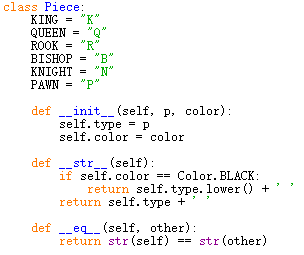
**COSC 3P71 Final Project**

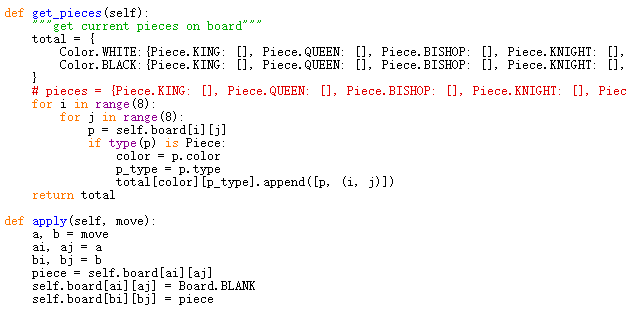
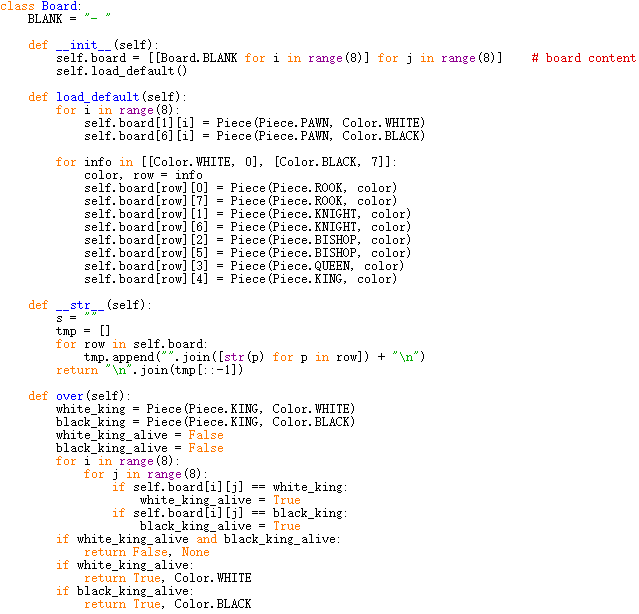
Ji Wang

[jw17tl@brocku.ca](mailto:jw17tl@brocku.ca)

**Piece Class**

There are six different classes of chess pieces in total. An object of a Piece class will have a color variable to indicate which side this piece is on.

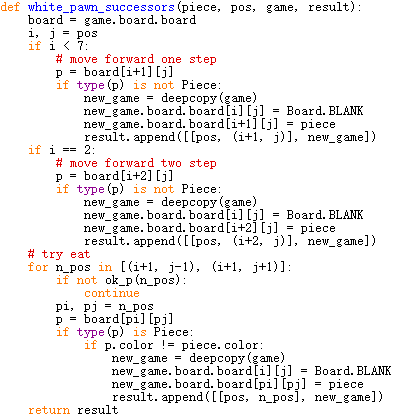


**The Board Setup**

**Pawn**

A pawn is implemented to move forward one square at a time except that the first move can be two squares (the rule of *en passant*). A pawn can capture the pieces at the diagonal position in its moving direction. (Apart from pawns, all other pieces can capture pieces at the destination of their moves) For example, if a white pawn is at c4 and a black pawn is at d5, they are both capable of capturing each other. Another special case of move for a pawn is that it can be promoted to one of rook, bishop, knight and queen if it reaches either row 1 or 8.

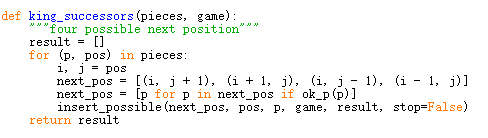
I implemented the white pawns and black pawns individually as follows:



Normally a pawn moves one square forward, but when its position is at the second row from its side (row 2 for white pawns or row 7 for black pawns), it can move 2 squares ahead.

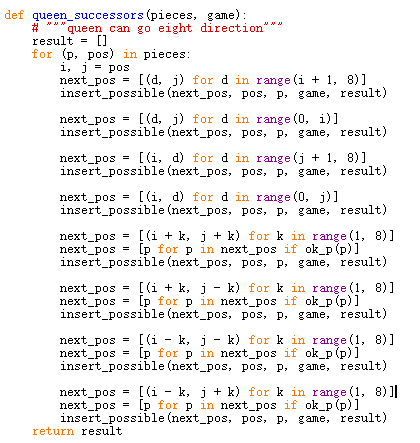
**King**

A king is implemented to only move one square at a time in all eight directions. Capturing the king will end the game and the capturer is the winner.



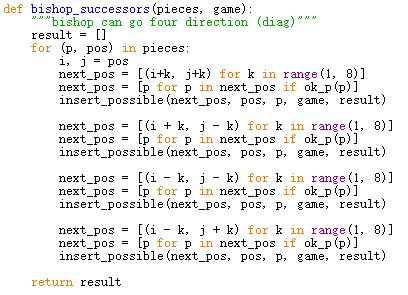
**Queen**

A queen can move arbitrary number of squares in all eight directions.



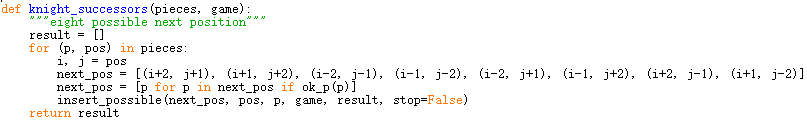
**Bishop**

A bishop can move arbitrary number of squares diagonally.



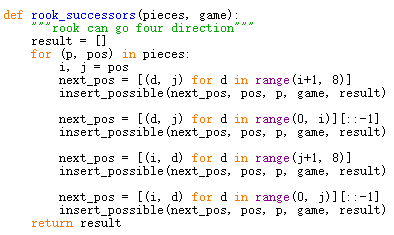
**Knight**

A knight can move two squares horizontally or vertically and then one square vertically or horizontally. For example, if a knight is at d4, it can move to b3, f3, b5, f5, c2, e2, c6 and e6.



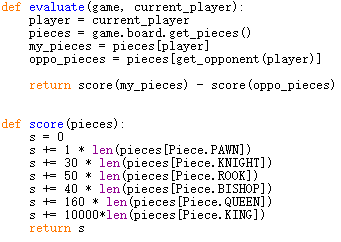
**Rook**

A rook can move arbitrary number of squares horizontally or vertically.



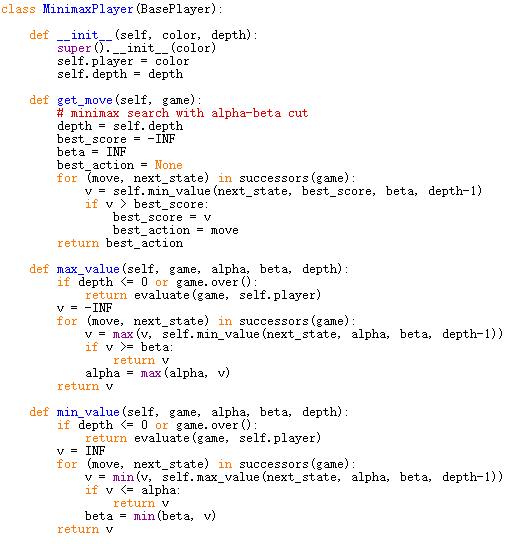
**Evaluation function and the game score**

The scores (or weights) of pieces are 1, 30, 40, 50, 160 and 10000 for pawn, knight, bishop, rook, queen and king respectively. The evaluation function will take these multiplied by the number of such piece on board and evaluate the score after next move or capture.



**Alpha-Beta Pruning**

Alpha-beta pruning is used to enhance the minimax game tree search. In the minimax algorithm, a maximizer and minimizer play against each other based on the score that the evaluation function gives. Maximizer wants higher score and minimizer wants lower score. One move of one piece will generate a possible board state. All possible board states are viewed as nodes in the game tree and both players take turns to determine what move to make comparing the scores of all nodes at their turn. The problem arises when the depth of the tree becomes large. In this case, number of possible board states is exponential to the number of moves, which is already large. It will take a long time for the AI to determine even the most trivial move. Alpha-beta pruning is the solution to this problem. It ignores the nodes that are unnecessary to examine. Alpha is the value of best choice so far along the maximizer’s path. Beta is the value of best choice so far along the maximizer’s path. Anything that is worse than alpha or beta is ignored. This can save considerable amount of time to make the AI look like that it can “think” fast.



**Configuration**

To modify the depth of the game tree search: simply change the integer value of depth.

To play the game against AI: comment out the MinimaxPlayer function call for black\_player using # symbol. The console will ask you to input move.

To watch two AI’s play against each other: comment out the HumanPlayer function call for black\_player.