**FSDS MAY BATCH 2022(Python Assignment -13)**

**Submitted by: Shubham Tiwari**

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

Ans: Yes, it is possible to create a program or function that employs both positive and negative indexing. The repercussion would depend on the specific program or function being created and how it is being used.

For example, in Python, lists support both positive and negative indexing, so a program that uses a list with both positive and negative indexing would not have any negative repercussions as long as the program is written correctly.

However, if the program is using negative indexing in a way that is not intended or not well-defined, it could lead to unexpected behavior or errors. It is always important to thoroughly test and debug any program to ensure that it is working as intended.

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: There are several ways to create a Python list with 1,000 elements and set all of them to the same value.

One way is to use a list comprehension with the **\*** operator. This allows you to create a list by repeating a single value a certain number of times. Here's an example:

**my\_list = [0] \* 1000**

Another way is to use the **range()** function in combination with the **\*** operator. This allows you to create a list by specifying a range of integers and repeating a value a certain number of times. Here's an example:

**my\_list = [0 for \_ in range(1000)]**

A third way is using the **list()** function with the **\*** operator. This allows you to create a list by repeating an iterable a certain number of times. Here's an example:

**my\_list = list([0]) \* 1000**

It is worth noting that the latter two methods will create new copies of the values in the list, while the first one will simply repeat the same object references. If the values are simple immutable types like integers, the memory usage may be the same, but if the values are mutable types like lists, the memory usage will be different.

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: In Python, We can use list slicing to extract a portion of a list and create a new list with those elements.

To get every other element from a list, we can use slicing with a step of 2. The slice notation is **start:stop:step**, where **start** is the index of the first element to include, **stop** is the index of the first element to exclude, and **step** is the number of indices to skip between elements.

For example:

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

**new\_list = original\_list[0::2]**

**print(new\_list)**

This will return **[1, 3, 5, 7, 9]**.

We can also use list comprehension to achieve the same result. Here's an example:

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

**new\_list = [original\_list[i] for i in range(0, len(original\_list), 2)]**

**print(new\_list)**

This will also return **[1, 3, 5, 7, 9]**.

We can adjust the start and stop indexes to slice a different portion of the list.

It is worth noting that this approach does not change the original list, it creates a new one.

Q4. Explain the distinctions between indexing and slicing.

Ans: Indexing and slicing are two ways of accessing elements in a sequence (such as a list or a string) in Python.

Indexing is used to access a single element in a sequence by specifying its position or index. The index is an integer value that starts at 0 for the first element, 1 for the second element, and so on. For example, if you have a list **my\_list = [1, 2, 3, 4]**, you can access the second element (which is 2) by using **my\_list[1]**.

Slicing, on the other hand, is used to access a range of elements in a sequence. Instead of specifying a single index, a slice specifies a start index, an end index, and a step size. The start index is the first element of the slice (inclusive), the end index is the first element that is not in the slice (exclusive) and the step size is the number of elements to skip between elements in the slice. For example, if you have a list **my\_list = [1, 2, 3, 4, 5]**, you can extract a sub-list containing elements 2, 3, 4 by using **my\_list[1:4]**.

In summary, indexing is used to access a single element in a sequence, while slicing is used to access a range of elements.

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

Ans: If one of the slicing expression's indexes is out of range, an "IndexError" will be raised. For example, if a list has 5 elements and you try to access the element at index 5, an "IndexError" will be raised because the list only has 5 elements, the valid indexes for this list are 0,1,2,3,4.

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 we pass a list to a function and you want the function to be able to change the values of the list, you should avoid passing the list by value. Instead, the list should be passed by reference.

In Python, when a list is passed to a function, a reference to the list is passed, not a copy of the list. However, if we want to change the values of the list and make sure that the changes persist after the function returns, we should avoid reassigning the variable inside the function to a new list or a completely different object, this will make the reference to the original list invalid and the changes will not persist.

Q7. What is the concept of an unbalanced matrix?

Ans: When creating a matrix of a large size, it is necessary to use either list comprehension or a loop because creating the matrix manually element by element would be too time-consuming and not efficient.

List comprehension is a concise way to create a list by applying an expression to each element in a sequence. It can be used to create a matrix by applying the same expression to each element in a nested list. For example, you can create a matrix of zeroes with the same size as a given matrix by using list comprehension as follows:

**matrix\_size = (3, 4)**

**matrix = [[0 for \_ in range(matrix\_size[1])] for \_ in range(matrix\_size[0])]**

This will create a matrix with 3 rows and 4 columns filled with zeroes.

A loop can also be used to create a matrix by iterating over a range of values and adding the elements to the matrix. For example, you can create a matrix of ones with the same size as a given matrix by using a loop as follows:

**matrix\_size = (3, 4)**

**matrix = []**

**for i in range(matrix\_size[0]):**

**row = []**

**for j in range(matrix\_size[1]):**

**row.append(1)**

**matrix.append(row)**

This will create a matrix with 3 rows and 4 columns filled with ones.

Both list comprehension and loops allow you to create a matrix with a large size in a relatively short amount of time, but list comprehension is more concise and readable in this case.

In summary, using list comprehension or loops to create arbitrarily large matrices is more efficient than creating the matrix manually because it saves time and it's more readable.

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

Ans: It is necessary to use either list comprehension or a loop to create arbitrarily large matrices because creating a matrix with a fixed number of elements can be done using a simple list or array initialization, however, when the size of the matrix is not known ahead of time, a loop or list comprehension is needed to generate the necessary number of elements and organize them into the desired matrix structure.