The Alchemist's Laboratory - a package for any alchemist!

This assignment asks you to refactor existing code and package it in a form that can be tested, installed and accessed by other users.

The code to actually solve the problem is already given, but as roughly sketched out code in this notebook.

Your job consists in converting the code into a formally structured package, with unit tests, docstrings, a command line interface, and demonstrating your ability to use `git` version control.

The exercise will be semi-automatically marked, so it is *very* important that you adhere in your solution to the correct file and folder name convention and structure, as defined in the rubric below. An otherwise valid solution which doesn't work with our marking tool will **not** be given credit.

First, we set out the problem we are solving, and its informal solution. Next, we specify in detail the target for your tidy solution. Finally, to assist you in creating a good solution, we state the marks scheme we will use.

Alchemists and their 2-shelf standard laboratories

The code we will be working with describes an imaginary laboratory, where different substances can exist. The alchemists who own this type of lab can only have two shelves. Each shelf holds any number of substances, with potentially multiple copies of the same substance.

Each substance is uniquely identified by its name. We will assume that the reactions follow a very simple rule: a substance called `{X}` reacts with *exactly* those substances called anti`{X}`. Here, *{X}* is a placeholder for any value. For example, nutronium can interact with antinutronium, but not with phlebotinum or with antiflamen; similarly, antiA1 interacts with A1 but not with A. If the alchemist attempts a reaction that can happen (for instance, antivolpus and volpus), that reaction always succeeds.

The alchemist will try items from the lower shelf until finding one that can react with something from the upper shelf (the alchemist is also smart enough to know which things can react, so does not try impossible reactions). Substances from the lower shelf are tried in order, but the substance from the upper shelf is chosen randomly from the possible targets.

Our data structure for Merlin's laboratory on a certain day could be something like:

In [1]:
```
merlins = {
    'lower': ['involucrata', 'serpyllum', 'antifirma', 'antieycholra', 'psittaccina'
],
    'upper': ['alcea', 'antiinvolucrata', 'campanula', 'antiserpyllum', 'antiinvoluc
rata', 'firma'],
}
```

Dáithí, Merlin's helper, set the substances in such order following Merlin's instructions. Merlin - as any professional alchemist - knows which substances he needs to combine without even reading the label. He starts taking one at a time, in order, from the lower shelf, and with a simple gaze finds one that reacts with it from the upper one.

With this set-up, Merlin would take `'involucrata'` from the lower shelf, and get the first `'antiinvolucrata'` that he sees, therefore producing one of two posible outcomes:

```
merlins = {
    'lower': ['serpyllum', 'antifirma', 'antieycholra', 'psittaccina'],
    'upper': ['alcea', 'campanula', 'antiserpyllum', 'antiinvolucrata', 'firm
a'],
}
```

or

```
merlins = {
    'lower': ['serpyllum', 'antifirma', 'antieycholra', 'psittaccina'],
    'upper': ['alcea', 'antiinvolucrata', 'campanula', 'antiserpyllum', 'firm
a'],
}
```

Dáithí, as learner of the black-magic craft of software engineering, has come up with some code that helps him to simulate what Merlin is going to do and how many reactions are going to happen. Since all these reactions produce three hours of non-stop rain, he wants to inform Brigid - the weather forecast wizard - before it happens. This way she can put these numbers into her analysis and tell the people whether they need to take their rainshades.

Dáithí has created the following functions to simulate Merlin's steps. He is using the structure shown above to represent the laboratory, the shelves and the substances. Each substance is represented by a string.

The first function `can_react` returns `True` or `False`, depending on whether two substances react together.

In [2]:
```
def can_react(substance1, substance2):
    return (substance1 == "anti" + substance2) or (substance2 == "anti" + substance1
)
```

Then he has another function that updates the shelves, removing the substances from the lower and upper shelves, by giving the name of the substance from the lower shelf and the position of the opposite substance taken from the upper shelf:

In [3]:
```
def update_shelves(shelf1, shelf2, substance1, substance2_index):
    index1 = shelf1.index(substance1)
    shelf1 = shelf1[:index1] + shelf1[index1+1:]
    shelf2 = shelf2[:substance2_index] + shelf2[substance2_index+1:]
    return shelf1, shelf2
```

He created a function that generates a reaction as Merlin would do it. It goes after each substance in the lower shelf and makes it react with an opposite substance from the upper shelf.

In [4]:
```
import random

def do_a_reaction(shelf1, shelf2):
    for substance1 in shelf1:
        possible_targets = [i for i, target in enumerate(shelf2) if can_react(substa
nce1, target)]
        if not possible_targets:
            continue
        else:
            substance2_index = random.choice(possible_targets)
            return update_shelves(shelf1, shelf2, substance1, substance2_index)
    return shelf1, shelf2
```

Finally, Dáithí created a function that runs all the possible reactions, prints how many happened, and returns the final state of the shelves.

In [5]:
```
def run_full_experiment(shelf1, shelf2):
    count = 0
    ended = False
    while not ended:
        shelf1_new, shelf2_new = do_a_reaction(shelf1, shelf2)
        if shelf1_new != shelf1:
            count += 1
        ended = (shelf1_new == shelf1) and (shelf2_new == shelf2)
        shelf1, shelf2 = shelf1_new, shelf2_new
    print("Total number of reactions: {}".format(count))
    return shelf1, shelf2
```

Dáithí can now simulate an experiment by defining the two shelves, and calling `run_full_experiment`:

In [6]:
```
lower_shelf = ["A", "antiB", "C"]
upper_shelf = ["antiC", "antiA"]

final_shelves = run_full_experiment(lower_shelf, upper_shelf)
print("Final contents\n\t - lower:{}\n\t - upper:{}".format(*final_shelves))
```

```
Total number of reactions: 2
Final contents
         - lower:['antiB']
         - upper:[]
```

When Brigid found that Dáithí was giving her the exact number of reactions and therefore her forecasts were more accurate, she started to talk about it to other wizards. And other wizards were sending express pigeons to Dáithí asking him to calculate the same for the alchemists in their kingdoms.

Dáithí couldn't handle all the requests on time so he decided the best way to proceed was to create a library and publish it in magit-central so any alchemist helper could use it to know what will happen in their laboratories - but also, even more importantly, they could help to make it better.

However, Dáithí doesn't know how to create a library for others to use, and here's where your help is required!

Packaging the reactions: your mission!

You must submit your exercise solution to Moodle as a single uploaded Zip format archive. (You must use only the zip tool, not any other archiver, such as .tgz or .rar. If we cannot unzip the archiver with zip, you will receive zero marks.)

The folder structure inside your zip archive must have a single top-level folder, whose folder name is your student number, so that on running unzip this folder appears. This top level folder must contain all the parts of your solution. You will lose marks if, on unzip, your archive creates other files or folders at the same level as this folder, as we will be unzipping all the assignments in the same place on our computers when we mark them!

Inside your top level folder, you should create a `setup.py` file to make the code installable. You should also create some other files, per the lectures, that should be present in all research software packages. (Hint, there are three of these.)

Your tidied-up version of the solution code should be in a sub-folder called `alchemist` which will be the python package itself. It will contain an `__init__`.py file, and the code itself must be in a file called `laboratory.py`. This should define a class `Laboratory`; instead of a data structure and associated functions, you must refactor this into a class and methods.

Thus, if you run python in your top-level folder, you should be able to do `from alchemist.laboratory import Laboratory`. If you cannot do this, you will receive zero marks.

You must create a command-line entry point, called `abracadabra`. This should use the entry_points facility in `setup.py`, to point toward a module designed for use as the entry point, in `alchemist/command.py`. This should use the argparse library. When invoked with `abracadabra alchemist.yml` the command must print on standard output the final state of the shelves. If run with the optional flag `--reactions` then it should only show the number of reactions produced.

The `alchemist.yml` file should be a yaml file containing a structure representing the substances on the lower and uppper shelves. Use the same structure as the sample code above, even though you'll be building a `Laboratory` object from this structure rather than using it directly.

You must create unit tests which cover a number of examples. These should be defined in `alchemist/tests/test_laboratory.py`. Don't forget to add an `__init__`.py file to that folder too, so that at the top of the test file you can say "`from ..laboratory import Laboratory`". If your unit tests use a fixture file to DRY up tests, this must be called `alchemist/tests/fixtures.yml`. For example, this could contain a yaml array of many laboratories' setups. Remember to also include at least a negative test. More specific, a test that checks that a `TypeError` is raised when the program is called with a different number of shelves than the accepted.

You should `git init` inside your student-number folder, as soon as you create it, and `git commit` your work regularly as the exercise progresses.

Due to our automated marking tool, **only** work that has a valid git repository, and follows the folder and file structure described above, will receive credit.

Due to the need to avoid plagiarism, do *not* use a public github repository for your work - instead, use git on your local disk (with `git commit` but not `git push`), and *ensure* the secret `.git` folder is part of your zipped archive.

Marks Scheme

Note that because of our automated marking tool, a solution which does not match the standard solution structure defined above, with file and folder names exactly as stated, may not receive marks, even if the solution is otherwise good. "Follow on marks" are not guaranteed in this case.

- Code in `laboratory.py`, implementing the full experiment reaction (5 marks)
  - Which works (1 mark)
  - Cleanly laid out and formatted - PEP8 (1 mark)
  - Defining the class `Laboratory` (and maybe `Substance`) with a valid object-oriented structure (1 mark)
  - Breaking down the solution sensibly into subunits (1 mark)
  - Structured so that it could be used as a base for other type of reactions (1 mark)
- Command line entry point (5 marks)
  - Accepting a laboratory definition text file as input (1 mark)
  - With an optional parameter to output the number of reactions (1 mark)
  - Which prints the result to standard out (1 mark)
  - Which correctly uses the Argparse library (1 mark)
  - Which is itself cleanly laid out and formatted (1 mark)
- setup.py file (5 marks)
  - Which could be used to pip install the project (1 mark)
  - With appropriate metadata, including version number and author (1 mark)
  - Which packages code (but not tests), correctly. (1 mark)
  - Which specifies library dependencies (1 mark)
  - Which points to the entry point function (1 mark)
- Three other metadata files: (3 marks)
  - Hint: Who did it, how to reference it, who can copy it.
- Unit tests: (5 marks)
  - Which test some obvious cases (1 mark)
  - Which correctly handle random selections (1 mark)
  - Which test how the code fails when invoked incorrectly (1 mark)
  - Which use a fixture file or other approach to avoid overly repetitive test code (1 mark)
  - Which are themselves cleanly laid out code (1 mark)
- Version control: (2 marks)
  - Sensible commit sizes (1 mark)
  - Appropriate commit comments (1 mark)

Total: 25 marks

some hints

- Remember, this has to work only for the world-wide 2-shelf standard laboratories.
- Also, empty shelves are allowed.
- And... do `antianti` products exist?