Shell Scripts

Objectives

- Write a shell script that runs a command or series of commands for a fixed set of files.
- Run a shell script from the command line.
- Write a shell script that operates on a set of files defined by the user on the command line.
- Create pipelines that include user-written shell scripts.

We are finally ready to see what makes the shell such a powerful programming environment. We are going to take the commands we repeat frequently and save them in files so that we can re-run all those operations again later by typing a single command. For historical reasons, a bunch of commands saved in a file is usually called a shell script (../../gloss.html#shell-script), but make no mistake: these are actually small programs.

Let's start by putting the following line in the file `middle.sh`:

```
head -20 cholesterol.pdb | tail -5
```

This is a variation on the pipe we constructed earlier: it selects lines 16-20 of the file `cholesterol.pdb`. Remember, we are *not* running it as a command just yet: we are putting the commands in a file.

Once we have saved the file, we can ask the shell to execute the commands it contains. Our shell is called `bash`, so we run the following command:

```
$ bash middle.sh
```

```
ATOM     14  C           1      -1.463  -0.666   1.001  1.00  0.00
ATOM     15  C           1       0.762  -0.929   0.295  1.00  0.00
ATOM     16  C           1       0.771  -0.937   1.840  1.00  0.00
ATOM     17  C           1      -0.664  -0.610   2.293  1.00  0.00
ATOM     18  C           1      -4.705   2.108  -0.396  1.00  0.00
```

Sure enough, our script's output is exactly what we would get if we ran that pipeline directly.

Text vs. Whatever

We usually call programs like Microsoft Word or LibreOffice Writer "text editors", but we need to be a bit more careful when it comes to programming. By default, Microsoft Word uses `.docx` files to store not only text, but also formatting information about fonts, headings, and so on. This extra information isn't stored as characters, and doesn't mean anything to tools like `head`: they expect input files to contain nothing but the letters, digits, and punctuation on a standard computer keyboard. When editing programs, therefore, you must either use a plain text editor, or be careful to save files as plain text.

What if we want to select lines from an arbitrary file? We could edit `middle.sh` each time to change the filename, but that would probably take longer than just retyping the command. Instead, let's edit `middle.sh` and replace `cholesterol.pdb` with a special variable called `$1`:

```
$ cat middle.sh
```

```
head -20 $1 | tail -5
```

Inside a shell script, `$1` means "the first filename (or other parameter) on the command line". We can now run our script like this:

```
$ bash middle.sh cholesterol.pdb
```

```
ATOM     14  C           1      -1.463  -0.666   1.001  1.00  0.00
ATOM     15  C           1       0.762  -0.929   0.295  1.00  0.00
ATOM     16  C           1       0.771  -0.937   1.840  1.00  0.00
ATOM     17  C           1      -0.664  -0.610   2.293  1.00  0.00
ATOM     18  C           1      -4.705   2.108  -0.396  1.00  0.00
```

or on a different file like this:

```
$ bash middle.sh vitamin-a.pdb
```

```
ATOM     14  C           1       1.788  -0.987  -0.861
ATOM     15  C           1       2.994  -0.265  -0.829
ATOM     16  C           1       4.237  -0.901  -1.024
ATOM     17  C           1       5.406  -0.117  -1.087
ATOM     18  C           1      -0.696  -2.628  -0.641
```

We still need to edit `middle.sh` each time we want to adjust the range of lines, though. Let's fix that by using the special variables `$2` and `$3`:

```
$ cat middle.sh
```

```
head $2 $1 | tail $3
```

```
$ bash middle.sh vitamin-a.pdb -20 -5
```

```
ATOM     14  C           1       1.788  -0.987  -0.861
ATOM     15  C           1       2.994  -0.265  -0.829
ATOM     16  C           1       4.237  -0.901  -1.024
ATOM     17  C           1       5.406  -0.117  -1.087
ATOM     18  C           1      -0.696  -2.628  -0.641
```

This works, but it may take the next person who reads `middle.sh` a moment to figure out what it does. We can improve our script by adding some comments (../../gloss.html#comment) at the top:

```
$ cat middle.sh
```

```
# Select lines from the middle of a file.
# Usage: middle.sh filename -end_line -num_lines
head $2 $1 | tail $3
```

A comment starts with a `#` character and runs to the end of the line. The computer ignores comments, but they're invaluable for helping people understand and use scripts.

What if we want to process many files in a single pipeline? For example, if we want to sort our `.pdb` files by length, we would type:

```
$ wc -l *.pdb | sort -n
```

because `wc -l` lists the number of lines in the files and `sort -n` sorts things numerically. We could put this in a file, but then it would only ever sort a list of `.pdb` files in the current directory. If we want to be able to get a sorted list of other kinds of files, we need a way to get all those names into the script. We can't use `$1`, `$2`, and so on because we don't know how many files there are. Instead, we use the special variable `$*`, which means, "All of the command-line parameters to the shell script." Here's an example:

```
$ cat sorted.sh
```

```
wc -l $* | sort -n
```

```
$ bash sorted.sh *.dat backup/*.dat
```

```
     29 chloratin.dat
     89 backup/chloratin.dat
     91 sphagnoi.dat
    156 sphag2.dat
    172 backup/sphag-merged.dat
    182 girmanis.dat
```

Why Isn't It Doing Anything?

What happens if a script is supposed to process a bunch of files, but we don't give it any filenames? For example, what if we type:

```
$ bash sorted.sh
```

but don't say `*.dat` (or anything else)? In this case, `$*` expands to nothing at all, so the pipeline inside the script is effectively:

```
wc -l | sort -n
```

Since it doesn't have any filenames, `wc` assumes it is supposed to process standard input, so it just sits there and waits for us to give it some data interactively. From the outside, though, all we see is it sitting there: the script doesn't appear to do anything.

We have two more things to do before we're finished with our simple shell scripts. If you look at a script like:

```
wc -l $* | sort -n
```

you can probably puzzle out what it does. On the other hand, if you look at this script:

```
# List files sorted by number of lines.
wc -l $* | sort -n
```

you don't have to puzzle it out—the comment at the top tells you what it does. A line or two of documentation like this make it much easier for other people (including your future self) to re-use your work. The only caveat is that each time you modify the script, you should check that the comment is still accurate: an explanation that sends the reader in the wrong direction is worse than none at all.

Second, suppose we have just run a series of commands that did something useful—for example, that created a graph we'd like to use in a paper. We'd like to be able to re-create the graph later if we need to, so we want to save the commands in a file. Instead of typing them in again (and potentially getting them wrong) we can do this:

```
$ history | tail -4 > redo-figure-3.sh
```

The file `redo-figure-3.sh` now contains:

```
297 goostats -r NENE01729B.txt stats-NENE01729B.txt
298 goodiff stats-NENE01729B.txt /data/validated/01729.txt > 01729-difference
s.txt
299 cut -d ',' -f 2-3 01729-differences.txt > 01729-time-series.txt
300 ygraph --format scatter --color bw --borders none 01729-time-series.txt f
igure-3.png
```

After a moment's work in an editor to remove the serial numbers on the commands, we have a completely accurate record of how we created that figure.

Unnumbering

Nelle could also use `colrm` (short for "column removal") to remove the serial numbers on her previous commands. Its parameters are the range of characters to strip from its input:

```
$ history | tail -5
  173  cd /tmp
  174  ls
  175  mkdir bakup
  176  mv bakup backup
  177  history | tail -5
$ history | tail -5 | colrm 1 7
cd /tmp
ls
mkdir bakup
mv bakup backup
history | tail -5
history | tail -5 | colrm 1 7
```

In practice, most people develop shell scripts by running commands at the shell prompt a few times to make sure they're doing the right thing, then saving them in a file for re-use. This style of work allows people to recycle what they discover about their data and their workflow with one call to `history` and a bit of editing to clean up the output and save it as a shell script.

Nelle's Pipeline: Creating a Script

An off-hand comment from her supervisor has made Nelle realize that she should have provided a couple of extra parameters to `goostats` when she processed her files. This might have been a disaster if she had done all the analysis by hand, but thanks to for loops, it will only take a couple of hours to re-do.

But experience has taught her that if something needs to be done twice, it will probably need to be done a third or fourth time as well. She runs the editor and writes the following:

```
# Calculate reduced stats for data files at J = 100 c/bp.
for datafile in $*
do
    echo $datafile
    goostats -J 100 -r $datafile stats-$datafile
done
```

(The parameters `-J 100` and `-r` are the ones her supervisor said she should have used.) She saves this in a file called `do-stats.sh` so that she can now re-do the first stage of her analysis by typing:

```
$ bash do-stats.sh *[AB].txt
```

She can also do this:

```
$ bash do-stats.sh *[AB].txt | wc -l
```

so that the output is just the number of files processed rather than the names of the files that were processed.

One thing to note about Nelle's script is that it lets the person running it decide what files to process. She could have written it as:

```
# Calculate reduced stats for  A and Site B data files at J = 100 c/bp.
for datafile in *[AB].txt
do
    echo $datafile
    goostats -J 100 -r $datafile stats-$datafile
done
```

The advantage is that this always selects the right files: she doesn't have to remember to exclude the 'Z' files. The disadvantage is that it *always* selects just those files—she can't run it on all files (including the 'Z' files), or on the 'G' or 'H' files her colleagues in Antarctica are producing, without editing the script. If she wanted to be more adventurous, she could modify her script to check for command-line parameters, and use `*[AB].txt` if none were provided. Of course, this introduces another tradeoff between flexibility and complexity.

Key Points

- Save commands in files (usually called shell scripts) for re-use.
- `bash filename` runs the commands saved in a file.
- `$*` refers to all of a shell script's command-line parameters.
- `$1`, `$2`, etc., refer to specified command-line parameters.
- Letting users decide what files to process is more flexible and more consistent with built-in Unix commands.

Leah has several hundred data files, each of which is formatted like this:

```
2013-11-05,deer,5
2013-11-05,rabbit,22
2013-11-05,raccoon,7
2013-11-06,rabbit,19
2013-11-06,deer,2
2013-11-06,fox,1
2013-11-07,rabbit,18
2013-11-07,bear,1
```

Write a shell script called `species.sh` that takes any number of filenames as command-line parameters, and uses `cut`, `sort`, and `uniq` to print a list of the unique species appearing in each of those files separately.

Write a shell script called `longest.sh` that takes the name of a directory and a filename extension as its parameters, and prints out the name of the file with the most lines in that directory with that extension. For example:

```
$ bash longest.sh /tmp/data pdb
```

would print the name of the `.pdb` file in `/tmp/data` that has the most lines.

If you run the command:

```
history | tail -5 > recent.sh
```

the last command in the file is the `history` command itself, i.e., the shell has added `history` to the command log before actually running it. In fact, the shell *always* adds commands to the log before running them. Why do you think it does this?

Joel's `data` directory contains three files: `fructose.dat`, `glucose.dat`, and `sucrose.dat`. Explain what a script called `example.sh` would do when run as `bash example.sh *.dat` if it contained the following lines:

```
# Script 1
echo *.*
```

```
# Script 2
for filename in $1 $2 $3
do
    cat $filename
done
```

```
# Script 3
echo $*.dat
```