Version Control with git

Foreword

If you have not heard of the idea of version control, or use git regularly, you are about to learn one of the most important tools in software development, second to only a text editor. It is so important that I put it right after the introduction session, even before the programming language session. We are going to learn about git, mostly from a contributor perspective. Down the road in this course you will also learn about git from a maintainer perspective.

There are many great guides on the internet about how to use git (This being my favorite guide), and it is not very useful for me to go through every existing git commands. In this session, we are going to learn some of the basic commands which you will use most frequently. In case you want to dig deeper into the git rabbit hole, `git help [topic]` actually provide a great selection of tutorials for you to follow.

By the end of this session, you will know how to use all of the following commands

Command	**Function**
init	Initialize a new git repository
add	Add files to the staging area
rm	Remove files from the staging area
commit	Commit changes to the repository
push	Push changes to a remote repository
pull	Pull changes from a remote repository
fetch	Fetch changes from a remote repository
log	Show the commit history
checkout	Checkout another branch
status	Show the status of the repository
branch	List, create, or delete branches
remote	Manage remote repositories
diff	Show changes between commits
rebase	Reapply commits on top of another base tip
merge	Join two or more development histories together
blame	Show what revision and author last modified each line of a file
reset	Reset current HEAD to the specified state
revert	Revert some existing commits
tag	Actions related to a specific tag in the code base

Version Control with git

Git system

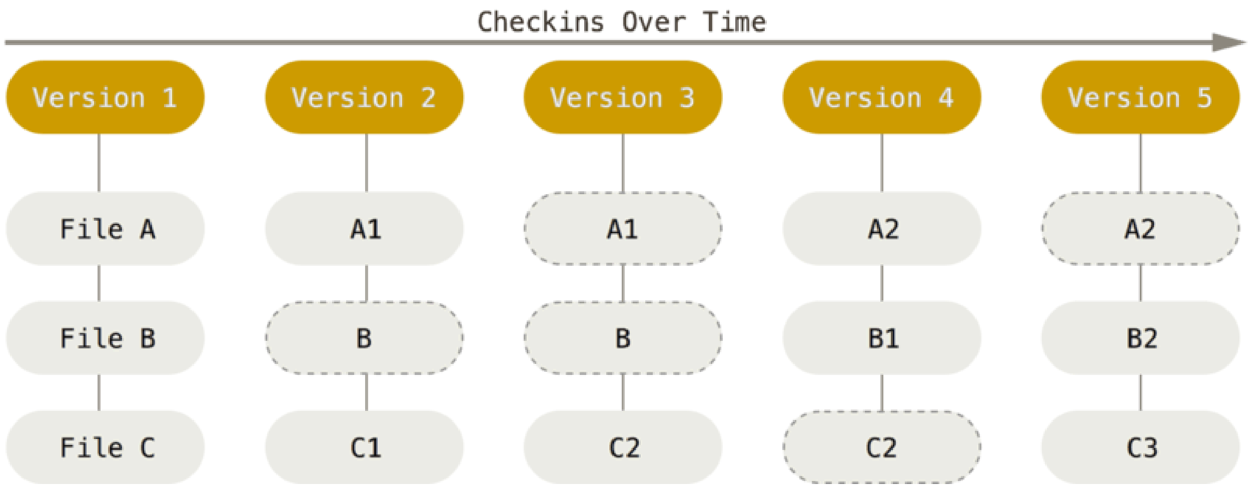

Figure 1: Git versioning system, taken from git

Exercises

Starting a repository

Step 1 - Configuring the committer info

Before we start a repository, let's first configure some info about the committer. You can do this by running the following commands:

```
git config --global user.name "Your Name"
git config --global user.email "Your Email"
```

Then you should be able to see the configuration by running `git config --list`.

Step 2 - Initializing a repository

To start a git repository on your local machine, first create a new directory for your semester project, then navigate to it. For example, on my local machine, it is going to be a series of command like this:

```
mkdir ./semester_project
cd ./semester_project
```

Once in the directory, you can do `ls -a` to see all the files in the directory, which should be completely empty at this point.

Then we will run the command `git init` to initialize a new git repository.

Now if you do `ls -a` again, you will see a new directory called `.git`. This is where git stores all the information about your repository.

Working on code

Step 0 - Check git status

Since your directory do not have any files in it, you can run `git status` to see the status of the repository. You will see that git has detected that the directory is clean.

Version Control with git

Step 1 - Creating a file

Now let's try to move the ReadME.md file you created last week into this directory, and commit the changes with git. Once you move the file into the directory, you can run `git status` to see the status of the repository. You will see that git has detected a new file in the directory.

Step 2 - Adding the file

To add the file to the staging area, you can run `git add ReadME.md`. You can then run `git status` again to see the status of the repository. You will see that git has detected a new file in the staging area.

Step 3 - Committing the file

To make the changes permanent, you can run `git commit -m "Add ReadME.md"`. You can then run `git status` again to see the status of the repository. You will see that git has detected that the directory is clean.

What happened here is you have created a new commit in the repository, which will create a new snapshot of the repository. You can see the history of the repository by running `git log`.

Step 4 - The reason why Git is great for your health

Now we can see the biggest reason why you should always use git: to prevent accidental deletion of files. You have worked so hard on the ReadME.md file, and you accidentally deleted it with `rm ReadME.md`. Normally you would go through the five stages of grief, but with git, your file can be recovered. Let's check the status of the repository with `git status`. You will see that git has detected that the file has been deleted. To get your file back, you can simply run `git restore ReadME.md`. Now you should see the file back in the directory, and the status of the repository is clean.

If you really hate your `ReadME.md` for some reason and you want to nuke it out of existence

Rules of thumb for committing

You should develop the habit of committing your changes frequently. A good rule of thumb is to commit your changes whenever you have made a significant change to the code, or whenever you are about to leave your computer. It doesn't matter whether your code is perfect or not (At least considering you are only working locally), you can always go back to the previous commit if you mess up.

Version Control with git

Branching off

The next important features of git is branching. For a small personal project it might not be very useful, but for any real world applications that you are planning to release, maintain a code base, and keep adding new features, branching is an awesome feature that you should use.

The idea of branching is you can maintain multiple "branches" of the code, which could be different from each other. Say you are maintaining a codebase that is catered to an audience, and their scientific workflow depends on your code, you may not want to put untested changes in the public such that it breaks downstream workflows. Instead of commiting everything to the main branch (assuming that's where the published version of the code based on), we can create different development branches of the code, make our changes and test them there, and only merge the changes to the main branch when we are sure that the changes are good. A schematic of the branching system is shown below.

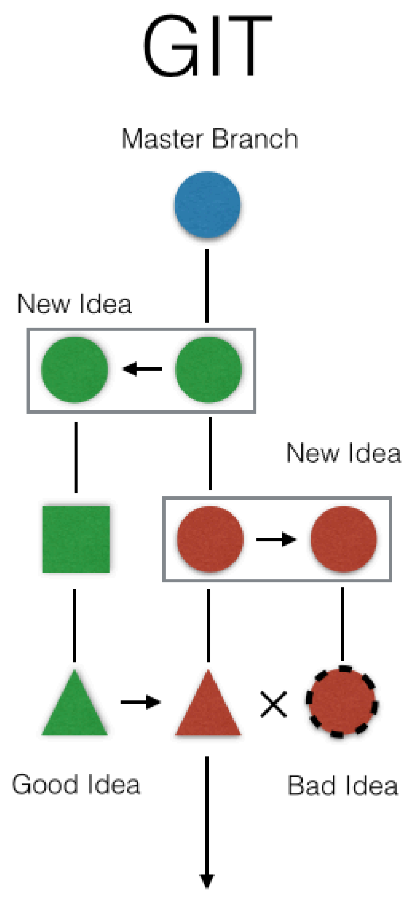

Figure 2: Git branching system, taken from gitbook

In my daily work, there are a lot of boxes my collaborators and I need to check before we merge changes to the main branch, such as testing and code review. These are good practices we will go through in the later part of this course.

Version Control with git

Step 1 - Creating a branch

In order to create a new branch, you can run `git branch [branch_name]`. You can then run `git branch` to see all the branches in the repository. You can then run `git checkout [branch_name]` or `git switch [branch_name]` to switch to the new branch. I also use a one-liner `git checkout -b [branch_name]` to create a new branch and switch to it at the same time. Once you create the branch, you can view the branch you are on by running `git branch`.

If you are running out of idea for naming the branch, let's use development for now.

Step 2 - Making changes on the branch

Now you can make changes to the code on the development branch. Let's add some more text to the ReadME.md file. Once you are done, you can run `git status` to see the status of the repository. You will see that git has detected that the file has been modified. Commit it like you did before.

Step 3 - Merging the branch

The changes you have made in this branch has not propagated back to the main branch you were working on. Assuming most of your collaborators or your community is not working on this sepecfic new branch you were working on, it would be good to push the changes back to the main branch. The way to do this is through merging the branch. First, you have to switch to your "target" branch, which is the branch you want to merge the changes to. In this case, it is the main branch. Then you can run `git merge development` to merge the changes from the development branch to the main branch. Once you have merge the changes, you should be able to see the commit history of the main branch by running `git log`. Once you are satisfied with the changes, you can delete the development branch by running `git branch -d development`.

Rules of thumb for branching

1. **You should not be afraid of branching out.** There is basically no overhead or price you have to pay for branching, so there is no reason not to branch off when you want to work on something significant. Branching is a powerful feature to keep different development tasks separated. Sometimes you want to fix a bug, sometimes you want to inroduce a new feature. The rules of thumb here is whenever you have **independent** tasks, you should branch off.
2. **You can branch off from another branch too.** When there is a significant change you want to make to the codebase, say you want to refactor the core of the code, you can first branch off the main branch such that only tested and reviewed changes are merged to the main branch. And then, you can branch off from the development branch into smaller feature branches to work on the smaller features. Once you are done with the smaller features, instead of merging back to the main branch directly, you can merge the changes into the development branch for further testing and review.

Online collaboration

Now you have tried to work with `git` locally, it is time to go online. `git` can prevent you from accidental `rm -rf`, but if the only copy of the code is on your computer, it will not be able to save your data if the computer itself is compromised. The most popular online platform for `git` is `GitHub` If you do not have an account on `GitHub` yet, go to this link to create one: https://github.com/join. Once you have an account, you are ready to push your code online. * In preparing this lecture note, somehow one of my local filesystem actually get corrupted and I couldn't perform any update on my files. Thanks to `GitHub`, I just nuked the directory and cloned the repository back to my local machine. Nothing is lost.

Version Control with git

Step 1 - Creating a repository on GitHub

Go to the `GitHub` website and log in. You should see a `+` sign on the top right corner of the page. Click on it and select `New repository`. You will be asked to fill in some information about the repository, such as the name of the repository, whether it is public or private, and whether you want to add a README file. Let's keep the repository public from the get-go, and skip adding the README and other files.

Step 2 - Pushing the code to GitHub

Once you have created the repository, you will see a page with a URL on the top, choose the https URL instead of the ssh URL and copy it (If you choose private repositories, this will not work and you will have to use the ssh URL). Run the following command in your terminal to add the remote repository to your local repository: `git remote add origin [URL]`. You can then run `git remote -v` to see the remote repository you have added, which should have the URL you just copied. You can then run `git push -u origin main` to push the code to the remote repository. If everything goes well, you should be able to see the code on the GitHub page.

Step 3 - Pulling the code from GitHub

Now `GitHub` is not only a one-way cloud drive for you to store your code, but an online platform such that a community can work on the same code together. This means you will have to pull the changes from `GitHub` to your local machine from time to time. Let's modified the ReadME.md file on the `GitHub` page using its online editor, then commit the changes to the main branch. It doeesn't have to be a significant change, just add a line of text to the file. Once you have committed the changes, you can run `git pull` to pull the changes from the remote repository to your local machine. You should see the changes you made on the `GitHub` page reflected in the ReadME.md file on your local machine. Another option as opposed to pulling is to run `git fetch` to fetch the changes from the remote repository, and then run `git merge origin/main` to merge the changes to your local repository.

Rules of thumb for collaboration

1. **You should always pull before you push.** This is to make sure that you are not overwriting someone else's changes. If you push your changes without pulling the changes from the remote repository, you might overwrite someone else's changes, which is not a good practice.
2. **You should always pull before you start working.** This is to make sure that you are working on the latest version of the code. If you start working on an old version of the code, you might have to resolve a lot of conflicts when you try to push your changes to the remote repository.
3. **Read the CONTRIBUTING.md file.** This is a file that is usually included in the repository to tell you how to contribute to the repository. It usually tells you how to format your commit messages, how to create a pull request, and how to get your changes reviewed.

Dealing with conflicts

Conflicts can arise when the same files are worked on by multiple people at the same time. This is a common problem and can be annoying from time to time, espeically when it happens close to a deadline. If you adhere to the best practices we mentioned above, there should be minimal conflicts. But in case it does happen, here is how you can resolve it.

Version Control with git

Step 1 - Creating a conflict

Let's create a conflict by modifying the ReadME.md file on the `GitHub` page using its online editor, and commit the changes to the main branch. Then modify the ReadME.md file on your local machine (Make sure you are changing the same content in a different way, e.g. modified the same line but with different input), and commit the changes to the main branch. Once you have committed the changes, you can run `git pull` to pull the changes from the remote repository to your local machine. You should see a message saying that there is a conflict in the ReadME.md file.

Step 2 - Resolving the conflict

To resolve the conflict, you can open the ReadME.md file in your text editor. You will see something like this:

```
<<<<<<< HEAD
This is the content you have on your local machine
=======
This is the content you have on the remote repository
>>>>>>> [commit hash]
```

You can then decide which content you want to keep, or you can keep both contents. Just go ahead and edit your file and make sure those lines includes <<< and >>> are gone. Once you have resolved the conflict, you can run `git add ReadME.md` to add the file to the staging area, and then run `git commit -m "Resolve conflict"` to commit the changes. You can then run `git push` to push the changes to the remote repository.

Rules of thumb for conflicts

1. **Check with the person who made the conflicting changes.** If you are not sure which content to keep, you can always ask the person who made the conflicting changes. They might have a good reason for making the changes, and they might be able to help you resolve the conflict. You can use `git blame ReadME.md` to see who made the conflicting changes.
2. **Don't panic.** Conflicts are a normal part of working with git, and they are not the end of the world. You can always resolve them by following the steps above.

Nuke buttons for conflicts - `git reset` and `git revert`

Sometimes you may regret making a particular changes (or regret you accepted someone's changes) and you want to roll them back, `git revert` and `git reset` comes in handy. If someone did something to your repo that you decide you want to roll back the changes, you should use `git revert`. This will create a new commit that undoes the changes made in the previous commit, meaning if you check `git log`, you should be able to see both the unwanted the commits and the commits that you use to undo those changes. On the other hand, if you have some commits you want to get rid of entirely, say some local commits which you have not send it to the public repo, you can use `git reset --hard [commit hash]`. This will reset the HEAD to the commit you specified, and all the commits after that will be gone. In general, you should use `git revert` over a hard reset. Having the commit history available is better for decision making down the road

Working with public repositories

Step 1 - Forking a repository

Add your project to the class's awesome list