搞定Python 装饰器

2018年1月10日
18:01

简单 12 步理解 Python 装饰器

2016/05/12 · 基础知识 · 5 评论 · 装饰器

分享到：31

本文由 伯乐在线 - 7even 翻译，艾凌风 校稿。未经许可，禁止转载！

英文出处：Simeon Franklin。欢迎加入翻译组。

好吧，我标题党了。作为 Python 教师，我发现理解装饰器是学生们从接触后就一直纠结的问题。那是因为装饰器确实难以理解！想弄明白装饰器，需要理解一些函数式编程概念，并且要对Python中函数定义和函数调用语法中的特性有所了解。使用装饰器非常简单（见步骤10），但是写装饰器却很复杂。

虽然我没法让装饰器变得简单，但也许通过将问题进行一步步的讲解，可以帮助你更容易理解装饰器。由于装饰器较为复杂，文章会比较长，请坚持住！我会尽量使每个步骤简单明了，这样如果你理解了各个步骤，就能理解装饰器的原理。本文假定你具备最基础的 Python 知识，另外本文对工作中大量使用 Python 的人将大有帮助。

此外需要说明的是，本文中 Python 代码示例是用 doctest 模块来执行的。代码看起来像是交互式 Python 控制台会话（>>> 和 … 表示 Python 语句，输出则另起一行）。偶然有以“doctest”开头的“奇怪”注释——那些只是 doctest 的指令，可以忽略。

1. 函数

在 Python 中，使用关键字 def 和一个函数名以及一个可选的参数列表来定义函数。函数使用 return 关键字来返回值。定义和使用一个最简单的函数例子：

```
Python
1 >>> def foo():
2 ...     return 1
3 >>> foo()
4 1
```

函数体（和 Python 中所有的多行语句一样）由强制性的缩进表示。在函数名后面加上括号就可以调用函数。

2. 作用域

在 Python 函数中会创建一个新的作用域。Python 高手也称函数有自己的命名空间。也就是说，当在函数体中遇到变量时，Python 会首先在该函数的命名空间中寻找变量名。Python 有几个函数用来查看命名空间。下面来写一个简单函数来看看局部变量和全局变量的区别。

```
Python
1 >>> a_string = "This is a global variable"
2 >>> def foo():
3 ...     print locals()
4 >>> print globals() # doctest: +ELLIPSIS
5 {..., 'a_string': 'This is a global variable'}
6 >>> foo() # 2
7 {}
```

内建函数 globals 返回一个包含所有 Python 能识别变量的字典。（为了更清楚的描述，输出时省略了 Python 自动创建的变量。）在注释 #2 处，调用了 foo 函数，在函数中打印局部变量的内容。从中可以看到，函数 foo 有自己单独的、此时为空的命名空间。

3. 变量解析规则

当然，以上并不意味着我们不能在函数内部使用全局变量。Python 的作用域规则是， 变量的创建总是会创建一个新的局部变量但是变量的访问（包括修改）在局部作用域查找然后是整个外层作用域来寻找匹配。所以如果修改 foo 函数来打印全部变量，结果将是我们希望的那样：

Python

```
>>> a_string = "This is a global variable"
>>> def foo():
...     print a_string # 1
>>> foo()
This is a global variable
```

在 #1 处，Python 在函数 foo 中搜索局部变量 a_string，但是没有找到，然后继续搜索同名的全局变量。

另一方面，如果尝试在函数里给全局变量赋值，结果并不是我们想要的那样：

Python

```
>>> a_string = "This is a global variable"
>>> def foo():
...     a_string = "test" # 1
...     print locals()
>>> foo()
{'a_string': 'test'}
>>> a_string # 2
'This is a global variable'
```

从上面代码可见，全部变量可以被访问（如果是可变类型，甚至可以被修改）但是（默认）不能被赋值。在函数 #1 处，实际上是创建了一个和全局变量相同名字的局部变量，并且“覆盖”了全局变量。通过在函数 foo 中打印局部命名空间可以印证这一点，并且发现局部命名空间有了一项数据。在 #2 处的输出可以看到，全局命名空间里变量 a_string 的值并没有改变。

4. 变量生命周期

值得注意的是，变量不仅是在命名空间中有效，它们也有生命周期。思考下面的代码：

Python

```
>>> def foo():
...     x = 1
>>> foo()
>>> print x # 1
Traceback (most recent call last):
  ...
NameError: name 'x' is not defined
```

这个问题不仅仅是因为 #1 处的作用域规则（虽然那是导致 NameError 的原因），也与 Python 和很多其他语言中函数调用的实现有关。没有任何语法可以在该处取得变量 x 的值——它确确实实不存在！函数 foo 的命名空间在每次函数被调用时重新创建，在函数结束时销毁。

5. 函数的实参和形参

Python 允许向函数传递参数。形参名在函数里为局部变量。

Python

```
>>> def foo(x):
...     print locals()
>>> foo(1)
{'x': 1}
```

Python 有一些不同的方法来定义和传递函数参数。想要深入的了解，请参考 Python 文档关于函数的定义。来说一个简单版本：函数参数可以是强制的位置参数或者可选的有默认值的关键字参数。

Python

```
>>> def foo(x, y=0): # 1
...     return x - y
>>> foo(3, 1) # 2
2
>>> foo(3) # 3
3
>>> foo() # 4
Traceback (most recent call last):
  ...
TypeError: foo() takes at least 1 argument (0 given)
>>> foo(y=1, x=3) # 5
2
```

在 #1 处，定义了有一个位置参数 x 和一个关键字参数 y的函数。接着可以看到，在 #2 处通过普通传参的方式调用该函数——实参值按位置传递给了 foo 的参数，尽管其中一个参数是作为关键字参数定义的。在 # 3 处可以看到，调用函数时可以无需给关键字参数传递实参——如果没有给关键字参数 y 传值，Python 将使用声明的默认值 0 为其赋值。当然，参数 x （即位置参数）的值不能为空——在 #4 示范了这种错误异

常。

都很清楚简单，对吧？接下来有些复杂了—— Python 支持在函数调用时使用关键字实参。看 #5 处，虽然函数是用一个关键字形参和一个位置形参定义的，但此处使用了两个关键字实参来调用该函数。因为参数都有名称，所以传递参数的顺序没有影响。

反过来也是对的。函数 foo 的一个参数被定义为关键字参数，但是如果按位置顺序传递一个实参——在 # 2 处调用 foo(3, 1)，给位置形参 x 传实参 3 并给第二个形参 y 传第二个实参（整数 1），尽管 y 被定义为关键字参数。

哇哦！说了这么多看起来可以简单概括为一点：函数的参数可以有名称或位置。也就是说这其中稍许的不同取决于是函数定义还是函数调用。可以对用位置形参定义的函数传递关键字实参，反过来也可行！如果还想进一步了解请查看 Python 文档。

6. 内嵌函数

Python 允许创建内嵌函数。即可以在函数内部声明函数，并且所有的作用域和生命周期规则仍然适用。

Python

```
>>> def outer():
...     x = 1
...     def inner():
...         print x # 1
...     inner() # 2
...
>>> outer()
1
```

以上代码看起来有些复杂，但它仍是易于理解的。来看 #1 —— Python 搜索局部变量 x 失败，然后在属于另一个函数的外层作用域里寻找。变量 x 是函数 outer 的局部变量，但函数 inner 仍然有外层作用域的访问权限（至少有读和修改的权限）。在 #2 处调用函数 inner。值得注意的是，inner 在此处也只是一个变量名，遵循 Python 的变量查找规则——Python 首先在 outer 的作用域查找并找到了局部变量 inner。

7. 函数是 Python 中的一级对象

在 Python 中有个常识：函数和其他任何东西一样，都是对象。函数包含变量，它并不那么特殊。

Python

```
>>> issubclass(int, object) # all objects in Python inherit from a common baseclass
True
>>> def foo():
...     pass
>>> foo.__class__ # 1
<type 'function'>
>>> issubclass(foo.__class__, object)
True
```

也许你从未考虑过函数可以有属性——但是函数在 Python 中，和其他任何东西一样都是对象。（如果对此感觉困惑，稍后你会看到 Python 中的类也是对象，和其他任何东西一样！）也许这有点学术的感觉——在 Python 中函数只是常规的值，就像其他任意类型的值一样。这意味着可以将函数当做实参传递给函数，或者在函数中将函数作为返回值返回。如果你从未想过这样使用，请看下面的可执行代码：

看下面的闭包的例子：是不是就可以理解：x是属于函数 outer() 的一个属性

Python

```
>>> def add(x, y):
...     return x + y
>>> def sub(x, y):
...     return x - y
>>> def apply(func, x, y): # 1
...     return func(x, y) # 2
>>> apply(add, 2, 1) # 3
3
>>> apply(sub, 2, 1)
1
```

这个示例对你来说应该不陌生——add 和 sub 是标准的 Python 函数，都是接受两个值并返回一个计算的值。在 #1 处可以看到变量接收一个就像其他普通变量一样的函数。在 #2处调用了传递给 apply 的函数 fun——在 Python 中双括号是调用操作符，调用变量名包含的值。在 #3 处展示了在 Python 中把函数作为值传参并没有特别的语法——和其他变量一样，函数名就是变量标签。

也许你之前见过这种写法—— Python 使用函数作为实参，常见的操作如：通过传递一个函数给 key 参数，

为值传参并没有特别的语法——和其他变量一样，函数名就是变量标签。

也许你之前见过这种写法—— Python 使用函数作为实参，常见的操作如：通过传递一个函数给 key 参数，来自定义使用内建函数 sorted。但是，将函数作为值返回会怎样？思考下面代码：

Python

```
>>> def outer():
...     def inner():
...         print "Inside inner"
...     return inner # 1
...
>>> foo = outer() #2
>>> foo # doctest:+ELLIPSIS
<function inner at 0x...>
>>> foo()
Inside inner
```

这看起来也许有点怪异。在 #1 处返回一个其实是函数标签的变量 inner。也没有什么特殊语法——函数 outer 返回了并没有被调用的函数 inner。还记得变量的生命周期吗？每次调用函数 outer 的时候，函数 inner 会被重新定义，但是如果函数 ouer 没有返回 inner，当 inner 超出 outer 的作用域，`inner` 的生命周期将结束。

在 #2 处将获得返回值即函数 inner，并赋值给新变量 foo。可以看到如果鉴定 foo，它确实包含函数 inner，通过使用调用操作符（双括号，还记得吗？）来调用它。虽然看起来可能有点怪异，但是目前为止并没有什么很难理解的，对吧？hold 住，因为接下来会更怪异！

8. 闭包

先不着急看闭包的定义，让我们从一段示例代码开始。如果将上一个示例稍微修改下：

Python

```
>>> def outer():
...     x = 1
...     def inner():
...         print x # 1
...     return inner
>>> foo = outer()
>>> foo.func_closure # doctest: +ELLIPSIS
(<cell at 0x...: int object at 0x...>,)
```

从上一个示例可以看到，`inner` 是 outer 返回的一个函数，存储在变量 foo 里然后用 foo() 来调用。但是它能运行吗？先来思考一下作用域规则。

Python 中一切都按作用域规则运行—— x 是函数 outer 中的一个局部变量，当函数 inner 在 #1 处打印 x 时，Python 在 inner 中搜索局部变量但是没有找到，然后在外层作用域即函数 outer 中搜索找到了变量 x。

但如果从变量的生命周期角度来看应该如何呢？变量 x 对函数 outer 来说是局部变量，即只有当 outer 运行时它才存在。只有当 outer 返回后才能调用 inner，所以依据 Python 运行机制，在调用 inner 时 x 就应该不存在了，那么这里应该有某种运行错误出现。

结果并不是如此，返回的 inner 函数正常运行。**Python 支持一种名为函数闭包的特性，意味着 在非全局作用域定义的 inner 函数在定义时记得外层命名空间是怎样的。`inner` 函数包含了外层作用域变量，通过查看它的 func_closure 属性可以看出这种函数闭包特性。**

记住——每次调用函数 outer 时，函数 inner 都会被重新定义。此时 x 的值没有变化，所以返回的每个 inner 函数和其它的 inner 函数运行结果相同，但是如果稍做一点修改呢？

Python

```
>>> def outer(x):
...     def inner():
...         print x # 1
...     return inner
>>> print1 = outer(1)
>>> print2 = outer(2)
>>> print1()
1
>>> print2()
2
```

从这个示例可以看到闭包——函数记住其外层作用域的事实——可以用来构建本质上有一个硬编码参数的自

定义函数。虽然没有直接给 inner 函数传参 1 或 2，但构建了能“记住”该打印什么数的 inner 函数自定义版本。

闭包是强大的技术——在某些方面来看可能感觉它有点像面向对象技术：`outer` 作为 inner的构造函数，有一个类似私有变量的 x。闭包的作用不胜枚举——如果你熟悉 Python中 sorted 函数的参数 key，也许你已经写过 lambda 函数通过第二项而非第一项来排序一些列表。也可以写一个 itemgetter 函数，接收一个用于检索的索引并返回一个函数，然后就能恰当的传递给 key 参数了。

但是这么用闭包太没意思了！让我们再次从头开始，写一个装饰器。

9. 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。让我们从简单的开始，直到能写出实用的装饰器。

通过修饰器的加工： 在inner()当中能够满足 运行执行 some_func的情况下；对，some_func的执行效果进行 2次 加工修饰，其中，inner()返回的是，修饰后的效果逻辑代码段，返回的新的执行效果； 而外部outer() 返回的是 经过 代码逻辑修饰的新的 修饰函数 这里的例子中叫：inner； 而，outer()有唯一的输入参数就是 some_func

Python

```
>>> def outer(some_func):
...     def inner():
...         print "before some_func"
...         ret = some_func() # 1
...         return ret + 1
...     return inner
>>> def foo():
...     return 1
>>> decorated = outer(foo) # 2
>>> decorated()
before some_func
2
```

请仔细看这个装饰器示例。首先，定义了一个带单个参数 some_func 的名为 outer 的函数。然后在 outer 内部定义了一个内嵌函数 inner。`inner` 函数将打印一行字符串然后调用 some_func，并在 #1 处获取其返回值。在每次 outer 被调用时，`some_func` 的值可能都会不同，但不论 some_func 是什么函数，都将调用它。最后，`inner` 返回 some_func() 的返回值加 1。在 #2 处可以看到，当调用赋值给 decorated 的返回函数时，得到的是一行文本输出和返回值 2，而非期望的调用 foo 的返回值 1。

```
decorated = outer(foo) # 2
```

decorated

我们可以说变量 decorated 是 foo 的装饰版——即 foo 加上一些东西。事实上，如果写了一个实用的装饰器，可能会想用装饰版来代替 foo，这样就总能得到“附带其他东西”的 foo 版本。用不着学习任何新的语法，通过将包含函数的变量重新赋值就能轻松做到这一点：

Python

```
>>> foo = outer(foo)
>>> foo # doctest: +ELLIPSIS
<function inner at 0x...>
```

现在任意调用 foo() 都不会得到原来的 foo，而是新的装饰器版！明白了吗？来写一个更实用的装饰器。

想象一个提供坐标对象的库。它们可能主要由一对对的 x、y坐标组成。遗憾的是坐标对象不支持数学运算，并且我们也无法修改源码。然而我们需要做很多数学运算，所以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）。

Python

```
>>> 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)
>>> one = Coordinate(100, 200)
>>> two = Coordinate(300, 200)
>>> add(one, two)
Coord: {'y': 400, 'x': 400}
```

但是如果 add 和 sub 函数必须有边界检测功能呢？也许只能对正坐标进行加或减，并且返回值也限制为正坐标。如下：

Python

```
>>> one = Coordinate(100, 200)
>>> two = Coordinate(300, 200)
>>> three = Coordinate(-100, -100)
>>> sub(one, two)
Coord: {'y': 0, 'x': -200}
>>> add(one, three)
Coord: {'y': 100, 'x': 0}
```

但我们希望在不修改 one、two 和 three的基础上，one 和 two 的差值为 {x: 0, y: 0}，one 和 three 的和为 {x: 100, y: 200}。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。

Python

```
>>> def wrapper(func):
...     def checker(a, b): # 1
...         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)
>>> sub(one, two)
Coord: {'y': 0, 'x': 0}
>>> add(one, three)
Coord: {'y': 200, 'x': 100}
```

装饰器和之前一样正常运行——返回了一个修改版函数，但在这次示例中通过检测和修正输入参数和返回值，将任何负值的 x 或 y 用 0 来代替，实现了上面的需求。

是否这么做是见仁见智的，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，不可否认，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。事实上，如果要装饰的函数是我们自己实现的，可以使装饰器应用得更明确一点。

10. 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。

Python

```
>>> 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！

关于这里的理解可以附加上下面的内容增加理解：

```
#我们可以这样解决
import datetime
def extrafoo(func):
    def inner():
        print('from inner to execute:',func.__name__)
        print('the',func.__name__,'result:',func())
        print('extra:',datetime.datetime.now())
    return inner

@extrafoo#装饰器特性，被装饰的函数定义之后立即运行。
def foo1():
    return 'this is foo1 function—'


#@是python装饰器的简便写法，也叫语法糖
#装饰器语法糖在要被包裹的函数前声明。@后面的函数名，是包裹下边函数的函数名extrafoo
#该语法糖省略了
#decorated=foo(test)
#decorated()


# 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。


foo1()
```

11. *args 和 **kwargs

这一个知识点加在这里非常到位

上面我们写了一个实用的装饰器，但它是硬编码的，只适用于特定类型的函数——带有两个参数的函数。内部函数 checker 接收两个参数，然后继续将参数传给闭包中的函数。如果我们想要一个能适用任何函数的装饰器呢？让我们来实现一个为每次被装饰函数的调用添加一个计数器的装饰器，但不改变被装饰函数。这意味着这个装饰器必须接收它所装饰的任何函数的调用信息，并且在调用这些函数时将传递给该装饰器的任何参数都传递给它们。

碰巧，Python 对这种特性提供了语法支持。请务必阅读 Python Tutorial 以了解更多，但在定义函数时使用 * 的用法意味着任何传递给函数的额外位置参数都是以 * 开头的。如下：

Python

```
>>> def one(*args):
...     print args # 1
>>> one()
()
>>> one(1, 2, 3)
(1, 2, 3)
>>> def two(x, y, *args): # 2
...     print x, y, args
>>> two('a', 'b', 'c')
a b ('c',)
```

第一个函数 one 简单的打印了传给它的任何位置参数（如果有）。在 #1 处可以看到，在函数内部只是简单的用到了变量 args —— *args 只在定义函数时用来表示位置参数将会保存在变量 args 中。Python 也允许指定一些变量，并捕获任何在 args 里的额外参数，如 #2 处所示。

* 符号也可以用在函数调用时，在这里它也有类似的意义。**在调用函数时，以 * 开头的变量表示该变量内容需被取出用做位置参数。再举例如下：**

Python

```
>>> def add(x, y):
...     return x + y
>>> lst = [1,2]
>>> add(lst[0], lst[1]) # 1
3
```

```
>>> add(*lst) # 2
3
```

在 #1 处的代码和 #2 处的作用相同——可以手动做的事情，在 #2 处 Python 帮我们自动处理了。**这看起来不错，*args 可以表示在调用函数时从迭代器中取出位置参数， 也可以表示在定义函数时接收额外的位置参数。**

接下来介绍稍微复杂一点的用来表示字典和键值对的 **，就像 * 用来表示迭代器和位置参数。很简单吧？

Python

```
>>> def foo(**kwargs):
...     print kwargs
>>> foo()
{}
>>> foo(x=1, y=2)
{'y': 2, 'x': 1}
```

当定义一个函数时，使用 **kwargs 来表示所有未捕获的关键字参数将会被存储在字典 kwargs 中。此前 args 和 kwargs 都不是 Python 中语法的一部分，但在函数定义时使用这两个变量名是一种惯例。和 * 的使用一样，可以在函数调用和定义时使用 **。

Python

```
>>> dct = {'x': 1, 'y': 2}
>>> def bar(x, y):
...     return x + y
>>> bar(**dct)
3
```

12. 更通用的装饰器

用学到的新知识，可以写一个记录函数参数的装饰器。为简单起见，仅打印到标准输出：

Python

```
>>> def logger(func):
...     def inner(*args, **kwargs): #1
...         print "Arguments were: %s, %s" % (args, kwargs)
...         return func(*args, **kwargs) #2
...     return inner
```

注意在 #1 处函数 inner 接收任意数量和任意类型的参数，然后在 #2 处将他们传递给被包装的函数。这样一来我们可以包装或装饰任意函数，而不用管它的签名。

Python

```
>>> @logger
... def fool(x, y=1):
...     return x * y
>>> @logger
... def foo2():
...     return 2
>>> fool(5, 4)
Arguments were: (5, 4), {}
20
>>> fool(1)
Arguments were: (1,), {}
1
>>> foo2()
Arguments were: (), {}
2
```

每一个函数的调用会有一行日志输出和预期的返回值。

再聊装饰器

如果你一直看到了最后一个实例，祝贺你，你已经理解了装饰器！你可以用新掌握的知识做更多的事了。

你也许考虑需要进一步的学习：Bruce Eckel 有一篇很赞的关于装饰器文章，他使用了对象而非函数来实现了装饰器。你会发现 OOP 代码比纯函数版的可读性更好。Bruce 还有一篇后续文章 providing arguments to decorators，用对象实现装饰器也许比用函数实现更简单。最后，你可以去研究一下内建包装函数 functools，它是一个在装饰器中用来修改替换函数签名的装饰器，使得这些函数更像是被装饰的函数。