**DAY: 4 LAB PROGRAMS**

**1. Counting Elements**

**Given an integer array arr, count how many elements x there are, such that x + 1 is also in arr. If there are duplicates in arr, count them separately.**

**Example**

**Input: arr = [1,2,3]**

**Output: 2**

**Explanation: 1 and 2 are counted cause 2 and 3 are in arr.**

**Example 2:**

**Input: arr = [1,1,3,3,5,5,7,7]**

**Output: 0**

**Explanation: No numbers are counted, cause there is no 2, 4, 6, or 8 in arr.**

**Constraints:**

* **1 <= arr.length <= 1000**
* **0 <= arr[i] <= 1000**

def count\_elements(arr):

count = 0

num\_counts = {}

for num in arr:

num\_counts[num] = num\_counts.get(num, 0) + 1

for num in arr:

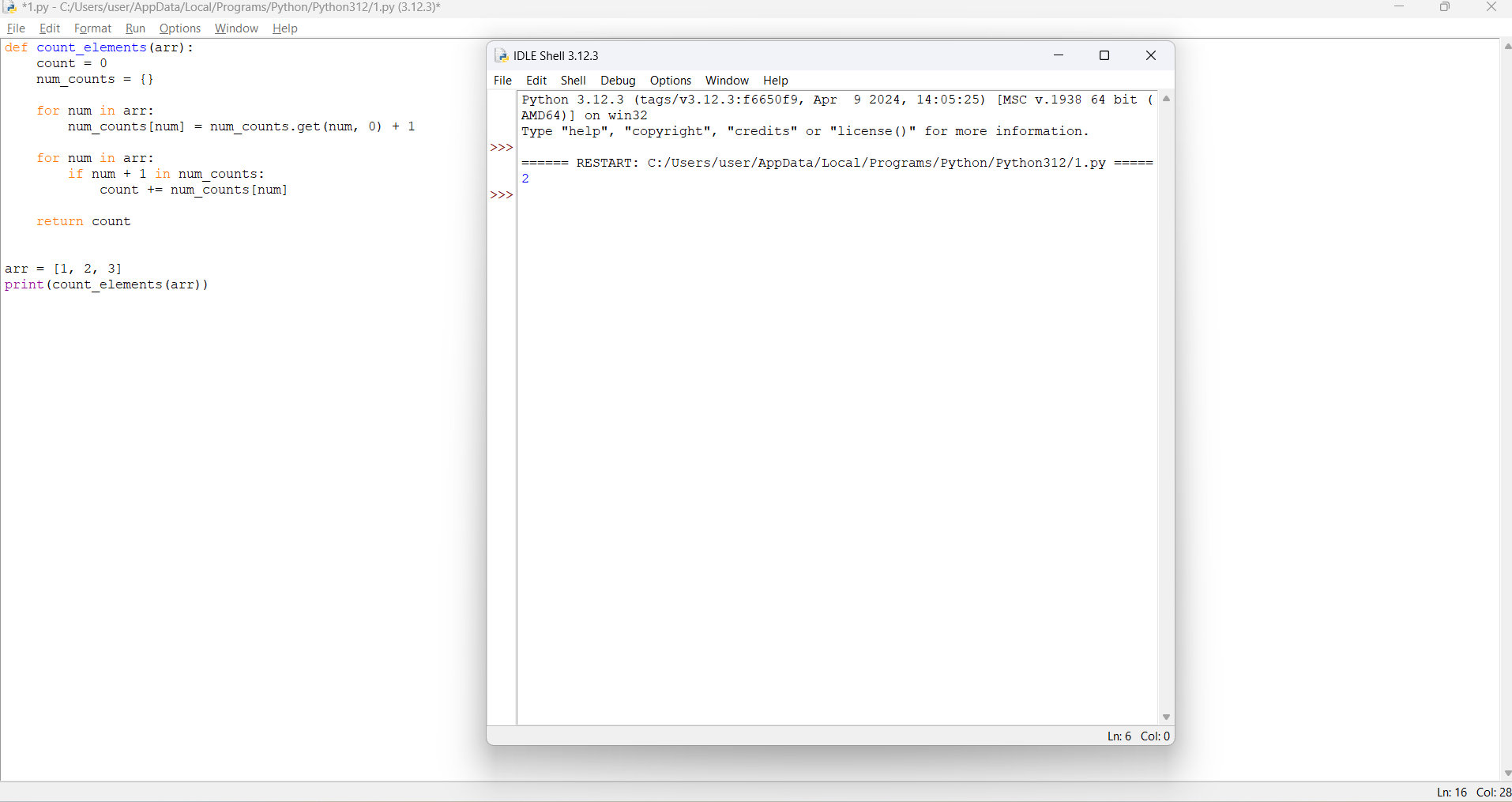
if num + 1 in num\_counts:

count += num\_counts[num]

return count

arr = [1, 2, 3]

print(count\_elements(arr))



**2. Perform String Shifts**

**You are given a string s containing lowercase English letters, and a matrix shift, where shift[i] = [directioni, amounti]:**

* **directioni can be 0 (for left shift) or 1 (for right shift).**
* **amounti is the amount by which string s is to be shifted.**
* **A left shift by 1 means remove the first character of s and append it to the end.**
* **Similarly, a right shift by 1 means remove the last character of s and add it to the beginning.**

**Return the final string after all operations.**

**Example 1:**

**Input: s = "abc", shift = [[0,1],[1,2]] Output: "cab" Explanation:**

**[0,1] means shift to left by 1. "abc" -> "bca" [1,2] means shift to right by 2. "bca" -> "cab" Example 2:**

**Input: s = "abcdefg", shift = [[1,1],[1,1],[0,2],[1,3]] Output: "efgabcd" Explanation:**

**[1,1] means shift to right by 1. "abcdefg" -> "gabcdef"**

**[1,1] means shift to right by 1. "gabcdef" -> "fgabcde"**

**[0,2] means shift to left by 2. "fgabcde" -> "abcdefg" [1,3] means shift to right by 3. "abcdefg" -> "efgabcd" Constraints:**

* **1 <= s.length <= 100**
* **s only contains lower case English letters.**
* **1 <= shift.length <= 100**
* **shift[i].length == 2**
* **directioni is either 0 or 1.**
* **0 <= amounti <= 100**

def string\_shift(s, shifts):

total\_shift = 0

for shift in shifts:

direction, amount = shift

if direction == 0:

total\_shift -= amount

else:

total\_shift += amount

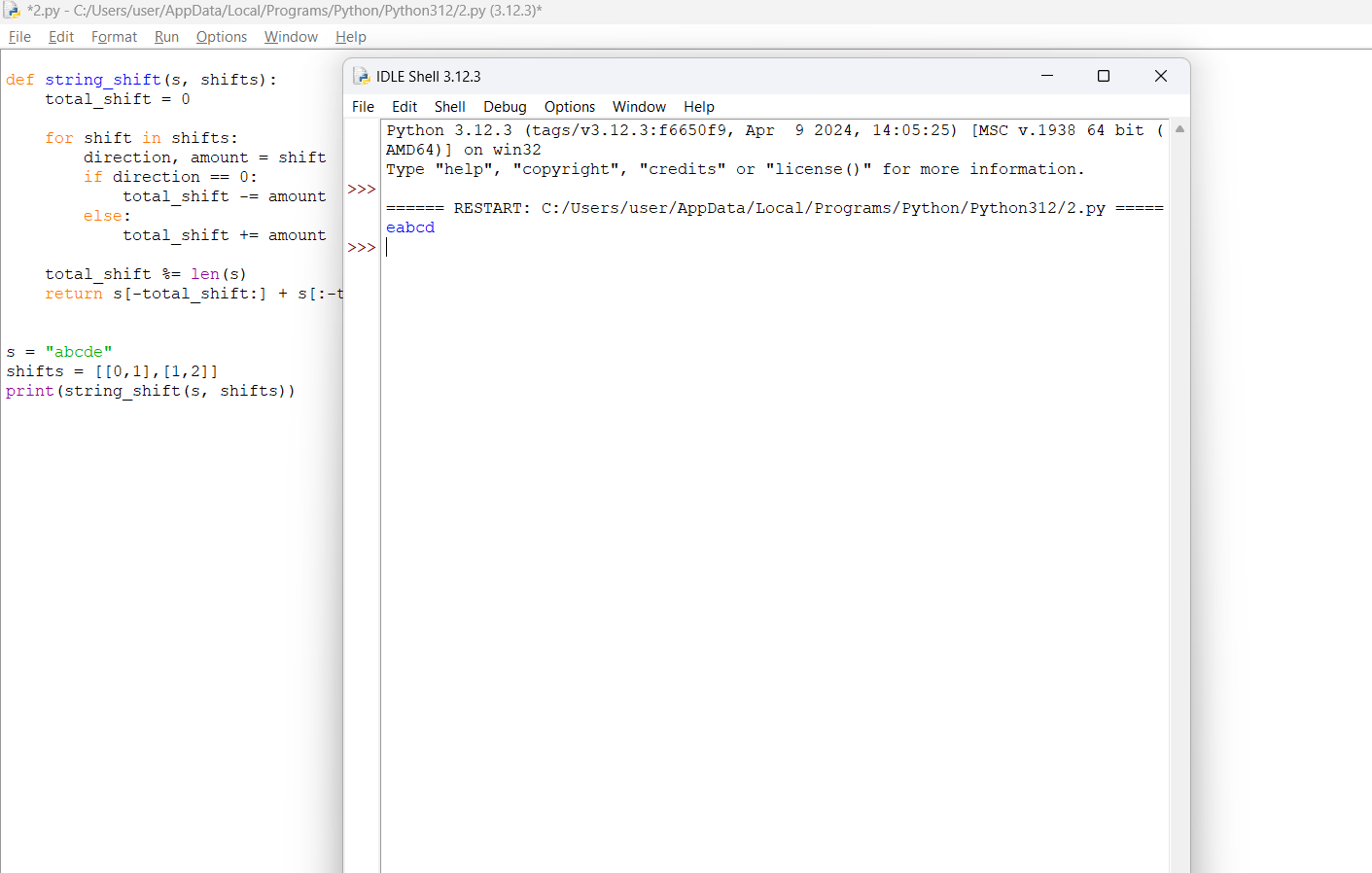
total\_shift %= len(s)

return s[-total\_shift:] + s[:-total\_shift]

s = "abcde"

shifts = [[0,1],[1,2]]

print(string\_shift(s, shifts))



**3. Leftmost Column with at Least a One**

**A row-sorted binary matrix means that all elements are 0 or 1 and each row of the matrix is sorted in non-decreasing order.**

**Given a row-sorted binary matrix binaryMatrix, return the index (0-indexed) of the leftmost column with a 1 in it. If such an index does not exist, return -1.**

**You can't access the Binary Matrix directly. You may only access the matrix using a BinaryMatrix interface:**

* **BinaryMatrix.get(row, col) returns the element of the matrix at index (row, col) (0-indexed).**
* **BinaryMatrix.dimensions() returns the dimensions of the matrix as a list of 2 elements [rows, cols], which means the matrix is rows x cols.**

**Submissions making more than 1000 calls to BinaryMatrix.get will be judged Wrong Answer. Also, any solutions that attempt to circumvent the judge will result in disqualification.**

**For custom testing purposes, the input will be the entire binary matrix mat. You will not have access to the binary matrix directly.**

**Example 1:**

****

**Input: mat = [[0,0],[1,1]]**

**Output: 0**

**Example 2:**

****

**Input: mat = [[0,0],[0,1]]**

**Output: 1**

**Example 3:**

****

**Input: mat = [[0,0],[0,0]]**

**Output: -1**

**Constraints:**

* **rows == mat.length**
* **cols == mat[i].length**
* **1 <= rows, cols <= 100**
* **mat[i][j] is either 0 or 1.**
* **mat[i] is sorted in non-decreasing order.**

class BinaryMatrix:

def \_\_init\_\_(self, mat):

self.mat = mat

def get(self, row, col):

return self.mat[row][col]

def dimensions(self):

return [len(self.mat), len(self.mat[0])]

def leftmost\_column\_with\_one(binaryMatrix):

rows, cols = binaryMatrix.dimensions()

min\_col = float('inf')

for row in range(rows):

left, right = 0, cols - 1

# Binary search to find the leftmost 1 in each row

while left <= right:

mid = left + (right - left) // 2

if binaryMatrix.get(row, mid) == 1:

min\_col = min(min\_col, mid)

right = mid - 1

else:

left = mid + 1

return min\_col if min\_col != float('inf') else -1

mat1 = [[0,0],[1,1]]

mat2 = [[0,0],[0,1]]

mat3 = [[0,0],[0,0]]

binaryMatrix1 = BinaryMatrix(mat1)

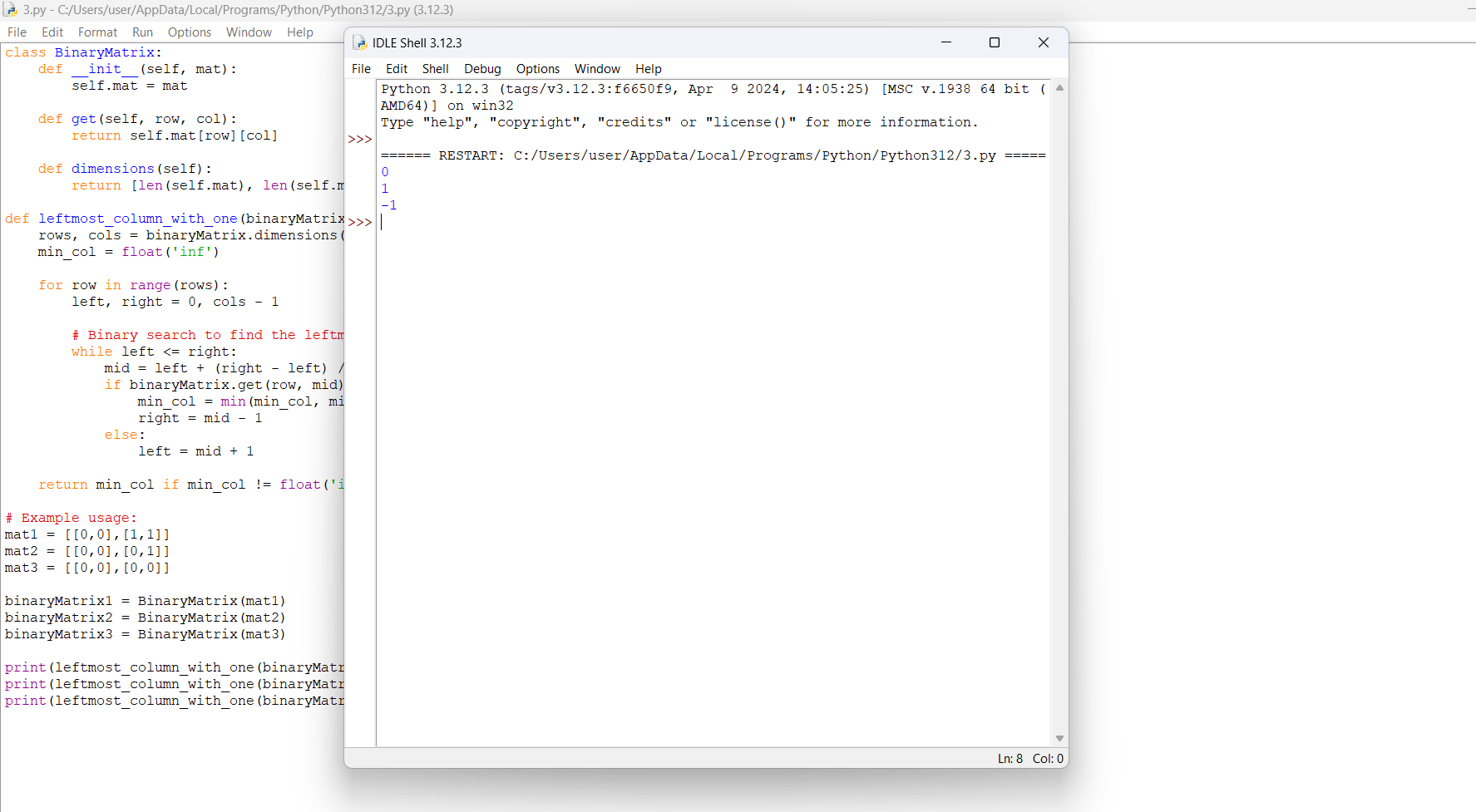
binaryMatrix2 = BinaryMatrix(mat2)

binaryMatrix3 = BinaryMatrix(mat3)

print(leftmost\_column\_with\_one(binaryMatrix1))

print(leftmost\_column\_with\_one(binaryMatrix2))

print(leftmost\_column\_with\_one(binaryMatrix3))



**4. First Unique Number**

**You have a queue of integers, you need to retrieve the first unique integer in the queue. Implement the FirstUnique class:**

* **FirstUnique(int[] nums) Initializes the object with the numbers in the queue. ● int showFirstUnique() returns the value of the first unique integer of the queue, and returns -1 if there is no such integer.**
* **void add(int value) insert value to the queue.**

**Example 1:**

**Input:**

**["FirstUnique","showFirstUnique","add","showFirstUnique","add","showFirstUnique","a dd","showFirstUnique"]**

**[[[2,3,5]],[],[5],[],[2],[],[3],[]] Output:**

**[null,2,null,2,null,3,null,-1] Explanation:**

**FirstUnique firstUnique = new FirstUnique([2,3,5]); firstUnique.showFirstUnique(); // return 2 firstUnique.add(5); // the queue is now [2,3,5,5] firstUnique.showFirstUnique(); // return 2**

**firstUnique.add(2); // the queue is now [2,3,5,5,2] firstUnique.showFirstUnique(); // return 3**

**firstUnique.add(3); // the queue is now [2,3,5,5,2,3] firstUnique.showFirstUnique(); // return -1**

**Example 2:**

**Input:**

**["FirstUnique","showFirstUnique","add","add","add","add","add","showFirstUnique"] [[[7,7,7,7,7,7]],[],[7],[3],[3],[7],[17],[]] Output:**

**[null,-1,null,null,null,null,null,17] Explanation:**

**FirstUnique firstUnique = new FirstUnique([7,7,7,7,7,7]); firstUnique.showFirstUnique(); // return -1**

**firstUnique.add(7); // the queue is now [7,7,7,7,7,7,7] firstUnique.add(3); // the queue is now [7,7,7,7,7,7,7,3] firstUnique.add(3); // the queue is now [7,7,7,7,7,7,7,3,3] firstUnique.add(7); // the queue is now [7,7,7,7,7,7,7,3,3,7] firstUnique.add(17); // the queue is now [7,7,7,7,7,7,7,3,3,7,17] firstUnique.showFirstUnique(); // return 17**

**Example 3:**

**Input:**

**["FirstUnique","showFirstUnique","add","showFirstUnique"] [[[809]],[],[809],[]] Output: [null,809,null,-1] Explanation:**

**FirstUnique firstUnique = new FirstUnique([809]); firstUnique.showFirstUnique(); // return 809 firstUnique.add(809); // the queue is now [809,809] firstUnique.showFirstUnique(); // return -1**

**Constraints:**

* **1 <= nums.length <= 10^5**
* **1 <= nums[i] <= 10^8**
* **1 <= value <= 10^8**
* **At most 50000 calls will be made to showFirstUnique and add.**

from collections import defaultdict, deque

class FirstUnique:

def \_\_init\_\_(self, nums):

self.unique\_dict = defaultdict(int)

self.queue = deque()

for num in nums:

self.add(num)

def showFirstUnique(self):

while self.queue and self.unique\_dict[self.queue[0]] > 1:

self.queue.popleft()

return self.queue[0] if self.queue else -1

def add(self, value):

self.unique\_dict[value] += 1

if self.unique\_dict[value] == 1:

self.queue.append(value)

# Example usage:

firstUnique1 = FirstUnique([2,3,5])

print(firstUnique1.showFirstUnique()) # Output: 2

firstUnique1.add(5)

print(firstUnique1.showFirstUnique()) # Output: 2

firstUnique1.add(2)

print(firstUnique1.showFirstUnique()) # Output: 3

firstUnique1.add(3)

print(firstUnique1.showFirstUnique()) # Output: -1

firstUnique2 = FirstUnique([7,7,7,7,7,7])

print(firstUnique2.showFirstUnique()) # Output: -1

firstUnique2.add(7)

firstUnique2.add(3)

firstUnique2.add(3)

firstUnique2.add(7)

firstUnique2.add(17)

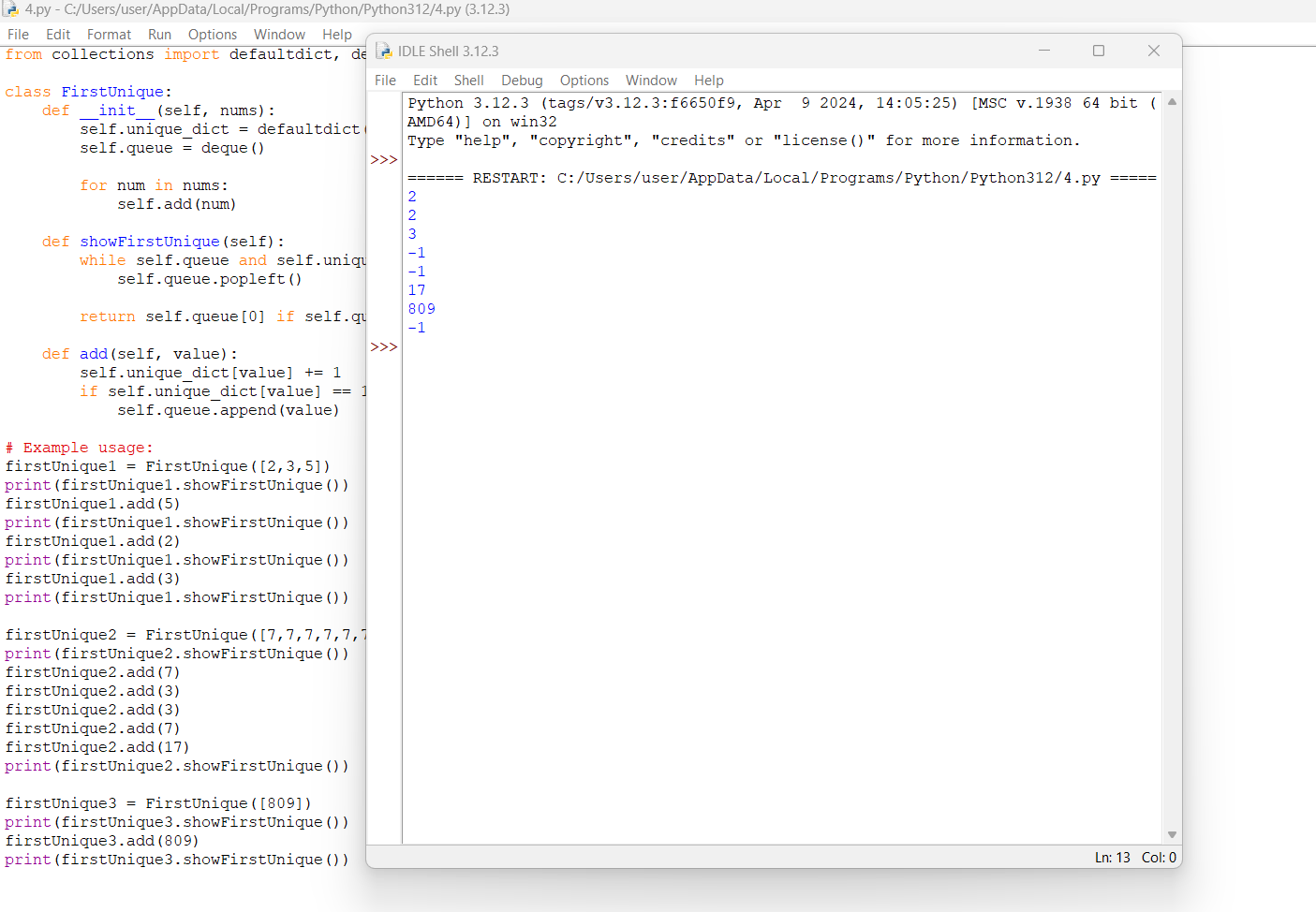
print(firstUnique2.showFirstUnique()) # Output: 17

firstUnique3 = FirstUnique([809])

print(firstUnique3.showFirstUnique()) # Output: 809

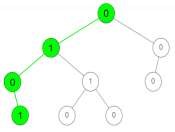
firstUnique3.add(809)

print(firstUnique3.showFirstUnique()) # Output: -1



1. **Check If a String Is a Valid Sequence from Root to Leaves Path in a Binary Tree Given a binary tree where each path going from the root to any leaf form a valid sequence, check if a given string is a valid sequence in such binary tree. We get the given string from the concatenation of an array of integers arr and the concatenation of all values of the nodes along a path results in a sequence in the given binary tree.**

**Example 1:**

****

**Input: root = [0,1,0,0,1,0,null,null,1,0,0], arr = [0,1,0,1] Output: true**

**Explanation:**

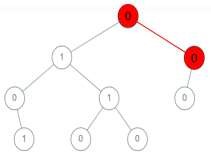
**The path 0 -> 1 -> 0 -> 1 is a valid sequence (green color in the figure).**

**Other valid sequences are:**

**0 -> 1 -> 1 -> 0**

**0 -> 0 -> 0**

**Example 2:**

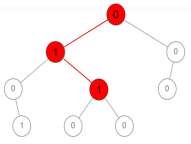
****

**Input: root = [0,1,0,0,1,0,null,null,1,0,0], arr = [0,0,1]**

**Output: false**

**Explanation: The path 0 -> 0 -> 1 does not exist, therefore it is not even a sequence.**

**Example 3:**

****

**Input: root = [0,1,0,0,1,0,null,null,1,0,0], arr = [0,1,1]**

**Output: false**

**Explanation: The path 0 -> 1 -> 1 is a sequence, but it is not a valid sequence.**

**Constraints:**

●

1

<= arr

.length <= 5000

●

0

<= arr[i] <=

9

**● Each node's value is between [0 - 9].**

class TreeNode:

def \_\_init\_\_(self, val=0, left=None, right=None):

self.val = val

self.left = left

self.right = right

def isValidSequence(root, arr):

def dfs(node, index):

if not node or index == len(arr):

return False

if node.val != arr[index]:

return False

if not node.left and not node.right and index == len(arr) - 1:

return True

return dfs(node.left, index + 1) or dfs(node.right, index + 1)

return dfs(root, 0)

# Example usage:

# Example 1:

root1 = TreeNode(0)

root1.left = TreeNode(1)

root1.right = TreeNode(0)

root1.left.left = TreeNode(0)

root1.left.right = TreeNode(1)

root1.right.left = TreeNode(0)

root1.left.left.right = TreeNode(1)

root1.left.right.left = TreeNode(0)

arr1 = [0, 1, 0, 1]

print(isValidSequence(root1, arr1)) # Output: True

# Example 2:

root2 = TreeNode(0)

root2.left = TreeNode(1)

root2.right = TreeNode(0)

root2.left.left = TreeNode(0)

root2.left.right = TreeNode(1)

root2.right.left = TreeNode(0)

root2.left.left.right = TreeNode(1)

root2.left.right.left = TreeNode(0)

arr2 = [0, 0, 1]

print(isValidSequence(root2, arr2)) # Output: False

# Example 3:

root3 = TreeNode(0)

root3.left = TreeNode(1)

root3.right = TreeNode(0)

root3.left.left = TreeNode(0)

root3.left.right = TreeNode(1)

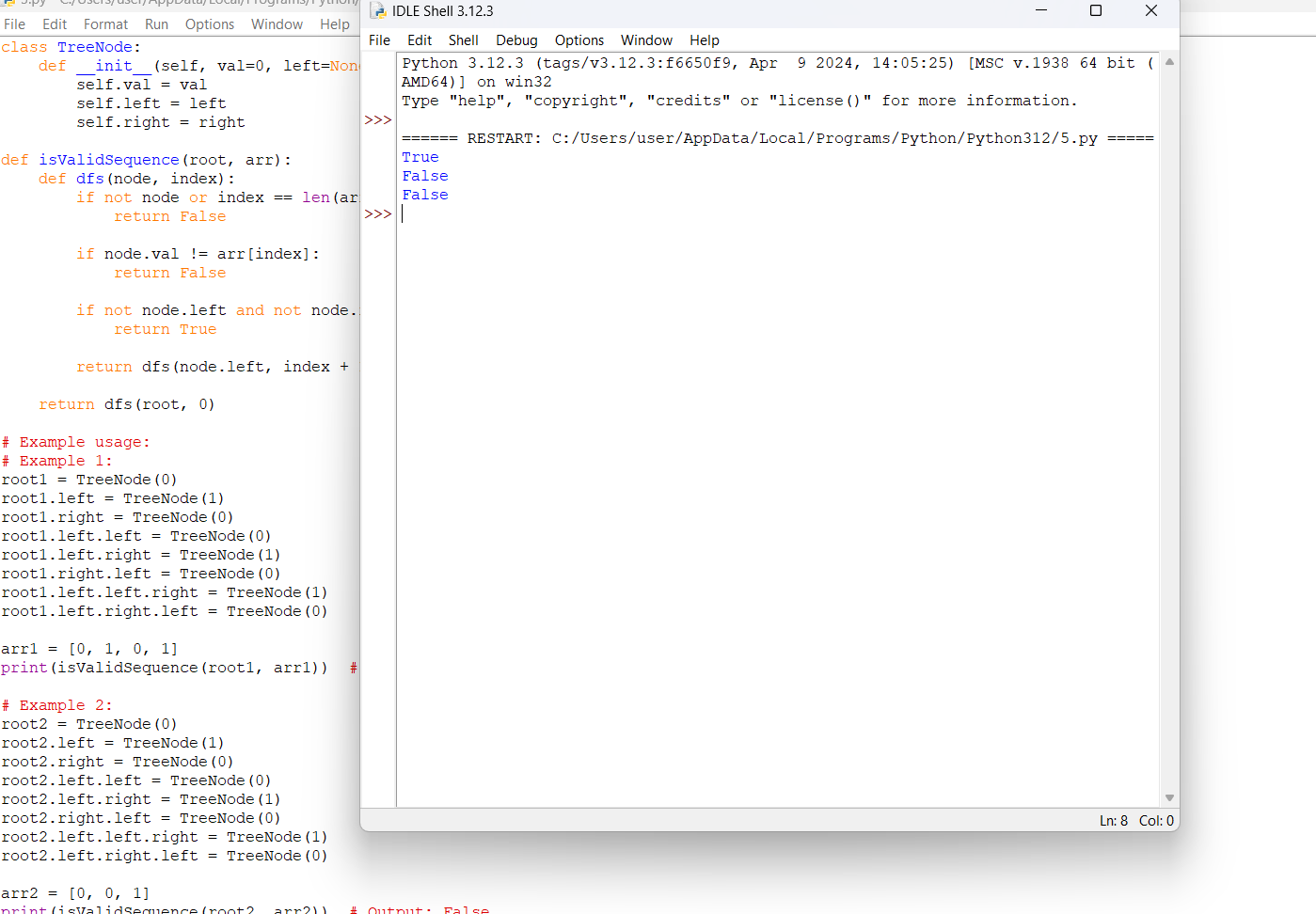
root3.right.left = TreeNode(0)

root3.left.left.right = TreeNode(1)

root3.left.right.left = TreeNode(0)

arr3 = [0, 1, 1]

print(isValidSequence(root3, arr3)) # Output: False



1. **Kids With the Greatest Number of Candies**

**There are n kids with candies. You are given an integer array candies, where each candies[i] represents the number of candies the ith kid has, and an integer extraCandies, denoting the number of extra candies that you have.**

**Return a boolean array result of length n, where result[i] is true if, after giving the ith kid all the extraCandies, they will have the greatest number of candies among all the kids, or false otherwise.**

**Note that multiple kids can have the greatest number of candies.**

**Example 1:**

**Input: candies = [2,3,5,1,3], extraCandies = 3**

**Output: [true,true,true,false,true]**

**Explanation: If you give all extraCandies to:**

* **Kid 1, they will have 2 + 3 = 5 candies, which is the greatest among the kids.**
* **Kid 2, they will have 3 + 3 = 6 candies, which is the greatest among the kids.**
* **Kid 3, they will have 5 + 3 = 8 candies, which is the greatest among the kids.**
* **Kid 4, they will have 1 + 3 = 4 candies, which is not the greatest among the kids. - Kid 5, they will have 3 + 3 = 6 candies, which is the greatest among the kids.**

**Example 2:**

**Input: candies = [4,2,1,1,2], extraCandies = 1 Output: [true,false,false,false,false] Explanation: There is only 1 extra candy.**

**Kid 1 will always have the greatest number of candies, even if a different kid is given the extra candy.**

**Example 3:**

**Input: candies = [12,1,12], extraCandies = 10**

**Output: [true,false,true]**

**Constraints:**

* + **n == candies.length**
  + **2 <= n <= 100**
  + **1 <= candies[i] <= 100**
  + **1 <= extraCandies <= 50**

def kidsWithCandies(candies, extraCandies):

max\_candies = max(candies)

result = []

for candy in candies:

if candy + extraCandies >= max\_candies:

result.append(True)

else:

result.append(False)

return result

# Example usage:

candies1 = [2, 3, 5, 1, 3]

extraCandies1 = 3

print(kidsWithCandies(candies1, extraCandies1)) # Output: [True, True, True, False, True]

candies2 = [4, 2, 1, 1, 2]

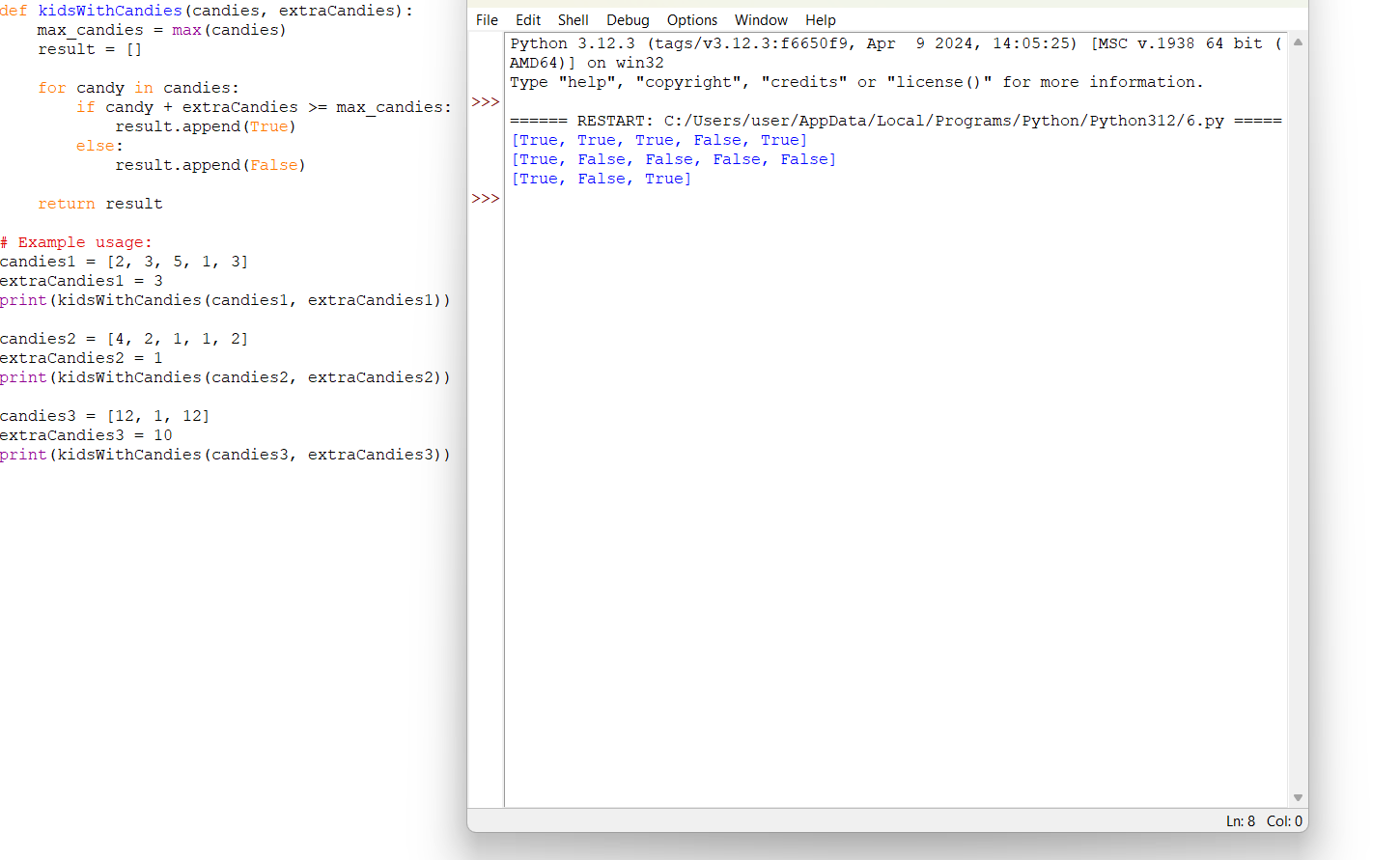
extraCandies2 = 1

print(kidsWithCandies(candies2, extraCandies2)) # Output: [True, False, False, False, False]

candies3 = [12, 1, 12]

extraCandies3 = 10

print(kidsWithCandies(candies3, extraCandies3)) # Output: [True, False, True]

****

**7. Max Difference You Can Get From Changing an Integer**

**You are given an integer num. You will apply the following steps exactly two times:**

* **Pick a digit x (0 <= x <= 9).**
* **Pick another digit y (0 <= y <= 9). The digit y can be equal to x.**
* **Replace all the occurrences of x in the decimal representation of num by y.**
* **The new integer cannot have any leading zeros, also the new integer cannot be 0. Let a and b be the results of applying the operations to num the first and second times, respectively.**

**Return the max difference between a and b.**

**Example 1:**

**Input: num = 555**

**Output: 888**

**Explanation: The first time pick x = 5 and y = 9 and store the new integer in a.**

**The second time pick x = 5 and y = 1 and store the new integer in b.**

**We have now a = 999 and b = 111 and max difference = 888**

**Example 2:**

**Input: num = 9**

**Output: 8**

**Explanation: The first time pick x = 9 and y = 9 and store the new integer in a.**

**The second time pick x = 9 and y = 1 and store the new integer in b.**

**We have now a = 9 and b = 1 and max difference = 8**

**Constraints:**

**● 1 <= num <= 108**

def maxDiff(num):

num\_str = str(num)

max\_val = int(num\_str)

min\_val = int(num\_str)

for i in range(10):

for j in range(10):

new\_num\_str = num\_str.replace(str(i), str(j))

if new\_num\_str[0] != '0':

max\_val = max(max\_val, int(new\_num\_str))

min\_val = min(min\_val, int(new\_num\_str))

return max\_val - min\_val

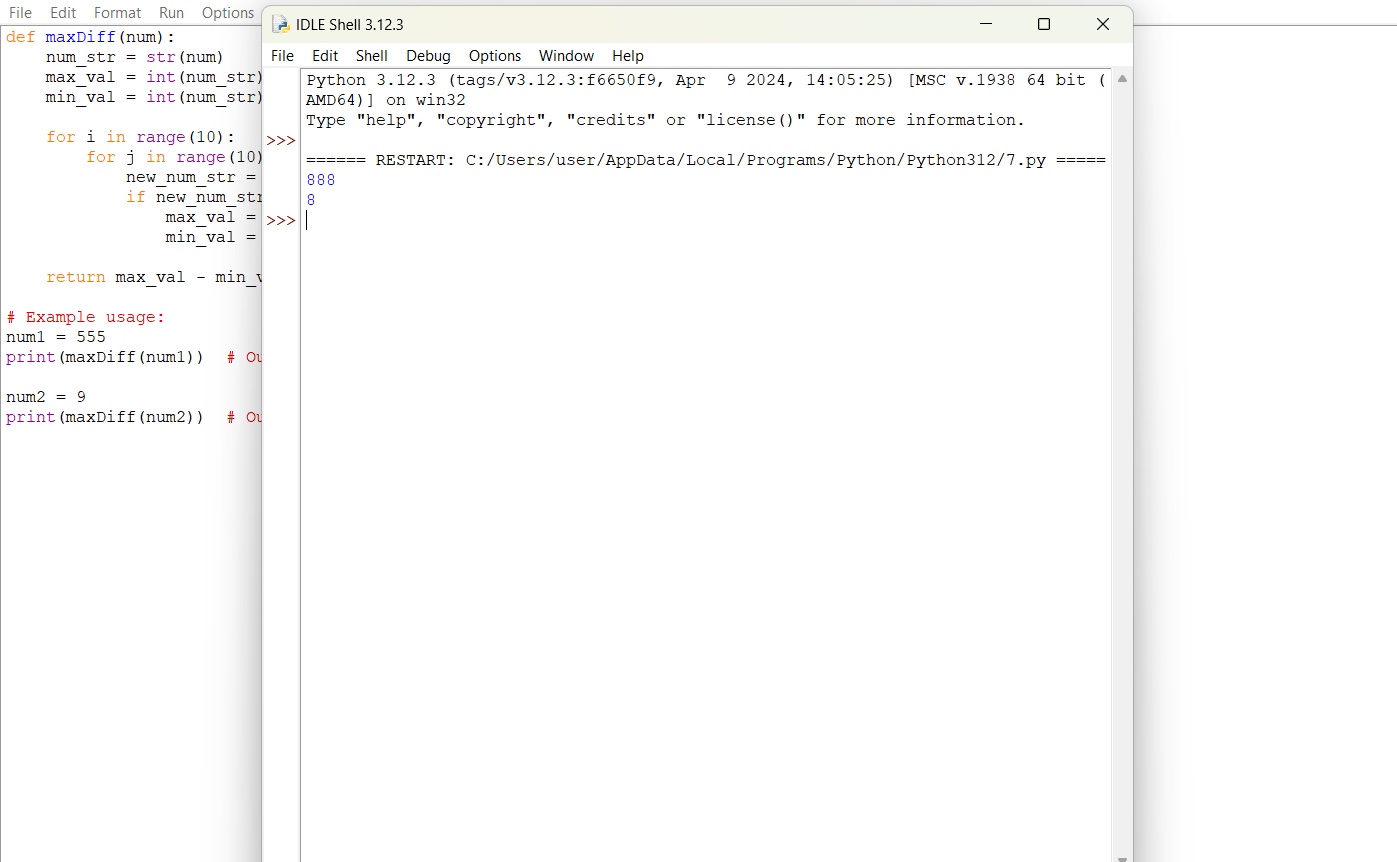
# Example usage:

num1 = 555

print(maxDiff(num1)) # Output: 888

num2 = 9

print(maxDiff(num2)) # Output: 8



**8. Check If a String Can Break Another String**

**Given two strings: s1 and s2 with the same size, check if some permutation of string s1 can break some permutation of string s2 or vice-versa. In other words s2 can break s1 or vice-versa.**

**A string x can break string y (both of size n) if x[i] >= y[i] (in alphabetical order) for all i between 0 and n-1.**

**Example 1:**

**Input: s1 = "abc", s2 = "xya"**

**Output: true**

**Explanation: "ayx" is a permutation of s2="xya" which can break to string "abc" which is a permutation of s1="abc".**

**Example 2:**

**Input: s1 = "abe", s2 = "acd"**

**Output: false**

**Explanation: All permutations for s1="abe" are: "abe", "aeb", "bae", "bea", "eab" and "eba" and all permutation for s2="acd" are: "acd", "adc", "cad", "cda", "dac" and "dca". However, there is not any permutation from s1 which can break some permutation from s2 and vice-versa.**

**Example 3:**

**Input: s1 = "leetcodee", s2 = "interview"**

**Output: true**

**Constraints:**

* **s1.length == n**
* **s2.length == n**
* **1 <= n <= 10^5**
* **All strings consist of lowercase English letters.**

def canBreak(s1, s2):

count\_s1 = [0] \* 26

count\_s2 = [0] \* 26

# Count occurrences of characters in s1

for char in s1:

count\_s1[ord(char) - ord('a')] += 1

# Count occurrences of characters in s2

for char in s2:

count\_s2[ord(char) - ord('a')] += 1

# Check if s1 can break s2

s1\_breaks\_s2 = True

s2\_breaks\_s1 = True

s1\_smaller\_count = 0

s2\_smaller\_count = 0

for i in range(26):

s1\_smaller\_count += count\_s1[i]

s2\_smaller\_count += count\_s2[i]

# If s1 has more smaller characters than s2

if s1\_smaller\_count < s2\_smaller\_count:

s1\_breaks\_s2 = False

# If s2 has more smaller characters than s1

if s2\_smaller\_count < s1\_smaller\_count:

s2\_breaks\_s1 = False

return s1\_breaks\_s2 or s2\_breaks\_s1

# Example usage:

s1\_1 = "abc"

s2\_1 = "xya"

print(canBreak(s1\_1, s2\_1)) # Output: True

s1\_2 = "abe"

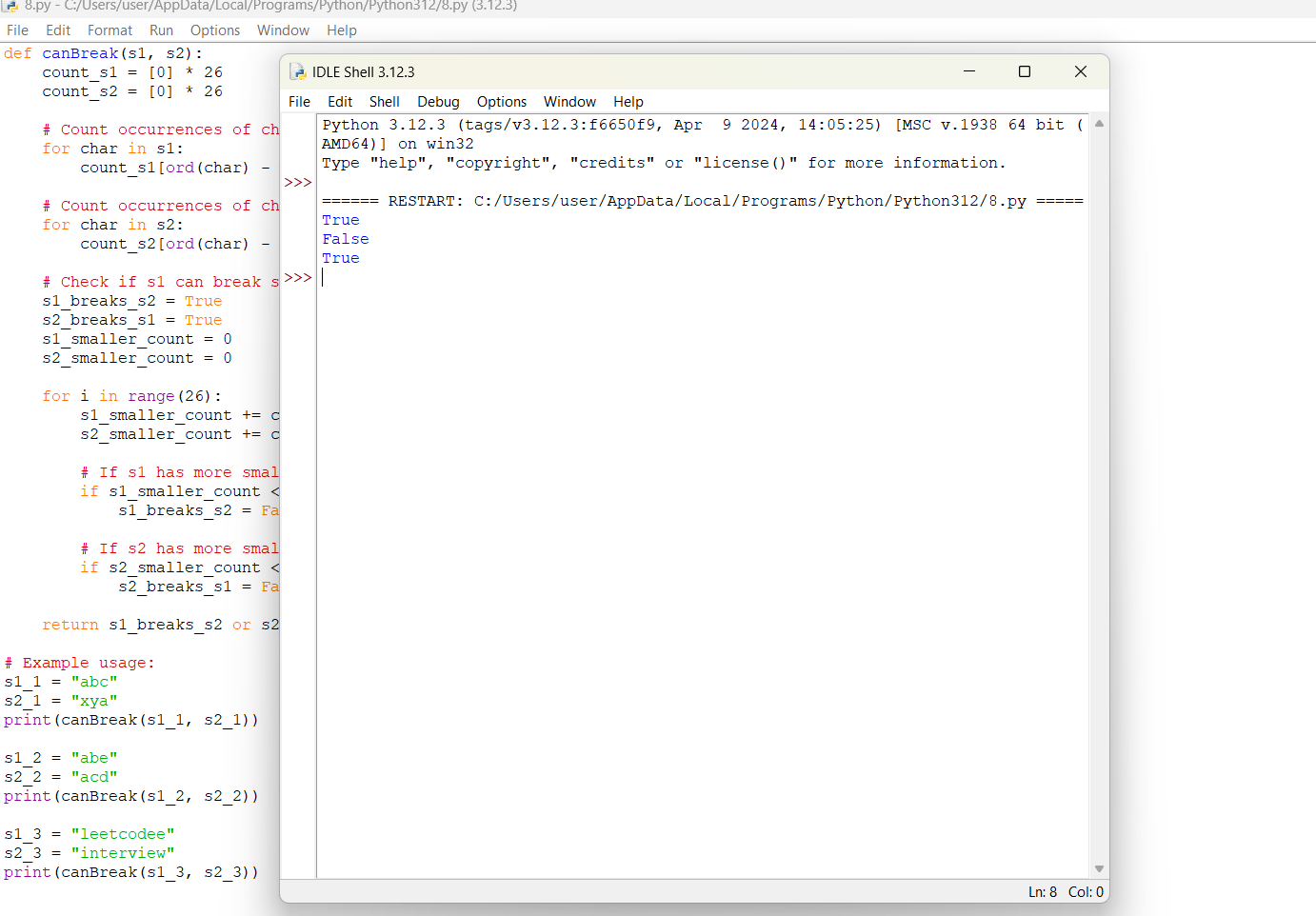
s2\_2 = "acd"

print(canBreak(s1\_2, s2\_2)) # Output: False

s1\_3 = "leetcodee"

s2\_3 = "interview"

print(canBreak(s1\_3, s2\_3)) # Output: True



**9. Number of Ways to Wear Different Hats to Each Other**

**There are n people and 40 types of hats labeled from 1 to 40.**

**Given a 2D integer array hats, where hats[i] is a list of all hats preferred by the ith person.**

**Return the number of ways that the n people wear different hats to each other. Since the answer may be too large, return it modulo 109 + 7.**

**Example 1:**

**Input: hats = [[3,4],[4,5],[5]]**

**Output: 1**

**Explanation: There is only one way to choose hats given the conditions.**

**First person choose hat 3, Second person choose hat 4 and last one hat 5.**

**Example 2:**

**Input: hats = [[3,5,1],[3,5]]**

**Output: 4**

**Explanation: There are 4 ways to choose hats:**

**(3,5), (5,3), (1,3) and (1,5)**

**Example 3:**

**Input: hats = [[1,2,3,4],[1,2,3,4],[1,2,3,4],[1,2,3,4]]**

**Output: 24**

**Explanation: Each person can choose hats labeled from 1 to 4.**

**Number of Permutations of (1,2,3,4) = 24. Constraints:**

* **n == hats.length**
* **1 <= n <= 10**
* **1 <= hats[i].length <= 40**
* **1 <= hats[i][j] <= 40**
* **hats[i] contains a list of unique integers.**

def numberWays(hats):

MOD = 10\*\*9 + 7

n = len(hats)

# dp[mask][i] represents the number of ways to assign hats to the people in the mask

dp = [[0] \* (1 << n) for \_ in range(41)]

dp[0][0] = 1

# hat\_to\_people[i] stores the list of people who like hat i

hat\_to\_people = [[] for \_ in range(41)]

for i, p in enumerate(hats):

for hat in p:

hat\_to\_people[hat].append(i)

# Iterate through all hats

for hat in range(1, 41):

for mask in range(1 << n):

# Copy previous state

dp[hat][mask] = dp[hat - 1][mask]

# Iterate through all people who like this hat

for person in hat\_to\_people[hat]:

# Check if the person is already wearing a hat

if mask & (1 << person):

dp[hat][mask] += dp[hat - 1][mask ^ (1 << person)]

dp[hat][mask] %= MOD

return dp[40][(1 << n) - 1]

# Example usage:

hats1 = [[3, 4], [4, 5], [5]]

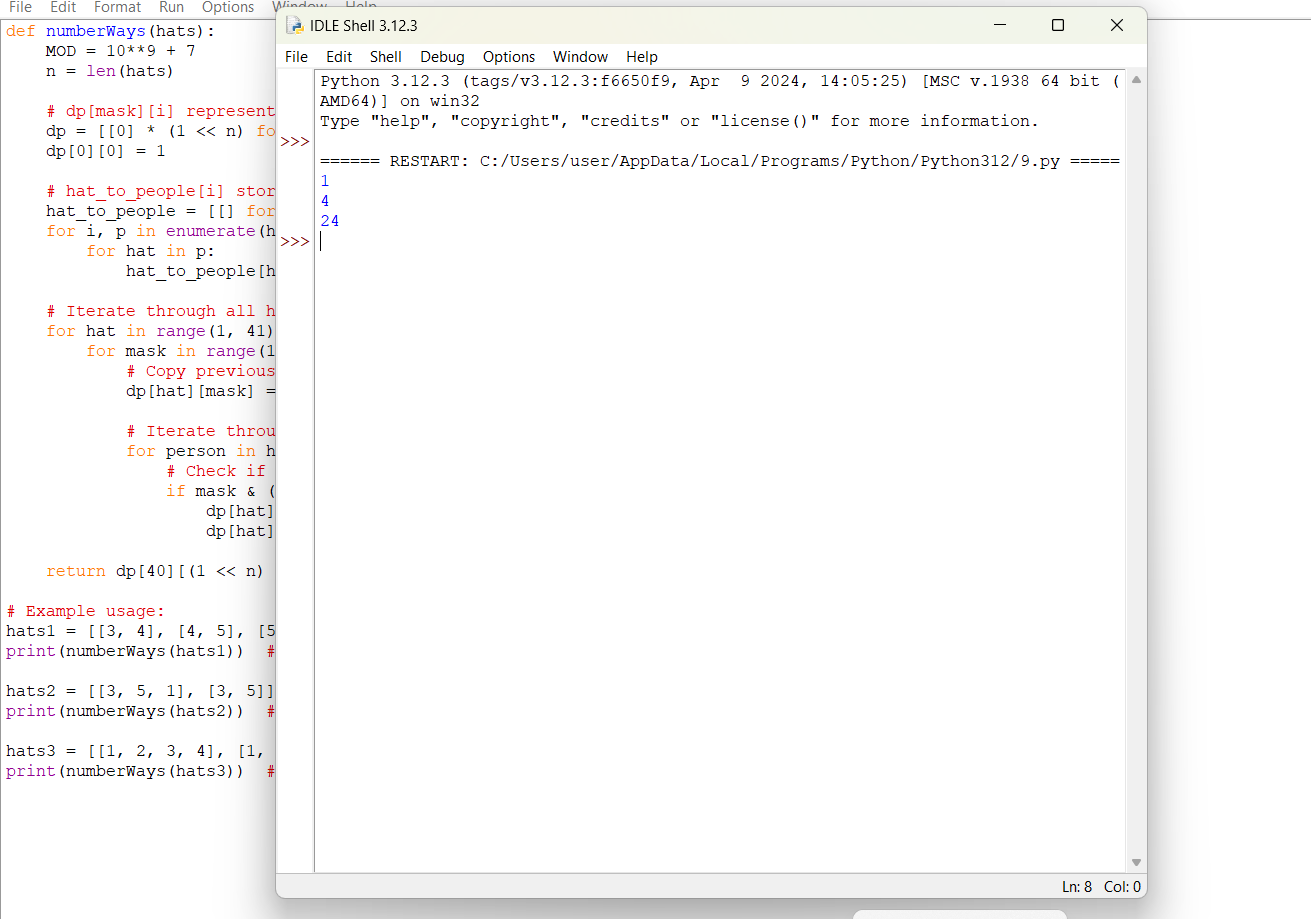
print(numberWays(hats1)) # Output: 1

hats2 = [[3, 5, 1], [3, 5]]

print(numberWays(hats2)) # Output: 4

hats3 = [[1, 2, 3, 4], [1, 2, 3, 4], [1, 2, 3, 4], [1, 2, 3, 4]]

print(numberWays(hats3)) # Output: 24



**10. Destination City**

**You are given the array paths, where paths[i] = [cityAi, cityBi] means there exists a direct path going from cityAi to cityBi. Return the destination city, that is, the city without any path outgoing to another city.**

**It is guaranteed that the graph of paths forms a line without any loop, therefore, there will be exactly one destination city.**

**Example 1:**

**Input: paths = [["London","New York"],["New York","Lima"],["Lima","Sao Paulo"]] Output: "Sao Paulo"**

**Explanation: Starting at "London" city you will reach "Sao Paulo" city which is the destination city. Your trip consist of: "London" -> "New York" -> "Lima" -> "Sao Paulo".**

**Example 2:**

**Input: paths = [["B","C"],["D","B"],["C","A"]]**

**Output: "A"**

**Explanation: All possible trips are:**

**"D" -> "B" -> "C" -> "A".**

**"B" -> "C" -> "A".**

**"C" -> "A".**

**"A".**

**Clearly the destination city is "A".**

**Example 3:**

**Input: paths = [["A","Z"]] Output: "Z" Constraints:**

**1 <= paths.length <= 100 paths[i].length == 2**

**1 <= cityAi.length, cityBi.length <= 10 cityAi != cityBi**

**All strings consist of lowercase and uppercase English letters and the space character.**

def destCity(paths):

outgoing\_cities = set()

all\_cities = set()

# Store all outgoing cities

for path in paths:

outgoing\_cities.add(path[0])

all\_cities.add(path[0])

all\_cities.add(path[1])

# Find the destination city

for city in all\_cities:

if city not in outgoing\_cities:

return city

# Example usage:

paths1 = [["London", "New York"], ["New York", "Lima"], ["Lima", "Sao Paulo"]]

print(destCity(paths1)) # Output: "Sao Paulo"

paths2 = [["B", "C"], ["D", "B"], ["C", "A"]]

print(destCity(paths2)) # Output: "A"

paths3 = [["A", "Z"]]

print(destCity(paths3)) # Output: "Z"

