寻找自我的博客

python 函数修饰器

分类： Python 2012-09-04 17:05 115人阅读 评论(0) 收藏 举报

装饰器的语法以@开头，接着是装饰器要装饰的函数的申明等。

其实总体说起来，装饰器其实也就是一个函数，一个用来包装函数的函数，装饰器在函数申明完成的时候被调用，调用之后申明的函数被换成一个被装饰器装饰过后的函数。
装饰器分为无参装饰和有参装饰
无参装饰很简单
定义方法如下：
比如先定义一个装饰方法：

```
def FirstDeco(func):

    print '第一个装饰器'

    return func

@FirstDeco

def test():

    print 'asdf'

....
```

申明完成之后显示

```
...
```

第一个装饰器
可见装饰器在函数定义完成的时候被触发
然后，咱们运行test
获得asdf
多参装饰：
多参装饰复杂一点，多参装饰的时候，装饰函数先处理参数，再生成一个新的装饰器函数，然后对函数进行装饰

具体代码如下：

```
def deco(x):
    print ('%s 开始新装饰')
    def newDeco(func):
        def test(a,b):
            print ('begin')
            returnv = func(a,b)
            print ('end')
            return returnv
```

```
        return test
    return newDeco...
```

这里定义了一个装饰其函数deco,里面有一个参数x,这个时候，我们没有直接使用func作为装饰函数的参数，而是只用了参数x作为参数，之后定义一个新的装饰函数，newdeco，该函数才装饰

然后定义如下：

```
>>> @deco(3)
... def mytest(x,y):
... if x>y:
... print x
... else:
... print y
...
%s 开始新装饰
```

运行之后的结果为

```
%s 开始新装饰
>>> mytest(3,4)
begin
4
end
```

参考资料：
装饰方法的产生：

Python2.2通过增加静态方法和类方法扩展了Python的对象模型。但是当时没有提供一个简化的语法去定义static/class方法，只得在定义好的方法尾部去调用staticmethod()/classmethod()方法达到目的。例如：

```
class C:
def meth (cls):

meth = classmethod(meth) # 使meth方法成为类方法
```

但是这样会造成一个问题：当一个方法比较长时，很容易忘记尾部的调用。为了简化这个操作一个新的语法被加了进来：方法装饰，以@开头后跟装饰方法名，如@staticmethod/@classmethod，由此产生出decorator方法及decorator模式。现在我们可以这样写：

```
class C:
@classmethod
def meth (cls):
```

可以对一个方法应用多个装饰方法:

```
@A
```

@B
@C
def f ():

#等价于下面的形式，Python会按照应用次序依次调用装饰方法（最近的先调用）
def f():
f = A(B(C(f)))
装饰方法解析：

每个decorator只是一个方法， 可以是自定义的或者内置的（如内置的@staticmethod/@classmethod）。decorator方法把要装饰的方法作为输入参数，在函数体内可以进行任意的操作(可以想象其中蕴含的威力强大，会有很多应用场景)，只要确保最后返回一个可执行的函数即可（可以是原来的输入参数函数，或者是一个新函数）。decorator的作用对象可以是模块级的方法或者类方法。decorator根据应用时的参数个数不同分为两类：无参数decorator，有参数decorator。下面分别介绍。

无参数decorator:
def deco(func):
"""无参数调用decorator声明时必须有一个参数，这个参数将接收要装饰的方法"""
print "Enter decorator" #进行额外操作
func.attr = 'decorated' #对函数进行操作，增加一个函数属性
return func #返回一个可调用对象(此例还是返回作为输入参数的方法)
#返回一个新函数时，新函数可以是一个全局方法或者decorator函数的内嵌函数，
#只要函数的签名和被装饰的函数相同

@deco
def MyFunc(): #应用@deco修饰的方法
print "Enter MyFunc"

MyFunc() #调用被装饰的函数

注意：当使用上述方法定义一个decorator方法时，函数体内的额外操作只在被装饰的函数首次调用时执行，如果要保证额外操作在每次调用被装饰的函数时都执行，需要换成如下的写法：
def deco(func):
def replaceFunc(): #定义一个内嵌函数，此函数包装了被装饰的函数，并提供额外操作的代码
print "Enter decorator" #进行额外操作
return func() #产生对被装饰函数的调用
return replaceFunc #由于返回的是这个新的内嵌函数，所以确保额外操作每次调用得以运行

@deco
def MyFunc(): #应用@deco修饰的方法
print "Enter MyFunc"

MyFunc() #调用被装饰的函数

有参数decorator:

```
def decoWithArgs(arg):
"""由于有参数的decorator函数在调用时只会使用应用时的参数而不接收被装饰的函数做为参数，
所以必须返回一个decorator函数， 由它对被装饰的函数进行封装处理"""
def newDeco(func): #定义一个新的decorator函数
def replaceFunc(): #在decorator函数里面再定义一个内嵌函数，由它封装具体的操作
print "Enter decorator" #进行额外操作
return func() #对被装饰函数进行调用
return replaceFunc
return newDeco #返回一个新的decorator函数

@decoWithArgs("demo")
def MyFunc(): #应用@decoWithArgs修饰的方法
print "Enter MyFunc"

MyFunc() #调用被装饰的函数
```

当我们对某个方法应用了装饰方法后， 其实就改变了被装饰函数名称所引用的函数代码块入口点，使其重新指向了由装饰方法所返回的函数入口点。由此我们可以用decorator改变某个原有函数的功能，添加各种操作，或者完全改变原有实现。