注意：这章看的不详细，主要看了我需要的东西，还有很多没有看。

第十一章：函数和函数式编程

在第二章，我们引入了函数，并介绍了函数的创建和调用。这一章，我们将在前面内容的基础上，详细的讲解函数的方方面面。除了预期特性之外， Python 中的函数还支持多种调用方式以及参数类型并实现了一些函数式编程接口。最后我们将以对 Python 变量的作用域和递归函数的讨论来结束本章的学习.

目录

[1·函数的定义 1](#_Toc350597998)

[2·函数的返回值 2](#_Toc350597999)

[1)当不返回任何东西的时候 2](#_Toc350598000)

[2)返回一个值或者一个值对象的情况 2](#_Toc350598001)

[3·函数的参数 3](#_Toc350598002)

[1)形式参数 3](#_Toc350598003)

[2)可变长度的参数（包括元组、字典） 4](#_Toc350598004)

[4·函数的装饰器 5](#_Toc350598005)

[5·内建函数 6](#_Toc350598006)

[6·为引入的模块起别名 6](#_Toc350598007)

[7·偏函数 6](#_Toc350598008)

[8·global 6](#_Toc350598009)

[10·闭包 7](#_Toc350598010)

[11·生成器 8](#_Toc350598011)

## 1·函数的定义

1）def语句

2）声明与定义比较

在声明和定义有区别的语言中，往往是因为函数的定义可能和其声明放在不同的文件中。python将这两者视为一体，函数的子句由声明的标题行以及随后的定义体组成的。

3）前向引用

和其他高级语言类似,Python 也不允许在函数未声明之前,对其进行引用或者调用.

def foo():

print 'in foo()'

bar()

def bar():

print 'in bar()'

太神奇了,这段代码可以非常好的运行,不会有前向引用的问题:

>>> foo()

in foo() in bar()

这段代码是正确的因为即使（在 foo()中）对 bar()进行的调用出现在 bar()的定义之前，但 foo()本身不是在 bar()声明之前被调用的。换句话说，我们声明 foo()，然后再声明 bar()，接着调用 foo()，但是到那时，bar()已经存在了，所以调用成功。

4）内部/内嵌函数

在函数体内创建另外一个函数（对象）是完全合法的。这种函数叫做内部/内嵌函数。因为现在python 支持静态地嵌套域（在 2.1 中引入但是到 2.2 时才是标准），内部函数实际上很有用的。内部函数一个有趣的方面在于整个函数体都在外部函数的作用域（即是你可以访问一个对象的区域；稍后会有更多关于作用域的介绍）之内。如果没有任何对 bar()的外部引用，那么除了在函数体内，任何地方都不能对其进行调用，这就是在上述代码执行到最后你看到异常的原因

另外一个函数体内创建函数对象的方式是使用 lambda 语句。我们会在稍后的 11.7.1 小节进行讲述。如果内部函数的定义包含了在外部函数里定义的对象的引用（这个对象甚至可以是在外部函数之外），内部函数会变成被称为闭包（closure）的特别之物。在接下来的11.8.4 小节，我们将对闭包进行更多的学习。在下一小节中，我们将介绍装饰器，但是例子程序也包含了闭包的预览。

## 2·函数的返回值

### 1)当不返回任何东西的时候

这些函数在 c 中默认为“void"的返回类型，意思是没有值返回。 在 python 中， 对应的返回对象类型是none。

例如：

>>> def hello():

... print 'hello world'

>>>

>>> res = hello()

hello world

>>> res

>>> print res

None

>>> type(res)

<type 'None'>

### 2)返回一个值或者一个值对象的情况

另外,与其他大多数的语言一样，python 里的函数可以返回一个值或者对象。只是在返回一个容器对象的时候有点不同，看起来像是能返回多个对象。好比说，你不能拿着大量零散的商品离开百货店，但是你可以将它们放在一个购物袋里，然后带着这个袋子从商店走出去，合理合法。

例如：

def foo():

return ['xyz', 1000000, -98.6]

def bar():

return 'abc', [42, 'python'], "Guido"

foo()函数返回一个列表，bar()函数返回一个元组。由于元组语法上不需要一定带上圆括号， 所以让人真的以为可以返回多个对象。如果我们要恰当地给这个元组加上括号，bar( )的定义看起来

会是这样：

def bar():

return ('abc', [4-2j, 'python'], "Guido")

从返回值的角度来考虑， 可以通过很多方式来存储元组。接下来的 3 种保存返回值的方式是等价的

>>> aTuple = bar()

>>> x, y, z = bar()

>>> (a, b, c) = bar()

>>>

>>> aTuple

('abc', [(4-2j), 'python'], 'Guido')

>>> x, y, z

('abc', [(4-2j), 'python'], 'Guido')

>>> (a, b, c)

('abc', [(4-2j), 'python'], 'Guido')

在对 x,y,z 和 a,b,c 的赋值中，根据值返回的顺序, 每个变量会接收到与之对应的返回值。而aTuple 直接获得函数隐式返回的整个元组。回想一下,元组既可以被分解成为单独的变量，也可以直接用单一变量对其进行引用。（参见 6.18.3）

简而言之，当没有显式地返回元素或者如果返回 None 时， python 会返回一个 None.那么调用者接收的就是 python 返回的那个对象,且对象的类型仍然相同。如果函数返回多个对象，python 把他们聚集起来并以一个元组返回。是的，尽管我们声称 python 比诸如 c 那样只允许一个返回值的语言灵活的多，但是老实说，python 也遵循了相同的传统。只是让程序员误以为可以返回多个对象。

表 11.1 返回值及其类型

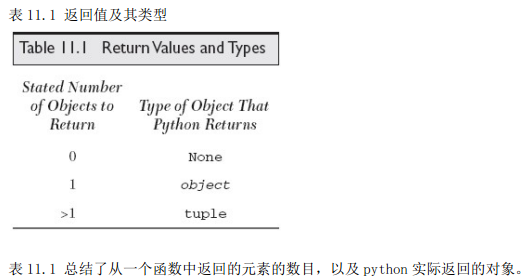


表 11.1 总结了从一个函数中返回的元素的数目，以及 python 实际返回的对象。

许多静态类型的语言主张一个函数的类型就是其返回值的类型。在 python 中， 由于 python 是动态地确定类型而且函数能返回不同类型的值，所以没有进行直接的类型关联。

## 3·函数的参数

### 1)形式参数

实际上，你也可以给出形参！这些参数包括标准的位置参数（根据参数的顺序赋值）和关键字参数（调用函数时按照函数定义时指定的形式参数名=实参句柄的方式指定实参值, 这样的参数就叫关键字参数），所以在 python 中允许的函数调用的完整语法为：

func(positional\_args, keyword\_args, \*tuple\_grp\_nonkw\_args, \*\*dict\_grp\_kw\_args) 该语法中的所有的参数都是可选的。

1. 关键字参数

关键字参数:调用函数时按照函数定义时指定的形式参数名=实参句柄的方式指定实参值, 这样的参数就叫关键字参数

再举个更实际的例子, 假设你有一个函数叫做 net\_conn(),需要两个参数 host 和 port:

def net\_conn(host, port):

net\_conn\_suite

只要按照函数声明中参数定义的顺序，输入恰当的参数，自然就可以调用这个函数:

net\_conn('kappa', 8080)

host 参数得到字符串'kappa',port 参数得到整数 8080.当然也可以不按照函数声明中的参数顺序输入，但是要输入相应的参数名,如下例:

net\_conn(port=8080, host='chino')

1. 位置参数
2. 默认参数

当参数允许"缺失“的时候，也可以使用关键字参数.这取决于函数的默认参数, 我们将在下一小节对它进行介绍。

### 2)可变长度的参数（包括元组、字典）

这种情况下，参数是变长的。

Python 同样允许程序员执行一个没有显式定义参数的函数，相应的方法是通过一个把元组（非关键字参数）或字典（关键字参数）作为参数组传递给函数。我们将在本章中讨论这两种形式。基本上，你可以将所有参数放进一个元组或者字典中，仅仅用这些装有参数的容器来调用一个函数，而不必显式地将它们放在函数调用中：

func(\*tuple\_grp\_nonkw\_args, \*\*dict\_grp\_kw\_args)

其中的 tuple\_grp\_nonkw\_args 是以元组形式体现的非关键字参数组, dict\_grp\_kw\_args 是装有关键字参数的字典。正如我们已经提到的，我们将在这章对这两者进行全面介绍，现在你只需知道，存在这样的特性允许你把变量放在元组和/或者字典里，并在没有显式地对参数进行逐个声明的情况下，调用函数。

使用\*和\*\*调用函数

\*, 将用星号指定的实参转换成一个元组, 元组的每一个元素作为一个参数进行传递. 例如a = [1, 2, 3, 4] func(\*a)则func接收到了4个参数.

\*\*, 用两个星号指定的实参必须是字典类型, 而且字典的键必须是str, 这样传递的参数实际上是以字典中的数据得到了N个关键字参数

可变参

位置可变参使用\*argument\_name定义, 函数内部得到的将是一个元组.

关键字可变参使用\*\*argument\_name定义, 函数内部得到的将是一个字典.

匿名函数与lambda: 使用lambda 参数列表: 表达式的方式可以创建一个匿名函数, lambda构建的函数可以使用可变参. 优点在于绕过了函数的栈分配, 性能得到了提升(测试了10^9数量级次数的空函数调用, 性能提升了1%左右)

## 4·函数的装饰器

和java中的注解语法一致, 以@开头, 接下来是装饰器函数的名字和可选的参数, 接下来是其修饰的函数

@decorator(decorator\_option\_arguments)

def function2bedecoratedname(func\_arguments):

4.1没有装饰器之前, 使用绑定的方式将方法注册成为类方法或静态方法:

4.1.1object.function = staticmethod(function) 或在类内部直接function = staticmethod(function)

4.1.2object.function = classmethod(function) 或在类内部直接function = classmethod(function)

4.2使用装饰器的语法

4.2.1静态方法

@staticmethod

def function():

4.2.2类方法

@classmethod

def function():

4.2.3带参数的装饰器

@deco1(deco\_arg)

@deco2

def func():

就相当于做了

func = deco1(deco\_arg)(deco2(func))

4.3装饰器其实就是函数, 装饰器接受函数对象, 并对函数对象进行包装处理, 就相当于java中的aop, 可以使用装饰器做一些日志, 安全性检查等通用型的功能

def mydecorator(func):

def wrappedFunc():

print ‘log is recorded’

return func()

return wrappedFunc

@mydecorator

def f():

print ‘Hello decorator’

print f()

4.4装饰器的注意点:

必须定义一个内部函数名字是wrappedFunc

装饰器函数返回的是内部函数wrappedFunc的引用(或别名)

wrappedFunc()函数包装器所接受的参数就是被装饰函数的参数列表

4.5装饰器可以有多个, 每行一个, 按照顺序写就可以了, 其实装饰器就相当于调用了对应的装饰器函数, 对于多个装饰器, 就类似数学上的函数嵌套. (g · f)(x) = g(f(x))

## 5·内建函数

1）apply(func[, nkw][, kw]): 用可选的参数来调用func, nkw是非关键字参数, kw是关键字参数, 返回值就是func调用之后的返回结果

* 1. filter(func, sequence): 调用一个布尔函数func迭代遍历序列中的每个元素, 返回一个func返回True的元素的序列.
  2. map(func, seq1[, seq2…]): 将函数func作用于给定序列中的每个元素, 并用一个列表来提供返回值, 如果func为None, func就表现为身份函数(id), 返回一个含有每个元素中元素集合的n个元组的列表
     1. 提供了多少个seq参数, func就接受多少个参数.
     2. 如果func存在, 将各个列表的值按顺序传入到func中, 返回的结果被组成一个新的列表作为map的返回.
     3. 如果func是None, 返回各个列表的类似压缩得到的元组的列表, 这里的压缩不同于zip, 是类似sql中的外联接的模式.
  3. reduce(func, seq[, init]): 将二元函数作用于seq序列的元素, 每次携带一对(先前的结果以及下一个元素), 连续的将现有的结果和下一个值作用在获得的随后的结果上, 最后减少序列为一个单一的返回值, 乳沟初始值init给定, 第一个比较会是init和第一个元素而不是序列的前两个元素. 有一点类似递归调用

## 6·为引入的模块起别名

引入一个模块的时候可以起一个别名: from module\_name import attribute\_name as alias

## 7·偏函数

使用functools模块中的partial()函数创建偏函数, 第一个参数是基函数的引用, 后面跟随基函数的调用中要使用的一些默认值…这里还可以使用关键字参数调用的方式:

toDecimal = partial(int, base = 2)

调用的时候, 只需要指定在partial中没有指定的参数, 比如: toDecimal(‘101010’)

由于在partial中指定的这些固定的参数是在参数列表的最左边, 所以, 如果不使用关键字参数, 可能会导致参数顺序错误.

### 8·global

在函数体内使用global关键字可以使全局变量的变量名在函数体内获得全局变量的引用, 在函数体内对全局变量的改变会影响全局变量, 例如:

a = 100

def f():

global a

a += 300

print a

得到的输出会是400, 并且, 全局变量a也被修改为400

注意: 使用global语法, 必须是在该全局变量名第一次在函数体内被使用之前

9·嵌套函数的内部变量作用域为自己的函数体(包含自己的内部子孙函数). 在函数体内可以使用的变量域为自己的所有父辈中的变量.

### 10·闭包

将内部函数自己的代码和作用于以外的外部函数的作用结合起来, 在一个独立的作用域中, 多用于安装计算, 隐藏状态, 函数对象和作用于中随意切换, 回调.

* 1. 自由变量: 定义在外部函数内(非全局变量), 但是由内部函数引用或者使用的变量.
  2. 闭包: 如果在一个函数内, 对在外部作用域的变量(自由变量)进行引用, 那么这个内部函数就被认为是闭包(closure)
  3. 闭包完成的功能很像类.
  4. 闭包将内部函数自己的代码和作用域以外的外部函数作用结合起来…闭包的词法变量不属于全局名字空间域也不属于局部的, 而是属于其他的名字空间, 带着”流浪”的作用域. 但是, 这又不同于对象, 对象的变量存活在对象的名字空间, 而闭包变量存活在一个函数的名字空间和作用域.
  5. 闭包的语法:
     1. 定义

def out\_function(arg1, arg2, arg3……): //定义一个外部函数

args = [arg1, arg2] //将要作为自由变量的参数放到一个可变对象中.

def inner\_function(self\_arg): //定义一个内部函数, 这里可以接受自己的参数

args[0] += args[1] + self\_arg //改变自由变量, 上面将自由变量放到一个可变对象中就是为了能够保存这种改变.

return args[0] //返回修改后的自由变量值

return inner\_function //返回内部函数

* + 1. 调用

closure1 = out\_function(1, 2, 3) //得到一个闭包(调用外部函数, 返回了内部函数的引用, 实际上就形成了一个闭包, 这个闭包的作用域内持有了外部函数内定义的args自由变量和内部函数本身)

closure2 = out\_function(4, 5, 6) //得到一个闭包

closure1(100) //调用闭包局部的实参是100

closure2(200) //调用闭包返回的

* 1. 调用闭包返回的结果很类似调用对象的方法, 但是, 闭包是一种能够有自己的独立作用域的方法, 而对象则是类实例.
  2. 得到一个闭包的实例(实际上就是一个方法), 查看它的func\_closure属性, 可以查看该闭包含有的自由变量.
  3. 一个闭包和装饰器结合的绝妙的例子见python-15-function下的closureAndDecorator4Log.py
  4. lambda定义的函数作用域, 生命周期和普通函数是一致的, lambda也可以不指定参数, 也可以使用全局或外部变量.
  5. 递归: 一个超级短的递归阶乘函数: f = lambda n: n == 1 and 1 or (n == 2 and 2 or n \* a(n - 1))

## 11·生成器

生成器使用生成器表达式创建, 和列表解析表达式语法相同, 使用圆括号包裹. 也是使用next方法, 完成遍历之后抛出StopIteration异常.

* 1. 定义: 挂起返回出中间值并多次继续的协同程序成为生成器.
  2. 生成器语法:

def mygenerator(arr):

for i in arr:

yield i

* 1. 生成器的创建使用了yield语法, 在调用next方法的时候, 返回下一个yield指示的值, 并让生成器在这个位置停留.
  2. 当然, yield也可以指示做某一个动作之后停止. 甚至还允许yield指示的动作返回的是一个None
  3. 生成器适用于较大的数据场合.
  4. 生成器的创建函数中不能有return
  5. 生成器还可以接收外部传送回来的数据进行交互. 在外部调用生成器的send方法可以向生成器发送数据, send方法接受一个参数, 这个参数的值就是yield argument\_name词法的返回值. send方法返回的是经过处理之后到下一次yield的值, 其实是和调用next()方法的返回值是一样的, 不过这次是经过了send进来的数据的处理方式.

def counter(start):

count = start

while True:

input = yield count

if input:

count += input

else:

count += 1