python入门四之函数

基本函数使用

1. 函数调用

要调用一个函数，需要知道函数的名称和参数，比如求绝对值的函数 abs，它接收一个参数。

可以直接从Python的官方网站查看文档：

http://docs.python.org/2/library/functions.html#abs

也可以在交互式命令行通过 help(abs) 查看abs函数的帮助信息。

简单介绍几种函数的调用：

abs（）:绝对值函数

>>> abs(100)

100

>>> abs(-20)

20

cmp（）：比较函数

而比较函数 cmp(x, y) 就需要两个参数，如果 x<y，返回 -1，如果x==y，返回 0，如果 x>y，返回 1：

>>> cmp(1, 2)

-1

>>> cmp(2, 1)

1

>>> cmp(3, 3)

0

int（）：转换为int类型

>>> int('123')

123

>>> int(12.34)

12

str（）：转换为string类型

>>> str(123)

'123'

>>> str(1.23)

'1.23'

编写函数

在Python中，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。

示例：

def my\_abs(x):

if x >= 0:

return x

else:

return -x

请注意，函数体内部的语句在执行时，一旦执行到return时，函数就执行完毕，并将结果返回。因此，函数内部通过条件判断和循环可以实现非常复杂的逻辑。

如果没有return语句，函数执行完毕后也会返回结果，只是结果为 None。

return None可以简写为return。

返回多值？

# math包提供了sin()和cos()函数，我们先用import引用它：

import math

def move(x, y, step, angle):

nx = x + step \* math.cos(angle)

ny = y - step \* math.sin(angle)

return nx, ny

这样我们就可以同时获得返回值：

>>> x, y = move(100, 100, 60, math.pi / 6)

>>> print x, y

151.961524227 70.0

但其实这只是一种假象，Python函数返回的仍然是单一值：

>>> r = move(100, 100, 60, math.pi / 6)

>>> print r

(151.96152422706632, 70.0)

用print打印返回结果，原来返回值是一个tuple！

但是，在语法上，返回一个tuple可以省略括号，而多个变量可以同时接收一个tuple，按位置赋给对应的值，所以，Python的函数返回多值其实就是返回一个tuple，但写起来更方便。

递归

使用递归函数需要注意防止栈溢出。在计算机中，函数调用是通过栈（stack）这种数据结构实现的，每当进入一个函数调用，栈就会加一层栈帧，每当函数返回，栈就会减一层栈帧。由于栈的大小不是无限的，所以，递归调用的次数过多，会导致栈溢出。可以试试计算 fact(10000)。

定义默认参数

函数的默认参数的作用是简化调用，你只需要把必须的参数传进去。但是在需要的时候，又可以传入额外的参数来覆盖默认参数值。

假设计算平方的次数最多，我们就可以把 n 的默认值设定为 2：

def power(x, n=2):

s = 1

while n > 0:

n = n - 1

s = s \* x

return s

这样一来，计算平方就不需要传入两个参数了：

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：

# OK:

def fn1(a, b=1, c=2):

pass

# Error:

def fn2(a=1, b):

pass

定义可变参数

如果想让一个函数能接受任意个参数，我们就可以定义一个可变参数：

def fn(\*args):

print args

可变参数的名字前面有个\*号，我们可以传入0个、1个或多个参数给可变参数：

>>> fn()

()

>>> fn('a')

('a',)

>>> fn('a', 'b')

('a', 'b')

>>> fn('a', 'b', 'c')

('a', 'b', 'c')

可变参数也不是很神秘，Python解释器会把传入的一组参数组装成一个tuple传递给可变参数，因此，在函数内部，直接把变量 args看成一个 tuple 就好了。

定义可变参数的目的也是为了简化调用。假设我们要计算任意个数的平均值，就可以定义一个可变参数：

def average(\*args):

...

这样，在调用的时候，可以这样写：

>>> average()

0

>>> average(1, 2)

1.5

>>> average(1, 2, 2, 3, 4)

2.4

高级函数

def add(x, y, f):

return f(x) + f(y)

如果传入abs作为参数f的值：

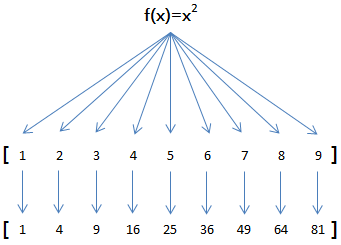
add(-5, 9, abs)

根据函数的定义，函数执行的代码实际上是：

abs(-5) + abs(9)

map（）函数

map()是 Python 内置的高阶函数，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。



因此，我们只需要传入函数f(x)=x\*x，就可以利用map()函数完成这个计算：

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]

注意：map()函数不改变原有的 list，而是返回一个新的 list。

利用map()函数，可以把一个 list 转换为另一个 list，只需要传入转换函数。

由于list包含的元素可以是任何类型，因此，map() 不仅仅可以处理只包含数值的 list，事实上它可以处理包含任意类型的 list，只要传入的函数f可以处理这种数据类型。

reduce（）函数

reduce()函数也是Python内置的一个高阶函数。reduce()函数接收的参数和 map()类似，一个函数 f，一个list，但行为和 map()不同，reduce()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。

例如，编写一个f函数，接收x和y，返回x和y的和：

def f(x, y):

return x + y

调用reduce(f, [1, 3, 5, 7, 9])时，reduce函数将做如下计算过程：

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；

再把结果和第5个元素计算：f(16, 9)，结果为25；

由于没有更多的元素了，计算结束，返回结果25。

上述计算实际上是对 list 的所有元素求和。虽然Python内置了求和函数sum()，但是，利用reduce()求和也很简单。

reduce()还可以接收第3个可选参数，作为计算的初始值。如果把初始值设为100，计算：

reduce(f, [1, 3, 5, 7, 9], 100)

结果将变为125，因为第一轮计算是：

计算初始值和第一个元素：f(100, 1)，结果为101。

filer（）函数

filter()函数是Python内置的另一个有用的高阶函数，filter()函数接收一个函数 f和一个list，这个函数f的作用是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。

例如，要从一个list [1, 4, 6, 7, 9, 12, 17]中删除偶数，保留奇数，首先，要编写一个判断奇数的函数：

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]

利用filter()，可以完成很多有用的功能，例如，删除 None 或者空字符串：

def is\_not\_empty(s):

return s and len(s.strip()) > 0

filter(is\_not\_empty, ['test', None, '', 'str', ' ', 'END'])

结果：['test', 'str', 'END']

注意: s.strip(rm) 删除 s 字符串中开头、结尾处的 rm 序列的字符。

当rm为空时，默认删除空白符（包括'\n', '\r', '\t', ' ')，如下：

a = ' 123'

a.strip()

结果： '123'

a='\t\t123\r\n'

a.strip()

结果：'123'

自定义排序函数

Python内置的sorted()函数可对list进行排序：

>>>sorted([36, 5, 12, 9, 21])  
[5, 9, 12, 21, 36]

但sorted()也是一个高阶函数，它可以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。（返回1则交换）

因此，如果我们要实现倒序排序，只需要编写一个reversed\_cmp函数：

def reversed\_cmp(x, y):

if x > y:

return -1

if x < y:

return 1

return 0

这样，调用 sorted() 并传入 reversed\_cmp 就可以实现倒序排序：

>>> sorted([36, 5, 12, 9, 21], reversed\_cmp)

[36, 21, 12, 9, 5]

sorted()也可以对字符串进行排序，字符串默认按照ASCII大小来比较：

>>> sorted(['bob', 'about', 'Zoo', 'Credit'])

['Credit', 'Zoo', 'about', 'bob']

'Zoo'排在'about'之前是因为'Z'的ASCII码比'a'小。

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！

例如，定义一个函数 f()，我们让它返回一个函数 g，可以这样写：

def f():

print 'call f()...'

# 定义函数g:

def g():

print 'call g()...'

# 返回函数g:

return g

仔细观察上面的函数定义，我们在函数 f 内部又定义了一个函数 g。由于函数 g 也是一个对象，函数名 g 就是指向函数 g 的变量，所以，最外层函数 f 可以返回变量 g，也就是函数 g 本身。

调用函数 f，我们会得到 f 返回的一个函数：

>>> x = f() # 调用f()

call f()...

>>> x # 变量x是f()返回的函数：

<function g at 0x1037bf320>

>>> x() # x指向函数，因此可以调用

call g()... # 调用x()就是执行g()函数定义的代码

请注意区分返回函数和返回值：

def myabs():

return abs # 返回函数

def myabs2(x):

return abs(x) # 返回函数调用的结果，返回值是一个数值

返回函数可以把一些计算延迟执行。例如，如果定义一个普通的求和函数：

def calc\_sum(lst):

return sum(lst)

调用calc\_sum()函数时，将立刻计算并得到结果：

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

10

但是，如果返回一个函数，就可以“延迟计算”：

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。

闭包

在函数内部定义的函数和外部定义的函数是一样的，只是他们无法被外部访问：

def g():

print 'g()...'

def f():

print 'f()...'

return g

将 g 的定义移入函数 f 内部，防止其他代码调用 g：

def f():

print 'f()...'

def g():

print 'g()...'

return g

但是，考察上一小节定义的 calc\_sum 函数：

def calc\_sum(lst):

def lazy\_sum():

return sum(lst)

return lazy\_sum

注意:发现没法把lazy\_sum移到calc\_sum的外部，因为它引用了calc\_sum的参数**lst**。

像这种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）。

闭包的特点是返回的函数还引用了外层函数的局部变量，所以，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。举例如下：

# 希望一次返回3个函数，分别计算1x1,2x2,3x3:

def count():

fs = []

for i in range(1, 4):

def f():

return i\*i

fs.append(f)

return fs

f1, f2, f3 = count()

你可能认为调用f1()，f2()和f3()结果应该是1，4，9，但实际结果全部都是9（请自己动手验证）。

原因就是当count()函数返回了3个函数时，这3个函数所引用的变量 i的值已经变成了3。由于f1、f2、f3并没有被调用，所以，此时他们并未计算 i\*i，当 f1 被调用时：

>>> f1()

9 # 因为f1现在才计算i\*i，但现在i的值已经变为3

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。

匿名函数

高阶函数可以接收函数做参数，有些时候，我们不需要显式地定义函数，直接传入匿名函数更方便。

在Python中，对匿名函数提供了有限支持。还是以map()函数为例，计算 f(x)=x2 时，除了定义一个f(x)的函数外，还可以直接传入匿名函数：

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

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

通过对比可以看出，匿名函数 lambda x: x \* x 实际上就是：

def f(x):

return x \* x

关键字lambda表示匿名函数，冒号前面的x 表示函数参数。

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。

使用匿名函数，可以不必定义函数名，直接创建一个函数对象，很多时候可以简化代码：

>>> sorted([1, 3, 9, 5, 0], lambda x,y: -cmp(x,y))

[9, 5, 3, 1, 0]

返回函数的时候，也可以返回匿名函数：

>>> myabs = lambda x: -x if x < 0 else x

>>> myabs(-1)

1

>>> myabs(1)

1

编写无参数decorater

Python的decorator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。

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。

考察一个@log的定义：

def log(f):

def fn(x):

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

return f(x)

return fn

对于阶乘函数，@log工作得很好：

@log

def factorial(n):

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

print factorial(10)

结果：

call factorial()...

3628800

但是，对于参数不是一个的函数，调用将报错：

@log

def add(x, y):

return x + y

print add(1, 2)

结果：

Traceback (most recent call last):

File "test.py", line 15, in <module>

print add(1,2)

TypeError: fn() takes exactly 1 argument (2 given)

因为add()函数需要传入两个参数，但是@log写死了只含一个参数的返回函数。

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：

def log(f):

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

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

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

return fn

现在，对于任意函数，@log都能正常工作。

带参数的decorator  
如果有的函数非常重要，希望打印出'[INFO] call xxx()...'，有的函数不太重要，希望打印出'[DEBUG] call xxx()...'，这时，log函数本身就需要传入'INFO'或'DEBUG'这样的参数，类似这样：

@log('DEBUG')

def my\_func():

pass

把上面的定义翻译成高阶函数的调用，就是：

my\_func = log('DEBUG')(my\_func)

上面的语句看上去还是比较绕，再展开一下：

log\_decorator = log('DEBUG')

my\_func = log\_decorator(my\_func)

上面的语句又相当于：

log\_decorator = log('DEBUG')

@log\_decorator

def my\_func():

pass

所以，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：

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

对于这种3层嵌套的decorator定义，你可以先把它拆开：

# 标准decorator:

def log\_decorator(f):

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

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

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

return wrapper

return log\_decorator

# 返回decorator:

def log(prefix):

return log\_decorator(f)

拆开以后会发现，调用会失败，因为在3层嵌套的decorator定义中，最内层的wrapper引用了最外层的参数prefix，所以，把一个闭包拆成普通的函数调用会比较困难。不支持闭包的编程语言要实现同样的功能就需要更多的代码。

完善decoator

@decorator可以动态实现函数功能的增加，但是，经过@decorator“改造”后的函数，和原函数相比，除了功能多一点外，有没有其它不同的地方？

在没有decorator的情况下，打印函数名：

def f1(x):

pass

print f1.\_\_name\_\_

**输出：**f1

有decorator的情况下，再打印函数名：

def log(f):

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

print 'call...'

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

return wrapper

@log

def f2(x):

pass

print f2.\_\_name\_\_

**输出：** wrapper

可见，由于decorator返回的新函数函数名已经不是'f2'，而是@log内部定义的'wrapper'。这对于那些依赖函数名的代码就会失效。decorator还改变了函数的\_\_doc\_\_等其它属性。如果要让调用者看不出一个函数经过了@decorator的“改造”，就需要把原函数的一些属性复制到新函数中：

def log(f):

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

print 'call...'

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

wrapper.\_\_name\_\_ = f.\_\_name\_\_

wrapper.\_\_doc\_\_ = f.\_\_doc\_\_

return wrapper

这样写decorator很不方便，因为我们也很难把原函数的所有必要属性都一个一个复制到新函数上，所以Python内置的functools可以用来自动化完成这个“复制”的任务：

import functools

def log(f):

@functools.wraps(f)

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

print 'call...'

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

return wrapper

最后需要指出，由于我们把原函数签名改成了(\*args, \*\*kw)，因此，无法获得原函数的原始参数信息。即便我们采用固定参数来装饰只有一个参数的函数：

def log(f):

@functools.wraps(f)

def wrapper(x):

print 'call...'

return f(x)

return wrapper

也可能改变原函数的参数名，因为新函数的参数名始终是 'x'，原函数定义的参数名不一定叫 'x'。

偏函数

当一个函数有很多参数时，调用者就需要提供多个参数。如果减少参数个数，就可以简化调用者的负担。

比如，int()函数可以把字符串转换为整数，当仅传入字符串时，int()函数默认按十进制转换：

>>> int('12345')

12345

但int()函数还提供额外的base参数，默认值为10。如果传入base参数，就可以做 N 进制的转换：

>>> int('12345', base=8)

5349

>>> int('12345', 16)

74565

假设要转换大量的二进制字符串，每次都传入int(x, base=2)非常麻烦，于是，我们想到，可以定义一个int2()的函数，默认把base=2传进去：

def int2(x, base=2):

return int(x, base)

这样，我们转换二进制就非常方便了：

>>> int2('1000000')

64

>>> int2('1010101')

85

functools.partial就是帮助我们创建一个偏函数的，不需要我们自己定义int2()，可以直接使用下面的代码创建一个新的函数int2：

>>> import functools

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

>>> int2('1000000')

64

>>> int2('1010101')

85

所以，functools.partial可以把一个参数多的函数变成一个参数少的新函数，少的参数需要在创建时指定默认值，这样，新函数调用的难度就降低了。