FP-in-Python Documentation

发布

Hanaasagi

5月 29, 2017

Table of Contents:

CHAPTER 1

(Avoiding)Flow Control

在典型的 Python 命令式编程(包括将命令式代码存放于类和方法)中， 代码块通常包含一些循环 (`for` or `while`)，状态变量的赋值， `dict`, `list`, `set`, 及各种各样来自于标准库或第三方包的 数据结构的修改，与分支操作(`if`/`elif`/`else`, `try`/`except`/`finally`)。所有的这些都是自然的，下意识便能够理解。 然而那些经常出现的问题，恰恰来自于这些状态变量，可变数据结构所带来的副作用； 他们虽然在模拟物理世界的概念，却难以准确解释程序中指定点处的状态数据

一个解决方案是不要专注于如何构建数据集合，而是描述数据集合中包含了什么。 当人们简单地认为：“这里有一些数据，我需要用它来做什么”时，通常应当尽可能推导， 而不是着急如何构建数据。最后一段所描述的命令式编程更多的关于 “How”， 而不是 “What”，我们可以经常在这两者中进行转移。

封装

一个将注意力放在 “how” 而不是 “what” 的明显途径是重构代码，将数据构建 放在一个相对隔离的地方，比如函数或方法内部。举个例子，考虑想下面这样的 命令式代码片段:

```
# configure the data to start with
collection = get_initial_state()
state_var = None
for datum in data_set:
    if condition(state_var):
        state_var = calculate_from(datum)
        new = modify(datum, state_var)
        collection.add_to(new)
    else:
        new = modify_differently(datum)
        collection.add_to(new)

# Now actually work with the data
for thing in collection
    process(thing)
```

我们可以简单地从当前的作用域中移除数据的构建(“how”)，然后将他们挤进一个 可以独立考虑的函数中（进行充分的抽象）。 例如:

```
# tuck away construction of data
def make_collection(data_set):
    collection = get_initial_state()
    state_var = None
    for datum in data_set:
        if condition(state_var):
            state_var = calculate_from(datum, state_var)
            new = modify(datum, state_var)
            collection.add_to(new)
        else:
            new = modify_differently(datum)
            collection.add_to(new)
    return collection
# Now actually work with the data
for thing in make_collection(data_set):
    process(thing)
```

我们没有改变程序的逻辑，但我们已经将焦点从如何构造 `collection` 转移到 `make_collection()` 创建了什么

推导

使用推导通常是一个既能使代码更紧凑，也能将我们的焦点从 “how” 转移到 “what” 的方式。 推导是一个使用和循环与分支操作相同关键字的表达式，但将注意力从过程转移到了数据本身。 简单地改写表达的形式一般会使我们对代码的理解以及理解的容易程度有很大的不同。三位运算符 (if-else 表达式)使用了不同顺序的关键字，也表现出了类似的对于关注点的重组。假设我们的代码是:

```
collection = list()
for datum in data_set:
    if condition(datum):
        collection.append(datum)
    else:
        new = modify(datum)
        collection.append(new)
```

我们可以写成更紧凑的形式:

```
collection = [d if condition(d) else modify(d)
              for d in data_set]
```

比起简单地保存几个字符和行更重要的是通过思考何为 `collection` 所带来的心理转变， 避免去考虑或调试 `collection` 在循环中某一处的状态。

列表推导是 Python 中最长的(longest???)，在某些方面是最简易的。 Python 现在也支持生成器推导， 集合推导和字典推导语法。作为一个警告，虽然你可以将推导进行任意深度的嵌套，但越过某个界限后， 往往会开始模糊。对于真正复杂的数据集合构建，重构成为函数，可读性会更高。

生成器

生成器推导和列表推导虽然有着相似的语法(生成器不使用方括号包裹，在某些上下文中需要使用圆括号包裹)， 但生成器是惰性的。也就是说，若没有调用对象的 *.next()* 方法，它们仅仅是对于如何获取数据的一种描述。 直到真正地被需要前，节省了长序列和延迟计算的内存。例如:

```
log_lines = (line for line in read_line(huge_log_file)
                  if complex_condition(line))
```

生成器推导也能够构造列表，但运行时更加友好。生成器推导也有着命令式的版本:

```
def get_log_lines(log_file):
    line = read_line(log_file)
    while True:
        try:
            if complex_condition(line):
                yield line
            line = read_line(log_file)
        except StopIteration:
            raise

log_lines = get_log_lines(huge_log_file)
```

命令式的版本也可以简化，但展示这个版本是为了举例说明 `for` 循环迭代一个可迭代对象的 幕后机制，那些我们从想法中抽象出的诸多细节。事实上，使用 `yield` 也是一种对于底层 迭代器协议(iterator protocol)的抽象。我们也能使用实现 `.__next__()` 和 `.__iter__()` 方法的类来达成目的:

```
class GetLogLines(object):

    def __init__(self, log_file):
        self.log_file = log_file
        self.line = None

    def __iter__(self):
        return self

    def __next__(self):
        if self.line is None:
            self.line = read_line(log_file)
        while not complex_condition(self.line):
            self.line = read_line(self.log_file)
        return self.line

log_lines = GetLogLines(huge_log_file)
```

抛开离题的迭代器协议和惰性部分，读者应当看到推导将注意力更多地集中到 “What”，而命令式版本尽管重构的非常成功，也依然在关注 “How”。

字典和集合

列表可以直接在推导中创建而不是先创建一个空列表，然后通过循环重复调用 `.append()`；类似地， 字典和集合也能够一次性创建，而不是在循环中重复调用 `.update()` 或 `.add()`，例如:

```
>>> {i:chr(65+i) for i in range(6)}
{0: 'A', 1: 'B', 2: 'C', 3: 'D', 4: 'E', 5: 'F'}
>>> {chr(65+i) for i in range(6)}
{'A', 'B', 'C', 'D', 'E', 'F'}
```

这些推导的命令式版本实现和之前所展示的例子相似。

递归

习惯于函数式编程的程序员经常使用递归来表达循环。这样做，我们可以避免在算法内改变任何变量或数据结构 的状态，更重要的是能够更多的了解 “What”。然而在考虑使用递归风格时，我们应该区分这两种情形： 1)递归仅仅是迭代的另一个名称；2)递归将问题划分为处理方式类似的较小问题

有两个原因使我们应当在这里提及这个区别：

1)使用递归虽然可能是一种高效的方法来遍历序列中的每一个元素，但真的不 “Pythonic”。这是 其他语言的风格，比如 Lisp，在 Python 中显得矫揉造作。

2)Python 在递归中相对较慢，还有着栈深度限制。虽然你可以通过 `sys.setrecursionlimit()` 来使其超过默认值 1000 (译者注 2.x 为 1000; 3.x 为 2000)。但这样做，可能会是个错误的决定。 Python 没有尾调用消除，在某些语言中使用尾调用消除可以高效地进行深度递归计算。让我们看一个递归 是迭代的另一种形式的例子，可能会有些无聊:

```
def running_sum(numbers, start=0):
    if len(numbers) == 0:
        print()
        return
    total = numbers[0] + start
    print(total, end=" ")
    running_sum(numbers[1:], total)
```

然而，很少有人推荐这种方法；相比之下，简单重复地修改状态变量的迭代可读性会更高，此外当 `numbers` 长度大于 1000 时，这个函数会成为解释栈溢出的完美示例。但在其他情况下，递归风格，即使是顺序操作， 仍能更直观地表达算法，并且是以一种更容易理解的方式。以阶乘举例

递归版本:

```
def factorialR(N):
    "Recursive factorial function"
    assert isinstance(N, int) and N >= 1
    return 1 if N <= 1 else N * factorialR(N-1)
```

迭代版本:

```
def factorialI(N):
  "Iterative factorial function"
  assert isinstance(N, int) and N >= 1
  product = 1
  while N >= 1:
      product *= N
      N -= 1
  return product
```

虽然这个算法使用 `product` 变量就能够被容易地表达，但使用递归表示更倾向于算法的 “What” 而不是 “How”。迭代版本中反复地改变 `product` 和 `N` 的值，感觉就像是在记账，而不是 寻求计算的本质(但迭代版本大概更快，并且没有限制)。

多说一点，我所知道的 Python 中 `factorial()` 的最快版本是函数式编程风格的，并且很好地 表达了算法的 “What”，不过要熟悉一些高阶函数:

```
from functools import reduce
from operator import mul
def factorialHOF(n):
    return reduce(mul, range(1, n+1), 1)
```

某些场景下，我们不得不使用递归。有时递归甚至是表达解决方案的唯一明显方式，例如当一个问题提供了 将自己分治的途径。分治换句话说就是能够对一个大集合的两半(或者，几个类似大小的块)进行相似的计算。 在这种情况下，递归深度不会太大，仅为 O(log N)， N 为序列的长度。举个例子，快速排序算法十分优雅，不需要任何状态变量或循环，完全通过递归来表达:

```
def quicksort(lst):
    "Quicksort over a list-like sequence"
    if len(lst) == 0:
        return lst
    pivot = lst[0]
    pivots = [x for x in lst if x == pivot]
    small = quicksort([x for x in lst if x < pivot])
    large = quicksort([x for x in lst if x > pivot])
    return small + pivots + large
```

函数体中使用的一些变量保存了中间结果，他们是不可变的(每次递归会创建新的栈空间，虽然变量名相同， 但并未改变上一个栈空间变量的指向)。如果我们想做，可以将定义写成单个的表达式，虽然影响可读性。 事实上，将它变成一个使用状态变量的迭代版本是有些困难的，并且比较晦涩。

作为一般建议，当一个问题看起来可以分割成为较小的问题时，尝试寻找递归表达的可能性是一个很好的做法， 特别是一个能避免状态变量和可变数据的版本。但大多数时候，在 Python 中，仅仅使用递归去完成迭代的功能 并不是一个好主意。

消除循环

Just for fun，让我们看看如何从 Python 程序中删除所有循环。大多数时候，无论是对于可读性还是 性能来说，这是个坏主意，but it is worth looking at how simple it is to do in a systematic fashion as background to contemplate those cases where it is actually a good idea.(译者不知所云：但值得一看的是，在系统化背景下消除循环是一个 不错的主意去容易的思考一些场景)

如果我们在 `for` 循环内部调用一个函数，那么建高阶函数 `map()` 可以给予我们帮助:

```
for e in it:    # statement-based loop
    func(e)
```

下面的代码完全等价，除了没有重复重地新绑定变量 `e`，因此没有状态:

```
map(func, it)    # map()-based "loop"
```

类似的技术可用于使用函数式的方法来实现顺序程序流。大多数命令式程序都是由诸如"先做这，再做那个， 然后再做其他的"这样的语句所组成。如果这些操作全放到了函数中，那么 `map()` 就可以让我们这样做:

```
# let f1, f2, f3 (etc) be functions that perform actions
# an execution utility function
do_it = lambda f, *args: f(*args)
# map()-based action sequence
map(do_it, [f1, f2, f3])
```

我们可以将函数序列和传递对应参数的迭代器进行组合:

```
>>> hello = lambda first, last: print("Hello", first, last)
>>> bye = lambda first, last: print("Bye", first, last)
>>> _ = list(map(do_it, [hello, bye],
>>>                     ['David','Jane'], ['Mertz','Doe']))
```

```
Hello David Mertz
Bye Jane Doe
```

当然，看这个例子，有人猜想结果会是将所有的参数传入每一个函数而不是从每一个列表取一个参数传入函数。 若实现上面的想法，不使用列表推导表达起来是比较困难的:

```
>>> do_all_funcs = lambda fns, *args: [
                           list(map(fn, *args)) for fn in fns]
>>> _ = do_all_funcs([hello, bye],
                     ['David','Jane'], ['Mertz','Doe'])
Hello David Mertz
Hello Jane Doe
Bye David Mertz
Bye Jane Doe
```

一般来说，我们的整个主程序原则上可以是一个含有用来完成程序的可迭代函数集的 `map()` 表达式。 转换虽然稍微复杂一些，但可以直接使用递归:

```
# statement-based while loop
while <cond>:
    <pre-suite>
    if <break_condition>:
        break
    else:
        <suite>

# FP-style recursive while loop
def while_block():
    <pre-suite>
    if <break_condition>:
        return 1
    else:
        <suite>
    return 0

while_FP = lambda: (<cond> and while_block()) or while_FP()
while_FP()
```

我们对于 `while` 的转换仍然需要一个 `while_block()` 函数，它可能不仅仅包含表达式，或许 还会有语句。我们可以使用上面提到的 `map()` 来进一步将组件转换成函数序列。如果我们这样做， 还可以返回一个的三元表达式。将这个进一步的函数式重构作为一个练习留给读者。它可能会比较丑陋， 并且不适合生产环境，但是很好玩。

使用通常的测试手段，<cond> 很难有利用价值，比如 `while myvar==7`，因为循环体(按设计) 不能改变任何变量的值。一种添加更有用的条件的方法是让 `while_block()` 返回一个更有趣的值， 并比较返回值是否为终止条件。以下是一个删除语句的具体示例:

```
# imperative version of "echo()"
def echo_IMP():
    while 1:
        x = input("IMP -- ")
        if x == 'quit':
            break
        else:
            print(x)
echo_IMP()
```

现在让我们来删除 `while` 循环:

```
# FP version of "echo()"
def identity_print(x):    # "identity with side-effect"
    print(x)
    return x

echo_FP = lambda: identity_print(input("FP -- "))=='quit' or echo_FP()
echo_FP()
```

我们已经完成的是，设法将一个涉及 I/0，循环和条件语句的小程序表达为纯粹的递归表达式(实际上，作为 一个可以传递到别处的函数对象)。我们仍然使用工具函数 `identity_print()`，这个函数是完全通用的， 能够在稍后创建的每个函数式程序表达式中重用。请注意，包含 `identity_print(x)` 的任何表达式的 计算结果和简单地包含 `x` 相同; 它只是包含了额外的 I/O 操作。

消除递归

和上面所给出的阶乘示例相似，有时我们可以通过 `functools.reduce()` 或其他的 fold 操作(其他 folds 不在 Python 标准库中，但可以通过第三方库构建)来实现隐式的递归。递归通常只是将一些更简单的 结果与累积的中间结果相结合的方法，这正是 `reduce()` 内部所做的。关于 `functools.reduce()` 的更多讨论在高阶函数章节。

CHAPTER 2

Callables

callables 指可进行调用的一切东西，翻译出来感觉不如不翻译

函数式编程的重点在于任何情况下只要参数相同，调用函数都会得到相同的结果(tautologously)。 Python 实际上提供了几种不同的方式去创建函数，或一些和函数相似的东西(比如可以被调用)。这些方法是

- 使用 `def` 关键字创建的普通函数
- 使用 `lambda` 关键字创建的匿名函数
- 定义了 `__call()__` 方法的类的实例
- 函数工厂返回的闭包
- 经过 `@staticmethod` 装饰或类 `__dict__` 内的静态方法
- 生成器函数

这个列表可能不是很详尽，但让人感觉有众多不同的方法来创建一些可以被调用的东西。当然，一个类实例的 一般方法也是可以被调用的，但通常使用他们时，重点放在于访问和修改那些可变状态。

Python 是一个支持多种编程范式的语言，但它的重点放在面向对象编程上。定义一个类，通常是为了生成对象， 对象就像一个数据容器，可以通过调用方法来改变其中的数据。这种风格在某些方面上与强调纯函数和不可变的 函数式编程相反。

任何访问实例状态来决定最终返回结果的方法都不是纯函数。我们所讨论的其他可调用类型也或多或少以各种方式 依赖了状态。这篇报告的作者长期以来一直在思考，是否可以在 Python 中使用一些黑魔法来明确地声明一个 纯函数。通过用一个假想的装饰器 `@purefunction` 来装饰，如果该函数会产生副作用则抛出异常， 但共识是似乎无法防范 Python 内部的每一个边界情况。

纯函数和无副作用代码的优点是很容易调试和测试。大部分返回结果依赖于状态的 `callables` 不能进行独立上下文的检测来摸清他们的行为。举个例子，单元测试(使用 `doctest` 或 `unittest`， 或一些诸如 `py.test` `nose` 的第三方的测试框架)可能会在某个上下文中成功，但在一些正在执行 的有状态程序中进行相同的调用会失败。当然，任何程序都至少会有某种输出(命令行、文件、数据库、网络或者 其他的)，所以副作用并不能被完全的消除。只有在考虑函数式编程术语时，我们可以暂且放到一边。

具名函数和匿名函数

显而易见的，在 Python 中创建 `callables` 的最明显的方法是具名函数和匿名函数。 原则上，两者之间的唯一区别是他们是否有 `a.__qualname__` 属性，即使都能够被绑定 到多个变量名上。在大多数场景中，lambda 表达式在 Python 中只用来作为回调函数或作为 简单的操作被内联至函数调用中。但就像我们先前所展示的那样，如果我们想要，控制流通常可以 被合并到单个匿名表达式中。举例说明如下:

```
>>> def hello1(name):
.....    print("Hello", name)
.....
>>> hello2 = lambda name: print("Hello", name)
>>> hello1('David')
Hello David
>>> hello2('David')
Hello David
>>> hello1.__qualname__
'hello1'
>>> hello2.__qualname__
'<lambda>'
>>> hello3 = hello2
# can bind func to other names
>>> hello3.__qualname__
'<lambda>'
>>> hello3.__qualname__ = 'hello3'
>>> hello3.__qualname__
'hello3'
```

函数是有用的的原因之一是他们从词法上隔绝了状态，避免了命令空间污染。这是一种有限制的不可变形式， 虽然你在函数中什么都没做，依然绑定了函数外部的状态变量(只读)。当然，这种保证也是非常有限的，因为 *global* 和 *nonlocal* 语句都允许将状态变量“泄漏”给一个函数。此外，许多数据类型本身是可变的，因此如果它们被传递到函数中可能会改变其值。此外，I/O 也可以改变状态，从而改变函数的结果(通过更改它 所读取的文件或数据库中的内容)

尽管存在上述所有的注意事项和限制，但想要专注于函数式编程风格的程序员可以有意地将许多函数写为纯函数 来进行数学和形式化的理解。In most cases, one only leaks state intentionally, and creating a certain subset of all your functionality as pure functions allows for cleaner code. They might perhaps be broken up by “pure” modules, or annotated in the function names or docstrings.

闭包和可调用实例

计算机科学中有一句名言:”class is data with operations attached while a closure is operations with data attached”(类是附带操作的数据，闭包是附带数据的操作)。在某种意义上说， 他们完成了相同的事情，将逻辑和数据放在同一个对象中。但毫无疑问有着不同，类强调可变和能重新绑定的状态， 闭包强调不可变和纯函数。至少在 Python 中，这个分界的两侧没有一个是绝对的，使用哪种要看自己的态度。

让我们构建一个示例，来展现这两种风格:

```
# A class that creates callable adder instances
class Adder(object):
    def __init__(self, n):
        self.n = n
    def __call__(self, m):
        return self.n + m
add5_i = Adder(5)    # "instance" or "imperative"
```

我们构造了一个能够被调用的实例，并且会将传入的参数与 5 相加。似乎已经够简单了，再来试试闭包版本:

```
def make_adder(n):
    def adder(m):
        return m + n
    return adder
add5_f = make_adder(5)    # "functional"
```

到目前为止，它们看起来相同，但实例中的可变状态提供了一个 attractive nuisance:

```
>>>add5_i(10)
15
>>>add5_f(10)         # only argument affects result
15
>>>add5_i.n = 10      # state is readily changeable
>>>add5_i(10)         # result is dependent on prior flow
20
```

Adder()``类和 ``make_adder() 所创建的 “adder” 的行为通常直到运行时才会确定。但一旦对象(add5_i、add5_f)被创建，闭包就能显现出纯函数的特点，然而类实例仍然依赖着状态。人们可能 会通过互相约定”不要改变状态”来解决这个问题，确实这是可能的(如果没有其他不了解的人导入并使用了你的 代码)，但是避免滥用编程风格是更好的习惯。

Python 闭包中的变量绑定有一点问题。它通过 name 而不是 value 来，这可能会引起混淆，但也有一个 简单的解决方案。例如，制造几个相关的闭包封装不同的数据:

```
# almost surely not the behavior we intended!
>>> adders = []
>>> for n in range(5):
        adders.append(lambda m: m+n)
>>> [adder(10) for adder in adders]
[14, 14, 14, 14, 14]
>>> n = 10
>>> [adder(10) for adder in adders]
[20, 20, 20, 20, 20]
```

幸运的是，一个小小的改变便能使行为符合预期:

```
>>> adders = []
>>> for n in range(5):
....     adders.append(lambda m, n=n: m+n)
....
>>> [adder(10) for adder in adders]
[10, 11, 12, 13, 14]
>>> n = 10
>>> [adder(10) for adder in adders]
[10, 11, 12, 13, 14]
>>> add4 = adders[4]
>>> add4(10, 100)    # Can override the bound value
110
```

Notice that using the keyword argument scope-binding trick allows you to change the closed-over value; but this poses much less of a danger for confusion than in the class instance. The overriding value for the named variable must be passed explictly in the call itself, not rebound somewhere remote in the program flow. Yes, the name add4 is no longer accurately descriptive for “add any two numbers,” but at least the change in result is syntactically local.

类方法

所有的类方法都是可调用的。然而，在大多数情况下，调用实例方法违背了函数式编程的风格。通常我们使用方法， 是因为我们要引用绑定在实例属性中的可变数据，因此对方法的每次调用可能产生参数相同结果不同的情况。

访问器和操作符

访问器(使用 `@property` 或其他方法所创建)也是可调用的。尽管访问器在使用上是有限制的(从函数式 编程的角度)，因为它们的 getter 没有参数，setter 没有返回值:

```
class Car(object):

    def __init__(self):
        self._speed = 100

    @property
    def speed(self):
        print("Speed is", self._speed)
        return self._speed

    @speed.setter
    def speed(self, value):
        print("Setting to", value)
        self._speed = value

# >> car = Car()
# >>> car.speed = 80   # Odd syntax to pass one argument
# Setting to 80
# >>> x = car.speed
# Speed is 80
```

我们通过 Python 赋值语法来向访问器传入参数。这种操作对于 Python 来说相当容易:

```
>>> class TalkativeInt(int):
        def __lshift__(self, other):
            print("Shift", self, "by", other)
            return int.__lshift__(self, other)
....
>>> t = TalkativeInt(8)
>>> t << 3
Shift 8 by 3
64
```

Python 中的每一个操作符对应着一个底层方法。虽然偶尔可能会产生可读性更高的 DSL，但为运算符定义特殊 的方法并没有改变调用底层函数的本质。

实例的静态方法

与函数式风格更为相近的一种类和方法的使用方法是是简单地将它们用作命名空间来存放各种相关的功能:

```
import math
class RightTriangle(object):
    "Class used solely as namespace for related functions"

    @staticmethod
    def hypotenuse(a, b):
        return math.sqrt(a**2 + b**2)
```

```
    @staticmethod
    def sin(a, b):
        return a / RightTriangle.hypotenuse(a, b)

    @staticmethod
    def cos(a, b):
        return b / RightTriangle.hypotenuse(a, b)
```

这样做避免了污染全局(或模块)命名空间，我们可以使用类名或相应的实例名来调用纯函数:

```
>>> RightTriangle.hypotenuse(3,4)
5.0
>>> rt = RightTriangle()
>>> rt.sin(3,4)
0.6
>>> rt.cos(3,4)
0.8
```

目前为止，定义静态方法的最直接明显的方式是使用装饰器。但是，在 Python 3.x 中你也可以不使用装饰器，“unbound method” 的概念已经退出舞台了。

```
>>> import functools, operator
>>> class Math(object):
...     def product(*nums):
...         return functools.reduce(operator.mul, nums)
...     def power_chain(*nums):
...         return functools.reduce(operator.pow, nums)
...
>>> Math.product(3,4,5)
60
>>> Math.power_chain(3,4,5)
3486784401
```

不过，若不使用 `@staticmethod`，则无法从实例调用静态方法，因为他们仍然会隐式传入 `self`:

```
>>> m = Math()
>>> m.product(3,4,5)
---------------------------------------------------------------------
TypeError
Traceback (most recent call last)
<ipython-input-5-e1de62cf88af> in <module>()
----> 1 m.product(3,4,5)

<ipython-input-2-535194f57a64> in product(*nums)
      2 class Math(object):
      3     def product(*nums):
----> 4         return functools.reduce(operator.mul, nums)
      5     def power_chain(*nums):
      6         return functools.reduce(operator.pow, nums)

TypeError: unsupported operand type(s) for *: 'Math' and 'int'
```

如果你的命名空间只是用来存放纯函数，那么没有理由不通过类来调用。但如果你想将纯函数和一些依赖于实例 可变状态的函数进行混合，那么你应当使用 `@staticmethod` 这个装饰器。

生成器函数

含有 `yield` 语句用来作为生成器的函数是 Python 中一类特殊的函数。调用它返回的不是一个普通的 值，而是一个能够被 `next()` 函数多次调用产生一序列值的迭代器。这将在惰性求值章节进行具体讨论。

虽然像 Python 的其他对象一样，存在着许多方法可以将状态引入到生成器中，但原则上生成器可以是纯的，就像纯函数那样。它产生一序列值(可能无限)，而不是单一的一个值，但是仍然基于传入的参数。不过请注意，生成器函数也有着内部状态；它是在调用签名(call signature)和返回值的边界，它们的行为就像一个无 副作用的“黑盒子”。一个简单的例子:

```
>>> def get_primes():
...     "Simple lazy Sieve of Eratosthenes"
...     candidate = 2
...     found = []
...     while True:
...         if all(candidate % prime != 0 for prime in found):
...             yield candidate
...             found.append(candidate)
...         candidate += 1
...
>>> primes = get_primes()
>>> next(primes), next(primes), next(primes)
(2, 3, 5)
>>> for _, prime in zip(range(10), primes):
...     print(prime, end=" ")
....
7 11 13 17 19 23 29 31 37 41
```

每次你通过 `get_primes()` 创建新对象，迭代器都是相同的无限惰性序列——另一个例子可能会传入 一些影响结果的值用于初始化——但对象本身是有状态的，正如它不断的被消费(consume)。

多分派(Multiple Dispatch)

编写多路执行代码的一个非常有趣的方法是通过名为 “multiple dispatch”(“multimethods”) 的技术。 这种技术为单个函数声明多个签名，并调用与参数的类型或属性相匹配的函数。这种技术通常允许我们去减少或 避免显示分支条件的使用，使用更直观的参数模式描述来作为替代。

很久之前，作者写了一个名叫 `multimethods` 的模块，它能够通过选项来灵活的进行线性分发，但它 只能运行在 Python 2.x 上，并且是在 Python 提出装饰器这个优雅概念之前写的。Matthew Rocklin's more recent `multipledispatch` is a modern approach for recent Python versions, albeit it lacks some of the theoretical arcana I explored in my ancient module. 理想情况下，在作者看来，未来的 Python 应当包含能进行多分派的标准语法或 API(但有可能这个任务会由 第三方库来完成)

为了解释多派发如何能使代码变得可读性更好并且较少发生错误，让我们用三种方式来实现剪刀/石头/布游戏。 下面的类是三个版本共有的:

```
class Thing(object): pass
class Rock(Thing): pass
class Paper(Thing): pass
class Scissors(Thing): pass
```

多分支

首先是纯命令版本，这会有许多重复的，嵌套的，条件块，会令人出错:

```
def beats(x, y):
    if isinstance(x, Rock):
        if isinstance(y, Rock):
            return None          # No winner
        elif isinstance(y, Paper):
            return y
```

```
        elif isinstance(y, Scissors):
            return x
        else:
            raise TypeError("Unknown second thing")
    elif isinstance(x, Paper):
        if isinstance(y, Rock):
            return x
        elif isinstance(y, Paper):
            return None           # No winner
        elif isinstance(y, Scissors):
            return y
        else:
            raise TypeError("Unknown second thing")
    elif isinstance(x, Scissors):
        if isinstance(y, Rock):
            return y
        elif isinstance(y, Paper):
            return x
        elif isinstance(y, Scissors):
            return None           # No winner
        else:
            raise TypeError("Unknown second thing")
    else:
        raise TypeError("Unknown first thing")

rock, paper, scissors = Rock(), Paper(), Scissors()
# >>> beats(paper, rock)
# <__main__.Paper at 0x103b96b00>
# >>> beats(paper, 3)
# TypeError: Unknown second thing
```

委托给对象

作为第二个尝试，我们会使用 Python 的 “duck typing” 来消除一些不必要的重复， “duck typing” 指我们比起类型更关注其所具有的方法，含有相同的方法就可以被调用，尽管它们的类型可能不同:

```
class DuckRock(Rock):
    def beats(self, other):
        if isinstance(other, Rock):
            return None          # No winner
        elif isinstance(other, Paper):
            return other
        elif isinstance(other, Scissors):
            return self
        else:
            raise TypeError("Unknown second thing")

class DuckPaper(Paper):
    def beats(self, other):
        if isinstance(other, Rock):
            return self
        elif isinstance(other, Paper):
            return None        # No winner
        elif isinstance(other, Scissors):
            return other
        else:
            raise TypeError("Unknown second thing")
```

```
class DuckScissors(Scissors):
    def beats(self, other):
        if isinstance(other, Rock):
            return other
        elif isinstance(other, Paper):
            return self
        elif isinstance(other, Scissors):
            return None     # No winner
        else:
            raise TypeError("Unknown second thing")

def beats2(x, y):
    if hasattr(x, 'beats'):
        return x.beats(y)
    else:
        raise TypeError("Unknown first thing")

rock, paper, scissors = DuckRock(), DuckPaper(), DuckScissors()
# >>> beats2(rock, paper)
# <__main__.DuckPaper at 0x103b894a8>
# >>> beats2(3, rock)
# TypeError: Unknown first thing
```

我们实际上并没有减少代码量，但是这个版本降低了每个 callable 的复杂性，并减少了条件语句的嵌套层级。 大多数的逻辑被放入独立的类中，而不是更深层次的分支。在面向对象编程中，我们可以将分派委托到对象 (but only to the one controlling object)。

模式匹配

作为最后的尝试，我们可以使用多分派(multiple dispatch)来更直接地表达所有的逻辑。这会更加易读， 尽管仍然有一定量的分支:

```
from multipledispatch import dispatch

@dispatch(Rock, Rock)
def beats3(x, y): return None

@dispatch(Rock, Paper)
def beats3(x, y): return y

@dispatch(Rock, Scissors)
def beats3(x, y): return x

@dispatch(Paper, Rock)
def beats3(x, y): return x

@dispatch(Paper, Paper)
def beats3(x, y): return None

@dispatch(Paper, Scissors)
def beats3(x, y): return x

@dispatch(Scissors, Rock)
def beats3(x, y): return y

@dispatch(Scissors, Paper)
def beats3(x, y): return x
```

```
@dispatch(Scissors, Scissors)
def beats3(x, y): return None

@dispatch(object, object)
def beats3(x, y):
    if not isinstance(x, (Rock, Paper, Scissors)):
        raise TypeError("Unknown first thing")
    else:
        raise TypeError("Unknown second thing")

# >>> beats3(rock, paper)
# <__main__.DuckPaper at 0x103b894a8>
# >>> beats3(rock, 3)
# TypeError: Unknown second thing
```

基于谓词的分派(Predicate-Based Dispatch)

使用分派来表达条件语句的一种奇葩方式是直接在函数签名中包含谓词(或者对函数使用装饰器，像 `multipledispatch`)。我目前不知道有这样的 Python 库，但让我们定一个假想的库来简单的说明这个 概念。这个假想库叫做 `predicative_dispatch`:

```
from predicative_dispatch import predicate

@predicate(lambda x: x < 0, lambda y: True)
def sign(x, y):
    print("x is negative; y is", y)

@predicate(lambda x: x == 0, lambda y: True)
def sign(x, y):
    print("x is zero; y is", y)

@predicate(lambda x: x > 0, lambda y: True)
def sign(x, y):
    print("x is positive; y is", y)
```

虽然这个小例子显然不是一个完整的规范，但读者能看到我们如何将大部分甚至全部条件语句移至函数调用签名中， 这会导致更小，更容易理解和调试的函数。

译者注，使用策略模式也可以减少 *if-else* 的嵌套层级

CHAPTER 3

惰性求值

Python 的一个强大功能是它的迭代器协议(我们很快便会了解到)。这个功能和函数式编程有这松散的联系，因为 Python 并不像 Haskell 这类语言提供了惰性数据结构(lazy data structures)。然而， 使用迭代器协议和 Python 的一些内置或标准库中的可迭代类型可以实现与惰性数据结构相同的效果。

让我们稍微详细地解释这方面的对比。像 Haskell 这样本质是惰性求值的语言，我们可以像下面这样来定义一个包含所有素数的列表:

```
-- Define a list of ALL the prime numbers
primes = sieve [2 ..]
  where sieve (p:xs) = p : sieve [x | x <- xs, (x `rem` p)/=0]
```

这篇文章不是来教 Haskell 的，但是你可以发现一个推导式，实际上这正是 Python 引入的推导式的原型。这里还涉及到深度递归，这在 Python 中不会奏效。

除了语法差异，甚至是进行非确定深度递归的能力外，有个显著的差异便是 Haskell 版本的素数实际上是一个 (无限的)序列，而不仅仅是能够顺序生成元素的对象(Callables 章节提到过)。特别地，您可以根据下标获取 Haskell 无限列表中的任意元素(译者注 `primes !! n`)，根据列表自身语法结构的需要，中间值会在 内部生成。

提醒一下，你可以将这套照搬进 Python 中，只是不是语言自身的固有语法，需要更多的手动构造。使用先前 讨论过的 `get_primes()` 生成器函数，我们可以编写自己的容器来模拟相同的事情，例如:

```
from collections.abc import Sequence
class ExpandingSequence(Sequence):
    def __init__(self, it):
        self.it = it
        self._cache = []
    def __getitem__(self, index):
        while len(self._cache) <= index:
            self._cache.append(next(self.it))
        return self._cache[index]
    def __len__(self):
        return len(self._cache)
```

新的容器同时具有惰性和可索引的特性:

```
>>> primes = ExpandingSequence(get_primes())
>>> for _, p in zip(range(10), primes):
....     print(p, end=" ")
....
2 3 5 7 11 13 17 19 23 29
>>> primes[10]
31
>>> primes[5]
13
>>> len(primes)
11
>>> primes[100]
547
>>> len(primes)
101
```

当然，我们可能还会向其中添加一些其他功能，因为惰性数据结构不是 Python 自有的。也许我们想对这个特殊 的序列进行切片操作。也许我们希望在打印时更好地表示对象。也许我们应该把长度报告为 `inf`，如果我们 能以某种方式证明它是无限的。所有这一切都是可能的，但它需要为每一种行为添加一些代码，而不是使用 Python 的默认行为。

迭代器协议

在 Python 中创建一个迭代器(或者说一个惰性序列)的最简单的方法是定义一个生成器函数，如"Callables" 一章所讨论的那样。只需在函数体内需要生成值的位置(通常在循环中)使用 `yield` 语句即可。

或者最简单的方法是使用 builtin 或标准库提供许多可迭代对象，而不是自己从头编写一个对象。生成器函数 是定义一个返回迭代器的函数的语法糖。

许多对象含有用来返回迭代器的 `.__iter__()` 方法，这个方法通常被内建函数 `iter()` 或者循环这个 对象(e.g., for item in collection: ...)时调用。

迭代器是通过调用 `iter(something)` 所产生的对象，它自身也具有 `.__iter__()` 方法，这个 方法会返回它本身。迭代器具有的另一个方法是 `.__next__()`。迭代器自身仍具有 `.__iter__()` 方法的原因是为了让 `iter()` 幂等。That is, this identity should always hold (or raise TypeError("object is not iterable")):

```
iter_seq = iter(sequence)
iter(iter_seq) == iter_seq
```

上面讲的有点抽象，所以我们来看几个具体的例子:

```
>>> lazy = open('06-laziness.md') # iterate over lines of file
>>> '__iter__' in dir(lazy) and '__next__' in dir(lazy)
True
>>> plus1 = map(lambda x: x+1, range(10))
>>> plus1                   # iterate over deferred computations
<map at 0x103b002b0>
>>> '__iter__' in dir(plus1) and '__next__' in dir(plus1)
True
>>> def to10():
...     for i in range(10):
...         yield i
...
>>> '__iter__' in dir(to10)
False
>>> '__iter__' in dir(to10()) and '__next__' in dir(to10())
```

```
True
>>> l = [1,2,3]
>>> '__iter__' in dir(l)
True
>>> '__next__' in dir(l)
False
>>> li = iter(l)          # iterate over concrete collection
>>> li
<list_iterator at 0x103b11278>
>>> li == iter(li)
True
```

在函数式编程中，或者为了追求可读性，使用生成器函数来编写自定义迭代器是再自然不过的事了。不过，我们还 可以创建遵循协议的类；通常它们也具有其他的行为(i.e., methods)，但大多数此类行为依赖于有状态和副作用。 例如:

```
from collections.abc import Iterable
class Fibonacci(Iterable):
    def __init__(self):
        self.a, self.b = 0, 1
        self.total = 0
    def __iter__(self):
        return self
    def __next__(self):
        self.a, self.b = self.b, self.a + self.b
        self.total += self.a
        return self.a
    def running_sum(self):
        return self.total

# >>> fib = Fibonacci()
# >>> fib.running_sum()
# 0
# >>> for _, i in zip(range(10), fib):
# ...     print(i, end=" ")
# ...
# 1 1 2 3 5 8 13 21 34 55
# >>> fib.running_sum()
# 143
# >>> next(fib)
# 89
```

这个例子不值一提，但它展示了一个实现迭代器协议，并且还提供了额外的方法来返回实例状态的类。由于使用状态 是面向对象程序员经常做的，在函数式编程风格中，我们通常会避免这样的类。

itertools 模块

itertools 模块是一个非常强大并且精心设计的用于 iterator algebra 的函数的集合。换句话说， 这些允许你使用高级的方式去组合迭代器，而无需去具体实例化任何比目前所需要的更多的东西。除了模块本身 的基本函数外，模块文档 provides a number of short, but easy to get subtly wrong, recipes for additional functions that each utilize two or three of the basic functions in combination. 前言中提到的第三方模块 more_itertools 提供了额外的函数，同样旨在避免常见的陷阱和边缘情况。

使用 itertools 的基本目的是避免在需要之前进行计算，避免大型集合实例化所需要的内存，避免在 真正需求之前可能会有较慢的I / O，等等。迭代器是惰性序列而不是已经实现的集合，并且当使用

itertools 和其他的函数相组合时，它们依旧保持特性。

这里有组合的一个简单的例子。比起有状态的 Fibonacci 类来维持累加和，让我们创建一个惰性迭代器来生成当前的值和总和:

```
>>> def fibonacci():
...     a, b = 1, 1
...     while True:
...         yield a
...         a, b = b, a+b
...
>>> from itertools import tee, accumulate
>>> s, t = tee(fibonacci())
>>> pairs = zip(t, accumulate(s))
>>> for _, (fib, total) in zip(range(7), pairs):
...     print(fib, total)
...
1 1
1 2
2 4
3 7
5 12
8 20
13 33
```

搞清楚如何正确，最佳地使用 itertools 模块中的函数通常需要经过缜密的思考，但一旦成功，可以获得显著的能力来处理那些无法用具体集合表现的大型甚至无穷的迭代器。

itertools 模块的文档包含了其函数的详细信息以及用于组合它们的许多简短样例。这篇文章没有地方去再次重复这些描述，所以仅展示其中的几个。注意实际上，zip()， map()， filter() 和 range() (某种意义上只是一个终止的 itertools.count())应当在 itertools 模块， 如果他们不是 builtins。所有的这些函数都会惰性生成序列项(大部分基于现有的迭代)，而不创建具体的序列。 all(), any(), sum(), min(), max() 和 functools.reduce() 也能用在 iterables 上，但他们需要将迭代器完全迭代，无法再保留其惰性了。函数 itertools.product() 可能在不应当放在这个模块中，因为它还创建具体的缓存序列，并且无法在无穷迭代器上操作。

Chaining Iterables

itertools.chain() 和 itertools.chain.from_iterable() 函数能够将多个迭代器进行 组合。Built-in zip() and itertools.zip_longest() also do this, of course, but in manners that allow incremental advancement through the iterables. 这样做的 结果是，连接无穷 iterables 在语法和语义上是有效的，没有实际的程序能够迭代完早期的可迭代。例如:

```
from itertools import chain, count
thrice_to_inf = chain(count(), count(), count())
```

概念上 thrice_to_inf 会三次计算至无穷，但实际上一次就足够了(in practice once would always be enough)。然而，对于很长的 iterables (非无穷)，使用 chain 是非常有用和简约的:

```
def from_logs(fnames):
    yield from (open(file) for file in fnames)
lines = chain.from_iterable(from_logs(
              ['huge.log', 'gigantic.log']))
```

请注意，在给出的示例中，我们甚至不需要传递具体的文件列表——这个列表可以是指定 API 的惰性迭代器。 除了使用 itertools 进行连接之外，我们还应该一同提及 collections.ChainMap()。字典 类型(通常任何的 collections.abc.Mapping 类型)是可迭代的。就像我们想要去连接多个类似序列 的 iterables 一样，我们有时希望连接多个映射(mappings)，而不需要直接创建一个更大的具体映射。ChainMap() 方便，并且没有改变构造它的底层映射。

CHAPTER 4

高阶函数

在上一章中，我们看到了建立在 itertools 模块的迭代器代数。在某些方面，高阶函数(通常缩写为“HOFs”)通过组合简单函数至新的函数来提供了类似的表达复杂概念的方式。一般来说，高阶函数是一个接受 一个或多个函数作为参数或者产生新函数作为结果的函数。这里提供了许多有趣的抽象。它们允许我们像使用 itertools 那样类似的方式进行连接和组合高阶函数。

functools 模块中包含一些有用的高阶函数，另一些则已经内建。通常认为，map(), filter(), 和 functools.reduce() 是高阶函数的最基本构建块，大多数函数式编程语言将他们作为原语(偶尔 使用其他的名字)。像 map/filter/reduce 作为构建块般基础的概念是柯里化。在 Python 中，柯里化 被叫做 partial()，它在 functools 模块中。这个函数，它接收一个函数以及零个或多个参数 然后进行预填充，返回一个较少参数的函数。调用时，就像参数已经传递过了一样。

内建函数 map() 和 filter() 等同于推导(特别是现在可以使用生成器推导)，而且大多数 Python 程序员都认为使用推导的版本更易于阅读。例如下面这些等效的组合:

```
# Classic "FP-style"
transformed = map(tranformation, iterator)
# Comprehension
transformed = (transformation(x) for x in iterator)

# Classic "FP-style"
filtered = filter(predicate, iterator)
# Comprehension
filtered = (x for x in iterator if predicate(x))
```

functools.reduce() 函数非常通用，非常强大，并且能非常巧妙地发挥其全部功能。它从一个 iterable 中依次去除相连元素，并以某种方式进行组合。reduce() 的最常见的用例涵盖了内建的 sum()， 下面一种更简洁的写法:

```
from functools import reduce
total = reduce(operator.add, it, 0)
# total = sum(it)
```

map() 和 filter() 也可以被认为是 reduce() 的特殊情况:

```
>>> add5 = lambda n: n+5
>>> reduce(lambda l, x: l+[add5(x)], range(10), [])
[5, 6, 7, 8, 9, 10, 11, 12, 13, 14]
>>> # simpler: map(add5, range(10))
>>> isOdd = lambda n: n%2
>>> reduce(lambda l, x: l+[x] if isOdd(x) else l, range(10),
[])
[1,3, 5, 7, 9]
>>> # simpler: filter(isOdd, range(10))
```

这些 `reduce()` 表达式是尴尬的，但它们也说明了该函数在其通用性方面的强大程度：任何需要从序列 取出连续元素来计算的操作都可以被表示为 reduction。

有几个常见的高阶函数没有被 Python 所包含，但很容易作为工具函数来创建(并且包含在许多第三方函数式编程工具集合中)。不同的库或其他编程语言可能使用与上述工具函数所不同的名称，但类似的功能是广泛存在 (我选择的名称也是如此)。

实用高阶函数

utility function 译者之前翻译成了工具函数，但本章节标题 *utility higher-order functions* 觉得这种翻译不太适用，所以又翻成了实用

一个方便的工具是 `compose()`。这是一个接受一序列函数并且返回一个函数，它接收一系列函数， 并返回一个函数。这个函数能将刚才传入的参数函数依次应用于数据参数:

```
def compose(*funcs):
    """Return a new function s.t.
    compose(f,g,...)(x) == f(g(...(x)))"""
    def inner(data, funcs=funcs):
        result = data
        for f in reversed(funcs):
            result = f(result)
        return result
    return inner

# >>> times2 = lambda x: x*2
# >>> minus3 = lambda x: x-3
# >>> mod6 = lambda x: x%6
# >>> f = compose(mod6, times2, minus3)
# >>> all(f(i)==((i-3)*2)%6 for i in range(1000000))
# True
```

对于这些集中于一行的数学运算(times2，minus3等)，我们可以简单地写下对应原始的数学表达式； 但如果是涉及分支，流控制，复杂逻辑的复合计算，这不是真的。

内建函数 `all()` 和 `any()` 在判断 iterable 的元素是否对于谓词成立上很有用；但也能通过 组合来判断元素是否对于谓词集合成立，我们可以向下面这样实现:

```
all_pred = lambda item, *tests: all(p(item) for p in tests)
any_pred = lambda item, *tests: any(p(item) for p in tests)
```

为了展示用法，我们来制造些谓词:

```
>>> is_lt100 = partial(operator.ge, 100)     # less than 100?
>>> is_gt10 = partial(operator.le, 10)       # greater than 10?
>>> from nums import is_prime                # implemented elsewhere
```

```
>>> all_pred(71, is_lt100, is_gt10, is_prime)
True
>>> predicates = (is_lt100, is_gt10, is_prime)
>>> all_pred(107, *predicates)
False
```

toolz 库有一个更通用的版本叫做 juxt()，它创建一个使用相同参数调用多个函数并返回一个 结果元组的函数。如下所示:

```
>>> from toolz.functoolz import juxt
>>> juxt([is_lt100, is_gt10, is_prime])(71)
(True, True, True)
>>> all(juxt([is_lt100, is_gt10, is_prime])(71))
True
>>> juxt([is_lt100, is_gt10, is_prime])(107)
(False, True, True)
```

上面用的实用高阶函数只是来说明可组合性的例子。看一下有关函数式编程的更详细的文本，例如，阅读 Haskell prelude 来获得有关实用高阶函数的许多概念。

operator 模块

像先前几个例子所展示的那样，每个使用 Python 中缀和前缀运算符来完成的操作都在 operator 模块 存在对应的函数。对于希望向高阶函数传入相当于某些语法操作的函数作为参数的地方，使用 operator 模块的函数更快，并且看起来比相应的匿名函数更优雅，例如:

```
# Compare ad hoc lambda with `operator` function
sum1 = reduce(lambda a, b: a+b, iterable, 0)
sum2 = reduce(operator.add, iterable, 0)
sum3 = sum(iterable)    # The actual Pythonic way
```

functools 模块

Python 一个包含高阶函数的很明显的地方便是 functools 模块，实际上的确有一些在这里。不过， 令人惊讶的是，在该模块中几乎没有多少实用高阶函数。随着新版本发布，已经出现了一些有趣的函数，但核心 开发人员并不想将其变成一个完全的函数式编程语言。另一方面，正如我们在上面的几个示例中看到的，许多 有用的高阶函数只需要几行(有时甚至是一行)来就能编写。

除了本章开头讨论 reduce() 之外，模块中的主要工具便是曾提到过的 partial()。这种操作 在许多语言中被称为柯里化，之前也提过一些使用 partial() 的例子。functools 模块的其余 部分是有用的装饰器，这是下一节的主题。

装饰器

虽然设计上容易被忘记，但 Python 中最常见的高阶函数用法可能就是装饰器。装饰器只是一种将一个函数作为 参数的语法糖，如果编写正确，则会返回一种新函数，它以某种方式增强原始函数(或方法或类)。提醒读者， 这两个代码片段定义的 some_func 和 other_func 是等价的:

```
@enhanced
def some_func(*args):
    pass
def other_func(*args):
    pass
other_func = enhanced(other_func)
```

使用装饰器语法，当然，高阶函数必须用于函数定义时。为了达到预定的目的，这通常是最好的应用。但原则上 同样的装饰器函数可以在程序的其他地方使用，例如以更动态的方式(e.g., mapping a decorator function across a runtime-generated collection of other functions)。然而，这是一个 不常用的例子。

装饰器被使用在标准库和常见的第三方库中的许多场合里。在某些方面，它们与以前称为"面向切面的程序设计" 的想法相结合。例如，装饰器函数 `asyncio.coroutine` 用于将函数标记为协程。在 `functools` 中，三个重要的装饰器函数是 `functools.lru_cache`， `functools.total_ordering` 和 `functools.wraps`。`functools.lru_cache` 能"记住""一个函数(缓存传入的参数，返回存储 的值，而不是重新进行计算或者 I/O)。`functools.total_ordering` 能让自定义类实现比较运算 更加容易。`functools.wraps` 使得更容易编写新的装饰器。所有这些都是重要和有价值的目的，但是 他们是本着使 Python 编程更简单的精神，而不是着眼于本章重点关注的可组合的高阶函数。

装饰器通常在你想要 poke into the guts of a function 时比你仅将它视为函数流或者组合中的可拔插 组件时更有用，它经常用来标记特定函数的目的或功能。

这篇文章只是大致讲解了 Python 函数式编程的相关技术和 some suggestions as to the advantages one often finds in aspiring in that direction. 编写没有副作用的函数可以避免调试错误时遇到 的一大类困难，而且编写更容易理解和进行可靠测试的小型功能单元可以避免更多的错误。

A rich literature on functional programming as a general technique often in particular languages which are not Python—is available and well respected. Studying one of many such classic books, some published by O'Reilly (including very nice video training on functional programming in Python), can give readers further insight into the nitty-gritty of functional programming techniques. Almost everything one might do in a more purely functional language can be done with very little adjustment in Python as well.