函数式编程指引

发布 *2.7.18*

Guido van Rossum
and the Python development team

五月 20, 2020

Python Software Foundation
Email: docs@python.org

Contents

作者 A. M. Kuchling

发布版本 0.31

本文档提供恰当的 Python 函数式编程范例，在函数式编程简单的介绍之后，将简单介绍 Python 中关于函数式编程的特性如 iterator 和 generator 以及相关库模块如 `itertools` 和 `functools` 等。

1 概述

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.

编程语言支持通过以下几种方式来解构具体问题：

- 大多数的编程语言都是 **过程式**的，所谓程序就是一连串告诉计算机怎样处理程序输入的指令。C、Pascal 甚至 Unix shells 都是过程式语言。
- 在 **声明式**语言中，你编写一个用来描述待解决问题的说明，并且这个语言的具体实现会指明怎样高效的进行计算。SQL 可能是你最熟悉的声明式语言了。一个 SQL 查询语句描述了你想要检索的数据集，并且 SQL 引擎会决定是扫描整张表还是使用索引，应该先执行哪些子句等等。
- **面向对象**程序会操作一组对象。对象拥有内部状态，并能够以某种方式支持请求和修改这个内部状态的方法。Smalltalk 和 Java 都是面向对象的语言。C++ 和 Python 支持面向对象编程，但并不强制使用面向对象特性。
- **函数式**编程则将一个问题分解成一系列函数。理想情况下，函数只接受输入并输出结果，对一个给定的输入也不会有影响输出的内部状态。著名的函数式语言有 ML 家族（Standard ML，Ocaml 以及其他变种）和 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.

在函数式程序里，输入会流经一系列函数。每个函数接受输入并输出结果。函数式风格反对使用带有副作用的函数，这些副作用会修改内部状态，或者引起一些无法体现在函数的返回值中的变化。完全不产生副作用的函数被称作“纯函数”。消除副作用意味着不能使用随程序运行而更新的数据结构；每个函数的输出必须只依赖于输入。

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.

函数式风格的 Python 程序并不会极端到消除所有 I/O 或者赋值的程度；相反，他们会提供像函数式一样的接口，但会在内部使用非函数式的特性。比如，函数的实现仍然会使用局部变量，但不会修改全局变量或者有其他副作用。

函数式编程可以被认为是面向对象编程的对立面。对象就像是颗小胶囊，包裹着内部状态和随之而来的能让你修改这个内部状态的一组调用方法，以及由正确的状态变化所构成的程序。函数式编程希望尽可能地消除状态变化，只和流经函数的数据打交道。在 Python 里你可以把两种编程方式结合起来，在你的应用（电子邮件信息，事务处理）中编写接受和返回对象实例的函数。

函数式设计在工作中看起来是个奇怪的约束。为什么你要消除对象和副作用呢？不过函数式风格有其理论和实践上的优点：

- 形式证明。
- 模块化。
- 组合性。
- 易于调试和测试。

1.1 形式证明

一个理论上的优点是，构造数学证明来说明函数式程序是正确的相对更容易些。

很长时间，研究者们对寻找证明程序正确的数学方法都很感兴趣。这和通过大量输入来测试，并得出程序的输出基本正确，或者阅读一个程序的源代码然后得出代码看起来没问题不同；相反，这里的目标是一个严格的证明，证明程序对所有可能的输入都能给出正确的结果。

证明程序正确性所用到的技术是写出 **不变量**，也就是对于输入数据和程序中的变量永远为真的特性。然后对每行代码，你说明这行代码执行前的不变量 X 和 Y 以及执行后稍有不同的不变量 X’ 和 Y’ 为真。如此一直到程序结束，这时候在程序的输出上，不变量应该会与期望的状态一致。

函数式编程之所以要消除赋值，是因为赋值在这个技术中难以处理；赋值可能会破坏赋值前为真的不变量，却并不产生任何可以传递下去的新的不变量。

不幸的是，证明程序的正确性很大程度上是经验性质的，而且和 Python 软件无关。即使是微不足道的程序都需要几页长的证明；一个中等复杂的程序的正确性证明会非常庞大，而且，极少甚至没有你日常所使用的程序（Python 解释器，XML 解析器，浏览器）的正确性能够被证明。即使你写出或者生成一个证明，验证证明也会是一个问题；里面可能出了差错，而你错误地相信你证明了程序的正确性。

1.2 模块化

函数式编程的一个更实用的优点是，它强制你把问题分解成小的方面。因此程序会更加模块化。相对于一个进行了复杂变换的大型函数，一个小的函数更明确，更易于编写，也更易于阅读和检查错误。

1.3 易于调试和测试

测试和调试函数式程序相对来说更容易。

调试很简单是因为函数通常都很小而且清晰明确。当程序无法工作的时候，每个函数都是一个可以检查数据是否正确的接入点。你可以通过查看中间输入和输出迅速找到出错的函数。

测试更容易是因为每个函数都是单元测试的潜在目标。在执行测试前，函数并不依赖于需要重现的系统状态；相反，你只需要给出正确的输入，然后检查输出是否和期望的结果一致。

1.4 组合性

当你编写函数式风格的程序时，你会写出很多带有不同输入和输出的函数。其中一些不可避免地会局限于特定的应用，但其他的却可以广泛的用在程序中。举例来说，一个接受文件夹目录返回所有文件夹中的 XML 文件的函数；或是一个接受文件名，然后返回文件内容的函数，都可以应用在很多不同的场合。

久而久之你会形成一个个人工具库。通常你可以重新组织已有的函数来组成新的程序，然后为当前的工作写一些特殊的函数。

2 迭代器

我会从 Python 的一个语言特性，编写函数式风格程序的重要基石开始说起：迭代器。

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.

你可以手动试验迭代器的接口。

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

可以用 `list()` 或 `tuple()` 这样的构造函数把迭代器具体化成列表或元组:

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

(下页继续)

(续上页)

```
>>> t
(1, 2, 3)
```

序列的解压操作也支持迭代器：如果你知道一个迭代器能够返回 N 个元素，你可以把他们解压到有 N 个元素的元组:

```
>>> 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 支持迭代器的数据类型

我们已经知道列表和元组支持迭代器。实际上，Python 中的任何序列类型，比如字符串，都自动支持创建迭代器。

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.

`dict()` 构造函数可以接受一个迭代器，然后返回一个有限的 `(key, value)` 元组的数据流:

```
>>> 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
    ...
```

集合可以从可遍历的对象获取内容，也可以让你遍历集合的元素:

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

3 生成器表达式和列表推导式

迭代器的输出有两个很常见的使用方式，1) 对每一个元素执行操作，2) 选择一个符合条件的元素子集。比如，给定一个字符串列表，你可能想去掉每个字符串尾部的空白字符，或是选出所有包含给定子串的字符串。

列表推导式和生成器表达时（简写:" listcomps" 和 "genexps"）让这些操作更加简明，这个形式借鉴自函数式程序语言 Haskell（https://www.haskell.org/）。你可以用以下代码去掉一个字符串流中的所有空白字符:

```
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]
```

你可以加上条件语句 `"if"` 来选取特定的元素:

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

通过列表推导式，你会获得一个 Python 列表；`stripped_list` 就是一个包含所有结果行的列表，并不是迭代器。生成器表达式会返回一个迭代器，它在必要的时候计算结果，避免一次性生成所有的值。这意味着，如果迭代器返回一个无限数据流或者大量的数据，列表推导式就不太好用了。这种情况下生成器表达式会更受青睐。

生成器表达式两边使用圆括号（"()"），而列表推导式则使用方括号（"[]"）。生成器表达式的形式为:

```
( 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 )
```

再次说明，列表推导式只有两边的括号不一样（方括号而不是圆括号）。

这些生成用于输出的元素会成为 expression 的后继值。其中 if 语句是可选的；如果给定的话 expression 只会在符合条件时计算并加入到结果中。

生成器表达式总是写在圆括号里面，不过也可以算上调用函数时用的括号。如果你想即时创建一个传递给函数的迭代器，可以这么写:

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

其中 for...in 语句包含了将要遍历的序列。这些序列并不必须同样长，因为它们会从左往右开始遍历，而 **不是**同时执行。对每个 sequence1 中的元素，sequence2 会从头开始遍历。sequence3 会对每个 sequence1 和 sequence2 的元素对开始遍历。

换句话说，列表推导式器是和下面的 Python 代码等价:

```
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.
```

这说明，如果有多个 for...in 语句而没有 if 语句，输出结果的长度就是所有序列长度的乘积。如果你的两个列表长度为 3，那么输出的列表长度就是 9:

```
>>> 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)]
```

为了不让 Python 语法变得含糊，如果 expression 会生成元组，那这个元组必须要用括号括起来。下面第一个列表推导式语法错误，第二个则是正确的:

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

4 生成器

生成器是一类用来简化编写迭代器工作的特殊函数。普通的函数计算并返回一个值，而生成器返回一个能返回数据流的迭代器。

毫无疑问，你已经对如何在 Python 和 C 中调用普通函数很熟悉了，这时候函数会获得一个创建局部变量的私有命名空间。当函数到达 return 表达式时，局部变量会被销毁然后把返回给调用者。之后调用同样的函数时会创建一个新的私有命名空间和一组全新的局部变量。但是，如果在退出一个函数时不扔掉局部变量会如何呢？如果稍后你能够从退出函数的地方重新恢复又如何呢？这就是生成器所提供的；他们可以被看成可恢复的函数。

这里有简单的生成器函数示例:

```
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.

这里有一个 `generate_ints()` 生成器的示例:

```
>>> 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
```

另外两个 `test_generators.py` 中的例子给出了 N 皇后问题（在 NxN 的棋盘上放置 N 个皇后，任何一个

都不能吃掉另一个），以及马的遍历路线（在 NxN 的棋盘上给马找出一条不重复的走过所有格子的路线）的解。

4.1 向生成器传递值

在 Python 2.4 及之前的版本中，生成器只产生输出。一旦调用生成器的代码创建一个迭代器，就没有办法在函数恢复执行的时候向它传递新的信息。你可以设法实现这个功能，让生成器引用一个全局变量或者一个调用者可以修改的可变对象，但是这些方法都很繁杂。

在 Python 2.5 里有一个简单的将值传递给生成器的方法。`yield` 变成了一个表达式，返回一个可以赋给变量或执行操作的值:

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

我建议你在处理 `yield` 表达式返回值的时候，**总是**两边写上括号，就像上面的例子一样。括号并不总是必须的，但是比起记住什么时候需要括号，写出来会更容易一点。

(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`.

这里有一个简单的每次加 1 的计数器，并允许改变内部计数器的值。

```
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
```

这是改变计数器的一个示例

```
>>> 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.

  如果你要在 `GeneratorExit` 发生的时候清理代码，我建议使用 `try: ... finally:` 组合来代替 `GeneratorExit`。

这些改变的累积效应是，让生成器从单向的信息生产者变成了既是生产者，又是消费者。

生成器也可以成为 **协程**，一种更广义的子过程形式。子过程可以从一个地方进入，然后从另一个地方退出（从函数的顶端进入，从 `return` 语句退出），而协程可以进入，退出，然后在很多不同的地方恢复（`yield` 语句）。

5 内置函数

我们可以看看迭代器常常用到的函数的更多细节。

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]
```

这也可以写成列表推导式:

```
>>> [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()` 常常用于遍历列表并记录达到特定条件时的下标:

```
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 小函数和 lambda 表达式

编写函数式风格程序时，你会经常需要很小的函数，作为谓词函数或者以某种方式来组合元素。

如果合适的 Python 内置的或者其他模块中的函数，你就一点也不需要定义新的函数:

```
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
```

另一种替代方案就是通常的使用 `def` 语句来定义函数:

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

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

(下页继续)

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

哪一种更受青睐呢？这是一个风格问题；我通常的做法是避免使用 `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]
```

你可以弄明白，不过要花上时间来理清表达式来搞清楚发生了什么。使用一个简短的嵌套的 `def` 语句可以让情况变得更好:

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

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

如果我仅仅使用一个 `for` 循环会更好:

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

或者使用内置的 `sum()` 和一个生成器表达式:

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

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

Fredrik Lundh 曾经建议以下一组规则来重构 `lambda` 的使用:

1) 写一个 lambda 函数。
2) 写一句注释来说明这个 lambda 究竟干了什么。
3) 研究一会这个注释，然后想出一个抓住注释本质的名字。
4) 用这个名字，把这个 lambda 改写成 def 语句。
5) 把注释去掉。

我非常喜欢这些规则，不过你完全有权利争辩这种消除 lambda 的风格是不是更好。

7 itertools 模块

`itertools` 模块包含很多常用的迭代器以及用来组合迭代器的函数。本节会用些小的例子来介绍这个模块的内容。

这个模块里的函数大致可以分为几类:

- 从已有的迭代器创建新的迭代器的函数。
- 接受迭代器元素作为参数的函数。
- 选取部分迭代器输出的函数。

- 给迭代器输出分组的函数。

7.1 创建新的迭代器

`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.)

这个迭代器设计用于长度相同的可迭代对象。如果可迭代对象的长度不一致，返回的数据流的长度会和最短的可迭代对象相同

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

然而，你应该避免这种情况，因为所有从更长的迭代器中取出的元素都会被丢弃。这意味着之后你也无法冒着跳过被丢弃元素的风险来继续使用这个迭代器。

`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 对元素使用函数

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 选择元素

另外一系列函数根据谓词选取一个迭代器中元素的子集。

`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 为元素分组

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')
```

(下页继续)

(续上页)

```
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 functools 模块

Python 2.5 中的 `functools` 模块包含了一些高阶函数。**高阶函数**接受一个或多个函数作为输入，返回新的函数。这个模块中最有用的工具是 `functools.partial()` 函数。

对于用函数式风格编写的程序，有时你会希望通过给定部分参数，将已有的函数构变形称新的函数。考虑一个 Python 函数 `f(a, b, c)`；你希望创建一个和 `f(1, b, c)` 等价的新函数 `g(b, c)`；也就是说你给定了 `f()` 的一个参数的值。这就是所谓的“部分函数应用”。

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.

这里有一个很小但很现实的例子:

```
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 operator 模块

前面已经提到了 `operator` 模块。它包含一系列对应于 Python 操作符的函数。在函数式风格的代码中，这些函数通常很有用，可以帮你省下不少时间，避免写一些琐碎的仅仅执行一个简单操作的函数。

这个模块里的一些函数:

- Math operations: `add()`, `sub()`, `mul()`, `div()`, `floordiv()`, `abs()`, ···
- 逻辑运算: `not_()`, `truth()`。
- 位运算: `and_()`, `or_()`, `invert()`。
- 比较: `eq()`, `ne()`, `lt()`, `le()`, `gt()`, 和 `ge()`。
- 确认对象: `is_()`, `is_not()`。

全部函数列表可以参考 operator 模块的文档。

9 修订记录和致谢

作者要感谢以下人员对本文各种草稿给予的建议，更正和协助：Ian Bicking,Nick Coghlan, Nick Efford, Raymond Hettinger, Jim Jewett, Mike Krell,Leandro Lameiro, Jussi Salmela, Collin Winter, Blake Winton。

0.1 版: 2006 年 6 月 30 日发布。

0.11 版: 2006 年 7 月 1 日发布。修正拼写错误。

0.2 版: 2006 年 7 月 10 日发布。将 genexp 与 listcomp 两节合二为一。修正拼写错误。

0.21 版: 加入了 tutor 邮件列表中建议的更多参考文件。

0.30 版: 添加了有关 `functional` 模块的小节，由 Collin Winter 撰写；添加了有关 operator 模块的简短小节；其他少量修改。

10 参考文献

10.1 通用文献

Structure and Interpretation of Computer Programs, Harold Abelson, Gerald Jay Sussman 和 Julie Sussman 著。全文可见 https://mitpress.mit.edu/sicp/ 。在这部计算机科学的经典教科书中，第二和第三章讨论了使用序列和流来组织程序内部的数据传递。书中的示例采用 Scheme 语言，但其中这些章节中描述的很多设计方法同样适用于函数式风格的 Python 代码。

http://www.defmacro.org/ramblings/fp.html: 一个使用 Java 示例的函数式编程的总体介绍，有很长的历史说明。

https://en.wikipedia.org/wiki/Functional_programming: 一般性的函数式编程的 Wikipedia 条目。

https://en.wikipedia.org/wiki/Coroutine: 协程条目。

https://en.wikipedia.org/wiki/Currying: 函数柯里化条目。

10.2 Python 相关

http://gnosis.cx/TPiP/：David Mertz 书中的第一章 *Text Processing in Python*，”Utilizing Higher-Order Functions in Text Processing”标题部分讨论了文本处理的函数式编程。

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 Python 文档

`itertools` 模块文档。

`operator` 模块文档。

PEP 289: “Generator Expressions”

PEP 342: “Coroutines via Enhanced Generators”描述了 Python 2.5 中新的生成器特性。

索引

P