Introduction to Git for Data Science

**Greg Wilson**

Co-founder of Software Carpentry

### Course Description

Version control is one of the power tools of programming. It allows you to keep track of what you did when, undo any changes you have decided you don't want, and collaborate at scale with other people. This course will introduce you to Git, a modern version control tool that is very popular with data scientists and software developers alike, and show you how it can help you get more done in less time and with less pain.

# Basic Workflow

This chapter explains what version control is and why you should use it, and introduces the most common steps in a common Git workflow.

|  |
| --- |
| **1.1 Want is version control?**  A version control system is a tool that manages changes made to the files and directories in a project. Many version control systems exist; this lesson focuses on one called Git, which is used by many of the data science tools covered in our other lessons. Its strengths are:  Nothing that is saved to Git is ever lost, so you can always go back to see which results were generated by which versions of your programs.  Git automatically notifies you when your work conflicts with someone else's, so it's harder (but not impossible) to accidentally overwrite work.  Git can synchronize work done by different people on different machines, so it scales as your team does.  Version control **isn't just for software**: books, papers, parameter sets, and anything that changes over time or needs to be shared can and should be stored and shared using something like Git. |
| **1.2 Where does Git store information?**  Each of your Git projects has two parts: the files and directories that you create and edit directly, and the extra information that Git records about the project's history. The combination of these two things is called a repository.  Git stores all of its extra information in a directory called .git located in the root directory of the repository. Git expects this information to be laid out in a very precise way, so you should never edit or delete anything in .git.  Suppose your home directory /home/repl contains a repository called dental, which has a sub-directory called data. Where is information about the history of the files in /home/repl/dental/data stored?  Answer: /home/repl/dental/.git (all of the information about a repository is stored under its root directory) |
| **1.3 How can I check the state of a repository?**  When you are using Git, you will frequently want to check the status of your repository. To do this, run the command ***git status***, which displays a list of the files that have been modified since the last time changes were saved.  You have been put in the dental repository. Use git status to discover which file(s) have been changed since the last save. Which file(s) are listed? |
| 1.4 How can I tell what I have changed?  Git has a staging area in which it stores files with changes you want to save that haven't been saved yet. Putting files in the staging area is like putting things in a box, while committing those changes is like putting that box in the mail: you can add more things to the box or take things out as often as you want, but once you put it in the mail, you can't make further changes.    git status shows you which files are in this staging area, and which files have changes that haven't yet been put there. In order to compare the file as it currently is to what you last saved, you can use git diff filename. git diff without any filenames will show you all the changes in your repository, while git diff directory will show you the changes to the files in some directory. |
| **1.4 What is in a diff?**  A **diff** is a formatted display of the differences between two sets of files. Git displays diffs like this:  diff --git a/report.txt b/report.txt  index e713b17..4c0742a 100644  --- a/report.txt  +++ b/report.txt  @@ -1,4 +1,4 @@  -# Seasonal Dental Surgeries 2017-18  +# Seasonal Dental Surgeries (2017) 2017-18  TODO: write executive summary.  This shows:   * The command used to produce the output (in this case, diff --git). In it, a and b are placeholders meaning "the first version" and "the second version". * An index line showing keys into Git's internal database of changes. We will explore these in the next chapter. * --- a/report.txt and +++ b/report.txt, which indicate that lines being removed are prefixed with -, while lines being added are prefixed with +. * A line starting with @@ that tells where the changes are being made. The pairs of numbers are start line,number of lines changed. Here, the diff output shows that 4 lines from line 1 are being removed and replaced with new lines. * A line-by-line listing of the changes with - showing deletions and + showing additions. (We have also configured Git to show deletions in red and additions in green.) Lines that haven't changed are sometimes shown before and after the ones that have in order to give context; when they appear, they don't have either + or - in front of them.   Desktop programming tools like [**RStudio**](https://www.rstudio.com/) can turn diffs like this into a more readable side-by-side display of changes; you can also use standalone tools like **[DiffMerge](https://sourcegear.com/diffmerge/" \t "_blank)** or **[WinMerge](http://winmerge.org/" \t "_blank)**.  You have been put in the dental repository. Use git diff data/northern.csv to look at the changes to that file. How many lines have been changed? |
| 1.5 What's the first step in saving changes? You commit changes to a Git repository in two steps:   1. Add one or more files to the staging area. 2. Commit everything in the staging area.   To add a file to the staging area, use git add filename. |
| 1.6 How can I tell what's going to be committed? To compare the state of your files with those in the staging area, you can use git diff -r HEAD. The -r flag means "compare to a particular revision", and HEAD is a shortcut meaning "the most recent commit".  You can restrict the results to a single file or directory using git diff -r HEAD path/to/file, where the path to the file is relative to where you are (for example, the path from the root directory of the repository).  We will explore other uses of -r and HEAD in the next chapter. |
| 1.7 Interlude: how can I edit a file? Unix has a bewildering variety of text editors. In this course, we will sometimes use a very simple one called Nano. If you type nano filename, it will open filename for editing (or create it if it doesn't already exist). You can then move around with the arrow keys, delete characters with the backspace key, and so on. You can also do a few other operations with control-key combinations:   * Ctrl-K: delete a line. * Ctrl-U: un-delete a line. * Ctrl-O: save the file ('O' stands for 'output'). * Ctrl-X: exit the editor. |
| 1.8 How do I commit changes? To save the changes in the staging area, you use the command git commit. It always saves everything that is in the staging area as one unit: as you will see later, when you want to undo changes to a project, you undo all of a commit or none of it.  When you commit changes, Git requires you to enter a **log message**. This serves the same purpose as a comment in a program: it tells the next person to examine the repository why you made a change.  By default, Git launches a text editor to let you write this message. To keep things simple, you can use -m "some message in quotes" on the command line to enter a single-line message like this:  git commit -m "Program appears to have become self-aware."  If you accidentally mistype a commit message, you can change it using the --amend flag.  git commit --amend - m "new message" |
| 1.9 How can I view a repository's history? The command git log is used to view the **log** of the project's history. Log entries are shown most recent first, and look like this:  commit 0430705487381195993bac9c21512ccfb511056d  Author: Rep Loop <repl@datacamp.com>  Date: Wed Sep 20 13:42:26 2017 +0000  Added year to report title.  The commit line displays a unique ID for the commit called a **hash**; we will explore these further in the next chapter. The other lines tell you who made the change, when, and what log message they wrote for the change.  When you run git log, Git automatically uses a pager to show one screen of output at a time. Press the **space bar** to go down a page or the **'q'** key to quit. |
| 1.10 How can I view a specific file's history? A project's entire log can be overwhelming, so it's often useful to inspect only the changes to particular files or directories. You can do this using git log path, where path is the path to a specific file or directory. The log for a file shows changes made to that file; the log for a directory shows when files were added or deleted in that directory, rather than when the contents of the directory's files were changed. |
| 1.11 How do I write a better log message? Writing a one-line log message with git commit -m "message"is good enough for very small changes, but your collaborators (including your future self) will appreciate more information. If you run git commit without -m "message", Git launches a text editor with a template like this:  # Please enter the commit message for your changes. Lines starting  # with '#' will be ignored, and an empty message aborts the commit.  # On branch master  # Your branch is up-to-date with 'origin/master'.  #  # Changes to be committed:  # modified: skynet.R  #  The lines starting with # are comments, and won't be saved. (They are there to remind you what you are supposed to do and what files you have changed.) Your message should go at the top, and may be as long and as detailed as you want. |
| Neat! This concludes chapter 1, where you learned about git diff, git status, git add and git commit. Quite something! Rush over to chapter 2 to continue your Git adventure! |

**2. Repositories**

This chapter digs a little deeper into how Git stores information and how you can explore a repository's history.

|  |
| --- |
| **2.1** How does Git store information? You may wonder what information is stored by each commit that you make. Git uses a three-level structure for this.   1. A **commit** contains metadata such as the author, the commit message, and the time the commit happened. In the diagram below, the most recent commit is at the bottom (feed0098), and vertical arrows point up towards the previous ("parent") commits. 2. Each commit also has a **tree**, which tracks the names and locations in the repository when that commit happened. In the oldest (top) commit, there were two files tracked by the repository. 3. For each of the files listed in the tree, there is a **blob**. This contains a compressed snapshot of the contents of the file when the commit happened. (Blob is short for binary large object, which is a SQL database term for "may contain data of any kind".) In the middle commit, report.md and draft.md were changed, so the blobs are shown next to that commit. data/northern.csv didn't change in that commit, so the tree links to the blob from the previous commit. Reusing blobs between commits help make common operations fast and minimizes storage space. |
| **2.2** What is a hash? Every commit to a repository has a unique identifier called a **hash** (since it is generated by running the changes through a pseudo-random number generator called a **hash function**). This hash is normally written as a 40-character hexadecimal string like 7c35a3ce607a14953f070f0f83b5d74c2296ef93, but most of the time, you only have to give Git the first 6 or 8 characters in order to identify the commit you mean.  Hashes are what enable Git to share data efficiently between repositories. If two files are the same, their hashes are guaranteed to be the same. Similarly, if two commits contain the same files and have the same ancestors, their hashes will be the same as well. Git can therefore tell what information needs to be saved where by comparing hashes rather than comparing entire files. |
| **2.3** How can I view a specific commit? To view the details of a specific commit, you use the command git show with the first few characters of the commit's hash. For example, the command git show 0da2f7 produces this:  commit 0da2f7ad11664ca9ed933c1ccd1f3cd24d481e42  Author: Rep Loop <repl@datacamp.com>  Date: Wed Sep 5 15:39:18 2018 +0000  Added year to report title.  diff --git a/report.txt b/report.txt  index e713b17..4c0742a 100644  --- a/report.txt  +++ b/report.txt  @@ -1,4 +1,4 @@  -# Seasonal Dental Surgeries 2017-18  +# Seasonal Dental Surgeries (2017) 2017-18  TODO: write executive summary.  The first part is the same as the log entry shown by git log. The second part shows the changes; as with git diff, lines that the change removed are prefixed with -, while lines that it added are prefixed with +. |
| **2.4** What is Git's equivalent of a relative path? A hash is like an absolute path: it identifies a specific commit. Another way to identify a commit is to use the equivalent of a relative path. The special label HEAD, which we saw in the previous chapter, always refers to the most recent commit. The label HEAD~1 then refers to the commit before it, while HEAD~2 refers to the commit before that, and so on.  Note that the symbol between HEAD and the number is a tilde ~, not a minus sign -, and that there cannot be spaces before or after the tilde. |
| **2.5** How can I see who changed what in a file? git log displays the overall history of a project or file, but Git can give even more information: the command git annotate fileshows who made the last change to each line of a file and when. For example, the first three lines of output from git annotate report.txt look something like this:  04307054 ( Rep Loop 2017-09-20 13:42:26 +0000 1)# Seasonal Dental Surgeries (2017) 2017-18  5e6f92b6 ( Rep Loop 2017-09-20 13:42:26 +0000 2)  5e6f92b6 ( Rep Loop 2017-09-20 13:42:26 +0000 3)TODO: write executive summary.  Each line contains five things, with two to four in parentheses.   1. The first eight digits of the hash, 04307054. 2. The author, Rep Loop. 3. The time of the commit, 2017-09-20 13:42:26 +0000. 4. The line number, 1. 5. The contents of the line, # Seasonal Dental Surgeries (2017) 2017-18.  **git annotate let's you see who modified a file and when.** |
| **2.6** How can I see what changed between two commits? git show with a commit ID shows the changes made in a particular commit. To see the changes between two commits, you can use git diff ID1..ID2, where ID1 and ID2 identify the two commits you're interested in, and the connector .. is a pair of dots. For example, git diff abc123..def456 shows the differences between the commits abc123 and def456, while git diff HEAD~1..HEAD~3 shows the differences between the state of the repository one commit in the past and its state three commits in the past. |
| **2.7** How do I add new files? Git does not track files by default. Instead, it waits until you have used git add at least once before it starts paying attention to a file.  In the diagram you saw at the start of the chapter, the untracked files won't have a blob, and won't be listed in a tree.  The untracked files won't benefit from version control, so to make sure you don't miss anything, git status will always tell you about files that are in your repository but aren't (yet) being tracked. |
| **2.8** How do I tell Git to ignore certain files? Data analysis often produces temporary or intermediate files that you don't want to save. You can tell it to stop paying attention to files you don't care about by creating a file in the root directory of your repository called .gitignore and storing a list of **wildcard** patterns that specify the files you don't want Git to pay attention to. For example, if .gitignore contains:  build  \*.mpl  then Git will ignore any file or directory called build (and, if it's a directory, anything in it), as well as any file whose name ends in .mpl.  Which of the following files would not be ignored by a .gitignore that contained the lines:  pdf  \*.pyc  backup **pdf does not contain any wildcards, so it only matches files called pdf.** |
| **2.9** How can I remove unwanted files? Git can help you clean up files that you have told it you don't want. The command git clean -n will show you a list of files that are in the repository, but whose history Git is not currently tracking. A similar command git clean -f will then delete those files.  Use this command carefully: git clean only works on untracked files, so by definition, their history has not been saved. If you delete them with git clean -f, they're gone for good. |
| **Keeping your repository clean is crucial to keep oversight.** |
| **2.10** How can I see how Git is configured? Like most complex pieces of software, Git allows you to change its default settings. To see what the settings are, you can use the command git config --list with one of three additional options:   * --system: settings for every user on this computer. * --global: settings for every one of your projects. * --local: settings for one specific project.   Each level overrides the one above it, so **local settings** (per-project) take precedence over **global settings** (per-user), which in turn take precedence over **system settings** (for all users on the computer).  Config settings are useful for storing your name and email address (to identify you in commit logs), choosing your favorite text editor and diff view tools, and customizing things just how you like them. |
| **2.11** How can I change my Git configuration? Most of Git's settings should be left as they are. However, there are two you should set on every computer you use: your name and your email address. These are recorded in the log every time you commit a change, and are often used to identify the authors of a project's content in order to give credit (or assign blame, depending on the circumstances).  To change a configuration value for all of your projects on a particular computer, run the command:  git config --global setting.name setting.value  with the setting's name and value in the appropriate places. The keys that identify your name and email address are user.name and user.email respectively. |