1.

def first\_palindromic\_string(words):

def is\_palindrome(s):

return s == s[::-1]

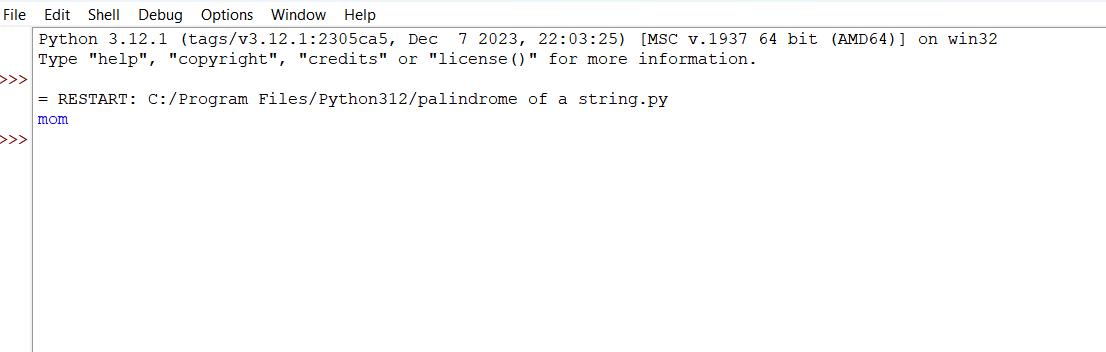
for word in words:

if is\_palindrome(word):

return word

return ""

print(first\_palindromic\_string(["hello","world"]))



2.

def calculate\_indices(nums1, nums2):

set\_nums2 = set(nums2)

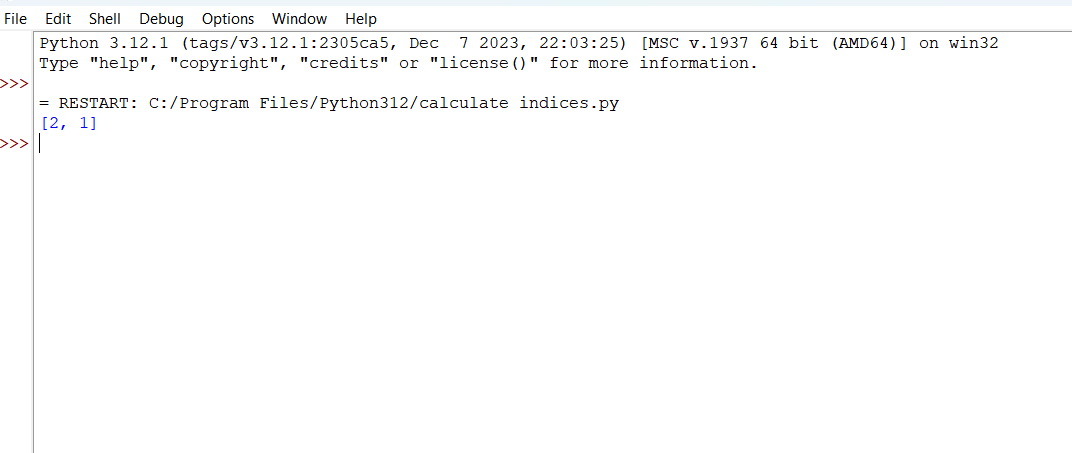
set\_nums1 = set(nums1)

answer1 = sum(1 for num in nums1 if num in set\_nums2)

answer2 = sum(1 for num in nums2 if num in set\_nums1)

return [answer1, answer2]

print(calculate\_indices([2, 3, 2], [1, 2]))



3.

def sum\_of\_squares\_of\_distinct\_counts(nums):

n = len(nums)

total\_sum = 0

for start in range(n):

distinct\_count = {}

for end in range(start, n):

num = nums[end]

if num in distinct\_count:

distinct\_count[num] += 1

else:

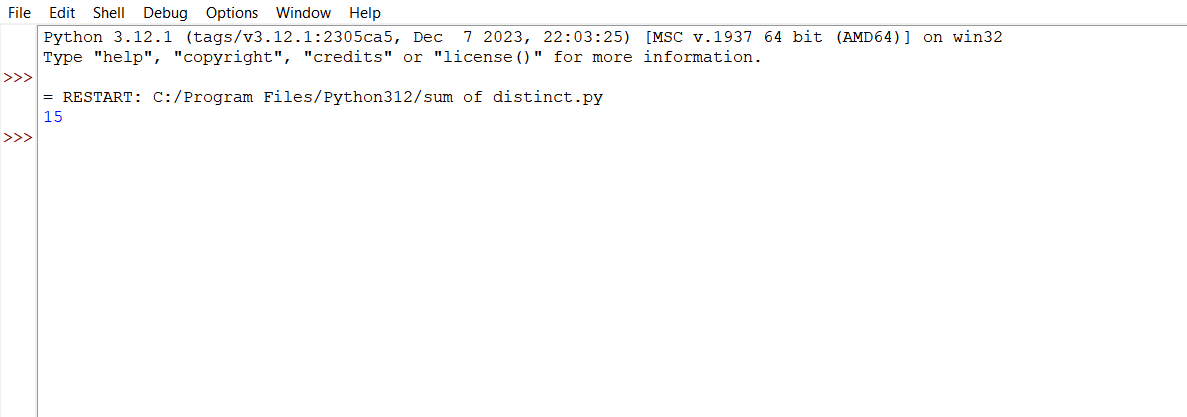
distinct\_count[num] = 1

count = len(distinct\_count)

total\_sum += count \* count

return total\_sum

print(sum\_of\_squares\_of\_distinct\_counts([1, 2, 1]))



4.

def count\_valid\_pairs(nums, k):

n = len(nums)

count = 0

for i in range(n):

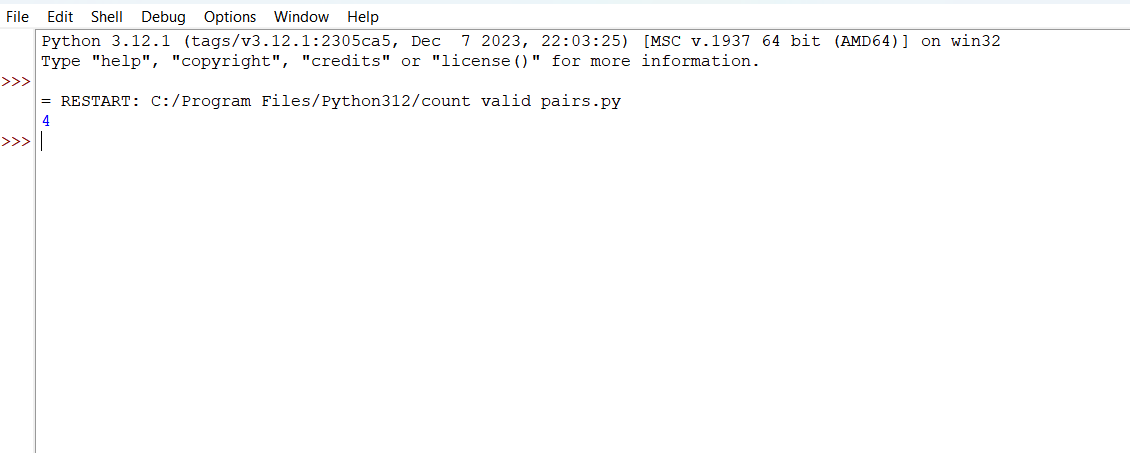
for j in range(i + 1, n):

if nums[i] == nums[j] and (i \* j) % k == 0:

count += 1

return count

print(count\_valid\_pairs([3, 1, 2, 2, 2, 1, 3], 2))



5.

def find\_maximum(nums):

max\_element = float('-inf')

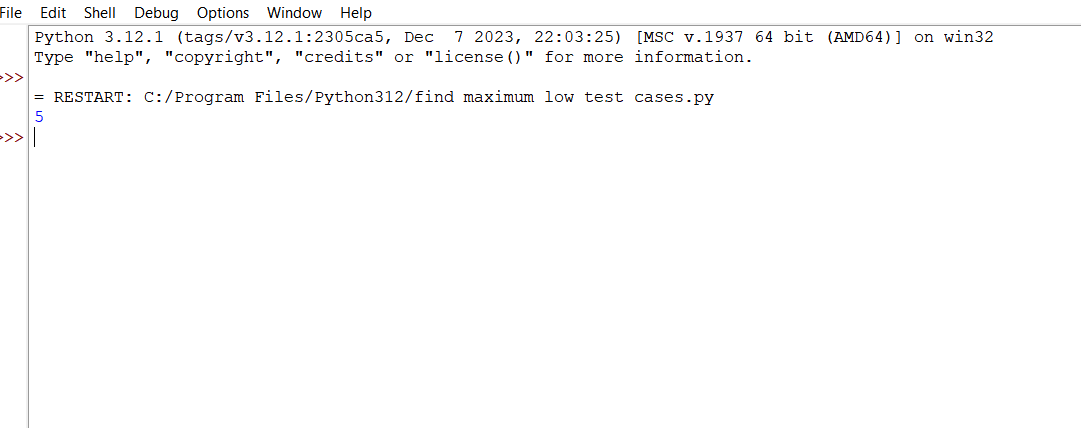
for num in nums:

if num > max\_element:

max\_element = num

return max\_element

print(find\_maximum([1, 2, 3, 4, 5]))



6.

def find\_max\_in\_sorted(nums):

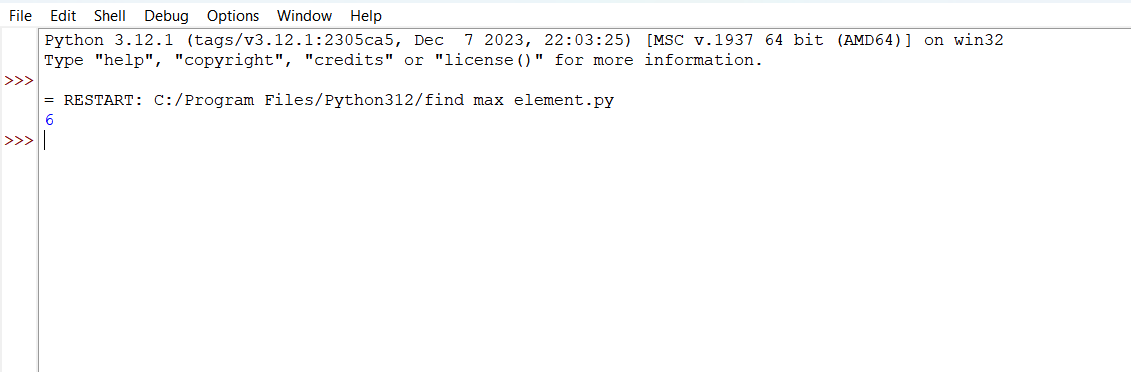
if not nums:

return None

nums.sort()

return nums[-1]

print(find\_max\_in\_sorted([3, 1, 6, 2, 4]))



7.

a=[2,3,4,3,2,1,5,6,4,5]

b=[]

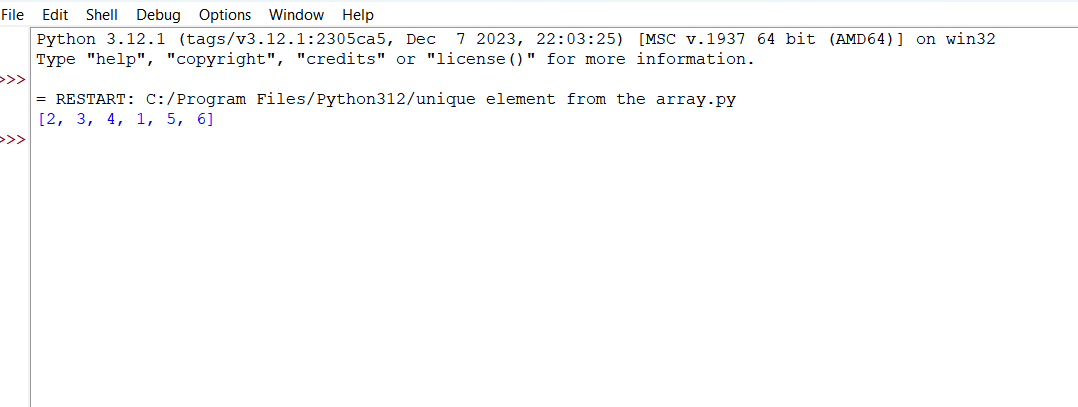
for i in range(len(a)):

for j in range(i,len(a)):

if a[i] not in b:

b.append(a[i])

print(b)



8.

a=[1,4,2,9,5,6,3]

n=len(a)

temp=0

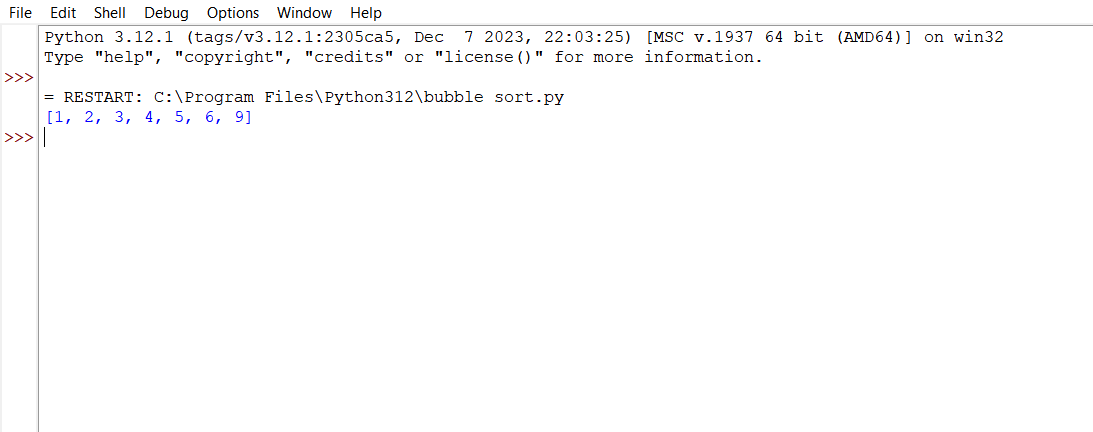
for i in range(0,n-1):

for j in range(0,n-1-i):

if(a[j]>a[j+1]):

a[j],a[j+1]=a[j+1],a[j]

print(a)



9.

def bin\_search(a,low,high,x):

if high>=low:

mid=(low+high)//2

if target>a[mid]:

return bin\_search(a,mid+1,high,x)

elif target<a[mid]:

return bin\_search(a,low,mid-1,x)

else:

return mid

else:

return -1

a={2,11,4,-1,7,8,9,6,7}

a=sorted(list(a))

print(a)

target=int(input("Enter target value:"))

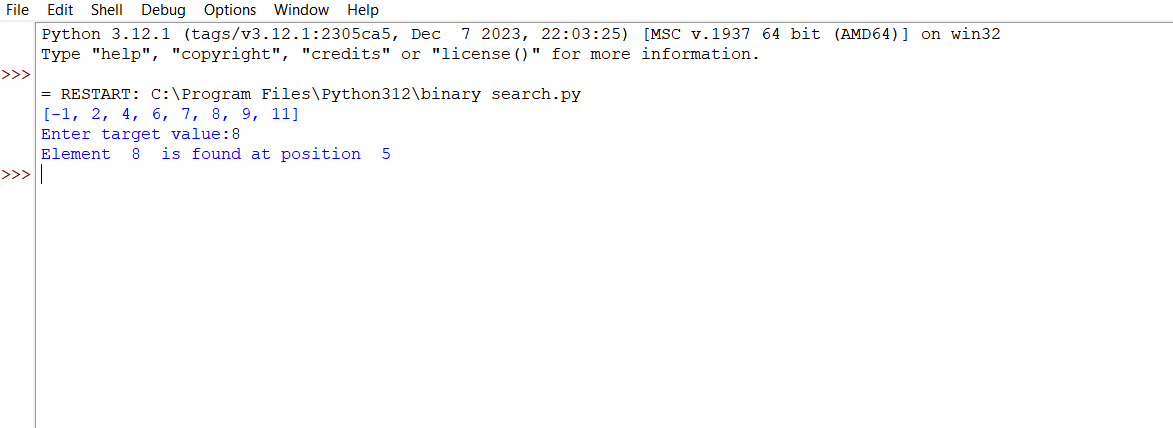
b=bin\_search(a,0,len(a)-1,target)

if a[b]==target:

print("Element ",target," is found at position ",b)

else:

print("Target element not found")



10.

def merge\_sort(nums):

if len(nums) <= 1:

return nums

mid = len(nums) // 2

left\_half = nums[:mid]

right\_half = nums[mid:]

merge\_sort(left\_half)

merge\_sort(right\_half)

i = j = k = 0

while i < len(left\_half) and j < len(right\_half):

if left\_half[i] < right\_half[j]:

nums[k] = left\_half[i]

i += 1

else:

nums[k] = right\_half[j]

j += 1

k += 1

while i < len(left\_half):

nums[k] = left\_half[i]

i += 1

k += 1

while j < len(right\_half):

nums[k] = right\_half[j]

j += 1

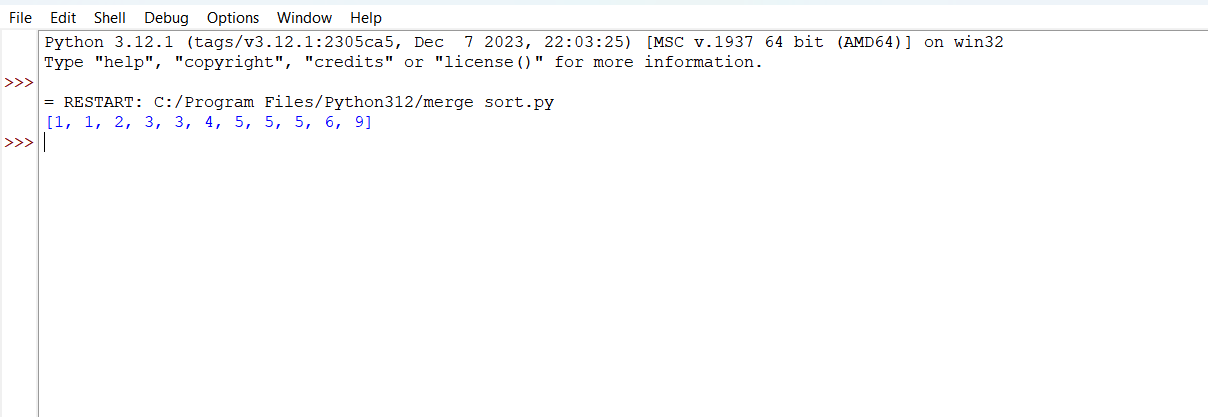
k += 1

return nums

nums = [3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5]

sorted\_nums = merge\_sort(nums)

print(sorted\_nums)



11.

def find\_ways(m, n, N, i, j):

memo = {}

def dp(x, y, steps):

if x < 0 or x >= m or y < 0 or y >= n:

return 1

if steps == 0:

return 0

if (x, y, steps) in memo:

return memo[(x, y, steps)]

ways = (

dp(x - 1, y, steps - 1) +

dp(x + 1, y, steps - 1) +

dp(x, y - 1, steps - 1) +

dp(x, y + 1, steps - 1)

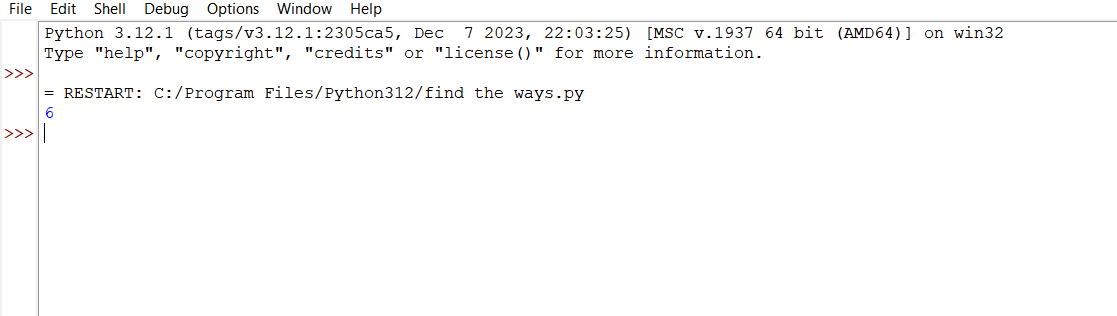
)

memo[(x, y, steps)] = ways

return ways

return dp(i, j, N)

print(find\_ways(2, 2, 2, 0, 0))



12.

def rob\_linear(nums):

n = len(nums)

if n == 0:

return 0

if n == 1:

return nums[0]

if n == 2:

return max(nums[0], nums[1])

dp = [0] \* n

dp[0] = nums[0]

dp[1] = max(nums[0], nums[1])

for i in range(2, n):

dp[i] = max(dp[i-1], dp[i-2] + nums[i])

return dp[-1]

def rob(nums):

n = len(nums)

if n == 0:

return 0

if n == 1:

return nums[0]

# Case 1: Rob houses from the first house to the second last house

case1 = rob\_linear(nums[:-1])

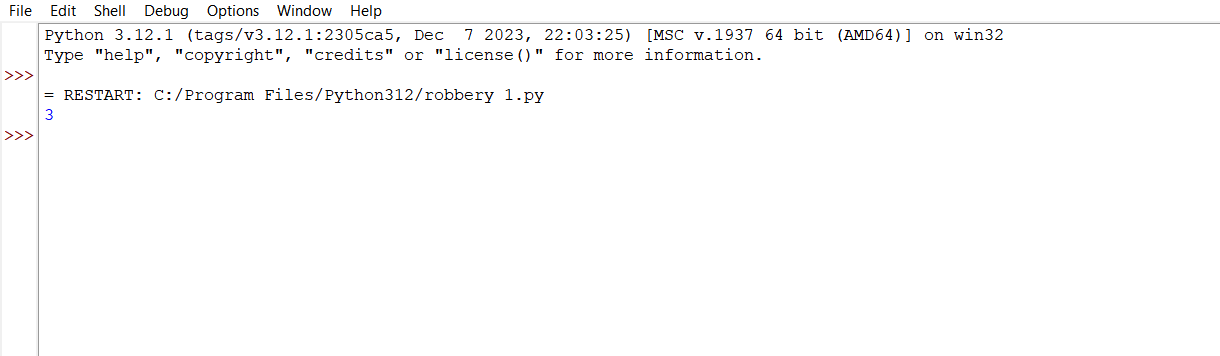
# Case 2: Rob houses from the second house to the last house

case2 = rob\_linear(nums[1:])

return max(case1, case2)

# Test cases

print(rob([2, 3, 2]))



13.

def climbStairs(n):

if n == 1:

return 1

if n == 2:

return 2

dp = [0] \* (n + 1)

dp[1] = 1

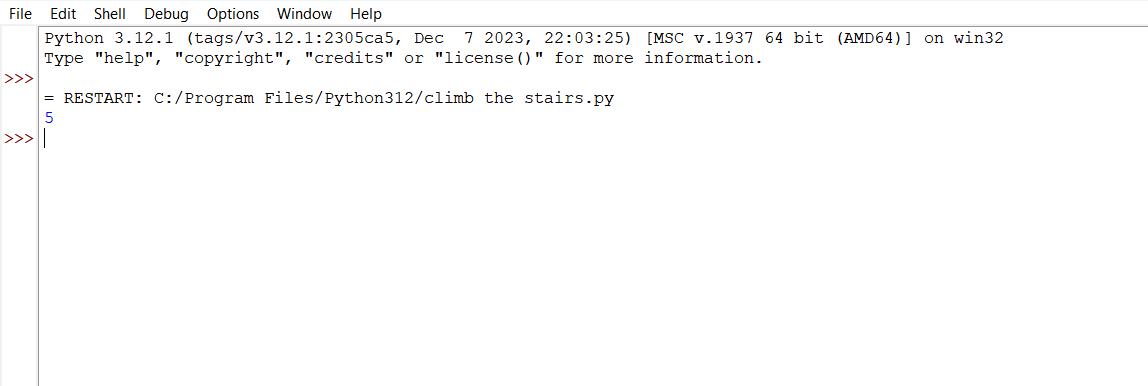
dp[2] = 2

for i in range(3, n + 1):

dp[i] = dp[i - 1] + dp[i - 2]

return dp[n]

print(climbStairs(4))



14.

def uniquePaths(m, n):

dp = [[1] \* n for \_ in range(m)]

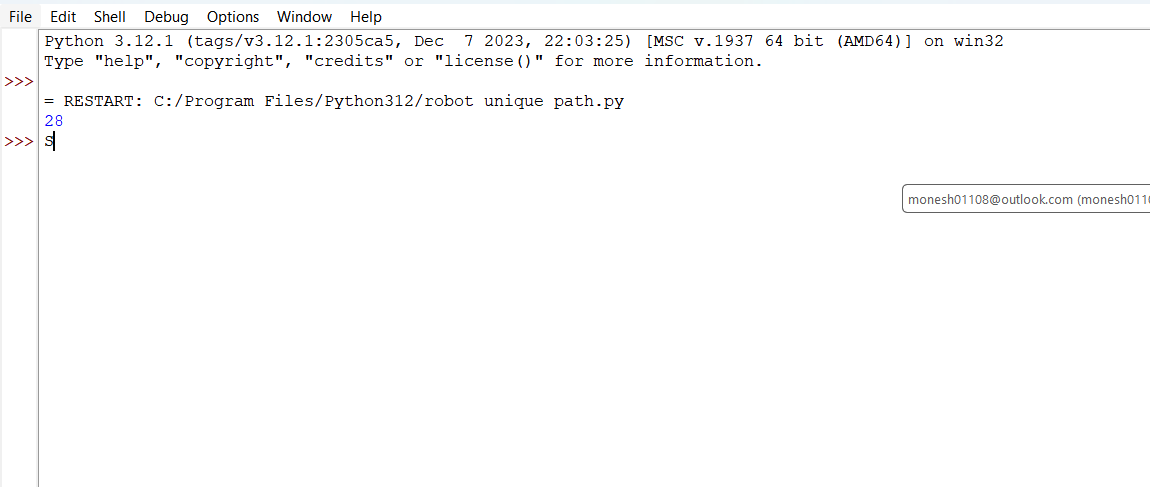
for i in range(1, m):

for j in range(1, n):

dp[i][j] = dp[i - 1][j] + dp[i][j - 1]

return dp[m - 1][n - 1]

print(uniquePaths(7, 3))



15.

def largeGroupPositions(s):

result = []

i = 0

while i < len(s):

start = i

while i < len(s) - 1 and s[i] == s[i + 1]:

i += 1

end = i

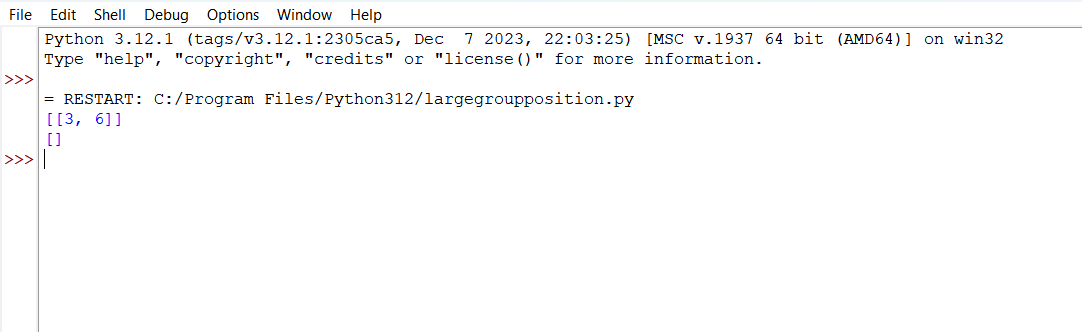
if end - start + 1 >= 3:

result.append([start, end])

i += 1

return result

print(largeGroupPositions("abbxxxxzzy"))



16.

def gameOfLife(board):

m, n = len(board), len(board[0])

# Directions for the 8 neighbors

directions = [

(-1, -1), (-1, 0), (-1, 1),

(0, -1), (0, 1),

(1, -1), (1, 0), (1, 1)

]

# Function to count live neighbors

def count\_live\_neighbors(x, y):

count = 0

for dx, dy in directions:

nx, ny = x + dx, y + dy

if 0 <= nx < m and 0 <= ny < n and abs(board[nx][ny]) == 1:

count += 1

return count

# Iterate through each cell

for i in range(m):

for j in range(n):

live\_neighbors = count\_live\_neighbors(i, j)

# Apply the rules

if board[i][j] == 1 and (live\_neighbors < 2 or live\_neighbors > 3):

board[i][j] = -1 # Mark as dead in the next state

if board[i][j] == 0 and live\_neighbors == 3:

board[i][j] = 2 # Mark as live in the next state

# Update the board to the next state

for i in range(m):

for j in range(n):

if board[i][j] == -1:

board[i][j] = 0

elif board[i][j] == 2:

board[i][j] = 1

# Example usage:

board1 = [

[0, 1, 0],

[0, 0, 1],

[1, 1, 1],

[0, 0, 0]

]

gameOfLife(board1)

print(board1) # Output: [[0, 0, 0], [1, 0, 1], [0, 1, 1], [0, 1, 0]]

board2 = [

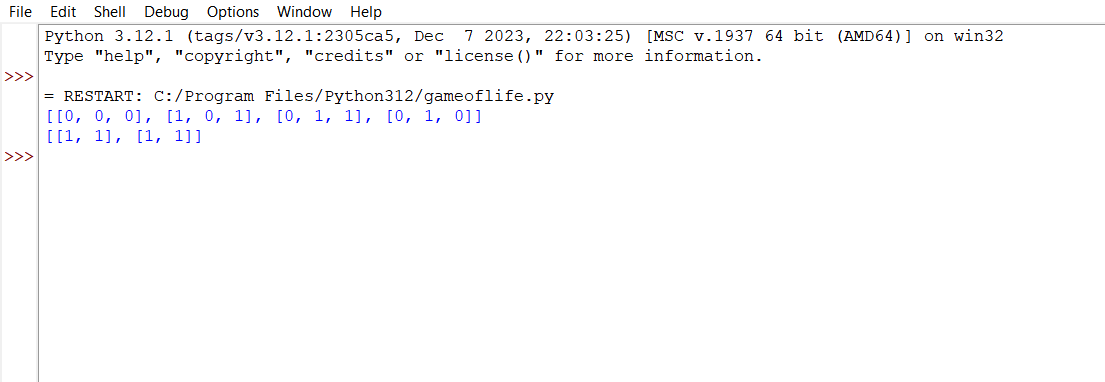
[1, 1],

[1, 0]

]

gameOfLife(board2)

print(board2) # Output: [[1, 1], [1, 1]]



17.

def champagneTower(poured, query\_row, query\_glass):

dp = [[0] \* (i + 1) for i in range(101)]

dp[0][0] = poured

for r in range(query\_row + 1):

for c in range(r + 1):

excess = (dp[r][c] - 1.0) / 2.0

if excess > 0:

dp[r + 1][c] += excess

dp[r + 1][c + 1] += excess

return min(1, dp[query\_row][query\_glass])

print(champagneTower(1, 1, 1))

print(champagneTower(2, 1, 1))

