Lab 14

Classes, Decorators, Pickling

The goal of this lab is to practice the ideas of classes, decorators and pickling. You will also get a preview of (or another look at) the idea of a *relational database*. In this context, that will mean that you have a class for each kind of object in your database, and instances of those classes to represent different individual examples of those objects. That's the *database* part. The *relational* part is that you make connections between these instances as needed to represent the connections that the data have in the real world.

Overview

You will create a basic gradebook application. There will be three classes: Student, Assignment, Section. Then, you'll create a way for a teacher to enter grades for students in a section of their class. You'll make one file called `gradebook.py` that will contain the classes and global functions, and then a second file called `grading.py` that will have the `main()` in it. You'll probably want to do a bunch of testing on `gradebook.py` as you write it. Be aware that you should either write your code that calls `gradebook.py` from `grading.py` or you will have to rebuild all your test data later.

The `Student` class

The `Student` class will need the following attributes:

1. `studentID`: Read-only integer (using the `@property` decorator) that is set automatically by the constructor. The IDs should start at 1, and each student added will get an ID that is 1 more than the last integer used.
2. `firstName`: String
3. `lastName`: String
4. `nextID`: Integer, class attribute, equal to the ID of the next student added.

The Student class will need the following methods:

1. `__init__`: Inputs: `firstName`, `lastName`. Returns: Student instance. Note: This will need to assign the value of the read-only studentID property using the nextID class attribute, and then modify the nextID value for the Student class. Also add this instance to the global `students` collection.
2. `__str__`: Return a human-friendly listing of the three instance attributes.
3. `gradeReport`: Just `self`. Notice that this includes the attributes `firstName`, `lastName`, `studentID`. Returns: Float, the student's average. In addition to returning the average, it prints out all of the student's assignments, with the name of the assignment, the student's grade, and how many points the assignment is out of (max

score). Also show the final average (= (total of score values)/(total of outOf values)). *You may assume that each student is only in one section, so the average uses all the grades you find for that student.*

The `Section` class

The `Section` class will need the following attributes:

1. `sectionID`: Read-only integer, same idea as `studentID`.
2. `courseName`: String, name of the course, e.g. `"Honors CS, Per. 7"`
3. `studentList`: List of Integers, each integer is a `studentID`.
4. `nextID`: Same idea as `Student.nextID`.

The `Section` class will need the following methods:

1. `__init__`: Input: String, the course name. Initialize the `studentList` to be an empty list. Assign the `sectionID` using the same idea as `Student.studentID`. Also add this instance to the `sections` collection.
2. `__str__`: Return a human-friendly string with `sectionID` and `courseName`.
3. `classList`: Return a human-friendly description of the section, with a formatted list of students (use the `Student.__str__` method!).
4. `addStudentByID`: Inputs: None, Returns: None. This should print a list of all students, including the student IDs. Then take an integer as input from the user, and add that `studentID` to the `Section.studentList`.
5. `addStudentByName`: Inputs: 2 Strings, `firstName` and `lastName`. Output: No returns. Check if a student with that `firstName` and `lastName` exists, add that record if so, let the user know if not found.

The `Assignment` class

The `Assignment` class represents one assignment for one student in one section. To be clear, if you have, say a Lab 13 for Software Development, Per. 1, then that is ***not*** an Assignment. If you have Luca Guizar's grade on Lab 13 for Software Development, Per. 1, ***that*** is an Assignment. If I were doing this for real, I would probably separate these into two classes, but I am trying to simplify things a bit here.

The `Assignment` class will need the following attributes:

1. `assignmentID`: Read-only integer, same idea as `studentID`.
2. `studentID`: Integer, ID of student whose grade this is.
3. `sectionID`: Integer, ID of the section that the student is in.
4. `title`: String, the name of the assignment (e.g. `"Lab 13"`)
5. `grade`: Integer, this student's score on this assignment.
6. `outOf`: Integer, the max/total points available for this assignment.
7. `nextID`: Same idea as `Student.nextID`.

The `Assignment` class will need the following methods:

1. `__init__`: Inputs: studentID, sectionID, title, grade, outOf. Assign the `assignmentID` using the same idea as `Student.studentID`. Also add this instance to the global `assignments` collection.
2. `__str__`: Return a human-friendly description that shows the `assignmentID`, `title`, `grade`, `outOf`, `section.courseName` and then uses `Student.__str__()` to show the student information.
3. `enterGrade`: Class Method. Inputs: String, the `title` of the assignment; Integer, the `outOf` value. Return: an `Assignment` instance. The method will need to first show the list of available sections, allow the user to choose a section (by ID), then show the list of students in that section, then allow the user to choose a student (by ID), and then allow the user to enter a `score`. Make sure you use the `section.classList` so that you only show the students from this section.

Global (Module-level) variables

Quick note: Normally, we avoid global variables. This is correct, you should. The variables I am creating here are global within `gradebook.py`, which we are going to `import` later. Which means that, when we use them, they will be attributes of the `gradebook` module instead of global variables. So, they are far less dangerous than actual global variables. With that said:

First, create three global variables: `students`, `sections`, `assignments`. You can make these lists or dictionaries, I'll leave that up to you. Each of these should contain all of the instances of that class. They all start empty.

Second, create a fourth global variable: `gradebook`. This has to be a dictionary. It needs six keys, with values that will be equal to the thing the key is named after, but it should just be initialized as an empty dictionary:

- `"studentNextID"`
- `"sectionNextID"`
- `"assignmentNextID"`
- `"students"`
- `"sections"`
- `"assignments"`

We will only use this variable when bringing data in and out of the file, otherwise we will use the global variables or the class attributes by the same name.

Global (Module-level) functions

1. `enterGrades`: Inputs: String, the `title` of the assignment; Integer, the `outOf` value. Return: None. Call `Assignment.enterGrade()` repeatedly with the same `title` and

`outOf` values. Append each entered assignment to `assignments`. Stop when `sectionID` entered is 0.

2. `showGrades`: Input: String, an assignment `title` that's been used before. Return: None. Prints a list of all the grades whose `title` matches the input, grouped so all the students in the same section are together, and sorted by last name within each section.
3. `adjustGrade`: Input: String, an assignment `title` that's been used before. Return: None. Using `showGrades()` to display the options, allow the user to input a `studentID` and `sectionID` and then allow the user to enter a new grade for that student on that assignment. Modify the existing `Assignment` instance to reflect that change. Extra fancy (optional): If the user enters a `studentID` that only appears once in the listing from `showGrades()` (which is most likely) then figure out the `sectionID` without asking the user for it.
4. `loadGradebook`: Input: String, a filename where a pickled gradebook could be stored. Return: The lists or dictionaries you are using for the `students`, `sections`, `assignments` variables.
    a. If the file does not exist: Create the three keys for the nextID values, with each having value = 1, and also create the three other keys, and set them equal to an empty dictionary or list, depending on what you decide to use for the collections of students, sections and assignments.
    b. If the file exists: Set up the `students`, `sections`, `assignments` variables from the values that you read, and set the three `nextID` class attributes from the values that you read.
5. `saveGradebook`: Input: String, a filename where the pickled gradebook will be written. Return: None. First, update all six keys to catch any changes that were made during the session, then pickle and write the contents of the global `gradebook` variable to the file.

Overall Setup

As it says above, make one file called `gradebook.py` that contains the class definitions with their attributes and methods, plus the definitions of the global variables and functions.

At the bottom of `gradebook.py`, not inside any function (not inside a `main()`) run:
`students, sections, assignments = loadGradebook('gradebook.dat')`

This will automatically set up the gradebook from its last saved state once this file is `import`ed.

In all those methods and functions, make sure that you do proper validation. The user should not be able to crash it unless they hit CTRL-C.

Make a second python file called `grades.py` that creates an interface for the gradebook. At the top, you should use `import gradebook` or `from gradebook import *`. Personally, I found the `import gradebook` construction to work better (and, in general, I always recommend that!).

However, be forewarned that any data you pickled and saved when you were testing `gradebook.py` will not be usable in `grades.py`. See below for more on this. There is a work-around, but it may not be worth it. See below for details. You will need to start over with a new `gradebook.dat` file and re-create your data using `grades.py`.

You can write the following functionality however you want. It should allow a teacher to:

- Add a student (including putting them into a Section)
- Add a section
- Enter some grades for an assignment
- Get a report of all of one student's grades, including that student's average
- Get a report of all of the grades from one section on one assignment
- Modify a grade that's already been stored
- Save and Quit
- Quit without saving (include a confirmation!)

If you are tempted to use the `from gradebook import *` construction, here's a warning:

Let's assume that you have some data in your `gradebook.dat` file, and that you have correctly written the `loadGradebook()` described above. Then, if you do this:

```
from gradebook import *

assignments = []
students[0].gradeReport()
```

Then you'd think that the grade report called for here should be empty, because you've just emptied the list of all assignments, so there are no assignments. But, the gradeReport is not, in fact, empty. Hmm.

Why? Because the variable assignments referred to in the `gradeReport()` method is the one from the `gradebook` module's scope, and the one we just assigned there is in the `__main__`. scope. But, they have the same name, so you can't assign the one from the `gradebook` scope. If you never assign a new local `assignments` variable, you can modify and read the one from `gradebook` with no problems. But, if you use `import gradebook` instead, you keep the local variables in scope `__main__` and the `gradebook` variables in scope `gradebook.variableName` and you can always access, assign and modify all the variables

in both scopes. This is one of the reasons I always recommend you use `import moduleName` instead of `from moduleName import *`.

Before submitting, use your interface to create:

- At least 10 students
- Spread out across at least 2 sections (No student is in more than one section)
- With at least 3 grades for each student

Submit all three files:

1. `gradebook.py`
2. `gradebook.dat`
3. `grades.py`

Note on Pickled Data

This is a little subtle, if you want to not pay much attention to this, it's ok. The important point is that you can't make a bunch of fake data when running `gradebook.py` directly, and then use that same fake data (by un-pickling it) when running `grades.py` and using `gradebook.py` as an import. You'll have to re-create the data or translate it. Translating might not be worth it. [See below for translation work-around.]

When you pickle data, the pickle function stores its data type in addition to the contents. So, when you store a `Student`, it stores the attribute values (firstName, lastName, studentID), and also stores that it is an instance of the `Student` class. This is part of what makes pickling awesome – it remembers exactly what everything is, including its type.

The problem is, it is even more specific than that. When you are running `gradebook.py` directly, it is running as `__main__`. (Remember how we did `if __name__==__main__:` ? The same thing – if we are running a file directly, Python labels it as `__main__`.) So, not only is that `Student` an instance of the `Student` class, it is even more specific, it is really an instance of the `__main__.Student` class. When you `import gradebook` into `grades.py`, the `Student` class isn't in `__main__` because now `__main__` is `grades.py`, and the definition of the `Student` class is not in `grades.py`, it's in `gradebook.py`. So, the `Student` class in `grades.py` is, more specifically, `gradebook.Student` (And this is true whether you've used `import gradebook` or `from gradebook import *`). So, when you try to un-pickle the data that you created earlier in `gradebook.py`, you are going to have an error because you've stored instances of the `__main__.Student` class, but that doesn't exist, you only have `gradebook.Student` available. So, `pickle` will crash out when trying to decode that data because it can't find the appropriate class to match the data to.

Note: To figure this out, I used the functions `locals()` and `globals()`, which give a listing of everything in the current namespace. You are encouraged to try this out to make sure you understand all of what I'm talking about above.

Also Note: While I was looking into this, I noted that there is a module/library called `dill` that extends the `pickle` library. I'm amused, even though I don't remember what it does.

Work-Around to translate data

So, I spent some time researching this and came up completely empty, so I'm guessing there is no pre-built way to do this. So, I hacked something, but it really isn't pretty. I'll walk you through the thought process below.

So, the problem is this:

- When the instances were saved, they were instances of, e.g., `__main__.Student` but we need them to be instances of `gradebook.Student`.
- So the solution is to create *both* classes, load, transfer from one to the other, and then save.

STEP 0: Always, always, first thing, make a copy of your data so that when (not if) you mess things up, you don't lose your data.

STEP 1: To transfer easily/automatically comes back to the `__repr__()` method. The `__repr__()` method is something that should be *machine-readable*, it should, effectively, produce the text to a call to the constructor that produces this instance. So, make sure you have good `__repr__()` methods for each class that meets that criterion. To test them, you can use the code, e.g.: `eval(student.__repr__())`. What the `eval()` function does is to treat the string argument it receives as executable code, and execute it. So, `eval(student.__repr__())` should return a functional call to the Student constructor, meaning it should return an actual Student instance. Check that this works for all three classes before moving on.
Use the `eval()` function with care, you can get into a lot of trouble with it. Whenever I use it, I do one run of the code where I just print out the executable code instead of executing it to check that I really want to run that the way it comes out. Then, I switch from `print()` to `eval()`.

STEP 2: Copy and paste the three classes from `gradebook.py` into `grades.py`. Then, make sure you are using `import gradebook`, and ***not*** `from gradebook import *` at the top of `grades.py`. Now, when you run `grades.py`, you have, e.g., ***both*** `__main__.Student` and `gradebook.Student`.

STEP 3: Insert the `loadGradebook()` function in `grades.py`, comment out the `loadGradebook()` call in `gradebook.py`. This will load the objects as instances of, e.g.

`__main__.Student`. Then, use for loops to go through the lists and re-assign them as instances of `gradebook.Student`, using e.g.:
`student = eval(f"gradebook.{student.__repr__()})")`.
Print out the lists and make sure they look right. Then run again and `saveGradebook()`.

STEP 4: After all this, you may find that the ID numbers don't line up. The ID that a student had before might be different than it gets assigned to when they get re-constructed. You might be able to re-construct them carefully enough that you don't have this, but if you deleted any records before, you'll have gaps in your ID sequence, etc. so it might difficult. So you might have to re-link the data by looking up the studentID's from the old data, and replacing these with the studentsID's from the new data, and same for the SectionIDs. If you do this be careful: for ex, if you have old Student IDs: 1, 3, 4, 7, and then you have new ones: 1, 2, 3, 4, if you change the old 4's to new 3's, then change the old 3's to new 2's, you'll get *both* the old 4's and the old 3's because you'll have two different collections all having studentID = 3. So, just pay attention to the possibility of 'collisions' like that. Generally if you assign the largest IDs first, you will probably have fewer problems like this. Once upon a time, my wife actually had a full-time job at a library relinking data like this by hand after a database migration for the library's catalog of books went awry.

STEP 5: Clean up: un-comment the `loadGradebook()` call in `gradebook.py`. Remove the class definitions from `grades.py`. Remove the `loadGradebook()` call in `grades.py`. Remove the `for` loops that converted the instances, and any `print()` or `pprint()` statements you used.

Now you should have all your data moved from `gradebook.py` to `grades.py`. Was it worth it? Only if you had a lot of data that's not easy to re-construct. Probably easier to print the data to the console and re-enter it, or, if you still have the code that created it in the first place, re-run that.