# Unit 2: The bash shell, UNIX utilities, and version control

September 3, 2015

Note that it can be difficult to distinguish what is shell-specific and what is just part of UNIX. Some of the material here is not bash-specific but general to UNIX.

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*, *csh*, *tcsh*, *ksh*). We'll assume usage of *bash*, as this is the default for Mac OS X, the BCE VM, and on the SCF machines and is very common for Linux.

## 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 the following sections:

- Command history and editing (Section 1.3.2)
- Regular expressions (Section 3)

However, the material in those sections may be useful to you, so if you have a chance, you may want to work through them. When we talk about string processing and regular expressions in R in Unit 4, we'll come back to the material on regular expressions.

# 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 last year's 243 class so we have a sufficient body of files we can do interesting things with.

```
git clone https://github.com/berkeley-stat243/stat243-fall-2014
```

**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-2014/data
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 = 'foo'
```

**Our third mission**: was *example.pdf* created in the five most recently modified R code files in the units directory? Note that here grep is expecting a file name as input so we can just use the pipe without the xargs.

```
cd ../units
ls -tr *.R | tail -n 5 | grep pdf
ls -tr *.R | tail -n 5 | grep numbers
ls -tr *.R | tail -n 5 | xargs grep 'example.pdf'
ls -tr *.R | tail -n 5 | xargs grep -l 'example.pdf'
```

Notice that `man tail` indicates it can take input from a FILE or from *stdin*. But if we want to pass the filenames to grep we need to ensure that grep realizes the input is the file names and

not stdin itself. Otherwise we are just grepping for 'example.pdf' in the literal strings that are the file name.

**Our fourth 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
tmp.txt
grep -v "help =" tmp.txt > libs.txt
sed -e 's/library(//' libs.txt | sed -e 's/quietly = TRUE//g' | sed
-e 's/;/\n/g'
sed -e 's/;/\n/g' libs.txt | sed -e 's/quietly = TRUE//g' | sed -e
's/library(//' > tmp.txt
sed -e 's/[\, )]//g' tmp.txt > libs.txt
echo "There are $(wc -l libs.txt | cut -d' ' -f1) unique packages we
will install."
```

```
Rscript -e "pkgs <- scan('libs.txt', what = 'character'); \
install.packages(pkgs, repos = 'http:/cran.cnr.berkeley.edu')"
```

**Our fifth 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,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
```

# 4 bash shell challenges

## 4.1 First challenge

Consider the listing of info in */proc/cpuinfo* on a Linux machine such as BCE. How would you write a shell command that returns "There are 6 processors on this machine" where 6 is determined based on the contents of */proc/cpuinfo* and not based on using *nproc*.

Note: before trying this out, please set up BCE so that it uses two virtual processors. Go to `Settings -> System -> Processor` and choose "2 CPU". Then start up the VM.

## 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 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.

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 Version Control

A very popular and powerful tool for version control is Git. More information is available in the tutorial on the basics of using Git. You'll have a chance to work through the tutorial and practice with Git in the first section/lab. During the course, you'll make use of Git for submitting problem sets and for doing the final project. I recommend you practice using it more generally while preparing your problem set solutions.

Some links to more resources:

- Git for Scientists: A Tutorial: http://nyuccl.org/pages/GitTutorial/
- Gitwash: workflow for scientific Python projects: http://matthew-brett.github.io/pydagogue/gitwash_build.html
- Git branching demo: http://pcottle.github.io/learnGitBranching/