Contents

1 Basic Test Results

```
Starting tests...
Tue 09 Jul 2024 23:08:25 IDT
8f9ac5a810a8f8fd60efb41e19dd166fdf6c4c40  -


Archive:  /tmp/bodek.51vlmx5k/intro2cs2/ex9/daniel.rez/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
from typing import Tuple, List, Optional, Dict

from car import Car

Coordinates = Tuple[int, int]


class Board:
    """
    Manages the game board by providing a 7x7 grid with a designated target cell.
    Facilitates operations such as adding cars, moving them based on defined rules, and checking game state conditions.
    Supports functionalities like visualizing the board, identifying legal moves, and verifying position statuses.
    """

    def __init__(self) -> None:
        """
        Initializes a Board object with a fixed size and a target location.
        Also initializes an empty dictionary to store the cars on the board.
        """
        self.__size = 7  # Size of the board (7x7 grid)
        self.__target = (3, 7)  # Target location to be reached for victory
        self.__cars: Dict[str, Car] = {}  # Dictionary to store cars by their names

    @property
    def cars(self):
        return self.__cars

    def target_location(self) -> Coordinates:
        """
        This function returns the coordinates of the location that should be
        filled for victory.
        :return: (row, col) of the goal location.
        """
        return self.__target

    def initialize_board(self):
        """
        Initializes the board with the appropriate size and adds an extra cell to the specific row.
        It populates the board with the positions of the cars.
        :return: A list of lists representing the board.
        """
        board = [['_' for _ in range(self.__size)] for _ in range(self.__size)]

        # Adds an extra cell to the third row to accommodate the target location
        if self.__size > 2:  # Ensures the board has at least 3 rows
            board[3].append('_')

        # Fills the board with car names based on their coordinates.
        for car in self.__cars.values():
            for row, col in car.car_coordinates():
                if 0 <= row < len(board) and 0 <= col < len(board[row]):
                    board[row][col] = car.get_name()

        return board

    def __str__(self) -> str:
        """
        This function is called when a board object is to be printed.
        :return: A string representing the board.
        """
        board = self.initialize_board()

        # Top border
        board_str = '*' * (self.__size * 2 + 2) + '\n'

        # Side borders
        for row in range(len(board)):
            board_str += '*' + ' '.join(board[row]) + ' *\n'

        # Bottom border
        board_str += '*' * (self.__size * 2 + 2) + '\n'

        return board_str

    def cell_list(self) -> List[Coordinates]:
        """
        This function returns the coordinates of cells in this board.
        :return: list of coordinates.
        """
        cells = [(row, col) for row in range(self.__size) for col in range(self.__size)]
        cells.append(self.__target)
        return cells

    def possible_moves(self) -> List[Tuple[str, str, str]]:
        """
        This function returns the legal moves of all cars in this board.
        :return: list of tuples of the form (name, move_key, description)
                 representing legal moves. The description should briefly
                 explain what is the movement represented by move_key.
        """
        moves = []
        for car in self.__cars.values():
            for move_key, description in car.possible_moves().items():
                # Validates that all required cells for the move are free and within board limits
                if all(self.is_within_bounds([req]) and not self.is_occupied(req) for req in
                       car.movement_requirements(move_key)):
                    moves.append((car.get_name(), move_key, description))
        return moves

    def cell_content(self, coordinates: Coordinates) -> Optional[str]:
        """
        Checks if the given coordinates are empty.
        :param coordinates: tuple of (row, col) of the coordinates to check.
        :return: The name of the car in "coordinates", None if it's empty.
        """
        for car in self.__cars.values():
            # Checks if any part of the car occupies the given coordinates
            if coordinates in car.car_coordinates():
                return car.get_name()
        return None

    def add_car(self, car: Car) -> bool:
        """
        Adds a car to the game.
        :param car: car object to add.
        :return: True upon success, False if failed.
        """
        # Checks if any part of the new car's position conflicts with existing cars
        if any(self.is_occupied(coord) for coord in car.car_coordinates()):
            print("Car position is already occupied.")
            return False
        # Ensures the entire car fits within the board boundaries
        if not self.is_within_bounds(car.car_coordinates()):
            print("Car position is out of bounds.")
            return False
        self.__cars[car.get_name()] = car
        return True

    def move_car(self, name: str, move_key: str) -> bool:
        """
        Moves car one step in a given direction.
        :param name: name of the car to move.
        :param move_key: the key of the required move.
        :return: True upon success, False otherwise.
        """
        # Checks if the car exists on the board
        if name not in self.__cars:
            return False
        car = self.__cars[name]
        # Validates move direction against car's orientation (horizontal/vertical)
        if (car.orientation == 0 and move_key in ['l', 'r']) or (car.orientation == 1 and move_key in ['u', 'd']):
            return False
        # Checks move feasibility (path clearance and within bounds)
        if all(self.is_within_bounds([req]) and not self.is_occupied(req) for req in
               car.movement_requirements(move_key)):
            car.move(move_key)
            return True
        return False

    def is_within_bounds(self, coords: List[Coordinates]) -> bool:
        """
        Checks if the given coordinates are within the bounds of the board or the target cell.
        :param coords: A list of tuples representing the coordinates to check.
        :return: True if all coordinates are within bounds, False otherwise.
        """
        # Checks each coordinate pair to see if it lies within the playable area or is the target cell
        return all((0 <= row < self.__size and 0 <= col < self.__size) or (row, col) == self.__target for row, col in coords

    def is_occupied(self, coord: Coordinates) -> bool:
        """
        Checks if a coordinate is occupied by any car.
        :param coord: The coordinate to check.
        :return: True if the coordinate is occupied, False otherwise.
        """
        return self.cell_content(coord) is not None
```

3 car.py

```python
from typing import Tuple, List, Dict

Coordinates = Tuple[int, int]


class Car:
    """
    Represents a car in the Rush Hour game, maintaining attributes such as name, length, and orientation.
    Encapsulates car properties, enforcing valid configurations through property validations.
    Supports movement operations, checking legality based on orientation and predefined rules.
    """
    # Class constants
    __VALID_NAMES = {'Y', 'B', 'O', 'W', 'G', 'R'}
    __MAX_LENGTH = 4
    __MIN_LENGTH = 2
    __VERTICAL = 0
    __HORIZONTAL = 1

    def __init__(self, name: str, length: int, location: Coordinates,
                 orientation: int) -> None:
        """
        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 location (row,col).
        :param orientation: One of either 0 (VERTICAL) or 1 (HORIZONTAL).
        """
        self.__name = name
        self.__length = length
        self.__location = location
        self.__orientation = orientation

    @property
    def name(self) -> str:
        return self.__name

    @name.setter
    def name(self, value: str) -> None:
        if value not in self.__VALID_NAMES:
            raise ValueError(f"Invalid car name: {value}. Valid names are: {self.__VALID_NAMES}")
        self.__name = value

    @property
    def length(self) -> int:
        return self.__length

    @length.setter
    def length(self, value: int) -> None:
        if not (self.__MIN_LENGTH <= value <= self.__MAX_LENGTH):
            raise ValueError(
                f"Invalid car length: {value}. Valid lengths are between {self.__MIN_LENGTH} and {self.__MAX_LENGTH}")
        self.__length = value

    @property
    def location(self) -> Coordinates:
        return self.__location

    @location.setter
    def location(self, value: Coordinates) -> None:
        if not (isinstance(value, tuple) and len(value) == 2 and all(isinstance(coord, int) for coord in value)):
            raise ValueError(f"Invalid location: {value}. Location must be a tuple of two integers.")
        self.__location = value

    @property
    def orientation(self) -> int:
        return self.__orientation

    @orientation.setter
    def orientation(self, value: int) -> None:
        if value not in (self.__VERTICAL, self.__HORIZONTAL):
            raise ValueError(
                f"Invalid orientation: {value}. Valid orientations are {self.__VERTICAL} for vertical and {self.__HORIZONTAL
        self.__orientation = value

    def car_coordinates(self) -> List[Coordinates]:
        """
        :return: A list of coordinates the car is in.
        """
        coordinates = []
        for i in range(self.__length):
            if self.__orientation == self.__VERTICAL:
                coordinates.append((self.__location[0] + i, self.__location[1]))
            else:
                coordinates.append((self.__location[0], self.__location[1] + i))
        return coordinates

    def possible_moves(self) -> Dict[str, str]:
        """
        :return: A dictionary of strings describing possible movements
                 permitted by this car.
        """
        if self.__orientation == self.__VERTICAL:
            return {
                'u': "Move up",
                'd': "Move down"
            }
        else:
            return {
                'l': "Move left",
                'r': 'Move right'
            }

    def movement_requirements(self, move_key: str) -> List[Coordinates]:
        """
        :param move_key: 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.
        """
        if move_key == 'u':
            # The cell above the car's current head must be empty
            return [(self.__location[0] - 1, self.__location[1])]
        elif move_key == 'd':
            # The cell below the car's current tail must be empty
            return [(self.__location[0] + self.__length, self.__location[1])]
        elif move_key == 'l':
            # The cell to the left of the car's current head must be empty
            return [(self.__location[0], self.__location[1] - 1)]
        elif move_key == 'r':
            return [(self.__location[0], self.__location[1] + self.__length)]
        else:
            return []

    def move(self, move_key: str) -> bool:
        """
        This function moves the car.
        :param move_key: A string representing the key of the required move.
        :return: True upon success, False otherwise
        """
        if move_key not in self.possible_moves():
            return False
        if move_key == 'u':
            self.__location = (self.__location[0] - 1, self.__location[1])
        elif move_key == 'd':
            self.__location = (self.__location[0] + 1, self.__location[1])
        elif move_key == 'l':
            self.__location = (self.__location[0], self.__location[1] - 1)
        elif move_key == 'r':
            self.__location = (self.__location[0], self.__location[1] + 1)

        return True

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


```

4 game.py

```python
import sys
from typing import Any, Dict, List, Union

import helper
from board import Board
from car import Car

JsonCoordinates = List[int]
CarConfiguration = List[Union[int, JsonCoordinates]]


class Game:
    """
    Represents a game session of 'Rush Hour', a puzzle game where players move cars on a grid to clear a path for the
    escape vehicle. This class manages game initialization, user interactions, and the game loop until completion.
    """

    def __init__(self, board: Board) -> None:
        """
        Initialize a new Game object.
        :param board: An object of type board
        """
        self.__board = board
        # Controls the continuation of the game loop.
        self.__continue_game = True

    @staticmethod
    def load_configuration(config_file: str) -> Dict[str, CarConfiguration]:
        """
        Loads the car configuration from a JSON file to set up the game board.
        :param config_file: The path to the configuration file.
        :return: A dictionary containing the configuration.
        """
        return helper.load_json(config_file)

    def setup_board(self, config: Dict[str, Any]) -> None:
        """
        This method initializes the board state before the game starts. Function iterates through each car
        configuration provided, creates a Car object, and attempts to place it on the board. If placement fails
        it outputs an error message and continues to place the rest of remaining cars. It also checks
        if the initial board setup results in a victory condition.
        """
        for name, config in config.items():
            length, location, orientation = config
            location_tuple = tuple(location)
            car = Car(name, length, location_tuple, orientation)
            if not self.__board.add_car(car):
                print(f"Failed to add car: {name}")
        if self.check_victory():
            print(self.__board)
            print("Victory! The car has reached the target location immediately after loading the configuration.")
            self.__continue_game = False

    def __single_turn(self):
        """
        Executes a single turn in the game. This method handles user input, validates it, executes moves, and checks
        for game victory.

        During a turn:
```

```
        1. The current state of the board is displayed.
        2. The user is prompted to enter a command to move a car or quit the game.
        3. Input is parsed and validated for correct format and feasibility of the requested move.
        4. If the input is valid, the move is executed on the board, and the board's state is updated.
        5. After the move, the game checks for a victory condition to determine if the game should end.
        """

        print(self.__board)
        user_input = input("Enter the car name and direction (e.g., Y,d) or '!' to quit: ").strip()

        if user_input == '!':
            print("Exiting the game.")
            self.__continue_game = False
            return

        if ',' not in user_input or len(user_input.split(',')) != 2:
            print("Invalid input format. Please use the format 'car_name,direction' (e.g., Y,d).")
            return

        car_name, direction = user_input.split(',')
        car_name = car_name.strip().upper()
        direction = direction.strip().lower()

        if car_name not in self.__board.cars:
            print(f"No car found with the name {car_name}.")
            return

        if direction not in ['u', 'd', 'l', 'r']:
            print("Invalid direction. Use 'u' for up, 'd' for down, 'l' for left, 'r' for right.")
            return

        if not self.__board.move_car(car_name, direction):
            print(f"Move '{direction}' for car '{car_name}' is not valid.")
        else:
            if self.check_victory():
                print(self.__board)
                print("Congratulations! You've won the game.")
                self.__continue_game = False

    def play(self) -> None:
        """
        The main driver of the Game. Manages the game until completion.
        :return: None
        """
        while self.__continue_game:
            self.__single_turn()

    def check_victory(self) -> bool:
        """
        Determines if the victory condition for the game has been met.
        :return: True if the car has reached the target location, False otherwise.

        If any car's coordinates include the target location, returns True, indicating the game has been won.
        """
        for car in self.__board.cars.values():
            coordinates = car.car_coordinates()
            if self.__board.target_location() in coordinates:
                return True
        return False


if __name__ == "__main__":
    # If config proved form command line
    if len(sys.argv) == 2:
        config_file = sys.argv[1]
    else:
        # Prompts the user to enter the configuration file path if not provided as a command line argument.
        config_file = input("Enter the path to the JSON configuration file: ").strip().strip('"')
    # Attempts to load the configuration, handling any errors that occur due to file access issues.

    # Initializes the game board.
    board = Board()
    # Creates a game instance with the initialized board.
    game = Game(board)
    try:
        # Loads the game configuration from the specified file.
        config = game.load_configuration(config_file)
        # Sets up the game board with cars from the loaded configuration.
        game.setup_board(config)
    except FileNotFoundError:
        print(f"Error: File '{config_file}' does not exist or could not be accessed.")
    except Exception as e:
        print(f"An unexpected error occurred: {str(e)}")
        sys.exit(1)

    # If ok,starts the game loop.
    game.play()
```