Playing Chess in Python

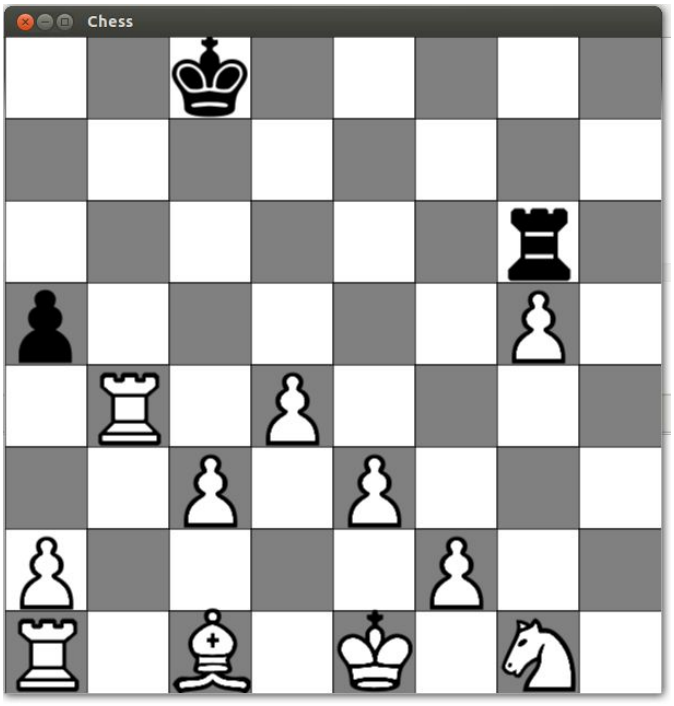
This document describes the chess.py Python program in Calico Python.

chess.py is a Chess Program with three different sample players. We explore them in some depth here to attempt to understand how each plays chess.

Game Play

The game in chess.py proceeds in the following manner:

1. The ChessBoard class generates all possible moves for each player, in turn.
2. Each player need only pick one in order to play a valid game of chess.
3. Players can reduce playing the game of chess into simply picking the **best** move at each turn.



You play a game using either the play function (for no graphics) or with gplay (with graphics). Either of these functions takes two arguments, a program that will play the black pieces, and a program that will play the white pieces. Sample programs are discussed below.

Concepts

Before we begin, there a few concepts we should explore:

ChessBoard class

The ChessBoard class keeps track of all of the items with playing a game of chess, expect for the state of the game (which is kept in the State class, discussed next). For example, the ChessBoard class is responsible for generating all possible moves, keeping track of the placement of all pieces, and making each move. The ChessBoard class has a two-dimensional 8 x 8 array called "board". Each of the 64 locations holds a single character (in the table below). Uppercase letters represent the white player's pieces, and lowercase letters represent the black player's pieces. Here is a representation of a ChessBoard.board into a game:

```
b moves r from (0, 0) to (0, 2)
Move: 44
  lower = b upper = W
  +-----------------+
  |               N | 0
  |   b k p       p | 1
  | r   n           | 2
  |                 | 3
  |       P         | 4
  | P   N     P   P | 5
  |   P P   P   P   | 6
  | R       K B   R | 7
  +-----------------+
    A B C D E F G H
    0 1 2 3 4 5 6 7
```

The first line "`b moves r from (0, 0) to (0, 2)`" indicates that black ('b') has moved the rook ('r') in the upper left-hand corner (0, 0) two places down to (0, 2) given as column, row. The representation above shows the state of the board after the move.

State class

The State class keeps track of whose turn it is, what color they are, special moves (eg, castling, en passant, etc.) and a history of past moves. This class can undo and redo moves, and keeps track of repeating states. The state.player is character indicating whose turn it is. The characters are either 'b' for black or 'w' for white.

The Player Program

A player program receives the board, the current state, and a list of possible moves. The board and state are instances of the above classes, ChessBoard and State, respectively. The each move is of the following form:

[FROM, PIECE, [TO, TO, TO...]]

That is, each move is a list composed of where the current piece is located (FROM in terms of X,Y), a character that represents the piece, and a list of all of the locations where that piece can move (the TO's, each in terms of X,Y). For example, the move:

[(3, 1), 'P', [(3, 2), (3,3)]]

describes a white pawn ('P') in position (3,1) which can move to either (3,2) or (3,3). X and Y range between 0 and 7 inclusive, and represent the square on the chess board.

Character Codes for White Pieces	**Character Codes for Black Pieces**
'P' - white pawn	'p' - black pawn
'R' - white rook	'r' - black rook
'N' - white knight	'n' - black knight
'B' - white bishop	'b' - black bishop
'Q' - white queen	'q' - black queen
'K' - white king	'k' - black king

In addition, a spot can also be ' ' (space character) to indicate that there is no piece in that spot.

Each player takes the board, a state, and set of possible moves and returns a fromPos (X, y) and a toPos (X, Y).

Here is a simple random player:

```
def randomPlayer1(board, state, moves):
    selection = random.choice(moves)
    fromPos = selection[0]
    toPos = random.choice(selection[2])
    return fromPos, toPos
```

This random player doesn't play very well. For one reason, the moves it selects are not uniformly random. The reason is that it first selects a piece, and then it select a possible move from that piece. To see why it is not very good, consider a set of moves such that there are two pieces. The first piece can make exactly one move. The second piece can make 16 moves. So there are 17 possible moves in total. However, they are not selected evenly (where each move would have a 1/17 chance of being selected). Instead, the first choice picks a piece, of which there is a 50/50 chance each piece is picked. If the first one is picked, then there is only one possible move, it will have be chosen. If the second piece is picked, then the code will select one of its 16 possible moves. But the first piece (and its only move) will be selected a full 50% of the time. Each of the other moves will be select 50% * 1/16 or about 3% of the time. Ironically, those pieces that fewer choices will be selected more than those moves of pieces with more choices.

Here is a better random player:

```
def randomPlayer2(board, state, moves):
    tofrom = []
    for move in moves:
        fromPos = move[0]
        for toPos  in move[2]:
            tofrom.append((fromPos, toPos))
    selection = random.choice(tofrom)
    fromPos = selection[0]
    toPos = selection[1]
    return fromPos, toPos
```

The only difference with this random player is that it separates all of the possible moves first, so that each move has an equal chance of being selected. Try playing randomPlayer1 against randomPlayer2 a number of times. Who wins the majority of the time?

Static Analysis/Board Evaluation

The next sample player takes each possible move, applies it to a temporary board and state, and then goes through the board, place by place, in order to compute an evaluation score for each resulting state. The moves are sorted by this score, and the best move is then returned:

```
def player1(board, state, moves):
    tofrom = []
    for move in moves:
        fromPos = move[0]
        for toPos  in move[2]:
            tofrom.append([fromPos, toPos, 0]) # place for score
    # Now, we go through and try each one on the board
```

```
    # and see what the results are:
    for move in tofrom:
        fromPos, toPos, score = move
        newboard = deepcopy(board)
        newstate = deepcopy(state)
        newboard.makeMove(newstate, fromPos, toPos)
        # go through board and return a score
        move[2] = staticAnalysis(newboard, newstate)
    tofrom.sort(key=lambda move: move[2]) # sort on score
    # return the highest from, to:
    return tofrom[-1][0], tofrom[-1][1]
```

The actual score is computed by the staticAnalysis function which is designed to evaluate the resulting board after each hypothesized move:

```
def staticAnalysis(board, state):
    score = random.random() # small random value
    for x in range(8):
        for y in range(8):
            color = board.getColor(x, y)
            piece = board.board[y][x].upper()
            if color == state.player:
                score += 1 # one of my pieces
            elif color == ' ':
                pass       # an empty place
            else:
                score -= 1 # the other player
    return score
```

This function examines each square of the board, compares the color ('b' or 'w') of the piece with your team color. If it matches, the score is increased by one. If it is an opponent piece, it subtracts one. Note that the score is initialized with a small random number. That makes it so that the same squares are not selected over and over in the case of a tie. Try changing the initial value of score to 0 to see what will happen.

The staticAnalysis function makes a much better player than either of the random players, but it still has major issues. How can you improve this static evaluation function?

Other useful Functions

You could incorporate many of the following functions into a Static Analysis function. Just make sure that state.player is 'b' or 'w' appropriately.

board.isCheck(state) - returns true if the current state's player's ('b' or 'w') King is in check.

board.isThreatened(state, x, y) - returns true if a location is threatened by the opponent.

board.hasAnyValidMoves(state) - returns true if state.player ('w' or 'b') has any valid moves.

board.getValid**NAME**Moves(state, position) - gets valid moves, where **NAME** is Queen, Rook, King, Bishop, Pawn, or Knight.

See the code for other additional functions.

Suggestions

1. Pawns get promoted when they get to the back row. Encourage them to get to the back row (eg, the closer they are to the opposite side, the better).
2. It is good to threaten opponent pieces.
3. It is good that your opponent's King has no valid moves.
4. Static analysis on the next move's state can only do so much good. It would be better if you could "look ahead" further and see the results of what your opponent could do, given what your proposed move did. And then see what you could do, then what they would do, etc. This is how real chess programs work. There are many algorithms for finding the best move by looking many moves ahead, such as minimax and alpha-beta pruning. You'll explore these ideas fully in Artificial Intelligence.
5. Your board evaluation function could change during the game. For example, you might use one evaluation function at the beginning, one in the middle, and another at the end. How can you tell where you are in a game?
6. There is a nice article on Chess Strategy at wikipedia: http://en.wikipedia.org/wiki/Chess_strategy