



Introduction to `git` Part I

Childhood Cancer
Data  **Lab**

x

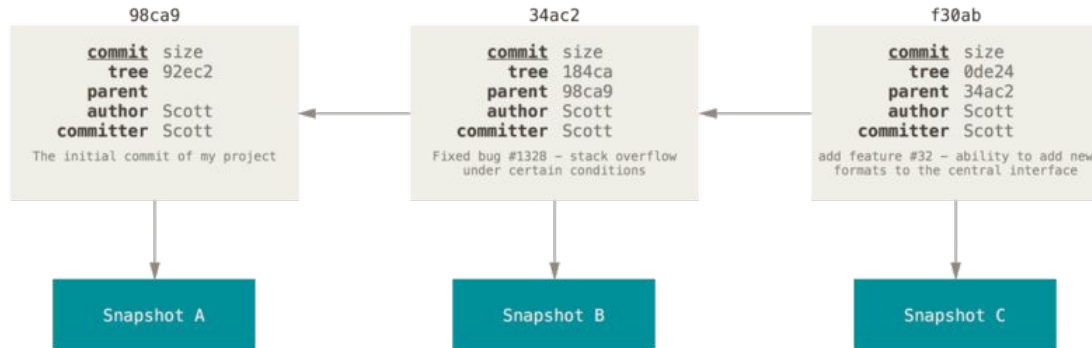


git is a version control system

Provides a framework for keeping snapshots, known in **git** as **commits**, of your project over time

We call a given project that **git** is tracking *a repository (repo for short)*

Repositories are usually used to track code, but they can track other files



git repositories hold a living record of your project

By storing project snapshots, you can safely* overwrite and/or remove files but still be able to recover them

*Unless you don't do it safely

Although **git** is not *technically* a backup, it's not not a backup!

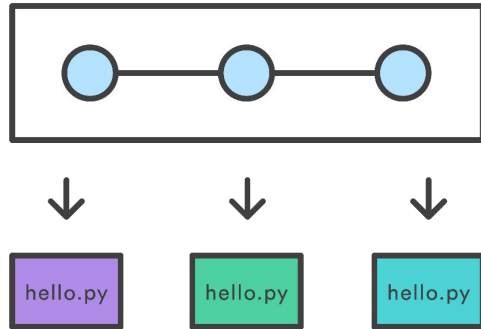


Image from [Atlassian](#)

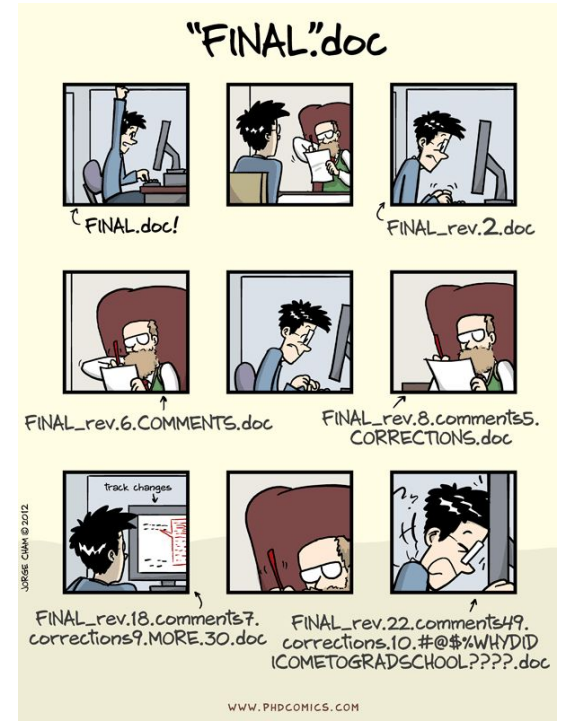


Image from [PhD Comics](#)

GitHub is a popular website for hosting and managing `git` repositories



git is both a powerful individual and collaborative tool

Working independently in a `git` repository and with a team can look a little different!

- We'll come back to different approaches for collaborative `git` workflows later

For the individual, `git` provides a record of your work, allows you to make changes without losing past work (reversion is possible!)

For the team, `git` provides a common framework and structure for collaborating without conflicts (if properly coordinated!)

- Some project management is baked into `git` and/or GitHub, but other aspects you'll need to bring to the table yourself
- 

git repositories are similar to lab notebooks



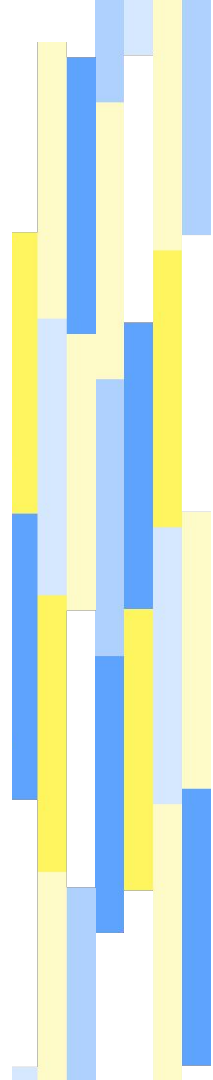
If used well, `git` (along with certain GitHub features!) can store a precise history of project development and changes, not unlike a lab notebook

But without any *context*, this story is a little bit

- Why did you write the code? What roles does it play in your project?
- How does this code fit in the overall picture? How does it relate to other pieces of code?
- Does the code output (screenshots, figures, etc.)? Why do you need those results? How do you interpret those results?
- What dependencies/environment are needed to be able to run the code?
- How do you run the code? How do you document the code?

ABD
Always 🙌 Be 🙌
Documenting 🙌

Documentation is
critical for **you**, your
team, and the broader
research community.



What should you add to a repository?

Code, *sometimes* results, and *sometimes* data

- In particular for scientific analyses, you may have figures and tables that code creates
- Depending on the scope & size of results/data files, they may or may not be suitable for a repository

A **LICENSE** file

- Protects both the rights of you (the authors) and the code itself
- If you are writing software, make sure that your **LICENSE** is compatible with your dependencies
- <https://choosealicense.com/>

That's right you guessed it - documentation

- What are the contents of your repository? How do you navigate the repository?
- What is the relationship among pieces of code? What is the relationship between code and other types of files?
- Are there any specific contribution guidelines folks need to follow?

What shouldn't you add to a repository?

Large files: GitHub cannot accommodate files ≥ 100 MB without extra (less reproducible) steps

- If your colleague has stored large files but you haven't taken the extra steps, you're going to have a bad time.

Unreleased/private data or results files, including PPI/PHI data or results

- Although it is possible to delete sensitive data from a repository's history, you really don't want to find yourself in that position

Any items that you don't have the permission to distribute

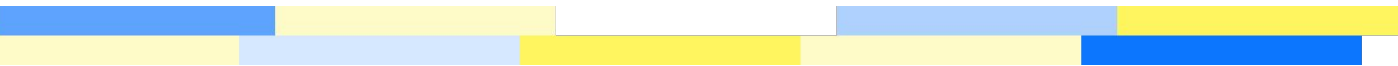
- Always check the **LICENSE** details of your dependencies, third-party datasets, etc.
- 

git repositories are not a replacement for project management tools

Think of `git` & GitHub as providing *complementary frameworks* to your overall project management strategy

You'll also need to develop (*and document!*) external processes that help your team work together in `git` & GitHub





The fundamental `git` flow



Creating a repository

You can create repositories locally, but it's usually easier to make them on GitHub (github.com/new) and then *clone them* to your computer.



Create a new repository

A repository contains all project files, including the revision history. Already have a project repository elsewhere? [Import a repository](#).

Required fields are marked with an asterisk (*).

Repository template

No template ▾

Start your repository with a template repository's contents.

Owner *

 sjspielman ▾

Repository name *

Great repository names are short and memorable. Need inspiration? How about [scaling-octo-adventure](#) ?

Description (optional)

 **Public**

Anyone on the internet can see this repository. You choose who can commit.

 **Private**

You choose who can see and commit to this repository.

Initialize this repository with:

Add a README file

This is where you can write a long description for your project. [Learn more about READMEs](#).

Add .gitignore

.gitignore template: None ▾

Choose which files not to track from a list of templates. [Learn more about ignoring files](#).

Choose a license

License: None ▾

A license tells others what they can and can't do with your code. [Learn more about licenses](#).

Set up your credentials with SSH keys



Reference:

<https://docs.github.com/en/authentication/connecting-to-github-with-ssh/about-ssh>

The Data Lab has started using 1Password to manage creds and it's 🔥 🔥

<https://developer.1password.com/docs/ssh/>

Clone the repository to "download" it

You only do this *one time*.

```
cd directory/to/store/your/repo
```

```
git clone git@github.com:ACCOUNT-NAME/REPO-NAME
```



Future interaction between your *remote and local* repositories is done with...

- `git push`: Send changes from local repo to remote repo
- `git pull`: Get changes from remote repo into your local repo

The "stage/commit/push" flow

1. **Pull** changes from GitHub to make sure your local repo is up-to-date, and ensure you're in the right branch (*stay tuned!*)
2. Make code changes, in small bites!
3. **Stage** the changes you want to be part of the next commit
 - a. Stage with `git add`, `git rm`, etc.
 - b. Best practice to stage one new/modified/removed file at a time to avoid problems
4. **Commit** your code changes with an informative message
 - a. Commit with `git commit -m "informative message"`
(*we know there are lots of choices here!)
5. **Push** your local commit(s) to the **remote repository** you cloned from GitHub
 - a. Push with `git push`



When it comes to commits, less is more

Smaller units of work are better for...

- You, when writing your code
- You, when reviewing someone else's code
- Your teammate, when reviewing your code

If you are reviewing someone's work, which commit message is most helpful?

- All the work I did on Tuesday
- Update notebook
- Modified notebook to change plot size

Which scenario is easier to look through?

- 1-3 changed files in a commit, or 25 changed files
- 400 lines of new code, or 4000 lines of new code



	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
○	AAAAAAAAA	3 HOURS AGO
○	ADKFJSLKDFJSDKLFJ	3 HOURS AGO
○	MY HANDS ARE TYPING WORDS	2 HOURS AGO
○	HAHAHAHAANDS	2 HOURS AGO

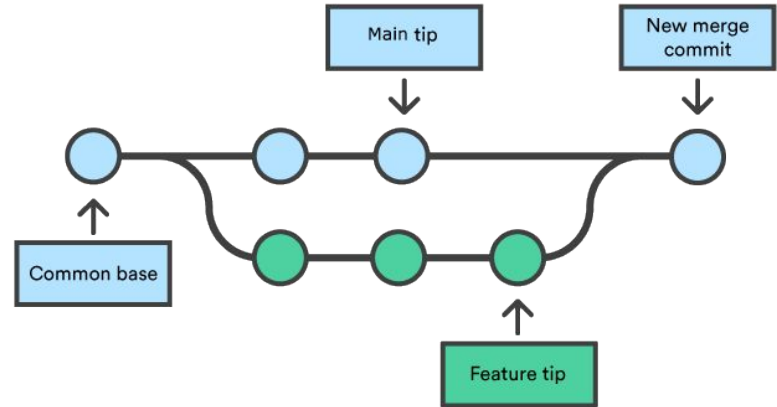
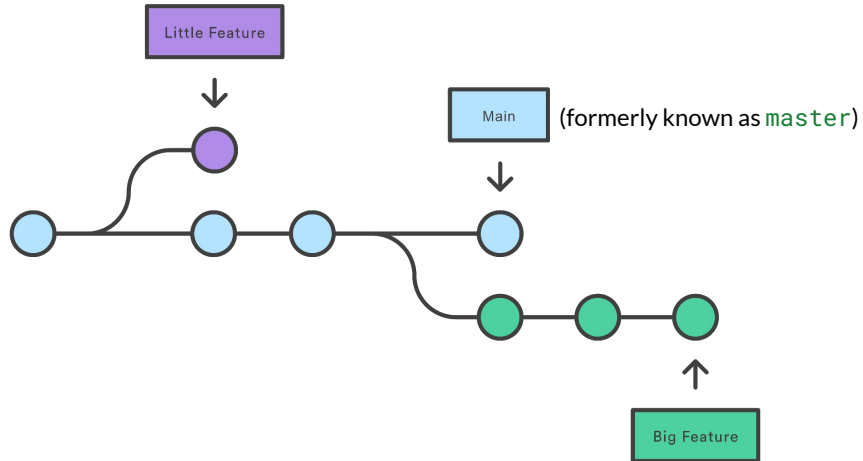
AS A PROJECT DRAGS ON, MY GIT COMMIT MESSAGES GET LESS AND LESS INFORMATIVE.

We often like to work in feature branches

Collaborative work in different branches is a key feature* of **git** version control

*Pun intended, with apologies.

We'll talk much more about branches soon!




Images from Atlassian [\(L\)](#) [\(R\)](#)

The unsung heroes `git diff` and `git status`

`git diff` should be run for each change before you stage it

- *You think you know what code you've written, but you have no idea.*
- `git diff` alone will show you changes for all modified files
- `git diff <modified file>` will show just your file of interest

`git status` should be run ~as often as possible, and then maybe more. It tells you...

- Which branch you are on
- Which files are modified/added/removed but unstaged
- Which changes are staged but not committed
- How many local (unpushed) commits you have
-  *Protip!* I have this alias on every machine I use, and I type it *constantly*: `alias gst="git status"`

The improved "stage/commit/push" flow



1. Run `git status`
2. Make code changes
3. Run `git status`
4. **Stage** the changes you want to be part of the next commit
 - a. `git diff` first, make any necessary changes, and then stage
5. Run `git status`
6. **Commit** your code changes with an informative message
7. Run `git status`
8. **Push** your local commit(s) to the **remote repository** you cloned from GitHub
9. Run `git status`

Safely undoing changes: A non-comprehensive introduction

Woops! I modified a file that I didn't mean to modify, *and I know this because I ran `git status`*

- `git checkout <filename>`
- `git restore <filename>` (`git >= 2.23`)

Woops! I committed a file that I didn't mean to modify, *and I know this because I ran `git status`*

- `git restore --staged <filename>`

Woops! I staged files that I wasn't ready to stage yet, *which I know because I ran `git status`*; can I undo the staging?

- `git reset`
- [See here](#) for more ways to use `git reset`, some of which are  *destructive* (will undo previous commits!)

Safely undoing changes: A non-comprehensive introduction

Woops! My commit message should have been different!

- `git commit --amend`

Woops! I want to undo the changes in a previous commit, but still preserve the project history and avoid destructive actions!

- `git revert <commit hash>`

For more inspiration and/or commiseration, may I suggest <https://dangitgit.com/>

Using `.gitkeep` files

`git`, in general, does not store empty directories, but sometimes we need them

By *convention* (not a formal `git` feature!), we add empty hidden files called `.gitkeep` into empty directories we want to retain. For example....

- A `data/` directory that stores data locally and that code expects, but data itself is far too big to actually keep under version control
- Directories that you create for future use when setting up your project, but don't yet have contents, e.g. `plots/`

Using `.gitignore` files




`.gitignore` files are hidden files that tell `git` to ignore certain files. They can stay in your repository folder, but won't be part of the repository itself.

Telling `git` to ignore certain files will....

- Make `git` stop telling you about untracked changes when you run `git status`
- Prevent you from staging/committing these files by accident, and running `git status` liberally will help key you into which files need to be ignored

Using `.gitignore` files

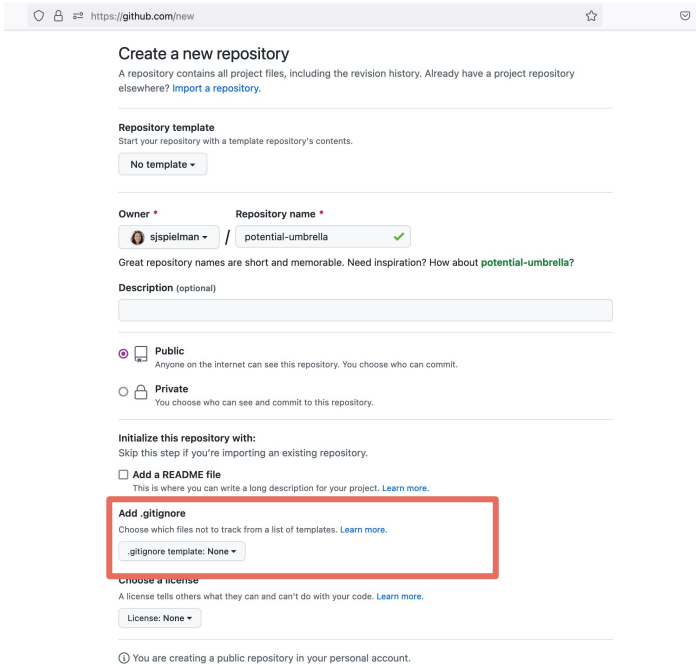
You can have many `.gitignore` files in a repo (in different directories) and/or a single `.gitignore` at the top of your repo that all subdirectories "inherit"

- You can also have a global file in `~/.gitignore` that will apply to all repos on your computer
-  *But caution:* Your collaborators probably don't have that file.
-   *But caution again:* You probably don't have that file synced up among machines (e.g., your computer and your lab server)

 *Protip!* Working on a Mac? `.DS_Store` does not spark joy. Ignore it!

GitHub has some useful `.gitignore` templates

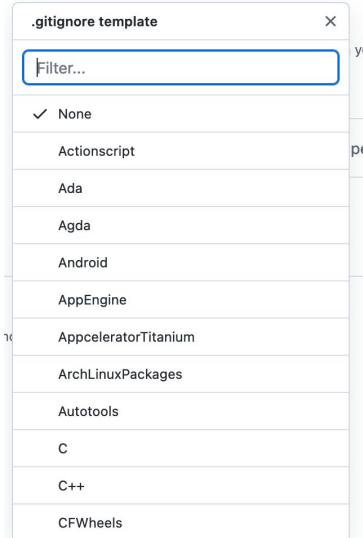
This is one example of a GitHub feature that helps you organize your projects!



Add .gitignore

Choose which files not to track from a list of templates.

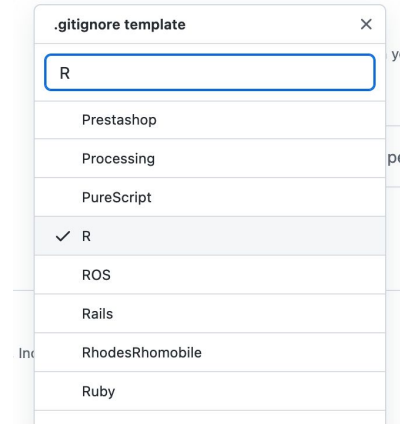
.gitignore template: None ▾



Add .gitignore

Choose which files not to track from a list of templates.

.gitignore template: R ▾





Let's see some `git` in action

