

Git & GitHub

Module 7

October 24, 2022

Housekeeping

- Quiz #2 at the end of class next week
- Prep quiz questions available
- Introduction of final project after today's module
- No homework this week

What is Version Control?

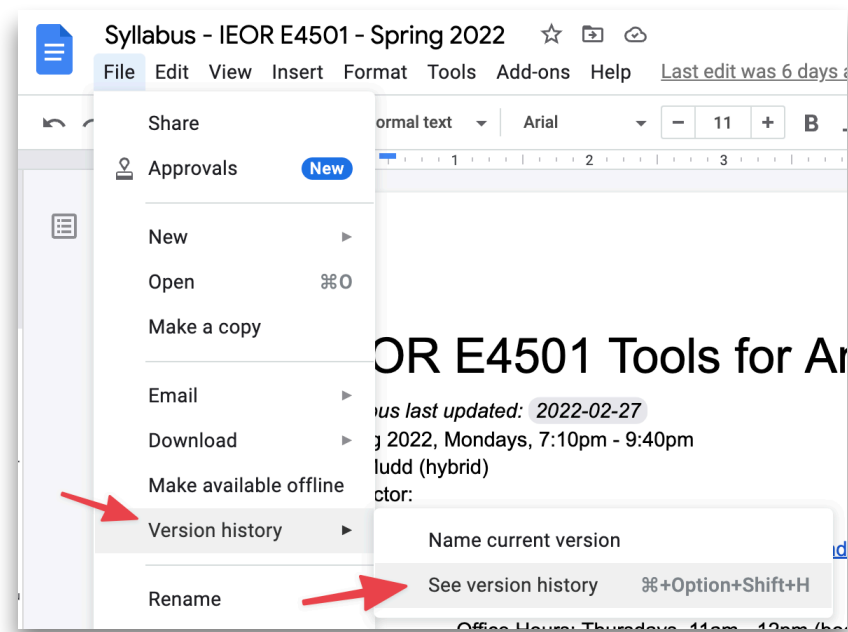
- The practice of tracking and managing changes to code
- Also known as “source control”
- Keeps track of every modification

- Version control allows individuals and teams track changes to a code base
- A code base can be anything made up of code - duh - but for y'all, it could be your completed homework, or your year-end final project.
- Version control tracks the changes made to code, and can help manage the "single source of truth" for your code.

Example: Google Docs

I'm sure you're all familiar with Google Docs.

And perhaps you're familiar with Google Doc's version history:



← February 27, 11:05 AM

Version history

🖨️

FIR

Total: 3 edits

IEOR E4501 Tools for Analytics

Syllabus last updated: [2022-02-19](#) - [2022-02-27](#)
Spring 2022, Mondays, 7:10pm - 9:40pm
833 Mudd (hybrid)
Instructor:
Lynn Root <lroot@columbia.edu>
Office hours: Fridays, 1pm - 2pm (book via [Calendar](#))
CAE:
Lini Guan <lg2183@columbia.edu>
Office Hours: Thursdays, 11am - 12pm (book via [Calendar](#))
Shikha Sethia <ssd1118@columbia.edu>
Office Hours: Wednesdays, 8am - 10am (join [Zoom](#))

Description

The goal of this course is to introduce students to the basics of programming in Python and the tools within the programmer's ecosystem. By the end of the course, students will have a working knowledge of Python and the tools within its numerical computing ecosystem.

Prerequisites

There are **NO** course prerequisites for this class.

Videos

To help with context during the course, students should watch the following videos.

- Python101 - Chapter 1
- Python101 - Chapter 2
- Python201 - Chapter 1 (Basics)
- Python201 - Chapter 2 (Types)
- Python201 - Chapter 3 (Strings)
- Python201 - Chapter 4 (Bytes)
- Python201 - Chapter 5 (Variables)
- Python201 - Chapter 6 (Loops)

Textbook

There is **NO** textbook for this class.

The [appendix below](#) will be helpful if you want to go above and beyond the material covered in the course.

Classes

Topics

The following topic schedule is a rough guide, and may be adjusted as we go through the course.

Only show named versions

SUNDAY

February 27, 11:05 AM

Current version

Lynn Root

FEBRUARY

February 11, 3:43 PM

Lynn Root

February 10, 8:45 AM

Lynn Root

February 2, 9:04 AM

Lynn Root

JANUARY

January 31, 9:14 AM

Lynn Root

January 26, 9:23 AM

Lynn Root

January 23, 12:49 PM

Lynn Root

January 22, 12:25 PM

Lynn Root

January 19, 7:01 PM

Lynn Root

January 19, 1:08 PM

Lynn Root

January 17, 10:08 PM

Lynn Root

Show changes

Now this is a bit clunky - it's not the best interface, but you can click through different snapshots of the versions here on the right to go back in time, to roll back to a previous version.

What is Version Control?

Original

```
my_list = [1, 2, 3, 4, 5]
for i in range(len(my_list)):
    print(my_list[i])
```

Your Change

```
my_list = [1, 2, 3, 4, 5]
for i in my_list:
    print(i)
```

Project Partner's Change

```
my_list = [1, 2, 3, 4, 5]
while i < len(my_list) + 1:
    print(my_list[i])
```

- Version control serves as sort of a safety net to protect code. It allows you to experiment - and if you don't like the results, you can roll back your changes safely.
- When working with other people on the same project, on the same code, it protects you from creating incompatible changes, or "conflicts."
- For instance, say you and another person are working on a project together. <CLICK> You have this piece of existing code. <CLICK> You make a change to a line on your local copy of the project, <CLICK> and your partner makes a change to the same line on their copy. How does this get resolved?
- This is what version control tries to protect.
- It also allows you to easily go back in time - maybe you introduced a bug in your change. Maybe your partner on your project got in their change for that `while` loop, and you want to revert it. Changes that are tracked can be easily undone.

What is Git?

- Created in 2005 by Linus Torvalds
- Same person who invented/created Linux
- He needed a better program to manage development on Linux
- The name comes from British slang “git”, meaning an unpleasant or stupid person
(Linus named it after himself)

Git is a program - it's a very popular version control system.

<CLICK>

It was created in 2005 by Finnish dude named Linus Torvalds. Infamous <strike>person</strike> asshole.

<CLICK>

This is the same person who invented/created the Linux operating system.

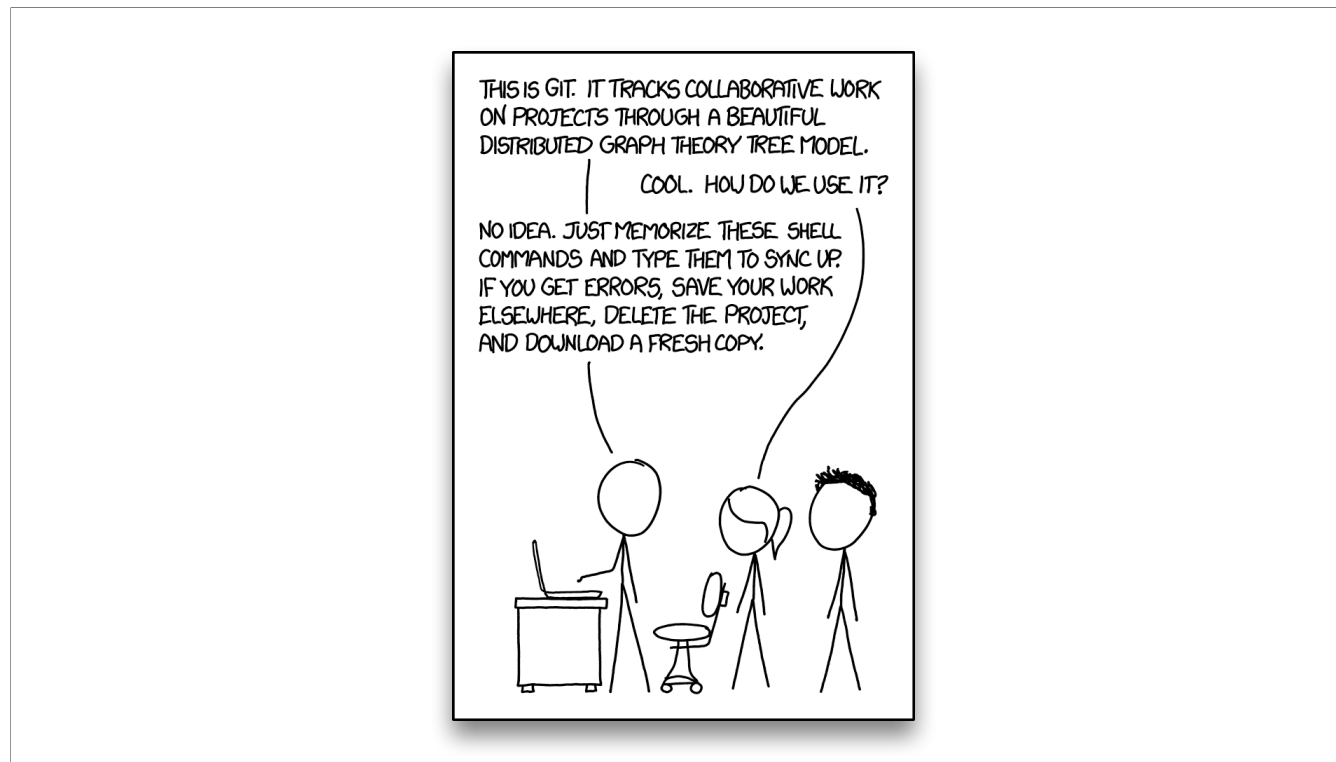
<CLICK>

Git came about because Linus and the other developers working on Linux needed a better program to manage its development. Actually, it came about because the software they were originally using revoked its "free use" permission, and no other version control software existed that allowed for a distributed environment - which I'll get into what that exactly means in a bit.

<CLICK>

Essentially, Git is an implementation of version control, and is used for:

- * tracking code changes
- * tracking _who_ made those changes
- * and collaborating over large and small projects efficiently.



Git is known for being confusing - this XKCD comic captures the sentiment pretty well.

It's very extensive - there are a _lot_ of features in it. But thankfully, you can get a lot with just a basic understanding of what's going on under the hood, and knowing only a small percentage of its commands.

What is Git?

```
git --help
```

Git is first a CLI tool, with many implementations of visual interfaces on top of it (which can be particularly helpful when first learning, and for more complex/larger projects).

Learning *how* to use git - via its CLI or visual interface - can lead to a lot of confusion at first. So we'll first start with git's underlying design and concepts. I first want you to understand the basics of git's design, its data model. Then, you'll essentially memorize a set of commands that manipulates the underlying data model.

Vocab: repository

A git **repository** is a collection of files – usually for a particular project – with associated history of changes and some metadata.

Also known as a **repo** for short.

Git operates on something called a **repository**. aka "repo"

A repository is just a project or a folder with one or more files containing code or documents. A git repository tracks the changes of the files over time. All the files in a repository directory are track, unless you explicitly tell git to ignore them. So this means, any changes to a file inside of a repository, git will notice.

Vocab: blobs and trees

Git refers to a file (any kind of file) as a **blob**. A directory is then a **tree**.

A tree can be made up of blobs (files) and trees (sub-directories).

my_repo/	(tree - root)
├── foo/	(tree)
│ └── bar.txt	(blob, contents = "hello world")
└── baz.txt	(blob, contents = "git is wonderful")

Within a repository, git's terminology for a file is a blob - just a bunch of bytes.

And then a directory is referred to as a tree. A tree then maps names to blobs or trees (so directories can contain other directories).

<click>

Vocab: commit

A git **commit** is a snapshot or "save point" of the repository. A git repository's history of changes is made up of **commits** at different points in time.

When creating history for a git repository, we create commits. It's a snapshot, a point in time of the history of the project.

Commits are the core building block units of a git project's timeline. They capture the complete state of a project at a point in time.

If a commit exists in a repo, you can revert back to it. Just like you can with GDocs's version history.

A commit can reflect a change to one file, or multiple files. It can even include the additions of new files, or deletion of files.

A commit has a `_hash_` - a unique identifier for the snapshot. It also contains some metadata, like when the commit was made, who made the commit, and a message describing the change that was made.

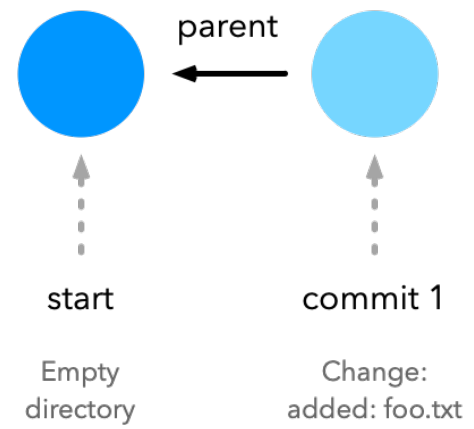
Vocab: commit



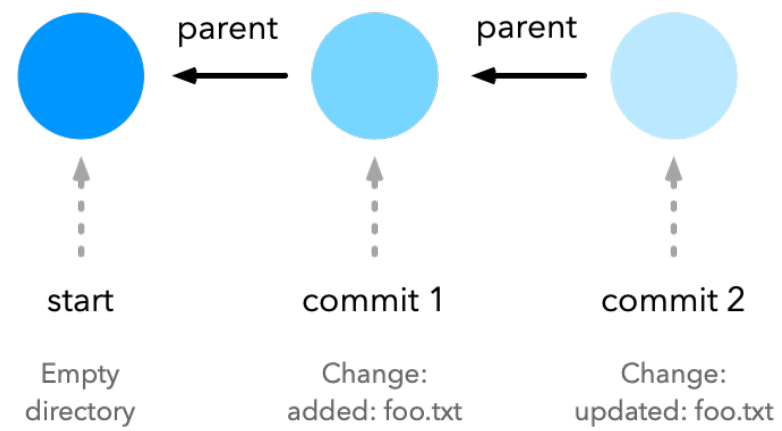
start

Empty
directory

Vocab: commit



Vocab: commit



Vocab: branch

Literally speaking, a git **branch** is a pointer to a particular commit.

In an abstract sense, a **branch** represents a line of development, or a line of changes.

The default branch is `master` (or `main`). Consider `master` as the "most real or current" version of your code/repository. We create branches off of `master` (or `main`).

We branch off of ``master`` or ``main`` to add a specific feature, or fix a bug. This way we don't mess up the "main" version of our code. We can experiment, iterate, test, have others review the code to give their input, all before we add it to the master version of our code.

Vocab: merge

To **merge** in git means to integrate changes from one branch to another branch.

Hopefully this makes sense -

Say we have a branch that adds some code, and we're done with all the changes we'd like to make. We can then `_merge_` this branch into the master branch (or whatever other branch) to incorporate our new changes.

The act of doing this actually automatically generates a new commit for us.

Here's a visual for branching and merging:

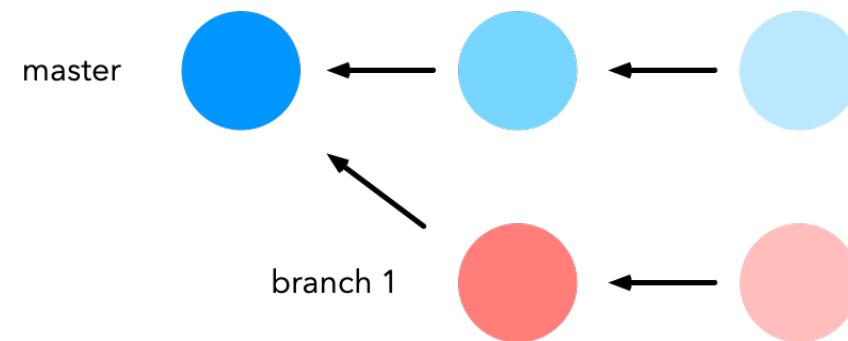
Vocab: merge



So we have snapshots that we refer to as commits. The concept of a snapshot implies there's some sort of relationship between snapshots.

For instance, like with our google doc version history, we can have a linear history - a list of snapshots in chronological order.

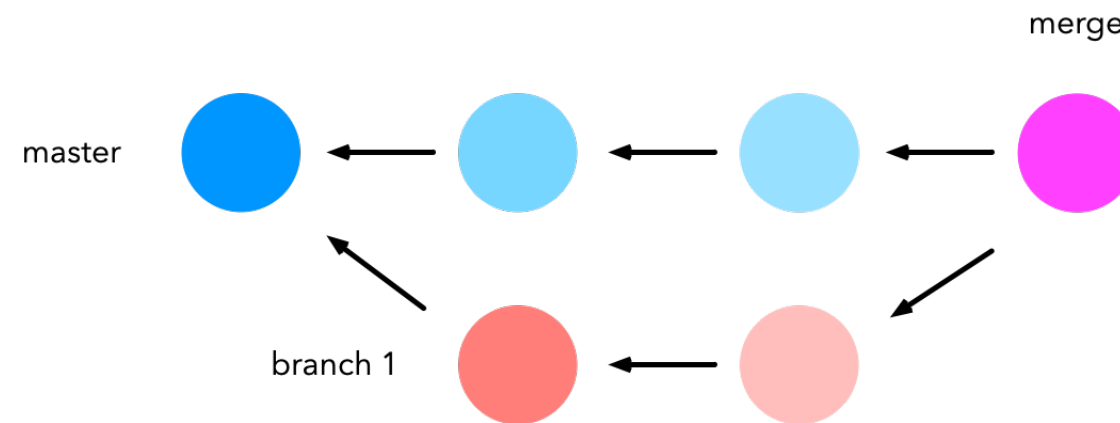
Vocab: merge



But git doesn't use this relationship model. The snapshot history (commit history) in Git is actually a dag - a directed acyclic graph (remember our data structures?).

All that means is a commit - or a snapshot - has at least one parent commit. When we create a branch, it means we have two separate timelines of history. This might correspond to two different features being developed in parallel, independently from each other.

Vocab: merge



We can then merge one branch into another to create a new snapshot - a new commit.

Git's Data Model

pseudocode

```
// a file is a bunch of bytes
type blob = array<byte>

// a directory contains named files and directories
type tree = map<string, tree | blob>

// a commit has parents, metadata, and the top-level tree
type commit = struct {
  parents: array<commit>
  author: string
  date: string
  message: string
}
```

Looking at this pseudocode, the data model for git isn't too complex.

We have a blob - which is just an array of bytes, aka a file.

We have a tree, which reflects a directory, and is a mapping of a name to other trees (subdirectory) and blobs.

And then finally, we have a commit. A commit has one or more parent commits, it has an author, a date, and a message - like description of what was changed and why.

Git *Object*

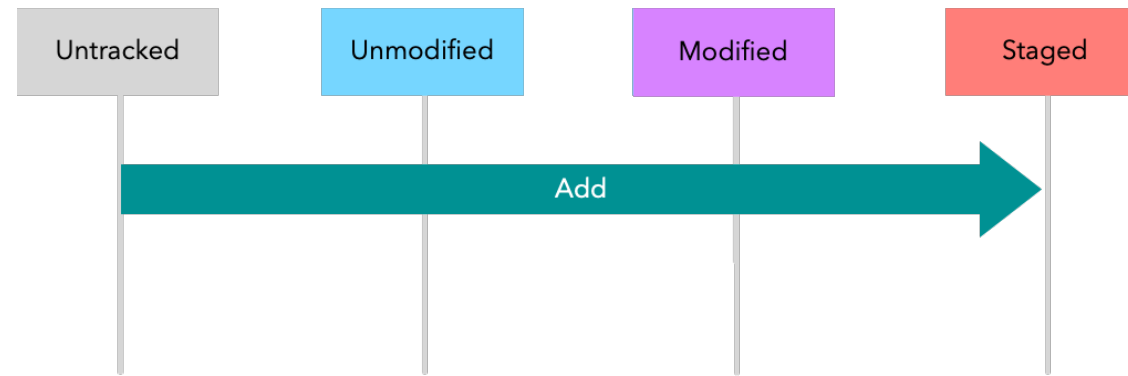
```
type object = blob | tree | commit
```

An "object" in git refers to either a blob, tree, or commit.

All objects in git have an "address". This address is just a unique identifier. Git generates this identifier by what's called "hashing" the contents of the object - whether it be a file, a directory, or a commit itself.

This bit may not make too much sense right now - hopefully it will when we see it in action in a little bit.

Git Lifecycle of Files



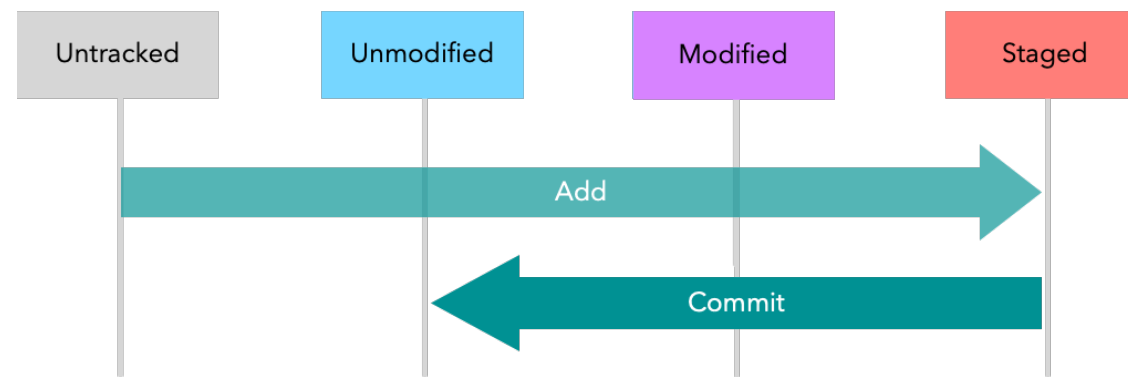
When you create a new file in a git repository, it's considered "untracked" at first. Git doesn't know about it.

You **add** it to the repository, which puts the file into our "staging" area. This is basically us queuing up changes before we officially commit them to our git history.

As well, by adding this file to our staging area, we've told git to start tracking this file.

Git's specific terminology for the staging area is the "index".

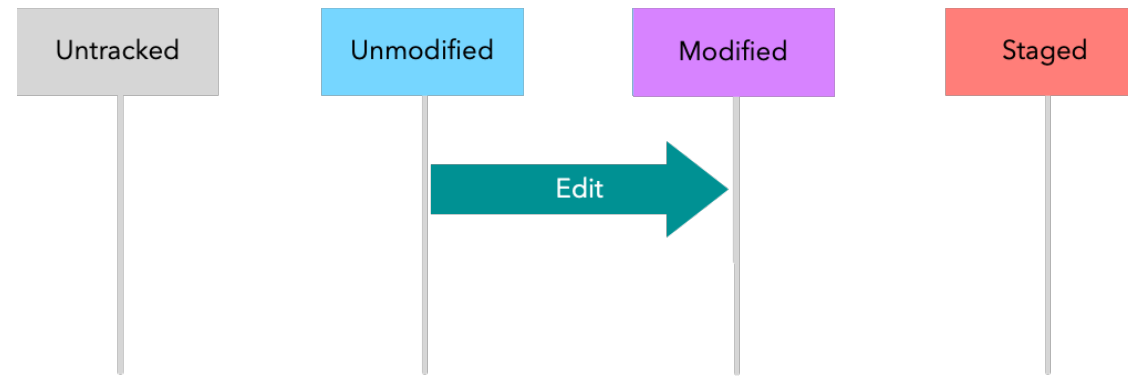
Git Lifecycle of Files



When we're done making changes, we then commit them to our repo. We've created a snapshot.

At this point, we now have a "clean" working directory. That means, none of our tracked files are modified.

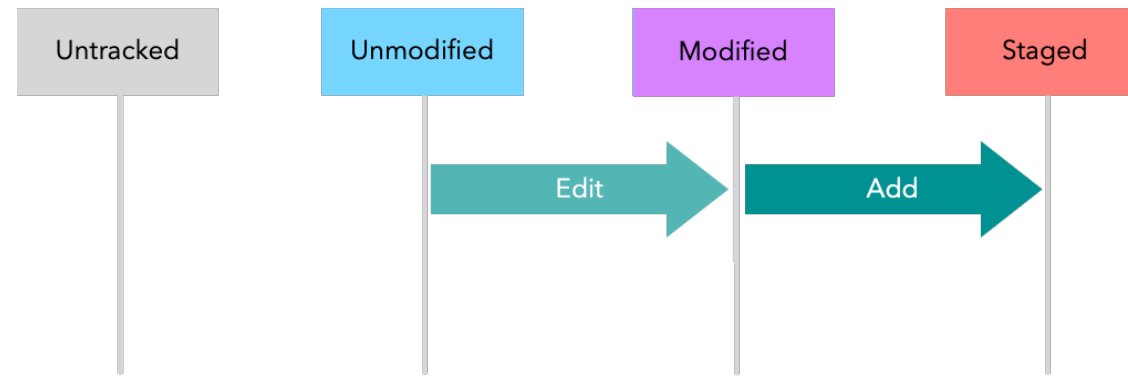
Git Lifecycle of Files



Let's say now we've made some edits to a file in our repo.

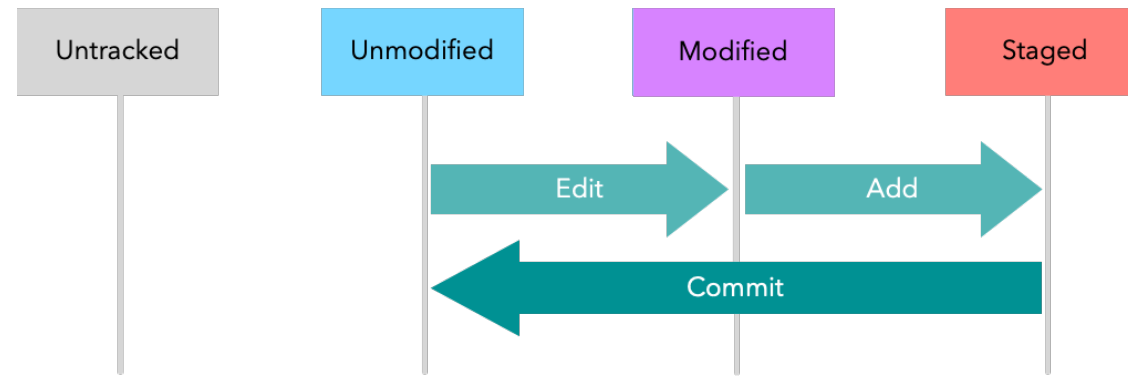
This file is now modified in our working directory, but it's not yet staged. Our working directory is considered "dirty" here - we have made some changes but haven't done anything with them in the git repo.

Git Lifecycle of Files



After we're done editing our file, we can add it to our staging environment. We're queuing it up to create another snapshot - another commit.

Git Lifecycle of Files



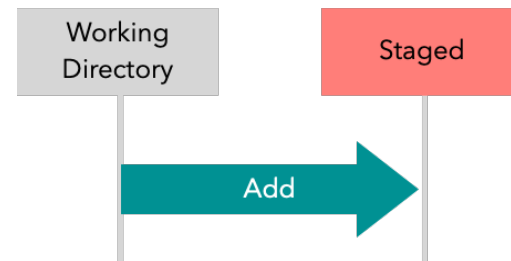
We then commit our changes, and we're back to our clean working directory.

Git Lifecycle Overview



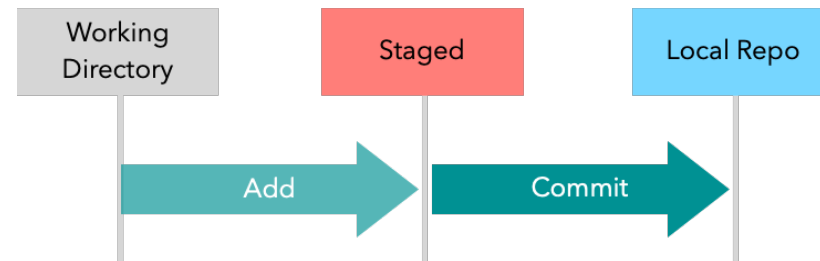
Stepping back a little bit to see a slightly larger view of a git repos life cycle, let's say we have our working directory - we're in our repo. We make some edits, create some new files...

Git Lifecycle Overview



Then, like before, we add those changes and new files to our staging environment. Maybe we go back, tweak those changes, fix typos, and add those changes to staging.

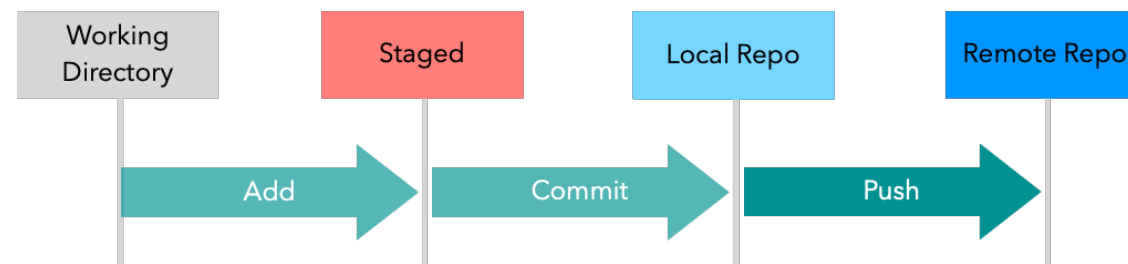
Git Lifecycle Overview



We're done with this round of edits, and like before, we commit them to our local repo - we've made a snapshot.

But it's only on our local copy. If we're working on a project with others, or maybe we're working on a project by ourselves but we want a copy of the repo that we can access on our personal computer as well as maybe a school library computer, we need to write those committed changes somewhere.

Git Lifecycle Overview

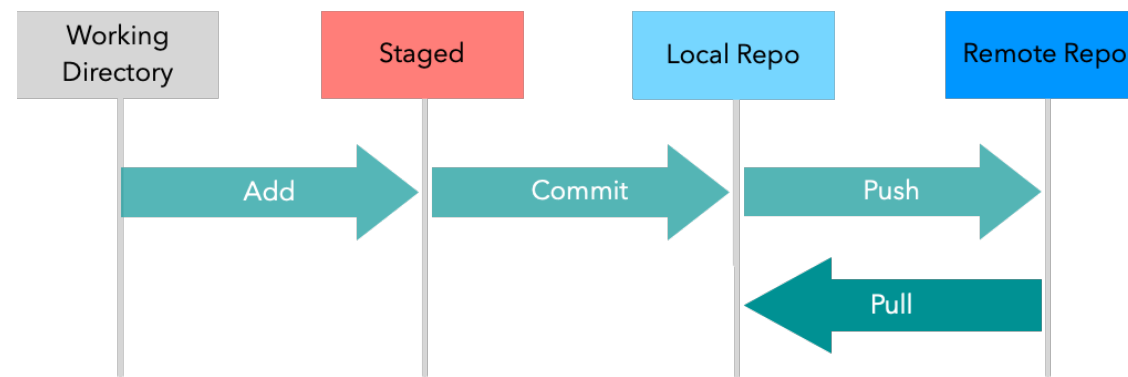


To do so, we `_push_` our commits to a remote repository.

Now the key piece of git being "[distributed](<https://www.perforce.com/blog/vcs/what-dvcs-anyway>)" and collaborative is having a remote repository - basically a server that has the "main" copy of your project's code that you and your team work off of. We often refer to this as the "origin" repo.

With a remote repo, you and your team mates can have a local copy of the project, make local changes, and push to the origin repository. It's like similar to a master branch, which we treat as the most real or current version of the code. But every person has a local master branch, and we need something to designate as the "most real or current" version of the overall repo.

Git Lifecycle Overview



Now say your team mate made some changes to a project you're both working on, but you don't have those changes in your local copy of the repo.

It's not like GDocs, where someone's edits is immediately available to view in your browser. We have to `_pull_` in the changes that were written to the remote repo into your local repo.

So remote repos brings us to GitHub...

What is GitHub?

- Hosts repositories (the “remote” server)
- Loosely: video files are to git repos as YouTube is to GitHub
- Other options for hosting repositories - BitBucket, GitLab



Before we start playing with git, let's first go over what GitHub is

—

GitHub is a website - a service - that hosts those remote repositories. It's that piece that allows multiple people to work on a single project at the same time.

Git isn't GitHub, and GitHub isn't Git - the two aren't interchangeable. GitHub is basically making Git sharable and collaborative - you work on a project with team mates. But then you can see what other people are working on; others can share their GitHub repositories with you for you to check out, to use, to invite new people to collaborate on.

GitHub isn't the only website to do this - there's also BitBucket and GitLab; and they're not the first. But they're definitely the most popular.

There are a couple of terms that are specific to GitHub, but not applicable to Git.

Vocab: pull request

Specific to GitHub, a **pull request** tells others about changes you've made to a branch you've created, asking to merge into a particular target branch, like `main` or `master`.

Vocab: issue

GitHub **issues** are a means for tracking work, including features and bugs.

Users of software file an **issue** against a project's repository if they have a feature they'd like to request, or encountered a bug.

Developers of a repository can use **issues** to track the work they need to do (like a TODO list of tasks to knock out).

Vocab: fork

A **fork** in GitHub means a copy of someone else's repository that you manage.

Someone who **forks** a repository in GitHub has *complete* control over the code base, like creating a duplicate of a Google Doc.

Aside: fork != clone

A **fork** is a complete copy of a repository under someone else's GitHub name/organization to your own.

A **clone** (a git term, not a GitHub term) clones an existing copy of a repository, often to your local machine.

When contributing to a shared project (not under your own GitHub name), first create a **fork** to have it under your own GitHub name. Then **clone** your fork to your local computer. Make changes, and push them to your fork on GitHub. Then make a **pull request** to the original project to request incorporating your changes into the main "upstream" project.

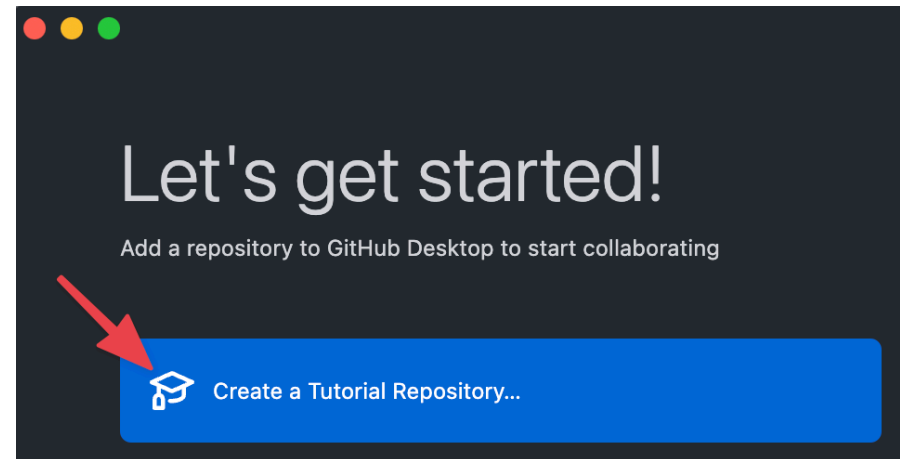
Exercise: Setup Git & GitHub

Download Lecture Notes for Guidance

1. Create a GitHub account:
https://education.github.com/discount_requests/student_application
2. Download & Install GitHub Desktop: <https://desktop.github.com>
3. Download & Install VSCode: <https://vscode.github.com>
4. Launch GitHub Desktop app & login to GitHub.
5. Follow the "Create a Tutorial Repository..."

After Step 5, follow the lecture notes for picture-by-picture steps to follow along.

Step 5: Follow the “Create a Tutorial Repository...”

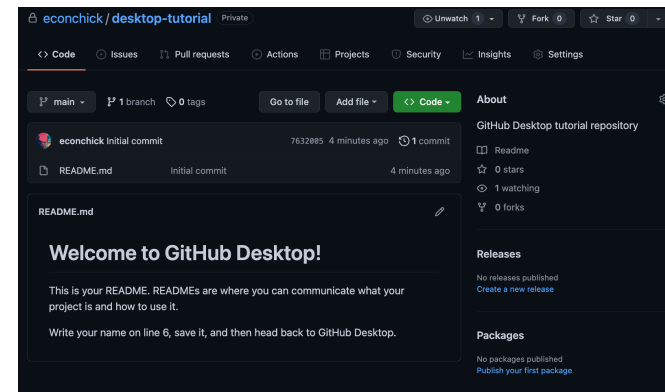


Step 6: View new repo on web

Let's look at it on the GitHub website: [github.com/\\$USERNAME/desktop-tutorial](https://github.com/$USERNAME/desktop-tutorial).

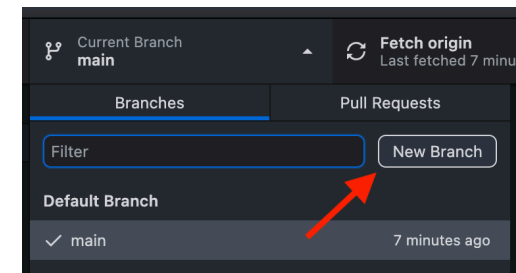
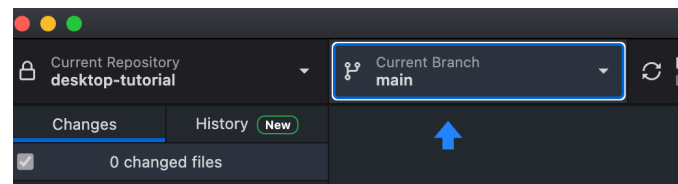
Replace \$USERNAME in the URL with your username that you created.

Click around if you want. You can see it created our first commit for us.



Step 7: Create a new branch

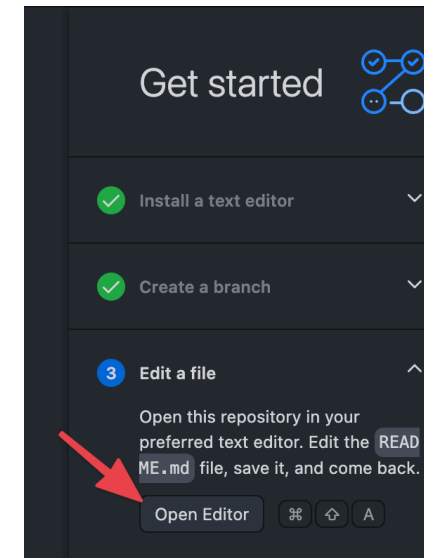
1. Return to the Desktop application.
2. Create a new branch by clicking on the “Current Branch”:



Step 8: Make a change, part 1

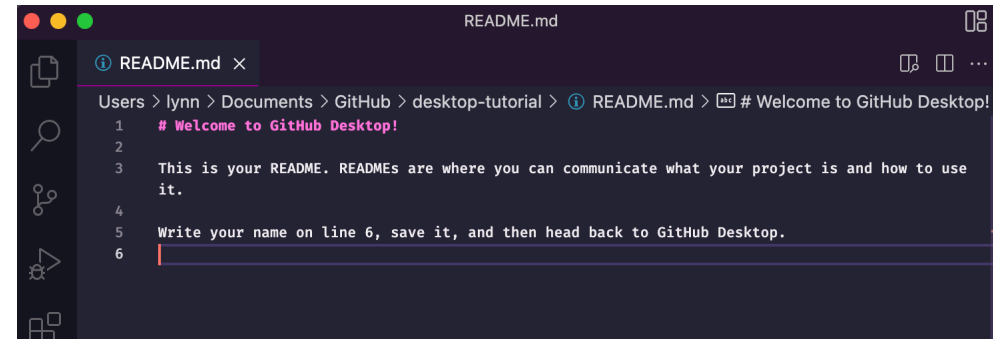
To the right, with “Step 3” of the tutorial in the app, click “Open Editor”.

This opens the VSCode app that you downloaded earlier.



Step 9: Make a change, part 2

Follow the instructions in the opened file, README.md

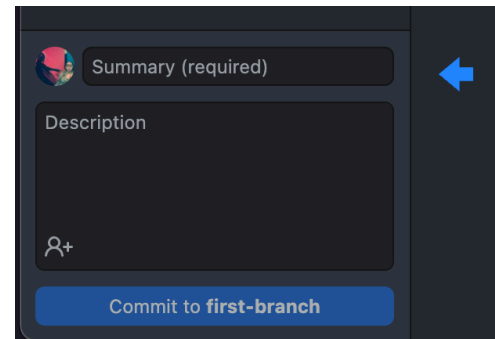
A screenshot of a code editor window titled 'README.md'. The editor has a dark theme. On the left, there is a sidebar with icons for file explorer, search, source control, and a run/debug console. The main area shows the content of the README.md file. The file path is 'Users > lynn > Documents > GitHub > desktop-tutorial > README.md'. The content of the file is as follows:

```
1 # Welcome to GitHub Desktop!
2
3 This is your README. READMEs are where you can communicate what your project is and how to use
  it.
4
5 Write your name on line 6, save it, and then head back to GitHub Desktop.
6
```

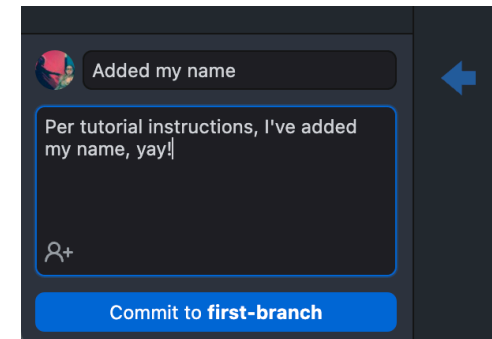
The cursor is positioned at the end of line 6.

Step 10: Make a commit

Return to the Desktop application. Then make a commit:



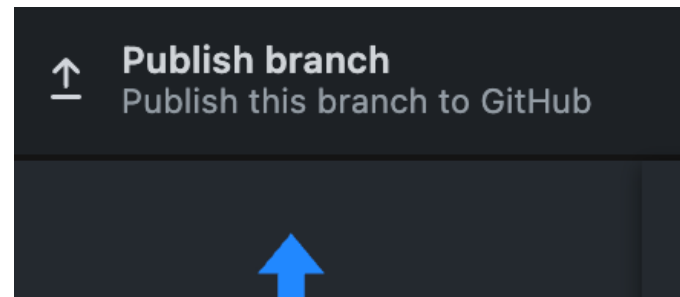
A screenshot of a commit summary screen in a dark-themed application. At the top left is a small circular profile picture icon. To its right is a text input field containing the placeholder text "Summary (required)". Below this is a larger text area labeled "Description". At the bottom left of the description area is a small icon of a person with a plus sign. At the bottom center is a blue button with the text "Commit to first-branch". On the right side of the screen is a vertical bar with a blue arrow pointing left.



A screenshot of the same commit summary screen as the previous one, but with a commit message entered. The text input field now contains "Added my name". The "Description" text area contains the text "Per tutorial instructions, I've added my name, yay!". The blue button at the bottom remains "Commit to first-branch". The layout and icons are identical to the previous screenshot.

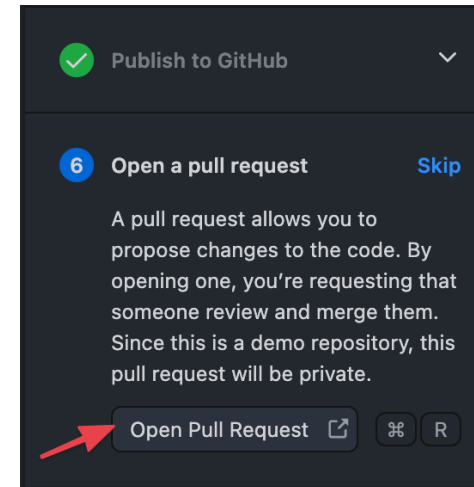
Step 11: Publish (push) to GitHub

Publish changes with your new branch to GitHub:



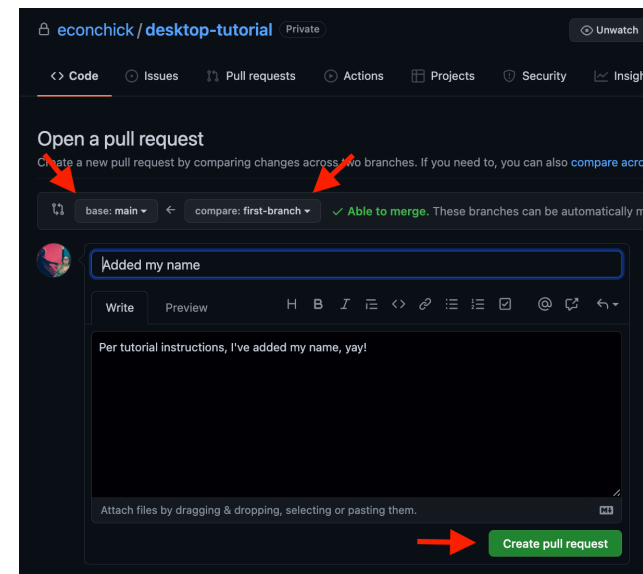
Step 12: Make a Pull Request

Still within the Desktop app, click on
“Open Pull Request”:



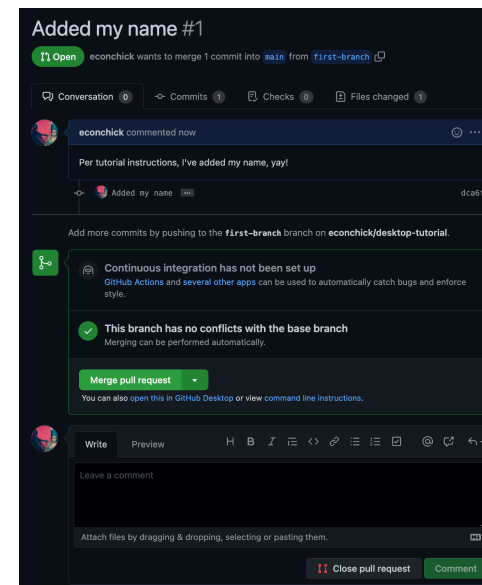
Step 13: Open a Pull Request, part 1

Now on the website, open a pull request against the “main” branch



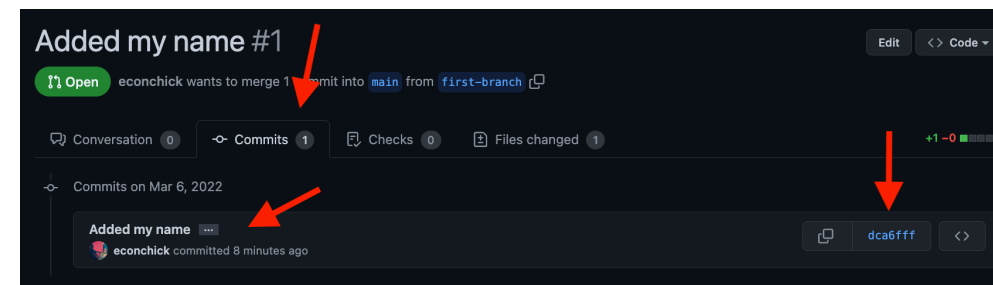
Step 14: Open a Pull Request, part 2

Click on the “Create Pull Request” button and you should see:



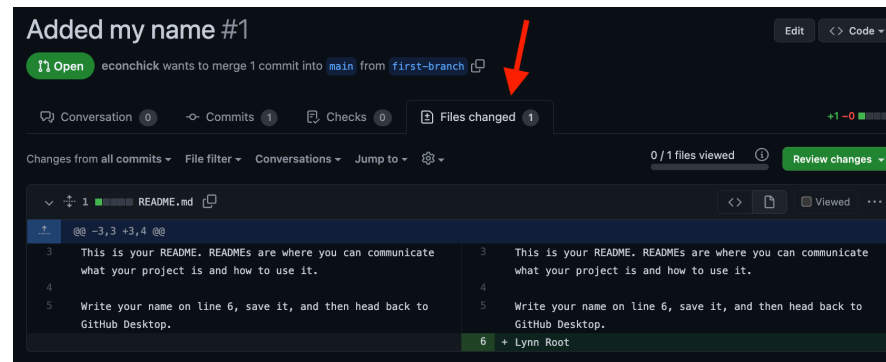
Step 15: View the commits

Click on the "Commits" tab to view the commit we made. We can see our message "headline", and the shortened commit ID. If we clicked the `...` button, we would see our full commit message body.



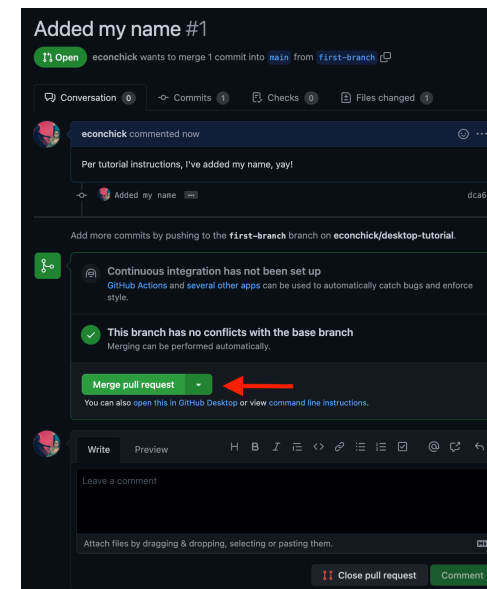
Step 16: View the changes

Clicking on the "Files changed" tab, we can see what's called a "diff" of our changes. The left side is the state of the file before our change, the right side is the state of the file after our change.



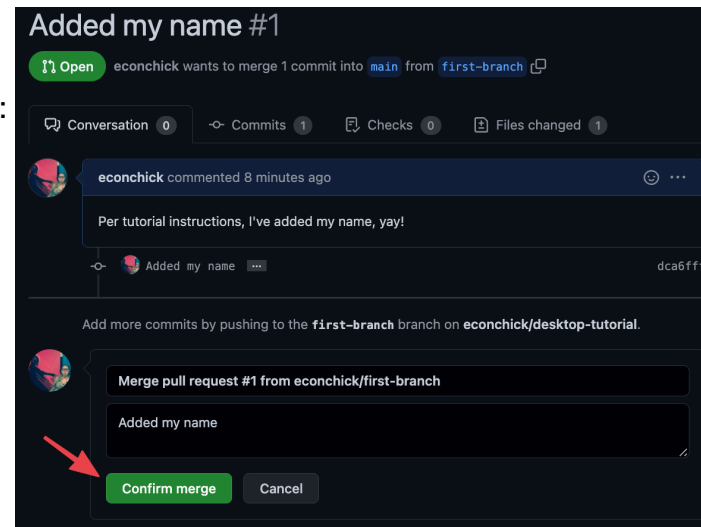
Step 17: Merge pull request, part 1

Return back to "Conversation" tab,
then click "Merge pull request"



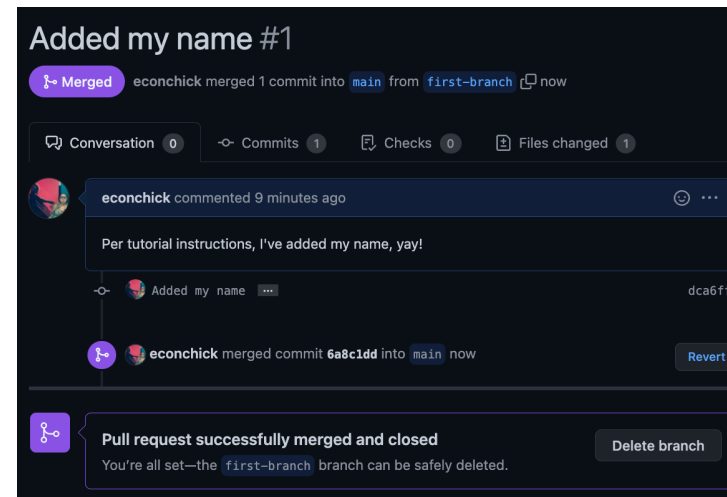
Step 18: Merge pull request, part 2

Then “confirm merge”:



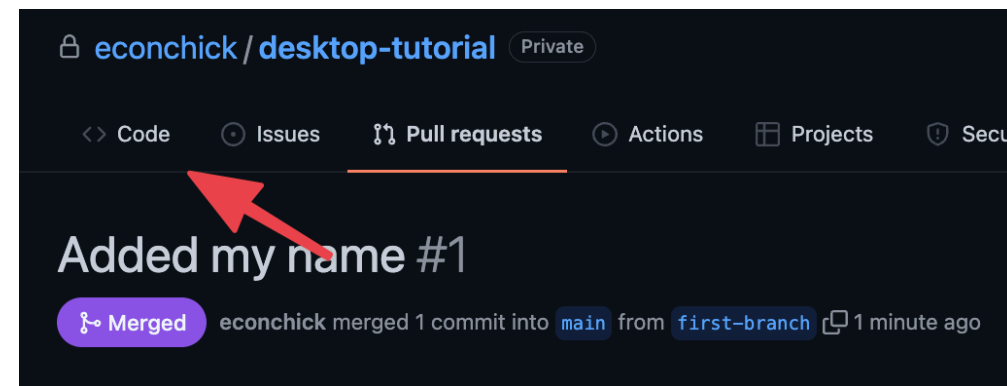
Step 19: Merge pull request, part 3

We've merged our first pull request!



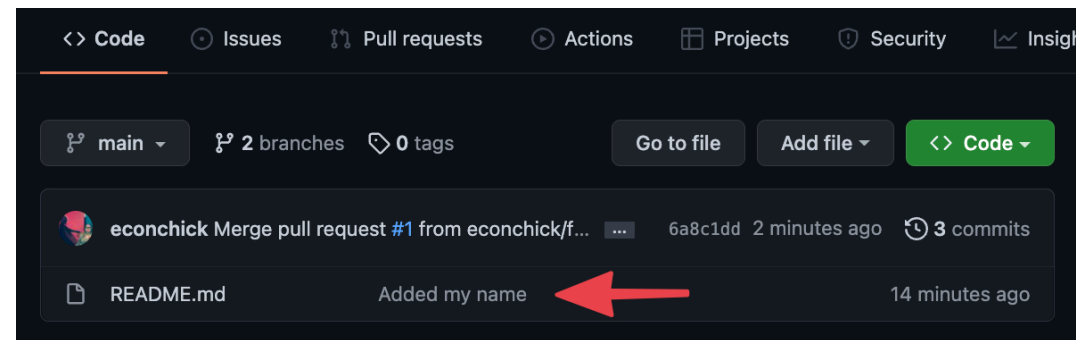
Step 20: Review our repo, part 1

Let's go back to the overall repository view by clicking on "Code":



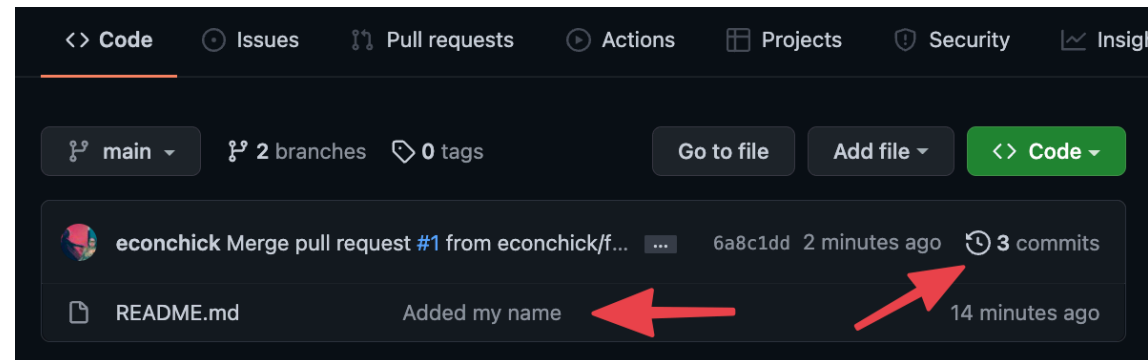
Step 21: Review our repo, part 2

We can see our commit and the message:



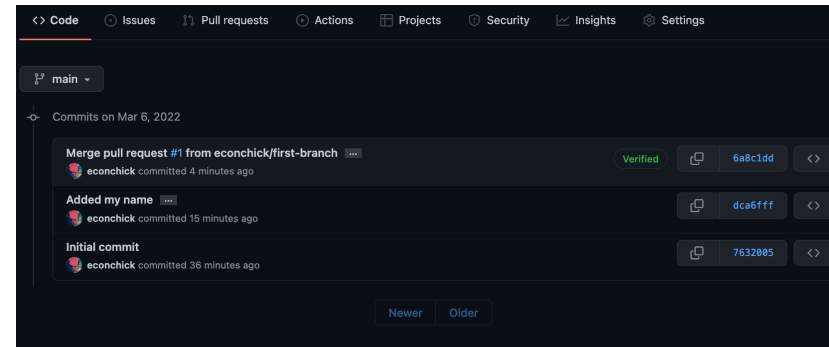
Step 22: Review our commits, part 1

We also see that there are 3 commits total in this repository. If we click on the "3 commits":



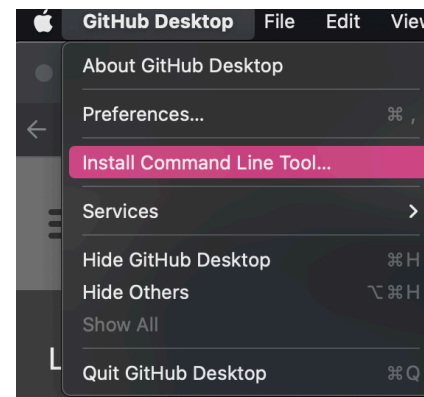
Step 23: Review our commits, part 2

We get a view of all the commits made to this repository (most recent is first). The very first one when GitHub desktop created the repo for us; the one that we made; and the one that was automatically generated when we merged our pull request (merging our branch into the `main` branch).



At Home: setup Git CLI

Two ways to install, either install the git CLI via GitHub Desktop:



At Home: setup Git CLI

Or through the “Install Git” section of this tutorial:

<https://www.atlassian.com/git/tutorials/install-git>

At Home: learn Git CLI

Follow the following tutorials from Atlassian:

Getting Started: <https://www.atlassian.com/git/tutorials/setting-up-a-repository>

Collaborating: <https://www.atlassian.com/git/tutorials/syncing>

Appendix: Resources

Theory

- The Missing Semester of Your CS Education - [Version Control](#) (from MIT)
- [About Git](#) (from GitHub)

Practice

- [Visualizing Git](#) - limited command functionality but great for helping visualize what's going on

General

- [git](#) documentation
- [Pro Git](#) (official git book)
- [git for the lazy](#)