "Concorrência com futures" para "Executores concorrentes", porque os executores são o recurso de alto nível mais importante tratado aqui. *Futures* são objetos de baixo nível, tratados na Seção 20.2.3, mas quase invisíveis no resto do capítulo.

Todos os exemplos de clientes HTTP agora usam a biblioteca *HTTPX* [fpy.li/httpx], que oferece APIs síncronas e assíncronas.

Simplifiquei a configuração para os experimentos na Seção 20.5, porque desde o Python 3.7 o pacote `http.server` [fpy.li/20-2] é *multi-thread*. Antes, o `http.server` só usava uma thread, então não servia para experimentos com clientes concorrentes, o que me obrigou a usar um servidor externo na primeira edição.

A Seção 20.3 agora demonstra como um executor simplifica o código que vimos na Seção 19.6.3.

Por fim, movi a maior parte da teoria para o Capítulo 19, *Modelos de concorrência em Python*.

20.2. Downloads concorrentes da Web

A concorrência é essencial para uma comunicação eficiente via rede: em vez de esperar de braços cruzados por respostas de máquinas remotas, a aplicação pode fazer outra coisa enquanto a resposta não chega.

Para demonstrar, escrevi três programas simples que baixam da Web imagens de 20 bandeiras de países. O primeiro, *flags.py*, roda sequencialmente: ele só requisita a imagem seguinte quando a anterior foi baixada e salva localmente. Os outros dois scripts fazem downloads concorrentes: eles requisitam várias imagens quase ao mesmo tempo, e as salvam conforme chegam. O script *flags_threadpool.py* usa o pacote `concurrent.futures`, enquanto *flags_asyncio.py* usa `asyncio`.

Os scripts baixam imagens de *fluentpython.com*, que usa uma CDN (*Content Delivery Network*, Rede de Entrega de Conteúdo), então você pode observar resultados mais lentos nos primeiros testes. Obtive os resultados no Exemplo 1 após vários testes, então o cache da CDN estava "quente" (carregado com os dados).

O Exemplo 1 mostra o resultado da execução dos três scripts, três vezes cada um.

*Exemplo 1. Saida dos scripts *flags.py*, *flags_threadpool.py*, e *flags_asyncio.py**

```
$ python3 flags.py
BD BR CD CN DE EG ET FR ID IN IR JP MX NG PH PK RU TR US VN ①
20 flags downloaded in 7.26s ②
$ python3 flags.py
BD BR CD CN DE EG ET FR ID IN IR JP MX NG PH PK RU TR US VN
20 flags downloaded in 7.20s
$ python3 flags.py
BD BR CD CN DE EG ET FR ID IN IR JP MX NG PH PK RU TR US VN
20 flags downloaded in 7.09s
$ python3 flags_threadpool.py
```

```

DE BD CN JP ID EG NG BR RU CD IR MX US PH FR PK VN IN ET TR
20 flags downloaded in 1.37s ③
$ python3 flags_threadpool.py
EG BR FR IN BD JP DE RU PK PH CD MX ID US NG TR CN VN ET IR
20 flags downloaded in 1.60s
$ python3 flags_threadpool.py
BD DE EG CN ID RU IN VN ET MX FR CD NG US JP TR PK BR IR PH
20 flags downloaded in 1.22s
$ python3 flags_asyncio.py ④
BD BR IN ID TR DE CN US IR PK PH FR RU NG VN ET MX EG JP CD
20 flags downloaded in 1.36s
$ python3 flags_asyncio.py
RU CN BR IN FR BD TR EG VN IR PH CD ET ID NG DE JP PK MX US
20 flags downloaded in 1.27s
$ python3 flags_asyncio.py
RU IN ID DE BR VN PK MX US IR ET EG NG BD FR CN JP PH CD TR ⑤
20 flags downloaded in 1.42s

```

- ① A saída de cada execução começa com os códigos dos países de cada bandeira a medida que as imagens são baixadas, e termina com uma mensagem mostrando o tempo decorrido. Note que as siglas dos países estão em ordem alfabética, para podermos comparar com a ordem dos resultados nas versões concorrentes.
- ② *flags.py* precisou em média de 7,18s para baixar 20 imagens.
- ③ A média para *flags_threadpool.py* foi 1,40s.
- ④ Já *flags_asyncio.py*, obteve um tempo médio de 1,35s.
- ⑤ Note a ordem dos códigos dos países: nos scripts concorrentes, as imagens foram baixadas em ordem diferente a cada vez.

As versões concorrentes são mais de cinco vezes mais rápidas que o script sequencial, mas entre as versões concorrentes não há diferença significativa de desempenho. Nestes exemplos, a tarefa é baixar 20 arquivos, com poucos kilobytes cada um. Se você escalar a tarefa para centenas de downloads, os scripts concorrentes podem superar o código sequencial por um fator de 20 ou mais.



Ao testar clientes HTTP concorrentes acessando servidores Web públicos, você pode acidentalmente lançar um ataque de negação de serviço (DoS, *Denial of Service*, negação de serviço), ou se tornar suspeito de tentar um ataque. No caso do Exemplo 1 não há problema, pois aqueles scripts estão codificados para realizar apenas 20 requisições. Mais adiante neste capítulo usaremos o pacote `http.server` de Python para executar localmente outros testes com centenas de requisições.

Vamos agora estudar as implementações de dois dos scripts testados no Exemplo 1: *flags.py* e *flags_threadpool.py*. Vou deixar o terceiro, *flags_asyncio.py*, para o Capítulo 21, mas queria demonstrar os três juntos para fazer duas observações:

1. Independente dos elementos de concorrência que você use—threads ou corrotinas—haverá um ganho enorme de desempenho sobre código sequencial em operações de E/S de rede, se o script for escrito corretamente.
2. Para clientes HTTP que podem controlar quantas requisições eles fazem, não há diferenças significativas de desempenho entre threads e corrotinas.^[2]

Agora vamos ao código.

20.2.1. Um script de download sequencial

O Exemplo 2 contém a implementação de *flags.py*, o primeiro script que rodamos no Exemplo 1. Não é muito interessante, mas vamos reutilizar a maior parte do código e das configurações para implementar os scripts concorrentes, então ele merece alguma atenção.



Por uma questão didática, o Exemplo 2 não faz tratamento de erros. Vamos tratar exceções em outros exemplos, mas agora vamos focar na estrutura básica do código, para facilitar a comparação deste script com os scripts que usam concorrência.

Exemplo 2. flags.py: script de download sequencial; algumas funções serão reutilizadas pelos outros scripts

```
import time
from pathlib import Path
from typing import Callable

import httpx ①

POP20_CC = ('CN IN US ID BR PK NG BD RU JP '
            'MX PH VN ET EG DE IR TR CD FR').split() ②

BASE_URL = 'https://www.fluentpython.com/data/flags' ③
DEST_DIR = Path('downloaded') ④

def save_flag(img: bytes, filename: str) -> None: ⑤
    (DEST_DIR / filename).write_bytes(img)

def get_flag(cc: str) -> bytes: ⑥
    url = f'{BASE_URL}/{cc}/{cc}.gif'.lower()
    resp = httpx.get(url, timeout=6.1, ⑦
                    follow_redirects=True) ⑧
    resp.raise_for_status() ⑨
    return resp.content

def download_many(cc_list: list[str]) -> int: ⑩
    for cc in sorted(cc_list): ⑪
        image = get_flag(cc)
        save_flag(image, f'{cc}.gif')
        print(cc, end=' ', flush=True) ⑫
    return len(cc_list)

def main(downloader: Callable[[list[str]], int]) -> None: ⑬
    DEST_DIR.mkdir(exist_ok=True) ⑭
    t0 = time.perf_counter() ⑮
    count = downloader(POP20_CC)
    elapsed = time.perf_counter() - t0
    print(f'\n{count} downloads in {elapsed:.2f}s')

if __name__ == '__main__':
    main(download_many) ⑯
```

- ① Importa a biblioteca `httpx`. Ela não é parte da biblioteca padrão. Assim, por convenção, a importação aparece após os módulos da biblioteca padrão e uma linha em branco.
- ② Lista do código de país ISO 3166 para os 20 países mais populosos, em ordem decrescente de população.
- ③ O diretório com as imagens das bandeiras.^[3]
- ④ Diretório local onde as imagens são salvas.
- ⑤ Salva os bytes de `img` em `filename` no `DEST_DIR`.
- ⑥ Dado um código de país, constrói a URL e baixa a imagem, retornando o conteúdo binário da resposta.
- ⑦ É uma boa prática adicionar um timeout razoável para operações de rede, para evitar ficar bloqueado sem motivo por vários minutos.
- ⑧ Por default, o `HTTPX` não segue redirecionamentos.^[4]
- ⑨ Não há tratamento de erros neste script, mas o método `.raise_for_status` lança uma exceção se o status da resposta HTTP não está na faixa 2XX, para evitar falhas silenciosas.
- ⑩ `download_many` é a função chave para comparar com as implementações concorrentes.
- ⑪ Percorre a lista de códigos de país em ordem alfabética, para facilitar a confirmação de que a ordem é preservada na saída; devolve o número de códigos de país baixados.
- ⑫ Mostra um código de país por vez na mesma linha, para vermos o progresso a cada download. O argumento `end=' '` substitui a quebra de linha no final de cada `print` por um espaço, assim todos os códigos de país aparecem na mesma linha. O argumento `flush=True` é necessário porque, por default, o Python usa um buffer de linha na saída padrão, então só após uma quebra de linha o resultado seria visível no terminal. A opção `flush=True` força o esvaziamento do buffer, então podemos ver as siglas aparecendo progressivamente.
- ⑬ `main` precisa ser invocada passando a função que fará os downloads; assim podemos usar `main` como uma função de biblioteca com outras implementações de `download_many` nos exemplos de `threadpool` e `asyncio`.
- ⑭ Cria o `DEST_DIR` se necessário; não acusa erro se o diretório existir.

- ⑮ Registra e apresenta o tempo decorrido após rodar a função `download`.
- ⑯ Invoca `main` com a função `download_many`.



A biblioteca *HTTPX* [fpy.li/httpx] é inspirada no pacote pythônico *requests* [fpy.li/20-5], mas foi desenvolvida sobre bases mais modernas. Em particular, *HTTPX* oferece APIs síncronas e assíncronas, então podemos usá-la em todos os exemplos de clientes HTTP neste capítulo e no próximo.

A biblioteca padrão do Python contém o módulo `urllib.request`, mas sua API é somente síncrona, e não é nada amigável.

Não há nada de novo em *flags.py*, mas serve de base para comparação com outros scripts, e o usei como uma biblioteca, para evitar código redundante ao implementá-los.

Vamos ver agora implementação concorrente usando `concurrent.futures`.

20.2.2. Download com `concurrent.futures`

Os principais recursos do pacote `concurrent.futures` são as classes `ThreadPoolExecutor` e `ProcessPoolExecutor`, que implementam uma API para submeter invocáveis (*callables*) para execução em um banco de threads ou processos (*thread pool* ou *process pool*). As classes gerenciam de forma transparente um banco de threads ou processos de trabalho, e filas para distribuição de tarefas e coleta de resultados. Mas a interface é de um nível muito alto, então não precisamos lidar diretamente com estes componentes em um caso de uso simples como nossos downloads de bandeiras.

O Exemplo 3 mostra a forma mais fácil de implementar os downloads de forma concorrente, usando o método `ThreadPoolExecutor.map`.

Exemplo 3. flags_threadpool.py: script de download com threads, usando futures.ThreadPoolExecutor

```
from concurrent import futures

from flags import save_flag, get_flag, main ①

def download_one(cc: str): ②
    image = get_flag(cc)
    save_flag(image, f'{cc}.gif')
    print(cc, end=' ', flush=True)
    return cc

def download_many(cc_list: list[str]) -> int:
    with futures.ThreadPoolExecutor() as executor: ③
        res = executor.map(download_one, sorted(cc_list)) ④

    return len(list(res)) ⑤

if __name__ == '__main__':
    main(download_many) ⑥
```

- ① Reutiliza algumas funções do módulo flags (Exemplo 2).
- ② Função para baixar uma única imagem; é ela que cada thread de trabalho vai executar.
- ③ Instancia o ThreadPoolExecutor como um gerenciador de contexto; o método `executor.exit` vai invocar `executor.shutdown(wait=True)`, que bloqueia até que todas as threads terminem de rodar.
- ④ O método `map` é similar à função embutida `map`, mas a função `download_one` será chamada de forma concorrente por múltiplas threads; ele devolve um gerador que você pode iterar para recuperar o valor devolvido por cada chamada da função—neste caso, cada chamada a `download_one` vai retornar um código de país.
- ⑤ Devolve o número de resultados obtidos. Se alguma das chamadas das threads levantar uma exceção, aquela exceção será levantada aqui quando a chamada implícita `next()`, dentro do construtor de `list`, tentar recuperar o valor de retorno correspondente, no iterador devolvido por `executor.map`.

- ⑥ Chama a função `main` do módulo `flags`, passando a versão concorrente de `download_many`.

Observe que a função `download_one` do Exemplo 3 é essencialmente o corpo do laço `for` na função `download_many` do Exemplo 2. Esta é uma refatoração comum quando em código concorrente: transformar o corpo de um laço `for` sequencial em uma função a ser chamada de modo concorrente.



O Exemplo 3 é muito curto porque pude reutilizar a maior parte das funções do script sequencial `flags.py`. Uma das melhores características do `concurrent.futures` é facilitar a execução concorrente de código sequencial.

O construtor de `ThreadPoolExecutor` recebe muitos argumentos além dos mostrados aqui, mas o primeiro e mais importante é `max_workers`, definindo o número máximo de threads de trabalho a serem executadas. Quando `max_workers` é `None` (o default), `ThreadPoolExecutor` decide seu valor usando a seguinte expressão, desde o Python 3.8:

```
max_workers = min(32, os.cpu_count() + 4)
```

A justificativa é apresentada na documentação de `ThreadPoolExecutor` [fpy.li/aj]:

Este valor default reserva pelo menos 5 threads de trabalho para tarefas de E/S. Ele utiliza no máximo 32 núcleos da CPU para tarefas de processamento que liberam a GIL. E ele evita usar implicitamente recursos demais em máquinas com muitos núcleos.

ThreadPoolExecutor agora também reutiliza threads de inativas antes de iniciar a quantidade de threads de trabalho definida em `max_workers`.

Concluindo: o valor default calculado de `max_workers` é razoável, e `ThreadPoolExecutor` evita iniciar novas threads de trabalho desnecessariamente. Entender a lógica por trás de `max_workers` pode ajudar a decidir quando e como definir um valor diferente do default em seu código.

A biblioteca chama-se `concurrency.futures`, mas não há qualquer *future* à vista no Exemplo 3, então você pode estar se perguntando onde eles estão. A próxima seção explica.

20.2.3. Onde estão os *futures*?

Os *futures* (literalmente "futuros") são componentes centrais de `concurrent.futures` e de `asyncio`, mas como usuários dessas bibliotecas, raramente os vemos. O Exemplo 3 depende de *futures* por trás do palco, mas o código apresentado não lida diretamente com objetos desta classe. Esta seção apresenta uma visão geral dos *futures*, com um exemplo mostrando-os em ação.

Desde o Python 3.4, há duas classes chamadas `Future` na biblioteca padrão: `concurrent.futures.Future` e `asyncio.Future`. Elas têm o mesmo propósito: uma instância de qualquer uma das classes `Future` representa um processamento adiado, que pode ou não ter sido completado. Isso é algo similar à classe `Deferred` no Twisted, a classe `Future` no Tornado, e objetos `Promise` em JavaScript moderno.

Os *futures* encapsulam operações pendentes de forma que possamos colocá-los em filas, verificar se terminaram, e recuperar resultados (ou exceções) quando eles ficam disponíveis.

É importante entender que eu e você não devemos criar *futures* diretamente: eles devem ser instanciados exclusivamente pelo framework de concorrência, seja ele `concurrent.futures` ou `asyncio`. O motivo é que uma instância de `Future` representa algo que será executado em algum momento, portanto precisa ser agendado para rodar, e quem agenda tarefas é o framework.

Especificamente, instâncias de `concurrent.futures.Future` são criadas como resultado de submeter um objeto invocável (*callable*) para execução a uma subclasse de `concurrent.futures.Executor`. Por exemplo, o método `Executor.submit()` recebe um invocável, agenda sua execução, e devolve um `Future`.

O código da aplicação não deve mudar o estado de um *future*: o framework de concorrência muda o estado de um *future* quando o processamento que ele representa termina, e não controlamos quando isso acontece.

Os dois tipos de *Future* têm um método `.done()` não-bloqueante, que devolve um `bool` informando se o invocável encapsulado por aquele *future* foi ou não executado. Entretanto, em vez de perguntar repetidamente se um *future* terminou, o código cliente em geral pede para ser notificado. Por isso as duas classes *Future* têm um método `.add_done_callback()`: você passa a ele uma função que será invocada com o *future* como único argumento, quando o *future* tiver terminado. Aquela função de callback será invocada na mesma thread ou processo de trabalho que rodou a função encapsulada no *future*.

Há também um método `.result()`, que funciona igual nas duas classes quando a execução do *future* termina: ele devolve o resultado do invocável, ou relança qualquer exceção que possa ter aparecido quando o invocável foi executado. Entretanto, quando o *future* não terminou, o comportamento do método `result` é bem diferente entre os dois sabores de *Future*. Em uma instância de `concurrency.futures.Future`, invocar `f.result()` vai bloquear a thread que chamou até o resultado ficar pronto. Um argumento `timeout` opcional pode ser passado, e se o *future* não tiver terminado após aquele tempo, o método `result` gera um `TimeoutError`. O método `asyncio.Future.result` não suporta um `timeout`, e `await` é a forma preferencial de obter o resultado de *futures* no `asyncio`—mas `await` não funciona com instâncias de `concurrency.futures.Future`.

Várias funções em ambas as bibliotecas devolvem *futures*; outras os usam em sua implementação de uma forma transparente para o usuário. Um exemplo deste último caso é o `Executor.map`, que vimos no Exemplo 3: ele devolve um iterador no qual `__next__` chama o método `result` de cada *future*, então recebemos os resultados dos *futures*, mas não os *futures* em si.

20.2.4. Lidando com *futures*

Para ver uma experiência prática com os *futures*, podemos reescrever o Exemplo 3 para usar a função `concurrent.futures.as_completed` [fpy.li/ak], que recebe um iterável de *futures* e devolve um iterador que entrega *futures* quando cada um encerra sua execução.

Usar `futures.as_completed` exige mudanças apenas na função `download_many`. A chamada ao `executor.map`, de alto nível, é substituída por dois laços `for`: um para criar e agendar os *futures*, o outro para recuperar seus resultados. Já que estamos aqui, vamos acrescentar algumas chamadas a `print` para mostrar cada *future* antes e depois do término de sua execução. O Exemplo 4 mostra o código da nova

função `download_many`. O código de `download_many` aumentou de 5 para 17 linhas, mas agora podemos inspecionar os misteriosos *futures*. As outras funções são idênticas às do Exemplo 3.

Exemplo 4. `flags_threadpool_futures.py`: substitui `executor.map` por `executor.submit` e `futures.as_completed` na função `download_many`

```
def download_many(cc_list: list[str]) -> int:
    cc_list = cc_list[:5] ①
    with futures.ThreadPoolExecutor(max_workers=3) as executor: ②
        to_do: list[futures.Future] = []
        for cc in sorted(cc_list): ③
            future = executor.submit(download_one, cc) ④
            to_do.append(future) ⑤
            print(f'Scheduled for {cc}: {future}') ⑥

        for count, future in enumerate(futures.as_completed(to_do), 1):⑦
            res: str = future.result() ⑧
            print(f'{future} result: {res!r}') ⑨

    return count
```

- ① Para esta demonstração, usa apenas os cinco países mais populosos.
- ② Configura `max_workers` para 3, para podermos ver os *futures* pendentes na saída.
- ③ Itera pelos códigos de país em ordem alfabética, para deixar claro que os resultados vão aparecer fora de ordem.
- ④ `executor.submit` agenda o invocável a ser executado, e devolve o *future* representando essa operação pendente.
- ⑤ Armazena cada *future*, para podermos recuperá-los mais tarde com `as_completed`.
- ⑥ Mostra uma mensagem com o código do país e seu respectivo *future*.
- ⑦ `as_completed` produz *futures* à medida que eles terminam.
- ⑧ Recupera o resultado deste *future*.
- ⑨ Exibe o *future* e seu resultado.

A chamada a `future.result()` nunca bloqueará a thread neste exemplo, pois `future` está vindo de `as_completed`. O Exemplo 5 mostra a saída de uma execução do Exemplo 4.

Exemplo 5. Saída de `flags_threadpool_futures.py`

```
$ python3 flags_threadpool_futures.py
Scheduled for BR: <Future at 0x100791518 state=running> ①
Scheduled for CN: <Future at 0x100791710 state=running>
Scheduled for ID: <Future at 0x100791a90 state=running>
Scheduled for IN: <Future at 0x101807080 state=pending> ②
Scheduled for US: <Future at 0x101807128 state=pending>
CN <Future at 0x100791710 state=finished returned str> result: 'CN' ③
BR ID <Future at 0x100791518 state=finished returned str> result: 'BR' ④
<Future at 0x100791a90 state=finished returned str> result: 'ID'
IN <Future at 0x101807080 state=finished returned str> result: 'IN'
US <Future at 0x101807128 state=finished returned str> result: 'US'

5 downloads in 0.70s
```

- ① Os *futures* são agendados em ordem alfabética; o `repr()` de um `future` mostra seu estado: os três primeiros estão `running`, pois há três threads de trabalho.
- ② Os dois últimos `futures` estão `pending`; esperando pelas threads de trabalho.
- ③ O primeiro CN aqui é a saída de `download_one` em uma thread de trabalho; o resto da linha é a saída de `download_many`.
- ④ Aqui, duas threads exibem códigos antes que `download_many` na thread principal possa mostrar o resultado da primeira thread.



Recomendo brincar com `flags_threadpool_futures.py`. Se você o rodar várias vezes, verá a ordem dos resultados variar. Aumentar `max_workers` para 5 aumentará a variação na ordem dos resultados. Diminuir aquele valor para 1 fará o script rodar sequencialmente, e a ordem dos resultados será sempre a ordem das chamadas a `submit`.

Vimos duas variantes do script de download usando `concurrent.futures`: uma no Exemplo 3 com `ThreadPoolExecutor.map` e uma no Exemplo 4 com `futures.as_completed`. Se tem curiosidade sobre o código de *flags_asyncio.py*, veja o Exemplo 3 do Capítulo 21, onde ele é explicado.

Agora vamos dar uma olhada rápida em um modo simples de evitar a GIL para tarefas que usam a CPU intensivamente, usando `concurrent.futures`.

20.3. Processos com `concurrent.futures`

A página de documentação [fpy.li/am] de `concurrent.futures` tem o subtítulo *Iniciando tarefas em paralelo*. O pacote permite computação paralela em máquinas multi-núcleo porque suporta a distribuição de trabalho entre vários processos Python usando a classe `ProcessPoolExecutor`.

As classes `ProcessPoolExecutor` e `ThreadPoolExecutor` implementam a interface `Executor` [fpy.li/20-9], então é fácil mudar de uma solução baseada em threads para uma baseada em processos usando `concurrent.futures`.

Não há vantagem em usar um `ProcessPoolExecutor` no exemplo de download de bandeiras ou em qualquer tarefa limitada por E/S. É fácil comprovar, alterando as seguintes linhas no Exemplo 3:

```
def download_many(cc_list: list[str]) -> int:
    with futures.ThreadPoolExecutor() as executor:
```

para:

```
def download_many(cc_list: list[str]) -> int:
    with futures.ProcessPoolExecutor() as executor:
```

O construtor de `ProcessPoolExecutor` também tem um parâmetro `max_workers`, que por default é `None`. Nesse caso, o executor limita o número de processos de trabalho ao número resultante de uma chamada a `os.cpu_count()`.

Processos usam mais memória e demoram mais para iniciar que threads, então o real valor de `ProcessPoolExecutor` é em tarefas de uso intensivo da CPU. Vamos voltar ao exemplo de teste de checagem de números primos da Seção 19.6, e reescrevê-lo com `concurrent.futures`.

20.3.1. A volta do checador de primos multi-núcleos

Na Seção 19.6.3, estudamos *procs.py*, um script que checava se alguns números grandes eram primos usando `multiprocessing`. No Exemplo 6 resolvemos o mesmo problema com o programa *proc_pool.py*, usando um `ProcessPool.Executor`. Do primeiro `import` até a chamada a `main()` no final, *procs.py* tem 43 linhas de código não-vazias, e *proc_pool.py* tem 31—uma redução de 28%.

Exemplo 6. proc_pool.py: procs.py reescrito com ProcessPoolExecutor

```
import sys
from concurrent import futures ①
from time import perf_counter
from typing import NamedTuple

from primes import is_prime, NUMBERS

class PrimeResult(NamedTuple): ②
    n: int
    flag: bool
    elapsed: float

def check(n: int) -> PrimeResult:
    t0 = perf_counter()
    res = is_prime(n)
    return PrimeResult(n, res, perf_counter() - t0)

def main() -> None:
    if len(sys.argv) < 2:
        workers = None ③
    else:
        workers = int(sys.argv[1])

    executor = futures.ProcessPoolExecutor(workers) ④
    actual_workers = executor._max_workers # type: ignore ⑤
```

```

print(f'Checking {len(NUMBERS)} numbers',
      f'with {actual_workers} processes:')

t0 = perf_counter()

numbers = sorted(NUMBERS, reverse=True) ⑥
with executor: ⑦
    for n, prime, elapsed in executor.map(check, numbers): ⑧
        label = 'P' if prime else ' '
        print(f'{n:16} {label} {elapsed:9.6f}s')

time = perf_counter() - t0
print(f'Total time: {time:.2f}s')

if __name__ == '__main__':
    main()

```

- ① Não precisamos importar multiprocessing, SimpleQueue etc.; concurrent.futures esconde tudo isso.
- ② A tupla PrimeResult e a função check são as mesmas que vimos em *procs.py*, mas não precisamos mais das filas nem da função worker.
- ③ Em vez de definir quantos processos de trabalho serão usados se um argumento não for passado na linha de comando, atribuímos None a workers e deixamos o ProcessPoolExecutor decidir.
- ④ Aqui criei o ProcessPoolExecutor antes do bloco with em ⑦, para poder mostrar o número real de processos na próxima linha.
- ⑤ max_workers é um atributo de instância não documentado de um ProcessPoolExecutor. Decidi usá-lo para mostrar o número de processos de trabalho criados quando a variável workers é None. O Mypy reclama quando eu acesso esse atributo, então coloquei o comentário type: ignore para silenciar o aviso.
- ⑥ Ordena os números a serem checados em ordem descendente. Isto vai mostrar a diferença no comportamento de *proc_pool.py* quando comparado a *procs.py*. Veja a explicação após esse exemplo.
- ⑦ Usa o executor como um gerenciador de contexto.

- ⑧ A chamada a executor `.map` devolve as instâncias de `PrimeResult` retornadas por `check` na mesma ordem dos argumentos `numbers`.

Se você rodar o Exemplo 6, verá os resultados aparecendo em ordem rigorosamente descendente, como mostrado no Exemplo 7. Por outro lado, a ordem da saída de *procs.py* (da Seção 19.6.1) é determinada pela dificuldade em checar se cada número é ou não primo. Por exemplo, *procs.py* mostra o resultado para 7777777777777777 próximo ao topo, pois ele tem 7 como divisor, então `is_prime` rapidamente determina que ele não é primo.

O número 7777777536340681 é 88191709². Como este fator é grande, `is_prime` vai demorar mais para determinar que é um número composto, e ainda mais para descobrir que 777777777777753 é primo. Por isso esses números aparecem perto do final da saída do *procs.py* original.

Ao rodar *proc_pool.py*, podemos observar não apenas a ordem descendente dos resultados, mas também que o programa parece emperrar após mostrar o resultado de 9999999999999999.

Exemplo 7. Saída de *proc_pool.py*

```
$ ./proc_pool.py
Checking 20 numbers with 12 processes:
9999999999999999      0.000024s ①
9999999999999917 P  9.500677s ②
7777777777777777      0.000022s ③
7777777777777753 P  8.976933s
7777777536340681      8.896149s
6666667141414921      8.537621s
6666666666666719 P  8.548641s
6666666666666666      0.000002s
5555555555555555      0.000017s
5555555555555503 P  8.214086s
5555553133149889      8.067247s
4444444448888889      7.546234s
4444444444444444      0.000002s
4444444444444423 P  7.622370s
3333335652092209      6.724649s
3333333333333333      0.000018s
3333333333333301 P  6.655039s
299593572317531 P  2.072723s
```

```
142702110479723 P 1.461840s
                2 P 0.000001s
Total time: 9.65s
```

- ① Esta linha aparece muito rápido.
- ② Esta linha demora mais de 9,5s para aparecer.
- ③ As linhas restantes aparecem quase imediatamente.

Agora vamos entender o comportamento estranho de *proc_pool.py*:

- Como mencionado antes, `executor.map(check, numbers)` devolve o resultado na mesma ordem em que `numbers` é submetido.
- Por default, *proc_pool.py* usa um número de processos de trabalho igual ao número de CPUs—isso é o que `ProcessPoolExecutor` faz quando `max_workers` é `None`. Neste laptop foram 12 processos.
- Como estamos submetendo `numbers` em ordem descendente, o primeiro é 9999999999999999; com 9 como divisor, ele retorna rapidamente.
- O segundo número é 9999999999999917, o maior número primo na amostra. Ele vai demorar mais que todos os outros para checar.
- Enquanto isso, os 11 processos restantes estarão checando outros números, que são ou primos ou compostos com fatores grandes ou compostos com fatores muito pequenos.
- Quando o processo de trabalho encarregado de 9999999999999917 finalmente determina que ele é primo, todos os outros processos já completaram suas últimas tarefas, então os resultados aparecem logo depois.



Apesar do progresso de *proc_pool.py* não ser tão visível quanto o de *procs.py* na Seção 19.6.3, o tempo total de execução é praticamente igual ao retratado na Figura 2 do Capítulo 19, para as mesmas quantidades de processos de trabalho e núcleos de CPU.

Entender como programas concorrentes se comportam não é fácil, então aqui está um segundo experimento que pode ajudar a visualizar o funcionamento de `Executor.map`.

20.4. Experimento com Executor .map

Vamos investigar `Executor.map`, agora usando um `ThreadPoolExecutor` com três threads de trabalho rodando cinco invocáveis que exibem mensagens com data/hora. O código está no Exemplo 8, o resultado no Exemplo 9.

Exemplo 8. `demo_executor_map.py`: Uma demonstração simples do método `map` de `ThreadPoolExecutor`

```
from time import sleep, strftime
from concurrent import futures

def display(*args): ①
    print(strftime('%H:%M:%S'), end=' ')
    print(*args)

def loiter(n): ②
    msg = '{}loiter({}): napping for {}s...'
    display(msg.format('\t'*n, n, n))
    sleep(n)
    msg = '{}loiter({}): done.'
    display(msg.format('\t'*n, n))
    return n + 10 ③

def main():
    display('Script starting.')
    executor = futures.ThreadPoolExecutor(max_workers=3) ④
    results = executor.map(loiter, range(5)) ⑤
    display('results:', results) ⑥
    display('Waiting for individual results:')
    for i, result in enumerate(results): ⑦
        display(f'result {i}: {result}')

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

- ① Esta função exibe o momento da execução no formato `[HH:MM:SS]` e os argumentos recebidos.
- ② `loiter` significa "vadiar"; esta função não faz nada além mostrar uma mensagem quando inicia, cochilar por `n` segundos, e mostrar uma mensagem

quando termina; usei tabulações para indentar as mensagens de acordo com o valor de `n`

- ③ `loiter` devolve `n + 10`, para que o valor devolvido seja diferente do argumento `n`, tornando mais visível o fluxo de dados.
- ④ Cria um `ThreadPoolExecutor` com três threads.
- ⑤ Submete cinco tarefas para o executor. Como há três threads, apenas três daquelas tarefas vão iniciar imediatamente: a chamadas `loiter(0)`, `loiter(1)`, e `loiter(2)`; `executor.map` não bloqueia, retorna imediatamente.
- ⑥ Exibe `results` devolvido por `executor.map`: é um gerador, como se vê na saída no Exemplo 9.
- ⑦ A chamada `enumerate` no laço `for` invocará implicitamente `next(results)`, que por sua vez invocará `f.result()` no *future* (interno) `f`, representando a primeira chamada, `loiter(0)`. O método `result` bloqueará a thread até que o *future* termine, portanto cada volta nesse laço vai esperar até que o próximo resultado esteja disponível.

Sugiro que você rode o Exemplo 8 para ver o resultado sendo exibido incrementalmente.

Para experimentar, altere o argumento `max_workers` do `ThreadPoolExecutor`. Mude também a chamada a `range`, que produz os argumentos para invocar `executor.map`—ou forneça uma lista de valores escolhidos, para criar intervalos diferentes.

O Exemplo 9 mostra um teste do Exemplo 8.

Exemplo 9. Amostra da execução de `demo_executor_map.py`, do Exemplo 8

```
$ python3 demo_executor_map.py
[15:56:50] Script starting. ①
[15:56:50] loiter(0): napping for 0s... ②
[15:56:50] loiter(0): done.
[15:56:50]     loiter(1): napping for 1s... ③
[15:56:50]           loiter(2): napping for 2s...
[15:56:50] results: <generator object result_iterator at 0x106517168> ④
[15:56:50]           loiter(3): napping for 3s... ⑤
[15:56:50] Waiting for individual results:
[15:56:50] result 0: 10 ⑥
```

```

[15:56:51]      loiter(1): done. ⑦
[15:56:51]                                loiter(4): napping for 4s...
[15:56:51] result 1: 11 ⑧
[15:56:52]      loiter(2): done. ⑨
[15:56:52] result 2: 12
[15:56:53]                                loiter(3): done.
[15:56:53] result 3: 13
[15:56:55]                                loiter(4): done. ⑩
[15:56:55] result 4: 14

```

- ① Este teste começou em 15:56:50.
- ② A primeira thread executa `loiter(0)`, então vai cochilar por 0s e retornar antes mesmo da segunda thread ter chance de começar, mas YMMV.^[5]
- ③ `loiter(1)` e `loiter(2)` começam imediatamente (como o banco de threads tem três threads de trabalho, é possível rodar três funções concorrentemente).
- ④ Isto mostra que `results` devolvido por `executor.map` é um gerador: nada até aqui é bloqueante, independente do número de tarefas e do valor de `max_workers`.
- ⑤ Como `loiter(0)` terminou, a primeira thread de trabalho está disponível para processar `loiter(3)`.
- ⑥ Aqui é onde a execução pode ser bloqueada, dependendo dos parâmetros passados nas chamadas a `loiter`: o método `__next__` do gerador `results` precisa esperar até o primeiro *future* completar. Neste caso, ele não vai bloquear porque a chamada a `loiter(0)` terminou antes deste laço iniciar. Observe que tudo até aqui aconteceu dentro do mesmo segundo: 15:56:50.
- ⑦ `loiter(1)` termina um segundo depois, em 15:56:51. A segunda thread está livre para iniciar `loiter(4)`.
- ⑧ O resultado de `loiter(1)` é exibido: 11. Agora o laço `for` ficará bloqueado, esperando o resultado de `loiter(2)`.
- ⑨ O padrão se repete: `loiter(2)` terminou, seu resultado é exibido; o mesmo ocorre com `loiter(3)`.
- ⑩ Há um intervalo de 2s até `loiter(4)` terminar, porque ela começou em 15:56:51 e cochilou por 4s.

A função `Executor.map` é fácil de usar, mas muitas vezes é preferível obter o resultado de cada tarefa assim que esteja pronta, e não necessariamente na ordem em que foram submetidas. Para isso, precisamos de uma combinação do método `Executor.submit` e da função `futures.as_completed` como vimos no Exemplo 4. Vamos voltar a essa técnica na Seção 20.5.2.



A combinação de `Executor.submit` e `futures.as_completed` é mais flexível que `executor.map`, pois você pode submeter invocáveis diferentes e argumentos diferentes. Já `executor.map` é projetado para rodar o mesmo invocável com argumentos diferentes. Além disso, o conjunto de *futures* que você passa para `futures.as_completed` pode vir de mais de um executor—talvez alguns tenham sido criados por uma instância de `ThreadPoolExecutor` enquanto outros vêm de um `ProcessPoolExecutor`.

Na próxima seção vamos retomar os exemplos de download de bandeiras com novos requisitos que vão nos obrigar a iterar sobre os resultados de `futures.as_completed` em vez de usar `executor.map`.

20.5. Barra de progresso e tratamento de erros

Como mencionado, os scripts na Seção 20.2 não têm tratamento de erros, para torná-los mais fáceis de ler e para comparar a estrutura das três abordagens: sequencial, com threads e assíncrona.

Para testar o tratamento de uma variedade de condições de erro, criei os exemplos `flags2`:

`flags2_common.py`

Este módulo contém as funções e configurações comuns, usadas por todos os exemplos da série `flags2`, incluindo a função `main`, que cuida de ler os argumentos da linha de comando, medir o tempo e mostrar os resultados. Isto é código de apoio, sem relevância direta para o assunto desse capítulo, então não vou incluir o código-fonte aqui, mas você pode vê-lo no «repositório de exemplos» [fpy.li/code], arquivo `20-executors/getflags/flags2_common.py` [fpy.li/20-10].

flags2_sequential.py

Um cliente HTTP sequencial com tratamento de erro correto e a exibição de uma barra de progresso. Sua função `download_one` também é usada por `flags2_threadpool.py`.

flags2_threadpool.py

Cliente HTTP concorrente, baseado em `futures.ThreadPoolExecutor`, para demonstrar o tratamento de erros e a integração da barra de progresso.

flags2_asyncio.py

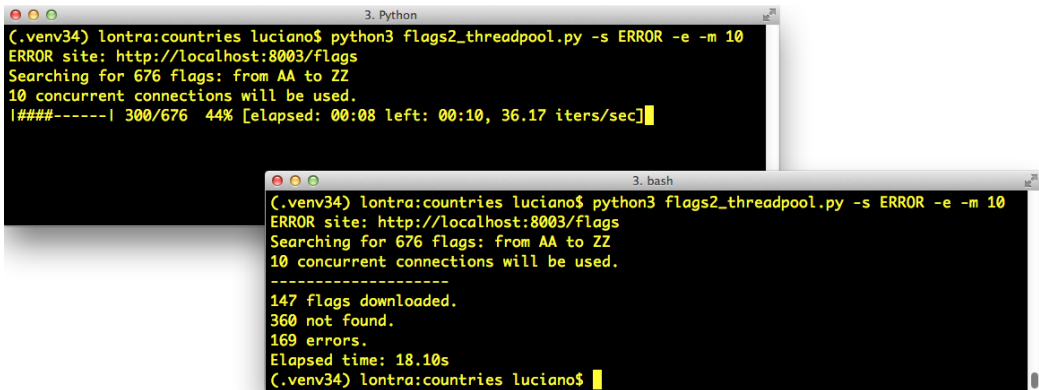
Mesma funcionalidade do exemplo anterior, mas implementado com `asyncio` e `httpx`. Será explicado na Seção 21.7, no Capítulo 21.



Tenha cuidado ao testar clientes concorrentes

Ao testar clientes HTTP concorrentes em servidores Web públicos, você pode gerar muitas requisições por segundo, e é assim que ataques de negação de serviço (DoS, *denial-of-service*) são feitos. Monitore cuidadosamente seus clientes quando for usar servidores públicos. Para testar, configure um servidor HTTP local. Veja instruções após o Exemplo 10.

A característica mais visível dos exemplos `flags2` é sua barra de progresso animada em modo texto, implementada com o pacote `tqdm` [fpy.li/20-11]. Publiquei no YouTube um «vídeo de 108s» [fpy.li/20-12] mostrando a barra de progresso e comparando a velocidade dos três scripts `flags2`. No vídeo, começo com o `download` sequencial, mas interrompo a execução após 32s. O script demoraria mais de 5 minutos para acessar 676 URLs e baixar 194 bandeiras. Então rodo o script que usa `threads` e o script que usa `asyncio`, três vezes cada um, e todas as vezes eles completam a tarefa em 6s ou menos (isto é mais de 60 vezes mais rápido). A Figura 1 mostra duas capturas de tela: durante e após a execução de `flags2_threadpool.py`.



```
3. Python
(.venv34) lontra:countries luciano$ python3 flags2_threadpool.py -s ERROR -e -m 10
ERROR site: http://localhost:8003/flags
Searching for 676 flags: from AA to ZZ
10 concurrent connections will be used.
|###-----| 300/676 44% [elapsed: 00:08 left: 00:10, 36.17 iters/sec]

3. bash
(.venv34) lontra:countries luciano$ python3 flags2_threadpool.py -s ERROR -e -m 10
ERROR site: http://localhost:8003/flags
Searching for 676 flags: from AA to ZZ
10 concurrent connections will be used.
-----
147 flags downloaded.
360 not found.
169 errors.
Elapsed time: 18.10s
(.venv34) lontra:countries luciano$
```

Figura 1. Acima: `flags2_threadpool.py` rodando com a barra de progresso em tempo real gerada pelo `tqdm`; abaixo: mesma janela do terminal após o script terminar de rodar.

O exemplo de uso mais simples do `tqdm` aparece em um `.gif` animado, no `README.md` [fpy.li/20-13] do projeto. Se você digitar o código abaixo no console de Python após instalar o pacote `tqdm`, uma barra de progresso animada aparecerá no lugar onde está o comentário:

```
>>> import time
>>> from tqdm import tqdm
>>> for i in tqdm(range(1000)):
...     time.sleep(.01)
...
>>> # -> a barra de progresso aparece aqui <-
```

Além do efeito bonitinho, o gerador `tqdm` também é interessante conceitualmente: ele consome qualquer iterável, e produz um iterador que, enquanto é consumido, mostra a barra de progresso e estima o tempo restante para completar todas as iterações. Para fazer a estimativa, o `tqdm` precisa receber um iterável que implemente `len`, ou o argumento `total=` com o número esperado de itens. Integrar o `tqdm` com nossos exemplos `flags2` permite observar mais profundamente o funcionamento real dos scripts concorrentes, obrigando-nos a usar as funções `futures.as_completed` [fpy.li/20-7] e `asyncio.as_completed` [fpy.li/20-15], para que o `tqdm` atualize a barra de progresso conforme cada `future` termina.

A outra característica dos exemplos `flags2` é a interface de linha de comando. Todos os três scripts aceitam várias opções para experimentar, que você pode ver

passando a opção -h. O Exemplo 10 mostra o texto de ajuda.

Exemplo 10. Tela de ajuda dos scripts da série flags2

```
$ python3 flags2_threadpool.py -h
usage: flags2_threadpool.py [-h] [-a] [-e] [-l N] [-m CONCURRENT] [-s
LABEL]
                                [-v]
                                [CC [CC ...]]

Download flags for country codes. Default: top 20 countries by
population.

positional arguments:
  CC                      country code or 1st letter (eg. B for BA...BZ)

optional arguments:
  -h, --help              show this help message and exit
  -a, --all               get all available flags (AD to ZW)
  -e, --every             get flags for every possible code (AA...ZZ)
  -l N, --limit N         limit to N first codes
  -m CONCURRENT, --max_req CONCURRENT
                          maximum concurrent requests (default=30)
  -s LABEL, --server LABEL
                          Server to hit; one of DELAY, ERROR, LOCAL, REMOTE
                          (default=LOCAL)
  -v, --verbose           output detailed progress info
```

Todos os argumentos são opcionais. Mas o -s/--server é essencial para os testes: ele permite escolher qual servidor HTTP e qual porta serão usados no teste. Passe um desses parâmetros (insensíveis a maiúsculas/minúsculas) para determinar onde o script vai buscar as bandeiras:

LOCAL

Usa localhost:8000/flags; esse é o default. Você deve configurar um servidor HTTP local, respondendo na porta 8000. Veja as instruções na nota a seguir.

REMOTE

Usa fluentpython.com/data/flags; este é meu site público, hospedado em um servidor compartilhado. Por favor, não o martele com requisições

concorrentes excessivas. O domínio *fluentpython.com* é gerenciado pela CDN da *Cloudflare* [fpy.li/20-16], então você pode notar que os primeiros downloads são mais lentos, mas ficam mais rápidos conforme o cache da CDN é carregado.

DELAY

Usa `localhost:8001/flags`; um servidor atrasando as respostas HTTP deve responder na porta 8001. Escrevi o *slow_server.py* para facilitar o experimento. Ele está no diretório *20-futures/getflags/* do repositório de código do *Python Fluente* [fpy.li/code]. Veja as instruções na nota a seguir.

ERROR

Usa `localhost:8002/flags`; um servidor devolvendo alguns erros HTTP deve responder na porta 8002. Instruções a seguir.

Configurando os servidores de teste

Escrevi instruções para subir servidores HTTP para testes usando apenas Python ≥ 3.9 (nenhuma biblioteca externa) em *20-executors/getflags/README.adoc* [fpy.li/20-17] no «repositório de exemplos» [fpy.li/code]. Em resumo, o *README.adoc* descreve como rodar:



```
python3 -m http.server
```

O servidor LOCAL na porta 8000

```
python3 slow_server.py
```

O servidor DELAY na porta 8001, que acrescenta um atraso aleatório de 0,5s a 5s antes de cada resposta

```
python3 slow_server.py 8002 --error-rate .25
```

O servidor ERROR na porta 8002, que além do atraso aleatório tem uma chance de 25% de retornar um erro "418 I'm a teapot" [fpy.li/20-18] como resposta

Por padrão, cada script *flags2*.py* irá baixar as bandeiras dos 20 países mais populosos do servidor LOCAL (localhost:8000/flags), usando um número default de conexões concorrentes, que varia de script para script. O Exemplo 11 mostra uma execução padrão do script *flags2_sequential.py* usando as configurações default. Para rodá-lo, você precisa de um servidor local, como explicado acima.

Exemplo 11. Rodando flags2_sequential.py com todos os defaults: site LOCAL, as 20 bandeiras dos países mais populosos, 1 conexão concorrente

```
$ python3 flags2_sequential.py
LOCAL site: http://localhost:8000/flags
Searching for 20 flags: from BD to VN
1 concurrent connection will be used.
-----
20 flags downloaded.
Elapsed time: 0.10s
```

Você pode selecionar as bandeiras a serem baixadas de várias formas. O Exemplo 12 mostra como baixar todas as bandeiras com códigos de país começando pelas letras A, B ou C.

Exemplo 12. Roda flags2_threadpool.py para obter do servidor DELAY todas as bandeiras com prefixos de códigos de país A, B ou C

```
$ python3 flags2_threadpool.py -s DELAY a b c
DELAY site: http://localhost:8001/flags
Searching for 78 flags: from AA to CZ
30 concurrent connections will be used.
-----
43 flags downloaded.
35 not found.
Elapsed time: 1.72s
```

Independente de como os códigos de país são selecionados, o número de bandeiras a serem obtidas pode ser limitado com a opção `-l/--limit`. O Exemplo 13 demonstra como fazer exatamente 100 requisições, combinando a opção `-a` para obter todas as bandeiras com `-l 100`.

Exemplo 13. Roda `flags2_asyncio.py` para baixar 100 bandeiras (-al 100) do servidor ERROR, usando 100 requisições concorrentes (-m 100)

```
$ python3 flags2_asyncio.py -s ERROR -al 100 -m 100
ERROR site: http://localhost:8002/flags
Searching for 100 flags: from AD to LK
100 concurrent connections will be used.
-----
73 flags downloaded.
27 errors.
Elapsed time: 0.64s
```

Vimos a interface de usuário dos exemplos *flags2*. Agora veremos como eles estão implementados.

20.5.1. Tratamento de erros nos exemplos *flags2*

A estratégia comum nos três exemplos para lidar com erros HTTP é que erros 404 (not found) são tratados pela função encarregada de baixar um único arquivo (`download_one`). Outras exceções são tratadas pela função `download_many` ou pela corrotina `supervisor`—no exemplo com `asyncio`.

Vamos novamente começar estudando o código sequencial, que é mais fácil de compreender, e boa parte dele será reutilizado pelo script com um banco de threads. O Exemplo 14 mostra as funções que efetivamente fazem os downloads nos scripts *flags2_sequential.py* e *flags2_threadpool.py*.

Exemplo 14. `flags2_sequential.py`: funções básicas encarregadas dos downloads; ambas são reutilizadas no `flags2_threadpool.py`

```
from collections import Counter
from http import HTTPStatus

import httpx
import tqdm # type: ignore ①

from flags2_common import main, save_flag, DownloadStatus ②

DEFAULT_CONCUR_REQ = 1
MAX_CONCUR_REQ = 1
```

```

def get_flag(base_url: str, cc: str) -> bytes:
    url = f'{base_url}/{cc}/{cc}.gif'.lower()
    resp = httpx.get(url, timeout=3.1, follow_redirects=True)
    resp.raise_for_status() ③
    return resp.content

def download_one(
    cc: str,
    base_url: str,
    verbose: bool = False) -> DownloadStatus:
    try:
        image = get_flag(base_url, cc)
    except httpx.HTTPStatusError as exc: ④
        res = exc.response
        if res.status_code == HTTPStatus.NOT_FOUND:
            status = DownloadStatus.NOT_FOUND ⑤
            msg = f'not found: {res.url}'
        else:
            raise ⑥
    else:
        save_flag(image, f'{cc}.gif')
        status = DownloadStatus.OK
        msg = 'OK'

    if verbose: ⑦
        print(cc, msg)

    return status

```

- ① Importa a biblioteca de exibição de barra de progresso tqdm, e diz ao Mypy para não checá-la.^[6]
- ② Importa algumas funções e um Enum do módulo `flags2_common`.
- ③ Dispara um `HTTPStatusError` se o código de status do HTTP não está em `range(200, 300)`.
- ④ `download_one` trata o `HTTPStatusError`, especificamente para tratar o código HTTP 404...
- ⑤ ...mudando seu status local para `DownloadStatus.NOT_FOUND`; `DownloadStatus` é um Enum importado de `flags2_common.py`.

- ⑥ Qualquer outra exceção de `HTTPStatusError` é re-emitida e propagada para quem chamou a função.
- ⑦ Se a opção de linha de comando `-v/--verbose` está vigente, o código do país e a mensagem de status são exibidos; é assim que você verá o progresso no modo `verbose`.

O Exemplo 15 é a versão sequencial de `download_many`. O código é simples, mas vale a pena estudar para compará-lo com as versões concorrentes que veremos depois. Note como ele exibe o progresso, trata erros e conta os downloads.

Exemplo 15. `flags2_sequential.py`: a implementação sequencial de `download_many`

```
def download_many(cc_list: list[str],
                  base_url: str,
                  verbose: bool,
                  _unused_concur_req: int) -> Counter[DownloadStatus]:
    counter: Counter[DownloadStatus] = Counter() ①
    cc_iter = sorted(cc_list) ②
    if not verbose:
        cc_iter = tqdm.tqdm(cc_iter) ③
    for cc in cc_iter:
        try:
            status = download_one(cc, base_url, verbose) ④
        except httpx.HTTPStatusError as exc: ⑤
            error_msg = ('HTTP error {resp.status_code}' +
                        ' - {resp.reason_phrase}')
            error_msg = error_msg.format(resp=exc.response)
        except httpx.RequestError as exc: ⑥
            error_msg = f'{exc} {type(exc)}'.strip()
        except KeyboardInterrupt: ⑦
            break
        else: ⑧
            error_msg = ''

        if error_msg:
            status = DownloadStatus.ERROR ⑨
        counter[status] += 1 ⑩
        if verbose and error_msg: ⑪
            print(f'{cc} error: {error_msg}')

    return counter ⑫
```


- ① Este Counter vai registrar os diferentes resultados possíveis dos downloads: `DownloadStatus.OK`, `DownloadStatus.NOT_FOUND`, ou `DownloadStatus.ERROR`.
- ② `cc_iter` preserva a lista de códigos de país recebidos como argumentos, em ordem alfabética.
- ③ Se não estamos rodando em modo `verbose`, `cc_iter` é passado para o `tqdm`, que devolve um iterador que produz os itens em `cc_iter` enquanto também anima a barra de progresso.
- ④ Faz chamadas sucessivas a `download_one`.
- ⑤ As exceções do código de status HTTP ocorridas em `get_flag` e não tratadas por `download_one` são tratadas aqui.
- ⑥ Outras exceções referentes à rede são tratadas aqui. Qualquer outra exceção vai interromper o script, porque a função `flags2_common.main`, que chama `download_many`, não tem um `try/except`.
- ⑦ Sai do laço se o usuário pressionar CTRL-C.
- ⑧ Se nenhuma exceção saiu de `download_one`, limpa a mensagem de erro.
- ⑨ Se houve um erro, muda o status local de acordo com o erro.
- ⑩ Incrementa o contador para aquele status.
- ⑪ No modo `verbose`, mostra a mensagem de erro do código de país atual, se houver.
- ⑫ Devolve counter para que `main` possa mostrar os números no relatório final.

Agora vamos estudar *flags2_threadpool.py*, o exemplo com banco de threads aperfeiçoado.

20.5.2. Usando `futures.as_completed`

Para integrar a barra de progresso do *tqdm* e tratar os erros a cada requisição, o script *flags2_threadpool.py* usa o `futures.ThreadPoolExecutor` com a função `futures.as_completed`. O Exemplo 16 é a listagem completa de *flags2_threadpool.py*. Apenas a função `download_many` é implementada; as outras funções são reutilizadas de *flags2_common.py* e *flags2_sequential.py*.

Exemplo 16. flags2_threadpool.py: listagem completa

```
from collections import Counter
from concurrent.futures import ThreadPoolExecutor, as_completed

import httpx
import tqdm # type: ignore

from flags2_common import main, DownloadStatus
from flags2_sequential import download_one ①

DEFAULT_CONCUR_REQ = 30 ②
MAX_CONCUR_REQ = 1000 ③

def download_many(cc_list: list[str],
                  base_url: str,
                  verbose: bool,
                  concur_req: int) -> Counter[DownloadStatus]:
    counter: Counter[DownloadStatus] = Counter()
    with ThreadPoolExecutor(max_workers=concur_req) as executor: ④
        to_do_map = {} ⑤
        for cc in sorted(cc_list): ⑥
            future = executor.submit(download_one, cc,
                                     base_url, verbose) ⑦
            to_do_map[future] = cc ⑧
        done_iter = as_completed(to_do_map) ⑨
        if not verbose:
            done_iter = tqdm.tqdm(done_iter, total=len(cc_list)) ⑩
        for future in done_iter: ⑪
            try:
                status = future.result() ⑫
            except httpx.HTTPStatusError as exc: ⑬
                error_msg = ('HTTP error {resp.status_code}' +
                             ' - {resp.reason_phrase}')
                error_msg = error_msg.format(resp=exc.response)
            except httpx.RequestError as exc:
                error_msg = f'{exc} {type(exc)}'.strip()
            except KeyboardInterrupt:
                break
            else:
                error_msg = ''
```

```

        if error_msg:
            status = DownloadStatus.ERROR
            counter[status] += 1
        if verbose and error_msg:
            cc = to_do_map[future] ⑭
            print(f'{cc} error: {error_msg}')

    return counter

if __name__ == '__main__':
    main(download_many, DEFAULT_CONCUR_REQ, MAX_CONCUR_REQ)

```

- ① Reutiliza `download_one` de `flags2_sequential` (Exemplo 14).
- ② Se a opção de linha de comando `-m/--max_req` não é passada, este será o número máximo de requisições concorrentes, definido como o tamanho do banco de threads; o número real pode ser menor se o número de bandeiras a serem baixadas for menor.
- ③ `MAX_CONCUR_REQ` limita o número máximo de requisições concorrentes independente do número de bandeiras a serem baixadas ou da opção de linha de comando `-m/--max_req`. É uma medida de segurança, para evitar iniciar threads demais, com seu uso significativo de memória.
- ④ Cria o executor com `max_workers` determinado por `concur_req`, calculado pela função `main` como o menor valor entre `MAX_CONCUR_REQ`, o tamanho de `cc_list`, ou a opção de linha de comando `-m/--max_req`. Assim evitamos criar mais threads que o necessário.
- ⑤ Este dict vai mapear cada instância de `Future`—representando um download—com o respectivo código de país, para exibição de erros.
- ⑥ Itera sobre a lista de códigos de país em ordem alfabética. A ordem dos resultados vai depender, mais do que de qualquer outra coisa, do tempo das respostas HTTP; mas se o tamanho do banco de threads (dado por `concur_req`) for muito menor que `len(cc_list)`, você poderá ver os downloads aparecendo em ordem alfabética.
- ⑦ Cada chamada a `executor.submit` agenda a execução de um invocável e devolve uma instância de `Future`. O primeiro argumento é o invocável, o restante são os argumentos que ele receberá.

- ⑧ Armazena o `future` e o código de país no `dict`.
- ⑨ `futures.as_completed` devolve um iterador que produz *futures* conforme cada tarefa é completada.
- ⑩ Se não estiver no modo `verbose`, passa o resultado de `as_completed` para a função `tqdm`, para mostrar a barra de progresso; como `done_iter` não tem `len`, precisamos informar o gerador `tqdm` qual o número de itens esperado com o argumento `total=`, para que ele possa estimar o trabalho restante.
- ⑪ Itera sobre os *futures* conforme eles vão terminando.
- ⑫ Chamar o método `result` em um *future* devolve o valor devolvido pela invocável ou dispara qualquer exceção que tenha sido capturada quando o invocável foi executado. Este método pode bloquear, esperando por uma resolução. Mas não nesse exemplo, porque `as_completed` só produz *futures* que terminaram sua execução.
- ⑬ Trata exceções em potencial; o resto desta função é idêntica à função `download_many` no Exemplo 15, exceto pela observação a seguir.
- ⑭ Para dar contexto à mensagem de erro, recupera o código de país do `to_do_map`, usando o `future` atual como chave. Isso não era necessário na versão sequencial, pois estávamos iterando sobre a lista de códigos de país, então sabíamos qual era o `cc` atual; aqui estamos iterando sobre *futures*.



O Exemplo 16 usa um padrão que é muito útil com `futures.as_completed`: construir um `dict` mapeando cada *future* a outros dados que podem ser úteis quando o *future* terminar de executar. Aqui o `to_do_map` mapeia cada *future* ao código de país atribuído a ele. Assim fica mais fácil o pós-processamento dos resultados dos *futures*, apesar deles serem produzidos fora de ordem.

As `threads` de Python são bastante adequadas a aplicações de uso intensivo de E/S, e o pacote `concurrent.futures` as torna relativamente simples de implementar em certos casos de uso. Com `ProcessPoolExecutor` você também pode resolver problemas de uso intensivo de CPU em múltiplos núcleos—se o processamento for “embarçosamente paralelo” [fpy.li/20-19]. Isso encerra nossa introdução básica a `concurrent.futures`.

20.6. Resumo do capítulo

Começamos o capítulo comparando dois clientes HTTP concorrentes com um sequencial, demonstrando que as soluções concorrentes mostram um ganho significativo de desempenho sobre o script sequencial.

Após estudar o primeiro exemplo, baseado no `concurrent.futures`, olhamos mais de perto os objetos *future*, instâncias de `concurrent.futures.Future` ou de `asyncio.Future`, enfatizando as semelhanças entre essas classes (suas diferenças serão examinadas no Capítulo 21). Vimos como criar *futures* chamando `Executor.submit`, e como iterar sobre *futures* que terminaram sua execução com `concurrent.futures.as_completed`.

Então discutimos o uso de múltiplos processos com a classe `concurrent.futures.ProcessPoolExecutor`, para evitar a GIL e usar múltiplos núcleos de CPU, simplificando o checador de números primos multi-núcleo que vimos antes no Capítulo 19.

Na seção seguinte, estudamos como funciona a classe `ThreadPoolExecutor`, com um exemplo didático, iniciando tarefas que não faziam nada por alguns segundos, exceto exibir seu status e a hora naquele instante.

Então voltamos para os exemplos de download de bandeiras. Melhorar aqueles exemplos com uma barra de progresso e tratamento de erro adequado nos ajudou a explorar melhor a função geradora `future.as_completed` mostrando um modelo comum: armazenar *futures* em um `dict` para vincular a eles informações adicionais quando são submetidos, para usá-las quando o *future* sai do gerador `as_completed`.

20.7. Para saber mais

O pacote `concurrent.futures` foi uma contribuição de Brian Quinlan, que o apresentou em uma palestra sensacional intitulada *The Future Is Soon!* [fpy.li/20-20] (O futuro é logo!), na PyCon Australia 2010. A palestra de Quinlan não tinha slides; ele mostra o que a biblioteca faz digitando código diretamente no console de Python. Como exemplo motivador, a apresentação inclui um pequeno vídeo com o cartunista/programador do XKCD, Randall Munroe, executando um ataque de negação de serviço (DoS) acidental contra o Google Maps, para criar um mapa

colorido de tempos de locomoção pela cidade. A introdução formal à biblioteca é a PEP 3148—futures: execute computations asynchronously [fpy.li/pep3148] (executar processamentos assíncronos). Na PEP, Quinlan escreveu que a biblioteca `concurrent.futures` foi "muito influenciada pelo pacote `java.util.concurrent` de Java."

Para mais recursos sobre `concurrent.futures`, por favor consulte o Capítulo 19. As referências que tratam de `threading` e `multiprocessing` de Python na Seção 19.9.1 também tratam do `concurrent.futures`.

Ponto de vista

Evitando Threads

Concorrência: um dos tópicos mais difíceis na ciência da computação (normalmente é melhor evitá-lo).^[7]

— David Beazley, instrutor de Python e cientista louco.

Concordo com as citações aparentemente contraditórias de David Beazley e Michele Simionato no início desse capítulo.

Assisti a um curso de graduação sobre concorrência no IME/USP. Tudo o que vimos foi programação de «threads POSIX» [fpy.li/an]. O que aprendi: não quero gerenciar threads e travas na unha, pela mesma razão que não quero gerenciar a alocação e desalocação de memória na unha. Estas tarefas são para programadores de sistemas, que têm o conhecimento, a inclinação e o tempo para fazê-las direito (ou assim esperamos). Sou pago para desenvolver aplicações, não sistemas operacionais. Não preciso desse controle fino de threads, travas, `malloc` e `free`—veja *Alocação dinâmica de memória em C* [fpy.li/ap].

Por isso acho o pacote `concurrent.futures` interessante: ele trata threads, processos, e filas como infraestrutura, algo a seu serviço, não algo que você precisa controlar diretamente. Claro, ele foi projetado pensando em tarefas simples, os assim chamados problemas "embaraçosamente paralelos"—ao contrário de sistemas operacionais ou servidores de banco de dados, como aponta Simionato na epígrafe deste capítulo.

Para problemas de concorrência mais complexos, threads e travas também não são a solução. Ao nível do sistema operacional, as threads nunca vão desaparecer. Mas todas as linguagens de programação que achei interessantes nos últimos 20 anos oferecem abstrações de alto nível para concorrência, como demonstra o excelente livro de Paul Butcher, *Seven Concurrency Models in Seven Weeks* [fpy.li/20-24] (Sete Modelos de Concorrência em Sete Semanas). Go, Elixir, e Clojure estão entre elas. Erlang—a linguagem de implementação do Elixir—é um exemplo claro de uma linguagem projetada desde o início pensando em concorrência. Erlang não me empolga só porque acho sua sintaxe deselegante. Python me acostumou mal.

José Valim, antigo mantenedor do *Ruby on Rails*, projetou o Elixir com uma sintaxe moderna e agradável. Como Lisp e Clojure, o Elixir implementa macros sintáticas. Isto é uma faca de dois gumes. Macros sintáticas permitem criar DSLs—sigla de *Domain-Specific Language* (Linguagem de Domínio Específico), facilitando determinadas tarefas. Mas a proliferação de sub-linguagens pode levar a bases de código incompatíveis e à fragmentação da comunidade. O Lisp se afogou em um mar de macros, cada empresa e grupo de desenvolvedores Lisp usando seu próprio dialeto arcano. A padronização que criou o Common Lisp resultou em uma linguagem inchada quando muitas macros foram incorporadas ao padrão. Espero que José Valim inspire a comunidade do Elixir a evitar um destino semelhante. Até agora, o cenário parece bom. A biblioteca de banco de dados *Ecto* [fpy.li/20-25] é muito agradável de usar: um grande exemplo do uso de macros para criar uma DSL flexível e amigável para interagir com bases de dados relacionais e não-relacionais.

Como, Elixir, Go é uma linguagem moderna com ideias novas. Mas, em alguns aspectos, é uma linguagem conservadora, se comparada ao Elixir. Go não tem macros, e sua sintaxe é mais simples que a de Python. Go não suporta herança ou sobrecarga de operadores, e oferece menos oportunidades para metaprogramação que Python. Estas limitações são consideradas vantagens. Elas proporcionam comportamento e desempenho mais previsíveis. Isto é uma grande vantagem em ambientes de missão crítica altamente concorrentes, onde o Go pretende substituir C++, Java e Python.

Enquanto Elixir e Go são competidores diretos no espaço da alta concorrência, seus projetos e filosofias atraem públicos diferentes. Ambos têm boas chances de prosperar. Mas historicamente, as linguagens mais conservadoras tendem a atrair mais programadores.

[1] Trecho do texto *Threads, processes and concurrency in Python: some thoughts* [fpy.li/20-1] (Threads, processos e concorrência em Python: algumas reflexões), resumido assim pelo autor: "Removendo exageros sobre a (não-)revolução dos múltiplos núcleos e alguns comentários sensatos (oxalá) sobre threads e outras formas de concorrência."

[2] Para servidores que podem receber requisições de muitos clientes, há uma diferença: as corrotinas escalam melhor, pois usam menos memória que as threads, e também reduzem o custo das mudanças de contexto, que mencionei na Seção 19.6.5.

[3] As imagens são originalmente do CIA World Factbook [fpy.li/20-4], uma publicação de domínio público do governo norte-americano. Copiei as imagens para o meu site, para evitar o risco de lançar um ataque de DoS contra *cia.gov*.

[4] Definir `follow_redirects=True` não é necessário neste exemplo, mas eu queria destacar essa importante diferença entre *HTTPX* e *requests*. Além disso, definir `follow_redirects=True` no exemplo permite que eu coloque os arquivos de imagem em outro lugar no futuro. Considero sensata a configuração default do *HTTPX*, `follow_redirects=False`, pois redirecionamentos inesperados podem mascarar requisições desnecessárias e complicar o diagnóstico de erros.

[5] Acrônimo de *your mileage may vary*, algo como "sua quilometragem pode variar", querendo dizer "seu caso pode ser diferente". Com threads, você nunca sabe a sequência exata de eventos que deveriam acontecer quase ao mesmo tempo; é possível que, em outra máquina, se veja `loiter(1)` começar antes de `loiter(0)` terminar, especialmente porque `sleep` sempre libera a GIL, então Python pode mudar para outra thread mesmo se você dormir por 0s.

[6] Em setembro de 2021 não havia dicas de tipo na versão (então) atual do `tqdm`. Tudo bem. O mundo não vai acabar por causa disso. Obrigado, Guido, pela tipagem opcional!

[7] Slide #9 do tutorial *A Curious Course on Coroutines and Concurrency* [fpy.li/20-21] (Um Curioso Curso sobre Corrotinas e Concorrência), apresentado na PyCon 2009.

Capítulo 21. Programação assíncrona

O problema com as abordagens usuais da programação assíncrona é que são propostas do tipo "tudo ou nada". Ou você reescreve todo o código, de forma que nada nele bloqueie, ou está só perdendo tempo.^[1]

— Alvaro Videla & Jason J. W. Williams, *RabbitMQ in Action*

Este capítulo trata de três grandes tópicos interligados:

- As instruções `async def`, `await`, `async with`, e `async for`;
- Objetos que suportam estas instruções através de métodos especiais como `__await__`, `__aiter__` etc., como corrotinas nativas e variantes assíncronas de gerenciadores de contexto, iteráveis, geradores e compreensões;
- `asyncio` e outras bibliotecas assíncronas.

Este capítulo parte das ideias de iteráveis e geradores (Capítulo 17, em particular a Seção 17.13), gerenciadores de contexto (Capítulo 18), e conceitos gerais de programação concorrente (Capítulo 19).

Vamos estudar clientes HTTP concorrentes similares aos vistos no Capítulo 20, reescritos com corrotinas nativas e gerenciadores de contexto assíncronos, usando a mesma biblioteca *HTTPX* de antes, mas agora através de sua API assíncrona. Veremos também como evitar o bloqueio do laço de eventos, delegando operações lentas para um executor de threads ou processos.

Após os exemplos de clientes HTTP, teremos duas aplicações simples de servidor, uma delas usando o framework *FastAPI*. A seguir estudaremos outros objetos suportados pelas palavras-chave `async/await`: funções geradoras assíncronas, compreensões assíncronas, e expressões geradoras assíncronas. Para enfatizar que estes recursos não estarão limitados ao `asyncio`, veremos um exemplo reescrito para usar o *Curio*—o inovador e influente framework inventado por David Beazley. Finalizando o capítulo, escrevi uma pequena seção sobre vantagens e armadilhas da programação assíncrona.

Temos um longo caminho à frente. Teremos espaço apenas para exemplos básicos, mas eles vão ilustrar as características mais importantes de cada ideia.



A documentação do `asyncio` [fpy.li/21-1] melhorou muito após Yury Selivanov^[2] reorganizá-la, dando maior destaque às funções úteis para desenvolvedores de aplicações. A maior parte da API de `asyncio` consiste em funções e classes voltadas para criadores de pacotes como frameworks Web e drivers de bancos de dados, ou seja, são necessários para criar bibliotecas assíncronas, mas não aplicações.

Para mais profundidade sobre `asyncio`, recomendo *Using Asyncio in Python* [fpy.li/hattingh] de Caleb Hattingh (O'Reilly). Transparência: Caleb é um dos revisores técnicos deste livro.

21.1. Novidades neste capítulo

Quando escrevi a primeira edição de *Python Fluente*, a biblioteca `asyncio` era provisória e as palavras-chave `async/await` não existiam. Por isso atualizei todos os exemplos deste capítulo. Também criei novos exemplos: scripts de sondagem de domínios, um serviço Web com *FastAPI*, e experimentos com o novo modo assíncrono do console do Python.

Novas seções tratam de recursos da linguagem inexistentes naquele momento, como corrotinas nativas, `async with`, `async for`, e os objetos que suportam essas instruções.

As ideias na Seção 21.13 refletem lições importantes que aprendi na prática, por isso eu a considero leitura essencial para qualquer um trabalhando com programação assíncrona. Elas podem ajudar você a evitar muitos problemas—no Python ou mesmo no Node.js.

Por fim, removi vários parágrafos sobre `asyncio.Futures`, que agora considero parte das APIs de baixo nível do `asyncio`.

21.2. Algumas definições.

Na Seção 17.13, vimos que Python oferece três tipos de corrotinas desde a versão 3.5:

Corrotina nativa

Uma função corrotina definida com `async def`. Uma corrotina nativa pode acionar outra corrotina nativa, usando a instrução `await`, semelhante ao funcionamento de `yield from` em corrotinas clássicas. A instrução `async def` sempre define uma corrotina nativa, mesmo se a instrução `await` não aparecer em seu corpo. A instrução `await` só pode ser usada dentro de uma corrotina nativa.^[3]

Corrotina clássica

Uma função geradora que consome dados enviados a ela via chamadas a `my_coro.send(data)`, e que lê aqueles dados usando `yield` em uma expressão. Corrotinas clássicas podem delegar para outras corrotinas clássicas usando `yield from`. Corrotinas clássicas não podem ser controladas por `await`, e não são mais suportadas pelo `asyncio`.

Corrotinas baseadas em geradores

Uma função geradora decorada com `@types.coroutine`—introduzido no Python 3.5. Esse decorador torna o gerador compatível com a nova instrução `await`.

Neste capítulo vamos nos concentrar nas corrotinas nativas, bem como nos geradores assíncronos:

Geradora assíncrona

Uma função geradora definida com `async def` que usa `yield` em seu corpo. Ela devolve um objeto gerador assíncrono que implementa `__anext__`, um método corrotina para obter o próximo item.



`@asyncio.coroutine` *não tem futuro*^[4]

O decorador `@asyncio.coroutine` para corrotinas clássicas e corrotinas baseadas em gerador foi descontinuado no Python 3.8, e está previsto para ser removido no Python 3.11, de acordo com o «Issue 43216» [fpy.li/21-2]. Por outro lado, `@types.coroutine` deve continuar existindo, como se vê aqui: «Issue 36921» [fpy.li/21-3]. Este decorador não é mais suportado pelo `asyncio`, mas é usado em código interno nos frameworks assíncronos *Curio* e *Trio*.

21.3. Sondando domínios com `asyncio`

Imagine que você esteja prestes a lançar um novo blog sobre Python, e planeje registrar um domínio usando uma palavra-chave de Python e o sufixo `.DEV`—por exemplo, `AWAIT.DEV`. O Exemplo 1 é um script usando `asyncio` que verifica vários domínios de forma concorrente. Essa é a saída produzida pelo script:

```
$ python3 blogdom.py
with.dev
+ elif.dev
+ def.dev
  from.dev
  else.dev
  or.dev
  if.dev
  del.dev
+ as.dev
  none.dev
  pass.dev
  true.dev
+ in.dev
+ for.dev
+ is.dev
+ and.dev
+ try.dev
+ not.dev
```

Observe que os domínios aparecem fora de ordem. Se você rodar o script, os verá sendo exibidos um após o outro, em intervalos variados. O sinal de `+` indica que sua máquina foi capaz de resolver o domínio via DNS. Caso contrário, o domínio não está em uso, e pode estar disponível para adquirir.^[4]

No *blogdom.py*, a sondagem de DNS é feita por objetos corrotinas nativas. Como as operações assíncronas são intercaladas, o tempo necessário para verificar 18 domínios é bem menor que se eles fossem verificados sequencialmente. Na verdade, o tempo total é quase o igual ao da resposta mais lenta, em vez da soma dos tempos de todas as respostas do DNS.

O Exemplo 1 mostra o código de *blogdom.py*.

Exemplo 1. *blogdom.py*: procura domínios para um blog sobre Python

```
#!/usr/bin/env python3
import asyncio
import socket
from keyword import kwlist, softkwlist

MAX_KEYWORD_LEN = 4 ①
KEYWORDS = sorted(kwlist + softkwlist)

async def probe(domain: str) -> tuple[str, bool]: ②
    loop = asyncio.get_running_loop() ③
    try:
        await loop.getaddrinfo(domain, None) ④
    except socket.gaierror:
        return (domain, False)
    return (domain, True)

async def main() -> None: ⑤
    names = (kw for kw in KEYWORDS if len(kw) <= MAX_KEYWORD_LEN) ⑥
    domains = (f'{name}.dev'.lower() for name in names) ⑦
    coros = [probe(domain) for domain in domains] ⑧
    for coro in asyncio.as_completed(coros): ⑨
        domain, found = await coro ⑩
        mark = '+' if found else ' '
        print(f'{mark} {domain}')

if __name__ == '__main__':
    asyncio.run(main()) ⑪
```

- ① Estabelece o comprimento máximo da palavra-chave para domínios, pois quanto menor, melhor.
- ② `probe` devolve uma tupla com o nome do domínio e um valor booleano; `True` significa que o domínio foi resolvido. Incluir o nome do domínio aqui facilita a exibição dos resultados.
- ③ Obtém uma referência para o laço de eventos do `asyncio`, para usá-la a seguir.
- ④ O método corrotina `loop.getaddrinfo(...)` [fpy.li/eq] devolve uma tupla de parâmetros com cinco partes [fpy.li/ar] para conectar ao endereço dado usando

um socket. Neste exemplo não precisamos do resultado. Se conseguirmos um resultado, o domínio foi resolvido; caso contrário, não.

- ⑤ `main` precisa ser uma corrotina, para podermos usar `await` aqui.
- ⑥ Gerador para produzir palavras-chave com tamanho até `MAX_KEYWORD_LEN`.
- ⑦ Gerador para produzir nome de domínio com o sufixo `.dev`.
- ⑧ Cria uma lista de objetos corrotina, invocando a corrotina `probe` com cada argumento `domain`.
- ⑨ `asyncio.as_completed` é um gerador que produz corrotinas que devolvem os resultados das corrotinas passadas a ele. Ele as produz na ordem em que elas terminam seu processamento, não na ordem em que foram submetidas. É similar ao `futures.as_completed`, que vimos no Capítulo 20, Exemplo 4.
- ⑩ Nesse ponto, sabemos que a corrotina terminou, pois é assim que `as_completed` funciona. Portanto, a expressão `await` não vai bloquear, mas precisamos dela para obter o resultado de `coro`. Se `coro` gerou uma exceção não tratada, ela será gerada novamente aqui.
- ⑪ `asyncio.run` inicia o laço de eventos e retorna só quando o laço termina. É prática comum em scripts com `asyncio` implementar `main` como uma corrotina e acioná-la com `asyncio.run` no `if name == 'main':`



A função `asyncio.get_running_loop` surgiu no Python 3.7, para uso dentro de corrotinas, como visto em `probe`. Se não houver um laço em execução, `asyncio.get_running_loop` gera um `RuntimeError`. Sua implementação é mais simples e mais rápida que a de `asyncio.get_event_loop`, que pode iniciar um laço de eventos se necessário. A função `asyncio.get_event_loop` está «descontinuada desde o Python 3.10» [fpy.li/as]; eventualmente será um apelido para `asyncio.get_running_loop`.

21.3.1. O truque de Guido para ler código assíncrono

Há muitos conceitos novos para entender no `asyncio`, mas a lógica básica do Exemplo 1 é fácil de compreender se você usar o truque sugerido pelo próprio Guido van Rossum: finja que as palavras-chave `async` e `await` não estão ali. Fazendo isso, você vai perceber que a lógica de uma corrotina pode ser lida como uma função sequencial.

Por exemplo, imagine que o corpo desta corrotina...

```
async def probe(domain: str) -> tuple[str, bool]:
    laço = asyncio.get_running_loop()
    try:
        await loop.getaddrinfo(domain, None)
    except socket.gaierror:
        return (domain, False)
    return (domain, True)
```

...funciona como a função abaixo, exceto que, magicamente, ela nunca bloqueia a execução do laço de eventos:

```
def probe(domain: str) -> tuple[str, bool]: # no async
    laço = asyncio.get_running_loop()
    try:
        loop.getaddrinfo(domain, None) # no await
    except socket.gaierror:
        return (domain, False)
    return (domain, True)
```

Usar a sintaxe `await loop.getaddrinfo(...)` evita o bloqueio, porque `await` suspende somente o objeto corrotina atual. Por exemplo, durante a execução da corrotina `probe('if.dev')`, um novo objeto corrotina é criado por `getaddrinfo('if.dev', None)`. Aplicar `await` sobre ele inicia a consulta de baixo nível `addrinfo` e devolve o controle para o laço de eventos, não para a corrotina `probe('if.dev')`, que está suspensa. Enquanto isso, o laço de eventos pode ativar outros objetos corrotina pendentes, tal como `probe('or.dev')`.

Quando o laço de eventos recebe a resposta da consulta `getaddrinfo('if.dev', None)`, aquele objeto corrotina específico prossegue sua execução, e devolve o controle para `probe('if.dev')`—que estava suspenso no `await`—e pode agora tratar alguma possível exceção e devolver a tupla com o resultado.

Até aqui, vimos `asyncio.as_completed` e `await` sendo aplicados apenas a corrotinas. Mas eles podem lidar com qualquer objeto "esperável". Esse conceito será explicado a seguir.

21.4. Novo conceito: esperável (*awaitable*)

A palavra-chave `for` funciona com "iteráveis" (*iterable*). A palavra-chave `await` funciona com "esperáveis" (*awaitable*).



Os tradutores da documentação do Python em português traduziram *awaitable* como "aguardável". Adotei "esperável" por ser mais simples. E também porque nem todo esperável é agradável ;-)

Como usuário do `asyncio`, estes são os esperáveis que você verá diariamente:

- Um *objeto corrotina nativo*, que você obtém invocando uma *função corrotina nativa*
- Uma tarefa `asyncio.Task`, criada ao invocar `asyncio.create_task(...)` passando um objeto corrotina nativo.

Entretanto, o código do usuário final nem sempre precisa acionar uma `Task` com `await`. Usamos `asyncio.create_task(coro())` para agendar `one_coro` para execução concorrente, sem esperar que retorne. Foi o que fizemos com a corrotina `spinner` em *spinner_async.py* (Exemplo 4 do Capítulo 19). Criar a tarefa é o suficiente para agendar o acionamento da corrotina pelo laço de eventos.



Mesmo que você não precise cancelar a tarefa ou esperar pelo resultado, é necessário preservar o objeto `Task` devolvido por `create_task`, salvando-o em uma variável ou coleção que você controla.^[5] O laço de eventos usa referências fracas para gerenciar as tarefas, o que significa que elas podem ser descartadas pelo coletor de lixo antes de serem acionadas. Por isso você precisa criar referências fortes para preservar cada tarefa na memória. Veja a documentação de `asyncio.create_task` [fpy.li/at]. Sobre referências fracas, escrevi o artigo «Weak References» [fpy.li/weakref] no *fluentpython.com*.

Por outro lado, usamos `await other_coro()` para executar `other_coro` agora mesmo e esperar que ela termine, porque precisamos do resultado para prosseguir. Em *spinner_async.py*, a corrotina supervisor usava `res = await slow()` para executar `slow` e aguardar seu resultado..

Ao implementar bibliotecas assíncronas ou contribuir para o próprio `asyncio`, você pode também encontrar estes esperáveis de baixo nível:

- Um objeto com um método `__await__` que devolve um iterador; por exemplo, uma instância de `asyncio.Future` (`asyncio.Task` é uma subclasse de `asyncio.Future`)
- Objetos escritos em outras linguagens usando a API Python/C, com uma função `tp_as_async.am_await`, que devolve um iterador (similar ao método `__await__`)

As bases de código existentes podem também conter um tipo adicional de esperável: *objetos corrotina baseados em geradores*, que estão no processo de serem descontinuados.



A PEP 492 «afirma» [fpy.li/21-7] que a expressão `await` "usa a implementação de `yield from` com um passo adicional de validação de seu argumento" e que "await só aceita um esperável." A PEP não explica aquela implementação em detalhes, mas cita a *PEP 380* [fpy.li/pep380], que introduziu `yield from`. Escrevi uma explicação detalhada no texto «Classic Coroutines» [fpy.li/oldcoro], seção «The Meaning of `yield from`» [fpy.li/21-8] (O significado de `yield from`).

Agora vamos estudar a versão `asyncio` do script para baixar figuras da Web.

21.5. Downloads com `asyncio` e *HTTPX*

O script *flags_asyncio.py* baixa um conjunto fixo de 20 bandeiras de *fluentpython.com*. Já mencionamos este script na Seção 20.2, mas agora vamos examiná-lo em detalhes, aplicando os conceitos que acabamos de ver.

No Python 3.10, o `asyncio` só suporta TCP e UDP, e não temos pacotes de cliente ou servidor HTTP assíncronos na biblioteca padrão. Estou usando o *HTTPX* [fpy.li/httpx] nos exemplos de clientes HTTP.

Vamos explorar o *flags_asyncio.py* a partir do final do módulo, isto é, olhando primeiro as funções que iniciam as ações no Exemplo 2.



Para deixar o código mais fácil de ler, *flags_asyncio.py* não faz tratamento de erros. Nesta introdução a `async/await` é bom se concentrar inicialmente no "caminho feliz" (*happy path*), para entender como funções comuns e corrotinas são organizadas em um programa. A partir da Seção 21.7, os exemplos incluem tratamento de erros e outros recursos.

Os exemplos *flags** aqui e no Capítulo 20 compartilham código e dados, então os coloquei juntos no diretório *example-code-2e/20-executors/getflags* [fpy.li/21-9].

Exemplo 2. *flags_asyncio.py*: funções de inicialização

```
def download_many(cc_list: list[str]) -> int: ①
    return asyncio.run(supervisor(cc_list)) ②

async def supervisor(cc_list: list[str]) -> int:
    async with AsyncClient() as client: ③
        to_do = [download_one(client, cc)
                  for cc in sorted(cc_list)] ④
        res = await asyncio.gather(*to_do) ⑤

    return len(res) ⑥

if __name__ == '__main__':
    main(download_many) ⑦
```

- ① Esta tem que ser uma função comum—não uma corrotina—para podermos passá-la para função `main` do módulo *flags.py* (Exemplo 2 do Capítulo 20) no passo ⑦.
- ② Executa o laço de eventos, acionando o objeto corrotina `supervisor(cc_list)` até que ele retorne. Esta função ficará bloqueada enquanto o laço de eventos

roda. O resultado de `supervisor(cc_list)` será devolvido por `asyncio_run`.

- ③ Operações assíncronas de cliente HTTP no `httpx` são métodos de `AsyncClient`, que também é um gerenciador de contexto assíncrono, para uso com `async with`. Trata-se de um gerenciador de contexto com métodos corrotina para inicialização e encerramento (detalhes logo mais na Seção 21.6).
- ④ Cria uma lista de objetos corrotina, invocando a corrotina `download_one` uma vez para cada bandeira a ser obtida.
- ⑤ Espera pela corrotina `asyncio.gather`, que aceita um ou mais argumentos esperáveis e aguarda até que todos terminem, devolvendo uma lista de resultados para os esperáveis fornecidos, na ordem em que foram submetidos.
- ⑥ `supervisor` devolve o tamanho da lista vinda de `asyncio.gather`.
- ⑦ Invocamos `main` do `flags.py` (Exemplo 2 do Capítulo 20)

Agora vamos revisar a parte superior de `flags_asyncio.py` (Exemplo 3).

Reorganizei as corrotinas para podermos lê-las na ordem em que são iniciadas pelo laço de eventos.

Exemplo 3. `flags_asyncio.py`: imports e funções de download

```
import asyncio

from httpx import AsyncClient ①

from flags import BASE_URL, save_flag, main ②

async def download_one(client: AsyncClient, cc: str): ③
    image = await get_flag(client, cc)
    save_flag(image, f'{cc}.gif')
    print(cc, end=' ', flush=True)
    return cc

async def get_flag(client: AsyncClient, cc: str) -> bytes: ④
    url = f'{BASE_URL}/{cc}/{cc}.gif'.lower()
    resp = await client.get(url, timeout=6.1,
                            follow_redirects=True) ⑤
    return resp.read() ⑥
```

- ① `httpx` precisa ser instalado; não vem com a biblioteca padrão.
- ② Reutiliza código de *flags.py* (Exemplo 2 do Capítulo 20).
- ③ `download_one` precisa ser uma corrotina nativa, para acionar `get_flag` com `await`. Quando recebe a resposta, exibe o código de país bandeira, e salva a imagem.
- ④ `get_flag` precisa receber o `AsyncClient` para fazer a requisição.
- ⑤ O método `get` de uma instância de `httpx.AsyncClient` devolve um objeto `ClientResponse`.
- ⑥ Operações de E/S de rede são implementadas como métodos corrotina, então eles são controlados de forma assíncrona pelo laço de eventos do `asyncio`.



Idealmente, a invocação de `save_flag` dentro de `get_flag` deveria ser assíncrona, evitando bloquear o laço de eventos com uma operação de E/S. Entretanto, atualmente `asyncio` não oferece uma API assíncrona para acessar o sistema de arquivos—como faz o `Node.js`.

A Seção 21.7.1 vai mostrar como delegar `save_flag` para uma `thread`.

O seu código delega para as corrotinas do `httpx` explicitamente, usando `await`, ou implicitamente, usando os métodos especiais dos gerenciadores de contexto assíncronos, como `AsyncClient` e `ClientResponse`—como veremos na Seção 21.6.

21.5.1. O segredo das corrotinas nativas: humildes geradores

A diferença fundamental entre os exemplos de corrotinas clássicas vistas na Seção 17.13 e *flags_asyncio.py* é que não há chamadas a `.send()` ou expressões `yield` visíveis nesse último. O seu código fica entre a biblioteca `asyncio` e as bibliotecas assíncronas que você estiver usando, como por exemplo a *HTTPX*. Isso está ilustrado na Figura 1.

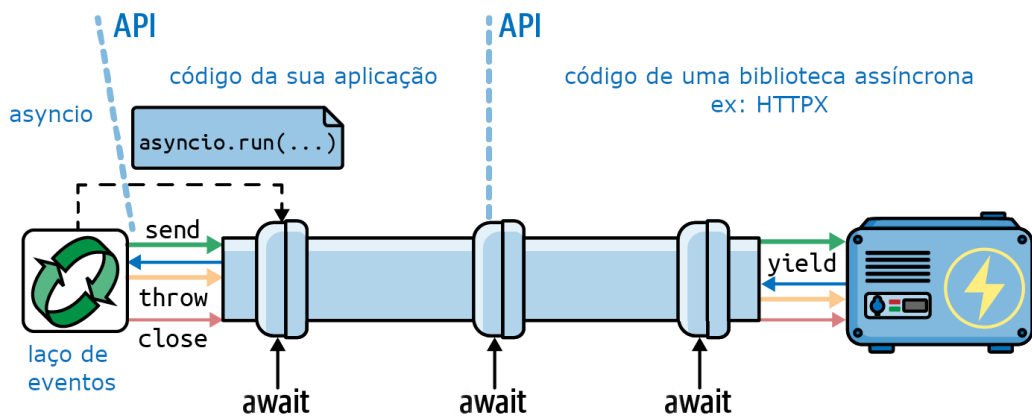


Figura 1. Em um programa assíncrono, uma função do usuário inicia o laço de eventos, agendando uma corrotina inicial com `asyncio.run`. Cada corrotina do usuário aciona a seguinte com uma expressão `await`, formando um canal que permite a comunicação direta entre uma biblioteca como a `HTTPX` e o laço de eventos do framework assíncrono.

Debaixo dos panos, o laço de eventos do `asyncio` faz as chamadas a `.send` que acionam as nossas corrotinas, e nossas corrotinas acionam outras corrotinas com `await`, inclusive corrotinas da biblioteca. Como já mencionado, a maior parte da implementação de `await` vem de `yield from`, que internamente invoca `.send` para acionar as corrotinas.

O canal `await` termina em um esperável de baixo nível da biblioteca `HTTPX`, que devolve um gerador que o laço de eventos pode acionar em resposta a eventos tais como E/S de rede ou timers. Os esperáveis e geradores no final desses canais `await` estão implementados nas profundezas das bibliotecas, não são parte de suas APIs e podem ser extensões Python/C.

Usando funções como `asyncio.gather` e `asyncio.create_task`, podemos ter múltiplos canais de `await` ao mesmo tempo, permitindo acionar operações de E/S concorrentes em um único laço de eventos, em uma única thread.

21.5.2. O problema do tudo ou nada

No Exemplo 3, note que eu reutilizei a função `get_flag` de `flags.py` (Exemplo 2 do Capítulo 20). Tive que reescrevê-la como uma corrotina para usar a API assíncrona do `HTTPX`. Para obter o melhor desempenho do `asyncio`, precisamos substituir todas as funções que fazem E/S por uma versão assíncrona, que seja

acionada com `await` ou `asyncio.create_task`. Assim o controle é devolvido ao laço de eventos, enquanto não chega a resposta da operação de envio ou recebimento de dados. Se for inviável reescrever a função bloqueante como uma corrotina, devemos executá-la em outra thread ou processo, como veremos na Seção 21.8.

Por isso escolhi a epígrafe desse capítulo, que inclui o conselho: "Ou você reescreve todo o código, de forma que nada nele bloqueie ou está só perdendo tempo."

Pela mesma razão, também não pude reutilizar a função `download_one` de *flags_threadpool.py* (Exemplo 3 do Capítulo 20). O código no Exemplo 3 aciona `get_flag` com `await`, então `download_one` precisa também ser uma corrotina. Para cada requisição, supervisor cria um objeto corrotina `download_one`, e eles são todos acionados pela corrotina `asyncio.gather`.

Vamos agora estudar a instrução `async with`, que apareceu em supervisor (Exemplo 2) e `get_flag` (Exemplo 3).

21.6. Gerenciadores de contexto assíncronos

Na Seção 18.2, vimos como um objeto pode ser usado para executar código antes e depois do corpo de um bloco `with`, se sua classe oferecer os métodos `__enter__` e `__exit__`.

Agora, considere o Exemplo 4, que usa o driver PostgreSQL *asyncpg* [fpy.li/21-10] compatível com o `asyncio` (documentação do *asyncpg* sobre transações [fpy.li/21-11]).

Exemplo 4. Código exemplo da documentação do driver PostgreSQL asyncpg

```
tr = connection.transaction()
await tr.start()
try:
    await connection.execute("INSERT INTO mytable VALUES (1, 2, 3)")
except:
    await tr.rollback()
    raise
else:
    await tr.commit()
```

Uma transação de banco de dados se presta naturalmente a protocolo de gerenciador de contexto: a transação precisa ser iniciada, dados são modificados com `connection.execute`, e então temos que fazer um *rollback* (reversão) ou um *commit* (confirmação), dependendo do resultado das mudanças.

Em um driver assíncrono como o *asyncpg*, a configuração e a execução precisam acontecer em corrotinas, para que outras operações possam ocorrer de forma concorrente. Entretanto, a implementação da instrução `with` clássica não suporta corrotinas na implementação dos métodos `__enter__` ou `__exit__`.

Por isso a *PEP 492—Coroutines with `async` and `await` syntax* [fpy.li/pep492] (Corrotinas com a sintaxe `async` e `await`) introduziu a instrução `async with`, que funciona com gerenciadores de contexto assíncronos: objetos que implementam os métodos `__aenter__` e `__aexit__` como corrotinas.

Usando `async with`, o Exemplo 4 pode ser escrito como esse outro exemplo da «documentação do *asyncpg*» [fpy.li/21-11]:

```
async with connection.transaction():
    await connection.execute("INSERT INTO mytable VALUES (1, 2, 3)")
```

Na «classe *asyncpg.Transaction*» [fpy.li/21-13], o método corrotina `__aenter__` executa `await self.start()`, e a corrotina `__aexit__` aciona um dos métodos corrotina privados `__rollback` ou `__commit`, dependendo da ocorrência ou não de uma exceção. Usar corrotinas para implementar *Transaction* como um gerenciador de contexto assíncrono permite ao *asyncpg* controlar muitas transações concorrentemente.



Caleb Hattingh sobre o asyncpg

Outro detalhe impressionante sobre o *asyncpg* é que ele também contorna a falta de suporte à alta-concorrência do PostgreSQL (que usa um processo servidor por conexão) implementando um banco de conexões para conexões internas ao próprio Postgres.

Isto significa que não precisamos de ferramentas adicionais (por exemplo o *pgbouncer*), como explicado na «documentação» [fpy.li/21-14] do *asyncpg*.

Voltando ao *flags_asyncio.py*, a classe `AsyncClient` do `httpx` é um gerenciador de contexto assíncrono, para poder acionar esperáveis em seus métodos corrotina especiais `__aenter__` e `__aexit__`.



A Seção 21.10.1.3 mostra como usar a `contextlib` de Python para criar um gerenciador de contexto assíncrono sem precisar escrever uma classe. Esta explicação aparece mais tarde nesse capítulo por causa de um pré-requisito: a Seção 21.10.1.

Agora vamos melhorar o exemplo `asyncio` de download de bandeiras com uma barra de progresso, que nos levará a explorar um pouco mais a API do `asyncio`.

21.7. Melhorando o download de bandeiras assíncrono

Vamos recordar a Seção 20.5, na qual o conjunto de exemplos `flags2` compartilhava a mesma interface de linha de comando, e todos mostravam uma barra de progresso enquanto os downloads aconteciam. Eles também incluíam tratamento de erros.



Encorajo você a brincar com os exemplos `flags2`, para desenvolver uma intuição sobre o funcionamento de clientes HTTP concorrentes. Use a opção `-h` para ver a tela de ajuda no Exemplo 10. Use as opções de linha de comando `-a`, `-e`, e `-l` para controlar o número de downloads, e a opção `-m` para estabelecer o número de downloads concorrentes. Execute testes com os servidores `LOCAL`, `REMOTE`, `DELAY`, e `ERROR`. Descubra o número ótimo de downloads concorrentes para maximizar a taxa de transferência de cada servidor. Varie as opções dos servidores de teste, como descrito na caixa *Configurando os servidores de teste* da Seção 20.5.

Por exemplo, o Exemplo 5 mostra uma tentativa de obter 100 bandeiras (`-a1 100`) do servidor `ERROR`, usando 100 conexões concorrentes (`-m 100`). Os 48 erros no resultado são ou HTTP 418 ou erros de tempo de espera excedido (*time-out*)—o [mau]comportamento esperado do *slow_server.py*.

Exemplo 5. Running flags2_asyncio.py

```
$ python3 flags2_asyncio.py -s ERROR -al 100 -m 100
ERROR site: http://localhost:8002/flags
Searching for 100 flags: from AD to LK
100 concurrent connections will be used.
100%|██████████████████████████████████████████████| 100/100 [00:03<00:00, 30.48it/s]
-----
52 flags downloaded.
48 errors.
Elapsed time: 3.31s
```



Seja responsável ao testar clientes concorrentes

Mesmo que o tempo total de download não seja muito diferente entre os clientes HTTP na versão com threads e na versão `asyncio` HTTP, o `asyncio` é capaz de enviar requisições mais rápido, então aumenta a probabilidade do servidor suspeitar de um ataque DoS. Para testar estes clientes concorrentes em sua capacidade máxima, por favor use servidores HTTP locais em seus testes, como explicado na caixa *Configurando os servidores de teste* da Seção 20.5.

Agora vejamos como o `flags2_asyncio.py` é implementado.

21.7.1. Usando `asyncio.as_completed` e uma `thread`

No Exemplo 3, passamos várias corrotinas para `asyncio.gather`, que devolve uma lista com os resultados das corrotinas na ordem em que foram submetidas. Isto significa que `asyncio.gather` só pode retornar quando todos os esperáveis terminarem. Entretanto, para atualizar a barra de progresso, precisamos receber cada resultado assim que ele está pronto.

Felizmente existe uma equivalente assíncrona da função geradora `as_completed` que usamos no exemplo de banco de threads com a barra de progresso, (Exemplo 16 do Capítulo 20).

O Exemplo 6 mostra o início do script *flags2_asyncio.py*, onde as corrotinas `get_flag` e `download_one` são definidas. O Exemplo 7 lista o restante do código-

fonte, com supervisor e download_many. O script é maior que *flags_asyncio.py* por causa do tratamento de erros.

Exemplo 6. flags2_asyncio.py: parte superior (inicial) do script; o resto do código está no Exemplo 7

```
import asyncio
from collections import Counter
from http import HTTPStatus
from pathlib import Path

import httpx
import tqdm # type: ignore

from flags2_common import main, DownloadStatus, save_flag

# low concurrency default to avoid errors from remote site,
# such as 503 - Service Temporarily Unavailable
DEFAULT_CONCUR_REQ = 5
MAX_CONCUR_REQ = 1000

async def get_flag(client: httpx.AsyncClient, ①
                  base_url: str,
                  cc: str) -> bytes:
    url = f'{base_url}/{cc}/{cc}.gif'.lower()
    resp = await client.get(url, timeout=3.1, follow_redirects=True) ②
    resp.raise_for_status()
    return resp.content

async def download_one(client: httpx.AsyncClient,
                      cc: str,
                      base_url: str,
                      semaphore: asyncio.Semaphore,
                      verbose: bool) -> DownloadStatus:
    try:
        async with semaphore: ③
            image = await get_flag(client, base_url, cc)
    except httpx.HTTPStatusError as exc: ④
        res = exc.response
        if res.status_code == HTTPStatus.NOT_FOUND:
            status = DownloadStatus.NOT_FOUND
            msg = f'not found: {res.url}'
```

```

        else:
            raise
    else:
        await asyncio.to_thread(save_flag, image, f'{cc}.gif') ⑤
        status = DownloadStatus.OK
        msg = 'OK'
        if verbose and msg:
            print(cc, msg)
        return status

```

- ① `get_flag` é muito similar à versão sequencial no Exemplo 14. Primeira diferença: ela requer o parâmetro `client`.
- ② Segunda e terceira diferenças: `.get` é um método de `AsyncClient`, e é uma corrotina, então precisamos acioná-la com `await`.
- ③ Usa o `semaphore` como um gerenciador de contexto assíncrono, assim o programa todo não é bloqueado; apenas essa corrotina é suspensa quando o contador do semáforo é zero. Veremos mais sobre isso na caixa *Semáforos no Python*, Seção 21.7.2.
- ④ A lógica de tratamento de erro é idêntica à de `download_one`, do Exemplo 14 do Capítulo 20.
- ⑤ Salvar a imagem é uma operação de E/S. Para não bloquear o laço de eventos, roda `save_flag` em uma `thread`.

No `asyncio`, toda a comunicação de rede é feita com corrotinas, mas não E/S de arquivos. Entretanto, E/S de arquivos também é "bloqueante"—pois ler/escrever arquivos é «milhares de vezes mais demorado» [fpy.li/21-15] que ler/escrever na RAM. Se você estiver usando «armazenamento conectado à rede» [fpy.li/av], acessar "o disco" significa fazer E/S na rede local.

Desde o Python 3.9, a corrotina `asyncio.to_thread` facilitou delegar operações de arquivo para um banco de threads fornecido pelo `asyncio`. Se você precisa suportar Python 3.7 ou 3.8, a Seção 21.8 mostra como fazer isso, adicionando algumas linhas ao seu programa. Mas antes, vamos terminar nosso estudo do código do cliente HTTP.

21.7.2. Limitando as requisições com um semáforo

Clientes de rede como os que estamos estudando devem ser *throttled* (limitados no desempenho) para não martelarem o servidor com um número excessivo de requisições concorrentes.

Um *semáforo* [fpy.li/aw] é uma estrutura primitiva de sincronização, mais flexível que uma trava. O mesmo semáforo é usado por várias corrotinas, com um número máximo configurável. Assim podemos limitar o número de corrotinas concorrentes ativas usando o semáforo. A caixa *Semáforos no Python* tem mais informações.

No *flags2_threadpool.py* (Exemplo 16 do Capítulo 20), a limitação de desempenho (*throttling*) foi feita na função `download_many` instanciando o `ThreadPoolExecutor` passando um número máximo de threads no argumento `max_workers`. Em *flags2_asyncio.py*, um `asyncio.Semaphore` é criado pela função `supervisor` (mostrada no Exemplo 7) e passado como o argumento `semaphore` para `download_one` no Exemplo 6.

Semáforos no Python

O cientista da computação Edsger W. Dijkstra inventou o «semáforo» [fpy.li/aw] no início dos anos 1960. É uma ideia simples, mas tão flexível que a maioria dos outros objetos de sincronização—como as travas e as barreiras—podem ser construídas a partir de semáforos. Há três classes `Semaphore` na biblioteca padrão de Python: uma em `threading`, outra em `multiprocessing`, e uma terceira em `asyncio`. Essas classes têm interfaces parecidas, mas suas implementações são bem diferentes. Aqui apresento a versão de `asyncio`.

Um `asyncio.Semaphore` tem um contador interno que é decrementado toda vez que acionamos o método corrotina `.acquire()` com `await`. O contador é incrementado quando invocamos o método `.release()`—que não é uma corrotina porque nunca bloqueia. O valor inicial do contador é definido quando o `Semaphore` é instanciado:

```
semaphore = asyncio.Semaphore(concur_req)
```

Fazer `await` em `.acquire()` não bloqueia quando o contador interno é maior que zero. Mas o contador está zerado, `.acquire()` suspende a corrotina que chamou `await` até que alguma outra corrotina chame `.release()` no mesmo `Semaphore`, incrementando assim o contador.

Em vez de usar estes métodos diretamente, é mais seguro usar o `semaphore` como um gerenciador de contexto assíncrono, como fiz na função `download_one` em Exemplo 6:

```
async with semaphore:
    image = await get_flag(client, base_url, cc)
```

O método corrotina `Semaphore.__aenter__` espera por `.acquire()` (usando `await` internamente), e seu método corrotina `__aexit__` invoca `.release()`. Este bloco `async with` garante que no máximo `concur_req` instâncias de corrotinas `get_flags` estarão ativas em qualquer momento.

Cada uma das classes `Semaphore` na biblioteca padrão tem uma subclasse `BoundedSemaphore`, que impõe uma restrição adicional: o contador interno não pode nunca ficar maior que o valor inicial, impedindo mais operações `.release()` que `.acquire()`.^[6]

Agora vamos olhar o resto do script no Exemplo 7.

Exemplo 7. `flags2_asyncio.py`: continuação de Exemplo 6

```
async def supervisor(cc_list: list[str],
                    base_url: str,
                    verbose: bool,
                    concur_req: int) -> Counter[DownloadStatus]: ①
    counter: Counter[DownloadStatus] = Counter()
    semaphore = asyncio.Semaphore(concur_req) ②
    async with httpx.AsyncClient() as client:
        to_do = [download_one(client, cc, base_url, semaphore, verbose)
                 for cc in sorted(cc_list)] ③
        to_do_iter = asyncio.as_completed(to_do) ④
        if not verbose:
            to_do_iter = tqdm.tqdm(to_do_iter, total=len(cc_list)) ⑤
        error: httpx.HTTPError | None = None ⑥
```

```

for coro in to_do_iter: ⑦
    try:
        status = await coro ⑧
    except httpx.HTTPStatusError as exc:
        error_msg = ('HTTP error {resp.status_code}' +
                     ' - {resp.reason_phrase}')
        error_msg = error_msg.format(resp=exc.response)
        error = exc ⑨
    except httpx.RequestError as exc:
        error_msg = f'{exc} {type(exc)}'.strip()
        error = exc ⑩
    except KeyboardInterrupt:
        break

    if error:
        status = DownloadStatus.ERROR ⑪
        if verbose:
            url = str(error.request.url) ⑫
            cc = Path(url).stem.upper() ⑬
            print(f'{cc} error: {error_msg}')
        counter[status] += 1

return counter

def download_many(cc_list: list[str],
                  base_url: str,
                  verbose: bool,
                  concur_req: int) -> Counter[DownloadStatus]:
    coro = supervisor(cc_list, base_url, verbose, concur_req)
    counts = asyncio.run(coro) ⑭

return counts

if __name__ == '__main__':
    main(download_many, DEFAULT_CONCUR_REQ, MAX_CONCUR_REQ)

```

- ① supervisor recebe os mesmos argumentos que a função `download_many`, mas não pode ser invocada diretamente de `main`, pois é uma corrotina e não uma função comum como `download_many`.
- ② Cria um `asyncio.Semaphore` que não vai permitir mais que `concur_req` corrotinas ativas entre aquelas usando este semáforo. O valor de `concur_req` é

calculado pela função `main` de *flags2_common.py*, baseado nas opções de linha de comando e nas constantes definidas em cada exemplo.

- ③ Cria uma lista de objetos corrotina, um para cada chamada à corrotina `download_one`.
- ④ Obtém um iterador que vai devolver objetos corrotina quando eles terminarem sua execução. Não coloquei essa chamada a `as_completed` diretamente no laço `for` abaixo porque posso precisar envolvê-la com o iterador `tqdm` para a barra de progresso, dependendo da opção de verbosidade na linha de comando.
- ⑤ Envolva o iterador `as_completed` com a função geradora `tqdm`, para mostrar o progresso.
- ⑥ Declara e inicializa `error` com `None`; esta variável será usada para salvar uma exceção além do bloco `try/except`, se alguma for levantada.
- ⑦ Itera pelos objetos corrotina que terminaram a execução; este laço é similar ao de `download_many` no Exemplo 16 do Capítulo 20.
- ⑧ Aciona a corrotina com `await` para obter seu resultado. Isto não bloqueia porque `as_completed` só produz corrotinas que já terminaram.
- ⑨ Esta atribuição é necessária porque o escopo da variável `exc` é limitado a esta cláusula `except`, mas preciso preservar o valor para uso posterior.
- ⑩ Mesmo que acima.
- ⑪ Se houve um erro, muda o status.
- ⑫ Em modo verboso, extrai a URL da exceção que foi levantada...
- ⑬ ...e extrai o nome do arquivo para mostrar o código do país em seguida.
- ⑭ `download_many` instancia o objeto corrotina `supervisor` e o passa para o laço de eventos com `asyncio.run`, coletando o contador que `supervisor` devolve quando o laço de eventos termina.

No Exemplo 7, não pudemos usar o mapeamento de `futures` para os códigos de país que vimos em Exemplo 16 do Capítulo 20, porque os esperáveis devolvidos por `asyncio.as_completed` não são necessariamente os mesmos esperáveis que passamos na invocação de `as_completed`. Internamente, a lógica do `asyncio` pode embrulhar os esperáveis que fornecemos por outros esperáveis que afinal produzirão os mesmos resultados.^[7]



Já que não podia usar os esperáveis como chaves para recuperar os códigos de país de um dict em caso de falha, tive que extrair o código de país da exceção. Para fazer isso, preservei a exceção na variável `error`, permitindo sua recuperação fora do bloco `try/except`. Python não é uma linguagem com escopo de bloco: instruções como laços e `try/except` não criam um escopo local nos blocos que eles gerenciam. Mas se uma cláusula `except` vincula uma exceção a uma variável, como as variáveis `exc` que acabamos de ver—aquele vínculo só existe dentro daquela cláusula `except` específica.

Isto encerra nossa discussão da funcionalidade de um cliente Web com tratamento de erros e barra de progresso, implementado com `asyncio`.

O próximo exemplo demonstra um modelo simples de execução de uma tarefa assíncrona após outra usando corrotinas.

21.7.3. Fazendo múltiplas requisições para cada download

Pessoas com experiência em JavaScript sabem que rodar uma função assíncrona após outra acabou gerando o padrão de funções aninhadas conhecido como *pyramid of doom* [fpy.li/21-20] (pirâmide da perdição). A palavra-chave `await` desfaz a maldição. Por isso `await` agora é parte de Python e de JavaScript. Vamos a um exemplo.

Suponha que você queira salvar cada bandeira com o nome do país e o código, em vez de apenas o código. Agora você precisa fazer duas requisições HTTP por bandeira: uma para pegar a imagem da bandeira propriamente dita, a outra para pegar o arquivo *metadata.json*, no mesmo diretório da imagem—é nesse arquivo que o nome do país está registrado.

Coordenar múltiplas requisições na mesma tarefa é fácil no script com `threads`: basta fazer uma requisição depois a outra, bloqueando a thread duas vezes, e preservando os dados (código e nome do país) em variáveis locais, prontas para serem usadas na hora de salvar o arquivo. Se você precisasse fazer o mesmo em um script assíncrono com `callbacks`, precisaria de funções aninhadas, de forma que o código e o nome do país estivessem disponíveis em clausuras até o

momento de salvar o arquivo, pois cada callback roda em um escopo local diferente. A palavra-chave `await` evita este aninhamento, permitindo que você acione as requisições assíncronas uma após a outra, compartilhando o escopo local da corrotina que aciona as ações.



Se você está trabalhando com programação de aplicações assíncronas no Python moderno e recorre a uma grande quantidade de callbacks, provavelmente está aplicando modelos antigos, que não fazem mais sentido no Python atual. Isso é justificável se você estiver escrevendo uma biblioteca que se conecta a código legado ou código de baixo nível sem suporte a corrotinas. De qualquer forma, a discussão no StackOverflow, *What is the use case for future.add_done_callback()? [fpy.li/21-21]* (Qual o caso de uso para `future.add_done_callback()`?) explica por que callbacks são necessários em código de baixo nível, mas não são muito úteis hoje em dia em código Python no nível da aplicação.

A terceira variante do script `asyncio` de download de bandeiras traz algumas mudanças:

`get_country`

Esta nova corrotina baixa o arquivo *metadata.json* daquele código de país, e extrai dele o nome do país.

`download_one`

Esta corrotina agora usa `await` para delegar para `get_flag` e para a nova corrotina `get_country`, usando o resultado dessa última para compor o nome do arquivo a ser salvo.

Vamos começar com o código de `get_country` (Exemplo 8). Note que é muito parecido com o `get_flag` do Exemplo 6.

Exemplo 8. flags3_asyncio.py: corrotina get_country

```
async def get_country(client: httpx.AsyncClient,
                      base_url: str,
                      cc: str) -> str: ①
    url = f'{base_url}/{cc}/metadata.json'.lower()
    resp = await client.get(url, timeout=3.1, follow_redirects=True)
    resp.raise_for_status()
    metadata = resp.json() ②
    return metadata['country'] ③
```

- ① Esta corrotina devolve uma string com o nome do país—se tudo correr bem.
- ② metadata vai receber um dict Python construído a partir do conteúdo JSON da resposta.
- ③ Devolve o nome do país.

Agora vamos ver o `download_one` modificado do Exemplo 9, que tem apenas algumas linhas diferentes da corrotina de mesmo nome do Exemplo 6.

Exemplo 9. flags3_asyncio.py: corrotina download_one

```
async def download_one(client: httpx.AsyncClient,
                       cc: str,
                       base_url: str,
                       semaphore: asyncio.Semaphore,
                       verbose: bool) -> DownloadStatus:

    try:
        async with semaphore: ①
            image = await get_flag(client, base_url, cc)
        async with semaphore: ②
            country = await get_country(client, base_url, cc)
    except httpx.HTTPStatusError as exc:
        res = exc.response
        if res.status_code == HTTPStatus.NOT_FOUND:
            status = DownloadStatus.NOT_FOUND
            msg = f'not found: {res.url}'
        else:
            raise
    else:
        filename = country.replace(' ', '_') ③
```

```
await asyncio.to_thread(save_flag, image, f'{filename}.gif')
status = DownloadStatus.OK
msg = 'OK'
if verbose and msg:
    print(cc, msg)
return status
```

- ① Retém o semaphore para acionar `get_flag...`
- ② ...e novamente para acionar `get_country`.
- ③ Usa o nome do país para criar um nome de arquivo. Como usuário da linha de comando, não gosto de espaços em nomes de arquivo.

Muito melhor que callbacks aninhados!

Coloquei as chamadas a `get_flag` e `get_country` em blocos `with` separados, controlados pelo semaphore porque é uma boa prática reter semáforos e travas pelo menor tempo possível.

Eu poderia ter agendado as funções `get_flag` e `get_country`, em paralelo, usando `asyncio.gather`, mas se `get_flag` levantar uma exceção não haverá imagem para salvar, então é inútil rodar `get_country`. Mas há casos em que faz sentido usar `asyncio.gather` para acessar várias APIs simultaneamente, em vez de esperar por uma resposta antes de fazer a próxima requisição

Em *flags3_asyncio.py*, a sintaxe `await` aparece seis vezes, e `async with` três vezes. Espero que você esteja pegando o jeito da programação assíncrona em Python.

Um desafio é saber quando você precisa usar `await` e quando você não pode usá-la. A resposta, em princípio, é fácil: use `await` para acionar corrotinas e outros esperáveis, como instâncias de `asyncio.Task`. Mas algumas APIs são complexas, misturam corrotinas e funções comuns de forma aparentemente arbitrária, como a classe `StreamWriter` que usaremos no Exemplo 14.

O Exemplo 9 encerra o grupo de exemplos *flags*. Vamos agora discutir o uso de executores de threads ou processos na programação assíncrona.

21.8. Delegando tarefas a executores

Uma vantagem importante do Node.js sobre o Python para programação assíncrona é a biblioteca padrão do Node.js, que inclui APIs assíncronas para todo tipo de E/S—não apenas para E/S de rede. No Python, se você não for cuidadosa, a E/S de arquivos pode degradar seriamente o desempenho de aplicações assíncronas, pois usar thread principal para ler e escrever no armazenamento bloqueia o laço de eventos.

Na corrotina `download_one` de Exemplo 6, usei a seguinte linha para salvar a imagem baixada:

```
await asyncio.to_thread(save_flag, image, f'{cc}.gif')
```

Como mencionei antes, o `asyncio.to_thread` foi acrescentado no Python 3.9. Se você precisa suportar 3.7 ou 3.8, substitua aquela linha pelas linhas em Exemplo 10.

Exemplo 10. Linhas para usar no lugar de `await asyncio.to_thread`

```
loop = asyncio.get_running_loop()           ①
loop.run_in_executor(None, save_flag,       ②
                        image, f'{cc}.gif')  ③
```

- ① Obtém uma referência para o laço de eventos.
- ② O primeiro argumento é o executor a ser utilizado; passar `None` seleciona o default, um `ThreadPoolExecutor` que está sempre disponível no laço de eventos do `asyncio`.
- ③ Você pode passar argumentos posicionais para a função a ser executada, mas se você precisar passar argumentos nomeados, use `functool.partial`, como descrito na «documentação de `run_in_executor`» [fpy.li/ax].

A função mais nova `asyncio.to_thread` é mais fácil de usar e mais flexível, já que também aceita argumentos nomeados.

A própria implementação de `asyncio` usa `run_in_executor` debaixo dos panos em alguns pontos. Por exemplo, a corrotina `loop.getaddrinfo(...)`, que vimos no

Exemplo 1 é implementada invocando a função `getaddrinfo` do módulo `socket`—uma função bloqueante que pode levar alguns segundos para retornar, pois depende de resolução de DNS.

Um padrão comum em APIs assíncronas é encapsular em corrotinas quaisquer chamadas bloqueantes que sejam detalhes de implementação, e usar `run_in_executor` dentro das corrotinas para executar as chamadas bloqueantes. Assim é possível apresentar uma interface consistente de corrotinas a serem acionadas com `await`, escondendo as threads que precisam ser usadas por razões pragmáticas.

Por exemplo, o driver assíncrono do MongoDB para Python, chamado *Motor* [fpy.li/21-23], tem uma API compatível com `async/await` que na verdade é uma fachada, escondendo um banco de threads que conversa com o MongoDB. O criador do *Motor*, A. Jesse Jiryu Davis, explica suas razões em *Response to "Asynchronous Python and Databases"* [fpy.li/21-24] (Resposta a "Python Assíncrono e os Bancos de Dados"). Spoiler: Jiryu Davis descobriu que um banco de threads tem melhor desempenho no caso de uso específico de um driver de banco de dados—desmascarando o mito de que abordagens assíncronas são sempre mais eficientes que threads para E/S de rede.

A principal razão para passar um `Executor` explícito para `loop.run_in_executor` é utilizar um `ProcessPoolExecutor`, se a função ocupar intensivamente a CPU. Assim ela rodará em um processo Python diferente, evitando a disputa pela GIL. Por seu alto custo de inicialização, seria melhor iniciar o `ProcessPoolExecutor` no supervisor, e passá-lo para as corrotinas que precisem utilizá-lo.

O revisor Caleb Hattingh (autor de *Using Asyncio in Python* [fpy.li/hattingh]) me passou o seguinte aviso sobre executores e o `asyncio`.



O aviso de Caleb sobre `run_in_executors`

Usar `run_in_executor` pode produzir problemas difíceis de depurar, já que o cancelamento não funciona da forma esperada. Corrotinas que usam executores apenas fingem terminar: a thread subjacente (se for um `ThreadPoolExecutor`) não tem um mecanismo de cancelamento. Por exemplo, uma thread de longa duração criada dentro de uma chamada a `run_in_executor` pode impedir que seu programa `asyncio` encerre de forma limpa: `asyncio.run` vai esperar para retornar

até o executor terminar completamente, e vai esperar para sempre se os serviços iniciados pelo executor não pararem sozinhos de alguma forma. Minha barba branca preferia que o nome da função fosse `run_in_executor_uncancellable`.

Agora saímos de scripts clientes para escrever servidores com o `asyncio`.

21.9. Programando servidores assíncronos

O exemplo clássico de um servidor TCP de brinquedo é um servidor eco [fpy.li/7g]. Vamos escrever brinquedos um pouco mais interessantes: utilitários de servidor para busca de caracteres Unicode, primeiro usando HTTP com a *FastAPI*, depois usando TCP puro apenas com `asyncio`.

Estes servidores permitem que os usuários pesquisem caracteres Unicode buscando palavras que ocorrem em seus nomes fornecidos pelo módulo `unicodedata` que discutimos na «Seção 4.9» [fpy.li/cn] (vol.1). A Figura 2 mostra uma sessão com o `web_mojifinder.py`, o primeiro servidor que escreveremos.

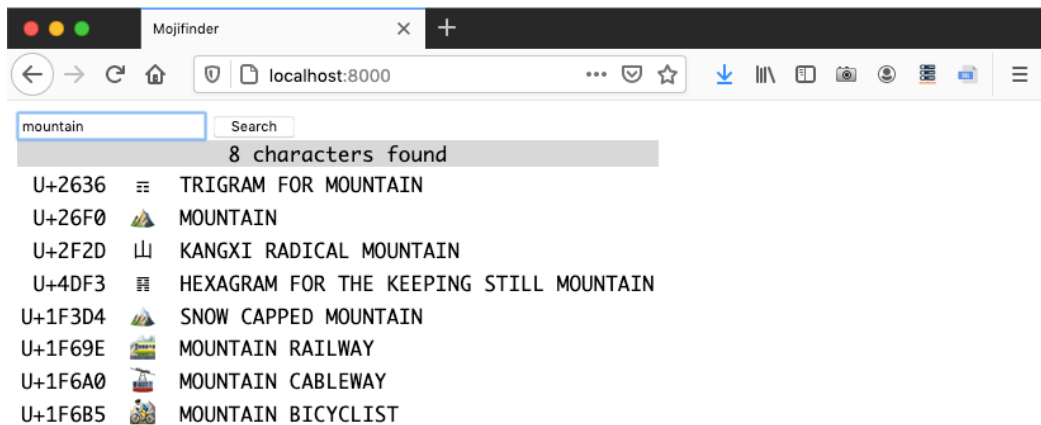


Figura 2. Janela de navegador mostrando os resultados da busca por "mountain" no serviço `web_mojifinder.py`.

A lógica de busca no Unicode nesses exemplos é a classe `InvertedIndex` no módulo `charindex.py` no «repositório de código do *Python Fluente*» [fpy.li/code]. Não há nada concorrente naquele pequeno módulo, então o box opcional a seguir contém apenas uma breve explicação sobre ele. Você pode pular para a implementação do servidor HTTP na Seção 21.9.1.

Um índice invertido normalmente mapeia palavras a documentos onde elas ocorrem. Nos exemplos *mojifinder*, cada "documento" é o nome de um caractere Unicode. A classe `charindex.InvertedIndex` indexa cada palavra que aparece no nome de cada caractere no banco de dados Unicode, e cria um índice invertido em um `defaultdict`. Por exemplo, para indexar o caractere U+0037—DIGIT SEVEN—o construtor de `InvertedIndex` anexa o caractere '7' aos registros sob as chaves 'DIGIT' e 'SEVEN'. Após indexar os dados do Unicode 13 incluídos no Python 3.10, 'DIGIT' será mapeado para 868 caracteres que têm esta palavra em seus nomes; e 'SEVEN' para 143, incluindo U+1F556—CLOCK FACE SEVEN OCLOCK e U+2790—DINGBAT NEGATIVE CIRCLED SANS-SERIF DIGIT SEVEN.

```
>>> from charindex import InvertedIndex
>>> idx.entries['CAT']
{'🐱', '🐾', '😺', '찰', '🔍', '🐼', '🐶', '🐩', '🐨', '🐯', '🐪', '🦊', '🐫', '🐇', '🐰', '🐹', '🐭', '🐮'}
>>> len(idx.entries['FACE'])
171
>>> idx.entries['FACE'] & idx.entries['CAT']
{'🐼', '🐾', '😺', '🐻', '🐨', '🐯', '🐪', '🦊', '🐫', '🐇', '🐰', '🐹'}
>>> idx.search('cat face')
{'🐼', '🐾', '😺', '🐻', '🐨', '🐯', '🐪', '🦊', '🐫', '🐇', '🐰', '🐹'}
```

O método `InvertedIndex.search` quebra a consulta em palavras separadas, e devolve a interseção dos registros para cada palavra. É por isso que buscar por "face" encontra 171 resultados, "cat" encontra 14, mas "cat face" apenas 10.

21.9. Programando servidores assíncronos

21.9.1. Um serviço Web com *FastAPI*

Escrevi o próximo exemplo—*web_mojifinder.py*—usando o *FastAPI* [fpy.li/21-28]: um dos frameworks ASGI para desenvolvimento Web em Python, mencionado na Seção 19.7.4. A Figura 2 é uma captura de tela da interface de usuário. É uma aplicação simples, de uma página só (SPA, *Single Page Application*): após o download inicial do HTML, a interface é atualizada via JavaScript no cliente, em comunicação com o servidor.

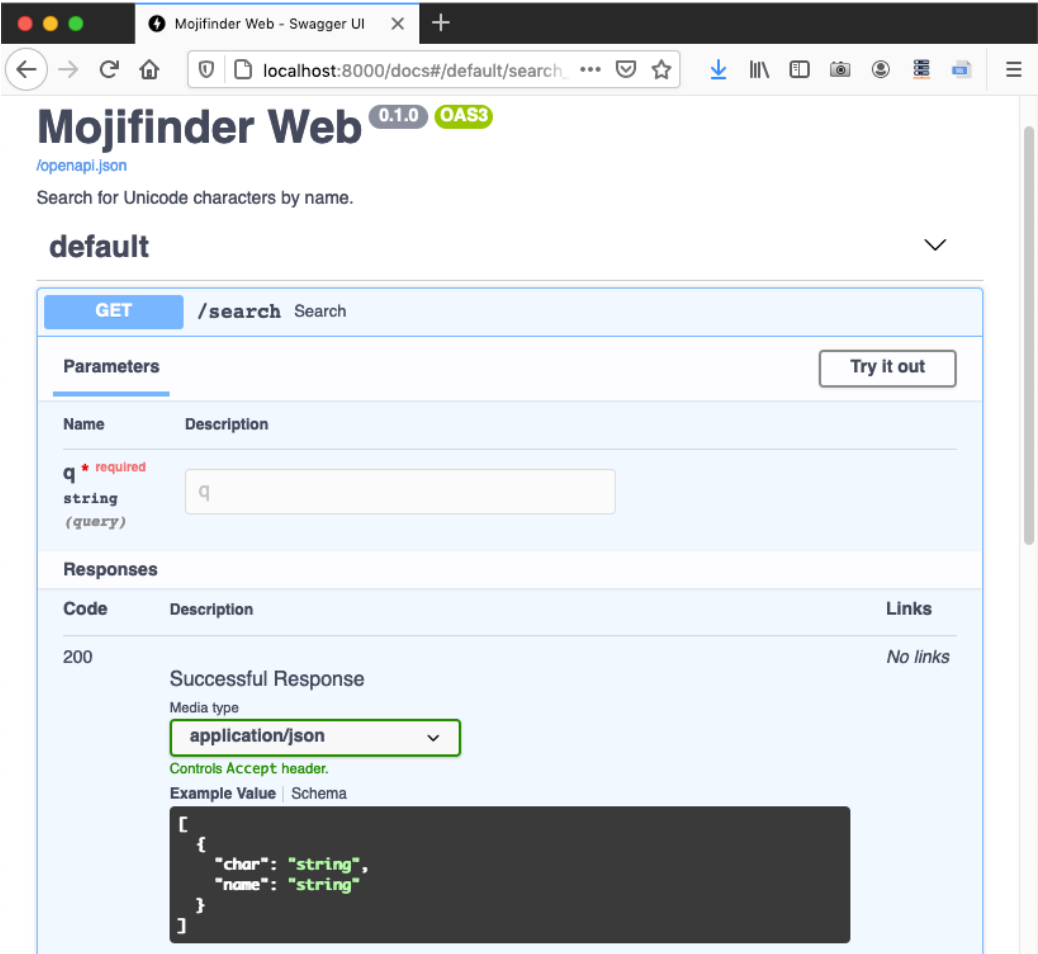


Figura 4. Documentação OpenAPI do ponto de acesso */search*, gerada automaticamente.

O *FastAPI* foi projetado para implementar o lado servidor de SPAs e aplicativos de celular, que consistem principalmente de pontos de acesso de APIs Web, devolvendo respostas JSON em vez de HTML renderizado no servidor. O *FastAPI* se vale de decoradores, dicas de tipo e introspecção de código para eliminar muito código repetitivo das APIs Web, e também gera uma documentação no padrão OpenAPI do *Swagger* [fpy.li/21-29]. A Figura 4 mostra a página /docs do *web_mojifinder.py*, gerada automaticamente.

O Exemplo 11 é o código de *web_mojifinder.py*, mas é só código do lado servidor. Quando você acessa a URL raiz /, o servidor envia o arquivo *form.html*, que contém 81 linhas de código, incluindo 54 linhas de JavaScript para comunicação com o servidor e preenchimento da tabela com os resultados. Se tiver interesse em ler JavaScript puro sem uso de frameworks, confira o *21-async/mojifinder/static/form.html* no «repositório de código» [fpy.li/code] do *Python Fluente*.

Para rodar o *web_mojifinder.py*, você precisa instalar dois pacotes e suas dependências: *FastAPI* e *uvicorn*.^[9]

Este é o comando para executar o Exemplo 11 com *uvicorn* em modo de desenvolvimento:

```
$ uvicorn web_mojifinder:app --reload
```

os parâmetros são:

web_mojifinder:app

O nome do pacote, dois pontos, e o nome da aplicação ASGI definida nele—*app* é o nome usado por convenção.

--reload

Faz o *uvicorn* monitorar mudanças no código-fonte da aplicação, e recarregá-la automaticamente. Útil apenas durante o desenvolvimento.

Vamos agora olhar o código-fonte do *web_mojifinder.py*.

Exemplo 11. *web_mojifinder.py*: código-fonte completo

```
from pathlib import Path
from unicodedata import name

from fastapi import FastAPI
from fastapi.responses import HTMLResponse
from pydantic import BaseModel

from charindex import InvertedIndex

STATIC_PATH = Path(__file__).parent.absolute() / 'static' ①

app = FastAPI( ②
    title='Mojifinder Web',
    description='Search for Unicode characters by name.',
)

class CharName(BaseModel): ③
    char: str
    name: str

def init(app): ④
    app.state.index = InvertedIndex()
    app.state.form = (STATIC_PATH / 'form.html').read_text()

init(app) ⑤

@app.get('/search', response_model=list[CharName]) ⑥
async def search(q: str): ⑦
    chars = sorted(app.state.index.search(q))
    return ({'char': c, 'name': name(c)} for c in chars) ⑧

@app.get('/', response_class=HTMLResponse, include_in_schema=False)
def form(): ⑨
    return app.state.form

# no main function ⑩
```

- ① Não relacionado ao tema desse capítulo, mas digno de nota: o uso elegante do operador / sobrecarregado por pathlib.^[10]

- ② Esta linha define a app ASGI. Ela poderia ser apenas `app = FastAPI()`. Os parâmetros são metadados para a documentação da API, gerada automaticamente.
- ③ Um schema *pydantic* para uma resposta JSON, com campos `char` e `name`. Como mencionado no «Capítulo 8» [fpy.li/8] (vol.1), o *pydantic* valida os dados durante a execução, com base nas anotações de tipo.
- ④ Cria o `index` e carrega o formulário HTML estático, anexando ambos ao `app.state` para uso posterior.
- ⑤ Roda `init` quando esse módulo é carregado pelo servidor ASGI.
- ⑥ Rota para o ponto de acesso `/search`; `response_model` usa aquele modelo `CharName` do *pydantic* para descrever o formato da resposta.
- ⑦ O *FastAPI* assume que qualquer parâmetro que apareça na assinatura da função ou da corrotina e que não esteja no caminho da rota será passado na string de consulta HTTP, isto é, `/search?q=cat`. Como `q` não tem default, a *FastAPI* devolverá um status 422 (Unprocessable Entity, *Entidade Não-Processável*) se `q` não estiver presente na string da consulta.
- ⑧ Devolver um iterável de dicts compatível com o schema `response_model` permite ao *FastAPI* criar uma resposta JSON de acordo com o `response_model` no decorador `@app.get`,
- ⑨ Funções regulares (isto é, não-assíncronas) também podem ser usadas para produzir respostas.
- ⑩ Este módulo não tem uma função principal. É carregado e acionado pelo servidor ASGI—neste exemplo, o *uvicorn*.

O Exemplo 11 não faz chamada direta ao `asyncio`. O *FastAPI* é construído sobre o toolkit ASGI *Starlette*, que por sua vez usa o `asyncio`.

Note também que o corpo de `search` não usa `await`, `async with`, ou `async for`, então poderia ser uma função comum. Defini `search` como uma corrotina apenas para mostrar que o *FastAPI* sabe como lidar com elas. Em uma aplicação real, a maioria dos pontos de acesso serão consultas a bancos de dados ou acessos a outros servidores remotos, então é uma vantagem importante do *FastAPI*—e dos frameworks ASGI em geral—suportarem corrotinas que podem se valer de bibliotecas assíncronas para E/S de rede.



As funções `init` e `form` para carregar e entregar o HTML estático do formulário são gambiarras para manter esse exemplo curto e fácil de rodar sem mais configurações.

A melhor prática é ter um proxy/balanceador de carga na frente do ASGI, servindo todos os recursos estáticos, e também usar uma *CDN* (Rede de Entrega de Conteúdo) quando possível.

Um proxy/balanceador de carga deste tipo é o *Traefik* [fpy.li/21-32], descrito como um *edge router* (roteador de ponta), que "recebe requisições em nome de seu sistema e descobre quais componentes são responsáveis por lidar com elas." O site do *FastAPI* apresenta «ferramentas de geração de projeto» [fpy.li/c5/] que organizam o código para usar o *Trafik* e outros softwares auxiliares.

Os entusiastas da tipagem estática podem ter notado que não coloquei dicas de tipo para os resultados devolvidos por `search` e `form`. Em vez disto, o *FastAPI* aceita o argumento nomeado `response_model=` nos decoradores de rota. A página *Response Model - Return Type* [fpy.li/21-34] da documentação do *FastAPI* explica:

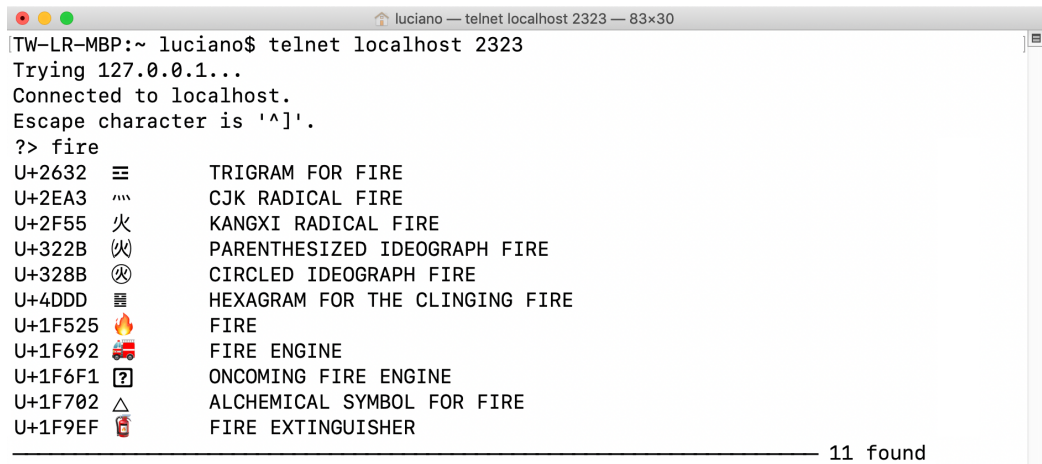
O modelo de resposta é declarado neste parâmetro em vez de como uma anotação de tipo de resultado devolvido por uma função, porque a função de rota pode não devolver aquele modelo de resposta mas sim um dict, um objeto do banco de dados ou algum outro modelo, e então usar o response_model para realizar a validação de campos e a serialização.

Por exemplo, em `search`, devolvi um gerador de itens `dict` e não uma lista de objetos `CharName`, mas isso basta para o *FastAPI* e o *pydantic* validarem meus dados e construir a resposta JSON apropriada, compatível com `response_model=list[CharName]`.

Agora vamos analisar outro servidor que usa *sockets* TCP e a biblioteca padrão do Python para responder consultas via Telnet no terminal.

21.9.2. Um servidor TCP com asyncio

O programa `tcp_mojifinder.py` usa TCP puro para se comunicar com um cliente como o Telnet ou o Netcat, então pode escrevê-lo usando `asyncio` sem dependências externas, e sem reinventar o HTTP. A Figura 5 mostra a interface de usuário em modo texto.



```
luciano — telnet localhost 2323 — 83x30
TW-LR-MBP:~ luciano$ telnet localhost 2323
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
?> fire
U+2632  ≡      TRIGRAM FOR FIRE
U+2EA3  𐤒      CJK RADICAL FIRE
U+2F55  火      KANGXI RADICAL FIRE
U+322B  (火)    PARENTHE sized IDEOGRAPH FIRE
U+328B  (𐤒)    CIRCLED IDEOGRAPH FIRE
U+4DDD  𠔁      HEXAGRAM FOR THE CLINGING FIRE
U+1F525  🔥      FIRE
U+1F692  🚒      FIRE ENGINE
U+1F6F1  🚒      ONCOMING FIRE ENGINE
U+1F702  ⚗      ALCHEMICAL SYMBOL FOR FIRE
U+1F9EF  🧯      FIRE EXTINGUISHER

11 found
```

Figura 5. Sessão de telnet com o servidor `tcp_mojifinder.py`: consultando "fire."

Este programa é duas vezes mais longo que o `web_mojifinder.py` (descontando o HTML e o JavaScript daquele exemplo). Por isso dividi a apresentação em três partes: Exemplo 12, Exemplo 14, e Exemplo 15.

O topo do arquivo `tcp_mojifinder.py`, com as instruções `import`, está no Exemplo 14. Mas vou começar descrevendo a corrotina `supervisor` e a função `main` que controlam o programa.

Exemplo 12. `tcp_mojifinder.py`: um servidor TCP simples; continua no Exemplo 14

```
async def supervisor(index: InvertedIndex, host: str, port: int) -> None:
    server = await asyncio.start_server(      ①
        functools.partial(finder, index),    ②
        host, port)                          ③

    socket_list = cast(tuple[TransportSocket, ...], server.sockets) ④
    addr = socket_list[0].getsockname()
    print(f'Serving on {addr}. Hit CTRL-C to stop.') ⑤
    await server.serve_forever()              ⑥
```

```
def main(host: str = '127.0.0.1', port_arg: str = '2323'):
    port = int(port_arg)
    print('Building index.')
    index = InvertedIndex() ⑦
    try:
        asyncio.run(supervisor(index, host, port)) ⑧
    except KeyboardInterrupt: ⑨
        print('\nServer shut down.')

if __name__ == '__main__':
    main(*sys.argv[1:])
```

- ① Este `await` devolve uma instância de `asyncio.Server`, um servidor TCP baseado em `sockets`. Por padrão, `start_server` cria e inicia o servidor, então ele está pronto para receber conexões.
- ② O primeiro argumento para `start_server` é `client_connected_cb`, uma função de *callback* para rodar ao iniciar a conexão com cada cliente. Pode ser uma função ou uma corrotina, mas deve aceitar exatamente dois argumentos: um `asyncio.StreamReader` e um `asyncio.StreamWriter`. Minha corrotina `finder` também precisa receber um `index`, por isso usei `functools.partial` para vincular aquele parâmetro e obter um invocável que recebe o leitor (`StreamReader`) e o escritor (`StreamWriter`).
- ③ `host` e `port` são o segundo e o terceiro argumentos de `start_server`. Veja a assinatura completa na «documentação do `asyncio`» [fpy.li/b2].
- ④ Este cast é necessário porque o `typeshed` tem uma dica de tipo desatualizada para a propriedade `sockets` da classe `Server` em maio de 2021. Veja «Issue #5535 no `typeshed`» [fpy.li/21-36].^[11]
- ⑤ Exibe o endereço e a porta do primeiro *socket* do servidor.
- ⑥ Apesar de `start_server` já ter iniciado o servidor como uma tarefa concorrente, preciso fazer `await` em `serve_forever`, para que meu supervisor seja suspenso aqui. Do contrário, o supervisor retornaria imediatamente, encerrando o laço iniciado com `asyncio.run(supervisor(...))`, e fechando o programa. A documentação de `Server.serve_forever` [fpy.li/b3] diz: "Este método pode ser chamado se o servidor já estiver aceitando conexões."
- ⑦ Constrói o índice invertido.^[12]

- ⑧ Inicia o laço de eventos rodando supervisor.
- ⑨ Captura KeyboardInterrupt para evitar um traceback ruidoso quando encerramos o servidor teclando CTRL-C no terminal onde ele está rodando.

Pode ser mais fácil entender como o controle flui em *tcp_mojifinder.py* estudando a saída que ele gera no console do servidor, listada no Exemplo 13.

Exemplo 13. tcp_mojifinder.py: o lado servidor da sessão mostrada na Figura 5

```
$ python3 tcp_mojifinder.py
Building index. ①
Serving on ('127.0.0.1', 2323). Hit CTRL-C to stop. ②
  From ('127.0.0.1', 58192): 'cat face' ③
    To ('127.0.0.1', 58192): 10 results.
  From ('127.0.0.1', 58192): 'fire' ④
    To ('127.0.0.1', 58192): 11 results.
  From ('127.0.0.1', 58192): '\x00' ⑤
Close ('127.0.0.1', 58192). ⑥
^C ⑦
Server shut down. ⑧
$
```

- ① Saída de `main`. Antes da próxima linha aparecer, notei um intervalo de 0,6s na minha máquina, enquanto o índice era construído.
- ② Saída de supervisor.
- ③ Primeira volta do laço `while` na função `finder` do Exemplo 14. A pilha TCP/IP atribuiu a porta 58192 a meu cliente Telnet. Se você conectar diversos clientes ao servidor, verá suas diferentes portas aparecerem na saída.
- ④ Segunda iteração do laço `while` em `finder`.
- ⑤ Teclei CTRL-C no terminal do cliente; o laço `while` em `finder` termina.
- ⑥ A corrotina `finder` exibe esta mensagem e encerra. Enquanto isso o servidor continua rodando, pronto para receber outros clientes.
- ⑦ Teclei CTRL-C no terminal do servidor; `server.serve_forever` é cancelado, encerrando supervisor e o laço de eventos.
- ⑧ Saída de `main`.

Após `main` construir o índice e iniciar o laço de eventos, a corrotina supervisor rapidamente exibe a mensagem `Serving on...`, e fica suspensa na última linha:

```
await server.serve_forever()
```

Neste ponto o controle flui para o laço de eventos do `asyncio` e lá permanece, voltando ocasionalmente para a corrotina `finder`, que devolve o controle de volta para o laço de eventos sempre que precisa esperar a rede para enviar ou receber dados.

Enquanto o laço de eventos estiver ativo, uma nova instância da corrotina `finder` será iniciada para cada cliente que se conecte ao servidor. Desta forma, milhares de clientes podem ser atendidos concorrentemente por este servidor simples. Isto segue até que ocorra um `KeyboardInterrupt` no servidor ou que seu processo seja encerrado pelo SO.

Agora vamos ver o início de `tcp_mojifinder.py`, com a corrotina `finder`.

Exemplo 14. tcp_mojifinder.py: continuação de Exemplo 12

```
import asyncio
import functools
import sys
from asyncio.trsock import TransportSocket
from typing import cast

from charindex import InvertedIndex, format_results ①

CRLF = b'\r\n'
PROMPT = b'?> '

async def finder(index: InvertedIndex, ②
                 reader: asyncio.StreamReader,
                 writer: asyncio.StreamWriter) -> None:
    client = writer.get_extra_info('peername') ③
    while True: ④
        writer.write(PROMPT) # can't await! ⑤
        await writer.drain() # must await! ⑥
        data = await reader.readline() ⑦
        if not data: ⑧
```



```

        break
    try:
        query = data.decode().strip() ⑨
    except UnicodeDecodeError: ⑩
        query = '\x00'
    print(f' From {client}: {query!r}') ⑪
    if query:
        if ord(query[:1]) < 32: ⑫
            break
        results = await search(query, index, writer) ⑬
        print(f' To {client}: {results} results.') ⑭

writer.close() ⑮
await writer.wait_closed() ⑯
print(f'Close {client}.') ⑰

```

- ① `format_results` é útil para mostrar os resultados de `InvertedIndex.search` em uma interface de usuário baseada em texto, como a linha de comando ou uma sessão Telnet.
- ② Para passar `finder` para `asyncio.start_server`, a envolvi com `functools.partial`, porque o servidor espera uma corrotina ou função que receba apenas os argumentos `reader` e `writer`.
- ③ Obtém o endereço do cliente remoto ao qual o socket está conectado.
- ④ Este laço controla um diálogo que persiste até um caractere de controle ser recebido do cliente.
- ⑤ O método `StreamWriter.write` não é uma corrotina, é uma função comum. Esta linha envia o prompt `?>`.
- ⑥ `StreamWriter.drain` esvazia o buffer de `writer`; ela é uma corrotina, então precisa ser acionada com `await`.
- ⑦ `StreamWriter.readline` é uma corrotina que devolve bytes.
- ⑧ Se nenhum byte foi recebido, o cliente fechou a conexão, então sai do loop.
- ⑨ Decodifica os bytes para str, usando a codificação UTF-8 como default.
- ⑩ Pode ocorrer um `UnicodeDecodeError` quando o usuário digita CTRL-C e o cliente Telnet envia caracteres de controle; se isso acontecer, substitui a consulta pelo caractere null, para simplificar.

- ⑪ Registra a consulta no console do servidor.
- ⑫ Sai do laço se um caractere de controle ou null foi recebido.
- ⑬ `search` realiza a busca; o código será apresentado a seguir.
- ⑭ Registra a resposta no console do servidor.
- ⑮ Fecha o `StreamWriter`.
- ⑯ Espera até `StreamWriter` fechar. Isso é recomendado na documentação do método `.close()` [fpy.li/b4].
- ⑰ Registra o final dessa sessão do cliente no console do servidor.

O último trecho deste código é a corrotina `search`, Exemplo 15.

Exemplo 15. `tcp_mojifinder.py`: corrotina `search`

```
async def search(query: str, ①
                    index: InvertedIndex,
                    writer: asyncio.StreamWriter) -> int:
    chars = index.search(query) ②
    lines = (line.encode() + CRLF for line ③
             in format_results(chars))
    writer.writelines(lines) ④
    await writer.drain() ⑤
    status_line = f'{"—" * 66} {len(chars)} found' ⑥
    writer.write(status_line.encode() + CRLF)
    await writer.drain()
    return len(chars)
```

- ① `search` precisa ser uma corrotina, pois escreve em um `StreamWriter` e precisa acionar o método corrotina `.drain()`.
- ② Consulta o índice invertido.
- ③ Esta expressão geradora produzirá strings de bytes codificadas em UTF-8 com o ponto de código Unicode, o caractere, seu nome e uma sequência CRLF (*Return+Line Feed*), isto é, `b'U+0039\t9\tDIGIT NINE\r\n'`.
- ④ Envia `lines`. Surpreendentemente, `writer.writelines` não é uma corrotina.
- ⑤ Mas `writer.drain()` é uma corrotina. Não esqueça do `await`!
- ⑥ Constrói e envia uma linha de status.

Observe que toda a E/S de rede em *tcp_mojifinder.py* é feita em bytes; precisamos decodificar os bytes recebidos da rede, e codificar strings antes de enviá-las. No Python 3, a codificação default é UTF-8, e foi o que usei implicitamente em todas as chamadas a `encode` e `decode` nesse exemplo.



Note que alguns dos métodos de E/S são corrotinas, e precisam ser acionados com `await`, enquanto outros são funções comuns. Por exemplo, `StreamWriter.write` é uma função, porque escreve em um buffer. Por outro lado, `StreamWriter.drain`—que esvazia o buffer e executa o E/S de rede—é uma corrotina, assim como `StreamReader.readline`—mas não `StreamWriter.writelines`! Enquanto escrevi a primeira edição desse livro, sugeri uma melhoria na «documentação da API» [fpy.li/b5] do `asyncio` para indicar com mais clareza as corrotinas (antes era preciso ler todo o texto sobre uma corrotina para encontrar a informação, porque elas eram formatadas como as funções).

O código de *tcp_mojifinder.py* se vale da «API de streams» [fpy.li/21-40] de alto nível do `asyncio`, que fornece um servidor pronto para usar, então só precisamos codar uma função de processamento, que pode ser um callback simples ou uma corrotina. Há também uma «API de Transportes e Protocolos» [fpy.li/bb] de baixo nível, inspirada nas abstrações de transporte e protocolo do framework *Twisted*. Veja a documentação do `asyncio` para mais informações, incluindo os «servidores echo e clientes TCP e UDP» [fpy.li/bc] implementados com a API de baixo nível.

Nosso próximo tópico é a instrução `async for` e os objetos que a fazem funcionar.

21.10. Iteráveis assíncronos

Na Seção 21.6 vimos como `async with` funciona com objetos que implementam os métodos `__aenter__` e `__aexit__`, devolvendo esperáveis—normalmente na forma de objetos corrotina.

De forma análoga, `async for` funciona com *iteráveis assíncronos*: objetos que implementam `__aiter__`. Entretanto, `__aiter__` precisa ser um método normal—não um método corrotina—e precisa devolver um *iterador assíncrono*.

Um iterador assíncrono fornece um método corrotina `__anext__` que devolve um esperável—muitas vezes um objeto corrotina. Também se espera que eles implementem `__aiter__`, que normalmente devolve `self`. Isso espelha a importante distinção entre iteráveis e iteradores que vimos na Seção 17.5.2.

A «documentação» [fpy.li/21-43] do driver assíncrono de PostgreSQL *aiopg* traz um exemplo que ilustra o uso de `async for` para iterar sobre as linhas de resultados devolvidas pelo objeto cursor, definido no driver do banco de dados.

```
async def go():
    pool = await aiopg.create_pool(dsn)
    async with pool.acquire() as conn:
        async with conn.cursor() as cur:
            await cur.execute("SELECT 1")
            ret = []
            async for row in cur:
                ret.append(row)
            assert ret == [(1,)]
```

Neste exemplo, a consulta vai devolver só uma linha, mas em um cenário realista é possível receber milhares de linhas na resposta a um `SELECT`. Para respostas grandes, o cursor não será carregado com todas as linhas de uma vez só. Por isso é importante que `async for row in cur:` não bloqueie o laço de eventos enquanto o cursor pode estar esperando por linhas adicionais. Ao implementar o cursor como um iterador assíncrono, *aiopg* pode devolver o controle para o laço de eventos a cada chamada a `__anext__`, e continuar mais tarde, quando mais linhas chegarem do PostgreSQL.

21.10.1. Funções geradoras assíncronas

Você pode implementar um iterador assíncrono escrevendo uma classe com `__anext__` e `__aiter__`, mas há um jeito mais fácil: escreva uma função declarada com `async def` que use `yield` em seu corpo. Isto é semelhante à forma como funções geradoras simplificam o implementar o padrão do Iterador clássico.

Vamos estudar um exemplo simples usando `async for` e implementando um gerador assíncrono. No Exemplo 1 vimos *blogdom.py*, um script que sondava nomes de domínio. Suponha agora que encontramos outros usos para a corrotina

probe definida ali, e decidimos colocá-la em um novo módulo (*domainlib.py*) com um novo gerador assíncrono `multi_probe`, que recebe uma lista de nomes de domínio e produz resultados conforme eles são sondados.

Vamos ver a implementação de *domainlib.py* logo, mas primeiro examinaremos como ele é usado com o novo console assíncrono de Python.

21.10.1.1. Experimentando com o console assíncrono de Python

Desde o Python 3.8 [fpy.li/21-44], é possível rodar o interpretador com a opção de linha de comando `-m asyncio`, para obter um "async REPL": um console de Python que importa `asyncio`, fornece um laço de eventos ativo, e aceita `await`, `async for`, e `async with` no prompt principal—que em qualquer outro contexto são erros de sintaxe quando usados fora de corrotinas nativas.^[13]

Para experimentar com o *domainlib.py*, vá ao diretório `21-async/domains/asyncio/` na sua cópia local do «repositório de código» [fpy.li/code] do *Python Fluente*. Então execute:

```
$ python -m asyncio
```

Você verá o console iniciar, mais ou menos assim:

```
asyncio REPL 3.9.1 (v3.9.1:1e5d33e9b9, Dec 7 2020, 12:10:52)
[Clang 6.0 (clang-600.0.57)] on darwin
Use "await" directly instead of "asyncio.run()".
Type "help", "copyright", "credits" or "license" for more information.
>>> import asyncio
>>>
```

Note como o cabeçalho diz que você pode usar `await` em vez de `asyncio.run()` para acionar corrotinas e outros esperáveis. E mais: eu não digitei `import asyncio`. O módulo `asyncio` é automaticamente importado e a instrução `import asyncio` é exibida para deixar isso evidente.

Vamos agora importar *domainlib.py* e brincar com suas duas corrotinas: `probe` e `multi_probe` (Exemplo 16).

Exemplo 16. Experimentando com domainlib.py após executar python3 -m asyncio

```
>>> await asyncio.sleep(3, 'Bom dia!') ①
'Bom dia!'
>>> from domainlib import *
>>> await probe('python.org') ②
Result(domain='python.org', found=True) ③
>>> names = 'python.org rust-lang.org golang.org xyz.invalid'.split() ④
>>> async for result in multi_probe(names): ⑤
...     print(*result, sep='\t')
...
golang.org      True    ⑥
xyz.invalid     False
python.org      True
rust-lang.org   True
>>>
```

- ① Experimente um simples `await` para ver o console assíncrono em ação. Dica: `asyncio.sleep()` pode receber um segundo argumento opcional que será devolvido através do `await`.
- ② Acione a corrotina `probe`.
- ③ A versão de `probe` em `domainlib` devolve uma `NamedTuple` chamada `Result`.
- ④ Faça uma lista de domínios. O domínio de nível superior `.invalid` é reservado para testes. Consultas ao DNS por tais domínios sempre recebem uma resposta `NXDOMAIN` dos servidores DNS, que significa "este domínio não existe."^[14]
- ⑤ Itera com `async for` sobre o gerador assíncrono `multi_probe` para exibir os resultados.
- ⑥ Note que os resultados não estão na ordem em que os domínios foram enviados a `multi_probe`. Eles aparecem quando cada resposta do DNS chega.

O Exemplo 16 mostra que `multi_probe` é um gerador assíncrono, pois é compatível com `async for`. Vamos executar mais alguns experimentos, continuando com o Exemplo 17.

Exemplo 17. Mais experimentos, continuação do Exemplo 16

```
>>> probe('python.org') ①
<coroutine object probe at 0x10e313740>
>>> multi_probe(names) ②
<async_generator object multi_probe at 0x10e246b80>
>>> for r in multi_probe(names): ③
...     print(r)
...
Traceback (most recent call last):
...
TypeError: 'async_generator' object is not iterable
```

- ① Invocar uma corrotina nativa devolve um objeto corrotina.
- ② Invocar um gerador assíncrono devolve um objeto `async_generator`.
- ③ Não podemos usar um laço `for` comum para percorrer geradores assíncronos, porque eles implementam `__aiter__` em vez de `__iter__`.

Geradores assíncronos são acionados pelas palavras-chave `async` `for`, que pode ser uma instrução de laço (como visto em Exemplo 16), mas também podem aparecer em compreensões assíncronas, que veremos mais tarde.

21.10.1.2. Implementando um gerador assíncrono

Aqui está o módulo *domainlib.py*, com o gerador assíncrono `multi_probe`:

Exemplo 18. *domainlib.py*: funções para sondar domínios

```
import asyncio
import socket
from collections.abc import Iterable, AsyncIterator
from typing import NamedTuple, Optional

class Result(NamedTuple): ①
    domain: str
    found: bool

OptionalLoop = Optional[asyncio.AbstractEventLoop] ②
```

```

async def probe(domain: str, loop: OptionalLoop = None) -> Result: ③
    if loop is None:
        loop = asyncio.get_running_loop()
    try:
        await loop.getaddrinfo(domain, None)
    except socket.gaierror:
        return Result(domain, False)
    return Result(domain, True)

async def multi_probe(
    domains: Iterable[str]) -> AsyncIterator[Result]: ④
    loop = asyncio.get_running_loop()
    coros = [probe(domain, loop) for domain in domains] ⑤
    for coro in asyncio.as_completed(coros): ⑥
        result = await coro ⑦
        yield result ⑧

```

- ① NamedTuple torna o resultado de probe mais fácil de ler e depurar.
- ② Este apelido de tipo serve para evitar que a linha seguinte fique grande demais em uma listagem impressa em um livro.
- ③ probe agora recebe um argumento opcional loop, para evitar chamadas repetidas a get_running_loop toda vez que esta corrotina é acionada no laço em multi_probe.
- ④ Uma função geradora assíncrona produz um objeto gerador assíncrono, que pode ser anotado como AsyncIterator[TipoDoItem].
- ⑤ Constrói uma lista de objetos corrotina probe, cada um com um domain diferente.
- ⑥ Isto não é async for porque asyncio.as_completed é um gerador clássico.
- ⑦ Aciona o objeto corrotina para obter o resultado.
- ⑧ Produz um result. Esta linha faz com que multi_probe seja um gerador assíncrono.



O corpo do laço for no Exemplo 18 poderia ser mais conciso:

```
for coro in asyncio.as_completed(coros):  
    yield await coro
```

Python interpreta isso como `yield (await coro)`, então funciona. Achei que poderia ser confuso usar esse atalho no primeiro exemplo de gerador assíncrono no livro, então dividi em duas linhas.

Uma vez que temos o *domainlib.py*, podemos demonstrar o uso do gerador assíncrono `multi_probe` em *domaincheck.py*: um script que recebe um sufixo de domínio e busca por domínios criados a partir de palavras-chave curtas de Python.

Aqui está uma amostra da saída de *domaincheck.py*:

```
$ ./domaincheck.py net  
FOUND          NOT FOUND  
=====        =====  
in.net  
del.net  
true.net  
for.net  
is.net  
  
                none.net  
try.net  
  
                from.net  
  
and.net  
or.net  
else.net  
with.net  
if.net  
as.net  
  
                elif.net  
                pass.net  
                not.net  
                def.net
```

Graças à *domainlib*, o código de *domaincheck.py* é bem direto:

Exemplo 19. domaincheck.py: utilitário para sondar domínios usando domainlib

```
#!/usr/bin/env python3
import asyncio
import sys
from keyword import kwlist

from domainlib import multi_probe

async def main(tld: str) -> None:
    tld = tld.strip('.')
    names = (kw for kw in kwlist if len(kw) <= 4) ①
    domains = (f'{name}.{tld}'.lower() for name in names) ②
    print('FOUND\t\tNOT FOUND') ③
    print('====\t\t=====')
    async for domain, found in multi_probe(domains): ④
        indent = ' ' if found else '\t\t' ⑤
        print(f'{indent}{domain}')

if __name__ == '__main__':
    if len(sys.argv) == 2:
        asyncio.run(main(sys.argv[1])) ⑥
    else:
        print('Please provide a TLD.', f'Example: {sys.argv[0]} COM.BR')
```

- ① Gera palavras-chave de tamanho até 4.
- ② Gera nomes de domínio com o sufixo recebido como TLD (*Top Level Domain*).
- ③ Formata um cabeçalho para a saída tabular.
- ④ Itera de forma assíncrona sobre `multi_probe(domains)`.
- ⑤ Define `indent` como zero ou dois tabs, para colocar o resultado na coluna apropriada.
- ⑥ Roda a corrotina `main` com o argumento de linha de comando passado.

Geradores têm uma outra utilidade, não relacionado à iteração: eles podem ser usados como gerenciadores de contexto. Isso também se aplica aos geradores assíncronos.

21.10.1.3. Geradores assíncronos como gerenciadores de contexto

Escrever nossos próprios gerenciadores de contexto assíncronos não é uma tarefa de programação frequente, mas se precisar, considere usar o decorador `@asynccontextmanager` [fpy.li/b6], incluído no módulo `contextlib` no Python 3.7. É análogo ao decorador `@contextmanager` que estudamos na Seção 18.2.2.

Um exemplo interessante da combinação de `@asynccontextmanager` com `loop.run_in_executor` aparece no livro de Caleb Hattingh, *Using Asyncio in Python* [fpy.li/hattingh]. O Exemplo 20 é o código de Caleb—com uma única mudança e o acréscimo das explicações.

Exemplo 20. Exemplo usando `@asynccontextmanager` e `loop.run_in_executor`

```
from contextlib import asynccontextmanager

@asynccontextmanager
async def web_page(url): ①
    laço = asyncio.get_running_loop() ②
    data = await loop.run_in_executor( ③
        None, download_webpage, url)
    yield data ④
    await loop.run_in_executor(None, update_stats, url) ⑤

async with web_page('google.com') as data: ⑥
    process(data)
```

- ① A função decorada precisa ser um gerador assíncrono.
- ② Pequena atualização no código de Caleb: usar o `get_running_loop`, mais leve, no lugar de `get_event_loop`.
- ③ Suponha que `download_webpage` é uma função bloqueante que usa a biblioteca *requests*; vamos rodá-la em uma thread separada, para evitar o bloqueio do laço de eventos.
- ④ Todas as linhas antes dessa expressão `yield` vão se tornar o método corrotina `__aenter__` do gerenciador de contexto assíncrono criado pelo decorador. O valor de `data` será vinculado à variável `data` após a cláusula `as` no comando `async with` abaixo.

- ⑤ As linhas após o `yield` se tornarão o método corrotina `__aexit__`. Aqui outra chamada bloqueante é delegada para um executor de threads.
- ⑥ Usa `web_page` com `async with`.

Isso é muito similar ao decorador sequencial `@contextmanager`. Por favor, consulte a Seção 18.2.2 para mais detalhes, inclusive o tratamento de erro na linha do `yield`. Para outro exemplo usando `@asynccontextmanager`, veja a documentação do `contextlib` [fpy.li/b6].

Por fim, vamos terminar nossa jornada pelas funções geradoras assíncronas comparando-as com as corrotinas nativas.

21.10.1.4. Geradores assíncronos versus corrotinas nativas

Aqui estão algumas semelhanças e diferenças fundamentais entre uma corrotina nativa e uma função geradora assíncrona:

- Ambas são declaradas com `async def`.
- Um gerador assíncrono sempre tem uma expressão `yield` em seu corpo—é isso que o torna um gerador. Uma corrotina nativa nunca contém um `yield`.
- Uma corrotina nativa pode devolver (`return`) algum valor diferente de `None`, mas um gerador assíncrono só pode usar instruções `return` vazias.
- Corrotinas nativas são esperáveis: elas podem ser acionadas por expressões `await` ou passadas para uma das muitas funções do `asyncio` que aceitam argumentos esperáveis, como `create_task` ou `gather`. Em contrapartida, geradores assíncronos não são esperáveis. Eles são iteráveis assíncronos, acionados por `async for` ou por compreensões assíncronas.

Hora de falar sobre as tais compreensões assíncronas.

21.10.2. Compreensões assíncronas e expressões geradoras assíncronas

A PEP 530—*Asynchronous Comprehensions* [fpy.li/pep530] introduziu o uso de `async for` e `await` na sintaxe de compreensões e expressões geradoras, a partir do Python 3.6.

A única sintaxe definida na PEP 530 que pode aparecer fora do corpo de uma `async def` é uma expressão geradora assíncrona.

21.10.2.1. Definindo e usando uma expressão geradora assíncrona

Dado o gerador assíncrono `multi_probe` do Exemplo 18, poderíamos escrever outro gerador assíncrono que devolvesse apenas os nomes de domínios encontrados. Aqui está uma forma de fazer isso—novamente usando o console assíncrono iniciado com `-m asyncio`:

```
>>> from domainlib import multi_probe
>>> names = 'python.org rust-lang.org golang.org xyz.invalid'.split()
>>> gen_found = (name async for name, found
...               in multi_probe(names) if found) ①
>>> gen_found
<async_generator object <genexpr> at 0x10a8f9700> ②
>>> async for name in gen_found: ③
...     print(name)
...
golang.org
python.org
rust-lang.org
```

- ① O uso de `async for` define uma expressão geradora assíncrona. Ela pode ser definida em qualquer lugar de um módulo Python.
- ② A expressão geradora assíncrona cria um objeto `async_generator`—exatamente o mesmo tipo de objeto devolvido por uma função geradora assíncrona como `multi_probe`.
- ③ O objeto gerador assíncrono é acionado pela instrução `async for`, que por sua vez só pode aparecer dentro do corpo de uma `async def` ou no console assíncrono mágico que usei nesse exemplo.

Resumindo: uma expressão geradora assíncrona pode ser definida em qualquer ponto do seu programa, mas só pode ser acionada dentro de uma corrotina nativa ou de uma função geradora assíncrona.

Agora veremos as demais construções sintáticas propostas na PEP 530.

21.10.2.2. Compreensões assíncronas

Diferente das expressões geradoras assíncronas, as compreensões assíncronas só podem ser definidas e usadas dentro de corrotinas nativas ou de funções geradoras assíncronas. Isso faz sentido porque as compreensões são ávidas (*eager*): elas são executadas imediatamente para construir uma lista. Em contraste, as expressões geradoras (assíncronas ou não), são preguiçosas (*lazy*). Elas criam um objeto gerador que só será executado quando um laço percorrer o gerador.

Yury Selivanov—autor da PEP 530—justificou a necessidade de compreensões assíncronas com três trechos curtos de código, reproduzidos a seguir.

Podemos concordar que deveria ser possível reescrever esse código:

```
result = []
async for i in aiter():
    if i % 2:
        result.append(i)
```

assim:

```
result = [i async for i in aiter() if i % 2]
```

Além disso, dada uma corrotina nativa `fun`, deveria ser possível escrever isso:

```
result = [await fun() for fun in funcs]
```



Usar `await` em uma compreensão de lista é similar a usar `asyncio.gather`. Mas `gather` nos dá um maior controle sobre o tratamento de exceções, graças ao seu argumento opcional `return_exceptions`. Caleb Hattingh recomenda sempre definir `return_exceptions=True` (o default é `False`). Veja a «documentação de `asyncio.gather`» [fpy.li/b7] para mais informações.

Voltemos ao console assíncrono mágico:

```
>>> names = 'python.org rust-lang.org golang.org xyz.invalid'.split()
>>> names = sorted(names)
>>> coros = [probe(name) for name in names]
>>> await asyncio.gather(*coros)
[Result(domain='golang.org', found=True),
Result(domain='xyz.invalid', found=False),
Result(domain='python.org', found=True),
Result(domain='rust-lang.org', found=True)]
>>> [await probe(name) for name in names]
[Result(domain='golang.org', found=True),
Result(domain='xyz.invalid', found=False),
Result(domain='python.org', found=True),
Result(domain='rust-lang.org', found=True)]
>>>
```

Usei `sorted` para ordenar a lista de nomes e mostrar que os resultados chegam na ordem em que foram submetidos, nos dois casos.

A PEP 530 também permite o uso de `async for` e `await` compreensões de `dict` e de `set`. Por exemplo, aqui está uma compreensão de `dict` para armazenar os resultados de `multi_probe` no console assíncrono:

```
>>> {name: found async for name, found in multi_probe(names)}
{'golang.org': True, 'python.org': True, 'xyz.invalid': False,
 'rust-lang.org': True}
```

Podemos usar a palavra-chave `await` na expressão antes das cláusulas `for` ou `async for`, e também na expressão após a cláusula `if`. Aqui está uma compreensão de `set` no console assíncrono, coletando apenas os domínios encontrados.

```
>>> {name for name in names if (await probe(name)).found}
{'rust-lang.org', 'python.org', 'golang.org'}
```

Precisei colocar parênteses adicionais ao redor da expressão `await` devido à precedência mais alta do operador `.` (ponto) de `__getattr__`.

Relembrando, todas essas compreensões só podem aparecer no corpo de uma `async def` ou no console assíncrono encantado.

Agora vamos discutir uma característica muito importante das instruções e expressões `async` e dos objetos que eles criam: Estas construções são muito usadas com o `asyncio` mas, na verdade, são independentes da biblioteca.

21.11. Sondando domínios com *Curio*

Os elementos da linguagem `async/await` de Python não estão presos a nenhum laço de eventos ou biblioteca específicos.^[15] Graças à API extensível fornecida por métodos especiais, qualquer pessoa suficientemente motivada pode escrever seu ambiente de runtime e um framework assíncrono para acionar corrotinas nativas, geradores assíncronos, etc.

Foi o que fez David Beazley em seu projeto *Curio* [fpy.li/21-49]. Beazley estava interessado em repensar como estes recursos da linguagem poderiam ser usados em um framework desenvolvido do zero, sem carregar uma bagagem do passado. Lembre-se de que o `asyncio` foi lançado no Python 3.4, quando não existiam as instruções `async`, e em vez de `await` usávamos `yield from`. Portanto, a API original do `asyncio` não podia oferecer gerenciadores de contexto assíncronos, iteradores assíncronos e tudo o mais que as palavras-chave `async/await` tornaram possível. O resultado é que o *Curio* tem uma API mais elegante e uma implementação mais simples quando comparado ao `asyncio`.



O *Curio* é uma prova de conceito, e David Beazley não está mais trabalhando no projeto. Sua influência mais marcante está no framework *Trio* [fpy.li/21-58], que continua evoluindo e tem mais suporte de bibliotecas.

Se você estiver usando Python 3.12 ou superior, precisará instalar um *fork* atualizado publicado no PyPI como *curio-compat* [fpy.li/c6].

O Exemplo 21 mostra o script *blogdom.py* (Exemplo 1) reescrito para usar o *Curio*.

Exemplo 21. *blogdom.py*: Exemplo 1, agora usando o *Curio*

```
from keyword import kwlist
import curio

MAX_KEYWORD_LEN = 4

async def probe(domain: str) -> tuple[str, bool]: ①
    try:
        await curio.socket.getaddrinfo(domain, None) ②
    except curio.socket.gaierror:
        return (domain, False)
    return (domain, True)

async def main() -> None:
    names = (kw for kw in kwlist if len(kw) <= MAX_KEYWORD_LEN)
    domains = (f'{name}.dev'.lower() for name in names)
    async with curio.TaskGroup() as group: ③
        for domain in domains:
            await group.spawn(probe, domain) ④
        async for task in group: ⑤
            domain, found = task.result
            mark = '+' if found else ' '
            print(f'{mark} {domain}')

if __name__ == '__main__':
    curio.run(main()) ⑥
```

- ① `probe` não precisa obter o laço de eventos, porque...
- ② `...getaddrinfo` é uma função de `curio.socket`, não um método de um objeto `loop`—como no `asyncio`.
- ③ Um `TaskGroup` é um conceito central no *Curio*, para monitorar e controlar várias corrotinas, garantindo que elas todas sejam acionadas e encerradas, sem deixar alguma para trás.
- ④ `TaskGroup.spawn` é como você inicia uma corrotina, gerenciada por uma instância específica de `TaskGroup`. A corrotina é embrulhada em uma `Task`.
- ⑤ Iterar com `async for` sobre um `TaskGroup` produz instâncias de `Task` à medida que cada uma termina. Isto corresponde à linha em Exemplo 1 que usa `for ... in as_completed(...)`:

- ⑥ O *Curio* foi pioneiro no uso dessa maneira simples de iniciar um programa assíncrono em Python.

Para ilustrar este último ponto: lendo os exemplos de código de `asyncio` na primeira edição do *Python Fluente*, verá linhas como estas repetidas várias vezes:

```
laço = asyncio.get_event_loop()
loop.run_until_complete(main())
loop.close()
```

21.11.1. Concorrência estruturada

Um `TaskGroup` do *Curio* é um gerenciador de contexto assíncrono que substitui várias APIs e padrões de codificação repetitivos do `asyncio`. Acabamos de ver como iterar sobre um `TaskGroup` torna a função `asyncio.as_completed(...)` desnecessária.

Outro exemplo: em vez da função especial `gather`, este trecho da documentação de "Task Groups" [fpy.li/21-50] coleta os resultados de todas as tarefas no grupo:

```
async with TaskGroup(wait=all) as g:
    await g.spawn(coro1)
    await g.spawn(coro2)
    await g.spawn(coro3)
print('Results:', g.results)
```

Objetos `TaskGroup` suportam «concorrência estruturada» [fpy.li/21-51]: uma disciplina de programação concorrente que organiza todas as atividades de um grupo de tarefas assíncronas em um bloco de código com apenas um ponto de entrada e uma saída. Isto é análogo à programação estruturada, que introduziu instruções de bloco para limitar os pontos de entrada e saída de condicionais, laços e sub-rotinas, e eliminou a instrução `GOTO` que permitia desviar a execução direto para qualquer outra linha do código. Quando usado como um gerenciador de contexto assíncrono, um `TaskGroup` garante que na saída do bloco, todas as tarefas criadas dentro dele estão finalizadas ou canceladas e qualquer exceção foi levantada.



A concorrência estruturada está sendo adotada pelo `asyncio`. Uma evidência é a *PEP 654—Exception Groups and except** [fpy.li/pep654], que foi aprovada para o Python 3.11. A seção *Motivation* [fpy.li/21-53] menciona as *nurseries* (creches) do *Trio*, que correspondem aos *TaskGroup* do *Curio*: "Implementar uma API de acionamento de tarefas melhor no `asyncio`, inspirada pelas *nurseries* do *Trio*, foi a principal motivação desta PEP."

Outra inovação importante do *Curio* é um suporte melhor para programar com corrotinas e threads na mesma base de código—uma necessidade de qualquer programa assíncrono não-trivial. Iniciar uma thread com `await spawn_thread(func, ...)` devolve um objeto `AsyncThread` com uma interface de `Task`. As threads podem chamar corrotinas, graças à função especial `AWAIT` (coro) [fpy.li/21-54]—escrita inteiramente com maiúsculas porque `await` agora é uma palavra-chave.

O *Curio* também oferece uma `UniversalQueue` que pode ser usada para coordenar o trabalho entre threads, corrotinas *Curio* e corrotinas `asyncio`. Sim, o *Curio* pode ser executado em uma thread ao lado do `asyncio` em outra thread, no mesmo processo, comunicando-se através de `UniversalQueue` e de `UniversalEvent`. A API destas classes "universais" é a mesma dentro e fora de corrotinas, mas em uma corrotina é preciso acionar os métodos com `await`.

Em outubro de 2021, quando estou escrevendo esse capítulo, a *HTTPX* é a primeira biblioteca HTTP cliente compatível com o *Curio* [fpy.li/21-55], mas não sei de nenhuma biblioteca assíncrona de banco de dados que o suporte nesse momento. No repositório do *Curio* há um conjunto impressionante de «exemplos de programação para rede» [fpy.li/21-56], incluindo um que utiliza *WebSocket*, e outro implementando o algoritmo concorrente *RFC 8305—Happy Eyeballs* [fpy.li/21-57], para conexão com pontos de acesso IPv6 revertendo rapidamente para IPv4 quando necessário.

O design do *Curio* foi muito influente. o framework *Trio* [fpy.li/21-58], iniciado por Nathaniel J. Smith, foi muito inspirado nele. O *Curio* pode também ter estimulado os contribuidores de Python a melhorar a usabilidade da API do `asyncio`. Por exemplo, em suas primeiras versões, os usuários do `asyncio` muitas vezes eram obrigados a obter e ficar passando um objeto `loop`, porque algumas funções essenciais eram métodos de `loop`, ou precisavam do laço de eventos como um

argumento. Em versões mais recentes de Python, acesso direto ao laço não é mais tão necessário e, várias funções que aceitavam um loop opcional estão agora descontinuando aquele argumento.

Anotações de tipo para tipos assíncronos é o nosso próximo tópico.

21.12. Dicas de tipo para objetos assíncronos

O tipo devolvido por uma corrotina nativa é o tipo do objeto que você obtém quando usa `await` naquela corrotina, que é o tipo do objeto devolvido pela instrução `return` no corpo da corrotina nativa. Isto é mais simples que as anotações de corrotinas clássicas, discutidas na Seção 17.13.3.

Neste capítulo vimos vários exemplos de corrotinas nativas anotadas, incluindo a probe do Exemplo 21:

```
async def probe(domain: str) -> tuple[str, bool]:
    try:
        await socket.getaddrinfo(domain, None)
    except socket.gaierror:
        return (domain, False)
    return (domain, True)
```

Se você precisar anotar um parâmetro que recebe um objeto corrotina como argumento, então o tipo genérico é:

```
class typing.Coroutine(Awaitable[V_co], Generic[T_co, T_contra, V_co]):
    ...
```

Aquele tipo e os tipos seguintes foram introduzidos no Python 3.5 e 3.6 para anotar objetos assíncronos:

```
class typing.AsyncContextManager(Generic[T_co]):
    ...
class typing.AsyncIterable(Generic[T_co]):
    ...
class typing.AsyncIterator(AsyncIterable[T_co]):
    ...
```

```

...
class typing.AsyncGenerator(AsyncIterator[T_co],
                           Generic[T_co, T_contra]):
    ...
class typing.Awaitable(Generic[T_co]):
    ...

```

Com Python ≥ 3.9, use os equivalentes definidos em `collections.abc`.

Quero destacar três aspectos destes tipos genéricos.

Primeiro: eles são todos covariantes no primeiro parâmetro de tipo, que é o tipo dos itens produzidos a partir destes objetos. Lembre-se da regra #1 da «Seção 15.7.4.4» [fpy.li/cj] (vol.2):

Se um parâmetro de tipo formal define um tipo para um dado que sai do objeto, ele pode ser covariante.

Segundo: `AsyncGenerator` e `Coroutine` são contra-variantes no penúltimo parâmetro. Aquele é o tipo do argumento passado ao método de baixo nível `.send()`, que o laço de eventos invoca para acionar geradores assíncronos e corrotinas. Desta forma, é um tipo de "entrada", então vale a regra #2 da variância:

Se um parâmetro de tipo formal define um tipo para um dado que entra no objeto após sua construção inicial, ele pode ser contravariante.

Terceiro: `AsyncGenerator` não tem tipo de retorno, ao contrário de `typing.Generator`, usado para anotar corrotinas clássicas, apesar do nome que sugere outra coisa, como vimos na Seção 17.13.3. Devolver um valor levantando `StopIteration(value)` foi a gambiarra que permitiu a geradores funcionarem como corrotinas clássicas suportando `yield from`, como vimos na Seção 17.13. Não há tal sobreposição entre os objetos assíncronos: objetos `AsyncGenerator` produzem itens mas não devolvem um resultado final, e são completamente separados de objetos corrotina que devolvem um resultado, mas não usam `yield`.

Por fim, vamos discutir rapidamente as vantagens e desafios da programação assíncrona.

21.13. Como a programação assíncrona funciona e como não funciona

As seções finais deste capítulo discutem ideias de alto nível sobre programação assíncrona, independente da linguagem ou da biblioteca usadas.

Vamos começar explicando por que a programação assíncrona é útil, seguido por um mito popular e como lidar com ele.

21.13.1. Correndo em círculos em torno de chamadas bloqueantes

Ryan Dahl, o inventor do Node.js, introduz a filosofia do projeto dizendo "Estamos fazendo E/S de forma totalmente errada."^[16] Ele define uma "função bloqueante" como uma função que faz E/S de arquivo ou rede, e argumenta que elas não podem ser tratadas da mesma forma que tratamos funções não-bloqueantes. Para explicar a razão disso, ele apresenta os números na segunda coluna da Tabela 8.

Tabela 8. Latência em computadores modernos para ler dados em diferentes dispositivos. A terceira coluna mostra os tempos proporcionais em uma escala fácil de entender para nós, humanos vagarosos.

Dispositivo	Ciclos de CPU	Escala proporcional "humana"
cache L1	3	3 segundos
cache L2	14	14 segundos
RAM	250	250 segundos
HD local	41.000.000	1,3 anos
rede	240.000.000	7,6 anos

Para entender a Tabela 8, tenha em mente que as CPUs modernas, com seus *clocks* em frequências na casa dos GHz, rodam bilhões de ciclos por segundo. Suponha que uma CPU rode exatamente 1 bilhão de ciclos por segundo. Tal CPU pode realizar mais de 333 milhões de leituras do cache L1 em 1 segundo, ou 4 (quatro!) leituras da rede no mesmo segundo. A terceira coluna da Tabela 8 coloca os números em perspectiva, multiplicando a segunda coluna por um fator constante. Então, em um universo alternativo, se uma leitura da RAM demorasse

250 segundos, uma leitura da rede demoraria 7,6 anos! A diferença quantitativa é tão grande que se torna uma diferença qualitativa importante: esperar 250 segundos é muito diferente de esperar 7,6 anos!

A Tabela 8 explica por que uma abordagem disciplinada da programação assíncrona pode levar a servidores de alto desempenho. O desafio é alcançar esta disciplina. O primeiro passo é reconhecer que um sistema limitado apenas por E/S é uma fantasia.

21.13.2. O mito dos sistemas limitados por E/S

Um meme exaustivamente repetido é que programação assíncrona é boa para *I/O bound systems* (sistemas limitados por E/S), ou seja, sistemas onde o gargalo é a entrada e saída de dados, e não o processamento de dados na CPU. Aprendi da forma mais difícil que não existem "sistemas limitados por E/S." Você pode ter *funções* limitadas por E/S. Talvez a maioria das funções no seu sistema sejam limitadas por E/S; isto é, elas passam mais tempo esperando por E/S do que realizando operações na CPU e na memória. Enquanto esperam, cedem o controle para o laço de eventos, que pode então acionar outras tarefas pendentes. Mas, inevitavelmente, qualquer sistema não-trivial terá partes limitadas pela CPU. Até mesmo sistemas triviais revelam isso, sob stress. Na caixa *Ponto de vista* ao final deste capítulo escrevi sobre dois programas assíncronos sofrendo com funções limitadas pela CPU que atrasavam o laço de eventos, com severos impactos no desempenho do sistema como um todo.

Dado que qualquer sistema não-trivial terá funções limitadas pela CPU, lidar com elas é a chave do sucesso na programação assíncrona.

21.13.3. Evitando as armadilhas do uso da CPU

Se você está usando Python em larga escala, precisa ter testes automatizados especificamente para detectar regressões de desempenho assim que elas acontecem. Isso é de importância crítica com código assíncrono, mas é relevante também para código Python baseado em threads—por causa da GIL. Se você esperar até a lentidão começar a incomodar a equipe de desenvolvimento, será tarde demais. A solução poderá exigir mudanças drásticas.

Aqui estão algumas opções para quando você identifica gargalos de uso da CPU:

- Delegar a tarefa para um banco de processos Python.
- Delegar a tarefa para uma fila de tarefas externa.
- Reescrever o código relevante em Cython, C, Rust ou alguma outra linguagem que compile para código de máquina e faça interface com a API Python/C, de preferência liberando a GIL.
- Decidir que pode tolerar a perda de desempenho e deixar como está—mas registre essa decisão, para ficar mais fácil revertê-la no futuro.

A fila de tarefas externa deveria ser escolhida e integrada o mais rápido possível, no início do projeto, para que ninguém na equipe hesite em usá-la quando necessário.

A opção de deixar como está entra na conta de «dívida tecnológica» [fpy.li/b8].

Programação concorrente é um tópico fascinante, e eu gostaria de escrever mais. Mas não é o foco principal deste livro, e este já é um dos capítulos mais longos, então vamos encerrar por aqui.

21.14. Resumo do capítulo

O problema com as abordagens usuais da programação assíncrona é que elas são propostas do tipo "tudo ou nada". Ou você reescreve todo o código, de forma que nada nele bloqueie, ou você está só perdendo tempo.

— Alvaro Videla e Jason J. W. Williams, *RabbitMQ in Action*

Escolhi esta epígrafe para este capítulo por duas razões. Em um nível mais alto, ela nos lembra de evitar o bloqueio do laço de eventos, delegando tarefas lentas para outra unidade de processamento, desde uma thread ou processo local, até uma fila de tarefas distribuída. Em um nível mais baixo, ela também é um aviso: no momento em que você escreve seu primeiro `async def`, seu programa vai inevitavelmente ver surgir mais e mais `async def`, `await`, `async with`, e `async for`. E o uso de bibliotecas não-assíncronas de repente pode complicar o seu trabalho.

Após os exemplos simples com o *spinner* no Capítulo 19, aqui nosso maior foco foi a programação assíncrona com corrotinas nativas, começando com o exemplo de sondagem de DNS *blogdom.py*, seguido pelo conceito de esperável. No código-fonte de *flags_asyncio.py* encontramos o primeiro exemplo de um gerenciador de contexto assíncrono: `httpx.AsyncClient`.

As variantes mais avançadas do programa de download de bandeiras apresentaram duas funções poderosas: o gerador `asyncio.as_completed` e a corrotina `loop.run_in_executor` para delegar tarefas para threads ou processos. Também vimos o conceito e a aplicação de um semáforo, para limitar o número de downloads concorrentes—como se espera de um cliente HTTP bem comportado.

A programação assíncrona para servidores foi apresentada com os exemplos *mojifinder*: um serviço Web usando a *FastAPI* e o *tcp_mojifinder.py*—este último utilizando apenas o protocolo TCP e o `asyncio`.

A seguir, iteração assíncrona e iteráveis assíncronos foram o principal tópico, com seções sobre `async for`, o console assíncrono de Python, geradores assíncronos, expressões geradoras assíncronas, e compreensões assíncronas.

O último exemplo do capítulo foi o *blogdom.py* reescrito com o framework *Curio*, demonstrando como os recursos de programação assíncrona de Python não estão presos ao pacote `asyncio`. O *Curio* também demonstra o conceito de *concorrência estruturada*, que poderá ter um grande impacto muitas linguagens de programação, trazendo mais clareza para o código concorrente.

Por fim, a Seção 21.13 apresentou o principal atrativo da programação assíncrona: não perder tempo esperando por E/S. Também vimos que não existem sistemas limitados só por E/S, e como lidar com as inevitáveis partes do seu programa assíncrono que utilizam intensivamente a CPU.

21.15. Para saber mais

A palestra de David Beazley na abertura da PyOhio 2016, *Fear and Awaiting in Async* [fpy.li/21-61] (Medo e espera em [programação] assíncrona) é uma introdução incrível com "código ao vivo" demonstrando os recursos da linguagem viabilizados pela contribuição de Yury Selivanov ao Python 3.5: as palavras-chave `async/await`. Em certo momento, Beazley reclama que `await` não

pode ser usada em compreensões de lista, mas isso foi resolvido por Selivanov na *PEP 530—Asynchronous Comprehensions* [fpy.li/pep530], implementada mais tarde naquele mesmo ano, no Python 3.6.

Fora isso, todo o resto da palestra de Beazley é atemporal, pois ele revela como os objetos assíncronos vistos neste capítulo funcionam, sem ajuda de qualquer framework—com uma simples função `run` que invoca `.send(None)` para acionar corrotinas. Apenas no final Beazley mostra o *Curio* [fpy.li/21-62], que ele havia começado a desenvolver naquele ano, como uma prova de conceito, para ver o quão longe seria possível levar a programação assíncrona usando apenas corrotinas, sem callbacks ou *futures*. Como vimos, dá para ir muito longe—como demonstra o *Curio* e o desenvolvimento posterior do *Trio* [fpy.li/21-58] por Nathaniel J. Smith. A documentação do *Curio* contém «links» [fpy.li/21-64] para outras palestras de Beazley sobre o assunto.

Além de criar o *Trio*, Nathaniel J. Smith escreveu dois artigos muito profundos, que eu recomendo: *Some thoughts on asynchronous API design in a post-async/await world* [fpy.li/21-65] (Algumas reflexões sobre o design de APIs assíncronas em um mundo pós-async/await), comparando os designs do *Curio* e do *asyncio*, e *Notes on structured concurrency, or: go statement considered harmful* [fpy.li/21-66] (Notas sobre concorrência estruturada, ou: a instrução `go` considerada nociva), sobre concorrência estruturada. Smith também deu uma longa e informativa resposta à questão: *_What is the core difference between asyncio and Trio?* [fpy.li/21-67] (Qual é a principal diferença entre *asyncio* e *Trio*?) no *StackOverflow*.

Para aprender mais sobre o pacote *asyncio*, já mencionei os melhores recursos que conheço no início do capítulo: a «documentação oficial» [fpy.li/b9], após a «profunda reorganização» [fpy.li/21-69] iniciada por Yury Selivanov em 2018, e o livro de Caleb Hattingh, *Using Asyncio in Python* [fpy.li/hattingh] (O'Reilly). Na documentação oficial, não deixe de ler «Desenvolvendo com *asyncio*» [fpy.li/ba], que documenta o modo de depuração do *asyncio* e também discute erros e armadilhas comuns, e como evitá-los.

Para uma introdução de 30 minutos, muito acessível, à programação assíncrona em geral e também ao *asyncio*, assista a palestra *Asynchronous Python for the Complete Beginner* [fpy.li/21-71] (Python Assíncrono para o Iniciante Total), de Miguel Grinberg, apresentada na PyCon 2017. Outra ótima introdução é *Demystifying Python's Async and Await Keywords* [fpy.li/21-72] (Desmistificando as

Palavras-Chave Async e Await de Python), apresentada por Michael Kennedy, onde aprendi sobre a biblioteca *unsync* [fpy.li/21-73], que fornece um decorador para delegar a execução de corrotinas, funções dedicadas a E/S e funções de uso intensivo de CPU para *asyncio*, *threading*, ou *multiprocessing*, conforme a necessidade.

Na EuroPython 2019, Lynn Root—uma das líderes mundiais das *PyLadies* [fpy.li/21-74]—apresentou a excelente *Advanced asyncio: Solving Real-world Production Problems* [fpy.li/21-75] (*Asyncio Avançado: Resolvendo Problemas de Produção do Mundo Real*), a partir de sua experiência usando Python como engenheira no Spotify.

Em 2020, Łukasz Langa gravou uma ótima série de vídeos sobre o *asyncio*, começando com *Learn Python's AsyncIO #1—The Async Ecosystem* [fpy.li/21-76] (Aprenda o AsyncIO de Python—O Ecossistema Async). Langa também fez um vídeo muito bacana, *AsyncIO + Music* [fpy.li/21-77], para a PyCon 2020, que mostra o *asyncio* aplicado a um domínio orientado a eventos muito concreto, e também explica esta aplicação do início ao fim.

Outra área dominada por programação orientada a eventos são os sistemas embarcados. Por isso Damien George adicionou o suporte a *async/await* em seu interpretador *MicroPython* [fpy.li/21-78] para microcontroladores. Na PyCon Australia 2018, Matt Trentini demonstrou a biblioteca *uasyncio* [fpy.li/21-79], um subconjunto de *asyncio* que é parte da biblioteca padrão do *MicroPython*.

Para uma visão de mais alto nível sobre a programação assíncrona em Python, leia o post *Python async frameworks—Beyond developer tribalism* [fpy.li/21-80] (Frameworks assíncronos de Python—para além do tribalismo dos desenvolvedores), de Tom Christie.

Por fim, recomendo *What Color Is Your Function?* [fpy.li/21-81] (Qual a Cor da Sua Função?) de Bob Nystrom, discutindo os modelos de execução incompatíveis entre funções comuns e funções assíncronas—que chamamos de corrotinas—em JavaScript, Python, C# e outras linguagens. Alerta de spoiler: a conclusão de Nystrom é que a linguagem que acertou nessa área foi Go, onde todas as funções têm a mesma cor. Gosto disso no Go. Mas também acho que Nathaniel J. Smith tem razão quando escreveu *Go statement considered harmful* [fpy.li/21-66] (Instrução *go* considerada nociva). Nada é perfeito, e programação concorrente é sempre difícil.

Ponto de vista

Como uma função lerda quase estragou as benchmarks do *uvloop*

Em 2016, Yury Selivanov lançou o *uvloop* [fpy.li/21-83], "um substituto rápido e direto para o laço de eventos embutido do *asyncio*." Os *benchmarks* (números de desempenho) apresentados no «post» [fpy.li/21-84] de Selivanov anunciando a biblioteca, em 2016, eram muito impressionantes: "ela é pelo menos 2x mais rápida que o *nodejs* e *gevent*, bem como qualquer outro framework assíncrono de Python. O desempenho do *asyncio* com o *uvloop* é próximo ao de programas em Go."

Entretanto, o post revela que a *uvloop* é capaz de competir com o desempenho do Go sob duas condições:

1. Que o Go seja configurado para usar uma única thread. Isso faz o runtime do Go se comportar de forma similar ao *asyncio*: a concorrência é alcançada por múltiplas corrotinas acionadas por um laço de eventos, tudo na mesma thread.^[17]
2. Que o código Python use a biblioteca *httptools* [fpy.li/21-85] além do próprio *uvloop*.

Selivanov explica que escreveu *httptools* após testar o desempenho do *uvloop* com a *aiohttp* [fpy.li/21-86]—uma das primeiras bibliotecas HTTP completas construídas sobre o *asyncio*:

Entretanto, o gargalo de desempenho no aiohttp estava em seu parser de HTTP, que era tão lento que pouco importava a velocidade da biblioteca de E/S subjacente. Para tornar as coisas mais interessantes, criamos uma biblioteca para Python usar a http-parser (a biblioteca em C do parser do Node.js, originalmente desenvolvida para o NGINX). A biblioteca é chamada httptools, e está disponível no Github e no PyPI.

Agora reflita sobre isso: os testes de desempenho HTTP de Selivanov consistiam de um simples servidor eco escrito em diferentes linguagens e usando diferentes bibliotecas, testados pela ferramenta de benchmarking *wrk* [fpy.li/21-87]. A maioria dos desenvolvedores consideraria um simples servidor eco um "sistema limitado por E/S", certo?

Mas no caso, a análise de cabeçalhos HTTP é intensiva em CPU, e tinha uma implementação lenta, em Python, na biblioteca *aiohttp* quando Selivanov realizou os testes em 2016. Sempre que uma função escrita em Python estava processando os cabeçalhos, o laço de eventos era bloqueado. O impacto foi tão significativo que Selivanov se deu ao trabalho extra de escrever o *httptools*. Sem a otimização do código que usa intensivamente a CPU, os ganhos de desempenho de um laço de eventos mais rápido eram perdidos.

Morte lenta

Em vez de um simples servidor eco, imagine um sistema Python complexo e em evolução, com milhares de linhas de código assíncrono, e conectado a muitas bibliotecas externas.

Anos atrás me pediram para ajudar a diagnosticar problemas de desempenho em um sistema assim. Ele era escrito em Python 2.7, com o framework *Twisted* [fpy.li/21-88]—uma biblioteca sólida, de alto desempenho, precursora do próprio *asyncio*.

Python era usado para construir uma fachada para a interface Web, integrando funcionalidades fornecidas por bibliotecas pré-existentes e ferramentas de linha de comando escritas em outras linguagens—mas não projetadas para execução concorrente.

O projeto era ambicioso: já estava em desenvolvimento há mais de um ano, mas ainda não estava em produção.^[18] Com o passar do tempo, os desenvolvedores notaram que o desempenho do sistema estava piorando, e o time não conseguia localizar os principais gargalos.

O que estava acontecendo: cada nova funcionalidade introduzia mais código intensivo em CPU, atrasando o laço de eventos do *Twisted*. O papel de Python como uma linguagem de integração entre processos externos implicava em muita interpretação de dados, serialização, desserialização, e conversões entre formatos. Não havia um gargalo único: o problema estava espalhado por incontáveis pequenas funções criadas ao longo de meses de desenvolvimento.

A solução seria repensar a arquitetura do sistema, reescrever muito código, usar uma fila de tarefas, e talvez criar microserviços ou bibliotecas customizadas, escritas em linguagens mais eficientes no processamento concorrente intensivo em CPU (eu sugeri Go para esta finalidade). Os gestores não quiseram fazer aquele investimento adicional, e o projeto foi cancelado semanas depois deste diagnóstico.

Quando contei essa história para Glyph Lefkowitz—fundador do projeto *Twisted*—ele falou que é prioritário decidir quais ferramentas serão usadas para executar tarefas intensivas em CPU sem atrapalhar o laço de eventos, logo no início de qualquer projeto envolvendo programação assíncrona. Esta conversa com Glyph foi a inspiração para a Seção 21.13.3.

[1] Videla & Williams, *RabbitMQ in Action (Manning, 2012)*, *Solving Problems with Rabbit: coding and patterns*, p. 61

[2] Selivanov implementou `async/await` no Python, e escreveu as PEPs relacionadas: 492 [fpy.li/pep492], 525 [fpy.li/pep525], e 530 [fpy.li/pep530].

[3] Há uma exceção a essa regra: se você iniciar Python com a opção `-m asyncio`, pode então usar `await` diretamente no prompt `>>>` para controlar uma corrotina nativa. Isto é explicado na Seção 21.10.1.1.

[4] `true.dev` está disponível por US\$ 360,00 ao ano no momento em que escrevo esta nota. Também vi que `for.dev` está registrado, mas seu DNS não está configurado.

[5] Agradeço ao leitor Samuel Woodward por reportar este erro para a O'Reilly em fevereiro de 2023

[6] Agradeço a Guto Maia, que notou que o conceito de semáforo não era explicado quando leu o primeiro rascunho deste capítulo.

[7] Iniciei uma discussão sobre esta questão no grupo `python-tulip`, intitulada *Which other futures may come out of asyncio.as_completed?* [fpy.li/21-19] (Que outros futures podem sair de `asyncio.as_completed`?). Guido e Victor Stinner e fornece detalhes sobre a implementação de `as_completed`, bem como a relação próxima entre *futures* e corrotinas no `asyncio`.

[8] O ponto de interrogação encaixotado na captura de tela não é um defeito do livro ou do ebook que você está lendo. É o caractere U+101EC—PHAISTOS DISC SIGN CAT, que não existe na fonte do terminal que usei. Ele vem do Disco de Festo [fpy.li/ay], um artefato antigo inscrito com pictogramas, descoberto na ilha de Creta.

[9] Você pode usar outro servidor ASGI no lugar do *uvicorn*, como o *hypercorn* ou o *Daphne*. Veja na documentação oficial do ASGI a «página sobre implementações» [fpy.li/21-30] para mais informações.

[10] Agradeço ao revisor técnico Miroslav Šedivý por apontar bons lugares para usar `pathlib` nos exemplos de código.

[11] O bug #5535 está resolvido desde outubro de 2021, mas o Mypy não lançou uma nova versão até o fechamento desta edição, então o erro permanece.

[12] O revisor técnico Leonardo Rochael apontou que a construção do índice poderia ser delegada a outra thread, usando `loop.run_with_executor()` na corrotina `supervisor`. Dessa forma o servidor

estaria pronto para receber requisições imediatamente, enquanto o índice é construído. Isso é verdade, mas como consultar o índice é a única coisa que esse servidor faz, isso não seria uma grande vantagem nesse exemplo.

[13] Isso é ótimo para experimentação, como o console do Node.js. Agradeço a Yury Selivanov por mais essa excelente contribuição para Python assíncrono.

[14] Veja *RFC 6761—Special-Use Domain Names* [fpy.li/21-45].

[15] Em contraste com o JavaScript, onde `async/await` são atrelados ao laço de eventos que é inseparável do ambiente de runtime, isto é, um navegador, o Node.js ou o Deno.

[16] Vídeo: *Introduction to Node.js* [fpy.li/21-59], em 4:55.

[17] Usar uma única thread era o default até o lançamento do Go 1.5. Anos antes, o Go já tinha ganho uma merecida reputação por permitir a criação de sistemas em rede de alta concorrência. Mais uma evidência de que a concorrência não exige múltiplas threads ou múltiplos núcleos de CPU.

[18] Independente de escolhas técnicas, esse foi talvez o maior erro daquele projeto: as partes interessadas não forçaram uma abordagem MVP—entregar o "Mínimo Produto Viável" o mais rápido possível e acrescentar novos recursos em um ritmo estável.

Capítulo 22. Atributos dinâmicos e propriedades

As propriedades são muito importantes porque tornam perfeitamente seguro, e até aconselhável, expor publicamente atributos de dados como parte da interface pública de sua classe.^[1]

— Martelli, Ravenscroft & Holden, Why properties are important (Porque propriedades são importantes)

No Python, atributos de dados (ou campos) e métodos são conhecidos conjuntamente como *atributos*. Um método é um atributo invocável. Atributos dinâmicos oferecem a mesma interface que os atributos de dados—isto é, `obj. atrib`—mas são computados sob demanda. Isso atende ao *Princípio de Acesso Uniforme* de Bertrand Meyer:

Todos os serviços oferecidos por um módulo devem estar disponíveis através de uma notação uniforme, que não revele se eles são implementados por armazenamento ou por computação.^[2]

— Bertrand Meyer, Object-Oriented Software Construction (Construção de Software Orientada a Objetos)

Há várias formas de implementar atributos dinâmicos em Python. Este capítulo trata das mais simples: o decorador `@property` e o método especial `__getattr__`.

Uma classe definida pelo usuário pode implementar `__getattr__` para oferecer uma variação de atributos dinâmicos que chamo de *atributos virtuais*: atributos que não são declarados explicitamente em lugar algum no código-fonte da classe, e que não estão presentes no `__dict__` das instâncias, mas que podem ser obtidos de algum outro lugar ou calculados sob demanda sempre que um usuário tenta ler um atributo inexistente tal como `obj.ausente`.

Programar atributos dinâmicos e virtuais é o tipo de metaprogramação que autores de frameworks fazem. Entretanto, como as técnicas básicas no Python são simples, podemos usá-las em tarefas cotidianas de processamento de dados. É por aí que iniciaremos esse capítulo.

22.1. Novidades neste capítulo

As atualizações deste capítulo foram motivadas pela discussão relativa a `@functools.cached_property` (introduzido no Python 3.8), o uso combinado de `@property` e `@functools.cache` (novo no 3.9). Isto afetou o código das classes `Record` e `Event`, que aparecem na Seção 22.3. Também fiz uma refatoração para aproveitar a otimização da *PEP 412—Key-Sharing Dictionary* [fpy.li/pep412] (Dicionário com chaves compartilhadas).

Para enfatizar as características mais relevantes, e ao mesmo tempo manter os exemplos legíveis, removi algum código não-essencial—fundindo a antiga classe `DbRecord` com `Record`, substituindo `shelve.Shelve` por um `dict`, e suprimindo a lógica para baixar o conjunto de dados da OSCON—que os exemplos agora carregam de um arquivo local, disponível no «repositório de código» [fpy.li/code] do *Python Fluente*.

22.2. Explorando dados com atributos dinâmicos

Nos próximos exemplos, vamos nos valer dos atributos dinâmicos para trabalhar com um conjunto de dados JSON publicado pela O'Reilly, para a conferência OSCON 2014. O Exemplo 1 mostra quatro registros daquele conjunto de dados.^[3]

Exemplo 1. Amostra de registros do `osconfeed.json`, com alguns dados abreviados.

```
{ "Schedule":
  { "conferences": [{"serial": 115 }],
    "events": [
      { "serial": 34505,
        "name": "Why Schools Don't Use Open Source to Teach Programming",
        "event_type": "40-minute conference session",
        "time_start": "2014-07-23 11:30:00",
        "time_stop": "2014-07-23 12:10:00",
        "venue_serial": 1462,
        "description": "Aside from the fact that high school
programming...",
        "website_url": "http://oscon.com/2014/sched/34505",
        "speakers": [157509],
        "categories": ["Education"] }
    ],
```

```

    "speakers": [
      { "serial": 157509,
        "name": "Robert Lefkowitz",
        "photo": null,
        "url": "http://sharewave.com/",
        "position": "CTO",
        "affiliation": "Sharewave",
        "twitter": "sharewaveteam",
        "bio": "Robert 'r0ml' Lefkowitz is the CTO at Sharewave..." }
    ],
    "venues": [
      { "serial": 1462,
        "name": "F151",
        "category": "Conference Venues" }
    ]
  }
}

```

O Exemplo 1 mostra 4 dos 895 registros do arquivo JSON. O conjunto completo total é um único objeto JSON, com a chave "Schedule" (cronograma), e seu valor é outro mapeamento com quatro chaves: "conferences" (conferências), "events" (eventos), "speakers" (palestrantes), e "venues" (locais). Cada uma destas quatro chaves aponta para uma lista de registros. No conjunto de dados completo, as listas de "events", "speakers" e "venues" contêm dezenas ou centenas de registros, mas "conferences" contém apenas aquele único registro exibido na segunda linha do Exemplo 1. Cada registro inclui um campo "serial", que é um identificador único do registro dentro da lista onde ele está.

Usei o console de Python para explorar os dados interativamente, como mostra o Exemplo 2.

Exemplo 2. Exploração interativa do osconfeed.json

```

>>> import json
>>> with open('data/osconfeed.json') as fp:
...     feed = json.load(fp) ①
>>> sorted(feed['Schedule'].keys()) ②
['conferences', 'events', 'speakers', 'venues']
>>> for key, value in sorted(feed['Schedule'].items()):
...     print(f'{len(value):3} {key}') ③
...

```

```

1 conferences
484 events
357 speakers
53 venues
>>> feed['Schedule']['speakers'][-1]['name'] ④
'Carina C. Zona'
>>> feed['Schedule']['speakers'][-1]['serial'] ⑤
141590
>>> feed['Schedule']['events'][40]['name']
'There *Will* Be Bugs'
>>> feed['Schedule']['events'][40]['speakers'] ⑥
[3471, 5199]

```

- ① feed é um dict contendo dicts e listas aninhados, com valores string e inteiros.
- ② Lista as quatro coleções de registros dentro de 'Schedule'.
- ③ Exibe a contagem de registros para cada coleção.
- ④ Navega pelos dicts e listas aninhados para obter o nome da última palestrante (speaker).
- ⑤ Obtém o número de série daquela palestrante.
- ⑥ Cada evento tem uma lista 'speakers', com o número de série de zero ou mais palestrantes.

22.2.1. Explorando dados JSON e similares com atributos dinâmicos

O Exemplo 2 é simples, mas esta sintaxe é inconveniente:

```
feed['Schedule']['events'][40]['name']
```

Em JavaScript, é possível obter o mesmo valor escrevendo `feed.Schedule.events[40].name`. Não é difícil implementar uma classe parecida com um dict para fazer o mesmo em Python—há inúmeras implementações na Web.^[4] Escrevi FrozenJSON, que é mais simples que a maioria das soluções porque só permite leitura: ela serve apenas para explorar os dados. FrozenJSON é recursiva, lidando automaticamente com mapeamentos e listas aninhados.

O Exemplo 3 demonstra FrozenJSON; código-fonte no Exemplo 4.

Exemplo 3. FrozenJSON, do Exemplo 4, permite ler atributos como `talk.name`, e invocar métodos como `feed.keys()` e `feed.items()`

```
>>> import json
>>> raw_feed = json.load(open('data/osconfeed.json'))
>>> feed = FrozenJSON(raw_feed) ①
>>> len(feed.Schedule.speakers) ②
357
>>> feed.keys()
dict_keys(['Schedule'])
>>> sorted(feed.Schedule.keys()) ③
['conferences', 'events', 'speakers', 'venues']
>>> for key, value in sorted(feed.Schedule.items()): ④
...     print(f'{len(value):3} {key}')
...
    1 conferences
484 events
357 speakers
    53 venues
>>> feed.Schedule.speakers[-1].name ⑤
'Carina C. Zona'
>>> talk = feed.Schedule.events[40]
>>> type(talk) ⑥
<class 'explore0.FrozenJSON'>
>>> talk.name
'There *Will* Be Bugs'
>>> talk.speakers ⑦
[3471, 5199]
>>> talk.flavor ⑧
Traceback (most recent call last):
...
KeyError: 'flavor'
```

- ① Cria uma instância de FrozenJSON a partir de `raw_feed`, feito de dicts e listas aninhados.
- ② FrozenJSON permite percorrer dicts aninhados usando a notação de atributos; aqui exibimos o tamanho da lista de palestrantes.
- ③ Métodos dos dicts subjacentes também podem ser acessados; por exemplo, `.keys()`, para recuperar os nomes das coleções de registros.

- ④ Usando `items()`, podemos buscar os nomes das listas de registros e seus conteúdos, para exibir o `len()` de cada uma.
- ⑤ Uma `list`, tal como `feed.Schedule.speakers`, permanece uma lista, mas os itens dentro dela, se forem mapeamentos, são convertidos em um `FrozenJSON`.
- ⑥ O item 40 na lista `events` era um objeto `JSON`; agora ele é uma instância de `FrozenJSON`.
- ⑦ Registros de eventos têm uma lista de `speakers` com os números de série dos palestrantes.
- ⑧ Tentar ler um atributo inexistente gera uma exceção `KeyError`, em vez da `AttributeError` usual.

A pedra angular da classe `FrozenJSON` é o método `__getattr__`, que já usamos no exemplo `Vector` da «Seção 12.6» [fpy.li/cp] (vol.2), para recuperar componentes de `Vector` por letra: `v.x`, `v.y`, `v.z`, etc.

É importante lembrar que o método especial `__getattr__` só é invocado pelo interpretador quando o processo habitual não consegue recuperar um atributo (isto é, quando o atributo acessado não é encontrado na instância, nem na sua classe ou suas superclasses).

O passo ⑧ do Exemplo 3 expõe um pequeno problema em meu código: tentar ler um atributo ausente deveria produzir uma exceção `AttributeError`, e não a `KeyError` gerada. Quando implementei o tratamento de erro para fazer isso, o método `__getattr__` se tornou duas vezes mais longo, ofuscando a essência da lógica que eu queria apresentar. Dado que os usuários devem saber que uma `FrozenJSON` é criada a partir de mapeamentos e listas, levantar `KeyError` não é tão confuso assim.

Exemplo 4. `explore0.py`: transforma um conjunto de dados `JSON` em um `FrozenJSON` contendo objetos `FrozenJSON` aninhados, listas e tipos simples

```
from collections import abc

class FrozenJSON:
    """A read-only façade for navigating a JSON-like object
    using attribute notation
    """
```

```

def __init__(self, mapping):
    self.__data = dict(mapping) ①

def __getattr__(self, name): ②
    try:
        return getattr(self.__data, name) ③
    except AttributeError:
        return FrozenJSON.build(self.__data[name]) ④

def __dir__(self): ⑤
    return self.__data.keys()

@classmethod
def build(cls, obj): ⑥
    if isinstance(obj, abc.Mapping): ⑦
        return cls(obj)
    elif isinstance(obj, abc.MutableSequence): ⑧
        return [cls.build(item) for item in obj]
    else: ⑨
        return obj

```

- ① Cria um dict a partir do argumento `mapping`. Isso garante que teremos um mapeamento ou algo que poderá ser convertido para isso. O sublinhado duplo no prefixo de `__data` o torna um atributo privado.
- ② `__getattr__` é invocado só quando não existe um atributo com aquele `name`.
- ③ Se `name` corresponde a um atributo da instância de dict `__data`, devolve aquele atributo. É assim que chamadas como `feed.keys()` são tratadas: o método `keys` é um atributo do dict `__data`.
- ④ Caso contrário, obtém o item `self.__data` com a chave `name`, e devolve o resultado da chamada `FrozenJSON.build()` com aquele argumento.
- ⑤ Implementar `__dir__` suporta a função embutida `dir()`, que por sua vez suporta *auto-complete* no console padrão de Python, bem como no IPython, no Jupyter Notebook, etc. Este código simples vai permitir *auto-complete* recursivo baseado nas chaves em `self.__data`, porque `__getattr__` cria instâncias de `FrozenJSON` sob demanda, facilitando a exploração interativa dos dados.

- ⑥ Este é um construtor alternativo, um uso comum do decorador `@classmethod`.
- ⑦ Se `obj` é um mapeamento, cria um `FrozenJSON` com ele. Este é um exemplo de tipagem ganso—veja a «Seção 13.5» [fpy.li/cq] (vol.2) caso precise rever este conceito.
- ⑧ Se for uma `MutableSequence`, criamos uma `list`, passando recursivamente cada item em `obj` para `.build()`.
- ⑨ Se não for um `dict` ou uma `list`, devolve o item como está.

Cada instância de `FrozenJSON` contém um atributo de instância privado `__data`, armazenado sob o nome `_FrozenJSON__data`, como explicado na «Seção 11.10» [fpy.li/cr] (vol.2).

Tentativas de recuperar atributos por outros nomes vão disparar `__getattr__`. Primeiro, esse método verá se o `dict` vinculado a `self.__data` tem um atributo (não uma chave!) com aquele nome. Assim podemos invocar métodos do `dict`, como `.items()`, pois neste caso o `FrozenJSON` vai invocar `self.__data.items()`. Se `self.__data` não tiver um atributo com o nome dado, `__getattr__` usa o nome como chave para recuperar um item de `self.__data`, e passa aquele item para `FrozenJSON.build`. Assim podemos navegar por estruturas aninhadas nos dados JSON, já que cada mapeamento aninhado é convertido para outra instância de `FrozenJSON` pelo método de classe `build`.

Observe que `FrozenJSON` não transforma ou armazena o conjunto de dados original. Conforme navegamos pelos dados, `__getattr__` cria continuamente instâncias de `FrozenJSON`. Isto funciona bem com um conjunto de dados não muito grande, em um script que só será usado para explorar ou converter os dados.

Qualquer script que gera dinamicamente nomes de atributos a partir de dados arbitrários precisa lidar com uma questão: as chaves nos dados podem não ser nomes adequados de atributos. A próxima seção fala disso.

22.2.2. O problema do nome de atributo inválido

O código de `FrozenJSON` não funciona com nomes de atributos que sejam palavras reservadas de Python. Por exemplo, se você criar um objeto assim:

```
>>> student = FrozenJSON({'name': 'Jim Bo', 'class': 1982})
```


não será possível acessar `student.class`, porque `class` é uma palavra reservada no Python:

```
>>> student.class
File "<stdin>", line 1
    student.class
      ^
SyntaxError: invalid syntax
```

Claro, sempre é possível fazer assim:

```
>>> getattr(student, 'class')
1982
```

Mas a ideia de `FrozenJSON` é oferecer acesso conveniente aos dados, então uma solução melhor é verificar se uma chave no mapeamento passado para `FrozenJSON.__init__` é uma palavra reservada e, em caso positivo, anexar um `_` a ela, de forma que o atributo possa ser acessado assim:

```
>>> student.class_
1982
```

Podemos fazer isto substituindo o `__init__` de uma linha do Exemplo 4 pela versão no Exemplo 5.

Exemplo 5. `explore1.py`: anexa um `_` a nomes de atributo que são palavras reservadas do Python

```
def __init__(self, mapping):
    self.__data = {}
    for key, value in mapping.items():
        if keyword.iskeyword(key): ①
            key += '_'
        self.__data[key] = value
```

- ① A função `keyword.iskeyword(...)` é o que precisamos; para usá-la, precisamos importar o módulo `keyword`; a importação está antes deste trecho do código.

Um problema similar pode surgir se uma chave em um registro JSON não for um identificador válido em Python:

```
>>> x = FrozenJSON({'2be': 'or not'})
>>> x.2be
File "<stdin>", line 1
    x.2be
      ^
SyntaxError: invalid syntax
```

Essas chaves problemáticas são fáceis de detectar no Python 3, porque a classe `str` oferece o método `s.isidentifier()`, que informa se `s` é um identificador Python válido, de acordo com a gramática da linguagem. Mas transformar uma chave que não seja um identificador válido em um nome de atributo válido não é trivial. Uma solução seria implementar `__getitem__` para permitir acesso a atributos usando uma notação como `x['2be']`. Em nome da simplicidade, não vou me preocupar com esse problema.

Após essa pequena conversa sobre os nomes de atributos dinâmicos, vamos examinar outra característica essencial de `FrozenJSON`: a lógica do método de classe `build`. Este método é invocado por `__getattr__` para devolver um tipo diferente de objeto, dependendo do valor do atributo que está sendo acessado: estruturas aninhadas são convertidas para instâncias de `FrozenJSON` ou listas de instâncias de `FrozenJSON`.

Em vez de usar um método de classe, podemos implementar esta lógica no método especial `__new__`, como veremos a seguir.

22.2.3. Criação flexível de objetos com `__new__`

Muitas vezes dizemos que o `__init__` é o "método construtor", mas isso é porque adotamos o jargão de outras linguagens. No Python, `__init__` recebe `self` como primeiro argumento, portanto o objeto já foi construído pelo interpretador quando ele invoca `__init__`. Além disso, `__init__` não devolve um valor. Então, na verdade, esse método é um inicializador, não propriamente um construtor.^[5]

Quando uma classe é invocada para criar uma instância, Python invoca o método especial `__new__` da classe para construir a instância.

É um método de classe, mas recebe tratamento especial, então o decorador `@classmethod` não é aplicado a ele. Python recebe a instância devolvida por `__new__`, e daí a passa como o primeiro argumento (`self`) para `__init__`. Raramente precisamos escrever um `__new__`, pois a implementação herdada de `object` atende aos casos comuns.

Se necessário, o método `__new__` pode devolver uma instância de uma classe diferente. Quando isso acontece, o interpretador não invoca `__init__`. Em outras palavras, a lógica de Python para criar um objeto é similar a esse pseudo-código:

```
# pseudocódigo
def criar(a_classe, algum_arg):
    novo_objeto = a_classe.__new__(algum_arg)
    if isinstance(novo_objeto, a_classe):
        novo_objeto.__init__(algum_arg)
    return novo_objeto

# as instruções abaixo são praticamente equivalentes
p = Quitute('pão de queijo')
p = criar(Quitute, 'pão de queijo')
```

O Exemplo 6 mostra uma variante de `FrozenJSON` onde refatorei a lógica do método `build` para o método `__new__`.

Exemplo 6. explore2.py: usando `__new__` para criar novos objetos, que podem ou não ser instâncias de `FrozenJSON`

```
from collections import abc
import keyword

class FrozenJSON:
    """A read-only façade for navigating a JSON-like object
    using attribute notation
    """

    def __new__(cls, arg): ①
        if isinstance(arg, abc.Mapping):
            return super().__new__(cls) ②
        elif isinstance(arg, abc.MutableSequence): ③
            return [cls(item) for item in arg]
```

```

    else:
        return arg

    def __init__(self, mapping):
        self.__data = {}
        for key, value in mapping.items():
            if keyword.iskeyword(key):
                key += '_'
            self.__data[key] = value

    def __getattr__(self, name):
        try:
            return getattr(self.__data, name)
        except AttributeError:
            return FrozenJSON(self.__data[name]) ④

    def __dir__(self):
        return self.__data.keys()

```

- ① Por ser um método de classe, `__new__` recebe como primeiro argumento é a própria classe, e os demais argumentos são os mesmos passados para `__init__`, exceto o `self`.
- ② O comportamento default é delegar para o `__new__` de uma superclasse. Neste caso, estamos invocando o `__new__` da classe `object`, passando `FrozenJSON` como único argumento.
- ③ As linhas restantes de `__new__` são exatamente as do antigo método `build`.
- ④ Aqui é onde invocávamos `FrozenJSON.build`; agora invocamos apenas a classe, e Python internamente invoca `FrozenJSON.__new__`.

O método `__new__` recebe uma classe como primeiro argumento porque, normalmente, o objeto criado será uma instância daquela classe. Então, em `FrozenJSON.__new__`, quando a expressão `super().__new__(cls)` invoca `object.__new__(FrozenJSON)`, a instância criada pela classe `object` será uma instância de `FrozenJSON`. O atributo `__class__` da nova instância terá uma referência para `FrozenJSON`, apesar de que a instância será construída por `object.__new__`, implementado em C, nas entranhas do Python.

O conjunto de dados da OSCON está organizado de um jeito pouco amigável à exploração interativa. Por exemplo, o evento no índice 40, intitulado 'There Will

Be Bugs' (Haverá Bugs) tem dois palestrantes, 3471 e 5199. Encontrar os nomes dos palestrantes é chato, pois esses são números de série e a lista `Schedule.speakers` não está indexada por eles. Para obter cada palestrante, precisamos iterar sobre a lista até encontrar um registro com o número de série correspondente. Nossa próxima tarefa é reestruturar os dados para preparar a recuperação automática de registros relacionados.

22.3. Propriedades computadas

Vimos o decorador `@property` pela primeira vez na «Seção 11.7» [fpy.li/c9] (vol.2). No Exemplo 7 daquela seção usamos duas propriedades na classe `Vector2d` para que os atributos `x` e `y` fossem limitados à leitura (*read-only*). Aqui veremos propriedades que calculam valores, levando a uma discussão sobre como armazenar tais valores.

Os registros na lista `'events'` dos dados da OSCON contêm números de série apontando para registros nas listas `'speakers'` e `'venues'`, como se fossem chaves estrangeiras em um banco de dados relacional. Por exemplo, esse é o registro de uma palestra (com a descrição parcial terminando em reticências):

```
{ "serial": 33950,
  "name": "There *Will* Be Bugs",
  "event_type": "40-minute conference session",
  "time_start": "2014-07-23 14:30:00",
  "time_stop": "2014-07-23 15:10:00",
  "venue_serial": 1449,
  "description": "If you're pushing the envelope of programming...",
  "website_url": "http://oscon.com/2014/sched/33950",
  "speakers": [3471, 5199],
  "categories": ["Python"] }
```

Vamos implementar uma classe `Event` com propriedades `venue` e `speakers`, para devolver automaticamente os dados relacionados—em outras palavras, "desreferenciar" o número de série. Dada uma instância de `Event`, o Exemplo 7 mostra o comportamento desejado.

Exemplo 7. Ler venue e speakers devolve objetos Record

```
>>> event ①
<Event 'There *Will* Be Bugs'>
>>> event.venue ②
<Record serial=1449>
>>> event.venue.name ③
'Portland 251'
>>> for spkr in event.speakers: ④
...     print(f'{spkr.serial}: {spkr.name}')
...
3471: Anna Martelli Ravenscroft
5199: Alex Martelli
```

- ① Dada uma instância de Event...
- ② ...acessar event.venue devolve um objeto Record em vez de um número de série.
- ③ Agora é fácil obter o nome do venue.
- ④ A propriedade event.speakers devolve uma lista de instâncias de Record.

Como sempre, vamos criar o código passo a passo, começando com a classe Record e uma função para ler dados JSON e devolver um dict com instâncias de Record.

22.3.1. Passo 1: criação de atributos baseados em dados

O Exemplo 8 mostra o doctest para orientar este primeiro passo.

Exemplo 8. Testando schedule_v1.py (do Exemplo 9)

```
>>> records = load(JSON_PATH) ①
>>> speaker = records['speaker.3471'] ②
>>> speaker ③
<Record serial=3471>
>>> speaker.name, speaker.twitter ④
('Anna Martelli Ravenscroft', 'annaraven')
```

- ① Constrói um dict com os dados JSON; a função load está no Exemplo 9.
- ② As chaves em records são strings criadas a partir do tipo de registro e do número de série.
- ③ speaker é uma instância da classe Record, definida no Exemplo 9.
- ④ Campos do JSON original podem ser acessados como atributos de instância de Record.

O código de *schedule_v1.py* está no Exemplo 9.

Exemplo 9. schedule_v1.py: reorganizando os dados de agendamento da OSCON

```
import json

JSON_PATH = 'data/osconfeed.json'

class Record:
    def __init__(self, **kwargs):
        self.__dict__.update(kwargs) ①

    def __repr__(self):
        return f'<{self.__class__.__name__} serial={self.serial!r}>' ②

def load(path=JSON_PATH):
    records = {} ③
    with open(path) as fp:
        raw_data = json.load(fp) ④
        for collection, raw_records in raw_data['Schedule'].items(): ⑤
            record_type = collection[:-1] ⑥
            for raw_record in raw_records:
                key = f'{record_type}.{raw_record["serial"]}' ⑦
                records[key] = Record(**raw_record) ⑧
    return records
```

- ① Isto é um atalho comum para construir uma instância com atributos criados a partir de argumentos nomeados (a explicação detalhada está abaixo).
- ② Usa o campo serial para criar a representação customizada de Record exibida no Exemplo 8.
- ③ load vai devolver um dict de instâncias de Record no final.

- ④ Analisa o JSON, devolvendo objetos Python nativos: listas, dicts, strings, números, etc.
- ⑤ Itera sobre as quatro listas principais, chamadas 'conferences', 'events', 'speakers', e 'venues'.
- ⑥ `record_type` é o nome da lista sem o último caractere, então `speakers` se torna `speaker`. No Python ≥ 3.9 , podemos fazer isso de forma mais explícita com `collection.removesuffix('s')`—veja a *PEP 616—String methods to remove prefixes and suffixes* [fpy.li/pep616] (Métodos de string para remover prefixos e sufixos).
- ⑦ Cria a key no formato 'speaker.3471'.
- ⑧ Cria uma instância de `Record` e a armazena em `records` com a chave `key`.

O método `Record.__init__` ilustra um velho truque. Lembre-se de que o `__dict__` de um objeto é onde são guardados seus atributos—a menos que `__slots__` seja declarado na classe, como vimos na «Seção 11.11» [fpy.li/28] (vol.2). Daí, atualizar o `__dict__` de uma instância é uma maneira fácil de criar um punhado de atributos naquela instância.^[6]



Dependendo da aplicação, a classe `Record` pode ter que lidar com chaves que não sejam nomes de atributo válidos, como vimos na Seção 22.2.2. Tratar essa questão nos desviaria da ideia principal deste exemplo, e o problema não ocorre no conjunto de dados que estamos explorando.

A definição de `Record` no Exemplo 9 é tão simples que você pode estar se perguntando por que não a usei antes, em vez de `FrozenJSON`. São duas razões. Primeiro, `FrozenJSON` funciona convertendo recursivamente os mapeamentos aninhados e listas, mas `Record` não precisa fazer isso, pois nosso conjunto de dados convertido não contém mapeamentos aninhados. Os registros contêm apenas strings, inteiros, listas de strings e listas de inteiros. A segunda razão: `FrozenJSON` permite acessar aos atributos no dict embutido `__data__`—para invocar métodos como `.keys()`. `Record` não oferece este acesso.



A biblioteca padrão de Python oferece classes similares a `Record`, onde cada instância tem um conjunto arbitrário de atributos criados a partir de argumentos nomeados passados a


```
__init__: types.SimpleNamespace [fpy.li/bd], argparse.Namespace
[fpy.li/be], e multiprocessing.managers.Namespace [fpy.li/bf]. Escrevi
a classe Record, mais simples, para destacar a ideia essencial:
__init__ preenchendo o __dict__ da instância.
```

Após reorganizar o conjunto de dados de cronograma, podemos aprimorar a classe Record para obter automaticamente registros de venue e speaker referenciados em um registro event. Vamos utilizar propriedades para fazer exatamente isso nos próximos exemplos.

22.3.2. Passo 2: Propriedades para recuperar um registro relacionado

O objetivo da próxima versão é: dado um registro event, ler sua propriedade venue vai devolver um Record. Isso é similar ao que o ORM (*Object Relational Mapping*, Mapeamento Relacional de Objetos) do Django faz quando acessamos um campo ForeignKey: em vez da chave, recebemos o objeto relacionado.

Vamos começar pela propriedade venue. Veja a interação parcial no Exemplo 10.

Exemplo 10. Extratos dos doctests de schedule_v2.py

```
>>> event = Record.fetch('event.33950') ①
>>> event ②
<Event 'There *Will* Be Bugs'>
>>> event.venue ③
<Record serial=1449>
>>> event.venue.name ④
'Portland 251'
>>> event.venue_serial ⑤
1449
```

- ① O método estático Record.fetch obtém um Record ou um Event do conjunto de dados.
- ② Observe que event é uma instância da classe Event.
- ③ Acessar event.venue devolve uma instância de Record.
- ④ Agora é fácil encontrar o nome de um event.venue.

⑤ `event` também tem um atributo `venue_serial`, vindo dos dados JSON.

`Event` é uma subclasse de `Record`, acrescentando um `venue` para obter os registros relacionados, e um método `__repr__` especializado.

O código dessa seção está no módulo `schedule_v2.py` [fpy.li/22-8], no repositório de código do *Python Fluente* [fpy.li/code]. O exemplo tem aproximadamente 50 linhas, então vou apresentá-lo em partes, começando pela classe `Record` aperfeiçoada.

Exemplo 11. `schedule_v2.py`: a classe `Record` com um novo método `fetch`

```
import inspect ①
import json

JSON_PATH = 'data/osconfeed.json'

class Record:

    __index = None ②

    def __init__(self, **kwargs):
        self.__dict__.update(kwargs)

    def __repr__(self):
        return f'<{self.__class__.__name__} serial={self.serial!r}>'

    @staticmethod ③
    def fetch(key):
        if Record.__index is None: ④
            Record.__index = load()
        return Record.__index[key] ⑤
```

① `inspect` será usado em `load`, lista do no Exemplo 13.

② No final, o atributo de classe privado `__index` preservará a referência ao `dict` devolvido por `load`.

③ `fetch` é um `staticmethod`, para deixar explícito que seu efeito não é influenciado pela classe ou pela instância de onde ele é invocado.

④ Preenche o `Record.__index`, se necessário.

- ⑤ E o utiliza para obter um registro com uma dada key.



Esse é um exemplo onde o uso de `staticmethod` faz sentido. O método `fetch` sempre age sobre o atributo de classe `Record.__index`, mesmo quando invocado desde uma subclasse, como `Event.fetch()`—que exploraremos a seguir. Seria equivocado programá-lo como um método de classe, pois o primeiro argumento, `cls`, nunca é usado.

Agora podemos usar a propriedade na classe `Event`, listada no Exemplo 12.

Exemplo 12. `schedule_v2.py`: a classe `Event`

```
class Event(Record): ①

    def __repr__(self):
        try:
            return f'<{self.__class__.__name__} {self.name!r}>' ②
        except AttributeError:
            return super().__repr__()

    @property
    def venue(self):
        key = f'venue.{self.venue_serial}'
        return self.__class__.fetch(key) ③
```

- ① `Event` estende `Record`.
- ② Se a instância tem um atributo `name`, esse atributo será usado para produzir uma representação customizada. Se não, invoca `__repr__` de `Record`.
- ③ A propriedade `venue` cria uma key a partir do atributo `venue_serial`, e a passa para o método de classe `fetch`, herdado de `Record` (a razão para usar `self.__class__` logo ficará clara).

A segunda linha do método `venue` no Exemplo 12 devolve `self.__class__.fetch(key)`. Por que não podemos invocar `self.fetch(key)` diretamente? A forma direta funciona com esse conjunto específico de dados da OSCON porque não há registro de evento com uma chave `'fetch'`. Mas, se um registro de evento tivesse uma chave chamada `'fetch'`, então dentro daquela

instância específica de `Event`, a referência `self.fetch` apontaria para o valor daquele campo, em vez do método de classe `fetch` que `Event` herda de `Record`. Esse é um bug sutil, e poderia facilmente escapar aos testes, pois depende do conjunto de dados.



Ao criar nomes de atributos de instância a partir de dados, sempre existe o risco de bugs causados pelo ocultamento de atributos da classe—como métodos—ou perda de dados pela sobrescrita acidental de atributos de instância existentes. Estes problemas talvez expliquem por que os dicts de Python não são como objetos JavaScript, e isto é uma vantagem.

Se a classe `Record` se comportasse mais como um mapeamento, implementando um `__getitem__` dinâmico em vez de um `__getattr__` dinâmico, não haveria risco de bugs por ocultamento ou sobrescrita. Um mapeamento customizado seria provavelmente a forma pythônica de implementar `Record`. Mas se eu tivesse seguido por aquele caminho, não estaríamos estudando os truques e as armadilhas da programação dinâmica de atributos.

A parte final deste exemplo é a função `load` revisada, no Exemplo 13.

Exemplo 13. `schedule_v2.py`: a função `load`

```
def load(path=JSON_PATH):
    records = {}
    with open(path) as fp:
        raw_data = json.load(fp)
    for collection, raw_records in raw_data['Schedule'].items():
        record_type = collection[:-1] ①
        cls_name = record_type.capitalize() ②
        cls = globals().get(cls_name, Record) ③
        if inspect.isclass(cls) and issubclass(cls, Record): ④
            factory = cls ⑤
        else:
            factory = Record ⑥
        for raw_record in raw_records: ⑦
            key = f'{record_type}.{raw_record["serial"]}'
            records[key] = factory(**raw_record) ⑧
    return records
```

- ① Até aqui, nenhuma mudança em relação ao load em *schedule_v1.py* (do Exemplo 9).
- ② Muda a primeira letra de `record_type` para maiúscula, para criar um possível nome de classe; por exemplo, `'event'` se torna `'Event'`.
- ③ Obtém um objeto com aquele nome do escopo global do módulo; se aquele objeto não existir, obtém a classe `Record`.
- ④ Se o objeto recém-obtido é uma classe, e é uma subclasse de `Record`...
- ⑤ ...vincula o nome `factory` a ele. Isto significa que `factory` pode ser qualquer subclasse de `Record`, dependendo do `record_type`.
- ⑥ Caso contrário, vincula `Record` ao nome `factory`.
- ⑦ O laço `for`, que cria a `key` e armazena os registros, é o mesmo de antes, exceto que...
- ⑧ ...o objeto armazenado em `records` é construído por `factory`, e pode ser uma instância de `Record` ou de uma subclasse, como `Event`, selecionada de acordo com o `record_type`.

Observe que o único `record_type` que tem uma classe customizada é `Event`, mas se você definir classes chamadas `Speaker` ou `Venue`, `load` usará automaticamente aquelas classes ao criar e armazenar registros, em vez da classe `Record`.

Agora vamos aplicar a mesma ideia à nova propriedade `speakers`, na classe `Events`.

22.3.3. Passo 3: Propriedade sobrescrevendo atributo existente

O nome da propriedade `venue` no Exemplo 12 não corresponde a um nome de campo nos registros da coleção `"events"`. Seus dados vêm de um campo chamado `venue_serial`. Por outro lado, cada registro na coleção `events` tem um campo `speakers`, contendo uma lista de números de série. Queremos expor essa informação na forma de uma propriedade `speakers` em instâncias de `Event`, que devolverá uma lista de instâncias de `Record`. Esta colisão de nomes exige uma atenção especial, como revela o Exemplo 14.

Exemplo 14. *schedule_v3.py*: a propriedade `speakers`

```
@property
def speakers(self):
    spkr_serials = self.__dict__['speakers'] ①
    fetch = self.__class__.fetch
    return [fetch(f'speaker.{key}')
            for key in spkr_serials] ②
```

- ① Os dados que precisamos estão em um atributo `speakers`, mas precisamos obtê-los diretamente do `__dict__` da instância, para evitar uma chamada recursiva à propriedade `speakers`.
- ② Devolve uma lista com todos os registros com chaves correspondendo aos números em `spkr_serials`.

Dentro do método `speakers`, uma tentativa de ler `self.speakers` invocará o mesmo método, provocando um `RecursionError`. Entretanto, acessando via `self.__dict__['speakers']`, evitamos o algoritmo de Python para busca de atributos, a propriedade não é acessada, e evitamos a recursão. Por esta razão, ler ou escrever dados diretamente no `__dict__` de um objeto é um truque comum em metaprogramação no Python.



O interpretador avalia `obj.my_attr` olhando primeiro a classe de `obj`. Se existe uma propriedade com o nome `my_attr`, aquela propriedade oculta um atributo de instância com o mesmo nome. Isto será demonstrado com exemplos na Seção 22.5.1, e o Capítulo 23 revelará que uma propriedade é implementada como um *descriptor* (descritor de atributo), uma abstração mais geral e poderosa.

Quando programei a compreensão de lista no Exemplo 14, meu cérebro réptil de programador pensou: "Isso talvez seja custoso." Na verdade não é, porque os eventos nos dados da OSCON contêm poucos palestrantes, então programar algo mais complexo seria uma otimização prematura. Entretanto, criar um *cache* de uma propriedade é uma necessidade comum—mas não trivial. Veremos então como fazer isso nos próximos exemplos.

22.3.4. Passo 4: Um *cache* de propriedades sob medida

Fazer *caching* de propriedades é uma necessidade comum, pois há a expectativa de que uma expressão como `event.venue` deveria ser pouco dispendiosa.^[7]

Alguma forma de *caching* poderia se tornar necessária caso o método

`Record.fetch`, invocado nas propriedades de `Event`, precise consultar um banco de dados ou uma API Web.

Na primeira edição de *Python Fluente*, programei a lógica customizada de *caching* para o método `speakers`, como mostra o Exemplo 15.

Exemplo 15. A lógica de caching customizada usando `hasattr` impede a otimização de compartilhamento de chaves

```
@property
def speakers(self):
    if not hasattr(self, '__speaker_objs'): ①
        spkr_serials = self.__dict__['speakers']
        fetch = self.__class__.fetch
        self.__speaker_objs = [fetch(f'speaker.{key}')
                               for key in spkr_serials]
    return self.__speaker_objs ②
```

- ① Se a instância não tem um atributo chamado `__speaker_objs`, obtém os objetos `speaker` e os armazena ali..
- ② Devolve `self.__speaker_objs`.

O *caching* caseiro no Exemplo 15 é bastante direto, mas criar atributos após a inicialização da instância frustra a otimização da PEP 412—Key-Sharing Dictionary [fpy.li/pep412] (Dicionário com chaves compartilhadas), como explicado na «Seção 3.9» [fpy.li/82] (vol.1). Dependendo do tamanho da massa de dados, a diferença de uso de memória pode ser importante.

Uma solução manual similar, compatível com a otimização de compartilhamento de chaves, implica em escrever um `__init__` para a classe `Event`, para criar o atributo de instância `__speaker_objs` inicializado para `None`, e então usá-lo no método `speakers`. Veja o Exemplo 16.

Exemplo 16. Armazenamento definido em `__init__` para viabilizar a otimização de compartilhamento de chaves

```
class Event(Record):

    def __init__(self, **kwargs):
        self.__speaker_objs = None
        super().__init__(**kwargs)

# 15 lines omitted...
@property
def speakers(self):
    if self.__speaker_objs is None:
        spkr_serials = self.__dict__['speakers']
        fetch = self.__class__.fetch
        self.__speaker_objs = [fetch(f'speaker.{key}')]
        for key in spkr_serials]
    return self.__speaker_objs
```

O Exemplo 15 e o Exemplo 16 ilustram técnicas simples de *caching* bastante comuns em bases de código Python legadas. Entretanto, em programas com múltiplas threads, *caches* manuais como aqueles introduzem condições de corrida que podem levar à corrupção de dados. Se duas threads estão lendo uma propriedade que não foi armazenada no *cache* anteriormente, a primeira thread precisará computar os dados para o atributo de *cache* `__speaker_objs` e a segunda thread pode ler um valor inconsistente do *cache*.

Felizmente, Python 3.8 introduziu o decorador `@functools.cached_property`, que é seguro para uso com threads (*thread safe*). Infelizmente, ele vem com algumas ressalvas, discutidas a seguir.

22.3.5. Passo 5: *Caching* de propriedades com `functools`

O módulo `functools` oferece três decoradores para *caching*. Vimos `@cache` e `@lru_cache` na «Seção 9.9.1» [fpy/li/ca] (vol.2). Python 3.8 incorporou `@cached_property`.

O decorador `functools.cached_property` faz *cache* do resultado de um método em uma variável de instância com o mesmo nome.

Por exemplo, no Exemplo 17, o valor computado pelo método `venue` é armazenado em um atributo `venue`, em `self`. Após isso, quando código cliente tenta ler `venue`, o recém-criado atributo de instância `venue` é usado, em vez do método.

Exemplo 17. Uso simples de uma `@cached_property`

```
@cached_property
def venue(self):
    key = f'venue.{self.venue_serial}'
    return self.__class__.fetch(key)
```

Na Seção 22.3.3, vimos que uma propriedade oculta um atributo de instância de mesmo nome. Se isso é verdade, como `@cached_property` pode funcionar? Se a propriedade sobrescreve o atributo de instância, o atributo `venue` será ignorado e o método `venue` será sempre invocado, computando a `key` e rodando `fetch` todas as vezes!

A resposta é um pouco triste: `cached_property` é um nome enganador. O decorador `@cached_property` não cria uma propriedade completa, ele cria um *descriptor não dominante*. Um *descriptor* é um objeto que gerencia o acesso a um atributo em outra classe. Vamos mergulhar nos *descriptores* no Capítulo 23. O decorador `property` é uma API de alto nível para criar um *descriptor dominante*. O Capítulo 23 inclui uma explicação completa sobre *descriptores dominantes* e *não dominantes*.

Por hora, vamos deixar de lado a implementação e nos concentrar nas diferenças entre `cached_property` e `property` do ponto de vista de um usuário. Raymond Hettinger os explica muito bem na «Documentação do Python» [fpy.li/bg]:

A mecânica de `cached_property()` é um tanto diferente da de `property()`. Uma propriedade normal bloqueia a escrita em atributos, a menos que um setter seja definido. Uma `cached_property`, por outro lado, permite a escrita.

O decorador `cached_property` só funciona para consultas e apenas quando um atributo de mesmo nome não existe. Quando acionada, `cached_property` escreve no atributo de mesmo nome. Leituras e escritas subsequentes daquele atributo têm precedência sobre o método decorado com `cached_property` e ele funciona como um atributo normal.

O valor "cacheado" (cached) pode ser excluído apagando-se o atributo. Isto permite que o método `cached_property` rode novamente.^[8]

Voltando à nossa classe `Event`: o comportamento específico de `@cached_property` o torna inadequado para decorar `speakers`, porque aquele método depende de um atributo existente também chamado `speakers`, contendo os números de série dos palestrantes do evento.



`@cached_property` tem algumas importantes limitações:

- Ele não pode ser usado como um substituto direto de `@property` se o método decorado acessa um atributo de instância de mesmo nome.
- Ele não pode ser usado em uma classe que defina `__slots__`.
- Ele impede a otimização de chaves compartilhadas do `__dict__` da instância, pois cria um atributo de instância após o `__init__`.

Apesar destas limitações, `@cached_property` supre uma necessidade comum de um modo simples, e é seguro para usar com threads. Seu código Python [fpy.li/22-13] é um exemplo do uso de uma *reentrant lock* [fpy.li/bp] (trava reentrante).^[9]

A documentação [fpy.li/bh] de `@cached_property` recomenda uma solução alternativa que podemos usar com `speakers`: empilhar decoradores.

Exemplo 18. Empilhando `@property` sobre `@cache`

```
@property ①
@cache ②
def speakers(self):
    spkr_serials = self.__dict__['speakers']
    fetch = self.__class__.fetch
    return [fetch(f'speaker.{key}')
            for key in spkr_serials]
```

① A ordem é importante: `@property` vai acima...

② ...de `@cache`.

Lembre-se do significado desta sintaxe com decoradores empilhados, que vimos no Exemplo 18 do «Capítulo 9» [fpy.li/9] (vol.2). As três primeiras linhas do Exemplo 18 fazem isso:

```
speakers = property(cache(speakers))
```

O `@cache` é aplicado a `speakers`, devolvendo uma nova função. Esta função é então decorada por `@property`, que a substitui por uma propriedade criada na hora.

Isto encerra nossa discussão de propriedades somente para leitura e decoradores de *caching*, explorando o conjunto de dados da OSCON.

Na próxima seção iniciamos uma nova série de exemplos, criando propriedades de leitura e escrita.

22.4. Propriedades para validação de atributos

Além de computar valores de atributos, as propriedades também são usadas para impor regras de negócio, transformando um atributo público em um atributo protegido por um *getter* e um *setter*, sem afetar o código cliente. Vamos explorar um exemplo mais elaborado.

22.4.1. `LineItem` Versão #1: Classe para um item em um pedido

Imagine uma aplicação para uma loja que vende alimentos orgânicos a granel, onde os fregueses podem encomendar nozes, frutas secas e cereais por peso. Neste sistema, cada pedido contém uma sequência de produtos, e cada produto é representado por uma instância de uma classe, como no Exemplo 19.

Exemplo 19. `bulkfood_v1.py`: a classe `LineItem` mais simples

```
class LineItem:

    def __init__(self, description, weight, price):
        self.description = description
        self.weight = weight
        self.price = price
```

```
def subtotal(self):  
    return self.weight * self.price
```

Este código é simples e agradável. Talvez simples demais. Exemplo 20 mostra um problema.

Exemplo 20. Peso negativo resulta em subtotal negativo

```
>>> raisins = LineItem('Golden raisins', 10, 6.95)  
>>> raisins.subtotal()  
69.5  
>>> raisins.weight = -20 # garbage in...  
>>> raisins.subtotal()    # garbage out...  
-139.0
```

Apesar de ser um exemplo inventado, não é tão fantasioso quanto se poderia imaginar. Aqui está uma história do início da Amazon.com:

Descobrimos que os clientes podiam encomendar uma quantidade negativa de livros! E nós creditaríamos seus cartões de crédito com o preço e, suponho, esperaríamos que eles nos enviassem os livros.^[10]

— Jeff Bezos, fundador da Amazon.com

Como evitar isso? Poderíamos mudar a interface de `LineItem` para usar um *getter* e um *setter* para o atributo `weight` (peso). Este seria o estilo Java, e não está errado, mas podemos fazer melhor. É natural definir o `weight` de um item apenas atribuindo um valor a este atributo; e talvez o sistema esteja em produção, com outras partes já acessando `item.weight` diretamente. Neste caso, o estilo Python seria substituir o atributo de dados por uma propriedade.

22.4.2. `LineItem` versão #2: Uma propriedade de validação

Implementar uma propriedade nos permitirá usar um *getter* e um *setter*, sem mudar a interface pública de `LineItem`: para definir o `weight` de um `LineItem` ainda poderemos escrever `raisins.weight = 12`.

O Exemplo 21 lista o código de uma propriedade de leitura e escrita para `weight`.

Exemplo 21. `bulkfood_v2.py`: um `LineItem` com uma propriedade `weight`

```
class LineItem:

    def __init__(self, description, weight, price):
        self.description = description
        self.weight = weight ①
        self.price = price

    def subtotal(self):
        return self.weight * self.price

    @property ②
    def weight(self): ③
        return self.__weight ④

    @weight.setter ⑤
    def weight(self, value):
        if value > 0:
            self.__weight = value ⑥
        else:
            raise ValueError('value must be > 0') ⑦
```

- ① Aqui o *setter* da propriedade já está em uso, assegurando que nenhuma instância com peso negativo possa ser criada.
- ② `@property` decora o método *getter*.
- ③ Todos os métodos que implementam a propriedade compartilham o mesmo nome do atributo público: `weight`.
- ④ O valor é armazenado em um atributo privado `__weight`.
- ⑤ O *getter* decorado ganha um atributo `.setter`, que também é um decorador; isto conecta o *getter* e o *setter*.
- ⑥ Se o valor for maior que zero, atualizamos o `__weight` privado.
- ⑦ Caso contrário, um `ValueError` é levantado.

Agora não é possível criar uma `LineItem` com peso inválido:

```
>>> walnuts = LineItem('walnuts', 0, 10.00)
Traceback (most recent call last):
...
ValueError: value must be > 0
```

Assim protegemos `weight` impedindo que usuários forneçam valores negativos. Fregueses normalmente não podem definir o preço de um produto, mas um erro administrativo ou um bug poderiam criar um `LineItem` com um `price` negativo. Para evitar isso, poderíamos também transformar `price` em uma propriedade, mas isso levaria a alguma repetição no nosso código.

Lembre-se da citação de Paul Graham no Capítulo 17: "Quando vejo padrões em meus programas, considero isso um mau sinal." A cura para a repetição é a abstração. Há duas maneiras de abstrair definições de propriedades: usar uma fábrica de propriedades ou uma classe descritora. A abordagem via classe descritora é mais flexível, e dedicaremos o Capítulo 23 a uma discussão completa deste mecanismo. Na verdade, propriedades são, elas mesmas, implementadas como classes descritoras. Mas aqui vamos seguir com nossa exploração das propriedades, implementando uma fábrica de propriedades em forma de função.

Mas antes de podermos implementar uma fábrica de propriedades, precisamos entender melhor as propriedades em si.

22.5. Propriedades em profundidade

Apesar de ser frequentemente usada como um decorador, `property` é na verdade uma classe embutida. No Python, funções e classes são muitas vezes intercambiáveis, pois ambas são invocáveis e não há um operador `new` para instanciação de objeto, então invocar um construtor não é diferente de invocar uma função fábrica. E ambas podem ser usadas como decoradores, desde que elas devolvam um novo invocável, que seja um substituto adequado do invocável decorado.

Esta é a assinatura completa do construtor de `property`:

```
property(fget=None, fset=None, fdel=None, doc=None)
```

Todos os argumentos são opcionais, e se uma função não é fornecida para algum deles, a operação correspondente não será permitida pela propriedade.

O tipo `property` foi introduzido no Python 2.2, mas a sintaxe `@` do decorador só surgiu no Python 2.4. Então, por alguns anos, propriedades eram definidas passando as funções de acesso nos dois primeiros argumentos.

A sintaxe "clássica" para definir propriedades sem decoradores é ilustrada pelo Exemplo 22.

Exemplo 22. `bulkfood_v2b.py`: igual ao Exemplo 21, sem usar decoradores

```
class LineItem:

    def __init__(self, description, weight, price):
        self.description = description
        self.weight = weight
        self.price = price

    def subtotal(self):
        return self.weight * self.price

    def get_weight(self): ①
        return self.__weight

    def set_weight(self, value): ②
        if value > 0:
            self.__weight = value
        else:
            raise ValueError('value must be > 0')

    weight = property(get_weight, set_weight) ③
```

① Um *getter* simples.

② Um *setter* simples.

③ Cria a *property* e a vincula a um atributo da classe.

Em algumas situações, a forma clássica é melhor que a sintaxe do decorador; o código da fábrica de propriedade, que discutiremos em breve, é um exemplo. Por outro lado, no corpo de uma classe com muitos métodos, os decoradores tornam explícito quais são os *getters* e os *setters*, sem depender da convenção do uso dos prefixos *get* e *set* em seus nomes.

A presença de uma propriedade em uma classe afeta como os atributos nas instâncias daquela classe podem ser encontrados, de uma forma que à primeira vista pode ser surpreendente. A próxima seção explica isso.

22.5.1. Propriedades sobrescrevem atributos de instância

Propriedades são sempre atributos de uma classe, mas elas controlam o acesso a atributos nas instâncias da classe.

Na «Seção 11.12» [fpy.li/cs] (vol.2), vimos que quando uma instância e sua classe têm um atributo de dados com o mesmo nome, o atributo de instância sobrescreve, ou oculta, o atributo da classe—ao menos quando lido através daquela instância. O Exemplo 23 ilustra esse ponto.

Exemplo 23. Atributo de instância oculta o atributo de classe data

```
>>> class Class: ①
...     data = 'the class data attr'
...     @property
...     def prop(self):
...         return 'the prop value'
...
>>> obj = Class()
>>> vars(obj) ②
{}
>>> obj.data ③
'the class data attr'
>>> obj.data = 'bar' ④
>>> vars(obj) ⑤
{'data': 'bar'}
>>> obj.data ⑥
'bar'
>>> Class.data ⑦
'the class data attr'
```


- ① Define `Class` com dois atributos de classe: o atributo `data` e a propriedade `prop`.
- ② `vars` devolve o `__dict__` de `obj`, mostrando que ele não tem atributos de instância.
- ③ Ler de `obj.data` obtém o valor de `Class.data`.
- ④ Escrever em `obj.data` cria um atributo de instância.
- ⑤ Inspecciona a instância, para ver o atributo de instância.
- ⑥ Ler agora de `obj.data` obtém o valor do atributo da instância. Quanto lido a partir da instância `obj`, o `data` da instância oculta o `data` da classe.
- ⑦ O atributo `Class.data` está intacto.

Agora vamos tentar sobrescrever o atributo `prop` na instância `obj`. Continuando a sessão de console anterior, temos o Exemplo 24.

Exemplo 24. Um atributo de instância não oculta uma propriedade da classe (continuando do Exemplo 23)

```
>>> Class.prop ①
<property object at 0x1072b7408>
>>> obj.prop ②
'the prop value'
>>> obj.prop = 'foo' ③
Traceback (most recent call last):
...
AttributeError: can't set attribute
>>> obj.__dict__['prop'] = 'foo' ④
>>> vars(obj) ⑤
{'data': 'bar', 'prop': 'foo'}
>>> obj.prop ⑥
'the prop value'
>>> Class.prop = 'baz' ⑦
>>> obj.prop ⑧
'foo'
```

- ① Ler `prop` diretamente de `Class` obtém o próprio objeto propriedade, sem executar seu método *getter*.
- ② Ler `obj.prop` executa o *getter* da propriedade.

- ③ Tentar definir um atributo `prop` na instância falha.
- ④ Inserir `'prop'` diretamente em `obj.__dict__` funciona.
- ⑤ Podemos ver que agora `obj` tem dois atributos de instância: `data` e `prop`.
- ⑥ Entretanto, ler `obj.prop` ainda executa o *getter* da propriedade. A propriedade não é ocultada pelo atributo de instância.
- ⑦ Sobrescrever `Class.prop` destrói o objeto propriedade.
- ⑧ Agora `obj.prop` obtém o atributo de instância. `Class.prop` não é mais uma propriedade, então ela não mais sobrescreve `obj.prop`.

Como uma demonstração final, vamos adicionar uma propriedade a `Class`, e vê-la sobrescrever um atributo de instância. O Exemplo 25 retoma a sessão onde o Exemplo 24 parou.

Exemplo 25. Uma nova propriedade de classe oculta o atributo de instância existente (continuando do Exemplo 24)

```
>>> obj.data ①
'bar'
>>> Class.data ②
'the class data attr'
>>> Class.data = property(lambda self: 'the "data" prop value') ③
>>> obj.data ④
'the "data" prop value'
>>> del Class.data ⑤
>>> obj.data ⑥
'bar'
```

- ① `obj.data` obtém o atributo de instância `data`.
- ② `Class.data` obtém o atributo de classe `data`.
- ③ Sobrescreve `Class.data` com uma nova propriedade.
- ④ `obj.data` está agora ocultado pela propriedade `Class.data`.
- ⑤ Apaga a propriedade.
- ⑥ `obj.data` agora lê novamente o atributo de instância `data`.

O ponto principal desta seção é que uma expressão como `obj.data` não começa a busca por `data` em `obj`. A busca na verdade começa em `obj.__class__`, e Python só olha para a instância `obj` se não houver uma propriedade chamada `data` na classe. Isto se aplica a *descritores dominantes* em geral, dos quais as propriedades são apenas um exemplo. Para mais detalhes, aguarde o Capítulo 23.

Voltemos às propriedades. Toda unidade de código de Python—módulos, funções, classes, métodos—pode conter uma docstring. O próximo tópico mostra como anexar documentação às propriedades.

22.5.2. Documentação de propriedades

Quando ferramentas como uma IDE ou a função `help()` do console precisam mostrar a documentação de uma propriedade, elas extraem a informação do atributo `__doc__` da propriedade.

Se usada com a sintaxe clássica de invocação, `property` pode receber a string de documentação no argumento `doc`:

```
weight = property(get_weight, set_weight, doc='weight in kilograms')
```

Alternativamente, a docstring do método *getter*—aquele que recebe o decorador `@property`—é usada como documentação da propriedade toda. A Figura 1 mostra telas de ajuda geradas a partir do código no Exemplo 26.

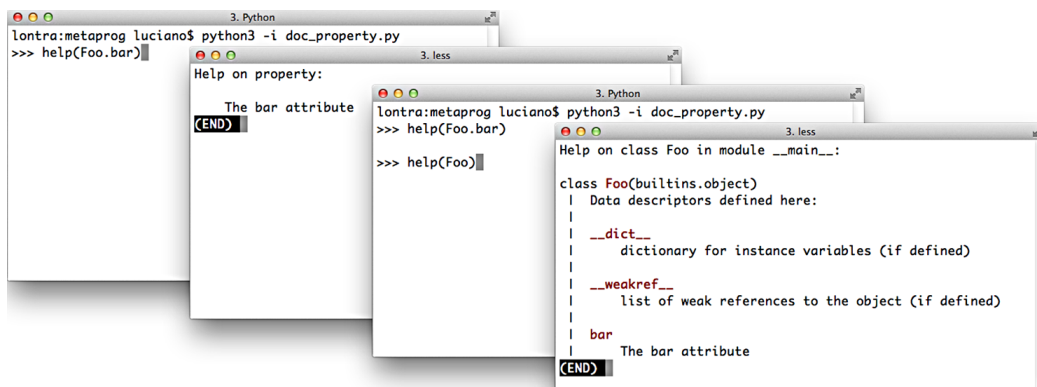


Figura 1. Capturas de tela do console de Python para os comandos `help(Foo.bar)` e `help(Foo)`. O código-fonte está no Exemplo 26.

Exemplo 26. Documentação para uma propriedade

```
class Foo:

    @property
    def bar(self):
        """The bar attribute"""
        return self.__dict__['bar']

    @bar.setter
    def bar(self, value):
        self.__dict__['bar'] = value
```

Agora que cobrimos o essencial sobre as propriedades, vamos voltar para a questão de proteger os atributos `weight` e `price` de `LineItem`, para que eles só aceitem valores maiores que zero—mas sem codar na unha dois pares de *getters/setters* praticamente idênticos.

22.6. Uma fábrica de propriedades

Vamos programar uma fábrica para criar propriedades `quantity` (quantidade). Os atributos gerenciados representam quantidades que não podem ser negativas ou zero na aplicação. O Exemplo 27 mostra a aparência cristalina da classe `LineItem` usando duas instâncias de propriedades `quantity`: uma para gerenciar o atributo `weight`, a outra para o `price`.

Exemplo 27. *bulkfood_v2prop.py*: a fábrica de propriedades `quantity` em ação

```
class LineItem:
    weight = quantity('weight') ①
    price = quantity('price') ②

    def __init__(self, description, weight, price):
        self.description = description
        self.weight = weight ③
        self.price = price

    def subtotal(self):
        return self.weight * self.price ④
```

- ① A fábrica cria a propriedade customizada, `weight`...
- ② e a propriedade, `price`, ambas como atributos da classe.
- ③ A propriedade já está ativa, rejeitando peso negativo ou 0.
- ④ As propriedades também são usadas aqui, para recuperar os valores armazenados na instância.

Recorde que propriedades são atributos de classe. Ao criar cada propriedade `quantity`, precisamos passar o nome do atributo de `LineItem` que será gerenciado por aquela propriedade específica. É chato escrever `weight` duas vezes assim:

```
weight = quantity('weight')
```

Mas evitar tal repetição é complicado, pois a propriedade não tem como saber qual nome de atributo será vinculado a ela. Lembre-se: o lado direito de uma atribuição é avaliado primeiro, então quando `quantity()` é invocada, o atributo `weight` ainda não existe na classe.



Aperfeiçoar a propriedade `quantity` para que o usuário não precise redigitar o nome do atributo é um problema não trivial de metaprogramação, que resolveremos no Capítulo 23.

O Exemplo 28 apresenta a fábrica de propriedades `quantity`.

Exemplo 28. `bulkfood_v2prop.py`: a fábrica de propriedades `quantity`

```
def quantity(storage_name): ①

    def qty_getter(instance): ②
        return instance.__dict__[storage_name] ③

    def qty_setter(instance, value): ④
        if value > 0:
            instance.__dict__[storage_name] = value ⑤
        else:
            raise ValueError('value must be > 0')

    return property(qty_getter, qty_setter) ⑥
```

- ① O argumento `storage_name`, onde os dados de cada propriedade são armazenados; para `weight`, o nome do armazenamento será `'weight'`.
- ② O primeiro argumento do `qty_getter` poderia se chamar `self`, mas soaria estranho, pois isso não é o corpo de uma classe; `instance` se refere à instância de `LineItem` onde o atributo será armazenado.
- ③ `qty_getter` se refere a `storage_name`, então ele será preservado na clausura desta função; o valor é obtido diretamente de `instance.__dict__`, para contornar a propriedade e evitar uma recursão infinita.
- ④ `qty_setter` é definido, e também recebe `instance` como primeiro argumento.
- ⑤ O `value` é armazenado diretamente no `instance.__dict__`, novamente contornando a propriedade.
- ⑥ Cria e devolve um objeto propriedade customizado.

As partes do Exemplo 28 que merecem um estudo mais cuidadoso giram em torno da variável `storage_name`.

Quando programamos uma propriedade da maneira tradicional, o nome do atributo onde um valor será armazenado está definido explicitamente nos métodos *getter* e *setter*. Mas aqui as funções `qty_getter` e `qty_setter` são genéricas, e dependem da variável `storage_name` para saber onde ler/escrever o atributo gerenciado no `__dict__` da instância. Cada vez que a fábrica `quantity` é invocada para criar uma propriedade, `storage_name` precisa ser definida com um valor único.

As funções `qty_getter` e `qty_setter` serão encapsuladas pelo objeto `property`, criado na última linha da função fábrica. Mais tarde, quando forem chamadas para cumprir seus papéis, estas funções lerão a `storage_name` de suas clausuras para saber onde os valores dos atributos gerenciados estão armazenados.

No Exemplo 29, criei e inspecionei uma instância de `LineItem`, expondo os atributos armazenados.

Exemplo 29. bulkfood_v2prop.py: explorando propriedades e atributos de armazenamento

```
>>> nutmeg = LineItem('Moluccan nutmeg', 8, 13.95)
>>> nutmeg.weight, nutmeg.price ①
(8, 13.95)
>>> nutmeg.__dict__ ②
{'description': 'Moluccan nutmeg', 'weight': 8, 'price': 13.95}
```

- ① Lendo o `weight` e o `price` através das propriedades que ocultam os atributos de instância de mesmo nome.
- ② Usando `vars` para inspecionar a instância `nutmeg`: aqui vemos os reais atributos de instância usados para armazenar os valores.

Observe como as propriedades criadas por nossa fábrica se valem do comportamento descrito na Seção 22.5.1: a propriedade `weight` sobrescreve o atributo de instância `weight`, de forma que qualquer referência a `self.weight` ou `nutmeg.weight` é tratada pelas funções da propriedade, e a única maneira de contornar a lógica da propriedade é acessando diretamente o `__dict__` da instância.

O código do Exemplo 28 pode ser um pouco complicado, mas é conciso: seu tamanho é idêntico ao do par *getter/setter* decorado que define apenas a propriedade `weight` no Exemplo 21. A definição de `LineItem` no Exemplo 27 é mais legível sem o ruído de *getters* e *setters*.

Em um sistema real, o mesmo tipo de validação pode aparecer em muitos campos espalhados por várias classes, e a fábrica `quantity` estaria em um módulo utilitário, pronta para uso em todas as partes do sistema que precisem dela. Depois, aquela fábrica simples poderia ser refatorada em uma classe descritora mais extensível, com subclasses especializadas realizando diferentes validações. Faremos isso no Capítulo 23.

Vamos agora encerrar a discussão das propriedades com a questão da exclusão de atributos.

22.7. Tratando a exclusão de atributos

Podemos usar a instrução `del` para excluir não apenas variáveis, mas também atributos:

```
>>> class Demo:
...     pass
...
>>> d = Demo()
>>> d.color = 'green'
>>> d.color
'green'
>>> del d.color
>>> d.color
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'Demo' object has no attribute 'color'
```

Na prática, a exclusão de atributos não é algo que se faça todo dia no Python, e a necessidade de lidar com isso no caso de uma propriedade é ainda mais rara. Mas tal operação é suportada, e consigo pensar em um exemplo bobo para demonstrá-la.

Em uma definição de propriedade, o decorador `@my_property.deleter` encapsula o método responsável por excluir o atributo gerenciado pela propriedade. Como prometido, o tolo Exemplo 30 foi inspirado pela cena com o Cavaleiro Negro, do filme *Monty Python e o Cálice Sagrado*.^[11]

Exemplo 30. blackknight.py

```
class BlackKnight:

    def __init__(self):
        self.phrases = [
            ('an arm', "'Tis but a scratch."),
            ('another arm', "It's just a flesh wound."),
            ('a leg', "I'm invincible!"),
            ('another leg', "All right, we'll call it a draw.")
        ]
```



```

@property
def member(self):
    print('next member is:')
    return self.phrases[0][0]

@member.deleter
def member(self):
    member, text = self.phrases.pop(0)
    print(f'BLACK KNIGHT (loses {member}) -- {text}')

```

Os doctests em *blackknight.py* estão no Exemplo 31.

Exemplo 31. blackknight.py: doctests para Exemplo 30 (o Cavaleiro Negro nunca reconhece a derrota)

```

>>> knight = BlackKnight()
>>> knight.member
next member is:
'an arm'
>>> del knight.member
BLACK KNIGHT (loses an arm) -- 'Tis but a scratch.
>>> del knight.member
BLACK KNIGHT (loses another arm) -- It's just a flesh wound.
>>> del knight.member
BLACK KNIGHT (loses a leg) -- I'm invincible!
>>> del knight.member
BLACK KNIGHT (loses another leg) -- All right, we'll call it a draw.

```

Usando a sintaxe clássica de invocação em vez de decoradores, o argumento `fdel` configura a função de exclusão. Por exemplo, a propriedade `member` seria escrita assim no corpo da classe `BlackKnight`:

```

member = property(member_getter, fdel=member_deleter)

```

Se você não estiver usando uma propriedade, a exclusão de atributos pode ser tratada implementando o método especial de nível mais baixo `__delattr__`, apresentado na Seção 22.8.3. Programar uma classe tola com `__delattr__` fica como exercício para a leitora que queira procrastinar.

Propriedades são recursos poderosos, mas algumas vezes alternativas mais simples ou de nível mais baixo são preferíveis. Na seção final deste capítulo, vamos revisar algumas das APIs essenciais oferecidas pelo Python para programação de atributos dinâmicos.

22.8. Atributos e funções essenciais para tratamento de atributos

Ao longo deste capítulo, e mesmo antes no livro, usamos algumas funções embutidas e métodos especiais oferecidos pelo Python para lidar com atributos dinâmicos. Esta seção os reúne em um único lugar para uma visão geral, pois sua documentação está espalhada na documentação oficial.

22.8.1. Atributos especiais que afetam o tratamento de atributos

O comportamento de muitas das funções e dos métodos especiais elencados nas próximas seções dependem de três atributos especiais:

`__class__`

Uma referência à classe do objeto (isto é, `obj.__class__` é o mesmo que `type(obj)`). Python procura por métodos especiais tal como `__getattr__` apenas na classe do objeto, e não nas instâncias em si.

`__dict__`

Um mapeamento que armazena os atributos passíveis de escrita de um objeto ou de uma classe. Um objeto que tenha um `__dict__` pode ter novos atributos arbitrários definidos a qualquer tempo. Se uma classe tem um atributo `__slots__`, então suas instâncias não podem ter um `__dict__`. Veja `__slots__` (abaixo).

`__slots__`

Um atributo que pode ser definido em uma classe para economizar memória. `__slots__` é uma tuple de strings, nomeando os atributos permitidos^[12]. Se o nome `'__dict__'` não estiver em `__slots__`, as instâncias daquela classe então não terão um `__dict__` próprio, e apenas os atributos listados em `__slots__` serão permitidos naquelas instâncias. Revise a «Seção 11.11» [fpy.li/28] (vol.2) para recordar esse tópico.

22.8.2. Funções embutidas para tratamento de atributos

Essas cinco funções embutidas executam leitura, escrita e introspecção de atributos de objetos:

`dir([object])`

Lista a maioria dos atributos de um objeto. A documentação oficial [fpy.li/bj] diz que o objetivo de `dir` é o uso interativo, então ele não fornece uma lista completa de atributos, mas um conjunto de nomes "interessantes". `dir` pode inspecionar objetos implementados com ou sem um `__dict__`. O próprio atributo `__dict__` não é listado por `dir`, mas as chaves de `__dict__` são listadas. Vários atributos especiais de classes, como `__mro__`, `__bases__` e `__name__`, também não são listados por `dir`. Você pode customizar o comportamento de `dir` implementando o método especial `__dir__`, como vimos no Exemplo 4. Se o argumento opcional `object` não for passado, `dir` lista os nomes definidos no escopo atual.

`getattr(object, name[, default])`

Devolve o atributo identificado pela string `name` no `object`. O principal caso de uso é obter atributos (ou métodos) cujos nomes são conhecidos só durante a execução do programa. Esta função pode recuperar um atributo da classe do objeto ou de uma superclasse. Se tal atributo não existir, `getattr` gera uma `AttributeError` ou devolve o valor `default`, se este argumento for passado. Um ótimo exemplo de uso de `getattr` aparece no método `Cmd.onecmd` [fpy.li/22-19], no pacote `cmd` da biblioteca padrão, onde ela é usada para obter e executar um comando definido pelo usuário.

`hasattr(object, name)`

Devolve `True` se o atributo nomeado existir em `object`, ou puder ser obtido de alguma forma através dele (por herança, por exemplo). A «documentação» [fpy.li/bk] explica: "Isto é implementado chamando `getattr(object, name)` e vendo se levanta um `AttributeError` ou não."

`setattr(object, name, value)`

Atribui o `value` ao atributo denominado `name` do `object`, se o `object` permitir essa operação. Isto pode criar um novo atributo ou sobrescrever um atributo existente.

`vars([object])`

Devolve o `__dict__` de `object`; `vars` não funciona com instâncias de classes que definem `__slots__` e não têm um `__dict__` (compare com `dir`, que aceita essas instâncias). Sem argumentos, `vars()` faz o mesmo que `locals()`: devolve um `dict` representando o escopo local.

22.8.3. Métodos especiais para tratamento de atributos

Quando implementados em uma classe definida pelo usuário, os métodos especiais listados abaixo controlam a recuperação, a atualização, a exclusão e a listagem de atributos.

Acessos a atributos, usando a notação de ponto ou as funções embutidas `getattr`, `hasattr` e `setattr` disparam os métodos especiais correspondentes, listados aqui. A leitura e escrita de atributos como chaves no `__dict__` da instância não dispara esses métodos especiais—e esta é a forma habitual de evitá-los quando necessário.

A seção "3.3.11. Pesquisa de método especial" [fpy.li/bm] do capítulo "Modelo de dados" adverte:

Para classes customizadas, as invocações implícitas de métodos especiais só têm garantia de funcionar corretamente se definidas em um tipo de objeto [a classe], não no dicionário de instância do objeto.

Em outras palavras, assuma que os métodos especiais serão acessados na própria classe, mesmo quando o alvo da invocação é uma instância. Por esta razão, métodos especiais não são ocultados por atributos de instância de mesmo nome.

Nos exemplos a seguir, assuma que há uma classe chamada `Class`, que `obj` é uma instância de `Class`, e que `atrib` é um atributo de `obj`.

Para cada um destes métodos especiais, não importa se o acesso ao atributo é feito usando a notação de ponto ou uma das funções embutidas listadas na Seção 22.8.2. Por exemplo, tanto `obj.atrib` quanto `getattr(obj, 'atrib')` disparam `Class.__getattr__`(`obj`, 'atrib').

`__delattr__(self, name)`

Sempre invocado quando ocorre uma tentativa de excluir um atributo usando a instrução `del`; por exemplo, `del obj.trib` dispara `Class.__delattr__(obj, 'trib')`. Se `trib` for uma propriedade, seu método de exclusão nunca será invocado se a classe implementar `__delattr__`.

`__dir__(self)`

Invocado quando `dir` é invocado sobre um objeto, para fornecer uma lista de atributos; por exemplo, `dir(obj)` dispara `Class.__dir__(obj)`. Também é usado pelo recurso de auto-completar em todos os consoles modernos de Python.

`__getattr__(self, name)`

Invocado apenas quando uma tentativa de obter o atributo nomeado falha, após `obj`, `Class` e suas superclasses serem pesquisadas. As expressões `obj.ausente`, `getattr(obj, 'ausente')` e `hasattr(obj, 'ausente')` podem disparar `Class.__getattr__(obj, 'ausente')`, mas apenas se um atributo com o nome 'ausente' não for encontrado em `obj` ou em `Class` e suas superclasses.

`__getattribute__(self, name)`

Sempre invocado quando há uma tentativa de obter o atributo nomeado diretamente a partir de código Python (o interpretador pode ignorar isso em alguns casos, por exemplo para obter o método `__repr__`). A notação de ponto e as funções embutidas `getattr` e `hasattr` disparam este método. O método `__getattr__` só é invocado após `__getattribute__`, e apenas quando este gera um `AttributeError`. Para acessar atributos da instância `obj` sem entrar em uma recursão infinita, implementações de `__getattribute__` devem usar `super().__getattribute__(obj, name)`.

`__setattr__(self, name, value)`

Sempre invocado quando há uma tentativa de atribuir um valor ao atributo nomeado. A notação de ponto e a função embutida `setattr` disparam esse método; por exemplo, tanto `obj.trib = 42` quanto `setattr(obj, 'trib', 42)` disparam `Class.__setattr__(obj, 'trib', 42)`.



Na prática, por serem invocados incondicionalmente, afetando praticamente todos os acessos a atributos, os métodos especiais `__getattribute__` e `__setattr__` são mais difíceis de usar corretamente que `__getattr__`, que só trata acessos a atributos que não existem. Usar propriedades ou descritores costuma ser mais fácil que definir `__getattribute__` ou `__setattr__`.

Isso conclui nosso mergulho nas propriedades, nos métodos especiais e nas outras técnicas de programação de atributos dinâmicos.

22.9. Resumo do capítulo

Começamos nossa discussão dos atributos dinâmicos mostrando exemplos práticos de classes simples para facilitar a exploração de um conjunto de dados em JSON. O primeiro exemplo foi a classe `FrozenJSON`, que converte listas e dicts aninhados em instâncias aninhadas de `FrozenJSON`, e em listas de instâncias da mesma classe. O código de `FrozenJSON` demonstrou o uso do método especial `__getattr__` para converter estruturas de dados sob demanda, sempre que seus atributos eram lidos. A última versão de `FrozenJSON` mostrou o uso do método construtor `__new__` para transformar uma classe em uma fábrica flexível de objetos, não restrita a instâncias de si mesma.

Convertemos então o conjunto de dados JSON em um dict que armazena instâncias da classe `Record`. A primeira versão de `Record` tinha apenas algumas linhas e apresentou o truque de usar `self.__dict__.update(**kwargs)` para criar atributos arbitrários a partir de argumentos nomeados passados para `__init__`. A segunda passagem acrescentou a classe `Event`, implementando a recuperação automática de registros relacionados através de propriedades.

Propriedades que devolvem valores calculados algumas vezes necessitam de *caching*, e falamos de algumas formas de fazer isso. Após descobrir que `@functools.cached_property` não é sempre aplicável, aprendemos sobre uma alternativa: a combinação de `@property` acima de `@functools.cache`, nesta ordem.

A discussão sobre propriedades continuou com a classe `LineItem`, onde criamos uma propriedade para evitar que um atributo `weight` receba valores negativos ou zero, que não fazem sentido na lógica do negócio. Após um aprofundamento da sintaxe e da semântica das propriedades, criamos uma fábrica de propriedades

para aplicar a mesma validação a `weight` e a `price`, sem precisar escrever múltiplos *getters* e *setters*. A fábrica de propriedades se apoiou em conceitos sutis—como clausuras e a sobreposição de atributos de instância por propriedades—para fornecer uma solução genérica elegante, usando para isso o mesmo número de linhas que usamos antes para escrever manualmente a definição de uma única propriedade.

Por fim, demos uma rápida passada pelo tratamento da exclusão de atributos com propriedades, seguida por um resumo dos principais atributos especiais, funções embutidas e métodos especiais que suportam a metaprogramação de atributos no núcleo da linguagem Python.

22.10. Leitura Complementar

A documentação oficial para as funções embutidas de tratamento de atributos e introspecção é o capítulo «Funções embutidas» [fpy.li/bn] da *Biblioteca Padrão de Python*. Os métodos especiais relacionados e o atributo especial `__slots__` estão documentados em *A Referência da Linguagem Python*, em «Personalizando o acesso aos atributos» [fpy.li/br]. A semântica de como métodos especiais são invocados ignorando as instâncias está explicada em «Pesquisa de método especial» [fpy.li/bs]. A seção «Atributos especiais» [fpy.li/bt] trata dos atributos `__class__` e `__dict__`.

O *Python Cookbook*, 3rd. ed. [fpy.li/pycook3], de David Beazley e Brian K. Jones (O'Reilly), tem várias receitas relacionadas aos tópicos deste capítulo, mas eu destaco três delas: A *Recipe 8.8. Extending a Property in a Subclass* (Estendendo uma Propriedade em uma Subclasse) trata da espinhosa questão de sobrescrever métodos dentro de uma propriedade herdada de uma superclasse; a *Recipe 8.15. Delegating Attribute Access* (Delegando o Acesso a Atributos) implementa uma classe de fachada, demonstrando a maioria dos métodos especiais da Seção 22.8.3 deste livro; e a *Recipe 9.21. Avoiding Repetitive Property Methods* (Evitando Métodos de Propriedade Repetitivos), que foi a base da função fábrica de propriedades que apresentei no Exemplo 28.

O *Python in a Nutshell*, 3rd. ed. [fpy.li/pynut3], de Alex Martelli, Anna Ravenscroft e Steve Holden (O'Reilly), é rigoroso e objetivo. Eles dedicam apenas três páginas a propriedades, mas isto foi possível porque o livro segue um estilo de apresentação axiomático: as 15 ou 16 páginas precedentes fornecem uma

descrição minuciosa da semântica das classes de Python, a partir do zero, incluindo descritores, que são como as propriedades são implementadas debaixo dos panos. Assim, quando Martelli et al. chegam às propriedades, eles concentram várias ideias profundas naquelas três páginas—incluindo o trecho que selecionei para abrir este capítulo.

Bertrand Meyer—citado na definição do Princípio do Acesso Uniforme no início do capítulo—foi um pioneiro da metodologia *Design by Contract* [Projeto por Contrato], projetou a linguagem Eiffel, e escreveu o excelente *Object-Oriented Software Construction*, 2ª ed. (Pearson). Os primeiros seis capítulos são uma das melhores introduções conceituais à análise e design orientados a objetos que tenho notícia. O capítulo 11 apresenta a programação por contrato, e o capítulo 35 traz as avaliações de Meyer de algumas das mais influentes linguagens orientadas a objetos: Simula, Smalltalk, CLOS (the Common Lisp Object System), Objective-C, C++, e Java, com comentários curtos sobre algumas outras. Spoiler: na última página do livro o autor revela que a "notação" extremamente legível usada como pseudocódigo ao longo do livro é na verdade a linguagem Eiffel.

Ponto de Vista

O Princípio de Acesso Uniforme de Meyer é esteticamente atraente. Como um programador usando uma API, eu não deveria ter de me preocupar se `product.price` simplesmente lê um atributo de dados ou executa uma computação. Como um consumidor e um cidadão, eu me importo: no comércio online atual, o valor de `product.price` muitas vezes depende de quem está perguntando, então ele certamente não é um mero atributo de dados. Na verdade, é uma prática comum apresentar um preço mais baixo se a consulta vem de fora da loja—por exemplo, de um mecanismo de comparação de preços. Isso pune os fregueses fiéis, que gostam de explorar sua loja favorita. Mas estou divagando.

A digressão anterior toca um ponto relevante para a programação: apesar do Princípio de Acesso Uniforme fazer todo sentido em um mundo ideal, na realidade os usuários de uma API podem precisar saber se ler `product.price` é potencialmente dispendioso ou demorado demais. Isto é um problema geral das abstrações em programação: elas dificultam raciocinar sobre o custo de computar uma expressão durante a execução. Por outro lado, abstrações permitem aos usuários da linguagem fazerem mais com menos código. É uma troca. Como de hábito em questões de engenharia de software, o «wiki original» [fpy.li/22-26] de Ward Cunningham contém argumentos inteligentes sobre os prós e contras do *Uniform Access Principle* [fpy.li/22-27] (Princípio de Acesso Uniforme).

Em linguagens de programação orientadas a objetos, a aplicação ou violação do Princípio de Acesso Uniforme muitas vezes gira em torno da sintaxe de leitura de atributos de dados públicos versus a invocação de métodos *getter/setter*.

Smalltalk e Ruby resolvem esta questão de uma forma simples e elegante: elas não suportam nenhuma forma de atributos de dados públicos. Todo atributo de instância nessas linguagens é privado, então qualquer acesso a eles deve passar por métodos. Mas sua sintaxe torna isso indolor: em Ruby, `product.price` invoca o *getter* de `price`; em Smalltalk, ele é simplesmente `product price`.

Na outra ponta do espectro, a linguagem Java permite ao programador escolher entre quatro modificadores de nível de acesso—incluindo o default sem nome que o «Java Tutorial» [fpy.li/22-28] chama de "*package-private*".

A prática geral, entretanto, não concorda com a sintaxe definida pelos projetistas de Java. Todos no campo de Java concordam que atributos devem ser `private`, e é necessário escrever isso explicitamente todas as vezes, porque não é o default. Quando todos os atributos são privados, todo acesso a eles de fora da classe precisa passar por métodos de acesso. Os IDEs de Java incluem atalhos para gerar métodos de acesso automaticamente. Infelizmente, o IDE não ajuda quando você precisa ler aquele código seis meses depois. É problema seu navegar por um oceano de métodos de acesso que não fazem nada, para encontrar aqueles que adicionam valor, implementando alguma lógica do negócio.

Alex Martelli representa a maioria da comunidade Python quando chama métodos de acesso de "idiomatismos patéticos", e então apresenta exemplos que parecem muito diferentes, mas fazem a mesma coisa:^[13]

```
um_objeto.contagem += 1
# bem melhor que...
um_objeto.set_contagem(um_objeto.get_contagem() + 1)
```

Algumas vezes, ao projetar uma API, me pergunto se todo método que não recebe argumentos (além de `self`) é uma função pura (isto é, devolve um valor mas não tem efeitos colaterais) não deveria ser substituído por uma propriedade somente de leitura. Neste capítulo, o método `ListItem.subtotal` (no Exemplo 27) seria um bom candidato a se tornar uma propriedade somente para leitura. Claro, isso exclui métodos projetados para modificar o objeto, tal como `my_list.clear()`. Seria uma péssima ideia transformar isso em uma propriedade, de tal forma que o mero acesso a `my_list.clear` apagaria o conteúdo da lista!

Na biblioteca *GPIO Pingo* [fpy.li/22-29] (mencionada na «Seção 3.5.2» [fpy.li/ct], vol.1), grande parte da API do usuário está baseada em propriedades. Por exemplo, para ler o valor atual de uma porta analógica escrevemos `pin.value`, e para definir o modo de uma porta, `pin.mode = OUT`.

Por trás da cortina, ler o valor de uma porta analógica ou definir o modo de uma porta digital pode implicar em bastante código, dependendo do driver específico da placa. Decidimos usar propriedades no Pingo porque queríamos que a API fosse confortável de usar até mesmo em ambientes interativos como um Jupyter Notebook, e achamos que `pin.mode = OUT` é mais bonito e fácil de digitar que `pin.set_mode(OUT)`.

Apesar de achar a solução de Smalltalk e de Ruby mais limpa, acho que a abordagem de Python faz mais sentido que a de Java. Podemos começar simples, programando elementos de dados como atributos públicos, pois sabemos que eles sempre podem ser encapsulados por propriedades (ou descritores, dos quais falaremos no próximo capítulo).

`__new__` é melhor que `new`

Em Python a instanciação de um objeto usa mesma sintaxe que a invocação de uma função: `my_obj = spam()`, onde `spam` pode ser uma classe ou qualquer outro invocável.

Outras linguagens, influenciadas pela sintaxe do C++, têm um operador `new` que faz a instanciação parecer diferente de uma invocação. Na maior parte do tempo, o usuário de uma API não se importa se `spam` é uma função ou uma classe. Por anos tive a impressão que `property` era uma função. No uso cotidiano, não faz diferença.

Há muitas boas razões para substituir construtores por fábricas.^[14] Um motivo comum é limitar o número de instâncias, devolvendo objetos construídos anteriormente (como no padrão de projeto Singleton). Um uso relacionado é fazer *caching* de uma construção de objeto dispendiosa. Além disso, às vezes é conveniente devolver objetos de tipos diferentes, dependendo dos argumentos passados.

Programar um construtor é simples; fornecer uma fábrica aumenta a flexibilidade às custas de mais código. Em linguagens com um operador `new`, o projetista de uma API precisa decidir se vai fornecer um construtor simples ou investir em uma fábrica. Se a escolha inicial estiver errada, a correção pode ser cara—tudo porque `new` é um operador.

Às vezes pode ser conveniente pegar o caminho inverso, e substituir uma função simples por uma classe.

Em Python, classes e funções são intercambiáveis em muitas situações. Não apenas pela ausência de um operador `new` para construir instâncias, mas também porque existe o método especial `__new__`, que pode transformar uma classe em uma fábrica que produz objetos de diferentes tipos (como vimos na Seção 22.2.3), ou devolve instâncias pré-fabricadas em vez de criar uma nova instância toda vez.

Esta dualidade função-classe seria mais fácil de aproveitar se a PEP 8—Style Guide for Python Code [fpy.li/22-31] (Guia de Estilo para Código Python) não recomendasse `CamelCase` para nomes de classes. Por outro lado, dezenas de classes na biblioteca padrão têm nomes apenas em minúsculas (por exemplo, `property`, `str`, `defaultdict`, etc.). Daí que talvez o uso de nomes de classe só com minúsculas seja uma vantagem, e não um bug. Mas independente do ponto de vista, esta inconsistência no uso de maiúsculas e minúsculas nos nomes de classes na biblioteca padrão de Python é desconfortável.

Apesar da invocação de uma função não ser diferente da invocação de uma classe, é bom saber qual é qual porque podemos criar uma subclasse de uma classe, mas não de uma função. Então eu, pessoalmente, uso `CamelCase` nas classes que escrevo, e gostaria que todas as classes na biblioteca padrão de Python seguissem alguma convenção para não evitar casos como `collections.OrderedDict` e `collections.defaultdict`.

[1] Alex Martelli, Anna Ravenscroft & Steve Holden, *Python in a Nutshell*, Third Edition [fpy.li/pynut3] (O'Reilly), p. 123.

[2] Bertrand Meyer, *Object-Oriented Software Construction*, 2nd ed. (Pearson), p. 57.

[3] A OSCON—O'Reilly Open Source Conference (*Conferência O'Reilly de Código Aberto*)—foi uma vítima da pandemia de COVID-19. O arquivo JSON original de 744 KB, que usei para esses exemplos, não está mais disponível online hoje (10 de janeiro de 2021). Você pode obter uma cópia do *osconfeed.json* [fpy.li/22-1] no repositório de exemplos do livro.

[4] Dois exemplos são `AttrDict` [fpy.li/22-2] e `addict` [fpy.li/22-3].

[5] Em Java acontece a mesma coisa: a variável mágica `this` dentro de um "construtor" em Java já aponta para um objeto previamente construído e alocado na memória, e o que resta para o seu código é apenas inicializá-lo.

[6] `Bunch` ou "punhado" é o nome da classe usada por Alex Martelli para compartilhar essa dica em uma receita de 2001 intitulada "The simple but handy 'collector of a bunch of named stuff' class"

(Uma classe simples mas prática 'coletora de um punhado de coisas nomeadas') [fpy.li/22-4].

[7] Isso é, na verdade, uma desvantagem do Princípio de Acesso Uniforme de Meyer, mencionada no início deste capítulo. Quem tiver interesse nesta discussão pode ler o Ponto de Vista opcional.

[8] Fonte: documentação de `@functools.cached_property` [fpy.li/bg]. Sei que o autor dessa explicação é Raymond Hettinger porque ele a escreveu em resposta a um problema que eu reportei:

bpo42781—functools.cached_property docs should explain that it is non-overriding [fpy.li/22-11] (a documentação de `functools.cached_property` deveria explicar que ele é não-dominante). Hettinger é um grande colaborador da documentação oficial de Python e de pacotes importantes da biblioteca padrão, como o sensacional `itertools`.

[9] Veja «Mutex recursivo» [fpy.li/bq] na Wikipédia.

[10] Citação direta de Jeff Bezos no artigo do *Wall Street Journal* *Birth of a Salesman* [fpy.li/22-16] (O Nascimento de um Vendedor) publicado em 15 de outubro de 2011. Pelo menos em 2023, é necessário ser assinante para ler o artigo.

[11] Aquela cena sangrenta está disponível no Youtube [fpy.li/22-17] quando reviso essa seção, em outubro de 2021.

[12] Alex Martelli assinala que, apesar de `__slots__` poder ser definido como uma `list`, é melhor ser explícito e sempre usar uma `tuple`, pois modificar a lista em `__slots__` após o processamento do corpo da classe não tem qualquer efeito. Assim, seria equivocado usar uma sequência mutável ali.

[13] Alex Martelli, *Python in a Nutshell*, 2ª ed. (O'Reilly), p. 101.

[14] As razões que menciono foram apresentadas no artigo intitulado *Java's new Considered Harmful* [fpy.li/22-30] (new de Java considerado nocivo), de Jonathan Amsterdam, publicado na *Dr. Dobbs Journal*, e em *Consider static factory methods instead of constructors* (Considere substituir construtores por métodos fábrica estáticos), que é o Item 1 do premiado livro *Effective Java*, 3ª ed., de Joshua Bloch (Addison-Wesley).

Capítulo 23. Descritores de Atributos

Aprender sobre descritores não apenas dá acesso a um conjunto maior de ferramentas, também cria uma compreensão melhor sobre o funcionamento do Python e uma apreciação pela elegância de seu design.^[1]

— Raymond Hettinger, guru e mantenedor do Python

Descritores são uma forma de reutilizar a mesma lógica de acesso em múltiplos atributos. Por exemplo, são descritores os campos de registros em um ORM (*Object Relational Mapping*, Mapeamento Objeto-Relacional) como o *SQLAlchemy* ou ORM do *Django*. Nestes sistemas, os descritores controlam a conversão e validação de dados dos atributos de objetos Python para campos nas tabelas do banco de dados.

Um descritor é uma classe que implementa um protocolo dinâmico, composto pelos métodos `__get__`, `__set__`, e `__delete__`. A classe `property` implementa o protocolo de descritor completo. Você pode implementar apenas parte do protocolo de descritor, como ocorre com protocolos dinâmicos em geral. Na verdade, a maioria dos descritores que vemos em código real implementa apenas `__get__` e `__set__`, e muitos têm só um destes métodos.

Descritores são um recurso característico de Python, presentes não apenas no nível das aplicações, mas também na infraestrutura da linguagem. Funções definidas pelo usuário são descritores. Veremos como o protocolo do descritor permite que métodos operem como métodos vinculados ou funções, dependendo de como são acessados.

Entender os descritores é crucial para dominar Python. Este capítulo é sobre isso.

Nas próximas páginas vamos refatorar o exemplo da loja de alimentos orgânicos a granel, visto na Seção 22.4, substituindo propriedades por descritores. Isto tornará mais fácil reutilizar a lógica de validação de atributos em diferentes classes.

Vamos estudar os conceitos de descritores dominantes e não dominantes, e entender que as funções de Python são descritores. Para finalizar, veremos algumas dicas para a implementação de descritores.

23.1. Novidades neste capítulo

O exemplo do descritor `Quantity`, na Seção 23.2.3, ficou muito mais simples graças ao método especial `__set_name__`, adicionado ao protocolo de descritor no Python 3.6. Naquela mesma seção, removi o exemplo da fábrica de propriedades, pois ele se tornou irrelevante: o ponto ali era mostrar uma solução alternativa para o problema de `Quantity`, mas com `__set_name__` o exemplo ficou redundante. Removi a classe `AutoStorage`, que aparecia na Seção 23.2.4 pelo mesmo motivo.

23.2. Descritor para validação de atributos

Como vimos na Seção 22.6, uma fábrica de propriedades é uma forma de evitar código repetitivo de *getters* e *setters*, aplicando padrões de programação funcional. Uma fábrica de propriedades é uma função de ordem superior que cria um conjunto de funções de acesso parametrizadas e constrói uma instância de propriedade customizada, com clausuras para preservar configurações como `storage_name`. A forma orientada a objetos de resolver o mesmo problema é uma classe descritora.

Vamos seguir com a série de exemplos `LineItem` de onde paramos, na Seção 22.6, refatorando a fábrica de propriedades `quantity` em uma classe descritora `Quantity`. Isso vai torná-la mais fácil de usar.

23.2.1. `LineItem` versão #3: Um descritor simples

Como dito na introdução, uma classe que implemente um método `__get__`, um `__set__`, ou um `__delete__` é um descritor. Para usar um descritor, declaramos instâncias dele como atributos em outra classe.

Vamos criar um descritor `Quantity`, e a classe `LineItem` vai usar duas instâncias de `Quantity`: uma para gerenciar o atributo `weight`, a outra para `price`. Um diagrama ajuda: dê uma olhada na Figura 1.

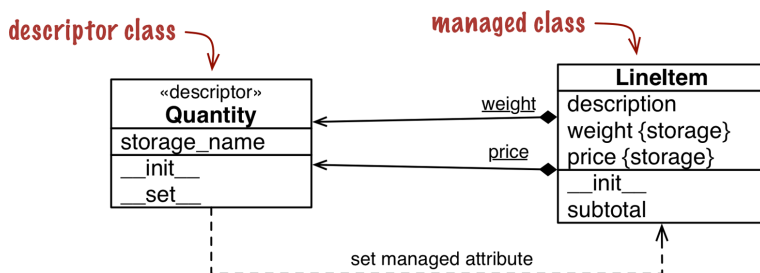


Figura 1. Diagrama de classe UML para `LineItem` usando uma classe descritora chamada `Quantity`. Atributos sublinhados no UML são atributos de classe. Observe que `weight` e `price` são instâncias de `Quantity` vinculadas à classe `LineItem` (são atributos da classe), mas cada instância de `LineItem` também tem atributos `weight` e `price`, onde estes valores são armazenados na própria instância.

Note que a palavra `weight` aparece duas vezes na Figura 1, pois na verdade há dois atributos diferentes chamados `weight`: um é um atributo de classe de `LineItem`, o outro é um atributo de instância que existirá em cada objeto `LineItem`. O mesmo se aplica a `price`.

23.2.1.1. Termos para entender descritores

Implementar e usar descritores envolve várias peças. Vou utilizar termos e definições abaixo nas descrições dos exemplos desse capítulo. Será mais fácil entendê-los após ver o código, mas quis colocar todas as definições no início, para você poder voltar aqui quando precisar.

Classe descritora (*descriptor class*)

Uma classe que implementa o protocolo de descritor. Por exemplo, `Quantity` na Figura 1.

Classe gerenciada (*managed class*)

A classe onde as instâncias do descritor são declaradas, como atributos de classe. Na Figura 1, `LineItem` é a classe gerenciada.

Instância do descritor (*descriptor instance*)

Cada instância de uma classe descritora declarada como um atributo de classe na classe gerenciada. Na Figura 1, cada instância do descritor está representada pela seta de composição com um nome sublinhado (na UML, o sublinhado indica um atributo de classe). Os diamantes pretos tocam a classe `LineItem`, que contém as instâncias do descritor.

Instância gerenciada (*managed instance*)

Uma instância da classe gerenciada. Neste exemplo, instâncias de `LineItem` são as instâncias gerenciadas (elas não aparecem no diagrama de classe).

Atributo de armazenamento (*storage attribute*)

Um atributo da instância gerenciada que guarda o valor de um atributo gerenciado para aquela instância específica. Na Figura 1, os atributos de instância `weight` e `price` de `LineItem` são atributos de armazenamento. Eles são diferentes das instâncias do descritor, que são sempre atributos de classe.

Atributo gerenciado (*managed attribute*)

Um atributo público na classe gerenciada que é controlado por uma instância de um descritor, com os valores preservados em atributos de armazenamento. Uma instância do descritor e um atributo de armazenamento fornecem a infraestrutura para um atributo gerenciado.

É importante entender que instâncias de `Quantity` são atributos de classe de `LineItem`. Este ponto fundamental é realçado pelas engenhocas (*mills*) e bugigangas (*gizmos*) na Figura 2.

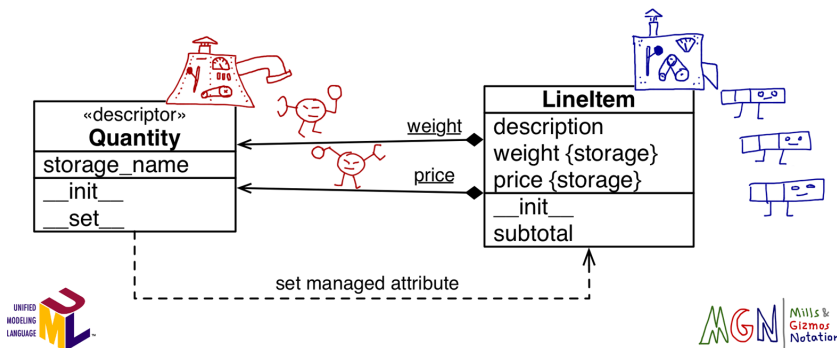


Figura 2. Diagrama de classe UML anotado com MGN (Mills & Gizmos Notation—Notação de Engenhocas e Bugigangas): classes são engenhocas que produzem bugigangas—as instâncias. A engenhoca `Quantity` produz duas bugigangas de cabeça redonda, que são anexadas à engenhoca `LineItem`: `weight` e `price`. A engenhoca `LineItem` produz bugigangas retangulares que têm seus próprios atributos `weight` e `price`, onde aqueles valores são armazenados.

Apresentando a notação Engenhocas & Bugigangas (Mills & Gizmos)

Após explicar descritores várias vezes, percebi que a UML não é muito boa para mostrar as relações entre classes e instâncias, tal como a relação entre uma classe gerenciada e as instâncias do descritor.^[2] Daí inventei minha própria "linguagem", a Notação Engenhocas e Bugigangas (MGN), que uso para anotar diagramas UML.

Desenhei a MGN para tornar mais evidente a diferença entre classes e instâncias. Veja a Figura 3. Na MGN, uma classe aparece como uma "engenhoca", (*mill*) uma máquina complicada que produz bugigangas (*gizmos*). Classes/engenhocas são máquinas com alavancas e mostradores. As bugigangas são as instâncias, e elas têm uma aparência mais simples. Quando este livro é produzido em cores, as bugigangas têm a mesma cor da engenhoca que as produziu.



Figura 3. Esboço MGN mostrando a classe `LineItem` produzindo três instâncias, e `Quantity` produzindo duas. Uma instância de `Quantity` está recuperando um valor armazenado em uma instância de `LineItem`.

Para este exemplo, desenhei instâncias de `LineItem` como linhas em uma fatura tabular, com três células representando os três atributos (`description`, `weight` e `price`). Como as instâncias de `Quantity` são descritores, elas têm uma lente de aumento para ler (*get*) os valores, e uma garra para escrever (*set*) os valores. Quando chegarmos às metaclasses, você me agradecerá por esses desenhos.

Chega de rabiscos por enquanto. Aqui está o código: o Exemplo 1 mostra a classe descritora `Quantity`, e o Exemplo 2 lista a nova classe `LineItem` usando duas instâncias de `Quantity`.

Exemplo 1. `bulkfood_v3.py`: o descritor `Quantity` não aceita valores negativos

```
class Quantity: ①

    def __init__(self, storage_name):
        self.storage_name = storage_name ②

    def __set__(self, instance, value): ③
        if value > 0:
            instance.__dict__[self.storage_name] = value ④
        else:
            msg = f'{self.storage_name} must be > 0'
            raise ValueError(msg)

    def __get__(self, instance, owner): ⑤
        return instance.__dict__[self.storage_name]
```

- ① O descritor é um recurso baseado em protocolo: não é necessário herdar de uma classe específica, basta implementar `__get__` ou `__set__`.
- ② Cada instância de `Quantity` terá um atributo `storage_name`: é o nome do atributo de armazenamento que vai preservar o valor nas instâncias gerenciadas.
- ③ O `__set__` é invocado quando ocorre uma tentativa de atribuir um valor a um atributo gerenciado. Aqui, `self` é a instância do descritor `Quantity`, (isto é, `LineItem.weight` ou `LineItem.price`), `instance` é a instância gerenciada (uma instância de `LineItem`) e `value` é o valor que está sendo atribuído.
- ④ Precisamos armazenar o valor do atributo diretamente no `__dict__`; invocar `setattr(instance, self.storage_name, value)` dispararia novamente o método `__set__`, levando a uma recursão infinita.
- ⑤ Precisamos implementar `__get__`, pois o nome do atributo gerenciado pode não ser igual ao `storage_name`. O argumento `owner` será explicado a seguir.

Implementar `__get__` é necessário porque um usuário poderia escrever isto:

```
class Casa:
    quartos = Quantity('cômodos')
```

Na classe `Casa`, o atributo gerenciado é `quartos`, mas o atributo de armazenamento é `cômodos`. Dada uma instância de `Casa` chamada `meu_lar`, acessar e modificar `meu_lar.quartos` passa pela instância do descritor `Quantity` vinculado a `quartos`, mas acessar e modificar `meu_lar.cômodos` não passa pelo descritor.

Observe que `__get__` recebe três argumentos: `self`, `instance` e `owner`. O argumento `owner` é uma referência à classe gerenciada (por exemplo, `LineItem`), e é útil se você quiser que o descritor suporte o acesso a um atributo de classe—talvez para emular o comportamento default de Python, de procurar um atributo de classe quando o nome não é encontrado na instância.

Se um atributo gerenciado como `weight`, é acessado através da classe como `LineItem.weight`, o método `__get__` do descritor recebe `None` no argumento `instance`.

Para suportar introspecção e outras técnicas de metaprogramação pelo usuário, é uma boa prática fazer `__get__` devolver a instância do descritor quando o atributo gerenciado é acessado através da classe. Para fazer isso, escreveríamos `__get__` assim:

```
def __get__(self, instance, owner):
    if instance is None:
        return self
    else:
        return instance.__dict__[self.storage_name]
```

O Exemplo 2 demonstra o uso de `Quantity` em `LineItem`.

Exemplo 2. *bulkfood_v3.py*: descritores *Quantity* gerenciam atributos em *LineItem*

```
class LineItem:
    weight = Quantity('weight') ①
    price = Quantity('price') ②

    def __init__(self, description, weight, price): ③
        self.description = description
        self.weight = weight
        self.price = price

    def subtotal(self):
        return self.weight * self.price
```

- ① A primeira instância do descritor vai gerenciar o atributo *weight*.
- ② A segunda instância do descritor vai gerenciar o atributo *price*.
- ③ O restante do corpo da classe é tão simples e limpo como o código original em *bulkfood_v1.py* (no Exemplo 19).

O código no Exemplo 2 evita a venda de trufas por \$0:^[3]

```
>>> truffle = LineItem('White truffle', 100, 0)
Traceback (most recent call last):
...
ValueError: value must be > 0
```



Ao programar os métodos `__get__` e `__set__` de um descritor, tenha em mente o significado dos argumentos `self` e `instance`: `self` é a instância do descritor, `instance` é a instância gerenciada. Descritores que gerenciam atributos de instância devem armazenar os valores nas instâncias gerenciadas. É por isso que Python fornece o argumento `instance` aos métodos do descritor.

Pode ser tentador armazenar o valor de cada atributo gerenciado no próprio do descritor, mas é um erro.

Em outras palavras, seria errado armazenar o valor no `self.__dict__`:

```
self.__dict__[self.storage_name] = value # errado
```

O correto é armazenar o valor no `instance.__dict__`:

```
instance.__dict__[self.storage_name] = value # certo
```

Para entender por que a primeira opção não funciona, pense no significado dos dois primeiros argumentos passados a `__set__`: `self` e `instance`. Aqui, `self` é a instância do descritor, que na verdade é um atributo de classe da classe gerenciada. Você pode ter milhares de instâncias de `LineItem` na memória em um dado momento, mas terá apenas duas instâncias dos descritores: os atributos de classe `LineItem.weight` e `LineItem.price`. Então, qualquer coisa armazenada nas próprias instâncias do descritor é na verdade parte de um atributo de classe de `LineItem`, e portanto é compartilhada por todas as instâncias de `LineItem`.

23.2.2. Evitando repetir o nome do atributo gerenciado

Um inconveniente do Exemplo 2 é a necessidade de repetir os nomes dos atributos quando os descritores são instanciados no corpo da classe gerenciada. Seria bom se a classe `LineItem` pudesse ser declarada assim:

```
class LineItem:
    weight = Quantity()
    price = Quantity()

    # o restante dos métodos permanece igual
```

Da forma como está escrito, o Exemplo 2 exige nomear explicitamente cada `Quantity`, algo não apenas inconveniente, mas também perigoso. Se um programador, ao copiar e colar código, se esquecer de editar os dois nomes, e terminar com uma linha como `price = Quantity('weight')`, o programa vai se comportar de forma muito errática, sobrescrevendo o valor de `weight` sempre que `price` for definido.

O problema é que o lado direito de uma atribuição é executado antes da variável existir—como vimos no «Capítulo 6» [fpy.li/6] (vol.1). A expressão `Quantity()` é avaliada para criar uma instância do descritor, e não há como o código na classe `Quantity` adivinhar o nome da variável à qual o descritor será vinculado (por exemplo, `weight` ou `price`).

Felizmente, o protocolo de descritor agora suporta o método `__set_name__`. Veremos a seguir como usá-lo.



Nomear automaticamente o atributo de armazenamento de um descritor era uma tarefa espinhosa. Na primeira edição do *Python Fluente*, dediquei várias páginas e muitas linhas de código neste capítulo e no seguinte para apresentar diferentes soluções, incluindo o uso de um decorador de classe e depois metaclasses (no Capítulo 24). Tudo isso ficou mais simples no Python 3.6.

23.2.3. `LineItem` versão #4: Nomeando atributos de armazenamento automaticamente

Para não ter que repetir o nome do atributo ao criar uma instância de descritor, vamos implementar `__set_name__`, para definir o `storage_name` de cada instância de `Quantity`. O método especial `__set_name__` foi acrescentado ao protocolo de descritor no Python 3.6.

O interpretador invoca `__set_name__` em cada descritor encontrado no corpo de uma `class`—se o descritor implementar esse método.^[4]

No Exemplo 3, a classe descritora `Quantity` não precisa de um `__init__`. Em vez disso, `__set_item__` armazena o nome do atributo de armazenamento.

Exemplo 3. `bulkfood_v4.py`: `__set_name__` define o nome para cada instância do descritor `Quantity`

```
class Quantity:

    def __set_name__(self, owner, name): ①
        self.storage_name = name         ②
```



```

def __set__(self, instance, value): ③
    if value > 0:
        instance.__dict__[self.storage_name] = value
    else:
        msg = f'{self.storage_name} must be > 0'
        raise ValueError(msg)

# no __get__ needed ④

class LineItem:
    weight = Quantity() ⑤
    price = Quantity()

    def __init__(self, description, weight, price):
        self.description = description
        self.weight = weight
        self.price = price

    def subtotal(self):
        return self.weight * self.price

```

- ① `self` é a instância do descritor (não a instância gerenciada), `owner` é a classe gerenciada e `name` é o nome do atributo de `owner` ao qual essa instância do descritor foi atribuída no corpo da classe de `owner`.
- ② Isso é o que o `__init__` fazia no Exemplo 1.
- ③ O método `__set__` aqui é exatamente igual ao do Exemplo 1.
- ④ Não é necessário implementar `__get__`, porque o nome do atributo de armazenamento é igual ao nome do atributo gerenciado. A expressão `product.price` obtém o atributo `price` diretamente da instância de `LineItem`.
- ⑤ Não é necessário passar o nome do atributo gerenciado para o construtor de `Quantity`. Esse era o objetivo dessa versão.

Olhando para o Exemplo 3, pode parecer muito código só para gerenciar um par de atributos, mas é importante perceber que agora abstraímos a lógica do descritor em uma unidade de código separada e reutilizável: a classe `Quantity`.

Normalmente não definimos um descritor no mesmo módulo em que ele é usado, mas em um módulo auxiliar separado, para ser usado por toda a aplicação—ou mesmo por muitas aplicações, se estivermos desenvolvendo uma biblioteca ou um framework.

Tendo isso em mente, o Exemplo 4 representa melhor o uso típico de um descritor.

Exemplo 4. `bulkfood_v4c.py`: uma definição mais limpa de `LineItem`; a classe descritora `Quantity` agora reside no módulo importado `model_v4c`

```
import model_v4c as model ①

class LineItem:
    weight = model.Quantity() ②
    price = model.Quantity()

    def __init__(self, description, weight, price):
        self.description = description
        self.weight = weight
        self.price = price

    def subtotal(self):
        return self.weight * self.price
```

① Importa o módulo `model_v4c`, onde `Quantity` é implementada.

② Coloca `model.Quantity` em uso.

Usuários do Django vão perceber que o Exemplo 4 se parece muito com uma definição de modelo. Isso não é uma coincidência: os campos de modelos Django são descritores.

Já que descritores são implementados como classes, podemos aproveitar a herança para reutilizar parte do código que já temos em novos descritores. É o que faremos na próxima seção.

23.2.4. LinItem versão #5: um novo tipo descritor

A loja imaginária de comida orgânica encontra um problema: de alguma forma, uma instância de um produto foi criada com uma descrição vazia, e o pedido não pode ser processado. Para prevenir isso, criaremos um novo descritor: `NonBlank`. Ao projetar `NonBlank`, percebemos que ele será muito parecido com o descritor `Quantity`, exceto pela lógica de validação.

Isto sugere uma refatoração, resultando em `Validated`, uma classe abstrata que sobrescreve um método `__set__`, invocando o método `validate`, que precisa ser implementado por subclasses.

Vamos então reescrever `Quantity` e implementar `NonBlank`, herdando de `Validated` e programando apenas os métodos `validate`.

A relação entre `Validated`, `Quantity` e `NonBlank` é uma aplicação do padrão *Template Method* (Método Gabarito), como descrito no clássico *Design Patterns*:

O Método Gabarito define um algoritmo em termos de operações abstratas que subclasses sobrescrevem para fornecer o comportamento concreto.^[5]

No Exemplo 5, `Validated.__set__` é um método gabarito e `self.validate` é a operação abstrata.

Exemplo 5. model_v5.py: the Validated ABC

```
import abc

class Validated(abc.ABC):

    def __set_name__(self, owner, name):
        self.storage_name = name

    def __set__(self, instance, value):
        value = self.validate(self.storage_name, value) ①
        instance.__dict__[self.storage_name] = value ②

    @abc.abstractmethod
    def validate(self, name, value): ③
        """return validated value or raise ValueError"""
```

- ① `__set__` delega a validação para o método `validate`...
- ② ...e então usa o `value` devolvido para atualizar o valor armazenado.
- ③ `validate` é um método abstrato; este é o método que "preenche" gabarito `__set__`, definindo parte do seu comportamento.

Alex Martelli prefere chamar este padrão de projeto *Self-Delegation* (Auto-Delegação), e concordo que é um nome mais descritivo: a primeira linha de `__set__` auto-delega para `validate`, ou seja, invoca outro método na mesma instância de uma subclasse concreta de `Validated`.^[6]

As subclasses concretas de `Validated` neste exemplo são `Quantity` e `NonBlank`:

Exemplo 6. model_v5.py: Quantity e NonBlank, subclasses concretas de Validated

```
class Quantity(Validated):
    """a number greater than zero"""

    def validate(self, name, value): ①
        if value <= 0:
            raise ValueError(f'{name} must be > 0')
        return value

class NonBlank(Validated):
    """a string with at least one non-space character"""

    def validate(self, name, value):
        value = value.strip()
        if not value: ②
            raise ValueError(f'{name} cannot be blank')
        return value ③
```

- ① Implementação exigida pelo método abstrato `Validated.validate`.
- ② Se não sobrar nada após a remoção dos espaços em branco antes e depois do valor, este é rejeitado.
- ③ Exigir que os métodos `validate` concretos devolvam o valor validado dá a eles a oportunidade de limpar, converter ou normalizar os dados recebidos. Neste caso, `value` é devolvido sem espaços iniciais ou finais.

Usuários de *model_v5.py* não precisam saber todos esses detalhes. O que importa é poder usar `Quantity` e `NonBlank` para automatizar a validação de atributos de instância. Veja a última classe `LineItem` no Exemplo 7.

Exemplo 7. `bulkfood_v5.py`: `LineItem` usando os descritores `Quantity` e `NonBlank`

```
import model_v5 as model ①

class LineItem:
    description = model.NonBlank() ②
    weight = model.Quantity()
    price = model.Quantity()

    def __init__(self, description, weight, price):
        self.description = description
        self.weight = weight
        self.price = price

    def subtotal(self):
        return self.weight * self.price
```

① Importa o módulo `model_v5`, dando a ele um nome amigável.

② Usa `model.NonBlank`. O restante do código não foi modificado.

Os exemplos de `LineItem` que vimos neste capítulo demonstram um uso típico de descritores para gerenciar atributos de dados. Descritores como `Quantity` são chamados *descritores dominantes*, pois seu método `__set__` sobrescreve (isto é, intercepta e impede) a atribuição de um atributo de instância com o mesmo nome na instância gerenciada. Entretanto, há também *descritores não dominantes*. Vamos explorar essa diferença detalhadamente na próxima seção.

23.3. Descritores dominantes ou não dominantes

Recordando, há uma importante assimetria na forma como Python lida com atributos. Em uma classe sem descritores, ler um atributo através de uma instância devolve o atributo definido na instância. Mas se tal atributo não existir na instância, um atributo de classe será obtido. Por outro lado, uma atribuição a um atributo em uma instância cria o atributo na instância, sem afetar a classe de forma alguma.

Esta assimetria também afeta descritores, criando duas grandes categorias de descritores, dependendo do método `__set__` estar ou não implementado. Se `__set__` estiver presente, a classe é um descritor dominante (*overriding descriptor*); caso contrário, ela é um descritor não dominante (*non-overriding descriptor*). Estes termos farão sentido quando examinarmos os comportamentos de descritores nos próximos exemplos.

Usei algumas funções auxiliares definidas no Exemplo 8 para observar as diferentes categorias de descritores, Sua lógica não é importante, mas elas são usadas nas invocações a `print_args` no Exemplo 9.

Exemplo 8. `descriptor_kinds.py`: funções auxiliares para formatar e exibir objetos

```
def cls_name(obj_or_cls):
    cls = type(obj_or_cls)
    if cls is type:
        cls = obj_or_cls
    return cls.__name__.split('.')[-1]

def display(obj):
    cls = type(obj)
    if cls is type:
        return f'<class {obj.__name__}>'
    elif cls in [type(None), int]:
        return repr(obj)
    else:
        return f'<{cls_name(obj)} object>'

def print_args(name, *args):
    pseudo_args = ', '.join(display(x) for x in args)
    print(f'-> {cls_name(args[0])}.__{name}__({pseudo_args})')
```

Agora vamos criar três classes de descritores, e uma classe `Managed` onde os descritores são instalados.

Exemplo 9. descriptor_kinds.py: para estudar os comportamentos de descritores

```
class Overriding: ①
    """a.k.a. data descriptor or enforced descriptor"""

    def __get__(self, instance, owner):
        print_args('get', self, instance, owner) ②

    def __set__(self, instance, value):
        print_args('set', self, instance, value)

class OverridingNoGet: ③
    """an overriding descriptor without ``__get__``"""

    def __set__(self, instance, value):
        print_args('set', self, instance, value)

class NonOverriding: ④
    """a.k.a. non-data or shadowable descriptor"""

    def __get__(self, instance, owner):
        print_args('get', self, instance, owner)

class Managed: ⑤
    over = Overriding()
    over_no_get = OverridingNoGet()
    non_over = NonOverriding()

    def spam(self): ⑥
        print(f'-> Managed.spam({display(self)})')
```

- ① Uma classe descritora dominante com `__get__` e `__set__`.
- ② A função `print_args` é chamada por todos os métodos do descritor neste exemplo.
- ③ Um descritor dominante sem um método `__get__`.
- ④ Nenhum método `__set__` aqui, então este é um descritor não dominante.

- ⑤ A classe gerenciada, usando uma instância de cada uma das classes descritoras.
- ⑥ O método `spam` está aqui para efeito de comparação, pois métodos também são descritores.

Nas próximas seções, examinaremos o comportamento de leitura e escrita de atributos na classe `Managed` e em uma de suas instâncias, passando por cada um dos diferentes descritores definidos.

23.3.1. Descritor dominante

Um descritor que implementa o método `__set__` é um *descritor dominante* pois, apesar de ser um atributo de classe, um descritor que implementa `__set__` irá sobrescrever tentativas de atribuição a atributos de instância. É assim que o Exemplo 3 foi implementado. Propriedades também são descritores dominantes: se você não fornecer uma função *setter*, o `__set__` default da classe `property` vai gerar um `AttributeError`, para sinalizar que o atributo é somente para leitura.



Contribuidores e autores da comunidade Python usam termos diferentes ao discutir esses conceitos. Adotei "descritor dominante" (*overriding descriptor*), do livro *Python in a Nutshell*. A documentação oficial de Python usa "descritor de dados" (*data descriptor*) mas "descritor dominante" destaca o comportamento especial. Descritores dominantes também são chamados "descritores forçados" (*enforced descriptors*). Sinônimos para descritores não dominantes incluem "descritores sem dados" (*nondata descriptors*, na documentação oficial em português) ou "descritores ocultáveis" (*shadowable descriptors*).

Dado o código no Exemplo 9, alguns experimentos com um descritor dominante podem ser vistos no Exemplo 10.

Exemplo 10. O comportamento de um descritor dominante

```
>>> obj = Managed() ①
>>> obj.over ②
-> Overriding.__get__(<Overriding object>, <Managed object>, <class
Managed>)
>>> Managed.over ③
-> Overriding.__get__(<Overriding object>, None, <class Managed>)
>>> obj.over = 7 ④
-> Overriding.__set__(<Overriding object>, <Managed object>, 7)
>>> obj.over ⑤
-> Overriding.__get__(<Overriding object>, <Managed object>, <class
Managed>)
>>> obj.__dict__['over'] = 8 ⑥
>>> vars(obj) ⑦
{'over': 8}
>>> obj.over ⑧
-> Overriding.__get__(<Overriding object>, <Managed object>, <class
Managed>)
```

- ① Cria o objeto `Managed`, para testes.
- ② `obj.over` aciona o método `__get__` do descritor, passando a instância gerenciada `obj` como segundo argumento.
- ③ `Managed.over` aciona o método `__get__` do descritor, passando `None` como segundo argumento (instance).
- ④ Atribuir a `obj.over` aciona o método `__set__` do descritor, passando o valor 7 como último argumento.
- ⑤ Ler `obj.over` ainda invoca o método `__get__` do descritor.
- ⑥ Contorna o descritor, definindo um valor diretamente no `obj.__dict__`.
- ⑦ Verifica se aquele valor está no `obj.__dict__`, sob a chave `over`.
- ⑧ Entretanto, mesmo com um atributo de instância chamado `over`, o descritor `Managed.over` continua interceptando tentativas de ler `obj.over`.

23.3.2. Descritor dominante sem `__get__`

Propriedades e outros descritores dominantes, como os campos de modelo do Django, implementam tanto `__set__` quanto `__get__`. Mas também é possível implementar apenas `__set__`, como vimos no Exemplo 2. Neste caso, apenas a escrita é controlada pelo descritor. Ler o descritor através de uma instância irá devolver o próprio objeto descritor, pois não há um `__get__` para tratar daquele acesso. Se um atributo de instância de mesmo nome for criado com um novo valor, através de acesso direto ao `__dict__` da instância, o método `__set__` continuará interceptando tentativas posteriores de definir aquele atributo, mas a leitura do atributo vai simplesmente devolver o novo valor na instância, em vez de devolver o objeto descritor. Em outras palavras, o atributo de instância vai ocultar o descritor, mas apenas na leitura. Veja o Exemplo 11.

Exemplo 11. Descritor dominante sem `__get__`

```
>>> obj.over_no_get ①
<__main__.OverridingNoGet object at 0x665bcc>
>>> Managed.over_no_get ②
<__main__.OverridingNoGet object at 0x665bcc>
>>> obj.over_no_get = 7 ③
-> OverridingNoGet.__set__(<OverridingNoGet object>, <Managed object>, 7)
>>> obj.over_no_get ④
<__main__.OverridingNoGet object at 0x665bcc>
>>> obj.__dict__['over_no_get'] = 9 ⑤
>>> obj.over_no_get ⑥
9
>>> obj.over_no_get = 7 ⑦
-> OverridingNoGet.__set__(<OverridingNoGet object>, <Managed object>, 7)
>>> obj.over_no_get ⑧
9
```

- ① Este descritor dominante não tem um método `__get__`, então ler `obj.over_no_get` obtém a instância do descritor a partir da classe.
- ② A mesma coisa acontece se obtivermos a instância do descritor diretamente da classe gerenciada.
- ③ Tentar definir um valor para `obj.over_no_get` invoca o método `__set__` do descritor.

- ④ Como nosso `__set__` não faz modificações, ler `obj.over_no_get` obtém a instância do descritor na classe gerenciada.
- ⑤ Definindo um atributo de instância chamado `over_no_get` direto no `__dict__` da instância.
- ⑥ Agora aquele atributo de instância `over_no_get` oculta o descritor, mas só na leitura.
- ⑦ Tentar atribuir um valor a `obj.over_no_get` continua passando pelo `__set__` do descritor.
- ⑧ Mas na leitura, aquele descritor é ocultado enquanto existir um atributo de instância de mesmo nome.

23.3.3. Descritor não dominante

Um descritor que não implementa `__set__` é um descritor não dominante. Definir um atributo de instância com o mesmo nome vai ocultar o descritor, tornando-o incapaz de tratar aquele atributo naquela instância específica. Métodos e a `@functools.cached_property` são implementados como descritores não dominantes. O Exemplo 12 mostra o comportamento de um descritor não dominante.

Exemplo 12. Comportamento de um descritor não dominante

```
>>> obj = Managed()
>>> obj.non_over ①
-> NonOverriding.__get__(<NonOverriding object>, <Managed object>, <class
Managed>)
>>> obj.non_over = 7 ②
>>> obj.non_over ③
7
>>> Managed.non_over ④
-> NonOverriding.__get__(<NonOverriding object>, None, <class Managed>)
>>> del obj.non_over ⑤
>>> obj.non_over ⑥
-> NonOverriding.__get__(<NonOverriding object>, <Managed object>, <class
Managed>)
```

- ① `obj.non_over` aciona o método `__get__` do descritor, passando `obj` como segundo argumento.

- ② `Managed.non_over` é um descritor não dominante, então não há um `__set__` para interferir com essa atribuição.
- ③ O `obj` agora tem um atributo de instância chamado `non_over`, que oculta o atributo do descritor de mesmo nome na classe `Managed`.
- ④ O descritor `Managed.non_over` ainda está lá, e intercepta esse acesso através da classe.
- ⑤ Se o atributo de instância `non_over` for excluído...
- ⑥ ...então ler `obj.non_over` encontra o método `__get__` do descritor; mas observe que o segundo argumento é a instância gerenciada.

Nos exemplos anteriores, vimos várias atribuições a um atributo de instância com nome igual ao do descritor, com resultados diferentes dependendo da presença ou não de um método `__set__` no descritor.

A definição de atributos na classe não pode ser controlada por descritores ligados à mesma classe. Em especial, isso significa que os próprios atributos do descritor podem ser danificados por atribuições à classe, como explicado na próxima seção.

23.3.4. Sobrescrevendo um descritor em uma classe

Independente de o descritor ser dominante ou não, ele pode ser sobrescrito por uma atribuição à classe. Isso é uma técnica de *monkey-patching* mas, no Exemplo 13, os descritores são substituídos por números inteiros, algo que certamente quebraria a lógica de qualquer classe que dependesse dos descritores para seu funcionamento correto.

Exemplo 13. Qualquer descritor pode ser sobrescrito na própria classe

```
>>> obj = Managed() ①
>>> Managed.over = 1 ②
>>> Managed.over_no_get = 2
>>> Managed.non_over = 3
>>> obj.over, obj.over_no_get, obj.non_over ③
(1, 2, 3)
```

- ① Cria uma nova instância para testes posteriores.

- ② Sobrescreve os atributos dos descritores na classe.
- ③ Os descritores realmente desapareceram.

O Exemplo 13 expõe outra assimetria entre a leitura e a escrita de atributos: apesar da leitura de um atributo de classe poder ser controlada por um `__get__` de um descritor ligado à classe gerenciada, a escrita em um atributo de classe não pode ser tratada por um `__set__` de um descritor ligado à mesma classe.



Para controlar a escrita a atributos em uma classe, é preciso associar descritores à classe da classe—em outras palavras, à metaclasses. Por default, a metaclasses de classes definidas pelo usuário é `type`, e não podemos acrescentar atributos a `type`. Mas, no Capítulo 24, vamos criar nossas próprias metaclasses.

Vamos ver agora como descritores são usados para implementar métodos no Python.

23.4. Métodos são descritores

Uma função dentro de uma classe se torna um método vinculado (*bound method*) quando invocada em uma instância, porque todas as funções definidas pelo usuário têm um método `__get__`, e portanto operam como descritores quando associados a uma classe. O Exemplo 14 demonstra a leitura do método `spam`, da classe `Managed`, apresentada no Exemplo 9.

Exemplo 14. Um método é um descritor não dominante

```
>>> obj = Managed()
>>> obj.spam ①
<bound method Managed.spam of <descriptor kinds.Managed object at
0x74c80c>>
>>> Managed.spam ②
<function Managed.spam at 0x734734>
>>> obj.spam = 7 ③
>>> obj.spam
7
```

- ① Ler de `obj.spam` obtém um objeto método vinculado.

- ② Mas ler de `Managed.spam` obtém uma função.
- ③ Atribuir um valor a `obj.spam` oculta o atributo de classe, tornando o método `spam` inacessível a partir da instância `obj`.

Funções não implementam `__set__`, portanto não são descritores dominantes, como mostra a última linha do Exemplo 14.

A outra lição fundamental do Exemplo 14 é que `obj.spam` e `Managed.spam` devolvem objetos diferentes. Como de hábito com descritores, o `__get__` de uma função devolve uma referência para a própria função quando o acesso ocorre através da classe gerenciada. Mas quando o acesso vem através da instância, o `__get__` da função devolve um método vinculado: um invocável que embrulha a função e vincula a instância gerenciada (no exemplo, `obj`) ao primeiro argumento da função (isto é, `self`), como faz a função `functools.partial`, que vimos na «Seção 7.8.2» [fpy.li/cv] (vol.1). Para um entendimento mais profundo deste mecanismo, dê uma olhada no Exemplo 15.

Exemplo 15. `method_is_descriptor.py`: uma classe `Text`, derivada de `UserString`

```
import collections

class Text(collections.UserString):

    def __repr__(self):
        return f'Text({self.data!r})'

    def reverse(self):
        return self[::-1]
```

Vamos então investigar o método `Text.reverse`. Veja o Exemplo 16.

Exemplo 16. Experimentos com um método

```
>>> word = Text('forward')
>>> word ①
Text('forward')
>>> word.reverse() ②
Text('drawrof')
>>> Text.reverse(Text('backward')) ③
```

```

Text('drawkcab')
>>> type(Text.reverse), type(word.reverse) ④
(<class 'function'>, <class 'method'>)
>>> [Text.reverse(x) for x in ['abc', (1, 2), Text('stressed')]] ⑤
['cba', (2, 1), Text('desserts')]
>>> Text.reverse.__get__(word) ⑥
<bound method Text.reverse of Text('forward')>
>>> Text.reverse.__get__(None, Text) ⑦
<function Text.reverse at 0x...>
>>> word.reverse ⑧
<bound method Text.reverse of Text('forward')>
>>> word.reverse.__self__ ⑨
Text('forward')
>>> word.reverse.__func__ is Text.reverse ⑩
True

```

- ① Seguindo a convenção, o repr de uma instância de Text parece uma chamada ao construtor de Text que criaria uma instância igual.
- ② O método reverse devolve o texto escrito de trás para frente.
- ③ Um método invocado na classe funciona como uma função.
- ④ Observe os tipos diferentes: function e method.
- ⑤ Text.reverse opera como uma função, mesmo ao trabalhar com objetos que não são instâncias de Text.
- ⑥ Toda função é um descritor não dominante. Invocar seu __get__ com uma instância obtém um método vinculado àquela instância.
- ⑦ Invocar o __get__ da função com None como argumento instance obtém a própria função.
- ⑧ A expressão word.reverse invoca Text.reverse.__get__(word), devolvendo o método vinculado, mas sem invocá-lo.
- ⑨ O objeto método vinculado tem um atributo __self__, contendo uma referência à instância na qual o método foi invocado.
- ⑩ O atributo __func__ do método vinculado é uma referência à função original, implementada na classe gerenciada.

O método vinculado contém um método `__call__`, que trata a invocação em si. Este método chama a função original, referenciada em `__func__`, passando o atributo `__self__` do método como primeiro argumento. É assim que funciona a vinculação automática do argumento `self` convencional.

A conversão de funções em métodos vinculados é um exemplo perfeito de como descritores são usados na infraestrutura da linguagem.

Após este mergulho profundo no funcionamento de descritores e métodos, vamos repassar alguns conselhos práticos sobre seu uso.

23.5. Dicas para usar descritores

A lista a seguir trata de algumas consequências práticas das características dos descritores descritas acima:

Use `property` para não complicar demais

A classe embutida `property` cria descritores dominantes, implementando `__set__` e `__get__`, mesmo se um método *setter* não for definido.^[7] O `__set__` default de uma propriedade gera um `AttributeError: can't set attribute` (não é permitido setar o atributo), então uma propriedade é a forma mais fácil de criar um atributo somente para leitura, evitando o problema descrito a seguir.

Descritores somente para leitura exigem um `__set__`

Se você usar uma classe descritora para implementar um atributo somente para leitura, precisa lembrar de programar tanto `__get__` quanto `__set__`. Caso contrário, você terá um descritor não-dominante, e setar um atributo com o mesmo nome em uma instância vai ocultar o descritor. O método `__set__` de um atributo somente para leitura deve apenas levantar `AttributeError` com uma mensagem adequada.^[8]

Descritores de validação podem funcionar apenas com `__set__`

Em um descritor projetado apenas para validação, o método `__set__` deve verificar o argumento `value` recebido e, se ele for válido, atualizar o `__dict__` da instância diretamente, usando o nome da instância do descritor como chave. Assim, ler o atributo de mesmo nome a partir da instância será tão rápido quanto possível, pois não vai precisar de um `__get__`. Veja o Exemplo 3.

Um *cache* eficiente pode ser feito só com `__get__`

Implementando só o método `__get__`, você cria um descritor não dominante. Eles são úteis para executar alguma computação custosa e então armazenar o resultado, definindo um atributo com o mesmo nome na instância^[9]. O atributo de mesmo nome na instância vai ocultar o descritor, daí acessos subsequentes àquele atributo vão buscá-lo diretamente no `__dict__` da instância, sem acionar mais o `__get__` do descritor. O decorador `@functools.cached_property` constrói um descritor não dominante.

Métodos não especiais podem ser ocultados por atributos de instância

Como funções e métodos implementam apenas `__get__`, eles são descritores não dominantes. Uma atribuição simples, como `meu_obj.o_método = 7`, significa que acessos posteriores a `o_método` através daquela instância irão obter o número 7—sem afetar a classe ou outras instâncias. Isto se aplica aos métodos especiais. O interpretador só acessa métodos especiais na própria classe. Em outras palavras, `repr(x)` é executado como `x.__class__.__repr__(x)`, então um atributo `__repr__`, definido em `x`, não tem efeito em `repr(x)`. Pela mesma razão, a existência de um atributo chamado `__getattr__` em uma instância não vai subverter o algoritmo normal de acesso a atributos.

O fato de métodos poderem ser sobrescritos tão facilmente pode soar frágil e propenso a erros. Mas eu, pessoalmente, em mais de 20 anos programando em Python, nunca tive problemas com isso. Por outro lado, se você estiver criando muitos atributos dinâmicos, onde os nomes dos atributos vêm de dados que você não controla (como fizemos na parte inicial desse capítulo), então você precisa estar atenta para isso, e talvez implementar alguma filtragem ou reescrita (*escaping*) dos nomes dos atributos dinâmicos, para preservar sua sanidade.



A classe `FrozenJSON` no Exemplo 5 está a salvo de atributos de instância ocultando métodos, pois seus únicos métodos são métodos especiais e o método de classe `build`. Métodos de classe são seguros desde que sejam sempre acessados através da classe, como fiz com `FrozenJSON.build` no Exemplo 5—mais tarde substituído por `__new__` no Exemplo 6. As classes `Record` e `Event`, apresentadas na Seção 22.3, também estão a salvo: elas implementam apenas métodos especiais, métodos estáticos e propriedades. Propriedades são descritores dominantes, então não são ocultadas por atributos de instância.

Para encerrar esse capítulo, vamos falar de dois recursos que vimos com as propriedades, mas não no contexto dos descritores: documentação e o tratamento de tentativas de excluir um atributo gerenciado.

23.6. Docstrings e exclusão de descritores

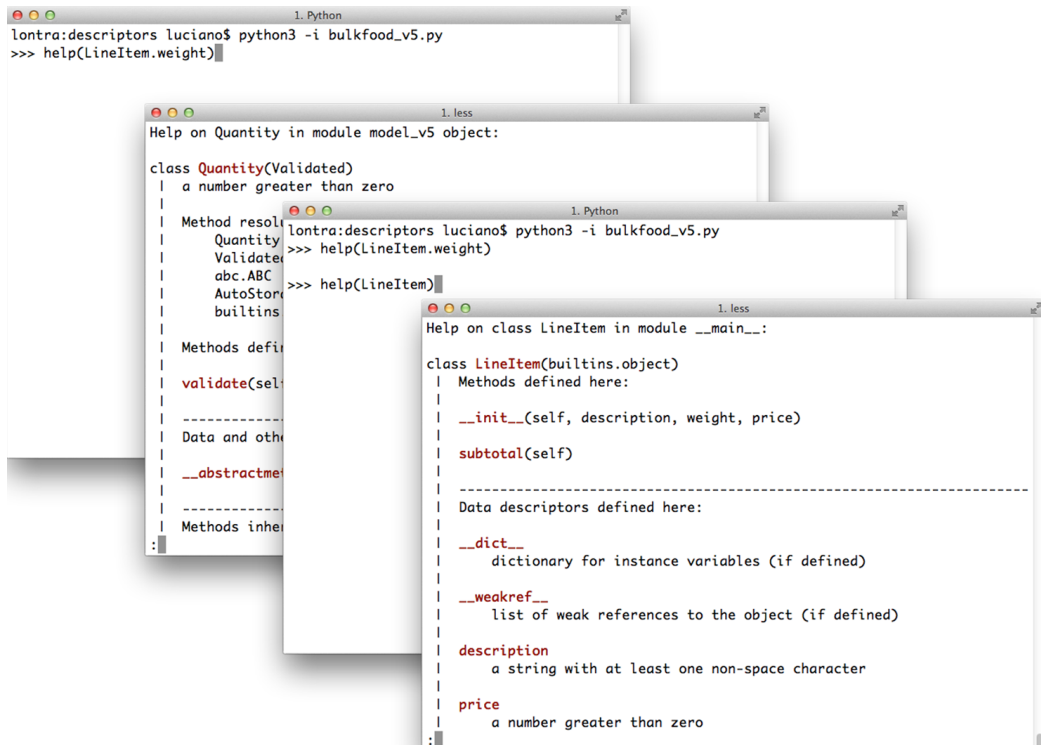


Figura 4. Capturas de tela da console de Python após os comandos `help(LineItem.weight)` e `help(LineItem)`.

A docstring de uma classe descritora é usada para documentar todas as instâncias do descritor na classe gerenciada. A Figura 4 mostra as telas de ajuda para a classe `LineItem` com os descritores `Quantity` e `NonBlank`, do Exemplo 6 e do Exemplo 7.

Isso é um tanto insatisfatório. No caso de `LineItem`, seria bom acrescentar, por exemplo, a informação de que `weight` deve ser expresso em quilogramas. Isso seria trivial com propriedades, pois cada propriedade controla um atributo gerenciado específico. Mas com descritores, a mesma classe descritora `Quantity` é usada para `weight` e `price`.^[10]

O segundo detalhe que discutimos com propriedades, mas não com descritores, é o tratamento de tentativas de apagar um atributo gerenciado. Isso pode ser feito implementando um método `__delete__` na classe descritora. Omiti deliberadamente falar de `__delete__`, porque seu uso no mundo real é raro. Se você precisar disso, por favor consulte a seção «Implementando descritores» [fpy.li/c7] na documentação do *Modelo de dados de Python*. Escrever um exemplo com uma classe descritora boba com `__delete__` fica como exercício para o leitor com excesso de tempo livre.

23.7. Resumo do capítulo

O primeiro exemplo deste capítulo foi uma continuação dos exemplos `LineItem` do Capítulo 22. No Exemplo 2, substituímos propriedades por descritores. Vimos que um descritor é uma classe que fornece instâncias instaladas como atributos na classe gerenciada. Discutir esse mecanismo exigiu uma terminologia especial, apresentando termos como *instância gerenciada* e *atributo de armazenamento*.

Na Seção 23.2.3, removemos a exigência de descritores `Quantity` serem instanciados com um `storage_name` explícito. A solução foi implementar o método especial `__set_name__` em `Quantity`, para preservar o nome da propriedade gerenciada como `self.storage_name`.

A Seção 23.2.4 mostrou como criar uma subclasse de uma classe descritora abstrata, para compartilhar código ao programar descritores especializados com alguma funcionalidade em comum.

Examinamos então os comportamentos diferentes de descritores, fornecendo ou omitindo o método `__set__`, criando uma distinção fundamental entre descritores dominantes e não dominantes. Por meio de testes detalhados, revelamos quando os descritores estão no controle, e quando são ocultados, contornados ou sobrescritos.

Em seguida, estudamos uma categoria específica de descritores não dominantes: métodos. Experimentos no console revelaram como uma função associada a uma classe se torna um método ao ser acessada através de uma instância, graças ao protocolo de descritores.

Para concluir o capítulo, a Seção 23.5 trouxe dicas práticas, e a Seção 23.6 forneceu um rápido olhar sobre como documentar descritores.



Como observado na Seção 23.1, vários exemplos deste capítulo se tornaram mais simples graças ao método especial `__set_name__` do protocolo de descritor, adicionado no Python 3.6. Isso é evolução da linguagem!

23.8. Para saber mais

Além da referência obrigatória ao capítulo «Modelo de dados» [fpy.li/2j], o «Guia de descritores» [fpy.li/bv], de Raymond Hettinger, é um recurso valioso-e parte da excelente «coleção de HOWTOS» [fpy.li/bw] na documentação oficial de Python.

Como sempre, em se tratando de assuntos relativos ao modelo de objetos de Python, o *Python in a Nutshell*, 3ª ed. (O'Reilly), de Martelli, Ravenscroft, e Holden é competente e objetivo. Martelli também fez a apresentação *Python's Object Model* (O Modelo de Objetos do Python), tratando com profundidade de propriedades e descritores: «slides» [fpy.li/23-5] e «video» [fpy.li/23-6].



Cuidado, qualquer tratamento de descritores escrito ou gravado antes da PEP 487 ser adotada, em 2016, corre o risco de conter exemplos desnecessariamente complicados hoje, pois `__set_name__` não era suportado nas versões de Python anteriores a 3.6.

Para mais exemplos práticos, o *Python Cookbook*, 3ª ed., de David Beazley e Brian K. Jones (O'Reilly), traz muitas receitas ilustrando descritores, entre as quais destaco 6.12. *Reading Nested and Variable-Sized Binary Structures* (Lendo Estruturas Binárias Aninhadas e de Tamanho Variável), 8.10. *Using Lazily Computed Properties* (Usando Propriedades Computadas de Forma Preguiçosa), 8.13. *Implementing a Data Model or Type System* (Implementando um Modelo de Dados ou um Sistema de Tipos) e 9.9. *Defining Decorators As Classes* (Definindo Decoradores como Classes). Essa última receita trata das questões profundas envolvidas na interação entre decoradores de função, descritores e métodos, e de como um decorador de função implementado como uma classe, com `__call__`, também precisa implementar `__get__` se quiser funcionar com métodos.

A PEP 487—*Simpler customization of class creation* [fpy.li/pep487] (Customização simplificada da criação de classes) introduziu o método especial `__set_name__` e inclui um exemplo de um «descriptor de validação» [fpy.li/23-7].

Ponto de vista

O design do parâmetro `self`

A exigência de declarar `self` explicitamente como o primeiro parâmetro em métodos foi uma decisão de design controversa no Python. Eu me acostumei em menos de 20 anos! Esta decisão é um exemplo de "pior é melhor" (*worse is better*): a filosofia de design descrita pelo cientista da computação Richard P. Gabriel em *The Rise of Worse is Better* [fpy.li/23-8] (A Ascensão do Pior é Melhor). A primeira prioridade dessa filosofia é "simplicidade", que Gabriel apresenta assim:

O design deve ser simples, tanto na implementação quanto na interface. É mais importante simplificar a implementação do que a interface. A simplicidade é a consideração mais importante em um design.

O `self` explícito de Python incorpora esta filosofia de design. A implementação é simples—até mesmo elegante—em prejuízo da usabilidade: uma assinatura de método como `def zfill(self, width):` não corresponde, visualmente, à invocação `label.zfill(8)`.

A linguagem Modula-3, que Guido estudou antes de inventar o Python, introduziu esta convenção com o mesmo identificador, `self`. Mas há uma diferença crucial: em Modula-3, interfaces são declaradas separadamente de sua implementação, e na declaração da interface o argumento `self` é omitido. Então, da perspectiva do usuário, um método aparece em uma declaração de interface com a mesma quantidade de parâmetros necessários para invocá-lo.

Ao longo do tempo, as mensagens de erro de Python relacionadas a argumentos de métodos se tornaram mais claras. Em um método definido pelo usuário com um argumento além de `self`, se o usuário invocasse `obj.meth()`, Python 2.7 gerava:

```
TypeError: meth() takes exactly 2 arguments (1 given)
(meth() recebe exatamente 2 argumentos (1 passado))
```

No Python 3, a confusa contagem de argumentos não é mencionada, mas o argumento ausente é nomeado:

```
TypeError: meth() missing 1 required positional argument: 'x'
(1 argumento posicional obrigatório faltando em meth(): 'x')
```

Além do uso de `self` como um argumento explícito, a exigência de qualificar cada acesso a atributos de instância com `self` também é criticada. Veja, por exemplo, o famoso post *Python Warts* (Verrugas de Python) de A. M. Kuchling («cópia arquivada» [fpy.li/23-9] no *Internet Archive*); o próprio Kuchling não se incomoda muito com o qualificador `self`, mas ele o menciona—provavelmente ecoando opiniões do grupo *comp.lang.python*. Pessoalmente não me importo em digitar o qualificador `self`: é bom para distinguir variáveis locais de atributos. Minha questão é com o uso de `self` na instrução `def`.

Quem estiver triste com o `self` explícito de Python pode se sentir bem melhor após considerar a «semântica desconcertante» [fpy.li/23-10] do `this` implícito em JavaScript. Guido teve boas razões para fazer `self` funcionar como funciona, e ele escreveu sobre elas em *Adding Support for User-Defined Classes* [fpy.li/23-11] (Adicionando Suporte a Classes Definidas pelo Usuário), em seu blog, *The History of Python* (A História de Python).

[1] Raymond Hettinger, «HowTo - Guia de descritores» [fpy.li/bv].

[2] Classes e instâncias são representadas por retângulos em diagramas de classe UML. Há diferenças visuais, mas instâncias raramente aparecem em diagramas de classe, então desenvolvedores podem não reconhecê-las como tal.

[3] O quilo de trufas brancas custa milhares de reais. Impedir a venda de trufas por \$0,01 fica como exercício para a leitora com espírito de aventura. Conheço o caso real de uma pessoa que comprou uma enciclopédia sobre estatística de 1.800 dólares por 18 dólares, devido a um erro em uma loja online (neste caso não foi na *Amazon.com*).

[4] Mais precisamente, `__set_name__` é invocado por `type.__new__`—o construtor de objetos que representam classes. A classe embutida `type` é na verdade uma metaclasses, a classe default de classes definidas pelo usuário. Isso é um pouco difícil de entender de início, mas fique tranquila: o Capítulo 24 é dedicado à configuração dinâmica de classes, incluindo o conceito de metaclasses.

- [5] Gamma et al., *Design Patterns: Elements of Reusable Object-Oriented Software*, p. 326.
- [6] Slide #50 da palestra *Python Design Patterns* [fpy.li/23-1], de Alex Martelli. Altamente recomendada.
- [7] Um método `__delete__` também é fornecido pelo decorador `property`, mesmo se você não definir um método *deleter* (de exclusão).
- [8] Python não é consistente nestas mensagens. Tentar modificar o atributo `c.real` de um número `complex` resulta em um `AttributeError: readonly attribute` (atributo somente para leitura), mas uma tentativa de mudar `c.conjugate` (um método de `complex`) levanta um `AttributeError: 'complex' object attribute 'conjugate' is read-only` (o atributo 'conjugate' do objeto 'complex' é somente para leitura). Até "read-only" está escrito de maneira diferente nas mensagens em inglês).
- [9] Entretanto, lembre-se de que criar atributos de instância após o método `__init__` frustra a otimização de memória através de compartilhamento de chaves, como discutido na «Seção 3.9» [fpy.li/82] (vol.1).
- [10] Customizar o texto de ajuda para cada instância do descritor é surpreendentemente difícil. Uma solução exige criar dinamicamente uma classe invólucro (*wrapper*) para cada instância do descritor.

Capítulo 24. Metaprogramação de classes

Todo mundo sabe que depurar um programa é duas vezes mais difícil que escrever o mesmo programa. Mas daí, se você der tudo de si ao escrever o programa, como vai conseguir depurá-lo?^[1]

— Brian W. Kernighan e P. J. Plauger, *The Elements of Programming Style*

Metaprogramação de classes é a arte de criar ou customizar classes durante a execução do programa. Em Python, classes são objetos de primeira classe, então uma função pode criar uma nova classe a qualquer momento, sem usar a palavra-chave `class`, e sem manipular código-fonte ou bytecodes.

Decoradores de classes também são funções, mas servem para inspecionar, modificar ou até substituir a classe decorada por outra classe. Por fim, metaclasses são a ferramenta mais avançada para metaprogramação de classes: elas permitem a criação de categorias de classes inteiramente novas, com características especiais, como as classes base abstratas, que já vimos antes.

Metaclasses são poderosas, mas difíceis de justificar na prática, e ainda mais difíceis de entender direito. Decoradores de classe resolvem muitos dos mesmos problemas, e são mais fáceis de compreender. Além disso, Python 3.6 implementou a *PEP 487—Simpler customization of class creation* [fpy.li/pep487] (Customização simplificada da criação de classes), fornecendo métodos especiais para tarefas que antes exigiam metaclasses ou decoradores de classe.^[2]

Este capítulo apresenta as técnicas de metaprogramação de classes em ordem ascendente de complexidade.



Esse é um tópico empolgante, e é fácil se deixar levar pelo entusiasmo. Então preciso deixar aqui esse conselho.

Em nome da legibilidade e facilidade de manutenção, você provavelmente deveria evitar as técnicas descritas neste capítulo em aplicações.

Por outro lado, caso queira escrever o próximo grande framework de Python, estas são suas ferramentas de trabalho.

24.1. Novidades neste capítulo

Todo o código do capítulo *Metaprogramação de Classes* da primeira edição do *Python Fluente* ainda funciona corretamente. Entretanto, alguns dos exemplos antigos podem ser bastante simplificados com recursos surgidos desde o Python 3.6.

Substituí aqueles exemplos por outros, enfatizando os novos recursos de metaprogramação ou acrescentando novos requisitos para justificar o uso de técnicas mais avançadas. Alguns destes exemplos novos se valem de dicas de tipo para fornecer fábricas de classes similares ao decorador `@dataclass` e a `typing.NamedTuple`.

A Seção 24.10 é nova, trazendo algumas considerações de alto nível sobre a aplicabilidade das metaclasses.



Algumas das melhores refatorações tratam de remover código tornado redundante por formas modernas e mais simples de resolver o mesmo problema. Isso se aplica tanto a código em produção quanto a livros.

Vamos começar revisando os atributos e métodos definidos no *Modelo de Dados* de Python para todas as classes.

24.2. Classes como objetos

Como acontece com a maioria dos elementos da linguagem Python, classes também são objetos. Toda classe tem alguns atributos definidos no *Modelo de Dados* de Python, documentados na seção *Atributos Especiais* [fpy.li/bt] do capítulo *Tipos Embutidos* da *Biblioteca Padrão de Python*. Três destes atributos já apareceram várias vezes no livro: `__class__`, `__name__`, e `__mro__`. Outros atributos de classe padrão são:

`cls.__bases__`

A tupla de classes base da classe.

`cls.__qualname__`

O nome qualificado de uma classe ou função, que é um caminho pontuado, desde o escopo global do módulo até a definição da classe. Isso é relevante quando a classe é definida dentro de outra classe. Por exemplo, na classe `0x` [fpy.li/24-2] que ilustra um modelo na documentação do Django, há uma classe aninhada chamada `Meta`. O `__qualname__` de `Meta` é `0x.Meta`, mas seu `__name__` é apenas `Meta`. A especificação para este atributo está na *PEP 3155—Qualified name for classes and functions* [fpy.li/24-3] (Nome qualificado para classes e funções).

`cls.__subclasses__()`

Este método devolve uma lista das subclasses imediatas da classe. A implementação usa referências fracas, para evitar referências circulares entre a superclasse e suas subclasses. Cada subclasse tem referências fortes para suas superclasses em seu atributo `__bases__`. O método lista as subclasses na memória naquele momento. Subclasses em módulos ainda não importados não aparecerão no resultado.

`cls.mro()`

O interpretador invoca este método quando está criando uma classe, para obter a tupla de superclasses armazenada no atributo `__mro__` da classe. Uma metaclasses pode sobrescrever este método, para customizar a ordem de resolução de métodos da classe em construção.



Nenhum dos atributos mencionados nesta seção aparece na lista devolvida pela função `dir(...)`.

Agora, se uma classe é um objeto, o que é a classe de uma classe?

24.3. type: a fábrica de classes embutida

Normalmente pensamos em `type` como uma função que devolve a classe de um objeto, porque é isso que `type(my_object)` faz: devolve `my_object.__class__`.

Entretanto, `type` é uma classe que cria uma nova classe, quando invocada com três argumentos.

Considere esta classe simples:

```
class MyClass(MyMixin, MySuperClass):
    x = 42

    def x2(self):
        return self.x * 2
```

Usando o construtor `type`, podemos criar uma classe idêntica durante a execução, com o seguinte código:

```
MyClass = type('MyClass',
               (MyMixin, MySuperClass),
               {'x': 42, 'x2': lambda self: self.x * 2},
               )
```

Quando Python lê uma instrução `class`, invoca `type` para construir um objeto classe assim:

```
new_class = type(name, bases, cls_dict)
```

Onde os parâmetros são:

name

O identificador que aparece após a palavra-chave `class`, por exemplo, `MyClass`.

bases

A tupla de superclasses passada entre parênteses após o identificador da classe, ou (object,), caso nenhuma superclasse seja mencionada na instrução `class`.

cls_dict

Um mapeamento entre nomes de atributos e valores. Invocáveis se tornam métodos, como vimos na Seção 23.4.



O construtor `type` aceita argumentos nomeados opcionais, que são ignorados por `type`, mas são passados para `__init_subclass__`, que deve consumi-los. Vamos estudar esse método especial na Seção 24.5, mas tratarei o uso de argumentos nomeados. Para saber mais sobre isso, por favor leia a *PEP 487* [fpy.li/pep487] sobre formas modernas de customizar a criação de classes.

A classe `type` é uma *metaclass*: uma classe que cria classes. Em outras palavras, instâncias da classe `type` são classes. A biblioteca padrão contém algumas outras metaclasses, mas `type` é a default:

```
>>> type(7)
<class 'int'>
>>> type(int)
<class 'type'>
>>> type(NotFoundError)
<class 'type'>
>>> class Whatever:
...     pass
...
>>> type(Whatever)
<class 'type'>
```

Vamos criar metaclasses customizadas na Seção 24.8.

Agora, vamos usar a classe embutida `type` para criar uma função que constrói classes.

24.4. Uma função fábrica de classes

A biblioteca padrão contém uma função fábrica de classes que já apareceu várias vezes aqui: `collections.namedtuple`. No «Capítulo 5» [fpy.li/5] (vol.1) também vimos `typing.NamedTuple` e `@dataclass`. Estas fábricas de classe usam técnicas que veremos neste capítulo.

Vamos começar com uma fábrica muito simples, para classes de objetos mutáveis—a substituta mais simples possível de `@dataclass`.

Suponha que eu esteja escrevendo uma aplicação para um *pet shop*, e queira armazenar dados sobre cães como registros simples. Mas não quero escrever código padronizado como esse:

```
class Dog:
    def __init__(self, name, weight, owner):
        self.name = name
        self.weight = weight
        self.owner = owner
```

Tédio: `name, name, name, weight, weight, weight...` E ainda falta um bom `repr`:

```
>>> rex = Dog('Rex', 30, 'Bob')
>>> rex
<__main__.Dog object at 0x2865bac>
```

Inspirados por `collections.namedtuple`, vamos criar uma `record_factory`, que cria classes simples como `Dog` dinamicamente. O Exemplo 1 mostra como ela deve funcionar.

Exemplo 1. Testando `record_factory`, uma fábrica de classes simples

```
>>> Dog = record_factory('Dog', 'name weight owner') ①
>>> rex = Dog('Rex', 30, 'Bob')
>>> rex ②
Dog(name='Rex', weight=30, owner='Bob')
>>> name, weight, _ = rex ③
>>> name, weight
```

```

('Rex', 30)
>>> "{2}'s dog weighs {1}kg".format(*rex) ④
"Bob's dog weighs 30kg"
>>> rex.weight = 32 ⑤
>>> rex
Dog(name='Rex', weight=32, owner='Bob')
>>> Dog.__mro__ ⑥
(<class 'factories.Dog'>, <class 'object'>)

```

- ① A fábrica pode ser invocada como `namedtuple`: nome da classe, seguido de uma string contendo os nomes dos atributos separados por espaços.
- ② Um repr agradável.
- ③ Instâncias são iteráveis, então elas podem ser convenientemente desempacotadas em uma atribuição...
- ④ ...ou quando são passadas para funções como `format`.
- ⑤ As instâncias são mutáveis.
- ⑥ A classe recém-criada herda de `object`—não tem relação com nossa fábrica.

O código para `record_factory` está no Exemplo 2.^[3]

Exemplo 2. `record_factory.py`: uma classe fábrica simples

```

from collections.abc import Iterable, Iterator

FieldNames = str | Iterable[str] ①

def record_factory(cls_name: str, field_names: FieldNames) -> type: ②

    slots = parse_identifiers(field_names) ③

    def __init__(self, *args, **kwargs) -> None: ④
        attrs = dict(zip(self.__slots__, args))
        attrs.update(kwargs)
        for name, value in attrs.items():
            setattr(self, name, value)

    def __iter__(self) -> Iterator: ⑤
        for name in self.__slots__:
            yield getattr(self, name)

```

```

def __repr__(self): ⑥
    values = ', '.join(f'{name}={value!r}'
                        for name, value in zip(self.__slots__, self))
    cls_name = self.__class__.__name__
    return f'{cls_name}({values})'

cls_attrs = dict( ⑦
    __slots__=slots,
    __init__=__init__,
    __iter__=__iter__,
    __repr__=__repr__,
)

return type(cls_name, (object,), cls_attrs) ⑧

def parse_identifiers(names: FieldNames) -> tuple[str, ...]:
    if isinstance(names, str):
        names = names.replace(',', ' ').split() ⑨
    if not all(s.isidentifier() for s in names):
        raise ValueError('names must all be valid identifiers')
    return tuple(names)

```

- ① O usuário pode fornecer os nomes dos campos como uma string única ou como um iterável de strings.
- ② Aceita argumentos como os dois primeiros de `collections.namedtuple`; devolve `type`—isto é, uma classe.
- ③ Cria uma tupla de nomes que será o `__slots__` da nova classe.
- ④ Esta função se tornará o método `__init__` na nova classe. Ela aceita argumentos posicionais e/ou nomeados.^[4]
- ⑤ Produz os valores dos campos na ordem dada por `__slots__`.
- ⑥ Produz um repr agradável, iterando sobre `__slots__` e `self`.
- ⑦ Monta um dicionário de atributos de classe.
- ⑧ Cria e devolve a nova classe, invocando o construtor de `type`.
- ⑨ Converte `names` separados por espaços ou vírgulas em uma lista de `str`.

O Exemplo 2 é a primeira vez que vemos `type` em uma dica de tipo: indica que `record_factory` devolve uma classe.

A última linha de `record_factory` no Exemplo 2 cria uma classe cujo nome é o valor de `cls_name`, com `object` como sua única classe base imediata, e um espaço de nomes (*namespace*) carregado com `__slots__` e os métodos de instância `__init__`, `__iter__`, e `__repr__`.

Poderíamos ter dado qualquer outro nome ao atributo de classe `__slots__`, mas daí teríamos que implementar `__setattr__` para validar os nomes dos atributos em uma atribuição, porque em nossas classes similares a registros queremos que o conjunto de atributos seja sempre o mesmo e na mesma ordem. Entretanto, lembre-se de que a principal característica de `__slots__` é economizar memória quando estamos lidando com milhões de instâncias, e que usar `__slots__` traz algumas desvantagens, discutidas na «Seção 11.11» [fpy.li/28] (vol.2).



Instâncias de classes criadas por `record_factory` não são serializáveis—isto é, elas não podem ser exportadas em formato binário pela função `dump` do módulo `pickle`. Resolver este problema está além do escopo deste exemplo, cujo objetivo é mostrar a classe `type` funcionando em um caso de uso simples. Para uma solução completa, estude o código-fonte de `collections.namedtuple`; procure pela palavra "pickling".

Vamos ver agora como emular fábricas de classes mais modernas, como `typing.NamedTuple`, que recebe uma classe definida pelo usuário, escrita com a instrução `class`, e a enriquece automaticamente com mais funcionalidade.

24.5. Apresentando `__init_subclass__`

Tanto `__init_subclass__` quanto `__set_name__` foram propostos na *PEP 487—Simpler customization of class creation* [fpy.li/pep487]. Falamos pela primeira vez do método especial para descritores `__set_name__` na Seção 23.2.3. Agora vamos estudar `__init_subclass__`.

No «Capítulo 5» [fpy.li/5] (vol.1), vimos como `typing.NamedTuple` e `@dataclass` usam a sintaxe de `class` para definir atributos em uma classe, que então é enriquecida com métodos úteis, como `__init__`, `__repr__`, `__eq__`, etc.

Ambas as fábricas de classes leem as dicas de tipo na instrução `class` do usuário para enriquecer a classe. Estas dicas de tipo também permitem que checadores de tipos estáticos validem código que define ou lê aqueles atributos. Entretanto, `NamedTuple` e `@dataclass` não usam as dicas de tipo para validação de atributos durante a execução. A classe `Checked`, no próximo exemplo, faz isso.



Não é possível suportar toda dica de tipo estática concebível para checagem de tipos durante a execução. Entretanto, alguns tipos que são também classes concretas podem ser usados com `Checked`. Isto inclui tipos simples, usados com frequência para o conteúdo de campos, como `str`, `int`, `float`, e `bool`, bem como listas destes tipos.

O Exemplo 3 mostra como usar `Checked` para criar uma classe `Movie`.

Exemplo 3. `initsub/checkedlib.py`: doctest para a criação de uma subclasse `Movie` de `Checked`

```
>>> class Movie(Checked): ①
...     title: str ②
...     year: int
...     box_office: float
...
>>> movie = Movie(title='The Godfather', year=1972, box_office=137)
③
>>> movie.title
'The Godfather'
>>> movie ④
Movie(title='The Godfather', year=1972, box_office=137.0)
```

- ① `Movie` herda de `Checked`—que definiremos mais tarde, no Exemplo 5.
- ② Cada atributo é anotado com um construtor. Aqui usei tipos embutidos.
- ③ Instâncias de `Movie` devem ser criadas usando argumentos nomeados.
- ④ Em troca, temos um `__repr__` agradável.

Os construtores usados como dicas de tipo podem ser qualquer invocável que receba zero ou um argumento, e devolva um valor adequado ao tipo do campo pretendido ou rejeite o argumento, gerando um `TypeError` ou um `ValueError`.

Usar tipos embutidos para as anotações no Exemplo 3 significa que os valores devem ser aceitáveis pelo construtor do tipo. Para `int`, isso significa qualquer `x` tal que `int(x)` devolva um `int`. Para `str`, qualquer coisa serve durante a execução, pois `str(x)` funciona com qualquer `x` no Python.^[5]

Quando chamado sem argumentos, o construtor deve devolver um valor default de seu tipo.^[6]

Esse é o comportamento padrão de construtores embutidos no Python:

```
>>> int(), float(), bool(), str(), list(), dict(), set()
(0, 0.0, False, '', [], {}, set())
```

Em uma subclasse de `Checked` como `Movie`, parâmetros ausentes criam instâncias com os valores default devolvidos pelos construtores dos campos. Por exemplo:

```
>>> Movie(title='Life of Brian')
Movie(title='Life of Brian', year=0, box_office=0.0)
```

Os construtores são usados para validação durante a instanciação, e quando um atributo é definido diretamente em uma instância:

```
>>> megahit = Movie(title='Avatar', year=2009,
...                  box_office='billions')
Traceback (most recent call last):
...
TypeError: 'billions' is not compatible with box_office:float
>>> movie.year = 'MCMLXXII'
Traceback (most recent call last):
...
TypeError: 'MCMLXXII' is not compatible with year:int
```

Subclasses de Checked e a verificação estática de tipos

Em um arquivo de código-fonte `.py` contendo uma instância `movie` da classe `Movie`, como definida no Exemplo 3, o Mypy marca essa atribuição como um erro de tipo:

```
movie.year = 'MCMLXXII'
```

Entretanto, o Mypy não consegue detectar erros de tipo nesta chamada ao construtor:

```
megahit = Movie(title='Avatar', year='MMIX')
```

Isso porque `Movie` herda `Checked.__init__`, e a assinatura daquele método deve aceitar qualquer argumento nomeado, para suportar classes arbitrárias definidas pelo usuário.

Por outro lado, se você declarar um campo de uma subclasse de `Checked` com a dica de tipo `list[float]`, o Mypy pode sinalizar atribuições de listas com tipos incompatíveis, mas `Checked` vai ignorar o parâmetro de tipo e tratá-lo como igual a `list`.

Vamos ver agora a implementação de `checkedlib.py`. A primeira classe é o descritor `Field`, como mostra o Exemplo 4.

Exemplo 4. `initsub/checkedlib.py`: a classe descritora `Field`

```
from collections.abc import Callable ①
from typing import Any, NoReturn, get_type_hints

class Field:
    def __init__(self, name: str, constructor: Callable) -> None: ②
        if not callable(constructor) or constructor is type(None): ③
            raise TypeError(f'{name!r} type hint must be callable')
        self.name = name
        self.constructor = constructor
```

```
def __set__(self, instance: Any, value: Any) -> None:
    if value is ...: ④
        value = self.constructor()
    else:
        try:
            value = self.constructor(value) ⑤
        except (TypeError, ValueError) as e: ⑥
            type_name = self.constructor.__name__
            msg = (f'{value!r} is not compatible with ' +
                  f'{self.name}:{type_name}')
            raise TypeError(msg) from e
    instance.__dict__[self.name] = value ⑦
```

- ① Lembre-se, desde o Python 3.9, o tipo `Callable` para anotações é a `ABC` em `collections.abc`, e não `typing.Callable` que foi descontinuado.
- ② Essa é a dica de tipo mínima para `Callable` mínima; o parâmetro de tipo e o tipo devolvido são implicitamente `Any`.
- ③ Para checagem durante a execução, usamos o embutido `callable`.^[7] O teste contra `type(None)` é necessário porque Python entende `None` em um tipo como `NoneType`, a classe de `None` (e portanto invocável), mas esse é um construtor inútil, que apenas devolve `None`.
- ④ Se `Checked.__init__` definir `value` como `...` (o objeto embutido `Ellipsis`), invocamos o construtor sem argumentos.
- ⑤ Caso contrário, invocamos o construtor com o `value` fornecido.
- ⑥ Se `constructor` levantar uma destas exceções, levantamos um `TypeError` com uma mensagem útil, incluindo os nomes do campo e do construtor; por exemplo, `'MMIX'` não é compatível com `year:int`.
- ⑦ Se nenhuma exceção for levantada, o `value` é armazenado no `instance.__dict__`.

Em `__set__`, precisamos capturar `TypeError` e `ValueError`, pois os construtores embutidos podem levantar qualquer uma ou outra, dependendo do argumento. Por exemplo, `float(None)` levanta `TypeError`, mas `float('A')` levanta `ValueError`. Por outro lado, `float('8')` não causa qualquer erro, e devolve `8.0`. Vamos combinar que, neste exemplo simples, este é um recurso, não um bug.



Na Seção 23.2.3, vimos o conveniente método especial `__set_name__` para descritores. Não precisamos disso na classe `Field`, porque os descritores não são instanciados no código-fonte cliente; o usuário declara tipos que são construtores, como visto na classe `Movie` (no Exemplo 3). Em vez disso, as instâncias do descritor `Field` são criadas durante a execução, pelo método `Checked.__init_subclass__`, que veremos no Exemplo 5.

Vamos agora nos concentrar na classe `Checked`, que dividi em duas listagens. O Exemplo 5 mostra a parte inicial da classe, incluindo os métodos mais importantes para este exemplo. O restante dos métodos está no Exemplo 6.

Exemplo 5. `initsub/checkedlib.py`: os métodos mais importantes da classe `Checked`

```
class Checked:
    @classmethod
    def _fields(cls) -> dict[str, type]: ①
        return get_type_hints(cls)

    def __init_subclass__(subclass) -> None: ②
        super().__init_subclass__() ③
        for name, constructor in subclass._fields().items(): ④
            setattr(subclass, name, Field(name, constructor)) ⑤

    def __init__(self, **kwargs: Any) -> None:
        for name in self._fields(): ⑥
            value = kwargs.pop(name, ...) ⑦
            setattr(self, name, value) ⑧
        if kwargs: ⑨
            self.__flag_unknown_attrs(*kwargs) ⑩
```

- ① Escrevi este método de classe para isolar a chamada a `typing.get_type_hints` do resto da classe. Se precisasse suportar apenas versões de Python ≥ 3.10 , invocaria `inspect.get_annotations` em vez disso. Reveja a «Seção 15.5.1» [fpy.li/2a] (vol.2) para uma discussão dos problemas com essas funções.
- ② `__init_subclass__` é chamado sempre que uma subclasse da classe atual é definida. Ele recebe aquela nova subclasse como seu primeiro argumento—e por isso nomeei o argumento `subclass` em vez do habitual `cls`. Para mais

informações sobre isso, veja a seguir a caixa `__init_subclass__` *não é um método de classe típico*.

- ③ `super().__init_subclass__()` não é estritamente necessário, mas é serve para suportar superclasses que implementem `__init_subclass__()` na mesma árvore de herança. Veja a «Seção 14.4» [fpy.li/cw] (vol.2).
- ④ Itera sobre `name` e constructor em cada campo...
- ⑤ ...criando um atributo em subclass com aquele `name` vinculado a um descritor `Field`, parametrizado com `name` e constructor.
- ⑥ Para cada `name` nos campos da classe...
- ⑦ ...obtem o `value` correspondente de `kwargs` e o remove de `kwargs`. Usar `...` (o objeto `Ellipsis`) como default nos permite distinguir entre argumentos com valor `None` de argumentos ausentes.^[8]
- ⑧ Invocar `setattr` aciona `Checked.__setattr__` (Exemplo 6).
- ⑨ Se houver itens remanescentes em `kwargs`, seus nomes não correspondem a qualquer dos campos declarados, e `__init__` vai falhar.
- ⑩ Este erro é informado por `__flag_unknown_attrs`, listado no Exemplo 6. Ele recebe um argumento `*names` com os nomes de atributos desconhecidos. Usei um único asterisco em `*kwargs`, para passar suas chaves como uma sequência de argumentos, sem passar os valores.

`__init_subclass__` não é um método de classe típico

O primeiro argumento que Python passa para `__init_subclass__` é uma classe. Entretanto, esta não é a classe onde `__init_subclass__` está definido, mas sim uma subclasse que herda daquela classe. Este comportamento é diferente de `__new__` e de métodos de classe decorados com `@classmethod`, onde o primeiro argumento é sempre a própria classe.

Então `__init_subclass__` não é um método de classe no sentido usual, e considero enganoso nomear seu primeiro argumento `cls`. Prefiro o nome `subcls`. A «documentação de `__init_subclass__`» [fpy.li/c2] chama o argumento de `cls`, mas explica: "...chamado sempre que se cria uma subclasse da classe que o contém. `cls` é então a nova subclasse..."

Vamos examinar os métodos restantes da classe `Checked`, continuando do Exemplo 5. Observe que prefixei os nomes dos métodos `_fields` e `_asdict` com `_`, pela mesma razão pela qual isso é feito na API de `collections.namedtuple`: reduzir a possibilidade de colisões de nomes com nomes de campos definidos pelo usuário.

Exemplo 6. `initsub/checkedlib.py`: métodos restantes da classe `Checked`

```
def __setattr__(self, name: str, value: Any) -> None: ①
    if name in self._fields(): ②
        cls = self.__class__
        descriptor = getattr(cls, name)
        descriptor.__set__(self, value) ③
    else: ④
        self.__flag_unknown_attrs(name)

def __flag_unknown_attrs(self, *names: str) -> NoReturn: ⑤
    plural = 's' if len(names) > 1 else ''
    extra = ', '.join(f'{name!r}' for name in names)
    cls_name = repr(self.__class__.__name__)
    raise AttributeError(f'{cls_name} object has ' +
                        f'no attribute{plural} {extra}')

def _asdict(self) -> dict[str, Any]: ⑥
    return {
        name: getattr(self, name)
        for name, attr in self.__class__.__dict__.items()
        if isinstance(attr, Field)
    }

def __repr__(self) -> str: ⑦
    kwargs = ', '.join(
        f'{key}={value!r}' for key, value in self._asdict().items()
    )
    return f'{self.__class__.__name__}({kwargs})'
```

- ① Intercepta qualquer tentativa de definir um atributo de instância. Isto é necessário para evitar a definição de um atributo desconhecido.
- ② Se o nome do atributo é conhecido, busca o descriptor correspondente.
- ③ Normalmente não é preciso invocar o `__set__` do descriptor explicitamente.

Aqui é necessário porque `__setattr__` intercepta todas as tentativas de definir um atributo em uma instância, mesmo na presença de um descritor dominante, tal como `Field`.

- ④ Caso contrário, o atributo `name` é desconhecido, e uma exceção será levantada por `__flag_unknown_attrs`.
- ⑤ Cria uma mensagem de erro útil, listando todos os argumentos inesperados, e levanta `AttributeError`. Este é um raro exemplo do tipo especial `NoReturn`, tratado na «Seção 8.5.12» [fpy.li/8f] (vol.1).
- ⑥ Cria um `dict` a partir dos atributos de um objeto `Movie`. Eu chamaria este método de `_as_dict`, mas segui a convenção iniciada com o método `_asdict` em `collections.namedtuple`.
- ⑦ Implementar um `__repr__` agradável é a principal razão para ter `_asdict` neste exemplo.

O exemplo `Checked` mostra como tratar descritores dominantes ao implementar `__setattr__` para bloquear a definição arbitrária de atributos após a instanciação. É possível debater se vale a pena implementar `__setattr__` neste exemplo. Sem ele, definir `movie.director = 'Greta Gerwig'` funcionaria, mas o atributo `director` não seria verificado de forma alguma, não apareceria no `__repr__` nem seria incluído no `dict` devolvido por `_asdict`—ambos definidos no Exemplo 6.

Em `record_factory.py` (no Exemplo 2), solucionei essa questão usando o atributo de classe `__slots__`. Entretanto, essa solução mais simples não é viável aqui, como explicado a seguir.

24.5.1. Por que `__init_subclass__` não pode configurar `__slots__`?

O atributo `__slots__` só tem efeito quando é um dos itens do espaço de nomes da classe passado para `type.__new__`. Acrescentar `__slots__` a uma classe existente não funciona. Python invoca `__init_subclass__` apenas após a classe ser criada—neste ponto, é tarde demais para configurar `__slots__`. Um decorador de classes também não pode configurar `__slots__`, pois ele é aplicado ainda mais tarde que `__init_subclass__`. Vamos explorar essas questões de sincronia na Seção 24.7.

Para configurar `__slots__` durante a execução, nosso código precisa criar o espaço de nomes da classe a ser passado como último argumento de `type.__new__`. Para fazer isso, podemos escrever uma função fábrica de classes, como *record_factory.py*, ou optar pelo caminho radical, e implementar uma metaclasses. Veremos como configurar `__slots__` dinamicamente na Seção 24.8.

Antes da *PEP 487* [fpy.li/pep487] simplificar a customização da criação de classes com `__init_subclass__`, no Python 3.7, uma funcionalidade similar só poderia ser implementada usando um decorador de classe. Pergunte-me como.

24.6. Um decorador de classes

Um decorador de classes tem comportamento semelhante a um decorador de funções: recebe uma classe decorada como argumento, e devolve uma classe para substituir a classe decorada. Decoradores de classe normalmente devolvem a própria classe decorada, após injetar novos métodos nela. Talvez, a razão mais comum para escolher um decorador de classes, em vez do `__init_subclass__`, é evitar interferência com outros mecanismos de classes, como herança e metaclasses. Esta justificativa aparece no resumo da *PEP 557–Data Classes* [fpy.li/24-9] para explicar por que ela foi implementada como um decorador de classes.

Nesta seção vamos estudar *checkeddeco.py*, que oferece a mesma funcionalidade de *checkedlib.py*, mas usando um decorador de classe. Como sempre, começamos examinando um exemplo de uso, extraído dos doctests em *checkeddeco.py* (no Exemplo 7).

Exemplo 7. checkeddeco.py: criando uma classe Movie decorada com @checked

```
>>> @checked
... class Movie:
...     title: str
...     year: int
...     box_office: float
...
>>> movie = Movie(title='The Godfather', year=1972, box_office=137)
>>> movie.title
'The Godfather'
>>> movie
Movie(title='The Godfather', year=1972, box_office=137.0)
```

A única diferença entre o Exemplo 7 e o Exemplo 3 é a forma como a classe `Movie` é declarada: ela é decorada com `@checked` em vez de ser uma subclasse de `Checked`. Fora isso, o comportamento externo é o mesmo, incluindo a validação de tipo e a atribuição de valores default, apresentados após o Exemplo 3, na Seção 24.5.

Vamos olhar agora para a implementação de *checkeddeco.py*. As importações e a classe `Field` são as mesmas de *checkedlib.py*, listadas no Exemplo 4. Em *checkeddeco.py* não há qualquer outra classe, apenas funções.

A lógica antes implementada em `__init_subclass__` agora é parte da função `checked`—o decorador de classes listado no Exemplo 8.

Exemplo 8. checkeddeco.py: o decorador de classes

```
def checked(cls: type) -> type: ①
    for name, constructor in _fields(cls).items(): ②
        setattr(cls, name, Field(name, constructor)) ③

    cls._fields = classmethod(_fields) # type: ignore ④

    instance_methods = ( ⑤
        __init__,
        __repr__,
        __setattr__,
        _asdict,
        __flag_unknown_attrs,
    )
    for method in instance_methods: ⑥
        setattr(cls, method.__name__, method)

    return cls ⑦
```

- ① Lembre-se de que classes são instâncias de `type`. Estas dicas de tipo indicam que este é um decorador de classes: recebe uma classe e devolve uma classe.
- ② `_fields` agora é uma função de alto nível definida mais tarde no módulo (Exemplo 9).
- ③ Troca cada atributo devolvido por `_fields` por uma instância do descritor `Field` é o que `__init_subclass__` fazia no Exemplo 5. Aqui há mais trabalho a ser feito...

- ④ Cria um método de classe a partir de `_fields`, e o adiciona à classe decorada. O comentário `type: ignore` é necessário, porque o Mypy reclama que `type` não tem um atributo `_fields`.
- ⑤ Funções ao nível do módulo, que se tornarão métodos de instância da classe decorada.
- ⑥ Adiciona cada um dos `instance_methods` a `cls`.
- ⑦ Devolve a `cls` decorada, cumprindo o contrato básico de um decorador de classes.

Todas as funções no primeiro nível de *checkeddeco.py* estão prefixadas com um sublinhado, exceto o decorador `checked`. Essa convenção de nomenclatura faz sentido por duas razões:

- `checked` é parte da interface pública do módulo *checkeddeco.py*, as outras funções não.
- As funções no Exemplo 9 serão injetadas na classe decorada, e o `_` inicial reduz as chances de um conflito de nomes com atributos e métodos definidos pelo usuário na classe decorada.

O restante de *checkeddeco.py* está listado no Exemplo 9. Aquelas funções no nível do módulo contêm o mesmo código dos métodos correspondentes na classe `Checked` de *checkedlib.py*. Elas foram explicadas no Exemplo 5 e no Exemplo 6.

Observe que a função `_fields` tem dois papéis em *checkeddeco.py*. Ela é usada como uma função normal na primeira linha do decorador `checked` e será também injetada como um método de classe na classe decorada.

Exemplo 9. checkeddeco.py: os métodos que serão injetados na classe decorada

```
def _fields(cls: type) -> dict[str, type]:
    return get_type_hints(cls)

def __init__(self: Any, **kwargs: Any) -> None:
    for name in self._fields():
        value = kwargs.pop(name, ...)
        setattr(self, name, value)
    if kwargs:
        self.__flag_unknown_attrs(*kwargs)
```

```

def __setattr__(self: Any, name: str, value: Any) -> None:
    if name in self._fields():
        cls = self.__class__
        descriptor = getattr(cls, name)
        descriptor.__set__(self, value)
    else:
        self.__flag_unknown_attrs(name)

def __flag_unknown_attrs(self: Any, *names: str) -> NoReturn:
    plural = 's' if len(names) > 1 else ''
    extra = ', '.join(f'{name!r}' for name in names)
    cls_name = repr(self.__class__.__name__)
    raise AttributeError(f'{cls_name} has no attribute{plural} {extra}')

def _asdict(self: Any) -> dict[str, Any]:
    return {
        name: getattr(self, name)
        for name, attr in self.__class__.__dict__.items()
        if isinstance(attr, Field)
    }

def __repr__(self: Any) -> str:
    kwargs = ', '.join(
        f'{key}={value!r}' for key, value in self._asdict().items()
    )
    return f'{self.__class__.__name__}({kwargs})'

```

O módulo *checkeddeco.py* implementa um decorador de classes simples, mas usável. O `@dataclass` de Python faz mais. Ele suporta várias opções de configuração, acrescenta métodos à classe decorada, trata ou avisa sobre conflitos com métodos definidos pelo usuário na classe decorada, e até percorre o `__mro__` para coletar atributos definidos pelo usuário declarados em superclasses da classe decorada. O «código-fonte» [fpy.li/24-10] do pacote `dataclasses` no Python 3.9 tem mais de 1200 linhas.

Para fazer metaprogramação de classes, precisamos saber quando o interpretador Python avalia cada bloco de código durante a criação de uma classe. É disso que falaremos a seguir.

24.7. O que acontece quando: importação versus execução

Programadores Python falam de "momento da importação" (*import time*) versus "momento de execução" (*run time*), mas estes termos não têm definições precisas e há uma zona cinzenta entre eles.

No momento da importação, o interpretador:

1. Analisa o código-fonte do módulo `.py` em uma passagem, da primeira até a última linha. É aqui que um `SyntaxError` pode ocorrer.
2. Compila o *bytecode* a ser executado.
3. Executa o código no nível superior do módulo compilado.

Se existir um arquivo `.pyc` atualizado no `__pycache__` local, a análise e a compilação são omitidas, pois o *bytecode* está pronto para ser executado.

Apesar de a análise e a compilação serem definitivamente atividades de "importação", outras coisas podem acontecer durante este processo, pois quase todas as instruções no Python são executáveis, pois podem rodar código do usuário e modificar o estado do programa.

Em especial, a instrução `import` não é meramente uma declaração, como é em Java, onde ela serve para informar o compilador sobre os pacotes necessários. Em Python, `import` serve para isso, mas também carrega e executa todo o código no nível superior de um módulo, quando ele é importado para o processo Python pela primeira vez. Importações posteriores do mesmo módulo usarão um *cache*, e então o único efeito será a vinculação dos objetos importados a nomes no módulo cliente. O código executado em consequência de um `import` pode fazer qualquer coisa, incluindo ações típicas da "execução", como escrever em um arquivo de log ou conectar-se a um banco de dados.^[9] Por isso a fronteira entre a "importação" e a "execução" é difusa: `import` pode acionar todo tipo de comportamento de "execução", porque a instrução `import` e a função embutida `__import__()` podem ser usadas dentro de qualquer função normal.

Tudo isso é bastante abstrato e sutil, então vamos fazer alguns experimentos para ver o que acontece, e quando.

24.7.1. Experimentos com as etapas de avaliação

Considere um script *evaldemo.py*, que usa um decorador de classes, um descritor e uma fábrica de classes baseada em `__init_subclass__`, todos definidos em um módulo *builderlib.py*. Os módulos usados têm várias chamadas a `print`, para revelar o que acontece por baixo dos panos. Fora isso, eles não fazem nada de útil. O objetivo destes experimentos é observar a ordem na qual essas chamadas a `print` acontecem.



Aplicar um decorador de classes e uma fábrica de classes com `__init_subclass__` juntos, em uma única classe, é provavelmente um sinal de excesso de engenharia ou de desespero. Esta combinação incomum é útil nestes experimentos, para comparar em que momento um decorador de classes e `__init_subclass__` alteram a classe.

Vamos começar examinando *builderlib.py*, dividido em duas partes: o Exemplo 10 e o Exemplo 11.

Exemplo 10. builderlib.py: primeira parte do módulo

```
print('@ builderlib module start')

class Builder: ①
    print('@ Builder body')

    def __init_subclass__(cls): ②
        print(f'@ Builder.__init_subclass__({cls!r})')

        def inner_0(self): ③
            print(f'@ SuperA.__init_subclass__:inner_0({self!r})')

        cls.method_a = inner_0

    def __init__(self):
        super().__init__()
        print(f'@ Builder.__init__({self!r})')
```

```
def deco(cls): ④
    print(f'@ deco({cls!r})')

    def inner_1(self): ⑤
        print(f'@ deco:inner_1({self!r})')

    cls.method_b = inner_1
    return cls ⑥
```

- ① Essa é uma fábrica de classes para implementar...
- ② ...um método `__init_subclass__`.
- ③ Define uma função para ser adicionada à subclasse na atribuição abaixo.
- ④ Um decorador de classes.
- ⑤ Função a ser adicionada à classe decorada.
- ⑥ Devolve a classe recebida como argumento.

Continuando *builderlib.py* no Exemplo 11...

Exemplo 11. builderlib.py: a parte final do módulo

```
class Descriptor: ①
    print('@ Descriptor body')

    def __init__(self): ②
        print(f'@ Descriptor.__init__({self!r})')

    def __set_name__(self, owner, name): ③
        args = (self, owner, name)
        print(f'@ Descriptor.__set_name__({args!r})')

    def __set__(self, instance, value): ④
        args = (self, instance, value)
        print(f'@ Descriptor.__set__({args!r})')

    def __repr__(self):
        return '<Descriptor instance>'

print('@ builderlib module end')
```


- ① Uma classe descritora para demonstrar quando...
- ② ...uma instância do descritor é criada, e quando...
- ③ ...`__set_name__` será invocado durante a criação da classe owner.
- ④ Como os outros métodos, este `__set__` não faz nada, exceto exibir seus argumentos.

Se importarmos *builderlib.py* no console de Python, veremos o seguinte:

```
>>> import builderlib
@ builderlib module start
@ Builder body
@ Descriptor body
@ builderlib module end
```

Note que as linhas exibidas por *builderlib.py* têm uma `@` à esquerda.

Agora voltamos a atenção para *evaldemo.py*, que acionará métodos especiais em *builderlib.py* (no Exemplo 12).

Exemplo 12. evaldemo.py: script para experimentar com builderlib.py

```
#!/usr/bin/env python3

from builderlib import Builder, deco, Descriptor

print('# evaldemo module start')

@deco ①
class Klass(Builder): ②
    print('# Klass body')

    attr = Descriptor() ③

    def __init__(self):
        super().__init__()
        print(f'# Klass.__init__({self!r})')

    def __repr__(self):
        return '<Klass instance>'
```

```
def main(): ④
    obj = Klass()
    obj.method_a()
    obj.method_b()
    obj.attr = 999

if __name__ == '__main__':
    main()

print('# evaldemo module end')
```

- ① Aplica um decorador.
- ② Cria uma subclasse de Builder para acionar seu `__init_subclass__`.
- ③ Instancia o descritor.
- ④ Isso só será chamado se o módulo for executado como o programa principal.

As chamadas a `print` em *evaldemo.py* têm um `#` como prefixo. Se você abrir o console novamente e importar *evaldemo.py*, a saída aparece no Exemplo 13.

Exemplo 13. Experimentos de console com evaldemo.py

```
>>> import evaldemo
@ builderlib module start ①
@ Builder body
@ Descriptor body
@ builderlib module end
# evaldemo module start
# Klass body ②
@ Descriptor.__init__(<Descriptor instance>) ③
@ Descriptor.__set_name__(<Descriptor instance>,
    <class 'evaldemo.Klass'>, 'attr') ④
@ Builder.__init_subclass__(<class 'evaldemo.Klass'>) ⑤
@ deco(<class 'evaldemo.Klass'>) ⑥
# evaldemo module end
```

- ① As primeiras quatro linhas são o resultado de `from builderlib import ...`. Elas não aparecerão se você usar a mesma sessão do console do experimento anterior, pois *builderlib.py* já estará importado.

- ② Isto sinaliza que Python começou a ler o corpo de `Klass`. Neste momento o objeto classe ainda não existe.
- ③ A instância do descritor é criada e vinculada a `attr`, no espaço de nomes que Python passará para o construtor default do objeto classe: `type.__new__`.
- ④ Neste ponto, a função embutida de Python `type.__new__` já criou o objeto `Klass` e invoca `__set_name__` em cada instância das classes do descritor que oferecem aquele método, passando `Klass` como argumento `owner`.
- ⑤ `type.__new__` então chama `__init_subclass__` na superclasse de `Klass`, passando `Klass` como único argumento.
- ⑥ Quando `type.__new__` devolve o objeto classe, Python aplica o decorador. Neste exemplo, a classe devolvida por `deco` está vinculada a `Klass` no espaço de nomes do módulo

A implementação de `type.__new__` está escrita em C. O comportamento que acabei de descrever está documentado na seção «Criando o objeto classe» [fpy.li/bx], no capítulo *Modelo de Dados* da referência de Python.

Note que a função `main()` de *evaldemo.py* (Exemplo 12) rodou durante a sessão no console (Exemplo 13), portanto nenhuma instância de `Klass` foi criada. Todas as ações que vimos foram acionadas por operações no momento da importação: `importar builderlib` e `definir Klass`.

Se você executar *evaldemo.py* como um script, verá a mesma saída do Exemplo 13, com mais linhas logo antes do final. As linhas adicionais são o resultado da execução de `main()` no Exemplo 14:

Exemplo 14. Executando evaldemo.py como um programa

```
$ ./evaldemo.py
[... 9 linhas omitidas ...]
@ deco(<class '__main__.Klass'>) ①
@ Builder.__init__(<Klass instance>) ②
# Klass.__init__(<Klass instance>)
@ SuperA.__init_subclass__:inner_0(<Klass instance>) ③
@ deco:inner_1(<Klass instance>) ④
@ Descriptor.__set__(<Descriptor instance>, <Klass instance>, 999) ⑤
# evaldemo module end
```

- ① As 10 primeiras linhas—incluindo essa—são as mesma que aparecem no Exemplo 13.
- ② Saída de `super().__init__()` em `Klass.__init__`.
- ③ Saída de `obj.method_a()` em `main`; o `method_a` foi injetado por `SuperA.__init_subclass__`.
- ④ Saída de `obj.method_b()` em `main`; `method_b` foi injetado por `deco`.
- ⑤ Saída de `obj.attr = 999` em `main`.

Uma classe base com `__init_subclass__` ou um decorador de classes são ferramentas poderosas, mas elas estão limitadas a trabalhar sobre uma classe já criada por `type.__new__` por baixo dos panos. Nas raras ocasiões em que for preciso ajustar os argumentos passados a `type.__new__`, uma metaclasses é necessária. Esse é o destino final deste capítulo—e deste livro!

24.8. Introdução às metaclasses

[Metaclasses] são uma mágica tão profunda que 99% dos usuários jamais deveriam se preocupar com elas. Quem se pergunta se precisa delas, não precisa (quem realmente precisa de metaclasses sabe com certeza, e não precisa que lhe expliquem a razão).^[10]

— Tim Peters, inventor do algoritmo timsort e prolífico mantenedor de Python

Uma metaclasses é uma fábrica de classes. Diferente de `record_factory`, do Exemplo 2, uma metaclasses é escrita como uma classe. Em outras palavras, uma metaclasses é uma classe cujas instâncias são classes. A Figura 1 usa a *Notação de Engenhocas e Bugigangas* para ilustrar uma metaclasses: uma engenhoca que produz outra engenhoca.

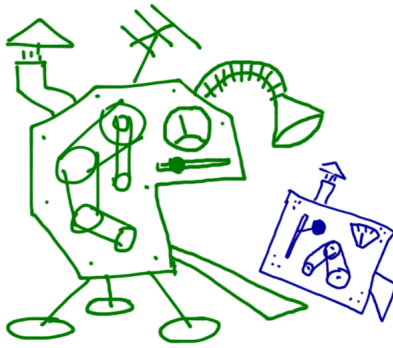


Figura 1. Uma metaclasses é uma classe que cria classes.

Pense no modelo de objetos de Python: classes são objetos, portanto cada classe deve ser uma instância de alguma outra classe. Por default, as classes de Python são instâncias de `type`. Em outras palavras, `type` é a metaclasses da maioria das classes, sejam elas embutidas ou definidas pelo usuário:

```
>>> str.__class__  
<class 'type'>  
>>> from bulkfood_v5 import LineItem  
>>> LineItem.__class__  
<class 'type'>  
>>> type.__class__  
<class 'type'>
```

Para evitar regressões infinitas, a classe de `type` é `type`, como mostra a última linha.

Observe que não estou dizendo que `str` ou `LineItem` são subclasses de `type`. Estou dizendo que `str` e `LineItem` são instâncias de `type`. Elas são todas subclasses de `object`. A Figura 2 pode ajudar você a contemplar essa estranha realidade.

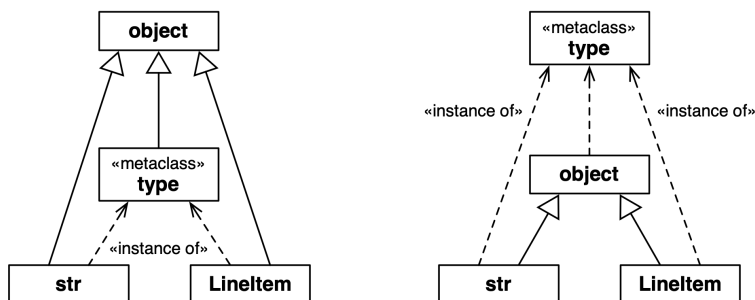


Figura 2. Os dois diagramas são verdadeiros. O da esquerda enfatiza que `str`, `type`, e `LineItem` são subclasses de `object`. O da direita ressalta que `str`, `object`, e `LineItem` são instâncias de `type`, pois todas são classes.



As classes `object` e `type` têm uma relação singular: `object` é uma instância de `type`, e `type` é uma subclasse de `object`. Esta relação é "mágica": ela não pode ser expressa em Python, porque cada uma das classes teria que existir antes da outra poder ser definida. O fato de `type` ser uma instância de si mesma também é mágico.

O próximo trecho mostra que a classe de `collections.Iterable` é `abc.ABCMeta`. Observe que `Iterable` é uma classe abstrata, mas `ABCMeta` é uma classe concreta—afinal, `Iterable` é uma instância de `ABCMeta`:

```
>>> from collections.abc import Iterable
>>> Iterable.__class__
<class 'abc.ABCMeta'>
>>> import abc
>>> from abc import ABCMeta
>>> ABCMeta.__class__
<class 'type'>
```

Por fim, a classe de `ABCMeta` também é `type`. Toda classe é uma instância de `type`, direta ou indiretamente, mas só metaclasses são também subclasses de `type`. Esta é a relação mais importante para entender as metaclasses: uma metaclasses, tal como `ABCMeta`, herda de `type` o poder de criar classes. A Figura 3 ilustra essa relação fundamental.

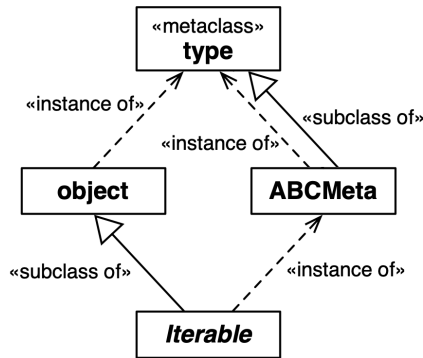


Figura 3. Iterable é uma subclasse de object e uma instância de ABCMeta. Tanto object quanto ABCMeta são instâncias de type, mas a relação crucial aqui é que ABCMeta também é uma subclasse de type, porque ABCMeta é uma metaclasses. Neste diagrama, Iterable é a única classe abstrata.

A lição importante aqui é que metaclasses são subclasses de type, e é isso que permite a elas funcionarem como fábricas de classes. Uma metaclasses pode customizar suas instâncias implementando métodos especiais, como demonstram as próximas seções.

24.8.1. Como uma metaclasses customiza uma classe

Para usar uma metaclasses, é crucial entender como `__new__` funciona em qualquer classe. Isto foi discutido na Seção 22.2.3.

A mesma mecânica se repete no nível "meta", quando uma metaclasses está prestes a criar uma nova instância, que é uma classe. Considere a declaração abaixo:

```
class Klass(SuperKlass, metaclass=MetaKlass):
    x = 42
    def __init__(self, y):
        self.y = y
```

Para processar o bloco `class` acima, o interpretador invoca `MetaKlass.__new__`, cuja implementação herdada da classe `type` recebe os seguintes argumentos:

meta_cls

A própria metaclasses, porque `__new__` funciona como um método de classe.

Ex: `MetaKlass`

cls_name

O nome da classe a ser criada, como uma string. Ex: `'Klass'`

bases

Uma tupla com as superclasses da classe a ser criada. Ex: `(SuperKlass,)`

cls_dict

Os métodos e outros atributos da classe a ser criada. Ex:

```
{'x': 42, '__init__': <function initializer at 0x1009c4040>}
```

Ao implementar `MetaKlass.__new__`, podemos inspecionar e modificar aqueles argumentos antes de passá-los para `super().__new__`, que por fim invocará `type.__new__` para criar o novo objeto classe. Após `super().__new__` retornar, podemos aplicar processamentos adicionais à classe recém-criada, antes de devolvê-la para o Python.

Depois de invocar `MetaKlass.__new__` e receber a nova `Klass`, Python então invoca `SuperKlass.__init_subclass__`, passando a classe que criamos. Se existir um decorador de classe acima do bloco `class`, ele é aplicado depois que `__init_subclass__` retorna. Finalmente, Python vincula o objeto classe a seu nome no espaço de nomes atual. No caso mais comum, a instrução `class` está no primeiro nível do módulo, e a `Klass` é inserida no espaço de nomes global do módulo.

O processamento mais comum realizado no `__new__` de uma metaclasses é adicionar ou substituir itens no `cls_dict`, que representa o espaço de nomes da classe em construção. Por exemplo, podemos injetar métodos na classe em construção adicionando funções a `cls_dict`. Entretanto, observe que adicionar métodos pode também ser feito após a classe ser criada, e é por essa razão que podemos fazer isso usando `__init_subclass__` ou um decorador de classe.

Um atributo que precisa ser adicionado a `cls_dict` antes de se executar `type.__new__` é `__slots__`, como discutido na Seção 24.5.1. O método `__new__` de uma metaclasses é o lugar ideal para configurar `__slots__`. A próxima seção mostra como fazer isso.

24.8.2. Um belo exemplo de metaclasses

A metaclasses `MetaBunch` apresentada aqui é uma variação do último exemplo no Capítulo 4 do *Python in a Nutshell, 3rd ed.*, de Alex Martelli, Anna Ravenscroft, e Steve Holden, escrito para rodar sob Python 2.7 e 3.5.^[11] Assumindo o uso de Python 3.6 ou mais recente, pude simplificar ainda mais o código.

Mas primeiro vamos ver o que a classe base `Bunch` oferece:

```
>>> class Point(Bunch):
...     x = 0.0
...     y = 0.0
...     color = 'gray'
...
>>> Point(x=1.2, y=3, color='green')
Point(x=1.2, y=3, color='green')
>>> p = Point()
>>> p.x, p.y, p.color
(0.0, 0.0, 'gray')
>>> p
Point()
```

Subclasses de `Bunch` usam atributos de classe com valores atribuídos, que então se tornam os valores default dos atributos de instância. O `__repr__` gerado omite os argumentos cujos valores são iguais aos defaults.

`MetaBunch`—a metaclasses de `Bunch`—gera `__slots__` para a nova classe a partir de atributos de classe declarados na classe do usuário. Isto impede que atributos com nomes não declarados sejam criados na instanciação ou por atribuição posterior.

```

>>> Point(x=1, y=2, z=3)
Traceback (most recent call last):
...
AttributeError: No slots left for: 'z'
>>> p = Point(x=21)
>>> p.y = 42
>>> p
Point(x=21, y=42)
>>> p.flavor = 'banana'
Traceback (most recent call last):
...
AttributeError: 'Point' object has no attribute 'flavor' and no
__dict__ for setting new attributes

```

Vamos agora mergulhar no elegante código de MetaBunch, no Exemplo 15.

Exemplo 15. metabunch/from3.6/bunch.py: a metaclasses MetaBunch e a classe Bunch

```

class MetaBunch(type): ①
    def __new__(meta_cls, cls_name, bases, cls_dict): ②

        defaults = {} ③

        def __init__(self, **kwargs): ④
            for name, default in defaults.items(): ⑤
                setattr(self, name, kwargs.pop(name, default))
            if kwargs: ⑥
                extra = ', '.join(kwargs)
                raise AttributeError(f'No slots left for: {extra!r}')

        def __repr__(self): ⑦
            rep = ', '.join(f'{name}={value!r}'
                            for name, default in defaults.items()
                            if (value := getattr(self, name)) != default)
            return f'{cls_name}({rep})'

        new_dict = dict(__slots__=[],
                        __init__=__init__,
                        __repr__=__repr__) ⑧

```

```

    for name, value in cls_dict.items(): ⑨
        if name.startswith('__') and name.endswith('__'): ⑩
            if name in new_dict:
                msg = f"Can't set {name!r} in {cls_name!r}"
                raise AttributeError(msg)
            new_dict[name] = value
        else: ⑪
            new_dict['__slots__'].append(name)
            defaults[name] = value
    return super().__new__(meta_cls, cls_name, bases, new_dict) ⑫

class Bunch(metaclass=MetaBunch): ⑬
    pass

```

- ① Para criar uma nova metaclasses, herdamos de `type`.
- ② `__new__` funciona como um método de classe, mas a classe é uma metaclasses, então prefiro nomear o primeiro argumento `meta_cls` (`mcs` é uma alternativa comum). Os três argumentos seguintes são os mesmos da assinatura de três parâmetros de `type()`, quando invocada diretamente para criar uma classe.
- ③ `defaults` vai preservar um mapeamento de nomes de atributos e seus valores default.
- ④ Isto será injetado na nova classe.
- ⑤ Lê `defaults` e define o atributo de instância correspondente, com o valor extraído de `kwargs`, ou um valor default.
- ⑥ Se ainda houver itens em `kwargs`, isso significa que não há posição restante onde possamos colocá-los. Adotamos a prática de *falhar rápido*, então não queremos ignorar silenciosamente os itens em excesso.
- ⑦ `__repr__` devolve uma string que se parece com uma chamada ao construtor—por exemplo, `Point(x=3)`, exibindo somente os atributos com valores diferentes dos respectivos valores default.
- ⑧ Inicializa o espaço de nomes para a nova classe.
- ⑨ Itera sobre o espaço de nomes da classe fornecida pelo usuário.
- ⑩ Se um `name` no formato *dunder* é encontrado, copia o item para o espaço de nomes da nova classe, a menos que ele já esteja lá. Isto evita que usuários

sobrescrevam `__init__`, `__repr__` e outros atributos definidos pelo Python, como `__qualname__` e `__module__`.

- ⑪ Se `name` não for um *dunder*, acrescenta `name` a `__slots__` e armazena seu `value` em `defaults`.
- ⑫ Cria e devolve a nova classe.
- ⑬ Fornece uma classe base, assim os usuários não precisam ver `MetaBunch`.

`MetaBunch` funciona porque tem a oportunidade de configurar `__slots__` antes de invocar `super().__new__` para criar a classe final. Como sempre em metaprogramação, o fundamental é entender a sequência de ações. Vamos fazer outro experimento sobre as etapas de avaliação, agora com uma metaclasses.

24.8.3. Experimento com as etapas de avaliação de metaclasses

Esta é uma variação da Seção 24.7.1, acrescentando uma metaclasses para ter mais emoção. O módulo *builderlib.py* é o mesmo de antes, mas o script principal agora é *evaldemo_meta.py*, listado no Exemplo 16.

Exemplo 16. evaldemo_meta.py: experimentando com uma metaclasses

```
#!/usr/bin/env python3

from builderlib import Builder, deco, Descriptor
from metalib import MetaKlass ①

print('# evaldemo_meta module start')

@deco
class Klass(Builder, metaclass=MetaKlass): ②
    print('# Klass body')

    attr = Descriptor()

    def __init__(self):
        super().__init__()
        print(f'# Klass.__init__({self!r})')

    def __repr__(self):
        return '<Klass instance>'
```

```
def main():
    obj = Klass()
    obj.method_a()
    obj.method_b()
    obj.method_c() ③
    obj.attr = 999

if __name__ == '__main__':
    main()

print('# evaldemo_meta module end')
```

- ① Importa `MetaKlass` de *metalib.py*, que veremos no Exemplo 18.
- ② Declara `Klass` como uma subclasse de `Builder` e uma instância de `MetaKlass`.
- ③ Este método é injetado por `MetaKlass.__new__`, como veremos adiante.



Em nome da ciência, o Exemplo 16 desafia qualquer razão prática e aplica três técnicas diferentes de metaprogramação em `Klass`: um decorador, uma classe base usando `__init_subclass__`, e uma metaclasses customizada. Se você fizer isto em código de produção, não me culpe. Aqui o objetivo é observar a ordem na qual as três técnicas interferem no processo de criação de uma classe.

Como no experimento anterior com as etapas de avaliação, este exemplo não faz nada, apenas exibe mensagens revelando o fluxo de execução. O Exemplo 17 mostra a primeira parte do código de *metalib.py*—o restante está no Exemplo 18.

Exemplo 17. *metatlib.py*: a classe `NosyDict`

```
print('% metalib module start')

import collections

class NosyDict(collections.UserDict):
    def __setitem__(self, key, value):
        args = (self, key, value)
        print(f'% NosyDict.__setitem__ {args!r}')
        super().__setitem__(key, value)

    def __repr__(self):
        return '<NosyDict instance>'
```

Escrevi a classe `NosyDict` para sobrescrever `__setitem__` e exibir cada `key` e cada `value` conforme eles são definidos. A metaclasses vai usar uma instância de `NosyDict` para guardar o espaço de nomes da classe em construção, revelando um pouco mais sobre o funcionamento interno do Python.

A principal atração de *metatlib.py* é a metaclasses no Exemplo 18. Ela implementa o método especial `__prepare__`, um método de classe que Python só invoca em metaclasses. O método `__prepare__` oferece a primeira oportunidade para influenciar o processo de criação de uma nova classe.



Ao programar uma metaclasses, acho útil adotar a seguinte convenção de nomenclatura para argumentos de métodos especiais:

- Usar `cls` em vez de `self` para métodos de instância, pois a instância é uma classe.
- Usar `meta_cls` em vez de `cls` para métodos de classe, pois a classe é uma metaclasses. Lembre-se de que `__new__` se comporta como um método de classe mesmo sem o decorador `@classmethod`.

Exemplo 18. metalib.py: a MetaKlass

```
class MetaKlass(type):
    print('% MetaKlass body')

    @classmethod ①
    def __prepare__(meta_cls, cls_name, bases): ②
        args = (meta_cls, cls_name, bases)
        print(f'% MetaKlass.__prepare__{{args!r}}')
        return NosyDict() ③

    def __new__(meta_cls, cls_name, bases, cls_dict): ④
        args = (meta_cls, cls_name, bases, cls_dict)
        print(f'% MetaKlass.__new__{{args!r}}')
        def inner_2(self):
            print(f'% MetaKlass.__new__:inner_2({self!r})')

        cls = super().__new__(
            meta_cls, cls_name, bases, cls_dict.data) ⑤
        cls.method_c = inner_2 ⑥
        return cls ⑦

    def __repr__(cls): ⑧
        cls_name = cls.__name__
        return f"<class {cls_name!r} built by MetaKlass>"

print('% metalib module end')
```

- ① `__prepare__` deve ser declarado como um método de classe. Ele não é um método de instância, pois a classe em construção ainda não existe quando Python invoca `__prepare__`.
- ② Python invoca `__prepare__` em uma metaclasses para obter um mapeamento para guardar o espaço de nomes da classe em construção.
- ③ Devolve uma instância de `NosyDict` para ser usada como o espaço de nomes.
- ④ `cls_dict` é uma instância de `NosyDict` devolvida por `__prepare__`.
- ⑤ `type.__new__` exige um dict de verdade como último argumento (não um mapeamento qualquer), então fornecemos o atributo `data` de `NosyDict`, herdado de `UserDict`.

- ⑥ Injeta um método na classe recém-criada.
- ⑦ Como sempre, `__new__` devolve o objeto que acaba de ser criado—neste caso, a nova classe.
- ⑧ Definir `__repr__` em uma metaclasses permite customizar o `repr()` de objetos classe.

O principal caso de uso para `__prepare__` antes de Python 3.6 era fornecer um `OrderedDict` para preservar os atributos de uma classe em construção, para que o `__new__` da metaclasses pudesse processar aqueles atributos na ordem em que aparecem no código-fonte da definição de classe do usuário. Agora que `dict` preserva a ordem de inserção, `__prepare__` raramente é necessário. Veremos um uso criativo para ele na Seção 24.11.

Importar *metalib.py* no console de Python não é muito empolgante. Observe o uso de `%` para prefixar as linhas geradas por esse módulo:

```
>>> import metalib
% metalib module start
% MetaKlass body
% metalib module end
```

Muitas coisas acontecem quando importamos *evaldemo_meta.py*, como visto no Exemplo 19.

Exemplo 19. Experimento com evaldemo_meta.py no console

```
>>> import evaldemo_meta
@ builderlib module start
@ Builder body
@ Descriptor body
@ builderlib module end
% metalib module start
% MetaKlass body
% metalib module end
# evaldemo_meta module start ①
% MetaKlass.__prepare__(<class 'metalib.MetaKlass'>, 'Klass', ②
                        (<class 'builderlib.Builder'>,))
% NosyDict.__setitem__(<NosyDict instance>, '__module__',
                      'evaldemo_meta') ③
```



```

% NosyDict.__setitem__(<NosyDict instance>, '__qualname__', 'Klass')
# Klass body
@ Descriptor.__init__(<Descriptor instance>) ④
% NosyDict.__setitem__(<NosyDict instance>, 'attr',
    <Descriptor instance>) ⑤
% NosyDict.__setitem__(<NosyDict instance>, '__init__',
    <function Klass.__init__ at ...>) ⑥
% NosyDict.__setitem__(<NosyDict instance>, '__repr__',
    <function Klass.__repr__ at ...>)
% NosyDict.__setitem__(<NosyDict instance>, '__classcell__', <cell at ...:
empty>)
% MetaKlass.__new__(<class 'metaclasslib.MetaKlass'>, 'Klass',
    (<class 'builderlib.Builder'>,),
    <NosyDict instance>) ⑦
@ Descriptor.__set_name__(<Descriptor instance>,
    <class 'Klass' built by MetaKlass>, 'attr') ⑧
@ Builder.__init_subclass__(<class 'Klass' built by MetaKlass>)
@ deco(<class 'Klass' built by MetaKlass>)
# evaldemo_meta module end

```

- ① As linhas acima desta são exibidas durante a importação de *builderlib.py* e *metaclasslib.py*.
- ② Python invoca `__prepare__` para iniciar o processamento de uma instrução `class`.
- ③ Antes de analisar o corpo da classe, Python acrescenta `__module__` e `__qualname__` ao espaço de nomes de uma classe em construção.
- ④ A instância do descritor é criada...
- ⑤ ...e vinculada a `attr` no espaço de nomes da classe.
- ⑥ Os métodos `__init__` e `__repr__` são definidos e adicionados ao espaço de nomes.
- ⑦ Após terminar o processamento do corpo da classe, Python chama `MetaKlass.__new__`.
- ⑧ Após o método `__new__` da metaclasses devolver a classe recém-criada, os métodos `__set_name__`, `__init_subclass__` e o decorador `@deco` são invocados nesta ordem,

Se executarmos *evaldemo_meta.py* como um script, `main()` é invocada, e mais coisas acontecem. Veja o Exemplo 20.

Exemplo 20. Rodando evaldemo_meta.py como um programa

```
$ ./evaldemo_meta.py
[... 20 linhas omitidas ...]
@ deco(<class 'Klass' built by MetaKlass>) ①
@ Builder.__init__(<Klass instance>)
# Klass.__init__(<Klass instance>)
@ SuperA.__init_subclass__:inner_0(<Klass instance>)
@ deco:inner_1(<Klass instance>)
% MetaKlass.__new__:inner_2(<Klass instance>) ②
@ Descriptor.__set__(<Descriptor instance>, <Klass instance>, 999)
# evaldemo_meta module end
```

- ① As primeiras 21 linhas—incluindo esta—são as mesmas que aparecem no Exemplo 19.
- ② Acionado por `obj.method_c()` em `main`; `method_c` foi injetado por `MetaKlass.__new__`.

Vamos agora voltar à ideia da classe `Checked`, com descritores `Field` implementando validação de tipo durante a execução, e ver como aquilo pode ser feito com uma metaclasses.

24.9. Checked, agora com metaclasses

Não quero estimular a otimização prematura nem excessos de engenharia (*over*), então aqui temos um cenário de ficção para justificar reescrever *_checkedlib.py* com `__slots__`, exigindo a aplicação de uma metaclasses. Fique à vontade para pular a historinha.

Senta que lá vem história

Nosso *checkedlib.py* usando `__init_subclass__` é um sucesso na empresa, e em qualquer dado momento nossos servidores de produção têm milhões de instâncias de subclasses de *Checked* em suas memórias.

Analisando (*profiling*) o uso de memória durante a execução, constatamos que usar `__slots__` pode reduzir os custos de hospedagem, por duas razões:

- Menos uso de memória, já que as instâncias de *Checked* não precisarão ter seus próprios `__dict__`
- Melhor desempenho, pela remoção de `__setattr__`, que foi criado só para bloquear atributos indesejados, mas é acionado na instanciação e para todas as definições de atributos antes de `Field.__set__` ser invocado para fazer seu trabalho

O módulo *metaclass/checkedlib.py*, que estudaremos a seguir, é um substituto direto para *initsub/checkedlib.py*. Os doctests contidos nos dois módulos são idênticos, bem como os arquivos *checkedlib_test.py* para o *pytest*.

A complexidade de *checkedlib.py* é ocultada do usuário. Aqui está o código-fonte de um script que usa o pacote:

Exemplo 21. metaclass/checked_demo.py: exemplo de uso da classe Checked

```
from checkedlib import Checked

class Movie(Checked):
    title: str
    year: int
    box_office: float

if __name__ == '__main__':
    movie = Movie(title='The Godfather', year=1972, box_office=137)
    print(movie)
    print(movie.title)
```

Esta definição concisa da classe `Movie` se vale de três instâncias do descritor de validação `Field`, uma configuração de `__slots__`, cinco métodos herdados de `Checked` e uma metaclasses para juntar tudo isso. A única parte visível de `checkedlib` é a classe base `Checked`.

Estude a Figura 4. A Notação Engenhocas e Bugigangas complementa o diagrama de classes UML, tornando mais visível a relação entre classes e instâncias.

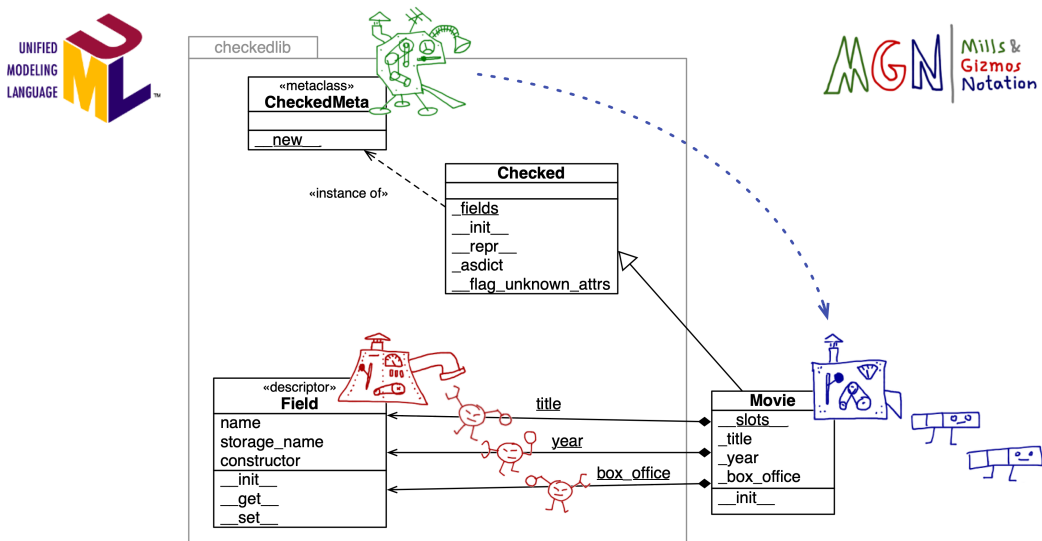


Figura 4. Diagrama de classes UML com MGN: a meta-engenhoca `CheckedMeta` cria a engenhoca `Movie`. A engenhoca `Field` cria os descritores `title`, `year`, e `box_office`, que são atributos de classe de `Movie`. Os dados dos campos são armazenados nos atributos `__title__`, `__year__` e `__box_office__` em cada instância de `Movie`. Note a fronteira do pacote `checkedlib`. O desenvolvedor de `Movie` não precisa entender todo o maquinário dentro de `checkedlib.py`.

Por exemplo, uma classe `Movie` usando a nova `checkedlib.py` é uma instância de `CheckedMeta` e uma subclasse de `Checked`. Os atributos de classe `title`, `year` e `box_office` de `Movie` são três instâncias diferentes de `Field`. Cada instância de `Movie` tem seus próprios atributos `__title__`, `__year__` e `__box_office__`, para armazenar os valores dos campos correspondentes.

Vamos agora estudar o código, começando pela classe `Field` do Exemplo 22. A classe descritora `Field` está um pouco diferente. Nos exemplos anteriores, cada instância do descritor `Field` armazenava seu valor na instância gerenciada, usando um atributo de mesmo nome. Por exemplo, na classe `Movie`, o descritor

title armazenava o valor do campo em um atributo title na instância gerenciada. Isso tornava desnecessário que Field implementasse um método `__get__`.

Entretanto, quando uma classe como Movie usa `__slots__`, ela não pode ter atributos de classe e atributos de instância com o mesmo nome. Cada instância do descritor é um atributo de classe, e agora precisamos de atributos de armazenamento separados em cada instância. O código usa o nome do descritor prefixado por um único `_`. Portanto, instâncias de Field têm atributos `name` e `storage_name` distintos, e implementamos `Field.__get__`.

O Exemplo 22 mostra o código-fonte de Field, com os textos explicativos descrevendo apenas as mudanças nessa versão.

Exemplo 22. metaclass/checkedlib.py: o descritor Field com storage_name e __get__

```
class Field:
    def __init__(self, name: str, constructor: Callable) -> None:
        if not callable(constructor) or constructor is type(None):
            raise TypeError(f'{name!r} type hint must be callable')
        self.name = name
        self.storage_name = '_' + name ①
        self.constructor = constructor

    def __get__(self, instance, owner=None):
        if instance is None: ②
            return self
        return getattr(instance, self.storage_name) ③

    def __set__(self, instance: Any, value: Any) -> None:
        if value is ...:
            value = self.constructor()
        else:
            try:
                value = self.constructor(value)
            except (TypeError, ValueError) as e:
                type_name = self.constructor.__name__
                msg = (f'{value!r} is not compatible ' +
                       f'with {self.name}:{type_name}')
                raise TypeError(msg) from e
            setattr(instance, self.storage_name, value) ④
```

- ① Define `storage_name` a partir do argumento `name`.
- ② Se `__get__` recebe `None` como argumento `instance`, o descritor está sendo lido desde a própria classe gerenciada, não de uma instância gerenciada. Neste caso devolvemos o descritor.
- ③ Caso contrário, devolve o valor armazenado no atributo chamado `storage_name`.
- ④ `__set__` agora usa `setattr` para definir ou atualizar o atributo gerenciado.

O Exemplo 23 mostra o código da metaclasses que controla este exemplo.

Exemplo 23. metaclass/checkedlib.py: a metaclasses CheckedMeta

```
class CheckedMeta(type):

    def __new__(meta_cls, cls_name, bases, cls_dict): ①
        if '__slots__' not in cls_dict: ②
            slots = []
            type_hints = cls_dict.get('__annotations__', {}) ③
            for name, constructor in type_hints.items(): ④
                field = Field(name, constructor) ⑤
                cls_dict[name] = field ⑥
                slots.append(field.storage_name) ⑦

            cls_dict['__slots__'] = slots ⑧

        return super().__new__(
            meta_cls, cls_name, bases, cls_dict) ⑨
```

- ① `__new__` é o único método implementado em `CheckedMeta`.
- ② Só altera a classe se seu `cls_dict` não incluir `__slots__`. Se `__slots__` já existe, assumimos que esta é a classe base `Checked` e não uma subclasse definida pelo usuário, e cria a classe sem modificações.
- ③ Nos exemplos anteriores usamos `typing.get_type_hints` para obter as dicas de tipo, mas aquilo exige uma classe existente como primeiro argumento. Neste ponto, a classe que estamos configurando ainda não existe, então precisamos recuperar `__annotations__` diretamente do `cls_dict`—o espaço de nomes da classe em construção, que Python passa como último argumento para o `__new__` da metaclasses.

- ④ Itera sobre `type_hints` para...
- ⑤ ...criar um `Field` para cada atributo anotado...
- ⑥ ...sobrescreve o item correspondente em `cls_dict` com a instância de `Field`...
- ⑦ ...e acrescenta o `storage_name` do campo à lista que usaremos para...
- ⑧ ...preencher o `__slots__` no `cls_dict`—o espaço de nomes da classe em construção.
- ⑨ Por fim, invocamos `super().__new__`.

A última parte de *metaclass/checkedlib.py* é a classe `Checked`. Usuários da *checkedlib* criam subclasses de `Checked`, como `Movie` no Exemplo 21.

O código desta versão de `Checked` é o mesmo da `Checked` em *initsub/checkedlib.py* (listada no Exemplo 5 e no Exemplo 6), com três modificações:

1. O acréscimo de um `__slots__` vazio, para sinalizar a `CheckedMeta.__new__` que esta classe não precisa de processamento especial.
2. A remoção de `__init_subclass__`, cujo trabalho agora é feito por `CheckedMeta.__new__`.
3. A remoção de `__setattr__`, pois a definição de `__slots__` na classe definida pelo usuário já impede a criação de atributos não declarados.

O Exemplo 24 é a listagem completa da versão final de `Checked`.

Exemplo 24. metaclass/checkedlib.py: a classe base Checked

```
class Checked(metaclass=CheckedMeta):
    __slots__ = () # skip CheckedMeta.__new__ processing

    @classmethod
    def _fields(cls) -> dict[str, type]:
        return get_type_hints(cls)

    def __init__(self, **kwargs: Any) -> None:
        for name in self._fields():
            value = kwargs.pop(name, ...)
            setattr(self, name, value)
        if kwargs:
            self.__flag_unknown_attrs(*kwargs)
```

```

def __flag_unknown_attrs(self, *names: str) -> NoReturn:
    plural = 's' if len(names) > 1 else ''
    extra = ', '.join(f'{name!r}' for name in names)
    cls_name = repr(self.__class__.__name__)
    raise AttributeError(
        f'{cls_name} object has no attribute{plural} {extra}')

def _asdict(self) -> dict[str, Any]:
    return {
        name: getattr(self, name)
        for name, attr in self.__class__.__dict__.items()
        if isinstance(attr, Field)
    }

def __repr__(self) -> str:
    kwargs = ', '.join(
        f'{key}={value!r}' for key, value in self._asdict().items()
    )
    return f'{self.__class__.__name__}({kwargs})'

```

Isto conclui nossa terceira versão de uma fábrica de classes com descritores validados. A próxima seção trata de algumas questões gerais relacionadas a metaclasses.

24.10. Metaclasses no mundo real

Metaclasses são poderosas mas complexas. Antes de se decidir a implementar uma metaclasses, considere os pontos a seguir.

24.10.1. Recursos modernos simplificam ou substituem as metaclasses

Ao longo do tempo, vários casos de uso comum de metaclasses se tornaram redundantes devido a novos recursos da linguagem:

Decoradores de classes

Mais simples de entender que metaclasses, e com menor probabilidade de causar conflitos com classes base e metaclasses.

`__set_name__`

Elimina a necessidade de uma metaclasses com lógica customizada para definir automaticamente o nome de um descritor.^[12]

`__init_subclass__`

Fornece uma forma de customizar a criação de classes que é transparente para o usuário final e ainda mais simples que um decorador—mas pode introduzir conflitos em uma hierarquia de classes complexa.

O dict embutido preservando a ordem de inserção de chaves

Eliminou a principal razão para usar `__prepare__`, que era fornecer um `OrderedDict` para armazenar o espaço de nomes de uma classe em construção. Python só invoca `__prepare__` em metaclasses e então, se fosse necessário processar o espaço de nomes da classe na ordem em que eles aparecem no código-fonte, antes de Python 3.6 era preciso usar uma metaclasses.

Em 2021, todas as versões sob manutenção ativa do CPython suportam todos os recursos listados acima.

Sigo defendendo esses recursos porque vejo muita complexidade desnecessária em nossa profissão, e as metaclasses são uma porta de entrada para a complexidade.

24.10.2. Metaclasses são um recurso estável da linguagem

As metaclasses foram introduzidas no Python em 2002, junto com as assim chamadas *new-style classes* (classes com novo estilo), descritores e propriedades.

É impressionante que o exemplo do `MetaBunch`, postado pela primeira vez por Alex Martelli em julho de 2002, ainda funcione no Python 3.9—a única modificação é a forma de especificar a metaclasses a ser usada, que no Python 3 fazemos com esta sintaxe:

```
class Bunch(metaclass=MetaBunch):  
    ...
```

Nenhum dos recursos modernos citados na Seção 24.10.1 quebrou código existente que usava metaclasses. Mas um código legado com metaclasses frequentemente pode ser simplificado através do uso daqueles recursos, especialmente ignorando versões de Python anteriores à 3.6—que hoje são obsoletas.

24.10.3. Uma classe só pode ter uma metaclasses

Se sua declaração de classe envolver duas ou mais metaclasses, você verá essa intrigante mensagem de erro:

```
TypeError: metaclass conflict: the metaclass of a derived class
must be a (non-strict) subclass of the metaclasses of all its bases
(conflito de metaclasses: a metaclasses de uma classe derivada deve
ser uma subclasse (não-estricta) das metaclasses de todas as suas bases)
```

Isso pode acontecer mesmo sem herança múltipla. Por exemplo, a declaração abaixo pode levantar aquele `TypeError`:

```
class Record(abc.ABC, metaclass=PersistentMeta):
    pass
```

Vimos que `abc.ABC` é uma instância da metaclasses `abc.ABCMeta`. Se aquela metaclasses `Persistent` não for uma subclasse de `abc.ABCMeta`, você tem um conflito de metaclasses.

Há duas maneiras de lidar com esse erro:

- Encontre outra forma de fazer o que precisa ser feito, evitando o uso de pelo menos uma das metaclasses envolvidas.
- Escreva a sua própria metaclasses `PersistentABCMeta` como uma subclasse tanto de `abc.ABCMeta` quanto de `PersistentMeta`, usando herança múltipla, e faça dela a única metaclasses de `Record`.^[13]



A solução de uma metaclasses com duas metaclasses base pode ser implementada para atender um prazo em caso de desespero. Na minha experiência, a programação de metaclasses sempre leva mais tempo que o esperado, tornando esta abordagem arriscada diante de um prazo inflexível. Se fizer isso e cumprir o prazo previsto, seu código pode conter bugs sutis. E mesmo na ausência de bugs conhecidos, esta abordagem deveria ser considerada uma dívida técnica, pelo simples fato de ser difícil de entender e manter.

24.10.4. Metaclasses devem ser detalhes de implementação

Além de `type`, existem apenas outras seis metaclasses em toda a biblioteca padrão de Python 3.9. As metaclasses mais utilizadas (indiretamente) provavelmente são `abc.ABCMeta`, `typing.NamedTupleMeta` e `enum.EnumMeta`. Nenhuma delas deve aparecer explicitamente no código da aplicação, mas somente em bibliotecas. Podemos considerá-las detalhes de implementação.

Apesar de ser possível fazer metaprogramações muito loucas com metaclasses, é melhor se ater ao princípio do menor espanto [fpy.li/24-15], de forma que a maioria dos usuários possa de fato considerar metaclasses detalhes de implementação.^[14]

Nos últimos anos, algumas metaclasses na biblioteca padrão de Python foram substituídas por outros mecanismos, sem afetar a API pública de seus pacotes. A forma mais simples de resguardar estas APIs para o futuro é oferecer uma classe normal, da qual usuários podem então criar subclasses para acessar a funcionalidade fornecida pela metaclasses, como fiz em vários exemplos.

Para encerrar nossa conversa sobre metaprogramação de classes, vou mostrar o pequeno exemplo de metaclasses mais interessante que encontrei durante minha pesquisa para esse capítulo.

24.11. Um *hack* de metaclasses com `__prepare__`

Quando atualizei esse capítulo para a segunda edição, procurei exemplos simples mas interessantes para substituir o código de `LineItem` no exemplo da loja de comida a granel, que não exige mais o uso de metaclasses desde o Python 3.6.

João S. O. Bueno—mais conhecido como JS na comunidade Python brasileira, me apresentou a ideia de metaclasses mais curiosa que já vi.

Uma aplicação de sua ideia é criar uma classe que gera constantes numéricas automaticamente:

```
>>> class Flavor(AutoConst):
...     banana
...     coconut
...     vanilla
...
>>> Flavor.vanilla
2
>>> Flavor.banana, Flavor.coconut
(0, 1)
```

Sim, isto funciona do jeito que está! Este código é um doctest em *autoconst_demo.py*.

Aqui está a classe base fácil de usar `AutoConst`, e a metaclasses por trás dela, implementadas em *autoconst.py*:

```
class AutoConstMeta(type):
    def __prepare__(name, bases, **kwargs):
        return WilyDict()

class AutoConst(metaclass=AutoConstMeta):
    pass
```

É só isso.

Claramente, o truque está em `WilyDict`.

Quando Python processa o espaço de nomes da classe do usuário e lê banana, ele procura aquele nome no mapeamento fornecido por `__prepare__`: uma instância de `WilyDict` que implementa o método `__missing__`, tratado na «Seção 3.5.2» [fpy.li/ct] (vol.1).

A instância de `WilyDict` inicialmente não contém uma chave 'banana', então o método `__missing__` é acionado. Ele cria um item na hora, com a chave 'banana' e o valor 0, e devolve este valor. Python se contenta com isso, e daí tenta acessar 'coconut'. `WilyDict` imediatamente adiciona aquele item com o valor 1, e o devolve. O mesmo acontece com 'vanilla', que é então mapeado para 2.

Já vimos `__prepare__` e `__missing__` antes. A verdadeira inovação é a forma como JS as juntou.

Aqui está o código-fonte de `WilyDict`, também de *autoconst.py*:

```
class WilyDict(dict):
    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)
        self.__next_value = 0

    def __missing__(self, key):
        if key.startswith('__') and key.endswith('__'):
            raise KeyError(key)
        self[key] = value = self.__next_value
        self.__next_value += 1
        return value
```

Enquanto experimentava, descobri que Python procurava `__name__` no espaço de nomes da classe em construção, fazendo com que `WilyDict` acrescentasse um item `__name__` e incrementasse `__next_value`. Eu então inseri uma instrução `if` em `__missing__`, para levantar um `KeyError` quando uma chave se parece com um atributo *dunder*.

Me diverti adicionando mais funcionalidades a `AutoConstMeta` e `AutoConst`, mas em vez de compartilhar meus experimentos, vou deixar vocês se divertirem, brincando com o *hack* genial de JS.

Aqui estão algumas ideias:

- Torne possível obter o nome da constante a partir do valor. Por exemplo, `Flavor[2]` devolveria `'vanilla'`. Você pode fazer isso implementando `__getitem__` em `AutoConstMeta`. Desde o Python 3.9, é possível implementar `__class_getitem__` na própria `AutoConst`.
- Suporte a iteração sobre a classe, implementando `__iter__` na metaclasses. Eu faria `__iter__` produzir as constantes na forma de pares (`name`, `value`).
- Implemente uma nova variante de `Enum`. Isso é um empreendimento épico, pois o pacote `enum` está cheio de truques, incluindo a metaclasses `EnumMeta`, com centenas de linhas de código e um método `__prepare__` bem complicado.

Divirta-se!



O método especial `__class_getitem__` foi introduzido no Python 3.9 para suportar tipos genéricos, como parte da *PEP 585—Type Hinting Generics in Standard Collections* [fpy.li/pep585] (Dicas de Tipos Genéricas em Coleções Padrão). Graças a `__class_getitem__`, os mantenedores de Python não precisaram escrever uma nova metaclasses para que os tipos embutidos implementassem `__getitem__`, de modo que fosse possível escrever dicas de tipo genéricas, tal como `list[int]`. Esse é um recurso limitado, mas representativo, de um caso de uso mais amplo para metaclasses: implementar operadores e outros métodos especiais para funcionarem ao nível da classe, tal como fazer a própria classe iterável, como as subclasses de `Enum`.

24.12. Para encerrar

Metaclasses, bem como decoradores de classes e `__init_subclass__`, são úteis para:

- Registro de subclasses
- Validação estrutural de subclasses
- Aplicar decoradores a muitos métodos ao mesmo tempo

- Serialização de objetos
- Mapeamento objeto-relacional
- Persistência automática de objetos
- Implementar métodos especiais a nível de classe
- Implementar recursos de classes encontrados em outras linguagens, como *traits* [fpy.li/24-17] e «programação orientada a aspecto» [fpy.li/c3]

Em alguns casos, a metaprogramação de classes também pode ajudar em questões de desempenho, executando tarefas no momento da importação que de outra forma seriam executadas repetidamente durante a execução.

Para finalizar, vamos nos lembrar do conselho final de Alex Martelli em seu ensaio *Pássaros aquáticos e as ABCs*, na «Seção 13.5» [fpy.li/cq] (vol.2):

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.

Acredito que o conselho de Martelli se aplica não apenas a ABCs e metaclasses, mas também a hierarquias de classe, sobrecarga de operadores, decoradores de funções, descritores, decoradores de classes e fábricas de classes usando `__init_subclass__`.

Em princípio, essas ferramentas poderosas existem para suportar o desenvolvimento de bibliotecas e frameworks. Naturalmente, as aplicações devem *usar* tais ferramentas, na forma oferecida pela biblioteca padrão de Python ou por pacotes externos. Mas *implementá-las* em código de aplicações é frequentemente resultado de uma abstração prematura.

Bons frameworks são extraídos, não inventados.^[15]

— DHH, criador de Ruby on Rails

24.13. Resumo do capítulo

Este capítulo começou com uma revisão dos atributos encontrados em objetos classe, como `__qualname__` e o método `__subclasses__()`. A seguir, vimos como a classe embutida `type` pode ser usada para criar classes durante a execução.

Apresentei o método especial `__init_subclass__` na primeira versão de uma classe base `Checked`, projetada para substituir dicas de tipo de atributos em subclasses definidas pelo usuário por instâncias do descritor `Field`, que usam construtores para validar o tipo dos atributos durante a execução.

Implementei a mesma ideia com um decorador de classes `@checked`, que acrescenta recursos a classes definidas pelo usuário, de forma similar ao que pode ser feito com `__init_subclass__`. Vimos que `__init_subclass__` ou um decorador de classes não conseguem configurar `__slots__` dinamicamente, pois atuam apenas após a criação da classe, e `__slots__` tem que ser definido antes.

Desvendamos os conceitos de "momento de importação" (*import time*) e "momento de execução" (*run time*) com experimentos mostrando a ordem na qual o código Python é executado quando módulos, descritores, decoradores de classe e `__init_subclass__` são acionados.

Nossa exploração de metaclasses começou com uma explicação geral de `type` como uma metaclasses, e sobre como metaclasses definidas pelo usuário podem implementar `__new__`, para customizar as classes que criam. Vimos então nossa primeira metaclasses customizada, o clássico exemplo `MetaBunch`, usando `__slots__`. A seguir, outro experimento com etapas de avaliação demonstrou como os métodos `__prepare__` e `__new__` de uma metaclasses são invocados antes de `__init_subclass__` e decoradores de classe, oferecendo mais oportunidades para customização de classes.

Estudamos uma terceira versão de uma fábrica de classes `Checked`, com descritores `Field` e uma configuração customizada de `__slots__`, seguida de considerações gerais sobre o uso de metaclasses na prática.

Por fim, vimos o hack `AutoConst`, inventado por João S. O. Bueno, baseado na brilhante ideia de uma metaclasses com `__prepare__` devolvendo um mapeamento que implementa `__missing__`. Em menos de 20 linhas de código, *autoconst.py* demonstra o poder da combinação de técnicas de metaprogramação no Python.

Nunca encontrei outra linguagem como Python, fácil para iniciantes, prática para profissionais e empolgante para hackers. Obrigado, Guido van Rossum e todos que a fazem ser assim.

24.14. Para saber mais

Caleb Hattingh—um dos revisores técnicos desse livro—escreveu o pacote *autoslot* [fpy.li/24-20], fornecendo uma metaclassa para a criação automática do atributo `__slots__` em uma classe definida pelo usuário, através da inspeção do bytecode de `__init__` e da identificação de todas as atribuições a atributos de `self`. Além de útil, esse pacote é um excelente exemplo para estudo: são apenas 74 linhas de código em *autoslot.py*, incluindo 20 linhas de comentários que explicam as partes mais difíceis.

As referências essenciais deste capítulo na documentação de Python são «Personalizando a criação de classe» [fpy.li/by] no capítulo *Modelo de Dados* da *Referência da Linguagem Python*, que cobre `__init_subclass__` e metaclasses. A «documentação da classe `type`» [fpy.li/c4] na página *Funções Embutidas*, e «Atributos especiais» [fpy.li/bt] do capítulo *Tipos embutidos* da *Biblioteca Padrão de Python* também são leituras fundamentais.

Na *Biblioteca Padrão de Python*, a «documentação do módulo `types`» [fpy.li/bz] trata de duas funções introduzidas no Python 3.3 que simplificam a metaprogramação de classes: `types.new_class` e `types.prepare_class`.

Decoradores de classes foram formalizados na *PEP 3129—Class Decorators* [fpy.li/24-25] (Decoradores de Classes), escrita por Collin Winter, com a implementação de referência desenvolvida por Jack Diederich. A palestra *Class Decorators: Radically Simple* [fpy.li/24-26] (Decoradores de Classes: Radicalmente Simples) na PyCon 2009, também de Jack Diederich, é uma rápida introdução a este recurso. Além de `@dataclass`, um exemplo interessante—e mais simples—de decorador de classes na biblioteca padrão de Python é `functools.total_ordering` [fpy.li/7q], que gera métodos especiais para comparação de objetos.

Para metaclasses, a principal referência na documentação de Python é a *PEP 3115—Metaclasses in Python 3000* [fpy.li/pep3115] (Metaclasses no Python 3000), onde o método especial `__prepare__` foi proposto.

O *Python in a Nutshell* 3rd. ed., de Alex Martelli, Anna Ravenscroft, e Steve Holden, é uma referência, mas foi escrito antes da *PEP 487* [fpy.li/pep487] simplificar o processo de customizar classes na sua criação. O principal exemplo de metaclasses no livro—*MetaBunch*—ainda é válido, pois não pode ser escrito com mecanismos mais simples. O *Effective Python* 2nd. ed. (Addison-Wesley), de Brett Slatkin, traz vários exemplos atualizados de técnicas de criação de classes, incluindo metaclasses.

Para aprender sobre as origens da metaprogramação de classes no Python, recomendo o artigo de Guido van Rossum de 2003, *Unifying types and classes in Python 2.2* [fpy.li/24-28] (Unificando tipos e classes no Python 2.2). O texto se aplica também ao Python moderno, pois cobre as chamadas "classes novo estilo"—o comportamento padrão no Python 3—incluindo descritores e metaclasses. Uma das referências citadas por Guido é *Putting Metaclasses to Work: a New Dimension in Object-Oriented Programming*, de Ira R. Forman e Scott H. Danforth (Addison-Wesley), livro para o qual ele deu cinco estrelas na *Amazon.com*, escrevendo o seguinte comentário:

Este livro contribuiu para o projeto das metaclasses no Python 2.2

Pena que esteja fora de catálogo; sempre me refiro a ele como o melhor tutorial que conheço para o difícil tópico da herança múltipla cooperativa, suportada pelo Python através da função `super()`.^[16]

Se você curte metaprogramação, talvez gostaria que Python suportasse o recurso definitivo de metaprogramação: macros sintáticas, como as oferecidas pela família de linguagens Lisp e—mais recentemente—pelo Elixir e pelo Rust. Macros sintáticas são mais poderosas e menos sujeitas a erros que as macros primitivas de substituição de código da linguagem C. Elas são funções especiais que reescrevem código-fonte para código padronizado, usando uma sintaxe customizada, antes da etapa de compilação, permitindo a desenvolvedores introduzir novas estruturas na linguagem sem modificar o compilador. Como a sobrecarga de operadores, macros sintáticas podem ser mal usadas. Mas, desde que a comunidade entenda e controle as desvantagens, elas suportam abstrações poderosas e amigáveis, como as DSLs (*Domain-Specific Languages*, Linguagens de Domínio Específico).

Em setembro de 2020, Marc Shannon, um dos mantenedores de Python, publicou a *PEP 638—Syntactic Macros* [fpy.li/pep638], uma proposta de macros sintáticas. Um ano após sua publicação inicial (quando escrevo essas linhas), a PEP 638 ainda era um rascunho e não havia discussões contínuas sobre ela. Claramente não é uma prioridade muito alta entre os mantenedores de Python. Eu gostaria de ver a PEP 638 sendo melhor discutida e, por fim, aprovada. Macros sintáticas permitiriam à comunidade Python experimentar com novos recursos controversos, tal como o "operador morsa" (*operador walrus*) (PEP 572 [fpy.li/pep572]), casamento de padrões (PEP 634 [fpy.li/pep634]) e regras alternativas para avaliação de dicas de tipo (PEPs 563 [fpy.li/pep563] e 649 [fpy.li/pep649]), antes que se fizessem modificações permanentes no núcleo da linguagem. Nesse meio tempo, podemos sentir o gosto das macros sintáticas com o pacote MacroPy [fpy.li/24-29].

Ponto de vista

Vou iniciar o último ponto de vista no livro com uma longa citação de Brian Harvey e Matthew Wright, dois professores de ciência da computação da Universidade da Califórnia (Berkeley e Santa Barbara). Em seu livro, *Simply Scheme: Introducing Computer Science* ("Simplesmente Scheme: Introduzindo a Ciência da Computação") (MIT Press), Harvey e Wright escreveram:

Há duas escolas de pensamento sobre o ensino de ciência da computação. Podemos representar as duas visões de uma forma caricatural, assim:

1. **A visão conservadora:** *Programas de computador se tornaram muito grandes e complexos para serem apreendidos pela mente humana. Portanto, a tarefa da educação na ciência da computação é ensinar os estudantes como se disciplinarem, de tal forma que 500 programadores medíocres possam se juntar e produzir um programa que atende às especificações.*
2. **A visão radical:** *Programas de computador se tornaram muito grandes e complexos para serem apreendidos pela mente humana. Portanto, a tarefa da educação na ciência da computação é ensinar os estudantes como expandir suas mentes até abarcar os programas, aprendendo a pensar com um vocabulário de ideias maiores, mais poderosas e mais flexíveis que as óbvias. Cada unidade de pensamento programático deve gerar uma grande recompensa para as capacidades do programa.^[17]*

— Brian Harvey e Matthew Wright, no prefácio de *Simply Scheme*

As descrições exageradas de Harvey e Wright versam sobre o ensino de ciência da computação, mas também se aplicam ao projeto de linguagens de programação. Neste ponto você já deve ter adivinhado que concordo com a visão "radical", e acredito que Python foi projetado neste espírito.

A ideia de propriedade é um grande avanço em relação à abordagem "métodos de acesso desde o início", praticamente exigida em Java e suportada pela geração de *getters/setters* por atalhos de teclado nas IDEs Java. A principal vantagem das propriedades é nos permitir começar a criar nossos programas simplesmente expondo atributos publicamente—no espírito do *KISS*—sabendo que um atributo público pode se tornar uma propriedade a qualquer momento sem quebrar código existente. Mas a ideia de descritor vai muito além disso, fornecendo um framework para abstrair lógica repetitiva para acessar atributos. Este framework é tão eficaz que mecanismos essenciais de Python o utilizam por baixo dos panos.

Outra ideia poderosa são as funções como objetos de primeira classe, abrindo caminho para funções de ordem superior. E acontece que a combinação de descritores e funções de ordem superior permite a unificação de funções e métodos. O `__get__` de uma função cria um objeto método sob demanda, vinculando a instância ao argumento `self`. Isto é elegante.^[18]

Por fim, temos a ideia de classes como objetos de primeira classe. É uma façanha impressionante de design que uma linguagem acessível para iniciantes forneça abstrações poderosas, como fábricas de classes, decoradores de classes, e metaclasses completas definidas pelo usuário. Melhor ainda, os recursos avançados estão integrados de forma que não atrapalham a programação casual (eles na verdade ajudam, por trás dos panos). A conveniência e o sucesso de frameworks como o Django e o SQLAlchemy devem muito às metaclasses. Ao longo dos anos, a metaprogramação de classes em Python está se tornando cada vez mais simples, pelo menos para os casos de uso comuns. Os melhores recursos da linguagem são aqueles que beneficiam a todos, mesmo que alguns usuários de Python não os conheçam. Mas esses usuários sempre podem aprender, e criar a próxima grande biblioteca.

Mande notícias sobre suas contribuições ao ecossistema e à comunidade de Python!

[1] Citação extraída do capítulo 2, *Expression* (Expressão), página 10, de *The Elements of Programming Style*, Second Edition (NT: "Elementos de Estilo de Programação"; não encontramos edição traduzida deste livro.)

[2] Isso não quer dizer que a PEP 487 quebrou código que usava aqueles recursos, mas apenas que parte do código que utilizava decoradores de classe ou metaclasses antes de Python 3.6 pode agora ser refatorado para usar classes comuns, resultando em um código mais simples e possivelmente mais eficiente.

[3] Agradeço ao meu amigo J. S. O. Bueno por ter contribuído com esse exemplo.

[4] Não acrescentei dicas de tipo aos argumentos porque os tipos reais são *Any*. Escrevi a dica do tipo devolvido (*None*) para que o *Mypy* não deixe de checar o método.

[5] Isso é verdade para qualquer objeto, exceto quando sua classe sobrescreve os métodos `__str__` ou `__repr__`, herdados de *object*, por uma implementação que não funcione.

[6] Essa solução evita usar *None* como default. Evitar valores nulos é uma boa ideia [fpy.li/24-5]. Em geral, eles são difíceis de evitar, mas em alguns casos isso é fácil. Tanto no Python quanto no SQL, prefiro representar dados ausentes em um campo de texto como uma string vazia em vez de *None* ou *NULL*. Aprender Go reforçou essa ideia: em Go, variáveis e campos *struct* de tipos primitivos são inicializados por default com um "valor zero" (*zero value*). Se você estiver curiosa, veja a página "Zero values" ("Valores zero") [fpy.li/24-6] no *Tour of Go* ("Tour do Go") online

[7] Na minha opinião, *callable* deveria se tornar adequado para dicas de tipo. Em 6 de maio de 2021, quando essa nota foi escrita, essa ainda era uma questão aberta [fpy.li/24-7].

[8] Como mencionado em Laços, sentinelas e pílulas venenosas, o objeto *Ellipsis* é um valor sentinela conveniente e seguro. Ele existe no Python há muito tempo, mas recentemente mais usos têm sido encontrados para ele, como vemos nas dicas de tipo e no *NumPy*.

[9] Não estou dizendo que é uma boa ideia abrir uma conexão com um banco de dados só porque o módulo foi importado, apenas apontando que isso pode ser feito.

[10] Mensagem a *comp.lang.python*, assunto: *Acrimony in c.l.p.* [fpy.li/24-12] (animosidade na c.l.p., o grupo *comp.lang.python*). Esta é outra parte da mesma mensagem de 23 de dezembro de 2002, que citei no *Prefácio de Python Fluente*. O *TimBot* estava inspirado naquele dia.

[11] Os autores gentilmente me deram permissão para usar seu exemplo. *MetaBunch* apareceu pela primeira vez em uma mensagem enviada por Martelli para o grupo *comp.lang.python*, em 7 de julho de 2002, com o assunto "a nice metaclass example (was Re: structs in python)" (*um belo exemplo de metaclasses (era Re: structs no python)*) [fpy.li/24-13], na sequência de uma discussão sobre estruturas de dados similares a registros no Python. O código original de Martelli, para Python 2.2, ainda roda após uma única modificação: para usar uma metaclasses no Python 3, é necessário usar o argumento nomeado *metaclass* na declaração da classe (por exemplo, *Bunch(metaclass=MetaBunch)*), em vez da convenção antiga, que era adicionar um atributo `__metaclass__` no corpo da classe.

[12] Na primeira edição de *Python Fluente*, as versões mais avançadas da classe *LineItem* usavam uma metaclasses apenas para definir o nome do armazenamento dos atributos. Veja o código nas metaclasses do exemplo da comida a granel [fpy.li/24-14], no repositório de código da primeira edição.

[13] Se você sentiu vertigem ao ponderar sobre as implicações de herança múltipla com metaclasses, bom para você. Eu também passaria longe dessa solução.

[14] Eu ganhei a vida por alguns anos escrevendo código para Django, antes de resolver estudar como os campos dos modelos Django eram implementados. Só então aprendi sobre descritores e metaclasses.

[15] Esta frase é muito citada. Encontrei uma citação direta antiga no blog de DHH, em post [fpy.li/24-19] de 2005.

[16] Comprei um exemplar usado, e achei uma leitura muito difícil. Não cheguei ao final.

[17] Brian Harvey e Matthew Wright, *Simply Scheme* (MIT Press, 1999), p. xvii. O livro completo está disponível em Berkeley.edu [fpy.li/24-30].

[18] *Machine Beauty: Elegance and the Heart of Technology* (Beleza de Máquina: A Elegância e o Coração da Tecnologia), de David Gelernter (Basic Books), começa com uma discussão intrigante sobre elegância e estética em obras de engenharia, de pontes a software. Os capítulos posteriores não são tão bons, mas o início vale o preço.