知乎 首发于 LetsGoAhead

[Python]基础语法：函数装饰器，类装饰器，对象装饰器

筱霁
随心所欲

80 人赞同了该文章

赞同 80

分享

这篇文章的内容本身并不是我原创的，函数装饰器的内容来自原来在伯乐在线网站中看到的一篇译文《简单 12 步理解 Python 装饰器》（原文链接：python.jobbole.com/8505...），前几天在温故的时候发现伯乐在线已经访问不到了，因此把自己当初作的笔记拿出来分享一下，因为当初理解装饰器的使用也让我折腾了一番。本文在《简单 12 步理解 Python 装饰器》的基础上，将最后的扩展阅读部分推荐的两篇文章的内容也加了进来，分别作为类装饰器和对象装饰器的内容，这两篇文章分别是 Decorators I: Introduction to Python Decorators 和 Python Decorators II: Decorator Arguments 。

作用域、内嵌函数、生命周期

Python中每个函数都是一个新的作用域，也可以理解为命名空间。

一般而言在函数中定义的变量、内嵌函数等，其生命周期即所在函数的作用域，当函数执行完毕后，函数内所定义的变量、内嵌函数等应该会消失。而在下一次函数被调用时又会被重新创建。

变量搜索

当在函数中访问一个新的变量时，Python会在当前的命名空间中寻找该变量是否存在。如果变量不存在则会从上一级命名空间中搜寻，直到顶层命名空间。例如：

```
def outer():
    x = 1
    def inner():
        print(x)
    inner()

outer() # 打印结果为 1
```

当调用 `outer()` 函数时，内嵌函数 `inner()` 会被调用，当执行到 `print(x)` 语句时，Python会在 `inner()` 中搜寻局部变量 `x` ，但是没有查找到，因此会在上一级命名空间，也就是 `outer()` 中搜寻，在上一级中搜寻到 `x` ，最后将其打印。

但是在函数中对一个变量进行定义或者赋值时，Python**只会在当前命名空间中搜寻**该变量，如果不存在则会创建一个新的变量。如果上一级命名空间中有同名的变量，那么上一级同名的变量会在当前作用域中被覆盖。

```
def outer():
    x = 1
    def inner():
        x = 2
        print("The x in inner() is %d" % x)
    print("The x in outer() is %d" % x)

outer()
# 打印结果为
# The x in inner() is 2
# The x in outer() is 1
```

▲ 赞同 80 ▼ 8 条评论 分享 喜欢

下面的代码介绍闭包的概念：

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

foo = outer()
foo()
print(foo.__closure__)
# 打印结果为
# 1
# (<cell at 0x00000000004AF2E8: int object at 0x000000006F14B440>,)
```

从作用域的角度， `foo()` 实际上是调用了内嵌函数 `inner()` ，当执行到 `inner()` 中的 `print(x)` 语句时，在 `inner()` 中没有搜寻到 `x` ，然后会在 `outer()` 的命名空间中搜寻，找到 `x` 后进行打印。

但从生命周期的角度， `foo` 的值为 `outer()` 函数的返回值，当执行 `foo()` 时， `outer()` 函数已经执行完毕了，此时其作用域内定义的变量 `x` 也应该已经销毁，因此执行 `foo()` 时，当执行到 `pirnt(x)` 语句应该会出错。但实际执行过程中却没有。

这其实是Python支持的函数**闭包**的特性，在上面的例子中可以作这样的理解：在非全局作用域定义的 `inner()` 函数在定义时会记住外层命名空间，此时 `inner()` 函数包含了外层作用域变量 `x` 。 `function` 对象的 `__closure__` 属性指示了该函数对象是否是闭包函数，若不是闭包函数，则该属性值为 `None` ，否则为一个非空元组。

将上述例子进行一些修改：

```
def outer(x):
    def inner():
        print("The value of x is %d " % x)
    return inner

foo1 = outer(1)
foo2 = outer(2)

foo1()
foo2()

# 打印结果
# The value of x is 1
# The value of x is 2
```

可以看到当传递给 `outer()` 不同的参数时，得到的 `inner()` 打印结果是不同的，这证明了闭包函数 `inner()` 记录了其外层命名空间。

我们可以利用闭包的特性得到一个对已有函数运行行为进行扩充或修改的新函数，而同时保留已有函数，不用对已有函数的代码进行修改。

做到这个的第一步是将函数作为参数传递到我们的“闭包创建函数”中。

装饰器

装饰器其实就是一个以函数作为参数并返回一个替换函数的可执行函数，即**装饰器是一个函数，它以函数作为参数，返回另一个函数。**

下面是一个装饰器的示例：

```
        print("before some_func")
        ret = some_func()
        return ret + 1
    return inner

def foo():
    return 1

decorated = outer(foo)

decorated()

# 打印结果
# before some_func
# 2
```

示例中 decorated 是 foo 的装饰版，即给 foo 加上了一些东西。在实际使用中实现装饰器后可能就想用装饰器替换原来的函数了，这只需要给 foo 重新赋值即可：

```
foo = outer(foo)
```

之后调用 foo() 就都是调用装饰器，而不是原来的 foo 。根据闭包的原理，原来的 foo() 在作为参数传递进 outer() 时， inner() 就已经对其作了“记录”。

假设有一个提供坐标对象的库。它们主要由一对对（x，y）坐标组成。但当前这些坐标对象不支持数学运算，并且无法修改源码。如果有一个需求需要对这些坐标对象做很多数学运算，例如要构造能够接收两个坐标对象的 add 和 sub 函数，并且做适当的数学运算。这些函数很容易实现（为方便演示，提供一个简单的 Coordinate 类）。

```
class Coordinate(object):
    def __init__(self, x, y):
        self.x = x
        self.y = y
    def __repr__(self):
        return "Coord: " + str(self.__dict__)

def add(a, b):
    return Coordinate(a.x + b.x, a.y + b.y)

def sub(a, b):
    return Coordinate(a.x - b.x, a.y - b.y)
```

但是如果 add() 和 sub() 函数必须有边界检测功能呢？例如只能对正坐标进行加或减，并且返回值也限制为正坐标，此时 add() 和 sub() 函数不能满足需求：

```
one = Coordinate(100, 200)
two = Coordinate(300, 200)
three = Coordinate(-100, -100)

print(sub(one, two))
print(add(one, three))

# 打印结果
# Coord: {'y': 0, 'x': -200}
# Coord: {'y': 100, 'x': 0}
```

如果希望在不修改 one 、 two 和 three 的基础上，使得 one 和 two 的差值为 {x: 0, y: 0} ， one 和 three 的和为 {x: 100, y: 200} ，那么可以使用装饰器的方法。接下来用一个边界检测装饰器来实现这一点，而不用对每个函数里的输入参数和返回值添加边界检测。

知乎 首发于 LetsGoAhead

```
            if a.x < 0 or a.y < 0:
                a = Coordinate(a.x if a.x > 0 else 0, a.y if a.y > 0 else 0)
            if b.x < 0 or b.y < 0:
                b = Coordinate(b.x if b.x > 0 else 0, b.y if b.y > 0 else 0)
            ret = func(a, b)
            if ret.x < 0 or ret.y < 0:
                ret = Coordinate(ret.x if ret.x > 0 else 0, ret.y if ret.y > 0 else 0)
            return ret
        return checker

    add = wrapper(add)
    sub = wrapper(sub)

    print(sub(one, two))
    print(add(one, three))

    # 打印结果
    # Coord: {'y': 0, 'x': 0}
    # Coord: {'y': 200, 'x': 100}
```

这种方案不是唯一的解决方案，但它让代码更加简洁：通过将边界检测从函数本身分离，使用装饰器包装它们，并应用到所有需要的函数。可替换的方案是：在每个数学运算函数返回前，对每个输入参数和输出结果调用一个函数。不过就对函数应用边界检测的代码量而言，使用装饰器至少是较少重复的。

函数装饰器@符号的应用

Python 2.4通过在函数定义前添加一个装饰器名和 @ 符号，来实现对函数的包装。在上面代码示例中，用了一个包装的函数来替换包含函数的变量来实现了装饰函数。

```
add = wrapper(add)
```

这种模式可以随时用来包装任意函数。但是如果定义了一个函数，可以用 @ 符号来装饰函数，如下：

```
@wrapper
def add(a, b):
    return Coordinate(a.x + b.x, a.y + b.y)
```

值得注意的是，这种方式和简单的使用 `wrapper()` 函数的返回值来替换原始变量的做法没有什么不同—— Python只是添加了一些语法糖来使之看起来更加明确。

使用装饰器很简单！虽说写类似 `staticmethod` 或者 `classmethod` 的实用装饰器比较难，但用起来仅仅需要在函数前添加 `@装饰器名` 即可！

更通用的函数装饰器

Python的函数定义中，允许有一个接收所有多余的位置参数的参数，以及一个接收所有多余的关键字参数的参数，这两个参数的参数名前要分别用 * 和 ** 修饰。

例如：

```
def test(arg1, arg2, arg3, arg4, *arg_others, **kwarg_others):
    print(arg1, arg2, arg3, arg4)
    print(type(arg_others), arg_others)
    print(type(karg_others), kwarg_others)

test(1, 3, 5, 7, 9, 11, arg5 = 'A', arg6 = 'B')
# 1 3 5 7
```

函数参数列表只有“任意位置参数”和“任意关键字参数”也是允许的：

```
def test(*args, **kwargs):
    print(args)
    print(kwargs)

test(1, 3, 5, 7, arg1 = 'A', arg2 = 'B')
# (1, 3, 5, 7)
# {'arg1': 'A', 'arg2': 'B'}
```

另外在给函数传参时，用 * 修饰的参数将会被认为是列表，而将所有元素取出作为位置参数；用 ** 修饰的参数会被认为是字典，而将所有键和值取出作为关键字参数。例如：

```
def test(arg1, arg2, arg3, arg4, arg5):
    print(arg1, arg2, arg3, arg4, arg5)

arg_list = [1, 3, 5]
kwarg_dict = {'arg4': 2, 'arg5': 4}

test(*arg_list, **kwarg_dict)
# 1 3 5 2 4
```

利用这种语法特性可以编写可以装饰不同参数数量的函数的函数装饰器，例如：

```
def wrapper(func):
    def inner(*args, **kwargs):
        func(*args, **kwargs)
    return inner

@wrapper
def func1(a, b, c):
    print('a=', a, 'b=', 'c=', c)

func1(1, b = 'boy', c = 'cat')
# a= 1 b= c= cat

@wrapper
def func2(v1, v2):
    print(v1, v2)

func2(100, 200)
# 100 200
```

使用类作为装饰器

类也可以作为装饰器，使用起来可能比函数装饰器更方便。首先看下面一个简单的例子：

```
class myDecorator(object):
    def __init__(self, f):
        print("inside myDecorator.__init__()")
        f() # Prove that function definition has completed
    def __call__(self):
        print("inside myDecorator.__call__()")

@myDecorator
def aFunction():
    print("inside aFunction()")

print("Finished decorating aFunction()")
```

例子中函数 aFunction() 就使用了类 myDecorator 作为装饰器修饰。例子的输出结果如下：

```
inside myDecorator.__init__()
inside aFunction()
Finished decorating aFunction()
inside myDecorator.__call__()
```

可以看出，在 aFunction() 函数声明处进入了类 myDecorator 的 __init__() 方法，但要注意，从第2个输出可以看出，此时函数 aFunction() 的定义已经完成了，在 __init__() 中调用的输入参数 f() ，实际上是调用了 aFunction() 函数。至此 aFunction() 函数的声明完成，包括装饰器声明的部分，然后输出了第3个输出。最后执行 aFunction() 时，可以看出实际上是执行了类 myDecorator 的 __call__() 方法（定义了 __call__() 方法的类的对象可以像函数一样被调用，此时调用的是对象的 __call__() 方法）。

这个例子其实不难理解，因为根据装饰器语法的含义，下面的代码：

```
@myDecorator
def aFunction():
    # 。。。
```

等价于

```
def aFunction():
    # 。。。

aFunction = myDecorator(aFunction)
```

因此被装饰后的函数 aFunction() 实际上已经是类 myDecorator 的对象。当再调用 aFunction() 函数时，实际上就是调用类 myDecorator 的对象，因此会调用到类 myDecorator 的 __call__() 方法。

因此使用类作为装饰器装饰函数来对函数添加一些额外的属性或功能时，一般会在类的 __init__() 方法中记录传入的函数，再在 __call__() 调用修饰的函数及其它额外处理。

下面是一个简单的例子：

```
class entryExit(object):
    def __init__(self, f):
        self.f = f
    def __call__(self):
        print("Entering", self.f.__name__)
        self.f()
        print("Exited", self.f.__name__)

@entryExit
def func1():
    print("inside func1()")

@entryExit
def func2():
    print("inside func2()")

func1()
func2()
```

例子的输出：

```
Entering func1
inside func1()
```

```
inside func2()
Exited func2
```

使用对象作为装饰器

根据装饰器的语法，对象当然也可以作为装饰器使用，对比使用类作为装饰器，使用对象作为装饰器有些时候更有灵活性，例如能够方便定制和添加参数。下面是一个例子：

```
class Decorator:
    def __init__(self, arg1, arg2):
        print('执行类Decorator的__init__()方法')
        self.arg1 = arg1
        self.arg2 = arg2

    def __call__(self, f):
        print('执行类Decorator的__call__()方法')
        def wrap(*args):
            print('执行wrap()')
            print('装饰器参数：', self.arg1, self.arg2)
            print('执行' + f.__name__ + '()')
            f(*args)
            print(f.__name__ + '()执行完毕')
        return wrap

@Decorator('Hello', 'World')
def example(a1, a2, a3):
    print('传入example()的参数：', a1, a2, a3)

print('装饰完毕')

print('准备调用example()')
example('Wish', 'Happy', 'EveryDay')
print('测试代码执行完毕')
```

例子的输出为：

```
执行类Decorator的__init__()方法
执行类Decorator的__call__()方法
装饰完毕
准备调用example()
执行wrap()
装饰器参数： Hello World
执行example()
传入example()的参数： Wish Happy EveryDay
example()执行完毕
测试代码执行完毕
```

根据装饰器的语法，下面的代码：

```
@Decorator('Hello', 'World')
def example(a1, a2, a3):
    #...
```

等价于

```
def example(a1, a2, a3):
    #...

example = Decorator('Hello', 'World')(example)
```

`Decorator` 对象的 `__call__()` 方法来“封装” `exmaple()`，最后 `example()` 函数的实际上是闭包后， `__call__()` 方法中定义的 `wrap()` 函数。

发布于 2019-11-26 23:00

Python 装饰器 Python 入门

写下你的评论...

8 条评论 默认 最新

无能的艾薇娜
😠比那种只讲了怎么用没讲原理的好多了！学废了！
2022-02-24 回复 3

Tedde
你这示例错了，变量搜索那两都没调用inner，怎么会打印东西
10-25 回复 喜欢

Tully Monster
我能否用类实现一个装饰器，这个装饰器能够装饰一个类，且能向该装饰器传参?
04-08 回复 喜欢

咸蛋黄肉松饭团
up写得很好😚
01-31 回复 喜欢

无名
类装饰器修改了被装饰对象的类型，这个问题很严重啊
2022-08-02 回复 喜欢

林甜球
讲得真好!
2022-04-30 回复 喜欢

张立跟
变量搜索 & 闭包 表述 不精确，后面的例子不错。
2021-04-05 回复 喜欢

阿斯蒂芬
第二个例子真详细, 还覆盖了多参数传入
2021-01-08 回复 喜欢

文章被以下专栏收录

LetsGoAhead
通过学习和分享来让我们一起提升自我，共同进步

推荐阅读

python中的函数装饰器
又到了周末了，一下午没有什么事情~，可以再更新一篇python的用法的文章。 这个系列算的上是我学习python的笔记，虽然专栏的标题叫python高级编程，但内容比较低级。写作的目的只是在

Python 函数装饰器
Python 函数装饰器 装饰器(Decorators)是 Python 的一个重要部分。简单地说：他们是修改其他函数的功能的函数。他们有助于让我们的代码更简短，也更

知乎 首发于 LetsGoAhead