



访问：11639次

积分：602

等级：

排名：千里之外


原创：48篇

转载：0篇

译文：0篇

评论：3条

文章搜索

 目录视图 摘要视图 订阅

赠书 | 异步2周年,技术图书免费选 程序员8月书讯 项目管理+代码托管+文档协作,开发更流畅

## Python 函数式编程（高阶函数、把函数作为参数、map()函数、reduce()函数、filter()函数、自定义排序函数、函数返回函数、闭包、匿名函数、装饰器decorator）

标签：Python 函数式编程 高阶函数 装饰器decorator map reduce filter

2017-07-10 18:12

64人阅读

评论(0)

 分类： Python学习（32） ▼

版权声明：本文为博主原创文章，未经博主允许不得转载。

目录(?) 

### 一、函数式编程

什么是函数式编程？

函数：function

函数式:functional

函数不等于函数式，好比计算不等于计算机

关闭





#### 阅读排行

- 神经网络与深度学习 使用 (4197)
- Java/Web调用Hadoop进 (3724)
- 神经网络与深度学习 1.6 (517)
- 神经网络与深度学习 第二 (442)
- 一层简单人工神经网络的 (280)
- 安卓4.4下MediaPlayer S (184)
- Vmware Ubuntu 下Hive的 (146)
- Android中采用MVP设计 (132)
- Vmware + Ubuntu + Had (130)
- Android开发设置EditText (114)

函数式编程的特点：

把计算视为函数而非指令

纯函数式编程：不需要变量，没有副作用，**测试**简单

支持高阶函数，代码简洁

#### Python支持的函数式编程

不是纯函数式编程：允许有变量

支持高阶函数，函数也可以作为变量传入

支持闭包：有了闭包就能返回函数

有限度地支持匿名函数

#### 二、高阶函数

变量可以指向函数

Eg:

```
print(abs(-5))
```

```
f = abs
```

```
print(f(-5))
```

关闭





\* Linux的任督二脉：进程调度和内存管理

\* 秒杀系统的一点思考

\* TCP网络通讯如何解决分包粘包问题

\* 技术与技术人员的价值

\* GitChat:人工智能 | 除了深度学习，机器翻译还需要啥？

#### 最新评论

一层简单人工神经网络的Java实现:  
mars50887355: 你好，几时多交流交流，QQ 50887355

Java/Web调用Hadoop进行MapF  
土豆拍死马铃薯: @zhuojiajin:是的，在同一台linux机器上

函数名就是指向函数的变量

高阶函数：能接收函数作为参数的函数

变量可以指向函数

函数的参数可以接收变量

一个函数可以接收另一个函数作为参数

能接收函数作为参数的函数称为高阶函数

Eg:

```
def add(x,y,f):  
    return f(x) + f(y)
```

```
print(add(-1,2,abs))
```

### 三、python把函数作为参数

在2.1小节中，我们讲了高阶函数的概念，并编写了一个简单的高阶函数：

```
def add(x, y, f):  
  
    return f(x) + f(y)
```

如果传入abs作为参数f的值：

关闭







```
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)
```

由于参数  $x$ ,  $y$  和  $f$  都可以任意传入，如果  $f$  传入其他函数，就可以得到不同的返回值。

### 任务

利用 `add(x,y,f)` 函数，计算：

```
import math
```

```
def add(x, y, f):
```

```
    return f(x) + f(y)
```

```
print add(25, 9, math.sqrt)
```

## 四、python中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  $f$  作用到 `list` 的每个元素，从而得到一个新的 `list` 并返回。

例如，对于 `list [1, 2, 3, 4, 5, 6, 7, 8, 9]`





如果希望把list的每个元素都作平方，就可以用map()函数：

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, 16, 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。

## 任务

假设用户输入的英文名字不规范，没有按照首字母大写，后续字母小写的规则（若干不规范的英文名字）变成一个包含规范英文名字的list：

输入：['adam', 'LISA', 'barT']

输出：['Adam', 'Lisa', 'Bart']

```
def format_name(s):
```

```
    return s[0].upper() + s[1:].lower()
```

关闭





```
print map(format_name, ['adam', 'LISA', 'barT'])
```

## 五、python中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：

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()**

**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

关闭







```
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**。

## 任务

Python内置了求和函数sum()，但没有求积的函数，请利用recude()来求积：

输入：[2, 4, 5, 7, 12]

输出：2\*4\*5\*7\*12的结果

```
def prod(x, y):
```

```
    return x*y
```

```
print reduce(prod, [2, 4, 5, 7, 12])
```

补充：

reduce函数：

在Python 3里,reduce()函数已经被从全局名字空间里移除了,它现在被放置在f

入：

```
>>> from functools import reduce
```

```
>>> print(l1)
```

```
[0, 1, 2, 3, 4, 5, 6]
```

关闭





```
>>> reduce(f4, l1)
```

```
21
```

## 六、python中filter()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** 对 **list** 中的每个元素进行判断，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，返回由符合条件的元素组成的新 **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'])
```

关闭







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'

## 任务

请利用filter()过滤出1~100中平方根是整数的数，即结果应该是：

[1,4, 9, 16, 25, 36, 49, 64, 81, 100]

```
import math
```

```
def is_sqr(x):
```

```
    r = int(math.sqrt(x))
```

```
    return r*r==x
```

```
for x in filter(is_sqr, range(1, 101)):
```

```
    print (x)
```

关闭





## 七、python中自定义排序函数

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，

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)
```

关闭





```
[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。

## 任务

对字符串排序时，有时候忽略大小写排序更符合习惯。请利用sorted()高阶函数，实现忽略大小写排序的

输入：['bob', 'about', 'Zoo', 'Credit']

输出：['about', 'bob', 'Credit', 'Zoo']

```
def cmp_ignore_case(s1, s2):
```

```
    u1 = s1.upper()
```

```
    u2 = s2.upper()
```

```
    if u1 < u2:
```

```
        return -1
```

```
    if u1 > u2:
```

```
        return 1
```

```
    return 0
```

```
print sorted(['bob', 'about', 'Zoo', 'Credit'], cmp_ignore_case)
```

关闭







补充：

python3 sorted取消了对cmp的支持。

python3 帮助文档：

sorted(iterable , key=None,reverse=False)

key接受一个函数，这个函数只接受一个元素，默认为None

reverse是一个布尔值。如果设置为True，列表元素将被倒序排列，默认为False

着重介绍key的作用原理：

key指定一个接收一个参数的函数，这个函数用于从每个元素中提取一个用于比较的关键字。默认值为None。

关闭

例1：

```
students = [('john', 'A', 15), ('jane', 'B', 12), ('dave','B', 10)]
```

```
def select(x):  
    return x[2]
```

```
#按照年龄来排序
```

```
print (sorted(students,key=select))
```





结果：[('dave', 'B', 10), ('jane', 'B', 12), ('john', 'A', 15)]

例2.这是一个字符串排序，排序规则：小写<大写<奇数<偶数

s = 'asdf234GDSdsf23' #排序:小写-大写-奇数-偶数

```
def select2(x):
```

```
    return x.isdigit(),x.isdigit() and int(x) % 2 == 0,x.isupper(),x
```

```
print(''.join(sorted(s, key=select2)))
```

原理：先比较元组的第一个值，FALSE<TRUE，如果相等就比较元组的下一个值，以此类推。

先看一下Boolean value 的排序：

```
print(sorted([True,Flase]))==>结果[False,True]
```

Boolean 的排序会将 False 排在前，True排在后。

1.x.isdigit()的作用是把数字放在前边,字母放在后边.

2.x.isdigit() and int(x) % 2 == 0的作用是保证奇数在前，偶数在后。

3.x.isupper()的作用是在前面基础上,保证字母小写在前大写在后.

关闭





4.最后的x表示在前面基础上,对所有类别数字或字母排序。

最后结果：addffssDGS33224

例3：一道面试题：

```
list1=[7, -8, 5, 4,0, -2, -5]
```

#要求1.正数在前负数在后 2.整数从小到大 3.负数从大到小

```
def select3(x):
```

```
    return x<0 , abs(x)
```

```
list1=[7, -8, 5, 4, 0, -2, -5]
```

#要求1.正数在前负数在后 2.整数从小到大 3.负数从大到小

```
print (sorted(list1,key=select3))
```

运行结果：[0, 4, 5, 7, -2, -5, -8]

解题思路：先按照正负排先后，再按照大小排先后。

关闭







## 八、python中返回函数

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 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：
```

关闭





```
<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)
```

关闭





```
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。

## 任务

请编写一个函数calc\_prod(lst)，它接收一个list，返回一个函数，返回函数可以计算参数的乘积。

```
from functools import reduce
```

```
def prod2(x, y):
```

```
    return x*y
```

```
#print (reduce(prod2, [2, 4, 5, 7, 12]))
```

```
def calc_prod(lst):
```

```
    def calc():
```

```
        return reduce(prod2, lst)
```

```
    return calc
```

关闭







```
f = calc_prod([1, 2, 3, 4])  
print (f())
```

## 九、python中闭包

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():
```

关闭





```
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)
```

关闭





```
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。

## 任务

返回闭包不能引用循环变量，请改写count()函数，让它正确返回能计算1x1、2x2、3x3的函数。

```
def count():
```

```
    fs = []
```

```
    for i in range(1, 4):
```

```
        def f():
```

```
            def g():
```

```
                return j*j
```

```
            return g
```

关闭







```
r = f(i)

fs.append(r)

return fs

f1, f2, f3 = count()

print f1(), f2(), f3()
```

## 十、python中匿名函数

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  $f(x)=x^2$  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 `lambdax: 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

关闭





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
```

## 任务

利用匿名函数简化以下代码：

```
def is_not_empty(s):
```

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

```
filter(is_not_empty, ['test', None, '', 'str', ' ', 'END'])
```

```
def is_not_empty(s):
```





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

```
print filter(lambda s : s and len(s.strip()) > 0, ['test', None, '', 'str', ' ', 'END'])
```

## 十一、装饰器 decorator

什么是装饰器？

定义了一个函数，想在运行时动态增加功能，又不想改动函数本身的代码？

## 十二、python中编写无参数decorator

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

关闭

使用 decorator 用Python提供的 **@** 语法，这样可以避免手动编写 **f = decorate**

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)
```

关闭





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。

## 任务

请编写一个`@performance`，它可以打印出函数调用的时间。

```
import time
```

```
def performance(f):
```

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

```
        t1 = time.time()
```

```
        r = f(*args, **kw)
```

```
        t2 = time.time()
```

关闭





```
print 'call %s() in %fs' % (f.__name__, (t2 - t1))
```

```
return r
```

```
return fn
```

```
@performance
```

```
def factorial(n):
```

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

```
print factorial(10)
```

### 十三、python中编写带参数decorator

考察上一节的 `@log` 装饰器：

```
def log(f):
```

```
    def fn(x):
```

```
        print 'call ' + f.__name__ + '()...'
```

```
        return f(x)
```

```
    return fn
```

发现对于被装饰的函数，log打印的语句是不能变的（除了函数名）。

关闭







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

```
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__)
```

关闭





```
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|

## 任务

上一节的@performance只能打印秒，请给 @performance 增加一个参数，允许传入's'或'ms'：

```
@performance('ms')
```

```
def factorial(n):
```

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

```
import time
```

```
def performance(unit):
```

```
    def perf_decorator(f):
```

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

关闭







```
t1 = time.time()

r = f(*args, **kw)

t2 = time.time()

t = (t2 - t1) * 1000 if unit=='ms' else (t2 - t1)

print 'call %s() in %f %s' % (f.__name__, t, unit)

return r

return wrapper

return perf_decorator
```

```
@performance('ms')
```

```
def factorial(n):
```

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

```
print factorial(10)
```

## 十四、python中完善decorator

`@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，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，所以直接打印f2.\_\_name\_\_就会失效。decorator还改变了函数的\_\_doc\_\_等其它属性。如果要想让调用者看到原来的函数名，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，无法获得固定参数来装饰只有一个参数的函数：



关闭





```
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'。

## 任务

请思考带参数的@decorator，@functools.wraps应该放置在哪：

```
def performance(unit):
```

```
    def perf_decorator(f):
```

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

```
            ???
```

```
        return wrapper
```

```
    return perf_decorator
```

```
import time, functools
```

关闭





```
def performance(unit):
```

```
    def perf_decorator(f):
```

```
        @functools.wraps(f)
```

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

```
            t1 = time.time()
```

```
            r = f(*args, **kw)
```

```
            t2 = time.time()
```

```
            t = (t2 - t1) * 1000 if unit=='ms' else (t2 - t1)
```

```
            print 'call %s() in %f %s' % (f.__name__, t, unit)
```

```
            return r
```

```
        return wrapper
```

```
    return perf_decorator
```

```
@performance('ms')
```

```
def factorial(n):
```

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

关闭





```
print factorial.__name__
```

## 十五、python中偏函数

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，于是默认把`base=2`传进去：

```
def int2(x, base=2):
```

关闭







```
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，少一样，新函数调用的难度就降低了。

## 任务

在第7节中，我们在`sorted`这个高阶函数中传入自定义排序函数就可以实现忽略大小写排序，现在我们把一个复杂调用变成一个简单的函数：





```
sorted_ignore_case(iterable)
```

```
import functools
```

```
sorted_ignore_case =functools.partial(sorted, cmp=lambda s1, s2: cmp(s1.upper(), s2.upper()))
```

```
print sorted_ignore_case(['bob', 'about','Zoo', 'Credit'])
```

顶  
0

踩  
0

上一篇 [Python 列表生成式](#)

下一篇 [Python 模块和包，使用自定义的模块和包](#)

#### 相关文章推荐

- [Python学习笔记2：函数式编程](#)
- [【直播】70天软考冲刺计划--任铄](#)
- [JavaScript匿名函数](#)
- [【直播】打通Linux脉络 进程、线程、调度--宋宝华](#)
- [廖雪峰Python的研读笔记（二）函数式编程](#)
- [【直播】机器学习之凸优化--马博士](#)
- [Python函数式编程](#)
- [一、Python 进阶](#)
- [【课程】3小时掌握Python](#)
- [函数式编程在Python中的应用](#)
- [【课程】深度学习入门](#)
- [python学习（四）](#)
- [Java调用oracle数据库](#)
- [Python函数式编程](#)





- 【套餐】MATLAB基础+MATLAB数据分析与统计--...

- 匿名函数的一点知识



查看评论

暂无评论

发表评论

用户名： haijunz

评论内容：



提交

\* 以上用户言论只代表其个人观点，不代表CSDN网站的观点或立场

关闭

公司简介 | 招贤纳士 | 广告服务 | 联系方式 | 版权声明 | 法律顾问 | 问题报告 | 合作伙伴 | 论坛反馈

网站客服

杂志客服

微博客服

webmaster@csdn.net

400-660-0108

| 北京创新乐知信息技术有限公司 版权所有 | 江苏知

京 ICP 证 09002463 号 | Copyright © 1999-2017, CSDN.NET, All Rights Reserved

