Python复习3 装饰器

1. 概念

装饰器是给现有的模块增添新的小功能，可以对原函数进行功能扩展，而且还不需要修改原函数的内容，也不需要修改原函数的调用。

装饰器是一种常见的设计模式，经常被用于有切面需求的场景，较为经典的应用有插入日志、增加计时逻辑来检测性能、加入事务处理等。装饰器是解决这类问题的绝佳设计，有了装饰器，我们可以抽离出大量函数中与函数功能本身无关的代码并继续重用。一言蔽之，装饰器的作用是为已经存在的对象添加额外的功能。

2. 写法

*装饰器的本质是一个闭包。这里回忆一下闭包的三要素：闭包三要素：1. 函数的嵌套 ; 2.内层函数使用外层函数的变量 ;3.外层函数返回内层函数的引用。

```python
def get_time(func):
    def wrapper():
        func()
        print("used time:xxx")
    return wrapper
# get_time是一个装饰器

# 不使用语法糖的版本:
def foo():
    print("foo")
foo = get_time(foo)
foo()
# 使用语法糖的版本:
@get_time
def foo():
    print("foo")
foo()
```

3. 装饰器的嵌套

```
def f_a(func):
    print("f_a")

    def w_a():
        print(1)
        func()
    return w_a

def f_b(func):
    print("f_b")

    def w_b():
        print(2)
        func()
    return w_b

@f_a
@f_b
def f_c():
    print("f_c")
    print(3)

f_c()
```

改代码的输出结果：

```
f_b
f_a
1
2
f_c
3
```

要理解其中比较重要的一点：

f_b作用在f_c上，而f_a又作用在了f_b上。

```
# 不使用语法糖的版本
f_c = f_a(f_b(f_c))
```

4. 带参数装饰器

```
@decorator(args)
def func():
    pass

func = decorator(arg)(func)
```

带有参数的装饰器，其实是在装饰器函数的外面又包裹了一个函数，使用该函数接收参数，返回的是装饰器函数。

下面这个例子就是在装饰器函数decorator外面又包裹了一个函数logging，使用该函数接收参数（flag）

```
# 添加输出日志的功能
def logging(flag):
    def decorator(fn):
        def inner(num1, num2):
            if flag == "+":
                print("--正在努力加法计算--")
            elif flag == "-":
                print("--正在努力减法计算--")
            result = fn(num1, num2)
            return result

        return inner

    return decorator


@logging("+")
def add(a, b):
    result = a + b
    return result


@logging("-")
def sub(a, b):
    result = a - b
    return result


add(1, 2)
sub(1, 2)
```

5. 类装饰器

```
class Decorator:

    def __init__(self, fn):
        print("inside Decorator.__init__()")
        self.fn = fn

    def __call__(self):
        self.fn()
        print("inside Decorator.__call__()")


@Decorator
def func():
    print("inside func()")


print("Finished decorating func()")
func()
```

输出：

```
inside Decorator.__init__()
Finished decorating func()
inside func()
inside Decorator.__call__()
```