1. **What exactly is []?**

In Python, `[]` represents an empty list. A list is a data structure that can hold an ordered collection of elements, which can be of any data type, including other lists. An empty list contains no elements.

**Example:**

my\_list = [] # Create an 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.)**

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

In Python, you can assign the value 'hello' :

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

spam[2] = 'hello'

For the next three queries, where the spam includes the list `['a', 'b', 'c', 'd']`

Query 1: How would you replace the value 'b' with 'z' in the list?

spam[1] = 'z'

Query 2: How would you add the value 'e' to the end of the list?

spam.append('e')

Query 3: How would you remove the value 'c' from the list?

spam.remove('c')

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

1. `'3' \* 2` results in the string `'33'`.

2. `int('33')` converts the string `'33'` to an integer `33`.

3. `int('3' \* 2) / 11` divides `33` by `11`

So, `spam[int(int('3' \* 2) / 11)]` is equivalent to `spam[3]`.

Given that `spam` is the list `['a', 'b', 'c', 'd']`, accessing index `3` will

return the value `'d'`.

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

The value of `spam[-1]` refers to the last element of the list stored in the variable `spam`.

If we assume that `spam` contains the list `['a', 'b', 'c', 'd']`, then the value of `spam[-1]` would be `'d'`, as `'d'` is the last element of the list.

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

spam[:2], it selects a subsequence of elements from the list starting at index 0 up to, but not including, the element at index 2.

If we assume that `spam` contains the list `['a', 'b', 'c', 'd']`, then `spam[:2]` would result in a new list containing the elements at indexes 0 and 1.

So, the value of `spam[:2]` would be `['a', 'b']`.

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

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

For the list `bacon = [3.14, 'cat', 11, 'cat', True]`, the value `'cat'` appears at index 1 and index 3. However, the `index()` method returns the index of the first occurrence, so `bacon.index('cat')` would return `1`.

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

The `append()` method in Python is used to add an element to the end of a list , it will add the value `99` to the end of the list.

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

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

The `remove()` method in Python is used to remove the first occurrence of a specified value from a list

[3.14, 11, 'cat', True]

1. **What are the list concatenation and list replication operators?**

**1. List Concatenation (+):** The `+` operator is used to concatenate (combine) two or more lists to create a new list. **Example:** list1 = [1, 2, 3]

list2 = [4, 5, 6]

concatenated\_list = list1 + list2 # Results in [1, 2, 3, 4, 5, 6]

**2. List Replication (\*):** The `\*` operator is used to replicate a list by repeating its elements a specified number of times.

**Example:**

original\_list = [1, 2, 3]

replicated\_list = original\_list \* 3 # Results in [1, 2, 3, 1, 2, 3, 1, 2, 3]

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

1**. append()` Method:**

- The `append()` method is used to add a single element to the end of a list.

- It takes one argument, which is the element you want to add.

**Example:**

my\_list = [1, 2, 3]

my\_list.append(4) # Adds 4 to the end of the list

**2. insert()` Method:**

- The `insert()` method is used to add a single element at a specific index in a list.

- It takes two arguments: the index where you want to insert the element and the element itself.

**Example:**

my\_list = [1, 2, 3]

my\_list.insert(1, 5) # Inserts 5 at index 1

1. **What are the two 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 the list.

- It takes one argument, which is the value you want to remove.

**Example:**

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

my\_list.remove(2) # Removes the first occurrence of 2

**2. pop()` Method:**

- The `pop()` method is used to remove an element from a specific index in the list.

- It can take an optional index argument (default is -1), indicating the index of the element to remove.

**Example:**

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

removed\_element = my\_list.pop(1) # Removes element at index 1 (2)

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

**1. Ordered Sequences**: Both lists and strings are ordered sequences of elements.

**2. Indexing:** Elements in both lists and strings can be accessed using index numbers. Indexing starts from 0 .

**3. Slicing:** Slicing is supported for both lists and strings. You can extract subsequences of elements

**4. Iteration:** Both lists and strings can be iterated over using loops (such as `for` loops) to access each individual element in sequence.

**5. Concatenation:** Both lists and strings can be concatenated using the `+` operator.

**6. Replication:** Both lists and strings support replication using the `\*` operator.

**7. Length:** The `len()` function can be used to determine the number of elements in both lists and strings.

**Mutability:** One of the main differences is that lists are mutable, meaning you can change, add, or remove elements from a list after it is created.

**Content Types:** Lists can contain elements of different types, such as integers, strings, floats, etc. Strings are specifically designed to hold sequences of characters.

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

**1. Mutability:**

- Lists are mutable, which means their elements can be modified, added, or removed after creation.

- Tuples are immutable, meaning their elements cannot be changed once they are assigned. Once a tuple is created, its elements remain fixed.

**2. Syntax:**

- Lists are defined using square brackets `[ ]`, and elements are separated by commas.

- Tuples are defined using parentheses `( )`, and elements are also separated by commas.

**3. Usage and Use Cases:**

- Lists are often used when you need a collection of items that might change over time. You can add, modify, or remove elements as needed.

- Tuples are used when you want to group related data together, especially if the data has a specific meaning or order. Tuples are commonly used for returning multiple values from functions, as dictionary keys (since they are hashable), and for situations where immutability is desired**.**

**4. Performance:**

- Tuples generally have a slightly smaller memory overhead compared to lists because of their immutability.

- Accessing elements in both lists and tuples is fast (O(1)) since they both use indexing.

**6. Iteration and Iterability:**

- Both tuples and lists can be iterated over using loops, like `for` loops.

- Since tuples are immutable, they can be used as dictionary keys or elements of sets, while lists cannot.

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

To create a tuple that contains only the integer `42`, you can use the following syntax in Python:

my\_tuple = (42,)

Notice the comma `,` after the `42`. It is necessary to distinguish between a single-element tuple and just an integer in parentheses

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

**1. Converting List to Tuple:**

To convert a list into a tuple, you can use the `tuple()` function. It takes the list as an argument and returns a tuple containing the same elements.

**Example:**

my\_list = [1, 2, 3]

my\_tuple = tuple(my\_list)

**2. Converting Tuple to List:**

To convert a tuple into a list, you can use the `list()` function. It takes the tuple as an argument and returns a list containing the same elements.

**Example:**

my\_tuple = (4, 5, 6)

my\_list = list(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 are not actually storing the list itself, but rather a reference to the list. This is a key concept in Python's memory model.

In other words, when you assign a list to a variable, the variable holds a reference to the memory location where the list data is stored. This reference allows you to access and manipulate the list's elements using the variable, but the variable is not the list itself.

original\_list = [1, 2, 3]

new\_list = original\_list

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

In Python, the `copy` module provides two functions for creating copies of objects: `copy()` and `deepcopy()`. Both functions are used to create copies of objects, but they handle nested objects (objects within objects) differently.

**1. copy.copy()` (Shallow Copy):**

- The `copy.copy()` function creates a shallow copy of an object. This means that the top-level object is duplicated, but objects within it are not recursively copied.

- Changes to nested objects (if any) within the copied object will affect the original object, and vice versa.

**2. copy.deepcopy()` (Deep Copy):**

- The `copy.deepcopy()` function creates a deep copy of an object. This means that both the top-level object and all the objects within it are recursively duplicated.

- Changes to nested objects within the copied object do not affect the original object, and vice versa.

import copy

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

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

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