Python Bootcamp & Masterclass

copying lists

gknxt

Lists are mutable

Lists are mutable. So, new elements can be added, existing elements can be modified and/or deleted.

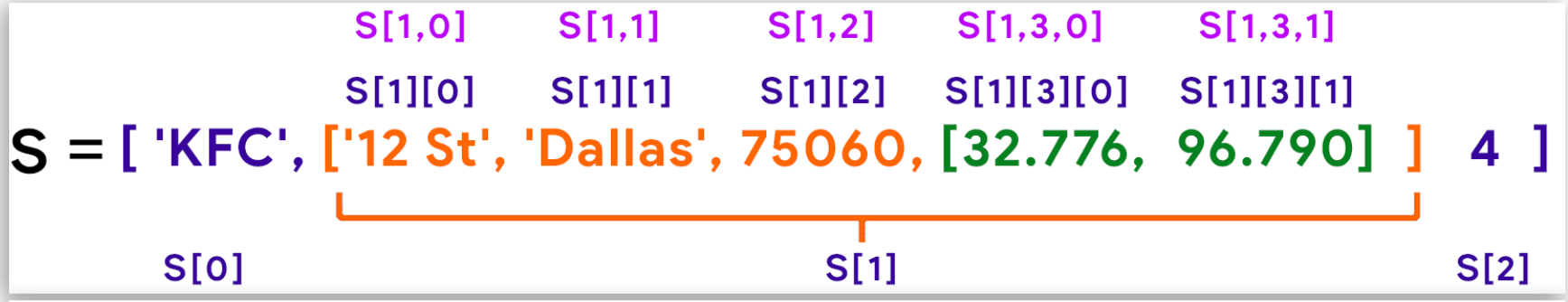
```
list1 = [1, 2, 3, 'a', 'b']
print(f"Original list {list1} at \t\t\t {hex(id(list1))}")
list1.remove('b')
print(f"List after deleting 'b' is {list1} at \t\t {hex(id(list1))}")
list1.append('c')
print(f"List after appending 'c' is {list1} at \t {hex(id(list1))}")
list1[4] = 'z'
print(f"List after modifing 'c' to 'z' is {list1} at {hex(id(list1))}")
```

```
Original list [1, 2, 3, 'a', 'b'] at                    0x200165ffe40
List after deleting 'b' is [1, 2, 3, 'a'] at            0x200165ffe40
List after appending 'c' is [1, 2, 3, 'a', 'c'] at      0x200165ffe40
List after modifing 'c' to 'z' is [1, 2, 3, 'a', 'z'] at 0x200165ffe40
```

Nested lists

A compound object (lists, sets, dicts etc) can contain compound objects.

They can be nested to any level needed.




```
s = ['KFC', ['12 St', 'Dallas', 75060, [32.776, 96.791]], 4]
s[1]
s[-2]
```

```
['12 St', 'Dallas', 75060, [32.776, 96.791]]

['12 St', 'Dallas', 75060, [32.776, 96.791]]
```

gk nxt

Unpacking a list of lists

There are several ways in which a list of lists can be unpacked (flattened)

Unpacking operator (*) can be used for unpacking a list of lists.

```
n = [[0, 1, 2.2], ['a', 'e', 12], ['Hello', 21, True]]
p = [*n[0], *n[1], *n[2]]   # unpacking (* is unpacking operator)
p
```

```
[0, 1, 2.2, 'a', 'e', 12, 'Hello', 21, True]
```

```
q = [[0, 1, 2.2], ['a', 'e', 12], ['Hello', 21, True]]
r = sum(q, [])
r
```

```
[0, 1, 2.2, 'a', 'e', 12, 'Hello', 21, True]
```

```
s = ['KFC', ['12 St', 'Dallas', 75060, [32.776, 96.791]], 4]
t = [s[0], *s[1], s[2]]
u = [t[0], t[1], t[2], t[3], *t[4], t[5]]
u
```

```
['KFC', '12 St', 'Dallas', 75060, 32.776, 96.791, 4]
```

Assignment statements

Assignment statements do not copy objects, they create bindings between a target and an object. So, assignment statement, instead of creating a new object, will simply point the new variable towards the existing object.

```
orig1 = [1, ['a', 2, 5.5], True]
copy1 = orig1  # No new object created, copy1 and orig1 points to the same object
id(copy1)
id(orig1)
copy1.append("New")
copy1
orig1
```

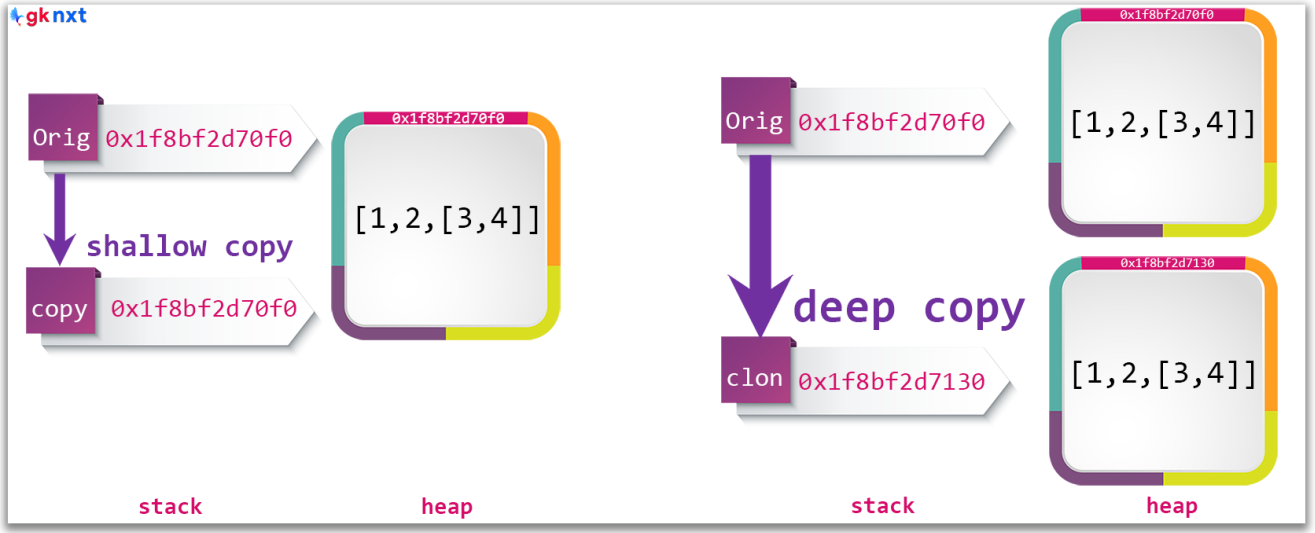
```
2199382473728

2199382473728

[1, ['a', 2, 5.5], True, 'New']

[1, ['a', 2, 5.5], True, 'New']
```

Shallow copy & deep copy

Assignment statements in Python do not create copies - they only bind names to objects. Sometimes copies of mutable objects or collections of mutable objects would be needed for data processing.

There are two ways to make copies of the objects:

1) shallow copy
2) deep copy

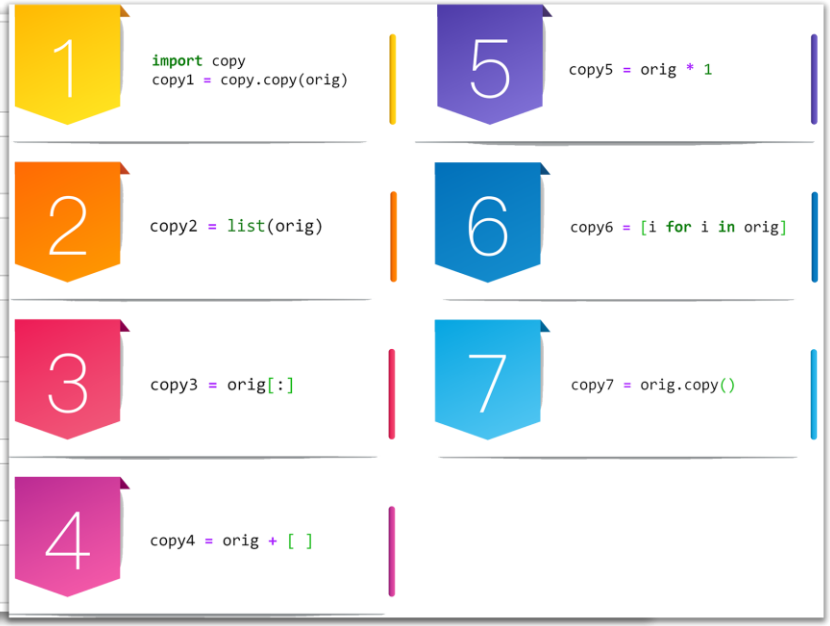


gknxt

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

A **shallow copy** can be created in many ways.

```
import copy
copy1 = copy.copy(orig)

copy2 = list(orig)

copy3 = orig[:]

copy4 = orig + [ ]

copy5 = orig * 1

copy6 = [i for i in orig]

copy7 = orig.copy()
```

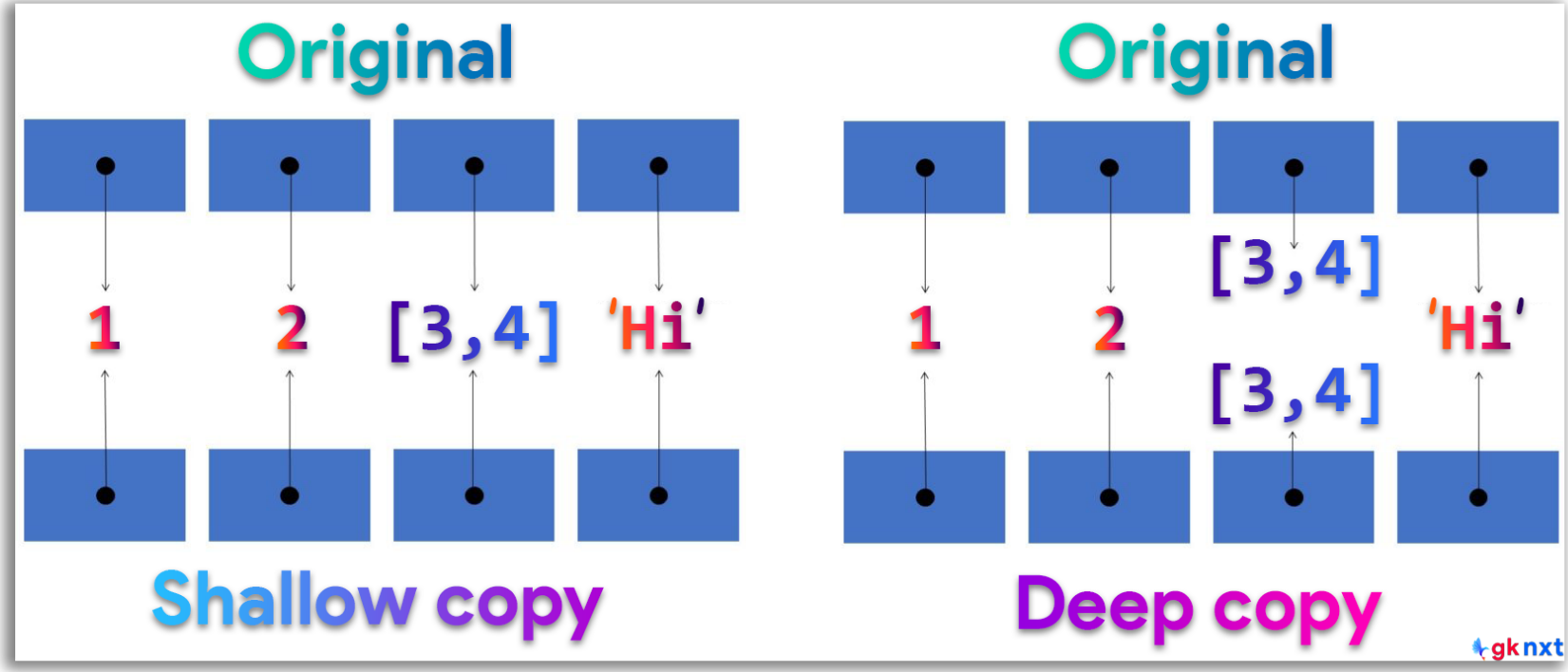


A **deep copy** constructs a new compound object and then, recursively, inserts copies into it of the objects found in the original. So, **deep copy** creates a new copy at a different memory location with no connection to the original object whatsoever. It can be created with the **deepcopy()** method from the **copy** module.

```
import copy
orig10 = [1, ['a', 2, 5.5], True]
copy10 = copy.deepcopy(orig10)
id(copy10)
id(orig10)
copy10[1][0] = 'z'
copy10
orig10
```

```
2199382534080

2199382541888

[1, ['z', 2, 5.5], True]

[1, ['a', 2, 5.5], True]
```

For immutable objects, shallow and deep copy are same. The difference between shallow and deep copying is only relevant for compound objects (objects that contain other objects) with mutable objects.



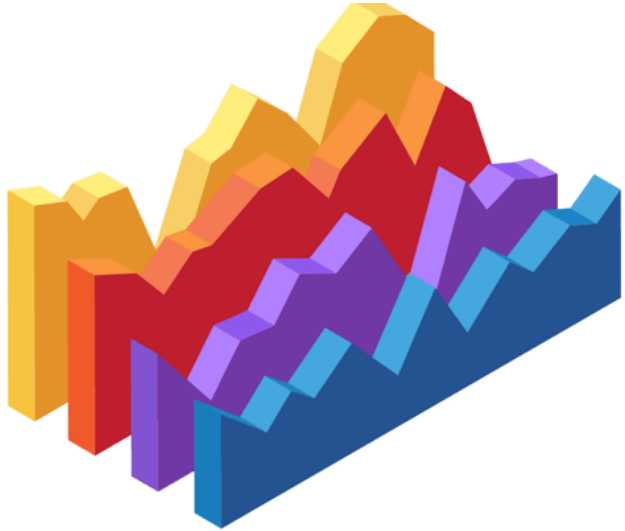
Differences between shallow copy & deep copy

	Shallow Copy	Deep Copy
Creation	Shallow copy can be created in many ways	Deep copy can only be created with deepcopy( )
Speed	Operations with shallow copy are faster	Operations with deep copy are slower
Memory	Shallow copy uses memory effectively	Deep copy does not use memory effectively
Errors	Shallow copying, in general, does not cause errors	Recursive objects may cause a recursive loop
Data	Generally, some data will be shared between copies made with shallow copy	No data is shared between copies made with deep copy
Process	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.

Shallow copy creates a new object and then populates it with **references** to the child objects (nested objects) found in the original. So, child objects in the original are not copied, just their references are copied. So, any changes to the child objects in the original will reflect in the copy and any changes to the child objects in the copy will reflect in the original. Therefore, the copy is not fully independent of the original

Deep copy creates a new object and then recursively populates it with copies of all objects including the child objects (nested objects) found in the original. So, deep copy creates a clone at a different memory location, so the original and the clone are two separate, independent objects and any changes made to one have no effect on the other.

Deep Copy is necessary when the original objects should not be modified. Deep copy is especially important in multithreading and multi-core programming, where separate parts of a program could attempt to modify the data at the same time, possibly corrupting it. So, having two independent copies is necessary in such situations.



Online Resources

For best python resources, please visit:

gknxt.com/python/

gknxt

Python
Bootcamp
& Masterclass

Thank You

for your Rating & Review

gknxt