title: The Unix Shell
subtitle: Shell Scripts
minutes: 20

The Unix Shell: Scripts

Learning 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 from shell's history.
- Explain the importance of comments in 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.

Writing a script

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 `articles` and creating a file called `group.sh`:

```
$ cd ~/programming-fundamentals/data/articles
$ touch group.sh
$ nano group.sh
```

```
cat africa*.txt | wc -l
```

This is a variation on command we created earlier: It concatenates all of the africa files into one large text file containing all africa articles.

Remember, we are *not* running it as a command just yet: we are putting the commands in a file.

Running a script

Once we have saved the file, we can ask the shell to execute the commands it contains.

First we have to tell the shell what program the script is in. If we want to run a Python script, we would enter in `python` first. If R, then `r` and so on.

Our shell is called `bash`, so we run the following command:

```
$ bash group.sh
```

```
136
```

Sure enough, our script's output is exactly what we would get if we ran that pipeline directly. It concatenated all of the africa files and then counted the number of lines, which is 136 (one line for each article).

> ## 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.

Variables in Scripts

What if we want concatenate an arbitrary group of files? We could edit `group.sh` each time to change the filename, but that would probably take longer than just retyping the command.

Instead, let's edit `group.sh` and replace `africa*.txt` with a special variable called "$@":

```
$ nano group.sh
```

```
cat "$@" | wc -l
```

Inside a shell script, $1 means "the first filename (or other parameter) on the command line". $2 means the second and so on. But in this case, 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 $@, which means, "All of the command-line parameters to the shell script." So ("$@" is equivalent to "$1" "$2" ...)

We put $@ inside double-quotes to handle the case of parameters containing spaces.

```
$ bash group.sh africa*.txt
```

```
136
```

or on a different file like this:

```
$ bash group.sh asia*.txt
```

```
145
```

Commenting

This works, but it may take the next person who reads `group.sh` a moment to figure out what it does. We can improve our script by adding some **comments** at the top:

```
$ cat group.sh
```

```
# Concatenates a group of files and returns the total number of lines.
# Usage: group.sh files
cat "$@" | wc -l
```

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.

Why Isn't It Doing Anything? {.callout}

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 group.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:

```
cat | wc -l
```

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.

Redirecting

We can use our script to redirect output files, too. Let's change the script to just:

```
$ cat group.sh
```

```
# Concatenates a group of files and returns the total number of lines.
# Usage: group.sh files
cat "$@"
```

Now we can use the > trick to output our result.

```
$ bash group.sh asia*.txt > all-asia.txt
$ cat all-asia.txt
```

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.

Scripts from History

Now, 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-
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
```

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 {.callout}

Rochelle 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
bash group.sh
bash group.sh asia*.txt > all-asia.txt
ls all-asia.txt
cat all-asia.txt
history | tail -5 | colrm 1 7
```

In order to get a range in your history, simply combine head and tail arguments, e.g.:

```
$ history | tail -15 | head -5 | colrm 1 7 > new_script.sh
```

would place the five commands from 15 commands before into a new_script.sh file without the serial numbers.

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.

Practical scripts

So what is a good example of a shell script for practical use? Look at regen_pdfs.sh:

```
cd programming-fundamentals
nano regen_pdfs.sh
```

This script is used for the materials you're reading now! While you may not have the program `markdown-pdf` installed, you can see how it's called in a bash script to batch convert a number of files. One of the true powers of bash is gluing together other programs, maybe written in different languages, to perform a sequence of events, especially events dealing with files.

Exercises

Challenge 1

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

```
> bash my_files/longest.sh /tmp/data pdb
```

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

Challenge 2

Joel's `data` directory contains three files: `fructose.dat`, `glucose.dat`, and `sucrose.dat`. Each of the `.dat` files contains only the word `sugar`. Explain what a script called `example.sh` (also in the directory!) would do when run as `bash example.sh *.dat` if it contained the following lines:

```
# Script a
echo *.*
```

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

```
# Script c
echo $@.dat
```

Now test your theory and redirect the output to `my_files/challenge_2a.txt`, `my_files/challenge_2b.txt`, and `my_files/challenge_2c.txt` respectively.

Challenge 3

What happens if you rename `example.sh` to `example.R`?

When you feel you have met these challenges successfully, cd into test/ and type

```
. 1-5_test.sh
```

Adapted from: [Software Carpentry (http://software-carpentry.org/v5/novice/shell/05-script.html)]