# Python Chess Project Documentation

## Overview

This project implements a chess game using Python and Pygame. It features a full chessboard, piece movement, move validation, special moves (castling, en passant, promotion), and a graphical interface.

---

## Class and Function Explanations

### 1. `Board`

- **Purpose:** Manages the chessboard state, piece placement, move logic, and rules.
- **Key Methods:**
  - `__init__`: Initializes the board, creates squares, and adds pieces.
  - `move(piece, move, testing=False)`: Moves a piece, handles captures, en passant, promotion, and castling.
  - `valid_move(piece, move)`: Checks if a move is valid for a piece.
  - `check_promotion(piece, final)`: Promotes a pawn to a queen if it reaches the last rank.
  - `castling(initial, final)`: Checks if a move is a castling move.
  - `set_true_en_passant(piece)`: Sets the en passant flag for pawns.
  - `in_check(piece, move)`: Checks if a move puts the king in check.
  - `calc_moves(piece, row, col, bool=True)`: Calculates all valid moves for a piece at a given position.
  - `_create()`: Initializes the board squares.
  - `_add_pieces(color)`: Adds all pieces for a given color.

### 2. `Square`

- **Purpose:** Represents a single square on the chessboard.
- **Key Methods:**
  - `__init__(row, col, piece=None)`: Initializes a square with position and optional piece.
  - `has_piece()`, `isempty()`: Checks if a square has a piece.
  - `has_team_piece(color)`, `has_enemy_piece(color)`: Checks for team/enemy pieces.
  - `isempty_or_enemy(color)`: Checks if square is empty or has an enemy piece.
  - `in_range(*args)`: Static method to check if coordinates are within board bounds.

### 3. `Piece` and Subclasses (`Pawn`, `Knight`, `Bishop`, `Rook`, `Queen`, `King`)

- **Purpose:** Base class for all chess pieces, with subclasses for each type.

- **Key Methods:**
  - `__init__`: Initializes piece attributes (name, color, value, etc.).
  - `set_texture(size=80)`: Sets the image path for the piece.
  - `add_move(move)`, `clear_moves()`: Manage the piece's possible moves.
- **Pawn**: Adds direction and en passant attributes.
- **King**: Adds references to left and right rooks for castling.

### 4. `Move`

- **Purpose:** Represents a move from one square to another.
- **Key Methods:**
  - `__init__(initial, final)`: Stores initial and final squares.
  - `__str__`, `__eq__`: String representation and equality check.

### 5. `Sound`

- **Purpose:** Handles sound effects for moves and captures.
- **Key Methods:**
  - `__init__(path)`: Loads a sound file.
  - `play()`: Plays the sound.

### 6. `Theme`, `Color`, `Config`

- **Purpose:** Manage board themes, colors, and configuration (including sounds and fonts).

### 7. `Dragger`

- **Purpose:** Handles piece dragging logic in the GUI.
- **Key Methods:**
  - `update_blit(surface)`: Draws the dragged piece.
  - `update_mouse(pos)`, `save_initial(pos)`: Track mouse position and initial drag.
  - `drag_piece(piece)`, `undrag_piece()`: Start and stop dragging.

### 8. `Game`

- **Purpose:** Manages the game state, turn logic, and rendering.
- **Key Methods:**
  - `show_bg(surface)`, `show_pieces(surface)`, `show_moves(surface)`, `show_last_move(surface)`, `show_hover(surface)`: Render various board elements.
  - `next_turn()`: Switches the active player.
  - `set_hover(row, col)`: Sets the hovered square.
  - `change_theme()`: Changes the board theme.
  - `play_sound(captured=False)`: Plays move or capture sound.

    - `reset()`: Resets the game.

**9. `Main`**

- **Purpose:** Main application loop, handles events and user interaction.

---

## Data Structures and Algorithms (DSA) Usage

### Data Structures

- **2D List (Matrix):**
    - `self.squares` in `Board` is a 2D list representing the chessboard (8x8 grid).
    - Each element is a `Square` object, which may contain a `Piece`.
- **Classes and Objects:**
    - OOP is used extensively to model pieces, squares, moves, and game state.
- **Lists:**
    - Each `Piece` has a `moves` list to store possible moves.
    - Themes are stored in a list in `Config`.

### Algorithms

- **Move Generation:**
    - For each piece, possible moves are generated using loops and conditionals based on chess rules.
    - For sliding pieces (bishop, rook, queen), loops incrementally check each direction until blocked.
- **Move Validation:**
    - The `in_check` method uses deep copies of the board and pieces to simulate moves and check if the king is in check.
    - This is a form of backtracking: simulate a move, check for validity, then revert.
- **Special Moves:**
    - **Castling:** Checks if the king and rook have moved, and if squares between them are empty.
    - **En Passant:** Uses flags and board state to determine if en passant is possible.
    - **Promotion:** Checks if a pawn reaches the last rank and replaces it with a queen.
- **Coordinate Mapping:**
    - The `Square` class uses a dictionary to map column indices to chess notation (a-h).

**DSA Concepts Used**

- **Object-Oriented Programming:** Encapsulation of board, pieces, moves, and game logic.
- **Matrix Traversal:** Used for board representation and move calculation.
- **Simulation/Backtracking:** Used in move validation to check for checks.
- **State Management:** Each piece and square maintains its own state (moved, en passant, etc.).
- **Event Handling:** The main loop processes user input and updates the game state accordingly.

---

## Summary

This chess project demonstrates the use of classic data structures (lists, matrices, objects) and algorithms (move generation, validation, simulation) to implement the rules and logic of chess. The code is modular, with each class responsible for a specific aspect of the game, making it easy to maintain and extend.