Chapter 1

Git

> git: 'gud' is not a git command. See 'git –help'
>
> Git

1.1 Intro

When you are developing software, keeping track of changes you make can be really helpful in debugging and avoiding or minimizing expensive errors (the most expensive is rolling out a buggy version of a piece of software that costs the company millions every minute it's out in the public). Git was created by Linux Torvalds [1] because the proprietary software they were using for version control stopped support for kernel development after the open source community tried to reverse engineer it (or so the story goes).

Now Git is the industry standard for version control and is the basis for various other version control systems. For this course we don't expect you to know git well, but enough that you can keep your own version control for your projects as you develop them.

1.2 Brief Overview

Version control is really useful in development because as projects get larger and more complicated, it becomes helpful to track changes that are made and be able to change versions quickly (maybe the newest version has bugs so you want to change to a previous version). Additionally, git allows for parallel development so if you are working on one thing, and your colleague another, then you don't have to wait for the other to finish their job before you can work on yours. This speeds up development.

Here we will offer a quick surface level overview of how git functions for version control.

[1] That Linus Torvalds, the one who created the linux kernel

1.2.1 Version Control

Version control is really important to help keep track of changes progress is made. To make this easy, Git will store *snapshots* of your code and then make a Directed Acyclic Graph (DAG) of these snapshots. A snapshot is basically a picture of what your work looks like at some specific time. Then when you put the pictures together in a graph, you can see the changes happen over time (like any scrapbook or photo album). The process of taking a snapshot is called *committing*. Each snapshot has a name so it is very easy to revert changes by telling git to go to a certain snapshot rather than just holding Ctl+Z until things look right.

This is all you really need when working alone (and if you work linearly). Professionally however, you will most likely be working with other people. It is also possible you are like me and work a little bit on a lot of things all at once. Regardless, you need a bit more besides just tracking your changes in these scenarios.

The most important thing to note is that Git is a distributed version control system, that means there is no central storage space for a code base. So when you are taking snapshots, they are only visible to you. You also can't see anyone else's snapshots. Thus, many people can be working concurrently on a project which can speed up development. That being said, there will come a time when you and your friend will need to combine your work to submit a final product. This process of combining work from multiple DAGs is called *merging*. A crude analogy would be that instead of emailing a file back and forth between you and a friend each making changes until a final version is made, you could each write your own file and then magically merge your two files together at the end.

1.2.2 Centralizing Git in the cloud

We said Git does not have a central storage system and this is technically correct, but as you have experienced, services like GitHub or GitLab exist which allow you to host your code in the cloud and have a central code source. These code sources are called *repositories* or *repos*. You can then copy this code to your machine (by *cloning* or *pulling* depending on what exactly you are copying), or copy your code from your machine to the repository (*pushing*). Additionally, for some services like GitHub, you can merge directly on the website rather than locally. This means that besides version control, Git also allows for publishing your code and putting it out there on the internet.

To allow many people to simultaneously work in the same repo but not step on each other toes, git allow has the ability to make *branches* so any changes you make do not accidentally modify the official version of a program as you develop. A branch is just an offshoot of the main branch. If you think about a biological tree, you can think of the official version as the trunk and a branch an offshoot of the trunk. The only difference is that sometimes these branches sometimes circle back around and merge back into the trunk.

You can read more about branching in the documentation on the git website.[2] We will

[2] GitHub documentation

only focus on how to commit, clone, push and pull.

1.3 Local Development

To develop locally, you need to first copy code from a repo onto your local machine. This process is called *cloning*. I would recommend making your own repo and playing around there.

If you are using the SSH protocol you can run the following:

```
git clone git@github.com:CliffBakalian/git-basics.git
```

When you clone a repository, it creates a directory with the same name as the repository. Once you cloned and cd'd into it, I would make a change, the simplest being `touch new_file`.

Once you made a change you can now get ready to take a snapshot to begin tracking changes and version control. To do this you must first tell git what files you want logged as changed. We do this with the add command.

```
git add new_file
```

You can add individual files or directories. This will move these files into what is called the *staging area*. You can see what files changed sine the last snapshot and what files have been added to the staging area with a 'git status'.

Once you have *staged* everything you want to, you can now take your snapshot with the `commit` command.

```
git commit -m "added a new file for testing purposes"
```

The `-m` flag says to use the proceeding string as the commit message or comment. This makes it easier to identify all the snapshots. You always need to give a message. If you don't, git will prompt you for one.

About the Staging Area

The staging area is the place where the snapshot will be taken. For an extended analogy: suppose you are a photographer and you need to set a scene. You may have props made, backgrounds drawn, and people ready to be posed. When you add a file to the staging area, it's like you are placing the prop, person, backdrop into the view or frame of the camera. Now sometimes a prop breaks, or you hire a too many models and you don't end up using them all. In this case, you don't have to add them, even if you did work with them. Whatever you stage however will be in the shot. Once everything is finalized and you are ready to hit the shutter, you can finally `commit` to the picture.

Once you made your commit, you can publish your changes to GitHub for the world to see, (or if you are using a private repo, just copy it to the cloud). We do this with the `push` command.

```
git push
```

Congrats! You just pushed your first commit!

If you used my repo, you won't be able to see the changes immediately because you don't have write access. Instead something called a *pull request* (PR) will be made. This is basically a message that says "Hey, I want to propose these changes". Everyone can see this message and it can be found on the pull request tab on GitHub. If I wanted to, I can say "Those changes make my code better, I will accept them" and then accept the PR. Then your code will be included in the official version that everyone gets when they clone or *pull*.

Now sometimes repos change due to someone else pushing or because a PR was accepted. To get their changes (or to update/resynch your local code with GitHub if you are behind) you can *pull* the changes to your local machine.

```
git pull
```

Now remember that we wanted to use Git for version control. We can easily change our code to any snapshot that we have previously taken. We first do this by looking at the history of all our changes and snapshots:

```
git log
```

Each snapshot has an ID which is a very long string which was generated by git (with hashing). The message you used for each commit is also listed. You can easily determine which version you want to change to based off the commit message (you did use meaningful messages right?). Once you have the ID of the commit you want to go back to you can `revert` to it.

```
git revert <commit_id>
```

This will change your directory to this moment in time and make a new snapshot which will be added to the end of the DAG. If you have any unsaved changes before the revert, you can decide to save the changes by committing or stashing them (`git stash`. I would recommend stashing because the messed up code will still be documented (and you probably want to keep it for future use). Stashing will save the changes to local memory so it won't be logged and is typically either forgotten or overwritten[3].

Reverting will keep a history of what you got rid of which is why I recommend it since it is helpful. However if you need to delete the history, you probably want to do a `reset`. Resetting is dangerous and I would only use in emergencies.

[3] This is actually useful when merging but you don't have to worry about that for this course