CS 115 Lab 9: Taking Care of Business

A core element of many modern businesses is the org chart: A chart specifying who in the company is responsible for whom.

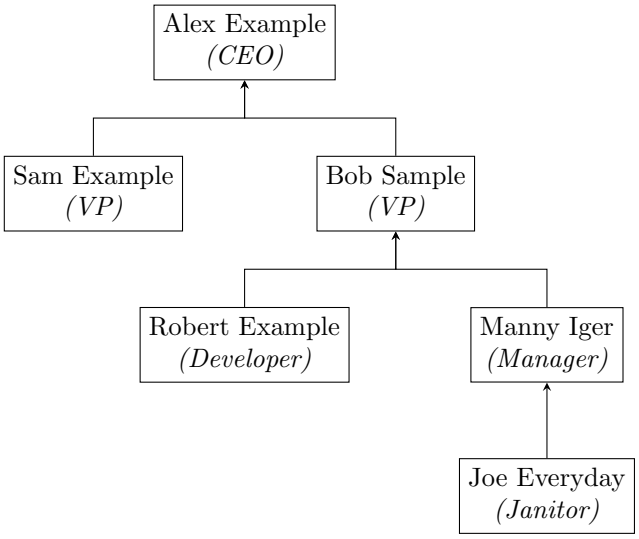

Figure 1. An org chart, containing six employees. These six employees are arranged in several levels. At the top is the CEO "Alex Example". Below them are two VPs, "Sam Example" and "Bob Sample", who are managed by Alex. Below them are a Developer "Robert Example" and a Manager "Manny Iger", who are managed by Bob Sample. Finally, there is a Janitor "Joe Everyday", who is managed by Manny. These relationships are represented by arrows: Each employee has an arrow pointing to their manager, who has an arrow pointing to their manager, and so on, all the way up.

In the example above, the CEO sits at the top of the company, and manages the two VPs. One of those VPs then manages the developer and the manager, with the manager also managing the janitor. In other words, every member of the chart directly manages the people directly below them on the chart.

As these are super common, it makes sense to be able to represent these relations inside of a Python program. Thus, your task in this lab will be to implement exactly that.

Task 1: Storing an Org Chart Node

To begin with, we just want to be able to represent an employee. We will do that using the `Employee` class, which is what you will be implementing today.

To being with, we want to implement the constructor. This should take in an employee's name and ID, and initialize the following fields:

- The `name` field should store the employee's name, as given.
- The `manager` field should store the employee's manager. This will itself be another `Employee` object, but should start out initialized to `None`, which we will use as the default value for someone who does not have a manager.
- The `id` field should be the employee's ID. This should be an integer.

Task 2: Assigning Managers

Now that we have our employees, we want to be able to assign them to work for each other.

Your next task is to implement the `assign` method. When called on an employee with a second employee argument, that second employee becomes a direct report of the first. In other words, the first employee (the one you are calling the method on) becomes the manager of the second (the one you are passing into the method).

A few things to be careful of for this one:

- Make sure you get the order of which employee is which correct. It can be easy to get them backwards, which will result in your org charts not being built up correctly.
- Note that we want to store a reference here. When we update an employee's `manager` field, that should store a reference to the manager (like we saw in box-and-arrow diagrams). You do not want to create a copy of the manager while doing this.

CA Check-In

After you've implemented the above, demonstrate it to the CA by using your class to represent the org chart above.

1. Show the CA your class.
2. Ask the CA to draw you an org chart.
3. *In the IDLE shell*, construct instances of `Employee` and assign them accordingly to match the org chart the CA drew.
4. Once the code runs and the CA verifies it reflects their org chart, share your current status and plan for Homework 5. We will be checking in on your progress with the homework between labs from now on.

Task 3: Introducing Magic Methods

To help with debugging and testing in the future, we now want to implement some magic methods.

Magic methods are just like normal methods, but replace standard operations. For example, we can define equality (`__eq__`) by implementing a method named `__eq__`. Then, when we compare two employees with the `==` operator, our `__eq__` method is the one that is invoked, letting us use the usual operators to accomplish our goals. For example,

```
employee1 = Employee("Mohit P", 45)
employee2 = Employee("Andrea", 10)
employee3 = Employee("Mohit P", 45)
>>> employee1 ==  employee2
False
>>> employee1 ==  employee3
True
```

In this lab, we will implement the following magic methods:

- `__eq__`: We want to be able to compare employees. We will say that two employees are the same if they have the same ID and the same name. So in case we accidentally enter the same employee twice, we should be able to detect to detect that by comparing their ID and name.
- `__str__`: We want to nicely format our employees for printing. In particular, we want to list their ID and name. As such, given an employee with ID 5 and name "Bob", we except an output of `"5: Bob"`. In other words, it is the ID, followed by a colon, followed by the name of the employee.

  If the employee is then also managed by someone, we want that information as well. If Bob above was managed by "Manny" (employee ID 4), we want to get the string `"5: Bob (Managed by 4: Manny)"`. Note that if the employee is not managed, that latter part should not appear.
- `__repr__` (optional): We also encourage you to create a nice `__repr__` method for easy debugginng. This should include all the relevant information. How this is implemented we leave up to you.

Task 4: Promotions

We now want to be able to move people up the org chart. When we `promote` an employee, we raise them one level up the org chart: They are now managed by the manager of their previous manager.

For example, in the org chart below, we promote "Joe Everyday" one level upwards.

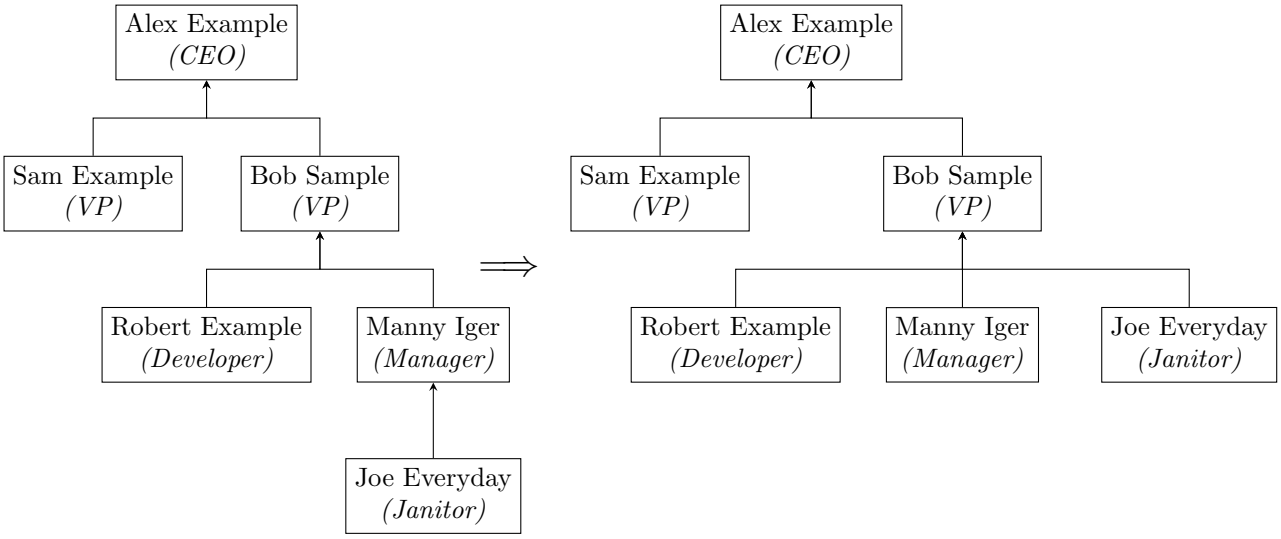

Figure 2. Two org charts. The first is the same org chart as before, with a CEO (Alex Example) managing two VPs (Sam Example and Bob Sample), one VP (Bob Sample) managing a Developer (Robert Example) and a Manager (Manny Iger), and with the Manager managing a Janitor named "Joe Everyday".
The second chart features the same employees, but the Janitor has been moved one level up, and now has an arrow to Bob Sample, rather than Manny Iger. Note that this the same VP who manages the Janitor's previous manager.

Your next task is to implement the `promote` method. When called on an employee, this should promote them one level up the org chart if possible. Note that you can be promoted to the top level, being managed by no one, but if you are already at the top, then you cannot be promoted.

The format for this method should be such that I can call `joe.promote()`.

Task 5: The Org Depth

Next, we want to calculate how deep an employee is in the org chart. In other words, how far down the tree are they located?

The rule for calculating the org depth is as follows:

- If you have no manager, you are at the top and your org depth is 1.
- Otherwise, your org depth is 1 higher than your manager's.

For example, in the org chart at the top of this lab, Alex has an org depth of 1, Sam has an org depth of 2, and Joe has an org depth of 4.

Your task is to implement the `org_depth` method, such that calling `bob.org_depth()` on some employee `bob` will return their depth in the org chart.

Rubric

Category	Description	Points
Name	For stating your name on every file for submission.	1
Pledge	For writing the full Stevens Honor Pledge on every file for submission.	3
Collaborators	For filling out the Collaboration Statement.	1
Docstrings	For having detailed docstrings on your code.	10
Check-In	For performing the CA Check-In during the lab.	10
Task 1: `__init__`	For correctly implementing the constructor.	10
Task 2: `assign`	For correctly implementing the `assign` method.	15
Task 3: `__eq__`	For correctly implementing the `__eq__` method.	10
Task 3: `__str__`	For correctly implementing the `__str__` method.	10
Task 4: `promote`	For correctly implementing the `promote` method.	15
Task 5: `org_depth`	For correctly implementing the `org_depth` method.	15
Total		100

Submission Instructions

For this lab, you must submit the following files, with exactly the following names:

- `lab9.py`
- `lab9_report.txt`

Remember to put your name and the Stevens Honor Pledge on every file you submit! Failure to put the pledge can result in marks being deducted, with repeat penalties potentially getting stricter between assignments.

Submissions must be handled through Gradescope (accessible through the homework page on Canvas). Grades will be released through Gradescope once the homework is graded.

We expect students to submit their labs to Gradescope and verify that their code passes the public tests before they leave the lab so CAs can help with any submission issues.

Tips, Suggestions, and Reminders

- **Remember to include docstrings!** If your submission does not have docstrings, or has docstrings that are not related to your code, you will not get credit for them.
- We have provided a test file for you to test your solutions. By running this test file, you should be able to verify that your files are named properly, and that your code works correctly.
- Code that doesn't compile (throws a `SyntaxError` on being run) will receive a 0 for the assignment! Make sure you submit code that can be run, even if it does not pass all the test cases.
- **Don't forget to add your name and pledge!**