Homework Turnin

Name:	Alexander A Miller
Email:	alexandermiller@email.arizona.edu
Section:	1E
Course:	CS 120 17au
Assignment:	hw6long
Receipt ID:	d511c761b1cf892f119c8a69cb5b7708

Turnin Successful!

The following file(s) were received:

rhymes-oo.py (8371 bytes)

```
"""
File: rhymes-oo.py
Author: Alexander Miller
Purpose:
* use input() to read in pronunciation dictionary and 'word'
* collect all words that rhyme with 'word'
    * given multiple pronunciations, find all words that rhyme \
    with each pronunciation
* program is CASE INSENSITIVE
* print out each word one per line in any order
* error handling:
    * pronunciation dictionary cannot be read: give error message and quit
        * "ERROR: Could not open file '' + filename
        * USE TRY
    * input word is not in pronunciation dictionary: give \
    error message and quit
        * "ERROR: the word input by the user is not in\
        the pronunciation dictionary'' + word
        * USE ASSERT

"""
import sys

class Word:
    """
    Description: instances have names (the word itself) as well as phonemes\
    saved to them
    """
    def __init__(self, line_list, p_stress_location):
        """
        Purpose: initializes instance of Word
        Parameters: line_list, p_stress_location
        Pre-Condition: line has been cleaned and p_stress_location found
        Post-Condition: Word instance will have name (word itself)\
        in upper case, and its corresponding phonemes
        """
        self._name = line_list[0].upper()
        self._harmonizing_phonemes = [line_list[p_stress_location:]]
        self._discordant_phoneme = [line_list[p_stress_location-1]]
        self._status = 'single'
# getters
    def get_name(self):
        """
        Purpose: returns name
        Returns: self._name
        """
```

```
        return self._name
    def harmonizing_phonemes(self):
        """
        Purpose: returns the harmonizing phonemes
        Returns: self._harmonizing_phonemes
        """
        return self._harmonizing_phonemes
    def discordant_phoneme(self):
        """
        Purpose: returns the discordant phoneme
        Returns: self._discordant_phoneme
        """
        return self._discordant_phoneme
    def status(self):
        """
        Purpose: reports status (multi vs single pronunciation)
        Returns: self._status
        """
        return self._status
# setters
    def convert_to_multi_pronunciation(self):
        """
        Purpose: converts status from single to multi (pronunciation)
        Returns: none
        """
        self._status = 'multi'

    def add_pronunciation(self, line_list, p_stress_location):
        """
        Purpose: adds new phonemes (multiple pronunciations)
        Parameters: line_list, p_stress_location
        """
        self._harmonizing_phonemes += [line_list[p_stress_location:]]
        self._discordant_phoneme += [line_list[p_stress_location-1]]
#misc
    def __str__(self):
        """
        Purpose: gives print instructions (name)
        Returns: self._name
        """
        return self._name
    def __eq__(self, other):
        """
        Purpose: compares two Word instances by comparing their phonemes \
        and returns if they rhyme (perfectly) or not
        Parameters: other
        Returns: True or None
        Post-Condition: established: words either rhyme perfectly or they don't
        """
        if other.get_name() != self.get_name(): # eliminate rhyming with itself
            self_iteration = 0
            other_iteration = 0
            while other_iteration != len(other.discordant_phoneme()):
                while self_iteration != len(self.discordant_phoneme()):
                    if (self.harmonizing_phonemes()[self_iteration] == \
                        other.harmonizing_phonemes()[other_iteration]) and\
                        (self.discordant_phoneme()[self_iteration] != \
                        other.discordant_phoneme()[other_iteration]):
                        return True
                    self_iteration += 1
                other_iteration += 1


class WordMap:
    """
    Description: instances read in files from input and initialize Word\
    instances and make comparisons between words based on user input
    """
    def __init__(self,name):
        """
        Purpose: initalizes WordMap instance
        Parameters: name
        """
        self._name = name
        self._word_tuple = ()
# getters
```

```
    def get_word_tuple(self):
        """
        Purpose: calls collection of Word instances
        Returns: self._word_tuple
        """
        return self._word_tuple
# setters
    def add(self,word_name):
        """
        Purpose: adds a Word instance to the word collection
        Parameters: word_name
        """
        self._word_tuple += (word_name,)
# misc
    def read_file(self):
        """
        Purpose: reads, cleans, and validates phoneme dicitonary,\
        finds primary stress, creates Word instances, adds multiple\
        pronunciation
        """
        filename = input()
        # objective: input file validation
        try:
            openfile = open(filename)
        except:
            print('ERROR: Could not open file' + filename)
            sys.exit(1)
        # objective: read in data from file
        ### ASSUMPTION: file is formatted correctly with good data
        for line in openfile:
            line = line.strip()
            # get rid of empty lines
            if line == '':
                continue
            line_list = line.split()
            # handling comment lines
            if line_list[0] == '#':
                continue
            line_list = line.split()
            # find and count primary stresses
            p_stress_counter = 0
            index = -1
            for i in line_list:
                index += 1
                if '1' in i:
                    p_stress_counter += 1
                    p_stress_location = index
            # objective: only create words w/ singular primary stress
            if p_stress_counter == 1:
                word_name = line_list[0]
                word_name = word_name.upper()
                # handle single pronunciation
                count = 0
                for x in self.get_word_tuple():
                    if x.get_name() == word_name:
                        count += 1
                if count == 0:
                    word_name = Word(line_list, p_stress_location)
                    self.add(word_name)
                else: # handle multi
                    for x in self.get_word_tuple():
                        if word_name == x.get_name():
                            x.convert_to_multi_pronunciation()
                            x.add_pronunciation(line_list, p_stress_location)
        openfile.close()
    def find_rhyming_words(self,x):
        """
        Purpose: finds rhyming words
        Parameters: x
        Post-Condition: rhyming words have been found and printed;\
        program terminates
        """
        # objective: find rhyming words
        success_list = []
        for i in self.get_word_tuple():
            if x == i:
```

```
                success_list += [i]
        for i in success_list:
            print(i)

def main():
    """
    Description: mission control
    """
    word_map = initialize_word_map_and_read_file()
    process_word_input(word_map)



def initialize_word_map_and_read_file():
    """
    Purpose: initializes WordMap instance and reads file (via WordMap instances),\
    initializes and adds Word instances to WordMap instance
    Returns: word_map
    Post-Condition: WordMap instance is fully initialized, file is read and\
    processed, words have been added to WordMap
    """
    name = 'word_map'
    word_map = WordMap(name)
    word_map.read_file()
    return word_map

def process_word_input(word_map):
    """
    Purpose: staging center for taking in user input, and comparing it with\
    WordMap instance's collection of Word objects and printing results
    Parameters: word_map
    Pre-Condition: WordMap and its elements fully initialized
    Post-Condition: program is finished
    """
    input_word = input()
    input_word = input_word.upper()
    # objective: verify that input word is in database
    for x in word_map.get_word_tuple():
        if input_word == x.get_name():
            break
    # assert trips if no object of the same name was found in word tuple
    assert input_word == x.get_name(), \
           'ERROR: the word input by the user is not in the dictionary ' + input_word
    word_map.find_rhyming_words(x)



main()
```

battleship.py (17577 bytes)

```
"""
File: battleship.py
Author: Alexander Miller
Purpose:
* read in player one's ship placements from input file  and initialize board
* read in player two's guesses from input file and respond based on effects of guess
    * if guess is not legal, give error message 'illegal guess', discard, and continue
    * 'miss' or 'miss (again)' vs. 'hit' and 'hit (again)'
    * print(''{} sunk''.format(ship)) ; print(''all ships sunk: game over'')
    * program will also terminate if there are no more guesses to respond to
* ships: A,B,S,D,P
* guesses of the format '(a,b) to (c,d)'
* possible errors:
    * placement file or guess file cannot be read
        * response: error message and quit
    * placement file improperly formatted (too many/few ships, multiple placements,\
    invalid coordinates, etc.)
        * response: error message and quit
    * guess not located on Board
        * response: error message 'illegal guess', ignore, and continue processing
"""
```

```
import math
import sys

class GridPos:
    """
    Description: instances describe grid position and contain information\
    about ships and guesses
    """
    def __init__(self, grid_pos, grid_pos_name):
        """
        Purpose: initializes GridPos instance
        Parameters: grid_pos, grid_pos_name
        Post-Condition: GridPos instance has name, position, ship\
        guess, and count attributes
        """
        self._name = grid_pos_name
        self._grid_position = grid_pos
        self._ship = None
        self._guess = 0
        self._update_count = 0
# getters
    def get_ship(self):
        """
        Purpose: way to get ship at that position
        Returns: self._ship
        """
        return self._ship
    def get_update_count(self):
        """
        Purpose: way to get update_count
        Returns: self._update_count
        """
        return self._update_count
    def guess(self):
        """
        Purpose: way to get number of guesses at that location
        Returns: self._guess
        """
        return self._guess
# setters
    def update_guess(self):
        """
        Purpose: way to get update number of guesses at that location
        """
        self._guess += 1
    def update_ship(self, ship_name):
        """
        Purpose: way to place ship at that position
        Parameters: ship_name
        """
        self._ship = ship_name
    def increment_update_count(self):
        """
        Purpose: increment update_count
        """
        self._update_count += 1
# misc
    def __str__(self):
        """
        Purpose: print order for GridPos object
        Returns: self._name
        """
        return self._name
    def __eq__(self, resp_grid_pos):
        """
        Purpose: provides way to compare positions based on name
        Parameters: resp_grid_pos
        Returns: self._name == resp_grid_pos
        """
        return self._name == resp_grid_pos



class Board:
    """
    Description: instances describe a 'board' containing ships (objects) and a\
```

```
    grid of GridPos objects
    """
    def __init__(self):
        """
        Purpose: initialize instance of Board
        Post-Condition: Board instance will have containers for\
        ships and a grid
        """
        self._ship_record = []
        self._grid = []
# getters
    def get_ship_record(self):
        """
        Purpose: get ship record
        Returns: self._ship_record
        """
        return self._ship_record
    def get_grid(self):
        """
        Purpose: get grid
        Returns: self._grid
        """
        return self._grid
# setters
    def update_grid(self, grid_pos_name):
        """
        Purpose: adds grid positions to grid
        Parameters: grid_pos_name
        """
        self._grid += [grid_pos_name]
    def update_ship(self, ship_name):
        """
        Purpose: adds ship name to ship record
        Parameters: ship_name
        """
        self._ship_record += [ship_name]
# misc
    def __len__(self):
        """
        Purpose: allows for a way to look at the "length" of board\
        in this case really the number of ships on the board
        Returns: len(self._ship_record)
        """
        return len(self._ship_record)


class Ship:
    """
    Description: instances represent 'ships' and contain information about\
    ship position, length, integrity, sunk/floating
    """
    def __init__(self, ship_name, ship_position, ship_length):
        """
        Purpose: initializes instance of ship
        Parameters: ship_name, ship_position, ship_length
        Post-Condition: ship instance will have name, position,\
        length, integrity, and sunk/floating attributes
        """
        self._name = ship_name
        self._position = ship_position
        self._length = ship_length
        self._integrity = ship_length
        self._sunk = 0
# getters
    def get_name(self):
        """
        Purpose: get name/type of ship
        Returns: self._name
        """
        return self._name
    def get_pos(self):
        """
        Purpose: get position of ship
        Returns: self._position
        """
        return self._position
```

```
    def integrity(self):
        """
        Purpose: get integrity of ship (shots sustained vs. health)
        Returns: self._integrity
        """
        return self._integrity
    def sunk(self):
        """
        Purpose: allows user to find out if the ship has sunk or not
        Returns: self._sunk
        """
        return self._sunk
# setters
    def got_hit(self):
        """
        Purpose: drops ship integrity in event of successful (first-time)\
        hit
        """
        self._integrity -= 1
    def got_sunk(self):
        """
        Purpose: updates ship status to 'sunk'
        """
        self._sunk = 1
# misc
    def __str__(self):
        """
        Purpose: gives print definition to ship (name)
        Returns: self._name
        """
        return self._name
    def __eq__(self, other):
        """
        Purpose: provides manner of comparison (name matches string)
        Returns: self._name == other
        """
        return self._name == other


def main():
    """
    Purpose: mission control
    """
    board = establish_grid()
    board = placement_processing(board)
    guess_processing(board)



def establish_grid():
    """
    Purpose: initialize board, establish grid of gridpos objects,\
    and adjoin to board
    Returns: board
    Post-Condition: board will be initialized and contain grid of \
    gridpos objects
    """
    accum2 = 0
    board = Board()
    while accum2 < 10:
        accum = 0
        while accum < 10:
            x_coord = accum
            y_coord = accum2
            grid_pos = [[x_coord, y_coord]]
            grid_pos_name = str(x_coord)+str(y_coord)
            grid_pos_name = GridPos(grid_pos, grid_pos_name)
            board.update_grid(grid_pos_name)
            accum += 1
        accum2 +=1
    return board



##### NOTE: START OF SUBFUNCTIONS FOR PROCEDURE_PROCESSING #####
```

```
# SUBFUNCTION 1
def endpoints_and_alignment(line_list):
    """
    *Note: component of placement_processing
    Purpose: organize start and endpoints logically \
    and establish vertical vs. horizontal alignment
    Parameters: line_list
    Returns: x1, x2, y1, y2, horizontal_alignment, vertical_alignment
    Pre-Condition: line_list has been acquired from file and is in good form
    Post-Condition: alignments established, coordinates refined and returned,\
    improperly aligned boats trip program quit
    """
    horizontal_alignment = 0
    vertical_alignment = 0
    # horizontal case:
    if int(line_list[1]) == int(line_list[3]):
        vertical_alignment = 1
        x1 = int(line_list[1])
        x2 = int(line_list[3])
        if int(line_list[2]) < int(line_list[4]):
            y1 = int(line_list[2])
            y2 = int(line_list[4])
        else:
            y2 = int(line_list[2])
            y1 = int(line_list[4])
    # vertical case:
    elif int(line_list[2]) == int(line_list[4]):
        horizontal_alignment = 1
        y1 = int(line_list[2])
        y2 = int(line_list[4])
        if int(line_list[1]) < int(line_list[3]):
            x1 = int(line_list[1])
            x2 = int(line_list[3])
        else:
            x2 = int(line_list[1])
            x1 = int(line_list[3])
    # neither vertical nor horizontal:
    else:
        print('ERROR: ship not horizontal or vertical')
        sys.exit(1)
    return x1,x2,y1,y2,horizontal_alignment, vertical_alignment

#SUBFUNCTION 2
def board_position_validation(x1,x2,y1,y2):
    """
    *Note: component of placement_processing function
    Purpose: validate that placements are on board
    Parameters: x1, x2, y1, y2
    Pre-Condition: coordinates have been properly organized by Subfunction 1
    Post-Condition: any illegal ship placements (off-the-board) have caused\
    a program quit
    """
    try:
        assert x1 < 10 and x1 >= 0
        assert x2 < 10 and x2 >= 0
        assert y1 < 10 and y1 >= 0
        assert y2 < 10 and y2 >= 0
    except:
        print('ERROR: ship out-of-bounds')
        sys.exit(1)

# SUBFUNCTION 3
def ship_overlap_validation(x1,x2,y1,y2,horizontal_alignment\
                            ,vertical_alignment,board,ship_name):
    """
    *Note: component of placement_processing function
    Purpose: update grid_pos with ships and validate overlapping ships
    Parameters: x1, x2, y1, y2, horizontal_alignment, vertical_alignment,\
    board, ship_name
    Returns: board
    Pre-Condition: alignments established and coordinates organized by\
    Subfunction 1
    Post-Condition: any overlapping ships have caused the program to quit and\
    the GridPos instances have been updated to hold ships
    """
```

```
    if horizontal_alignment == 1:
        for x in range(x1,x2+1):
            resp_grid_pos = str(x)+str(y2)
            for i in board.get_grid():
                if i == resp_grid_pos:
                    i.update_ship(ship_name)
                    i.increment_update_count()
                    if i.get_update_count() > 1:
                        print('ERROR: overlapping ship')
                        sys.exit(1)
    else:
        for y in range(y1,y2+1):
            resp_grid_pos = str(x1)+str(y)
            for i in board.get_grid():
                if i == resp_grid_pos:
                    i.update_ship(ship_name)
                    i.increment_update_count()
                    if i.get_update_count() > 1:
                        print('ERROR: overlapping ship')
                        sys.exit(1)
    return board

#SUBFUNCTION 4
def ship_length_and_type_validation(ship_name,ship_length):
    """
    *Note: component of placement_processing function
    Purpose: validate for proper ship length and type
    Parameters: ship_name, ship_length
    Pre-Condition: ship_length calculated using math import and \
    ship_name established
    Post-Condition: ships of 'wrong type' and wrong length cause program\
    to quit
    """
    type_error = 0
    try:
        if ship_name == 'A':
            assert ship_length == 5
        elif ship_name == 'B':
            assert ship_length == 4
        elif ship_name == 'S':
            assert ship_length == 3
        elif ship_name == 'D':
            assert ship_length == 3
        elif ship_name == 'P':
            assert ship_length == 2
        else:
            type_error = 1
            assert 1 == 0 # placeholder to trip except
    except:
        if type_error == 0:
            print('ERROR: incorrect ship size')
            sys.exit(1)
        else:
            print('ERROR: fleet composition incorrect')
            sys.exit(1)

#SUBFUNCTION 5
def validate_number_of_ships(board):
    """
    *Note: component of placement_processing function
    Purpose: validate number of ships
    Parameters: board
    Post-Conditions: improper fleet size causes program to quit
    """
    # objective: ensure that proper number of ships exist on Board
    try:
        assert len(board) == 5
    except:
        print('ERROR: fleet composition incorrect')
        sys.exit(1)

##### NOTE: END OF SUBFUNCTIONS FOR PLACEMENT_PROCESSING #####



def placement_processing(board):
```

```
    """
    *Note: makes use of plenty of subfunctions
    Purpose: read in from placement file and validate restrictions \
    using subfunctions
    Parameters: board
    Returns: board
    Post-Conditions: all placement file validations have been executed\
    with program only continuing if all prereqs have been met.
    Board, ship, and grid pos fully updated prior to guesses
    """
    placement_file = input()
    # objective: validate file input
    try:
        openfile = open(placement_file)
    except:
        print('ERROR: Could not open file ' + placement_file)
        sys.exit(1)
    # objective: process placement file
    for line in openfile:
        line = line.strip()
        # get rid of empty lines
        if line == '':
            continue
        line_list = line.split()
        # handling comment lines
        if line_list[0] == '#':
            continue
        for x in line_list:
            x = x.strip()
        ### ASSUMPTION: line is in correct format, split by whitespace
        try:
            assert len(line_list) == 5
        except:
            print('ERROR: fleet composition incorrect')
            sys.exit(1)
        ship_name = line_list[0]
        ship_name = ship_name.upper()

        # SUBFUNCTION 1
        # objective: logical organization of start and endpoints\
        # verification of horizontal/vertical ships
        x1,x2,y1,y2,horizontal_alignment,vertical_alignment = \
                endpoints_and_alignment(line_list)

        # SUBFUNCTION 2
        # objective: validate for positions on board
        board_position_validation(x1,x2,y1,y2)

        # objective: establish ship_position and ship_length
        ship_position = [[x1,y1],[x2,y2]]
        # note: adding 1 to account for first slot being included in length
        ship_length = int(math.sqrt((y2-y1)*(y2-y1)+(x2-x1)*(x2-x1)) + 1)

        # SUBFUNCTION 3
        # objective: update grid_pos with ships and validate overlapping ships
        board = ship_overlap_validation(x1,x2,y1,y2,\
                horizontal_alignment,vertical_alignment,board, ship_name)

        # SUBFUNCTION 4
        # objective: validate for proper ship length and type
        ship_length_and_type_validation(ship_name,ship_length)

        # objective: create ship object and put it on the Board
        ship_name = Ship(ship_name, ship_position, ship_length)
        board.update_ship(ship_name)
    # SUBFUNCTION 5
    # objective: validate number of ships
    validate_number_of_ships(board)
    openfile.close()
    return board



def guess_processing(board):
    """
    Purpose: validates guess file and determines if player wins/loses
```

```
    Parameters: board
    Pre-Condition: placement_processing has ensured that ships, gridpos,\
    and board have been prepared
    Post-Condition: player wins or loses
    """
    guess_file = input()
    # objective: validate file input
    try:
        openfile = open(guess_file)
    except:
        print('ERROR: Could not open file ' + guess_file)
        sys.exit(1)
    # objective: process guess_file data
    openfile = open(guess_file)
    sinks_total = 0
    for line in openfile:
        line = line.strip()
        # get rid of empty lines
        if line == '':
            continue
        line_list = line.split()
        # handling comment lines
        if line_list[0] == '#':
            continue
        line_list = line.split()
        ### ASSUMPTION: line is in correct format, split by whitespace
        for x in line_list:
            x = x.strip()
        assert len(line_list) == 2
        line_list[0] = int(line_list[0])
        line_list[1] = int(line_list[1])
        # objective: validate input coordinates
        try:
            assert line_list[0] >= 0
            assert line_list[0] < 10
            assert line_list[1] >= 0
            assert line_list[1] < 10
        except:
            print('illegal guess')
            continue
        # objective: update grid with shots and verify with ships
        shot = str(line_list[0])+str(line_list[1])
        for i in board.get_grid():
            if shot == i:
                i.update_guess()
                # get ship object from board record
                for x in board.get_ship_record():
                    if i.get_ship() == x:
                        ship = x
                # handles hits
                if i.get_ship() != None:
                    if i.guess() > 1: # has to be '1' to discount first guess
                        print('hit (again)')
                    else:
                        ship.got_hit()
                        print('hit')
                # handles misses
                else:
                    if i.guess() > 1:
                        print('miss (again)')
                    else:
                        print('miss')
                # objective: determine sunken ship
                if ship.integrity() == 0 and ship.sunk() == 0:
                    sinks_total += 1
                    ship.got_sunk()
                    print('{} sunk'.format(ship.get_name()))
                # objective: check if all ships have sunk
                if sinks_total == 5:
                    print('all ships sunk: game over')
                    return
    openfile.close()


main()
```