Chess game key features

## Overview

This is an implementation of a chess game in Python using the Pygame library.

## Features

- Player vs. Player mode
- Classic chess rules and moves
- Basic piece capturing
- Check/Check mate
- Stalemate
- En passant and Castling moves
- Pawn promotion
- UI indications for possible moves for each piece on-click
- UI to show captured pieces and game over
- UI Button to start new game

## Prerequisites

- Python 3.3

## Setup the project

- **Github link: https://github.com/akshith-katuri/chess-game**
- **Clone the repository: git clone https://github.com/akshith-katuri/chess-game.git**
- Update pip - **pip install --upgrade pip**
- Install Virtualenv which is a tool to set up your Python environments: **pip install virtualenv**
- Create virtual environment: **python3.8 -m venv venv**
- activate virtual environment:

venv/Scripts/activate.bat //In CMD

venv/Scripts/Activate.ps1 //In Powershel

- pip list
- Install Pygame library - **pip install pygame**
- Navigate to the project directory: **cd chess-game**

## How to Play

- Start the game by running **python chess_game.py**

## Controls

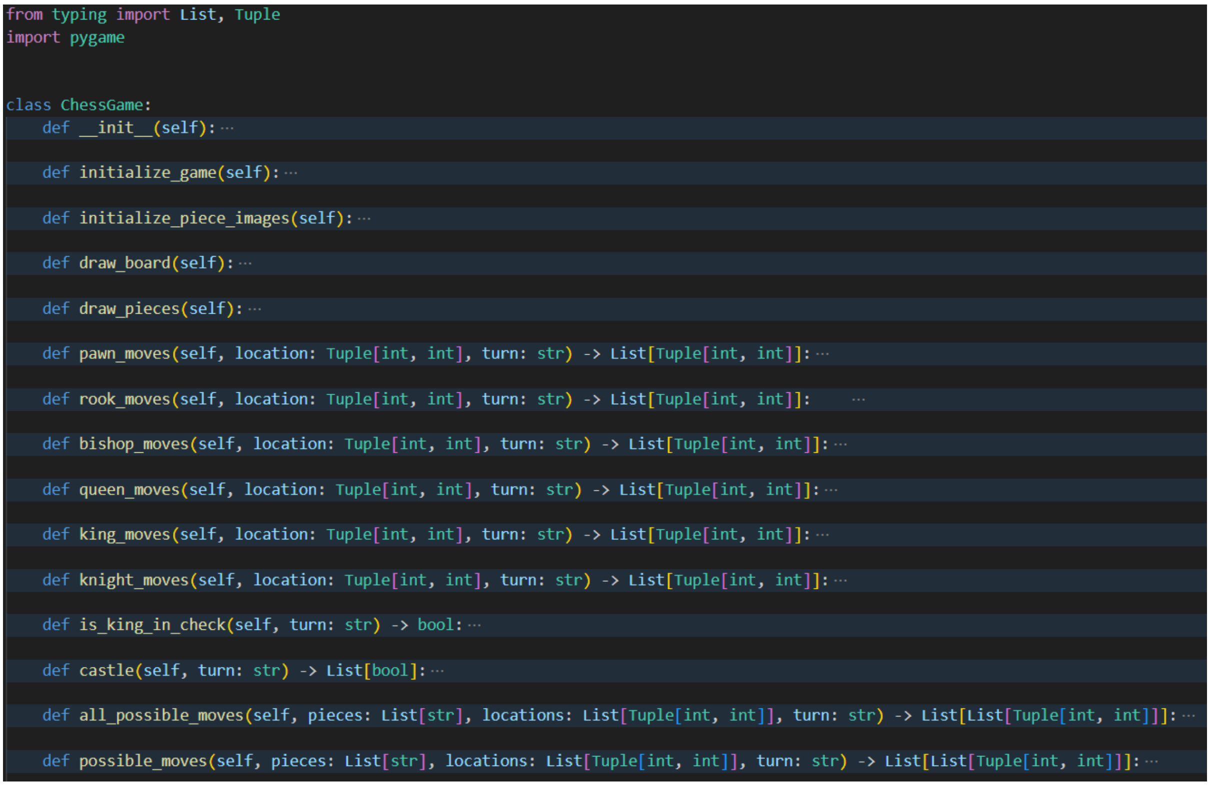
- Click on a piece to select it. Then You will see all possible moves of the selected piece

- Click on a valid square to move the selected piece.

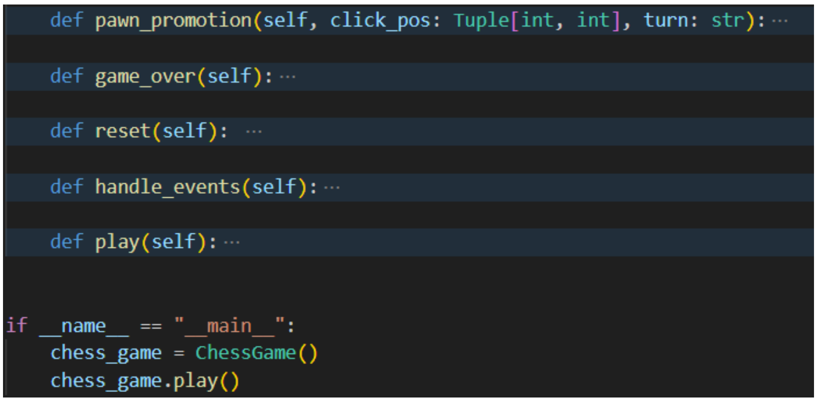
## Game Rules

- Follows standard chess rules

Code Structure:

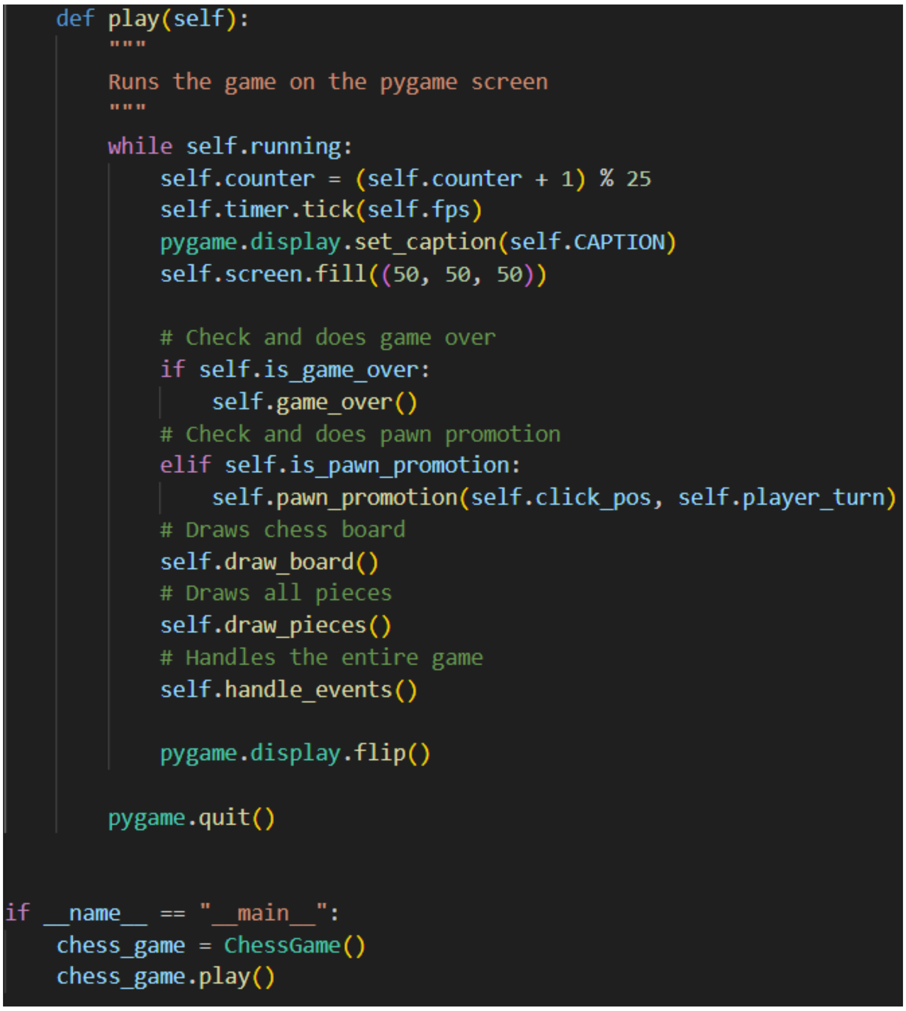


```
from typing import List, Tuple
import pygame


class ChessGame:
    def __init__(self):···

    def initialize_game(self):···

    def initialize_piece_images(self):···

    def draw_board(self):···

    def draw_pieces(self):···

    def pawn_moves(self, location: Tuple[int, int], turn: str) -> List[Tuple[int, int]]:···

    def rook_moves(self, location: Tuple[int, int], turn: str) -> List[Tuple[int, int]]:    ···

    def bishop_moves(self, location: Tuple[int, int], turn: str) -> List[Tuple[int, int]]:···

    def queen_moves(self, location: Tuple[int, int], turn: str) -> List[Tuple[int, int]]:···

    def king_moves(self, location: Tuple[int, int], turn: str) -> List[Tuple[int, int]]:···

    def knight_moves(self, location: Tuple[int, int], turn: str) -> List[Tuple[int, int]]:···

    def is_king_in_check(self, turn: str) -> bool:···

    def castle(self, turn: str) -> List[bool]:···

    def all_possible_moves(self, pieces: List[str], locations: List[Tuple[int, int]], turn: str) -> List[List[Tuple[int, int]]]:···

    def possible_moves(self, pieces: List[str], locations: List[Tuple[int, int]], turn: str) -> List[List[Tuple[int, int]]]:···
```



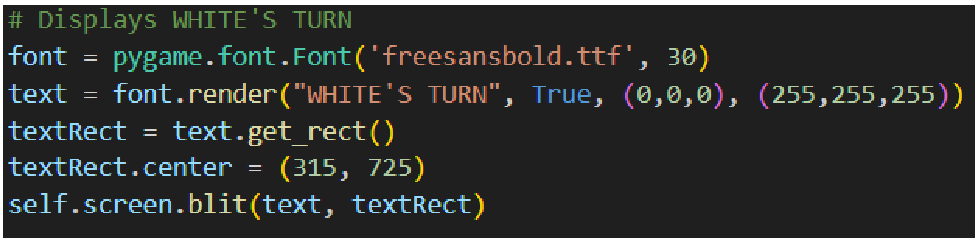
```
    def pawn_promotion(self, click_pos: Tuple[int, int], turn: str):···

    def game_over(self):···

    def reset(self):  ···

    def handle_events(self):···

    def play(self):···


if __name__ == "__main__":
    chess_game = ChessGame()
    chess_game.play()
```

How the game runs:

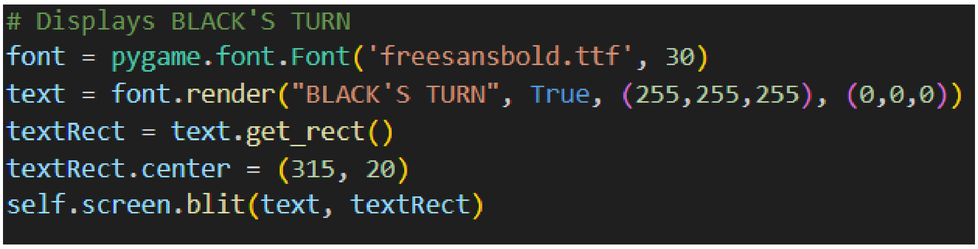


```python
    def play(self):
        """
        Runs the game on the pygame screen
        """
        while self.running:
            self.counter = (self.counter + 1) % 25
            self.timer.tick(self.fps)
            pygame.display.set_caption(self.CAPTION)
            self.screen.fill((50, 50, 50))

            # Check and does game over
            if self.is_game_over:
                self.game_over()
            # Check and does pawn promotion
            elif self.is_pawn_promotion:
                self.pawn_promotion(self.click_pos, self.player_turn)
            # Draws chess board
            self.draw_board()
            # Draws all pieces
            self.draw_pieces()
            # Handles the entire game
            self.handle_events()

            pygame.display.flip()

        pygame.quit()


if __name__ == "__main__":
    chess_game = ChessGame()
    chess_game.play()
```

Display turn:

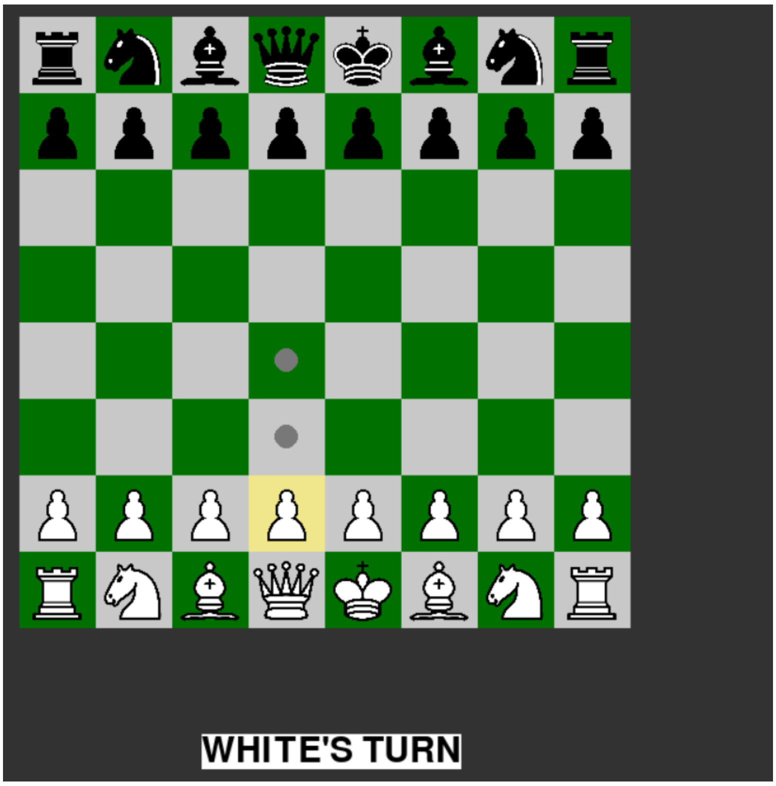


```python
# Displays WHITE'S TURN
font = pygame.font.Font('freesansbold.ttf', 30)
text = font.render("WHITE'S TURN", True, (0,0,0), (255,255,255))
textRect = text.get_rect()
textRect.center = (315, 725)
self.screen.blit(text, textRect)
```

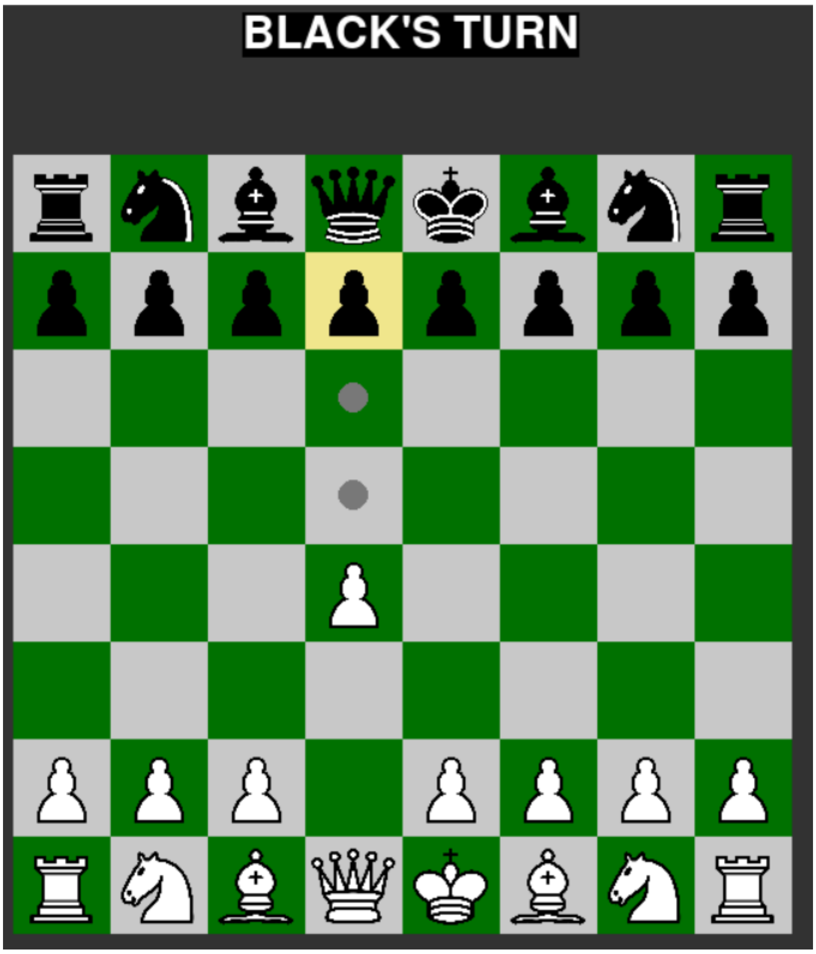


```python
# Displays BLACK'S TURN
font = pygame.font.Font('freesansbold.ttf', 30)
text = font.render("BLACK'S TURN", True, (255,255,255), (0,0,0))
textRect = text.get_rect()
textRect.center = (315, 20)
self.screen.blit(text, textRect)
```

Selected piece and its possible moves displayed:

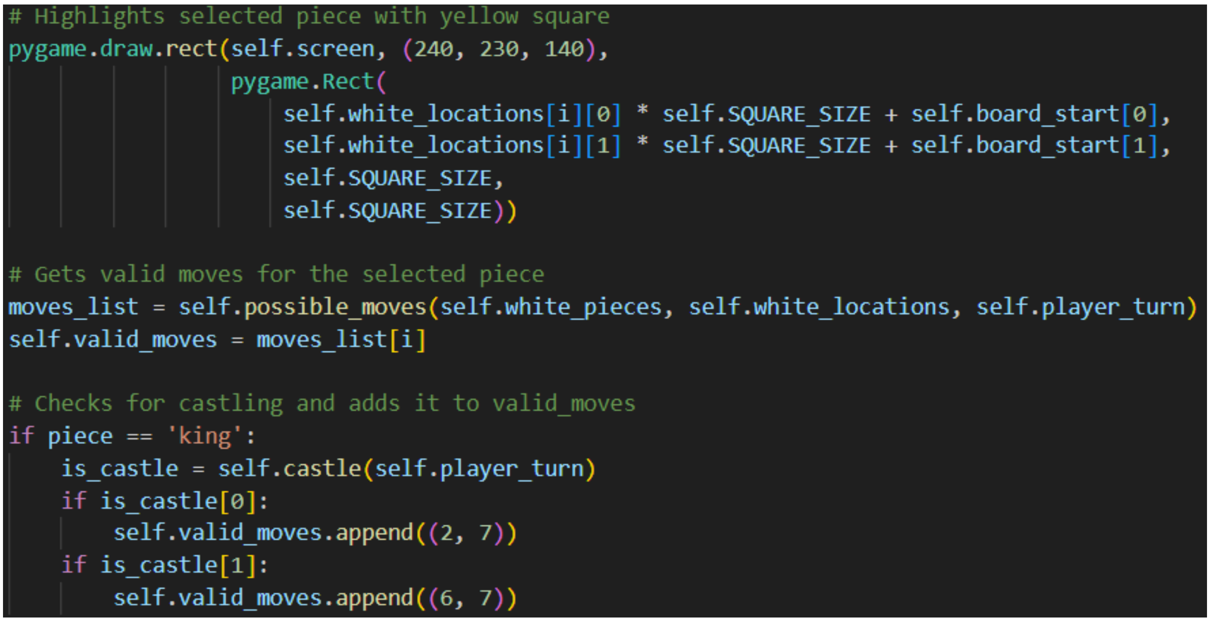


WHITE'S TURN



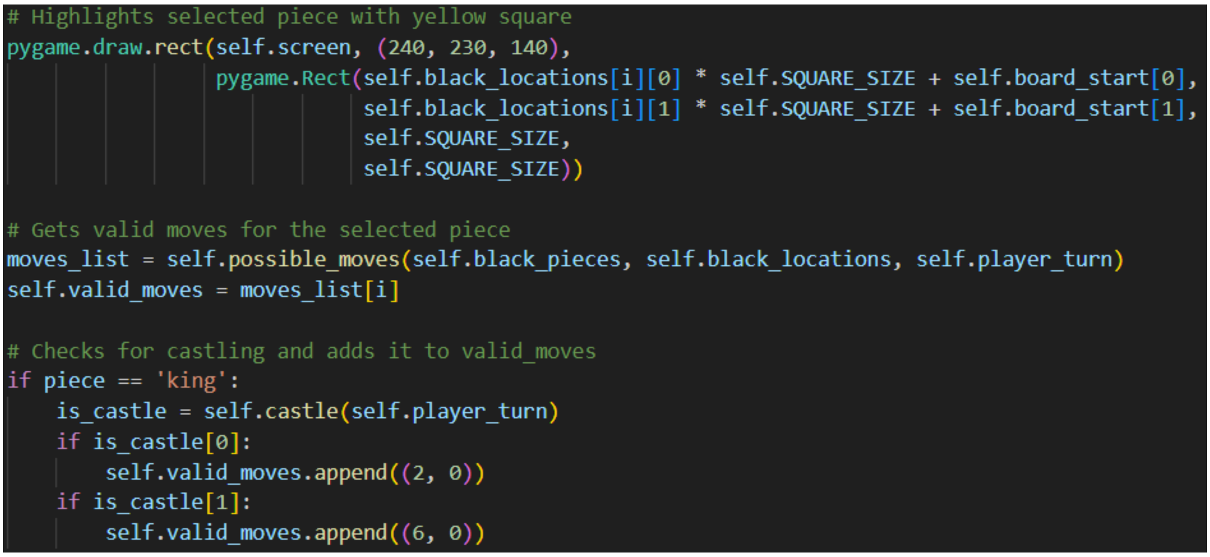
BLACK'S TURN

White player selection and valid moves:



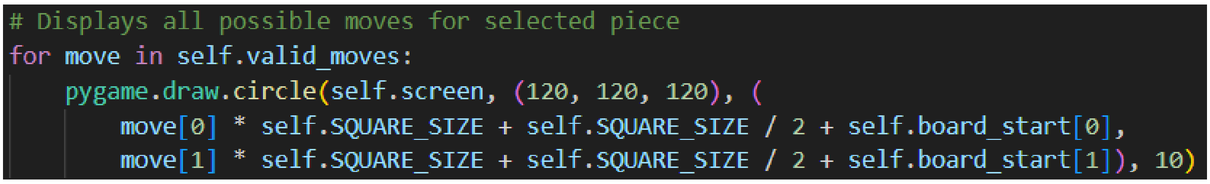
```
# Highlights selected piece with yellow square
pygame.draw.rect(self.screen, (240, 230, 140),
                 pygame.Rect(
                     self.white_locations[i][0] * self.SQUARE_SIZE + self.board_start[0],
                     self.white_locations[i][1] * self.SQUARE_SIZE + self.board_start[1],
                     self.SQUARE_SIZE,
                     self.SQUARE_SIZE))

# Gets valid moves for the selected piece
moves_list = self.possible_moves(self.white_pieces, self.white_locations, self.player_turn)
self.valid_moves = moves_list[i]

# Checks for castling and adds it to valid_moves
if piece == 'king':
    is_castle = self.castle(self.player_turn)
    if is_castle[0]:
        self.valid_moves.append((2, 7))
    if is_castle[1]:
        self.valid_moves.append((6, 7))
```

Black player selection and valid moves:



```
# Highlights selected piece with yellow square
pygame.draw.rect(self.screen, (240, 230, 140),
                 pygame.Rect(self.black_locations[i][0] * self.SQUARE_SIZE + self.board_start[0],
                             self.black_locations[i][1] * self.SQUARE_SIZE + self.board_start[1],
                             self.SQUARE_SIZE,
                             self.SQUARE_SIZE))

# Gets valid moves for the selected piece
moves_list = self.possible_moves(self.black_pieces, self.black_locations, self.player_turn)
self.valid_moves = moves_list[i]

# Checks for castling and adds it to valid_moves
if piece == 'king':
    is_castle = self.castle(self.player_turn)
    if is_castle[0]:
        self.valid_moves.append((2, 0))
    if is_castle[1]:
        self.valid_moves.append((6, 0))
```

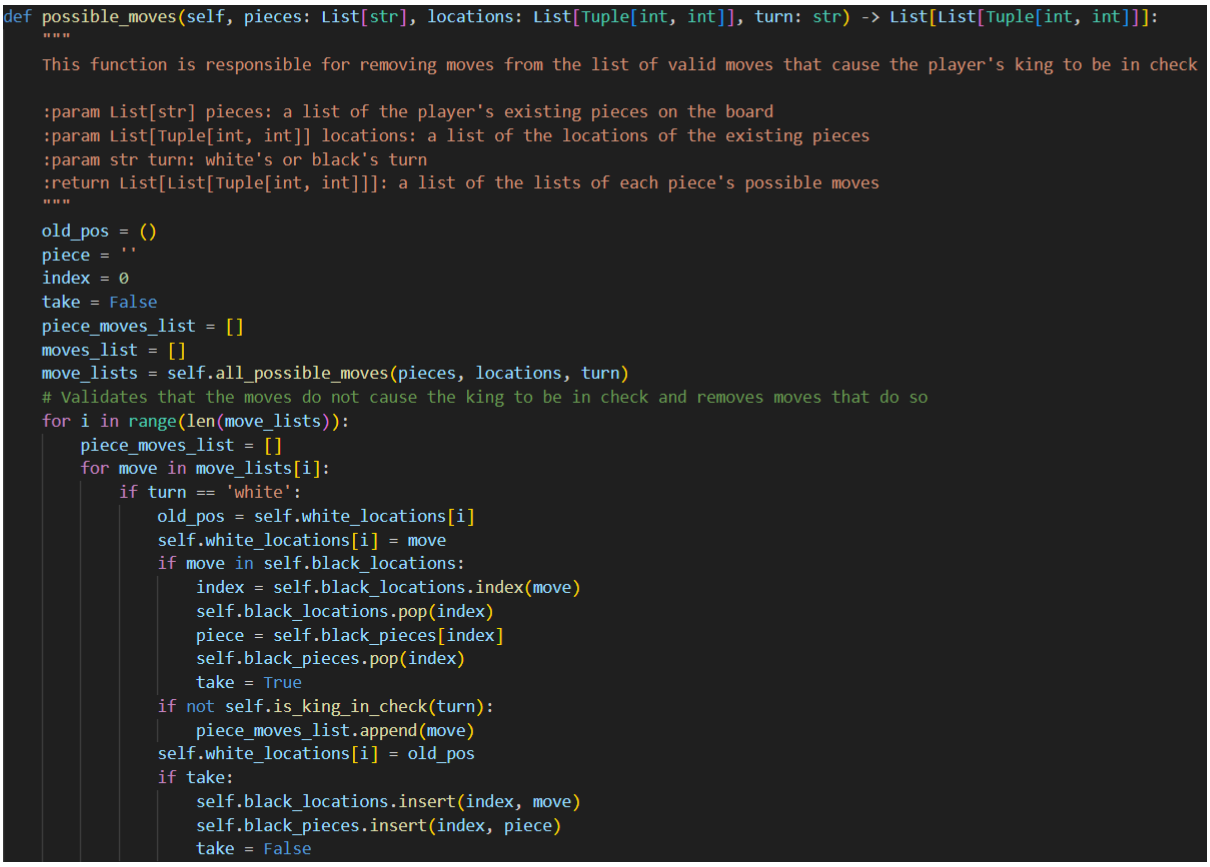
Displays valid moves:



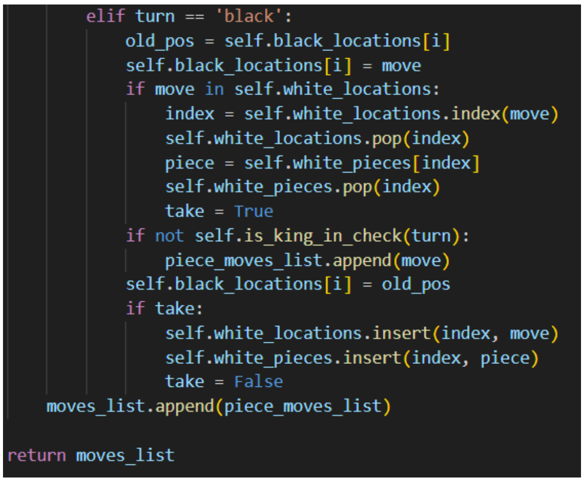
```
# Displays all possible moves for selected piece
for move in self.valid_moves:
    pygame.draw.circle(self.screen, (120, 120, 120), (
        move[0] * self.SQUARE_SIZE + self.SQUARE_SIZE / 2 + self.board_start[0],
        move[1] * self.SQUARE_SIZE + self.SQUARE_SIZE / 2 + self.board_start[1]), 10)
```

Calculates valid moves using these 2 functions:

possible_moves() function:

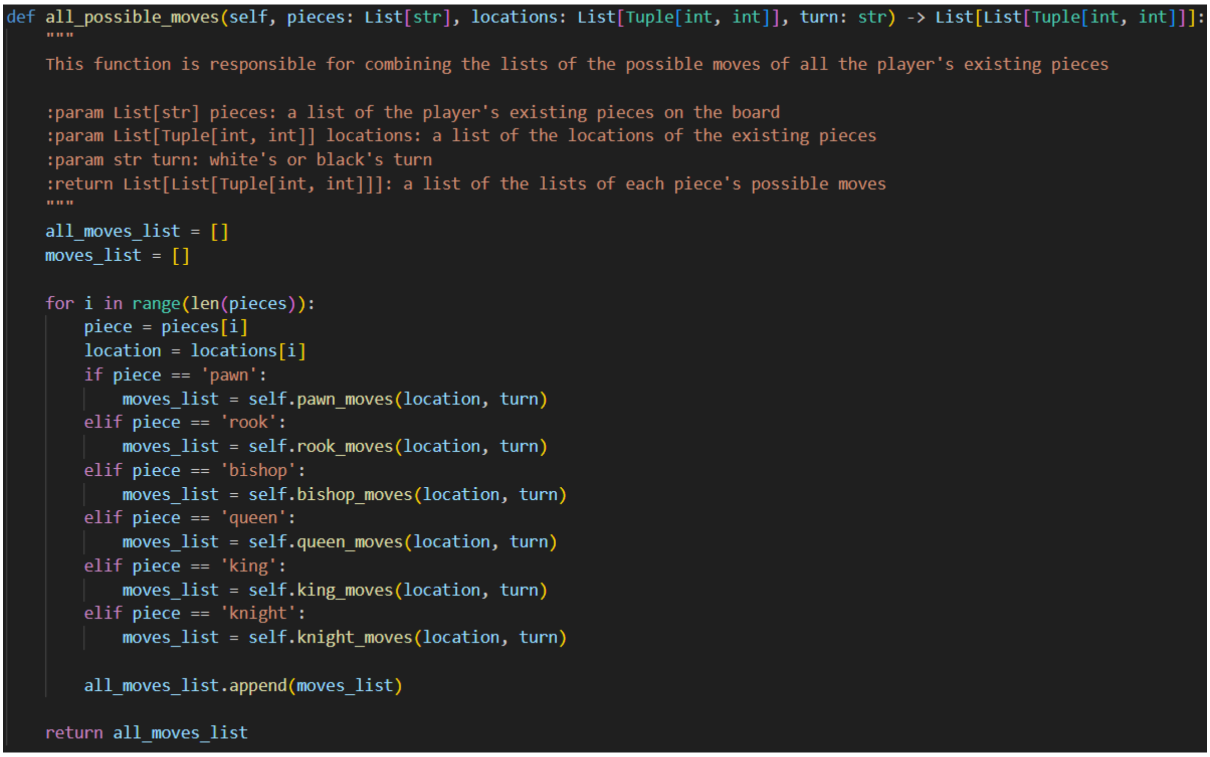


```python
def possible_moves(self, pieces: List[str], locations: List[Tuple[int, int]], turn: str) -> List[List[Tuple[int, int]]]:
    """
    This function is responsible for removing moves from the list of valid moves that cause the player's king to be in check

    :param List[str] pieces: a list of the player's existing pieces on the board
    :param List[Tuple[int, int]] locations: a list of the locations of the existing pieces
    :param str turn: white's or black's turn
    :return List[List[Tuple[int, int]]]: a list of the lists of each piece's possible moves
    """
    old_pos = ()
    piece = ''
    index = 0
    take = False
    piece_moves_list = []
    moves_list = []
    move_lists = self.all_possible_moves(pieces, locations, turn)
    # Validates that the moves do not cause the king to be in check and removes moves that do so
    for i in range(len(move_lists)):
        piece_moves_list = []
        for move in move_lists[i]:
            if turn == 'white':
                old_pos = self.white_locations[i]
                self.white_locations[i] = move
                if move in self.black_locations:
                    index = self.black_locations.index(move)
                    self.black_locations.pop(index)
                    piece = self.black_pieces[index]
                    self.black_pieces.pop(index)
                    take = True
                if not self.is_king_in_check(turn):
                    piece_moves_list.append(move)
                self.white_locations[i] = old_pos
                if take:
                    self.black_locations.insert(index, move)
                    self.black_pieces.insert(index, piece)
                    take = False
```



```python
        elif turn == 'black':
            old_pos = self.black_locations[i]
            self.black_locations[i] = move
            if move in self.white_locations:
                index = self.white_locations.index(move)
                self.white_locations.pop(index)
                piece = self.white_pieces[index]
                self.white_pieces.pop(index)
                take = True
            if not self.is_king_in_check(turn):
                piece_moves_list.append(move)
            self.black_locations[i] = old_pos
            if take:
                self.white_locations.insert(index, move)
                self.white_pieces.insert(index, piece)
                take = False
    moves_list.append(piece_moves_list)

return moves_list
```

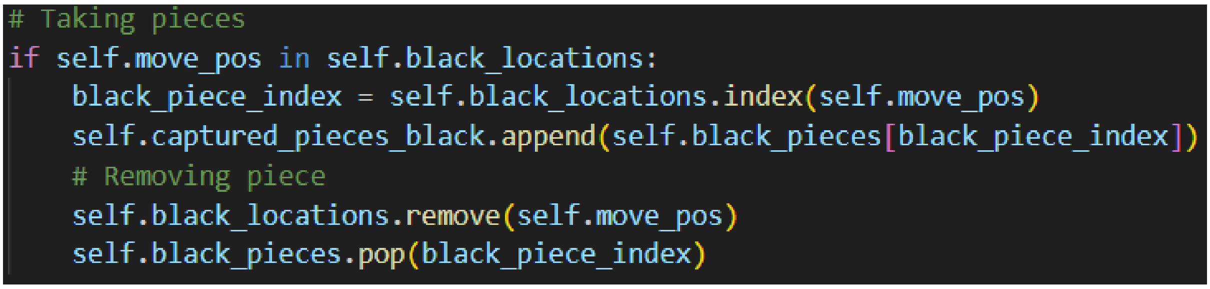
all_possible_moves() function:



```
def all_possible_moves(self, pieces: List[str], locations: List[Tuple[int, int]], turn: str) -> List[List[Tuple[int, int]]]:
    """
    This function is responsible for combining the lists of the possible moves of all the player's existing pieces

    :param List[str] pieces: a list of the player's existing pieces on the board
    :param List[Tuple[int, int]] locations: a list of the locations of the existing pieces
    :param str turn: white's or black's turn
    :return List[List[Tuple[int, int]]]: a list of the lists of each piece's possible moves
    """
    all_moves_list = []
    moves_list = []

    for i in range(len(pieces)):
        piece = pieces[i]
        location = locations[i]
        if piece == 'pawn':
            moves_list = self.pawn_moves(location, turn)
        elif piece == 'rook':
            moves_list = self.rook_moves(location, turn)
        elif piece == 'bishop':
            moves_list = self.bishop_moves(location, turn)
        elif piece == 'queen':
            moves_list = self.queen_moves(location, turn)
        elif piece == 'king':
            moves_list = self.king_moves(location, turn)
        elif piece == 'knight':
            moves_list = self.knight_moves(location, turn)

        all_moves_list.append(moves_list)

    return all_moves_list
```

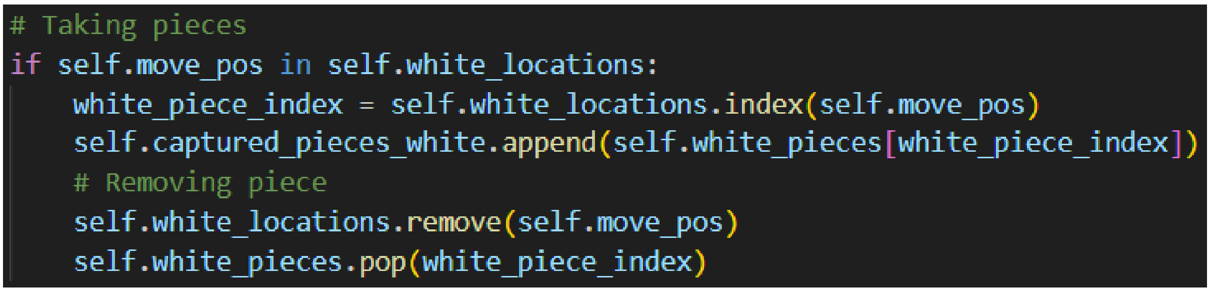
Taking pieces normally:

White player:



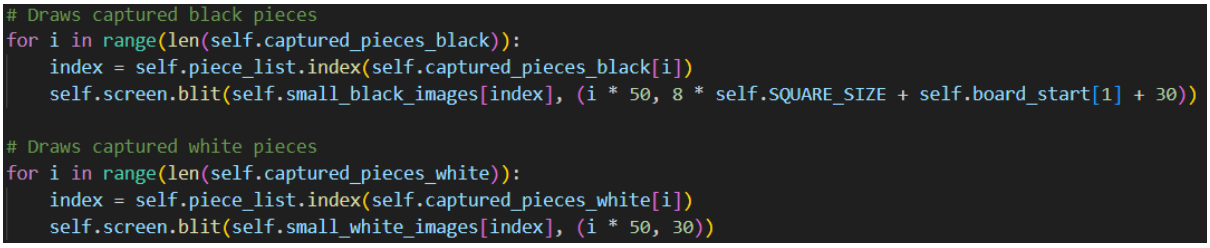
```
# Taking pieces
if self.move_pos in self.black_locations:
    black_piece_index = self.black_locations.index(self.move_pos)
    self.captured_pieces_black.append(self.black_pieces[black_piece_index])
    # Removing piece
    self.black_locations.remove(self.move_pos)
    self.black_pieces.pop(black_piece_index)
```

Black player:



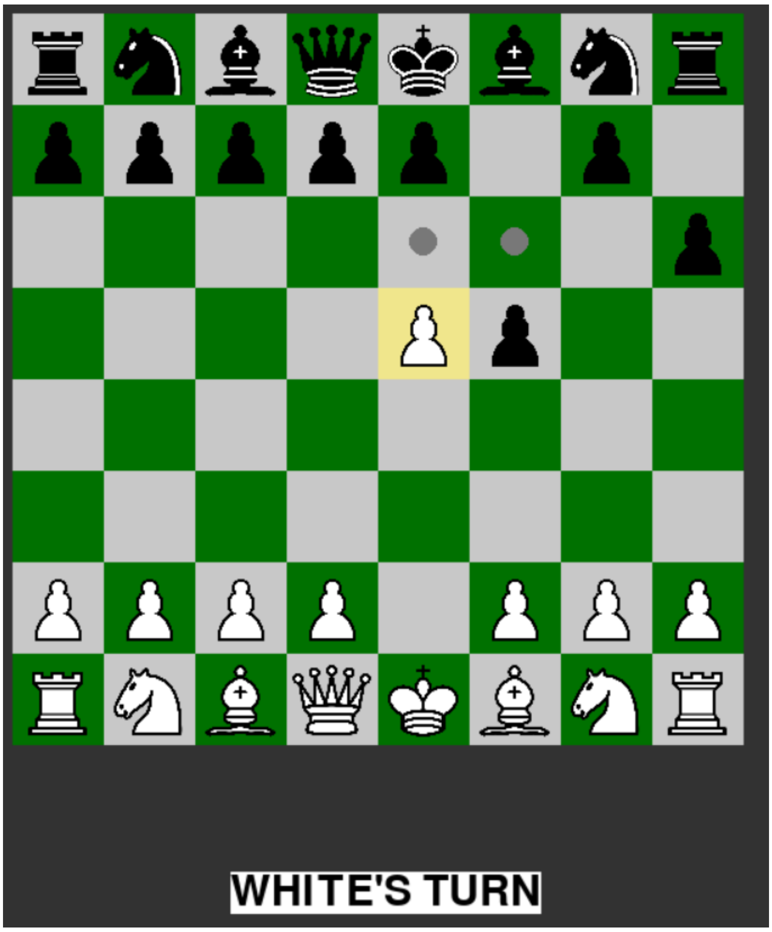
```
# Taking pieces
if self.move_pos in self.white_locations:
    white_piece_index = self.white_locations.index(self.move_pos)
    self.captured_pieces_white.append(self.white_pieces[white_piece_index])
    # Removing piece
    self.white_locations.remove(self.move_pos)
    self.white_pieces.pop(white_piece_index)
```

Displaying captured pieces:

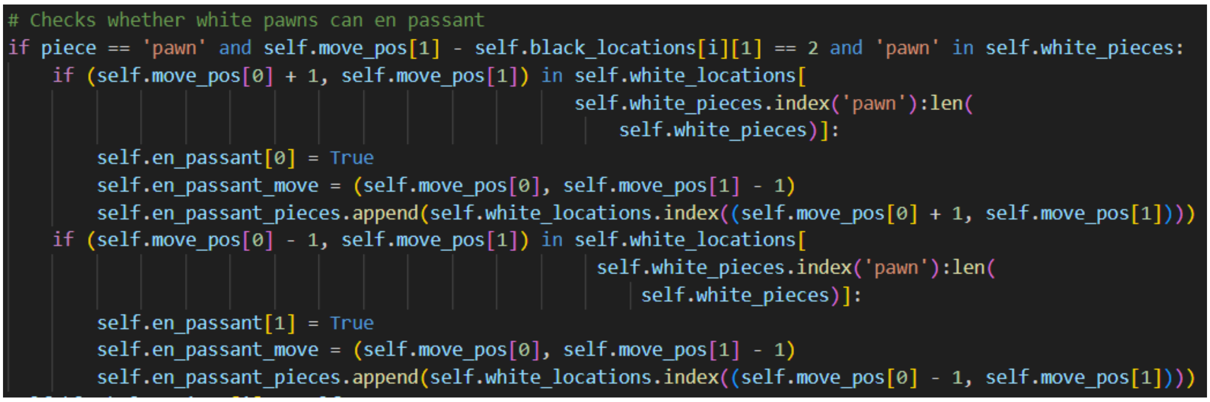


```python
# Draws captured black pieces
for i in range(len(self.captured_pieces_black)):
    index = self.piece_list.index(self.captured_pieces_black[i])
    self.screen.blit(self.small_black_images[index], (i * 50, 8 * self.SQUARE_SIZE + self.board_start[1] + 30))

# Draws captured white pieces
for i in range(len(self.captured_pieces_white)):
    index = self.piece_list.index(self.captured_pieces_white[i])
    self.screen.blit(self.small_white_images[index], (i * 50, 30))
```

En passant:

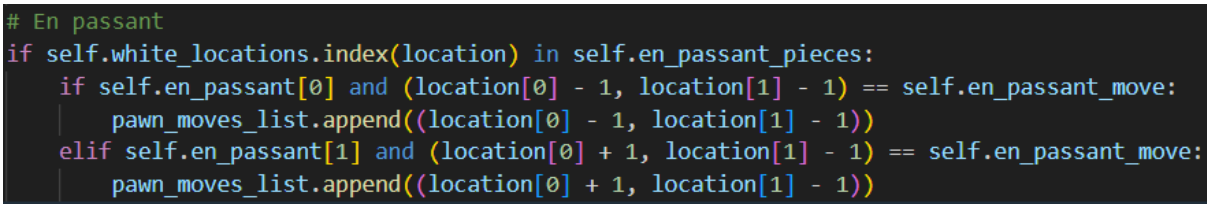


Checks if white pawn can en passant

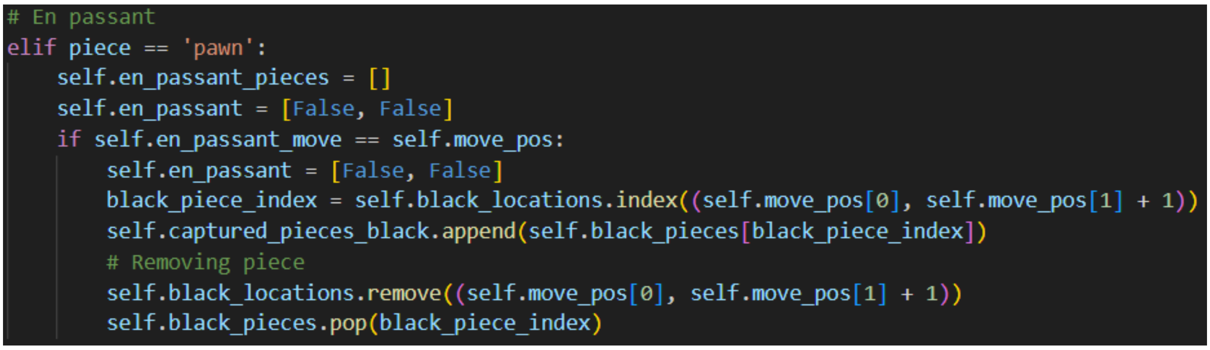


```python
# Checks whether white pawns can en passant
if piece == 'pawn' and self.move_pos[1] - self.black_locations[i][1] == 2 and 'pawn' in self.white_pieces:
    if (self.move_pos[0] + 1, self.move_pos[1]) in self.white_locations[
                                                    self.white_pieces.index('pawn'):len(
                                                        self.white_pieces)]:
        self.en_passant[0] = True
        self.en_passant_move = (self.move_pos[0], self.move_pos[1] - 1)
        self.en_passant_pieces.append(self.white_locations.index((self.move_pos[0] + 1, self.move_pos[1])))
    if (self.move_pos[0] - 1, self.move_pos[1]) in self.white_locations[
                                                      self.white_pieces.index('pawn'):len(
                                                          self.white_pieces)]:
        self.en_passant[1] = True
        self.en_passant_move = (self.move_pos[0], self.move_pos[1] - 1)
        self.en_passant_pieces.append(self.white_locations.index((self.move_pos[0] - 1, self.move_pos[1])))
```

White en passant move:

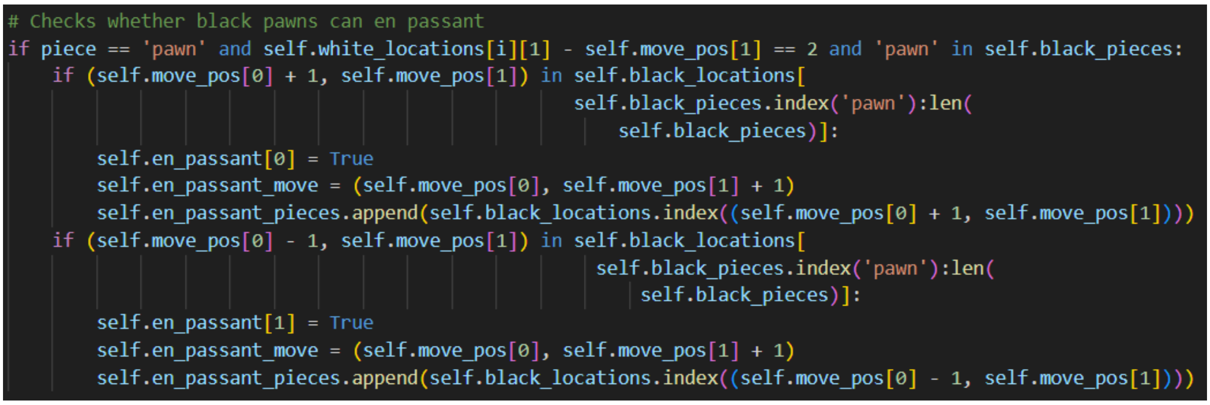


```
# En passant
if self.white_locations.index(location) in self.en_passant_pieces:
    if self.en_passant[0] and (location[0] - 1, location[1] - 1) == self.en_passant_move:
        pawn_moves_list.append((location[0] - 1, location[1] - 1))
    elif self.en_passant[1] and (location[0] + 1, location[1] - 1) == self.en_passant_move:
        pawn_moves_list.append((location[0] + 1, location[1] - 1))
```



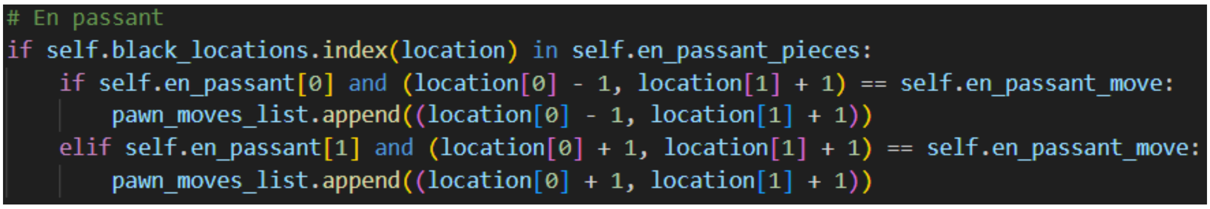
```
# En passant
elif piece == 'pawn':
    self.en_passant_pieces = []
    self.en_passant = [False, False]
    if self.en_passant_move == self.move_pos:
        self.en_passant = [False, False]
        black_piece_index = self.black_locations.index((self.move_pos[0], self.move_pos[1] + 1))
        self.captured_pieces_black.append(self.black_pieces[black_piece_index])
        # Removing piece
        self.black_locations.remove((self.move_pos[0], self.move_pos[1] + 1))
        self.black_pieces.pop(black_piece_index)
```

Checks if black pawn can en passant:

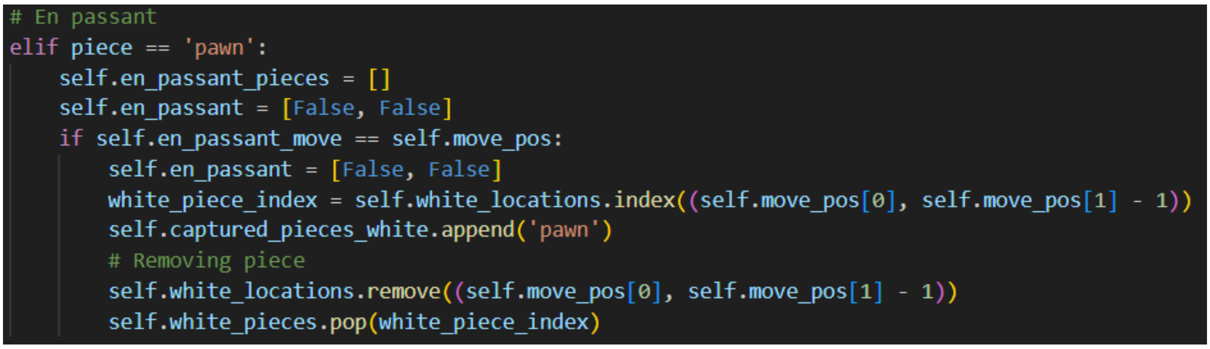


```
# Checks whether black pawns can en passant
if piece == 'pawn' and self.white_locations[i][1] - self.move_pos[1] == 2 and 'pawn' in self.black_pieces:
    if (self.move_pos[0] + 1, self.move_pos[1]) in self.black_locations[
                                                   self.black_pieces.index('pawn'):len(
                                                       self.black_pieces)]:
        self.en_passant[0] = True
        self.en_passant_move = (self.move_pos[0], self.move_pos[1] + 1)
        self.en_passant_pieces.append(self.black_locations.index((self.move_pos[0] + 1, self.move_pos[1])))
    if (self.move_pos[0] - 1, self.move_pos[1]) in self.black_locations[
                                                     self.black_pieces.index('pawn'):len(
                                                         self.black_pieces)]:
        self.en_passant[1] = True
        self.en_passant_move = (self.move_pos[0], self.move_pos[1] + 1)
        self.en_passant_pieces.append(self.black_locations.index((self.move_pos[0] - 1, self.move_pos[1])))
```

Black en passant move:

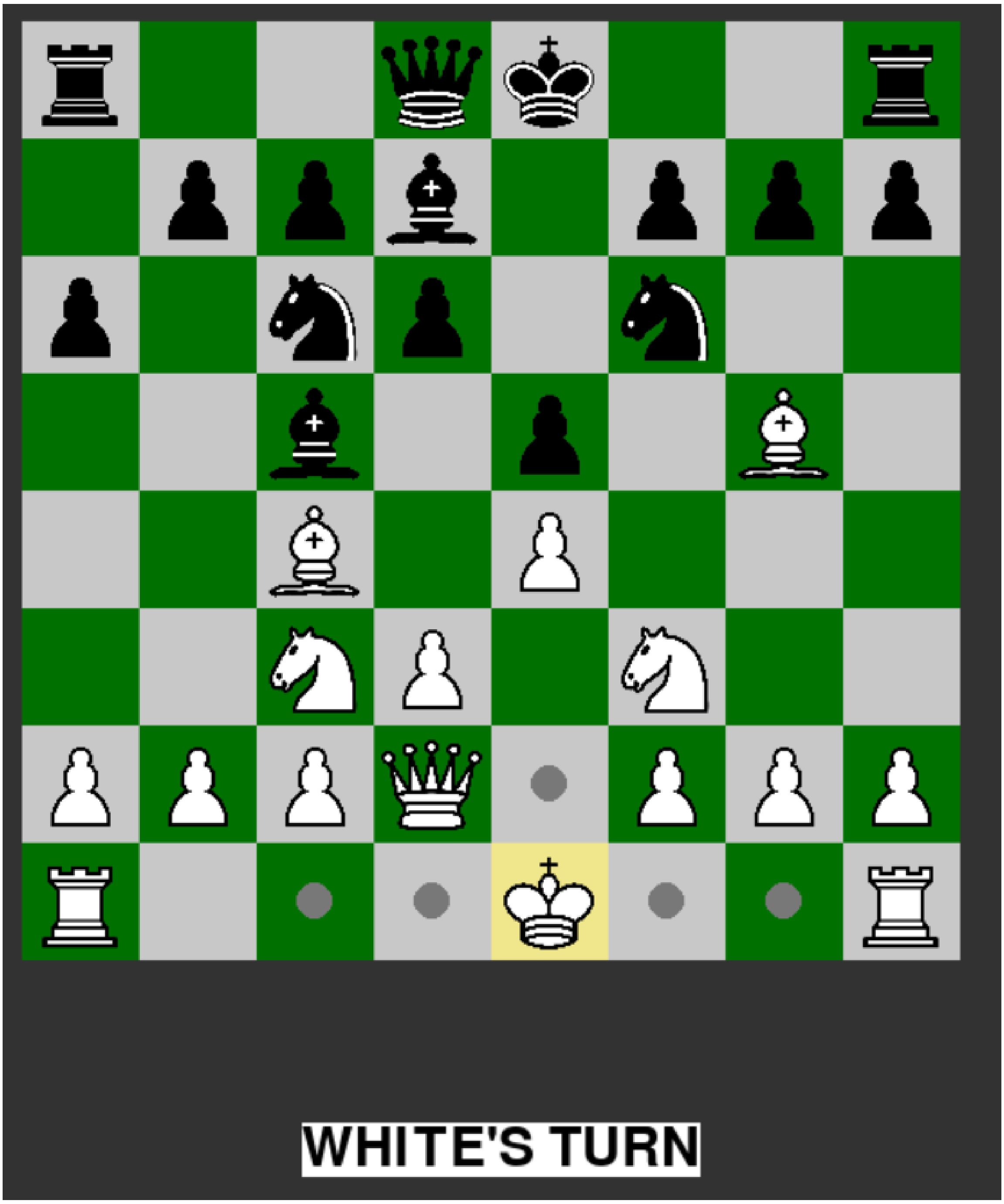


```
# En passant
if self.black_locations.index(location) in self.en_passant_pieces:
    if self.en_passant[0] and (location[0] - 1, location[1] + 1) == self.en_passant_move:
        pawn_moves_list.append((location[0] - 1, location[1] + 1))
    elif self.en_passant[1] and (location[0] + 1, location[1] + 1) == self.en_passant_move:
        pawn_moves_list.append((location[0] + 1, location[1] + 1))
```

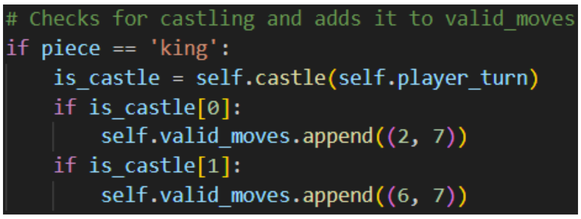


```
# En passant
elif piece == 'pawn':
    self.en_passant_pieces = []
    self.en_passant = [False, False]
    if self.en_passant_move == self.move_pos:
        self.en_passant = [False, False]
        white_piece_index = self.white_locations.index((self.move_pos[0], self.move_pos[1] - 1))
        self.captured_pieces_white.append('pawn')
        # Removing piece
        self.white_locations.remove((self.move_pos[0], self.move_pos[1] - 1))
        self.white_pieces.pop(white_piece_index)
```

Castling:

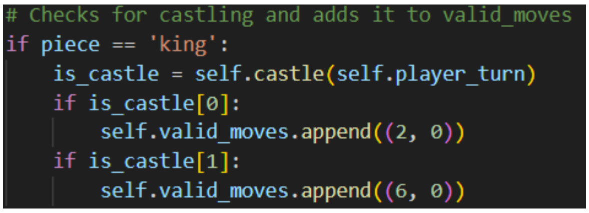


Checks for castling for white player:



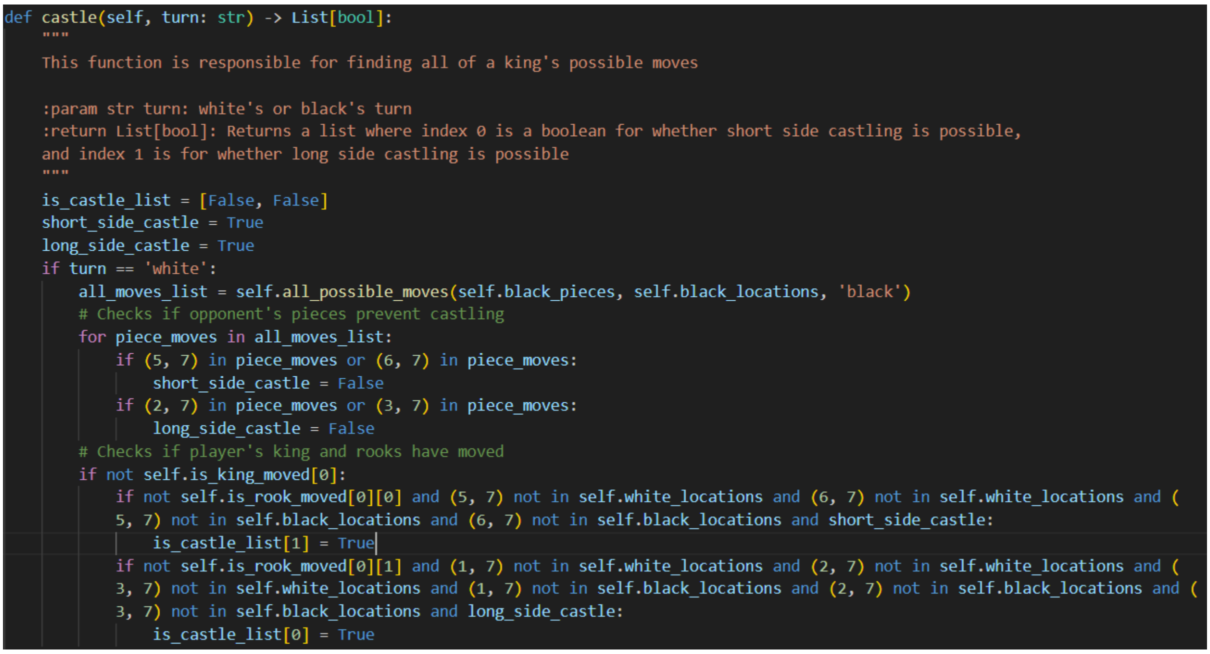
```
# Checks for castling and adds it to valid_moves
if piece == 'king':
    is_castle = self.castle(self.player_turn)
    if is_castle[0]:
        self.valid_moves.append((2, 7))
    if is_castle[1]:
        self.valid_moves.append((6, 7))
```

Checks for castling for black player:

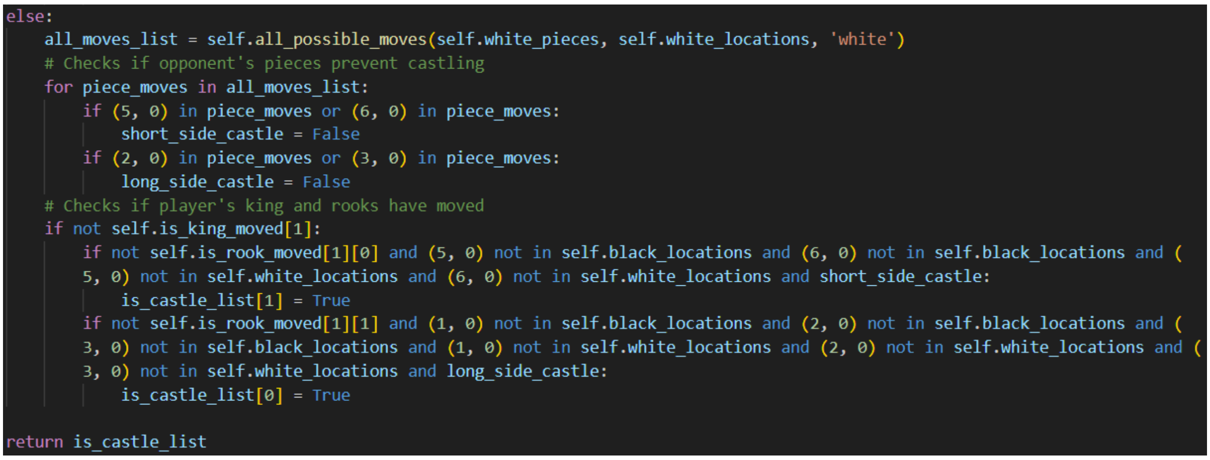


```
# Checks for castling and adds it to valid_moves
if piece == 'king':
    is_castle = self.castle(self.player_turn)
    if is_castle[0]:
        self.valid_moves.append((2, 0))
    if is_castle[1]:
        self.valid_moves.append((6, 0))
```

castle() function:

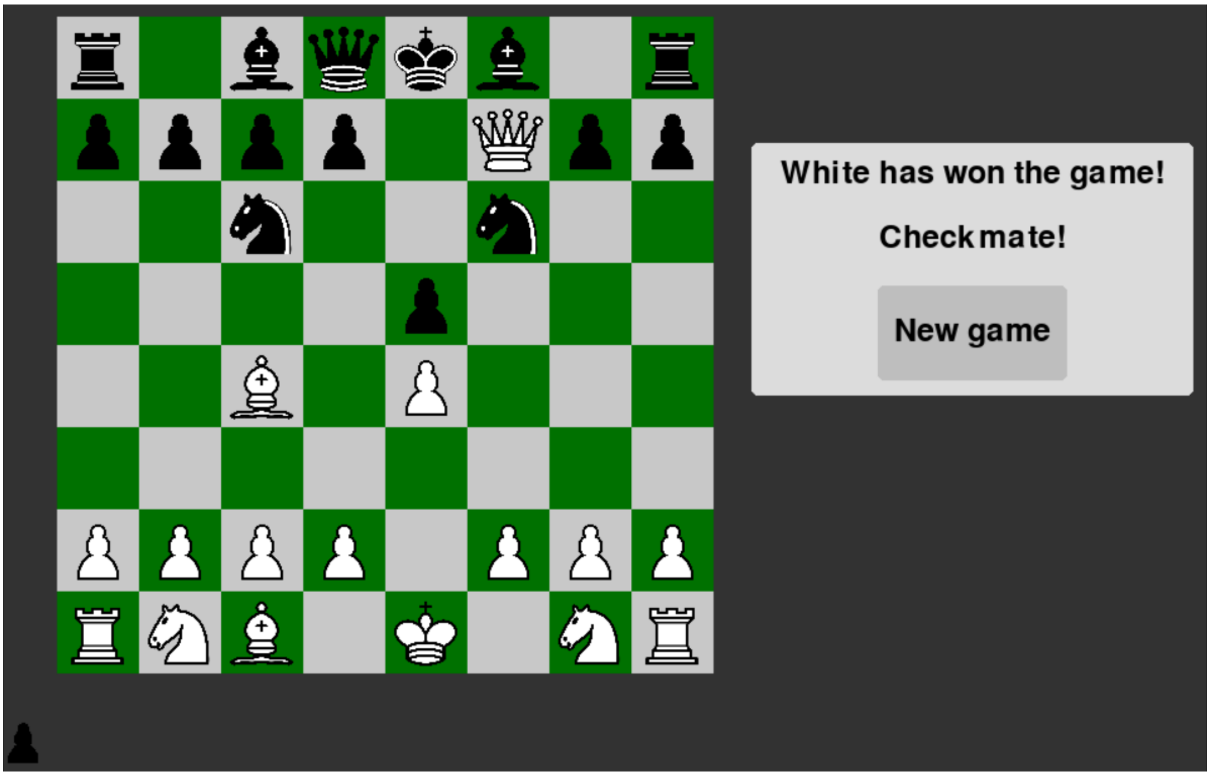


```
def castle(self, turn: str) -> List[bool]:
    """
    This function is responsible for finding all of a king's possible moves

    :param str turn: white's or black's turn
    :return List[bool]: Returns a list where index 0 is a boolean for whether short side castling is possible,
    and index 1 is for whether long side castling is possible
    """
    is_castle_list = [False, False]
    short_side_castle = True
    long_side_castle = True
    if turn == 'white':
        all_moves_list = self.all_possible_moves(self.black_pieces, self.black_locations, 'black')
        # Checks if opponent's pieces prevent castling
        for piece_moves in all_moves_list:
            if (5, 7) in piece_moves or (6, 7) in piece_moves:
                short_side_castle = False
            if (2, 7) in piece_moves or (3, 7) in piece_moves:
                long_side_castle = False
        # Checks if player's king and rooks have moved
        if not self.is_king_moved[0]:
            if not self.is_rook_moved[0][0] and (5, 7) not in self.white_locations and (6, 7) not in self.white_locations and (
            5, 7) not in self.black_locations and (6, 7) not in self.black_locations and short_side_castle:
                is_castle_list[1] = True
            if not self.is_rook_moved[0][1] and (1, 7) not in self.white_locations and (2, 7) not in self.white_locations and (
            3, 7) not in self.white_locations and (1, 7) not in self.black_locations and (2, 7) not in self.black_locations and (
            3, 7) not in self.black_locations and long_side_castle:
                is_castle_list[0] = True
```



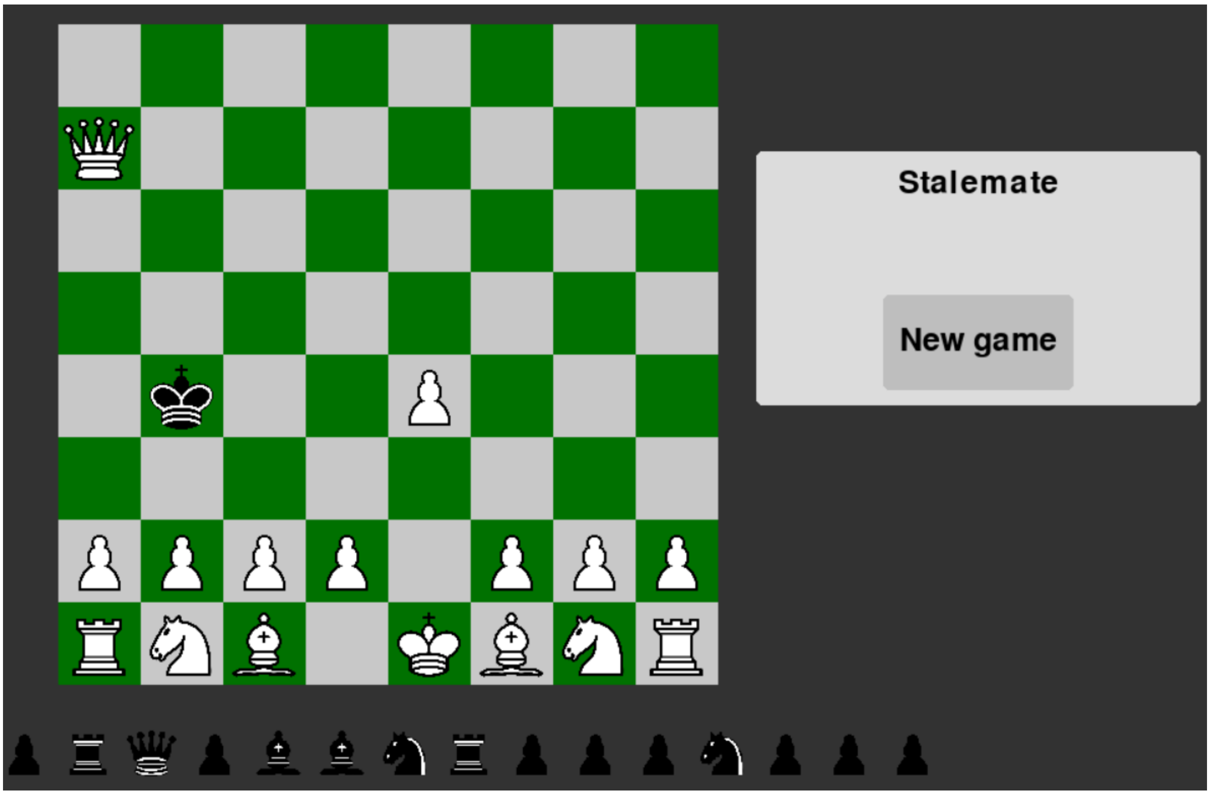
```
else:
    all_moves_list = self.all_possible_moves(self.white_pieces, self.white_locations, 'white')
    # Checks if opponent's pieces prevent castling
    for piece_moves in all_moves_list:
        if (5, 0) in piece_moves or (6, 0) in piece_moves:
            short_side_castle = False
        if (2, 0) in piece_moves or (3, 0) in piece_moves:
            long_side_castle = False
    # Checks if player's king and rooks have moved
    if not self.is_king_moved[1]:
        if not self.is_rook_moved[1][0] and (5, 0) not in self.black_locations and (6, 0) not in self.black_locations and (
        5, 0) not in self.white_locations and (6, 0) not in self.white_locations and short_side_castle:
            is_castle_list[1] = True
        if not self.is_rook_moved[1][1] and (1, 0) not in self.black_locations and (2, 0) not in self.black_locations and (
        3, 0) not in self.black_locations and (1, 0) not in self.white_locations and (2, 0) not in self.white_locations and (
        3, 0) not in self.white_locations and long_side_castle:
            is_castle_list[0] = True

return is_castle_list
```

Checkmate and Stalemate:



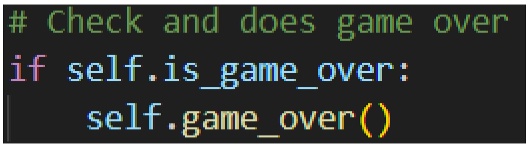
White has won the game!

Checkmate!



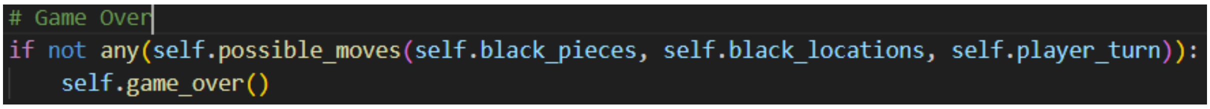
Stalemate

Calling the function game_over():



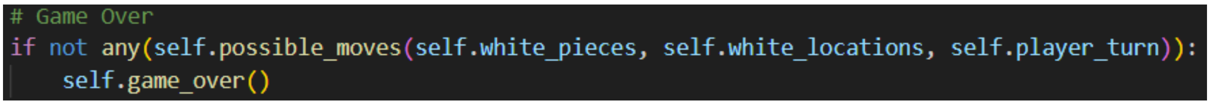
```python
# Check and does game over
if self.is_game_over:
    self.game_over()
```

White player wins:



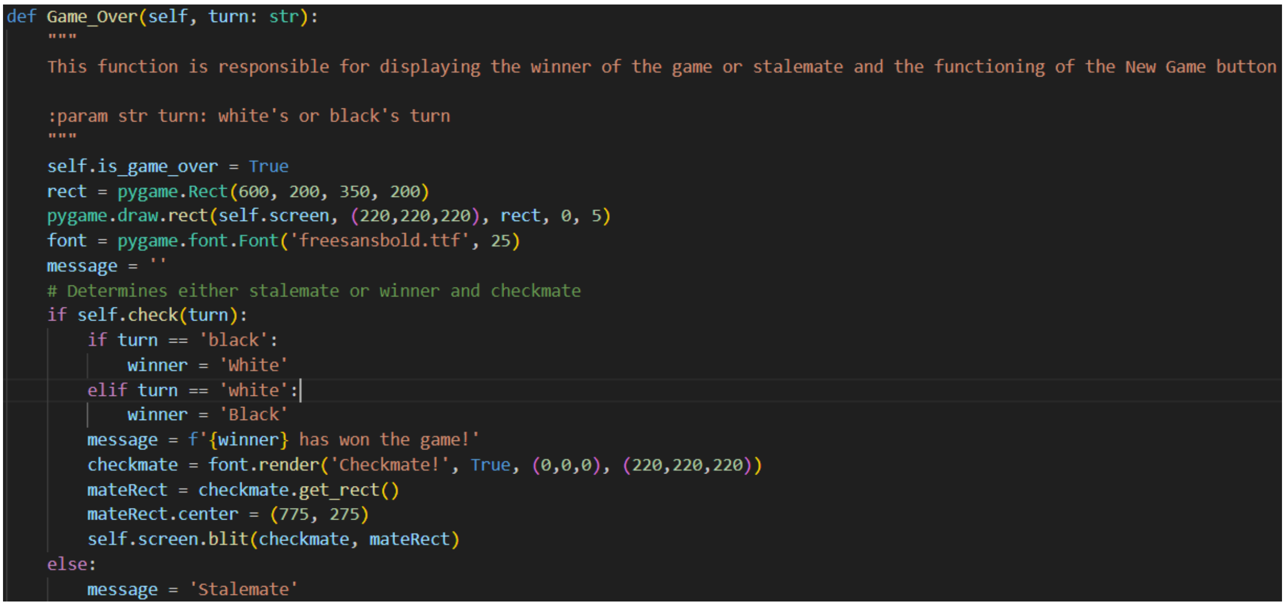
```python
# Game Over
if not any(self.possible_moves(self.black_pieces, self.black_locations, self.player_turn)):
    self.game_over()
```

Black player wins:

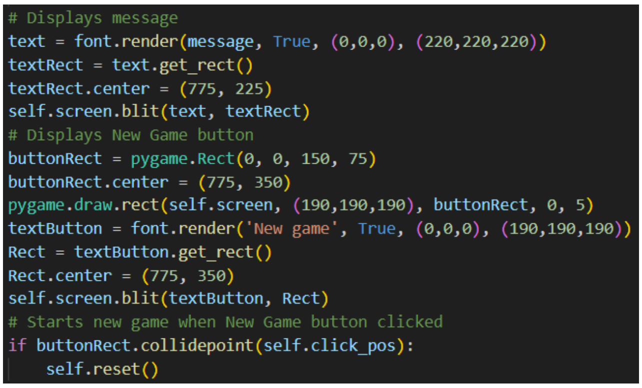


```python
# Game Over
if not any(self.possible_moves(self.white_pieces, self.white_locations, self.player_turn)):
    self.game_over()
```

game_over() function:

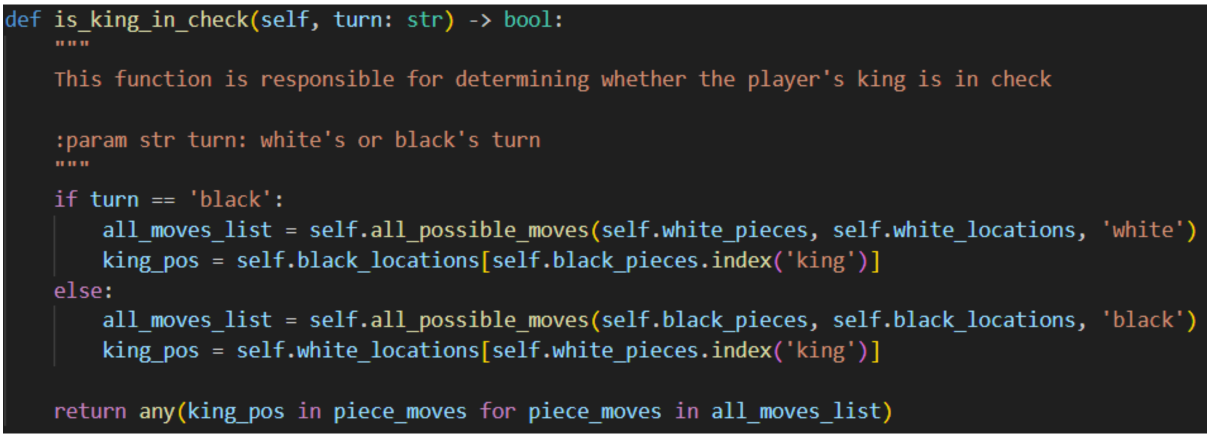


```python
def Game_Over(self, turn: str):
    """
    This function is responsible for displaying the winner of the game or stalemate and the functioning of the New Game button

    :param str turn: white's or black's turn
    """
    self.is_game_over = True
    rect = pygame.Rect(600, 200, 350, 200)
    pygame.draw.rect(self.screen, (220,220,220), rect, 0, 5)
    font = pygame.font.Font('freesansbold.ttf', 25)
    message = ''
    # Determines either stalemate or winner and checkmate
    if self.check(turn):
        if turn == 'black':
            winner = 'White'
        elif turn == 'white':
            winner = 'Black'
        message = f'{winner} has won the game!'
        checkmate = font.render('Checkmate!', True, (0,0,0), (220,220,220))
        mateRect = checkmate.get_rect()
        mateRect.center = (775, 275)
        self.screen.blit(checkmate, mateRect)
    else:
        message = 'Stalemate'
```



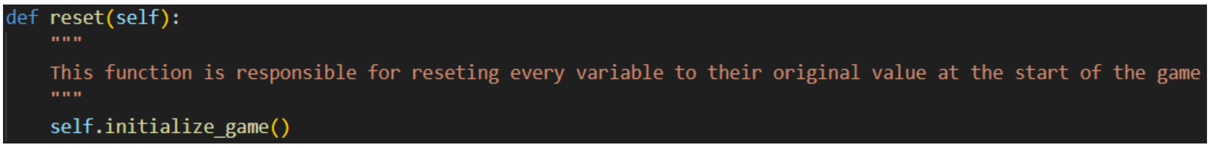
```python
# Displays message
text = font.render(message, True, (0,0,0), (220,220,220))
textRect = text.get_rect()
textRect.center = (775, 225)
self.screen.blit(text, textRect)
# Displays New Game button
buttonRect = pygame.Rect(0, 0, 150, 75)
buttonRect.center = (775, 350)
pygame.draw.rect(self.screen, (190,190,190), buttonRect, 0, 5)
textButton = font.render('New game', True, (0,0,0), (190,190,190))
Rect = textButton.get_rect()
Rect.center = (775, 350)
self.screen.blit(textButton, Rect)
# Starts new game when New Game button clicked
if buttonRect.collidepoint(self.click_pos):
    self.reset()
```

check() function:



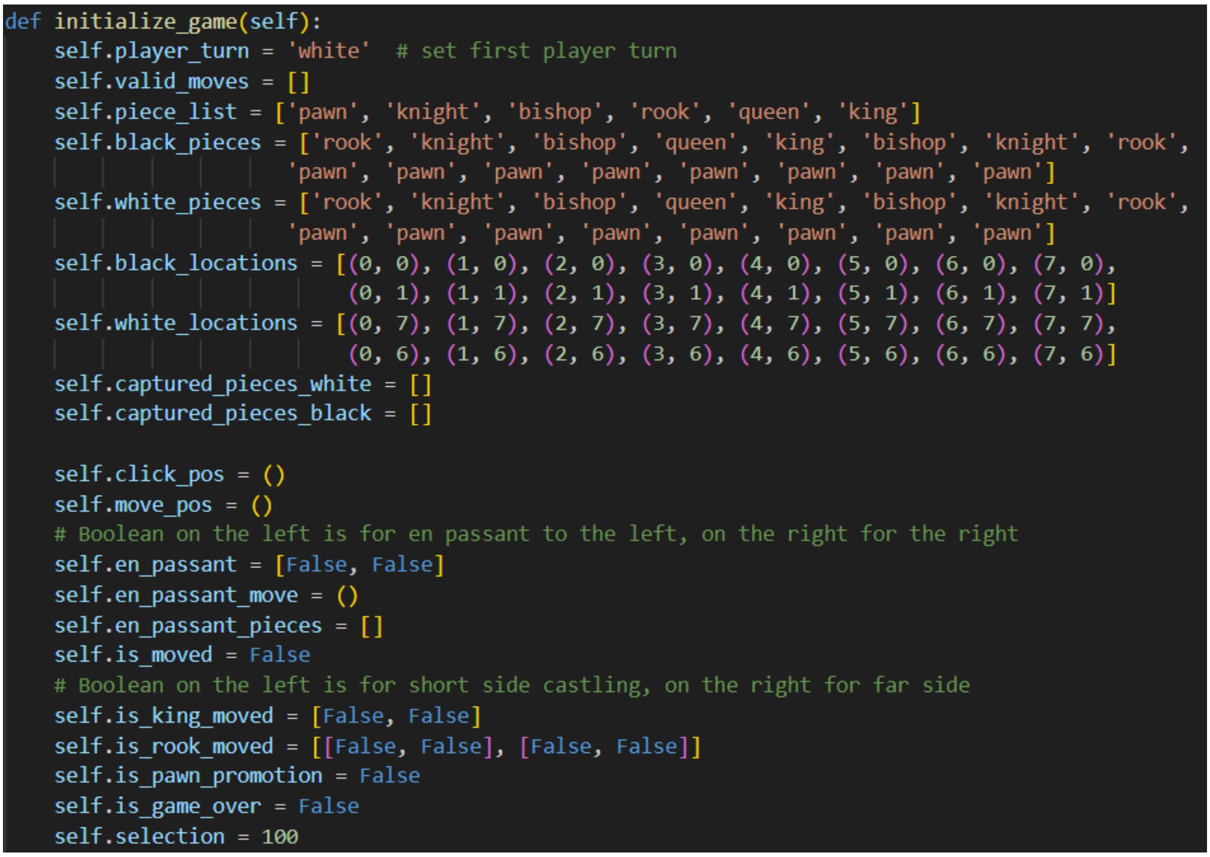
```python
def is_king_in_check(self, turn: str) -> bool:
    """
    This function is responsible for determining whether the player's king is in check

    :param str turn: white's or black's turn
    """
    if turn == 'black':
        all_moves_list = self.all_possible_moves(self.white_pieces, self.white_locations, 'white')
        king_pos = self.black_locations[self.black_pieces.index('king')]
    else:
        all_moves_list = self.all_possible_moves(self.black_pieces, self.black_locations, 'black')
        king_pos = self.white_locations[self.white_pieces.index('king')]

    return any(king_pos in piece_moves for piece_moves in all_moves_list)
```

reset() function:



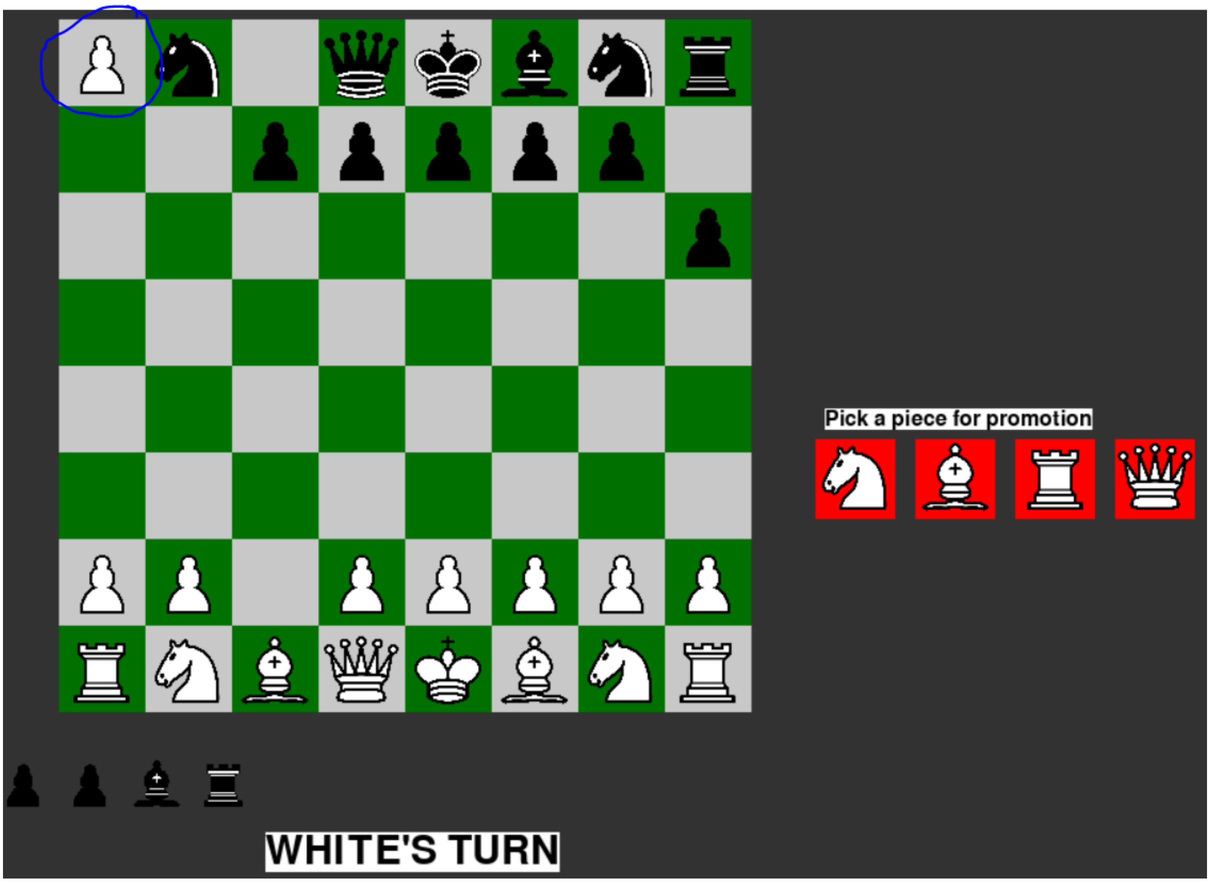
```python
def reset(self):
    """
    This function is responsible for reseting every variable to their original value at the start of the game
    """
    self.initialize_game()
```

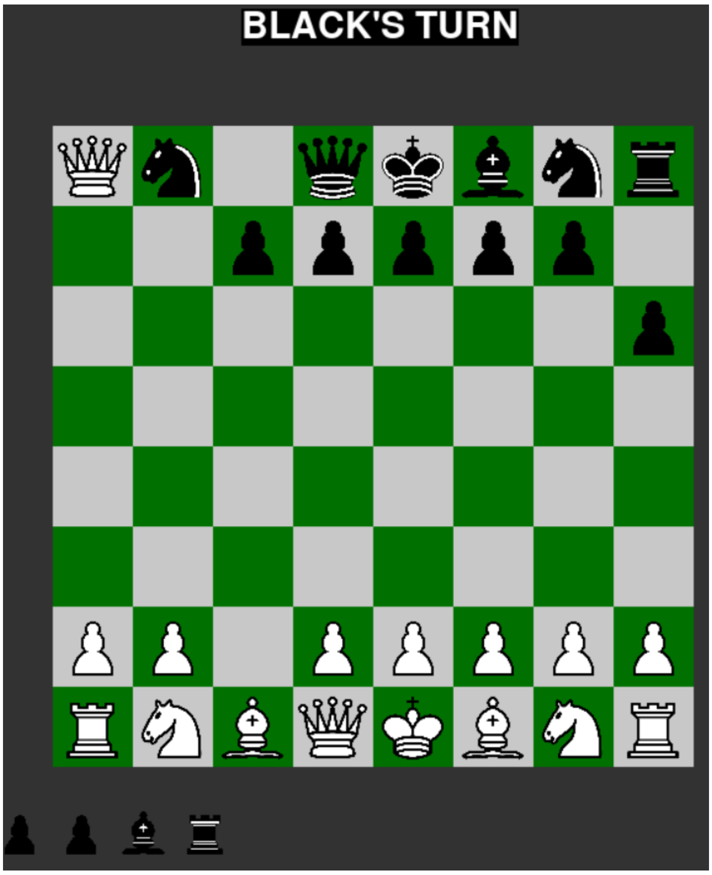
initialize_game() function:



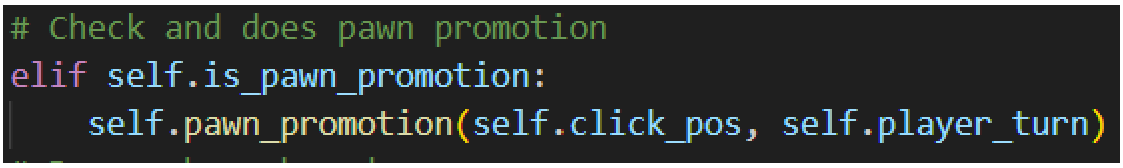
```python
def initialize_game(self):
    self.player_turn = 'white'  # set first player turn
    self.valid_moves = []
    self.piece_list = ['pawn', 'knight', 'bishop', 'rook', 'queen', 'king']
    self.black_pieces = ['rook', 'knight', 'bishop', 'queen', 'king', 'bishop', 'knight', 'rook',
                       'pawn', 'pawn', 'pawn', 'pawn', 'pawn', 'pawn', 'pawn', 'pawn']
    self.white_pieces = ['rook', 'knight', 'bishop', 'queen', 'king', 'bishop', 'knight', 'rook',
                       'pawn', 'pawn', 'pawn', 'pawn', 'pawn', 'pawn', 'pawn', 'pawn']
    self.black_locations = [(0, 0), (1, 0), (2, 0), (3, 0), (4, 0), (5, 0), (6, 0), (7, 0),
                            (0, 1), (1, 1), (2, 1), (3, 1), (4, 1), (5, 1), (6, 1), (7, 1)]
    self.white_locations = [(0, 7), (1, 7), (2, 7), (3, 7), (4, 7), (5, 7), (6, 7), (7, 7),
                            (0, 6), (1, 6), (2, 6), (3, 6), (4, 6), (5, 6), (6, 6), (7, 6)]
    self.captured_pieces_white = []
    self.captured_pieces_black = []

    self.click_pos = ()
    self.move_pos = ()
    # Boolean on the left is for en passant to the left, on the right for the right
    self.en_passant = [False, False]
    self.en_passant_move = ()
    self.en_passant_pieces = []
    self.is_moved = False
    # Boolean on the left is for short side castling, on the right for far side
    self.is_king_moved = [False, False]
    self.is_rook_moved = [[False, False], [False, False]]
    self.is_pawn_promotion = False
    self.is_game_over = False
    self.selection = 100
```

Pawn promotion (before and after):

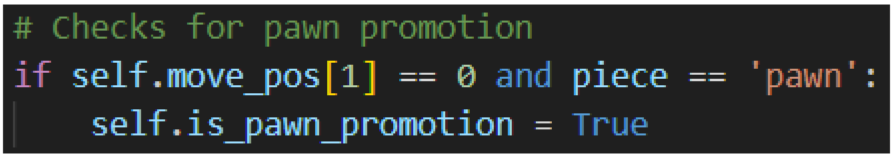




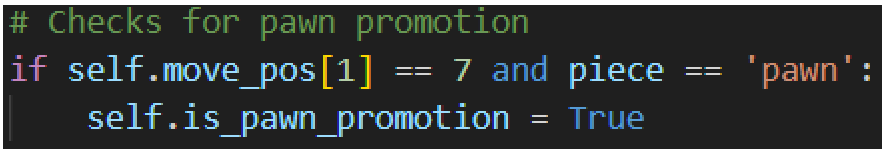
Calling the function:



```
# Check and does pawn promotion
elif self.is_pawn_promotion:
    self.pawn_promotion(self.click_pos, self.player_turn)
```

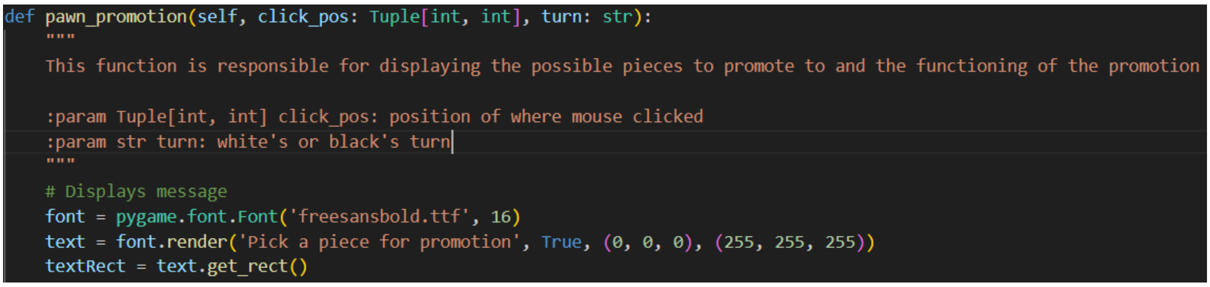


```
# Checks for pawn promotion
if self.move_pos[1] == 0 and piece == 'pawn':
    self.is_pawn_promotion = True
```

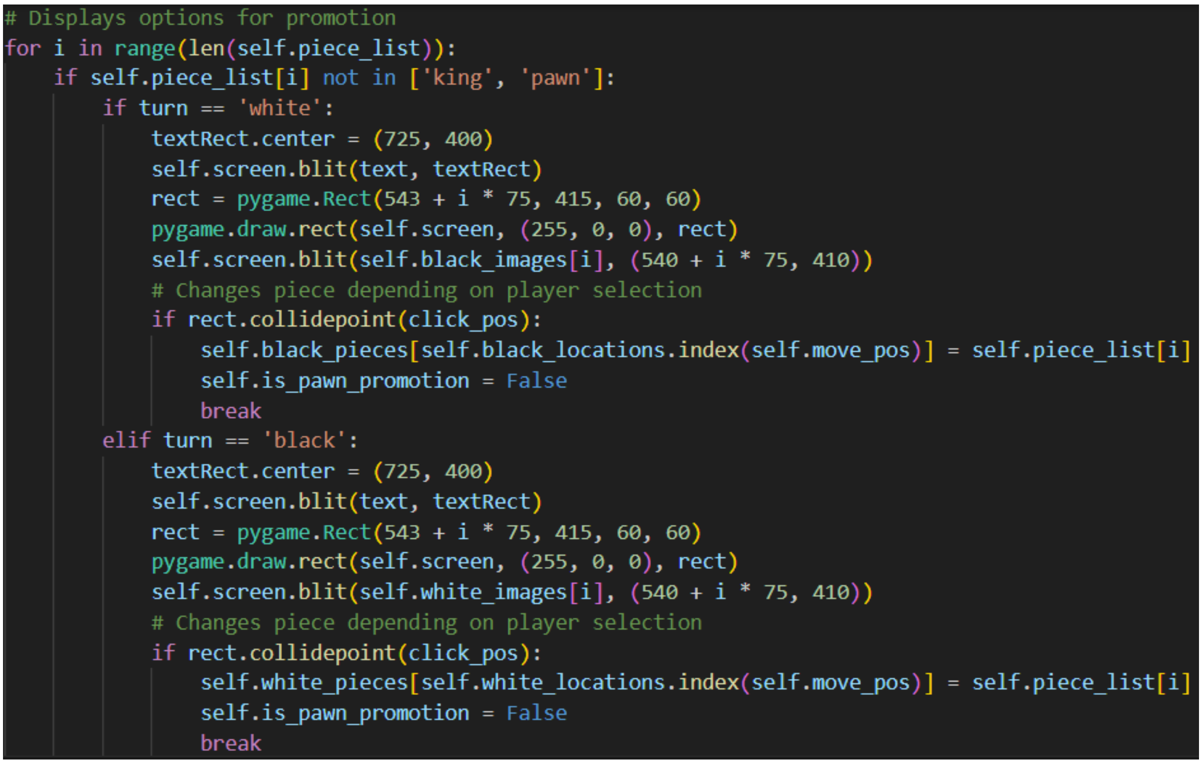


```
# Checks for pawn promotion
if self.move_pos[1] == 7 and piece == 'pawn':
    self.is_pawn_promotion = True
```

pawn_promotion() function:

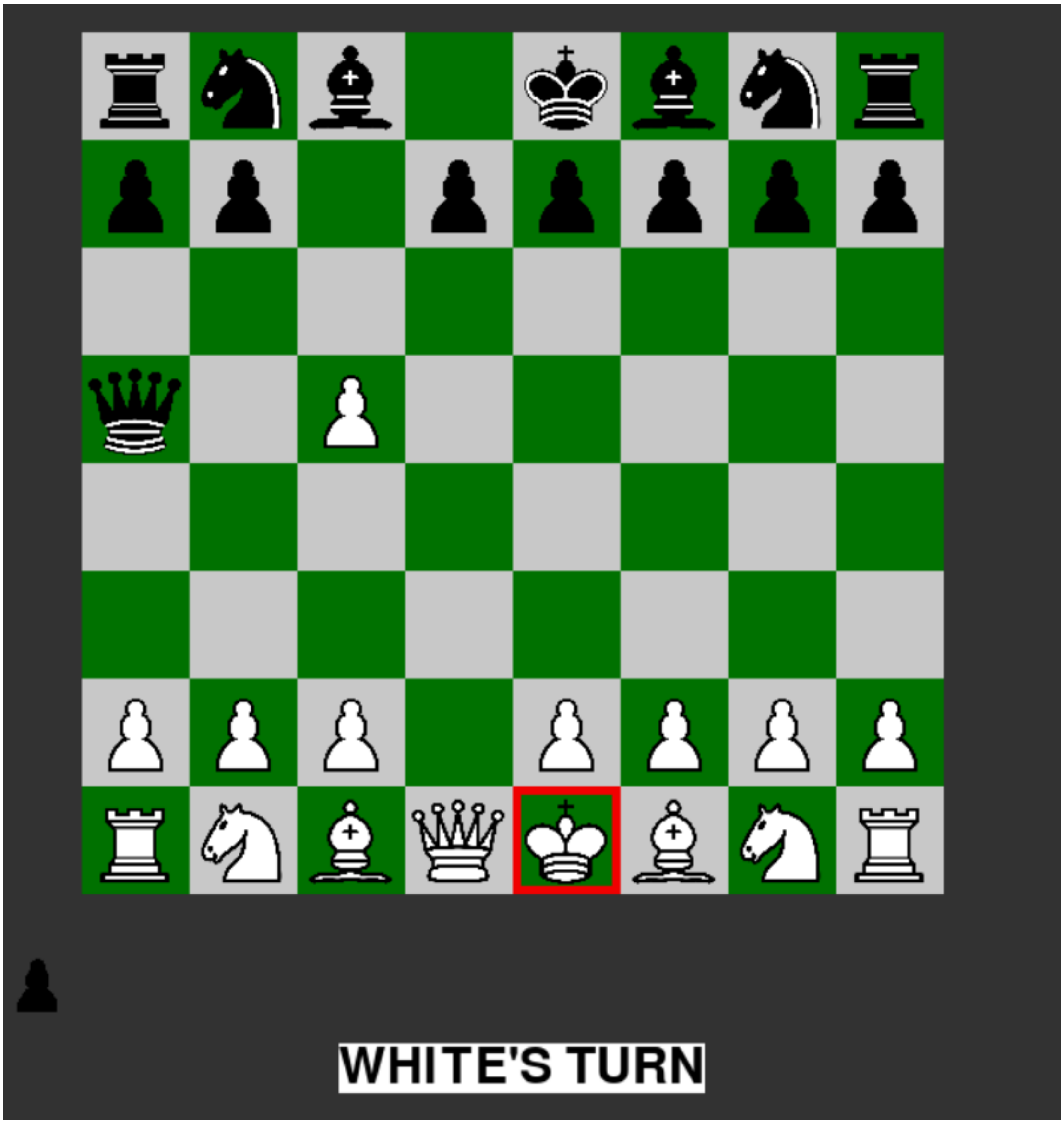


```
def pawn_promotion(self, click_pos: Tuple[int, int], turn: str):
    """
    This function is responsible for displaying the possible pieces to promote to and the functioning of the promotion

    :param Tuple[int, int] click_pos: position of where mouse clicked
    :param str turn: white's or black's turn
    """
    # Displays message
    font = pygame.font.Font('freesansbold.ttf', 16)
    text = font.render('Pick a piece for promotion', True, (0, 0, 0), (255, 255, 255))
    textRect = text.get_rect()
```

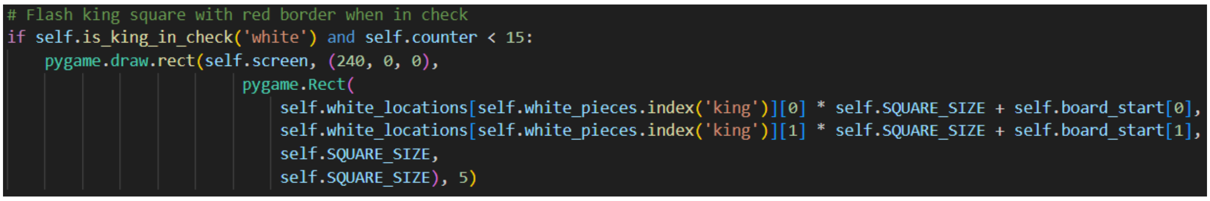


```
# Displays options for promotion
for i in range(len(self.piece_list)):
    if self.piece_list[i] not in ['king', 'pawn']:
        if turn == 'white':
            textRect.center = (725, 400)
            self.screen.blit(text, textRect)
            rect = pygame.Rect(543 + i * 75, 415, 60, 60)
            pygame.draw.rect(self.screen, (255, 0, 0), rect)
            self.screen.blit(self.black_images[i], (540 + i * 75, 410))
            # Changes piece depending on player selection
            if rect.collidepoint(click_pos):
                self.black_pieces[self.black_locations.index(self.move_pos)] = self.piece_list[i]
                self.is_pawn_promotion = False
                break
        elif turn == 'black':
            textRect.center = (725, 400)
            self.screen.blit(text, textRect)
            rect = pygame.Rect(543 + i * 75, 415, 60, 60)
            pygame.draw.rect(self.screen, (255, 0, 0), rect)
            self.screen.blit(self.white_images[i], (540 + i * 75, 410))
            # Changes piece depending on player selection
            if rect.collidepoint(click_pos):
                self.white_pieces[self.white_locations.index(self.move_pos)] = self.piece_list[i]
                self.is_pawn_promotion = False
                break
```

Flashing king square when in check:

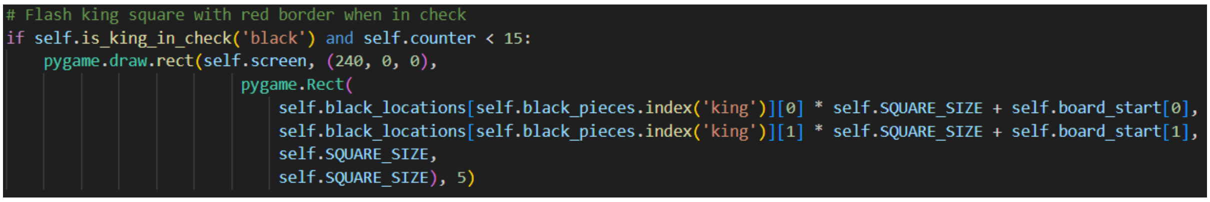


White player in check:



```
# Flash king square with red border when in check
if self.is_king_in_check('white') and self.counter < 15:
    pygame.draw.rect(self.screen, (240, 0, 0),
                     pygame.Rect(
                         self.white_locations[self.white_pieces.index('king')][0] * self.SQUARE_SIZE + self.board_start[0],
                         self.white_locations[self.white_pieces.index('king')][1] * self.SQUARE_SIZE + self.board_start[1],
                         self.SQUARE_SIZE,
                         self.SQUARE_SIZE), 5)
```

Black player in check:



```
# Flash king square with red border when in check
if self.is_king_in_check('black') and self.counter < 15:
    pygame.draw.rect(self.screen, (240, 0, 0),
                     pygame.Rect(
                         self.black_locations[self.black_pieces.index('king')][0] * self.SQUARE_SIZE + self.board_start[0],
                         self.black_locations[self.black_pieces.index('king')][1] * self.SQUARE_SIZE + self.board_start[1],
                         self.SQUARE_SIZE,
                         self.SQUARE_SIZE), 5)
```