# 12步轻松搞定python装饰器

初学者一开始很难搞定python的装饰器，也许因为装饰器确实很难懂。搞定装饰器需要你了解一些函数式编程的概念，当然还有理解在python中定义和调用函数相关语法的一些特点。

我们没法让装饰器变得简单，但是通过一步步的剖析，也许能够让你在理解装饰器的时候更自信一点。因为装饰器很复杂，这篇文章将会很长。

****1. 函数****

在python中，函数通过def关键字、函数名和可选的参数列表定义。通过return关键字返回值。我们举例来说明如何定义和调用一个简单的函数：

Python

|  |  |
| --- | --- |
| 1  2  3  4 | **def** foo():  **return** 1  foo()  1 |

方法体（当然多行也是一样的）是必须的，通过缩进来表示，在方法名的后面加上双括号()就能够调用函数

****2. 作用域****

在python中，函数会创建一个新的作用域。python开发者可能会说函数有自己的命名空间，差不多一个意思。这意味着在函数内部碰到一个变量的时候函数会优先在自己的命名空间里面去寻找。让我们写一个简单的函数看一下 本地作用域 和 全局作用域有什么不同：

Python

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | a\_string = "This is a global variable"  **def** foo():       print locals()  print globals() *# doctest: +ELLIPSIS*  {, 'a\_string': 'This is a global variable'}  foo() *# 2*  {} |

内置的函数globals返回一个包含所有python解释器知道的变量名称的字典（为了干净和洗的白白的，我省略了python自行创建的一些变量）。在#2我调用了函数 foo 把函数内部本地作用域里面的内容打印出来。我们能够看到，函数foo有自己独立的命名空间，虽然暂时命名空间里面什么都还没有。

****3. 变量解析规则****

当然这并不是说我们在函数里面就不能访问外面的全局变量。在python的作用域规则里面，创建变量一定会一定会在当前作用域里创建一个变量，但是访问或者修改变量时会先在当前作用域查找变量，没有找到匹配变量的话会依次向上在闭合的作用域里面进行查看找。所以如果我们修改函数foo的实现让它打印全局的作用域里的变量也是可以的：

Python

|  |  |
| --- | --- |
| 1  2  3  4  5 | a\_string = "This is a global variable"  **def** foo():       print a\_string *# 1*  foo()  This **is** a global variable |

在#1处，python解释器会尝试查找变量a\_string，当然在函数的本地作用域里面是找不到的，所以接着会去上层的作用域里面去查找。  
但是另一方面，假如我们在函数内部给全局变量赋值，结果却和我们想的不一样：

Python

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | 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' |

我们能够看到，全局变量能够被访问到（如果是可变数据类型(像list,dict这些)甚至能够被更改）但是赋值不行。在函数内部的#1处，我们实际上新创建了一个局部变量，隐藏全局作用域中的同名变量。我们可以通过打印出局部命名空间中的内容得出这个结论。我们也能看到在#2处打印出来的变量a\_string的值并没有改变。

****4. 变量生存周期****

值得注意的一个点是，变量不仅是生存在一个个的命名空间内，他们都有自己的生存周期，请看下面这个例子：

Python

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | **def** foo():       x = 1  foo()  print x *# 1*  Traceback (most recent call last):    NameError: name 'x' **is** **not** defined |

#1处发生的错误不仅仅是因为作用域规则导致的（尽管这是抛出了NameError的错误的原因）它还和python以及其它很多编程语言中函数调用实现的机制有关。在这个地方这个执行时间点并没有什么有效的语法让我们能够获取变量x的值，因为它这个时候压根不存在！函数foo的命名空间随着函数调用开始而开始，结束而销毁。

****5. 函数参数****

python允许我们向函数传递参数，参数会变成本地变量存在于函数内部。

Python

|  |  |
| --- | --- |
| 1  2  3  4 | **def** foo(x):       print locals()  foo(1)  {'x': 1} |

在Python里有很多的方式来定义和传递参数，完整版可以查看 python官方文档。我们这里简略的说明一下：函数的参数可以是必须的位置参数或者是可选的命名，默认参数。

Python

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | **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处我们定义了函数foo,它有一个位置参数x和一个命名参数y。在#2处我们能够通过常规的方式来调用函数，尽管有一个命名参数，但参数依然可以通过位置传递给函数。在调用函数的时候，对于命名参数y我们也可以完全不管就像#3处所示的一样。如果命名参数没有接收到任何值的话，python会自动使用声明的默认值也就是0。需要注意的是我们不能省略第一个位置参数x, 否则的话就会像#5处所示发生错误。

目前还算简洁清晰吧， 但是接下来可能会有点令人困惑。python支持函数调用时的命名参数（个人觉得应该是命名实参）。看看#5处的函数调用，我们传递的是两个命名实参，这个时候因为有名称标识，参数传递的顺序也就不用在意了。

当然相反的情况也是正确的：函数的第二个形参是y，但是我们通过位置的方式传递值给它。在#2处的函数调用foo(3,1)，我们把3传递给了第一个参数，把1传递给了第二个参数，尽管第二个参数是一个命名参数。

桑不起，感觉用了好大一段才说清楚这么一个简单的概念：函数的参数可以有名称和位置。这意味着在函数的定义和调用的时候会稍稍在理解上有点儿不同。我们可以给只定义了位置参数的函数传递命名参数（实参），反之亦然！如果觉得不够可以查看官方文档

****6. 嵌套函数****

Python允许创建嵌套函数。这意味着我们可以在函数里面定义函数而且现有的作用域和变量生存周期依旧适用。

Python

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | **def** outer():       x = 1  **def** inner():           print x *# 1*       inner() *# 2*    outer()  1 |

这个例子有一点儿复杂，但是看起来也还行。想一想在#1发生了什么：python解释器需找一个叫x的本地变量，查找失败之后会继续在上层的作用域里面寻找，这个上层的作用域定义在另外一个函数里面。对函数outer来说，变量x是一个本地变量，但是如先前提到的一样，函数inner可以访问封闭的作用域（至少可以读和修改）。在#2处，我们调用函数inner，非常重要的一点是，inner也仅仅是一个遵循python变量解析规则的变量名，python解释器会优先在outer的作用域里面对变量名inner查找匹配的变量.

****7. 函数是python世界里的一级类对象****

显而易见，在python里函数和其他东西一样都是对象。（此处应该大声歌唱）啊！包含变量的函数，你也并不是那么特殊！

Python

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | 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** |

你也许从没有想过，你定义的函数居然会有属性。没办法，函数在python里面就是对象，和其他的东西一样，也许这样描述会太学院派太官方了点：在python里，函数只是一些普通的值而已和其他的值一毛一样。这就是说你可以把函数想参数一样传递给其他的函数或者说从函数了里面返回函数！如果你从来没有这么想过，那看看下面这个例子：

Python

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | **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 |

这个例子对你来说应该不会很奇怪。add和sub是非常普通的两个python函数，接受两个值，返回一个计算后的结果值。在#1处你们能看到准备接收一个函数的变量只是一个普通的变量而已，和其他变量一样。在#2处我们调用传进来的函数：“()代表着调用的操作并且调用变量包含的值。在#3处，你们也能看到传递函数并没有什么特殊的语法。” 函数的名称只是很其他变量一样的表标识符而已。

你们也许看到过这样的行为：“python把频繁要用的操作变成函数作为参数进行使用，像通过传递一个函数给内置排序函数的key参数从而来自定义排序规则。那把函数当做返回值回事这样的情况呢：

Python

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | **def** outer():  **def** inner():           print "Inside inner"  **return** inner *# 1*    foo = outer() *#2*  foo *# doctest:+ELLIPSIS*  <function inner at 0x>  foo()  Inside inner |

这个例子看起来也许会更加的奇怪。在#1处我把恰好是函数标识符的变量inner作为返回值返回出来。这并没有什么特殊的语法：”把函数inner返回出来，否则它根本不可能会被调用到。“还记得变量的生存周期吗？每次函数outer被调用的时候，函数inner都会被重新定义，如果它不被当做变量返回的话，每次执行过后它将不复存在。

在#2处我们捕获住返回值 – 函数inner，将它存在一个新的变量foo里。我们能够看到，当对变量foo进行求值，它确实包含函数inner，而且我们能够对他进行调用。初次看起来可能会觉得有点奇怪，但是理解起来并不困难是吧。坚持住，因为奇怪的转折马上就要来了（嘿嘿嘿嘿，我笑的并不猥琐！）

****8. 闭包****

我们先不急着定义什么是闭包，先来看看一段代码，仅仅是把上一个例子简单的调整了一下：

Python

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | **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>,) |

在上一个例子中我们了解到，inner作为一个函数被outer返回，保存在一个变量foo，并且我们能够对它进行调用foo()。不过它会正常的运行吗？我们先来看看作用域规则。

所有的东西都在python的作用域规则下进行工作：“x是函数outer里的一个局部变量。当函数inner在#1处打印x的时候，python解释器会在inner内部查找相应的变量，当然会找不到，所以接着会到封闭作用域里面查找，并且会找到匹配。

但是从变量的生存周期来看，该怎么理解呢？我们的变量x是函数outer的一个本地变量，这意味着只有当函数outer正在运行的时候才会存在。根据我们已知的python运行模式，我们没法在函数outer返回之后继续调用函数inner，在函数inner被调用的时候，变量x早已不复存在，可能会发生一个运行时错误。

万万没想到，返回的函数inner居然能够正常工作。Python支持一个叫做函数闭包的特性，用人话来讲就是，嵌套定义在非全局作用域里面的函数能够记住它在被定义的时候它所处的封闭命名空间。这能够通过查看函数的func\_closure属性得出结论，这个属性里面包含封闭作用域里面的值（只会包含被捕捉到的值，比如x，如果在outer里面还定义了其他的值，封闭作用域里面是不会有的)

记住，每次函数outer被调用的时候，函数inner都会被重新定义。现在变量x的值不会变化，所以每次返回的函数inner会是同样的逻辑，假如我们稍微改动一下呢？

Python

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | **def** outer(x):  **def** inner():           print x *# 1*  **return** inner  print1 = outer(1)  print2 = outer(2)  print1()  1  print2()  2 |

从这个例子中你能够看到闭包 – 被函数记住的封闭作用域 – 能够被用来创建自定义的函数，本质上来说是一个硬编码的参数。事实上我们并不是传递参数1或者2给函数inner，我们实际上是创建了能够打印各种数字的各种自定义版本。

闭包单独拿出来就是一个非常强大的功能， 在某些方面，你也许会把它当做一个类似于面向对象的技术：outer像是给inner服务的构造器，x像一个私有变量。使用闭包的方式也有很多：你如果熟悉python内置排序方法的参数key，你说不定已经写过一个lambda方法在排序一个列表的列表的时候基于第二个元素而不是第一个。现在你说不定也可以写一个itemgetter方法，接收一个索引值来返回一个完美的函数，传递给排序函数的参数key。

不过，我们现在不会用闭包做这么low的事(⊙o⊙)…！相反，让我们再爽一次，写一个高大上的装饰器!

****9. 装饰器****

装饰器其实就是一个闭包，把一个函数当做参数然后返回一个替代版函数。我们一步步从简到繁来瞅瞅：

Python

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | **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 |

仔细看看上面这个装饰器的例子。们定义了一个函数outer，它只有一个some\_func的参数，在他里面我们定义了一个嵌套的函数inner。inner会打印一串字符串，然后调用some\_func，在#1处得到它的返回值。在outer每次调用的时候some\_func的值可能会不一样，但是不管some\_func的之如何，我们都会调用它。最后，inner返回some\_func() + 1的值 – 我们通过调用在#2处存储在变量decorated里面的函数能够看到被打印出来的字符串以及返回值2，而不是期望中调用函数foo得到的返回值1。

我们可以认为变量decorated是函数foo的一个装饰版本，一个加强版本。事实上如果打算写一个有用的装饰器的话，我们可能会想愿意用装饰版本完全取代原先的函数foo，这样我们总是会得到我们的”加强版“foo。想要达到这个效果，完全不需要学习新的语法，简单地赋值给变量foo就行了：

Python

|  |  |
| --- | --- |
| 1  2  3 | foo = outer(foo)  foo *# doctest: +ELLIPSIS*  <function inner at 0x> |

现在，任何怎么调用都不会牵扯到原先的函数foo，都会得到新的装饰版本的foo，现在我们还是来写一个有用的装饰器。

想象我们有一个库，这个库能够提供类似坐标的对象，也许它们仅仅是一些x和y的坐标对。不过可惜的是这些坐标对象不支持数学运算符，而且我们也不能对源代码进行修改，因此也就不能直接加入运算符的支持。我们将会做一系列的数学运算，所以我们想要能够对两个坐标对象进行合适加减运算的函数，这些方法很容易就能写出：

Python

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | **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} |

如果不巧我们的加减函数同时也需要一些边界检查的行为那该怎么办呢？搞不好你只能够对正的坐标对象进行加减操作，任何返回的值也都应该是正的坐标。所以现在的期望是这样：

Python

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | 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} |

我们期望在不更改坐标对象one, two, three的前提下one减去two的值是{x: 0, y: 0}，one加上three的值是{x: 100, y: 200}。与其给每个方法都加上参数和返回值边界检查的逻辑，我们来写一个边界检查的装饰器！

Python

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | **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} |

这个装饰器能想先前的装饰器例子一样进行工作，返回一个经过修改的函数，但是在这个例子中，它能够对函数的输入参数和返回值做一些非常有用的检查和格式化工作，将负值的x和 y替换成0。

显而易见，通过这样的方式，我们的代码变得更加简洁：将边界检查的逻辑隔离到单独的方法中，然后通过装饰器包装的方式应用到我们需要进行检查的地方。另外一种方式通过在计算方法的开始处和返回值之前调用边界检查的方法也能够达到同样的目的。但是不可置否的是，使用装饰器能够让我们以最少的代码量达到坐标边界检查的目的。事实上，如果我们是在装饰自己定义的方法的话，我们能够让装饰器应用的更加有逼格。

****10. 使用 @ 标识符将装饰器应用到函数****

Python2.4支持使用标识符@将装饰器应用在函数上，只需要在函数的定义前加上@和装饰器的名称。在上一节的例子里我们是将原本的方法用装饰后的方法代替:

Python

|  |  |
| --- | --- |
| 1 | add = wrapper(add) |

这种方式能够在任何时候对任意方法进行包装。但是如果我们自定义一个方法，我们可以使用@进行装饰：

Python

|  |  |
| --- | --- |
| 1  2  3 | @wrapper  **def** add(a, b):  **return** Coordinate(a.x + b.x, a.y + b.y) |

需要明白的是，这样的做法和先前简单的用包装方法替代原有方法是一毛一样的， python只是加了一些语法糖让装饰的行为更加的直接明确和优雅一点。

****11. \*args and \*\*kwargs****

我们已经完成了一个有用的装饰器，但是由于硬编码的原因它只能应用在一类具体的方法上，这类方法接收两个参数，传递给闭包捕获的函数。如果我们想实现一个能够应用在任何方法上的装饰器要怎么做呢？再比如，如果我们要实现一个能应用在任何方法上的类似于计数器的装饰器，不需要改变原有方法的任何逻辑。这意味着装饰器能够接受拥有任何签名的函数作为自己的被装饰方法，同时能够用传递给它的参数对被装饰的方法进行调用。

非常巧合的是Python正好有支持这个特性的语法。可以阅读 Python Tutorial 获取更多的细节。当定义函数的时候使用了\*，意味着那些通过位置传递的参数将会被放在带有\*前缀的变量中， 所以：

Python

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | **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只是简单地讲任何传递过来的位置参数全部打印出来而已，你们能够看到，在代码#1处我们只是引用了函数内的变量args, \*args仅仅只是用在函数定义的时候用来表示位置参数应该存储在变量args里面。Python允许我们制定一些参数并且通过args捕获其他所有剩余的未被捕捉的位置参数，就像#2处所示的那样。  
\*操作符在函数被调用的时候也能使用。意义基本是一样的。当调用一个函数的时候，一个用\*标志的变量意思是变量里面的内容需要被提取出来然后当做位置参数被使用。同样的，来看个例子：

Python

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | **def** add(x, y):  **return** x + y  lst = [1,2]  add(lst[0], lst[1]) *# 1*  3  add(\*lst) *# 2*  3 |

#1处的代码和#2处的代码所做的事情其实是一样的，在#2处，python为我们所做的事其实也可以手动完成。这也不是什么坏事，\*args要么是表示调用方法大的时候额外的参数可以从一个可迭代列表中取得，要么就是定义方法的时候标志这个方法能够接受任意的位置参数。  
接下来提到的\*\*会稍多更复杂一点，\*\*代表着键值对的餐宿字典，和\*所代表的意义相差无几，也很简单对不对：

Python

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | **def** foo(\*\*kwargs):       print kwargs  foo()  {}  foo(x=1, y=2)  {'y': 2, 'x': 1} |

当我们定义一个函数的时候，我们能够用\*\*kwargs来表明，所有未被捕获的关键字参数都应该存储在kwargs的字典中。如前所诉，argshe kwargs并不是python语法的一部分，但在定义函数的时候，使用这样的变量名算是一个不成文的约定。和\*一样，我们同样可以在定义或者调用函数的时候使用\*\*。

Python

|  |  |
| --- | --- |
| 1  2  3  4  5 | dct = {'x': 1, 'y': 2}  **def** bar(x, y):  **return** x + y  bar(\*\*dct)  3 |

****12. 更通用的装饰器****

有了这招新的技能，我们随随便便就可以写一个能够记录下传递给函数参数的装饰器了。先来个简单地把日志输出到界面的例子：

Python

|  |  |
| --- | --- |
| 1  2  3  4  5 | **def** logger(func):  **def** inner(\*args, \*\*kwargs): *#1*           print "Arguments were: %s, %s" % (args, kwargs)  **return** func(\*args, \*\*kwargs) *#2*  **return** inner |

请注意我们的函数inner，它能够接受任意数量和类型的参数并把它们传递给被包装的方法，这让我们能够用这个装饰器来装饰任何方法。

Python

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | @logger  **def** foo1(x, y=1):  **return** x \* y  @logger  **def** foo2():  **return** 2  foo1(5, 4)  Arguments were: (5, 4), {}  20  foo1(1)  Arguments were: (1,), {}  1  foo2()  Arguments were: (), {}  2 |

随便调用我们定义的哪个方法，相应的日志也会打印到输出窗口，和我们预期的一样。