<u>Assignment 4</u>

<u>Answer 1</u>:-

[] represents an empty list, which is a built-in data structure used to store an ordered collection of items. Lists are one of the most versatile and commonly used data structures in Python because they can hold elements of any data type (e.g., integers, strings, other lists).

<u>Empity list</u>:-

empity_list=[ ]

<u>Answer 2</u>:-

```
# Initial list
spam = [2, 4, 6, 8, 10]

# assign ‘hello’ as the third value
spam[2]=‘hello’

# resulting list
print(spam)
```

<u>Answer 3</u>:-

Let's break down the expression spam[int(int('3'*2)/11)] step by step, assuming spam = ['a', 'b', 'c', 'd']:

Step-by-Step Breakdown:

1. **'3' * 2:**
   This repeats the string '3' two times, resulting in '33'.
2. **int('33'):**
   Converts the string '33' into the integer 33.
3. **33 / 11:**
   Divides 33 by 11, resulting in 3.0.
4. **int(3.0):**
   Converts 3.0 into the integer 3.
5. spam[3]:
   Accesses the element at index 3 of the list spam.
   Since spam = ['a', 'b', 'c', 'd'], the element at index 3 is 'd'.

<u>Answer 4</u>:-

when spam = ['a', 'b', 'c', 'd'], using spam[-1] accesses the last element of the list.

So, the value of spam[-1] is ‘d’.

Answer 5:-

when spam = ['a', 'b', 'c', 'd'], using spam[:2] accesses the first two elements of the list.
So, the value of spam[‘a’, ‘b’].

Answer 6:-

The value of bacon.index('cat') in Python will be 1.

This is because the index() method returns the first occurrence of the specified element in the list, and 'cat' first appears at index 1 in the list [3.14, 'cat', 11, 'cat', True].

Answer 7:-

If bacon starts with the list [3.14, 'cat', 11, 'cat', True] and you execute bacon.append(99), it adds 99 to the end of the list. The updated list will look like this:

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

Answer 8:-

If bacon starts with the list [3.14, 'cat', 11, 'cat', True] and you execute bacon.remove('cat'), it removes the first occurrence of 'cat' from the list.
The updated list will look like this:

[3.14, 11, 'cat', True]

Answer 9:-

List Concatenation and Replication:-

As with strings, we can use the operators + and * to concatenate and replicate lists.

When + appears between two lists, the expression will be evaluated as a new list that contains the elements from both lists. The elements in the list on the left of + will appear first, and the elements on the right will appear last.

When * appears between a list and an integer, the expression will be evaluated as a new list that consists of several copies of the original list concatenated together. The number of copies is set by the integer.

Answer 10:-

The append() and insert() methods in Python are used to add elements to a list, but they work differently:

1. **append()**

- Purpose: Adds an element to the end of the list.
- Syntax: list.append(element)

Key Features:
- Accepts a single argument (the element to be added).
- Modifies the original list.
- Does not allow you to specify the position of the new element.

2. **Insert():**

- Purpose: Adds an element at a specific position in the list.
- Syntax: list.insert(index, element)

Key Features:
- Requires two arguments: the index (position) and the element to insert.
- Shifts elements at and after the specified index one position to the right.
- Modifies the original list.

Answer 11:-

The two most common ones are remove() and pop() are as follows:

1. **remove()**
- Purpose: Removes the first occurrence of a specified value from the list.
- Syntax: list.remove(value)

Key Features:
- You specify the value to remove.
- If the value is not found, it raises a ValueError.
- Modifies the original list.

2. **pop()**

- Purpose: Removes and returns an element at a specified index (default is the last item).
- Syntax: list.pop([index])

Key Features:
- Optional index argument (if omitted, it removes the last element).
- If the index is out of range, it raises an IndexError.
- Modifies the original list and provides access to the removed element**.**

Answer 12:-

1. Indexing:

- Both lists and strings allow accessing elements by their index, starting from 0.

2. Slicing

- You can extract subsets (slices) of both lists and strings using slicing syntax (start:end:step)**.**

3. Length

- The len() function works for both lists and strings, returning the number of elements or characters**.**

4. Iteration

- Both lists and strings can be iterated using loops (e.g., for loops).

Answer 13:-

The difference between list and tuple are:-

1. Lists are mutable whereas the tuples are of immutable data type.
2. The implication of iterations is Time-consuming whereas the tuples iterations are comparatively Faster.
3. Lists consume more memory and Tuple consumes less memory as compared to the list.
4. Lists have several built-in methods whereas tuple consumes less memory as compared to the list.

Answer 14.

To create a tuple in Python that contains only the integer 42, you need to include a trailing comma after the value to distinguish it from a plain integer. Here's how you do it:

my_tuple=(42,)

Without the comma, it would just be interpreted as a parenthesized integer:

not_a_tuple=(42)

The trailing comma is essential for creating a single-element tuple.

Answer 15.

Getting a List from a Tuple
Use the list() function to convert a tuple to a list:

```
# Example Tuple
my_tuple = (1, 2, 3, 4)
```

```
# Convert to List
my_list = list(my_tuple)
print(my_list)
```

Output:-[1,2,3,4]

<u>Answer 16</u>:-

Variables that appear to "contain" list values actually contain references to lists, not the lists themselves. In Python, lists are mutable objects, and variables store a reference (or pointer) to the memory location where the list object is stored.

Explanation:

When you assign a list to a variable, the variable holds a reference to the list object in memory. If you assign the same list to another variable or pass it to a function, both variables point to the same list object, and changes to one affect the other.

<u>Answer 17:-</u>

1. **copy.copy() (Shallow Copy)**

Creates a shallow copy of an object.
Copies the outer object, but does not recursively copy nested objects (e.g., lists or dictionaries inside the object).
Changes to mutable objects (like lists or dictionaries) within the copied object will affect the original object.

2. **copy.deepcopy() (Deep Copy)**

Creates a deep copy of an object.
Recursively copies the object and all objects contained within it.
Changes to mutable objects inside the copied object will not affect the original object