# 第5章

# 一等函数

> 不管别人怎么说或怎么想，我从未觉得 Python 受到来自函数式语言的太多影响。我非常熟悉命令式语言，如 C 和 Algol 68，虽然我把函数定为一等对象，但是我并不把 Python 当作函数式编程语言。[1]
>
> ——Guido van Rossum
> Python 仁慈的独裁者

在 Python 中，函数是一等对象。编程语言理论家把“一等对象”定义为满足下述条件的程序实体：

- 在运行时创建
- 能赋值给变量或数据结构中的元素
- 能作为参数传给函数
- 能作为函数的返回结果

在 Python 中，整数、字符串和字典都是一等对象——没什么特别的。如果在 Python 之前，你使用的语言并未把函数当作一等公民，那么本章以及第三部分余下的内容将重点讨论把函数作为对象的影响和实际应用。

人们经常将“把函数视作一等对象”简称为“一等函数”。这样说并不完美，似乎表明这是函数中的特殊群体。在 Python 中，所有函数都是一等对象。

注 1：摘录自 Guido 的 The History of Python 博客，“Origins of Python's Functional Features”（http://python-history.blogspot.jp/2009/04/origins-of-pythons-functional-features.html）。

# 5.1 把函数视作对象

示例 5-1 中的控制台会话表明，Python 函数是对象。这里我们创建了一个函数，然后调用它，读取它的 __doc__ 属性，并且确定函数对象本身是 function 类的实例。

**示例 5-1** 创建并测试一个函数，然后读取它的 __doc__ 属性，再检查它的类型

```
>>> def factorial(n):  ❶
...     '''returns n!'''
...     return 1 if n < 2 else n * factorial(n-1)
...
>>> factorial(42)
1405006117752879898543142606244511569936384000000000
>>> factorial.__doc__  ❷
'returns n!'
>>> type(factorial)  ❸
<class 'function'>
```

❶ 这是一个控制台会话，因此我们是在“运行时”创建一个函数。
❷ __doc__ 是函数对象众多属性中的一个。
❸ factorial 是 function 类的实例。

__doc__ 属性用于生成对象的帮助文本。在 Python 交互式控制台中，help(factorial) 命令输出的内容如图 5-1 所示。

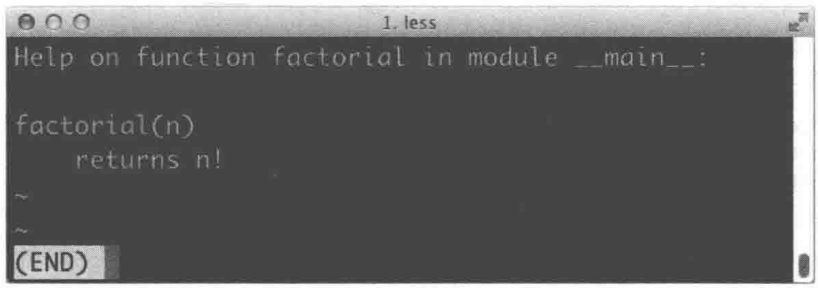


图 5-1：factorial 函数的帮助界面；输出的文本来自函数对象的 __doc__ 属性

示例 5-2 展示了函数对象的“一等”本性。我们可以把 factorial 函数赋值给变量 fact，然后通过变量名调用。我们还能把它作为参数传给 map 函数。map 函数返回一个可迭代对象，里面的元素是把第一个参数（一个函数）应用到第二个参数（一个可迭代对象，这里是 range(11)）中各个元素上得到的结果。

**示例 5-2** 通过别的名称使用函数，再把函数作为参数传递

```
>>> fact = factorial
>>> fact
<function factorial at 0x...>
>>> fact(5)
120
>>> map(factorial, range(11))
<map object at 0x...>
```

```
>>> list(map(fact, range(11)))
[1, 1, 2, 6, 24, 120, 720, 5040, 40320, 362880, 3628800]
```

有了一等函数，就可以使用函数式风格编程。函数式编程的特点之一是使用高阶函数——这是下一节的话题。

## 5.2 高阶函数

接受函数为参数，或者把函数作为结果返回的函数是高阶函数（higher-order function）。map 函数就是一例，如示例 5-2 所示。此外，内置函数 sorted 也是；可选的 key 参数用于提供一个函数，它会应用到各个元素上进行排序，参见 2.7 节。

例如，若想根据单词的长度排序，只需把 len 函数传给 key 参数，如示例 5-3 所示。

**示例 5-3　根据单词长度给一个列表排序**

```
>>> fruits = ['strawberry', 'fig', 'apple', 'cherry', 'raspberry', 'banana']
>>> sorted(fruits, key=len)
['fig', 'apple', 'cherry', 'banana', 'raspberry', 'strawberry']
>>>
```

任何单参数函数都能作为 key 参数的值。例如，为了创建押韵词典，可以把各个单词反过来拼写，然后排序。注意，示例 5-4 中列表里的单词没有变，我们只是把反向拼写当作排序条件，因此各种浆果（berry）都排在一起。

**示例 5-4　根据反向拼写给一个单词列表排序**

```
>>> def reverse(word):
...     return word[::-1]
>>> reverse('testing')
'gnitset'
>>> sorted(fruits, key=reverse)
['banana', 'apple', 'fig', 'raspberry', 'strawberry', 'cherry']
>>>
```

在函数式编程范式中，最为人熟知的高阶函数有 map、filter、reduce 和 apply。apply 函数在 Python 2.3 中标记为过时，在 Python 3 中移除了，因为不再需要它了。如果想使用不定量的参数调用函数，可以编写 fn(*args, **keywords)，不用再编写 apply(fn, args, kwargs)。

map、filter 和 reduce 这三个高阶函数还能见到，不过多数使用场景下都有更好的替代品。详情参阅下一节。

### map、filter和reduce的现代替代品

函数式语言通常会提供 map、filter 和 reduce 三个高阶函数（有时使用不同的名称）。在 Python 3 中，map 和 filter 还是内置函数，但是由于引入了列表推导和生成器表达式，它们变得没那么重要了。列表推导或生成器表达式具有 map 和 filter 两个函数的功能，而且更易于阅读，如示例 5-5 所示。

**示例 5-5　计算阶乘列表：map 和 filter 与列表推导比较**

```
>>> list(map(fact, range(6)))  ❶
[1, 1, 2, 6, 24, 120]
>>> [fact(n) for n in range(6)]  ❷
[1, 1, 2, 6, 24, 120]
>>> list(map(factorial, filter(lambda n: n % 2, range(6))))  ❸
[1, 6, 120]
>>> [factorial(n) for n in range(6) if n % 2]  ❹
[1, 6, 120]
>>>
```

❶ 构建 0! 到 5! 的一个阶乘列表。
❷ 使用列表推导执行相同的操作。
❸ 使用 map 和 filter 计算直到 5! 的奇数阶乘列表。
❹ 使用列表推导做相同的工作，换掉 map 和 filter，并避免了使用 lambda 表达式。

在 Python 3 中，map 和 filter 返回生成器（一种迭代器），因此现在它们的直接替代品是生成器表达式（在 Python 2 中，这两个函数返回列表，因此最接近的替代品是列表推导）。

在 Python 2 中，reduce 是内置函数，但是在 Python 3 中放到 functools 模块里了。这个函数最常用于求和，自 2003 年发布的 Python 2.3 开始，最好使用内置的 sum 函数。在可读性和性能方面，这是一项重大改善（见示例 5-6）。

**示例 5-6　使用 reduce 和 sum 计算 0~99 之和**

```
>>> from functools import reduce  ❶
>>> from operator import add  ❷
>>> reduce(add, range(100))  ❸
4950
>>> sum(range(100))  ❹
4950
>>>
```

❶ 从 Python 3.0 起，reduce 不再是内置函数了。
❷ 导入 add，以免创建一个专求两数之和的函数。
❸ 计算 0~99 之和。
❹ 使用 sum 做相同的求和；无需导入或创建求和函数。

sum 和 reduce 的通用思想是把某个操作连续应用到序列的元素上，累计之前的结果，把一系列值归约成一个值。

all 和 any 也是内置的归约函数。

all(iterable)

　如果 iterable 的每个元素都是真值，返回 True；all([]) 返回 True。

any(iterable)

　只要 iterable 中有元素是真值，就返回 True；any([]) 返回 False。

10.6 节将深入说明 reduce 函数，我会不断改进一个示例，为这个函数提供有意义的上下

文。本书后面的 14.11 节将重点讨论可迭代对象，届时会概述各个归约函数。

为了使用高阶函数，有时创建一次性的小型函数更便利。这便是匿名函数存在的原因，下一节将会讨论。

# 5.3 匿名函数

lambda 关键字在 Python 表达式内创建匿名函数。

然而，Python 简单的句法限制了 lambda 函数的定义体只能使用纯表达式。换句话说，lambda 函数的定义体中不能赋值，也不能使用 while 和 try 等 Python 语句。

在参数列表中最适合使用匿名函数。例如，示例 5-7 使用 lambda 表达式重写了示例 5-4 中排序押韵单词的示例，这样就省掉了 reverse 函数。

**示例 5-7　使用 lambda 表达式反转拼写，然后依此给单词列表排序**

```
>>> fruits = ['strawberry', 'fig', 'apple', 'cherry', 'raspberry', 'banana']
>>> sorted(fruits, key=lambda word: word[::-1])
['banana', 'apple', 'fig', 'raspberry', 'strawberry', 'cherry']
>>>
```

除了作为参数传给高阶函数之外，Python 很少使用匿名函数。由于句法上的限制，非平凡的 lambda 表达式要么难以阅读，要么无法写出。

### Lundh 提出的 lambda 表达式重构秘笈

如果使用 lambda 表达式导致一段代码难以理解，Fredrik Lundh 建议像下面这样重构。

(1) 编写注释，说明 lambda 表达式的作用。
(2) 研究一会儿注释，并找出一个名称来概括注释。
(3) 把 lambda 表达式转换成 def 语句，使用那个名称来定义函数。
(4) 删除注释。

这几步摘自“Functional Programming HOWTO”（https://docs.python.org/3/howto/functional.html），这是一篇必读文章。

lambda 句法只是语法糖：与 def 语句一样，lambda 表达式会创建函数对象。这是 Python 中几种可调用对象的一种。下一节会说明所有可调用对象。

# 5.4 可调用对象

除了用户定义的函数，调用运算符（即 ()）还可以应用到其他对象上。如果想判断对象能否调用，可以使用内置的 callable() 函数。Python 数据模型文档列出了 7 种可调用对象。

用户定义的函数

使用 def 语句或 lambda 表达式创建。

内置函数

使用 C 语言（CPython）实现的函数，如 len 或 time.strftime。

内置方法

使用 C 语言实现的方法，如 dict.get。

方法

在类的定义体中定义的函数。

类

调用类时会运行类的 __new__ 方法创建一个实例，然后运行 __init__ 方法，初始化实例，最后把实例返回给调用方。因为 Python 没有 new 运算符，所以调用类相当于调用函数。（通常，调用类会创建那个类的实例，不过覆盖 __new__ 方法的话，也可能出现其他行为。19.1.3 节会见到一个例子。）

类的实例

如果类定义了 __call__ 方法，那么它的实例可以作为函数调用。参见 5.5 节。

生成器函数

使用 yield 关键字的函数或方法。调用生成器函数返回的是生成器对象。

生成器函数在很多方面与其他可调用对象不同，详情参见第 14 章。生成器函数还可以作为协程，参见第 16 章。

Python 中有各种各样可调用的类型，因此判断对象能否调用，最安全的方法是使用内置的 callable() 函数：

```
>>> abs, str, 13
(<built-in function abs>, <class 'str'>, 13)
>>> [callable(obj) for obj in (abs, str, 13)]
[True, True, False]
```

接下来说明如何把类的实例变成可调用的对象。

## 5.5 用户定义的可调用类型

不仅 Python 函数是真正的对象，任何 Python 对象都可以表现得像函数。为此，只需实现实例方法 __call__。

示例 5-8 实现了 BingoCage 类。这个类的实例使用任何可迭代对象构建，而且会在内部存储一个随机顺序排列的列表。调用实例会取出一个元素。

**示例 5-8** bingocall.py：调用 BingoCage 实例，从打乱的列表中取出一个元素

```
import random

class BingoCage:
```

```
    def __init__(self, items):
        self._items = list(items)  ❶
        random.shuffle(self._items)  ❷

    def pick(self):  ❸
        try:
            return self._items.pop()
        except IndexError:
            raise LookupError('pick from empty BingoCage')  ❹

    def __call__(self):  ❺
        return self.pick()
```

❶ `__init__` 接受任何可迭代对象；在本地构建一个副本，防止列表参数的意外副作用。
❷ `shuffle` 定能完成工作，因为 `self._items` 是列表。
❸ 起主要作用的方法。
❹ 如果 `self._items` 为空，抛出异常，并设定错误消息。
❺ `bingo.pick()` 的快捷方式是 `bingo()`。

下面是示例 5-8 中定义的类的简单演示。注意，`bingo` 实例可以作为函数调用，而且内置的 `callable(...)` 函数判定它是可调用的对象：

```
>>> bingo = BingoCage(range(3))
>>> bingo.pick()
1
>>> bingo()
0
>>> callable(bingo)
True
```

实现 `__call__` 方法的类是创建函数类对象的简便方式，此时必须在内部维护一个状态，让它在调用之间可用，例如 `BingoCage` 中的剩余元素。装饰器就是这样。装饰器必须是函数，而且有时要在多次调用之间“记住”某些事 [ 例如备忘（memoization），即缓存消耗大的计算结果，供后面使用 ]。

创建保有内部状态的函数，还有一种截然不同的方式——使用闭包。闭包和装饰器在第 7 章讨论。

下面讨论把函数视作对象处理的另一方面：运行时内省。

## 5.6 函数内省

除了 `__doc__`，函数对象还有很多属性。使用 `dir` 函数可以探知 `factorial` 具有下述属性：

```
>>> dir(factorial)
['__annotations__', '__call__', '__class__', '__closure__', '__code__',
'__defaults__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__',
'__format__', '__ge__', '__get__', '__getattribute__', '__globals__',
'__gt__', '__hash__', '__init__', '__kwdefaults__', '__le__', '__lt__',
'__module__', '__name__', '__ne__', '__new__', '__qualname__', '__reduce__',
'__reduce_ex__', '__repr__', '__setattr__', '__sizeof__', '__str__',
```

```
'__subclasshook__']
>>>
```

其中大多数属性是 Python 对象共有的。本节讨论与把函数视作对象相关的几个属性，先从 `__dict__` 开始。

与用户定义的常规类一样，函数使用 `__dict__` 属性存储赋予它的用户属性。这相当于一种基本形式的注解。一般来说，为函数随意赋予属性不是很常见的做法，但是 Django 框架这么做了。参见“The Django admin site”文档（https://docs.djangoproject.com/en/1.10/ref/contrib/admin/）中对 `short_description`、`boolean` 和 `allow_tags` 属性的说明。这篇 Django 文档中举了下述示例，把 `short_description` 属性赋予一个方法，Django 管理后台使用这个方法时，在记录列表中会出现指定的描述文本：

```
def upper_case_name(obj):
    return ("%s %s" % (obj.first_name, obj.last_name)).upper()
upper_case_name.short_description = 'Customer name'
```

下面重点说明函数专有而用户定义的一般对象没有的属性。计算两个属性集合的差集便能得到函数专有属性列表（见示例 5-9）。

**示例 5-9　列出常规对象没有而函数有的属性**

```
>>> class C: pass  # ❶
>>> obj = C()  # ❷
>>> def func(): pass  # ❸
>>> sorted(set(dir(func)) - set(dir(obj))) # ❹
['__annotations__', '__call__', '__closure__', '__code__', '__defaults__',
'__get__', '__globals__', '__kwdefaults__', '__name__', '__qualname__']
>>>
```

❶ 创建一个空的用户定义的类。
❷ 创建一个实例。
❸ 创建一个空函数。
❹ 计算差集，然后排序，得到类的实例没有而函数有的属性列表。

表 5-1 对示例 5-9 中列出的属性做了简要说明。

**表5-1：用户定义的函数的属性**

| 名称 | 类型 | 说明 |
|---|---|---|
| `__annotations__` | `dict` | 参数和返回值的注解 |
| `__call__` | `method-wrapper` | 实现 () 运算符；即可调用对象协议 |
| `__closure__` | `tuple` | 函数闭包，即自由变量的绑定（通常是 `None`） |
| `__code__` | `code` | 编译成字节码的函数元数据和函数定义体 |
| `__defaults__` | `tuple` | 形式参数的默认值 |
| `__get__` | `method-wrapper` | 实现只读描述符协议（参见第 20 章） |
| `__globals__` | `dict` | 函数所在模块中的全局变量 |
| `__kwdefaults__` | `dict` | 仅限关键字形式参数的默认值 |

（续）

| 名称 | 类型 | 说明 |
| --- | --- | --- |
| __name__ | str | 函数名称 |
| __qualname__ | str | 函数的限定名称，如 Random.choice（参阅 PEP 3155，https://www.python.org/dev/peps/pep-3155/） |

后面几节会讨论 __defaults__、__code__ 和 __annotations__ 属性，IDE 和框架使用它们提取关于函数签名的信息。但是，为了深入了解这些属性，我们要先探讨 Python 为声明函数形参和传入实参所提供的强大句法。

## 5.7 从定位参数到仅限关键字参数

Python 最好的特性之一是提供了极为灵活的参数处理机制，而且 Python 3 进一步提供了仅限关键字参数（keyword-only argument）。与之密切相关的是，调用函数时使用 * 和 ** “展开”可迭代对象，映射到单个参数。下面通过示例 5-10 中的代码展示这些特性，实际使用的代码在示例 5-11 中。

**示例 5-10** tag 函数用于生成 HTML 标签；使用名为 cls 的关键字参数传入“class”属性，这是一种变通方法，因为“class”是 Python 的关键字

```
def tag(name, *content, cls=None, **attrs):
    """生成一个或多个HTML标签"""
    if cls is not None:
        attrs['class'] = cls
    if attrs:
        attr_str = ''.join(' %s="%s"' % (attr, value)
                           for attr, value
                           in sorted(attrs.items()))
    else:
        attr_str = ''
    if content:
        return '\n'.join('<%s%s>%s</%s>' %
                         (name, attr_str, c, name) for c in content)
    else:
        return '<%s%s />' % (name, attr_str)
```

tag 函数的调用方式很多，如示例 5-11 所示。

**示例 5-11** tag 函数（见示例 5-10）众多调用方式中的几种

```
>>> tag('br') ❶
'<br />'
>>> tag('p', 'hello') ❷
'<p>hello</p>'
>>> print(tag('p', 'hello', 'world'))
<p>hello</p>
<p>world</p>
>>> tag('p', 'hello', id=33) ❸
'<p id="33">hello</p>'
>>> print(tag('p', 'hello', 'world', cls='sidebar')) ❹
```

```
<p class="sidebar">hello</p>
<p class="sidebar">world</p>
>>> tag(content='testing', name="img")  ❺
'<img content="testing" />'
>>> my_tag = {'name': 'img', 'title': 'Sunset Boulevard',
...           'src': 'sunset.jpg', 'cls': 'framed'}
>>> tag(**my_tag)  ❻
'<img class="framed" src="sunset.jpg" title="Sunset Boulevard" />'
```

❶ 传入单个定位参数，生成一个指定名称的空标签。
❷ 第一个参数后面的任意个参数会被 *content 捕获，存入一个元组。
❸ tag 函数签名中没有明确指定名称的关键字参数会被 **attrs 捕获，存入一个字典。
❹ cls 参数只能作为关键字参数传入。
❺ 调用 tag 函数时，即便第一个定位参数也能作为关键字参数传入。
❻ 在 my_tag 前面加上 **，字典中的所有元素作为单个参数传入，同名键会绑定到对应的具名参数上，余下的则被 **attrs 捕获。

仅限关键字参数是 Python 3 新增的特性。在示例 5-10 中，cls 参数只能通过关键字参数指定，它一定不会捕获未命名的定位参数。定义函数时若想指定仅限关键字参数，要把它们放到前面有 * 的参数后面。如果不想支持数量不定的定位参数，但是想支持仅限关键字参数，在签名中放一个 *，如下所示：

```
>>> def f(a, *, b):
...     return a, b
...
>>> f(1, b=2)
(1, 2)
```

注意，仅限关键字参数不一定要有默认值，可以像上例中 b 那样，强制必须传入实参。

下面说明函数参数的内省，以一个 Web 框架中的示例为引子，然后再讨论内省技术。

## 5.8 获取关于参数的信息

HTTP 微框架 Bobo 中有个使用函数内省的好例子。示例 5-12 是对 Bobo 教程中“Hello world”应用的改编，说明了内省怎么使用。

**示例 5-12** Bobo 知道 hello 需要 person 参数，并且从 HTTP 请求中获取它

```
import bobo

@bobo.query('/')
def hello(person):
    return 'Hello %s!' % person
```

bobo.query 装饰器把一个普通的函数（如 hello）与框架的请求处理机制集成起来了。装饰器会在第 7 章讨论，这不是这个示例的关键。这里的关键是，Bobo 会内省 hello 函数，发现它需要一个名为 person 的参数，然后从请求中获取那个名称对应的参数，将其传给 hello 函数，因此程序员根本不用触碰请求对象。

安装 Bobo，然后启动开发服务器，执行示例 5-12 中的脚本（例如，bobo -f hello.py）。访问 http://localhost:8080/ 看到的消息是“Missing form variable person”，HTTP 状态码是 403。这是因为，Bobo 知道调用 hello 函数必须传入 person 参数，但是在请求中找不到同名参数。示例 5-13 在 shell 会话中使用 curl 展示了这个行为。

**示例 5-13　如果请求中缺少函数的参数，Bobo 返回 403 forbidden 响应；curl -i 的作用是把首部转储到标准输出**

```
$ curl -i http://localhost:8080/
HTTP/1.0 403 Forbidden
Date: Thu, 21 Aug 2014 21:39:44 GMT
Server: WSGIServer/0.2 CPython/3.4.1
Content-Type: text/html; charset=UTF-8
Content-Length: 103

<html>
<head><title>Missing parameter</title></head>
<body>Missing form variable person</body>
</html>
```

然而，如果访问 http://localhost:8080/?person=Jim，响应会变成字符串 'Hello Jim!'，如示例 5-14 所示。

**示例 5-14　传入所需的 person 参数才能得到 OK 响应**

```
$ curl -i http://localhost:8080/?person=Jim
HTTP/1.0 200 OK
Date: Thu, 21 Aug 2014 21:42:32 GMT
Server: WSGIServer/0.2 CPython/3.4.1
Content-Type: text/html; charset=UTF-8
Content-Length: 10

Hello Jim!
```

Bobo 是怎么知道函数需要哪个参数的呢？它又是怎么知道参数有没有默认值呢？

函数对象有个 __defaults__ 属性，它的值是一个元组，里面保存着定位参数和关键字参数的默认值。仅限关键字参数的默认值在 __kwdefaults__ 属性中。然而，参数的名称在 __code__ 属性中，它的值是一个 code 对象引用，自身也有很多属性。

为了说明这些属性的用途，下面在 clip.py 模块中定义 clip 函数，如示例 5-15 所示，然后再审查它。

**示例 5-15　在指定长度附近截断字符串的函数**

```
def clip(text, max_len=80):
    """在max_len前面或后面的第一个空格处截断文本
    """
    end = None
    if len(text) > max_len:
        space_before = text.rfind(' ', 0, max_len)
        if space_before >= 0:
            end = space_before
        else:
```

```
            space_after = text.rfind(' ', max_len)
            if space_after >= 0:
                end = space_after
    if end is None:  # 没找到空格
        end = len(text)
    return text[:end].rstrip()
```

示例 5-16 审查示例 5-15 中定义的 clip 函数，查看 __defaults__、__code__.co_varnames 和 __code__.co_argcount 的值。

**示例 5-16　提取关于函数参数的信息**

```
>>> from clip import clip
>>> clip.__defaults__
(80,)
>>> clip.__code__  # doctest: +ELLIPSIS
<code object clip at 0x...>
>>> clip.__code__.co_varnames
('text', 'max_len', 'end', 'space_before', 'space_after')
>>> clip.__code__.co_argcount
2
```

可以看出，这种组织信息的方式并不是最便利的。参数名称在 __code__.co_varnames 中，不过里面还有函数定义体中创建的局部变量。因此，参数名称是前 *N* 个字符串，*N* 的值由 __code__.co_argcount 确定。顺便说一下，这里不包含前缀为 * 或 ** 的变长参数。参数的默认值只能通过它们在 __defaults__ 元组中的位置确定，因此要从后向前扫描才能把参数和默认值对应起来。在这个示例中 clip 函数有两个参数，text 和 max_len，其中一个有默认值，即 80，因此它必然属于最后一个参数，即 max_len。这有违常理。

幸好，我们有更好的方式——使用 inspect 模块。

下面来看一下示例 5-17。

**示例 5-17　提取函数的签名**[2]

```
>>> from clip import clip
>>> from inspect import signature
>>> sig = signature(clip)
>>> sig  # doctest: +ELLIPSIS
<inspect.Signature object at 0x...>
>>> str(sig)
'(text, max_len=80)'
>>> for name, param in sig.parameters.items():
...     print(param.kind, ':', name, '=', param.default)
...
POSITIONAL_OR_KEYWORD : text = <class 'inspect._empty'>
POSITIONAL_OR_KEYWORD : max_len = 80
```

这样就好多了。inspect.signature 函数返回一个 inspect.Signature 对象，它有一个 parameters 属性，这是一个有序映射，把参数名和 inspect.Parameter 对象对应起来。各个 Parameter 属性也有自己的属性，例如 name、default 和 kind。特殊的 inspect._empty 值表示没有默认值，考

注 2：在 Python 3.5 中，本示例的 sig 的值是：<Signature (text, max_len=80)>。 ——编者注

虑到 None 是有效的默认值（也经常这么做），而且这么做是合理的。

kind 属性的值是 _ParameterKind 类中的 5 个值之一，列举如下。

POSITIONAL_OR_KEYWORD

可以通过定位参数和关键字参数传入的形参（多数 Python 函数的参数属于此类）。

VAR_POSITIONAL

定位参数元组。

VAR_KEYWORD

关键字参数字典。

KEYWORD_ONLY

仅限关键字参数（Python 3 新增）。

POSITIONAL_ONLY

仅限定位参数；目前，Python 声明函数的句法不支持，但是有些使用 C 语言实现且不接受关键字参数的函数（如 divmod）支持。

除了 name、default 和 kind，inspect.Parameter 对象还有一个 annotation（注解）属性，它的值通常是 inspect._empty，但是可能包含 Python 3 新的注解句法提供的函数签名元数据（注解在下一节讨论）。

inspect.Signature 对象有个 bind 方法，它可以把任意个参数绑定到签名中的形参上，所用的规则与实参到形参的匹配方式一样。框架可以使用这个方法在真正调用函数前验证参数，如示例 5-18 所示。

**示例 5-18　把 tag 函数（见示例 5-10）的签名绑定到一个参数字典上**[3]

```
>>> import inspect
>>> sig = inspect.signature(tag)  ❶
>>> my_tag = {'name': 'img', 'title': 'Sunset Boulevard',
...           'src': 'sunset.jpg', 'cls': 'framed'}
>>> bound_args = sig.bind(**my_tag)  ❷
>>> bound_args
<inspect.BoundArguments object at 0x...>  ❸
>>> for name, value in bound_args.arguments.items():  ❹
...     print(name, '=', value)
...
name = img
cls = framed
attrs = {'title': 'Sunset Boulevard', 'src': 'sunset.jpg'}
>>> del my_tag['name']  ❺
>>> bound_args = sig.bind(**my_tag)  ❻
Traceback (most recent call last):
```

注 3：在 Python 3.5 中，本示例的 bound_args 的值是：<BoundArguments (name='img', cls='framed', attrs={'title': 'Sunset Boulevard', 'src': 'sunset.jpg'})>。——编者注

```
    ...
TypeError: 'name' parameter lacking default value
```

❶ 获取 tag 函数（见示例 5-10）的签名。
❷ 把一个字典参数传给 .bind() 方法。
❸ 得到一个 inspect.BoundArguments 对象。
❹ 迭代 bound_args.arguments（一个 OrderedDict 对象）中的元素，显示参数的名称和值。
❺ 把必须指定的参数 name 从 my_tag 中删除。
❻ 调用 sig.bind(**my_tag)，抛出 TypeError，抱怨缺少 name 参数。

这个示例在 inspect 模块的帮助下，展示了 Python 数据模型把实参绑定给函数调用中的形参的机制，这与解释器使用的机制相同。

框架和 IDE 等工具可以使用这些信息验证代码。Python 3 的另一个特性——函数注解——增进了这些信息的用途，参见下一节。

## 5.9 函数注解

Python 3 提供了一种句法，用于为函数声明中的参数和返回值附加元数据。示例 5-19 是示例 5-15 添加注解后的版本，二者唯一的区别在第一行。

**示例 5-19 有注解的 clip 函数**

```
def clip(text:str, max_len:'int > 0'=80) -> str:  ❶
    """在max_len前面或后面的第一个空格处截断文本
    """
    end = None
    if len(text) > max_len:
        space_before = text.rfind(' ', 0, max_len)
        if space_before >= 0:
            end = space_before
        else:
            space_after = text.rfind(' ', max_len)
            if space_after >= 0:
                end = space_after
    if end is None:  # 没找到空格
        end = len(text)
    return text[:end].rstrip()
```

❶ 有注解的函数声明。

函数声明中的各个参数可以在 : 之后增加注解表达式。如果参数有默认值，注解放在参数名和 = 号之间。如果想注解返回值，在 ) 和函数声明末尾的 : 之间添加 -> 和一个表达式。那个表达式可以是任何类型。注解中最常用的类型是类（如 str 或 int）和字符串（如 'int > 0'）。在示例 5-19 中，max_len 参数的注解用的是字符串。

注解不会做任何处理，只是存储在函数的 __annotations__ 属性（一个字典）中：

```
>>> from clip_annot import clip
>>> clip.__annotations__
{'text': <class 'str'>, 'max_len': 'int > 0', 'return': <class 'str'>}
```

'return' 键保存的是返回值注解，即示例 5-19 中函数声明里以 -> 标记的部分。

Python 对注解所做的唯一的事情是，把它们存储在函数的 __annotations__ 属性里。仅此而已，Python 不做检查、不做强制、不做验证，什么操作都不做。换句话说，注解对 Python 解释器没有任何意义。注解只是元数据，可以供 IDE、框架和装饰器等工具使用。写作本书时，标准库中还没有什么会用到这些元数据，唯有 inspect.signature() 函数知道怎么提取注解，如示例 5-20 所示。

**示例 5-20　从函数签名中提取注解**

```
>>> from clip_annot import clip
>>> from inspect import signature
>>> sig = signature(clip)
>>> sig.return_annotation
<class 'str'>
>>> for param in sig.parameters.values():
...     note = repr(param.annotation).ljust(13)
...     print(note, ':', param.name, '=', param.default)
<class 'str'> : text = <class 'inspect._empty'>
'int > 0'     : max_len = 80
```

signature 函数返回一个 Signature 对象，它有一个 return_annotation 属性和一个 parameters 属性，后者是一个字典，把参数名映射到 Parameter 对象上。每个 Parameter 对象自己也有 annotation 属性。示例 5-20 用到了这几个属性。

在未来，Bobo 等框架可以支持注解，并进一步自动处理请求。例如，使用 price:float 注解的参数可以自动把查询字符串转换成函数期待的 float 类型；quantity:'int > 0' 这样的字符串注解可以转换成对参数的验证。

函数注解的最大影响或许不是让 Bobo 等框架自动设置，而是为 IDE 和 lint 程序等工具中的静态类型检查功能提供额外的类型信息。

深入分析函数之后，本章余下的内容介绍标准库中为函数式编程提供支持的常用包。

## 5.10　支持函数式编程的包

虽然 Guido 明确表明，Python 的目标不是变成函数式编程语言，但是得益于 operator 和 functools 等包的支持，函数式编程风格也可以信手拈来。接下来的两节分别介绍这两个包。

### 5.10.1　operator模块

在函数式编程中，经常需要把算术运算符当作函数使用。例如，不使用递归计算阶乘。求和可以使用 sum 函数，但是求积则没有这样的函数。我们可以使用 reduce 函数（5.2.1 节是这么做的），但是需要一个函数计算序列中两个元素之积。示例 5-21 展示如何使用 lambda 表达式解决这个问题。

**示例 5-21　使用 reduce 函数和一个匿名函数计算阶乘**

```
from functools import reduce
```

```
def fact(n):
    return reduce(lambda a, b: a*b, range(1, n+1))
```

operator 模块为多个算术运算符提供了对应的函数，从而避免编写 lambda a, b: a*b 这种平凡的匿名函数。使用算术运算符函数，可以把示例 5-21 改写成示例 5-22 那样。

**示例 5-22　使用 reduce 和 operator.mul 函数计算阶乘**

```
from functools import reduce
from operator import mul

def fact(n):
    return reduce(mul, range(1, n+1))
```

operator 模块中还有一类函数，能替代从序列中取出元素或读取对象属性的 lambda 表达式：因此，itemgetter 和 attrgetter 其实会自行构建函数。

示例 5-23 展示了 itemgetter 的常见用途：根据元组的某个字段给元组列表排序。在这个示例中，按照国家代码（第 2 个字段）的顺序打印各个城市的信息。其实，itemgetter(1) 的作用与 lambda fields: fields[1] 一样：创建一个接受集合的函数，返回索引位 1 上的元素。

**示例 5-23　演示使用 itemgetter 排序一个元组列表（数据来自示例 2-8）**

```
>>> metro_data = [
...     ('Tokyo', 'JP', 36.933, (35.689722, 139.691667)),
...     ('Delhi NCR', 'IN', 21.935, (28.613889, 77.208889)),
...     ('Mexico City', 'MX', 20.142, (19.433333, -99.133333)),
...     ('New York-Newark', 'US', 20.104, (40.808611, -74.020386)),
...     ('Sao Paulo', 'BR', 19.649, (-23.547778, -46.635833)),
... ]
>>>
>>> from operator import itemgetter
>>> for city in sorted(metro_data, key=itemgetter(1)):
...     print(city)
...
('Sao Paulo', 'BR', 19.649, (-23.547778, -46.635833))
('Delhi NCR', 'IN', 21.935, (28.613889, 77.208889))
('Tokyo', 'JP', 36.933, (35.689722, 139.691667))
('Mexico City', 'MX', 20.142, (19.433333, -99.133333))
('New York-Newark', 'US', 20.104, (40.808611, -74.020386))
```

如果把多个参数传给 itemgetter，它构建的函数会返回提取的值构成的元组：

```
>>> cc_name = itemgetter(1, 0)
>>> for city in metro_data:
...     print(cc_name(city))
...
('JP', 'Tokyo')
('IN', 'Delhi NCR')
('MX', 'Mexico City')
('US', 'New York-Newark')
('BR', 'Sao Paulo')
>>>
```

itemgetter 使用 [] 运算符，因此它不仅支持序列，还支持映射和任何实现 __getitem__ 方法的类。

attrgetter 与 itemgetter 作用类似，它创建的函数根据名称提取对象的属性。如果把多个属性名传给 attrgetter，它也会返回提取的值构成的元组。此外，如果参数名中包含 .（点号），attrgetter 会深入嵌套对象，获取指定的属性。这些行为如示例 5-24 所示。这个控制台会话不短，因为我们要构建一个嵌套结构，这样才能展示 attrgetter 如何处理包含点号的属性名。

**示例 5-24** 定义一个 namedtuple，名为 metro_data（与示例 5-23 中的列表相同），演示使用 attrgetter 处理它

```
>>> from collections import namedtuple
>>> LatLong = namedtuple('LatLong', 'lat long')  # ❶
>>> Metropolis = namedtuple('Metropolis', 'name cc pop coord')  # ❷
>>> metro_areas = [Metropolis(name, cc, pop, LatLong(lat, long))  # ❸
...     for name, cc, pop, (lat, long) in metro_data]
>>> metro_areas[0]
Metropolis(name='Tokyo', cc='JP', pop=36.933, coord=LatLong(lat=35.689722,
long=139.691667))
>>> metro_areas[0].coord.lat  # ❹
35.689722
>>> from operator import attrgetter
>>> name_lat = attrgetter('name', 'coord.lat')  # ❺
>>>
>>> for city in sorted(metro_areas, key=attrgetter('coord.lat')):  # ❻
...     print(name_lat(city))  # ❼
...
('Sao Paulo', -23.547778)
('Mexico City', 19.433333)
('Delhi NCR', 28.613889)
('Tokyo', 35.689722)
('New York-Newark', 40.808611)
```

❶ 使用 namedtuple 定义 LatLong。
❷ 再定义 Metropolis。
❸ 使用 Metropolis 实例构建 metro_areas 列表；注意，我们使用嵌套的元组拆包提取 (lat, long)，然后使用它们构建 LatLong，作为 Metropolis 的 coord 属性。
❹ 深入 metro_areas[0]，获取它的纬度。
❺ 定义一个 attrgetter，获取 name 属性和嵌套的 coord.lat 属性。
❻ 再次使用 attrgetter，按照纬度排序城市列表。
❼ 使用标号❺中定义的 attrgetter，只显示城市名和纬度。

下面是 operator 模块中定义的部分函数（省略了以 _ 开头的名称，因为它们基本上是实现细节）：[4]

```
>>> [name for name in dir(operator) if not name.startswith('_')]
['abs', 'add', 'and_', 'attrgetter', 'concat', 'contains',
```

注 4：Python 3.5 中增加了 imatmul 和 matmul。 ——编者注

```
'countOf', 'delitem', 'eq', 'floordiv', 'ge', 'getitem', 'gt',
'iadd', 'iand', 'iconcat', 'ifloordiv', 'ilshift', 'imod', 'imul',
'index', 'indexOf', 'inv', 'invert', 'ior', 'ipow', 'irshift',
'is_', 'is_not', 'isub', 'itemgetter', 'itruediv', 'ixor', 'le',
'length_hint', 'lshift', 'lt', 'methodcaller', 'mod', 'mul', 'ne',
'neg', 'not_', 'or_', 'pos', 'pow', 'rshift', 'setitem', 'sub',
'truediv', 'truth', 'xor']
```

这 52 个名称中大部分的作用不言而喻。以 i 开头、后面是另一个运算符的那些名称（如 iadd、iand 等），对应的是增量赋值运算符（如 +=、&= 等）。如果第一个参数是可变的，那么这些运算符函数会就地修改它；否则，作用与不带 i 的函数一样，直接返回运算结果。

在 operator 模块余下的函数中，我们最后介绍一下 methodcaller。它的作用与 attrgetter 和 itemgetter 类似，它会自行创建函数。methodcaller 创建的函数会在对象上调用参数指定的方法，如示例 5-25 所示。

**示例 5-25** methodcaller 使用示例：第二个测试展示绑定额外参数的方式

```
>>> from operator import methodcaller
>>> s = 'The time has come'
>>> upcase = methodcaller('upper')
>>> upcase(s)
'THE TIME HAS COME'
>>> hiphenate = methodcaller('replace', ' ', '-')
>>> hiphenate(s)
'The-time-has-come'
```

示例 5-25 中的第一个测试只是为了展示 methodcaller 的用法，如果想把 str.upper 作为函数使用，只需在 str 类上调用，并传入一个字符串参数，如下所示：

```
>>> str.upper(s)
'THE TIME HAS COME'
```

示例 5-25 中的第二个测试表明，methodcaller 还可以冻结某些参数，也就是部分应用（partial application），这与 functools.partial 函数的作用类似。详情参见下一节。

## 5.10.2 使用functools.partial冻结参数

functools 模块提供了一系列高阶函数，其中最为人熟知的或许是 reduce，我们在 5.2.1 节已经介绍过。余下的函数中，最有用的是 partial 及其变体，partialmethod。

functools.partial 这个高阶函数用于部分应用一个函数。部分应用是指，基于一个函数创建一个新的可调用对象，把原函数的某些参数固定。使用这个函数可以把接受一个或多个参数的函数改编成需要回调的 API，这样参数更少。示例 5-26 做了简单的演示。

**示例 5-26** 使用 partial 把一个两参数函数改编成需要单参数的可调用对象

```
>>> from operator import mul
>>> from functools import partial
>>> triple = partial(mul, 3)  ❶
>>> triple(7)  ❷
```

```
21
>>> list(map(triple, range(1, 10)))  ❸
[3, 6, 9, 12, 15, 18, 21, 24, 27]
```

❶ 使用 mul 创建 triple 函数，把第一个定位参数定为 3。

❷ 测试 triple 函数。

❸ 在 map 中使用 triple；在这个示例中不能使用 mul。

使用 4.6 节介绍的 unicode.normalize 函数再举个例子，这个示例更有实际意义。如果处理多国语言编写的文本，在比较或排序之前可能会想使用 unicode.normalize('NFC', s) 处理所有字符串 s。如果经常这么做，可以定义一个 nfc 函数，如示例 5-27 所示。

**示例 5-27　使用 partial 构建一个便利的 Unicode 规范化函数**

```
>>> import unicodedata, functools
>>> nfc = functools.partial(unicodedata.normalize, 'NFC')
>>> s1 = 'café'
>>> s2 = 'cafe\u0301'
>>> s1, s2
('café', 'café')
>>> s1 == s2
False
>>> nfc(s1) == nfc(s2)
True
```

partial 的第一个参数是一个可调用对象，后面跟着任意个要绑定的定位参数和关键字参数。

示例 5-28 在示例 5-10 中定义的 tag 函数上使用 partial，冻结一个定位参数和一个关键字参数。

**示例 5-28　把 partial 应用到示例 5-10 中定义的 tag 函数上**

```
>>> from tagger import tag
>>> tag
<function tag at 0x10206d1e0>  ❶
>>> from functools import partial
>>> picture = partial(tag, 'img', cls='pic-frame')  ❷
>>> picture(src='wumpus.jpeg')
'<img class="pic-frame" src="wumpus.jpeg" />'  ❸
>>> picture
functools.partial(<function tag at 0x10206d1e0>, 'img', cls='pic-frame')  ❹
>>> picture.func  ❺
<function tag at 0x10206d1e0>
>>> picture.args
('img',)
>>> picture.keywords
{'cls': 'pic-frame'}
```

❶ 从示例 5-10 中导入 tag 函数，查看它的 ID。

❷ 使用 tag 创建 picture 函数，把第一个定位参数固定为 'img'，把 cls 关键字参数固定为 'pic-frame'。

❸ picture 的行为符合预期。

❹ partial() 返回一个 functools.partial 对象。[5]
❺ functools.partial 对象提供了访问原函数和固定参数的属性。

functools.partialmethod 函数（Python 3.4 新增）的作用与 partial 一样，不过是用于处理方法的。

functools 模块中的 lru_cache 函数令人印象深刻，它会做备忘（memoization），这是一种自动优化措施，它会存储耗时的函数调用结果，避免重新计算。第 7 章将会介绍这个函数，还将讨论装饰器，以及旨在用作装饰器的其他高阶函数：singledispatch 和 wraps。

## 5.11 本章小结

本章的目标是探讨 Python 函数的一等本性。这意味着，我们可以把函数赋值给变量、传给其他函数、存储在数据结构中，以及访问函数的属性，供框架和一些工具使用。高阶函数是函数式编程的重要组成部分，即使现在不像以前那样经常使用 map、filter 和 reduce 函数了，但是还有列表推导（以及类似的结构，如生成器表达式）以及 sum、all 和 any 等内置的归约函数。Python 中常用的高阶函数有内置函数 sorted、min、max 和 functools.partial。

Python 有 7 种可调用对象，从 lambda 表达式创建的简单函数到实现 __call__ 方法的类实例。这些可调用对象都能通过内置的 callable() 函数检测。每一种可调用对象都支持使用相同的丰富句法声明形式参数，包括仅限关键字参数和注解——二者都是 Python 3 引入的新特性。

Python 函数及其注解有丰富的属性，在 inspect 模块的帮助下，可以读取它们。例如，Signature.bind 方法使用灵活的规则把实参绑定到形参上，这与 Python 使用的规则一样。

最后，本章介绍了 operator 模块中的一些函数，以及 functools.partial 函数，有了这些函数，函数式编程就不太需要功能有限的 lambda 表达式了。

## 5.12 延伸阅读

接下来的两章继续探讨使用函数对象编程。第 6 章说明一等函数如何简化某些经典的面向对象设计模式，第 7 章说明函数装饰器（一种特别的高阶函数）和支持装饰器的闭包机制。

《Python Cookbook（第 3 版）中文版》（David Beazley 和 Brian K. Jones 著）的第 7 章是对本章和第 7 章很好的补充，那一章基本上使用不同的方式探讨了相同的概念。

Python 语言参考手册中的“3.2. The standard type hierarchy”一节（https://docs.python.org/3/reference/datamodel.html#the-standard-type-hierarchy）对 7 种可调用类型和其他所有内置类型做了介绍。

---

注 5：functools.py 的源码（https://hg.python.org/cpython/file/default/Lib/functools.py）表明，functools.partial 类是使用 C 语言实现的，而且默认使用这个实现。如果这个实现不可用，从 Python 3.4 起，functools 模块为 partial 提供了纯 Python 实现。

本章讨论的 Python 3 专有特性有各自的 PEP："PEP 3102—Keyword-Only Arguments"（https://www.python.org/dev/peps/pep-3102/）和"PEP 3107—Function Annotations"（https://www.python.org/dev/peps/pep-3107/）。

若想进一步了解目前对注解的使用，Stack Overflow 网站中有两个问答值得一读：一个是"What are good uses for Python3's 'Function Annotations'"（http://stackoverflow.com/questions/3038033/what-are-good-uses-for-python3s-function-annotations），Raymond Hettinger 给出了务实的回答和深入的见解；另一个是"What good are Python function annotations?"（http://stackoverflow.com/questions/13784713/what-good-are-python-function-annotations），某个回答中大量引用了 Guido van Rossum 的观点。

如果你想使用 `inspect` 模块，"PEP 362—Function Signature Object"（https://www.python.org/dev/peps/pep-0362/）值得一读，可以帮助了解实现细节。

A. M. Kuchling 的文章"Python Functional Programming HOWTO"（http://docs.python.org/3/howto/functional.html）对 Python 函数式编程做了很好的介绍。不过，那篇文章的重点是使用迭代器和生成器，这是第 14 章的话题。

`fn.py`（https://github.com/kachayev/fn.py）是为 Python 2 和 Python 3 提供函数式编程支持的包。据作者 Alexey Kachayev 介绍，`fn.py` 提供了 Python"所缺少的函数式特性"。这个包提供的 `@recur.tco` 装饰器为 Python 中的无限递归实现了尾调用优化。此外，`fn.py` 还提供了很多其他函数、数据结构和诀窍。

Stack Overflow 网站中的问题"Python: Why is functools.partial necessary?"（http://stackoverflow.com/questions/3252228/python-why-is-functools-partial-necessary）有个详实（而有趣）的回答，答主是 Alex Martelli，他是经典的《Python 技术手册》一书的作者。

Jim Fulton 开发的 Bobo 或许是第一个称得上是面向对象的 Web 框架。如果你对这个框架感兴趣，想进一步学习它最近的重写版本，先从"Introduction"（http://bobo.readthedocs.io/en/latest/）入手。在 Joel Spolsky 的博客中，Phillip J. Eby 在评论中提到了 Bobo 的一些早期历史（http://discuss.fogcreek.com/joelonsoftware/default.asp?cmd=show&ixPost=94006）。

## 杂　谈

### 关于 Bobo

我的 Python 编程生涯从 Bobo 开始。1998 年，我在自己的第一个 Python Web 项目中使用了 Bobo。当时我在寻找编写 Web 应用的面向对象方式，尝试过一些 Perl 和 Java 框架之后，我发现了 Bobo。

1997 年，Bobo 开创了对象发布概念：直接把 URL 映射到对象层次结构上，无需配置路由。看到这种做法的精妙之处后，我被 Bobo 吸引住了。Bobo 还能通过分析处理请求的方法或函数的签名来自动处理 HTTP 查询。

Bobo 由 Jim Fulton 创建，他被人称为“Zope 教皇”（The Zope Pope），因为他在 Zope 框架的开发中的起到领衔作用。Zope 是 Plone CMS、SchoolTool、ERP5 和其他大型 Python 项目的基础。Jim 还是 ZODB（Zope Object Database）的创建者，这是一个事务型对象数据库，提供了 ACID（“atomicity, consistency, isolation, and durability”，原子性、一致性、隔离性和耐久性），它的设计目的是简化 Python 的使用。

后来，为了支持 WSGI 和现代的 Python 版本（包括 Python 3），Jim 从头重写了 Bobo。写作本书时，Bobo 使用 `six` 库做函数内省，这是为了兼容 Python 2 和 Python 3，因为这两个版本在函数对象和相关的 API 上做了修改。

### Python 是函数式语言吗

2000 年左右，我在美国做培训，Guido van Rossum 到访了教室（他不是讲师）。在课后的问答环节，有人问他 Python 的哪些特性是从其他语言借鉴而来的。他答道：“Python 中一切好的特性都是从其他语言中借鉴来的。”

布朗大学的计算机科学教授 Shriram Krishnamurthi 在其论文“Teaching Programming Languages in a Post-Linnaean Age”（http://cs.brown.edu/~sk/Publications/Papers/Published/sk-teach-pl-post-linnaean/）的开头这样写道：

> 编程语言“范式”已近末日，它们是旧时代的遗留物，令人厌烦。既然现代语言的设计者对范式不屑一顾，那么我们的课程为什么要像奴隶一样对其言听计从？

在那篇论文中，下面这一段点名提到了 Python：

> 对 Python、Ruby 或 Perl 这些语言还要了解什么呢？它们的设计者没有耐心去精确实现林奈层次结构；设计者按照自己的意愿从别处借鉴特性，创建出完全无视过往概念的大杂烩。

Krishnamurthi 指出，不要试图把语言归为某一类；相反，把它们视作特性的聚合更有用。

为 Python 提供一等函数打开了函数式编程的大门，不过这并不是 Guido 的目的。他在“Origins of Python's Functional Features”一文（http://python-history.blogspot.com/2009/04/origins-of-pythons-functional-features.html）中说，`map`、`filter` 和 `reduce` 的最初目的是为 Python 增加 `lambda` 表达式。这些特性都由 Amrit Prem 贡献，添加在 1994 年发布的 Python 1.0 中（参见 CPython 源码中的 Misc/HISTORY 文件，https://hg.python.org/cpython/file/default/Misc/HISTORY）。

`lambda`、`map`、`filter` 和 `reduce` 首次出现在 Lisp 中，这是最早的一门函数式语言。然而，Lisp 没有限制在 `lambda` 表达式中能做什么，因为 Lisp 中的一切都是表达式。Python 使用的是面向语句的句法，表达式中不能包含语句，而很多语言结构都是语句，例如 `try/catch`，我编写 `lambda` 表达式时最想念这个语句。Python 为了提高句法的可读性，必须付出这样的代价。[6] Lisp 有很多优点，可读性一定不是其中之一。

注 6：此外，还有一个问题：把代码粘贴到 Web 论坛时，缩进会丢失。当然，这是题外话。

讽刺的是，从另一门函数式语言（Haskell）中借用列表推导之后，Python 对 map、filter，以及 lambda 表达式的需求极大地减少了。

除了匿名函数句法上的限制之外，影响函数式编程惯用法在 Python 中广泛使用的最大障碍是缺少尾递归消除（tail-recursion elimination），这是一项优化措施，在函数的定义体“末尾”递归调用，从而提高计算函数的内存使用效率。Guido 在另一篇博客文章（“Tail Recursion Elimination”，http://neopythonic.blogspot.com/2009/04/tail-recursion-elimination.html）中解释了为什么这种优化措施不适合 Python。这篇文章详细讨论了技术论证，不过前三个也是最重要的原因与易用性有关。Python 作为一门易于使用、学习和教授的语言并非偶然，有 Guido 在为我们把关。

综上，从设计上看，不管函数式语言的定义如何，Python 都不是一门函数式语言。Python 只是从函数式语言中借鉴了一些好的想法。

### 匿名函数的问题

除了 Python 独有的句法上的局限，在任何一门语言中，匿名函数都有一个严重的缺点：没有名称。

我是半开玩笑的。函数有名称，栈跟踪更易于阅读。匿名函数是一种便利的简洁方式，人们乐于使用它们，但是有时会忘乎所以，尤其是在鼓励深层嵌套匿名函数的语言和环境中，如 JavaScript 和 Node.js。匿名函数嵌套的层级太深，不利于调试和处理错误。Python 中的异步编程结构更好，或许就是因为 lambda 表达式有局限。我保证，后面会进一步讨论异步编程，但是必须等到第 18 章。顺便说一下，promise 对象、期物（future）和 deferred 对象是现代异步 API 中使用的概念。把它们与协程结合起来，能避免掉入“回调地狱”。18.5 节会说明如何不用回调来做异步编程。