Assignment 2: Martian Robot Society

- Due Nov 26, 2024 by 11p.m.
- Points 0

Remember to complete the **domain-comprehension quiz (https://q.utoronto.ca/courses/353223/quizzes/420239)** by **Nov 12 before 11:00pm**.

Due date: Tuesday, Nov 26, 2024 at 11:00 p.m. sharp.

You must complete this assignment individually.

Learning Goals

After completing this assignment, you will be able to:

- model different kinds of real-world hierarchical data using trees
- implement recursive operations on trees (both non-mutating and mutating)

You will also continue to practice writing test suites for the code you are developing.

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). **Read these rules carefully.**

DOs:

- Throughout the starter code, watch for the note: This method must call itself recursively. Such methods will earn credit only if they are recursive.
- All code you write must be consistent with the provided code, docstrings, and the provided UI and client code.
  - While writing your code, 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.
- You may 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. Doctests are not required, unless helpful to you. (Ideally, you should also write a set of pytest cases for each method you write to help you test them.)
- You may 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.
- You may remove unused imports from the Typing module

DON'Ts:

- **You must not use list.sort or sorted in your assignment.** Instead, the helper function called merge has been written for you. You will find it useful when you're merging sorted lists together.
- Do NOT add public attributes or methods to any class. All added attributes and methods must be private.
- Do 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.
- Do NOT add any more import statements to your code, except for imports from the typing module
- Do NOT mutate an object in a method or a function if the docstring doesn't say that it will be mutated.
- Never have a method return an alias to a list; if you need to return an existing list within a method, make sure it is a copy of the original.

Introduction

It is the year 3142. Robots have finally overtaken the world, and their society involves a very strict hierarchy where every robot knows their place in society. (Also, it should be noted that humans had obviously re-located to Mars by this time, so the planet which the robots have overtaken is Mars, not Earth.) As we've discussed in class, trees are a fundamental data structure used to model all sorts of hierarchical data. In this assignment, you will be modelling the organization of the Martian Robot Society using trees.

Every robot in the Martian Robot Society is considered a citizen of Mars. The nodes in our tree will each represent one citizen. Citizens all have subordinate-superior relationships, where one citizen may work under another. Additionally, some citizens are leaders of a specific district within the society. All citizens that work under a leader are considered part of that district.

Note that a district could be a geographical area, or just a domain of responsibility, like Finance.

Setup and starter code

Download the **assignment materials (https://q.utoronto.ca/courses/353223/files/34364524?wrap=1)** **(https://q.utoronto.ca/courses/353223/files/34364524/download?download_frd=1)** and unzip it into your `assignments/a2/` folder.

The code is in three layers:

- `society_hierarchy.py`: Defines classes that keep track of information about the Martian robot society. This is the only file that you will modify. **All classes, methods, and functions that you need to write are in this file.**
- `society_ui.py`: Defines a graphical user interface for interacting with information about the Martian robot society. Run this module to interact with the user interface. You do not have to read or

understand the code in this file. Do not modify this file.

- `client_code.py`: A layer of code that is between the user interface and the "back end" defined in `society_hierarchy.py.` It uses the code you will be writing in `society_hierarchy.py` to make the UI work. You may look through this file to see example usage of the methods and functions you will implement. Do not modify this file.

In addition, we are providing:

- `citizens.csv`: A sample file describing a robot society hierarchy. You can use it to create a society for testing by: (a) uncommenting the `print(create_from_file_demo())` line at the end of `society_hierarchy.py`, or (b) using the "Load society from file" button in the UI and choosing this file.
- `a2_starter_tests.py`: Some basic tests cases that you should add to in order to test your own code.

You should create tests of your own and add them to `a2_starter_tests.py` to thoroughly test your code. There are many methods for you to implement in this assignment: manually testing them all will prove difficult. The `society_ui.py` has only very basic functionality is not designed for thorough testing. (It's also not too fleshed out or well-made and will fail silently on bad input), so you shouldn't rely on it for testing. The UI is just meant as a tool to help you visualize the tree when experimenting with your code.

Problem description

The program for this assignment consists of three main classes:

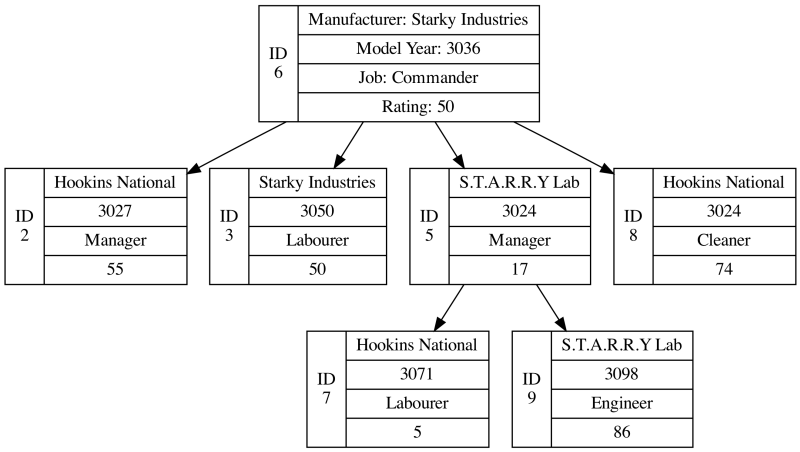
1. `Citizen`: A class representing a citizen in a Martian Robot Society.
2. `Society`: A class representing the entire Martian Robot Society.
3. `DistrictLeader`: A class representing a district leader, a special type of citizen.

Please read through the following sections carefully to understand what is required for this program. Afterwards, we'll provide a breakdown of the tasks.

Citizen class

As mentioned before, each node of the Martian Robot Society tree represents a citizen of this society. Each citizen will have its own set of characteristics: their citizen ID number, manufacturer (the name of which company manufactured this particular robot), model year, job (the role this robot takes on in the society), and their rating (kind of like a credit score; basically a measure of how good of a citizen they are, represented as an integer from 0 to 100).

Each citizen may have one superior and any number of subordinates. For example, consider the following tree of citizens (attributes are labelled by name only for the root Citizen, to avoid clutter):



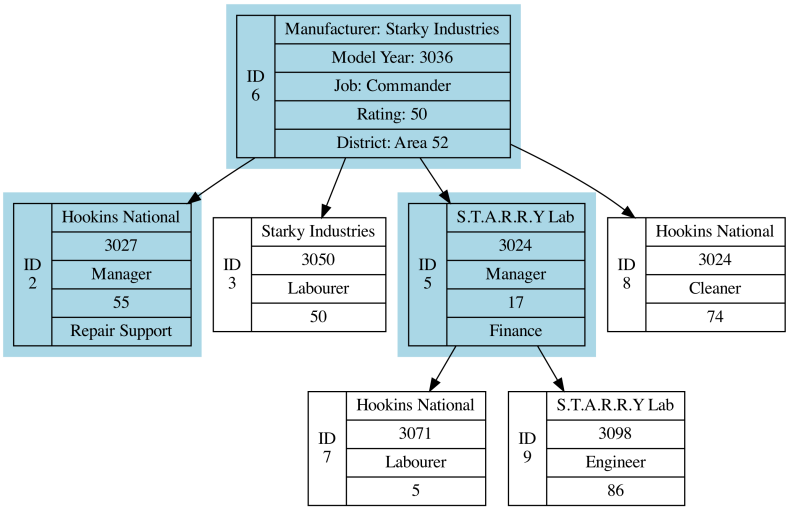
Notice that ID numbers are unique to citizens: the numbers used don't matter, but they will always be positive integers and there cannot be any duplicates within the hierarchy.

While all citizens have superior-subordinate relationships, there are 2 types of subordinate:

1. Direct subordinates: These are subordinates that work directly under another citizen. For example, Citizen ID: 2 is a direct subordinate of Citizen ID: 6.
2. Indirect subordinates: These are subordinates that *do not* work directly under another citizen. For example, being the subordinate of a subordinate. In our example above, Citizen ID: 7 is an indirect subordinate of Citizen ID: 6.

DistrictLeader class

DistrictLeader is a subclass of Citizen. While district leaders are fairly similar to a regular citizen, they also keep track of the district that they lead. All subordinates (both direct and indirect) are said to be part of the district. For example, consider the following hierarchy (attributes are labelled by name only for the root Citizen, to avoid clutter):



Citizen ID: 5 is the DistrictLeader of the district named "Finance". As such, both the Citizens ID: 7 and ID: 9 are also considered to be in (or belong to) the "Finance" district. Additionally, since they're all under Citizen ID: 6 in the hierarchy, who is the leader of "Area 52", they are also part of the "Area 52" district.

It is possible for a citizen to not belong to any district. For example, if Citizen ID: 6 were not a DistrictLeader, then Citizens ID: 3 and ID: 8 would not belong to any district.

If a citizen belongs to multiple districts, we will use the term "immediate district" to refer to the one that is at the lowest level in the tree. For example, in the above tree, Citizen ID: 5, Citizen ID: 7 and Citizen ID: 9 all have "Finance" as their immediate district.

Throughout your program, you may assume that no two DistrictLeaders have the same district; we will not test your code on a society in which two DistrictLeaders have the same district. And our testing will not ask your code to make an update that would create a society in which two DistrictLeaders have the same district.

Society class

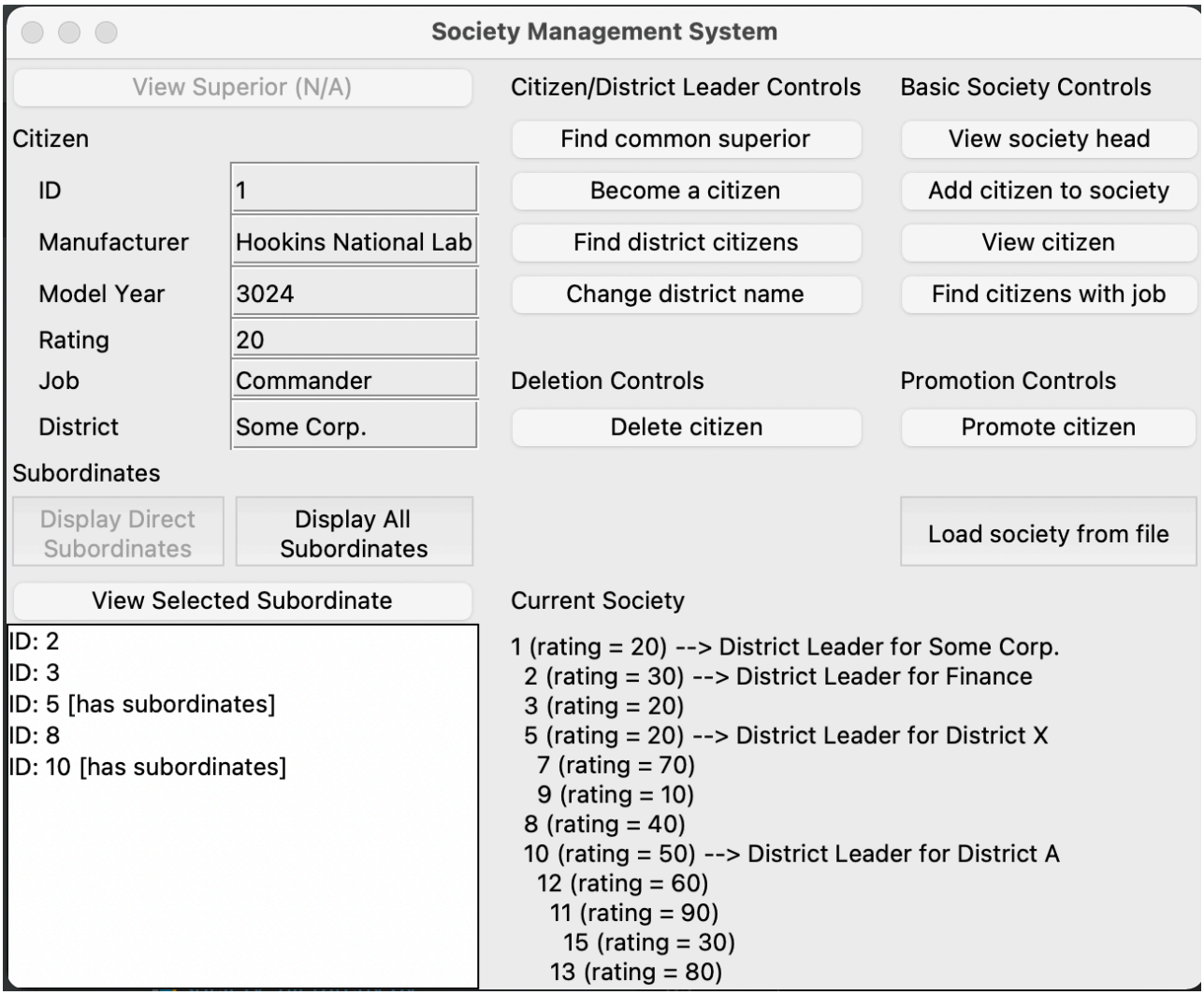
The Society class is responsible for keeping track of the head of the entire society (which is the root of the hierarchy), and providing operations that take the whole society into consideration. Most operations which involve accessing the citizens are done through the society class, such as adding a citizen to the society (when new robots are produced), removing one (when robots are deconstructed), or finding one with a specific citizen ID number.

Task 0: Getting started

1. Download the starter code into your folder for Assignment 2.
2. Mark your `a2` folder as 'sources root' in Pycharm by right-clicking your a2 folder, and selecting "Mark Directory as" -> "Sources Root." This will save you from having to see any unresolved references warnings.
3. Open `society_hierarchy.py` and familiarize yourself with the starter code.
4. Open `client_code.py`: As you work through the assignment, you may look here to see how your methods and functions are used, and what they are expected to return.
5. Run `society_ui.py`: While nothing will work at the moment, this will be one main way of interacting with your code. Of course you should be writing pytests, but you may find the interface helpful in debugging.

society_ui.py

Running `society_ui.py` will provide you with an interface that will eventually look as follows:



Initially, running this file will give you an empty UI which raises errors whenever you do anything.

After completing Task 2, you should be able to run `society_ui.py` to add Citizens to the Society. Many of the buttons in this interface won't be functional until you implement their corresponding methods.

Do **not** rely on `society_ui.py` as your only means of testing your code. The interface working correctly is a good sign, but it does not necessarily catch potential errors in your code.

Task 1: Citizen and Society

Your first task is to implement some basic functionality for the `Citizen` and `Society` classes.

One important note: When writing methods that return multiple citizens, **make sure they're always in ascending order of their IDs**. For example, a citizen with the ID 1 should always appear before a citizen with the ID 2 for such methods.

You must not use `list.sort` or `sorted` in your assignment. Instead, the helper function called `merge` has been written for you. You will find it useful when you're merging sorted lists together.

Citizen IDs are all unique; you do not need to handle duplicates, other than to ensure in later tasks when adding new citizens to the society that you do not end up with duplicates in your tree.

Task 1.1: Citizen class - helper methods

To give the `Citizen` class its basic functionality, complete the following helper methods in the class, according to their docstrings:

- add_subordinate
- remove_subordinate
- become_subordinate_to
- get_citizen

These methods may be useful helpers when you implement other methods in the module. They are also called in our docstring examples, and will be tested.

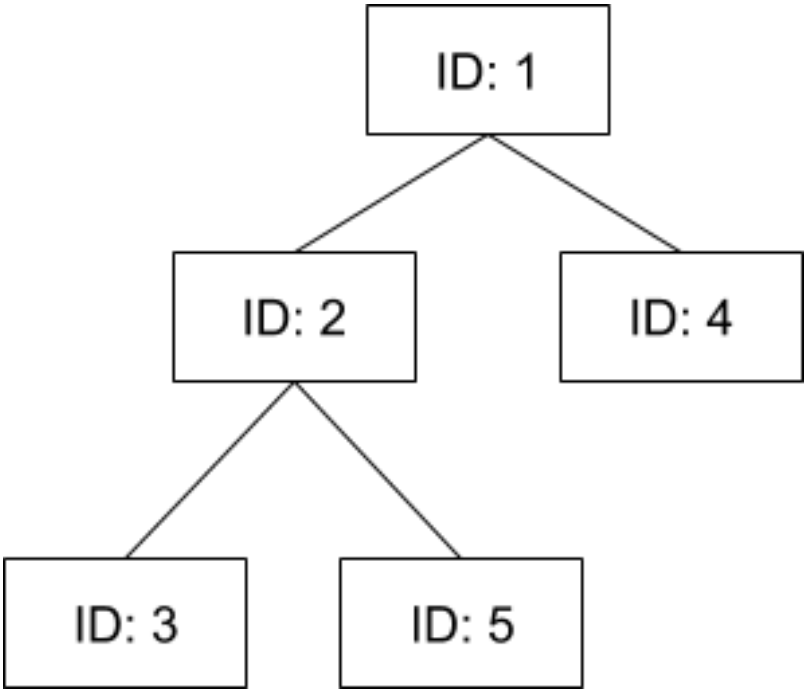
Task 1.2: Citizen class - methods for basic functionality

Next, complete the following methods according to their docstrings. Watch for opportunities to use existing methods as helpers rather than re-do the work.

- get_all_subordinates
- get_society_head
- get_closest_common_superior

Further description of closest common superior

To explain what a "common superior" is, consider the following (abstract) scenario:



The closest common superior of the citizens with IDs 3 and 5 is the one with ID 2. While the citizen with ID 1 is also a common superior of 3 and 5, this is not the closest common superior. The closest common superior of the citizens with IDs 3 and 4 is the one with ID 1, as it's the only superior they have in common.

For simplicity, we will say that the closest common superior of the citizens with ID 2 and 3 is 2. This way, any two citizens in our Society have a closest common superior.

Task 1.3: Society class

Your next task is to implement the basic functionality in the `Society` class by completing the following methods:

- get_citizen
- add_citizen
- get_all_citizens
- get_citizens_with_job

Progress check!

Initialize a `Society` and a `Citizen` (i.e. create a `Society` and some `Citizen` objects, and add those citizens to the society) to test your code and understand how these two classes work together.

You can now try out all the Basic Society Controls and “Find common superior” buttons in the UI. To play around with a new Society in the UI, begin by adding a Citizen with no superior (this makes this Citizen the current Society's head).

Note that the UI does not let you add a citizen with an invalid superior ID (that is, if you add a citizen and choose a superior ID which is not already an existing citizen, then no citizen will be added to the society).

Task 2: District Leaders

Your next task is to implement a subclass of `Citizen` called `DistrictLeader`.

Note that in our tree, a district is not limited to being a geographical location; rather, it's a domain of societal responsibility that falls to this particular leader. You may assume that each district has only one leader (i.e. no two district leaders in a Society will be in charge of the same district).

Task 2.1: DistrictLeader class

Complete the following methods for the `DistrictLeader` class:

- `__init__`
- `get_district_citizens`

For `get_district_citizens`, the method must return a list of all citizens in the district **including** this DistrictLeader. See our **discussion of DistrictLeader class above** for an explanation of which citizens belong to a district.

Task 2.2: Update Citizen class and override methods

With the introduction of the concept of leaders, we will build on the current `Citizen` functionality by adding two new methods to the `Citizen` class and then overriding them as appropriate in the new `DistrictLeader` class.

Complete the following two methods in the `Citizen` class (doctests are not required, unless helpful):

- `get_district_name`

- `rename_district`

The functionality of these methods in `Citizen` depends on them being properly overridden within `DistrictLeader`, so override these inherited methods in the `DistrictLeader` class:

- `get_district_name`
- `rename_district`

Hint: Note that as you recurse, you're going *up the tree*, getting closer and closer to the DistrictLeader, and when you get to the DistrictLeader, it will know the district name.

Task 2.3 Changing between DistrictLeader/Citizen roles

Sometimes we may want to update certain citizens within the society to change their role. To add this functionality, complete this method in the `Society` class:

- `change_citizen_type`

Progress check!

After this step is done, you should be able to use all the Citizen/District Leader controls in the UI, as well as the 'Load Society from File' button to load the provided `citizens.csv` file data.

We have also provided you with some functions at the end of the `society_hiearchy.py` file which create sample societies that you could play around with.

Task 3: Promotions and Deletions

Now we will add some methods that change the structure of the tree and involve the promotion and deletion of robot citizens within a society.

As these methods mutate the structure of your organization, you'll want to test these methods extremely thoroughly so that no Citizens get accidentally removed when they are not supposed to be. We recommend you draw pictures to keep track of what your organization looks like, and any changes that occur.

This task is the most complex, and the most linked to the previous tasks, so thorough testing is critical here.

Note that Task 3.1 and Task 3.2 are largely independent, so if you are stuck on one, feel free to move on to the other before coming back to it.

Task 3.1 Promotions

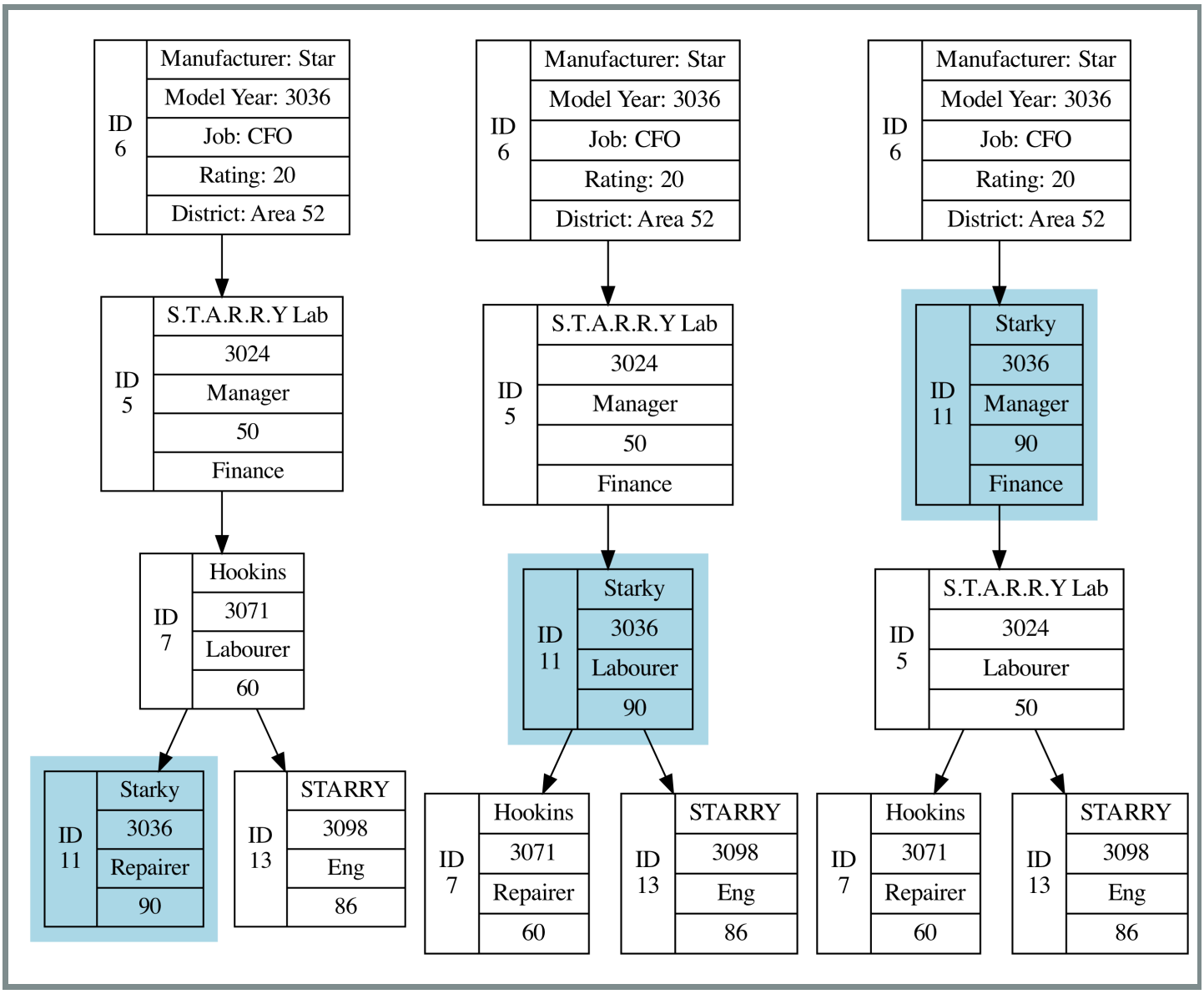
Add the following methods to the `Society` class:

- `_swap_up` (a helper method for `promote_citizen`)
- `promote_citizen`

Only ordinary citizens, not district leaders, can be promoted. When a citizen is promoted, it swaps spots in the hierarchy with its superior *if* it has a rating higher than its superior. They each keep the same data (CID, manufacturer, rating, etc.) except that they swap jobs. And if the superior is a district leader, they swap their citizen type also: the district becomes an ordinary citizen, and the ordinary citizen becomes a district leader). The citizen continues to be promoted until they are below a superior who has a rating greater than or equal to them, or they become a district leader, or they reach the top of the entire hierarchy.

Note that if the Citizen passed to `promote_citizen` is one whose original superior has a rating greater than or equal to them, then nothing happens.

Consider the following scenario (where DistrictLeaders have the district that they're leaders of indicated):



Suppose we wanted to promote Citizen (ID: 11). We would continually swap this Citizen up the hierarchy, performing 2 swaps in total, until it reaches the DistrictLeader position and makes a final swap with the old DistrictLeader since the citizen's rating is higher than the old leader.

As noted above, the promotion occurs within the same district, so once this citizen is promoted to DistrictLeader, we do not continue further (despite the rating of the DistrictLeader's being superior).

Task 3.2: Deletions

First, complete the following helper method in the `Citizen` class, according to its docstring:

- `get_highest_rated_subordinate`

Then, add the following methods to the `Society` class, according to their docstrings:

- `delete_citizen`

Progress check!

After you've completed this task, you should be able to run everything in `society_ui.py` without error!

As we've said, though, running `society_ui.py` is a poor way to test your code, so test these new methods on their own via doctests and pytests. **You'll want to make sure your previously implemented methods still work properly, especially after calling these methods.** There are many opportunities for your Society to break in this task (i.e. improperly updated subordinates, superiors, or heads).

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:

- Review your code and look for ways to simplify the design, or improve the efficiency. Good code means more than just working code.
- Pay attention to any violations of the Python style guidelines that PyCharm points out. Fix them!
- In `society_hierarchy.py`, run the provided PyTA call to check for errors and warnings. Fix them!
- Check your docstrings 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 print statements.
- Remove the "TODO"s wherever you have completed the task.
- Take pride in your gorgeous code!

Submission instructions

1. Login to MarkUs.
2. **DOES YOUR CODE RUN**?!
3. Submit the file `society_hierarchy.py`.
4. On a fresh Teaching Lab machine, make a new folder download all of the files you submitted into it. Test your code thoroughly. Your code will be tested on the Teaching Lab machines, so it must run in that environment. This step will also confirm that you submitted the right version of the required files, and didn't introduce an error at the last moment, for example by adding a comment or changing a variable name.

5. Congratulations, you are finished with Assignment 2, your last assignment for CSC148! Take a well-deserved break and go back to binging Season Whatever of That Show You Watch!