**1. Use logic programming in Python to check for prime numbers.**

**Code**

num = 30

# To take input from the user

#num = int(input("Enter a number: "))

# define a flag variable

flag = False

# prime numbers are greater than 1

if num > 1:

# check for factors

for i in range(2, num):

if (num % i) == 0:

# if factor is found, set flag to True

flag = True

# break out of loop

break

# check if flag is True

if flag:

print(num, "is not a prime number")

else:

print(num, "is a prime number")

**output**

30 is not a prime number

>

**2.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(board):

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(selection) 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,playr)):

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]]

Board after 1 move

[[0 0 0]

[0 0 0]

[1 0 0]]

Board after 2 move

[[0 0 0]

[0 2 0]

[1 0 0]]

Board after 3 move

[[0 1 0]

[0 2 0]

[1 0 0]]

Board after 4 move

[[0 1 0]

[2 2 0]

[1 0 0]]

Board after 5 move

[[1 1 0]

[2 2 0]

[1 0 0]]

Board after 6 move

[[1 1 0]

[2 2 0]

[1 2 0]]

Board after 7 move

[[1 1 0]

[2 2 0]

[1 2 1]]

Board after 8 move

[[1 1 0]

[2 2 2]

[1 2 1]]

Winner is: 2

**3.Write a Program to Implement Depth First Search using Python.**

# program to print DFS traversal # from a given given graph

from collections import defaultdict

# This class represents a directed graph using # adjacency list representation

class Graph:

# Constructor

def \_\_init (self):

# default dictionary to store graph

self.graph = defaultdict(list)

# function to add an edge to graph def addEdge(self, u, v):

self.graph[u].append(v)

# A function used by DFS def DFSUtil(self, v, visited):

# Mark the current node as visited # and print it

visited.add(v) print(v, end=' ')

# Recur for all the vertices # adjacent to this vertex

for neighbour in self.graph[v]: if neighbour not in visited:

self.DFSUtil(neighbour, visited)

# The function to do DFS traversal. It uses # recursive DFSUtil()

def DFS(self, v):

# Create a set to store visited vertices visited = set()

# Call the recursive helper function # to print DFS traversal self.DFSUtil(v, visited)

# Driver code

# Create a graph given # in the above diagram

g = Graph() g.addEdge(0, 1)

g.addEdge(0, 2)

g.addEdge(1, 2)

g.addEdge(2, 0)

g.addEdge(2, 3)

g.addEdge(3, 3)

print("Following is DFS from (starting from vertex 2)") g.DFS(2)

Output:

Following is Depth First Traversal (starting from vertex2)

2 0 1 9 3