17-214 Recitation 1: Git

What is Git and why do I need it?

Git is a version control system, which is a fancy way to say that you can take *named* snapshots (i.e. make saves) of your code base with it throughout development, and you can easily revert back to any such saves should something go wrong.

Sooner or later you will write code that breaks, and you'd wish to "undo" what you wrote. Without git, you'll have to figure out what you changed and revert them manually, which is time-costly and error-prone. With git, if you made a save when your code was working, you can revert back to that in one command. You also document your work incrementally for others.

Git also has collaboration related features such as branching.

Some Important Git Concepts

Git Repository: A git repository is a database of files and their history. On a local computer it will be stored in a .git directory. A repository on your computer is called a *local repository*, and a (typically shared) repository stored somewhere online (e.g. on GitHub) is called a *remote repository*.

Git Commit: A git commit is a snapshot / save of your git repository. You make a save of your code base by making a git commit. A commit has an ID and a description. It follows another commit. You can later revert back to it.

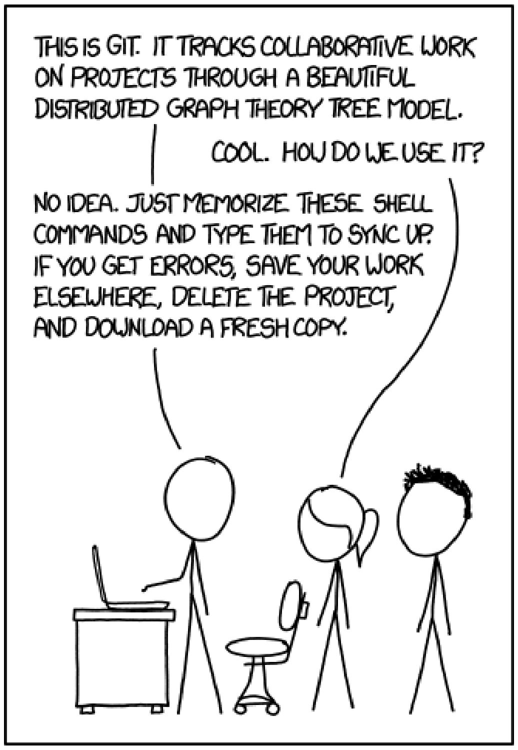
Working directory: Code from a git repository can be *checked out* in a directory. This will copy all files from the (typically last) commit from the git database to your directory where you can edit them.

Staging Area: Before you can make a git commit, you need to specify the files you want to include in your commit. In git you do this by *adding* them to the staging area, where you can double check that you are saving what you want. The *commit* command will create a commit of all files staged, but not of other changed files in your working directory.

Remote Repository: A remote repository is like a folder on Google Drive. Just like how you could backup your local files online to Google Drive, you can backup your local Git Repository online as a remote repository. This also enables collaboration: remote repositories are usually shared, when your collaborators *push* (upload) new changes to a shared remote repository, you can *fetch* (download) them to your computer.

Branching: Git branching allows you to create new development paths for your code. Each path is called a branch. Changes on one branch are independent from changes made on a different branch. Later, if desired, changes from different branches can be merged together. People commonly use branching when implementing new features. They create a new "development branch" and implement new code on it, keeping the default "master branch" clean. If the new code works out, they merge changes from the development branch into the master branch. If it doesn't work out, they just discard the development branch. This ensures that the master branch always contains working code.

Some Important Git Commands


THIS IS GIT. IT TRACKS COLLABORATIVE WORK ON PROJECTS THROUGH A BEAUTIFUL DISTRIBUTED GRAPH THEORY TREE MODEL.
COOL. HOW DO WE USE IT?
NO IDEA. JUST MEMORIZE THESE SHELL COMMANDS AND TYPE THEM TO SYNC UP. IF YOU GET ERRORS, SAVE YOUR WORK ELSEWHERE, DELETE THE PROJECT, AND DOWNLOAD A FRESH COPY.

Git is not particularly intuitive to use, but understanding the difference between local repository, working directory, staging area, and remote directory will help to make sense.

There are a couple of visual frontends for Git and IDE integrations to explore, but let's start with the command line interface for now.

Note: replace <NAME> with the specific name in your situation.

$ git clone <remote repository link> (Clone, i.e. download in entirety a git repository from a remote location and check it out in a local working directory)

$ git pull (Pull, i.e. download, any new changes from the remote repository not previously downloaded and merge them into your local working directory)

$ git add <filename1> <filename2>... (Add files to the staging area)

$ git status (Shows which files have been changed and which files are on the staging area)

$ git commit -m "<some message>" (Make a commit saving everything on the staging area. The save is tagged with a message <some message>)

$ git log (List all the commits you've made)

$ git push (Push, i.e. upload, any new commits you made locally to the remote repository)

$ git branch "<branch name>" (Create a new branch named <branch name>)

$ git checkout <branch name> (Check out files from the git repository to your local working directory of the branch named <branch name>)

What is GitHub and why do I need it?

GitHub is to git repositories as Google Drive is to regular folders and documents. It’s just one of the many services where you can store your code online as a remote repository. Just as you can collaborate on shared files on Google Drive, you can collaborate through shared repositories stored on GitHub.

We use a remote repository on GitHub to share code with you, and you submit your work by pushing your commits to this shared repository.

Exercise

In this exercise you'll practice a typical git workflow: cloning a remote repository, making some code changes, making git commits for your changes and pushing them to the remote repository.

Don't have Git yet?

No worries, follow the download instructions on https://git-scm.com/downloads

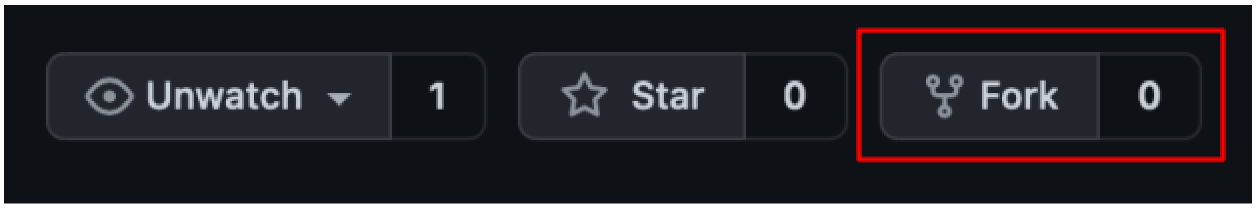
Don't have a GitHub account?

No problem, create one on https://github.com

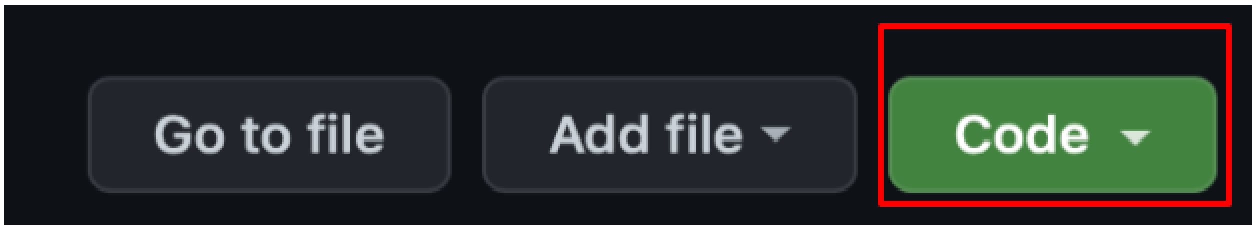
Let's get started

Go to the remote repository at https://github.com/SeanEZhou/GitPractice

Click on the fork button on the upper right corner. This creates a copy of the remote repository for yourself. We're only doing this to avoid everyone pushing to the same remote repository and creating conflicts.



Next, navigate to your own forked remote repository, and then click on the green "code" button to copy the URL of this remote repository so that you can clone it.



Now open your terminal and navigate to a location you want to clone the remote repository to (e.g. desktop), and run the following command

$ git clone <The URL of the remote repository>

This should download the entire repository to your computer. Navigate into the repository. Aside from this handout, the repository contains two files, **DiningPlace.txt** and **CampusActivities.txt**, each with two questions for you to answer. We'll answer the first question in each file and make a commit. Then we'll answer the second question in each file and make another commit.

Take some time to answer the first question in each file. Answer however you like, we're just simulting a code change here.

Now that we've written some code, let's make a commit to save our progress. Run

$ git status

to see which files have been changed. Normally you should see two changed files. Add them to the staging area by running

$ git add <changed file path>

Now run

$ git status

again to double check if the file you want to save has been added to the staging area. If so, we can now make a commit to save it by running

$ git commit -m "answered first question"

COMMENT	DATE
CREATED MAIN LOOP & TIMING CONTROL	14 HOURS AGO
ENABLED CONFIG FILE PARSING	9 HOURS AGO
MISC BUGFIXES	5 HOURS AGO
CODE ADDITIONS/EDITS	4 HOURS AGO
MORE CODE	4 HOURS AGO
HERE HAVE CODE	4 HOURS AGO
AAAAAAAA	3 HOURS AGO
ADKFJSLKDFJSDKLFJ	3 HOURS AGO
MY HANDS ARE TYPING WORDS	2 HOURS AGO
HAAAAAAAANDS	2 HOURS AGO

AS A PROJECT DRAGS ON, MY GIT COMMIT MESSAGES GET LESS AND LESS INFORMATIVE.

This would save your progress in a commit with the message tag "answered first question". Can you think of better commit messages? Generally, what makes a good commit message? You can see the commit you just made by running:

$ git log

Now let's move on to answering the second question in each file.

Now that we've made more progress since the last save, let's save one more time by running the same set of commands as before:

See if you can do it by yourself this time.

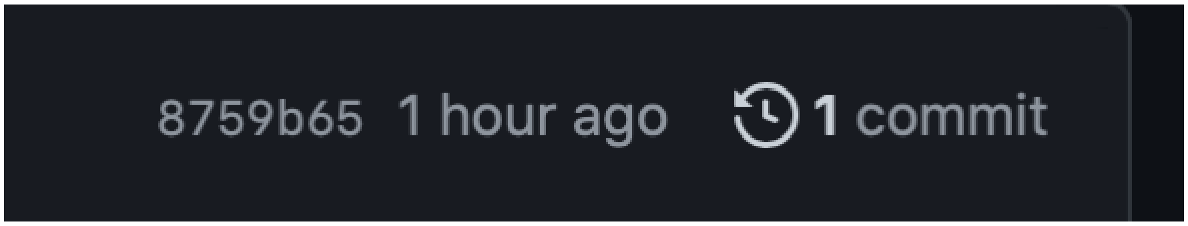
Assuming your second commit is successful, let's run

$ git log

again to check the commits you've made.

You should now see both of the commits you made, the most recent one at the top. Now, it would be possible to revert to an earlier commit. Let's just say, for example, something went terribly wrong when answering your second question (e.g. you accidentally deleted a file), it's possible to revert to your first commit, and your repository will return to the stage when you just finished answering the first question. It's even possible to revert to the "initial commit" which will return your repository back to when you just cloned it.

Now go back to your browser and checkout your remote repository. Click on "commit"



How many commits are there? Does your remote repository contain the commits you just made on your computer? Why not?

Of course, they wouldn't show up yet because the local commits are not yet pushed to the remote repository. To do so, run

$ git push

Now, refresh your browser and see what happens.

Congrats! You have finished a typical git workflow. Reflect on what you did with the different commands.

Finishing early? Try branching (Optional)

Run the following command with a name for your branch

$ git branch "<branch name>"

This creates a new branch that you can see by running

```
$ git branch
```

Notice how you are on the default “master” branch? Switch to the branch you created by running

```
$ git checkout <branch name>
```

Now that you’re on a different branch, make some code changes and save them with commits. Run git log to see your commits.

You can also combine the previous two commands into one as follows:

```
$ git branch -b <branch name>
```

which just tells git to create a new branch and switch to that.

Try to push the new branch to the remote repository by running **git push**. Does it work? Can you fix it based on the error message? If it’s fixed, open your browser and check the new branch on your remote repository.

Now navigate back to the master branch and run git log again. Does it show the commits you made on another branch? How is this related to our discussion about branching earlier?

Good Git Practices

- Make clean, single purpose commits.
- Leave meaningful but concise commit messages.
- Commit early, commit often
- Don’t alter published history
- Don’t commit generated files

Discussion: Why and how do we follow these practices?

Good practices help you and other engineers understand your development process.

Final question (for group discussion, if time left)

Why would you use Git, if you could just use Time Machine or Dropbox to constantly backup everything?