

Introduction to Git and GitHub

Pin Shuai

CEE 5430/6430 Hydrologic Modeling

2/3/2025

Announcements and reminders

- Assignment#3 is due today (2/3)
- Assignment#4 will be posted later today (due **2/10**)
- Docker image has been updated. Use `docker pull` to update.

Objectives

- Introduction to Git
- Basic Git commands
- Collaboration with Git and GitHub

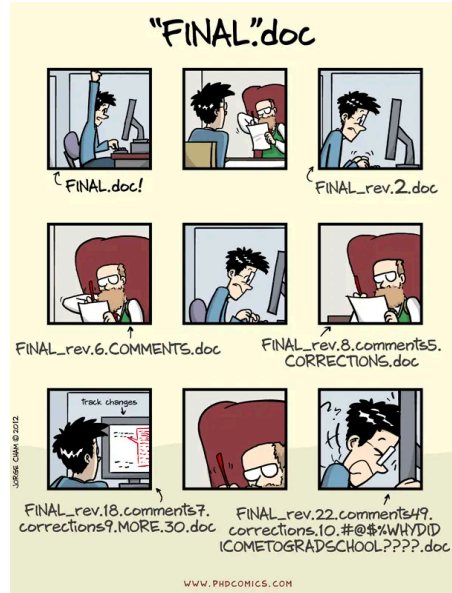
What is Git?

- Version control tool that tracks file change history (like track changes for word but much more sophisticated)
- Popular among software developers
- GitHub is a web platform that is used to store, share, and work on Git repositories. It is not the same thing as Git.
- Other popular platform includes GitLab, BitBucket, SourceForge ...



Why should I use Git?

- Keep track of changes to any of your files
- Works best with plain text and relatively small files (e.g., `.txt`, `.md`, `.py`, etc)



How to install git

- Command line
 - Windows: [Download for Windows](#); recommended to select the “Use Git from the Windows Command Prompt” option
 - Mac: type `git` in Terminal and it will prompt you to install it
- GUI
 - [GitHub Desktop](#) for Win/Mac

Terminology

Distributed version control

Git is a distributed version control system. Everyone working on the project has a copy of the repository. There is no (required) single source of truth or central server.

- Everyone can always edit the same file.

Terminology

Repository

A repository is where the revision history of a project is stored. It's a hidden directory, `.git`, within the directory your project resides in.

- You can reveal the hidden files using `ls -a` in command line
- Normally you don't need to deal with the `.git` directory directly

▲ Warning

Do not delete `.git` directory. If you delete the `.git` directory, you lose your project history (and you are stuck with the current state of your files)! Use GitHub as backup.

Terminology

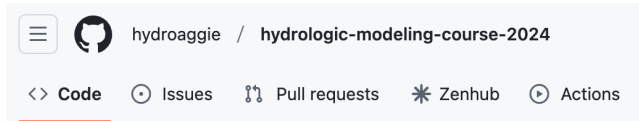
Local vs Remote repositories

- **Local repositories** are repositories that are in your local computer.
- **Remote repositories** are Git repositories that are not local to your computer. They're often hosted on platforms like GitHub or GitLab.
 - e.g., our course repo on GitHub <https://github.com/hydroaggie/hydrologic-modeling-course-2024>
- You will need a remote repository if you want to backup your codes, share codes and collaborate with others.

Terminology

Local vs Remote repositories

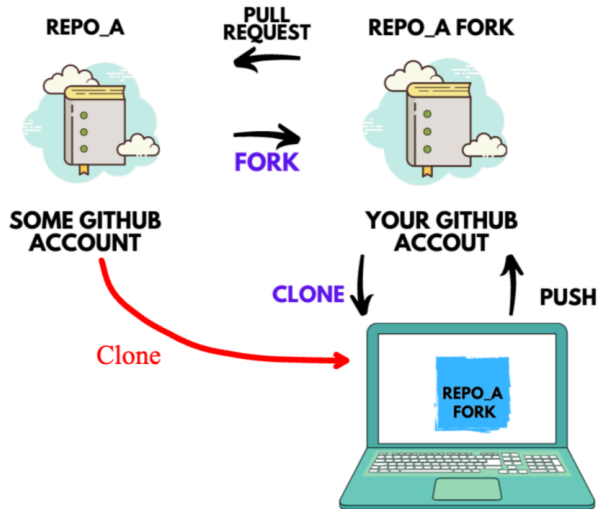
- Remote repository hosts often add features that are not a part of Git itself such as:
 - **Issues**, which allow users to ask questions or report problems
 - **Pull requests**, which allow users to suggest changes. *This is the typical way to contribute to other's repository.*
 - **Continuous integration and continuous delivery (CI/CD)** build and test codes every time the repository is updated
 - **Advanced security features** detect potential security issues like exposed tokens or credentials.



Terminology

Clone vs Fork

- Cloning is you making a local copy of a remote repository (could be yours or someone else's)
- If you Fork first then you have your own version of the repository remotely that you can pull and push changes to. Fork can avoid making changes directly on someone else's repository. **This is the recommended way for contributing to others' repos.**
- More on this later.



Terminology

Commits

Git **does not** keep track of every change you make unless you commit it; it is not the same as having an undo button or a file history. Instead, you **commit** changes as you see fit.

*With Git, every time you commit, or save the state of your project, Git basically takes a picture of what all your files look like at that moment and stores a reference to that **snapshot**. --Pro Git (2nd Edition), Scott Chacon and Ben Straub, 2014*

- Commits have metadata such as the **committer**, the **author** (usually the same as the committer, but not always), the **time**, and a **message** describing the changes.
- Every commit has an associated hash (unique ID, e.g., **c6452be**), which is the name of the commit according to Git. This is how you reference a specific commit in Git.


Terminology

Commits

- Always write good commit messages. A good commit message is concise, descriptive, and informative (not just "update something").

Commits on Jan 25, 2025

add git/github setup instructions **Verified** 8982b39

 pinshuai authored 1 minute ago

Commit message (with arrow pointing to the commit message)

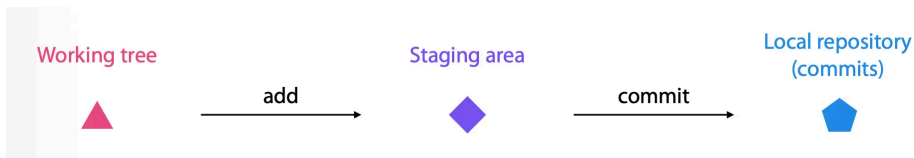
Hash (with arrow pointing to the commit hash)

Terminology

Working tree (directory) and staging area

Before you can make a commit, you need to tell Git which changes you want it to keep track of.

- You first work on files in the **working tree**.
- You then move files with changes you want to commit to the **staging area**.
- Only changes in the staging area are included in commits.



Terminology

Three File Stages

- **Untracked/modified:** the file is new or modified, but is not part of git's version control
- **Staged:** the file has been added to git's version control but changes have not been committed
- **Committed:** the change has been committed (created a new version)



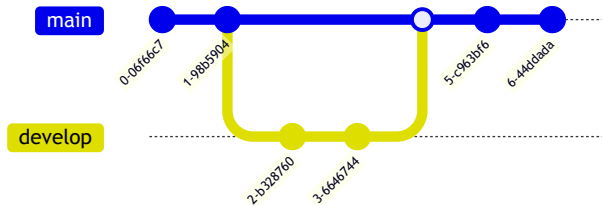
Terminology

Branches

Git branches are effectively a pointer to a snapshot of your changes. They allow you to continue to do work (e.g., adding new features or fixing a bug) without messing with that main branch.

- **Main:** the default branch
- **Develop:** adding new features or fix bugs

You can create as many branches as you like, but make sure you merge them often.



Git commands

To start:

- `git init` Create an empty Git repository or reinitialize an existing one. This will create a `.git` directory.
- `git config` Configure your username, email address, etc

```
git config --global user.name "Your Name"  
git config --global user.email "your.name@your.domain"  
git config --global core.editor nano # or vim if you are  
comfortable
```

- `git branch` show current branch name

Exercise 1

1. Open the Terminal in JupyterLab
2. Create a new directory (e.g., `~/work/myRepo`) and initialize a Git repository in it
3. Use `ls -a` to reveal the hidden `.git` directory
4. Configure your name, email, and editor using `git config`
5. Confirm your configuration using `git config --list`

Git commands

- `git status`: show file status (untracked, modified, committed, etc.)
- `git add FILENAME`: add untracked files to staging area
 - use `git add .` or `git add --all` to add all untracked files
 - use `git rm --cached FILENAME` to remove files from the staging area
- `git commit` commit the staged files to local repository (or version history). This will open a text editor for a commit message
 - Or `git commit -m YOUR_MESSAGE`: commit and specify a message without opening a text editor
- `git log` see commits in the log

Exercise 2

1. Navigate to the newly created git repository, e.g., `cd ~/work/myRepo`
2. Create a readme file `README.md` and add some description in it. Save the file.

```
nano README.md # or vi README.md
```

3. Use `git status` to show file status
4. Use `git add` to add the readme file and use `git status` again to check file status
5. Use `git commit -m "COMMIT_MESSAGE"` to commit and use `git status` again to check file
6. Use `git log` to see your commit history

Git commands

- `git diff` view current unstaged differences
 - `git diff --cached` view staged differences
 - `git diff earlier-commit-hash later-commit-hash` compare differences between two commits
 - `git diff branch-1 branch-2` compare differences between two branches

```
$ git diff 897c121 a3374a9
diff --git a/README.md b/README.md
index ce9dfeb..9d1987e 100644
--- a/README.md
+++ b/README.md
@@ -1,2 @@
  this is a readme file.
+second commit      # this shows the diff b/w commits; plus sign indicates added content
while minus sign indicates removed content
```

Exercise 3

1. Edit the `README.md` in Exercise 2 with some new texts
2. See the changes made using `git diff`
3. Add the file and commit it
4. View commit history with `git log`
5. Use `git diff earlier-commit-hash later-commit-hash` to view the changes made between those two commits

Working with remote repository

Adding a Remote to an Existing Local Repository

- If you initialize your repo locally, you can use `git remote` to configure a remote repo's information.

```
git remote add <remote_name> <remote_repository_URL> # Add the Remote Repository
```

Note, you will need to setup the remote URL first. For example, you can first create a new (empty) repository on GitHub. Then use `git remote add` to configure the local repo. For example,

```
git remote add origin https://github.com/username/repository.git
```

- Verify the remote URL

```
git remote -v
```

Working with remote repository

Copying an existing Git repository

- Clone a repository from GitHub or other platform

```
git clone <remote_repository_URL>
```

- View the remote URL

```
$ git remote -v  
origin https://github.com/hydroaggie/hydrologic-modeling-course-2024.git (fetch)  
origin https://github.com/hydroaggie/hydrologic-modeling-course-2024.git (push)
```

Tip

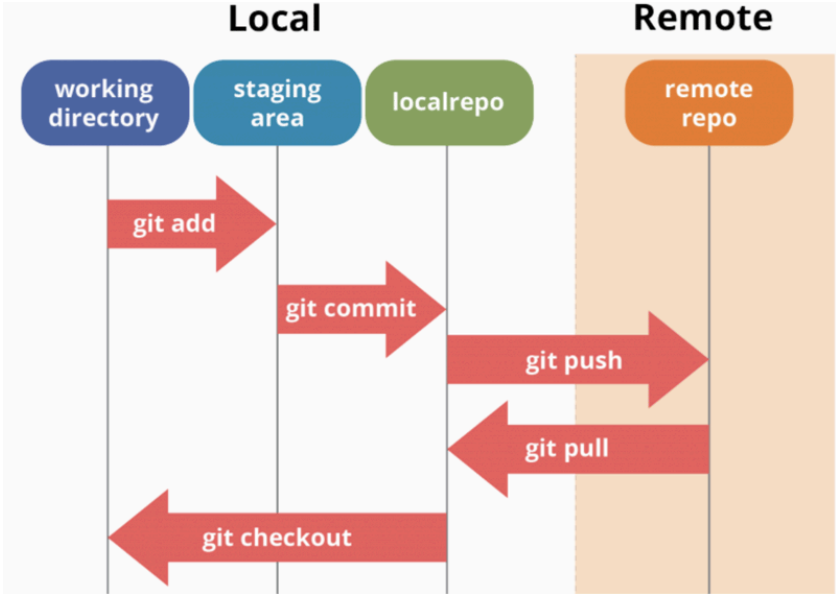
It is easier to create the repository on GitHub first. Then clone it to your local system.

Working with remote repository

Synchronizing a local repository with a remote one

- `git fetch`: fetch latest changes from remote repository into your local repo
- `git pull`: fetch latest changes from remote repository AND **merge** into your local repo
- `git push`: push local changes to remote repository (e.g., GitHub) if you have permissions (**always pull before push!**)
 - `git push -u origin <branch_name>` push your branch to the remote; `-u` short for `--set-upstream`
- `git checkout <branch_name>`: switch branches or checkout a new branch; use `-b` to create the new branch if it does not exist. E.g., `git checkout -b <new_branch>`

Git workflow



Exercise 4

- Create a new repo `myRepo` under your GitHub account (**do not initialize with a README file**).
- Copy the remote url. E.g., `https://github.com/USERNAME/myRepo.git`, and add the remote to your local repo using `git remote add origin <remote_url>`
- Use `git remote -v` to confirm the remote url.
- Use `git push -u origin main` to push local repo to the remote repo on GitHub. You'll need a personal access token for the authentication (see next slide).

```
$ git push -u origin main
Username for 'https://github.com': pinshuai
Password for 'https://pinshuai@github.com':
```

- Go back to your GitHub repo and view the changes you just pushed.

Create a Personal Access Token (PAT)

- GitHub no longer accepts password for authentication. To push and pull from GitHub on CLI, users must create a **personal access token** (PAT) -- similar to a password but is more secure.
- Follow the steps [here](#) to create the PAT on GitHub. Simply go to account **Settings** --> **Developer Settings** --> **Personal access tokens**, click **Fine-grained tokens**--> **Generate new token**. Make sure to add **Read and Write** access to the repository permission!

Create a Personal Access Token (PAT)

Repository permissions 2 Selected

Repository permissions permit access to repositories and related resources.

Actions ⓘ Workflows, workflow runs and artifacts.	Access: No access ▾
Administration ⓘ Repository creation, deletion, settings, teams, and collaborators.	Access: No access ▾
Attestations ⓘ Create and retrieve attestations for a repository.	Access: No access ▾
Code scanning alerts ⓘ View and manage code scanning alerts.	Access: No access ▾
Codespaces ⓘ Create, edit, delete and list Codespaces.	Access: No access ▾
Codespaces lifecycle admin ⓘ Manage the lifecycle of Codespaces, including starting and stopping.	Access: No access ▾
Codespaces metadata ⓘ Access Codespaces metadata including the devcontainers and machine type.	Access: No access ▾
Codespaces secrets ⓘ Restrict Codespaces user secrets modifications to specific repositories.	Access: No access ▾
Commit statuses ⓘ Commit statuses.	Access: No access ▾
Contents ⓘ Repository contents, commits, branches, downloads, releases, and merges.	Access: Read and write ▾

Create a Personal Access Token (PAT)

- Once you have the token, you can authenticate on the command line

Store your token

You may choose to store your token using `git config --global credential.helper store` first so that you don't need to enter the token every time.

```
$ git config --global credential.helper store
$ git push
Username: your_username
Password: your_token # this is the token you get from GitHub
```

Exercise 5

- Go to your `myRepo` on GitHub. Edit the `README.md` file by replacing everything with `Hello, Utah!`. Commit the changes on GitHub.
- Go back to your local repo. Fetch the changes from remote using `git fetch`. Notice that we have not merged the changes into your local README file. Use `cat README.md` to confirm.
- Now try `git pull` to merge your changes. Use `cat README.md` to confirm.

Other useful commands

- `git restore <file>` to discard changes in working directory
 - `git restore --staged <file>` to unstage a file
- `git revert COMMIT-HASH` revert previous commits. This will preserve history (different from recover) and it will add a new commit history
 - use `git revert HEAD` to revert the last commit
- `git reset HEAD^` undo commit
 - `git reset --hard origin/main` reset current repo to remote main branch (**will delete all new files including ignored files/folders!!! Backup all files before doing this!**)

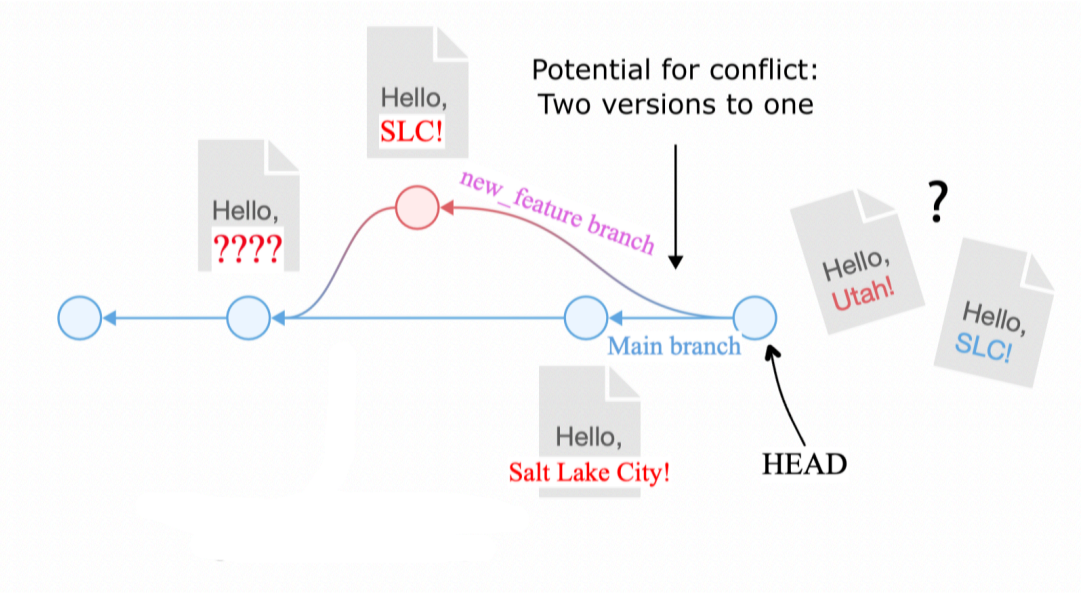
Managing conflicts

- `git merge <new_branch>`: merge a new branch into the existing (e.g., main) branch
- Conflicts can occur in several situations (e.g., during merge) when you're using Git.
- They're a normal part of using Git and not an indication that you've done something wrong (the conflict is between versions, not between people!).
- They help prevent you from losing information or overwriting someone else's work.
- E.g., When merging branches, you can encounter conflicts if **the parent commits have modifications to the same parts of the project**. Git has no way of knowing which (if any) is the authoritative or correct version.
- This may also happen when you are working on two (or more) copies of the same repo on the same branch

Exercise 6

- In your local `myRepo`, checkout a new branch using `git checkout -b new_feature`. Use `git branch` to confirm the current branch (shown with `*` in front)
- Edit the `README.md` by changing the word "Utah" to "SLC". Add and commit the README file with the message "change Utah to SLC".
- Now switch back to the main branch using `git checkout main`
- Edit the `README.md` by changing "Utah" to "Salt Lake City". Commit the file with the message "change Utah to Salt Lake City".
- Use `git diff main new_feature` to see the differences between main and new_feature branch
- Try merging the `new_feature` branch into the `main` branch using `git merge new_feature`
- You will see a Merge Conflict! Why?

Exercise 6



Managing conflicts

- Open the `README.md` file, and you will see something like below:

```
<<<<<<< HEAD
Hello, Salt Lake City!
=====
Hello, SLC!
>>>>>>> new_feature
```

- To fix the conflicts, decide if you want to keep only your branch's changes, keep only the other branch's changes, or make a brand new change, which may incorporate changes from both branches.
- Delete the conflict markers `<<<<<<< HEAD, =====, >>>>>>>` `<branch_name>` and make the changes you want in the final merge.

Exercise 6 cont'd

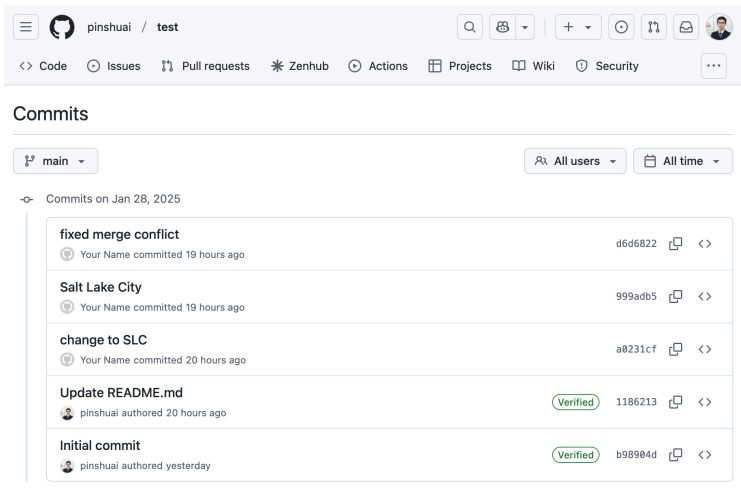
- Fix the conflict by editing the `README.md`. Save the edits.
- Mark the resolution using `git add`. You will see the following message:

```
All conflicts fixed but you are still merging.  
  (use "git commit" to conclude merge)
```

- Commit the changes using `git commit` with the message "`fix merge conflict`"
- Use `git log --graph` to see a graphic overview of how a repository is branched and merged over time.

Exercise 6 cont'd

- Push the commits to your **myRepo** repo on GitHub using **git push**.
- Open your repo on GitHub to verify the changes in the commit history.



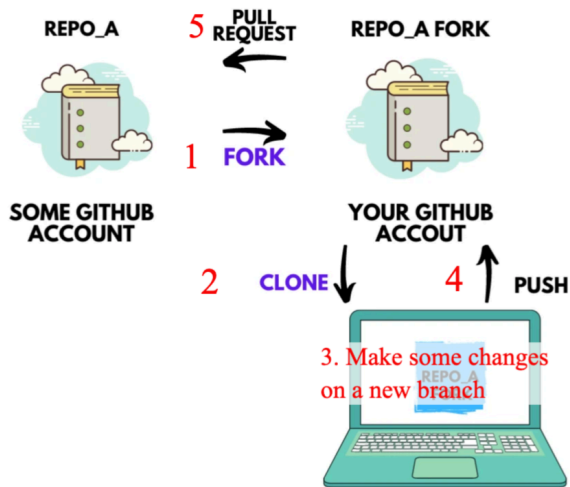
The screenshot shows the GitHub interface for a repository named 'test' by user 'pinshuai'. The 'Commits' section is active, showing a list of commits on the 'main' branch. The commits are ordered from newest to oldest. The most recent commit is 'fixed merge conflict' (d6d6822), followed by 'Salt Lake City' (999adb5), 'change to SLC' (a0231cf), 'Update README.md' (1186213, verified), and 'Initial commit' (b98904d, verified). The interface includes navigation tabs for Code, Issues, Pull requests, Zenhub, Actions, Projects, Wiki, and Security. Search and filter options are also visible.

Commit Message	Author	Commit Hash	Time
fixed merge conflict	Your Name	d6d6822	19 hours ago
Salt Lake City	Your Name	999adb5	19 hours ago
change to SLC	Your Name	a0231cf	20 hours ago
Update README.md	pinshuai	1186213	20 hours ago
Initial commit	pinshuai	b98904d	yesterday

Collaboration through Git and GitHub

If you are a contributor (i.e., no permission to the original repo), a typical workflow looks like this:

1. **Fork** a public git repository to your personal account
2. Open the forked repository on GitHub and **clone** it locally through GUI (e.g, [GitHub Desktop](#)) or terminal (e.g., PowerShell)
3. **Create a new branch** (e.g., `git checkout -b NEW_BRANCH`) and some changes. **This is important to not mess with the main branch.**
4. Add, commit, and **push** the changes to GitHub
5. On GitHub, submit a PR (**pull request**). After review, a PR is approved by the original repository owner



Collaboration through Git and GitHub

If you are a collaborator of a GitHub repo:

1. **Clone** the repo. No need to fork.
2. **Create a new branch** (e.g., `git checkout -b NEW_BRANCH`) and some changes.
3. Add, commit, and **push** the changes to GitHub
4. On GitHub, submit a PR (**pull request**).

Note

When you work collaboratively, you should always work on a branch that is not the `main`.

A few notes

- Learning Git can be frustrating at first, so be patient!
- Once you master it, Git becomes your Swiss Army Knife -- you won't regret it!
- Many GUIs are available for using Git (e.g., GitHub Desktop, VSCode extensions, etc).
- Do not store personal information (e.g., passwords) on public repository
- Hosts like GitHub will reject larger files (as of Fall 2024, GitHub “blocks files larger than 100 MB” and warns about files larger than 50 MB.)

README file

- A README file tells potential users and contributors about your project.
- The README is often written in [Markdown](#), which allows you to add headings, links, tables, images, and other features. (see [tutorials](#)).
- This is what's displayed on project pages on GitHub and similar sites.

Note

It's highly recommended to include a README file in every project!

LICENSE file


- A LICENSE file tells users the conditions under which they can use, redistribute, and even commercialize your project contents. Potential users or contributors may avoid your project if the license is unclear.
- We recommend including a LICENSE in every project!

Open-Source Software License Types


Copyleft Licenses	Permissive Licenses
GNU General Public License (GPL)	Apache License
GNU LGPLv3.0 (Lesser General Public License)	BSD License
Mozilla Public License	MIT License
Eclipse Public License	Unlicense

Getting help

- `git --help` get help on using git
- [Git Cheat Sheet](#)

**Git Cheat Sheet**

For more awesome cheat sheets visit [rebellabs.org!](https://rebellabs.org)



Create a Repository

From scratch - Create a new local repository
`$ git init [project name]`

Download from an existing repository
`$ git clone my_url`

Observe your Repository

List new or modified files not yet committed
`$ git status`

Show the changes to files not yet staged
`$ git diff`

Show the changes to staged files
`$ git diff --cached`

Show all staged and unstaged file changes
`$ git diff HEAD`

Show the changes between two commit ids
`$ git diff commit1 commit2`

List the change dates and authors for a file
`$ git blame [file]`

Show the file changes for a commit id and/or file
`$ git show [commit]:[file]`

Show full change history
`$ git log`

Show change history for file/directory including diffs
`$ git log -p [file/directory]`

Working with Branches

List all local branches
`$ git branch`

List all branches, local and remote
`$ git branch -av`

Switch to a branch, my_branch, and update working directory
`$ git checkout my_branch`

Create a new branch called new_branch
`$ git branch new_branch`

Delete the branch called my_branch
`$ git branch -d my_branch`

Merge branch_a into branch_b
`$ git checkout branch_b`
`$ git merge branch_a`

Tag the current commit
`$ git tag my_tag`

Make a change

Stages the file, ready for commit
`$ git add [file]`

Stage all changed files, ready for commit
`$ git add .`

Commit all staged files to versioned history
`$ git commit -m "commit message"`

Commit all your tracked files to versioned history
`$ git commit -am "commit message"`

Unstages file, keeping the file changes
`$ git reset [file]`

Revert everything to the last commit
`$ git reset --hard`

Synchronize

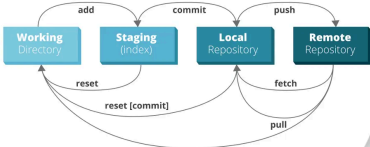
Get the latest changes from origin (no merge)
`$ git fetch`

Fetch the latest changes from origin and merge
`$ git pull`


Fetch the latest changes from origin and rebase
`$ git pull --rebase`

Push local changes to the origin
`$ git push`

Finally!
When in doubt, use `git help`
`$ git command --help`
Or visit <https://training.github.com/> for official GitHub training.



```
graph LR; WD[Working Directory] -- add --> S[Staging (Index)]; S -- commit --> LR[Local Repository]; LR -- push --> RR[Remote Repository]; RR -- fetch --> LR; LR -- pull --> S; S -- reset --> WD; LR -- reset [commit] --> WD;
```



References and credits

- [CHPC presentation on Git](#)^[1]
- Software Carpentry: [Version Control with Git: Summary and Setup](#)
- [Git - Documentation](#)

^[1] Part of course slides are from the tutorial