Unit 3: The bash shell and UNIX utilities

September 9, 2021

Note that it can be difficult to distinguish what is shell-specific and what is just part of the operating system (i.e., a UNIX-style operating system). Some of the material here is not bash-specific but general to UNIX. I'll use 'UNIX' to refer to a family of operating systems that descend from the path-breaking UNIX operating system developed at AT&T's Bell Labs in the 1970s. These include MacOS and various flavors of Linux (e.g., Ubuntu, Debian, CentOS, Fedora).

For your work on this unit, either bash on a Linux machine, the older version of bash on MacOS, or zsh on MacOS (or Linux) is fine. I'll probably demo everything using bash on a Linux machine, and there are some annoying differences from the older bash on MacOS that may be occasionally confusing.

Any tutorials mentioned below are available at
http://statistics.berkeley.edu/computing/training/tutorials.

Reference: Newham and Rosenblatt, Learning the bash Shell, 2nd ed.

1 Shell basics

The shell is the interface between you and the UNIX operating system. When you are working in a terminal window (i.e., a window with the command line interface), you're interacting with a shell.

There are multiple shells (*sh*, *bash*, *zsh, csh*, *tcsh*, *ksh*). We'll assume usage of *bash*, as this is a very commonly-used shell in Linux, plus it's the default for Mac OS X (until Catalina, for which *zsh* is the default), the SCF machines, and the UC Berkeley campus cluster (Savio).

UNIX shell commands are designed to each do a specific task really well and really fast. They are modular and composable, so you can build up complicated operations by combining the commands. These tools were designed decades ago, so using the shell might seem old-fashioned, but the shell still lies at the heart of modern scientific computing. By using the shell you can automate your work and make it reproducible.

2 Using the bash shell

Please see the tutorial on using the bash shell. For our purposes in this Unit, you should work through all of the material EXCEPT as follows:

- If you don't have access to a remote Unix-based machine (such as one of the SCF machines), then you wouldn't be able to practice with *ssh* and *scp* in Section 1.1. That's fine. Just read it over to get the main idea.
- You can skip Section 1.4.1.
- You can skip regular expressions (Section 3), but do look at Section 3.6.1 and the 'text substitution' use of *sed* in Section 3.6.2.
- Feel free to skip Section 4 for now; I'll ask you to read it more carefully later.

When we talk about string processing and regular expressions in R in Unit 5, we'll come back to the material on regular expressions.

When looking at Sections 3.6.1 and 3.6.2, note that you can use *grep* to look for and *sed* to replace simple strings in files without needing to know how to specify patterns based on regular expressions. So try to see how to do that without getting bogged down in the examples in those sections that use the complicated regular expression syntax. For example:

```
grep "," file.txt  # look for lines with commas in file.txt
sed -i 's/,/;/g' file.txt  # replace commas with semicolons in file.txt
```

3 bash shell examples

Here we'll work through a few examples to start to give you a feel for using the bash shell to manage your workflows and process data.

First let's get the files from the 243 class in 2020 so we have a sufficient body of files we can do interesting things with.

```
git clone https://github.com/berkeley-stat243/stat243-fall-2020
```

Our first mission is some basic manipulation of a data file. Suppose we want to get a sense for the number of weather stations in different states using the *coop.txt* file.

```
cd stat243-fall-2020/data
gunzip coop.txt.gz
cut -b50-70 coop.txt | less
cut -b60-61 coop.txt | sort | uniq
cut -b60-61 coop.txt | sort | uniq -c
```

Now I could of course read the data in R at this stage (or I could read the original dataset, though sometimes it's good to read just the fields of interest to reduce memory use).

Our second mission: how can I count the number of fields in a CSV file programmatically?

```
tail -n 1 cpds.csv | grep -o ',' | wc -l
nfields=$(tail -n 1 cpds.csv | grep -o ',' | wc -l)

nfields=$((${nfields}+1))
echo $nfields

## alternatively, we can use `bc`
nfields=$(echo "${nfields}+1" | bc)
```

Trouble-shooting: How could the syntax above get the wrong answer?

Extension: We could write a function that can count the number of fields in any file.

Extension: How could I see if all of the lines have the same number of fields?

Our third mission: was *example.pdf* created in the five most recently modified R code files in the units directory?

```
cd ../units
grep -l 'example.pdf' unit13-graphics.R
ls -tr *.R
## if unit13-graphics.R is not amongst the 5 most recently used,
## let's artificially change the timestamp so it is recently used.
touch unit13-graphics.R

ls -tr *.R | tail -n 5
ls -tr *.R | tail -n 5 | grep pdf
ls -tr *.R | tail -n 5 | grep "13-gr"
ls -tr *.R | tail -n 5 | xargs grep 'example.pdf'
ls -tr *.R | tail -n 5 | xargs grep -l 'example.pdf'
```

```
## here's how we could do it by explicitly passing the file names
## rather than using xargs
grep -l 'example.pdf' $(ls -tr *.R | tail -n 5)
```

Notice that `man tail` indicates it can take input from a FILE or from *stdin*. Here it uses *stdin*, so it is gives the last five lines of the output of *ls*, not the last five lines of the files indicated in that output.

`man grep` also indicates it can take input from a FILE or from *stdin*. However, we want grep to operate on the content of the files indicated in stdin. So we use *xargs* to convert *stdin* to be recognized as arguments, which then are the FILE inputs to *grep*.

Our fourth mission: write a function that will move the most recent *n* files in your Downloads directory to another directory.

In general, we want to start with a specific case, and then generalize to create the function.

```
ls -rt ~/Downloads | tail -n 1

## sometimes the ~ behaves weirdly in scripting, so let's use full path
mv "/accounts/gen/vis/paciorek/Downloads/$(ls -rt \
   /accounts/gen/vis/paciorek/Downloads | tail -n 1)" ~/Desktop

function mvlast() {
    mv "/accounts/gen/vis/paciorek/Downloads/$(ls -rt \
       /accounts/gen/vis/paciorek/Downloads | tail -n $2)" $1
}
```

That code only works if the files don't have spaces in their names. If there are spaces, we need double quotes around each file name, which is hard to do in the shell because double quotes are interpreted by the shell as giving the beginning and ending of strings, rather than being passed along for further processing.

Our fifth mission: automate the process of determining what R packages are used in all of the R code here and install those packages on a new machine.

```
grep library unit[1-9]*.R
grep --no-filename library *.R
grep --no-filename "^library" *.R
grep --no-filename "^library" *.R | sort | uniq
grep --no-filename "^library" *.R | sort | uniq | cut -d'#' -f1
```

```
grep --no-filename "^library" *.R | sort | uniq | cut -d'#' -f1 | \
    tee libs.txt
grep -v "help =" libs.txt > tmp2.txt
sed 's/;/\n/g' tmp2.txt | sed 's/ //g' |
    sed 's/library(//' | sed 's/)//g' > libs.txt
## note: on a Mac, use 's/;/\\\n/g'   -- see https://superuser.com/questions
echo "There are $(wc -l libs.txt | cut -d' ' -f1) \
unique packages we will install."
## note: on Linux, wc -l puts the number as the first characters of the outp
## on a Mac, there may be a bunch of spaces preceding the number, so try thi
## echo "There are $(wc -l libs.txt | tr -s ' ' | cut -d' ' -f2) \
## unique packages we will install."
```

```
Rscript -e "pkgs <- scan('libs.txt', what = 'character'); \
install.packages(pkgs, repos = 'https://cran.r-project.org')"
```

Our sixth mission: suppose I've accidentally started a bunch of jobs (perhaps with a for loop in bash!) and need to kill them.

```
echo "Sys.sleep(1e5)" > job.R
nJobs=30
for (( i=1; i<=${nJobs}; i++ )); do
   R CMD BATCH --no-save job.R job-${i}.out &
done
```

```
ps -o pid,pcpu,pmem,user,cmd -C R
ps -o pid,pcpu,pmem,user,cmd,start_time --sort=start_time -C R | tail -n 30
ps -o pid --sort=start_time -C R | tail -n ${nJobs} | xargs kill

# on a Mac:
ps -o pid,pcpu,pmem,user,command | grep exec/R
# not clear how to sort by start time
ps -o pid,command | grep exec/R | cut -d' ' -f1 |  tail -n ${nJobs} | xargs
```

4 bash shell challenges

4.1 First challenge

Consider the file *unit3-bash.sh*. How would you write a shell command that returns "There are 2 occurrences of the word 'bash' in this file."

Extra: make your code into a function that can operate on any file indicated by the user and any word of interest.

4.2 Second challenge

1. For Belgium, determine the minimum unemployment value (field #6) in *cpds.csv* in a programmatic way.

2. Have what is printed out to the screen look like "Belgium 6.2".

3. Now store the unique values of the countries in a variable, first stripping out the quotation marks and removing the space in "New Zealand", which causes problems because of the space in its name.

4. Figure out how to automate step 1 to do the calculation for all the countries and print to the screen.

5. How would you instead store the results in a new file?

4.3 Third challenge

Consider the data in the *RTADataSub.csv* file. This is a subset of data giving freeway travel times for segments of a freeway in an Australian city. The data are from a kaggle.com competition. We want to try to understand the kinds of data in each field of the file. The following would be particularly useful if the data were in many files or the data were many gigabytes in size.

First, take the fourth column. Figure out the unique values in that column.

Next, automate the process of determining if any of the values are non-numeric so that you don't have to scan through all of the unique values looking for non-numbers. You'll need to look for the following regular expression pattern "[^0-9]", which is interpreted as NOT any of the numbers 0 through 9.

Now, do it for all the fields, except the first one. Have your code print out the result in a human-readable way understandable by someone who didn't write the code.

4.4 Fourth challenge

Here's an advanced one - you'll probably need to use *sed*, but the brief examples of text substitution in the using bash tutorial should be sufficient to solve the problem.

Consider a CSV file that has rows that look like this:

```
1,"America, United States of",45,96.1,"continental, coastal"
2,"France",33,807.1,"continental, coastal"
```

While R would be able to handle this using *read.table()*, using *cut* in UNIX won't work because of the commas embedded within the fields. The challenge is to convert this file to one that we can use *cut* on, as follows.

Figure out a way to make this into a new delimited file in which the delimiter is not a comma. At least one solution that will work for this particular two-line dataset does not require you to use regular expressions, just simple replacement of fixed patterns.

5 Regular expressions

Some of this material is duplicated from Section 3 of the String Processing tutorial.

5.1 Overview

Regular expressions are a domain-specific language for finding patterns and are one of the key functionalities in scripting languages such as Python and R, as well as the UNIX commands *grep*, *sed*, and *awk*.

The basic idea of regular expressions is that they allow us to find matches of strings or patterns in strings, as well as do substitution. Regular expressions are good for tasks such as:

- extracting pieces of text from documents;
- creating variables from information found in text;
- cleaning and transforming text into a uniform format;
- mining text by treating documents as data; and
- scraping the web for data.

That said, if we can avoid using regular expressions, it's generally a good idea to use more specialized code that understands the structure of particular formats. For example, recall our case of

using R functions that treat HTML or XML or JSON in a structured way based on the exact syntax of HTML/XML/JSON. Doing that work using regular expressions would have been more difficult and error-prone.

See Section 3 of the *Using the bash shell* tutorial for details on regular expression syntax. For other resources, Duncan Temple Lang (UC Davis Statistics) has written a nice tutorial that is part of the string processing tutorial repository or check out Sections 9.9 and 11 of Paul Murrell's book.

Also, here's a cheatsheet on regular expressions (see the second page) and here is a website where you can interactively test regular expressions on example strings.

5.2 Versions of regular expressions

One thing that can cause headaches is differences in version of regular expression syntax used. As discussed the grep man page, *extended regular expressions* are standard, with *basic regular expressions* providing somewhat less functionality and *Perl regular expressions* additional functionality. In R, as can be seen in `help(regex)`, *stringr* provides *ICU regular expressions*, which are based on Perl regular expressions. More details can be found in the regex Wikipedia page.

The tutorial on using bash provides a full documentation of the various *extended regular expressions* syntax, which we'll focus on here. This should be sufficient for most usage and should be usable in R and Python, but if you notice something funny going on, it might be due to differences between the regular expressions versions.

5.3 General principles for working with regex

The syntax is very concise, so it's helpful to break down individual regular expressions into the component parts to understand them. As Murrell notes, since regex are their own language, it's a good idea to build up a regex in pieces as a way of avoiding errors just as we would with any computer code. *str_detect* in R's *stringr* and *re.findall* in Python are particularly useful in seeing **what** was matched to help in understanding and learning regular expression syntax and debugging your regex. As with many kinds of coding, I find that debugging my regex is usually what takes most of my time.

5.4 Practice problems

Write a regular expression that matches the following:

1. Only the strings "cat", "at", and "t".

2. The strings "cat", "caat", "caaat", etc.

3. “dog”, “Dog”, “dOg”, “doG”, “DOg”, etc. (the word dog in any combination of lower and upper case).

4. Any line with exactly two words separated by any amount of whitespace (spaces or tabs). There may or may not be whitespace at the beginning or end of the line.

5. Any positive number with or without a decimal point.