Assignment 1

Assignment 1: Racoon Raiders!

Due date: Friday, March 11, 2022 before 8:00 pm sharp, Toronto time.

You may complete this assignment individually or with ONE partner. Follow **these instructions (https://github.com/MarkUsProject/Markus/wiki/Student-Guide#how-to-form-group)** to declare your group on MarkUs. Please review the section of the course syllabus on **working with a partner. (https://q.utoronto.ca/courses/249810#assignmentpolicies)**

The **Assignment 1 FAQ on Piazza (https://piazza.com/class/ky50y49v8002n5?cid=1003)** is required reading. Please check it for any updates each time you sit down to work on this assignment.

Learning goals

By the end of this assignment you should be able to:

- read complex code you didn't write and understand its design and implementation, including:
  - reading the class and method docstrings carefully (including attributes, representation invariants, preconditions, etc.)
  - understanding relationships between classes, by applying your knowledge of composition and inheritance.
- complete a partial implementation of a class, including:
  - reading the representation invariants to enforce important facts about implementation decisions.
  - reading the preconditions to factor in assumptions that they permit.
  - writing the required methods according to their docstrings.
- Make design decisions about how to implement code to conform to a specified interface.
- Write code that implements algorithms that are somewhat involved to reason about.
- Confidently work with code involving both inheritance and composition.

You should also have continued to develop these habits:

- Implementing tests for a method before implementing the method.
- Running your test suite throughout the development of your code.

Please read this handout carefully and ask questions if there are any steps you are unsure about.

The assignment involves code that is larger and more complex than Assignment 0. If you find it difficult to understand at first, that is normal – we are stretching you to do more challenging things. Expect that you will need to read carefully, and that you will need to read things multiple times.

General coding guidelines

These guidelines are designed to help you write well-designed code that adheres to the interfaces we have defined (and thus will be able to pass our test cases).

For class `GameBoard`, all added attributes must be private. You may create new public methods in the `GameBoard` class if you find it useful to do so, but do NOT add public methods to any other classes.

All code you write must be consistent with the provided code, docstrings, and the provided `a1_game.py` client code.

All child classes must correctly implement the interface defined by their abstract parent classes.

Overall, you must NOT change any function and method interfaces that we provided in the starter code. In particular, do **NOT**:

- change the interface (parameters, parameter type annotations, or return types) to any of the methods or functions you have been given in the starter code.
- change the type annotations of any public or private attributes you have been given in the starter code.

In addition to the above, you must **NOT**:

- add any more import statements to your code, except for imports from the `typing` module.
- change the inheritance hierarchy.
- mutate an object in a method or a function if the docstring doesn't say that it will be mutated.

On the other hand, you are welcome to:

- remove unused imports from the Typing module.
- create new private helper methods for the classes you have been given.
  - if you do create new private methods, you must provide type annotations for every parameter and return value. You must also write a full docstring for such methods, as described in the **function design recipe (https://www.teach.cs.toronto.edu/~csc148h/winter/notes/python-recap/function_design_recipe.pdf)**.
- create new private attributes for the classes you have been given.
  - if you do create new private attributes you must give them a type annotation and include a description of them in the class's docstring as described in the **class design recipe (https://www.teach.cs.toronto.edu/~csc148h/winter/notes/object-oriented-programming/class_design_recipe.pdf)**.

While writing your code, you may assume that all arguments passed to the methods and functions you have been given in the starter code will respect the preconditions and type annotations outlined in the starter code.

All code that you write should follow the function design recipe and the class design recipe.

Introduction

In this assignment, you will develop a game that is somewhat similar to **Rodent's Revenge (https://en.wikipedia.org/wiki/Rodent%27s_Revenge)** .

During the game, raccoons try to climb into one of the available garbage cans. The main player aims to prevent the raccoons from reaching the garbage cans by trapping them using recycling bins.

Here is a little demo of how the game will look once you finish the assignment:

The game takes place on a grid whose width and height are both measured in tiles (also called squares). In the example above, the grid has width 8 and height 6.

The origin is at the top-left corner. The values on the x-axis increase from left to right. The values on the y-axis increase from top to bottom. Below is an example of the tile locations for a game board whose width and height are both 6.

(0, 0)	(1, 0)	(2, 0)	(3, 0)	(4, 0)	(5, 0)
(0, 1)	(1, 1)	(2, 1)	(3, 1)	(4, 1)	(5, 1)
(0, 2)	(1, 2)	(2, 2)	(3, 2)	(4, 2)	(5, 2)
(0, 3)	(1, 3)	(2, 3)	(3, 3)	(4, 3)	(5, 3)
(0, 4)	(1, 4)	(2, 4)	(3, 4)	(4, 4)	(5, 4)
(0, 5)	(1, 5)	(2, 5)	(3, 5)	(4, 5)	(5, 5)

Each character takes up exactly one tile, except that a raccoon can crawl inside an open garbage can, in which case there will be two characters (a raccoon and a garbage can) on the same tile.

Some characters are movable (the main player, raccoons, and recycling bins) while others are static (garbage cans). Some of the movable characters can take turns (the main player and raccoons). Each turn-taking character can move exactly one tile on its turn, in one of the following directions: up, down, left or right. Raccoons are given turns less frequently than the main player, as can be seen in the example gameplay above.

When the game starts, the player is placed on the top-left corner. Raccoons, garbage cans and recycling bins are placed randomly on the board. Two types of raccoons exist in the game: regular raccoons and smart raccoons (with glasses), that move randomly or smartly respectively, aiming to crawl inside the garbage cans.

When a raccoon reaches an unoccupied, unlocked garbage can, it crawls inside and stays there. If the garbage can is locked, the raccoon will use up its turn to unlock it.

The player's objective is to prevent the raccoons from getting into the garbage cans by trapping them with recycling bins, while trying to keep as many recycling bins clumped together as possible. We want to trap the raccoons, but we do not want to leave recycling bins all over the neighbourhood in doing so!

The player moves using the four arrow keys and pushes around the recycling bins in an attempt to trap the raccoons.

Recycling bins can't move autonomously but can be pushed by the main player in the hope of trapping the raccoons. When pushed by the main player, a recycling bin moves in the same direction as the player. If the new tile happens to be occupied with another recycling bin, this recycling bin is also moved in the same direction as well. If there is a row or column of recycling bins being pushed, they all move. Here is a little example.

A raccoon is considered trapped if it is surrounded in all four directions (diagonals don't matter) by recycling bins, other raccoons (including ones in garbage cans), the player, or the border of the game board.

The game ends once all raccoons are either trapped or inside garbage cans.

When the game ends, your score is displayed. The score is determined based on the number of trapped raccoons, as well as the maximum number of adjacent recycling bins.

Note that sometimes you may get into a game state where it will become impossible for the game to end (i.e. you can't possibly trap the remaining raccoons). You will just need to close the game window. In this case, you never see a score.

The program

The program for this assignment consists of the following classes in `a1.py`:

- `Character`: An abstract class that represents any character in the game. All characters must define `move` and `get_char` methods.
- `TurnTaker`: An abstract class that inherits from `Character` that represents any character in the game that can take a turn. All turn-takers must define a `take_turn` method in addition to the `Character` abstract methods.
- `Raccoon`: A class representing a raccoon that moves around randomly.
- `SmartRaccoon`: A class representing a raccoon that moves in a less random way. This is drawn as a raccoon with glasses.
- `GarbageCan`: A class representing a garbage can that a raccoon can climb into. A garbage can cannot be moved around. It is either locked or unlocked; a player can lock it, but a raccoon can open it.
- `RecyclingBin`: A class representing a recycling bin that a raccoon cannot climb into, but a player can move around to trap the raccoons.
- `Player`: The player that the user can move around via the arrow keys on their keyboard.
- `GameBoard`: The game board that keeps track of the game objects. In this board, there is at most one item on each tile, with this exception: A raccoon can climb into an unlocked garbage can.

The required public interfaces to every class, method, and function have been designed, but we have left some private implementation decisions up to you.

The file `a1_game.py` is client code for the code you complete in `a1.py` and handles the actual playing of the game. You do NOT need to write any code in `a1_game.py`, but you can run it as you complete the assignment to see your progress towards the working game!

Your tasks

We will guide you through a sequence of tasks, in order to gradually build the pieces of your implementation. Note that the pieces are laid out in logical order, not in order of difficulty.

Save **a1.zip (https://q.utoronto.ca/courses/249810/files/19450177?wrap=1)** ↓ **(https://q.utoronto.ca/courses/249810/files/19450177/download?download_frd=1)** to your computer and

extract its contents. Copy all the extracted files and folders into your `csc148/assignments/a1` folder. The zip file includes:

- `a1.py`: Starter code that you will complete and **submit**.
- `a1_game.py`: the interface used to run the game. You do not have to read or understand the code in this file. Do not modify this file except to change the numbers of raccoons, cans, and bins if you want to. You can complete the entire assignment without running this module, but you may find it is a more fun way to debug at times.
- `a1_starter_tests.py`: Some basic tests cases that you should add to in order to test your own code.
- some other files containing images for the gameplay.

Each part of the code that you need to write is marked with a comment saying "TODO". Remove the "TODO" comments as you complete the code and make sure you remove them all before your final submission.

Your tasks are listed below.

Task 0:

Read the code so that you have an understanding of:

a. the overall architecture of the code including the use of inheritance,

b. what every method does,

c. the choices we've made so far for how to represent that data in each class, and

d. how the various classes have composition relationships.

Tip: Use the Structure tab in PyCharm to help you more easily navigate through the various classes and their methods.

Task 1:

We'll start by having you decide how you want to store the game's characters in the `GameBoard` class. Decide based on what you will need in order to implement the following methods according to their docstrings:

- `place_character`
- `at`
- `to_grid`
- `__str__`

Then create appropriate private attributes in the `GameBoard` class to carry out your plan.

Note: Make sure to carefully read the docstrings for `Character.__init__` and `place_character`, as they are closely related. In particular, note that `place_character` must only be called from `Character.__init__`, where it serves to associate the `Character` being initialized with the `GameBoard` that it is on. Remember, each `Character` knows what board it is on and which tile it is on. The `GameBoard` only really needs to know which `Character`s are on it.

Note: For any private attributes that you add, think about any appropriate representation invariants (RIs) that you may want to include. These RIs will NOT be marked, but we encourage you to think about them, as they may help you reason about your code. Any RIs that you add should be consistent with the RIs that already exist in the provided classes.

Now implement those four methods.

Note: You may find the `get_char` method in the `Character` class helpful when implementing `to_grid`.

Once you have completed these methods, you can try running `a1_game.py` and you should see a populated board displayed. You won't be able to move any of the characters yet though.

Task 2:

We'll now develop the code that will allow the `Player` to take a turn in the game.

- Implement the `move` method in the `Player` class according to its docstring.
- You will find that in order for a `Player` to move a `RecyclingBin`, it will be necessary to also implement the `move` method in the `RecyclingBin` class according to its docstring.
- Finally, add code to the `give_turns` method in the `GameBoard` class to allow the player to take their turn.

At this point, you should be able to move around the board, push recycling bins, and lock garbage cans.

Task 3:

Now we will add the code that checks for whether all the raccoons have either been trapped or have crawled into garbage cans.

- Implement the `check_trapped` method in the `Raccoon` class according to its docstring.
- Implement the `check_game_end` method in the `GameBoard` class according to its docstring.

Now, when you play the game, it will end if you trap all the raccoons. (They can't move yet, so it should be easy enough, although it depends on the board's configuration of course.)

Task 4:

- Implement the `move` method in the `Raccoon` class according to its docstring.
- Implement the `take_turn` method in the `Raccoon` class according to its docstring.
- In the `SmartRaccoon` class, override the `take_turn` method according to its docstring.
- Finally, add code to the provided `give_turns` method in the `GameBoard` class according to its docstring so that raccoons get to take their turns too!

You should now have a fully functioning game that you can play! Almost done!

Task 5:

- If you did not do it in Task 3, finish implementing `check_game_end` in the `GameBoard` class so that the score is calculated as described in the docstring when the game ends. In doing so, implement the `adjacent_bins_score` method in the `GameBoard` class according to its docstring.

If you are having trouble getting started with tackling `adjacent_bins_score`, here is a short description of how you might approach the problem:

Go through all the tiles that have recycling bins and, for each one, see how many it is adjacent to. Then we can return the biggest result we find. But how do we find the number of recycling bins adjacent to a particular recycling bin? We can start accumulating a count by adding in all the recycling bins *directly adjacent*. But then we need to remember to count all the recycling bins that are adjacent to *those recycling bins*. We can do that by recording those recylcing bins somewhere (perhaps in something like a stack or a queue) and coming back to check them later. Be careful not to double count though. You can avoid double-counting by keeping track of which recycling bins you have already considered.

Now when you play the game you will see the score displayed at the end.

And that's it!

Task "Fun" (not for credit):

- Feel free to improve `a1_game.py` and let us know what you come up with!

Tips and suggestions

- The method docstrings are written to be as precise as possible, and that sometimes makes them challenging to read. It will help a great deal if you make a small example that the method might have to handle, and then use that to figure out what the method is supposed to do. "Be the method" yourself by enacting it on that data before you try to write code that will do it.

About testing

We have provided several things to help you get started with testing your code:

- the doctests in the starter code,
- unit tests, written using pytest, in `a1_starter_tests.py`, and
- a slightly larger set of tests that you can run via MarkUs (we will announce when the self tests are available on MarkUs).

However, we have further hidden tests that we will use to assess your code. Your assignment grade will be based on the autotesting that we do, and *you can be sure we'll try to break your code as hard as we can*, so you should also! To test your own code thoroughly, add more tests to `a1_starter_tests.py`.

The most efficient way to produce code that works is to create and run your test cases as early as possible, and to re-run them after every change to your code. A very disciplined approach is to design and implement your unit tests for a method before you write the method. (This is called "test-driven development".) You can do this one method at a time, getting each one working before moving on to the next. Don't forget that each method has doctests that you can run – even if you have written the code before writing its unit tests.

Note: We will not directly test any helper methods that you define.

Polish!

Take some time to polish up. This step will improve your mark, but it also feels so good. Here are some things you can do:

- Pay attention to any violations of the Python style guidelines that PyCharm points out. Fix them!
- In each module, run the provided `python_ta.check_all()` code to check for PythonTA errors. Fix them! PythonTA is also included in the tests we provide on MarkUs. The full feedback will be in a file added to your submission called pyta_feedback.txt.
- Check your docstrings for any helper methods you wrote to make sure they are precise and complete, and that they follow the conventions of the Function Design Recipe and the Class Design Recipe.
- Read through and polish your internal comments.
- Remove any code you added just for debugging, such as calls to the `print` function.
- Remove any `pass` statement where you have added the necessary code.
- Remove the word "TODO" wherever/whenever you have completed the task.
- Take pride in your gorgeous code!

Submission instructions

Submit your file `a1.py` on MarkUs. No other files need to be submitted. We strongly recommend that you submit early and often. We will grade the latest version you submit within the permitted submission period.

Be sure to run the tests we've provided within MarkUs one last time before the due date. This will make sure that you didn't accidentally submit the wrong version of your code, or worse yet, the starter code!

Marking scheme

The marking scheme will be approximately as follows:

- pyTA: 10 marks, with 1 mark deducted for each occurrence of each pyTA error.
- following the coding guidelines above: 5
- self-tests, to be provided a bit later in MarkUs: 20
- hidden tests: 65

There will be no marks associated with defining your own test cases with pytest or hypothesis.