第十一章 函数和函数式编程

什么是函数

函数是对程序逻辑进行结构化或过程化的一种编程方法。
函数是一个”黑盒”能不带入任何的参数，经过一定的处理后返回一个零或者非零值。
过程是简单，特殊的函数，没有返回值的函数。
python 的过程就是函数，因为解释器会隐式地返回默认值 `None` (类似于C中的 `void`),即Python函数会默认返回None

元组的语法不需要一定带上圆括号。Python的函数返回多个对象时，Python会把他们聚集成一个元组返回。

Return Values and Types

Stated Number of Objects to Return	Type of Object That Python Returns

等于0	None
等于1	Object
大于1	tuple

调用函数

关键字参数

```
def net_conn(host, port):
    pass

# 调用
net_conn('kappa',8080)
# 关键字参数调用-如果参数有默认值,可以不想该参数传值
net_conn(port=8080,host='chino')
```

函数调用的完整语法

```
func(positional_args, keyword_args,*tuple_grp_nonkw_args, **dict_grp_kw_args)

#positional_args 位置参数
#keyword_args 关键字参数
#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
```

创建函数

```
def function_name(arguments):
    "function_documentation_string"
    function_body_suite
```

Python中将函数的声明和定义视为一体。

函数的属性

函数属性是 python 另外一个使用了句点属性标识并拥有名字空间的领域。
`__doc__`

内部/内嵌函数

一个简单定义内部函数的方式是在方法体内定义一个函数。
一个函数体内创建函数对象的方式是使用 lambda 语句。

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。

函数(与方法)的装饰器

装饰器背后的主要动机源自 python 面向对象编程。装饰器是在函数调用之上的修饰。这些修饰仅是当声明一个函数或者方法的时候，才会应用的额外调用。

```
# 定义一个装饰器
decorator(dec_opt_args)
def func2Bdecorated(func_opt_args):
    pass
```

多个装饰器可以组合使用

```
@deco2
@deco1
def func(arg1, arg2, ...):
    pass

def func(arg1, arg2, ...):
    pass
func = deco2(deco1(func))
```

有参数和无参数的装饰器

```python
@deco
def foo():
    pass

foo = deco(foo)

# 带参数的装饰器
@decomaker(deco_args)
def foo():
    pass

@deco1(deco_arg)
@deco2
def func():
    pass

# This is equivalent to
func = deco1(deco_arg)(deco2(func)) # 经过装饰器修改后返回修改后的函数对象，将其重新赋值给原来的函数
```

装饰器实际就是函数。

装饰器的例子

```
from time import ctime, sleep

def tsfunc(func):
    def wrappedFunc():
        print '[%s] %s() called' %(ctime(), func.__name__)
        return func()
    return wrappedFunc

@tsfunc
def foo():
    pass

foo()
sleep(4)

for i in range(2):
    sleep(1)
    foo()

# output
[Wed Mar 30 17:33:46 2016] foo() called
[Wed Mar 30 17:33:51 2016] foo() called
[Wed Mar 30 17:33:52 2016] foo() called
```

PEP 318 中有更多关于装饰器的内容

传递函数

函数有一个独一无二的特征使它同其他对象区分开来，那就是函数是可调用的。

所有的对象都是通过引用来传递的，包括函数。与C/C++一样

```
def foo():
    print 'in foo()'
bar = foo #将函数引用foo赋值给另外一个变量bar
bar() # 函数调用


# 已函数作为其他函数的参数
def bar(argfunc):
    argfunc()
bar(foo)

#output
in foo()
```

形式参数

函数一旦开始执行，就能访问这个函数名

位置参数

位置参数必须以在被调用函数中定义的准确顺序来传递。

关键字参数已经被证明能给不按顺序的位置参数提供参数，结合默认参数，它们同样也能被用于跳过缺失参数

从互联网上抓取一个 Web 页面并暂时储存到一个本地文件的例子，
能用来测试 web 站点页面的完整性或者能监测一个服务器的负载（通过测量可链接性或者下载速度

```
def firstNonBlank(lines):
    for eachLine in lines:
        if not eachLine.strip():
            continue
        else:
            return eachLine

def firstLast(webpage):
    f = open(webpage)
    lines = f.readlines() # 读取所有的行
    f.close()
    print firstNonBlank(lines)
    lines.reverse()
    print firstNonBlank(lines)
    #print lines

def download(url='htpp://www',process=firstLast):
    try:
        retval = urlretrieve(url)[0]
    except IOError:
        retval = None
    if retval: # do some processing
        process(retval)

#output
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/loose.dt
d">

</html>
```

可变长度的参数

非关键字可变长参数(元组)

可变长的参数元组必须在位置和默认参数之后,带元组的函数声明语法如下
位置参数必须放在关键字参数之前

```
def function_name([formal_args,] *vargs_tuple):
    "function_documentation_string"
    function_body_suite
# 元组参数保存了所有传递给函数的"额外"的参数(匹配了所有位置和具名参数后剩余的)
# 如果没有给出额外的参数，元组为空

# 非关键字可变长参数（元组）
def tupleVarArgs(arg1, arg2='defalutB', *theRest):
    'display regular args and non-keyword variable args'
    print 'formal arg 1:', arg1
    print 'formal arg 2:', arg2
    for eachXtrArg in theRest:
        print 'another arg:',eachXtrArg

tupleVarArgs('abc')
tupleVarArgs(23, 4.56)
tupleVarArgs('abc', 123, 'xyz', 456.789)

# output
formal arg 1: abc
formal arg 2: defalutB
formal arg 1: 23
formal arg 2: 4.56
formal arg 1: abc
formal arg 2: 123
another arg: xyz
another arg: 456.789
```

关键字变量参数(Dictionary)

有不定数目的或者额外集合的关键字的情况中， 参数被放入一个字典中，字典中键为参数名，值为相应的参值。

关键字变量参数应该为函数定义的最后一个参数，带**

语法如下

```
def function_name([formal_args,][*vargst,] **vargsd):
    function_documentation_string function_body_suite
# 为了区分关键字参数和非关键字非正式参数，使用了双星号（**）
# **是被重载了的以便不与幂运算发生混淆。
```

```
def dictVarArgs(arg1, arg2='defaultB', **theRest):
    'display 2 regular args and keyword variable args'
    print 'formal arg1:', arg1
    print 'formal arg2:', arg2
    for eachXtrArg in theRest.keys():
        print 'Xtra arg %s: %s' % (eachXtrArg, str(theRest[eachXtrArg]))
```

函数式编程

Python提供的以 4 种内建函数和 lambda 表达式的形式出现

一个完整的 lambda“语句”代表了一个表达式，这个表达式的定义体必须和声明放在同一行。
下匿名函数的语法:

```
lambda [arg1[, arg2, ... argN]]: expression
```

简单地用 lambda 创建了一个函数（对象），但是既没有在任何地方保存它,也没有调用它。

```
lambda :True
```

这个函数对象的引用计数在函数创建时被设置为 True，但是因为没有引用保存下来，计数又回到零，然后被垃圾回收掉。

lambda语句的目的是由于性能的原因，在调用时绕过函数的栈分配。

内建函数 apply()、filter()、map()、reduce()

- `filter()`

```
def filter(bool_func, seq):
    filtered_seq = []
    for eachItem in seq:
        if bool_func(eahcItem):
        filtered_seq.append(eachItem)
    return filtered_seq
```

- `map()`

```
def map(func, seq):
    mapped_seq = []
    for eachItem in seq:
        mapped_seq.append(func(eachItem))
    return mapped_seq

map(lambda x: x**2, range(6)
[0, 1, 4, 9, 16, 25]

# map被列表解析取代
[x+2 for x in range(6)]
[2, 3, 4, 5, 6, 7]

map(lambda x, y: (x+y, x-y), [1,3,5], [2,4,6])
[(3, -1), (7, -1), (11, -1)]
```

形式更一般的 map（ ）能以多个序列作为其输入。如果是这种情况， 那么 map()会并行地迭代每个序列。

在第一次调用时， map()会将每个序列的第一个元素捆绑到一个元组中， 将 func 函数作用到map（ ）上， 当 map（ ）已经完成执行的时候，并**将元组的结果返回到 mapped_seq 映射的，最终以整体返回的序列上。**

```
map(None, [1,3,5], [2,4,6])
[(1, 2), (3, 4), (5, 6)]

zip([1,3,5], [2,4,6])
[(1, 2), (3, 4), (5, 6)]
```

- `reduce()`

```
print 'the total is:', reduce((lambda x,y: x+y), range(5))

# ((((0 + 1) + 2) + 3) + 4) => 10
```

偏函数应用(Partial Function Application=PFA)

`functools` 模块
`functional`模块中的`partial()`函数来创建 PFA

PEP 309关于 `functools` 模块的文档中阅读到更多关于 PFA 的资料。

变量的作用域

global

```
global var1[, var2[, ... varN]]
```

闭包

If references are made from inside an inner function to an object defined in any outer scope (but not in the global scope), the inner function then is known as a `closure`.

The variables defined in the outer function but used or referred to by the inner function are called `free variables`.

A closure combines an inner function's own code and scope along with the scope of an outer function. Closure lexical variables do **not belong to the global namespace scope or the local one**—they belong to someone else's namespace and carry an "on the road" kind of scope. (Note that they differ from objects in that those variables live in an object's namespace while closure variables live in a function's namespace and scope.)

闭包将内部函数自己的代码和作用域以及外部函数的作用结合起来。 闭包的词法变量不属于全局名字空间域或者局部的–而属于其他的名字空间，着"流浪"的作用域。

Callbacks are just functions. Closures are functions, too, but they carry some additional scope with them. They are just functions with an extra feature … another scope.

如果在一个内部函数里，对在外部作用域(但不是在全局作用域）的变量进行引用，那么内部函数就被认为是`closure`,引用的变量成为**自由变量。**

定义在外部函数内的但由内部函数引用或者使用的变量被称为**自由变量。**

闭包将内部函数自己的代码和作用域以及外部函数的作用结合起来。

变量的作用域和名字空间

任何时候，总有一个或者两个活动的作用域。

要么在只能访问全局作用域的模块的最高级，要么在一个我们能访问函数局部作用域和全局作用域的函数体内执行。

生成器

PEP255,PEP342
`yield` 语句的功能， 返回一个值给调用者并暂停执行

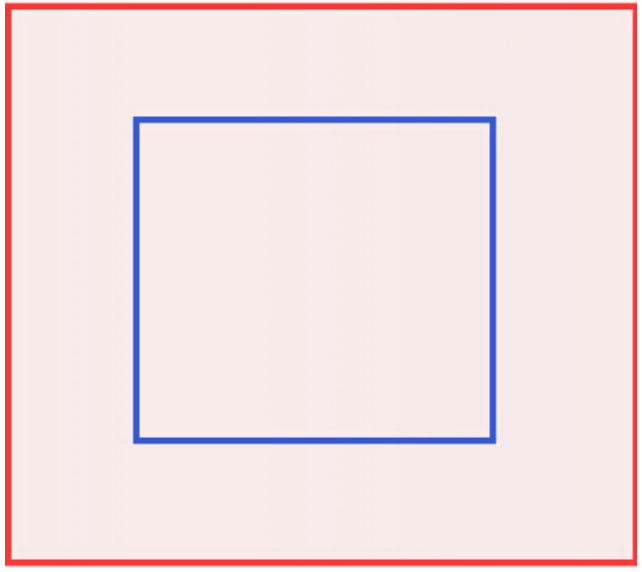
当生成器的 `next()` 方法被调用的时候，它会准确地从离开地方继续（当它返回[一个值以及]控制给调用者时）

总结

232,309,318262

变量的作用域

LEGB:L>E>G>B
L=Local函数内部作用域
E=Encloure函数与外部函数(但不是全局作用域)之间–Enclosing函数内部与内嵌函数之前
G=Global全局作用域
B=Build-in内置作用域



红框之外成为`global`作用域
红框为`outer function`,蓝框为`inner function`，位于篮框内的变量为`local`,而位于红框和篮框之前的作用域就是`Enclosing`作用域

关于闭包的例子

```
def my_sum(*args):
    return sum(args)

def my_average(*args):
    return sum(args) / len(args)

# 将相同的逻辑放在一个function中
def dec(func):
    def in_dec(*args):
        print args
        if len(args) == 0:
            return 0
        for val in args:
            if not isinstance(val, int):
                return 0
        return func(*args)  # 调用func,在内部函数in_dec中调用func,内部函数会将func放入enclosure属性中

    return in_dec

#dec return in_dec -> my_sum
my_sum = dec(my_sum)
my_average = dec(my_average)

print my_sum(1, 2, 3, 4, 5, 6) #这时my_sum已经不是一开始定义的my_sum了,此时的my_sum指向了in_dec(即my_sum为i
n_dec的引用)
print my_average(1, 2, 3, 4, 5, 6) # 与my_sum一样
```

关于装饰器的例子(装饰器其实就是对闭包的使用)

```python
def my_sum(*args):
    return sum(args)

def my_average(*args):
    return sum(args) / len(args)

def dec(func):
    print 'call dec'
    def in_dec(*args): # my_sum
        print 'in_dec arsg: ',args
        if len(args) == 0:
            return 0
        for val in args:
            if not isinstance(val, int):
                return 0
        return func(*args)  # 调用待装饰的函数func,在内部函数in_dec中调用func,内部函数会将func放入enclosure属性中
    print 'return in_dec'
    return in_dec


print '*' * 40

# 调用dec装饰器,此时会返回一个in_dec,并且in_dec赋值给my_sum
# my_sum = dec(my_sum)这是装饰器做的主要一件事
# 就是将待装饰的函数my_sum功能丰富，内部还继续调用待装饰的函数muy_sum

# @dec 相当于my_sum = dec(my_sum)
@dec
```

```
def my_sum(*args): # my_sum = in_dec
    return sum(args)

print my_sum(1, 2, 3, 4, 5, 6, 7, 8)

@dec
def my_average(*args):
    return sum(args) / len(args)

print my_average(1, 2, 3, 4, 5, 6, 7, 8)
```