1. Implement a Python class MaxHeap that supports the following operations: insert, delete, and get\_max. Ensure the operations maintain the properties of a max-heap.

class MaxHeap:

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

self.heap = []

def insert(self, val):

self.heap.append(val)

self.heapify\_up(len(self.heap) - 1)

def delete(self):

if len(self.heap) == 0:

return None

if len(self.heap) == 1:

return self.heap.pop()

max\_val = self.heap[0]

self.heap[0] = self.heap.pop()

self.heapify\_down(0)

return max\_val

def get\_max(self):

if len(self.heap) == 0:

return None

return self.heap[0]

def heapify\_up(self, index):

parent\_index = (index - 1) // 2

if index > 0 and self.heap[index] > self.heap[parent\_index]:

self.heap[index], self.heap[parent\_index] = self.heap[parent\_index], self.heap[index]

self.heapify\_up(parent\_index)

def heapify\_down(self, index):

left\_child\_index = 2 \* index + 1

right\_child\_index = 2 \* index + 2

largest = index

if left\_child\_index < len(self.heap) and self.heap[left\_child\_index] > self.heap[largest]:

largest = left\_child\_index

if right\_child\_index < len(self.heap) and self.heap[right\_child\_index] > self.heap[largest]:

largest = right\_child\_index

if largest != index:

self.heap[index], self.heap[largest] = self.heap[largest], self.heap[index]

self.heapify\_down(largest)

heap = MaxHeap()

heap.insert(10)

heap.insert(20)

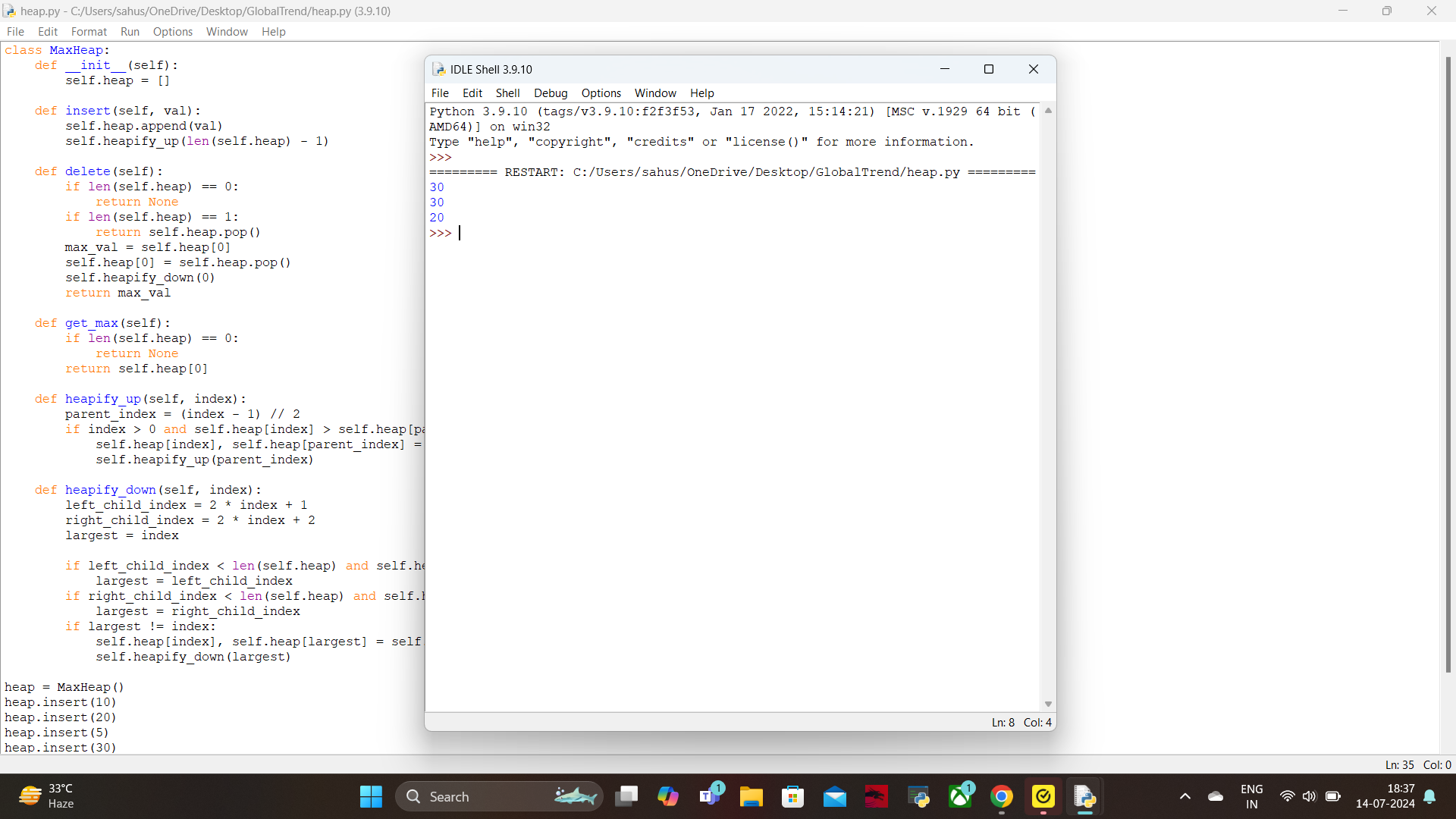
heap.insert(5)

heap.insert(30)

print(heap.get\_max()) #30

print(heap.delete()) #30

print(heap.get\_max()) #20



5. Write a Python function to compute the nth Fibonacci number using recursion.

class Fibonacci:

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

self.n = n;

def get\_number(self, n):

if(n == 0 or n == 1):

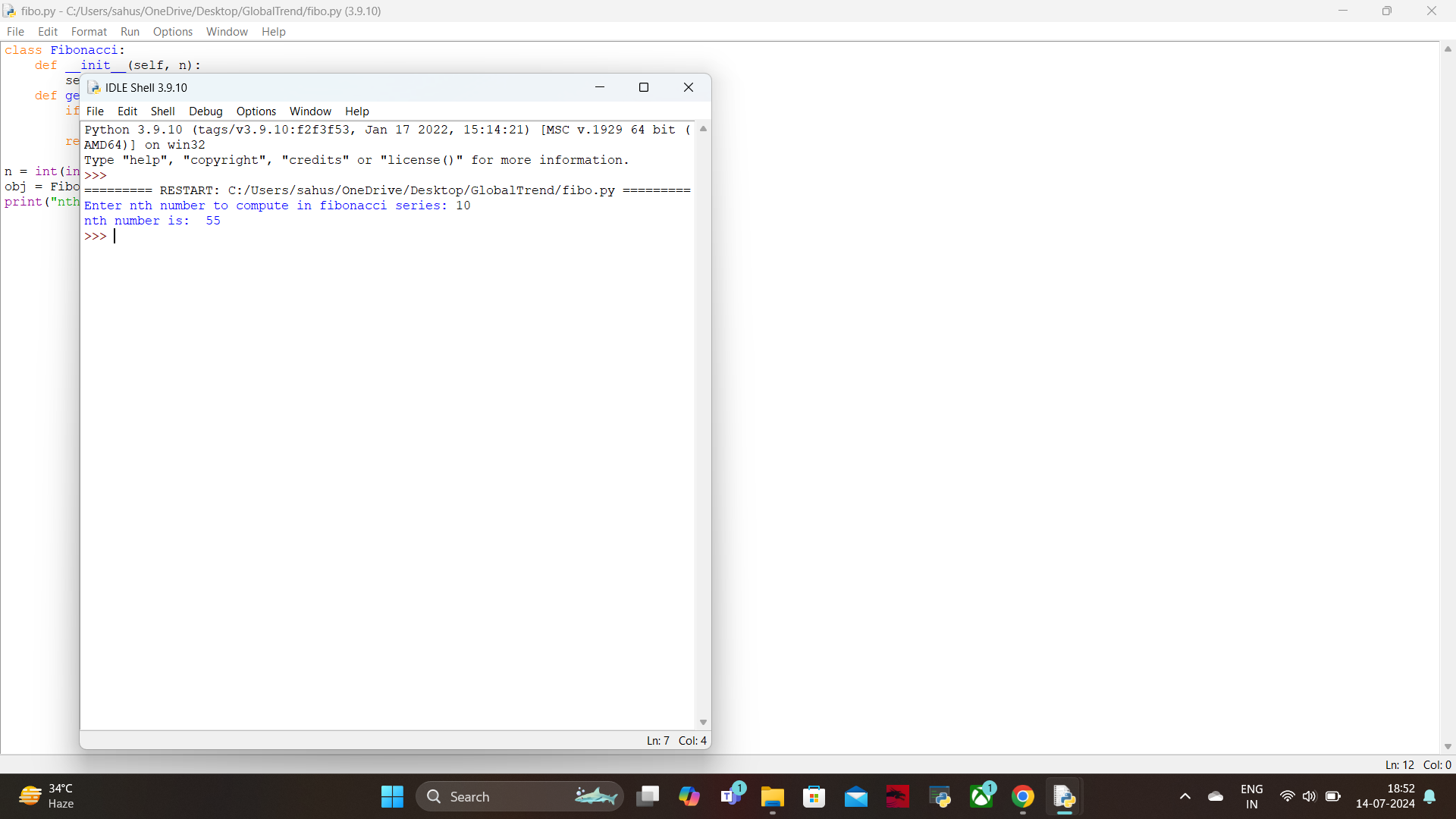
return n

return self.get\_number(n - 1) + self.get\_number(n - 2)

n = int(input("Enter nth number to compute in fibonacci series: "))

obj = Fibonacci(n)

print("nth number is: ", obj.get\_number(n))



6. Write a Python function that divides two numbers and handles the case where the divisor is zero by returning a custom error message.

class ErrorHandling:

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

pass

def safe\_divide(self, dividend, divisor):

try:

result = dividend / divisor

except ZeroDivisionError:

return "Error: Division by zero is not allowed."

return result

dividend = 10

divisor = 0

obj = ErrorHandling()

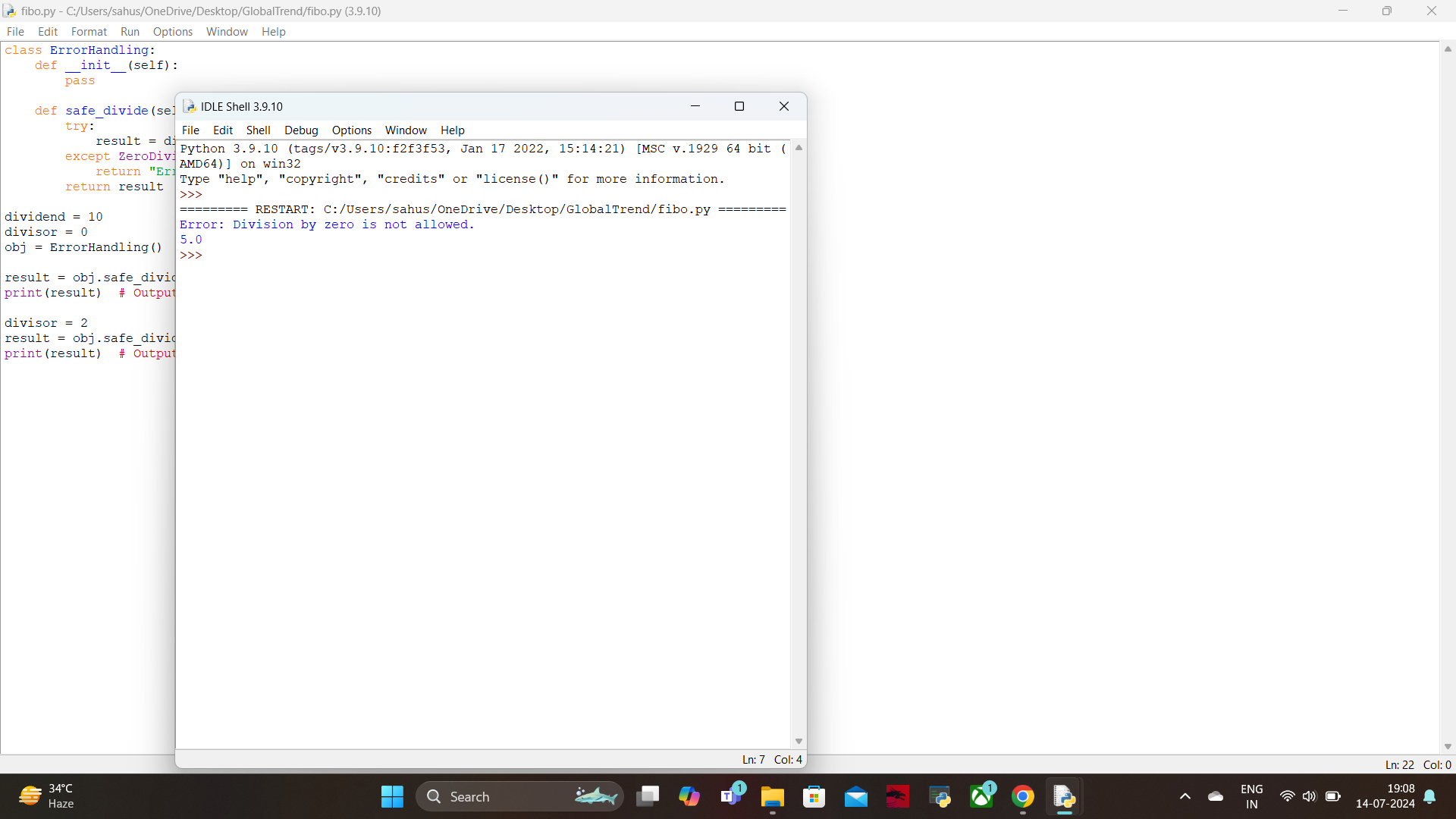
result = obj.safe\_divide(dividend, divisor)

print(result) # Output: Error: Division by zero is not allowed.

divisor = 2

result = obj.safe\_divide(dividend, divisor)

print(result) # Output: 5.0



7. Write a Python decorator that measures the execution time of a function and logs it. Apply this decorator to a function that performs a computationally expensive task.

import time

import logging

logging.basicConfig(level=logging.INFO, format='%(asctime)s - %(message)s')

def time\_logger(func):

def wrapper(\*args, \*\*kwargs):

start\_time = time.time()

result = func(\*args, \*\*kwargs)

end\_time = time.time()

execution\_time = end\_time - start\_time

logging.info(f"Function '{func.\_\_name\_\_}' executed in {execution\_time:.4f} seconds")

return result

return wrapper

@time\_logger

def expensive\_computation(n):

total = 0

for i in range(n):

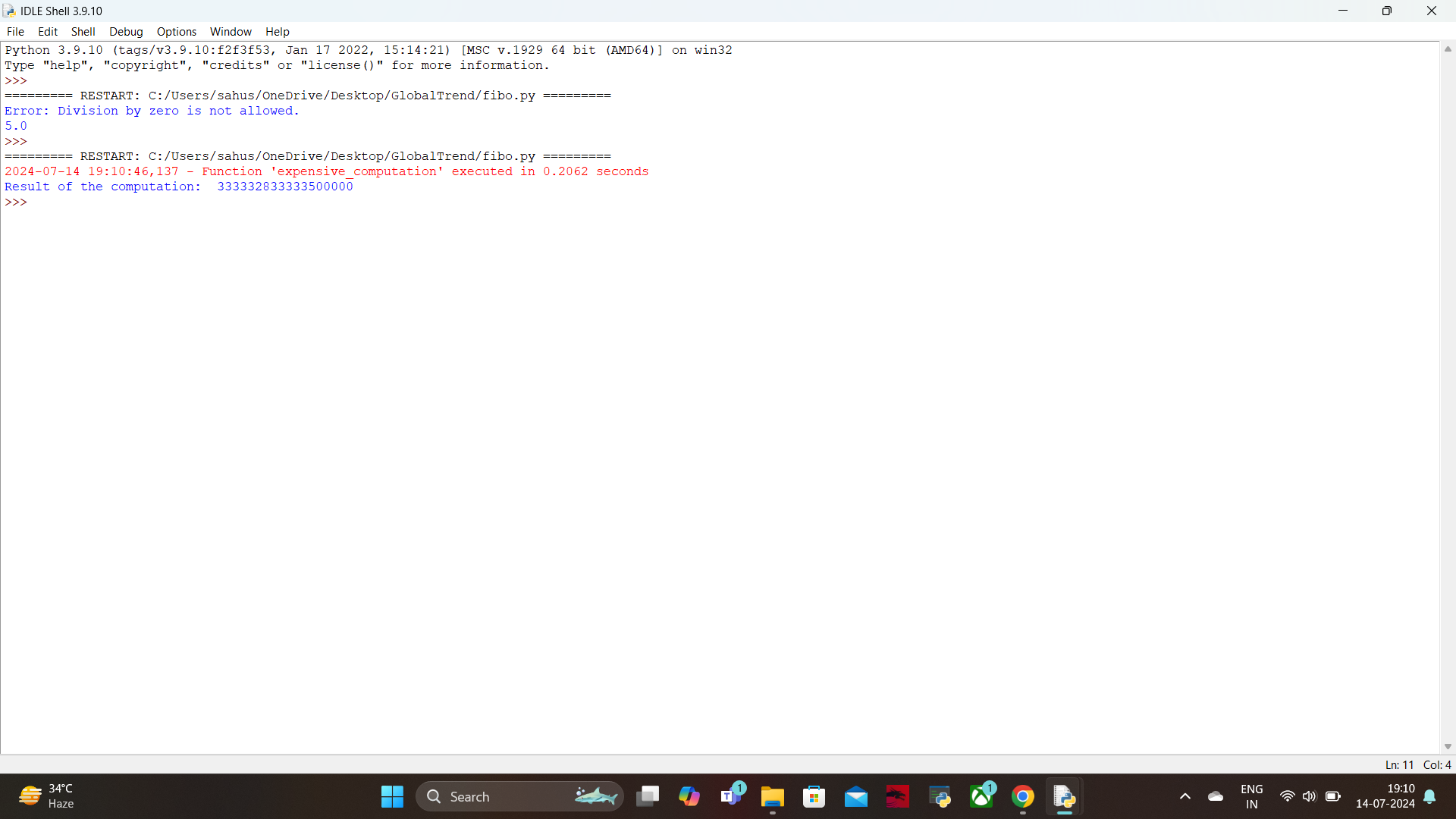
total += i \*\* 2

return total

n = 10\*\*6

result = expensive\_computation(n)

print("Result of the computation: ",result)



8. Write a Python function that takes two numbers and an operator (as a string) and performs the corresponding arithmetic operation (addition, subtraction, multiplication, or division).

def arithmetic\_operation(num1, num2, operator):

if operator == '+':

return num1 + num2

elif operator == '-':

return num1 - num2

elif operator == '\*':

return num1 \* num2

elif operator == '/':

try:

return num1 / num2

except ZeroDivisionError:

return "Error: Division by zero is not allowed."

else:

return "Error: Invalid operator. Please use one of +, -, \*, /."

num1 = 10

num2 = 5

result = arithmetic\_operation(num1, num2, '+')

print(f"Result of {num1} + {num2} = {result}")

result = arithmetic\_operation(num1, num2, '-')

print(f"Result of {num1} - {num2} = {result}")

result = arithmetic\_operation(num1, num2, '\*')

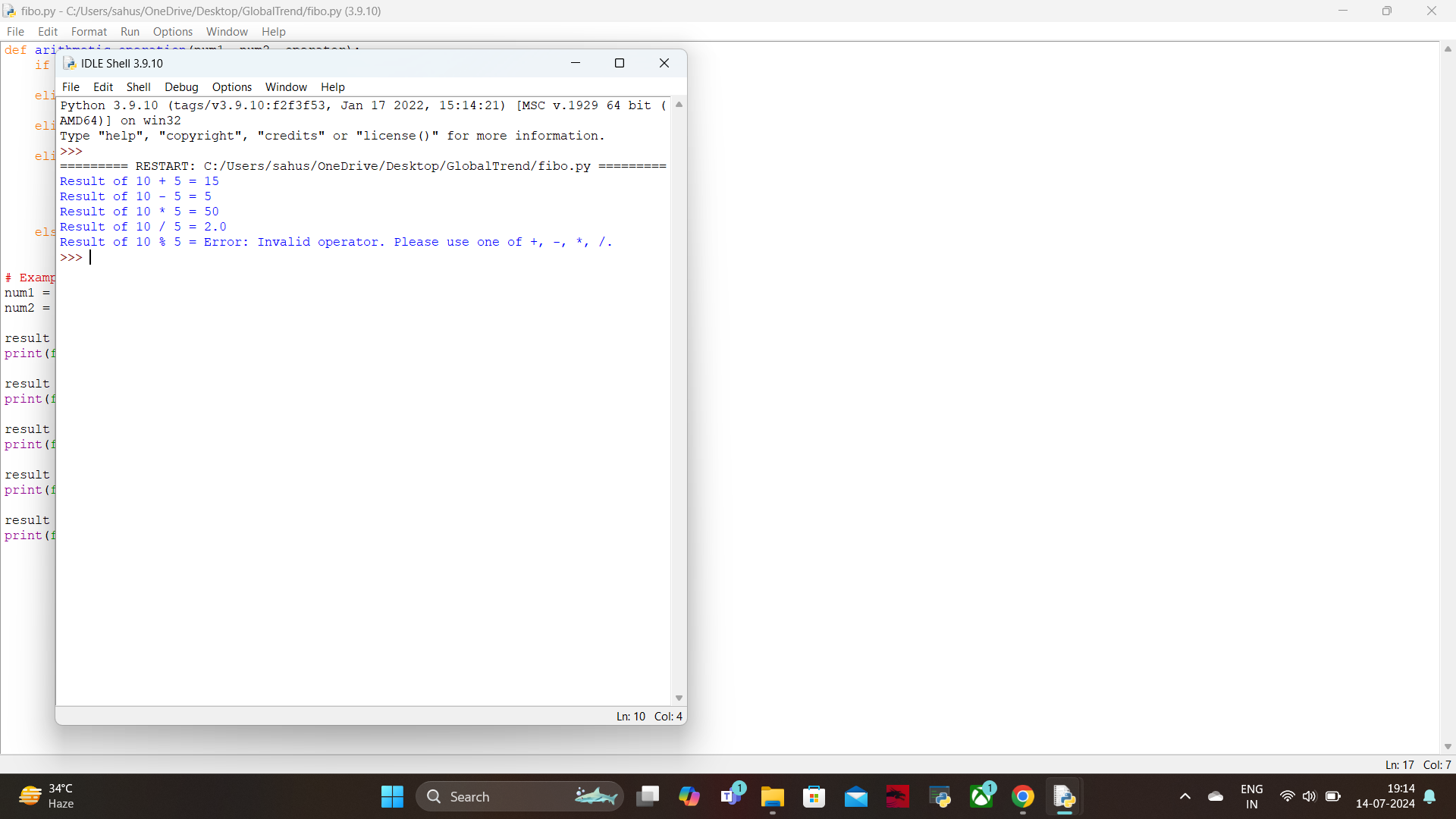
print(f"Result of {num1} \* {num2} = {result}")

result = arithmetic\_operation(num1, num2, '/')

print(f"Result of {num1} / {num2} = {result}")

result = arithmetic\_operation(num1, num2, '%')

print(f"Result of {num1} % {num2} = {result}")



9. Write a Python function that generates a random password. The password should contain a mix of uppercase letters, lowercase letters, digits, and special characters.

import random

import string

def generate\_random\_password(length):

if length < 4:

return "Error: Password length should be at least 4 characters."

uppercase\_letters = string.ascii\_uppercase

lowercase\_letters = string.ascii\_lowercase

digits = string.digits

special\_characters = string.punctuation

# Ensure the password contains at least one of each type of character

password = [

random.choice(uppercase\_letters),

random.choice(lowercase\_letters),

random.choice(digits),

random.choice(special\_characters)

]

# Fill the rest of the password length with a mix of all characters

all\_characters = uppercase\_letters + lowercase\_letters + digits + special\_characters

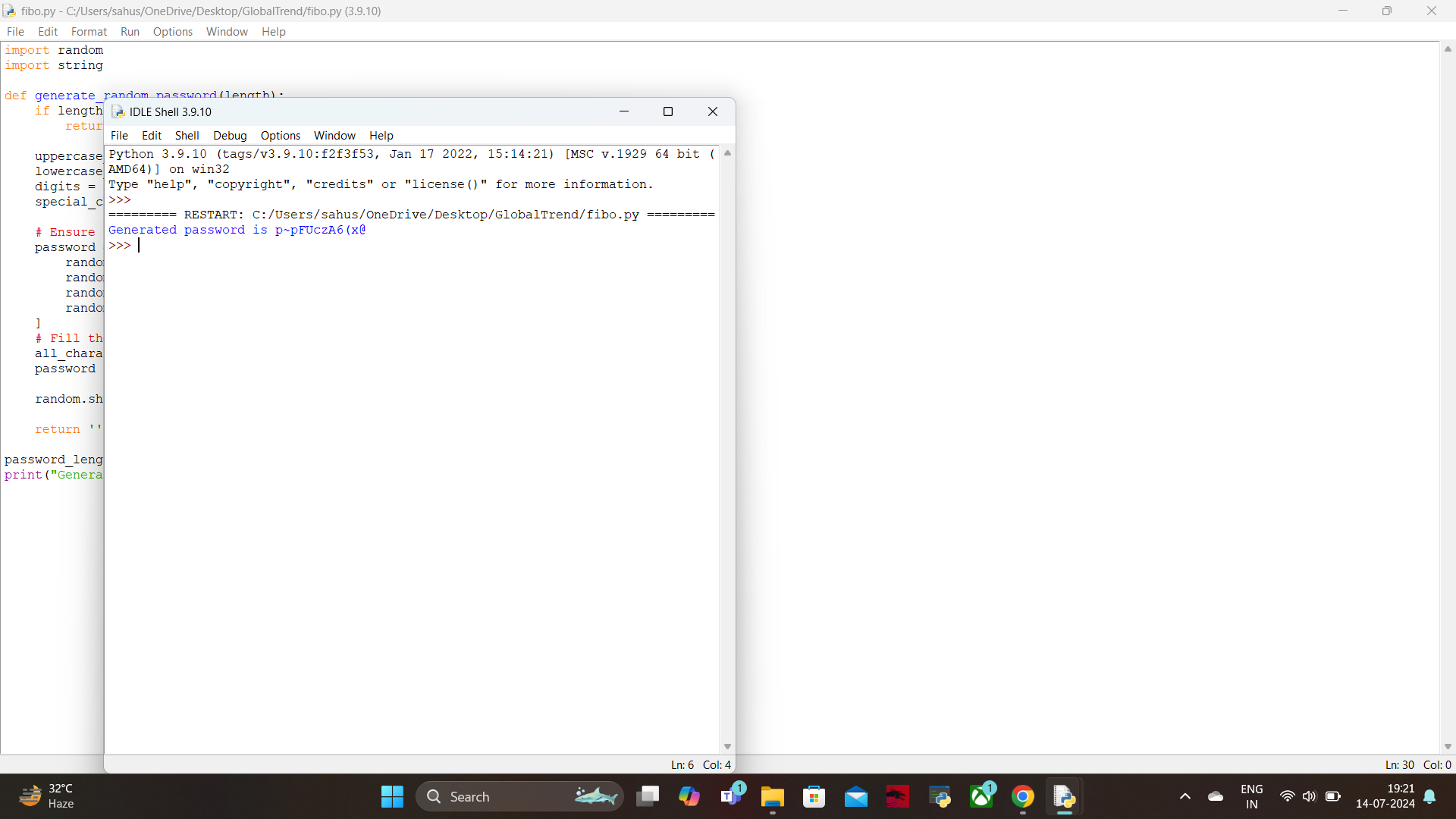
password += random.choices(all\_characters, k=length - 4)

random.shuffle(password)

return ''.join(password)

password\_length = 12

print("Generated password is", generate\_random\_password(password\_length))



10. Write a Python function that takes a 2D list (matrix) and returns its transpose.

def transpose\_matrix(matrix):

rows = len(matrix)

cols = len(matrix[0])

transposed = [[None] \* rows for \_ in range(cols)]

for r in range(rows):

for c in range(cols):

transposed[c][r] = matrix[r][c]

return transposed

matrix = [

[1, 2, 3],

[4, 5, 6],

[7, 8, 9]

]

transposed\_matrix = transpose\_matrix(matrix)

print("Original matrix:")

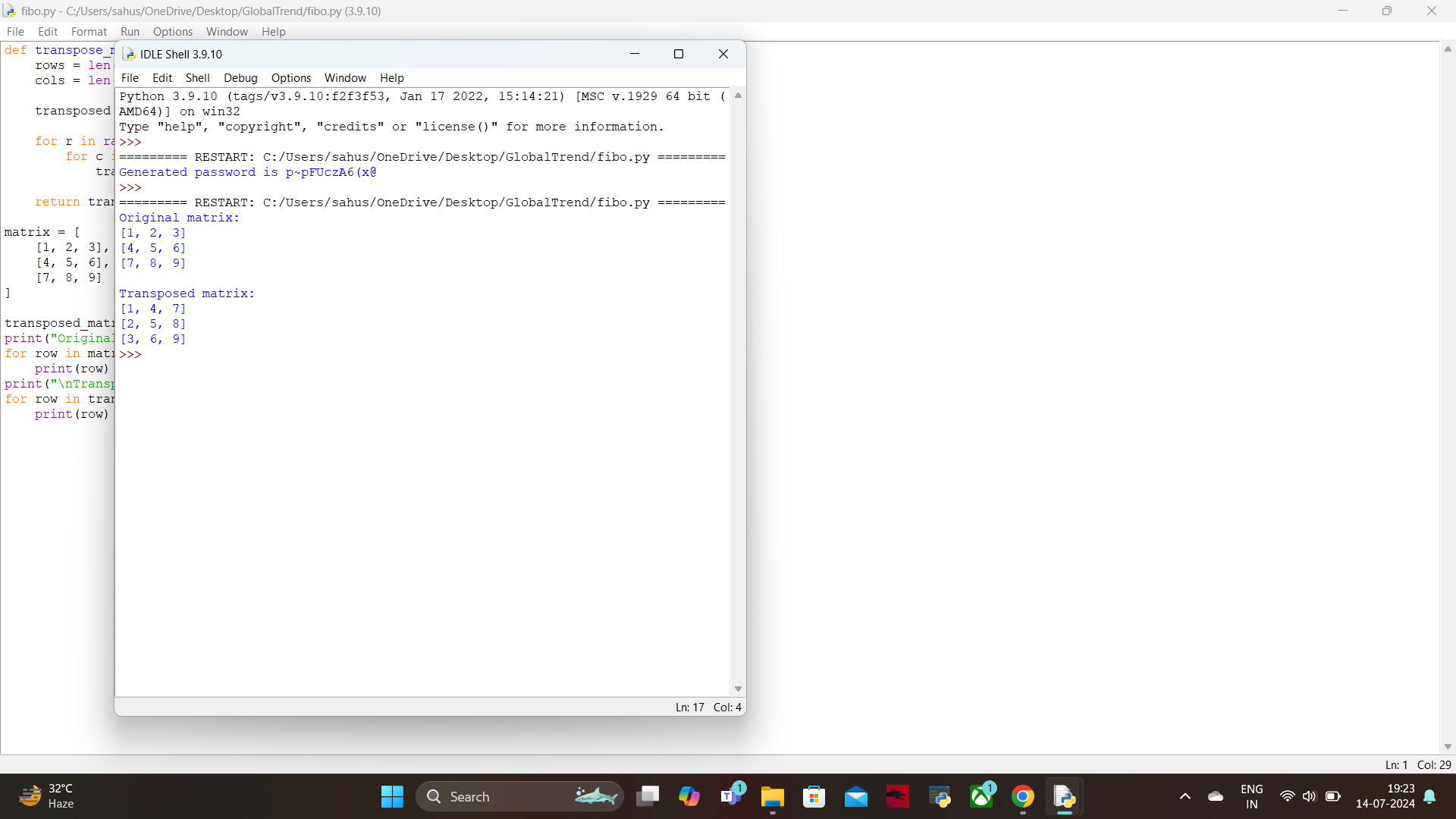
for row in matrix:

print(row)

print("\nTransposed matrix:")

for row in transposed\_matrix:

print(row)



1. Write a Python function that takes a list of URLs, attempts to download their content, and retries up to 3 times if an error occurs. Use appropriate error handling to manage different types of exceptions.

import requests

from requests.exceptions import RequestException

import time

def download\_urls(urls):

max\_retries = 3

results = {}

for url in urls:

attempt = 0

while attempt < max\_retries:

try:

response = requests.get(url)

response.raise\_for\_status() # Raise an HTTPError for bad responses

results[url] = response.content

print("Downloaded {} successfully!".format(url))

break

except RequestException as e:

print("Error downloading {}: {}".format(url, e))

attempt += 1

if attempt < max\_retries:

print("Retrying ({}/{}) after 1 second...".format(attempt + 1, max\_retries))

time.sleep(1)

else:

results[url] = None

print("Reached maximum retries for {}. Skipping.".format(url))

return results

urls = [

"https://www.example.com",

"https://www.nonexistentwebsite12345.com",

"https://www.python.org"

]

results = download\_urls(urls)

for url, content in results.items():

if content:

print("Content from {}: {}...".format(url, content[:100]))

else:

print("Failed to download content from {}".format(url))

