This file is split into four parts: to play the game, testing, extensibility, and game backend

TO PLAY THE GAME

Type the following code in the console to begin a game of Battleship in the console:

```
b = BattleShip()
b.__main__()
```

Player 1 then Player 2 will be prompted to fill in the coordinates of the ships to place on their boards and the game will begin. Note that the start and end coordinates of each ship are inclusive. For example, a Destroyer that is aligned horizontally and begins at (0,0) would end at (0,1). At each player's turn, he/she can type the following four actions:

(1) `attack`
The player will then be prompted to input the coordinates of the board cell to attack and the player's turn will end.

(2) `personal_board`
The player can display his/her own board and see the board cells that contain ships and have been hit, missed, or sunk.

(3) `opponent_board`
The player can display the opponent's board and see which cells have been hit, missed, or sunk but cannot see the locations of the opponent's ships.

(4) `quit`
The game will end.

TESTING

First type:

```
b = BattleShip()
b.__main__()
```

Test that both player 1 and player 2 can place pieces on a board. After each ship is placed on the board, the current state of the board will be displayed. Below is an example of what the console will look like after inputting coordinates for the destroyer and cruiser:

```
Player 1 please enter the locations of your ships. The start and end points are inclusive.
DESTROYER start point:
row: 0
col: 0

DESTROYER end point:
row: 0
col: 1

1 1 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
```

```
0 0 0 0 0 0 0 0 0 0

CRUISER start point:
row: 1
col: 0

CRUISER end point:
row: 1
col: 1

1 1 0 0 0 0 0 0 0 0
2 2 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
```

TEST CASE for both players – enter the numbers in order for both player 1 and player 2:
0 0 0 1 1 0 1 1 2 0 2 2 3 0 3 3 4 0 4 4

This sequence of numbers will place the pieces in the following arrangement:

- Destroyer: (0,0) to (0,1)
- Cruiser: (1,0) to (1,1)
- Submarine: (2,0) to (2,2)
- Battleship: (3,0) to (3,3)
- Carrier: (4,0) to (4,4)

Test that players can view their personal boards and opponents' boards. Type `personal_board` or `opponent_board when prompted with 'attack personal_board opponent_board quit'`

Test that players can quit the game at any time. Type `quit` when prompted with `'attack personal_board opponent_board quit'`

Test that both player 1 and player 2 can miss when attacking the opponent. The player must first type 'attack' and then the coordinates of the board cell to attack. Below is an example of the output and input format:

```
Player 1 move.

attack personal_board opponent_board quit: attack
Enter attack location.

x_coord: 6
y_coord: 6

miss
```

TEST CASE for both players – enter the numbers and words in order for player 1 and player 2:
attack 6 6 (player 1 attacks (6,6) => 'miss')
attack 9 9 (player 1 attacks (9,9) => 'miss')

Test that both player 1 and player 2 can hit and sink ships of the opponent and that the game will end correctly. The player must type 'attack' each time. Below is an example of the output and input format:

```
Player 1 move.
```

```
attack personal_board opponent_board quit: attack
Enter attack location.

x_coord: 0
y_coord: 0

hit
```

TEST CASE for both players – enter the numbers in the left column in order for player 1 and player 2:

attack 0 0	(player 1 attacks (0,0) => 'hit')
attack 0 0	(player 2 attacks (0,0) => 'hit)'
attack 0 1	(player 1 attacks (0,1) => 'sunk: destroyer')
attack 0 1	(player 2 attacks (0,1) => 'sunk: destroyer')
attack 1 0	(player 1 attacks (1,0) => 'hit')
attack 1 0	(player 2 attacks (1,0) => 'hit')
attack 1 1	(player 1 attacks (1,1) => 'sunk: cruiser)
attack 1 1	(player 2 attacks (1,1) => 'sunk: cruiser')
attack 2 0	(player 1 attacks (2,0) => 'hit')
attack 2 0	(player 2 attacks (2,0) => 'hit')
attack 2 1	(player 1 attacks (2,1) => 'hit')
attack 2 1	(player 2 attacks (2,1) => 'hit')
attack 2 2	(player 1 attacks (2,2) => 'sunk: submarine')
attack 2 2	(player 2 attacks (2,2) => 'sunk: submarine')
attack 3 0	(player 1 attacks (3,0) => 'hit')
attack 3 0	(player 2 attacks (3,0) => 'hit')
attack 3 1	(player 1 attacks (3,1) => 'hit')
attack 3 1	(player 2 attacks (3,1) => 'hit')
attack 3 2	(player 1 attacks (3,2) => 'hit')
attack 3 2	(player 2 attacks (3,2) => 'hit')
attack 3 3	(player 1 attacks (3,3) => 'sunk: battleship')
attack 3 3	(player 2 attacks (3,3) => 'sunk: battleship')
attack 4 0	(player 1 attacks (4,0) => 'hit')
attack 4 0	(player 2 attacks (4,0) => 'hit')
attack 4 1	(player 1 attacks (4,1) => 'hit')
attack 4 1	(player 2 attacks (4,1) => 'hit')
attack 4 2	(player 1 attacks (4,2) => 'hit')
attack 4 2	(player 2 attacks (4,2) => 'hit')
attack 4 3	(player 1 attacks (4,3) => 'hit')
attack 4 3	(player 2 attacks (4,3) => 'hit')
attack 4 4	(player 1 attacks (4,4) => 'Game over: Player 1 wins!')

This sequence of numbers will accomplish the following:

Player 1 attacks and misses (6,6)
Player 2 attacks and misses (9,9)

Exceptions are thrown when:

- Entered start and endpoints of a ship yield an invalid piece size
- Entered start and endpoints of a ship result in a diagonal orientation
- Entered start and endpoints of a ship cause it to overlap with another ship that has already been placed on the board
- Entered coordinates of a cell to be attacked are out of bounds
- Entered coordinates of a cell to be attacked point to a cell that has been attacked before.

EXTENSIBILITY

Potential ways to extend the scope of this implementation of Battleship

- Create a new piece, e.g. 2X2 Patrol piece
    - Create a Patrol class that is the subclass of Piece and modify the constructor __init__ to specify how to take in start and endpoints (e.g. top left and bottom right coordinates), when an exception is thrown for an invalid size given by the parameters, and define the ID and name instance variables.
    - In the BattleShip class __main__() method, add 'patrol' to the list of ship names and Patrol((start coordinates), (end coordinates), size) to the list of Piece objects.
    - In the place_piece method of PersonalBoard, add an additional if statement so that a Patrol piece is created when given the name 'patrol'
- 1931 *Salvo* edition: players target a specified number of squares at one time and all of the squares are attacked simultaneously
    - Modify the __main__ method of the BattleShip class so that multiple attack cells can be inputted every time a player attacks the opponent.
- Variant where players don't need to announce that a ship has sunk
    - Modify the opponent_move method in the PersonalBoard class so that when the opponent scores a hit, only 'hit' can be returned and not 'sunk: (ship name).'
    - Modify the print_board_opp_pov method so that only 'H', 'M', and '0' are displayed to signify hit, missed, and untouched cells.
- Variant where instead of announcing whether a shot is a hit or miss immediately, players say how many of the opponent's past three shots were hits
    - Modify the print_board_opp_pov() method in BattleShip class so that only some of the hits on the board are displayed every three attacks made by the opponent.

GAME BACKEND:

An outline of the backend for the game with comments for clarity. Green text represents block comments.

```
'''
Piece is the parent class of all the unique types of ships in the game and
keeps track of its start and end points on a matrix, the cells that it
occupies on a matrix, the number of cells of its occupied cells that have been
hit, and its size.
'''
class Piece(object):
    '''
    params:
        start: tuple with board coordinates
        end: tuple with board coordinates

    Initialize a Piece object with the following instance variables:
        self.start: tuple with board coordinates
        self.end: tuple with board coordinates
        self.occupied_cells: dictionary where the keys are tuples that represent
                             all board cells the Piece occupies & values are initialized
                             to 0 and set to 'H' when the cell is hit
        self.hit_cells: number of cells hit by the opponent, initialized to 0
        self.size: the size of the ship

    exception:
        if the start and end coordinates do not form the correct piece size;
        if the start and end coordinates are not on the same horizontal
        or vertical line
    '''
    def __init__(self, start, end, size):
        #IMPLEMENTATION

    '''
    params:
        cell: tuple of board coordinates
```

```
    precondition: the cell that the opponent has fired at is a hit

    This method indicates that the cell has been hit by setting the value at the
    corresponding key of self.occupied_cells to 'H'. This method also increments
    self.hit_cells by 1 to indicate that a cell occupied by the piece has been
    hit
    '''
    def hit(self, cell):
        #IMPLEMENTATION

    '''
    This method returns true if all cells that the piece occupies have been hit;
    otherwise, return false.

    return: boolean
    '''
    def is_sunk(self):
        #IMPLEMENTATION

'''
subclass of Piece

size specification: 5

additional instance variables:
    ID number: 5
    name: carrier
'''
class Carrier(Piece):
    '''
    params:
        start: tuple with board coordinates
        end: tuple with board coordinates

    same method as super class __init__(start, end) method

    additional functions:
        set the following variables:
            ID = 5
            name = 'carrier'
        throw exception: if the start and end coordinates do not form
                                    a piece of size 5
    '''
    def __init__(self, start, end, size = 5):
        #IMPLEMENTATION

'''
subclass of Piece

size specification: 4

additional instance variables:
    ID number: 4
    name: battleship
'''
class Battleship(Piece):
    '''
    params:
        start: tuple with board coordinates
        end: tuple with board coordinates

    same method as super class __init__(start, end) method

    additional functions:
        set the following variables:
            ID = 4
            name = 'battleship'
        throw exception: if the start and end coordinates do not form
```

```
                        a piece of size 4
    '''
    def __init__(self, start, end, size = 4):
        #IMPLEMENTATION

'''
subclass of Piece

size specification: 3

additional instance variables:
    ID number: 3
    name: submarine
'''
class Submarine(Piece):
    '''
    params:
        start: tuple with board coordinates
        end: tuple with board coordinates

    same method as super class __init__(start, end) method

    additional functions:
        set the following variables:
            ID = 3
            name = 'submarine'
        throw exception: if the start and end coordinates do not form
                        a piece of size 3
    '''
    def __init__(self, start, end, size = 3):
        #IMPLEMENTATION

'''
subclass of Piece

size specification: 2

additional instance variables:
    ID number: 2
    name: 'cruiser'
'''
class Cruiser(Piece):
    '''
    params:
        start: tuple with board coordinates
        end: tuple with board coordinates

    same method as super class __init__(start, end) method

    additional functions:
        set the following variables:
            ID = 2
            name = 'cruiser'
        throw exception: if the start and end coordinates do not form
                        a piece of size 2
    '''
    def __init__(self, start, end, size = 2):
        #IMPLEMENTATION

'''
subclass of Piece

size specification: 2

additional instance variables:
    ID number: 1
    name: 'destroyer'
'''
```

```
class Destroyer(Piece):
    '''
    params:
        start: tuple with board coordinates
        end: tuple with board coordinates

    same method as super class __init__(start, end) method

    additional functions:
        set the following variables:
            ID = 1
            name = 'destroyer'
        throw exception: if the start and end coordinates do not form
                         a piece of size 2
    '''
    def __init__(self, start, end, size = 2):
        #IMPLEMENTATION

'''
PersonalBoard represents each player's game board as a 10X10 matrix and keeps
track of all the pieces in the game, where the pieces are in the board, what
board cells have been attacked (denoted by 'M' for miss, 'H' for hit, and
'S' for sunk), and number of ships that have been sunk.
'''
class PersonalBoard(object):
    '''
    Initialize a PersonalBoard object with the following instance variables:
        self.board: 10x10 matrix with default values set to 0
        self.pieces: dictionary where keys are piece names and values are the
                     corresponding Piece objects on the board
        self.sunk_ships: number of ships sunk on the board
    '''
    def __init__(self):
        #IMPLEMENTATION

    '''
    params:
        start: tuple with board coordinates
        end: tuple with board coordinates
        name: the name of the Piece

    This function adds a Piece (as specified by the name parameter)
    to self.pieces where the key is the name of the ship and the value is the
    Piece, and places the Piece on the PersonalBoard by filling the cells the
    Piece occupies of self.board with the ID number of the Piece.

    exception:
        if any of the cells that the Piece occupies is out of bounds
        if any of the cells that the Piece occupies overlaps with another Piece
    '''
    def place_piece(self, start, end, name):
        #IMPLEMENTATION

    '''
    params:
        cells: tuple with board coordinates
        ships: list of names of the Pieces on the board

    This method generates a response after the opponent has attacked a
    specificed cell on the PerosnalBoard. If the board cell has value 0 it
    means that there is no ship on that board cell, so 'miss' is returned and
    the board cell is marked with a 'M'. If the board cell has a 'S', 'M', or
    'H', an exception is raised since that board cell has already been
    attacked before. Otherwise, the board cell contains a nonzero number that
    is the ID of a Piece on the PersonalBoard, so the ship with that ID has been
    hit. This method then checks to see if that hit has caused the ship to sink,
    in which case all of the occupied cells of the ship are marked with S,
    the number of sunk ships increments by 1, and 'sunk: ' + (name of the ship)
```

```
    is returned. Otherwise, the ship doesn't sink and the board cell is marked
    with a 'H' and 'hit' is returned.

    exception:
        if the cell being attacked is out of bounds
        if the cell being attacked has been attacked before

    return: string
    '''
    def opponent_move(self, cell, ships):
        #IMPLEMENTATION

    '''
    This method checks if the game is over by comparing the number of
    sunk ships with the number of pieces initially placed on the board.

    return: boolean
    '''
    def game_over(self):
        #IMPLEMENTATION

'''
Battleship executes the game by prompting players to enter input, processing
the input, outputting feedback (e.g. displaying a player's personal board and the
opponent's board), and ending the game when all of the ships of
a player have sunk.
'''
class BattleShip(object):
    '''
    This initializes two PersonalBoards, one for player 1 and the other for
    player 2
    '''
    def __init__(self):
        #IMPLEMENTATION

    '''
    params:
        board: a matrix to print

    This method prints the board of a player to reveal where all the ships
    have been placed and what cells have been missed, hit, or sunk.
    '''
    def print_board(self, board):
        #IMPLEMENTATION

    '''
    params:
        board: a matrix to print

    This method prints the board from the point of view of the opponent. Ship
    locations are not revealed since the ID numbers are printed as zeros but
    cells that have been missed, hit, or sunk are displayed.
    '''
    def print_board_opp_pov(self, board):
        #IMPLEMENTATION

    '''
    params:
        player: a PersonalBoard object
        ships: list of names of the Pieces on the board

    This method iterates through all the types of ships, collects user
    input on the rows and columns of the start and endpoints of each
    ship, calls the place_piece function to put the ship on the board, and
    calls the print_board function to show the current state of the board
    and where the pieces are located.
    '''
    def collect_input(self, player, ships):
```

```
    #IMPLEMENTATION

'''
This method executes the game and is user-interactive. It is split into four
parts

(1) default settings
    An array of ship names is initiazlied and indexed by 1.
(2) user input for ship placement
    Both players are prompted to place their ships on their boards and the
    collect_input function is called
(3) game play
    The game keeps running until one player sinks all the ships of another
    player. An index that is incremented with each player's turn keeps
    track of whose turn it is (even --> player 1, odd --> player 2). A
    player's turn doesn't end until they miss when they attack. At each
    turn, a player can choose the following functions
        attack: collects user input on which board cell to attack and calls
                the opponent_move function
        personal_board: displays the player's own board by calling print_board
        opponent_board: displays opponent's board with hidden ship locations
                        by calling print_board_opp_pov
        quit: quits the game and returns 'Game over'

(4) game is over
    The player who has won the game is displayed.
'''
def __main__(self):
    #IMPLEMENTATION
```