Basics of version control

Matthew Evans

Part II Computational Physics, Lent 2019

Introduction . . . 1
What is version control and why should I care? . . . 1
What is Git? . . . 2
What is a repository? . . . 3
What is a commit? . . . 3
Worked examples . . . 4
More advanced usage . . . 10
Branching, merging and collaboration . . . 10
Test-driven development . . . 10
Appendices . . . 10
Basic subcommand cheatsheet . . . 10
Useful links . . . 12

Introduction

What is version control and why should I care?

Put simply, a version control system (VCS) (a.k.a. revision or source control) is a system that tracks and manages changes to a set of data (e.g. source code). The Emacs website summarises VCS by three capabilities:

- Reversibility: the ability to back up to a previous state if you discover that some modification you did was a mistake or a bad idea.
- Concurrency: the ability to have many people modifying the same collection of files knowing that conflicting modifications can be detected and resolved.
- History: the ability to attach historical data to your data, such as explanatory comments about the intention behind each change. Even for a programmer working solo, change histories are an important aid to memory; for a multi-person project, they are a vitally important form of communication among developers.

Say you are writing a report; instead of renaming different versions of files `report.tex`, `report_final.tex`, `report_FINAL.tex`, `report_FINALFINAL.tex`, `report_FINAL_abcdef.tex`, ad infinitum, version control systems allow you to save the entire history of a file

as a series of staged changes, often called commits or revisions. This may not seem so useful for a linear process such as a single person writing a document, but for the non-linear process of software development, version control is a must. Every single serious software project exists under some form of version control, almost by definition. Learning effective version control is a vital skill, not just for careers in the tech industry, but also for producing sustainable software and thus reproducible science.

The aim of this tutorial is to teach basic concepts of version control that you might consider using for the coursework. A few hours of investment, and maybe some moments of confusion, will hopefully lead to a productivity boost. We shall work with the Git (`git`) version control system as, at the time of writing this, it dominates the market, with Subversion (`svn`) and Mercurial (`hg`) lagging behind, as evidenced by e.g. Google trends.

What is Git?

Git was created by Linus Torvalds in 2005 [1] to manage the source code of the Linux kernel, with the requirements that it be fast, distributed and secure. At the time, the Linux kernel was 6 million lines long [2], with thousands of developers worldwide [3]; whilst famous for creating Linux, many argue that Git is Linus Torvalds' greatest technical achievement.

`git`, like its forebears `svn` and `hg`, is what is called a distributed version control system. This means that there is no "master copy" of a project, and instead the entire history of a project is mirrored on the computer of every developer (and potentially user). This becomes extremely useful when multiple people are actively developing a project for reasons we shall touch on later.

Git has somewhat of a reputation for being difficult to learn and master, due its 21 different subcommands, and their myriad options. Most simple use cases, however, require only a few of these commands. A complete and in-depth documentation of their operation can be found at the command-line by running `git help <subcommand>`, whilst a brief summary on the subsection of Git that we will discuss can be found in the cheatsheet in the appendix. Several graphical user interfaces (GUIs) also exist for Git, which you may prefer, though I do not have one to recommend.

Without going into detail, Git calculates *hashes* (SHA-1[4]) of the files it tracks and uses them to quickly compare files. The files themselves are compressed and stored as changes relative to one another (even so, the `.git` folder can become rather large). If you are interested in what Git is doing under the hood, have a look at Chapter 10 of Pro Git.

[1] Linus Torvalds started writing Git on April 3rd 2005, it was then hosting its own source code by April 7th (source), and then was hosting the entire Linux kernel (2.6.12-rc2) by April 16th.

[2] According to the spike in 2005 on the Linux kernel's code frequency graph on Github, which is actually so large it breaks the rendering of the scale (each division is 1 million lines of code).

[3] ~14,500 people posted on the Linux kernel mailing list between 1995 and 2000, according to this report.

[4] The first SHA-1 "hash collision" (different files contents with the same hash: could allow for malicious file injection) occured in 2017 so Git is probably going to migrate to a more secure hashing algorithm.

What is a repository?

A repository (or repo) is a top-level directory of files and directories that is managed by a version control system, containing e.g. the source of an entire software project. Often, the term repository refers to a location on a remote server that maintains a central copy of the project that developers can push changes in their *local repository* to, and pull other people's changes from. All of the project history is stored in the `.git` folder at the top-level of the repository.

In the past, developers would often directly push to or pull from each other's local repositories, but nowadays most projects will host their VCS on either a private server, or trust of the many web-based version control service providers. The main players in this field are GitHub, BitBucket and GitLab, which each provide similar web interfaces to Git (or otherwise) repositories. These web services are very useful to individual users who don't want to run their own server, and in fact this very tutorial is hosted on GitHub. Open source software has really benefited from these web-facing services that provide centralised, discoverable web pages for projects, as well as ways to talk to and raise issues with developers, lowering the bar to making contributions yourself. Additionally, web-based services allow for the automation of many useful checks on software, as we shall see in the test-driven development section.

Each provider has its own advantages and disadvntages to consider but for our usage they are all broadly similar (and all allow unlimited private repos for free, often providing extra benefits for students [5]). One of the useful features of distributed VCS is that you can easily transfer your entire code history to a different provider, since you are constantly mirroring your own version of the project locally [6].

What is a commit?

A commit (a.k.a. revision, changeset) is a set of file modifications grouped under the same user-provided descriptive comment and a randomly-generated hash, providing a snapshot in time of the entire repository. It's important to note that commits stack on top of each other (in the sense of stack memory: last in, first out).

Here are some practical questions you might ask about commits:

- How often should I commit?
  - The changes to code that you commit can be as fine or coarse-grained as necessary, depending on personal preference.
  - When writing a new code, the first commit might not occur until
- What makes a good commit message?
  - Simply a short description of the changes made. For example, good messages include "fixed typo in example.cpp", "added diffraction plotting function", "got ODE solver working" and "attempted second example", but

[5] For example, GitHub offer the free student developer pack which provides a lot of resources on third-party plugins and graphical interfaces to Git, and BitBucket provide their own BitBucket education accounts.

[6] When Microsoft acquired GitHub in late 2018, giving them soft power over a large majority of open source software, anyone who objected to Microsoft's business practices could simply point their repository at a new remote to move all commit history.

"fixed example", "added function", "code now working" and "end of first practical" are less good. If you want to write a long description, write a short description as the first line, then use new lines to add detail.

- Can I edit previous commits?
  - You can edit the last commit, before pushing, using `git commit --amend`, but not any older commits.
- Can I undo a commit?
  - You can revert *to* a commit, so you can "undo" the most recent commit, but you can't undo an older commit.

Worked examples

You may find these examples easier to follow online, where text can be copied more easily: https://www.github.com/ml-evs/part2-computing-git-tutorial.

Setup First things first, check that you have Git installed on the machine you're using for these tutorials. On Linux/*nix (including Windows Subsystem for Linux), at the command-line:

```
$ git --version
git version 2.20.1
```

If you are using Windows directly, you need Git installed to reach this shell.

The next thing to do is register who you are to Git:

```
$ git config --global user.name "<your_name_here>"
$ git config --global user.email "<your_email_here>"
```

Example 1.1: a contrived local repository In this example, we will make a local Git repository, add some files to it, commit them, make some changes, then commit the changes. Very exciting. First, lets make a file with some text in it, by reciting Moby Dick from memory…

```
$ echo "Call me Moby. Some years ago - never mind how long precisely -
    having little or no money in my purse, and nothing particular to
    interest me on shore, I thought I would sail about a little and see
     the water part of the world." >> moby.txt
$ git status
On branch master

No commits yet

Untracked files:
  (use "git add <file>..." to include in what will be committed)
```

```
  moby.txt

nothing added to commit but untracked files present (use "git add" to
    track)
```

As you can see, although the file is in the correct folder, it is not being tracked by Git until we `add` it.

```
$ git add moby.txt
$ git status
On branch master

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)

  new file:   moby.txt
```

We can now write our first commit, bearing in mind the advice on what makes a good commit message (try to write your own).

```
$ git commit
```

Another quick `git status` will show that there is now `nothing to commit, working tree clean`. We're now ready to add the next sentence, which will let us look at a `diff` between the current state of our history (called `HEAD`). Git `diff` will open the diff in `less`, press Q to quit.

```
$ echo "The morning was so damp and misty that it was only with great
    difficulty that the day succeeded in breaking." >> moby.txt
$ git diff
diff --git a/moby.txt b/moby.txt
index c256b41..6a234d6 100644
--- a/moby.txt
+++ b/moby.txt
@@ -1 +1,2 @@
 Call me Moby. Some years ago - never mind how long precisely - having
     little or no money in my purse, and nothing particular to interest
      me on shore, I thought I would sail about a little and see the
     water part of the world.
+The morning was so damp and misty that it was only with great
    difficulty that the day succeeded in breaking.
```

Let's commit this sentence. If you use `git commit` as before, you will notice that this change has not yet been "staged". We can either run `git add moby.txt` to stage it (useful for commits with many files to be added), or just commit the file directly:

```
$ git commit moby.txt -m 'Added another sentence of Moby Dick from
    memory'
```

What does our commit history look like now?

```
$ git log
commit 3efacaf57c09387c701110d73f354286e3d4e669 (HEAD -> master)
Author: Matthew Evans <me388@cam.ac.uk>
Date:   Mon Jan 28 21:13:15 2019 +0000

    Added another sentence of Moby Dick from memory

commit 81ec9e18c39dc6919146fcb9677a11a31d76719d
Author: Matthew Evans <me388@cam.ac.uk>
Date:   Mon Jan 28 21:12:40 2019 +0000

    Added first sentences of Moby Dick
```

You will notice that `git log` provides us with a long commit hash, who made the commit and when, and tells us where the current `HEAD` is.

Those of you who have ever read a book might realise that my memory of Moby Dick isn't very good, so let's fix that. Either edit `moby.txt` in your editor, or use the following `sed` command to fix the first sentence.

```
$ sed -i 's/Moby/Ishmael/g' moby.txt
$ git diff
diff --git a/moby.txt b/moby.txt
index 6a234d6..e966d0d 100644
--- a/moby.txt
+++ b/moby.txt
@@ -1,2 +1,2 @@
-Call me Moby. Some years ago - never mind how long precisely - having
    little or no money in my purse, and nothing particular to interest
    me on shore, I thought I would sail about a little and see the
    water part of the world.
+Call me Ishmael. Some years ago - never mind how long precisely -
    having little or no money in my purse, and nothing particular to
    interest me on shore, I thought I would sail about a little and see
     the water part of the world.
 The morning was so damp and misty that it was only with great
     difficulty that the day succeeded in breaking.
```

You will now see that our `diff` compares the line as it exists in `HEAD`, and after our changes. You should now `commit` this change before we try some other sub-commands.

If you're paying attention to the important stuff in this tutorial (the literature), you will notice that the last sentence we added was actually from Dostoyevsky's "The Idiot",

which is the book I wanted to type out for some reason(?) all along. Easy mistake to make, I know. Let's fix that!

First, let's rename our file to something more appropriate, using `git mv` to ensure that the history of the file is preserved across the rename (using just Linux's `mv` would make Git think we had deleted the file and made a completely new unrelated file). We will then use a magic `sed` one-liner to replace the first line in our file with the correct sentence from The Idiot, then commit the file in anger at how we could have been so stupid.

```
$ git mv moby.txt idiot.txt
$ sed '1s/.*/Towards the end of November, during a thaw, at nine o'
    clock one morning, a train on the Warsaw and Petersburg railway was
     approaching the latter city at full speed.' idiot.txt
$ git commit idiot.txt -m 'Fixed mistake where I wrote out Moby Dick
    instead of The Idiot, like The Idiot I am'
```

Coming back 5 minutes later, you think that you probably shouldn't have written that commit message, so let's fix that by writing something more suitable, that impresses everyone.

```
$ git commit --amend
```

Just to prove our repository has kept all of its history, and get the `diff` from our first commit:

```
$ git log
commit a0e8df557279270a9fa686c0ef7a44e347c189ce (HEAD -> master)
Author: Matthew Evans <me388@cam.ac.uk>
Date:   Mon Jan 28 21:40:41 2019 +0000

    Fixed subtle bug in Linux kernel (I am very smart) that caused Moby
         Dick to be written to file instead of The Idiot

commit e2b7ad72976d4c00ab6d0214ab41d3c13869c2dd
Author: Matthew Evans <me388@cam.ac.uk>
Date:   Mon Jan 28 21:32:41 2019 +0000

    Fixed name in first sentence of Moby Dick

commit 92b27f011520c0ae99261203bb12b4fa4b11bbf4
Author: Matthew Evans <me388@cam.ac.uk>
Date:   Mon Jan 28 21:11:58 2019 +0000

    Added another sentence of Moby Dick from memory

commit 473e3ad8df6fd2b6602a71bb1d764d7e4898b1ef
Author: Matthew Evans <me388@cam.ac.uk>
Date:   Mon Jan 28 21:11:23 2019 +0000
```

```
    Added first sentences of Moby Dick

$ git diff 473e3a
diff --git a/idiot.txt b/idiot.txt
new file mode 100644
index 0000000..9925ab9
--- /dev/null
+++ b/idiot.txt
@@ -0,0 +1,2 @@
+Towards the end of November, during a thaw, at nine o clock one
    morning, a train on the Warsaw and Petersburg railway was
    approaching the latter city at full speed.
+The morning was so damp and misty that it was only with great
    difficulty that the day succeeded in breaking.
diff --git a/moby.txt b/moby.txt
deleted file mode 100644
index c256b41..0000000
--- a/moby.txt
+++ /dev/null
@@ -1 +0,0 @@
-Call me Moby. Some years ago - never mind how long precisely - having
    little or no money in my purse, and nothing particular to interest
    me on shore, I thought I would sail about a little and see the
    water part of the world.
```

(Note how we only needed to use the first 6 characters of the commit hash when running `git`. Alternatively we could have used `git diff HEAD~4` to reference the commit 4 behind the current `HEAD`.

Example 1.2: Remote version control Our aim is now to take a repository to the cloud. For this, you will need an account with your favourite cloud VCS, for example GitHub, Bitbucket or GitLab. For the sake of this example, I will follow the live demo from the lecture and make a repository for my solutions to the exercises on GitHub.

Every provider should have a simple interface for creating a new repository, which we can give any name we want. GitHub tells me that `part2-computing-exercises` is a fine name, we can select whether we want to the repository to be private or public, and then we can decide whether we want to select a license for the code [7]. After creating the remote repository, we will be provided with a url and some instructions on how to set up our local copy; this example will expand those instructions below.

Now, on our local machine, we navigate to the top-level folder that we want to track with version control, which in my case, has sub-directories for each exercise.

```
$ ls
exercise1  exercise2  exercise3a  exercise3b
```

[7] This will put the text of a particular legally-binding code license in the main folder of our repository. See choosealicense.com for tips as to which license is most suitable for you.

We now simply execute `git init` to initialise the local Git repository.

```
$ git init
Initialized empty Git repository in /home/matthew/documents/teaching/
    part_II/computing/part2-computing-exercises/.git/
```

You should see that the directory `.git` has been made (it will be hidden) which contains all of the objects Git uses to track your files.

```
$ ls .git
branches  config  description  HEAD  hooks  info  objects  refs
```

A call of `git status` will tell us that we are on the master branch, have no commits, and some untracked files. In order to begin tracking files in the `exercise1/` folder, we "stage" them for comitting using `git add`:

```
$ git add exercise1
$ git status
$ git status
On branch master

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)

    new file:   exercise1/README.md
    new file:   exercise1/cornu.pdf
    new file:   exercise1/diffraction_D_0.3m.pdf
    new file:   exercise1/diffraction_D_0.5m.pdf
    new file:   exercise1/diffraction_D_1.0m.pdf
    new file:   exercise1/sine_nd_mc.pdf
    new file:   exercise1/supp_task2.py
    new file:   exercise1/task1.py
    new file:   exercise1/task2.py
```

The files are now ready to be commited, and a simple `git commit` will bring up the editor for us to enter our commit message.

> Notice how only the core source files/outputs are included in the repository. You don't want to add "cruft" to your repository, such as `__pycache__/` folders or `*.pyc` files. To ignore these automatically, you can create a file called `.gitignore` that contains the above strings on separate lines to ignore them forever.

We should now tell our local Git repository where it's remote server is. As stated in the GitHub instructions, we do this using the following command:

```
$ git remote add origin https://github.com/ml-evs/part2-computing-
    exercises.git
```

Finally, we can now push our first commit to GitHub's servers.

```
$ git push -u origin master
```

If you want to investigate GitHub's interface for e.g. listing commits, providing diffs, creating issues and making pull requests, feel free to look at the repository for this tutorial, https://github.com/ml-evs/part2-computing-git-tutorial. If you spot any mistakes or typos, you could even fork the repository and make a pull request!

More advanced usage

Coming soon…?

Branching, merging and collaboration

Test-driven development

Appendices

Basic subcommand cheatsheet

`clone` Make your own local copy of a repository.

```
git clone https://github.com/ml-evs/part2-computing-git-tutorial.
```

`init` Make a new empty repository in the current directory (i.e. sets up `.git` folder).

```
git init
```

`status` Tell me which (if any) tracked files have been changed in my current working directory, since the last commit.

```
git status
```

`add` Register the current state of this file with Git, without comitting it (yet).

```
git add version_control.md
```

`commit` Take a snapshot of the current changes, and give that snapshot a descriptive message. If called without a filename as an argument, the commit will include all changes that have been *staged* (call `git status` to check this). The flag `-a` will commit *all* changes to tracked files in the repo. This command will open your editor to write the message (determined by environment variable `EDITOR`); alternatively, the message can be provided at the command line with the `-m` flag (see below).

```
git commit version_control.md -m 'Added "commit" section to the subcommand
 cheatsheet'
```

`diff` Show the uncommited (well, strictly unstaged) changes to all files in the repository, or for a particular file if requested, as a "diff" (i.e. the difference) between the current state and previously commited state (by default, can alternatively view changes between any two previous commits).

```
git diff version_control.md
```

`push` Uploads the history of the local repository to a pre-configured remote server (see example 1), after first verifying that there are no clashes (i.e. that the server does not possess extra commits to the same branch as being pushed).

```
git push
```

`pull` Download any "new history" on the remote server for this repository, making sure that there are no clashes with local changes.

```
git pull
```

`checkout` If called on a file that has been modified, delete those changes and rever the state of the file back to the last commit (the `HEAD` state). Also used when branching (see `git help checkout` for more).

```
git checkout version_control.md
```

`log` Prints the list of historical commit messages and their hashes, either those that applied to a particular file, or if called without argument, the entire repository.

```
git log version_control.md
```

`mv` Does the same as Linux's `mv`, but keeps tracking the file in Git. Otherwise, Git treats the old file as deleted, and the new filename as a brand new file with no history.

```
git mv old_name.txt new_name.txt
```

`rm` Remove a file from disk, and from any further Git tracking (the deletion of the file will need to be commited). The history of the file will remain under version control, as well as all previous versions.

```
git rm old_file.txt
```

Omissions (non-exhaustive) Here are some extra Git subcommands that you will eventually find a use for in more complex projects.

- `branch`
- `merge`

- `reset`
- `revert`
- `reflog`
- `bisect`
- `rebase`
- `tag`

Useful links

- The Git website has lots of useful resources for learning Git in more detail.
- List of GUIs for Git on the Git website.
- The Git source code is itself hosted on Github.
- Atlassian (owners of BitBucket) provide a more thorough cheatsheet of git commands
- GitHub student developer pack, BitBucket Education for free stuff.
- For open source projects, how to choose a software license.
- The Journal of Open Source Software (JOSS) takes submissions as git repositories and uses GitHub's issue tracker for the review process.
- The Sustainble Software Institute is a UK-wide push for improving the quality of research software, along with the Research Software Engineer (RSE) movement to create new jobs titles for those in academia working predominantly on software.