# ASSIGNMENT 4

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

[ ] represents the empty list, ie list having no elements in it. Empty lists are often used as placeholders, add or remove elements from them as needed during the program's execution

Example

List_1=[ ]

**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.**

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

spam[int(int('3' * 2) / 11)]  is 'd'

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

spam[-1] is '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')?**

There is a small mistake in given question list(single quotes mistake) I have attached the both answers

bacon=[3.14, 'cat,' 11, 'cat,' True]  # syntax error

if

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

bacon.index('cat')

o/p it will return 1st index position of cat i.e. 1

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

There is a small mistake in given question list(single quotes mistake) I have attached the both answers

bacon=[3.14, 'cat,' 11, 'cat,' True]  # syntax error

if

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

bacon.append(99)

the bacon list changes to [3.14, 'cat', 11, 'cat', True, 99]

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

There is a small mistake in given question list(single quotes mistake) I have attached the both answers

bacon=[3.14, 'cat,' 11, 'cat,' True]  # syntax error

if

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

bacon.remove('cat')

the bacon list changes to [3.14,  11, 'cat', True, ]

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

In Python, the list concatenation operator is +, and the list replication operator is *

List Concatenation (+) Operator: The + operator is used to concatenate (combine) two or more lists to create a new list containing all the elements from the original lists .It doesn't modify the original lists instead, it creates a new list with the combined elements.

Example:

```
list1 = [1, 2, 3]
list2 = [4, 5, 6]
result = list1 + list2          # Concatenate lists
print(result)                   # Output: [1, 2, 3, 4, 5, 6]
```

List Replication (*) Operator : The * operator is used to replicate a list by repeating its elements a specified number of times to create a new list.It doesn't modify the original list; it generates a new list with repeated elements.

Example:

```
List_1 = [1, 2, 3]

list_2= list_1 * 3                  # Replicate the list three times

  print(list_2)                      # Output: [1, 2, 3, 1, 2, 3, 1, 2, 3]
```

These operators are handy for manipulating and creating lists in Python. Just remember that they create new lists, leaving the original lists unchanged.

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

The append() and insert() methods are both used to add elements to a list in Python, but they do so in slightly different ways. append() is used to add an element to the end of a list, while insert() is used to add an element at a specific position within the list.

**append() Method** - The append() method is used to add an element to the end of a list. It takes one argument, which is the element you want to add, and appends it to the list and It does not require specifying an index in which the element is always added at the end.

Example:

```
  my_list = [1, 2, 3]

  my_list.append(4)

  print(my_list)                    # Output: [1, 2, 3, 4]
```

**insert() Method**- The insert() method is used to add an element to a specific position (index) within a list.It takes two arguments: the first argument is the index where you want to insert the element, and the second argument is the element itself.The element is inserted at the specified index, pushing existing elements to the right.

Example:

```
my_list = [1, 2, 3]

my_list.insert(1, 4)                    # Insert 4 at index 1

 print(my_list)                         # Output: [1, 4, 2, 3]
```

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

In Python, there are two primary methods for removing items from a list. These two methods provide different ways to remove items from a list based on your requirements. remove() is used to remove by value, while pop() is used to remove by index and also returns the removed element.

**remove() Method**: The remove() method is used to remove the first occurrence of a specified element from a list. It takes one argument, which is the element you want to remove. If the element is found in the list, the remove() method removes it. If the element is not present, it raises a ValueError exception.

Example:

```
my_list = [1, 2, 3, 2, 4]
my_list.remove(2)                    # Remove the first occurrence of 2
print(my_list)                       # Output: [1, 3, 2, 4]
```

**pop() Method:** The pop() method is used to remove and return an element from a list at a specified index.It takes one optional argument, which is the index of the element to remove. If no index is specified, it removes and returns the last element from the list.If the specified index is out of bounds or not provided, it raises an IndexError exception.

Example:

```
my_list = [1, 2, 3, 4, 5]
popped_element = my_list.pop(2)      # Remove and return the element at index 2 (3)
print(popped_element)                # Output: 3
print(my_list)                       # Output: [1, 2, 4, 5]
```

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

List values and string values have some similarities and can be described as follows:

**Sequences**: Both lists and strings are sequences of elements. A string is a sequence of characters, while a list is a sequence of arbitrary values (which can include characters, numbers, objects, etc.).

**Indexing:** You can access individual elements of both lists and strings using indexing. In Python, indexing starts at 0 for the first element.

```
my_string = "Hello"
my_list = [1, 2, 3]
print(my_string[0])          # Access the first character of the string: 'H'
print(my_list[1])            # Access the second element of the list: 2
```

**Slicing:** You can use slicing to extract a portion of both lists and strings by specifying a range of indices.

```
my_string = "Python"
my_list = [1, 2, 3, 4, 5]
sub_string = my_string[1:4]          # Extract a substring: 'yth'
sub_list = my_list[2:4]              # Extract a sublist: [3, 4]
```

**Iteration:** You can iterate over the elements of both lists and strings using loops or other iteration techniques.

```
my_string = "Python"

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

for char in my_string:

    print(char)                          # Iterate and print each character

for num in my_list:

    print(num)                           # Iterate and print each element
```

**Length:** You can find the length (the number of elements or characters) of both lists and strings using the len() function.

```
my_string = "Hello, World!"

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

string_length = len(my_string)           # Length of the string: 13

list_length = len(my_list)               # Length of the list: 5
```

**Concatenation**: You can concatenate (combine) multiple strings or lists to create a new string or list.

```
string1 = "Hello, "

string2 = "World!"

concatenated_string = string1 + string2          # Concatenate strings

list1 = [1, 2, 3]

list2 = [4, 5, 6]

concatenated_list = list1 + list2                # Concatenate lists
```

While lists and strings share these similarities, it's important to note that they are distinct data types with their own set of methods and behaviours. Lists are mutable ( while strings are immutable .Therefore, while they have similarities, they also have important differences in terms of how can manipulate and work with them in Python.

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

Tuples and lists in Python are both used to store collections of items, but they have some key differences:

**Mutability:** Lists are mutable, which means can change, add, or remove elements after the list is created. we can modify individual elements, append new elements, or remove elements using methods

like append(), insert(), remove() Tuples, on the other hand, are immutable, meaning their elements cannot be changed after the tuple is created. Once we define a tuple, you cannot add, remove, or modify elements within it.

Example:

```
my_list = [1, 2, 3]

my_tuple = (1, 2, 3)

# Lists are mutable

my_list[0] = 4                    # Modifying an element

my_list.append(4)                 # Adding an element

my_list.remove(2)                 # Removing an element

# Tuples are immutable (the following will result in an error)

my_tuple[0] = 4                   # Attempting to modify an element will raise a TypeError
```

**Syntax:**Lists are defined using square brackets [ ], and their elements are separated by commas.Tuples are defined using parentheses ( ), and their elements are also separated by commas.

```
my_list = [1, 2, 3]

my_tuple = (1, 2, 3)
```

**Use Cases:**Lists are typically used when need a collection of items that may change over time, and want the flexibility to modify them.Tuples are used when you want to create a collection of items that should remain constant and not be modified during the program's execution. They are often used for items that logically belong together.

**Performance:**Lists can be slightly faster than tuples when it comes to iteration and element access due to their mutability. However, the difference in performance is often negligible for most applications.

**Size:** Because tuples are immutable, they can be more memory-efficient than lists when storing a large number of elements with the same data type.

In summary, the key differences between tuples and lists are mutability (lists are mutable, tuples are immutable), syntax (brackets vs. parentheses), and their intended use cases. we should choose between them based on whether you need a collection that can change (use a list) or one that should remain constant (use a tuple).

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

```
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 between a list and a tuple in Python, you can use the list() and tuple()

```
# Convert a list to a tuple

my_list = [1, 2, 3, 4]

my_tuple = tuple(my_list)

print(my_tuple)                    # Output: (5, 6, 7, 8)

# Convert a tuple to a list

my_tuple = (5, 6, 7, 8)

my_list = list(my_tuple)

print(my_list)                     # Output: [5, 6, 7, 8]
```

**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 lists themselves. Instead, they contain references or pointers to the lists stored in memory. In Python, many variables can refer to the same list, and modifying the list through one variable will affect all references to that list.

```
list1 = [1, 2, 3]

list2 = list1                      # Both list1 and list2 now refer to the same list

list2.append(4)                    # Modifying the list using list2

print(list1)                       # Output: [1, 2, 3, 4] (list1 is also affected)
```

In this example, list1 and list2 both refer to the same list in memory. When modify the list using list2, list1 is also affected because they both point to the same underlying list. So, variables that appear to "contain" list values are actually storing references to those lists, allowing you to work with the same list data through multiple variables. Understanding this behavior is crucial when working with mutable data structures like lists in Python.

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

The copy module provides two main functions for copying objects. copy.copy() and copy.deepcopy() These functions are used to create copies of objects, especially when dealing with complex data structures like nested lists or dictionaries. I the main difference between copy.copy() and copy.deepcopy() is how they handle nested objects. copy.copy() creates a shallow copy, which shares references to inner objects, while copy.deepcopy() creates a deep copy, which recursively duplicates all inner objects, ensuring independence between the original and copied structures. The choice between them depends on your specific requirements and the complexity of the data you're working with.

**copy.copy() (Shallow Copy)** :copy.copy() creates a shallow copy of an object. In a shallow copy, the outer object is duplicated, but the inner objects (e.g., elements within a list or attributes within an object) are not duplicated. Changes made to the inner objects in the copied structure will affect the original structure and vice versa if the inner objects are mutable (e.g., lists).

```
import copy

original_list = [1, [2, 3], 4]

shallow_copy = copy.copy(original_list)

shallow_copy[1].append(42)          # Both original_list and shallow_copy share the same inner list

print(original_list)                                                    # Output: [1, [2, 3, 42], 4]
```

copy.deepcopy() (Deep Copy):copy.deepcopy() creates a deep copy of an object.In a deep copy, both the outer object and all inner objects are duplicated recursively. This means that changes made to the inner objects in the copied structure will not affect the original structure, even if the inner objects are mutable.

```
import copy

original_list = [1, [2, 3], 4]

deep_copy = copy.deepcopy(original_list)

                        # Changes to the inner list in deep_copy do not affect the original list

deep_copy[1].append(42)

print(original_list)                    # Output: [1, [2, 3], 4]
```