```python
#Chess Brain
import pygame
import chess
# starts up the pygame library
pygame.init()

# Dimensions of the screen
screen_width = 1376
screen_height = 768

# Creating the pygame window
screen = pygame.display.set_mode((screen_width, screen_height))

# text parameters
border_font = pygame.font.Font('freesansbold.ttf', 32)
title_font = pygame.font.Font('freesansbold.ttf', 75)
captures_font = pygame.font.Font('freesansbold.ttf', 25)

# board create
white_piece_list = ["wR", "wKn", "wB", "wQ", "wK", "wP", "wP", "wP", "wP", "wP", "wP", "wP", "wP"]
black_piece_list = ["bR", "bKn", "bB", "bQ", "bK", "bP", "bP", "bP", "bP", "bP", "bP", "bP", "bP"]

# resizes the image
scale = 94
# sets a fps value
clock = pygame.time.Clock()

# imports the image
wPImg = pygame.image.load('Chess Graphics/chess graphics/chess pieces/wP.png')
wKImg = pygame.image.load('Chess Graphics/chess graphics/chess pieces/wK.png')
wQImg = pygame.image.load('Chess Graphics/chess graphics/chess pieces/wQ.png')
wRImg = pygame.image.load('Chess Graphics/chess graphics/chess pieces/wR.png')
wKnImg = pygame.image.load('Chess Graphics/chess graphics/chess pieces/wKn.png')
wBImg = pygame.image.load('Chess Graphics/chess graphics/chess pieces/wB.png')
bPImg = pygame.image.load('Chess Graphics/chess graphics/chess pieces/bP.png')
bKImg = pygame.image.load('Chess Graphics/chess graphics/chess pieces/bK.png')
bQImg = pygame.image.load('Chess Graphics/chess graphics/chess pieces/bQ.png')
bRImg = pygame.image.load('Chess Graphics/chess graphics/chess pieces/bR.png')
bKnImg = pygame.image.load('Chess Graphics/chess graphics/chess pieces/bKn.png')
bBImg = pygame.image.load('Chess Graphics/chess graphics/chess pieces/bB.png')

# sets the window size
# https://pythonprogramming.net/displaying-images-pygame/
gameDisplay = pygame.display.set_mode((1100, 800))

# creates the board
board = [["bR", "bKn", "bB", "bQ", "bK", "bB", "bKn", "bR"],
        ["bP", "bP", "bP", "bP", "bP", "bP", "bP", "bP"],
        [None, None, None, None, None, None, None, None],
        [None, None, None, None, None, None, None, None],
        [None, None, None, None, None, None, None, None],
        [None, None, None, None, None, None, None, None],
        ["wP", "wP", "wP", "wP", "wP", "wP", "wP", "wP"],
        ["wR", "wKn", "wB", "wQ", "wK", "wB", "wKn", "wR"]]

# assigns each image a name
piece_dict = {"wP": wPImg,
              "wK": wKImg,
              "wQ": wQImg,
              "wR": wRImg,
              "wKn": wKnImg,
              "wB": wBImg,
              "bP": bPImg,
              "bK": bKImg,
              "bQ": bQImg,
              "bR": bRImg,
              "bKn": bKnImg,
              "bB": bBImg}


# https://impythonist.wordpress.com/2017/01/01/modeling-a-chessboard-and-mechanics-of-its-pieces-in-python/
# assigning each space on the board a value (ex A4)
chess_map_from_alpha_to_index = {
   "a" : 0,
   "b" : 1,
   "c" : 2,
   "d" : 3,
   "e" : 4,
   "f" : 5,
   "g" : 6,
   "h" : 7}

chess_map_from_index_to_alpha = {
   0: "a",
   1: "b",
   2: "c",
   3: "d",
   4: "e",
   5: "f",
   6: "g",
   7: "h"}

chess_map_from_true_y_to_board_y = {
   0: "8",
   1: "7",
   2: "6",
   3: "5",
   4: "4",
   5: "3",
   6: "2",
   7: "1"}

chess_map_from_board_y_to_true_y = {
   8: "0",
   7: "1",
   6: "2",
   5: "3"
```

```
    4: "4",
    3: "5",
    2: "6",
    1: "7"}

piece_letter_to_name = {
    "P": "pawn",
    "R": "rook",
    "K": "knight",
    "B": "bishop",
    "Q": "queen"}

# capture lists
b_capture_list = []
w_capture_list = []

"""MOVEMENT ALGORITHMS (https://impythonist.wordpress.com/2017/01/01/modeling-a-chessboard-and-mechanics-of-its-pieces-in-python/)"""
# Rook Moves
def getRookMoves(pos, board):
    # A function(positionString, board) that returns the all possible moves of a rook stood on a given position
    column, row = list(pos.strip().lower())
    row = int(row) - 1
    # Chess map from alpha to index function retrieves notation conversion
    column = chess_map_from_alpha_to_index[column]
    x, y = row, column
    possmoves = []

    # Compute the moves in Rank
    for y in range(8):
        if y != column:
            possmoves.append((row, y))

    # Compute the moves in File
    for x in range(8):
        if x != row:
            possmoves.append((x, column))

    # adds all possible move values to a list
    possmoves = ["".join([chess_map_from_index_to_alpha[x[1]], str(x[0] + 1)]) for x in possmoves]
    possmoves.sort()
    return possmoves

# Knight Moves
def getKnightMoves(pos, board):
    # A function(positionString, board) that returns the all possible moves of a knight stood on a given position
    column, row = list(pos.strip().lower())
    row = int(row) - 1
    # Chess map from alpha to index function retrieves notation conversion
    column = chess_map_from_alpha_to_index[column]
    x, y = row, column
    possmoves = []

    # does all possible knight moves; puts them in try and except statements in case the move is off of the board
    try:
        temp = board[x + 1][y - 2]
        possmoves.append([x + 1, y - 2])
    except:
        pass
    try:
        temp = board[x + 2][y - 1]
        possmoves.append([x + 2, y - 1])
    except:
        pass
    try:
        temp = board[x + 2][y + 1]
        possmoves.append([x + 2, y + 1])
    except:
        pass
    try:
       temp = board[x + 1][y + 2]
       possmoves.append([x + 1, y + 2])
    except:
        pass
    try:
        temp = board[x - 1][y + 2]
        possmoves.append([x - 1, y + 2])
    except:
        pass
    try:
        temp = board[x - 2][y + 1]
        possmoves.append([x - 2, y + 1])
    except:
        pass
    try:
        temp = board[x - 2][y - 1]
        possmoves.append([x - 2, y - 1])
    except:
        pass
    try:
        temp = board[x - 1][y - 2]
        possmoves.append([x - 1, y - 2])
    except:
        pass

    # Filter all negative values
    temp = [x for x in possmoves if x[0] >= 0 and x[1] >= 0]
    allPossibleMoves = ["".join([chess_map_from_index_to_alpha[x[1]], str(x[0] + 1)]) for x in temp]
    allPossibleMoves.sort()
    return allPossibleMoves

# Bishop Moves
def getBishopMoves(pos, board):
    # A function(positionString, board) that returns the all possible moves of a bishop stood on a given position
    column, row = list(pos.strip().lower())
    row = int(row) - 1
    column = chess_map_from_alpha_to_index[column]
```

```
    x, y = row, column
    possmoves = []

    # moving diagonal all 4 ways
    for i in range(8):
        try:
            temp = board[x + i][y + i]
            possmoves.append([x + i, y + i])
        except:
            pass

        try:
            temp = board[x - i][y + i]
            possmoves.append([x - i, y + i])
        except:
            pass

        try:
            temp = board[x + i][y - i]
            possmoves.append([x + i, y - i])
        except:
            pass

        try:
            temp = board[x - i][y - i]
            possmoves.append([x - i, y - i])
        except:
            pass

    # Filter all negative values
    temp = [x for x in possmoves if x[0] >= 0 and x[1] >= 0]
    allPossibleMoves = ["".join([chess_map_from_index_to_alpha[x[1]], str(x[0] + 1)]) for x in temp]
    allPossibleMoves.sort()
    return allPossibleMoves

# QUEEN MOVES
def getQueenMoves(pos, board):
    # adding the bishop moves and rook moves to get queen moves
    bishop_subset = getBishopMoves(pos, board)
    rook_subset = getRookMoves(pos, board)
    Queen_Moves = bishop_subset + rook_subset
    return Queen_Moves

# King Moves
def getKingMoves(pos, board):
# A function(positionString, board) that returns the all possible moves of a king stood on a given position
    column, row = list(pos.strip().lower())
    row = int(row) - 1
    column = chess_map_from_alpha_to_index[column]
    x, y = row, column
    possmoves = []
    # does all the possible moves around the king besides the square the king is on
    for i in range(-1, 2):
        for j in range (-1, 2):
            if (i != 0) or (j != 0):
                try:
                    temp = board[x + i][y + j]
                    possmoves.append([x + i, y + j])
                except:
                    pass
    # Filter all negative values
    temp = [x for x in possmoves if x[0] >= 0 and x[1] >= 0]
    allPossibleMoves = ["".join([chess_map_from_index_to_alpha[x[1]], str(x[0] + 1)]) for x in temp]
    allPossibleMoves.sort()
    return allPossibleMoves


# The chess board
chessboard = pygame.image.load("Chess Graphics/chess graphics/chess board/chessboard.jpg").convert()
screen.blit(chessboard, (10, 10))
# centers the pieces in the square
x_offset = 27
y_offset = 27

# Draws the board
def draw_board(board, color_to_move, moves, check, checkmate):
    # fills background with grey color
    screen.fill((75, 75, 75))
    # draws the chessboard on the screen
    screen.blit(chessboard, (10, 10))
    # draws the pieces in their squares
    for y in range(len(board)):
        for x in range(len(board[0])):
            if board[y][x] != None:
                screen.blit(piece_dict[board[y][x]], (x_offset + scale*x, y_offset + scale*y))
    letters = ["a", "b", "c", "d", "e", "f", "g", "h"]
    numbers = ["1", "2", "3", "4", "5", "6", "7", "8"]
    numbers.reverse()

    # Draw letters on the bottom of the screen
    border_letters = [border_font.render(letter, True, (0, 0, 0)) for letter in letters]
    [screen.blit(border_letters[i], (50+scale*i, 763)) for i in range(len(letters))]

    # Draws numbers on the right of the screen
    border_numbers = [border_font.render(number, True, (0, 0, 0)) for number in numbers]
    [screen.blit(border_numbers[i], (763, 45+scale*i)) for i in range(len(numbers))]

    # Draws the title "Chess!" on the right side of the screen
    title = title_font.render("CHESS!", True, (0, 0, 0))
    screen.blit(title, (800, 30))

    # Draws the current player text
    current_player_turn = border_font.render("Current Player:", True, (0, 0, 0))
    screen.blit(current_player_turn, (815, 120))

    if color_to_move == "w":
        player = border_font.render("white", True, (255, 255, 255))
    else:
```

```
        player = border_font.render("black", True, (0, 0, 0))
    screen.blit(player, (890, 160))

    # Draws the moves counter
    moves = border_font.render("Moves: "+str(moves), True, (0, 0, 0))
    screen.blit(moves, (860, 220))

    # Draws Check on the right side of the screen
    if check == True:
        check_text = border_font.render("Check!", True, (255, 0, 0))
        screen.blit(check_text, (880, 270))

    # Draws Checkmate in the middle of the screen along with the color of who won
    if checkmate == True:
        screen.fill((75, 75, 75), (155, 340, 465, 125))
        checkmate_text = title_font.render("Checkmate!", True, (255, 0, 0))
        screen.blit(checkmate_text, (165, 350))
        if color_to_move == "w":
            winner_text = border_font.render("Black Wins!", True, (0, 0, 0))
            screen.blit(winner_text, (200, 420))
        if color_to_move == "b":
            winner_text = border_font.render("White Wins!", True, (0, 0, 0))
            screen.blit(winner_text, (200, 420))

    divider = title_font.render("______", True, (255, 255, 255))
    screen.blit(divider, (800, 190))

    # Draws which pieces have been captured
    capture_title_text = border_font.render("Captures", True, (0, 0, 0))
    screen.blit(capture_title_text, (865, 310))

    black_capture_text = border_font.render("Black", True, (0, 0, 0))
    screen.blit(black_capture_text, (810, 350))

    white_capture_text = border_font.render("White", True, (255, 255, 255))
    screen.blit(white_capture_text, (950, 350))

    black_captures = [captures_font.render(str(capture), True, (30, 30, 30)) for capture in b_capture_list]
    [screen.blit(black_captures[i], (820, 390+30*i)) for i in range(len(b_capture_list))]

    white_captures = [captures_font.render(str(capture), True, (30, 30, 30)) for capture in w_capture_list]
    [screen.blit(white_captures[i], (960, 390+30*i)) for i in range(len(w_capture_list))]

# Checks to see if the board is the same as the old state
def checkBoard(old_state, current_state):
    checkBoard = True
    for y in range(len(current_state)):
        for x in range(len(current_state[0])):
            if old_state[y][x] != current_state[y][x]:
                checkBoard = False
    return checkBoard

# Finds all the possible moves for a given piece
def possibleMoves(board, x, y, color_to_move):
    piece_y = y
    piece_x = x
    pos_moves = []
    if board[piece_y][piece_x] != None:
        if str(board[piece_y][piece_x])[0] != color_to_move:
            return pos_moves
        """
        dest_y = playerinputclicks[1][0]
        dest_x = playerinputclicks[1][1]
        """

        # Changes the board coordinates to chess notation
        board_piece_y = chess_map_from_true_y_to_board_y[piece_y]
        alpha_piece_x = chess_map_from_index_to_alpha[piece_x]
        """
        board_dest_y = chess_map_from_true_y_to_board_y[dest_y]
        alpha_dest_x = chess_map_from_index_to_alpha[dest_x]
        """
        pos_moves = []

        # Finds the possible moves for a pawn
        if str(board[piece_y][piece_x])[1] == "P":
            if str(board[piece_y][piece_x])[0] == "w":
                # Checks for diagonal capturing
                if piece_y != 0:
                    if board[piece_y-1][piece_x] == None:
                        pos_moves.append(alpha_piece_x+str((int(board_piece_y)+1)))
                    if piece_x != 7:
                        # Checks to see if the first letter of the string in a board position given a x and y coordinate; can be used to check the col
                        if str(board[piece_y-1][piece_x+1])[0] == "b":
                            pos_moves.append(chess_map_from_index_to_alpha[piece_x+1]+chess_map_from_true_y_to_board_y[piece_y-1])
                    if piece_x != 0:
                        if str(board[piece_y-1][piece_x-1])[0] == "b":
                            pos_moves.append(chess_map_from_index_to_alpha[piece_x-1]+chess_map_from_true_y_to_board_y[piece_y-1])
                    if piece_y == 6:
                        pos_moves.append(alpha_piece_x+"4")
                        if board[piece_y-2][piece_x] != None:
                            pos_moves.remove(alpha_piece_x+"4")
                # Does the promotion of a pawn to a Queen
                if piece_y == 0:
                    board[piece_y][piece_x] = "wQ"

            # Black and white are the same; only difference is the y coordinates and directions for capturing
            if str(board[piece_y][piece_x])[0] == "b":
                if piece_y != 7:
                    if board[piece_y+1][piece_x] == None:
                        pos_moves.append(alpha_piece_x+str((int(board_piece_y)-1)))
                    if piece_x != 7:
                        if str(board[piece_y+1][piece_x+1])[0] == "w":
                            pos_moves.append(chess_map_from_index_to_alpha[piece_x+1]+chess_map_from_true_y_to_board_y[piece_y+1])
                    if piece_x != 0:
                        if str(board[piece_y+1][piece_x-1])[0] == "w":
                            pos_moves.append(chess_map_from_index_to_alpha[piece_x-1]+chess_map_from_true_y_to_board_y[piece_y+1])
```

```
                if piece_y == 1:
                    pos_moves.append(alpha_piece_x+"5")
                    if board[piece_y+2][piece_x] != None:
                        pos_moves.remove(alpha_piece_x+"5")
            if piece_y == 7:
                board[piece_y][piece_x] = "bQ"

    # If the piece is a rook; the different lists organizes the moves to be radially outward from the rook
    if str(board[piece_y][piece_x])[1] == "R":
        pos_moves = getRookMoves(alpha_piece_x+board_piece_y, board)
        upper_y_moves = []
        lower_y_moves = []
        left_x_moves = []
        right_x_moves = []
        adj_pos_moves = []

        for move in pos_moves:
            if str(move)[0] == alpha_piece_x and int(str(move)[1]) > int(board_piece_y):
                upper_y_moves.append(move)
            if str(move)[0] == alpha_piece_x and int(str(move)[1]) < int(board_piece_y):
                lower_y_moves.append(move)
            if str(move)[1] == board_piece_y and chess_map_from_alpha_to_index[str(move)[0]] > piece_x:
                right_x_moves.append(move)
            if str(move)[1] == board_piece_y and chess_map_from_alpha_to_index[str(move)[0]] < piece_x:
                left_x_moves.append(move)

        lower_y_moves.reverse()
        left_x_moves.reverse()

        # It shrinks each list so that it can't go through pieces
        for move in upper_y_moves:
            x_val = chess_map_from_alpha_to_index[move[0]]
            board_y = move[1]
            y_val = chess_map_from_board_y_to_true_y[int(board_y)]

            if board[int(y_val)][int(x_val)] == None:
                adj_pos_moves.append(move)
            else:
                adj_pos_moves.append(move)
                break

        for move in lower_y_moves:
            x_val = chess_map_from_alpha_to_index[move[0]]
            board_y = move[1]
            y_val = chess_map_from_board_y_to_true_y[int(board_y)]

            if board[int(y_val)][int(x_val)] == None:
                adj_pos_moves.append(move)
            else:
                adj_pos_moves.append(move)
                break

        for move in right_x_moves:
            x_val = chess_map_from_alpha_to_index[move[0]]
            board_y = move[1]
            y_val = chess_map_from_board_y_to_true_y[int(board_y)]

            if board[int(y_val)][int(x_val)] == None:
                adj_pos_moves.append(move)
            else:
                adj_pos_moves.append(move)
                break

        for move in left_x_moves:
            x_val = chess_map_from_alpha_to_index[move[0]]
            board_y = move[1]
            y_val = chess_map_from_board_y_to_true_y[int(board_y)]

            if board[int(y_val)][int(x_val)] == None:
                adj_pos_moves.append(move)
            else:
                adj_pos_moves.append(move)
                break

        pos_moves = adj_pos_moves

    # If the piece is a queen; the different lists organizes the moves to be radially outward from the queen
    if str(board[piece_y][piece_x])[1] == "Q":
        pos_moves = getQueenMoves(alpha_piece_x+board_piece_y, board)

        upper_y_moves = []
        lower_y_moves = []
        left_x_moves = []
        right_x_moves = []
        adj_pos_moves = []

        for move in pos_moves:
            if str(move)[0] == alpha_piece_x and int(str(move)[1]) > int(board_piece_y):
                upper_y_moves.append(move)
            if str(move)[0] == alpha_piece_x and int(str(move)[1]) < int(board_piece_y):
                lower_y_moves.append(move)
            if str(move)[1] == board_piece_y and chess_map_from_alpha_to_index[str(move)[0]] > piece_x:
                right_x_moves.append(move)
            if str(move)[1] == board_piece_y and chess_map_from_alpha_to_index[str(move)[0]] < piece_x:
                left_x_moves.append(move)

        lower_y_moves.reverse()
        left_x_moves.reverse()

        # It shrinks each list so that it can't go through pieces
        for move in upper_y_moves:
            x_val = chess_map_from_alpha_to_index[move[0]]
            board_y = move[1]
            y_val = chess_map_from_board_y_to_true_y[int(board_y)]

            if board[int(y_val)][int(x_val)] == None:
                adj_pos_moves.append(move)
```

```
                else:
                    adj_pos_moves.append(move)
                    break

            for move in lower_y_moves:
                x_val = chess_map_from_alpha_to_index[move[0]]
                board_y = move[1]
                y_val = chess_map_from_board_y_to_true_y[int(board_y)]

                if board[int(y_val)][int(x_val)] == None:
                    adj_pos_moves.append(move)
                else:
                    adj_pos_moves.append(move)
                    break

            for move in right_x_moves:
                x_val = chess_map_from_alpha_to_index[move[0]]
                board_y = move[1]
                y_val = chess_map_from_board_y_to_true_y[int(board_y)]

                if board[int(y_val)][int(x_val)] == None:
                    adj_pos_moves.append(move)
                else:
                    adj_pos_moves.append(move)
                    break

            for move in left_x_moves:
                x_val = chess_map_from_alpha_to_index[move[0]]
                board_y = move[1]
                y_val = chess_map_from_board_y_to_true_y[int(board_y)]

                if board[int(y_val)][int(x_val)] == None:
                    adj_pos_moves.append(move)
                else:
                    adj_pos_moves.append(move)
                    break

            # These lists are for each diagonal direction; organizing radially outward from the queen
            upper_right_moves = []
            upper_left_moves = []
            lower_right_moves = []
            lower_left_moves = []

            for move in pos_moves:
                if str(move)[0] > alpha_piece_x and int(str(move)[1]) > int(board_piece_y):
                    upper_right_moves.append(move)
                if str(move)[0] > alpha_piece_x and int(str(move)[1]) < int(board_piece_y):
                    lower_right_moves.append(move)
                if str(move)[0] < alpha_piece_x and int(str(move)[1]) > int(board_piece_y):
                    upper_left_moves.append(move)
                if str(move)[0] < alpha_piece_x and int(str(move)[1]) < int(board_piece_y):
                    lower_left_moves.append(move)

            upper_left_moves.reverse()
            lower_left_moves.reverse()

            # It shrinks each list so that it can't go through pieces
            for move in upper_right_moves:
                x_val = chess_map_from_alpha_to_index[move[0]]
                board_y = move[1]
                y_val = chess_map_from_board_y_to_true_y[int(board_y)]

                if board[int(y_val)][int(x_val)] == None:
                    adj_pos_moves.append(move)
                else:
                    adj_pos_moves.append(move)
                    break

            for move in upper_left_moves:
                x_val = chess_map_from_alpha_to_index[move[0]]
                board_y = move[1]
                y_val = chess_map_from_board_y_to_true_y[int(board_y)]

                if board[int(y_val)][int(x_val)] == None:
                    adj_pos_moves.append(move)
                else:
                    adj_pos_moves.append(move)
                    break

            for move in lower_right_moves:
                x_val = chess_map_from_alpha_to_index[move[0]]
                board_y = move[1]
                y_val = chess_map_from_board_y_to_true_y[int(board_y)]

                if board[int(y_val)][int(x_val)] == None:
                    adj_pos_moves.append(move)
                else:
                    adj_pos_moves.append(move)
                    break

            for move in lower_left_moves:
                x_val = chess_map_from_alpha_to_index[move[0]]
                board_y = move[1]
                y_val = chess_map_from_board_y_to_true_y[int(board_y)]

                if board[int(y_val)][int(x_val)] == None:
                    adj_pos_moves.append(move)
                else:
                    adj_pos_moves.append(move)
                    break

            pos_moves = adj_pos_moves

        # Same as the other pieces but for bishop
        if str(board[piece_y][piece_x])[1] == "B":
            pos_moves = getBishopMoves(alpha_piece_x+board_piece_y, board)
            upper_right_moves = []
            upper_left_moves = []
```

```
            upper_left_moves = []
            lower_right_moves = []
            lower_left_moves = []
            adj_pos_moves = []

            for move in pos_moves:
                if str(move)[0] > alpha_piece_x and int(str(move)[1]) > int(board_piece_y):
                    upper_right_moves.append(move)
                if str(move)[0] > alpha_piece_x and int(str(move)[1]) < int(board_piece_y):
                    lower_right_moves.append(move)
                if str(move)[0] < alpha_piece_x and int(str(move)[1]) > int(board_piece_y):
                    upper_left_moves.append(move)
                if str(move)[0] < alpha_piece_x and int(str(move)[1]) < int(board_piece_y):
                    lower_left_moves.append(move)

            upper_left_moves.reverse()
            lower_left_moves.reverse()

            for move in upper_right_moves:
                x_val = chess_map_from_alpha_to_index[move[0]]
                board_y = move[1]
                y_val = chess_map_from_board_y_to_true_y[int(board_y)]

                if board[int(y_val)][int(x_val)] == None:
                    adj_pos_moves.append(move)
                else:
                    adj_pos_moves.append(move)
                    break

            for move in upper_left_moves:
                x_val = chess_map_from_alpha_to_index[move[0]]
                board_y = move[1]
                y_val = chess_map_from_board_y_to_true_y[int(board_y)]

                if board[int(y_val)][int(x_val)] == None:
                    adj_pos_moves.append(move)
                else:
                    adj_pos_moves.append(move)
                    break

            for move in lower_right_moves:
                x_val = chess_map_from_alpha_to_index[move[0]]
                board_y = move[1]
                y_val = chess_map_from_board_y_to_true_y[int(board_y)]

                if board[int(y_val)][int(x_val)] == None:
                    adj_pos_moves.append(move)
                else:
                    adj_pos_moves.append(move)
                    break

            for move in lower_left_moves:
                x_val = chess_map_from_alpha_to_index[move[0]]
                board_y = move[1]
                y_val = chess_map_from_board_y_to_true_y[int(board_y)]

                if board[int(y_val)][int(x_val)] == None:
                    adj_pos_moves.append(move)
                else:
                    adj_pos_moves.append(move)
                    break

            pos_moves = adj_pos_moves

        # If the piece is a king or a knight
        if str(board[piece_y][piece_x])[1] == "K":
            # If statement determines if it is a knight, otherwise it is a king
            if len(str(board[piece_y][piece_x])) == 3:
                pos_moves = getKnightMoves(alpha_piece_x+board_piece_y, board)
            else:
                pos_moves = getKingMoves(alpha_piece_x+board_piece_y, board)
    return pos_moves

# Checks to see if the King of the enemy team is in Check
def checkCheck(board, color_to_move):
    pos_moves = []

    if color_to_move == "w":
        king_color = "b"
    if color_to_move == "b":
        king_color = "w"
    # king_cords = [0, 4]
    # Finding all the possible moves for one team
    for y in range(len(board)):
        for x in range(len(board[0])):
            pos_moves.append(possibleMoves(board, x, y, color_to_move))
    # Finds the coordinates of the enemy King
    for y in range(len(board)):
        try: king_cords = [y,board[y].index(king_color+"K")]
        except: pass

    # Alpha King Cords converts the enemy King coordinates into chess notation
    alpha_king_cords = str(chess_map_from_index_to_alpha[king_cords[1]]+chess_map_from_true_y_to_board_y[king_cords[0]])
    # Checks to see if one of the possible moves is to capture the enemy King

    check = False
    for i in pos_moves:
        if alpha_king_cords in i:
            check = True
    return check

# Checks to see if there is a checkmate
def checkCheckmate(board, color_to_move):

    #checkCheck(board, color_to_move)

    # Creates a fake board to manipulate without affecting the real board
    test_board = [[board[y][x] for x in range(len(board[0]))] for y in range(len(board))]
```

```
    checkmate = False
    pos_moves = []
    uncheckables = []

    if color_to_move == "w":
        for y in range(len(board)):
            for x in range(len(board[0])):
                # Finds the possible moves for every white piece on the board
                pos_moves = []
                if str(board[y][x])[0] == "w":
                    pos_moves.extend(possibleMoves(board, x, y, color_to_move))
                    # Tests to see if each of the moves causes resolve to check or not
                    for move in pos_moves:
                        test_board = [[board[y][x] for x in range(len(board[0]))] for y in range(len(board))]
                        dest_x = int(chess_map_from_alpha_to_index[str(move)[0]])
                        dest_y = int(chess_map_from_board_y_to_true_y[int(str(move)[1])])
                        if board[y][x] != None:
                            if str(board[dest_y][dest_x])[0] != str(board[y][x])[0]:
                                test_board[dest_y][dest_x] = test_board[y][x]
                                test_board[y][x] = None
                            if checkCheck(test_board, "b") == False:
                                uncheckables.append(move)
    # Same things as above but for black pieces
    if color_to_move == "b":
        for y in range(len(board)):
            for x in range(len(board[0])):
                pos_moves = []
                if str(board[y][x])[0] == "b":
                    pos_moves.extend(possibleMoves(board, x, y, color_to_move))
                    for i in pos_moves:
                        test_board = [[board[y][x] for x in range(len(board[0]))] for y in range(len(board))]
                        dest_x = int(chess_map_from_alpha_to_index[str(i)[0]])
                        dest_y = int(chess_map_from_board_y_to_true_y[int(str(i)[1])])
                        if board[y][x] != None:
                            if str(board[dest_y][dest_x])[0] != str(board[y][x])[0]:
                                test_board[dest_y][dest_x] = test_board[y][x]
                                test_board[y][x] = None
                            if checkCheck(test_board, "w") == False:
                                uncheckables.append(i)
    # If no safe moves are found, checkmate
    if len(uncheckables) == 0:
        checkmate = True
    return checkmate

# Finds the possible moves but accounts for checks and captures; prevents players from moving into check or capturing their own pieces
def makeMove(board, playerinputclicks, color_to_move):
    piece_y = playerinputclicks[0][0]
    piece_x = playerinputclicks[0][1]

    if color_to_move == "w":
        check_color = "b"
    if color_to_move == "b":
        check_color = "w"


    check_board = [[board[y][x] for x in range(len(board[0]))] for y in range(len(board))]

    if board[piece_y][piece_x] != None:
        if str(board[piece_y][piece_x])[0] != color_to_move:
            return board
        dest_y = playerinputclicks[1][0]
        dest_x = playerinputclicks[1][1]

        board_piece_y = chess_map_from_true_y_to_board_y[piece_y]
        alpha_piece_x = chess_map_from_index_to_alpha[piece_x]

        board_dest_y = chess_map_from_true_y_to_board_y[dest_y]
        alpha_dest_x = chess_map_from_index_to_alpha[dest_x]

        # Finds every possible move for the selected piece; if destination square is one of the possible moves, it is allowed
        pos_moves = possibleMoves(board, piece_x, piece_y, color_to_move)
        for i in pos_moves:
            if(i == alpha_dest_x+board_dest_y):
                if board[piece_y][piece_x] != None:
                    # Lines 824 - 836 simulates a move in order to see if the player is trying to move into check
                    # Prevents a piece from capturing a piece on the same team
                    if str(board[dest_y][dest_x])[0] == str(board[piece_y][piece_x])[0]:
                        pass
                    # Checks to see if the piece is trying to capture an enemy piece
                    elif str(board[dest_y][dest_x])[0] != str(board[piece_y][piece_x])[0]:
                        check_board[dest_y][dest_x] = check_board[piece_y][piece_x]
                        check_board[piece_y][piece_x] = None
                    # Allows the piece to move onto a blank square
                    else:
                        check_board[dest_y][dest_x] = check_board[piece_y][piece_x]
                        check_board[piece_y][piece_x] = None

                    # Checks to see if the player moved a piece into check
                    if checkCheck(check_board, check_color) == True:
                        return board
                    # Actually moving a piece; same conditions as above
                    if str(board[dest_y][dest_x])[0] == str(board[piece_y][piece_x])[0]:
                        pass
                    elif str(board[dest_y][dest_x])[0] != str(board[piece_y][piece_x])[0]:
                        # Adds captured pieces to a list for display
                        if str(board[dest_y][dest_x])[0] == "b":
                            b_capture_list.append(piece_letter_to_name[str(board[dest_y][dest_x])[1]])
                        if str(board[dest_y][dest_x])[0] == "w":
                            w_capture_list.append(piece_letter_to_name[str(board[dest_y][dest_x])[1]])
                        board[dest_y][dest_x] = board[piece_y][piece_x]
                        board[piece_y][piece_x] = None
                    else:
                        board[dest_y][dest_x] = board[piece_y][piece_x]
                        board[piece_y][piece_x] = None
                else:
                    pass

    # Checks to see if there is checkmate
```

```
    checkMate = checkCheckmate(board, color_to_move)
    return board

# Main game loop things https://levelup.gitconnected.com/chess-python-ca4532c7f5a4 and https://www.youtube.com/watch?v=o24J3WcBGLg
running = True
selectedsquare = ()
playerinputclicks = []

color_to_move = "w"

# Creates a copy of the board for comparison in the board check function
old_state = [[board[y][x] for x in range(len(board[0]))] for y in range(len(board))]
check = False
checkmate = False

moves = 0
while (running): #press end game then loop stops
    # Creates a list of all inputs from user computer
    for event in pygame.event.get():
        # Closes the window if the player clicks the x in the top right
        if event.type == pygame.QUIT:
            running = False
        # Checks if the player is clicking on a square
        elif event.type == pygame.MOUSEBUTTONDOWN:
            # Finds the location of the mouse on the screen
            location = pygame.mouse.get_pos()
            # Maps the location in a specific square
            col = location[0] // 100 #sqsize = height // dimesion (8)
            row = location[1] // 100

            selectedsquare = (row, col)
            playerinputclicks.append(selectedsquare)
            if selectedsquare == (row, col):
                selectedsquare = ()
                #playerinputclicks = []
                # Checks to see if the player has also chosen a destination square
                if len(playerinputclicks) >= 2:
                    board = makeMove(board, playerinputclicks, color_to_move)
                    checkmate = checkCheckmate(board, color_to_move)
                    draw_board(board, color_to_move, moves, check, checkmate)
                    selectedsquare = ()
                    playerinputclicks = []

            # Alternates the players turn only if the board has changed (Allows for a player to pick up and put a piece back down)
            if not checkBoard(old_state, board):
                check = checkCheck(board, color_to_move)
                if color_to_move == "w":
                    color_to_move = "b"
                elif color_to_move == "b":
                    color_to_move = "w"
                old_state = [[board[y][x] for x in range(len(board[0]))] for y in range(len(board))]
                moves += 1

    # Draws the board
    draw_board(board, color_to_move, moves, check, checkmate)
    # Updates the screen
    pygame.display.update()
    # Sets FPS to 60
    clock.tick(60)
pygame.quit()
```