

Python Chess Game - ICT323 Project Report



Team Number: 71-75

Course: ICT323 - Introduction to Python Programming

Bells University of Technology

Project Title: Chess Game Development

TABLE OF CONTENTS

1. Project Outline

2. Implementation

- Game Construction
- Key Highlights
- Algorithms and Logic

3. Challenges and Solutions

4. Potential Enhancements

5. Conclusion

PROJECT OVERVIEW

The Python Chess Game undertaking, created for the ICT323 seminar on Python Programming Improvement, presents a utilitarian two-player chess game. The essential point of this venture was to execute a completely intelligent chess game with a rich arrangement of highlights, involving Python and the Pygame library for graphical delivering and occasion taking care of. The game complies with the authority chess rules and supports generally standard and unique moves, including castling, en passant, and pawn advancement.

A huge piece of the improvement zeroed in on making a natural, simple to-explore UI (UI) while guaranteeing that the center chess mechanics worked accurately. The game permits players to control either the white or dark pieces, and the chessboard is progressively refreshed in light of player input. Moreover, high level chess rules like piece development approval, check, checkmate location, and unique moves were carried out. The task likewise expected to make an outwardly captivating encounter by using Pygame's strong graphical capacities.

The undertaking was created in Python because of the language's clarity and broad libraries like Pygame, which gives devices to taking

care of graphical result and client input. With an emphasis on great programming plan standards, like measured quality and meaningfulness, the codebase was coordinated into independent modules that oversee various parts of the game, including piece development, unique move rationale, and game state the executives.

IMPLEMENTATION

2.1 Game Construction

The game was organized into different modules for lucidity and to advance viability. These modules are intended to deal with various parts of the game freely, making it more straightforward to oversee and scale the codebase. The fundamental parts of the game design are illustrated beneath:

- **Main File:** This record deals with the general game circle, occasion taking care of, and delivering of the chessboard. It controls the progression of the game, refreshes the game state, and tunes in for player inputs (e.g., mouse snaps to move pieces).
- **Constants.py:** Characterizes the beginning design of the chessboard and the graphical resources (pictures of the chess pieces). This document contains constants like piece types (ruler, sovereign, and so forth) and introductory piece arrangements on the board.
- **Additions.py:** This file handles extraordinary moves, including castling, en passant, and pawn advancement. It likewise deals with the rationale for exchanging turns among players and guaranteeing all move rules are upheld.

2.2 Key Highlights

Graphical User Interface (GUI)

The graphical UI is one of the main parts of this undertaking. It was intended to guarantee that the game was both practical and outwardly engaging. Pygame was utilized to make the game's graphical parts, including the chessboard, pieces, and cooperation with the player. A portion of the critical highlights of the GUI include:

- **Dynamic Chessboard:** The chessboard switches back and forth among light and dull squares, outwardly addressing a standard chessboard. Each square is plainly checked, and player collaborations are worked with by featuring squares that are clicked or floated over.
- **Piece Display:** Chess pieces are addressed as great, scaled pictures, making it simple for players to recognize them outwardly.

The pictures are put away in the 'resources' registry and stacked powerfully toward the beginning of the game.

- **Visual Indicators:** To upgrade the ongoing interaction experience, visual pointers are utilized to show substantial moves, checks, and dangers. For example, squares that can be arrived at by the chose

piece are featured in green, making it simple for the player to pick their best course of action.

Chess Rule Enforcement

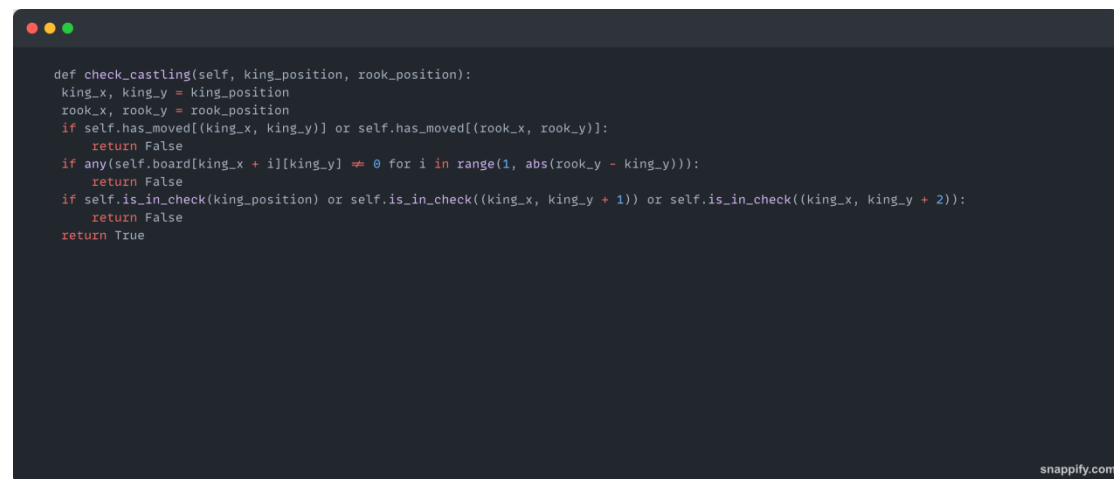
The game rigorously adheres to the guidelines of chess, guaranteeing that all pieces move as per their particular development rules. This incorporates standard developments like the sovereign moving slantingly, evenly, and in an upward direction, as well as unique principles, for example;

- **Turn-Based Gameplay:** Players substitute turns, with each turn permitting a player to move a solitary piece.
- **Piece Movement:** The game keeps players from taking unlawful actions, like moving a piece beyond its permitted range, or moving a part of a square involved by a united piece.
- **Check and Checkmate Detection:** The game incorporates rationale for distinguishing when a player's top dog is under tight restraints or checkmate. The game closures when one player's the best is checkmated.

Special Moves

Unique chess moves are a basic piece of the game. These include:

Castling: The game guarantees that castling can happen under unambiguous circumstances specifically, that the lord and rook included have not moved previously, and that there are no pieces between them. Furthermore, the ruler can't be within proper limits, nor might it at any point travel through a square enduring an onslaught.



```
def check_castling(self, king_position, rook_position):
    king_x, king_y = king_position
    rook_x, rook_y = rook_position
    if self.has_moved[(king_x, king_y)] or self.has_moved[(rook_x, rook_y)]:
        return False
    if any(self.board[king_x + i][king_y] != 0 for i in range(1, abs(rook_y - king_y))):
        return False
    if self.is_in_check(king_position) or self.is_in_check((king_x, king_y + 1)) or self.is_in_check((king_x, king_y + 2)):
        return False
    return True
```

snappify.com

En Passant: The en passant rule permits a pawn to catch a rival's pawn that has recently pushed two squares ahead from its beginning position. This extraordinary move is taken care of by distinguishing the place of the adversary's pawn and guaranteeing the circumstances are met for an en passant catch.


```
def check_en_passant(self, pawn, target_position):
    # If a pawn moves two squares forward, and it's beside an opponent's pawn
    if self.board[target_position[0]][target_position[1]] == self.opponent_color and self.board[pawn[0]][pawn[1]] == 'P':
        if abs(pawn[0] - target_position[0]) == 1 and abs(pawn[1] - target_position[1]) == 1:
            return True
    return False
```

snappify.com

Pawn Promotion: At the point when a pawn arrives at the rival's back rank, it is naturally advanced. The player can decide to elevate the pawn to a sovereign, rook, minister, or knight. The advancement rationale is carried out by checking in the event that a pawn has arrived at the last position and, provoking the player for their decision.

```
def promote_pawn(self, pawn_position):
    x, y = pawn_position
    if self.board[x][y] == 'P' and x == 0:
        promotion_choice = self.get_promotion_choice() # Prompt user for choice
        self.board[x][y] = promotion_choice
```

snappify.com

Game State Management

The game deals with its state by monitoring the board design, the ongoing player's turn, and whether any pieces are under wraps or checkmate. The game purposes a progression of checks and conditions to decide when the game ought to end, when another move is substantial, and when to tell players of a check or checkmate circumstance.

2.3 Algorithms and Logic

Move Calculation Algorithm

Each kind of piece has an extraordinary calculation for computing substantial moves. For instance, the **rook** gets in an orderly fashion across quite a few squares, evenly or in an upward direction.

This move estimation is finished by emphasizing through all potential squares toward every path until a piece is experienced.

```
def calculate_rook_moves(self, x, y):
    moves = []
    # Check vertical and horizontal directions for available moves
    for dx, dy in [(-1, 0), (1, 0), (0, -1), (0, 1)]:
        nx, ny = x, y
        while 0 <= nx + dx < 8 and 0 <= ny + dy < 8:
            nx, ny = nx + dx, ny + dy
            if self.board[nx][ny] == 0: # Empty space
                moves.append((nx, ny))
            elif self.board[nx][ny] == self.opponent_color:
                moves.append((nx, ny))
                break
            else: # Ally piece
                break
    return moves
```

snappify.com

This code computes all likely moves for a rook by emphasizing over every bearing (up, down, left, right) and checking for void squares or rival pieces.

CHALLENGES AND SOLUTIONS

Challenge 1: Implementing Special Moves

Executing exceptional chess moves, such as castling, en passant, and pawn advancement, introduced difficulties in dealing with the game state and guaranteeing that the circumstances for these moves were accurately checked. These moves include exceptional circumstances that vary from standard piece development.

- **Solution:**Careful testing and modularization helped separate the issue. Every unique move was embodied in its own capability, and assistant capabilities were utilized to check conditions like whether the ruler was within proper limits or whether pawns had recently moved two squares.

Challenge 2: Designing the User Interface

Planning an unmistakable and easy to understand point of interaction was another test. The connection point expected to show the chessboard and pieces as well as demonstrate substantial moves, checks, and game status continuously. Moreover, the player expected to connect with the game by choosing pieces and taking actions.

- **Solution:** The utilization of visual markers, like featuring legitimate squares, and giving clear criticism to exceptional moves like pawn advancement or castling, helped make the game more instinctive. Moreover, the game gave prompts when fundamental (e.g., when a pawn arrives at the back rank for advancement).

POTENTIAL ENHANCEMENTS

Advance AI Implementation

The undertaking could be improved by adding a man-made intelligence adversary, permitting players to play against the PC. This would include executing a calculation like **Minimax** to foresee moves and recreate canny ongoing interaction.

Undo Functionality

A fix element would permit players to return their last move, making it simpler for players to address botches. This would require saving game states at each turn and giving a component to reestablish the past state.

Performance Improvements

Advancing the move computations and the graphical delivering can work on the game's presentation, particularly in additional complicated situations with bigger sheets or longer games. Methods like move pruning and reserving could be utilized.

CONCLUSION

The Python Chess Game undertaking effectively shows the capacity to carry out an exhaustive chess game, from essential ongoing interaction mechanics to further developed highlights like exceptional moves. By using Python's Pygame, the task had the option to accomplish a harmony between usefulness, client experience, and execution. Future upgrades could incorporate simulated intelligence reconciliation, extra game modes, and further advancement. This venture gives serious areas of strength for venturing into more modern game improvement procedures.