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

Ans: -

```
def index_example(lst):
    first_element = lst[0]  # Positive indexing
    last_element = lst[-1]  # Negative indexing
    return first_element, last_element
```

In this function, lst[0] and lst[-1] return the first and last elements of the list, respectively. There's no repercussion in using both positive and negative indexing in the same function

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.

Ans:-

```
lst = [0]*1000
```

This creates a list with 1,000 elements, all set to 0

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.)

Ans:-

```
lst = [1, 2, 3, 4, 5, 6, 7, 8, 9]
new_lst = lst[::2]  # This will give [1, 3, 5, 7, 9]
```

In this example, lst[::2] returns a new list with every other element from the original list

Q4. Explain the distinctions between indexing and slicing.

Ans:-
Indexing and slicing are two fundamental concepts in Python. They help you access specific elements in a sequence, such as a list, tuple, or string.

Indexing is the process of accessing an element in a sequence using its position in the sequence (its index). In Python, indexing starts from 0, which means the first element in a sequence is at position 0, the second element is at position 1, and so on.

Slicing is the process of accessing a sub-sequence of a sequence by specifying a starting and ending index. In Python, you perform slicing using the colon : operator

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

Ans: - In Python, if one of the slicing expression's indexes is out of range, it does not result in an error. Instead, it will return an empty sequence or a partial sequence, depending on the circumstances.

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?

Ans:- If you pass a list to a function in Python, and you want the function to be able to change the values of the list, you should avoid reassigning the list within the function. Here's an example:

```
def change_list(lst):
    lst[0] = "changed"  # This will change the original list

def wrong_change_list(lst):
    lst = ["changed"]  # This will NOT change the original list
```

In the change_list function, lst[0] = "changed" changes the first element of the original list. In the wrong_change_list function, lst = ["changed"] creates a new local list within the function and does not affect the original list

Q7. What is the concept of an unbalanced matrix?

Ans: - An unbalanced matrix is a term used in the context of power systems, where it refers to a situation where the magnitudes and phases of the voltage phasor components are different11. In another context, a matrix is considered unbalanced if not all cells in the matrix are balanced. A cell of the matrix is balanced if the number of cells in that matrix that are adjacent to that cell is strictly greater than the value written in this cell

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

Ans:- List comprehension or loops are necessary to create arbitrarily large matrices because they allow for the efficient generation of complex data structures. For example, you can create a 1000x1000 matrix filled with zeros using list comprehension as follows:

```
matrix = [[0 for _ in range(1000)] for _ in range(1000)]
```

This creates a list of lists (a matrix), where each inner list (a row of the matrix) is created by a list comprehension that generates 1000 zeros