**装饰器系列(1)-你也能看懂的装饰器基础知识**

最近在python代码整洁的相关知识的时候，在次看到了装饰器的相关知识，正好在微信公众号后台正好有想伙伴在问装饰器的相关的内容，那么趁着这次机会我们来一起来看一下。

1. **什么是装饰器**

Python装饰器最早是在PEP-318中提出和引入的，具体详情大家可以参考以下链接进行了解：

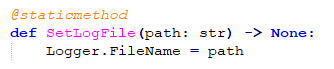
<https://sphinx-locales.github.io/peps/pep-0318/>

那么这里稍微解释一下什么是PEP，PEP是[Python](https://baike.baidu.com/item/Python?fromModule=lemma_inlink" \t "https://baike.baidu.com/item/PEP/_blank) Enhancement Proposals的缩写。一个PEP是一份为Python社区提供各种增强功能的技术规格，也是提交新特性，以便让社区指出问题，精确化技术文档的提案。

在学习装饰器之前，我们要搞清楚一件事情。在Python中函数与其他任何事物一样也是一种常规的对象，这意味着什么呢？意味着我们可以将其赋给变量，作为函数参数进行使用等操作。

在Python中装饰器是一个重要的函数，对没错它是一个函数。它一般的作用是让其他已经实现的函数在不用修改任何代码的条件下增加一些新的功能。比如说性能测试，脚本函数的运行时间，日志打印和事务处理等。简单的说：装饰器的作用就是对已经定义的函数添加额外功能。

装饰器我们在学习中是常见的，比如我们在看一些模块文件的源码的时候，会发现一个已经定义的函数的上方出现一个@的注释。没错，这就是装饰器的语法糖，比如以下的一串代码就使用了[Python](http://c.biancheng.net/python/" \t "_blank) 的函数装饰器 staticmethod。



其中@可以理解为装饰器的符号，其实上述代码如果不依赖装饰器来实现，则是这样：

**def SetLogFile(path):**

**Logger.FileName = path**

**SetLogFile= staticmethod(SetLogFile)**

可以看出，没有装饰器的时候我们可以在完成函数的定义之后，人工进行一次函数的包装和赋值的操作即可，另外可以看出，装饰器有一种函数调用函数的感觉。那么接来下我们循序渐进的进行讲解，首先来看一下函数中的函数。

1. **在函数中使用函数**

大家在平时的代码中可能不会在定义的函数中再定义一个函数，那么我们就先来看一个在函数中定义另一个函数的代码：

*# 嵌套的函数*def add\_total(name):  
 print(**'传递的参数为：',** name)  
 def add1():  
 return **'我是+1的操作'** def add2():  
 return **'我是+2的操作'** def add3():  
 return **'我是+3的操作'** print(add1())  
 print(add2())  
 print(add3())  
add\_total(**'Hello'**)

可以看出上述代码中，我们在一个函数中又定义了三个函数，即嵌套使用了函数。且add1等函数的返回结果是字符串。

当然了我们可以把返回的参数由字符串更改为返回函数，比如这样：

def add\_total(name):  
 print('传递的参数为：',name)  
 def add1():  
 return '我是add1的操作'  
 def add2():  
 return '我是add2的操作'  
 def add3():  
 return '我是add3的操作'  
 if "Hello" in name:  
 return add3()  
res = add\_total("Hello Python知识学堂")  
print(res)

输出结果为：

传递的参数为： Hello Python知识学堂

我是+3的操作

这里我们对函数add\_total()的参数进行判断，如果参数name中有“Hello”，我们就返回函数add3()。

**这里要注意的是return add3()的时候，会执行函数add3()，如果return add3的话就不会再次运行add3函数，这一点大家还是要稍微注意点的。**

既然函数也是一种常规的对象，那么将函数进行传递将发生什么呢？我们来看代码：

def add\_total():  
 print('这是总和的计算')  
  
def test\_output(func\_parameter):  
 print('输出测试')  
 print(func\_parameter())  
 return "success"  
res = test\_output(add\_total)  
print(res)

输出为：

输出测试

这是总和的计算

None

Success

上述代码中我们将函数add\_total()传递给函数test\_output(),即函数作为了参数进行了使用，**注意print(func\_parameter())这串代码输出的结果。**那么现在就存在一个问题，假设我们要对案例1中的三个函数add1(),add(),add3()都增加统计时间的代码，那应该就是这样：

import time  
  
def add\_total(name,a,b):  
 print('传递的参数为：',name)  
 def add1(a,b):  
 t1 = time.time()  
 c1 = a + b  
 t2 = time.time()  
 print("add1函数，花费的时间是：{}".format(t2-t1))  
 return '我是add1的操作'

def add2(a,b):  
 t1 = time.time()  
 c2 = a + b  
 t2 = time.time()  
 print("add2函数，花费的时间是：{}".format(t2 - t1))  
 return '我是add2的操作'

def add3(a,b):  
 t1 = time.time()  
 c3 = a + b  
 t2 = time.time()  
 print("add2函数，花费的时间是：{}".format(t2 - t1))  
 return '我是add3的操作'  
   
 add1(a, b)  
 add2(a, b)  
 add3(a, b)  
add\_total('Hello',1,3)

这个方法确实可以，但是人们在使用的时候，发现这种方式真的很复杂不够友好。假设add\_total()函数存在很多的子函数呢,那么如果我们需要继续增加一个add函数并且记录运行时间，那么我们又需要增加计算时间的重复代码，从这个角度上来说是非常难受的。因此大家发明了装饰器这个语法糖。

1. **简单装饰器**

我们先使用装饰器来对上述代码进行改写，然后再来解释：

def use\_time(func):  
 def wrapper(\*args,\*\*kwargs):  
 start\_t = time.time()  
 myf = func(\*args,\*\*kwargs)  
 cost\_time = time.time()-start\_t  
 print(f"函数 {func.\_\_name\_\_} 运行所用的时间为 {cost\_time}")  
 return myf  
 return wrapper  
  
def add\_total(name,a,b):  
 print('传递的参数为：',name)  
 @use\_time  
 def add1(a,b):  
 c1 = a + b  
 return '我是add1的操作'

@use\_time  
 def add2(a,b):  
 c2 = a + b  
 return '我是add2的操作'

@use\_time  
 def add3(a,b):  
 c3 = a + b  
 return '我是add3的操作'  
 add1(a, b)  
 add2(a, b)  
 add3(a, b)  
  
add\_total('Hello',1,3)

通过上述代码我们就能够很好的解决了上述的痛点，这解释一下其中的代码含义：

（1）@use\_time表示一个装饰器。

（2）@符号就是装饰器的语法糖，我们就可以省去add1= use\_time(add1)这句赋值。

（3）use\_time()返回一个名字叫 wrapper的函数，也可以使用其他名称,但是建议使用这个名称，规范一些。

在上面的代码中，use\_time装饰器接受待装饰函数func作为参数，这个不难理解。另外我们在函数内部定义了一个名为wrapper的函数。一般来说我们会将这个wrapper叫做包装函数，此处案例中我们的包装函数接受任意数目的不定(可变)参数(\*args,\*\*kwargs)，这个包装函数通过调用接受到的func进行进一步的工作。包装函数将作为装饰器的返回值，替换了被我们传入的func函数。

当我们运行到add1(a,b)的时候，解释器发现其使用了装饰器，便将函数add1()作为参数传递给use\_time(func)函数。之后便运行函数myf = func(\*args,\*\*kwargs)，此时这个函数就相当于执行add1(a,b)并返回“我是add1的操作”；运行到add2(a,b)的时候也会同样执行上述的操作...

可以看出通过装饰器，添加上计算时间的新函数替换了原来的add1函数、add2函数和add3函数，但仍通过add1()等函数名调用，所以看起来就是add1等函数被“装饰”了。

1. **函数装饰器**

在可装饰的对象中，可能函数是最为简单的对象了。可以使用装饰器将各种常见的操作：验证参数，条件检查，重复尝试，修改签名等等。

上述第三节的内容就是函数装饰器的相关的案例，总体上还是很好的阐述了函数装饰器的相关细节。比如我们这次在创建一个案例，实现多次进行尝试打开一个网站的一个装饰器，我们一起来看一下：

*# coding = utf-8*import requests  
def retry(func):  
 def wrapper(\*args, \*\*kwargs):  
 trytime = 3  
 while trytime > 0:  
 try:  
 print("执行的函数名称为：",func.\_\_name\_\_)  
 return func(\*args, \*\*kwargs)  
 except Exception as e:  
 print("Exception ", repr(e))  
 trytime -= 1  
 if trytime <=0:  
 print('此次打开出错：error')  
 return wrapper

@retry  
def test\_url(url):  
 res = requests.get(url, timeout=1)  
 data = res.json()  
 return data  
url = "http://www.baidudududdu.com"  
test\_url(url)

本次我们传入了一个不存在的url,输出的结果如下：



可以看出我们的程序一共尝试了3次的请求网站的操作，大家可以根据这段代码继续进行改**进，比如加入一个参数或者使用多个参数，那么怎么实现呢？不着急我们将在下期的文章中讨论。**

1. **类装饰器**

函数装饰器的基本知识相信大家都看明白了，接下来我们一起看一下类装饰器的相关知识。其实也就是将之前装饰函数改为装饰类就行了，我们先写一个装饰器，然后再来进行相关知识点的解释。

在写代码之前，先简单的举个例子介绍一下\_\_call\_\_函数：

class Subject(object):  
 def \_\_init\_\_(self,name):  
 self.name=name  
 def \_\_call\_\_(self):  
 print("hello "+ self.name)  
  
a = Subject('Python')  
*# 1.第一种方式调用*print(a.\_\_call\_\_())  
*# 2.第二种方式调用*print(a())

**输出为：**

hello Python

None

hello Python

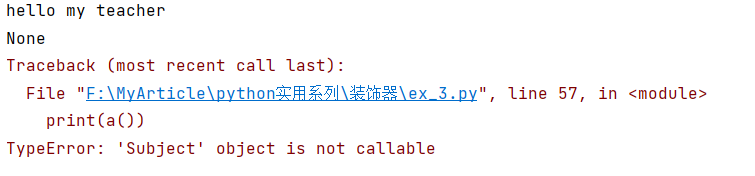
None

这两种效果是一致的，可以看出：

类中的\_\_call\_\_()方法其实是把一个类的实例化对象变成了可调用对象，也就是说把一个类的实例化对象变成了可调用对象，如当类里没有实现\_\_call\_\_()时，此时的对象只是个类的实例，不是一个可调用的对象，当调用它时会报错：‘Subject’ object is not callable，比如这样：

class Subject(object):  
 def \_\_init\_\_(self,name):  
 self.name=name  
 *# def \_\_call\_\_(self):  
 # print("hello "+ self.name)* def teacher(self):  
 print("hello my teacher")  
  
a = Subject('Python')  
*# 1.第一种方式调用*print(a.teacher())  
*# 2.第二种方式调用*print(a())

此时的输出为：



当然了，这个\_\_call\_\_函数是可以带参数的，这里我们就不举例了，接下来我们来看一下类装饰器的案例：

class truck(object):  
 def \_\_init\_\_(self,func):  
 self.func = func  
 print('这是一个初始化的操作')  
 *# call函数* def \_\_call\_\_(self):  
 *# 装饰函数*

print('[INFO]:the function {}() is running...'.format(self.func.\_\_name\_\_))print('回家驾驶卡车')  
 self.func()  
@truck  
def smallcar():  
 print(f'Tony 的 mini car')  
smallcar()

**输出如下：**

这是一个初始化的操作

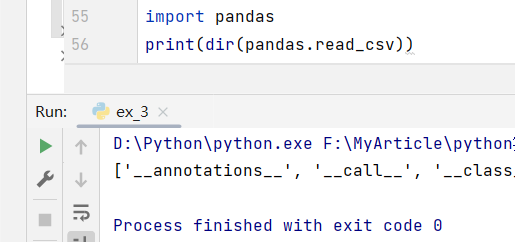
回家驾驶卡车

[INFO]:the function smallcar() is running...

Tony 的 mini car

整体上还会很好理解的，这里说明几点：

1. 只要类装饰器来装饰函数的时候，默认会调用类的\_\_init\_\_方法。
2. 类中实现了\_\_call\_\_方法，实例化的对象可以直接调用。
3. 在python中只要是函数，他就自动实现了\_\_call\_\_方法，可以使用dir进行查看：



当然了，以上就是一些基础的知识，下次的文章中我们来聊聊在装饰器中怎么使用参数，即有参装饰器的相关知识。

1. **参考文档：**

（1）《流畅的Python》-Luciano Ramalho

（2）<https://www.liaoxuefeng.com/wiki/1016959663602400/1017451662295584>

（3）https://www.cnblogs.com/tobyqin/p/python-decorator.html

https://blog.csdn.net/tryhardsilently/article/details/90767627

<https://www.runoob.com/w3cnote/python-func-decorators.html>

<https://www.zhihu.com/question/26930016/answer/1047233982>

**<https://zhuanlan.zhihu.com/p/51158386>**

<https://zhuanlan.zhihu.com/p/269012332>

<https://foofish.net/python-decorator.html>

https://blog.csdn.net/weixin\_33711647/article/details/92549215?spm=1001.2101.3001.6650.3&utm\_medium=distribute.pc\_relevant.none-task-blog-2%7Edefault%7ECTRLIST%7ERate-3-92549215-blog-126249772.t0\_edu\_mix&depth\_1-utm\_source=distribute.pc\_relevant.none-task-blog-2%7Edefault%7ECTRLIST%7ERate-3-92549215-blog-126249772.t0\_edu\_mix&utm\_relevant\_index=6

https://www.jb51.net/article/231486.htm

**@wraps(func)**

https://zhuanlan.zhihu.com/p/150839678