Scaffolding assignment
Due Friday, 2013 Oct 4

UCSC BME 205 Fall 2013
Bioinformatics: models and algorithms

(Last Update: 16:22 PDT 24 September 2013)

Purpose

The purpose of this assignment is to build a program that doesn't do very much, but which has a structure similar to other programs you will write later in the quarter. You will likely start writing those programs by gutting this one and putting in new code to replace the trivial functions of this program with ones that do the tasks of the specific assignment.

A program intended to be reused in this way is sometimes called a *template* or a *scaffold*. (The term *scaffolding* in the pedagogic literature refers to the teacher breaking a complex task into easier subtasks for the student, which is also appropriate for this assignment.)

The task

The basic task for this program is a fairly trivial one in Python: break input text into words and count the frequency of each word. It is, perhaps, not a very bioinformatic task, but we will later do similar things in more bioinformatic contexts.

The program should be a UNIX filter, that is, it should read from standard input (stdin) and output to standard output (stdout). If there are any information or error messages, they should be sent to the standard error output (stderr), not stdout.

The input will be plain English text using the standard ASCII character set (not some proprietary format like Microsoft .doc files, and not including ligatures, accented characters, or Unicode). Obviously, more complicated versions of this program would be written that handle different languages, more complete character sets, and a variety of file types, but our intent is to build a scaffold that is easily gutted, so we are keeping the functionality to a minimum here.

The output should be two columns separated by a tab: the first column containing a word, the second column containing the number of occurrences of the word in the input. Because I'll be checking the output of the turned in homework with simple comparison programs, there should no comments or decorations of the output. Anything other than word-tab-number pairs, one pair per line, will be invalid output.

Aside: *As a general rule of thumb for writing bioinformatics programs, you should be flexible and forgiving of minor variations from a standard when reading input, but very scrupulous to follow standards precisely when producing output. That way your program will not be the obstruction that causes a pipeline to fail.*

There are several possible orders for the word-count pairs in the output: unsorted, ascending counts, descending counts, and alphabetical. The order of the words in the output will be specified by a command-line argument to the program. This part of the assignment is to make sure that your scaffold has an argparse

command-line parser in it, as you will need that in most subsequent assignments.

So that I can test everyone's program automatically, your program file must be named "wordcount" (not "word_count", not "hw1", not "wordcount.py"). I will issue one of the following commands:

```
wordcount --ascend < foo.txt > foo-ascending
wordcount --descend < foo.txt > foo-decending
wordcount --alpha < foo.txt > foo-alphabetical
wordcount --alpha --descend < foo.txt > foo-alphabetical
```

If both `--alpha` and `--descend` are specified, then the output should be in reverse alphabetical order. If the program does not provide identical outputs to my "correct" program, you will be required to redo the assignment. There are no partial points on this assignment for "almost working" programs.

Because different correct outputs are possible with the same input, I will not test an "unsorted" option, and you need not provide one. Unsorted as a default makes sense in some contexts, where the output will be used by another program, and sorting would be unnecessary overhead.

Aside:*The astute programmer will have noticed that I have provided a nearly impossible task above. When sorting in ascending or descending order, there can be several ties---how can you guarantee identical output when the order is not completely specified? This sort of ambiguity in specifications is extremely common, so you should always be looking for it---it is one place bugs often creep in. Don't worry, I won't leave you hanging this time.*

As a tie-breaker, when sorting by counts, words with identical counts should be output in alphabetical order. The ascending/descending choice applies only to the counts, so the output

```
alpha   10
beta    10
aaa     3
```

would be a correct output file for a `--descend` option.

Aside:*The astute programmer would still be irritated with me for incomplete specs, as I've never defined what I mean by a "word", nor exactly what I mean by "alphabetical". These are not trivial questions---library schools spend a fair amount of time discussing different alphabetization schemes and what the standards are in different systems. Because this assignment is about building a scaffold, not about natural-language processing, we are going to use a trivial definition.*

For the purposes of this assignment, a "word" is a contiguous sequence of characters from the set {abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ'}. All other characters in the input will serve as separators.

For the purposes of this assignment, words are "alphabetically" ordered by the Python string comparison operator. Note that this is not the normal notion of alphabetical order, since the case of the letters matters, and "Z" comes before "a".

Program structure

Because the purpose of this assignment is for you to build a reusable template, I'm going to require that certain features of Python be used and to strongly recommend a particular structure for the program.

I will require

- a generator function using the "yield" statement,
- using the argparse module to parse command-line arguments,
- using the help-string features of argparse to provide user documentation,
- a docstring at the beginning of the program explaining how to use the program (including this docstring in the argparse help can reduce the chance of conflicts between what the programmer is told and what the user is told),
- a docstring for every function in the program,
- meaningful, understandable names for all variables,
- comments for every variable that has a scope of more than 5 lines (except iterator variables in for statements),
- making the program be a UNIX executable with public execute permission, and
- that testing the program on a unix-like (Linux or Mac OS X) system.

I strongly recommend that that file start with the line

```
#!/usr/bin/env python2.7
```

so that the UNIX, Linux, or OS X operating system can find the right version of python to run the program. Note that Mac OS X systems are likely to have an older version of Python installed, but it is fairly easy to install Python 2.7 (installers for the most systems are available at http://www.python.org/getit/).

Warning for Windows users: *Windows uses a different line-ending convention from other systems, using the old teletype convention of a separate carriage return and line feed, rather than a single new-line character. When you transfer a Windows text file to a unix-like system, the extraneous carriage returns can cause problems. Some transfer programs will remove the extra characters, but if you encounter a Windows text file on a unix-like system, you can strip out the junk with the "tr" command:*

```
tr -d '\r' < file-with-returns > file-without-returns
```

If you edit your program on a Windows machine, you may need to strip out the returns before it will run on other systems.

I strongly recommend that the program be broken into four major functions:

- a read_word generator function that takes a file-like object as a parameter and yields one word at a time,
- a parse_arguments function that sets up the argparse command-line parser and calls it on the arguments to the program,
- an output function that takes a "dict" hash table and a formatting argument and outputs the contents of the dict, and
- a main function that calls the other three.

Here is an example of what the main function might look like:

```
def main(args):
    """parses command line arguments, reads text from stdin, and
    prints two-column word-count pairs to stdout in the order specified
    in the command line arguments
    """
    options = parse_args(args)
    counts = collections.defaultdict(int)
        # counts['abc'] is the number of times the word 'abc' occurs
        # in the input.
        # defaultdict used to make counts of unseen words be 0
```

```
    for word in read_word(sys.stdin):
        counts[word] += 1
    print_output(sys.stdout, counts, options.order)

if __name__ == "__main__" :
    sys.exit(main(sys.argv))
```

The mysterious boilerplate at the end is commonly found in Python files (or something similar is used). If the Python file is executed directly by Python, then the "main" function is called with the system-provided command-line arguments and the program exits with the value returned by main. But if the Python file is imported into another Python program, then the "main" function is not automatically called, though all the functions in the file are available for use in the importing program. We'll look at building modules for importing into other programs in later assignments.

Late added notes