

```

1 #Chess Brain
2 import pygame
3 import chess
4 # starts up the pygame library
5 pygame.init()
6
7 # Dimensions of the screen
8 screen_width = 1376
9 screen_height = 768
10
11 # Creating the pygame window
12 screen = pygame.display.set_mode((screen_width, screen_height))
13
14 # text parameters
15 border_font = pygame.font.Font('freesansbold.ttf', 32)
16 title_font = pygame.font.Font('freesansbold.ttf', 75)
17 captures_font = pygame.font.Font('freesansbold.ttf', 25)
18
19 # board create
20 white_piece_list = ["wR", "wKn", "wB", "wQ", "wK", "wP", "wP", "wP", "wP", "wP", "wP", "wP", "wP"]
21 black_piece_list = ["bR", "bKn", "bB", "bQ", "bK", "bP", "bP", "bP", "bP", "bP", "bP", "bP", "bP"]
22
23 # resizes the image
24 scale = 94
25 # sets a fps value
26 clock = pygame.time.Clock()
27
28 # imports the image
29 wPImg = pygame.image.load('Chess Graphics/chess graphics/chess pieces/wP.png')
30 wKImg = pygame.image.load('Chess Graphics/chess graphics/chess pieces/wK.png')
31 wQImg = pygame.image.load('Chess Graphics/chess graphics/chess pieces/wQ.png')
32 wRImg = pygame.image.load('Chess Graphics/chess graphics/chess pieces/wR.png')
33 wKnImg = pygame.image.load('Chess Graphics/chess graphics/chess pieces/wKn.png')
34 wBImg = pygame.image.load('Chess Graphics/chess graphics/chess pieces/wB.png')
35 bPImg = pygame.image.load('Chess Graphics/chess graphics/chess pieces/bP.png')
36 bKImg = pygame.image.load('Chess Graphics/chess graphics/chess pieces/bK.png')
37 bQImg = pygame.image.load('Chess Graphics/chess graphics/chess pieces/bQ.png')
38 bRImg = pygame.image.load('Chess Graphics/chess graphics/chess pieces/bR.png')
39 bKnImg = pygame.image.load('Chess Graphics/chess graphics/chess pieces/bKn.png')
40 bBImg = pygame.image.load('Chess Graphics/chess graphics/chess pieces/bB.png')
41
42 # sets the window size
43 # https://pythonprogramming.net/displaying-images-pygame/
44 gameDisplay = pygame.display.set_mode((1100, 800))
45
46 # creates the board
47 board = [[["bR", "bKn", "bB", "bQ", "bK", "bB", "bKn", "bR"],
48           ["bP", "bP", "bP", "bP", "bP", "bP", "bP", "bP"],
49           [None, None, None, None, None, None, None, None],
50           [None, None, None, None, None, None, None, None],
51           [None, None, None, None, None, None, None, None],
52           [None, None, None, None, None, None, None, None],
53           ["wP", "wP", "wP", "wP", "wP", "wP", "wP", "wP"],
54           ["wR", "wKn", "wB", "wQ", "wK", "wB", "wKn", "wR"]]]
55
56 # assigns each image a name
57 piece_dict = {"wP": wPImg,
58               "wK": wKImg,
59               "wQ": wQImg,
60               "wR": wRImg,
61               "wKn": wKnImg,
62               "wB": wBImg,
63               "bP": bPImg,
64               "bK": bKImg,
65               "bQ": bQImg,
66               "bR": bRImg,
67               "bKn": bKnImg,
68               "bB": bBImg}
69
70
71 # https://impythonist.wordpress.com/2017/01/01/modeling-a-chessboard-and-mechanics-of-its-pieces-in-python/
72 # assigning each space on the board a value (ex A4)
73 chess_map_from_alpha_to_index = {
74     "a" : 0,
75     "b" : 1,
76     "c" : 2,
77     "d" : 3,
78     "e" : 4,
79     "f" : 5,
80     "g" : 6,
81     "h" : 7}
82
83 chess_map_from_index_to_alpha = {
84     0: "a",
85     1: "b",
86     2: "c",
87     3: "d",
88     4: "e",
89     5: "f",
90     6: "g",
91     7: "h"}
92
93 chess_map_from_true_y_to_board_y = {
94     0: "8",
95     1: "7",
96     2: "6",
97     3: "5",
98     4: "4",
99     5: "3",
100    6: "2",
101    7: "1"}
102
103 chess_map_from_board_y_to_true_y = {
104     8: "0",
105     7: "1",
106     6: "2",
107     5: "3"

```

```

107: 0: "0",
108: 4: "4",
109: 3: "5",
110: 2: "6",
111: 1: "7"}
112:
113: piece_letter_to_name = {
114:     "P": "pawn",
115:     "R": "rook",
116:     "K": "knight",
117:     "B": "bishop",
118:     "Q": "queen"}
119:
120: # capture lists
121: b_capture_list = []
122: w_capture_list = []
123:
124: """MOVEMENT ALGORITHMS (https://impythonist.wordpress.com/2017/01/01/modeling-a-chessboard-and-mechanics-of-its-pieces-in-python/)"""
125: # Rook Moves
126: def getRookMoves(pos, board):
127:     # A function(positionString, board) that returns the all possible moves of a rook stood on a given position
128:     column, row = list(pos.strip().lower())
129:     row = int(row) - 1
130:     # Chess map from alpha to index function retrieves notation conversion
131:     column = chess_map_from_alpha_to_index[column]
132:     x, y = row, column
133:     possmoves = []
134:
135:     # Compute the moves in Rank
136:     for y in range(8):
137:         if y != column:
138:             possmoves.append((row, y))
139:
140:     # Compute the moves in File
141:     for x in range(8):
142:         if x != row:
143:             possmoves.append((x, column))
144:
145:     # adds all possible move values to a list
146:     possmoves = ["".join([chess_map_from_index_to_alpha[x[1]], str(x[0] + 1)]) for x in possmoves]
147:     possmoves.sort()
148:     return possmoves
149:
150: # Knight Moves
151: def getKnightMoves(pos, board):
152:     # A function(positionString, board) that returns the all possible moves of a knight stood on a given position
153:     column, row = list(pos.strip().lower())
154:     row = int(row) - 1
155:     # Chess map from alpha to index function retrieves notation conversion
156:     column = chess_map_from_alpha_to_index[column]
157:     x, y = row, column
158:     possmoves = []
159:
160:     # does all possible knight moves; puts them in try and except statements in case the move is off of the board
161:     try:
162:         temp = board[x + 1][y - 2]
163:         possmoves.append([x + 1, y - 2])
164:     except:
165:         pass
166:     try:
167:         temp = board[x + 2][y - 1]
168:         possmoves.append([x + 2, y - 1])
169:     except:
170:         pass
171:     try:
172:         temp = board[x + 2][y + 1]
173:         possmoves.append([x + 2, y + 1])
174:     except:
175:         pass
176:     try:
177:         temp = board[x + 1][y + 2]
178:         possmoves.append([x + 1, y + 2])
179:     except:
180:         pass
181:     try:
182:         temp = board[x - 1][y + 2]
183:         possmoves.append([x - 1, y + 2])
184:     except:
185:         pass
186:     try:
187:         temp = board[x - 2][y + 1]
188:         possmoves.append([x - 2, y + 1])
189:     except:
190:         pass
191:     try:
192:         temp = board[x - 2][y - 1]
193:         possmoves.append([x - 2, y - 1])
194:     except:
195:         pass
196:     try:
197:         temp = board[x - 1][y - 2]
198:         possmoves.append([x - 1, y - 2])
199:     except:
200:         pass
201:
202:     # Filter all negative values
203:     temp = [x for x in possmoves if x[0] >= 0 and x[1] >= 0]
204:     allPossibleMoves = ["".join([chess_map_from_index_to_alpha[x[1]], str(x[0] + 1)]) for x in temp]
205:     allPossibleMoves.sort()
206:     return allPossibleMoves
207:
208: # Bishop Moves
209: def getBishopMoves(pos, board):
210:     # A function(positionString, board) that returns the all possible moves of a bishop stood on a given position
211:     column, row = list(pos.strip().lower())
212:     row = int(row) - 1
213:     column = chess_map_from_alpha_to_index[column]
214:     x, y = row, column

```

```

214     x, y = row, column
215     possmoves = []
216
217     # moving diagonal all 4 ways
218     for i in range(8):
219         try:
220             temp = board[x + i][y + i]
221             possmoves.append([x + i, y + i])
222         except:
223             pass
224
225         try:
226             temp = board[x - i][y + i]
227             possmoves.append([x - i, y + i])
228         except:
229             pass
230
231         try:
232             temp = board[x + i][y - i]
233             possmoves.append([x + i, y - i])
234         except:
235             pass
236
237         try:
238             temp = board[x - i][y - i]
239             possmoves.append([x - i, y - i])
240         except:
241             pass
242
243     # Filter all negative values
244     temp = [x for x in possmoves if x[0] >= 0 and x[1] >= 0]
245     allPossibleMoves = ["".join([chess_map_from_index_to_alpha[x[1]], str(x[0] + 1)]) for x in temp]
246     allPossibleMoves.sort()
247     return allPossibleMoves
248
249 # QUEEN MOVES
250 def getQueenMoves(pos, board):
251     # adding the bishop moves and rook moves to get queen moves
252     bishop_subset = getBishopMoves(pos, board)
253     rook_subset = getRookMoves(pos, board)
254     Queen_Moves = bishop_subset + rook_subset
255     return Queen_Moves
256
257 # King Moves
258 def getKingMoves(pos, board):
259     # A function(positionString, board) that returns the all possible moves of a king stood on a given position
260     column, row = list(pos.strip().lower())
261     row = int(row) - 1
262     column = chess_map_from_alpha_to_index[column]
263     x, y = row, column
264     possmoves = []
265     # does all the possible moves around the king besides the square the king is on
266     for i in range(-1, 2):
267         for j in range(-1, 2):
268             if (i != 0) or (j != 0):
269                 try:
270                     temp = board[x + i][y + j]
271                     possmoves.append([x + i, y + j])
272                 except:
273                     pass
274
275     # Filter all negative values
276     temp = [x for x in possmoves if x[0] >= 0 and x[1] >= 0]
277     allPossibleMoves = ["".join([chess_map_from_index_to_alpha[x[1]], str(x[0] + 1)]) for x in temp]
278     allPossibleMoves.sort()
279     return allPossibleMoves
280
281 # The chess board
282 chessboard = pygame.image.load("Chess Graphics/chess graphics/chess board/chessboard.jpg").convert()
283 screen.blit(chessboard, (10, 10))
284 # centers the pieces in the square
285 x_offset = 27
286 y_offset = 27
287
288 # Draws the board
289 def draw_board(board, color_to_move, moves, check, checkmate):
290     # fills background with grey color
291     screen.fill((75, 75, 75))
292     # draws the chessboard on the screen
293     screen.blit(chessboard, (10, 10))
294     # draws the pieces in their squares
295     for y in range(len(board)):
296         for x in range(len(board[0])):
297             if board[y][x] != None:
298                 screen.blit(piece_dict[board[y][x]], (x_offset + scale*x, y_offset + scale*y))
299     letters = ["a", "b", "c", "d", "e", "f", "g", "h"]
300     numbers = ["1", "2", "3", "4", "5", "6", "7", "8"]
301     numbers.reverse()
302
303     # Draw letters on the bottom of the screen
304     border_letters = [border_font.render(letter, True, (0, 0, 0)) for letter in letters]
305     [screen.blit(border_letters[i], (50+scale*i, 763)) for i in range(len(letters))]
306
307     # Draws numbers on the right of the screen
308     border_numbers = [border_font.render(number, True, (0, 0, 0)) for number in numbers]
309     [screen.blit(border_numbers[i], (763, 45+scale*i)) for i in range(len(numbers))]
310
311     # Draws the title "Chess!" on the right side of the screen
312     title = title_font.render("CHESS!", True, (0, 0, 0))
313     screen.blit(title, (800, 30))
314
315     # Draws the current player text
316     current_player_turn = border_font.render("Current Player:", True, (0, 0, 0))
317     screen.blit(current_player_turn, (815, 120))
318
319     if color_to_move == "w":
320         player = border_font.render("white", True, (255, 255, 255))
321     else:

```

```

322     player = border_font.render("black", True, (0, 0, 0))
323     screen.blit(player, (890, 160))
324
325     # Draws the moves counter
326     moves = border_font.render("Moves: "+str(moves), True, (0, 0, 0))
327     screen.blit(moves, (860, 220))
328
329     # Draws Check on the right side of the screen
330     if check == True:
331         check_text = border_font.render("Check!", True, (255, 0, 0))
332         screen.blit(check_text, (880, 270))
333
334     # Draws Checkmate in the middle of the screen along with the color of who won
335     if checkmate == True:
336         screen.fill((75, 75, 75), (155, 340, 465, 125))
337         checkmate_text = title_font.render("Checkmate!", True, (255, 0, 0))
338         screen.blit(checkmate_text, (165, 350))
339         if color_to_move == "w":
340             winner_text = border_font.render("Black Wins!", True, (0, 0, 0))
341             screen.blit(winner_text, (200, 420))
342         if color_to_move == "b":
343             winner_text = border_font.render("White Wins!", True, (0, 0, 0))
344             screen.blit(winner_text, (200, 420))
345
346     divider = title_font.render("_____", True, (255, 255, 255))
347     screen.blit(divider, (800, 190))
348
349     # Draws which pieces have been captured
350     capture_title_text = border_font.render("Captures", True, (0, 0, 0))
351     screen.blit(capture_title_text, (865, 310))
352
353     black_capture_text = border_font.render("Black", True, (0, 0, 0))
354     screen.blit(black_capture_text, (810, 350))
355
356     white_capture_text = border_font.render("White", True, (255, 255, 255))
357     screen.blit(white_capture_text, (950, 350))
358
359     black_captures = [captures_font.render(str(capture), True, (30, 30, 30)) for capture in b_capture_list]
360     [screen.blit(black_captures[i], (820, 390+30*i)) for i in range(len(b_capture_list))]
361
362     white_captures = [captures_font.render(str(capture), True, (30, 30, 30)) for capture in w_capture_list]
363     [screen.blit(white_captures[i], (960, 390+30*i)) for i in range(len(w_capture_list))]
364
365     # Checks to see if the board is the same as the old state
366     def checkBoard(old_state, current_state):
367         checkBoard = True
368         for y in range(len(current_state)):
369             for x in range(len(current_state[0])):
370                 if old_state[y][x] != current_state[y][x]:
371                     checkBoard = False
372         return checkBoard
373
374     # Finds all the possible moves for a given piece
375     def possibleMoves(board, x, y, color_to_move):
376         piece_y = y
377         piece_x = x
378         pos_moves = []
379         if board[piece_y][piece_x] != None:
380             if str(board[piece_y][piece_x])[0] != color_to_move:
381                 return pos_moves
382             """
383             dest_y = playerinputclicks[1][0]
384             dest_x = playerinputclicks[1][1]
385             """
386
387             # Changes the board coordinates to chess notation
388             board_piece_y = chess_map_from_true_y_to_board_y[piece_y]
389             alpha_piece_x = chess_map_from_index_to_alpha[piece_x]
390             """
391             board_dest_y = chess_map_from_true_y_to_board_y[dest_y]
392             alpha_dest_x = chess_map_from_index_to_alpha[dest_x]
393             """
394             pos_moves = []
395
396             # Finds the possible moves for a pawn
397             if str(board[piece_y][piece_x])[1] == "P":
398                 if str(board[piece_y][piece_x])[0] == "w":
399                     # Checks for diagonal capturing
400                     if piece_y != 0:
401                         if board[piece_y-1][piece_x] == None:
402                             pos_moves.append(alpha_piece_x+str((int(board_piece_y)+1)))
403                     if piece_x != 7:
404                         # 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
405                         if str(board[piece_y-1][piece_x+1])[0] == "b":
406                             pos_moves.append(chess_map_from_index_to_alpha[piece_x+1]+chess_map_from_true_y_to_board_y[piece_y-1])
407                     if piece_x != 0:
408                         if str(board[piece_y-1][piece_x-1])[0] == "b":
409                             pos_moves.append(chess_map_from_index_to_alpha[piece_x-1]+chess_map_from_true_y_to_board_y[piece_y-1])
410                     if piece_y == 6:
411                         pos_moves.append(alpha_piece_x+"4")
412                         if board[piece_y-2][piece_x] != None:
413                             pos_moves.remove(alpha_piece_x+"4")
414                     # Does the promotion of a pawn to a Queen
415                     if piece_y == 0:
416                         board[piece_y][piece_x] = "wQ"
417
418             # Black and white are the same; only difference is the y coordinates and directions for capturing
419             if str(board[piece_y][piece_x])[0] == "b":
420                 if piece_y != 7:
421                     if board[piece_y+1][piece_x] == None:
422                         pos_moves.append(alpha_piece_x+str((int(board_piece_y)-1)))
423                     if piece_x != 7:
424                         if str(board[piece_y+1][piece_x+1])[0] == "w":
425                             pos_moves.append(chess_map_from_index_to_alpha[piece_x+1]+chess_map_from_true_y_to_board_y[piece_y+1])
426                     if piece_x != 0:
427                         if str(board[piece_y+1][piece_x-1])[0] == "w":
428                             pos_moves.append(chess_map_from_index_to_alpha[piece_x-1]+chess_map_from_true_y_to_board_y[piece_y+1])

```

```

429         if piece_y == 1:
430             pos_moves.append(alpha_piece_x+"5")
431             if board[piece_y+2][piece_x] != None:
432                 pos_moves.remove(alpha_piece_x+"5")
433         if piece_y == 7:
434             board[piece_y][piece_x] = "bQ"
435
436 # If the piece is a rook; the different lists organizes the moves to be radially outward from the rook
437 if str(board[piece_y][piece_x])[1] == "R":
438     pos_moves = getRookMoves(alpha_piece_x+board_piece_y, board)
439     upper_y_moves = []
440     lower_y_moves = []
441     left_x_moves = []
442     right_x_moves = []
443     adj_pos_moves = []
444
445     for move in pos_moves:
446         if str(move)[0] == alpha_piece_x and int(str(move)[1]) > int(board_piece_y):
447             upper_y_moves.append(move)
448         if str(move)[0] == alpha_piece_x and int(str(move)[1]) < int(board_piece_y):
449             lower_y_moves.append(move)
450         if str(move)[1] == board_piece_y and chess_map_from_alpha_to_index[str(move)[0]] > piece_x:
451             right_x_moves.append(move)
452         if str(move)[1] == board_piece_y and chess_map_from_alpha_to_index[str(move)[0]] < piece_x:
453             left_x_moves.append(move)
454
455     lower_y_moves.reverse()
456     left_x_moves.reverse()
457
458 # It shrinks each list so that it can't go through pieces
459 for move in upper_y_moves:
460     x_val = chess_map_from_alpha_to_index[move[0]]
461     board_y = move[1]
462     y_val = chess_map_from_board_y_to_true_y[int(board_y)]
463
464     if board[int(y_val)][int(x_val)] == None:
465         adj_pos_moves.append(move)
466     else:
467         adj_pos_moves.append(move)
468         break
469
470 for move in lower_y_moves:
471     x_val = chess_map_from_alpha_to_index[move[0]]
472     board_y = move[1]
473     y_val = chess_map_from_board_y_to_true_y[int(board_y)]
474
475     if board[int(y_val)][int(x_val)] == None:
476         adj_pos_moves.append(move)
477     else:
478         adj_pos_moves.append(move)
479         break
480
481 for move in right_x_moves:
482     x_val = chess_map_from_alpha_to_index[move[0]]
483     board_y = move[1]
484     y_val = chess_map_from_board_y_to_true_y[int(board_y)]
485
486     if board[int(y_val)][int(x_val)] == None:
487         adj_pos_moves.append(move)
488     else:
489         adj_pos_moves.append(move)
490         break
491
492 for move in left_x_moves:
493     x_val = chess_map_from_alpha_to_index[move[0]]
494     board_y = move[1]
495     y_val = chess_map_from_board_y_to_true_y[int(board_y)]
496
497     if board[int(y_val)][int(x_val)] == None:
498         adj_pos_moves.append(move)
499     else:
500         adj_pos_moves.append(move)
501         break
502
503 pos_moves = adj_pos_moves
504
505 # If the piece is a queen; the different lists organizes the moves to be radially outward from the queen
506 if str(board[piece_y][piece_x])[1] == "Q":
507     pos_moves = getQueenMoves(alpha_piece_x+board_piece_y, board)
508
509     upper_y_moves = []
510     lower_y_moves = []
511     left_x_moves = []
512     right_x_moves = []
513     adj_pos_moves = []
514
515     for move in pos_moves:
516         if str(move)[0] == alpha_piece_x and int(str(move)[1]) > int(board_piece_y):
517             upper_y_moves.append(move)
518         if str(move)[0] == alpha_piece_x and int(str(move)[1]) < int(board_piece_y):
519             lower_y_moves.append(move)
520         if str(move)[1] == board_piece_y and chess_map_from_alpha_to_index[str(move)[0]] > piece_x:
521             right_x_moves.append(move)
522         if str(move)[1] == board_piece_y and chess_map_from_alpha_to_index[str(move)[0]] < piece_x:
523             left_x_moves.append(move)
524
525     lower_y_moves.reverse()
526     left_x_moves.reverse()
527
528 # It shrinks each list so that it can't go through pieces
529 for move in upper_y_moves:
530     x_val = chess_map_from_alpha_to_index[move[0]]
531     board_y = move[1]
532     y_val = chess_map_from_board_y_to_true_y[int(board_y)]
533
534     if board[int(y_val)][int(x_val)] == None:
535         adj_pos_moves.append(move)

```

```

536         else:
537             adj_pos_moves.append(move)
538             break
539
540     for move in lower_y_moves:
541         x_val = chess_map_from_alpha_to_index[move[0]]
542         board_y = move[1]
543         y_val = chess_map_from_board_y_to_true_y[int(board_y)]
544
545         if board[int(y_val)][int(x_val)] == None:
546             adj_pos_moves.append(move)
547         else:
548             adj_pos_moves.append(move)
549             break
550
551     for move in right_x_moves:
552         x_val = chess_map_from_alpha_to_index[move[0]]
553         board_y = move[1]
554         y_val = chess_map_from_board_y_to_true_y[int(board_y)]
555
556         if board[int(y_val)][int(x_val)] == None:
557             adj_pos_moves.append(move)
558         else:
559             adj_pos_moves.append(move)
560             break
561
562     for move in left_x_moves:
563         x_val = chess_map_from_alpha_to_index[move[0]]
564         board_y = move[1]
565         y_val = chess_map_from_board_y_to_true_y[int(board_y)]
566
567         if board[int(y_val)][int(x_val)] == None:
568             adj_pos_moves.append(move)
569         else:
570             adj_pos_moves.append(move)
571             break
572
573     # These lists are for each diagonal direction; organizing radially outward from the queen
574     upper_right_moves = []
575     upper_left_moves = []
576     lower_right_moves = []
577     lower_left_moves = []
578
579     for move in pos_moves:
580         if str(move)[0] > alpha_piece_x and int(str(move)[1]) > int(board_piece_y):
581             upper_right_moves.append(move)
582         if str(move)[0] > alpha_piece_x and int(str(move)[1]) < int(board_piece_y):
583             lower_right_moves.append(move)
584         if str(move)[0] < alpha_piece_x and int(str(move)[1]) > int(board_piece_y):
585             upper_left_moves.append(move)
586         if str(move)[0] < alpha_piece_x and int(str(move)[1]) < int(board_piece_y):
587             lower_left_moves.append(move)
588
589     upper_left_moves.reverse()
590     lower_left_moves.reverse()
591
592     # It shrinks each list so that it can't go through pieces
593     for move in upper_right_moves:
594         x_val = chess_map_from_alpha_to_index[move[0]]
595         board_y = move[1]
596         y_val = chess_map_from_board_y_to_true_y[int(board_y)]
597
598         if board[int(y_val)][int(x_val)] == None:
599             adj_pos_moves.append(move)
600         else:
601             adj_pos_moves.append(move)
602             break
603
604     for move in upper_left_moves:
605         x_val = chess_map_from_alpha_to_index[move[0]]
606         board_y = move[1]
607         y_val = chess_map_from_board_y_to_true_y[int(board_y)]
608
609         if board[int(y_val)][int(x_val)] == None:
610             adj_pos_moves.append(move)
611         else:
612             adj_pos_moves.append(move)
613             break
614
615     for move in lower_right_moves:
616         x_val = chess_map_from_alpha_to_index[move[0]]
617         board_y = move[1]
618         y_val = chess_map_from_board_y_to_true_y[int(board_y)]
619
620         if board[int(y_val)][int(x_val)] == None:
621             adj_pos_moves.append(move)
622         else:
623             adj_pos_moves.append(move)
624             break
625
626     for move in lower_left_moves:
627         x_val = chess_map_from_alpha_to_index[move[0]]
628         board_y = move[1]
629         y_val = chess_map_from_board_y_to_true_y[int(board_y)]
630
631         if board[int(y_val)][int(x_val)] == None:
632             adj_pos_moves.append(move)
633         else:
634             adj_pos_moves.append(move)
635             break
636
637     pos_moves = adj_pos_moves
638
639     # Same as the other pieces but for bishop
640     if str(board[piece_y][piece_x])[1] == "B":
641         pos_moves = getBishopMoves(alpha_piece_x+board_piece_y, board)
642         upper_right_moves = []
643         upper_left_moves = []

```

```

643     upper_left_moves = []
644     lower_right_moves = []
645     lower_left_moves = []
646     adj_pos_moves = []
647
648     for move in pos_moves:
649         if str(move)[0] > alpha_piece_x and int(str(move)[1]) > int(board_piece_y):
650             upper_right_moves.append(move)
651         if str(move)[0] > alpha_piece_x and int(str(move)[1]) < int(board_piece_y):
652             lower_right_moves.append(move)
653         if str(move)[0] < alpha_piece_x and int(str(move)[1]) > int(board_piece_y):
654             upper_left_moves.append(move)
655         if str(move)[0] < alpha_piece_x and int(str(move)[1]) < int(board_piece_y):
656             lower_left_moves.append(move)
657
658     upper_left_moves.reverse()
659     lower_left_moves.reverse()
660
661     for move in upper_right_moves:
662         x_val = chess_map_from_alpha_to_index[move[0]]
663         board_y = move[1]
664         y_val = chess_map_from_board_y_to_true_y[int(board_y)]
665
666         if board[int(y_val)][int(x_val)] == None:
667             adj_pos_moves.append(move)
668         else:
669             adj_pos_moves.append(move)
670             break
671
672     for move in upper_left_moves:
673         x_val = chess_map_from_alpha_to_index[move[0]]
674         board_y = move[1]
675         y_val = chess_map_from_board_y_to_true_y[int(board_y)]
676
677         if board[int(y_val)][int(x_val)] == None:
678             adj_pos_moves.append(move)
679         else:
680             adj_pos_moves.append(move)
681             break
682
683     for move in lower_right_moves:
684         x_val = chess_map_from_alpha_to_index[move[0]]
685         board_y = move[1]
686         y_val = chess_map_from_board_y_to_true_y[int(board_y)]
687
688         if board[int(y_val)][int(x_val)] == None:
689             adj_pos_moves.append(move)
690         else:
691             adj_pos_moves.append(move)
692             break
693
694     for move in lower_left_moves:
695         x_val = chess_map_from_alpha_to_index[move[0]]
696         board_y = move[1]
697         y_val = chess_map_from_board_y_to_true_y[int(board_y)]
698
699         if board[int(y_val)][int(x_val)] == None:
700             adj_pos_moves.append(move)
701         else:
702             adj_pos_moves.append(move)
703             break
704
705     pos_moves = adj_pos_moves
706
707     # If the piece is a king or a knight
708     if str(board[piece_y][piece_x])[1] == "K":
709         # If statement determines if it is a knight, otherwise it is a king
710         if len(str(board[piece_y][piece_x])) == 3:
711             pos_moves = getKnightMoves(alpha_piece_x+board_piece_y, board)
712         else:
713             pos_moves = getKingMoves(alpha_piece_x+board_piece_y, board)
714     return pos_moves
715
716 # Checks to see if the King of the enemy team is in Check
717 def checkCheck(board, color_to_move):
718     pos_moves = []
719
720     if color_to_move == "w":
721         king_color = "b"
722     if color_to_move == "b":
723         king_color = "w"
724     # king_cords = [0, 4]
725     # Finding all the possible moves for one team
726     for y in range(len(board)):
727         for x in range(len(board[0])):
728             pos_moves.append(possibleMoves(board, x, y, color_to_move))
729     # Finds the coordinates of the enemy King
730     for y in range(len(board)):
731         try: king_cords = [y, board[y].index(king_color+"K")]
732         except: pass
733
734     # Alpha King Cords converts the enemy King coordinates into chess notation
735     alpha_king_cords = str(chess_map_from_index_to_alpha[king_cords[1]]+chess_map_from_true_y_to_board_y[king_cords[0]])
736     # Checks to see if one of the possible moves is to capture the enemy King
737
738     check = False
739     for i in pos_moves:
740         if alpha_king_cords in i:
741             check = True
742     return check
743
744 # Checks to see if there is a checkmate
745 def checkCheckmate(board, color_to_move):
746
747     #checkCheck(board, color_to_move)
748
749     # Creates a fake board to manipulate without affecting the real board
750     test_board = [[board[y][x] for x in range(len(board[0]))] for y in range(len(board))]

```

```

751     checkmate = False
752     pos_moves = []
753     uncheckables = []
754
755     if color_to_move == "w":
756         for y in range(len(board)):
757             for x in range(len(board[0])):
758                 # Finds the possible moves for every white piece on the board
759                 pos_moves = []
760                 if str(board[y][x])[0] == "w":
761                     pos_moves.extend(possibleMoves(board, x, y, color_to_move))
762                     # Tests to see if each of the moves causes resolve to check or not
763                     for move in pos_moves:
764                         test_board = [[board[y][x] for x in range(len(board[0]))] for y in range(len(board))]
765                         dest_x = int(chess_map_from_alpha_to_index[str(move)[0]])
766                         dest_y = int(chess_map_from_board_y_to_true_y[int(str(move)[1])])
767                         if board[y][x] != None:
768                             if str(board[dest_y][dest_x])[0] != str(board[y][x])[0]:
769                                 test_board[dest_y][dest_x] = test_board[y][x]
770                                 test_board[y][x] = None
771                                 if checkCheck(test_board, "b") == False:
772                                     uncheckables.append(move)
773
774     # Same things as above but for black pieces
775     if color_to_move == "b":
776         for y in range(len(board)):
777             for x in range(len(board[0])):
778                 pos_moves = []
779                 if str(board[y][x])[0] == "b":
780                     pos_moves.extend(possibleMoves(board, x, y, color_to_move))
781                     for i in pos_moves:
782                         test_board = [[board[y][x] for x in range(len(board[0]))] for y in range(len(board))]
783                         dest_x = int(chess_map_from_alpha_to_index[str(i)[0]])
784                         dest_y = int(chess_map_from_board_y_to_true_y[int(str(i)[1])])
785                         if board[y][x] != None:
786                             if str(board[dest_y][dest_x])[0] != str(board[y][x])[0]:
787                                 test_board[dest_y][dest_x] = test_board[y][x]
788                                 test_board[y][x] = None
789                                 if checkCheck(test_board, "w") == False:
790                                     uncheckables.append(i)
791
792     # If no safe moves are found, checkmate
793     if len(uncheckables) == 0:
794         checkmate = True
795         return checkmate
796
797 # Finds the possible moves but accounts for checks and captures; prevents players from moving into check or capturing their own pieces
798 def makeMove(board, playerinputclicks, color_to_move):
799     piece_y = playerinputclicks[0][0]
800     piece_x = playerinputclicks[0][1]
801
802     if color_to_move == "w":
803         check_color = "b"
804     if color_to_move == "b":
805         check_color = "w"
806
807     check_board = [[board[y][x] for x in range(len(board[0]))] for y in range(len(board))]
808
809     if board[piece_y][piece_x] != None:
810         if str(board[piece_y][piece_x])[0] != color_to_move:
811             return board
812         dest_y = playerinputclicks[1][0]
813         dest_x = playerinputclicks[1][1]
814
815         board_piece_y = chess_map_from_true_y_to_board_y[piece_y]
816         alpha_piece_x = chess_map_from_index_to_alpha[piece_x]
817
818         board_dest_y = chess_map_from_true_y_to_board_y[dest_y]
819         alpha_dest_x = chess_map_from_index_to_alpha[dest_x]
820
821     # Finds every possible move for the selected piece; if destination square is one of the possible moves, it is allowed
822     pos_moves = possibleMoves(board, piece_x, piece_y, color_to_move)
823     for i in pos_moves:
824         if i == alpha_dest_x+board_dest_y:
825             if board[piece_y][piece_x] != None:
826                 # Lines 824 - 836 simulates a move in order to see if the player is trying to move into check
827                 # Prevents a piece from capturing a piece on the same team
828                 if str(board[dest_y][dest_x])[0] == str(board[piece_y][piece_x])[0]:
829                     pass
830                 # Checks to see if the piece is trying to capture an enemy piece
831                 elif str(board[dest_y][dest_x])[0] != str(board[piece_y][piece_x])[0]:
832                     check_board[dest_y][dest_x] = check_board[piece_y][piece_x]
833                     check_board[piece_y][piece_x] = None
834                 # Allows the piece to move onto a blank square
835                 else:
836                     check_board[dest_y][dest_x] = check_board[piece_y][piece_x]
837                     check_board[piece_y][piece_x] = None
838
839     # Checks to see if the player moved a piece into check
840     if checkCheck(check_board, check_color) == True:
841         return board
842
843     # Actually moving a piece; same conditions as above
844     if str(board[dest_y][dest_x])[0] == str(board[piece_y][piece_x])[0]:
845         pass
846     elif str(board[dest_y][dest_x])[0] != str(board[piece_y][piece_x])[0]:
847         # Adds captured pieces to a list for display
848         if str(board[dest_y][dest_x])[0] == "b":
849             b_capture_list.append(piece_letter_to_name[str(board[dest_y][dest_x])[1]])
850         if str(board[dest_y][dest_x])[0] == "w":
851             w_capture_list.append(piece_letter_to_name[str(board[dest_y][dest_x])[1]])
852         board[dest_y][dest_x] = board[piece_y][piece_x]
853         board[piece_y][piece_x] = None
854     else:
855         board[dest_y][dest_x] = board[piece_y][piece_x]
856         board[piece_y][piece_x] = None
857
858     else:
859         pass
860
861     # Checks to see if there is checkmate

```



```

858     checkMate = checkCheckmate(board, color_to_move)
859     return board
860
861 # Main game loop things https://levelup.gitconnected.com/chess-python-ca4532c7f5a4 and https://www.youtube.com/watch?v=o24J3WcBGLg
862 running = True
863 selectedsquare = ()
864 playerinputclicks = []
865
866 color_to_move = "w"
867
868 # Creates a copy of the board for comparison in the board check function
869 old_state = [[board[y][x] for x in range(len(board[0]))] for y in range(len(board))]
870 check = False
871
872 checkmate = False
873
874 moves = 0
875 while (running): #press end game then loop stops
876     # Creates a list of all inputs from user computer
877     for event in pygame.event.get():
878         # Closes the window if the player clicks the x in the top right
879         if event.type == pygame.QUIT:
880             running = False
881         # Checks if the player is clicking on a square
882         elif event.type == pygame.MOUSEBUTTONDOWN:
883             # Finds the location of the mouse on the screen
884             location = pygame.mouse.get_pos()
885             # Maps the location in a specific square
886             col = location[0] // 100 #sqsize = height // dimesion (8)
887             row = location[1] // 100
888
889             selectedsquare = (row, col)
890             playerinputclicks.append(selectedsquare)
891             if selectedsquare == (row, col):
892                 selectedsquare = ()
893                 #playerinputclicks = []
894                 # Checks to see if the player has also chosen a destination square
895                 if len(playerinputclicks) >= 2:
896                     board = makeMove(board, playerinputclicks, color_to_move)
897                     checkmate = checkCheckmate(board, color_to_move)
898                     draw_board(board, color_to_move, moves, check, checkmate)
899                     selectedsquare = ()
900                     playerinputclicks = []
901
902             # Alternates the players turn only if the board has changed (Allows for a player to pick up and put a piece back down)
903             if not checkBoard(old_state, board):
904                 check = checkCheck(board, color_to_move)
905                 if color_to_move == "w":
906                     color_to_move = "b"
907                 elif color_to_move == "b":
908                     color_to_move = "w"
909                 old_state = [[board[y][x] for x in range(len(board[0]))] for y in range(len(board))]
910                 moves += 1
911
912 # Draws the board
913 draw_board(board, color_to_move, moves, check, checkmate)
914 # Updates the screen
915 pygame.display.update()
916 # Sets FPS to 60
917 clock.tick(60)
918 pygame.quit()

```