Cheatsheet: Git

Luke Johnston

2015-03-23

Git is an amazing and very powerful tool that is useful for managing your projects and tools, letting you experiment and try out new things in your files without worrying about losing anything. However, because of its power and usefulness, it can be confusing for beginners as 1) Git requires the use of the command line (or terminal) to run Git commands, and 2) there are a large number of commands and options available. So, we put together the commands that are the most useful and most common — and the only ones you may ever use!

Before using Git: Initial setup

This following commands should be run first before any work is done using Git.

```
git config --global user.name "Your Name"
git config --global user.email "you@some.domain"
git config --global color.ui "auto"
git config --global core.editor "your_editor"
git config --global push.default current
git config --list
```

These commands basically tell Git:

- Who you are (`user.name`)
- What your email address is (`user.email`; used when working on a multiple person project)
- What the colour output should be after running Git commands (`color.ui`)
- What your **text** editor is that you use (`core.editor`). See our instructions for more details about editors, but briefly:
- On Windows, you will likely use Notepad (which comes pre-installed on Windows) or Notepad++
- On Mac, there is TextWrangler (which I believe comes pre-installed) or TextMate
- On Linux, it depends on which distro you use, but in general gedit or nano usually are pre-installed
- How you want the push and pull default behaviour (`push.default`; more on this in the GitHub lesson)

Useful (and common) Git commands

`git init`

Tell Git to start tracking a folder by creating a git history repository. This essentially tells Git to start watching your folder and all the files and folders within it.

Example code:

```
## Comment: cd = change directory (aka folder)
cd ~/Documents/yourprojectname/
git init
```

`git status`

Tell Git to list out all the activity within a folder under watch by Git (after using `git init`). The status command will list all files or folders that have been added or changed inside the repository.

Example code:

```
cd ~/Documents/yourprojectname/
git status
```

`git add <files>`

Add the files you want Git to watch inside the history repository as created with `git init`. You can specify as many or as little files or folders as you want. *Note*: the add command does **not** save the files to the git history. All the `git add` command does is tell Git to start watching the files and put the files into the "staging area" where they will next be saved to the history (see the below `git commit` command).

Example code:

```
## Comment: cd to where files have been changed
cd ~/Documents/yourprojectname/
## Comment: Pretending we want to add three files
git add foldername/newfilename1 newfilename2 foldername2/changedfile
```

`git commit`

Tells Git to save your `git add`'ed files to its history. This is the main purpose and use for Git. After you commit a file(s) and typed out a commit message (*be detailed about what you did*!!), Git will save (or take a "snap shot") of your files and put it into its history. Once committed, it is saved into the history, allowing you to go back to that commit/save point at any time!

Example code:

```
cd ~/Documents/yourprojectname/
## Comment: You have two options...
## Commit and let a text editor pop up so you can
## write your commit message
git commit
## Comment: ... Or you can use the -m option
git commit -m "Type out your commit message here"
```

`git log`

Displays the history of your repository and the messages you added to each commit, as well as the date of the change, and who made the change. This is analogous to a logbook for those in the basic science. This is a really useful feature if you are working on a multi-person project or if you are coming back to a project after several months of not touching it and completely forgetting what you were doing last. `git log` has a *large* number of options available that customize the appearance and the information provided by the log command.

Example code:

```
cd ~/Documents/yourprojectname/
git log
```

`git checkout`

Allows you to go back (or forward) throughout your Git history as well as to change branches (see the `git branch` command below). If you wanted to go back a few commits, you would run `git checkout commitnumber`. You get the commit number by running git log and using the first few letters and numbers of the commit. We will likely show an example in the workshop.

Example code:

```
## Comment: Go back to a previous commit (use the commit hash)
git checkout d45gfd3 ## Example commit hash, found using git log
## Comment: Go to another branch (ie: "testing")
git checkout testing
## Comment: Go back to your main branch ("master")
git checkout master
```

git branch <branch-name>

This is another very useful command from Git, and (I feel) one of its biggest strengths! This command basically makes a branch like on a tree, letting you experiment with your files and statistical analyses without having to worry about messing around with your main files. If you want to eventually bring the experimental branch into the "master" branch, you can use `git merge` (which we will not be covering as it is a slightly more advanced command).

Example code:

```
## Create a branch named "experiment"
git branch experiment
## Move to the experiment branch
git checkout experiment
## edit your files and save, do whatever...
git add changedfiles
git commit -m "Added a statistical test with different variables"
## Move back to the master branch
git checkout master
## All files should be in their original state
```

git diff <commit-id (optional)> <file(s)>

This command tells Git to compare the contents of, at the basic level, two files. These two files are by default a file you most recently changed (but not committed) with the same file's content in the history. However, you can also compare a file across different commits in the history. `git diff` shows the differences between the files by highlighting in red deletions and in green additions to the file. This is useful if you forgot what you changed in the file and you are going to commit the file into the history, using it to help you write a better commit messages.

Example code:

```
## Compare a recently changed file with its recent
```

```
## commit history
git diff filename
## Compare a file across two commits (using commit hashes)
## Usage: git diff hash1..hash2 file (hashes are *very* unique)
git diff 54gfd..75g84 filename
```