NEA - [redacted]

[redacted]

CANDIDATE NUMBER: [redacted]

CHESS ENGINE WITH CHANGE IN LEVEL DIFFICULTIES DURING GAMEPLAY

Analysis

Problem Identification

The origins of chess can be traced back to the sixth century in India. It's a two player abstract strategy game which takes place on a 64 square board (an 8 x 8 grid) with six types of pieces which are able to traverse the board in different ways. It implements a turn based mechanic, meaning that players alternate moving pieces on the board. Chess is a game with "perfect information", which in game theory means that there is no hidden information from either player.

Chess is a classic board game which is accessible and fun for people of all ages, and many people have a chess elo which is a number indicating their skill level at the game. Your elo is determined by the number of players you win or lose to and their strengths.

According to the National Museums of Liverpool, it is estimated that a chess board has around 10^{40} legal (or allowed) combinations, which is why computer scientists have taken a keen interest in creating chess AIs/engines. Chess can be considered a "playground" for AI research.

- https://www.liverpoolmuseums.org.uk/stories/which-greater-number-of-atoms-universe-or-number-of-chess-moves

Chess engine, assess game positions and choose the optimum move, these systems make use of artificial intelligence and algorithms. The first engine was created in the 1950s, but in the 1990s these algorithms were able to compete with human players at the high levels. In 1997, IBM's computer, "Deep Blue", defeated the world chess champion at the time, Garry Kasparov, was the first time a computer had defeated a world chess champion.

Today people can play and train against many chess robots on websites such as chess.com or lichess.org.

My friend and classmate [redacted] is a casual chess player who has a chess elo of around 600 (which is a beginner's level). [redacted] is one of many chess.com users who uses their training robots to get better at the game. His main motivation to get better at chess is to consistently beat his father in a match, who is a strong player.

In late September 2022, [redacted] was telling me about his quest to beat his father at chess and said that he wished that he could lower the difficulty of the engine while he was playing the game, as he said that he struggles coming up with strategies towards the end of the game.

I am also an avid chess player and could not think of any websites or softwares which allow you to change the difficulty during gameplay, so I asked my other friend and classmate, [redacted], if he knew of any. He couldn't name any either, but said it would be a good idea if someone made one.

This is why for my project, I plan to create a chess engine where you can change the difficulty during gameplay.

Stakeholders

In chess theory, there are three main phases of a game; the opening, middlegame and endgame. Each phase of a chess game requires the player to exercise different skills, for example in the opening a player will have to memorise large amounts of theories and variations while in the middlegame a player will have to be more observant and manoeuvre pieces to exploit their opponent's weaknesses. Based on this, a player may be stronger at different phases of the game, and naturally people will be stronger in some areas than others.

- https://www.britannica.com/topic/chess/Development-of-theory

The target audience will be anyone who wishes to change the computer difficulty during gameplay, based on the phase of the game. It will be designed with beginner to intermediate players in mind, as they stand to gain more from this kind of focused practice during game phases than advanced players. Additionally, an advanced player won't have a need to toggle the difficulty as they will most likely be playing at a consistently high level.

My two main stakeholders will be [redacted] and [redacted], as they both like the idea of changing the level of difficulty during gameplay in order to train. They are both students at my sixth form and are my friends, so I am able to easily contact them for feedback.

- [redacted] is a beginner player with a rating of around 600. He finds that he plays his opening pretty confidently but begins to make silly mistakes during the middle and end of the game. He represents the lower level players of my game.

- [redacted] is an intermediate player with a rating of around 1050. He started playing in late 2020 being inspired by the popular Netflix series "The Queen's Gambit." [redacted] is reaching the stage of chess where he must start to diversify the openings he uses in order to beat players at his level. He would rather train against a computer than real people in order to experiment with openings without it negatively affecting his rating. [redacted] wants to be able to play with the computer at a low difficulty when he's playing with a new opening he's unfamiliar with, then raise the difficulty to his current level during the middle and end games. [redacted] represents the slightly higher level players of my game.

[redacted] and [redacted] will help shape my project so it can be used by the target audience.

Target Platform

I asked [redacted] and [redacted] whether they generally preferred to play chess on their laptops or mobile phones, and they both agreed on their laptops. Therefore my game will be designed for a desktop.

The project will be a piece of software where the user can play a game of chess obeying against a computer the classical rules, and change the difficulty at the beginning of the game and during gameplay.

It will be written in python, and the pygame library will be used to aid in creating a user interface.

Interview

I asked [redacted] and [redacted] some questions to aid me in understanding what features and needs my game must fulfil.

Questions

1. **What do you think are essential features of a chess game?**

[redacted]: “The ability to play as either colour, moving the pieces in accordance to the rules of the game and being able to checkmate to win the game.”

2. **What features do you like on other chess platforms?**

[redacted]: “I like having the ability to customise the board to fit my liking. I would personally prefer a brown or grey option to mimic the aesthetic of wooden or metallic.”

3. **How do you like to move pieces on a virtual chess board? (e.g. dragging a piece to the square, clicking a piece to a square, etc..)**

[redacted]: “I prefer clicking as it can help avoid potential issues with the dragging getting let go too early and causing a misplay that can cause a player to mess up unintentionally”

4. **How many levels of AI difficulty do you think is good for a chess program to have?**

[redacted]: “I would personally want around 6 levels of difficulty as it is wide range enough to train against but not to many difficulties to get confused”

5. **Are there any other stylistic choices that you wish to be a part of the game?**

[redacted]: “I want the background to be yellow because chess makes you think of yellow”

Reviewing Responses

Based on my responses, functionalities that I will include the ability to play as either colour, the ability to play and win by the rules of chess and the ability to change between six difficulty levels.

Aesthetic features will include the ability to change the board colour to brown or grey and a yellow background.

Research

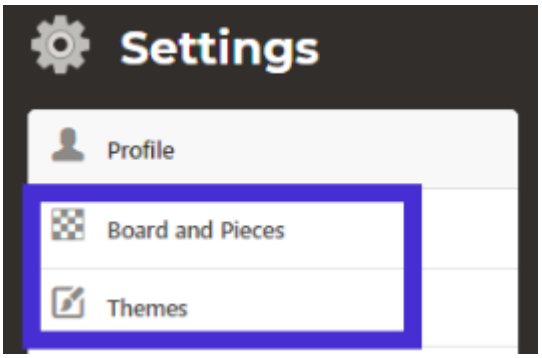
The two most popular chess websites are chess.com and lichess.org. So I did research on both of them.

Chess.com

Chess.com is an internet chess server created in 2007. On the website you have the option to play chess online against another player or go against one of their computers.

- chess.com logo

When a user makes an account, they are able to customise the sprites of the pieces and change the board colour to their preferences.



- Image of chess.com's settings allowing user to change style of board and pieces

By using a drop down menu, a user can easily select which styles they want.

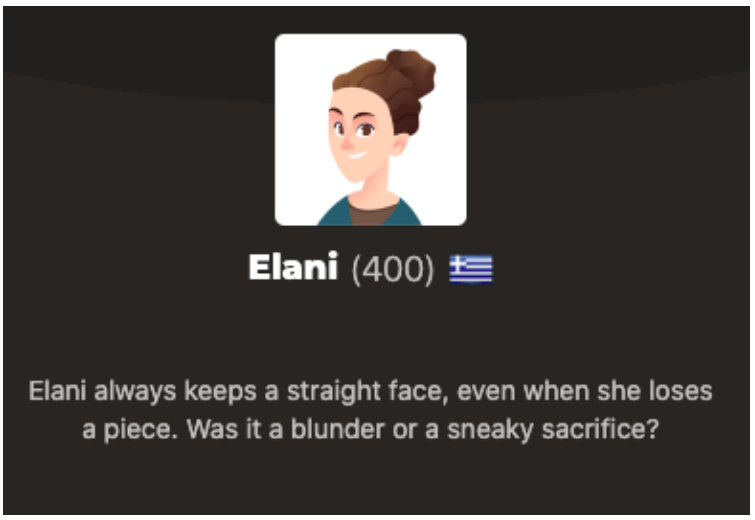
- Chess.com's drop down menu

The ability for a user to adjust these creates a more personalised experience when playing the game.

To add to their user centred experience, the player is able to choose whether they want to drag or click the pieces to their new positions.

Another usability feature that they implement is being able to visualise where each piece is able to be placed before placing it there. This is especially useful for novice players who are not confident with the rules of the game.

Some of the site's chess robots have different names, personalities, skill levels and playing styles. For example, the robot below is named Elani, who is at a beginner level and whose style is to sacrifice pieces to get a better position.



- Description above from chess.com

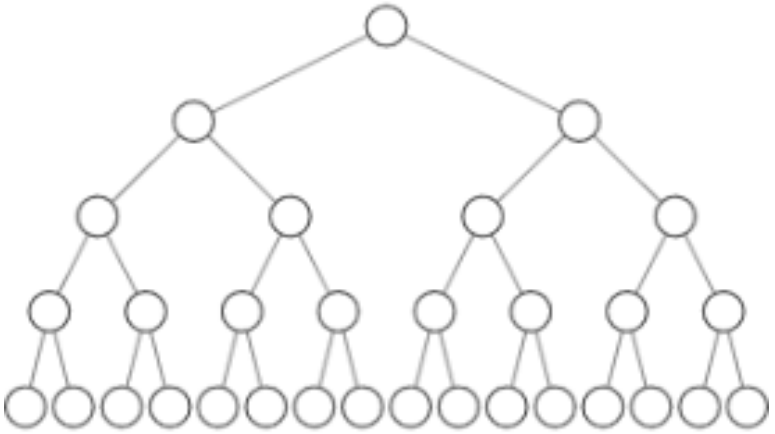
Giving each robot these traits makes it feel like the user is playing a real person, which is far more engaging then playing something that appears lifeless.

Each engine personality on the site uses the Komodo chess engine developed in 2010. Chess.com acquired this engine in 2018. The reasoning was because Komodo is able to run at different playing strengths, with different styles and opening books better than most other engines.

- Komodo Logo

(An opening book is a database of chess openings given to computer chess programs so they know what moves to play at the beginning of a game)

To make the decision of the best move to make, Komodo implements a Monte Carlo tree search. This method generates a search tree which is a way to visualise all of the possible moves after one particular move.



- Example of a search tree. The node at the top is known as the “root” node representing the first move in a sequence of moves. The example shows the root “branching” to two child nodes representing the next possible moves given the move of the root node has occurred. Those child nodes break down further showing more possible moves beyond that.

A Monte Carlo tree search is a heuristic search algorithm. This means that the algorithm will start by choosing random possible moves to begin its search, then assign a value to each move based on how promising that scenario looks. If a move has a higher value, then the algorithm will continue to search along that path while choosing random possibilities from that point. The tree will then “expand” on that side.

Monte Carlo is much more time efficient than other search algorithms, however it is not necessarily guaranteed to achieve the “best” result as it will tend to follow a certain path due to its heuristics.

Lichess

Lichess is the second largest chess server in the world after chess.com, launched in 2010. Similar to its rival platform, on lichess there is the option to play against another player of a computer.

- Lichess logo

When playing against a computer, lichess offers the eight levels of difficulties. These can be selected by clicking on the corresponding button as it is an easy and obvious way to change the difficulty. The different levels mean that the computer will "look" a different number of moves ahead.

- Image of computer strength level menu on lichess.org

Lichess uses the open source chess engine known as stockfish to run its computer. Stockfish was released in 2008 and is well known as one of the most powerful engines, having been a consistent medalist at the chess.com computer championships.

- chess.com

- Stockfish logo

Stockfish implements a minimax algorithm to decide on the best move. The minimax algorithm, similarly to the Monte Carlo algorithm, generates a tree data structure consisting of possible moves, and the possible moves following those. The main difference between the two algorithms is that the Monte Carlo algorithm uses heuristics while the minimax algorithm doesn't use heuristics.

Without the dependency on heuristics, the minimax algorithm can travel all the way down to each end node, evaluate the position, travel back up the tree to determine how good the move is, and repeat this for all possible moves. This means that a minimax algorithm is guaranteed to get the "best" move, but will take longer to find it.

To save time in the search, most minimax algorithms will use an additional algorithm known as alpha-beta pruning. This stops evaluating a move when at least one possibility has been found that proves the move to be worse than a previously examined move, hence terminating/cut-off the search for that move. This allows for a fairly accurate approximation, meaning that there is a low chance that the computer will miss the "best" move.

Stockfish takes this a step further by amending the algorithm to use a specific enhancement known as late move reduction. This orders each list of moves such that if a cut-off is going to occur, then the first few moves will be the ones which will cause it (meaning that the list of moves is ordered from

most to least likely). Saving time in this way allows for stockfish to search over 30 moves ahead in a given scenario.

Other Chess Engines

There are a multitude of other chess engines which exist today.

IBM Deep Blue was the first engine with the capabilities of beating a chess grandmaster. It also used an alpha-beta algorithm to decide on moves. To determine whether a move was good or not, the computer took in many general parameters such as material, king safety space in the centre of the board, and piece positioning. The evaluation function also contained an opening book of over 4,000 positions and 700,000 grandmaster games.

- One of the two cabinets of Deep Blue in its exhibit at the Computer History Museum in California

Google DeepMind is a British based artificial intelligence research and development laboratory. In 2017 they released a preprint introducing their chess AI AlphaZero. AlphaZero was originally coded with only the rules of chess, and by playing against itself and using machine learning it taught itself to play chess without an opening or endgame book. After four hours of doing this, AlphaZero was playing at a higher level than the greatest difficulty of stockfish. Machine learning greatly reduces the number of moves needed to be explored versus a minimax or Monte Carlo algorithm.

- Google DeepMind's logo

Adjustments Due to Research

My research led me to realise that there are some things I should implement in my project and that I have some limitations.

Implementations

It is clear to me that I will have to use some sort of search tree in order to find the "best" move. I decided that it was best to use a minimax algorithm with alpha-beta pruning like stockfish does. This is because it has a very low chance of missing the best move versus the Monte Carlo algorithm which has a slightly higher chance of missing it. Also, the Monte Carlo algorithm requires a heuristic to be added to each move, which is a value decided by the programmer (so me). I don't feel that I am knowledgeable enough in chess to assign such values to moves, and I do assign those values then it can lead to bias which can affect the gameplay. I can avoid this issue entirely by using a minimax algorithm.

On lichess, the computer can change the difficulty of looking more or less ahead. I like this way of changing the difficulty as it seems easier to implement and it complements the minimax algorithm nicely as I can easily tell it to stop searching after it reaches that number of moves. Therefore, there will be six levels of difficulty (as per [redacted]'s request), each looking one more move ahead.

To change the board colour and difficulty, the user can press buttons similar to lichess' interface. This provides an easy and simplistic way for my users to change the game settings. I may also implement chess.com's feature of visualising where pieces go before they're played which can be useful for less experienced players.

Limitations

Chess.com has several dozens of robots with their own playing styles, however due to time constraints I am not capable of creating 50 chess personalities which play uniquely and have names and backstories. This is not a requirement for my stakeholder so I will not include these.

The use of opening books seems to be a common theme in many chess engines. However, due to my limited knowledge of chess openings, I am incapable of generating entire sequences of moves for the computer to read from. It will also be extremely time consuming to learn enough about chess to do so.

Lastly, machine learning will be extremely difficult to implement due to the limited computational power of my laptop, which vastly differs from the many servers DeepMind had to train AlphaZero. Besides, I wouldn't know how to change the game difficulty if I trained a computer with machine learning

Solvable By Computational Solutions

This problem can be solved through a computational solution as my stakeholders wish to play chess against a computer in order to train. Playing chess against a person can be done over a physical board, however training with another person requires someone to be available to play against you and can be demoralising if you lose. If you play against a computer, then you can play whenever you want and will not feel as hurt if you lose.

Thinking Abstractly

Some details of a real life chess game are not needed when playing against a computer, which is why abstraction is needed.

- A chess piece in real life is represented by a model in 3-dimensional space. Instead of representing a piece as a 3D model, which will require more computing power to render to the screen, I will represent each piece as a 2D sprite on the screen, removing the third dimension.
- Unnecessary sound of a chess piece being moved on the board will be removed as it does not affect game experience.
- The full motion of a piece does not have to be shown when being moved to a different square. Instead when a piece is clicked to a square it can teleport there, removing the dragging movement from one square to another.
- Buttons on screen can change some game modes. This will remove the need to manually set up the board and switch to a different board of a different colour.

Thinking Ahead

The inputs, outputs and tools in the game must be considered to make generating a solution easier.

- Python is a programming language primarily used in AI and is generally good for computation. It can also enable me to create a highly modular solution to the problem, and both my stakeholders have a python environment on their computers. Therefore, I will undertake the project in python.
- The user will most likely use a mouse or touchpad in order to input data into the game. The player should be able to use it to click on buttons. Also the player should be able to click on a piece they want to move, and click on the new position of a piece.
- When a piece is moved, the screen must output the board, but with the piece in its new position and its old position blank.
- When a chess match ends, the screen must output the result of the game, which will be the winner of the game or if the game ended in a draw/stalemate.
- The pygame library in python enables the use of special functions such as creating a game window and tracking the location of a mouse on screen. These functions are essential for

taking in data and displaying the output, so I will use the pygame to aid me in completing the project.

Thinking Procedurally and Decomposition

A chess game has two main physical aspects; a chess board and chess pieces.

- A chess board is where the chess game takes place. The board can be represented by a 8 x 8 2D array, where each cell in the array is a square. The board may have some functions to evaluate the current state of the board based on the pieces on it or have functions allowing the movement of pieces.
    - A board can be broken up into 64 individual squares. Each square can have a different colour, and perhaps can act as a button to move pieces.
- Someone will play chess using six types of chess pieces; pawns, knights, bishops, rooks, queens and the king. Each piece can traverse/move on the board in different ways. A template can be built for the generic qualities of a piece, then each specific type of piece can be based on that template.
    - Each piece can move from its original position to a final position depending on the rules of each piece. A template might be made to define what is meant by a move.

The final aspect I will need to consider is the AI player;

- The AI should allow for different levels of difficulty based on what the player wants to play against.
- Depending on the level, the AI player should probably look a certain number of moves ahead. It will then try to determine whether the position is good or bad, then backtrack to the original position to get the first move in that sequence.

Thinking Logically

Selection and repetition will be important to complete the game.

- A game loop is needed to repeatedly check the game/board state and update the screen accordingly.
- The code must branch to see if the player has clicked on the screen. It must branch further to see if a button has been pressed or if the board has been pressed.
- If a player has made a move, flip the turn so that the other player can make a move.
- The program must also repeatedly look at possible future moves ahead to determine if the next move is good or bad. It then must backtrack to the first move in that sequence to then play it. A recursive solution for this may be better than an iterative solution.
- Stop the game if the board is in checkmate or stalemate, then output the outcome.

Thinking Concurrently

If the program can carry out more than one process at once, it can increase the efficiency of the algorithm.

- Checking if the game has ended, and checking for the allowed/valid moves will happen at the same time.

Hardware and Software Requirements

I am using Python 3 to create the game on my personal computer, which is a MacBook Air with macOS Catalina. Additionally I am also using the pygame library give me some assisted functionalities/ The ideal hardware requirements to run python and pygame, from the python website, are as follows;

Ideal requirements for Python 3

CPU	x86 64-bit (Intel/AMD architecture)
RAM	4 GB
Disk Space	5 GB

My computer meets these requirements, so I am set to undertake the project.

I would recommend for my end user's computer to have the following requirements;

- 4 GB of RAM: This allows enough space to load the pieces and board onto the screen and calculate the best move without using virtual memory. Anything less may use virtual memory which can increase the runtime of the program as it will have to fetch information from secondary storage.
- 1.2 GHz processor: The program will have to repeatedly check for inputs and render graphics to the screen. A too low of s clock speed can cause the game to lag or be unresponsive to inputs.
- Ability to run python: My project will be made using python and distributed to my end user as python code. Therefore, the user must be able to run python on their computer every time they want to open and play the game.
- 11 inch Monitor/Screen: The game screen will be displayed visually to the user as a window. To see the game screen, the user must have a large enough monitor/screen to be able to comfortably fit the game screen on it.
- Mouse clicker: To input information into the game, the user must move a mouse on a screen and click in order to place an input. This is a standard input device for a computer so it makes sense for information to be taken in by this.

Most desktop computers are able to run python and all have the ability to control a mouse on a screen, so this makes my game accessible to many people. I also checked with [redacted] and [redacted] that their computers will be able to run the game and meet the requirements. The end user will also need some sort of monitor to see the game screen, which all desktops can do.

Success Criteria

The following criteria should be met for the project to be considered a success;

#	Criteria	Sub-Criteria	Justification
1	Create a game screen		To provide a visual interface for the user
2	Being able to exit the game screen		Allows the player to stop playing the game

3	Menu window to change the game settings		Gives the user an clear to place to change the game settings more easily
4	Ability to play as either colour with buttons	Play as white	Playing as either colour is a vital part in chess, as it affects who's turn it is first. The player should be able to choose if they want to play first or second.
		Play as black	
5	Ability to change board colour with buttons	Change board colour to brown	Option to change board colour allows for the player to have a more personalised experience
		Change board colour to grey	
6	Click on the board to move a piece during the player's turn		Using a clicker is the method to make a piece move so that a player can complete their turn
7	Computer makes a move on its turn		The computer AI should make a move on its turn to continue with the game
8	Have the option of six levels of user difficulty		Allows for a wide range of difficulties for the user to train against
9	Ability to change the computer difficulty during gameplay by with buttons		Allows the user to train against an easier or harder opponent in different parts of the game. This criteria must be met as this is the reason I am creating the game
10	Should follow the rules of classical chess	Pawn makes valid moves	Each piece should follow the rules of chess to traverse/move on the board in their own ways.
		Knight makes valid moves	
		Bishop makes valid moves	
		Rook makes valid moves	
		Queen makes valid moves	
		King makes valid moves	
11	Visualise and mark where pieces can go before moving them		Being able to visualise where pieces go can help novice users

			play the game correctly
12	Should output game result when the game has ended	Output that white won if white wins by checkmate	The player should know that the game has ended and how it has ended
		Output that black won if black wins by checkmate	
		Output that the game ended in stalemate/a draw is the game ends by stalemate	

Design

System Diagram

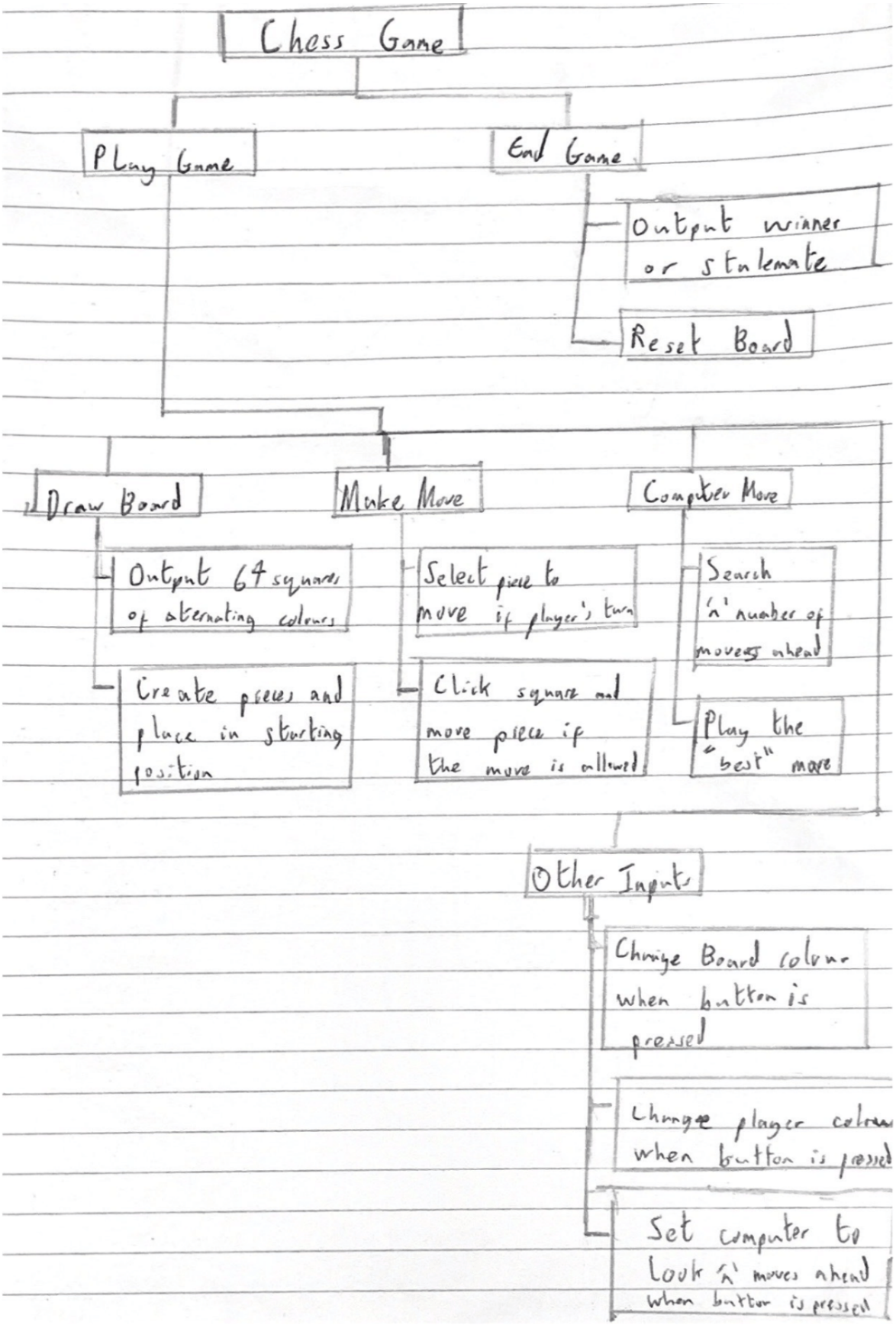
A decomposition diagram is a good way to help me break down the game into smaller more manageable parts and modules. This will enable me to focus on each sub-problem in order to create the game and to figure out what classes and functions I need to make in order to complete the game. For example, in the diagram below, when playing the game I must draw the board to the screen with a function which outputs all the squares and all the pieces.

A flow chart of the main game loop of the game can also help me determine the structure of the program. It will allow me to figure out what key variables and flags I will need. For example, in the flowchart below, the program will have to see whether or not the board state is in stalemate which will probably be a flag I'll need to create, and I will probably need a variable to indicate who's turn it is to determine if the player or computer should make a move. Based on my flowchart, it seems that the turn variable should flip every time a move is made. The game will end when the board state is either checkmate or stalemate.

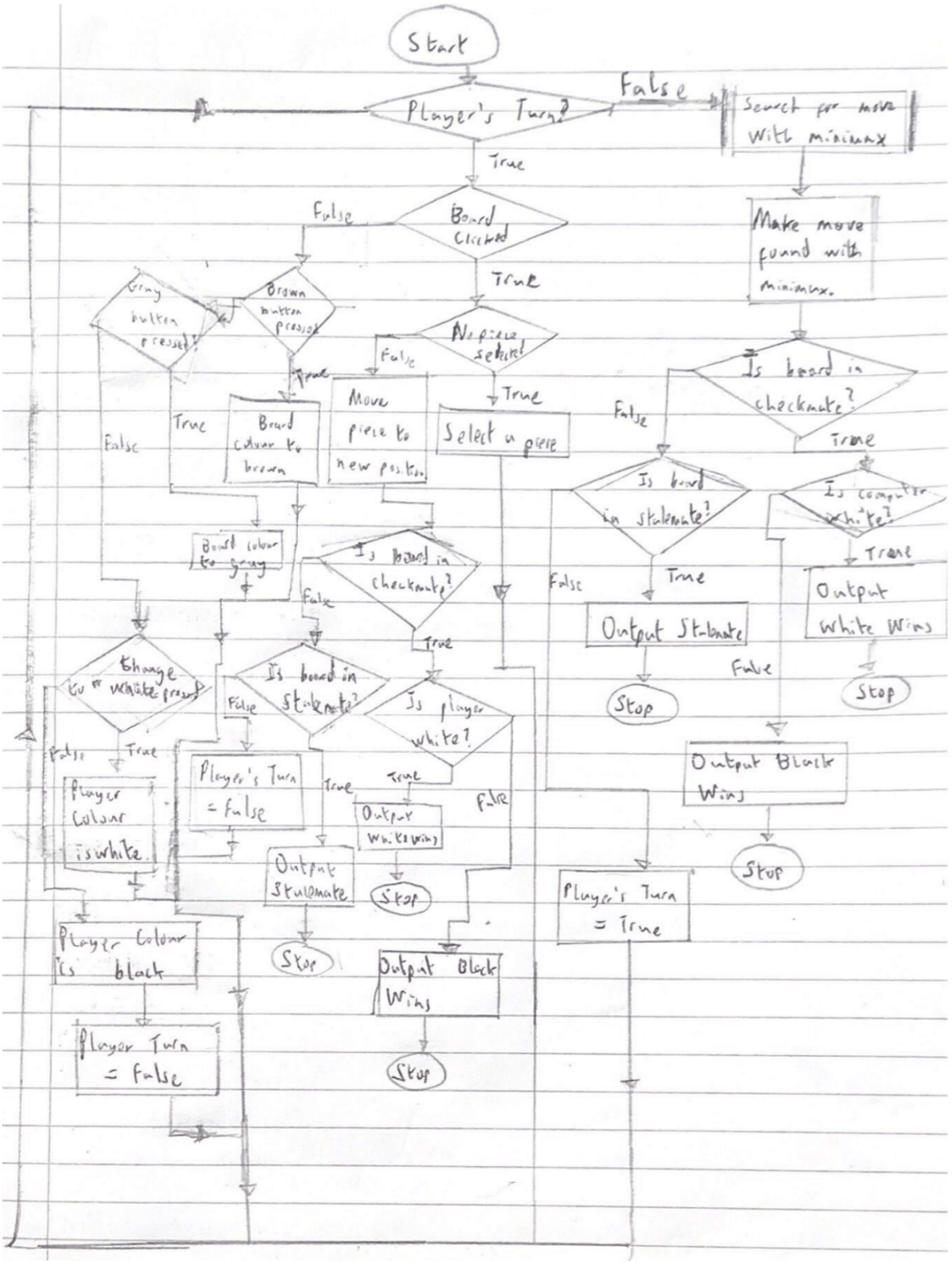
These will help me to practise computational methods while implementing the project. Deciding on inputs and outputs such as changing the board colour is seen as "thinking ahead," while deciding what decisions to make in the game and to end the game is seen as "thinking logically." This will make it easier to systematically complete the project.

I can decompose the problem further by dividing my solution into the objects needed to create the game; a board, squares, pieces, a chess move, and an AI. Doing this enables me to focus on smaller parts of the game, then put it together to make a complete game.

Decomposition Chart



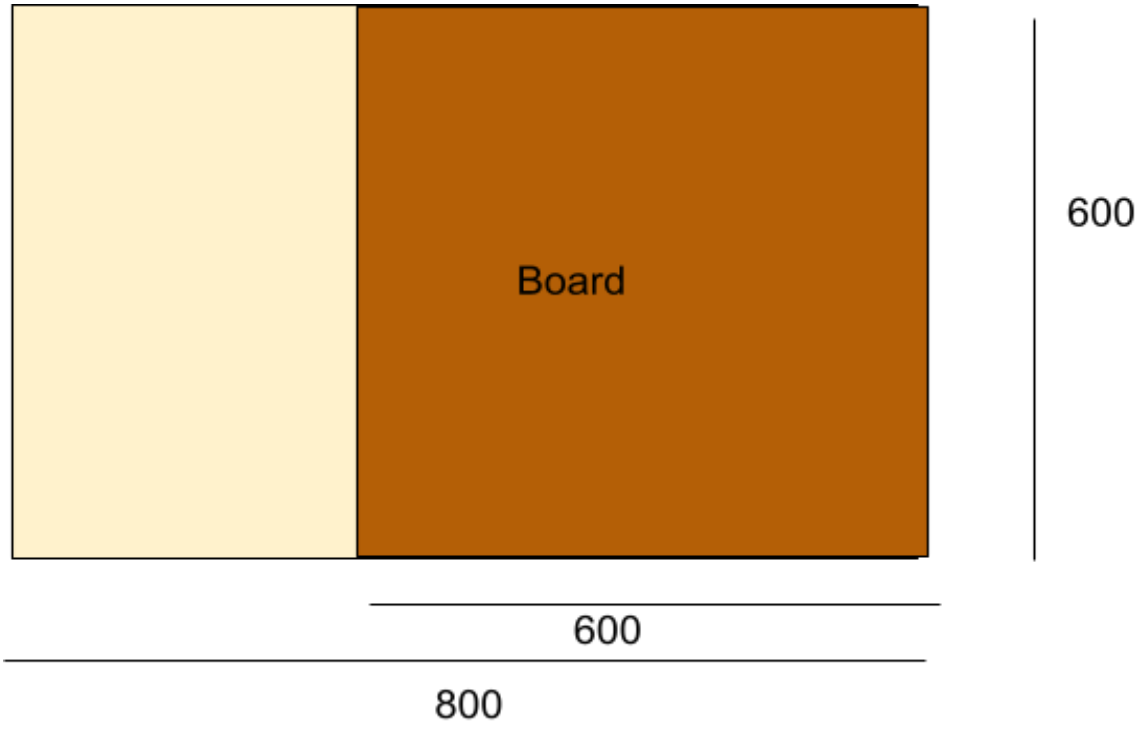
Flow Chart of Game Loop



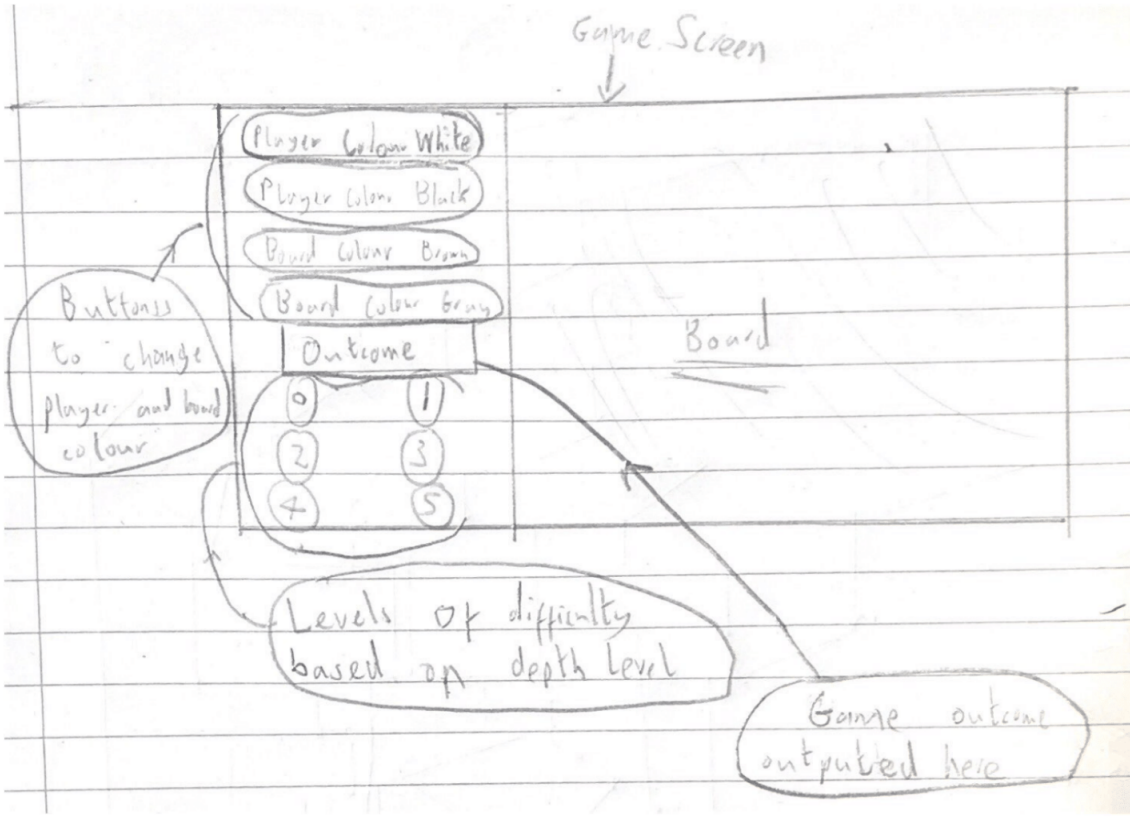
Designing Screen and User Interface

The game screen has the dimensions of 800 x 600 pixels to ensure that it is small enough to fit on almost any computer screen but large enough for the user to comfortably see it without straining their eyes.

The chess board must be a square shape, therefore the largest dimension the board can be on the given screen is 600 x 600 pixels. It is important for the board to be as large as possible to make sure the end user can clearly see the game as it is being played. The board is placed on the right side of the screen as it feels more natural to have a menu or other functions on the left side of the screen.



The remaining part of the screen will be for the buttons to control the board colour, player, colour, and AI difficulty. A space for the game outcome is also provided so the user can see how the game ended. The buttons will be of their associating colour so that the user can easily understand what each of them do. Difficulty levels get harder as the numbers increase, making it easier for the user to perceive how much more difficult one level is to another.



The squares on the board should alternate between light and dark squares, in accordance with the classic design of a chess board.

When the game starts, the game should set up the pieces with white pieces at the bottom of the screen if the player is white, or with the black pieces at the bottom if the player is black. This is similar to other chess platforms.

- White pieces on bottom when player is white on chess.com

- Black pieces on bottom when player is black on chess.com

Doing this will give the user perspective they would have as if they were playing over a real board, giving a more lifelike experience while playing.

Traditional images of the pieces will be used so that the player can clearly distinguish between them.

Designing Square Class

The board on the screen consists of 64 squares. Each square will have an x and y position on the board. Depending on their position, they may be either a dark or a light coloured square such that the board consists of alternating coloured squares. The main square colour is given by the board colour.

Square Class
x y position square piece_on_square
getSquare() getPosition() getPieceOnSquare() setPieceOnSquare()

Pseudocode

```
Class Square()

	Procedure constructor(boardColour, squarePos)

		light_brown_square = image(light brown square image)
		dark_brown_square = image(dark brown square image)
		light_grey_square = image(light grey square image)
		dark_grey_square  = image(dark grey square image)

		x = squarePos[0]
		y = squarePos[1]

		IF (x % 2 == y % 2) THEN
			IF boardColour == “brown” THEN
				square = light_brown_square
			ELSE
				square = dark_brown_square

			IF boardColour == “grey” THEN
				square = light_grey_square
			ELSE
				square = dark_grey_square

		piece_on_square = NULL

	Function getSquare()
		return Square

	Function getPosition()
		return position
```

```
Function getPieceOnSquare()
	return piece_on_square

Procedure setPieceOnSquare(piece)
	piece_on_square = piece
```

Structures

Type	Name	Data Type	Explanation	Reason
Variable	light_brown_square	Surface	To store the light brown square image	Gives square access to image to become the square its supposed to be
Variable	dark_brown_square	Surface	To store the dark brown square image	Gives square access to image to become the square its supposed to be
Variable	light_grey_square	Surface	To store the light grey square image	Gives square access to image to become the square its supposed to be
Variable	dark_grey_square	Surface	To store the dark grey square image	Gives square access to image to become the square its supposed to be
Attribute	x	Integer	To store horizontal position of the square	Use the x coordinate to place square on board and calculate what colour the square should be
Attribute	y	Integer	To store vertical position of the square	Use the y coordinate to place square on board and calculate what colour the square should be
Attribute	square	Surface	Stores the actual square image	Actual square image must be saved to be placed on the screen
Attribute	piece_on_square	Object	Stores the piece on square	Each square may or may not have a piece on it. It's important to keep track so we know whether a square has a piece on it so we can figure

				out how a piece can move on a board

Test Data

We can test whether or not the correct square is being outputted to the screen, as well as if the position and piece on the square is correct by printing those values.

Structure being Tested	Test Data	Expected Outcome	Reasoning
getPosition	Position (0, 0)	Prints position (0, 0)	To check if given position in the class is correct
getPieceOnSquare	Null	Prints Null	To check if given piece in the class is correct
getSquare	Position (0, 0) and board colour “brown”	Outputs light brown square	To see if class calculates the correct square with given position and colour
getSquare	Position (1, 0) and board colour “brown”	Outputs dark brown square	To see if class calculates the correct square with given position and colour
getSquare	Position (0, 0) and board colour “grey”	Outputs light grey square	To see if class calculates the correct square with given position and colour
getSquare	Position (1, 0) and board colour “grey”	Outputs dark grey square	To see if class calculates the correct square with given position and colour

Designing Moves Class

The simplest way to define a move in chess is with two positions on the board; a start and end position. By defining a move in a class, I am able to group together both board positions for easy access.

Move
oPos nPos
getOPos() getNPos()

Pseudocode

```
Class Move
	Procedure constructor(oldPosition, newPosition)
		oPos = oldPosition
		nPos = newPosition
```

```
Function getOPos()
	return oPos

Function getNPos()
	return nPos
```

Structures

Type	Name	Data Type	Explanation	Reason
Attribute	oPos	Tuple	Stores the x and y positions of the original position	Represents the start point for a move to see which piece to move
Attribute	nPos	Tuple	Stores the x and y positions of the new position	Represents the end point for a move to place the piece there

Designing Piece Classes

The six types of pieces in chess (pawn, knight, bishop, rook, queen and king) are probably the most important objects in the game. They have similar properties but can traverse/move on the board in different ways and have different images. Therefore I can make a parent class for the pieces where each type of piece can inherit similar properties from. Each piece will have its own algorithm to find out its allowed moves. Some pieces are able to move in a similar way to each other, such as a queen which moves like a rook (along straight lines) or like a bishop (along diagonal lines), so the piece class can contain methods allowing all pieces to move like that and those methods can be called by the pieces which need it.

Each piece must have a value so that the computer can make decisions based on which piece in "better," so I will used former chess grandmaster Bobby Fischer's evaluation of the pieces to inform my decision on the valuation;

Pawn	Knight	Bishop	Rook	Queen	King
10	30	30 + an infinitely small amount more than the knight	50	90	Infinity (as game ends when its captured)

Pieces
x_position y_position position colour pieceType

value moved image validMoves
addMove() getColour() getType() getPositionX() getPositionY() getValue() getImage() getMoved() getValidMoves() setMovedTrue() setPosition() checkStraights() checkDiagonals()

Pawn inherits Piece
direction pieceType = “pawn” value = 10 image = image(pawn image for that colour)
getValidMoves() ***overwritten***

Knight inherits Piece
pieceType = “knight” value = 30 image = image(knight image for that colour)
getValidMoves() ***overwritten***

Bishop inherits Piece
pieceType = “bishop” value = 30.01 image = image(bishop image for that colour)
getValidMoves() ***overwritten***

Rook inherits Piece
pieceType = “rook” value = 50 image = image(rook image for that colour)

getValidMoves() ***overwritten***

Queen inherits Piece
pieceType = “queen” value = 90 image = image(queen image for that colour)
getValidMoves() ***overwritten***

King inherits Piece
pieceType = “king” value = infinity image = image(king image for that colour)
getValidMoves() ***overwritten*** canCastle()

Pseudocode for Piece

```
Class Piece
	Procedure constructor(nColour, nx_position, ny_position, npieceType, nValue)

		colour = nColour
		x_position = nx_position
		y_position = ny_position
		position = (x_position, y_position)
		pieceType = npieceType
		value = nValue
		moved = False
		image = Null
		validMoves = []


	Procedure addMove(move)
		validMoves.add(move)

	Function getColour()
		return colour

	Function getType()
		return pieceType

	Function getXPosition()
		return x_position

	Function getYPosition()
		return y_position
```

```
Function getValue()
	return value

Function getMoved()
	return moved

Function getValidMoves()
	return validMoves

Procedure setMovedTrue()
	moved = True

Procedure setPosition(position)
	x_position = position[0]
	y_position = position[1]

Procedure checkStraights(board)
	// North positions
	FOR i = y_position + 1 to 8
		IF board[x_position][i]==(blank space) THEN
			move = new Move(position, (x_position, i))
			addMove(move)
		ELSE IF board[x_position][i]==(opposing piece) THEN
			move = new Move(position, (x_position, i))
			addMove(move)
			ENDLOOP
		ELSE
			ENDLOOP

	// East positions
	FOR i = x_position + 1 to 8
		IF board[i][y_position]==(blank space) THEN
			move = new Move(position, (i, y_position))
			addMove(move)
		ELSE IF board[i][y_position]==(opposing piece) THEN
			move = new Move(position, (i, y_position))
			addMove(move)
			ENDLOOP
		ELSE
			ENDLOOP

	// South positions
	FOR i = y_position - 1 to -1; i --
		IF board[x_position][i]==(blank space) THEN
			move = new Move(position, (x_position, i))
			addMove(move)
		ELSE IF board[x_position][i]==(opposing piece) THEN
			move = new Move(position, (x_position, i))
			addMove(move)
			ENDLOOP
		ELSE
			ENDLOOP
```

```
        // West Positions
        FOR i = x_position - 1 to -1; i --
                IF board[i][y_position]==(blank space) THEN
                        move = new Move(position, (i, y_position))
                        addMove(move)
                ELSE IF board[i][y_position]==(opposing piece) THEN
                        move = new Move(position, (i, y_position))
                        addMove(move)
                        ENDLOOP
                ELSE
                        ENDLOOP

Procedure checkDiagonals(board)

        // North-East positions
        FOR i = 1 to 8:
                IF board[x_positoin+i][y_position+i]==(blank space) THEN
                        move = new Move(position, (x_positoin+i, y_position+i)
                        addMove(move)

                ELSE IF board[x_positoin+i][y_position+i]==(opposing piece)THEN
                        move = new Move(position, (x_positoin+i, y_position+i))
                        addMove(move)
                        ENDLOOP
                ELSE
                        ENDLOOP

        // North-West positions
        FOR i = 1 to 8:
                IF board[x_positoin-i][y_position+i]==(blank space) THEN
                        move = new Move(position, (x_positoin-i, y_position+i))
                        addMove(move)

                ELSE IF board[x_positoin-i][y_position+i]==(opposing piece)THEN
                        move = new Move(position, (x_positoin-i, y_position+i))
                        addMove(move)
                        ENDLOOP
                ELSE
                        ENDLOOP

        // South-East positions
        FOR i = 1 to 8:
                IF board[x_positoin+i][y_position-i]==(blank space) THEN
                        move = new Move(position, (x_positoin+i, y_position-i))
                        addMove(move)

                ELSE IF board[x_positoin+i][y_position-i]==(opposing piece)THEN
                        move = new Move(position, (x_positoin+i, y_position-i))
                        addMove(move)
                        ENDLOOP
                ELSE
                        ENDLOOP
```

```
        // South-West positions
        FOR i = 1 to 8:
              IF board[x_positoin-i][y_position0-i]==(blank space) THEN
                    move = new Move(position, (x_positoin-i, y_position-i))
                    addMove(move)

              ELSE IF board[x_positoin-i][y_position-i]==(opposing piece)THEN
                    move = new Move(position, (x_positoin-i, y_position-i))
                    addMove(move)
                    ENDLOOP
              ELSE
                    ENDLOOP

FUNCTION getValidMove()
      return validMoves
```

Structures for Pieces

Type	Name	Data Type	Explanation	Reason
Attribute	colour	String	A piece can be either “white” or “black” in chess	The piece colour determines the image of the piece and whether it is moved by the player or computer
Attribute	x_position	Integer	Contains x coordinate of a piece on the board	Can be used to calculate the piece’s possible moves based on its position
Attribute	y_position	Integer	Contains y coordinate of a piece on the board	Can be used to calculate the piece’s possible moves based on its position
Attribute	pieceType	String	To identify what type of piece the object is	Depending on the type of piece, it can move on the board differently
Attribute	value	Float	To give a numerical valuation to a piece	When the AI is made, the computer should be able to decide which pieces are worth so it can make logical decisions
Attribute	moved	Boolean	A flag indicating whether a piece has moved or not	Depending on whether a piece has moved, it may or

				may not be able to perform special moves such as castling
Attribute	image	Surface	Store the image of the piece	Every piece needs an image to be represented on the screen
Attribute	validMoves	Array	A list of all the moves a piece is allowed to make	Will help to compile a database of all the moves a player or computer is allowed to choose from by getting the allowed moves for each individual piece
Variable	move	Object	Temporarily placeholder for a move	As a move is calculated, it is saved temporarily to be added to the possible list of valid moves which will be used later

Pseudocode for Pawn

```
Class Pawn inherits Piece

	Procedure constructor(nColour, x, y)
		Super constructor(nClour, x, y, “pawn”, 10)

		IF colour == “white” THEN
			image = image(white pawn image)
		ELSE
			image = image(black pawn image)

		direction = 0

	FUNCTION getValidMoves(board)

		IF player’s colour == piece colour THEN
			direction = (+) 1 // to move forward
		ELSE
			direction = -1 // to move backwards

		// To move one space forward
		IF board[x_position][y_position + direction]==(blank space) THEN
			move = new Move(position, (x_position, y_position + direction))
			addMove(move)
```

```
            // To move two spaces forward
            IF board[x_position][y_position + 2 * direction]==(blank space) THEN

                  IF moved == False THEN
                        move=new Move(position,(x_position, y_position+
2*direction))
                        addMove(move)

            // To capture to the right
            IF board[x_position + 1][y_position+direction]==(opponent piece) THEN
                  move=new Move(position, (x_position+1, y_position+direction))
                  addMove(move)

            // To capture to the left
            IF board[x_position - 1][y_position+direction]==(opponent piece) THEN
                  move=new Move(position, (x_position-1, y_position+direction))
                  addMove(move)

            return validMoves
```

Structures of Pawn

Type	Name	Data Type	Explanation	Reason
Attribute	direction	Integer	Determines whether a pawn should move up or down the board	Depending on the player's colour, the orientation of the board will be different and the pawn should know to move forward relative to the orientation and its colour

Test Data for Pawn

Placing a pawn in the middle of the board can allow us to test whether or not it can move to the correct position. We can also check if the pawn can capture opposing pieces by placing them in positions where our pawn should be able to capture them.

Structure being Tested	Test Data	Expected Outcome	Reasoning
getValidMoves()	Position (3, 4) and moved = false	Pawn shows the ability to move two spaces forward or move one space forward	Rules of chess state that pawn can move one or two spaces forward on the first move as long as a piece does not block it
getValidMoves()	Position (3, 4) and moved = true	Pawn shows the ability to move one space forward only	Pawn should be able to move one space forward only if it has been already

			been move, should not move two spaces forward
getValidMoves()	Piece on right diagonal of pawn and piece on left diagonal of pawn	Pawn shows the ability to capture pieces on left and right diagonal	Pawn should only be able to capture pieces on its left or right diagonals

Pseudocode for Knight

```
Class Knight inherits Piece
	Procedure constructor(nColour, x, y)
		Super constructor(nClour, x, y, “knight”, 30)

		IF colour == “white” THEN
			image = image(white knight image)
		ELSE
			image = image(black knight image)

	FUNCTION getValidMoves(board)

		move = Null

		possible_moves = [
			(x_position+2, y_position+1),
			(x_position+2, y_position-1),
			(x_position-2, y_position+1),
			(x_position-2, y_position-1),
			(x_position+1, y_position+2),
			(x_position+1, y_position-2),
			(x_position-1, y_position+2),
			(x_position-1, y_position-2)
		]

		FOR move in possible_move
			IF board[move[0]][move[1]]==(black space)or(opposing piece)THEN
				m = new Move(position, move)
				addMove(m)
		return validMoves
```

Structures of Knight

Type	**Name**	**Data Type**	**Explanation**	**Reason**
Variable	possible_moves	List	Knight moves in an “L” shape such that one length is 2 spaces and the other is 1, this list stores the 8 combinations	The program test every move on the list to see if its allowed, if it is then it is added to validMoves

Test Data for Knight

A good way to test the knight is to place it in its starting position, as we are not concerned with the pieces in between the start and end positions.

Structure being Tested	Test Data	Expected Outcome	Reasoning
getValidMoves()	Position (1, 7) and pawn of same colour in scope of knight	Knight shows ability to move in “L” shape pattern	Knight should be able to move to a blank space in the “L” shape pattern, but also shouldn’t be able to capture its own pieces
getValidMoves()	Opposing piece in scope of knight	Knight shows ability to capture enemy pawn	Knight should be able to capture enemy pieces only

Pseudocode for Bishop

```
Class Bishop inherits Piece
	Procedure constructor(nColour, x, y)
		Super constructor(nClour, x, y, “bishop”, 30.01)

		IF colour == “white” THEN
			image = image(white bishop image)
		ELSE
			image = image(black bishop image)

	FUNCTION getValidMoves(board)
		checkDiagonals()
		return validMoves
```

Test Data for Bishop

Placing a bishop in the middle of the board in its starting position can test if it can move diagonally, capture enemy pieces and not capture friendly pieces.

Structure being Tested	Test Data	Expected Outcome	Reasoning
getValidMoves()	Position (3, 4)	Bishop shows the ability to move diagonally, capture enemy pieces and not capture friendly pieces	According to the rules of chess a bishop should move diagonally, be able to capture enemy pieces, and not be able to capture its own pieces

Pseudocode for Rook

```
Class Rook inherits Piece
	Procedure constructor(nColour, x, y)
		Super constructor(nClour, x, y, “rook”, 50)

		IF colour == “white” THEN
			image = image(white rook image)
```

```
            ELSE
                  image = image(black rook image)

      FUNCTION getValidMoves(board)
            checkStraights()
            return validMoves
```

Test Data for Rook

Similarly to the bishop, placing a rook in the middle of the board can test if it can move along straights, capture enemy pieces and not capture friendly pieces.

Structure being Tested	Test Data	Expected Outcome	Reasoning
getValidMoves()	Position (3, 4)	Rook shows the ability to move along straights, capture enemy pieces and not capture friendly pieces	According to the rules of chess a rook should move along straights, be able to capture enemy pieces, and not be able to capture friendly pieces

Pseudocode for Queen

```
Class Queen inherits Piece
      Procedure constructor(nColour, x, y)
            Super constructor(nClour, x, y, “queen”, 90)
            IF colour == “white” THEN
                  image = image(white queen image)
            ELSE
                  image = image(black queen image)

      FUNCTION getValidMoves(board)
            checkStraights()
            checkDiagonals()
            return validMoves
```

Test Data for Queen

The queen moves like the bishop and the rook, so placing it on the same coordinates in the middle of a set board can check this.

Structure being Tested	Test Data	Expected Outcome	Reasoning
getValidMoves()	Position (3, 4)	Queen shows the ability to move along straights and diagonals, capture enemy pieces and not capture friendly pieces	According to the rules of chess a queen should move along straights and diagonals, be able to capture enemy pieces, and not be able to capture friendly pieces

Pseudocode for King

```
Class King inherits Piece
	Procedure constructor(nColour, x, y)
		Super constructor(nClour, x, y, “king”, infinity)
		IF colour == “white” THEN
			image = image(white king image)
		ELSE
			image = image(black king image)

	FUNCTION getValidMoves(board)
		move = Null

		possible_moves = [
			(x_position+1, y_position),
			(x_position+1, y_position+1),
			(x_position+1, y_position-1),
			(x_position, y_position+1),
			(x_position-1, y_position),
			(x_position-1, y_position+1),
			(x_position-1, y_position-1),
			(x_position, y_position-1)
		]
		FOR move in possible_move
			IF board[move[0]][move[1]]==(black space)or(opposing piece)THEN
				m = new Move(position, move)
				addMove(m)
		return validMoves

	FUNCTION canCastle(board)

		IF moved == False and (no pieces between king and rook) THEN
			return True
		ELSE
			return False
```

Test Data for King

King will be tested similarly to the other pieces. The function allowing for castling will probably need to be edited further during development as I don’t know how I can have pieces know their locations relative to each other.

Structure being Tested	**Test Data**	**Expected Outcome**	**Reasoning**
getValidMoves()	Position (3, 4)	King shows the ability to one space in any direction, capture enemy pieces and not capture friendly pieces	According to the rules of chess a king should move one space in any direction, be able to capture enemy pieces, and not be able to capture friendly pieces

Designing Board Functionalities

The board class will contain most of the functionalities for the game.

Board
board board_colour player_colour turn moves_on_board piece_selected player_white_start_pos player_black_start_pos pawns_space_table knight_space_table bishop_space_table rook_space_table queen_space_table king_space_table
setupBoard() drawBoard() evaluate() boardClicker() isInCheck() isCheckmate() isStalemate()

Pseudocode

- (Tables for spaces are from freecodecamp.org)

```
Class Board
	PROCEDURE constructor(nboard_colour, nplayer_colour)
		board = (8 x 8 matrix)
		board_colour = nboard_colour
		player_colour = nplayer_colour
		turn = “white”
		moves_on_board = []
		piece_selected = Null

		player_white_start_pos = [
			['r', 'n', 'b', 'q', 'k', 'b', 'n', 'r'],
			['p', 'p', 'p', 'p', 'p', 'p', 'p', 'p'],
			['.', '.', '.', '.', '.', '.', '.', '.'],
			['.', '.', '.', '.', '.', '.', '.', '.'],
			['.', '.', '.', '.', '.', '.', '.', '.'],
			['.', '.', '.', '.', '.', '.', '.', '.'],
			['P', 'P', 'P', 'P', 'P', 'P', 'P', 'P'],
			['R', 'N', 'B', 'Q', 'K', 'B', 'N', 'R']
		]
		player_black_start_pos = [
```

```
        ['R', 'N', 'B', 'K', 'Q', 'B', 'N', 'R'],
        ['P', 'P', 'P', 'P', 'P', 'P', 'P', 'P'],
        ['.', '.', '.', '.', '.', '.', '.', '.'],
        ['.', '.', '.', '.', '.', '.', '.', '.'],
        ['.', '.', '.', '.', '.', '.', '.', '.'],
        ['.', '.', '.', '.', '.', '.', '.', '.'],
        ['p', 'p', 'p', 'p', 'p', 'p', 'p', 'p'],
        ['r', 'n', 'b', 'k', 'q', 'b', 'n', 'r']
]

pawns_space_table = [
        [0.0,  0.0,  0.0,  0.0,  0.0,  0.0,  0.0,  0.0],
        [5.0,  5.0,  5.0,  5.0,  5.0,  5.0,  5.0,  5.0],
        [1.0,  1.0,  2.0,  3.0,  3.0,  2.0,  1.0,  1.0],
        [0.5,  0.5,  1.0,  2.5,  2.5,  1.0,  0.5,  0.5],
        [0.0,  0.0,  0.0,  2.0,  2.0,  0.0,  0.0,  0.0],
        [0.5, -0.5, -1.0,  0.0,  0.0, -1.0, -0.5,  0.5],
        [0.5,  1.0, 1.0,  -2.0, -2.0,  1.0,  1.0,  0.5],
        [0.0,  0.0,  0.0,  0.0,  0.0,  0.0,  0.0,  0.0]
]

knight_space_table = [
        [-5.0, -4.0, -3.0, -3.0, -3.0, -3.0, -4.0, -5.0],
        [-4.0, -2.0,  0.0,  0.0,  0.0,  0.0, -2.0, -4.0],
        [-3.0,  0.0,  1.0,  1.5,  1.5,  1.0,  0.0, -3.0],
        [-3.0,  0.5,  1.5,  2.0,  2.0,  1.5,  0.5, -3.0],
        [-3.0,  0.0,  1.5,  2.0,  2.0,  1.5,  0.0, -3.0],
        [-3.0,  0.5,  1.0,  1.5,  1.5,  1.0,  0.5, -3.0],
        [-4.0, -2.0,  0.0,  0.5,  0.5,  0.0, -2.0, -4.0],
        [-5.0, -4.0, -3.0, -3.0, -3.0, -3.0, -4.0, -5.0]
]

bishop_space_table = [
        [-2.0, -1.0, -1.0, -1.0, -1.0, -1.0, -1.0, -2.0],
        [-1.0,  0.0,  0.0,  0.0,  0.0,  0.0,  0.0, -1.0],
        [-1.0,  0.0,  0.5,  1.0,  1.0,  0.5,  0.0, -1.0],
        [-1.0,  0.5,  0.5,  1.0,  1.0,  0.5,  0.5, -1.0],
        [-1.0,  0.0,  1.0,  1.0,  1.0,  1.0,  0.0, -1.0],
        [-1.0,  1.0,  1.0,  1.0,  1.0,  1.0,  1.0, -1.0],
        [-1.0,  0.5,  0.0,  0.0,  0.0,  0.0,  0.5, -1.0],
        [-2.0, -1.0, -1.0, -1.0, -1.0, -1.0, -1.0, -2.0]
]

rook_space_table = [
        [ 0.0,  0.0,  0.0,  0.0,  0.0,  0.0,  0.0,  0.0],
        [ 0.5,  1.0,  1.0,  1.0,  1.0,  1.0,  1.0,  0.5],
        [-0.5,  0.0,  0.0,  0.0,  0.0,  0.0,  0.0, -0.5],
        [-0.5,  0.0,  0.0,  0.0,  0.0,  0.0,  0.0, -0.5],
        [-0.5,  0.0,  0.0,  0.0,  0.0,  0.0,  0.0, -0.5],
        [-0.5,  0.0,  0.0,  0.0,  0.0,  0.0,  0.0, -0.5],
        [-0.5,  0.0,  0.0,  0.0,  0.0,  0.0,  0.0, -0.5],
        [0.0,   0.0,  0.0,  0.5,  0.5,  0.0,  0.0,  0.0]
]
```

```
        queen_space_table = [
                [-2.0, -1.0, -1.0, -0.5, -0.5, -1.0, -1.0, -2.0],
                [-1.0,  0.0,  0.0,  0.0,  0.0,  0.0,  0.0, -1.0],
                [-1.0,  0.0,  0.5,  0.5,  0.5,  0.5,  0.0, -1.0],
                [-0.5,  0.0,  0.5,  0.5,  0.5,  0.5,  0.0, -0.5],
                [ 0.0,  0.0,  0.5,  0.5,  0.5,  0.5,  0.0, -0.5],
                [-1.0,  0.5,  0.5,  0.5,  0.5,  0.5,  0.0, -1.0],
                [-1.0,  0.0,  0.5,  0.0,  0.0,  0.0,  0.0, -1.0],
                [-2.0, -1.0, -1.0, -0.5, -0.5, -1.0, -1.0, -2.0]
        ]

        king_space_table = [
                [-3.0, -4.0, -4.0, -5.0, -5.0, -4.0, -4.0, -3.0],
                [-3.0, -4.0, -4.0, -5.0, -5.0, -4.0, -4.0, -3.0],
                [-3.0, -4.0, -4.0, -5.0, -5.0, -4.0, -4.0, -3.0],
                [-3.0, -4.0, -4.0, -5.0, -5.0, -4.0, -4.0, -3.0],
                [-2.0, -3.0, -3.0, -4.0, -4.0, -3.0, -3.0, -2.0],
                [-1.0, -2.0, -2.0, -2.0, -2.0, -2.0, -2.0, -1.0],
                [ 2.0,  2.0,  0.0,  0.0,  0.0,  0.0,  2.0,  2.0],
                [ 2.0,  3.0,  1.0,  0.0,  0.0,  1.0,  3.0,  2.0]
        ]


PROCEDURE setUpBoard()
        FOR i = 0 to 8
                FOR j = 0 to 8
                        IF player_colour == “white” THEN
                                square = player_white_start_pos[i][j]
                        ELSE
                                square = player_black_start_pos[i][j]

                        IF square = “R” THEN
                                board[i][j] = new Rook(“white”, i, j)
                        ELSE IF square = “N” THEN
                                board[i][j] = new Knight(“white”, i, j)
                        ELSE IF square = “B” THEN
                                board[i][j] = new Bishop(“white”, i, j)
                        ELSE IF square = “Q” THEN
                                board[i][j] = new Queen(“white”, i, j)
                        ELSE IF square = “K” THEN
                                board[i][j] = new King(“white”, i, j)
                        ELSE IF square = “P” THEN
                                board[i][j] = new Pawn(“white”, i, j)
                        ELSE IF square = “r” THEN
                                board[i][j] = new Rook(“black”, i, j)
                        ELSE IF square = “n” THEN
                                board[i][j] = new Knight(“black”, i, j)
                        ELSE IF square = “b” THEN
                                board[i][j] = new Bishop(“black”, i, j)
                        ELSE IF square = “q” THEN
                                board[i][j] = new Queen(“black”, i, j)
```

```
                    ELSE IF square = “k” THEN
                           board[i][j] = new King(“black”, i, j)
                    ELSE IF square = “p” THEN
                           board[i][j] = new Pawn(“black”, i, j)
PROCEDURE drawBoard()
      // Output squares
      square = Null
      FOR i = 0 to 8
            FOR j = 0 to 8
                    square = new Square(boardColour, (i, j))
                    OUTPUT_TO_SCREEN(square)
      // Output pieces
      piece = Null
      FOR i = 0 to 8
            FOR j = 0 to 8
                    piece = board[i][j]
                    OUTPUT_TO_SCREEN(piece)
FUNCTION evaluate()
      eval = 0
      piece = Null
      FOR i = 0 to 8
            FOR j = 0 to 8
                    piece = board[i][j]
                    IF piece.getColour() == player_colour THEN
                           eval += TABLE_VALUE(piece) + piece.getValue()
                    ELSE
                           eval -= TABLE_VALUE(piece) + piece.getValue()
      return eval

PROCEDURE boardClicker()
      IF piece_selected is Null THEN
            piece_selected = PIECE_ON_SQUARE_CLICKED_ON
      ELSE
            IF MOVE_TRYING_TO_BE_MADE is in piece_selected.getValidMoves
                    MOVE(piece_selected)
                    piece_selected = Null

FUNCTION isInCheck()
      // Find king position
      king_pos = Null
      in_check = False

      FOR i = 0 to 8
            FOR j = 0 to 8
                    piece = board[i][j]
                    IFpiece.getType()==“king”and piece.getColour()==turnTHEN
                           king_pos == (i, j)


      // See if end position of opposing pieces is the king’s position
      FOR i = 0 to 8
            FOR j = 0 to 8
                    piece = board[i][j]
```

```
                        IF piece.getColour() is not turn THEN
                              moves = piece.getValidMoves()
                                    FOR move in moves
                                          IF move.getNPos() == king_pos THEN
                                                in_check = True
            return in_check

      FUNCTION isCheckmate()
            IF isInCheck() and moves_on_board.length() == 0 THEN
                  return True
            ELSE
                  return False
      FUNCTION isStalemate()
            IF(moves_on_board.length()==0 and isInCheck==False) or
INSUFFICIENT_MATERIAL THEN
                  return True
            ELSE
                  return False
```

Structures

Type	Name	Data Type	Explanation	Reason
Attribute	board	2D array	A blank 8 x 8 matrix with pieces and blank spaces	This represents the board which the game takes place on and where the pieces will “move”, as a chess board is abstracted into a grid
Attribute	board_colour	String	Stores a board colour of either “brown” or “grey”	The colour of the board on the screen will be different based on this, so this is an indicator of what colour to draw the board
Attribute	player_colour	String	Stores a player colour of either “white” or “black”	The orientation and the pieces the player will be able to move will be different based on this, so this is an important indicator
Attribute	turn	String	Stores which colour’s turn it is; “white” or “black”	Depending on who’s turn it is, either the player will be allowed to move or the computer must make a move. The turn tells the program which is

				happening
Attribute	moves_on_board	Array	A list of the valid moves on the board at the current time	The player and computer need a database of moves to select from
Attribute	piece_selected	Object	Temporarily store the piece selected by the player to move	Something must indicate whether the player is trying to make a move and which piece they are trying to move
Attribute	player_white_start_pos	2D array	The starting layout of the board if the player is “white”	The program needs somewhere to read the starting position from if the player is “white”
Attribute	player_black_start_pos	2D array	The starting layout of the board if the player is “black”	The program needs somewhere to read the starting position from if the player is “black”
Attribute	pawns_space_table	2D array	The points for each square on the board for the position of the pawn	The computer must be able to give points to either player depending on position of pawn to run the AI
Attribute	knight_space_table	2D array	The points for each square on the board for the position of the knight	The computer must be able to give points to either player depending on position of knight to run the AI
Attribute	bishop_space_table	2D array	The points for each square on the board for the position of the bishop	The computer must be able to give points to either player depending on position of bishop to run the AI
Attribute	rook_space_table	2D array	The points for each square on the board for the position of the rook	The computer must be able to give points to either player depending on position of rook to run the AI
Attribute	queen_space_table	2D array	The points for each square on the board for the	The computer must be able to give points to either player

			position of the queen	depending on position of queen to run the AI
Attribute	king_space_table	2D array	The points for each square on the board for the position of the king	The computer must be able to give points to either player depending on position of king to run the AI
Variable	eval	Float	A numerical value based on the pieces on the board and positioning of those pieces on the board. Positive values indicate the player is doing better against the computer while negative values indicate that the player is worse	Assigning numerical values to board states makes each state more easily comparable to one another, so the AI can use this to decide on the “best” move for it to make
Variable	king_pos	Tuple	Position of the current player’s king	Is compared to the end positions of all the opponents pieces. If an end position is the same as the king’s position then the player with the king is in check as the king is threatened to be captured

Test Data

Structure being Tested	**Test Data**	**Expected Outcome**	**Reasoning**
evaluate()	Original starting position	Return a value around 0	Both sides of the board should receive the same evaluation at the start, so it should balance out to 0
evaluate()	Original starting position plus an extra queen	Return a value around 90	One side will have around 90 more points due to having an extra queen
isInCheck()	Create a position with no attack on the king	Returns False	If the board state is not in check then the the flag should return false
isInCheck()	Create a position	Returns True	If the board state is in

	with an attack on the king		check then the the flag should return true
isCheckmate()	Create a position which is not checkmate	Returns false	If the board state is not in checkmate then the flag should return false
isCheckmate()	Create a position which is checkmate	Returns true	If the board state is in checkmate then the flag should return true
isStalemate()	Create a position which is not stalemate	Returns false	If the board state is not in stalemate then the flag should return false
isStalemate()	Create a position which is stalemate	Returns true	If the board state is in stalemate then the flag should return true

Designing AI Class

At its lowest level, the AI will make random moves. Any level above that will look one more move ahead (via the minimax algorithm) hence creating greater difficulty as the computer would've seen more combinations. The minimax algorithm is recursive and will create a search tree to find the best move. Alpha-beta pruning is used to eliminate some branches from the tree to help speed up the search. The algorithm is a depth-first search algorithm, and backtracks to output the "best" move.

AI
depth
randomMove() minimax() bestMove()

Pseudocode

```
Class AI
	PROCEDURE constructor(nDepth)
		depth = nDepth

	FUNCTION randomMove(board)
		moves = board.getMovesOnBoard()
		move = CHOOSE_RANDOM(moves)
		return move

	FUNCTION minimax(depth, board, alpha=-infinity, beta=infinity)

		// Clause to end search
		IF depth ==0 or board.isCheckmate() or board.isStalemate()
			return board.evaluate(), best_move
```

```
IF turn is not player_colour THEN
	best_move = None
	// Starting maximum score is a very large negative number
	maxSore = -infinity

	// Look over every move
	FOR move in board.moves_on_board

		// Copy board, move piece and call the function again
		board_clone = COPY(board)
		MOVEPIECE(board_clone)
		score = minimax(depth-1, board_clone, alpha, beta)

		// Recalculate alpha
		alpha = MAX(alpha, maxScore)

		// Break if maximising is greater than minimising
		IF beta <= alpha THEN
			break


		// See if a better move was found
		IF score >= maxScore THEN
			maxScore = score
			best_move = move

	return best_move

ELSE
	best_move = None
	// Starting minimum score is a very large number
	minScore = infinity

	// Look over every move
	FOR move in board.moves_on_board

		// Copy board, move piece and call the function again
		board_clone = COPY(board)
		MOVEPIECE(board_clone)
		score = minimax(depth-1, board_clone, alpha, beta)

		// Recalculate beta
		beta = MIN(beta, maxScore)

		// Break if maximising is greater than minimising
		IF beta <= alpha THEN
			break
		// See if a better move was found
		IF score <= minScore THEN
			minScore = score
			best_move = move
	return best_move
```

```
FUNCTION bestMove(board)
	// Random moves if depth level is 0
	IF depth = 0 THEN
		return randomMove(board)

	// Otherwise use the minimax algorithm
	ELSE
		return minimax(depth, board)
```

Structures

Type	Name	Data Type	Explanation	Reason
Attribute	depth	Integer	The number of moves ahead the program is looking ahead in the search tree	Computer must keep track of how far down the tree it has searched, so this value de-increments for every level travelled down
Variable	alpha	Float	The highest-value choice we have found so far at any point along the path	Alpha finds the score which maximises the computer's winning chances. This value is used to cut-off nodes that won't affect the final decision of the "best" move. It will cut-off a node when greater than or equal to the minimising
Variable	beta	Float	The lowest-value choice we have found so far at any point along the path	Beta finds the score which minimises the player's winning chances. This value is used to cut-off nodes that won't affect the final decision of the "best" move. It will cut-off a node when less than or equal to the maximising
Variable	best_move	Object	Stores the best move found so far	Need to store the best move so the function can return it when the search is done
Variable	score	Float	Stores the score of the best move found so far	Algorithm must keep track of the best score so far to compare with other scores found

Planning For Full Game Testing

Structures must be tested at the end of the game to see if everything works together. [redacted] and [redacted] will help carry this out by trying out all the features of the game and achieving different positions on the chess board to see how the game reacts.

Features to Change the Settings

#	Structure being Tested	Test Data	Type	Reasoning
1	board_colour	Click on brown button	Valid	Player should be able to change board colour via buttons on the screen
2	board_colour	Click on grey button	Valid	Player should be able to change board colour via buttons on the screen
3	board_colour	Press “B” key	Invalid	Player shouldn’t be able to change the board colour via the keys
4	player_colour	Click on white button	Valid	Player should be able to change the colour they play as via buttons on the screen
5	player_colour	Click on black button	Valid	Player should be able to change the colour they play as via buttons on the screen
6	player_colour	Press on “W” key	Invalid	Player shouldn’t be able to change the colour they play as via keys
7	AI	Click on “2” button	Valid	Player should be able to change difficulty level via buttons on the screen
8	AI	Press “2” key	Invalid	Player shouldn’t be able to change difficulty level via keys

Features During Gameplay

#	Structure being Tested	Test Data	Type	Reasoning
1	boardClicker()	Click on your pawn then click one space up	Valid	Pieces should only be able to move when clicked to another square
2	boardClicker()	Drag your pawn from one square to another	Invalid	Pieces shouldn't be able to move when dragged on
3	boardClicker()	Click on your pawn and press up arrow	Invalid	Pieces shouldn’t be moved with keys

4	boardClicker()	Click on opponent's pawn and click one square ahead	Invalid	Player shouldn't be able to move opponent's pieces
5	getValidMoves()	Click on pawn to move up one space	Valid	One space up is a valid move for a pawn
6	getValidMoves()	Click on pawn to move right one space	Invalid	One space right is not a valid move for a pawn
7	getValidMoves()	Click on knight to move in an "L" shape	Valid	"L" shape is a valid move for a knight
8	getValidMoves()	Click on knight to move straight	Invalid	Straight direction is not a valid move for a knight
9	getValidMoves()	Click on bishop to move diagonal	Valid	Diagonal moves are valid for a bishop
10	getValidMoves()	Click on bishop to move in an "L" shape	Invalid	"L" shape is not a valid move for a bishop
11	getValidMoves()	Click on rook to move straight	Valid	Straight moves are valid for a rook
12	getValidMoves()	Click on rook to move diagonal	Invalid	Diagonal moves are not valid for the rook
13	getValidMoves()	Click on queen to move straight	Valid	Straight moves are valid for a queen
14	getValidMoves()	Click on queen to move in an "L" shape	Invalid	"L" shape is not valid for a queen
15	getValidMoves()	Click on king to move one space right	Valid	One space right is valid for a king
16	getValidMoves()	Click on king to move two spaces up	Invalid	Two spaces up is not valid for a king
17	isStalemate()	Board position where stalemate is true, and game should output that its stalemate	Valid	Player should be alerted when game ends in stalemate
18	isStalemate()	Board position where stalemate is false, and game should output that	Invalid	Player shouldn't be told game ended in stalemate when board state is not in stalemate

		its stalemate		
19	isCheckmate()	Board position where black is checkmated, and game should output that white won	Valid	Player should be told if white won by checkmate if white won by checkmate
20	isCheckmate()	Board position where white is checkmated, and game should output that black won	Valid	Player should be told if black won by checkmate if black won by checkmate
21	isCheckmate()	Board position where no one is checkmated but game outputs checkmate has occurred	Invalid	Player shouldn't be told game ended in checkmate when board state is not in checkmate
22	minimax()	Level 5 difficulty selected	Extreme	Program should be able to search at the maximum depth level of 5 if user wants it to

Validation

The only way to input information into the game is through clicking on the screen and moving the mouse. This makes it very easy for me to control the outputs the user receives and to implement validation, as the only input I have to deal with is . For instance, the button to turn the board brown will only ever turn the board brown. Iterative and post development testing will aid me in ensuring only valid inputs are made and the expected outcomes occur.

Each button will be tested to see if it only carries out the expected outcomes;

- When the white button is clicked on, the only output should be flipping the player colour to white.
- When the black button is clicked on, the only output should be flipping the player colour to black.
- When the brown button is clicked on, the only output should be turning the board colour brown.
- When the grey button is clicked on, the only output should be turning the board colour grey.

The game must also follow the rules of chess and should therefore only allow valid chess moves for each piece, and only allow the user to move their pieces to those positions. Therefore, testing must be carried out for each piece to make sure that it can only move to its allowed positions only;

- Pawns will move one space in the forward direction, can move two spaces forward on its first move, and capture diagonally forward. When a pawn makes it to the end of the board, then it can be promoted to a queen, rook, bishop or knight. There is a controversial special move for a pawn known as *en passant*, which I will include in case the player would like to play it.

- Knights will move in an “L” shape and are able to move over other pieces. It moves two spaces in a direction and one space in a perpendicular direction.
- Bishops will move diagonally on the board in north-east, north-west, south-east or south-west directions.
- Rooks will move horizontally and vertically through the straights of a board in the north, south, east or west directions.
- Queens will move both like the bishop and rook. It will move in the north, south, east, west, north-east, north-west, south-east or south-west directions.
- The king will move one space in any direction. It also can perform a move known as *castling* where it moves to spaces to the right or left and a rook is placed next to it if both pieces are unmoved and there are no pieces between them. The king should not be able to move into any place where it can be attacked by an enemy piece.

The rules of each piece can be placed into a database or list of possible moves on the board then checked when the user makes a move. If the user attempts to make a move not on the list then it is disallowed, otherwise it is made.

The game can end in one of three ways, and should only output the correct way the game has ended;

- If black is checkmated, the game should output that white wins only.
- If white is checkmated, the game should output that black wins only.
- If the game ends in a draw, the game should show that it ended in stalemate only.

By simulating different board positions, I can test to see if the game outputs the correct outcome when it occurs, and doesn’t output anything when they do not occur.

Client Sign-Off

I asked both [redacted] and [redacted] their thoughts on the design.

- [redacted] was happy that the game follows the rules of classical chess and said that he especially liked options of the board colours.. I asked if there were any other features that could be added to make the game more user friendly. He suggested colour coding each of the buttons based on its purpose so that is clear what the functionality of each button is. I liked the idea and will implement this into the game as per [redacted]’s request.

- [redacted] liked the layout of the screen and that the pieces are able to move correctly. However, he was unhappy with the fact that the *en passant* move was being included in the game as he said that “it isn’t allowed in some tournaments, so there isn’t a point in me training with it.” I asked [redacted] whether it was alright if I removed *en passant* from the game and he said yes, because he didn’t even know that it was a valid move in the game. Therefore, *en passant* won’t be in the final game.

[redacted] and [redacted]’s dated their signatures here to show approval for the design and agreement to play the game to help with post development testing;

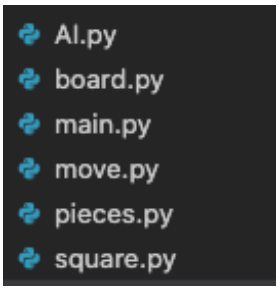
[redacted] ***- 30/11/2022***
[redacted] ***- 02/12/2022***

Development

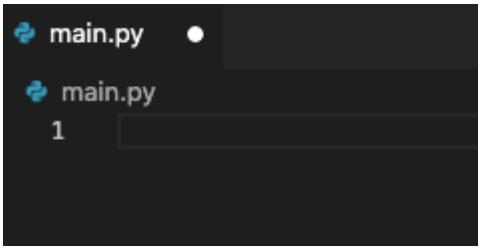
Setting Up Workspace - 19/12/2022

The code is divided into six python documents in order to keep the project organised; main, board, move, piece, square and AI. This is because the modular nature of the solution will require the use of Object Oriented Programming, and numerous documents will aid in keeping each class separate and keeping the program organised.

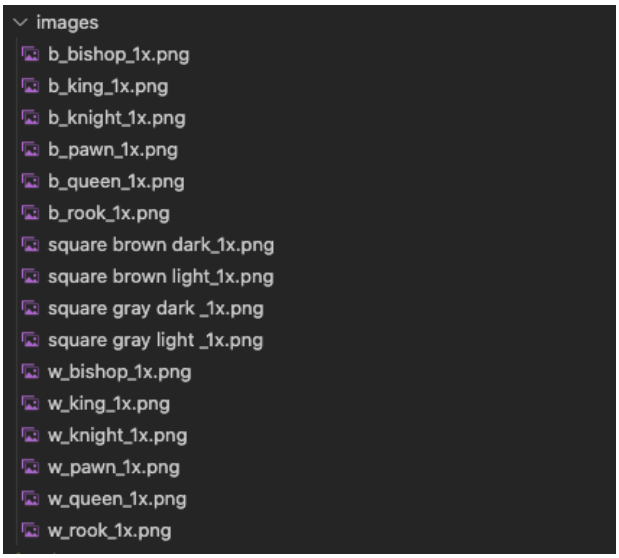
The first step is to create each document and place it into the VS Code workplace.



Each python document can open and is currently blank, as seen in main.py.



Next, I need to bring in the images for the game into the workspace. There are 16 images total in the game, an image of the six types of pieces as white, an image for each type of piece as black, and four images of squares to draw the board. The images are saved in a png format. The dimension of each image is 1 x 1 pixel. (all images are copyright free from shutterstock.com)



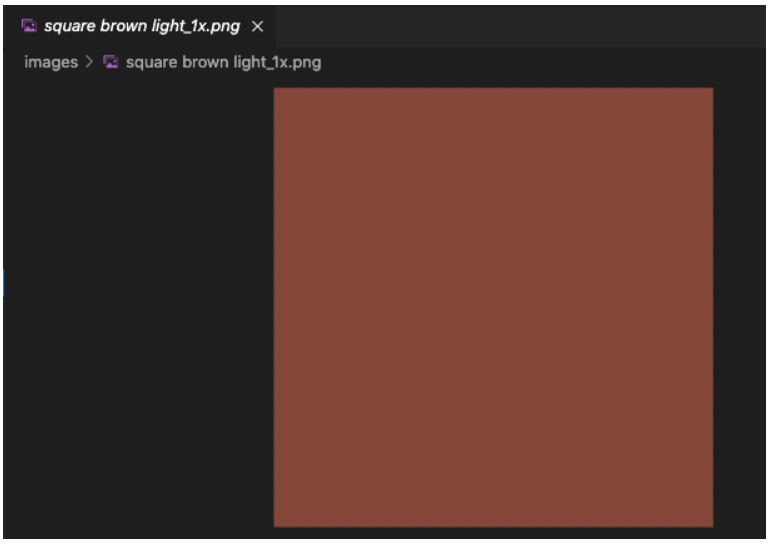
The ability to access an image in the workspace is tested with the white and black bishops to make sure the pieces of both colours can be accessed.

All the other pieces were tested and they can all be accessed.

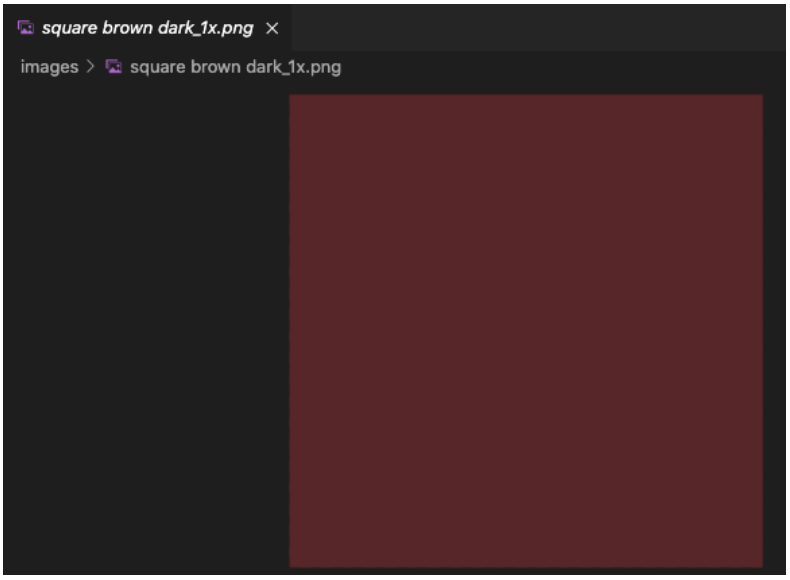
Four images of squares are needed because [redacted] and [redacted] requested to have the functionality of choosing the colour scheme of the board. Hence two squares will represent a brown scheme while the other two will represent a grey scheme.

The light and dark squares of both colour schemes are tested to see if they can be accessed.

Light brown square;

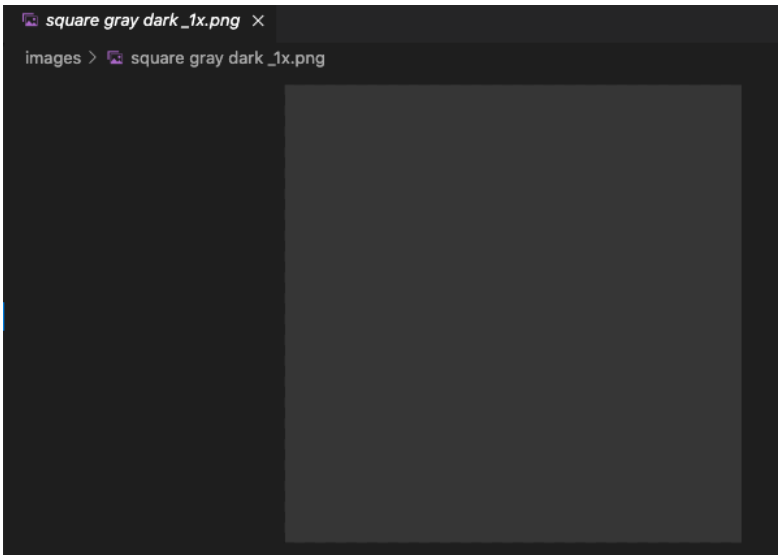


Dark brown square;



Light grey square;

Dark grey square;



The test is successful for all the squares.

Testing If Can Access Workspace (1)

What's being tested	Inputs	Reasoning	Expected Outcome	Actual Outcome
Images can be accessed in workspace	None	Images need to be able to be accessible to start the project	Each image can be accessed in workspace	Each image is accessed in workspace

The test was successful.

Client Feedback

I showed the images and assets to [redacted] and [redacted] who are very happy with the images of pieces and squares being used in the project. They said they reminded them of classical chess pieces.

Setting Up Pygame and Prototyping the Screen - 19/12/2022

Pygame is a set of Python modules designed for writing video games. It provides extremely useful functions which can help me create a game screen, place images on the screen and more. Hence why I am using it to create my game.

To use these modules, I must first import pygame into main.py, then call the *init()* function to activate it.

```
main.py > ...
1   import pygame
2
3   pygame.init()
```

init() will initialise all imported pygame modules.

The dimensions of the screen are going to be 800 x 600 pixels to comfortably fit on [redacted]'s and [redacted]'s screen, but also large enough to comfortably fit a board and menu together. The width and height of the screen are stored in the variable *WIDTH* and *HEIGHT* respectively.

```
# Create Screen
WIDTH = 800
HEIGHT  = 600
```

We can create a screen object called *screen* by passing *WIDTH* and *HEIGHT* as a tuple in the pygame function below.

```
WIDTH = 800
HEIGHT  = 600
screen = pygame.display.set_mode((WIDTH, HEIGHT))
```

After the screen is created, the game loop needs to be set up. A game loop is needed to process inputs, update the game state and render the game to the screen. The game loop must exist in main.py as it's purpose is to control the flow of the game.

A variable which is called *running* is set to True to indicate that the game is running.

```
running = True
```

As long as *running* is true, the game will continue.

```
while running:
    pass
```

Before using the game loop to show our screen, we must add a break clause in the game loop in order for a user to exit the game screen and quit the game. To do this, we must loop over every pygame event and check to see if the user has tried to quit the game. If this is true, then we call the *quit()* and *exit()* function in pygame to leave the game.

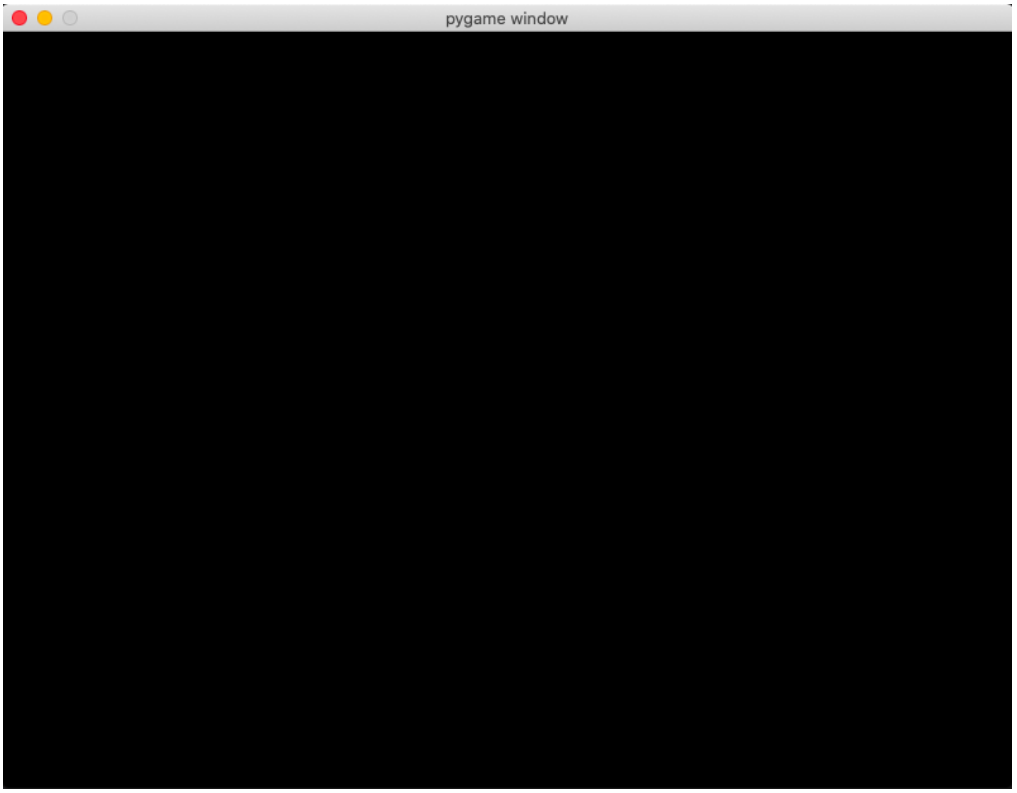
```
while running:

    for event in pygame.event.get():

        if event.type == pygame.QUIT:

            pygame.quit()
            exit()
```

At the end of the game loop we call the following pygame function to make sure our screen can be updated based on the state of the game in every iteration of the game loop.

```
pygame.display.update()
```

Testing If Screen Appears and If Can Exit Screen (2)

The game so far is run to test to see if a screen appears.



A screen called “pygame window” appears, which is 800 x 600 pixels.

What's being tested	Inputs	Reasoning	Expected Outcome	Actual Outcome
If game screen appears on computer and game loop	Width and Height of screen	Need a game screen to play game	Screen appears on computer	Screen appeared on computer

The test was successful.

Next I tested if the window can close with the red exit button on the top left corner.

What's being tested	Inputs	Reasoning	Expected Outcome	Actual Outcome
If screen will close	Exit button	Should be able to close/quit game	Screen closes	Screen closed

The test was a success.

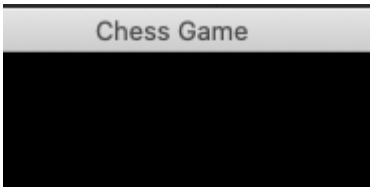
More Prototyping with The Screen- 20/12/2022

I decided that I didn't like the screen to be called "pygame window", so I added the following code before the game loop changing the caption of the screen.

```
pygame.display.set_caption("Chess Game")
```

Testing Screen Caption (3)

The game is run.



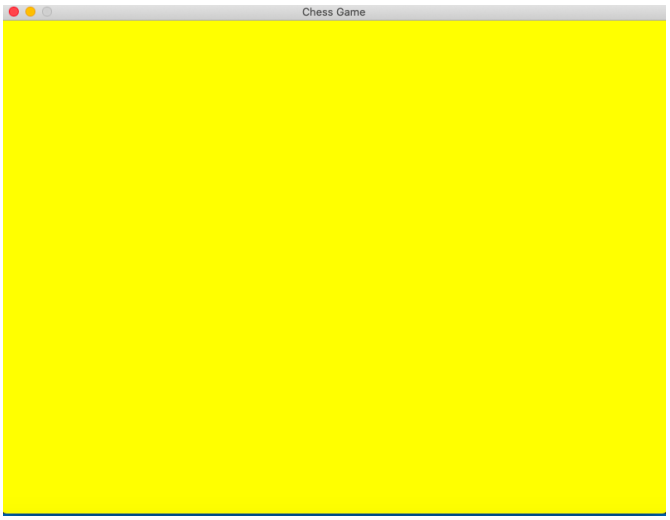
What's being tested	Inputs	Reasoning	Expected Outcome	Actual Outcome
Screen caption	Name of screen caption	Screen caption should be called something relevant to the game	Screen caption is "Chess Game"	Screen caption is "Chess Game"

The test was a success.

Testing Screen Colour (4)

The background of the game should be yellow as per the request of [redacted]. So the numbers 255, 255 and 0 are passed through the *fill()* function of the *screen* (in the order given) to produce a yellow background for the screen. This is the hexadecimal colour code for yellow.

```
screen.fill((255, 255, 0))
```



What's being tested	Inputs	Reasoning	Expected Outcome	Actual Outcome
Colour of screen	(255, 255, 0) - hexadecimal representation for yellow	Screen caption should be called something relevant to the game	Yellow background	Yellow background

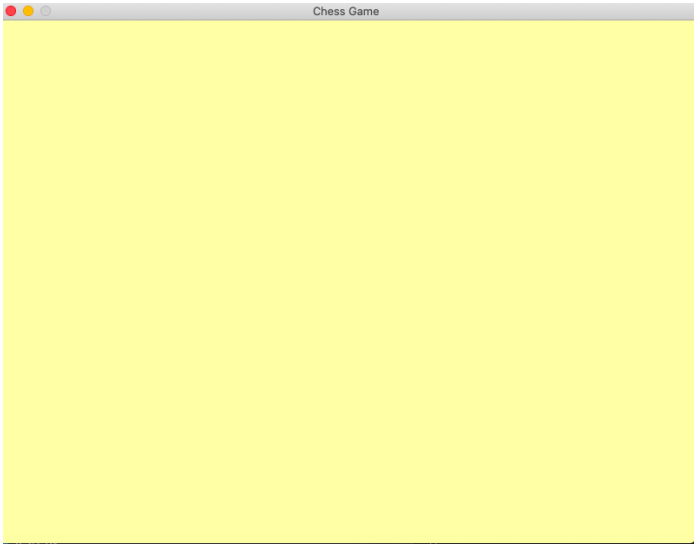
The test was a success.

Client Feedback

[redacted] saw the yellow background and disliked it. He said that it was too bright and intense on his eyes, so requested it to be changed to a lighter shade of yellow.

In response, I adjusted the hexadecimal code of the fill.

```
screen.fill((255, 255, 165))
```



[redacted] approved of this shade of yellow which is less intense on his eyes.

Making The Square Class - 23/12/2022

A chess board is made up of 64 squares with 8 rows and 8 columns. In square.py, we will have a class called *Square,* which will act as a template for all the squares on the board.

The first thing we do is import pygame and create the class.

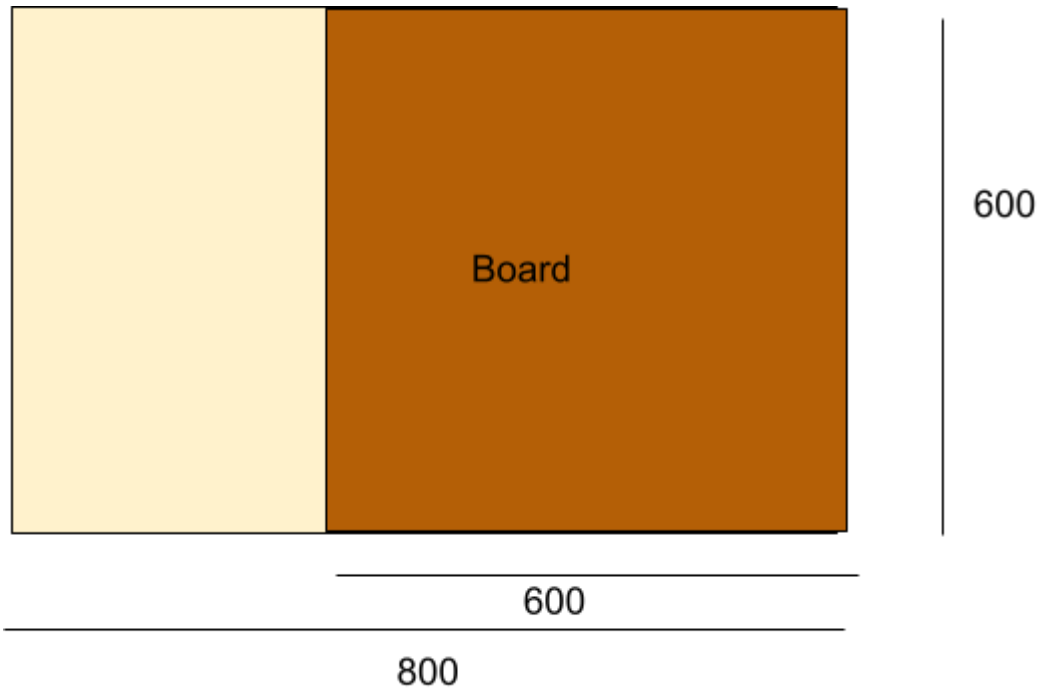
```
import pygame


class Square():
```

The squares on the board alternate in between dark and light colours, with the square in the right corner being light. Therefore the colour of each square is dependent on its position and the colour scheme of the board (either brown or grey).

When we create an instance of the square class, we want to take in the board colour and square position to determine its colour.

```
def __init__(self, board_colour, square_pos):
```

Below is the planned layout of the screen;



The dimension of the actual board is 600 x 600 pixels. We must fit 64 squares onto the board (which is 8 squares per row and 8 squares per column).

To calculate the dimensions of each square;

600 pixels / 8 squares (in a row or column) = 75 pixels per square

Hence each square must have dimensions of 75 x 75 pixels.

When we instantiate a new square, we want to load the images of all the squares and scale them up to be 75 x 75 pixels (as they are originally 1 x 1 pixel).

The below code first loads the image file named square brown dark_1x.png from a folder called "images". This image is loaded using the *image.load()* function and is then converted to an alpha surface format using the convert_alpha() method.

Next, the code scales up the image using the *transform.scale()* function, which takes two arguments: the first is the surface to be scaled (in this case, square_brown_dark), and the second is a tuple specifying the new dimensions of the surface after scaling (in this case, 75 pixels in width and 75 pixels in height).

```
# Load and scale up squares
square_brown_dark = pygame.image.load('/images/square brown dark_1x.png').convert_alpha()
square_brown_dark = pygame.transform.scale(square_brown_dark, (75, 75))
```

The same is done for the other three squares.

```
# Load and scale up squares
square_brown_dark = pygame.image.load('/images/square brown dark_1x.png').convert_alpha()
square_brown_dark = pygame.transform.scale(square_brown_dark, (75, 75))
square_brown_light = pygame.image.load('/images/square brown light_1x.png').convert_alpha()
square_brown_light = pygame.transform.scale(square_brown_light, (75, 75))
square_gray_dark = pygame.image.load('/images/square gray dark _1x.png').convert_alpha()
square_gray_dark = pygame.transform.scale(square_gray_dark, (75, 75))
square_gray_light = pygame.image.load('/images/square gray light _1x.png').convert_alpha()
square_gray_light = pygame.transform.scale(square_gray_light, (75, 75))
```

The expected input of the argument *square_pos* is a tuple with two numbers, where the first is the x-coordinate and the second is the y-coordinate. The attributes *x, y,* and *pos* are dependent on *square_pos*.

```
# Square position
self.x = square_pos[0]
self.y = square_pos[1]
self.pos = square_pos
```

If the square is located in the right hand corner, then x and y coordinates will be 7 and 7 respectively. We want this square to be a light colour. If each square alternates in colour then the coordinates of the light squares will be both even or both odd, meaning that the sum of x and y will always be even.

Using this logic, the condition below determines the square attribute of the square.

```
if (self.x % 2 == self.y % 2):
    # Is light if sum of x and y is even
    if board_colour == "brown":
        self.square = square_brown_light
    elif board_colour == "gray":
        self.square = square_gray_light
else:
    # Otherwise is dark
    if board_colour == "brown":
        self.square = square_brown_dark
    elif board_colour == "gray":
        self.square = square_gray_dark
```

The condition checks if the sum of the x and y coordinates of the tile is an even number, which would mean that the tile is part of the light squares on the chessboard. If this condition is true, it assigns a light square image to *square*. If the condition is false, it assigns a dark square image to the same variable.

The board_colour variable is used to determine whether the chessboard is brown or grey, and this is used to assign the appropriate coloured square image. If *board_colour* is "brown", the code assigns

one of two brown-coloured square images to *square*, depending on whether the tile is light or dark. If *board_colour* is "gray", the code assigns one of two grey-coloured square images to *square*.

The final attribute of the *Square* class is *piece_on_square* which stores the current piece on the square and has the default value of None.

```
self.piece_on_square = None
```

The class has the getter methods *getSquare()*, *getPosition()* and *getPieceOnSquare()*, which return the *square*, *pos* and *piece_on_square* attributes respectively.

```
def getSquare(self):
    return self.square

def getPosition(self):
    return self.pos

def getPieceOnSquare(self):
    return self.piece_on_square
```

The class has one setter method called *setPieceOnSquare(),* which takes in one parameter called *piece*. It will set the attribute *piece_on_square* to the value of *piece.*

```
def setPieceOnSquare(self, piece):

    self.piece_on_square = piece
```

Testing Square Getter Methods (5)

By importing the *Square* class into main and creating a temporary square object called *s* we can test the getter methods of the *Square* class.

```
from square import Square
```

```
s = Square(board_colour="brown", square_pos=(0, 0))
```

We can then print the outputs of *getPosition()* and *getPieceOnSquare()* to see if it matches with the expected output.

```
print(s.getPosition(), s.getPieceOnSquare())
```

Output;

```
Hello from the pygame community. https://www.pygame.org/contribute.html
(0, 0) None
```

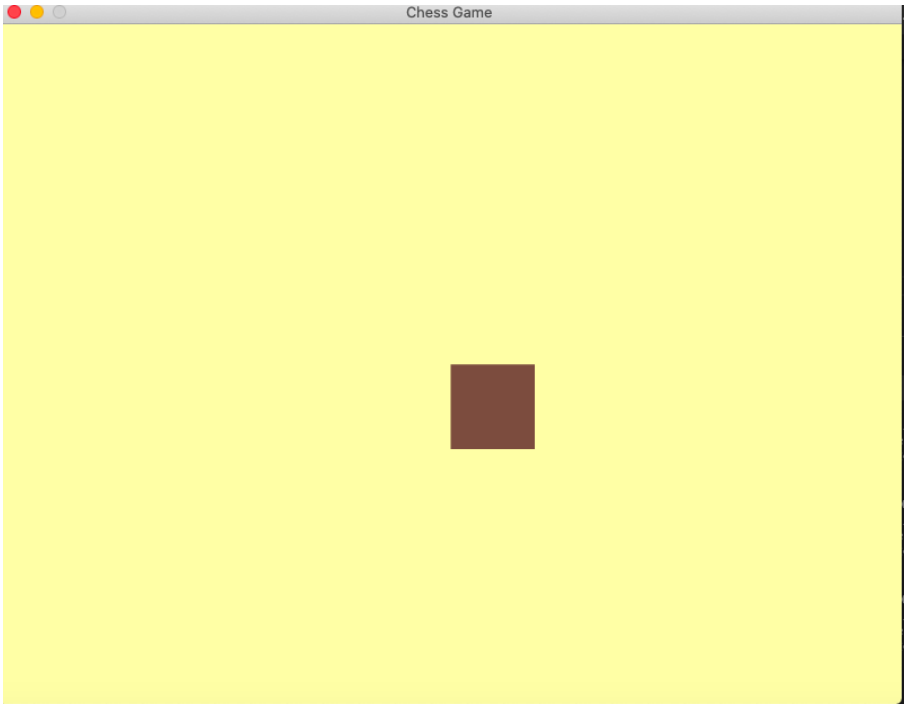
What's being tested	Inputs	Reasoning	Expected Outcome	Actual Outcome
getPosition() and *getPieceOnSquare()* of class *Square*	Position (0, 0) and piece 'None'	Ensure functions output attributes for later use	Outputs (0, 0) and 'None' to the terminal	Outputs (0, 0) and None to the terminal

The test was a success.

To test *getSquare()*, we will use the *blit()* function of screen to place the square at coordinates (400, 300).

```
while running:

    screen.fill((255, 255, 165))
    screen.blit(s.getSquare(), (400, 300))
```

Output;



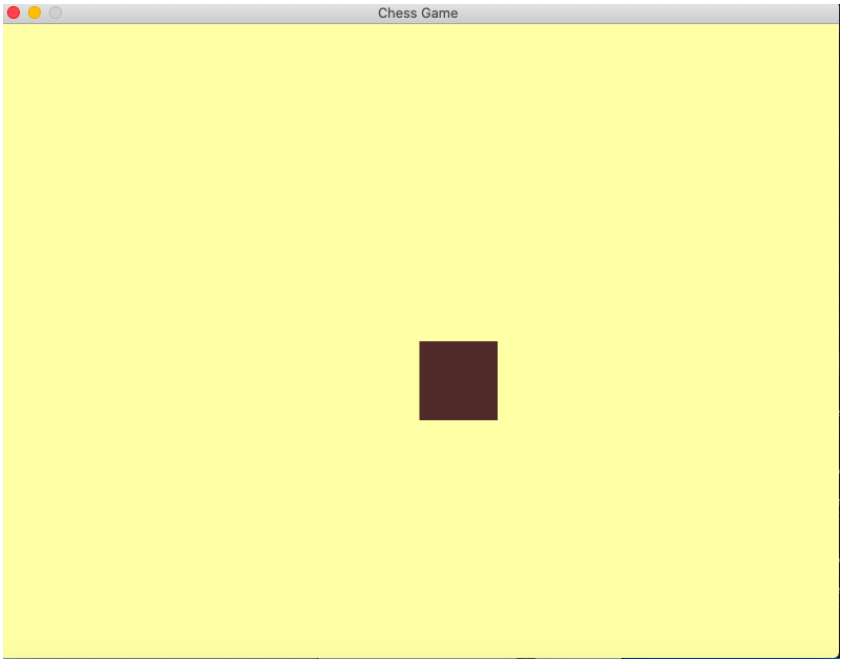
What's being tested	Inputs	Reasoning	Expected Outcome	Actual Outcome
getSquare() of class *Square*	Position (0, 0), board colour "brown" and coordinates (400, 300)	Make sure each square can blit to screen before drawing board	Light brown square at coordinates (400, 300) of size 75 x 75	Light brown square at coordinates (400, 300) of size 75 x 75

The test was a success.

Adjusting the position of the *s* to (1, 0) should turn the square into a dark brown square.

```
s = Square(board_colour="brown", square_pos=(1, 0))
```

Output;



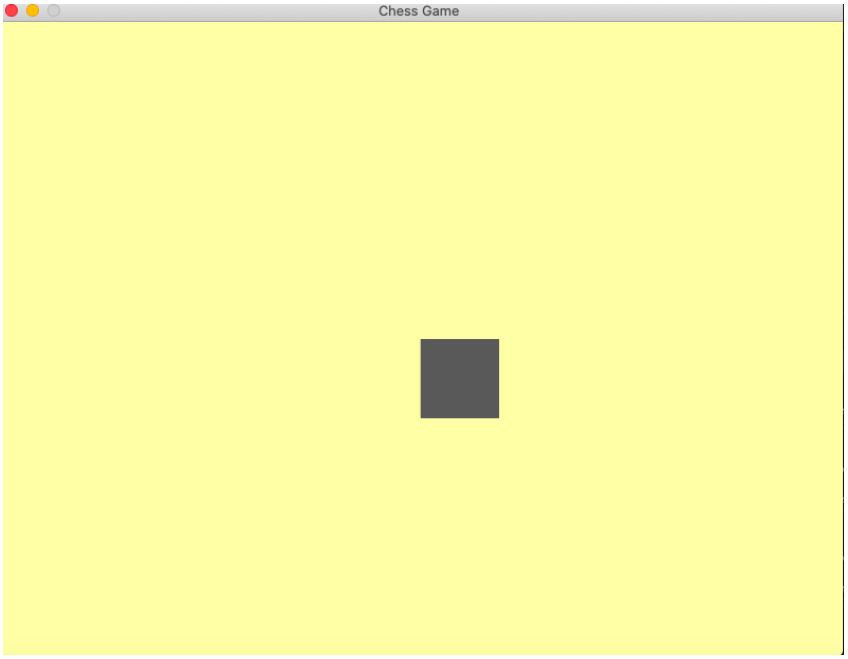
What's being tested	Inputs	Reasoning	Expected Outcome	Actual Outcome
getSquare() of class *Square*	Position (1, 0), board colour "brown" and coordinates (400, 300)	Make sure each square can blit to screen before drawing board	Dark brown square at coordinates (400, 300) of size 75 x 75	Dark brown square at coordinates (400, 300) of size 75 x 75

The test was a success

Changing *board_colour* to "gray" and coordinate to (0, 0) should change the square to a light grey square.

```
s = Square(board_colour="gray", square_pos=(0, 0))
```

Output;



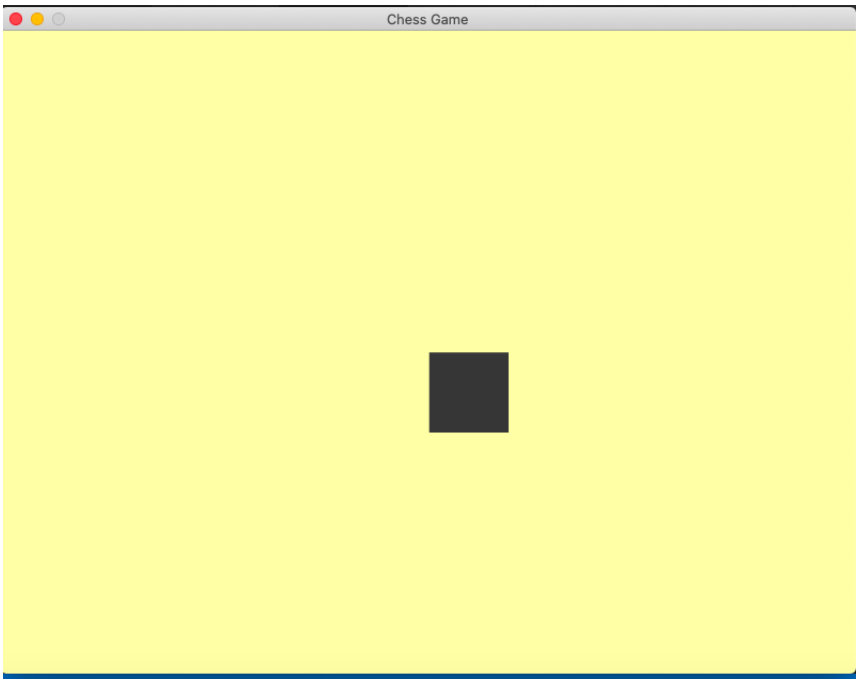
What's being tested	Inputs	Reasoning	Expected Outcome	Actual Outcome
getSquare() of class *Square*	Position (0, 0), board colour "gray" and coordinates (400, 300)	Make sure each square can blit to screen before drawing board	Light grey square at coordinates (400, 300) of size 75 x 75	Light grey square at coordinates (400, 300) of size 75 x 75

The test was a success.

Adjusting the position of the *s* to (1, 0) should turn the square into a dark grey square.

```
s = Square(board_colour="gray", square_pos=(1, 0))
```

Output;



Making The Class For The Pieces- 24/12/2022

Each piece class inherits from a parent class called *Piece*. This class exists in pieces.py. Pygame is imported to deal with the images

```
import pygame

class Piece():

    def __init__(self, colour, positionX, positionY, pieceType, value):
        self.colour = colour
        self.positionX  = positionX
        self.positionY = positionY
        self.pieceType = pieceType
        self.value = value
        self.validMoves = []
        self.moved = False
        self.image = None
```

The constructor takes five arguments: *colour*, *positionX*, *positionY*, *pieceType*, and *value*.

- *colour*: the *colour* of the piece, either "white" or "black"
- *positionX*: the x-coordinate of the piece on the chessboard (the column)
- *positionY*: the y-coordinate of the piece on the chessboard (the row)

- *pieceType*: the type of the piece, such as "rook" or "knight"
- *value*: the point value of the piece in the game of chess (for example, a queen is worth 90 points)
- *validMoves*: a list of valid moves for the piece
- *moved*: a boolean that represents whether or not the piece has moved. This is used to determine if the piece is eligible for certain moves, such as castling.
- *image*: a variable that stores the image of the piece

The piece class also has the following methods;

```
def addMove(self, move):
    self.validMoves.append(move)

def getColour(self):
    return self.colour

def getType(self):
    return self.pieceType

def getPositionX(self):
    return self.positionX

def getPositionY(self):
    return self.positionY

def getPos(self):
    return (self.positionX, self.positionY)

def getValue(self):
    return self.value
```

```
def getImage(self):
    return self.image

def getMoved(self):
    return self.moved

def getValidMoves(self, board):
    return self.validMoves

def setMovedTrue(self):
    self.moved = True

def setPos(self, pos):
    self.positionX = pos[0]
    self.positionY = pos[1]
```

Here's what each method is supposed to do;

- *addMove()*: takes in the parameter *move* and adds it to the *validMoves* list
- *getColour():* returns the colour of the piece
- *getType()*: returns the type of the piece
- *getPositionX()*: returns the x-coordinate of the piece
- *getPositionY()*: returns the y-coordinate of the piece
- *getPos()*: returns the the position of the piece on the board
- *getValue()*: returns the value of the piece
- *getImage()*: returns the image of the piece
- *getMoved():* returns True is the piece has moved at least once, otherwise returns False
- *getValidMoves()*: returns the list of valid moves for that piece
- *setMovedTrue()*: sets the moved attribute of the piece to True. It is called when the piece is moved for the first time.
- *setPos()*: sets the current position of the piece on the chess board to the given position. It takes a tuple called *pos* as an argument.

Each of the 6 chess pieces have their own class which inherits from the *Piece* parent class. This is to set unique properties to each piece such as their movement or image. The *getValidMoves()* function in each subclass will overwrite the function in pieces as each piece moves uniquely. Each image is scaled up to 75 x 75 pixels to fit on the square.

In *Pawn,* the default type is "pawn" and initial value is set to 10. If its colour is "white" the white pawn image is loaded, otherwise the black pawn image is loaded.

```
class Pawn(Piece):

    def __init__(self, colour, positionX, positionY):
        super().__init__(colour, positionX, positionY, "pawn", 10)

        if self.colour == "white":
            image = pygame.image.load('/images/w_pawn_1x.png')
            self.image = pygame.transform.scale(image, (75, 75))

        else:
            image = pygame.image.load('/images/b_pawn_1x.png')
            self.image = pygame.transform.scale(image, (75, 75))

        self.direction = None

    def getValidMoves(self, board):

        # Code for movement will go here

        return self.validMoves
```

In *Knight,* the default type is “knight” and initial value is set to 30. If its colour is “white” the white pawn image is loaded, otherwise the black pawn image is loaded.

```
class Knight(Piece):

    def __init__(self, colour, positionX, positionY):
        super().__init__(colour, positionX, positionY, "knight", 30)

        if self.colour == "white":
            image = pygame.image.load('/images/w_knight_1x.png')
            self.image = pygame.transform.scale(image, (75, 75))

        else:
            image = pygame.image.load('/images/b_knight_1x.png')
            self.image = pygame.transform.scale(image, (75, 75))

    def getValidMoves(self, board):

        # Code for movement will go here

        return self.validMoves
```

In *Bishop,* the default type is “bishop” and initial value is set to 30.01 (as bishops are worth slightly more than knights). If its colour is “white” the white pawn image is loaded, otherwise the black pawn image is loaded.

```
class Bishop(Piece):

    def __init__(self, colour, positionX, positionY):
        super().__init__(colour, positionX, positionY, "bishop", 30.01)

        if self.colour == "white":
            image = pygame.image.load('/images/w_bishop_1x.png')
            self.image = pygame.transform.scale(image, (75, 75))

        else:
            image = pygame.image.load('/images/b_bishop_1x.png')
            self.image = pygame.transform.scale(image, (75, 75))

    def getValidMoves(self, board):

        # Code for movement will go here

        return self.validMoves
```

In *Rook,* the default type is “rook” and initial value is set to 50. If its colour is “white” the white pawn image is loaded, otherwise the black pawn image is loaded.

```
class Rook(Piece):

    def __init__(self, colour, positionX, positionY):
        super().__init__(colour, positionX, positionY, "rook", 50)

        if self.colour == "white":
            image = pygame.image.load('/images/w_rook_1x.png')
            self.image = pygame.transform.scale(image, (75, 75))

        else:
            image = pygame.image.load('/images/b_rook_1x.png')
            self.image = pygame.transform.scale(image, (75, 75))



    def getValidMoves(self, board):

        # Code for movement will go here

        return self.validMoves
```

In *Queen,* the default type is “queen” and initial value is set to 90. If its colour is “white” the white pawn image is loaded, otherwise the black pawn image is loaded.

```
class Queen(Piece):

    def __init__(self, colour, positionX, positionY):
        super().__init__(colour, positionX, positionY, "queen", 90)

        if self.colour == "white":
            image = pygame.image.load('/images/w_queen_1x.png')
            self.image = pygame.transform.scale(image, (75, 75))

        else:
            image = pygame.image.load('/images/b_queen_1x.png')
            self.image = pygame.transform.scale(image, (75, 75))

    def getValidMoves(self, board):

        # Code for movement will go here

        return self.validMoves
```

In *King,* the default type is “king” and initial value is set to 999,999 (an arbitrarily large number to represent infinity). If its colour is “white” the white pawn image is loaded, otherwise the black pawn image is loaded.

```
class King(Piece):

    def __init__(self, colour, positionX, positionY):
        # value of king is winning the game so king value is infinity
        super().__init__(colour, positionX, positionY, "king", 999999)

        if self.colour == "white":
            image = pygame.image.load('/images/w_king_1x.png')
            self.image = pygame.transform.scale(image, (75, 75))

        else:
            image = pygame.image.load('/images/b_king_1x.png')
            self.image = pygame.transform.scale(image, (75, 75))

    def getValidMoves(self, board):

        # Code for movement will go here

        return self.validMoves
```

Testing If Pieces Appear On Screen (6)

By importing the classes in pieces.py into main.py and creating individual piece objects to ‘bilt’ to the screen (similar to what we did with the squares), we can test to see if each image of each piece can blit to the screen.

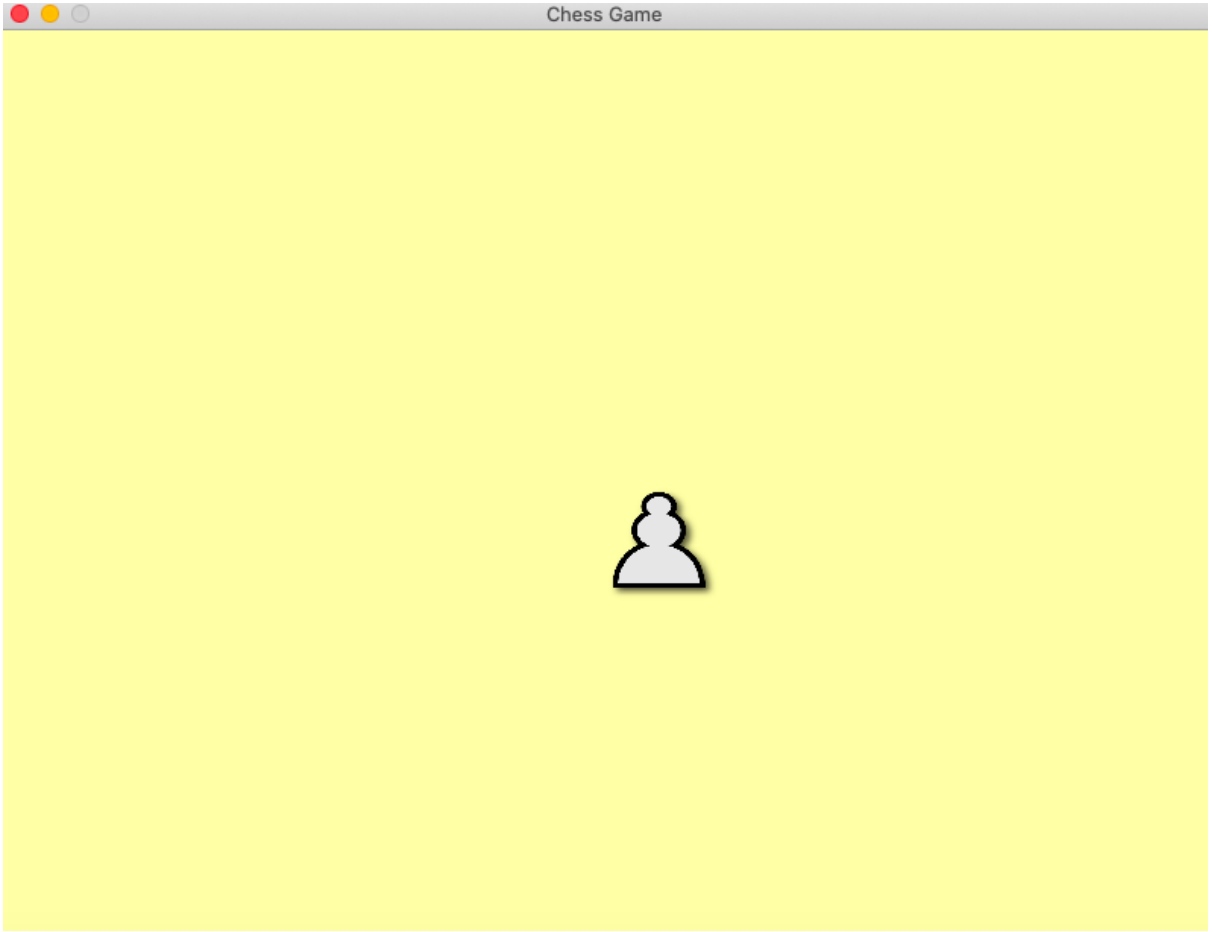
```
from pieces import *
```

For a white pawn;

```
p = Pawn("white", 0, 0)
```

```
while running:

    screen.fill((255, 255, 165))
    screen.blit(p.getImage(), (400, 300))
```

Output;

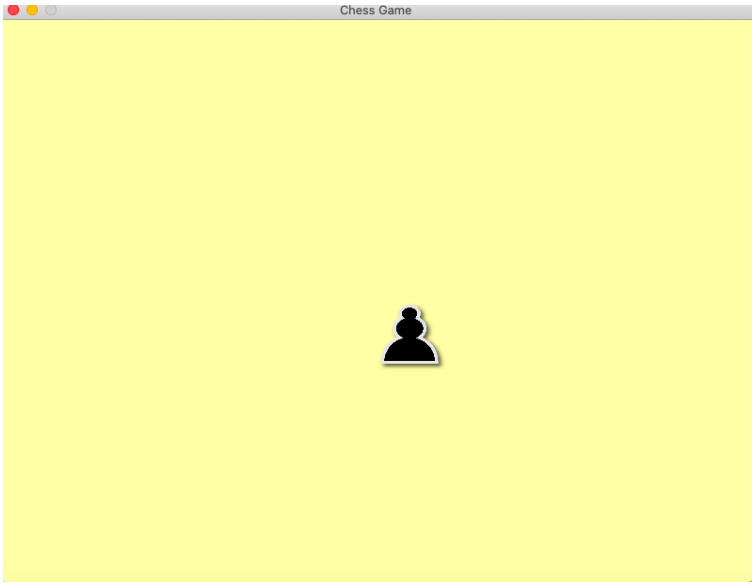


What's being tested	Inputs	Reasoning	Expected Outcome	Actual Outcome
If white pawn appears on screen	Colour "white", position (0, 0) and coordinates (400, 300)	Must see if each piece appears on screen to before I can setup board	White pawn image on screen at coordinate (400, 300)	White pawn image on screen at coordinate (400, 300)

For a black pawn

```
p = Pawn("black", 0, 0)
```

Output;

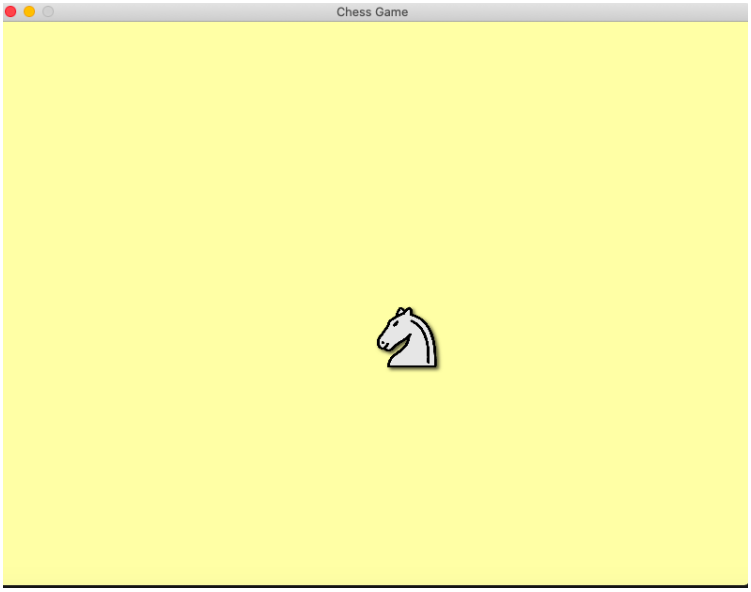


What's being tested	Inputs	Reasoning	Expected Outcome	Actual Outcome
If black pawn appears on screen	Colour "black", position (0, 0) and coordinates (400, 300)	Must see if each piece appears on screen to before I can setup board	Black pawn image on screen at coordinate (400, 300)	Black pawn image on screen at coordinate (400, 300)

For a white knight;

```
p = Knight("white", 0, 0)
```

Output;

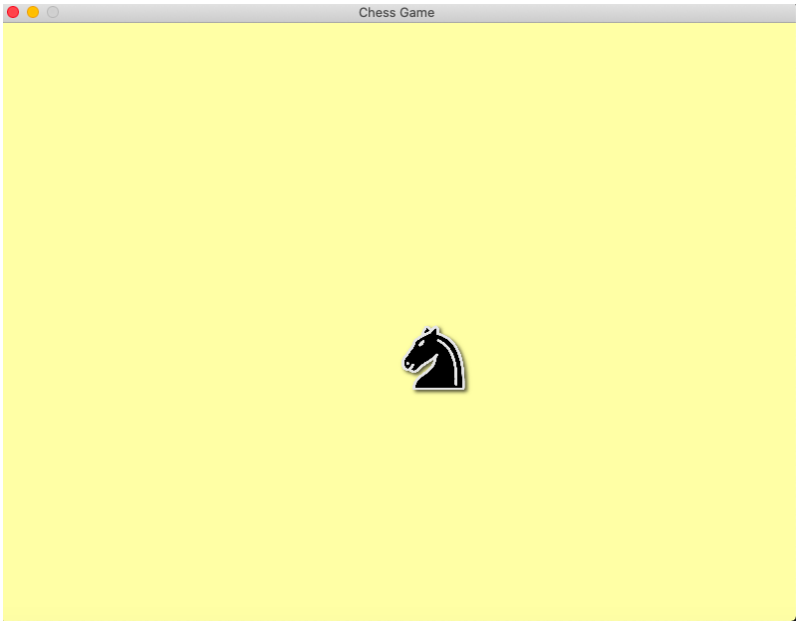


What's being tested	Inputs	Reasoning	Expected Outcome	Actual Outcome
If white knight appears on screen	Colour "white", position (0, 0) and coordinates (400, 300)	Must see if each piece appears on screen to before I can setup board	White knight image on screen at coordinate (400, 300)	White knight image on screen at coordinate (400, 300)

For a black knight;

```
p = Knight("black", 0, 0)
```

Output;

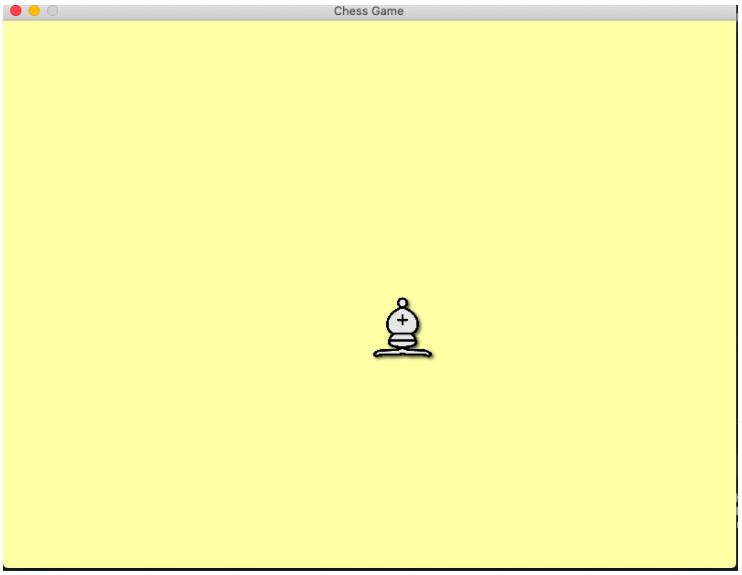


What's being tested	Inputs	Reasoning	Expected Outcome	Actual Outcome
If black knight appears on screen	Colour "black", position (0, 0) and coordinates (400, 300)	Must see if each piece appears on screen to before I can setup board	Black knight image on screen at coordinate (400, 300)	Black knight image on screen at coordinate (400, 300)

For a white bishop;

```
p = Bishop("white", 0, 0)
```

Output;

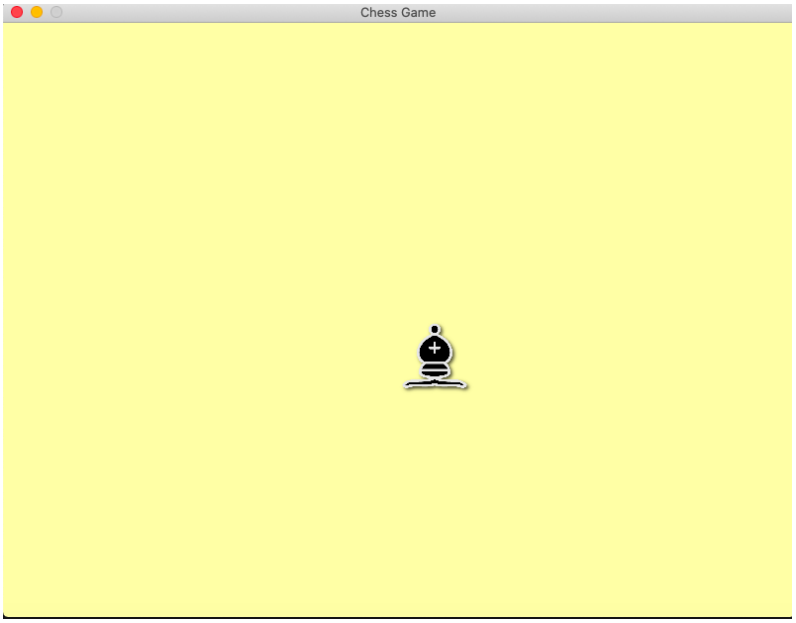


What's being tested	Inputs	Reasoning	Expected Outcome	Actual Outcome
If white bishop appears on screen	Colour "white", position (0, 0) and coordinates (400, 300)	Must see if each piece appears on screen to before I can setup board	White bishop image on screen at coordinate (400, 300)	White bishop image on screen at coordinate (400, 300)

For a black bishop;

```
p = Bishop("black", 0, 0)
```

Output;

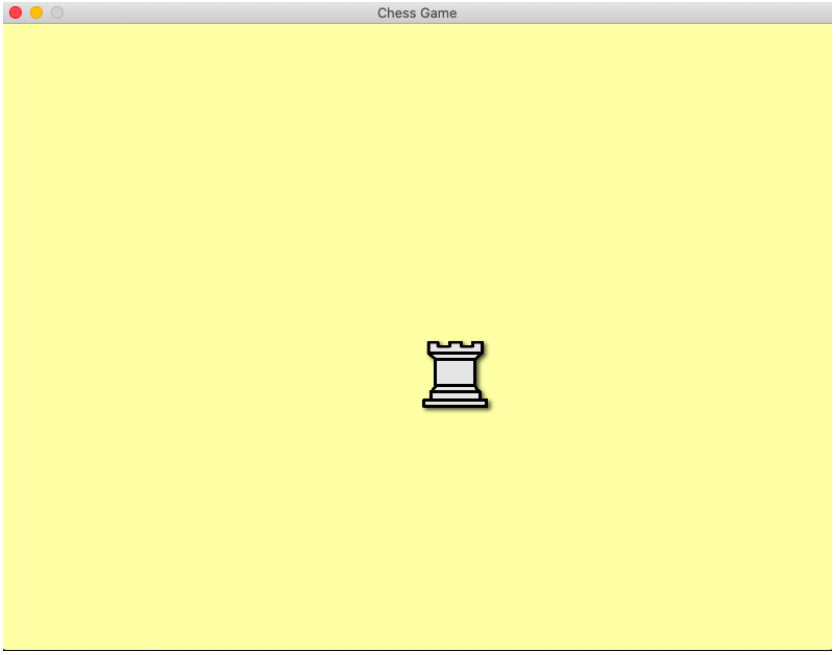


What's being tested	Inputs	Reasoning	Expected Outcome	Actual Outcome
If black bishop appears on screen	Colour "black", position (0, 0) and coordinates (400, 300)	Must see if each piece appears on screen to before I can setup board	Black bishop image on screen at coordinate (400, 300)	Black bishop image on screen at coordinate (400, 300)

For a white rook;

```
p = Rook("white", 0, 0)
```

Output;

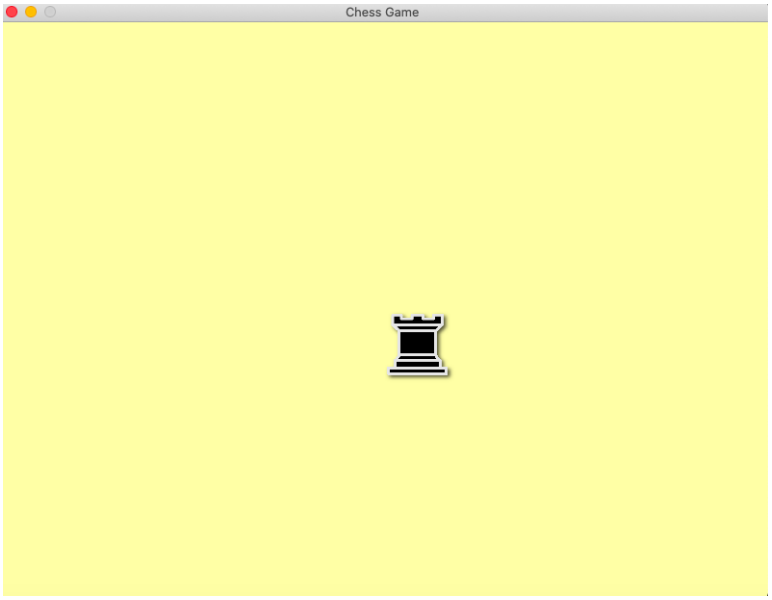


What's being tested	Inputs	Reasoning	Expected Outcome	Actual Outcome
If white rook appears on screen	Colour "white", position (0, 0) and coordinates (400, 300)	Must see if each piece appears on screen to before I can setup board	White rook image on screen at coordinate (400, 300)	White rook image on screen at coordinate (400, 300)

For a black rook;

```
p = Rook("black", 0, 0)
```

Output;

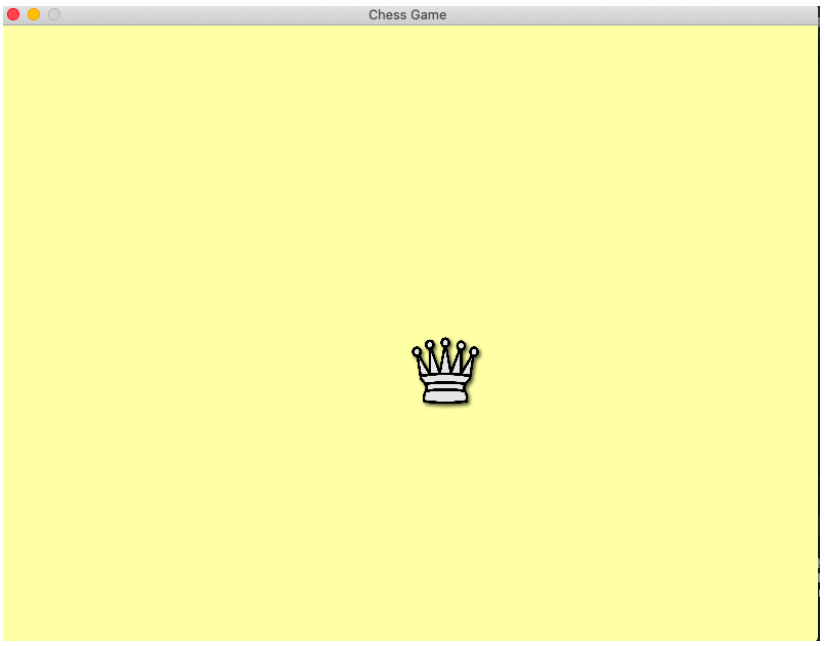


What's being tested	Inputs	Reasoning	Expected Outcome	Actual Outcome
If black rook appears on screen	Colour "black", position (0, 0) and coordinates (400, 300)	Must see if each piece appears on screen to before I can setup board	Black rook image on screen at coordinate (400, 300)	Black rook image on screen at coordinate (400, 300)

For a white queen;

```
p = Queen("white", 0, 0)
```

Output;

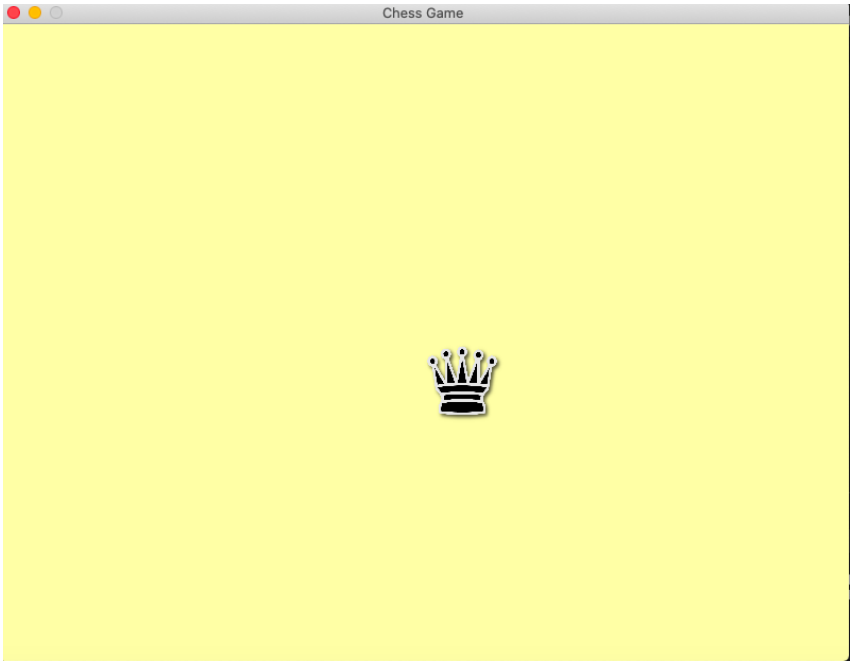


What's being tested	Inputs	Reasoning	Expected Outcome	Actual Outcome
If white queen appears on screen	Colour "white", position (0, 0) and coordinates (400, 300)	Must see if each piece appears on screen to before I can setup board	White queen image on screen at coordinate (400, 300)	White queen image on screen at coordinate (400, 300)

For a black queen;

```
p = Queen("black", 0, 0)
```

Output;

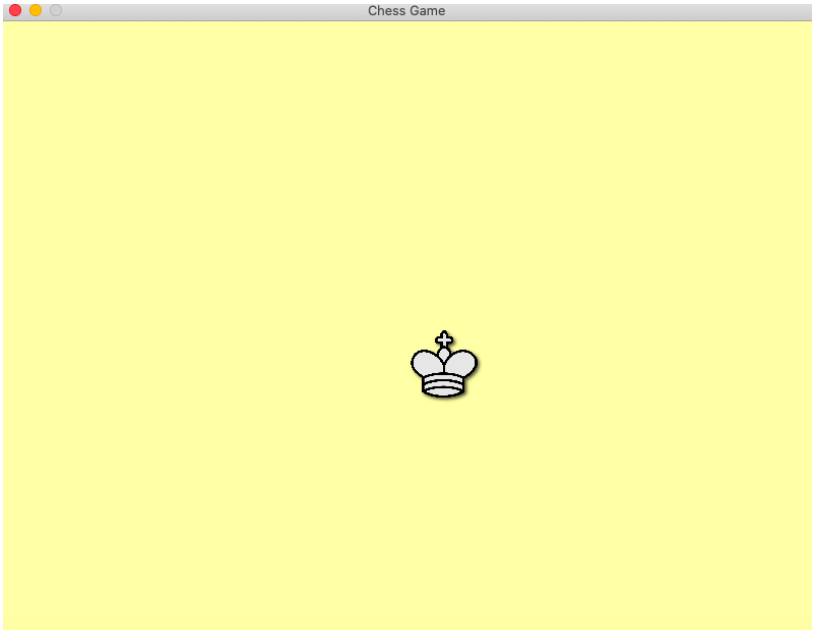


What's being tested	Inputs	Reasoning	Expected Outcome	Actual Outcome
If black queen appears on screen	Colour "black", position (0, 0) and coordinates (400, 300)	Must see if each piece appears on screen to before I can setup board	Black queen image on screen at coordinate (400, 300)	Black queen image on screen at coordinate (400, 300)

For a white king;

```
p = King("white", 0, 0)
```

Output;

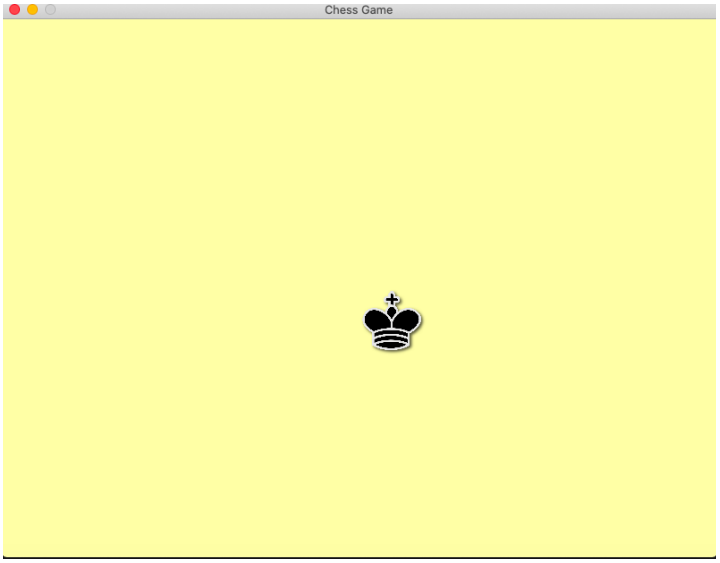


What's being tested	Inputs	Reasoning	Expected Outcome	Actual Outcome
If white king appears on screen	Colour "white", position (0, 0) and coordinates (400, 300)	Must see if each piece appears on screen to before I can setup board	White king image on screen at coordinate (400, 300)	White king image on screen at coordinate (400, 300)

For a black king;

```
p = King("black", 0, 0)
```

Output;



What's being tested	Inputs	Reasoning	Expected Outcome	Actual Outcome
If black king appears on screen	Colour "black", position (0, 0) and coordinates (400, 300)	Must see if each piece appears on screen to before I can setup board	Black king image on screen at coordinate (400, 300)	Black king image on screen at coordinate (400, 300)

Making The Board Class - 27/12/2022

The *Board* class will provide most of the functionality for the game, and will be placed in board.py. *Board* contains attributes to assist with this. It will take in the arguments *board_colour* and *player_colour*.

Pygame and *Pieces* imported at the start of board.py.

```
import pygame
from pieces import *
```

```
class Board():

    def __init__(self, board_colour, player_colour):
        # Board as an empty 8x8 matrix - 0 represent empty square
        self.board = [[0 for i in range(0, 8)] for j in range(0, 8)]

        self.squares = []

        # Board setup if player is white
        self.player_white_start_pos = [

            ['r', 'n', 'b', 'q', 'k', 'b', 'n', 'r'],
            ['p', 'p', 'p', 'p', 'p', 'p', 'p', 'p'],
            ['.', '.', '.', '.', '.', '.', '.', '.'],
            ['.', '.', '.', '.', '.', '.', '.', '.'],
            ['.', '.', '.', '.', '.', '.', '.', '.'],
            ['.', '.', '.', '.', '.', '.', '.', '.'],
            ['P', 'P', 'P', 'P', 'P', 'P', 'P', 'P'],
            ['R', 'N', 'B', 'Q', 'K', 'B', 'N', 'R']
        ]
        # Board setup is player is black
        self.player_black_start_pos = [
            ['R', 'N', 'B', 'K', 'Q', 'B', 'N', 'R'],
            ['P', 'P', 'P', 'P', 'P', 'P', 'P', 'P'],
            ['.', '.', '.', '.', '.', '.', '.', '.'],
            ['.', '.', '.', '.', '.', '.', '.', '.'],
            ['.', '.', '.', '.', '.', '.', '.', '.'],
            ['.', '.', '.', '.', '.', '.', '.', '.'],
            ['p', 'p', 'p', 'p', 'p', 'p', 'p', 'p'],
            ['r', 'n', 'b', 'k', 'q', 'b', 'n', 'r']
        ]
```

- *board*: 2D list representing the current state of the chess board. The value at each position is either a *Piece* object or 0 if the square is empty.
- squares: a list of all the squares on the board
- *player_white_start_pos* and *player_black_start_pos*: 2D lists representing the initial positions of each player's pieces on the board. This is used to set up the board at the start of a new game.

Board also contains;

```
self.board_colour = board_colour

self.player_colour = player_colour

self.turn = 'white'

self.moves = []
```

- *board_colour*: A string representing the colour of the board, either "brown" or "gray"
- *player_colour*: A string representing the colour of the player, either "white" or "black"
- *turn*: A string representing the current turn, either "white" or "black"
- *moves:* A list of all legal moves in the current board state

Before we can draw a chess board to a screen, we must first set up a virtual board with our *Pieces*. The *setupBoard()* method in *Board* will do this based on the *player_colour*.

```
def setupBoard(self):

    square = None
    for i in range(0, 8):
        for j in range(0, 8):

            # Choose board template
            if self.player_colour == "white":
                square = self.player_white_start_pos[i][j]
            elif self.player_colour == "black":
                square = self.player_black_start_pos[i][j]
```

```
            # Corresponding ascii value for each square
            if square == 'R':
                self.board[i][j] = Rook("white", i, j)
            elif square == 'B':
                self.board[i][j] = Bishop("white", i, j)
            elif square == 'N':
                self.board[i][j] = Knight("white", i, j)
            elif square == 'Q':
                self.board[i][j] = Queen("white", i, j)
            elif square == 'K':
                self.board[i][j] = King("white", i, j)
            elif square == 'P':
                self.board[i][j] = Pawn("white", i, j)
            elif square == 'r':
                self.board[i][j] = Rook("black", i, j)
            elif square == 'b':
                self.board[i][j] = Bishop("black", i, j)
            elif square == 'n':
                self.board[i][j] = Knight("black", i, j)
            elif square == 'q':
                self.board[i][j] = Queen("black", i, j)
            elif square == 'k':
                self.board[i][j] = King("black", i, j)
            elif square == 'p':
                self.board[i][j] = Pawn("black", i, j)
```

This function sets up the initial setup of the chess board, based on the player colour. It initialises a 2D list of squares with pieces in their starting positions, by creating a new object for each piece class (*Rook*, *Bishop*, *Knight*, *Queen*, *King*, or *Pawn*) and setting its colour and position.

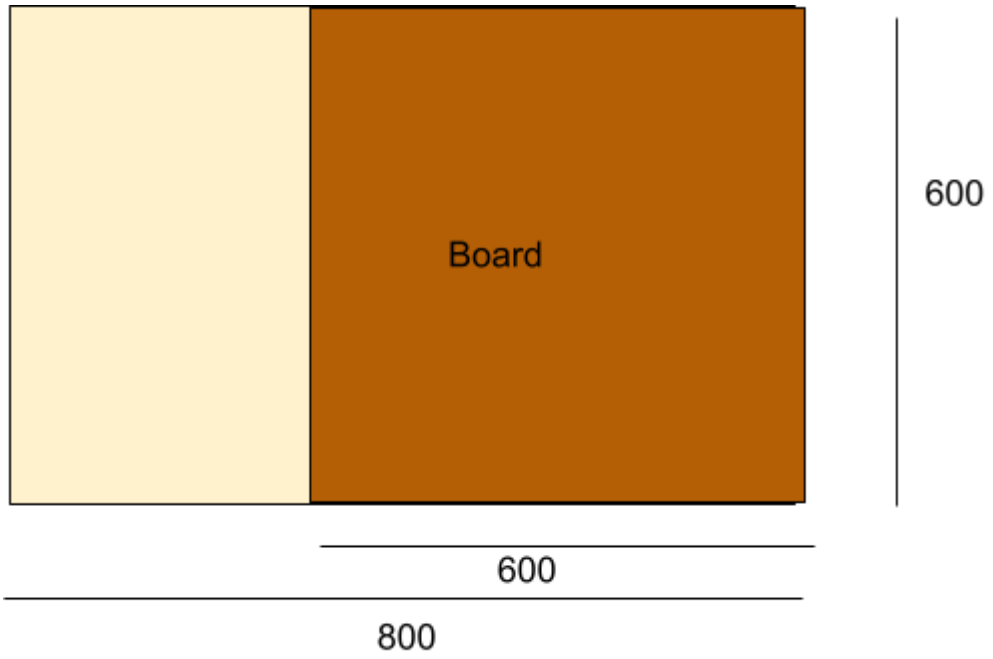
The function loops through the rows and columns of the board, gets the piece (represented by a character) corresponding to the current square from the player's starting configuration list, and creates a new object of the appropriate piece class with the corresponding colour and position. It then stores this object in the 2D list of squares at the corresponding position.

Prototyping Drawing Empty Board To Screen- 28/12/2022

I now have everything I need to draw a board to the screen.

The method to draw the board is called *drawBoard()* and is in the *Board* class.

The layout of the screen is as follows;



This means that each row of the square must be translated by 200 pixels to the right. The mathematical operator the will be used to draw the board are;

- For rows: (75*x) + 200
- For columns: (75*y)

The *drawBoard()* function will take in one parameter, *the_screen*.

```
def drawBoard(self, the_screen):

    # Temporay placeholder for each square
    square = None

    # Draw 64 squares to represent 8x8 board
    for i in range(0, 8):
        for j in range(0,8):
            # Square colour is dependent on the player's colour, board colour and coordinates
            square = Square(self.board_colour, (i, j))
            # Blit square onto screen with formula
            the_screen.blit(square.getSquare(), ((75*i)+200, (75*j)))
            # Add square to square list
            self.squares.append(square)
```

The function blits the 8x8 board onto the screen. It then appends each square to the self.squares list.

The formula used to calculate the position of each square is (75*i)+200 for the x coordinate and (75*j) for the y coordinate. This will draw the squares in a 8x8 grid with a margin of 200 pixels from the left edge of the screen. This makes sure that the screen has the previous layout.

The *Square* class that is used to create each square has the *getSquare()* method that returns a square with the correct dimensions and colour, as shown previously.

Testing If Board Appears On The Screen (7)

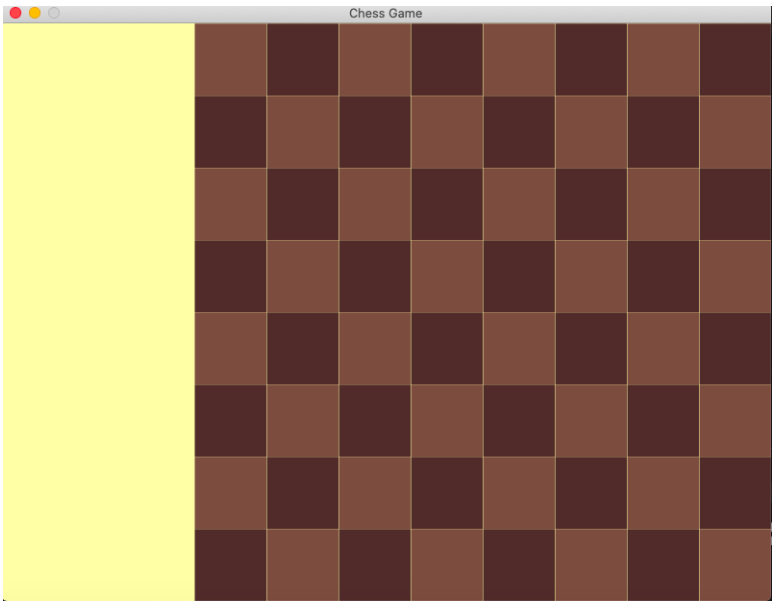
To test this, the board class is imported into main.py, a new *Board* is created with a colour of "brown" and player colour of "white". We can then call the *drawBoard()* function in the game loop, passing *screen* as the parameter

```
from board import Board
```

```
b = Board("brown", "white")
```

```
while running:

    screen.fill((255, 255, 165))
    b.drawBoard(screen)
```

Output;

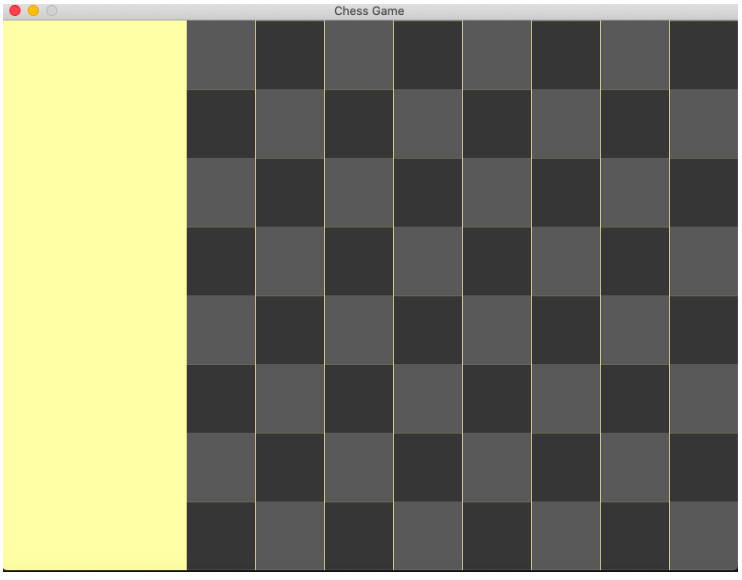


What's being tested	Inputs	Reasoning	Expected Outcome	Actual Outcome
If *drawBoard()* places a brown chess board on the screen	*board_colour* of "brown" and *player_colour* of "white"	User needs to see a visible board of both colour schemes to play the game	Brown chess board on right side of the screen	Brown chess board on right side of the screen

By changing *board_colour* to "gray", a grey board should appear on the screen.

```
b = Board("gray", "white")
```

Output;



What's being tested	Inputs	Reasoning	Expected Outcome	Actual Outcome
If *drawBoard()* places a grey chess board on the screen	*board_colour* of "gray" and *player_colour* of "white"	User needs to see a visible board of both colour schemes to play the game	Grey chess board on right side of the screen	Grey chess board on right side of the screen

Client Feedback

I showed both coloured boards to [redacted] and [redacted], who were happy with the setup of both screens. They asked "can we change the board colour while on the screen?" I explained to them that I plan to create the chess game first then create a simple UI (user interface) which will involve toggling the board colour. They understood and said they would wait until the chess game was done and ask again.

Computer Overheating - 28/12/2022

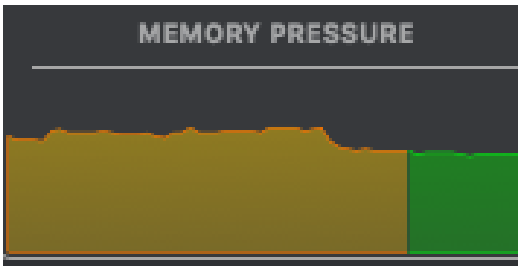
After leaving my computer open for around 15 minutes and forgetting to close the Pygame window, I noticed that my computer was beginning to overheat and its fan was spinning.

I checked the activity monitor of my computer and found that its memory pressure has increased by a lot.

I checked which programs were causing this problem and found that it was with python.

- 1.66GB is way to high for python

I realised that the Pygame window was opened and quickly closed it. The memory pressure of my computer went back to normal after that.



I was confused on why python was putting that much strain on the CPU, so I went onto stackoverflow.com and found a thread started by someone who had a similar problem.

Does running pygame usually make computers warm

- From stackoverflow.com

I found this response and suggestion;

limit the frames per second to limit CPU usage with `pygame.time.Clock.tick` The method `tick()` of a `pygame.time.Clock` object, delays the game in that way, that every iteration of the loop consumes the same period of time. See `pygame.time.Clock.tick()`:

> This method should be called once per frame.

- From stackoverflow.com

I decided to add my own game clock to fix this, storing it in a variable called *clock*.

```
clock = pygame.time.Clock()
```

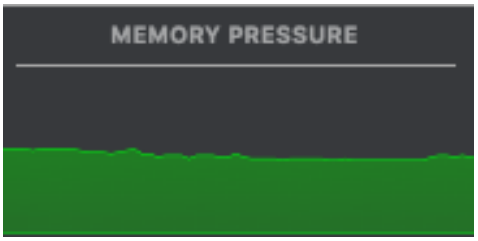
```
pygame.display.update()
clock.tick(24)
```

I decided that 24 frames per second was good for my game as it won't contain any intense graphics.

Testing Clock (8)

To test the clock, I ran my game and left it running for 45 minutes as I took a break to have a snack.

When I came back, my computer was not overheating, the fan was not on, the memory pressure was low.



What's being tested	Inputs	Reasoning	Expected Outcome	Actual Outcome
If clock keeps computer from overheating	Pygame window running for 45 minutes	Make sure game does not damage user's hardware	Low memory pressure and no indication of computer overheating	Low memory pressure and no indication of computer overheating

Prototyping Drawing Pieces Onto Board - 30/12/2022

By adding a second loop nested for-loop in *drawBoard()* to iterate through the *board*, we can draw each to *Piece* object to the board *b* with the *getImage()* function and the same calculating previously used with the squares. For every piece added, it is placed on its corresponding square.

```
# Draw pieces to the screen by looping through the board
for i in range(0, 8):
    for j in range(0, 8):
        # Check if a piece is on that square
        if self.board[i][j] != 0:
            # Blit piece onto screen with formula
            the_screen.blit(self.board[i][j].getImage(), ((75*i)+200, (75*j)))
            # Add piece onto corresponding square
            for s in self.squares:
                if s.getPosition() == (i,j):
                    s.setPieceOnSquare(self.board[i][j])
```

Testing Drawing Pieces Onto Board (9)

Calling the *setupBoard()* function, each piece will be placed on its corresponding square based on whether the player is "white" or "black". The pieces of the player's colour should be at the bottom of the screen.

```
b.setupBoard()
```

```
b = Board("brown", "white")
```

Output;

What's being tested	Inputs	Reasoning	Expected Outcome	Actual Outcome
Drawing board to screen with images of pieces	Player's colour; "white"	Ensure board can be setup to begin a game with player's colour	Board on screen with white pieces setup at the bottom and black pieces on top	Board on screen with white pieces on the right and black pieces on the left

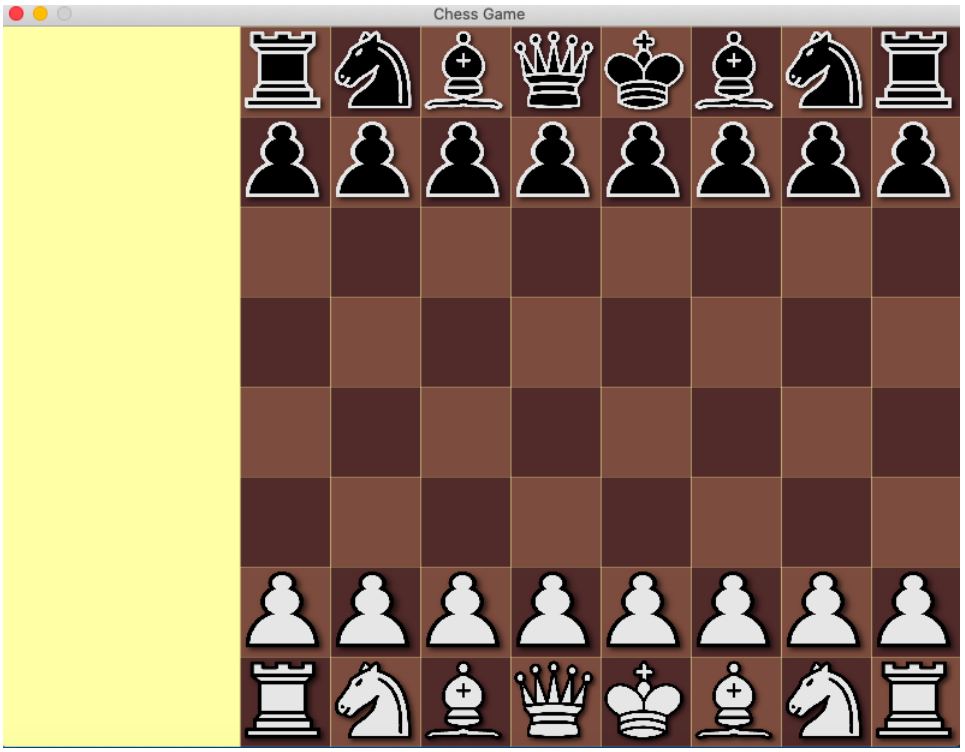
This test was a failure.

The board looks to be reflected in the line y = x

Looking at my code, I realised that *i* and *j* in my for loops represent x and y respectively. If I switch around *i* and *j,* this should be the inverse of the reflection so it should place the pieces correctly.

```
# Draw pieces to the screen by looping through the board
for i in range(0, 8):
    for j in range(0, 8):
        # Check if a piece is on that square
        if self.board[i][j] != 0:
            # Blit piece onto screen with formula
            the_screen.blit(self.board[i][j].getImage(), ((75*j)+200, (75*i)))
            # Add piece onto corresponding square
            for s in self.squares:
                if s.getPosition() == (j,i):
                    s.setPieceOnSquare(self.board[i][j])
```

Output;



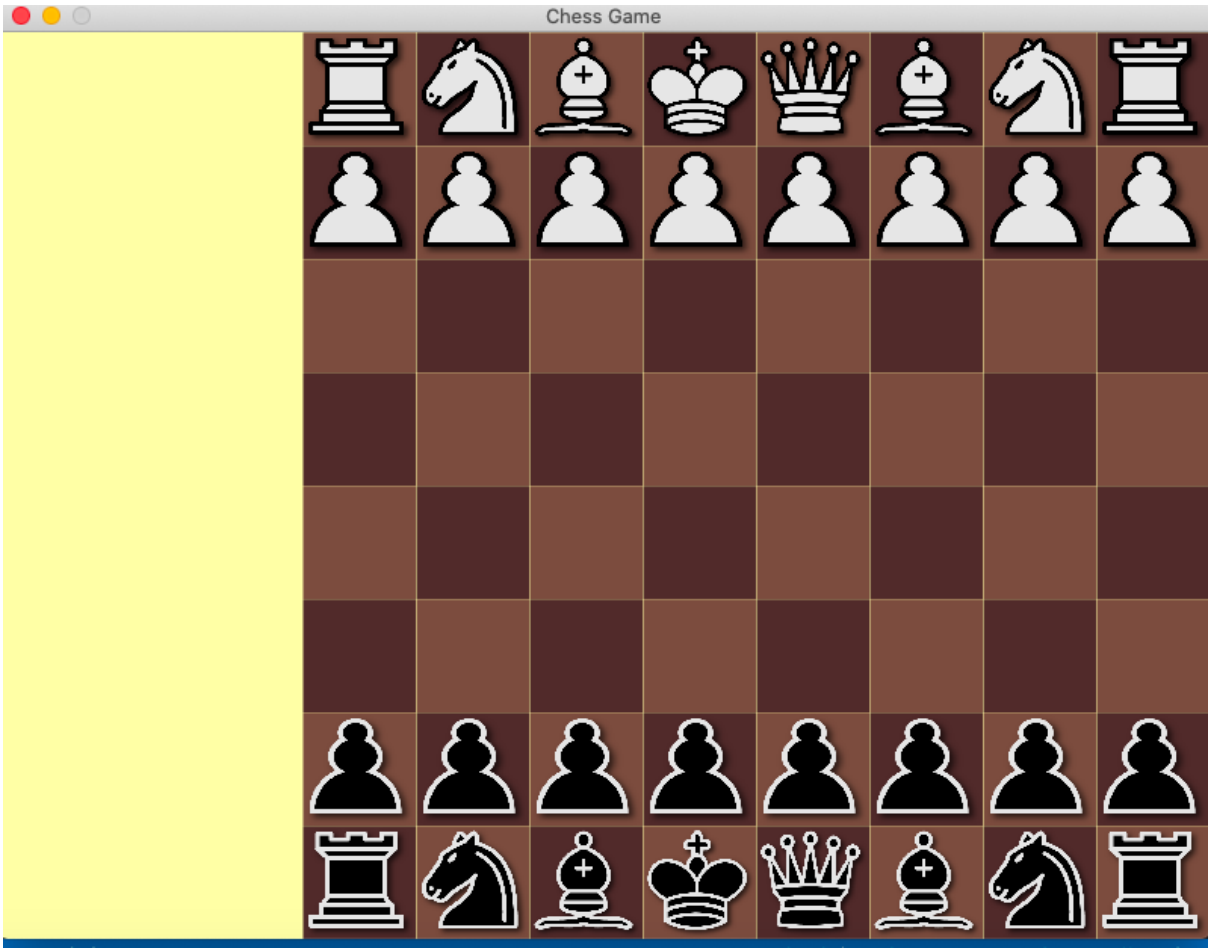
What's being tested	Inputs	Reasoning	Expected Outcome	Actual Outcome
If white pieces are at bottom of screen	Player's colour; "white", *i* as y and *j* as x	Player's colour pieces should be at the bottom of the board	White pieces at bottom of board and black pieces at top of board	White pieces at bottom of board and black pieces at top of board

This test was a success.

Changing *player_colour* to "black" should cause the black pieces to appear at the bottom and the white pieces to appear at the top.

```
b = Board("brown", "black")
```

Output;



What's being tested	Inputs	Reasoning	Expected Outcome	Actual Outcome
Layout of board if player is black	Player's colour; "black"	Player's colour pieces should be at the bottom of the board	Black pieces at bottom of board and white pieces at top of board	Black pieces at bottom of board and white pieces at top of board

Client Feedback 1

[redacted] and [redacted] said that the pieces were large enough relative to the square size. They liked the layout of the board for white, but [redacted] pointed out that in chess, when playing as black, the dark square must be in the bottom right hand corner. My black board for the player has a light square in the corner.

I figured that I had to edit the *Square* class to take into account *player_colour* when deciding the square's colour. This means that the square object must take in the additional argument of player colour. [redacted] seemed very upset when I presented this.

```
class Square():

    def __init__(self, board_colour, square_pos, player_colour):
```

I expanded the condition which decides the square colour to make sure that if the player colour is black, the board will alternate its colours in the flipped direction.

```
if ((self.x % 2 == self.y % 2) and (player_colour == 'white'))  or ((self.x % 2 != self.y % 2) and (player_colour == 'black')):
    # Is light if sum of both coordinates are even and colour is white, or if sum is odd and colour is black
    if board_colour == "brown":
        self.square = square_brown_light
    elif board_colour == "gray":
        self.square = square_gray_light

else:
    # Otherwise is dark
    if board_colour == "brown":
        self.square = square_brown_dark
    elif board_colour == "gray":
        self.square = square_gray_dark
```

Testing If Square Is In Correct Corner (10)

First, I'll test if the new algorithm changes the bottom right hand corner when playing as white remains dark.

I got this error;

```
    square = Square(self.board_colour, (j, i))
TypeError: __init__() missing 1 required positional argument: 'player_colour'
```

What's being tested	Inputs	Reasoning	Expected Outcome	Actual Outcome
If bottom right hand square is light	Player's colour; "white"	Player's colour should be the colour of the bottom right hand corner	Light square at bottom right hand corner	TypeError; missing argument

This test was a failure.

Looking at the error, I realised that I forgot to pass my actual argument into each of my *Square* objects. This is an easy fix, all I must do is pass the *Board*'s *player_colour* as the parameter.

```
# Square colour is dependent on the player's colour, board colour and coordinates
square = Square(self.board_colour, (j, i), self.player_colour)
```

I tried running my code again and no errors occurred.

What's being tested	Inputs	Reasoning	Expected Outcome	Actual Outcome
If bottom right hand square is light	Player's colour; "white"	Player's colour should be the colour of the bottom right hand corner	Light square at bottom right hand corner	Light square at bottom right hand corner

This test was a success.

I tried the same for the black squares.

```
b = Board("brown", "black")
```

Output;

What's being tested	Inputs	Reasoning	Expected Outcome	Actual Outcome
If bottom right hand square is dark	Player's colour; "black"	Player's colour should be the colour of the bottom right hand corner	Dark square at bottom right hand corner	Dark square at bottom right hand corner

This test was a success.

Client Feedback 2

[redacted] was happy with the board layout now and explained that this was important when chess games were notated. I now know why they made a big deal when I presented the previous layout.

Making The Move Class - 03/01/2023

A chess move can be simplified to two basic properties; containing a start position and containing an end position.

In move.py, I made the *Move* class with attributes *oPos* and *nPos* (indicating the old and new position respectively) and has getter methods for both attributes.

```
class Move():

    def __init__(self, oPos, nPos):

        self.oPos = oPos
        self.nPos = nPos

    def getOPos(self):
        # Get start position
        return self.oPos

    def getNPos(self):
        # Get end position
        return self.nPos
```

Prototyping Movement For King - 03/01/2023

Before I started coding the legal moves for each piece, I added the *getBoard()* function to *Board* so that I can access the *board* attribute outside of the class.

```
def getBoard(self):
    return self.board
```

I decided that coding the movements for the knight and king will be easiest to start with, because with these pieces I am only concerned with their end position, not every position in between. These pieces sort of "teleport" into their final positions.

I started with the king.

The first step is to add a list of *possible_moves* in the *getValidMoves()* function in the *King* class. The king can move one square in any direction (including one step in diagonals). That means that there are 8 possible moves.

```
def getValidMoves(self, board):

    possible_moves= [
        (self.positionX+1, self.positionY),
        (self.positionX+1, self.positionY+1),
        (self.positionX+1, self.positionY-1),
        (self.positionX, self.positionY+1),
        (self.positionX-1, self.positionY),
        (self.positionX-1, self.positionY+1),
        (self.positionX-1, self.positionY-1),
        (self.positionX, self.positionY-1)
    ]
```

Next I will loop over every possible move, and check if they are within the bounds of the board (between indexes 0 and 8).

Then the function will get the board layout from the given *board* argument and check if the possible move has a piece on it. If it does have a piece on it, it checks if the piece is of an opposite colour then allows the move (indicating a capture of an enemy piece). If no piece is on the square it also allows the move.

Each allowed move is stored in the *validMoves* list of that piece and added using the *addMove()* function. The move added has an *oPos* of the current position and *nPos* of the new position. The function returns the list of *validMoves*.

```
for move in possible_moves:

    # If move on board
    if (move[0] < 8) and (move[1] < 8) and (move[0] >= 0) and (move[1] >= 0):
        # If move is on piece of opposite colour allow capture
        if board.getBoard()[move[1]][move[0]] != 0:
            if board.getBoard()[move[1]][move[0]].getColour() != self.colour:
                m = Move(self.getPos(), (move))
                self.addMove(m)
        # Move to empty square
        else:
            m = Move(self.getPos(), (move))
            self.addMove(m)

return self.validMoves
```

Testing King Legal Moves (11)

To test the legal moves of the king, I placed in position (3, 4) on my starting template board.

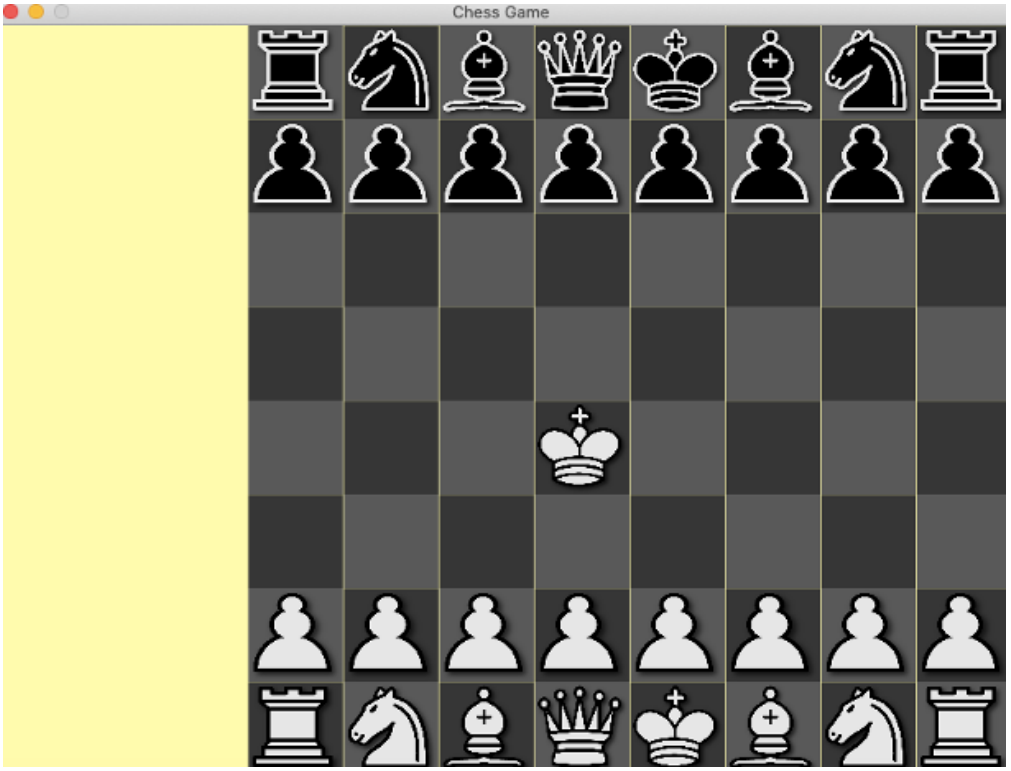
```
# Board setup if player is white
self.player_white_start_pos = [

    ['r', 'n', 'b', 'q', 'k', 'b', 'n', 'r'],
    ['p', 'p', 'p', 'p', 'p', 'p', 'p', 'p'],
    ['.', '.', '.', '.', '.', '.', '.', '.'],
    ['.', '.', '.', '.', '.', '.', '.', '.'],
    ['.', '.', '.', 'K', '.', '.', '.', '.'],
    ['.', '.', '.', '.', '.', '.', '.', '.'],
    ['P', 'P', 'P', 'P', 'P', 'P', 'P', 'P'],
    ['R', 'N', 'B', 'Q', 'K', 'B', 'N', 'R']
]
```

I plan to edit the template board whenever I need to test different board positions for different pieces and scenarios of the game.

I wrote this script in main.py to output the possible positions of the king at coordinate (3, 4) to ensure that the legal moves are being calculated correctly.

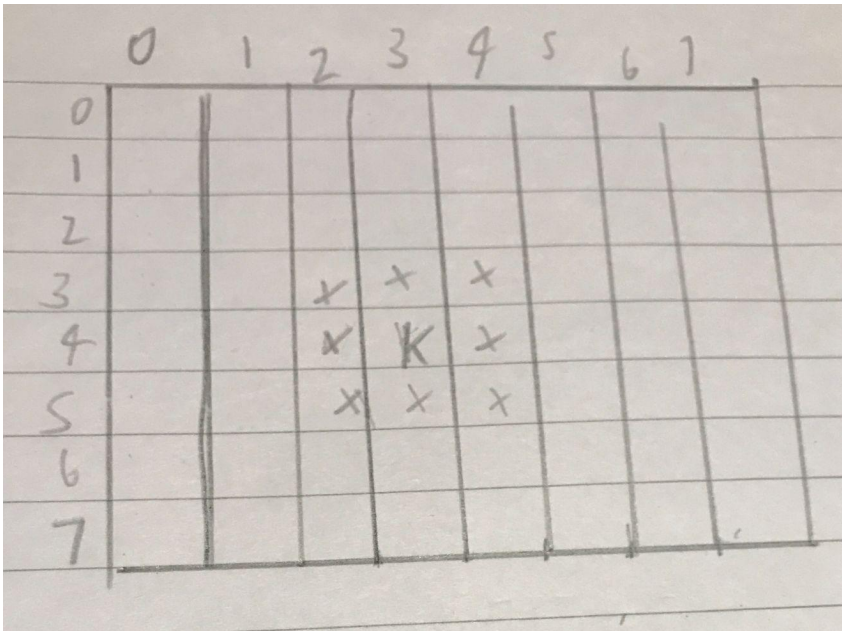
```
for x in b.getBoard()[4][3].getValidMoves(b):
    print(x.getNPos())
```

Output;




```
Hello from the pygame community. https://www.pygame.org/contribute.html
(4, 4)
(4, 5)
(4, 3)
(3, 5)
(2, 4)
(2, 5)
(2, 3)
(3, 3)
```

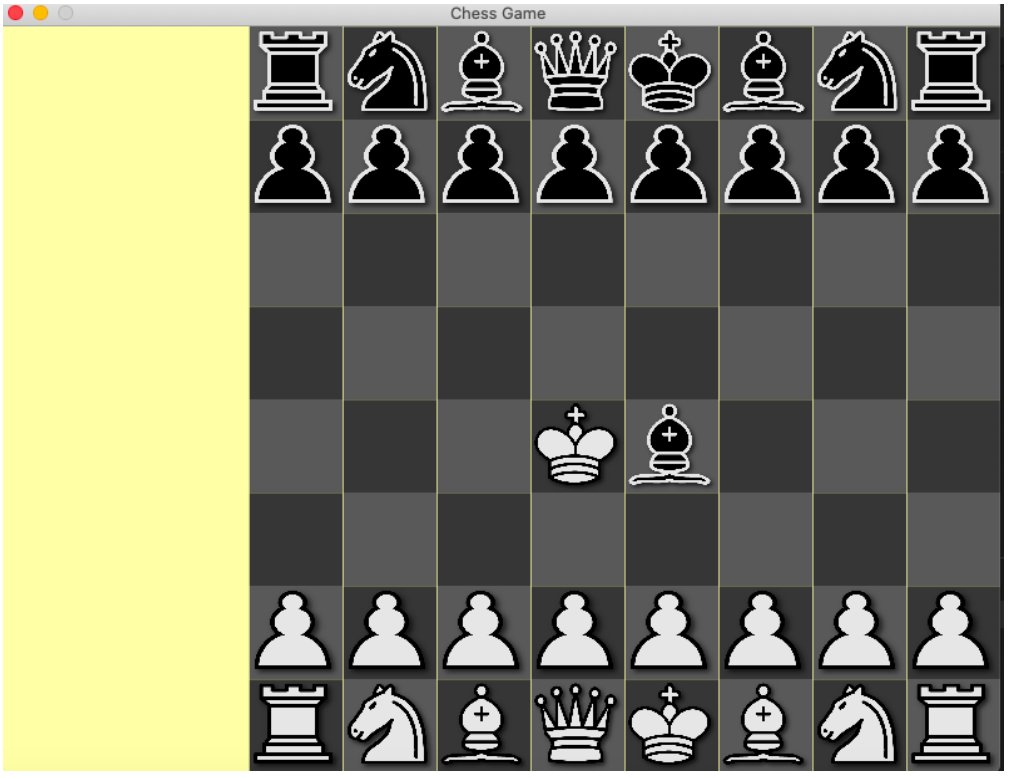
I decided to mark the coordinates on paper as it is easier to check if they're correct.



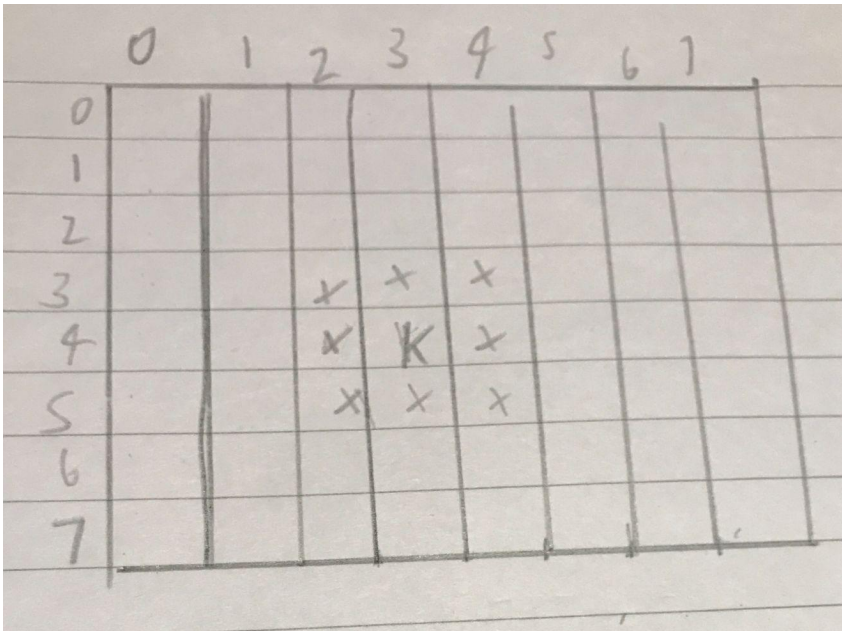
What's being tested	Inputs	Reasoning	Expected Outcome	Actual Outcome
King's legal moves	None	Make sure every piece obeys the rules of chess	Valid list of position which the king can move	Valid list of position which the king can move

Test was a success.

Now test to see if it can capture an opposite coloured piece.

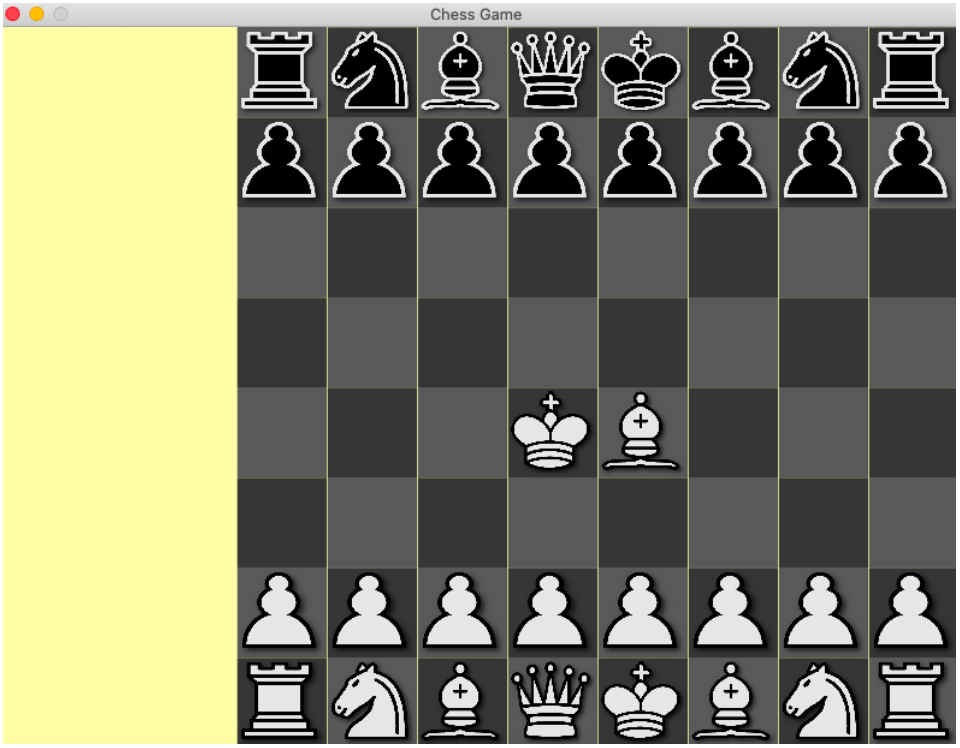



```
Hello from the pygame community. https://www.pygame.org/contribute.html
(4, 4)
(4, 5)
(4, 3)
(3, 5)
(2, 4)
(2, 5)
(2, 3)
(3, 3)
```

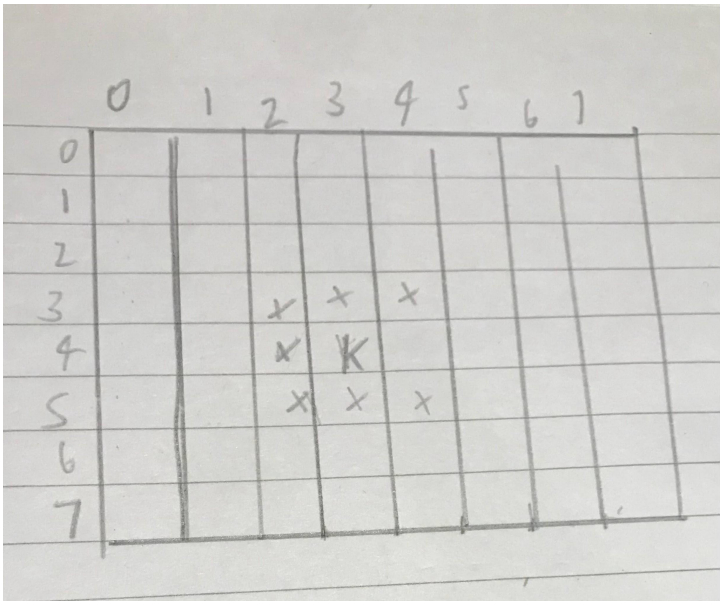


What's being tested	Inputs	Reasoning	Expected Outcome	Actual Outcome
If king can capture bishop	None	Make sure every piece obeys the rules of chess	One of the positions in list is (4, 4) which is same position as bishop	One of the positions in list is (4, 4) which is same position as bishop

Finally, let's ensure it cannot capture its own piece.




```
(4, 5)
(4, 3)
(3, 5)
(2, 4)
(2, 5)
(2, 3)
(3, 3)
```



What's being tested	Inputs	Reasoning	Expected Outcome	Actual Outcome
If king won't capture its own piece	None	Make sure every piece obeys the rules of chess	One of the positions in list is **not** (4, 4) which is same position as bishop	One of the positions in list is **not** (4, 4) which is same position as bishop

Legal moves for the king are correct.

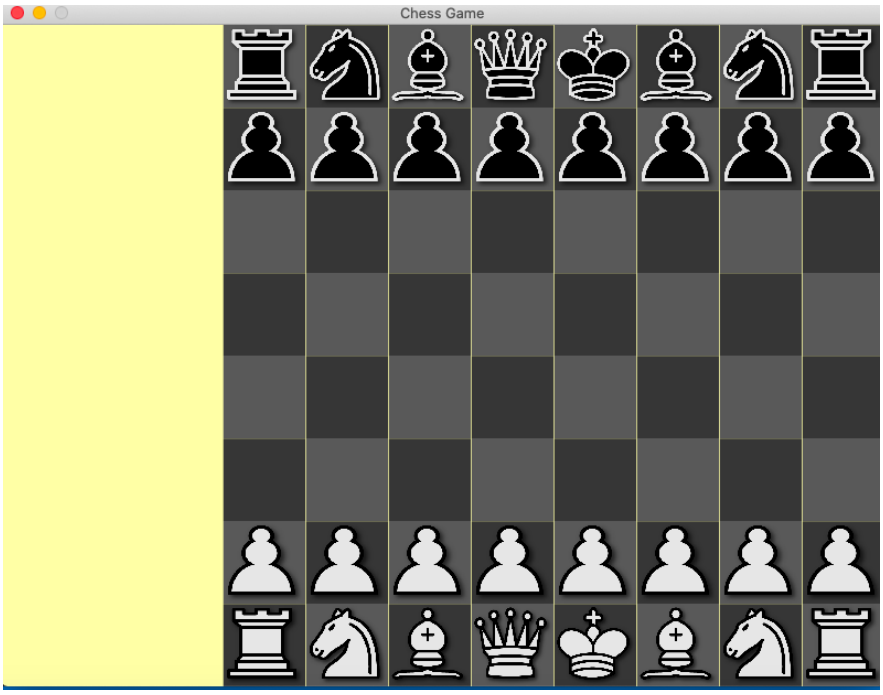
Prototyping Movement For Knight - 03/01/2023

The code for getting valid moves for the knight is similar to getting thos for the king. The only difference is that the knight moves in an 'L' shape so the list of *possible_moves* will be different. There are 8 possible moves for the knight.

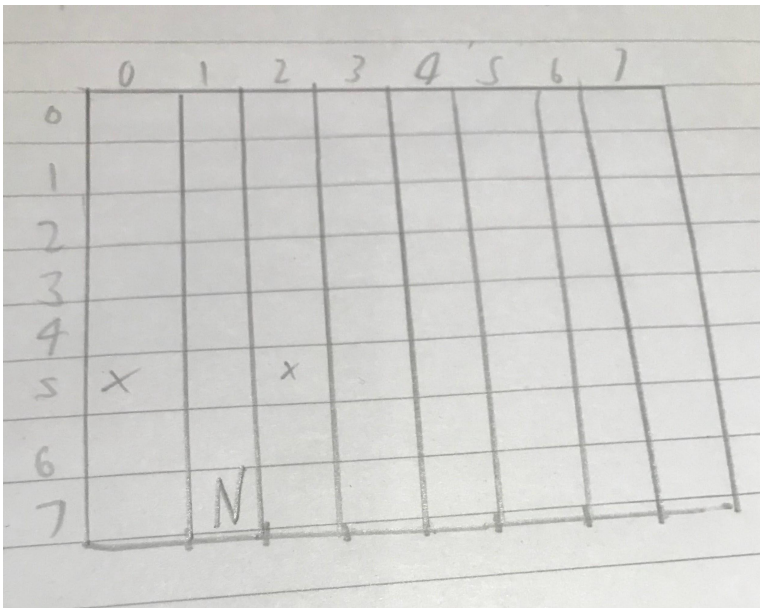
```
def getValidMoves(self, board):

    self.validMoves = []

    move = None

    # Knight Moves
    possible_moves = [
        (self.positionX+2, self.positionY+1),
        (self.positionX+2, self.positionY-1),
        (self.positionX-2, self.positionY+1),
        (self.positionX-2, self.positionY-1),
        (self.positionX+1, self.positionY+2),
        (self.positionX+1, self.positionY-2),
        (self.positionX-1, self.positionY+2),
        (self.positionX-1, self.positionY-2)
    ]

    for move in possible_moves:

        # If move on board
        if (move[0] < 8) and (move[1] < 8) and (move[0] >= 0) and (move[1] >= 0):
            # If move is on piece of opposite colour allow capture
            if board.getBoard()[move[1]][move[0]] != 0:
                if board.getBoard()[move[1]][move[0]].getColour() != self.colour:
                    m = Move(self.getPos(), (move))
                    self.addMove(m)
            # Move to empty square
            else:
                m = Move(self.getPos(), (move))
                self.addMove(m)

    return self.validMoves
```

Testing Knight Legal Moves (12)

I used the same script to test the legal moves as I did for the king, but instead changed the coordinates to (1, 7), which is the original starting position of a knight.



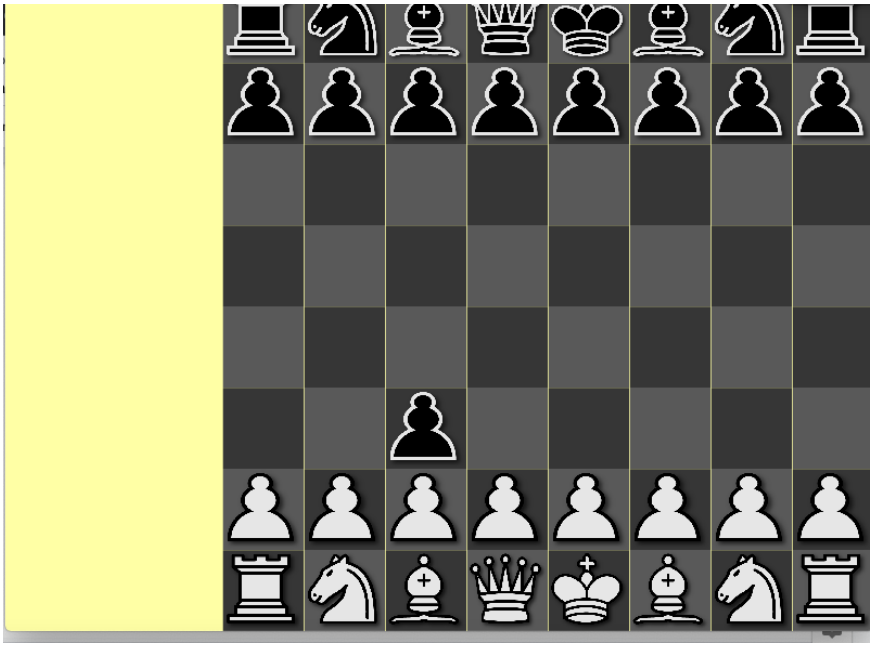

```
Hello from the pygame community. https://www.pygame.org/contribute.html
(2, 5)
(0, 5)
```



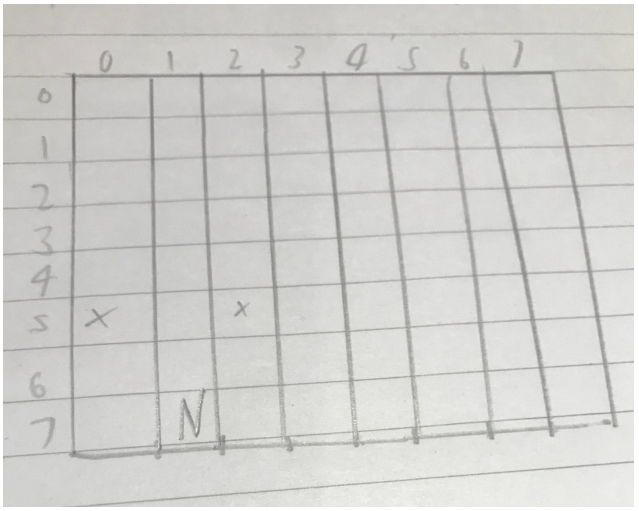
This test if the knight can go to an empty square, and if the knight can't capture its own coloured pieces.

What's being tested	Inputs	Reasoning	Expected Outcome	Actual Outcome
Knight's legal moves	None	Make sure every piece obeys the rules of chess	Valid list of position which the knight can move	Valid list of position which the knight can move

When I put an enemy pawn in scope of the knight;



```
Hello from the pygame community. https://www.pygame.org/contribute.html
(2, 5)
(0, 5)
```



What's being tested	Inputs	Reasoning	Expected Outcome	Actual Outcome
If knight can capture pawn	None	Make sure every piece obeys the rules of chess	One of the positions in list is (2, 5) which is same position as pawn	One of the positions in list is (2, 5) which is same position as pawn

Legal moves for the knight are correct.

Prototyping Movement For Rook - 05/01/2023

The difference between the movement of the rook from the king and knight is that the king and knight have a set list of 8 moves while the rook's moves are dependent on its position on the board.

Because I am going to have to reuse the rooks movement code when making the queen's movement code (as the queen moves like both a rook and bishop), I am creating a method in *Piece* called *checkStraightsl()* which calculates possible horizontal and vertical moves for a piece. This means that both the queen and rook can use this function.

It uses different iterations to get moves which are North, South, East and West. Iterations for West and North increment backwards and iterations for South and East are forward. East and West use the x-coordinate of the piece while North and South use the y-coordinate of the piece. The iteration validates that we aren't checking squares outside of the *board*.

```
def checkStaights(self, board):

    move = None

    # Get South
    for i in range(self.positionY+1, 8):
        if board.getBoard()[i][self.positionX] == 0:
            move = Move(self.getPos(), (self.positionX, i))
            self.addMove(move)
        elif board.getBoard()[i][self.positionX].getColour() != self.colour:
            move = Move(self.getPos(), (self.positionX, i))
            self.addMove(move)
            break
        else:
            break
```

```
# Get East
for i in range(self.positionX+1, 8):
    if board.getBoard()[self.positionY][i] == 0:
        move = Move(self.getPos(), (i, self.positionY))
        self.addMove(move)
    elif board.getBoard()[self.positionY][i].getColour() != self.colour:
        move = Move(self.getPos(), (i, self.positionY))
        self.addMove(move)
        break
    else:
        break
```

```
# Get West
for i in range(self.positionX-1, -1, -1):
    if board.getBoard()[self.positionY][i] == 0:
        move = Move(self.getPos(), (i, self.positionY))
        self.addMove(move)
    elif board.getBoard()[self.positionY][i].getColour()!= self.colour:
        move = Move(self.getPos(), (i, self.positionY))
        self.addMove(move)
        break
    else:
        break
```

```
# Get North
for i in range(self.positionY-1, -1, -1):
    if board.getBoard()[i][self.positionX] == 0:
        move = Move(self.getPos(), (self.positionX, i))
        self.addMove(move)
    elif board.getBoard()[i][self.positionX].getColour()!= self.colour:
        move = Move(self.getPos(), (self.positionX, i))
        self.addMove(move)
        break
    else:
        break
```

If a line is clear, the method creates a move object representing a legal move for the rook to that new square, and adds it to the rook's list of legal moves.

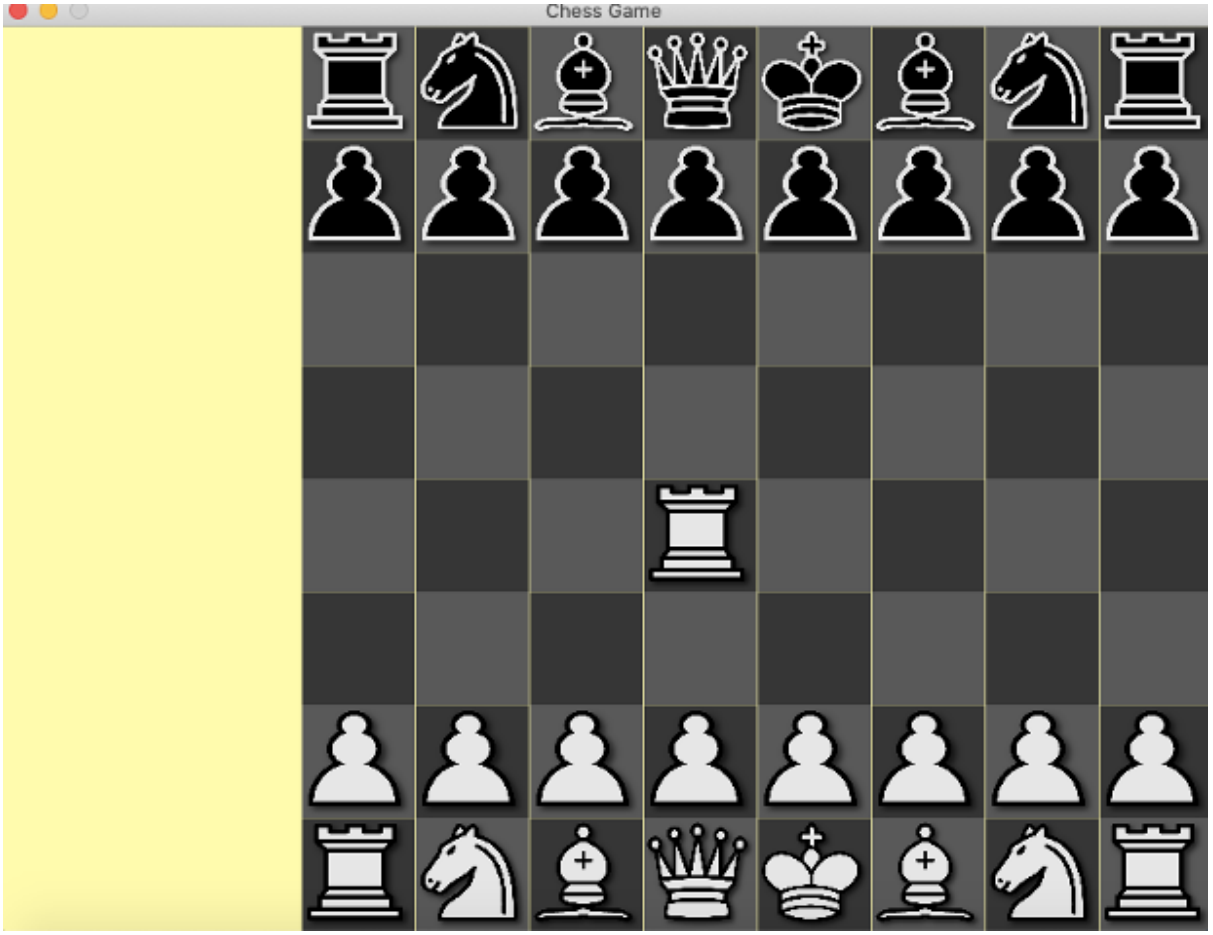
If a line is obstructed by an opposing piece, the method creates a move object for the rook to capture that piece, and then stops checking that direction for any further legal moves in that line.

The function is called upon in the *getValidMoves()* function for the *Rook*.

```
def getValidMoves(self, board):

    self.validMoves = []

    self.checkStaights(board)
    return self.validMoves
```

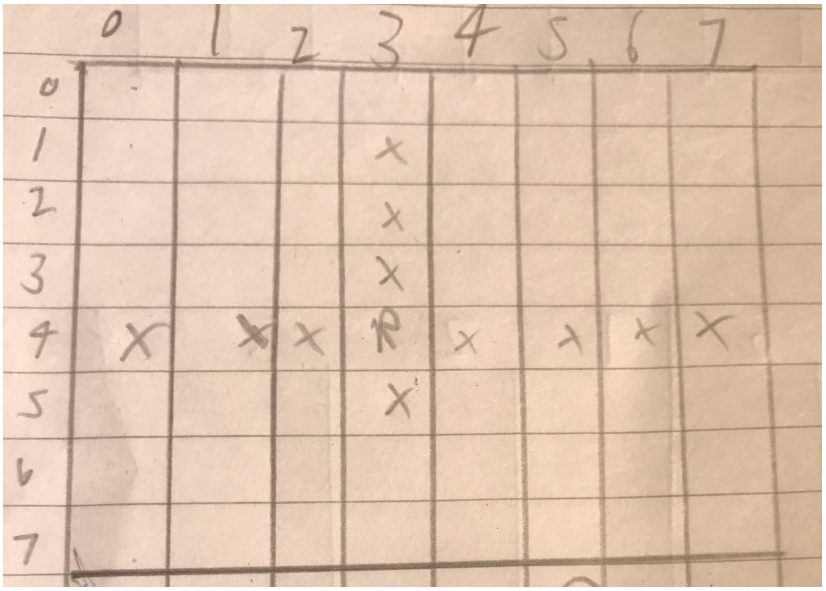
Testing Rook Legal Moves (13)

I placed a rook at coordinates (3, 4), used the same script that I used for knight and king testing, and marked the legal moves on paper.



This position tests moving to an empty square, capturing enemy pieces and not capturing friendly pieces.

```
Hello from the pygame community. https://www.pygame.org/contribute.html
(3, 5)
(3, 6)
(4, 4)
(5, 4)
(6, 4)
(7, 4)
(2, 4)
(1, 4)
(0, 4)
(3, 3)
(3, 2)
(3, 1)
```



What's being tested	Inputs	Reasoning	Expected Outcome	Actual Outcome
Rook's legal moves	None	Make sure every piece obeys the rules of chess	Valid list of position which the rook can move	Valid list of position which the rook can move

Legal moves for the rook are correct.

Prototyping Movement For Bishop - 05/01/2023

The bishop's diagonal movement can be represented by the following lines;

- y = x
- y = -x

Where the bishop's current position represents the origin and it can travel bidirectionally on both lines.

Like we did for the rook, I created the *checkDiagonals()* function in *Piece* as the diagonal moves are also needed for the queen.

Directions South-East and North-West are represented by the line y = x, while South-West and North-East are represented by the line y = -x.

```
def checkDiagonals(self, board):

    move = None

    # Get South-East
    for i in range(1,8):
        if self.positionX+i < 8 and self.positionY+i < 8:
            if board.getBoard()[self.positionY+i][self.positionX+i] == 0:
                move = Move(self.getPos(), (self.positionX+i, self.positionY+i))
                self.addMove(move)
            elif board.getBoard()[self.positionY+i][self.positionX+i].getColour() != self.colour:
                move = Move(self.getPos(), (self.positionX+i, self.positionY+i))
                self.addMove(move)
                break
            else:
                break
        else:
            break
```

```
# Get South-West
for i in range(1,8):
    if self.positionX-i >= 0 and self.positionY+i < 8:
        if board.getBoard()[self.positionY+i][self.positionX-i] == 0:
            move = Move(self.getPos(), (self.positionX-i, self.positionY+i))
            self.addMove(move)
        elif board.getBoard()[self.positionY+i][self.positionX-i].getColour()!= self.colour:
            move = Move(self.getPos(), (self.positionX-i, self.positionY+i))
            self.addMove(move)
            break
        else:
            break
    else:
        break
```

```
# Get North-East
for i in range(1,8):
    if self.positionX+i < 8 and self.positionY-i >= 0:
        if board.getBoard()[self.positionY-i][self.positionX+i] == 0:
            move = Move(self.getPos(), (self.positionX+i, self.positionY-i))
            self.addMove(move)
        elif board.getBoard()[self.positionY-i][self.positionX+i].getColour()!= self.colour:
            move = Move(self.getPos(), (self.positionX+i, self.positionY-i))
            self.addMove(move)
            break
        else:
            break
    else:
        break
```

```
# Get North-West
for i in range(1,8):
    if self.positionX-i >= 0 and self.positionY-i >= 0:
        if board.getBoard()[self.positionY-i][self.positionX-i] == 0:
            move = Move(self.getPos(), (self.positionX-i, self.positionY-i))
            self.addMove(move)
        elif board.getBoard()[self.positionY-i][self.positionX-i].getColour()!= self.colour:
            move = Move(self.getPos(), (self.positionX-i, self.positionY-i))
            self.addMove(move)
            break
        else:
            break
    else:
        break
```

If a diagonal is clear, the method creates a move object representing a legal move for bishop rook to that new square, and adds it to the bishop's list of legal moves.

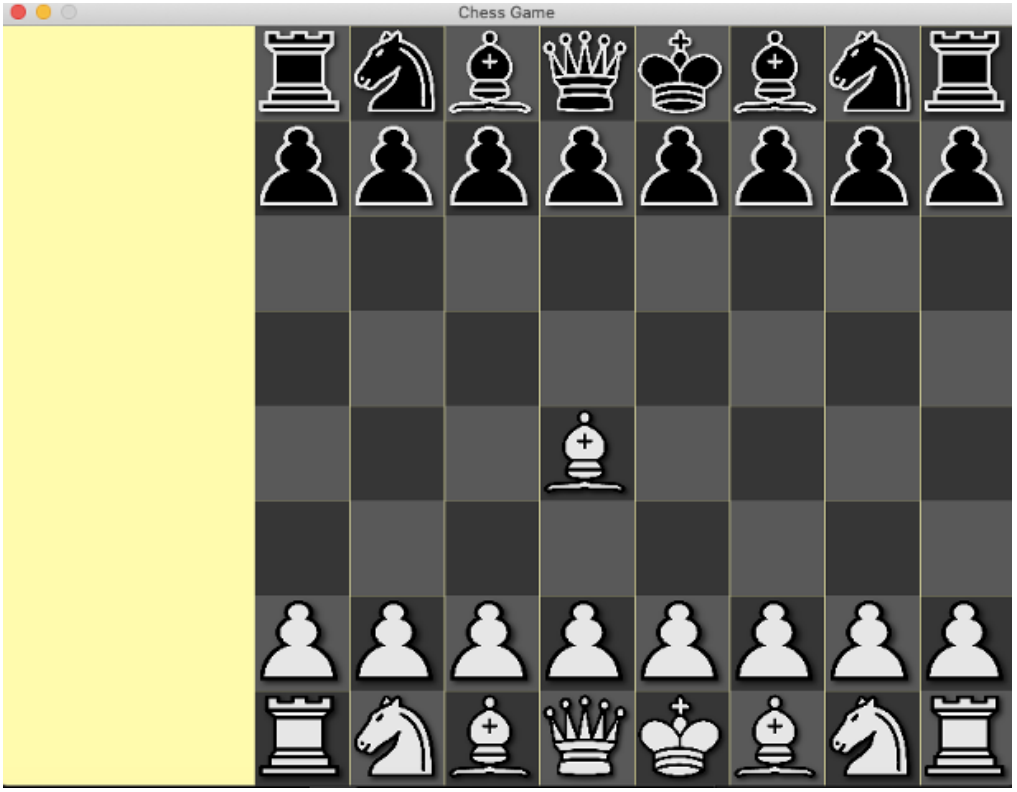
If a diagonal is obstructed by an opposing piece, the method creates a move object for the bishop to capture that piece, and then stops checking that direction for any further legal moves in that line.

The function is called upon in the *getValidMoves()* function for the *Bishop*.

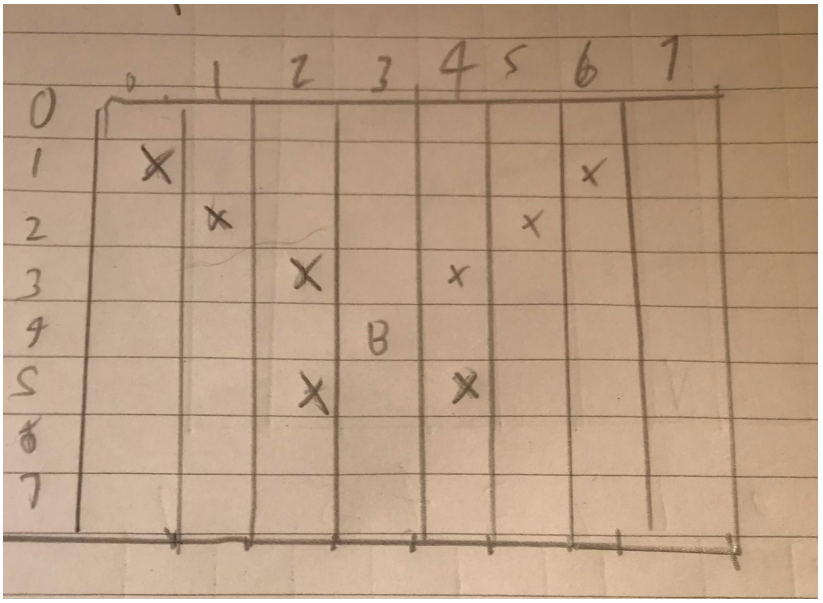
```
def getValidMoves(self, board):

    self.validMoves = []

    self.checkDiagonals(board)
    return self.validMoves
```

Testing Bishop Legal Moves (14)

Placing the bishop on (3, 4) tests moving to an empty square, capturing enemy pieces and not capturing friendly pieces.




```
(4, 5)
(2, 5)
(4, 3)
(5, 2)
(6, 1)
(2, 3)
(1, 2)
(0, 1)
```



What's being tested	Inputs	Reasoning	Expected Outcome	Actual Outcome
Bishop legal moves	None	Make sure every piece obeys the rules of chess	Valid list of position which the bishop can move	Valid list of position which the bishop can move

Legal moves for the bishop are correct.

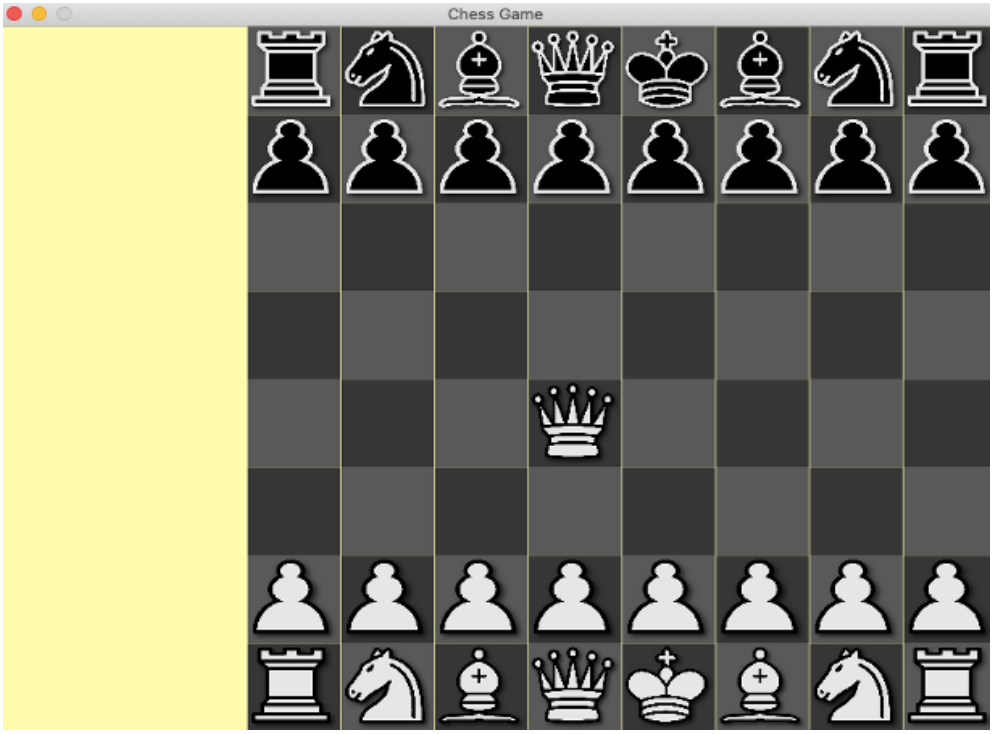
Prototyping Movement For Queen - 05/01/2023

Because we already have the code to get the straights and diagonals, so calling the *checkStriaghts()* and *checkDiagonals()* in the *getValidMoves()* function of the *Queen*, will get all the legal moves for the queen.

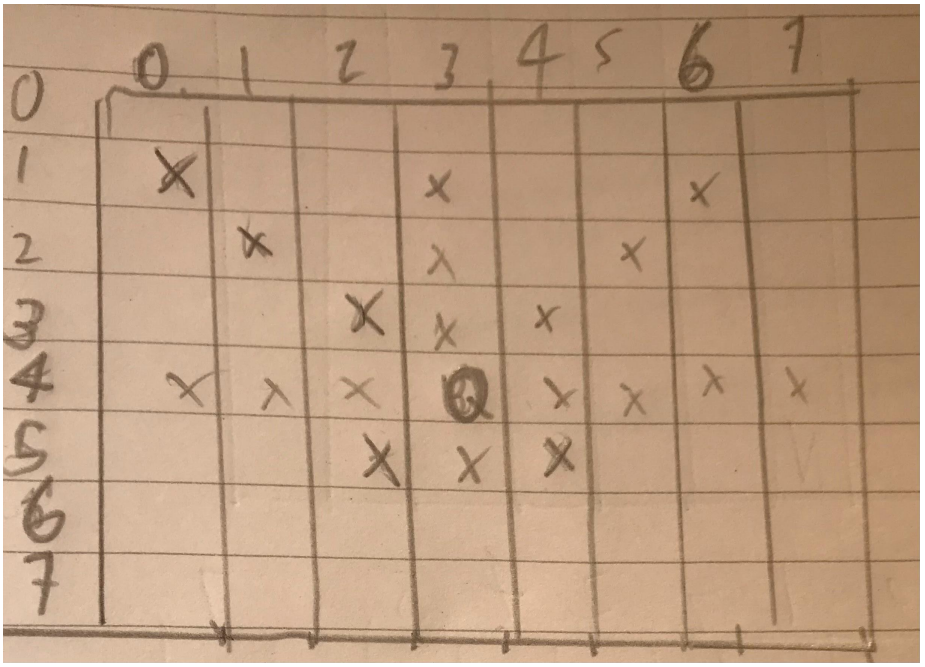
```
def getValidMoves(self, board):

    self.validMoves = []

    self.checkStaights(board)
    self.checkDiagonals(board)

    return self.validMoves
```

Testing Queen Legal Moves (15)

Placing the queen on (3, 4) tests moving to an empty square, capturing enemy pieces and not capturing friendly pieces.




```
Hello from the pygame community. https://www.pygame.org/contribute.html
(3, 5)
(4, 4)
(5, 4)
(6, 4)
(7, 4)
(2, 4)
(1, 4)
(0, 4)
(3, 3)
(3, 2)
(3, 1)
(4, 5)
(2, 5)
(4, 3)
(5, 2)
(6, 1)
(2, 3)
(1, 2)
(0, 1)
```



What's being tested	Inputs	Reasoning	Expected Outcome	Actual Outcome
Queen legal moves	None	Make sure every piece obeys the rules of chess	Valid list of position which the queen can move	Valid list of position which the queen can move

Legal moves for the queen are correct.

Prototyping Movement For Pawn - 08/01/2023

I saved the movement of the pawn for last because I thought it would be most difficult to code. Depending on which side of the board a pawn is, it will have to move; north if it's the player's pawn or south if it is the enemy's pawn.

Additionally, if the pawn has not moved then it has the option of moving two or one square in its given direction.

On top of that, pawns capture diagonally, not in the direction of movement like other pieces.

The attribute *direction* is added to *Pawn*,

```
self.direction = None
```

First I need to check the *player_colour* of the board, so I must make the *getPlayerColour()* function to do this.

```
def getPlayerColour(self):
    return self.player_colour
```

Then based on the player colour I can determine the *direction* of motion for a pawn.

```
def getValidMoves(self, board):

    self.validMoves = []

    move = None

    if board.getPlayerColour() == self.colour:
        self.direction = -1
    else:
        self.direction = 1
```

North is -1 and south is +1.

A pawn can always move one square in the given direction if there are no pieces obstructing its movement. This can be done by adding the *direction* to the y-coordinate.

```
# One squares forward
if board.getBoard()[self.positionY+self.direction][self.positionX] == 0:

    move =  Move((self.positionX, self.positionY), (self.positionX, self.positionY+self.direction))
    self.addMove(move)
```

If the pawn hasn't moved, then it has the option to move two squares, so add two times the *direction* to the y-coordinate if *moved* is False.

```
# Two squares forward
if (board.getBoard()[self.positionY+self.direction][self.positionX] == 0) and (board.getBoard()[self.positionY+2*self.direction][self.positionX == 0) and (self.moved == False):
    move =  Move((self.positionX, self.positionY), (self.positionX, self.positionY+2*self.direction))
    self.addMove(move)
```

Note that we don't have to be concerned with board boundaries when pushing pawns as when they reach the end of the board they will be promoted to queens.

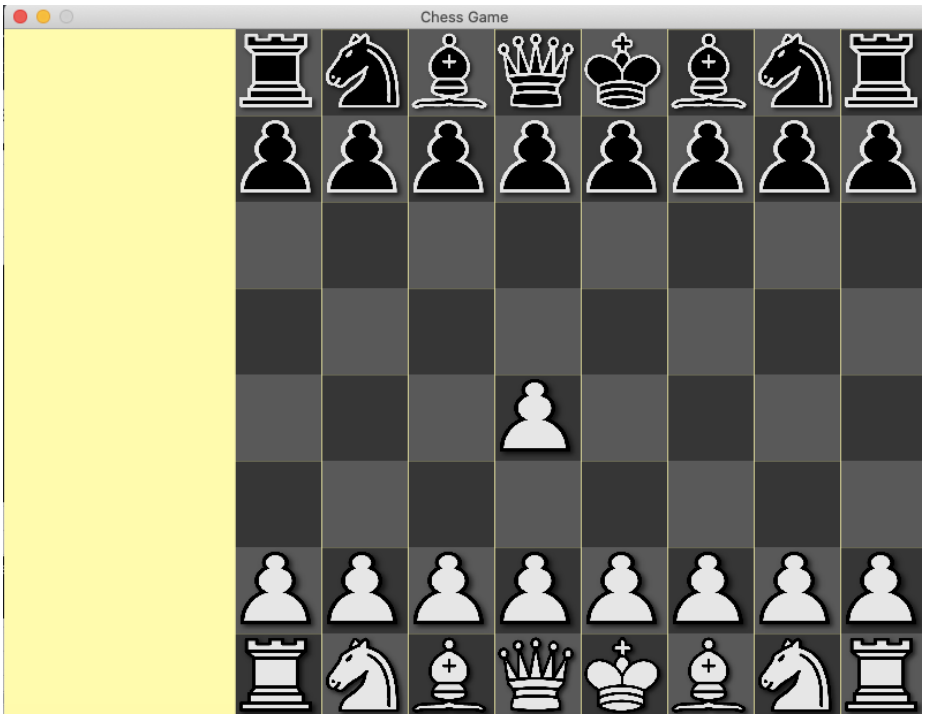
Pawn's can capture diagonally right by adding 1 to the x-coordinate, or diagonally left by subtracting 1 from the x-coordinate.

```
# Check right diagonal capture
if ((self.positionX + 1) < 8):
    if board.getBoard()[self.positionY+self.direction][self.positionX+1]!= 0:
        if board.getBoard()[self.positionY+self.direction][self.positionX+1].getColour()!= self.colour:
            move = Move((self.positionX, self.positionY), (self.positionX+1, self.positionY+self.direction))
            self.addMove(move)

# Check left diagonal capture
if ((self.positionX - 1) > 0):
    if board.getBoard()[self.positionY+self.direction][self.positionX-1]!= 0:
        if board.getBoard()[self.positionY+self.direction][self.positionX-1].getColour()!= self.colour:
            move = Move((self.positionX, self.positionY), (self.positionX-1, self.positionY+self.direction))
            self.addMove(move)

return self.validMoves
```

Testing Pawn Legal Moves (16)

A pawn is placed at (3, 4) where *moved* is set to False.

I didn't draw out the board for pawn movements as their positions are much simpler and easier to keep track of.




```
Hello from the pygame community. https://www.pygame.org/contribute.html
(3, 3)
(3, 2)
```

What's being tested	Inputs	Reasoning	Expected Outcome	Actual Outcome
White pawn pushes up by two	None	Make sure every piece obeys the rules of chess	Two new positions printed, in front of the current position, in correct direction	Two new positions printed, in front of the current position, in correct direction

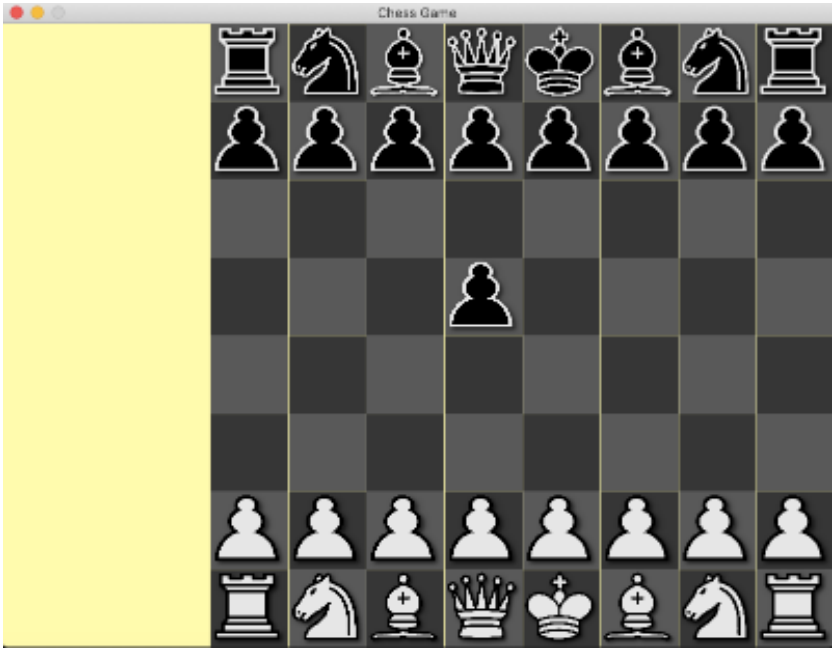
When moved is true with the pawn still at (3, 4);

```
self.moved = True
```

```
Hello from the pygame community. https://www.pygame.org/contribute.html
(3, 3)
```

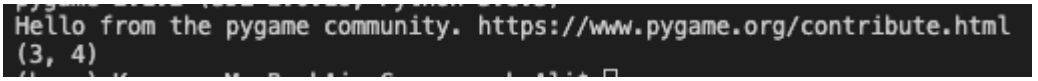
What's being tested	Inputs	Reasoning	Expected Outcome	Actual Outcome
White pawn pushes up by one	None	Make sure every piece obeys the rules of chess	One new positions printed, in front of the current position, in correct direction	One new positions printed, in front of the current position, in correct direction

Now check a black pawn on (3, 3), when *moved* is False.



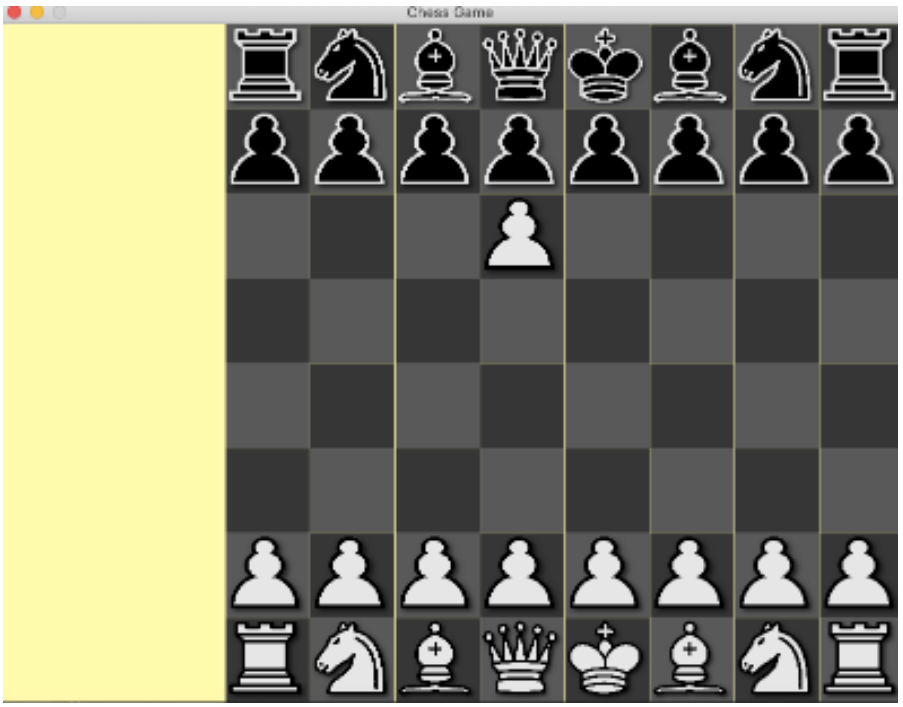
What's being tested	Inputs	Reasoning	Expected Outcome	Actual Outcome
Black pawn pushes up by two	None	Make sure every piece obeys the rules of chess	Two new positions printed, in front of the current position, in correct direction	Two new positions printed, in front of the current position, in correct direction

When black pawn *moved* is True;



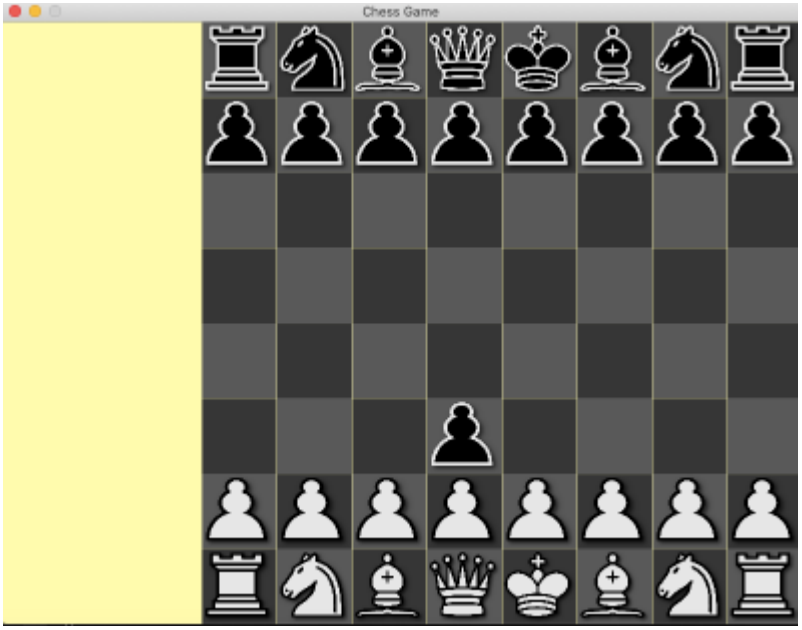
What's being tested	Inputs	Reasoning	Expected Outcome	Actual Outcome
Black pawn pushes up by two	None	Make sure every piece obeys the rules of chess	One new positions printed, in front of the current position, in correct direction	One new positions printed, in front of the current position, in correct direction

By placing a white pawn on (3, 2) I can check if a white pawn can capture diagonally and ensure it cannot capture forward.



What's being tested	Inputs	Reasoning	Expected Outcome	Actual Outcome
White pawn diagonal capture	None	Make sure every piece obeys the rules of chess	Two positions on right and left diagonal indicating a capture	Two positions on right and left diagonal indicating a capture

Placing a black pawn at (3, 5) can check for diagonal capture for the black pawn.



What's being tested	Inputs	Reasoning	Expected Outcome	Actual Outcome
Black pawn diagonal capture	None	Make sure every piece obeys the rules of chess	Two positions on right and left diagonal indicating a capture	Two positions on right and left diagonal indicating a capture

Legal moves for the queen are correct.

Client Feedback

I told [redacted] and [redacted] I can generate a list of legal moves for each piece. They asked if I could move the pieces on the board. I told them not yet. [redacted] told me to come back once they can move the pieces on the board as a list of legal moves means nothing if the pieces can't physically move.

Making and Prototyping The Board Clicker - 12/01/2023

Based on [redacted]'s instructions, I decided to find a way to move the pieces on the board.

I started by making the *boardClicker()* function in *Board*. It takes in the parameters *clickedX* and *clickedY* which are the x and y coordinates of the mouse.

I used an inverse function to get the mouse coordinates and convert them into square coordinates;

- For x, it is (*clickedX*-200) // 75
- For y, it is (*clickedY* // 75)

I wrote tha *getSquareFromPos()* method to return a square from the board by inserting its coordinates.

```
def getSquareFromPos(self, pos):
    for square in self.squares:
        if square.getPosition() == pos:
            return square
```

```
def boardClicker(self, clickedX, clickedY):

    x = (clickedX-200)//75
    y = (clickedY//75)

    clicked_square = self.getSquareFromPos((x, y))
```

Next I have to select the piece which I clicked on. To do this I check if the board currently has a piece selected, and if *clicked_square* has a piece on it and the *colour* of that piece is the same as the colour of the player whose turn it is, then I want to select the piece.

```
if self.piece_selected is None:
    if clicked_square.getPieceOnSquare() is not None:
        if clicked_square.getPieceOnSquare().getColour() == self.turn:
            self.piece_selected = clicked_square.getPieceOnSquare()
```

Lastly, if a piece is selected, I want to set its original position to 0 indicating that its black, place the piece on the new square, say the piece has been moved, flip the *turn* of the board, and set the *piece_selected* to None.

```
elif self.piece_selected.getColour() == self.turn:

    # Set original space to blank
    self.board[self.piece_selected.getPositionY()][self.piece_selected.getPositionX()] = 0

    # Change to new position
    self.piece_selected.move(clicked_square.getPosition())
    self.board[self.piece_selected.getPositionY()][self.piece_selected.getPositionX()] = 0

    # Piece has been moved
    self.piece_selected.setMovedTrue()

    # if turn is white flip to black, else flip to white
    self.turn == "white" if self.turn == "black" else "black"
    self.piece_selected = None
```

By using pygame to track the mouse position and using the *MOUSEBUTTONDOWN* event handler, I get the coordinates of my mouse on the screen and call *boardClicker()* when *MOUSEBUTTONDOWN* occurs. This takes place in my game loop.

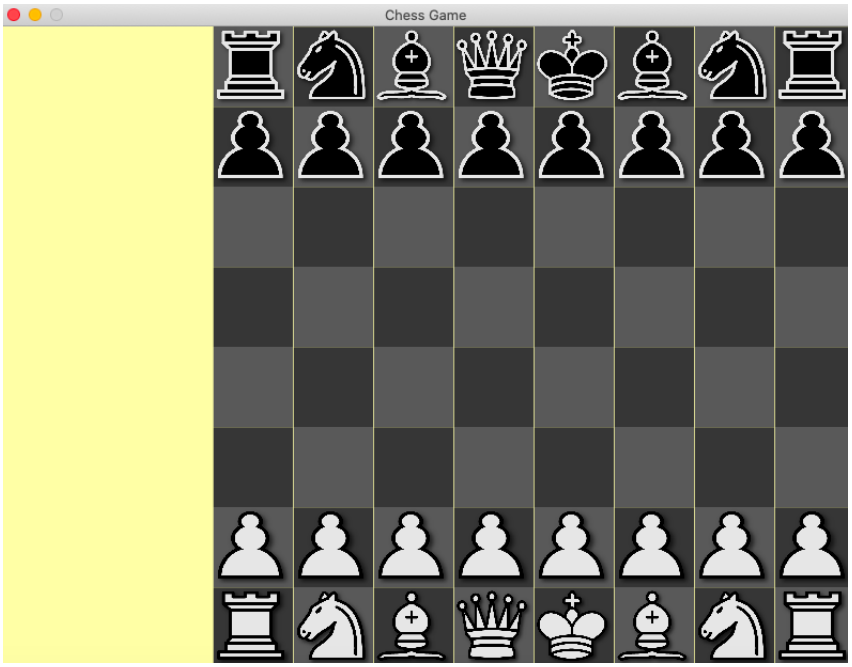
```
clickedX, clickedY = pygame.mouse.get_pos()

for event in pygame.event.get():

    if event.type == pygame.QUIT:

        pygame.quit()
        exit()
    elif event.type == pygame.MOUSEBUTTONDOWN:

        b.boardClicker(clickedX, clickedY)
```

Testing Board Clicker (17)

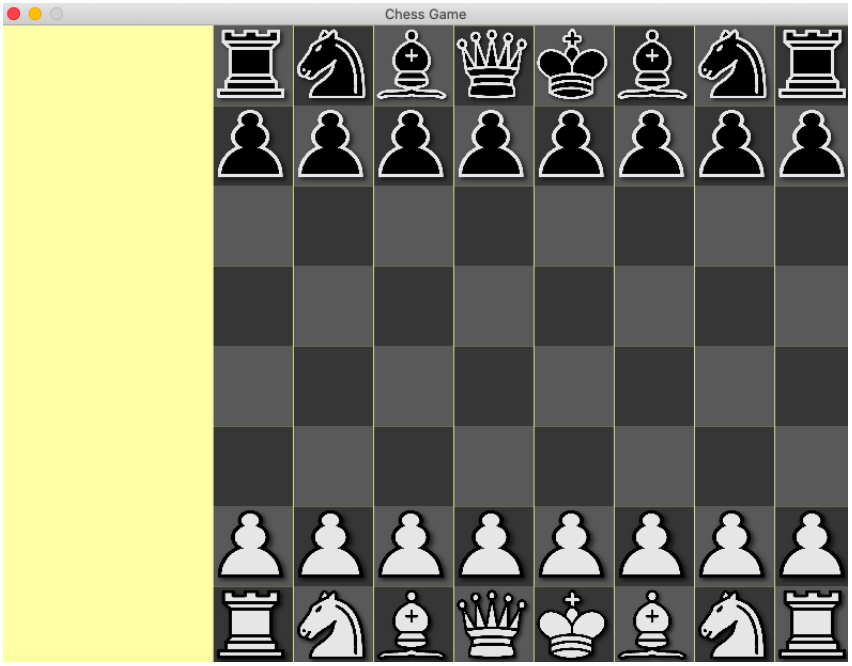
When I click on the white light square bishop when the turn is white, I expect it to be selected. Then when I click another square, I expect the light square bishop to disappear from its current square and move to its new square.

When I try this, the bishop doesn't move.

Before;



After;



What's being tested	Inputs	Reasoning	Expected Outcome	Actual Outcome
Moving white light square bishop from original square to new square	Mouse x and y coordinates on screen	So user can move a piece on the board to play game	Bishop moves to new square	Bishop does not move

Something in my code is wrong, and I have no idea what. So I started systematically testing different aspects of the *boardClciker()*.

I tested to see if my program could detect if I have clicked with the mouse by adding a print() statement every time the event *MOUSEBUTTONDOWN* has occurred.

```
    exit()
elif event.type == pygame.MOUSEBUTTONDOWN:

    b.boardClicker(clickedX, clickedY)

    print('clicked')
```

When I click the screen;

```
Hello from the pygame community. https://ww
w.pygame.org/contribute.html
clicked
clicked
```

What's being tested	Inputs	Reasoning	Expected Outcome	Actual Outcome
If program detect when mouse has clicked	Click on screen	See if the problem with *boardClicker()* is that mouse is not clicking	"clicked" printing every time screen is clicked	"clicked" printing every time screen is clicked

This is not the problem.

Then I decided to print *clickedX* and *clickedY* to the screen to see if it was recording the correct position of the mouse.

```
for event in pygame.event.get():

    if event.type == pygame.QUIT:

        pygame.quit()
        exit()
    if event.type == pygame.MOUSEBUTTONDOWN:

        b.boardClicker(clickedX, clickedY)

    print(clickedX, clickedY)
```

```
242 159
242 Terminal
242 159
409 330
409 330
366 395
366 395
443 75
500 349
493 365
456 481
```

What's being tested	Inputs	Reasoning	Expected Outcome	Actual Outcome
Mouse position coordinates	Mouse moving around on screen	See if the problem with *boardClicker()* is that it is not getting the mouse position	Sensible coordinates of the mouse based on where it is on the screen	Sensible coordinates of the mouse based on where it is on the screen

This is not the problem either.

Now I'll check if the *boardClicker()* function is being called when the mouse is clicked. I placed a print statement in function for testing.

```
def boardClicker(self, clickedX, clickedY):

    x = (clickedX-200)//75
    y = (clickedY//75)

    clicked_square = self.getSquareFromPos((x, y))

    print('hi im working!!!')
```

```
Hello from the pygame community. https://www.pygam
e.org/contribute.html
hi im working!!!
hi im working!!!
```

What's being tested	Inputs	Reasoning	Expected Outcome	Actual Outcome
If *boardClicker()* function is being called upon	Mouse click on screen	See if the problem with *boardClicker()* is that it is not getting called upon	Print statement on "hi im working!!!"	Print statement on "hi im working!!!"

This is not the problem.

Then I decided to check if the inverse function to get the square position from coordinates worked properly by printing *x* and *y*.

```
def boardClicker(self, clickedX, clickedY):

    x = (clickedX-200)//75
    y = (clickedY//75)

    print(x, y)
```

```
TERMINAL
7 6
0 7
0 7
2 7
3 7
3 7
5 7
6 7
6 7
7 7
```

What's being tested	Inputs	Reasoning	Expected Outcome	Actual Outcome
Check is inverse function to get square position works	Clicking on square	See if the problem with *boardClicker()* is that the square position function does not work	When I click on a square, the position of that square prints	When I click on a square, the position of that square prints

This is not the problem.

I tested whether or not the value of *piece_selected* ever changes from None, so I printed the piece selected.

```
print(self.piece_selected)
self.turn = "white" if self.turn == "black" else "black"
```

When I try to click on any white piece on the board;

```
Hello from the pygame community. https://www.pygame.org/contribute.html
None
None
```

What's being tested	Inputs	Reasoning	Expected Outcome	Actual Outcome
If piece selected is stored	Mouse position	See if the problem with *boardClicker()* is that the piece is not being selected	Piece object will be printed	"None" is printed

The problem is that the piece is not being selected.

Looking through my code, I have no clue why the piece won't be selected so I decided to look at it the next day.

Prototyping The Board Clicker Continued - 18/01/2023

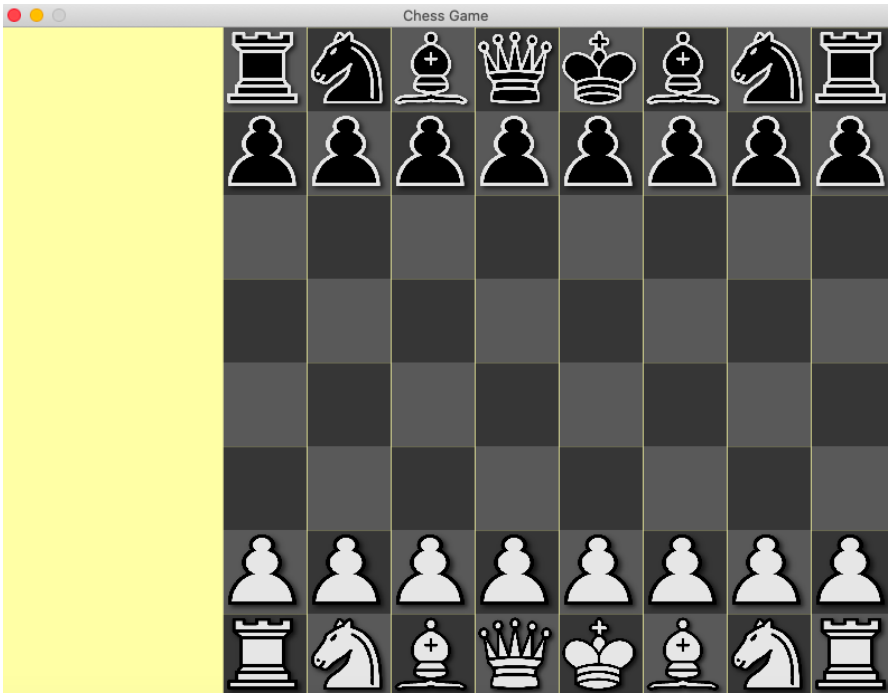
I was so irritated that the code wouldn't work so I took a few days break from the project. I decided to not look at my old code for *boardClicker(),* and instead try to rewrite with a clear head.

```
def boardClicker(self, clickedX, clickedY):

    x = (clickedX-200)//75
    y = (clickedY//75)

    for square in self.squares:
        if square.getPosition() == (x, y):
            if square.getPieceOnSquare() is not None:
                if self.turn == square.getPieceOnSquare().getColour():
                    self.piece_selected = square.getPieceOnSquare()
                    square.setPieceOnSquare(None)

                elif self.piece_selected is not None:
                    self.board[y][x] = self.piece_selected
                    self.piece_selected.setMovedTrue()
                    square.setPieceOnSquare(self.piece_selected)
                    self.piece_selected = None
```

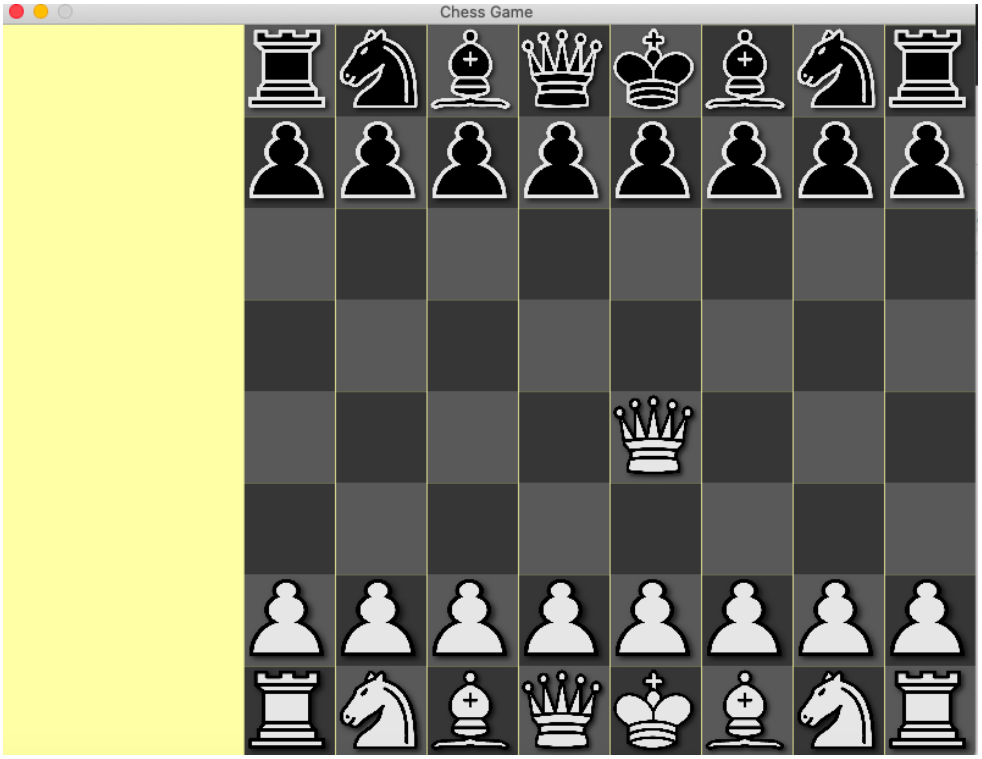
Testing Board Clicker Again (18)

I tested the board clicker again by clicking on the white queen and moving it to the middle of the board.

Before;



After;



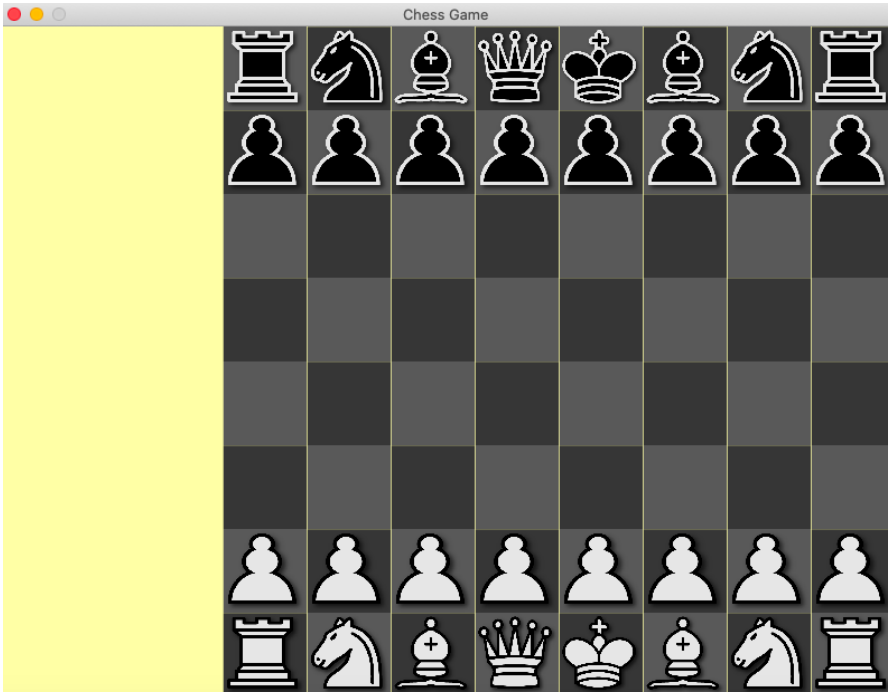
What's being tested	Inputs	Reasoning	Expected Outcome	Actual Outcome
boardClicker() function	Mouse position	See if *boardClicker()* finally works	Queen should move from original position to new position	Queen duplicates onto new position and remains on original position

The function is creating a duplicate of a piece and not removing an original. This is because I forgot to set the original space of the board to zero.

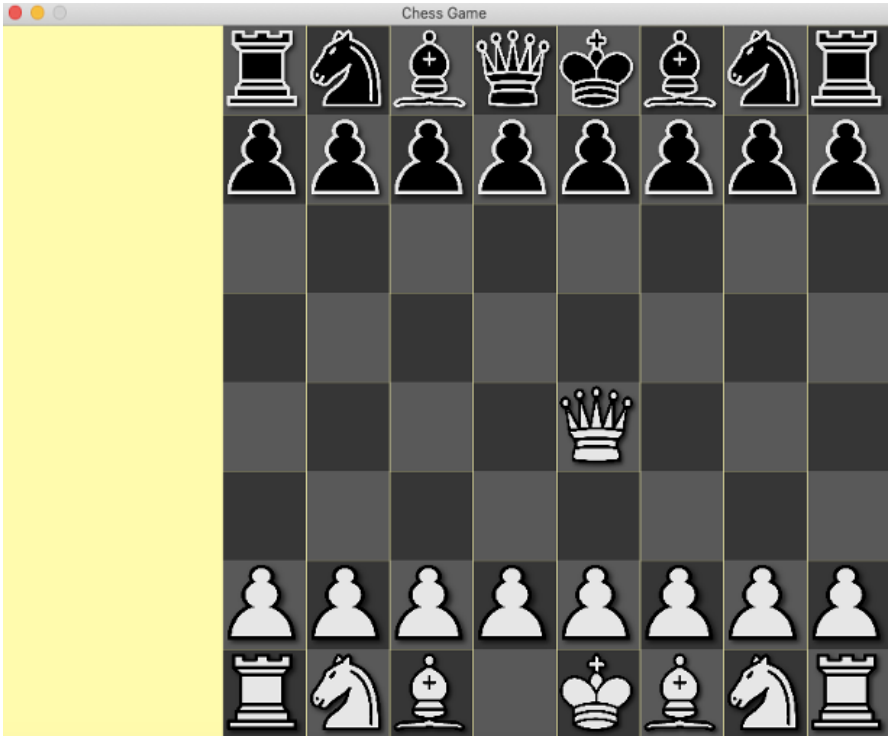
```
if square.getPieceOnSquare() is not None:
    if self.turn == square.getPieceOnSquare().getColour():
        self.piece_selected = square.getPieceOnSquare()
        self.board[y][x] = 0
        square.setPieceOnSquare(None)
```

So I tested again.

Before;



After;



What's being tested	Inputs	Reasoning	Expected Outcome	Actual Outcome
boardClicker() function	Mouse position	See if *boardClicker()* finally works	Queen should move from original position to new position	Queen moves from original position to new position

It finally works!

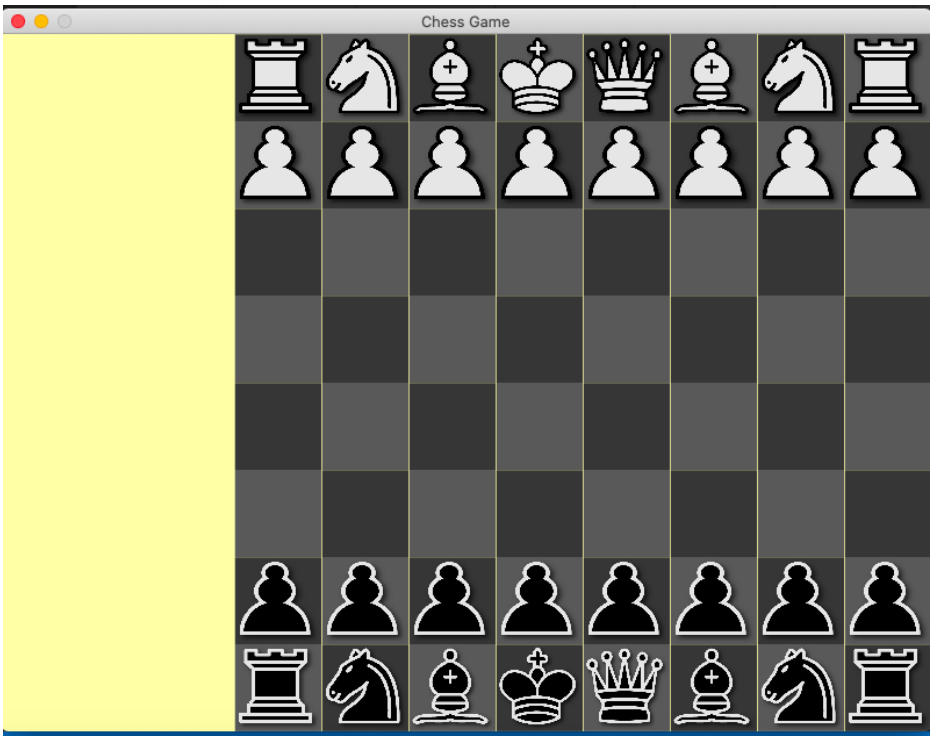
Now I will test with the black pieces by setting *player_colour* to "black" and the board *turn* to "black".

```
b = Board("gray", "black")
```

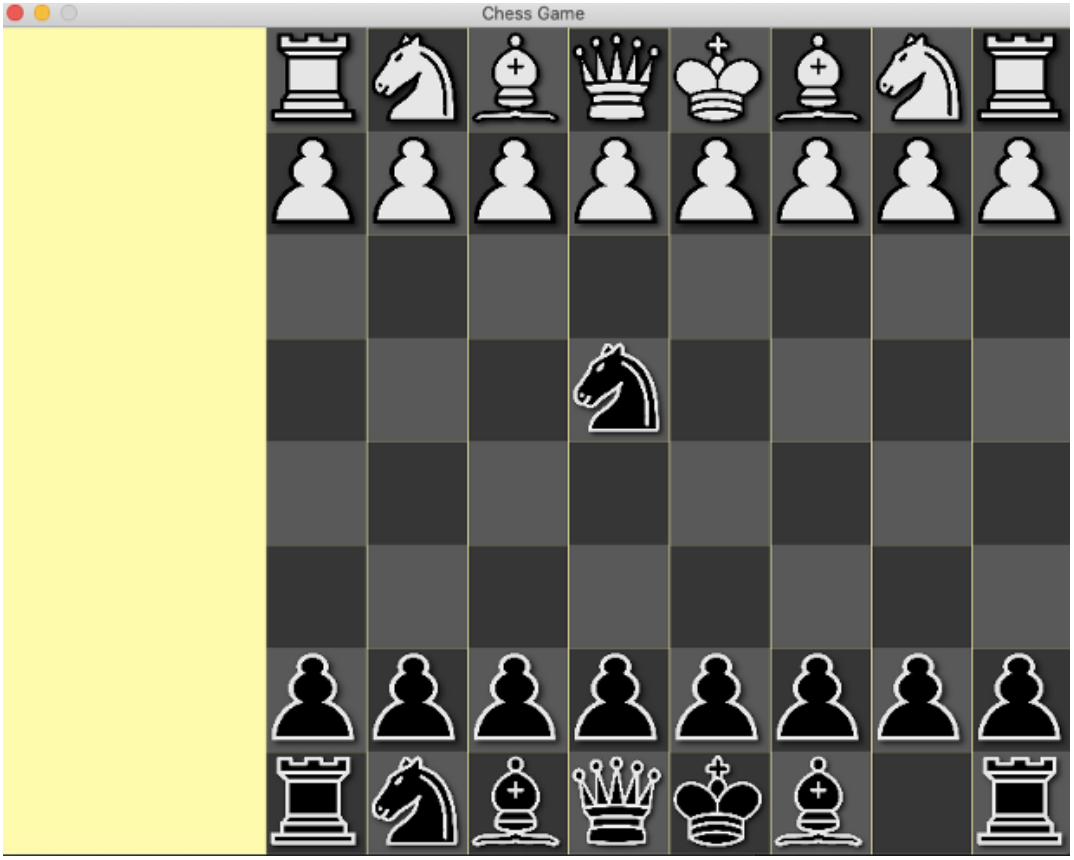
```
self.turn = 'black'
```

I tested moving the black knight to the middle of the board.

Before;



After;



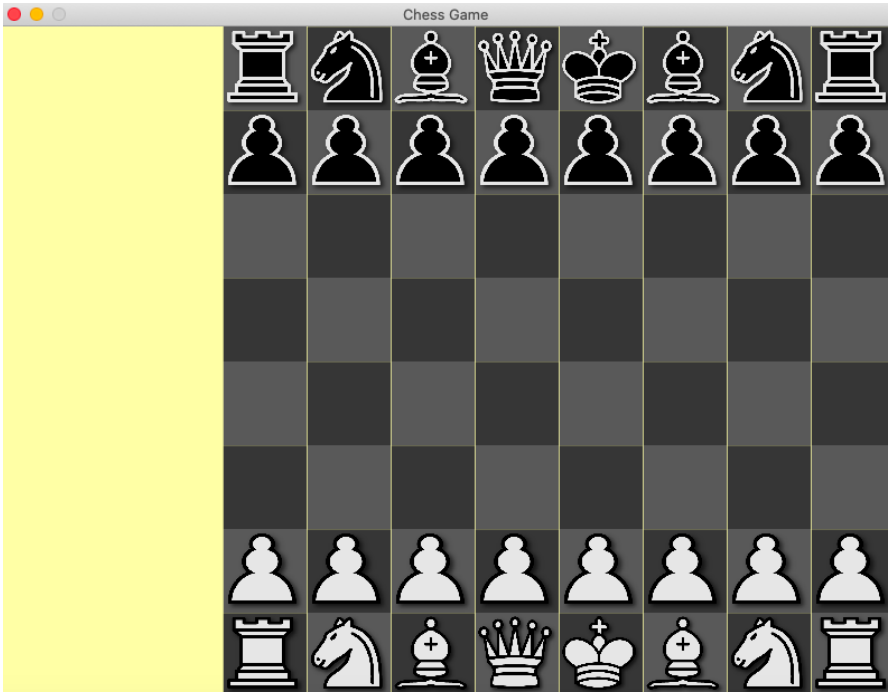
What's being tested	Inputs	Reasoning	Expected Outcome	Actual Outcome
boardClicker() function	Mouse position	See if *boardClicker()* works when player is black	Knight should move from original position to new position	Knight moves from original position to new position

The function works for black.

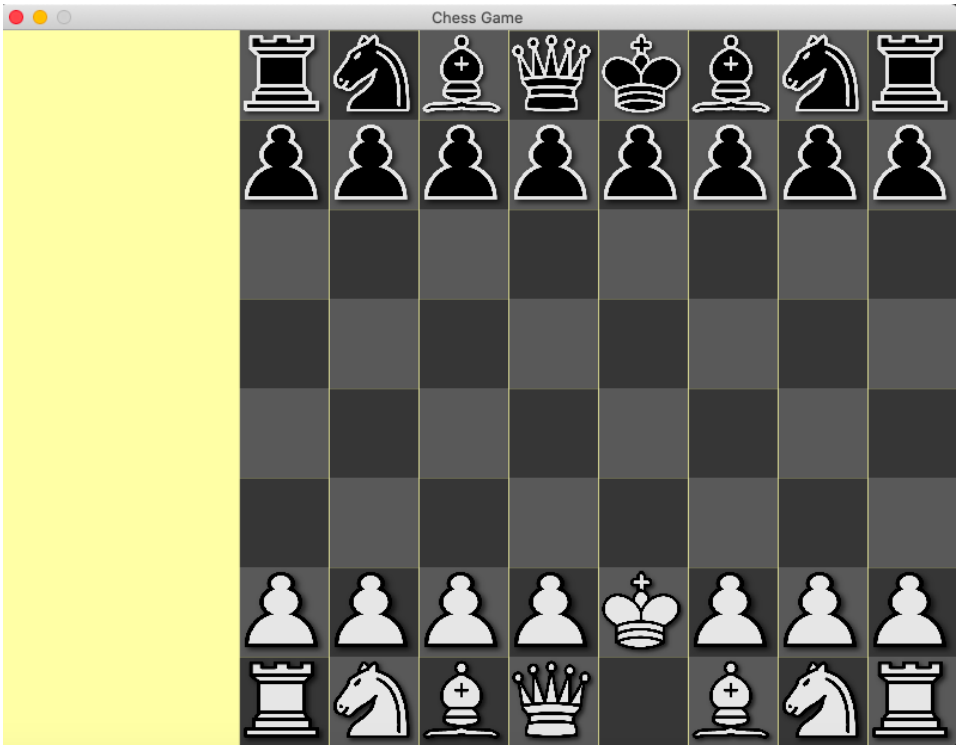
Client Feedback

I was very excited to show [redacted] and [redacted] that they were now able to move pieces. They seemed happy with the ability to move. While they were prototyping, they found that the same coloured pieces were able to capture each other.

Before;



After;



[redacted] asked why this was possible. I told them that this will be fixed when I linked *boardClicker()* with a list of legal moves, which is my next step. He understood and said they looked forward to testing it then.

Generating Legal Moves List And Connecting It To The Board Clicker - 21/01/2023

Within the *Board* class, I need to get a list of all the moves allowed on a board in its current state.

This is what *getMoves()* is for.

```
def getMoves(self):

    # Gets all possible moves for player whos turn it is
    piece = None
    moves = []
    piece_moves = []
    self.moves = []

    for i in range(0, 8):
        for j in range(0, 8):
            piece = self.board[i][j]
            if piece != 0:
                # Only get moves if that colour's turn
                if piece.getColour() == self.turn:
                    # Add every move of that piece onto list
                    piece_moves = piece.getValidMoves(self)
                    for move in piece_moves:
                        moves.append(move)
    self.moves = moves
    return moves
```

This function returns a list of all possible moves that can be made by the player whose turn it currently is. It does this by iterating through all the pieces on the board, and for each piece that belongs to the current player, it calls the piece's *getValidMoves()* method to obtain a list of all valid moves for that piece. Each valid move is then added to a list of all possible moves, which is returned at the end of the function.

I also wrote the *movePiece()* method as I will probably need it later (when the computer makes a move).

```
def movePiece(self, move):

    # Select piece and set space to blank
    piece = self.board[move.getOPos()[1]][move.getOPos()[0]]
    self.board[move.getOPos()[1]][move.getOPos()[0]] = 0

    # Move piece
    piece.setPos(move.getNPos())
    piece.setMovedTrue()
    self.board[move.getNPos()[1]][move.getNPos()[0]] = piece

    # Set square to blank
    for square in self.squares:
        if square.getPosition() == move.getOPos():
            square.setPieceOnSquare(None)

    # Set piece on square
    for square in self.squares:
        if square.getPosition() == move.getNPos():
            square.setPieceOnSquare(piece)


    self.turn = "white" if self.turn == "black" else "black"
```

The method first selects the piece that needs to be moved, sets the space where it was to blank, and then moves the piece to its new position. After that, it sets the square where the piece moved from to blank, and sets the piece on the new square. Finally, it switches the turn of the player, so the other player can make their move.

I then edited *boardClicker()* by adding the *can_move* and *selectedMove* variables.

```
def boardClicker(self, clickedX, clickedY):

    self.getMoves()

    # Inverse function to get square coordinates
    x = (clickedX-200)//75
    y = (clickedY//75)

    can_move = False
    selectedMove = None

    if self.piece_selected is not None:
        for piece_move in self.piece_selected.getValidMoves(self):
            if (piece_move.getOPos() == self.piece_selected.getPos()) and (piece_move.getNPos() == (x, y)):

                for move in self.moves:
                    if (move.getOPos() == piece_move.getOPos()) and (move.getNPos() == piece_move.getNPos()):

                        can_move = True
                        selectedMove = move
                        break

    else:
        for move in self.moves:
            if move.getOPos() == (x, y):
                can_move = True
```

The function checks if a piece is already selected. If so, it loops through the valid moves for that piece and checks if the clicked square is a valid move. If it is, it loops through all possible moves and checks if the selected move is one of them. If it is, it sets *can_move* to True and sets the *selectedMove* variable to the move that was clicked.

If no piece is selected, the function simply loops through all possible moves and checks if the clicked square is a valid starting square for any of the moves.

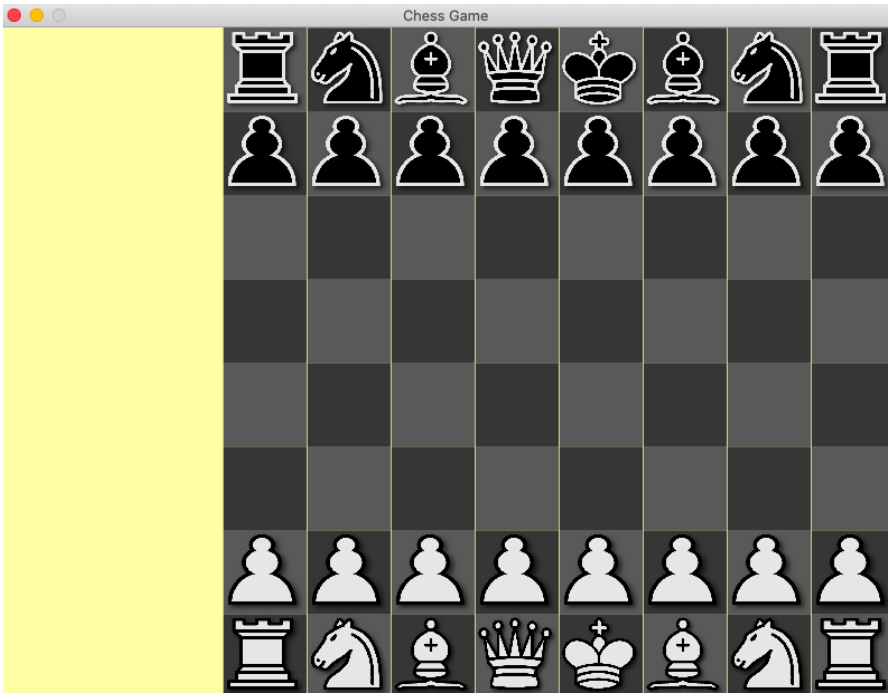
The function is edited further to only work if *can_move* is True. *movePiece()* is used throughout to simplify the code.

```
# Iterate throught squares to get square with clicked coordinates
for square in self.squares:
    if square.getPosition() == (x, y) and (can_move):

        # If there is a piece on that square
        if (square.getPieceOnSquare() is not None) and (self.player_colour == self.turn):

            # Select a piece
            if (self.turn == square.getPieceOnSquare().getColour()) and (self.piece_selected is None):
                self.piece_selected = square.getPieceOnSquare()
                square.setPieceOnSquare(None)

            # Capture a piece
            elif (self.piece_selected is not None) and (square.getPieceOnSquare().getColour() != self.piece_selected.getColour()):
                self.movePiece(selectedMove)
                self.piece_selected = None

        # Move to empty square
        elif (self.player_colour == self.turn) and (self.piece_selected is not None):
            self.movePiece(selectedMove)
            self.piece_selected = None
```

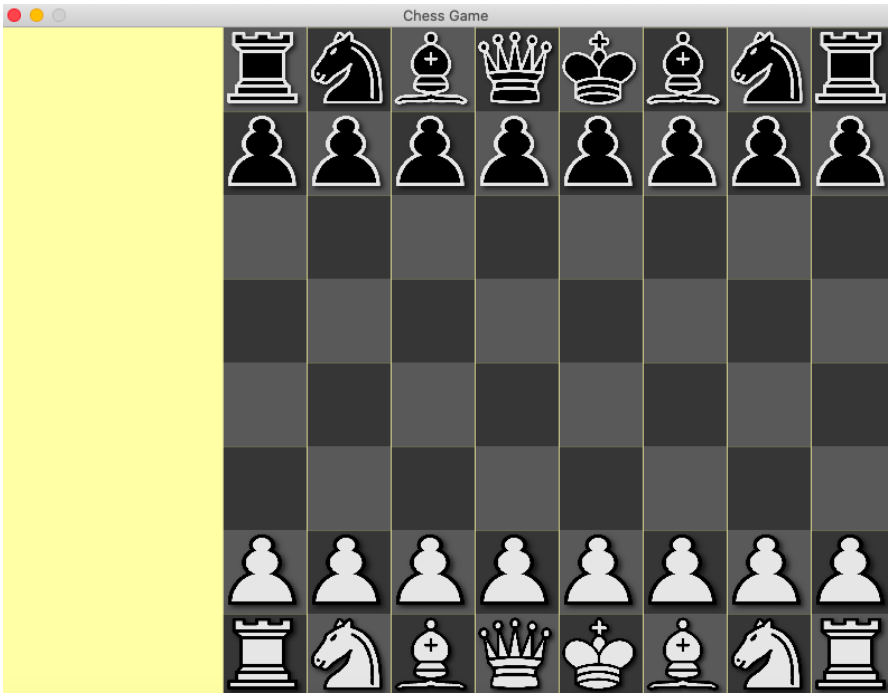
Testing Moving To Valid Moves (19)

I am going to try to move the white knight from its original square to an illegal square, then try to move it to a legal square.

Before;

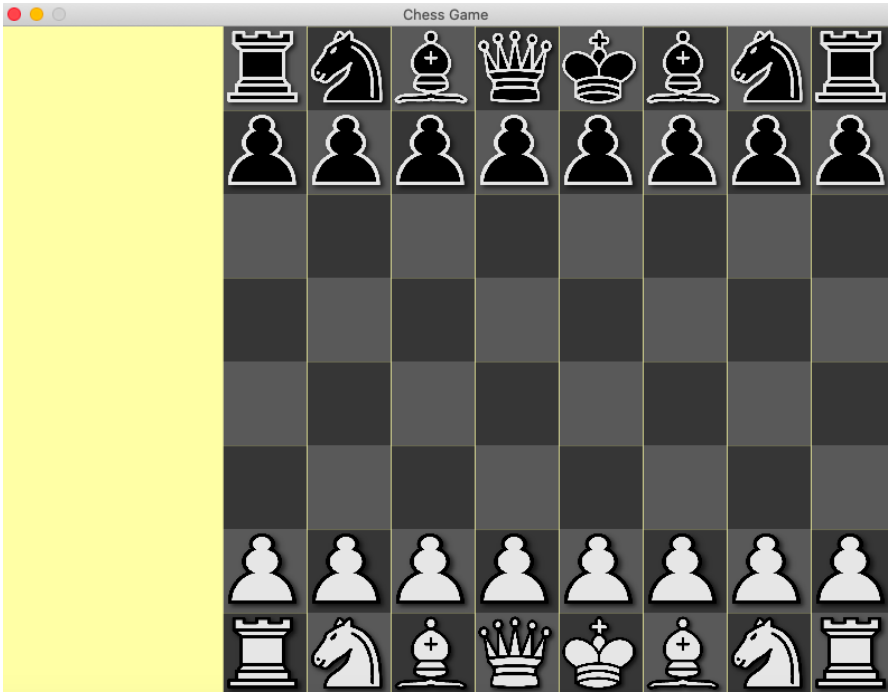


After trying to move to illegal square;

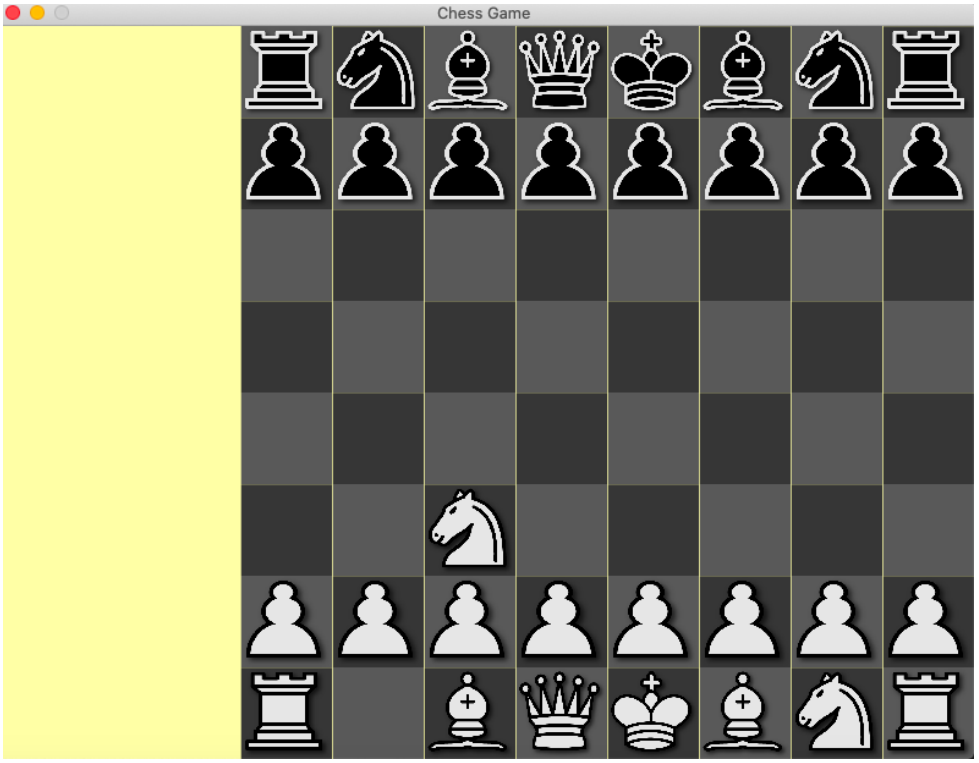


Now try a legal move for the knight.

Before;



After legal move played;



What's being tested	Inputs	Reasoning	Expected Outcome	Actual Outcome
Moving knight to illegal space and moving knight to legal space	None	Ensure pieces can only make legal moves	Knight should move from original position to legal position, not illegal position	Knight moves from original position to legal position, not illegal position

Only legal moves can be made.

Client Feedback

I showed [redacted] and [redacted] that they could make legal moves. They tested this with different pieces and [redacted] attempted to castle (a move which moves the king and rook at the same time), and found that he couldn't do it. He asked me about this and I realised that I had completely forgotten to code in special moves (castling and pawn promotion). I decided to code on castling.

Castling and Prototyping Castling - 22/01/2023

Castling is a move involving an unmoved king and an unmoved rook. It is a move that moves the king two spaces toward the rook and places the rook next to the king on the opposite side.

The *canCastle()* method is placed in the King class, as castling is considered to be the king's move. It takes in the parameter *board*, to check if given the current board state castling is possible.

```
def canCastle(self, board):

    # Variables for castle flags and pieces
    piece = None
    castleLeft = None
    castleRight = None
    leftRook = None
    rightRook = None

    # Find corresponding rook on the board
    for i in range(0, 8):
        for j in range(0, 8):
            piece = board.getBoard()[i][j]
            if piece != 0:

                # Check if piece is a rook, if it is the same colour, if it hasn't moved and the king hasn't moved
                if (piece.getColour() == self.colour) and (piece.getMoved() == False) and (piece.getType() == "rook") and (self.moved == False):

                    # Check which direction the king can castle based on the position of the rook
                    if piece.getPositionX() < self.positionX:
                        # Set left rook flag to true if left rook is found
                        castleLeft = True
                        leftRook = piece

                    if piece.getPositionX() > self.positionX:
                        # Set right rook flag to true if right rook is found
                        castleRight = True
                        rightRook = piece
```

```
# Check if squares between left rook and king are  blank, else the king can't castle
if castleLeft:
    for i in range(leftRook.getPositionX()+1, self.positionX):
        if board.getBoard()[self.positionY][i] != 0:
            castleLeft = False

# Check if squares between right rook and king are  blank, else the king can't castle
if castleRight:
    for i in range(self.positionX+1, rightRook.getPositionX()):
        if board.getBoard()[self.positionY][i]!= 0:
            castleRight = False

# If flag for left rook is true, then add minus two spaces in x direction to moves list
if castleLeft:
    self.addMove(Move(self.getPos(), (self.positionX-2, self.positionY)))

# If flag for right rook is true, then add two spaces in x direction to moves list
if castleRight:
    self.addMove(Move(self.getPos(), (self.positionX+2, self.positionY)))
```

The method first initialises some variables for storing the flags *castleLeft* and *castleRight*, and memory locations (*piece*, *leftRook*, *rightRook*) of the pieces related to the castle move. It then loops over every square on the -board and looks for a rook that belongs to the same colour as the king, has not moved yet, and is on the same row as the king. If such a rook is found, the method determines whether the king can castle to the left or right side by comparing the rook's position to the king's position.

If the king can castle to the left, the method checks that the squares between the left rook and the king are empty. Similarly, if the king can castle to the right, the method checks that the squares between the right rook and the king are empty. If either check fails, the corresponding castle flag is set to False.

Finally, if the castle flags are still True, the method adds the corresponding castle move to the king's list of possible moves.

The *canCastle()* method is called within the *King*'s *getValidMoves()* method to add castling to the list of *validMoves*.

```
# Check for castled moves
self.canCastle(board)
```

The last thing we need is to amend the *movePiece()* method to move the rook if the player attempts to castle.

We need to first create a memory location to store the *castled_rook*.

```
# If castle store, rook here
castled_rook = None
```

Then we can write the code in *movePiece()* to castle.

First, the code checks if the piece making the move is a king, and if the move involves castling. It does this by checking if the new position is two squares left or two squares right. If so, it stores the rook involved in the castling move in the *castled_rook* variable.

```
# Check for castle and store rook
if piece.getType() == "king":

    if move.getOPos() == (piece.getPositionX()-2, piece.getPositionY()):
        castled_rook = self.board[move.getNPos()[1]][7]

    elif move.getOPos() == (piece.getPositionX()+2, piece.getPositionY()):
        castled_rook = self.board[move.getNPos()[1]][0]
```

Next, the code sets the square where the rook was originally located to be blank, and updates the *piece_on_square* property of the *Square* instance that corresponds to that square to None.

```
# Set rook's square to blank if castled
if castled_rook is not None:

    self.board[castled_rook.getPositionY()][castled_rook.getPositionX()] = 0

    for square in self.squares:
        if square.getPosition() == castled_rook.getPos():
            square.setPieceOnSquare(None)
```

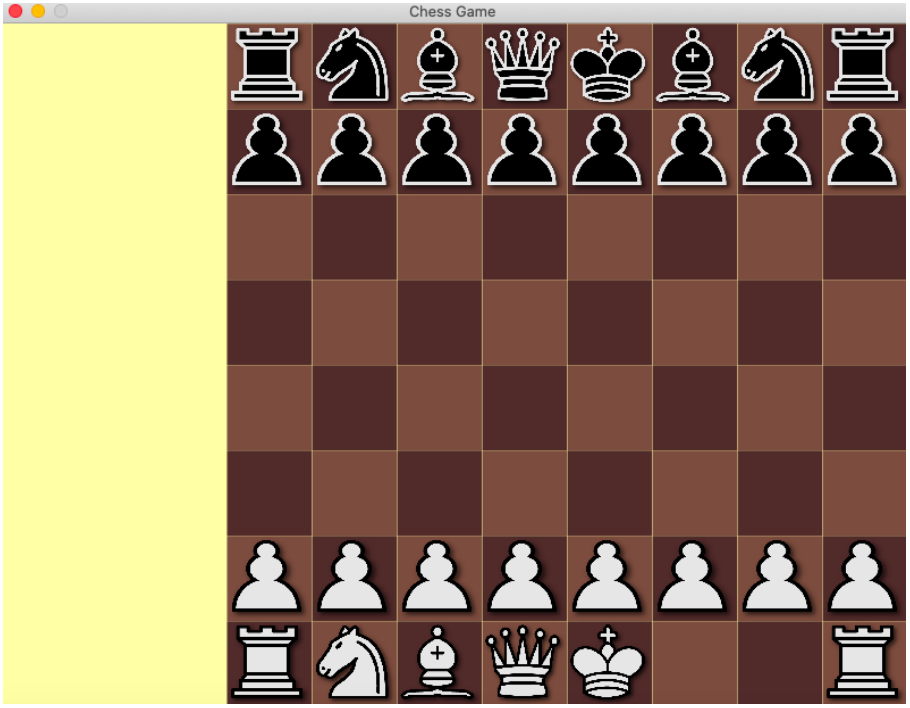
Then, the code determines where the rook should be placed after castling. If the rook is castling to the left, it will be placed one square to the right of the king. If the rook is castling to the right, it will be placed one square to the left of the king.

After the new position of the rook is determined, the code updates the position of the rook and sets the *moved* attribute of the rook to True. It also updates the *piece_on_square* property of the *Square.*

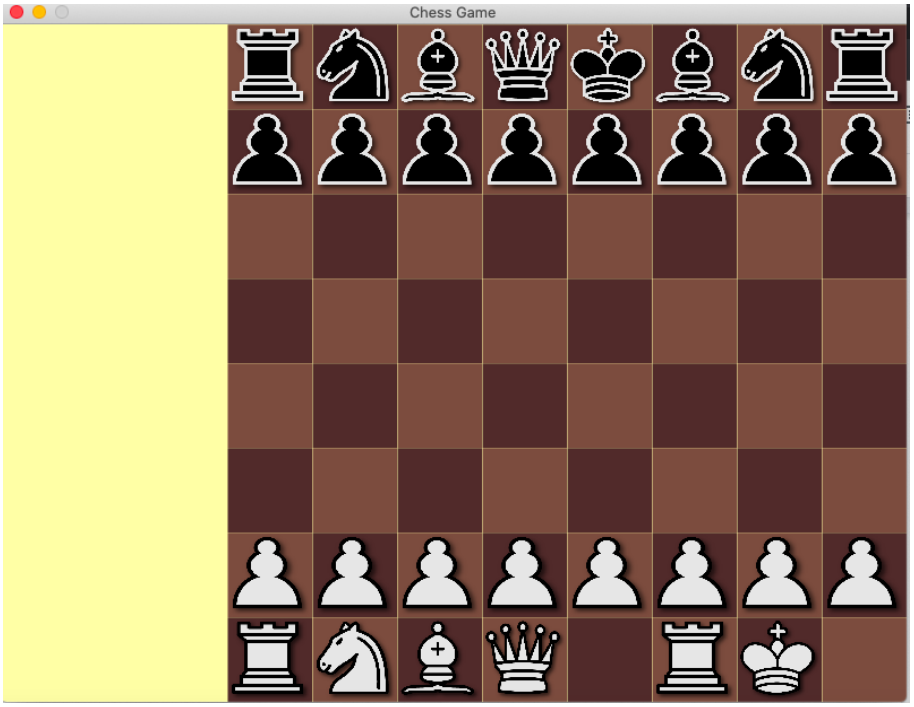
```
# Set rook's square to blank if castled
if castled_rook is not None:

    self.board[castled_rook.getPositionY()][castled_rook.getPositionX()] = 0

    for square in self.squares:
        if square.getPosition() == castled_rook.getPos():
            square.setPieceOnSquare(None)


    # Place rook on new square if castled
    if castled_rook.getPos() == (7, move.getNPos()[1]):
        castled_rook.setPos((piece.getPositionX()-1, piece.getPositionY()))
        castled_rook.setMovedTrue()
        self.board[piece.getPositionY()][piece.getPositionX()-1] = castled_rook

    else:
        castled_rook.setPos((piece.getPositionX()+1, piece.getPositionY()))
        castled_rook.setMovedTrue()
        self.board[piece.getPositionY()][piece.getPositionX()+1] = castled_rook

    for square in self.squares:
        if square.getPosition() == castled_rook.getPos():
            square.setPieceOnSquare(castled_rook)
```

Testing Castling (20)

Castling on the white's king's side is going to be tested by removing the bishop and knight from between the rook and king.

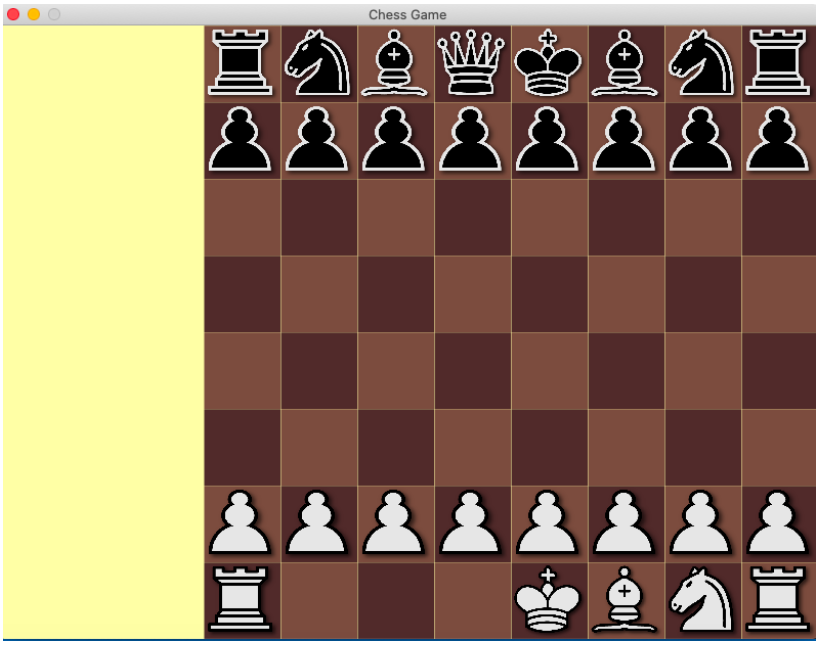


Then the castling move is attempted by trying to move the king two squares right.

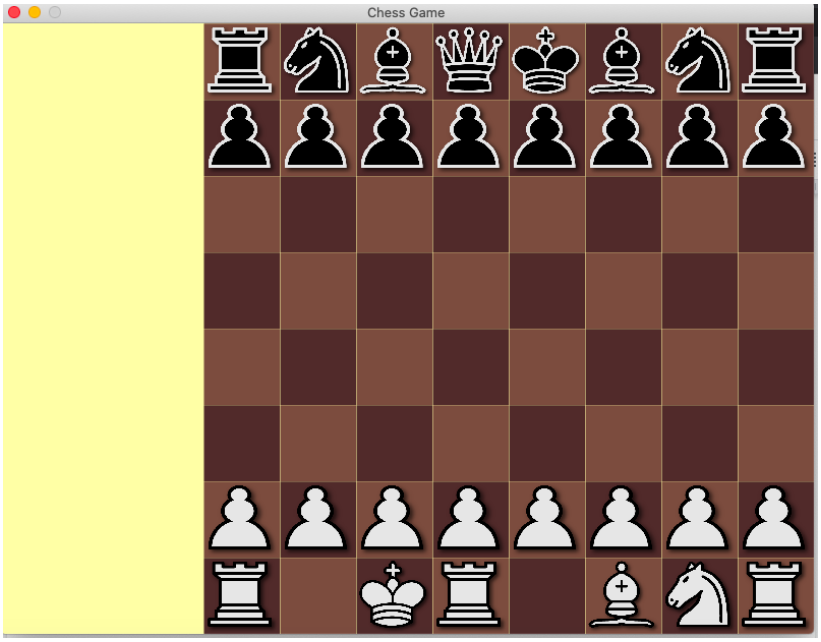


What's being tested	Inputs	Reasoning	Expected Outcome	Actual Outcome
Castling on white's king's side	Click two squares to right of white king	Test if the castling move works correctly	King moves two space right and rook moves to the left of king	King moves two space right and rook moves to the left of king

Testing of white's castle on the queen's side by removing the queen, bishop and knight.



Attempt to castle by clicking two squares to the left of the king.



What's being tested	Inputs	Reasoning	Expected Outcome	Actual Outcome
Castling on white's queen's side	Click two squares to left of white king	Test if the castling move works correctly	King moves two space left and rook moves to the right of king	King moves two space left and rook moves to the right of king

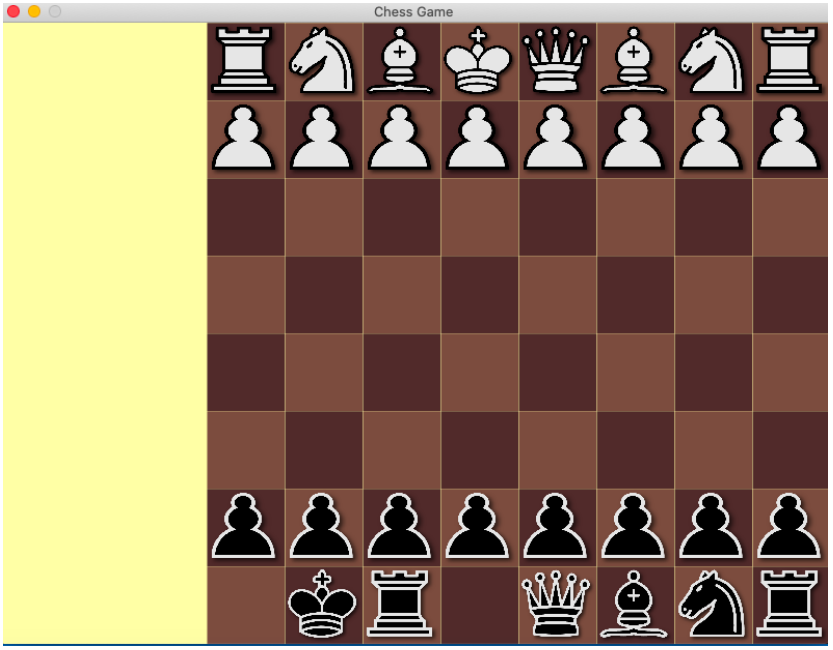
To test castling for black, the board's *turn* is set to "black" and *player_colour* is set to "black".

```
self.turn = 'black'
```

```
b = Board("brown", "black")
```

For castling on the king's side, remove the bishop and knight.

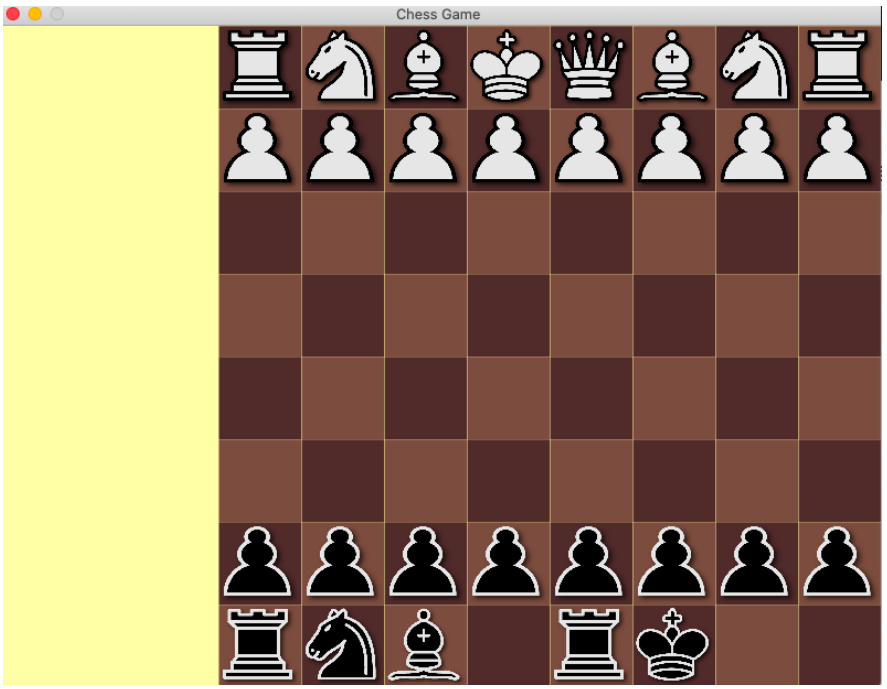
Then click two squares to the left of the king to try and castle.



What's being tested	Inputs	Reasoning	Expected Outcome	Actual Outcome
Castling on black king's side	Click two squares to left of black king	Test if the castling move works correctly	King moves two space left and rook moves to the right of king	King moves two space left and rook moves to the right of king

Testing of black's castle on the queen's side by removing the queen, bishop and knight.

Then click two squares to the right of the king to try the castle.



What's being tested	Inputs	Reasoning	Expected Outcome	Actual Outcome
Castling on black's queen's side	Click two squares to right of black king	Test if the castling move works correctly	King moves two space right and rook moves to the left of king	King moves two space right and rook moves to the left of king

Castling works correctly for both colours and on both sides.

Client Feedback

I showed [redacted] that he was now able to play the castling move. He was happy to see that he can play this move and tested it out. "Castling's super important, I play it within the first five moves," said [redacted]

Prototyping Pawn Promotion - 24/01/2023

My next step is to code pawn promotion. In chess, when a pawn reaches the end of the board, it can turn into a queen, rook, bishop or knight. I asked [redacted] if he had any suggestions on how to implement pawn promotion. He then said, "I only ever promote my pawns to queens, so I guess I would want the game to only do that." I asked [redacted] if he was okay with having pawns promoted only to queens, and he said he was happy with that because like [redacted], he only ever promotes to a queen. Based on this, I will make the pawns only promote to queen's.

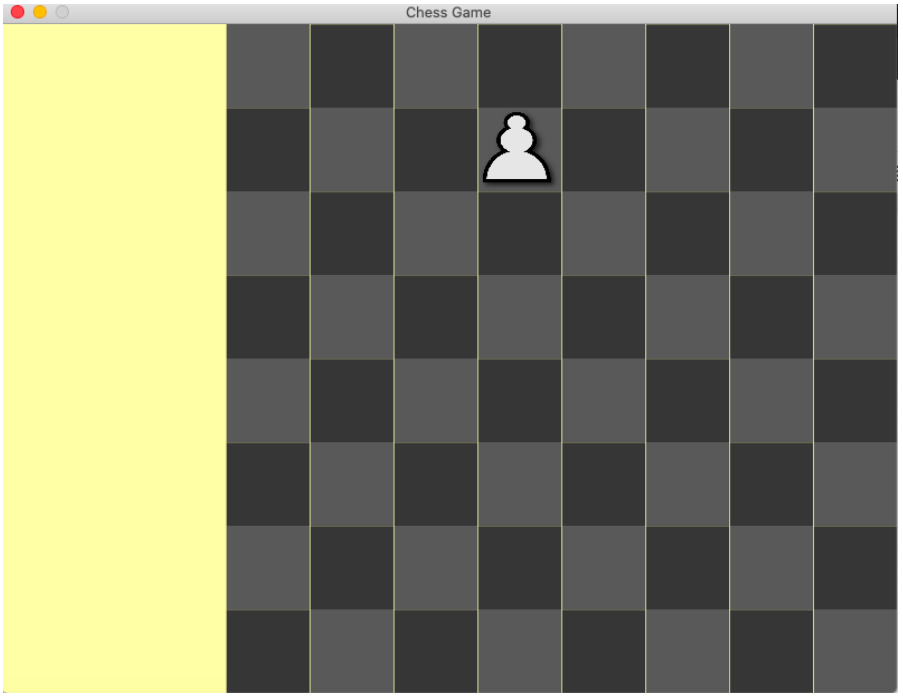
To implement pawn promotion, I once again amend the *movePiece()* function in the *Board* class.

```
# Pawn promotion
if (piece.getType() == "pawn") and ((piece.getPositionY() == 7) or (piece.getPositionY() == 0)):
    self.board[move.getNPos()[1]][move.getNPos()[0]] = Queen(piece.getColour(), piece.getPositionX(), piece.getPositionY())
```

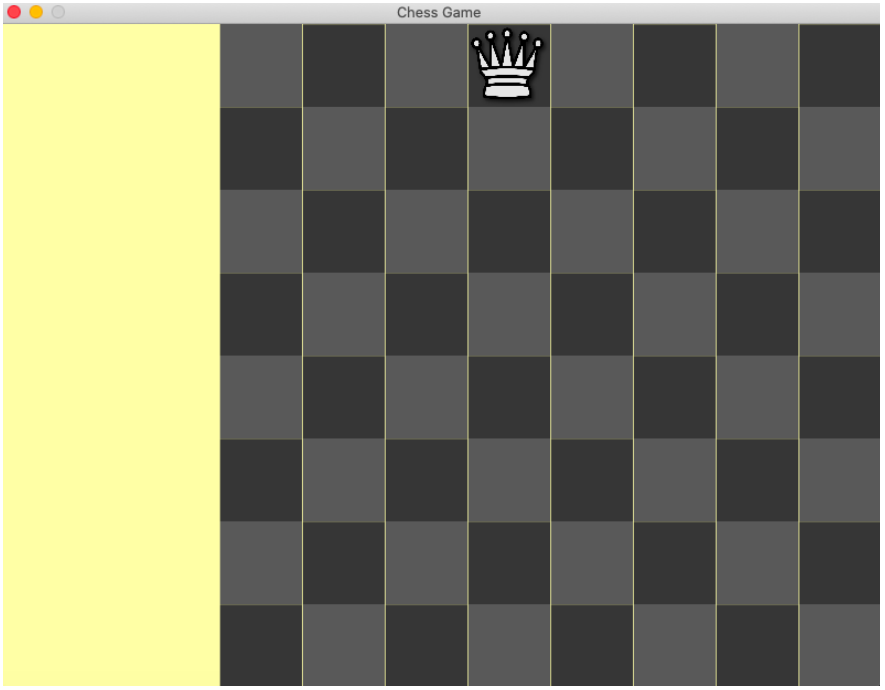
This conditional statement checks if the piece being moved is a pawn and whether it is at the end of the board (on the 0th or 7th row). If this is true, it creates a queen in that position the same colour of the pawn.

Testing Pawn Promotion (21)

The board is set with one pawn, on the second to last row.



The pawn is pushed up one square and should be promoted to a queen.



What's being tested	Inputs	Reasoning	Expected Outcome	Actual Outcome
Pawn promotion	Click moving pawn to the end of the board	Pawn must promote to queen to follow rules of chess and to fulfil client's request	Pawn turns into a queen after moving to the end row	Pawn turns into a queen after moving to the end row

Pawn promotion works. There is no need to test for black as colour of piece does not affect the algorithm, only position.

Client Feedback

[redacted] and [redacted] tested pawn promotion, and [redacted] said he was excited to see that all special moves were in the game.

In Check Flag - 01/02/2023

When a player is in check, then that means that an opposing piece is threatening to capture the player's king. According to the rules of chess, if a player is in check then they have to make a move getting them out of check. Hence, the first thing I'm going to do is create the isInCheck() function in Board which will act as a flag indicating if the player is in check.

```
def isInCheck(self):

    king_pos = None
    in_check = False
    moves = []

    # Find king position
    for i in range(0, 8):
        for j in range(0, 8):
            if self.board[i][j] != 0:
                if (self.board[i][j].getType() == "king") and (self.board[i][j].getColour() == self.turn):
                    king_pos = (j, i)
```

It initialises three variables:

- *king_pos*: The position of the current player's king, which is initially set to None.
- *in_check*: A boolean variable that tracks whether the king is in check, which is initially set to False.
- *moves*: An empty list that will be used to store the valid moves of opposing pieces.

It uses a nested for loop to search through the board and find the player's king. It sets the *king_pos* variable to the position of the king when it finds it.

```
# Flip turn to see opponents move
piece = None

for i in range(0, 8):
    for j in range(0, 8):
        if self.board[i][j]!= 0:
            piece = self.board[i][j]
            if piece.getColour() != self.turn:
                moves = piece.getValidMoves(self)
                # If endpoint for any move is the same position as king, then king is in check
                for move in moves:
                    if move.getNPos() == king_pos:
                        in_check = True

return in_check
```

It then uses another nested for loop to iterate through all the pieces on the board that belong to the opponent. For each piece, it:

- Retrieves a list of valid moves for that piece by calling the *getValidMoves()* method.
- Checks whether any of those moves end on the same square as the current player's king. If it finds such a move, it sets *in_check* to True.

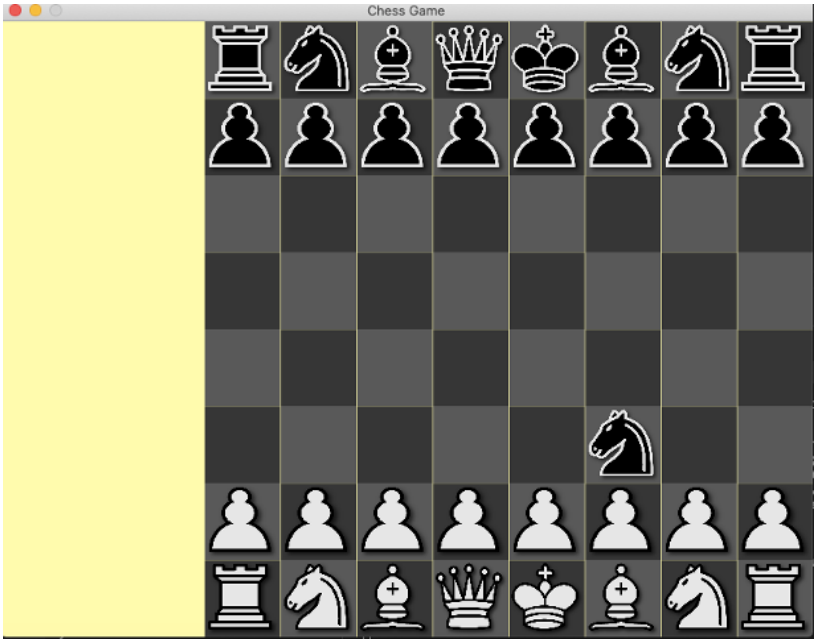
Finally it returns the *in_check*.

Testing The Check Flag (22)

By setting up the board in a position where it should be in check and printing the output of *isInCheck()*, we can check if the function works.

```
print(b.isInCheck())
```

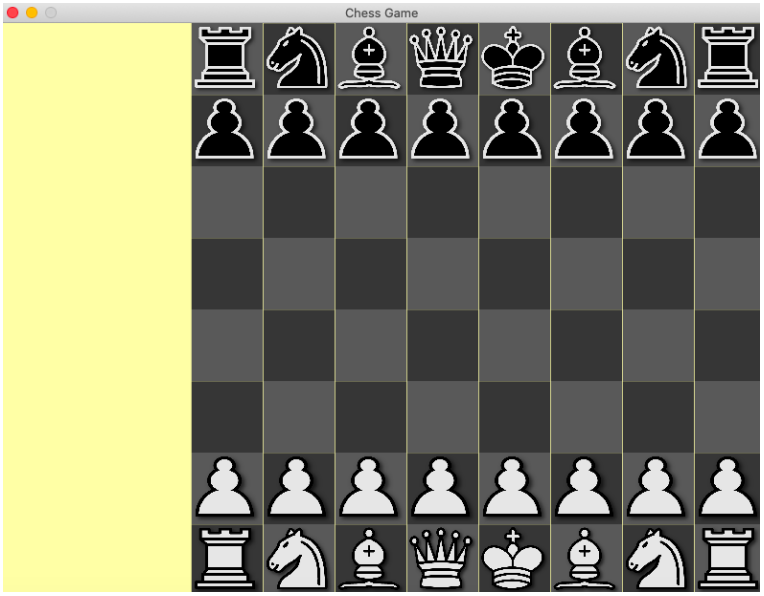
A position where the player is in check;



Output;

```
True
```

In a situation where the player is not in check;



Output;

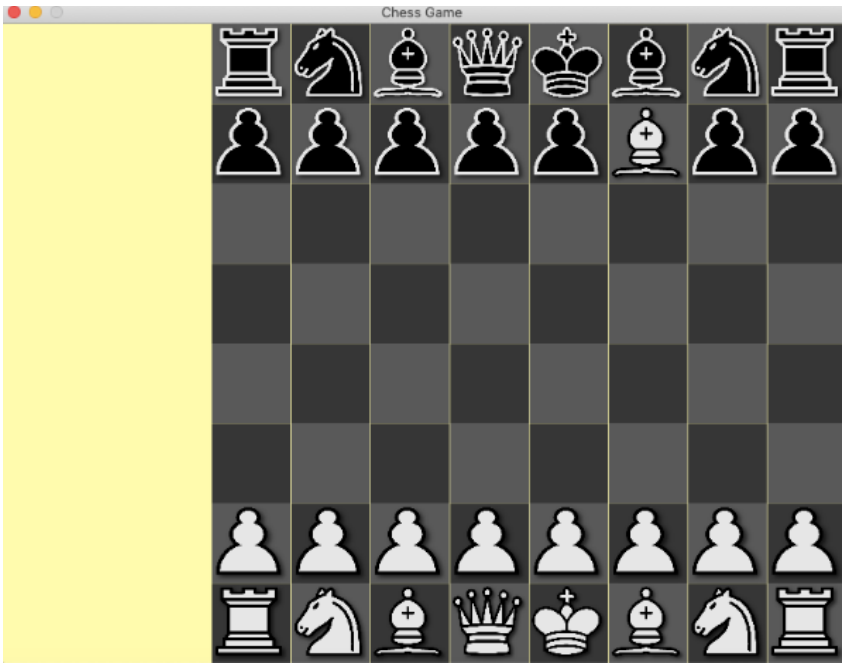
```
False
```

What's being tested	Inputs	Reasoning	Expected Outcome	Actual Outcome
In check flag	None	Program must be able to detect if board state is in check to perform later functions	Output true if the player is in check and output false if player is not in check	Output true if the player is in check and output false if player is not in check

This only tests if the in check function works with the player. To test if it will work with the computer, we can keep the *player_colour* as "white" and flip the *turn* to "black".

```
self.turn = 'black'
```

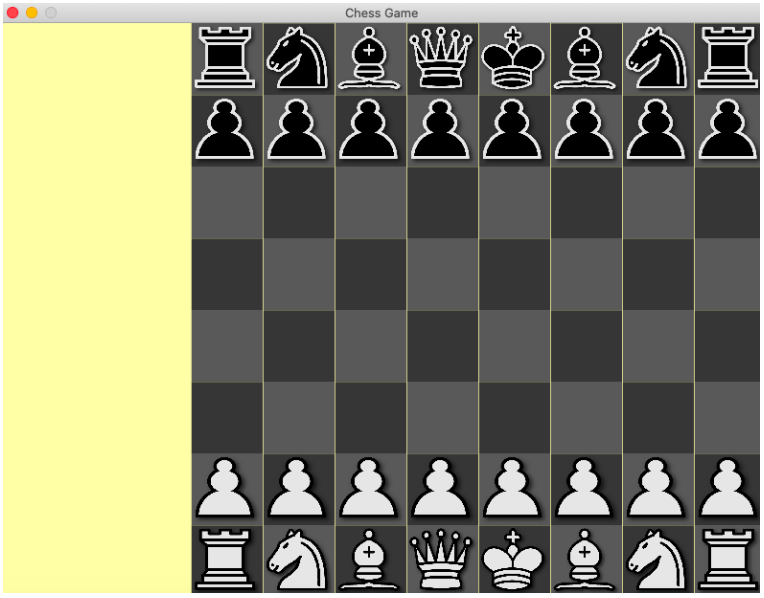
In a position where the computer is in check;



Output;

```
True
```

In a situation where the computer is not in check;



Output;

```
False
```

What's being tested	Inputs	Reasoning	Expected Outcome	Actual Outcome
In check flag for computer	None	Program must be able to detect if board state is in check to perform later functions	Output true if the computer is in check and output false if player is not in check	Output true if the computer is in check and output false if player is not in check

The flag works both for the player and the computer.

Filtering Moves and Prototyping Check Flag - 02/02/2023

Just because we get moves from the board, doesn't mean that they are valid as being in check will limit which moves are allowed. In *Board*, *getMoves()* only gets every single allowed move, but doesn't filter out moves leaving the position in check. This is why in *Board* I created the additional method *getValidMoves()* filters out those moves leaving the board state in check.

```
def getValidMoves(self):
    valid_moves = []
    moves = self.getMoves()

    for move in moves:

        # temporaily store positions
        piece_in_n_pos = self.board[move.getNPos()[1]][move.getNPos()[0]]
        piece_in_o_pos = self.board[move.getOPos()[1]][move.getOPos()[0]]

        # replace new position with new piece and old position with 0
        self.board[move.getNPos()[1]][move.getNPos()[0]] = piece_in_o_pos
        self.board[move.getOPos()[1]][move.getOPos()[0]] = 0

        # if board state is not in check add move to list
        if self.isInCheck() == False:
            valid_moves.append(move)

        # return board to original state
        self.board[move.getOPos()[1]][move.getOPos()[0]] = piece_in_o_pos
        self.board[move.getNPos()[1]][move.getNPos()[0]] = piece_in_n_pos

    self.moves = valid_moves

    return self.moves
```

The function starts by initialising an empty list called *valid_moves* to store all the valid moves of the piece. It then calls a function *getMoves()* to get all the possible moves.

The function then loops through each of these moves, temporarily stores the pieces that are in the start and end positions of the move, and then replaces the end position with the piece at the start position and sets the start position to be empty. This simulates making the move on the board.

The function then checks whether the board is now in check by calling *isInCheck()*. If the board is not in check, then the move is added to *valid_moves*.

The board is then returned to its original state by setting the start position back to its original piece and the end position to its original piece.

The function finally returns *valid_moves*, and the *moves* attribute of *Board* is set equal to *valid_moves*.

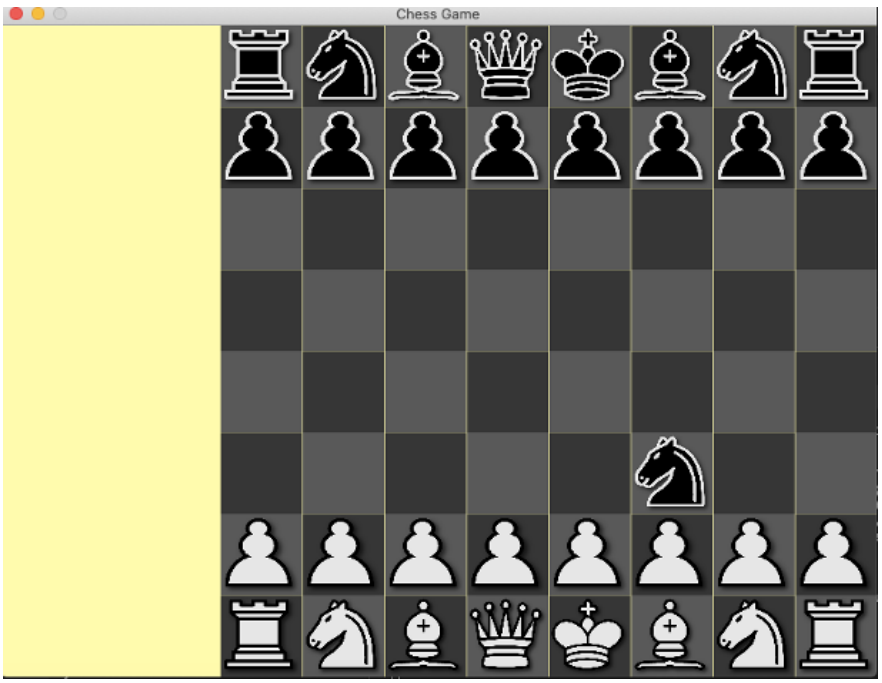
The *boardClicker()* method is amended to call *getValidMoves()* instead of *getMoves()*. This ensures that the *moves* list only contains moves that get the board state out of check.

```
def boardClicker(self, clickedX, clickedY):

    self.getValidMoves()
```

Testing Filtering Moves Based on being in Check (23)

There are three ways to get out of check; capturing the attacking piece, blocking the attack or moving your king out of sight of the attack.

A scenario is created where the board is in check and the only option is capturing the black knight.



There should only be three legal moves which can get the board out of check; either white pawn diagonal capturing the black knight or the white knight capturing the black knight.

The following script prints the number of legal moves at the start position of each move. This is used to check if the valid moves are taking into account whether the board is in check.

```
m = b.getValidMoves()
print(len(b.getValidMoves()))
for i in m:
    print(m.getOPos())
```

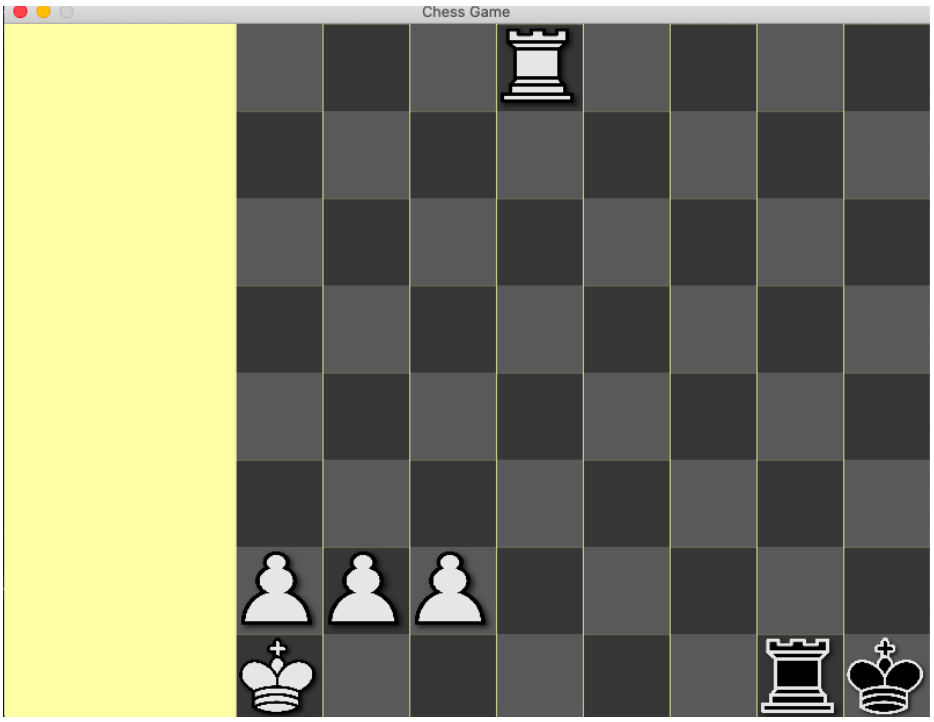
Output;

```
Hello from the pygame community. https://www.pygame.org/contribute.html
3
(4, 6)
(6, 6)
(6, 7)
```

What's being tested	Inputs	Reasoning	Expected Outcome	Actual Outcome
If program knows to capture a piece when in check	None	To test if the program recognises on of the three ways to get out of check	Three valid moves with the coordinates indicating the capture of a knight	Three valid moves with the coordinates indicating the capture of a knight

Capturing the piece to get out of check is successful.

Now a scenario is created where the only legal move is to block the attack with the white rook.



Output;

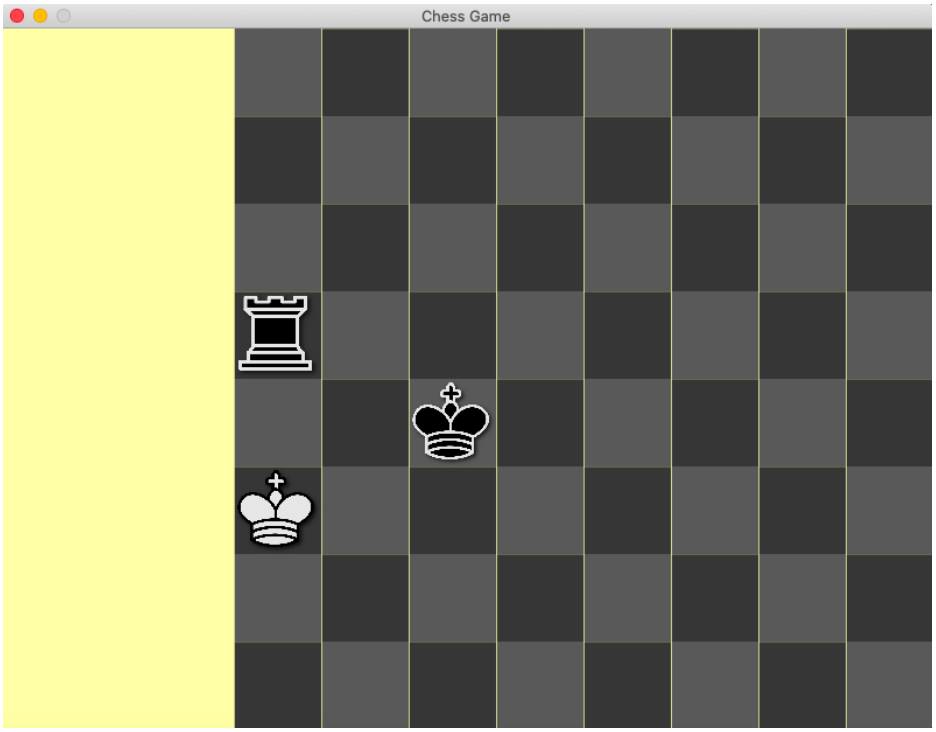
```
Hello from the pygame community. https://www.pygame.org/contribute.html
1
(3, 0)
```

What's being tested	Inputs	Reasoning	Expected Outcome	Actual Outcome
If program knows to block when in check	None	To test if the program recognises on of the three ways to get out of check	One valid moves with the coordinates indicating blocking with the rook	One valid moves with the coordinates indicating blocking with the rook

Blocking with a piece to get out of check is successful.

The final test is to see if the king can move out of the way when in check.

A scenario is created where the only option is to move the king out of the attack.



Output;

```
Hello from the pygame community. https://www.pygame.org/contribute.html
1
(0, 5)
```

What's being tested	Inputs	Reasoning	Expected Outcome	Actual Outcome
If program knows to move when in check	None	To test if the program recognises on of the three ways to get out of check	One valid moves with the coordinates indicating moving of the king	One valid moves with the coordinates indicating moving of the king

Moving the king out the way when in check is successful.

Therefore the program is able to identify all legal moves when the board state is in check.

Win Condition: Prototyping Checkmate Flag - 04/02/2023

A chess game can end in two ways; either one side checkmates the other causing there to be a winner or if checkmate is not possible leading to stalemate hence a draw.

Two conditions must be true if checkmate occurs; one of the sides must be in check, and the side in check must not have any legal moves.

The function *isCheckmate()* in *Board* will be a flag to indicate if the board state is checkmate.

```
def isCheckmate(self):
    self.getValidMoves()

    # If no legal moves
    if (len(self.moves) == 0):
        # and is in check
        if (self.isInCheck() == True):
            # then checkmate
            return True
    # Otherwise false
    else:
        return False
```

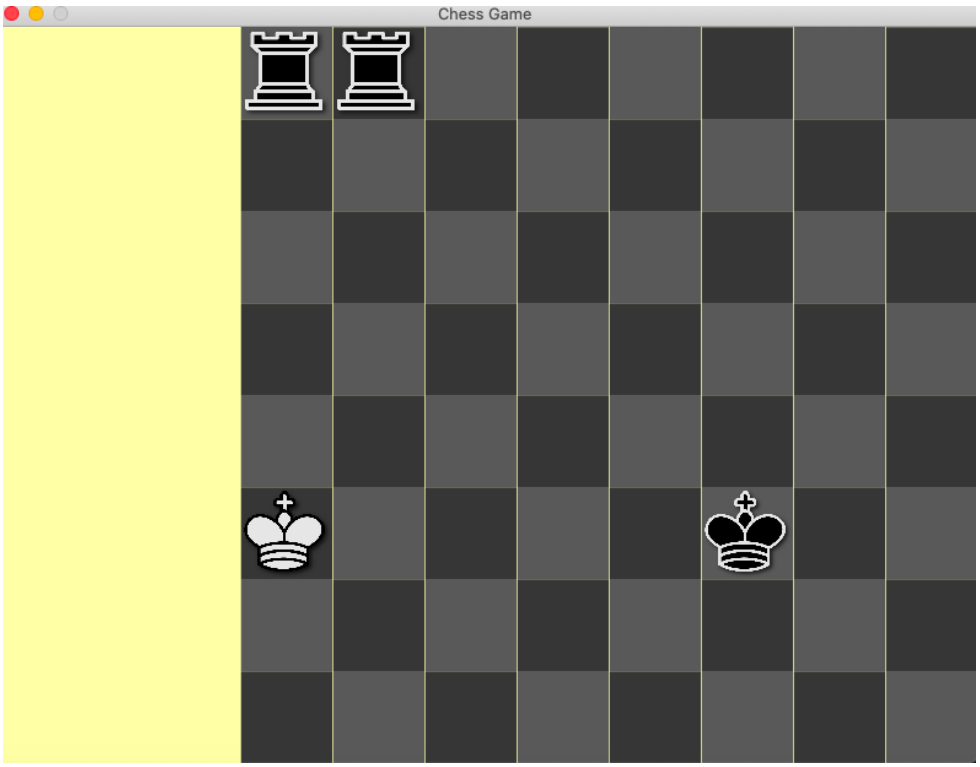
It finds the length of the *moves* attribute, and if it's 0 then there are no legal moves for that side. The *isInCheck()* function is called to see if the current board state is in check.

Testing Checkmate Flag (24)

The script below prints whether or not the state of the board is checkmate.

```
print(b.isCheckmate())
```

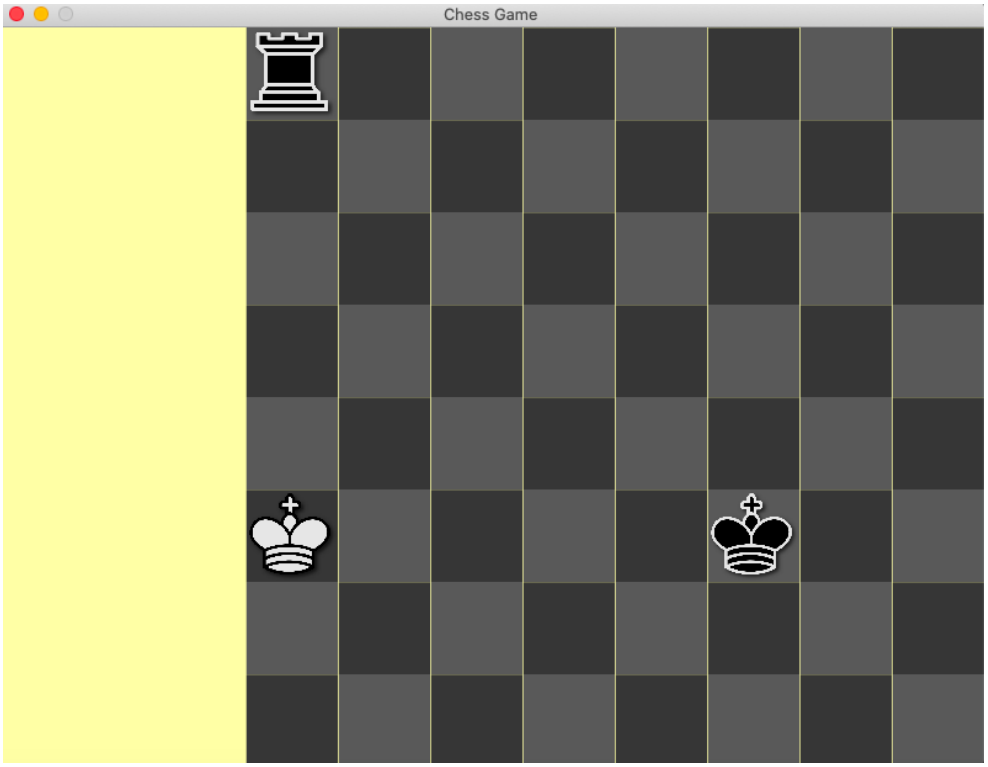
A scenario where the board state is checkmate is created.



Output;

```
True
```

Now for a scenario where the board state is not checkmate.



Output;

```
False
```

What's being tested	Inputs	Reasoning	Expected Outcome	Actual Outcome
Checkmate Flag	None	To detect whether the game ends by checkmate	Prints True if the board is in checkmate, otherwise prints False	Prints True if the board is in checkmate, otherwise prints False

Draw Condition: Prototyping Stalemate Flag - 04/02/2023

A draw in chess is called a stalemate. Stalemate can occur if one side has no legal moves and is not in check. It can also occur when there are not enough pieces on the board for either side to checkmate, which is called the insufficient material clause.

I did research on what combination of pieces can invoke the insufficient material clause and found the below;

- King vs king with no other pieces.
- King and bishop vs king.
- King and knight vs king.

- King and bishop vs king and bishop

  - www.chessstrategyonline.com

I will base my conditions for the insufficient material clause based on this research. Before I started coding the stalemate flag, I asked [redacted] if there were any other clauses that he could think of for the insufficient material clause. He said that a king and a minor piece vs a king and a minor piece can invoke the clause, so I will add that condition. (A minor piece being a bishop or a knight).

```
def isStalemate(self):
    self.getValidMoves()
    is_stalemate = False
    white_pieces = []
    black_pieces = []

    # Clauses for the insufficient material clause
    insufficient_material_scenerios = [
        (['king'], ['king']),
        (['king'], ['king', 'knight']),
        (['king'], ['bishop', 'king']),
        (['bishop', 'king'], ['king', 'knight']),
        (['king', 'bishop'], ['bishop', 'king']),
        (['king', 'knight'], ['king', 'knight'])
    ]
```

is_stalemate is set to False, and *white_pieces* and *black_pieces* are set as empty lists.

insufficient_material_scenarios is a list that contains various scenarios in which there are not enough pieces on the board to checkmate.

```
# If no moves are possible and not in check, then stalemate is true
if (len(self.moves) == 0):
    if (self.isInCheck() == False):
        is_stalemate = True
```

If there are no valid moves for the current player and the king is not in check, then *is_stalemate* is set to True.

```
# Get pieces on board
piece = None
for i in range(0, 8):
    for j in range(0, 8):
        piece = self.board[i][j]
        if piece != 0:
            if piece.getColour() == "white":
                white_pieces.append(piece.getType())
            else:
                black_pieces.append(piece.getType())
```

A nested loop iterates over each square on the board, and *white_pieces* and *black_pieces* are distributed with the types of pieces that are currently on the board for each colour.

```
temp_w = white_pieces.copy()
temp_w.sort()
white_pieces = temp_w
temp_b = black_pieces.copy()
temp_b.sort()
black_pieces = temp_b.copy()
```

The *white_pieces* and *black_pieces* lists are sorted in alphabetical order.

```
pieces = (white_pieces, black_pieces)

# See if current board state matches insufficient material clauses
for scenerio in insufficient_material_scenerios:
    if (pieces == scenerio) or (pieces == scenerio[::-1]):
        is_stalemate = True


return is_stalemate
```

pieces is a tuple of *white_pieces* and *black_pieces* is created. Each scenario in insufficient_material_scenarios is checked to see if it matches the current board state. If a match is found, then *is_stalemate* is set to True.

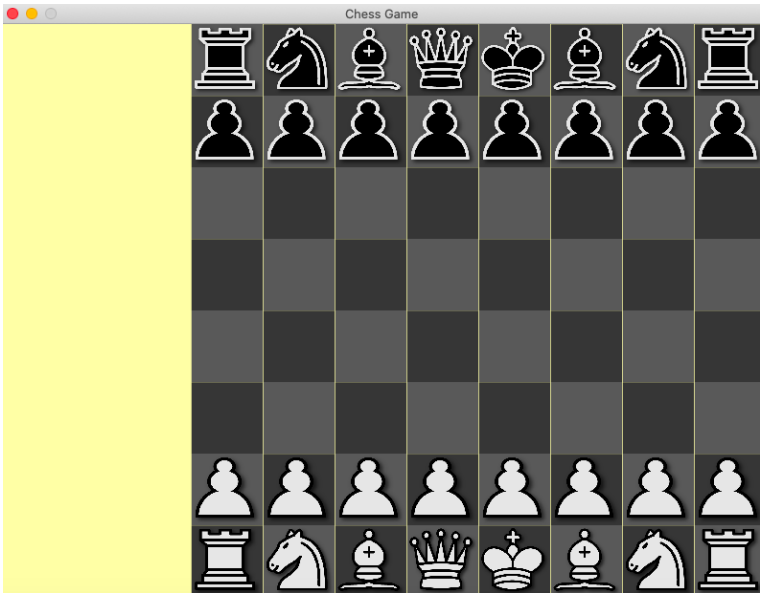
[::-1] is used to reverse the order of the tuples in the scenarios, in case the pieces are in a different order.

Finally, the value of *is_stalemate* is returned.

Testing Stalemate Flag (25)

Stalemate flag will return false if the board state is not stalemate.

```
print(b.isStalemate())
```

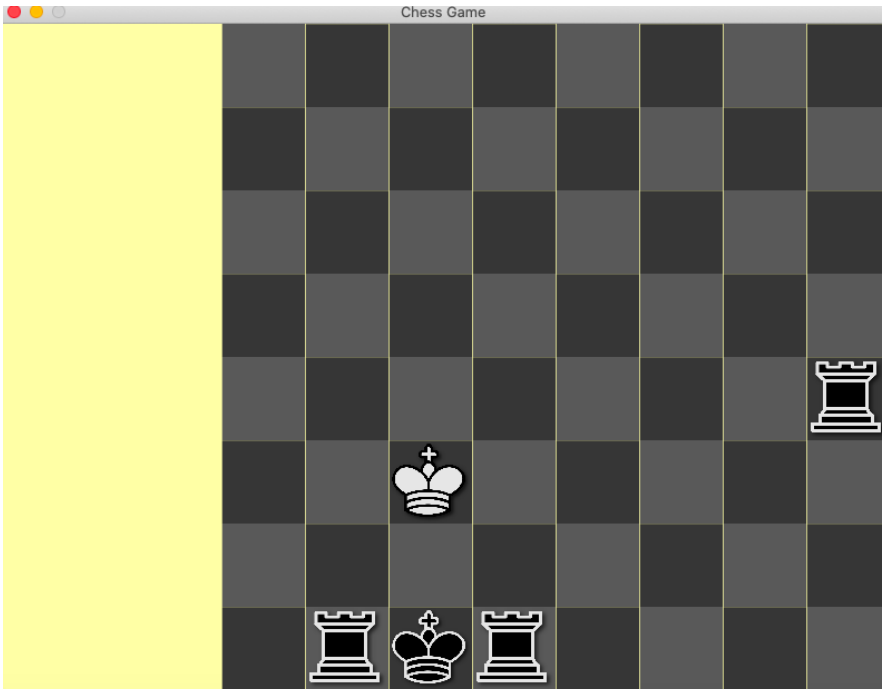


Output;

```
False
```

What's being tested	Inputs	Reasoning	Expected Outcome	Actual Outcome
Stalemate Flag	None	To detect whether the game ends by stalemate	Prints True if the board is in stalemate, otherwise prints False	Prints True if the board is in stalemate, otherwise prints False

Stalemate flag must be tested to see if it returns True if a side has no valid moves.

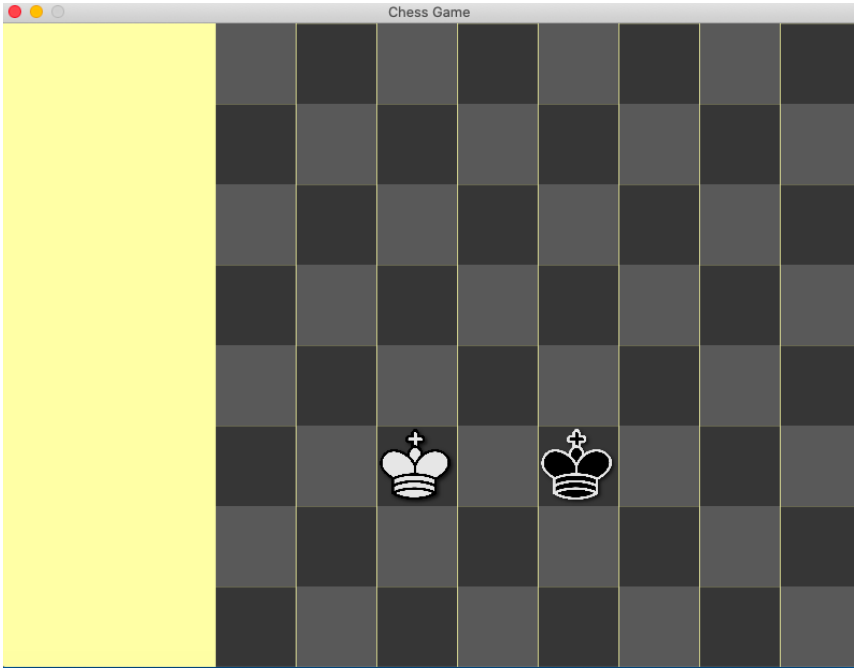


Output;

What's being tested	Inputs	Reasoning	Expected Outcome	Actual Outcome
Stalemate Flag	None	To detect whether the game ends by stalemate	Prints True if the board is in stalemate, otherwise prints False	Prints True if the board is in stalemate, otherwise prints False

Finally, the stalemate flag returns true if the insufficient material clause is met.

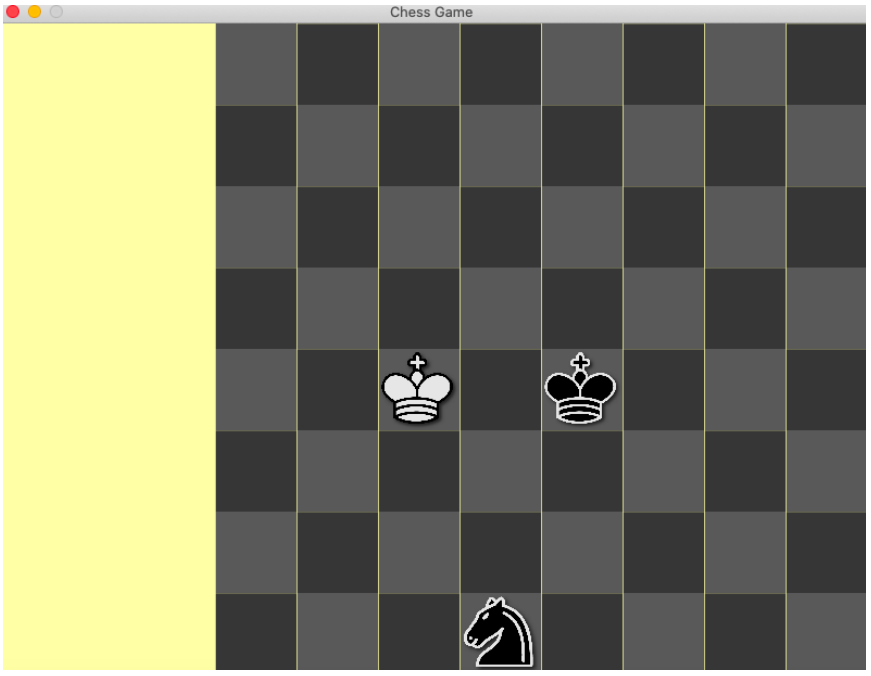
King vs King



Output;

```
True
```

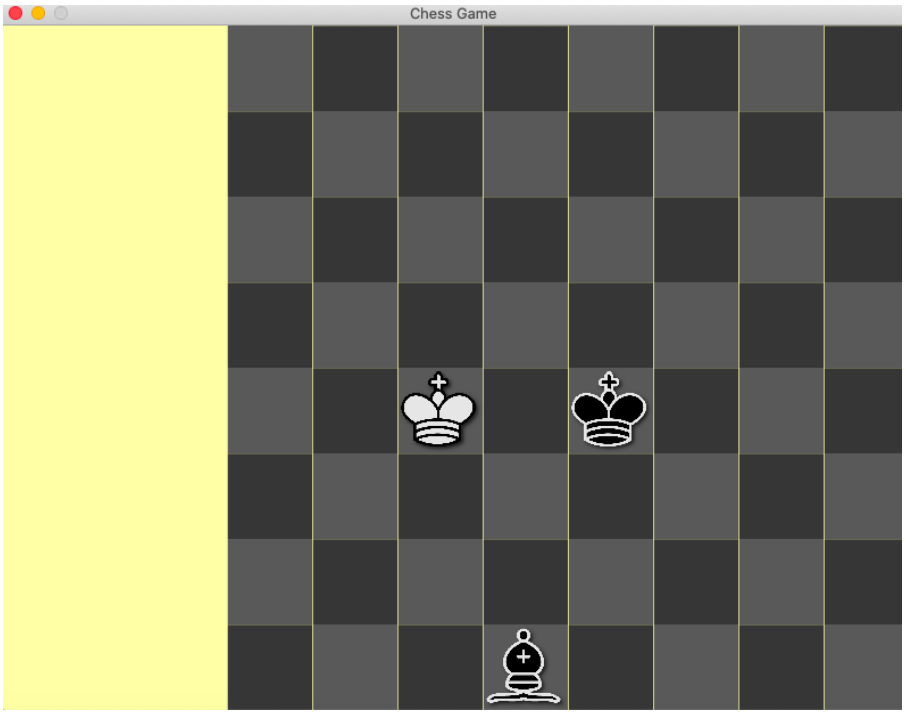
King vs King & Knight



Output;

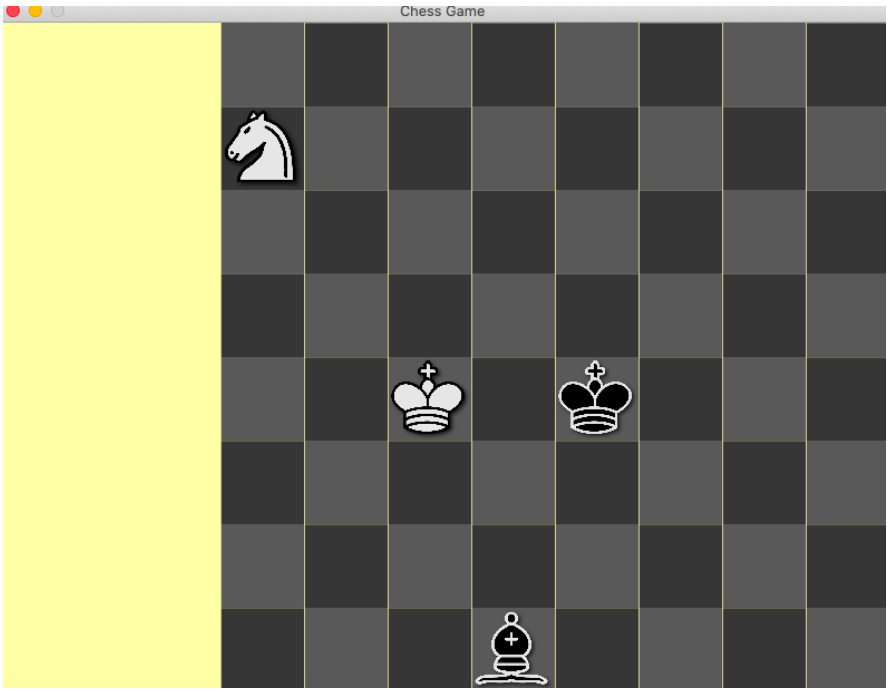
```
True
```

King vs King and Bishop



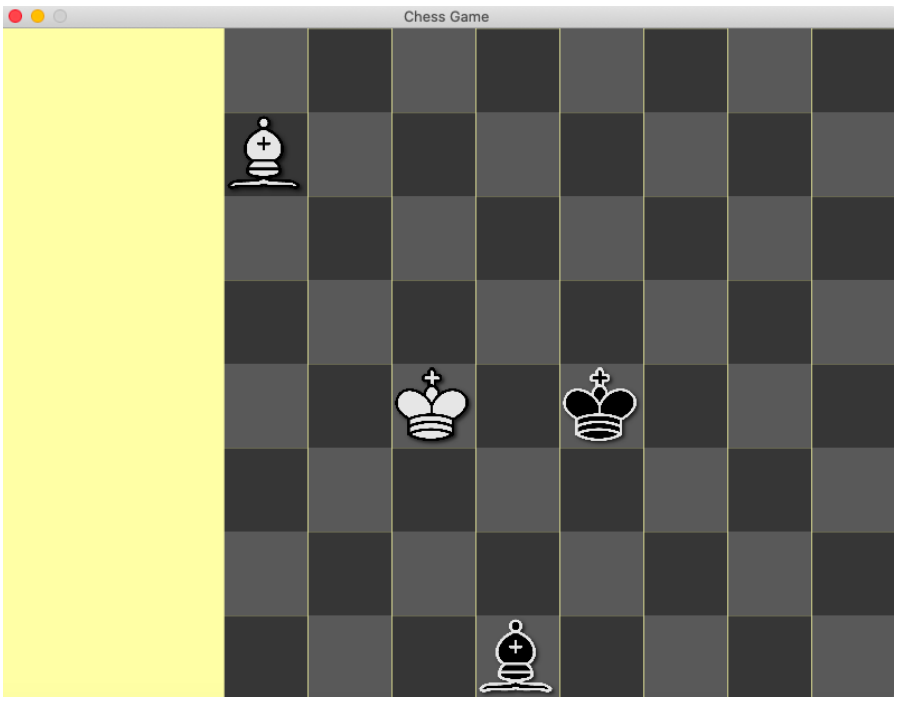
Output;

King & Bishop vs King & Knight



Output;

King & Bishop vs King & Bishop

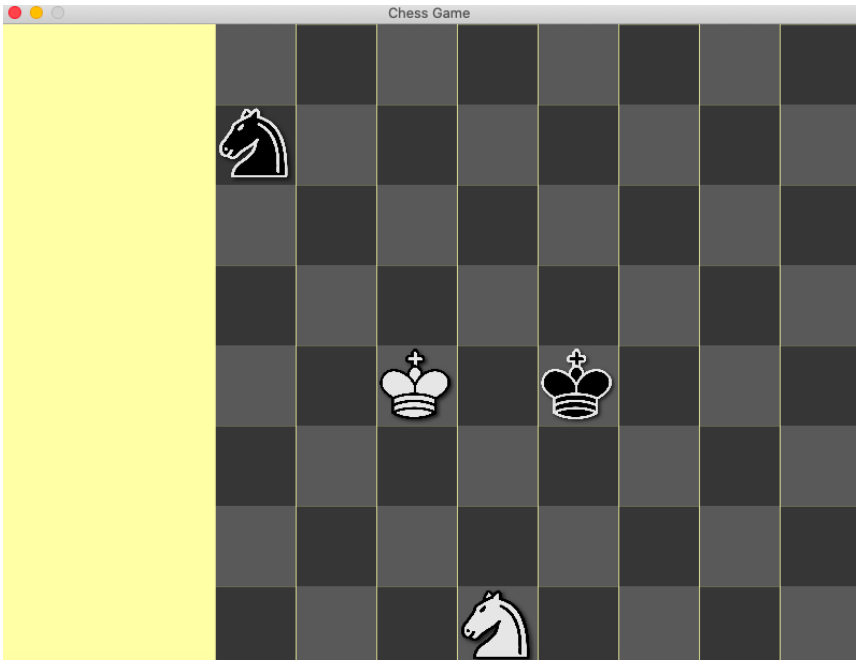


Output;

```
False
```

- This part of the test is a failure

King & Knight vs King & Knight



Output;

```
True
```

What's being tested	Inputs	Reasoning	Expected Outcome	Actual Outcome
Stalemate Flag	None	To detect whether the game ends by stalemate	Prints True for all clauses of insufficient material	Does not print True for all clauses of insufficient material

King & Bishop vs King & Bishop prints false, so this test is a failure

I looked at my code and realised that the tuple that checks for this scenario does not list the "bishop" and "king" in alphabetical order.

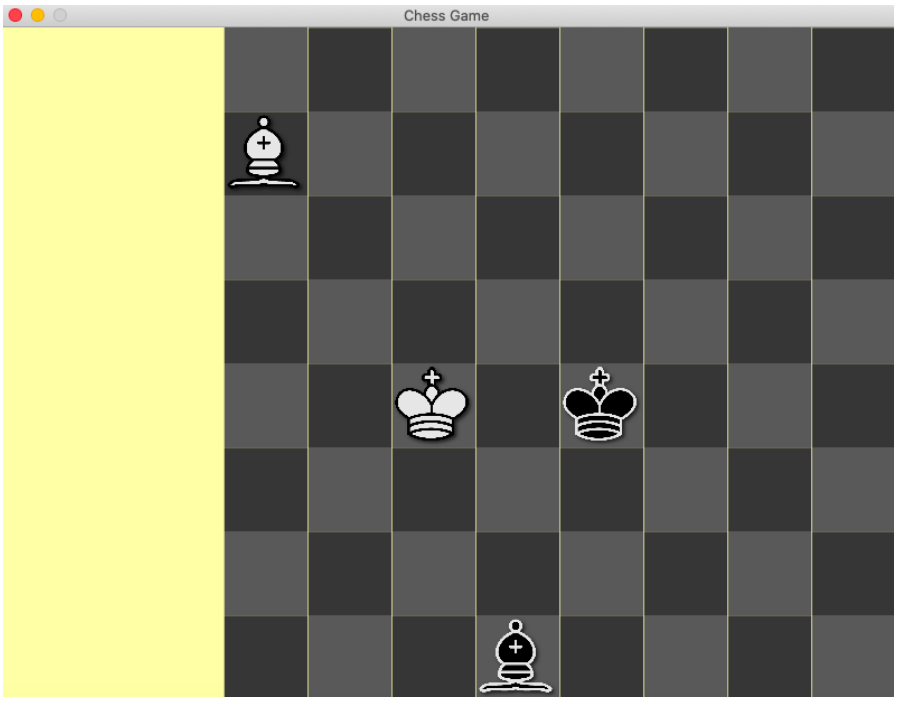
```
(['king', 'bishop'], ['bishop', 'king']),
```

It must be in alphabetical order to work as the list of pieces on the board will be in alphabetical order. (each list is different if the order of the elements are different).

This is an easy fix, all I have to do is switch the "king" and "bishop" around in the first list.

```
(['bishop', 'king'], ['bishop', 'king']),
```

Now when I test the scenario again;



Output;

```
True
```

What's being tested	Inputs	Reasoning	Expected Outcome	Actual Outcome
Stalemate Flag	None	To detect whether the game ends by stalemate	Prints True for all clauses of insufficient material	Prints True for all clauses of insufficient material

The stalemate flag works properly.

Prototyping a Game and Making Random Moves - 06/02/2023

In the AI,py, *random* and the *Board* class are imported.

```
import random
from board import Board
```

I created the AI class containing the attribute *depth*.

```
class AI ():

    def __init__(self, depth):
        self.depth = depth
```

The class contains the method *playRandomMove()*.

```
def playRandomMove(self, board):

    # Choose random move and return it

    moves = board.getValidMoves()

    move = random.choice(moves)

    return move
```

It gets the list of valid moves on the board, chooses a random move and returns the move.

A final function called *bestMove()* is made. If the *depth* is 0 then play a random move.

```
def bestMove(self, board):
    # If depth is 0, then play random move
    if self.depth == 0:
        return self.playRandomMove(board)
```

In main.py, the outcome of the game is printed depending on the board state.

```
if b.isStalemate():
    print("Stalemate")
    break

if b.isCheckmate():
    if b.getTurn() == "white":
        print("White Wins!")
    else:
        print("Black Wins!")
```

The *AI* class is imported into main, and a variable called *comp* is an instance of *AI* with a *depth* of 0.

```
comp = AI(0)
```

Finally, the computer makes a move if it's their turn.

```
move = comp.bestMove(b)

b.movePiece(move)
```

Now the computer should be able to play a full game of chess with random moves.

Testing Random Moves (26)

By running the code, you should be able to play a full game of chess.

The game was played until I won with white.

What's being tested	Inputs	Reasoning	Expected Outcome	Actual Outcome
Ability to play a full game of chess with computer	None	The game is to play chess against the computer	Full game with the computer can be player	Full game with the computer is be player

Client Feedback

Both [redacted] and [redacted] both played two full games of chess with both colours.

[redacted] said that "the platform runs kind of like lichess or chess.com, but doesn't have the same feel as a website which is good."

[redacted] said, "Wow, it's for real coming together."

The next step is to code the AI.

The Evaluation Function - 06/02/2023

The board should be able to calculate its value based on the position of the pieces. A table attribute for each piece is created as a 2D array, and reversed to calculate the other side of the board.

```
# King-square table
self.king_space_player = [
    [-3.0, -4.0, -4.0, -5.0, -5.0, -4.0, -4.0, -3.0],
    [-3.0, -4.0, -4.0, -5.0, -5.0, -4.0, -4.0, -3.0],
    [-3.0, -4.0, -4.0, -5.0, -5.0, -4.0, -4.0, -3.0],
    [-3.0, -4.0, -4.0, -5.0, -5.0, -4.0, -4.0, -3.0],
    [-2.0, -3.0, -3.0, -4.0, -4.0, -3.0, -3.0, -2.0],
    [-1.0, -2.0, -2.0, -2.0, -2.0, -2.0, -2.0, -1.0],
    [ 2.0,  2.0,  0.0,  0.0,  0.0,  0.0,  2.0,  2.0],
    [ 2.0,  3.0,  1.0,  0.0,  0.0,  1.0,  3.0,  2.0]
]

king_space_compt = self.king_space_player.copy()
king_space_compt.reverse()
self.king_space_compt = king_space_compt
```

```
  # Queen-square table
  self.queen_space_player = [
      [-2.0, -1.0, -1.0, -0.5, -0.5, -1.0, -1.0, -2.0],
      [-1.0,  0.0,  0.0,  0.0,  0.0,  0.0,  0.0, -1.0],
      [-1.0,  0.0,  0.5,  0.5,  0.5,  0.5,  0.0, -1.0],
      [-0.5,  0.0,  0.5,  0.5,  0.5,  0.5,  0.0, -0.5],
      [ 0.0,  0.0,  0.5,  0.5,  0.5,  0.5,  0.0, -0.5],
      [-1.0,  0.5,  0.5,  0.5,  0.5,  0.5,  0.0, -1.0],
      [-1.0,  0.0,  0.5,  0.0,  0.0,  0.0,  0.0, -1.0],
      [-2.0, -1.0, -1.0, -0.5, -0.5, -1.0, -1.0, -2.0]
  ]

  queen_space_compt = self.queen_space_player.copy()
  queen_space_compt.reverse()
  self.queen_space_compt = queen_space_compt
```

```
# Rook-square table
self.rook_space_player = [
    [ 0.0,  0.0,  0.0,  0.0,  0.0,  0.0,  0.0,  0.0],
    [ 0.5,  1.0,  1.0,  1.0,  1.0,  1.0,  1.0,  0.5],
    [-0.5,  0.0,  0.0,  0.0,  0.0,  0.0,  0.0, -0.5],
    [-0.5,  0.0,  0.0,  0.0,  0.0,  0.0,  0.0, -0.5],
    [-0.5,  0.0,  0.0,  0.0,  0.0,  0.0,  0.0, -0.5],
    [-0.5,  0.0,  0.0,  0.0,  0.0,  0.0,  0.0, -0.5],
    [-0.5,  0.0,  0.0,  0.0,  0.0,  0.0,  0.0, -0.5],
    [0.0,   0.0,  0.0,  0.5,  0.5,  0.0,  0.0,  0.0]
]

rook_space_compt = self.rook_space_player.copy()
rook_space_compt.reverse()
self.rook_space_compt = rook_space_compt
```

```
# Bishop-square table
self.bishop_space_player = [
    [-2.0, -1.0, -1.0, -1.0, -1.0, -1.0, -1.0, -2.0],
    [-1.0,  0.0,  0.0,  0.0,  0.0,  0.0,  0.0, -1.0],
    [-1.0,  0.0,  0.5,  1.0,  1.0,  0.5,  0.0, -1.0],
    [-1.0,  0.5,  0.5,  1.0,  1.0,  0.5,  0.5, -1.0],
    [-1.0,  0.0,  1.0,  1.0,  1.0,  1.0,  0.0, -1.0],
    [-1.0,  1.0,  1.0,  1.0,  1.0,  1.0,  1.0, -1.0],
    [-1.0,  0.5,  0.0,  0.0,  0.0,  0.0,  0.5, -1.0],
    [-2.0, -1.0, -1.0, -1.0, -1.0, -1.0, -1.0, -2.0]
]

bishop_space_compt = self.bishop_space_player.copy()
bishop_space_compt.reverse()
self.bishop_space_compt = bishop_space_compt
```

```
# Knight-square table
self.knight_space_player = [
    [-5.0, -4.0, -3.0, -3.0, -3.0, -3.0, -4.0, -5.0],
    [-4.0, -2.0,  0.0,  0.0,  0.0,  0.0, -2.0, -4.0],
    [-3.0,  0.0,  1.0,  1.5,  1.5,  1.0,  0.0, -3.0],
    [-3.0,  0.5,  1.5,  2.0,  2.0,  1.5,  0.5, -3.0],
    [-3.0,  0.0,  1.5,  2.0,  2.0,  1.5,  0.0, -3.0],
    [-3.0,  0.5,  1.0,  1.5,  1.5,  1.0,  0.5, -3.0],
    [-4.0, -2.0,  0.0,  0.5,  0.5,  0.0, -2.0, -4.0],
    [-5.0, -4.0, -3.0, -3.0, -3.0, -3.0, -4.0, -5.0]
]

knight_space_compt = self.knight_space_player.copy()
knight_space_compt.reverse()
self.knight_space_compt = knight_space_compt
```

```
# Pawn-square table
self.pawn_space_player = [
    [0.0,  0.0,  0.0,  0.0,  0.0,  0.0,  0.0,  0.0],
    [5.0,  5.0,  5.0,  5.0,  5.0,  5.0,  5.0,  5.0],
    [1.0,  1.0,  2.0,  3.0,  3.0,  2.0,  1.0,  1.0],
    [0.5,  0.5,  1.0,  2.5,  2.5,  1.0,  0.5,  0.5],
    [0.0,  0.0,  0.0,  2.0,  2.0,  0.0,  0.0,  0.0],
    [0.5, -0.5, -1.0,  0.0,  0.0, -1.0, -0.5,  0.5],
    [0.5,  1.0, 1.0,  -2.0, -2.0,  1.0,  1.0,  0.5],
    [0.0,  0.0,  0.0,  0.0,  0.0,  0.0,  0.0,  0.0]
]

pawn_space_compt = self.pawn_space_player.copy()
pawn_space_compt.reverse()
self.pawn_space_compt = pawn_space_compt
```

Based on the position and the value of pieces, the *evaluation()* of the board will be based upon whether it is advantageous for the player.

```
def evaluate(self):

    # Placeholder for each piece and evaluation starts at 0
    piece = None
    eval = 0

    for i in range(0, 8):
        for j in range(0, 8):

            if self.board[i][j] != 0:

                piece = self.board[i][j]

                if self.player_colour == "white":

                    # if playing as white, positive points for white pieces and positions on board
                    if piece.getColour() == "white":
                        if piece.getType() == "pawn":
                            eval += self.pawn_space_player[i][j]
                        elif piece.getType() == "bishop":
                            eval += self.bishop_space_player[i][j]
                        elif piece.getType() == "knight":
                            eval += self.knight_space_player[i][j]
                        elif piece.getType() == "rook":
                            eval += self.rook_space_player[i][j]
                        elif piece.getType() == "queen":
                            eval += self.queen_space_player[i][j]
                        elif piece.getType() == "king":
                            eval += self.king_space_player[i][j]

                        eval += piece.getValue()
```

```
                    # if playing as white, negative points for black pieces and positions on board, reverse board for calc
                    else:
                        if piece.getType() == "pawn":
                            eval -= self.pawn_space_compt[i][j]
                        elif piece.getType() == "bishop":
                            eval -= self.bishop_space_compt[i][j]
                        elif piece.getType() == "knight":
                            eval -= self.knight_space_compt[i][j]
                        elif piece.getType() == "rook":
                            eval -= self.rook_space_compt[i][j]
                        elif piece.getType() == "queen":
                            eval -= self.queen_space_compt[i][j]
                        elif piece.getType() == "king":
                            eval -= self.king_space_compt[i][j]

                        eval -= piece.getValue()

                else:
                    # if playing as black, negative points for white pieces and positions on board, reverse board for calc
                    if piece.getColour() == "white":
                        if piece.getType() == "pawn":
                            eval -= self.pawn_space_compt[i][j]
                        elif piece.getType() == "bishop":
                            eval -= self.bishop_space_compt[i][j]
                        elif piece.getType() == "knight":
                            eval -= self.knight_space_compt[i][j]
                        elif piece.getType() == "rook":
                            eval -= self.rook_space_compt[i][j]
                        elif piece.getType() == "queen":
                            eval -= self.queen_space_compt[i][j]
                        elif piece.getType() == "king":
                            eval -= self.king_space_compt[i][j]

                        eval -= piece.getValue()
```

```
            # if playing as white, negative points for black pieces and positions on board
            else:
                if piece.getType() == "pawn":
                    eval += self.pawn_space_player[i][j]
                elif piece.getType() == "bishop":
                    eval += self.bishop_space_player[i][j]
                elif piece.getType() == "knight":
                    eval += self.knight_space_player[i][j]
                elif piece.getType() == "rook":
                    eval += self.rook_space_player[i][j]
                elif piece.getType() == "queen":
                    eval += self.queen_space_player[i][j]
                elif piece.getType() == "king":
                    eval += self.king_space_player[i][j]

                eval += piece.getValue()

    return eval
```

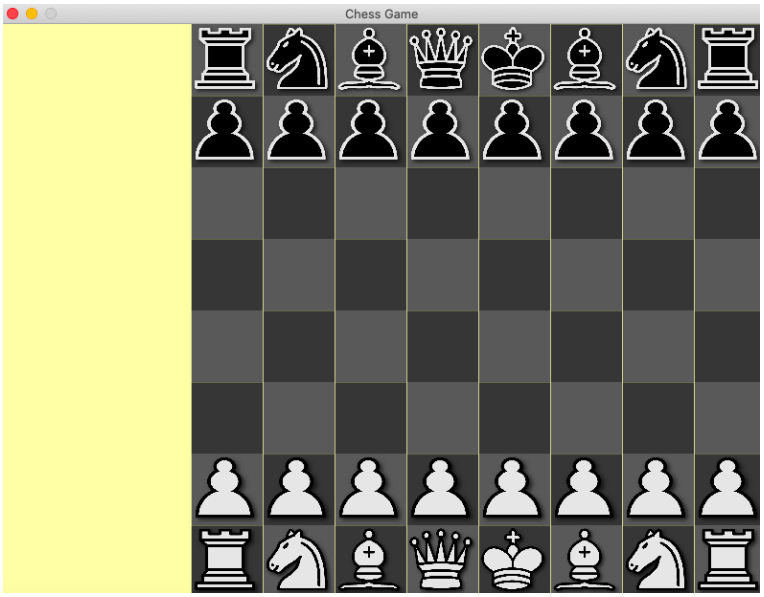
Initially, it sets the values of the variables *piece* and *eval* to None and 0, respectively. After that, it loops through the board, checking each square to see if a piece is there. If there is a piece, it determines whether it belongs to the player or the opponent, and adds or deducts points appropriately. The kind of piece and its placement on the board affect the number of points that are added or deducted.

eval is returned.

Testing Evaluation (27)

Printing the evaluation at the beginning of the game (where pieces are on the starting squares) should output a number around 0.

```
print(b.evaluate())
```

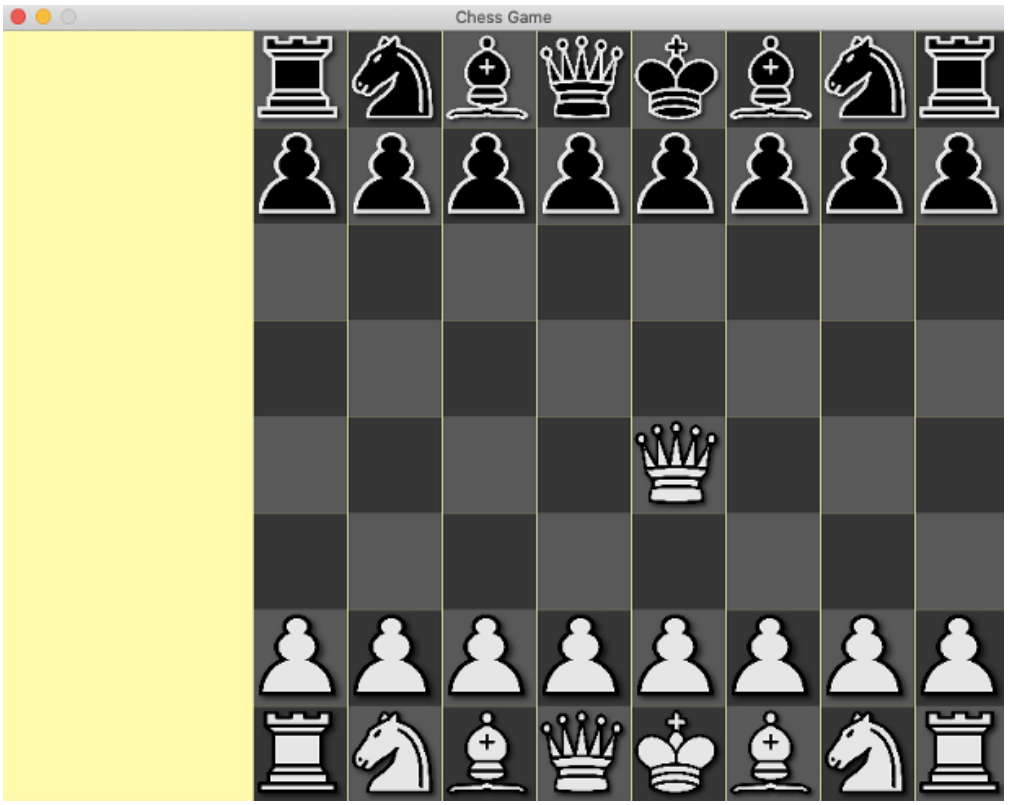


Output;

```
pygame 2.1.2 (SDL 2.0.18, Python 3.8.3)
Hello from the pygame community. https://www.pygame.org/con
tribute.html
5.243805389909539e-12
```

What's being tested	Inputs	Reasoning	Expected Outcome	Actual Outcome
Evaluation function	None	To calculate a number of points based on board state	A value around 0 should be printed	$5.24 * 10^{-12}$ is printed which is extremely close to 0, so test is a success

When a piece is added to the board of the player's colour, the evaluation should output a positive number.



Output;

```
Hello from the pygame community. https://www.pygame.org/con
tribute.html
90.50000000000524
```

What's being tested	Inputs	Reasoning	Expected Outcome	Actual Outcome
Evaluation function	None	To calculate a number of points based on board state	Prints a large positive value	Prints 90.5, which is a large positive number

The evaluation function works.

Prototyping Minimax Algorithm - 07/02/2023

A recursive minimax algorithm is used to decide on which move to play based on the number of points that can be gained by the computer, looking however many moves ahead.

The algorithm should make a move on a copy of the board to simulate making the move to evaluate the position. So the method *clone()* is written in *Board*, which makes a copy of the board.

```
def clone(self):
    clone = self.copy()
    return clone
```

In the *AI* class, the *minimax()* function is made.

```
def minimax(self, depth, board, alpha=-100000000000, beta=100000000000):

    # Break is depth is 0 or game has ended
    if depth == 0 or board.isCheckmate() or board.isStalemate():
        return (board.evaluate(), None)


    moves = board.getValidMoves()
```

The *minimax()* algorithm takes four parameters; *depth*, *board*, *alpha*, and *beta*. *depth* is the maximum depth of the search tree, *board* is the current game state, and *alpha* and *beta* are the minimum and maximum scores that the minimax algorithm has found so far.

The function first checks if the depth is 0 or if the game has ended in checkmate or stalemate. If so, it returns a tuple with the evaluation of *board* and None. *moves* are the valid moves on the *board* at that time.

The *bestMove()* method amends.

```
def bestMove(self, board):
    # If depth is 0, then play random move
    if self.depth == 0:
        return self.playRandomMove(board)

    # Otherwise play minimax with depth
    else:
        depth = self.depth
        best_move = self.minimax(depth, board)[1]

    return best_move
```

```
if board.turn != board.player_colour:
    best_move = None
    maxScore = -100000000000

    # Look at every move
    for move in moves:

        b = board.clone()

        # Move a piece and call functioin again
        b.movePiece(move)

        score = self.minimax(depth - 1, b, alpha, beta)


        # Calculate alpha
        alpha = max(alpha, score[0])

        # Break if minimising is greater than maximising
        if beta <= alpha:
            break

        if score[0] >= maxScore:
            maxScore = score[0]
            best_move = move

    # Return best move for maximiser
    return (maxScore, best_move)
```

It checks if it's the current player's turn to move by comparing *turn* with *player_colour*.

If it's not the player's turn, the function tries to find the move that maximises the score by iterating over each move and calling *minimax()* recursively by reducing *depth* by 1.

The score is stored in *score*, and *alpha* becomes the maximum of alpha and the first element of *score*.

If *beta* is less than or equal to *alpha*, it means that the best possible score that can be obtained by the minimizer is less than or equal to the best possible score that can be obtained by the maximizer, so the function can stop searching and return the previous. Otherwise, the function continues to search for the move that maximises the *score* and updates *maxScore* and *best_move* accordingly.

```
else:
    best_move = None
    minScore = 100000000000

    for move in moves:

        b = board.clone()

        # Move a piece and call functioin again
        b.movePiece(move)

        score = self.minimax(depth - 1, b, alpha, beta)

        # Calculate beta
        beta = min(beta, score[0])

        if score[0] <= minScore:
            minScore = score[0]
            best_move = move

        # Break if minimising is greater than maximising
        if beta <= alpha:
            print(beta, alpha)
            break

    # Return best move for minimiser
    return (minScore, best_move)
```

It checks if it's the current player's turn to move by comparing *turn* with *player_colour*.

If it is the player's turn, the function tries to find the move that maximises the score by iterating over each move and calling *minimax()* recursively by reducing *depth* by 1.

The score is stored in *score*, and *beta* becomes the minimum of *beta* and the first element of *score*.

If *beta* is less than or equal to *alpha*, it means that the best possible score that can be obtained by the minimizer is less than or equal to the best possible score that can be obtained by the maximizer, so the function can stop searching and return the previous. Otherwise, the function continues to search for the move that minimises the *score* and updates *minScore* and *best_move* accordingly.

The function returns the tuple with the best score and the corresponding best move for the player.

The function *bestMove()* is amended so that if the *depth* attribute is not 0, then the *minimax()* function is called with that depth.

```
def bestMove(self, board):
    # If depth is 0, then play random move
    if self.depth == 0:
        return self.playRandomMove(board)

    # Otherwise play minimax with depth
    else:
        depth = self.depth
        best_move = self.minimax(depth, board)[1]

    return best_move
```

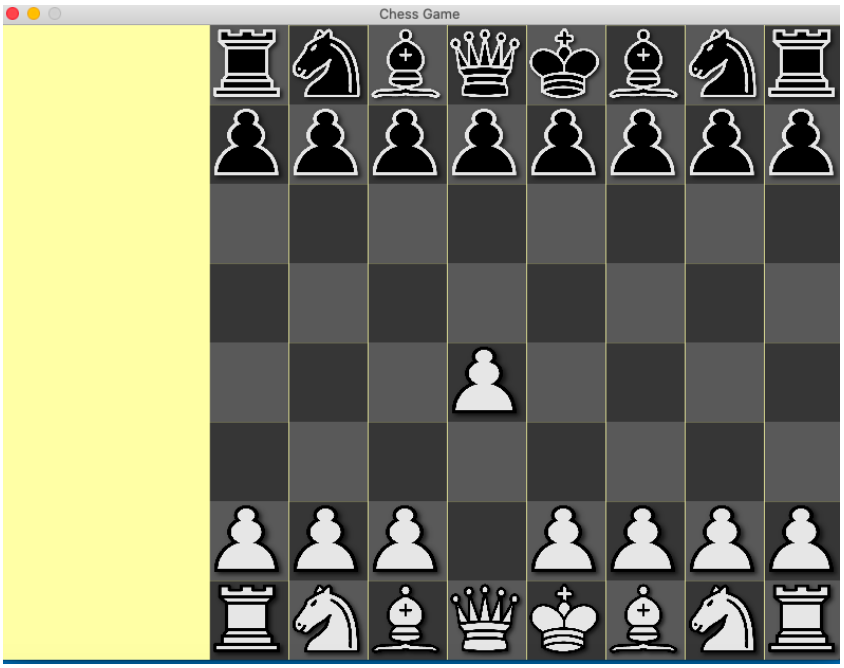
The *best_move* is the second element of the returned tuple.

Testing Minimax Algorithm (28)

comp is set to an *AI* of depth level 2.

The computer should now look ahead 2 moves to determine the best move.

When I play a move;



I got the following error;

```
Hello from the pygame community. https://www.pygame.org/contribute.html
TypeError: can't pickle pygame.Surface
```

What's being tested	Inputs	Reasoning	Expected Outcome	Actual Outcome
Minimax algorithm	None	For the computer to determine the best move	Computer to make a logical move in response to my move	TypeError: can't pickle pygame.Surface

This test was a failure.

I was confused and at first thought it had to be an issue with the *copy()* function as it came from a library that I am unfamiliar with. I then found the python documentation online;

> The difference between shallow and deep copying is only relevant for compound objects (objects that contain other objects, like lists or class instances):
>
> - A *shallow copy* constructs a new compound object and then (to the extent possible) inserts *references* into it to the objects found in the original.
> - A *deep copy* constructs a new compound object and then, recursively, inserts *copies* into it of the objects found in the original.
>
> - docs.python.org

I decided it made sense for the new board to be an independent copy of the previous board, so I had to make a deep copy rather than a shallow copy.

```
def clone(self):
    clone = self.deepcopy()
    return clone
```

I replaced the *copy()* function with the *deepcopy()* function and tried running the code again.

I got the same error;

```
Hello from the pygame community. https://www.pygame.org/contribute.html
TypeError: can't pickle pygame.Surface
```

What's being tested	Inputs	Reasoning	Expected Outcome	Actual Outcome
clone() function and minimax algorithm	None	For the computer to determine the best move by making independent copies of the board	Computer to make a logical move in response to my move	TypeError: can't pickle pygame.Surface

It appears the issue is with copying the board all together. I searched for the error online and found this;

pickle saving pygame Surface (python)

Asked 9 years, 6 months ago Modified 8 years, 9 months ago Viewed 6k times

You don't want to pickle a surface.

- stackoverflow.com

Because the *Board* class has a pygame surface within it, an instance of *Board* is unable to be copied.

I tried to get around this by using the '=' operator to make another board within the board via the *clone()* function.

```
def clone(self):
    clone = self
    return clone
```

I got the same error again;

```
Hello from the pygame community. https://www.pygame.org/contribute.html
TypeError: can't pickle pygame.Surface
```


What's being tested	Inputs	Reasoning	Expected Outcome	Actual Outcome
clone() function and minimax algorithm	None	For the computer to determine the best move by making independent copies of the board	Computer to make a logical move in response to my move	TypeError: can't pickle pygame.Surface

Apparently the '=' operator in python indirectly calls some form of a copy function.

I got fed up so I gave up on the problem, and decided to come back to it with a clear head.

Prototyping Minimax Algorithm Again - 18/02/2023

A week and a half later, I woke up with an epiphany. I don't need to make a copy of the board if the board can revert back to its previous state. If the board can do this, I can play all the possible moves on the original board and therefore never have to make a copy of the board.

I started by creating a method which can save the current board state.

```
self.previous_board = []
```

The *previos_board* attribute saves all the previous board states in an array.

```
def saveBoard(self):
    # Create blank board
    saved = [[0 for i in range(0, 8)] for j in range(0, 8)]

    piece = None

    # PLace pieces of current board onto saved board
    for i in range(0, 8):
        for j in range(0, 8):
            if self.board[i][j] != 0:
                piece = self.board[i][j]
                saved[i][j] = piece

    # Add saved board to list
    self.previous_board.append(saved)
```

I added the parameter *other_board* to the *setupBoard()* method. This is to reconfigure the board within the minimax algorithm to check different scenarios.

```
def setupBoard(self, other_board):

    square = None
    for i in range(0, 8):
        for j in range(0, 8):

            # Choose board template
            if other_board is None:
                if self.player_colour == "white":
                    square = self.player_white_start_pos[i][j]
                elif self.player_colour == "black":
                    square = self.player_black_start_pos[i][j]

            else:
                square =  other_board[i][j]
```

I realised that my *setupBoard()* method requires an ascii board to be set into it, so I made the *ascii()* method to convert the current board into ascii characters.

```
def ascii(self, board):

    # Decrypt board to ascii characters to
    ascii = [['.' for i in range(0, 8)] for j in range(0, 8)]
    piece = None
```

```
for i in range(0, 8):
    for j in range(0, 8):
        piece = board[i][j]
        if piece!= 0:
            if piece.getType() == "pawn":
                if piece.getColour() == "white":
                    ascii[i][j] = "P"
                else:
                    ascii[i][j] = "p"
            if piece.getType() == "knight":
                if piece.getColour() == "white":
                    ascii[i][j] = "N"
                else:
                    ascii[i][j] = "n"
            if piece.getType() == "bishop":
                if piece.getColour() == "white":
                    ascii[i][j] = "B"
                else:
                    ascii[i][j] = "b"
```

```
            if piece.getType() == "rook":
                if piece.getColour() == "white":
                    ascii[i][j] = "R"
                else:
                    ascii[i][j] = "r"
            if piece.getType() == "queen":
                if piece.getColour() == "white":
                    ascii[i][j] = "Q"
                else:
                    ascii[i][j] = "q"
            if piece.getType() == "king":
                if piece.getColour() == "white":
                    ascii[i][j] = "K"
                else:
                    ascii[i][j] = "k"
    return ascii
```

Now I can make the *unmakeMove()* function.

```
def unmakeMove(self):
    # Replace board with previous board and flip turn back
    board = self.previous_board[-1]
    board = self.ascii(board)
    self.setupBoard(board)
    self.previous_board.pop()
    self.turn = "white" if self.turn == "black" else "black"
```

It accesses the last element of *previous_board*, sets the board up in the layout of the previous board, removes that board from the list, and flips the turn.

Now I can rewrite the *minimax()* function to undo moves instead of cloning the board.

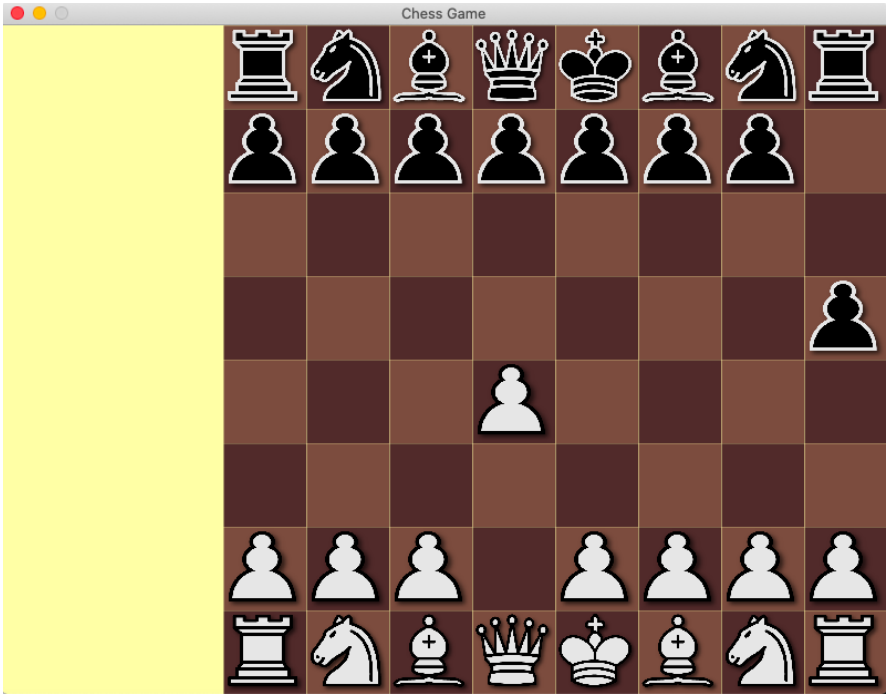
```
board.saveBoard()

# Move a piece and call functioin again
board.movePiece(move)

score = self.minimax(depth - 1, board, alpha, beta)

board.unmakeMove()
```

The board is saved and the piece is moved. When unstacked, the move will be undone.

Testing Minimax Algorithm Again (29)

The algorithm is tested again.

When I play a move;



The computer responds with a move!

However, continuing on with the game, it appears that the computer doesn't make good moves;

It is white's turn. In less than ten moves the computer already would've lost a queen for nothing (as it can be captured by the bishop without punishment).

What's being tested	Inputs	Reasoning	Expected Outcome	Actual Outcome
Minimax algorithm	None	For the computer to determine the best move	Computer to make a logical move in response to my move	Computer doesn't necessarily play logical moves

I looked back at the algorithm to see if there was a particular reason the computer seemed to make illogical moves. I realised that my evaluation function returned a positive value in favour of the player, however the minimax algorithm searches for the best move for the computer (so positive in favour of the computer). To fix this, I have to return a negative evaluation in *minimax()* otherwise the computer will search for the "worst" possible move, not the "best" move.

```
return (-board.evaluate(), None)
```

I played it again;

Several moves into this game, I can tell that the computer seems "smarter" as it is practising good chess principles such as using pawns to take up space in the centre and developing the knight to an "active" square (where it can attack a pawn).

What's being tested	Inputs	Reasoning	Expected Outcome	Actual Outcome
Minimax algorithm	None	For the computer to determine the best move	Computer to make a logical move in response to my move	Computer to make a logical move in response to my move

The minimax algorithm works, meaning that I have created a chess AI!

Client Feedback

[redacted] and [redacted] played numerous games of different depth levels from 1 to 5.

[redacted] said; “I think from (depth level) 3 onwards it gets really difficult. It doesn’t make any blunders that I can see. I think I can warm up with the first couple levels to make my gameplay better, then when I feel ready, I can attempt the harder ones. It’s super good, well done!”

[redacted] said; “Wow, it’s challenging and fun! I find it real hard to beat and depending on the computer level I can practise different openings. The only thing is I had to wait eight to nine minutes to see 5 moves ahead. Other than that, great!”

The timing issue which [redacted] mentioned is not something I can fix with the given timeline and hardware requirements. I explained this to him and he said that it was OK and he was content with the game as is because it accomplishes its purpose.

Prototyping Buttons For Board And Player Colour - 21/02/2023

The final step is to create a User Interface for the game. Buttons to change the player colour and board colour will be on the left side of the screen. They will be rectangles of dimensions 200 x 75 pixels, and coloured based on its associating functionality.

Using the *draw.rect()* function in pygame, I started by drawing the rectangles to *screen*,

```
# Draw rectangles for buttons
pygame.draw.rect(screen, (255, 255, 255), pygame.Rect(0, 0, 200, 75))
pygame.draw.rect(screen, (0, 0, 0), pygame.Rect(0, 75, 200, 75))
pygame.draw.rect(screen, (150, 75, 0), pygame.Rect(0, 150, 200, 75))
pygame.draw.rect(screen, (125, 125, 125), pygame.Rect(0, 225, 200, 75))
```

Within the *MOUSEBUTTONDOWN* event handler, I wrote a condition to say to only call *boardClicker()* when the board is clicked, which is when *clickedX* is greater than 200.

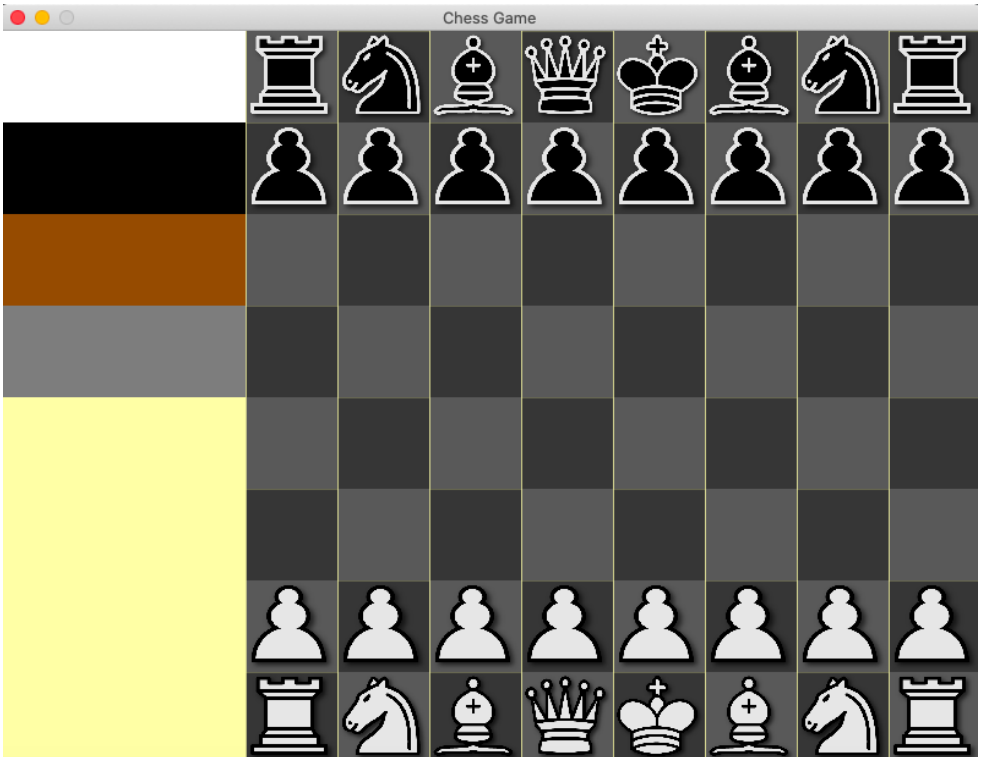
```
if (event.type == pygame.MOUSEBUTTONDOWN):

    if clickedX > 200:


        b.boardClicker(clickedX, clickedY)
```

Still within the *MOUSEBUTTONDOWN* event handler, conditions are added based on *clickedY* to determine which button has been pressed. It is based on the position and the height (which is 75 pixels) of the buttons.

```
elif clickedY < 75:
    # Change Player Colour to white
    boardColour  = b.getBoardColour()
    b = Board(boardColour, "white")
    b.setupBoard(None)

elif clickedY < 150:
    # Change Player Colour to black
    boardColour = b.getBoardColour()
    b = Board(boardColour, "black")
    b.setupBoard(None)

elif clickedY < 225:
    # Change Board Colour to brown
    playerColour = b.getPlayerColour()
    b = Board("brown", playerColour)
    b.setupBoard(None)

elif clickedY < 300:
    # Change Board Colour to gray
    playerColour = b.getPlayerColour()
    b = Board("gray", playerColour)
    b.setupBoard(None)
```

Testing Colour Buttons (30)

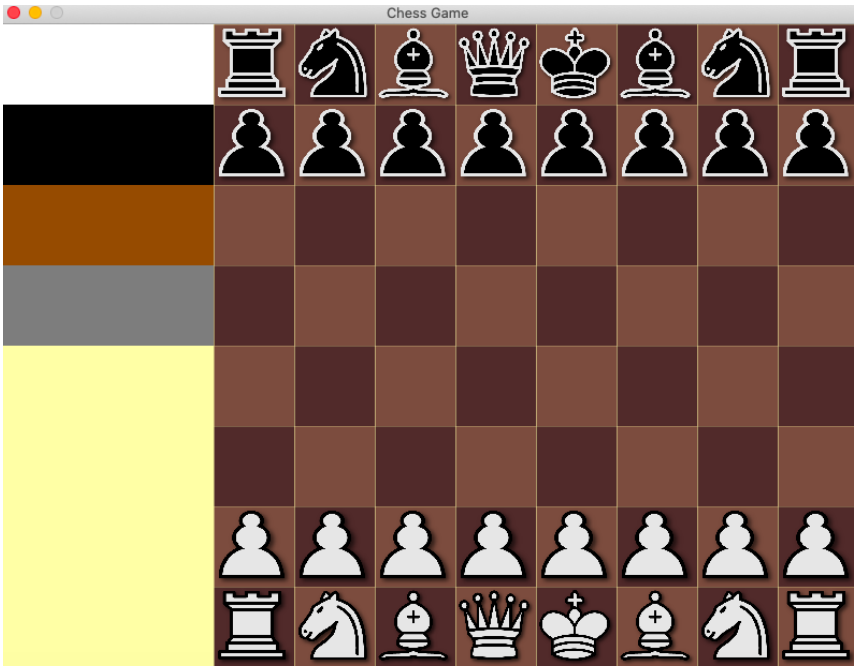
First we'll test if the buttons appear on the screen. When the game is run;



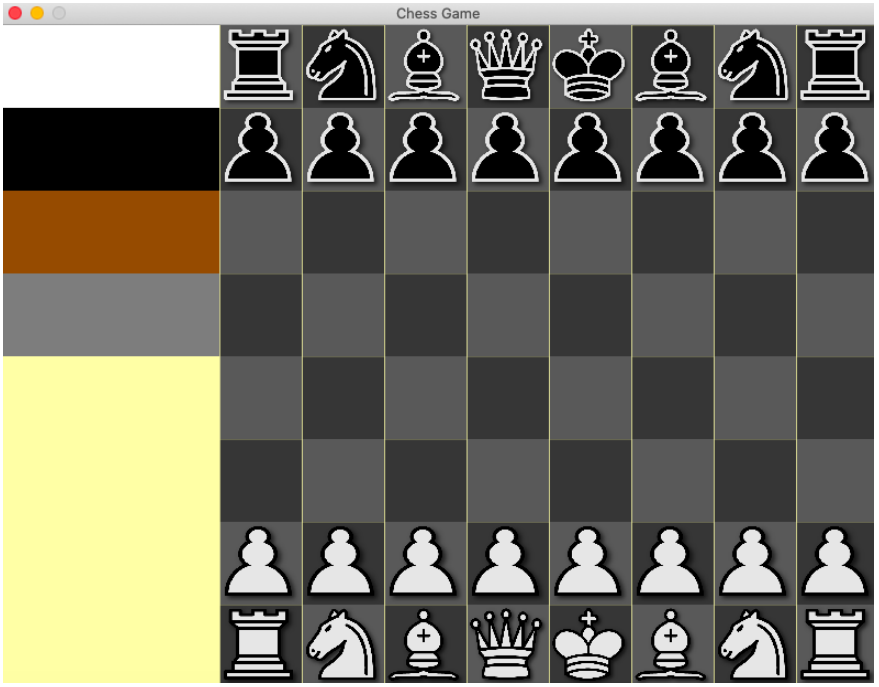
What's being tested	Inputs	Reasoning	Expected Outcome	Actual Outcome
Buttons appearing on screen	None	User must see buttons in order to click on them	4 buttons appear on the screen of the correct colours and of dimension 200 x 75	4 buttons appear on the screen of the correct colours and of dimension 200 x 75

Now I'll test the brown and grey buttons which toggle the board colour.

When brown button is pressed;



Then when the grey button is pressed;

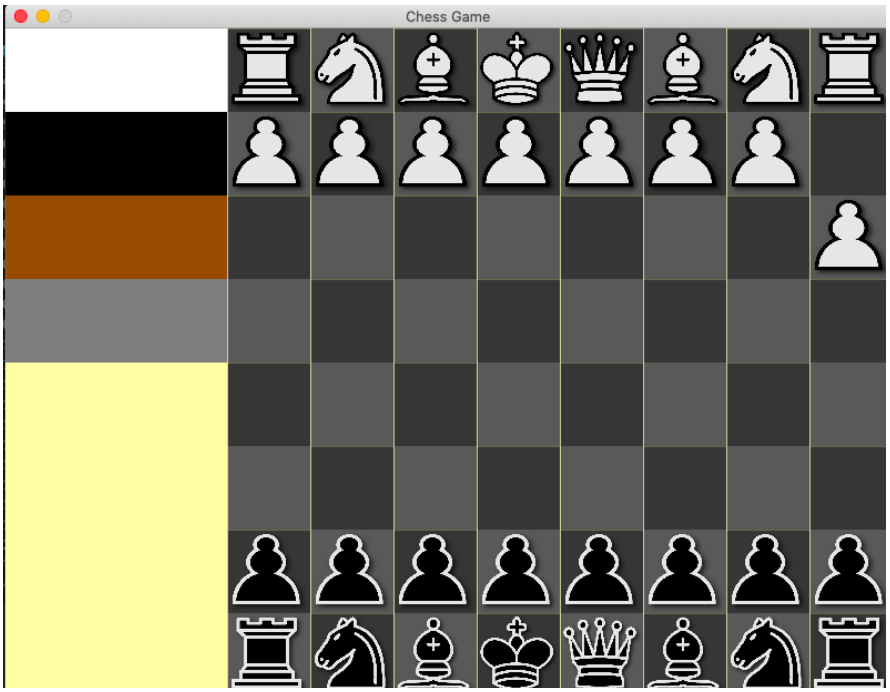


What's being tested	Inputs	Reasoning	Expected Outcome	Actual Outcome
Toggling the board colour with buttons	Click on brown and grey buttons	User should be able to toggle board colour with buttons	Board turns to the corresponding colour when button is pressed	Board turns to the corresponding colour when button is pressed

The brown and grey buttons work.

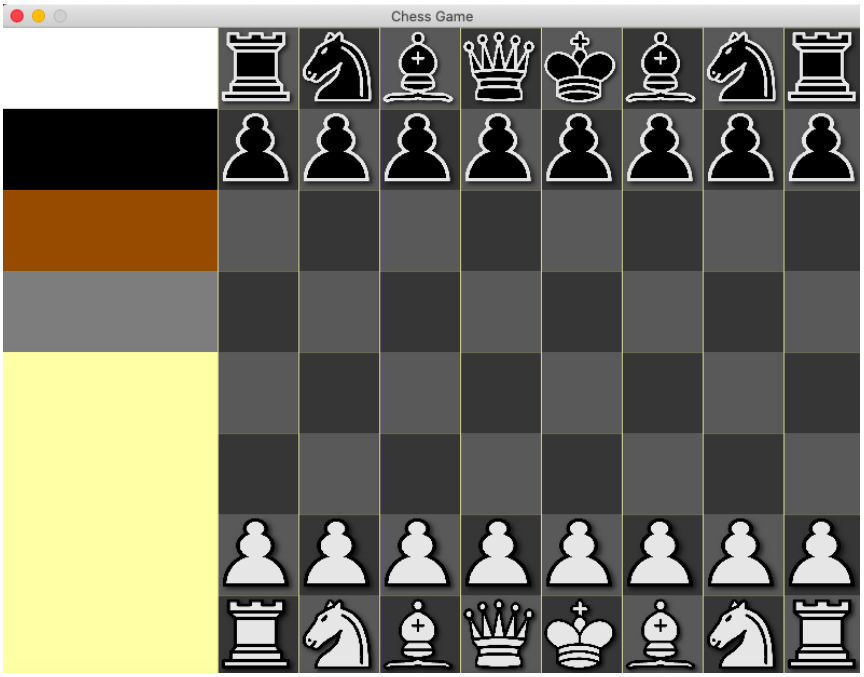
Now it's time to test the change of player colour.

When the black button is pressed (with the original board state being the previous board shown);



(The pawn is moved because the computer will automatically make a move when it's its turn)

Then I tested the white button, when the white button is pressed;



What's being tested	Inputs	Reasoning	Expected Outcome	Actual Outcome
Toggling the player colour with buttons	Click on back and white buttons	User should be able to toggle player colour with buttons	Player colour is set to the corresponding button colour when button is clicked	Player colour is set to the corresponding button colour when button is clicked

The black and white buttons work.

Client Feedback

[redacted] and [redacted] tried out the game buttons and were very happy. [redacted] said, "I am so stoked that the colour can change with the button! The colours clearly indicate what each of them do, so I don't think there's a need to label them."

Prototyping Outputting Game Output - 21/02/2023

The game outcome must be outputted as text to the screen, so a *FONT* and *COLOUR* variable are set to *georgia* font of size 40 and a brown colour.

```
# Create font and text colour
FONT =  pygame.font.SysFont("georgia", 40)
TEXT_COLOUR = (100, 45, 0)
```

The *draw_text()* function is written in order to easily place text on the *screen*.

```
# function to draw text to the screen
def draw_text(text, font, colour, x_pos, y_pos):
    image = font.render(text, True, colour)
    screen.blit(image, (x_pos, y_pos))
```

text is a string of text that will be displayed.

x_pos is the x-coordinate of the top-left corner of where the text will be displayed, and *y_pos* is the y-coordinate of the top-left corner of where the text will be displayed.

When the board state is stalemate, “Stalemate” will be drawn to the *screen*.

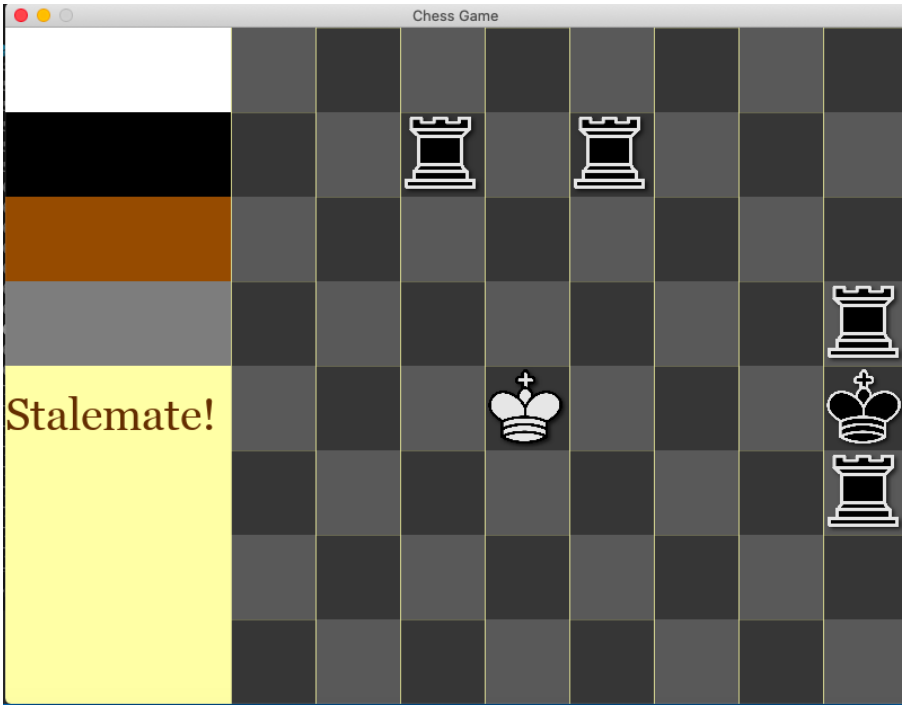
```
if b.isStalemate():
    draw_text("Stalemate!", FONT, TEXT_COLOUR, 0, 320)
```

When the board state is checkmate, the player’s who’s *turn* it is will be the winner.

```
# Output winner if board state is checkmate
if b.isCheckmate():
    if b.getTurn() == "white":
        draw_text("White Wins!", FONT, TEXT_COLOUR, 0, 320)
    else:
        draw_text("Black Wins!", FONT, TEXT_COLOUR, 0, 320)
```

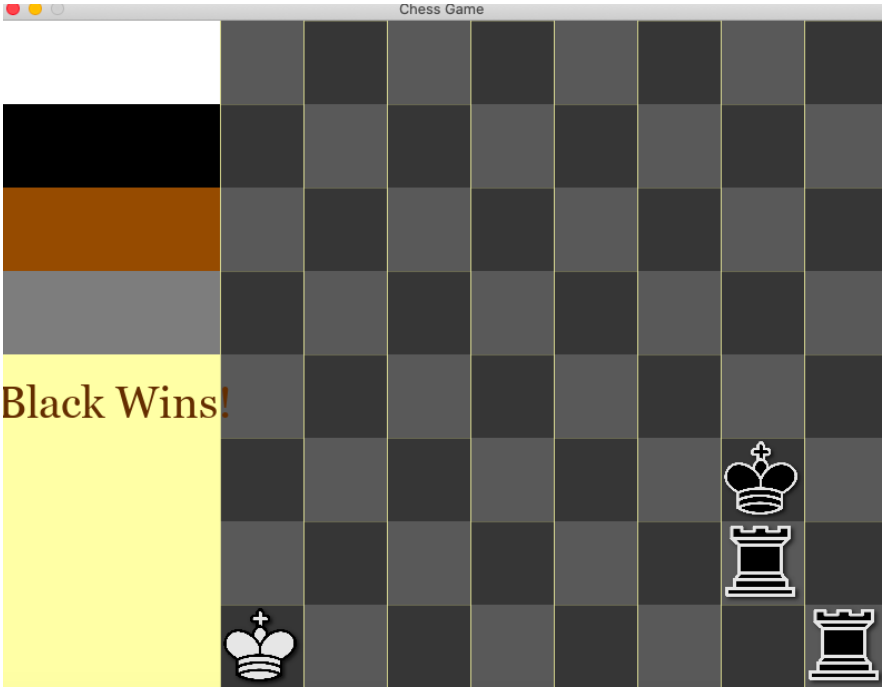
Testing Drawing Game Output (31)

When the game state is stalemate;



What's being tested	Inputs	Reasoning	Expected Outcome	Actual Outcome
Outputting Stalemate	None	Must let the user know game has ended in stalemate is game state is stalemate	"Stalemate!" drawn to screen in brown colour and georgia font, in reasonable size	"Stalemate!" drawn to screen in brown colour and georgia font, in reasonable size

When the board is in a state where black wins;

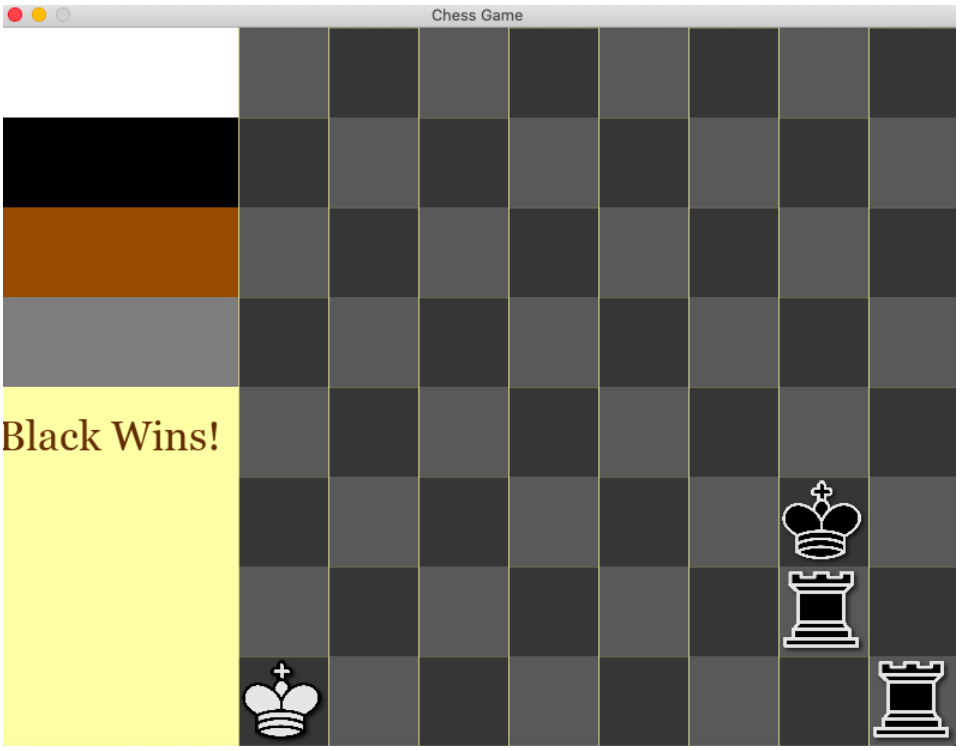


What's being tested	Inputs	Reasoning	Expected Outcome	Actual Outcome
Outputting Black Wins	None	Must let the user know game has ended in checkmate and if black has won	"Black Wins!" drawn to screen in brown colour and georgia font, in reasonable size	"Black Wins!" drawn to screen, but text overlaps the board

The text must be made smaller so that it doesn't overlap the board. The size of the font is changed from 40 to 35.

```
FONT =  pygame.font.SysFont("georgia", 35)
```

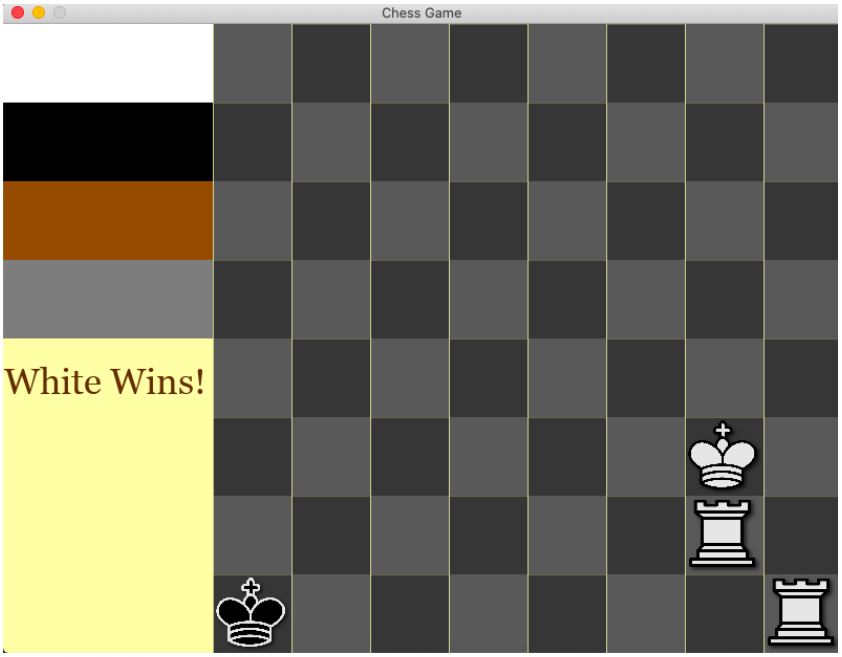
The code is run again;



What's being tested	Inputs	Reasoning	Expected Outcome	Actual Outcome
Outputting Black Wins	None	Must let the user know game has ended in checkmate and if black has won	"Black Wins!" drawn to screen and it not overlap the board	"Black Wins!" drawn to screen and it not overlap the board

The text "Black Wins!" does not overlap the board.

Lastly, we will test for when white wins;



What's being tested	Inputs	Reasoning	Expected Outcome	Actual Outcome
Outputting Black Wins	None	Must let the user know game has ended in checkmate and if white has won	"White Wins!" drawn to screen in brown colour and georgia font, in reasonable size	"White Wins!" drawn to screen in brown colour and georgia font, in reasonable size

Client Feedback

[redacted] and [redacted] said they were happy with the font, size and location of where the game outcome appears. [redacted] said that he wanted to be able to start a new game as soon as the previous one was finished, so I called the *setupBoard()* method after the game output was called.

```
# Output winning colour if the board state is checkmate
if b.isCheckmate():
    if b.getTurn() != "white":
        draw_text("White Wins!", FONT, TEXT_COLOUR, 0, 320)
    else:
        draw_text("Black Wins!", FONT, TEXT_COLOUR, 0, 320)

    b.setupBoard(None)

# Output Stalemate if board state is stalemate
if b.isStalemate():
    draw_text("Stalemate!", FONT, TEXT_COLOUR, 0, 320)

    b.setupBoard(None)
```

I don’t need to test this as I have tested this function many times before.

Prototyping Changing User Difficulty - 22/02/2023

The depth level and difficulty will be determined by the button the number buttons the player presses on the left of the *screen*. If the player selects 0, then the computer will make random moves, otherwise it will look at that depth level.

First the numbers are drawn to the *screen*.

```
# Draw level difficulty buttons to screen
draw_text("0", FONT, TEXT_COLOUR, 25, 400)
draw_text("1", FONT, TEXT_COLOUR, 125, 400)
draw_text("2", FONT, TEXT_COLOUR, 25, 475)
draw_text("3", FONT, TEXT_COLOUR, 125, 475)
draw_text("4", FONT, TEXT_COLOUR, 25, 550)
draw_text("5", FONT, TEXT_COLOUR, 125, 550)
```

Within the *MOUSEBUTTONDOWN* event handler, the coordinates of *clickedX* and *clickedY* are checked to find which number is pressed. Depending on the number pressed, *comp* is reset to an *AI* of that depth.

```
elif clickedY < 450 and clickedX < 100:
    # Random moves
    comp = AI(0)

elif clickedY < 450 and clickedX < 200:
    # Depth level 1
    comp = AI(1)

elif clickedY < 525 and clickedX < 100:
    # Depth level 2
    comp = AI(2)

elif clickedY < 525 and clickedX < 200:
    # Depth level 3
    comp = AI(3)

elif clickedY < 600 and clickedX < 100:
    # Depth level 4
    comp = AI(4)

elif clickedY < 600 and clickedX < 200:
    # Depth level 5
    comp = AI(5)
```

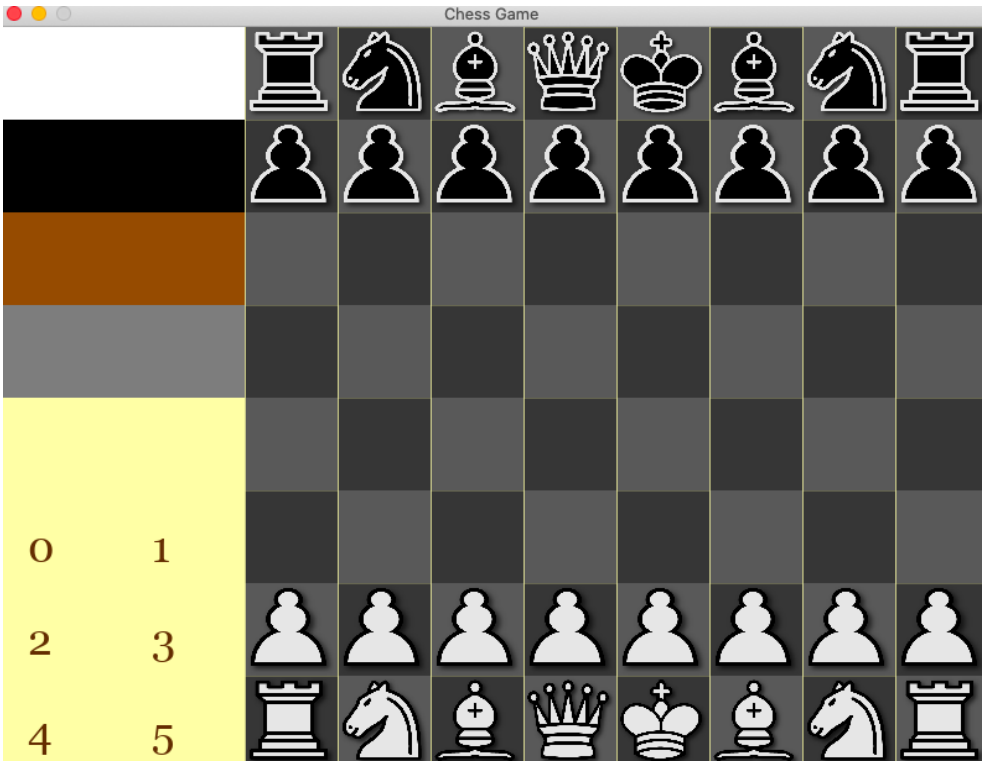
Clicking on each button should allow the player to change the difficulty during gameplay.

Testing Changing Difficulty (32)

print() statements are placed in each condition to indicate the button pressed. Depending on what is printed will indicate if the difficulty has been changed.

```
elif clickedY < 450 and clickedX < 100:
    # Random moves
    comp = AI(0)
    print("Random Moves")

elif clickedY < 450 and clickedX < 200:
    # Depth level 1
    comp = AI(1)
    print(1)

elif clickedY < 525 and clickedX < 100:
    # Depth level 2
    comp = AI(2)
    print(2)

elif clickedY < 525 and clickedX < 200:
    # Depth level 3
    comp = AI(3)
    print(3)

elif clickedY < 600 and clickedX < 100:
    # Depth level 4
    comp = AI(4)
    print(4)

elif clickedY < 600 and clickedX < 200:
    # Depth level 5
    comp = AI(5)
    print(5)
```

When run the output shows the numbers on the *screen*;



When 0 is pressed;

When 1 is pressed;

When 2 is pressed;

When 3 pressed;

When 4 is pressed;

When 5 is pressed;

What's being tested	Inputs	Reasoning	Expected Outcome	Actual Outcome
Toggling difficulty with buttons	Click on each number	User should be able to change difficulty of engine	Prints the corresponding difficulty when each button is pressed	Prints the corresponding difficulty when each button is pressed

It works!

Client Feedback

[redacted] and [redacted] played the game for a while to help test the change of user difficulty during gameplay.

[redacted]said; “It’s amazing! I can tell that the computer makes worse and better moves when I change the difficulty.”

[redacted]n said; “Not gonna lie, wasn’t convinced you'd do it. Super impressive, well done!”

The project is complete!

Evaluation

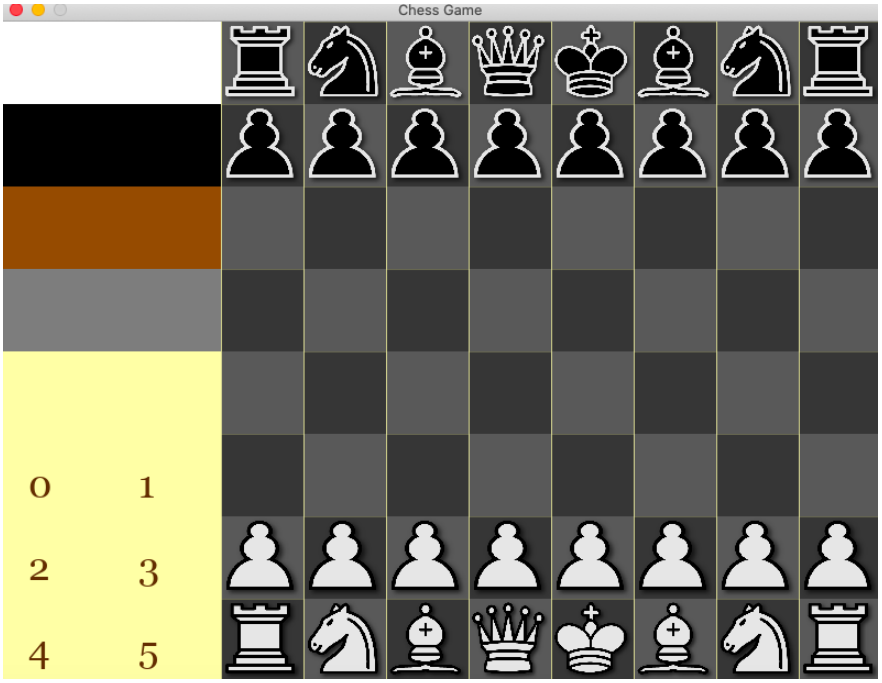
Post Development Testing

Now that the game is complete, testing for the full game must be carried out. [redacted] and [redacted] tried out different features of the game and achieved different game positions to help me twitch this.

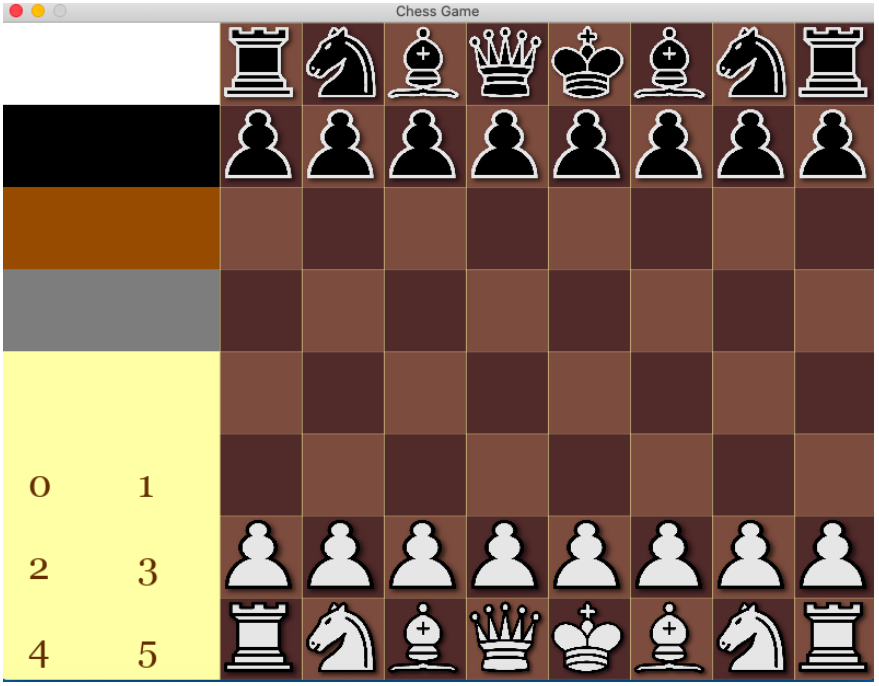
Features to Change the Settings

#	Structure being Tested	Test Data	Type	Reasoning	Test Pass of Fail
1	board_colour	Click on brown button	Valid	Player should be able to change board colour via buttons on the screen	Pass

Before;



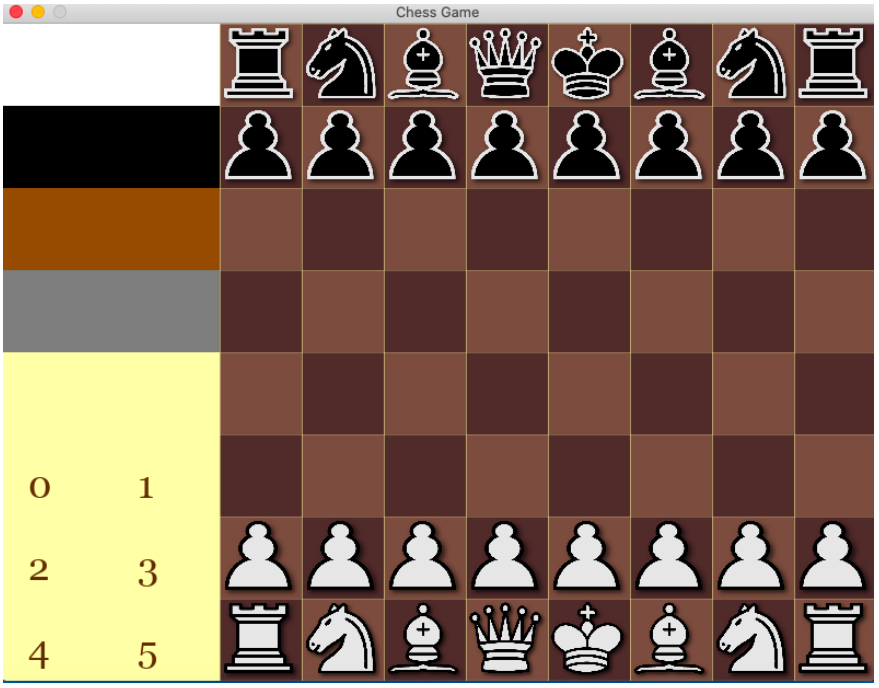
After when brown button is pressed;



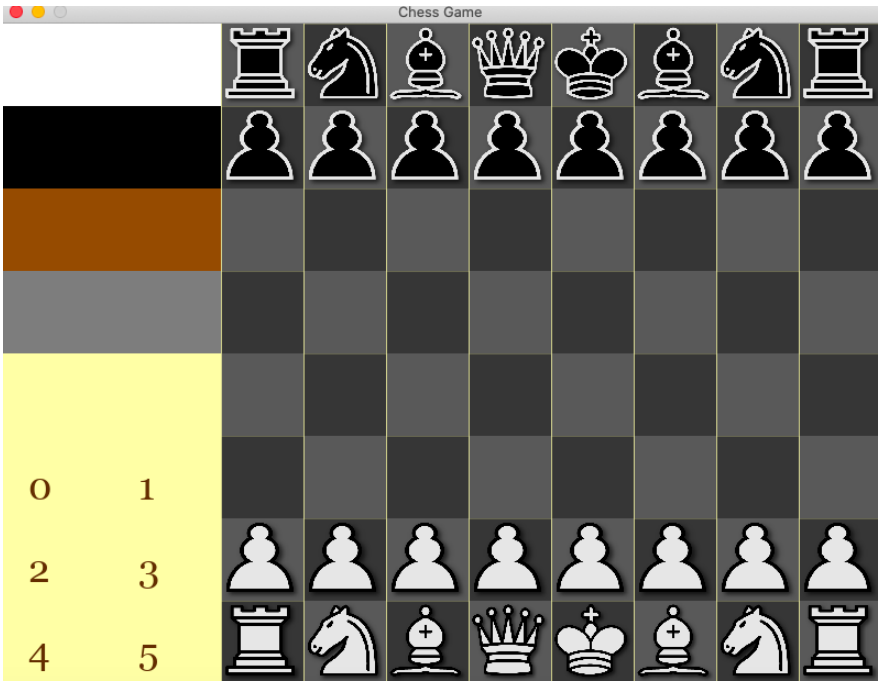
Board colour becomes brown when the brown button is pressed.

2	board_colour	Click on grey button	Valid	Player should be able to change board colour via buttons on the screen	Pass

Before;



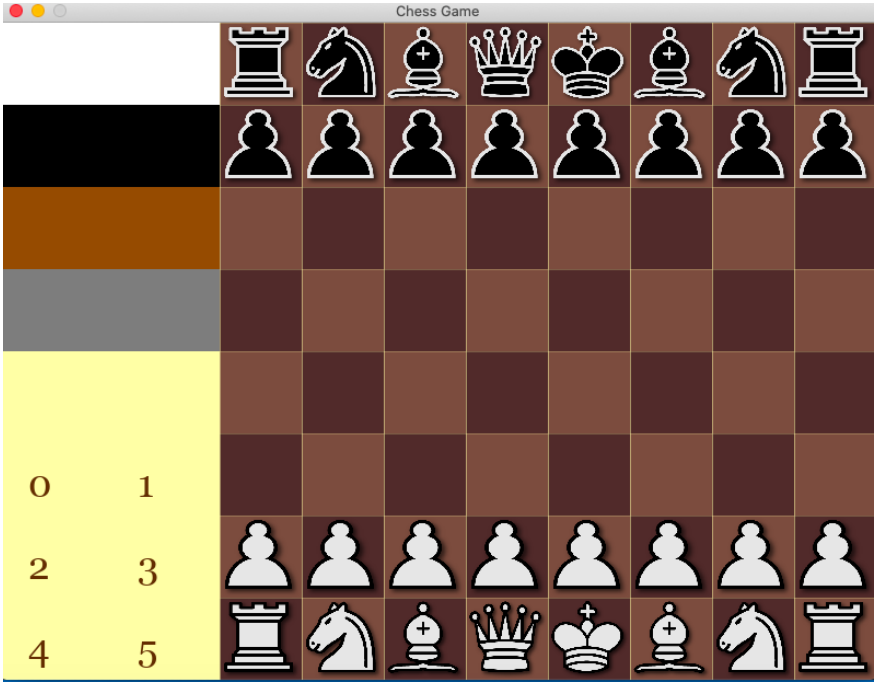
After when grey button is pressed;



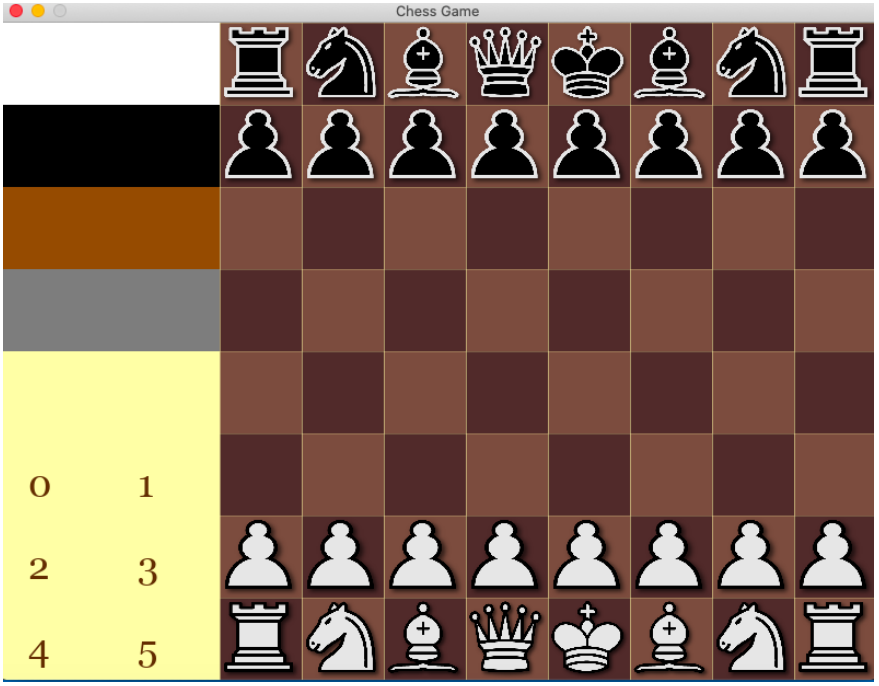
Board colour becomes grey when the grey button is pressed.

3	board_colour	Press “B” key	Invalid	Player shouldn’t be able to change the board colour via the keys	Pass

Before;



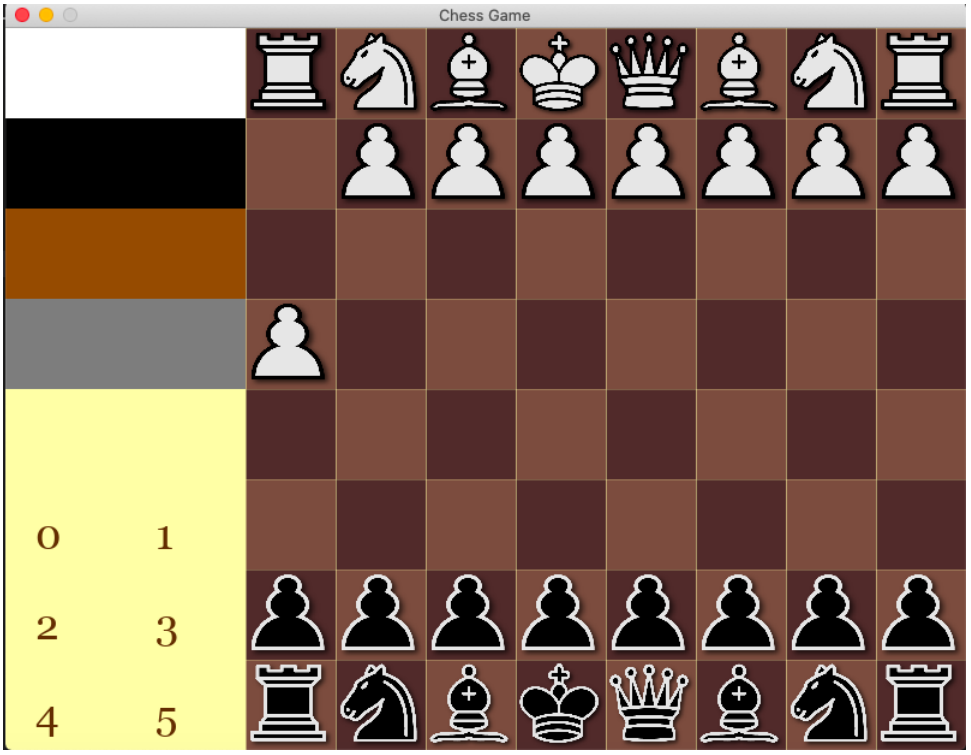
After “B” key is pressed;



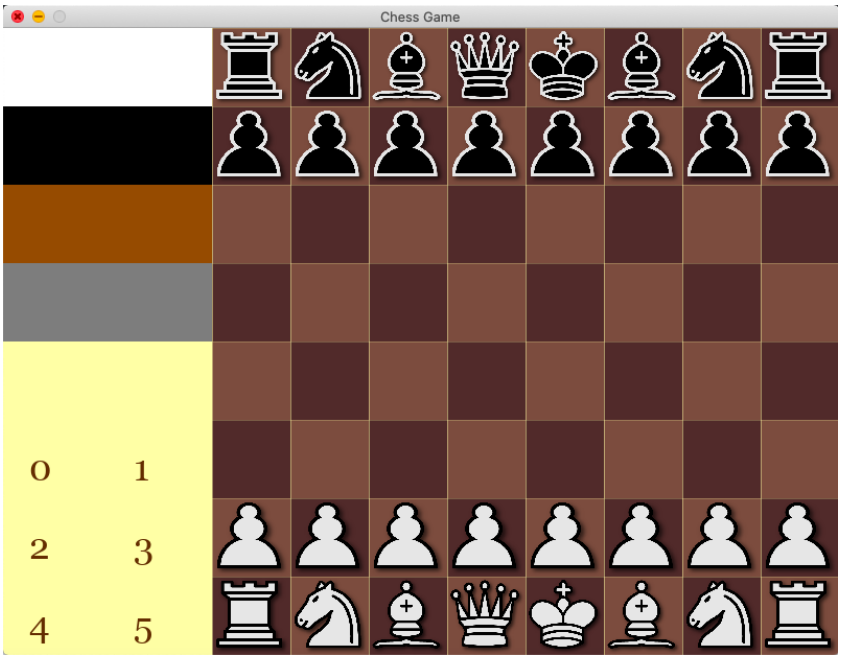
Board colour is unaffected by the keys pressed.

4	player_colour	Click on white button	Valid	Player should be able to change the colour they play as via buttons on the screen	Pass

Before;



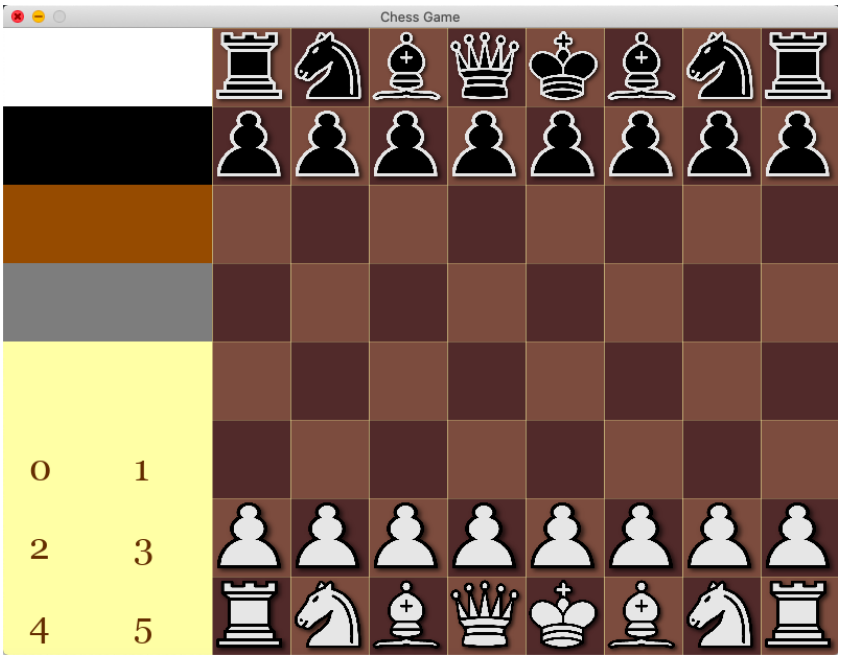
After white button is pressed;



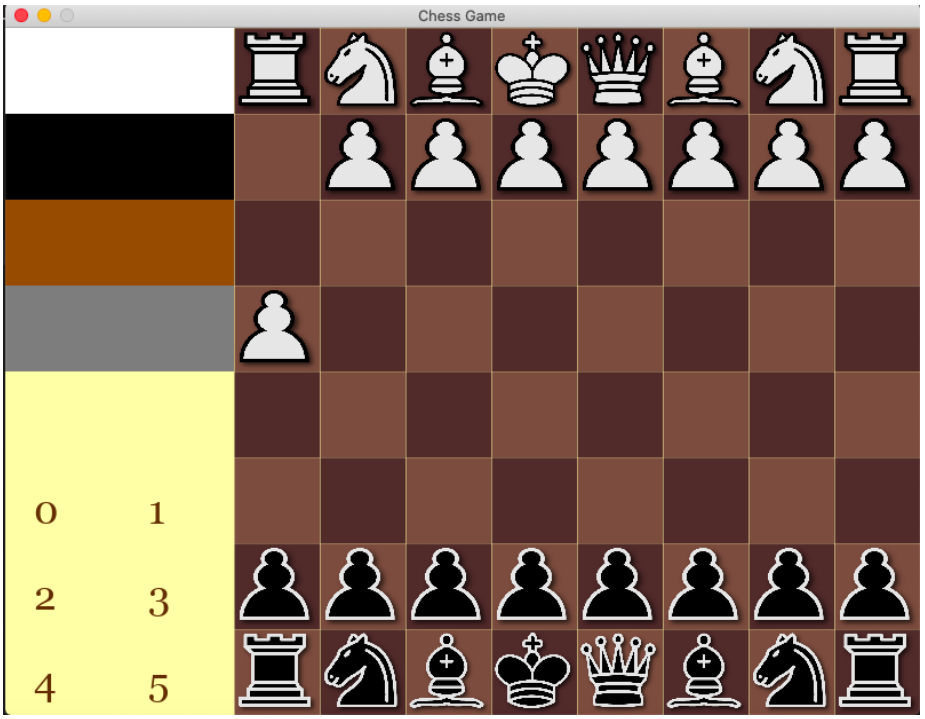
Player colour becomes white when white button is pressed.

5	player_colour	Click on black button	Valid	Player should be able to change the colour they play as via buttons on the screen	Pass

Before;



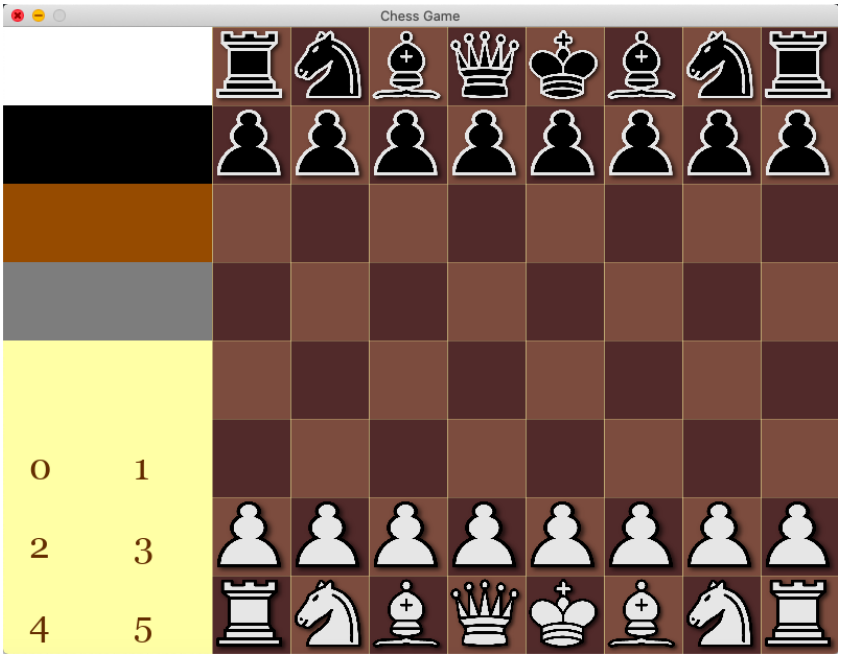
After black button is pressed;



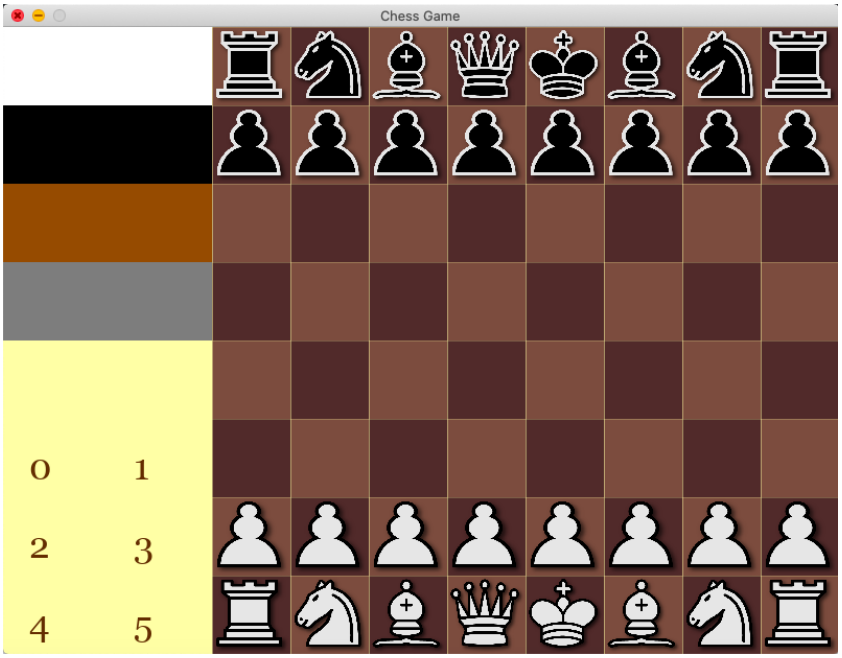
The player colour becomes black when the black button is pressed.

6	player_colour	Press on "W" key	Invalid	Player shouldn't be able to change the colour they play as via keys	Pass

Before;



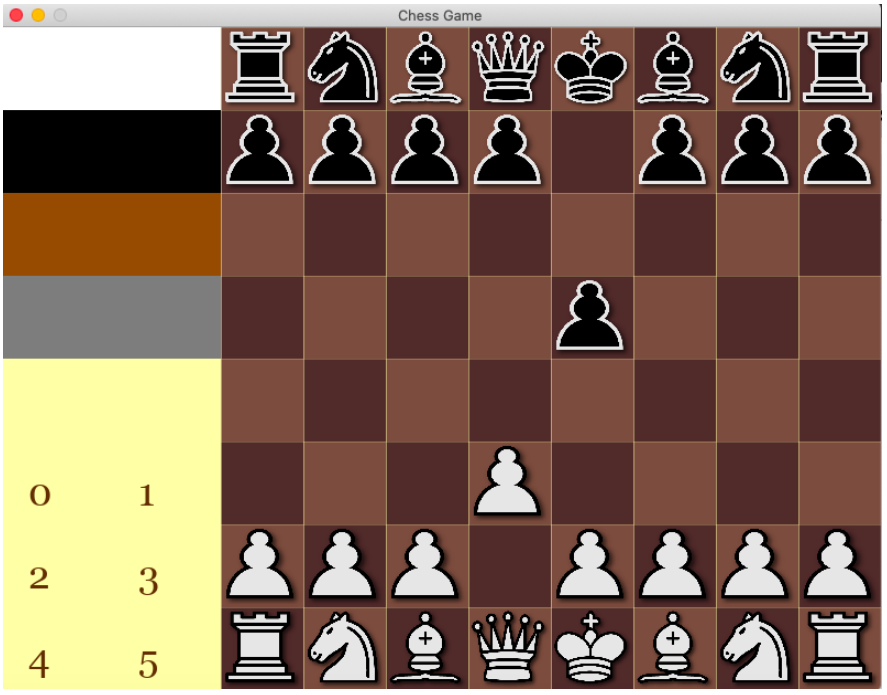
After “W” key is pressed;



Player colour is unaffected by the keys which are pressed.

7	AI	Click on “2” button	Valid	Player should be able to change difficulty level via buttons on the screen	Pass

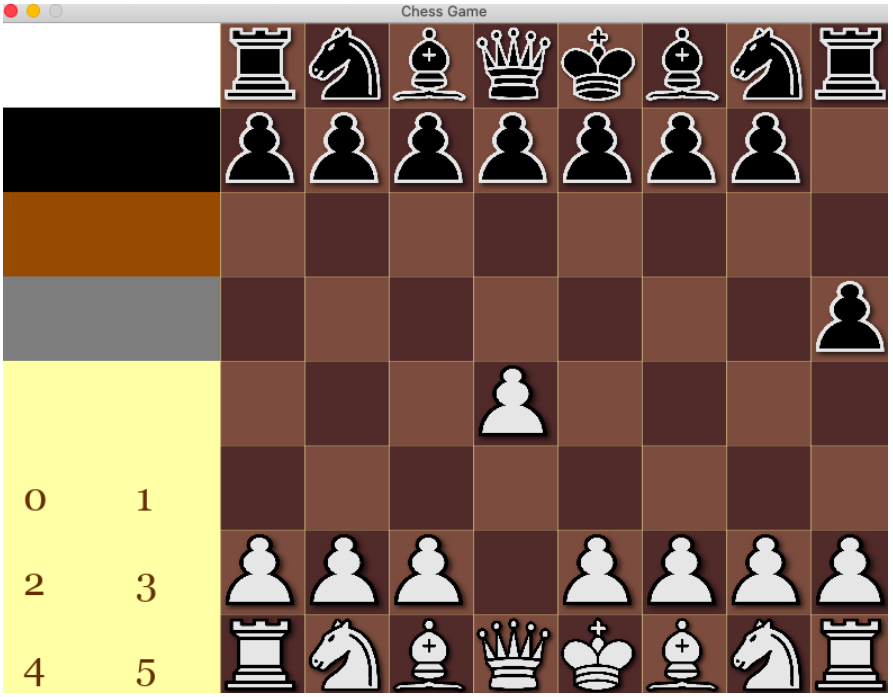
When “2” button is pressed;



Computer responds with a realistic move for a depth level of 2, as pawns in the centre of the board will give it higher points.

8	AI	Press “2” key	Invalid	Player shouldn’t be able to change difficulty level via keys	Pass

When original depth is 0 and “2” key is pressed;

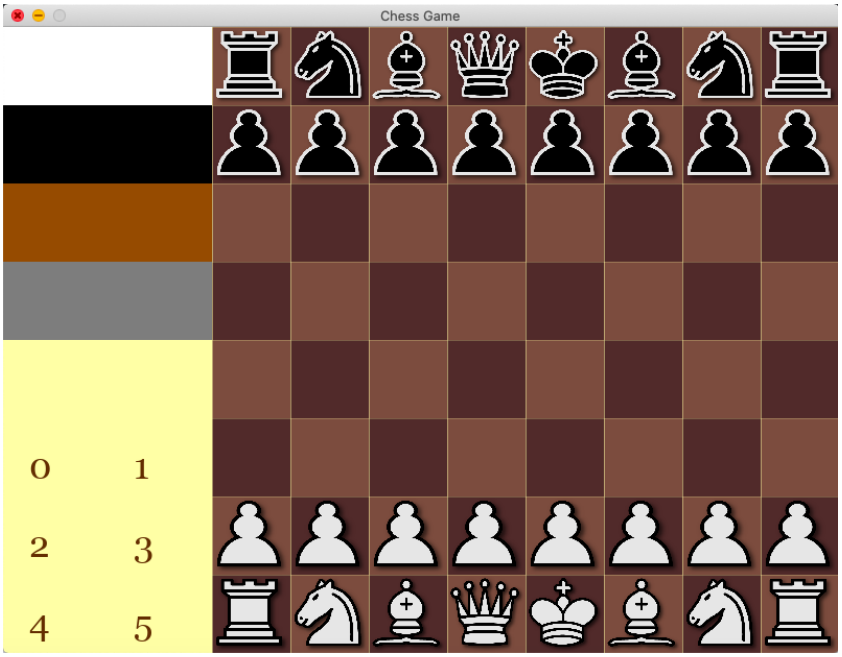


Computer makes a random move which is the default difficulty, so difficulty is not affected by the keys.

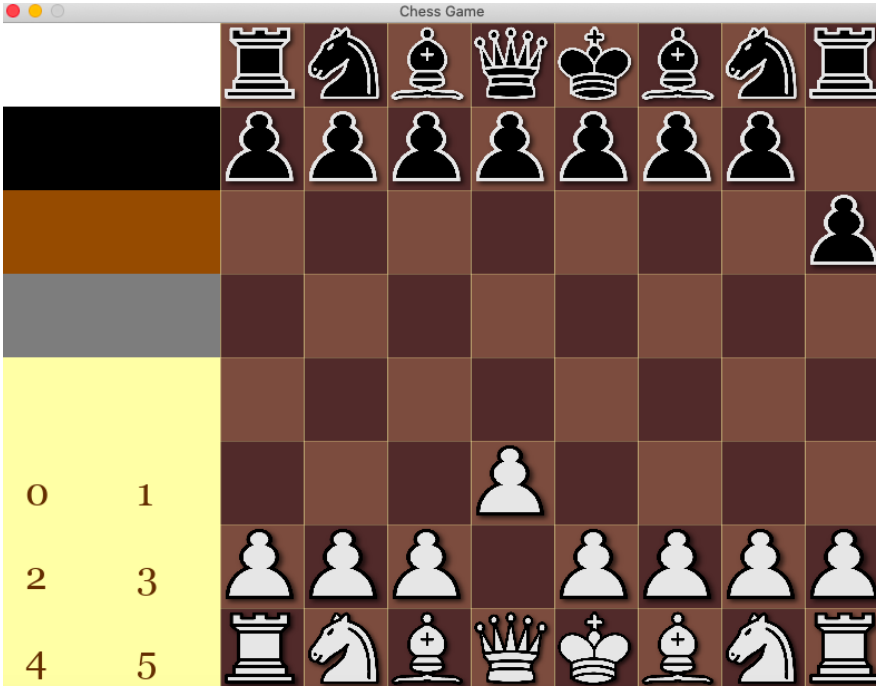
Features During Gameplay

#	Structure being Tested	Test Data	Type	Reasoning	Test Pass of Fail
1	boardClicker()	Click on your pawn then click one space up	Valid	Pieces should only be able to move when clicked to another square	Pass

Before;



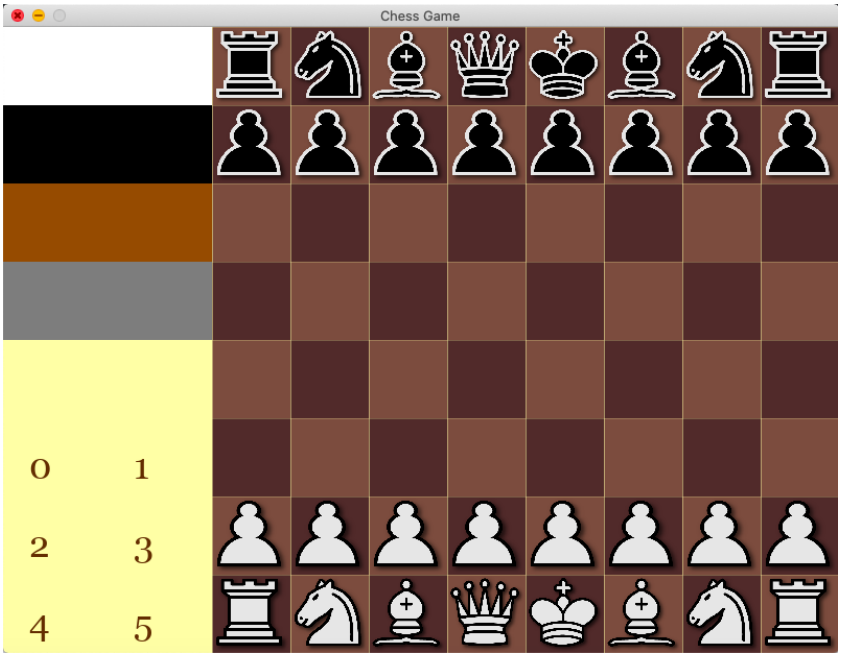
After when pawn is clicked and one space forward is clicked;



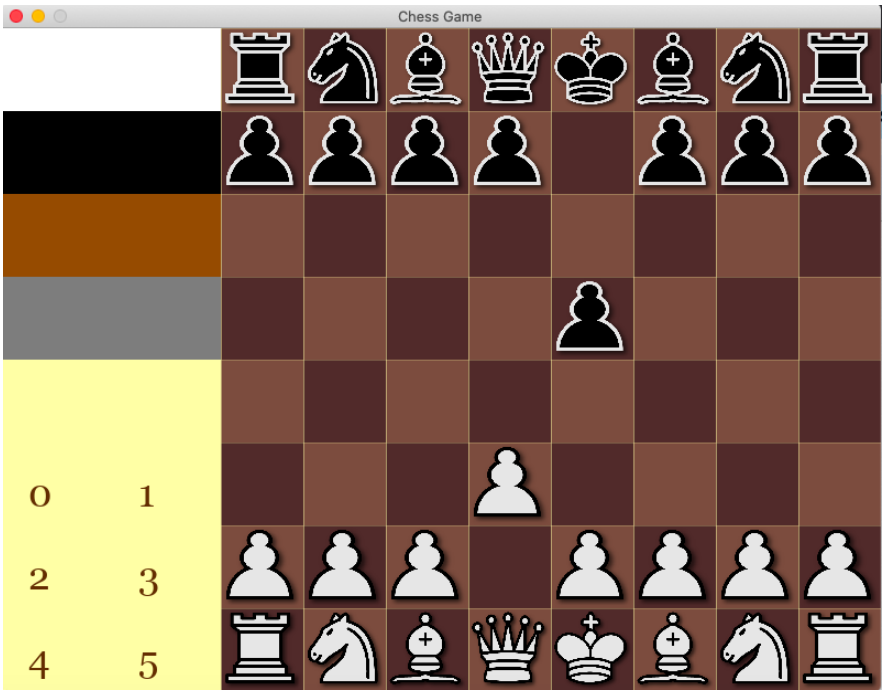
A piece can be clicked to required space as intended.

2	boardClicker()	Drag your pawn from one square to another	Invalid	Pieces shouldn't be able to move when dragged on	Fail

Before;



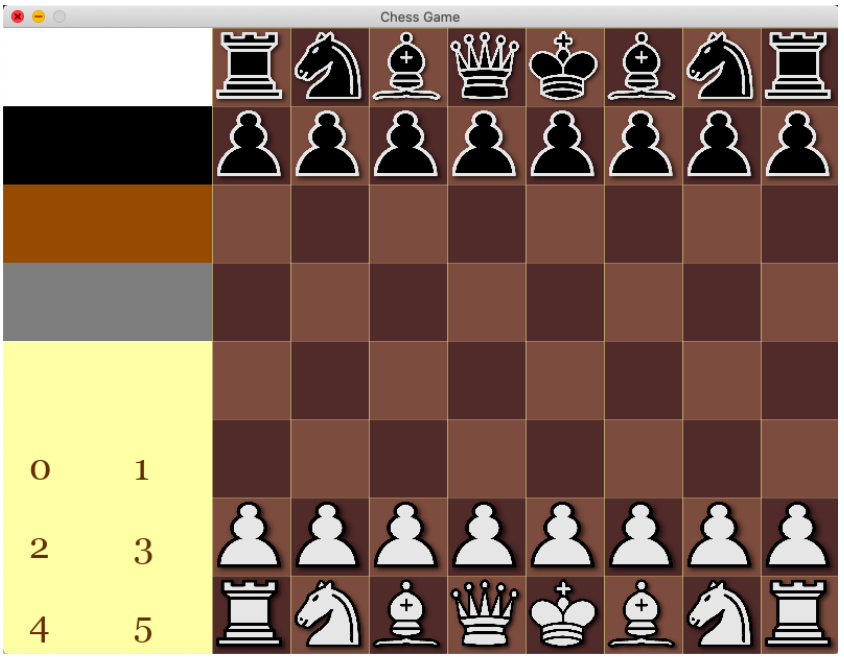
After piece is dragged to square;



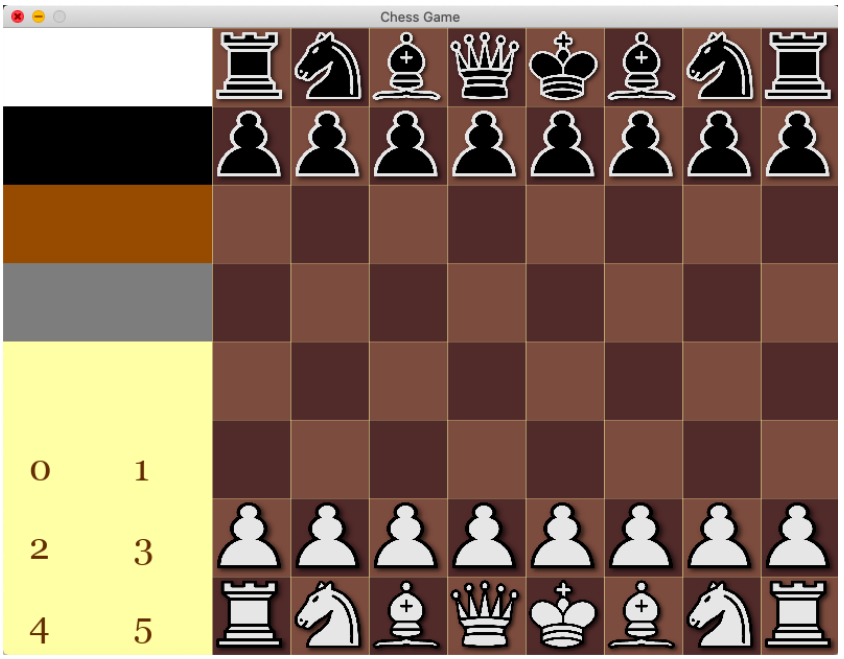
This test failed as the piece shouldn't be able to be dragged to its end position. This is probably because the pygame event handler detects the mouse being pressed and therefore when you attempt to drag. However the pawn did move to the correct square when dragged so the input is valid. I asked [redacted] and [redacted] if they were fine with being able to drag a piece to the correct position so long as it doesn't affect being able to click a piece to its position. Therefore I am not concerned with this test failing as it doesn't affect the rest of the gameplay and if anything enhances the gameplay as now the user has the option to click or drag pieces to the position.

3	boardClicker()	Click on your pawn and press up arrow	Invalid	Pieces shouldn't be moved with keys	Pass

Before;



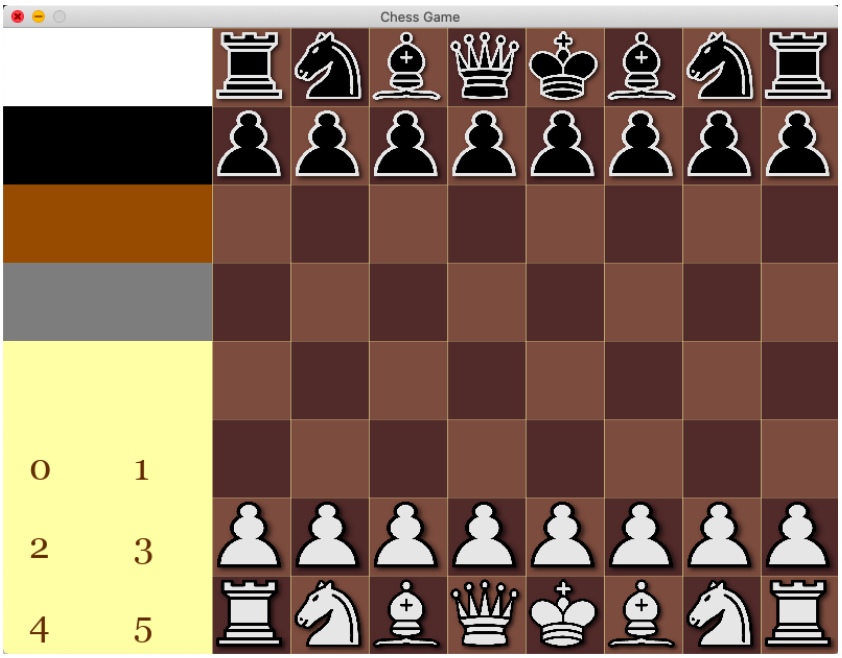
After pawn is clicked and up arrow is pressed;



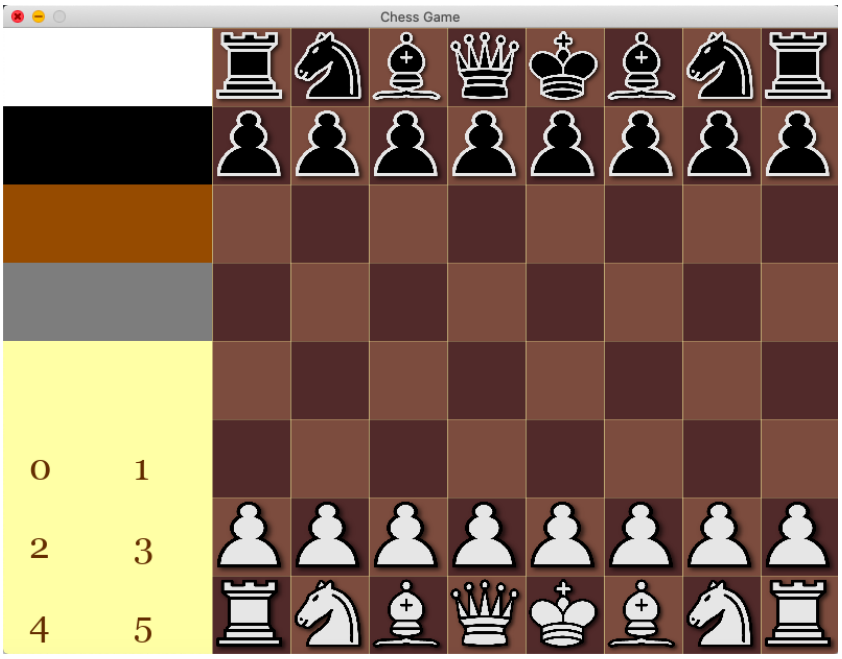
Keys do not affect piece movement as the pawn doesn't move due to keys.

4	boardClicker()	Click on opponent's pawn and click one square ahead	Invalid	Player shouldn't be able to move opponent's pieces	Pass

Before;



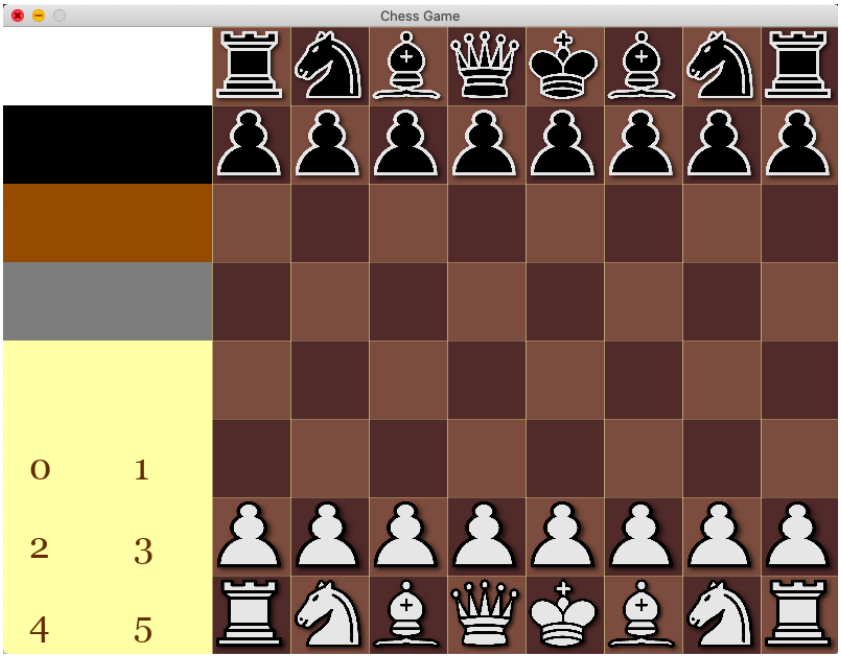
After opposite player pawn is clicked and clicked one square ahead;



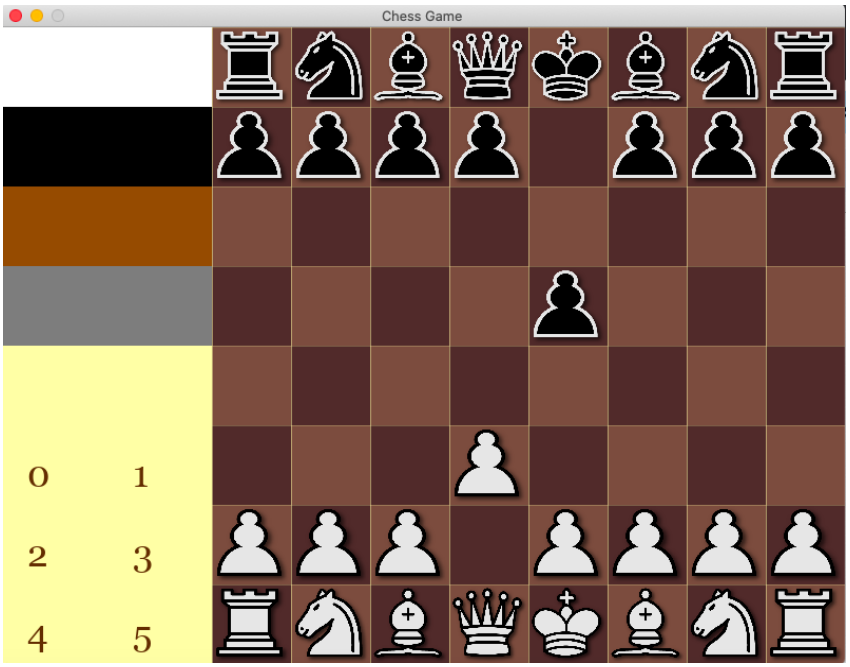
You don't have the ability to move opponents' pieces, only your own.

5	getValidMoves()	Click on pawn to move up one space	Valid	One space up is a valid move for a pawn	Pass

Before;



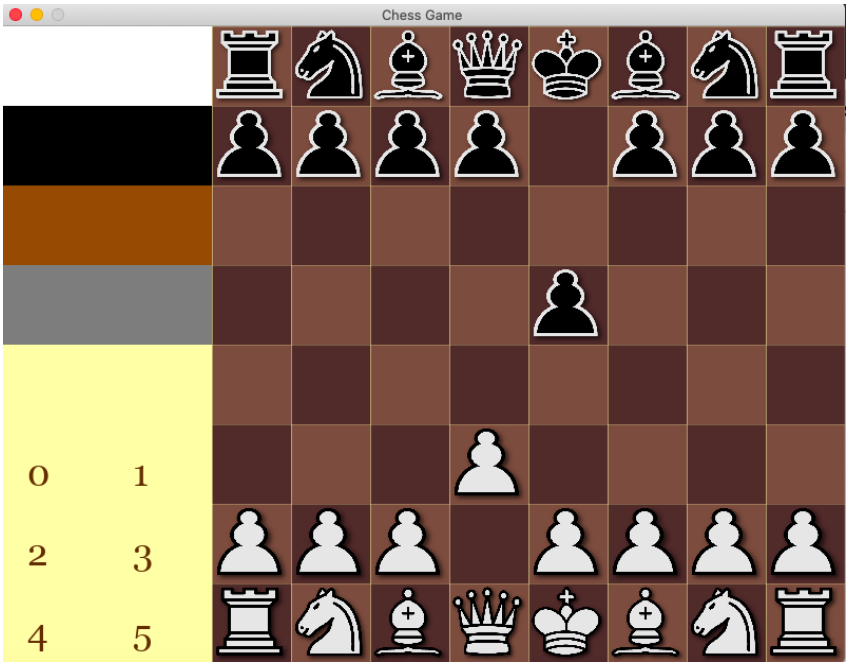
After pawn is clicked one space forward;



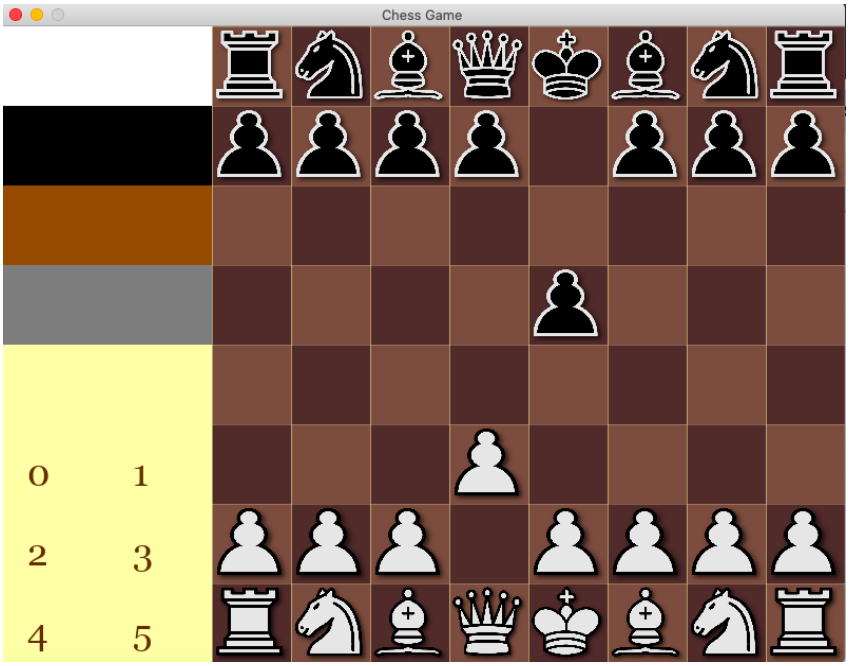
Pawns are able to move to valid positions.

6	getValidMoves()	Click on pawn to move right one space	Invalid	One space right is not a valid move for a pawn	Pass

Before;



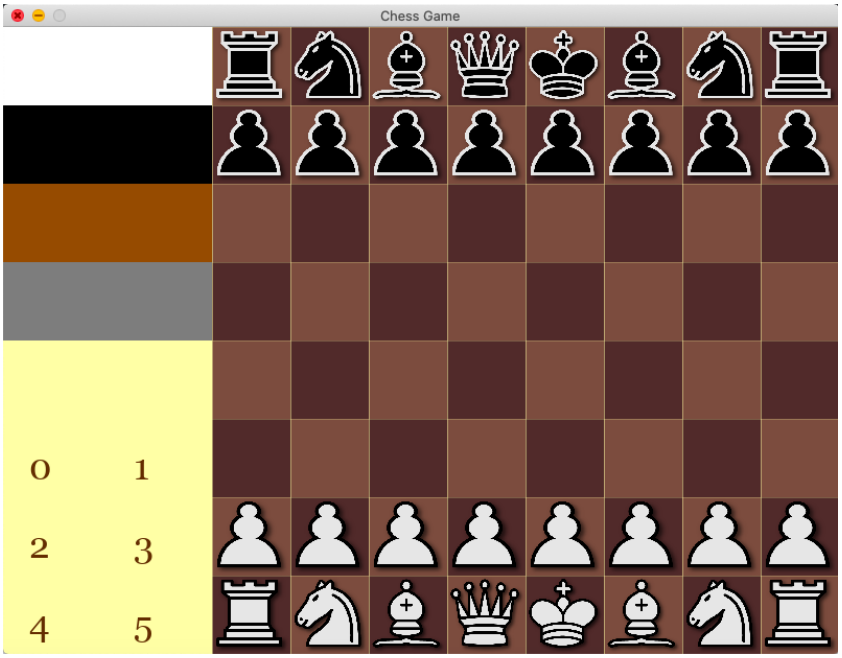
After pawn is clicked one space right;



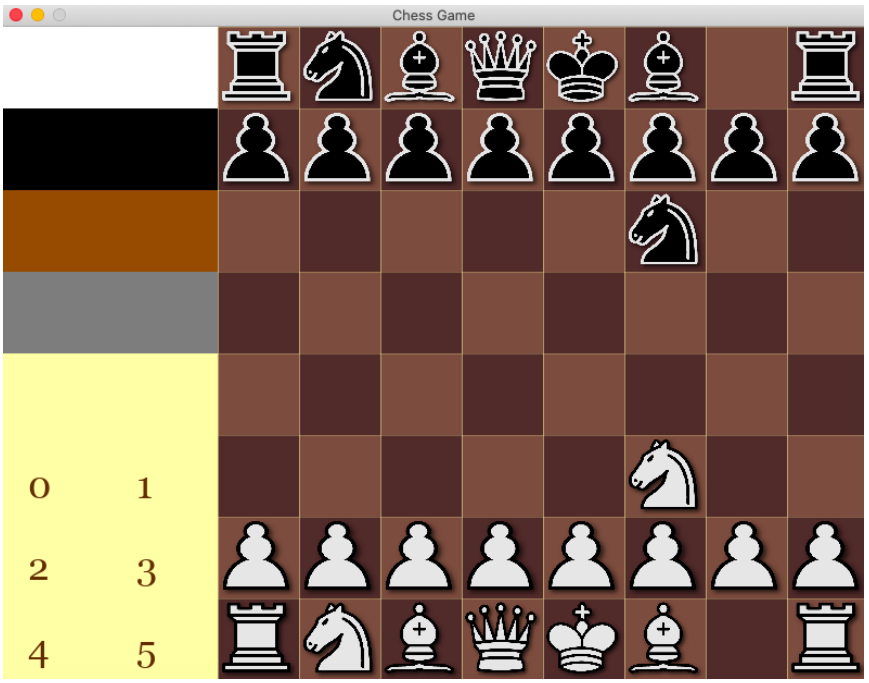
Invalid move for the pawn is disallowed.

7	getValidMoves()	Click on knight to move in an “L” shape	Valid	“L” shape is a valid move for a knight	Pass

Before;



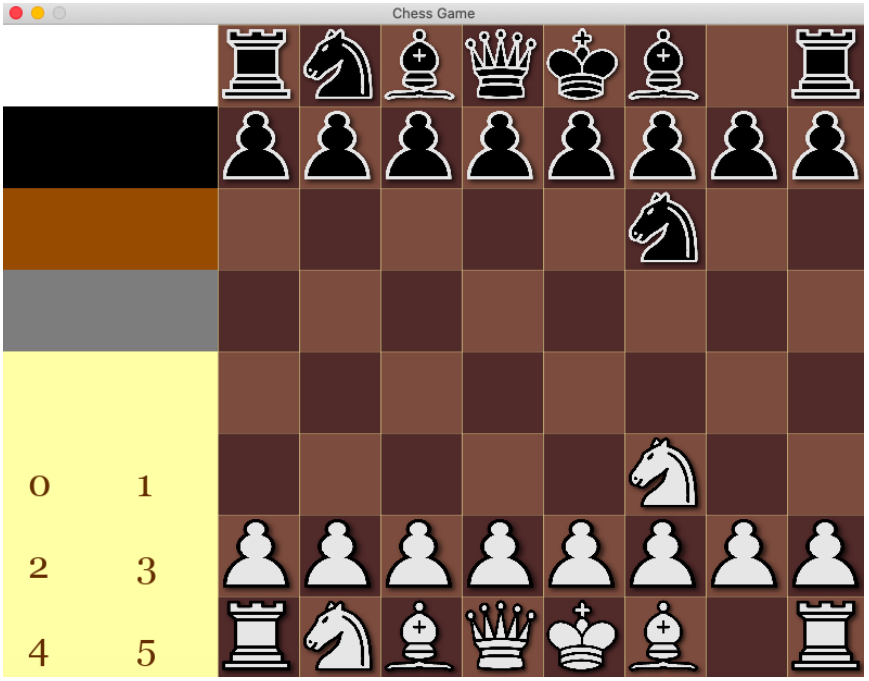
After knight is clicked to move in an “L” shape;



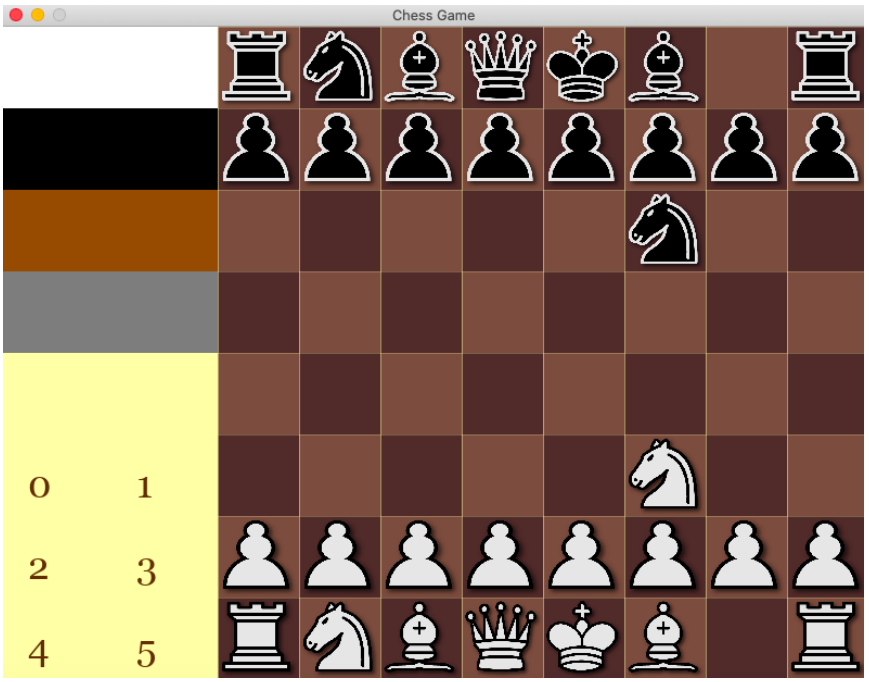
Knights are able to move to valid positions.

8	getValidMoves()	Click on knight to move straight	Invalid	Straight direction is not a valid move for a knight	Pass

Before;



After knight is clicked to move straight;

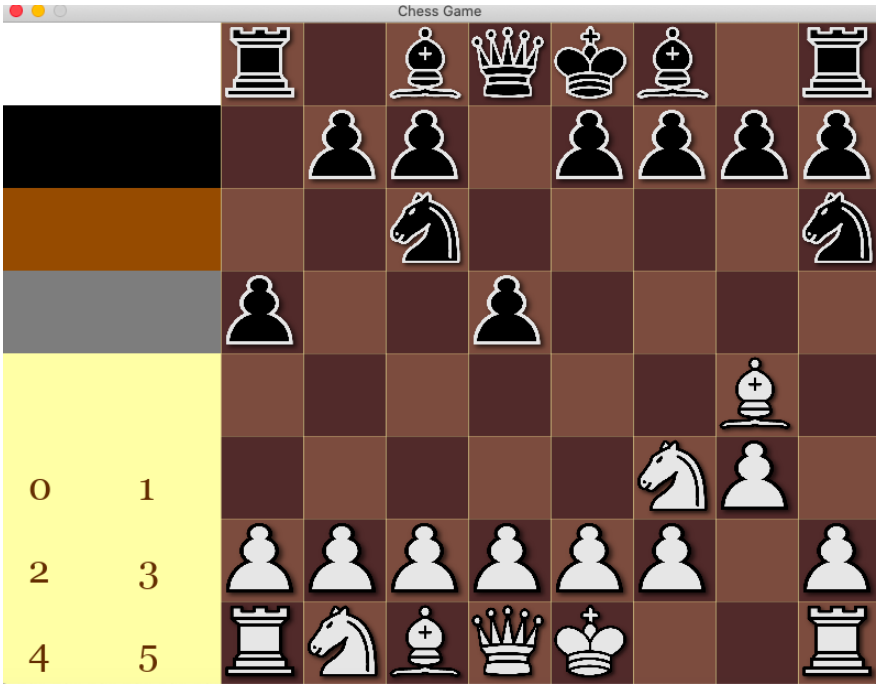


Invalid move for the knight is disallowed.

9	getValidMoves()	Click on bishop to move diagonal	Valid	Diagonal moves are valid for a bishop	Pass

Before;

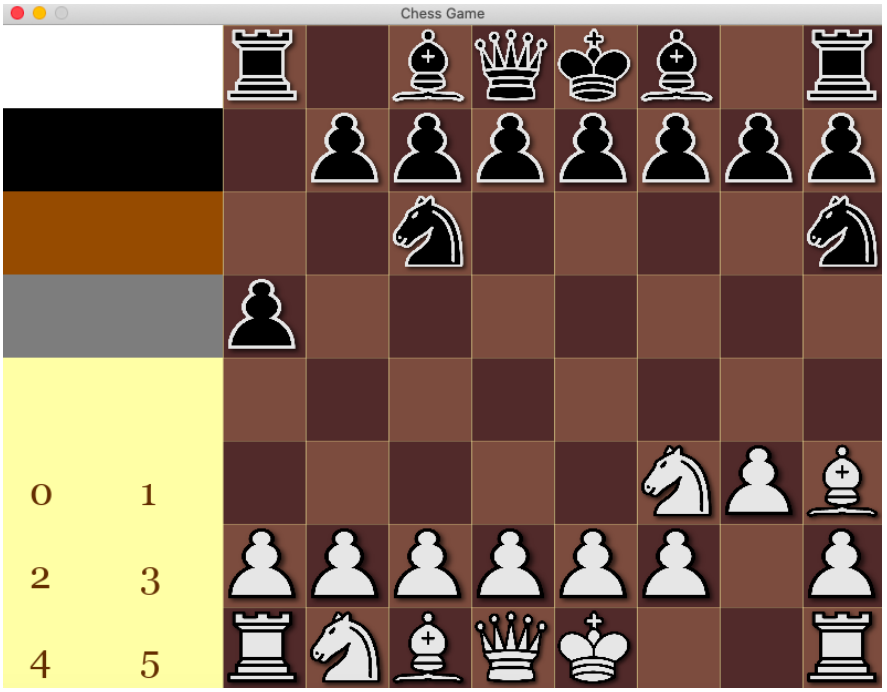
After bishop is clicked to move diagonally;



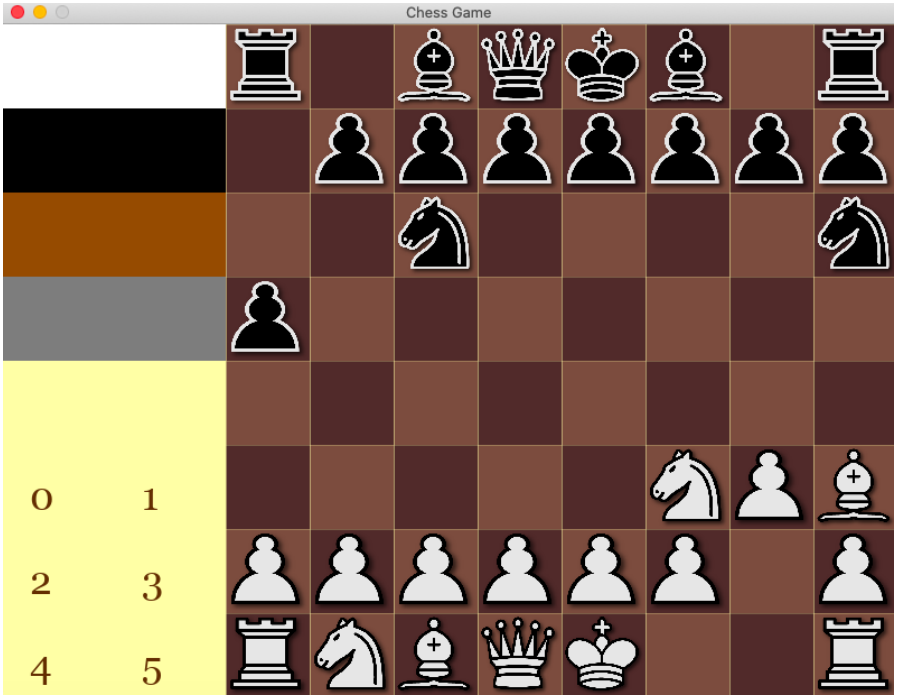
Bishops are able to move to valid positions.

10	getValidMoves()	Click on bishop to move in an “L” shape	Invalid	“L” shape is not a valid move for a bishop	Pass

Before;



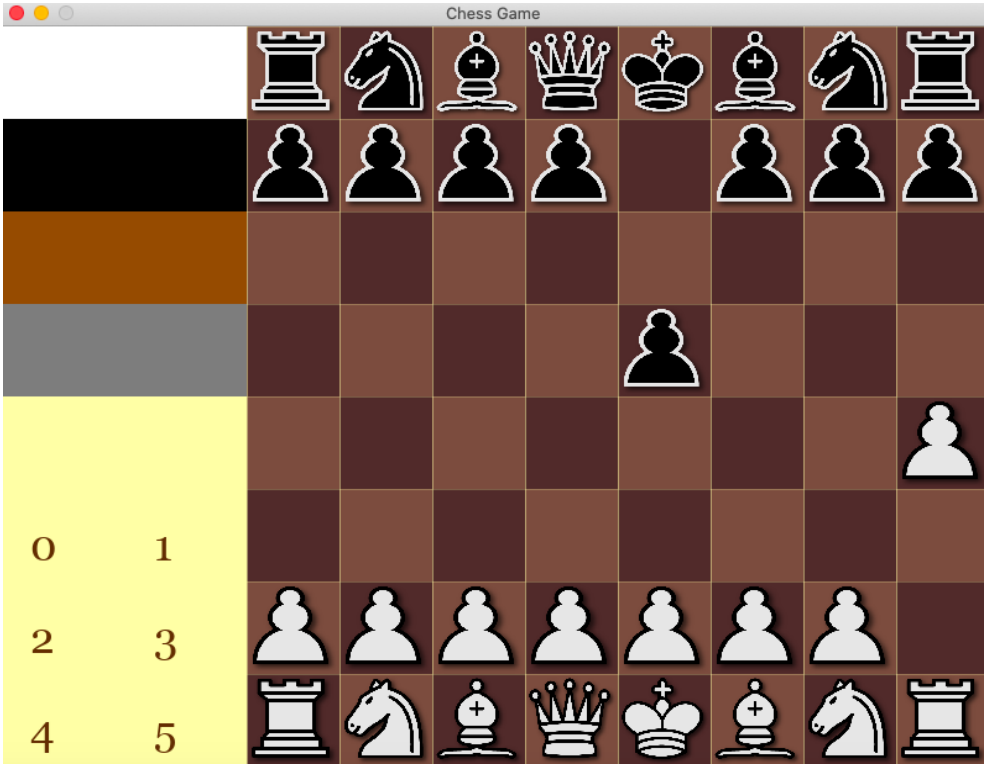
After bishop is clicked to move in an “L” shape;



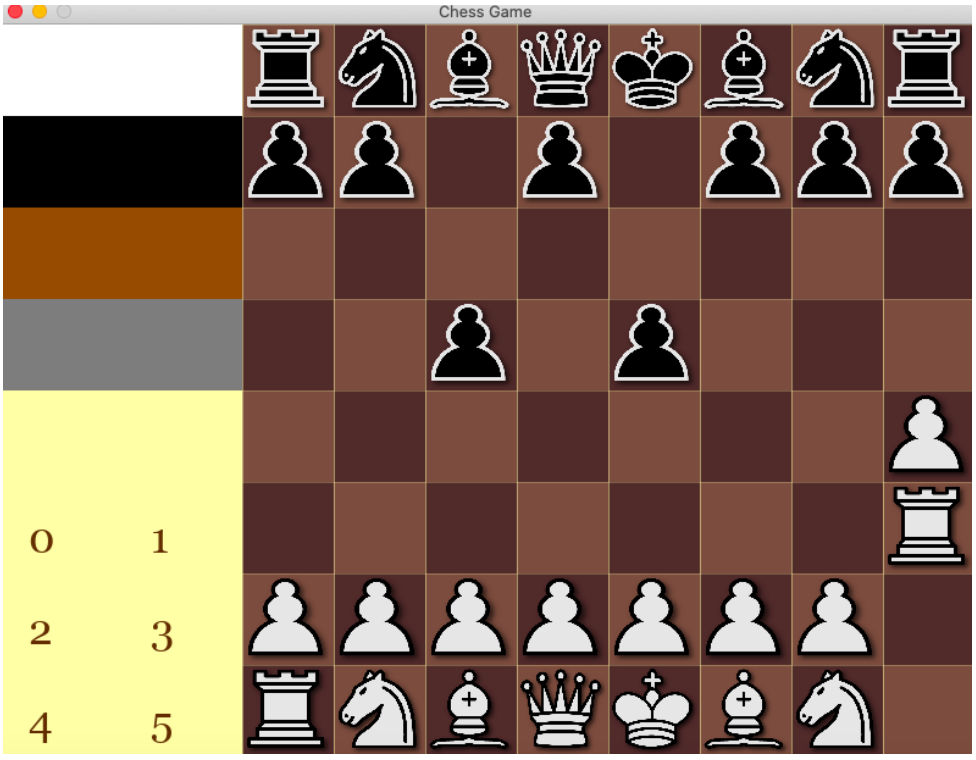
Invalid move for the bishop is disallowed.

11	getValidMoves()	Click on rook to move straight	Valid	Straight moves are valid for a rook	Pass

Before;



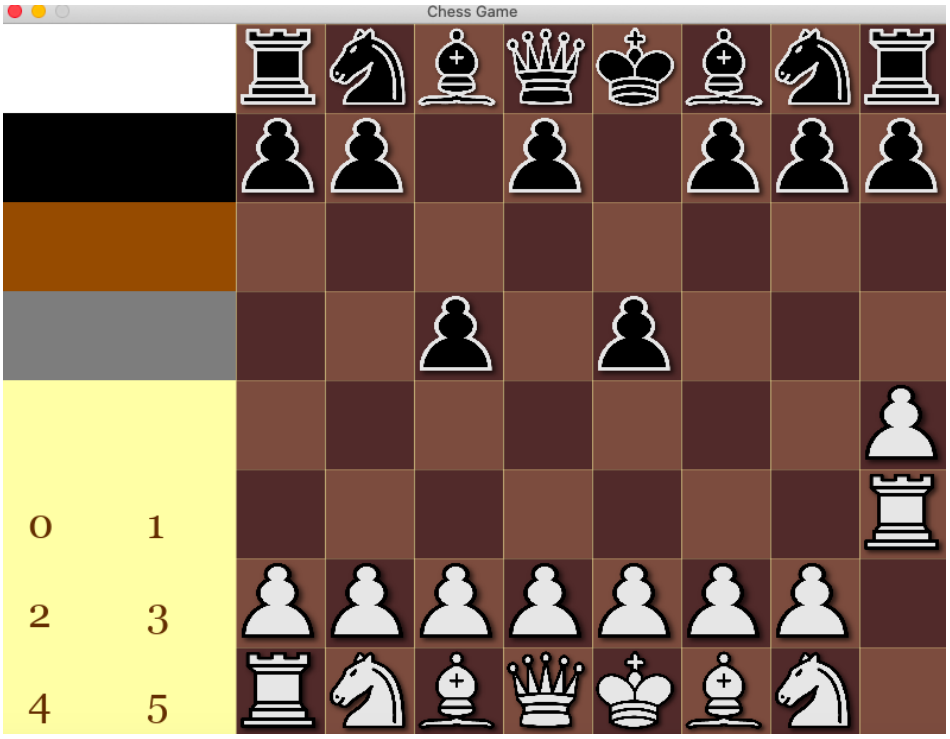
After rook is clicked to move straight;



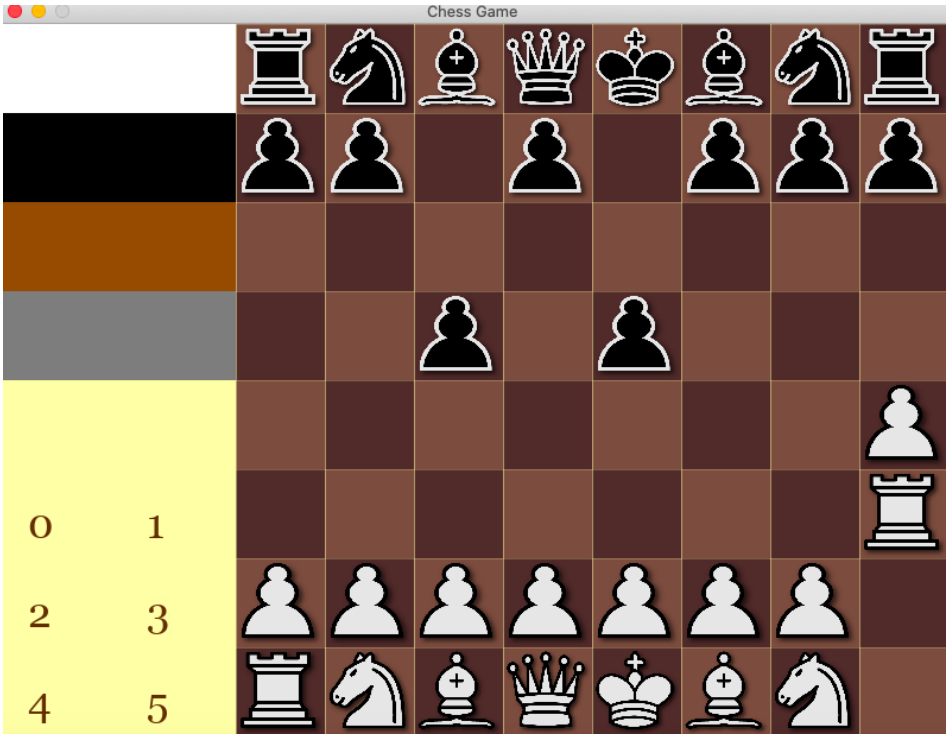
Rooks are able to move to valid positions.

12	getValidMoves()	Click on rook to move diagonal	Invalid	Diagonal moves are not valid for the rook	Pass

Before;



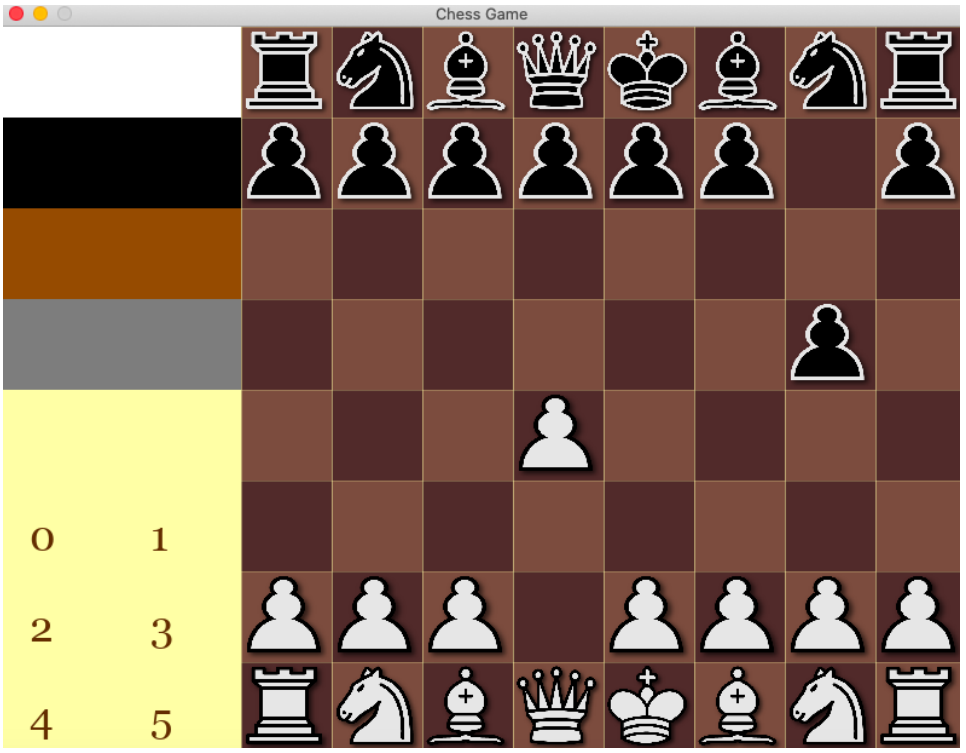
After rook is clicked to move diagonally;



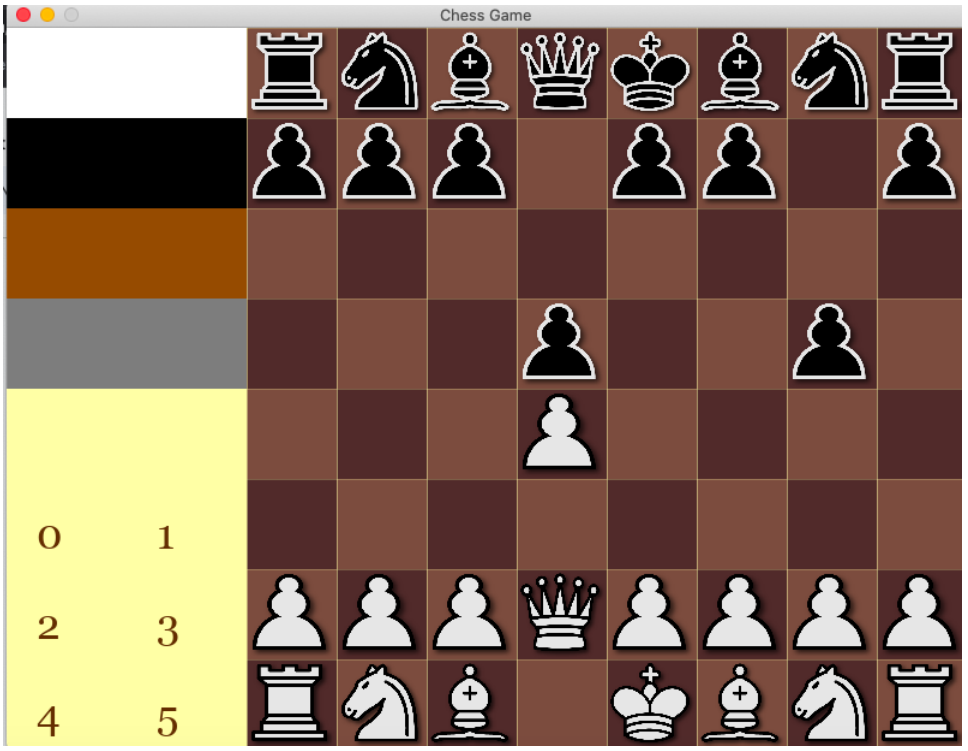
Invalid moves for the rook are disallowed.

13	getValidMoves()	Click on queen to move straight	Valid	Straight moves are valid for a queen	Pass

Before



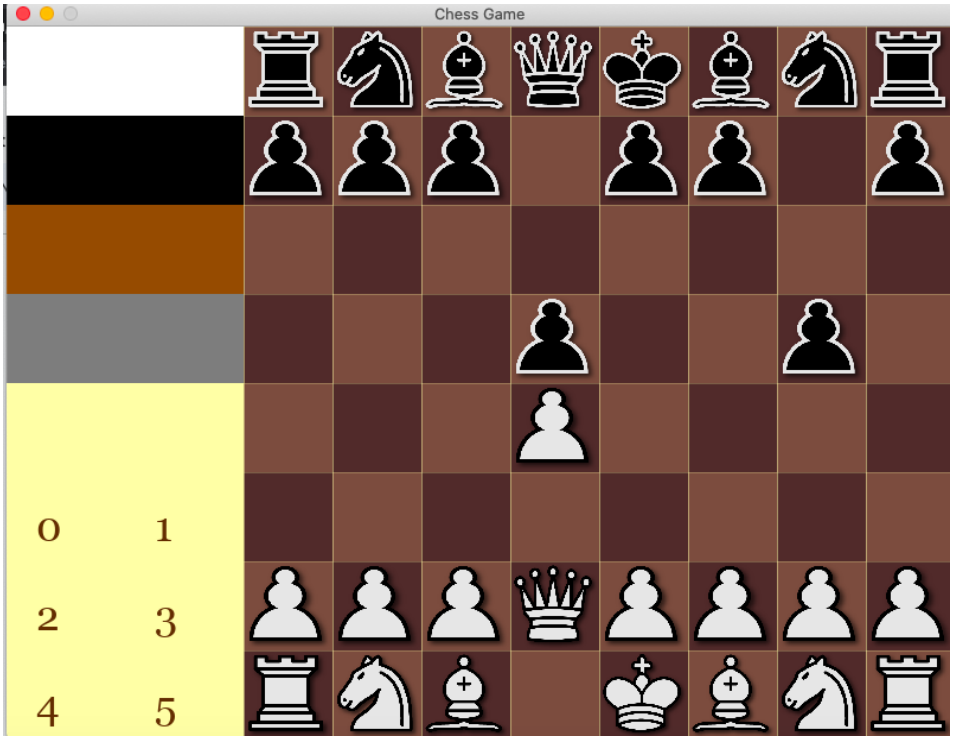
After queen is clicked to move straight;



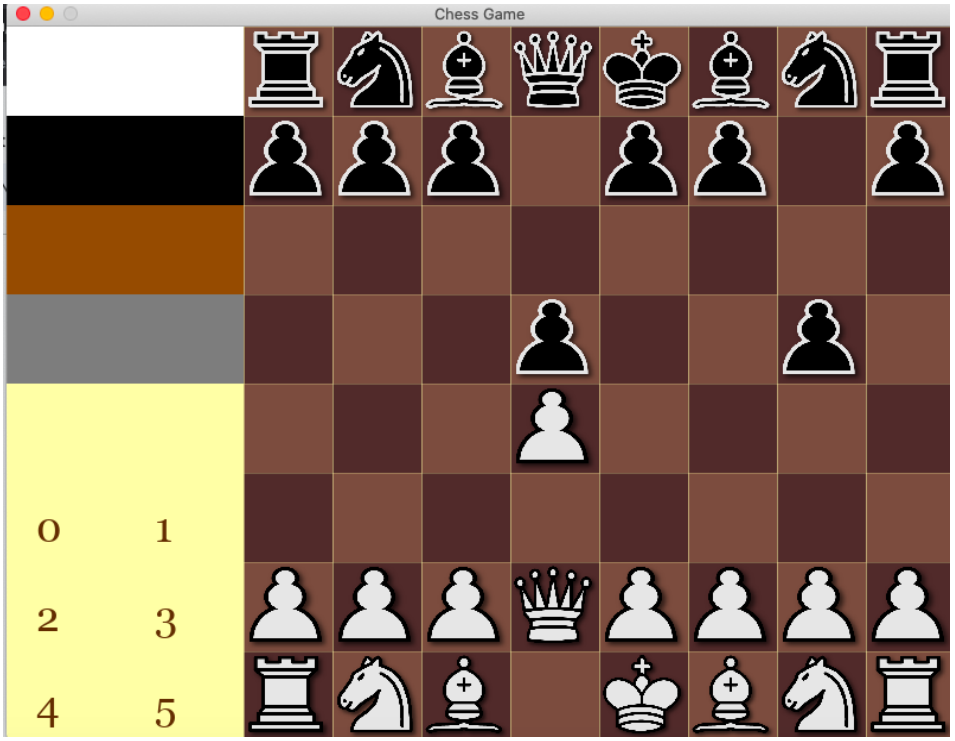
The queen is able to move to valid positions.

14	getValidMoves()	Click on queen to move in an “L” shape	Invalid	“L” shape is not valid for a queen	Pass

Before;



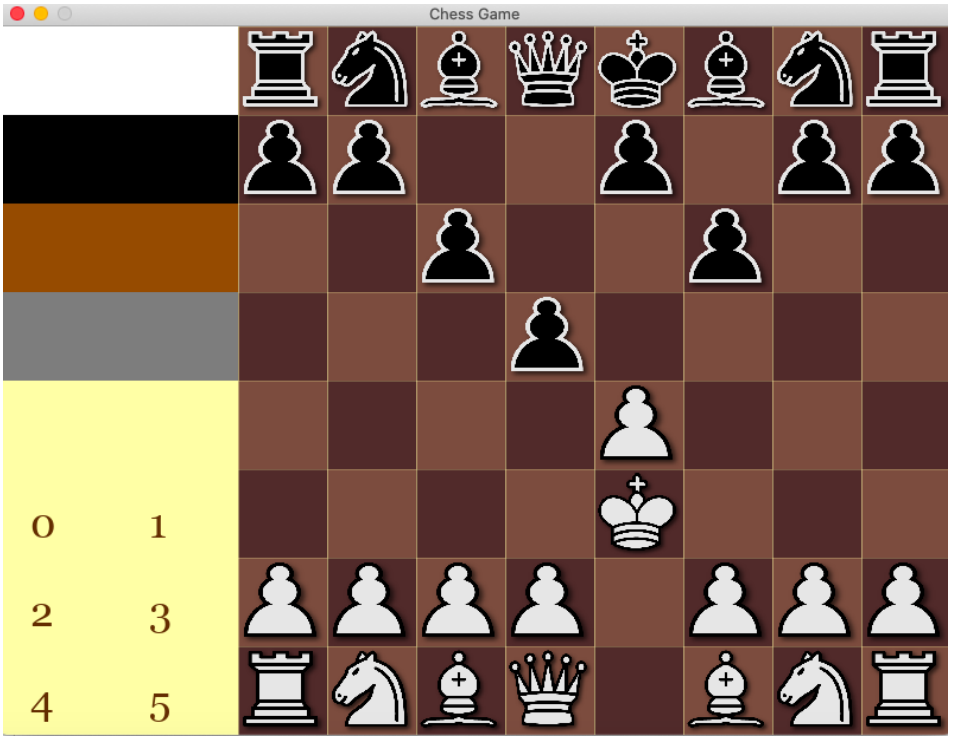
After queen is clicked to move in an “L” shape;



Invalid moves for the queen are disallowed.

15	getValidMoves()	Click on king to move one space right	Valid	One space right is valid for a king	Pass

Before;

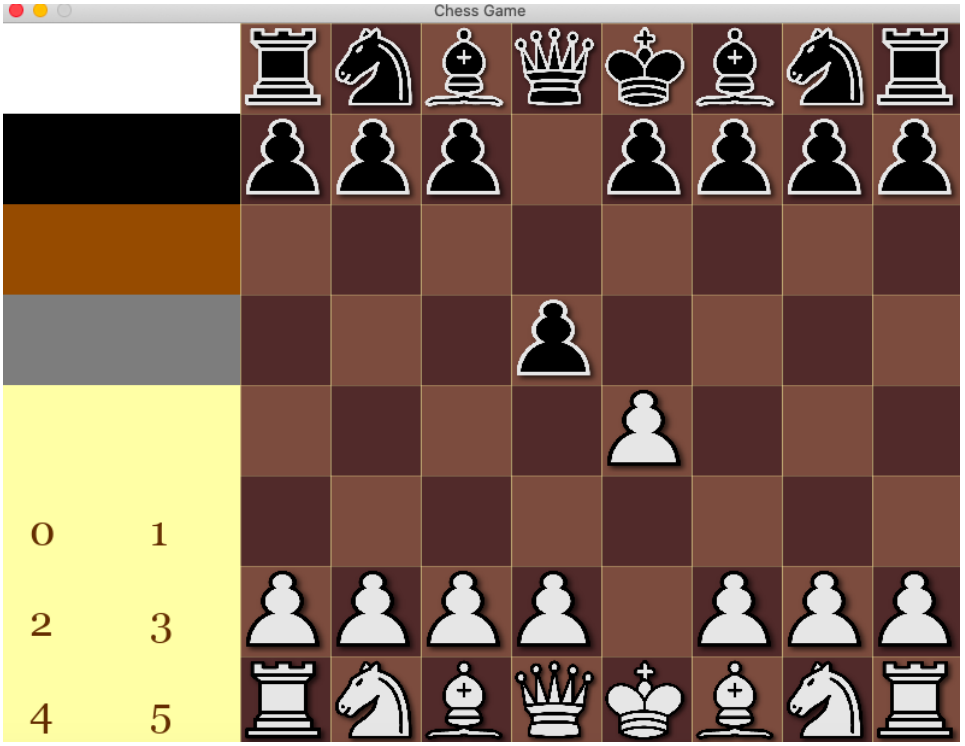


After king is clicked one space to the right;

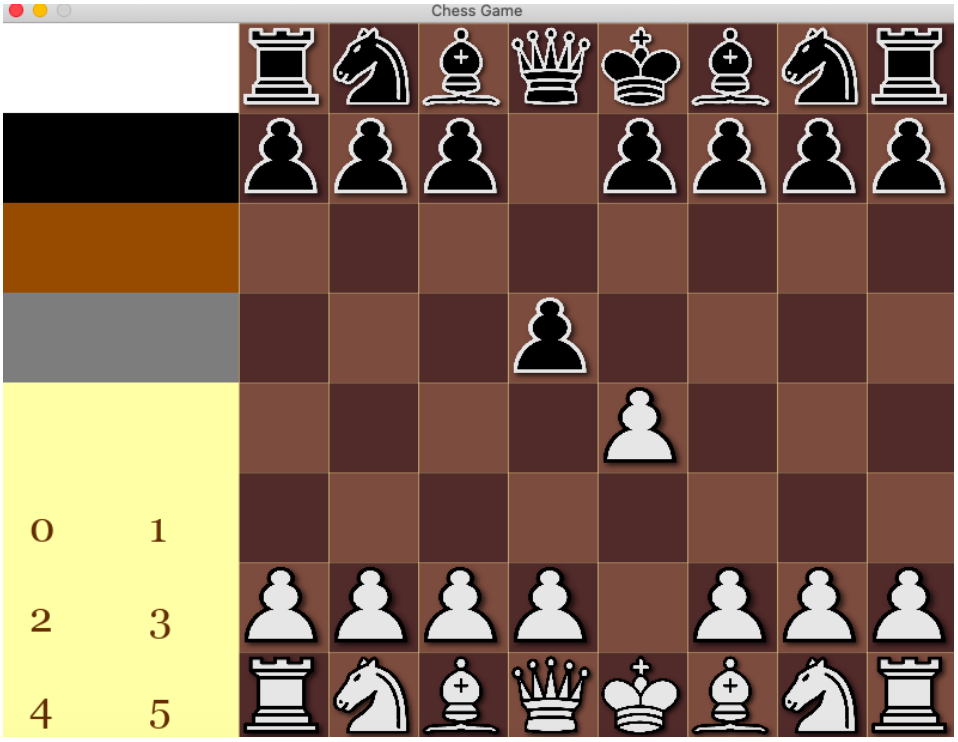
The king is able to move to valid positions.

16	getValidMoves()	Click on king to move two spaces up	Invalid	Two spaces up is not valid for a king	Pass

Before;



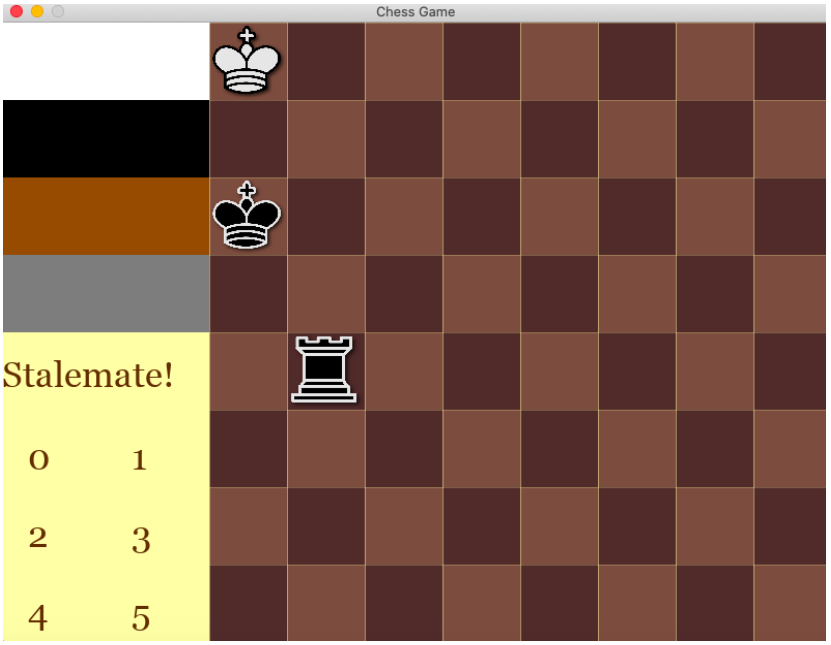
After king is clicked two spaces up;



Invalid moves for the king are disallowed.

17	isStalemate()	Board position where stalemate is true, and game should output that its stalemate	Valid	Player should be alerted when game ends in stalemate	Pass

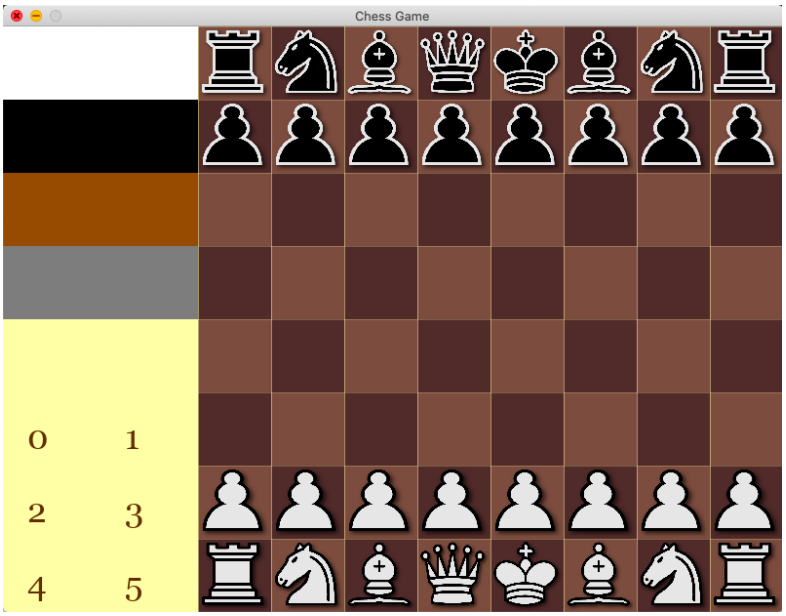
When the board position is stalemate;



The game outputs stalemate when the board position is stalemate, which is the correct game outcome.

18	isStalemate()	Board position where stalemate is false, and game should output that its stalemate	Invalid	Player shouldn't be told game ended in stalemate when board state is not in stalemate	Pass

When the board position is not stalemate;



The game doesn't output stalemate when the position is not in stalemate.

19	isCheckmate()	Board position where black is checkmated, and game should output that white won	Valid	Player should be told if white won by checkmate if white won by checkmate	Pass

When the board position is where black is checkmated;

The game outputs that white wins if the white checkmates black, which is the correct outcome.

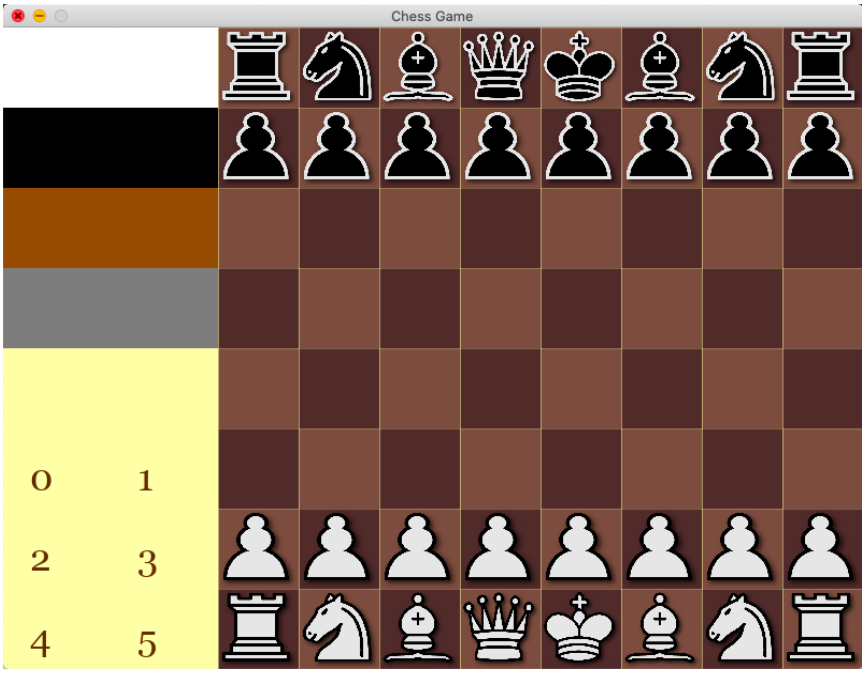
20	isCheckmate()	Board position where white is checkmated, and game should output that black won	Valid	Player should be told if black won by checkmate if black won by checkmate	Pass

When the board position is where white is checkmated;

The game outputs that black wins if the black checkmates white, which is the correct outcome.

21	isCheckmate()	Board position where no one is checkmated but game outputs checkmate has occurred	Invalid	Player shouldn't be told game ended in checkmate when board state is not in checkmate	Pass

When the board position is not in checkmate;

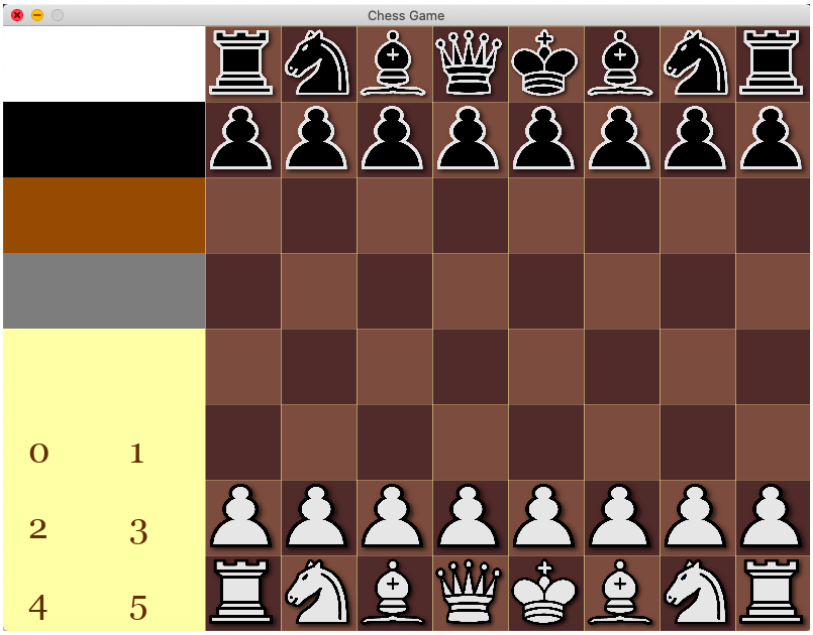


Game does not output checkmate when the board is not in checkmate.

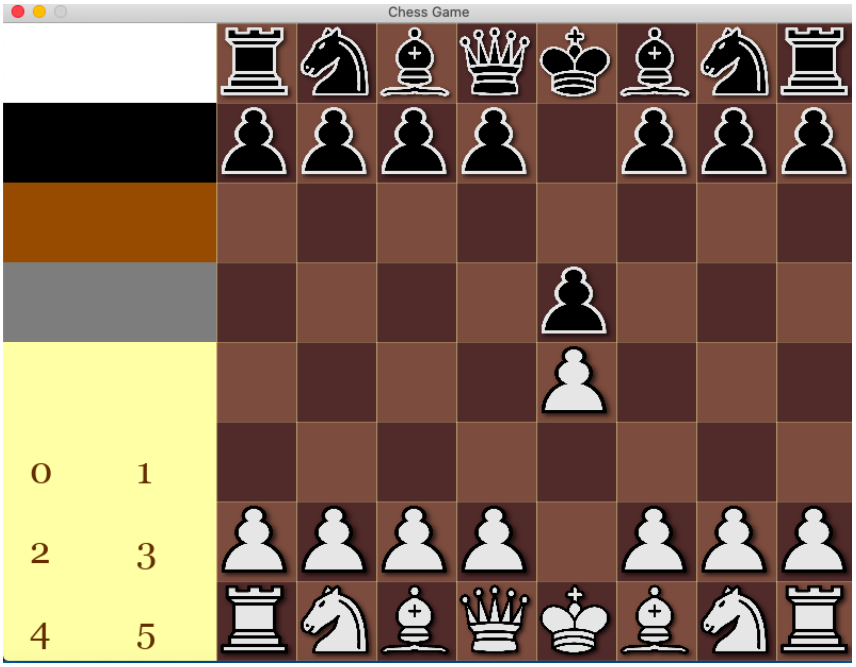
22	minimax()	Level 5 difficulty selected	Extreme	Program should be able to search at the maximum depth level of 5 if user wants it to	Pass

The maximum depth level allowed in the game is a 5. This is the maximum strain that the program will put on the computer. Setting the game to the maximum level will allow me to see how the program will react under the most extreme condition.

When a move is made with a depth level of 5, from the starting position;



After waiting 10 minutes and 14 seconds, the following board position appeared;



The program doesn't appear to have any problem searching 5 moves ahead, so this test passes. However, it does take a long time for the program to decide on a move, which is consistent with [redacted]'s comments during the second testing of the minimax algorithm. Unfortunately, I cannot do anything to fix this problem without an increase of computation power allowing for more than one computer to work together to search for the best move. I am limited by the fact that the program runs a singular computer.

Summary of Post Development Tests

All features to change the game settings work as intended and do not output any unexpected results. This is evident through test 1 to 8 of this section. The end user can therefore be confident that each button does what it is supposed to due to our validation testing.

All pieces correctly obey the rules of chess, evident from the test of the features during gameplay (test 5 to 16). The user can therefore play a game of chess based on the rules of the chess, as each piece can move correctly.

The board clicker works a little differently than expected as you are able to click and drag a piece to a new position, rather than only being able to click. This is probably because the pygame event handler I used thinks of dragging as a form of clicking on the screen. When dragged, the piece still goes to the correct position despite it not being an intended feature. I decided to leave it in the game as it will not affect any other part of the game and also gives the user the option to drag pieces to the correct position. The board clicker is not affected by the keys which I am happy about. The board clicker disallows the player from moving the opponents pieces which is not allowed in chess. (All of this is demonstrated in tests 1 to 4 of features during gameplay).

The game can end in only three ways; stalemate, white wins or black wins. Tests 17 to 21 demonstrate that the game outputs the correct outcomes only when those outcomes occur on the board, which is a success.

In the most extreme case, the user will want to play chess at the highest difficulty, being a depth level of 5. The program and computer are able to search for a move at that depth without any strain, however it took over 10 minutes to search for a move. This will require the end user to be patient when waiting for the computer to make a move which can be very irritating as they may want an immediate response. There isn't anything I can do about this though given the hardware requirements and timeline of the project. The test passes though as the program was able to find the move.

Overall most of the tests in this stage passed, which was mostly due to consistent and thorough testing during the development process.

Client Statements of Post Development Test

- [redacted]: "All the buttons do what they're supposed to do and it's clear which ones to press to do what you want. When I dragged the piece it moved to where I put it, but I don't think you coded it to do that. I don't really care about the dragging though as long as I can move the pieces by clicking. The game tells you clearly how the game ended, which is good. I especially like the ability to change the board colours in the game."

- [redacted]: "Every piece look s like it moves correctly and plays how classical chess should (be played), so that's really good. The buttons on the side clearly change the difficulty of the game so that I can play at different levels in one game, which I was excited about. However, the only thing is that the robot took a really long time to make a move when I put it on the hardest level."

Evaluating Success Criteria

The final developed solution is compared against each of the success criteria. This will tell me whether the game met its initial requirements.

#	Criteria	Sub-Criteria	Justification
1	Create a game screen		To provide a visual interface for the user

As soon as the game runs, a game screen is created and appears in front of the user. The user can see the chess board and other features helping them visualise and play the game. Test 2 in development shows the screen being created, test 9 shows the pieces and board appearing on the screen and test 30 shows the buttons appearing on the screen. This success criteria is met.

[redacted] "The screen is large enough for me to see the game being played, and I think seeing the pieces on the board is essential for any chess game with proper interface."

2	Being able to exit the game screen		Allows the player to stop playing the game

The user is able to exit the game window at any time that they are finished with the game. They can do this by pressing the red button on the top corner of the window. Test 2 in the development demonstrates the ability to do this. This criteria was met.

3	Menu window to change the game settings		Gives the user an clear to place to change the game settings more easily

There is a game menu, but it does not exist in its own window. This was due to a slight change in plan during the design to have the menu on the side of the screen, as shown in test 30 of development. Even though it is not in its own window, the menu is in a clear place for the user to change the settings easily. However putting the menu in its own window could potentially allow for the board size on the screen to be larger making it easier for the user to see the game being played. Therefore this criteria has been partially met.

[redacted]: "I really don't miss the menu window at all and wouldn't have thought of it if you'd not mentioned it. Thinking about it, it'd be nice to have another window for the game settings but it might've made changing difficulty a bit harder."

4	Ability to play as either colour with buttons	Play as white	Playing as either colour is a vital part in chess, as it affects who's turn it is first. The player should be able to choose if they want to play first or second
		Play as black	

Using the buttons in test 30, the user can decide whether or not they want to play as either white or black if they want to play first or second. This gives the user different gameplay options which can be easily changed with these buttons. The ability to change the player colour is shown in test 9 of development. This criteria has been met.

[redacted]: “This part of the game was way too important to miss out cause I use different strategies with white and black. I am happy with how this was implemented.”

5	Ability to change board colour with buttons	Change board colour to brown	Option to change board colour allows for the player to have a more personalised experience
		Change board colour to grey	

Similarly to the player colour, the board colour can be changed via the buttons in test 30, allowing the user to change it easily using the buttons based on their preference. The ability to change the board colour is demonstrated in test 7 of development. This criteria has been met.

[redacted]: “I like being able to personalise my gameplay experience based on the colour board I want to play on. Changing the colour is simple and it always goes to the colour I ask for.”

6	Click on the board to move a piece during the player’s turn		Using a clicker is the method to make a piece move so that a player can complete their turn

The board clicker caused a lot of issues during development (test 17), but as shown in test 18 it is able to click on a piece and move it to its corresponding position. Testing of the gameplay features (test 4) showed that you were not able to move your opponents pieces which is against the rules of chess, but it also revealed that you were also able to drag pieces from there original to new positions as the pygame event handler seems to considered clicking to be the same thing as dragging (test 2). There isn’t anything I can do to fix this without recreating the game and my stakeholders seem bothered by this. The criteria has therefore been partially met.

[redacted]: “I mean, as long as I can click the pieces to the right place I don’t really care if you can drag the pieces.”

7	Computer makes a move on its turn		The computer AI should make a move on its turn to continue with the game

This criteria is very important as you can’t play a full game of chess without it. As shown in test 26 and 29 of development, the program can do this for random and calculated moves. An issue that arose on post development testing is that it takes a long time for the program to search at a high depth level, which is unfortunately something I can’t fix in the limited time frame of the project. This criteria has been met.

Darshan: “Not gonna lie, it is kind of irating waiting for a move to happen for that long, but it always does come and the computer makes a good decision so the AI works fine.”

8	Have the option of six levels of user difficulty		Allows for a wide range of difficulties for the user to train against

The user is able to play against the six level difficulties shown in test 29 and 32 of development. The difficulty begins with random moves and increases as the program searches one further move ahead for the best move. This is the best way to create an increasing difficulty in each level. This criteria was met.

[redacted]: “The game gets progressively harder when you go up in levels. The levels get noticeably harder after the third level, after that I struggle to beat it. This makes it good for training depending on what level I want to play at at the time.”

9	Ability to change the computer difficulty during gameplay by with buttons		Allows the user to train against an easier or harder opponent in different parts of the game. This criteria must be met as this is the reason I am creating the game

This is the most important criteria as it is the reason I am creating the game, and I am happy to say that it was met as shown in test 32 of development. The buttons to change the difficulty are numbers which increase as the levels get harder, which assigns a numerical value for “easy” and “hard” levels so the user can intuitively choose the associating level they would like to play against.

[redacted]: “It's really easy to figure out which level to select based on the numbering system. Lowering the difficulty in the middle game really increases my confidence heading into an endgame and I feel that it overall improved my skill at the game.”

[redacted]: “I tried out a new opening called the *Russo Gambit* which I saw on youtube from my favourite (chess) streamer, and thanks to adjusting the difficulty I was able to experiment with it without any consequences. I plan on trying out other openings against it. One drawback though is that the AI doesn’t seem to understand opening theory too well as it plays very experimental lines.”

10	Should follow the rules of classical chess	Pawn makes valid moves	Each piece should follow the rules of chess to traverse/move on the board in their own ways.
		Knight makes valid moves	
		Bishop makes valid moves	

		Rook makes valid moves	
		Queen makes valid moves	
		King makes valid moves	

Each piece on a chess board can move differently from one another and interact differently. They must all follow the rules of classical chess for the entire game to follow the rules of classical chess.

- Pawns move one space in the forward direction, move two spaces forward on its first move, and capture diagonally forward (test 16 of development). When the pawn reaches the end of the board it promotes to a queen only as [redacted] requested (test 21)
- Knights move in an "L" shape and are able to move over other pieces. It moves two spaces in a direction and one space in a perpendicular direction (test 12).
- Bishops move diagonally on the board in north-east, north-west, south-east or south-west directions (test 14).
- Rooks move horizontally and vertically through the straights of a board in the north, south, east or west directions (test 13).
- Queens will move both like the bishop and rook. It will move in the north, south, east, west, north-east, north-west, south-east or south-west directions (test 15).
- The king will move one space in any direction (test 11). It also can perform a move known as *castling* where it moves to spaces to the right or left and a rook is placed next to it if both pieces are unmoved and there are no pieces between them (test 20). The king should not be able to move into any place where it can be attacked by an enemy piece (test 22 & 23).

Additional evidence that the pieces can only move to allowed positions is shown in the test of features during gameplay, 5 to 16.

This criteria was met.

[redacted]: "All pieces look like they move like they're supposed to."

11	Visualise and mark where pieces can go before moving them		Being able to visualise where pieces go can help novice users play the game correctly

Unfortunately, this feature was not added due to time constraints of the project and it being a low priority feature. This is not something requested by my stakeholders, but something I wanted to add as I thought it would be a cool feature. To implement this in the future, I will need to import a new marker image into the game and when a piece is clicked on, have a function to check the valid moves for that piece and place the image onto the corresponding squares. This feature may be useful for novice players who don't know the rules of chess, however my target audience would already know how to play chess and will want to learn to get better at the game. This criteria was not met.

[redacted]: "I mean, I forgot that this was supposed to be a feature till you brought it up just now."

12	Should output game result when the game has ended	Output that white won if white wins by checkmate	The player should know that the game has ended and how it has ended
		Output that black won if black wins by checkmate	
		Output that the game ended in stalemate/a draw is the game ends by stalemate	

The game can end in one of three ways, and should only output the correct way the game has ended.

- If black is checkmated, the game outputs that white wins only.
- If white is checkmated, the game outputs that black wins only.
- If the game ends in a draw, the game shows that it ended in stalemate only.

All results are outputted correctly in test 31 of development and test 17 to 21 of the testing of game features.

This criteria was met.

Maintenance

If the game is to be potentially distributed, maintenance and limitations must be addressed in order for the end user to play the game with minimal issues.

Future Maintenance

Myself and other developers may have to edit the code in the future in order to make improvements or fix bugs. Some of the comments in the source code may be a little bit vague for developers who are not familiar with chess like the one below.

```
# One squares forward
```

The comment is in reference to the pawn's ability to move one space forward, but if someone in the future is looking at the code and doesn't know the rules of chess, it may be confusing to them. This is why I edited the comments to be more specific and easier to understand for those people;

```
# The pawn can move one space forward
```

This makes it easier to add updates to the code as the functionality of everything is clearly labelled and explained for a person with very little knowledge of chess.

The feature of visualising where pieces go in success criteria 11 was not met. In a future update to the game, I am planning on implementing this feature to widen my target market to beginner players who don't know any of the rules of chess and want to learn about the game without needing a person to play against. To implement this feature I will have to import a new marker image, read through the list of valid moves for a piece when clicked, and place the image on its corresponding squares.

The AI of the program can further be improved by giving it more game awareness. Right now, it only accounts for the positioning and value of each piece to determine the best move. These are considered the two most important aspects of a good position in chess as stated by international chess master Levy Roszman, but there are also many other factors which the program can consider while evaluating a position such as keeping track of squares which are covered and seeing whether two of the same pieces are aligned creating a *battery* (a special chess tactic). Taking this into account will allow the AI to play more accurately and realistically, enhancing the gameplay experience.

A limitation of the game is that the user is restricted to only playing by the rules of classical chess, when in reality there are many other different variations of chess that can be played. A popular example of this is *Fischer Random* chess, where the positions of the pieces on the back straight are randomised and reflected on both sides. Another variant which the user may want to play is one which *en passant* is a valid move as they may feel that the moves enhances the game. As updates for the game come out, other chess variants will become available to the user who can then play against the AI for that particular variant. Options for variants and AI's for each variant will give the user more choice in how they want to play chess and also expands the target audience to recreational players, not just people who want to train to get better at classical chess.

To further personalise the user's experience, eventually I would like to give them more choice of board colours which they can choose from. The colours can be themed as well depending on the time of year or general interest. For instance, a red and white board can be released during christmas time or a board of shades of ocean blue can be released for those users with an interest in water sports. Additionally, the chess pieces can have corresponding skin colours which the user can choose from. The user can then mix and match their preferred board and pieces to find which ones they enjoy playing with the most.

If the game is to be distributed, then I would make a dedicated place for the user to check for updates. Success criteria 3 of creating a menu window was partially met. Before distribution, I would fully implement this feature and in order for there to be space to add a place where the user can check for updates. This provided the user with a clear place to check for updates as they come out.

Additionally if the game is distributed, it will most likely be released as open source software where people are able to collaborate and help slowly improve the game to make the AI smarter. However, I will have to licence the software to ensure that I can control who can modify and distribute my work.

For every update made to the code, I plan on making a copy of the original source code before editing and testing the new code. Saving all the code will make it easier to go back and keep track of all the features implemented in the game as they come along, and are a good reference for anyone in the future who makes an update to the code. This way all versions of the game will still exist.

Limitations and Approaches

The main limitation of the final solution is that it can take over 10 minutes to look at a depth level of 5. This is extremely inefficient and as [redacted] commented before it can be irating for the user to wait that long for the AI to respond with a move. This is because the program must search through a very large number of possibilities of moves. As the depth level increases, the problem increases exponentially (time complexity of $O(k^n)$). Ideally, a large number of machines will run simultaneously in order to speed up the problem through parallel processing, however there is a low chance of my end user having access to that type of hardware to run the code. I have two ideas which may help reduce the time taken;

- Use of an opening book: As spoken about in the analysis, an opening book is a database of chess openings given to computer chess programs so they know what moves to play at the beginning of a game. Many chess engines use them in order to enhance the gameplay experience. If I use an opening book for the program to decide on moves at the beginning of a game, it will help save time as the program can read a move directly from a database which will be much quicker than calculating moves. The drawback of this is that the computer can only do this in the first few moves then have to resort back to calculating the best move when the moves in the opening book end. Also, an opening book requires deep knowledge of the game to create, so in order to accomplish this I will probably have to consult a highly rated chess player, which I currently do not have access to. This will hopefully become an update at a later time, when I either can consult with a chess expert or if I learn enough about chess to create it myself.

- Using a weighted search algorithm: Using a weighted search algorithm with heuristics, such as the Monte Carlo search algorithm rather than a minimax algorithm, may speed up the program's decision making process a little quicker. In this way, the program can get rid of moves which are highly unlikely to be the "best move." Drawbacks in this is that in extreme cases, the "best move" may be disregarded entirely, and the valuation of the heuristic also requires a deep knowledge of chess to assign. However, the time complexity will be reduced by only a small amount and will barely be noticeable at a lower depth. If users are having a large problem with the time of moves being made, this will be available in a later update.

Right now the end product is restricted to only users who know the rules of chess, but many people might want to play the game but may not know how to play chess. In future versions, the game will hopefully include the option to play a game tutorial. In this tutorial, the user will be able to simulate each of the pieces on a board to see how they are allowed to move. By experimenting with each piece, the user can slowly learn the rules of chess by slowly learning the rules of each piece. Fully implementing success criteria 11 will also allow beginners to learn the rules of chess, as it can allow them to see exactly where each piece can move, providing a helpful aid during gameplay. The user should be able to activate and deactivate these features depending on how they want to play.

A features which could potentially be useful in the game to help people train is the ability to run a computer analysis of games By allowing the computer to run a game review, players will be able to see the key moves in the game which will tell them what they did well or what mistakes they made, hence allowing them to to improve their gameplay. To run a computer analysis of a game, I will have to do a lot more development of the AI used in the game in order to give it further understanding of the game beyond material and positioning of pieces, so that it can give the user accurate and helpful

feedback on their games. Therefore this will not probably be added until I feel confident in the AI's ability to give constructive feedback and fix the timing issue of the AI.

Along with a computer game review, users may also find it helpful to save previous games to look back upon. Studying your previous games can help a player understand their mistakes and adjust their strategies in order to become better chess players. Users may also want to save a game review so they have a reference to how the computer thinks they played. The problem with implementing this is that it will take up a lot of memory to store every game as the program will have to store every board state of every move of each saved game, and the user may also need more RAM to load up previous games which they want to look at. If this is to be added in the future, then each user must have a limited number of games which they are able to save to prevent them from making the program too heavy to run.

Another limitation is that the program is not written to be a proper piece of software and requires the user to run the python script every time they would like to play the game. The user may find it annoying to do this or not be technologically savvy enough to figure out how to run it. If the game is to be widely distributed, I may have to use a game engine such as Unity or Godot to eventually recreate the game in the form of a piece of software. By using a game engine, success criteria 6 can be fully met as I will be able to use a new event handler which will disable the dragging of pieces and only allow for pieces to get clicked to their positions. The reason which I cannot fix this currently is because I am unable to change the pygame event handler to only allow for clicking. Making the program into a software will make distribution a lot easier as a user cannot edit the code, and users will probably find it easier to run.

Finally, the user is restricted to only being able to play the game on their laptop or desktop computers when some people may enjoy playing chess on their mobile devices. This may cause those people to have a poor experience playing the game as they don't like playing on laptops or may deter them from playing the game entirely. To widen access to the game, I may later have to make a version of the game in the form of an app for IOS and android devices. To accomplish this, I will most likely create this version using Java or Objective-C, which are languages commonly used in application development because python (language which my project is currently in) won't run as an app on those devices.

Code Listings

main.py

```
import pygame
from board import Board
from AI import AI

pygame.init()

# Create Screen
WIDTH = 800
HEIGHT  = 600
clock = pygame.time.Clock()
screen = pygame.display.set_mode((WIDTH, HEIGHT))

pygame.display.set_caption("Chess Game")

# function to draw text to the screen
def draw_text(text, font, colour, x_pos, y_pos):
    image = font.render(text, True, colour)
    screen.blit(image, (x_pos, y_pos))

# Create font and text colour
FONT =  pygame.font.SysFont("georgia", 35)
TEXT_COLOUR = (100, 45, 0)

# Keeps game running
running = True

# Initial player colour white and board colour brown
b = Board("brown", "white")

b.setupBoard(None)

# Initial computer level is zero
comp = AI(0)


while running:

    # Draw and fill the screen
    screen.fill((255, 255, 165))
    b.drawBoard(screen)
```

```
    # Draw rectangles for buttons
    pygame.draw.rect(screen, (255, 255, 255), pygame.Rect(0, 0, 200, 75))
    pygame.draw.rect(screen, (0, 0, 0), pygame.Rect(0, 75, 200, 75))
    pygame.draw.rect(screen, (150, 75, 0), pygame.Rect(0, 150, 200, 75))
    pygame.draw.rect(screen, (125, 125, 125), pygame.Rect(0, 225, 200, 75))

    # Draw level difficulty buttons to screen
    draw_text("0", FONT, TEXT_COLOUR, 25, 400)
    draw_text("1", FONT, TEXT_COLOUR, 125, 400)
    draw_text("2", FONT, TEXT_COLOUR, 25, 475)
    draw_text("3", FONT, TEXT_COLOUR, 125, 475)
    draw_text("4", FONT, TEXT_COLOUR, 25, 550)
    draw_text("5", FONT, TEXT_COLOUR, 125, 550)

    clickedX, clickedY = pygame.mouse.get_pos()

    # Output winning colour if the board state is checkmate
    if b.isCheckmate():
        if b.getTurn() != "white":
            draw_text("White Wins!", FONT, TEXT_COLOUR, 0, 320)
        else:
            draw_text("Black Wins!", FONT, TEXT_COLOUR, 0, 320)

        b.setupBoard(None)

    # Output Stalemate if board state is stalemate
    if b.isStalemate():
        draw_text("Stalemate!", FONT, TEXT_COLOUR, 0, 320)

        b.setupBoard(None)

    for event in pygame.event.get():

        if event.type == pygame.QUIT:

            pygame.quit()
            exit()

        if (event.type == pygame.MOUSEBUTTONDOWN):

            if clickedX > 200:


                b.boardClicker(clickedX, clickedY)
```

```
        elif clickedY < 75:
            # Change Player Colour to white
            boardColour  = b.getBoardColour()
            b = Board(boardColour, "white")
            b.setupBoard(None)

        elif clickedY < 150:
            # Change Player Colour to black
            boardColour = b.getBoardColour()
            b = Board(boardColour, "black")
            b.setupBoard(None)

        elif clickedY < 225:
            # Change Board Colour to brown
            playerColour = b.getPlayerColour()
            b = Board("brown", playerColour)
            b.setupBoard(None)

        elif clickedY < 300:
            # Change Board Colour to gray
            playerColour = b.getPlayerColour()
            b = Board("gray", playerColour)
            b.setupBoard(None)

        elif clickedY < 450 and clickedX < 100:
            # Random moves
            comp = AI(0)

        elif clickedY < 450 and clickedX < 200:
            # Depth level 1
            comp = AI(1)

        elif clickedY < 525 and clickedX < 100:
            # Depth level 2
            comp = AI(2)

        elif clickedY < 525 and clickedX < 200:
            # Depth level 3
            comp = AI(3)

        elif clickedY < 600 and clickedX < 100:
            # Depth level 4
            comp = AI(4)
```

```
            elif clickedY < 600 and clickedX < 200:
                # Depth level 5
                comp = AI(5)

    # Computer's move
    if b.getPlayerColour() != b.getTurn():

        move = comp.bestMove(b)

        b.movePiece(move)



    pygame.display.update()
    clock.tick(24)
```

board.py

```
from pieces import *
import random
from square import Square

class Board():

    def __init__(self, board_colour, player_colour):
        # Board as an empty 8x8 matrix - 0 represent empty square
        self.board = [[0 for i in range(0, 8)] for j in range(0, 8)]

        self.previous_board = []

        self.squares = []

        # Board setup if player is white
        self.player_white_start_pos = [

            ['r', 'n', 'b', 'q', 'k', 'b', 'n', 'r'],
            ['p', 'p', 'p', 'p', 'p', 'p', 'p', 'p'],
            ['.', '.', '.', '.', '.', '.', '.', '.'],
            ['.', '.', '.', '.', '.', '.', '.', '.'],
            ['.', '.', '.', '.', '.', '.', '.', '.'],
            ['.', '.', '.', '.', '.', '.', '.', '.'],
            ['P', 'P', 'P', 'P', 'P', 'P', 'P', 'P'],
            ['R', 'N', 'B', 'Q', 'K', 'B', 'N', 'R']
        ]
        # Board setup is player is black
        self.player_black_start_pos = [
```

```
        ['R', 'N', 'B', 'K', 'Q', 'B', 'N', 'R'],
        ['P', 'P', 'P', 'P', 'P', 'P', 'P', 'P'],
        ['.', '.', '.', '.', '.', '.', '.', '.'],
        ['.', '.', '.', '.', '.', '.', '.', '.'],
        ['.', '.', '.', '.', '.', '.', '.', '.'],
        ['.', '.', '.', '.', '.', '.', '.', '.'],
        ['p', 'p', 'p', 'p', 'p', 'p', 'p', 'p'],
        ['r', 'n', 'b', 'k', 'q', 'b', 'n', 'r']
    ]
    # King-square table for points
    self.king_space_player = [
        [-3.0, -4.0, -4.0, -5.0, -5.0, -4.0, -4.0, -3.0],
        [-3.0, -4.0, -4.0, -5.0, -5.0, -4.0, -4.0, -3.0],
        [-3.0, -4.0, -4.0, -5.0, -5.0, -4.0, -4.0, -3.0],
        [-3.0, -4.0, -4.0, -5.0, -5.0, -4.0, -4.0, -3.0],
        [-2.0, -3.0, -3.0, -4.0, -4.0, -3.0, -3.0, -2.0],
        [-1.0, -2.0, -2.0, -2.0, -2.0, -2.0, -2.0, -1.0],
        [ 2.0,  2.0,  0.0,  0.0,  0.0,  0.0,  2.0,  2.0],
        [ 2.0,  3.0,  1.0,  0.0,  0.0,  1.0,  3.0,  2.0]
    ]

    king_space_compt = self.king_space_player.copy()
    king_space_compt.reverse()
    self.king_space_compt = king_space_compt

    # Queen-square table for points
    self.queen_space_player = [
        [-2.0, -1.0, -1.0, -0.5, -0.5, -1.0, -1.0, -2.0],
        [-1.0,  0.0,  0.0,  0.0,  0.0,  0.0,  0.0, -1.0],
        [-1.0,  0.0,  0.5,  0.5,  0.5,  0.5,  0.0, -1.0],
        [-0.5,  0.0,  0.5,  0.5,  0.5,  0.5,  0.0, -0.5],
        [ 0.0,  0.0,  0.5,  0.5,  0.5,  0.5,  0.0, -0.5],
        [-1.0,  0.5,  0.5,  0.5,  0.5,  0.5,  0.0, -1.0],
        [-1.0,  0.0,  0.5,  0.0,  0.0,  0.0,  0.0, -1.0],
        [-2.0, -1.0, -1.0, -0.5, -0.5, -1.0, -1.0, -2.0]
    ]

    queen_space_compt = self.queen_space_player.copy()
    queen_space_compt.reverse()
    self.queen_space_compt = queen_space_compt


    # Rook-square table for points
    self.rook_space_player = [
        [ 0.0,  0.0,  0.0,  0.0,  0.0,  0.0,  0.0,  0.0],
```

```
        [ 0.5,  1.0,  1.0,  1.0,  1.0,  1.0,  1.0,  0.5],
        [-0.5,  0.0,  0.0,  0.0,  0.0,  0.0,  0.0, -0.5],
        [-0.5,  0.0,  0.0,  0.0,  0.0,  0.0,  0.0, -0.5],
        [-0.5,  0.0,  0.0,  0.0,  0.0,  0.0,  0.0, -0.5],
        [-0.5,  0.0,  0.0,  0.0,  0.0,  0.0,  0.0, -0.5],
        [-0.5,  0.0,  0.0,  0.0,  0.0,  0.0,  0.0, -0.5],
        [0.0,   0.0,  0.0,  0.5,  0.5,  0.0,  0.0,  0.0]
    ]

    rook_space_compt = self.rook_space_player.copy()
    rook_space_compt.reverse()
    self.rook_space_compt = rook_space_compt

    # Bishop-square table
    self.bishop_space_player = [
        [-2.0, -1.0, -1.0, -1.0, -1.0, -1.0, -1.0, -2.0],
        [-1.0,  0.0,  0.0,  0.0,  0.0,  0.0,  0.0, -1.0],
        [-1.0,  0.0,  0.5,  1.0,  1.0,  0.5,  0.0, -1.0],
        [-1.0,  0.5,  0.5,  1.0,  1.0,  0.5,  0.5, -1.0],
        [-1.0,  0.0,  1.0,  1.0,  1.0,  1.0,  0.0, -1.0],
        [-1.0,  1.0,  1.0,  1.0,  1.0,  1.0,  1.0, -1.0],
        [-1.0,  0.5,  0.0,  0.0,  0.0,  0.0,  0.5, -1.0],
        [-2.0, -1.0, -1.0, -1.0, -1.0, -1.0, -1.0, -2.0]
    ]

    bishop_space_compt = self.bishop_space_player.copy()
    bishop_space_compt.reverse()
    self.bishop_space_compt = bishop_space_compt

    # Knight-square table for points
    self.knight_space_player = [
        [-5.0, -4.0, -3.0, -3.0, -3.0, -3.0, -4.0, -5.0],
        [-4.0, -2.0,  0.0,  0.0,  0.0,  0.0, -2.0, -4.0],
        [-3.0,  0.0,  1.0,  1.5,  1.5,  1.0,  0.0, -3.0],
        [-3.0,  0.5,  1.5,  2.0,  2.0,  1.5,  0.5, -3.0],
        [-3.0,  0.0,  1.5,  2.0,  2.0,  1.5,  0.0, -3.0],
        [-3.0,  0.5,  1.0,  1.5,  1.5,  1.0,  0.5, -3.0],
        [-4.0, -2.0,  0.0,  0.5,  0.5,  0.0, -2.0, -4.0],
        [-5.0, -4.0, -3.0, -3.0, -3.0, -3.0, -4.0, -5.0]
    ]

    knight_space_compt = self.knight_space_player.copy()
    knight_space_compt.reverse()
    self.knight_space_compt = knight_space_compt
```

```
        # Pawn-square table for points
        self.pawn_space_player = [
            [0.0,  0.0,  0.0,  0.0,  0.0,  0.0,  0.0,  0.0],
            [5.0,  5.0,  5.0,  5.0,  5.0,  5.0,  5.0,  5.0],
            [1.0,  1.0,  2.0,  3.0,  3.0,  2.0,  1.0,  1.0],
            [0.5,  0.5,  1.0,  2.5,  2.5,  1.0,  0.5,  0.5],
            [0.0,  0.0,  0.0,  2.0,  2.0,  0.0,  0.0,  0.0],
            [0.5, -0.5, -1.0,  0.0,  0.0, -1.0, -0.5,  0.5],
            [0.5,  1.0, 1.0,  -2.0, -2.0,  1.0,  1.0,  0.5],
            [0.0,  0.0,  0.0,  0.0,  0.0,  0.0,  0.0,  0.0]
        ]

        pawn_space_compt = self.pawn_space_player.copy()
        pawn_space_compt.reverse()
        self.pawn_space_compt = pawn_space_compt

        self.board_colour = board_colour

        self.player_colour = player_colour

        self.turn = 'white'

        self.moves = []

        self.piece_selected = None

    def setupBoard(self, other_board):

        square = None
        for i in range(0, 8):
            for j in range(0, 8):

                # Choose board template
                if other_board is None:
                    if self.player_colour == "white":
                        square = self.player_white_start_pos[i][j]
                    elif self.player_colour == "black":
                        square = self.player_black_start_pos[i][j]

                else:
                    square =  other_board[i][j]

                # Corresponding ascii value for each square
                if square == 'R':
                    self.board[i][j] = Rook("white", j, i)
```

```
            elif square == 'B':
                self.board[i][j] = Bishop("white", j, i)
            elif square == 'N':
                self.board[i][j] = Knight("white", j, i)
            elif square == 'Q':
                self.board[i][j] = Queen("white", j, i)
            elif square == 'K':
                self.board[i][j] = King("white", j, i)
            elif square == 'P':
                self.board[i][j] = Pawn("white", j, i)
            elif square == 'r':
                self.board[i][j] = Rook("black", j, i)
            elif square == 'b':
                self.board[i][j] = Bishop("black", j, i)
            elif square == 'n':
                self.board[i][j] = Knight("black", j, i)
            elif square == 'q':
                self.board[i][j] = Queen("black", j, i)
            elif square == 'k':
                self.board[i][j] = King("black", j, i)
            elif square == 'p':
                self.board[i][j] = Pawn("black", j, i)
            elif square == '.':
                self.board[i][j] = 0

    def evaluate(self):

        # Placeholder for each piece and evaluation starts at 0
        piece = None
        eval = 0

        for i in range(0, 8):
            for j in range(0, 8):

                if self.board[i][j] != 0:

                    piece = self.board[i][j]

                    if self.player_colour == "white":

                        # if playing as white, positive points for white pieces and
     positions on board
                        if piece.getColour() == "white":
                            if piece.getType() == "pawn":
                                eval += self.pawn_space_player[i][j]
```

```
                    elif piece.getType() == "bishop":
                        eval += self.bishop_space_player[i][j]
                    elif piece.getType() == "knight":
                        eval += self.knight_space_player[i][j]
                    elif piece.getType() == "rook":
                        eval += self.rook_space_player[i][j]
                    elif piece.getType() == "queen":
                        eval += self.queen_space_player[i][j]
                    elif piece.getType() == "king":
                        eval += self.king_space_player[i][j]

                    eval += piece.getValue()
                # if playing as white, negative points for black pieces and
positions on board, reverse board for calc
                else:
                    if piece.getType() == "pawn":
                        eval -= self.pawn_space_compt[i][j]
                    elif piece.getType() == "bishop":
                        eval -= self.bishop_space_compt[i][j]
                    elif piece.getType() == "knight":
                        eval -= self.knight_space_compt[i][j]
                    elif piece.getType() == "rook":
                        eval -= self.rook_space_compt[i][j]
                    elif piece.getType() == "queen":
                        eval -= self.queen_space_compt[i][j]
                    elif piece.getType() == "king":
                        eval -= self.king_space_compt[i][j]

                    eval -= piece.getValue()

            else:
                # if playing as black, negative points for white pieces and
positions on board, reverse board for calc
                if piece.getColour() == "white":
                    if piece.getType() == "pawn":
                        eval -= self.pawn_space_compt[i][j]
                    elif piece.getType() == "bishop":
                        eval -= self.bishop_space_compt[i][j]
                    elif piece.getType() == "knight":
                        eval -= self.knight_space_compt[i][j]
                    elif piece.getType() == "rook":
                        eval -= self.rook_space_compt[i][j]
                    elif piece.getType() == "queen":
                        eval -= self.queen_space_compt[i][j]
                    elif piece.getType() == "king":
```

```
                            eval -= self.king_space_compt[i][j]

                        eval -= piece.getValue()
                    # if playing as white, negative points for black pieces and
     positions on board
                    else:
                        if piece.getType() == "pawn":
                            eval += self.pawn_space_player[i][j]
                        elif piece.getType() == "bishop":
                            eval += self.bishop_space_player[i][j]
                        elif piece.getType() == "knight":
                            eval += self.knight_space_player[i][j]
                        elif piece.getType() == "rook":
                            eval += self.rook_space_player[i][j]
                        elif piece.getType() == "queen":
                            eval += self.queen_space_player[i][j]
                        elif piece.getType() == "king":
                            eval += self.king_space_player[i][j]

                        eval += piece.getValue()

    return eval

def boardClicker(self, clickedX, clickedY):

    self.getValidMoves()

    # Inverse function to get square coordinates
    x = (clickedX-200)//75
    y = (clickedY//75)

    can_move = False
    selectedMove = None

    if self.piece_selected is not None:
        for piece_move in self.piece_selected.getValidMoves(self):
            if (piece_move.getOPos() == self.piece_selected.getPos()) and
     (piece_move.getNPos() == (x, y)):

                for move in self.moves:
                    if (move.getOPos() == piece_move.getOPos()) and (move.getNPos() ==
     piece_move.getNPos()):

                        can_move = True
                        selectedMove = move
```

```
                        break

    else:
        for move in self.moves:
            if move.getOPos() == (x, y):
                can_move = True


    # Iterate through squares to get square with clicked coordinates
    for square in self.squares:
        if square.getPosition() == (x, y) and (can_move):

            # If there is a piece on that square
            if (square.getPieceOnSquare() is not None) and (self.player_colour ==
     self.turn):

                # Select a piece
                if (self.turn == square.getPieceOnSquare().getColour()) and
     (self.piece_selected is None):
                    self.piece_selected = square.getPieceOnSquare()
                    square.setPieceOnSquare(None)

                # Capture a piece
                elif (self.piece_selected is not None) and
     (square.getPieceOnSquare().getColour() != self.piece_selected.getColour()):
                    self.movePiece(selectedMove)
                    self.piece_selected = None

            # Move to empty square
            elif (self.player_colour == self.turn) and (self.piece_selected is not
     None):
                self.movePiece(selectedMove)
                self.piece_selected = None

def drawBoard(self, the_screen):

    # Temporary placeholder for each square
    square = None

    # Draw 64 squares to represent 8x8 board
    for i in range(0, 8):
        for j in range(0,8):
            # Square colour is dependent on the player's colour, board colour and
     coordinates
            square = Square(self.board_colour, (j, i), self.player_colour)
```

```
            # Blit square onto screen with formula
            the_screen.blit(square.getSquare(), ((75*i)+200, (75*j)))
            # Add square to square list
            self.squares.append(square)

    # Draw pieces to the screen by looping through the board
    for i in range(0, 8):
        for j in range(0, 8):
            # Check if a piece is on that square
            if self.board[i][j] != 0:
                # Blit piece onto screen with formula
                the_screen.blit(self.board[i][j].getImage(), ((75*j)+200, (75*i)))
                # Add piece onto corresponding square
                for s in self.squares:
                    if s.getPosition() == (j,i):
                        s.setPieceOnSquare(self.board[i][j])

def isInCheck(self):

    king_pos = None
    in_check = False
    moves = []

    # Find king position
    for i in range(0, 8):
        for j in range(0, 8):
            if self.board[i][j] != 0:
                if (self.board[i][j].getType() == "king") and
     (self.board[i][j].getColour() == self.turn):
                    king_pos = (j, i)

    # Flip turn to see opponents move
    piece = None

    for i in range(0, 8):
        for j in range(0, 8):
            if self.board[i][j]!= 0:
                piece = self.board[i][j]
                if piece.getColour() != self.turn:
                    moves = piece.getValidMoves(self)
                    # If endpoint for any move is the same position as king, then king
     is in check
                    for move in moves:
                        if move.getNPos() == king_pos:
                            in_check = True
```

```
        return in_check

    def isCheckmate(self):
        self.getValidMoves()

        # If no legal moves
        if (len(self.moves) == 0):
             # and is in check
             if (self.isInCheck() == True):
                # then checkmate
                return True
        # Otherwise false
        else:
            return False

    def isStalemate(self):
        self.getValidMoves()
        is_stalemate = False
        white_pieces = []
        black_pieces = []

        # Clauses for the insufficient material clause
        insufficient_material_scenerios = [
            (['king'], ['king']),
            (['king'], ['king', 'knight']),
            (['king'], ['bishop', 'king']),
            (['bishop', 'king'], ['king', 'knight']),
            (['bishop', 'king'], ['bishop', 'king']),
            (['king', 'knight'], ['king', 'knight'])
        ]

        # If no moves are possible and not in check, then stalemate is true
        if (len(self.moves) == 0):
             if (self.isInCheck() == False):
                is_stalemate = True

        # Get pieces on board
        piece = None
        for i in range(0, 8):
            for j in range(0, 8):
                piece = self.board[i][j]
                if piece != 0:
                    if piece.getColour() == "white":
                        white_pieces.append(piece.getType())
```

```
                else:
                    black_pieces.append(piece.getType())

    temp_w = white_pieces.copy()
    temp_w.sort()
    white_pieces = temp_w
    temp_b = black_pieces.copy()
    temp_b.sort()
    black_pieces = temp_b.copy()

    pieces = (white_pieces, black_pieces)

    # See if current board state matches insufficient material clauses
    for scenario in insufficient_material_scenerios:
        if (pieces == scenario) or (pieces == scenario[::-1]):
            is_stalemate = True


    return is_stalemate

def getMoves(self):

    # Gets all possible moves for player whose turn it is
    piece = None
    moves = []
    piece_moves = []
    self.moves = []

    for i in range(0, 8):
        for j in range(0, 8):
            piece = self.board[i][j]
            if piece != 0:
                # Only get moves if piece colour is the same as turn
                if piece.getColour() == self.turn:
                    # Add every move of that piece onto list
                    piece_moves = piece.getValidMoves(self)
                    for move in piece_moves:
                        moves.append(move)

    return moves

def getValidMoves(self):
    valid_moves = []
    moves = self.getMoves()
```

```
        for move in moves:

            # temporarily store positions
            piece_in_n_pos = self.board[move.getNPos()[1]][move.getNPos()[0]]
            piece_in_o_pos = self.board[move.getOPos()[1]][move.getOPos()[0]]

            # replace new position with new piece and old position with 0
            self.board[move.getNPos()[1]][move.getNPos()[0]] = piece_in_o_pos
            self.board[move.getOPos()[1]][move.getOPos()[0]] = 0

            # if board state is not in check add move to list
            if self.isInCheck() == False:
                valid_moves.append(move)

            # return board to original state
            self.board[move.getOPos()[1]][move.getOPos()[0]] = piece_in_o_pos
            self.board[move.getNPos()[1]][move.getNPos()[0]] = piece_in_n_pos

        self.moves = valid_moves

        return self.moves

    def saveBoard(self):
        # Create a blank board
        saved = [[0 for i in range(0, 8)] for j in range(0, 8)]

        piece = None

        # PLace pieces of current board onto saved board
        for i in range(0, 8):
            for j in range(0, 8):
                if self.board[i][j] != 0:
                    piece = self.board[i][j]
                    saved[i][j] = piece

        # Add saved board to list
        self.previous_board.append(saved)

    def movePiece(self, move):

        # Select piece and set space to blank
        piece = self.board[move.getOPos()[1]][move.getOPos()[0]]
        self.board[move.getOPos()[1]][move.getOPos()[0]] = 0

        # Move piece
```

```
        piece.setPos(move.getNPos())
        piece.setMovedTrue()
        self.board[move.getNPos()[1]][move.getNPos()[0]] = piece

        # Pawn promotion
        if (piece.getType() == "pawn") and ((piece.getPositionY() == 7) or
         (piece.getPositionY() == 0)):
            self.board[move.getNPos()[1]][move.getNPos()[0]] = Queen(piece.getColour(),
         piece.getPositionX(), piece.getPositionY())

        # If castle store, rook here
        castled_rook = None

        # Set square to blank
        for square in self.squares:
            if square.getPosition() == move.getOPos():
                square.setPieceOnSquare(None)
        # Set piece on square
        for square in self.squares:
            if square.getPosition() == move.getNPos():
                square.setPieceOnSquare(piece)


        # Check for castle and store rook
        if piece.getType() == "king":

            if move.getOPos() == (piece.getPositionX()-2, piece.getPositionY()):
                castled_rook = self.board[move.getNPos()[1]][7]

            elif move.getOPos() == (piece.getPositionX()+2, piece.getPositionY()):
                castled_rook = self.board[move.getNPos()[1]][0]

        # Set rook's square to blank if castled
        if castled_rook is not None:

            self.board[castled_rook.getPositionY()][castled_rook.getPositionX()] = 0

            for square in self.squares:
                if square.getPosition() == castled_rook.getPos():
                    square.setPieceOnSquare(None)


            # Place rook on new square if castled
            if castled_rook.getPos() == (7, move.getNPos()[1]):
                castled_rook.setPos((piece.getPositionX()-1, piece.getPositionY()))
```

```
                castled_rook.setMovedTrue()
                self.board[piece.getPositionY()][piece.getPositionX()-1] = castled_rook

            else:
                castled_rook.setPos((piece.getPositionX()+1, piece.getPositionY()))
                castled_rook.setMovedTrue()
                self.board[piece.getPositionY()][piece.getPositionX()+1] = castled_rook

            for square in self.squares:
                if square.getPosition() == castled_rook.getPos():
                    square.setPieceOnSquare(castled_rook)



        self.turn = "white" if self.turn == "black" else "black"

    def unmakeMove(self):
        # Replace board with previous board and flip turn back
        board = self.previous_board[-1]
        board = self.ascii(board)
        self.setupBoard(board)
        self.previous_board.pop()
        self.turn = "white" if self.turn == "black" else "black"

    def getBoard(self):
        return self.board

    def getPlayerColour(self):
        return self.player_colour

    def getTurn(self):
        return self.turn

    def getBoardColour(self):
        return self.board_colour

    def ascii(self, board):

        # Decrypt board to ascii characters to
        ascii = [['.' for i in range(0, 8)] for j in range(0, 8)]
        piece = None

        for i in range(0, 8):
            for j in range(0, 8):
                piece = board[i][j]
```

```
                if piece!= 0:
                    if piece.getType() == "pawn":
                        if piece.getColour() == "white":
                            ascii[i][j] = "P"
                        else:
                            ascii[i][j] = "p"
                    if piece.getType() == "knight":
                        if piece.getColour() == "white":
                            ascii[i][j] = "N"
                        else:
                            ascii[i][j] = "n"
                    if piece.getType() == "bishop":
                        if piece.getColour() == "white":
                            ascii[i][j] = "B"
                        else:
                            ascii[i][j] = "b"
                    if piece.getType() == "rook":
                        if piece.getColour() == "white":
                            ascii[i][j] = "R"
                        else:
                            ascii[i][j] = "r"
                    if piece.getType() == "queen":
                        if piece.getColour() == "white":
                            ascii[i][j] = "Q"
                        else:
                            ascii[i][j] = "q"
                    if piece.getType() == "king":
                        if piece.getColour() == "white":
                            ascii[i][j] = "K"
                        else:
                            ascii[i][j] = "k"
        return ascii
```

squares.py

```
import pygame

class Square():

    def __init__(self, board_colour, square_pos, player_colour):

        # Load and scale up squares
        square_brown_dark = pygame.image.load('/images/square brown
         dark_1x.png').convert_alpha()
        square_brown_dark = pygame.transform.scale(square_brown_dark, (75, 75))
```

```
        square_brown_light = pygame.image.load('/images/square brown
         light_1x.png').convert_alpha()
        square_brown_light = pygame.transform.scale(square_brown_light, (75, 75))
        square_gray_dark = pygame.image.load('/images/square gray dark
         _1x.png').convert_alpha()
        square_gray_dark = pygame.transform.scale(square_gray_dark, (75, 75))
        square_gray_light = pygame.image.load('/images/square gray light
         _1x.png').convert_alpha()
        square_gray_light = pygame.transform.scale(square_gray_light, (75, 75))

        # Square position
        self.x = square_pos[0]
        self.y = square_pos[1]
        self.pos = square_pos

        if ((self.x % 2 == self.y % 2) and (player_colour == 'white'))  or ((self.x % 2 !=
         self.y % 2) and (player_colour == 'black')):
            # Is light if sum of both coordinates are even and colour is white, or if sum is
         odd and colour is black
            if board_colour == "brown":
                self.square = square_brown_light
            elif board_colour == "gray":
                self.square = square_gray_light

        else:
            # Otherwise is dark
            if board_colour == "brown":
                self.square = square_brown_dark
            elif board_colour == "gray":
                self.square = square_gray_dark

        self.piece_on_square = None

    def getSquare(self):
        return self.square

    def getPosition(self):
        return self.pos

    def getPieceOnSquare(self):
        return self.piece_on_square

    def setPieceOnSquare(self, piece):

        self.piece_on_square = piece
```

pieces.py

```
import pygame
from move import Move

class Piece():

    def __init__(self, colour, positionX, positionY, pieceType, value):
        self.colour = colour
        self.positionX  = positionX
        self.positionY = positionY
        self.pieceType = pieceType
        self.value = value
        self.validMoves = []
        self.moved = False
        self.image = None

    def addMove(self, move):
        self.validMoves.append(move)

    def getColour(self):
        return self.colour

    def getType(self):
        return self.pieceType

    def getPositionX(self):
        return self.positionX

    def getPositionY(self):
        return self.positionY

    def getPos(self):
        return (self.positionX, self.positionY)

    def getValue(self):
        return self.value

    def getImage(self):
        return self.image

    def getMoved(self):
        return self.moved
```

```
def checkStaights(self, board):

    move = None

    # Get South
    for i in range(self.positionY+1, 8):
        if board.getBoard()[i][self.positionX] == 0:
            move = Move(self.getPos(), (self.positionX, i))
            self.addMove(move)
        elif board.getBoard()[i][self.positionX].getColour() != self.colour:
            move = Move(self.getPos(), (self.positionX, i))
            self.addMove(move)
            break
        else:
            break

    # Get East
    for i in range(self.positionX+1, 8):
        if board.getBoard()[self.positionY][i] == 0:
            move = Move(self.getPos(), (i, self.positionY))
            self.addMove(move)
        elif board.getBoard()[self.positionY][i].getColour() != self.colour:
            move = Move(self.getPos(), (i, self.positionY))
            self.addMove(move)
            break
        else:
            break

    # Get West
    for i in range(self.positionX-1, -1, -1):
        if board.getBoard()[self.positionY][i] == 0:
            move = Move(self.getPos(), (i, self.positionY))
            self.addMove(move)
        elif board.getBoard()[self.positionY][i].getColour()!= self.colour:
            move = Move(self.getPos(), (i, self.positionY))
            self.addMove(move)
            break
        else:
            break

    # Get North
    for i in range(self.positionY-1, -1, -1):
        if board.getBoard()[i][self.positionX] == 0:
            move = Move(self.getPos(), (self.positionX, i))
            self.addMove(move)
```

```
            elif board.getBoard()[i][self.positionX].getColour()!= self.colour:
                move = Move(self.getPos(), (self.positionX, i))
                self.addMove(move)
                break
            else:
                break

    def checkDiagonals(self, board):

        move = None

        # Get South-East
        for i in range(1,8):
            if self.positionX+i < 8 and self.positionY+i < 8:
                if board.getBoard()[self.positionY+i][self.positionX+i] == 0:
                    move = Move(self.getPos(), (self.positionX+i, self.positionY+i))
                    self.addMove(move)
                elif board.getBoard()[self.positionY+i][self.positionX+i].getColour() !=
         self.colour:
                    move = Move(self.getPos(), (self.positionX+i, self.positionY+i))
                    self.addMove(move)
                    break
                else:
                    break
            else:
                break

        # Get South-West
        for i in range(1,8):
            if self.positionX-i >= 0 and self.positionY+i < 8:
                if board.getBoard()[self.positionY+i][self.positionX-i] == 0:
                    move = Move(self.getPos(), (self.positionX-i, self.positionY+i))
                    self.addMove(move)
                elif board.getBoard()[self.positionY+i][self.positionX-i].getColour()!=
         self.colour:
                    move = Move(self.getPos(), (self.positionX-i, self.positionY+i))
                    self.addMove(move)
                    break
                else:
                    break
            else:
                break

        # Get North-East
        for i in range(1,8):
```

```
            if self.positionX+i < 8 and self.positionY-i >= 0:
                if board.getBoard()[self.positionY-i][self.positionX+i] == 0:
                    move = Move(self.getPos(), (self.positionX+i, self.positionY-i))
                    self.addMove(move)
                elif board.getBoard()[self.positionY-i][self.positionX+i].getColour()!=
        self.colour:
                    move = Move(self.getPos(), (self.positionX+i, self.positionY-i))
                    self.addMove(move)
                    break
                else:
                    break
            else:
                break

        # Get North-West
        for i in range(1,8):
            if self.positionX-i >= 0 and self.positionY-i >= 0:
                if board.getBoard()[self.positionY-i][self.positionX-i] == 0:
                    move = Move(self.getPos(), (self.positionX-i, self.positionY-i))
                    self.addMove(move)
                elif board.getBoard()[self.positionY-i][self.positionX-i].getColour()!=
        self.colour:
                    move = Move(self.getPos(), (self.positionX-i, self.positionY-i))
                    self.addMove(move)
                    break
                else:
                    break
            else:
                break

    def getValidMoves(self, board):
        return self.validMoves

    def setMovedTrue(self):
        self.moved = True

    def setPos(self, pos):
        self.positionX = pos[0]
        self.positionY = pos[1]


class Pawn(Piece):

    def __init__(self, colour, positionX, positionY):
        super().__init__(colour, positionX, positionY, "pawn", 10)
```

```
    if self.colour == "white":
        image = pygame.image.load('/images/w_pawn_1x.png')
        self.image = pygame.transform.scale(image, (75, 75))

    else:
        image = pygame.image.load('/images/b_pawn_1x.png')
        self.image = pygame.transform.scale(image, (75, 75))

    self.direction = None

def getValidMoves(self, board):

    self.validMoves = []

    move = None

    if board.getPlayerColour() == self.colour:
        self.direction = -1
    else:
        self.direction = 1

    # The pawn can move one space forward
    if board.getBoard()[self.positionY+self.direction][self.positionX] == 0:

        move =  Move((self.positionX, self.positionY), (self.positionX,
     self.positionY+self.direction))
        self.addMove(move)

    # The pawn can move two spaces forward when unmoved
    if (board.getBoard()[self.positionY+self.direction][self.positionX] == 0) and
     (board.getBoard()[self.positionY+2*self.direction][self.positionX] == 0) and
     (self.moved == False):
        move =  Move((self.positionX, self.positionY), (self.positionX,
     self.positionY+2*self.direction))
        self.addMove(move)

    # The pawn can capture diagonally to the right
    if ((self.positionX + 1) < 8):
        if board.getBoard()[self.positionY+self.direction][self.positionX+1]!= 0:
            if
     board.getBoard()[self.positionY+self.direction][self.positionX+1].getColour()!=
     self.colour:
                move = Move((self.positionX, self.positionY), (self.positionX+1,
     self.positionY+self.direction))
```

```
                self.addMove(move)

        # The pawn can capture diagonally to the left
        if ((self.positionX - 1) > 0):
            if board.getBoard()[self.positionY+self.direction][self.positionX-1]!= 0:
                if
         board.getBoard()[self.positionY+self.direction][self.positionX-1].getColour()!=
         self.colour:
                move = Move((self.positionX, self.positionY), (self.positionX-1,
         self.positionY+self.direction))
                self.addMove(move)

        return self.validMoves

class Knight(Piece):

    def __init__(self, colour, positionX, positionY):
        super().__init__(colour, positionX, positionY, "knight", 30)

        if self.colour == "white":
            image = pygame.image.load('/images/w_knight_1x.png')
            self.image = pygame.transform.scale(image, (75, 75))

        else:
            image = pygame.image.load('/images/b_knight_1x.png')
            self.image = pygame.transform.scale(image, (75, 75))

    def getValidMoves(self, board):

        self.validMoves = []

        move = None

        # Moves for the knight
        possible_moves = [
            (self.positionX+2, self.positionY+1),
            (self.positionX+2, self.positionY-1),
            (self.positionX-2, self.positionY+1),
            (self.positionX-2, self.positionY-1),
            (self.positionX+1, self.positionY+2),
            (self.positionX+1, self.positionY-2),
            (self.positionX-1, self.positionY+2),
            (self.positionX-1, self.positionY-2)
        ]
```

```
        for move in possible_moves:

            # If move on board
            if (move[0] < 8) and (move[1] < 8) and (move[0] >= 0) and (move[1] >= 0):
                # If move is on piece of opposite colour allow capture
                if board.getBoard()[move[1]][move[0]] != 0:
                    if board.getBoard()[move[1]][move[0]].getColour() != self.colour:
                        m = Move(self.getPos(), (move))
                        self.addMove(m)
                # Move to empty square
                else:
                    m = Move(self.getPos(), (move))
                    self.addMove(m)

        return self.validMoves

class Bishop(Piece):

    def __init__(self, colour, positionX, positionY):
        super().__init__(colour, positionX, positionY, "bishop", 30.01)

        if self.colour == "white":
            image = pygame.image.load('/images/w_bishop_1x.png')
            self.image = pygame.transform.scale(image, (75, 75))

        else:
            image = pygame.image.load('/images/b_bishop_1x.png')
            self.image = pygame.transform.scale(image, (75, 75))

    def getValidMoves(self, board):

        self.validMoves = []

        # Bishops move diagonally
        self.checkDiagonals(board)
        return self.validMoves

class Rook(Piece):

    def __init__(self, colour, positionX, positionY):
        super().__init__(colour, positionX, positionY, "rook", 50)

        if self.colour == "white":
            image = pygame.image.load('/images/w_rook_1x.png')
            self.image = pygame.transform.scale(image, (75, 75))
```

```
        else:
            image = pygame.image.load('/images/b_rook_1x.png')
            self.image = pygame.transform.scale(image, (75, 75))



    def getValidMoves(self, board):

        self.validMoves = []

        # Rook can move straight
        self.checkStaights(board)
        return self.validMoves

class Queen(Piece):

    def __init__(self, colour, positionX, positionY):
        super().__init__(colour, positionX, positionY, "queen", 90)

        if self.colour == "white":
            image = pygame.image.load('/images/w_queen_1x.png')
            self.image = pygame.transform.scale(image, (75, 75))

        else:
            image = pygame.image.load('/images/b_queen_1x.png')
            self.image = pygame.transform.scale(image, (75, 75))

    def getValidMoves(self, board):

        self.validMoves = []

        # The queen moves on straights and along diagonals
        self.checkStaights(board)
        self.checkDiagonals(board)

        return self.validMoves

class King(Piece):

    def __init__(self, colour, positionX, positionY):
        # value of king is winning the game so king value is infinity
        super().__init__(colour, positionX, positionY, "king", 999999)

        if self.colour == "white":
```

```
            image = pygame.image.load('/images/w_king_1x.png')
            self.image = pygame.transform.scale(image, (75, 75))

        else:
            image = pygame.image.load('/images/b_king_1x.png')
            self.image = pygame.transform.scale(image, (75, 75))

    def getValidMoves(self, board):

        move = None
        self.validMoves = []

        # Moves for the king
        possible_moves= [
            (self.positionX+1, self.positionY),
            (self.positionX+1, self.positionY+1),
            (self.positionX+1, self.positionY-1),
            (self.positionX, self.positionY+1),
            (self.positionX-1, self.positionY),
            (self.positionX-1, self.positionY+1),
            (self.positionX-1, self.positionY-1),
            (self.positionX, self.positionY-1)
        ]

        for move in possible_moves:

            # If move on board
            if (move[0] < 8) and (move[1] < 8) and (move[0] >= 0) and (move[1] >= 0):
                # If move is on piece of opposite colour allow capture
                if board.getBoard()[move[1]][move[0]] != 0:
                    if board.getBoard()[move[1]][move[0]].getColour() != self.colour:
                        m = Move(self.getPos(), (move))
                        self.addMove(m)
                # Move to empty square
                else:
                    m = Move(self.getPos(), (move))
                    self.addMove(m)

            # Check for castled moves
            self.canCastle(board)

        return self.validMoves

    def canCastle(self, board):
```

```
        # Variables for castle flags and pieces
        piece = None
        castleLeft = None
        castleRight = None
        leftRook = None
        rightRook = None

        # Find corresponding rook on the board
        for i in range(0, 8):
            for j in range(0, 8):
                piece = board.getBoard()[i][j]
                if piece != 0:

                    # Check if piece is a rook, if it is the same colour, if it hasn't moved
         and the king hasn't moved
                    if (piece.getColour() == self.colour) and (piece.getMoved() == False)
         and (piece.getType() == "rook") and (self.moved == False):

                        # Check which direction the king can castle based on the position of
         the rook
                        if piece.getPositionX() < self.positionX:
                            # Set left rook flag to true if left rook is found
                            castleLeft = True
                            leftRook = piece

                        if piece.getPositionX() > self.positionX:
                            # Set right rook flag to true if right rook is found
                            castleRight = True
                            rightRook = piece

        # Check if squares between left rook and king are  blank, else the king can't castle
        if castleLeft:
            for i in range(leftRook.getPositionX()+1, self.positionX):
                if board.getBoard()[self.positionY][i] != 0:
                    castleLeft = False

        # Check if squares between right rook and king are  blank, else the king can't
         castle
        if castleRight:
            for i in range(self.positionX+1, rightRook.getPositionX()):
                if board.getBoard()[self.positionY][i]!= 0:
                    castleRight = False

        # If flag for left rook is true, then add minus two spaces in x direction to moves
         list
```

```
        if castleLeft:
            self.addMove(Move(self.getPos(), (self.positionX-2, self.positionY)))

        # If flag for right rook is true, then add two spaces in x direction to moves list
        if castleRight:
            self.addMove(Move(self.getPos(), (self.positionX+2, self.positionY)))
```

move.py

```
class Move():

    def __init__(self, oPos, nPos):

        self.oPos = oPos
        self.nPos = nPos

    def getOPos(self):
        # Get start position
        return self.oPos

    def getNPos(self):
        # Get end position
        return self.nPos
```

AI.py

```
import random

class AI ():

    def __init__(self, depth):
        self.depth = depth

    def playRandomMove(self, board):

        # Choose random move and return it

        moves = board.getValidMoves()
```

```
        move = random.choice(moves)

        return move

    def minimax(self, depth, board, alpha=-100000000000, beta=100000000000):

        # Break is depth is 0 or game has ended
        if depth == 0 or board.isCheckmate() or board.isStalemate():
            return (-board.evaluate(), None)



        moves = board.getValidMoves()

        if board.turn != board.player_colour:
            best_move = None
            maxScore = -100000000000

            # Look at every move
            for move in moves:

                board.saveBoard()

                # Move a piece and call function again
                board.movePiece(move)

                score = self.minimax(depth - 1, board, alpha, beta)

                board.unmakeMove()

                # Calculate alpha
                alpha = max(alpha, score[0])

                # Break if maximising is greater than minimising
                if beta <= alpha:
                    break

                if score[0] >= maxScore:
                    maxScore = score[0]
                    best_move = move

            # Return best move for maximiser
            return (maxScore, best_move)

        else:
```

```
        best_move = None
        minScore = 100000000000

        for move in moves:

            board.saveBoard()

            # Move a piece and call function again
            board.movePiece(move)

            score = self.minimax(depth - 1, board, alpha, beta)

            board.unmakeMove()

            # Calculate beta
            beta = min(beta, score[0])

            if score[0] <= minScore:
                minScore = score[0]
                best_move = move

            # Break if maximising is greater than minimising
            if beta <= alpha:
                break

        # Return best move for minimiser
        return (minScore, best_move)

    def bestMove(self, board):
        # If depth is 0, then play random move
        if self.depth == 0:
            return self.playRandomMove(board)
        # Otherwise play minimax with depth
        else:
            depth = self.depth
            best_move = self.minimax(depth, board)[1]

        return best_move
```