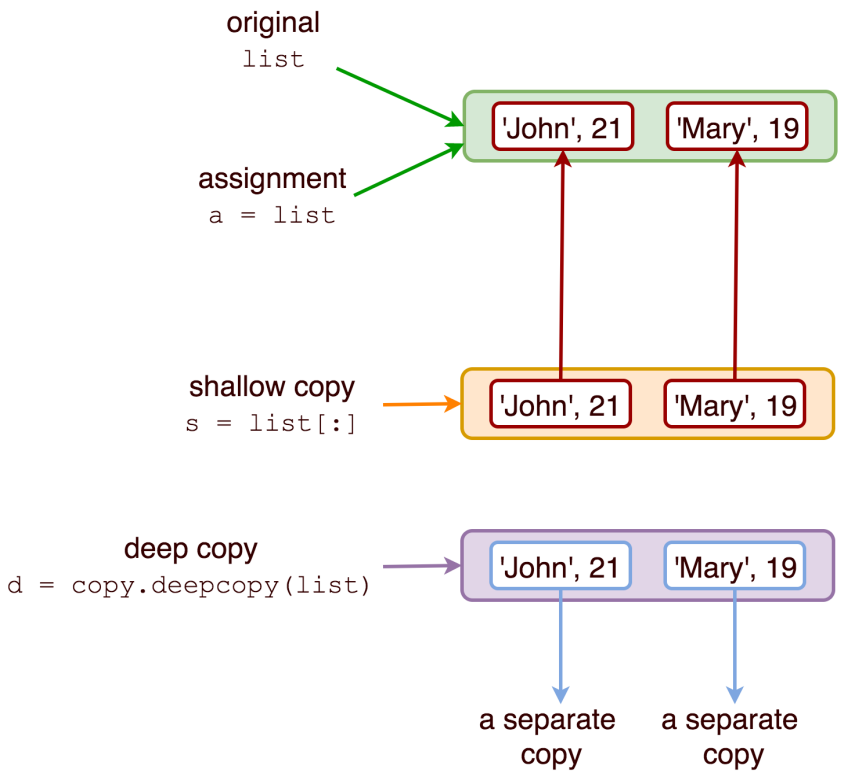
Python: Assignment vs Shallow Copy vs Deep Copy

Thawsitt Naing Follow
Dec 26, 2017 · 5 min read

Copying a list in Python might be trickier than you think. There are 3 ways you can do it: simply using the assignment operator (=), making a shallow copy and making a deep copy. In this article, I will explain what each operation does and how they are different.



Mutable elements in a shallow copy point to the original elements but those in a deep copy don't

. . .

Assignment Operator (=)

> *Assignment with an = on lists does not make a copy. Instead, assignment makes the two variables point to the one list in memory.*

```
b = colors   ## Does not copy the list
```

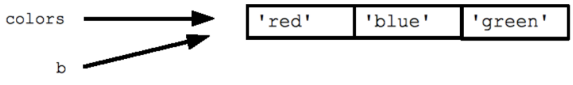


Image: Google Developers

So, if you edit the new list, changes will be reflected on the original list.

```
colors = ['red', 'blue', 'green']
b = colors


>> b.append('white')


>> b
['red', 'blue', 'green', 'white']


>> colors
['red', 'blue', 'green', 'white']
```

Easy enough, right? Let's look at other ways to copy objects.

Before we discuss shallow copy and deep copy, keep in mind that

> ***The difference between shallow and deep copying is only relevant for compound objects (e.g. a list of lists, or class instances)***.

. . .

Shallow Copy

> *A shallow copy constructs a new compound object and then (to the extent possible) inserts references into it to the objects found in the original.*

It is OK if you find it confusing now. It will make sense soon.

Let's look at the easy bits first. Here is how you make a shallow copy. In Python 3, you can use `list.copy()`. However, I prefer the equivalent expression `list[:]` because it works in both Python 2 and 3.

Shallow copy is different from assignment in that **it creates a new object**. So, if you make changes to the new list, such as adding or removing items, it won't affect the original list.

```
a = [[1, 2], [2, 4]]
b = a[:] ## shallow copy
```

```
>> b.append([3, 6])


>> b
>> [[1, 2], [2, 4], [3, 6]]

>> a
>> [[1, 2], [2, 4]]
```

You might think making a shallow copy is simple, **BUT** here is the tricky part. If the original list is a compound object (e.g. a list of lists), the elements in the new object are ***referenced*** to the original elements. (which is why it is called a shallow copy). So, if you modify the *mutable* elements like lists, the changes will be reflected on the original elements. This will become more clear with an example.

```
a = [[1, 2], [2, 4]]
b = a[:] #shallow copy


>> b[0].append(3)  ## Edit the first element (i.e. [1, 2])


>> b
>> [[1, 2, 3], [2, 4]]

>> a
>> [[1, 2, 3], [2, 4]]
```

Do you understand the difference between the two examples?

In the first example, we added a new item to `b` but this change is ***not*** reflected on the original list because `a` and `b` have separate pointers.

In the second example, we changed the first element of list `b` and this change ***is*** reflected on the original list. This is because the first element `b[0]` is referenced to `a[0]`.

List `b` has its own pointer, but its elements do not. The copied elements in `b` are pointing to the original elements in `a`. This is a characteristic of shallow copy. Now, take a loot at the definition of shallow copy again and see if it makes sense.

. . .

Deep Copy

> *A deep copy constructs a new compound object and then, recursively, inserts copies into it of the objects found in the original.*

Again, don't focus on the definition yet. The deep copy is different from shallow copy in that the copied elements have their own pointers and are not referenced to the original elements. Therefore, **no matter how you modify the deep copy, the changes will NOT be reflected on the original list.**

Creating a deep copy is slower, because you are making new copies for everything. You will need to import `copy` module to make a deep copy.

```
a = [[1, 2], [2, 4]]


>> import copy
>> b = copy.deepcopy(a) ## deep copy


>> b[0].append(3)  ## Edit the first element (i.e. [1, 2])


>> b
>> [[1, 2, 3], [2, 4]]

>> a
>> [[1, 2], [2, 4]] ## does not affect the original list
```

Hopefully, the diagram at the beginning of this article will solidify your understanding. Pay special attention to the pointers, as they are crucial to understand the differences between each operation.

. . .

Pointers

In cPython, you can use `id()` method to get the address of an object in memory.

```
>> test = ['a', 'b', 'c']
>> id(test)
4367617736
```

This will be helpful for examining the pointer locations of our newly created list objects. Let's dive right into it.

```
>> original = [1, [2, 3]] # Two elements: 1 and [2, 3]

>> a = original # assignment
>> id(a) == id(original)
True       # same pointer!

>> s = original[:] # shallow-copy
>> id(s) == id(original)
False      # s has its own pointer!
>> id(s[1]) == id(original[1])
True       # The list [2,3] in s references the original li

>> import copy
```

The best way to understand Python functions is to open a **Python shell** in your terminal and play around with it. This is how I learn most concepts in Python.

. . .

TL;DR

For simple lists such as a list of integers:

- Use assignment `=` if you want the new changes to affect the original list.
- Use *shallow* copy `[:]` if you **don't** want the new changes to affect the original list.

For compound objects (e.g. a list of lists):

- Use assignment `=` if you want the new changes to affect the original list.
- Use *deep* copy if you **don't** want the new changes to affect the original list.
- Remember: deep copy makes sure that the newly copied object is **not** referenced to the original object in any way.

I hope this helped. Feel free to suggest any improvement or fix.

. . .

References:

- Google Developers: https://developers.google.com/edu/python/lists
- Python documentation: https://docs.python.org/2/library/copy.html
- Stackoverflow: link

. . .

Thawsitt Naing is a senior studying computer science at Stanford University (class of 2019). His interests include Python, web technologies, and machine learning.