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

A1.   
Yes, we can create a program or function that employs both positive and negative indexing. Positive indexing starts from 0 and moves left to right, while negative indexing starts from -1 and moves right to left. If we mix positive and negative indexing, we may not get the expected results, as the negative index will start counting from the end of the string, which may not be what we intended. It's generally better to stick to one type of indexing in a program or function to avoid confusion and unintended results.

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.

A2. The most effective way to create a list with 1,000 elements set to the same value is to use the \* operator to multiply a list containing the single value by 1,000:

my\_list = [initial\_value] \* 1000

Using the **\*** operator is more efficient than creating a loop to append elements to the list one at a time, because the multiplication is done in C and is faster than running a loop in Python.

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.)

A3. To get every other element of a list, starting from the first element, we can use slice notation with a step of 2:

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

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

print(new\_list) # [1, 3, 5, 7, 9]

This will slice the list starting from the first element (**my\_list[0]**), skipping one element after each slice until the end of the list is reached.

Q4. Explain the distinctions between indexing and slicing.

A4. In Python, both indexing and slicing are ways to access and retrieve elements from sequences like strings, lists, and tuples. However, there are some distinctions between indexing and slicing:

* Indexing retrieves a single element from a sequence, while slicing retrieves a contiguous subsequence (a slice) of elements from a sequence.
* Indexing is done using the syntax **sequence[index]**, where **index** is an integer that specifies the position of the element to retrieve, counting from 0. Slicing is done using the syntax **sequence[start:end:step]**, where **start** is an integer that specifies the starting position of the slice (inclusive), **end** is an integer that specifies the ending position of the slice (exclusive), and **step** is an optional integer that specifies the step size for the slice (default is 1).
* Indexing returns a single element of the same type as the sequence (e.g., a character for a string, an item for a list), while slicing returns a new sequence of the same type as the original sequence (e.g., a new string or list).
* Indexing raises an **IndexError** if the index is out of range (i.e., less than 0 or greater than or equal to the length of the sequence), while slicing returns an empty sequence if the slice is out of range.
* Indexing is often used to access individual elements of a sequence, while slicing is often used to extract subsequences or subarrays for further processing.

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

A5. If one of the slicing expression's indexes is out of range, Python will not raise an error; instead, it will simply use the nearest valid index value. For example, if the starting index is out of range, Python will start at the first index. Similarly, if the ending index is out of range, Python will end at the last index.

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?

A6. If we want a function to be able to change the values of a list that we pass to it, we should avoid creating a new list with the same name inside the function. This will create a local variable that shadows the global list, and any changes made to the local list will not affect the global list. Instead, we should modify the elements of the original list directly.

Q7. What is the concept of an unbalanced matrix?

A7. An unbalanced matrix is a two-dimensional data structure where each row can have a different number of elements. In other words, the number of columns in the matrix may vary across different rows. This is in contrast to a balanced matrix, where all rows have the same number of elements. Unbalanced matrices can be represented using nested lists in Python, where each inner list corresponds to a row of the matrix. The flexibility of unbalanced matrices can be useful in situations where the data being represented does not fit neatly into a uniform structure.

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

A8. In Python, lists are dynamic, which means that they can change in size during runtime. Therefore, using a loop or a list comprehension is necessary to create arbitrarily large matrices, as it allows for the dynamic allocation of memory for the matrix elements. If a fixed-size data structure such as an array is used, it would not be possible to create arbitrarily large matrices as the memory would be exhausted.