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

"[]" in Python represents an empty list, which is a built-in data structure for storing an ordered collection of items, which can be of different types. Lists are mutable, meaning you can add, remove or change elements after its creation. Lists are defined using square brackets, with items separated by commas. For example:

empty\_list = []

list\_of\_numbers = [1, 2, 3, 4, 5]

list\_of\_strings = ['hello', 'world']

list\_of\_mixed\_types = [1, 'hello', 3.14]

**2. In a list of values stored in a variable called spam, how would you assign the value &#39;hello&#39; as the third value? (Assume [2, 4, 6, 8, 10] are in spam.)**

You can assign the value 'hello' as the third value in the list stored in the variable spam as follows:

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

spam[2] = 'hello'

print(spam)

Output:

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

In this code, the value of the third element in the list spam is changed from 6 to 'hello' by using indexing and assignment. The first item in a list has an index of 0, so spam[2] refers to the third item in the list.

**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)]?**

The value of spam[int(int('3' \* 2) / 11)] is 'd'.

Here's what happens step by step:

'3' \* 2 returns the string '33'.

int('33') returns the integer 33.

33 / 11 returns the floating-point number 3.0.

int(3.0) returns the integer 3.

spam[3] returns the value 'd'.

So, the final result of the expression spam[int(int('3' \* 2) / 11)] is 'd'.

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

Regarding spam[-1], it returns the last element of the list spam, which is 'd'. Negative indexing in Python starts from the end of the list, with -1 being the index of the last element.

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

Regarding spam[:2], it returns a new list that consists of the first two elements of the original list spam, which is ['a', 'b']. The colon operator : is used to slice a list, and spam[:2] returns all elements from the beginning of the list up to (but not including) the index 2.

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')?**

The value of bacon.index('cat') is 1

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

Regarding bacon.append(99), it adds the integer 99 to the end of the list bacon, making its value change from [3.14, 'cat,' 11, 'cat,' True] to [3.14, 'cat,' 11, 'cat,' True, 99]. The list.append(element) method is used to add an element to the end of a list.

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

Regarding bacon.remove('cat'), it removes the first occurrence of the string 'cat' from the list bacon, making its value change from [3.14, 'cat',' 11, 'cat', True] to [3.14, 11, 'cat', True]. The list.remove(element) method is used to remove the first occurrence of an element from a list.

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

The list concatenation operator in Python is + and the list replication operator is \*.

For example:

list1 = [1, 2, 3]

list2 = [4, 5, 6]

# list concatenation

list3 = list1 + list2

print(list3) # [1, 2, 3, 4, 5, 6]

# list replication

list4 = list1 \* 3

print(list4) # [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 are used to add elements to a list in Python.

The append() method adds an element to the end of the list:

list = [1, 2, 3]

list.append(4)

print(list) # [1, 2, 3, 4]

The insert() method, on the other hand, adds an element at a specified index:

list = [1, 2, 3]

list.insert(1, 4)

print(list) # [1, 4, 2, 3]

So, the main difference between the two is that append() adds an element to the end of the list, while insert() adds an element at a specified index.

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

In Python, there are two methods for removing items from a list: remove() and pop().

The remove() method removes the first occurrence of the specified value:

list = [1, 2, 3, 4, 5]

list.remove(3)

print(list) # [1, 2, 4, 5]

The pop() method removes the item at a specified index (or the last item if index is not specified), and returns the removed item:

list = [1, 2, 3, 4, 5]

item = list.pop(2)

print(list) # [1, 2, 4, 5]

print(item) # 3

So, the difference between the two is that remove() removes the first occurrence of the specified value, while pop() removes the item at a specified index and returns the removed item.

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

In Python, both lists and strings are sequences, which means that they are ordered collections of elements.

One of the similarities between lists and strings is that both support indexing and slicing, allowing you to access and manipulate specific elements or sub-sequences within the data structure.

In addition, both lists and strings are immutable in Python, meaning that once you have created them, you cannot change the values of their elements. To modify a list or a string, you need to create a new one.

So, although lists and strings are two different data structures, they share many similarities in terms of their functionality and behavior.

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

Tuples and lists are both used to store collections of items in Python. However, there are some key differences between them:

Mutability: Lists are mutable, meaning that you can change the elements they contain after they have been created. Tuples, on the other hand, are immutable, which means that their elements cannot be changed after they have been created.

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

Performance: Tuples are generally faster than lists when it comes to accessing and iterating over elements, since they have a more compact representation in memory.

Use case: Lists are more commonly used for general-purpose storage of items, where the order of the items is important and the elements may need to be changed. Tuples are typically used when you want to ensure that the elements are not changed, such as to define a point in 2D space or a date.

So, while both tuples and lists have their own strengths and weaknesses, the choice between them usually comes down to whether you need mutability or immutability in your data structure.

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

In Python, you can type a tuple that only contains the integer 42 using parentheses and a comma:

t = (42,)

Note the comma after the 42 - this is necessary to specify that you are creating a tuple with a single element, rather than just a value in parentheses. If you omit the comma, Python will treat the expression as just a regular value in parentheses, not a tuple:

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

In Python, you can convert a list to a tuple using the tuple() function:

list = [1, 2, 3, 4, 5]

tuple = tuple(list)

print(tuple) # (1, 2, 3, 4, 5)

And you can convert a tuple to a list using the list() function:

tuple = (1, 2, 3, 4, 5)

list = list(tuple)

print(list) # [1, 2, 3, 4, 5]

So, to summarize, you can convert a list to a tuple using tuple(list\_value), and you can convert a tuple to a list using list(tuple\_value).

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

In Python, variables that "contain" list values are not lists themselves, but instead contain references to the lists.

In other words, when you assign a list to a variable, the variable does not store the actual list, but a reference to the list. This means that if you assign a list to multiple variables, each of those variables will reference the same list in memory. Modifying the list through one of the variables will affect all of the others, since they all refer to the same list.

For example:

list\_a = [1, 2, 3, 4, 5]

list\_b = list\_a

list\_b.append(6)

print(list\_a) # [1, 2, 3, 4, 5, 6]

print(list\_b) # [1, 2, 3, 4, 5, 6]

In this example, both list\_a and list\_b are variables that contain references to the same list in memory, rather than the list itself. This means that modifying the list through one of the variables (e.g., by appending to list\_b) affects both variables, since they both refer to the same list.

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

In Python, the copy module provides two functions for making copies of lists or other objects: copy.copy() and copy.deepcopy().

copy.copy() is used to create a shallow copy of an object. A shallow copy of an object creates a new object that references the same objects in memory as the original, but with a different identity. For example, if you have a list of lists, a shallow copy of the outer list will create a new list, but the inner lists will still be references to the same objects.

import copy

original = [[1, 2, 3], [4, 5, 6]]

shallow\_copy = copy.copy(original)

print(original is shallow\_copy) # False

print(original[0] is shallow\_copy[0]) # True

copy.deepcopy() is used to create a deep copy of an object. A deep copy of an object creates a completely new and independent copy of the object, including all nested objects. For example, if you have a list of lists, a deep copy of the outer list will create a new list and new inner lists, with no references to the original objects.

import copy

original = [[1, 2, 3], [4, 5, 6]]

deep\_copy = copy.deepcopy(original)

print(original is deep\_copy) # False

print(original[0] is deep\_copy[0]) # False

So, in summary, use copy.copy() for a shallow copy and copy.deepcopy() for a deep copy, depending on whether you want a new reference to the same objects or a completely independent copy.