1. What exactly is []?

Ans- It is an empty list. We can add & remove init.

2. 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- Link- https://colab.research.google.com/drive/1t-SA7YOtTmEtxLmjeA26-jFB7LdG7SFk?usp=sharing

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

spam[2]='hello'

spam

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

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

Spam[3]

‘d’

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

‘d’

5. 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.

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

Ans- 1

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

Ans- [3.14, 'cat’, 11, 'cat’, True,99]

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

Ans- [3.14, 11, 'cat', True]

9. What are the list concatenation and list replication operators?

Ans- the list concatenation operator and the list replication operator are used to manipulate and combine lists. Here's a description of each operator:

1. List Concatenation Operator (+): The list concatenation operator + allows you to concatenate or combine two or more lists into a single list.

```

list1 = [1, 2, 3]

list2 = [4, 5, 6]

concatenated\_list = list1 + list2

print(concatenated\_list) # Output: [1, 2, 3, 4, 5, 6]

```

B). List Replication Operator (\*): The list replication operator \* allows you to replicate or repeat a list a specified number of times. It creates a new list by repeating the elements of the operand list.

``` original\_list = [1, 2, 3]

replicated\_list = original\_list \* 3

print(replicated\_list) # Output: [1, 2, 3, 1, 2, 3, 1, 2, 3]

```

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

Ans- The append() and insert() methods are used to add elements to a list, but they differ in how they add the elements. Here's the difference between the two methods:

A). append() Method: The append() method is used to add an element to the end of a list. It takes a single argument, which is the element to be added, and appends it to the end of the list.

``

my\_list = [1, 2, 3]

my\_list.append(4)

print(my\_list) # Output: [1, 2, 3, 4]

```

B). insert() Method: The insert() method is used to add an element at a specific position in a list. It takes two arguments: the index position where the element should be inserted and the element to be added.

```

my\_list = [1, 2, 3]

my\_list.insert(1, 4)

print(my\_list) # Output: [1, 4, 2, 3]

```

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

Ans- Two methods for removing items from a list: remove() and pop().

A). remove() Method: The remove() method is used to remove the first occurrence of a specific value from a list. It takes a single argument, which is the value to be removed. If the value is found in the list, it is removed; otherwise, a ValueError is raised.

```my\_list = [1, 2, 3, 2, 4]

my\_list.remove(2)

print(my\_list) # Output: [1, 3, 2, 4]

```

B). pop() Method: The pop() method is used to remove and return an element at a specific index position from a list. It takes an optional argument, which is the index of the element to be removed. If no argument is provided, pop() removes and returns the last element in the list.

```

my\_list = [1, 2, 3, 4]

popped\_element = my\_list.pop(1)

print(popped\_element) # Output: 2

print(my\_list) # Output: [1, 3, 4]

```

12. Describe how list values and string values are identical.

Ans- List values and string values are identical in terms of being ordered collections that support indexing, slicing, iteration, length calculation, and concatenation. Both can store a sequence of elements or characters. However, lists are mutable objects that allow for modification of their content, while strings are immutable and cannot be directly changed after creation. Despite these differences in mutability, the basic operations and behavior shared by both types make them similar in many ways.

13. What's the difference between tuples and lists?

Ans- Tuples are immutable and declare using () while lists are mutable & declare using [].

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

tuple\_Int=(42, )

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

Ans- To convert a list to a tuple in Python, you can use the tuple() function

my\_list = [1, 2, 3]

my\_tuple = tuple(my\_list)

print(my\_tuple) # Output: (1, 2, 3)

& to convert tuple to a list we can use list() function

my\_list=list(my\_tuple)

my\_list

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

Ans- Variables in Python that "contain" list values actually hold references to list-objects. The variables themselves do not directly store the list values, but rather point to the memory location where the list object resides. This means that multiple variables can refer to the same list object, enabling modifications to the list through any of those variables. Understanding this reference-based behavior is important when working with lists to avoid confusion, manage memory effectively, and correctly handle assignments, function calls, or modifications to list contents. By recognizing that variables contain references to list objects, you can manipulate and interact with lists in Python more accurately and efficiently.

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

Ans- In Python's copy module, both copy.copy() and copy.deepcopy() are used to create copies of objects. However, they differ in how they handle nested objects and references within the copied object. Here's how you can distinguish between them:

* copy.copy(): copy.copy() creates a shallow copy of an object. It copies the top-level object and its references, but not the nested objects. In other words, the new copied object will have references to the same nested objects as the original object. If any of the nested objects are modified, the changes will be reflected in both the original and copied objects.
* copy.deepcopy(): copy.deepcopy() creates a deep copy of an object. It recursively copies the top-level object and all the nested objects, creating completely independent copies. Any modifications made to the nested objects in the copied object will not affect the original object or any other copies.

import copy

original\_list = [1, [2, 3]]

shallow\_copy = copy.copy(original\_list)

deep\_copy = copy.deepcopy(original\_list)

original\_list[0] = 10

original\_list[1][0] = 20

print(original\_list) # Output: [10, [20, 3]]

print(shallow\_copy) # Output: [10, [20, 3]]

print(deep\_copy) # Output: [1, [2, 3]]