装饰器

一、闭包(抛转、引例)

1. 函数引用

```
def test1():
    print("--- in test1 func----")

#调用函数
test1()

#引用函数
ret = test1

print(id(ret))
print(id(test1))

#通过引用调用函数
ret()
```

运行结果:

```
--- in test1 func----
140212571149040
140212571149040
--- in test1 func----
```

2. 什么是闭包

```
#定义一个函数
def test(number):
```

```
3
4       #在函数内部再定义一个函数，并且这个函数用到了外边函数的变量，那么将这个函数以及用到的一些变量称之为闭包
5       def test_in(number_in):
6           print("in test_in 函数, number_in is %d"%number_in)
7           return number+number_in
8       #其实这里返回的就是闭包的结果
9       return test_in
10
11
12  #给test函数赋值，这个20就是给参数number
13  ret = test(20)
14
15  #注意这里的100其实给参数number_in
16  print(ret(100))
17
18  #注意这里的200其实给参数number_in
19  print(ret(200))
```

运行结果：

```
1  in test_in 函数, number_in is 100
2  120
3
4  in test_in 函数, number_in is 200
5  220
```

3. 看一个闭包的实际例子：

```
def line_conf(k, b):
    def line(x):
        return k*x + b
    return line

line1 = line_conf(1, 1)
line2 = line_conf(4, 5)
print(line1(5))
print(line2(5))
```

这个例子中，函数line与变量a,b构成闭包。在创建闭包的时候，我们通过line_conf的参数a,b说明了这两个变量的取值，这样，我们就确定了函数的最终形式(y = x + 1和y = 4x + 5)。我们只需要变换参数a,b，就可以获得不同的直线表达函数。由此，我们可以看到，闭包也具有提高代码可复用性的作用。

如果没有闭包，我们需要每次创建直线函数的时候同时说明a,b,x。这样，我们就需要更多的参数传递，也减少了代码的可移植性。

闭包思考:

```
1.闭包似优化了变量，原来需要类对象完成的工作，闭包也可以完成
2.由于闭包引用了外部函数的局部变量，则外部函数的局部变量没有及时释放，消耗内存
```

4. 修改外部函数中的变量

```
def counter(start=0):
    def incr():
        nonlocal start
        start += 1
        return start
    return incr

c1 = counter(5)
print(c1())
```

```
print(c1())

c2 = counter(50)
print(c2())
print(c2())

print(c1())
print(c1())

print(c2())
print(c2())
```

二、装饰器详解

装饰器是程序开发中经常会用到的一个功能，用好了装饰器，开发效率如虎添翼，所以这也是Python面试中必问的问题，但对于好多初次接触这个知识的人来讲，这个功能有点绕，自学时直接绕过去了，然后面试问到了就挂了，因为装饰器是程序开发的基础知识，这个都不会，别跟人家说你会Python, 看了下面的文章，保证你学会装饰器。

1、先明白这段代码

```
#### 第一波 ####
def foo():
    print('foo')

foo     #表示是函数
foo()   #表示执行foo函数

#### 第二波 ####
def foo():
    print('foo')

foo = lambda x: x + 1

```

```
foo()   # 执行下面的lambda表达式，而不再是原来的foo函数，因为foo这个名字被重新指向了另外一个匿名函数
```

2、需求来了

初创公司有N个业务部门，1个基础平台部门，基础平台负责提供底层的功能，如：数据库操作、redis调用、监控API等功能。业务部门使用基础功能时，只需调用基础平台提供的功能即可。如下：

```
################ 基础平台提供的功能如下 ################

def f1():
    print('f1')

def f2():
    print('f2')

def f3():
    print('f3')

def f4():
    print('f4')

################ 业务部门A 调用基础平台提供的功能 ################

f1()
f2()
f3()
f4()

################ 业务部门B 调用基础平台提供的功能 ################

f1()
f2()
f3()
```

```
f4()
```

目前公司有条不紊的进行着，但是，以前基础平台的开发人员在写代码时候没有关注验证相关的问题，即：基础平台的提供的功能可以被任何人使用。现在需要对基础平台的所有功能进行重构，为平台提供的所有功能添加验证机制，即：执行功能前，先进行验证。

老大把工作交给 Low B，他是这么做的：

> 跟每个业务部门交涉，每个业务部门自己写代码，调用基础平台的功能之前先验证。诶，这样一来基础平台就不需要做任何修改了。太棒了，有充足的时间泡妹子...

当天Low B 被开除了...

老大把工作交给 Low BB，他是这么做的：

```python
################ 基础平台提供的功能如下 ################

def f1():
    # 验证1
    # 验证2
    # 验证3
    print('f1')

def f2():
    # 验证1
    # 验证2
    # 验证3
    print('f2')

def f3():
    # 验证1
    # 验证2
    # 验证3
    print('f3')

```

```python
def f4():
    # 验证1
    # 验证2
    # 验证3
    print('f4')

################ 业务部门不变 ################
### 业务部门A 调用基础平台提供的功能###

f1()
f2()
f3()
f4()

### 业务部门B 调用基础平台提供的功能 ###

f1()
f2()
f3()
f4()
```

过了一周 Low BB 被开除了...

老大把工作交给 Low BBB，他是这么做的：

> 只对基础平台的代码进行重构，其他业务部门无需做任何修改

```python
################ 基础平台提供的功能如下 ################

def check_login():
    # 验证1
    # 验证2
    # 验证3
    pass


def f1():

```

```
    check_login()

    print('f1')

def f2():

    check_login()

    print('f2')

def f3():

    check_login()

    print('f3')

def f4():

    check_login()

    print('f4')
```

老大看了下Low BBB 的实现，嘴角漏出了一丝的欣慰的笑，语重心长的跟Low BBB聊了个天：

老大说：

写代码要遵循`开放封闭`原则，虽然在这个原则是用的面向对象开发，但是也适用于函数式编程，简单来说，它规定已经实现的功能代码不允许被修改，但可以被扩展，即：

- 封闭：已实现的功能代码块
- 开放：对扩展开放

如果将开放封闭原则应用在上述需求中，那么就不允许在函数 f1 、f2、f3、f4的内部进行修改代码，老板就给了Low BBB一个实现方案：

```
def w1(func):
```

```
    def inner():
        # 验证1
        # 验证2
        # 验证3
        func()
    return inner

@w1
def f1():
    print('f1')
@w1
def f2():
    print('f2')
@w1
def f3():
    print('f3')
@w1
def f4():
    print('f4')
```

对于上述代码，也是仅仅对基础平台的代码进行修改，就可以实现在其他人调用函数 f1 f2 f3 f4 之前都进行【验证】操作，并且其他业务部门无需做任何操作。

Low BBB心惊胆战的问了下，这段代码的内部执行原理是什么呢?

老大正要生气，突然Low BBB的手机掉到地上，恰巧屏保就是Low BBB的女友照片，老大一看一紧一抖，喜笑颜开，决定和Low BBB交个好朋友。

详细的开始讲解了：

单独以f1为例：

```python
def w1(func):
    def inner():
        # 验证1
        # 验证2
        # 验证3
        func()
    return inner

@w1
def f1():
    print('f1')
```

python解释器就会从上到下解释代码，步骤如下：

1. def w1(func): ==>将w1函数加载到内存
2. @w1

没错， 从表面上看解释器仅仅会解释这两句代码，因为函数在 没有被调用之前其内部代码不会被执行。

从表面上看解释器着实会执行这两句，但是 @w1 这一句代码里却有大文章， @函数名 是python的一种语法糖。

上例@w1内部会执行一下操作：

执行w1函数

> 执行w1函数 ，并将 @w1 下面的函数作为w1函数的参数，即：**@w1 等价于 w1(f1)** 所以，内部就会去执行：
>
> ```python
> def inner():
>  #验证 1
>  #验证 2
>  #验证 3
>  f1()      # func是参数，此时 func 等于 f1
> return inner# 返回的 inner，inner代表的是函数，非执行函数 ,其实就是将原来的 f1 函数塞进另外一个函数中
> ```

w1的返回值

> 将执行完的w1函数返回值 赋值 给@w1下面的函数的函数名f1 即将w1的返回值再重新赋值给 f1，即：

```
新f1 = def inner():
          #验证 1
          #验证 2
          #验证 3
          原来f1()
      return inner
```

> 所以，以后业务部门想要执行 f1 函数时，就会执行 新f1 函数，在新f1函数内部先执行验证，再执行原来的f1函数，然后将原来f1 函数的返回值返回给了业务调用者。

如此一来， 即执行了验证的功能，又执行了原来f1函数的内容，并将原f1函数返回值 返回给业务调用着

Low BBB 你明白了吗？要是没明白的话，我晚上去你家帮你解决吧！！！

3. 再议装饰器

```python
#定义函数：完成包裹数据
def makeBold(fn):
    def wrapped():
        return "<b>" + fn() + "</b>"
    return wrapped

#定义函数：完成包裹数据
def makeItalic(fn):
    def wrapped():
        return "<i>" + fn() + "</i>"
    return wrapped

@makeBold
def test1():
```

```
    return "hello world-1"

# test2 = makeItalic(test2)
# <i>hello world-1</i>
@makeItalic
def test2():
    return "hello world-2"

# test2 = makeBold(makeItalic(test2))
# <b><i>hello world-1</i></b>
@makeBold
@makeItalic
def test3():
    return "hello world-3"

print(test1())
print(test2())
print(test3())
```

运行结果:

```
<b>hello world-1</b>
<i>hello world-2</i>
<b><i>hello world-3</i></b>
```

4. 万能装饰器

```
from time import ctime, sleep

def timefun(func):
    def wrappedfunc(*args, **kwargs):
        print("%s called at %s"%(func.__name__, ctime()))
        func(*args, **kwargs)
    return wrappedfunc

@timefun
def foo(a, b, c):
```

```
    print(a+b+c)

foo(3,5,7)
sleep(2)
foo(2,4,9)
```

5. 装饰器示例

例1:无参数的函数

```
from time import ctime, sleep

def timefun(func):
    def wrappedfunc():
        print("%s called at %s"%(func.__name__, ctime()))
        func()
    return wrappedfunc

@timefun
def foo():
    print("I am foo")

foo()
sleep(2)
foo()
```

上面代码理解装饰器执行行为可理解成

```
foo = timefun(foo)
#foo先作为参数赋值给func后,foo接收指向timefun返回的wrappedfunc
foo()
#调用foo(),即等价调用wrappedfunc()
#内部函数wrappedfunc被引用，所以外部函数的func变量(自由变量)并没有释放
#func里保存的是原foo函数对象
```

例2:被装饰的函数有参数

```
from time import ctime, sleep

def timefun(func):
    def wrappedfunc(a, b):
        print("%s called at %s"%(func.__name__, ctime()))
        print(a, b)
        func(a, b)
    return wrappedfunc

@timefun
def foo(a, b):
    print(a+b)

foo(3,5)
sleep(2)
foo(2,4)
```

例3:被装饰的函数有不定长参数

```
from time import ctime, sleep

def timefun(func):
    def wrappedfunc(*args, **kwargs):
        print("%s called at %s"%(func.__name__, ctime()))
        func(*args, **kwargs)
    return wrappedfunc

@timefun
def foo(a, b, c):
    print(a+b+c)

foo(3,5,7)
sleep(2)
foo(2,4,9)
```

例4:装饰器中的return

```
from time import ctime, sleep

def timefun(func):
    def wrappedfunc():
        print("%s called at %s"%(func.__name__, ctime()))
        func()
    return wrappedfunc

@timefun
def foo():
    print("I am foo")

@timefun
def getInfo():
    return '----hahah---'

foo()
sleep(2)
foo()


print(getInfo())
```

执行结果:

```
foo called at Fri Nov  4 21:55:35 2016
I am foo
foo called at Fri Nov  4 21:55:37 2016
I am foo
getInfo called at Fri Nov  4 21:55:37 2016
None
```

如果修改装饰器为`return func()`，则运行结果:

```
foo called at Fri Nov  4 21:55:57 2016
I am foo
foo called at Fri Nov  4 21:55:59 2016
I am foo
getInfo called at Fri Nov  4 21:55:59 2016
----hahah---
```