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

Ans:-  In Python Positive indexing, we pass a **positive index that we want to access in square brackets**. The index number starts from 0 which denotes the first character of a string.

The negative index is **used to remove any new-line spaces from the string and allow the string to except the last character that is given as** S[:-1]. The negative index is also used to show the index to represent the string in correct order. **It supports negative indexing of arrays**

b = "Hello, World!"

print(b[-5:-2])

print(b[1:5])

output:-

orl

ello

no, there is no repercussion.

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:-

In python, at least 3 to 4 ways to create and initialize lists of a given size:

**Simple loop with append:**

my\_list = []

for i in range(1000):

my\_list.append(0)

**Simple loop with +=:**

my\_list = []

for i in range(1000):

my\_list += [0]

**List comprehension:**

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

**List and integer multiplication:**

my\_list = [0] \* 1000

The last method is the fastest by far. However, it should only be used with immutable items (such as integers). This is because it will create a list with references to the same item.

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:- You can generate a slice object **using the built-in function slice()** . If you want to repeatedly select the items at the same position, you only need to generate the slice object once. slice(start, stop, step) is equivalent to start:stop:step . If two arguments are specified, step is set to None .

color = ['Red', 'Green', 'White', 'Black', 'Pink', 'Yellow','mergenta','maroon','golden']

print(color[1:9:2])

output:-

[ 'Green', 'Black', 'Yellow',,'maroon']

Q4. Explain the distinctions between indexing and slicing.

Ans:-

. “Indexing” means referring to an element of an iterable by its position within the iterable. “Slicing” means **getting a subset of elements** from an iterable based on their indices.

Indexing: Indexing is used to obtain individual elements. Slicing: Slicing **is used to obtain a sequence of elements**. Indexing and Slicing can be be done in Python Sequences types like list, string, tuple, range objects.

Slicing in Python is **similar to indexing but returns a sequence of items instead of a single item**. The indices used for slicing are also zero-based. There are two variants of the slicing syntax: sequence[start:stop] and sequence[start:stop:step]. ... That's how you get a shallow copy of a sequence.

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

Ans:- It might be surprising at first, but it makes sense when you think about it. Indexing returns a single item, but slicing returns a subsequence of items. So when you try to index a nonexistent value, there's nothing to return. But when you slice a sequence outside of bounds, **you can still return an empty sequence**.

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:-Y ou **can send any data types of argument to a** function (string, number, list, dictionary etc.), and it will be treated as the same data type inside the function.

As usual, when we pass b to B() , the function has the local variable x refer to the same object as b . ... **Since no new object is created and the change occurred in place of the object**, when we print b, we get the modified list.

We’ve seen that what we do to a variable like number\_1 above inside a function doesn’t affect its global value. But number\_1 is an integer, which is a pretty basic data type. What happens if we try the same experiment with a different data type, like a list? Below, we’ll create a function called duplicate\_last() that will duplicate the final entry in any list we pass it as an argument.

initial\_list = [1, 2, 3]

def duplicate\_last(a\_list):

last\_element = a\_list[-1]

a\_list.append(last\_element)

return a\_list

new\_list = duplicate\_last(a\_list = initial\_list)

print(new\_list)

print(initial\_list)

[1, 2, 3, 3]

[1, 2, 3, 3]

As we can see, here the global value of initial\_list was updated, even though its value was only changed inside the function!

Q7. What is the concept of an unbalanced matrix?

Ans:- Whenever the cost matrix of an assignment problem is not a square matrix, that is, **whenever the number of sources is not equal to the number of destinations**, the assignment problem is called an unbalanced assignment problem. To make unbalanced assignment problem, a balanced one, a dummy facility(s) or a dummy job(s) (as the case may be) is introduced with zero cost or time.

When Hungarian method is applied to solve unbalanced assignment problem in which the numbers of jobs are more than the number of machines, **the procedure assigns some of the jobs to dummy machines which actually ignore the execution of those jobs**.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Job** | | | | |
| **Person** | **1** | **2** | **3** | **4** |
| **A** | 20 | 25 | 22 | 28 |
| **B** | 15 | 18 | 23 | 17 |
| **C** | 19 | 17 | 21 | 24 |

**Solution**

Since the number of persons is less than the number of jobs, we introduce a dummy person (D) with zero values. The revised assignment problem is given below:

Table

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Job** | | | | |
| **Person** | **1** | **2** | **3** | **4** |
| **A** | 20 | 25 | 22 | 28 |
| **B** | 15 | 18 | 23 | 17 |
| **C** | 19 | 17 | 21 | 24 |
| **D (dummy)** | 0 | 0 | 0 | 0 |

Ans. = 20 + 17 + 17 + 0 = 54.

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

Ans:- List comprehensions are often not only more readable but also faster than using “for loops.” They can simplify your code, but if **you put too much logic inside**, they will instead become harder to read and understand. ... In many cases, “for loops” will be your only choice.

>>> M1 = [[1, 2, 3],

[4, 5, 6],

[7, 8, 9]]

>>> M2 = [[9, 8, 7],

[6, 5, 4],

[3, 2, 1]]

We can index rows and columns within rows:

>>> M1[2]

[7, 8, 9]

>>> M1[2][2]

9

List comprehensions are powerful tools for processing such structures because they automatically scan rows and columns for us. For example, though this structure stores the matrix by rows, to collect the second column we can simply iterate across the rows and pull out the desired column. We can also iterate through positions in the rows and index as we go:

>>> [r[2] for r in M1]

[3, 6, 9]

Here, we pulled out column 3 from M1. We can get the same result from the following list comprehension.

>>> [M1[r][2] for r in (0, 1, 2)]

[3, 6, 9]