# 函数式编程指引

**发布 *3.6.8***

**Guido van Rossum**
**and the Python development team**

**五月 10, 2019**

**Python Software Foundation**
**Email: docs@python.org**

## Contents

---

**作者** A. M. Kuchling

**发布版本** 0.32

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

本章介绍函数式编程的基本概念。如您仅想学习 Python 语言的特性，可跳过本章直接查看迭代器.

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。

一些语言的设计者选择强调一种特定的编程方式。这通常会让以不同的方式来编写程序变得困难。其他多范式语言则支持几种不同的编程方式。Lisp，C++ 和 Python 都是多范式语言；使用这些语言，你可以编写主要为过程式，面向对象或者函数式的程序和函数库。在大型程序中，不同的部分可能会采用不同的方式编写；比如 GUI 可能是面向对象的而处理逻辑则是过程式或者函数式。

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。

一些语言对纯洁性要求非常严格，以至于没有像 `a=3` 或 `c = a + b` 这样的赋值表达式，但是完全消除副作用非常困难。比如，显示在屏幕上或者写到磁盘文件中都是副作用。举个例子，在 Python 里，调用函数 `print()` 或者 `time.sleep()` 并不会返回有用的结果；它们的用途只在于副作用，向屏幕发送一段文字或暂停一秒钟。

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。

迭代器是一个表示数据流的对象；这个对象每次只返回一个元素。Python 迭代器必须支持 `__next__()` 方法；这个方法不接受参数，并总是返回数据流中的下一个元素。如果数据流中没有元素，`__next__()` 会抛出 `StopIteration` 异常。迭代器未必是有限的；完全有理由构造一个输出无限数据流的迭代器。

内置的 `iter()` 函数接受任意对象并试图返回一个迭代器来输出对象的内容或元素，并会在对象不支持迭代的时候抛出 `TypeError` 异常。Python 有几种内置数据类型支持迭代，最常见的就是列表和字典。如果一个对象能生成迭代器，那么它就会被称作 iterable。

你可以手动试验迭代器的接口。

```
>>> L = [1, 2, 3]
>>> it = iter(L)
>>> it  #doctest: +ELLIPSIS
<...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 有不少要求使用可迭代的对象的地方，其中最重要的就是 `for` 表达式。在表达式 `for X in Y`，Y 要么自身是一个迭代器，要么能够由 `iter()` 创建一个迭代器。以下两种表达是等价的:

```
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)
>>> 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)
```

像 max() 和 min() 这样的内置函数可以接受单个迭代器参数，然后返回其中最大或者最小的元素。"in" 和 "not in" 操作也支持迭代器：如果能够在迭代器 iterator 返回的数据流中找到 X 的话，则 "X in iterator" 为真。很显然，如果迭代器是无限的，这么做你就会遇到问题；max() 和 min() 永远也不会返回；如果元素 X 也不出现在数据流中，"in" 和 "not in" 操作同样也永远不会返回。

注意你只能在迭代器中顺序前进；没有获取前一个元素的方法，除非重置迭代器，或者重新复制一份。迭代器对象可以提供这些额外的功能，但迭代器协议只明确了 __next__() 方法。函数可能因此而耗尽迭代器的输出，如果你要对同样的数据流做不同的操作，你必须重新创建一个迭代器。

## 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。

对字典调用 iter() 会返回一个遍历字典的键的迭代器:

```
>>> 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.

对字典使用 iter() 总是会遍历键，但字典也有返回其他迭代器的方法。如果你只遍历值或者键/值对，你可以明确地调用 values() 或 items() 方法得到合适的迭代器。

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))  #doctest: +SKIP
{'Italy': 'Rome', 'US': 'Washington DC', 'France': 'Paris'}
```

文件也可以通过调用 readline() 来遍历，直到穷尽文件中所有的行。这意味着你可以像这样读取文件中的每一行:

```
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 = {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]  #doctest: +NORMALIZE_WHITESPACE
[('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
```

任何包含了 yield 关键字的函数都是生成器函数；Python 的 bytecode 编译器会在编译的时候检测到并因此而特殊处理。

当你调用一个生成器函数，它并不会返回单独的值，而是返回一个支持生成器协议的生成器对象。当执行 yield 表达式时，生成器会输出 i 的值，就像 return 表达式一样。yield 和 return 最大的区别在于，到达 yield 的时候生成器的执行状态会挂起并保留局部变量。在下一次调用生成器 __next__() 方法的时候，函数会恢复执行。

这里有一个 generate_ints() 生成器的示例:

```
>>> gen = generate_ints(3)
>>> gen  #doctest: +ELLIPSIS
<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
```

同样，你可以写出 for i in generate_ints(5)，或者 a, b, c = generate_ints(3)。

在生成器函数里面，return value 会触发从 __next__() 方法抛出 StopIteration(value) 异常。一旦抛出这个异常，或者函数结束，处理数据的过程就会停止，生成器也不会再生成新的值。

你可以手动编写自己的类来达到生成器的效果，把生成器的所有局部变量作为实例的成员变量存储起来。比如，可以这么返回一个整数列表：把 self.count 设为 0，然后通过 count`()。然而，对于一个中等复杂程度的生成器，写出一个相应的类可能会相当繁杂。

包含在 Python 库中的测试套件 Lib/test/test_generators.py 里有很多非常有趣的例子。这里是一个用生成器实现树的递归中序遍历示例。:

```
# 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** 解释了具体的规则，也就是 `yield` 表达式必须括起来，除非是出现在最顶级的赋值表达式的右边。这意味着你可以写 `val = yield i`，但是必须在操作的时候加上括号，就像 "val = (yield i) + 12")

可以调用 `send(value)()` <generator.send> 方法向生成器发送值。这个方法会恢复执行生成器的代码，然后 `yield` 表达式返回特定的值。如果调用普通的 `__next__`方法，``yield`()` 会返回 `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)  #doctest: +SKIP
>>> next(it)  #doctest: +SKIP
0
>>> next(it)  #doctest: +SKIP
1
>>> it.send(8)  #doctest: +SKIP
8
>>> next(it)  #doctest: +SKIP
9
>>> next(it)  #doctest: +SKIP
Traceback (most recent call last):
  File "t.py", line 15, in <module>
    it.next()
StopIteration
```

因为 `yield` 很多时候会返回 `None`，所以你应该总是检查这个情况。不要在表达式中使用 `yield` 的值，除非你确定 `send()` 是唯一的用来恢复你的生成器函数的方法。

除了 `send()` 之外，生成器还有两个其他的方法:

- `throw(type, value=None, traceback=None)` 用于在生成器内部抛出异常；这个异常会在生成器暂停执行的时候由 `yield` 表达式抛出。

- `generator.close()` 会在生成器内部抛出 `GeneratorExit` 异常来结束迭代。当接收到这个异常时，生成器的代码会抛出 `GeneratorExit` 或者 `StopIteration`；捕捉这个异常作其他处理是非法的，并会出发 `RuntimeError`。`close()` 也会在 Python 垃圾回收器回收生成器的时候调用。

  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。

Python 内置的两个函数 `map()` 和 `filter()` 复制了生成器表达式的两个特性:

`map(f, iterA, iterB, ...)` **返回一个遍历序列的迭代器** `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']
```

你当然也可以用列表推导式达到同样的效果。

`filter(predicate, iter)` 返回一个遍历序列中满足指定条件的元素的迭代器，和列表推导式的功能相似。**predicate** （谓词）是一个在特定条件下返回真值的函数；要使用函数 `filter()`，谓词函数必须只能接受一个参数。

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

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

这也可以写成列表推导式:

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

`enumerate(iter, start=0)` 计数可迭代对象中的元素，然后返回包含每个计数（从 **start** 开始）和元素两个值的元组。:

```
>>> 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, key=None, reverse=False)` 会将 iterable 中的元素收集到一个列表中，然后排序并返回结果。其中 *key* 和 *reverse* 参数会传递给所创建列表的 `sort()` 方法。:

```
>>> 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]
```

(对排序更详细的讨论可参见 sortinghowto。)

内置函数 `any(iter)` 和 `all(iter)` 会查看一个可迭代对象内容的逻辑值。`any()` 在可迭代对象中任意一个元素为真时返回 `True`，而 `all()` 在所有元素为真时返回 `True`:

```
>>> 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, ...)` 从每个可迭代对象中选取单个元素组成列表并返回:

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

它并不会在内存创建一个列表并因此在返回前而耗尽输入的迭代器；相反，只有在被请求的时候元组才会创建并返回。(这种行为的技术术语叫惰性计算，参见 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

```
zip(['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。

# 6 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。

## 6.1 创建新的迭代器

itertools.count(start, step) 返回一个等分的无限数据流。初始值默认为 0，间隔默认为 1，你也选择可以指定初始值和间隔:

```
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) 保存一份所提供的可迭代对象的副本，并返回一个能产生整个可迭代对象序列的新迭代器。新迭代器会无限重复这些元素。:

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

itertools.repeat(elem, [n]) 返回 $n$ 次所提供的元素，当 $n$ 不存在时，返回无数次所提供的元素。

```
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, ...) 接受任意数量的可迭代对象作为输入，首先返回第一个迭代器的所有元素，然后是第二个的所有元素，如此一直进行下去，直到消耗掉所有输入的可迭代对象。

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

itertools.islice(iter, [start], stop, [step]) 返回一个所输入的迭代器切片的数据流。如果只单独给定 *stop* 参数的话，它会返回从起始算起 *stop* 个数量的元素。如果你提供了起始下标 *start*，你会得到 *stop-start* 个元素；如果你给定了 *step* 参数，数据流会跳过相应的元素。和 Python 里的字符串和列表切片不同，你不能在 *start*, *stop* 或者 *step* 这些参数中使用负数。:

```
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
```

(下页继续)

(续上页)

```
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 itertools.tee(iter, [n]) 可以复制一个迭代器；它返回 *n* 个能够返回源迭代器内容的独立迭代器。如果你不提供参数 *n*，默认值为 2。复制迭代器需要保存源迭代器的一部分内容，因此在源迭代器比较大的时候会显著地占用内存；同时，新迭代器中的一个会比其他迭代器占用更多的内存。

operator 模块包含一组对应于 Python 操作符的函数。比如 `operator.add(a, b)`（把两个数加起来），`operator.ne(a, b)`（和 `a != b` 相同），以及 `operator.attrgetter('id')`（返回获取 `.id` 属性的可调用对象）。

`itertools.starmap(func, iter)` 假定可迭代对象能够返回一个元组的流，并且利用这些元组作为参数来调用 *func*:

```
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 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.filterfalse(predicate, iter)` 和 `filter()` 相反，返回所有让 predicate 返回 false 的元素:

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

`itertools.takewhile(predicate, iter)` 返回一直让 predicate 返回 true 的元素。一旦 predicate 返回 false，迭代器就会发出终止结果的信号。:

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

(下页继续)

(续上页)

```
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)` 在 predicate 返回 true 的时候丢弃元素，并且返回可迭代对象的剩余结果。:

```
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)` 接受两个迭代器，然后返回 *data* 中使相应地 *selector* 中的元素为真的元素；它会在任一个迭代器耗尽的时候停止:

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

## 6.4 组合函数

`itertools.combinations(iterable, r)` 返回一个迭代器，它能给出输入迭代器中所包含的元素的所有可能的 $r$ 元元组的组合。:

```
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)
```

每个元组中的元素保持着 可迭代对象返回他们的顺序。例如，在上面的例子中数字 1 总是会在 2, 3, 4 或 5 前面。一个类似的函数，`itertools.permutations(iterable, r=None)`，取消了保持顺序的限制，返回所有可能的长度为 $r$ 的排列:

```
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)
```

如果你不提供 $r$ 参数的值，它会使用可迭代对象的长度，也就是说会排列所有的元素。

注意这些函数会输出所有可能的位置组合，并不要求 可迭代对象的内容不重复:

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

同一个元组 `('a', 'a', 'b')` 出现了两次，但是两个‘a’字符来自不同的位置。

`itertools.combinations_with_replacement(iterable, r)` 函数放松了一个不同的限制：元组中的元素可以重复。从概念讲，为每个元组第一个位置选取一个元素，然后在选择第二个元素前替换掉它。:

```
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 Grouping elements

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 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.

Here' s a small but realistic example:

```
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])
```

(下页继续)

(续上页)

```
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 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.

Some of the functions in this module are:

- 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 Small functions and the lambda expression

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 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`.

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?

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

You can figure it out, but it takes time to disentangle the expression to figure out what's going on. Using a short nested `def` statements makes things a little bit better:

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

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

But it would be best of all if I had simply used a `for` loop:

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

Or the `sum()` built-in and a generator expression:

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

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

Fredrik Lundh once suggested the following set of rules for refactoring uses of `lambda`:

1. Write a lambda function.
2. Write a comment explaining what the heck that lambda does.
3. Study the comment for a while, and think of a name that captures the essence of the comment.

4. Convert the lambda to a def statement, using that name.
5. Remove the comment.

I really like these rules, but you’ re free to disagree about whether this lambda-free style is better.

# 9 Revision History and Acknowledgements

The author would like to thank the following people for offering suggestions, corrections and assistance with various drafts of this article: Ian Bicking, Nick Coghlan, Nick Efford, Raymond Hettinger, Jim Jewett, Mike Krell, Leandro Lameiro, Jussi Salmela, Collin Winter, Blake Winton.

Version 0.1: posted June 30 2006.

Version 0.11: posted July 1 2006. Typo fixes.

Version 0.2: posted July 10 2006. Merged genexp and listcomp sections into one. Typo fixes.

Version 0.21: Added more references suggested on the tutor mailing list.

Version 0.30: Adds a section on the `functional` module written by Collin Winter; adds short section on the operator module; a few other edits.

# 10 引用文献

## 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 还在 IBM 的 DeveloperWorks 站点上关于函数式编程写了一系列共 3 篇文章；参见 ‘part 1<https://www.ibm.com/developerworks/linux/library/l-prog/index.html>‘___,‘part2<https://www.ibm.com/developerworks/linux/library/l-prog2/index.html>‘___, 和 ‘part 3<https://www.ibm.com/developerworks/linux/library/l-prog3/index.html>‘___,

## 10.3 Python 文档

`itertools` 模块文档。

`functools` 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