1. **What exactly is [ ]?**

Ans. [] is an empty list in python. Whenever, we want to assign a variable as a list type so we simply write it like **a = [ ]**

1. **In a list of values stored in a variable called spam, how would you assign the value 'hello' as the third value? (Assume [2, 4, 6, 8, 10] are in spam.)**

Ans. spam[2] = “hello”  
(Note: Index starts from 0. So, 3rd Value will be at index 2)

**Let's pretend the spam includes the list ['a', 'b', 'c', 'd'] for the next three queries.**

1. **What is the value of spam[int(int('3' \* 2) / 11)]?**

Ans. ‘d’

1. **What is the value of spam[-1]?**

Ans. ‘d’

1. **What is the value of spam[:2]?**

Ans. [‘a’, ’b’]

**Let's pretend bacon has the list [3.14, 'cat,' 11, 'cat,' True] for the next three questions.**

1. **What is the value of bacon.index('cat')?**

Ans. 1

1. **How does bacon.append(99) change the look of the list value in bacon?**

Ans. It will append 99 at the end of the list like [3.14, 'cat', 11, 'cat', True, 99]

1. **How does bacon.remove('cat') change the look of the list in bacon?**

Ans. It will remove the first occurrence of “cat” from the list. And new list will be:

[3.14, 11, 'cat', True, 99]

1. **What are the list concatenation and list replication operators?**

Ans. + and \*

Let a=[1,2,3] and b=[4,5,6]

Then a+b= [1,2,3,4,5,6] and a\*3= [1,2,3,1,2,3,1,2,3]

1. **What is difference between the list methods append() and insert()?**

Ans. append(): It adds the data value at the end of the list and takes one argument.

Insert(): It takes 2 arguments (index, object) and inserts the object at a particular index.

1. **What are the two methods for removing items from a list?**

Ans. 1) remove(): It removes the first occurrence of the element from the list.

2) pop(): It removes the occurrence of the element at a particular index.

1. **Describe how list values and string values are identical.**

Ans. The similarity between Lists and Strings in Python is **that both are sequences**.

1. **What's the difference between tuples and lists?**

Ans. Tuples are immutable while Lists are mutable. Tuples are **more memory efficient** than the lists.

1. **How do you type a tuple value that only contains the integer 42?**

Ans. tup=(42**,**)

We specifically need to type comma (**,**) after integer value otherwise Python will understand it as Integer value only and not as a tuple.

1. **How do you get a list value's tuple form? How do you get a tuple value's list form?**

Ans.

aTuple = (67, 28, 41, 37)

aList = list(aTuple)

print(type(aList)) ====== <class ‘list’>

print(aList) ====== [67, 28, 41, 37]

aList = [True, 28, 'Tiger])

aTuple = tuple(aList)

print(type(aTuple)) ====== <class 'tuple'>

print(aTuple) ====== (True, 28, 'Tiger')

1. **Variables that "contain" list values are not necessarily lists themselves. Instead, what do they contain?**

Ans. Variables that contain list values are not neccesarily lists themselves as they can be tuple, dictionary also.

Ex:

A= [ [1,2], [“hello,1] ]

B= ( [1,2], [“hello,1] )

C= {“D”: [1,2], “E”: [3,4] }

1. **How do you distinguish between copy.copy() and copy.deepcopy()?**

Ans. copy.copy()= 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.

### 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.