# Implement Adversarial Search and Games:

**Minimax Algorithm**

**Tool**: Python

**Libraries Used**: math, random, platform, time and os

**Sample Problem**:

**Tic-tac-toe** is a game for 2 players, *X* and *O*, who take turns marking the spaces in a 3×3 grid. The player who succeeds in placing three of their marks in a horizontal, vertical, or diagonal row wins the game. An AI would be developed using Minimax algorithm to play the game optimally against a live human.

**Input Type**:

1. Choice of indicator (X or O)
2. Choice of first move. (y/n) where y signifies that Human wants to go first and n signifies that AI/Computer goes first.
3. Number between 1 and 9 signifying positions on a 3\*3 grid corresponding to the follow number to coordinate mappings.

1: [0, 0], 2: [0, 1], 3: [0, 2],  
4: [1, 0], 5: [1, 1], 6: [1, 2],  
7: [2, 0], 8: [2, 1], 9: [2, 2],

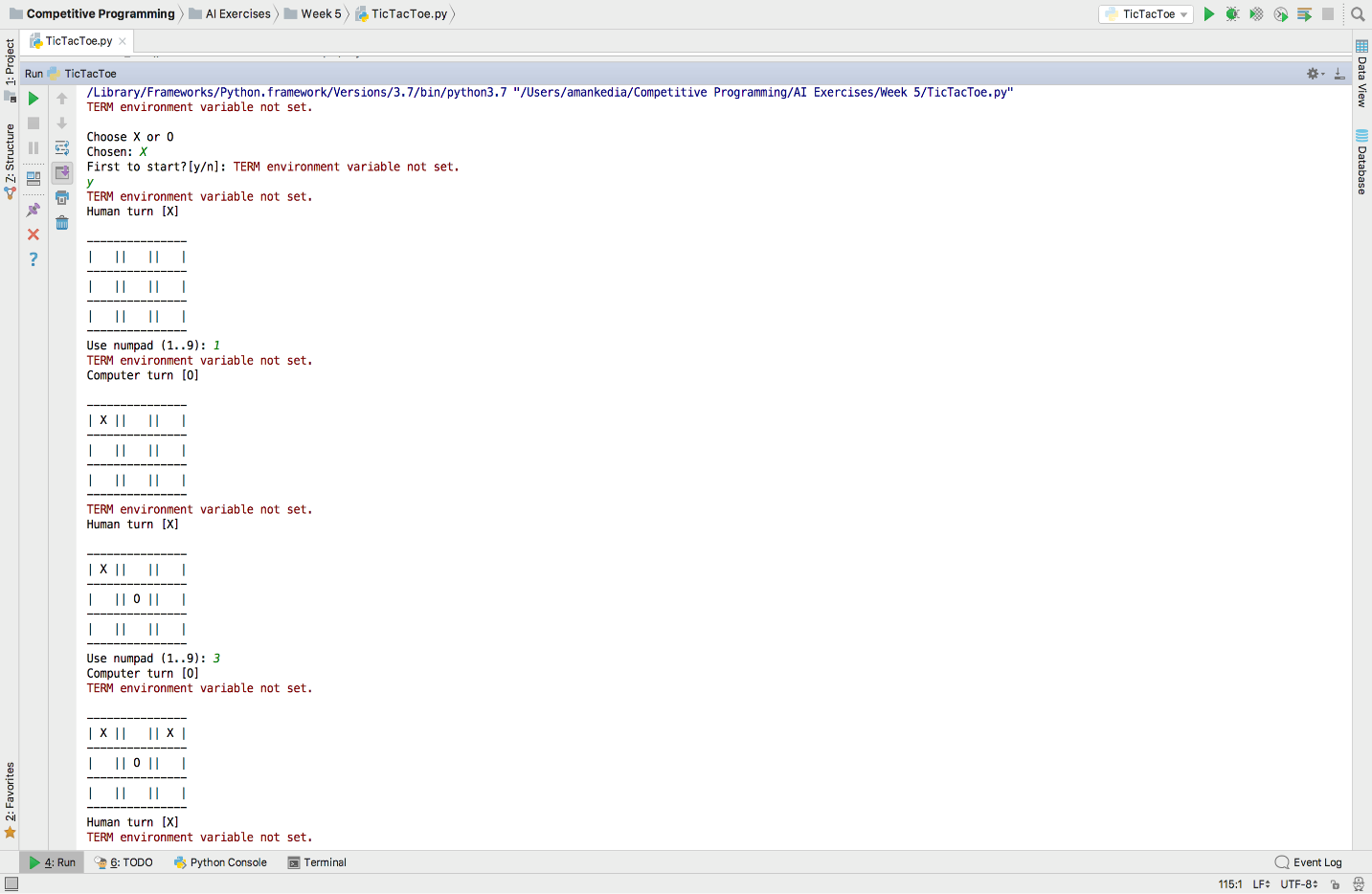
**Search / Game Playing Technique:** Minimax algorithm

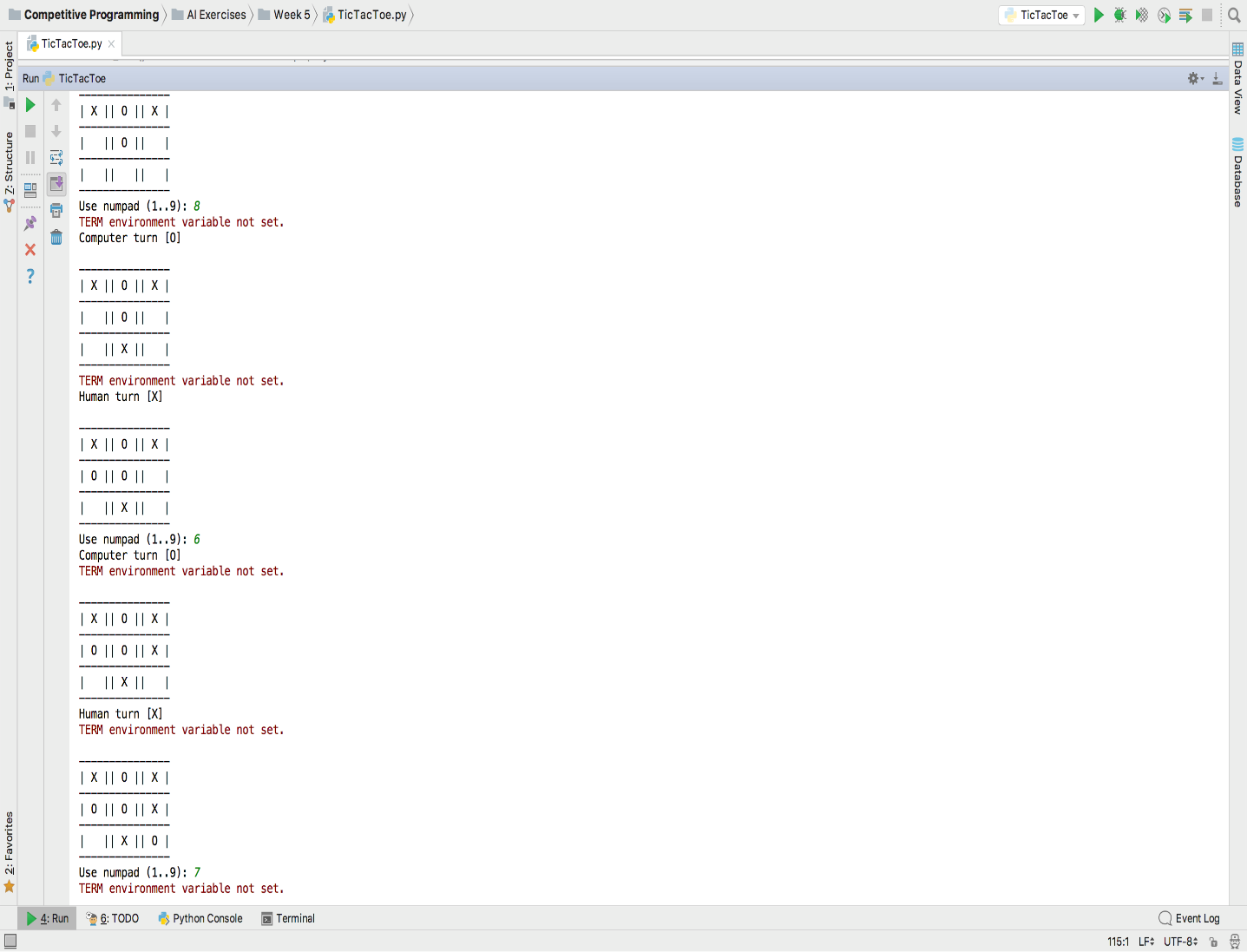
**Output Type**:

1. 3\*3 board representing state at every point in time
2. A final message in terms of who or whether it was a draw.

**Implementation**

*#!/usr/bin/env python3***from** math **import** inf **as** infinity  
**from** random **import** choice  
**import** platform  
**import** time  
**from** os **import** system  
  
**"""  
An implementation of Minimax AI Algorithm in Tic Tac Toe,  
using Python.  
"""**HUMAN = -1  
COMP = +1  
board = [  
 [0, 0, 0],  
 [0, 0, 0],  
 [0, 0, 0],  
]  
  
  
**def** evaluate(state):  
 *"""  
 Function for heuristic evaluation of state.* **:param** *state: the state of the current board* **:return***: +1 if the computer wins; -1 if the human wins; 0 draw  
 """* **if** wins(state, COMP):  
 score = +1  
 **elif** wins(state, HUMAN):  
 score = -1  
 **else**:  
 score = 0  
  
 **return** score  
  
  
**def** wins(state, player):  
 *"""  
 This function tests if a specific player wins. Possibilities:  
 \* Three rows [X X X] or [O O O]  
 \* Three cols [X X X] or [O O O]  
 \* Two diagonals [X X X] or [O O O]* **:param** *state: the state of the current board* **:param** *player: a human or a computer* **:return***: True if the player wins  
 """* win\_state = [  
 [state[0][0], state[0][1], state[0][2]],  
 [state[1][0], state[1][1], state[1][2]],  
 [state[2][0], state[2][1], state[2][2]],  
 [state[0][0], state[1][0], state[2][0]],  
 [state[0][1], state[1][1], state[2][1]],  
 [state[0][2], state[1][2], state[2][2]],  
 [state[0][0], state[1][1], state[2][2]],  
 [state[2][0], state[1][1], state[0][2]],  
 ]  
 **if** [player, player, player] **in** win\_state:  
 **return True  
 else**:  
 **return False  
  
  
def** game\_over(state):  
 *"""  
 This function test if the human or computer wins* **:param** *state: the state of the current board* **:return***: True if the human or computer wins  
 """* **return** wins(state, HUMAN) **or** wins(state, COMP)  
  
  
**def** empty\_cells(state):  
 *"""  
 Each empty cell will be added into cells' list* **:param** *state: the state of the current board* **:return***: a list of empty cells  
 """* cells = []  
  
 **for** x, row **in** enumerate(state):  
 **for** y, cell **in** enumerate(row):  
 **if** cell == 0:  
 cells.append([x, y])  
  
 **return** cells  
  
  
**def** valid\_move(x, y):  
 *"""  
 A move is valid if the chosen cell is empty* **:param** *x: X coordinate* **:param** *y: Y coordinate* **:return***: True if the board[x][y] is empty  
 """* **if** [x, y] **in** empty\_cells(board):  
 **return True  
 else**:  
 **return False  
  
  
def** set\_move(x, y, player):  
 *"""  
 Set the move on board, if the coordinates are valid* **:param** *x: X coordinate* **:param** *y: Y coordinate* **:param** *player: the current player  
 """* **if** valid\_move(x, y):  
 board[x][y] = player  
 **return True  
 else**:  
 **return False  
  
  
def** minimax(state, depth, player):  
 *"""  
 AI function that choice the best move* **:param** *state: current state of the board* **:param** *depth: node index in the tree (0 <= depth <= 9),  
 but never nine in this case (see iaturn() function)* **:param** *player: an human or a computer* **:return***: a list with [the best row, best col, best score]  
 """* **if** player == COMP:  
 best = [-1, -1, -infinity]  
 **else**:  
 best = [-1, -1, +infinity]  
  
 **if** depth == 0 **or** game\_over(state):  
 score = evaluate(state)  
 **return** [-1, -1, score]  
  
 **for** cell **in** empty\_cells(state):  
 x, y = cell[0], cell[1]  
 state[x][y] = player  
 score = minimax(state, depth - 1, -player)  
 state[x][y] = 0  
 score[0], score[1] = x, y  
  
 **if** player == COMP:  
 **if** score[2] > best[2]:  
 best = score *# max value* **else**:  
 **if** score[2] < best[2]:  
 best = score *# min value* **return** best  
  
  
**def** clean():  
 *"""  
 Clears the console  
 """* os\_name = platform.system().lower()  
 **if 'windows' in** os\_name:  
 system(**'cls'**)  
 **else**:  
 system(**'clear'**)  
  
  
**def** render(state, c\_choice, h\_choice):  
 *"""  
 Print the board on console* **:param** *state: current state of the board  
 """* chars = {  
 -1: h\_choice,  
 +1: c\_choice,  
 0: **' '** }  
 str\_line = **'---------------'** print(**'\n'** + str\_line)  
 **for** row **in** state:  
 **for** cell **in** row:  
 symbol = chars[cell]  
 print(**f'| {symbol} |'**, end=**''**)  
 print(**'\n'** + str\_line)  
  
  
**def** ai\_turn(c\_choice, h\_choice):  
 *"""  
 It calls the minimax function if the depth < 9,  
 else it choices a random coordinate.* **:param** *c\_choice: computer's choice X or O* **:param** *h\_choice: human's choice X or O* **:return***:  
 """* depth = len(empty\_cells(board))  
 **if** depth == 0 **or** game\_over(board):  
 **return** clean()  
 print(**f'Computer turn [{c\_choice}]'**)  
 render(board, c\_choice, h\_choice)  
  
 **if** depth == 9:  
 x = choice([0, 1, 2])  
 y = choice([0, 1, 2])  
 **else**:  
 move = minimax(board, depth, COMP)  
 x, y = move[0], move[1]  
  
 set\_move(x, y, COMP)  
 time.sleep(1)  
  
  
**def** human\_turn(c\_choice, h\_choice):  
 *"""  
 The Human plays choosing a valid move.* **:param** *c\_choice: computer's choice X or O* **:param** *h\_choice: human's choice X or O* **:return***:  
 """* depth = len(empty\_cells(board))  
 **if** depth == 0 **or** game\_over(board):  
 **return** *# Dictionary of valid moves* move = -1  
 moves = {  
 1: [0, 0], 2: [0, 1], 3: [0, 2],  
 4: [1, 0], 5: [1, 1], 6: [1, 2],  
 7: [2, 0], 8: [2, 1], 9: [2, 2],  
 }  
  
 clean()  
 print(**f'Human turn [{h\_choice}]'**)  
 render(board, c\_choice, h\_choice)  
  
 **while** move < 1 **or** move > 9:  
 **try**:  
 move = int(input(**'Use numpad (1..9): '**))  
 coord = moves[move]  
 can\_move = set\_move(coord[0], coord[1], HUMAN)  
  
 **if not** can\_move:  
 print(**'Bad move'**)  
 move = -1  
 **except** (EOFError, KeyboardInterrupt):  
 print(**'Bye'**)  
 exit()  
 **except** (KeyError, ValueError):  
 print(**'Bad choice'**)  
  
  
**def** main():  
 *"""  
 Main function that calls all functions  
 """* clean()  
 h\_choice = **''** *# X or O* c\_choice = **''** *# X or O* first = **''** *# if human is the first  
  
 # Human chooses X or O to play* **while** h\_choice != **'O' and** h\_choice != **'X'**:  
 **try**:  
 print(**''**)  
 h\_choice = input(**'Choose X or O\nChosen: '**).upper()  
 **except** (EOFError, KeyboardInterrupt):  
 print(**'Bye'**)  
 exit()  
 **except** (KeyError, ValueError):  
 print(**'Bad choice'**)  
  
 *# Setting computer's choice* **if** h\_choice == **'X'**:  
 c\_choice = **'O'  
 else**:  
 c\_choice = **'X'** *# Human may starts first* clean()  
 **while** first != **'Y' and** first != **'N'**:  
 **try**:  
 first = input(**'First to start?[y/n]: '**).upper()  
 **except** (EOFError, KeyboardInterrupt):  
 print(**'Bye'**)  
 exit()  
 **except** (KeyError, ValueError):  
 print(**'Bad choice'**)  
  
 *# Main loop of this game* **while** len(empty\_cells(board)) > 0 **and not** game\_over(board):  
 **if** first == **'N'**:  
 ai\_turn(c\_choice, h\_choice)  
 first = **''** human\_turn(c\_choice, h\_choice)  
 ai\_turn(c\_choice, h\_choice)  
  
 *# Game over message* **if** wins(board, HUMAN):  
 clean()  
 print(**f'Human turn [{h\_choice}]'**)  
 render(board, c\_choice, h\_choice)  
 print(**'YOU WIN!'**)  
 **elif** wins(board, COMP):  
 clean()  
 print(**f'Computer turn [{c\_choice}]'**)  
 render(board, c\_choice, h\_choice)  
 print(**'YOU LOSE!'**)  
 **else**:  
 clean()  
 render(board, c\_choice, h\_choice)  
 print(**'DRAW!'**)  
  
 exit()  
  
  
**if** \_\_name\_\_ == **'\_\_main\_\_'**:  
 main()

**Screenshot/Output**





**Lab Exercises**

1. Develop Tic Tac Toe Game Playing AI using Alpha Beta Pruning.
2. Develop Connect Four Game Playing AI using Minimax Algorithm.