1. What exactly is []?

`[]` represents an empty list.

Lists are one of the built-in data types in Python and are used to store collections of items. They are denoted by square brackets `[ ]` and can contain elements of different data types, such as integers, strings, or even other lists.

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.)

To assign the value 'hello' as the third value in a list stored in a variable called `spam`, you can use the indexing and assignment syntax in Python. Since Python uses zero-based indexing, the third value would correspond to index 2.

Here's how you can accomplish it:

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

spam[2] = 'hello'

print(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)]?

=> ‘d’

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

=> ‘d’

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

=> ['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')?

=> 1

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

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

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

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

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

The list concatenation operator is `+`, and the list replication operator is `\*`. Here's a brief explanation of each:

1. List Concatenation (`+`):

The `+` operator is used for concatenating two or more lists together, resulting in a new list that contains all the elements from the concatenated lists. It does not modify the original lists, but rather creates a new list. Here's an example:

list1 = [1, 2, 3]

list2 = [4, 5, 6]

concatenated\_list = list1 + list2

print(concatenated\_list)

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

2. List Replication (`\*`):

The `\*` operator is used to replicate a list by a given number of times. It creates a new list by repeating the elements of the original list. Here's an example:

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()?

The append() and insert() methods in Python are used to add elements to a list, but they differ in how and where the elements are added:

1. The append() method is used to add an element to the end of a list. It modifies the original list by adding the element as the last item.

2. The insert() method is used to add an element at a specific position in a list. It allows you to specify both the index and the element you want to insert. It modifies the original list by shifting the existing elements to accommodate the new element

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

There are two common methods for removing items from a list:

1. `remove()` method:

The `remove()` method is used to remove the first occurrence of a specified value from a list. It searches for the value in the list, removes it, and modifies the list by shifting the remaining elements to fill the gap.

2. `pop()` method:

The `pop()` method is used to remove an element from a specific index in a list and return its value. If no index is specified, it removes and returns the last element from the list. It modifies the list by removing the element and adjusting the indices of the remaining elements.

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

List values and string values in Python share some similarities, but they are fundamentally different in their nature and behaviour. However, there are a few similarities between them:

1. Sequence Type: Both lists and strings are sequence types in Python. They can store multiple elements or characters in an ordered manner, allowing indexing and slicing operations to access individual elements or subsequence’s.

2. Indexing and Slicing: Both lists and strings support indexing and slicing operations. You can use square brackets (`[]`) to access individual elements or a range of elements within a list or a string.

3. Iteration: Both lists and strings can be iterated over using loops. You can use constructs like `for` loops to iterate through the elements or characters of a list or a string.

4. Length: Both lists and strings have a length, which can be obtained using the `len()` function. It returns the number of elements in a list or the number of characters in a string.

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

Tuples and lists are both data structures in Python used to store collections of items, but they differ in several key aspects:

1. Mutability: Lists are mutable, meaning their elements can be modified, added, or removed after the list is created. On the other hand, tuples are immutable, and once a tuple is created, its elements cannot be changed. To modify a tuple, you need to create a new tuple with the desired elements.

2. Syntax: Lists are represented using square brackets `[ ]`, while tuples are represented using parentheses `( )`.

3. Operations: Lists offer a wide range of operations, such as appending, inserting, or removing elements, as well as sorting and reversing the order. Tuples, being immutable, have fewer operations available. However, both lists and tuples support common operations like indexing, slicing, and iterating over the elements.

4. Use Cases: Lists are commonly used when you need a mutable collection that can be modified throughout your program. They are suitable for situations where you may need to add, remove, or change elements dynamically. Tuples, with their immutability, are often used to represent a collection of related values that should not be modified, such as coordinates, database records, or configurations.

5. Performance: Tuples are generally more lightweight and faster to access compared to lists because they are immutable. Since tuples cannot be modified, Python can optimize their storage and access. Lists, being mutable, require additional memory and have more overhead due to the ability to change their size and content.

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 enclose the value within parentheses. Here's an example:

my\_tuple = (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. It takes an iterable, such as a list, and returns a tuple containing the elements of the iterable. Here's an example:

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

my\_tuple = tuple(my\_list)

print(my\_tuple)

Output: (1, 2, 3, 4)

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 contain the list itself, but rather a reference to the list. In Python, variables are essentially labels or names assigned to objects in memory. When a list is assigned to a variable, the variable holds a reference to the memory location where the list is stored.

In other words, the variable acts as a pointer or a reference to the list object. This means that variables storing list values contain the memory address of the list object, rather than directly containing the list data.

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

The `copy.copy()` and `copy.deepcopy()` functions are both part of the `copy` module in Python, and they are used to create copies of objects. However, they differ in how they create the copies, specifically when dealing with nested objects or objects containing references.

1. `copy.copy()`:

The `copy.copy()` function creates a shallow copy of an object. It creates a new object and populates it with references to the nested objects found in the original object. In other words, it copies the top-level object and references the same nested objects as the original. Shallow copying is suitable when the object being copied does not contain mutable objects or when you don't need independent copies of nested objects.For example:

import copy

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

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

copied\_list[0] = 100

copied\_list[1].append(4)

print(copied\_list) # Output: [100, [2, 3, 4]]

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

In the example above, after shallow copying the `original\_list` to `copied\_list`, modifying the nested list (`copied\_list[1]`) affects both the copied list and the original list because they refer to the same nested object.

2. `copy.deepcopy()`:

The `copy.deepcopy()` function creates a deep copy of an object. It recursively creates new objects and copies all the nested objects found in the original object, ensuring that the copied objects are independent and do not share references with the original.

Deep copying is suitable when you need fully independent copies of nested objects or when dealing with mutable objects that you want to modify without affecting the original.For example:

import copy

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

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

copied\_list[0] = 100

copied\_list[1].append(4)

print(copied\_list) # Output: [100, [2, 3, 4]]

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

In the example above, after deep copying the `original\_list` to `copied\_list`, modifying the nested list (`copied\_list[1]`) only affects the copied list, while the original list remains unchanged.

To summarize, `copy.copy()` creates a shallow copy, which shares references to nested objects, while `copy.deepcopy()` creates a deep copy, ensuring independent copies of all nested objects. The choice between them depends on the specific requirements of your program and the nature of the objects being copied.