**理解Python命名机制**

本文最初发表于恋花蝶的博客（http://blog.csdn.net/lanphaday），欢迎转载，但必须保留此声明且不得用于商业目的。谢谢。

**引子**

我热情地邀请大家猜测下面这段程序的输出：

class A(object):

       def \_\_init\_\_(self):

              self.\_\_private()

              self.public()

       def \_\_private(self):

              print 'A.\_\_private()'

       def public(self):

              print 'A.public()'

class B(A):

       def \_\_private(self):

              print 'B.\_\_private()'

       def public(self):

              print 'B.public()'

b = B()

**初探**

正确的答案是：

A.\_\_private()

B.public()

如果您已经猜对了，那么可以不看我这篇博文了。如果你没有猜对或者心里有所疑问，那我的这篇博文正是为您所准备的。

一切由为什么会输出“A.\_\_private()”开始。但要讲清楚为什么，我们就有必要了解一下Python的命名机制。

据 Python manual，变量名（标识符）是Python的一种原子元素。当变量名被绑定到一个对象的时候，变量名就指代这个对象，就像人类社会一样，不是吗？当变量名出现在代码块中，那它就是本地变量；当变量名出现在模块中，它就是全局变量。模块相信大家都有很好的理解，但代码块可能让人费解些。在这里解释一下：

代码块就是可作为可执行单元的一段Python程序文本；模块、函数体和类定义都是代码块。不仅如此，每一个交互脚本命令也是一个代码块；一个脚本文件也是一个代码块；一个命令行脚本也是一个代码块。

接下来谈谈变量的可见性，我们引入一个范围的概念。范围就是变量名在代码块的可见性。如果一个代码块里定义本地变量，那范围就包括这个代码块。如果变量定义在一个功能代码块里，那范围就扩展到这个功能块里的任一代码块，除非其中定义了同名的另一变量。但定义在类中的变量的范围被限定在类代码块，而不会扩展到方法代码块中。

**迷踪**

据上节的理论，我们可以把代码分为三个代码块：类A的定义、类B的定义和变量b的定义。根据类定义，我们知道代码给类A定义了三个成员变量（Python的函数也是对象，所以成员方法称为成员变量也行得通。）；类B定义了两个成员变量。这可以通过以下代码验证：

>>> print '/n'.join(dir(A))

\_A\_\_private

\_\_init\_\_

public

>>> print '/n'.join(dir(B))

\_A\_\_private

\_B\_\_private

\_\_init\_\_

public

咦，为什么类A有个名为\_A\_\_private的 Attribute 呢？而且\_\_private消失了！这就要谈谈Python的私有变量轧压了。

**探究**

懂Python的朋友都知道Python把以两个或以上下划线字符开头且没有以两个或以上下划线结尾的变量当作私有变量。私有变量会在代码生成之前被转换为长格式（变为公有）。转换机制是这样的：在变量前端插入类名，再在前端加入一个下划线字符。这就是所谓的私有变量轧压（Private name mangling）。如类A里的\_\_private标识符将被转换为\_A\_\_private，这就是上一节出现\_A\_\_private和\_\_private消失的原因了。

再讲两点题外话：

一是因为轧压会使标识符变长，当超过255的时候，Python会切断，要注意因此引起的命名冲突。

二是当类名全部以下划线命名的时候，Python就不再执行轧压。如：

>>> class \_\_\_\_(object):

       def \_\_init\_\_(self):

              self.\_\_method()

       def \_\_method(self):

              print '\_\_\_\_.\_\_method()'

>>> print '/n'.join(dir(\_\_\_\_))

\_\_class\_\_

\_\_delattr\_\_

\_\_dict\_\_

\_\_doc\_\_

\_\_getattribute\_\_

\_\_hash\_\_

\_\_init\_\_

\_\_method              # 没被轧压

\_\_module\_\_

\_\_new\_\_

\_\_reduce\_\_

\_\_reduce\_ex\_\_

\_\_repr\_\_

\_\_setattr\_\_

\_\_str\_\_

\_\_weakref\_\_

>>> obj = \_\_\_\_()

\_\_\_\_.\_\_method()

>>> obj.\_\_method()      # 可以外部调用

\_\_\_\_.\_\_method()

现在我们回过头来看看为什么会输出“A.\_\_private()”吧！

**真相**

相信现在聪明的读者已经猜到答案了吧？如果你还没有想到，我给你个提示：真相跟C语言里的宏预处理差不多。

因为类A定义了一个私有成员函数（变量），所以在代码生成之前先执行私有变量轧压（注意到上一节标红的那行字没有？）。轧压之后，类A的代码就变成这样了：

class A(object):

       def \_\_init\_\_(self):

              self.\_A\_\_private()          # 这行变了

              self.public()

       def \_A\_\_private(self):           # 这行也变了

              print 'A.\_\_private()'

       def public(self):

              print 'A.public()'

是不是有点像C语言里的宏展开啊？

因为在类B定义的时候没有覆盖\_\_init\_\_方法，所以调用的仍然是A.\_\_init\_\_，即执行了self.\_A\_\_private()，自然输出“A.\_\_private()”了。

下面的两段代码可以增加说服力，增进理解：

>>> class C(A):

       def \_\_init\_\_(self):          # 重写\_\_init\_\_，不再调用self.\_A\_\_private

              self.\_\_private()       # 这里绑定的是\_C\_private

              self.public()

       def \_\_private(self):

              print 'C.\_\_private()'

       def public(self):

              print 'C.public()'

>>> c = C()

C.\_\_private()

C.public()

############################

>>> class A(object):

       def \_\_init\_\_(self):

              self.\_A\_\_private()   # 调用一个没有定义的函数，Python会把它给我的 ^\_^～

              self.public()

       def \_\_private(self):

              print 'A.\_\_private()'

       def public(self):

              print 'A.public()'

>>>a = A()

A.\_\_private()

A.public()