Python装饰器

一.什么是装饰器?

Python装饰器就是用于拓展原来函数功能的一种函数，这个函数的特殊之处在于它的返回值也是一个函数(装饰器相当于一个嵌套函数)，使用Python装饰器的好处就是在不用更改原函数代码前提下给函数增加新的功能，一般而言，我们要想拓展原来函数代码，最直接的办法就是侵入代码里面修改.

例如:

In [2]:
```
#示例代码1.原函数
import time
def func():
    print("hello")
    time.sleep(1)
    print("world")

func()

#原函数代码需要1秒时间才能执行完成.
```

```
hello
world
```

以上我们最原始的的一个函数,然后我们试图记录下这个函数执行的总时间,那最简单的做法如下:

In [5]:
```
#示例代码2.
import time
def func():
    startTime = time.time()
    print("hello")
    time.sleep(1)
    print("world")
    endTime = time.time()
    msecs = endTime - startTime
    print("time is %s ms" %msecs)
func()
```

```
hello
world
time is 1.00231313705 ms
```

假设func函数的代码是公司的核心代码,旁人不可以轻易对其进行修改,那么我们应该怎么做呢?

In [7]:
```
#示例代码3.

def deco(func):
    startTime = time.time()
    func()
    endTime = time.time()
    msecs = endTime - startTime
    print("time is %s ms" %msecs)


def func():
    print("hello")
    time.sleep(1)
    print("world")

#func()
deco(func)
```

```
hello
world
time is 1.00202894211 ms
```

这里我们定义了一个函数deco， 它的参数是一个函数,然后给这个函数嵌入了计时功能,但是如果公司的核心代码区域有多个func函数,从func01 — func100000，那么想要拓展这些函数的功能,需要重复执行deco(func)多次,不够方便,所以我们可以使用装饰器来解决这个问题.

In [3]:
```
#装饰器示例代码1.

import time
def deco(func):
    def _deco():
        print('start...')
        begtime = time.time()
        func()
        #print x
        #time.sleep(1)
        endtime = time.time()
        print('called func cost time %s' %(endtime - begtime))
    return _deco

@deco
def myfunc():
    print('myfunc() called')
    time.sleep(1)
    return 'ok!'

myfunc()
```

```
start...
myfunc() called
called func cost time 1.0036578178405762
```

这里的deco函数就是最原始的装饰器,它的参数是一个函数,然后返回值也是一个函数,其中作为参数的这个函数func()就在返回函数__deco()的内部执行,然后在函数func()前面加上@deco，func()函数就相当于被注入了计时功能，现在只要调用func(),它就已经变身为"新的功能更多"的函数了,所以这里装饰器就像一个注入符号,有了它拓展了原来函数的功能既不需要侵入函数内更改代码,也不需要重复执行原函数.

In [13]:
```
#装饰器示例代码2. 对带参数的函数进行装饰

def deco(func):
    def _deco(a, b):
        print 'before myfunc() called.'
        ret = func(a, b)
        print 'after myfunc() called result %s' %ret
        return ret
    return _deco

@deco
def myfunc(a, b):
    print 'myfunc %s, %s called' %(a, b)
    return a+b
myfunc(1, 2)
myfunc(3, 4)
```

```
before myfunc() called.
myfunc 1, 2 called
after myfunc() called result 3
before myfunc() called.
myfunc 3, 4 called
after myfunc() called result 7
```

Out[13]: 7

In [14]:
```
#装饰器示例代码3. 对参数不确定的函数进行装饰

def deco(func):
    def _deco(*args, **kwargs):
        print 'before %s called.' % func.__name__
        ret = func(*args, **kwargs)
        print 'after %s called. result %s' %(func.__name__, ret)
    return _deco

@deco
def myfunc(a, b):
    print 'myfunc(%s, %s) called.' %(a, b)
    return a + b

myfunc(1, 2)

@deco
def myfunc2(a, b, c):
    print 'myfunc(%s, %s, %s) called' %(a, b, c)
    return a+b+c
myfunc2(1, 2, 3)
```

```
before myfunc called.
myfunc(1, 2) called.
after myfunc called. result 3
before myfunc2 called.
myfunc(1, 2, 3) called
after myfunc2 called. result 6
```

In [16]:
```
#装饰器示例代码4. 让装饰器带参数,

def deco(arg):
    def _deco(func):
        def __deco():
            print 'before %s called [%s].' %(func.__name__, arg)
            func()
            print 'after %s called [%s].' %(func.__name__, arg)
        return __deco
    return _deco

@deco('mymodule1')
def myfunc():
    print 'myfunc() called.'

@deco('mymodule2')
def myfunc2():
    print 'myfunc2() called'

myfunc()
myfunc2()
```

```
before myfunc called [mymodule1].
myfunc() called.
after myfunc called [mymodule1].
before myfunc2 called [mymodule2].
myfunc2() called
after myfunc2 called [mymodule2].
```

In [17]:
```
#装饰器示例代码5. 多个装饰器修饰同一个函数

def deco1(func):
    def _deco():
        beg = time.time()
        print '开始的时间戳是: %s' %beg
        func()
        end = time.time()
        print '结束的时间戳是: %s' %end
        print '任务总共花费时间: %s' %(end-beg)
    return  _deco

def deco2(func):
    def _deco():
        print '这是装饰器2的开始...'
        func()
        print '这是装饰器2的结束...'
    return _deco

@deco1
@deco2
def func():
    print 'begin...'
    time.sleep(1)
    print 'end...'

func()
```

```
开始的时间戳是: 1539912171.35
这是装饰器2的开始...
begin...
end...
这是装饰器2的结束...
结束的时间戳是: 1539912172.35
任务总共花费时间: 1.00173020363
```

内层装饰器先返回

In [3]:
```
import time
def deco1(func):
    def _deco():
        begintime=time.time()
        print('func1 start')
        x = func()
        print(x, 'ssss')
        time.sleep(2)
        endtime=time.time()
        print('func cost time %s' %(endtime-begintime))
        return x + 5

    return _deco

def deco2(func):
    def _deco():
        print("deco2 is beginning...")
        x = func() + 2
        print("deco2 is ending...")
        return x #=3
    return _deco

@deco1
@deco2
def func1():
    print("hello")
    return 1

func1()
```

```
func1 start
deco2 is beginning...
hello
deco2 is ending...
3 ssss
func cost time 2.003434896469116
```

Out[3]: 8

In [19]:
```
#装饰器示例代码6.装饰器带类参数

import time

class locker:
    def __init__(self):
        print 'locker init should be called!!'

    @classmethod
    def acquired(cls):
        print 'locker function acquire called!!'

    @classmethod
    def release(cls):
        print 'locker function release called!!'

def deco(cls):
    def _deco(func):
        def __deco():
            cls.acquired()
            func()
            cls.release()
        return __deco
    return _deco

@deco(locker)
def func():
    """
    xxxx
    """
    print 'function begin...'
    time.sleep(1)
    print 'function end...'

func()
```

```
locker function acquire called!!
function begin...
function end...
locker function release called!!
```

二. 装饰器补充

Python装饰器在实现的时候, 被装饰后的函数其实已经是另外一个函数了(函数名等函数属性会发生改变), 简言之使用装饰器时, 原函数会损失一些信息.

In [20]:
```
#示例代码7.

def deco(func):
    def _deco(x):
        print func.__name__, func.__doc__
        result = func(x)
        return result
    return _deco

@deco
def func(x):
    """
    function func do something
    """
    time.sleep(1)
    return x + x * x


print func.__name__
print func.__doc__
```

```
_deco
None
```

被装饰的函数名叫func, 但是变成了*deco, func函数中的帮助文档, 也已经不再存在. 为了解决这个问题，Python的functools包中提供了一个叫wraps的装饰器来消除这样的副作用. 当写一个装饰器的时候, 最好在实现之前加上functools的wrap, 它能保留原有函数的名称和*__doc__.

In [22]:
```
#示例代码8.

import time
from functools import wraps

def deco(func):
    @wraps(func)
    def _deco(x):
        print func.__name__, func.__doc__
        result = func(x)
        return result
    return _deco

@deco
def func(x):
    """
    function func do something
    """
    time.sleep(1)
    return x + x * x


print func.__name__
print func.__doc__
```

```
func

    function func do something
```

In [ ]: