Assignment 1 - Student Management Console

Clarifications

- (Sept 30) From the automarking TA: please remember to **not import any modules** other than your own. You should have no reason to do so. However, do **import a Stack**, or include it directly in your code; the module will not be provided!
- (Sept 29) All commands, student names, and course names are case-sensitive! This means that `Create student blah` is invalid, and `create student dave` and `create student DaVe` create two different students.
- (Sept 29) You may submit additional files for your back-end. However, make sure your central file for the front-end is still sms.py!

Introduction

As we've talked about in lecture, one of the best uses of object-oriented programming is to model complex entities and relationships in the real world. In this assignment, you'll implement a simple, interactive program that someone in the Undergraduate Office could use to keep track of student enrolment in the coming term.

There are two major components to this assignment:

1. The *front end*, a command-line (i.e., text-based) interface in which the user may enter commands from the keyboard or files to manage student data, and
2. The *back end*, the class and methods that you will write to execute the command-line inputs.

This assignment covers material from the first three weeks of the course: object-oriented design, exception handling, and stacks; this is also an excellent opportunity to brush up on your Python skills!

How to succeed on this assignment (a.k.a. general instructions)

1. The assignment is **due SATURDAY, OCTOBER 11 at noon**, but start early! After the first week (lectures + lab), you will know enough to be able to complete at least half of the assignment.
2. Aim to spend about 2 hours per week working on this, but *don't* neglect your regular class work.
3. Read the instructions carefully. If you are less detail-oriented than a computer, you will make mistakes.
4. You may **work in groups of up to 3**. Communicate with your partners!
5. You may **not** import any external modules unless explicitly told to.

Starter code:

- sms.py
- student.py

- sms_test.py
- student_test.py

Assignment Specifications: Front End

Open sms.py - this is your starter code for the interactive part of your application. The core of the interactive command line is the `run` function, which features an infinite loop that accepts commands from the user and executes them.

```
def run():
    while True:
        command = input('')
        if command == 'exit':
            break
        else:
            print(command)
```

Right now, there is very little in the code - it simply repeats whatever string you type in, exiting when you type `exit`.

For your assignment, you must support the following commands. Read the following descriptions carefully: this is the **only** part of your assignment we'll test, but our testing will be thorough!

System Commands

Important: use Python's `split` method to turn a string into a list of words. This will also ignore all whitespace in a line, so you'll be able to use commands like `enrol   david CSC148`. You may assume that student and course names are single words (i.e., don't contain any whitespace), and are *case-sensitive* (so `'David'` is a different student than `'david'`).

- `create student <name>`: create a new student with name `<name>`.
    - If a student with the same name has already been created, a message `ERROR: Student <name> already exists.` is printed, and no new student is created.
- `enrol <name> <course>`: enrol student `<name>` into `<course>`. You may assume that courses are represented by a *course code*, e.g. `CSC148H1`.
    - If no student named `<name>` exists, a message `ERROR: Student <name> does not exist.` is printed.
    - If the student is already taking the course, do nothing.
    - (harder) If there are already 30 students enrolled in `<course>`, a message `ERROR: Course <course> is full.`, and the student is *not* enrolled in the course.
- `drop <name> <course>`: remove student `<name>` from `<course>`.
    - If no student named `<name>` exists, a message `ERROR: Student <name> does not exist.` is printed.
    - If the student isn't taking the course or the course doesn't exist, do nothing.
- `list-courses <name>`: print a list of codes of all the course taken by `<name>`, in the format `<name> is taking <course1>, <course2>, ..., <course-n>`, where the courses

are listed in **alphabetical order**.

    - If no student named `<name>` exists, a message `ERROR: Student <name> does not exist.` is printed.
    - If `<name>` isn't enrolled in any courses, print `<name> is not taking any courses.`

- `common-courses <name1> <name2>`: print a list of all courses taken by *both* `<name1>` and `<name2>`, in the format `<course-1>, <course-2>, ..., <course-n>`.

    - The courses should be listed in **alphabetical order**.
    - If one of `<name1>` or `<name2>` doesn't exist, a message `ERROR: Student <name1/2> does not exist.` If they both don't exist, two error messages, each on separate lines, should be printed (`<name1>` first).
    - If the students have no courses in common, you should print the empty string.

- `class-list <course>`: print a list of all students enrolled in `<course>`, in the format `<name1>, <name2>, ..., <name-n>`, where the names are in alphabetical order.

    - If no one is taking `<course>`, print the message `No one is taking <course>.`

In addition to these, there are three system meta-commands that your program needs to handle:

- `exit`: end the loop. This is implemented for you, and you shouldn't change it! :)
- `undo`: reverse the last command.

    - Only has an effect for the commands that change the data: `create`, `enrol`, and `drop`.
    - Has *no* effect if the last command was anything else. This *includes* unrecognized commands.
    - You cannot "undo an undo". If you run `undo` twice in a row, the previous *two* commands are reversed.
    - If there are no remaining commands to undo, a message `ERROR: No commands to undo.` is printed.
    - The command `create student <name>` has a possibility of *failure* (if the student `<name>` already exists). However, if the user tries to undo a failed `create student` command, this *should not* delete the student! Instead, just do nothing. :)
    - Similarly, do nothing if the user tries to undo a failed `enrol/drop` command.
    - Hint: Use a stack!

- `undo <n>`: reverses the last `<n>` commands.

    - If `<n>` isn't a positive natural number, print `ERROR: <n> is not a positive natural number.`
    - Note: this is completely equivalent to entering `undo` into the console `<n>` times, **except** that if there are no more commands to undo, the warning message `ERROR: No commands to undo.` is only printed **once**.

Finally, if *any other command/text* is given, an error message `Unrecognized command!` is printed, and the input is ignored.

Getting started

Whew. Now that you have a good idea about the commands your program will need to support, make sure you really understand all of the details by watching this video and looking at the provided sample tests. Of course, our tests will be much more

comprehensive, but you are more than welcome to add your own. (However, you *won't* be assessed on these tests.)

Next, we highly recommend that you leave `sms.py` alone, and work on completing the *back-end*.

Assignment Specifications: Back End

Unlike the labs and exercises, this assignment is specifically geared towards giving you a more open-ended task for you to **build your own software**.

Therefore, how you implement the back end - the class(es) you'll use to store your data - is completely up to you. If you open up student.py, you might be in for a surprise. It's empty!

Design and Implementation

Your biggest task is to **design and implement all of the classes you need to achieve the functionality described in the previous section**. It is up to you to decide what your classes, attributes, and methods should be. Remember that we won't be testing these directly, but instead only testing the front end.

However, we will be grading your code for good **object-oriented design**, **documentation**, and **coding style**. See the marking scheme for details.

Testing

You must create a set of unit tests in the file `student_test.py` for all of your class methods in the back end. Your tests should be reasonably complete, and cover all corner cases in your code.

Note that you do **not** need to submit any tests for the front-end `run` function (although as we stated above, some are provided and you're encouraged to make your own).

Integration

To promote good code organization, you are required to separate the code powering the interactive console in `sms.py` with the code for your class(es) modelling the problem entities in `student.py`.

Of course, you'll need to use your class(es) in `sms.py` as well. Follow these guidelines when you do:

1. Nothing is going to work unless you import your classes with a `from student import <something>, <something_else>, ...` statement at the beginning of `sms.py`.
2. **Don't** use any of your classes' attributes in `sms.py` directly. Instead, define class methods (with meaningful names) to update them.
3. Don't feel like you have to write all of your code in one file or the other. Part of this assignment is finding the right balance in how you organize your code, and there isn't one right answer.

Submission instructions

1. Login to MarkUs, and create a group if necessary.

2. Does your code run?
3. Submit 3 files (all others will be ignored):
    - sms.py
    - student.py
    - student_test.py
4. Go through this checklist. Make changes if necessary.
5. Does your code run?!
6. Read the marking scheme. Make changes.
7. Does your code run?!
8. Check your tests. Run your tests. Run our tests. Write more tests. Run more tests.
9. **Does your code run?!**
10. Submit the 3 files, again. Then either go to step 4 or 11.
11. Congratulations, you're done! Go have some chocolate. :)

David Liu (liudavid at cs dot toronto dot edu)
Come find me in BA4260
Site design by Deyu Wang (dyw999 at gmail dot com)