Programação Funcional COMO FAZER

Release 3.13.0b3

Guido van Rossum and the Python development team

julho 07, 2024

Python Software Foundation
Email: docs@python.org

Sumário

Autor
A. M. Kuchling

Versão
0.32

Nesse documento, nós vamos passear pelos recursos do Python que servem para implementar programas de forma funcional. Após uma introdução dos conceitos da programação funcional, nós veremos as propriedades da linguagem como iteradores e geradores e bibliotecas de módulos relevantes como um `itertools` e `functools`.

1 Introdução

Essa seção explica o conceito básico da programação funcional; se você está interessado em aprender sobre os recursos da linguagem Python, pule essa etapa para a seção *Iteradores*.

As linguagens de programação permitem decompor problemas de diversas maneiras diferentes:

- A Maioria das linguagens de programação são **procedurais**: os programas são listas de instruções que dizem ao computador o que fazer com as entradas do programa. C, Pascal, e mesmo o Unix shells são linguagens procedurais.
- Em linguagens **declarativas**, você escreve uma especificação que descreve o problema a ser resolvido, e a implementação do idioma descreve como executar a computação de forma eficiente. O SQL é o idioma declarativo com o qual provavelmente você está familiarizado; Uma consulta SQL descreve o conjunto de dados que deseja recuperar e o mecanismo SQL decide se deseja escanear tabelas ou usar índices, quais subcláusulas devem ser realizadas primeiro, etc.
- Os programas **orientados a objetos** manipulam coleções de objetos. Os objetos têm estado interno e métodos de suporte que consultam ou modificam esse estado interno de alguma forma. Smalltalk e Java são linguagens orientadas a objetos. C++ e Python são idiomas que suportam programação orientada a objetos, mas não forçam o uso de recursos de orientação a objetos.
- A programação **funcional** decompõe um problema em um conjunto de funções. Idealmente, as funções apenas recebem entradas e produzem saídas, e não têm nenhum estado interno que afete a saída produzida para uma determinada entrada. Linguagens funcionais bem conhecidos incluem a família ML (Standard ML, OCaml e outras variantes) e Haskell.

Os designers de algumas linguagens de computadores escolhem enfatizar uma abordagem particular da programação. Isso muitas vezes torna difícil escrever programas que usam uma abordagem diferente. Outros idiomas são linguagens com multiparadigmas que suportam várias abordagens diferentes. Lisp, C++ e Python são multiparadigmas; Você pode escrever programas ou bibliotecas que são em grande parte processuais, orientadas a objetos ou funcionais em todos esses idiomas. Em um grande programa, diferentes seções podem ser escritas usando diferentes abordagens; A GUI pode ser orientada a objetos enquanto a lógica de processamento é processual ou funcional, por exemplo.

Em um programa funcional, a entrada flui através de um conjunto de funções. Cada função opera em sua entrada e produz alguma saída. O estilo funcional desencoraja funções com efeitos colaterais que modificam o estado interno ou fazem outras alterações que não são visíveis no valor de retorno da função. As funções que não têm efeitos colaterais são chamadas **puramente funcionais**. Evitar efeitos colaterais significa não usar estruturas de dados que sejam atualizadas à medida que um programa é executado; A saída de cada função só deve depender da sua entrada.

Algumas linguagens são muito estritas a respeito da pureza e não tem nem instruções de atribuição como por exemplo `a=3` ou `c = a + b`, mas é difícil evitar todos os efeitos colateirais, como exibir valores na tela ou escrever em um arquivo em disco. Um outro exemplo é a chamada da função `print()` ou da função `time.sleep()`, nenhuma das quais devolve um valor útil. Ambas são chamadas apenas por conta do efeito colateral de mandar algum texto para a tela ou pausar a execução por um segundo.

Os programas Python escritos em estilo funcional geralmente não irão ao extremo de evitar todas as I/O ou todas as atribuições; Em vez disso, eles fornecerão uma interface de aparência funcional, mas usarão recursos não funcionais internamente. Por exemplo, a implementação de uma função ainda usará atribuições para variáveis locais, mas não modificará variáveis globais ou terá outros efeitos colaterais.

A programação funcional pode ser considerada o oposto da programação orientada a objetos. Os objetos são pequenas cápsulas contendo algum estado interno, juntamente com uma coleção de chamadas de método que permitem modificar este estado, e os programas consistem em fazer o conjunto correto de mudanças de estado. A programação funcional quer evitar as mudanças de estado tanto quanto possível e funciona com o fluxo de dados entre as funções. Em Python você pode combinar as duas abordagens escrevendo funções que levam e retornam instâncias que representam objetos em seu aplicativo (mensagens de e-mail, transações, etc.).

O design funcional pode parecer uma restrição estranha para trabalhar por baixo. Por que você deve evitar objetos e efeitos colaterais? Existem vantagens teóricas e práticas para o estilo funcional:

- Probabilidade formal.
- Modularidade.
- Componibilidade.
- Fácil de depurar e testar.

1.1 Probabilidade formal

Um benefício teórico é que é mais fácil construir uma prova matemática de que um programa funcional é correto.

Durante muito tempo os pesquisadores estiveram interessados em encontrar maneiras de provar matematicamente programas corretos. Isso é diferente de testar um programa em inúmeras entradas e concluir que sua saída geralmente é correta, ou ler o código-fonte de um programa e concluir que o código parece certo; O objetivo é uma prova rigorosa de que um programa produz o resultado certo para todas as entradas possíveis.

A técnica usada para comprovar os programas corretos é escrever **invariantes**, propriedades dos dados de entrada e das variáveis do programa que são sempre verdadeiras. Para cada linha de código, você mostra que se os invariantes X e Y forem verdadeiros **antes** da linha ser executada, os invariantes X' e Y' ligeiramente diferentes são verdadeiros **após** a linha ser executada. Isso continua até chegar ao final do programa, em que ponto as invariantes devem corresponder às condições desejadas na saída do programa.

A evitação das tarefas funcionais ocorreu porque as tarefas são difíceis de tratar com esta técnica; As atribuições podem invadir invariantes que eram verdadeiras antes da atribuição sem produzir novos invariantes que possam ser propagados para a frente.

Infelizmente, prpvar que os programas estão corretos são praticamente impraticáveis e não relevantes para o software Python. Mesmo os programas triviais exigem provas de várias páginas; A prova de correção para um programa moderadamente complicado seria enorme, e poucos ou nenhum dos programas que você usa diariamente (o interpretador Python, seu analisador XML, seu navegador) poderia ser comprovado correto. Mesmo que você tenha anotado ou gerado uma prova, então haveria a questão de verificar a prova; Talvez haja um erro nisso, e você acredita erroneamente que você provou o programa corretamente.

1.2 Modularidade

Um benefício mais prático da programação funcional é que isso força você a quebrar seu problema em pequenos pedaços. Os programas são mais modulares como resultado. É mais fácil especificar e escrever uma pequena função que faz uma coisa do que uma grande função que realiza uma transformação complicada. Pequenas funções também são mais fáceis de ler e verificar erros.

1.3 Fácil de depurar e testar

Testar e depurar um programa de estilo funcional é mais fácil.

A depuração é simplificada porque as funções são geralmente pequenas e claramente especificadas. Quando um programa não funciona, cada função é um ponto de interface onde você pode verificar se os dados estão corretos. Você pode observar as entradas e saídas intermediárias para isolar rapidamente a função responsável por um erro.

O teste é mais fácil porque cada função é um assunto potencial para um teste unitário. As funções não dependem do estado do sistema que precisa ser replicado antes de executar um teste; Em vez disso, você só precisa sintetizar a entrada certa e depois verificar se o resultado corresponde às expectativas.

1.4 Componibilidade

À medida que você trabalha em um programa de estilo funcional, você escreverá várias funções com diferentes entradas e saídas. Algumas dessas funções serão inevitavelmente especializadas em um aplicativo particular, mas outras serão úteis em uma grande variedade de programas. Por exemplo, uma função que leva um caminho de diretório e retorna todos os arquivos XML no diretório, ou uma função que leva um nome de arquivo e retorna seu conteúdo, pode ser aplicada em muitas situações diferentes.

Com o tempo você formará uma biblioteca pessoal de utilitários. Muitas vezes você montará novos programas organizando funções existentes em uma nova configuração e escrevendo algumas funções especializadas para a tarefa atual.

2 Iteradores

Começarei por olhar para um recurso de linguagem Python que é uma base importante para escrever programas de estilo funcional: iteradores.

Um iterador é um objeto que representa um fluxo de dados; este objeto retorna os dados um elemento por vez. Um iterador Python deve suportar um método chamado `iterator.__next__()` que não leva argumentos e sempre retorna o próximo elemento do fluxo. Se não houver mais elementos no fluxo, `iterator.__next__()` deve aumentar a exceção `StopIteration`. Os iteradores não precisam ser finitos; no entanto, é perfeitamente razoável escrever um iterador que produza um fluxo infinito de dados.

A função embutida `iter()` leva um objeto arbitrário e tenta retornar um iterador que retornará os conteúdos ou elementos do objeto, caso o contrario retorna `TypeError` se o objeto não suportar a iteração. Vários dos tipos de dados embutidos do Python suportam a iteração, sendo as listas e os dicionários mais comuns. Um objeto é chamado iterable se você pode obter um iterador para ele.

Você pode experimentar a interface de iteração manualmente:

```
>>> L = [1, 2, 3]
>>> it = iter(L)
>>> it
<...iterator object at ...>
>>> it.__next__()  # same as next(it)
1
>>> next(it)
2
>>> next(it)
3
>>> next(it)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
>>>
```

Python espera objetos iteráveis em vários contextos diferentes, sendo o mais importante a instrução `for`. Na declaração `for X in Y`, Y deve ser um iterador ou algum objeto para o qual `iter()` pode criar um iterador. Estas duas declarações são equivalentes:

```
for i in iter(obj):
    print(i)

for i in obj:
    print(i)
```

Iteradores também podem ser materializados como listas ou tuplas, utilizando funções de construção `list()` ou `tuple()`:

```
>>> L = [1, 2, 3]
>>> iterator = iter(L)
>>> t = tuple(iterator)
>>> t
(1, 2, 3)
```

A descompilação de sequência também suporta iteradores: se você sabe que um iterador retornará N elementos, você pode descompactá-los em uma N-tupla:

```
>>> L = [1, 2, 3]
>>> iterator = iter(L)
>>> a, b, c = iterator
>>> a, b, c
(1, 2, 3)
```

Funções embutidas, tais como `max()` e `min()` podem ter um único argumento de iterador e retornarão o elemento maior ou menor. Os operadores `"in"` e `"not in"` também aceitam iteradores: `X in iterator` é verdadeiro se X for encontrado no fluxo retornado pelo iterador. Você enfrentará problemas óbvios se o iterador for infinito; `max()`, `min()` nunca retornará, e se o elemento X nunca aparecer no fluxo, os operadores `"in"` e `"not in"` não retornarão também.

Observe que você só pode avançar em um iterador; não há como obter o elemento anterior, redefinir o iterador ou fazer uma cópia dele. Os objetos iteradores podem opcionalmente fornecer esses recursos adicionais, mas o protocolo do iterador especifica apenas o método `iterator.__next__()`. As funções podem, portanto, consumir toda a saída do iterador, e se você precisa fazer algo diferente com o mesmo fluxo, você terá que criar um novo iterador.

2.1 Tipos de Dados que Suportam Iteradores

Já vimos como listas e tuplas suportam iteradores. De fato, qualquer tipo de sequência de Python, como strings, suportará automaticamente a criação de um iterador.

Chamar `iter()` em um dicionário retorna um iterador que irá percorrer as chaves do dicionário:

```
>>> m = {'Jan': 1, 'Feb': 2, 'Mar': 3, 'Apr': 4, 'May': 5, 'Jun': 6,
...      'Jul': 7, 'Aug': 8, 'Sep': 9, 'Oct': 10, 'Nov': 11, 'Dec': 12}
>>> for key in m:
...     print(key, m[key])
Jan 1
Feb 2
Mar 3
Apr 4
May 5
Jun 6
Jul 7
Aug 8
Sep 9
Oct 10
```

(continua na próxima página)

(continuação da página anterior)

```
Nov 11
Dec 12
```

Note que a partir de Python 3.7, a ordem de iteração em um dicionário é garantidamente a mesma que a ordem de inserção. Em versões anteriores, o comportamento não era especificado e podia variar entre implementações.

Aplicando `iter()` para um dicionário sempre percorre as teclas, mas os dicionários têm métodos que retornam outros iteradores. Se você deseja iterar sobre valores ou pares de chave/valor, você pode chamar explicitamente os métodos `values()` ou `items()` para obter um iterador apropriado.

O construtor `dict()` pode aceitar um iterador que retorna um fluxo finito de tuplas `(chave, valor)`:

```
>>> L = [('Italy', 'Rome'), ('France', 'Paris'), ('US', 'Washington DC')]
>>> dict(iter(L))
{'Italy': 'Rome', 'France': 'Paris', 'US': 'Washington DC'}
```

Os arquivos também suportam a iteração chamando o método `readline()` até que não haja mais linhas no arquivo. Isso significa que você pode ler cada linha de um arquivo como este:

```
for line in file:
    # do something for each line
    ...
```

Os conjuntos podem tirar seus conteúdos de uma iterável e permitir que você faça uma iteração sobre os elementos do conjunto:

```
>>> S = {2, 3, 5, 7, 11, 13}
>>> for i in S:
...     print(i)
2
3
5
7
11
13
```

3 Expressões do gerador e compreensões de lista

Duas operações comuns na saída de um iterador são 1) executando alguma operação para cada elemento, 2) selecionando um subconjunto de elementos que atendam a alguma condição. Por exemplo, com uma lista de cadeias de caracteres, você pode querer retirar o espaço em branco de cada linha ou extrair todas as sequências de caracteres que contenham uma determinada substring.

As compreensões de lista e as expressões do gerador (forma curta: "listcomps" e "genexps") são uma notação concisa para tais operações, emprestado da linguagem de programação funcional Haskell (https://www.haskell.org/). Você pode tirar todos os espaços em branco de um fluxo de strings com o seguinte código:

```
>>> line_list = ['  line 1\n', 'line 2  \n', ' \n', '']

>>> # Generator expression -- returns iterator
>>> stripped_iter = (line.strip() for line in line_list)

>>> # List comprehension -- returns list
>>> stripped_list = [line.strip() for line in line_list]
```

Você pode selecionar apenas determinados elementos adicionando uma condição `"if"`:

```
>>> stripped_list = [line.strip() for line in line_list
...                  if line != ""]
```

Com uma compreensão de lista, você recebe uma lista Python; `Stripped_list` é uma lista contendo as linhas resultantes, e não um iterador. As expressões do gerador retornam um iterador que calcula os valores conforme necessário, não precisando materializar todos os valores ao mesmo tempo. Isso significa que as compreensões da lista não são úteis se você estiver trabalhando com iteradores que retornam um fluxo infinito ou uma quantidade muito grande de dados. As expressões do gerador são preferíveis nessas situações.

As expressões do gerador são cercadas por parênteses ("()") e as compreensões da lista são cercadas por colchetes ("[]"). As expressões do gerador têm a forma:

```
( expression for expr in sequence1
             if condition1
             for expr2 in sequence2
             if condition2
             for expr3 in sequence3
             ...
             if condition3
             for exprN in sequenceN
             if conditionN )
```

Novamente, para uma compreensão de lista, apenas os suportes externos são diferentes (colchetes em vez de parênteses).

Os elementos do resultado gerado serão os valores sucessivos de `expression`. As cláusulas `if` são todas opcionais; Se presente, `expression` só é avaliado e adicionado ao resultado quando `condition` é verdadeiro.

As expressões do gerador sempre devem ser escritas dentro de parênteses, mas os parênteses que sinalizam uma chamada de função também contam. Se você quiser criar um iterador que será imediatamente passado para uma função, você pode escrever:

```
obj_total = sum(obj.count for obj in list_all_objects())
```

As cláusulas `for...in` contêm as sequências a serem repetidas. As sequências não precisam ser do mesmo comprimento, porque são iteradas da esquerda para a direita, **não** em paralelo. Para cada elemento em `sequence1`, `sequence2` é enrolado desde o início. `sequence3` é então percorrido para cada par resultante de elementos de `sequence1` e `sequence2`.

Em outras palavras, uma lista de compreensão ou expressão do gerador é equivalente ao seguinte código Python:

```
for expr1 in sequence1:
    if not (condition1):
        continue   # Skip this element
    for expr2 in sequence2:
        if not (condition2):
            continue   # Skip this element
        ...
        for exprN in sequenceN:
            if not (conditionN):
                continue   # Skip this element

            # Output the value of
            # the expression.
```

Isso significa que, quando existem várias cláusulas `for...in`, mas não `if`, o comprimento da saída resultante será igual ao produto dos comprimentos de todas as sequências. Se você tiver duas listas de comprimento 3, a lista de saída tem 9 elementos de comprimento:

```
>>> seq1 = 'abc'
>>> seq2 = (1, 2, 3)
>>> [(x, y) for x in seq1 for y in seq2]
[('a', 1), ('a', 2), ('a', 3),
 ('b', 1), ('b', 2), ('b', 3),
 ('c', 1), ('c', 2), ('c', 3)]
```

Para evitar a introdução de uma ambiguidade na gramática de Python, se `expression` estiver criando uma tupla, ela deve estar cercada de parênteses. A primeira lista de compreensão abaixo é um erro de sintaxe, enquanto o segundo está correto:

```
# Syntax error
[x, y for x in seq1 for y in seq2]
# Correct
[(x, y) for x in seq1 for y in seq2]
```

4 Geradores

Os geradores são uma classe especial de funções que simplificam a tarefa de escrever iteradores. As funções convencionais calculam um valor e o retornam, mas os geradores retornam um iterador que retorna um fluxo de valores.

Você está sem dúvida familiarizado com o funcionamento das funções convencionais em Python ou C. Quando você chama uma função, ela recebe um espaço de nome privado onde suas variáveis locais são criadas. Quando a função atinge uma instrução `return`, as variáveis locais são destruídas e o valor retornado ao chamador. Uma chamada posterior para a mesma função cria um novo espaço de nome privado e um novo conjunto de variáveis locais. Mas, e se as variáveis locais não fossem descartadas ao sair de uma função? E se você pudesse retomar a função onde ele deixou? Isto é o que os geradores fornecem; Eles podem ser pensados como funções resumíveis.

Aqui está um exemplo simples de uma função geradora:

```
>>> def generate_ints(N):
...     for i in range(N):
...         yield i
```

Qualquer função contendo um argumento nomeado `yield` é uma função de gerador; isso é detectado pelo compilador de bytecode do Python, que compila a função especialmente como resultado.

Quando você chama uma função de gerador, não retorna um único valor; em vez disso, ele retorna um objeto gerador que suporte o protocolo do iterador. Ao executar a expressão `yield`, o gerador exibe o valor de `i`, semelhante a uma instrução `return`. A grande diferença entre `yield` e uma declaração `return` é que, ao atingir um `yield`, o estado de execução do gerador é suspenso e as variáveis locais são preservadas. Na próxima chamada ao método do gerador `__next__()`, a função irá continuar a ser executada.

Aqui está um uso simples do gerador `generate_ints()`

```
>>> gen = generate_ints(3)
>>> gen
<generator object generate_ints at ...>
>>> next(gen)
0
>>> next(gen)
1
>>> next(gen)
2
>>> next(gen)
Traceback (most recent call last):
  File "stdin", line 1, in <module>
  File "stdin", line 2, in generate_ints
StopIteration
```

Você pode igualmente escrever `for i in generate_ints(5)`, ou `a, b, c = generate_ints(3)`.

Dentro de uma função de gerador, `return value` faz com que `StopIteration(value)` seja gerado a partir do método `__next__()`. Uma vez que isso acontece, ou a parte inferior da função é alcançada, a procissão dos valores termina e o gerador não pode render mais valores.

Você pode conseguir o efeito de geradores manualmente, escrevendo sua própria classe e armazenando todas as variáveis locais do gerador como variáveis de instância. Por exemplo, retornar uma lista de inteiros pode ser feito

configurando `self.count` para 0, e tendo o `iterator.__next__()` incrementar o método `self.count` e devolvê-lo. No entanto, para um gerador moderadamente complicado, escrever uma classe correspondente pode ser muito mais complicado.

O conjunto de teste incluído na biblioteca do Python, lib/test/test_generators.py, contém vários exemplos mais interessantes. Aqui está um gerador que implementa uma passagem em ordem de uma árvore usando geradores de forma recursiva.

```
# A recursive generator that generates Tree leaves in in-order.
def inorder(t):
    if t:
        for x in inorder(t.left):
            yield x

        yield t.label

        for x in inorder(t.right):
            yield x
```

Dois outros exemplos no `test_generators.py` produzem soluções para o problema N-Queens (colocando N rainhas em um tabuleiro de xadrez NxN para que nenhuma rainha ameace a outra) e o Passeio do Cavalo (encontrando uma rota que leva um cavalo para cada quadrado de um tabuleiro de xadrez NxN sem visitar nenhum quadrado duas vezes).

4.1 Passando valores para um gerador

No Python 2.4 e anteriores, os geradores apenas produziram saída. Uma vez que o código de um gerador foi invocado para criar um iterador, não havia como passar qualquer nova informação na função quando sua execução foi retomada. Você pode cortar essa habilidade fazendo com que o gerador olhe para uma variável global ou passando em algum objeto mutável que os chamadores então modifiquem, mas essas abordagens são desordenadas.

No Python 2.5 há uma maneira simples de passar valores para um gerador. `yield` tornou-se uma expressão, retornando um valor que pode ser atribuído a uma variável ou operado de outra forma:

```
val = (yield i)
```

Eu recomendo que você **sempre** coloque parênteses em torno de uma expressão `yield` quando você está fazendo algo com o valor retornado, como no exemplo acima. Os parênteses nem sempre são necessários, mas é mais fácil adicioná-los sempre em vez de ter que lembrar quando são necessários.

(**PEP 342** explica as regras exatas, é que uma expressão `yield` deve sempre ser entre parênteses, exceto quando ocorre na expressão de nível superior no lado direito de uma atribuição. Isso significa que você pode escrever `val = yield i`, mas tem que usar parênteses quando há uma operação, como em `val = (yield i) + 12`.)

Os valores são enviados para um gerador chamando seu método `send(value)`. Este método retoma o código do gerador e a expressão `yield` retorna o valor especificado. Se o método regular `__next__()` for chamado, o `yield` retorna `None`.

Aqui está um contador simples que aumenta em 1 e permite alterar o valor do contador interno.

```
def counter(maximum):
    i = 0
    while i < maximum:
        val = (yield i)
        # If value provided, change counter
        if val is not None:
            i = val
        else:
            i += 1
```

E aqui um exemplo de mudança de contador:

```
>>> it = counter(10)
>>> next(it)
0
>>> next(it)
1
>>> it.send(8)
8
>>> next(it)
9
>>> next(it)
Traceback (most recent call last):
  File "t.py", line 15, in <module>
    it.next()
StopIteration
```

Porque `yield` retornará frequentemente `None`, você deve verificar sempre este caso. Não use apenas seu valor em expressões, a menos que tenha certeza de que o método `send()` será o único método usado para retomar a função do gerador.

In addition to `send()`, there are two other methods on generators:

- `throw(value)` is used to raise an exception inside the generator; the exception is raised by the `yield` expression where the generator's execution is paused.
- `close()` raises a `GeneratorExit` exception inside the generator to terminate the iteration. On receiving this exception, the generator's code must either raise `GeneratorExit` or `StopIteration`; catching the exception and doing anything else is illegal and will trigger a `RuntimeError`. `close()` will also be called by Python's garbage collector when the generator is garbage-collected.

  If you need to run cleanup code when a `GeneratorExit` occurs, I suggest using a `try: ... finally:` suite instead of catching `GeneratorExit`.

The cumulative effect of these changes is to turn generators from one-way producers of information into both producers and consumers.

Generators also become **coroutines**, a more generalized form of subroutines. Subroutines are entered at one point and exited at another point (the top of the function, and a `return` statement), but coroutines can be entered, exited, and resumed at many different points (the `yield` statements).

5 Funções embutidas

Let's look in more detail at built-in functions often used with iterators.

Two of Python's built-in functions, `map()` and `filter()` duplicate the features of generator expressions:

`map(f, iterA, iterB, ...)` returns an iterator over the sequence
`f(iterA[0], iterB[0]), f(iterA[1], iterB[1]), f(iterA[2], iterB[2]), ....`

```
>>> def upper(s):
...     return s.upper()
```

```
>>> list(map(upper, ['sentence', 'fragment']))
['SENTENCE', 'FRAGMENT']
>>> [upper(s) for s in ['sentence', 'fragment']]
['SENTENCE', 'FRAGMENT']
```

É claro que você pode alcançar o mesmo efeito com uma compreensão de lista.

`filter(predicate, iter)` returns an iterator over all the sequence elements that meet a certain condition, and is similarly duplicated by list comprehensions. A **predicate** is a function that returns the truth value of some condition; for use with `filter()`, the predicate must take a single value.

```
>>> def is_even(x):
...     return (x % 2) == 0
```

```
>>> list(filter(is_even, range(10)))
[0, 2, 4, 6, 8]
```

isso também pode ser escrito como uma compreensão de lista:

```
>>> list(x for x in range(10) if is_even(x))
[0, 2, 4, 6, 8]
```

`enumerate(iter, start=0)` counts off the elements in the iterable returning 2-tuples containing the count (from *start*) and each element.

```
>>> for item in enumerate(['subject', 'verb', 'object']):
...     print(item)
(0, 'subject')
(1, 'verb')
(2, 'object')
```

`enumerate()` is often used when looping through a list and recording the indexes at which certain conditions are met:

```
f = open('data.txt', 'r')
for i, line in enumerate(f):
    if line.strip() == '':
        print('Blank line at line #%i' % i)
```

`sorted(iterable, key=None, reverse=False)` collects all the elements of the iterable into a list, sorts the list, and returns the sorted result. The *key* and *reverse* arguments are passed through to the constructed list's `sort()` method.

```
>>> import random
>>> # Generate 8 random numbers between [0, 10000)
>>> rand_list = random.sample(range(10000), 8)
>>> rand_list
[769, 7953, 9828, 6431, 8442, 9878, 6213, 2207]
>>> sorted(rand_list)
[769, 2207, 6213, 6431, 7953, 8442, 9828, 9878]
>>> sorted(rand_list, reverse=True)
[9878, 9828, 8442, 7953, 6431, 6213, 2207, 769]
```

(For a more detailed discussion of sorting, see the sortinghowto.)

The `any(iter)` and `all(iter)` built-ins look at the truth values of an iterable's contents. `any()` returns `True` if any element in the iterable is a true value, and `all()` returns `True` if all of the elements are true values:

```
>>> any([0, 1, 0])
True
>>> any([0, 0, 0])
False
>>> any([1, 1, 1])
True
>>> all([0, 1, 0])
False
>>> all([0, 0, 0])
False
>>> all([1, 1, 1])
True
```

`zip(iterA, iterB, ...)` takes one element from each iterable and returns them in a tuple:

```
zip(['a', 'b', 'c'], (1, 2, 3)) =>
  ('a', 1), ('b', 2), ('c', 3)
```

It doesn't construct an in-memory list and exhaust all the input iterators before returning; instead tuples are constructed and returned only if they're requested. (The technical term for this behaviour is lazy evaluation.)

This iterator is intended to be used with iterables that are all of the same length. If the iterables are of different lengths, the resulting stream will be the same length as the shortest iterable.

```
zip(['a', 'b'], (1, 2, 3)) =>
  ('a', 1), ('b', 2)
```

You should avoid doing this, though, because an element may be taken from the longer iterators and discarded. This means you can't go on to use the iterators further because you risk skipping a discarded element.

6 O módulo itertools

The `itertools` module contains a number of commonly used iterators as well as functions for combining several iterators. This section will introduce the module's contents by showing small examples.

The module's functions fall into a few broad classes:

- Funções que criam um novo iterador com base em um iterador existente.
- Funções para tratar os elementos de um iterador como argumentos de funções.
- Funções para selecionar partes da saída de um iterador.
- A function for grouping an iterator's output.

6.1 Criando novos iteradores

`itertools.count(start, step)` returns an infinite stream of evenly spaced values. You can optionally supply the starting number, which defaults to 0, and the interval between numbers, which defaults to 1:

```
itertools.count() =>
  0, 1, 2, 3, 4, 5, 6, 7, 8, 9, ...
itertools.count(10) =>
  10, 11, 12, 13, 14, 15, 16, 17, 18, 19, ...
itertools.count(10, 5) =>
  10, 15, 20, 25, 30, 35, 40, 45, 50, 55, ...
```

`itertools.cycle(iter)` saves a copy of the contents of a provided iterable and returns a new iterator that returns its elements from first to last. The new iterator will repeat these elements infinitely.

```
itertools.cycle([1, 2, 3, 4, 5]) =>
  1, 2, 3, 4, 5, 1, 2, 3, 4, 5, ...
```

`itertools.repeat(elem, [n])` returns the provided element *n* times, or returns the element endlessly if *n* is not provided.

```
itertools.repeat('abc') =>
  abc, abc, abc, abc, abc, abc, abc, abc, abc, abc, ...
itertools.repeat('abc', 5) =>
  abc, abc, abc, abc, abc
```

`itertools.chain(iterA, iterB, ...)` takes an arbitrary number of iterables as input, and returns all the elements of the first iterator, then all the elements of the second, and so on, until all of the iterables have been exhausted.

```
itertools.chain(['a', 'b', 'c'], (1, 2, 3)) =>
  a, b, c, 1, 2, 3
```

`itertools.islice(iter, [start], stop, [step])` returns a stream that's a slice of the iterator. With a single *stop* argument, it will return the first *stop* elements. If you supply a starting index, you'll get *stop-start* elements, and if you supply a value for *step*, elements will be skipped accordingly. Unlike Python's string and list slicing, you can't use negative values for *start*, *stop*, or *step*.

```
itertools.islice(range(10), 8) =>
  0, 1, 2, 3, 4, 5, 6, 7
itertools.islice(range(10), 2, 8) =>
  2, 3, 4, 5, 6, 7
itertools.islice(range(10), 2, 8, 2) =>
  2, 4, 6
```

`itertools.tee(iter, [n])` replicates an iterator; it returns *n* independent iterators that will all return the contents of the source iterator. If you don't supply a value for *n*, the default is 2. Replicating iterators requires saving some of the contents of the source iterator, so this can consume significant memory if the iterator is large and one of the new iterators is consumed more than the others.

```
itertools.tee( itertools.count() ) =>
   iterA, iterB

where iterA ->
   0, 1, 2, 3, 4, 5, 6, 7, 8, 9, ...

and   iterB ->
   0, 1, 2, 3, 4, 5, 6, 7, 8, 9, ...
```

6.2 Calling functions on elements

The `operator` module contains a set of functions corresponding to Python's operators. Some examples are `operator.add(a, b)` (adds two values), `operator.ne(a, b)` (same as `a != b`), and `operator.attrgetter('id')` (returns a callable that fetches the `.id` attribute).

`itertools.starmap(func, iter)` assumes that the iterable will return a stream of tuples, and calls *func* using these tuples as the arguments:

```
itertools.starmap(os.path.join,
                  [('/bin', 'python'), ('/usr', 'bin', 'java'),
                   ('/usr', 'bin', 'perl'), ('/usr', 'bin', 'ruby')])
=>
  /bin/python, /usr/bin/java, /usr/bin/perl, /usr/bin/ruby
```

6.3 Selecionando elementos

Another group of functions chooses a subset of an iterator's elements based on a predicate.

`itertools.filterfalse(predicate, iter)` is the opposite of `filter()`, returning all elements for which the predicate returns false:

```
itertools.filterfalse(is_even, itertools.count()) =>
  1, 3, 5, 7, 9, 11, 13, 15, ...
```

`itertools.takewhile(predicate, iter)` returns elements for as long as the predicate returns true. Once the predicate returns false, the iterator will signal the end of its results.

```
def less_than_10(x):
    return x < 10

itertools.takewhile(less_than_10, itertools.count()) =>
  0, 1, 2, 3, 4, 5, 6, 7, 8, 9

itertools.takewhile(is_even, itertools.count()) =>
  0
```

`itertools.dropwhile(predicate, iter)` discards elements while the predicate returns true, and then returns the rest of the iterable's results.

```
itertools.dropwhile(less_than_10, itertools.count()) =>
  10, 11, 12, 13, 14, 15, 16, 17, 18, 19, ...

itertools.dropwhile(is_even, itertools.count()) =>
  1, 2, 3, 4, 5, 6, 7, 8, 9, 10, ...
```

`itertools.compress(data, selectors)` takes two iterators and returns only those elements of *data* for which the corresponding element of *selectors* is true, stopping whenever either one is exhausted:

```
itertools.compress([1, 2, 3, 4, 5], [True, True, False, False, True]) =>
   1, 2, 5
```

6.4 Combinatoric functions

The `itertools.combinations(iterable, r)` returns an iterator giving all possible *r*-tuple combinations of the elements contained in *iterable*.

```
itertools.combinations([1, 2, 3, 4, 5], 2) =>
  (1, 2), (1, 3), (1, 4), (1, 5),
  (2, 3), (2, 4), (2, 5),
  (3, 4), (3, 5),
  (4, 5)

itertools.combinations([1, 2, 3, 4, 5], 3) =>
  (1, 2, 3), (1, 2, 4), (1, 2, 5), (1, 3, 4), (1, 3, 5), (1, 4, 5),
  (2, 3, 4), (2, 3, 5), (2, 4, 5),
  (3, 4, 5)
```

The elements within each tuple remain in the same order as *iterable* returned them. For example, the number 1 is always before 2, 3, 4, or 5 in the examples above. A similar function, `itertools.permutations(iterable, r=None)`, removes this constraint on the order, returning all possible arrangements of length *r*:

```
itertools.permutations([1, 2, 3, 4, 5], 2) =>
  (1, 2), (1, 3), (1, 4), (1, 5),
  (2, 1), (2, 3), (2, 4), (2, 5),
  (3, 1), (3, 2), (3, 4), (3, 5),
  (4, 1), (4, 2), (4, 3), (4, 5),
  (5, 1), (5, 2), (5, 3), (5, 4)

itertools.permutations([1, 2, 3, 4, 5]) =>
  (1, 2, 3, 4, 5), (1, 2, 3, 5, 4), (1, 2, 4, 3, 5),
  ...
  (5, 4, 3, 2, 1)
```

If you don't supply a value for *r* the length of the iterable is used, meaning that all the elements are permuted.

Note that these functions produce all of the possible combinations by position and don't require that the contents of *iterable* are unique:

```
itertools.permutations('aba', 3) =>
  ('a', 'b', 'a'), ('a', 'a', 'b'), ('b', 'a', 'a'),
  ('b', 'a', 'a'), ('a', 'a', 'b'), ('a', 'b', 'a')
```

The identical tuple `('a', 'a', 'b')` occurs twice, but the two 'a' strings came from different positions.

The `itertools.combinations_with_replacement(iterable, r)` function relaxes a different constraint: elements can be repeated within a single tuple. Conceptually an element is selected for the first position of each tuple and then is replaced before the second element is selected.

```
itertools.combinations_with_replacement([1, 2, 3, 4, 5], 2) =>
  (1, 1), (1, 2), (1, 3), (1, 4), (1, 5),
  (2, 2), (2, 3), (2, 4), (2, 5),
  (3, 3), (3, 4), (3, 5),
  (4, 4), (4, 5),
  (5, 5)
```

6.5 Agrupando elementos

The last function I'll discuss, `itertools.groupby(iter, key_func=None)`, is the most complicated. `key_func(elem)` is a function that can compute a key value for each element returned by the iterable. If you don't supply a key function, the key is simply each element itself.

`groupby()` collects all the consecutive elements from the underlying iterable that have the same key value, and returns a stream of 2-tuples containing a key value and an iterator for the elements with that key.

```
city_list = [('Decatur', 'AL'), ('Huntsville', 'AL'), ('Selma', 'AL'),
             ('Anchorage', 'AK'), ('Nome', 'AK'),
             ('Flagstaff', 'AZ'), ('Phoenix', 'AZ'), ('Tucson', 'AZ'),
             ...
            ]

def get_state(city_state):
    return city_state[1]

itertools.groupby(city_list, get_state) =>
  ('AL', iterator-1),
  ('AK', iterator-2),
  ('AZ', iterator-3), ...

where
iterator-1 =>
  ('Decatur', 'AL'), ('Huntsville', 'AL'), ('Selma', 'AL')
iterator-2 =>
  ('Anchorage', 'AK'), ('Nome', 'AK')
iterator-3 =>
  ('Flagstaff', 'AZ'), ('Phoenix', 'AZ'), ('Tucson', 'AZ')
```

`groupby()` assumes that the underlying iterable's contents will already be sorted based on the key. Note that the returned iterators also use the underlying iterable, so you have to consume the results of iterator-1 before requesting iterator-2 and its corresponding key.

7 O módulo functools

The `functools` module contains some higher-order functions. A **higher-order function** takes one or more functions as input and returns a new function. The most useful tool in this module is the `functools.partial()` function.

For programs written in a functional style, you'll sometimes want to construct variants of existing functions that have some of the parameters filled in. Consider a Python function `f(a, b, c)`; you may wish to create a new function `g(b, c)` that's equivalent to `f(1, b, c)`; you're filling in a value for one of `f()`'s parameters. This is called "partial function application".

The constructor for `partial()` takes the arguments `(function, arg1, arg2, ..., kwarg1=value1, kwarg2=value2)`. The resulting object is callable, so you can just call it to invoke `function` with the filled-in arguments.

Aqui está um pequeno mas bem realístico exemplo:

```
import functools

def log(message, subsystem):
    """Write the contents of 'message' to the specified subsystem."""
    print('%s: %s' % (subsystem, message))
    ...

server_log = functools.partial(log, subsystem='server')
server_log('Unable to open socket')
```

`functools.reduce(func, iter, [initial_value])` cumulatively performs an operation on all the iterable's elements and, therefore, can't be applied to infinite iterables. *func* must be a function that takes two elements and returns a single value. `functools.reduce()` takes the first two elements A and B returned by the iterator and calculates `func(A, B)`. It then requests the third element, C, calculates `func(func(A, B), C)`, combines this result with the fourth element returned, and continues until the iterable is exhausted. If the iterable returns no values at all, a `TypeError` exception is raised. If the initial value is supplied, it's used as a starting point and `func(initial_value, A)` is the first calculation.

```
>>> import operator, functools
>>> functools.reduce(operator.concat, ['A', 'BB', 'C'])
'ABBC'
>>> functools.reduce(operator.concat, [])
Traceback (most recent call last):
  ...
TypeError: reduce() of empty sequence with no initial value
>>> functools.reduce(operator.mul, [1, 2, 3], 1)
6
>>> functools.reduce(operator.mul, [], 1)
1
```

If you use `operator.add()` with `functools.reduce()`, you'll add up all the elements of the iterable. This case is so common that there's a special built-in called `sum()` to compute it:

```
>>> import functools, operator
>>> functools.reduce(operator.add, [1, 2, 3, 4], 0)
10
>>> sum([1, 2, 3, 4])
10
>>> sum([])
0
```

For many uses of `functools.reduce()`, though, it can be clearer to just write the obvious `for` loop:

```
import functools
# Instead of:
product = functools.reduce(operator.mul, [1, 2, 3], 1)

# You can write:
product = 1
for i in [1, 2, 3]:
    product *= i
```

A related function is `itertools.accumulate(iterable, func=operator.add)`. It performs the same calculation, but instead of returning only the final result, `accumulate()` returns an iterator that also yields each partial result:

```
itertools.accumulate([1, 2, 3, 4, 5]) =>
  1, 3, 6, 10, 15

itertools.accumulate([1, 2, 3, 4, 5], operator.mul) =>
  1, 2, 6, 24, 120
```

7.1 O módulo operator

The `operator` module was mentioned earlier. It contains a set of functions corresponding to Python's operators. These functions are often useful in functional-style code because they save you from writing trivial functions that perform a single operation.

Algumas funcionalidades desse módulo são:

- Math operations: `add()`, `sub()`, `mul()`, `floordiv()`, `abs()`, ...
- Logical operations: `not_()`, `truth()`.
- Bitwise operations: `and_()`, `or_()`, `invert()`.
- Comparisons: `eq()`, `ne()`, `lt()`, `le()`, `gt()`, and `ge()`.
- Object identity: `is_()`, `is_not()`.

Consult the operator module's documentation for a complete list.

8 Pequenas funções e as expressões lambda

When writing functional-style programs, you'll often need little functions that act as predicates or that combine elements in some way.

If there's a Python built-in or a module function that's suitable, you don't need to define a new function at all:

```
stripped_lines = [line.strip() for line in lines]
existing_files = filter(os.path.exists, file_list)
```

If the function you need doesn't exist, you need to write it. One way to write small functions is to use the `lambda` expression. `lambda` takes a number of parameters and an expression combining these parameters, and creates an anonymous function that returns the value of the expression:

```
adder = lambda x, y: x+y

print_assign = lambda name, value: name + '=' + str(value)
```

An alternative is to just use the `def` statement and define a function in the usual way:

```
def adder(x, y):
    return x + y

def print_assign(name, value):
    return name + '=' + str(value)
```

Which alternative is preferable? That's a style question; my usual course is to avoid using `lambda`.

Um dos motivos da minha preferência é que `lambda` é bastante limitado nas funções que pode definir. O resultado deve ser computável como uma única expressão, o que significa que você não pode ter comparações multi-canais `if ... elif ... else` ou `try ... except`. Se você tentar fazer muito em uma declaração `lambda`, você acabará com uma expressão excessivamente complicada que é difícil de ler. Rápido, o que o seguinte código está fazendo?

```
import functools
total = functools.reduce(lambda a, b: (0, a[1] + b[1]), items)[1]
```

Você pode descobrir isso, mas leva tempo para desenredar a expressão para descobrir o que está acontecendo. Usar uma breve instrução de `def` aninhada torna as coisas um pouco melhor:

```
import functools
def combine(a, b):
    return 0, a[1] + b[1]

total = functools.reduce(combine, items)[1]
```

Mas seria o melhor de tudo se eu tivesse usado simplesmente um bucle `for`:

```
total = 0
for a, b in items:
    total += b
```

Ou o `sum()` embutida e uma expressão do gerador:

```
total = sum(b for a, b in items)
```

Muitas utilizações de `functools.reduce()` são mais claras quando escritas como loops `for`.

Fredrik Lundh sugeriu uma vez o seguinte conjunto de regras para refatoração de usos de `lambda`:

1. Escrever uma função lambda.
2. Escreva um comentário explicando o que o lambda faz.
3. Estude o comentário por um tempo e pense em um nome que capture a essência do comentário.
4. Converta a lambda para uma declaração de definição, usando esse nome.
5. Remover o comentário.

Eu realmente gosto dessas regras, mas você está livre para discordar sobre se esse estilo sem lambda é melhor.

9 Histórico de Revisão e Reconhecimentos

O autor agradece as seguintes pessoas por oferecer sugestões, correções e assistência com vários rascunhos deste artigo: Ian Bicking, Nick Coghlan, Nick Efford, Raymond Hettinger, Jim Jewett, Mike Krell, Leandro Lameiro, Jussi Salmela, Collin Winter, Blake Winton.

Versão 0.1: publicado em 30 de junho de 2006.

Versão 0.11: publicado em 1º de julho de 2006. Correções de erros de escrita.

Versão 0.2: publicado em 10 de julho de 2006. Incorporou as seções genexp e listcomp em uma. Correções de erros de escrita.

Versão 0.21: adicionou mais referências sugeridas na lista de correspondência do tutor.

Versão 0.30: Adiciona uma seção no módulo `functional` escrito por Collin Winter; Adiciona seção curta no módulo do operador; Algumas outras edições.

10 Referências

10.1 Geral

Structure and Interpretation of Computer Programs, by Harold Abelson and Gerald Jay Sussman with Julie Sussman. The book can be found at https://mitpress.mit.edu/sicp. In this classic textbook of computer science, chapters 2 and 3 discuss the use of sequences and streams to organize the data flow inside a program. The book uses Scheme for its examples, but many of the design approaches described in these chapters are applicable to functional-style Python code.

https://www.defmacro.org/ramblings/fp.html: A general introduction to functional programming that uses Java examples and has a lengthy historical introduction.

https://pt.wikipedia.org/wiki/Programação_funcional: Informação geral do Wikipédia para descrever programação funcional.

https://pt.wikipedia.org/wiki/Corrotina: Entrada para corrotinas.

https://en.wikipedia.org/wiki/Partial_application: Entry for the concept of partial function application.

https://pt.wikipedia.org/wiki/Currying: Entrada para o conceito de currying.

10.2 Python-specific

https://gnosis.cx/TPiP/: The first chapter of David Mertz's book *Text Processing in Python* discusses functional programming for text processing, in the section titled "Utilizing Higher-Order Functions in Text Processing".

Mertz also wrote a 3-part series of articles on functional programming for IBM's DeveloperWorks site; see part 1, part 2, and part 3,

10.3 Documentação do Python

Documentação para o módulo `itertools`.

Documentation for the `functools` module.

Documentação para o módulo `operator`.

PEP 289: "Gerador de Expressões"

PEP 342: "Coroutines via Enhanced Generators" descreve os novos recursos do gerador no Python 2.5.

Índice

P