Objective. This week we're going to look at a number of small scripts or programs that are not currently functional. The goal of the lab is to give you practice on the journey to developing the skills to find mistakes in your programs. This lab considers code that might have three types of errors:

1. **Syntax errors.** These are textual errors that are identified by Python before it attempts to run the program. 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 definition, found in a file *physics.py*:

   ```python
   def meanSpeed(speeds):
       return sum(speeds)/len(speeds)
   ```

   It is possible that *speeds* is empty. If so, the computation attempts to illegally divide by zero:

   ```
   Traceback (most recent call last):
   File "physics.py", line 29, in meanSpeed
   return 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* (like the builtin debugger *pdb*) or a web-based program visualizer, like the Python Tutor.

To get started, you should clone the lab04 repository in the usual manner:

```
git clone https://evolene.cs.williams.edu/cs134-labs/22xyz3/lab04.git ~/cs134/lab04
```

Remember to replace 22xyz3 with your CS account.

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

1. **The swedishpuzzle.py script.** This script is inspired by a Swedish word puzzle game called Rövarspråket whose rules are described in the script. Ideally, this script takes a letter and a word as command line arguments and prints a new word according to the rules. You can run the script as follows:

   ```
   $ python3 swedishpuzzle.py o stubborn
   sostotubobboronoron
   ```

   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.

Each of these scripts contains several errors. Your grade will depend on how carefully you've tested the scripts. A reminder that you will want to add a call to the test() function in if `__name__ == "__main__":` to run your doctests. Again, you need only correct two scripts, but more experience will make you a better programmer.
Submit your work. As you finish identifying and removing bugs in each script, do not forget to add, commit, and push your work to the server for grading.

Perpetual Grading Expectations. In addition to the lab-specific grading guidelines contained above and in GradeSheet.txt, remember that you are assessed on whether your code meets the general course expectations. For full coverage of what style entails, please see the Style Guide.

As a guide to avoiding common errors, make sure to follow these guidelines:

* Always sign the honor code. It protects your work.
* Follow the 80-character rule. This makes your code more readable.
* Use meaningful variable names. Help the reader understand your code.
* Comment your code. Logic that was hard to develop is often hard to understand.
* Write entirely new tests of your own design. Ours can be improved.
* Every doctest should be meaningfully different. Address standard use and unusual “corner” cases.
* When done, review the lab handout and make sure you’ve addressed the requirements.

*