Python 编程 计算机科学 Python 入门 装饰器

关注者 1,944 被浏览 350,925

如何理解Python装饰器?

尽量有中文的资料，浅显一些，好理解的，谢谢

关注问题 写回答 邀请回答 好问题 16 2 条评论 分享 ···

67 个回答

默认排序

刘志军
来公众号"Python之禅"免费领干货

2,601 人赞同了该回答

先来个形象比方

内裤可以用来遮羞，但是到了冬天它没法为我们防风御寒，聪明的人们发明了长裤，有了长裤后宝宝再也不冷了，装饰器就像我们这里说的长裤，在不影响内裤作用的前提下，给我们的身子提供了保暖的功效。

再回到我们的主题

装饰器本质上是一个Python函数，它可以让其他函数在不需要做任何代码变动的前提下增加额外功能，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。

先来看一个简单例子：

```
def foo():
    print('i am foo')
```

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

```
def foo():
    print('i am foo')
    logging.info("foo is running")
```

bar()、bar2()也有类似的需求，怎么做？再写一个logging在bar函数里？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：专门处理日志 ，日志处理完之后再执行真正的业务代码

```
def use_logging(func):
    logging.warn("%s is running" % func.__name__)
    func()

def bar():
    print('i am bar')

use_logging(bar)
```


相关问题

3.6.4IDLE无法输入中文？ 5 个回答

英文很差，究竟能不能学编程？ 8 个回答

设计出一种中文编程语言可行吗？为什么现有的编程语言全是英文的？ 7 个回答

学python必须会英文吗？ 19 个回答

相关推荐

Abaqus GUI 程序开发指南（Python 语言）
58 人读过 阅读

数据结构——Python 语言描述
54 人读过 阅读

数据结构和算法（Python 和 C++ 语言描述）
97 人读过 阅读

逻辑上不难理解， 但是这样的话，我们每次都要将一个函数作为参数传递给use_logging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bar()，但是现在不得不改成use_logging(bar)。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

```
def use_logging(func):

    def wrapper(*args, **kwargs):
        logging.warn("%s is running" % func.__name__)
        return func(*args, **kwargs)
    return wrapper

def bar():
    print('i am bar')

bar = use_logging(bar)
bar()
```



```
def use_logging(func):

    def wrapper(*args, **kwargs):
        logging.warn("%s is running" % func.__name__)
        return func(*args)
    return wrapper

@use_logging
def foo():
    print("i am foo")

@use_logging
def bar():
    print("i am bar")

bar()
```

如上所示，这样我们就可以省去bar = use_logging(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。

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@use_logging，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。

```
def use_logging(level):
    def decorator(func):
        def wrapper(*args, **kwargs):
            if level == "warn":
                logging.warn("%s is running" % func.__
```

```
            return func(*args)
        return wrapper

    return decorator

@use_logging(level="warn")
def foo(name='foo'):
    print("i am %s" % name)

foo()
```

上面的use_logging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。我们可以将它理解为一个含有参数的闭包。当我 们使用@use_logging(level="warn")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。

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


```
def __call__(self):
    print ('class decorator runing')
    self._func()
    print ('class decorator ending')

@Foo
def bar():
    print ('bar')

bar()
```

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：

装饰器

```
def logged(func):
    def with_logging(*args, **kwargs):
        print func.__name__ + " was called"
        return func(*args, **kwargs)
    return with_logging
```

函数

```
@logged
def f(x):
   """does some math"""
   return x + x * x
```

该函数完成等价于：

```
def f(x):
    """does some math"""
    return x + x * x
f = logged(f)
```

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

```
print f.__name__    # prints 'with_logging'
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装饰器函数中，这使得装饰器函数也有和原函数一样的元信息了。

```
from functools import wraps
def logged(func):
    @wraps(func)
```



```
print f.__name__  # prints 'f'
print f.__doc__   # prints 'does some math'
```

内置装饰器

@staticmathod、@classmethod、@property

装饰器的顺序

```
@a
@b
@c
def f ():
```

等效于

```
f = a(b(c(f)))
```

编辑于 2016-12-13

NET.Dzreal
多点生活（成都）科技有限公司 研发工程师

366 人赞同了该回答

晚上失眠，怒上知乎答题！

刚好最近我的python专栏里写过一篇装饰器相关的，**不说废话，直接上干货！**

目录如下：

一、装饰器是什么?


```
# 函数
def add(a, b):
    res = a + b
    return res
```

你可能会这么写

```
import time


def add(a, b)
    start_time = time.time()
    res = a + b
    exec_time = time.time() - start_time
    print("add函数，花费的时间是：{}".format(exec_time))
    return res
```

这个时候，老板又让你计算减法函数（sub）的时间。不用装饰器的话，你又得重复写一段减法的代码。

```
def sub(a, b)
    start_time = time.time()
    res = a - b
    exec_time = time.time() - start_time
    print("sub函数，花费的时间是：{}".format(exec_time))
    return res
```

这样显得很麻烦，也不灵活，**万一计算时间的代码有改动，你得每个函数都要改动。**

所以，我们需要引入装饰器。

使用装饰器之后的代码是这样的

```
import time


# 定义装饰器
def time_calc(func):
    def wrapper(*args, **kargs):
        start_time = time.time()
        f = func(*args,**kargs)
        exec_time = time.time() - start_time
        return f
    return wrapper

# 使用装饰器
@time_calc
def add(a, b):
    return a + b

@time_calc
def sub(a, b):
    return a - b
```


定义装饰器

```
def decorator(func):
    def wrapper(*args,**kargs):
        # 可以自定义传入的参数
        print(func.__name__)
        # 返回传入的方法名参数的调用
        return func(*args,**kargs)
    # 返回内层函数函数名
    return wrapper
```

二、使用装饰器

假设decorator是定义好的装饰器。

方法一：不用语法糖@符号

```
# 装饰器不传入参数时
f = decorator(函数名)

# 装饰器传入参数时
f = (decorator(参数))(函数名)
```

方法二：采用语法糖@符号

```
# 已定义的装饰器
@decorator
def f():
    pass

# 执行被装饰过的函数
f()
```

装饰器可以传参，也可以不用传参。

自身不传入参数的装饰器（采用两层函数定义装饰器）

```
def login(func):
    def wrapper(*args,**kargs):
        print('函数名:%s'% func.__name__)
        return func(*args,**kargs)
    return wrapper

@login
def f():
    print('inside decorator!')

f()

# 输出:
# >> 函数名:f
# >> 函数本身:inside decorator!
```



```
# 等价于 ==> (login(text))(f) ==> 返回 wrapper
@login('this is a parameter of decorator')
def f():
    print('2019-06-13')

# 等价于 ==> (login(text))(f)() ==> 调用 wrapper() 并返回 f()
f()

# 输出:
# => this is a parameter of decorator----f
# => 2019-06-13
```

三、内置装饰器

常见的内置装饰器有三种，@property、@staticmethod、 `@classmethod`

@property

把类内方法当成属性来使用，必须要有返回值，相当于getter；

假如没有定义 @func.setter 修饰方法的话，就是只读属性

```
class Car:

    def __init__(self, name, price):
        self._name = name
        self._price = price

    @property
    def car_name(self):
        return self._name

     # car_name可以读写的属性
     @car_name.setter
     def car_name(self, value):
         self._name = value
```

```
    # car_price是只读属性
    @property
    def car_price(self):
        return str(self._price) + '万'

benz = Car('benz', 30)

print(benz.car_name)   # benz
benz.car_name = "baojun"
print(benz.car_name)   # baojun
print(benz.car_price)  # 30万
```

@staticmethod

静态方法，不需要表示自身对象的self和自身类的cls参数，就跟使用函数一样。

@classmethod


```
    def instance_method(self):
        print("调用实例方法")

    @classmethod
    def class_method(cls):
        print("调用类方法")
        print("在类方法中 访问类属性 text: {}".format(cls.text))
        print("在类方法中 调用实例方法 instance_method: {}".format(cls().instance_method

    @staticmethod
    def static_method():
        print("调用静态方法")
        print("在静态方法中 访问类属性 text: {}".format(Demo.text))
        print("在静态方法中 调用实例方法 instance_method: {}".format(Demo().instance_met

if __name__ == "__main__":
    # 实例化对象
    d = Demo()

    # 对象可以访问 实例方法、类方法、静态方法
    # 通过对象访问text属性
    print(d.text)

    # 通过对象调用实例方法
    d.instance_method()

    # 通过对象调用类方法
    d.class_method()

    # 通过对象调用静态方法
    d.static_method()

    # 类可以访问类方法、静态方法
    # 通过类访问text属性
    print(Demo.text)

    # 通过类调用类方法
    Demo.class_method()

    # 通过类调用静态方法
    Demo.static_method()
```

@staticmethod 和 **@classmethod** 的 **区别** 和 **使用场景**：

在上述例子中，我们可以看出，

区别

在定义静态类方法和类方法时，**@staticmethod** 装饰的静态方法里面，想要访问类属性或调用实例方法，必须需要把类名写上；

而**@classmethod**装饰的类方法里面，会传一个cls参数，代表本类，这样就能够避免手写类名的硬编码。

在调用静态方法和类方法时，实际上写法都差不多，一般都是通过 类名.静态方法() 或 类名.类方法()。

假如**需要用到与类相关的属性或方法，然后又想表明这个方法是整个类通用的**，而**不是对象特异的**，就可以使用**类方法@classmethod。**

希望看到这个回答的朋友，永远不要失眠～

另外还有一些之前在github上总结的：测试开发面试资源、复习资料汇总

最后，欢迎加入【程序员知乎交流圈】↓↓↓

程序员交流圈 - 知乎
www.zhihu.com

置顶帖我放了头条内推链接，欢迎投递～（北京上海深圳广州杭州成都武汉都在招）

编辑于 2020-05-28

Jackpop
哈尔滨工业大学 计算数学硕士

145 人赞同了该回答

我从以下几点，由浅入深详细讲解一下Python装饰器：

- 什么事装饰器?
- 为什么用装饰器?
- 在哪里用装饰器?

然后以示例+讲解相结合的方式阐述，同时会讲解一些在很多教程和书籍中不会涉及到的内容。

什么是Python装饰器?

顾名思义，从字面意思就可以理解，它是用来"装饰"Python的工具，使得代码更具有Python简洁的风格。换句话说，它是一种函数的函数，因为装饰器传入的参数就是一个函数，然后通过实现各种功能来对这个函数的功能进行增强。

为什么用装饰器?

前面提到了，装饰器是通过某种方式来增强函数的功能。当然，我们可以通过很多方式来增强函数的功能，只是装饰器有一个无法替代的优势--简洁。

你只需要在每个函数上方加一个@就可以对这个函数进行增强。

在哪里用装饰器?

装饰器最大的优势是用于解决重复性的操作，其主要使用的场景有如下几个：

- 计算函数运行时间
- 给函数打日志

子来看一下它的作用。

如果你要对多个函数进行统计运行时间，不使用装饰器会是这样的，

```
from time import time, sleep

def fun_one():
    start = time()
    sleep(1)
    end = time()
    cost_time = end - start
    print("func one run time {}".format(cost_time))

def fun_two():
    start = time()
    sleep(1)
    end = time()
    cost_time = end - start
    print("func two run time {}".format(cost_time))

def fun_three():
    start = time()
    sleep(1)
    end = time()
    cost_time = end - start
    print("func three run time {}".format(cost_time))
```

在每个函数里都需要获取开始时间**start**、结束时间**end**、计算耗费时间**cost_time**、加上一个输出语句。

使用装饰器的方法是这样的，

```
def run_time(func):
    def wrapper():
        start = time()
        func()                  # 函数在这里运行
        end = time()
        cost_time = end - start
        print("func three run time {}".format(cost_time
```

```
    return wrapper

@run_time
def fun_one():
    sleep(1)

@run_time
def fun_two():
    sleep(1)

@run_time
def fun_three():
    sleep(1)
```

通过编写一个统计时间的装饰器**run_time**，函数的作为装饰器的参数，然后返回一个统计时间的函数wrapper，这就是装饰器的写法，用专业属于来说这叫**闭包**，简单来说就是函数内嵌套函数。然后再每个函数上面加上**@run_time**来调用这个装饰器对不同的函数进行统计时间。

可见，统计时间这4行代码是重复的，一个函数需要4行，如果100个函数就需要400行，而使用装

除了上述简单的用法还有一些更高级的用法，比如用装饰器进行**类型检查**、添加**带参数**的的装饰器等。它们的用法大同小异，关于高级用法，这里以**带参数的装饰器**为例进行介绍。

不要把问题想的太复杂，带参数的装饰器其实就是在上述基本的装饰器的基础上在外面套一层接收参数的函数，下面通过一个例子说明一下。

以上述例子为基础，前面的**简单示例**输出的信息是，

```
func three run time 1.0003271102905273
func three run time 1.0006263256072998
func three run time 1.000312328338623
```

现在我认为这样的信息太**单薄**，需要它携带更多的信息，例如函数名称、日志等级等，这时候可以把函数名称和日志等级作为装饰器的参数，下面来时实现以下。

```
def logger(msg=None):
    def run_time(func):
        def wrapper(*args, **kwargs):
            start = time()
            func()                  # 函数在这里运行
            end = time()
            cost_time = end - start
            print("[{}] func three run time {}".format(msg, cost_time))
        return wrapper
    return run_time

@logger(msg="One")
def fun_one():
    sleep(1)

@logger(msg="Two")
def fun_two():
    sleep(1)

@logger(msg="Three")
def fun_three():
    sleep(1)
```

```
fun_one()
fun_two()
fun_three()
```

可以看出，我在**示例**基本用法里编写的装饰器外层又嵌套了一层函数用来接收参数**msg**，这样的话在每个函数(func_one、func_two、func_three)前面调用时可以给装饰器传入参数，这样的输出结果是，

```
[One] func three run time 1.0013229846954346
[Two] func three run time 1.000720500946045
[Three] func three run time 1.0001459121704102
```

自定义属性的装饰器

上述介绍的几种用法中其实有一个问题，就是装饰器**不够灵活**，我们预先定义了装饰器**run_time**，它就会按照我们定义的流程去工作，只具备这固定的一种功能，当然，我们前面介绍的通过**带参数的装饰器**让它具备了一定的灵活性，但是依然不够灵活，其实，我们还可以对装饰器添加一些属

那么问题来了，如果我们预先定义一个打印日志的装饰器，

```
def logger_info(func):
    logmsg = func.__name__
    def wrapper():
        func()
        log.log(logging.INFO, "{} if over.".format(logmsg))
    return wrapper
```

logging.INFO是打印日志的等级，如果我们仅仅写一个基本的日志装饰器**logger_info**，那么它的灵活度太差了，因为如果我们要输出**DEBUG**、**WARING**等级的日志，还需要重新写一个装饰器。

解决这个问题，有两个解决方法：

- 利用前面所讲的带参数装饰器，把**日志等级**传入装饰器
- 利用自定义属性来修改**日志等级**

由于第一种已经以统计函数运行时间的方式进行讲解，这里主要讲解第二种方法。

先看一下代码，

```
import logging
from functools import partial

def wrapper_property(obj, func=None):
    if func is None:
        return partial(attach_wrapper, obj)
    setattr(obj, func.__name__, func)
    return func

def logger_info(level, name=None, message=None):
    def decorate(func):

        logmsg = message if message else func.__name__

        def wrapper(*args, **kwargs):
            log.log(level, logmsg)
            return func(*args, **kwargs)
```

```
        @wrapper_property(wrapper)
        def set_level(newlevel):
            nonlocal level
            level = newlevel

        @wrapper_property(wrapper)
        def set_message(newmsg):
            nonlocal logmsg
            logmsg = newmsg

        return wrapper

    return decorate


@logger_info(logging.WARNING)
def main(x, y):
    return x + y
```

这里面最重要的是**wrapper_property**这个函数，它的功能是把一个函数**func**编程一个对象**obj**的


```
# WARNING:Test:main
# 6
```

来改变一下输出日志等级，

```
main.set_level(logging.ERROR)
main(5, 5)

# 输出
# ERROR:Test:main
# 10
```

输出日志等级改成了**ERROR**。

保留元信息的装饰器

很多教程中都会介绍装饰器，但是大多数都是千篇一律的围绕基本用法在展开，少部分会讲一下带参数的装饰器，但是有一个细节很少有教程提及，那就是**保留元信息的装饰器**。

什么是函数的元信息?

就是函数携带的一些基本信息，例如函数名、函数文档等，我们可以通过**func.__name__**获取函数名、可以通过**func.__doc__**获取函数的文档信息，用户也可以通过**注解**等方式为函数添加元信息。

例如下面代码，

```
from time import time

def run_time(func):
    def wrapper(*args, **kwargs):
        start = time()
        func()                  # 函数在这里运行
        end = time()
        cost_time = end - start
        print("func three run time {}".format(cost_time))
    return wrapper
```

```
@run_time
def fun_one():
    '''
    func one doc.
    '''
    sleep(1)

fun_one()

print(fun_one.__name__)
print(fun_one.__doc__)

# 输出
# wrapper
# None
```

可以看出，通过使用装饰器，函数**fun_one**的元信息都丢失了，那怎么样才能保留装饰器的元信息呢?

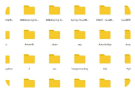
可以通过使用Python自带模块**functools**中的**wraps**来保留函数的元信息


```
        func()                  # 函数在这里运行
        end = time()
        cost_time = end - start
        print("func three run time {}".format(cost_time))
    return wrapper

@run_time
def fun_one():
    '''
    func one doc.
    '''
    sleep(1)

fun_one()

print(fun_one.__name__)
print(fun_one.__doc__)

# 输出
# fun_one
# func one doc.
```

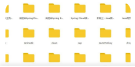
只需要在代码中加入**箭头**所指的一行即可保留函数的元信息。

干货

干货 | 2019年共享免费资源整理(上)：学习资源篇
mp.weixin.qq.com



干货 | 2019年共享免费资源整理(下)：实用工具篇
mp.weixin.qq.com



编辑于 2020-02-13

ucag
对计算机感兴趣的人

406 人赞同了该回答

之前写了一个1.0版，现在把更加完整的2.0版奉上。
1.0版已经被我杀掉了。
----------2.0版----------

在python中，不管什么东西都是对象。对象是什么东西呢?
对象就是你可以用来随意使用的模型。当你需要的时候就拿一个，不需要就让它放在那，垃圾回收机制会自动将你抛弃掉的对象回收。
可能对这个理解有一点云里雾里的感觉，甚至还觉得对这个概念很陌生。其实如果你都学到装饰器这里了，你已经使用过不少对象啦。
比如，我写了一个函数：

```
def cal(x, y):
    result = x + y
    return result
```

这时，你可以说，你创造了一个叫做cal的函数对象。
然后，你这样使用了它：

```
cal(1,2)
```

或者，你这样使用了它：

```
calculate = cal
calculate（1，2）
```

在第一种方式下，你直接使用了cal这个函数对象；
在第二种方式下，你把一个名为calculate的变量指向了cal这个函数对象。如果各位对类的使用很熟悉的话，可以把这个过程看作“实例化”。
也就是说，对象，就像是一个模子，当你需要的时候，就用它倒一个模型出来，每一个模型可以有自己不同的名字。在上面的例子中，calculate是一个模型，而你写的cal函数就是一个模子。

2、请理解函数**带括号和不带括号**时分别代表什么意思。
在上一个例子中，如果你只是写一个cal（也就是没有括号），那么此时的cal仅仅是代表一个函数对象；当你这样写cal(1, 2)时，就是在告诉编译器“执行cal这个函数”。

3、请确保能够理解**带星号的参数**是什么意思。
这个属于函数基础，要是你还没有听说过，那么就该回去好好复习一下了。具体讲解我就略过了。
----------正文分割线----------

1、装饰器是什么?

装饰器，顾名思义，就是用来“装饰”的。
它长这个样：

```
@xxx
```

其中"xxx"是你的装饰器的名字。
它能装饰的东西有：函数、类

2、为什么我需要装饰器?
有一句名言说的好（其实是我自己说的）：
“每一个轮子都有自己的用处”
所以，每一个装饰器也有自己的用处。
装饰器主要用来“偷懒”（轮子亦是如此）。
比如：
你写了很多个简单的函数，你想知道在运行的时候是哪些函数在执行，并且你又觉得这个没有必要写测试，只是想要很简单的在执行完毕之前给它打印上一句“Start”，那该怎么办呢？你可以这样：

值）……
如果你知道了装饰器，情况就开始渐渐变得好一些了，你知道可以这样写了：

```
def log(func):
    def wrapper(*arg, **kw):
        print 'Start %s' % func
        return func(*arg, **kw)
    return wrapper

@log
def func_a(arg):
    pass

@log
def func_b(arg):
    pass

@log
def func_c(arg):
    pass
```

其中，log函数是装饰器。
把装饰器写好了之后，只需要把需要装饰的函数前面都加上@log就可以了。在这个例子中，我们一次性就给三个函数加上了print语句。
可以看出，装饰器在这里为我们节省了代码量，并且在你的函数不需要装饰的时候直接把@log去掉就可以了，只需要用编辑器全局查找然后删除即可，快捷又方便，不需要自己手工的去寻找和删除print的语句在哪一行。
----------重点分割线----------
3、装饰器原理
在上一段中，或许你已经注意到了"log**函数**是装饰器"这句话。没错，装饰器是**函数。**
接下来，我将带大家探索一下，装饰器是怎么被造出来的，来直观的感受一下装饰器的原理。
先回到刚才的那个添加'Start'问题。
假设你此时还不知道装饰器。
将会以Solution的方式呈现。
S1 我有比在函数中直接添加print语句更好的解决方案!
于是你这样做了：

```
def a():
    pass
def b():
    pass
def c():
    pass

def main():
    print 'Start a'
    a()
    print 'Start b'
    b()
    print 'Start c'
    c()
```

感觉这样做好像没什么错，并且还避免了修改原来的函数，如果要手工删改print语句的话也更方便了。嗯，有点进步了，很不错。
S2 我觉得刚刚那个代码太丑了，还可以再优化一下！
于是你这样写了：


```
    print 'Start %s'% func
    func()

def main():
    decorator(a)
    decorator(b)
    decorator(c)
```

你现在写了一个函数来代替你为每一个函数写上print语句，好像又节省了不少时间。你欣喜的喝了一口coffee，对自己又一次做出了进步感到很满意。
嗯，确实是这样。
于是你选择出去上了个厕所，把刚刚憋的尿全部都排空（或许还有你敲代码时喝的coffee）。
回来之后，顿时感觉神清气爽！你定了定神，看了看自己刚才的“成果”，似乎又感到有一些不满意了。
因为你想到了会出现这样的情况：

```
def main():
    decorator(a)
    m = decorator(b)
    n = decorator(c) + m
    for i in decorator(d):
        i = i + n
    ......
```

来，就说你看到满篇的decorator你晕不晕！大声说出来！
S3 你又想了一个更好的办法。
于是你这样写了：

```
def a():
    pass
def b():
    pass
def c():
    pass

def decorator(func):
    print 'Start %s' % func
    return func
```

```
a = decorator(a)
b = decorator(b)
c = decorator(c)

def main():
    a()
    b()
    c()
```

这下总算是把名字给弄回来了，这样就不会晕了。你的嘴角又一次露出了欣慰的笑容（内心OS：哈哈哈，爷果然很6！）。于是你的手习惯性的端起在桌上的coffee，满意的抿了一口。
coffee的香味萦绕在唇齿之间，你满意的看着屏幕上的代码，突然！脑中仿佛划过一道闪电！要是a、b、c 三个函数带参数我该怎么办？！
你放下coffee，手托着下巴开始思考了起来，眉头紧锁。
像这样写肯定不行：

```
a = decorator(a(arg))
```


S4 你飞速的写下如下代码。

```
def a(arg):
    pass
def b(arg):
    pass
def c(arg):
    pass

def decorator(func):
    def wrapper(*arg, **kw)
        print 'Start %s' % func
        return func(*arg, **kw)
    return wrapper

a = decorator(a)
b = decorator(b)
c = decorator(c)

def main():
    a(arg)
    b(arg)
    c(arg)
```

decorator函数返回的是wrapper, wrapper是一个函数对象。而a = decorator(a)就相当于是把 a 指向了 wrapper, 由于wrapper可以有参数，于是变量 a 也可以有参数了！
终于！你从焦灼中解脱了出来！
不过， 有了前几次的经验，你这一次没有笑。
你又仔细想了想，能不能将a = decorator(a)这个过程给自动化呢?
于是你的手又开始在键盘上飞快的敲打，一会儿过后，你终于完成了你的“作品”。
你在python中添加了一个语法规则，取名为“@”，曰之“装饰器”。
你此时感觉有些累了， 起身打开门， 慢步走出去，深吸一口气，感觉阳光格外新鲜。
你的脸上终于露出了一个大大的笑容。
----------乱侃结束分割线----------
讲到这里，我想大家应该差不多都明白了装饰器的原理。
在评论中有知友问到，要是我的装饰器中也有参数该怎么办呢?

要是看懂了刚才添加参数的解决方案，也就不觉得难了。
再加一层就解决了。

```
def decorator(arg_of_decorator):
    def log(func):
        def wrapper(*arg, **kw):
            print 'Start %s' % func
            #TODO Add here sentences which use arg_of_decorator
            return func(*arg, **kw)
        return wrapper
    return log
```

感谢阅读：)

请大家着重理解思路。
答主是一名在校大学生，主修英语专业，目前大二，希望能够和各位知友多多交流。要是有同道中人关注我就更好了！

139 人赞同了该回答

> Python装饰器是程序开发中经常使用到的功能，熟练掌握装饰器会让你的编程思路更加广阔

要理解装饰器首先我们要理解以下两点：

1. 在 Python 中“函数是一等对象” 。即函数是一种特殊类型的变量，可以和其余变量一样，可以作为参数传递给函数，也可以作为返回值返回。Python 中的整数、字符串和字典等都是一等对象。
2. 函数装饰器在导入模块时立即执行，而被装饰的函数只在明确调用时运行。

在讲装饰器之前，先要了解闭包

我们先看一段代码：

```
def outer(i):
    j = 1
    def inner():
        x = i + j
        print(x)
    return inner

test = outer(5)
test()

#输出 6
```

在上边的代码中，我们有一个外部函数“outer()”和该函数的内部函数“inner()”，以及外部函数的局部变量“i”、“j”。在内部函数中我们使用了外部函数的局部变量，最后返回了内部函数的引用，也就是“inner”。

那么我们可以给闭包一个定义：通过调用含有一个内部函数加上该外部函数持有的外部局部变量的外部函数产生的一个实例函数。

一般情况下，如果一个函数结束，函数的内部所有东西都会被释放，局部变量都会消失。闭包是一种特殊情况，如果外函数在结束时发现有自己的临时变量将会在内部函数中用到，就会把这个临时变量绑定给了内部函数，然后再结束。

闭包就说到这，不在做过多的赘述，如果题主不是很明白可以再去查阅相关资料。

那么装饰器和闭包又有什么关系呢?

我们首先来看一下装饰器的概念：装饰器本质上是一个Python函数，它可以让其他函数在不需要做任何代码变动的前提下增加额外功能，装饰器的返回值也是一个函数对象。

而实际上，装饰器就是一个闭包，把一个函数当做参数然后返回一个替代版函数。

> 以下是一个简单的例子：

我希望在不修改“sayHi”函数的情况下在其之前再输出一句话，这种在代码运行期间动态增加功能的方式，称之为“装饰器”。 本质上，装饰器就是一个返回函数的高阶函数

> 我们可以这么写：

```
def sayName(func):
    def inner():
        print("I'm Yu")
        return func
    return inner()

def sayHi():
    print('Hello, World')
s = sayName(sayHi)
s()
```

> 输出：

```
I'm Yu
Hello, World
```

但代码美中不足的是，我们每次给sayHi增加功能都需要用到类似 `s = sayName(sayHi)` 这句话。

python为了简化这种情况，提供了一个语法糖“@”。

> 简化上边的代码:

```
def sayName(func):
    def inner():
        print("I'm Yu")
        return func()
    return inner

@sayName
def sayHi():
```

```
    print('Hello, World')

sayHi()
```

> 输出：

```
I'm Yu
Hello, World
```

以上代码中，首先，在装饰器函数sayName中，sayName需要接受一个参数func，在其内部又定义了一个inner函数，在inner函数中增加一句输出，并返回func对象，然后sayName函数返回内部函数inner，其实就是一个闭包函数。

接下来在sayHi上边增加一个@sayName，其意义就是在python解释器之行到此处时，会调用装饰器函数"sayName"，并把被装饰得函数"sayHi"作为参数传入。此时的sayHi已经不是未加装饰时的函数了，而是指向sayName.inner函数地址。在接下来调用sayHi()时，其实就是调用sayName.inner。


```
def sayName(func):
    def inner(name):
        print("I'm Yu")
        return func(name)
    return inner

@sayName
def sayHi(name):
    print('Hi,' + name)

sayHi('siri')
```

> 输出：

```
I'm Yu
Hi,siri
```

两个装饰器装饰函数

这里测试两个装饰器装饰一个函数的结果：

```
def sayName(func):
    print('name')
    def inner():
        print("I'm Yu")
        return func()
    return inner

def sayAge(func):
    print('age')
    def inner():
        print("i'm 30")
        return  func()
    return inner

@sayName
@sayAge
def sayHi():
```

```
    print('Hello, World')

sayHi()
```

> 输出:

```
age
name
I'm Yu
i'm 30
Hello, World
```

我们来分析一下输出这个结果的原因：

首先，python解释器执行到第一个装饰器@sayName，在接下来发现装饰器下边不是一个函数而是另一个装饰器，解释器会执行第二个的装饰器@sayAge，然后把sayHi函数传入装饰器，所以首先输出了“age”，当@sayAge装饰完成，此时的sayHi函数地址指向了sayAge.inner的地址，解


```
def now(time):
    def sayName(func):
        def inner(name):
            print('现在是: %s' % time)
            print("I'm Yu")
            return func(name)
        return inner
    return sayName

@now('2016/10/30')
def sayHi(name):
    print('Hello,' + name)

sayHi('siri')
```

> 输出:

```
现在是: 2016/10/30
I'm Yu
Hello,siri
```

以上就是Python中，我对装饰器的理解，希望对题主有一定的帮助。

发布于 2019-01-11