#Write a Program to Implement Breadth First Search using Python.

```
graph = {
  'A' : ['B','C'],
  'B' : ['D', 'E'],
  'C' : ['F'],
  'D' : [],
  'E' : ['F'],
  'F' : []
}
visited = [] # List to keep track of visited nodes.
queue = [] #Initialize a queue

def bfs(visited, graph, node):
  visited.append(node)
  queue.append(node)

  while queue:
    s = queue.pop(0)
    print (s, end = " ")

    for neighbour in graph[s]:
      if neighbour not in visited:
        visited.append(neighbour)
        queue.append(neighbour)

# Driver Code
bfs(visited, graph, 'A')
```

Output:-

```
A B C D E F
```

EXPERIMENT 2

#Write a Program to Implement Depth First Search using Python.

```
# Using a Python dictionary to act as an adjacency list
graph = {
   'A' : ['B','C'],
   'B' : ['D', 'E'],
   'C' : ['F'],
   'D' : [],
   'E' : ['F'],
   'F' : []
}

visited = set() # Set to keep track of visited nodes.
def dfs(visited, graph, node):
   if node not in visited:
      print (node)
      visited.add(node)
      for neighbour in graph[node]:
         dfs(visited, graph, neighbour)

# Driver Code
dfs(visited, graph, 'A')
```

Output:-

```
A
B
D
E
F
C
```

EXPERIMENT 3

#Write a Program to Implement Tic-Tac-Toe game using Python.

```
# Tic-Tac-Toe Program using
# random number in Python

# importing all necessary libraries
import numpy as np
import random
from time import sleep

# Creates an empty board
def create_board():
    return(np.array([[0, 0, 0],
                     [0, 0, 0],
                     [0, 0, 0]]))
# Check for empty
places on board
def
possibilities(boar
d):
    l = []

    for i in range(len(board)):
        for j in range(len(board)):

            if board[i][j] == 0:
                l.append((i, j))
    return(l)

# Select a random
place for the player
def
random_place(board
, player):
```

```
        selection = possibilities(board)
        current_loc =
        random.choice(selecti
        on) board[current_loc]
        = player
        return(board)

# Checks whether the
player has three # of
their marks in a
horizontal row def
row_win(board,
player):
        for x in range(len(board)):
                win = True

                for y in range(len(board)):
                        if board[x, y] != player:
                                win = False
                                continue

                if win == True:
                        return(win)
        return(win)

# Checks whether the
player has three # of
their marks in a
vertical row
def col_win(board, player):
        for x in range(len(board)):
                win = True

                for y in range(len(board)):
                        if board[y][x] != player:
                                win = False
                                continue

                if win == True:
```

```
                    return(win)
        return(win)

# Checks whether the
player has three # of
their marks in a
diagonal row def
diag_win(board,
player):
        win = True
        y = 0
        for x in range(len(board)):
                if board[x, x] != player:
                        win = False
        if win:
                return win
        win = True
        if win:
                for x in range(len(board)):
                        y = len(board) - 1 - x
                        if board[x, y] != player:
                                win = False
        return win

# Evaluates whether there is
# a winner or a tie
def evaluate(board):
        winner = 0

        for player in [1, 2]:
                if (row_win(board, player) or
                        col_win(board,player) or
                        diag_win(board,player)):

                        winner = player

        if np.all(board != 0) and winner == 0:
                winner = -1
        return winner
```

```
# Main function to start the game
def play_game():
        board, winner, counter = create_board(), 0, 1
        print(board)
        sleep(2)

        while winner == 0:
                for player in [1, 2]:
                        board = random_place(board, player)
                        print("Board after " + str(counter) + " move")
                        print(board)
                        sleep(2)
                        counter += 1
                        winner = evaluate(board)
                        if winner != 0:
                                break
        return(winner)

# Driver Code
print("Winner is: " + str(play_game()))
```

Output:-

```
[[0 0 0]
 [0 0 0]
 [0 0 0]]
Boar
d
afte
r 1
move
[[0
0
0]
 [0 0 0]
 [1 0 0]]
Boar
d
afte
```

```
r 2
move
[[0
0
0]
 [0 2 0]
 [1 0 0]]
Boar
d
afte
r 3
move
[[0
1
0]
 [0 2 0]
 [1 0 0]]
Boar
d
afte
r 4
move
[[0
1
0]
 [2 2 0]
 [1 0 0]]
Boar
d
afte
r 5
move
[[1
1
0]
 [2 2 0]
 [1 0 0]]
Boar
d
afte
r 6
move
[[1
1
0]
```

```
 [2 2 0]
 [1 2 0]]
Boar
d
afte
r 7
move
[[1
1
0]
 [2 2 0]
 [1 2 1]]
Boar
d
afte
r 8
move
[[1
1 0]
 [2 2 2]
 [1 2 1]]
Winner is: 2
```