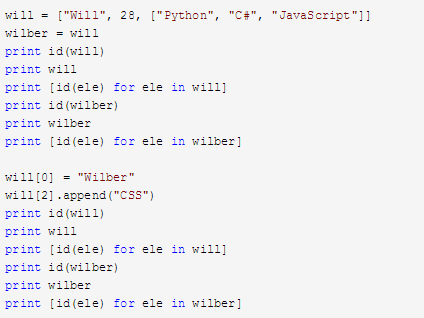
Python中，对象的赋值，拷贝（深/浅拷贝）之间是有差异的，如果使用的时候不注意，就可能产生意外的结果。

下面本文就通过简单的例子介绍一下这些概念之间的差别。

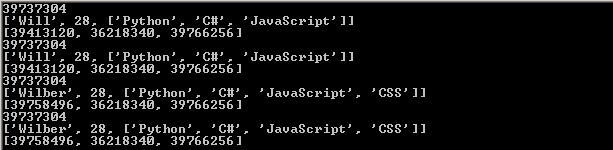
**对象赋值**

https://www.cnblogs.com/Eva-J/p/5534037.html

直接看一段代码：



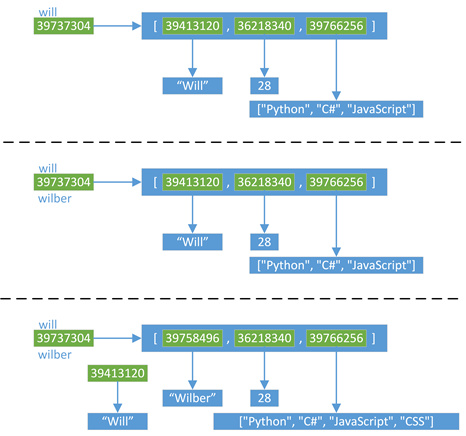
代码的输出为：



下面来分析一下这段代码：

* 首先，创建了一个名为will的变量，这个变量指向一个list对象，从第一张图中可以看到所有对象的地址（每次运行，结果可能不同）
* 然后，通过will变量对wilber变量进行赋值，那么wilber变量将指向will变量对应的对象（内存地址），也就是说”wilber is will”，”wilber[i] is will[i]”
* 第三张图中，由于will和wilber指向同一个对象，所以对will的中的元素的任何修改都会体现在wilber上（**因为修改里面的元素，这个对象的地址不会改变），但若是对这个对象重新赋值了，就会为它重新分配个内存空间，变量的内存地址也会改变，两个变量之间就没关系了**

这里需要注意的一点是，str是不可变类型，所以当修改的时候会替换旧的对象，产生一个新的地址39758496



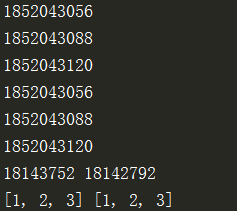
**注意：**

以上情况是赋值后，will[0]=xxx这样去改变元素的值，will和wilber指向的地址没变，所以两个会一起改变。

但是若wilber = [8,9,9]这样重新赋给一个新值，则wilber会指向一个新地址，彻底跟will分开了

这跟php中的$b=$a类似，并不是$b=&$a那样！！！

|  |
| --- |
| list = [1,2,3] list2 = list  list=[1,2,3] *for* ele *in* list:  print(id(ele)) *for* ele *in* list2:  print(id(ele)) print(id(list),id(list2)) print(list,list2) |



从上面的例子可以看出list重新赋值[1,2,3],他跟list2已经不指向同一个内存地址了，这也等于浅拷贝。只是他里面的元素还是指向同一个地址。

若list里有list元素，该list元素的地址也会变

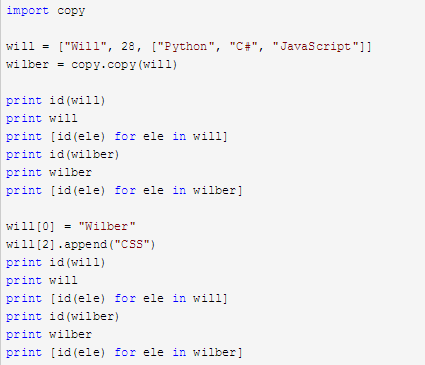
|  |
| --- |
| list = [1,2,[3,6,9]] list2 = list  list=[1,2,[3,6,9]] *for* ele *in* list:  print(id(ele)) *for* ele *in* list2:  print(id(ele)) print(id(list),id(list2)) print(list,list2) |

综上所述：

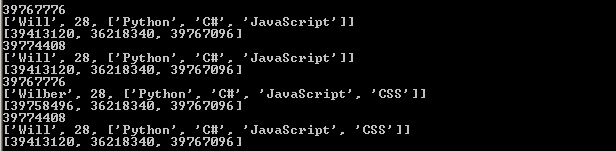
当对list=[]重新赋值的时候，其实list跟list2已经完全脱离关系了。

**浅拷贝**

下面就来看看浅拷贝的结果：



代码结果为：



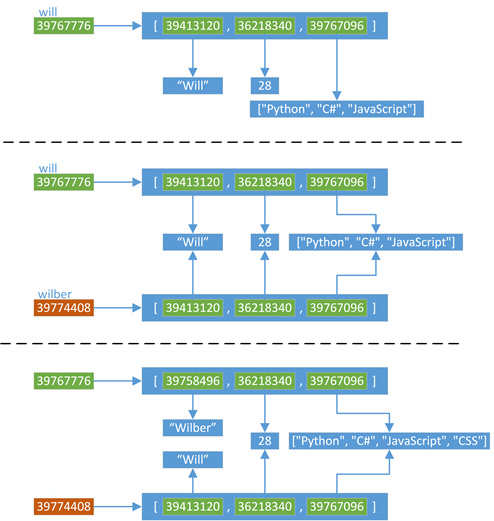
分析一下这段代码：

* 首先，依然使用一个will变量，指向一个list类型的对象
* 然后，通过copy模块里面的浅拷贝函数copy()，对will指向的对象进行浅拷贝，然后浅拷贝生成的新对象赋值给wilber变量

浅拷贝会创建一个新的对象，这个例子中”wilber is not will”  
但是，对于对象中的元素，浅拷贝就只会使用原始元素的引用（内存地址），也就是说”wilber[i] is will[i]”

* 当对will进行修改的时候

由于list的第一个元素是不可变类型，所以will对应的list的第一个元素会使用一个新的对象39758496  
但是list的第三个元素是一个可不类型，修改操作不会产生新的对象，所以will的修改结果会相应的反应到wilber上

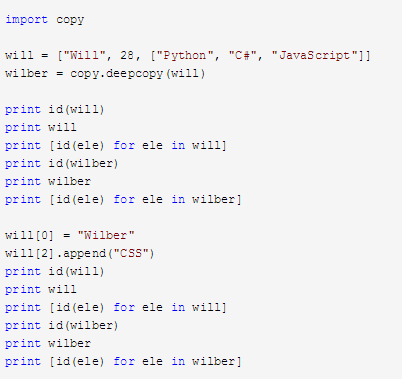


总结一下，当我们使用下面的操作的时候，会产生浅拷贝的效果：

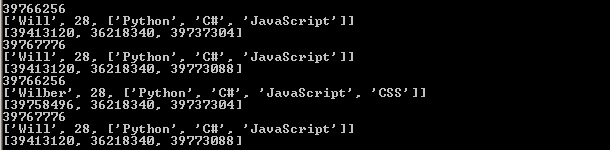
* **使用切片[:]操作**
* **使用工厂函数（如list/dir/set）**
* **使用copy模块中的copy()函数**

**深拷贝**

最后来看看深拷贝：



代码的结果为：



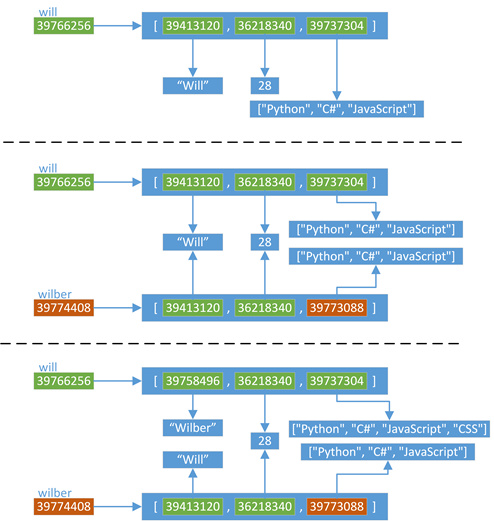
分析一下这段代码：

* 首先，同样使用一个will变量，指向一个list类型的对象
* 然后，通过copy模块里面的深拷贝函数deepcopy()，对will指向的对象进行深拷贝，然后深拷贝生成的新对象赋值给wilber变量

跟浅拷贝类似，深拷贝也会创建一个新的对象，这个例子中”wilber is not will”  
但是，对于对象中的元素，深拷贝都会重新生成一份（有特殊情况，下面会说明），而不是简单的使用原始元素的引用（内存地址）  
例子中will的第三个元素指向39737304，而wilber的第三个元素是一个全新的对象39773088，也就是说，”wilber[2] is not will[2]”

* 当对will进行修改的时候

由于list的第一个元素是不可变类型，所以will对应的list的第一个元素会使用一个新的对象39758496  
但是list的第三个元素是一个可变类型，修改操作不会产生新的对象，但是由于”wilber[2] is not will[2]”，所以will的修改不会影响wilber



**拷贝的特殊情况**

其实，对于拷贝有一些特殊情况：

* 对于非容器类型（如数字、字符串、和其他’原子’类型的对象）没有拷贝这一说

也就是说，对于这些类型，”obj is copy.copy(obj)” 、”obj is copy.deepcopy(obj)”

* 如果元祖变量只包含原子类型对象，则不能深拷贝，看下面的例子



**总结**

本文介绍了对象的赋值和拷贝，以及它们之间的差异：

* Python中对象的赋值都是进行对象引用（内存地址）传递
* 使用copy.copy()，可以进行对象的浅拷贝，它复制了对象，但对于对象中的元素，依然使用原始的引用.
* 如果需要复制一个容器对象，以及它里面的所有元素（包含元素的子元素），可以使用copy.deepcopy()进行深拷贝
* 对于非容器类型（如数字、字符串、和其他’原子’类型的对象）没有被拷贝一说
* 如果元祖变量只包含原子类型对象，则不能深拷贝，看下面的例子