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

Answer :

def get\_substring(string, start\_index, end\_index):

substring = string[start\_index:end\_index]

return substring

# Example usage

string = "Hello, World!"

substring1 = get\_substring(string, 2, 9) # Positive indexing: "llo, Wo"

substring2 = get\_substring(string, -6, -1) # Negative indexing: "World"

print(substring1)

print(substring2)

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.

Answer : If you want to create a Python list with 1,000 elements, all set to the same value, one of the most effective ways is to use the list multiplication (\*) operator along with a single-element list.

value = 42 # The value you want to set for all elements

my\_list = [value] \* 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.)

Answer ;

To slice a list in Python and obtain specific elements while skipping the rest, you can use the slicing syntax with a step value. The step value determines the increment between the elements to be included in the slice.

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

new\_list = my\_list[::2] # Slicing with a step of 2

print(new\_list)

Q4. Explain the distinctions between indexing and slicing.

Answer :

Indexing and slicing are both operations used to access elements or subsequences from a sequence like a string or a list in Python, but they have distinct purposes and behaviors.

Indexing:

Indexing refers to accessing an individual element within a sequence by specifying its position or index.

Indexing in Python starts at 0, so the first element is at index 0, the second at index 1, and so on.

To perform indexing, you use square brackets ([]) with the index value inside them.

Indexing returns the specific element at the given index.

The result of indexing is a single element, not a subsequence.

Example: my\_list[2] would access and return the element at index 2 in my\_list.

Slicing:

Slicing allows you to extract a contiguous subsequence (a slice) from a sequence by specifying a range of indices.

Slicing is performed using the same square bracket notation ([]), but with start, stop, and optional step values separated by colons (:).

The start index is inclusive, the stop index is exclusive (i.e., up to, but not including, the stop index), and the step value determines the increment between indices.

Slicing returns a new sequence containing the specified subsequence.

The result of slicing is a subsequence, not a single element.

Example: my\_list[1:4] would extract and return a new list containing elements at indices 1, 2, and 3 from my\_list.

In summary, indexing retrieves a single element from a sequence based on its position, while slicing extracts a subsequence (multiple elements) from a sequence by specifying a range of indices. Indexing returns a single element, and slicing returns a new sequence containing the specified subsequence.

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

Answer :

If one of the slicing expression's indexes is out of range, Python handles it gracefully without raising an error. Instead, it adjusts the range to the valid indices of the sequence and returns the resulting slice.

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?

Answer :

If you pass a list to a function and you want the function to be able to change the values of the list so that the list is different after the function returns, you should avoid reassigning a new object to the list parameter itself. In other words, you should avoid modifying the reference of the list parameter.

Q7. What is the concept of an unbalanced matrix?

Answer :

In the context of matrices, an "unbalanced matrix" typically refers to a matrix where the number of rows is not equal to the number of columns. In other words, it is a matrix that does not have an equal number of elements in each row and column.

In a balanced or square matrix, the number of rows is equal to the number of columns, resulting in a symmetric shape. For example, a 3x3 matrix is a balanced matrix because it has 3 rows and 3 columns.

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

Answer :

When creating arbitrarily large matrices in Python, it is necessary to use either list comprehension or a loop because these constructs provide a way to generate the matrix elements dynamically based on a pattern or algorithm. Here are the reasons why list comprehension or a loop is necessary:

Memory Efficiency: Creating a large matrix directly using repeated concatenation or multiplication operators (\*) can result in excessive memory usage and poor performance. Each concatenation or multiplication operation creates a new copy of the matrix, leading to unnecessary memory consumption. In contrast, list comprehension or a loop allows you to generate the matrix elements incrementally without excessive memory overhead.

Flexibility: List comprehension and loops provide flexibility in generating matrix elements based on specific patterns, conditions, or algorithms. You can incorporate conditional statements, mathematical expressions, or other logic to control the creation of matrix elements. This flexibility is crucial when you need to generate matrices with specific patterns or properties.

Dynamic Size: List comprehension and loops enable you to dynamically control the size and shape of the matrix based on variables or input parameters. You can use variables or expressions to determine the dimensions of the matrix and generate the elements accordingly. This dynamic sizing is essential when dealing with matrices of varying sizes or when the size is determined at runtime.

Code Readability: List comprehension, when used appropriately, can provide a concise and readable way to create matrices. It allows you to express the matrix generation logic succinctly, making the code more readable and maintainable. Loops, although potentially more verbose, provide explicit control and can be more readable in certain cases.