Name :- Deepawali . B. Mhaisagar
Assignment no 4

1. What exactly is []?
ANS
The symbol "[]" typically represents an empty list or an empty array in various programming languages and contexts. It is used to denote a collection or container that does not contain any elements or values.

In programming, a list or an array is a data structure that can hold multiple values of the same or different types. The square brackets "[]" are often used to create and access elements within a list or array. When the brackets are empty, it signifies that the list or array is empty.

For example, in Python, you can create an empty list using "[]":

```
my_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.)

ANS

To assign the value 'hello' as the third value in a list stored in a variable called `spam`, you can use the index notation and assign the value directly. In Python, list indices start from 0, so the third value corresponds to index 2.

```
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)]?

ANS d

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

ANS b

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

ANS

[“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’)?

ANS 1

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

ANS

The bacon.append(99) method call adds the value 99 to the end of the bacon list. After executing bacon.append(99), the list bacon will be modified to [3.14, 'cat', 11, 'cat', True, 99]. The append() method appends an element to the end of a list, expanding its size by one element.

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

ANS
The bacon.remove('cat') method call removes the first occurrence of the element 'cat' from the bacon list. After executing bacon.remove('cat'), the list bacon will be modified.

If we consider the original bacon list as [3.14, 'cat', 11, 'cat', True], the remove('cat') operation will remove the first occurrence of 'cat'. After removal, the modified bacon list will be [3.14, 11, 'cat', True]. The second occurrence of 'cat' remains unaffected.

ANS
In Python, the list concatenation operator is the plus sign (+), and the list replication operator is the asterisk (*). Here's how they work:

List Concatenation Operator (+): The plus sign (+) operator is used to concatenate, or combine, two or more lists into a single list. It creates a new list containing all the elements from the operand lists in the order they appear. Example:

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

list2 = [4, 5, 6]

concatenated_list = list1 + list2

print(concatenated_list)
```

O/P [1, 2, 3, 4, 5, 6]

List Replication Operator *():* *The asterisk ()* operator is used for list replication, which creates a new list by repeating the elements of a list a certain number of times.

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

replicated_list = list1 * 3

print(replicated_list)
```

O/P [1, 2, 3, 1, 2, 3, 1, 2, 3]

In the above example, the * operator replicates list1 three times, resulting in a new list replicated_list that contains the elements of list1 repeated three times.

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

ANS
The append() and insert() methods in Python are used to add elements to a list, but they differ in terms of where the new element is inserted and how it affects the existing elements in the list. Here's a breakdown of the differences between the two methods:

append(element):

- The append() method is used to add an element to the end of a list.
- It takes a single argument, which is the element to be added to the list.
- The append() method modifies the original list by adding the element at the end, without changing the position of existing elements.
- The append() method always adds the element at the end of the list, regardless of the list's current length.

example

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

my_list.append(4)

print(my_list)
```

O/P [1, 2, 3, 4]
insert(index, element):

- The insert() method is used to add an element at a specific position in a list.
- It takes two arguments: the index where the element should be inserted and the element itself.
- The insert() method modifies the original list by inserting the element at the specified index and shifting the existing elements to accommodate the new element.

- The insert() method allows you to control the position where the element is added by specifying the index.

```
my_list = [1, 2, 3]
my_list.insert(1, 5)
print(my_list)
```

O/P [1, 5, 2, 3]

11. What are the two methods for removing items from a list?
ANS
In Python, there are two commonly used methods for removing items from a list:

remove(element):

- The remove() method is used to remove the first occurrence of a specified element from a list.
- It takes a single argument, which is the element to be removed.
- The remove() method modifies the original list by removing the specified element if it exists. If the element is not found in the list, it raises a ValueError.
- The remove() method only removes the first occurrence of the element. If the element appears multiple times in the list, only the first occurrence will be removed.

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

my_list.remove(2)

print(my_list)
```

O/P [1, 3, 2, 4]

pop(index):

- The pop() method is used to remove an element from a list at a specified index.
- It takes an optional argument, which is the index of the element to be removed. If no index is specified, it defaults to the last element.
- The pop() method modifies the original list by removing the element at the specified index and returning its value.
- The pop() method allows you to access and remove an element at a specific index, reducing the list size by one.

```
my_list = [1, 2, 3, 4]
removed_element = my_list.pop(2)
print(my_list)
print(removed_element)
```

O/P [1, 2, 4]

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

ANS

While lists and strings are both types of sequences in Python, they have some similarities that make them similar in certain aspects. Here are the ways in which list values and string values are identical:

Sequential Data: Both lists and strings are sequential data structures, meaning they store elements or characters in a specific order.
Indexing: Both lists and strings allow accessing individual elements or characters using indexing. In Python, indexing starts from 0, so you can access elements by their position in the sequence.
Slicing: Lists and strings support slicing, which allows you to extract a portion of the sequence by specifying a range of indices.
Iteration: Both lists and strings can be iterated over using loops, such as for loops, to access each element or character in the sequence.
Length: You can determine the length of both lists and strings using the len() function, which returns the number of elements or characters in the sequence.

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

ANS

In Python, tuples and lists are both sequence data types, but they have some key differences in terms of their mutability, syntax, and usage. Here are the main differences between tuples and lists:

Mutability:

- Lists are mutable, meaning their elements can be modified, added, or removed after the list is created. You can change, add, or remove elements using various list methods.
- Tuples, on the other hand, are immutable, meaning their elements cannot be modified after the tuple is created. Once a tuple is defined, you cannot change its elements. Tuples do not have methods for modifying their elements.

Syntax:

- Lists are defined using square brackets ([]), and elements within the list are separated by commas. Lists can have zero or more elements.
- Tuples are defined using parentheses (()), and elements within the tuple are separated by commas. Tuples can have zero or more

elements. Although parentheses are optional when defining tuples, it's common to use them to distinguish tuples from other data types.

Usage and Purpose:

- Lists are commonly used when you need a mutable sequence to store and manipulate a collection of related elements. Lists are suitable for scenarios where you need to add, remove, or modify elements frequently.
- Tuples are often used when you want to group together related elements that should not be modified. Tuples are suitable for scenarios where you want to ensure the integrity and immutability of the data.

Performance:

- Tuples are generally more memory-efficient and provide slightly better performance compared to lists, especially when you have a large collection of data that does not need to be modified. This is because tuples are immutable and require less memory allocation and overhead.

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

ANS

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

my_tuple = (42,)

In Python, a comma (`,`) is required to distinguish a tuple with a single element from an integer enclosed in parentheses. Without the trailing comma, Python would interpret the expression `(42)` as just an integer value rather than a tuple.

By including the comma after the integer 42, you create a tuple with a single element. This allows you to differentiate between a tuple and a simple expression enclosed in parentheses.

15. 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 into its tuple form, you can use the `tuple()` function. This function takes an iterable (such as a list) as an argument and returns a tuple containing the elements of the iterable.

Here's an example:

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

my_tuple = tuple(my_list)

print(my_tuple)
```

O/P (1, 2, 3, 4, 5)

To convert a tuple value into its list form, you can use the `list()` function. This function takes an iterable (such as a tuple) as an argument and returns a list containing the elements of the iterable.

Here's an example:

```
my_tuple = (1, 2, 3, 4, 5)

my_list = list(my_tuple)

print(my_list)
```

O/P [1, 2, 3, 4, 5]

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

ANS

Variables that "contain" list values in Python are not actually lists themselves. Instead, they contain references to list objects. In Python, when you assign a list to a variable, the variable holds a reference to the memory location where the list object is stored. This behavior is a result of Python's object-oriented nature and memory management. It allows multiple variables to reference and interact with the same list object without duplicating the data.

Consider the following example:

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

other_list = my_list
```

In this example, my_list and other_list are two separate variables that reference the same list object [1, 2, 3]. They are not independent copies of the list; instead, they are different names for the same underlying list object.

If you modify the list through one variable, the change will be reflected when accessing the list through the other variable

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

ANS

The copy module in Python provides two functions for creating copies of objects: copy.copy() and copy.deepcopy(). These functions are used to create copies of objects, but they differ in terms of the level of copying performed. Here's how they can be distinguished:

copy.copy() (Shallow Copy):

copy.copy() performs a shallow copy of an object.

Shallow copy creates a new object and inserts references to the same elements as the original object.If the object being copied contains mutable elements (e.g., lists, dictionaries), the references to these elements will be shared between the original and copied object. Modifying the mutable elements through one object will affect both objects.However, the object itself is new, so changes to the copied object won't affect the original object.Shallow copy is a relatively faster operation compared to deep copy.copy.copy() is suitable when you want a new object with references to the same elements as the original object.

copy.deepcopy() (Deep Copy):

copy.deepcopy() performs a deep copy of an object.Deep copy creates a completely independent copy of the object and its nested elements All elements, including nested objects, are recursively copied, creating a new object with new references to all elements. Deep copy ensures that modifying elements in one object does not affect the other. Deep copy can be slower and consume more memory compared to shallow copy, especially for large or complex objects.copy.deepcopy() is suitable when you want a completely independent copy of the object, including nested elements.

Here's an example that demonstrates the difference between shallow copy and deep copy:

```
import copy

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

shallow_copy = copy.copy(original_list)

deep_copy = copy.deepcopy(original_list)



# Modifying the original list and shallow copy

original_list[0] = 99
```

```
original_list[1].append(100)



print(original_list)    # Output: [99, [2, 3, 100], 4]

print(shallow_copy)     # Output: [1, [2, 3, 100], 4]

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

In this example, modifying the original list affects both the shallow copy and the original list because the nested list is shared. However, the deep copy remains unaffected since it creates a completely independent copy of the original list.To summarize, copy.copy() performs a shallow copy, creating a new object with references to the same elements. copy.deepcopy() performs a deep copy, creating a new object with completely independent copies of all elements, including nested elements. The choice between them depends on whether you need a shallow copy with shared references or a deep copy with independent copies of all elements.