The bash shell, UNIX utilities, and version control

August 29, 2013

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.

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 OSX and on the SCF machines and is very common for Linux.

1. What shell am I using?
   ```
   > echo $SHELL
   ```
2. To change to bash on a one-time basis:
   ```
   > bash
   ```
3. To make it your default:
   ```
   > chsh /bin/bash
   ```
   /bin/bash should be whatever the path to the bash shell is, which you can figure out using **`which bash`**

Shell commands can be saved in a file (with extension *.sh*) and this file can be executed as if it were a program. To run a shell script called *file.sh*, you would type **`./file.sh`**. Note that if you just typed **`file.sh`**, the operating system will generally have trouble finding the script and recognizing that it is executable. To be sure that the operating system knows what shell to use to interpret the script, the first line of the script should be `#!/bin/bash` (in the case that you're using the bash shell).

2 Tab completion

When working in the shell, it is often unnecessary to type out an entire command or file name, because of a feature known as tab completion. When you are entering a command or filename in the shell, you can, at any time, hit the tab key, and the shell will try to figure out how to complete the name of the command or filename you are typing. If there is only one command in the search path and you're using tab completion with the first token of a line, then the shell will display its value and the cursor will be one space past the completed name. If there are multiple commands that match the partial name, the shell will display as much as it can. In this case, hitting tab twice will display a list of choices, and redisplay the partial command line for further editing. Similar behavior with regard to filenames occurs when tab completion is used on anything other than the first token of a command.

Note that R does tab completion for objects (including functions) and filenames.

3 Command history

By using the up and down arrows, you can scroll through commands that you have entered previously. So if you want to rerun the same command, or fix a typo in a command you entered, just scroll up to it and hit enter to run it or edit the line and then hit enter.

Note that you can use emacs-like control sequences (**`C-a`**, **`C-e`**, **`C-k`**) to navigate and delete characters, just as you can at the prompt in the shell usually.

You can also rerun previous commands as follows:

```
> !-n # runs the nth previous command
> !xt # runs the last command that started with 'xt'
```

If you're not sure what command you're going to recall, you can append **`:p`** at the end of the text you type to do the recall, and the result will be printed, but not executed. For example:

```
> !xt:p
```

You can then use the up arrow key to bring back that statement for editing or execution.

You can also search for commands by doing **`C-r`** and typing a string of characters to search for in the search history. You can hit return to submit, **`C-c`** to get out, or **`ESC`** to put the result on the regular command line for editing.

4 Wildcards in filenames

The shell will expand certain special characters to match patterns of file names, before passing those filenames on to a program. Note that the programs themselves don't know anything about

Table 1. Wildcards.

Syntax	What it matches
?	any single character
***	zero or more characters
[$c_1c_2\ldots$]	any character in the set
[!$c_1c_2\ldots$]	anything not in the set
[$c_1 - c_2$]	anything in the range from c_1 to c_2
{string1,string2,...}	anything in the set of strings

wildcards; it is the shell that does the expansion, so that programs don't see the wildcards. Table 1 shows some of the special characters that the shell uses for expansion:

Here are some examples of using wildcards:

- List all files ending with a digit:

```
> ls *[0-9]
```

- Make a copy of *filename* as *filename.old*

```
> cp filename{,.old}
```

- Remove all files beginning with *a* or *z*:

```
> rm [az]*
```

- List all the R code files with a variety of suffixes:

```
> ls *.{r,q,R}
```

The *echo* command can be used to verify that a wildcard expansion will do what you think it will:

```
> echo cp filename{,.old} # returns cp filename filename.old
```

If you want to suppress the special meaning of a wildcard in a shell command, precede it with a backslash (\). Note that this is a general rule of thumb in many similar situations when a character has a special meaning but you just want to treat it as a character.

5 Basic UNIX utilities

Table 2 shows some basic UNIX programs, which are sometimes referred to as filters. The general syntax for a UNIX program is

```
> command -options argument1 argument2 ...
```

For example, **`> grep -i graphics file.txt`** looks for *graphics* (argument 1) in *file.txt* (argument2) with the option *-i*, which says to ignore the case of the letters. **`> less file.txt`**

Table 2. UNIX utilities.

Name	What it does
tail	shows last few lines of a file
less	shows a file one screen at a time
cat	writes file to screen
wc	counts words and lines in a file
grep	finds patterns in files
wget or curl	download files from the web
sort	sorts a file by line
nl	numbers lines in a file
diff	compares two files
uniq	removes repeated (sequential) rows
cut	extracts fields (columns) from a file

simply pages through a text file (you can navigate up and down) so you can get a feel for what's in it.

UNIX programs often take options that are identified with a minus followed by a letter, followed by the specific option (adding a space before the specific option is fine). Options may also involve two dashes, e.g., **`R --no-save`**. Here's another example that tells *tail* to keep refreshing as the file changes:

```
tail -f dat.txt
```

A few more tidbits about *grep*:

```
grep ^read code.r # returns lines that start with 'read'
grep dat$ code.r # returns lines that end with 'dat'
grep 7.7 dat.txt # returns lines with two sevens separated by a single character
grep 7.*7 dat.txt # returns lines with two sevens separated by any number of characters
```

If you have a big data file and need to subset it by line (e.g., with *grep*) or by field (e.g., with *cut*), then you can do it really fast from the UNIX command line, rather than reading it with R, SAS, Perl, etc.

Much of the power of these utilities comes in piping between them (see Section 6) and using wildcards (see Section 4) to operate on groups of files. The utilities can also be used in shell scripts to do more complicated things.

Table 3. Redirection.

Syntax	What it does
`cmd > file`	sends stdout from *cmd* into *file*, overwriting *file*
`cmd >> file`	appends stdout from *cmd* to *file*
`cmd < file`	execute *cmd* reading stdin from *file*
`cmd <infile >outfile 2>errors`	reads from *infile*, sending stdout to *outfile* and stderr to *errors*
cmd <infile >outfile 2>&1	reads from *infile*, sending stdout and stderr to *outfile*
`cmd1 | cmd2`	sends stdout from *cmd1* as stdin to *cmd2* (a pipe)

Note that *cmd* may include options and arguments as seen in the previous section.

6 Redirection

UNIX programs that involve input and/or output often operate by reading input from a stream known as standard input (*stdin*), and writing their results to a stream known as standard output (*stdout*). In addition, a third stream known as standard error (*stderr*) receives error messages, and other information that's not part of the program's results. In the usual interactive session, standard output and standard error default to your screen, and standard input defaults to your keyboard. You can change the place from which programs read and write through redirection. The shell provides this service, not the individual programs, so redirection will work for all programs. Table 3 shows some examples of redirection.

Operations where output from one command is used as input to another command (via the | operator) are known as pipes; they are made especially useful by the convention that many UNIX commands will accept their input through the standard input stream when no file name is provided to them.

Here's an example of finding out how many unique entries there are in the 2rd column of a data file whose fields are separated by commas:

```
cut -d',' -f2 mileage2009.csv | sort | uniq | wc
```

To see if there are any "S" values in certain fields (fixed width) of a set of files (note I did this on 22,000 files (5 Gb or so) in about 15 minutes on my old desktop; it would have taken hours to read the data into R):

```
> cut -b29,37,45,53,61,69,77,85,93,101,109,117,125,133,141,149,
157,165,173,181,189,197,205,213,221,229,237,245,253,261,269 USC*.dly
| grep "S" | less
```

A closely related, but subtly different, capability is offered by the use of backticks (‘). When the shell encounters a command surrounded by backticks, it runs the command and replaces the backticked expression with the output from the command; this allows something similar to a pipe, but is appropriate when a command reads its arguments directly from the command line instead of through standard input. For example, suppose we are interested in searching for the text *pdf* in the

last 4 R code files (those with suffix .r or .R) that were modified in the current directory. We can find the names of the last 4 files ending in ".R" or ".r" which were modified using

```
> ls -t *.{R,r} | head -4
```

and we can search for the required pattern using *grep*. Putting these together with the backtick operator we can solve the problem using

```
> grep pdf `ls -t *.{R,r} | head -4`
```

Note that piping the output of the *ls* command into *grep* would not achieve the desired goal, since *grep* reads its filenames from the command line, not standard input.

You can also redirect output as the arguments to another program using the *xargs* utility. Here's an example:

```
> which bash | xargs chsh
```

And you can redirect output into a shell variable (see section 9) by putting the command that produces the output in parentheses and preceding with a *$*. Here's an example:

```
> files=$(ls) # NOTE - don't put any spaces around the '='
> echo $files
```

7 Job Control

Starting a job When you run a command in a shell by simply typing its name, you are said to be running in the foreground. When a job is running in the foreground, you can't type additional commands into that shell, but there are two signals that can be sent to the running job through the keyboard. To interrupt a program running in the foreground, use **C-c**; to quit a program, use **C-**. While modern windowed systems have lessened the inconvenience of tying up a shell with foreground processes, there are some situations where running in the foreground is not adequate.

The primary need for an alternative to foreground processing arises when you wish to have jobs continue to run after you log off the computer. In cases like this you can run a program in the background by simply terminating the command with an ampersand (*&*). However, before putting a job in the background, you should consider how you will access its results, since *stdout* is not preserved when you log off from the computer. Thus, redirection (including redirection of *stderr*) is essential when running jobs in the background. As a simple example, suppose that you wish to run an R script, and you don't want it to terminate when you log off. (Note that this can also be done using **R CMD BATCH**, so this is primarily an illustration.)

```
> R --no-save <code.R >code.Rout 2>&1 &
```

If you forget to put a job in the background when you first execute it, you can do it while it's running in the foreground in two steps. First, suspend the job using the **C-z** signal. After receiving the signal, the program will interrupt execution, but will still have access to all files and other resources.

Next, issue the `bg` command, which will put the stopped job in the background.

Listing and killing jobs Since only foreground jobs will accept signals through the keyboard, if you want to terminate a background job you must first determine the unique process id (PID) for the process you wish to terminate through the use of the *ps* command. For example, to see all the jobs running on a particular computer, you could use a command like:

```
> ps -aux
```

Among the output after the header (shown here) might appear a line that looks like this:

```
USER PID %CPU %MEM VSZ RSS TTY STAT START TIME COMMAND
paciorek 11998 97.0 39.1 1416644 1204824 pts/16 R+ Jul27 1330:01
/usr/lib64/R/bin/exec/R
```

In this example, the *ps* output tells us that this R job has a PID of *11998*, that it has been running for 1330 minutes (!), is using 97% of CPU and 39% of memory, and that it started on July 27. You could then issue the command:

```
> kill 11998
```

or, if that doesn't work

```
> kill -9 11998
```

to terminate the job. Another useful command in this regard is *killall*, which accepts a program name instead of a process id, and will kill all instances of the named program. E.g.,

```
> killall R
```

Of course, it will only kill the jobs that belong to you, so it will not affect the jobs of other users. Note that the *ps* and *kill* commands only apply to the particular computer on which they are executed, not to the entire computer network. Thus, if you start a job on one machine, you must log back into that same machine in order to manage your job.

Monitoring jobs and memory use The *top* command also allows you to monitor the jobs on the system and in real-time. In particular, it's useful for seeing how much of the CPU and how much memory is being used, as well as figuring out a PID as an alternative to *ps*. You can also renice jobs (see below) and kill jobs from within top: just type *r* or *k*, respectively, and proceed from there.

One of the main things to watch out for is a job that is using close to 100% of memory and much less than 100% of CPU. What is generally happening is that your program has run out of memory and is using virtual memory on disk, spending most of its time writing to/from disk, sometimes called *paging* or *swapping*. If this happens, it can be a very long time, if ever, before your job finishes.

Nicing a job The most important thing to remember when starting a job on a machine that is not your personal machine is how to be a good citizen. This often involves 'nicing' your jobs. This is required on the SCF machines, but the compute servers should automatically nice your jobs. Nicing a job puts it at a lower priority so that a user working at the keyboard has higher priority in using the CPU. Here's how to do it, giving the job a low priority of 19, as required by SCF:

```
> nice -19 R CMD BATCH --no-save in.R out.Rout &
```

If you forget and just submit the job without nicing, you can reduce the priority by doing:

```
> renice +19 11998
```

where *11998* is the PID of your job.

On many larger UNIX cluster computers, all jobs are submitted via a job scheduler and enter a queue, which handles the issue of prioritization and jobs conflicting. Syntax varies by system and queueing software, but may look something like this for submitting an R job:

```
> bsub -q long R CMD BATCH --no-save in.R out.Rout # just an example;
this will not work on the SCF network
```

8 Aliases

Aliases allow you to use an abbreviation for a command, to create new functionality or to insure that certain options are always used when you call an existing command. For example, I'm lazy and would rather type **`q`** instead of **`exit`** to terminate a shell window. You could create the alias as follow

```
> alias q="exit"
```

As another example, suppose you find the *-F* option of *ls* (which displays **/** after directories, ***** after executable files and @ after links) to be very useful. The command

```
> alias ls="ls -F"
```

will insure that the *-F* option will be used whenever you use *ls*. If you need to use the unaliased version of something for which you've created an alias, precede the name with a backslash (\). For example, to use the normal version of *ls* after you've created the alias described above, just type

```
> \ls
```

The real power of aliases is only achieved when they are automatically set up whenever you log in to the computer or open a new shell window. To achieve that goal with aliases (or any other bash shell commands), simply insert the commands in the file *.bashrc* in your home directory. See the *example.bashrc* file in the repository for some of what's in my *.bashrc* file.

9 Shell Variables

We can define shell variables that will help us when writing shell scripts. Here's an example of defining a variable:

```
> name="chris"
```

The shell may not like it if you leave any spaces around the = sign. To see the value of a variable we need to precede it by *$*:

```
> echo $chris
```

You can also enclose the variable name in curly brackets, which comes in handy when we're embedding a variable within a line of code to make sure the shell knows where the variable name ends:

```
> echo ${chris}
```

There are also special shell variables called environment variables that help to control the shell's behavior. These are generally named in all caps. Type **`env`** to see them. You can create your own environment variable as follows:

```
> export NAME="chris"
```

The *export* command ensures that other shells created by the current shell (for example, to run a program) will inherit the variable. Without the export command, any shell variables that are set will only be modified within the current shell. More generally, if one wants a variable to always be accessible, one would include the definition of a variable with an *export* command in your *.bashrc* file.

Here's an example of modifying an environment variable:

```
> export CDPATH=.:~/research:~/teaching
```

Now if you have a subdirectory *bootstrap* in your *research* directory, you can type **`cd bootstrap`** no matter what your pwd is and it will move you to *~/research/bootstrap*. Similarly for any subdirectory within the *teaching* directory.

Here's another example of an environment variable that puts the username, hostname, and pwd in your prompt. This is handy so you know what machine you're on and where in the filesystem you are.

```
> export PS1="\u@\h:\w> "
```

For me, this is one of the most important things to put in my *.bashrc* file. The **** syntax tells bash what to put in the prompt string: *u* for username, *h* for hostname, and *w* for working directory.

10 Functions

You can define your own utilities by creating a shell function. This allows you to automate things that are more complicated than you can do with an alias. One nice thing about shell functions is that the shell automatically takes care of function arguments for you. It places the arguments given by the user into local variables in the function called (in order): *$1 $2 $3* etc. It also fills *$#* with the number of arguments given by the user. Here's an example of using arguments in a function that saves me some typing when I want to copy a file to the SCF filesystem:

```
function putscf() {
   scp $1 paciorek@bilbo.berkeley.edu:~/$2
}
```

To use this function, I just do the following to copy *unit1.pdf* from the current directory on whatever non-SCF machine I'm on to the directory *~/teaching/243* on SCF:

```
> putscf unit1.pdf teaching/243/.
```

Of course you'd want to put such functions in your *.bashrc* file.

11 If/then/else

We can use if-then-else type syntax to control the flow of a shell script. For an example, see *niceR()* in the demo code file *niceR.sh* for this unit.

For more details, look in Newham&Rosenblatt or search online.

12 For loops

for loops in shell scripting are primarily designed for iterating through a set of files or directories. Here's an example:

```
for file in $(ls *txt)
do
   mv $file ${file/.txt/.R}
   # this syntax replaces .txt with .R in $file
done
```

You could also have done that with `for file in 'ls *txt'`

Another use of *for* loops is automating file downloads: see the demo code file. And, in my experience, *for* loops are very useful for starting a series of jobs: see the demo code files in the repository: *forloopDownload.sh* and *forloopJobs.sh*.

13 How much shell scripting should I learn?

You can do a fair amount of what you need from within R using the *system()* function. This will enable you to avoid dealing with a lot of shell programming syntax (but you'll still need to know how to use UNIX utilities, wildcards, and pipes to be effective). Example: a fellow student in grad school programmed a tool in R to extract concert information from the web for bands appearing in her iTunes library. Not the most elegant solution, but it got the job done.

For more extensive shell programming, it's probably worth learning Python and doing it there rather than using a shell script. In particular iPython makes it very easy to interact with the operating system.

14 Version Control using Git

[UNDER CONSTRUCTION]