**ADITYA AMIN**

**ASSIGN : 13**

Q1. Can you create a programme or function that employs both positive and negative indexing? Is there any repercussion if you do so?

def get\_character(string, index):

if -len(string) <= index < len(string):

return string[index]

else:

return None

# Example usage

my\_string = "Hello, world!"

print(get\_character(my\_string, 0)) # Output: 'H'

print(get\_character(my\_string, -1)) # Output: '!'

print(get\_character(my\_string, 12)) # Output: 'r'

print(get\_character(my\_string, -14)) # Output: 'o'

print(get\_character(my\_string, 20)) # Output: None

Q2. What is the most effective way of starting with 1,000 elements in a Python list? Assume that all elements should be set to the same value.

If you want to start with a Python list containing 1,000 elements, all set to the same value, the most effective and efficient way is to use list multiplication. You can multiply a list with a single element by the desired number of repetitions to generate a list with the specified length.

my\_list = [0] \* 1000

Q3. How do you slice a list to get any other part while missing the rest? (For example, suppose you want to make a new list with the elements first, third, fifth, seventh, and so on.)

To slice a list and obtain specific elements while skipping the rest, you can utilize the extended slice syntax in Python. By specifying the appropriate step value, you can achieve the desired result. Here's an example:

my\_list = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

new\_list = my\_list[::2]

Q4. Explain the distinctions between indexing and slicing.

indexing allows you to access individual elements at specific positions in a sequence, while slicing enables you to extract a subset of elements by specifying a range of indices. Indexing provides access to a single element, whereas slicing returns a new sequence containing multiple elements within the specified range.

Q5. What happens if one of the slicing expression's indexes is out of range?

Start index out of range:

If the start index is greater than or equal to the length of the sequence, an empty sequence (string, list, etc.) is returned.

For example, my\_list[10:] where my\_list has a length of 5 would result in an empty list [].

Stop index out of range:

If the stop index is greater than the length of the sequence, the slice operation extends to the end of the sequence.

For example, my\_string[:100] where my\_string has a length of 12 would include all elements up to the end of the string without raising an error.

Negative indexes out of range:

If negative indexes are used and are out of range, they are adjusted to the valid range by adding the length of the sequence.

For example, my\_list[-10:-5] where my\_list has a length of 5 would be equivalent to my\_list[0:0] and result in an empty list [].

In all cases, Python ensures that the slicing operation proceeds without raising an error, even if the indexes provided are out of range. Instead of causing an exception, Python adjusts the indexes to fit within the valid range of indices and returns the resulting slice accordingly.

Q6. If you pass a list to a function, and if you want the function to be able to change the values of the list—so that the list is different after the function returns—what action should you avoid?

If you want a function to be able to change the values of a list passed as an argument, you should avoid reassigning a completely new object to the variable representing the list. In other words, you should avoid creating a new list object using assignment (=) inside the function.

Q7. What is the concept of an unbalanced matrix?

The concept of an unbalanced matrix typically refers to a matrix where the number of rows is not equal to the number of columns. In other words, an unbalanced matrix is a matrix that does not have an equal number of rows and columns.

Q8. Why is it necessary to use either list comprehension or a loop to create arbitrarily large matrices?

When creating arbitrarily large matrices, it becomes necessary to use either list comprehension or a loop due to the following reasons:

Memory Efficiency: Creating a large matrix requires allocating memory for all its elements. Using list comprehension or a loop allows you to generate and store the elements one by one or in small batches, avoiding the need to store the entire matrix in memory at once. This approach is more memory-efficient as it reduces the memory footprint during matrix creation.

Laziness: Generating the elements of a large matrix on-the-fly using list comprehension or a loop allows for a "lazy" evaluation. It means that the elements are computed or retrieved only when needed, rather than pre-computing and storing them all at once. This can be particularly beneficial when dealing with memory constraints or situations where not all elements of the matrix will be accessed or processed.

Computation Efficiency: Using list comprehension or a loop provides flexibility to apply computations or transformations on the fly while generating the matrix. It allows you to incorporate conditional statements, mathematical operations, or other logic within the comprehension or loop, providing efficient ways to manipulate the matrix elements during creation.

Scalability: List comprehension and loops provide a scalable approach to handle arbitrarily large matrices. By iterating or comprehending the elements in smaller chunks, you can generate matrices of any desired size without being limited by memory or computational constraints.