Copy in Python (Deep Copy and Shallow Copy)

In Python, Assignment statements do not copy objects, they create bindings between a target and an object. When we use = operator user thinks that this creates a new object; well, it doesn't. It only creates a new variable that shares the reference of the original object. Sometimes a user wants to work with mutable objects, in order to do that user looks for a way to create "real copies" or "clones" of these objects. Or, sometimes a user wants copies that user can modify without automatically modifying the original at the same time, in order to do that we create copies of objects.

A copy is sometimes needed so one can change one copy without changing the other. In Python, there are two ways to create copies :

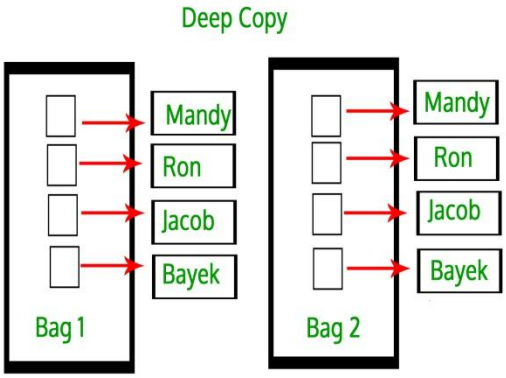
1. **Deep copy**
2. **Shallow copy**

In order to make these copy, we use copy module. We use copy module for shallow and deep copy operations. For Example

```
# importing copy module
import copy

# initializing list 1
li1 = [1, 2, [3,5], 4]


# using copy for shallow copy
li2 = copy.copy(li1)

# using deepcopy for deepcopy
li3 = copy.deepcopy(li1)
```

In the above code, the copy() returns a shallow copy of list and deepcopy() return a deep copy of list.

Deep copy



Deep copy is a process in which the copying process occurs recursively. It means first constructing a new collection object and then recursively populating it with copies of the child objects found in the original. In case of deep copy, a copy of object is copied in other object. It means that **any changes** made to a copy of object **do not reflect** in the original object. In python, this is implemented using "**deepcopy()**" function.

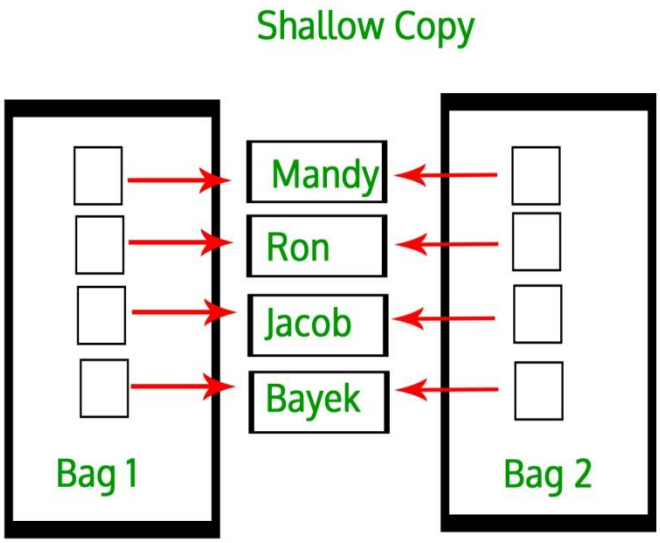
```
# Python code to demonstrate copy operations

# importing "copy" for copy operations
import copy

# initializing list 1
li1 = [1, 2, [3,5], 4]

# using deepcopy to deep copy
li2 = copy.deepcopy(li1)

# original elements of list
print ("The original elements before deep copying")
for i in range(0,len(li1)):
    print (li1[i],end=" ")

print("\r")

# adding and element to new list
li2[2][0] = 7

# Change is reflected in l2
print ("The new list of elements after deep copying ")
for i in range(0,len( li1)):
    print (li2[i],end=" ")

print("\r")

# Change is NOT reflected in original list
# as it is a deep copy
print ("The original elements after deep copying")
for i in range(0,len( li1)):
    print (li1[i],end=" ")
```

Output:

```
The original elements before deep copying

1 2 [3, 5] 4

The new list of elements after deep copying

1 2 [7, 5] 4

The original elements after deep copying

1 2 [3, 5] 4
```

In the above example, the change made in the list **did not** effect in other lists, indicating the list is deep copied.

Shallow copy



A shallow copy means constructing a new collection object and then populating it with references to the child objects found in the original. The copying process does not recurse and therefore won't create copies of the child objects themselves. In case of shallow copy, a reference of object is copied in other object. It means that **any changes** made to a copy of object **do reflect** in the original object. In python, this is implemented using "**copy()**" function.

 # Python code to demonstrate copy operations

```
# importing "copy" for copy operations
import copy

# initializing list 1
li1 = [1, 2, [3,5], 4]

# using copy to shallow copy
li2 = copy.copy(li1)

# original elements of list
print ("The original elements before shallow copying")
for i in range(0,len(li1)):
    print (li1[i],end=" ")

print("\r")

# adding and element to new list
li2[2][0] = 7

# checking if change is reflected
print ("The original elements after shallow copying")
```

```
for i in range(0,len( li1)):
    print (li1[i],end=" ")
```

Output:

```
The original elements before shallow copying
1 2 [3, 5] 4
The original elements after shallow copying
1 2 [7, 5] 4
```

In the above example, the change made in the list **did** effect in other list, indicating the list is shallow copied.

Important Points:

The difference between shallow and deep copying is only relevant for compound objects (objects that contain other objects, like lists or class instances):

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

Python Shallow Copy and Deep Copy

Copy an Object in Python

In Python, we use `=` operator to create a copy of an object. You may think that this creates a new object; it doesn't. It only creates a new variable that shares the reference of the original object.

Let's take an example where we create a list named `old_list` and pass an object reference to `new_list` using `=` operator.

```
old_list = [[1, 2, 3], [4, 5, 6], [7, 8, 'a']]

new_list = old_list

new_list[2][2] = 9

print('Old List:', old_list)

print('ID of Old List:', id(old_list))

print('New List:', new_list)

print('ID of New List:', id(new_list))
```

When we run above program, the output will be:

```
Old List: [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
ID of Old List: 140673303268168

New List: [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
ID of New List: 140673303268168
```

As you can see from the output both variables `old_list` and `new_list` shares the same id i.e `140673303268168`.
So, if you want to modify any values in `new_list` or `old_list`, the change is visible in both.

Essentially, sometimes you may want to have the original values unchanged and only modify the new values or vice versa. In Python, there are two ways to create copies:

1. **Shallow Copy**
2. **Deep Copy**

To make these copy work, we use the `copy` module.

Copy Module

We use the `copy` module of Python for shallow and deep copy operations. Suppose, you need to copy the compound list say `x`. For example:

```
import copy
copy.copy(x)
copy.deepcopy(x)
```

Here, the `copy()` return a shallow copy of `x`. Similarly, `deepcopy()` return a deep copy of `x`.

Shallow Copy

A shallow copy creates a new object which stores the reference of the original elements.

So, a shallow copy doesn't create a copy of nested objects, instead it just copies the reference of nested objects. This means, a copy process does not recurse or create copies of nested objects itself.

Example 2: Create a copy using shallow copy

```
import copy

old_list = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]

new_list = copy.copy(old_list)

print("Old list:", old_list)

print("New list:", new_list)
```

When we run the program , the output will be:

```
Old list: [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
New list: [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

In above program, we created a nested list and then shallow copy it using `copy()` method.
This means it will create new and independent object with same content. To verify this, we print the both `old_list` and `new_list`.
To confirm that `new_list` is different from `old_list`, we try to add new nested object to original and check it.

Example 3: Adding [4, 4, 4] to old_list, using shallow copy

```
import copy

old_list = [[1, 1, 1], [2, 2, 2], [3, 3, 3]]

new_list = copy.copy(old_list)

old_list.append([4, 4, 4])

print("Old list:", old_list)

print("New list:", new_list)
```

When we run the program, it will output:

```
Old list: [[1, 1, 1], [2, 2, 2], [3, 3, 3], [4, 4, 4]]
New list: [[1, 1, 1], [2, 2, 2], [3, 3, 3]]
```

In the above program, we created a shallow copy of `old_list`. The `new_list` contains references to original nested objects stored in `old_list`. Then we add the new list i.e `[4, 4, 4]` into `old_list`. This new sublist was not copied in `new_list`.
However, when you change any nested objects in `old_list`, the changes appear in `new_list`.

Example 4: Adding new nested object using Shallow copy

```
import copy

old_list = [[1, 1, 1], [2, 2, 2], [3, 3, 3]]

new_list = copy.copy(old_list)

old_list[1][1] = 'AA'

print("Old list:", old_list)

print("New list:", new_list)
```

When we run the program, it will output:

```
Old list: [[1, 1, 1], [2, 'AA', 2], [3, 3, 3]]
New list: [[1, 1, 1], [2, 'AA', 2], [3, 3, 3]]
```

In the above program, we made changes to `old_list` i.e `old_list[1][1] = 'AA'`. Both sublists of `old_list` and `new_list` at index `[1][1]` were modified. This is because, both lists share the reference of same nested objects.

Deep Copy

A deep copy creates a new object and recursively adds the copies of nested objects present in the original elements.

Let's continue with example 2. However, we are going to create deep copy using `deepcopy()` function present in `copy` module. The deep copy creates independent copy of original object and all its nested objects.

Example 5: Copying a list using deepcopy()

```
import copy

old_list = [[1, 1, 1], [2, 2, 2], [3, 3, 3]]

new_list = copy.deepcopy(old_list)

print("Old list:", old_list)

print("New list:", new_list)
```

When we run the program, it will output:

```
Old list: [[1, 1, 1], [2, 2, 2], [3, 3, 3]]
New list: [[1, 1, 1], [2, 2, 2], [3, 3, 3]]
```

In the above program, we use `deepcopy()` function to create copy which looks similar.
However, if you make changes to any nested objects in original object `old_list`, you'll see no changes to the copy `new_list`.

Example 6: Adding a new nested object in the list using Deep copy

```
import copy

old_list = [[1, 1, 1], [2, 2, 2], [3, 3, 3]]

new_list = copy.deepcopy(old_list)

old_list[1][0] = 'BB'

print("Old list:", old_list)

print("New list:", new_list)
```

When we run the program, it will output:

```
Old list: [[1, 1, 1], ['BB', 2, 2], [3, 3, 3]]
New list: [[1, 1, 1], [2, 2, 2], [3, 3, 3]]
```

In the above program, when we assign a new value to `old_list`, we can see only the `old_list` is modified. This means, both the old_list and the `new_list` are independent. This is because the `old_list` was recursively copied, which is true for all its nested objects.

References:

https://www.geeksforgeeks.org/copy-python-deep-copy-shallow-copy/

https://www.programiz.com/python-programming/shallow-deep-copy