```python
#!/usr/bin/python3

import numpy as np
import argparse
from abc import ABC, abstractmethod

# coordinate class to improve readability
class coordinate:
 def __init__(self, x=None,y=None):
  if x is None:
   self.x=-1
  else:
   self.x=x
  if y is None:
   self.y=-1
  else:
   self.y=y
 def __str__(self):
  return f'(x:{self.x}, y:{self.y})'
 def __repr__(self):
  return self.__str__()


# recursive print method to speed up debugging by indenting and showing how deep we have gone.
def recPrint(depth, *args):
 numSpaces=4
 print(" " * numSpaces * depth,depth + 1,". ", end="")
 for arg in args:
  print(arg, end="")
 print()


# abstract base class defining functionality of game required to be able to run minimax on the
game.
class MinimaxGame(ABC):
 @abstractmethod
 def make_move(self, player, move):
  pass
 @abstractmethod
 def revert_last_move(self):
  pass
 @abstractmethod
 def get_legal_moves(self):
  pass
 @abstractmethod
 def is_terminal(self):
  pass
 @abstractmethod
 def utility(self):
  pass


# minimax algorithm that determines the utility of a move based on players choosing the optimal
move
# optimized to give the move with the lowest depth (win in the fewest moves)
# optimized to remove unneccessary branches using alpha beta pruning
def minimax(player, game: MinimaxGame, depth, alpha, beta):
 # base case - checks if the game is over
 if game.is_terminal():
  return game.utility(), depth

 scores = []
 depths = []
 legal_actions = game.get_legal_moves()

 for action in legal_actions:
  game.make_move(player, action)
```

```
  minimax_score, minimax_depth = minimax(player * -1, game, depth + 1, alpha, beta)
  scores.append(minimax_score)
  depths.append(minimax_depth)

  game.revert_last_move()

  # alpha beta pruning
  if player == 1:
   best_score = max(scores)
   alpha = max(alpha, best_score)
  else:
   best_score = min(scores)
   beta = min(beta, best_score)

  if beta <= alpha:
   break

 best_depth = 30

 for ind_score, score in enumerate(scores):
  if score == best_score and depths[ind_score] < best_depth:
   best_depth = depths[ind_score]

 return best_score, best_depth

# tictactoe game class that defines how tic tac toe will be played
class TicTacToe(MinimaxGame):
 # defines what data the tictactoe game stores
 def __init__(self, board=None, player=1) -> None:
  if board is None:
   self.board = self.init_board()
  else:
   self.board = board.copy()
  self.player = player
  self.history=[]

 # sets up the game board
 def init_board(self):
  return np.array([[0,0,0],[0,0,0],[0,0,0]])

 # prints the game board
 def print_board(self):
  print("\n\n")
  for v, row in enumerate(self.board):
   for i, value in enumerate(row):
    if value == 1:
     print('X', end=' ')
    elif value == -1:
     print('O', end=' ')
    else:
     print(' ', end=' ')
    if i < len(row) - 1:
     print('|', end=' ')
   if v < len(self.board) - 1:
    print("\n---------")
  print("\n\n")

 # gets all legal moves in the game
 def get_legal_moves(self):
  legalMoves=[]
  for ind_y,y in enumerate(self.board):
   for ind_x,x in enumerate(y):
    if x == 0:
     legalMoves.append(coordinate(ind_x,ind_y))
  return legalMoves
```

```
# makes a move for a player in a coordinate
def make_move(self, player, move):
 self.board[move.y][move.x] = player
 self.history.append(move)

# undos the last move
def revert_last_move(self):
 last_move: coordinate = self.history.pop()
 if last_move.x == -1 or last_move.y == -1:
  print("error reverting last move, last move was never filled")
 self.board[last_move.y][last_move.x] = 0

# determines if the board is full
def board_full(self):
 for y in self.board:
  for x in y:
   if x == 0:
    return False
 return True

# checks to see if either player has won and returns 0 otherwise
def eval_win(self):
 # check rows
 for x in range(0,3):
  if self.board[x][0] == self.board[x][1] == self.board[x][2] == 1:
   return 1
  if self.board[x][0] == self.board[x][1] == self.board[x][2] == -1:
   return -1
 # check columns
 for x in range(0,3):
  if self.board[0][x] == self.board[1][x] == self.board[2][x] == 1:
   return 1
  if self.board[0][x] == self.board[1][x] == self.board[2][x] == -1:
   return -1
 #check diagonals
 if self.board[0][0] == self.board[1][1] == self.board[2][2] == 1:
  return 1
 if self.board[0][0] == self.board[1][1] == self.board[2][2] == -1:
  return -1
 if self.board[2][0] == self.board[1][1] == self.board[0][2] == 1:
  return 1
 if self.board[2][0] == self.board[1][1] == self.board[0][2] == -1:
  return -1
 return 0

# callback to the eval win function to satisfy interface
def utility(self):
 return self.eval_win()

# defines if the game is over
def is_terminal(self):
 if self.eval_win() == 1 or self.eval_win() == -1:
  return True
 if self.board_full():
  return True
 return False

# actually plays the game between two cpu players
def play_game(self):
 self.print_board()
 while self.eval_win() == 0 and not self.board_full():
  print("player ", self.player,"'s turn:")
  optimal_move = []
  if self.player == 1:
   best_score = -2
  else:
   best_score = 2
```

```
   best_depth = 30

   # updates the best move that is currently stored
   def update_best_move(score, depth, ind_y, ind_x):
    nonlocal best_score, best_depth
    best_score = score
    best_depth = depth
    optimal_move.clear()
    optimal_move.append(ind_y)
    optimal_move.append(ind_x)

   # evaluates all possible moves and filters for the best one
   for ind_y, y in enumerate(self.board):
    for ind_x, x in enumerate(y):
     if x == 0:
      self.board[ind_y][ind_x] = self.player
      print("finding minmax for ",self.player," on (",ind_x + 1,ind_y + 1,")")
      score, depth = minimax(self.player * -1, self, 0, -2, 2)
      print( "minimax score: ", score, " minimax depth: ", depth,"\n")
      self.board[ind_y][ind_x] = 0


      if self.player == 1:
       if score > best_score or (score == best_score and depth < best_depth):
        update_best_move(score, depth, ind_y, ind_x)
      else:
       if score < best_score or (score == best_score and depth < best_depth):
        update_best_move(score, depth, ind_y, ind_x)

   # play the best move
   self.board[optimal_move[0]][optimal_move[1]]=self.player
   print("player: ", self.player, " plays: ", optimal_move)
   self.player *= -1
   self.print_board()
  return self.board, self.eval_win()

# loads in the game board if one is provided
def load_board( filename ):
 return np.loadtxt( filename)

# main logic which sets up the board and calls the play game function
def main():
 parser = argparse.ArgumentParser(description='Play tic tac toe')
 parser.add_argument('-f', '--file', default=None, type=str ,help='load board from file')
 parser.add_argument('-p', '--player', default=1, type=int, choices=[1,-1] ,help='player that
playes first, 1 or -1')
 args = parser.parse_args()

 board = load_board(args.file) if args.file else None
 testcase =  np.array([[ 0,0,0],
                       [-1,1,0],
                       [-1,0,0]])
 ttt = TicTacToe(testcase, args.player)

 b,p = ttt.play_game()
 print("final board: \n{}".format(b))
 print("winner: player {}".format(p))

if __name__ == '__main__':
 main()
```