Now Boarding!

Everyone who travels by plane experiences the feeling at the departure gate that the queue for the plane will not move at all, and surely there must be a quicker way of boarding. In this assignment, you will embark on a search for such a way and compare the established methods. The inspiration for the assignment is the video "The Better Boarding Method Airlines Won't Use" by CGP Grey, available at: https://www.youtube.com/watch?v=oAHbLRjF0vo.

In your solution, you will create a simulation of the boarding process. In each simulation, you will create a model of an airplane that you will gradually fill with passengers. The different boarding methods will be determined by the order in which passengers are added based on their seats. As in reality, each seat will be represented by a pair of values: row (e.g. 12) and seat (e.g. C). For the simplicity of solution, we will assume that each aircraft has seats from A to F (two groups of three seats are placed in each row). We also assume that all seats are to be occupied by passengers.

You will create your simulations and models according to the object-oriented programming paradigm, and your project must include at least three classes:

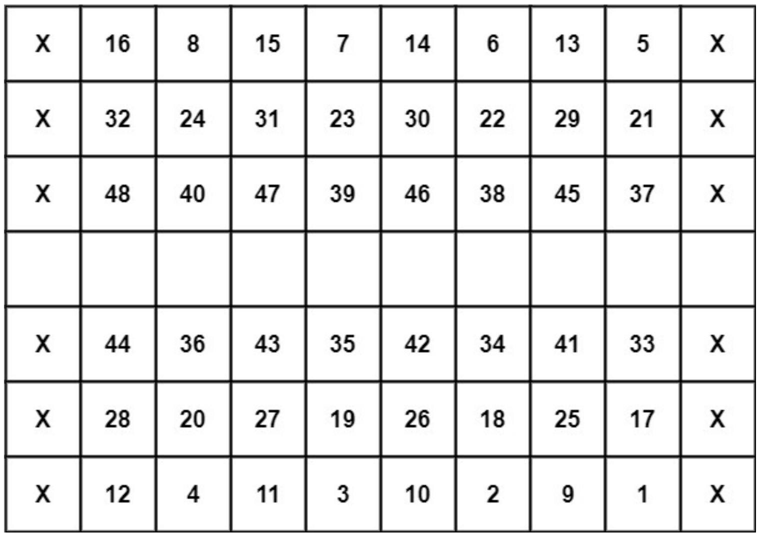
- `Passenger` (in the `passenger.py` file) - the class represents a single passenger with seat and luggage assigned.
- `Plane` (in the `plane.py` file) - the class represents a plane with a specified length. From an implementation point of view, the plane will be represented as a list of lists, e.g. a plane with 8 rows will be represented as follows

X	1F	2F	3F	4F	5F	6F	7F	8F	X
X	1E	2E	3E	4E	5E	6E	7E	8E	X
X	1D	2D	3D	4D	5D	6D	7D	8D	X
X	1C	2C	3C	4C	5C	6C	7C	8C	X
X	1B	2B	3B	4B	5B	6B	7B	8B	X
X	1A	2A	3A	4A	5A	6A	7A	8A	X

  where the middle empty row is the aisle and the two outermost columns (marked X) are just helper columns that simplify the simulation for us. At each position in this field there will be a list of passengers who are at that position. **Note**: when representing an aircraft using a two-dimensional array, the rows are the seats with the same letter, the rows of seats are in columns!
- `Boarding` and subclasses (all in the file `boarding.py`) - the class represents the method of boarding the aircraft. The `Boarding` class defines the general functionality, the subclasses specify a specific way of boarding or adding passengers to the plane.

Within the assignment, you simulate six boarding methods, defining a new `Boarding` subclass for each method:

1. back-to-front - you divide the passengers into four groups based on which row they sit in. You let the passengers from the rearmost group on the plane first (in random order), then the passengers from the second rearmost group, and so on. Passengers sitting in the front are the last to board. You can assume that the length of the plane, i.e. the number of rows, will be divisible by 4.
2. front-to-back - you divide the passengers into four groups based on which row they sit in. You will let the passengers from the frontmost group on the plane first (in random order), then the passengers from the second frontmost group, and so on. Passengers sitting at the back of the plane are the last to board. You can assume that the length of the plane, i.e. the number of rows, will be divisible by 4.
3. window-to-aisle - you divide the passengers into three groups based on where they sit within the three seats (by the window, in the middle, or by the aisle). You will first let in the passengers seated by the window, seats A and F (in random order), then the passengers seated in the middle, seats B and E, and finally the passengers seated by the aisle, seats C and D.
4. aisle-to-window - you divide the passengers into three groups based on where they sit within the three seats (window, middle, or aisle). You will first let the aisle passengers, seats C and D, on the plane (in random order), then the middle passengers, seats B and E, and finally the window passengers, seats A and F.
5. random - you do not divide the passengers into any groups, you let them onto the plane in random order.
6. Steffen's perfect - you let the passengers onto the plane according to the procedure defined by Jason Steffen: from the window to the aisle, every other row from the back, alternating sides. That is, you let the passengers on the plane first in the seats in even rows and in seat A. Then the passengers in the even-numbered rows in seat F board. Next, the passengers in the odd-numbered rows in seat A board. You will finish the boarding of the passengers seated at the window with the group in odd rows in seat F. Boarding continues according to the same rule for seats B and E or C and D:



X	16	8	15	7	14	6	13	5	X
X	32	24	31	23	30	22	29	21	X
X	48	40	47	39	46	38	45	37	X
X	44	36	43	35	42	34	41	33	X
X	28	20	27	19	26	18	25	17	X
X	12	4	11	3	10	2	9	1	X

Class `Passenger` (3 points)

In the file `passenger.py` you will find the code skeleton of the `Passenger` class.
The class constructor has three parameters: `row` (row number), `seat` (passenger seat letter) and `no_of_bags` (number of bags). The class has the following internal variables:
- `self.row` - the number of the row where the passenger has a seat (parameter `row`)
- `self.seat` - seat letter (parameter `seat`)
- `self.bags` - expresses the number of steps the passenger needs to take to put his/her luggage in the overhead bin (parameter `no_of_bags * 4`)
- `self.plane` - pointer to the aircraft the passenger is on; set to `None` in the constructor
- self.current_position - expresses the current position of the passenger on the plane, set to `[None, None]` in the constructor

Your task is to complete the functionality as follows:

`get_position(self)` - the method returns the passenger's position in an array (row and column in the list of lists) that represents the aircraft the passenger is in. If the passenger has not yet been added to the aircraft, the method returns `None`.
Attention: the row number of the passenger's seat does not match the row number in the array that represents the aircraft!

`get_seat(self)` – the method returns the passenger's seat position on the plane (row and column in the list of lists).
Attention: the row number of the passenger's seat does not match the row number in the array that represents the aircraft!

`add_to_plane(self, plane)` – the method adds a passenger to the plane. The method has one additional parameter: a pointer to the plane to which we want to add the passenger. In the method, update the value of the internal variable `plane` and the `current_position` number pair to `[0, 3]` (0 represents the zeroth row of seats, 3 is the aisle - the index of the column and row in the two-dimensional array representing the plane).

`can_sit(self)` – the method determines if the passenger has a clear path to his seat if he is already standing in the given row. If the passenger has a seat next to the aisle, the method always returns `True`, otherwise it returns `True` if the seats between the passenger's seat and the aisle are not occupied (e.g.: if the passenger wants to sit in 4E but someone is sitting in 4D, the method returns `False`). The function returns `False` if the passenger has not yet been added to the aircraft or is not standing in the corridor.

`forced_to_move(self, x, y)` – the method represents the forced movement of a passenger (the passenger himself does not want to move, but it is necessary because he is in the way of someone else). The method has two additional parameters - `x` and `y` - that represent the new position of the passenger in the plane. In the method you have to update the value of the `current_position` pair.
Be careful about the correct representation of the current position!

`move(self)` – the method represents the movement of a passenger in the aircraft. If the passenger has not yet been added to the aircraft, the function raises a `TypeError`. If the passenger is on the plane, the method always returns two values - the passenger's new position (seat row and seat or aisle numbering) We can describe the passenger's movement with a few rules:

1. until the passenger is in his seat row, he stays in the corridor and always takes one step forward if the next position is vacant
2. if the passenger is already in the row of his seat, he first stores his luggage (decrease the value of `self.bags` by 1 until it reaches 0, the passenger stays in his position)
3. if the passenger is already in the row of his seat and has no luggage, he will check if he has a clear path to his seat. If so, he sits down; if not, he asks other passengers to make way for him (see `move_row` and `return_row` in the `Plane` class).

Class `Plane` (3 points)

In the file `plane.py` you will find the code skeleton of the `Plane` class.
The class constructor has one parameter: `length` (number of rows in the plane). The class has the following internal variables:
- `self.length` - number of rows in the plane (`length` parameter)
- `self.seats` - a two-dimensional list of lists, where each row represents seats with the same letter and each column represents one row of seats. At each position of this array there is a list of passengers that are in that position.

Your task is to complete the functionality as follows:

`print_plane(self)` – the method prints a simple representation of the plane on the screen. (The method serves as a guide for you, it will not be evaluated).

`add_passengers(self, psg_list)` – the method will add passengers from the list to the plane. It takes one parameter, `psg_list`, which is a list with passengers. For each passenger, call the method from the `Passenger` class to add it to the plane, and then add the passenger to the list at position `[3, 0]` as well (reverse order of the `Passenger` class's position representation).

`is_empty(self, row, seat)` – the method returns `True` if the list at position `[seat, row]` is empty, `False` otherwise. If the given position does not exist, returns `True`.

`move_row(self, row, seat_letter)` – this method is used to move passengers from their seats in order to clear the way for the next passenger to their seat. The method takes two parameters - the row and the letter of the seat the passenger wants to sit in. The method first detects whether the position `[seat, row + 1]` is vacant, and if so, moves each seated passenger to that position (adds them to the appropriate list). If the position is not free, the method does nothing, and the new passenger must continue to wait.

`return_row(self, row)` - the method is the opposite of the `move_row` method, i.e. it returns all passengers from position `[3, row + 1]` to their seats in one step if they are to sit in the given row.

`move_passengers(self)` – the method moves all passengers in the aisle and updates their position in the two-dimensional array. Use the move method from the `Passenger` class to get the new passenger position. To avoid deadlocks, start with updating the position of the passengers at the end of the plane (the last place in the corridor).

`boarding_finished(self)` - the method will detect if the boarding is finished. Returns `True` if there are no more passengers in the corridor and all seats are occupied. Otherwise, it returns `False`.

Class `Boarding` (1 point + 0.5 points for each subclass)

In the file `boarding.py` you will find the skeleton code of the `Boarding` class.
The class constructor has no parameters, the class has one internal variable:
- `self.plane` - pointer to the plane, set to `None` in the constructor

Your task is to complete the functionality in the following way:

`generate_boarding(self, plane_length)` – the method will generate a boarding case in the given way. In the main `Boarding` class this is an empty method, in subclasses it should generate passengers according to the boarding method rules (see above) and add them to the plane (call the `add_passengers` method from `Plane`). When generating passengers, always first create each group of passengers according to their location, then randomly shuffle the group of passengers, and only then add them to the plane. The method has one parameter, namely the length of the plane for which you want to create a simulation.

`run_simulation(self, plane_length)` – the method runs the simulation of passengers boarding the plane. The method takes one parameter, namely the length of the plane (number of rows) for which you want to create the simulation. In the method you have to generate the passenger boarding order (using `generate_boarding`). The boarding is done by moving the passengers again until the boarding is complete - all seats are occupied. The method returns a single value, the number of steps needed to complete the boarding. Implement the method only in the `Boarding` class.

`test_boarding_method(self, plane_length, no_simulation)` – the method runs several simulations of boarding by the given method. The method has two parameters: `plane_length` (number of rows in the plane) and `no_simulation` (number of simulations). The method returns two values: the average number of steps required to complete the boarding, and a list of the results of each simulation. The method is implemented directly in the `Boarding` class.

Later implement the `generate_boarding` methods in the subclasses `BoardingFTB` (front to back), `BoardingBTF` (back to front), `BoardingWTA` (window to aisle), `BoardingATW` (aisle to window), `BoardingRandom` (completely random order), `BoardingSteffen` (Steffen's perfect).