# python基础（5）：深入理解 python 中的赋值、引用、拷贝、作用域

发表于2年前(2013-07-21 01:14)   阅读（7019） | 评论（[3](http://my.oschina.net/leejun2005/blog/145911#comments)） **16**人收藏此文章, [我要收藏](javascript:add_to_favor(145911,3))

赞4

[**8月22日珠海 OSC 源创会正在报名，送机械键盘和开源无码内裤**](http://city.oschina.net/zhuhai/event/244519?channel=text)  http://my.oschina.net/img/hot3.png?t=1435877996000

目录[-]

 [1、先来看个问题吧：](http://my.oschina.net/leejun2005/blog/145911#OSC_h3_1)

 [2、引用 VS 拷贝：](http://my.oschina.net/leejun2005/blog/145911#OSC_h3_2)

 [3、增强赋值以及共享引用：](http://my.oschina.net/leejun2005/blog/145911#OSC_h3_3)

 [4、python 从 2k 到 3k，语句变函数引发的变量作用域问题](http://my.oschina.net/leejun2005/blog/145911#OSC_h3_4)

 [5、深入理解 python 变量作用域及其陷阱](http://my.oschina.net/leejun2005/blog/145911#OSC_h3_5)

 [5.1 可变对象 & 不可变对象](http://my.oschina.net/leejun2005/blog/145911#OSC_h4_6)

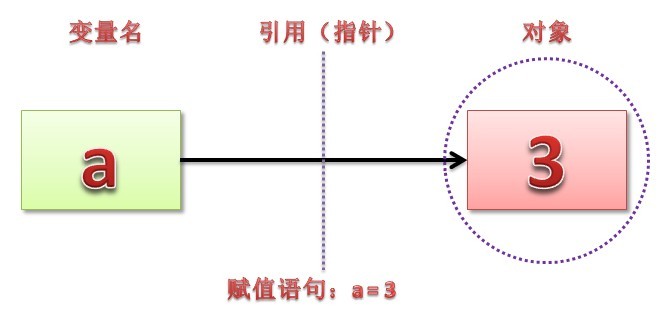
 [5.2 函数值传递](http://my.oschina.net/leejun2005/blog/145911#OSC_h4_7)

 [5.3 为什么修改全局的dict变量不用global关键字](http://my.oschina.net/leejun2005/blog/145911#OSC_h4_8)

 [5.4 可变对象 list 的 = 和 append/extend 差别在哪？](http://my.oschina.net/leejun2005/blog/145911#OSC_h4_9)

 [5.5 陷阱：使用可变的默认参数](http://my.oschina.net/leejun2005/blog/145911#OSC_h4_10)

在 python 中赋值语句总是建立对象的引用值，而不是复制对象。因此，python 变量更像是指针，而不是数据存储区域，

[](http://static.oschina.net/uploads/space/2013/0722/011541_xE4k_568818.jpg)

这点和大多数 OO 语言类似吧，比如 C++、java 等 ~

### 1、先来看个问题吧：

[在Python中，令values=[0,1,2];values[1]=values,为何结果是[0,[...],2]?](http://www.zhihu.com/question/21000872)

[?](http://my.oschina.net/leejun2005/blog/145911)

|  |  |
| --- | --- |
| 1  2  3  4 | >>> values = [0, 1, 2]  >>> values[1] = values  >>> values  [0, [...], 2] |

我预想应当是

[0, [0, 1, 2], 2]

但结果却为何要赋值无限次？

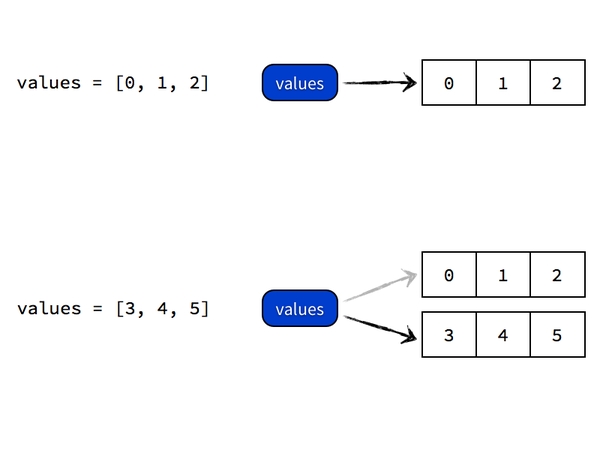
可以说 Python 没有赋值，只有引用。你这样相当于创建了一个引用自身的结构，所以导致了无限循环。为了理解这个问题，有个基本概念需要搞清楚。  
  
Python 没有「变量」，我们平时所说的变量其实只是「标签」，是引用。

执行

values = [0, 1, 2]

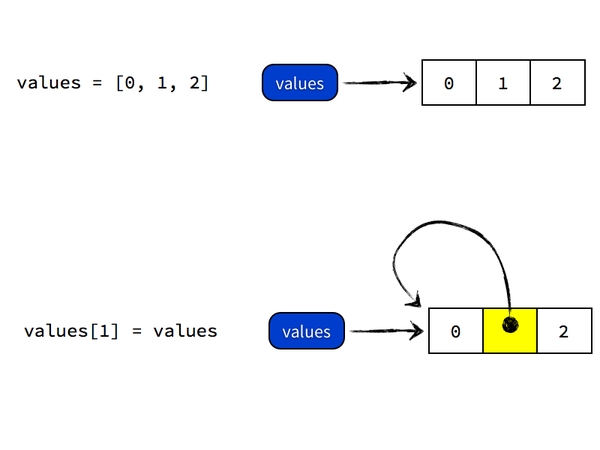
的时候，Python 做的事情是首先创建一个列表对象 [0, 1, 2]，然后给它贴上名为 values 的标签。如果随后又执行

values = [3, 4, 5]

的话，Python 做的事情是创建另一个列表对象 [3, 4, 5]，然后把刚才那张名为 values 的标签从前面的 [0, 1, 2] 对象上撕下来，重新贴到 [3, 4, 5] 这个对象上。   
  
至始至终，并没有一个叫做 values 的列表对象容器存在，Python 也没有把任何对象的值复制进 values 去。过程如图所示：   
[](http://static.oschina.net/uploads/img/201307/21011458_e6oW.jpg)

执行

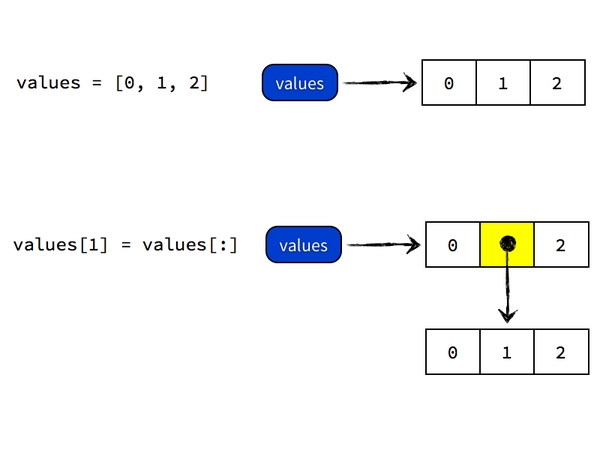
values[1] = values

的时候，Python 做的事情则是把 values 这个标签所引用的列表对象的第二个元素指向 values 所引用的列表对象本身。执行完毕后，values 标签还是指向原来那个对象，只不过那个对象的结构发生了变化，从之前的列表 [0, 1, 2] 变成了 [0, ?, 2]，而这个 ? 则是指向那个对象本身的一个引用。如图所示：   
[](http://static.oschina.net/uploads/img/201307/21011459_8n1Y.jpg)   
要达到你所需要的效果，即得到 [0, [0, 1, 2], 2] 这个对象，你不能直接将 values[1] 指向 values 引用的对象本身，而是需要吧 [0, 1, 2] 这个对象「复制」一遍，得到一个新对象，再将 values[1] 指向这个复制后的对象。Python 里面复制对象的操作因对象类型而异，复制列表 values 的操作是

values[:] #生成对象的拷贝或者是复制序列，不再是引用和共享变量，但此法只能顶层复制

所以你需要执行

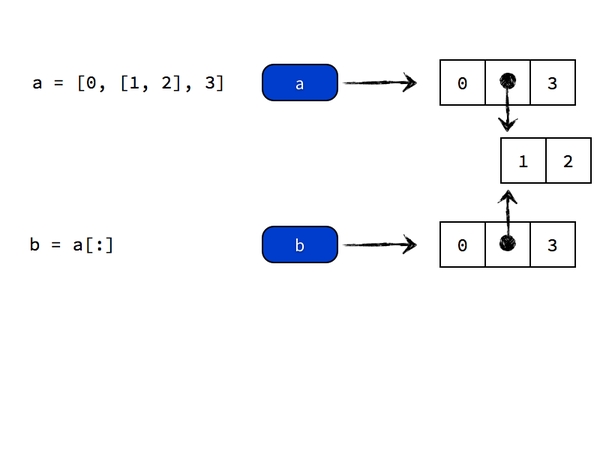
values[1] = values[:]

Python 做的事情是，先 dereference 得到 values 所指向的对象 [0, 1, 2]，然后执行 [0, 1, 2][:] 复制操作得到一个新的对象，内容也是 [0, 1, 2]，然后将 values 所指向的列表对象的第二个元素指向这个复制二来的列表对象，最终 values 指向的对象是 [0, [0, 1, 2], 2]。过程如图所示：   
[](http://static.oschina.net/uploads/img/201307/21011459_EJcH.jpg)

往更深处说，values[:] 复制操作是所谓的「浅复制」(shallow copy)，当列表对象有嵌套的时候也会产生出乎意料的错误，比如

[?](http://my.oschina.net/leejun2005/blog/145911)

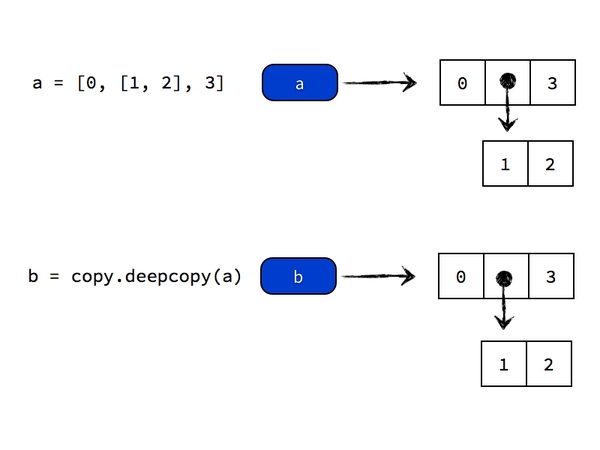
|  |  |
| --- | --- |
| 1  2  3  4 | a = [0, [1, 2], 3]  b = a[:]  a[0] = 8  a[1][1] = 9 |

问：此时 a 和 b 分别是多少？   
  
正确答案是 a 为 [8, [1, 9], 3]，b 为 [0, [1, 9], 3]。发现没？b 的第二个元素也被改变了。想想是为什么？不明白的话看下图   
[](http://static.oschina.net/uploads/img/201307/21011459_MqCC.jpg)

正确的复制嵌套元素的方法是进行「深复制」(deep copy)，方法是

[?](http://my.oschina.net/leejun2005/blog/145911)

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | import copy    a = [0, [1, 2], 3]  b = copy.deepcopy(a)  a[0] = 8  a[1][1] = 9 |

[](http://static.oschina.net/uploads/img/201307/21011459_E4CB.jpg)

### 2、引用 VS 拷贝：

（1）没有限制条件的分片表达式（L[:]）能够复制序列，但此法只能浅层复制。

（2）字典 copy 方法，D.copy() 能够复制字典，但此法只能浅层复制

（3）有些内置函数，例如 list，能够生成拷贝 list(L)

（4）copy 标准库模块能够生成完整拷贝：deepcopy 本质上是递归 copy

（5）对于不可变对象和可变对象来说，浅复制都是复制的引用，只是因为复制不变对象和复制不变对象的引用是等效的（因为对象不可变，当改变时会新建对象重新赋值）。所以看起来浅复制只复制不可变对象（整数，实数，字符串等），对于可变对象，浅复制其实是创建了一个对于该对象的引用，也就是说只是给同一个对象贴上了另一个标签而已。

[?](http://my.oschina.net/leejun2005/blog/145911)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26 | L = [1, 2, 3]  D = {'a':1, 'b':2}  A = L[:]  B = D.copy()  print "L, D"  print  L, D  print "A, B"  print A, B  print "--------------------"  A[1] = 'NI'  B['c'] = 'spam'  print "L, D"  print  L, D  print "A, B"  print A, B      L, D  [1, 2, 3] {'a': 1, 'b': 2}  A, B  [1, 2, 3] {'a': 1, 'b': 2}  --------------------  L, D  [1, 2, 3] {'a': 1, 'b': 2}  A, B  [1, 'NI', 3] {'a': 1, 'c': 'spam', 'b': 2} |

### 3、增强赋值以及共享引用：

x = x + y，x 出现两次，必须执行两次，性能不好，合并必须新建对象 x，然后复制两个列表合并

属于复制/拷贝

x += y，x 只出现一次，也只会计算一次，性能好，不生成新对象，只在内存块末尾增加元素。

当 x、y 为list时， += 会自动调用 extend 方法进行合并运算，in-place change。

属于共享引用

[?](http://my.oschina.net/leejun2005/blog/145911)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | L = [1, 2]  M = L  L = L + [3, 4]  print L, M  print "-------------------"  L = [1, 2]  M = L  L += [3, 4]  print L, M      [1, 2, 3, 4] [1, 2]  -------------------  [1, 2, 3, 4] [1, 2, 3, 4] |

### 4、python 从 2k 到 3k，语句变函数引发的变量作用域问题

先看段代码：

[?](http://my.oschina.net/leejun2005/blog/145911)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | def test():      a = False      exec ("a = True")      print ("a = ", a)  test()    b = False  exec ("b = True")  print ("b = ", b) |

在 python 2k 和 3k 下 你会发现他们的结果不一样：

[?](http://my.oschina.net/leejun2005/blog/145911)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | 2K：  a =  True  b =  True    3K：  a =  False  b =  True |

这是为什么呢？

因为 3k 中 exec 由语句变成函数了，而在函数中变量默认都是局部的，也就是说

你所见到的两个 a，是两个不同的变量，分别处于不同的命名空间中，而不会冲突。

具体参考 《learning python》P331-P332

知道原因了，我们可以这么改改：

[?](http://my.oschina.net/leejun2005/blog/145911)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | def test():      a = False      ldict = locals()      exec("a=True",globals(),ldict)      a = ldict['a']      print(a)    test()    b = False  exec("b = True", globals())  print("b = ", b) |

这个问题在  stackoverflow 上已经有人问了，而且 python 官方也有人报了 bug。。。

具体链接在下面：

<http://stackoverflow.com/questions/7668724/variables-declared-in-execed-code-dont-become-local-in-python-3-documentatio>

<http://bugs.python.org/issue4831>

<http://stackoverflow.com/questions/1463306/how-does-exec-work-with-locals>

这是一个典型的 python 2k 移植到 3k 不兼容的案例，类似的还有很多，也算是移植的坑吧~

具体的 2k 与 3k 有哪些差异可以看这里：

使用 2to3 将代码移植到 Python 3

<http://woodpecker.org.cn/diveintopython3/porting-code-to-python-3-with-2to3.html>

### 5、深入理解 python 变量作用域及其陷阱

#### 5.1 可变对象 & 不可变对象

在Python中，对象分为两种：可变对象和不可变对象，不可变对象包括int，float，long，str，tuple等，可变对象包括list，set，dict等。需要注意的是：这里说的不可变指的是值的不可变。对于不可变类型的变量，如果要更改变量，则会创建一个新值，把变量绑定到新值上，而旧值如果没有被引用就等待垃圾回收。另外，不可变的类型可以计算hash值，作为字典的key。可变类型数据对对象操作的时候，不需要再在其他地方申请内存，只需要在此对象后面连续申请(+/-)即可，也就是它的内存地址会保持不变，但区域会变长或者变短。

[?](http://my.oschina.net/leejun2005/blog/145911)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | >>> a = 'xianglong.me'  >>> id(a)  140443303134352  >>> a = '1saying.com'  >>> id(a)  140443303131776  # 重新赋值之后，变量a的内存地址已经变了  # 'xianglong.me'是str类型，不可变，所以赋值操作知识重新创建了str '1saying.com'对象，然后将变量a指向了它    >>> a\_list = [1, 2, 3]  >>> id(a\_list)  140443302951680  >>> a\_list.append(4)  >>> id(a\_list)  140443302951680  # list重新赋值之后，变量a\_list的内存地址并未改变  # [1, 2, 3]是可变的，append操作只是改变了其value，变量a\_list指向没有变 |

#### 5.2 函数值传递

[?](http://my.oschina.net/leejun2005/blog/145911)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | def func\_int(a):      a += 4    def func\_list(a\_list):      a\_list[0] = 4    t = 0  func\_int(t)  print t  # output: 0    t\_list = [1, 2, 3]  func\_list(t\_list)  print t\_list  # output: [4, 2, 3] |

 对于上面的输出，不少Python初学者都比较疑惑：第一个例子看起来像是传值，而第二个例子确实传引用。其实，解释这个问题也非常容易，主要是因为可变对象和不可变对象的原因：对于可变对象，对象的操作不会重建对象，而对于不可变对象，每一次操作就重建新的对象。

    在函数参数传递的时候，Python其实就是把参数里传入的变量对应的对象的引用依次赋值给对应的函数内部变量。参照上面的例子来说明更容易理解，func\_int中的局部变量"a"其实是全部变量"t"所指向对象的另一个引用，由于整数对象是不可变的，所以当func\_int对变量"a"进行修改的时候，实际上是将局部变量"a"指向到了整数对象"1"。所以很明显，func\_list修改的是一个可变的对象，局部变量"a"和全局变量"t\_list"指向的还是同一个对象。

#### 5.3 为什么修改全局的dict变量不用global关键字

为什么修改字典d的值不用global关键字先声明呢？

[?](http://my.oschina.net/leejun2005/blog/145911)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | s = 'foo'  d = {'a':1}  def f():      s = 'bar'      d['b'] = 2  f()  print s  # foo  print d  # {'a': 1, 'b': 2} |

这是因为，在s = 'bar'这句中，它是“有歧义的“，因为它既可以是表示引用全局变量s，也可以是创建一个新的局部变量，所以在python中，默认它的行为是创建局部变量，除非显式声明global，global定义的本地变量会变成其对应全局变量的一个别名，即是同一个变量。

在d['b']=2这句中，它是“明确的”，因为如果把d当作是局部变量的话，它会报KeyError，所以它只能是引用全局的d,故不需要多此一举显式声明global。

上面这两句赋值语句其实是不同的行为，一个是**rebinding（不可变对象）**, 一个是**mutation（可变对象）**.

但是如果是下面这样：

[?](http://my.oschina.net/leejun2005/blog/145911)

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | d = {'a':1}  def f():      d = {}      d['b'] = 2  f()  print d  # {'a': 1} |

在d = {}这句，它是”有歧义的“了，所以它是创建了局部变量d，而不是引用全局变量d，所以d['b']=2也是操作的局部变量。

推而远之，这一切现象的本质就是”它是否是明确的“。

仔细想想，就会发现不止dict不需要global，所有”明确的“东西都不需要global。因为int类型str类型之类的不可变对象，每一次操作就重建新的对象，他们只有一种修改方法，即x = y， 恰好这种修改方法同时也是创建变量的方法，所以产生了歧义，不知道是要修改还是创建。而dict/list/对象等可变对象，操作不会重建对象，可以通过dict['x']=y或list.append()之类的来修改，跟创建变量不冲突，不产生歧义，所以都不用显式global。

#### 5.4 可变对象 list 的 = 和 append/extend 差别在哪？

接上面 5.3 的理论，下面咱们再看一例常见的错误：

[?](http://my.oschina.net/leejun2005/blog/145911)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | # coding=utf-8  # 测试utf-8编码  import sys  reload(sys)  sys.setdefaultencoding('utf-8')    list\_a = []  def a():      list\_a = [1]      ## 语句1  a()  print list\_a    # []    print "======================"    list\_b = []  def b():      list\_b.append(1)    ## 语句2  b()  print list\_b    # [1] |

大家可以看到为什么 语句1 不能改变 list\_a 的值，而 语句2 却可以？他们的差别在哪呢？

**因为 = 创建了局部变量，而 .append() 或者 .extend() 重用了全局变量。**

#### 5.5 陷阱：使用可变的默认参数

我多次见到过如下的代码：

[?](http://my.oschina.net/leejun2005/blog/145911)

|  |  |
| --- | --- |
| 1  2  3 | def foo(a, b, c=[]):  # append to c  # do some more stuff |

永远不要使用可变的默认参数，可以使用如下的代码代替：

[?](http://my.oschina.net/leejun2005/blog/145911)

|  |  |
| --- | --- |
| 1  2  3  4  5 | def foo(a, b, c=None):      if c is None:          c = []      # append to c      # do some more stuff |

与其解释这个问题是什么，不如展示下使用可变默认参数的影响：

[?](http://my.oschina.net/leejun2005/blog/145911)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | In[2]: def foo(a, b, c=[]):  ...        c.append(a)  ...        c.append(b)  ...        print(c)  ...  In[3]: foo(1, 1)  [1, 1]  In[4]: foo(1, 1)  [1, 1, 1, 1]  In[5]: foo(1, 1)  [1, 1, 1, 1, 1, 1] |

同一个变量c在函数调用的每一次都被反复引用。这可能有一些意想不到的后果。

REF：

[1] 《learning python》：P130、P134、P202、P204 、P245

<http://www.zhihu.com/question/21000872/answer/16856382>

[2] 理解 Python 的 LEGB

<http://blog.segmentfault.com/sunisdown/1190000000640834>

[3] Python函数参数默认值的陷阱和原理深究

<http://cenalulu.github.io/python/default-mutable-arguments/>

[4] 潜在的Python陷阱

<http://python.jobbole.com/81564/>

[5] 陷阱！python参数默认值

<http://segmentfault.com/a/1190000000743526>

[6] Python中的变量、引用、拷贝和作用域

<http://xianglong.me/article/python-variable-quote-copy-and-scope/>

[7] Python入门基础知识(1) :locals() 和globals()

<http://www.cnblogs.com/wanxsb/archive/2013/05/07/3064783.html>

# 详解 Python 中的下划线命名规则

发表于5个月前(2015-03-15 00:01)   阅读（294） | 评论（[0](http://my.oschina.net/leejun2005/blog/387136#comments)） **26**人收藏此文章, [我要收藏](javascript:add_to_favor(387136,3))

赞0

[**8月22日珠海 OSC 源创会正在报名，送机械键盘和开源无码内裤**](http://city.oschina.net/zhuhai/event/244519?channel=text)  http://my.oschina.net/img/hot3.png?t=1435877996000

目录[-]

 [1、单下划线（\_）](http://my.oschina.net/leejun2005/blog/387136#OSC_h1_1)

 [1.1 在解释器中：](http://my.oschina.net/leejun2005/blog/387136#OSC_h2_2)

 [1.2 作为一个名称：](http://my.oschina.net/leejun2005/blog/387136#OSC_h2_3)

 [1.3 国际化：](http://my.oschina.net/leejun2005/blog/387136#OSC_h2_4)

 [2、名称前的单下划线（如：\_shahriar）](http://my.oschina.net/leejun2005/blog/387136#OSC_h1_5)

 [3、名称前的双下划线（如：\_\_shahriar）](http://my.oschina.net/leejun2005/blog/387136#OSC_h1_6)

 [4、名称前后的双下划线（如：\_\_init\_\_）](http://my.oschina.net/leejun2005/blog/387136#OSC_h1_7)

 [5、题外话 if \_\_name\_\_ == "\_\_main\_\_":](http://my.oschina.net/leejun2005/blog/387136#OSC_h1_8)

 [6、总结：](http://my.oschina.net/leejun2005/blog/387136#OSC_h1_9)

 [7、Refer：](http://my.oschina.net/leejun2005/blog/387136#OSC_h1_10)

在 python 中，下划线命名规则往往令初学者相当疑惑：单下划线、双下划线、双下划线还分前后……那它们的作用与使用场景到底有何区别呢？今天就来聊聊这个话题。

# 1、单下划线（\_）

通常情况下，单下划线（\_）会在以下3种场景中使用： 

## 1.1 在解释器中：

在这种情况下，“\_”代表交互式解释器会话中上一条执行的语句的结果。这种用法首先被标准CPython解释器采用，然后其他类型的解释器也先后采用。

[?](http://my.oschina.net/leejun2005/blog/387136)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | >>> \_ Traceback (most recent call last):  File "<stdin>", line 1, in <module>  NameError: name '\_' is not defined  >>> 42  >>> \_  42  >>> 'alright!' if \_ else ':('  'alright!'  >>> \_  'alright!' |

## 1.2 作为一个名称：

这与上面一点稍微有些联系，此时“\_”作为临时性的名称使用。这样，当其他人阅读你的代码时将会知道，你分配了一个特定的名称，但是并不会在后面再次用到该名称。例如，下面的例子中，你可能对循环计数中的实际值并不感兴趣，此时就可以使用“\_”。

[?](http://my.oschina.net/leejun2005/blog/387136)

|  |  |
| --- | --- |
| 1  2  3 | n = 42  for \_ in range(n):      do\_something() |

## 1.3 国际化：

也许你也曾看到”\_“会被作为一个函数来使用。这种情况下，它通常用于实现国际化和本地化字符串之间翻译查找的函数名称，这似乎源自并遵循相应的C约定。例如，在[Django文档“转换”章节](https://docs.djangoproject.com/en/dev/topics/i18n/translation/)中，你将能看到如下代码：

[?](http://my.oschina.net/leejun2005/blog/387136)

|  |  |
| --- | --- |
| 1  2  3  4  5 | from django.utils.translation import ugettext as \_  from django.http import HttpResponse  def my\_view(request):      output = \_("Welcome to my site.")      return HttpResponse(output) |

可以发现，场景二和场景三中的使用方法可能会相互冲突，所以我们需要避免在使用“\_”作为国际化查找转换功能的代码块中同时使用“\_”作为临时名称。

# 2、名称前的单下划线（如：\_shahriar）

程序员使用名称前的单下划线，用于指定该名称属性为“私有”。这有点类似于惯例，为了使其他人（或你自己）使用这些代码时将会知道以“\_”开头的名称只供内部使用。正如Python文档中所述：   
以下划线“\_”为前缀的名称（如\_spam）应该被视为API中非公开的部分（不管是函数、方法还是数据成员）。此时，应该将它们看作是一种实现细节，在修改它们时无需对外部通知。   
正如上面所说，这确实类似一种惯例，因为它对解释器来说确实有一定的意义，如果你写了代码“from <模块/包名> import \*”，那么以“\_”开头的名称都不会被导入，除非模块或包中的“\_\_all\_\_”列表显式地包含了它们。了解更多请查看“ [Importing \* in Python](http://shahriar.svbtle.com/importing-star-in-python) ”。   
不过值得注意的是，如果使用 import a\_module 这样导入模块，仍然可以用 a\_module.\_some\_var 这样的形式访问到这样的对象。

另外单下划线开头还有一种一般不会用到的情况在于使用一个 C 编写的扩展库有时会用下划线开头命名，然后使用一个去掉下划线的 Python 模块进行包装。如 struct 这个模块实际上是 C 模块 \_struct 的一个 Python 包装。

# 3、名称前的双下划线（如：\_\_shahriar）

名称（具体为一个方法名）前双下划线（\_\_）的用法并不是一种惯例，对解释器来说它有特定的意义。Python中的这种用法是为了避免与子类定义的名称冲突。Python文档指出，“\_\_spam”这种形式（至少两个前导下划线，最多一个后续下划线）的任何标识符将会被“\_classname\_\_spam”这种形式原文取代，在这里“classname”是去掉前导下划线的当前类名。例如下面的例子：

[?](http://my.oschina.net/leejun2005/blog/387136)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | >>> class A(object):  ...     def \_internal\_use(self):  ...         pass  ...     def \_\_method\_name(self):  ...         pass  ...  >>> dir(A())  ['\_A\_\_method\_name', ..., '\_internal\_use'] |

正如所预料的，“\_internal\_use”并未改变，而“\_\_method\_name”却被变成了“\_ClassName\_\_method\_name”：\_\_开头 的 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 名字改编 （Private name mangling） 。 此时，如果你创建A的一个子类B，那么你将不能轻易地覆写A中的方法“\_\_method\_name”，

[?](http://my.oschina.net/leejun2005/blog/387136)

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | >>> class B(A):  ...     def \_\_method\_name(self):  ...         pass  ...  >>> dir(B())  ['\_A\_\_method\_name', '\_B\_\_method\_name', ..., '\_internal\_use'] |

然而如果你知道了这个规律，最终你还是可以访问这个“私有”变量的。  
私有变量名字改编意在给出一个在类中定义"私有"实例变量和方法的简单途径，避免派生类的实例变量定义产生问题，或者与外界代码中的变量搞混。  
要注意的是混淆规则（私有变量名字改编）主要目的在于避免意外错误，被认作为私有的变量仍然有可能被访问或修改(使用\_classname\_\_membername)，在特定的场合它也是有用的，比如调试的时候。

上述的功能几乎和Java中的final方法和C++类中标准方法（非虚方法）一样。

再讲两点题外话：  
一是因为轧压（改编）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（改编）。

无论是单下划线还是双下划线开头的成员，都是希望外部程序开发者不要直接使用这些成员变量和这些成员函数，只是双下划线从语法上能够更直接的避免错误的使用，但是如果按照 \_类名\_\_成员名 则依然可以访问到。单下划线的在动态调试时可能会方便一些，只要项目组的人都遵守下划线开头的成员不直接使用，那使用单下划线或许会更好。

# 4、名称前后的双下划线（如：\_\_init\_\_）

这种用法表示Python中特殊的方法名。其实，这只是一种惯例，对Python系统来说，这将确保不会与用户自定义的名称冲突。通常，你将会覆写这些方法，并在里面实现你所需要的功能，以便Python调用它们。例如，当定义一个类时，你经常会覆写“\_\_init\_\_”方法。

双下划线开头双下划线结尾的是一些 Python 的“魔术”对象，如类成员的 \_\_init\_\_、\_\_del\_\_、\_\_add\_\_、\_\_getitem\_\_ 等，以及全局的 \_\_file\_\_、\_\_name\_\_ 等。 Python 官方推荐永远不要将这样的命名方式应用于自己的变量或函数，而是按照文档说明来使用。虽然你也可以编写自己的特殊方法名，但不要这样做。

[?](http://my.oschina.net/leejun2005/blog/387136)

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | >>> class C(object):  ...     def \_\_mine\_\_(self):  ...         pass  ...  >>> dir(C)  ... [..., '\_\_mine\_\_', ...] |

其实，很容易摆脱这种类型的命名，而只让Python内部定义的特殊名称遵循这种约定 :)

# 5、题外话 if \_\_name\_\_ == "\_\_main\_\_":

所有的 Python 模块都是对象并且有几个有用的属性，你可以使用这些属性方便地测试你所书写的模块。

模块是对象, 并且所有的模块都有一个内置属性 \_\_name\_\_。一个模块的 \_\_name\_\_ 的值要看您如何应用模块。如果 import 模块, 那么 \_\_name\_\_的值通常为模块的文件名, 不带路径或者文件扩展名。但是您也可以像一个标准的程序一样直接运行模块, 在这种情况下 \_\_name\_\_的值将是一个特别的缺省值：\_\_main\_\_。

[?](http://my.oschina.net/leejun2005/blog/387136)

|  |  |
| --- | --- |
| 1  2  3 | >>> import odbchelper  >>> odbchelper.\_\_name\_\_  'odbchelper' |

一旦了解到这一点, 您可以在模块内部为您的模块设计一个测试套件, 在其中加入这个 if 语句。当您直接运行模块, \_\_name\_\_ 的值是 \_\_main\_\_, 所以测试套件执行。当您导入模块, \_\_name\_\_的值就是别的东西了, 所以测试套件被忽略。这样使得在将新的模块集成到一个大程序之前开发和调试容易多了。  
在 MacPython 上, 需要一个额外的步聚来使得 if \_\_name\_\_ 技巧有效。 点击窗口右上角的黑色三角, 弹出模块的属性菜单, 确认 Run as \_\_main\_\_ 被选中。

# 6、总结：

Python 用下划线作为变量前缀和后缀指定特殊变量。  
\_xxx       不能用'from module import \*'导入  
\_\_xxx\_\_  系统定义名字  
\_\_xxx     类中的私有变量名  
核心风格：避免用下划线作为变量名的开头。  
因为下划线对解释器有特殊的意义，而且是内建标识符所使用的符号，我们建议程序员避免用下划线作为变量名的开头。一般来讲，变量名\_xxx被看作是“私有的”，在模块或类外不可以使用。当变量是私有的时候，用\_xxx 来表示变量是很好的习惯。因为变量名\_\_xxx\_\_对Python 来说有特殊含义，对于普通的变量应当避免这种命名风格。  
"单下划线" 开始的成员变量叫做保护变量，意思是只有类对象和子类对象自己能访问到这些变量；  
"双下划线" 开始的是私有成员，意思是只有类对象自己能访问，连子类对象也不能访问到这个数据。  
以单下划线开头（如\_foo）的代表不能直接访问的类属性，需通过类提供的接口进行访问，不能用“from xxx import \*”而导入；以双下划线开头的（如\_\_foo）代表类的私有成员；以双下划线开头和结尾的（\_\_foo\_\_）代表python里特殊方法专用的标识，如 \_\_init\_\_() 代表类的构造函数。

附 PEP 规范：

[?](http://my.oschina.net/leejun2005/blog/387136)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | PEP-0008:    In addition, the following special forms using leading or trailing underscores are recognized (these can generally be combined with any case convention):        - \_single\_leading\_underscore: weak "internal use" indicator. E.g. "from M import \*" does not import objects whose name starts with an underscore.        - single\_trailing\_underscore\_: used by convention to avoid conflicts with Python keyword, e.g.          Tkinter.Toplevel(master, class\_='ClassName')        - \_\_double\_leading\_underscore: when naming a class attribute, invokes name mangling (inside class FooBar, \_\_boo becomes \_FooBar\_\_boo; see below).        - \_\_double\_leading\_and\_trailing\_underscore\_\_: "magic" objects or attributes that live in user-controlled namespaces. E.g. \_\_init\_\_,        \_\_import\_\_ or \_\_file\_\_. Never invent such names; only use them as documented. |

# 7、Refer：

[1] Importing `\*` in Python

<http://shahriar.svbtle.com/importing-star-in-python>

[2] 理解Python的双下划线命名

<http://blog.csdn.net/zhu_liangwei/article/details/7667745>

[3] Python 的类的下划线命名有什么不同？

<http://www.zhihu.com/question/19754941>