TOY   
WATER JUG

-def pour(j1, j2):

m1, m2, f = 2, 5, 3

print(" %d %d" % (j1, j2))

if j2 is f:

return

elif j2 is m2:

pour(0, j1)

elif j1 != 0 and j2 is 0:

pour(0, j1)

elif j1 is f:

pour(j1, 0)

elif j1 < m1:

pour(m1, j2)

elif j1 < (m2-j2):

pour(0, (j1+j2))

else:

pour(j1-(m2-j2), (m2-j2)+j2)

print("JUG1 JUG2")

pour(0, 0)

TIC TAC TOE

import random

class TicTacToe:

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

self.board = []

def create\_board(self):

for i in range(3):

row = []

for j in range(3):

row.append('-')

self.board.append(row)

def get\_random\_first\_player(self):

return random.randint(0, 1)

def fix\_spot(self, row, col, player):

self.board[row][col] = player

def is\_player\_win(self, player):

win = None

n = len(self.board)

# checking rows

for i in range(n):

win = True

for j in range(n):

if self.board[i][j] != player:

win = False

break

if win:

return win

# checking columns

for i in range(n):

win = True

for j in range(n):

if self.board[j][i] != player:

win = False

break

if win:

return win

# checking diagonals

win = True

for i in range(n):

if self.board[i][i] != player:

win = False

break

if win:

return win

win = True

for i in range(n):

if self.board[i][n - 1 - i] != player:

win = False

break

if win:

return win

return False

for row in self.board:

for item in row:

if item == '-':

return False

return True

def is\_board\_filled(self):

for row in self.board:

for item in row:

if item == '-':

return False

return True

def swap\_player\_turn(self, player):

return 'X' if player == 'O' else 'O'

def show\_board(self):

for row in self.board:

for item in row:

print(item, end=" ")

print()

def start(self):

self.create\_board()

player = 'X' if self.get\_random\_first\_player() == 1 else 'O'

while True:

print(f"Player {player} turn")

self.show\_board()

# taking user input

row, col = list(

map(int, input("Enter row and column numbers to fix spot: ").split()))

print()

# fixing the spot

self.fix\_spot(row - 1, col - 1, player)

# checking whether current player is won or not

if self.is\_player\_win(player):

print(f"Player {player} wins the game!")

break

# checking whether the game is draw or not

if self.is\_board\_filled():

print("Match Draw!")

break

# swapping the turn

player = self.swap\_player\_turn(player)

# showing the final view of board

print()

self.show\_board()

# starting the game

tic\_tac\_toe = TicTacToe()

tic\_tac\_toe.start()

SUDOKU

# N is the size of the 2D matrix N\*N

N = 9

# A utility function to print grid

def printing(arr):

for i in range(N):

for j in range(N):

print(arr[i][j], end=" ")

print()

# Checks whether it will be

# legal to assign num to the

# given row, col

def isSafe(grid, row, col, num):

# Check if we find the same num

# in the similar row , we

# return false

for x in range(9):

if grid[row][x] == num:

return False

# Check if we find the same num in

# the similar column , we

# return false

for x in range(9):

if grid[x][col] == num:

return False

# Check if we find the same num in

# the particular 3\*3 matrix,

# we return false

startRow = row - row % 3

startCol = col - col % 3

for i in range(3):

for j in range(3):

if grid[i + startRow][j + startCol] == num:

return False

return True

# Takes a partially filled-in grid and attempts

# to assign values to all unassigned locations in

# such a way to meet the requirements for

# Sudoku solution (non-duplication across rows,

# columns, and boxes) \*/

def solveSudoku(grid, row, col):

# Check if we have reached the 8th

# row and 9th column (0

# indexed matrix) , we are

# returning true to avoid

# further backtracking

if (row == N - 1 and col == N):

return True

# Check if column value becomes 9 ,

# we move to next row and

# column start from 0

if col == N:

row += 1

col = 0

# Check if the current position of

# the grid already contains

# value >0, we iterate for next column

if grid[row][col] > 0:

return solveSudoku(grid, row, col + 1)

for num in range(1, N + 1, 1):

# Check if it is safe to place

# the num (1-9) in the

# given row ,col ->we

# move to next column

if isSafe(grid, row, col, num):

# Assigning the num in

# the current (row,col)

# position of the grid

# and assuming our assigned

# num in the position

# is correct

grid[row][col] = num

# Checking for next possibility with next

# column

if solveSudoku(grid, row, col + 1):

return True

# Removing the assigned num ,

# since our assumption

# was wrong , and we go for

# next assumption with

# diff num value

grid[row][col] = 0

return False

# Driver Code

# 0 means unassigned cells

grid = [[3, 0, 6, 5, 0, 8, 4, 0, 0],

[5, 2, 0, 0, 0, 0, 0, 0, 0],

[0, 8, 7, 0, 0, 0, 0, 3, 1],

[0, 0, 3, 0, 1, 0, 0, 8, 0],

[9, 0, 0, 8, 6, 3, 0, 0, 5],

[0, 5, 0, 0, 9, 0, 6, 0, 0],

[1, 3, 0, 0, 0, 0, 2, 5, 0],

[0, 0, 0, 0, 0, 0, 0, 7, 4],

[0, 0, 5, 2, 0, 6, 3, 0, 0]]

if (solveSudoku(grid, 0, 0)):

printing(grid)

else:

print("no solution exists ")