Theory: Copy of an object

34 minutes 0 / 5 problems solved

Often you need to have several copies of the same object. You already know that assigning one variable to another does not create a new object, instead, the new variable **refers** to the **same object**. For immutable objects, e.g. strings, it is enough: you can easily access and change variables independently of other variables. However, with mutable objects, if you assign one variable to another and then modify an object using any of these variables, it will affect both variables. This is so because with *immutable* objects, each time you try to modify them, a new object is created. *Mutable* objects, in contrary, are modified in place. Think, what if we want to copy a mutable object and alter the original without changing the copy? In this topic, we will learn to do so.

§1. Copy a mutable: shallow copy

To copy an object, we can use shallow copying, it is often used with mutable containers. As you know, mutable containers, e.g. a list, store **references** to objects that store values. With shallow copying, we can create a copy of a list that contains the same objects, but the reference to a container will be new.

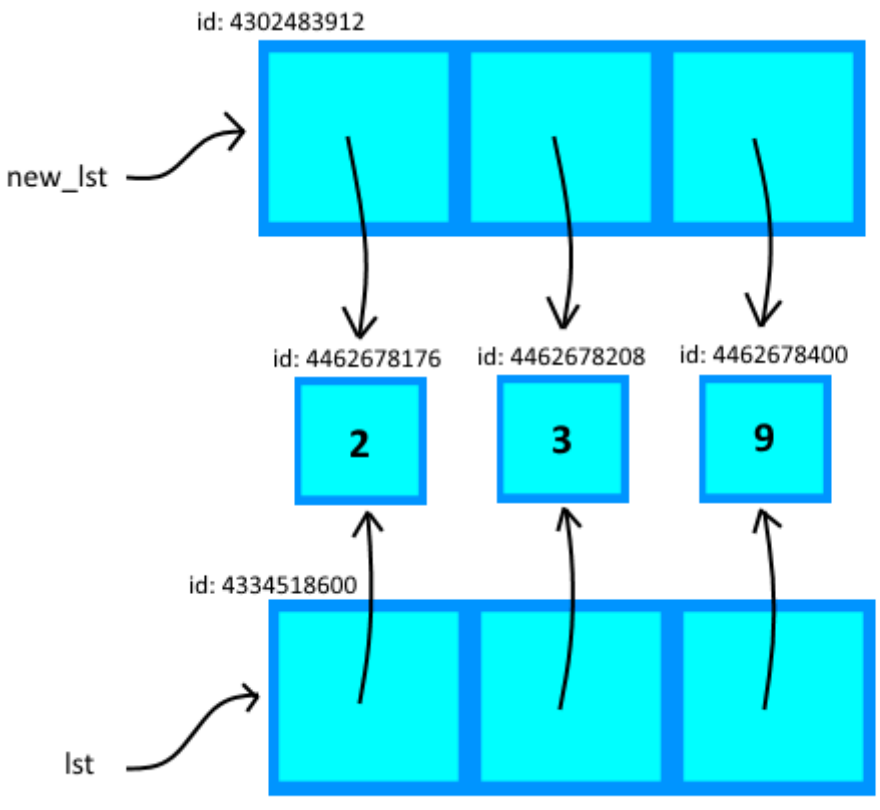
There are two ways to do this. The universal way is to use the `copy` function from the `copy` module, it works with any object.

```
import copy

lst = [2, 3, 9]
new_lst = copy.copy(lst)
```

Also, some containers such as list, set, dictionary have the `copy` *method* which can be used instead of the `copy` *function*.

```
lst = [2, 3, 9]
new_lst = lst.copy()
```

In both cases, a **new list** is created that stores references to **the same objects**.



Now, if we change one list, this will not affect the other one.

Current topic:

Copy of an object

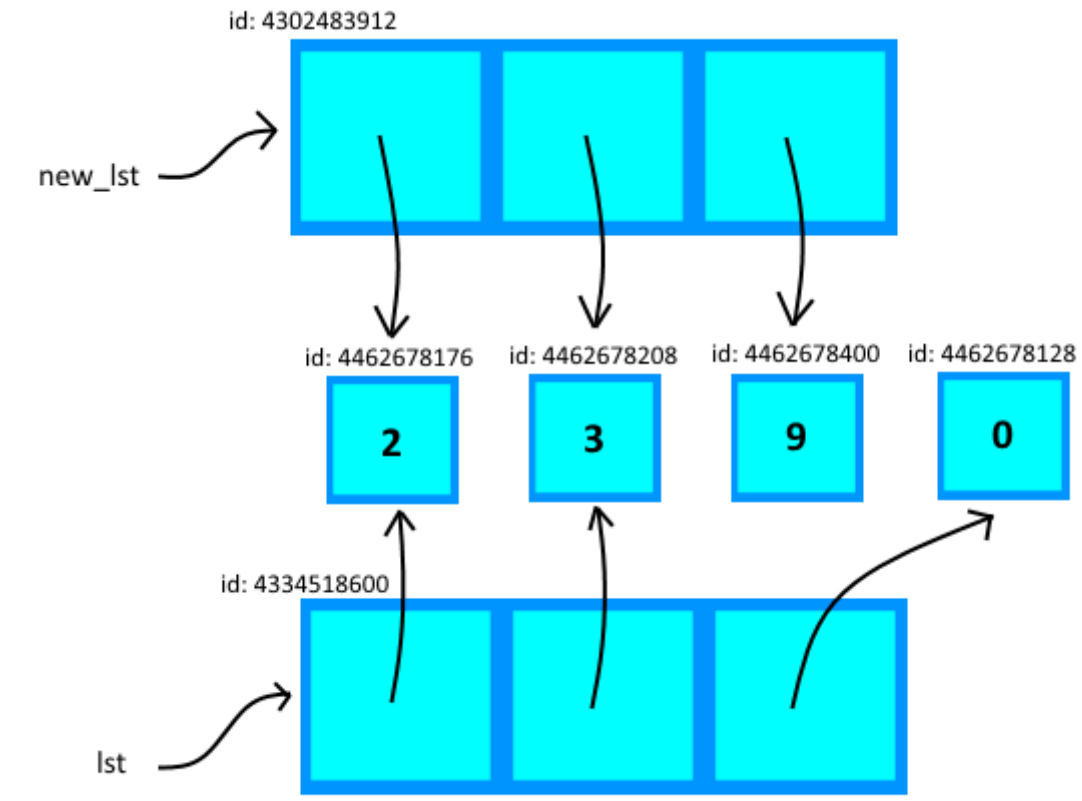
Topic depends on:

- Dictionary (Stage 1) 6
- Nested lists
- Objects in Python
- Load module (Stage 1) 5

Table of contents:

```
print(lst, id(lst))           # [2, 3, 9] 4334518600
print(new_lst, id(new_lst))   # [2, 3, 9] 4302483912

# we change an element of the first list
lst[2] = 0
print(lst, id(lst))           # [2, 3, 0] 4334518600

# the new list remains the same
print(new_lst, id(new_lst))   # [2, 3, 9] 4302483912
```

The modified list will store a reference to a new object:



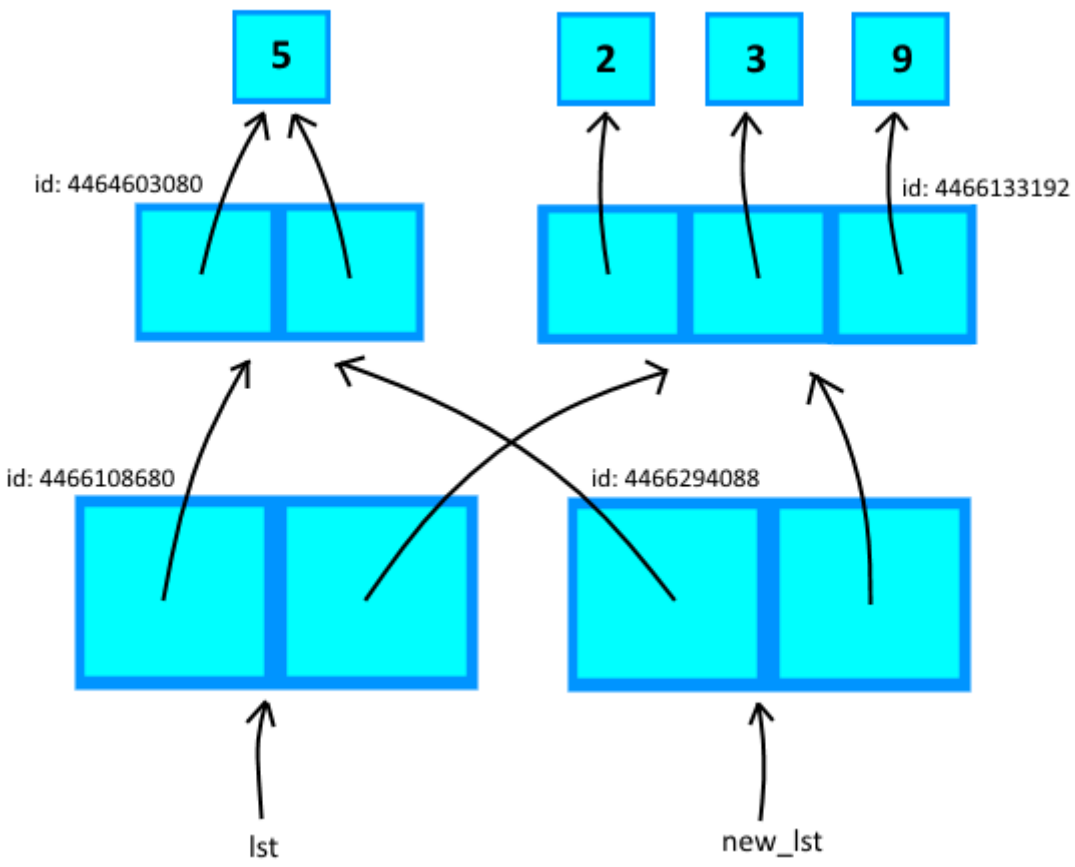
§2. Not so simple

Let's now see another example with a dictionary. When we use shallow copying, the values stored in a dictionary are not copied (look at the result of the `id` function):

```
menu = {
   'breakfast': ['porridge'],
   'lunch': ['soup', 'main course', 'compote'],
   'dinner': ['main course', 'dessert', 'tea'],
}

copy_menu = menu.copy()

# a new container is created:


print(id(menu), id(copy_menu))                    # 4353003592 4353042040


  # but the stored values are the same


print(id(menu['lunch']), id(copy_menu['lunch']))    # 4353060296 4353060296
```

The thing is, when we use shallow copying, only the container is duplicated but the elements remain the same.

However, you might want to copy the elements as well. For example, when elements of a list are lists themselves and you want to modify them without changing the copy. Here is how a shallow copy works without copying inner elements:



In this example, the copied list `new_lst` is a new object that, however, contains references to the same lists as the initial `lst`. So, when we change the list that is an element of `lst`, we also change it in `new_lst`.

> Note that there is only one object that stores number 5 because Python reuses objects containing small integers for the sake of memory usage optimization.

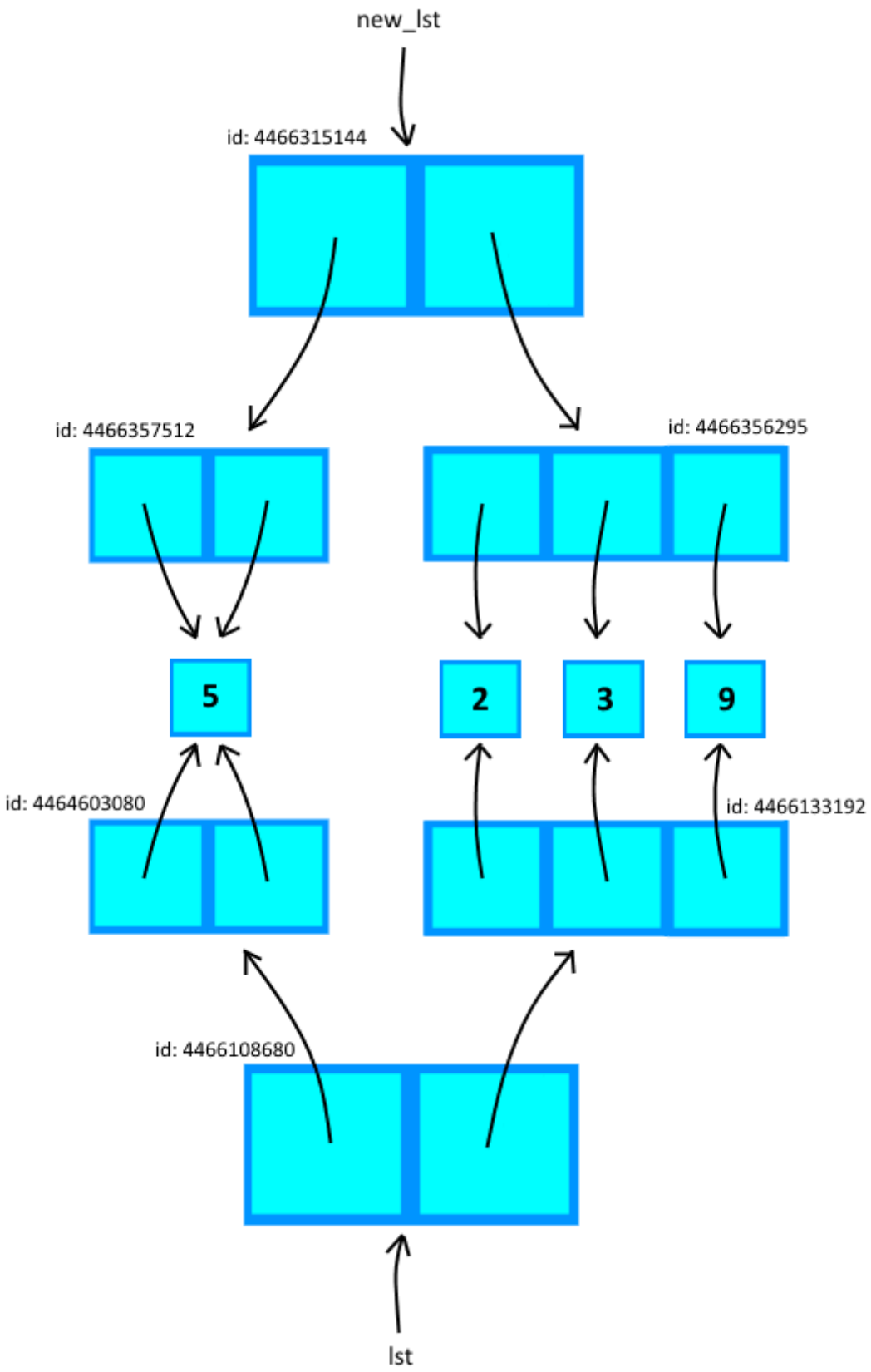
Let's see how to create a copy of a list that doesn't share its elements with the initial one.

§3. Copy a mutable: deep copy

The `deepcopy` function from the `copy` module creates a clone that doesn't relate to the initial object.

```
lst = [[5, 5], [2, 3, 9]]
new_lst = copy.deepcopy(lst)
```

If the given object is a container that stores references to other objects, this function recursively copies those objects as well, not only references to them. As a result, a new container with new elements is created.



You can notice that the `deepcopy` function doesn't copy objects that contain integers. Again, this is because Python tries to optimize the use of memory. Since integers in Python are **immutable**, this doesn't create any problems.

Now you can modify the initial object how you want and this will not affect the copy.

```
lst[0].append(5)

print(lst)                    # [[5, 5, 5], [2, 3, 9]]
print(new_lst)                # [[5, 5], [2, 3, 9]]
```

Using a deep copy may sound as a perfect solution whenever we want to copy an object: why bother about mutability if we can just use `copy.deepcopy()`? However, the `deepcopy` function has a problem: it consumes a lot of memory because it duplicates the entire structure of an object. Shallow copy uses less memory, so it's better to make sure that changing one of the copies will not affect the rest and use the shallow copying in this case.

§4. Conclusion

- The `copy` function duplicates the container but not the items in the container. It consumes less memory comparing to the `deepcopy` function, but might create problems because the copies share their elements.

- The `deepcopy` function creates a copy of the container and also recursively clones all items in the container. The resulting copies are independent but this requires a lot of memory.

Report a typo

82 users liked this theory. **2** didn't like it. **What about you?**

Start practicing

Comments (5) Hints (0) Useful links (1) Show discussion