

装饰器

在Python这个国家里，装饰器以及后面讲到的迭代器，生成器都是十二分重要的高级函数。
如果将装饰器比作取经路上的一个大boss，那么想干掉它必须拿到三件法宝

法宝一（作用域）：

法宝二（函数即对象）：

在python的世界里，函数和我们之前的[1,2,3], 'abc', 8等一样都是对象，而且**函数是最高级的对象**（对象是类的实例化，可以调用相应的方法，函数是包含变量对象的对象，牛逼！）。

```
1 def foo():
2     print('i am the foo')
3     bar()
4
5 def bar():
6     print('i am the bar')
7
8 foo()
9 # def bar():      #报错
10 #     print('i am the bar')
```

带着这个问题，我们聊一聊函数在内存的存储情况：

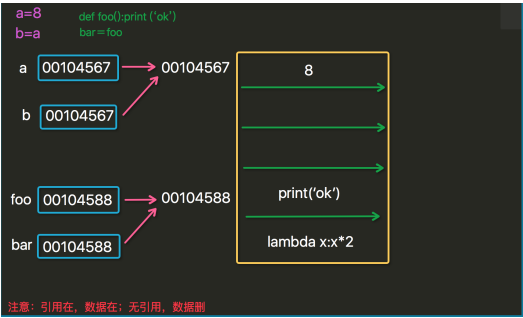


图1

函数对象的调用仅仅比其它对象多了一个()而已！foo, bar与a, b一样都是个变量名。
那上面的问题也就解决了，只有函数加载到内存才可以被调用。
既然函数是对象，那么自然满足下面两个条件：

1. 其可以被赋给其他变量

```
1 def foo():
2     print('foo')
3 bar=foo
4 bar()
5 foo()
6 print(id(foo), id(bar))  #4321123592 4321123592
```

2. 其可以被定义在另外一个函数内(作为参数&作为返回值), 类似于整形，字符串等对象。

```
1 #*****函数名作为参数*****
2 def foo(func):
3     print('foo')
```

```

4         func()
5
6     def bar():
7         print('bar')
8
9     foo(bar)
10
11     #*****函数名作为返回值*****
12
13     def foo():
14         print('foo')
15         return bar
16
17     def bar():
18         print('bar')
19
20     b=foo()
21     b()

```

注意：这里说的函数都是指函数名，比如foo；而foo()已经执行函数了，foo()是什么类型取决于return的内容是什么类型!!!

另外，如果大家理解不了对象，那么就将函数理解成变量，因为函数对象总会由一个或多个变量引用，比如foo, bar。

法宝三（函数的嵌套以及闭包）：

抛一个小问题：

```

1     def foo():
2         print('foo')
3         def bar():
4             print('bar')
5             # bar()
6         bar()

```

是的，bar就是一个变量名，有自己的作用域的。

Python允许创建嵌套函数。通过在函数内部def的关键字再声明一个函数即为嵌套：

```

1     #想执行inner函数,两种方法
2     def outer():
3         x = 1
4         def inner():
5             print (x) # 1
6             # inner() # 2
7             return inner
8
9     # outer()
10    in_func=outer()
11    in_func()

```

在这里，你有没有什么疑问？如果没有，那我问你：

1 两种调用方式有区别吗，不都是在外边调用inner吗？

```

1     in_func=outer()
2     in_func()
3     #####
4     inner()(已经加载到内存啦)

```

2

```

1     def outer():
2         x=1    #函数outer执行完毕即被销毁
3         print(x)

```

既然如此，i()执行的时候outer函数已经执行完了，为什么inner还可以调用outer里的变量x呢？

哈，这就涉及到我们叫讲的闭包啦！

因为：outer里return的inner是一个闭包函数，有x这个环境变量。

OK，那么什么是闭包呢？

闭包(closure)是函数式编程的重要的语法结构。

定义：如果在一个内部函数里，对外部作用域（但不是在全局作用域）的变量进行引用，那么内部函数就被认为是闭包(closure)。

如上实例，inner就是内部函数，inner里引用了外部作用域的变量x（x在外部作用域outer里面，不是全局作用域），

则这个内部函数inner就是一个闭包。

再稍微讲究一点的解释是，闭包=函数块+定义函数时的环境，inner就是函数块，x就是环境，当然这个环境可以有很多，不止一个简单的x。

```
1 | print(in_func.__closure__[0].cell_contents)
```

用途省略

| |
|-----|
| 用途1 |
| 用途2 |

装饰器概念

说了这么多，终于到了我们的装饰器了。

装饰器本质上是一个函数，该函数用来处理其他函数，它可以让其他函数在不需要修改代码的前提下增加额外的功能，装饰器的返回值也是一个函数对象。它经常用于有切面需求的场景，比如：插入日志、性能测试、事务处理、缓存、权限校验等应用场景。装饰器是解决这类问题的绝佳设计，有了装饰器，我们就可以抽离出大量与函数功能本身无关的雷同代码并继续重用。概括的讲，装饰器的作用就是为已经存在的对象添加额外的功能。

业务生产中大量调用的函数：

```
1 | def foo():
2 |     print('hello foo')
3 | foo()
```

现在有一个新的需求，希望可以记录下函数的执行时间，于是在代码中添加日志代码：

```
1 | import time
2 | def foo():
3 |     start_time=time.time()
4 |     print('hello foo')
5 |     time.sleep(3)
6 |     end_time=time.time()
7 |     print('spend %s'%(end_time-start_time))
8 |
9 | foo()
```

bar()、bar2()也有类似的需求，怎么做？再在bar函数里调用时间函数？这样就造成大量雷同的代码，为了减少重复写代码，我们可以这样做，重新定义一个函数：专门设定时间：

```
1 | import time
2 | def show_time(func):
3 |     start_time=time.time()
4 |     func()
5 |     end_time=time.time()
6 |     print('spend %s'%(end_time-start_time))
7 |
8 |
9 | def foo():
10 |     print('hello foo')
11 |     time.sleep(3)
```

```

12
13 show_time(foo)

```

逻辑上不难理解，而且运行正常。但是这样的话，你基础平台的函数修改了名字，容易被业务线的人投诉的，因为我们每次都要将一个函数作为参数传递给show_time函数。而且这种方式已经破坏了原有的代码逻辑结构，之前执行业务逻辑时，执行运行foo()，但是现在不得不改成show_time(foo)。那么有没有更好的方式的呢？当然有，答案就是装饰器。

简单装饰器

if `foo()==show_time(foo)` :问题解决!

所以，我们需要show_time(foo)返回一个函数对象，而这个函数对象内则是核心业务函数:执行func()与装饰函数时间计算，修改如下：

```

1 import time
2
3 def show_time(func):
4     def wrapper():
5         start_time=time.time()
6         func()
7         end_time=time.time()
8         print('spend %s'%(end_time-start_time))
9
10    return wrapper
11
12
13 def foo():
14     print('hello foo')
15     time.sleep(3)
16
17 foo=show_time(foo)
18 foo()

```

函数show_time就是装饰器，它把真正的业务方法func包裹在函数里面，看起来像foo被上下时间函数装饰了。在这个例子中，函数进入和退出时，被称为一个横切面(Aspect)，这种编程方式被称为面向切面的编程(Aspect-Oriented Programming)。

@符号是装饰器的语法糖，在定义函数的时候使用，避免再一次赋值操作

```

1 import time
2
3 def show_time(func):
4     def wrapper():
5         start_time=time.time()
6         func()
7         end_time=time.time()
8         print('spend %s'%(end_time-start_time))
9
10    return wrapper
11
12 @show_time #foo=show_time(foo)
13 def foo():
14     print('hello foo')
15     time.sleep(3)
16
17
18 @show_time #bar=show_time(bar)
19 def bar():
20     print('in the bar')
21     time.sleep(2)
22
23 foo()
24 print('*****')
25 bar()

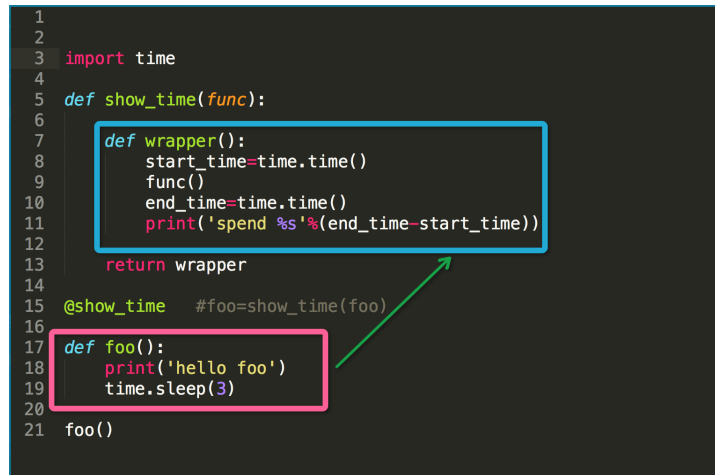
```

如上所示，这样我们就可以省去bar = show_time(bar)这一句了，直接调用bar()即可得到想要的结果。如果我们有其他的类似函数，我们可以继续调用装饰器来修饰函数，而不用重复修改函数或者增加新的封装。这样，我

们就提高了程序的可重复利用性，并增加了程序的可读性。

这里需要注意的问题：`foo=show_time(foo)`其实是把wrapper引用的对象引用给了foo，而wrapper里的变量func之所以可以用，就是因为wrapper是一个闭包函数。

key:



```

1
2
3 import time
4
5 def show_time(func):
6     def wrapper():
7         start_time=time.time()
8         func()
9         end_time=time.time()
10        print('spend %s'%(end_time-start_time))
11    return wrapper
12
13 @show_time #foo=show_time(foo)
14
15 def foo():
16     print('hello foo')
17     time.sleep(3)
18
19 foo()

```

@show_time帮我们做的事情就是当我们执行业务逻辑foo()时，执行的代码由粉框部分转到蓝框部分，仅此而已！

装饰器在Python使用如此方便都要归因于Python的函数能像普通的对象一样能作为参数传递给其他函数，可以被赋值给其他变量，可以作为返回值，可以被定义在另外一个函数内。

带参数的被装饰函数

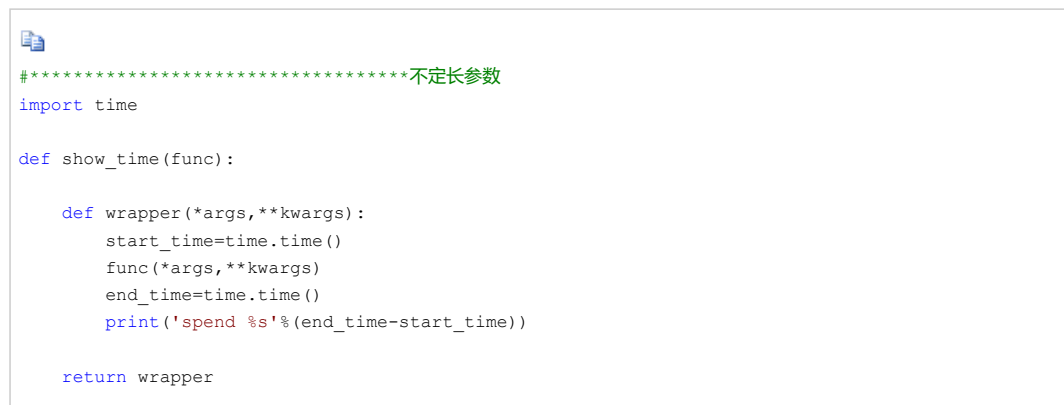
```

1 import time
2
3 def show_time(func):
4
5     def wrapper(a,b):
6         start_time=time.time()
7         func(a,b)
8         end_time=time.time()
9         print('spend %s'%(end_time-start_time))
10
11    return wrapper
12
13 @show_time #add=show_time(add)
14 def add(a,b):
15
16     time.sleep(1)
17     print(a+b)
18
19 add(2,4)

```

注意

不定长参数



```

*****不定长参数
import time

def show_time(func):

    def wrapper(*args,**kwargs):
        start_time=time.time()
        func(*args,**kwargs)
        end_time=time.time()
        print('spend %s'%(end_time-start_time))

    return wrapper

```

```

@show_time    #add=show_time(add)
def add(*args,**kwargs):

    time.sleep(1)
    sum=0
    for i in args:
        sum+=i
    print(sum)

add(2,4,8,9)

```

带参数的装饰器

装饰器还有更大的灵活性，例如带参数的装饰器：在上面的装饰器调用中，比如@show_time，该装饰器唯一的参数就是执行业务的函数。装饰器的语法允许我们在调用时，提供其它参数，比如@decorator(a)。这样，就为装饰器的编写和使用提供了更大的灵活性。

```

1  import time
2
3  def time_logger(flag=0):
4
5      def show_time(func):
6
7          def wrapper(*args,**kwargs):
8              start_time=time.time()
9              func(*args,**kwargs)
10             end_time=time.time()
11             print('spend %s'%(end_time-start_time))
12
13             if flag:
14                 print('将这个操作的时间记录到日志中')
15
16             return wrapper
17
18         return show_time
19
20
21 @time_logger(3)
22 def add(*args,**kwargs):
23     time.sleep(1)
24     sum=0
25     for i in args:
26         sum+=i
27     print(sum)
28
29 add(2,7,5)

```

@time_logger(3) 做了两件事：

- (1) time_logger(3): 得到**闭包函数**show_time，里面保存环境变量flag
- (2) @show_time : add = show_time(add)

上面的time_logger是允许带参数的装饰器。它实际上是对原有装饰器的一个函数封装，并返回一个装饰器(一个含有参数的闭包函数)。当我们使用@time_logger(3)调用的时候，Python能够发现这一层的封装，并把参数传递到装饰器的环境中。

多层装饰器

```

1  def makebold(fn):
2      def wrapper():
3          return "<b>" + fn() + "</b>"
4      return wrapper
5
6  def makeitalic(fn):
7      def wrapper():
8          return "<i>" + fn() + "</i>"

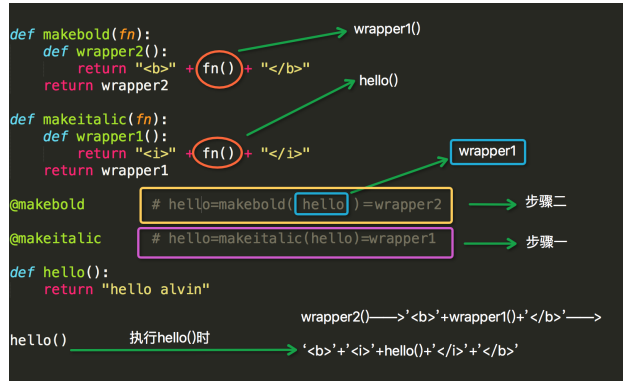
```

```

9         return wrapper
10
11 @makebold
12 @makeitalic
13 def hello():
14     return "hello alvin"
15
16 hello()

```

过程:



类装饰器

再来看看类装饰器，相比函数装饰器，类装饰器具有灵活度大、高内聚、封装性等优点。使用类装饰器还可以依靠类内部的`__call__`方法，当使用`@`形式将装饰器附加到函数上时，就会调用此方法。

```

import time

class Foo(object):
    def __init__(self, func):
        self._func = func

    def __call__(self):
        start_time=time.time()
        self._func()
        end_time=time.time()
        print('spend %s'%(end_time-start_time))

@Foo #bar=Foo(bar)

def bar():

    print('bar')
    time.sleep(2)

bar() #bar=Foo(bar)()>>>>>>没有嵌套关系了,直接active Foo的 __call__方法

```

functools.wraps

使用装饰器极大地复用了代码，但是他有一个缺点就是原函数的元信息不见了，比如函数的docstring、`__name__`、参数列表，先看例子：

```

def foo():
    print("hello foo")

print(foo.__name__)
#####

def logged(func):
    def wrapper(*args, **kwargs):

        print(func.__name__ + " was called")
        return func(*args, **kwargs)

```

```

    return wrapper

@logged
def cal(x):
    return x + x * x

print(cal.__name__)

#####
# foo
# wrapper

```

解释:

```

1 | @logged
2 | def f(x):
3 |     return x + x * x

```

等价于:

```

1 | def f(x):
2 |     return x + x * x
3 | f = logged(f)

```

不难发现，函数f被wrapper取代了，当然它的docstring，__name__就是变成了wrapper函数的信息了。

```

1 | print f.__name__    # prints 'wrapper'
2 | print f.__doc__    # prints None

```

这个问题就比较严重的，好在我们有functools.wraps，wraps本身也是一个装饰器，它能把原函数的元信息拷贝到

```

1 | from functools import wraps
2 |
3 |
4 | def logged(func):
5 |
6 |     @wraps(func)
7 |
8 |     def wrapper(*args, **kwargs):
9 |         print (func.__name__ + " was called")
10 |         return func(*args, **kwargs)
11 |     return wrapper
12 |
13 | @logged
14 | def cal(x):
15 |     return x + x * x
16 |
17 | print(cal.__name__) #cal

```

内置装饰器

@staticmethod

@classmethod

@property

学习类的时候我们详细介绍的...

补充

迭代函数被装饰

<http://stackoverflow.com/questions/739654/how-can-i-make-a-chain-of-function-decorators-in-python/159>

好文要顶

已关注

收藏该文

Yuan先生

关注 - 1

粉丝 - 3927

12

0

我在关注他 取消关注

posted @ 2016-09-01 15:16 Yuan先生 阅读(7731) 评论(3) 编辑 收藏

Post Comment

- #1楼 2017-10-18 14:38 | 公众号python学习开发

好文章

回复 引用

支持(0) 反对(0)
- #2楼 2017-11-20 18:01 | 克赛

全程跟着老男孩

回复 引用

支持(1) 反对(0)
- #3楼 2018-02-07 10:38 | Lilinpei

顶

回复 引用

支持(0) 反对(0)

刷新评论 刷新页面 返回顶部

- 【推荐】了不起的开发者，势不可挡的华为，园子里的品牌专区
- 【推荐】有道智云周年庆，API服务大放送，注册即送100元体验金！
- 【推荐】超50万行VC++源码：大型组态工控、电力仿真CAD与GIS源码库
- 【推荐】精品问答：大数据计算技术 1000 问



相关博文：

- 装饰器
 - 装饰器
 - 装饰器
 - 装饰器
 - 装饰器？装饰器！技能+10
- » 更多推荐...

最新 IT 新闻：

- 造车新势力2020生死局：同一起跑线 迥异结局
 - 华为和苹果狭路相逢 为何VR不亮AR亮？
 - 蚂蚁1.4万亿造富盛宴：员工能在其中收获多少财富？
 - 迷失的120万TikTok印度网红 YouTube能接盘吗？
 - Twitter第二季度净亏损12.28亿美元 同比转亏
- » 更多新闻...