Assignment 2: Course Prerequisites

Clarifications:

1. (Nov 3) A course that has already been taken should be "takeable", i.e., calling the method `is_takeable` on it should return `True`.

Introduction

Choosing courses can be one of the most confusing and time-consuming non-academic experiences at university. Among the many factors that go into planning your courses are the *prerequisite constraints*, i.e., which courses must be taken before you can take a particular course.

The Faculty of Arts and Science Course Calendar lists all courses and their prerequisites, but this information is poorly represented in a list because course prerequisites are often not linear. In fact, course prerequisite structure resembles that of a tree, and is pretty nicely visualised in tools like this one.

In this assignment, you'll use trees to store and manipulate course prerequisite data, using this information to write a program to help students plan their courses for their entire university career.

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

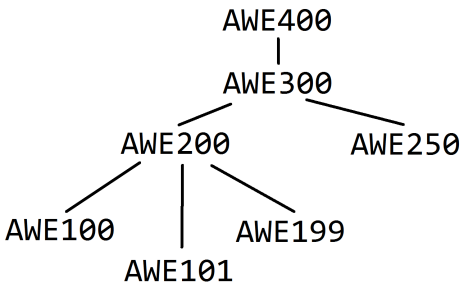
1. The assignment is **due SUNDAY, NOVEMBER 23 at 9am**, but start early!
2. Aim to spend about 2 hours per week working on this, but *don't* neglect your regular class work.
3. Read the instructions carefully. If you are less detail-oriented than a computer, you will make mistakes.
4. You may **work in groups of up to 3**. Communicate with your partners!
5. You may **not** import any external modules unless explicitly told to.

Starter code (updated with tests) :

- course.py
- planner.py
- course_test.py (sample tests added)
- planner_test.py (sample tests only; do not submit)
- single.txt (sample test only; do not submit)
- binary_simple.txt (sample test only; do not submit)

Assignment Specifications: Data Model

As hinted at earlier, we'll model course prerequisites using our familiar tree data structure. But be aware that the tradeoff for using trees is a large simplification of the original problem: in general, courses can share prerequisites (look at the link above), but this can't be captured using trees alone.



The above diagram illustrates what we mean by a tree-like data structure: course AWE400 has a single prerequisite, AWE300, which has two prerequisites, AWE200 and AWE250. AWE250 is a leaf, which means it has no prerequisites (it can be taken at any time).

We have done some of the class design for you, in course.py. Read the docstring of this class carefully; in particular, make sure you understand the purpose of each of its three attributes. When you are ready, implement the **five** core methods according to their specifications, and **write unit tests** for them in course_test.py. These tests will be graded for correctness and completeness when you submit this assignment.

Some notes:

1. You will have to define your own error classes, as specified by the docstrings.
2. You may add additional attributes to `Course`, but you **must** use the three providied attributes correctly. If is your responsibility to document any additional attributes you create.
3. Of course, you may define your own helper functions and methods! For this assignment, you are not required to submit tests for them, only for the core methods.
4. **Clarification (Oct 26)**: this method `missing_prereqs` should be recursive, i.e., if `self` has a missing prerequisite course CSC148, and CSC148 has a missing prerequisite CSC108, then both "CSC108" and "CSC148" should appear in the returned list.

Assignment Specification: Term Planner

Now open planner.py, which contains `TermPlanner` and `parse_course_data`. Before we get started actually using our `Course` objects to help us plan course schedules, we're going to make a detour into a more sophisticated way of creating our model.

Getting Course Data

It becomes old quite quickly to manually create course objects and add them to a tree. Instead, you are first going to write a helper function `parse_course_data(filename)`, which takes as input the filename of a file with the following format:

- On each line are two strings representing two names of courses, where the *first course is a prerequisite of the second*. You may assume courses are represented by single-word alphanumeric strings (no spaces or special characters).

- The lines can be in *any order*; there are no restrictions on which prerequisites have to be listed first, nor do all of the prerequisites for a given course need to be listed in consecutive lines!

Here is a sample file:

```
AWE100 AWE200
AWE200 AWE300
AWE101 AWE200
AWE300 AWE400
AWE199 AWE200
AWE250 AWE300
```

Note that this file encodes the exact same prerequisite information as the diagram above.

Your task in implementing `parse_course_data` is to read in such a file, create the necessary `Course` objects, *link them together by their prerequisite structure*, and then return the root (top-most) course.

Some notes:

1. You may assume the filename is valid (so don't worry about handling any "file not found" errors or the like).
2. You may assume the input files always have the correct structure (two strings per line).
3. **You may assume that the input files always give you a valid tree-like prerequisite structure**. (Again, this is false in the "real world", but good enough for our purposes.) However, it is your job to determine which course is the root!
4. You **must** use the "best practice" Python way of opening files for reading, found here.
5. You are not required to submit test cases for this function, although of course we strongly recommend that you test it!

Computing with `TermPlanner`

The class `TermPlanner` is responsible for answering two main questions regarding *course prerequisites* and *schedules*:

- Does my schedule satisfy all of the prerequisites?
- Here's a list of courses that I want to take. Create a valid term schedule for me, please!

In this assignment, we'll represent a *schedule* as a list of lists, where each nested list contains at most **five** strings, each string representing the name of a course. This is a simple example of a schedule over 3 terms:

```
[['CSC108', 'MAT135', 'SOC101', 'CHM138', 'CSC165'],
 ['CSC148', 'MAT136'],
 ['CSC207']]
```

Once a `TermPlanner` object is instantiated with a `Course`, it uses it to determine whether given schedules are valid or not. A **valid schedule** is one where:

- Every course in the schedule exists in the `Course` tree.

- There are no duplicate courses. (**Assume every course is a one-term course!**)
- Most importantly, every course is taken after all of its prerequisite courses have been taken.

Complete the two methods `is_valid` and `generate_schedule` according to their docstrings. You are not required to submit test cases for these two methods, though of course we'll be testing them, and so should you!

Note that you have considerable freedom in `generate_schedule`, because for a given list of courses there are possibly many different valid schedules that could work. For the purposes for this assignment, the schedule you output can be *any schedule satisfying the following properties*:

- It is a **valid** schedule (i.e., running `is_valid` on it should return `True`).
- Each term contains at most **five** courses.
- Most importantly, the schedule should be **greedy**: if a term currently has fewer than 5 courses, and another course can be taken at that time, then that course should be added to the term. Put another way, in the final schedule that's returned, if a term has fewer than 5 courses, it's because **no other untaken course** could be taken during that term.

The last condition exists mainly to rule out schedules consisting of only one course per term, which are considerably easier to produce.

Once again, as long as the schedule your algorithm outputs satisfies these properties, it will be considered correct. In particular, there is no limit on the number of terms your schedule can span.

Clarifications:

1. (Oct 27) You can assume the input of `generate_schedule` only has courses that appear in the course tree.
2. (Oct 27) If no valid schedule can be formed from the input courses, return an empty list.

Design

Even though we have provided the main two classes for you, there are still quite a few design decisions that you need to make. In particular, the `TermPlanner` methods are among the most complex you've written in this course, and so should be split into helper methods, possibly across both classes `Course` and `TermPlanner`. Like the first assignment, no method/function body should be longer than **15 lines**.

Submission instructions

1. Login to MarkUs, and create a group if necessary.
2. Does your code run?
3. Submit your files (additional files are acceptable):
    - course.py
    - planner.py
    - course_test.py

4. Go through this checklist. Make changes if necessary.
5. Does your code run?!
6. Read the marking scheme (forthcoming). 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 files, again. Then either go to step 4 or 11.
11. Congratulations, you're done! Go have some chocolate. :)

OR Courses (optional)

Now that you have your program up and running, let's throw in a twist. You've probably noticed in your own course planning that some courses require *either one course or another* as prerequisites. That is, there are courses whose prerequisites are an "OR" rather than an "AND"; we could say, for example, that CSC263 requires CSC236 *or* CSC240 as a prerequisite.

After you've submitted your work, try modifying your program to handle "OR" prerequisites as well. For simplicity, you might want to start with courses whose prerequisites are either *all* required, or only *one* of which is required. Once you've figured that out, you can try extending this to handle a course whose prerequisites are "AWE101, and either AWE150 or AWE151".

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