**Git/GitHub/RStudio presentation outline**

What is Git?

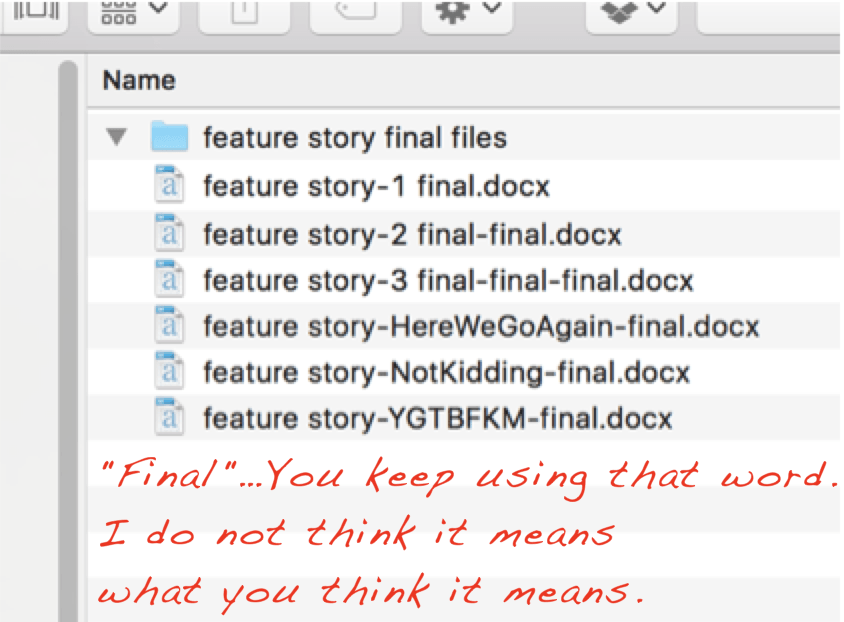
* A version control system (VCS)

Why use VCS?

* Changes are tracked
* Changes are linked to the person who made them
* Changes are traceable and reversible
* There’s only one truly final version of the project
  + No more “report v1”, “report v2”, “report v3 final”, “report v4”
  + No more keeping track of file versions that are getting emailed back and forth
* Can sync to cloud (GitHub)
* Makes it easier to collaborate via pull requests and merges
* Complete history of all changes in a project, can revert to previous versions as needed

A screenshot of a cell phone

Description automatically generated



Upsides of Git (versus other VCSs)

* Easy integration with GitHub
  + It is a distributed VCS: file histories can be saved both locally and on GitHub

What is GitHub?

* A FREE website that allows you to store code, documents, even some data on the cloud
* It is more than cloud storage; it is cloud storage with version control via Git
* Launched October 2007 (<https://github.com/about>)
* Used by over 50 million people to hosts over 100 million project repositories (<https://github.com/about>)
* Used by many major companies, including Airbnb, Facebook, Netflix, reddit, Lyft (<https://stackshare.io/github>)
* Includes major software projects, like the [Linux operating system](https://github.com/torvalds/linux), [Facebook’s react JavaScript library](https://github.com/facebook/react), the [TensorFlow machine learning framework](https://github.com/tensorflow/tensorflow), [Microsoft’s Visual Studio](https://github.com/microsoft/vscode), [Window’s Terminal](https://github.com/microsoft/terminal), and more
  + Fun fact: Git was actually created by the founder of Linux, Linus Torvalds, in 2005 to help manage the development of the Linux kernel

**Major features of GitHub**

**Pull requests**: allow you to collaborate with other projects/repos

In the picture below:

1. A group of people are working on a project
2. Someone wants to make some changes to the project
   1. This person can fork (AKA copy) the original repo and
   2. Make changes to the project
3. That person will then commit the changes and push them to his/her GitHub repo
4. That person will then ask the group of people in (1) to review the changes and merge them into the original repo via a **pull request**
5. If the changes are approved, the pull request will be accepted and the changes will automatically be merged into the project

A screenshot of a cell phone

Description automatically generated

**4**

**3**

**b**

**a**

**5**

**2**

**1**

* **Issue tracking**:
  + Mostly for long term projects
  + Users (or others) can submit problems with the project
  + Owners can organize and track the issues
* **Wikis**: documentation, usage guides

**Installing and Configuring Git**

Create a GitHub account: [github.com](http://github.com/)

Download Git: <https://git-scm.com/downloads>

Download Meld (covered later): [meldmerge.org](https://meldmerge.org/)

After installing:

* On Mac, you’ll work from Terminal
* On Windows, you’ll work from Git Bash
  + Click the start button then search for Git Bash
* I will refer to both as Bash

Open your Bash program

Confirm Git is installed:

git --version

Tell Git who you are:

git config --global user.email "yourGitHub@email.com"

git config --global user.name "yourGitHubusername"

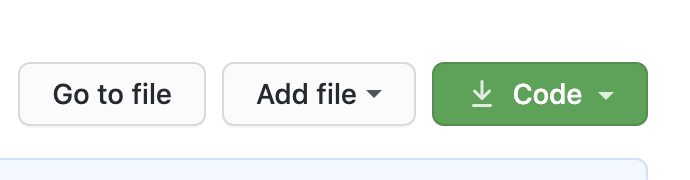
Configure Meld:

* git config --global diff.tool meld (note the period)
* For Windows only: git config --global difftool.meld.path = ‘C:/Program Files (x86)/Meld/meld/Meld.exe’ (no period)
* git config --global difftool.prompt false (no period)
* git config --global merge.tool meld (note the period)
* For Windows only: git config --global mergetool.meld.path = ‘C:/Program Files (x86)/Meld/meld/Meld.exe’ (no period)
* git config --global mergetool.keepBackup false (no period)

Confirm with:

git config --list

**Setting up Git(Hub) for a project**

1. Create new GitHub repo
   1. Go over public/private option and README file
   2. Cover gitignore and license briefly, more later
2. Open new repo, edit README online
   1. Show version history
3. Sync GitHub repo to local machine
   1. Click download code button: 
   2. Get the URL
4. In Git Bash, navigate to the where you want the project to be
   1. cd <path to directory>
5. In Git Bash, type:
   1. git clone <paste URL here>
6. Confirm the repo was cloned by typing ls to list files
   1. Should see the README file

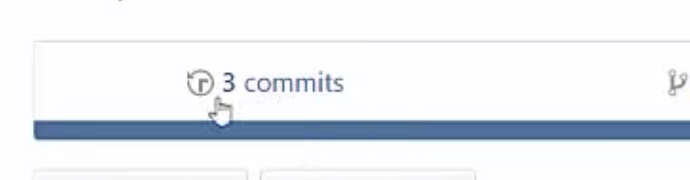
**Using Your First GitHub Repo**

1. Make a new analysis script file.
2. Save it in the folder for the repo.
3. In bash: git status
   1. Will show the new file in red: file has been created but needs to be staged

Whenever you add something to a git project, you’ll need to add it to something called the staging area. The staging area is just a virtual space where you can hold files until you’re ready to commit all of them them/push all of them to your version control system. The staging area is also sometimes called the index.

That bring us to another important term, commits. A commit is simply a snapshot of your project at a specific point in time. The most recent commit is called the HEAD. Every commit has a unique commit ID, sometimes called the SHA1 hash.

1. In bash: git add <file to be added>
   1. This adds it to the staging area
2. In bash: git status
   1. Will show the new file in green: has been staged but not committed
3. In bash: git commit -m ‘add hello world file’
   1. Takes files from staging area and *commits them locally*
   2. The argument -m is telling it the message you want to attach to the commit
4. To confirm the commit was successful, look at the log: git log
   1. It will show the commit ID, the author, the date, and the commit message
5. To sync the change to GitHub, run git push
   1. Confirm by looking at GitHub repo (will likely need to refresh page)

GitHub has more user-friendly commit log: 

1. Make some changes to the analysis script
2. In bash: git status
   1. Will show the file in red: changes haven’t been added to staging area
3. In bash: git add <file to be added>
   1. This adds it to the staging area
4. In bash: git commit -m ‘added new feature to script’
5. In bash: git pull to make sure you have the latest version (mainly important when working with others on the same project)
6. In bash: git push to send your changes to GitHub

To see differences between current and past commits:

* In bash: git diff <old commit ID>
  + Can also run git difftool to see the differences using Meld.
    - Older code will be on the left, new code on the right
* From GitHub commit log: click the commit ID to see differences

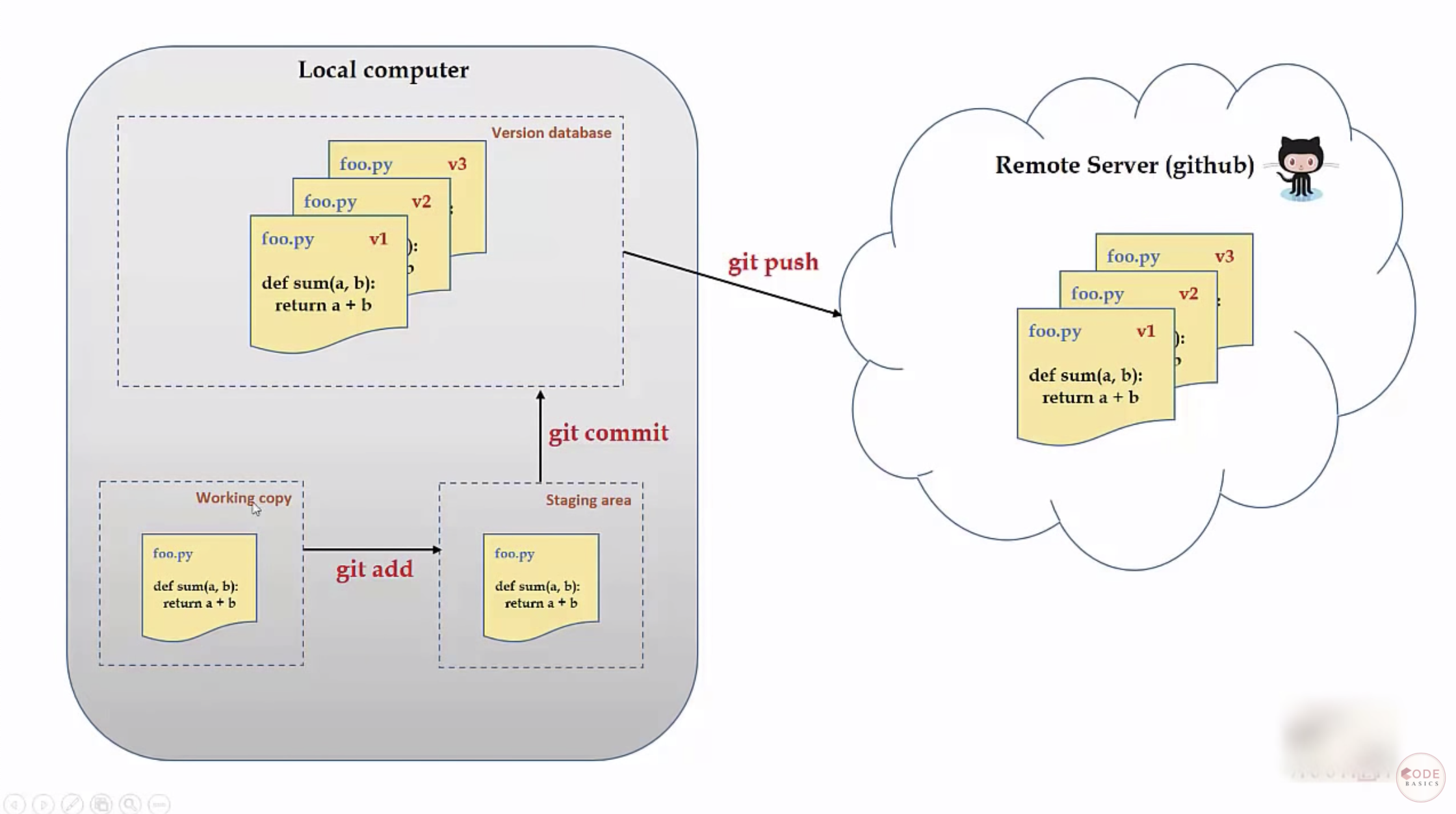
To see differences between two specific commits:

* In bash: git diff <older commit ID> <newer commit ID>
* Or, in bash: git diff HEAD~1 HEAD~2 where the number after the tilde (~) indicates how many commits back to go. Eg, HEAD~1 is one commit before HEAD and HEAD~2 is two commits ago.
* No way to do this with GitHub

To see difference between GitHub version and local files: git diff origin/master

* origin/master is the version on GitHub

To see difference between HEAD (latest local commit) and local files: git diff



Source: CodeBasics

**Undoing/Reverting/Resetting Changes**

Undoing uncommitted changes

* Make a change to the code, save the change
* Bash: git checkout -- <file>
* Works when undo won’t (e.g., when you’ve saved a file then closed it)
* What about when multiple files have changed?
  + Bash: git checkout -- . will undo all changes to all files in the directory

Undoing committed changes:

* Make a change, save, add, commit
  + Can combine the add and commit lines by running git commit -am ‘description’ This will add and commit **all changes** in the directory, not just changes to a specific file
* Can’t run git checkout command because the changes have been committed
* Instead, run git revert
* git revert requires the commit ID of *the commit you want to undo*, so run git log to get the git ID
* Then run git revert <commit ID>
* Type :q to close the interface
* Run git log to see the revert was successful

Going back to a previous commit

* Run git log to get the commit ID that you want to reset back to
* Run git reset --hard <commit ID> : **uncommit + unstage** + ***delete*** changes
* THIS CAN BE DANGEROUS: it will delete all the commits between your current commit and the one you reset back to
* Run git reset --soft <commit ID> : **uncommit** changes, changes are left staged
  + Use if, for example, what you just committed is incomplete, or you misspelled your commit message, or both. This will undo the commit but keep any changes in the staging area
* Run git reset --mixed <commit ID> : **uncommit + unstage changes**, changes are left in working tree
  + Use if, for example, you want to remove changes from both the commit log and the staging area

A screenshot of a cell phone

Description automatically generated

In other words, --soft is discarding last commit, --mixed is discarding last commit and add, --hard is discarding last commit, add, and any changes you made that are the same with git checkout HEAD

**Using Branches**

Branches can be used instead of duplicating a file then saving one as the original and making changes in the other.

To list all branches: git branch

* The branch with a star next to it is the active branch (will also likely be green)

To create a new branch: git branch <branch name>

To move to the new branch: git checkout <branch name>

To create a new branch and move to it at the same time: git checkout -b <branch name>

* The -b flag is for branch

Now that I’m on the new branch, I can make my changes, add, and commit them.

For the branch and any changes in it to be visible on GitHub, you have to run git push while in the new branch. This can be done after the merge too.

Go back to master branch (git checkout master) and you’ll see the changes are gone.

Go back to new branch (git checkout <branch name>) and the changes are back.

To merge the changes back to master:

* Go back to base branch where you want the changes to go: git checkout master
* Merge the new branch into master: git merge <new branch>
* Save your commit message with :q

Importantly, even if you make changes in master branch, when you merge the new branch back into it, the changes to master and the changes in new branch will be tracked and merged.

If you want to delete a branch: git branch -d <branch name>

* The -d flag is for delete
* You cannot delete the branch you are on, you have to switch to another one first

You can also use branches to go back and revise previous commits:

* git checkout -b <new branch name> <ID of prior commit>
* Make your changes here
* git checkout master
* git merge <name of branch created above>

**The .gitignore file**

Sometimes there are files or folders that you don’t want git to track and/or you don’t want to/can’t upload them to GitHub (e.g., datasets that are too big to upload to GitHub, datasets with sensitive information, temporary files created when running analyses or editing documents, etc.).

This is where the .gitignore file comes in.

To create and setup a .gitignore file:

* Run touch .gitignore to create it
* Open the file from your file browser
* List the files/directories you want git to ignore
  + Add comments by putting # before the comment
  + Can use wildcards:
    - Use ? to match any single character except a slash (/)
    - Use \* to match any characters except a slash (/)
    - Place ! before any patterns you want to negate
    - Adding \*.docx would ignore all Word files in the main directory
    - Adding \*\*/\*.docx would ignore all Word files anywhere in the directory
    - Adding abc/\*\* would ignore all files within the abc directory
    - If you have \*\*/\*.docx and !\*\*/My project.docx would ignore all Word files anywhere in the directory except for ones called My project.docx

Changes to your .gitignore file are tracked by git

You can have a .gitignore file for each directory in your project, but this can get confusing if the .gitignore file at the root of the directory conflicts with another .gitignore file in a subfolder.

**Merge conflicts**

Sometimes Git can’t tell what version of a file to keep. For example, two contributors might make different edits the same line of code. How does Git know which version to keep? The answer: you tell it.

* Make changes to a local file, save, close, and commit.
* Make other changes to the same line of the same file on GitHub.
* Run git pull to sync remote changes.
  + This will show there is a merge conflict
* Open the local file to show the merge conflicts

The sections below <<<<<< HEAD and above ========= are your local changes

The sections below ======== and above >>>>>>> SHA-1 are your remote changes

The file can be edited manually by just keeping the parts you want and removing the lines that Git added. But, that’s not very helpful or convenient if there are lots of changes.

Instead, run git mergetool

* Left pane will show local file
* Middle pane shows merged file
* Right pane shows upstream (GitHub) file
* Click arrows from side panes to accept the changes
* Save and close Meld when done
* Commit the changes: git commit -m ‘conflicts resolved and merged’
* Push the changes: git push

**Pull Requests**

**Pull requests** allow you to collaborate with other projects/repos

In the picture below:

1. A group of people are working on a project
2. Someone wants to make some changes to the project
   1. This person can fork (AKA copy) the original repo and
   2. Make changes to the project
3. That person will then commit the changes and push them to his/her GitHub repo
4. That person will then ask the group of people in (1) to review the changes and merge them into the original repo via a **pull request**
5. If the changes are approved, the pull request will be accepted and the changes will automatically be merged into the project

A screenshot of a cell phone

Description automatically generated

**4**

**3**

**b**

**a**

**5**

**2**

**1**

How to do a pull request:

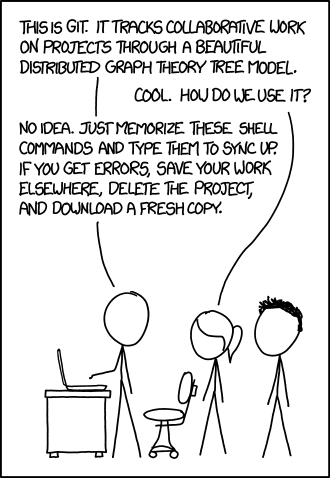
* On GitHub, find the repo you want to modify
* Click the button to fork (AKA clone) the repository
* A screenshot of a cell phone

  Description automatically generated
* You *can* then pull the repo to your local computer to make changes
  + Make the changes locally, then push back to GitHub
* You can also make minor changes directly on GitHub
* When you’re done making changes and want to merge your changes into the original repo, click the pull requests tab on your repo
* A close up of a logo

  Description automatically generated
* Click the button to make a new pull request
* A picture containing drawing

  Description automatically generated
* Fill out the pull request form
* Once you have completed the pull request, the owner of the original repo will be able to review your changes
* The owner can accept the changes and merge the final result into the original repo
* The owner can also comment on specific changes, which will be sent back to the requester
* When the requester makes those changes, the owner will be notified. The owner can then review again and merge the changes into the original repo if desired.

**That was a lot…now the simple(r) way**



https://imgs.xkcd.com/comics/git.png

Almost everything covered so far can be done using only GitHub (no command line tools, no fussy installs).

**The downsides of Git and GitHub**

* Git and GitHub have difficulty understanding some file types
  + MS Office is the big one
  + Version control still works exactly as it should, but the file structures of MS Office documents make human-readable diffs and merges almost impossible
  + Some third-party tools can help (see [Microsoft Office and Git](https://www.ficonsulting.com/filabs/MSOfficeGit), [ReZip](https://github.com/costerwi/rezip), ‘[Microsoft Office, Open Office, PDF diffs with Git](https://gist.github.com/mbrehin/634f46aeacb0a9a6da5e)’, and [Git advanced (text) diff](https://medium.com/@mbrehin/git-advanced-diff-odt-pdf-doc-xls-ppt-25afbf4f1105)), but may make Git run slower
  + MS Office has SharePoint, but it requires checking files in and out
    - Only one person can work on a document at a time
* To really use them to their fullest, you almost have to use the command line
  + You can usually get by pretty well with just the GitHub UI
* If you move or rename files outside of git, git will treat that as if the original file was deleted and the renamed file was added as a new and ***untracked*** file