Q1. Can you create a programme or function that employs both positive and negative indexing? Is

there any repercussion if you do so?

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.

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.)

Q4. Explain the distinctions between indexing and slicing.

Q5. What happens if one of the slicing expression&#39;s indexes is out of range?

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?

Q7. What is the concept of an unbalanced matrix?

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

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

Yes, you can create a program that uses both positive and negative indexing. Python allows you to mix positive and negative indexing to access elements from both ends of a sequence.

**Example**:

python

def mix\_indexing(sequence):

# Positive indexing

first\_element = sequence[0]

# Negative indexing

last\_element = sequence[-1]

# Mixing positive and negative indexing

mixed\_element = sequence[1:-1]

return first\_element, last\_element, mixed\_element

my\_list = [10, 20, 30, 40, 50]

print(mix\_indexing(my\_list)) # Outputs: (10, 50, [20, 30, 40])

**Repercussions**:

* **Readability**: Mixing positive and negative indexing can make code harder to read and understand, especially for those unfamiliar with the indexing rules.
* **Error-prone**: It can be error-prone if not handled carefully, particularly with complex sequences or when the sequence length changes.

### **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.**

The most effective way to create a list with 1,000 elements, all set to the same value, is to use list multiplication. This approach is concise and efficient.

**Example**:

python

my\_list = [0] \* 1000 # 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.)**

You can use slicing with a step parameter to achieve this. The step parameter allows you to specify which elements to include in the new list.

**Example**:

python

my\_list = [10, 20, 30, 40, 50, 60, 70, 80, 90, 100]

new\_list = my\_list[::2] # Starts from the beginning and takes every second element

print(new\_list) # Outputs: [10, 30, 50, 70, 90]

### **Q4. Explain the distinctions between indexing and slicing.**

**Indexing**:

* **Purpose**: Accesses a single element from a sequence (like a list or string) based on its position.
* **Syntax**: sequence[index]

**Example**:  
python  
  
my\_list = [1, 2, 3, 4, 5]

print(my\_list[2]) # Outputs: 3

**Slicing**:

* **Purpose**: Creates a new sequence containing a subset of elements from the original sequence.
* **Syntax**: sequence[start:end:step]

**Example**:  
python  
  
my\_list = [1, 2, 3, 4, 5]

print(my\_list[1:4]) # Outputs: [2, 3, 4]

**Key Differences**:

* **Indexing** retrieves a single item, while slicing creates a new list or string from a range of items.
* **Slicing** allows for more complex operations with start, end, and step, whereas indexing only retrieves a single item.

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

If an index in a slicing expression is out of range, Python handles it gracefully and does not raise an error. The out-of-range index is simply clamped to the valid range, and the slicing operation will adjust accordingly.

**Examples**:

python

my\_list = [1, 2, 3, 4, 5]

print(my\_list[1:10]) # Outputs: [2, 3, 4, 5] (end index 10 is beyond the list length)

print(my\_list[-10:2]) # Outputs: [1, 2] (start index -10 is clamped to the beginning)

### **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?**

To ensure that the function can modify the list and have those changes reflected outside the function, you should avoid creating a new list within the function and simply modifying the existing list. Avoid creating a new list by assignment, as it will not affect the original list.

**Example of Avoiding Problem**:

python

def modify\_list(lst):

lst[0] = 100 # Modifies the existing list

my\_list = [1, 2, 3]

modify\_list(my\_list)

print(my\_list) # Outputs: [100, 2, 3]

**Example of Problematic Action**:

python

def modify\_list(lst):

lst = [100, 200] # This creates a new list and does not modify the original list

my\_list = [1, 2, 3]

modify\_list(my\_list)

print(my\_list) # Outputs: [1, 2, 3] (no change)

### **Q7. What is the concept of an unbalanced matrix?**

An **unbalanced matrix** refers to a matrix where rows have different numbers of columns. This contrasts with a balanced or rectangular matrix, where every row has the same number of columns. Unbalanced matrices are often encountered in certain types of data processing or when dealing with ragged arrays.

**Example of Unbalanced Matrix**:

python

matrix = [

[1, 2, 3],

[4, 5],

[6, 7, 8, 9]

]

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

In Python, creating large matrices often requires generating elements in a structured manner, and using list comprehension or loops allows for dynamic creation and initialization of these matrices.

**List Comprehension**:

* **Usage**: Efficient and compact way to create matrices.

**Example**:  
python  
  
matrix = [[0 for \_ in range(1000)] for \_ in range(1000)] # Creates a 1000x1000 matrix with all elements set to 0

**Loops**:

* **Usage**: Provides more flexibility and control, especially for complex initialization logic.

**Example**:  
python  
  
matrix = []

for i in range(1000):

row = [0] \* 1000

matrix.append(row)

Both methods ensure that the matrix is created with the desired size and structure, making them essential for handling large matrices.