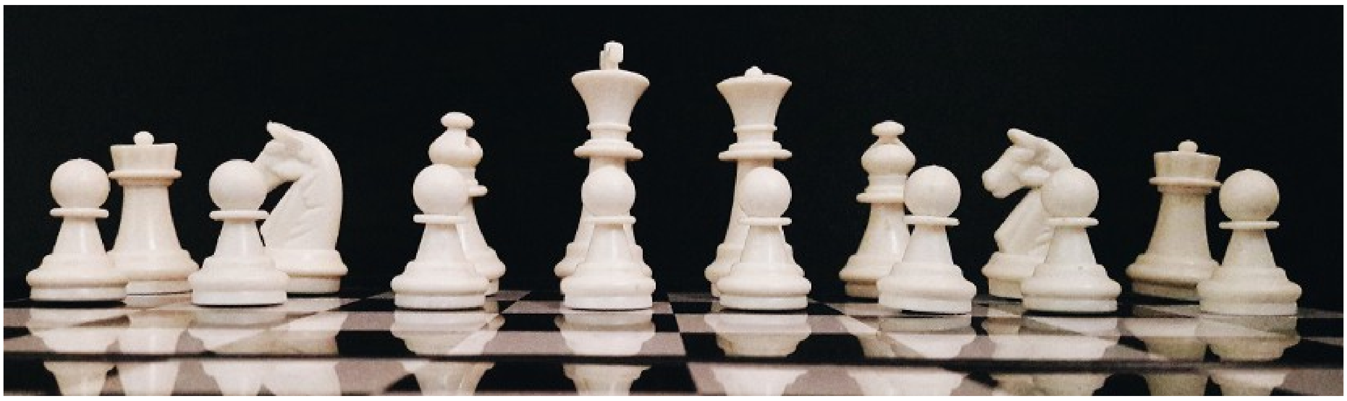
Andreas Stöckl Follow

Dec 24, 2018 · 5 min read · Listen

Writing a chess program in one day



The post is about how to write a simple computer chess program within one day with only a few lines of code. The program will be written in Python and contains all main parts of a chess engine. It will be the basis of refinements and enhancements which I will show in future postings.

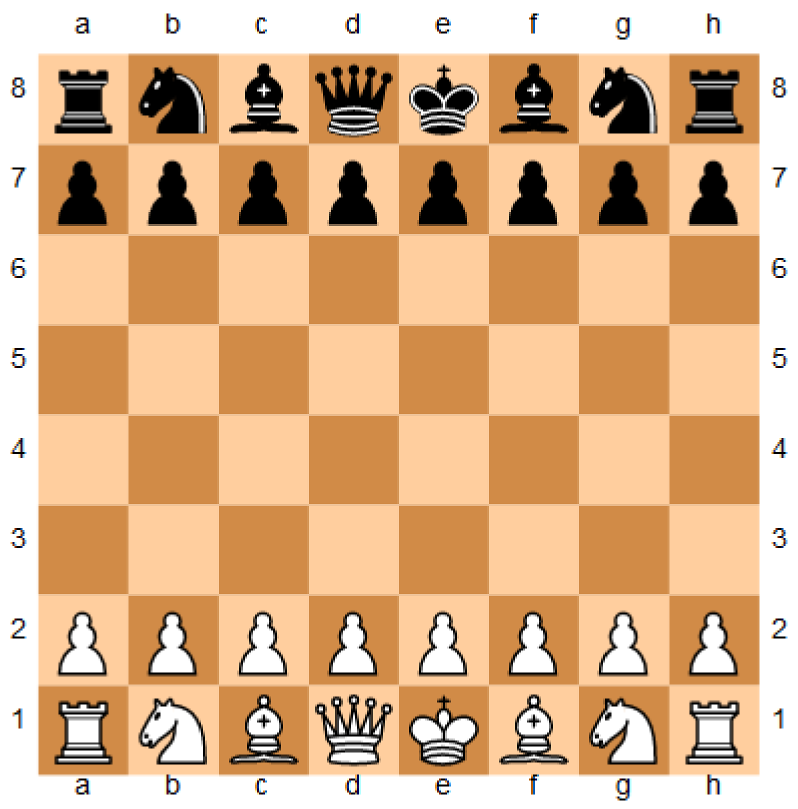
Every chess program has 3 important parts:
- The representation of the board
- The board evaluation
- The search

As a starting point, I use the Python package "chess" https://python-chess.readthedocs.io, which is a library for move generation, move validation, support for printing the board and more.

Board representation

```
import chess
import chess.svg
```

```
from IPython.display i

board = chess.Board()
SVG(chess.svg.board(bo
```



Display the board

The main component of the chess library is a “Board”-object which represents the pieces on the chess board, and has methods for move-generation and checking the status of the board (for example checking for mate). The object has a move-stack on which you can push and pop moves, for making a move and taking back a move.

An SVG component can be used to display a graphical representation of the board as above in a “Jupyter”-notebook.

Board evaluation

A position on the chess board can be evaluated if it is won by one side or it is a draw. If none of these conditions is satisfied we need an estimate of how likely it is that a player wins.

In this simple implementation, the estimate is done by two factors the “material” (pieces on board) and the positions of the pieces. For each sort of pieces, different values are calculated depending on the squares the pieces are located. This is done with so-called “piece-sq nentation.

```python
def evaluate_board():

    if board.is_checkmate():
        if board.turn:
            return -9999
        else:
            return 9999
    if board.is_stalemate():
        return 0
    if board.is_insufficient_material():
        return 0

    wp = len(board.pieces(chess.PAWN, chess.WHITE))
    bp = len(board.pieces(chess.PAWN, chess.BLACK))
    wn = len(board.pieces(chess.KNIGHT, chess.WHITE))
    bn = len(board.pieces(chess.KNIGHT, chess.BLACK))
    wb = len(board.pieces(chess.BISHOP, chess.WHITE))
    bb = len(board.pieces(chess.BISHOP, chess.BLACK))
    wr = len(board.pieces(chess.ROOK, chess.WHITE))
    br = len(board.pieces(chess.ROOK, chess.BLACK))
    wq = len(board.pieces(chess.QUEEN, chess.WHITE))
    bq = len(board.pieces(chess.QUEEN, chess.BLACK))

    material = 100*(wp-bp)+320*(wn-bn)+330*(wb-bb)+500*(wr-br)+900*(wq-bq)

    pawnsq = sum([pawntable[i] for i in board.pieces(chess.PAWN, chess.WHITE)])
    pawnsq= pawnsq + sum([-pawntable[chess.square_mirror(i)]
                                    for i in board.pieces(chess.PAWN, chess.BLACK)])
    knightsq = sum([knightstable[i] for i in board.pieces(chess.KNIGHT, chess.WHITE)])
    knightsq = knightsq + sum([-knightstable[chess.square_mirror(i)]
                                    for i in board.pieces(chess.KNIGHT, chess.BLACK)])
    bishopsq= sum([bishopstable[i] for i in board.pieces(chess.BISHOP, chess.WHITE)])
    bishopsq= bishopsq + sum([-bishopstable[chess.square_mirror(i)]
                                    for i in board.pieces(chess.BISHOP, chess.BLACK)])
    rooksq = sum([rookstable[i] for i in board.pieces(chess.ROOK, chess.WHITE)])
    rooksq = rooksq + sum([-rookstable[chess.square_mirror(i)]
                                    for i in board.pieces(chess.ROOK, chess.BLACK)])
    queensq = sum([queenstable[i] for i in board.pieces(chess.QUEEN, chess.WHITE)])
    queensq = queensq + sum([-queenstable[chess.square_mirror(i)]
                                    for i in board.pieces(chess.QUEEN, chess.BLACK)])
    kingsq = sum([kingstable[i] for i in board.pieces(chess.KING, chess.WHITE)])
    kingsq = kingsq + sum([-kingstable[chess.square_mirror(i)]
                                    for i in board.pieces(chess.KING, chess.BLACK)])
    eval = material +                                    + kingsq
    if board.turn:
        return eval
    else:
        return -eval
```

The function returns -9999 if white is mated, 9999 if black is mated and 0 for a draw. In all other situations, it returns an evaluation as the sum of the material and the sum of position values via piece-square tables. If black is in turn then the negative value is returned as needed by the negamax implementation of the search (see below).

Piece-square tables

I used the piece-square tables from https://www.chessprogramming.org/Simplified_Evaluation_Function

For each sort of piece, a different table is defined. If the value on a square is positive then the program tries to place a piece on that square if the value is negative it avoids to move to that square. The value of the whole position is calculated by summing over all pieces of both sides.

For pawns, the program is encouraged to advance the pawns. Additionally, we try to discourage the engine from leaving central pawns unmoved. Pawns on f2, g2 or c2 and b2 should not move tof3, etc.

Knights are simply encouraged to go to the center. Standing on the edge is a bad idea.

Bishops should avoid corners and borders.

Rooks should occupy the 7th rank and avoid a, h columns

Queens should avoid corners and borders and stay in the center.

Kings should stand behind the pawn shelter. This is only good for the opening and middle game phase. The endgame needs a different table. I will do this in a future enhancement of the program.

```
pawntable = [
 0,  0,  0,  0,  0,  0,  0,  0,
 5, 10, 10,-20,-20, 10
 5, -5,-10,  0,  0,-1
 0,  0,  0, 20, 20,
 5,  5, 10, 25, 25, 1
10, 10, 20, 30, 30, 20, 10, 10,
50, 50, 50, 50, 50, 50, 50, 50,
 0,  0,  0,  0,  0,  0,  0,  0]

knightstable = [
-50,-40,-30,-30,-30,-30,-40,-50,
-40,-20,  0,  5,  5,  0,-20,-40,
-30,  5, 10, 15, 15, 10,  5,-30,
-30,  0, 15, 20, 20, 15,  0,-30,
-30,  5, 15, 20, 20, 15,  5,-30,
-30,  0, 10, 15, 15, 10,  0,-30,
-40,-20,  0,  0,  0,  0,-20,-40,
-50,-40,-30,-30,-30,-30,-40,-50]

bishopstable = [
-20,-10,-10,-10,-10,-10,-10,-20,
-10,  5,  0,  0,  0,  0,  5,-10,
-10, 10, 10, 10, 10, 10, 10,-10,
-10,  0, 10, 10, 10, 10,  0,-10,
-10,  5,  5, 10, 10,  5,  5,-10,
-10,  0,  5, 10, 10,  5,  0,-10,
-10,  0,  0,  0,  0,  0,  0,-10,
-20,-10,-10,-10,-10,-10,-10,-20]

rookstable = [
  0,  0,  0,  5,  5,  0,  0,  0,
 -5,  0,  0,  0,  0,  0,  0, -5,
 -5,  0,  0,  0,  0,  0,  0, -5,
 -5,  0,  0,  0,  0,  0,  0, -5,
 -5,  0,  0,  0,  0,  0,  0, -5,
 -5,  0,  0,  0,  0,  0,  0, -5,
  5, 10, 10, 10, 10, 10, 10,  5,
 0,  0,  0,  0,  0,  0,  0,  0]

queenstable = [
-20,-10,-10, -5, -5,-10,-10,-20,
-10,  0,  0,  0,  0,  0,  0,-10,
-10,  5,  5,  5,  5,  5,  0,-10,
  0,  0,  5,  5,  5,  5,  0, -5,
 -5,  0,  5,  5,  5,  5,  0, -5,
-10,  0,  5,  5,  5,  5,  0,-10,
-10,  0,  0,  0,  0,  0,  0,-10,
-20,-10,-10, -5, -5,-10,-10,-20]
```

```
kingstable = [
 20, 30, 10,  0,  0, 
 20, 20,  0,  0,  0, 
-10,-20,-20,-20,-20,-
-20,-30,-30,-40,-40,-30,-30,-20,
-30,-40,-40,-50,-50,-40,-40,-30,
-30,-40,-40,-50,-50,-40,-40,-30,
-30,-40,-40,-50,-50,-40,-40,-30,
-30,-40,-40,-50,-50,-40,-40,-30]
```

Search

For lookahead, I use Depth-First search that starts at the root and explores up to a fixed depth along each branch before backtracking. The value of the position is calculated via Minimax and Alphabeta pruning is used with the Negamax implementation.

```
def alphabeta( alpha, beta, depthleft ):
    bestscore = -9999
    if( depthleft == 0 ):
        return quiesce( alpha, beta )
    for move in board.legal_moves:
        board.push(move)
        score = -alphabeta( -beta, -alpha, depthleft - 1 )
        board.pop()
        if( score >= beta ):
            return score
        if( score > bestscore ):
            bestscore = score
        if( score > alpha ):
            alpha = score
    return bestscore
```

At the maximum search depth, the search is extended by searching all capture moves, the so-called quiescence search to avoid the "Horizon Effect".

```
def quiesce( alpha, beta ):
    stand_pat = evaluate_board()
    if( stand_pat >= beta ):
        return beta
    if( alpha < stand_pat ):
        alpha = stand_pat

    for move in board
        if board.is_c
            board.pus
            score = -quiesce( -beta, -alpha )
            board.pop()

            if( score >= beta ):
                return beta
            if( score > alpha ):
                alpha = score
    return alpha
```

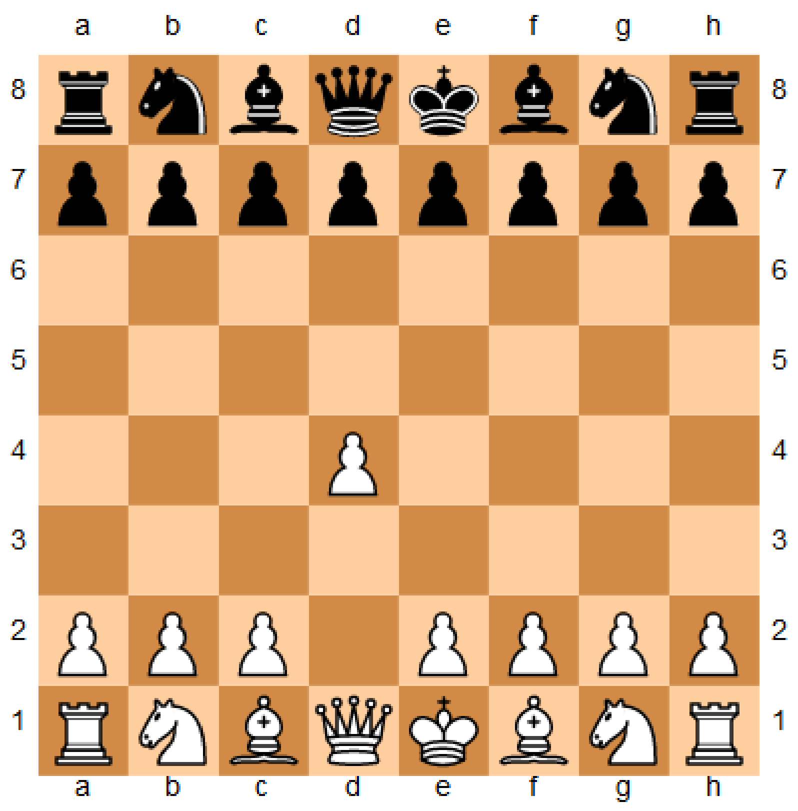
The function which implements the move selection in the root position consists of two parts. The first part tries to find a move in the opening book and gives it back. The “chess” library has a function to access opening books in the “Polyglot” format. I used the “bookfish” opening book which I downloaded from http://rebel13.nl/download/books.html. A random weighted move is selected from all possible moves of the book in this position.

The second part calculates the move if the book is empty. For each move in the position, the search (alphabeta) is conducted and the best move is chosen.

```
import chess.polyglot

def selectmove(depth):
    try:
        move = chess.polyglot.MemoryMappedReader("bookfish.bin").weighted_choice(board).move()
        movehistory.append(move)
        return move
    except:
        bestMove = chess.Move.null()
        bestValue = -99999
        alpha = -100000
        beta = 100000
        for move in board.legal_moves:
            board.push(move)
            boardValue = -alphabeta(-beta, -alpha, depth-1)
            if boardValue > bestValue:
                bestValue = boardValue;
                bestMove = move
            if( boardValue > alpha ):
                alpha = boardValue
            board.pop()
        movehistory.a
        return bestMo
```

To play against the program inside a Jupyter notebook you can evaluate the following cell to compute a computer move (with a search depth of 3), and display the board.

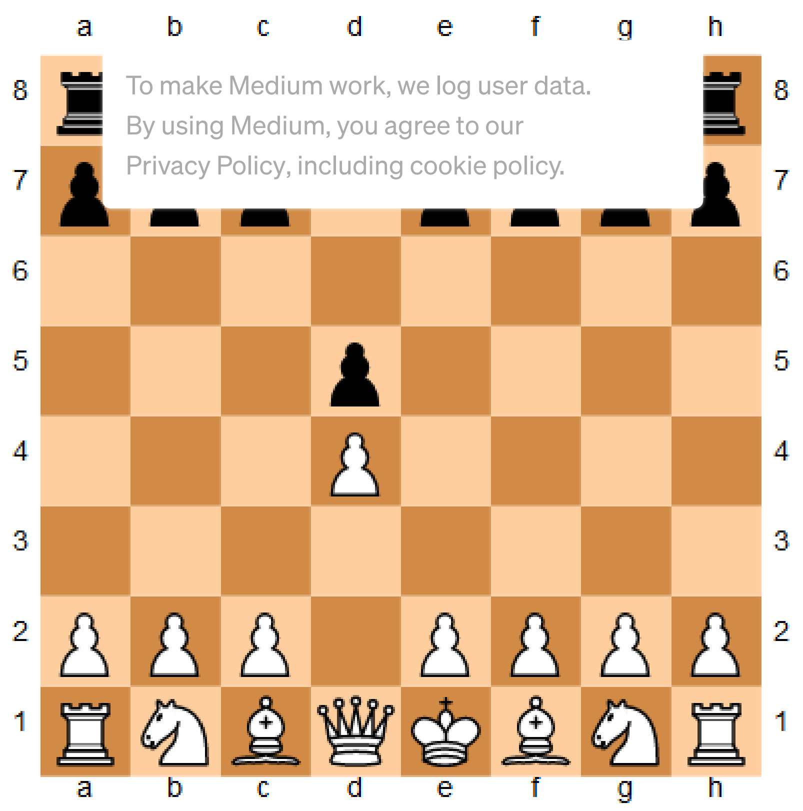
```
movehistory =[]
board = chess.Board()
mov = selectmove(3)
board.push(mov)
SVG(chess.svg.board(board=board,size=400))
```



First move

To make a human move you can use:

```
board.push_san("d5")
SVG(chess.svg.board(board=board,size=400))
```


To make Medium work, we log user data.
By using Medium, you agree to our
Privacy Policy, including cookie policy.

Human move

Match against Stockfish

Stockfish https://stockfishchess.org/ is one of the strongest chess-engines in these days. I use it to test my program with a search depth of 3.

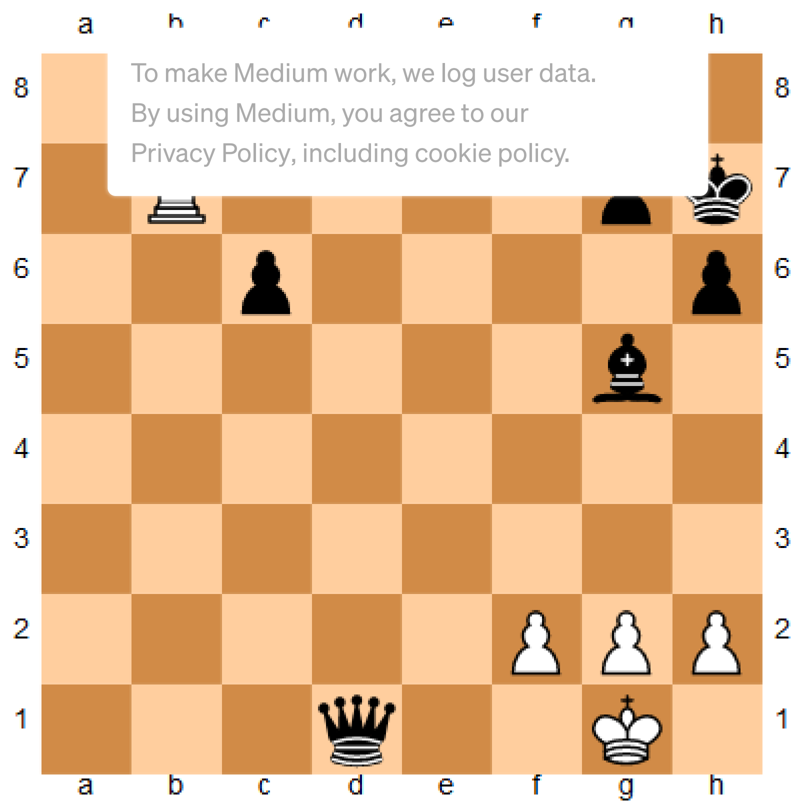
After loading the Stockfish engine with the UCI component of the library I wrote some code to collect all the moves of the game in a list ("movehistory") and write them to a portable game notation file.

At the end of the game, the final board position is displayed.

```
import chess.pgn
import datetime
import chess.uci

engine = chess.uci.popen_engine("C:/Users/p20133/Documents/pychess/stockfish/Windows/stockfish_
engine.uci()
engine.name

movehistory =[]
game = chess.pgn.Game()
game.headers["Event"] = "Example"
game.headers["Site"] = "Linz"
game.headers["Date"] = str(datetime.datetime.now().date())
game.headers["Round"] = 1
game.headers["White"] = "MyChess"
game.headers["Black"]
board = chess.Board()
while not board.is_ga
    if board.turn:
        move = selectmove(3)
        board.push(move)
    else:
        engine.position(board)
        move = engine.go(movetime=1000).bestmove
        movehistory.append(move)
        board.push(move)

game.add_line(movehistory)
game.headers["Result"] = str(board.result(claim_draw=True))
print(game)
print(game, file=open("test.pgn", "w"), end="\n\n")

SVG(chess.svg.board(board=board,size=400))
```

[Event “Example”]
[Site “Linz”]
[Date “2018–12–24”]
[Round “1”]
[White “MyChess”]
[Black “Stockfish9”]
[Result “0–1”]

1. d4 d5 2. c4 e6 3. Nc3 Nf6 4. Nf3 Bb4 5. cxd5 exd5 6. Bg5 O-O 7. e3 h6 8. Bh4 Bf5 9. Bd3 Bxd3 10. Qxd3 Nbd7 11. O-O c6 12. a3 Be7 13. Bxf6 Nxf6 14. e4 Nxe4 15. Nxe4 dxe4 16. Qxe4 Bf6 17. Ne5 Re8 18. Rfe1 Qb6 19. b4 Rad8 20. Rad1 a5 21. bxa5 Qxa5 22. Rb1 Qa4 23. Rb4 Qxa3 24. Rxb7 Qa4 25. Qb1 Qxd4 26. Nxf7 Rxe1+ 27. Qxe1 Rd7 28. Qe8+ Kh7 29. Ng5+ Bxg5 30. Qxd7 Qa1+ 31. Qd1 Qxd1# 0–1



Endposition

In the opening phase, both programs used the opening books. Considering the limited amount of knowledge of my program it played some solid moves. The end of the game showed the negative effects of the limited search depth of my engine.

There are a lot of things that can be done to improve the strength of the program. For example a more sophisticated evaluation function or a better search procedure. I will cover this in future posts.

The next post:

https://medium.com/@andreasstckl/an-incremental-evaluation-function-and-a-testsuite-for-computer-chess-6fde22aac137

The Jupyter notebooks can be found at https://github.com/astoeckl/mediumchess/