**python的functools模块常用方法介绍**

2017年08月17日 08:25:20

阅读数：330

functools模块中有两个方法是从\_functools引入的，分别是partial和reduce函数。   
functools模块是python2.5版本新增的模块。

**1.偏函数partial**   
python中的偏函数与数学的偏函数意义不同，python中的partial函数用于当某个函数的参数值为固定时使用。参数可以为位置参数args，也可以为keyword参数kargs   
**(1) 不带默认参数的加法函数**   
例如，当我们要求三个数的加法时，我们可以定义一个函数如下：

from functools import partial

def add(a, b, c):

return a + b + c

print add(1, 2, 3)

* 1
* 2
* 3
* 4

输出：

6

* 1

**(2) 带默认参数的加法函数**   
如果参数a为固定值100，则我们可以利用partial函数改写如下：

def add(a, b, c):

return a + b + c

print "add", add(1, 2, 3)

add\_plus = partial(add, 100)

print "partial(add, 100)", add\_plus(5, 6)

add\_plus = partial(add, 100, 200)

print "partial(add, 100, 200)", add\_plus(5)

* 1
* 2
* 3
* 4
* 5
* 6
* 7

输出结果为：

add 6

partial(add, 100) 111

partial(add, 100, 200) 305

* 1
* 2
* 3

当函数中某些参数有默认值时，partial函数也能为默认参数匹配正确的位置。   
定义一个含有默认参数的函数：

def add(a, b=333, c):

return a + b + c

print "add", add(1, 2)

* 1
* 2
* 3

输出结果：

File "E:\python\_program\leetcode\python\_knowledge\functools\_test.py", line 10

def add(a, b=333, c):

SyntaxError: non-default argument follows default argument

* 1
* 2
* 3

抛出SyntaxError异常，原来python函数的设置有默认值的参数必须放在函数参数的最后。   
重新定义：

def add(a, b=333, c=6):

print "a,b,c的值分别为：",a,b,c

return a + b + c

print "add", add(1, 2)

add\_plus = partial(add, 100)

print "partial(add, 100)", add\_plus(5, 7)

add\_plus = partial(add, 100, 200)

print "partial(add, 100, 200)", add\_plus(5)

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8

输出结果：

add

a,b,c的值分别为： 1 2 6

9

partial(add, 100)

a,b,c的值分别为： 100 5 7

112

partial(add, 100, 200)

a,b,c的值分别为： 100 200 5

305

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9

从返回结果可以看出，   
当定义偏函数为partial(add, 100)时，其将100按add方法的顺序赋值给了参数a,因此调用add\_plus(5, 7)时，5赋值给了b,7赋值给了c,因此相加的结果为112。   
当定义偏函数为partial(add, 100, 200)时，其将100给了默认参数a,200给了默认函数b，因此调用add\_plus(5)时，5赋值给了c，因此相加的结果为305。   
**注意：调用add\_plus方法的参数个数=add方法的参数个数-partial的默认参数个数。**   
**(3) 带位置参数和keyword参数的函数**

def add(\*args, \*\*kargs):

print "args的值为：",args

print "kargs的值为：",kargs

print "add", add(1, 2)

new\_add = partial(add,1,2,3,a=100,b=200,c=300)

print "new\_add", new\_add(4,5,e=400,f=500)

* 1
* 2
* 3
* 4
* 5
* 6
* 7

输出结果：

add args的值为： (1, 2)

kargs的值为： {}

None

new\_add args的值为： (1, 2, 3, 4, 5)

kargs的值为： {'a': 100, 'c': 300, 'b': 200, 'e': 400, 'f': 500}

None

* 1
* 2
* 3
* 4
* 5
* 6

从结果可以看出，函数可以传递任何形式的参数，位置参数会追加到args中，keyword参数会追加到kargs。

**2.reduce**   
collections中的reduce函数与python内置的reduce()函数功能相同，在该模块中设置该函数的目的是为了与python3兼容。

**3.update\_wrapper**

**4.wraps**   
functools.wraps(wrapped[, assigned][, updated])   
This is a convenience function for invoking partial(update\_wrapper, wrapped=wrapped, assigned=assigned, updated=updated) as a function decorator when defining a wrapper function.

例如(python2.7.2官方文档中的例子):

>>> from functools import wraps

>>> def my\_decorator(f):

... @wraps(f)

... def wrapper(\*args, \*\*kwds):

... print 'Calling decorated function'

... return f(\*args, \*\*kwds)

... return wrapper

...

>>> @my\_decorator

... def example():

... """Docstring"""

... print 'Called example function'

...

>>> example()

Calling decorated function

Called example function

>>> example.\_\_name\_\_

'example'

>>> example.\_\_doc\_\_

'Docstring'

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 20

结果说明：   
example函数被装饰器my\_decorator修饰，装饰器的内部函数被wraps函数修饰，需要注意的是，被wraps修饰后，example函数的名字和说明文档与example函数一致，但是去掉@wraps(f)修饰后，输出的example函数名称为wrapper，说明文档为None。

**5.total\_ordering(2.7版本新增功能)**   
total\_ordering是实现更多比较排序方法类的装饰器。简化了比较操作。该类必须定义一下方法中的一种 **lt**(), **le**(), **gt**(), or **ge**()，另外必须定义 **eq**() 方法。   
例子：

class Student:

def \_\_init\_\_(self, firstname, lastname):

self.firstname = firstname

self.lastname = lastname

def \_\_eq\_\_(self, other):

return ((self.lastname.lower(), self.firstname.lower()) ==

(other.lastname.lower(), other.firstname.lower()))

def \_\_lt\_\_(self, other):

return ((self.lastname.lower(), self.firstname.lower()) <

(other.lastname.lower(), other.firstname.lower()))

a = Student("ab", "cd")

b = Student("ac", "cd")

print "a<b", a<b

print "a", a

print "b", b

print "sorted([a,b])", sorted([a,b])

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 20

输出：

a<b True

a <\_\_main\_\_.Student instance at 0x0000000002A46288>

b <\_\_main\_\_.Student instance at 0x0000000002A462C8>

sorted([a,b]) [<\_\_main\_\_.Student instance at 0x0000000002A46288>, <\_\_main\_\_.Student instance at 0x0000000002A462C8>]

* 1
* 2
* 3
* 4
* 5

**注意：python的list、dict、int、string 都是可以比较大小的。**

**6.cmp\_to\_key(2.7版本新增功能)**   
functools.cmp\_to\_key(func)   
该功能主要是为了解决python版本，函数不兼容的问题，比如sorted(), min(), max(), heapq.nlargest(), heapq.nsmallest(), itertools.groupby()等函数。

**拿sorted()函数举例**   
在python2.x版本中，sorted函数的格式为sorted(iterable[, cmp[, key[, reverse]]])   
，在python3.x版本中，sorted函数的格式为sorted(iterable[, key][, reverse]) 。所以当我们需要将python2.x的程序移植到python3.x的环境中时，我们需要把原来cmp参数位置的函数，转变为可以传递给参数key函数，此时我们就需要用到collections模块下的cmp\_to\_key方法。

###### functools.cmp\_to\_key(func)

2016年04月13日 16:23:58

阅读数：1755

将老式的比较函数（comparison function）转化为关键字函数（key function）。与接受key function的工具一同使用（如 sorted(), min(), max(), heapq.nlargest(), itertools.groupby())。该函数主要用来将程序转成 Python 3 格式的，因为 Python 3 中不支持比较函数。

　　比较函数是可调用的，接受两个参数，比较这两个参数并根据他们的大小关系返回负值、零或正值中的某一个。关键字函数也是可调用的，接受一个参数，同时返回一个可以用作排序关键字的值。

　　例如：

sorted(iterable, key=cmp\_to\_key(locale.strcoll))

**class** Solution:  
# @param {integer[]} nums # @return {string} **def** largestNumber(self, nums):  
**from** functools **import** cmp\_to\_key  
 key = cmp\_to\_key(**lambda** x,y: int(y) - int(x))  
 res = **', '**.join(sorted(map(str, nums), key=key))  
**return** res **or '0'**nums = [-1,-2,3,4,9,2,3,4,5]  
s = Solution()  
**print** map(str, nums) #列表不去重**print** s.largestNumber(nums)  
nums = {1,02,3,4,9,2,3,4,5}  
**print** map(str, nums) #字典会去重**print** s.largestNumber(nums)

输出：

['-1', '-2', '3', '4', '9', '2', '3', '4', '5']

9, 5, 4, 4, 3, 3, 2, -1, -2

['1', '2', '3', '4', '5', '9']

9, 5, 4, 3, 2, 1

cmp\_to\_key函数的源码如下：

def cmp\_to\_key(mycmp):

"""Convert a cmp= function into a key= function"""

class K(object):

\_\_slots\_\_ = ['obj']

def \_\_init\_\_(self, obj, \*args):

self.obj = obj

def \_\_lt\_\_(self, other):

return mycmp(self.obj, other.obj) < 0

def \_\_gt\_\_(self, other):

return mycmp(self.obj, other.obj) > 0

def \_\_eq\_\_(self, other):

return mycmp(self.obj, other.obj) == 0

def \_\_le\_\_(self, other):

return mycmp(self.obj, other.obj) <= 0

def \_\_ge\_\_(self, other):

return mycmp(self.obj, other.obj) >= 0

def \_\_ne\_\_(self, other):

return mycmp(self.obj, other.obj) != 0

def \_\_hash\_\_(self):

raise TypeError('hash not implemented')

return K

* 1、**比较函数**是任何的可以接受两个参数，并且在函数中比较他们的函数。如果第一个参数小于第二个参数，则返回一个负数；如果相等，则返回0；如果大于返回正数。   
  key函数接受一个参数，并按照返回的值排列顺序。   
  sorted(iterable[, cmp[, key[, reverse]]]) 中的cmp参数形如cmp=lambda x,y: cmp(x.lower(), y.lower()).   
  cmp\_to\_key()函数使用实例：

from functools import cmp\_to\_key

s = [{"key":3},{"key":1},{"key":5}]

key = cmp\_to\_key(lambda x,y: x["key"] - y["key"])

b = sorted(s,key=key,reverse=True)

print b

输出结果：

[{'key': 5}, {'key': 3}, {'key': 1}]