```
 1 from basicsearch_lib.board import Board
 2 from math import sqrt
 3 from random import shuffle
 4
 5 ##########################################################
   #############
 6 # Todo List:
 7 #   todo - get_actions()
 8 #   todo - move()
 9 #   todo - is my way of inheritance calling super correct?
10 ##########################################################
   #############
11
12
13 class TileBoard(Board):  # does Board inside () mean
   TileBoard is a Board ?
14     """Tile Board is an object that
15     is a representation of a n-puzzles"""
16
17     def __init__(self, n, forced_state=None):
18         """This is a constructor that will
19         create a board of a particular size
20         @param n and will allow a list to be
21         specifically assigned if desired"""
22         self.n = n
23         self.total_number_of_tiles = n + 1
24         self.odd_half_way_point = n // 2 + 1
25         self.sqrt_total_number_of_tiles = int(sqrt(self.
   total_number_of_tiles))
26         self.forced_state = forced_state  # <List>
27         super().__init__(self.sqrt_total_number_of_tiles,
   self.sqrt_total_number_of_tiles)  # todo: does this have
   to be super(TileBoard, self)?
28
29         if forced_state is not None:
30             self.board = self.list_to_tile_board(self.
   forced_state)
31             #if not self.board.solvable():
32             # todo: throw error
33
34         #else: # todo: shuffle board and make sure it's
   solvable
35
36     def solved(self): # todo: test with different number
   of rows and cols
37         goal_state = TileBoard.get_goal_state(self.n)
38         return self.__eq__(goal_state)
39
40     def __eq__(self, other_object):
```

```
41          """This will overload the == operator to
42          check if two boards have the exact same state"""
43          return self.board == other_object.board
44
45      def state_tuple(self):
46          """This will convert a board to a tuple"""
47          temp_list = []
48          for i in range(self.rows):
49              for j in range(self.cols):
50                  temp_list.append(self.get(i,j))
51          return tuple(temp_list)
52
53      def get_goal_state(n):  # static method todo: why do I
     need to pass in a Tileboard object??
54          temp_board = TileBoard(n)
55          item = 1
56          for i in range(temp_board.rows):
57              for j in range(temp_board.cols):
58                  if item == temp_board.odd_half_way_point:
59                      temp_board.place(i, j, None)
60                  else:
61                      if item >= temp_board.
    odd_half_way_point+1: # Note: <= saved the day
62                          temp_board.place(i, j, item-1)
63                      else:
64                          temp_board.place(i, j, item)
65                  item += 1
66          return temp_board
67
68      def solvable(self):
69          """Using the Inversion order technique we can
70          figure out if a particular board (<List> object)
    instance is solvable or not.
71
72          The inversion order is the sum of all permutation
    inversions for each tile.
73          Conditions:
74          - Two elements a[i] and a[j] form an inversion if
    i < j and a[i] > a[j]
75          If the <Board> object has an even number of rows,
    then the row of the
76          blank must be added to the inversion number.
77
78          Note: The board will be solvable if the inversion
    order is even.
79          """
80          inversion_order = 0
81          target_value = None
82          board_as_list = list(self.state_tuple())
```

```
 83         for idx in range(len(board_as_list)-1): #  todo
    check if minus 1 is correct
 84             for compare_idx in range((idx+1), len(
    board_as_list)):
 85                 count = 0
 86                 if board_as_list[idx] and board_as_list[
    compare_idx] is not None:  # really important for future
    comparing
 87                     if board_as_list[idx] > board_as_list
    [compare_idx]:
 88                         count += 1
 89                 inversion_order += count
 90         if self.get_rows() % 2 == 0:
 91             inversion_order += board_as_list.
    find_item_idx(target_value)[0]  # todo - should I catch
    False?
 92         if inversion_order % 2 == 0:
 93             return True
 94         else:
 95             return False
 96
 97     def find_item_idx(self, target_item):
 98         """Returns a tuple containing the index of the
    found item, in the form (row, col)
 99         If not found it will return false"""
100         # todo - confirm that there will not be duplicate
     tiles
101         for row_idx in range(self.get_rows()):
102             if self.board[row_idx].__contains__(
    target_item):
103                 idx_of_target_item = (row_idx, self.board
    [row_idx].index(target_item)) # todo - does .index return
     the first instance?
104                 return idx_of_target_item
105         return False #  todo - throw error , should I do
    this?
106
107     def list_to_tile_board(self, list_item): # todo:
    check logic
108         idx = 0
109         temp_tile_board = Board(self.get_rows(), self.
    get_cols())  # todo: this needs to become a TileBoard
110         while idx <= len(list_item) -1 :
111             for i in range(self.
    sqrt_total_number_of_tiles):
112                 for j in range(self.
    sqrt_total_number_of_tiles):
113                     temp_tile_board.place(i, j, list_item
    [idx])
```

```
114                     idx += 1
115         return temp_tile_board  # todo: check if this is
    right
116
117     #def get_actions(self):
118         # """This will return a list of possible actions
    that can be called on the board"""
119
120     #def move(self, offset):
121         #  Note: Make sure this is a deep copy, so we don
    't manipulate the pointer
122
123
124
125 print(TileBoard.get_goal_state(24))
126 #test_board = TileBoard(24,[1,2,3,4,None,5,6,7,8])  #
    todo throw an error on this
127 test_tile_board = TileBoard(8, [1, 2, 3, 4, None, 5, 6, 7
    , 8])
128 print(test_tile_board.board)
129 print(test_tile_board.solved())  # apparently not
    implemented
130 #print(test_tile_board.find_item_idx(None)[1])
131 #print(test_tile_board.solvable())
132
133
134
135 ##############################################################
    #############
136 #  Self Note:
137 ##############################################################
    #############
138 d = [[1,2], [1,3], [3,None]]
139 # print(d.find_item_idx(None)) # Won't work because d
    does not have attributes .get_rows
140 test_board = Board(3,3)
141 # print(test_board.find_item_index(None))  #<Board>
    Objects wont work either becuase doesn't contain children
     classes
```