**functools模块和函数工具**

标准库functools提供了若干关于函数的函数，这些工具函数以函数作为参数，返回修改后的函数。

**1.@functools.wraps装饰器**

函数包含若干属性，如\_\_name\_\_、\_\_doc\_\_等。函数被装饰后其特殊属性将变成相应装饰器函数的特殊属性内容，当使用反射时，可能会导致意外结果。使用functools模块的wraps装饰器用于解决该问题。使用wraps装饰函数foo，则foo装饰的函数的特殊属性保留。

@functools.wraps( wrapped,assigned = WRAPPER\_ASSIGNMENTS, updated = WRAPPER\_UPDATES )

其中assigned为元组，用于指定使用的wrapped函数的特殊属性，默认为functools模块常量；updated为元组，用于指定更新的wrapped函数的特殊属性。

>>>import functools

>>>functools.WRAPPER\_ASSIGNMENTS

( ‘\_\_module\_\_’ , ‘\_\_name\_\_’ , ‘\_\_qualname\_\_’ , ‘\_\_doc\_\_’ , ‘\_\_annotations\_\_’)

>>>functools.WRAPPER\_UPDATES

(‘\_\_dict\_\_’)

示例：

>>>import time functools

def foo(f):

‘’’foo function doc’’’

@functools.wraps(f)

def wrapper(\*x,\*\*y):

print(‘调用函数：’, f.\_\_name)

return f( \*x,\*\*y)

return wrapper

@foo

def bar(x):

‘’’bar function doc’’’

return x\*\*2

if \_\_name\_\_==’\_\_main\_\_’:

print(bar(2))

print(bar.\_\_name\_\_)

print(bar.\_\_doc\_\_)

>>>调用函数：bar

4

bar

bar function doc

如果没有加wraps装饰，则效果是：

>>>import time functools

def foo(f):

‘’’foo function doc’’’

def wrapper(\*x,\*\*y):

‘’’wrapper function doc’’’

print(‘调用函数：’, f.\_\_name)

return f( \*x,\*\*y)

return wrapper

@foo

def bar(x):

‘’’bar function doc’’’

return x\*\*2

if \_\_name\_\_==’\_\_main\_\_’:

print(bar(2))

print(bar.\_\_name\_\_)

print(bar.\_\_doc\_\_)

>>>调用函数：bar

4

wrapper

wrapper function doc

**2.functools.update\_wrapper函数**

使用update\_\_wrapper函数也可以指定wrapper函数使用指定的特殊属性。

functools.update\_wrapper( wrapper,wrapped,assogned = WRAPPER\_ASSIGNMENTS,update = WRAPPER\_UPDATES)

示例：

>>>import time functools

def foo(f):

‘’’foo function doc’’’

def wrapper(\*x,\*\*y):

‘’’wrapper function doc’’’

print(‘调用函数：’, f.\_\_name)

return f( \*x,\*\*y)

return functools.update\_wrapper(wrapper,f)

@foo

def bar(x):

‘’’bar function doc’’’

return x\*\*2

if \_\_name\_\_==’\_\_main\_\_’:

print(bar(2))

print(bar.\_\_name\_\_)

print(bar.\_\_doc\_\_)

>>>调用函数：bar

4

bar

bar function doc

**3.functools.total\_ordering装饰器**

支持大小比较的对象需要实现特殊方法：\_\_eq\_\_、\_\_lt\_\_、\_\_le\_\_、\_\_ge\_\_、\_\_gt\_\_。使用functools模块的total\_ordering装饰器装饰类，则只需要实现\_\_eq\_\_,以及\_\_lt\_\_、\_\_le\_\_、\_\_ge\_\_、\_\_gt\_\_中的任意一个即可。

示例：

>>>from functools import total\_ordering

@total\_ordering

class Door(object):

def \_\_init\_\_(self):

self.value = 0

self.first\_name = ''

self.last\_name = ''

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

print '=== my eq==='

return (self.first\_name, self.last\_name) == (other.first\_name, other.last\_name)

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

print '=== my total\_ordering==='

return (self.first\_name, self.last\_name) > (other.first\_name, other.last\_name)

a = Door()

b = Door()

a.first\_name = 'ouyang'

a.last\_name = 'guoge'

b.first\_name = 'aaaa'

b.last\_name = 'bbbb'

print a == b

print a > b

print a < b

print a <= b

print a >= b

>>> === my eq===

False

=== my total\_ordering===

True

=== my total\_ordering===

False

=== my total\_ordering===

False

=== my total\_ordering===

True

**[4.@functools.lru\_cache装饰器](mailto:4.@functools.lru_cache装饰器)**

可以缓存最多maxsize谷歌此函数的调用结果，从而提高程序执行效率，特别适合耗时的函数或i/o函数

@functools.lru\_cache(maxsize=128,typed=False)

maxsize为最大缓存次数，为None则无限制，设置为2的n次方时性能最佳。若typed为True，则不同参数类型的调用将分别缓存，例如f(3)和f(3.0)。

**5.partial对象**

重新绑定函数的可选参数，生成一个可调用（callable）的partial对象：

functools.partial( func , \*args , \*\*keywords)

其中func是函数，args是其位置参数，keywords是关键字参数

partial主要用于设置预先已知的参数，从而减少调用时传递参数的个数。示例：

>>>import functools,math

>>>pow2 = functools.partial( math.pow,2) #封装pow（x，y[，z]），指定参数x=2

>>>for i in range(11):

print(format(pow2(i), ‘0.0f’))

>>>1 2 4 8 16 32 64 128 256 512 1024

**6.reduce函数**

reduce使用指定的带两个参数的函数func，对一个数据集合（可迭代对象）的所有数据进行下列操作：使用第1,2个数据作为参数用func函数运算，得到的结果再与第3个数据作为参数用func函数运算，以此类推最后得到一个结果。可选的initializer为初始值。

functools.reduce(func,iterator [,initializer])

>>>import functools operator

functools.reduce(operator.add,[1,2,3,4,5]) #((((1+2)+3)+4)+5)=15

>>>15

>>>import functools operator

functools.reduce(operator.add,[1,2,3,4,5],10) #10+((((1+2)+3)+4)+5)=25

>>>25