The bash shell and UNIX utilities

Chris Paciorek

2023-07-26

Table of contents

PDF

Reference:

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

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 the 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) are 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 (in particular the options to various commands can differ on MacOS).

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 providing 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). It's fine for you to use zsh.

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. And once you know how to use it, you'll find that enter commands quickly and without a lot of typing.

2. Using the bash shell

For this Unit, we'll rely on the bash shell tutorial for the details of how to use the shell. We won't cover the page on Managing Processes. For the moment, we won't cover the page on Regular Expressions, but when we talk about string processing and regular expressions in Unit 5, we'll come back to that material.

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 2021 so we have a sufficient body of files we can do interesting things with.

```
git clone https://github.com/berkeley-stat243/stat243-fall-2021
```

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-2021/data
gzip -cd coop.txt.gz | less
gunzip coop.txt.gz
cut -b50-70 coop.txt | less
cut -b60-61 coop.txt | uniq
cut -b60-61 coop.txt | sort | uniq
cut -b60-61 coop.txt | sort | uniq -c
## all in one line with no change to the original file:
gzip -cd coop.txt.gz | cut -b60-61 coop.txt | sort | uniq -c
```

I could have done that in R or Python, but it would have required starting the program up and reading all the data into memory.

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 by code 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*.

Here are some of the ways we can pass information from a command to somewhere else:

- piping allows us to pass information from one command to another command via stdout to stdin
- `$()` allows us to store the result of a command in a variable
  - also used to create a temporary variable to pass the output from one command as an option or argument (e.g., the FILE argument) of another command

- file redirection operators such as > and >> allow us to pass information from a command into a file

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/vis/paciorek/Downloads/$(ls -rt \
   /accounts/vis/paciorek/Downloads | tail -n 1)" ~/Desktop

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

Note the need for the quotes to deal with cases where a file has a space in its name.

If we wanted to handle multiple files, we could do it with a loop:

```
function mvlast() {
    for ((i=1; i<=${1}; i++)); do
        mv "/accounts/vis/paciorek/Downloads/$(ls -rt \
           /accounts/vis/paciorek/Downloads | tail -n 1)" ${2}
    done
}
```

Side note: if we were just moving files from the current working directory and with files without spaces in their names, it should be possible to use `tail -n ${1}` without the loop.

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/307165/newlines-in-se
```

```
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 output
## on a Mac, there may be a bunch of spaces preceding the number, so try this:
## echo "There are $(wc -l libs.txt | tr -s ' ' | cut -d' ' -f2) \
## unique packages we will install."


## @knitr mission5a                -
Rscript -e "pkgs <- scan('libs.txt', what = 'character'); \
install.packages(pkgs, repos = 'https://cran.r-project.org')"
```

You wouldn't want to use this code to accomplish this task in reality - there are packages such as *renv* and *packrat* that can accomplish this. The main point was to illustrate how one can quickly hack together some code to do fairly complicated tasks.

Our sixth mission: suppose I've accidentally started a bunch of jobs (perhaps with a for loop in bash!) and need to kill them. (This example uses syntax from the Managing Processes page of the bash tutorial, so it goes beyond what you were asked to read for this Unit.)

```
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

# on Linux:
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 kill
```

4. bash shell challenges

4.1 First challenge

Consider the file *cpds.csv*. How would you write a shell command that returns "There are 8 occurrences of the word 'Belgium' in this file.", where '8' is actually the correct number of times the word occurs.

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

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.

1. First, take the fourth column. Figure out the unique values in that column.
2. 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.
3. 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.

4.3 Third 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.
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?

4.4 Fourth challenge

Let's return to the `RTADataSub.csv` file and the issue of missing values.

1. Create a new file without any rows that have an 'x' (which indicate a missing value).
2. Turn the code into a function that also prints out the number of rows that are being removed and that sends its output to stdout so that it can be used with piping.
3. Now modify your function so that the user could provide the missing value string, the input filename and the output filename as arguments.

4.5 Fifth 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.