```
# Assignment4.py
# Lizzie Siegle and Sujin Kay

# Implements a program to play the game of Konane (Hawaiian Checkers).

import random
from copy import deepcopy

"""
CREATES THE NODE CLASS (to be used to represent various game states)
"""
class Node:
   def __init__(self, board, move, level, depth_limit, player, who_first):
      "Instance variables."
      self.board = board                          # current board/game state
      self.move = move                            # move that brought you to this state
      self.level = level                          # node at level L in the search tree
      self.depth_limit = depth_limit              # depth limit for searching
      self.player = player                        # which player goes next
      self.who_first = who_first                  # who goes first in the game


"""
BOARD CLASS and FOR BOARD/GAME FUNCTIONALITY
"""
class BOARD:
   def __init__(self, width):
      self.width = width
      self.board = [[' ']*(self.width + 1) for row in range(self.width + 1)]
```

```
"Creates the 8x8 board display, using X for dark pieces and O for light pieces."
def create_board(self, width):
    for row in range(self.width + 1):
        for col in range(self.width + 1):
            # Set board's horizontal and vertical coordinate lines."
            if (row == 0):
                self.board[row][col] = col
                self.board[row][0] = ' '                    # replace coordinate in (0,0) with a blank space

            elif (col == 0):
                self.board[row][col] = row

            # Set board's alternating X's and O's.
            elif ((row + col) % 2 == 0):
                self.board[row][col] = 'X'

            else:
                self.board[row][col] = 'O'

    return self.board



"Prints the board."
def print_board(self, board):
    for row in range(self.width + 1):
        for col in range(self.width + 1):
            print (board[row][col]),                    # print on one line
        print
    print
```

```
"""
FOR GENERATING POSSIBLE MOVES
"""

"Generates all possible first moves."
def possible_first_moves(self):
    first_moves = []

    first_moves.append((1, 1))                                  # top left corner
    first_moves.append((self.width/2, self.width/2))            # middle piece on left side
    first_moves.append((self.width/2+1, self.width/2+1))        # middle piece on right side
    first_moves.append((self.width, self.width))                # bottom right corner

    return first_moves


"Generates all possible second moves."
def possible_second_moves(self, first_move):
    second_moves = []

    # if first move removed top left piece
    if (self.board[1][1] == ' '):
        second_moves.append((1, 2))
        second_moves.append((2, 1))
        return second_moves

    # if first move removed bottom right piece
    elif (self.board[self.width][self.width] == ' '):
        second_moves.append((self.width-1, self.width))
        second_moves.append((self.width, self.width-1))
        return second_moves
```

```
    # if first move removed one of the middle pieces
    else:
        second_moves.append((first_move[0]+1, first_move[1]))
        second_moves.append((first_move[0]-1, first_move[1]))
        second_moves.append((first_move[0], first_move[1]+1))
        second_moves.append((first_move[0], first_move[1]-1))
        return second_moves


"Generates all possible moves, INCLUDING multiple jumps."
def generate_moves(self, board, X_or_O):
    possible_moves = []
    jump_to = (0, 0)
    up = down = left = right = 2

    for row in range(self.width + 1):
        for col in range(self.width + 1):
            "Specify whether looking for Dark/Light moves."
            if (board[row][col] == X_or_O):

                "Can this piece move North?"
                # current position
                current_pos = (row, col)

                # is move within scope of the board?
                while ((current_pos[0] - up) > 0):
                    jump_to = (current_pos[0] - up, col)

                    # is there a blank space where we need to jump, and is there an opponent's piece to jump over?
                    if (board[jump_to[0]][jump_to[1]] == ' ' and board[jump_to[0]+1][jump_to[1]] != ' '):
```

```
        # if so, then append this move
        possible_moves.append((row, col, jump_to[0], jump_to[1]))

        # update how far you'll jump next time (for multiple jumps)
        current_pos = jump_to
    else:
        break


"Can this piece move South?"
# current position
current_pos = (row, col)

# is move within scope of the board?
while ((current_pos[0] + down) < self.width+1):
    jump_to = (current_pos[0] + down, col)

    # is there a blank space where we need to jump, and is there an opponent's piece to jump over?
    if (board[jump_to[0]][jump_to[1]] == ' ' and board[jump_to[0]-1][jump_to[1]] != ' '):

        # if so, then append this move
        possible_moves.append((row, col, jump_to[0], jump_to[1]))

        # update how far you'll jump next time (for multiple jumps)
        current_pos = jump_to
    else:
        break
```

```
"Can this piece move East?"
# current position
current_pos = (row, col)

# is move within scope of the board?
while ((current_pos[1] + right) < self.width+1):
    jump_to = (row, current_pos[1] + right)

    # is there a blank space where we need to jump, and is there an opponent's piece to jump over?
    if (board[jump_to[0]][jump_to[1]] == ' ' and board[jump_to[0]][jump_to[1]-1] != ' '):

        # if so, then append this move
        possible_moves.append((row, col, jump_to[0], jump_to[1]))

        # update how far you'll jump next time (for multiple jumps)
        current_pos = jump_to
    else:
        break


"Can this piece move West?"
# current position
current_pos = (row, col)

# is move within scope of the board?
while ((current_pos[1] - left) > 0):
    jump_to = (row, current_pos[1] - left)

    # is there a blank space where we need to jump, and is there an opponent's piece to jump over?
    if (board[jump_to[0]][jump_to[1]] == ' ' and board[jump_to[0]][jump_to[1]+1] != ' '):
```

```
                # if so, then append this move
                possible_moves.append((row, col, jump_to[0], jump_to[1]))

                # update how far you'll jump next time (for multiple jumps)
                current_pos = jump_to
            else:
                break

    return possible_moves


"""
SPECIAL CASES: FIRST AND SECOND MOVES
"""

def first_second_move(self, user_turn):
    first_moves = self.possible_first_moves()

    # if the user goes first...
    if (user_turn):
        # give instructions
        print "For the first move, only " + str(first_moves[0]) + ", " + str(first_moves[1]) + ", " + str(first_moves[2]) + ", or " +
                str(first_moves[3]) + " allowed."

        # ask for user input
        coord = input("Enter your move, in the form (x, y): ")

        # make sure that input is valid
        while (coord not in first_moves):
            print "Error -- invalid move. Please try again."
            coord = input("Enter your move, in the form (x, y): ")
```

```
    # remove this piece from the board
    print "User removes " + str(coord)
    self.board[coord[0]][coord[1]] = ' '
    self.print_board(self.board)

    # computer goes second, chooses a random legal move
    second_moves = self.possible_second_moves(coord)
    coord2 = random.choice(second_moves)

    # remove this piece from the board
    print "Computer removes " + str(coord2)
    self.board[coord2[0]][coord2[1]] = ' '
    self.print_board(self.board)

    return self.board

# if the computer goes first...
else:
    # choose a random legal move
    coord = random.choice(first_moves)

    # remove this piece from the board
    print "Computer removes " + str(coord)
    self.board[coord[0]][coord[1]] = ' '
    self.print_board(self.board)

    # give user instructions
    second_moves = self.possible_second_moves(coord)
    print "For the second move, can only remove a piece ajacent to first move."
```

```
        # ask for user input
        coord2 = input("Enter your move, in the form (x, y): ")

        # make sure that input is valid
        while (coord2 not in second_moves):
            print "Error -- invalid move. Please try again."
            coord2 = input("Enter your move, in the form (x, y): ")

        # remove this piece from the board
        print "User removes " + str(coord2)
        self.board[coord2[0]][coord2[1]] = ' '
        self.print_board(self.board)

        return self.board



"""
FOR GETTING AND MAKING MOVES
"""

"Asks the User for their move, or decided the best move for the Computer."
def get_move(self, user_turn, possible_moves, who_first):
    # if it's the user's move...
    if (user_turn):
        # ask for user input -- needs TWO coordinates
        coordinates = input("Enter your move, in the form (x, y, x2, y2): ")

        # make sure that input is valid
        while (coordinates not in possible_moves):
            print "Error -- invalid move. Please try again."
```

```
            coordinates = input("Enter your move, in the form (x, y, x2, y2): ")

        return coordinates


    # if it's the computer's move...
    else:
        "RandomPlayer: chooses a random move out of possible legal moves."
        #best_move = random.choice(possible_moves)

        "SmartPlayer: chooses the best move using minimax and alphabeta pruning."
        if who_first == 'User':
            first_node = Node(self.board, None, 0, 4, 'O', who_first)          # 4 is the depth limit
        else:
            first_node = Node(self.board, None, 0, 4, 'X', who_first)

        bv_move = self.minimax_alpha_beta(first_node, float('-inf'), float('inf'))
        best_move = bv_move[1]

        return best_move


"Plays the move on the game board."
def make_move(self, move, X_or_O, to_print):
    # only print if moves are actually being made on game board (not for creating game boards for successor moves)
    if (to_print):
        if (X_or_O == 'X'):
            print "Dark moves (" + str(move[0]) + ", " + str(move[1]) + ") to (" + str(move[2]) + ", " + str(move[3]) + ")"
        else:
            print "Light moves (" + str(move[0]) + ", " + str(move[1]) + ") to (" + str(move[2]) + ", " + str(move[3]) + ")"
```

```
# if the move is horizontal (coordinates are in the same row)
if (move[0] == move[2]):
    # remove your original piece
    self.board[move[0]][move[1]] = ' '

    # each time you jump, remove the opponent's piece that youre jumping over
    current_col = move[1]

    # if you're going East...
    if (move[3] > move[1]):
        while (current_col < move[3]):
            # remove the opponent's piece from the board
            self.board[move[0]][current_col+1] = ' '
            current_col += 2

    # if you're going West...
    else:
        while (current_col > move[3]):
            # remove the opponent's piece from the board
            self.board[move[0]][current_col-1] = ' '
            current_col -= 2

    # insert your jumping piece to the final spot
    self.board[move[2]][move[3]] = X_or_O

    if (to_print == True):
        self.print_board(self.board)

    return self.board
```

```
# if the move is vertical (coordinates are in the same column)
else:
    # remove your original piece
    self.board[move[0]][move[1]] = ' '

    # each time you jump, remove the opponent's piece that youre jumping over
    current_row = move[0]

    # if you're going North...
    if (move[0] > move[2]):
        while (current_row > move[2]):
            # remove the opponent's piece from the board
            self.board[current_row-1][move[1]] = ' '
            current_row -= 2

    # if you're going South...
    else :
        while (current_row < move[2]):
            # remove the opponent's piece from the board
            self.board[current_row+1][move[1]] = ' '
            current_row += 2

    self.board[move[2]][move[3]] = X_or_O

    if (to_print == True):
        self.print_board(self.board)

    return self.board
```

```
"""
MINIMAX AND STATIC EVALUATIONS
"""

"Generates all successor nodes for a current board game state."
def generate_successor_nodes(self, board, current_node):
    successor_nodes = []

    # generate all possible moves for the opponent
    possible_successor_moves = self.generate_moves(board, current_node.player)

    # for each possible move...
    for move in possible_successor_moves:
        # create a copy of the current game board
        current_state = deepcopy(self)

        # make the move on the copy of the board
        current_state.make_move(move, current_node.player, False)

        # create new node containing this new board, depending on who is the next player
        if current_node.player == 'X':
            successor_node = Node(current_state.board, move, current_node.level+1, current_node.depth_limit, 'O',
                                            current_node.who_first)
            successor_nodes.append(successor_node)

        else:
            successor_node = Node(current_state.board, move, current_node.level+1, current_node.depth_limit, 'X',
                                            current_node.who_first)
            successor_nodes.append(successor_node)

    return successor_nodes
```

```
"Static evaluation function for possible moves (#our_moves - #opponent's_moves)."
def static_eval(self, node):
    score = 0

    # figure out which piece corresponds to which player
    if node.who_first == 'User':
        user_piece = 'X'
        computer_piece = 'O'
    else:
        user_piece = 'O'
        computer_piece = 'X'

    # count the number of moves the computer can make
    successor_moves = self.generate_moves(node.board, computer_piece)
    num_moves = len(successor_moves)

    # count the number of moves the user can make
    opp_successor_moves = self.generate_moves(node.board, user_piece)
    num_opponent_moves = len(opp_successor_moves)

    # calculate score (computer's possible moves - user's possible moves)
    score = num_moves - num_opponent_moves
    return score


"Minimax -- returns the best move as defined by the static evaluation function."
def minimax(self, node):
    # if node is at depth limit...
    if (node.level == node.depth_limit):
        # do a static evaluation, return result and the best move
        return (self.static_eval(node), node.move)
```

```
# generate successor nodes
successor_nodes = self.generate_successor_nodes(node.board, node)

# if node is at a maximizing level (if level is even)
if (node.level % 2 == 0):
    best_move = ()
    cbv = float("-inf")

    # for each successor node, call minimax recursively
    for successor in successor_nodes:
        bv_move = self.minimax(successor)

        # look for the highest bv
        if bv_move[0] > cbv:
            cbv = bv_move[0]
            best_move = successor.move

    return (cbv, best_move)

# if node is at a minimizing level (if level is odd)
else:
    best_move = ()
    cbv = float("inf")

    # for each successor node, call minimax recursively
    for successor in successor_nodes:
        bv_move = self.minimax(successor)

        # look for the lowest bv
        if bv_move[0] < cbv:
```

```
            cbv = bv_move[0]
            best_move = successor.move

        return (cbv, best_move)


"Minimax with alpha-beta pruning."
def minimax_alpha_beta(self, node, A, B):
    # if node is at depth limit
    if (node.level == node.depth_limit):
        # do a static evaluation, return result and the best move
        return (self.static_eval(node), node.move)

    # generate successor nodes
    successor_nodes = self.generate_successor_nodes(node.board, node)

    # if node is at a maximizing level (if level is even)
    if (node.level % 2 == 0):
        best_move = ()

        # for each successor node, call minimax recursively
        for successor in successor_nodes:
            bv_move = self.minimax_alpha_beta(successor, A, B)

            if bv_move[0] > A:
                A = bv_move[0]
                best_move = successor.move
            if A >= B:
                return (B, best_move)

        return (A, best_move)
```

```
    # if node is at a minimizing level (if level is odd)
    else:
        best_move = ()

        # for each successor node, call minimax recursively
        for successor in successor_nodes:
            bv_move = self.minimax_alpha_beta(successor, A, B)
            if bv_move[0] < B:
                B = bv_move[0]
            if B <= A:
                return (A, best_move)

        return (B, best_move)




"""
PLAY THE GAME
"""

def play_game(width):
    "Initialize the board game."
    board = BOARD(width)
    board.create_board(width)

    winner = None               # identifies the winner
    user_piece = None           # is User X or O?
    computer_piece = None       # is Computer X or O?
    user_turn = None            # keeps track of whose turn it is
```

```
"Decide who goes first."
who_first = raw_input("Who goes first? (Enter 'User' or 'Computer'): ")

if (who_first == 'User'):
   print "User goes first (User is 'X', Computer is 'O')."
   user_piece = 'X'
   computer_piece = 'O'
   user_turn = True

else:
   print "Computer goes first (Computer is 'X', User is 'O')."
   user_piece = 'O'
   computer_piece = 'X'
   user_turn = False


"Play the first and second moves."
if (user_turn):
   board.first_second_move(user_turn)
   user_turn = True

else:
   board.first_second_move(user_turn)
   user_turn = False


"Start playing!"
# while the game is not over...
while (winner == None):
   # if it's the user's turn...
```

```
if (user_turn):
   # generate all possible moves for the user
   possible_moves = board.generate_moves(board.board, user_piece)

   # check if there's a winner (if no moves generates, opponent wins)
   if (possible_moves == []):
      winner = 'Computer'
      break

   # get the move from the user
   print "User's turn."
   user_move = board.get_move(user_turn, possible_moves, who_first)

   # make move on the board
   board.make_move(user_move, user_piece, True)
   user_turn = False

# if it's the computer's turn...
else:
   # generate all possible moves for the user
   possible_moves = board.generate_moves(board.board, computer_piece)

   # check if there's a winner (if no moves generates, opponent wins)
   if (possible_moves == []):
      winner = 'User'
      break

   # get the move from the computer
   print "Computer's turn."
   computer_move = board.get_move(user_turn, possible_moves, who_first)
```

```
        # make move on the board
        board.make_move(computer_move, computer_piece, True)
        user_turn = True


    "Congratulate the winner and end the game."
    print winner + " won! Game over."


"Call the play_game() function."
play_game(8)
```