Tic-tac-toe

Github:

https://github.com/pjavier98/AI/tree/master/tic-tac-toe

tic-tac-toe.py

```
from board import *
from util import *
from random import randint

def game(board, isHuman):
  while True:
    score = board.evaluate_function()
    if score == 10 and isHuman:
        print('Human is the winner')
        break
    elif score == -10 and not isHuman:
        print('AI is the winner')
        break

    isHuman = not isHuman
    if board.isMovesLeft():
      if isHuman:
        print('Human turns')
        print('Enter the row and column: ', end='')
        row, column = input().split()
        row = int(row)
        column = int(column)
        # return
        board.update_field(row, column, True)

      else:
        print('AI turns')
        # Find and update the best row and column based on the board
        board.findBestMove()
        print('row: ', board.bestMoveRow)
        print('column: ', board.bestMoveColumn)
        board.update_field(-1, -1, False)
      print('\n', str(board))
    else:
      print('There was a draw')
      break

def main():
  print('####################################')
  print('# Welcome to the tic-tac-toe game #')
  print('####################################\n')
```

```
  row = randint(0, 2)
  column = randint(0, 2)

  board = Board()

  board.update_initial_state(row, column)

  # False -> IA start
  print(str(board))
  game(board, False)

main()
```

board.py

```
class Board:
        def __init__(self):
                self.bestMoveRow = -1
                self.bestMoveColumn = -1
                self.player = 'x'
                self.opponent = 'o'
                self.depth = 0
                self.board = [
                        [ '_', '_', '_' ],
                        [ '_', '_', '_' ],
                        [ '_', '_', '_' ]
                ]

        def __str__(self):
                table = self.board
                return ('    0   1   2\n  #############\n0 #  {} | {} |
{}\n  # -----------\n1 #  {} | {} | {}\n  # -----------\n2 #  {} | {} |
{}\n\n\n'
                        .format(table[0][0], table[0][1], table[0][2],
                                                        table[1][0],
table[1][1], table[1][2],
                                                        table[2][0],
table[2][1], table[2][2]))

        def update_initial_state(self, row, column):
                self.board[row][column] = self.opponent

        def update_field(self, row, column, isHuman):
                if isHuman:
                        self.board[row][column] = self.player
                else:
                        self.board[self.bestMoveRow][self.bestMoveColumn] =
self.opponent

        def isMovesLeft(self):
                for i in range(3):
```

```
                        for j in range(3):
                                if (self.board[i][j] == '_'):
                                        return True
                return False

        def evaluate_function(self):
                table = self.board
                # Checking for Rows for X or O victory.
                for row in range(3):
                        if table[row][0]==table[row][1] and table[row][1]
== table[row][2]:
                                if table[row][0] == self.player:
                                        return 10
                                elif table[row][0] == self.opponent:
                                        return -10

                # Checking for Columns for X or O victory.
                for column in range(3):
                                if table[0][column] == table[1][column] and
table[1][column] == table[2][column]:
                                        if table[0][column] == self.player:
                                                return 10
                                        elif table[0][column] ==
self.opponent:
                                                return -10


                # Checking for Diagonals for X or O victory.
                if table[0][0] == table[1][1] and table[1][1] == table[2]
[2]:
                        if table[0][0] == self.player:
                                return 10
                        elif table[0][0] == self.opponent:
                                return -10

                if table[0][2] == table[1][1] and table[1][1] == table[2]
[0]:
                        if table[0][2] == self.player:
                                return 10
                        elif table[0][2] == self.opponent:
                                return -10

                # Else if none of them have won then return 0
                return 0

        def minimax(self, depth, isMax):
                score = self.evaluate_function();

                # If Maximizer has won the game return his/her
                if score == 10:
                        return score

                # If Minimizer has won the game return his/her
                if score == -10:
```

```
                        return score

                # If there are no more moves and no winner then
                if not self.isMovesLeft():
                        return 0

                # If this maximizer's move
                if isMax:
                        best = -1000

                        # Traverse all cells
                        for i in range(0, 3):
                                for j in range(0, 3):
                                        # Check if cell is empty
                                        if (self.board[i][j] == '_'):
                                                # Make the move
                                                self.board[i][j] =
self.player

                                                # Call minimax recursively
and choose the maximum value
                                                best = max(best,
self.minimax(depth + 1, not isMax))

                                                # Undo the move
                                                self.board[i][j] = '_'
                        return best

                # If this minimizer's move
                else:
                        best = 1000

                        # Traverse all cells
                        for i in range(0, 3):
                                for j in range(0, 3):
                                        # Check if cell is empty
                                        if (self.board[i][j] == '_'):
                                                # Make the move
                                                self.board[i][j] =
self.opponent

                                                # Call minimax recursively
and choose the minimum value
                                                best = min(best,
self.minimax(depth + 1, not isMax))

                                                # Undo the move
                                                self.board[i][j] = '_';
                        return best

        def findBestMove(self):
                bestVal = -1000

                # Traverse all cells, evaluate minimax function for
```

```
                # all empty cells. And return the cell with optimal value.
                for i in range(0, 3):
                        for j in range(0, 3):
                                # Check if cell is empty
                                if self.board[i][j] == '_':
                                        # Make the move
                                        self.board[i][j] = self.opponent

                                        # Compute evaluation function for
this move
                                        moveVal = self.minimax(0, True)

                                        # Undo the move
                                        self.board[i][j] = '_'

                                        # If the value of the current move
is
                                        # more than the best value, then
update best
                                        if moveVal > bestVal:
                                                # print('testando')
                                                # for i in range(3):
                                                #
print(self.board[i])
                                                # print('i: ', i)
                                                # print('j: ', j)
                                                self.bestMoveRow = i
                                                self.bestMoveColumn = j
                                                bestVal = moveVal
```

util.py

```
def read_input(lower, upper):
  while True:
    try:
      number = int(input())
      if (number >= lower and number <= upper):
        return number
    except:
      pass
    print("Invalid input, please choose again from [" + str(lower) + "-" +
str(upper) + "]")
```