Programação Funcionado COMOFAZER

Release 2.7.18

Guido van Rossum
and the Python development team

maio 07, 2020

Python Software Foundation
Email: docs@python.org

Sumário

Autor A. M. Kuchling

Release 0.31

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

This section explains the basic concept of functional programming; if you're just interested in learning about Python language features, skip to the next section.

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

- A Maioria das linguagens de programação são **procedural**: 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 orientados 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.

The designers of some computer languages choose to emphasize one particular approach to programming. This often makes it difficult to write programs that use a different approach. Other languages are multi-paradigm languages that support several different approaches. Lisp, C++, and Python are multi-paradigm; you can write programs or libraries that are largely procedural, object-oriented, or functional in all of these languages. In a large program, different sections might be written using different approaches; the GUI might be object-oriented while the processing logic is procedural or functional, for example.

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.

Some languages are very strict about purity and don't even have assignment statements such as `a=3` or `c = a + b`, but it's difficult to avoid all side effects. Printing to the screen or writing to a disk file are side effects, for example. For example, in Python a `print` statement or a `time.sleep(1)` both return no useful value; they're only called for their side effects of sending some text to the screen or pausing execution for a second.

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, os programas de prova corretas são praticamente impraticáveis e não relevantes para o Python software. 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 intérprete 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.

An iterator is an object representing a stream of data; this object returns the data one element at a time. A Python iterator must support a method called `next()` that takes no arguments and always returns the next element of the stream. If there are no more elements in the stream, `next()` must raise the `StopIteration` exception. Iterators don't have to be finite, though; it's perfectly reasonable to write an iterator that produces an infinite stream of data.

The built-in `iter()` function takes an arbitrary object and tries to return an iterator that will return the object's contents or elements, raising `TypeError` if the object doesn't support iteration. Several of Python's built-in data types support iteration, the most common being lists and dictionaries. An object is called an **iterable** object if you can get an iterator for it.

Você pode experimentar a interface de iteração manualmente:

```
>>> L = [1,2,3]
>>> it = iter(L)
>>> print it
<...iterator object at ...>
>>> it.next()
1
```

(continua na próxima página)

(continuação da página anterior)

```
>>> it.next()
2
>>> it.next()
3
>>> it.next()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
>>>
```

Python expects iterable objects in several different contexts, the most important being the `for` statement. In the statement `for X in Y`, Y must be an iterator or some object for which `iter()` can create an iterator. These two statements are equivalent:

```
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)
```

Built-in functions such as `max()` and `min()` can take a single iterator argument and will return the largest or smallest element. The `"in"` and `"not in"` operators also support iterators: `X in iterator` is true if X is found in the stream returned by the iterator. You'll run into obvious problems if the iterator is infinite; `max()`, `min()` will never return, and if the element X never appears in the stream, the `"in"` and `"not in"` operators won't return either.

Note that you can only go forward in an iterator; there's no way to get the previous element, reset the iterator, or make a copy of it. Iterator objects can optionally provide these additional capabilities, but the iterator protocol only specifies the `next()` method. Functions may therefore consume all of the iterator's output, and if you need to do something different with the same stream, you'll have to create a new iterator.

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.

Calling `iter()` on a dictionary returns an iterator that will loop over the dictionary's keys:

```
>>> 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]
Mar 3
Feb 2
Aug 8
Sep 9
Apr 4
Jun 6
Jul 7
Jan 1
May 5
Nov 11
Dec 12
Oct 10
```

Note that the order is essentially random, because it's based on the hash ordering of the objects in the dictionary.

Applying `iter()` to a dictionary always loops over the keys, but dictionaries have methods that return other iterators. If you want to iterate over keys, values, or key/value pairs, you can explicitly call the `iterkeys()`, `itervalues()`, or `iteritems()` methods to get an appropriate iterator.

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', 'US': 'Washington DC', 'France': 'Paris'}
```

Files also support iteration by calling the `readline()` method until there are no more lines in the file. This means you can read each line of a file like this:

```
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 = set((2, 3, 5, 7, 11, 13))
for i in S:
    print i
```

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 da lista e as expressões do gerador (formulário curto: "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', ...]

# 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 lista de compreensão, 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 regulares 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 regulares 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
```

Any function containing a `yield` keyword is a generator function; this is detected by Python's bytecode compiler which compiles the function specially as a result.

When you call a generator function, it doesn't return a single value; instead it returns a generator object that supports the iterator protocol. On executing the `yield` expression, the generator outputs the value of `i`, similar to a `return` statement. The big difference between `yield` and a `return` statement is that on reaching a `yield` the generator's state of execution is suspended and local variables are preserved. On the next call to the generator's `.next()` method, the function will resume executing.

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

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

You could equally write `for i in generate_ints(5)`, or `a,b,c = generate_ints(3)`.

Inside a generator function, the `return` statement can only be used without a value, and signals the end of the procession of values; after executing a `return` the generator cannot return any further values. `return` with a value, such as `return 5`, is a syntax error inside a generator function. The end of the generator's results can also be indicated by raising `StopIteration` manually, or by just letting the flow of execution fall off the bottom of the function.

You could achieve the effect of generators manually by writing your own class and storing all the local variables of the generator as instance variables. For example, returning a list of integers could be done by setting `self.count` to 0, and having the `next()` method increment `self.count` and return it. However, for a moderately complicated generator, writing a corresponding class can be much messier.

The test suite included with Python's library, `test_generators.py`, contains a number of more interesting examples. Here's one generator that implements an in-order traversal of a tree using generators recursively.

```
# 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 ameaça a outra) e o Passeio do Cavalo (encontrando uma rota que leva um cavaleiro para cada casa de um tabuleiro de xadrez NxN sem visitar nenhum quadrado duas vezes).

4.1 Passando valores para um geador

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 explains the exact rules, which are that a `yield`-expression must always be parenthesized except when it occurs at the top-level expression on the right-hand side of an assignment. This means you can write `val = yield i` but have to use parentheses when there's an operation, as in `val = (yield i) + 12`.)

Values are sent into a generator by calling its `send(value)` method. This method resumes the generator's code and the `yield` expression returns the specified value. If the regular `next()` method is called, the `yield` returns `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)
>>> print it.next()
0
>>> print it.next()
1
>>> print it.send(8)
8
>>> print it.next()
9
>>> print it.next()
Traceback (most recent call last):
  File "t.py", line 15, in <module>
    print it.next()
StopIteration
```

Because `yield` will often be returning `None`, you should always check for this case. Don't just use its value in expressions unless you're sure that the `send()` method will be the only method used to resume your generator function.

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

- `throw(type, value=None, traceback=None)` 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 Built-in

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

Two of Python's built-in functions, `map()` and `filter()`, are somewhat obsolete; they duplicate the features of list comprehensions but return actual lists instead of iterators.

`map(f, iterA, iterB, ...)` returns a list containing `f(iterA[0], iterB[0]), f(iterA[1], iterB[1]), f(iterA[2], iterB[2]), ...`.

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

```
>>> map(upper, ['sentence', 'fragment'])
['SENTENCE', 'FRAGMENT']
```

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

As shown above, you can achieve the same effect with a list comprehension. The `itertools.imap()` function does the same thing but can handle infinite iterators; it'll be discussed later, in the section on the `itertools` module.

`filter(predicate, iter)` returns a list that contains 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
```

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

isso também pode ser escrito como uma compreensão de lista:

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

`filter()` also has a counterpart in the `itertools` module, `itertools.ifilter()`, that returns an iterator and can therefore handle infinite sequences just as `itertools.imap()` can.

`reduce(func, iter, [initial_value])` doesn't have a counterpart in the `itertools` module because it 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. `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
>>> reduce(operator.concat, ['A', 'BB', 'C'])
'ABBC'
>>> reduce(operator.concat, [])
Traceback (most recent call last):
  ...
TypeError: reduce() of empty sequence with no initial value
>>> reduce(operator.mul, [1,2,3], 1)
6
>>> reduce(operator.mul, [], 1)
1
```

If you use `operator.add()` with `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:

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

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

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

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

`enumerate(iter)` counts off the elements in the iterable, returning 2-tuples containing the count 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, [cmp=None], [key=None], [reverse=False])` collects all the elements of the iterable into a list, sorts the list, and returns the sorted result. The `cmp`, `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 Sorting mini-HOWTO in the Python wiki at https://wiki.python.org/moin/HowTo/Sorting.)

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
```

6 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` statement. `lambda` takes a number of parameters and an expression combining these parameters, and creates a small function that returns the value of the expression:

```
lowercase = lambda x: x.lower()

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

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

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

```
def lowercase(x):
    return x.lower()

def print_assign(name, value):
```

(continua na próxima página)

(continuação da página anterior)

```
    return name + '=' + str(value)

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

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

One reason for my preference is that `lambda` is quite limited in the functions it can define. The result has to be computable as a single expression, which means you can't have multiway `if... elif... else` comparisons or `try... except` statements. If you try to do too much in a `lambda` statement, you'll end up with an overly complicated expression that's hard to read. Quick, what's the following code doing?

```
total = 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:

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

total = 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()` incorporado e uma expressão do gerador:

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

Many uses of `reduce()` are clearer when written as `for` loops.

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.

7 The itertools module

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.

7.1 Criando novos iteradores

`itertools.count(n)` returns an infinite stream of integers, increasing by 1 each time. You can optionally supply the starting number, which defaults to 0:

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

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

It's similar to the built-in `zip()` function, but 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.

```
itertools.izip(['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.

`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, ...
```

7.2 Calling functions on elements

Two functions are used for calling other functions on the contents of an iterable.

`itertools.imap(f, iterA, iterB, ...)` returns a stream containing `f(iterA[0], iterB[0])`, `f(iterA[1], iterB[1])`, `f(iterA[2], iterB[2])`, `...`:

```
itertools.imap(operator.add, [5, 6, 5], [1, 2, 3]) =>
  6, 8, 8
```

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 `f()` using these tuples as the arguments:

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

7.3 Selecionando elementos

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

`itertools.ifilter(predicate, iter)` returns all the elements for which the predicate returns true:

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

itertools.ifilter(is_even, itertools.count()) =>
  0, 2, 4, 6, 8, 10, 12, 14, ...
```

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

```
itertools.ifilterfalse(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, ...
```

7.4 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 state

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')
```

(continua na próxima página)

(continuação da página anterior)

```
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.

8 The functools module

The `functools` module in Python 2.5 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')
```

8.1 The operator module

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()`, `div()`, `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.

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. Typo correções.

Versão 0.2: publicado em 10 de julho de 2006. Incorporou as seções genexp e listcomp em uma. Typo corrige.

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

Estrutura e Interpretação de Programas de Computador, de Harold Abelson e Gerald Jay Sussman com Julie Sussman. Texto completo em https://mitpress.mit.edu/sicp/. Neste clássico livro de informática, os capítulos 2 e 3 discutem o uso de sequências e fluxos para organizar o fluxo de dados dentro de um programa. O livro usa o Scheme para seus exemplos, mas muitas das abordagens de design descritas nestes capítulos são aplicáveis ao código Python de estilo funcional.

http://www.defmacro.org/ramblings/fp.html: Uma introdução geral sobre programação funcional que utiliza exemplos Java e tem uma introdução histórica longa.

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

https://pt.wikipedia.org/wiki/Corotina: Entrada para corotinas.

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

10.2 Python-specific

http://gnosis.cx/TPiP/: O primeiro capítulo do livro de David Mertz *Text Processing in Python* onde discute programação funcional para processamento de texto, em uma secção entitulada "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`.

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