**Python进阶：**

**[函数：](#_Toc31027)** [1](#_Toc31027)

**[高阶函数：](#_Toc12441)** [3](#_Toc12441)

[map（函数名，list）： 3](#_Toc10788)

[reduce（函数名，list，[计数初始值，默认是0]）： 3](#_Toc640)

[filter（函数名，list）： 3](#_Toc32227)

[sorted（L，比较函数）： 3](#_Toc17416)

**[装饰器：](#_Toc14237)** [4](#_Toc14237)

**[闭包：](#_Toc32261)** [5](#_Toc32261)

**[匿名函数：](#_Toc11721)** [5](#_Toc11721)

**[模块：](#_Toc8762)** [6](#_Toc8762)

**[包：](#_Toc21404)** [6](#_Toc21404)

**函数：**

定义：函数就是最基本的一种代码抽象的方式

编写：定义一个函数要使用 **def 语句**，依次写出函数名、括号、括号中的参数和冒号:，然后，在缩进块中 编写函数体，函数的返回值用 return 语句返回。

如：

def my\_abs(x):

if x >= 0:

return x

else:

return -x

调用：要调用一个函数，需要知道函数的**名称和参数**

**返回值：**

**返回多个值**：Python的函数返回多值其实就是返回一个tuple，而多个变量可以同时接收一个tuple，按位置赋给对应的值

**返回函数：**Python的函数不但可以返回int、str、list、dict等数据类型，还可以返回函数！

如：

def calc\_sum(lst):

def lazy\_sum():

return sum(lst)

return lazy\_sum

# 调用calc\_sum()并没有计算出结果，而是返回函数:

>>> f = calc\_sum([1, 2, 3, 4])

>>> f

<function lazy\_sum at 0x1037bfaa0>

# 对返回的函数进行调用时，才计算出结果:

>>> f()

10

由于可以返回函数，我们在后续代码里就可以决定到底要不要调用该函数。

**参数：**

**默认参数**：给参数设定了初始值，由于函数的参数按从左到右的顺序匹配，所以默认参数只能定义在必需参数的后面 如：def my\_abs(x = 1):

**可变参数：让一个函数能接受任意个参数**

收集参数：

在定义时，可变参数的前面有一个\*，Python解释器会收集其余的位置参数组装成一个**tuple**传递给可变参数

在定义时，可变参数的前面有两个\*\*，Python解释器会收集其余的含有默认值的参数组装成一个**dict**传递给可变参数

收集参数逆过程：

在调用函数时，可变参数的前面有一个\*，将类型为元组的参数分配给定义的函数的参数列表

在调用函数时，可变参数的前面有两个\*\*，将类型为字典的参数分配给定义的函数的参数列表

例子：

def story(\*\*kwds):

return ‘Once upon a time, there was a ’\

‘%(jobs)s called %(name)s.’%kwds

print story(job = ‘king’,name = ‘Tim’)

>>>Once upon a time, there was a king called Tim

params = {‘job’：‘king’，‘name’：‘Tim’}

print story(params)

>>>错误

print story(\*\*params)

>>>Once upon a time, there was a king called Tim

**高阶函数：**

**定义**：将函数作为参数，传递到其他函数

如：

def add(x, y, f):

return f(x) + f(y)

add(-5, 9, abs)

**Python中内置高阶函数：**

map（函数名，list）：

它接收一个函数 f 和一个 list，并通过把函数 f 依次作用在 list 的每个元素上，得到一个新的 list 并返回。

如：

def f(x):

return x\*x

print map(f, [1, 2, 3, 4, 5, 6, 7, 8, 9])

输出结果：

[1, 4, 9, 10, 25, 36, 49, 64, 81]

reduce（函数名，list，[计数初始值，默认是0]）：

它接收的一个函数 f和一个list，传入的函数 f 必须接收两个参数，

reduce()对list的每个元素反复调用函数f，并返回最终结果值。

如：

def f(x, y):

return x + y

调用 reduce(f, [1, 3, 5, 7])时，reduce函数将做如下计算：

先计算头两个元素：f(1, 3)，结果为4；

再把结果和第3个元素计算：f(4, 5)，结果为9；

再把结果和第4个元素计算：f(9, 7)，结果为16；

由于没有更多的元素了，计算结束，返回结果16。

filter（函数名，list）：

接收一个函数 f 和一个list，这个函数 f 的对每个元素进行判断，返回 True或 False，filter()根据判断结果自动过滤掉不符合条件的元素，返回由符合条件元素组成的新list。

如：

def is\_odd(x):

return x % 2 == 1

然后，利用filter()过滤掉偶数：

filter(is\_odd, [1, 4, 6, 7, 9, 12, 17])

结果：[1, 7, 9, 17]

sorted（L，比较函数）：

接收一个比较函数来实现自定义排序，

比较函数的定义是，传入两个待比较的元素 x, y，

如果 x 应该排在 y 的前面，返回 -1，

如果 x 应该排在 y 的后面，返回 1。

如果 x 和 y 相等，返回 0。

**装饰器：**

**定义：**本质上就是一个高阶函数，它接收一个函数作为参数，然后，返回一个新函数。

**作用**：在代码运行期间动态增加功能的方式

**使用**： decorator 用Python提供的 **@**语法，这样可以避免手动编写**f = decorate(f)**这样的代码。

不带参数的：

一个@log的定义：

def log(f):

def fn(x):

print 'call ' + f.\_\_name\_\_ + '()...'

return f(x)

return fn

//阶乘函数

@log

def factorial(n):

return reduce(lambda x,y: x\*y, range(1, n+1))

print factorial(10)

结果：

call factorial()...

3628800

要让 @log 自适应任何参数定义的函数，可以利用Python的**\*args** 和 **\*\*kw**，保证任意个数的参数总是能正常调用：

不带参数的，但能使装饰圈不受函数参数的干扰：

def log(f):

def fn(\*args, \*\*kw):

print 'call ' + f.\_\_name\_\_ + '()...'

return f(\*args, \*\*kw)

return fn

带参数的：

带参数的log函数首先返回一个decorator函数，再让这个decorator函数接收my\_func并返回新函数

如：@log('DEBUG')

相当于my\_func = log('DEBUG')(my\_func)

相当于log\_decorator = log('DEBUG') my\_func = log\_decorator(my\_func)

def log(prefix):

def log\_decorator(f):

def wrapper(\*args, \*\*kw):

print '[%s] %s()...' % (prefix, f.\_\_name\_\_)

return f(\*args, \*\*kw)

return wrapper

return log\_decorator

@log('DEBUG')

def test():

pass

print test()

执行结果：

[DEBUG] test()...

None

函数：

functools:把原函数的一些属性复制到新函数中

使用：@functools.wraps(f)

Functools.partial：帮助我们创建一个偏函数的，把一个参数多的函数变成一个参数少的新函数

如：

import functools

>>> int2 = functools.partial(int, base=2)

>>> int2('1000000')

64

**闭包：**

**定义：**内层函数引用了外层函数的变量（参数也算变量），然后返回内层函数的情况，称为闭包（Closure） 。

如：

def calc\_sum(lst):

def lazy\_sum():

return sum(lst)

return lazy\_sum

要正确使用闭包，就要确保引用的局部变量在函数返回后不能变。因此，返回函数不要引用任何循环变量，或者后续会发生变化的变量。

（因为它是函数在被调用时，才执行这个函数，引用的是调用它是变量的当前值）

**匿名函数：**

**定义：**不必定义函数名，直接创建一个函数对象

（匿名函数有个限制，就是只能有一个表达式，不写return，返回值就是该表达式的结果。）

**编写：**关键字**lambda**表示匿名函数，冒号前面的 x 表示函数参数。

如：map(lambda x: x \* x, [1, 2, 3, 4, 5, 6, 7, 8, 9])

[1, 4, 9, 16, 25, 36, 49, 64, 81]

**模块：**

**定义**：单独的一个 .py 文件

模块的导入：**import**

只导入模块中的函数：**from** 模块名 **import** 函数名 [**as** 别名]

安装第三方模块：

①easy\_install

②pip（已经内置到Python2.7）

在命令行中运行pip install 第三方模块名

**包：**

**定义**：相当于一个文件夹，可以有多级，作用类似于namespace，解决了模块重名的问题

**包和目录的区别**：包的每一层目录都要有\_\_init\_\_.py文件