Contents

1 Basic Test Results

```
Starting tests...
Wed Jun 10 21:27:03 IDT 2020
a79db6e60ed2678ae8289348ef3efb5dc5013b95  -


Archive:  /tmp/bodek._DNrs4/intro2cs2/ex9/ofirm57/presubmission/submission
  inflating: src/board.py
  inflating: src/car.py
  inflating: src/game.py


Running presubmit code tests...
12 passed tests out of 12 in test set named 'funcnames'.
result_code    funcnames    12    1
16 passed tests out of 16 in test set named 'carbase'.
result_code    carbase    16    1
6 passed tests out of 6 in test set named 'boardbase'.
result_code    boardbase    6    1
Done running presubmit code tests

Finished running the presubmit tests

Additional notes:

The presubmit tests check only for the existence of the correct function names.
Make sure to thoroughly test your code.

```

2 board.py

```python
#################################################################
# FILE : ex9.py
# WRITER :ofir  , ofirm57 , 205660731
# EXERCISE : intro2cs2 ex9 2020
#################################################################



class Board:
    """
    Add a class description here.
    Write briefly about the purpose of the class
    """
    SIDE = 7
    EXIT_TARGET = (3, 7)

    def __init__(self):
        self.__side = Board.SIDE
        self.__cars = {}


    def __str__(self):
        """
        This function is called when a board object is to be printed.
        :return: A string of the current status of the board
        """
        board_str = ''
        bord_mat = self.get_mat_bord()
        for i, j in enumerate(bord_mat):
            if i == Board.EXIT_TARGET[0]:
                board_str += '$' + " ".join(j) + '→ \n'
            else:
                board_str += '$' + " ".join(j) + '$\n'
        return board_str


    def cell_list(self):
        """ This function returns the coordinates of cells in this board
        :return: list of coordinates
        """
        #In this board, returns a list containing the cells in the square
        #from (0,0) to (6,6) and the target cell (3,7)
        lst_of_cor = []
        side = self.__side
        for i in range(side):
            for j in range(side):
                lst_of_cor.append((i, j))
        lst_of_cor.append(Board.EXIT_TARGET)
        return lst_of_cor


    def possible_moves(self):
        """ This function returns the legal moves of all cars in this board
        :return: list of tuples of the form (name,movekey,description)
                 representing legal moves
        """
        # full_places - list with the tupls whos not free
        full_places = self.cars_board_coordinates()
        cell_list = self.cell_list()
        move_lst = []
        for car in self.__cars:
            car_dict_move = self.__cars[car].possible_moves()
            keys = car_dict_move.items()
            for movekey in keys:
                mast_be_empty = self.__cars[car].movement_requirements(movekey[0]) #
                if mast_be_empty[0] not in cell_list:
                    continue
                if mast_be_empty[0] in full_places:
                    continue
                form_tup = (car, movekey[0],  movekey[1])
                move_lst.append(form_tup)
        return move_lst


    def target_location(self):
        """
        This function returns the coordinates of the location which is to be filled for victory.
        :return: (row,col) of goal location
        """
        return Board.EXIT_TARGET #In this board, returns (3,7)


    def cell_content(self, coordinate):
        """
        Checks if the given coordinates are empty.
        :param coordinate: tuple of (row,col) of the coordinate to check
        :return: The name if the car in coordinate, None if empty
        """
        for car in self.__cars:
            car_coordinates = self.__cars[car].car_coordinates()
            if coordinate in car_coordinates:
                return car
        return


    def add_car(self, car):
        """
        Adds a car to the game.
        :param car: car object of car to add
        :return: True upon success. False if failed
        """
        coor_new_car = car.car_coordinates()
        if not self.cars_board_coordinates():
            contained_coordinates = []
        else:
            contained_coordinates = self.cars_board_coordinates()
        cells = self.cell_list()
        for coordinate in coor_new_car:
            if coordinate in contained_coordinates:
                return False
            if coordinate not in cells:
                return False
        if car.get_name() in self.name_board_cars(): #repeat names check
            return False
        if not car.possible_moves():
            return False # orientation check
        self.__cars[car.get_name()] = car
        return True


    def move_car(self, name, movekey):
        """
        moves car one step in given direction.
        :param name: name of the car to move
        :param movekey: Key of move in car to activate
        :return: True upon success, False otherwise
        """
        # move_option form - [('O','d',"some description")..()]
        move_option = self.possible_moves()
        for i in move_option:
            if i[0] == name and i[1] == movekey:
                self.__cars[name].move(movekey)
                return True
        return False


    def cars_board_coordinates(self):
        """:return: list with tappel of all cars coordinates"""
        if not self.__cars:
            return []
        cars_board_coor = []
        for car_in_board in self.__cars:
            car_coordinates = self.__cars[car_in_board].car_coordinates()
            cars_board_coor.extend(car_coordinates)
        return cars_board_coor


    def get_mat_bord(self):
        """:return list of list - the bord game (matrix)"""

        length = self.__side
        the_cars = self.__cars
        bord_lst = []
        for row in range(length):
            bord_lst.append(['_'] * length)# empty mat
        for car in the_cars:
            car_cor = self.__cars[car].car_coordinates() #form[(row,line)..()]
            for i in car_cor:
                bord_lst[i[0]][i[1]] = car # car = name =or ['O' or 'R'...]
        return bord_lst


    def name_board_cars(self):
        """:return the car thet exixt in the bord """
        lst_of_cars_names = []
        for car in self.__cars:
            lst_of_cars_names.append(car)
        return lst_of_cars_names

```

3 car.py

```python
#################################################################
# FILE : ex9.py
# WRITER :ofir  , ofirm57 , 205660731
# EXERCISE : intro2cs2 ex9 2020
#################################################################

HORIZONTAL = 1 # l,r
VERTICAL = 0 # u,d
MOVE_UP = "u"
MOVE_DOWN = "d"
MOVE_LEFT = "l"
MOVE_RIGHT = "r"


class Car:
    """
   This class has all the information about the car and the things that can
    be operated on the car
    """

    def __init__(self, name, length, location, orientation):
        """
        A constructor for a Car object
        :param name: A string representing the car's name
        :param length: A positive int representing the car's length.
        :param location: A tuple representing the car's head (row, col) location
        :param orientation: One of either 0 (VERTICAL) or 1 (HORIZONTAL)
        """
        self.__name = name
        self.__length = length
        self.__location = location
        self.__orientation = orientation



    def car_coordinates(self):
        """
        :return: A list of coordinates the car is in
        """
        car_coordinat = [self.__location]
        row_num = car_coordinat[0][0]
        cole_num = car_coordinat[0][1]
        length = self.__length

        if self.__orientation == HORIZONTAL:
            for col_cord in range(length - 1):
                car_coordinat.append((row_num, cole_num + col_cord + 1))
            return car_coordinat

        elif self.__orientation == VERTICAL:
            for row_cord in range(length - 1):
                car_coordinat.append((row_num + row_cord + 1, cole_num))
            return car_coordinat


    def possible_moves(self):
        """
        :return: A dictionary of strings describing possible movements permitted by this car.
        """
        if self.__orientation == HORIZONTAL:
            horizontal_movement = {'l': 'move the car left!',
                                   'r': 'move the car right!'}
            return horizontal_movement
        if self.__orientation == VERTICAL:
            vertical_movment = {'u': 'move the car up!',
                                'd': 'move the car down!'}
            return vertical_movment


    def movement_requirements(self, movekey):
        """
        :param movekey: A string representing the key of the required move.
        :return: A list of cell locations which must be empty in order for this move to be legal.
        """

        last_cord = self.car_coordinates()[-1]
        first_cord = self.car_coordinates()[0]
        if movekey == MOVE_DOWN:
            mast_be_empty = [(last_cord[0] + 1, last_cord[1])]
            return mast_be_empty

        elif movekey == MOVE_UP:
            mast_be_empty = [(first_cord[0] - 1, first_cord[1])]
            return mast_be_empty

        elif movekey == MOVE_RIGHT:
            mast_be_empty = [(last_cord[0], last_cord[1] + 1)]
            return mast_be_empty

        elif movekey == MOVE_LEFT:
            mast_be_empty = [(first_cord[0], first_cord[1] - 1)]
            return mast_be_empty


    def move(self, movekey):
        """
        :param movekey: A string representing the key of the required move.
        :return: True upon success, False otherwise
        """

        loc = self.__location
        if self.__orientation == HORIZONTAL: # l,r
            if movekey == MOVE_RIGHT:
                self.__location = (loc[0], loc[1] + 1)
                return True
            elif movekey == MOVE_LEFT:
                self.__location = (loc[0], loc[1] - 1)
                return True
            else:
                return False

        elif self.__orientation == VERTICAL: # u,d
            if movekey == MOVE_UP:
                self.__location = (loc[0] - 1, loc[1])
                return True
            elif movekey == MOVE_DOWN:
                self.__location = (loc[0] + 1, loc[1])
                return True
            else:
                return False
        if movekey not in ['r', 'l', 'u', 'd']:
            return False




    MOVE_UP = "u"
```

```
    MOVE_DOWN = "d"
    MOVE_LEFT = "l"
    MOVE_RIGHT = "r"

    def get_name(self):
        """:return: The name of this car.  """
        return self.__name

```

4 game.py

```python
#################################################################
# FILE : ex9.py
# WRITER :ofir  , ofirm57 , 205660731
# EXERCISE : intro2cs2 ex9 2020
#################################################################
import helper
import sys

MOVE_KEYS = ['u', 'd', 'l', 'r']
CAR_NAMES = ['G', 'B', 'R', 'Y', 'W', 'O']

WELCOME_MSG = 'for exit enter ! \n ' \
              'lets play -\n Please enter car name and movekey:'
EXIT = '!'
MOVE_PROBLEM = 'there was problem whit your move'
NEXT_TURN_MSG = 'next turn'
INCORRECT_INPUT_MSG = 'Incorrect input, try again!'
INCORRECT_VALUES_MSG = 'Incorrect car name or movekey, try again!'
WIN_MSG = 'CONGRATULATIONS YOU WON !!!!'

class Game:
    """
    Add class description here
    """

    def __init__(self, board, game_cars):
        """
        Initialize a new Game object.
        :param board: An object of type board
        """
        self.__board = board
        self.__game_cars = game_cars
        for car in self.__game_cars:
            car_data = self.__game_cars[car]
            c_obj = Car(str(car), car_data[0], tuple(car_data[1]), car_data[2])
            if car in CAR_NAMES: #name check
                if 2 <= car_data[0] or car_data[0] <= 4: #length check
                    if self.__board.cell_content(tuple(car_data[1])) is None:
                        self.__board.add_car(c_obj)


    def __single_turn(self):
        """
        Note - this function is here to guide you and it is *not mandatory*
        to implement it.

        The function runs one round of the game :
            1. Get user's input of: what color car to move, and what
                direction to move it.
            2. Check if the input is valid.
            3. Try moving car according to user's input.

        Before and after every stage of a turn, you may print additional
        information for the user, e.g., printing the board. In particular,
        you may support additional features, (e.g., hints) as long as they
        don't interfere with the API.
        """

        print(self.__board)
        user_choice = input(WELCOME_MSG) #type(user_choice)= str
        if user_choice == EXIT:
            return True
        if len(user_choice) != 3:
            print(INCORRECT_INPUT_MSG)
            return
        the_car, user_movekey = user_choice.split(',')  # enter to variable

        if the_car not in self.__game_cars or \
                user_movekey not in MOVE_KEYS:   #variable iligle
            print(INCORRECT_VALUES_MSG)
            return
        if self.__board.move_car(the_car, user_movekey):
            print(NEXT_TURN_MSG)
        else:
            print(MOVE_PROBLEM)



    def play(self):
        """
        The main driver of the Game. Manages the game until completion.
        :return: None
        """
        target = self.__board.target_location()
        while self.__board.cell_content(target) is None: # if not win
            exit_game = self.__single_turn()
            if exit_game:
                break
        else:
            print(WIN_MSG)



if __name__ == "__main__":

    from board import *
    from car import *

    the_board = Board()
    game = Game(the_board, helper.load_json(sys.argv[1]))
    game.play()
```