1. **‘[]’** is used to denote an **empty list**. A list is a built-in data structure that holds an ordered collection of items. The items in a list can be of any data type, and they are enclosed within square brackets [], separated by commas.
   Example :- numbers = [1, 2, 3, 4, 5]

2. spam = [2, 4, 6, 8, 10]

   # Assign the string "hello" to the third value in the list
   spam[2] = "hello"

   # Print the list
   print(spam)

   output :- [2, 4, 'hello', 8, 10]

3. **spam=[a,b,c,d]**
   spam[int(int('3'* 2) / 11)] evaluates 3rd element of list ie ‘ **d’**

4. spam[-1] is last element of list **‘d’**

5. spam[:2] is called slicing. It will select portion of the list from the beginning up to, but not including, the element at index 2. Therefore answer will **[a,b,]**

6. **bacon = [3.14, 'cat', 11, 'cat', True]**

   bacon.index('cat') will give index of cat(first occurrence) ie **1**

7. The bacon.append(99) operation adds the value 99 to the end of the list. new list after operation will be :-
   bacon = [3.14, 'cat', 11, 'cat', True, 99]

8. The first occurrence of 'cat' will be removed from the list bacon.now list will look like:-
   bacon = [3.14, 11, 'cat', True, 99].

9. The list concatenation operator allows you to concatenate, or combine, two or more lists into a single list using a **'+' operator.**

   example of list concatenation:

   list1 = [1, 2, 3]
   list2 = [4, 5, 6]
   concatenate_list = list1 + list2
   print(concatenate_list)

   Output:
   [1, 2, 3, 4, 5, 6]

The list replication operator is the **asterisk (*).** It allows you to replicate, or repeat, a list multiple times.

example of list replication:

```
list1 = [1, 2, 3]
replicate_list = list1 * 3
print(replicate_list)
```

Output:
```
[1, 2, 3, 1, 2, 3, 1, 2, 3]
```

10. The difference between append() and insert() is that append() always adds elements to the end of the list, while insert() allows you to specify the index where the element should be inserted.

    Example :- **for append()**

    ```
    list = [1, 2, 3]
    list.append(4)     # adding value '4' at the end of the list
    print(list)
    ```

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

    Example **for insert() :-**

    ```
    list2 = [1, 2, 3]
    list2.insert(1, 4)    # insert value '4' at index '1' in list2
    print(list2)
    ```

    Output: [1, 4, 2, 3]

11. Two methods for removing items from list are :-

    a) **remove():** The remove() method is used to remove the first occurrence of a specified value from a list

    ```
    list = [1, 2, 3, 2]
    list.remove(2)
    print(list)
    ```

    Output:
    ```
    [1, 3, 2]
    ```

    Here in above code the 'remove(2)' method call removes the first occurrence of the value 2 from the list

b) **pop():** The pop() method is used to remove an element from a list based on its index. It modifies the list and returns the removed element.

```
list2 = [1, 2, 3]
removed_element = list2.pop(1)
print(list2)
print(removed_element)
```

Output:

```
[1, 3]
2
```

In the example above, the pop(1) method call removes the element at index 1 (which is 2) from the list2 and returns it. The list is modified, and the removed element is stored in the variable removed_element.

12. List values and string values share following similarities:-

a)**Indexing:** Both lists and strings support indexing. You can access individual elements with the corresponding index.

Example:

```
list = [1, 2, 3]
str = "Hello"
print(list[0])  # Output: 1
print(str[1])  # Output: 'e'
```

b) **Slicing**: Both lists and strings support slicing, which allows you to extract a portion of the sequence by specifying a start and end index.

Example:

```
list = [1, 2, 3, 4, 5]
string = "Hello, vipin!"
print(list[1:4])  # Output: [2, 3, 4]
print(string[7:])  # Output: "vipin!"
```

c) **Length:** Both lists and strings have a length that can be obtained using 'len()' function.

Example:

```
list = [1, 2, 3]
str = "games"
print(len(list))  # Output: 3
print(len(str))  # Output: 5
```

c)**Iteration:** Both lists and strings can be iterated over using loops

Example:
```
list = [1, 2, 3]
string = "game"
for item in list:
   print(item)
for char in string:
   print(char)
```

Output:
```
1
2
3
g
a
m
e
```

d)**Concatenation**: Both lists and strings support concatenation, allowing you to combine multiple sequences into a single sequence.

Example:
```
list1 = [1, 2, 3]
list2 = [4, 5, 6]
concatenate_list = list1 + list2

string1 = "Hello"
string2 = "World!"
concatenate_string = string1 + ", " + string2
```

Output:
```
concatenate_list: [1, 2, 3, 4, 5, 6]
concatenate_string: "Hello, World!"
```

13. Fundamental difference between list and tuples are as following:-

Syntax: Lists are defined using square brackets **('[]')**, while tuples are defined using parentheses ('**()**').

Example of a list:
list = [1, 2, 3]

Example of a tuple:
tuple = (1, 2, 3)

Mutability: Lists are mutable, meaning you can modify, add, or remove elements after the list is created. Tuples, on the other hand, are immutable, meaning they cannot be modified once created

Example of a list:
list = [1, 2, 3]
list[0] = 6 **# Modify the list**
print(list) # **Output: [6, 2, 3]**

Example of a tuple:
tuple = (1, 2, 3)
tuple[0] = 4 # Trying to modify the tuple - **raises an error**

14. To type a tuple value that only contains the integer 42, we use code **'tuple_value = (42,)'** print(tuple_value) will print 42 only as output

15. To convert a list value to its tuple form, you can use the tuple() function as follow:-
list = [1, 2, 3]
main_tuple = tuple(list)
print(main_tuple)

Output:
(1, 2, 3)

To convert a tuple value to its list form, you can use the list() function, which takes an iterable (such as a tuple) as an argument and returns a list containing the same elements. Here's an example:

tuple = (1, 2, 3)
list = list(tuple)
print(list)

Output:
[1, 2, 3]

By using the tuple() function, you can convert a list to a tuple, and by using the list() function, you can convert a tuple to a list.

16. Variables that "contain" list values in Python are actually **references or pointers** to the list objects. In other words, the variables store **memory addresses that point to the location where** the list is stored in the computer's memory

17. **copy.copy()** :- It creates a new object and then copies the references to the nested objects found within the original object. In the case of a list, a shallow copy creates a new list object but copies the references to the elements of the original list. If the elements are mutable objects, changes made to the original elements will be reflected in the copied lHowever, if new elements are added to or removed from the copied list, the original list remains unaffected.

```
old_list = [1, [2, 3]]
new_list = copy.copy(old_list)

old_list[0] = 100
old_list[1].append(4)

print(old_list)     # Output: [100, [2, 3, 4]]
print(new_list)     # Output: [1, [2, 3, 4]]
```

In the example above, modifying the original list and its nested object (old_list[1]) affects both the original and copied lists because they share the same nested object reference.

copy.deepcopy(): This function performs a deep copy of an object. It creates a completely independent copy of the object and recursively copies all objects found within the original object. In the case of a list, a deep copy creates a new list object and also copies all the elements within the original list
changes made to the original list or its elements will not affect the copied list.

```
old_list = [1, [2, 3]]
new_list = copy.deepcopy(old_list)

old_list[0] = 100
old_list[1].append(4)

print(old_list)     # Output: [100, [2, 3, 4]]
print(new_list)     # Output: [1, [2, 3]].
```

In the example above, modifying the original list and its nested object (old_list[1]) does not affect the copied list because a deep copy creates independent copies of all objects.