|  |  |
| --- | --- |
| **#** | **Question + Solution** |
| 1 | Validate y-axisThe Y-Axis needs a presence, type and range check addedGetMoveCoordinate receives input from the user so this is the place to add the presence and type checks.CheckValidMove validates the numeric coordinate that was entered so this is the place to add the range check.def GetMoveCoordinates(): X = int(input('Enter x coordinate: ')) Y = int(input('Enter y coordinate: ')) return X, Ydef CheckValidMove(XCoordinate, YCoordinate, Board): ValidMove = True # Check x coordinate is valid if (XCoordinate < 1) or (XCoordinate > 3): ValidMove = False return ValidMoveInput for the Y coordinate is converted to an int. So if you hit return without entering a Y value there will be a runtime error as an empty string cannot be converted to an integer.Therefore we need to add presence validation to ensure a value is entered.We can add a try/except to catch the runtime error and request the user add a value (i.e. do a presence check). It will also do the type, i.e. datatype, validation:def GetMoveCoordinates(): X = int(input('Enter x coordinate: ')) # 1) add presence and integer type check using try/except valid = False while not valid: try: Y = int(input('Enter y coordinate: ')) valid = True except: print("Please enter a numeric value for Y") return X, YWe add the *valid* boolean variable to ensure that the while loop iterates until the user enters a numeric value for the Y coordinate. Putting the Y = int(input('Enter y coordinate: ')) statement within a try block means that if it succeeds, the *valid* variable is set to True, so the while loop exits. If it fails then execution steps to the except: label, the error message is printed and the *valid* boolean variable remains false, so the while loop iterates again. This continues until the user enters an integer value.While we’re here let’s also add this presence and type check for the X coordinate:def GetMoveCoordinates(): # 1) add presence and integer type check using try/except valid = False while not valid: try: X = int(input('Enter x coordinate: ')) valid = True except: print("Please enter a numeric value for X") valid = False while not valid: try: Y = int(input('Enter y coordinate: ')) valid = True except: print("Please enter a numeric value for Y") return X, YNow the user is repeatedly asked for X and Y values until until integer ones are input:Now to add the range check for the Y coordinate. This is most easily done in the CheckValidMove function in the same way that the X coordinate range check is implemented.def CheckValidMove(XCoordinate, YCoordinate, Board): ValidMove = True # Check x coordinate is valid if (XCoordinate < 1) or (XCoordinate > 3): ValidMove = False # 1) add range check for Y coordinate elif (YCoordinate < 1) or (YCoordinate > 3): ValidMove = False return ValidMoveSo, now if the coordinate is less than 1 or greater than 3 (i.e. not between 1 and 3) then the coordinate will fail the range check. Note that the Y range check is added as an *elif* rather than a new *if* as there is no point running this branch of the selection if the X coordinate range check fails. |
| 2 | Type/presence check validation on coordinatesand some for the x-axis tooWe’ve already done this in 1) above.However, note that this method of using try/except is perfectly acceptable, but an alternative way to do this is to take the coordinates as string inputs (i.e. do not convert the input to ints) and then simply test that the input is not an empty string and that it contains whole numbers, only (by using the Python *isdigit* function). This way you can output more specific error messages:def GetMoveCoordinates(): valid = False while not valid: X = input('Enter x coordinate: ') # 1) presence check if X == '': print("You did not input a value!") # 1) type (integer) check elif not X.isdigit(): print("You must input a whole number") else: valid = True valid = False while not valid: Y = input('Enter y coordinate: ') # 1) presence check if Y == '': print("You did not input a value!") # 1) type (integer) check elif not Y.isdigit(): print("You must input a whole number") else: valid = True # 1) string X, Y output as integers return int(X), int(Y)With these validations we can be sure that (string) variables X and Y will convert to integers without error so the X and Y string values are returned as integers in the return statement. |
| 3 | Stop the overwriting of symbolsCurrently you can choose a coordinate that already has a symbolThis test shows *X* added at (1,1) and then overwriting with *O* by entering I1,1) again:therefore change CheckValidMove and return False (i.e. the move is invalid) if the element in the Board array does not contain a blank string (i.e. it is X or Y). Add as *elif* rather than a new *if* because there is no point running this branch of the selection if either of the previous range checks have failed:def CheckValidMove(XCoordinate, YCoordinate, Board): ValidMove = True # Check x coordinate is valid if (XCoordinate < 1) or (XCoordinate > 3): ValidMove = False # 1) add range check for Y coordinate elif (YCoordinate < 1) or (YCoordinate > 3): ValidMove = False # 3) check that the element is empty before allowing a symbol to be placed there elif Board[XCoordinate][YCoordinate] != ' ': ValidMove = False return ValidMoveNow we see an error message when selecting a coordinate already occupied: |
| 4 | Check diagonal for winCurrently diagonal lines don't win... change the code so they doA winning line is determined in the CheckXOr0HasWon function. We need to add a couple of extra checks for the diagonals:def CheckXOr0HasWon(Board): XOrOHasWon = False for Column in range(1, 4): if (Board[Column][1] == Board[Column][2]) and (Board[Column][2] == Board[Column][3]) and ( Board[Column][2] != ' '): XOrOHasWon = True for Row in range(1, 4): if (Board[1][Row] == Board[2][Row]) and (Board[2][Row] == Board[3][Row]) and (Board[2][Row] != ' '): XOrOHasWon = True # 4) set 'win' for downward diagonal if Board[1][1] == Board[2][2] and Board[2][2] == Board[3][3] and Board[2][2] != ' ': XOrOHasWon = True # 4) set 'win' for upward diagonal elif Board[3][1] == Board[2][2] and Board[2][2] == Board[1][3] and Board[2][2] != ' ': XOrOHasWon = True return XOrOHasWonNote that the second diagonal check is added as an *elif* rather than a new *if* as there is no point checking this if the first diagonal check for a win succeeded.Downwards diagonal wins: Board [1][1], [2][2], [3][3]:Upwards diagonal wins: [1][3], [2][2], [3][1] |
| 5 | Name the player's goHave it so the code says it's Bob's turn etc..Before the player enters their coordinates, output their name by checking the value of CurrentSymbol which contains either the value PlayerOneSymbol or the value PlayerTwoSymbol. If the CurrentSymbol has the same value as PlayerOneSymbol then we know it must be player one’s turn, else it must be player two’s turn. We can get the names from the PlayerOneName and PlayerTwoName variables. Note that the print statement uses “” so that we can output the single apostrophe (‘):CurrentSymbol = StartSymbolwhile (not GameHasBeenWon) and (not GameHasBeenDrawn): # Play until a player wins or the game is drawn # 5) get player name based on current symbol if CurrentSymbol == PlayerOneSymbol: print(PlayerOneName + "'s turn") else: print(PlayerTwoName + "'s turn") ValidMove = False; while not ValidMove: # Get a valid moveoutput: |
| 6 | Who is the winner?The winner isn't named at the moment (just the symbol) add a nice msg naming the winnerActually, this is already done!if GameHasBeenWon: # Update scores and display result if (PlayerOneSymbol == CurrentSymbol): print(PlayerOneName + " congratulations you win!") PlayerOneScore += 1 else: print(PlayerTwoName + " congratulations you win!") PlayerTwoScore += 1 |
| 7 | Check both players don't have the same nameHaving two players with the same name would be a pain.. validate this outCurrently, the same name can be added for both players:if \_\_name\_\_ == "\_\_main\_\_": PlayerOneName = input('What is the name of player one? ') PlayerTwoName = input('What is the name of player two? ')So add validation to ensure names are different:if \_\_name\_\_ == "\_\_main\_\_": # 7) data validation - ensure player names are different while PlayerOneName == PlayerTwoName: PlayerOneName = input('What is the name of player one? ') PlayerTwoName = input('What is the name of player two? ') if PlayerOneName == PlayerTwoName: print("Players' names must be different!\n")Output: |
| 8 | Change the size of the gridA 4x4 or 5x5 grid would be nice...  but it's a surprising amount of work if you have it so 3 in a line winsThe quickest way to do this is to choose a maximum size (e.g. 5x5) , initialise the array accordingly:Board = [[0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0], ](note that we create a 6x6 array because row 0 and column 0 are not used in the code) and then we ensure that all existing iterations loop to 6 rather than 4, e.g.:# 8) loop from 1 to 6 to play 5x5 XOdef DisplayBoard(Board): print(' | 1 2 3 4 5') print('--+-----------') for Row in range(1, 6): print(Row, '|', end=' ') for Column in range(1, 6): print(Board[Column][Row], end=' ') print() print('\n')# 8) loop from 1 to 6 to play 5x5 XOdef ClearBoard(Board): for Row in range(1, 6): for Column in range(1, 6): Board[Column][Row] = ' 'Coordinate validation should allow 1 to 5:def CheckValidMove(XCoordinate, YCoordinate, Board): ValidMove = True # Check x coordinate is valid # 9) validation check now allows coordinate values 1 to 5  if (XCoordinate < 1) or (XCoordinate > 5): ValidMove = False # 1) add range check for Y coordinate elif (YCoordinate < 1) or (YCoordinate > 5): ValidMove = False # 3) check that the element is empty before allowing a symbol to be placed there elif Board[XCoordinate][YCoordinate] != ' ': ValidMove = False return ValidMoveNumber of moves for a draw becomes 5x5:if not GameHasBeenWon: # 8) Updated for 5x5 game if NoOfMoves == 25: # Check if maximum number of allowed moves has been reached GameHasBeenDrawn = TrueThe final and tricky part is checking for three consecutive symbols in the array. Something like the below does the trick. Note that rather than hardcoding to loop to element number 6, I have used variable: *sizeOfBoard*. This means that it’s easy to extend the game so that the user can input any size of board (e.g. 10) and the game will work. You’d need to intialise the array by using the append method in a loop, make *sizeOfBoard* a global variable and replace instances of 6 with sizeOfBoard…def CheckXOr0HasWon(Board): # 8) new variable SizeOfBoard so can extend if necessary (for 5x5 board) sizeOfBoard = 5 XOrOHasWon = False for Row in range(1, sizeOfBoard + 1): # ...check if 3 consecutive elements across have the same symbol and are not empty # ensure don't exceed columns by looping to SizeOfBoard - 2 for Column in range(1, sizeOfBoard - 1): if Board[Column][Row] == Board[Column + 1][Row] and Board[Column][Row] == Board[Column + 2][Row] and Board[Column][Row] != ' ': XOrOHasWon = True # for each column... for Column in range(1, sizeOfBoard + 1): # ...check if 3 consecutive elements down have the same symbol and are not empty # ensure don't exceed rows by looping to SizeOfBoard - 2 for Row in range(1, sizeOfBoard - 1): if Board[Column][Row] == Board[Column][Row + 1] and Board[Column][Row] == Board[Column][Row + 2] and Board[Column][Row] != ' ': XOrOHasWon = True # diagonal down - 3 consecutive symbols the same and not empty for Column in range(1, sizeOfBoard - 1): for Row in range(1, sizeOfBoard - 1): if Board[Column][Row] == Board[Column + 1][Row + 1] and Board[Column][Row] == Board[Column + 2][Row + 2] and Board[Column][Row] != ' ': XOrOHasWon = True # diagonal up - 3 consecutive symbols the same and not empty # Row is inverse of Column for Column in range(1, sizeOfBoard - 1): for Row in range(3, sizeOfBoard + 1): if Board[Column][Row] == Board[Column + 1][Row - 1] and Board[Column][Row] == Board[Column + 2][Row - 2] and Board[Column][Row] != ' ': XOrOHasWon = True return XOrOHasWonOutput example: |
| 9 | Draw scores both players 0.5 pointsDraws don't score at the moment. award each player 0.5 for a drawJust two additional lines of code needed for this:else: print("A draw this time!") # 9) Add 0.5 to both players' scores if a draw PlayerOneScore += 0.5 PlayerTwoScore += 0.5 |
| 10 | Play the computer (random validated placing of scores)Random is fairly easy... a player that follows a strategy is trickier but not impossibleFirst create and initialise a new global variable:PlayComputer = FalseThen determine if the user wishes to play the computer:if \_\_name\_\_ == "\_\_main\_\_": # 7) data validation - ensure player names are different while PlayerOneName == PlayerTwoName: PlayerOneName = input('What is the name of player one? ') # 10) Determine if wish to play the computer playComputer = '' while playComputer not in ['Y', 'N']: playComputer = input('Do you wish to play the computer (Y or N)? ').upper() if playComputer not in ['Y', 'N']: print('You must input Y or N') if playComputer == 'Y': PlayerTwoName = 'Computer' else: PlayerTwoName = input('What is the name of player two? ') if PlayerOneName == PlayerTwoName: print("Players' names must be different!\n") if playComputer == 'Y': PlayComputer = TrueThen, when it’s player two’s turn, check if we are playing the computer (which will always be player two), and if so, rather than calling *GetMoveCoordinates* we call a new function that returns (valid) random coordinates, which we’ll call *GetRandomCoordinates*:if CurrentSymbol == PlayerOneSymbol: print(PlayerOneName + "'s turn")else: print(PlayerTwoName + "'s turn")if PlayComputer and CurrentSymbol == PlayerTwoSymbol: XCoord, YCoord = GetRandomCoordinates()else: ValidMove = False; while not ValidMove: # Get a valid move XCoord, YCoord = GetMoveCoordinates() ValidMove = CheckValidMove(XCoord, YCoord, Board) if not ValidMove: print("Coordinates invalid, please try again")And the function could look something like this:def GetRandomCoordinates(): valid = False while not valid: X = random.randint(1, 3) Y = random.randint(1, 3) if Board[X][Y] == ' ': valid = True return X, YThe output will look something like this if we allow the computer to win:…A winning strategy is for the computer to try every possible move that the opponent can make, test if any of them win, and then place a blocking symbol there. This can be extended using recursion to try every move in this way going forward, and weighting winning moves. Check out this video to see how this would work : <https://www.youtube.com/watch?v=trKjYdBASyQ> |
| 11 | Place the section of code where the players enter their names in a separate procedure, with the appropriate parameters passed-in.Don't fully factorize the code.. but have a go at a new procedure called EnterPlayerNameSo, the part of the code which collects and validates the player names could to put into its own procedure like so:def EnterPlayerName(): playerOneName = '' playerTwoName = '' playingComputer = False while playerOneName == playerTwoName: playerOneName = input('What is the name of player one? ') # 10) Determine if wish to play the computer playComputer = '' while playComputer not in ['Y', 'N']: playComputer = input('Do you wish to play the computer (Y or N)? ').upper() if playComputer not in ['Y', 'N']: print('You must input Y or N') if playComputer == 'Y': playerTwoName = 'Computer' else: playerTwoName = input('What is the name of player two? ') if playerOneName == playerTwoName: print("Players' names must be different!\n") if playComputer == 'Y': playingComputer = True return playerOneName, playerTwoName, playingComputerand call this new function at the start of the program execution:if \_\_name\_\_ == "\_\_main\_\_": PlayerOneName, PlayerTwoName, PlayComputer = EnterPlayerName()Note that we could pass in PlayerOneName, PlayerTwoName and PlayComputer global variables as parameters to this function but it’s not necessary. Also, I have created local variables playerOneName, playerTwoName and playingComputer (by changing the case of the first character these are new variables) so as not to get them confused with the global variables. |
| 12 | Convert Symbol to uppercase automatically if user inputs lowercase x or oRather than validate the X O choice just convert it to upper caseFor this we use the *upper* function/method to convert a string to upper case, so the code:while PlayerOneSymbol not in ['X', 'O']: # Choose player one's symbol PlayerOneSymbol = input(PlayerOneName + ' what symbol do you wish to use, X or O? ') if PlayerOneSymbol not in ['X', 'O']: print('Symbol to play must be uppercase X or O')becomes:while PlayerOneSymbol not in ['X', 'O']: # Choose player one's symbol PlayerOneSymbol = input(PlayerOneName + ' what symbol do you wish to use, X or O? ').upper() if PlayerOneSymbol not in ['X', 'O']: print('Symbol to play must be X or O')We’ve removed the ‘uppercase’ part of the error message as that’s nolonger required: |

The full code:

# Skeleton Program Code for AQA Comp1 Summer 2010 Examination
# this code should be used in conjunction with the Preliminary
# Materials
# written by the AQA COMP1 Programmer Team
# developed using Python v3.0

import random

Board = [[0, 0, 0, 0, 0, 0],
 [0, 0, 0, 0, 0, 0],
 [0, 0, 0, 0, 0, 0],
 [0, 0, 0, 0, 0, 0],
 [0, 0, 0, 0, 0, 0],
 [0, 0, 0, 0, 0, 0],
 ]
PlayerOneName = ""
PlayerTwoName = ""
XCoord = 0
YCoord = 0
NoOfMoves = 0
ValidMove = False
GameHasBeenWon = False
GameHasBeenDrawn = False
CurrentSymbol = ''
StartSymbol = ''
PlayerOneSymbol = ''
PlayerTwoSymbol = ''
Answer = ''
PlayerOneScore = 0.0
PlayerTwoScore = 0.0
PlayComputer = False

# 8) loop from 1 to 6 to play 5x5 XO
def DisplayBoard(Board):
 print(' | 1 2 3 4 5')
 print('--+-----------')
 for Row in range(1, 6):
 print(Row, '|', end=' ')
 for Column in range(1, 6):
 print(Board[Column][Row], end=' ')
 print()
 print('\n')

# 8) loop from 1 to 6 to play 5x5 XO
def ClearBoard(Board):
 for Row in range(1, 6):
 for Column in range(1, 6):
 Board[Column][Row] = ' '

def CheckValidMove(XCoordinate, YCoordinate, Board):
 ValidMove = True
 # Check x coordinate is valid
 if (XCoordinate < 1) or (XCoordinate > 5):
 ValidMove = False
 # 1) add range check for Y coordinate
 elif (YCoordinate < 1) or (YCoordinate > 5):
 ValidMove = False
 # 3) check that the element is empty before allowing a symbol to be placed there
 elif Board[XCoordinate][YCoordinate] != ' ':
 ValidMove = False
 return ValidMove

def GetMoveCoordinates():
 valid = False
 while not valid:
 X = input('Enter x coordinate: ')
 # 1) presence check
 if X == '':
 print("You did not input a value!")
 # 1) type (integer) check
 elif not X.isdigit():
 print("You must input a whole number")
 else:
 valid = True

 valid = False
 while not valid:
 Y = input('Enter y coordinate: ')
 # 1) presence check
 if Y == '':
 print("You did not input a value!")
 # 1) type (integer) check
 elif not Y.isdigit():
 print("You must input a whole number")
 else:
 valid = True

 # 1) string X, Y output as integers
 return int(X), int(Y)

def GetRandomCoordinates():
 valid = False
 while not valid:
 X = random.randint(1, 5)
 Y = random.randint(1, 5)
 if Board[X][Y] == ' ':
 valid = True
 return X, Y

def CheckXOr0HasWon(Board):
 # 10) new variable SizeOfBoard so can extend if necessary
 sizeOfBoard = 5
 XOrOHasWon = False

 for Row in range(1, sizeOfBoard + 1):
 # ...check if 3 consecutive elements across have the same symbol and are not empty
 # ensure don't exceed columns by looping to SizeOfBoard - 2
 for Column in range(1, sizeOfBoard - 1):
 if Board[Column][Row] == Board[Column + 1][Row] and Board[Column][Row] == Board[Column + 2][Row] and Board[Column][Row] != ' ':
 XOrOHasWon = True

 # for each column...
 for Column in range(1, sizeOfBoard + 1):
 # ...check if 3 consecutive elements down have the same symbol and are not empty
 # ensure don't exceed rows by looping to SizeOfBoard - 2
 for Row in range(1, sizeOfBoard - 1):
 if Board[Column][Row] == Board[Column][Row + 1] and Board[Column][Row] == Board[Column][Row + 2] and Board[Column][Row] != ' ':
 XOrOHasWon = True

 # diagonal down - 3 consecutive symbols the same and not empty
 for Column in range(1, sizeOfBoard - 1):
 for Row in range(1, sizeOfBoard - 1):
 if Board[Column][Row] == Board[Column + 1][Row + 1] and Board[Column][Row] == Board[Column + 2][Row + 2] and Board[Column][Row] != ' ':
 XOrOHasWon = True

 # diagonal up - 3 consecutive symbols the same and not empty
 # Row is inverse of Column
 for Column in range(1, sizeOfBoard - 1):
 for Row in range(3, sizeOfBoard + 1):
 if Board[Column][Row] == Board[Column + 1][Row - 1] and Board[Column][Row] == Board[Column + 2][Row - 2] and Board[Column][Row] != ' ':
 XOrOHasWon = True

 return XOrOHasWon

def GetWhoStarts():
 RandomNo = random.randint(0, 100)
 if (RandomNo % 2) == 0:
 WhoStarts = 'X'
 else:
 WhoStarts = 'O'
 return WhoStarts

def EnterPlayerName():
 playerOneName = ''
 playerTwoName = ''
 playingComputer = False
 while playerOneName == playerTwoName:
 playerOneName = input('What is the name of player one? ')

 # 10) Determine if wish to play the computer
 playComputer = ''
 while playComputer not in ['Y', 'N']:
 playComputer = input('Do you wish to play the computer (Y or N)? ').upper()
 if playComputer not in ['Y', 'N']:
 print('You must input Y or N')

 if playComputer == 'Y':
 playerTwoName = 'Computer'
 else:
 playerTwoName = input('What is the name of player two? ')

 if playerOneName == playerTwoName:
 print("Players' names must be different!\n")

 if playComputer == 'Y':
 playingComputer = True

 return playerOneName, playerTwoName, playingComputer

if \_\_name\_\_ == "\_\_main\_\_":
 PlayerOneName, PlayerTwoName, PlayComputer = EnterPlayerName()

 while PlayerOneSymbol not in ['X', 'O']:
 # Choose player one's symbol
 PlayerOneSymbol = input(PlayerOneName + ' what symbol do you wish to use, X or O? ').upper()
 if PlayerOneSymbol not in ['X', 'O']:
 print('Symbol to play must be X or O')
 if PlayerOneSymbol == 'X':
 PlayerTwoSymbol = 'O'
 else:
 PlayerTwoSymbol = 'X'
 StartSymbol = GetWhoStarts()
 while Answer not in ['N', 'n']: # Play a game
 NoOfMoves = 0;
 GameHasBeenDrawn = False;
 GameHasBeenWon = False;
 ClearBoard(Board);
 print("\n")
 DisplayBoard(Board);
 if StartSymbol == PlayerOneSymbol:
 print(PlayerOneName + " starts playing " + StartSymbol)
 else:
 print(PlayerTwoName + " starts playing " + StartSymbol)
 print("\n")
 CurrentSymbol = StartSymbol
 while (not GameHasBeenWon) and (not GameHasBeenDrawn): # Play until a player wins or the game is drawn
 # 5) get player name based on current symbol
 if CurrentSymbol == PlayerOneSymbol:
 print(PlayerOneName + "'s turn")
 else:
 print(PlayerTwoName + "'s turn")

 if PlayComputer and CurrentSymbol == PlayerTwoSymbol:
 XCoord, YCoord = GetRandomCoordinates()
 else:
 ValidMove = False;
 while not ValidMove: # Get a valid move
 XCoord, YCoord = GetMoveCoordinates()
 ValidMove = CheckValidMove(XCoord, YCoord, Board)
 if not ValidMove:
 print("Coordinates invalid, please try again")

 Board[XCoord][YCoord] = CurrentSymbol
 DisplayBoard(Board)
 GameHasBeenWon = CheckXOr0HasWon(Board)
 NoOfMoves += 1
 if not GameHasBeenWon:
 # 8) Updated for 5x5 game
 if NoOfMoves == 25: # Check if maximum number of allowed moves has been reached
 GameHasBeenDrawn = True
 else:
 if (CurrentSymbol == 'X'):
 CurrentSymbol = 'O'
 else:
 CurrentSymbol = 'X'
 if GameHasBeenWon: # Update scores and display result
 if (PlayerOneSymbol == CurrentSymbol):
 print(PlayerOneName + " congratulations you win!")
 PlayerOneScore += 1
 else:
 print(PlayerTwoName + " congratulations you win!")
 PlayerTwoScore += 1
 else:
 print("A draw this time!")
 # 9) Add 0.5 to both players' scores if a draw
 PlayerOneScore += 0.5
 PlayerTwoScore += 0.5
 print("\n")
 print(PlayerOneName + ", your score is: " + str(PlayerOneScore))
 print(PlayerTwoName + ", your score is: " + str(PlayerTwoScore))
 print();
 if (StartSymbol == PlayerOneSymbol):
 StartSymbol = PlayerTwoSymbol
 else:
 StartSymbol = PlayerOneSymbol;
 Answer = input("Another game Y/N? ")