## Artificial and Computational Intelligence

## Assignment – 2

Group: 114

|  |  |  |
| --- | --- | --- |
| **Name** | **BITS ID** | **Contribution** |
| Rahul Sharma | 2022AA05167 | 25% |
| Saurabh Sharma | 2022AA05383 | 25% |
| Palash Sharma | 2022AA05097 | 25% |
| Ravi Shankar Sharma | 2022AA05485 | 25% |

# **Question 1: Game Problem**

## **Code Implementation and Design Logic**

The code implements the Minimax algorithm to play a game of Tic-Tac-Toe. Here's a brief description of each function-

**count\_winning\_paths(player, board):** This is a helper function that takes the current player and board position as input and returns the number of winning paths for that player. It checks each row, column, and two diagonals for any 4 consecutive positions filled with the player's symbol.

**board\_fill(brd, player):** This is another helper function that takes the current board position and player as input and returns a new board position where all empty positions are filled with the player's symbol. This is used to count the winning paths for both players.

**static\_evaluation(board, max\_player, min\_player):** This function evaluates the current board position for the maximizing player and returns a score. It uses board\_fill() and count\_winning\_paths() functions to count the number of winning paths for both players, and returns the difference between the two scores.

**get\_possible\_moves(board):** This is a helper function that returns a list of all possible moves in the current game state. It checks for all empty positions on the board and returns them as tuples.

**make\_move(board, move, player):** This is another helper function that returns the new game state after making a move for the given player. It takes the current board position, move tuple, and player symbol as input and returns a new board position with the player's symbol added at the specified position.

**terminal\_state\_check(board, player):** This function checks whether the current state is a terminal state, i.e., whether any player has won or the game is tied. It checks for a win based on row, column, and two diagonal checks. It returns 'win' if a player has won, 'tie' if the game is tied, and 'running' if the game is still in progress.

**minimax(board, player, depth):** This is the main function that implements the Minimax algorithm. It takes the current board position, current player, and depth as input and returns the best move and score for the current player. It uses recursion to evaluate all possible moves up to a specified depth, to avoid computation complexity we have taken depth=3 in this implementation. It returns the move with the highest or lowest score, depending on whether the current player is the maximizing or minimizing player.

**play\_game():** This is the main function that starts the game. It initializes the game board and players, and alternates between the players to make moves until the game is over. It uses terminal\_state\_check() function to check whether the game is over, and minimax() function to get the best move for the computer player. It also displays the game board after each move and declares the winner at the end of the game.

## **Static Evaluation Function**

In this problem statement, we have simply taken computer as a max player having coin “T” and other human player as min player having coin “X”. To define static evaluation function, we did the following steps –

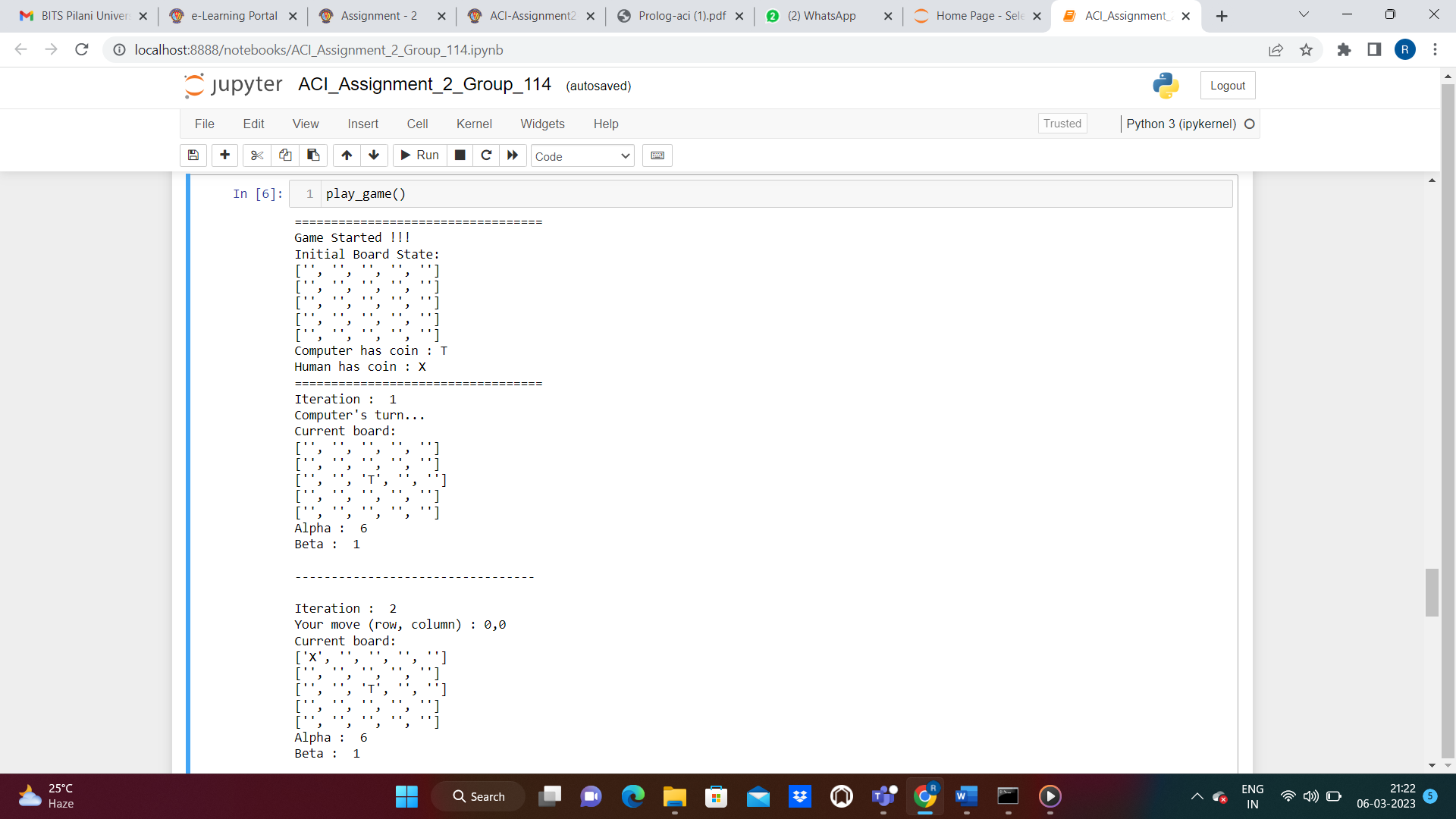
* We took the current board positions and filled all the empty board positions by max player’s coin “T”.
* Once the board is filled, we checked the win paths for max player. The win paths were calculated using below criteria –
* Combinations of 4 consecutive coins of max player (“T”) in rows.
* Combinations of 4 consecutive coins of max player (“T”) in columns.
* Combinations of 4 consecutive coins of max player (“T”) in both diagonals.
* Once total number of win paths calculated for max player, we took the original board (which was used to fill max player’s coin) and now we will empty board positions with min player’s coin “X”.
* Once the board is filled, we checked the win paths for min player. The win paths were calculated using below criteria –
* Combinations of 4 consecutive coins of min player (“X”) in rows.
* Combinations of 4 consecutive coins of min player (“X”) in columns.
* Combinations of 4 consecutive coins of min player (“X”) in both diagonals.
* Finally we do **max player’s total win paths - min player’s total win paths.** To calculate static evaluation value.

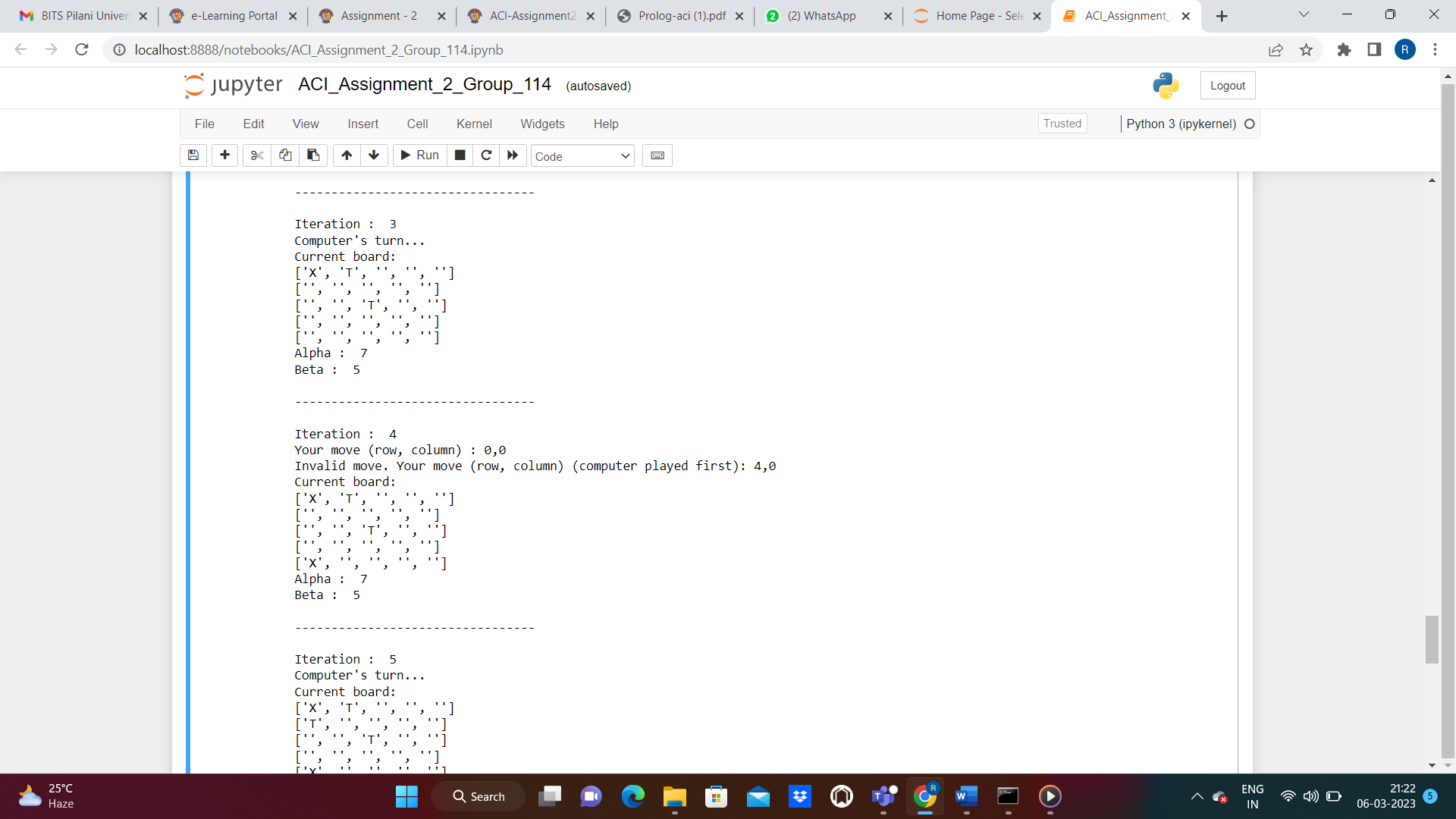
### **Justification**

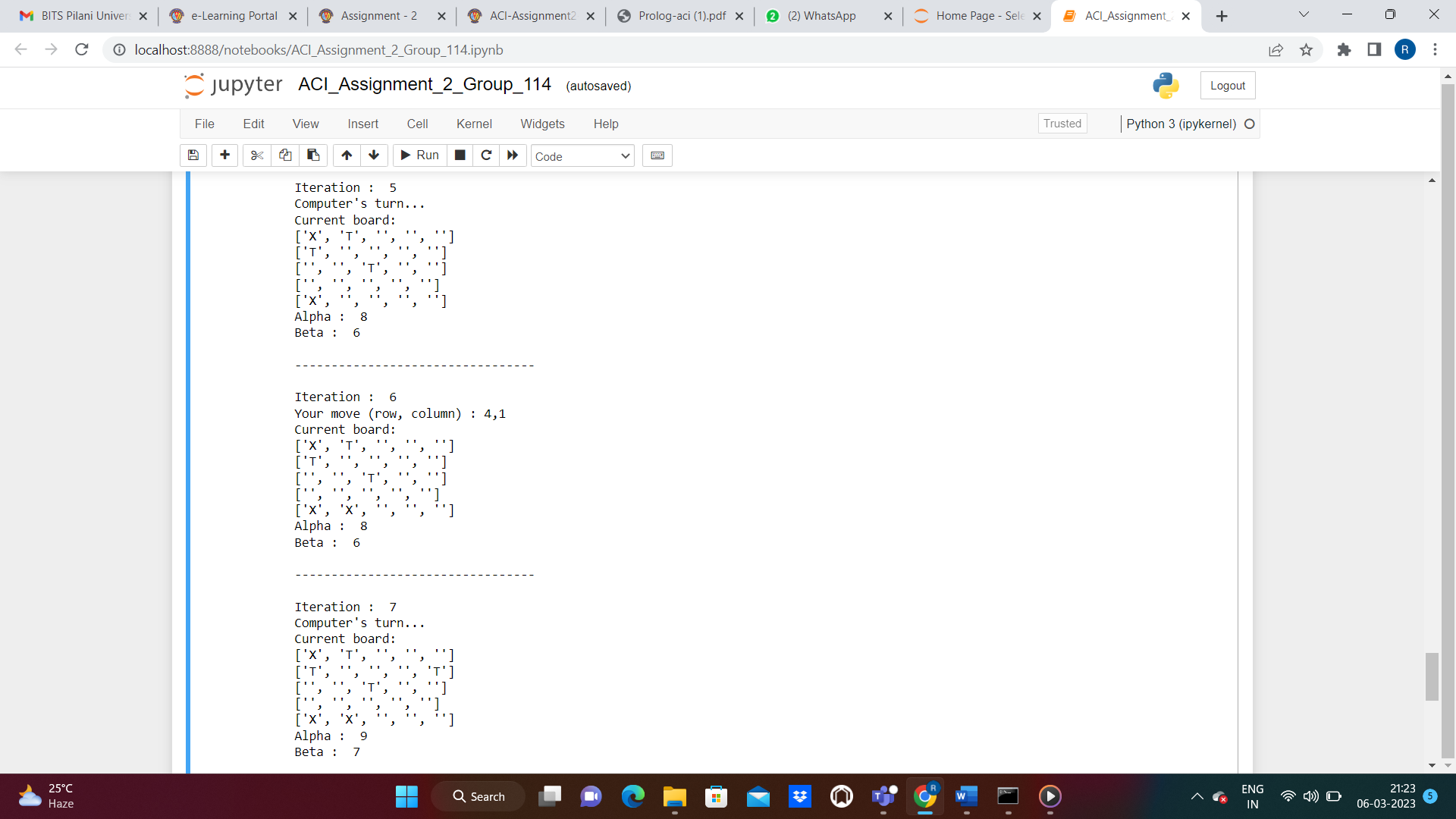
We have taken this static evaluation function to estimate how good the board position is with respect to max player. If the total win paths for max player is more than min player, we get positive number as static evaluation value and max player tries to maximize it. On the other hand min player tries to minimize it to increase its win chances.

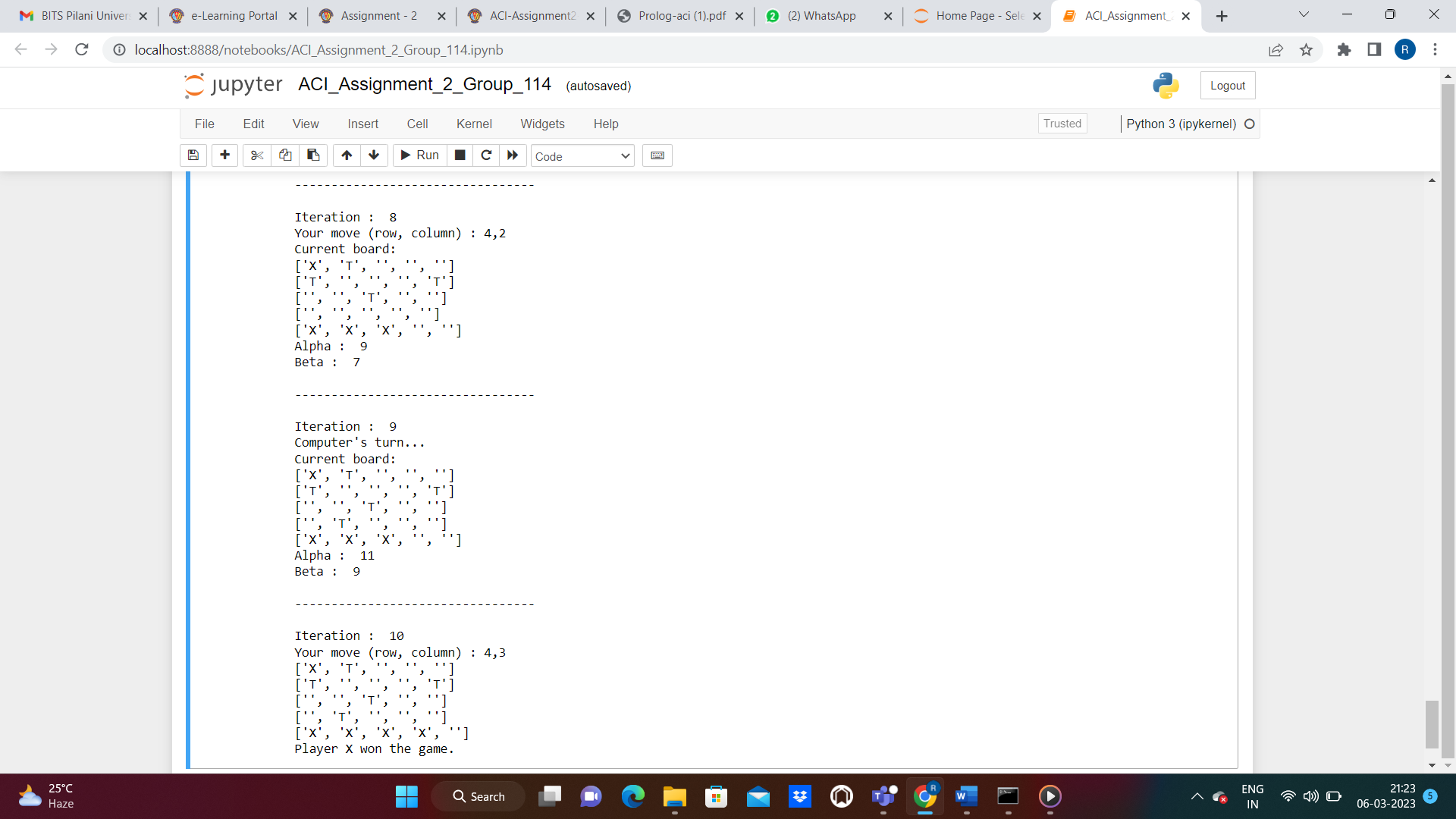
## **Screenshots**

### **Case 1: Human player win**

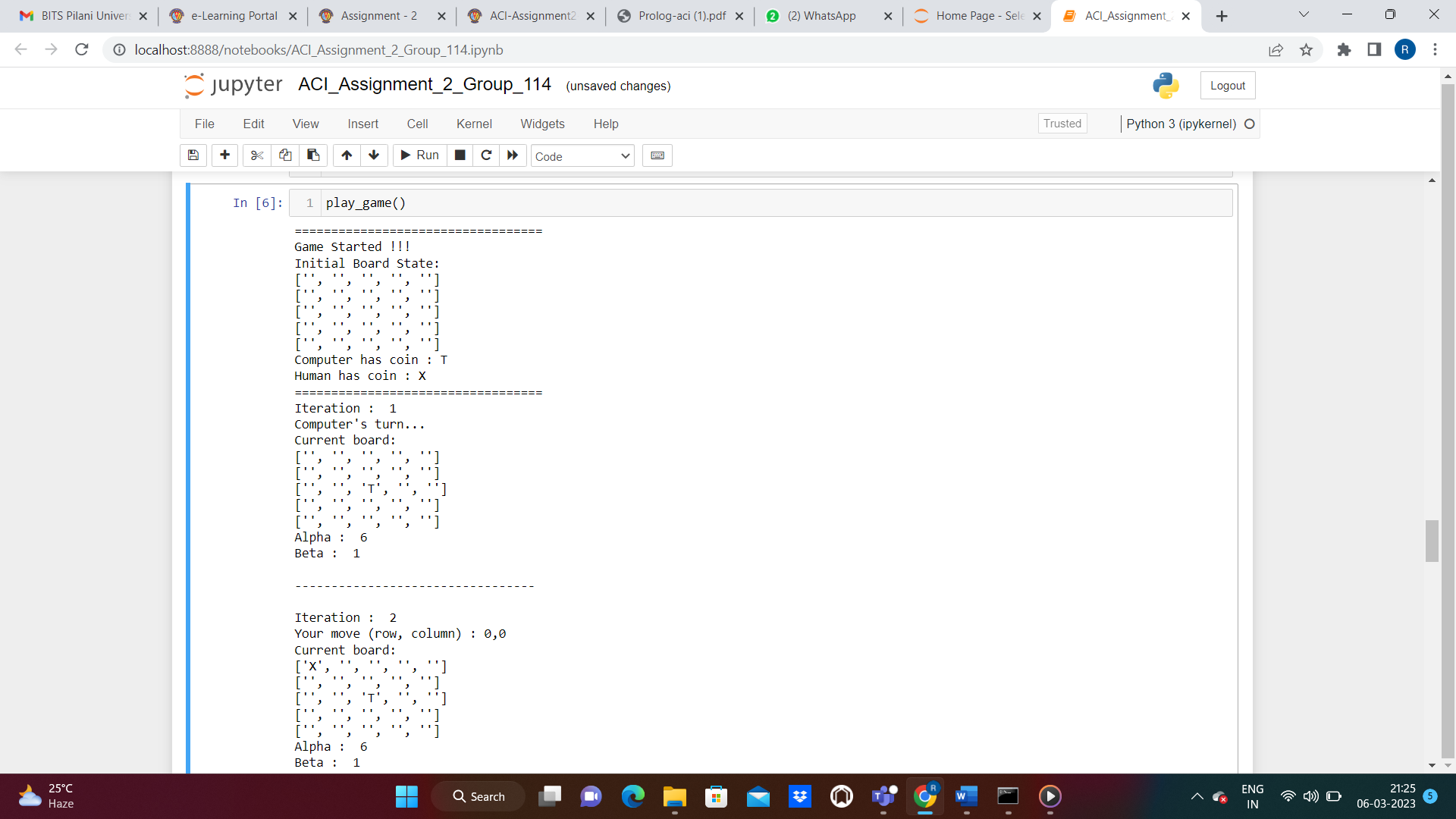


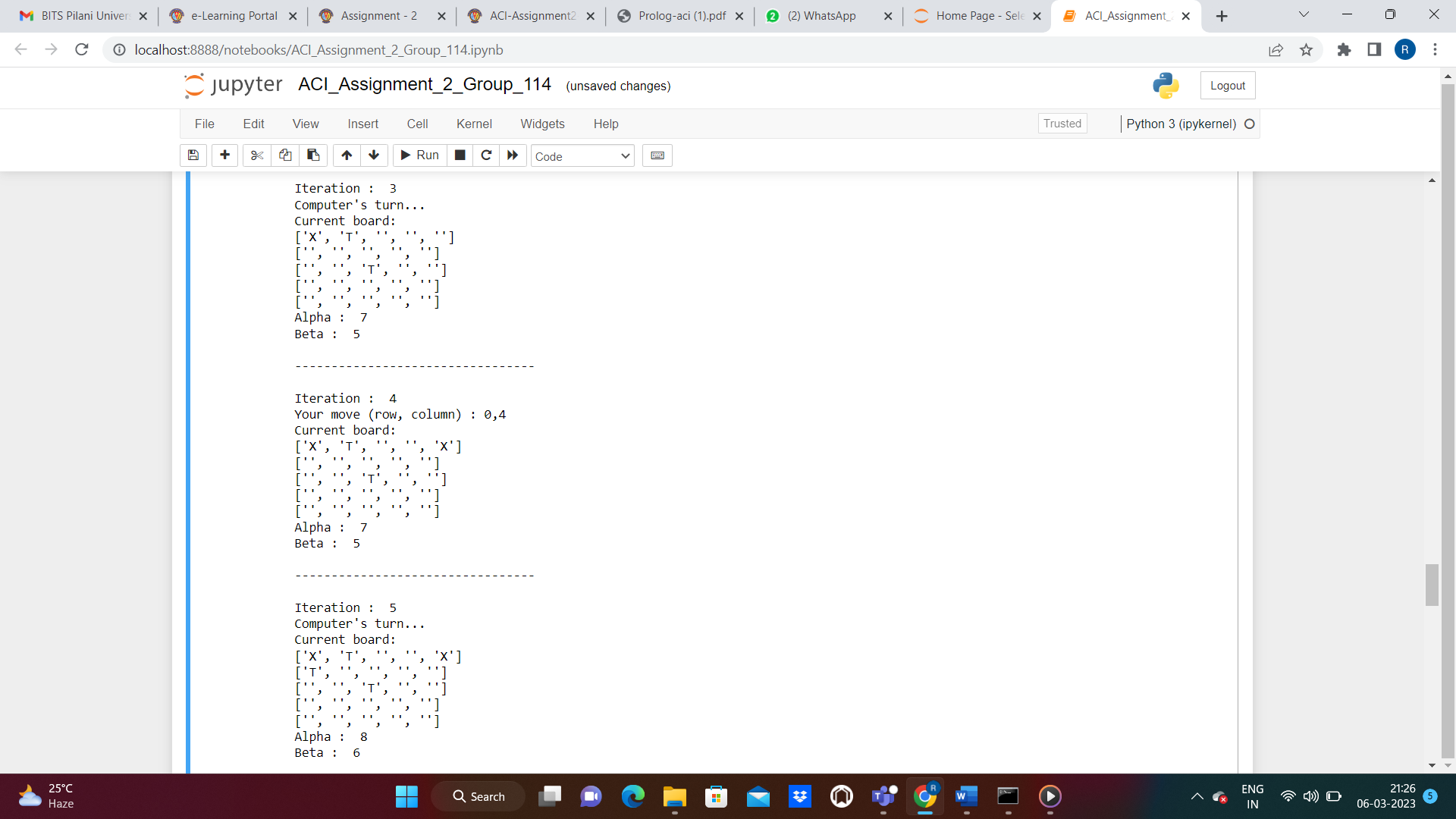


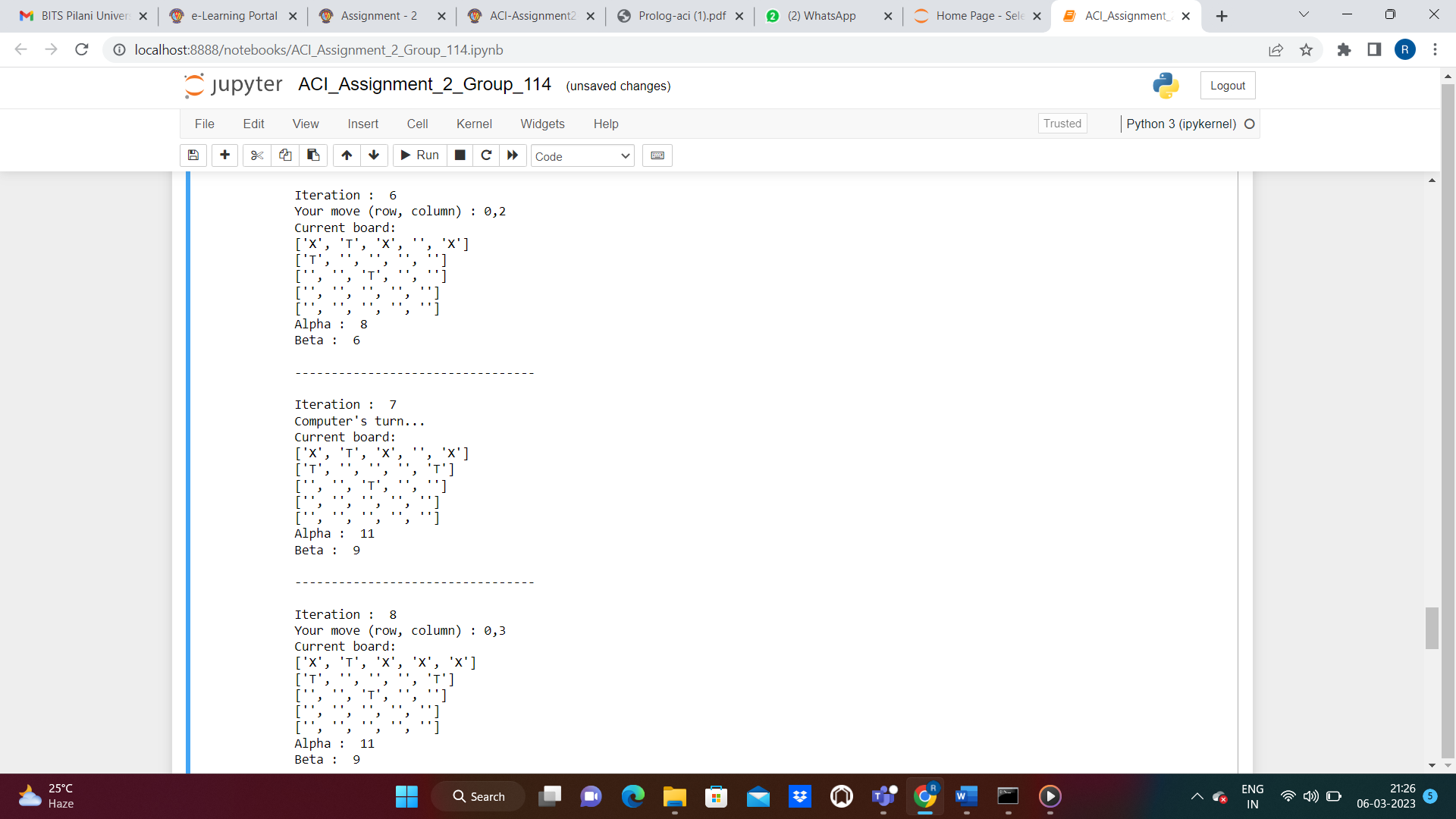


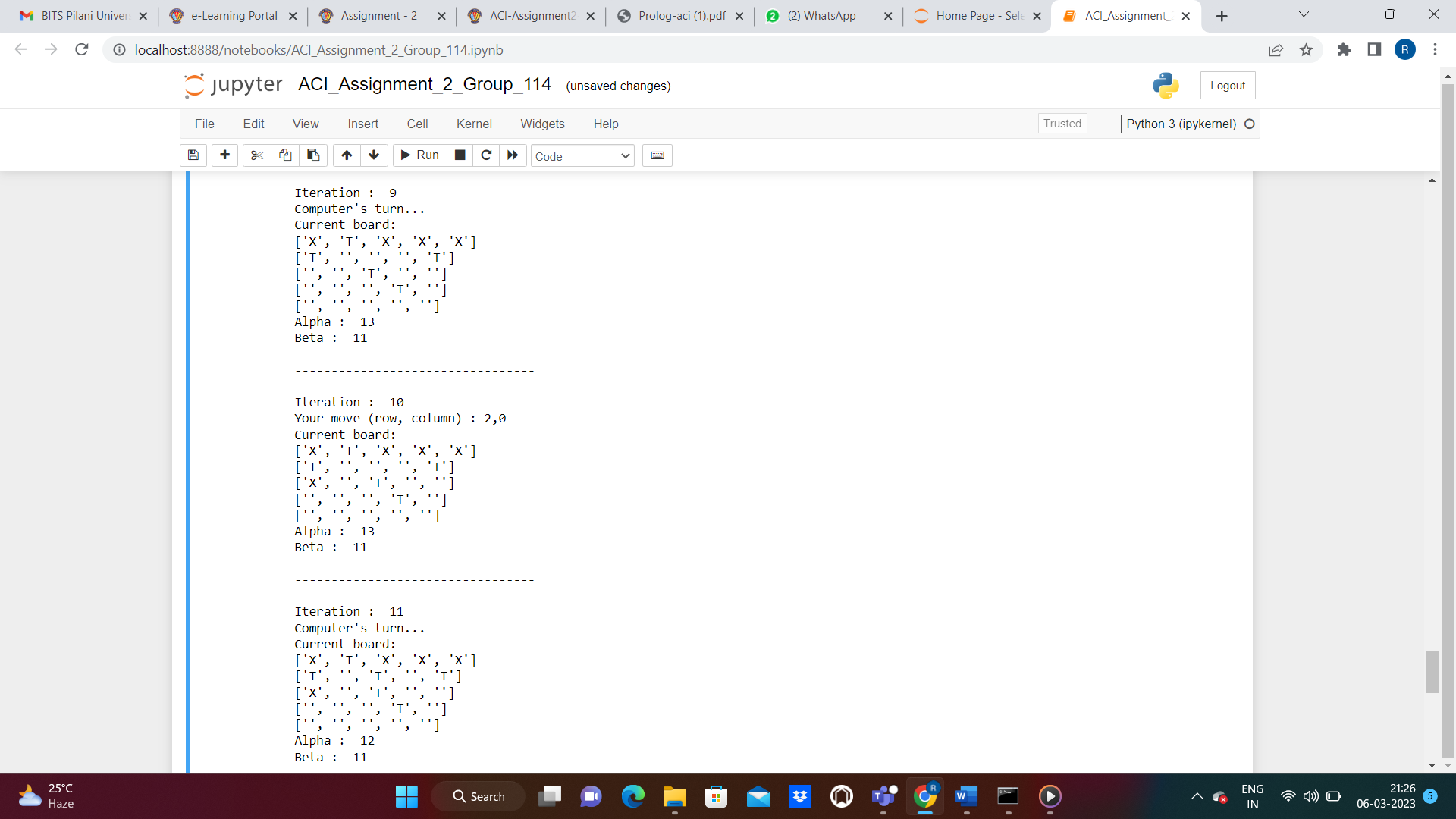


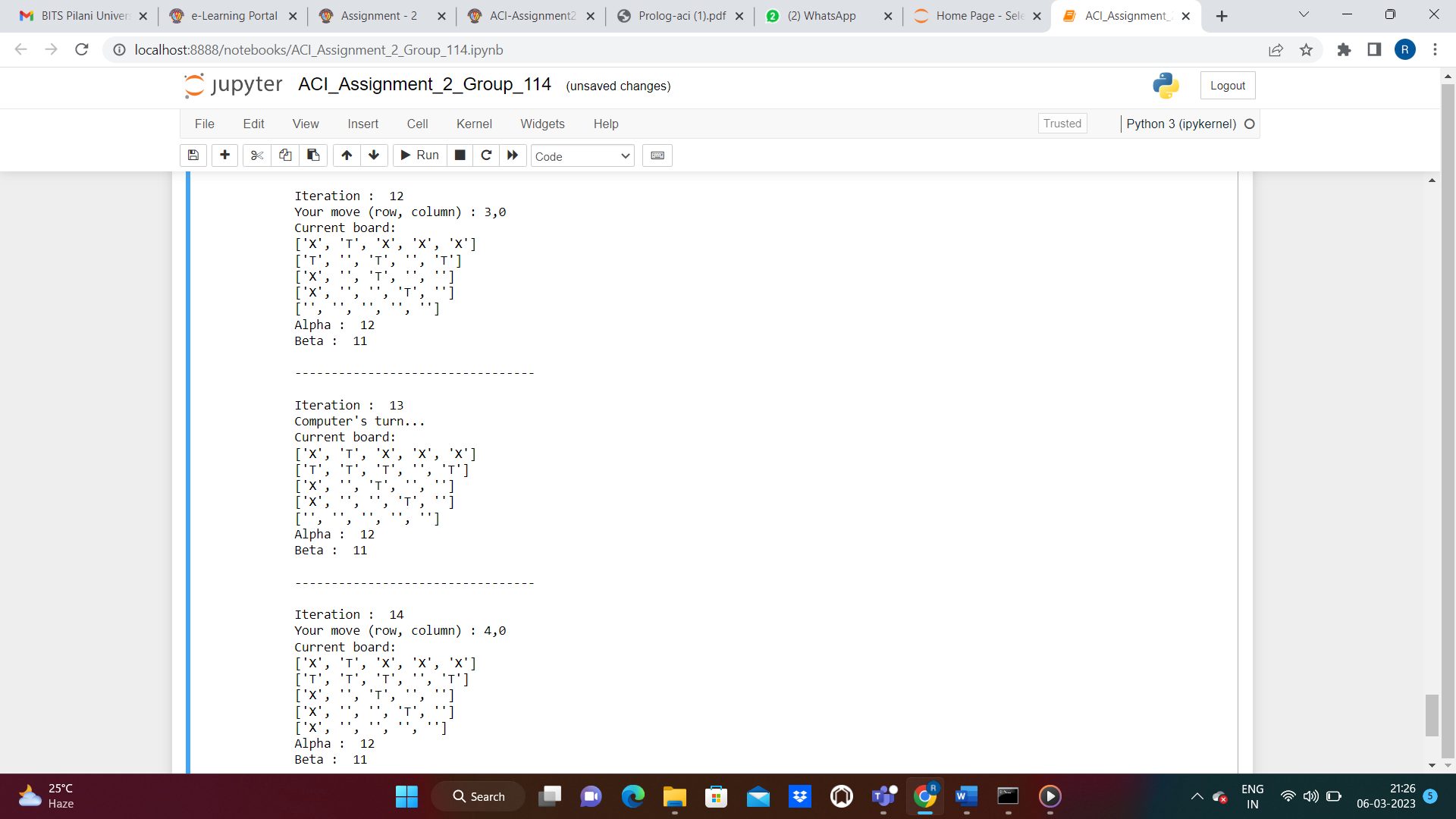
### **Case 2: Computer win**

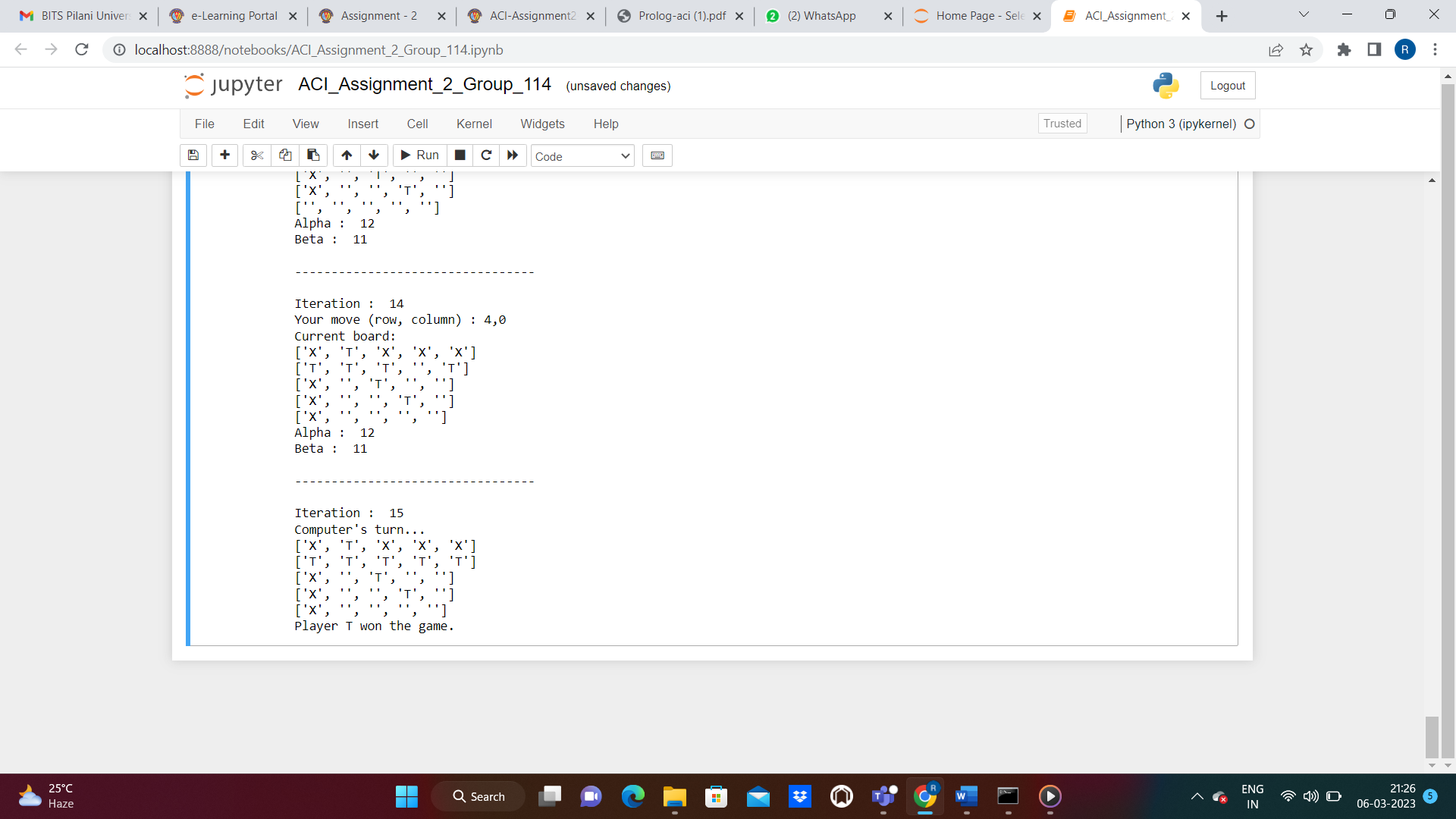




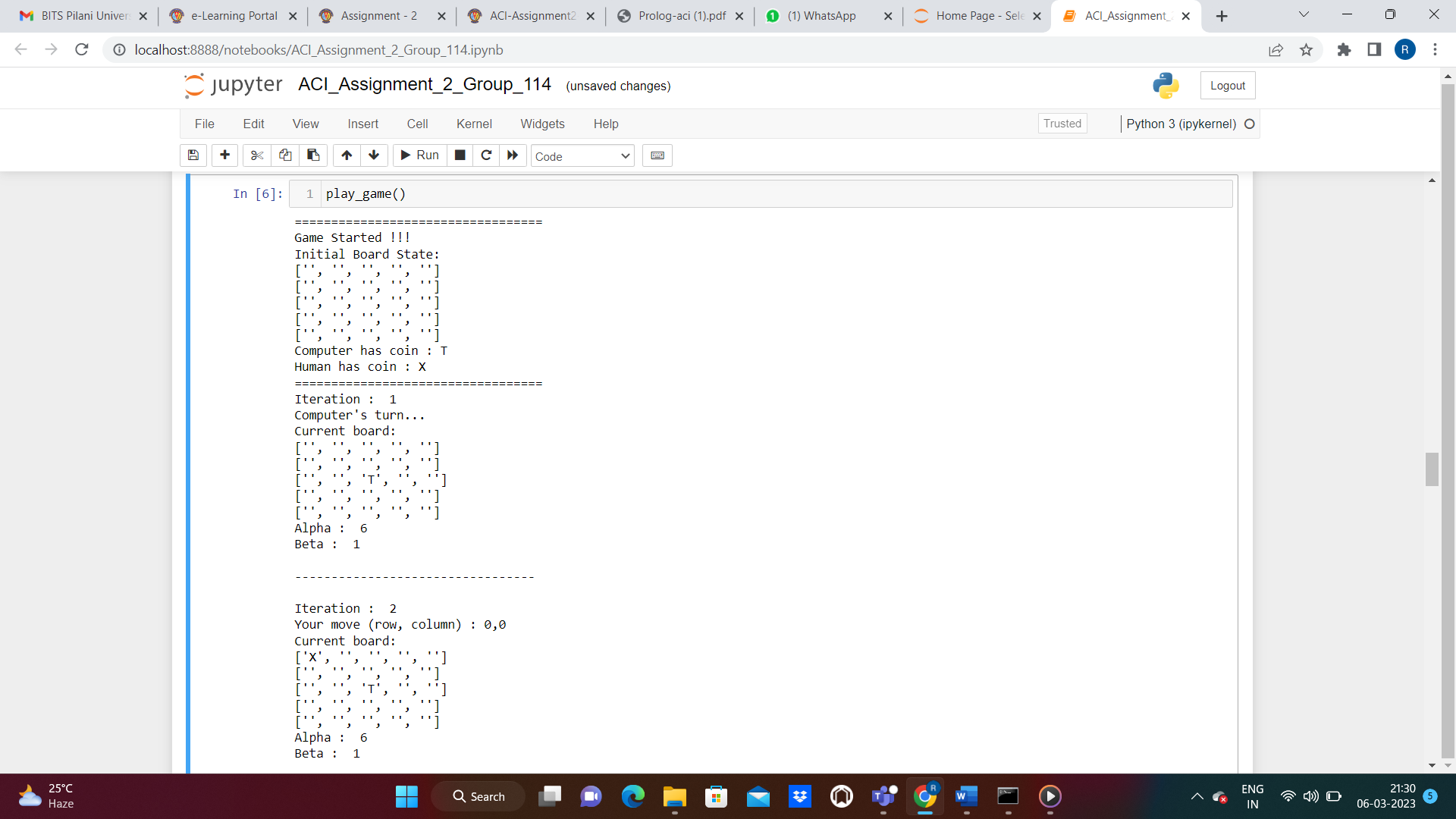




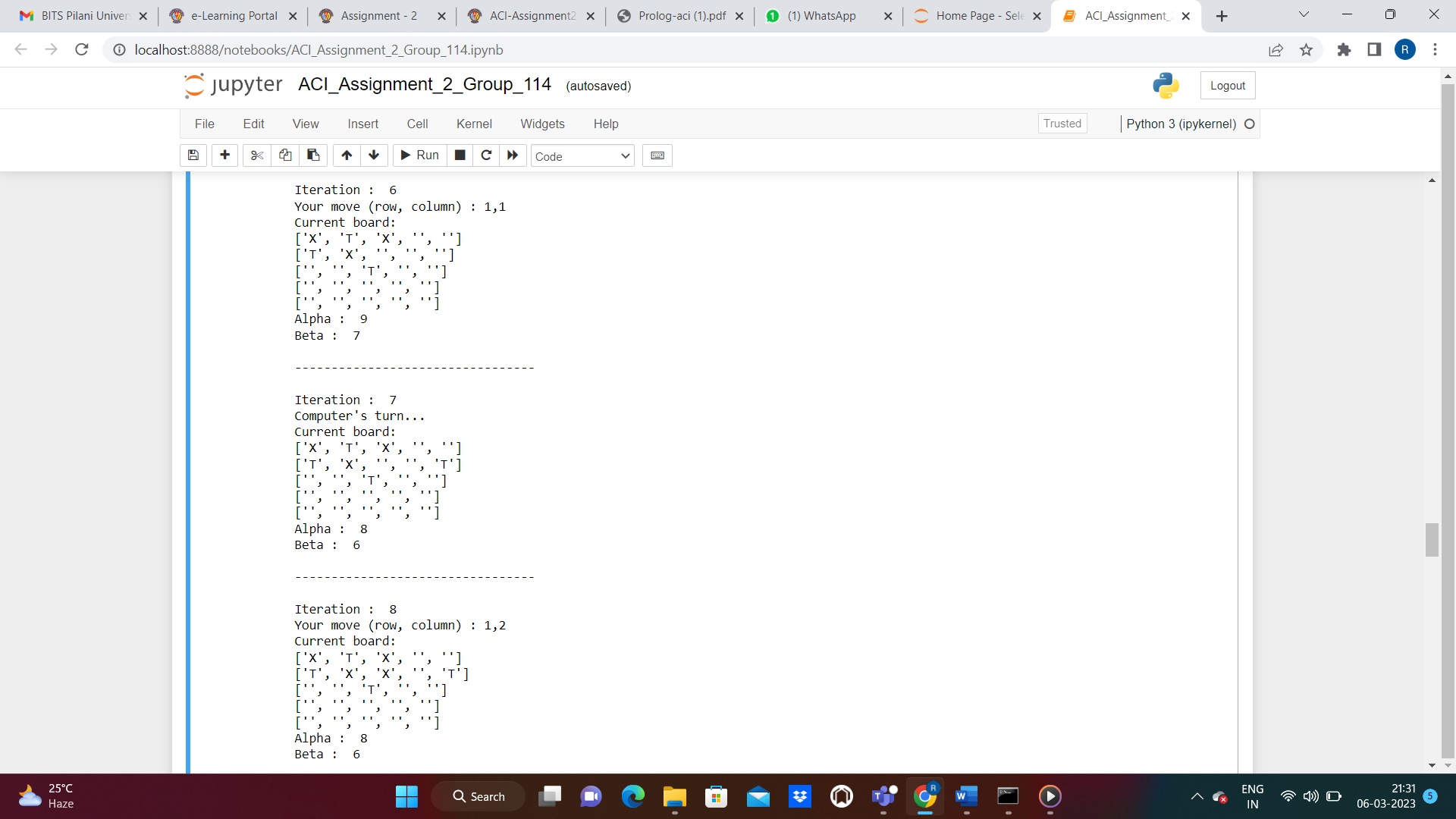


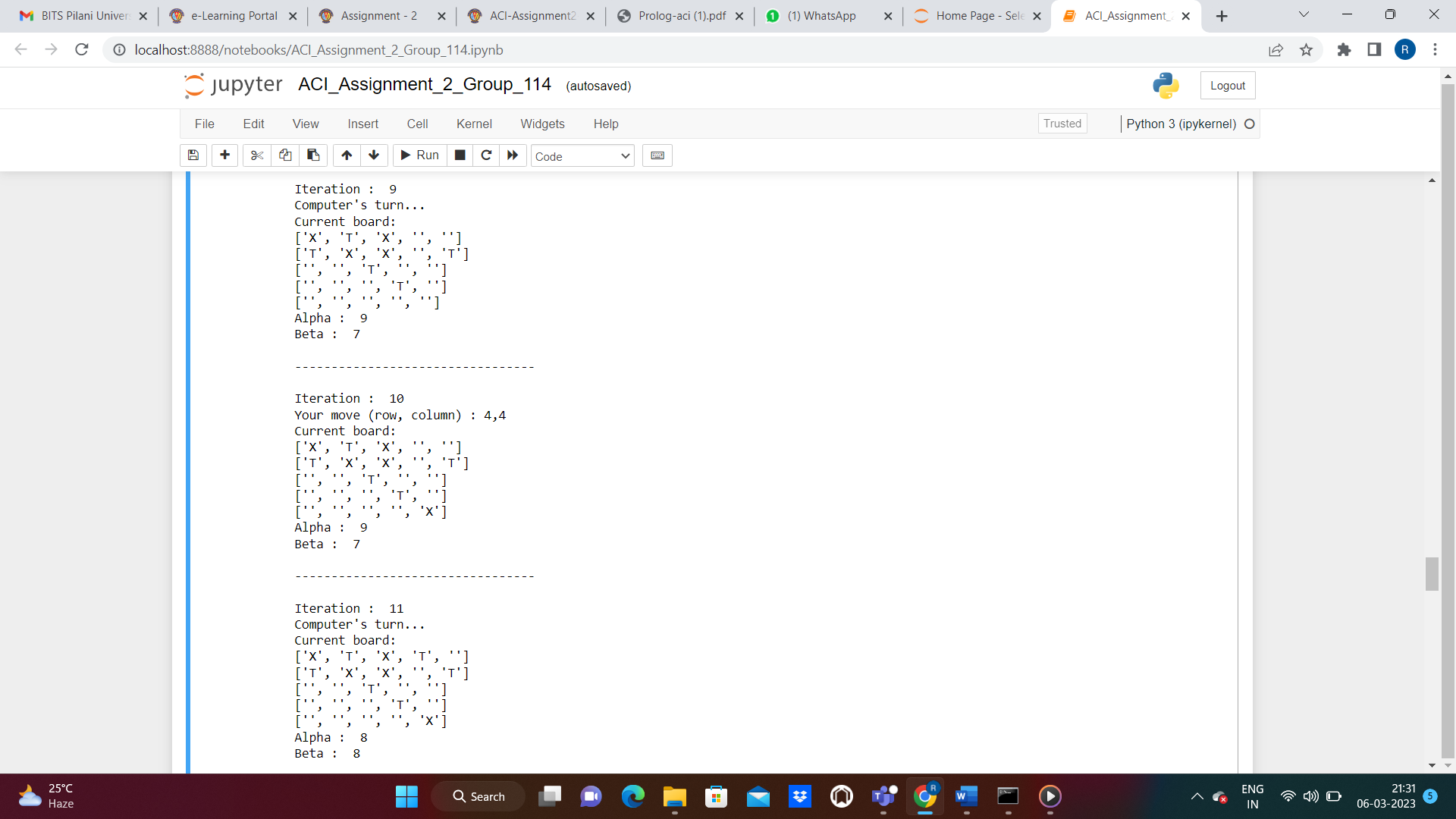


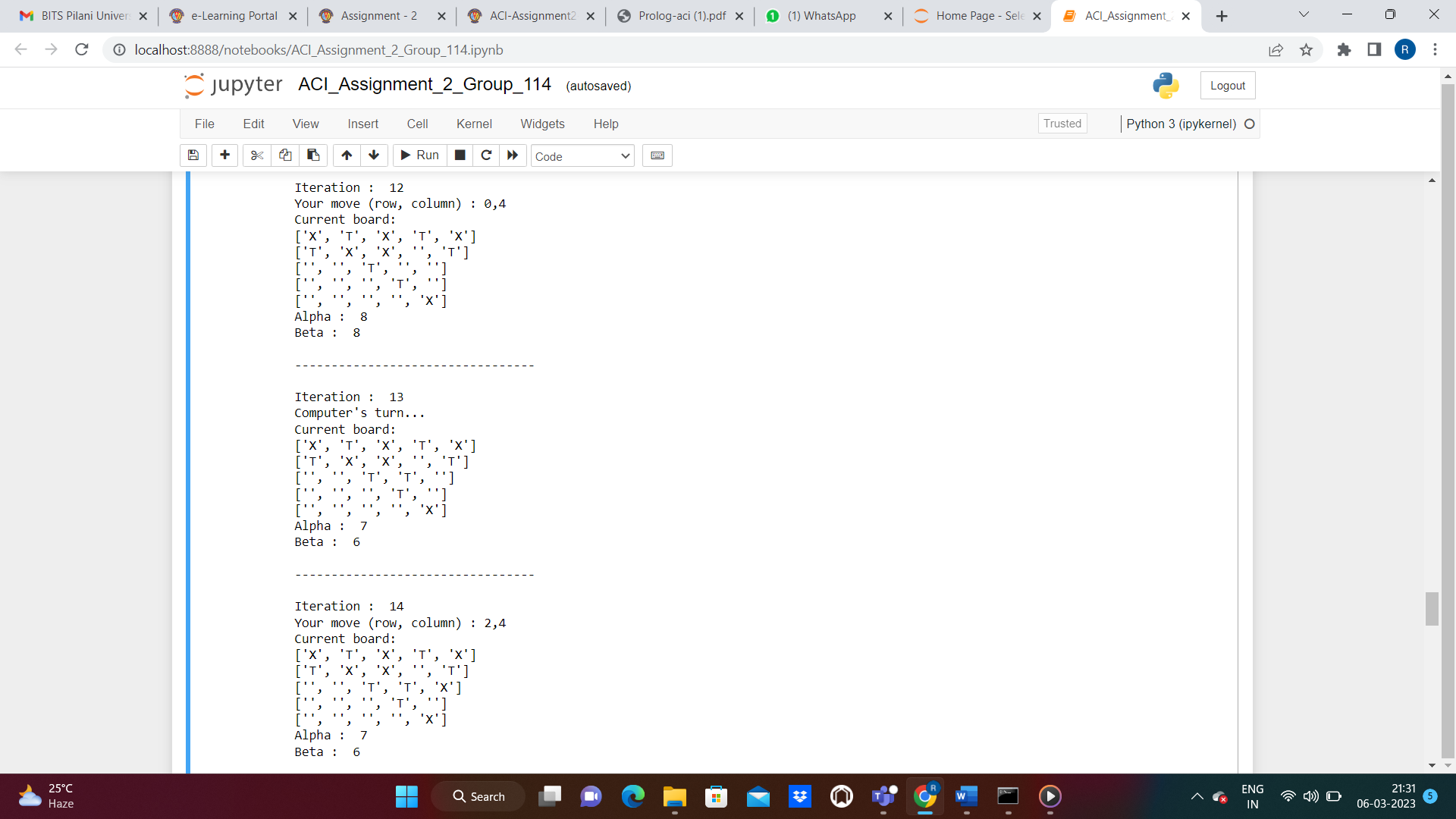
### **Case 3: Tie**

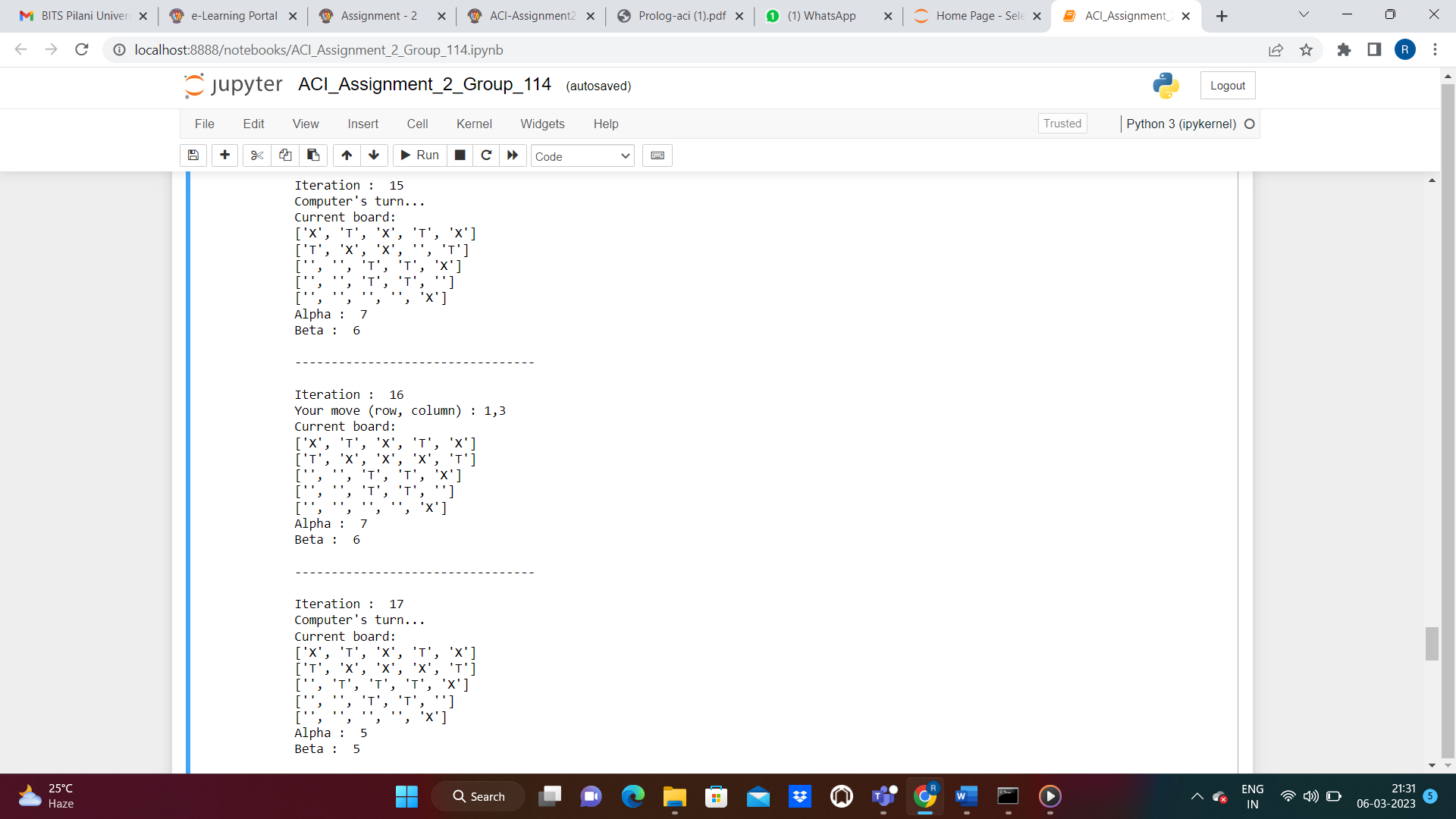


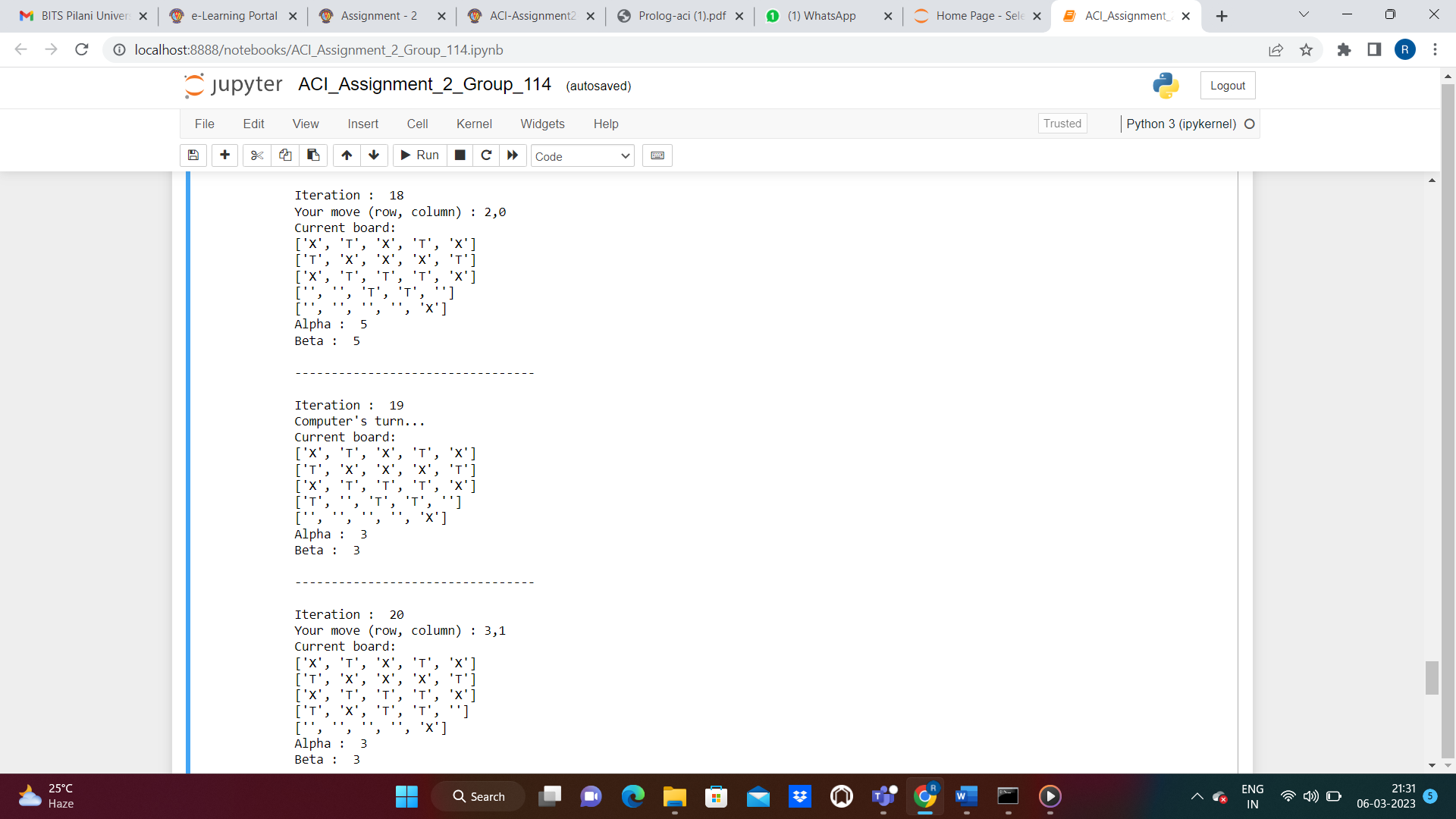


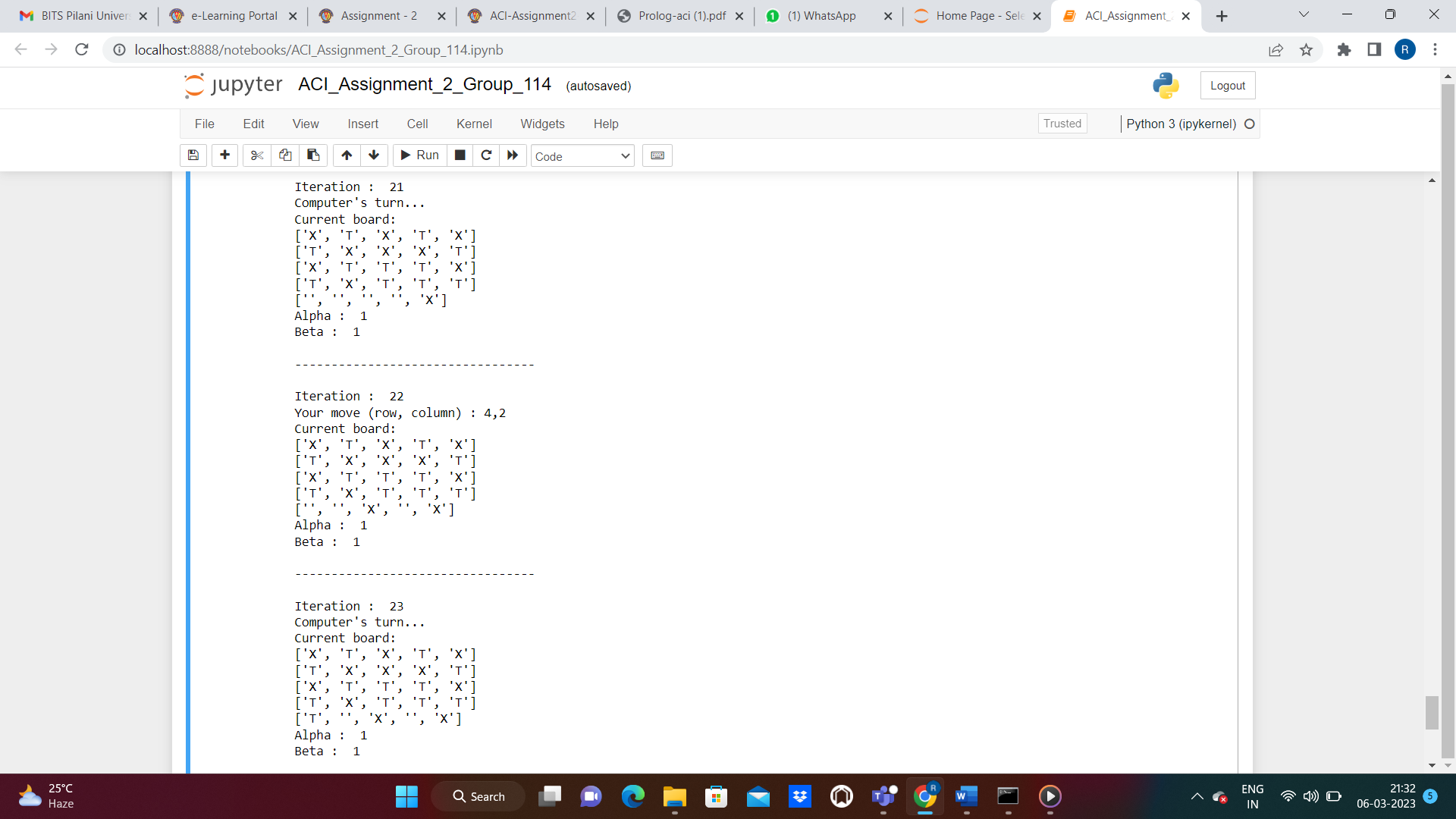


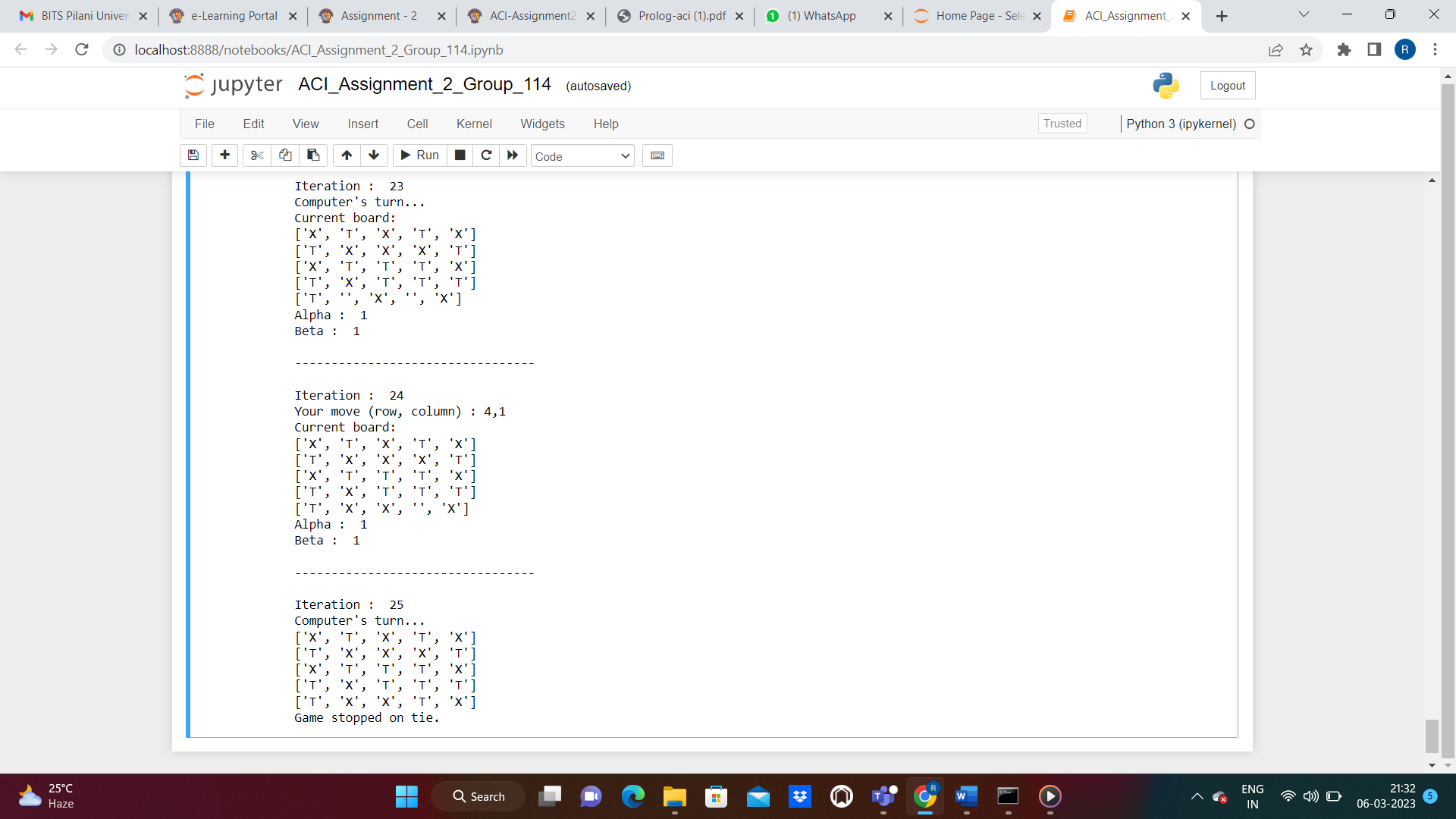








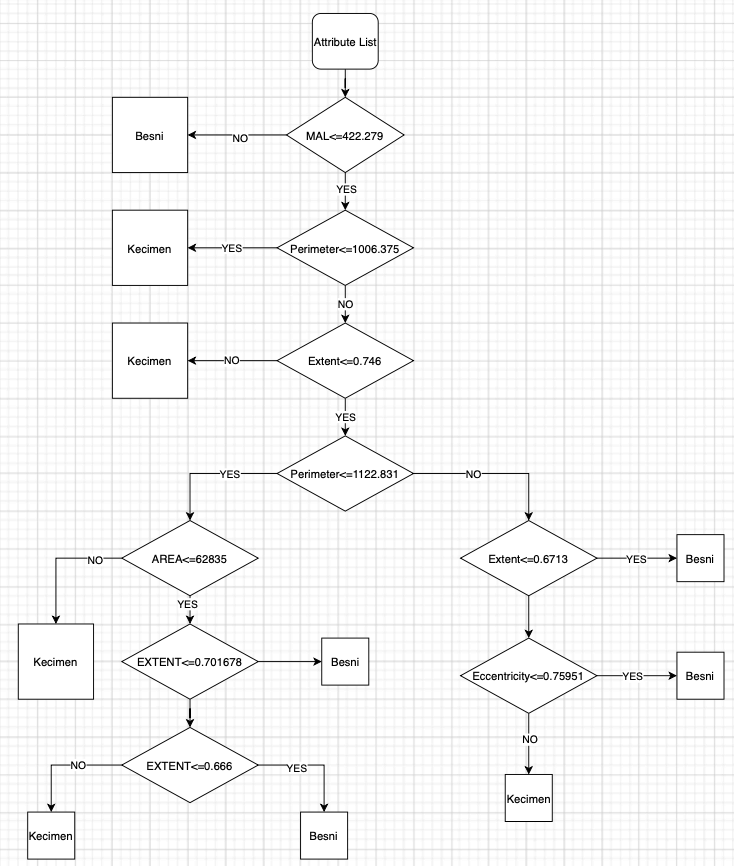




# **Question 2: Logic Problem**

## **Logic Flow**

To understand the logic problem, we have created a flow chart as given under -



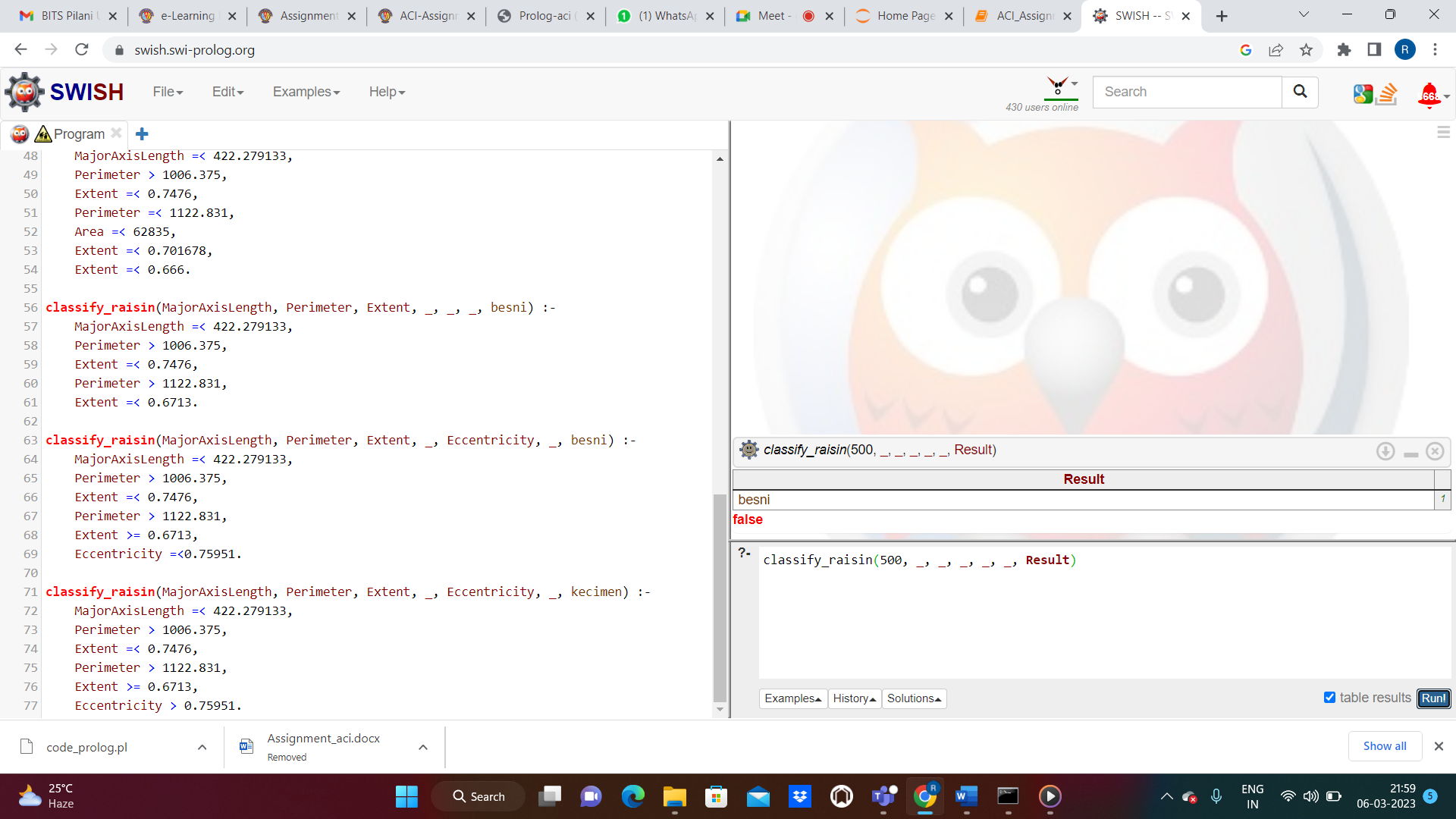
## **If – Else Rules for logic problem**

Here are the if-else rules derived from the given decision tree:

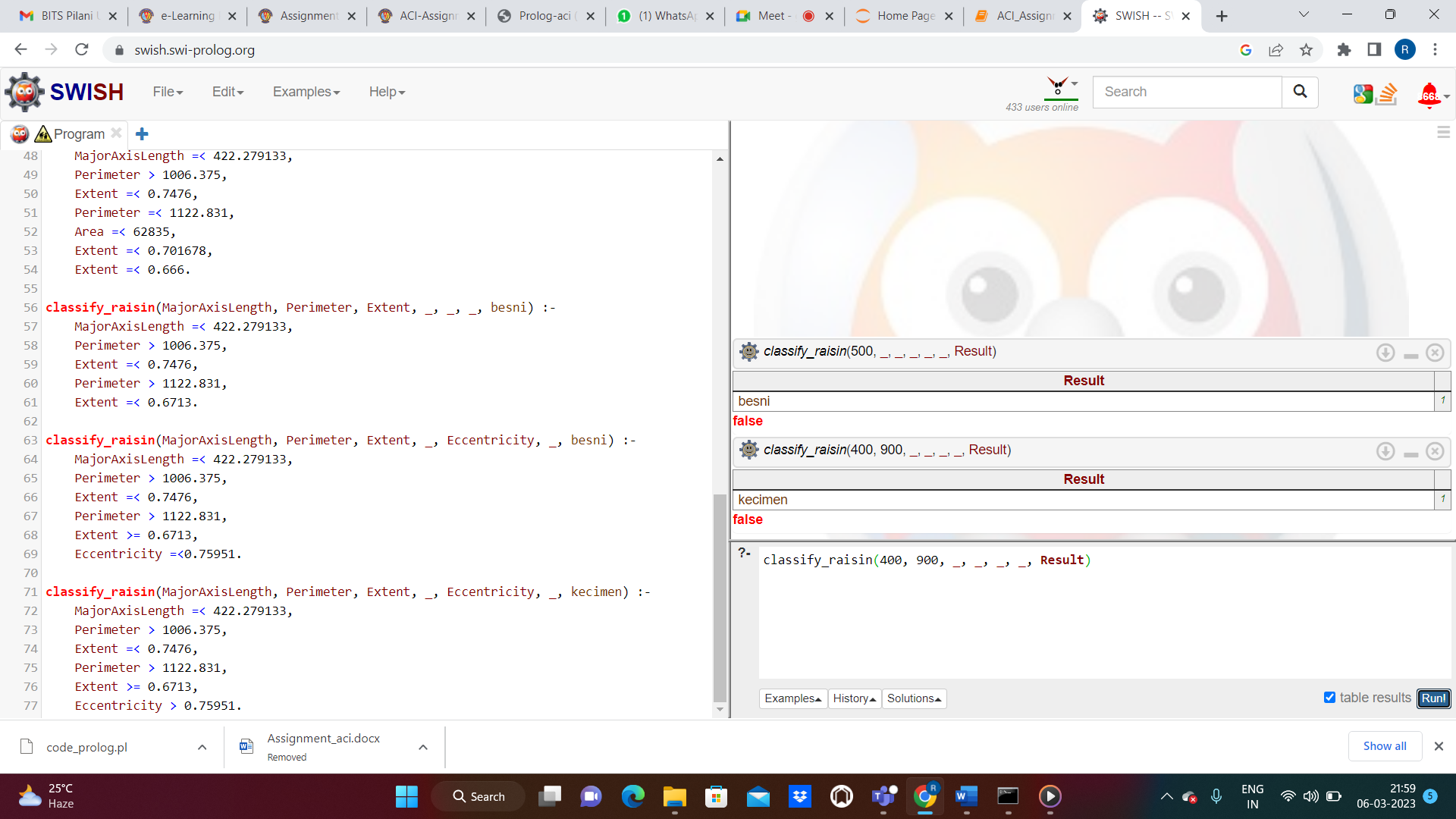
* If MajorAxisLength > 422.279133, then the class is Besni with probability 410.0/41.0
* If MajorAxisLength <= 422.279133 and Perimeter <= 1006.375, then the class is Kecimen with probability 287.0/24.0
* If MajorAxisLength <= 422.279133 and Perimeter > 1006.375 and Extent <= 0.7476:
  + If Perimeter <= 1122.831 and Area <= 62835 and Extent <= 0.701678 and Extent <= 0.666255, then the class is Besni with probability 4.0
  + If Perimeter <= 1122.831 and Area <= 62835 and Extent <= 0.701678 and Extent > 0.666255, then the class is Kecimen with probability 4.0
  + If Perimeter <= 1122.831 and Area > 62835 and Extent > 0.701678, then the class is Besni with probability 6.0
  + If Perimeter > 1122.831 and Extent <= 0.671309, then the class is Besni with probability 7.0
  + If Perimeter > 1122.831 and Extent > 0.671309 and Eccentricity <= 0.75951, then the class is Besni with probability 22.0/7.0
  + If Perimeter > 1122.831 and Extent > 0.671309 and Eccentricity > 0.75951, then the class is Kecimen with probability 9.0/1.0
* If MajorAxisLength <= 422.279133 and Perimeter > 1006.375 and Extent > 0.7476, then the class is Kecimen with probability 27.0

## **Sample iterations and snapshots**

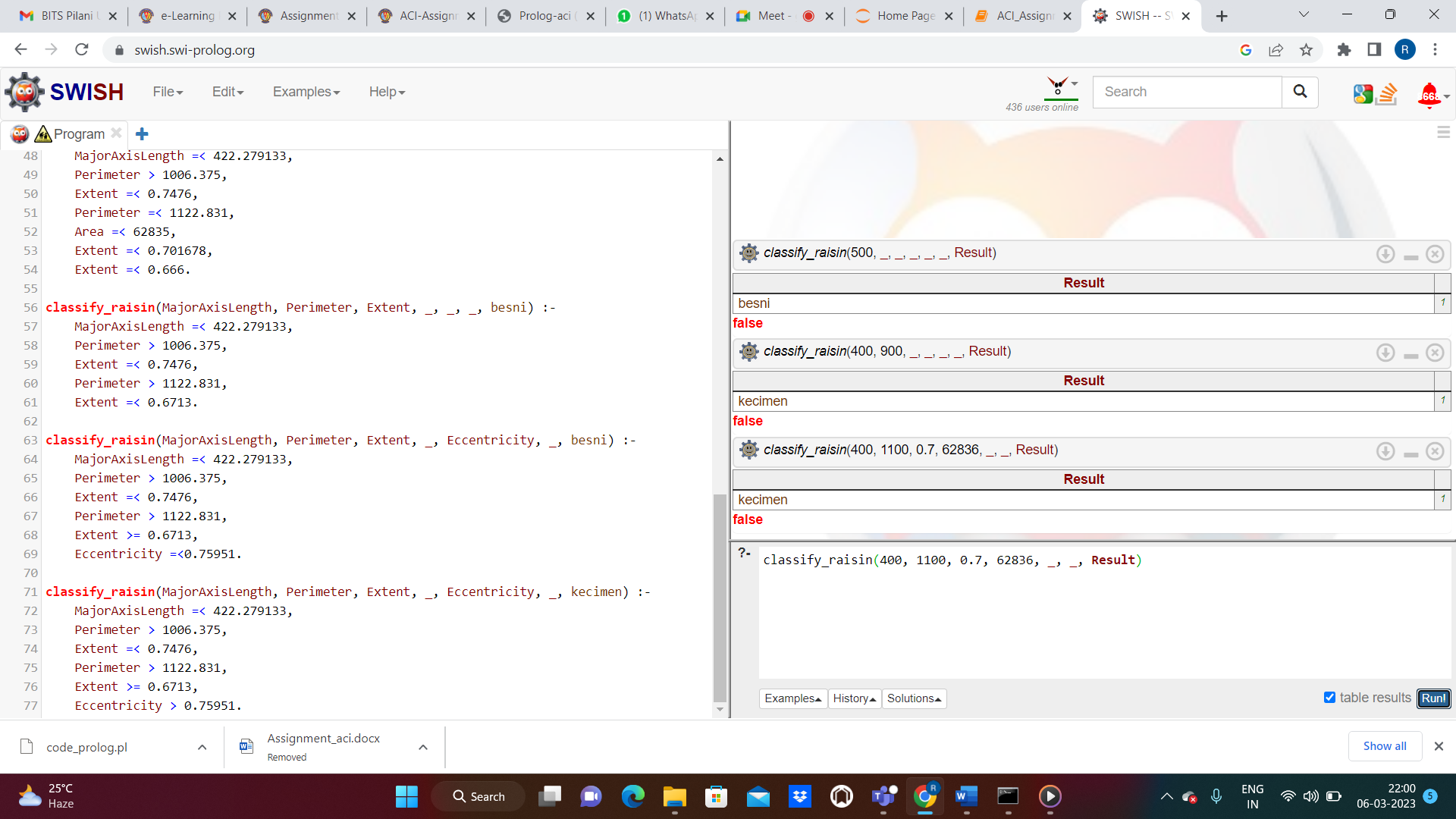
|  |  |  |
| --- | --- | --- |
| **Iteration#** | **Input** | **Expected Output** |
| 1 | classify\_raisin(500, \_, \_, \_, \_, \_, Result) | besni |



|  |  |  |
| --- | --- | --- |
| **Iteration#** | **Input** | **Expected Output** |
| 2 | classify\_raisin(400, 900, \_, \_, \_, \_, Result) | kecimen |



|  |  |  |
| --- | --- | --- |
| **Iteration#** | **Input** | **Expected Output** |
| 3 | classify\_raisin(400, 1100, 0.7, 62836, \_, \_, Result) | kecimen |



|  |  |  |
| --- | --- | --- |
| **Iteration#** | **Input** | **Expected Output** |
| 4 | classify\_raisin(400, 1123, 0.7, 62836, 0.7, \_, Result) | besni |

