Python无形装X教程

我们先看一段简单的代码:

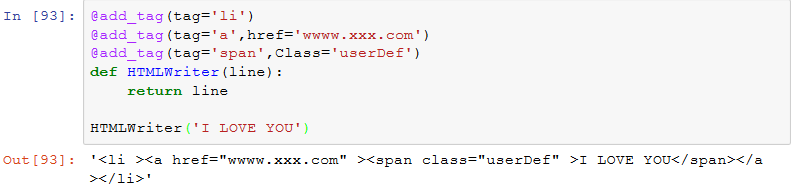


writeHTML这个方程功能及其简单，就是写出一段HTML的code。在读了return的内容后，使用者也明白输入的变量该是什么类型。但这样写法唯一的缺点就是不够复杂，让人一目了然，让人避无可避。很多人看到这个方程的第一印象就是：这个方程在我刚学编程的时候就会写！为了装X，展示Python的基本功，我们可以将这个方程拆开来，写的复杂一点：

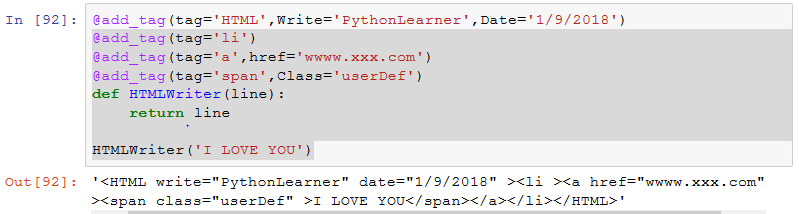


新版的writeHTML方程由四部分构成：：Li、A、Span和writeHTML四个方程。这样看起来似乎高端了一点，但认真的读者在看完writeHTML方程return的内容，在三次往上瞥了几次那三个简单的方程后，会觉得写这个方程的人真的是吃饱了撑的，为什么一个简单的方程要拆成四个方程来写，这样和直接写一个方程有什么区别？装X失败。

那么看这一段代码：



绝对没有人会看懂，但看起来好像功能很强大的样子，而且思路绝对清晰，我们还能再加一些东西：



怎么教会别人怎么用呢？你只需要告诉别人，写一个@add\_tag并输入tag和这个tag相应的属性，就好了。而且你在讲的时候一定要这样：



然后转身离开，不带走一片云彩，任凭留下男人嫉妒的眼光，和小女孩崇拜的眼神。装X成功！那么会有读者朋友问了，这个充满B格的@XXX呢？怎么才能写出这样装X气息十足的代码呢？

好了，这就引入了我们这期的主题：怎么才能用Python正确地无形装X呢？我们就要引入Python中的装X神器——修饰器（decorator）的概念了。关于上文中@add\_tag的代码，将贴在文末。

问题一：什么是修饰器？

这个问题要分两部分回答。对于方程型修饰器来说，就是方程的方程，我们先看一个简单的例子：



我们首先先看方程型修饰器的构建方法，我们建立修饰方程dec， 在dec中构件本地方程wrapper，wrapper可以接受任意的变量(arguments)和有关键词的变量(keyword arguments)<参阅小贴士1>。修饰方程的返回值是wrapper，是一个方程。我们要求输入修饰方程的变量fun是callable的object。<参阅小贴士2>。这个修饰器的功能是在每次使用被修饰的方程时都会在输出窗口打印出一行“deced”。

在写了修饰器后，我们再定义一个需要要被修饰的方程func1，它的功能是打印出输入的变量。在定义func1的时候，我们用@dec来添加修饰器，再call 修饰完的方程。如我们所描述的一样，先是打印出“deced”然后才是被修饰方程所展示的内容。

这样解释是比较有逼格的解释方法，下面我们用朴素的方法再解释一下。

究竟什么是修饰器？其实修饰器就是一个方程，它的输入也是方程，返回值是另一个方程，返回值中的方程要以满足以下条件为根本，根据需求增加输入方程的功能：

1. 能够实现与输入方程同样的功能；
2. 能够实现于输入方程同样的输出；
3. 能够接收与输入方程相同的输入。

实现的具体方法就是，首先写一个修饰方程。写修饰方程的方法与上述相同。然后我们再写另一个方程fun2，怎么修饰这个方程呢？我们以同样的名字fun2来储存修饰方程以fun2为变量的返回值，这样fun2就变成了修饰完毕的方程，我们看以下代码来展现修饰完毕的结果：

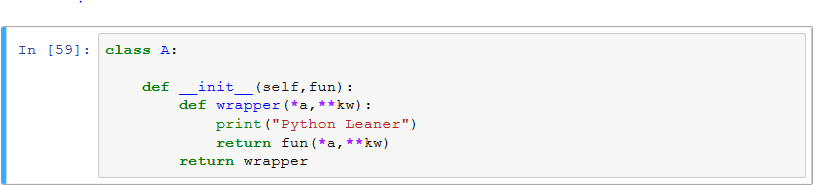


与第一个结果一样，fun2也被成功修饰了！我们用细细品位这个过程(<local>标签意为内嵌方程)：

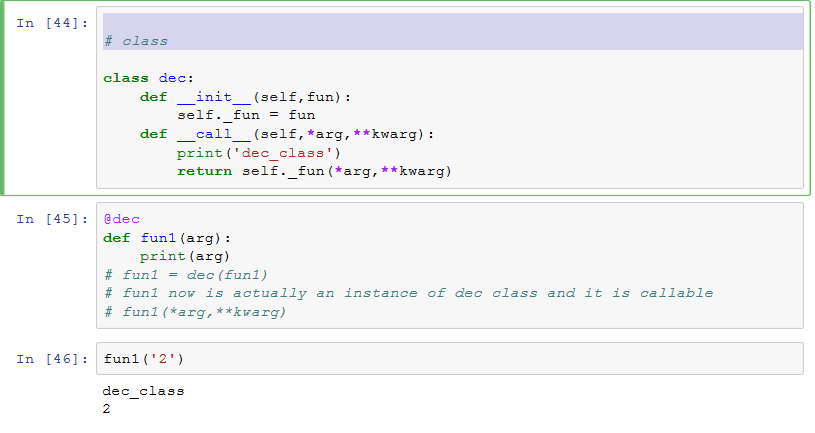
1. dec(fun2) 返回值为wrapper<local>，是一个方程
2. 返回值wrapper<local>被与fun2这个identifier对应起来，于是fun2 的值就是wrapper<local>，此时fun2是一个方程
3. fun(‘dec-decorated’) 其实就是wrapper(‘dec-decorated’)<local>
4. wrapper(‘dec-decorated’)的返回值为fun2(‘dec-decorated’)，但在wrapper<local> return前，print出”deced”。

所以修饰器的功能就是将输入的方程(fun2)变为另一个方程(wrapper)，我们只需确保这个方程(wrapper)的返回值与原方程是一致的，并且能够接受原方程的参量。为此，我们将wrapper的返回值设为对原方程的运行，将wrapper的参量变为任意参量，我们就完美满足了修饰器所要满足的三个条件。

截至到目前为止，我们知道function能够当修饰器，那么class能当修饰器吗？答案是肯定的。下面的代码可以实现吗？



当然不能，首先是\_\_init\_\_不能有返回值，其次是就算这个class能够实现，A(fun)是class A的一个instance，在没有定义\_\_call\_\_ method的情况下，A(fun)不是callable的。我们可以看到function能当修饰器的原因之一是因为function是天生callable的，因此，我们需要对class A的instance定义call的method，使它能够成为callable。具体的实现代码如下：

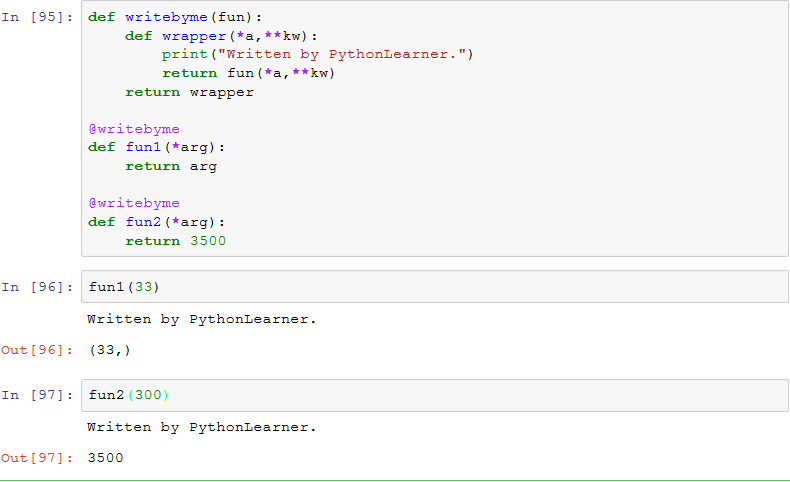


我们用同样的办法将这些步骤来分开(第二个框的注释部分)：

1. dec(fun1) 是class dec的一个instance，我们在class dec中定义了instance \_\_call\_\_的方法，因此dec(fun1)现在是callable的object。
2. dec(fun1)被对应于identifier fun1，因此fun1(‘2’) 就是dec(fun1)(‘2’)。
3. 通过\_\_call\_\_方法的定义我们不难预测出输出值。

截至到目前为止，我们已经知道了修饰器的写法和使用方法。但也只是入门级别的概念，绝对不够用来装X。至今我们的修饰器只能有一个参量，输入的fun。既然修饰器是一个方程，我们能不能添加别的参量？当然可以，这就涉及另外一层修饰，我们称之为修饰器的修饰器(decorated decorator)。

假设我们写了很多方程，我们想别的使用者在使用的时候知道这个方程是我们写的，在每个方程中写一个print(“Written by XXX”)太明显，也太傻X。我们可以用修饰器的方法如下：



正当你看着自己的杰作而沾沾自喜，恰巧有一个妹子用了你的方程，她发现只要在自己写的function前面写一个@writebyme就行了，结果试了几下，结果发现用了你的@writebyme就会输出你的名字，而不是她的名字！她会说什么？“什么鬼东西，傻X”。好吧，我们这时候需要更强大的修饰器了，首先先看以下代码：



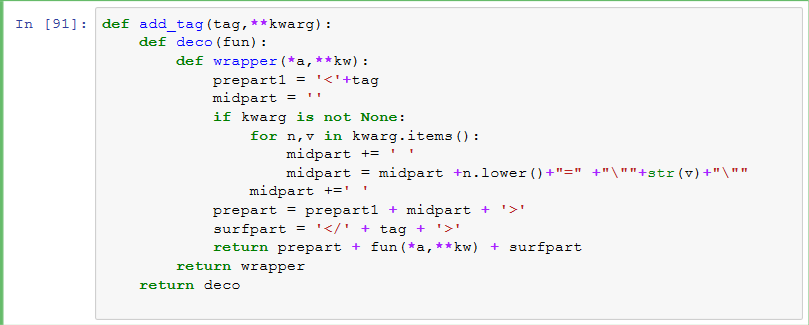
我们在decorator的外层又加了一层函数，而这个函数的返回值是这个decorator！这个函数支持自由参量的输入。我们继续拆分一下实现步骤：

1. Wirteby是一个function，它支持的变量是关键词变量。
2. Writeby(user = ‘Pythonlearner’,date=’01/09/2018’) 返回值是deco，deco是一个内嵌方程。它是一个完整的修饰器。
3. 在加上@后，fun1 变为wirteby(user = ‘Pythonlearner’,date=’01/09/2018’)(fun1)，也就是deco(fun1)，也就是wrapper。
4. 接下来的解释与修饰器的部分相同。

也就是说，我们给修饰器又加了一层修饰，让他本身的功能更强大了。至此，我们对装X利器Decorator的初级讲解就完了。

为什么是初级呢？因为你根本还没了解透Python里的function啊，怎么能写出更复杂的decorator呢！敬请期待下一期，手把手教你玩坏Python方程！

现在我们可以贴上文章初始部分的代码了，希望读者能够自行拆分，理解decorator的运行机制，开启装X之路！

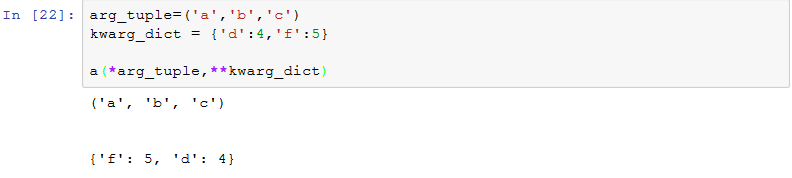


小贴士1：变量(arguments) 和 关键词变量(keyword arguments)。

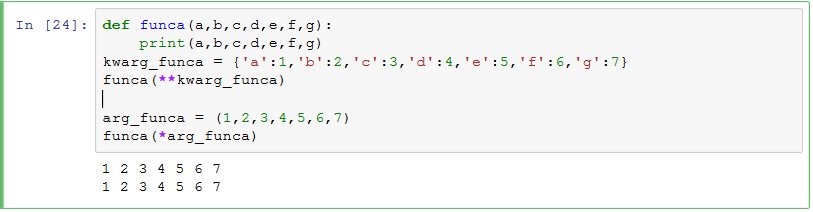
这是Python方程可以接受的变量的两个形式。



我们在这里构建了一个方程a，他的功能是打印出变量和关键词变量，我们可以看到a在接收’a’, ’b’, ’c’这三个自由变量的时候，其实是把他们放在一个tuple中进行储存的，而对于关键词变量，则是以dict的形式储存的，而这个dict中的关键词正是变量的名称，而值正是相对应变量所被赋予的值。于是我们可以引申出下面传递方程变量的方法：

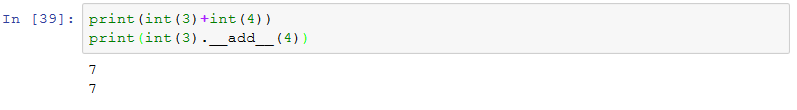


这种做法的好处是如果一个方程有太多的输入（inputs）或许多方程有相同的参量的时候，能大大节省输入的时间成本、减少输入错误的概率和减轻参量赋值的管理负担。



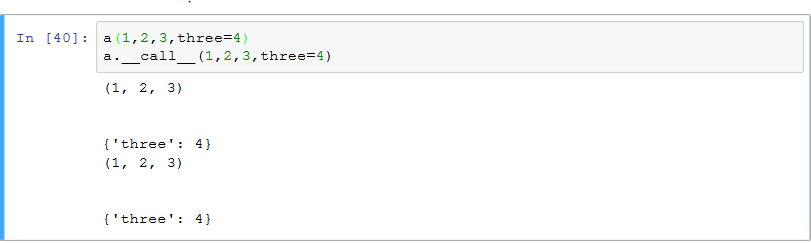
小贴士2：什么是callable？

我们知道Python中有+、-等数学操作符号，那么这些操作是怎么定义和实现的呢？我们称这种方法为运算符重载(operator overloading)，实现的具体过程是通过一些特殊命名的方法(Specially named method)。例如最简单的加法：

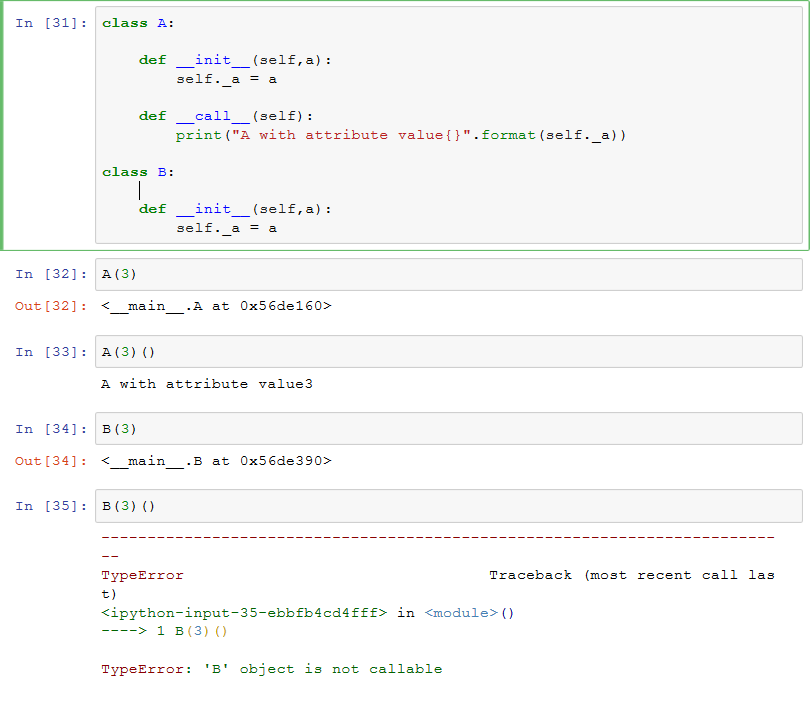


“+”这个操作具体实现过程实际上是通过一个\_\_add\_\_的方法定义的。我们知道int其实是一个类(class)。那么\_\_add\_\_也仅是定义于int这个类中的一种方法，只不过这种方法是特殊命名并由系统保留并指定功能的，“+”是重载的运算符(operator)。

我们还用小贴士1中的方程a，那么与使用方程的a对应的特殊命名的方法是什么呢？就是\_\_call\_\_。我们在写出方程的名字a，再在()里输入变量，其实()就是重载\_\_call\_\_方法的运算符。



我们称定义了\_\_call\_\_这种method的object为callable。再看下面例子：



这里我们定义了class A和class B。A中我们定义了instance method \_\_call\_\_ 。通过对\_\_call\_\_的定义，我们成功将A的一个instance A(3)变成了callable 的object。但没有进行此定义的B的instance就不是一个callable的object。