TYPE-A:LIST MANIPULATION:CH-11

1) Discuss the utility and significance of Lists in Python, briefly.

sol:

Lists in Python are ordered, mutable collections that can store items of different types. They are significant because they allow easy storage, access, and manipulation of multiple values in a single variable. Lists provide powerful features such as indexing, slicing, appending, inserting, and deleting elements, making them essential for organizing data and performing iterative operations efficiently in programs.

2) What do you understand by mutability? What does "in place" memory updation mean?

sol:

Mutability refers to whether an object's value can be changed after it is created. Mutable objects, like lists, can be modified, while immutable objects, like strings, cannot. "In place" memory updation means that the object's existing memory location is modified directly when its value changes, instead of creating a new object in memory.

3) Start with the list [8, 9, 10]. Do the following using list functions:

1. Set the second entry (index 1) to 17
2. Add 4, 5 and 6 to the end of the list
3. Remove the first entry from the list
4. Sort the list
5. Double the list
6. Insert 25 at index 3

sol:

a) lst[index] = value → lst[1] = 17
b) extend() → lst.extend([4, 5, 6])
c) pop() → lst.pop(0)
d) sort() → lst.sort()
e) * operator → lst = lst * 2
f) insert() → lst.insert(3, 25)

4) If a is [1, 2, 3]

a) what is the difference (if any) between a * 3 and [a, a, a]?
b) is a * 3 equivalent to a + a + a?
c) what is the meaning of a[1:1] = 9?
d) what's the difference between a[1:2] = 4 and a[1:1] = 4?

sol:

a) a * 3 repeats the elements of the list three times to give [1, 2, 3, 1, 2, 3, 1, 2, 3], while [a, a, a] creates a new list containing three references to the original list a: [[1, 2, 3], [1, 2, 3], [1, 2, 3]].

b) Yes, a * 3 is equivalent to a + a + a, both produce [1, 2, 3, 1, 2, 3, 1, 2, 3].

c) a[1:1] = 9 will raise an error because the right-hand side of a slice assignment must be an iterable (like a list). Correct usage would be a[1:1] = [9], which inserts 9 at index 1 without removing any elements.

d) a[1:2] = 4 will raise an error for the same reason; it tries to replace the slice at index 1 with a non-iterable. Correctly, a[1:2] = [4] replaces the element at index 1 with 4. In contrast, a[1:1] = [4] inserts 4 at index 1 without removing any elements. The difference is that one replaces, the other inserts.

5) What's a[1 : 1] if a is a list of at least two elements? And what if the list is shorter?

sol:

For a list a with at least two elements, a[1:1] is an empty list [] because the start and end index are the same, so no elements are selected.

If the list has fewer than two elements (for example, length 0 or 1), a[1:1] still returns an empty list []. Python does not raise an error; it just returns an empty slice.

6) How are the statements lst = lst + 3 and lst += [3] different, where lst is a list? Explain.

sol:

The statements lst = lst + 3 and lst += [3] are different because:

- lst = lst + 3 is invalid and will raise a TypeError since you cannot add a list and an integer directly. To make it work, you would need to write lst = lst + [3]. This creates a new list by concatenation and assigns it back to lst.
- lst += [3] is valid and works correctly. It modifies the original list in place by appending 3 to it, without creating a new list.

So the main difference is that + creates a new list, while += updates the existing list.

7) How are the statements lst += "xy" and lst = lst + "xy" different, where lst is a list? Explain.

sol:

When lst is a list:

- lst += "xy" works and modifies the list in place, adding each character of the string "xy" as separate elements to the list. For example, if lst = [1, 2], after lst += "xy" it becomes [1, 2, 'x', 'y'].
- lst = lst + "xy" raises a TypeError because + expects another list for concatenation, not a string. You would need to write lst = lst + list("xy") to make it work.

The main difference is that += can extend the list with any iterable in place, while + requires compatible types and creates a new list.

8) What's the purpose of the del operator and pop method? Try deleting a slice.

sol:

Feature	del Operator	pop() Method
Purpose	Deletes an element, a slice, or entire list	Removes and returns an element at a specified index
Effect	Permanently removes elements from the list	Removes element but can use its value
Example	del lst[1:4] → removes a slice	x = lst.pop(2) → removes element at index 2 and returns it

9) What does each of the following expressions evaluate to? Suppose that L is the list ["These", ["are", "a", "few", "words"], "that", "we", "will", "use"].

A. L[1][0::2]
B. "a" in L[1][0]
C. L[:1] + L[1]
D. L[2::2]
E. L[2][2] in L[1]

sol:
A. ['are', 'few']
B. True
C. ['These', 'are', 'a', 'few', 'words']
D. ['that', 'will']
E. False

10) What are list slices? What for can you use them?

sol:
List slices are portions of a list selected using a range of indexes. They allow you to access, modify, or extract multiple elements at once. You can use slices to read a part of the list, replace elements, insert new elements, or delete elements efficiently.

11) Does the slice operator always produce a new list?

sol:
Yes, the slice operator in Python always produces a new list. It does not modify the original list unless you explicitly assign values to the slice. Changes made to the new list do not affect the original list unless you use assignment.

12) Compare lists with strings. How are they similar and how are they different?

sol:

Feature	Lists	Strings
Mutability	Mutable (can change elements)	Immutable (cannot change)
Data Type of Items	Can store different data types	Only characters

Indexing/Slicing	Supported	Supported
Iteration	Supported	Supported
Use Cases	Store collections of items	Store text or characters

13) What do you understand by true copy of a list? How is it different from shallow copy?

sol:

A true copy (or deep copy) of a list is a copy in which all elements, including nested lists, are copied as new objects. Changes made to the copied list do not affect the original list at any level.

A shallow copy creates a new list, but the nested elements are still references to the original objects. Changes to nested elements in the shallow copy will affect the original list.

14) An index out of bounds given with a list name causes error, but not with list slices. Why?

sol:

When you access a list using a single index, Python expects the index to be within the valid range, otherwise it raises an IndexError. But with list slices, Python automatically adjusts out-of-range start or end indexes to the nearest valid index and returns the corresponding elements (or an empty list), so no error occurs.

15) What is the difference between appending a list and extending a list?

sol:

Feature	**append()**	**extend()**
Purpose	Adds a single element to the end	Adds all elements of an iterable to the end
Argument	Any object (including a list)	An iterable (like list, tuple, string)
Effect on list	Adds the object as a **single element**	Adds each element of the iterable individually
Example	lst.append([4,5]) → [1,2,3,[4,5]]	lst.extend([4,5]) → [1,2,3,4,5]

16) Do functions max( ), min( ), sum( ) work with all types of lists.

sol:

The functions max(), min(), and sum() do not work with all types of lists. They work only with lists that contain numeric values (for sum()) or comparable elements (for max() and min())Lists containing mixed data types like numbers and strings, or only non-numeric values, will raise an.error.

17) What is the difference between sort( ) and sorted( )?

sol:

Feature	sort()	sorted()
Type	List method	Built-in function
Effect on list	Sorts the original list **in place**	Returns a **new sorted list** without changing the original
Return value	None	New sorted list
Usage example	lst.sort()	new_lst = sorted(lst)