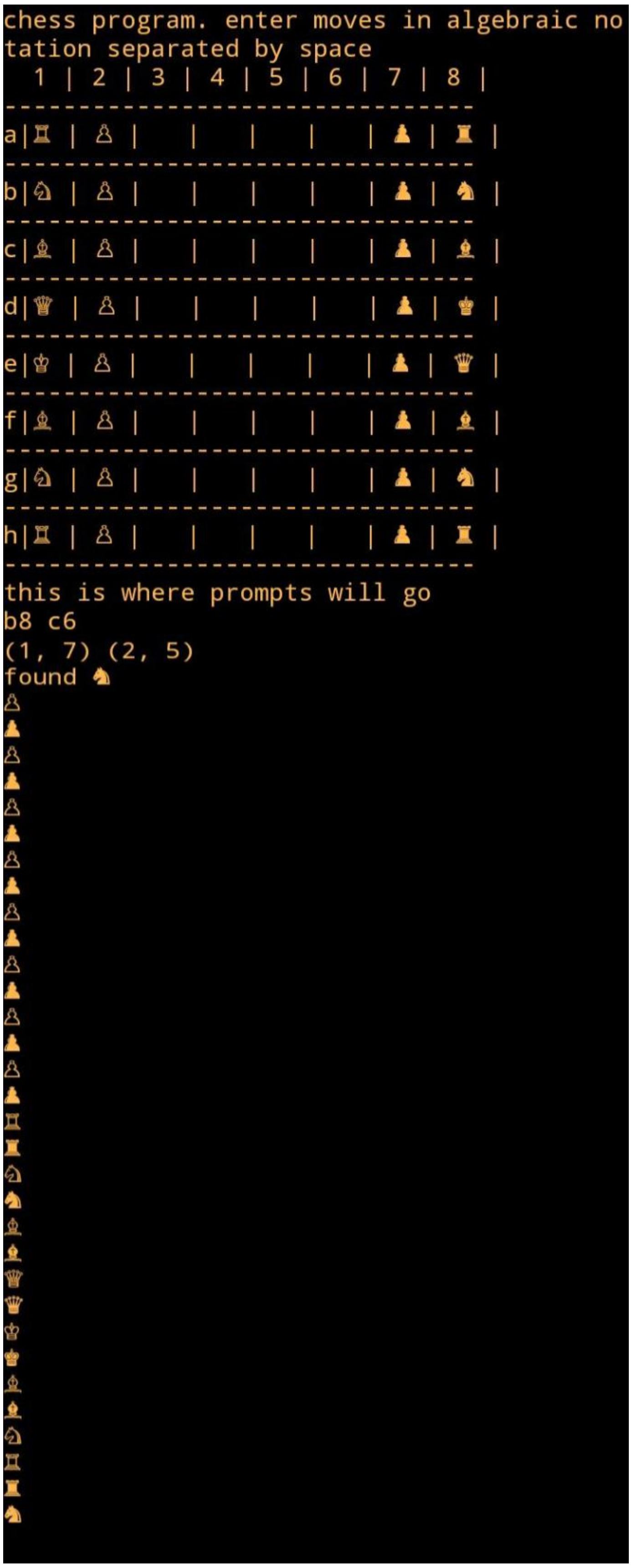
CHESS GAME CODING

```
"""CONVENTIONS:
positions are done row-column from the bottom left and are both
numbers. This corresponds to the alpha-number system in traditional
chess while being computationally useful. they are specified as
tuples
"""
import itertools
WHITE = "white"
BLACK = "black"
class Game:
    #ive decided since the number of pieces is capped but the type
of pieces is not (pawn transformations), I've already coded much of
the modularity to support just using a dictionary of pieces
    def __init__(self):
        self.playersturn = BLACK
        self.message = "this is where prompts will go"
        self.gameboard = {}
        self.placePieces()
        print("chess program. enter moves in algebraic notation
separated by space")
        self.main()

    def placePieces(self):

        for i in range(0,8):
            self.gameboard[(i,1)] =
Pawn(WHITE,uniDict[WHITE][Pawn],1)
            self.gameboard[(i,6)] =
Pawn(BLACK,uniDict[BLACK][Pawn],-1)

        placers = [Rook,Knight,Bishop,Queen,King,Bishop,Knight,Rook]

        for i in range(0,8):
            self.gameboard[(i,0)] =
placers[i](WHITE,uniDict[WHITE][placers[i]])
            self.gameboard[((7-i),7)] =
placers[i](BLACK,uniDict[BLACK][placers[i]])
        placers.reverse()


    def main(self):

        while True:
            self.printBoard()
            print(self.message)
            self.message = ""
            startpos,endpos = self.parseInput()
            try:
                target = self.gameboard[startpos]
            except:
                self.message = "could not find piece; index probably
out of range"
                target = None

            if target:
                print("found "+str(target))
                if target.Color != self.playersturn:
                    self.message = "you aren't allowed to move that
piece this turn"
                    continue
                if
target.isValid(startpos,endpos,target.Color,self.gameboard):
                    self.message = "that is a valid move"
                    self.gameboard[endpos] =
self.gameboard[startpos]
                    del self.gameboard[startpos]
                    self.isCheck()
                    if self.playersturn == BLACK:
                        self.playersturn = WHITE
                    else : self.playersturn = BLACK
                else :
                    self.message = "invalid move" +
str(target.availableMoves(startpos[0],startpos[1],self.gameboard))
                    print(target.availableMoves(startpos[0],startpos
[1],self.gameboard))
            else : self.message = "there is no piece in that space"

    def isCheck(self):
        #ascertain where the kings are, check all pieces of opposing
color against those kings, then if either get hit, check if its
checkmate
        king = King
        kingDict = {}
        pieceDict = {BLACK : [], WHITE : []}
        for position,piece in self.gameboard.items():
            if type(piece) == King:
                kingDict[piece.Color] = position
            print(piece)
            pieceDict[piece.Color].append((piece,position))
        #white
        if self.canSeeKing(kingDict[WHITE],pieceDict[BLACK]):
            self.message = "White player is in check"
        if self.canSeeKing(kingDict[BLACK],pieceDict[WHITE]):
            self.message = "Black player is in check"
    def canSeeKing(self,kingpos,piecelist):
        #checks if any pieces in piece list (which is an array of
(piece,position) tuples) can see the king in kingpos
        for piece,position in piecelist:
            if
piece.isValid(position,kingpos,piece.Color,self.gameboard):
                return True

    def parseInput(self):
        try:
            a,b = input().split()
            a = ((ord(a[0])-97), int(a[1])-1)
            b = (ord(b[0])-97, int(b[1])-1)
            print(a,b)
            return (a,b)
        except:
            print("error decoding input. please try again")
            return((-1,-1),(-1,-1))

    """def validateInput(self, *kargs):
        for arg in kargs:
            if type(arg[0]) is not type(1) or type(arg[1]) is not
type(1):
                return False
        return True"""

    def printBoard(self):
        print("  1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |")
        for i in range(0,8):
            print("-"*32)
            print(chr(i+97),end="|")
            for j in range(0,8):
                item = self.gameboard.get((i,j)," ")
                print(str(item)+' |', end = " ")
            print()
        print("-"*32)



    """game class. contains the following members and methods:
    two arrays of pieces for each player
    8x8 piece array with references to these pieces
    a parse function, which turns the input from the user into a
list of two tuples denoting start and end points
    a checkmateExists function which checks if either players are in
checkmate
    a checkExists function which checks if either players are in
check (woah, I just got that nonsequitur)
```

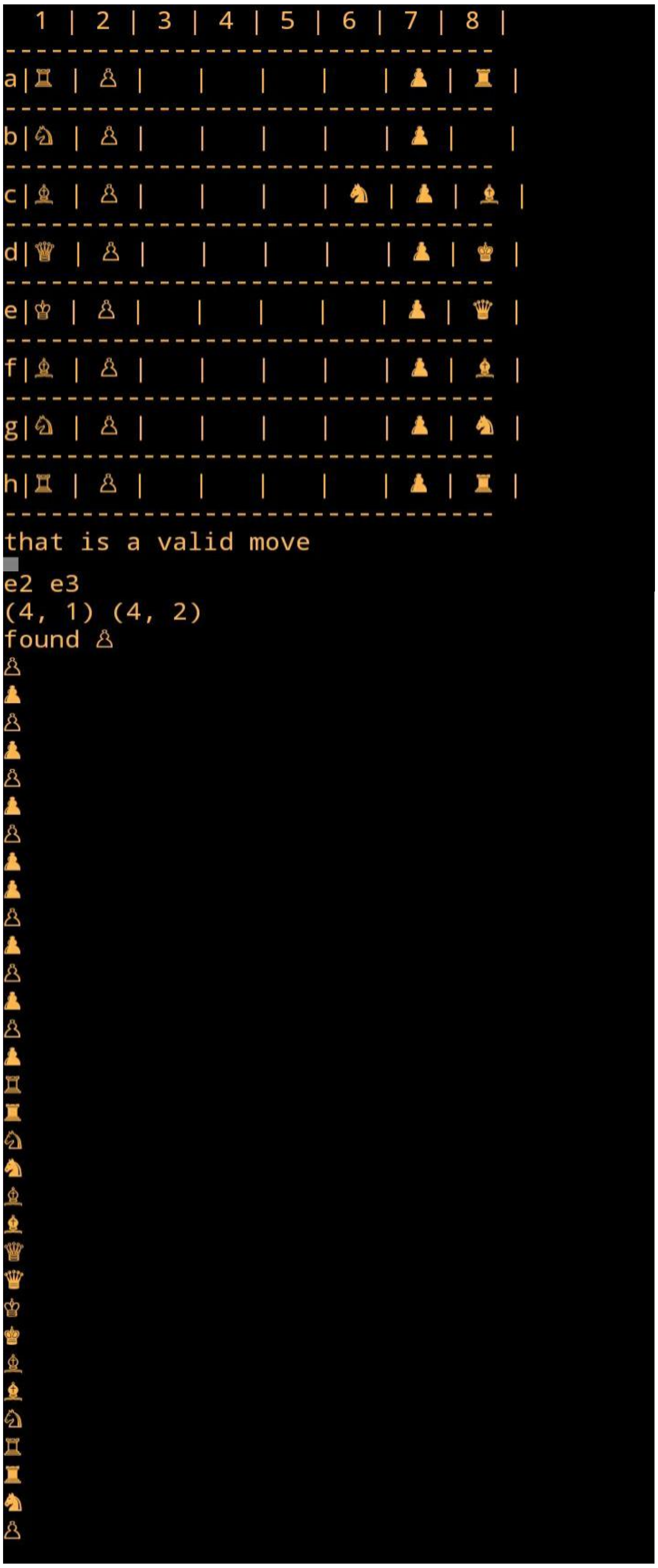
```
    a main loop, which takes input, runs it through the parser, asks
the piece if the move is valid, and moves the piece if it is. if the
move conflicts with another piece, that piece is removed.
ischeck(mate) is run, and if there is a checkmate, the game prints a
message as to who wins
    """

class Piece:

    def __init__(self,color,name):
        self.name = name
        self.position = None
        self.Color = color
    def isValid(self,startpos,endpos,Color,gameboard):
        if endpos in
self.availableMoves(startpos[0],startpos[1],gameboard, Color =
Color):
            return True
        return False
    def __repr__(self):
        return self.name

    def __str__(self):
        return self.name

    def availableMoves(self,x,y,gameboard):
        print("ERROR: no movement for base class")

    def AdNauseum(self,x,y,gameboard, Color, intervals):
        """repeats the given interval until another piece is run
into.
        if that piece is not of the same color, that square is added
and
         then the list is returned"""
        answers = []
        for xint,yint in intervals:
            xtemp,ytemp = x+xint,y+yint
            while self.isInBounds(xtemp,ytemp):
                #print(str((xtemp,ytemp))+"is in bounds")

                target = gameboard.get((xtemp,ytemp),None)
                if target is None: answers.append((xtemp,ytemp))
                elif target.Color != Color:
                    answers.append((xtemp,ytemp))
                    break
                else:
                    break
                xtemp,ytemp = xtemp + xint,ytemp + yint
        return answers

    def isInBounds(self,x,y):
        "checks if a position is on the board"
        if x >= 0 and x < 8 and y >= 0 and y < 8:
            return True
        return False

    def noConflict(self,gameboard,initialColor,x,y):
        "checks if a single position poses no conflict to the rules
of chess"
        if self.isInBounds(x,y) and (((x,y) not in gameboard) or
gameboard[(x,y)].Color != initialColor) : return True
        return False


chessCardinals = [(1,0),(0,1),(-1,0),(0,-1)]
chessDiagonals = [(1,1),(-1,1),(1,-1),(-1,-1)]

def knightList(x,y,int1,int2):
    """sepcifically for the rook, permutes the values needed around
a position for noConflict tests"""
    return [(x+int1,y+int2),(x-int1,y+int2),(x+int1,y-int2),(x-
int1,y-int2),(x+int2,y+int1),(x-int2,y+int1),(x+int2,y-int1),(x-
int2,y-int1)]
def kingList(x,y):
    return [(x+1,y),(x+1,y+1),(x+1,y-1),(x,y+1),(x,y-1),(x-1,y),(x-
1,y+1),(x-1,y-1)]



class Knight(Piece):
    def availableMoves(self,x,y,gameboard, Color = None):
        if Color is None : Color = self.Color
        return [(xx,yy) for xx,yy in knightList(x,y,2,1) if
self.noConflict(gameboard, Color, xx, yy)]

class Rook(Piece):
    def availableMoves(self,x,y,gameboard ,Color = None):
        if Color is None : Color = self.Color
        return self.AdNauseum(x, y, gameboard, Color,
chessCardinals)

class Bishop(Piece):
    def availableMoves(self,x,y,gameboard, Color = None):
        if Color is None : Color = self.Color
        return self.AdNauseum(x, y, gameboard, Color,
chessDiagonals)
```

```
class Queen(Piece):
    def availableMoves(self,x,y,gameboard, Color = None):
        if Color is None : Color = self.Color
        return self.AdNauseum(x, y, gameboard, Color,
chessCardinals+chessDiagonals)

class King(Piece):
    def availableMoves(self,x,y,gameboard, Color = None):
        if Color is None : Color = self.Color
        return [(xx,yy) for xx,yy in kingList(x,y) if
self.noConflict(gameboard, Color, xx, yy)]

class Pawn(Piece):
    def __init__(self,color,name,direction):
        self.name = name
        self.Color = color
        #of course, the smallest piece is the hardest to code.
direction should be either 1 or -1, should be -1 if the pawn is
traveling "backwards"
        self.direction = direction
    def availableMoves(self,x,y,gameboard, Color = None):
        if Color is None : Color = self.Color
        answers = []
        if (x+1,y+self.direction) in gameboard and
self.noConflict(gameboard, Color, x+1, y+self.direction) :
answers.append((x+1,y+self.direction))
        if (x-1,y+self.direction) in gameboard and
self.noConflict(gameboard, Color, x-1, y+self.direction) :
answers.append((x-1,y+self.direction))
        if (x,y+self.direction) not in gameboard and Color ==
self.Color : answers.append((x,y+self.direction))# the condition
after the and is to make sure the non-capturing movement (the only
fucking one in the game) is not used in the calculation of checkmate
        return answers

uniDict = {WHITE : {Pawn : "♙", Rook : "♖", Knight : "♘", Bishop
: "♗", King : "♔", Queen : "♕" }, BLACK : {Pawn : "♟", Rook :
"♜", Knight : "♞", Bishop : "♝", King : "♚", Queen : "♛" }}

Game()
```

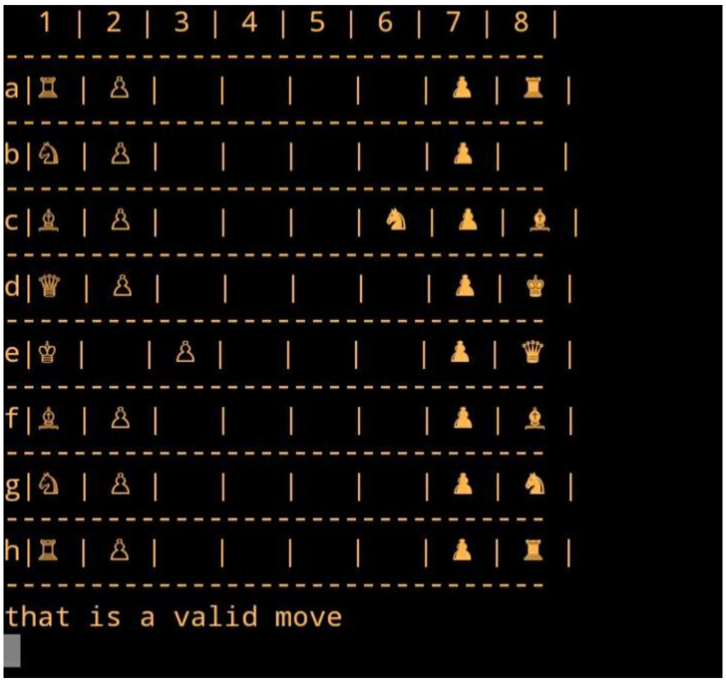
OUTPUT:



```
chess program. enter moves in algebraic no
tation separated by space
  1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
--------------------------------
a|♖ | ♙ |   |   |   |   | ♟ | ♜ |
--------------------------------
b|♘ | ♙ |   |   |   |   | ♟ | ♞ |
--------------------------------
c|♗ | ♙ |   |   |   |   | ♟ | ♝ |
--------------------------------
d|♕ | ♙ |   |   |   |   | ♟ | ♚ |
--------------------------------
e|♔ | ♙ |   |   |   |   | ♟ | ♛ |
--------------------------------
f|♗ | ♙ |   |   |   |   | ♟ | ♝ |
--------------------------------
g|♘ | ♙ |   |   |   |   | ♟ | ♞ |
--------------------------------
h|♖ | ♙ |   |   |   |   | ♟ | ♜ |
--------------------------------
this is where prompts will go
b8 c6
(1, 7) (2, 5)
found ♞
♙
♟
♙
♟
♙
♟
♙
♟
♙
♟
♙
♟
♙
♟
♙
♟
♖
♜
♘
♞
♗
♝
♕
♛
♔
♚
♗
♝
♘
♖
♜
♞
```



```
  1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
--------------------------------
a|♖ | ♙ |   |   |   |   | ♟ | ♜ |
--------------------------------
b|♘ | ♙ |   |   |   |   | ♟ |   |
--------------------------------
c|♗ | ♙ |   |   |   | ♞ | ♟ | ♝ |
--------------------------------
d|♕ | ♙ |   |   |   |   | ♟ | ♚ |
--------------------------------
e|♔ | ♙ |   |   |   |   | ♟ | ♛ |
--------------------------------
f|♗ | ♙ |   |   |   |   | ♟ | ♝ |
--------------------------------
g|♘ | ♙ |   |   |   |   | ♟ | ♞ |
--------------------------------
h|♖ | ♙ |   |   |   |   | ♟ | ♜ |
--------------------------------
that is a valid move

e2 e3
(4, 1) (4, 2)
found ♙
♙
♟
♙
♟
♙
♟
♙
♟
♟
♙
♟
♙
♟
♙
♟
♖
♜
♘
♞
♗
♝
♕
♛
♔
♚
♗
♝
♘
♖
♜
♞
♙
```



```
  1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
--------------------------------
a|♖ | ♙ |   |   |   |   | ♟ | ♜ |
--------------------------------
b|♘ | ♙ |   |   |   |   | ♟ |   |
--------------------------------
c|♗ | ♙ |   |   |   | ♞ | ♟ | ♝ |
--------------------------------
d|♕ | ♙ |   |   |   |   | ♟ | ♚ |
--------------------------------
e|♔ |   | ♙ |   |   |   | ♟ | ♛ |
--------------------------------
f|♗ | ♙ |   |   |   |   | ♟ | ♝ |
--------------------------------
g|♘ | ♙ |   |   |   |   | ♟ | ♞ |
--------------------------------
h|♖ | ♙ |   |   |   |   | ♟ | ♜ |
--------------------------------
that is a valid move
```