Shell Scripts

Overview

Teaching: 30 min
Exercises: 15 min
Questions

- How can I save and re-use commands?

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 shell scripts you, and others, have written.

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**, but make no mistake: these are actually small programs.

Let's start by going back to `molecules/` and creating a new file, `middle.sh` which will become our shell script:

```
$ cd molecules
$ nano middle.sh
```

The command `nano middle.sh` opens the file `middle.sh` within the text editor "nano" (which runs within the shell). If the file does not exist, it will be created. We can use the text editor to directly edit the file – we'll simply insert the following line:

```
head -n 15 octane.pdb | tail -n 5
```

This is a variation on the pipe we constructed earlier: it selects lines 11-15 of the file `octane.pdb`. Remember, we are *not* running it as a command just yet: we are putting the commands in a file.

Then we save the file (`Ctrl-O` in nano), and exit the text editor (`Ctrl-X` in nano). Check that the directory `molecules` now contains a file called `middle.sh`.

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      9  H           1      -4.502   0.681   0.785  1.00  0.00
ATOM     10  H           1      -5.254  -0.243  -0.537  1.00  0.00
ATOM     11  H           1      -4.357   1.252  -0.895  1.00  0.00
ATOM     12  H           1      -3.009  -0.741  -1.467  1.00  0.00
ATOM     13  H           1      -3.172  -1.337   0.206  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 make it more versatile:

```
$ nano middle.sh
```

Now, within “nano”, replace the text `octane.pdb` with the special variable called `$1` :

```
head -n 15 "$1" | tail -n 5
```

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

```
$ bash middle.sh octane.pdb
```

```
ATOM      9  H           1      -4.502   0.681   0.785  1.00  0.00
ATOM     10  H           1      -5.254  -0.243  -0.537  1.00  0.00
ATOM     11  H           1      -4.357   1.252  -0.895  1.00  0.00
ATOM     12  H           1      -3.009  -0.741  -1.467  1.00  0.00
ATOM     13  H           1      -3.172  -1.337   0.206  1.00  0.00
```

or on a different file like this:

```
$ bash middle.sh pentane.pdb
```

```
ATOM      9  H           1       1.324   0.350  -1.332  1.00  0.00
ATOM     10  H           1       1.271   1.378   0.122  1.00  0.00
ATOM     11  H           1      -0.074  -0.384   1.288  1.00  0.00
ATOM     12  H           1      -0.048  -1.362  -0.205  1.00  0.00
ATOM     13  H           1      -1.183   0.500  -1.412  1.00  0.00
```

📌 Double-Quotes Around Arguments

For the same reason that we put the loop variable inside double-quotes, in case the filename happens to contain any spaces, we surround `$1` with double-quotes.

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` for the number of lines to be passed to `head` and `tail` respectively:

```
$ nano middle.sh
```

```
head -n "$2" "$1" | tail -n "$3"
```

We can now run:

```
$ bash middle.sh pentane.pdb 15 5
```

```
ATOM      9  H           1       1.324   0.350  -1.332  1.00  0.00
ATOM     10  H           1       1.271   1.378   0.122  1.00  0.00
ATOM     11  H           1      -0.074  -0.384   1.288  1.00  0.00
ATOM     12  H           1      -0.048  -1.362  -0.205  1.00  0.00
ATOM     13  H           1      -1.183   0.500  -1.412  1.00  0.00
```

By changing the arguments to our command we can change our script's behaviour:

```
$ bash middle.sh pentane.pdb 20 5
```

```
ATOM     14  H           1      -1.259   1.420   0.112  1.00  0.00
ATOM     15  H           1      -2.608  -0.407   1.130  1.00  0.00
ATOM     16  H           1      -2.540  -1.303  -0.404  1.00  0.00
ATOM     17  H           1      -3.393   0.254  -0.321  1.00  0.00
TER      18              1
```

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** at the top:

```
$ nano middle.sh
```

```
# Select lines from the middle of a file.
# Usage: bash middle.sh filename end_line num_lines
head -n "$2" "$1" | tail -n "$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 (including your future self) understand and use scripts. 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.

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 (recall that `wc` stands for 'word count', adding the `-l` option means 'count lines' instead) 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 arguments to the shell script." We also should put `$@` inside double-quotes to handle the case of arguments containing spaces (`"$@"` is equivalent to `"$1"  "$2"` …) Here's an example:

```
$ nano sorted.sh
```

```
# Sort filenames by their length.
# Usage: bash sorted.sh one_or_more_filenames
wc -l "$@" | sort -n
```

```
$ bash sorted.sh *.pdb ../creatures/*.dat
```

```
9 methane.pdb
12 ethane.pdb
15 propane.pdb
20 cubane.pdb
21 pentane.pdb
30 octane.pdb
163 ../creatures/basilisk.dat
163 ../creatures/unicorn.dat
433 total
```

List Unique Species

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
```

An example of this type of file is given in `data-shell/data/animal-counts/animals.txt` .

We can use the command `cut -d , -f 2 animals.txt | sort | uniq` to produce the unique species in `animals.txt` . In order to avoid having to type out this series of commands every time, a scientist may choose to write a shell script instead.

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

Solution

```
# Script to find unique species in csv files where species is the second data field
# This script accepts any number of file names as command line arguments

# Loop over all files
for file in $@
do
        echo "Unique species in $file:"
        # Extract species names
        cut -d , -f 2 $file | sort | uniq
done
```

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 -n 5 > redo-figure-3.sh
```

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

```
297 bash goostats NENE01729B.txt stats-NENE01729B.txt
298 bash goodiff stats-NENE01729B.txt /data/validated/01729.txt > 01729-differences.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 figure-3.png
301 history | tail -n 5 > redo-figure-3.sh
```

After a moment's work in an editor to remove the serial numbers on the commands, and to remove the final line where we called the `history` command, we have a completely accurate record of how we created that figure.

Why Record Commands in the History Before Running Them?

If you run the command:

```
$ history | tail -n 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?

Solution

If a command causes something to crash or hang, it might be useful to know what that command was, in order to investigate the problem. Were the command only be recorded after running it, we would not have a record of the last command run in the event of a crash.

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

Nelle's supervisor insisted that all her analytics must be reproducible. The easiest way to capture all the steps is in a script.

First we return to Nelle's data directory:

```
$ cd ../north-pacific-gyre/2012-07-03/
```

She runs the editor and writes the following:

```
# Calculate stats for data files.
for datafile in "$@"
do
    echo $datafile
    bash goostats $datafile stats-$datafile
done
```

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 NENE*[AB].txt
```

She can also do this:

```
$ bash do-stats.sh NENE*[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 stats for Site A and Site B data files.
for datafile in NENE*[AB].txt
do
    echo $datafile
    bash goostats $datafile stats-$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 arguments, and use `NENE*[AB].txt` if none were provided. Of course, this introduces another tradeoff between flexibility and complexity.

Variables in Shell Scripts

In the `molecules` directory, imagine you have a shell script called `script.sh` containing the following commands:

```
head -n $2 $1
tail -n $3 $1
```

While you are in the `molecules` directory, you type the following command:

```
bash script.sh '*.pdb' 1 1
```

Which of the following outputs would you expect to see?

1. All of the lines between the first and the last lines of each file ending in `.pdb` in the `molecules` directory
2. The first and the last line of each file ending in `.pdb` in the `molecules` directory
3. The first and the last line of each file in the `molecules` directory
4. An error because of the quotes around `*.pdb`

Solution

The correct answer is 2.

The special variables $1, $2 and $3 represent the command line arguments given to the script, such that the commands run are:

```
$ head -n 1 cubane.pdb ethane.pdb octane.pdb pentane.pdb propane.pdb
$ tail -n 1 cubane.pdb ethane.pdb octane.pdb pentane.pdb propane.pdb
```

The shell does not expand `'*.pdb'` because it is enclosed by quote marks. As such, the first argument to the script is `'*.pdb'` which gets expanded within the script by `head` and `tail`.

Find the Longest File With a Given Extension

Write a shell script called `longest.sh` that takes the name of a directory and a filename extension as its arguments, 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.

Solution

```
# Shell script which takes two arguments:
#    1. a directory name
#    2. a file extension
# and prints the name of the file in that directory
# with the most lines which matches the file extension.

wc -l $1/*.$2 | sort -n | tail -n 2 | head -n 1
```

Script Reading Comprehension

For this question, consider the `data-shell/molecules` directory once again. This contains a number of `.pdb` files in addition to any other files you may have created. Explain what each of the following three scripts would do when run as `bash script1.sh *.pdb`, `bash script2.sh *.pdb`, and `bash script3.sh *.pdb` respectively.

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

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

```
# Script 3
echo $@.pdb
```

Solutions

In each case, the shell expands the wildcard in `*.pdb` before passing the resulting list of file names as arguments to the script.

Script 1 would print out a list of all files containing a dot in their name. The arguments passed to the script are not actually used anywhere in the script.

Script 2 would print the contents of the first 3 files with a `.pdb` file extension. `$1`, `$2`, and `$3` refer to the first, second, and third argument respectively.

Script 3 would print all the arguments to the script (i.e. all the `.pdb` files), followed by `.pdb`. `$@` refers to *all* the arguments given to a shell script.

```
cubane.pdb ethane.pdb methane.pdb octane.pdb pentane.pdb propane.pdb.pdb
```

Debugging Scripts

Suppose you have saved the following script in a file called `do-errors.sh` in Nelle's `north-pacific-gyre/2012-07-03` directory:

```
# Calculate stats for data files.
for datafile in "$@"
do
    echo $datfile
    bash goostats $datafile stats-$datafile
done
```

When you run it:

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

the output is blank. To figure out why, re-run the script using the `-x` option:

```
bash -x do-errors.sh NENE*[AB].txt
```

What is the output showing you? Which line is responsible for the error?

Solution

The `-x` option causes `bash` to run in debug mode. This prints out each command as it is run, which will help you to locate errors. In this example, we can see that `echo` isn't printing anything. We have made a typo in the loop variable name, and the variable `datfile` doesn't exist, hence returning an empty string.

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 arguments.
- `$1` , `$2` , etc., refer to the first command-line argument, the second command-line argument, etc.
- Place variables in quotes if the values might have spaces in them.
- Letting users decide what files to process is more flexible and more consistent with built-in Unix commands.

(../05-loop/index.html)

(../07-find/i