```
'''
Created on Feb 21, 2015
Modified Feb 26, 2018 - Expanded comments

@author: mroch
'''

from basicsearch_lib.board import Board
from copy import copy, deepcopy
import operator

class CheckerBoard(Board):
    '''
    CheckerBoard - Class for representing a checkerboard
    and making legal moves.

    All references to players in that are accessible externally to this
    class should use the pawn names.

    Note that this implementation is designed for readability, not
    efficiency.  There are many changes that could be made to
    improve the speed of this class. As an example, the board could
    be represented as a single list of 32 positions.

    You should not be making changes to the design for this
    assignment, but if you want to have fun later, redesigning
    it for efficiency could be fun and improve the number of plys
    that can be efficiently searched.

    Board notation:
    Board is arranged as with black pieces on top, red pieces on bottom
    Positions are denoted (row, column).
    Examples form the initial board setup before play begins
        left-most red pawn in the row closest to the red player:  (7,0)
        right-most black pawn farthest from the red player:  (0,7)
    Note that playable columns alternate.  In row 0, they are 1, 3, 5, 7
    and in row 1 they are 0, 2, 4, 6 making a modulo 2 counting scheme
    useful for determining which columns are valid.

    Initial board:
       0  1  2  3  4  5  6  7
    0  .  b  .  b  .  b  .  b
    1  b  .  b  .  b  .  b  .
    2  .  b  .  b  .  b  .  b
    3  .  .  .  .  .  .  .  .
    4  .  .  .  .  .  .  .  .
    5  r  .  r  .  r  .  r  .
    6  .  r  .  r  .  r  .  r
    7  r  .  r  .  r  .  r  .
    '''

    # class variables and methods  ----------------------------------------

    # Lists for pawn, king, and player checks
    pawns = ['r', 'b']      # red and black pawns
    kings = ['R', 'B']      # red and black kings
    players = [['r', 'R'], ['b', 'B']]  # pieces for each player

    # Possible moves that will need to be validated for any position
```

```
60      # pawns[0] red player moves towards top of board (row 0)
61      # pawns[1] black player moves towards bottom of board (row N)
62      pawnmoves = {pawns[0] : [ (-1, -1), (-1, 1)],
63                   pawns[1] : [ (1, -1), (1, 1)]}
64      # kings can move forwards and backwards
65      kingmoves = [ (-1, 1), (1, 1), (-1, -1), (1, -1) ]
66
67
68      step = 2  # Number of steps between valid columns
69
70      # Number of moves for smallest tour
71      # Tours end in the place they started and can only be done
72      # by kings
73      shortest_tour = 4
74
75      # class methods - useful for evaluation methods
76      @classmethod
77      def piece_types(cls, player):
78          """piece_types - Return pawn and king values for specified player
79          e.g. piece_types('r') returns ['r', 'R']
80          """
81
82          try:
83              index = cls.pawns.index(player)
84          except ValueError:
85              raise ValueError("No such player")
86
87          return cls.players[index]
88
89      @classmethod
90      def other_player(cls, player):
91          "other_player(player) - Return other player pawn based on a pawn"
92          try:
93              index = cls.pawns.index(player)
94          except ValueError:
95              raise ValueError("No such player")
96
97          return cls.pawns[(index + 1) % 2]
98
99      @classmethod
100     def ispawn(cls, piece):
101         "True if piece is a pawn"
102         return piece in cls.pawns
103
104     @classmethod
105     def isking(cls, piece):
106         "True if piece is a king"
107         return piece in cls.kings
108
109     @classmethod
110     def isplayer(cls, player, piece):
111         """isplayer - Does a piece belong to a player.
112         Given a player name (value of cls.pawns r/b unless changed)
113         and a piece from a board, does this piece belong to the
114         specified player?
115         Example:  isplayer('r', 'R') returns True
116                   isplayer('r', None) returns False
117                   isplayer('r', 'b') returns False
118         """
```

```
119         try:
120             index = cls.pawns.index(player)
121         except ValueError:
122             raise ValueError("No such player")
123 
124         return piece in cls.players[index]
125 
126     @classmethod
127     def playeridx(cls, player):
128         "playeridx(player) - Give idx of player based on pawn name"
129 
130         try:
131             pidx = cls.pawns.index(player)
132         except ValueError:
133             raise ValueError("Unknown player")
134         return pidx
135 
136     @classmethod
137     def identifypiece(cls, piece):
138         """identifytpiece(piece)
139         Returns a tuple indicating (playeridx, kingpred)
140         Used to find the player index of a piece and whether the piece
141         is a king (True) or pawn (False)
142         e.g. identifypiece('b') returns (1,False)
143         """
144         try:
145             # Check if it is a pawn and note index
146             idx = cls.pawns.index(piece)
147             kingP = False  # king predicate - not a king
148         except ValueError:
149             # Similar for king
150             try:
151                 idx = cls.kings.index(piece)
152                 kingP = True
153             except ValueError:
154                 raise ValueError("Unknown piece type")
155 
156         return (idx, kingP)
157 
158 
159 
160 
161     # instance methods ----------------------------------------------
162 
163     def __init__(self):
164         "CheckerBoard - Create a new checkerboard"
165 
166         # Create the board
167         self.edgesize = 8  # Number of squares per edge
168         # Checkers only move on the dark squares, so game space
169         # is only half as many states.  Note the number of valid
170         # locations per row.
171         self.locations_per_row = int(self.edgesize / self.step)
172 
173         super(CheckerBoard, self).__init__(self.edgesize, self.edgesize,
174                                            displaycol=3)
175         # for each row, indicate whether the squares that pieces move
176         # in are offset by 0 or 1.
177         # This lets us know that in some rows columns are 0, 2, 4, ...
```

```
178         # and in others they are (0, 2, 4, ...)+1 = (1, 3, 5, 7, ...
179         # We store a 0 or 1 offset value for each row.
180         self.coloffset = [(r + 1) % self.step for r in range(self.edgesize)]
181
182         rowpieces = 3  # Initial rows of checkers for each side
183
184         # rows in which the players are kinged
185         self.kingrows = [0, self.edgesize - 1]
186
187         # Valid spaces are offset in each row.  At top left of board
188         # row 0, col 0, the column offset is 0 before we reach the first
189         # valid location.
190         # In the next row, we need to move one to the right, e.g. [1,1]
191         # before we reach a valid checker position.  Row three is back
192         # to an offset of 0:  [2, 0]
193         # Establish a set of offsets that indicate whether column 0 or 1
194         # is the first valid position
195
196         self.pawnsN = [0, 0]  # Number remaining pawns, indexed by player
197         self.kingsN = [0, 0]  # Number remaining kings
198         for row in range(self.rows):
199             # Place pawns in this row?
200             if row < rowpieces or row >= self.rows - rowpieces:
201                 if row < rowpieces:
202                     playeridx = 1
203                 else:
204                     playeridx = 0
205                 for col in range(self.locations_per_row):
206                     self.place(row, col * self.step + self.coloffset[row],
207                                self.pawns[playeridx])
208                     self.pawnsN[playeridx] += 1
209             # Place spaces in illegal positions to make board more readable
210             for col in range(self.locations_per_row):
211                 self.board[row][col * self.step+(self.coloffset[row]+1)%2]=' '
212
213
214         self.movecount = 0
215         # Counters for draw detection
216
217         # Note that World Checker/Draughts Federation (WCDF) rules indicate that
218         # reaching the same configuration board configuration 3 times in a row
219         # is also a draw, but this is not implemented.
220
221         # Used for detecting draws which are defined as N moves without
222         # advancing a pawn AND no captures
223         self.drawthreshN = 40
224         self.lastcapture = 0  # move # of last capture
225         self.lastpawnadvance = 0  # move number of last pawn advance
226
227     def disttoking(self, player, row):
228         "disttoking - how many rows from king position for player given row"
229
230         # find row offset of any legal move for a pawn,
231         # that is, which way does the pawn move?
232         direction = self.pawnmoves[player][0][0]
233         if direction < 0:
234             distance = row  # red
235         else:
236             distance = self.rows -1 - row  # black
```

```
237         return distance
238
239     def get_pawnsN(self):
240         "get_pawnsN - Return counts of pawns"
241         return self.pawnsN
242
243     def get_kingsN(self):
244         "get_kingsN - Return counts of kings"
245         return self.kingsN
246
247     def isempty(self, row, col):
248         "isempty - Is the specified space empty?"
249         return self.board[row][col] == None
250
251     def clearboard(self):
252         """clearboard - remove all pieces
253         Useful for building specific board configurations
254         WARNING:  Piece counts will be incorrect after calling
255         this.  Call update_counts() to correct after placing new pieces
256         """
257
258         # Iterate over every piece and remove it
259         for (r, c, piece) in self:
260             self.place(r, c, None)
261
262     def update_counts(self):
263         """update_counts - When mucking around with the board, the counts
264         of pawns and kings may be corrupted.  This method updates them.  Valid
265         moves will not cause any problems, this is mainly for testing.
266         """
267         self.pawnsN = [0, 0]
268         self.kingsN = [0, 0]
269
270         # Iterate through pieces
271         for (_r, _c, piece) in self:
272             # Find player index and piece type
273             (playerId, kingP) = self.identifypiece(piece)
274             # update appropriate player piece count
275             if kingP:
276                 self.kingsN[playerId] += 1
277             else:
278                 self.pawnsN[playerId] += 1
279
280     def place(self, row, col, piece):
281         "place(row, col, piece) - put a piece on the board"
282
283         # Overrides parent as some spaces are illegal
284         if col < 0 or col > self.cols or row < 0 or row > self.rows:
285             raise ValueError('Bad row or column')
286         if (col + self.coloffset[row]) % self.step == 1:
287             if self.coloffset[row]:
288                 raise ValueError("Column must be odd for row %d" % (row))
289             else:
290                 raise ValueError("Column must be even for row %d" % (row))
291         self.board[row][col] = piece
292
293     def is_terminal(self):
294         """is_terminal - check if game over
295         Returns tuple (terminal, winner)
```

```
296         terminal - True implies game over
297         winner - only applicable if terminal is true
298             indicates winner by player color or None for draw
299         """
300
301         # Add the pawns and kings together
302         piececounts = list(map(operator.add, self.pawnsN, self.kingsN))
303         if not piececounts[0]:
304             winner = self.pawns[1]
305             terminal = True
306         elif not piececounts[1]:
307             winner = self.pawns[0]
308             terminal = True
309         else:
310             winner = None
311             # Check for draws
312             terminal = \
313                 self.movecount - self.lastpawnadvance >= self.drawthreshN or \
314                 self.movecount - self.lastcapture >= self.drawthreshN
315
316         return (terminal, winner)
317
318     def get_actions(self, player):
319         """Return actions for specified player, CheckerBoard.pawns[i]
320         Valid actions are lists of the following form:
321
322         [move1, move2, move3, ..., moveN] where each move consists of
323         a list of two or more tuples
324
325         The first tuple represents the original position (row, col) of
326         the piece, e.g. (5,4)
327
328         A second tuple is either a simple move represented as (row, col) or
329         a capture which is a 3-tuple with the third element being a
330         tuple indicating the captured piece.
331
332         Examples:
333         possible opening move by player at bottom of board
334            0  1  2  3  4  5  6  7                    0  1  2  3  4  5  6  7
335         0  .  b  .  b  .  b  .  b                 0  .  b  .  b  .  b  .  b
336         1  b  .  b  .  b  .  b  .                 1  b  .  b  .  b  .  b  .
337         2  .  b  .  b  .  b  .  b                 2  .  b  .  b  .  b  .  b
338         3  .  .  .  .  .  .  .  .   action        3  .  .  .  .  .  .  .  .
339         4  .  .  .  .  .  .  .  .   [(5,4),(4,3)] 4  .  .  .  r  .  .  .  .
340         5  r  .  r  .  r  .  r  .   results in    5  r  .  r  .  .  .  r  .
341         6  .  r  .  r  .  r  .  r                 6  .  r  .  r  .  r  .  r
342         7  r  .  r  .  r  .  r  .                 7  r  .  r  .  r  .  r  .
343
344         captures are mandatory.  If any captures exist, normally valid
345         non-capture move actions will not be returned.
346
347         given the following board position, red player captures are as
348         follows:
349            0  1  2  3  4  5  6  7
350         0  .  b  .  b  .  b  .  b
351         1  b  .  b  .  b  .  b  .
352         2  .  b  .  .  .  .  .  b
353         3  .  .  .  .  .  .  b  .
354         4  .  .  .  b  .  .  . <r>   red player candidate moves are shown
```

```
        5  r  . <r> . <r> .  .  .    with <> to make it easier to see
        6  .  r  .  r  .  r  .  r
        7  r  .  r  .  r  .  r  .
        [[(4, 7), (2, 5, (3, 6))],
         [(5, 2), (3, 4, (4, 3))],
         [(5, 4), (3, 2, (4, 3))]]

        Example of multiple jump moves by red player.  As per World Checkers
        Draughts Federation Rules, once started a multiple jump move must
        be made to completion.
           0  1  2  3  4  5  6  7
        0  .  b  .  b  .  b  .  b
        1  b  .  r  .  b  .  .  .
        2  .  r  .  .  .  b  .  b
        3  .  .  .  .  .  .  .  .
        4  .  .  .  r  .  b  .  .
        5  .  .  .  .  .  . <r> .
        6  .  r  .  r  .  r  .  r
        7  r  .  .  .  r  .  .  .
        [[(5, 6), (3, 4, (4, 5)), (1, 6, (2, 5))]]
        Note that had multiple capture moves been possible, it is not mandatory
        to take the one with the most jumps
        """

        try:
            pidx = self.pawns.index(player)
        except ValueError:
            raise ValueError("Unknown player")

        # If we see any captures along the way, we will stop looking
        # for moves that do not capture as they will be filtered out
        # at the end.
        moves = []

        # Scan each square
        for r in range(self.rows):
            for c in range(self.coloffset[r], self.cols, self.step):
                piece = self.board[r][c]
                # If square contains pawn/king of player who will be moving
                if piece in self.players[pidx]:
                    # Determine types of moves that can be made
                    if piece == self.pawns[pidx]:
                        movepaths = self.pawnmoves[player]
                    else:
                        movepaths = self.kingmoves
                    # Generate moves based on possible directions
                    newmoves = self.genmoves(r, c, movepaths, pidx)
                    moves.extend(newmoves)

        # Check if any captures are possible
        # If so, remove all non-capture moves as player must make
        # a capture move if one is available.
        captureP = False  # capture predicate
        for a in moves:
            # each action is
            #    [(rsrc, csrc), (rdst, cdst)]  (non-capture case)
            # or [(rsrc, csrc), (rdst, cdst, (rcapture, ccapture), ...]
            captureP = len(a[1]) > 2
            if captureP:
```

```
414                 break
415         if captureP:
416             # Remove non capture moves as player must capture if possible
417             # We only need to check the first destination to see if it
418             # has a capture tuple after the destination row and column
419             moves = [m for m in moves if len(m[1]) > 2]
420 
421         return moves
422 
423 
424     def __iter__(self):
425         """iter - Board iterator
426         Returns (r, c, piece) for non empty spaces.
427         Might be helpful for board evaluation
428         """
429         for r in range(self.rows):
430             for c in range(self.coloffset[r], self.cols, self.step):
431                 if self.board[r][c]:
432                     yield (r, c, self.board[r][c])
433 
434     def move(self, move, validate=[], verbose=False):
435         """move - Apply a move and return a new board
436         move should be a list of the format described in get_actions
437         It is assumed that the move is valid unless validate is set to a
438         list of moves (presumably produced by get_actions(), get_actions is
439         not called as this has probably already been computed.
440         """
441 
442         if validate:
443             if move not in validate:
444                 raise ValueError("Invalid move")
445 
446         # Only need to copy the board and counter arrays
447         # Everything else is static and can be a shallow copy
448         newboard = copy(self)
449         newboard.pawnsN = copy(self.pawnsN)
450         newboard.kingsN = copy(self.kingsN)
451         newboard.board = deepcopy(self.board)
452         newboard.movecount += 1  # Record new move
453 
454         (firstr, firstc) = (lastr, lastc) = move[0]
455         piece = self.get(lastr, lastc)
456         oldpiece = piece  # Just in case we change and want to print
457         newboard.place(lastr, lastc, None)  # Remove from current position
458 
459         captures = 0  # number of captures for verbose output and draw detection
460         # Loop through move sequence, removing any
461         # captured pieces as we go along
462         for item in move[1:]:
463             if len(item) > 2:
464                 # Capture, remove captured piece
465                 captures = captures + 1
466                 posn = item[2]  # captured position
467                 capturedpiece = newboard.get(posn[0], posn[1])
468 
469                 # Remove the piece
470                 newboard.place(posn[0], posn[1], None)
471 
472                 # Decrement count for captured piece
```

```
473                 (pieceidx, kingP) = newboard.identifypiece(capturedpiece)
474                 if kingP:
475                     newboard.kingsN[pieceidx] -= 1
476                 else:
477                     newboard.pawnsN[pieceidx] -= 1
478 
479             # update last known location of moving piece, might happen
480             # more than once in a multiple jump move
481             # In any case, last row and column will represent the final
482             # position of the piece when we finish the loop
483             (lastr, lastc) = item[0:2]
484 
485         if lastr == 0 or lastr + 1 == self.rows:
486             # At end, do we need to crown a pawn?
487             try:
488                 # find the appropriate pawn type and crown it
489                 playeridx = self.pawns.index(piece)
490                 piece = self.players[playeridx][1]  # king
491             except ValueError:
492                 pass  # not a pawn
493 
494         # Put the piece as the last location of the move sequence
495         newboard.place(lastr, lastc, piece)
496 
497         if captures:
498             # Captured something, note the move for draw detection
499             newboard.lastcapture = newboard.movecount
500 
501         if self.ispawn(oldpiece):
502             # Advanced a pawn, note move number for draw detection
503             newboard.lastpawnadvance = newboard.movecount
504             if oldpiece != piece:
505                 # Kinged, update counts
506                 (pieceidx, kingP) = newboard.identifypiece(oldpiece)
507                 newboard.pawnsN[pieceidx] -= 1
508                 newboard.kingsN[pieceidx] += 1
509 
510         if verbose:
511             # Show the move if folks are interested...
512             print()
513             print("Move %s from " % (oldpiece), (firstr, firstc))
514             print(self)
515             print("move: ", end=' ')
516             if captures > 0:
517                 print("captures %d, " % (captures), end=' ')
518             if oldpiece != piece:
519                 print("kinged, ")
520             print()
521             print(newboard)
522 
523         return newboard
524 
525     def onboard(self, r, c):
526         "onboard - Specified row and column on the board?"
527         return r >= 0 and r < self.rows and c >= 0 and c < self.cols
528 
529     def genmoves(self, r, c, movepaths, playeridx):
530         """genmoves - Generate moves from a specific position
531         r,c - position
```

```
        movepaths - list of possible offsets (move directions) for piece
            e.g. for kings:  [ (-1, 1), (1, 1), (-1, -1), (1, -1) ]
            pawns will have a subset of this moving forward or backward
        player - current player 0|1

        Returns list of possible moves (see get_actions) and captures
        """

        actions = self.__movehelper(r, c, movepaths, playeridx, [])
        return actions



    def __movehelper(self, r, c, movepaths, playeridx, history):
        """__movehelper - Helper finds possible moves from a given position.
        Helper function for genmoves
        r,c - position
        movepaths - list of possible offsets for piece
        playeridx - current playeridx 0|1
        history - list of moves made along a path - [] on first call
        Returns list of possible moves (see get_actions) and captures
        which indicates if a capture has been produced by the moves
        generated here or was already true.

        This function is called recursively to track move paths
        """

        otherplayer = (playeridx + 1) % 2
        actions = []
        for m in movepaths:
            rmove = r + m[0]
            cmove = c + m[1]
            # move only valid if it will be on the board
            if self.onboard(rmove, cmove):

                # check if blocked by opposing player, might be able to jump
                if self.board[rmove][cmove] in self.players[otherplayer]:
                    # Blocked See if capture possible by moving one more time
                    rjump = rmove + m[0]
                    cjump = cmove + m[1]
                    if self.onboard(rjump, cjump) and \
                        self.__valid_capture((rmove, cmove), (rjump, cjump),
                                             history):
                        # Note jump
                        if history:
                            # append to a copy of previous jumps so far
                            # We need to copy history as move sequences
                            # can branch, resulting in different moves
                            # with a common past.
                            capture = copy(history)
                            capture.append((rjump, cjump, (rmove, cmove)))
                        else:
                            # first jump
                            capture = [(r, c), (rjump, cjump, (rmove, cmove))]

                        # Crown a king?
                        # As pawns can only move forward, just look if we have
                        # moved to the first or last row.
                        if rjump == 0 or rjump == self.rows:
```

```
                                checkerboard
591                         # Piece has moved onto a first or last row
592                         # If this is a pawn, we stop even if there
593                         # is another capture available
594
595                         # Was this a pawn?
596                         (rstart, cstart) = (capture[0][0], capture[0][1])
597                         if self.get(rstart, cstart) == self.pawns[playeridx]:
598                             # Can't move any more
599                             return [capture]
600
601                     # We can make this move, but if we can continue
602                     # to capture, we are obligated to do so.
603                     # See if we can continue.
604                     # If no more moves are possible, will simply
605                     # return the current move as one possible action
606                     #
607                     # Note:  If we wanted to not force subsequent
608                     # available jumps after the first one, we could
609                     # append the current capture move, and remove the
610                     # code that returns [history] when there are no
611                     # available actions.
612                     more = self.__movehelper(rjump, cjump, movepaths,
613                                              playeridx, capture)
614                     for m in more:
615                         actions.append(m)
616
617             # Regular move possible if not blocked and no history
618             # of captures
619             elif not self.board[rmove][cmove] and not history:
620                 actions.append([(r, c), (rmove, cmove)])
621
622     if history and not actions:
623         # One or more captures have been made, but when we called
624         # __movehelper to see if there were any more, there were
625         # no valid actions.  We set actions to be a list containing
626         # the history that was required to arrive here.
627         actions = [history]
628
629
630     return actions
631
632 def __valid_capture(self, capturedpiece, moveto, history):
633     """__valid_capture
634         capturedpiece - (r,c) tuple to be captured
635         moveto - position to which we will jump
636         history - previous jumps
637
638     already_captured - Prevent taking a piece more than once
639     helper function for __movehelper
640     """
641
642     valid = True  # Until we learn otherwise
643
644     # Verify that there's no piece at the destination.
645     # If there is a piece, check to see if it was the starting
646     # position as it is possible to do a jump tour
647     if self.board[moveto[0]][moveto[1]]:
648         # Something's there.  If it's the starting piece, that's okay,
649         # otherwise not good.  Don't bother checking if there are
```

```
650             # not enough moves.
651             if len(history) >= self.shortest_tour:
652                 valid = history[0] == moveto
653             else:
654                 valid = False
655 
656         if valid and len(history) > 1:
657             for move in history[1:]:
658                 # Anything in the history is a capture as __movehelper
659                 # recursively builds the history on multi-jump moves.
660                 # All capture nodes consist of a 3-tuple:
661                 #  (newrow, newcol, (capturedrow, capturedcol))
662                 # First position in history is the starting one, anything
663                 # afterwards is a capture move.
664 
665                 # If we already captured this piece, we cannot capture
666                 # it again.
667                 if move[2] == capturedpiece:
668                     valid = False
669 
670         return valid
671 
672     def recount_pieces(self):
673         """recount_pieces() - Recount pawns and kings
674         This utility function is not normally needed.  However, when
675         configuration custom boards where pieces are manually placed
676         (e.g. in boardlibary), the board counts will not longer be accurate.
677         This resets the counters based on the current configuration.
678         """
679         self.pawnsN = [0, 0]
680         self.kingsN = [0, 0]
681         for (r, c, piece) in self:
682             (playeridx, kingP) = self.identifypiece(piece)
683             if kingP:
684                 self.kingsN[playeridx] += 1
685             else:
686                 self.pawnsN[playeridx] += 1
687 
688 
689 
690 
```