Liubit
Forget Jesus. The stars died so that you could be here today.

< 2020年2月 >

日	一	二	三	四	五	六
26	27	28	29	30	31	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	27	28	29
1	2	3	4	5	6	7

随笔分类

ADSL/VDSL
PON(1)
python(10)
Tcl/Tk
计算机网络(6)
其他(1)
语音(1)
自动化(2)

【转载】[python基础] python中变量存储的方式

为了解决刚刚python 2 循环运算中，变量存储的地址与期望值的地址不同的问题，稍微检索了下python中变量的存储方式

（虽然并没有解决问题，但应该可以猜测是python 3更新期间，修改（或者说bugfix?）了变量存储的方式，所以在python 3中便没有遇到这个问题）

以下为转载记录，来源见结尾

在高级语言中，变量是对内存及其地址的抽象。

对于python而言，**python的一切变量都是对象**，变量的存储，采用了引用语义的方式，存储的只是一个变量的值所在的内存地址，而不是这个变量的只本身。

引用语义：在python中，变量保存的是对象(值)的引用，我们称为引用语义。采用这种方式，变量所需的存储空间大小一致，因为变量只是保存了一个引用。也被称为对象语义和指针语义。

值语义：有些语言采用的不是这种方式，它们把变量的值直接保存在变量的存储区里，这种方式被我们称为值语义，例如C语言，采用这种存储方式，每一个变量在内存中所占的空间就要根据变量实际的大小而定，无法固定下来。

由于python中的变量都是采用的引用语义，数据结构可以包含基础数据类型，导致了在python中每个变量中都存储了这个变量的地址，而不是值本身；

对于复杂的数据结构来说，里面的存储的也只只是每个元素的地址而已，下面给出基础类型和数据结构类型变量重新赋值的存储变化：

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

1.数据类型重新初始化对python语义引用的影响

变量的每一次初始化，都开辟了一个新的空间，将新内容的地址赋值给变量。对于下图来说，我们重复的给str1赋值，其实在内存中的变化如下右图：

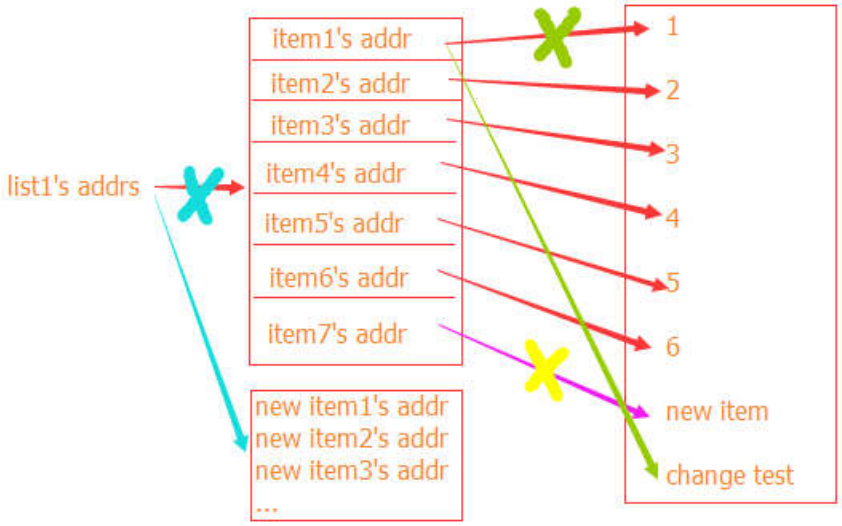
```
>>> str1 = 'hello world'
>>> print id(str1)
48672272
>>> str1 = 'new hello world'
>>> print id(str1)
48672224
```

从上图我们可以看出，str1在重复的初始化过程中，是因为str1中存储的元素地址由'hello world'的地址变成了'new hello world'的。

2.数据结构内部元素变化重对python语义引用的影响

对于复杂的数据类型来说，改变其内部的值对于变量的影响：

```
>>> lst1 = [1,2,3,4,5,6]
>>> print id(lst1)
48574728
>>> lst1.append('new item')
>>> print lst1
[1, 2, 3, 4, 5, 6, 'new item']
>>> print id(lst1)
48574728
>>> lst1.pop()
'new item'
>>> print lst1
[1, 2, 3, 4, 5, 6]
>>> print id(lst1)
48574728
>>> lst1[0] = 'change test'
>>> print lst1
['change test', 2, 3, 4, 5, 6]
>>> print id(lst1)
48574728
>>> lst1 = [1,2,3,4,5]
>>> print id(lst1)
48221192
>>>
```

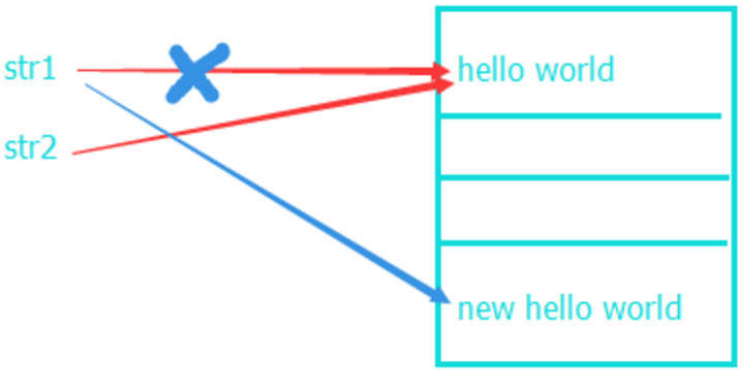


当对列表中的元素进行一些增删改的操作的时候，是不会影响到lst1列表本身对于整个列表地址的，只会改变其内部元素的地址引用。可是当我们对于一个列表重新初始化(赋值)的时候，就给lst1这个变量重新赋予了一个地址，覆盖了原本列表的地址，这个时候，lst1列表的内存id就发生了改变。上面这个道理用在所有复杂的数据类型中都是一样的。

变量赋值

1.str的赋值

```
>>> str1 = 'hello world'
>>> print id(str1)
41863664
>>> str2 = str1
>>> print id(str2)
41863664
>>> str1 = 'new hello world'
>>> print str1
new hello world
>>> print str2
hello world
>>> print id(str1)
45133920
>>> print id(str2)
41863664
```



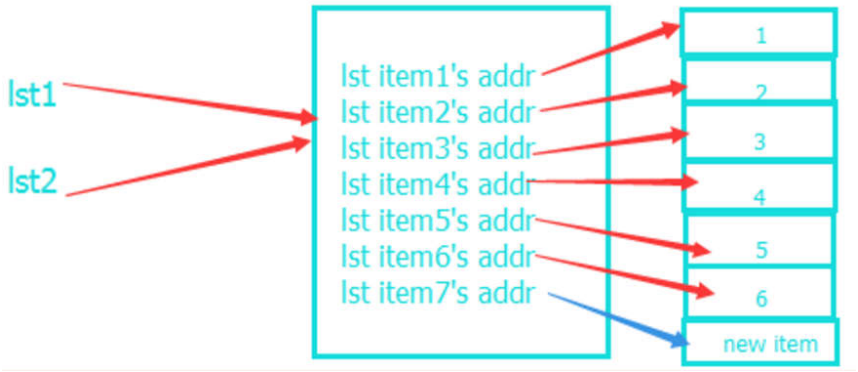
我们刚刚已经知道，str1的再次初始化（赋值）会导致内存地址的改变，从上图的结果我们可以看出修改了str1之后，被赋值的str2从内存地址到值都没有受到影响。

看内存中的变化，起始的赋值操作让str1和str2变量都存储了'hello world'所在的地址，重新对str1初始化，使str1中存储的地址发生了改变，指向了新建的值，此时str2变量存储的内存地址并未改变，所以不受影响。

2.复杂的数据结构中的赋值

刚刚我们看了简单数据类型的赋值，现在来看复杂数据结构变化对应内存的影响。

```
>>> lst1 = [1,2,3,4,5,6]
>>> lst2 = lst1
>>> print id(lst1)
48226504
>>> print id(lst2)
48226504
>>> lst1.append('new item')
>>> print lst1
[1, 2, 3, 4, 5, 6, 'new item']
>>> print lst2
[1, 2, 3, 4, 5, 6, 'new item']
>>> print id(lst1)
48226504
>>> print id(lst2)
48226504
```



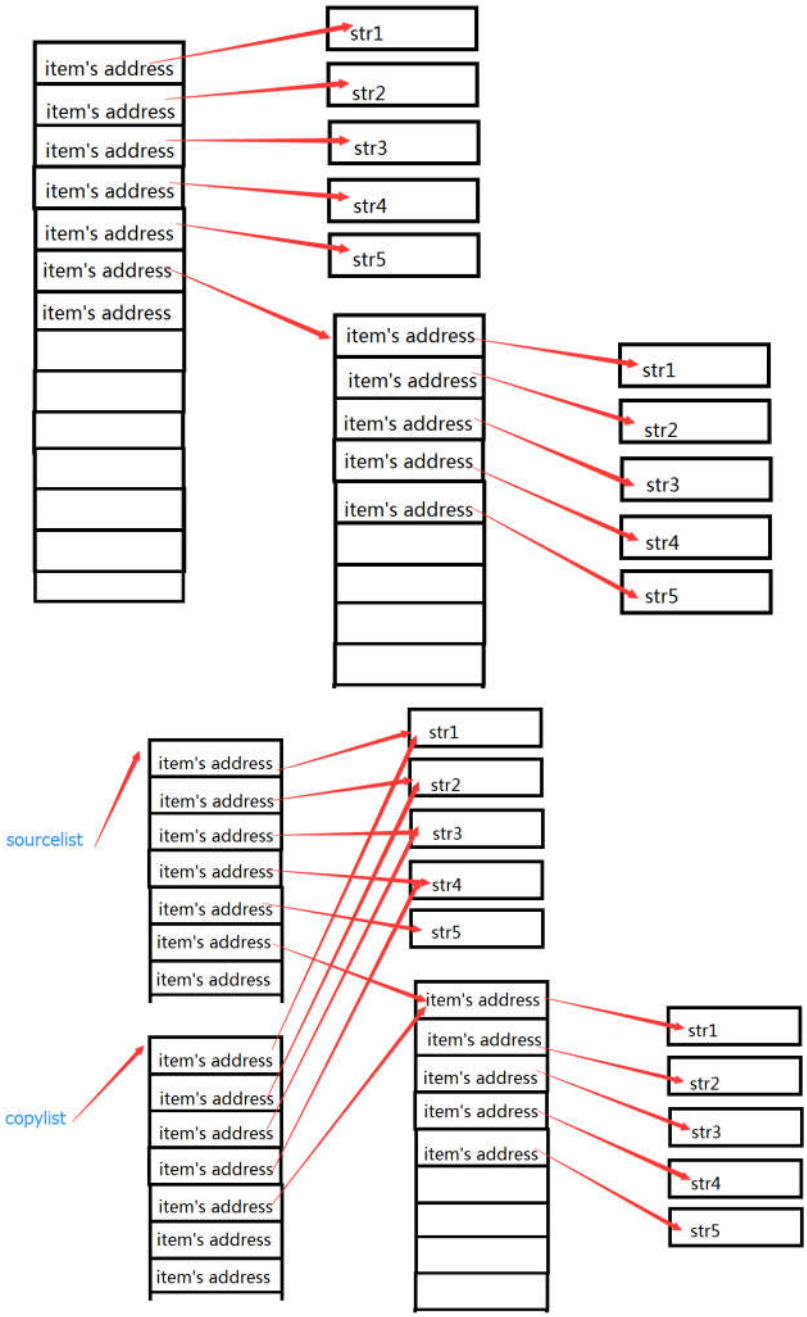
上图对列表的增加修改操作，没有改变列表的内存地址，lst1和lst2都发生了变化。

对照内存图我们不难看出，在列表中添加新值时，列表中又多存储了一个新元素的地址，而列表本身的地址没有变化，所以lst1和lst2的id均没有改变并且都被添加了一个新的元素。

简单的比喻一下，我们出去吃饭，lst1和lst2就像是同桌吃饭的两个人，两个人公用一张桌子，只要桌子不变，桌子上的菜发生了变化两个人是共同感受的。

浅拷贝

首先，我们来了解一下浅拷贝。浅拷贝：不管多么复杂的数据结构，浅拷贝都只会copy一层。下面就让我们看一张图，来了解一下浅浅拷贝的概念。



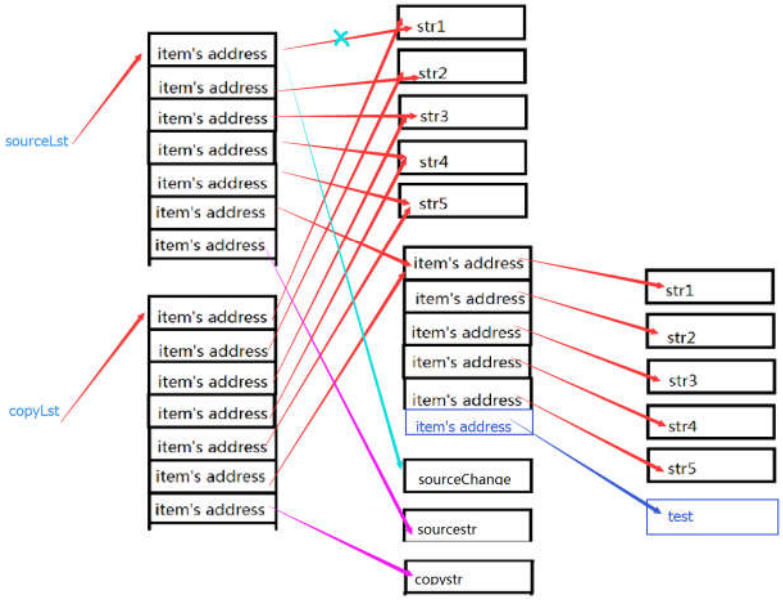
看上面两张图，我们加入左图表示的是一个列表sourcelist，sourcelist = ['str1','str2','str3','str4','str5',['str1','str2','str3','str4','str5']];

右图在原有的基础上多出了一个浅拷贝的copylist，copylist = ['str1','str2','str3','str4','str5',['str1','str2','str3','str4','str5']];

sourcelist和copylist表面上看起来一模一样，但是实际上在内存中已经生成了一个新列表，copy了sourceLst，获得了一个新列表，存储了5个字符串和一个列表所在内存的地址。

我们看下面分别对两个列表进行的操作，红色的框框里面是变量初始化，初始化了上面的两个列表；我们可以分别对这两个列表进行操作，例如插入一个值，我们会发现什么呢？如下所示：

从上面的代码我们可以看出，对于sourceLst和copyLst列表添加一个元素，这两个列表好像是独立的一样都分别发生了变化，但是当我修改lst的时候，这两个列表都发生了变化，这是为什么呢？我们就来看一张内存中的变化图：



我们可以知道sourceLst和copyLst列表中都存储了一坨地址，当我们修改了sourceLst1的元素时，相当于用'sourceChange'的地址替换了原来'str1'的地址，所以sourceLst的第一个元素发生了变化。而copyLst还是存储了str1的地址，所以copyLst不会发生改变。

当sourceLst列表发生变化，copyLst中存储的lst内存地址没有改变，所以当lst发生改变的时候，sourceLst和copyLst两个列表就都发生了改变。

这种情况发生在字典套字典、列表套字典、字典套列表，列表套列表，以及各种复杂数据结构的嵌套中，所以当我们的数据类型很复杂的时候，用copy去进行浅拷贝就要非常小心。。。

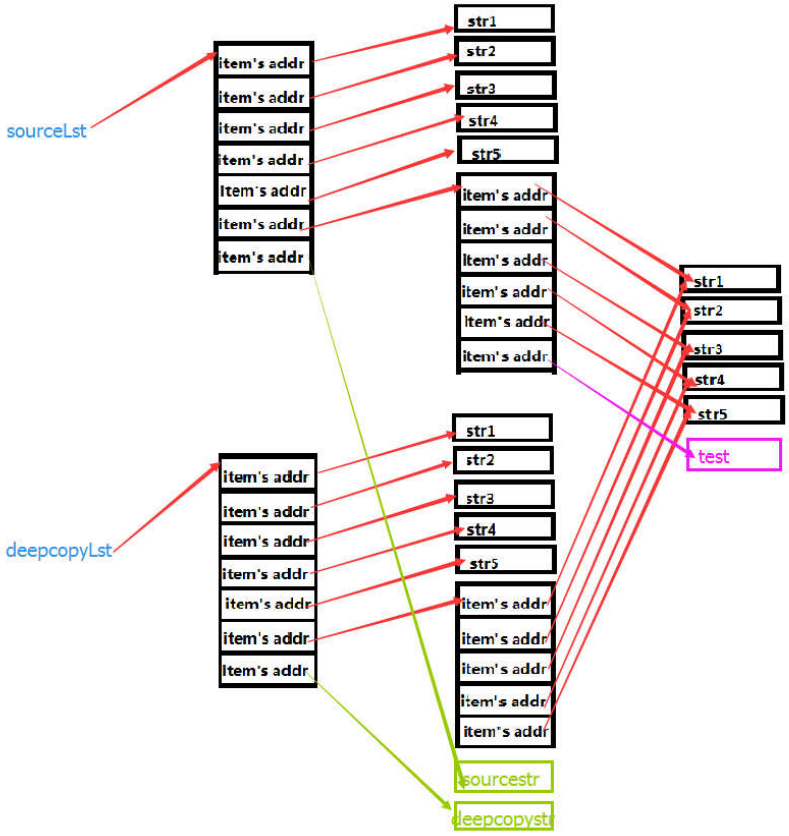
深拷贝

深拷贝——即python的copy模块提供的另一个deepcopy方法。深拷贝会完全复制原变量相关的所有数据，在内存中生成一套完全一样的内容，在这个过程中我们对这两个变量中的一个进行任意修改都不会影响其他变量。下面我们就来试验一下。

```
import copy
lst = ['str1','str2','str3','str4','str5']
sourceLst = ['str1','str2','str3','str4','str5',lst]
deepcopyLst = copy.deepcopy(sourceLst)
print '->',sourceLst
print '->',deepcopyLst,'\n'

sourceLst.append('sourcestr')
deepcopyLst.append('deepcopystr')
print '-->',sourceLst
print '-->',deepcopyLst,'\n'

lst.append('test')
print '--->',sourceLst
print '--->',deepcopyLst,'\n'
```

```
Run 8
C:\Python27\python.exe D:/Mywork/working/testPackage/day5homework/8.py
-> ['str1', 'str2', 'str3', 'str4', 'str5', ['str1', 'str2', 'str3', 'str4', 'str5']]
-> ['str1', 'str2', 'str3', 'str4', 'str5', ['str1', 'str2', 'str3', 'str4', 'str5']]

--> ['str1', 'str2', 'str3', 'str4', 'str5', ['str1', 'str2', 'str3', 'str4', 'str5'], 'sourcestr']
--> ['str1', 'str2', 'str3', 'str4', 'str5', ['str1', 'str2', 'str3', 'str4', 'str5'], 'deepcopystr']

---> ['str1', 'str2', 'str3', 'str4', 'str5', ['str1', 'str2', 'str3', 'str4', 'str5', 'test'], 'sourcestr']
---> ['str1', 'str2', 'str3', 'str4', 'str5', ['str1', 'str2', 'str3', 'str4', 'str5'], 'deepcopystr']
```

看上面的执行结果，这一次我们不管是对直接对列表进行操作还是对列表内嵌套的其他数据结构操作，都不会产生拷贝的列表受影响的情况。我们再来看看这些变量在内存中的状况：



看了上面的内容，我们就知道了深拷贝的原理。其实深拷贝就是在内存中重新开辟一块空间，不管数据结构多么复杂，只要遇到可能发生改变的数据类型，就重新开辟一块内存空间把内容复制下来，直到最后一层，

不再有复杂的数据类型，就保持其原引用。这样，不管数据结构多么的复杂，数据之间的修改都不会相互影响。这就是深拷贝~~~

此文链接:
http://makaidong.com/maikerniuniu/1280_9073599.html

转载请注明出处：python变量存储

分类: python

标签: python

Liubit
关注 - 0
粉丝 - 1
+加关注

« 上一篇: [python基础] python 2与python 3的区别，一个关于对象的未知的坑
» 下一篇: [python基础] 同时赋值多个变量与变量值交换

posted on 2017-10-14 20:36 Liubit 阅读(5649) 评论(0) 编辑 收藏

相关博文：

- python变量存储
- Python数据结构之----数据存储与深浅拷贝
- python——赋值与深浅拷贝
- python3【基础】-赋值与深浅拷贝
- 详解Python变量在内存中的存储

» 更多推荐...

最新 IT 新闻：

- 股价10天9涨停！在线教育"冲击波"远不止这些
- 顺丰无人机打通医疗物资运送空中通道：7分钟可抵金银潭医院
- "超临界水分解"新方法将含碳废物变能源
- 台积电超罕见将募资20亿美元，5nm/3nm备充沛银弹战三星
- 诺基亚宣布退出MWC 2020：全球TOP4通信巨头仅剩华为参展

» 更多新闻...

历史上的今天：

2017-10-14 【转载】[python基础] python中变量存储的方式

2017-10-14 [python基础] python 2与python 3的区别，一个关于对象的...