Zhangfeiyang Hao, Kevin Liu and David Zhou

Prof. Robert Vincent

420-LCW-MS PROGRAMMING TECHNIQUES section 2

May 5th 2023

Chess Game: Final Project for Programming Techniques

A. Mini-Manual

1. Introduction

This is a two-player chess game that runs on Python. The game is played on an 8x8 board with pieces on squares named using the standard chess notation. The game follows the blitz rules in which there is no check or checkmate. Instead, the game ends when the king is captured.

2. Gameplay:

At the start of the game, each player has 16 pieces: one king, one queen, two bishops, two knights, two rooks and eight pawns. The game starts with white moving first. Each player takes turns moving one of their pieces, with the goal of capturing the opposing player's king.

3. Movement:

Official chess legal moves:

- King: Can move one square in any direction.
- Queen: Can move any number of squares diagonally, horizontally, or vertically.
- Rook: Can move any number of squares horizontally or vertically
- Bishop: Can move any number of squares diagonally.

- Knight: Moves in an L-shape pattern, two squares in one direction and one square perpendicular to that.
- Pawn: Moves forward one or two squares on its first move and then one square forward on each subsequent move.The pawn can capture diagonally one square ahead on either side.

For all pieces, they must move within the board, and cannot capture pieces of the same color.

4. How to move

To move a piece, input the starting and ending squares separated by a hyphen. For example moving a pawn from e2 to e4, would be denoted by “e2-e4”. If the move is legal, the new position will be drawn. Else, the board state will remain the same. The game ends whenever a king is captured.

Special move (Castling): Castling is exchanging the king and the rook by moving the king two squares right or three squares right, then putting the rook on the other side. The notation for short castle is e1-0 (for white) and e8-0 (for black). Long castle is denoted by e1-1 (for white) and e8-1 (for black).

5. Ending the game:

The game ends once either king is captured. Since this game implements blitz rules, there is no such thing as check or checkmate. If the king accidentally walks into a capturable square, the opposition may just take the king to end the game. Thus, there is no draw.

6. Bot:

There is an option to toggle on a bot as well as choosing the bot’s piece color. This is only a basic bot where it plays any legal move without consideration for the move’s strength.

B. Design Guide

Everything was coded in the Python coding language.

1. Pygame

In order to program the project, we used pygame which is a platform designed to create video games. We used it to create the GUI which consists of the screen (including the box, labels, and pieces), the buttons, the input box, and the instruction box. For all of them, we first create a font for the text, then we create a rectangle to put elements inside of it, draw the rectangle and write the text. For the input box, the program uses the method `UITextEntryLine` from pygame in order for the user to interact with the game. As for the buttons, the program checks whether the user's left mouse click is within the button's position and updates the bot's task accordingly.

Documentation:

https://www.pygame.org/docs/ref/display.html

https://www.pygame.org/docs/ref/event.html

https://www.pygame.org/docs/ref/rect.html

https://www.pygame.org/docs/ref/font.html

2. Determining turns and castling

For both situations we created global variables. For turns, we set its value to 'w' (white) initially since white is the first to move. Then we update this variable whenever a move gets played. For castling, one condition is to check whether the rooks or the king moved or not. To do that we created many global variables that are the respective tiles where kings and rooks are positioned initially and set them to True at the game's start. Since they are values of keys, the

program instead verifies whenever a tile becomes empty (‘-’) instead of checking if the piece itself moved or not and updates the variable’s boolean value.

3. Moving Pieces

We created a dictionary to represent the board where the keys are the tiles (files from a to h and rank from 1 to 8) and we assigned their values to ‘-’ for an empty tile and the piece’s name as a string otherwise.

Then, in order to verify whether a move is legal, we implemented functions that create a list for all possible legal moves of a piece and then we verify whether the user’s input lies in this legal moves list.

For pawns, rooks, bishops, kings, and queens, their legal moves are implemented by following the official rules. Since the program uses the official chess notation, it needs to convert the notation to x,y coordinates. For example, the white pawn in front of the white king would be at the tile ‘e4’ in chess notation. In order to be able to calculate the moves, we separate the rank and file and convert the rank in its Unicode value and add the necessary move increments. So, if we want a piece to go from e4 file to f4 file, we add 1 to the unicode of the file and reconvert it to letters in order to fit the chess notation. And since the rank is always a digit, we only need to transform it to int and make the adjustments.

- Pawns: Checks whether the tile in front of the pawn is empty, and appends accordingly. Then, if it is on the 2nd rank and it is white and 7th rank if it’s black, it can move two squares forward if that tile is empty. Since our board has x and y coordinates, the black pawns move downwards (-1 increment) and the white pawns upwards (+1 increment).

- Rooks: Append all horizontal and vertical squares if they are empty/occupied by a piece of opposite color.
- Bishops: Appends all diagonals square if they are empty/occupied by a piece of the opposite color
- Kings: Create a list with all its possible moves which are all tiles located 1 distance from its original position. Then, append those that are legal: not occupied by a piece of the same color and empty.
- Queens: Appends all the bishop's and rook's legal moves of that square since the queens' movement is those of a bishop + rook.

For the knight, it was more mathematical as we need it to move like the L-shape.

- Knight: Create a list with all the different offset for the knight which is one component must move 2 tiles and the other 1 (component as in x,y). Then we loop through every possible offset and convert them to actual board positions. Append moves that are legal: not occupied by a piece of the same color and empty.

For castling, the program has two helper functions (castle_kingside() and castle_queenside()) that verifies whether castling is legal or not. The program checks if the relevant piece moved or not (See b) 2.Determining turns and castling) and if the squares between the rook and king are empty. If all conditions are satisfied, the program appends the castling move (See a) 4.How to move) in the legal move list.

After every legal move, we call the function position_drawer() which draws the current state of the board to update our moves.

Documentation:

https://sakkpalota.hu/index.php/en/chess/rules#check

4. Win/Lose

Function that loops through every tile and appends the piece in a list. Whenever a king is missing, that means there is a winner and a loser. Then, the program indicates on the input box that there is a result which prevents the user from typing in the input box anymore. Call the function when the code runs.

5. Bot

We also created a bot that is able to play against a human opponent by letting it play a random move from all the possible moves on the board. It does it by looping through every square, checking the piece_type, and then appending all the legal moves in a bigger list. From this list, the program uses the python random library to randomly select a move to play.

C. Program organization

The chess game runs on a single python file called ‘chess.py’ which consists of all the essential codes for game play, a simple chess bot, the interactive interface, etc. In the ‘chess.py’ file, codes are divided into different sections according to their functioning in running the game.

1. Interface coding

For the interface features that will remain the same throughout the whole game (e.g. chess board, squares, pieces and input box), we define their format and draw them on the screen at the beginning of the file in the dictionary named playing_board. Each section has a comment indicating the interface that it designs.

2. Helper Functions

Following the interface coding, there are a few helper functions to avoid coding the same piece of code constantly which are getting the piece's type or color.

3. Gameplay coding

Following the helper functions are functions that determine the game's winner, legal castling and obtaining the legal move for a certain piece. These functions are named based on their functioning and most of them either read or modify the 'playing_board' dictionary.

4. Game status and visualization updating

Following the functions is a while loop inside of which the screen will constantly update its display, ensuring that the screen displays the correct visualization based on the user's moves.

For example, the interface of the buttons are re-defined and re-drawn constantly in the while loop so that visual effects on these surfaces update correctly.

5. Bot implementation

At the top of the file, we implemented a bot that plays the opposite side from the player and does simple moves to ensure that the gameplay is smooth. The bot is defined as a class at the beginning of the code and has basic attributes to satisfy users' choices. More functionings of the bot could be implemented by adding methods and attributes to the bot class.