SHALLOW AND DEEP COPY

INTRODUCTION

As we have seen in the chapter "Data Types and Variables", Python has a strange behaviour - in comparison with other programming languages - when assigning and copying simple data types like integers and strings. The difference between shallow and deep copying is only relevant for compound objects, which are objects containing other objects, like lists or class instances.

In the following code snippet y points to the same memory location than X. This changes, when we assign a different value to y. In this case y will receive a separate memory location, as we have seen in the chapter "Data Types and Variables".

```
>>> x = 3
>>> y = x
```

But even if this internal behaviour appears strange compared to programming languages like C, C++ and Perl, yet the observable results of the assignments answer our expectations. But it can be problematic, if we copy mutable objects like lists and dictionaries.

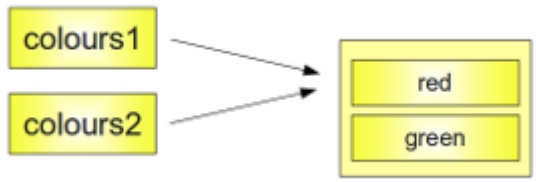
Python creates real copies only if it has to, i.e. if the user, the programmer, explicitly demands it.

We will introduce you to the most crucial problems, which can occur when copying mutable objects, i.e. when copying lists and dictionaries.

COPYING A LIST

```
>>> colours1 = ["red", "green"]
>>> colours2 = colours1
>>> colours2 = ["rouge", "vert"]
```

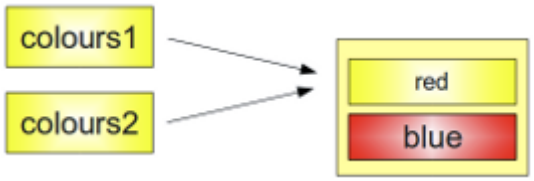
```
>>> print colours1
['red', 'green']
```



In the example above a simple list is assigned to colours1. In the next step we assign colour1 to colours2. After this, a new list is assigned to colours2.

As we have expected, the values of colours1 remained unchanged. Like it was in our example in the chapter "Data types and variables", a new memory location had been allocated for colours2, because we have assigned a complete new list to this variable.

```
>>> colours1 = ["red", "green"]
>>> colours2 = colours1
>>> colours2[1] = "blue"
>>> colours1
['red', 'blue']
```



But the question is, what will happen, if we change an element of the list of colours2 or colours1?
In the example above, we assign a new value to the second element of colours2. Lots of beginners will be astonished that the list of colours1 has been "automatically" changed as well.
The explanation is that there has been no new assignment to colours2, only to one of its elements.

COPY WITH THE SLICE OPERATOR

It's possible to completely copy shallow list structures with the slice operator without having any of the side effects, which we have described above:

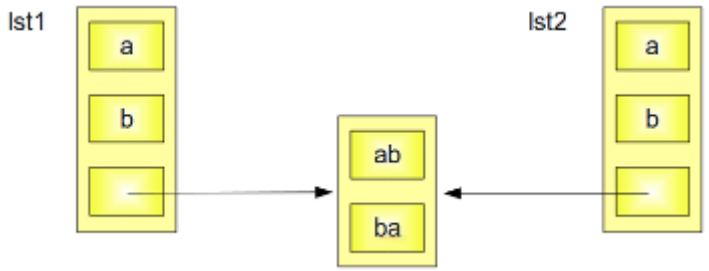
```
>>> list1 = ['a','b','c','d']
>>> list2 = list1[:]
>>> list2[1] = 'x'
>>> print list2
```

```
['a', 'x', 'c', 'd']
>>> print list1
['a', 'b', 'c', 'd']
>>>
```

But as soon as a list contains sublists, we have the same difficulty, i.e. just pointers to the sublists.

```
>>> lst1 = ['a','b',['ab','ba']]
>>> lst2 = lst1[:]
```

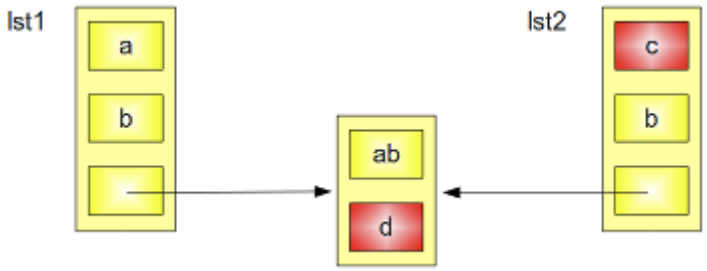
This behaviour is depicted in the following diagram:



If you assign a new value to the 0th Element of one of the two lists, there will be no side effect. Problems arise, if you change one of the elements of the sublist.

```
>>> lst1 = ['a','b',['ab','ba']]
>>> lst2 = lst1[:]
>>> lst2[0] = 'c'
>>> lst2[2][1] = 'd'
>>> print(lst1)
['a', 'b', ['ab', 'd']]
```

The following diagram depicts what happens, if one of the elements of a sublist will be changed: Both the content of lst1 and lst2 are changed.



USING THE METHOD DEEPCOPY FROM THE MODULE COPY

A solution to the described problems is to use the module "copy". This module provides the method "copy", which allows a complete copy of a arbitrary list, i.e. shallow and other lists.

The following script uses our example above and this method:

```
from copy import deepcopy

lst1 = ['a','b',['ab','ba']]

lst2 = deepcopy(lst1)

lst2[2][1] = "d"
lst2[0] = "c";

print lst2
print lst1
```

If we save this script under the name of deep_copy.py and if we call the script with "python deep_copy.py", we will receive the following output:

```
$ python deep_copy.py
['c', 'b', ['ab', 'd']]
['a', 'b', ['ab', 'ba']]
```