1. What exactly is []?

Ans:

[] is an empty list. It is a data structure that can hold an ordered collection of elements. Lists in Python are mutable, meaning that you can modify them by adding, removing, or changing elements.

Example:

empty\_list = []

print(empty\_list)

another\_empty\_list = list()

print(another\_empty\_list)

1. 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:

If the variable spam contains the list ['a', 'b', 'c', 'd'], we can assign the value 'hello' as the third value using the same indexing and assignment approach. Here's how you can do it:

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

spam[2] = 'hello'

print(spam) # Output: ['a', 'b', 'hello', 'd']

In this case, the third value in the list has an index of 2. By assigning the value 'hello' to spam[2], we are replacing the existing value at that index with 'hello'.

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

Ans:

The value of spam [int(int(‘3’\*2) /11] can be calculated as follows:

‘3’\*2 creates a string ‘33’.

33/11 performs integer division, resulting in 3.

Finally spam[3] accesses the element at index 3 in the list stored in the variable spam.

Therefore, the expression spam[int(int('3' \* 2) / 11)] evaluates to 'd'.

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

Ans:

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

Based on the previous information you provided, if spam contains the list ['a', 'b', 'hello', 'd'], then spam[-1] will evaluate to 'd'.

Negative indexing allows you to access elements in reverse order, where -1 represents the last element, -2 represents the second-to-last element, and so on. Therefore, spam[-1] corresponds to the last element in the list, which is 'd'.

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

Ans:

The value of spam[:2] represents a slice of the list stored in the variable spam that includes elements up to (but not including) index 2.

Based on the previous information you provided, if spam contains the list ['a', 'b', 'hello', 'd'], then spam[:2] will evaluate to ['a', 'b'].

When using slicing notation start:end, the elements from the start index (inclusive) up to the end index (exclusive) are included in the slice. In this case, :2 means to include elements from the beginning of the list up to (but not including) index 2.

Therefore, the value of spam[:2] is ['a', 'b'].

**Lets 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’)?

Ans:

The index() method returns the index of the first occurrence of the specified element in a list. In this case, it searches for the index of the string 'cat' within the list bacon.

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

index = bacon.index('cat')

print(index)

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

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

Ans:

The bacon.append(99) operation adds the value 99 to the end of the list stored in the variable bacon. It modifies the list in place. Here's how the list bacon would look after the append() operation:

Before: [3.14, 'cat', 11, 'cat', True]

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

The append() method is used to add an element to the end of a list. In this case, it appends the value 99 to the existing list, resulting in the updated list with the additional element at the end.

So, the list bacon will be modified to [3.14, 'cat', 11, 'cat', True, 99] after the bacon.append(99) operation.

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

Ans:

The bacon.remove('cat') operation removes the first occurrence of the string 'cat' from the list stored in the variable bacon. It modifies the list in place. Here's how the list bacon would look after the remove() operation:

Before: [3.14, 'cat', 11, 'cat', True]

After: [3.14, 11, 'cat', True]

The remove() method removes the first occurrence of a specified element from the list. In this case, it removes the first instance of the string 'cat' from the list bacon.

After the bacon.remove('cat') operation, the first occurrence of 'cat' is removed from the list, resulting in the updated list [3.14, 11, 'cat', True].

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

Ans:

i) List Concatenation Operator (+):

The list concatenation operator + is used to combine two or more lists into a single list. It creates a new list that contains all the elements from the original lists in the order they appear. The original lists remain unchanged. Here's an example:

list1 = [1, 2, 3]

list2 = [4, 5, 6]

concatenated\_list = list1 + list2

print(concatenated\_list)

ii) List Replication Operator (\*):

The list replication operator \* is used to create a new list by repeating the elements of an existing list a certain number of times. It creates a new list with multiple copies of the original list's elements. Here's an example:

original\_list = [1, 2, 3]

replicated\_list = original\_list \* 3

print(replicated\_list)

The elements of original\_list are replicated three times using the \* operator, resulting in the new list replicated\_list that contains three copies of the elements from original\_list.

Both the list concatenation operator + and the list replication operator \* can be useful for manipulating and creating new lists in Python.

1. 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 and where the elements are added:

i) append() Method:

Syntax: list.append(element)

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.

Example:

my\_list = [1, 2, 3]

my\_list.append(4)

print(my\_list)

In this example, the append() method is used to add the element 4 to the end of the list my\_list. The resulting list is [1, 2, 3, 4].

ii) insert() Method:

Syntax: list.insert(index, element)

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

It modifies the original list by shifting existing elements to the right to make space for the new element.

Example:

my\_list = [1, 2, 3]

my\_list.insert(1, 4)

print(my\_list)

In this example, the insert() method is used to add the element 4 at index 1 in the list my\_list. The existing elements are shifted to the right, resulting in the list [1, 4, 2, 3].

To summarize, the main difference between append() and insert() is that append() adds an element to the end of the list, while insert() inserts an element at a specific index, pushing existing elements to the right.

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

Ans:

i) remove() Method:

Syntax: list.remove(element)

The remove() method is used to remove the first occurrence of a specified element from a list.

It modifies the original list by removing the first occurrence of the specified element.

Example:

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

my\_list.remove(2)

print(my\_list)

In this example, the remove() method is used to remove the first occurrence of the element 2 from the list my\_list. The resulting list is [1, 3, 2, 4, 2, 5].

ii) pop() Method:

Syntax: list.pop(index)

The pop() method is used to remove and return an element from a list at the specified index.

It modifies the original list by removing the element at the specified index.

Example:

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

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

print(my\_list)

print(removed\_element)

In this example, the pop() method is used to remove the element at index 2 (which is 3) from the list my\_list and store it in the variable removed\_element. The resulting list is [1, 2, 4, 5], and the removed element is 3.

The remove() method removes the first occurrence of a specific element, while the pop() method removes an element at a specific index and returns its value. It's important to note that both methods modify the original list.

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

Ans:

i) Sequence Types:

Both lists and strings are sequence types in Python. They can contain multiple elements arranged in a specific order. This means that you can access individual elements by their index, iterate over them using loops, and perform various sequence operations.

ii) Indexing and Slicing:

Both lists and strings support indexing and slicing. You can access individual elements of a list or a string by their index, where the index starts from 0. Additionally, you can extract a portion of a list or a string using slicing, which allows you to create new lists or strings.

iii) Immutable vs. Mutable:

One significant difference between lists and strings is that strings are immutable, while lists are mutable. This means that once a string is created, you cannot modify its individual characters. Instead, you need to create a new string. On the other hand, lists can be modified by adding, removing, or changing elements directly.

iv) Homogeneous vs. Heterogeneous:

Lists can contain elements of different types, allowing for heterogeneity. For example, a list can contain integers, strings, floats, or even other lists. In contrast, strings are homogeneous and can only contain characters.

v) Methods and Operations:

Lists and strings have their own set of methods and operations specific to their type. For example, strings have methods like upper(), lower(), split(), and more, which are used for string manipulation. Lists have methods like append(), remove(), pop(), and more, which are used for list manipulation.

While lists and strings have some similarities, their differences in mutability and element types make them suitable for different purposes. Lists are often used when dealing with collections of elements, while strings are used to represent and manipulate textual data.

1. What’s the difference between tuples and lists?

Ans:

Tuples and lists are both fundamental data structures in Python, but they have some important differences in terms of their mutability, syntax, and intended usage. Here are the key differences between tuples and lists:

i) Mutability:

Tuples are immutable, meaning their elements cannot be modified once they are assigned. You cannot add, remove, or change elements in a tuple after it is created.

Lists are mutable, allowing for modifications to the elements. You can add, remove, or change elements in a list.

ii)Syntax:

Tuples are defined using parentheses (), or without any delimiters by default. For example: my\_tuple = (1, 2, 3) or my\_tuple = 1, 2, 3.

Lists are defined using square brackets []. For example: my\_list = [1, 2, 3].

iii)Usage and Intended Purpose:

Tuples are typically used to represent a collection of heterogeneous elements, where the order and structure of the elements matter. Tuples are useful when you want to create an immutable sequence of values, such as coordinates, database records, or function arguments.

Lists are commonly used to represent a collection of homogeneous or heterogeneous elements that can be modified. Lists are versatile and can be used for dynamic storage, sorting, appending, or removing elements as needed.

iv)Performance:

Tuples are generally more memory-efficient and have a slightly faster performance compared to lists, primarily due to their immutability.

Lists, being mutable, may require more memory and have a slightly slower performance when modifying elements or resizing the list.

v) Methods:

Lists have more built-in methods and operations compared to tuples. Some list methods include append(), pop(), insert(), and sort(), which allow for dynamic modification of the list.

Tuples have fewer methods since they are immutable. They primarily include methods such as count() and index(), which enable basic operations on the tuple.

In summary, tuples are immutable and often used for fixed collections of elements, while lists are mutable and suitable for dynamic storage and manipulation of elements. Understanding the differences between tuples and lists allows you to choose the appropriate data structure based on your specific requirements.

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

Ans:

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

my\_tuple = (42,)

The key is to include a comma , after the integer value, even if it's the only element in the tuple. This is necessary to differentiate it from just a parentheses pair () used for grouping or other purposes.

Without the comma, Python would interpret (42) as an integer in parentheses rather than a tuple. Adding the trailing comma explicitly indicates that you intend to create a tuple with a single element.

So, (42,) creates a tuple containing only the integer 42. The comma ensures that it is recognized as a tuple with a single element rather than a plain integer.

You can verify the tuple by printing it:

print(my\_tuple)

The output shows the tuple (42,) indicating that it contains the integer value 42.

1. How do you get a list value’s tuple form? How do you get a tuple value’s list form?

Ans:

To convert a list value to its tuple form, you can use the tuple() function. It takes an iterable, such as a list, and returns a tuple containing the same elements. Here's an example:

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

my\_tuple = tuple(my\_list)

print(my\_tuple)

In this example, the tuple() function is used to convert the list my\_list to a tuple. The resulting tuple is (1, 2, 3, 4).

To convert a tuple value to its list form, you can use the list() function. It takes an iterable, such as a tuple, and returns a list containing the same elements. Here's an example:

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

my\_list = list(my\_tuple)

print(my\_list)

In this example, the list() function is used to convert the tuple my\_tuple to a list. The resulting list is [1, 2, 3, 4].

Both tuple() and list() functions provide a convenient way to convert between the two data structures. They allow you to convert a list to a tuple and vice versa, enabling you to work with the appropriate data structure based on your specific needs.

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

Ans:

In Python, variables are essentially labels or names that refer to objects. When you assign a list value to a variable, the variable is bound to the memory address where the list object is stored. It means the variable holds a reference or pointer to the list object rather than the actual list data.

This distinction is important because it affects how variables behave when working with mutable objects like lists. Multiple variables can reference the same list object, and modifying the list through one variable will affect all other variables that refer to the same list.

Here's an example to illustrate this concept:

list1 = [1, 2, 3]

list2 = list1

# Modifying list2 affects list1 as well

list2.append(4)

print(list1)

In this example, list1 and list2 both refer to the same list object. Modifying the list through list2 (by appending 4) also affects list1. This is because both variables hold references to the same underlying list object.

Understanding this reference behavior is essential to work effectively with lists and other mutable objects in Python. It allows you to manipulate and share data between variables while being aware of the potential side effects caused by mutability.

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

Ans:

i) copy.copy():

The copy.copy() function creates a shallow copy of an object.

A shallow copy creates a new object but references the same elements as the original object.

If the object being copied contains references to mutable objects, both the original and copied object will refer to the same mutable objects. Modifying the mutable objects through one reference will affect both the original and copied object.

copy.copy() is suitable for creating independent copies when the object contains immutable elements or when you want the copied object to share mutable elements with the original object.

ii) copy.deepcopy():

The copy.deepcopy() function creates a deep copy of an object.

A deep copy creates a new object and recursively copies all elements, including any nested objects, creating new references for all objects.

If the object being copied contains references to mutable objects, the deep copy creates new copies of the mutable objects, resulting in independent objects.

copy.deepcopy() is suitable when you want to create a completely independent copy of an object, including all its nested objects.