SEM-2 > Assignment 10 > 10_1_np_8queens.py > ...

```python
""" 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.

Write a program to place the queens randomly in the chess board so that all the conditions are satisfied.
Find the solutions to the problem. """

import numpy as np

def checkvalidity(perm):
    for i in range(len(perm)):
        for j in range(i + 1, len(perm)):
            if abs(perm[i] - perm[j]) == abs(i - j):
                return False
    return True

def random_solution():
    while True:
        perm = np.random.permutation(8)
        if checkvalidity(perm):
            return perm

def displayboard(perm):
    board = np.zeros((8, 8), dtype=int)
    for row, col in enumerate(perm):
        board[row, col] = 1  # 1 represents a queen
    return board

res = random_solution()
board = displayboard(res)

print(f"Random valid queen positions (row index = row, value = column) : {res}\n")
print("\nBoard representation (1 = queen) :-- \n", board)
```

```
Random valid queen positions (row index = row, value = column) : [5 3 0 4 7 1 6 2]


Board representation (1 = queen) :--
 [[0 0 0 0 0 1 0 0]
 [0 0 0 1 0 0 0 0]
 [1 0 0 0 0 0 0 0]
 [0 0 0 0 1 0 0 0]
 [0 0 0 0 0 0 0 1]
 [0 1 0 0 0 0 0 0]
 [0 0 0 0 0 0 1 0]
 [0 0 1 0 0 0 0 0]]
PS C:\Users\shuvr\OneDrive\Documents\CODING\PYTHON CODES\College Py Codes>
```

SEM-2 > Assignment 10 > 10_3_np_magic_square.py > ms_even

```python
""" A magic square is an N x 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 """

import numpy as np

def ms_odd(n):
    magic = np.zeros((n, n), dtype=int)
    i, j = 0, n // 2
    for num in range(1, n * n + 1):
        magic[i, j] = num
        i2, j2 = (i - 1) % n, (j + 1) % n
        if magic[i2, j2]:
            i = (i + 1) % n
        else:
            i, j = i2, j2
    return magic

def ms_mod4(n):
    magic = np.arange(1, n * n + 1).reshape(n, n)
    idx = np.full((n, n), False)
    for i in range(n):
        for j in range(n):
            if (i % 4 == j % 4) or ((i % 4 + j % 4) == 3):
                idx[i, j] = True
    magic[idx] = (n * n + 1) - magic[idx]
    return magic

```

```python
def ms_even(n):
    half_n = n // 2
    sub_square = ms_odd(half_n)
    magic = np.zeros((n, n), dtype=int)
    add = [0, 2, 3, 1]
    for i in range(2):
        for j in range(2):
            magic[i*half_n:(i+1)*half_n, j*half_n:(j+1)*half_n] = \
                sub_square + (add[i*2 + j] * (half_n**2))
    k = (n - 2) // 4
    for i in range(half_n):
        for j in range(n):
            if j < k or j >= n - k or (j == k and i == k):
                if j < half_n:
                    magic[i, j], magic[i + half_n, j] = magic[i + half_n, j], magic[i, j]
    return magic

def ms_generator(n):
    if n % 2 == 1:
        return ms_odd(n)
    elif n % 4 == 0:
        return ms_mod4(n)
    else:
        return ms_even(n)

for N in range(4, 9):
    print(f"\nMagic Square (N={N}) :--")
    square = ms_generator(N)
    print(square)
    print(f"Magic constant = {N * (N**2 + 1) // 2}")
```

```
Magic Square (N=4) :--
[[16  2  3 13]
 [ 5 11 10  8]
 [ 9  7  6 12]
 [ 4 14 15  1]]
Magic constant = 34

Magic Square (N=5) :--
[[17 24  1  8 15]
 [23  5  7 14 16]
 [ 4  6 13 20 22]
 [10 12 19 21  3]
 [11 18 25  2  9]]
Magic constant = 65

Magic Square (N=6) :--
[[35  1  6 26 19 24]
 [30 32  7 21 23 25]
 [31  9  2 22 27 20]
 [ 8 28 33 17 10 15]
 [ 3  5 34 12 14 16]
 [ 4 36 29 13 18 11]]
Magic constant = 111
```

```
Magic Square (N=7) :--
[[30 39 48  1 10 19 28]
 [38 47  7  9 18 27 29]
 [46  6  8 17 26 35 37]
 [ 5 14 16 25 34 36 45]
 [13 15 24 33 42 44  4]
 [21 23 32 41 43  3 12]
 [22 31 40 49  2 11 20]]
Magic constant = 175


Magic Square (N=8) :--
[[64  2  3 61 60  6  7 57]
 [ 9 55 54 12 13 51 50 16]
 [17 47 46 20 21 43 42 24]
 [40 26 27 37 36 30 31 33]
 [32 34 35 29 28 38 39 25]
 [41 23 22 44 45 19 18 48]
 [49 15 14 52 53 11 10 56]
 [ 8 58 59  5  4 62 63  1]]
Magic constant = 260
```

SEM-2 > Assignment 10 > 10_4_cat_to_pol.py > ...

```python
""" Take N (N >= 10) random 2-dimensional points represented in cartesian coordinate space.
Store them in a numpy array.
Convert them to polar coordinates. """

import numpy as np
N = int(input("Enter N (>=10) : "))
if N>=10:
    cart_pts = np.random.uniform(-10, 10, (N, 2))
    print("Cartesian Coordinates (x, y) :--\n", cart_pts)
    x = cart_pts[:, 0]
    y = cart_pts[:, 1]
    r = np.sqrt(x**2 + y**2) # r
    theta = np.arctan2(y, x) # theta
    pol_coords = np.column_stack((r, theta))
    print("\nPolar Coordinates (r, θ in radians) :--\n", pol_coords)
else:
    print("Wrong input. Run the program again\n")
```

```
Enter N (>=10) : 11
Cartesian Coordinates (x, y) :--
 [[-6.01810718  8.53249612]
 [ 0.91077953  0.76414563]
 [-0.16608449  7.47774442]
 [-6.25024607 -9.11009843]
 [-7.6304408  -5.56690942]
 [ 2.59864323  9.17107569]
 [ 0.62088408 -6.45026939]
 [-7.23366175 -8.07274325]
 [ 8.86664603  2.76541295]
 [-9.49692497 -1.72281711]
 [ 9.63483298 -7.11396185]]

Polar Coordinates (r, θ in radians) :--
 [[10.44131716  2.18508126]
 [ 1.18888094  0.69807427]
 [ 7.4795886   1.59300318]
 [11.04805274 -2.17211802]
 [ 9.44532198 -2.51129814]
 [ 9.53213386  1.29468197]
 [ 6.48008274 -1.4748349 ]
 [10.83951317 -2.30143032]
 [ 9.28789107  0.3023285 ]
 [ 9.65192637 -2.96213635]
 [11.97657964 -0.63600889]]
```

SEM-2 > Assignment 10 > 10_5_np_str_asth.py > ...

```python
""" 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).
Sample for testing : ['apple','bees','orange','peach','banana','hello'] """

import numpy as np

width = 15
pad = "_"
arr = np.array(eval(input(f"Enter a list of strings (size of each string <= {width}) : ")))
centered = np.array([s.center(width, pad) for s in arr])
leftpad = np.array([s.ljust(width, pad) for s in arr])
rightpad = np.array([s.rjust(width, pad) for s in arr])

print("Original strings :--\n", arr)
print("\nCentered :--\n", centered)
print("\nLeft-justified :--\n", leftpad)
print("\nRight-justified :--\n", rightpad)
```

```
Enter a list of strings (size of each string <= 15) : ['apple','bees','orange','peach','banana','hello']
Original strings :--
 ['apple' 'bees' 'orange' 'peach' 'banana' 'hello']

Centered :--
 ['_____apple_____' '______bees_____' '_____orange____' '_____peach_____'
 '_____banana____' '_____hello_____']

Left-justified :--
 ['apple__________' 'bees___________' 'orange_________' 'peach__________'
 'banana_________' 'hello__________']

Right-justified :--
 ['__________apple' '___________bees' '_________orange' '__________peach'
 '_________banana' '__________hello']
```

SEM-2 > Assignment 10 > 10_6_np_mpl_bisection.py > ...

```python
""" 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. """

import numpy as np
import matplotlib.pyplot as plt

def f(x):
    return x**3 + 2*x**2 - 4

np.random.seed(0)
found = False
while not found:
    a, b = np.random.uniform(-10, 10, 2)
    if f(a) * f(b) < 0:
        found = True
        if a > b:
            a, b = b, a
print(f"Initial interval  : a = {a:.4f}, b = {b:.4f}")

tolerance = 1e-6
max_iter = 100
updates = []
for i in range(max_iter):
    mid = (a + b) / 2.0
    updates.append(mid)
    if abs(f(mid)) < tolerance or (b - a) / 2 < tolerance:
        break
    if f(a) * f(mid) < 0:
        b = mid
    else:
        a = mid
updates = np.array(updates)
plt.figure(figsize=(10, 5))
x_vals = np.linspace(updates[0] - 1, updates[0] + 1, 400)
plt.plot(x_vals, f(x_vals), label="f(x)")
plt.axhline(0, color='gray', linestyle='--')
plt.plot(updates, f(updates), 'ro--', label="Bisection updates")
plt.title("Bisection Method Convergence")
plt.xlabel("x")
plt.ylabel("f(x)")
plt.legend()
plt.grid(True)
plt.show()
print(f"\nApproximate root after {len(updates)} iterations : x = {updates[-1]:.6f}")
```

```
Initial interval  : a = 0.9763, b = 4.3038
```