Elliott Phillips

Data Scientist
BSc OR II

Git It?
A comprehensive guide to Git versioning

Last updated: October 19, 2021

1.1 Why Git?

Is it worth learning Git?

TL;DR – yes, it definitely is.

Understanding how to use Git and GitHub effectively is one of the most important skills for the modern developer. This book aims to equip you with the fundamentals of Git with hands-on example. By following along, you'll learn to use Git to manage your software projects and collaborate with other developers.

1.2 Getting started

Git, created by Linus Torvalds in 2005, is a system for keeping track of changes that happen across a set of files. To initialise a new Git repository, open a directory on your local system with an editor like VS Code then run `git init` from the command line. Git repos live in the hidden .git directory and keep track of all the changes that happen to files as you work on a codebase.

Commits are a method of taking snapshots of the current state of your files, and each commit has its own unique ID and is linked to its parent, allowing us to travel back in time to a previous version of our files.

The head represents the most recent commit. If we make some changes and commit them to the repo, the head moves forward but we still have a reference to our previous commits so we can always go back to it.

Where Git really shines is enabling managed collaborative working. Software typically isn't developed linearly – you may have multiple teams working on different features for the same codebase simultaneously. Git makes that possible by branching. Create a branch by running `git branch` and then run `git checkout` to move into that branch. You can now safely work on your feature in this branch without affecting the code or files in the master branch.

Commits made in these branches live in an alternate universe with its own unique history. Eventually, you'll likely want to merge a branch's history with the master's history. Run `git merge` on your feature branch, whose 'tip' now becomes the head of the master branch, or, in other words, our fragmented universe has become one.

2.1 The perfect commit

2.1.1 Committing files

The following commands help you quicken and amend your commits prior to pushing to a remote repository

```
$ git add .
$ git commit -m "Your message"
```

You can skip the git add using the -am flag to automatically add all the files in the cwd

```
$ git commit -am "Your message"
```

If you've committed with a spelling error, update the latest commit message with

```
$ git commit --amend -m "Updated message"
```

or if you've forgotten to add a file to your latest commit:

```
$ git add extra_file.txt
$ git commit --amend --no-edit
```

where the –no-edit flag retains the original commit message

2.1.2 Stash

If you have changes that almost work but they can't really be committed yet because they break everything else or aren't up to par with the code style, or you just don't want your colleagues to see

it yet, git stash will remove the changes from your working directory and save them for later use without committing them to the repo

```
$ git stash
$ git pop
```

You can assign the stash a name to reference it later with git pop when you're ready to add those changes back into your code. Use git stash list to view all aliased stashes, find the stash you can to retrieve followed by git stash apply with the corresponding index to use it

```
$ git stash save coolstuff
$ git stash list
$ git stash apply 0
```

2.1.3 Destroy things

Imagine you have a remote repository on GitHub and a local version on your machine you've been making changes to, but things haven't been going too well and you just want to go back to the original state on the remote repo.

First do a git fetch to grab the latest code in the remote repo, then use reset with the –hard flag to overwrite your local code with the remote code, but be careful, your local changes will be lost forever.

```
$ git fetch origin
$ git reset --hard origin/master
```

But you might still be left with some random untracked files and build artefacts here or there. Use the git clean command to remove those files as well

```
$ git clean -df
```

2.1.4 Checkout

If you recently switch out of a branch and forgot its name, you can use git checkout followed by a dash to go directory back into the previously branch you were working on

```
$ git checkout -
```

2.1.5 Switch

If you find yourself working on a branch and need to move your current untracked changes to another (new or existing) branch, then git switch is the perfect use case for when you just want to test something, but you're not sure it's worth it.

```
$ git switch -c new-branch
```

You can achieve the same effect with git checkout -b new-branch, but checkout does more than just switching branches, so switch was created for specificity.

3.1 Example

Demonstrative materials.