The bash shell, UNIX utilities, and version control

September 1, 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

At a basic level, a simple principle is to have version numbers for all your work: code, datasets, manuscripts. Whenever you make a change to a dataset, increment the version number. For code and manuscripts, increment when you make substantial changes or have obvious breakpoints in your workflow.

The basic idea of version control software (VCS) is that instead of manually trying to keep track of what changes you've made to code, data, and documents, you use software to help you manage the process. This has several benefits:

- easily allowing you to go back to earlier versions
- allowing you to have multiple version you can switch between
- allowing you to share work easily without worrying about conflicts
- providing built-in backup

The material that follows is borrowed from Jarrod Millman and Fernando Perez.

14.1 VCS Overview

There are a number of version control systems including CVS and subversion, which use client-server models. Git is a distributed version control system. VCS store your material in a *repository*.

The next couple figures show graphical representations of how a repository is structured.

![Credit: ProGit book, by Scott Chacon, CC License.](figs/threecommits.png)

Recall that we have been using Git in a very simple fashion. We've cloned my class repository onto our local machines and have updated materials from that repository.

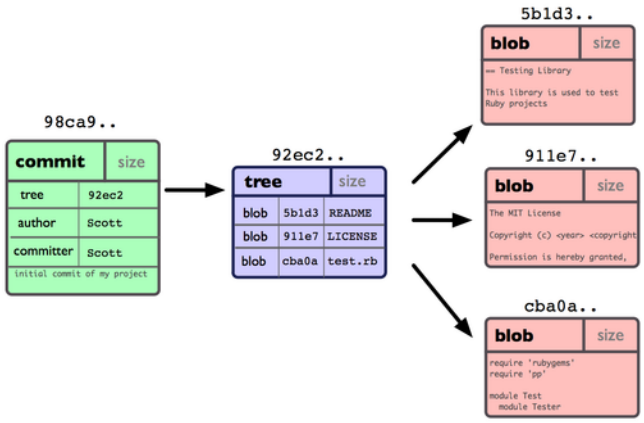

Figure 1. A snapshot of work at a point in time as stored in a 'commit'. Credit: ProGit book, by Scott Chacon, CC License.

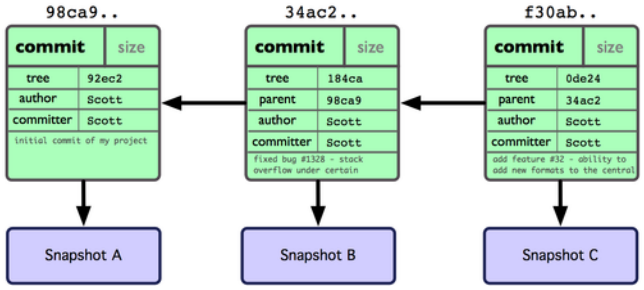

Figure 2. A sequence of commits. Credit: ProGit book, by Scott Chacon, CC License..

```
cd /tmp
git clone https://github.com/paciorek/stat243-fall-2013
git pull
## Cloning into 'stat243-fall-2013'...
```

The goal of this section is to learn how to do more with Git to actually manage a project of your own.

14.2 Hashing

Hashing provides a way to have a unique identifier for a given set of information, such as a file or set of files.

A toy "implementation"

```
library("digest")
# first commit
data1 <- "This is the start of my paper2."
meta1 <- "date: 8/20/13"
hash1 <- digest(c(data1, meta1), algo = "sha1")
cat("Hash:", hash1)

## Hash: 32cff5042299244c8c248f6804bb9756912e5492

# second commit, linked to the first
data2 <- "Some more text in my paper..."
meta2 <- "date: 8/20/13"
# Note we add the parent hash here!
hash2 <- digest(c(data2, meta2, hash1), algo = "sha1")
cat("Hash:", hash2)

## Hash: 43925778805d33de28aba5697a967f04613f9153
```

14.3 Stage 1: Local, single-user, linear workflow

Simply type `git` (or `git help`) to see a full list of all the 'core' commands. We'll now go through most of these via small practical exercises:

```
git help
## usage: git [--version] [--exec-path[=<path>]] [--html-path] [--man-path]
##            [-p|--paginate|--no-pager] [--no-replace-objects] [--bare]
##            [--git-dir=<path>] [--work-tree=<path>] [--namespace=<name>]
##            [-c name=value] [--help]
##            <command> [<args>]
##
## The most commonly used git commands are:
##    add        Add file contents to the index
##    bisect     Find by binary search the change that introduced a bug
##    branch     List, create, or delete branches
##    checkout   Checkout a branch or paths to the working tree
##    clone      Clone a repository into a new directory
##    commit     Record changes to the repository
##    diff       Show changes between commits, commit and working tree, etc
##    fetch      Download objects and refs from another repository
##    grep       Print lines matching a pattern
##    init       Create an empty git repository or reinitialize an existing
##    log        Show commit logs
##    merge      Join two or more development histories together
##    mv         Move or rename a file, a directory, or a symlink
##    pull       Fetch from and merge with another repository or a local bra
##    push       Update remote refs along with associated objects
##    rebase     Forward-port local commits to the updated upstream head
##    reset      Reset current HEAD to the specified state
##    rm         Remove files from the working tree and from the index
##    show       Show various types of objects
##    status     Show the working tree status
##    tag        Create, list, delete or verify a tag object signed with GP
##
## See 'git help <command>' for more information on a specific command.
```

14.3.1 Initializing a Git repository

We use `git init` to create an empty repository

```
cd /tmp
rm -rf git-demo
git init git-demo
## Initialized empty Git repository in /tmp/git-demo/.git/
```

Let's look at what git did:

```
cd /tmp/git-demo
ls -al
ls -al .git
## total 48
## drwxr-xr-x   3 paciorek scfstaff  4096 Sep  1 12:19 .
## drwxrwxrwt 245 root     root     36864 Sep  1 12:19 ..
## drwxr-xr-x   7 paciorek scfstaff  4096 Sep  1 12:19 .git
## total 40
## drwxr-xr-x 7 paciorek scfstaff 4096 Sep  1 12:19 .
## drwxr-xr-x 3 paciorek scfstaff 4096 Sep  1 12:19 ..
## drwxr-xr-x 2 paciorek scfstaff 4096 Sep  1 12:19 branches
## -rw-r--r-- 1 paciorek scfstaff   92 Sep  1 12:19 config
## -rw-r--r-- 1 paciorek scfstaff   73 Sep  1 12:19 description
## -rw-r--r-- 1 paciorek scfstaff   23 Sep  1 12:19 HEAD
## drwxr-xr-x 2 paciorek scfstaff 4096 Sep  1 12:19 hooks
## drwxr-xr-x 2 paciorek scfstaff 4096 Sep  1 12:19 info
## drwxr-xr-x 4 paciorek scfstaff 4096 Sep  1 12:19 objects
## drwxr-xr-x 4 paciorek scfstaff 4096 Sep  1 12:19 refs
```

14.4 Adding content to a repository

Now let's edit our first file in the test directory with a text editor... I'm doing it programatically here for automation purposes, but you'd normally be editing by hand

```
cd /tmp/git-demo
echo "My first bit of text" > file1.txt
```

Now we can tell git about this new file using the 'add' command:
<<chunk, engine='bash'>>= cd /tmp/git-demo git add file1.txt @

We can now ask git about what happened with ‘status‘:

<<chunk, engine=’bash’>>= cd /tmp/git-demo git status @

‘git commit‘: permanently record our changes in git’s database

For now, we are *always* going to call ‘git commit‘ either with the ‘-a‘ option *or* with specific filenames (‘git commit file1 file2...‘). This delays the discussion of an aspect of git called the *index* (often referred to also as the ’staging area’) that we will cover later. Most everyday work in regular scientific practice doesn’t require understanding the extra moving parts that the index involves, so on a first round we’ll bypass it. Later on we will discuss how to use it to achieve more fine-grained control of what and how git records our actions.

<<chunk, engine=’bash’>>= cd /tmp/git-demo git commit -a -m"This is our first commit" @

In the commit above, we used the ‘-m‘ flag to specify a message at the command line. If we don’t do that, git will open the editor we specified in our configuration above and require that we enter a message. By default, git refuses to record changes that don’t have a message to go along with them (though you can obviously ’cheat’ by using an empty or meaningless string: git only tries to facilitate best practices, it’s not your nanny).

‘git log‘: what has been committed so far

<<chunk, engine=’bash’>>= cd /tmp/git-demo git log @

‘git diff‘: what have I changed?

Let’s do a little bit more work... Again, in practice you’ll be editing the files by hand, here we do it via shell commands for the sake of automation (and therefore the reproducibility of this tutorial!)

<<chunk, engine=’bash’>>= cd /tmp/git-demo echo "And now some more text..." >> file1.txt @

And now we can ask git what is different:

<<chunk, engine=’bash’>>= cd /tmp/git-demo git diff @

The cycle of git virtue: work, commit, work, commit, ...

<<chunk, engine=’bash’>>= cd /tmp/git-demo git commit -a -m"I have made great progress on this critical matter." @

* ‘git log‘ revisited

First, let’s see what the log shows us now:

<<chunk, engine=’bash’>>= cd /tmp/git-demo git log @

Sometimes it’s handy to see a very summarized version of the log:

<<chunk, engine=’bash’>>= cd /tmp/git-demo git log –oneline –topo-order –graph @

Git supports *aliases:* new names given to command combinations. Let’s make this handy shortlog an alias, so we only have to type ‘git slog‘ and see this compact log:

<<chunk, engine='bash'>>= cd /tmp/git-demo # We create our alias (this saves it in git's permanent configuration file): git config –global alias.slog "log –oneline –topo-order –graph" # And now we can use it git slog @

‘git mv‘ and ‘rm‘: moving and removing files

While ‘git add‘ is used to add fils to the list git tracks, we must also tell it if we want their names to change or for it to stop tracking them. In familiar Unix fashion, the ‘mv‘ and ‘rm‘ git commands do precisely this:

<<chunk, engine='bash'>>= cd /tmp/git-demo git mv file1.txt file-newname.txt git status @

Note that these changes must be committed too, to become permanent! In git's world, until something hasn't been committed, it isn't permanently recorded anywhere.

<<chunk, engine='bash'>>= cd /tmp/git-demo git commit -a -m"I like this new name better" echo "Let's look at the log again:" git slog @

And ‘git rm‘ works in a similar fashion.

Optional: Exercise

Add a new file ‘file2.txt‘, commit it, make some changes to it, commit them again, and then remove it (and don't forget to commit this last step!).

14.5 Branches

What is a branch? Simply a *label for the 'current' commit in a sequence of ongoing commits*:

There can be multiple branches alive at any point in time; the working directory is the state of a special pointer called HEAD. In this example there are two branches, *master* and *testing*, and *testing* is the currently active branch since it's what HEAD points to:

Once new commits are made on a branch, HEAD and the branch label move with the new commits:

This allows the history of both branches to diverge:

But based on this graph structure, git can compute the necessary information to merge the divergent branches back and continue with a unified line of development:

Local user, branching: an example

Let's now illustrate all of this with a concrete example. Let's get our bearings first:

<<chunk, engine='bash'>>= cd /tmp/git-demo git status ls @

We are now going to try two different routes of development: on the ‘master‘ branch we will add one file and on the ‘experiment‘ branch, which we will create, we will add a different one. We will then merge the experimental branch into ‘master‘.

<<chunk, engine=’bash’>>= cd /tmp/git-demo git branch experiment git checkout experiment @

<<chunk, engine=’bash’>>= cd /tmp/git-demo echo "Some crazy idea" > experiment.txt git add experiment.txt git commit -a -m"Trying something new" git slog @

<<chunk, engine=’bash’>>= cd /tmp/git-demo git checkout master git slog @

<<chunk, engine=’bash’>>= cd /tmp/git-demo echo "All the while, more work goes on in master..." >> file-newname.txt git commit -a -m"The mainline keeps moving" git slog @

<<chunk, engine=’bash’>>= cd /tmp/git-demo ls @

<<chunk, engine=’bash’>>= cd /tmp/git-demo git merge experiment git slog @

14.6 Using remotes as a single user

We are now going to introduce the concept of a *remote repository*: a pointer to another copy of the repository that lives on a different location. This can be simply a different path on the filesystem or a server on the internet.

For this discussion, we’ll be using remotes hosted on the [GitHub.com](http://github.com) service, but you can equally use other services like [BitBucket](http://bitbucket.org) or [Gitorious](http://gitorious.org) as well as host your own.

<<chunk, engine=’bash’>>= cd /tmp/git-demo ls echo "Let’s see if we have any remote repositories here:" git remote -v @

Since the above cell didn’t produce any output after the ‘git remote -v‘ call, it means we have no remote repositories configured. We will now proceed to do so. Once logged into GitHub, go to the [new repository page](https://github.com/new) and make a repository called ‘test‘. Do **not** check the box that says ‘Initialize this repository with a README‘, since we already have an existing repository here. That option is useful when you’re starting first at Github and don’t have a repo made already on a local computer.

We can now follow the instructions from the next page:

<<chunk, engine=’bash’>>= cd /tmp/git-demo git remote add origin git@github.com:jarrodmillman/test.git git push -u origin master @

Let’s see the remote situation again:

<<chunk, engine=’bash’>>= cd /tmp/git-demo git remote -v @

We can now [see this repository publicly on github](https://github.com/jarrodmillman/test).

Let's see how this can be useful for backup and syncing work between two different computers. I'll simulate a 2nd computer by working in a different directory...

<<chunk, engine='bash'>>= cd /tmp/ # Here I clone my 'test' repo but with a different name, test2, to simulate a 2nd computer git clone git@github.com:jarrodmillman/test.git test2 cd test2 pwd git remote -v @

Let's now make some changes in one 'computer' and synchronize them on the second.

<<chunk, engine='bash'>>= cd /tmp/test2 # working on computer #2 echo "More new content on my experiment" >> experiment.txt git commit -a -m"More work, on machine #2" @

Now we put this new work up on the github server so it's available from the internet

<<chunk, engine='bash'>>= cd /tmp/test2 # working on computer #2 git push @

Now let's fetch that work from machine #1:

<<chunk, engine='bash'>>= cd /tmp/git-demo git pull @

An important aside: conflict management

While git is very good at merging, if two different branches modify the same file in the same location, it simply can't decide which change should prevail. At that point, human intervention is necessary to make the decision. Git will help you by marking the location in the file that has a problem, but it's up to you to resolve the conflict. Let's see how that works by intentionally creating a conflict.

We start by creating a branch and making a change to our experiment file:

<<chunk, engine='bash'>>= cd /tmp/git-demo git branch trouble git checkout trouble echo "This is going to be a problem..." >> experiment.txt git commit -a -m"Changes in the trouble branch" @

And now we go back to the master branch, where we change the *same* file:

<<chunk, engine='bash'>>= cd /tmp/git-demo git checkout master echo "More work on the master branch..." >> experiment.txt git commit -a -m"Mainline work" @

So now let's see what happens if we try to merge the 'trouble' branch into 'master':

<<chunk, engine='bash'>>= cd /tmp/git-demo git merge trouble @

Let's see what git has put into our file:

<<chunk, engine='bash'>>= cd /tmp/git-demo cat experiment.txt @

At this point, we go into the file with a text editor, decide which changes to keep, and make a new commit that records our decision. To automate my edits, I use the 'sed' command.

<<chunk, engine='bash'>>= cd /tmp/git-demo sed -i '/^</d' experiment.txt sed -i '/^>/d' experiment.txt sed -i '/^=/d' experiment.txt @

I've now made the edits, in this case I decided that both pieces of text were useful, so I just accepted both additions.

<<chunk, engine='bash'>>= cd /tmp/git-demo cat experiment.txt @

Let's then make our new commit:

<<chunk, engine='bash'>>= cd /tmp/git-demo git commit -a -m"Completed merge of trouble, fixing conflicts along the way" git slog @

Note: While it's a good idea to understand the basics of fixing merge conflicts by hand, in some cases you may find the use of an automated tool useful. Git supports multiple [merge tools](https://www.kernel.org/pub/software/scm/git/docs/git-mergetool.html): a merge tool is a piece of software that conforms to a basic interface and knows how to merge two files into a new one. Since these are typically graphical tools, there are various to choose from for the different operating systems, and as long as they obey a basic command structure, git can work with any of them.

14.7 Collaborating on github with a small team

Single remote with shared access: we are going to set up a shared collaboration with one partner (the person sitting next to you). This will show the basic workflow of collaborating on a project with a small team where everyone has write privileges to the same repository.

Note for SVN users: this is similar to the classic SVN workflow, with the distinction that commit and push are separate steps. SVN, having no local repository, commits directly to the shared central resource, so to a first approximation you can think of 'svn commit' as being synonymous with 'git commit; git push'.

We will have two people, let's call them Alice and Bob, sharing a repository. Alice will be the owner of the repo and she will give Bob write privileges.

We begin with a simple synchronization example, much like we just did above, but now between *two people* instead of one person. Otherwise it's the same:

- Bob clones Alice's repository. - Bob makes changes to a file and commits them locally. - Bob pushes his changes to github. - Alice pulls Bob's changes into her own repository.

Next, we will have both parties make non-conflicting changes each, and commit them locally. Then both try to push their changes:

- Alice adds a new file, 'alice.txt' to the repo and commits. - Bob adds 'bob.txt' and commits. - Alice pushes to github. - Bob tries to push to github. What happens here?

The problem is that Bob's changes create a commit that conflicts with Alice's, so git refuses to apply them. It forces Bob to first do the merge on his machine, so that if there is a conflict in the merge, Bob deals with the conflict manually (git could try to do the merge on the server, but in that case if there's a conflict, the server repo would be left in a conflicted state without a human to fix things up). The solution is for Bob to first pull the changes (pull in git is really fetch+merge), and then push again.

14.8 More Git 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.h
- Git branching demo: http://pcottle.github.io/learnGitBranching/