**TIC TAC TOE**

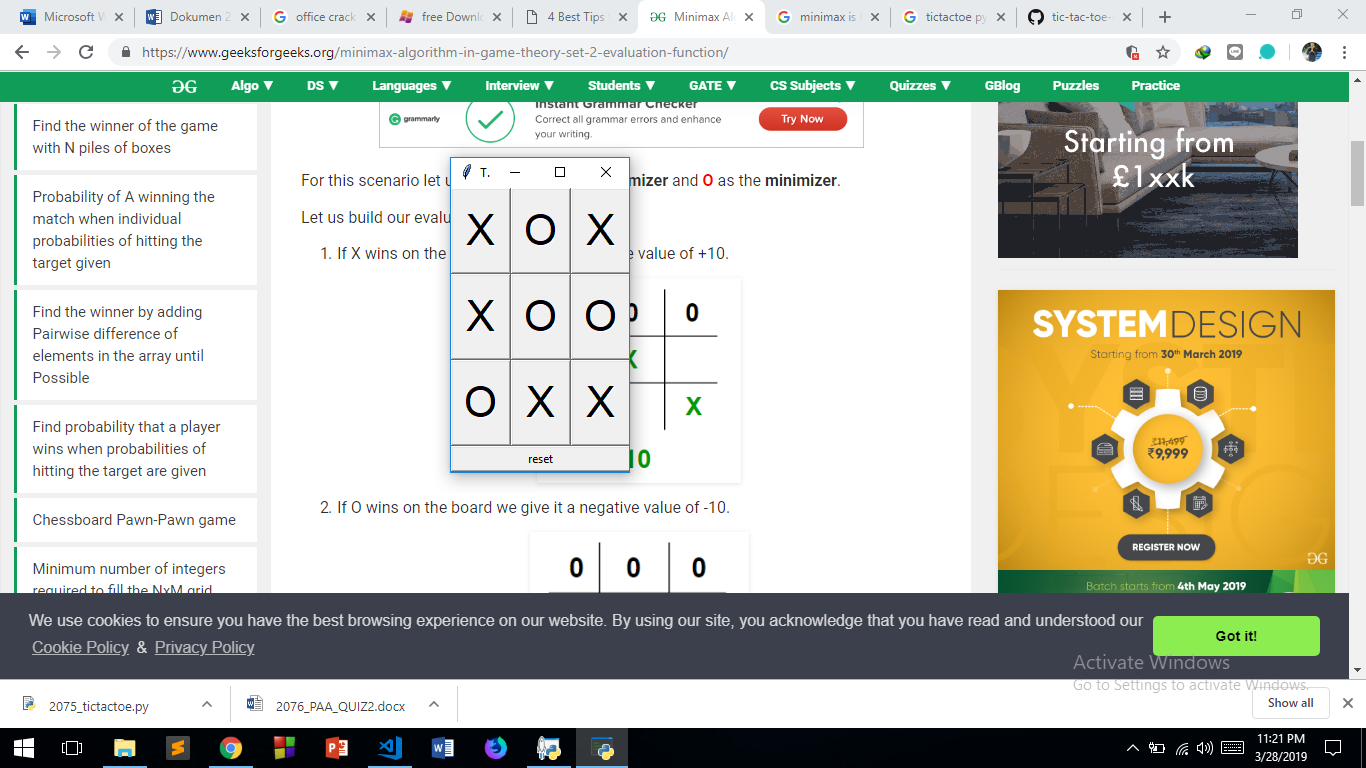
**Implementation Using Minimax Algorithm**

* 1. **Introduction**

Minimax is a kind of [backtracking](https://www.geeksforgeeks.org/tag/backtracking/) algorithm that is used in decision making and game theory to find the optimal move for a player, assuming that your opponent also plays optimally. In Minimax the two players are called maximizer and minimizer. The **maximizer** tries to get the highest score possible while the **minimizer** tries to do the opposite and get the lowest score possible. Basicly, minimax using BFS alghorithm on its implementation

Every board state has a value associated with it. In a given state if the maximizer has upper hand then, the score of the board will tend to be some positive value. If the minimizer has the upper hand in that board state then it will tend to be some negative value. The values of the board are calculated by some heuristics which are unique for every type of game.

* 1. **Rules of the Game**
* The game is to be played between two people (in this program between HUMAN and COMPUTER).
* One of the player chooses ‘O’ and the other ‘X’ to mark their respective cells.
* The game starts with one of the players and the game ends when one of the players has one whole row/ column/ diagonal filled with his/her respective character (‘O’ or ‘X’).
* If no one wins, then the game is said to be draw.



* 1. **Analysis**

If we represent our board as a 3×3 , then we have to check each row, each column and the diagonals to check if either of the players have gotten 3 in a row. The basic idea behind the evaluation function is to give a high value for a board if **maximizer**‘s turn or a low value for the board if **minimizer**‘s turn.

* + If X wins on the board we give it a positive value of -1.
  + If O wins on the board we give it a negative value of +1.
  + If no one has won or the game results in a draw then we give a value of +0.

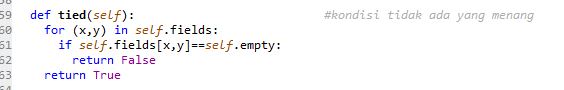
#### Finding the Best Move :

To check whether or not the current move is better than the best move we take the help of **minimax()** function which will consider all the possible ways the game can go and returns the best value for that move, assuming the opponent also plays optimally.



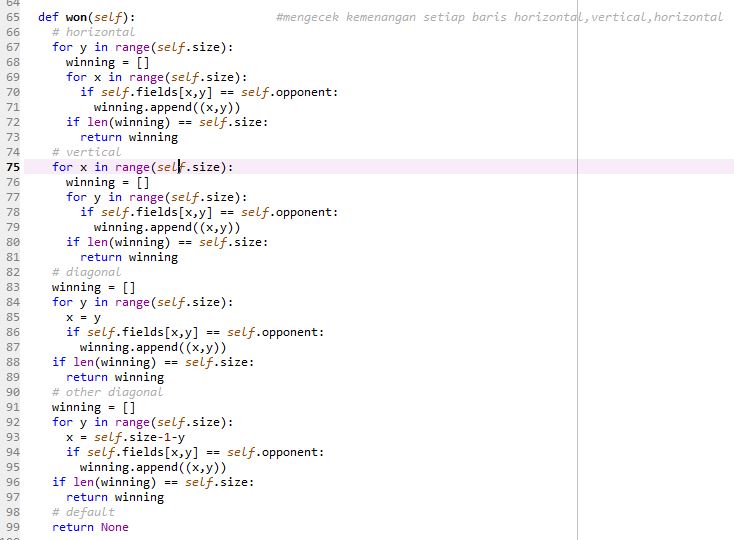
#### Checking for GameOver state :

To check whether the game is over and to make sure there are no moves left we use **tied()** function. It is a simple straightforward function which checks whether a move is available or not and returns true or false respectively, it also indicate that the game end in tied state.



*Checking for Winning state :*

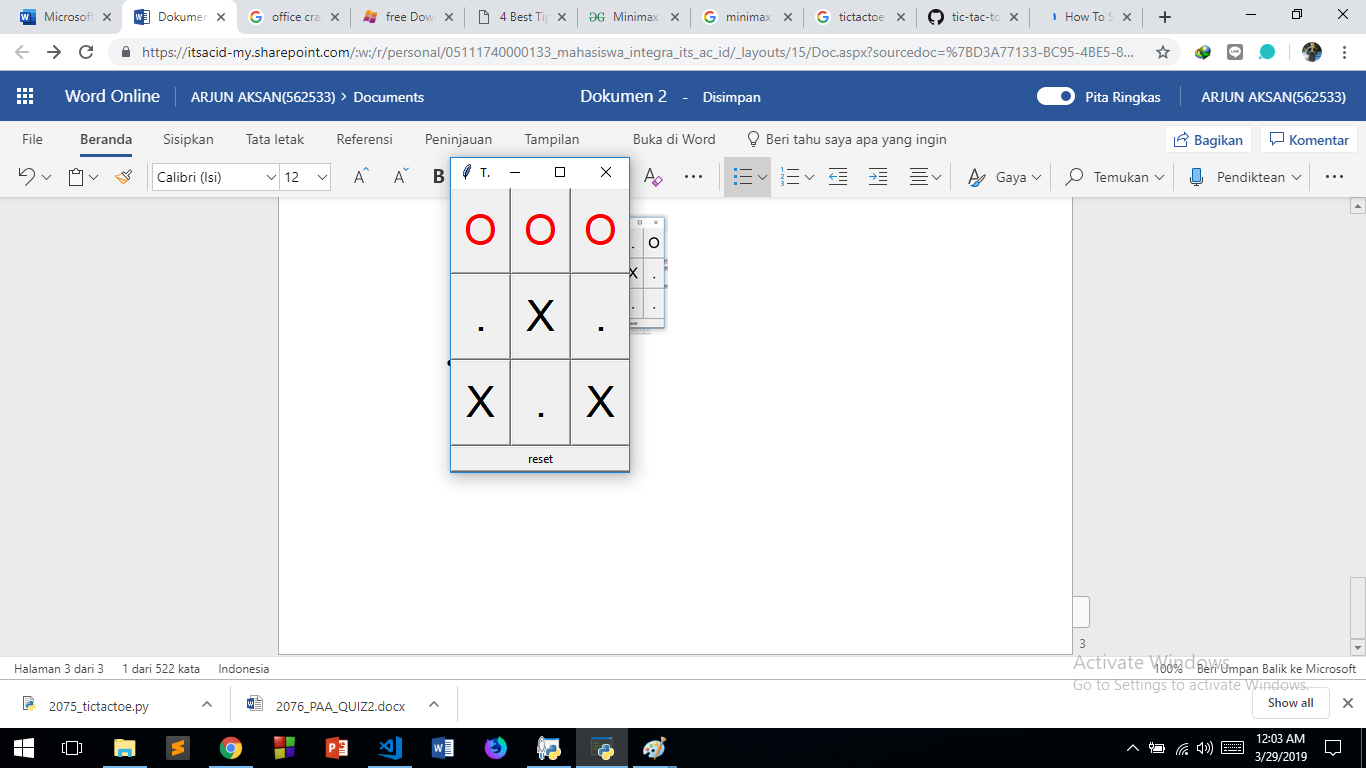
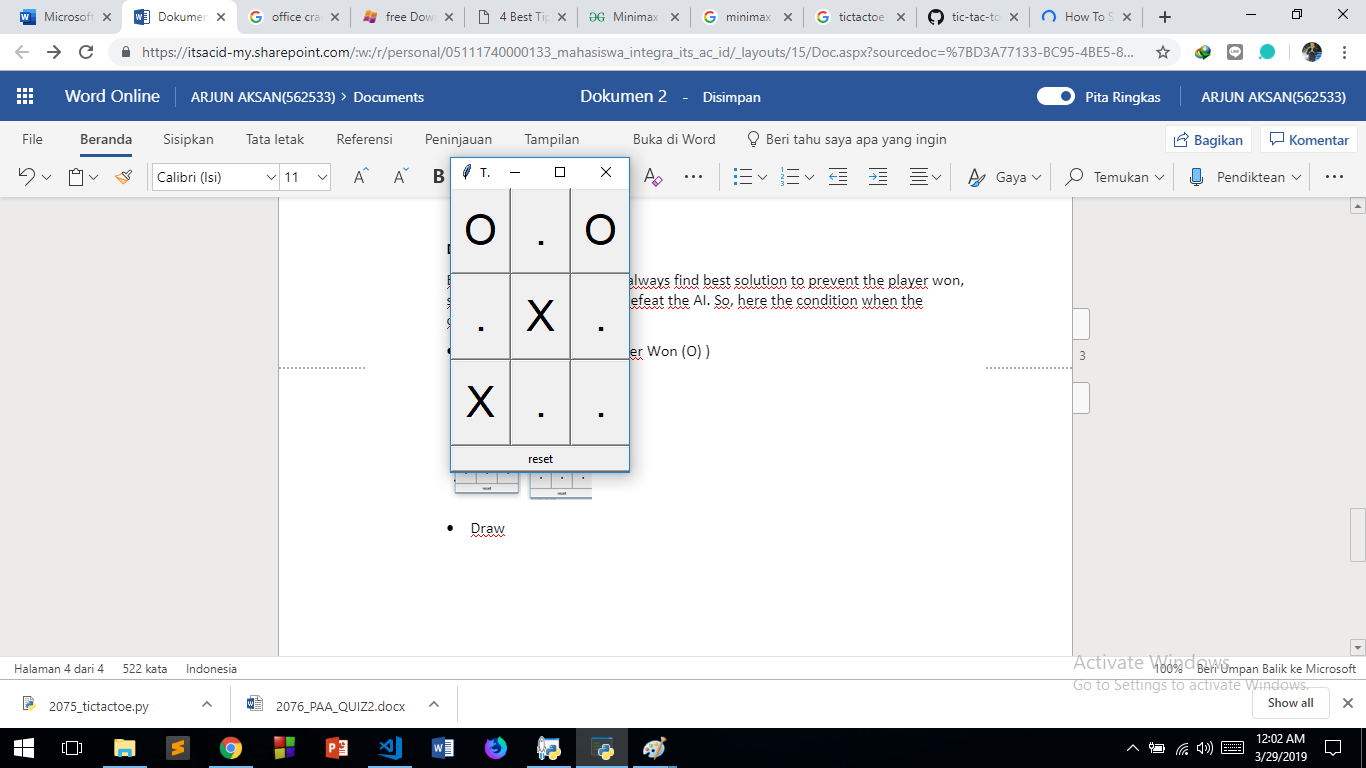
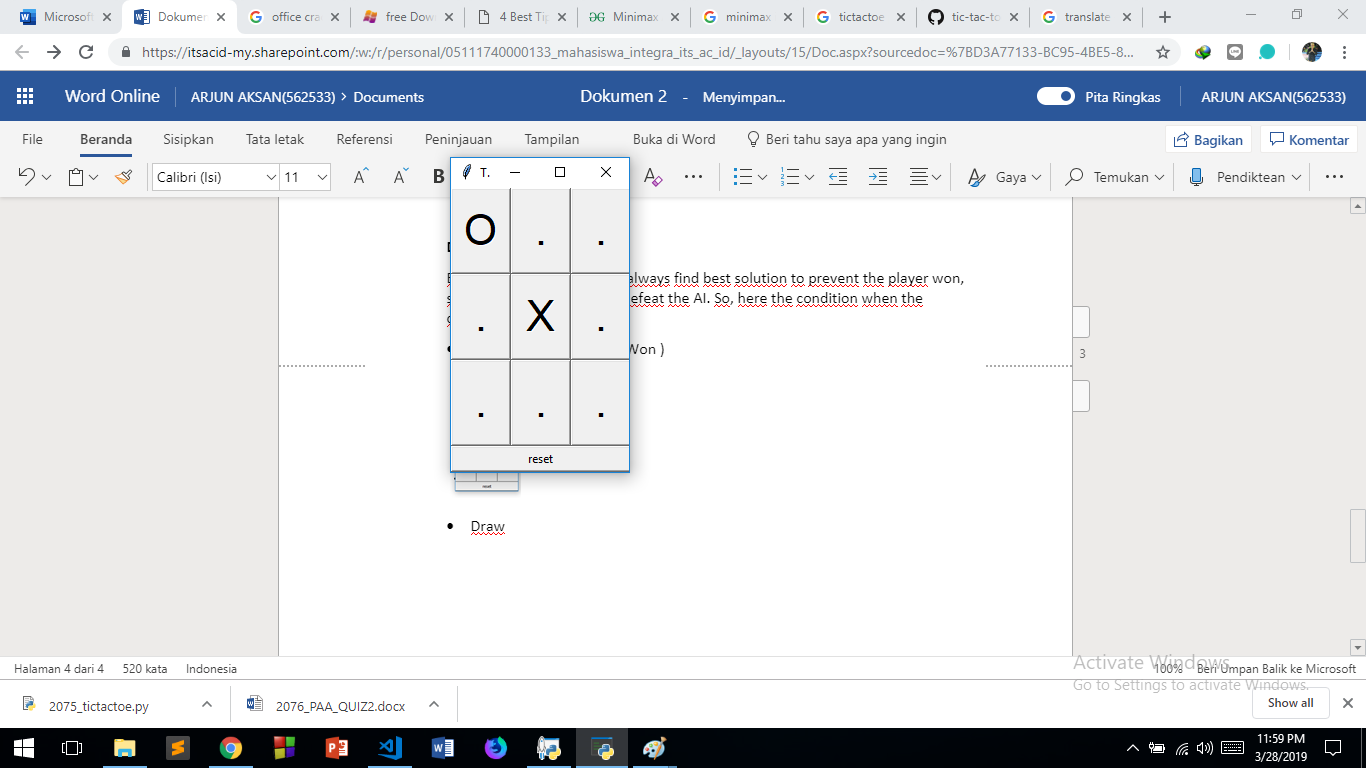
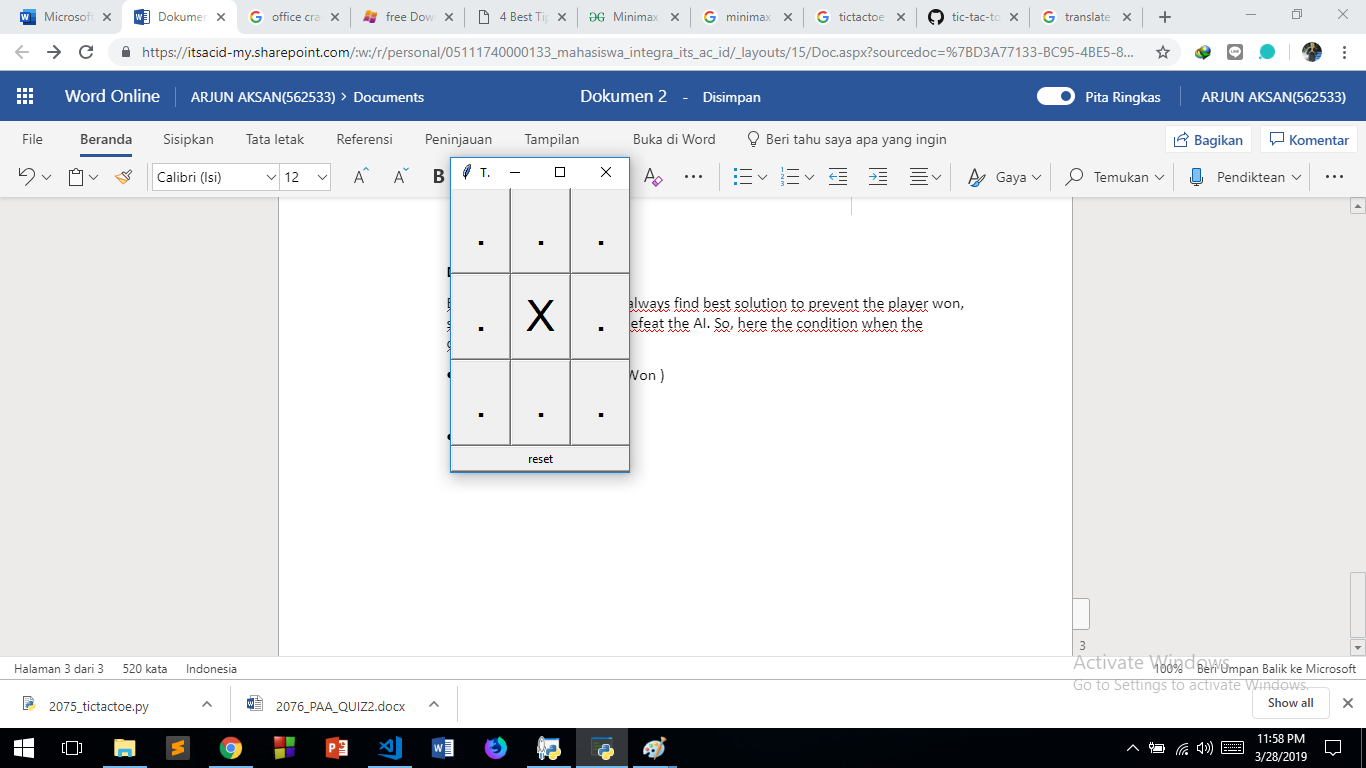
The otherways, the game should stop when one of the player reach winning condition (horizontally, vertically, or diagonally). So we need **won()** function to check it.



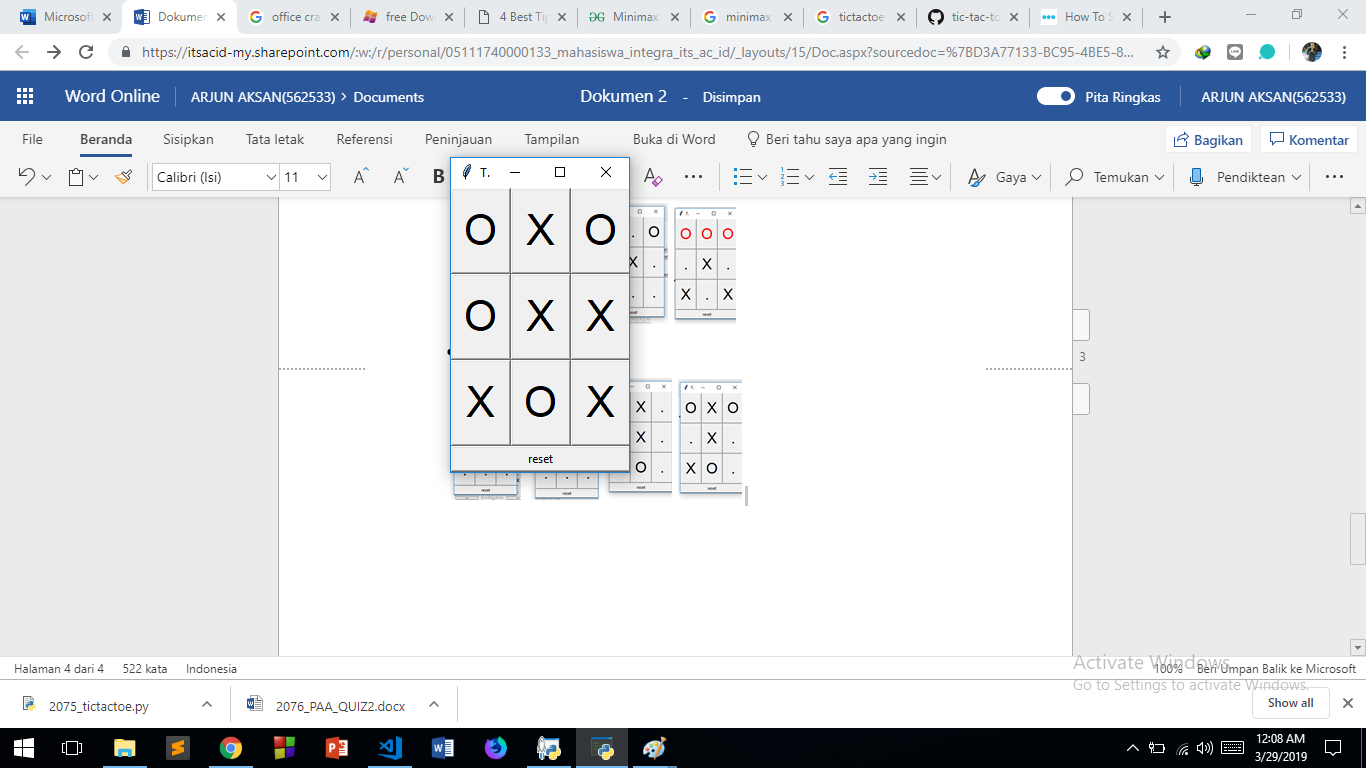
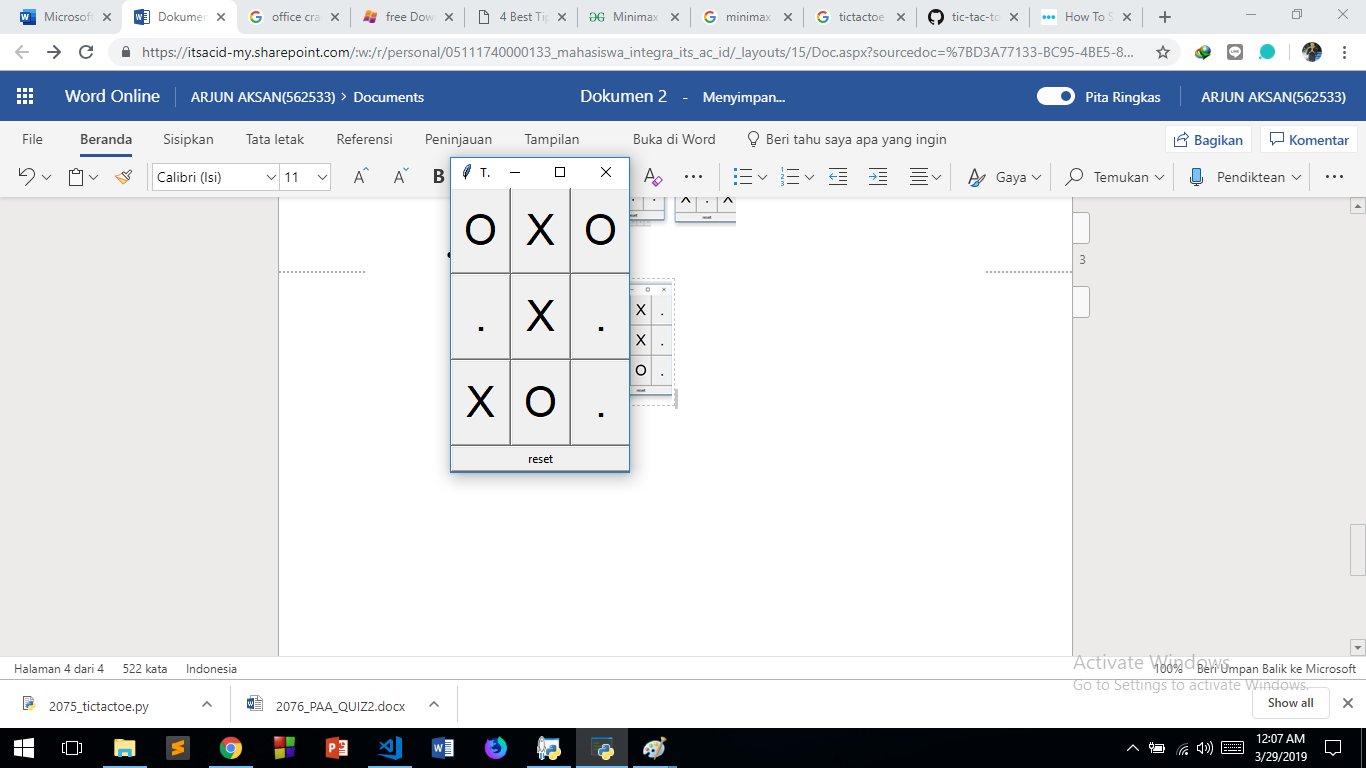
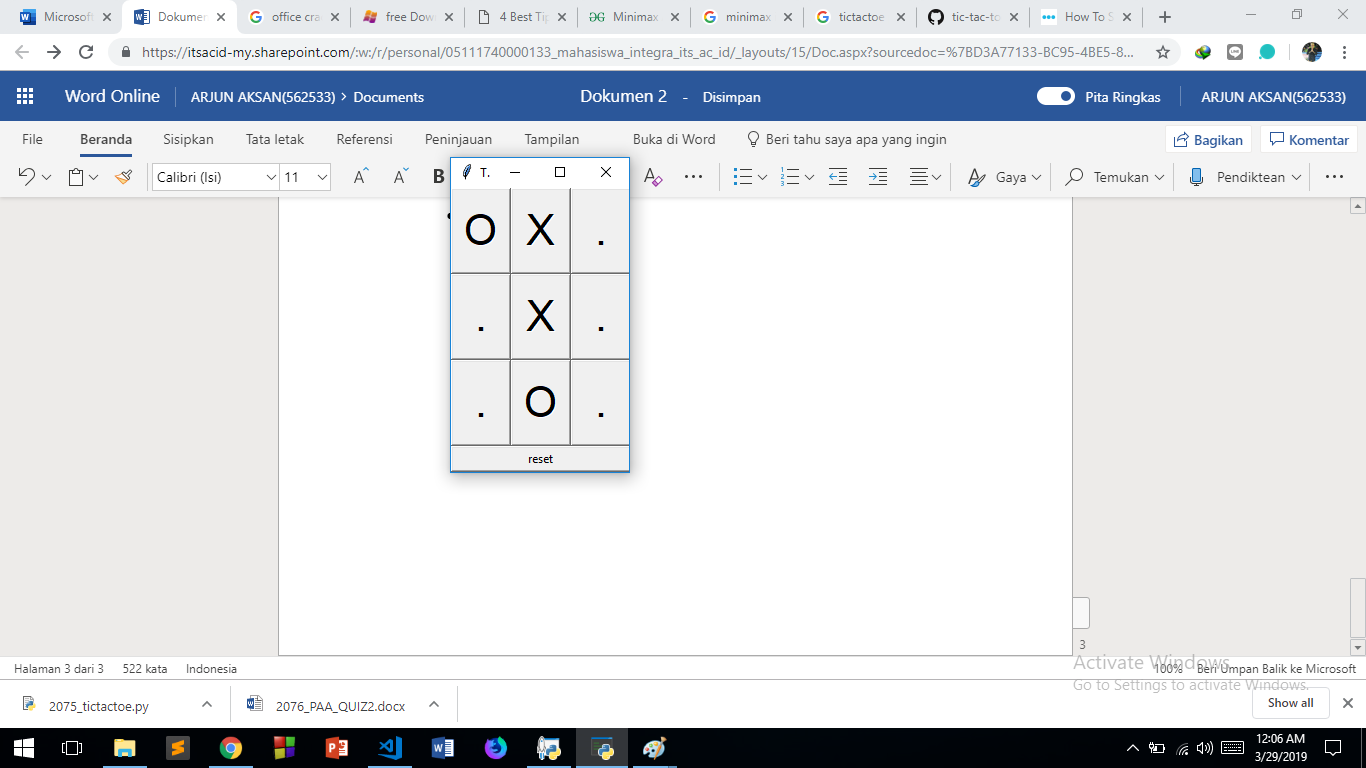
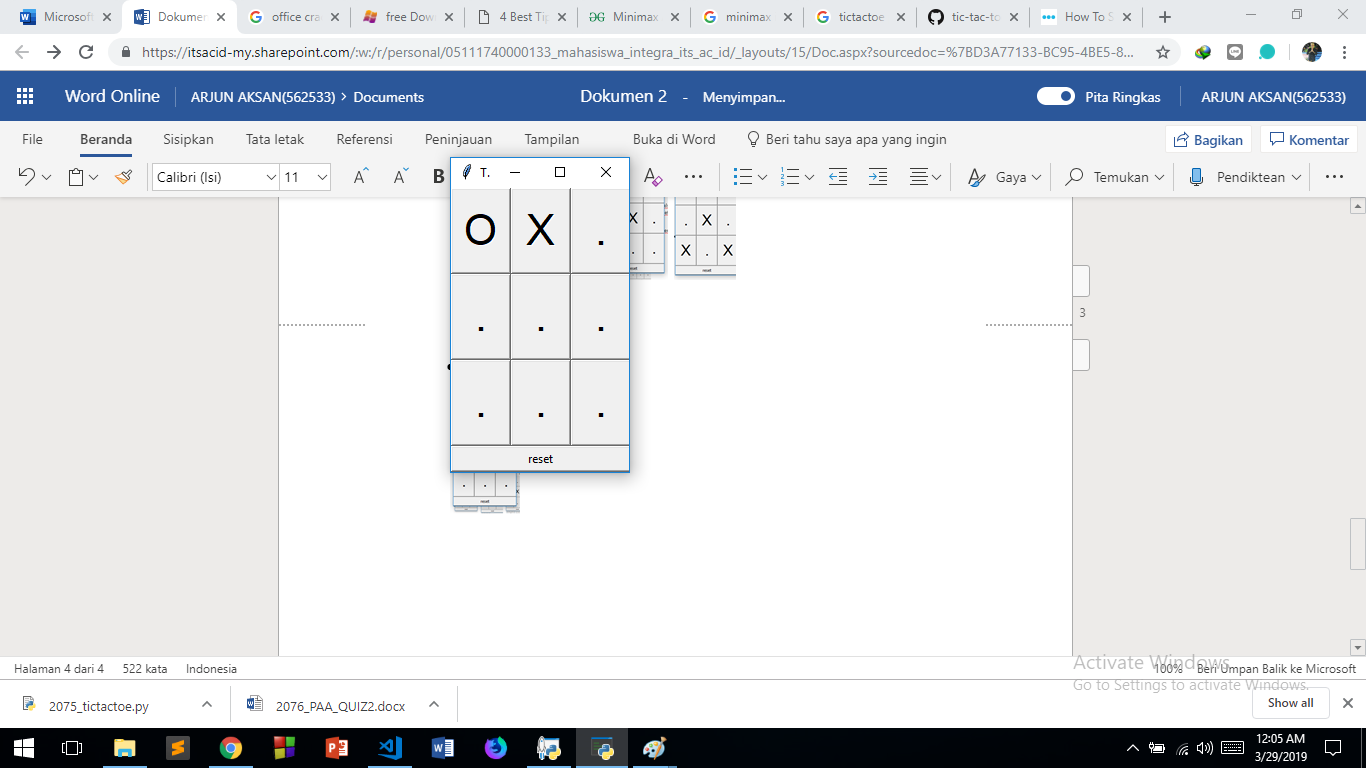
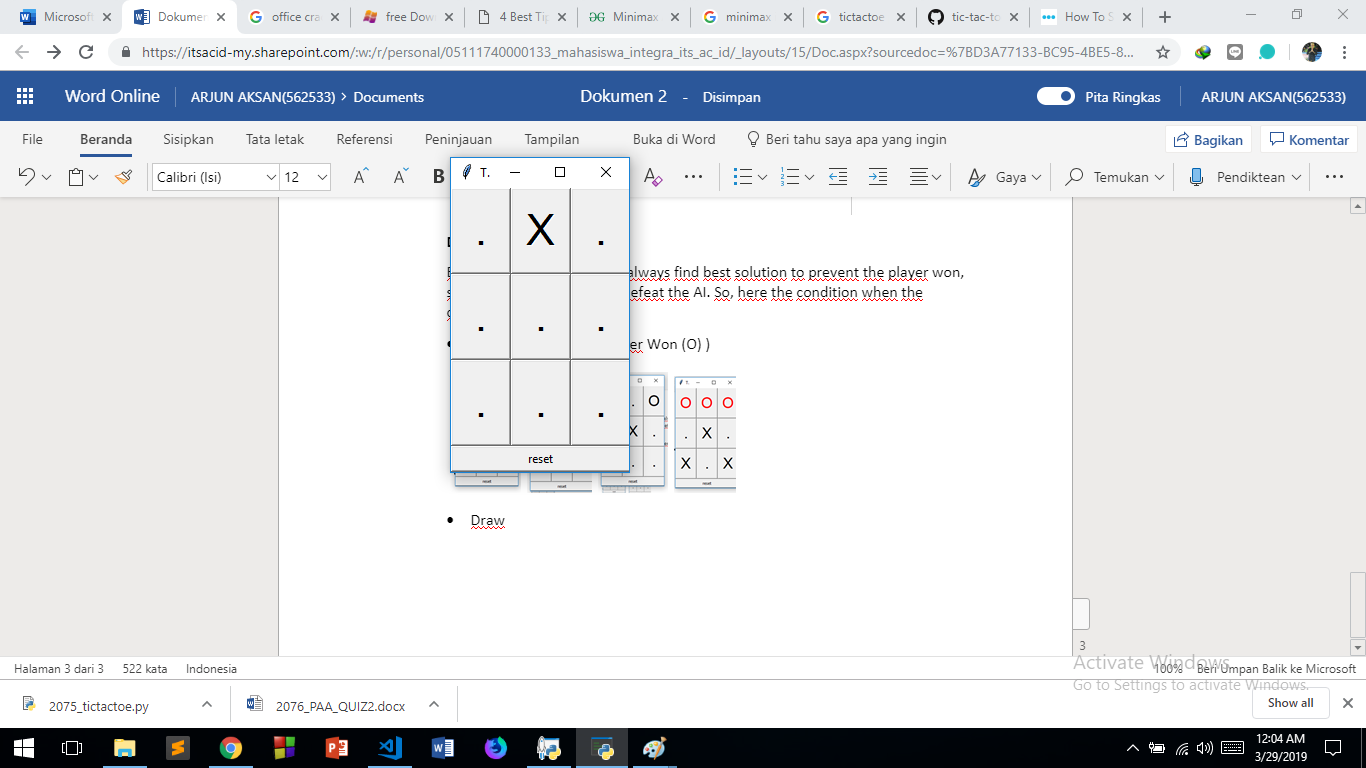
* 1. **Demo**

Because the computer side always find best solution to prevent the player won, so its almost impossible to defeat the AI. So, here the condition when the computer won or draw.

* + Player Lose (X) ( Computer Won (O) )



* + Draw



* 1. **Source Code**

import sys

if sys.version\_info >= (3, 0): *#untuk interface (GUI)*

from tkinter import Tk, Button

from tkinter.font import Font

else:

from Tkinter import Tk, Button

from tkFont import Font

from copy import deepcopy

class Board:

def \_\_init\_\_(self,other=None): *#menginisialisasi komponen dari board*

self.player = 'X'

self.opponent = 'O'

self.empty = '.'

self.size = 3

self.fields = {}

for y in range(self.size):

for x in range(self.size):

self.fields[x,y] = self.empty

*# copy constructor*

if other:

self.\_\_dict\_\_ = deepcopy(other.\_\_dict\_\_) *#mengcopy self dan semua isinya ke other.\_dict\_*

def move(self,x,y):

board = Board(self) *#mereturn status terbaru dari setiap state*

board.fields[x,y] = board.player

(board.player,board.opponent) = (board.opponent,board.player)

return board

def \_\_minimax(self, player):

if self.won():

if player:

return (-1,None)

else:

return (+1,None)

elif self.tied():

return (0,None)

elif player:

best = (-2,None)

for x,y in self.fields:

if self.fields[x,y]==self.empty:

value = self.move(x,y).\_\_minimax(not player)[0] *#implementasi BFS player*

if value>best[0]:

best = (value,(x,y))

return best

else:

best = (+2,None)

for x,y in self.fields:

if self.fields[x,y]==self.empty: *#implementasi BFS opponent*

value = self.move(x,y).\_\_minimax(not player)[0]

if value<best[0]:

best = (value,(x,y))

return best

def best(self): *#melooping minimax*

return self.\_\_minimax(True)[1]

def tied(self): *#kondisi tidak ada yang menang*

for (x,y) in self.fields:

if self.fields[x,y]==self.empty:

return False

return True

def won(self): *#mengecek kemenangan setiap baris horizontal,vertical,horizontal*

*# horizontal*

for y in range(self.size):

winning = []

for x in range(self.size):

if self.fields[x,y] == self.opponent:

winning.append((x,y))

if len(winning) == self.size:

return winning

*# vertical*

for x in range(self.size):

winning = []

for y in range(self.size):

if self.fields[x,y] == self.opponent:

winning.append((x,y))

if len(winning) == self.size:

return winning

*# diagonal*

winning = []

for y in range(self.size):

x = y

if self.fields[x,y] == self.opponent:

winning.append((x,y))

if len(winning) == self.size:

return winning

*# other diagonal*

winning = []

for y in range(self.size):

x = self.size-1-y

if self.fields[x,y] == self.opponent:

winning.append((x,y))

if len(winning) == self.size:

return winning

*# default*

return None

class GUI:

def \_\_init\_\_(self):

self.app = Tk()

self.app.title('TicTacToe')

self.app.resizable(width=True, height=True)

self.board = Board()

self.font = Font(family="Helvetica", size=32)

self.buttons = {}

for x,y in self.board.fields:

handler = lambda x=x,y=y: self.move(x,y)

button = Button(self.app, command=handler, font=self.font, width=2, height=1)

button.grid(row=y, column=x)

self.buttons[x,y] = button

handler = lambda: self.reset()

button = Button(self.app, text='reset', command=handler)

button.grid(row=self.board.size+1, column=0, columnspan=self.board.size, sticky="WE")

self.update()

def reset(self):

self.board = Board()

self.update()

def move(self,x,y):

self.app.config(cursor="watch")

self.app.update()

self.board = self.board.move(x,y)

self.update()

move = self.board.best()

if move:

self.board = self.board.move(\*move)

self.update()

self.app.config(cursor="")

def update(self):

for (x,y) in self.board.fields:

text = self.board.fields[x,y]

self.buttons[x,y]['text'] = text

self.buttons[x,y]['disabledforeground'] = 'black'

if text==self.board.empty:

self.buttons[x,y]['state'] = 'normal'

else:

self.buttons[x,y]['state'] = 'disabled'

winning = self.board.won()

if winning:

for x,y in winning:

self.buttons[x,y]['disabledforeground'] = 'red'

for x,y in self.buttons:

self.buttons[x,y]['state'] = 'disabled'

for (x,y) in self.board.fields:

self.buttons[x,y].update()

def mainloop(self):

self.app.mainloop()

if \_\_name\_\_ == '\_\_main\_\_':

GUI().mainloop()