

一周一个Python语法糖：（一）装饰器



寒夏凉秋 (/u/afd9c2bdd131) [+ 关注](#)

0.2 2017.04.03 20:10* 字数 1421 阅读 2929 评论 3 喜欢 8

(/u/afd9c2bdd131)

Decorator

首先，我们来认识一下**装饰器**是什么：

装饰器是给现有的模块增添新的小功能

（在不改变原有模块功能的基础上）

假如我有个简单笔，它只能用一种颜色进行写字
我现在给它加上一只笔芯，它能换种颜色写字（又能换回来~）
这就是装饰器的朴素比喻

一、初探装饰器

手动写个装饰器吧

```
#自定义装饰函数
def decorator(fn):
    def wrapper(*args):
        #这里装饰器的作用是在函数调用前增加一句话表示装饰成功

        print("this is decorator fo %s" % fn.__name__)
        fn(*args)
    return wrapper
def hello(name):
    print('hello,%s' %name)
if __name__=="__main__":
    #用赋值的形式进行装饰器
    hello=decorator(hello)
    hello("cool")
```

输出结果为：

```
this is decorator fo hello  
  
hello,cool
```

Paste_Image.png

首先，我们要知道，在python中，函数也是一种**对象**（万物皆对象！）

1. 函数可以赋值给一个变量（学过C语言的可以联想下函数指针）
2. 函数可以定义在另一个函数内部

这也意味着一个函数可以返回另一个函数

```
hello=decorator(hello)
```

这一句代码中，将**hello**函数作为**变量**传入**decorator装饰器**中，然后hello方法在decorator中的函数**wrapper函数实现**，同时包装新的功能，将**新的函数wrapper**作为变量返回，所以hello的新值是 经过decorator装饰的wrapper新方法。

所以，装饰器装饰函数的时候，将函数作为变量传入装饰器内部，实际调用的是装饰器内部的函数（添加新功能之后的函数）

二、@语法糖

Python 中装饰器语法并不用每次都赋值语句。

在函数定义的时候就加上@+装饰器名字即可

再来我们刚才的例子吧：

```
def decorator(fn):  
    def wrapper(*args):  
        print("this is decorator fo %s" % fn.__name__)  
        fn(*args)  
    return wrapper  
@decorator  
def hello(name):  
    print('hello,%s' %name)  
if __name__=="__main__":  
    hello("cool")
```

2.装饰器的顺序:

比如我们有两个装饰器:

```
@decorator_one
@decorator_two
def hello()
    pass
```

这句代码实际上类似于:

```
hello=decorator_one(decorator_two(hello))
```

两个装饰器一层层地往外装饰

3.带参数的装饰器

我们说过，装饰器其实也是一种函数，所以它自身也是能带参数的

```
@decorator(arg1, arg2)
def func():
    pass
```

类似于

```
func = decorator(arg1,arg2)(func)
```

来个实际点的例子吧:

我们手写html的时候需要各种补全（那个用编辑器的当然爽得飞起！）

但是，如果是在python中用字符串去表示html标签的时候，就~坑爹了

。总不能每个标签我都写一个方法吧

最方便的方法，写一个带参数的装饰器！

HTML.py

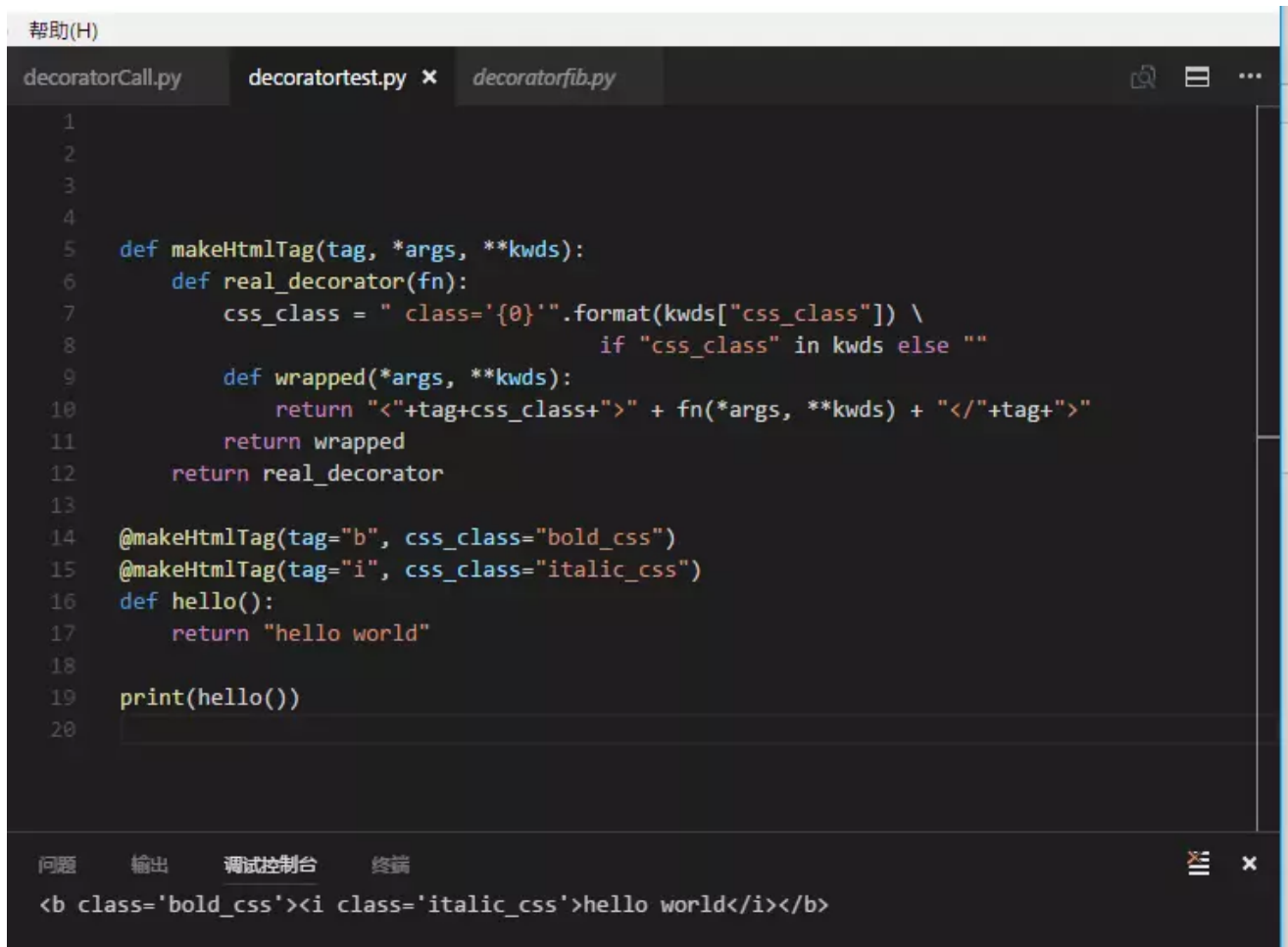
```
def makeHtmlTag(tag, *args, **kwargs):
    def real_decorator(fn):
        css_class = " class='{0}'".format(kwargs["css_class"]) \
            if "css_class" in kwargs else ""
        def wrapped(*args, **kwargs):
            return "<"+tag+css_class+">" + fn(*args, **kwargs) + "</"+tag+">"
        return wrapped
    return real_decorator

@makeHtmlTag(tag="b", css_class="bold_css")
@makeHtmlTag(tag="i", css_class="italic_css")
def hello():
    return "hello world"

print(hello())

print(hello())
```

运行结果：



```
def makeHtmlTag(tag, *args, **kwargs):
    def real_decorator(fn):
        css_class = " class='{0}'".format(kwargs["css_class"]) \
            if "css_class" in kwargs else ""
        def wrapped(*args, **kwargs):
            return "<"+tag+css_class+">" + fn(*args, **kwargs) + "</"+tag+">"
        return wrapped
    return real_decorator

@makeHtmlTag(tag="b", css_class="bold_css")
@makeHtmlTag(tag="i", css_class="italic_css")
def hello():
    return "hello world"

print(hello())

print(hello())
```

问题 输出 调试控制台 终端

<b class='bold_css'><i class='italic_css'>hello world</i>

Paste_Image.png

关于几点说明：

(1) 在装饰器makeHtmlTag中，*args代表了参数元组，假如你传入的参数分别是1, 2, 3, 4, 则args= (1, 2, 3, 4)

****kwargs**则参数字典，返回一个key为参数变量名，value为变量值的字典

这样我们就可以很方便地不用改动函数本身去获取函数传递的参数并进行装饰了

（2）使用装饰器的时候有个缺陷就是不能在过程中更改某个装饰器的参数值（比如该例子中 hello 的便签就永远是b,i了）

如果你觉得这样写太！麻！烦！了！什！么！鬼！

为什么我要在函数体中再定义一个函数体！！！！

难道还要我一层层剥开你的心吗？

4. 用类的方式去写一个装饰器

```
class makeHtmlTagClass(object):

    def __init__(self, tag, css_class=""):
        self._tag = tag
        self._css_class = " class='{0}'".format(css_class) \
            if css_class != "" else ""

    def __call__(self, fn):
        def wrapped(*args, **kwargs):
            return "<" + self._tag + self._css_class + ">" \
                + fn(*args, **kwargs) + "</" + self._tag + ">"
        return wrapped

@makeHtmlTagClass(tag="b", css_class="bold_css")
@makeHtmlTagClass(tag="i", css_class="italic_css")
def hello(name):
    return "Hello, {}".format(name)

print hello("Hao Chen")
```

关于说明：

（1）我们将整个类作为一个装饰器，工作流程：

通过__init__（）方法初始化类

通过__call__（）方法调用真正的装饰方法

（2）当装饰器有参数的时候，init() 成员就不能传入fn了，而fn是在call的时候传入的。

（fn代表要装饰的函数）

decorator万能的？

No!No!No!

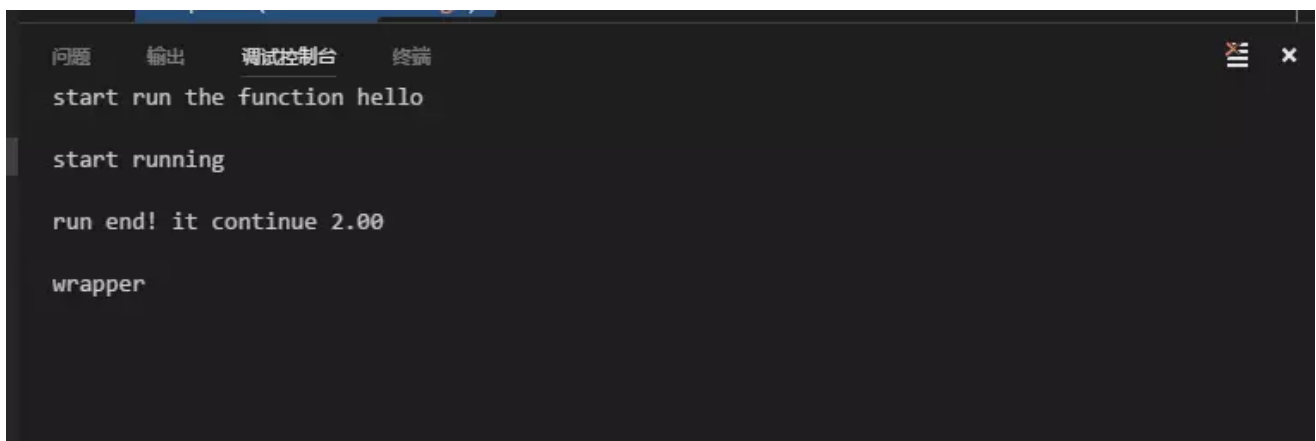
有时候我想加入日志系统。来记录我某个函数的运行情况。

```
import time
def logger(fn):
    def wrapper(*args,**kwargs):
        ts=time.time()
        print('start run the function %s' % fn.__name__)
        result=fn(*args,**kwargs)
        te=time.time()
        print('run end! it continue %.2f'%(te-ts))
        return result
    return wrapper

@logger
def hello():
    print('start running')
    time.sleep(2)
    return 2

hello()
print(hello.__name__)
```

运行结果：



Paste_Image.png

W T F?

我的hello.__name__不是应该是hello吗？

唯一解释：就想一开始说的，装饰器原理：

```
hello=decorator(hello)
```

hello实际上已经变成了经过装饰器修饰的方法了

（主公，我身在曹营心在汉呀！！！！）

怎么办？

Python的functool包中提供了一个叫wrap的decorator来消除这样的副作用

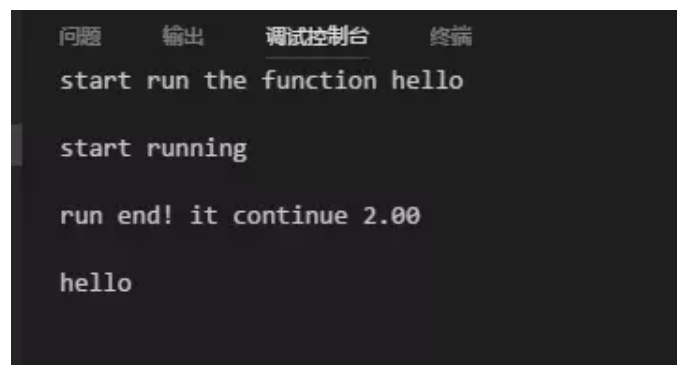
新版logger.py

```
from functools import wraps
import time
def logger(fn):
    @wraps(fn)
    def wrapper(*args,**kwargs):
        ts=time.time()
        print('start run the function %s' % fn.__name__)
        result=fn(*args,**kwargs)
        te=time.time()
        print('run end! it continue %.2f'%(te-ts))
        return result
    return wrapper

@logger
def hello():
    print('start running')
    time.sleep(2)
    return 2

hello()
print(hello.__name__)
```

运行结果：



Paste_Image.png

5.装饰器获取参数的值

比如我某个函数是批量运行的，我需要加个日志系统来知道这个函数进行了多少次，，这就需要获取函数运行时的参数了

方法：用inspect模块的getcallargs方法去获取原函数的参数

返回的是一个字典，根据字典的key去获取参数的值

```
from functools import wraps
from inspect import getcallargs
class Logger(object):
    def __init__(self,filename=''):
        if filename!='':
            self._filename=filename
        else:
            self._filename="log/errorlog"
    def __call__(self,fn):
        @wraps(fn)
        def wrapper(*args,**kwargs):
            func_args=getcallargs(fn,*args,**kwargs)
            if 'param1' in func_args.keys():
                param1=func_args['param1']
            if 'param2' in func_args.keys():
                param2=func_args['param2']
            with open(self._filename,'a') as logfile_handle:
                logfile_handle.write(param1+'/'+param2+'\t finished\n')
                logfile_handle.flush()
            result=fn(*args,**kwargs)
            return result
        return wrapper

@Logger(filename='testlog')
def test(param1,param2,param3,param4,param5):
    print('Ok,finished')
    return param5

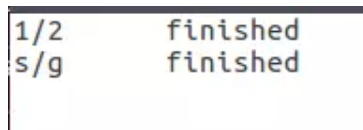
if __name__=="__main__":
    print(test('1','2','3','4','5'))
    print(test('s','g','r','e','b'))
```

运行结果：



2017-04-03 19-29-07屏幕截图.png

打开日志文件：



2017-04-03 19-29-56屏幕截图.png

我们拿到了函数的第一个参数跟第二个参数的值并保存到文件中

一些装饰器的例子（日志那个请看上文）

```
from functools import wraps

def memo(fn):
    cache = {}
    miss = object()

    @wraps(fn)
    def wrapper(*args):
        result = cache.get(args, miss)
        if result is miss:
            result = fn(*args)
            cache[args] = result
        return result

    return wrapper

@memo
def fib(n):
    if n < 2:
        return n
    return fib(n - 1) + fib(n - 2)
```

这是斐波拉契数列的递归算法。

因为fib (1) fib (0) 是重复计算的值，将其利用装饰器预先缓存（先计算好fib (1) , fib (0) ），调用的时候直接返回字典的值，达到优化fib (n) 的思路

很经典

当我读懂这段代码的时候，

我内心大喊一声：**WO CAO! 还能这样玩!**

2.web后端通过URL的路由来调用相关注册

```
class MyApp():
    def __init__(self):
        self.func_map = {}

    def register(self, name):
        def func_wrapper(func):
            self.func_map[name] = func
            return func
        return func_wrapper

    def call_method(self, name=None):
        func = self.func_map.get(name, None)
        if func is None:
            raise Exception("No function registered against - " + str(name))
        return func()

app = MyApp()

@app.register('/')
def main_page_func():
    return "This is the main page."

@app.register('/next_page')
def next_page_func():
    return "This is the next page."

print app.call_method('/')
print app.call_method('/next_page')
```

注：decorator类中没有`call()`，但是wrapper返回了原函数。所以，原函数没有发生任何变化。

这例子只是用来注册url 方法并调用
防止web访问位置url~

无返回值的异步多线程调用（涉及数据变化请用锁）

```
from threading import Thread
from functools import wraps

def async(func):
    @wraps(func)
    def async_func(*args, **kwargs):
        func_h1 = Thread(target = func, args = args, kwargs = kwargs)
        func_h1.start()
        return func_h1

    return async_func

if __name__ == '__main__':
    from time import sleep

    @async
    def print_somedata():
        print 'starting print_somedata'
        sleep(2)
        print 'print_somedata: 2 sec passed'
        sleep(2)
        print 'print_somedata: 2 sec passed'
        sleep(2)
        print 'finished print_somedata'

    def main():
        print_somedata()
        print 'back in main'
        print_somedata()
        print 'back in main'

    main()
```