```
from copy import deepcopy
INF = float('inf')


class Color:
    WHITE = "w"
    BLACK = "b"


class Piece:
    KING = "K"
    QUEEN = "Q"
    ROOK = "R"
    BISHOP = "B"
    KNIGHT = "N"
    PAWN = "P"

    def __init__(self, p, color):
        self.type = p
        self.color = color

    def __str__(self):
        if self.color == Color.BLACK:
            return self.type.lower() + ' '
        return self.type + ' '

    def __eq__(self, other):
        return str(self) == str(other)


class Board:
    BLANK = "- "

    def __init__(self):
        self.board = [[Board.BLANK for i in range(8)] for j in range(8)]    # board content
        self.load_default()

    def load_default(self):
        for i in range(8):
            self.board[1][i] = Piece(Piece.PAWN, Color.WHITE)
            self.board[6][i] = Piece(Piece.PAWN, Color.BLACK)
        for info in [[Color.WHITE, 0], [Color.BLACK, 7]]:
            color, row = info
            self.board[row][0] = Piece(Piece.ROOK, color)
            self.board[row][7] = Piece(Piece.ROOK, color)
            self.board[row][1] = Piece(Piece.KNIGHT, color)
            self.board[row][6] = Piece(Piece.KNIGHT, color)
            self.board[row][2] = Piece(Piece.BISHOP, color)
            self.board[row][5] = Piece(Piece.BISHOP, color)
            self.board[row][3] = Piece(Piece.QUEEN, color)
            self.board[row][4] = Piece(Piece.KING, color)

    def __str__(self):
        s = ""
        tmp = []
        for row in self.board:
            tmp.append("".join([str(p) for p in row]) + "\n")
        return "\n".join(tmp[::-1])

    def over(self):
        white_king = Piece(Piece.KING, Color.WHITE)
        black_king = Piece(Piece.KING, Color.BLACK)
        white_king_alive = False
        black_king_alive = False
        for i in range(8):
            for j in range(8):
                if self.board[i][j] == white_king:
                    white_king_alive = True
                if self.board[i][j] == black_king:
                    black_king_alive = True
        if white_king_alive and black_king_alive:
            return False, None
        if white_king_alive:
            return True, Color.WHITE
        if black_king_alive:
            return True, Color.BLACK


    def get_pieces(self):
        """get current pieces on board"""
        total = {
            Color.WHITE:{Piece.KING: [], Piece.QUEEN: [], Piece.BISHOP: [], Piece.KNIGHT: [],
Piece.ROOK: [], Piece.PAWN: []},
            Color.BLACK:{Piece.KING: [], Piece.QUEEN: [], Piece.BISHOP: [], Piece.KNIGHT: [],
Piece.ROOK: [], Piece.PAWN: []}
        }
        # pieces = {Piece.KING: [], Piece.QUEEN: [], Piece.BISHOP: [], Piece.KNIGHT: [],
Piece.ROOK: [], Piece.PAWN: []}
        for i in range(8):
            for j in range(8):
                p = self.board[i][j]
                if type(p) is Piece:
                    color = p.color
                    p_type = p.type
                    total[color][p_type].append([p, (i, j)])
        return total

    def apply(self, move):
        a, b = move
        ai, aj = a
        bi, bj = b
        piece = self.board[ai][aj]
        self.board[ai][aj] = Board.BLANK
        self.board[bi][bj] = piece


def get_opponent(player):
    return Color.BLACK if player == Color.WHITE else Color.WHITE


def successors(game):
    """
    get the possible move and next game state from current
    """
    player = game.current_player
    pieces = game.board.get_pieces()
    my_pieces = pieces[player]
    result = []
    result.extend(pawn_successors(my_pieces[Piece.PAWN], game))
    result.extend(queen_successors(my_pieces[Piece.QUEEN], game))
    result.extend(knight_successors(my_pieces[Piece.KNIGHT], game))
    result.extend(bishop_successors(my_pieces[Piece.BISHOP], game))
    result.extend(rook_successors(my_pieces[Piece.ROOK], game))
    result.extend(king_successors(my_pieces[Piece.KING], game))
    return result


def pawn_successors(pieces, game):
    """
       check pawn of white and black separately
    """
    result = []
    for (p, pos) in pieces:
        if p.color == Color.WHITE:
            white_pawn_successors(p, pos, game, result)
        else:
            black_pawn_successors(p, pos, game, result)
    return result


def white_pawn_successors(piece, pos, game, result):
    """
        for pawn, if it's at beginning position,then it's can move one step or two
                and possible to eat the piece of corner
    """
    board = game.board.board
    i, j = pos
    if i < 7:
        # move forward one step
        p = board[i+1][j]
        if type(p) is not Piece:
            new_game = deepcopy(game)
            new_game.board.board[i][j] = Board.BLANK
            new_game.board.board[i+1][j] = piece
            result.append([[pos, (i+1, j)], new_game])
    if i == 2:
        # move forward two step
        p = board[i+2][j]
        if type(p) is not Piece:
            new_game = deepcopy(game)
            new_game.board.board[i][j] = Board.BLANK
            new_game.board.board[i+2][j] = piece
            result.append([[pos, (i+2, j)], new_game])
    # try eat
    for n_pos in [(i+1, j-1), (i+1, j+1)]:
        if not ok_p(n_pos):
            continue
        pi, pj = n_pos
        p = board[pi][pj]
        if type(p) is Piece:
            if p.color != piece.color:
                new_game = deepcopy(game)
                new_game.board.board[i][j] = Board.BLANK
                new_game.board.board[pi][pj] = piece
                result.append([[pos, n_pos], new_game])
    return result


def black_pawn_successors(piece, pos, game, result):
    """
    for pawn, if it's at beginning position,then it's can move one step or two
            and possible to eat the piece of corner
    """
    board = game.board.board
    i, j = pos
    if i > 0:
        # move forward one step
        p = board[i - 1][j]
        if type(p) is not Piece:
            new_game = deepcopy(game)
            new_game.board.board[i][j] = Board.BLANK
            new_game.board.board[i - 1][j] = piece
            result.append([[pos, (i-1, j)], new_game])
    if i == 6:
        # move forward two step
        p = board[i - 2][j]
        if type(p) is not Piece:
            new_game = deepcopy(game)
            new_game.board.board[i][j] = Board.BLANK
            new_game.board.board[i - 2][j] = piece
            result.append([[pos, (i-2, j)], new_game])
    # try eat
    for n_pos in [(i - 1, j - 1), (i - 1, j + 1)]:
        if not ok_p(n_pos):
            continue
        pi, pj = n_pos
        p = board[pi][pj]
        if type(p) is Piece:
            if p.color != piece.color:
                new_game = deepcopy(game)
                new_game.board.board[i][j] = Board.BLANK
                new_game.board.board[pi][pj] = piece
                result.append([[pos, n_pos], new_game])
    return result


def knight_successors(pieces, game):
    """eight possible next position"""
    result = []
    for (p, pos) in pieces:
        i, j = pos
        next_pos = [(i+2, j+1), (i+1, j+2), (i-2, j-1), (i-1, j-2), (i-2, j+1), (i-1, j+2), (i+2, j-1),
(i+1, j-2)]
        next_pos = [p for p in next_pos if ok_p(p)]
        insert_possible(next_pos, pos, p, game, result, stop=False)
    return result


def queen_successors(pieces, game):
    # """queen can go eight direction"""
    result = []
    for (p, pos) in pieces:
        i, j = pos
        next_pos = [(d, j) for d in range(i + 1, 8)]
        insert_possible(next_pos, pos, p, game, result)

        next_pos = [(d, j) for d in range(0, i)]
        insert_possible(next_pos, pos, p, game, result)

        next_pos = [(i, d) for d in range(j + 1, 8)]
        insert_possible(next_pos, pos, p, game, result)

        next_pos = [(i, d) for d in range(0, j)]
        insert_possible(next_pos, pos, p, game, result)
        next_pos = [(i + k, j + k) for k in range(1, 8)]
        next_pos = [p for p in next_pos if ok_p(p)]
        insert_possible(next_pos, pos, p, game, result)

        next_pos = [(i + k, j - k) for k in range(1, 8)]
        next_pos = [p for p in next_pos if ok_p(p)]
        insert_possible(next_pos, pos, p, game, result)

        next_pos = [(i - k, j - k) for k in range(1, 8)]
        next_pos = [p for p in next_pos if ok_p(p)]
        insert_possible(next_pos, pos, p, game, result)

        next_pos = [(i - k, j + k) for k in range(1, 8)]
        next_pos = [p for p in next_pos if ok_p(p)]
        insert_possible(next_pos, pos, p, game, result)
    return result


def ok_p(p):
    # check if the position is legal
    i, j = p
    return 0<=i<8 and 0<=j<8


def bishop_successors(pieces, game):
    """bishop can go four direction (diag)"""
    result = []
    for (p, pos) in pieces:
        i, j = pos
        next_pos = [(i+k, j+k) for k in range(1, 8)]
        next_pos = [p for p in next_pos if ok_p(p)]
        insert_possible(next_pos, pos, p, game, result)

        next_pos = [(i + k, j - k) for k in range(1, 8)]
        next_pos = [p for p in next_pos if ok_p(p)]
        insert_possible(next_pos, pos, p, game, result)

        next_pos = [(i - k, j - k) for k in range(1, 8)]
        next_pos = [p for p in next_pos if ok_p(p)]
        insert_possible(next_pos, pos, p, game, result)
        next_pos = [(i - k, j + k) for k in range(1, 8)]
        next_pos = [p for p in next_pos if ok_p(p)]
        insert_possible(next_pos, pos, p, game, result)

    return result


def rook_successors(pieces, game):
    """rook can go four direction"""
    result = []
    for (p, pos) in pieces:
        i, j = pos
        next_pos = [(d, j) for d in range(i+1, 8)]
        insert_possible(next_pos, pos, p, game, result)

        next_pos = [(d, j) for d in range(0, i)][::-1]
        insert_possible(next_pos, pos, p, game, result)

        next_pos = [(i, d) for d in range(j+1, 8)]
        insert_possible(next_pos, pos, p, game, result)

        next_pos = [(i, d) for d in range(0, j)][::-1]
        insert_possible(next_pos, pos, p, game, result)
    return result


def king_successors(pieces, game):
    """four possible next position"""
    result = []
    for (p, pos) in pieces:
        i, j = pos
        next_pos = [(i, j + 1), (i + 1, j), (i, j - 1), (i - 1, j)]
        next_pos = [p for p in next_pos if ok_p(p)]
        insert_possible(next_pos, pos, p, game, result, stop=False)
    return result


def insert_possible(pos_list, pos, piece, game, result, stop=True):
    """for a list of position to check if can place the piece, if stop, then when meet a
piece the check stop"""
    board = game.board.board
    oi, oj = pos
    for next_pos in pos_list:
        i, j = next_pos
        p = board[i][j]
        if type(p) is not Piece:
            new_game = deepcopy(game)
            new_game.board.board[i][j] = piece
            new_game.board.board[oi][oj] = Board.BLANK
            result.append([[pos, next_pos], new_game])
        else:
            if p.color != piece.color:
                new_game = deepcopy(game)
                new_game.board.board[i][j] = piece
                new_game.board.board[oi][oj] = Board.BLANK
                result.append([[pos, next_pos], new_game])
            if stop:
                break


def evaluate(game, current_player):
    player = current_player
    pieces = game.board.get_pieces()
    my_pieces = pieces[player]
    oppo_pieces = pieces[get_opponent(player)]

    return score(my_pieces) - score(oppo_pieces)


def score(pieces):
    s = 0
    s += 1 * len(pieces[Piece.PAWN])
    s += 30 * len(pieces[Piece.KNIGHT])
    s += 50 * len(pieces[Piece.ROOK])
    s += 40 * len(pieces[Piece.BISHOP])
    s += 160 * len(pieces[Piece.QUEEN])
    s += 10000*len(pieces[Piece.KING])
    return s


class Game:
    def __init__(self, board, current_player):
        self.board = board
        self.current_player = current_player
        self.winner = ""

    def over(self):
        over, winner = self.board.over()
        self.winner = winner
        return over

    def apply(self, move):
        self.board.apply(move)
        self.current_player = Color.BLACK if self.current_player==Color.WHITE else
Color.WHITE


class BasePlayer:

    def __init__(self, color):
        self.color = color

    def get_move(self, board):
        raise Exception("not implement error")



class MinimaxPlayer(BasePlayer):

    def __init__(self, color, depth):
        super().__init__(color)
        self.player = color
        self.depth = depth

    def get_move(self, game):
        # minimax search with alpha-beta cut
        depth = self.depth
        best_score = -INF
        beta = INF
        best_action = None
        for (move, next_state) in successors(game):
            v = self.min_value(next_state, best_score, beta, depth-1)
            if v > best_score:
                best_score = v
                best_action = move
        return best_action

    def max_value(self, game, alpha, beta, depth):
        if depth <= 0 or game.over():
            return evaluate(game, self.player)
        v = -INF
        for (move, next_state) in successors(game):
            v = max(v, self.min_value(next_state, alpha, beta, depth-1))
            if v >= beta:
                return v
            alpha = max(alpha, v)
        return v

    def min_value(self, game, alpha, beta, depth):
        if depth <= 0 or game.over():
            return evaluate(game, self.player)
        v = INF
        for (move, next_state) in successors(game):
            v = min(v, self.max_value(next_state, alpha, beta, depth-1))
            if v <= alpha:
                return v
            beta = min(beta, v)
        return v


def to_pos(p):
    col, row = p[0], p[1]
    return ( int(row)-1, "abcdefgh".index(col))


class HumanPlayer(BasePlayer):

    def get_move(self, board):
        move = input("input your move (pos,pos, eg, a7,a6):")
        p1, p2 = move.split(",")
        return to_pos(p1), to_pos(p2)       # trans to array index
```

```
def run():
    depth = 4  # control depth of search tree
    current_player = Color.WHITE
    game = Game(Board(), current_player)
    white_player = MinimaxPlayer(Color.WHITE, depth)     # ai player
    #black_player = MinimaxPlayer(Color.BLACK, depth)
    black_player = HumanPlayer(Color.BLACK)       # human input player
    while not game.over():
        # game turn
        if game.current_player == Color.WHITE:
            move = white_player.get_move(game)
        else:
            move = black_player.get_move(game)
        game.apply(move)
        # print(move)
        print(game.board)
        # current_player = Color.BLACK if current_player == Color.WHITE else
Color.WHITE
    print("game over")
    print("winner:", game.winner)


if __name__ == "__main__":
    run()
```