Week 3 Exercises

Physics 91SI Spring 2011 Handout 12

Alex Ji and Zahan Malkani

From here on out, you're always going to be working either in a repository that you create or a repository that you clone. Eventually you'll have to push it back, which means you have to commit things. We won't tell you to commit your code at the various stages of development, but you should do so whenever you have written some working piece of code.

Three-Vector Interface

Create a new Mercurial repository in your working directory titled `exercise3vector`. In this repository, write an interface and tests for a vector class in 3D Euclidean space (Math 51 R^3). Name the class `ThreeVector`. It is up to you what methods and instance variables you include, but make it as useful as possible. For instance, you will want to be able to add two different `ThreeVector`s and get the resulting `ThreeVector`. If you're having trouble thinking of useful things, ask Alex or Zahan. You might want to start with pencil and paper.

The interface should be specified by writing a "skeleton" class. In other words, create a class called `ThreeVector` that has the names of any "public" methods defined but with NO IMPLEMENTATION. You should include comments at the beginning of each method detailing how the method is supposed to be used. See below for an example.

Remember that your tests should fully describe how the function is supposed to work. You should make them runnable by `nosetests` by naming them properly. Write at least 5-10 tests for each function in the vector class.

When you're done, push this repository into YOUR OWN submissions folder (there should be an empty repository named `exercise3vector` in there).

Example of the class file:

```
#Interface by Alex Ji
#Implementation by ...

class ThreeVector:
    def method1(self, var1, var2):
        """This is a comment describing what method1 is supposed to do
        and what it returns, if anything. Note that you can have multiple
        lines inside this quote, but you shouldn't make it too long."""
        pass
    def method2(self, var1, var2):
        """Here's another method just for kicks. Note that you should probably
        describe what variables are passed into the methods, and you should
        also give them names that aren't "var1" and "var2"."""
        pass
```

C-shell/`bash` Environment Variables

If there is someone else who has completed their ThreeVector interface, skip this and do the ThreeVector implementation before coming back to this problem. Otherwise, start on this and pause when someone else has finished their ThreeVector interface.

Environmental variables are a super useful tool that come built in to your shell. They are generally used to maintain configuration options and keep 'state' across your computer. They are stored in the 'environment' of the computer, able to be accessed from any program or by you at any point. The specific syntax on how to set and use them differs for different shells (C-shell, Bash), but we will generally describe variables that are useful. Remember these variables are set across shells, so something you set in Bash will be propogated to C-Shell and vice-versa.

Note: These variables are often set in special scripts that are run each time you start a shell and login. Namely there are

C-Shell `.cshrc`

Bash `.bashrc`

The syntax to set variables is as follows:

C-Shell `setenv VAR value`

Bash `export VAR=value`

SHELL

In case you want to see which shell you are currently working in, `echo $SHELL` will achieve that. Do not set this.

PATH

Used to keep track of where the operating system searches from programs invoked on the command line. `hg` and `ls`, all live somewhere. **Add** more paths to this with
`setenv PATH $PATH:/other/path`

PYSTARTUP

Useful for Python programmers (you!), this is the name of a **file** that is invoked when the `python` interpreter is run on the command line. (You should probably make it a hidden file.) Mine has a useful trick:

```
try:
    import readline
except ImportError:
    print("Module readline not available.")
else:
    import rlcompleter
    readline.parse_and_bind("tab: complete")
```

Now try hitting tab while typing in the command line interpreter.

PHYS_91SI_SUB

In your `.cshrc`, you might want to set a shortcut to the awfully long path to our shared class space: `/afs/ir.stanford.edu/class/physics91SI/`

Three-Vector Implementation

Now, find someone else who has also completed their Three-Vector interface. Get their SUNet ID and clone the repository containing their interface into your working directory (you'll probably have to name it something different in your work directory, like `exercise3vector-implementation`). Implement the interface that they have started, making sure to add your name as a comment at the top. We suggest using `numpy` arrays in your implementation. The link for `numpy` documentation can be found on the Week 3 outline we gave you on Tuesday.

If you're having trouble, you may ask Alex or Zahan, but DO NOT ASK THE PERSON WHO WROTE THE INTERFACE ANY QUESTIONS.

Don't take more than 20-30 minutes to do this, since we want you to get to "Testing Averages". If you've worked on it for 30 minutes and you're still not done, just commit the code as is. Push the code into the repository you pulled from. (aka the default, the one in your partner's submissions folder, the one not in your own submissions folder).

We'll announce when we've reached the last 10 minutes of the section. At this point, you can talk to the person who wrote the interface that you implemented and discuss difficulties, ambiguities, etc.

Testing Averages

This exercise is based on a Software Carpentry exercise located here (please look at it only after you've completed this exercise): `http://software-carpentry.org/4_0/test/exercises/`

The results of a set of experiments are stored in a file, where the i-th line stores the results of the i-th experiment as a comma-separated list of integers. There has been code written to find the experiment with the lowest average, and your task is to write tests for it.

Clone the repository called `exercise3averages` from the `src` folder in the `physics91SI` directory into your work directory. There is a file called `averages.py` with functions `min_avg` and `avg_line`. Read these functions to see how they work.

First, refactor `min_avg` so that it can be tested without dependence on external files. See `http://software-carpentry.org/4_0/test/interface/` and search for "refactor" if you are confused.

Next, write Nose test cases for the two original functions. Don't forget to consider what happens with empty files. Call the file `test_averages.py`. You can assume that other than the possibility of being empty, the file is well formatted (i.e. each line is a comma-separated list of integers).

The StringIO object will be helpful:
`http://docs.python.org/release/2.6.4/library/stringio.html`

While reading the code or writing tests, you might notice that the function specification is ambiguous. It doesn't specify what to return when two or more lines have the same average. Decide what a suitable response for this is, and add tests describing that to a file called `test_same_averages.py`. Write a comment at the top of this test file saying what you decided as the specification.

After you have written the tests, implement `same_averages.py` to the specification you created through those tests.

Push your results to `exercise3averages` in your submissions folder.

Four Vector

This is not required, but if you have time consider how you would extend the three-vector interface for a four-vector class (as used in special relativity).