6.S04 Practice Quiz 3

This quiz is **open-book and closed-internet**. Feel free to use any physical materials you have brought, but you may not access resources online (except [funprog][funprog]). Proctors will be available to answer administrative questions and clarify specifications of coding problems, but they should not be relied on for coding help.

Each problem is worth 25 points, except for problem 3 (which has subparts and is worth 30 points), for a total of 80 points.

You *must* submit your quiz via [funprog][funprog] before the deadline in order to receive credit. This quiz assumes you have Python 2.7 installed on your machine.

The `resources` directory contains **Python documentation** for commonly-used data structures.

Question 1: `count_viable`

Ilia is on tour running an animal circus. He has trained a number of different animals, each with a particular weight. Unfortunately, he is limited by the weight capacity of his trailer. **A set of animals `s` is *viable* if the total weight of the animals in `s` is at most the trailer's weight capacity.**

Your task is to count the number of *viable* sets of animals, given the weights of the animals and the weight capacity of the trailer. We recommend that you perform a search over possible sets of animals.

For Question 1 you will implement the `count_viable()` method to the specification below.

INPUTS:

1. `weights` : a list of positive weights, where `weights[i]` is the weight of the `i` th animal.
2. `capacity` : the weight capacity of Adam's trailer, guaranteed to be positive.

OUTPUT:

The number of *viable* sets of animals. Recall that the different orderings of the same elements are *not* different sets. Don't forget the empty set (no animals)!

EXAMPLES:

1. `count_viable([2, 3, 4], 1)` should return `1` (only the empty set is *viable*).
2. `count_viable([1, 2], 3)` should return `4` .

Question 2: course requirements

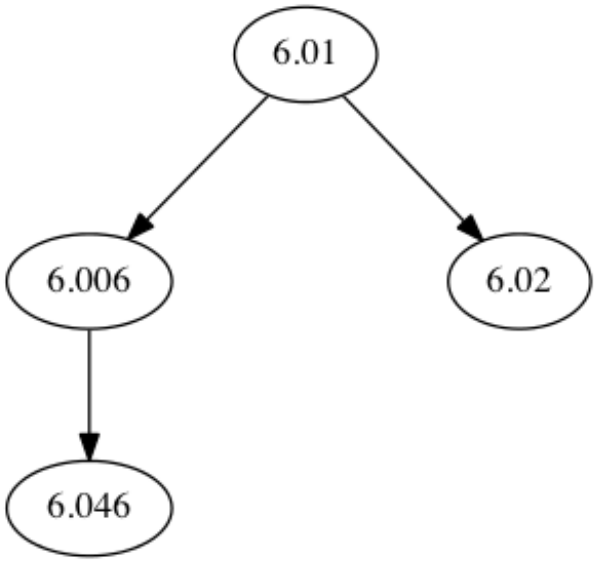
You are given a set of classes and their prerequisites. Your task is to find an order in which to take all of the classes so that the prerequisite requirements are satisfied. Two classes cannot be taken simultaneously.

One way to represent the classes and their prerequisites is a directed graph. We can represent each class as a node and the prerequisite requirements as directed edges. If A is a prerequisite for B, then we draw a directed edge from A to B.

For example, consider the following list of classes and their prerequisites:

- 6.01, Prereqs: None
- 6.006, Prereqs: 6.01
- 6.02, Prereqs: 6.01
- 6.046, Prereqs: 6.006

We can represent the relationships in the following directed graph:



We represent the directed graph as a dictionary `class_graph`, in which `class_graph[node]` is a list of `node`'s children. For the graph above, `class_graph` is the following.

```
{
  "6.01": ["6.006", "6.02"],
  "6.006": ["6.046"],
  "6.02": [],
  "6.046": []
}
```

To solve the original problem, we must now find an order in which to visit all of the nodes so that

each node is visited after its parents. Note that the solution is not unique – we only need to identify one. We outline an algorithm below. This problem is known as topological sorting.

One approach is to build up a desired ordering one class at a time. We maintain a list of classes that would be valid to take next. Initially, this valid list consists of all classes with no prereqs (nodes with no parents). Until we have taken all of the classes, we iteratively do the following.

1. Remove a class `X` from the valid list and add `X` to the ordering. (We choose to take class `X`.)
2. For each child `C` of `X`, mark that the `X` prereq for `C` has been satisfied. If `C` no longer has any unsatisfied prereqs, add `C` to the valid list.

You should convince yourself that this algorithm works. For Question 2 you will implement the `find_valid_ordering()` method to the specification below.

INPUTS:

1. `class_graph`: a directed graph, as a dictionary, that represents a set of classes and their prerequisites.

OUTPUT:

A list of all classes, in order, to take so that the prerequisites are satisfied.

EXAMPLE:

In the example above, one possible solution to `find_valid_ordering(class_graph)` is `["6.01", "6.006", "6.02", "6.046"]`.

Question 3: a database of classes

This question consists of 2 parts.

You are given a database of scheduling information about a set of classes over a semester. We make the following assumptions about the schedule:

1. A class meets at most once on any given day.
2. Meetings last exactly 1 hour.
3. Meetings can only begin on the hour at 09:00, 10:00, …, 16:00.
4. The semester consists of 15 weeks: Week `"1"`, Week `"2"`, …, Week `"15"`.

Each day in the semester is represented as a tuple: `(WEEK, DAY_OF_WEEK)`. For example, the fifth Friday of the semester is denoted as `("5", "Friday")`. Note that **both elements in the**

tuple are strings.

The database consists of two parts, the `default_db` database and the `update_db` database. `default_db` contains the original schedule that is now **outdated**. `default_db` is a list of weekly entries. Each entry is itself a list of strings: a class name, a time of day, and a day of the week. The time is the hour at which the meeting starts, according to a 24-hour clock. These meetings were originally scheduled to occur every week.

The following is a small example `default_db` that contains three weekly entries.

```
[
  ["6.S04", "15", "Monday"],
  ["6.S04", "14", "Friday"],
  ["6.01", "10", "Tuesday"],
]
```

Next, the `update_db` database contains updates to the original schedule. There are two types of updates: ADDs and DELETEs. We represent `update_db` as a list of updates. Each update is itself a list of strings: an update type, a class name, a time, a day of the week, and a week number. Unlike the weekly entries in `default_db`, each update in `update_db` only gives information for a specific day. Each ADD update refers to a meeting that does not already exist in `default_db`. Each DELETE update refers to a meeting that exists in `default_db`.

Extending the example above, the following is an `update_db` that contains two updates.

```
[
  ["ADD", "6.S04", "12", "Thursday", "15"],
  ["DELETE", "6.S04", "15", "Monday", "15"]
]
```

After updating the original schedule in `default_db` with `update_db`, we see that our schedule should consist of the following meetings:

1. `"6.S04"` meets at `"15"` on `"Monday"` s on every week *except* Week `"15"`.
2. `"6.S04"` meets at `"12"` on `"Thursday"` in Week `"15"`.
3. `"6.S04"` meets at `"14"` on `"Friday"` every week.
4. `"6.01"` meets at `"10"` on `"Tuesday"` every week.

In the following parts, you will implement methods to retrieve information about the schedule. They require an implementation of the `build_rep()` method. `build_rep()` processes instances of the two databases and returns a new representation of the information. The output of `build_rep()` is passed into the other methods. Building effective representations should be very

helpful. **You should modify the default representation** Consider using objects to encapsulate the database contents.

The `default_db` and `update_db` databases will be provided as arguments to `build_rep()`, but not the other methods. For your convenience, you can look through raw text versions of the databases in the `resources/database/` folder. It might be useful to search these files when debugging. We do not recommend printing the entire databases at once, since they are quite large.

Part A: Test Cases 11-13

For Question 3 Part A you will implement the `get_class_days()` method to the specification below.

INPUTS:

1. `class_list`, a list of classes as strings
2. `rep`, the representation of `default_db` and `update_db` produced by `build_rep()`

OUTPUT:

A list of lists `class_days` where `class_days[i]` is a list, in no particular order, of all days on which `class_list[i]` meets. If `class_list[i]` never meets, then `class_days[i]` should be an empty list `[]`. The days should be represented as string tuples of the form `(WEEK, DAY_OF_WEEK)`.

EXAMPLES:

Using the above example,

1. `get_class_dates(["6.S04"], rep)` could return `[[("1", "Monday"), ("2", "Monday"), ..., ("14", "Monday"), ("15", "Thursday"), ("1", Friday), ("2", "Friday"), ..., ("15", "Friday")]]`.
2. `get_class_dates(["18.01", "6.01"], rep)` could return `[[], [("1", "Tuesday"), ("2", "Tuesday"), ..., ("15", "Tuesday")]]`.

Part B: Test Cases 14-16

For Question 3 Part B you will implement the `get_late_classes()` method to the specification below.

INPUTS:

1. `time`, a time of day as a string. Only whole number strings `"9"`, `"10"`, ... , `"16"` are allowed.
2. `rep`, the representation of `default_db` and `update_db` produced by `build_rep()`.

OUTPUT:

A list, in no particular order, of all classes that never meet before `time` during the semester.

EXAMPLES:

Using the above example,

1. `get_late_classes("10", rep)` could return `["6.S04", "6.01"]`.
2. `get_late_classes("12", rep)` should return `["6.S04"]`.
3. `get_late_classes("14", rep)` should return `[]`.