```
from colorama import Back, init
import sys
import numpy as np

init(autoreset = True)

# piece object
class p:
    def __init__(self, name, color):
        self.name = name
        self.color = color

# n stands for empty tile
board = [
            [ ' ', 'A', 'B', 'C', 'D', 'E', 'F', 'G', 'H'],
            [ '1', p('r','b'), p('k','b'), p('b','b'), p('q','b'), p('K','b'), p('b','b'), p('k','b'),
p('r','b') ],
            [ '2', p('p','b'), p('p','b'), p('p','b'), p('p','b'), p('p','b'), p('p','b'), p('p','b'),
p('p','b') ],
            [ '3', p('n','n'), p('n','n'), p('n','n'), p('n','n'), p('n','n'), p('n','n'), p('n','n'),
p('n','n') ],
            [ '4', p('n','n'), p('n','n'), p('n','n'), p('n','n'), p('n','n'), p('n','n'), p('n','n'),
p('n','n') ],
            [ '5', p('n','n'), p('n','n'), p('n','n'), p('n','n'), p('n','n'), p('n','n'), p('n','n'),
p('n','n') ],
            [ '6', p('n','n'), p('n','n'), p('n','n'), p('n','n'), p('n','n'), p('n','n'), p('n','n'),
p('n','n') ],
            [ '7', p('p','w'), p('p','w'), p('p','w'), p('p','w'), p('p','w'), p('p','w'), p('p','w'),
p('p','w') ],
            [ '8', p('r','w'), p('k','w'), p('b','w'), p('q','w'), p('K','w'), p('b','w'), p('k','w'),
p('r','w') ],
        ]

# converts the attributes of a piece into unicode characters
n_to_uni = {
                'Kw': '\u2654', # white king
                'qw': '\u2655', # white queen
                'rw': '\u2656', # white rook
                'bw': '\u2657', # white bishop
                'kw': '\u2658', # white knight
                'pw': '\u2659', # white pawn
                'Kb': '\u265A', # black king
```

```
                'qb': '\u265B', # black queen
                'rb': '\u265C', # black rook
                'bb': '\u265D', # black bishop
                'kb': '\u265E', # black knight
                'pb': '\u265F', # black pawn
                'nn': ' '       # empty tile
            }

# this function prints the chess board. the board itself can vary
def print_board():
    for row in board:

        # prints the ABCDEFGH
        if row[0] == ' ':
            for tile in row:

                # everything but the 'H'
                if row.index(tile) != 8:
                    print(' ' + tile + ' ', end = '')

                # the 'H'
                if row.index(tile) == 8:
                    print(' ' + tile + ' ')

        # prints odd rows
        if board.index(row) % 2 == 1:
            for tile in row:

                # the number
                if row.index(tile) == 0:
                    print(' ' + tile + ' ', end = '')

                # the odd tiles
                if row.index(tile) in [1, 3, 5, 7]:
                    print(Back.GREEN + ' ' + n_to_uni[tile.name + tile.color] + ' ', end = '')

                # the even tiles, not the 8th tile
                if row.index(tile) in [2, 4, 6]:
                    print(Back.BLACK + ' ' + n_to_uni[tile.name + tile.color] + ' ', end = '')

                # the 8th tile
                if row.index(tile) == 8:
```

```
                    print(Back.BLACK + ' ' + n_to_uni[tile.name + tile.color] + ' ')

        # prints even rows
        if board.index(row) % 2 == 0 and board.index(row) > 0:
            for tile in row:

                # the number
                if row.index(tile) == 0:
                    print(' ' + tile + ' ', end = '')

                # odd numbered tiles
                if row.index(tile) in [1, 3, 5, 7]:
                    print(Back.BLACK + ' ' + n_to_uni[tile.name + tile.color] + ' ', end = '')

                # even numbered tiles, not the 8th tile
                if row.index(tile) in [2, 4, 6]:
                    print(Back.GREEN + ' ' + n_to_uni[tile.name + tile.color] + ' ', end = '')

                # the 8th tile
                if row.index(tile) == 8:
                    print(Back.GREEN + ' ' + n_to_uni[tile.name + tile.color] + ' ')


# this converts letters to numbers
l_to_n = {
            'a': 1,
            'b': 2,
            'c': 3,
            'd': 4,
            'e': 5,
            'f': 6,
            'g': 7,
            'h': 8
         }


# gets the move from a player
def get_move(player):

    # true only when the move exists
    exists = False

```

```
    col = False
    row = False
    to = False
    piece = False

    possible_cols = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h']
    possible_rows = ['1', '2', '3', '4', '5', '6', '7', '8']

    # gets the move, stopping when the move exists
    while exists == False:

        move = input('{name}\'s Move: '.format(name = player)).lower().replace(' ', '')

        if move == 'surrender' or move == 'concede':
            break

        # the move must have six characters: start tile, 'to', and end tile
        if len(move) == 6:
            if move[0] in possible_cols and move[4] in possible_cols:
                col = True
            if move[1] in possible_rows and move[5] in possible_rows:
                row = True
            if move[2:4] == 'to':
                to = True
            if board[int(move[1])][l_to_n[move[0]]].name != 'n':
                piece = True

        if col == True and row == True and to == True and piece == True:
            exists = True
        else:
            print('Invalid Move.')

    return move


# does the move designated by a player
def do_move(move, kind, s, e, p):

    color = p.color

    # does anything but a castle
    if kind == 'not a castle':
```

```
161
162             # assigns the piece's data to the end tile
163             board[int(e[1])][l_to_n[e[0]]].name = p.name
164             board[int(e[1])][l_to_n[e[0]]].color = p.color
165
166             # clears the start tile
167             board[int(s[1])][l_to_n[s[0]]].name = 'n'
168             board[int(s[1])][l_to_n[s[0]]].color = 'n'
169
170         # does a kingside castle
171         if kind == 'kingside castle':
172
173             # assigns the king's data to the end tile
174             board[int(e[1])][l_to_n[e[0]]].name = 'K'
175             board[int(e[1])][l_to_n[e[0]]].color = color
176
177             # clears the tile the king was on
178             board[int(s[1])][l_to_n[s[0]]].name = 'n'
179             board[int(s[1])][l_to_n[s[0]]].color = 'n'
180
181             # assigns the rook to its end tile
182             board[int(e[1])][6].name = 'r'
183             board[int(e[1])][6].color = color
184
185             # clears the tile the rook was on
186             board[int(e[1])][8].name = 'n'
187             board[int(e[1])][8].color = 'n'
188
189         # does a queenside castle
190         if kind == 'queenside castle':
191
192             # assigns the king's data to the end tile
193             board[int(e[1])][l_to_n[e[0]]].name = 'K'
194             board[int(e[1])][l_to_n[e[0]]].color = color
195
196             # clears the tile the king was on
197             board[int(s[1])][l_to_n[s[0]]].name = 'n'
198             board[int(s[1])][l_to_n[s[0]]].color = 'n'
199
200             # assigns the rook to its end tile
201             board[int(e[1])][4].name = 'r'
202             board[int(e[1])][4].color = color
```

```
203
204             # clears the tile the rook was on
205             board[int(e[1])][1].name = 'n'
206             board[int(e[1])][1].color = 'n'
207
208
209     # determines the type of a move
210     def determine_move_type(move, s, e, p):
211
212         is_castle = 'not a castle'
213
214         is_king = False
215         row = False
216         col = False
217
218         pos_row = ['1', '8']
219
220         # is the piece a king?
221         if p.name == 'K':
222             is_king = True
223
224         # is the piece in the correct row for a castle
225         if int(s[1]) - int(e[1]) == 0 and s[1] in pos_row and e[1] in pos_row:
226             row = True
227
228         # to which side of the board is the castle on
229         if row == True and s[0] == 'e' and e[0] == 'g':
230             is_castle = 'kingside castle'
231
232         if row == True and s[0] == 'e' and e[0] == 'c':
233             is_castle = 'queenside castle'
234
235         return is_castle
236
237
238     # determine all the possible moves for a white pawn
239     def white_pawn_possible_moves(r, c):
240
241         pos_moves = []
242         pos_attacks = []
243
244         # looks to see if a white pawn can move two forward on its first move
```

```
245         if r == 7 and board[r-1][c].name == 'n' and board[r-2][c].name == 'n':
246             pos_moves.append(str(r-2) + str(c))
247 
248         # looks to see if a white pawn can move one forward
249         if r != 1 and board[r-1][c].name == 'n':
250             pos_moves.append(str(r-1) + str(c))
251 
252 
253         # looks to see if a white pawn not in column A can attack to the left
254         if r != 1 and c != 1 and board[r-1][c-1].color == 'b':
255             pos_moves.append(str(r-1) + str(c-1))
256             pos_attacks.append(str(r-1) + str(c-1))
257 
258         # looks to see if a white pawn not in column H can attack to the right
259         if r != 1 and c != 8 and board[r-1][c+1].color == 'b':
260             pos_moves.append(str(r-1) + str(c+1))
261             pos_attacks.append(str(r-1) + str(c+1))
262 
263         attacks_and_moves = [pos_moves, pos_attacks]
264 
265         return attacks_and_moves
266 
267 
268     # determine all the possible moves for a black pawn
269     def black_pawn_possible_moves(r, c):
270 
271         pos_moves = []
272         pos_attacks = []
273 
274         # looks to see if a black pawn can move two forward on its first move
275         if r == 2 and board[r+1][c].name == 'n' and board[r+2][c].name == 'n':
276             pos_moves.append(str(r+2) + str(c))
277 
278         # looks to see if a black pawn can move one forward
279         if r != 8 and board[r+1][c].name == 'n':
280             pos_moves.append(str(r+1) + str(c))
281 
282         # looks to see if a black pawn not in column A can attack to the left
283         if r != 8 and c != 1 and board[r+1][c-1].color == 'w':
284             pos_moves.append(str(r+1) + str(c-1))
285             pos_attacks.append(str(r+1) + str(c-1))
286 
```

```
287         # looks to see if a black pawn not in column H can attack to the right
288         if r != 8 and c != 8 and board[r+1][c+1].color == 'w':
289             pos_moves.append(str(r+1) + str(c+1))
290             pos_attacks.append(str(r+1) + str(c+1))
291 
292         attacks_and_moves = [pos_moves, pos_attacks]
293 
294         return attacks_and_moves
295 
296 
297     # determine all the possible moves for a rook
298     def rook_possible_moves(r, c, color):
299 
300         pos_moves = []
301 
302         # check all the squares above the rook
303         i = 1
304         while r - i >= 1:
305 
306             # check for an empty tile
307             if board[r-i][c].color == 'n':
308                 pos_moves.append(str(r-i) + str(c))
309 
310             # check for an enemy piece
311             if board[r-i][c].color not in [color, 'n']:
312                 pos_moves.append(str(r-i) + str(c))
313                 break
314 
315             # check for a friendly piece
316             if board[r-i][c].color == color:
317                 break
318 
319             i += 1
320 
321         # check all the squares below the rook
322         i = 1
323         while r + i <= 8:
324 
325             # check for an empty tile
326             if board[r+i][c].color == 'n':
327                 pos_moves.append(str(r+i) + str(c))
328 
```

```
329             # check for an enemy piece
330             if board[r+i][c].color not in [color, 'n']:
331                 pos_moves.append(str(r+i) + str(c))
332                 break
333 
334             # check for a friendly piece
335             if board[r+i][c].color == color:
336                 break
337 
338             i += 1
339 
340         # check all the squares to the right of the rook
341         i = 1
342         while c + i <= 8:
343 
344             # check for an empty tile
345             if board[r][c+i].color == 'n':
346                 pos_moves.append(str(r) + str(c+i))
347 
348             # check for an enemy piece
349             if board[r][c+i].color not in [color, 'n']:
350                 pos_moves.append(str(r) + str(c+i))
351                 break
352 
353             # check for a friendly piece
354             if board[r][c+i].color == color:
355                 break
356 
357             i += 1
358 
359         # check all the squares to the left of the rook
360         i = 1
361         while c - i >= 1:
362 
363             # check for an empty tile
364             if board[r][c-i].color == 'n':
365                 pos_moves.append(str(r) + str(c-i))
366 
367             # check for an enemy piece
368             if board[r][c-i].color not in [color, 'n']:
369                 pos_moves.append(str(r) + str(c-i))
370                 break
```

```
371
372         # check for a friendly piece
373         if board[r][c-i].color == color:
374             break
375
376         i += 1
377
378     return pos_moves
379
380
381 # determine all the possible moves for a bishop
382 def bishop_possible_moves(r, c, color):
383
384     pos_moves = []
385
386     # check all the squares to the bottom right diagonal
387     i = 1
388     while r + i <= 8 and c + i <= 8:
389
390         # check for an empty tile
391         if board[r+i][c+i].color == 'n':
392             pos_moves.append(str(r+i) + str(c+i))
393
394         # check for an enemy piece
395         if board[r+i][c+i].color not in [color, 'n']:
396             pos_moves.append(str(r+i) + str(c+i))
397             break
398
399         # check for a friendly piece
400         if board[r+i][c+i].color == color:
401             break
402
403         i += 1
404
405     # check all the squares to the upper right diagonal
406     i = 1
407     while r - i >= 1 and c + i <= 8:
408
409         # check for an empty tile
410         if board[r-i][c+i].color == 'n':
411             pos_moves.append(str(r-i) + str(c+i))
412
```

```
413             # check for an enemy piece
414             if board[r-i][c+i].color not in [color, 'n']:
415                 pos_moves.append(str(r-i) + str(c+i))
416                 break
417 
418             # check for a friendly piece
419             if board[r-i][c+i].color == color:
420                 break
421 
422             i += 1
423 
424         # check all the squares to the bottom left diagonal
425         i = 1
426         while r + i <= 8 and c - i >= 1:
427 
428             # check for an empty tile
429             if board[r+i][c-i].color == 'n':
430                 pos_moves.append(str(r+i) + str(c-i))
431 
432             # check for an enemy piece
433             if board[r+i][c-i].color not in [color, 'n']:
434                 pos_moves.append(str(r+i) + str(c-i))
435                 break
436 
437             # check for a friendly piece
438             if board[r+i][c-i].color == color:
439                 break
440 
441             i += 1
442 
443         # check all the squares to the upper left diagonal
444         i = 1
445         while r - i >= 1 and c - i >= 1:
446 
447             # check for an empty tile
448             if board[r-i][c-i].color == 'n':
449                 pos_moves.append(str(r-i) + str(c-i))
450 
451             # check for an enemy piece
452             if board[r-i][c-i].color not in [color, 'n']:
453                 pos_moves.append(str(r-i) + str(c-i))
454                 break
```

```
455
456             # check for a friendly piece
457             if board[r-i][c-i].color == color:
458                 break
459
460             i += 1
461
462         return pos_moves
463
464
465     # determine all the possible moves for a queen
466     def queen_possible_moves(r, c, color):
467
468         # a queen just has the combined moves of a rook and a bishop
469         diagonals = bishop_possible_moves(r, c, color)
470         horizontals_and_verticals = rook_possible_moves(r, c, color)
471
472         # combine the bishop and rook moves
473         pos_moves = np.concatenate((diagonals, horizontals_and_verticals))
474
475         return pos_moves
476
477     # determine all the possible moves for a knight
478     def knight_possible_moves(r, c, color):
479
480         pos_moves = []
481
482         surrounding_tiles = [
483                                 [r+2, c+1],
484                                 [r+1, c+2],
485                                 [r-1, c+2],
486                                 [r-2, c+1],
487                                 [r-2, c-1],
488                                 [r-1, c-2],
489                                 [r+1, c-2],
490                                 [r+2, c-1]
491                             ]
492
493         # search the surrouding tiles for an empty tile or an enemy piece
494         for tile in surrounding_tiles:
495             if tile[0] in range(1,9) and tile[1] in range(1,9):
496                 if board[tile[0]][tile[1]].color != color:
```

```
                pos_moves.append(str(tile[0]) + str(tile[1]))

    return pos_moves


# produce an array of all tiles 1 away from a tile
def get_surrounding_tiles(r, c):

    surrounding_tiles = [
                            [r+1, c],
                            [r+1, c+1],
                            [r, c+1],
                            [r-1, c+1],
                            [r-1, c],
                            [r-1, c-1],
                            [r, c-1],
                            [r+1, c-1]
                        ]

    return surrounding_tiles

# determine every tile that an enemy piece can attack
def danger_tiles(color):

    danger = []

    # check the entire board looking for getting every possible tile thats attackable
    for row in range(1,9):
        for tile in range(1,9):
            if board[row][tile].color not in [color, 'n']:

                # pawn
                if board[row][tile].name == 'p':
                    # black pawn
                    if color == 'w':
                        for tiles in black_pawn_possible_moves(row, tile)[1]:
                            danger.append(tiles)
                    # white pawn
                    if color == 'b':
                        for tiles in white_pawn_possible_moves(row, tile)[1]:
                            danger.append(tiles)

```

```
539                 # rook
540                 elif board[row][tile].name == 'r':
541                     # black rook
542                     if color == 'w':
543                         for tiles in rook_possible_moves(row, tile, 'b'):
544                             danger.append(tiles)
545                     # white rook
546                     if color == 'b':
547                         for tiles in rook_possible_moves(row, tile, 'w'):
548                             danger.append(tiles)
549 
550                 # bishop
551                 elif board[row][tile].name == 'b':
552                     # black bishop
553                     if color == 'w':
554                         for tiles in bishop_possible_moves(row, tile, 'b'):
555                             danger.append(tiles)
556                     # white bishop
557                     if color == 'b':
558                         for tiles in bishop_possible_moves(row, tile, 'w'):
559                             danger.append(tiles)
560 
561                 # queen
562                 elif board[row][tile].name == 'q':
563                     # black queen
564                     if color == 'w':
565                         for tiles in queen_possible_moves(row, tile, 'b'):
566                             danger.append(tiles)
567                     # white queen
568                     if color == 'b':
569                         for tiles in queen_possible_moves(row, tile, 'w'):
570                             danger.append(tiles)
571 
572                 # knight
573                 elif board[row][tile].name == 'k':
574                     # black knight
575                     if color == 'w':
576                         for tiles in knight_possible_moves(row, tile, 'b'):
577                             danger.append(tiles)
578                     # white knight
579                     if color == 'b':
580                         for tiles in knight_possible_moves(row, tile, 'w'):
```

```
                                danger.append(tiles)

                    # king
                    elif board[row][tile].name == 'K':
                        for tiles in get_surrounding_tiles(row, tile):
                            danger.append(str(tiles[0]) + str(tiles[1]))

        return danger


# determine all the possible moves for a king
def king_possible_moves(r, c, color):

    # get all the tiles surrounding the king
    pos_moves = []

    nearby_tiles = get_surrounding_tiles(r, c)

    # white castling
    if r == 8 and c == 5 and color == 'w':
        # kingside castle
        if board[8][6].color == 'n' and board[8][7].color == 'n':
            if board[8][8].name == 'r' and board[8][8].color == 'w':
                nearby_tiles.append([8, 7])
        # queenside castle
        if board[8][4].color == 'n' and board[8][3].color == 'n' and board[8][2].color == 'n':
            if board[8][1].name == 'r' and board[8][1].color == 'w':
                nearby_tiles.append([8, 3])

    # black castling
    if r == 1 and c == 5 and color == 'b':
        # kingside castle
        if board[1][6].color == 'n' and board[1][7].color == 'n':
            if board[1][8].name == 'r' and board[1][8].color == 'b':
                nearby_tiles.append([1, 7])
        # queenside castle
        if board[1][4].color == 'n' and board[1][3].color == 'n' and board[1][2].color == 'n':
            if board[1][1].name == 'r' and board[1][1].color == 'b':
                nearby_tiles.append([1, 3])

    # search the surrouding tiles for an empty tile or an enemy piece
    for tile in nearby_tiles:
```

```
623             if tile[0] in range(1,9) and tile[1] in range(1,9):
624                 if board[tile[0]][tile[1]].color != color:
625                     pos_moves.append(str(tile[0]) + str(tile[1]))
626 
627         danger = danger_tiles(color)
628 
629         # take out all the tiles that can be attacked
630         for tile in pos_moves:
631             if tile in danger:
632                 pos_moves.remove(tile)
633 
634         return pos_moves
635 
636     # produce an array of the possible moves for any piece
637     def check_is_move_possible(s, p):
638 
639         pos_moves = []
640 
641         row = int(s[1])
642         col = l_to_n[s[0]]
643 
644         # white pawn
645         if p.name == 'p' and p.color == 'w':
646             pos_moves = white_pawn_possible_moves(row, col)[0]
647 
648         # black pawn
649         if p.name == 'p' and p.color == 'b':
650             pos_moves = black_pawn_possible_moves(row, col)[0]
651 
652         # rook
653         if p.name == 'r':
654             pos_moves = rook_possible_moves(row, col, p.color)
655 
656         # bishop
657         if p.name == 'b':
658             pos_moves = bishop_possible_moves(row, col, p.color)
659 
660         # queen
661         if p.name == 'q':
662             pos_moves = queen_possible_moves(row, col, p.color)
663 
664         # knight
```

```
665      if p.name == 'k':
666          pos_moves = knight_possible_moves(row, col, p.color)
667
668      # king
669      if p.name == 'K':
670          pos_moves = king_possible_moves(row, col, p.color)
671
672      return pos_moves
673
674
675  # an array of the two players in the game
676  players = [
677              'White',
678              'Black'
679            ]
680
681  #print the starting board
682  print('Let\'s play chess!')
683  print_board()
684
685  # this runs the entire game
686  game = True
687  while game == True:
688
689      # alternate each player
690      for player in players:
691
692          # get the move from the current player
693          exists = False
694          while exists == False:
695
696              move = get_move(player)
697
698              # check if someone surrenders
699              if move == 'concede' or move == 'surrender':
700                  if player == 'White':
701                      print('Black Wins!\n')
702                  if player == 'Black':
703                      print('White Wins!\n')
704                  sys.exit()
705
706              start = move[0:2]
```

```
            piece = board[int(start[1])][l_to_n[start[0]]]

            end_tile = False
            color = False

            # is the piece the player wants to move the right color?
            if player == 'White' and piece.color == 'w':
                color = True
            if player == 'Black' and piece.color == 'b':
                color = True

            # is the move in the piece's array of possible moves?
            if str(move[5]) + str(l_to_n[move[4]]) in check_is_move_possible(start, piece):
                end_tile = True

            # if the piece is the correct color and the move is possible, this while loop breaks
            if color == True and end_tile == True:
                break
            else:
                print('Invalid Move')

        # assign variables for the start and end tiles
        start = move[0:2]
        end = move[4:6]

        # get the piece thats being moved in the current turn
        piece = board[int(start[1])][l_to_n[start[0]]]
        name = piece.name
        color = piece.color

        # get the type of move
        castle = determine_move_type(move, start, end, piece)

        # do the move
        do_move(move, castle, start, end, piece)

        # turn pawns who have made it to the end into queens
        if name == 'p':
            if end[1] == '1' and color == 'w':
                board[int(end[1])][l_to_n[end[0]]].name = 'q'
            if end[1] == '8' and color == 'b':
                board[int(end[1])][l_to_n[end[0]]].name = 'q'
```

```
749
750         # print current board
751         print_board()
752
753         print('')
```