Shri Vile Parle Kelavani Mandal's

DWARKADAS J. SANGHVI COLLEGE OF ENGINEERING

(Autonomous College Affiliated to the University of Mumbai)

NAAC Accredited with "A" Grade (CGPA : 3.18)

Department of Computer Science and Engineering (Data Science)

Subject: Artificial Intelligence (DJ19DSC502)

AY: 2023-24

Experiment 1

(Problem Solving)

D 12

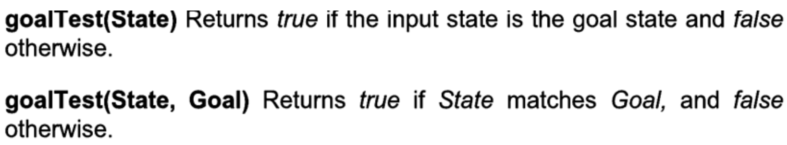
60009210105 **Amitesh Sawarkar**

Aim: Implement domain specific functions for given problems required for problem solving.

Theory:

There are two domain specific functions required in all problem solving methods.

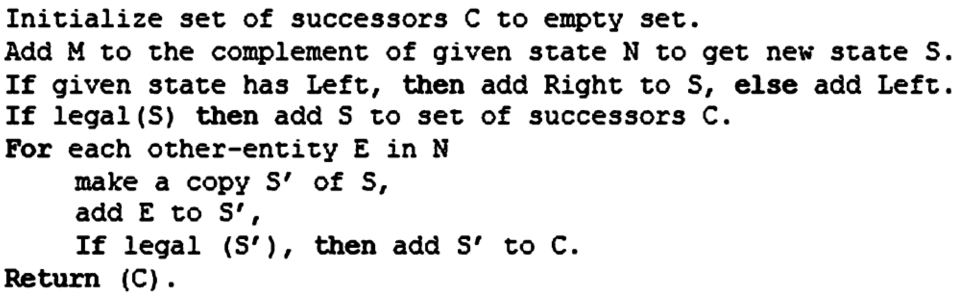
1. GoalTest Function:



goalTest(State) Returns *true* if the input state is the goal state and *false* otherwise.

goalTest(State, Goal) Returns *true* if *State* matches *Goal*, and *false* otherwise.

2. MoveGen function:



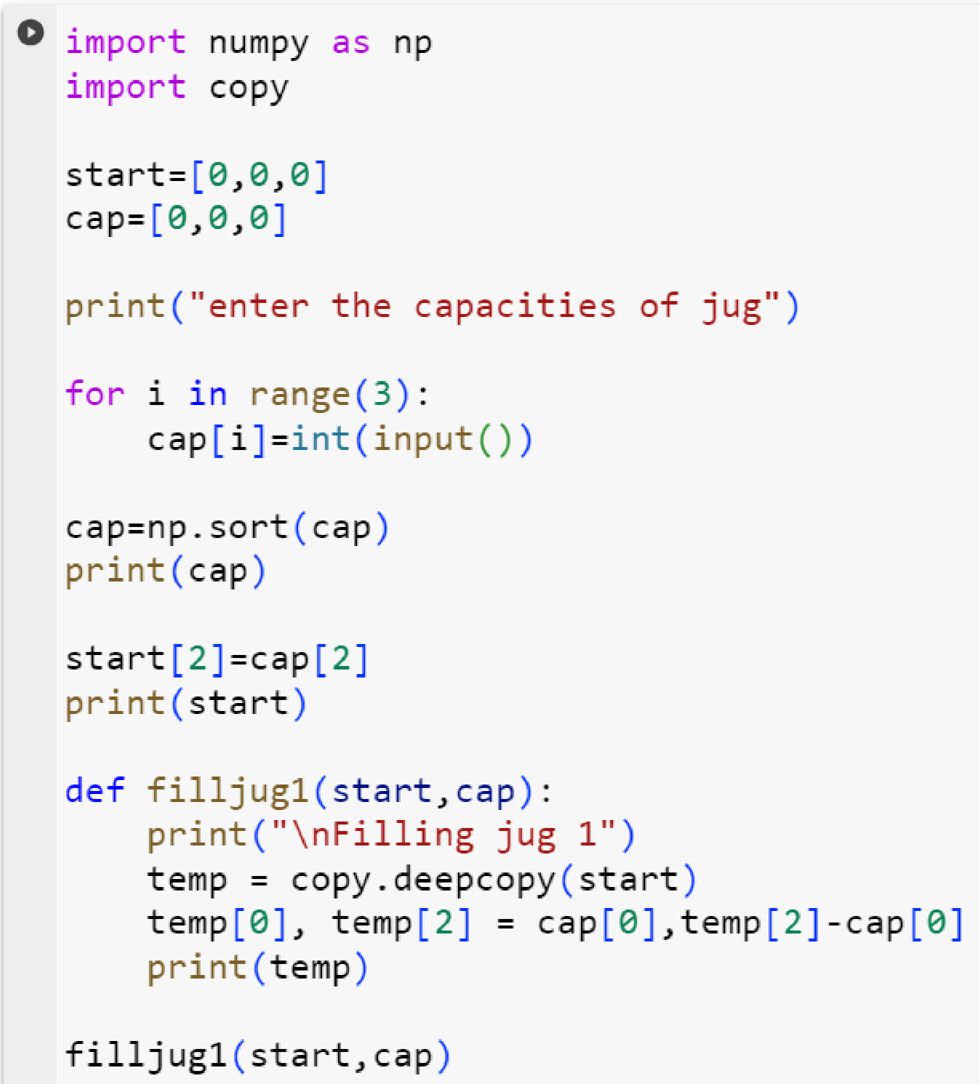
```
Initialize set of successors C to empty set.
Add M to the complement of given state N to get new state S.
If given state has Left, then add Right to S, else add Left.
If legal(S) then add S to set of successors C.
For each other-entity E in N
    make a copy S' of S,
    add E to S',
    If legal (S'), then add S' to C.
Return (C).
```

Lab Assignment to do:

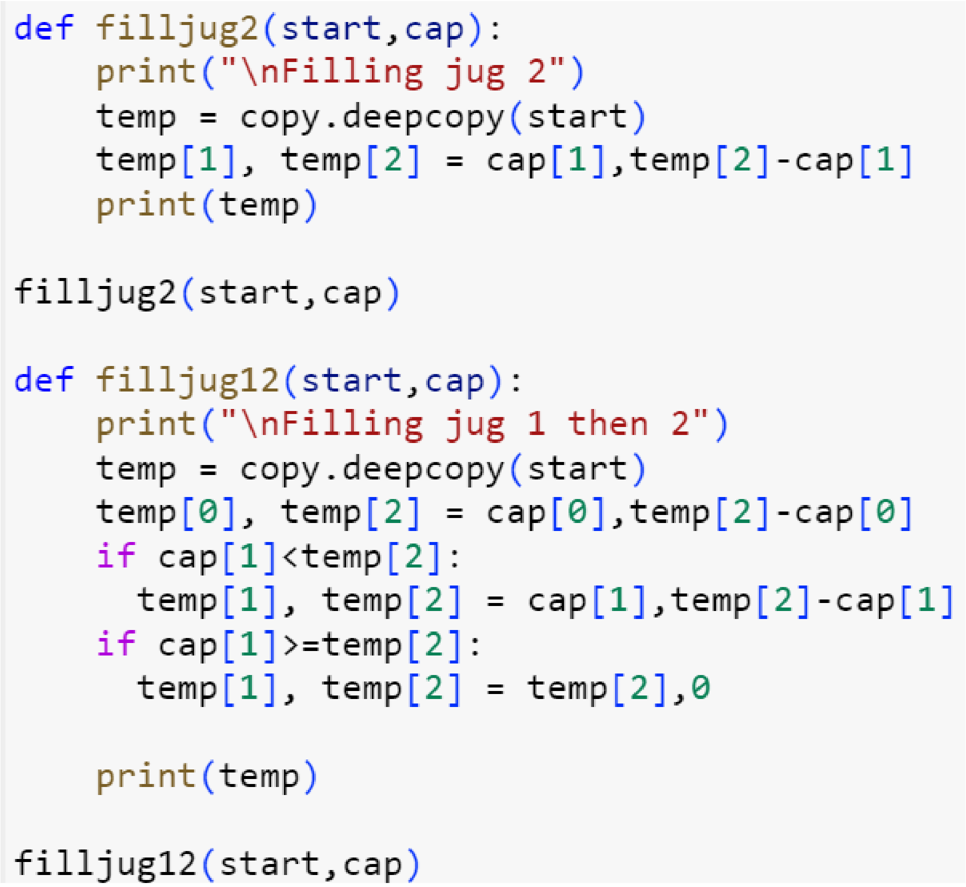
Create MoveGen and GoalTest Functions for the given problems

1. **Water Jug Problem**

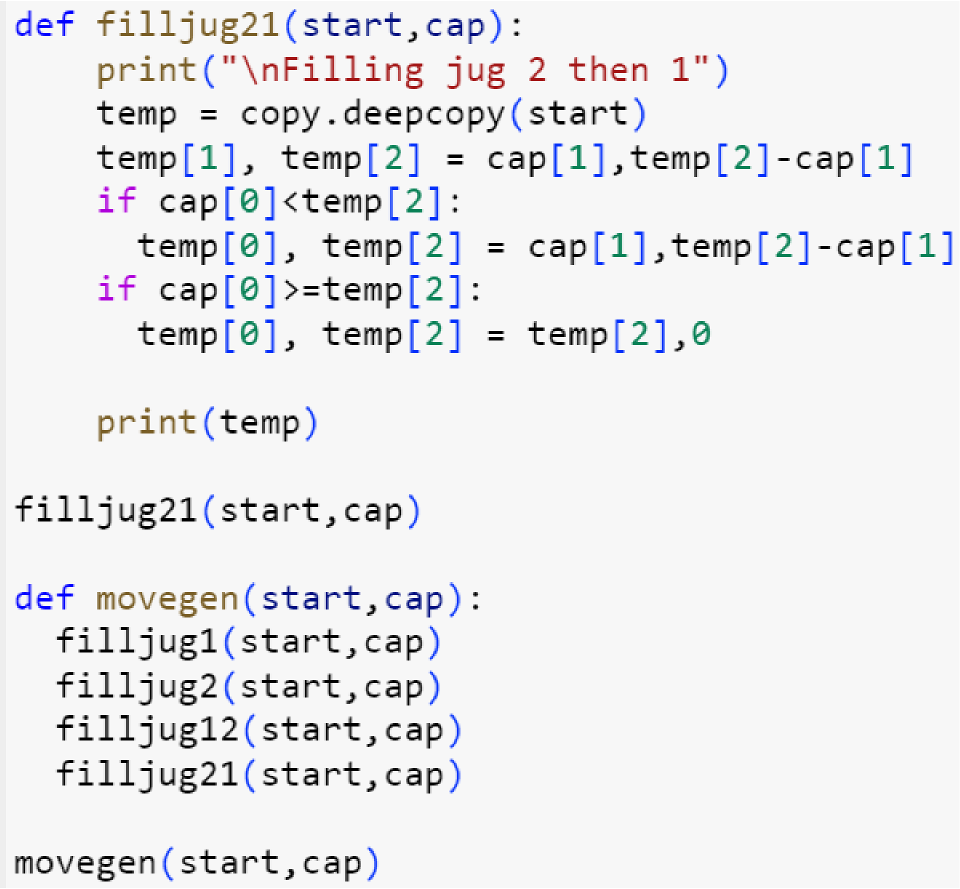
There are two jugs available of different volumes such as a 3 litres and a 7 litres and you have to measure a different volume such as 6 litre.



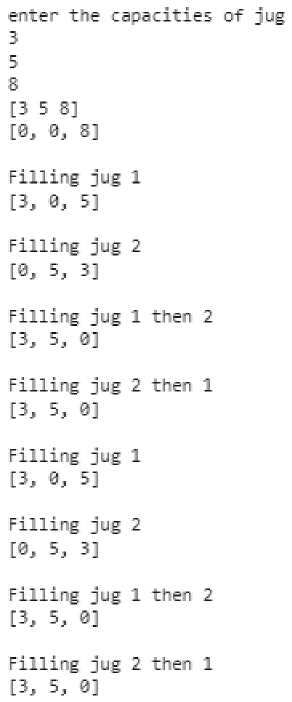
```
import numpy as np
import copy

start=[0,0,0]
cap=[0,0,0]

print("enter the capacities of jug")

for i in range(3):
    cap[i]=int(input())

cap=np.sort(cap)
print(cap)

start[2]=cap[2]
print(start)

def filljug1(start,cap):
    print("\nFilling jug 1")
    temp = copy.deepcopy(start)
    temp[0], temp[2] = cap[0],temp[2]-cap[0]
    print(temp)

filljug1(start,cap)
```



```
def filljug2(start,cap):
    print("\nFilling jug 2")
    temp = copy.deepcopy(start)
    temp[1], temp[2] = cap[1],temp[2]-cap[1]
    print(temp)

filljug2(start,cap)

def filljug12(start,cap):
    print("\nFilling jug 1 then 2")
    temp = copy.deepcopy(start)
    temp[0], temp[2] = cap[0],temp[2]-cap[0]
    if cap[1]<temp[2]:
      temp[1], temp[2] = cap[1],temp[2]-cap[1]
    if cap[1]>=temp[2]:
      temp[1], temp[2] = temp[2],0

    print(temp)

filljug12(start,cap)
```

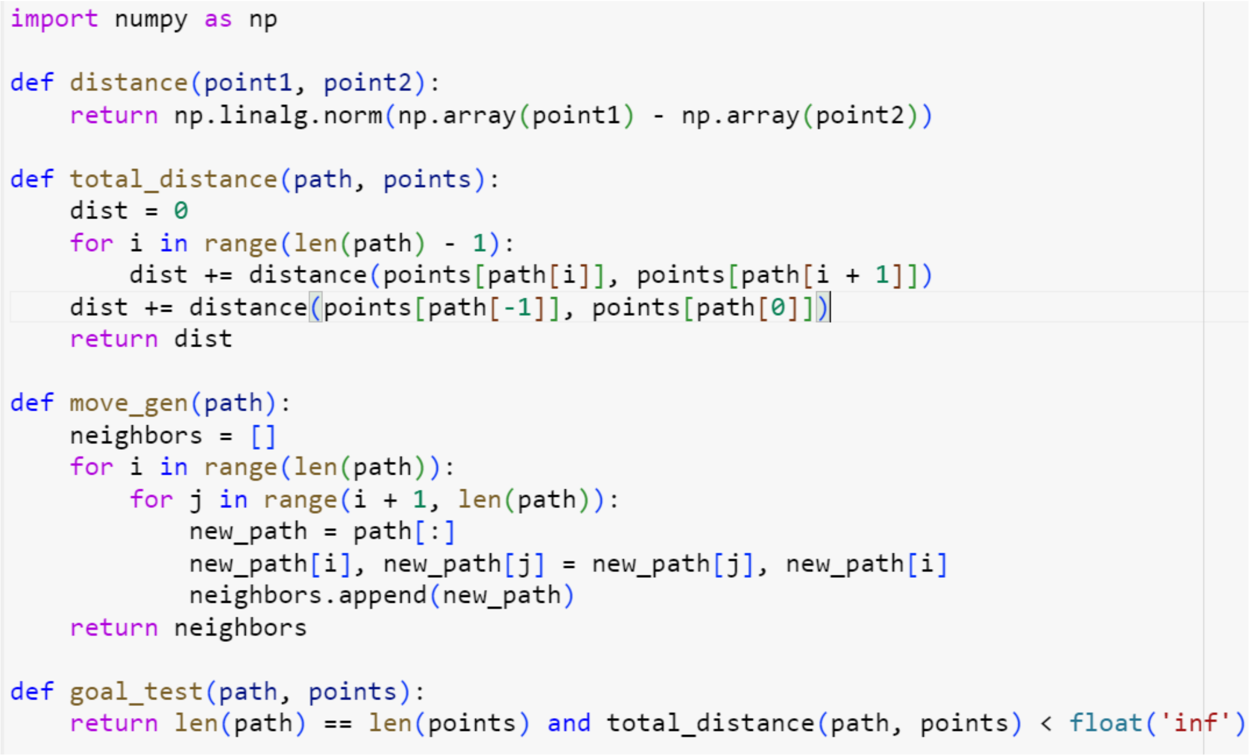


```python
def filljug21(start,cap):
    print("\nFilling jug 2 then 1")
    temp = copy.deepcopy(start)
    temp[1], temp[2] = cap[1],temp[2]-cap[1]
    if cap[0]<temp[2]:
      temp[0], temp[2] = cap[1],temp[2]-cap[1]
    if cap[0]>=temp[2]:
      temp[0], temp[2] = temp[2],0

    print(temp)

filljug21(start,cap)

def movegen(start,cap):
  filljug1(start,cap)
  filljug2(start,cap)
  filljug12(start,cap)
  filljug21(start,cap)

movegen(start,cap)
```

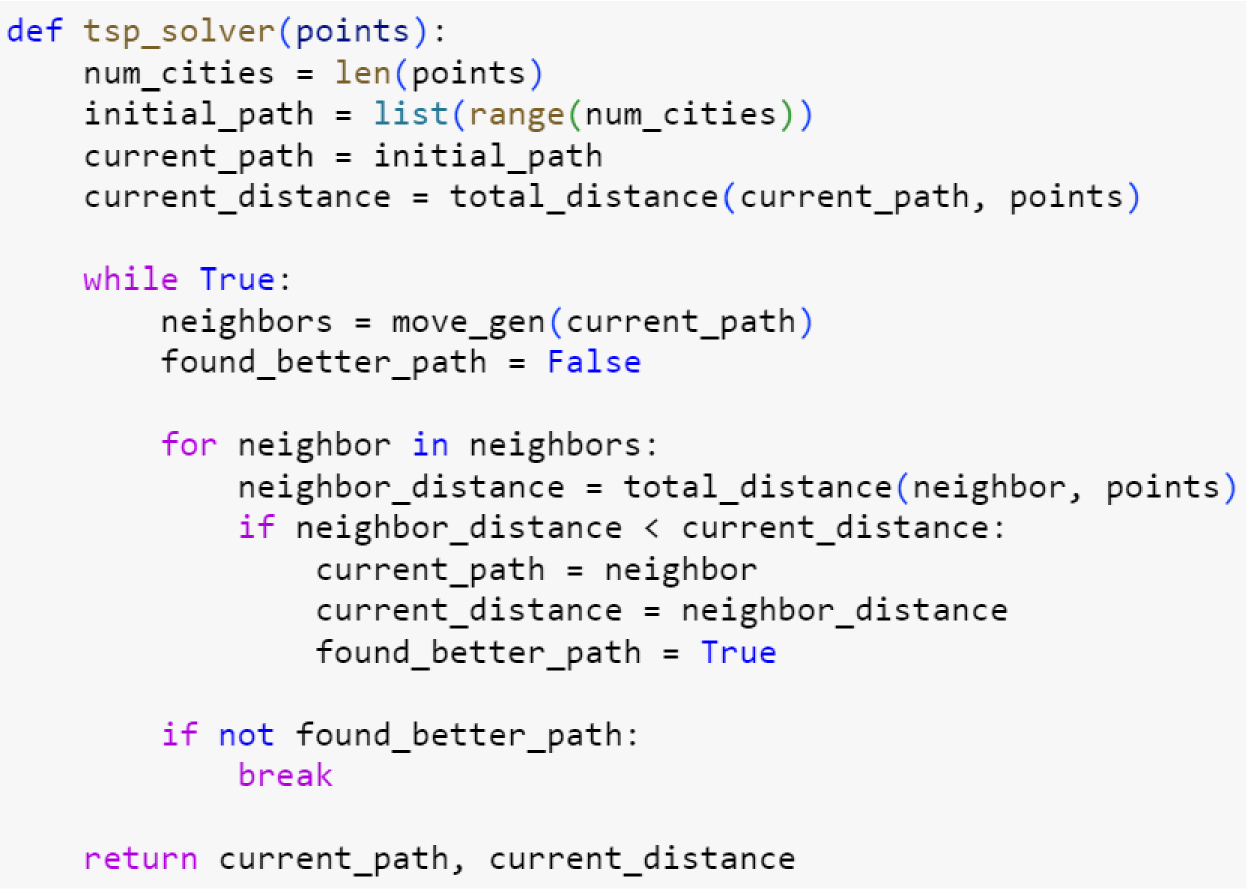


```
enter the capacities of jug
3
5
8
[3 5 8]
[0, 0, 8]

Filling jug 1
[3, 0, 5]

Filling jug 2
[0, 5, 3]

Filling jug 1 then 2
[3, 5, 0]

Filling jug 2 then 1
[3, 5, 0]

Filling jug 1
[3, 0, 5]

Filling jug 2
[0, 5, 3]

Filling jug 1 then 2
[3, 5, 0]

Filling jug 2 then 1
[3, 5, 0]
```

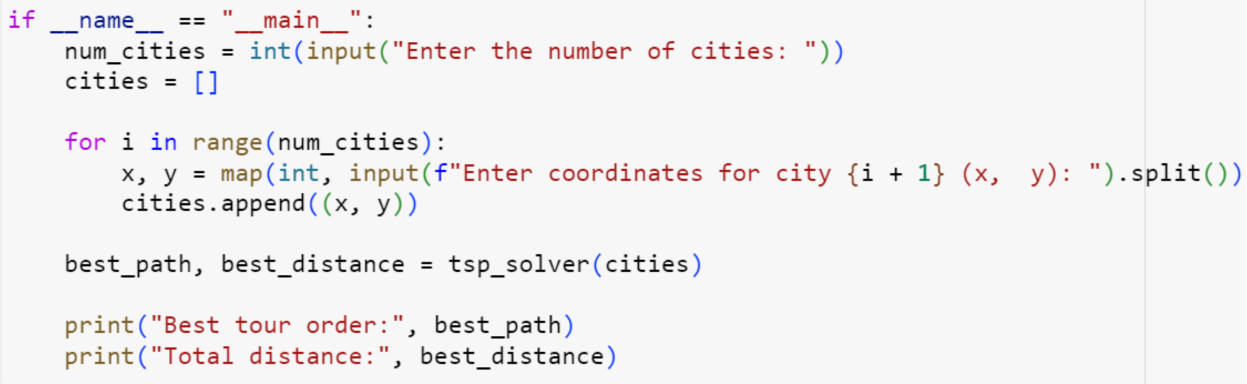
2. **Travelling Salesman Problem**
A salesman is travelling and selling his/her product to in different cities. The condition is that it has to travel each city just once.



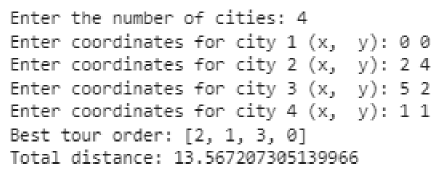
```
import numpy as np

def distance(point1, point2):
    return np.linalg.norm(np.array(point1) - np.array(point2))

def total_distance(path, points):
    dist = 0
    for i in range(len(path) - 1):
        dist += distance(points[path[i]], points[path[i + 1]])
    dist += distance(points[path[-1]], points[path[0]])
    return dist

def move_gen(path):
    neighbors = []
    for i in range(len(path)):
        for j in range(i + 1, len(path)):
            new_path = path[:]
            new_path[i], new_path[j] = new_path[j], new_path[i]
            neighbors.append(new_path)
    return neighbors

def goal_test(path, points):
    return len(path) == len(points) and total_distance(path, points) < float('inf')
```



```python
def tsp_solver(points):
    num_cities = len(points)
    initial_path = list(range(num_cities))
    current_path = initial_path
    current_distance = total_distance(current_path, points)

    while True:
        neighbors = move_gen(current_path)
        found_better_path = False

        for neighbor in neighbors:
            neighbor_distance = total_distance(neighbor, points)
            if neighbor_distance < current_distance:
                current_path = neighbor
                current_distance = neighbor_distance
                found_better_path = True

        if not found_better_path:
            break

    return current_path, current_distance
```



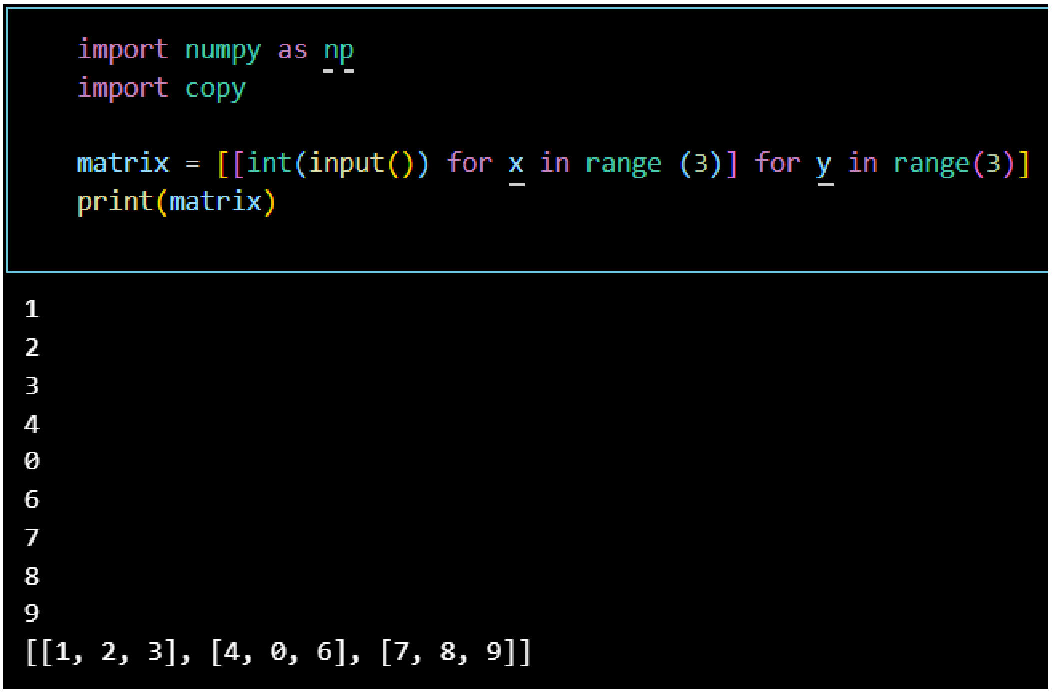
```python
if __name__ == "__main__":
    num_cities = int(input("Enter the number of cities: "))
    cities = []

    for i in range(num_cities):
        x, y = map(int, input(f"Enter coordinates for city {i + 1} (x,  y): ").split())
        cities.append((x, y))

    best_path, best_distance = tsp_solver(cities)

    print("Best tour order:", best_path)
    print("Total distance:", best_distance)
```



```
Enter the number of cities: 4
Enter coordinates for city 1 (x,  y): 0 0
Enter coordinates for city 2 (x,  y): 2 4
Enter coordinates for city 3 (x,  y): 5 2
Enter coordinates for city 4 (x,  y): 1 1
Best tour order: [2, 1, 3, 0]
Total distance: 13.567207305139966
```

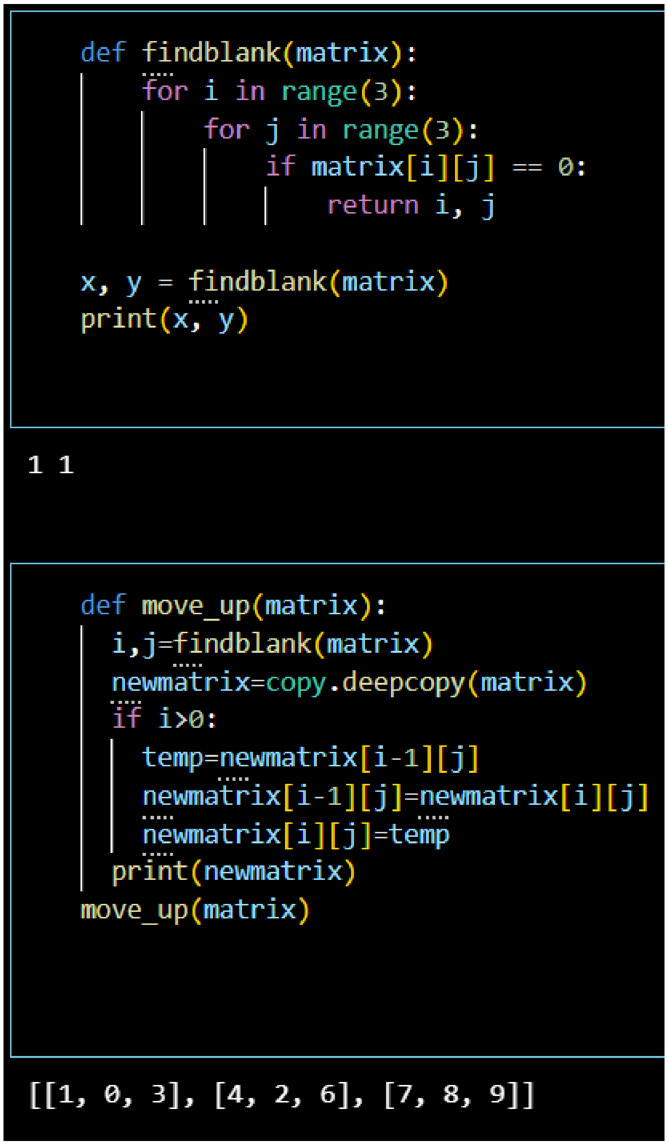
3**. 8 Puzzle Problem**

An initial state is given in a 8 puzzle where one place is blank out of 9 places. You can shift this blank space and get a different state to reach to a given goal state.



```python
import numpy as np
import copy

matrix = [[int(input()) for x in range (3)] for y in range(3)]
print(matrix)
```

```
1
2
3
4
0
6
7
8
9
[[1, 2, 3], [4, 0, 6], [7, 8, 9]]
```

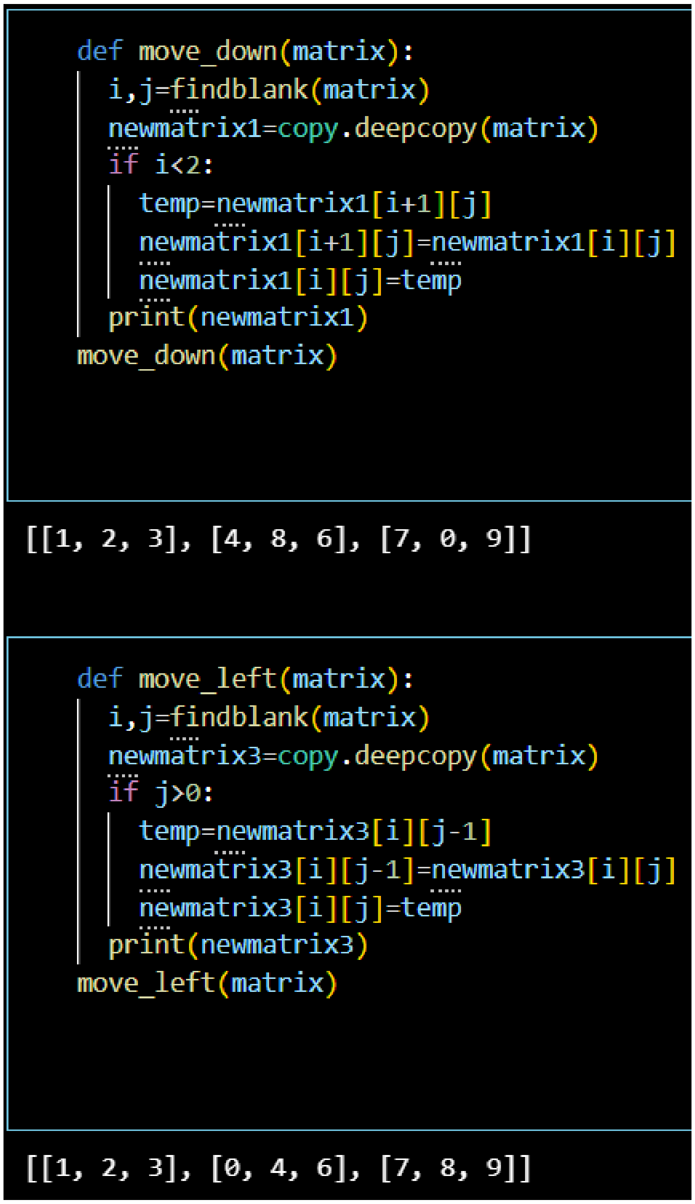


```
def findblank(matrix):
    for i in range(3):
        for j in range(3):
            if matrix[i][j] == 0:
                return i, j

x, y = findblank(matrix)
print(x, y)
```

```
1 1
```

```
def move_up(matrix):
  i,j=findblank(matrix)
  newmatrix=copy.deepcopy(matrix)
  if i>0:
    temp=newmatrix[i-1][j]
    newmatrix[i-1][j]=newmatrix[i][j]
    newmatrix[i][j]=temp
  print(newmatrix)
move_up(matrix)
```

```
[[1, 0, 3], [4, 2, 6], [7, 8, 9]]
```

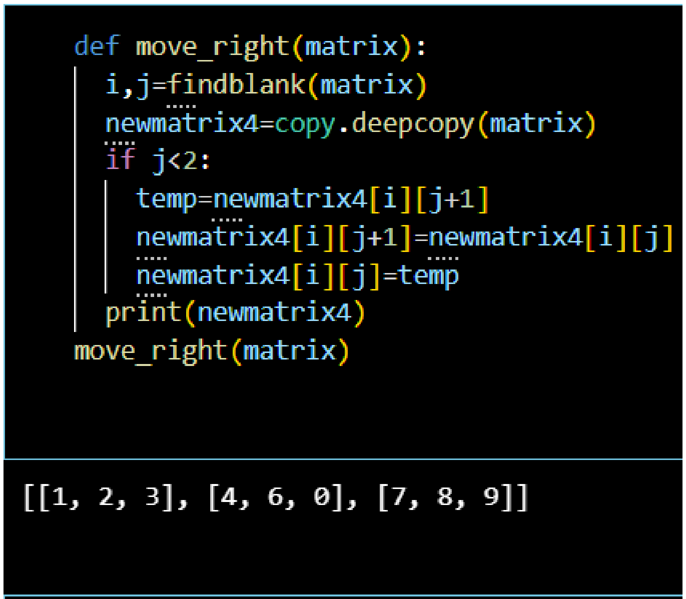


```
def move_down(matrix):
    i,j=findblank(matrix)
    newmatrix1=copy.deepcopy(matrix)
    if i<2:
        temp=newmatrix1[i+1][j]
        newmatrix1[i+1][j]=newmatrix1[i][j]
        newmatrix1[i][j]=temp
    print(newmatrix1)
move_down(matrix)
```

```
[[1, 2, 3], [4, 8, 6], [7, 0, 9]]
```

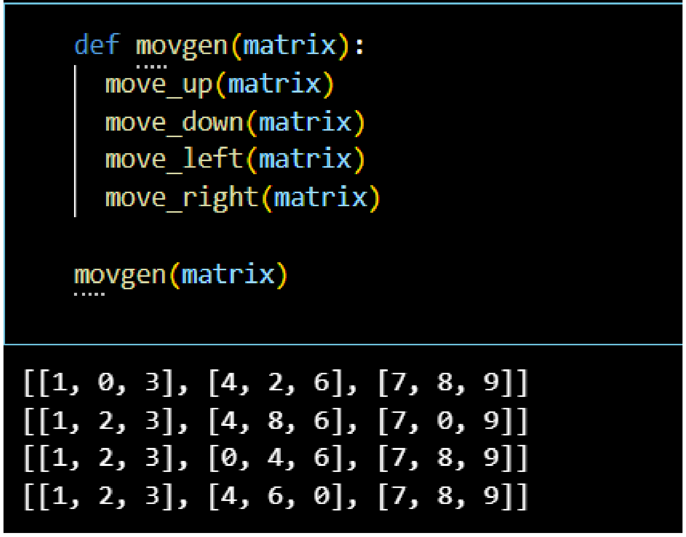
```
def move_left(matrix):
    i,j=findblank(matrix)
    newmatrix3=copy.deepcopy(matrix)
    if j>0:
        temp=newmatrix3[i][j-1]
        newmatrix3[i][j-1]=newmatrix3[i][j]
        newmatrix3[i][j]=temp
    print(newmatrix3)
move_left(matrix)
```

```
[[1, 2, 3], [0, 4, 6], [7, 8, 9]]
```



```
def move_right(matrix):
    i,j=findblank(matrix)
    newmatrix4=copy.deepcopy(matrix)
    if j<2:
        temp=newmatrix4[i][j+1]
        newmatrix4[i][j+1]=newmatrix4[i][j]
        newmatrix4[i][j]=temp
    print(newmatrix4)
move_right(matrix)
```

```
[[1, 2, 3], [4, 6, 0], [7, 8, 9]]
```



```
def movgen(matrix):
    move_up(matrix)
    move_down(matrix)
    move_left(matrix)
    move_right(matrix)

movgen(matrix)
```

```
[[1, 0, 3], [4, 2, 6], [7, 8, 9]]
[[1, 2, 3], [4, 8, 6], [7, 0, 9]]
[[1, 2, 3], [0, 4, 6], [7, 8, 9]]
[[1, 2, 3], [4, 6, 0], [7, 8, 9]]
```