Copy Operation

11 June 2025 12:04

Definition:

It is a phenomenon of copying the content from one variable to another Variable.

Example: L=[10,20,30]

```
L[1]=200
L
Output:-[10,20,30]
```

Now suppose you want the **original list** back without the modification. If you just modify the list directly, the original data is lost

To handle such cases, we need to **make a copy** of the original list **before modification**

```
b=[10,20,30]
```

In the above example, we first created a list and modified it using indexing. But later, we realized we wanted to keep the original values. If we modify the list directly, the original data gets changed.

To solve this, we can make a **copy of the original list** before making any modifications. This way, we preserve the original data in one variable (b) and can perform modifications on another (L).

This process is called a **copy operation**. It is useful when:

- You want to keep the original data unchanged.
- You need to create a separate modified version for another purpose.

Generally copy operation can be done on any values till dictionary. but most of the times will be using this copy operation in case of "List".

Q. Why List Only?

Copy operations are mostly needed with **mutable collections** like lists, dictionaries, and sets. Among these, **lists** are most frequently used in real-world Python programs, hence copy operations are most common with lists.

Copy Operation is Classified into 3 Types:

1.general copy/Normal copy
2.shallow copy
3.deep copy

1.General Copy:

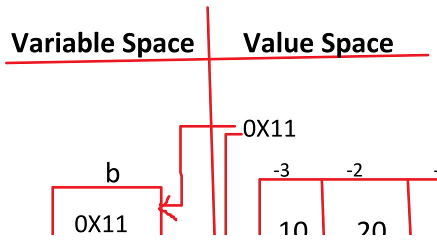
- It is a phenomenon of copying the content from one variable to another variable.[Variable space of one variable to another variable].

Syntax: Destination variable=Source Variable

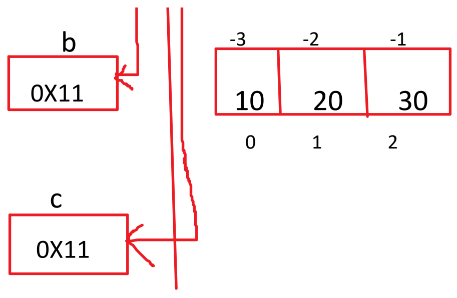
- The general copy can be applied on any data type

Example:

```
b=[10,20,30]
c=b
b
Output:-[10, 20, 30]
c
Output:-[10, 20, 30]
```



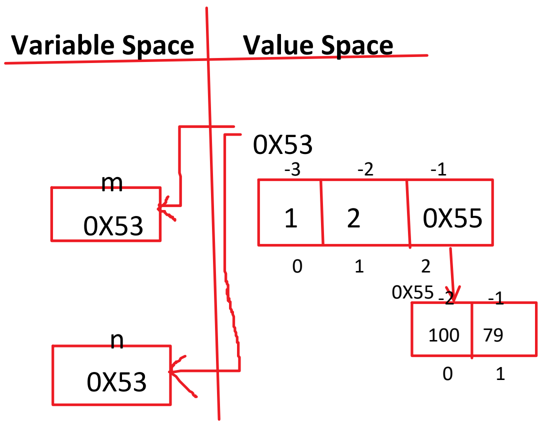
```
Output:-[10, 20, 30]
c
Output:-[10, 20, 30]
id(b)
Output:-2795202190528
id(c)
Output:-2795202190528
Modification With Respect to 'b':
b[0]=25
b
[25, 20, 30]
c
[25, 20, 30]
```



In the above example, we assigned b to c using c = b. This doesn't create a new list — it makes c point to the **same list** as b.
So when we modify b, the changes are also visible in c.
This happens because **both b and c refer to the same memory address**.
Therefore, a change in one variable **affects the other**. This is known as **general copy** or **reference copy**.

Example: 2----->Nested List

```
m=[1,2,[100,79]]
n=m
id(m)
Output:-2795203596864
id(n)
Output:-2795203596864
n
Output:-[1, 2, [100, 79]]
m
Output:-[1, 2, [100, 79]]
m[2][1]
Output:-79
m[2][1]=60
m
Output:-[1, 2, [100, 60]]
n
Output:-[1, 2, [100, 60]]
```



In the above example, we used a **nested list** and created a general copy using n = m.

- Both m and n point to the **same memory location**, as shown by the identical id() values.
- When we modify a value inside the nested list (e.g., m[2][1] = 60), the change is also reflected in n

This demonstrates that:
Even in **nested lists**, general copy does not create a new object it simply creates a new reference to the same object.
So, any modification done through one variable will also affect the other, whether the list is:

- A **linear list** (flat)
- Or a **nested list**

This is because both variables share the **same address in memory**.

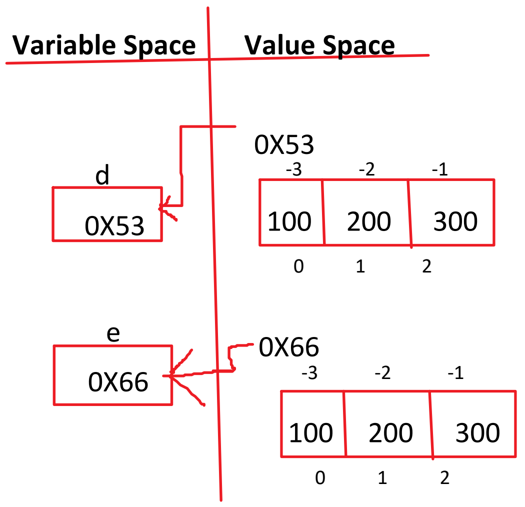
2.Shallow Copy:

- It is a phenomenon of copying the content from one variable to another variable.[value space of one variable to another variable].

Syntax: Destination variable=Source Variable. Copy()

Example:-

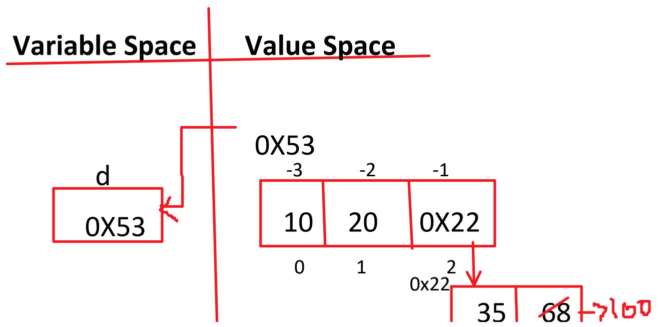
```
d=[100,200,300]
e=d. copy()
d
[100, 200, 300]
e
[100, 200, 300]
id(d)
2701938119808
id(e)
2701938122112
```
Modification With Respect to 'd'
```
d[1]=500
d
[100, 500, 300
e
[100, 200, 300]
```
Modification With Respect to 'e'
```
e[0]=30
e
[30, 200, 300]
d
[100, 500, 300]
```



In the above example, we created a list d and then made a **shallow copy** of it using e = d.copy().
This means that all the elements (values) of list 'd 'are copied into list 'e'.
However, the memory address of e is different from d. So, e is a separate list stored at a new location in memory.
Now, when we modify 'e', it does **not** affect 'd', and when we modify d, it does **not** affect e.
This is because **only the values are copied**, not the reference (memory address).
This behaviour is known as a **shallow copy**.

Example 2:- nested list

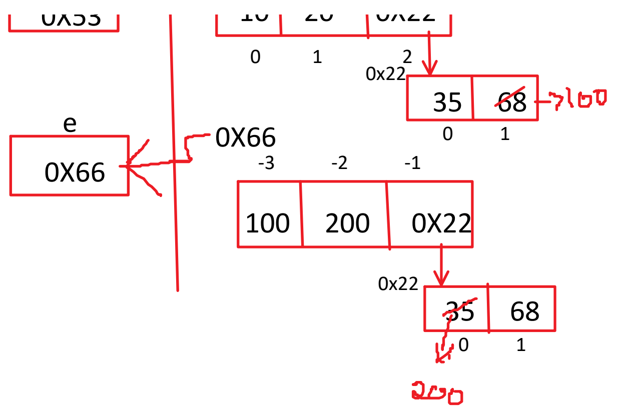
```
j=[10,20,[35,68]]
j
Output:-[10, 20, [35, 68]]
k=j
k
Output:-[10, 20, [35, 68]]
```
Modification With Respect to 'j'
```
j[2][1]
Output:-68
j[2][1]=100
j
Output:-[10, 20, [35, 100]]
```



```
j[2][1]=100
j
Output:-[10, 20, [35, 100]]
k
[10, 20, [35, 100]]
```

Modification With Respect to 'k'

```
k[2][0]=200
k
Output:-[10, 20, [200, 100]]
j
Output:-[10, 20, [200, 100]]
```



In the case of a linear list, if you make any modifications, they will not affect the original list because both variables point to different addresses.
However, for nested lists, if you make modifications, both variables will be affected because the nested elements still point to the same memory address.

Deep Copy:

- It is a phenomenon of copying the content of variable directly .it will not worry about memory allocation

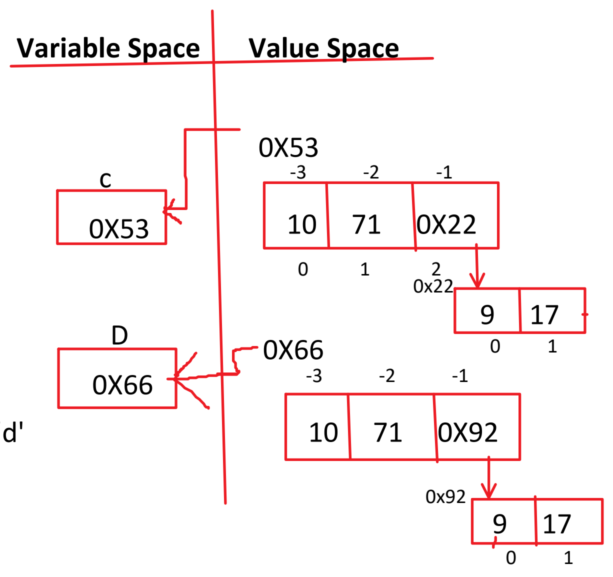
Syntax: import Copy
destination variable=copy.deepcopy(source variable)

Example:

```
c=[10,71,[9,17]]
import copy
d=copy. deepcopy(c)
id(d)
1628529342208
id(c)
1628529334400
d
Output:-[10, 71, [9, 17]]
c
Output:-[10, 71, [9, 17]]
```

Modification with respect to 'd'

```
d[2][0]=100
d
[10, 71, [100, 17]]
c
[10, 71, [9, 17]]
c[2][1]=300
c
[10, 71, [300, 100]]
d
[10, 71, [100, 17]]
```



in case of Deep copy both linear and nested values will not get affected.