ChessNull.

Presented to:
Dr Mayar Mostafa.
Eng Fady Maged.

Description of the project.

chess using genetic algorithm combines the strategy of chess with the computational powers of Evolutionary Computation.

In chess, GA can be employed to develop algorithms capable of decision-making in the chess board. The algorithm selects, mutates, and recombines to achieve improved gameplay. It's designed with two options of your choice:
(1. Ai mode) or (2. Two players mode)

Building a chess game using Ai Genetic Algorithms requires huge and detailed steps of implementation, described as follows.

Project is divided into 3 parts.

GUI

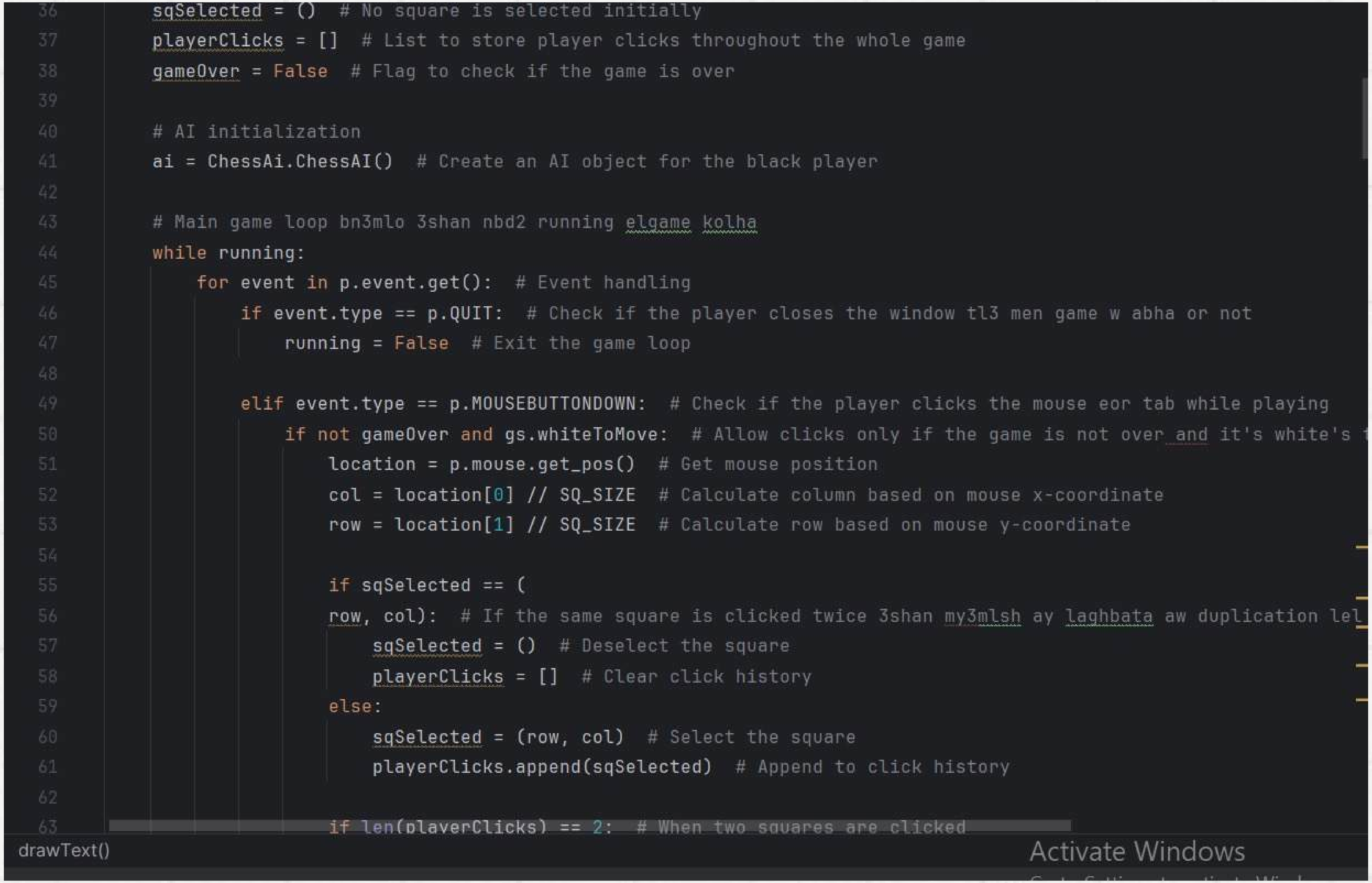
ChessAi(GA)

ChessEngine

GUI-Omar Nabil.

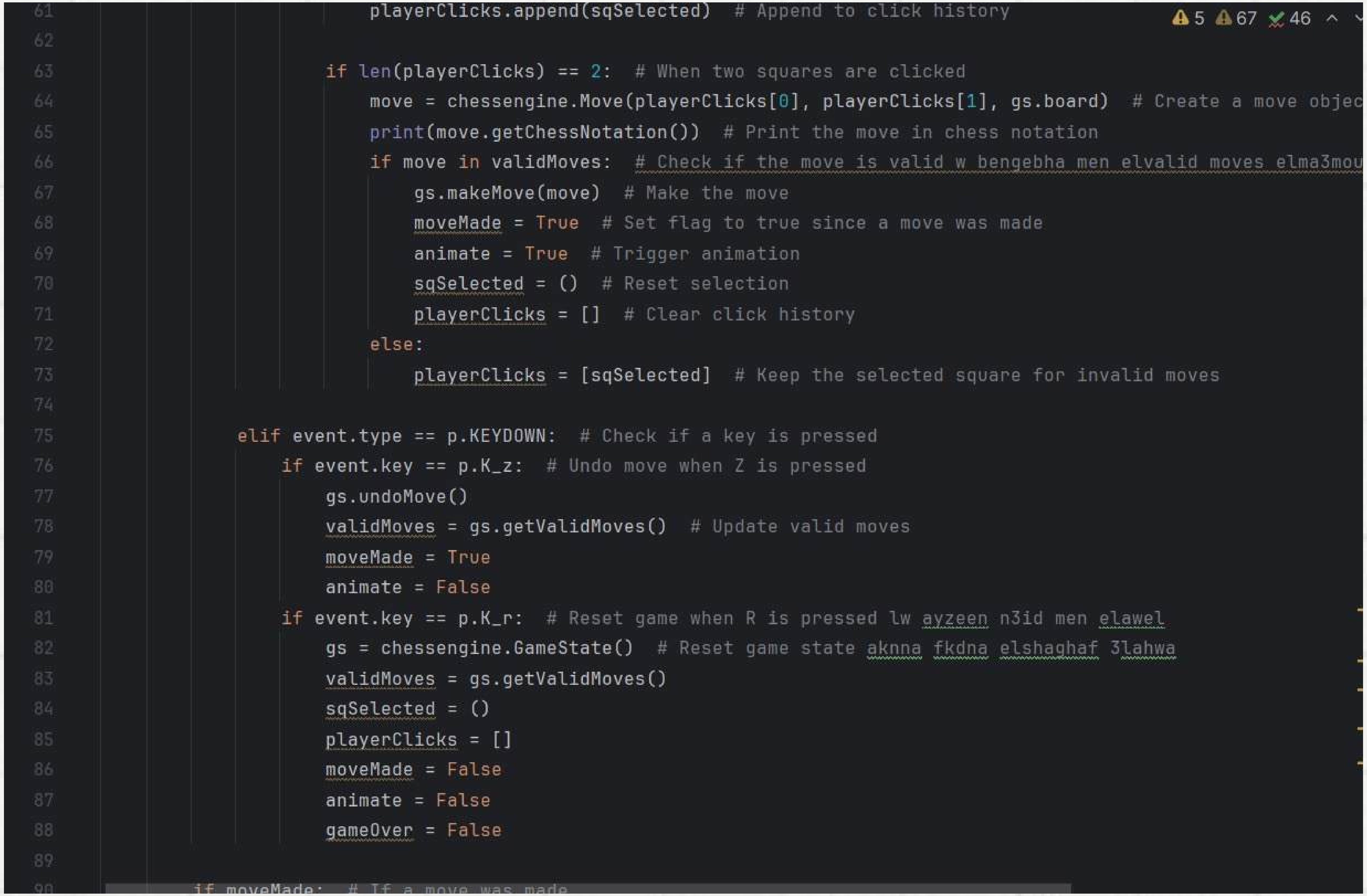
- The code is part of a chess game implementation that handles player moves and game state.
- It takes player input in the form of mouse clicks or keyboard presses.
- It validates player moves based on chess rules and updates the game state accordingly.
- It can handle undo moves and reset the game.
- It triggers animations to visualize the moves and game state changes.

GUI (Snippet of the code)



```
sqSelected = ()  # No square is selected initially
playerClicks = []  # List to store player clicks throughout the whole game
gameOver = False  # Flag to check if the game is over

# AI initialization
ai = ChessAi.ChessAI()  # Create an AI object for the black player

# Main game loop bn3mlo 3shan nbd2 running elgame kolha
while running:
    for event in p.event.get():  # Event handling
        if event.type == p.QUIT:  # Check if the player closes the window tl3 men game w abha or not
            running = False  # Exit the game loop

        elif event.type == p.MOUSEBUTTONDOWN:  # Check if the player clicks the mouse eor tab while playing
            if not gameOver and gs.whiteToMove:  # Allow clicks only if the game is not over and it's white's t
                location = p.mouse.get_pos()  # Get mouse position
                col = location[0] // SQ_SIZE  # Calculate column based on mouse x-coordinate
                row = location[1] // SQ_SIZE  # Calculate row based on mouse y-coordinate

                if sqSelected == (
                row, col):  # If the same square is clicked twice 3shan my3mlsh ay laghbata aw duplication lel
                    sqSelected = ()  # Deselect the square
                    playerClicks = []  # Clear click history
                else:
                    sqSelected = (row, col)  # Select the square
                    playerClicks.append(sqSelected)  # Append to click history

                if len(playerClicks) == 2:  # When two squares are clicked
```

GUI (Snippet of the code)

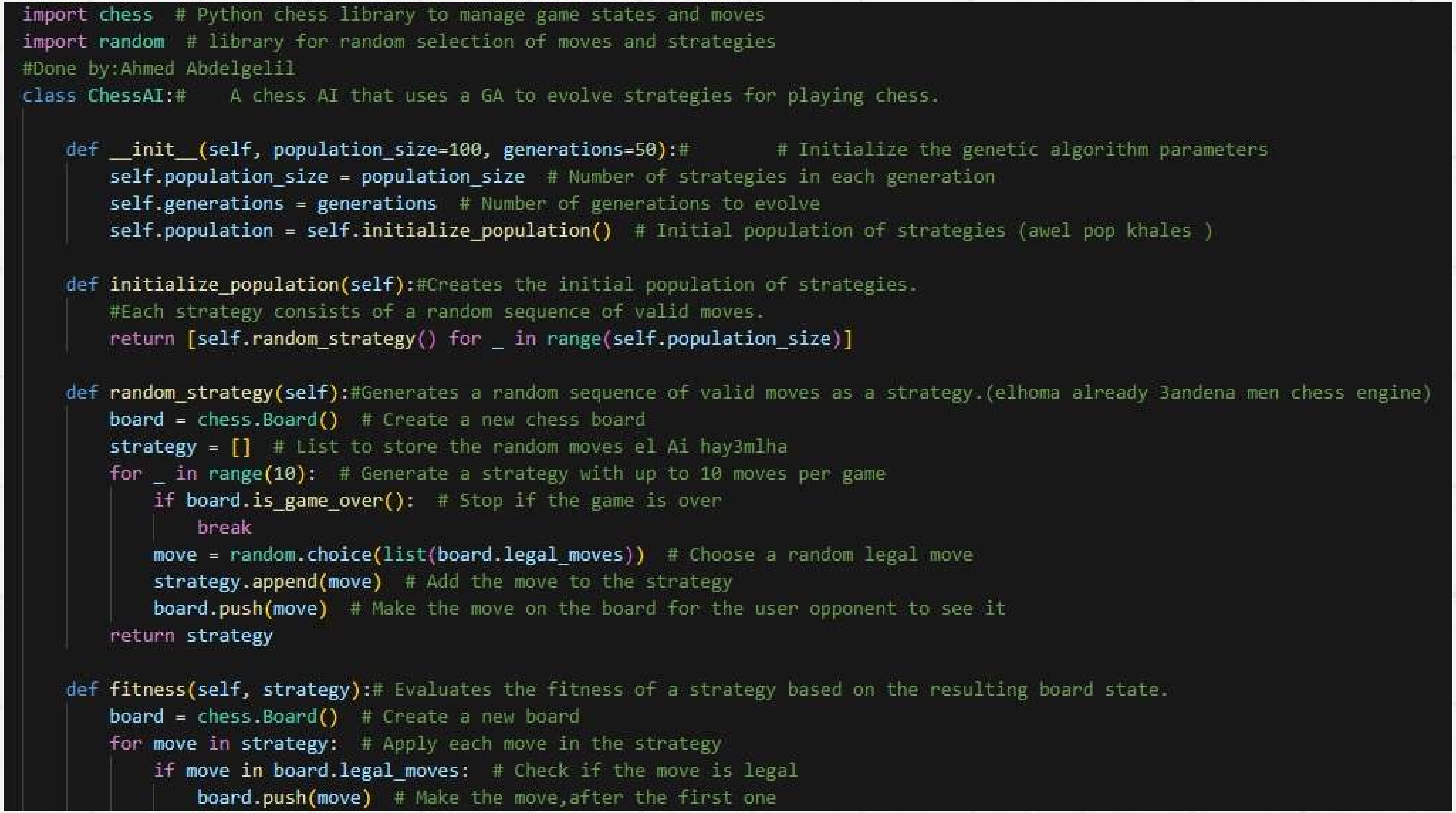


```
                            playerClicks.append(sqSelected)  # Append to click history

                        if len(playerClicks) == 2:  # When two squares are clicked
                            move = chessengine.Move(playerClicks[0], playerClicks[1], gs.board)  # Create a move objec
                            print(move.getChessNotation())  # Print the move in chess notation
                            if move in validMoves:  # Check if the move is valid w bengebha men elvalid moves elma3mou
                                gs.makeMove(move)  # Make the move
                                moveMade = True  # Set flag to true since a move was made
                                animate = True  # Trigger animation
                                sqSelected = ()  # Reset selection
                                playerClicks = []  # Clear click history
                            else:
                                playerClicks = [sqSelected]  # Keep the selected square for invalid moves

                elif event.type == p.KEYDOWN:  # Check if a key is pressed
                    if event.key == p.K_z:  # Undo move when Z is pressed
                        gs.undoMove()
                        validMoves = gs.getValidMoves()  # Update valid moves
                        moveMade = True
                        animate = False
                    if event.key == p.K_r:  # Reset game when R is pressed lw ayzeen n3id men elawel
                        gs = chessengine.GameState()  # Reset game state aknna fkdna elshaghaf 3lahwa
                        validMoves = gs.getValidMoves()
                        sqSelected = ()
                        playerClicks = []
                        moveMade = False
                        animate = False
                        gameOver = False

            if moveMade:  # If a move was made
```

ChessAi(GA)-Ahmed Abdelgelil.

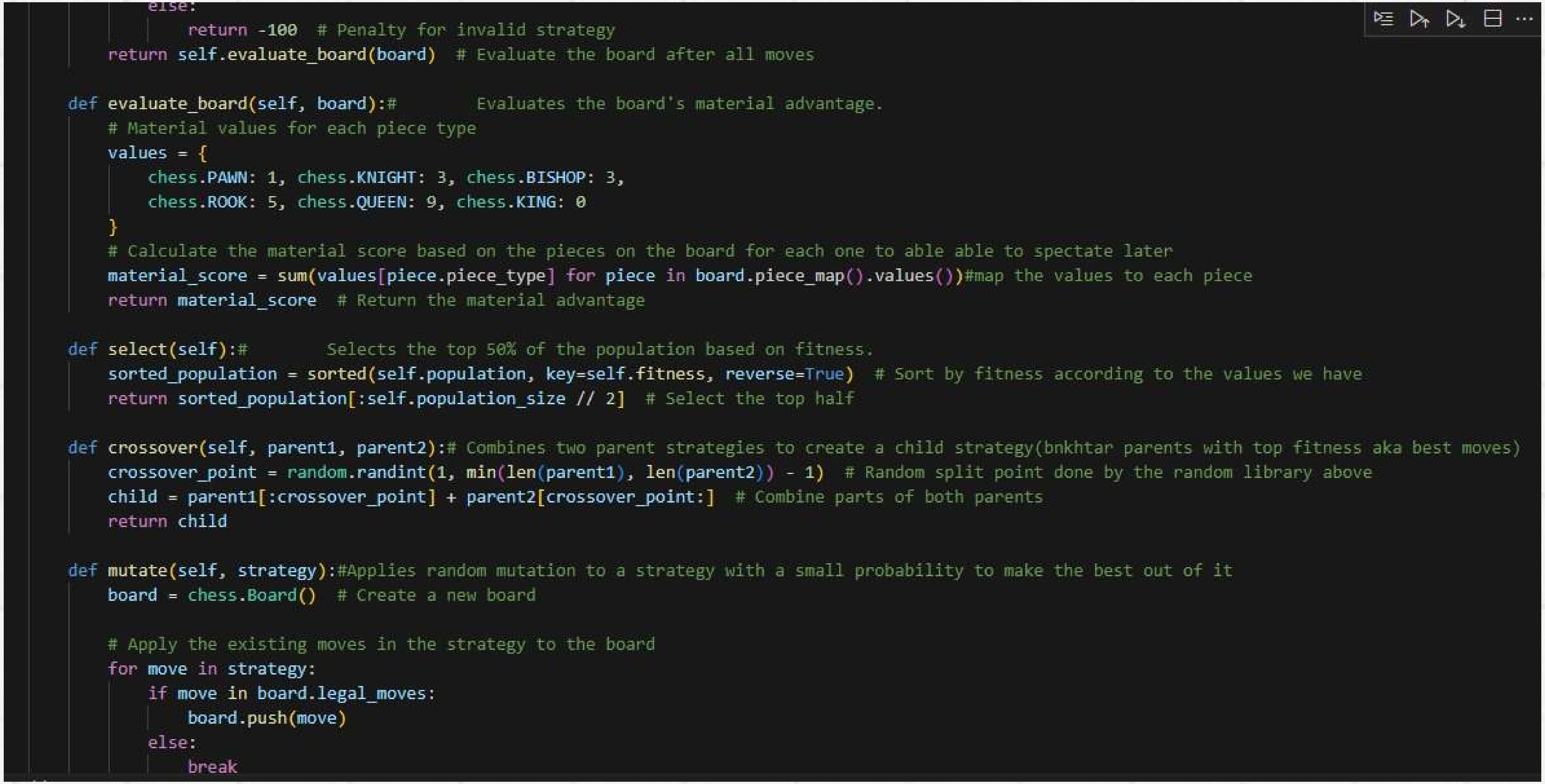
- This Python code implements a genetic algorithm to evolve strategies for playing chess.
- It initializes a population of random strategies and evaluates their fitness based on the board state.
- It uses selection, crossover, and mutation to create new generations of improved strategies.
- The code includes functions for evaluating board states, selecting top-performing strategies, and applying genetic operators.
- The ultimate goal is to evolve strategies that can play chess effectively at a pro level.

(GA) Snippets of the code.



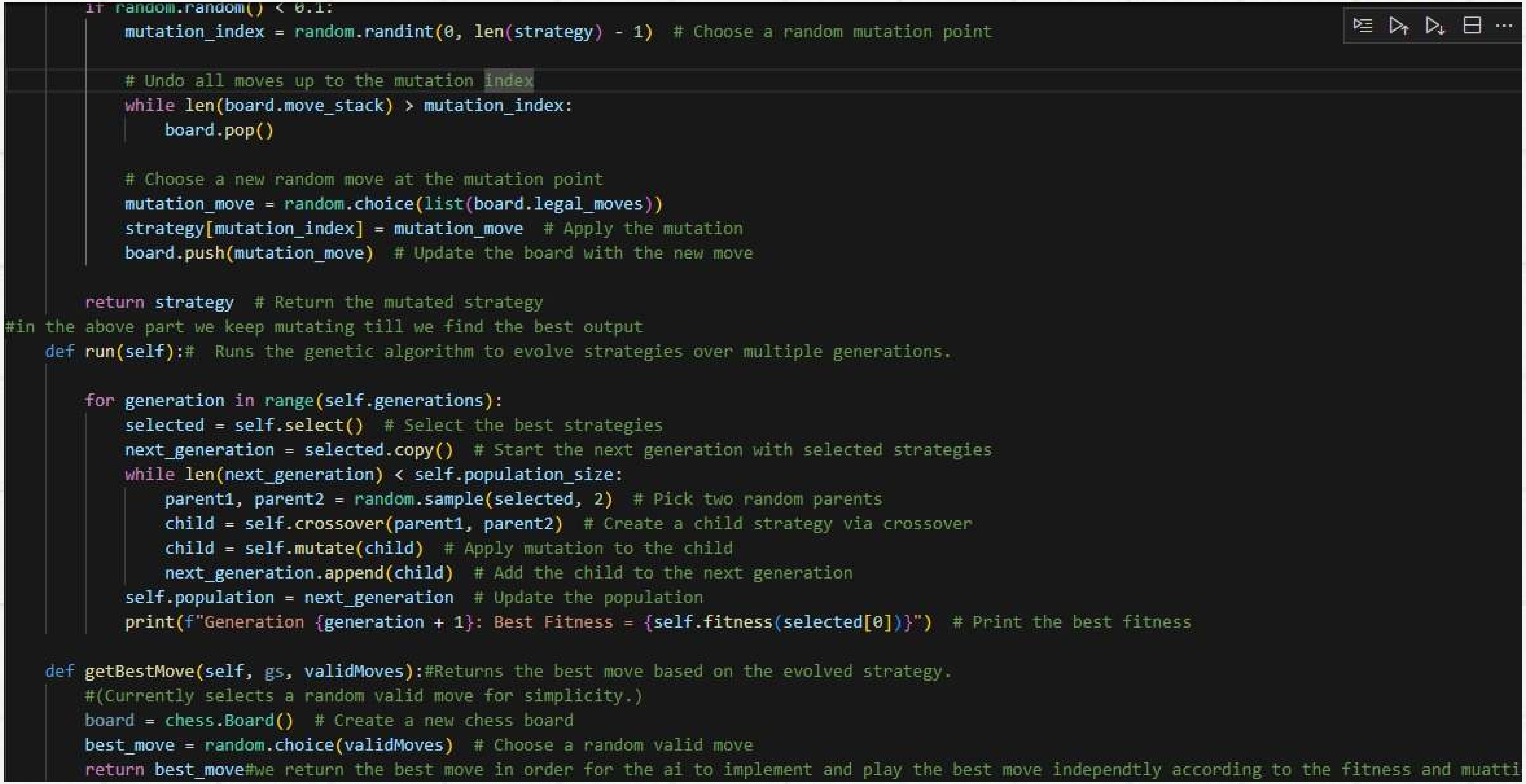
```python
import chess  # Python chess library to manage game states and moves
import random  # library for random selection of moves and strategies
#Done by:Ahmed Abdelgelil
class ChessAI:#    A chess AI that uses a GA to evolve strategies for playing chess.

    def __init__(self, population_size=100, generations=50):#        # Initialize the genetic algorithm parameters
        self.population_size = population_size  # Number of strategies in each generation
        self.generations = generations  # Number of generations to evolve
        self.population = self.initialize_population()  # Initial population of strategies (awel pop khales )

    def initialize_population(self):#Creates the initial population of strategies.
        #Each strategy consists of a random sequence of valid moves.
        return [self.random_strategy() for _ in range(self.population_size)]

    def random_strategy(self):#Generates a random sequence of valid moves as a strategy.(elhoma already 3andena men chess engine)
        board = chess.Board()  # Create a new chess board
        strategy = []  # List to store the random moves el Ai hay3mlha
        for _ in range(10):  # Generate a strategy with up to 10 moves per game
            if board.is_game_over():  # Stop if the game is over
                break
            move = random.choice(list(board.legal_moves))  # Choose a random legal move
            strategy.append(move)  # Add the move to the strategy
            board.push(move)  # Make the move on the board for the user opponent to see it
        return strategy

    def fitness(self, strategy):# Evaluates the fitness of a strategy based on the resulting board state.
        board = chess.Board()  # Create a new board
        for move in strategy:  # Apply each move in the strategy
            if move in board.legal_moves:  # Check if the move is legal
                board.push(move)  # Make the move,after the first one
```

(GA) Snippets of the code.



```
            else:
                return -100  # Penalty for invalid strategy
        return self.evaluate_board(board)  # Evaluate the board after all moves

    def evaluate_board(self, board):#        Evaluates the board's material advantage.
        # Material values for each piece type
        values = {
            chess.PAWN: 1, chess.KNIGHT: 3, chess.BISHOP: 3,
            chess.ROOK: 5, chess.QUEEN: 9, chess.KING: 0
        }
        # Calculate the material score based on the pieces on the board for each one to able able to spectate later
        material_score = sum(values[piece.piece_type] for piece in board.piece_map().values())#map the values to each piece
        return material_score  # Return the material advantage

    def select(self):#        Selects the top 50% of the population based on fitness.
        sorted_population = sorted(self.population, key=self.fitness, reverse=True)  # Sort by fitness according to the values we have
        return sorted_population[:self.population_size // 2]  # Select the top half

    def crossover(self, parent1, parent2):# Combines two parent strategies to create a child strategy(bnkhtar parents with top fitness aka best moves)
        crossover_point = random.randint(1, min(len(parent1), len(parent2)) - 1)  # Random split point done by the random library above
        child = parent1[:crossover_point] + parent2[crossover_point:]  # Combine parts of both parents
        return child

    def mutate(self, strategy):#Applies random mutation to a strategy with a small probability to make the best out of it
        board = chess.Board()  # Create a new board

        # Apply the existing moves in the strategy to the board
        for move in strategy:
            if move in board.legal_moves:
                board.push(move)
            else:
                break
```

(GA) Snippets of the code.



```
        if random.random() < 0.1:
            mutation_index = random.randint(0, len(strategy) - 1)  # Choose a random mutation point

            # Undo all moves up to the mutation index
            while len(board.move_stack) > mutation_index:
                board.pop()

            # Choose a new random move at the mutation point
            mutation_move = random.choice(list(board.legal_moves))
            strategy[mutation_index] = mutation_move  # Apply the mutation
            board.push(mutation_move)  # Update the board with the new move

        return strategy  # Return the mutated strategy
#in the above part we keep mutating till we find the best output
    def run(self):#  Runs the genetic algorithm to evolve strategies over multiple generations.

        for generation in range(self.generations):
            selected = self.select()  # Select the best strategies
            next_generation = selected.copy()  # Start the next generation with selected strategies
            while len(next_generation) < self.population_size:
                parent1, parent2 = random.sample(selected, 2)  # Pick two random parents
                child = self.crossover(parent1, parent2)  # Create a child strategy via crossover
                child = self.mutate(child)  # Apply mutation to the child
                next_generation.append(child)  # Add the child to the next generation
            self.population = next_generation  # Update the population
            print(f"Generation {generation + 1}: Best Fitness = {self.fitness(selected[0])}")  # Print the best fitness

    def getBestMove(self, gs, validMoves):#Returns the best move based on the evolved strategy.
        #(Currently selects a random valid move for simplicity.)
        board = chess.Board()  # Create a new chess board
        best_move = random.choice(validMoves)  # Choose a random valid move
        return best_move#we return the best move in order for the ai to implement and play the best move independtly according to the fitness and muatti
```

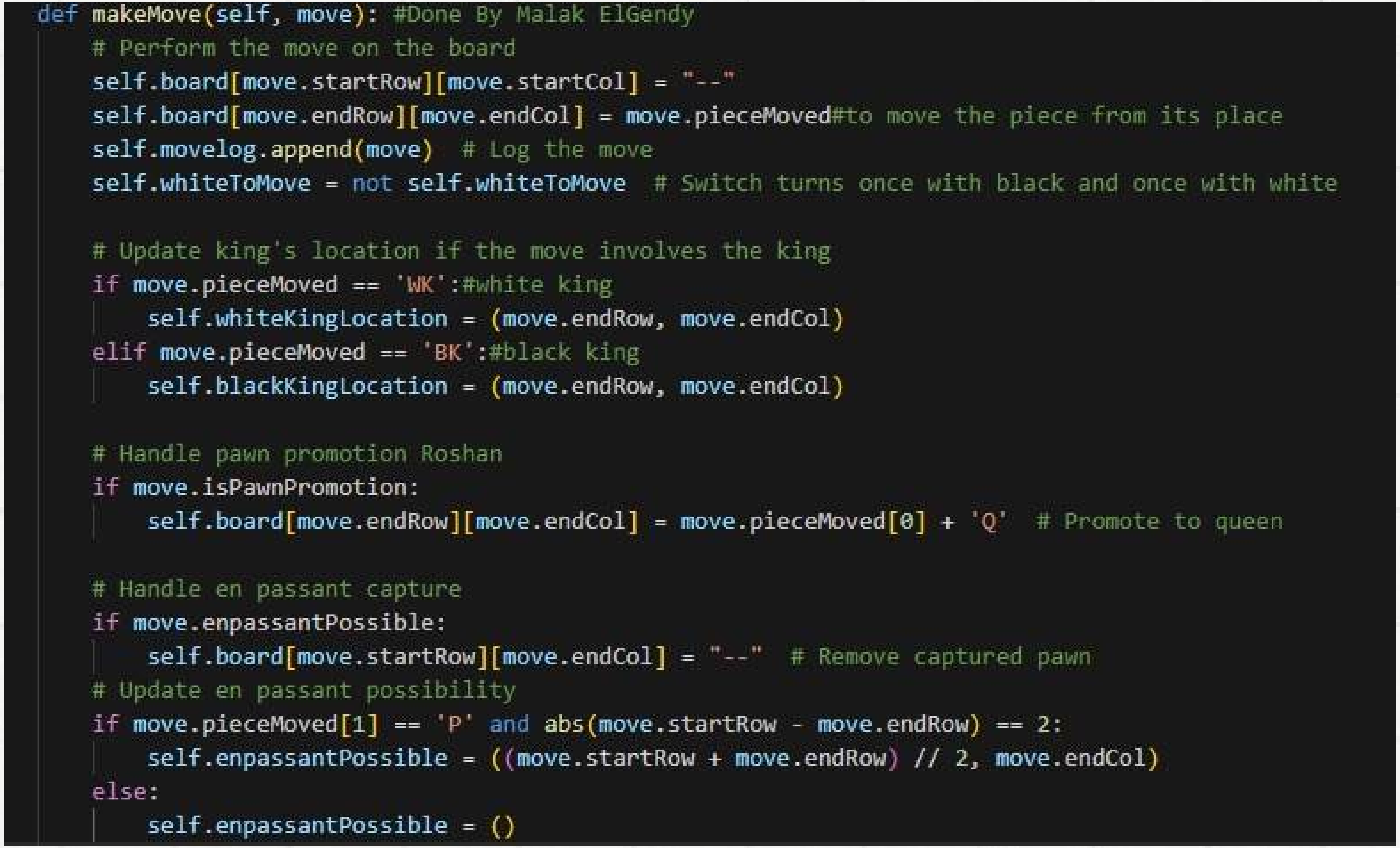
ChessEngine(MakeMove),(Square UnderAttack) ,(GetAllPossibleMoves)-MalakElGendy.

- The function updates the board by setting the starting square to an empty space and placing the moved piece on the destination square.
- It logs the move for future reference.
- It switches the turn between white and black.

- If the moved piece is a king, the function updates the king's location to the new position.

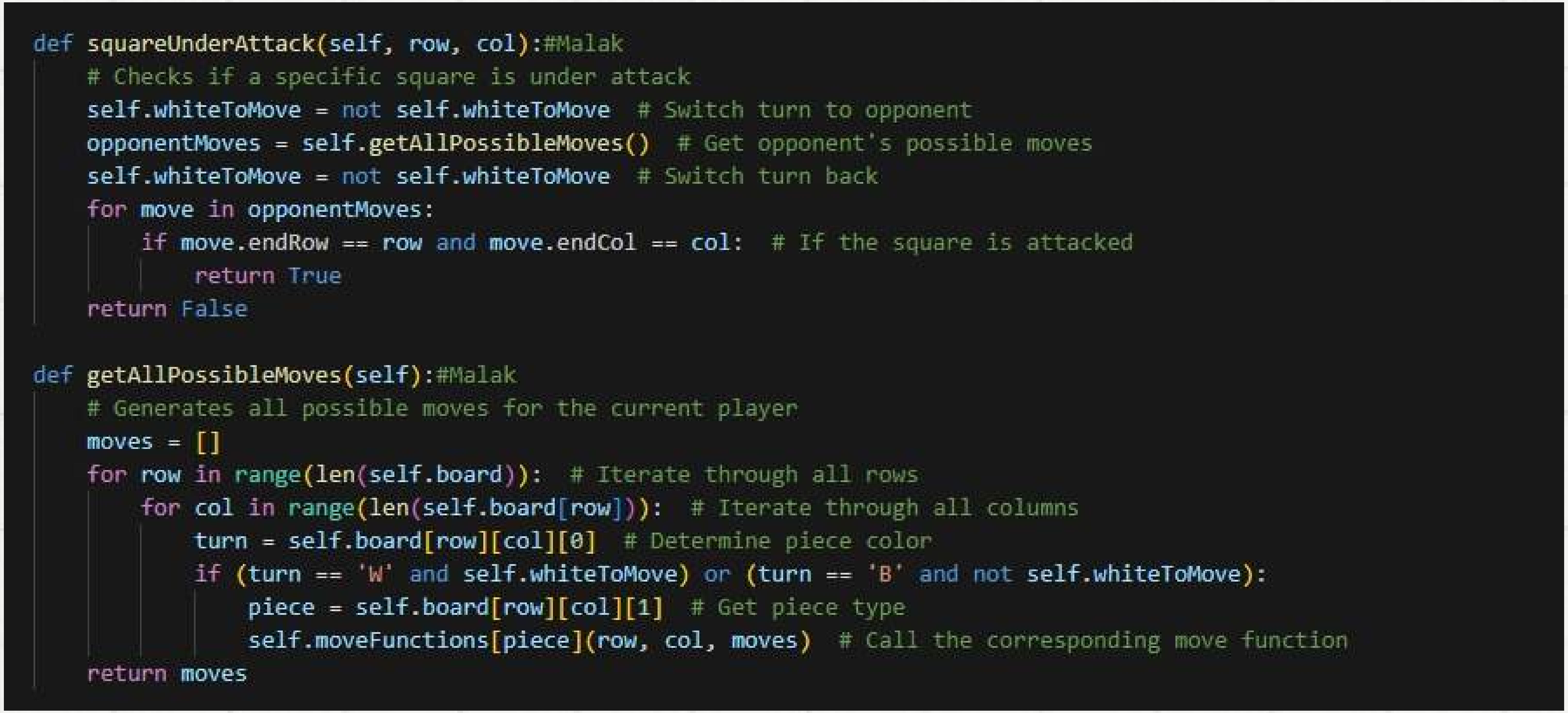
•If a pawn reaches the end of the board, it promotes to a queen by default.
•If an en passant capture is possible, the function removes the captured pawn from the board and updates the enpassantPossible flag.

ChessEngine(MakeMove) Snippets.



```
def makeMove(self, move): #Done By Malak ElGendy
    # Perform the move on the board
    self.board[move.startRow][move.startCol] = "--"
    self.board[move.endRow][move.endCol] = move.pieceMoved#to move the piece from its place
    self.movelog.append(move)  # Log the move
    self.whiteToMove = not self.whiteToMove  # Switch turns once with black and once with white

    # Update king's location if the move involves the king
    if move.pieceMoved == 'WK':#white king
        self.whiteKingLocation = (move.endRow, move.endCol)
    elif move.pieceMoved == 'BK':#black king
        self.blackKingLocation = (move.endRow, move.endCol)

    # Handle pawn promotion Roshan
    if move.isPawnPromotion:
        self.board[move.endRow][move.endCol] = move.pieceMoved[0] + 'Q'  # Promote to queen

    # Handle en passant capture
    if move.enpassantPossible:
        self.board[move.startRow][move.endCol] = "--"  # Remove captured pawn
    # Update en passant possibility
    if move.pieceMoved[1] == 'P' and abs(move.startRow - move.endRow) == 2:
        self.enpassantPossible = ((move.startRow + move.endRow) // 2, move.endCol)
    else:
        self.enpassantPossible = ()
```

ChessEngine(Square UnderAttack) ,(GetAllPossibleMoves) Snippets.

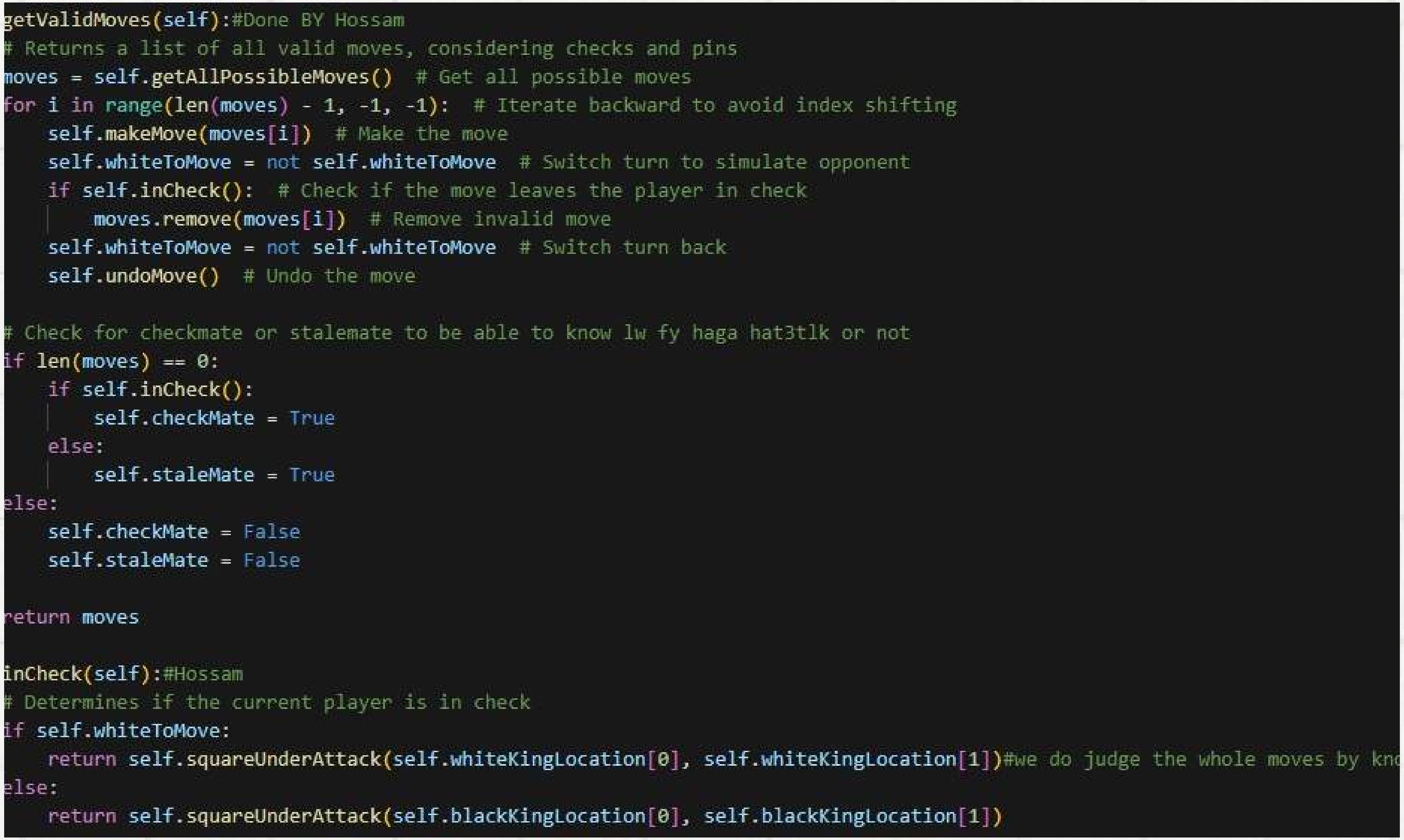


```
def squareUnderAttack(self, row, col):#Malak
    # Checks if a specific square is under attack
    self.whiteToMove = not self.whiteToMove  # Switch turn to opponent
    opponentMoves = self.getAllPossibleMoves()  # Get opponent's possible moves
    self.whiteToMove = not self.whiteToMove  # Switch turn back
    for move in opponentMoves:
        if move.endRow == row and move.endCol == col:  # If the square is attacked
            return True
    return False

def getAllPossibleMoves(self):#Malak
    # Generates all possible moves for the current player
    moves = []
    for row in range(len(self.board)):  # Iterate through all rows
        for col in range(len(self.board[row])):  # Iterate through all columns
            turn = self.board[row][col][0]  # Determine piece color
            if (turn == 'W' and self.whiteToMove) or (turn == 'B' and not self.whiteToMove):
                piece = self.board[row][col][1]  # Get piece type
                self.moveFunctions[piece](row, col, moves)  # Call the corresponding move function
    return moves
```

ChessEngine(Getvalidmoves)(checkmate)-Hossam Khairy.

- Generates all possible moves for the current player.
- Iterates through each move and simulates it on the board.
- Checks if the simulated move leaves the player in check.
- Removes invalid moves that would leave the player in check.
- Returns a list of valid moves, considering checkmate and stalemate conditions.

ChessEngine(GetValidMoves)(Checkmate)Snippets.

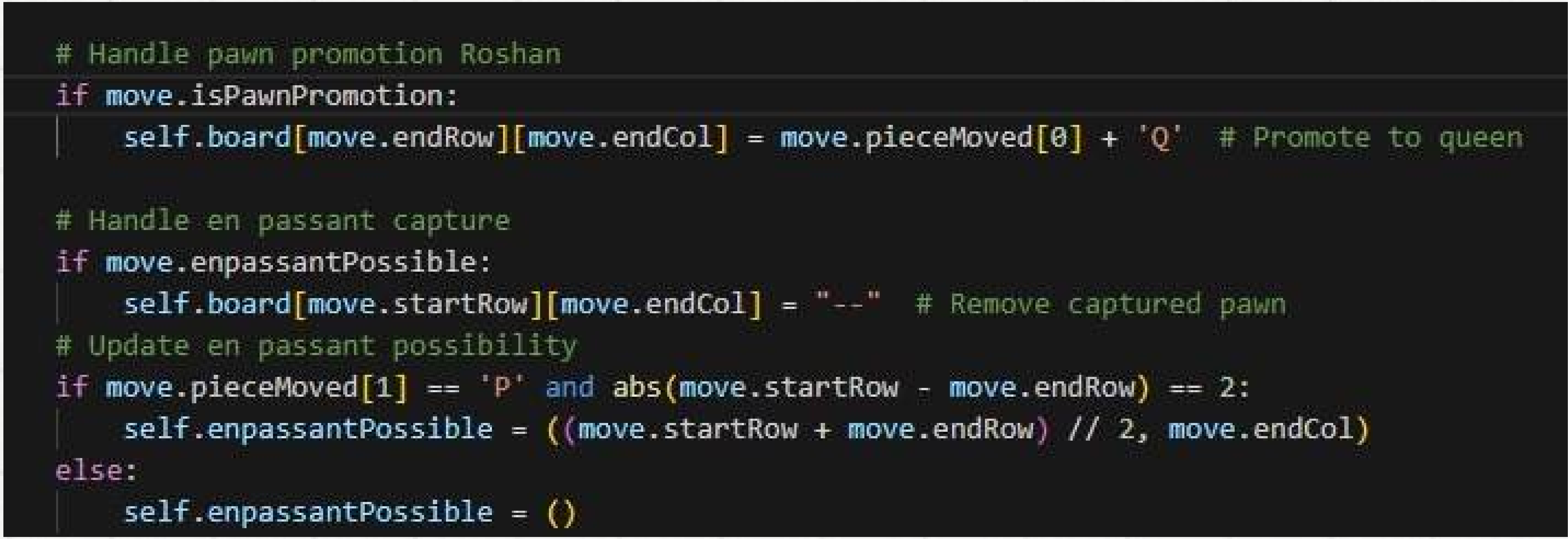


```
getValidMoves(self):#Done BY Hossam
# Returns a list of all valid moves, considering checks and pins
moves = self.getAllPossibleMoves()  # Get all possible moves
for i in range(len(moves) - 1, -1, -1):  # Iterate backward to avoid index shifting
    self.makeMove(moves[i])  # Make the move
    self.whiteToMove = not self.whiteToMove  # Switch turn to simulate opponent
    if self.inCheck():  # Check if the move leaves the player in check
        moves.remove(moves[i])  # Remove invalid move
    self.whiteToMove = not self.whiteToMove  # Switch turn back
    self.undoMove()  # Undo the move

# Check for checkmate or stalemate to be able to know lw fy haga hat3tlk or not
if len(moves) == 0:
    if self.inCheck():
        self.checkMate = True
    else:
        self.staleMate = True
else:
    self.checkMate = False
    self.staleMate = False

return moves

inCheck(self):#Hossam
# Determines if the current player is in check
if self.whiteToMove:
    return self.squareUnderAttack(self.whiteKingLocation[0], self.whiteKingLocation[1])#we do judge the whole moves by kn
else:
    return self.squareUnderAttack(self.blackKingLocation[0], self.blackKingLocation[1])
```

ChessEngine(PawnMoves)(Promotion)-Roshan Helmy.

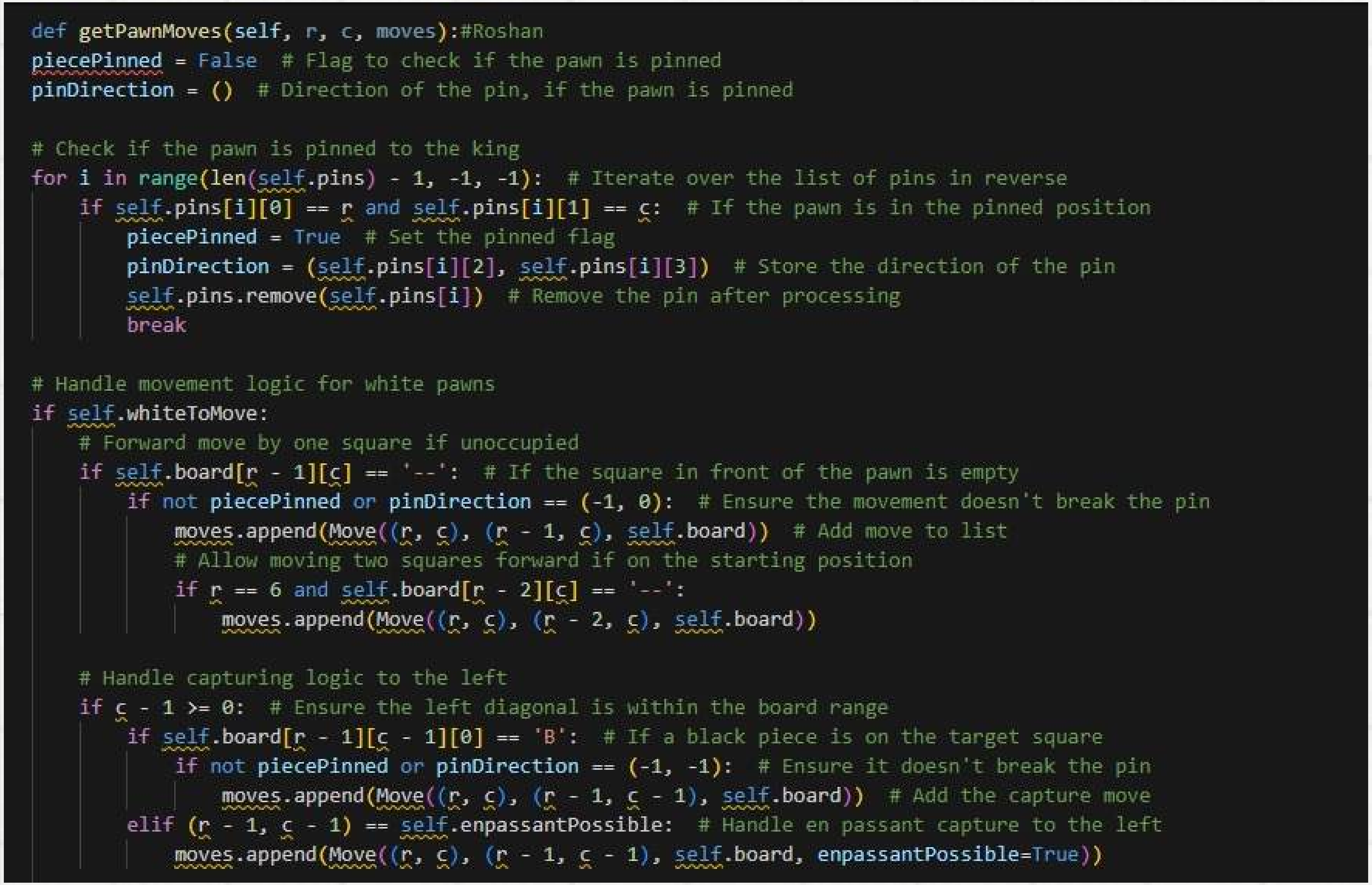
- Pawn Promotion: If a pawn reaches the end of the board, it is promoted to a queen by default. The code updates the board to reflect this change.
- En Passant Capture: If an en passant capture is possible, the code removes the captured pawn from the board and updates the enpassantPossible flag to indicate that en passant is no longer possible on the next move.
- White Pawn Movement:Forward Movement: The code checks if the square in front of the pawn is empty. If so, the pawn can move one square forward. Additionally, if the pawn is on its starting position, it can move two squares forward if both squares are empty.
- Black Pawn is the same as wihite but with reversed roles.

ChessEngine(PawnMoves)(Promotion)Snippets.



```python
# Handle pawn promotion Roshan
if move.isPawnPromotion:
    self.board[move.endRow][move.endCol] = move.pieceMoved[0] + 'Q'  # Promote to queen

# Handle en passant capture
if move.enpassantPossible:
    self.board[move.startRow][move.endCol] = "--"  # Remove captured pawn
# Update en passant possibility
if move.pieceMoved[1] == 'P' and abs(move.startRow - move.endRow) == 2:
    self.enpassantPossible = ((move.startRow + move.endRow) // 2, move.endCol)
else:
    self.enpassantPossible = ()
```

ChessEngine(PawnMoves)(Promotion)Snippets.

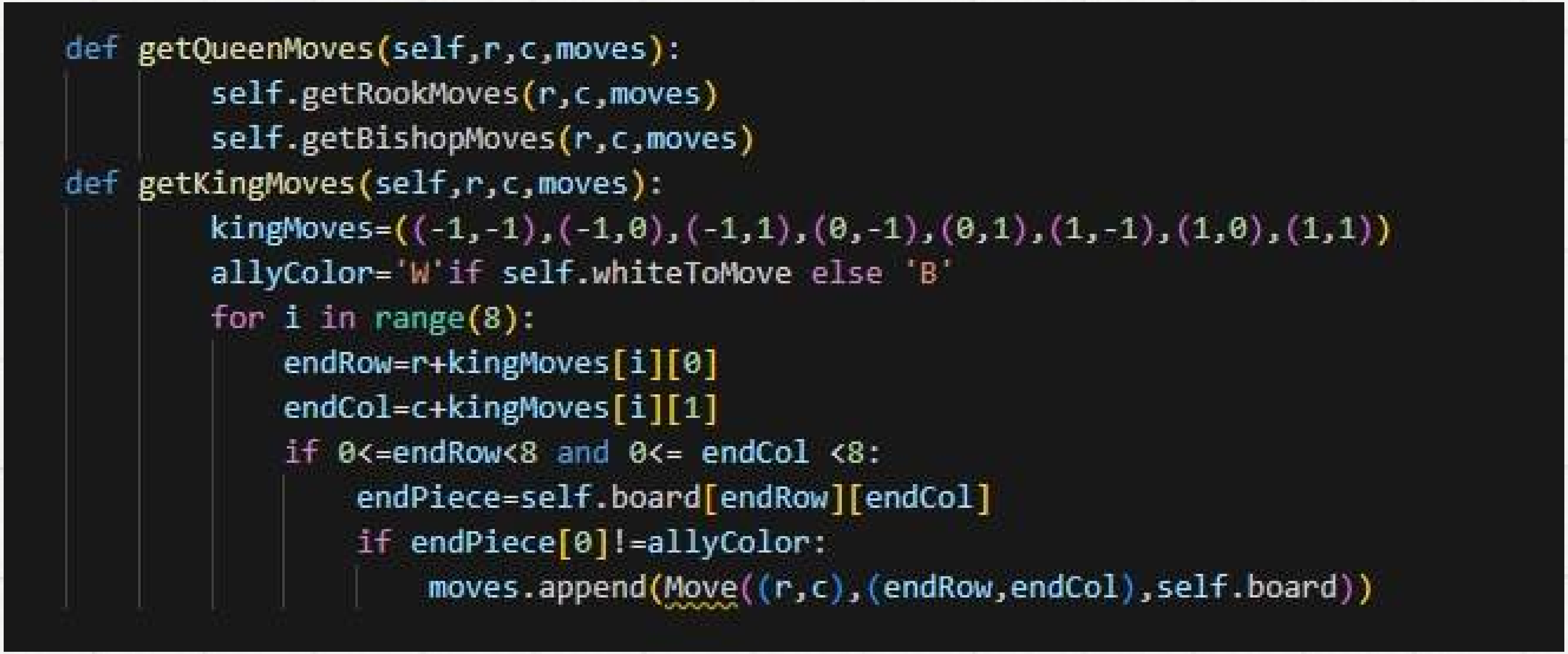


```
def getPawnMoves(self, r, c, moves):#Roshan
piecePinned = False  # Flag to check if the pawn is pinned
pinDirection = ()  # Direction of the pin, if the pawn is pinned

# Check if the pawn is pinned to the king
for i in range(len(self.pins) - 1, -1, -1):  # Iterate over the list of pins in reverse
    if self.pins[i][0] == r and self.pins[i][1] == c:  # If the pawn is in the pinned position
        piecePinned = True  # Set the pinned flag
        pinDirection = (self.pins[i][2], self.pins[i][3])  # Store the direction of the pin
        self.pins.remove(self.pins[i])  # Remove the pin after processing
        break

# Handle movement logic for white pawns
if self.whiteToMove:
    # Forward move by one square if unoccupied
    if self.board[r - 1][c] == '--':  # If the square in front of the pawn is empty
        if not piecePinned or pinDirection == (-1, 0):  # Ensure the movement doesn't break the pin
            moves.append(Move((r, c), (r - 1, c), self.board))  # Add move to list
            # Allow moving two squares forward if on the starting position
            if r == 6 and self.board[r - 2][c] == '--':
                moves.append(Move((r, c), (r - 2, c), self.board))

    # Handle capturing logic to the left
    if c - 1 >= 0:  # Ensure the left diagonal is within the board range
        if self.board[r - 1][c - 1][0] == 'B':  # If a black piece is on the target square
            if not piecePinned or pinDirection == (-1, -1):  # Ensure it doesn't break the pin
                moves.append(Move((r, c), (r - 1, c - 1), self.board))  # Add the capture move
        elif (r - 1, c - 1) == self.enpassantPossible:  # Handle en passant capture to the left
            moves.append(Move((r, c), (r - 1, c - 1), self.board, enpassantPossible=True))
```

ChessEngine(Queen and King Moves)-Ziad Yakout.

- getQueenMoves: This function calculates all possible moves for a queen on a chessboard. It does this by combining the moves of a rook and a bishop.
- getKingMoves: This function calculates all possible moves for a king on a chessboard. It checks the eight squares surrounding the king and adds valid moves to a list, considering the king's movement restrictions and the presence of other pieces.

ChessEngine(King and Queen Moves)Snippets.

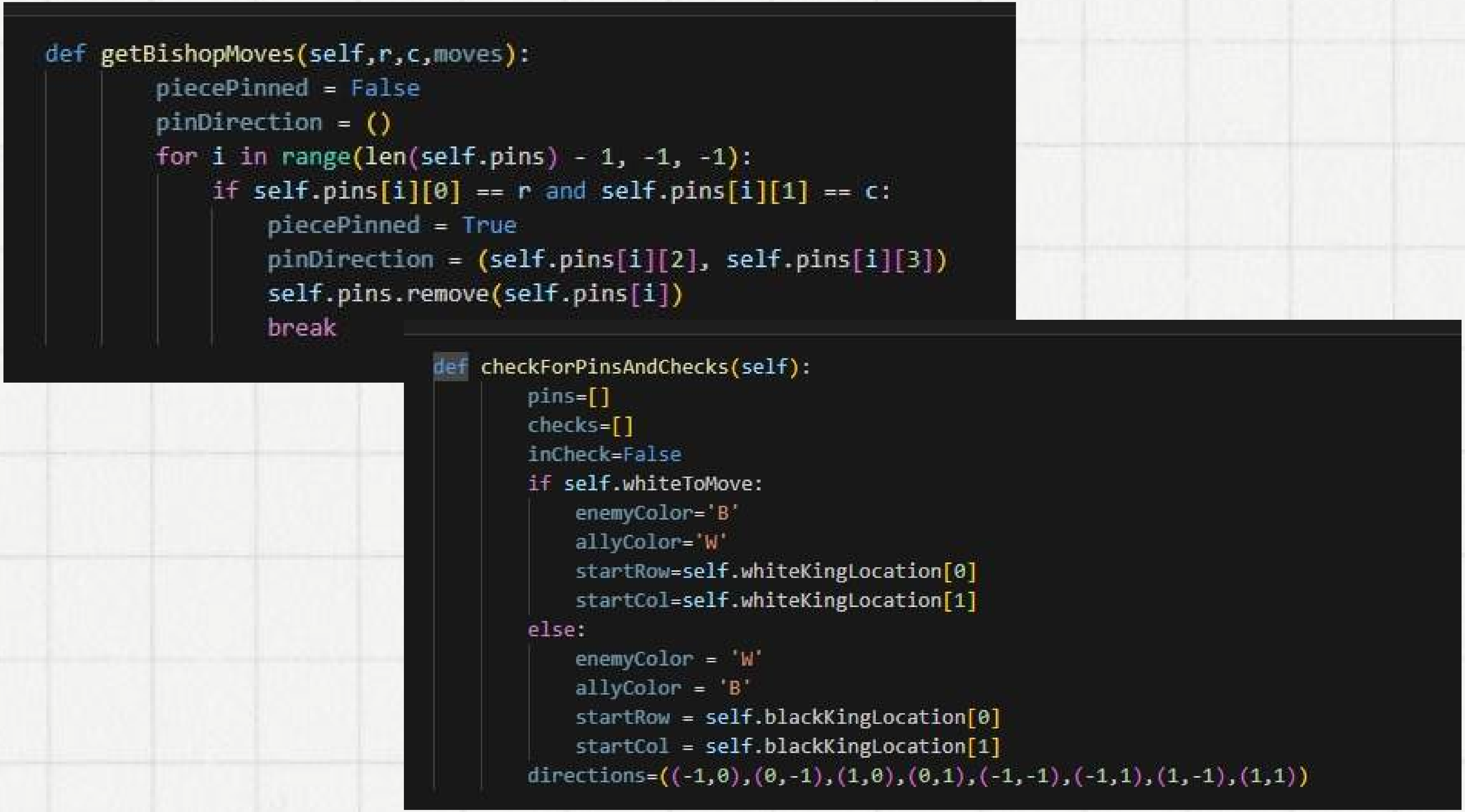


```python
def getQueenMoves(self,r,c,moves):
        self.getRookMoves(r,c,moves)
        self.getBishopMoves(r,c,moves)
def getKingMoves(self,r,c,moves):
        kingMoves=((-1,-1),(-1,0),(-1,1),(0,-1),(0,1),(1,-1),(1,0),(1,1))
        allyColor='W'if self.whiteToMove else 'B'
        for i in range(8):
            endRow=r+kingMoves[i][0]
            endCol=c+kingMoves[i][1]
            if 0<=endRow<8 and 0<= endCol <8:
                endPiece=self.board[endRow][endCol]
                if endPiece[0]!=allyColor:
                    moves.append(Move((r,c),(endRow,endCol),self.board))
```

ChessEngine(BishopMoves)(PinCheck)Omar Medhat.

- It first checks if the bishop is pinned to another piece.
- If the bishop is not pinned, it iterates over all possible diagonal moves and adds valid moves to a list.
- The function ensures that the moves do not put the king in check and that the bishop doesn't move through other pieces.
- This code defines a function checkForPinsAndChecks that checks for pins and checks on the current board state by iterating through possible moves and determining if the king is in danger.

ChessEngine(BishopMoves)(PinCheck)Snippets.



```
def getBishopMoves(self,r,c,moves):
        piecePinned = False
        pinDirection = ()
        for i in range(len(self.pins) - 1, -1, -1):
            if self.pins[i][0] == r and self.pins[i][1] == c:
                piecePinned = True
                pinDirection = (self.pins[i][2], self.pins[i][3])
                self.pins.remove(self.pins[i])
                break
```

```
def checkForPinsAndChecks(self):
        pins=[]
        checks=[]
        inCheck=False
        if self.whiteToMove:
            enemyColor='B'
            allyColor='W'
            startRow=self.whiteKingLocation[0]
            startCol=self.whiteKingLocation[1]
        else:
            enemyColor = 'W'
            allyColor = 'B'
            startRow = self.blackKingLocation[0]
            startCol = self.blackKingLocation[1]
        directions=((-1,0),(0,-1),(1,0),(0,1),(-1,-1),(-1,1),(1,-1),(1,1))
```

Challenges:

We first fixed piece movement by tracking clicks and updating cells. Then, we handled valid moves to ensure legality. Next, we checked if a move left the king in checkmate, making it invalid. For checkmate scenarios with no moves, we added a “Game Over” screen. Finally, AI was integrated.

Thank You So Much!

PRESENTED BY:
1-OMAR MAHMOUD NABIL TL.(23012090)
2- ROSHAN MOHAMED HELMY (23012096)
3-MALAK ALAAELDIN ABDELHAMID (23012095)
4-HOSSAM KHAIRY MOHAMED EID(23012099)
5-OMAR MEDHAT ABDELHADY (23012218)
6-ZIAD YAKOUT IBRAHIM (23012100)
7-AHMED MOHAMED ABDELGELIL (23012102)