1. What exactly is []?

[] represents empty list in python.

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.)

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

spam [2] = ‘hello’

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

spam = ['a', 'b', 'c', 'd']

spam[1] = ‘e’

spam[2] = ‘x’

spam[3] = ‘y’

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

y

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

y

5. What is the value of spam[:2]?

['a', 'e']

Let's pretend bacon has the list [3.14, 'cat,' 11, 'cat,' True] for the next three questions.

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

bacon[1] = 99

bacon.append(12.2)

bacon.append(45)

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

3

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

[3.14, 99, 11, 'cat', True, 12.2, 45, 99]

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

[3.14, 99, 11, True, 12.2, 45, 99]

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

List concatenation (+) is used to add two or more list in a single list by using “+” operator. It creates a number a new list by adding elements in two lists.

Eg:

list1 = [1, 2, 3]

list2 = [4, 5, 6]

concatenated\_list = list1 + list2

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

List replication is used to replicate or repear certain number of times. It creates a number a new list by repeating elements.

Eg:

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()?

append() is used to add element at end of the list by taking single argument.

Eg:

my\_list = [1,2,3]

my\_list.append(4)

print(my\_list)

insert() is used to add specific position in list. It takes two arguments namely the index and element.

Eg:

my\_list = [1,2,3]

my\_list.insert(0,0)

print(my\_list)

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

remove() is used to remove the first occurrence of a specified element in a list.

Eg:

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

my\_list.remove(2)

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

pop() is used to remove element at a specified index in a list.

Eg:

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

my\_list.pop(2)

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

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

List values and string values in Python have some similarities and identical characteristics, but they also have distinct differences. Here's how they are identical:

* Sequence Type: Both lists and strings are considered sequence types in Python. They are ordered collections of elements where each element can be accessed using indexing.
* Indexing: Both lists and strings support indexing, which allows accessing individual elements based on their position within the sequence. Indexing starts from 0, where the first element has an index of 0, the second element has an index of 1, and so on.
* Slicing: Lists and strings both support slicing operations, which enable extracting sub-sequences or portions of the original sequence by specifying a range of indices. Slicing allows you to retrieve a portion of the sequence as a new sequence.
* Iteration: Both lists and strings can be iterated over using loops or iteration methods. You can iterate over each element in a list or string and perform operations on them.

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

Tuples and lists are both sequence data types in Python, but they have some important differences. Here are the main differences between tuples and lists:

1. Mutability: Tuples are immutable, which means their elements cannot be modified after creation. Once a tuple is defined, you cannot add, remove, or modify elements within it. In contrast, lists are mutable, allowing you to modify, add, or remove elements as needed.

2. Syntax: Tuples are defined using parentheses ( ) or without any enclosing symbols, separating elements with commas. For example: my\_tuple = (1, 2, 3) or my\_tuple = 1, 2, 3. Lists, on the other hand, are defined using square brackets [ ], enclosing elements with commas. For example: my\_list = [1, 2, 3].

3. Usage and Purpose: Tuples are often used to represent collections of related values where the order and values are important, such as coordinates, dates, or database records. They are commonly used for data that should not be modified. Lists, on the other hand, are used when you need a mutable sequence of elements, allowing you to add, remove, or modify elements as required.

4. Performance: Tuples are generally more memory efficient and faster to access compared to lists because of their immutability. However, this difference may not be noticeable for small sequences. If you have a large collection of data that doesn't need to be modified, using tuples can offer performance advantages.

5. Available Methods: Lists have a wide range of built-in methods, such as append(), insert(), remove(), sort(), and more, which allow for easy manipulation of the list. Tuples, being immutable, have a limited number of methods available. Some common tuple methods include count() and index().

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

Eg:

my\_tuple = (42,)

print(my\_tuple) #output: (42,)

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

my\_tuple = (42,)

my\_list = list(my\_tuple)

print(my\_list)

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

my\_tuple = tuple(my\_list)

print(my\_tuple)

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

Variables that "contain" list values in Python do not actually store the list itself. Instead, they store a reference or pointer to the list object in memory. In other words, the variable contains the memory address where the list object is stored.

In Python, variables are references to objects rather than the objects themselves. When you assign a list to a variable, the variable is bound to the memory address of the list object. This allows multiple variables to refer to the same list object, providing a way to access and manipulate the list.

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

**copy.copy():**

1. The copy.copy() function creates a shallow copy of an object.
2. It creates a new object, but the contents of the object are references to the original object's elements.
3. If the original object contains mutable objects (such as lists or dictionaries), the copied object will reference the same mutable objects as the original.
4. Changes made to the mutable objects in the copied object will affect the original object, and vice versa.
5. However, the outer structure of the copied object will be independent of the original object.
6. It is useful when you want to create a new object that shares the same references as the original object's elements.

Eg:

import copy

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

copied\_list = copy.copy(original\_list)

original\_list[0] = 10

original\_list[1].append(4)

print(original\_list) # [10, [2, 3, 4], 4]

print(copied\_list) # [1, [2, 3, 4], 4]

**copy.deepcopy():**

1. The copy.deepcopy() function creates a deep copy of an object.
2. It creates a new object and recursively copies all objects found within the original object, including nested objects and their contents.
3. The copied object and its nested objects are completely independent of the original object and its nested objects.
4. Changes made to the copied object or its nested objects will not affect the original object.
5. It is useful when you want to create a new object that is a completely independent copy of the original object.

Eg:

import copy

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

deepcopied\_list = copy.deepcopy(original\_list)

original\_list[0] = 10

original\_list[1].append(4)

print(original\_list) # [10, [2, 3, 4], 4]

print(deepcopied\_list) # [1, [2, 3], 4]