Connect 4 AI – Methods used and their use

FourConnect:

_game has 6 rows and 7 columns. Note that rows are filled up in descending order (Row 6 has to be filled before filling Row 5 for a particular column).

1. **_CoinRowAfterAction(self, action):** Returns the first empty row for the 'action' column
2. **_CanMyopicPlayerWin(self, row, col) / _CanGameTreePlayerWin(self, row, col):** Checks if the respective player is winning in the current game. Both call the function:
    a. **_CanAPlayerWin(self, row, col, player):** Returns boolean value if 'player' can win in the current game state. Calls three methods:
        i. **_CheckHorizontal(self, row, col, player)**
        ii. **_CheckVertical(self,row,col,player)**
        iii. **_CheckDiag(self,row,col,player,diag=1):** 1 for principal & -1 for secondary diagonal
3. **MyopicPlayerAction(self):** Takes action for Myopic Player. Calls:
    a. **_FindBestMyopicAction(self):** Returns the best move/action Myopic Player can take. Calls:
        i. **_FindMyopicMoves(self):** checks all the possible moves the myopic player can make and categorizes them into: valid actions, losing actions, winning actions and blocking actions.
4. **GameTreePlayerAction(self,action):** Takes 'action' for GameTree Player.
5. **_TakeAction(self, action, player):** Fills the board with a 'player' coin after finding the row for the given 'action' column and checks if the action taken results in a win.
6. Functions that do basic operations on Current Game State:
    a. **PrintGameState(self,state=None)**
    b. **GetCurrentState(self)**
    c. **SetCurrentState(self, gameState)**

PlayGame:

1. **makeMove(self, board, col, player):** Returns board, row and column after making a move for ‘player’ in column ‘col’
2. **get_valid_locations(self, currentState):** Returns list of columns with at least one empty cell.
3. **is_terminal_node(self,currentState):** Returns true if no valid moves left or either player wins.
    1. **winning_move(self, currentState, player):** Returns boolean value if ‘player’ is winning in board ‘currentState’ after checking all horizontals, verticals and diagonals.
4. **utilityValue(self, currentState, player = AI_PLAYER):** Returns utility value for ‘player’ of current game state by adding evaluation scores of all possible windows. Calls:
    1. **evaluation(self, window, player = AI_PLAYER):** Returns score for given ‘window’ for ‘player’
5. **FindBestAction(self, currentState)**: Finds the best action after setting a certain depth for Alpha Beta pruning.
    1. **MiniMaxAlphaBeta(self, currentState, depth, player = AI_PLAYER):** Performs alpha-beta pruning with depth ‘depth’
        1. **minimizeBeta(self, currentState, depth, alpha, beta, player, opponent)**
        2. **maximizeAlpha(self, currentState, depth, alpha, beta, player, opponent)**
        3. **winning_positions_heuristic(self, currentState, move, player):** Returns the number of winning positions after making a move. Used in move ordering heuristic.
            1. **count_winning_positions(self, currentState, player):** Returns total possible cells in ‘currentState’ that could give a win to ‘player’.
6. **PlayGame():** Playing the game- Myopic player always starts first.
7. **main():** Plays 50 games and computes the average number of moves, number of games won by AI and execution time.