1. What exactly is []?

Ans: Empty list

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:

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

spam[2]='hello'

print(spam)

**Output:**

[2, 4, 'hello', 8, 10]

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)]?

Ans : d

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

Ans : 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')?

bacon= [3.14, 'cat', 11, 'cat' ,True]

print(bacon.index('cat'))

Ans: 1

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

Ans :

bacon= [3.14, 'cat', 11, 'cat' ,True]

bacon.append(99)

print(bacon)

Output :

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

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

Ans:

bacon= [3.14, 'cat', 11, 'cat' ,True]

bacon.remove('cat')

print(bacon)

Output : [3.14, 11, 'cat', True]

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

Ans :

**list concatenation**: Adds elements of one list to the another list at end.

**Example :**

bacon= [3.14, 'cat', 11, 'cat' ,True]

spam=['India','Pakistan','Australia','England',4]

bacon\_spam=bacon+spam

print(bacon)

print(spam)

print(bacon\_spam)

**Output :**

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

['India', 'Pakistan', 'Australia', 'England', 4]

[3.14, 'cat', 11, 'cat', True, 'India', 'Pakistan', 'Australia', 'England', 4]

**list replication :** List copy using =(assignment operator) This is the simplest method of cloning a list by using = operators. This operator assigns the old list to the new list using Python = operators. Here we will create a list and then we will copy the old list into the new list using assignment operators

**Example :**

bacon= [3.14, 'cat', 11, 'cat' ,True]

spam=bacon

print(bacon)

print(spam)

**Output :**

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

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

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

Ans :

**Example :**

The append() method appends an element to the end of the list.

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

**my\_list.append(5)**

**print(my\_list)**

**Output :**

[1, 2, 3, 4, 5]

**Example :**

The insert() methods inserted an element into the list at a particular index.

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

my\_list.insert(2, 6)

print(my\_list)

**Output :**

[1, 2, 6, 3, 4]

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

my\_list = [10, 20, 30, 40, 50]

index = 2

removed\_item = my\_list.pop(index)

print(f"Removed item: {removed\_item}")

print(f"Updated list: {my\_list}")

the pop() function is used to remove the item at index 2 from the list my\_list. The removed item is stored in the removed\_item variable, and the updated list is printed.

Output :

Removed item: 30

Updated list: [10, 20, 40, 50]

my\_list = [10, 20, 30, 20, 40]

value = 20

my\_list.remove(value)

print(f"Updated list: {my\_list}")

the remove() function is used to remove the first occurrence of the value 20 from the list my\_list. The function searches for the value in the list and removes it. The resulting list is then printed.

Output :

Updated list: [10, 30, 20, 40]

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

List values and string values share some similarities, but they are fundamentally different data types. Here's a comparison between list values and string values:

* Sequential Data: Both lists and strings are sequential data structures, meaning they store elements in a specific order. In a list, the elements can be of any data type and can be accessed and modified individually using indexing. Similarly, in a string, the elements are characters, and each character has a specific position (index) within the string.
* Indexing and Slicing: Both lists and strings allow indexing and slicing operations. You can access individual elements of a list or a string by their index, starting from 0. Additionally, you can extract a subset of elements from both data types using slicing, which involves specifying a range of indices.
* Iteration: Lists and strings can be iterated over using loops or other iteration mechanisms. You can access each element in a list or a string one by one, performing operations or extracting information as needed.
* Concatenation: Both lists and strings support concatenation, which means combining two or more values together. In a list, concatenation involves using the + operator to combine two lists into a single list. In a string, concatenation involves using the + operator to join two strings together.

However, there are important differences between lists and strings:

* Mutability: Lists are mutable, meaning you can change their elements after they are created. You can add or remove elements, modify existing elements, or rearrange the order of elements in a list. On the other hand, strings are immutable, which means once a string is created, you cannot change its individual characters. Instead, you need to create a new string if modifications are needed.
* Type of Elements: Lists can store elements of different data types, such as numbers, strings, booleans, or even other lists. In contrast, strings store sequences of characters, and each element within a string is a character.
* Specific Operations: Lists and strings have specific operations that are unique to their respective data types. For example, lists have methods like append(), extend(), and insert() for adding elements, while strings have methods like split(), join(), and replace() for string manipulation.

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

* Tuples and lists are both common data structures used in programming, but they have some important differences:
* **Mutability:** The main difference between tuples and lists lies in their mutability. Lists are mutable, meaning you can modify, add, or remove elements after the list is created. On the other hand, tuples are immutable, which means once a tuple is created, you cannot modify its elements. In other words, tuples are read-only and cannot be changed.
* **Syntax:** Tuples are typically defined using parentheses () or without any delimiters, separating elements with commas. For example: (1, 2, 3) or 1, 2, 3. Lists, on the other hand, are defined using square brackets []. For example: [1, 2, 3].
* **Usage and Purpose**: Lists are commonly used for storing collections of similar or related items, where the order and mutability of elements are important. They provide various methods for adding, removing, and modifying elements, making them suitable for dynamic data. Tuples, on the other hand, are often used to represent a collection of different values that are related but don't need to be modified. Tuples are useful in cases where immutability and the order of elements are desired properties.
* **Performance:** Due to their immutability, tuples are generally more memory efficient and faster to process compared to lists. Since tuples cannot be modified, the interpreter can optimize their storage and operations. Lists, being mutable, require extra memory and processing time to accommodate potential modifications.
* **Common Operations:** Both tuples and lists support common operations like indexing, slicing, and iteration. You can access individual elements of a tuple or a list using their index, and you can extract a subset of elements using slicing. Iteration over tuples and lists is done in a similar manner.

my\_tuple = (1, 2, 3)

my\_list = [1, 2, 3]

my\_tuple[0] = 4

my\_list[0] = 4

my\_list.append(4)

# my\_tuple.append(4)

print(my\_tuple)

print(my\_list)

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

To create a tuple value that only contains the integer 42, you can use parentheses () with a comma after the integer:

my\_tuple = (42,)

The trailing comma is important to distinguish it as a tuple instead of just a value in parentheses. Without the comma, it would be interpreted as an integer in parentheses rather than a tuple.

Alternatively, you can also create the tuple without using parentheses:

my\_tuple = 42,

In this case, the comma at the end indicates that it is a tuple with a single element.

Both of these methods will create a tuple with a single element, which is the integer value 42.

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

To convert a list value into its tuple form, you can use the tuple() function in most programming languages. Here's an example in Python:

my\_list = [1, 2, 3]

my\_tuple = tuple(my\_list)

print(my\_tuple)

Output:

(1, 2, 3)

In this example, the tuple() function is used to convert the my\_list list into a tuple my\_tuple. The tuple() function takes an iterable (such as a list) as an argument and returns a tuple containing the same elements in the same order.

Conversely, to convert a tuple value into its list form, you can use the list() function. Here's an example:

my\_tuple = (1, 2, 3)

my\_list = list(my\_tuple)

print(my\_list)

Output:

[1, 2, 3]

In this case, the list() function is used to convert the my\_tuple tuple into a list my\_list. The list() function takes an iterable (such as a tuple) as an argument and returns a list containing the same elements in the same order.

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

You are correct. In many programming languages, variables that "contain" list values do not actually store the list itself, but rather a reference or pointer to the list in memory. This concept is known as reference semantics or reference types.

When you assign a list to a variable, the variable does not directly store the list elements. Instead, it holds a reference to the memory location where the list is stored. In other words, the variable contains the memory address of the list.

This means that when you perform operations or modifications on the list through the variable, you are actually manipulating the list itself and not a separate copy. Multiple variables can refer to the same list, so changes made to the list through one variable will be reflected in other variables that refer to the same list.

Here's an example in Python to illustrate this behavior:

my\_list = [1, 2, 3]

another\_list = my\_list

another\_list.append(4)

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

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

In this example, both my\_list and another\_list refer to the same list in memory. When we append the value 4 to another\_list, the change is reflected in my\_list as well. This is because both variables are referencing the same underlying list.

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

The copy module in Python provides two methods, copy() and deepcopy(), which are used to create copies of objects. Here's how you can distinguish between them:

copy.copy():

copy.copy() creates a shallow copy of an object. It creates a new object with a new reference, but the individual elements within the object are still references to the same objects in memory.

If the original object contains mutable elements (e.g., lists, dictionaries), changes made to those mutable elements in either the original object or the copied object will be reflected in both.

However, if the original object contains immutable elements (e.g., integers, strings), changes made to those elements will not affect the other object.

Shallow copy creates a new object, but it may share references to objects contained within the original object.

Shallow copy is suitable when you want to create a new object with separate references to the elements but don't need a complete independent copy of the nested objects.

copy.deepcopy():

copy.deepcopy() creates a deep copy of an object. It creates a completely independent copy of the object and all its nested objects.

It recursively copies every element within the object, creating new objects with new references.

Modifying the elements within the original object or the copied object will not affect each other since they reference completely different objects.

Deep copy is useful when you want to create a new object that is completely independent of the original, including all the nested objects.

Here's an example to illustrate the difference:

import copy

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

# Shallow copy

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

# Deep copy

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

# Modifying the elements

original\_list[1].append(5)

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

print(shallow\_copy) # Output: [1, [2, 3, 5], 4]

print(deep\_copy) # Output: [1, [2, 3], 4]

In this example, modifying the nested list within the original\_list affects both the original\_list and the shallow copy. However, the deep copy remains unaffected, as it is completely independent.