Lab #11

Physics 91SI Spring 2013

Objective: In this lab, you will practice writing code so that other people can use it and understand it without becoming frustrated by it.

As usual, log on to `corn` and clone over the starter repository:

```
hg clone /afs/ir.stanford.edu/class/physics91si/src/lab11 lab11
```

Remember to `hg commit` often to save your changes, and submit your code at the end of class.

Part 1: A Calculator... with Units!

In 1999, the Mars Climate Orbiter stopped communicating with ground control and was never heard from again. Engineers at NASA later realized the program on the ground which was executing maneuvers was giving instructions in pounds of force, while the orbiter was designed to accept newtons. You're going to help prevent future disasters like this by making a calculator that can understand units and do calculations with them.

Open up `unitcalc.py`. This program is designed to be run from the command line with `./unitcalc.py <args>`, where `<args>` is a string of the form:

```
<number> <unit> <operator> <number> <unit>
```

separated by spaces. Your task is to implement this behavior.

- In `unitcalc()`, write code that will accept a string `s` of the form indicated above, parse it into its components, and, if the operation makes sense, perform it and return the result.
- You should include support for at least the operations `+` and `-`, and for quantities that have one of two basic units. If you have time before beginning Part 2, move on to `*` and `/`, and include support for compound units like `m/s` and for quantities that have no units at all.
- Include error handling so that when the user screws up the input, it throws a helpful error. Make new subclasses of `Exception` to clarify what the problem is. If applicable, such a subclass should have attributes that describe in more detail what the error was.
- For any new functions or classes that you write, include a docstring of a suitable length.

Part 2: Break Your Partner's Code

Find a group of two or three and have each person give his or her code to someone else. Now try to break the code you were given: think of things you can put in the command line that your partner didn't take into account! While you're waiting to start this part, move on to Part 3.

Part 3: Turn It into an Interactive Interpreter

Now your task is to use the function you wrote above inside an interactive interpreter, much like `ipython` – except all it does is calculate things for you. Inside of `main.py`, write code to check if the user provided no command-line arguments (i.e. ran the script using just `./unitcalc.py`) and, if that's true, go into an interactive mode: it will accept input from the user multiple times without quitting. In addition to the behavior you implemented already, implement whichever of the following that you want during the time remaining. They're arranged in an order you might find convenient. As usual, make sure you deal with user error, and write docstrings for additional functions you declare.

- Implement the basic loop that displays a prompt (you can design it yourself) and accepts user input, then calls the appropriate functions. Suggestion: use `raw_input()`.
- The challenge here is that you can't let the function you wrote in Part 1 throw errors, or else the interpreter will quit. Use `try...except` clauses to handle the errors and, if applicable, display a helpful message to the user.
- If you haven't yet implemented support for the operators `*` and `/` and for quantities with compound units like `m/s` or with no units at all, now is a good time.
- Include support for commands: `<cmd> <args> <opt-args>`. Suggestion: write the desired behavior as a new function inside of `unitcalc.py` and then use `locals()` (look it up or ask for help) to write code that checks if the provided command is a function you have declared and, if it is, call it.
    - `help <opt-cmd>`. If entered on its own or followed by a command that the interpreter recognizes, it will display useful information. In the case of commands, `help` should print the command's docstring.
    - `quit`. This is self-explanatory. Suggestion: to signal the basic loop to break, raise a custom exception that will be caught in the basic loop.
    - `avg <arg1> ... <argn>`. Returns the average of the arguments. This can get tricky with arguments that have units.
    - Anything else you can think of!
- Include support for variable assignment: `<varname> = <value>`. Suggestion: use `locals()` (look it up or ask for help). Variables should be usable in place of numbers by putting a `$` in front, e.g. `$varname + 3 m`.
- Extend `unitcalc()` to accept strings with more than one operator (and with parentheses), like `(3 m + 5 m) / 2 s`.