Q1:

Consider the 8 queen's problem, it is a 8\*8 chess board where you need to place queens

according to the following constraints.

a. Each row should have exactly only one queen.

b. Each column should have exactly only one queen.

c. No queens are attacking each other.

CODE:

import numpy as np

def is\_safe(board, row, col, n):

    #row

    for i in range(row):

        if board[i][col] == 1:

            return False

    #col

    for i in range(col):

        if board[row][i] == 1:

            return False

    #left diagonal

    for i, j in zip(range(row, -1, -1), range(col, -1, -1)):

        if board[i][j] == 1:

            return False

    # right diagonal

    for i, j in zip(range(row, -1, -1), range(col, n)):

        if board[i][j] == 1:

            return False

    return True

def nqueen(board, row, n, ans):

    if row == n:

        ans.append(board.copy())

        return

    for col in range(n):

        if is\_safe(board, row, col, n):

            board[row][col] = 1

            nqueen(board, row + 1, n, ans)

            board[row][col] = 0

n = int(input("Enter the size of the board (n): "))

board = np.zeros((n, n), dtype=int)

ans = []

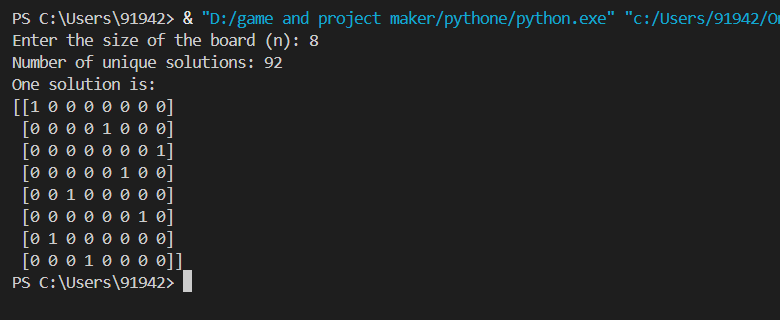
nqueen(board, 0, n, ans)

print(f"Number of unique solutions: {len(ans)}")

print(f"One solution is: ")

print(np.array(ans[0]))

OUTPUT:



Q3:

A magic square is an N×N grid of numbers in which the entries in each row, column and

main diagonal sum to the same number (equal to N(N^2+1)/2). Create a magic square for

N=4, 5, 6, 7, 8

CODE:

import numpy as np

def generate\_magic\_square(n):

    if n < 3 or n % 2 == 0:

        raise ValueError("Magic square is only possible for odd integers greater than or equal to 3.")

    magic\_square = np.zeros((n, n), dtype=int)

    i, j = 0, n // 2

    for num in range(1, n \* n + 1):

        magic\_square[i, j] = num

        new\_i, new\_j = (i - 1) % n, (j + 1) % n

        if magic\_square[new\_i, new\_j] != 0:  # If the cell is already filled

            i += 1

        else:

            i, j = new\_i, new\_j

    return magic\_square

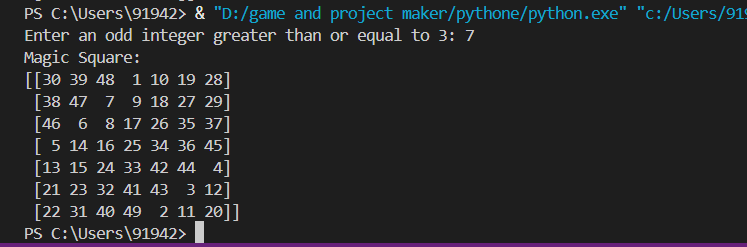
n = int(input("Enter an odd integer greater than or equal to 3: "))

magic\_square = generate\_magic\_square(n)

print("Magic Square:")

print(magic\_square)

OUTPUT:



Q4:

Take N (N >= 10) random 2-dimensional points represented in cartesian coordinate space.

Store them in a numpy array. Convert them to polar coordinates.

CODE:

import numpy as np

import random

table = np.random.rand(10,2)\*10

r=np.sqrt(table[:,0]\*\*2+table[:,1]\*\*2)

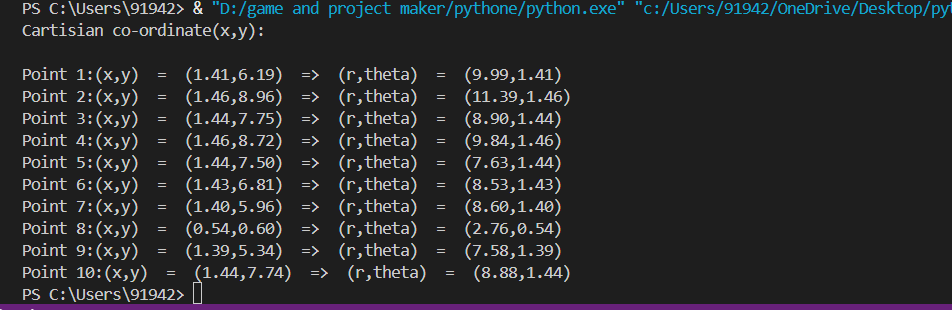
arg = np.arctan(table[:,1],table[:,0])

print("Cartisian co-ordinate(x,y):\n")

for i in range (len(table)):

    print(f"Point {i+1}:(x,y)  =  ({table[i][0]:.2f},{table[i][1]:.2f})  =>  (r,theta)  =  ({r[i]:.2f},{arg[i]:.2f})")

OUTPUT:



Q5:

Write a program to make the length of each element 15 of a given Numpy array and the

string centred, left-justified, right-justified with paddings of \_ (underscore).

CODE:

import numpy as np

def lmodi(word):

    return word.ljust(15, '\_')

def rmodi(word):

    return word.rjust(15, '\_')

def cmodi(word):

    return word.center(15, '\_')

n = int(input("Enter size of array: "))

a=[]

for j in range(n):

    a.append(input(f"Enter element {j + 1}: "))

print("Original list:", a)

arr = np.array(a, dtype=str)

arr\_l = np.array([lmodi(w) for w in arr])

arr\_r = np.array([rmodi(w) for w in arr])

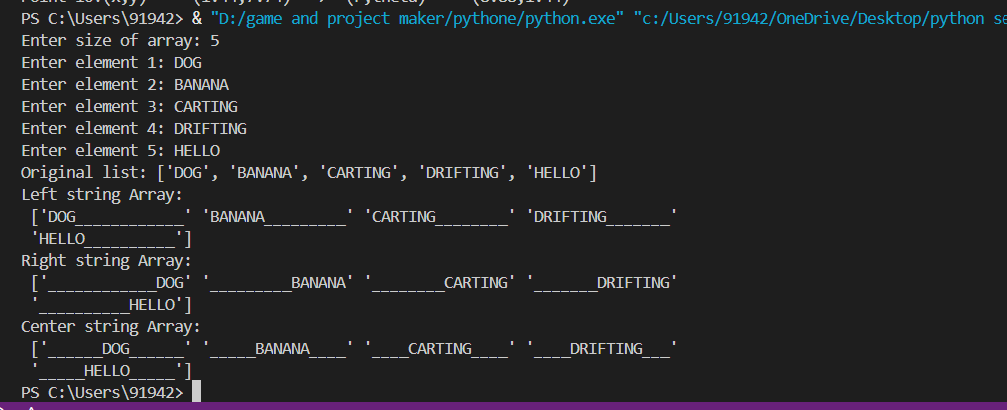
arr\_c = np.array([cmodi(w) for w in arr])

print("Left string Array:\n", arr\_l)

print("Right string Array:\n", arr\_r)

print("Center string Array:\n", arr\_c)

OUTPUT:



Q6:

The bisection method is a technique for finding solutions (roots) to equations with a single

unknown variable. Given a polynomial function f, try to find an initial interval off by

random probe. Store all the updates in an Numpy array. Plot the root finding process using

the matplotlib/pyplot library.

CODE:

import numpy as np

import matplotlib.pyplot as plt

def f(x):

    return x\*\*7 - 6\*x\*\*2 + 4\*x + 12

def bisection\_with\_plot(func, max\_attempts=100, tolerance=1e-6):

    found\_interval = False

    a, b = None, None

    attempts = 0

    while not found\_interval and attempts < max\_attempts:

        a = np.random.uniform(-10, 10)

        b = np.random.uniform(-10, 10)

        if func(a) \* func(b) < 0:

            found\_interval = True

        attempts += 1

    if not found\_interval:

        print("Failed to find a valid initial interval.")

        return None, None

    updates = []  # Store midpoint values

    while abs(b - a) > tolerance:

        c = (a + b) / 2

        updates.append(c)  # Store update

        if func(c) == 0:  # Exact root

            break

        elif func(c) \* func(a) < 0:  # Root in [a, c]

            b = c

        else:  # Root in [c, b]

            a = c

    return a, b, np.array(updates)

a, b, updates = bisection\_with\_plot(f)

if updates is not None:

    print(f"Approximate root: {(a + b) / 2}")

    x\_vals = np.linspace(-10, 10, 1000)

    y\_vals = f(x\_vals)

    plt.plot(x\_vals, y\_vals, label="f(x)")

    plt.scatter(updates, f(updates), color="red", label="Midpoint updates")

    plt.axhline(0, color="black", linestyle="--", linewidth=0.5)

    plt.xlabel("x")

    plt.ylabel("f(x)")

    plt.title("Bisection Method - Root Finding Process")

    plt.legend()

    plt.grid()

    plt.show()

OUTPUT:

