1. What exactly is []?

Ans. In Python, [] represents an empty list. Lists are ordered collections of items, and when there are no items in the list, it's represented as an empty pair of square brackets.

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. In Python, indexing starts at 0, so the third element has an index of 2.

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')?

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, 99]

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

Ans. In Python, the list concatenation operator is +, and the list replication operator is \*.

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

Ans. The append() and insert() methods are both used to add elements to a list in Python, but they differ in how they add elements and where they add them.

The append() method adds a single element to the end of the list.

The insert() method inserts a single element into the list at a specified index.

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

Ans. remove(): The remove() method removes the first occurrence of a specified value from the list.

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

my\_list.remove(2)

print(my\_list)

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

pop(): The pop() method removes the item at a specified index from the list and returns the removed item.

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

removed\_item = my\_list.pop(2)

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

print(removed\_item)

# Output: 3

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

Ans. List values and string values share several characteristics, making them similar or identical in many respects:

Ordered Collection: Both lists and strings are ordered collections of elements. The elements in both data types maintain a specific order, meaning you can access individual elements by their index.

Indexing and Slicing: Both lists and strings support indexing and slicing operations. You can access individual elements or subsequences of elements using square brackets and index/slice notation.

Iteration: We can iterate over both lists and strings using loops (e.g., for loops). This allows you to access each element or character in the sequence sequentially.

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

Ans. Tuples and lists are both collection data types in Python, but they have several differences:

Mutability: Lists are mutable, meaning you can modify their elements after they are created. You can add, remove, or change items in a list.

Tuples are immutable, meaning once they are created, you cannot modify their elements. You cannot add, remove, or change items in a tuple.

Syntax: Lists are enclosed in square brackets [ ], and elements are separated by commas.

Tuples are enclosed in parentheses ( ), and elements are separated by commas.

Performance: Since tuples are immutable, they are generally faster than lists when it comes to accessing elements.

Lists, being mutable, may incur more overhead due to the need to resize or reallocate memory when elements are added or removed.

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

Ans. my\_tuple = (42,)

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

Ans. Ex.1

my\_list = [1, 2, 3, 4, 5]

my\_tuple = tuple(my\_list)

print(my\_tuple)

# Output: (1, 2, 3, 4, 5)

Ex.2

my\_tuple = (1, 2, 3, 4, 5)

my\_list = list(my\_tuple)

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

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

Ans. In Python, everything is an object, including lists. When you create a list and assign it to a variable, the variable does not directly hold the list's elements. Instead, it holds a reference to the memory location where the list object is stored. This reference allows you to access and manipulate the list's elements indirectly through the variable.

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

Ans. copy.copy():   
copy.copy() performs a shallow copy of an object. It creates a new object and inserts references to the objects found in the original.

If the object being copied contains nested objects (e.g., lists or dictionaries), it copies the references to those nested objects, rather than creating new copies of them.

Changes made to mutable objects within the original object will be reflected in the copied object and vice versa.

Shallow copies are faster and use less memory compared to deep copies, but they may not fully isolate the original and copied objects.

copy.deepcopy():  
copy.deepcopy() performs a deep copy of an object. It creates a completely new object and recursively copies all the objects found in the original, including any nested objects.

If the object being copied contains nested objects, deepcopy() creates new copies of those nested objects as well, ensuring that the original and copied objects are fully independent.

Changes made to mutable objects within the original object will not affect the copied object, and vice versa.

Deep copies are slower and may use more memory compared to shallow copies, especially for complex or deeply nested objects, but they provide complete isolation between the original and copied objects.