Computer Science CS134C (Fall 2018)
Duane A. Bailey
Laboratory 4
Debugging (due Thursday/Friday)

Objective. Developing skills that help find mistakes in programs.

This week we're going to look at a number of small scripts or programs that are not currently functional. Each of the scripts contains a variety of errors:

1. *Syntax errors.* These are errors that are identified by python before the program begins running. For example, if you misspell the keyword `for` as `fro`, python will complain that your syntax is incorrect. These errors, while annoying, are typically the easiest to correct.

2. *Runtime errors.* When the program begins running, there are some conditions that may unexpectedly arise that python will complain about. For example, suppose you compute the average of a list `speeds` in the following manner:

   ```
   meanSpeed = sum(speeds)/len(speeds)
   ```

   it is possible that `speeds` could be empty. If that happens, then the computation results in an attempt to divide by zero, an illegal operation:

   ```
   Traceback (most recent call last):
     File ''physics.py'', line 29, in <module>
       meanSpeed = sum(speeds)/len(speeds)
   ZeroDivisionError: division by zero
   ```

3. *Logic errors.* Sometimes when we translate an algorithm into code we may introduce mistakes in logic. If we pick a random character from a 6 character string by selecting a random integer between 1 and 6, we are forgetting that string indices start at 0, not 1. That's a mistake of logic.

As we investigate our scripts this week, we will want to carefully read the hints that python gives us. For example, when syntax error messages are printed, they include line numbers. The error is very likely at or before that line number. Fortunately, all modern editors show us line numbers for the current location.

When runtime errors occur, the python system often prints a *stack trace* or *traceback*. This is a list of lines in the program that are currently being executed. Information about the most recent line appears near the bottom of the trace. Carefully reading the stack trace can give you important clues about what your program was doing when it stopped running.

The hardest errors to fix are logical errors. Since these errors stem from bad assumptions about the state of the program, it is often helpful to get a better view of the program's variables. The judicious use of `print` statements can help to isolate sections of the code that lead to unexpected values in state. This process, however, can take a long time as you run multiple experiments with different `print` statements in different locations. A more interactive approach is to use a *debugger*. In this course we'll make use of the *python debugger*, `pdb`. We'll discuss its use in lecture and lab, but you can find good documentation of `pdb` on the python website.

To get started, you should clone the `lab4` repository in the usual manner:

```
git clone ssh://22xyz9@davey.cs.williams.edu/~cs134/22xyz9/lab4.git ~/cs134/lab4
```

Required Tasks. We would like you to debug at least two of the scripts we've included in this week's repository. These scripts have varied purposes:

1. The `prime.py` script. Ideally, when this script is working, you can run it as follows:

   ```
   -> python3 prime.py 17
   17 is prime.
   ```

   The script contains a number of errors that should be fixed before it is fully functional. You should think about values that would be good to test.

2. The `shuffle.py` script. This script reads all the arguments that are provided and prints them, possibly shuffled. For example, the following is a possible behavior:

   ```
   -> python3 shuffle.py a b c
   c a b
   ```

   Again, think about ways that you could test the functioning of this script.

3. The `rot13.py` script. This is an implementation of the rotate-by-13 cypher once used by unix. It reads input, translates it, and writes the rotated text on the output. You can use it in the following way to translate `hello` into `uryyb` (remember, input is ended by hitting `Control-D`):

   ```
   -> python3 rot13.py
   hello!
   uryyb!
   ```

   We've included a more extensive test as well. When the program works, you can run `rot13.input` through the script and test to see if it matches the ideal `rot13.output` (a list of *pangrams*) with:

   ```
   -> python3 rot13.py <rot13.input | diff - rot13.output
   ```

   Any differences reported suggest remaining errors.

4. The `cribbage.py` script. This script allows you to score a hand in cribbage. The precise scoring is described in the script and requires care. This is the hardest script to debug. You can run it in the following way to score a cribbage hand with three 2's and two 9's:

   ```
   -> python3 cribbage.py 2S 2D 2H 9C 9S
   The score is 12.
   ```

   (The three 2's count for 6, the two 9's count for 2, and there are two ways to sum to 15, for 4 more.)

Each of these scripts contains several errors. Your grade will depend on how carefully you've tested the scripts. Again, you need only correct two scripts, but more experience will make you a better programmer.

Submit your work. As you finish debugging each script, do not forget to add, commit, and push your work to the server for grading.

⋆