Welcome to the June 2022 Reproducible Research Practices Workshop!

10 June 2022
Childhood Cancer Data Lab
https://alexslemonade.github.io/reproducible-research/

Alex's Lemonade Stand Foundation for Childhood Cancer

Childhood Cancer Data Lab
ALSF

Tell us about you!

- What’s your name?
- What are you studying?
- What summer activity are you looking forward to most?

Meet your instructors

JOSH

Joshua Shapiro

he/him/his

Data Scientist @ the Data Lab

PhD Ecology & Evolution, *UChicago*
Postdoc Integrative Genomics, *Princeton*

Research interests:

- **Evolutionary genomics**
- **Single cell workflows**

jashapiro

Meet your instructors

STEPHANIE
Stephanie Spielman
she/her/hers

Data Scientist @ the Data Lab

PhD Integrative Biology *UT Austin*
Postdoc Computational Molecular Evolution *Temple*

Research interests:

- **Protein evolution and phylogenetics**
- **Data science and bioinformatics education**

sjspielman

Meet your instructors

CHANTE

Chante Bethell

she/her/hers

Biological Data Analyst @ the Data Lab

Bachelor's in Bioinformatics from *Rowan University*

Research interests:

- **Functional motifs in the proteome**

cbethell

Meet your instructors

Ally

Ally Hawkins
she/her/hers

Data Scientist @ the Data Lab

PhD Cancer Biology *University of Michigan*
Postdoc Computational Biology *Cornell*

Research interests:

- **Single cell data analysis**
- **Origins of pediatric solid tumors**

allyhawkins

Meet your instructors

JACLYN

Jaclyn Taroni

she/her/hers

Director @ the Data Lab

PhD Genetics *Dartmouth*
Postdoc Computational Biology *UPenn*

Research interests:

- **Transcriptomics in rare, complex diseases**
- **Unsupervised pattern extraction**

jaclyn-taroni

Other staff you will see (or have seen already!)

JEN

Jen O'Malley
she/her/hers

Scientific Community Manager

- Promotes our resources to the community and helps administer offerings, like this workshop!
- Manages Data Lab communications

DEEPA

Deepa Prasad
she/her/hers

User Experience Designer

- Talks to researchers about their needs and frustrations
- Designs usable software

Single-cell Pediatric Cancer Atlas (ScPCA) Portal

- In 2019, ALSF funded 10 projects focused on single-cell and single-nuclei profiling with the goal of creating a publicly available atlas of pediatric cancer data.
- The Data Lab uniformly processes the generated data with an open source pipeline, and releases it through the portal. This growing database currently has 189 samples representing 28 cancer types.
- **Deepa is conducting usability testing of our data processing pipeline with community members** to ensure that external researchers can successfully run their own data!
- If you're interested in participating in testing to help us enhance this product, please let us know.

Explore the portal here:
https://scpca.alexslemonade.org/

Tell us about you!

- What’s your name?
- What are you studying?
- What summer activity are you looking forward to most?

Code of Conduct

Be kind, have fun

We value the involvement of everyone in the community. We are committed to creating a friendly and respectful place for learning, teaching, and contributing.

- Use welcoming and inclusive language.
- Be respectful of different viewpoints and experiences.
- Gracefully accept constructive criticism.
- Focus on what is best for the community.
- Show courtesy and respect towards other community members.

Read the full Code of Conduct:
https://alexslemonade.github.io/reproducible-research/code-of-conduct.html

If you at any time feel harassed or treated inappropriately, please contact ccdl@alexslemonade.org.

COVID-19 Safety Policy

With the goal of keeping everyone safe, healthy, and comfortable, we will comply with ALSF's COVID-19 safety policy during the workshop.

- Everyone is required to wear a surgical or respirator mask in the meeting room. Extra surgical masks are available!
- Hand sanitizer is available and HEPA filters are being used.
- Weather permitting, we will have our lunch break outside.
- If you are not feeling well, please do not stay for the entirety of the workshop. We will not withhold refunds or reimbursements if you must leave for this reason.

Read the COVID-19 policy and other participant information here:
https://alexslemonade.github.io/reproducible-research/participant_information.html

What you will learn (and what you won’t)

What you will learn

We will cover some common practices for reproducible computational research, including:

- Organizing your projects, including data, code, and documentation
- Navigating your computer from the command line interface
- Tracking and automating your work with scripts
- Making your code more readable, robust, and reusable
- Maintaining and tracking changes in your projects and code over time with git and GitHub
- Managing and tracking software and package versions for improved reproducibility

Our overarching goals: To introduce principles and techniques to achieve reproducible results in computational cancer research and to show you commonly-used approaches that you can apply to increase the impact of your research!

What you won’t learn

We won’t have time to cover:

- How to program in specific languages such as R or Python
- All the features and foibles of git and GitHub
- Workflow management systems such as CWL, Snakemake, or Nextflow

Schedule

9:00 - 10:15 am	Welcome Why does reproducibility matter?
10:15 - 10:45 am	Project Organization
10:45 - 11:00 am	Coffee break
11:00 - 11:30 am	Intro to UNIX and the command line
11:30 - 12:00 pm	Intro to Git Forking and cloning a repository
12:00 - 1:00 pm	Lunch
1:00 - 1:45 pm	Shell scripting
1:45 - 2:15 pm	The Git Stage/Commit/Push workflow `.gitignore` files
2:15 - 2:30 pm	Coffee break
2:30 - 3:15 pm	Organizing code in scripts and notebooks
3:15 - 3:45 pm	Managing packages and environments
3:45 - 4:15 pm	Working with branches in Git
4:15 - 5:00 pm	Open discussion and adjourn
7:00 pm	Optional dinner

Full schedule:
https://alexslemonade.github.io/reproducible-research/workshop-schedule.html

We want your feedback!

- Keep an eye out for our survey, which you will receive via email after the workshop ends. We value your honest feedback!

Stay in touch!

- You have been added to the `#reproducible-research-practices` Slack channel.
- Post public questions, share tips, make comments, and help your fellow participants.
- We encourage you stay in touch with the Data Lab team and others on Slack after the workshop ends!

Why does reproducibility matter?

Childhood Cancer Data Lab

Have you ever had problems reproducing…

- Someone else's research?
- Your research?
- Both?

Reproducibility vs replication

- Reproducibility
  - Authors provide all the necessary data and computer code to re-run the analysis and re-create the results
  - The exact same data/code are used to re-derive the exact same results
- Replication
  - A separate study arrives at the same scientific findings as another study
  - New data/code and analyses are performed that identify consistent results with previous work

	Same data	**Different data**
Same methods	*Reproducibility*	*Replicability*
Different methods	*Robustness*	*Generalizability*

Adapted from https://www.repro4everyone.org/resources/intro-reproducibility

Why do we like reproducible science?

- Reproducibility supports…
  - You!
  - Your collaborators and team!
  - Your community!
  - The scientific endeavor!

- Reproducibility makes your funders and journals happy
  - Requirements are increasing across the board

What are your barriers to reproducible science?

Some barriers others have noticed…

- Technical factors
  - Natural variation
  - Batch effects and sensitivity to equipment or experimental conditions
- Study design and statistics
  - Selective reporting and "P-hacking"
- Human factors
  - Confirmation bias
  - Poor record keeping and/or sharing
- Rewards and incentives
  - Hypercompetition and "publish or perish" culture
  - Paywalls

Adapted from https://www.repro4everyone.org/resources/intro-reproducibility

Lab notebooks in computational biology

- It feels weird to write about what you coded!
  - But it doesn't feel weird to write about your pipetting steps.
  - *But why should we treat these differently?*

Always 👋 Be 👋 Documenting

- Science on the computer is just like science at the bench - you need to *record* your steps, even if they seem obvious at the time!
  - If you'd write it down in a lab notebook, you should write it down for your coding as well
  - Write down things that didn't work too!
  - How do you write it down?

Future you is unimpressed with your code

- AKA, "Future You thinks Present You is an idiot" -Stephanie

- We tend to feel code we are writing is clear and obvious *in the moment we are writing it*
  - Future You seriously disagrees!

How can you appease Future You?

- Comment, comment, comment, COMMENT!
  - Comments are not a sign that you don't know what your doing
  - Comments are actually a sign that you are a more thoughtful (and advanced?) programmer
  - Strive to explain ***why*** you took each step, not just what the step was

- Document, document, document, DOCUMENT!
  - Documenting the work you're doing in README files is (part of) your lab notebook
  - Documenting your code can also help you gain insight into your code and find ways to improve it

Every little bit helps

- Everyone has to start somewhere!
- Nobody's code is perfect!

WORLD VIEW *A personal take on events*

Publish your computer code: it is good enough

Freely provided working code — whatever its quality — improves programming and enables others to engage with your research, says **Nick Barnes**.

https://www.nature.com/articles/467753a

Project Organization

Childhood Cancer Data Lab

Sources!

All ideas come from somewhere.

These are some sources that have inspired us (and provided material directly)

- Vince Buffalo: Bioinformatics Data Skills
- Jenny Bryan: https://speakerdeck.com/jennybc/how-to-name-files
- Danielle Navarro: https://slides.djnavarro.net/project-structure

Why does project organization matter?

- Finding things takes a lot of time and effort
- Standard and predictable organization saves time
- Be kind to yourself & others!
  - Make your stuff discoverable
  - Follow consistent patterns

But I can just search!

- Google has trained us to search for content, and searching is great! Sometimes.

```
@071112_SLXA-EAS1_s_7:5:1:817:345
GGGTGATGGCCGCTGCCGATGGCGTCAAATCCCACC
+071112_SLXA-EAS1_s_7:5:1:817:345
IIIIIIIIIIIIIIIIIIIIIIIIIIIII9IG9IC
@071112_SLXA-EAS1_s_7:5:1:801:338
GTTCAGGGATACGACGTTTGTATTTTAAGAATCTGA
+071112_SLXA-EAS1_s_7:5:1:801:338
IIIIIIIIIIIIIIIIIIIIIIIIIIIIII6IBI
```

- File names can be uninformative (we'll come back to that!)
- Data files often don't have searchable content
  - Even when it does, you might not know what to search for!
- **Metadata** describing the content might not be part of the file

REPORT

FILE NOT FOUND

A generation that grew up with Google is forcing professors to rethink their lesson plans

By Monica Chin | @mcsquared96 | Sep 22, 2021, 8:00am EDT

Illustrations by Micha Huigen

https://www.theverge.com/22684730/students-file-folder-directory-structure-education-gen-z

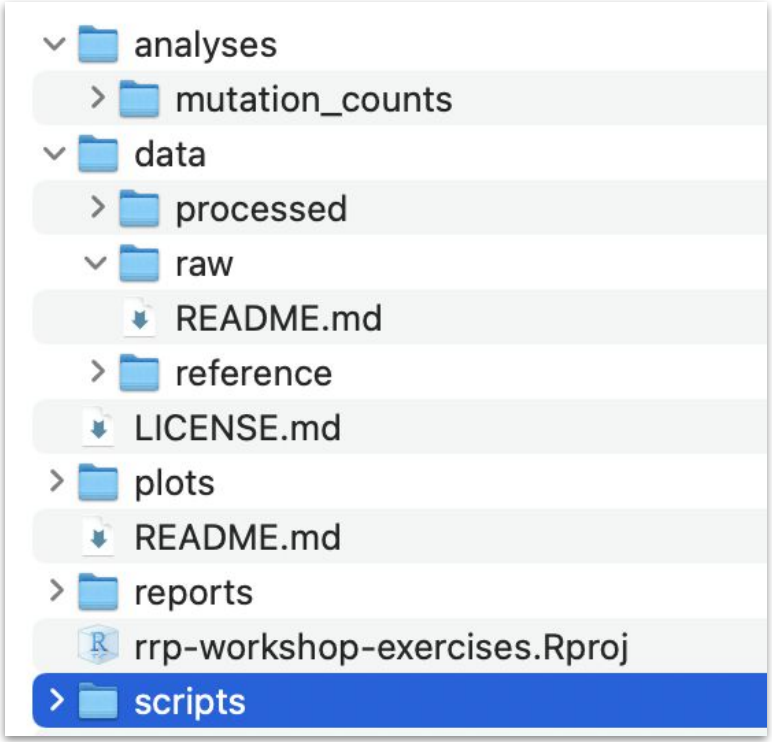
Where to start?

- **Use Folders/Directories!**

- Keep separate projects separated
- Separate sections for units within a project
  - Data
  - Code
  - Results
  - Reports

- **Documentation throughout!** ABD
  - Describe what files do and how they are organized

A typical project folder (for me!)



You'll see more of this project later!

The `data` folder

- This is where the big files go
- Often contains a `raw` subfolder
  - Files that came from external sources, untouched
- Maybe a separate subfolder or subfolders for `processed` files
  - trimmed, filtered, concatenated, etc.
- Use sub-subdirectories for organization:
  - by processing stage
  - by date
  - by sample
  - *Consider*: it is often easiest to process ***all*** the things in a folder together; organize by units of work

Spend time thinking about this organization! It will reward you later.

The Naming of [Files] is a difficult matter

(apologies to T. S. Eliot)

Some “good” file names:

- `cheese-ratings.tsv`
- `2022-02-12_cheeseshop-inventory.tsv`
- `01_compile-ratings.py`

Some “not so good” file names:

- `script.py`
- `AVRS638GVEW4.fastq.gz`
- `tastingnotes (3).docx`
- `My favorite cheeses FINAL3 for Anatole UPDATED.docx`

Jenny Bryan’s principles for file naming (modified)

- machine friendly
- human friendly
- sortable and computable

Machine friendly

- Avoid spaces
  - Old computer systems get confused by spaces
  - All computer systems are old underneath
  - Use underscores or dashes to separate words instead
- Use “standard” characters:
  - Letters, numbers, underscores, and dashes
  - Periods only for file extensions (`.txt`, `.tsv`, `.R`, `.tar.gz`)
  - Many characters have special meanings in code. Avoid them! (e.g. `* + ? | $ / “`)
- Be consistent with case
  - Don’t *assume* case has meaning: on some systems it does, and on some it doesn’t
  - But always *act* as if it does!
    - Never have two files that are the same but for case

Human friendly

- Names should contain information about file content
- Short names are tempting, but you may regret choosing them!
  - `01.R`
  - `data.txt`
  - `tests.py`
- Use long descriptive names
  - `01_download-ena-data.sh`
  - `fig01_penguin-weight-histogram.png`

Which files do you want to look for before a deadline?

Which files do you want to get from your collaborator?

Sortable

- Use numbers for consistent sorting
  - `fig01_project-overview.pdf`
  - `fig02_sample-descriptor-histogram.pdf`
  - `fig03_oncoprint.md`
  - Left pad with `0` for consistent number length; this helps the computer sort properly
    - `7` is sad when it gets sorted after `11`
- Dates: use ISO 8601
  - Year-month-day is unambiguous and sorts nicely!
  - `2000-05-04_jedi-council-attendance.tsv`
  - `2000-05-05_sith-council-attendance.tsv`

https://xkcd.com/1179/

Computable

- Use consistent name formats
  - Use file extensions
  - Separate “chunks” with underscores & keep consistent order

- “Wildcards” will be your friend:
  - `*` is the most common wildcard in UNIX
  - `*.txt`: refers to all files that end with `.txt` (hopefully all text files)
  - `2020-01-*`: all of the files from January 2020

Files you didn’t create

- All the guidelines and suggestions for file names are great for files you create, but sometimes files come from other sources
  - If you are lucky, they will follow nice conventions! 🎉
  - but often they won’t □

- To rename or not to rename, that is the question
  - Leaving the name as it was sent can make it easier to track in correspondence
  - Reasons to rename:
    - uniformative generic names: `data.txt`
    - add source or date information
    - converting spaces or other special characters (but try to write code that can handle these!)
  - *If you choose to rename, do it with a script and document the original name and source*

Reproducibility standards for machine learning in the life sciences

To make machine-learning analyses in the life sciences more computationally reproducible, we propose standards based on data, model and code publication, programming best practices and workflow automation. By meeting these standards, the community of researchers applying machine-learning methods in the life sciences can ensure that their analyses are worthy of trust.

Benjamin J. Heil, Michael M. Hoffman, Florian Markowetz, Su-In Lee,
Casey S. Greene and Stephanie C. Hicks

Table 1 | Proposed reproducibility standards

	Bronze	Silver	Gold
Data published and downloadable	x	x	x
Models published and downloadable	x	x	x
Source code published and downloadable	x	x	x
Dependencies set up in a single command		x	x
Key analysis details recorded		x	x
Analysis components set to deterministic		x	x
Entire analysis reproducible with a single command			x

https://www.nature.com/articles/s41592-021-01256-7

ONE CUP AT A TIME CLUB

But remember - It's an honor just to compete!

UNIX and the Command Line

Childhood Cancer Data Lab

Why learn UNIX and use the command line?

- The command line is, in part, the computational biologist's "lab notebook"
- You can capture small data manipulation steps that are normally not recorded to make research reproducible*
  - Manual manipulation of data files is challenging to troubleshoot, review, or improve*
  - *Don't take the "small steps" for granted! They matter a lot.*
- With UNIX, you can record all your precise steps - you have typed them rather than "point-and-clicked" them
  - Even better, you can run these steps as a script! Stay tuned…

*https://swcarpentry.github.io/shell-novice/guide/

Why learn UNIX and use the command line?

- Allows you to automate repetitive tasks*
  - Much less risk of human error, and *much easier* to re-run
  - You will thank yourself for putting in the time up front to write scripts that automate!

- Most bioinformatics/NGS tools are command-line only
- Take your science back into your own hands!
  - Running these tools yourself removes the mystery of what your core/bioinformatics collaborator is doing

*https://swcarpentry.github.io/shell-novice/guide/

Terminology

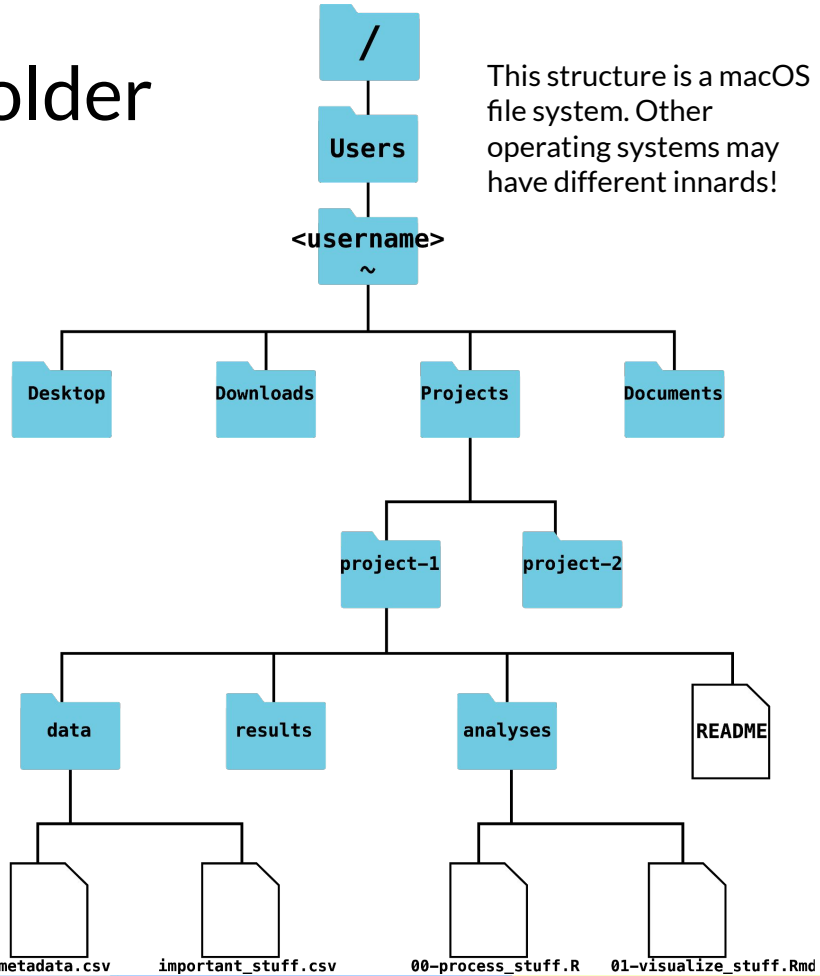
- UNIX is an *operating system* which features a hierarchical file system and provides commands (little computer programs) to help interact with it
  - We are using the term “UNIX” to refer to all UNIX-like systems like Linux and macOS

- The shell allows us to interact with UNIX and run those commands
  - We access the shell through the terminal (also known as command line)
  - There are many different shells out there, including BASH and zsh (pronounced "zee shell")*

- We use shell scripting languages to write code that the shell can interpret and execute
  - Extremely creatively called, for example, BASH scripting or zsh scripting

Directory is a fancy word for folder

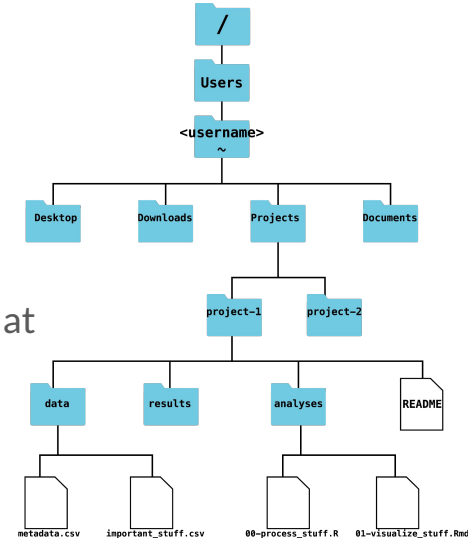
- Files and directories in most operating systems (including UNIX-like!) are organized hierarchically
- Every file/directory has a specific address, or **path**, in the hierarchy
- The top-level directory is known as **root** and denoted **/**
- Your home directory can be referred to with **~**



This structure is a macOS file system. Other operating systems may have different innards!

Relative vs. absolute paths

- The path is the address to a file or directory on your computer. There are two ways to formulate paths:
  - The absolute (or full) path represents the path of a given directory or file, *beginning at the root directory*
    - Because they begin at the root, absolute paths always start with /
  - The relative path represents the path of a given directory or file, beginning at (i.e. *relative to*) to the working/current directory 



- For example, the full path to 01-visualize-stuff.Rmd:
  - `/Users/username/Projects/project-1/analyses/01-visualize-stuff.Rmd`

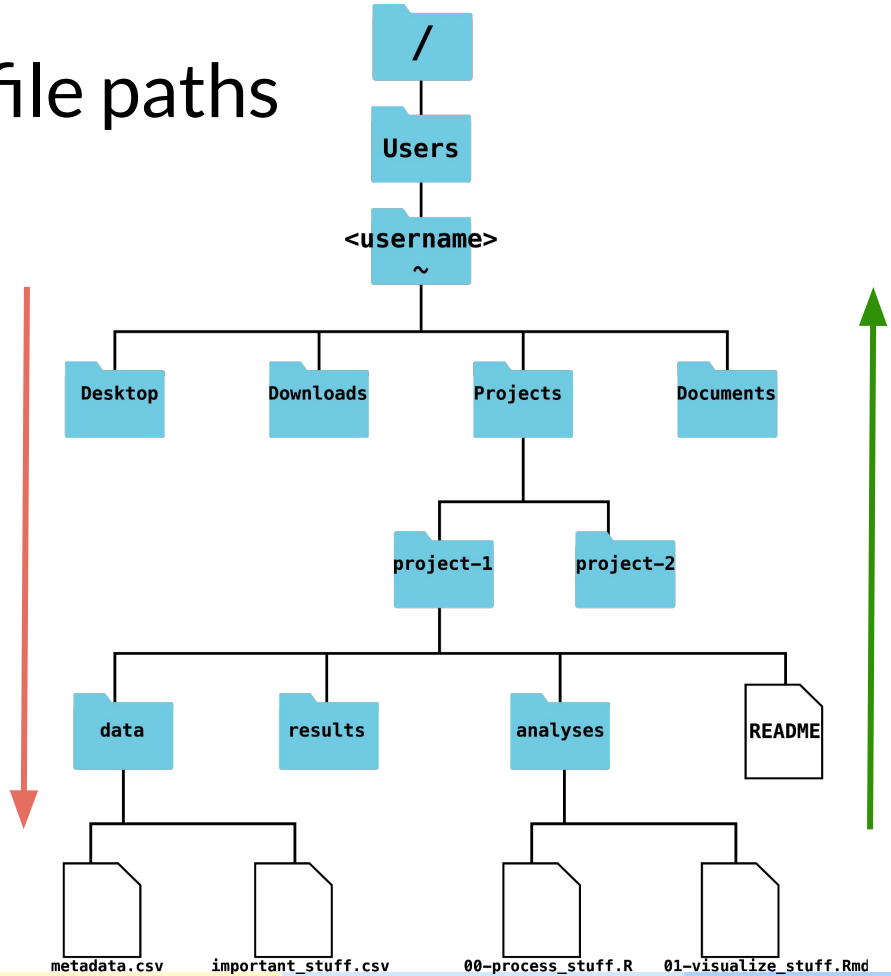
Paths analogy: How do I get to the store?

- The absolute path: 15 Main St, Anytown, Anystate, zip code 12345
  - If you have this address, you can find your way to this store from anywhere in the world

- The relative path: Make a left, walk 5 blocks, make a right, and then the store will be on your left.
  - With these directions, you can only find your way to this store from the exact location where you started (i.e, *relative to* your starting location)

Writing out file paths

We move **forwards (down)** in the hierarchy with a forward slash /



We move **backwards (up)** in the hierarchy with two dots **..**

Determining file paths

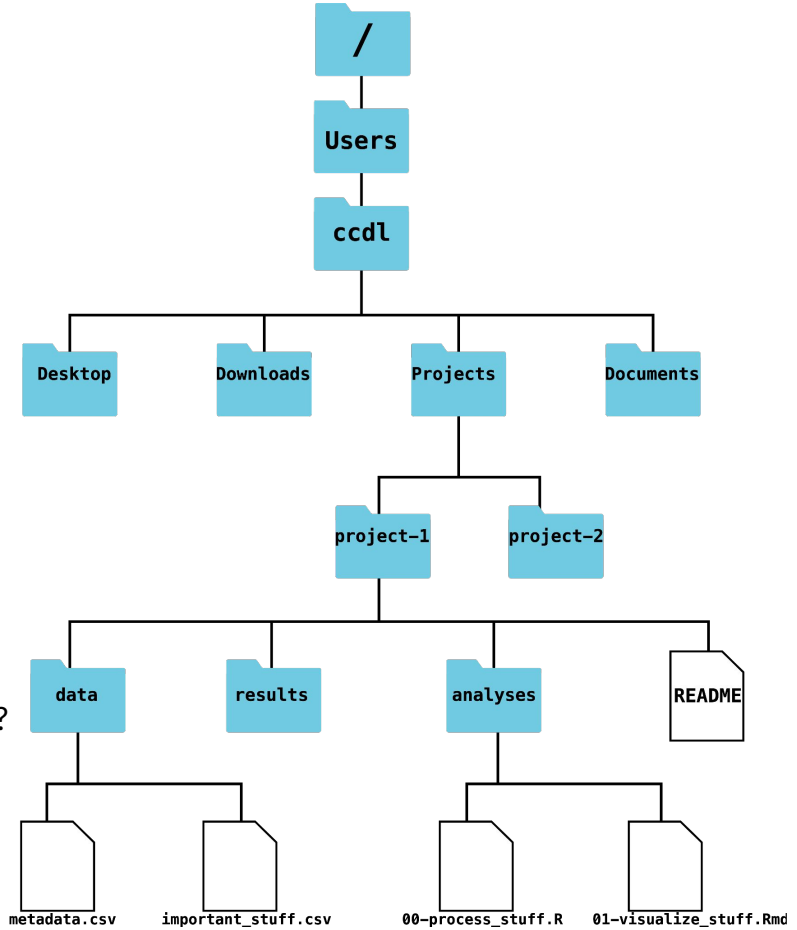
What is the **absolute path** to the project-1 directory?

What is the **relative path** to project-1 from the home directory?

What is the **relative path** to important_stuff.csv from the home directory?

What is the **relative path** to important_stuff.csv from the **analyses directory**?

What is the **relative path** to important_stuff.csv from the **data directory**?



Determining file paths

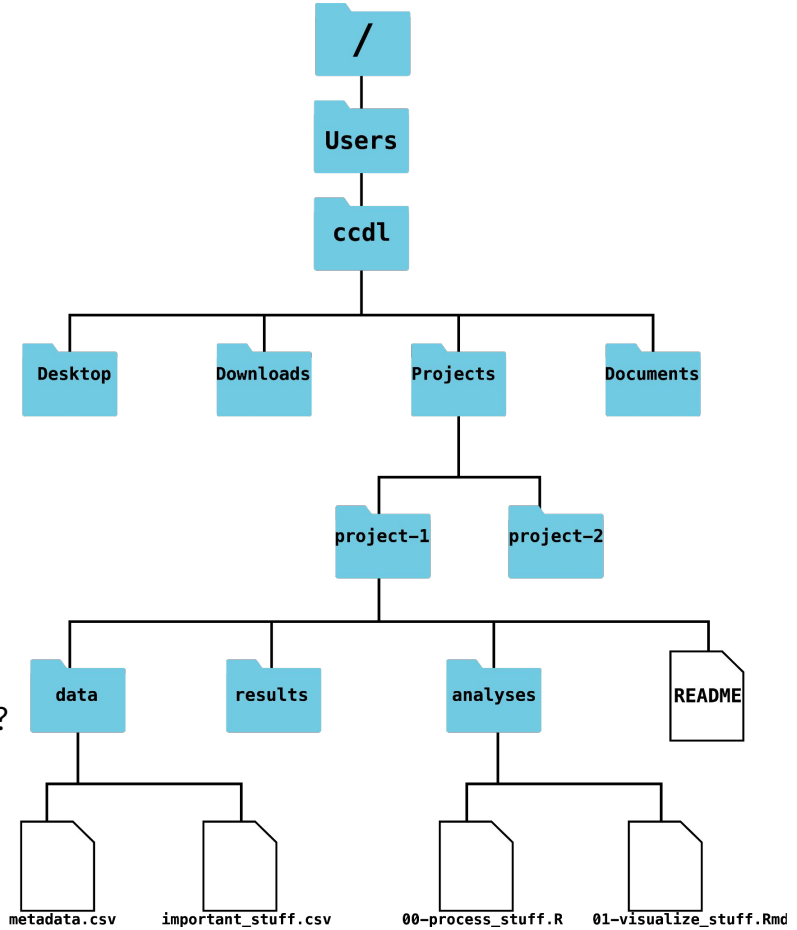
What is the **absolute path** to the project-1 directory?
`/Users/ccdl/Projects/project-1`
`~/Projects/project-1`

What is the **relative path** to project-1 from the home directory?
`Projects/project-1`

What is the **relative path** to important_stuff.csv from the home directory?
`Projects/project-1/data/important_stuff.csv`

What is the **relative path** to important_stuff.csv from the **analyses directory**?
`../data/important_stuff.csv`

What is the **relative path** to important_stuff.csv from the **data directory**?
`important_stuff.csv`
`./important_stuff.csv`

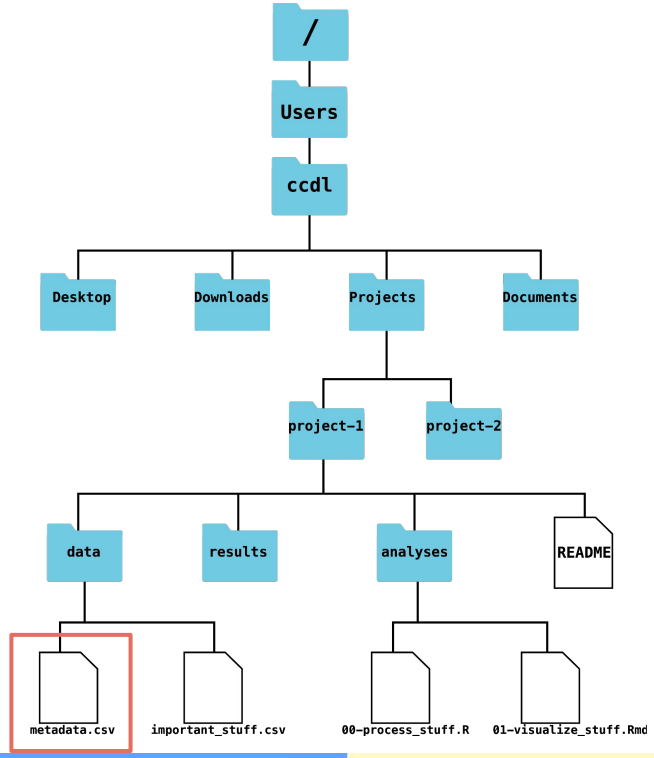


We prefer relative paths! Why do you think?

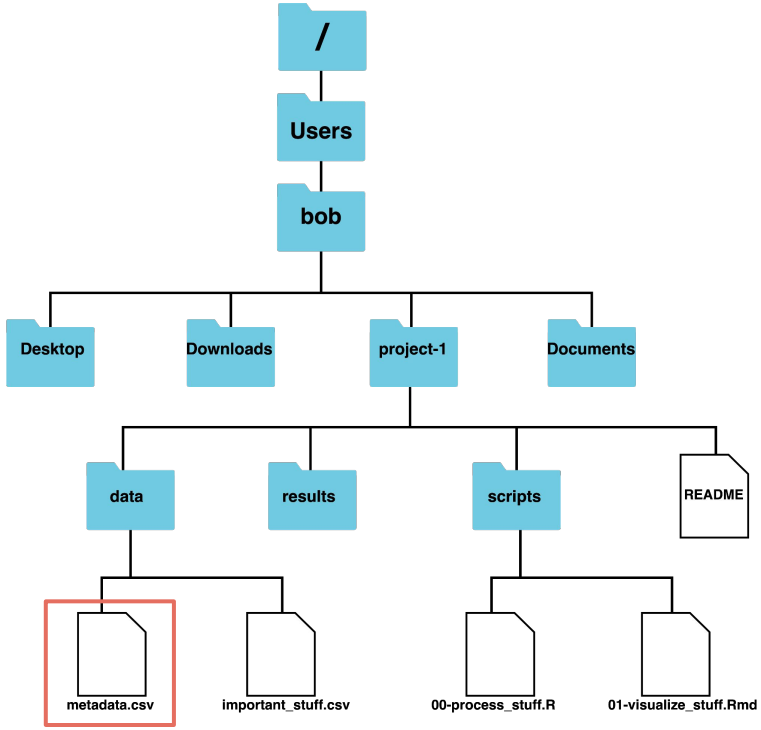
Consider each computer and think…

- What is the **absolute path** to metadata.csv?
- What is the **relative path** from scripts/ to metadata.csv?

A CCDL computer's file system



My collaborator Bob's file system



Let’s try out some UNIX commands

- All UNIX commands are actually little computer programs
  - The behavior of UNIX commands can also be modified with *flags*

Command	What it means	Why use it
`pwd`	**P**rint **w**orking **d**irectory	Figure out where you are on your computer?
`cd`	**C**hange **d**irectory	Move from directory A → B in your computer
`ls`	**L**i**s**t	List contents of a directory

- Note that UNIX is *case sensitive!* `CD` is not `cd`

Introduction to `git`

Childhood Cancer Data Lab

Why use version control?

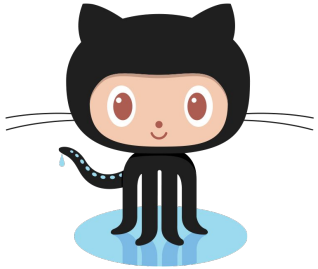
- `code-final.R, code-final-final.R, code-final-final-for-real-this-time.R`
- `file.R, file.R (1), file.R (2), file.R (3)`

- Provides a framework for collaborating, particularly on code
- Allows you to restore previous versions and compare between versions
  - Ever wish you could "Undo changes" to say, the version 2 months ago? Now you can!
  - Note: this functionality is *much* more powerful than “Track Changes” in Word or file recovery options in Google Drive

These days, we like to use `git` for version control

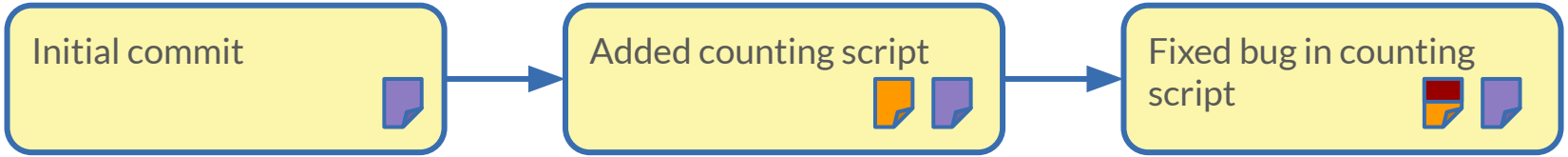
- **`git`** is an open-source software that performs version control
  - implemented as a set of command line tools
  - GitKraken is a GUI that (hopefully) makes `git` a bit easier to use
    - Other available GUIs: SourceTree, GitHub Desktop

- **github.com** is one of many websites that can host your **repositories**
  - Others include gitlab, bitbucket, gitbucket…
  - Using a host like GitHub means that you
    - have a backup at all times
    - can work on your code from different computers
    - collaborate with other more easily



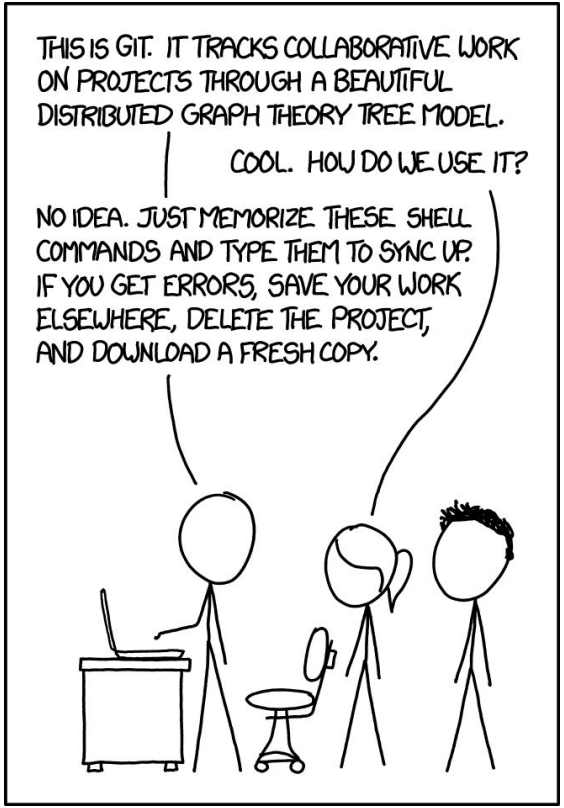
git terminology

- A **repository** is a directory that `git` is managing
  - It looks like a regular folder from your perspective
  - There can be multiple copies of the repository, often on different machines
- Files that `git` knows about are **tracked**, i.e., they are under version control
- We take snapshots called **commits** to record the state *and history* of tracked files



Words to the wise

- git can be tricky, and the feelings of frustrations you ~~may~~ will have are normal!
- You will make mistakes. *That's ok! So do we, and so does everyone else!*
- Remember: You have to start somewhere.



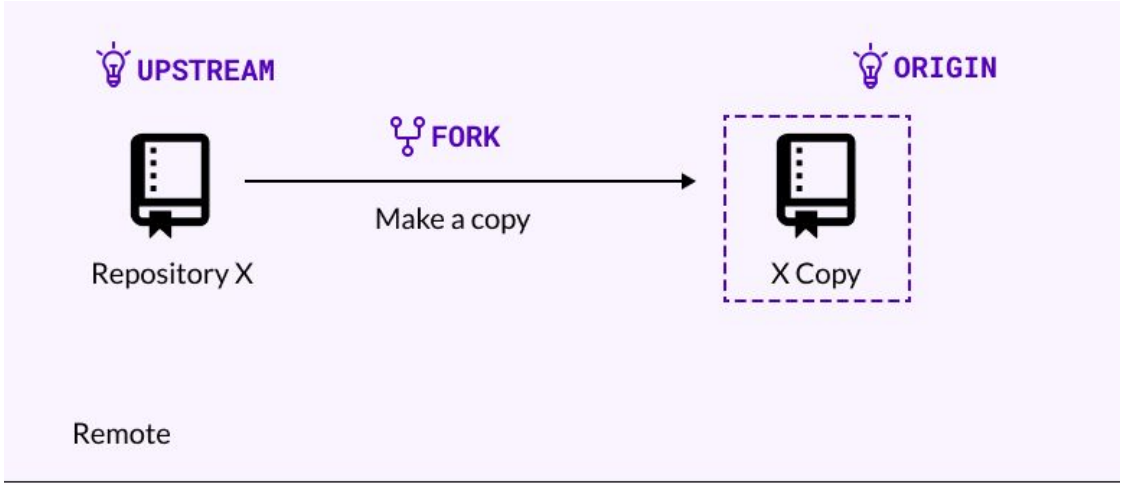
https://xkcd.com/1597/

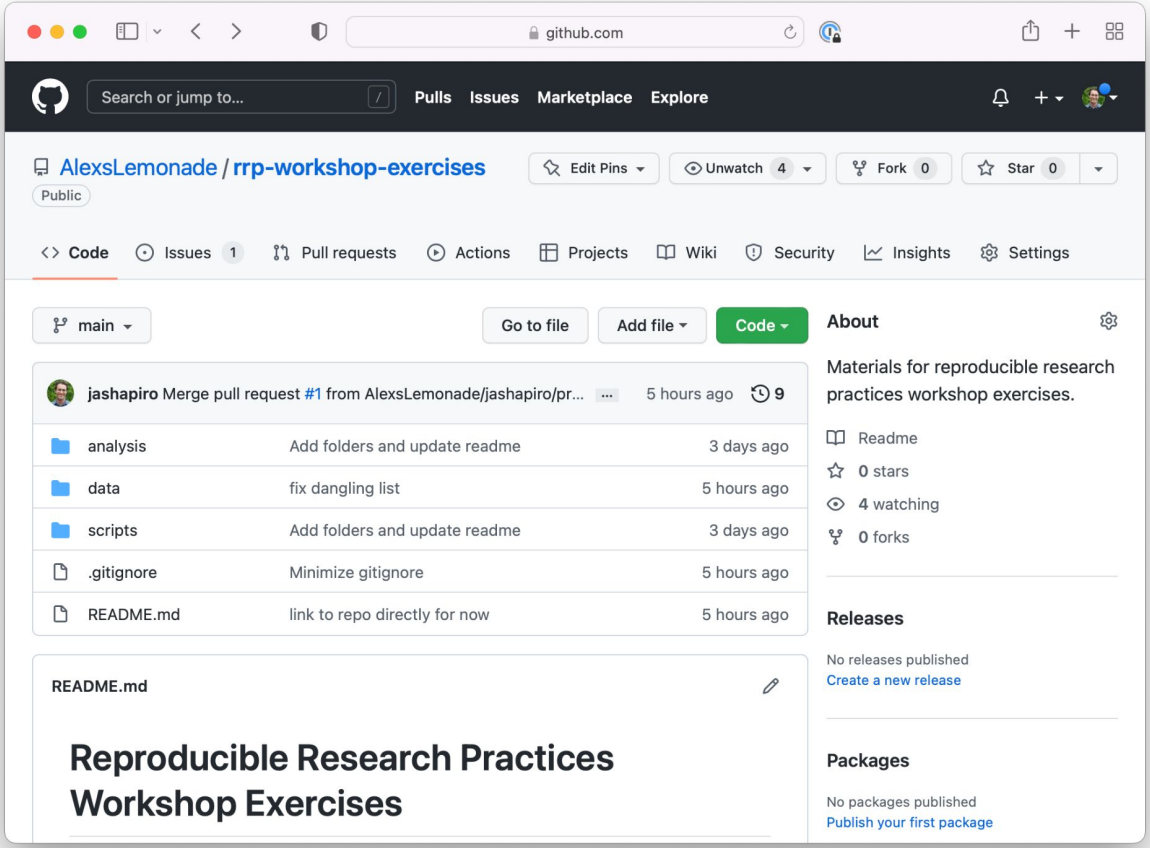
Forking a repository on GitHub

- Forking allows you to make ***your own*** copy of somebody else's repository
  - Anything that happens in the forked repository will not affect the "upstream" repository
  - You will be able to make whatever changes you want
  - Connections to what happens upstream will still exist, which can be handy

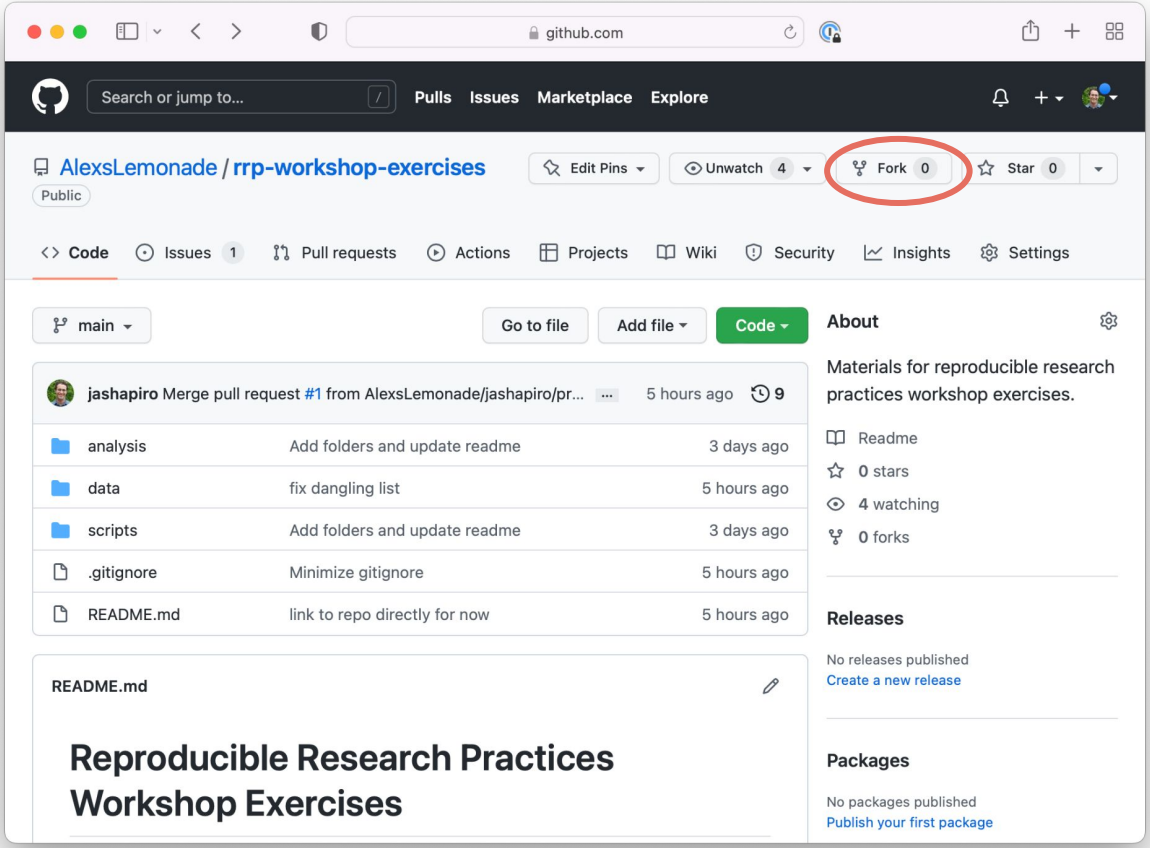
- Let's make a copy of the repository that holds examples for this workshop!

- Note that with your *own* project repositories, you would not need this step - you can just make a repository directly

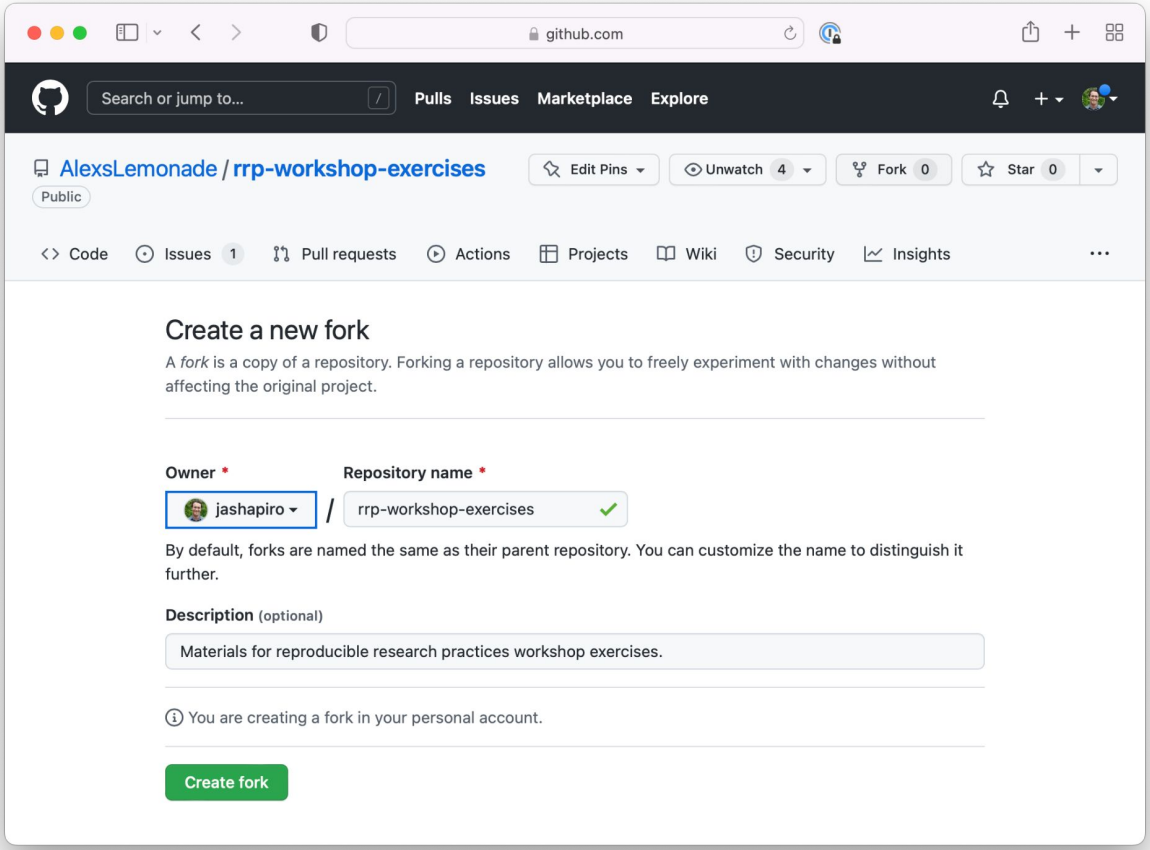

UPSTREAM
ORIGIN
FORK
Make a copy
Repository X
X Copy
Remote



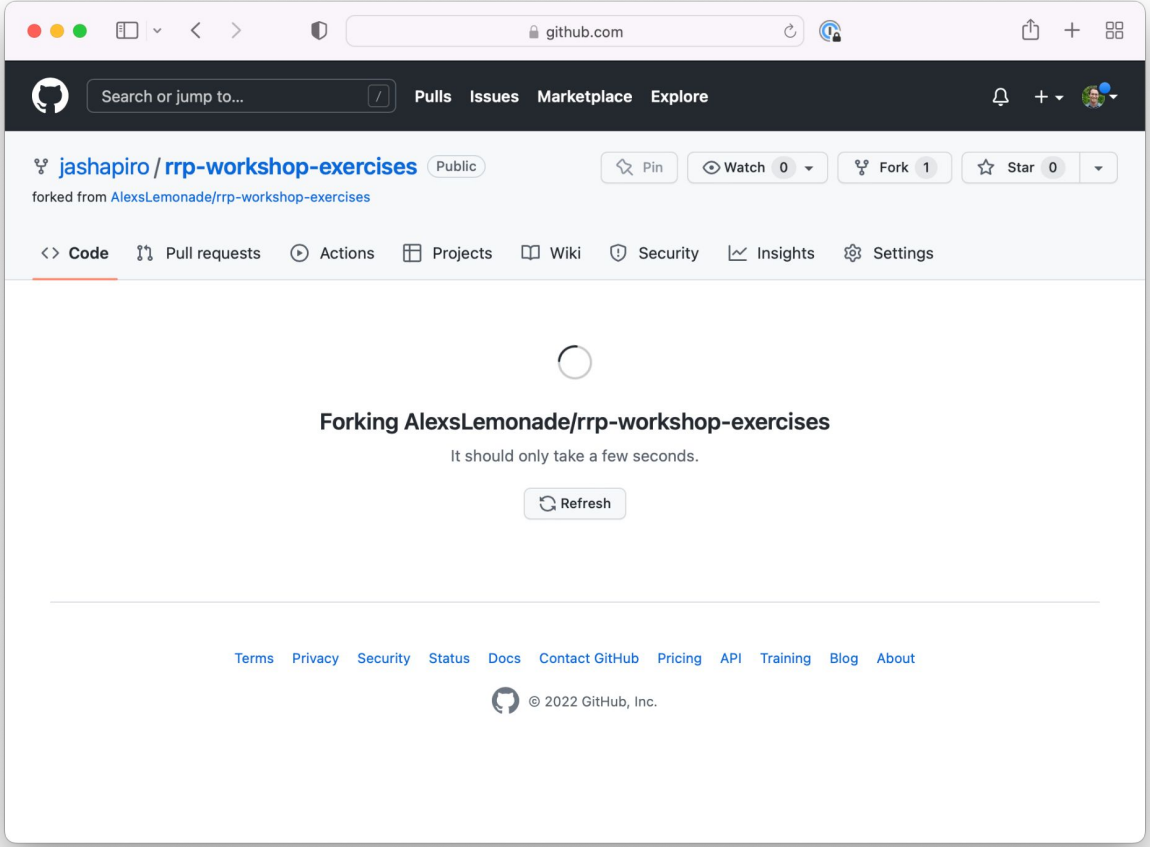
Go to https://github.com/AlexsLemonade/rrp-workshop-exercises
(make sure you are logged into GitHub)



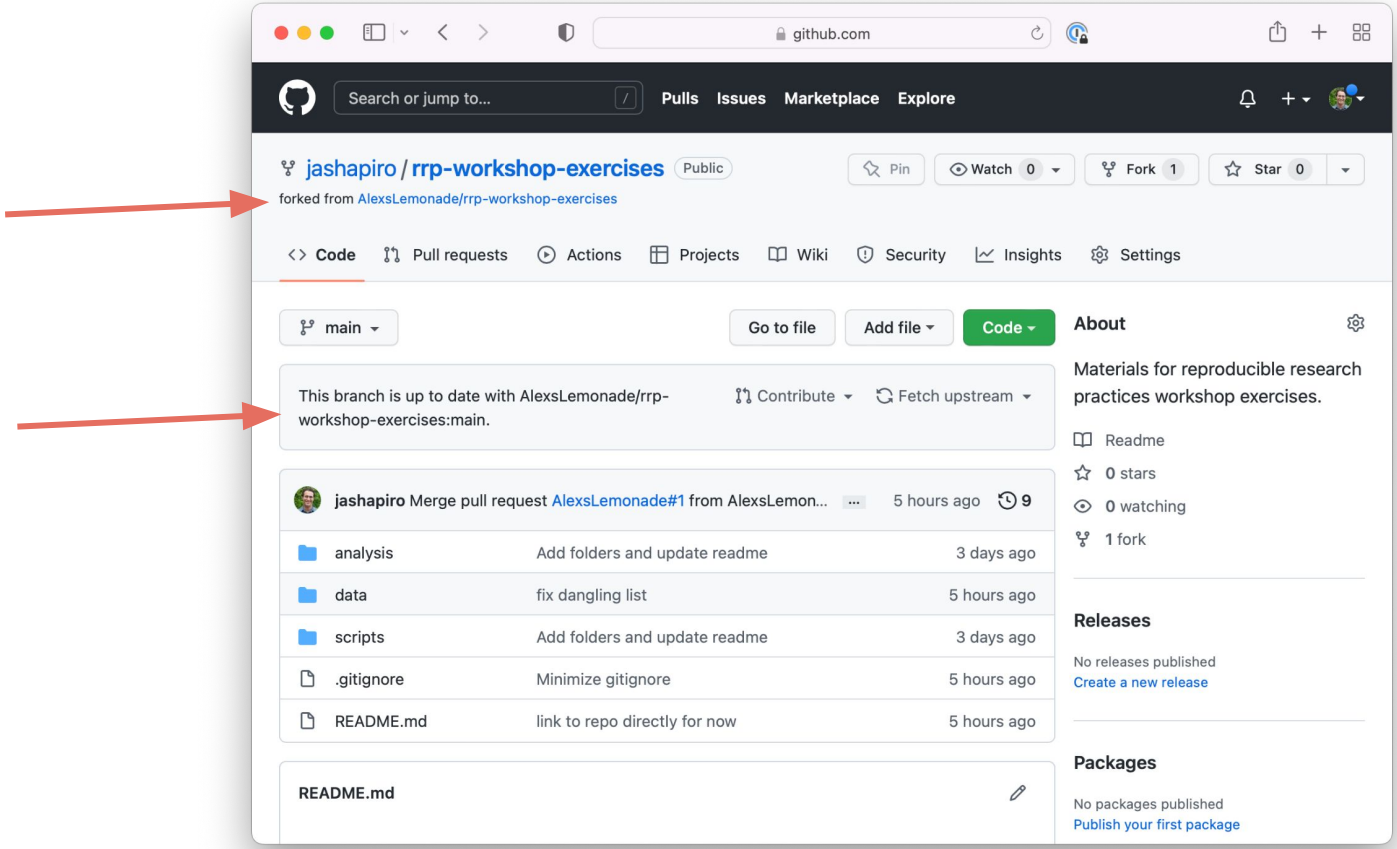
Click the “Fork” button



Change the name if you like, but probably not; click “Create fork”


github.com
Search or jump to...
Pulls
Issues
Marketplace
Explore
jashapiro / rrp-workshop-exercises
Public
forked from AlexsLemonade/rrp-workshop-exercises
Pin
Watch 0
Fork 1
Star 0
Code
Pull requests
Actions
Projects
Wiki
Security
Insights
Settings
Forking AlexsLemonade/rrp-workshop-exercises
It should only take a few seconds.
Refresh
Terms
Privacy
Security
Status
Docs
Contact GitHub
Pricing
API
Training
Blog
About
© 2022 GitHub, Inc.

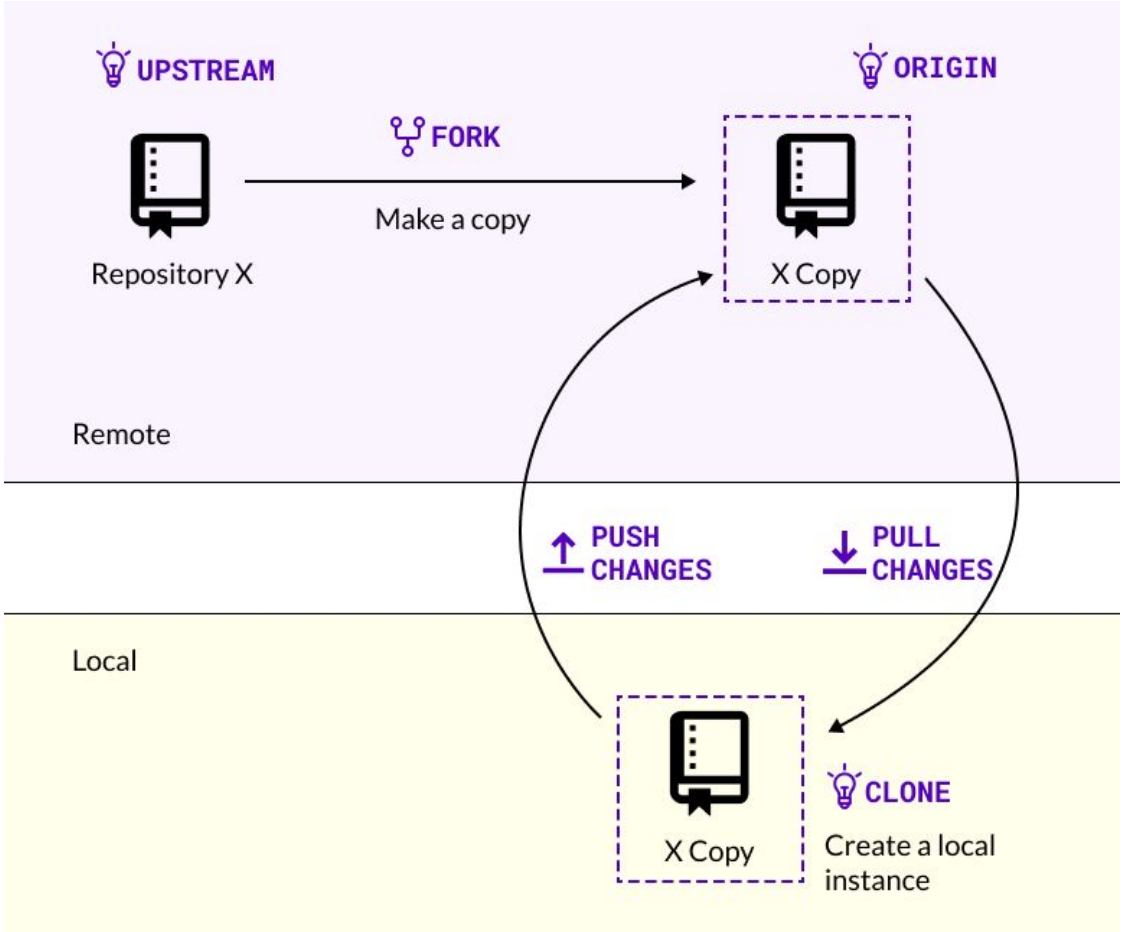
Wait…



Enjoy your new repo!

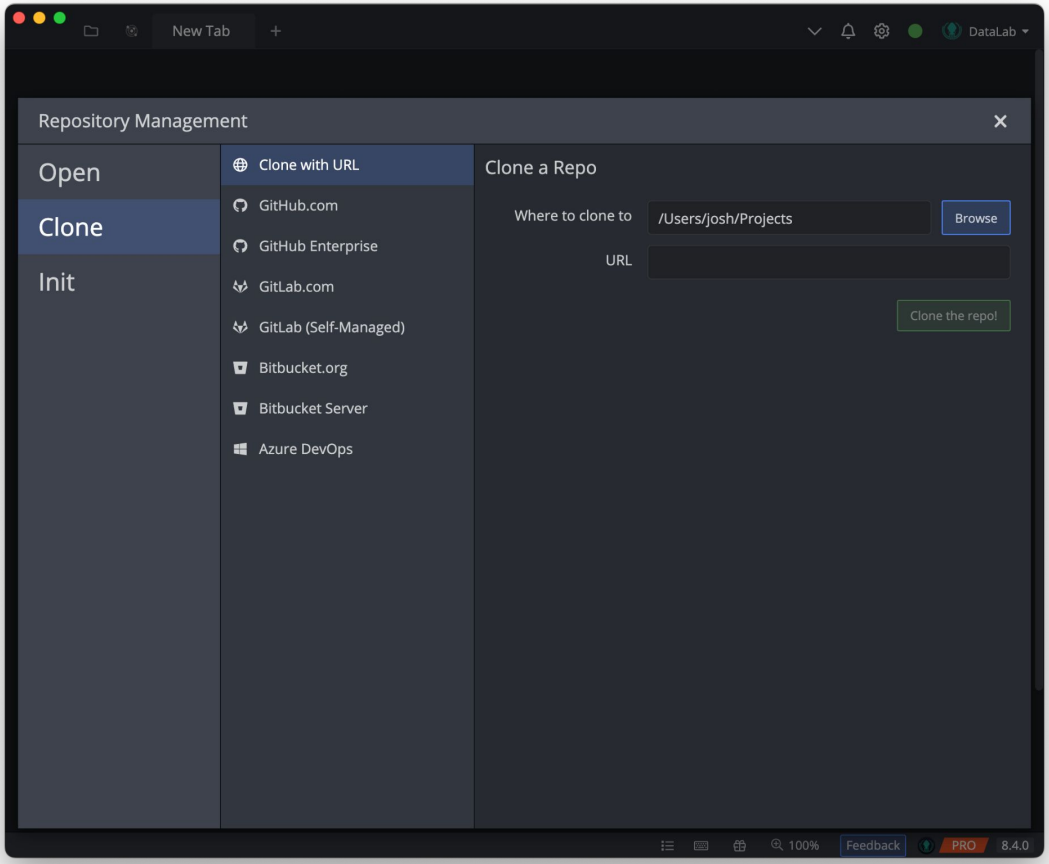
Cloning your new repository

- Having a repository on GitHub is great, but you really want a copy on your own computer!
  - This is true whether it is a repo you made "from scratch" or one you forked
- Cloning can happen at the command line or through a GUI like GitKraken


UPSTREAM
ORIGIN
FORK
Make a copy
Repository X
X Copy
Remote
PUSH CHANGES
PULL CHANGES
Local
X Copy
CLONE
Create a local instance

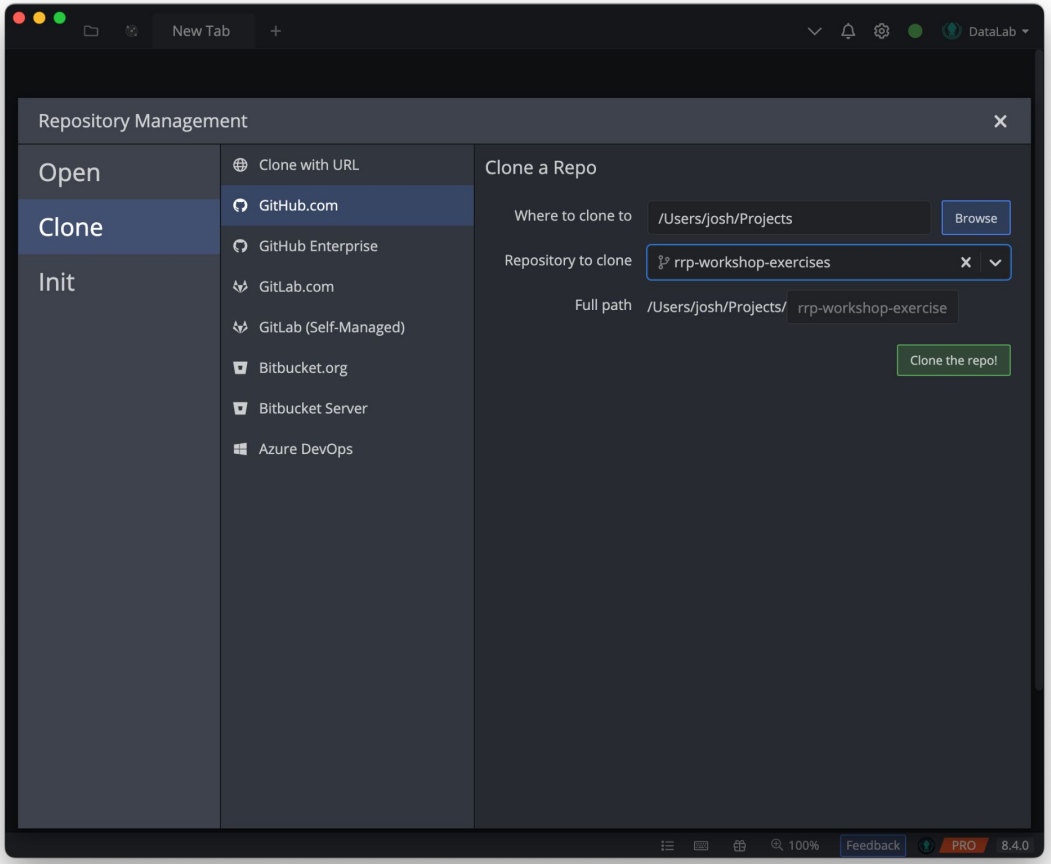
Cloning in GitKraken

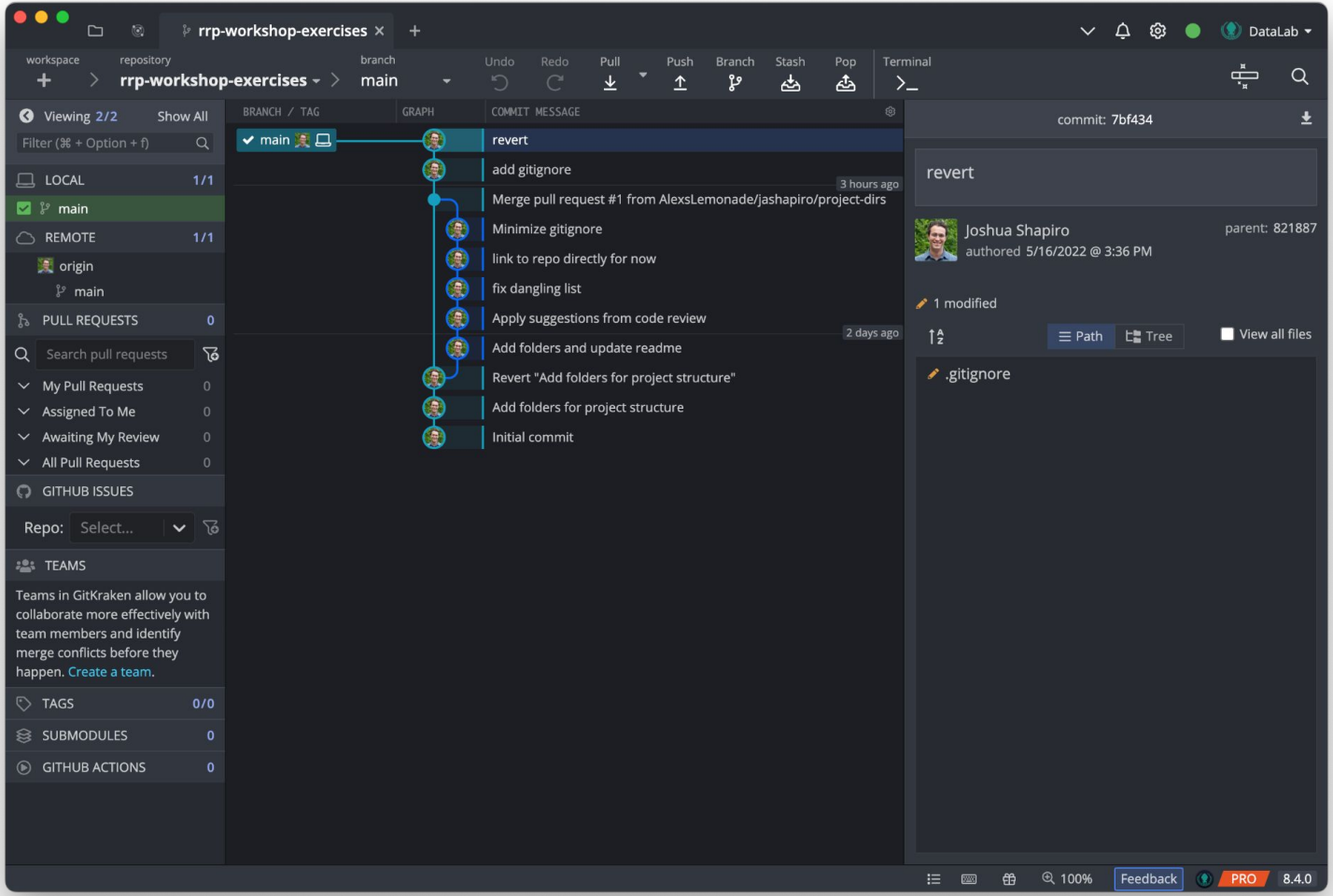
- Choose the destination
  - “Where to clone to”
- “Clone with URL” copied from GitHub



Cloning in GitKraken

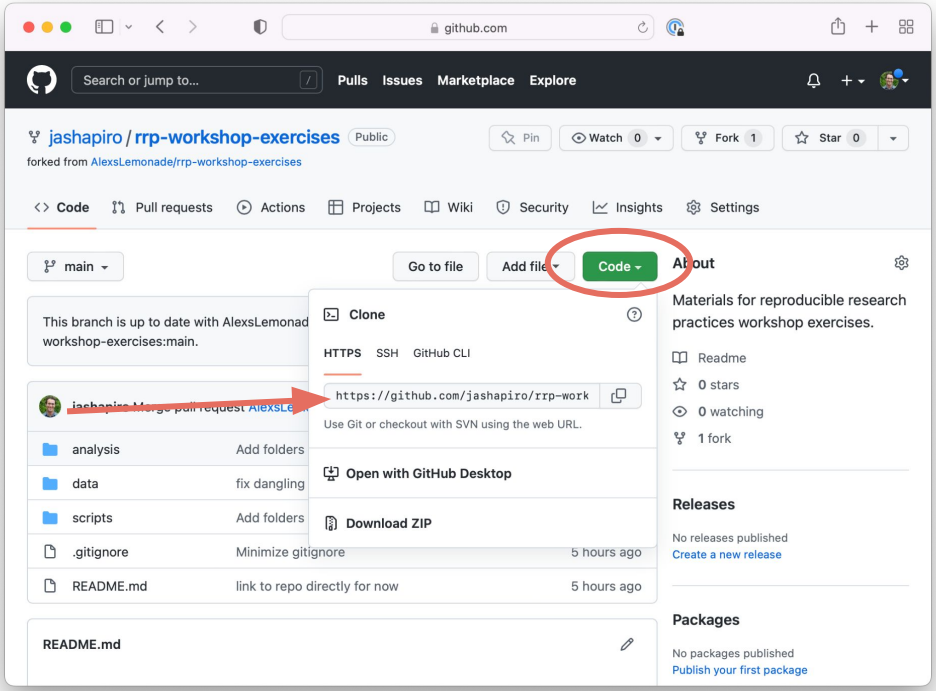
- Choose the destination
  - “Where to clone to”
- “Clone with URL” copied from GitHub
- OR select the GitHub.com repository from within GitKraken!




rrp-workshop-exercises
DataLab
workspace
repository
rrp-workshop-exercises
branch
main
Undo
Redo
Pull
Push
Branch
Stash
Pop
Terminal
Viewing 2/2
Show All
Filter (⌘ + Option + f)
LOCAL 1/1
main
REMOTE 1/1
origin
main
PULL REQUESTS 0
Search pull requests
My Pull Requests 0
Assigned To Me 0
Awaiting My Review 0
All Pull Requests 0
GITHUB ISSUES
Repo: Select...
TEAMS
Teams in GitKraken allow you to collaborate more effectively with team members and identify merge conflicts before they happen. Create a team.
TAGS 0/0
SUBMODULES 0
GITHUB ACTIONS 0
BRANCH / TAG
GRAPH
COMMIT MESSAGE
main
revert
add gitignore
3 hours ago
Merge pull request #1 from AlexsLemonade/jashapiro/project-dirs
Minimize gitignore
link to repo directly for now
fix dangling list
Apply suggestions from code review
2 days ago
Add folders and update readme
Revert "Add folders for project structure"
Add folders for project structure
Initial commit
commit: 7bf434
revert
Joshua Shapiro
authored 5/16/2022 @ 3:36 PM
parent: 821887
1 modified
Path
Tree
View all files
.gitignore
100%
Feedback
PRO
8.4.0

Cloning at the command line

- Find the repository URL and copy it:
  - Click the "Code" button
  - Copy the URL under "Clone"
  - the URL will end with a `.git` extension



Cloning at the command line

- Navigate to the location where you want to put the repository with `cd`
- `git clone <URL-from-GitHub>`

```
❯ cd ~/Projects
❯ git clone https://github.com/jashapiro/rrp-workshop-exercises.git
Cloning into 'rrp-workshop-exercises'...
remote: Enumerating objects: 31, done.
remote: Counting objects: 100% (31/31), done.
remote: Compressing objects: 100% (26/26), done.
remote: Total 31 (delta 8), reused 19 (delta 1), pack-reused 0
Receiving objects: 100% (31/31), 5.21 KiB | 5.21 MiB/s, done.
Resolving deltas: 100% (8/8), done.
```

Cloning at the command line

- Use cd to move into your new repository and ls to look around
  - **`cd rrp-workshop-exercises`**
  - **ls -a** (include all of the hidden files)

```
❯ cd rrp-workshop-exercises
❯ ls -a
.   .git        LICENSE.md  analyses  rrp-workshop-exercises.Rproj
..  .gitignore  README.md   data      scripts
```

Shell scripting

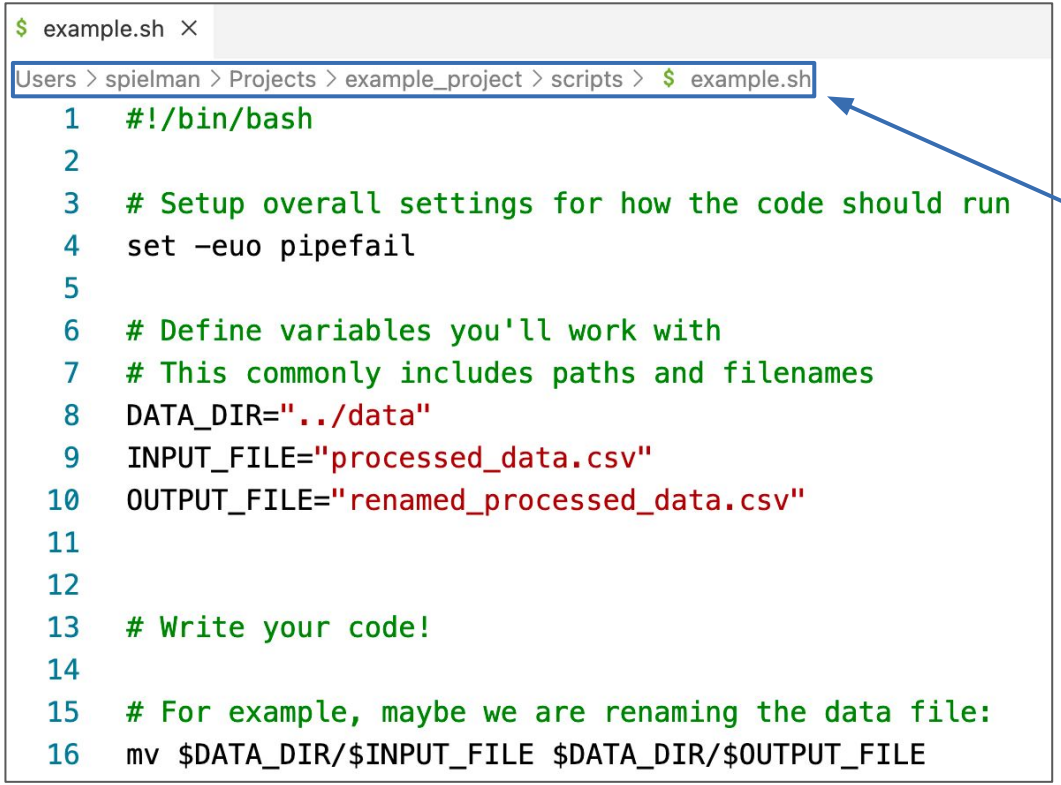
Childhood Cancer Data Lab

Shell scripts keep a *living* record of your UNIX-ing

- Using the command line interactively is great, but your commands aren't actually saved

- Shell scripts are files that run shell and UNIX commands
  - Both preserves the history of the commands you ran, *and runs them!*

The anatomy of a shell script



We generally like to run scripts from the directory where they are saved to avoid path confusion

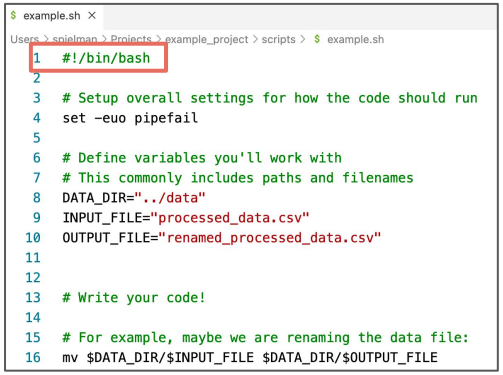
```
cd ~/Projects/example_project/scripts
```

Once in the right directory, we run this script from the terminal as:

```
sh example.sh
```

Best practices in scripting

```
1   #!/bin/bash
```



The very first line of shell scripts often contain a **shebang (#!)** indicating which shell interpreter be used when running this shell script.

This script will use the **BASH** shell

The **path to the given shell interpreter** immediately follows (no spaces!) the **#!**

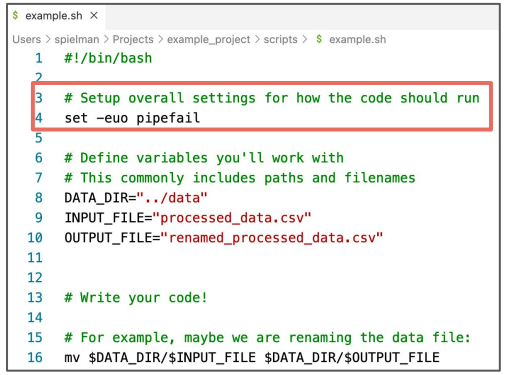
The shebang can be generally used to make any script *executable*, for interpretable languages.
It also cues you in to what language the code is in.

```
#!/usr/local/bin/python3
```

```
#!/usr/bin/env python
#!/usr/bin/env bash
```

Best practices in scripting

```
3   # Setup overall settings for how the code should run
4   set -euo pipefail
```



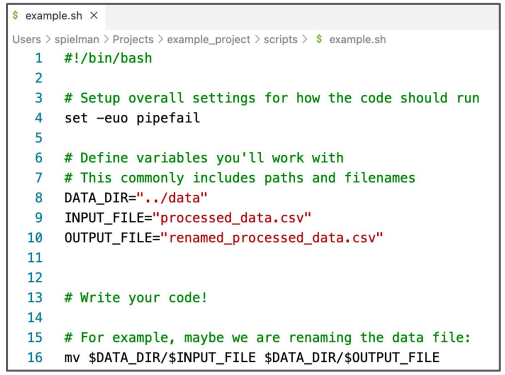
We like to use **`set`** to define preferences for how errors should be handled while running this script

- The **`-e`** flag causes the script to exit if any step has an error
- The **`-u`** flag causes the script to exit if a variable isn't defined
- The **`-o pipefail`** option causes the entire script to fail if any step in a pipeline fails

Note these can be used one-at-a-time, or pick only some options! All of these are legit (but **`o pipefail`** has to be kept together, and usually last):

```
set -e
set -u
set -o pipefail
set -eo pipefail
```

Defining variables




```
6   # Define variables you'll work with
7   # This commonly includes paths and filenames
8   DATA_DIR="../data"
9   INPUT_FILE="processed_data.csv"
10  OUTPUT_FILE="renamed_processed_data.csv"
```

Variables are *defined* as `VARIABLE_NAME="CONTENT"` ***without any spaces!!***

(Using quotes when defining variables is not strictly required, but it is best practice.)

Variables are *used* with **dollar signs:** `$CONTENT`, and sometimes braces: `${CONTENT}`

(Note there are more types of variables, like arrays, which are defined and used slightly differently! We'll just focus on single-value variables here).

Single/double quotes matter when using variables

When you want to refer to a variable in quotes, always use **double quotes**

```
MacBook-Pro :: ~ » DATA_DIR="data"
MacBook-Pro :: ~ » echo $DATA_DIR
data
MacBook-Pro :: ~ » echo "$DATA_DIR"
data
MacBook-Pro :: ~ » echo '$DATA_DIR'
$DATA_DIR
```

Variations on a variable

```
6   # Define variables you'll work with
7   # This commonly includes paths and filenames
8   DATA_DIR="../data"
9   INPUT_FILE="processed_data.csv"
10  OUTPUT_FILE="renamed_processed_data.csv"
```

You can combine variables with strings directly:

```
DATA_DIR="../data"
INPUT_FILE="processed_data.csv"
OUTPUT_FILE="renamed_$INPUT_FILE"
```

You can set up your variables to contain the path, if you want *(and it makes sense for the code!)*

```
DATA_DIR="../data"
INPUT_PATH="$DATA_DIR/processed_data.csv"
OUTPUT_PATH="$DATA_DIR/renamed_processed_data.csv"
```

Use curly braces { } to combine variables safely

```
DATA_DIR="../data"
INPUT_FILE="processed_data.csv"
PREFIX="renamed"
OUTPUT_FILE="${PREFIX}_${INPUT_FILE}"
```

Curly braces protect the variable

```
MacBook-Pro :: ~ » DATA_DIR="data"
MacBook-Pro :: ~ » INPUT_FILE="processed_data.csv"
MacBook-Pro :: ~ » PREFIX="renamed"
MacBook-Pro :: ~ » OUTPUT_FILE_BRACES="${PREFIX}_${INPUT_FILE}"
MacBook-Pro :: ~ » OUTPUT_FILE_NOBRACES="$PREFIX_$INPUT_FILE"
```

```
MacBook-Pro :: ~ » echo $OUTPUT_FILE_BRACES
renamed_processed_data.csv
MacBook-Pro :: ~ » echo $OUTPUT_FILE_NOBRACES
processed_data.csv
```

Putting it all together

$ example.sh ×

Users > spielman > Projects > example_project > scripts > $ example.sh

```
#!/bin/bash

# Setup overall settings for how the code should run
set -euo pipefail

# Define variables you'll work with
# This commonly includes paths and filenames
DATA_DIR="../data"
INPUT_FILE="processed_data.csv"
OUTPUT_FILE="renamed_processed_data.csv"


# Write your code!

# For example, maybe we are renaming the data file:
mv $DATA_DIR/$INPUT_FILE $DATA_DIR/$OUTPUT_FILE
```

Now let's write a script to…

- Download paired FASTQ reads (R1 and R2 files) programmatically
  - No "point-and-click" in browser!
  - https://trace.ncbi.nlm.nih.gov/Traces/sra/?study=SRP255885
  - https://sra-download.ncbi.nlm.nih.gov/traces/sra63/SRZ/011518/SRR11518889/NC16_S1_L004_R1_001.fastq.gz
  - https://sra-download.ncbi.nlm.nih.gov/traces/sra63/SRZ/011518/SRR11518889/NC16_S1_L004_R2_001.fastq.gz
- Save these files to the appropriate directory in your forked repository
- Ask how many lines are in each FASTQ file

The Git Stage/Commit/Push workflow

Childhood Cancer Data Lab

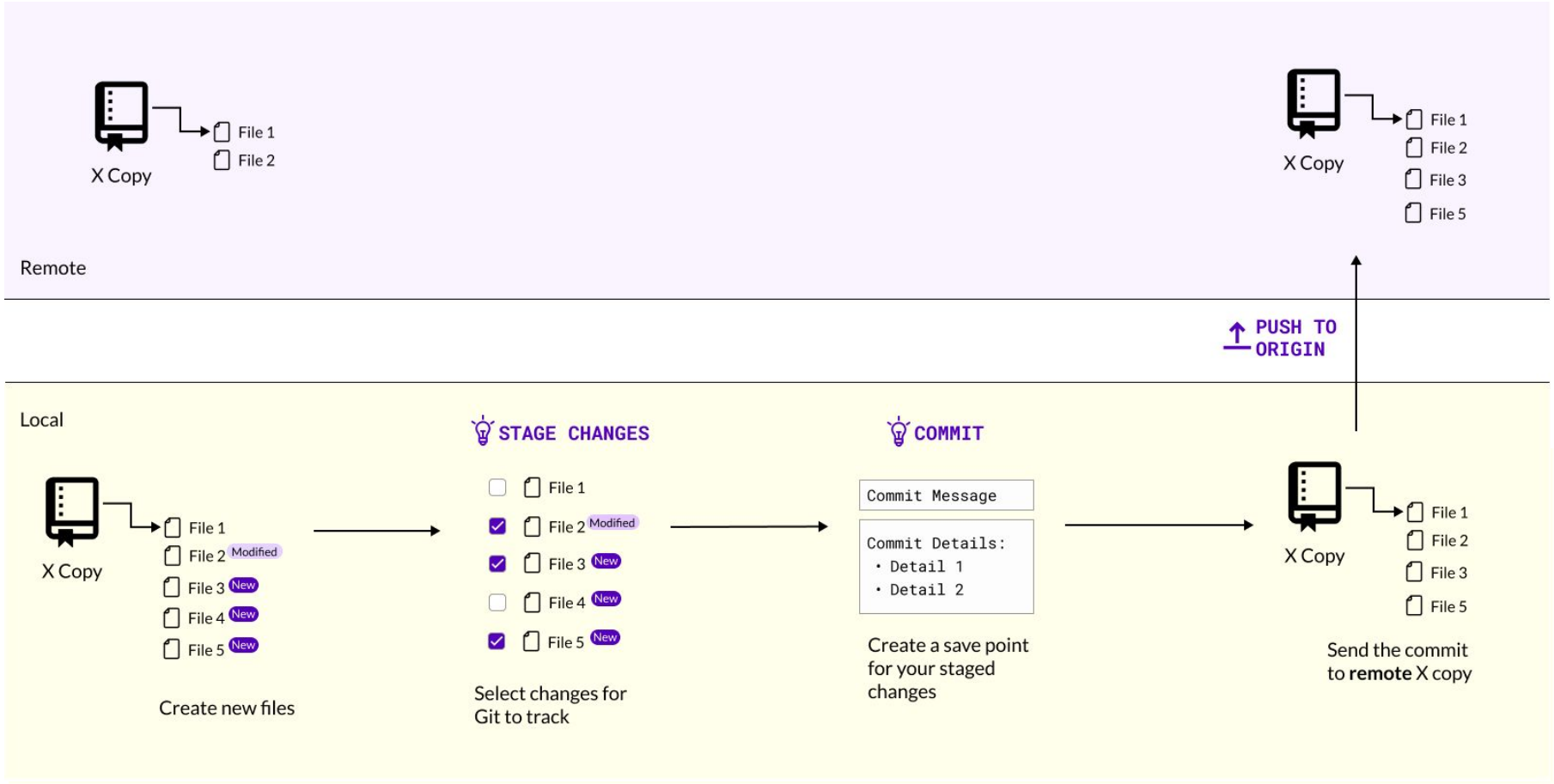
The "stage/commit/push" flow

1. Make code changes

2. **Stage** the changes you want to be part of the next commit
   a. Best practice to stage *one changed file at a time* to avoid problems
   b. Note: you may also hear this step referred to as "add"

3. **Commit** your code changes with an informative message

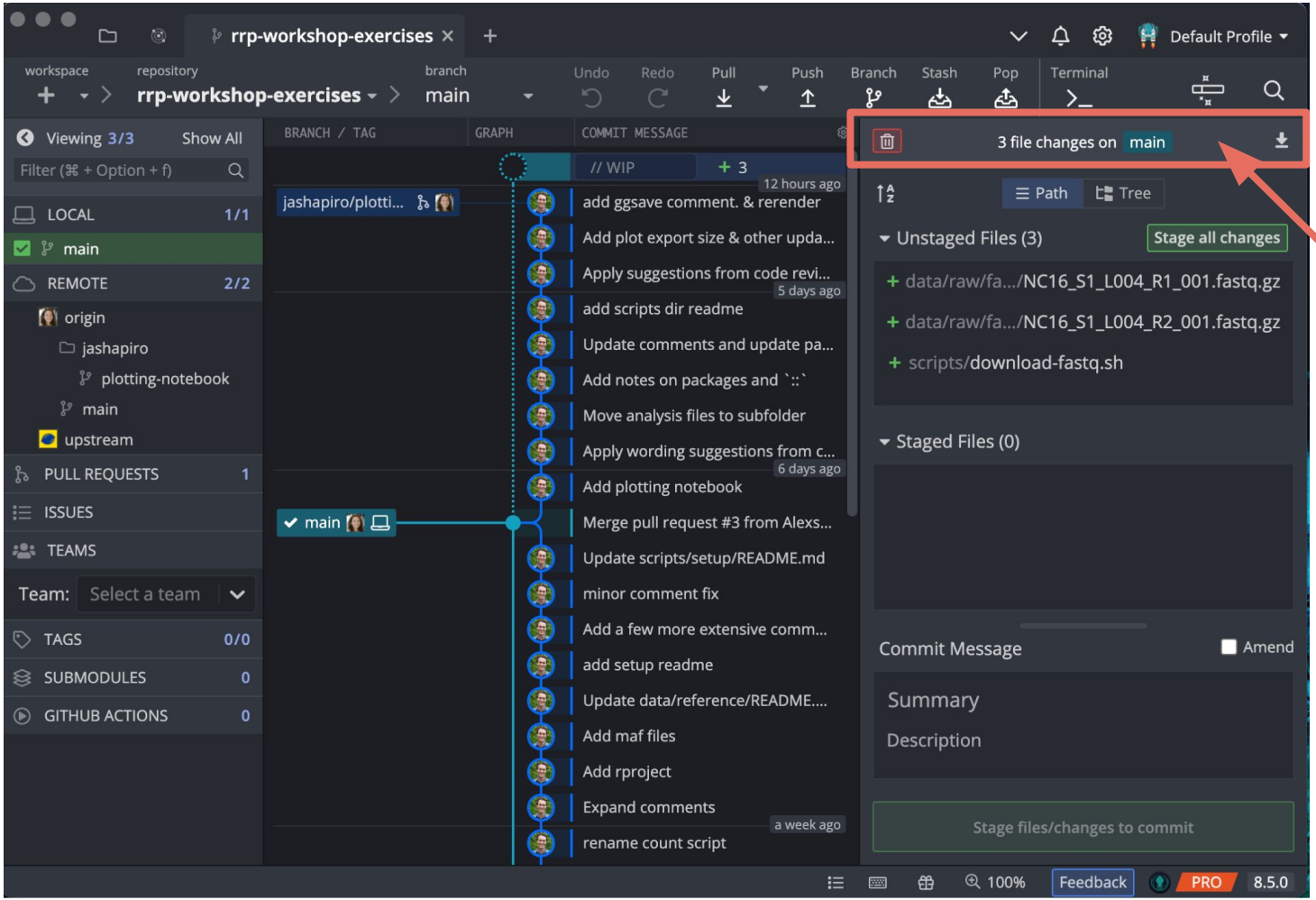
4. **Push** your local commit(s) to the **remote repository** you cloned on github.com

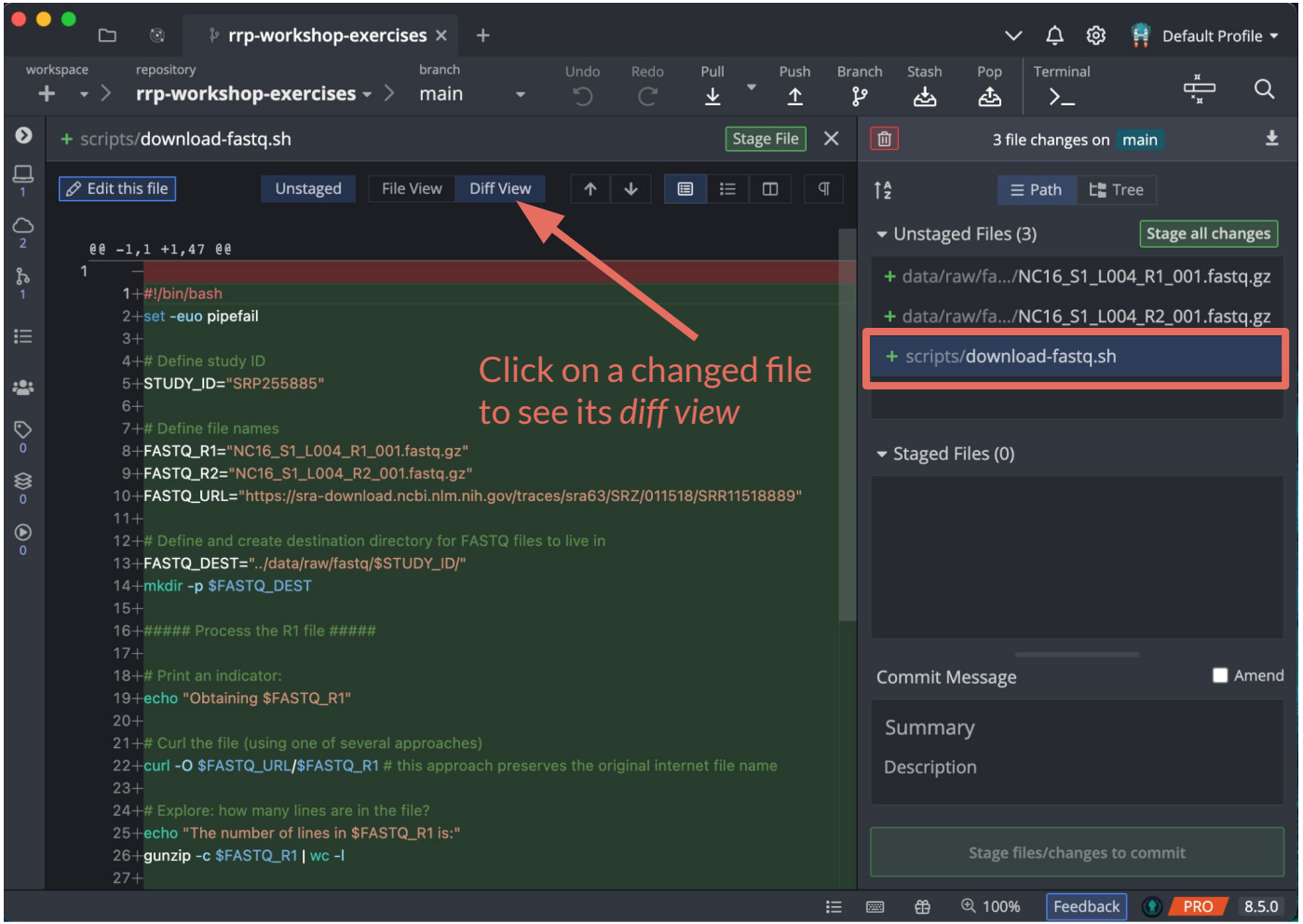

X Copy
File 1
File 2
X Copy
File 1
File 2
File 3
File 5
Remote
PUSH TO ORIGIN
Local
X Copy
File 1
File 2 Modified
File 3 New
File 4 New
File 5 New
Create new files
STAGE CHANGES
File 1
File 2 Modified
File 3 New
File 4 New
File 5 New
Select changes for Git to track
COMMIT
Commit Message
Commit Details:
• Detail 1
• Detail 2
Create a save point for your staged changes
X Copy
File 1
File 2
File 3
File 5
Send the commit to remote X copy

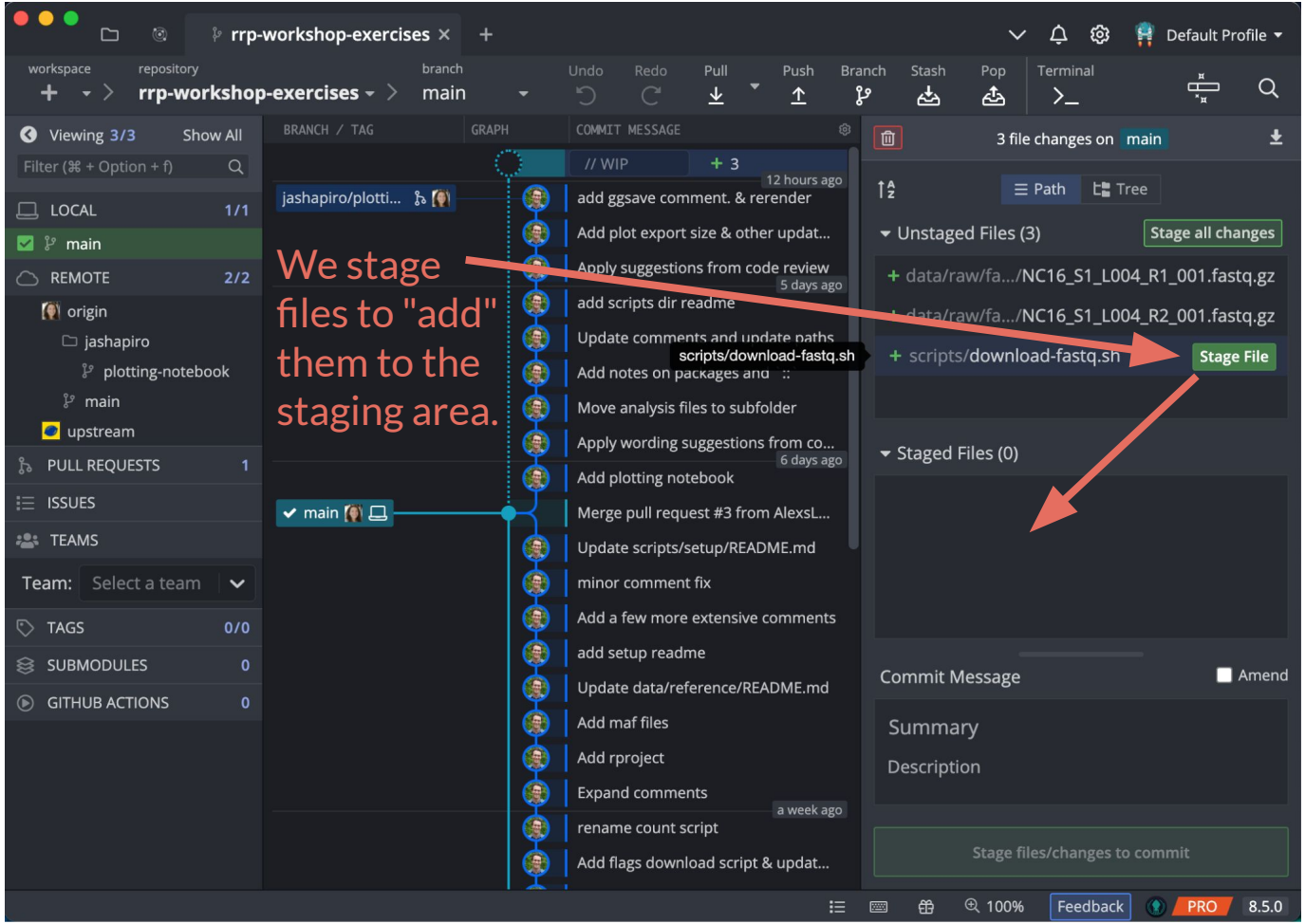
How much should you stage for a commit?

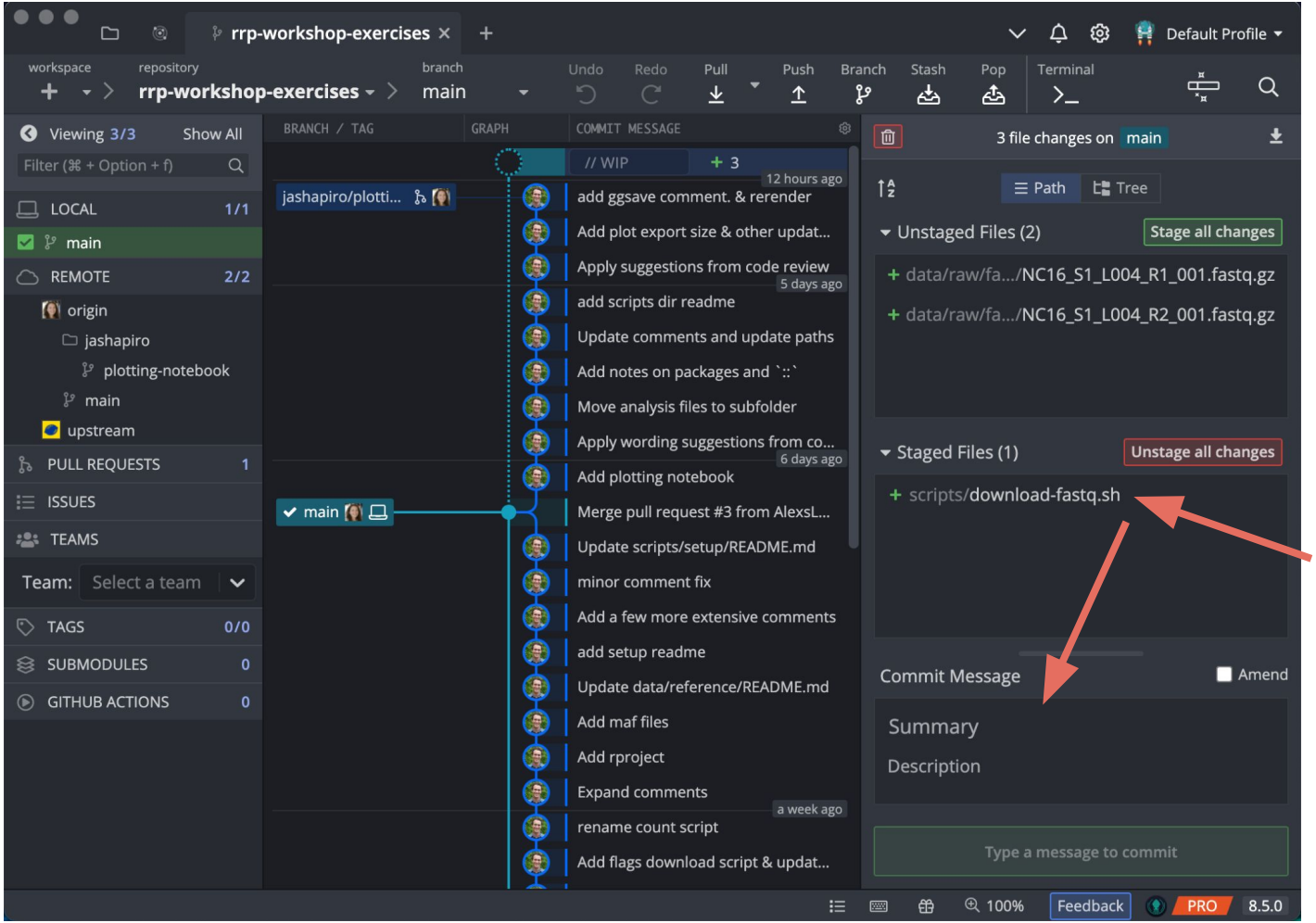
- Smaller units of work are generally better *for you!*
- Imagine you are looking in your code history. Which commit message is more helpful?
  - `All the work I did on Tuesday`
  - `Modified notebook to change plot size`
- Which scenario is easier to look through?
  - 1-3 changed files in a commit
  - 25 changed files in a commit

https://xkcd.com/1296/

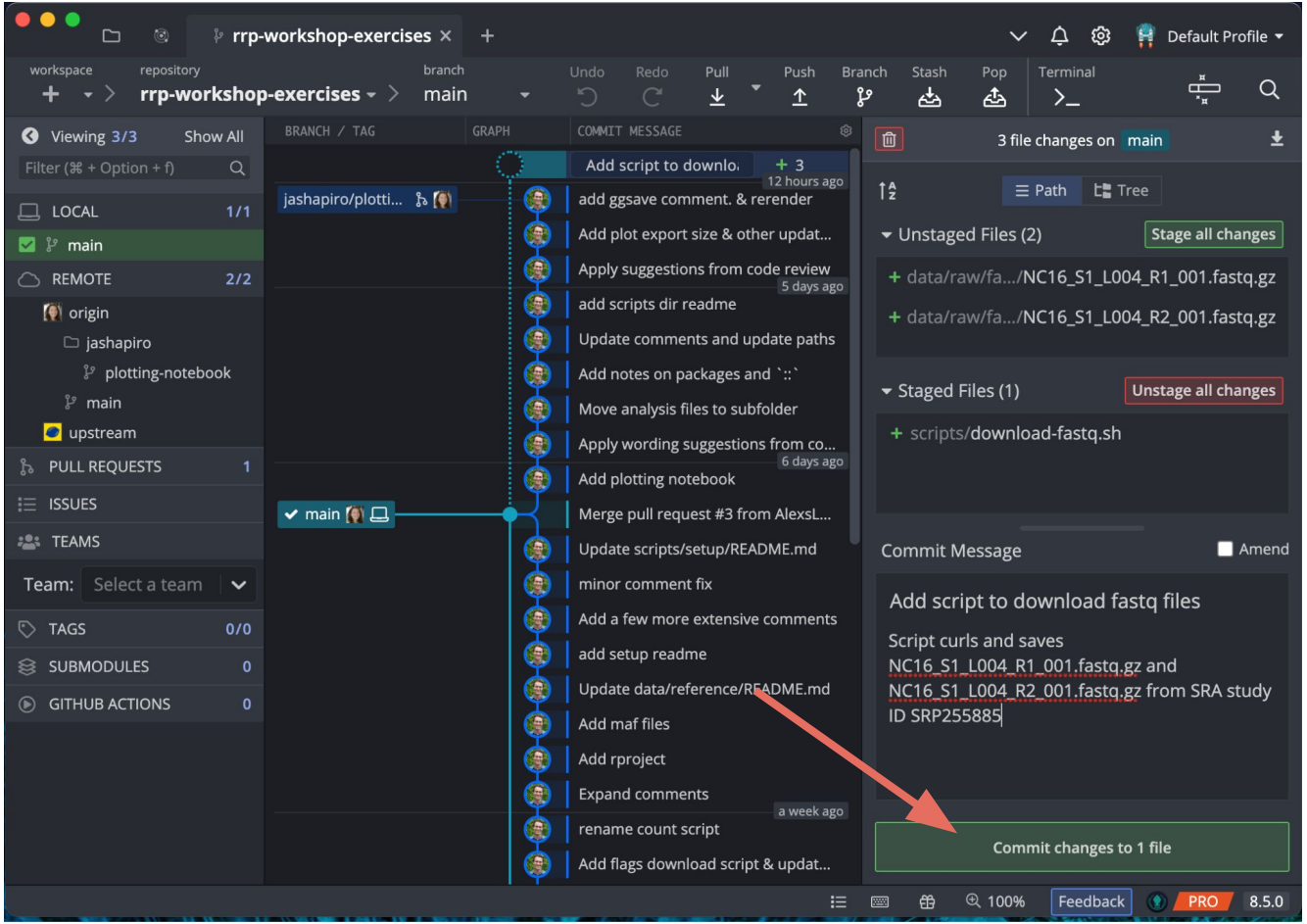

rrp-workshop-exercises
Default Profile
workspace
repository
rrp-workshop-exercises
branch
main
Undo
Redo
Pull
Push
Branch
Stash
Pop
Terminal
Viewing 3/3
Show All
Filter (⌘ + Option + f)
LOCAL 1/1
main
REMOTE 2/2
origin
jashapiro
plotting-notebook
main
upstream
PULL REQUESTS 1
ISSUES
TEAMS
Team: Select a team
TAGS 0/0
SUBMODULES 0
GITHUB ACTIONS 0
BRANCH / TAG
GRAPH
COMMIT MESSAGE
// WIP + 3
12 hours ago
jashapiro/plotti...
add ggsave comment. & rerender
Add plot export size & other upda...
Apply suggestions from code revi...
5 days ago
add scripts dir readme
Update comments and update pa...
Add notes on packages and `::`
Move analysis files to subfolder
Apply wording suggestions from c...
6 days ago
Add plotting notebook
main
Merge pull request #3 from Alexs...
Update scripts/setup/README.md
minor comment fix
Add a few more extensive comm...
add setup readme
Update data/reference/README....
Add maf files
Add rproject
Expand comments
a week ago
rename count script
3 file changes on main
Path
Tree
Unstaged Files (3)
Stage all changes
+ data/raw/fa.../NC16_S1_L004_R1_001.fastq.gz
+ data/raw/fa.../NC16_S1_L004_R2_001.fastq.gz
+ scripts/download-fastq.sh
Staged Files (0)
Commit Message
Amend
Summary
Description
Stage files/changes to commit
100%
Feedback
PRO
8.5.0

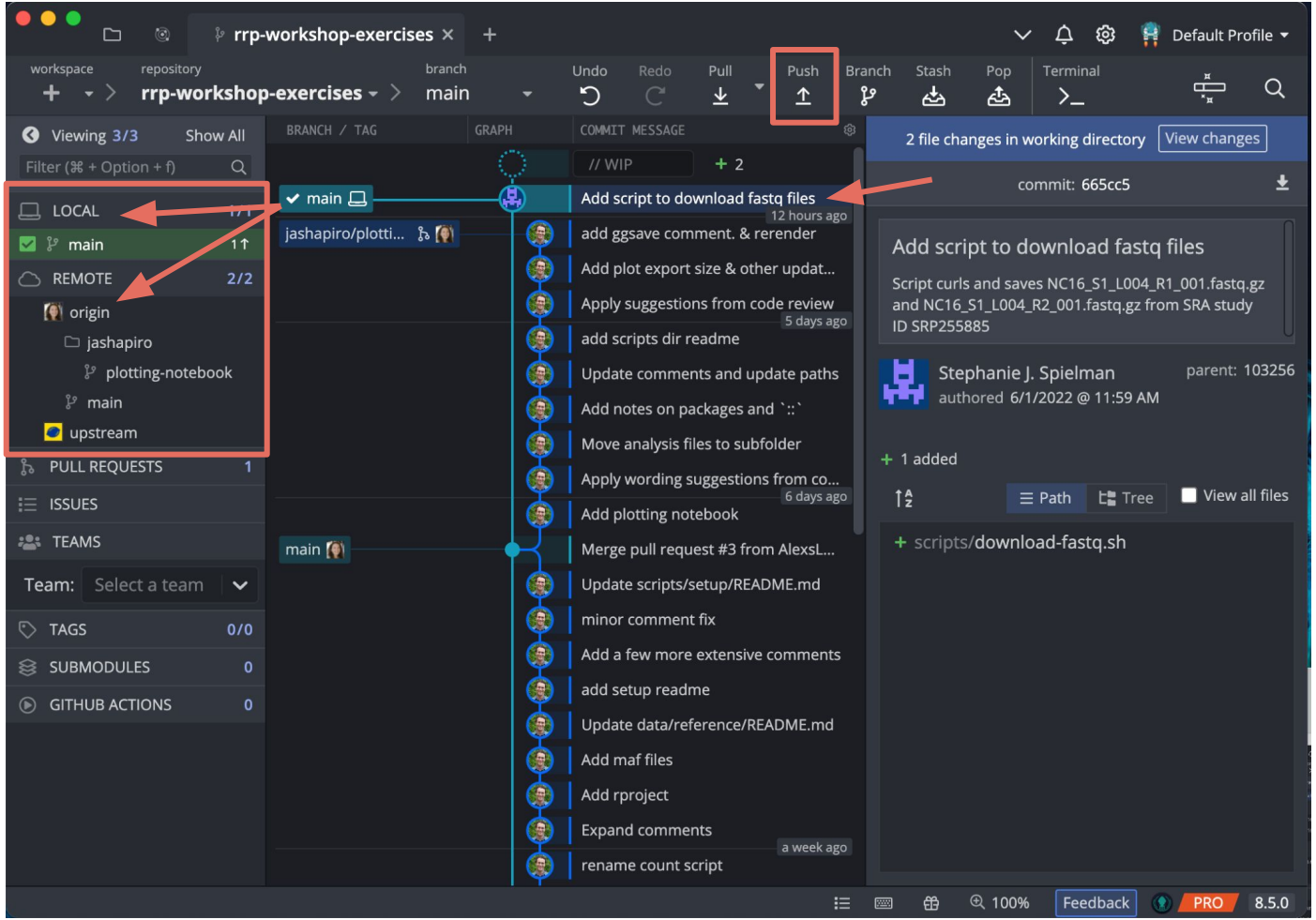

rrp-workshop-exercises
Default Profile
workspace
repository
rrp-workshop-exercises
branch
main
Undo
Redo
Pull
Push
Branch
Stash
Pop
Terminal
+ scripts/download-fastq.sh
Stage File
Edit this file
Unstaged
File View
Diff View
@@ -1,1 +1,47 @@
1 -
1+#!/bin/bash
2+set -euo pipefail
3+
4+# Define study ID
5+STUDY_ID="SRP255885"
6+
7+# Define file names
8+FASTQ_R1="NC16_S1_L004_R1_001.fastq.gz"
9+FASTQ_R2="NC16_S1_L004_R2_001.fastq.gz"
10+FASTQ_URL="https://sra-download.ncbi.nlm.nih.gov/traces/sra63/SRZ/011518/SRR11518889"
11+
12+# Define and create destination directory for FASTQ files to live in
13+FASTQ_DEST="../data/raw/fastq/$STUDY_ID/"
14+mkdir -p $FASTQ_DEST
15+
16+##### Process the R1 file #####
17+
18+# Print an indicator:
19+echo "Obtaining $FASTQ_R1"
20+
21+# Curl the file (using one of several approaches)
22+curl -O $FASTQ_URL/$FASTQ_R1 # this approach preserves the original internet file name
23+
24+# Explore: how many lines are in the file?
25+echo "The number of lines in $FASTQ_R1 is:"
26+gunzip -c $FASTQ_R1 | wc -l
27+
Click on a changed file to see its diff view
3 file changes on main
Path
Tree
Unstaged Files (3)
Stage all changes
+ data/raw/fa.../NC16_S1_L004_R1_001.fastq.gz
+ data/raw/fa.../NC16_S1_L004_R2_001.fastq.gz
+ scripts/download-fastq.sh
Staged Files (0)
Commit Message
Amend
Summary
Description
Stage files/changes to commit
100%
Feedback
PRO
8.5.0

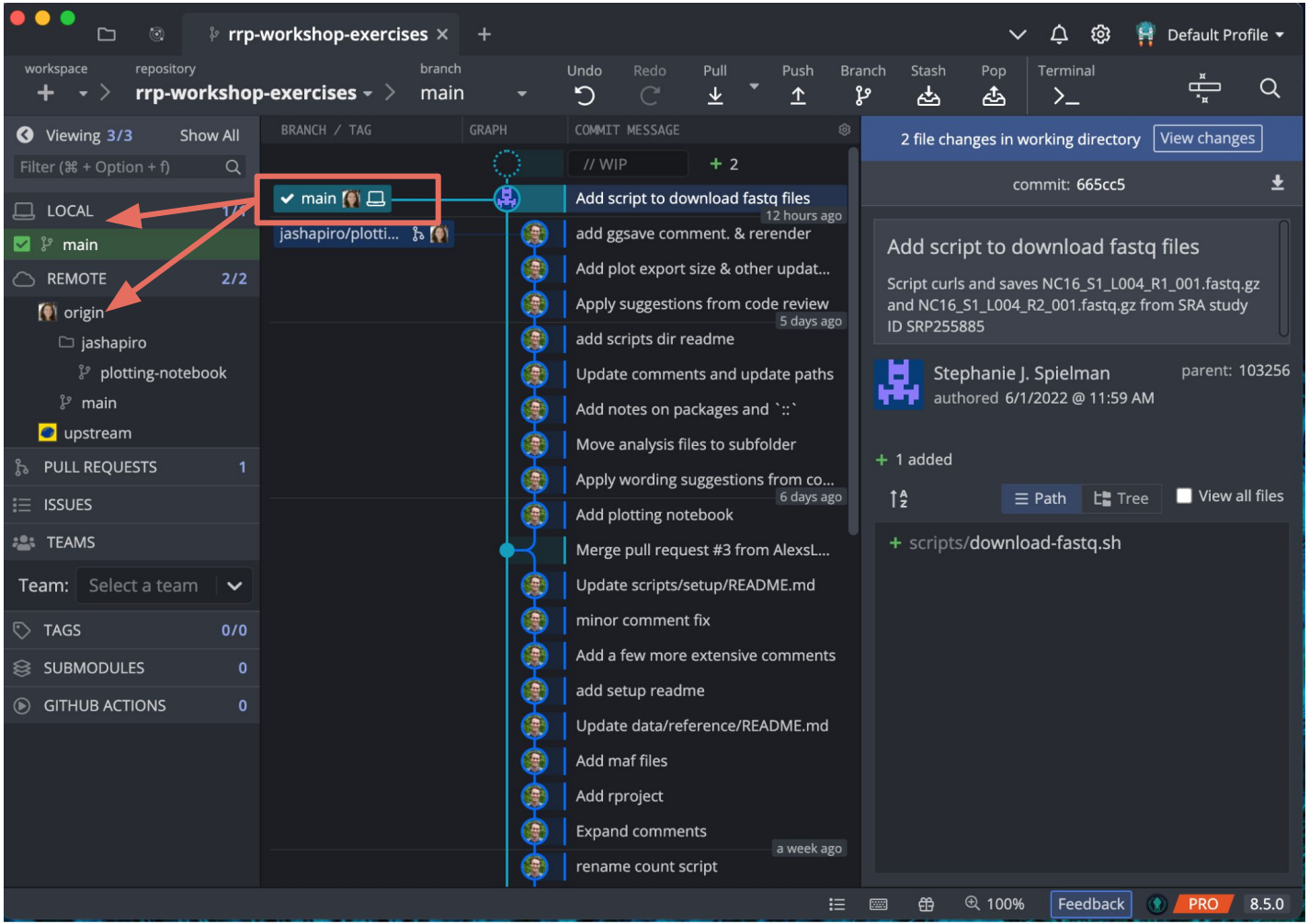

rrp-workshop-exercises
Default Profile
workspace
repository
rrp-workshop-exercises
branch
main
Undo
Redo
Pull
Push
Branch
Stash
Pop
Terminal
Viewing 3/3
Show All
Filter (⌘ + Option + f)
LOCAL 1/1
main
REMOTE 2/2
origin
jashapiro
plotting-notebook
main
upstream
PULL REQUESTS 1
ISSUES
TEAMS
Team: Select a team
TAGS 0/0
SUBMODULES 0
GITHUB ACTIONS 0
BRANCH / TAG
GRAPH
COMMIT MESSAGE
// WIP + 3
12 hours ago
jashapiro/plotti...
add ggsave comment. & rerender
Add plot export size & other updat...
Apply suggestions from code review
5 days ago
add scripts dir readme
Update comments and update paths
Add notes on packages and ::
Move analysis files to subfolder
Apply wording suggestions from co...
6 days ago
Add plotting notebook
main
Merge pull request #3 from AlexsL...
Update scripts/setup/README.md
minor comment fix
Add a few more extensive comments
add setup readme
Update data/reference/README.md
Add maf files
Add rproject
Expand comments
a week ago
rename count script
Add flags download script & updat...
We stage files to "add" them to the staging area.
3 file changes on main
Path
Tree
Unstaged Files (3)
Stage all changes
+ data/raw/fa.../NC16_S1_L004_R1_001.fastq.gz
+ data/raw/fa.../NC16_S1_L004_R2_001.fastq.gz
+ scripts/download-fastq.sh
scripts/download-fastq.sh
Stage File
Staged Files (0)
Commit Message
Amend
Summary
Description
Stage files/changes to commit
100%
Feedback
PRO
8.5.0



Now we are ready to *commit* these staged changes.


rrp-workshop-exercises
Default Profile
workspace
repository
rrp-workshop-exercises
branch
main
Undo
Redo
Pull
Push
Branch
Stash
Pop
Terminal
Viewing 3/3
Show All
Filter (⌘ + Option + f)
LOCAL 1/1
main
REMOTE 2/2
origin
jashapiro
plotting-notebook
main
upstream
PULL REQUESTS 1
ISSUES
TEAMS
Team: Select a team
TAGS 0/0
SUBMODULES 0
GITHUB ACTIONS 0
BRANCH / TAG
GRAPH
COMMIT MESSAGE
Add script to downlo
+ 3
12 hours ago
jashapiro/plotti...
add ggsave comment. & rerender
Add plot export size & other updat...
Apply suggestions from code review
5 days ago
add scripts dir readme
Update comments and update paths
Add notes on packages and `::`
Move analysis files to subfolder
Apply wording suggestions from co...
6 days ago
Add plotting notebook
main
Merge pull request #3 from AlexsL...
Update scripts/setup/README.md
minor comment fix
Add a few more extensive comments
add setup readme
Update data/reference/README.md
Add maf files
Add rproject
Expand comments
a week ago
rename count script
Add flags download script & updat...
3 file changes on main
Path
Tree
Unstaged Files (2)
Stage all changes
+ data/raw/fa.../NC16_S1_L004_R1_001.fastq.gz
+ data/raw/fa.../NC16_S1_L004_R2_001.fastq.gz
Staged Files (1)
Unstage all changes
+ scripts/download-fastq.sh
Commit Message
Amend
Add script to download fastq files
Script curls and saves NC16_S1_L004_R1_001.fastq.gz and NC16_S1_L004_R2_001.fastq.gz from SRA study ID SRP255885
Commit changes to 1 file
100%
Feedback
PRO
8.5.0


rrp-workshop-exercises
Default Profile
workspace
repository
rrp-workshop-exercises
branch
main
Undo
Redo
Pull
Push
Branch
Stash
Pop
Terminal
Viewing 3/3
Show All
Filter (⌘ + Option + f)
LOCAL
main 1↑
REMOTE 2/2
origin
jashapiro
plotting-notebook
main
upstream
PULL REQUESTS 1
ISSUES
TEAMS
Team: Select a team
TAGS 0/0
SUBMODULES 0
GITHUB ACTIONS 0
BRANCH / TAG
GRAPH
COMMIT MESSAGE
// WIP + 2
main
jashapiro/plotti...
Add script to download fastq files
12 hours ago
add ggsave comment. & rerender
Add plot export size & other updat...
Apply suggestions from code review
5 days ago
add scripts dir readme
Update comments and update paths
Add notes on packages and `::`
Move analysis files to subfolder
Apply wording suggestions from co...
6 days ago
Add plotting notebook
main
Merge pull request #3 from AlexsL...
Update scripts/setup/README.md
minor comment fix
Add a few more extensive comments
add setup readme
Update data/reference/README.md
Add maf files
Add rproject
Expand comments
a week ago
rename count script
2 file changes in working directory
View changes
commit: 665cc5
Add script to download fastq files
Script curls and saves NC16_S1_L004_R1_001.fastq.gz and NC16_S1_L004_R2_001.fastq.gz from SRA study ID SRP255885
Stephanie J. Spielman
parent: 103256
authored 6/1/2022 @ 11:59 AM
+ 1 added
Path
Tree
View all files
+ scripts/download-fastq.sh
100%
Feedback
PRO
8.5.0


rrp-workshop-exercises
Default Profile
workspace
repository
rrp-workshop-exercises
branch
main
Undo
Redo
Pull
Push
Branch
Stash
Pop
Terminal
Viewing 3/3
Show All
Filter (⌘ + Option + f)
LOCAL
main
REMOTE
2/2
origin
jashapiro
plotting-notebook
main
upstream
PULL REQUESTS
1
ISSUES
TEAMS
Team:
Select a team
TAGS
0/0
SUBMODULES
0
GITHUB ACTIONS
0
BRANCH / TAG
GRAPH
COMMIT MESSAGE
// WIP
+ 2
main
jashapiro/plotti...
Add script to download fastq files
12 hours ago
add ggsave comment. & rerender
Add plot export size & other updat...
Apply suggestions from code review
5 days ago
add scripts dir readme
Update comments and update paths
Add notes on packages and `::`
Move analysis files to subfolder
Apply wording suggestions from co...
6 days ago
Add plotting notebook
Merge pull request #3 from AlexsL...
Update scripts/setup/README.md
minor comment fix
Add a few more extensive comments
add setup readme
Update data/reference/README.md
Add maf files
Add rproject
Expand comments
a week ago
rename count script
2 file changes in working directory
View changes
commit: 665cc5
Add script to download fastq files
Script curls and saves NC16_S1_L004_R1_001.fastq.gz and NC16_S1_L004_R2_001.fastq.gz from SRA study ID SRP255885
Stephanie J. Spielman
authored 6/1/2022 @ 11:59 AM
parent: 103256
+ 1 added
Path
Tree
View all files
+ scripts/download-fastq.sh
100%
Feedback
PRO
8.5.0

Stage/commit/push from the command line

Before staging:

```
New-MacBook-Pro :: Projects/rrp-workshop-exercises/scripts <main> » git status
On branch main
Your branch is up to date with 'origin/main'.

Untracked files:
  (use "git add <file>..." to include in what will be committed)
        ../data/raw/fastq/
        download-fastq.sh

nothing added to commit but untracked files present (use "git add" to track)
```

Staging the `download_fastq.sh` script

We stage files with **`git add <filename>`**

```
MacBook-Pro :: Projects/rrp-workshop-exercises/scripts <main> » git add download_fastq.sh
MacBook-Pro :: Projects/rrp-workshop-exercises/scripts <main*> » git status
On branch main
Your branch is up to date with 'origin/main'.

Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        new file:   download_fastq.sh
```

Committing the staged files

We commit staged files with **`git commit`**

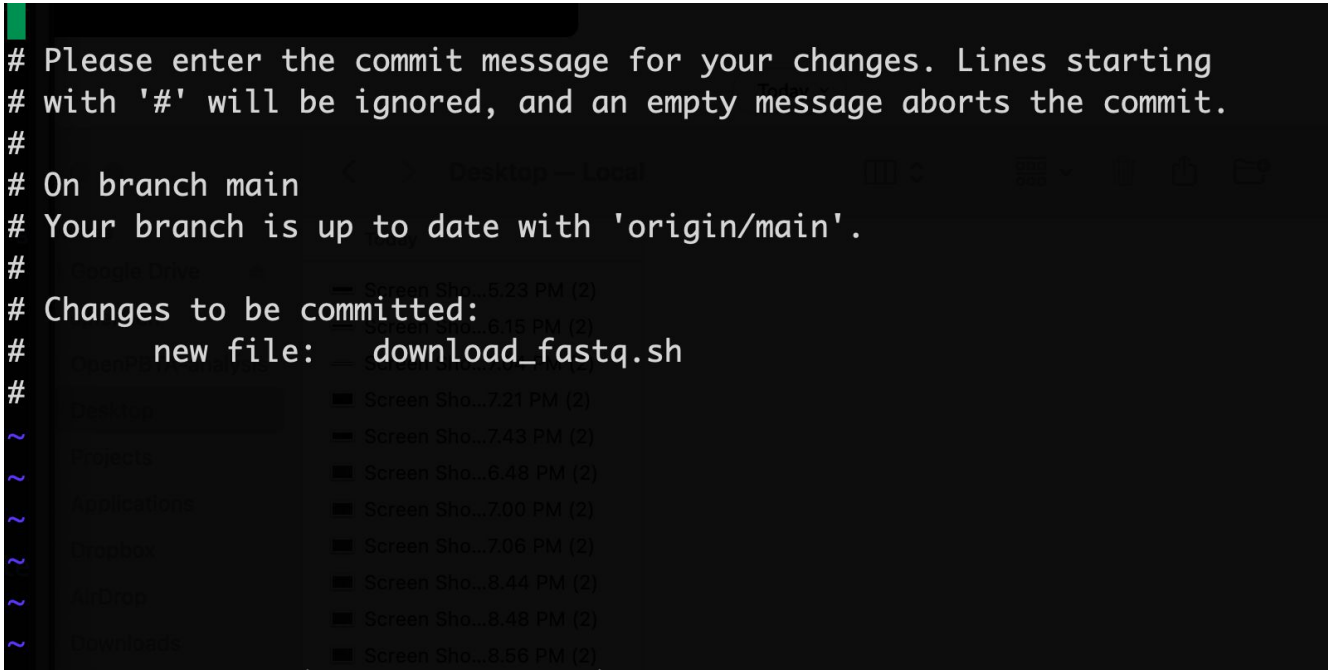
There are a few common approaches here…

Commit with `git commit`

- Pros:
  - Provide full commit information
  - See a clear record of what you are committing before you go through with it
- Cons:
  - You need to know how to use `vi`
  - Even if you know how to use `vi`, you might get stuck in `vi`

```
MacBook-Pro :: Projects/rrp-workshop-exercises/scripts <main*> » git commit
```

Commit with `git commit`



Did you end up here by accident and now you're stuck????

Type the following:

`:wq`

Commit with `git commit`

```
# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
#
# On branch main
# Your branch is up to date with 'origin/main'.
#
# Changes to be committed:
#       new file:   download_fastq.sh
#
~
~
~
~
~
~
~
:wq
```

Did you end up here by accident and now you're stuck????

Type the following:

`:wq`

And hit enter!

→ *Aborting commit due to empty commit message.*

Then, you'll have to redo the commit.

Commit with `git commit -m`

- Pros:
  - Provide your git commit message as a convenient argument!
  - Entirely bypass `vi` and avoid getting stuck in it
- Cons:
  - You don't clearly see what files you're actually committing (unless you ran `git status`)
  - You don't get to provide a more detailed description, which is sometimes useful for heavily involved code

```
MacBook-Pro :: Projects/rrp-workshop-exercises/scripts <main*> » git commit -m "Wrote script to
 download and save FASTQ files and count their lines"
[main 69e40b1] Wrote script to download and save FASTQ files and count their lines
 1 file changed, 50 insertions(+)
 create mode 100644 scripts/download_fastq.sh
```

Commit with `git commit -a`

- Pros:
  - Automatically stages modified/deleted files (but will not *add brand new* files)
- Cons:
  - Automatically stages modified/deleted files (but will not *add brand new* files)
  - You need to know how to use `vi`
  - Even if you know how to use `vi`, you might get stuck in `vi`

```
MacBook-Pro :: Projects/rrp-workshop-exercises/scripts <main*> » git commit -a
```

- *Hint: use **`git commit -a -m`** to avoid the vi trap and provide your message on the command line!!*

Push with git push

```
MacBook-Pro :: Projects/rrp-workshop-exercises/scripts <main> » git push
Enumerating objects: 6, done.
Counting objects: 100% (6/6), done.
Delta compression using up to 8 threads
Compressing objects: 100% (4/4), done.
Writing objects: 100% (4/4), 1.19 KiB | 1.19 MiB/s, done.
Total 4 (delta 1), reused 0 (delta 0)
remote: Resolving deltas: 100% (1/1), completed with 1 local object.
To github.com:sjspielman/rrp-workshop-exercises
   660b5e6..49c529e  main -> main
```

Not all files are meant to live under version control

- Sometimes we have files inside the repository directory that we very much *do not want* to be part of version control
  - Large files; github.com cannot generally accomodate files >= 100 MB
  - Unreleased/private data files
  - Other sneaky culprits like .DS_Store files on MacOS

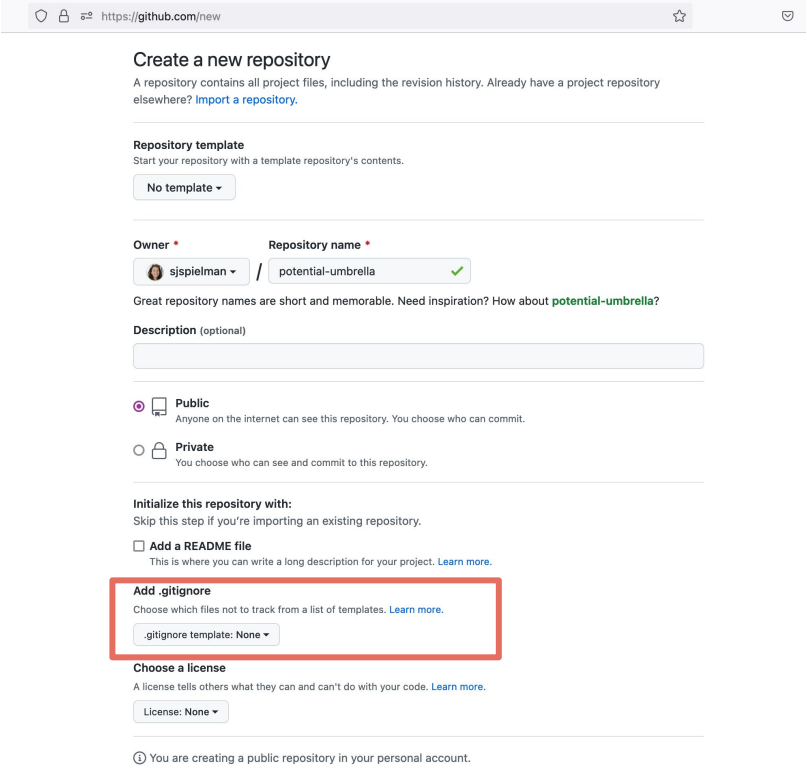
- It's pretty easy to mess up and accidentally stage/commit these files, and git loves to complain about unstaged files

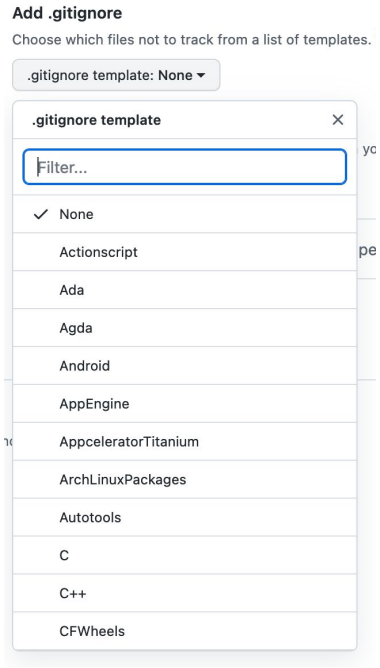
Using `.gitignore` files

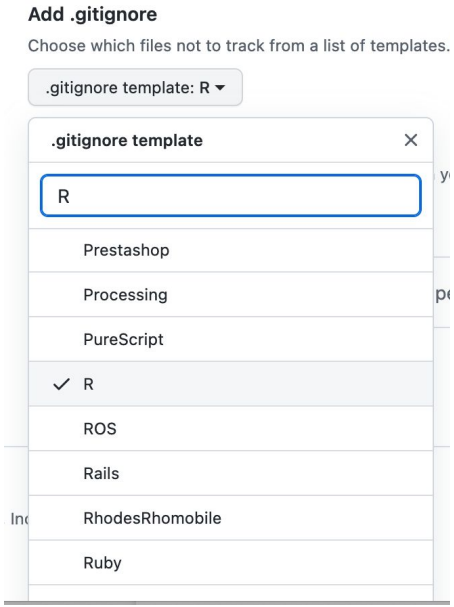
- `.gitignore` files are *hidden files* that tell `git` to ignore certain files
  - `.gitignore` will make `git` stop telling you about untracked changes
  - `.gitignore` will prevent you from staging/committing these files by accident
    - 🚨Word to the wise:🚨 Include files ASAP in your .gitignore before you accidentally commit them! (Trust us, we speak from experience….)

- 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.

github.com has some useful templates







Let's make a `.gitignore` file to also ignore...

```
data/raw/fastq/SRP255885/
```

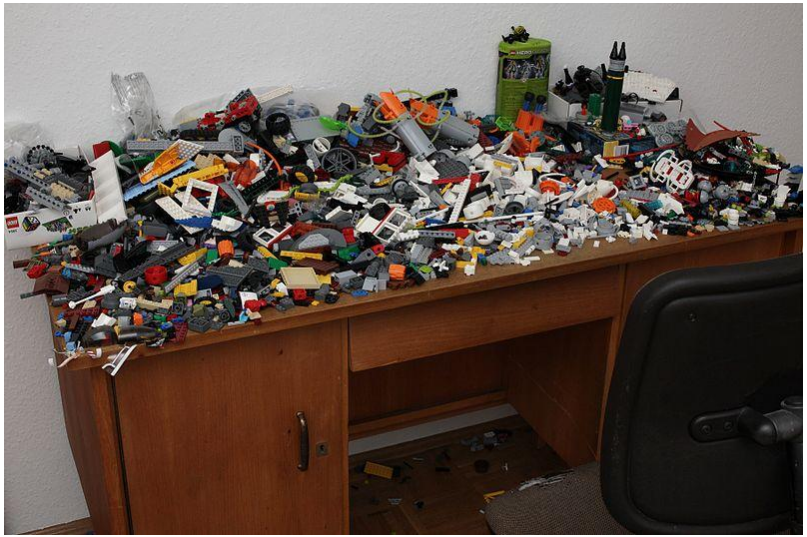
Organizing code in scripts and notebooks

Childhood Cancer Data Lab

Scripts and code get messy

- You don’t want to see my desk
  - Luckily, only I have to use it
  - But also, I waste a lot of time looking for things on my desk

- I try to keep my code more organized
  - But sometimes it starts off looking like my desk
  - Patterns and rules help!



Read style guides and use them (with judgement)

- Which style doesn't really matter, but find some agreement with collaborators
  - R tidyverse: https://style.tidyverse.org/
  - Google style guides (R, Python, others): https://google.github.io/styleguide/

- Don't get too hung up
  - Some style guides are really pedantic. Find your level of comfort
  - Packages are a mix of styles… use what fits

Style points to think about

- Variable and function name styles
  - `my_variable` vs. `myVariable`
  - `do_task()` vs. `task_doer()`
- Indentation and spacing
  - `counter=1` vs. `counter = 1`
  - tabs vs. spaces for indentation (very controversial!)
  - line length (try not to let lines get too long)
- Commenting style and format

Have we mentioned comments?

- Comments are for collaborators, and *Future You*
  - You *will* forget what you were thinking when you wrote your code
  - Your collaborators never knew what you were thinking at all
- Use comments for organization and documentation
  - Try to explain ***why*** you are doing what you are doing
  - Some “what” is okay, but remember that you can usually look up the functions you used
  - On occasion, people will tell you that code is or should be self-documenting
    - They are wrong
- ⚠️ Be aware of comments when updating code. You may need to update the comments too!⚠️

Let’s look at some examples

These examples are in R, but the concepts we talk about will be common across languages

In your `rrp-workshop-examples` repository, open:

`analyses/mutation_counts/01_count-gene-mutations.R`

Header comments

- Start your script or notebook with a description of its purpose

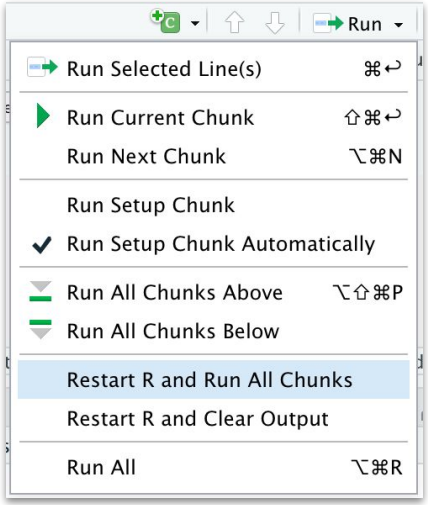
- What kinds of data does it expect?
- What output does it produce?
- What options are available?

- Example(s) of how to run the script

```
#!/usr/bin/env Rscript

# Count the number of samples mutated for each gene in a MAF file.

# This script reads in a MAF file and writes out a table of the genes with
# mutations. The table includes the number of samples that have at least one
# mutation in the gene, as well as the total number of mutations detected across
# all samples.

# Option descriptions:
#
# --maf :  File path to MAF file to be analyzed. Can be .gz compressed.
# --outfile : The path of the output file to create
# --vaf: Minimum variant allele fraction of mutations to include.
# --min_depth: Minimum sequencing depth to call mutations.
# --include_syn: Flag to include synonymous mutations in counts.

# Example invocation:
#
# Rscript 01_count-gene-mutations.R \
#   --maf mutations.maf.tsv.gz \
#   --outfile gene_counts.tsv
```

Even more comments! Computational Notebooks

- Rmarkdown, Jupyter
- Mix code, text and results in a single document
  - Record your thinking with styled text (markdown), headers for different sections, etc.
  - Plots and results appear right next to the code used to generate them
  - Publish results as html, pdf, or word documents
- Beware dragons:
  - Running code out of order is possible
  - Lingering environment variables can affect results
- Best practice: run everything from top to bottom in a clean environment



An example notebook

In your `rrp-workshop-examples` repository, open:

`analyses/mutation_counts/02_mutation-count-plots.Rmd`

To see the rendered version of this notebook, open:

`analyses/mutation_counts/02_mutation-count-plots.nb.html`

(in your browser)

Computational notebooks vs. scripts

- Notebooks are great for interactive work and reports
- Scripts can be better for repeated tasks with variable inputs and outputs
  - Scripts are often easier to integrate into a workflow

- A common pattern:
  - Start with a notebook
  - As the analysis matures or becomes repetitive, pull out code to a script
  - Generalize the script as needed
  - Keep a notebook with an analysis summary

A general plan for script and notebook content

(not exhaustive, not compulsory, somewhat controversial)

- A descriptive header comment first
  - describe the purpose of the script or notebook
- Setup code next
  - Packages and imported code
  - Inputs, outputs, and other “constant” parameters
  - Option parsing (scripts)
  - Custom function definitions
- Finally: the body of the code and content
  - Break up sections with comments and headings as appropriate

Packages and parameters

- Putting all `library()` or `import` statements near the beginning of the code makes it easier to see what packages are being used
  - All later code has access to the same environment
- Similarly, defining inputs and outputs early makes it clear what files are required, and what will be produced
- Parameters that won't be changed during the script: cutoffs, string names, etc.
  - You may want to modify these and rerun, so put them all together
  - Implicit in this: use variables for parameters that might get modified later

Functions before code

- Functions must be defined before they can be used
  - Generally this means appearing earlier in the script or notebook*
- Defining functions in multiple places within a script can make them hard to track down
- Having all the functions grouped together makes it easier to split them out for reuse

*It is sometimes possible to get a bit tricky: define a function called `main()` first with the core logic, then call it at the end of the script. This is common in some programming languages, but it doesn't work in notebooks.

Command line options

- You've seen how many UNIX commands take options (flags) at the command line
  - Your scripts can do this too!
- Writing code to process options can be tedious, but there are packages to help!
  - R: `optparse`
  - Python: `argparse`, `optparse`, `click`, others…
  - Things you get "for free"
    - parsing of options set at the command line & storing them in variables
    - default values for variables when no option is specified
    - basic error checking (missing values, types, etc.)
    - help documentation with `my_script -h`

`optparse` setup

```
34  option_list <- list(
35    make_option(
36      opt_str = c("--maf", "-m"),
37      type = "character",
38      default = NA,
39      help = "File path of MAF file to be analyzed. Can be .gz compressed."
40    ),
41    make_option(
42      opt_str = c("--outfile", "-o"),
43      type = "character",
44      default = "gene_counts.tsv",
45      help = "File path where output table will be placed."
46    ),
```

the command line option flags, long and short (long version will be the variable name)

the required data type

default value

help text; explain what the option is for

Function comments

- Much the same as the main header comments
- What are the arguments for the function?
  - What types and formats are expected?
  - What are any default values?
  - Be kind to yourself and others: check the argument values early and print good error messages
- What does the function return?
- Use language conventions for documentation:
  - Python: "docstring": a triple quoted comment right after the function definition
  - R: comments before the function
    - optional: use `roxygen2` format
      (automates documentation for packages, but a nice standard format for other use)
- *Remember:* Even though Present You is writing this function, *Future You does not recall it at all.*

```r
#' Plot the number of  mutations for each gene from a data frame
#'
#' Filters to a cutoff and sorts genes from most to least mutated, then
#' creates a bar plot of the mutation counts, optionally highlighting
#' genes of interest.
#'
#' @param mutations_df A data frame with columns `Hugo_Symbol` and `mutated_samples`
#' @param min_mutated The minimum `mutated_samples` value to include in the plot
#'  (default: 3)
#' @param highlight_genes A vector of genes to highlight in the plot (optional)
#' @param highlight_title The title for the highlighted genes legend
#'  (default: "Gene of interest")
#'
#' @return A ggplot2 plot object
#'
plot_gene_mutations <- function(
  mutations_df,
  min_mutated = 3,
  highlight_genes = c(),
  highlight_title = "Gene of interest"
){
```

Explicit “namespaces” to avoid conflicts

- Sometimes multiple packages have functions with the same name
- R: Use `package::function()` syntax to avoid ambiguity
  - `dplyr::filter()` vs. `stats::filter()`
  - Also provides in-code documentation: *What package did this strange function come from?*
  - Bonus: you don’t need a `library()` statement
- Python: `package.function()` syntax is standard
  - avoid `from package import function`
  - use `import pandas as pd` and similar if there is a common standard

```
> library(dplyr)

Attaching package: ‘dplyr’

The following objects are masked from ‘package:stats’:

    filter, lag

The following objects are masked from ‘package:base’:

    intersect, setdiff, setequal, union
```

Random number seeds

- Some code makes use of random numbers
  - simulations
  - fitting statistical models/machine learning
  - PCA, UMAP, tSNE
- But sometimes we want perfect reproducibility! (debugging, testing)
  - luckily, computers don't use real random numbers
  - random number generators are functions
    - given the same starting point (seed), they will give the same results
- Start your code by setting the seed for replicability
  - R: `set.seed(2022)`
  - Python: `random.seed(2022)`
  - other packages and tools: look at the docs for the correct option!
    - some packages don't use the language defaults

Managing Packages and Environments

Childhood Cancer Data Lab

Software is called “soft” for a reason

- Software is always changing!
  - New versions can bring new features and fix bugs 🎉
  - But also remove features you relied on 😟
  - Or alter behavior in unexpected ways 😱

- Changes in one piece of software can break other software
  - Sometimes this is intentional! Operating systems are constantly updating to break hacking tools

Changes occur at every level of the computing “stack”

- Individual scripts/analyses
- Packages within R, Python, etc.
- Individual programs (Cell Ranger, Salmon, etc., but also R & Python)
- Operating system
- Hardware

We want to do our best to **track** and **document** versions of as many layers as possible.

Ideally, we would like to **freeze** versions, so we and anyone else can come back and know results will be the same!

The “analysis” layer

- We have already talked about tracking your changes with Git and GitHub
  - If you know which commit of your scripts you used to produce an analysis, you can point people right to that
  - “**tags**” and “**releases**” on GitHub can make this easier when you have a particular commit you want to share (but we won’t be covering that in this workshop)

The package layer

- Research software tools in bioinformatics (and beyond) are often published as packages for R or Python
  - Examples: Seurat, scanpy, tidyverse, pandas, Bioconductor packages

- This makes them generally easy to install and update as research progresses
  - BUT easy to update means things can change fast
  - New versions may change results, even for existing functions!
    - Newer isn't always better; sometime you want to stick with the old way
  - Dependencies on other packages may require specific versions of other packages

Documenting package versions in R

- `sessionInfo()` is your friend
  - or `sessioninfo::session_info()`

```
R version 4.1.2 (2021-11-01)
Platform: x86_64-apple-darwin17.0 (64-bit)
Running under: macOS Monterey 12.4

Matrix products: default
LAPACK: /Library/Frameworks/R.framework/Versions/4.1/Reso

locale:
[1] en_US.UTF-8/en_US.UTF-8/en_US.UTF-8/C/en_US.UTF-8/en_

attached base packages:
[1] stats     graphics  grDevices utils     datasets  met

other attached packages:
[1] magrittr_2.0.3 ggplot2_3.3.6  dplyr_1.0.9

loaded via a namespace (and not attached):
 [1] bslib_0.3.1        jquerylib_0.1.4    RColorBrewer_1
 [5] compiler_4.1.2     tools_4.1.2        digest_0.6.29
 [9] jsonlite_1.8.0     evaluate_0.15      lifecycle_1.0.
[13] gtable_0.3.0       pkgconfig_2.0.3    rlang_1.0.2
[17] rstudioapi_0.13    yaml_2.3.5         parallel_4.1.2
```

```
─ Session info ──────────────────────────────────────────────────────────────
 setting  value
 version  R version 4.1.2 (2021-11-01)
 os       macOS Monterey 12.4
 system   x86_64, darwin17.0
 ui       RStudio
 language (EN)
 collate  en_US.UTF-8
 ctype    en_US.UTF-8
 tz       America/New_York
 date     2022-05-31
 rstudio  2022.02.2+485 Prairie Trillium (desktop)
 pandoc   2.17.1.1 @ /Applications/RStudio.app/Contents/MacOS/quarto/bin/ (via rmarkdown)

─ Packages ──────────────────────────────────────────────────────────────────
 package      * version date (UTC) lib source
 bit            4.0.4   2020-08-04 [1] CRAN (R 4.1.0)
 bit64          4.0.5   2020-08-30 [1] CRAN (R 4.1.0)
 bslib          0.3.1   2021-10-06 [1] CRAN (R 4.1.0)
 cli            3.3.0   2022-04-25 [1] CRAN (R 4.1.2)
 colorspace     2.0-3   2022-02-21 [1] CRAN (R 4.1.2)
 crayon         1.5.1   2022-03-26 [1] CRAN (R 4.1.2)
 digest         0.6.29  2021-12-01 [1] CRAN (R 4.1.0)
 dplyr        * 1.0.9   2022-04-28 [1] CRAN (R 4.1.2)
 ellipsis       0.3.2   2021-04-29 [1] CRAN (R 4.1.0)
 evaluate       0.15    2022-02-18 [1] CRAN (R 4.1.2)
```

But how do you recreate the same set of packages?

Installing packages based on `sessionInfo()` output could be very tedious!

Enter **`renv`**!

- **`renv`** is an R package for *tracking*, *freezing*, and *sharing* R environments, including all of the package versions that were installed.
- Each project can have its own environment, with its own set of packages
  - Different projects may require different versions of packages
  - **`renv`** can help manage these different sets of package versions
- When sharing a project/analysis, using **`renv`** allows everyone stay in sync with same packages and versions

How does `renv` work?

Rather than using the system R package library, renv creates a library for each project that R will use when running code for the project ("Project Library")

This `renv`-created library could be large, so we can't reasonably share the whole thing.

Instead, we create a file (`renv.lock`) that describes the library.

`renv` uses this file to track all of the packages we are using, and recreate the library with those packages as needed.

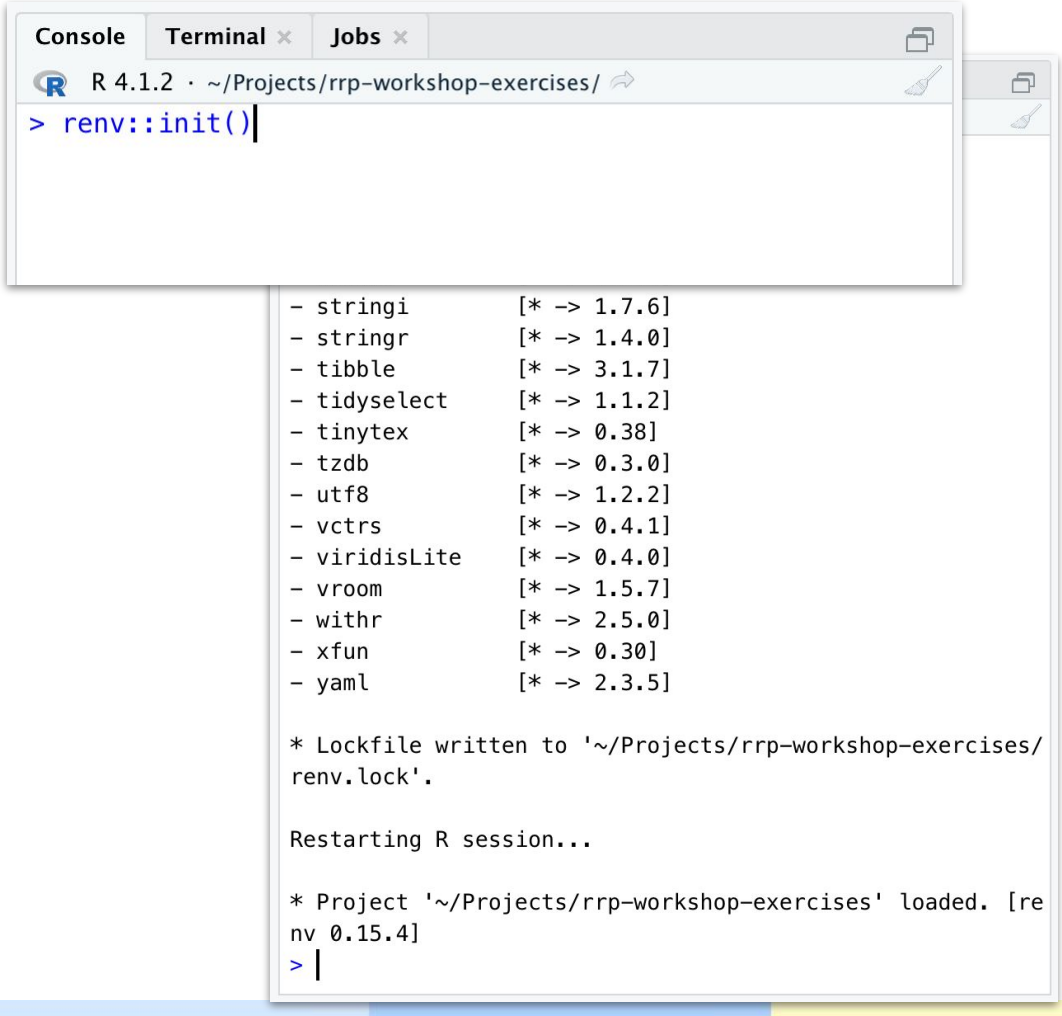
renv initialization

In the console, enter:

```
renv::init()
```

Lots of text will scroll by, and your R session will restart.

That's it! You are starting to track your R packages!



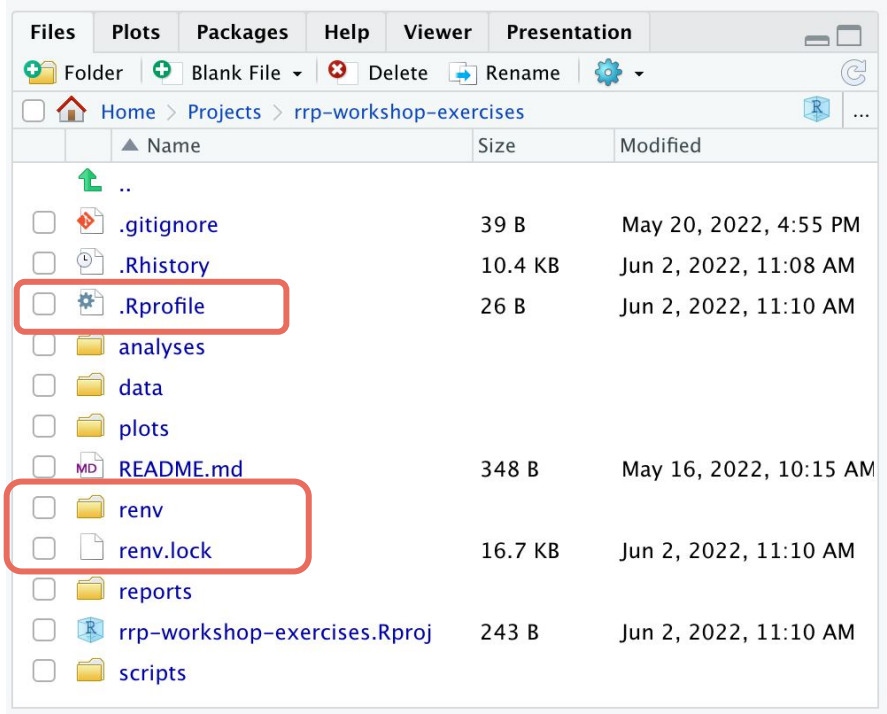
What did `renv::init()` do?

Added an **`renv.lock`** file, **`renv/`** folder and **`.Rprofile`** file
(or modified the one you had)

The **`.Rprofile`** file is run when R launches *for this project*, and it contains a command to configure `renv` on launch.

The **`renv/`** folder is where the Project Library and support files can be found

Newly installed packages for the project will be stored in this Project Library



The `renv.lock` file

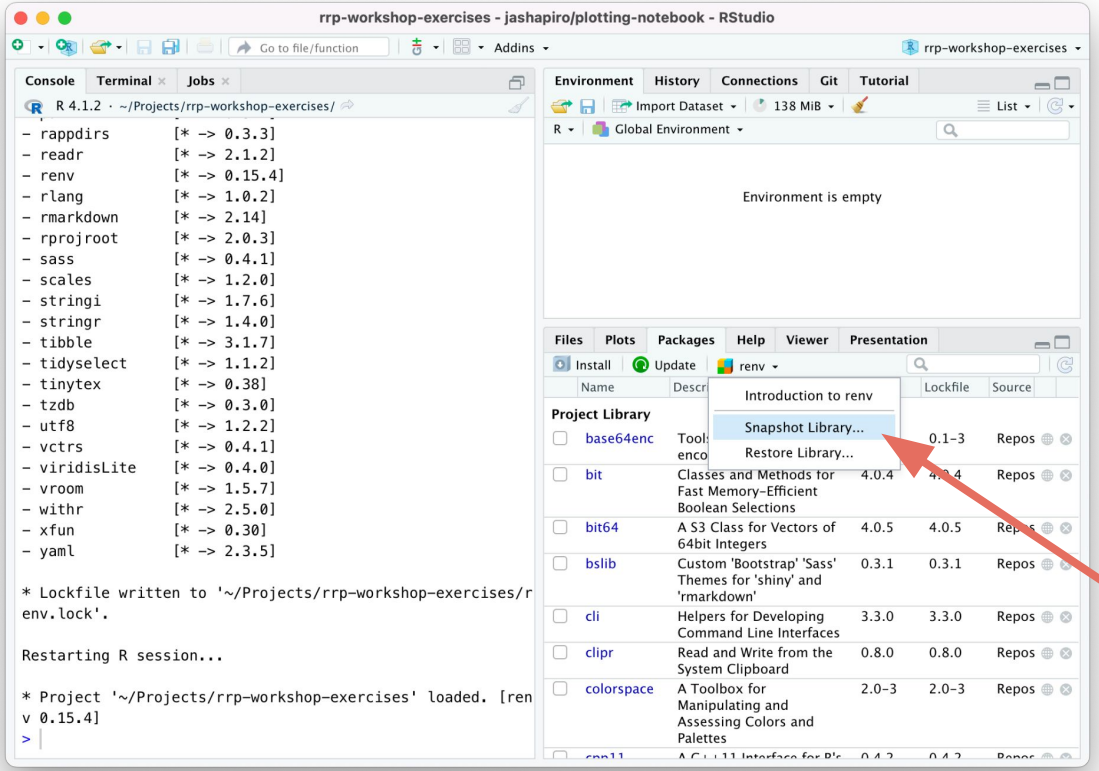
Taking a snapshot creates or updates the `renv.lock` file at the base of your project.

This file records…

- Which packages are installed
- The package versions
- Where the packages came from

Do not edit this file manually!

Updating the `renv.lock` file



An "renv" menu now appears in the Packages pane

Use "Snapshot Library..." to update the `renv.lock` file to the current setup, e.g. after you update or install new packages

Alternatively, in the console enter:

```
renv::snapshot()
```

Restoring a library from an `renv.lock` file

When working on a new machine, or if someone else updated the `renv.lock` file, you may need to update the Project Library

```
* Project '~/Projects/rrp-workshop-exercises' loaded. [renv 0.15.4]
* The project library is out of sync with the lockfile.
* Use `renv::restore()` to install packages recorded in the lockfile.
```

Follow the instructions! Enter **`renv::restore()`** in the console (or use the renv menu "Restore Library" option) to sync your Package Library with the recorded versions, installing any missing packages.

Which packages are included in `renv.lock`?

You might have many packages installed, but only use some in a given project.

`renv` tries to be smart about this, and only includes packages that it finds ***used*** within code in the project folder, or packages that are required by the packages that are used (so-called *dependencies*)

Sometimes renv misses a package (particularly for packages with optional dependencies), and you might need to create a file (we usually call ours `dependencies.R`) that only contains lines like:

```
library(missing_package)
```

which will force renv to include that package in the lockfile.

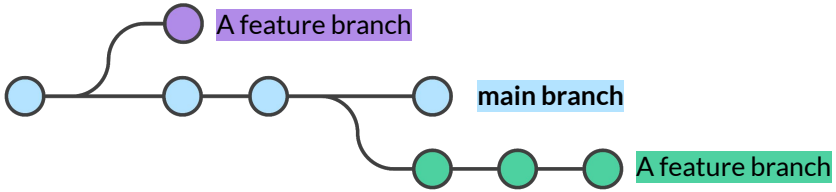
Other package management systems

- ***renv*** is pretty useful, but it only gets us so far… only R packages (and a bit of Python, in some situations)
  - For software outside R, other package management systems are required
- ***conda*** is one of the more popular and flexible package managers
  - Started as a Python package manager, but it can be used for any command line software
  - Like renv, you can create separate sets of software with different versions for different projects
  - LOTS of bioinformatics software available through **bioconda**
  - https://bioconda.github.io/user/install.html
- ***Docker*** and ***Singularity*** are another level up
  - “Containers” that include everything from the operating system up
  - Run one OS inside another, with *all the things* frozen to particular versions
  - Cloud platforms love containers…

Working with branches in Git

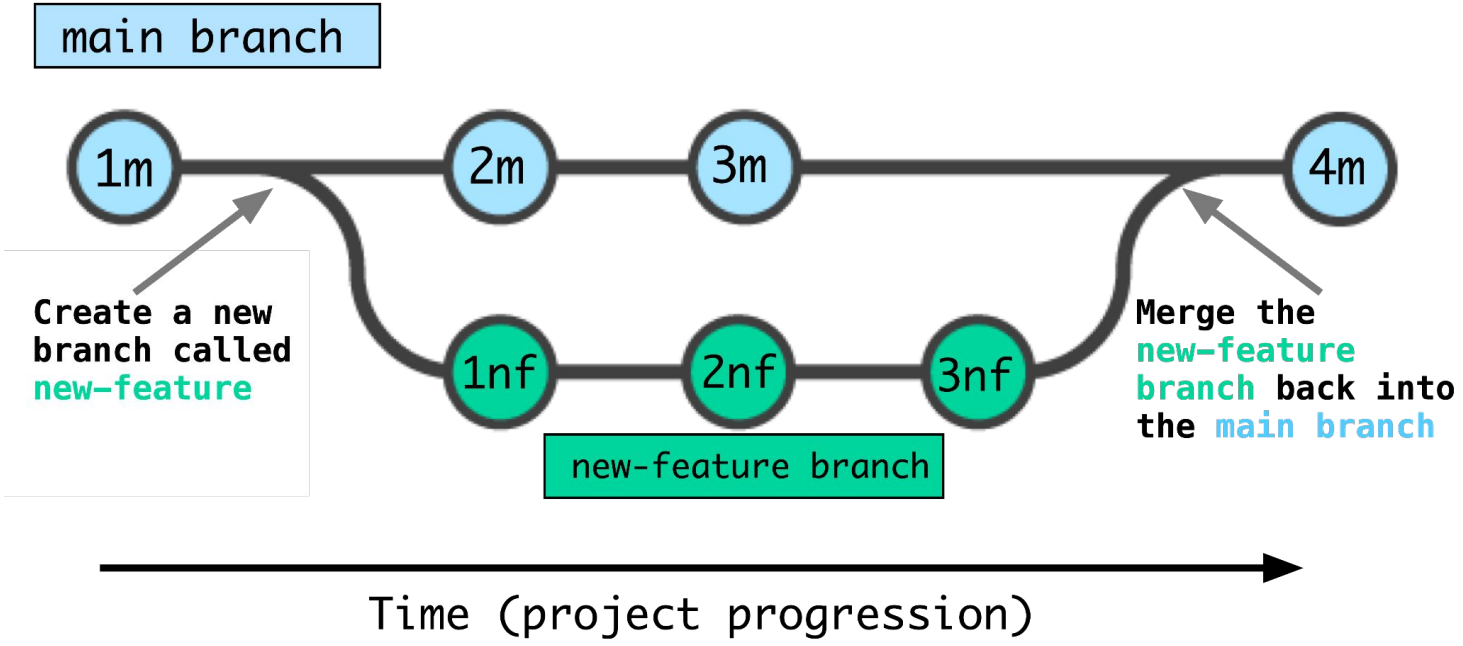
Childhood Cancer Data Lab

Branches in git



- Branches are like "repositories within repositories" 🤯
- Useful when you want to make changes (maybe experimental!) but you don't want to break the rest of your code
  - You can always switch back to a "clean" branch!
- Keep related changes together
  - All commits for a given new analysis or "feature" can be made within the same branch for easier tracking
  - Helps you to identify which commits are relevant to a given analysis
- If you wreck code in a branch, you've *only wrecked that branch!* Just delete it!
- Branches provide a great framework for collaboration and team science

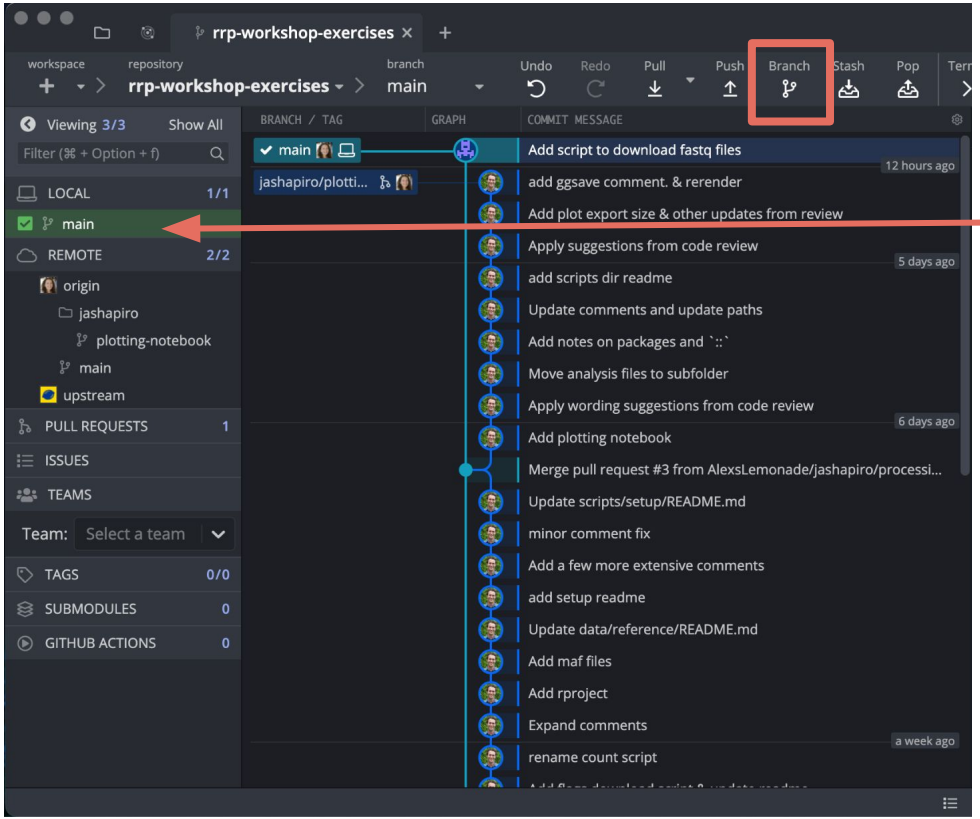
Source: https://www.atlassian.com/git/tutorials/using-branches



main branch history after merge

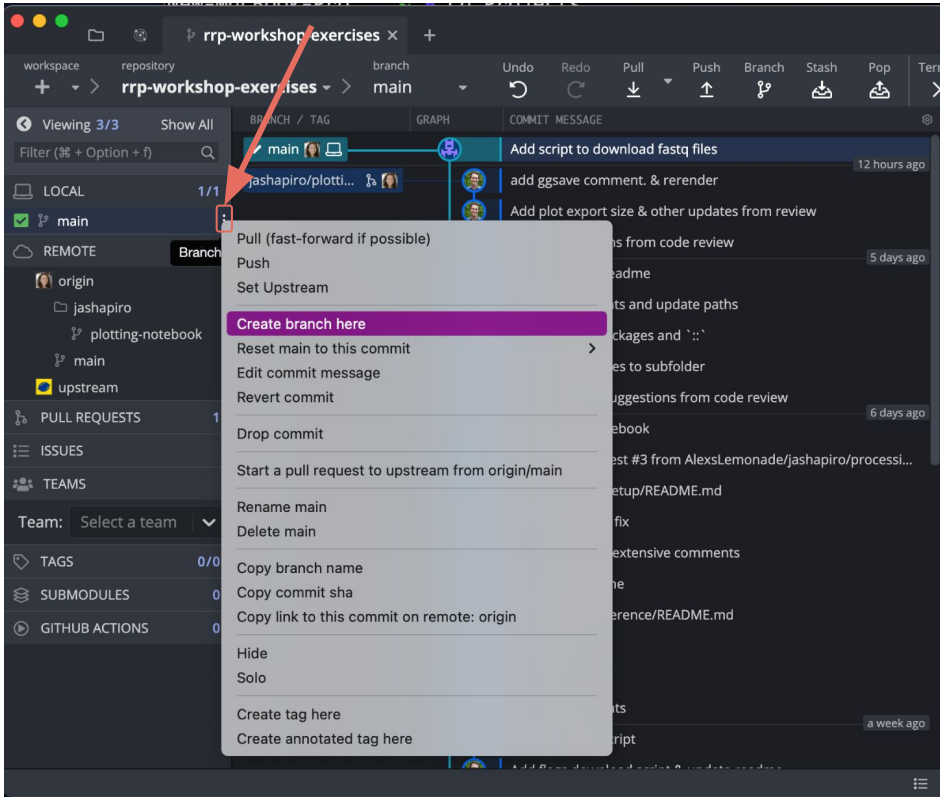
Modified from https://www.atlassian.com/git/tutorials/using-branches/git-merge

Creating a new branch

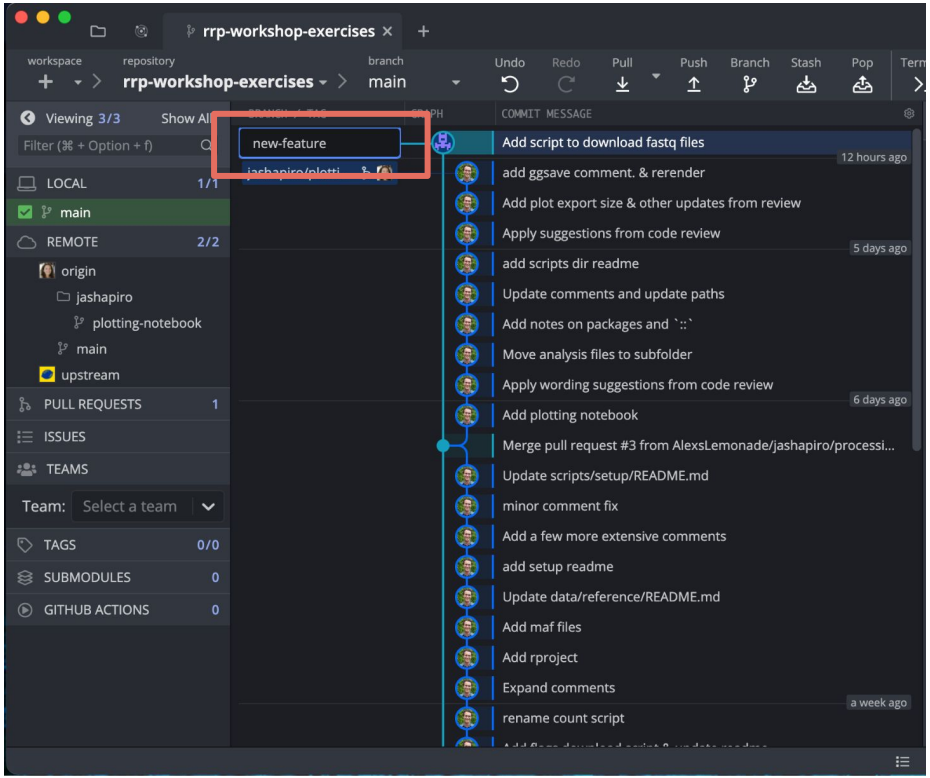


Because we are currently on the `main` branch, this new branch will be created off of the `main` branch's history.

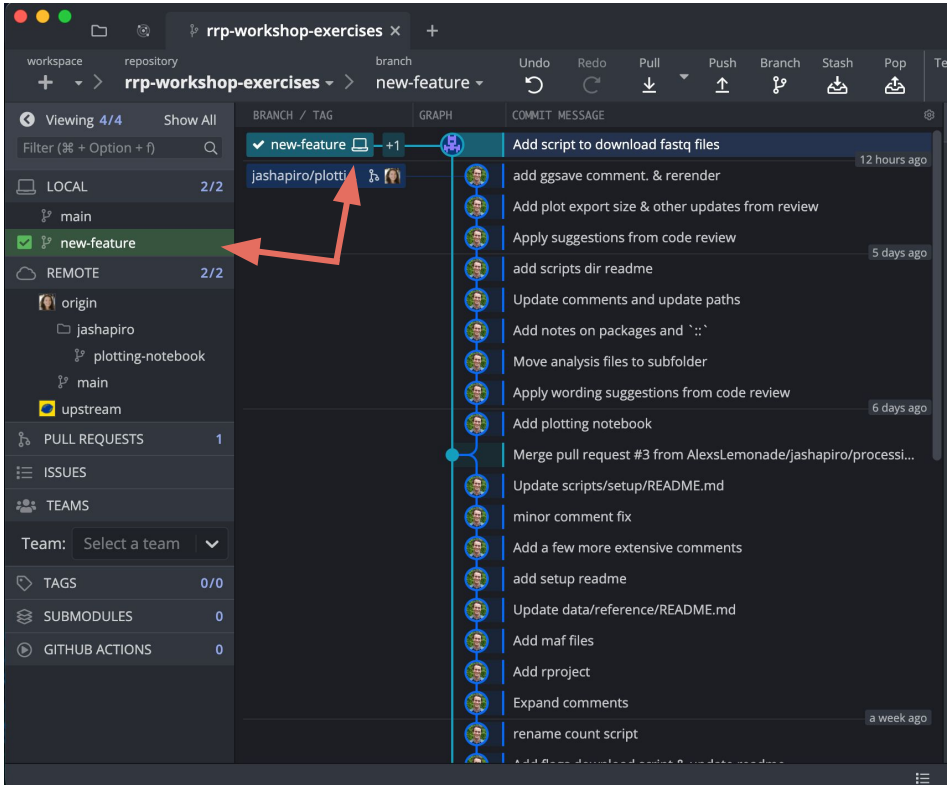
Alternatively…



Name your new branch by typing it in here



Now you're in your new branch!

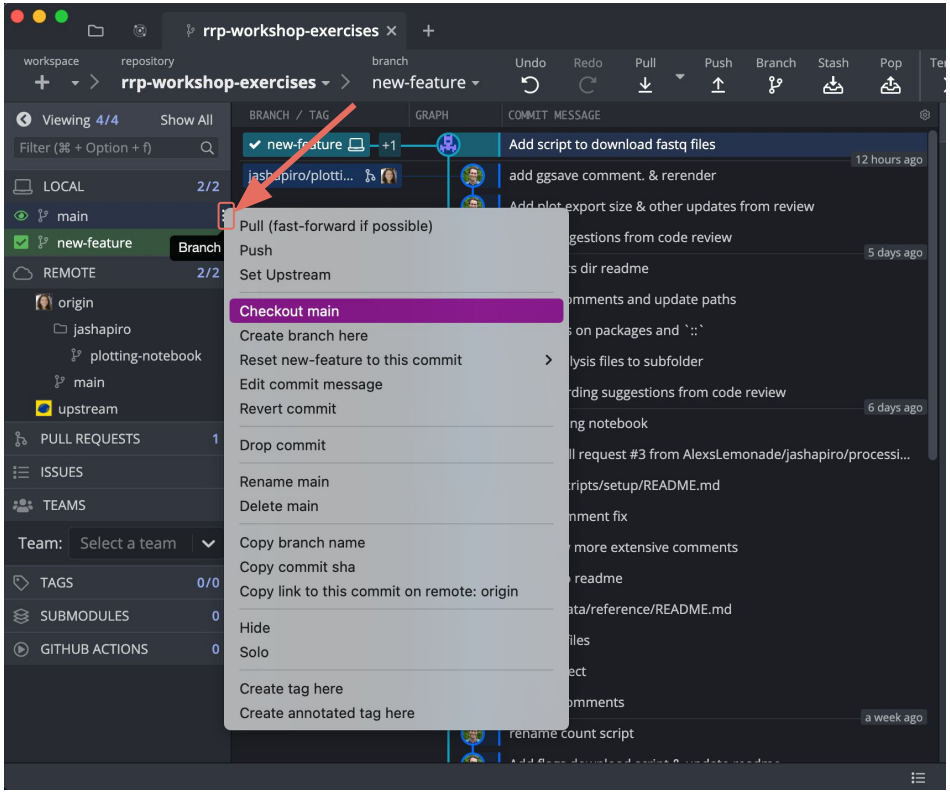


Note the different *icons* associated with **local** vs. **remote** repositories

- Local is a laptop icon
- *In this case*, remote is Stephanie's GitHub profile picture (but you aren't Stephanie!)

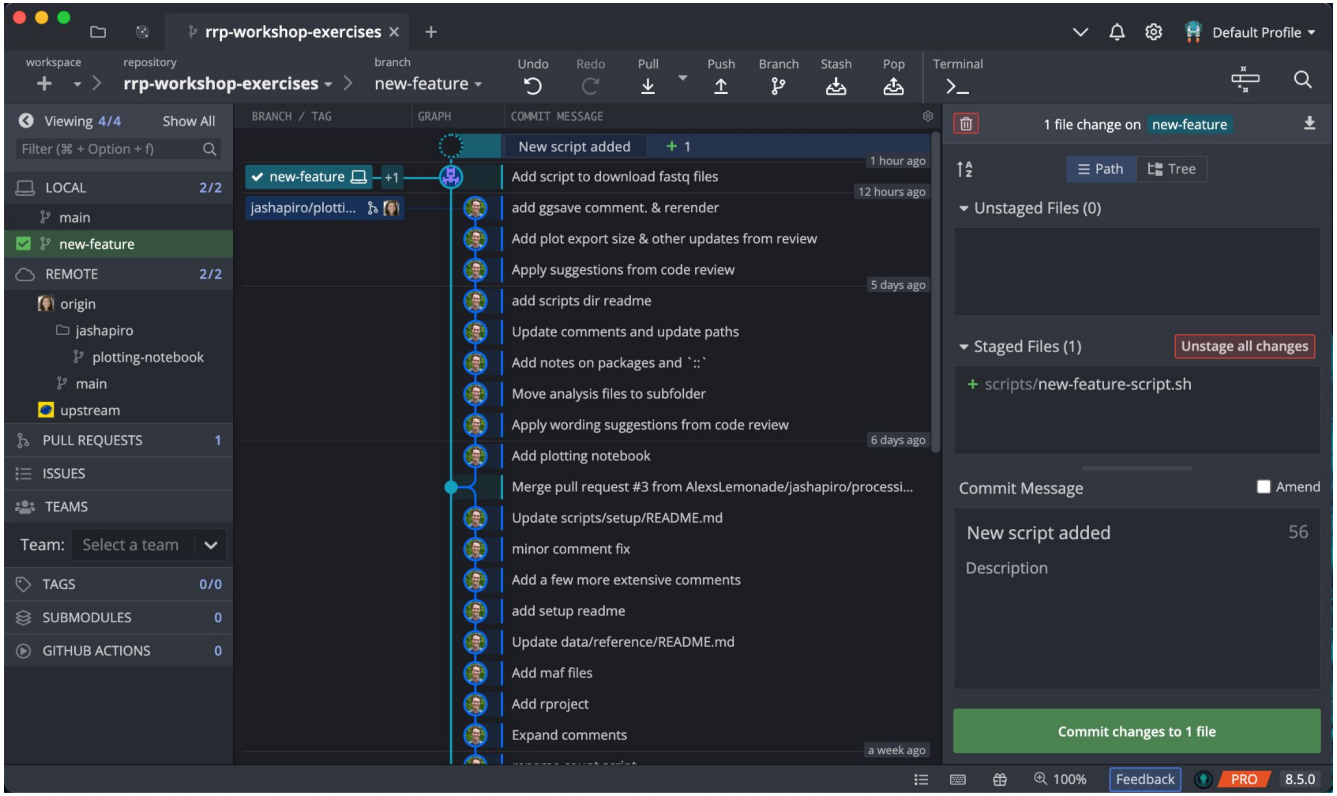
We created `new-feature` *locally*, so it does not (yet!) exist on our *remote*.

We can switch between branches with **checkout**

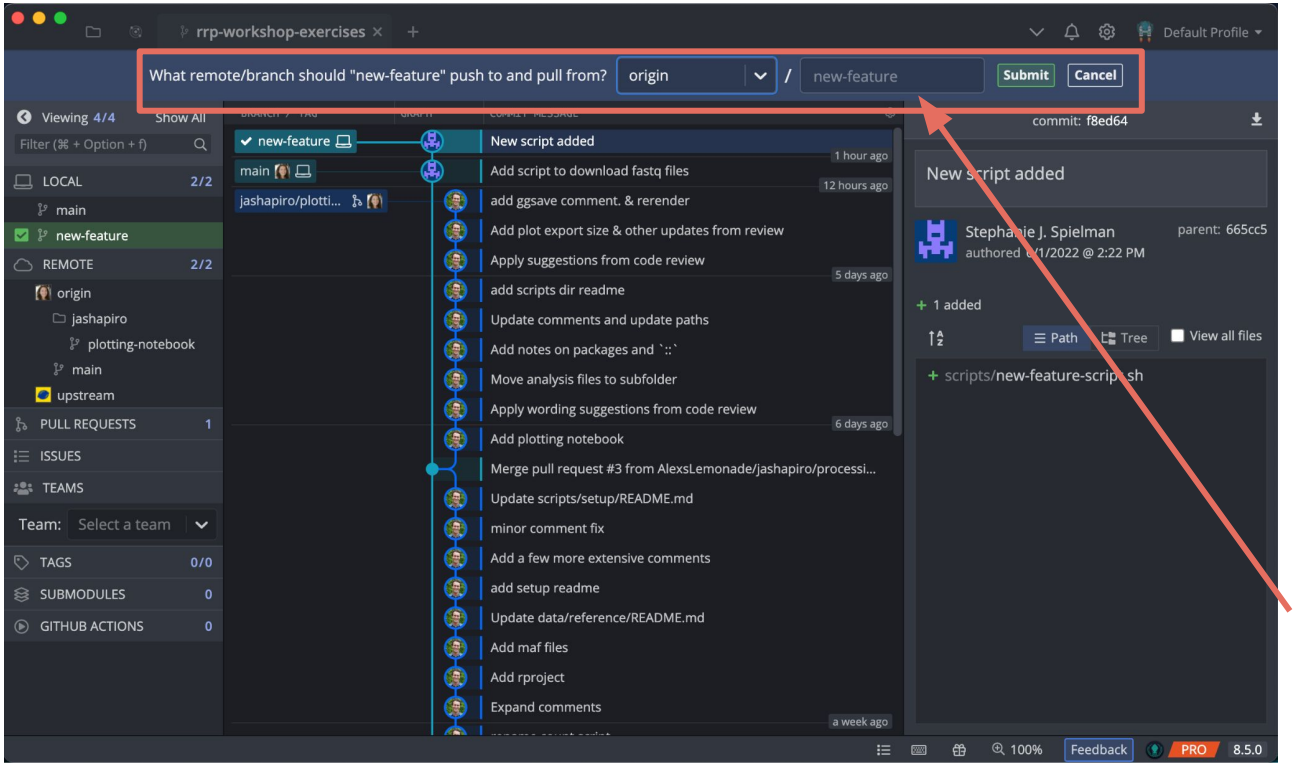


You can also double-click on a branch in GitKraken!

Make commits within your new branch as usual



Pushing prompts you to specify the remote branch

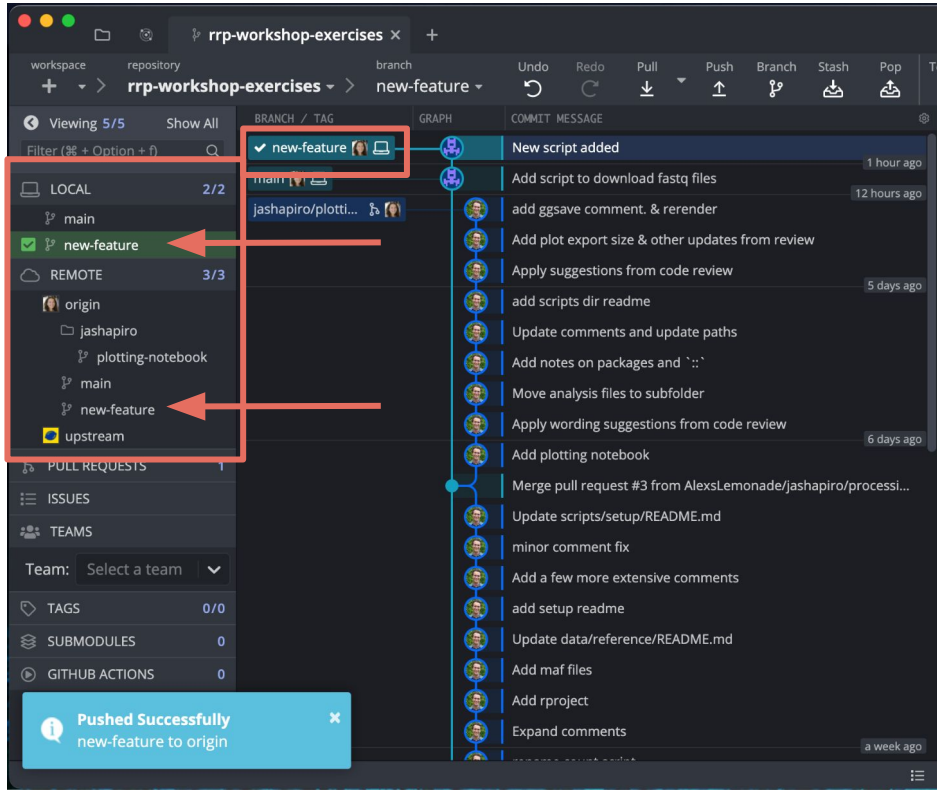


Because this `new-feature` branch doesn't exist (yet!) on the remote, `git` needs more info about where to push to.

This prompt will always occur *the first time* you push from a brand new branch.

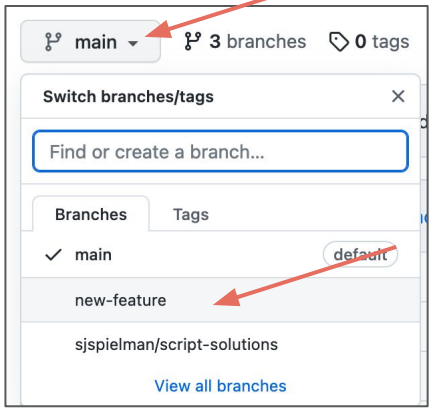
GitKraken helpfully guesses what you want your remote branch to be named! Click "Submit."

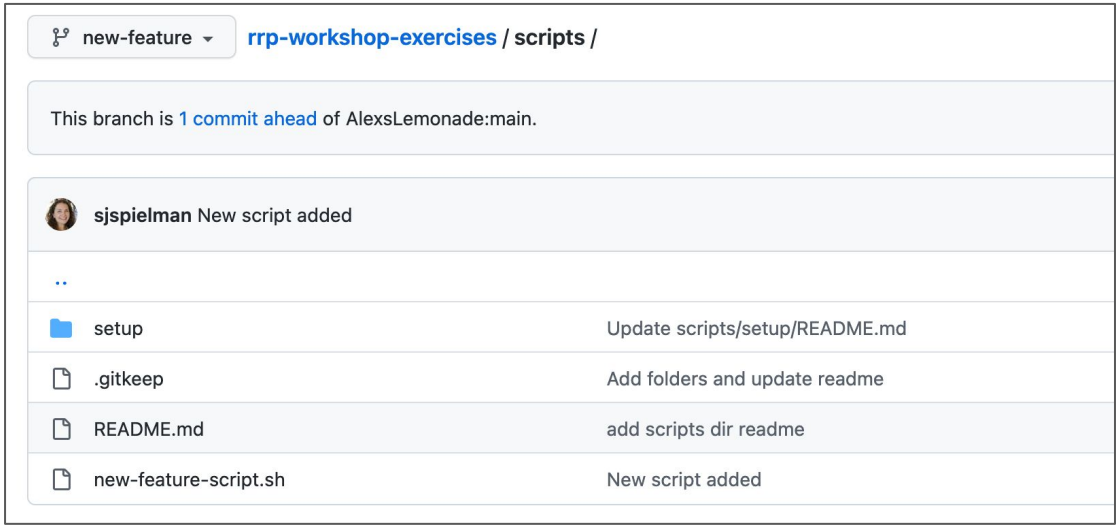
Pushing has created a corresponding *remote* branch



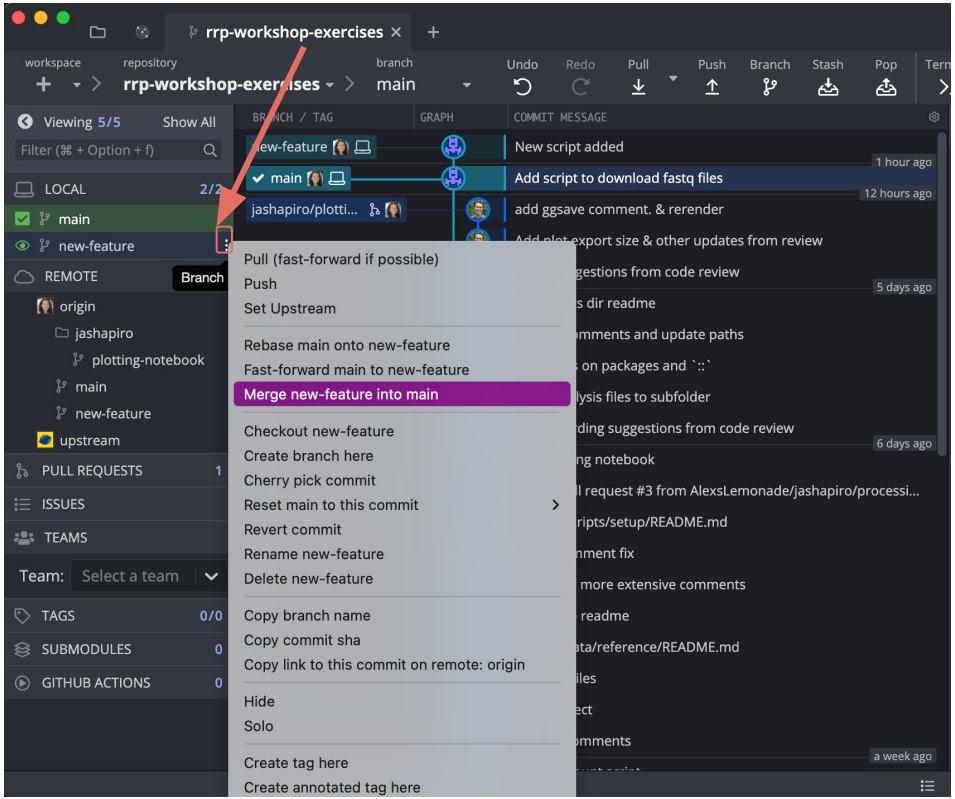
Have a look at the icons!
`new-feature` is now fully "synced" between local and remote.

Viewing branches on GitHub





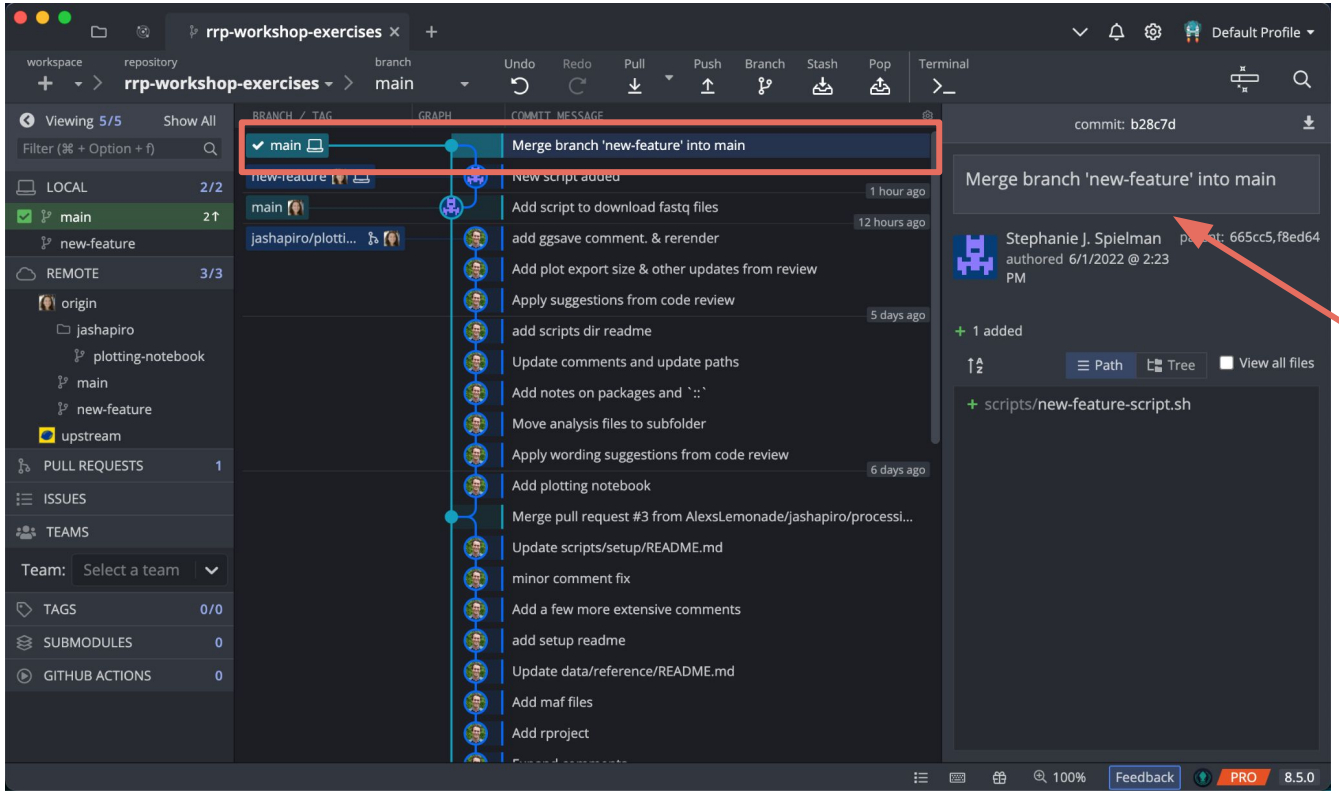
Merge into your main branch



But note!! You have to be in the branch you are merging into (see how `main` is highlighted in green?)

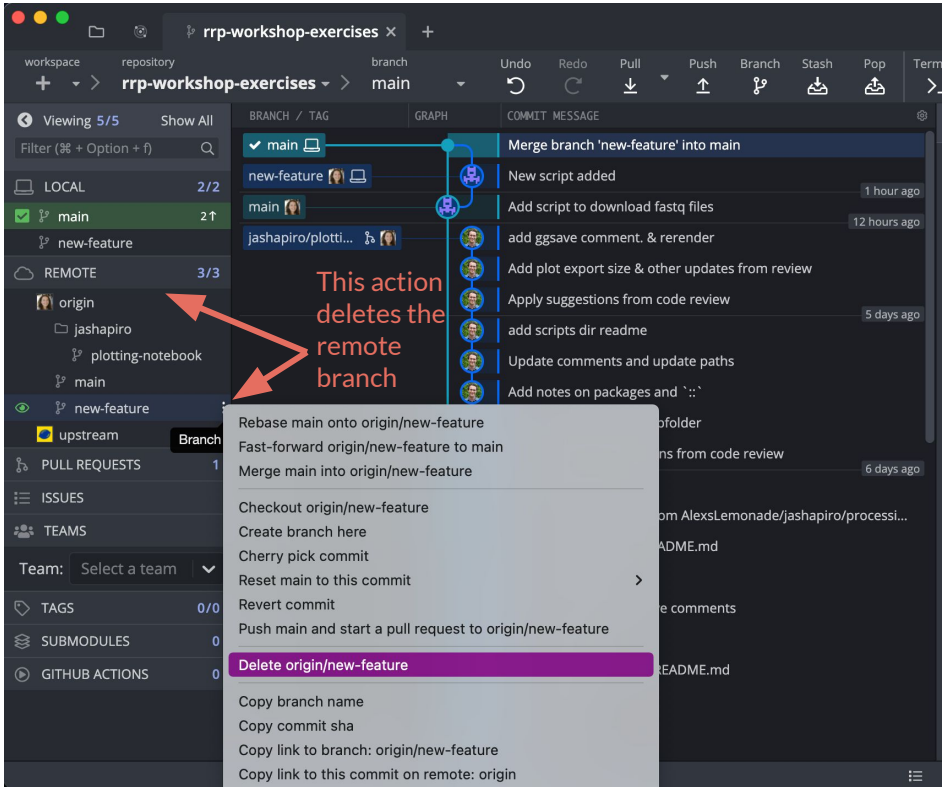
Bonus!! You can also point-and-click *drag* branches to merge them. Here, drag `new-feature` into `main` in the sidebar

Voila, your local histories have merged!



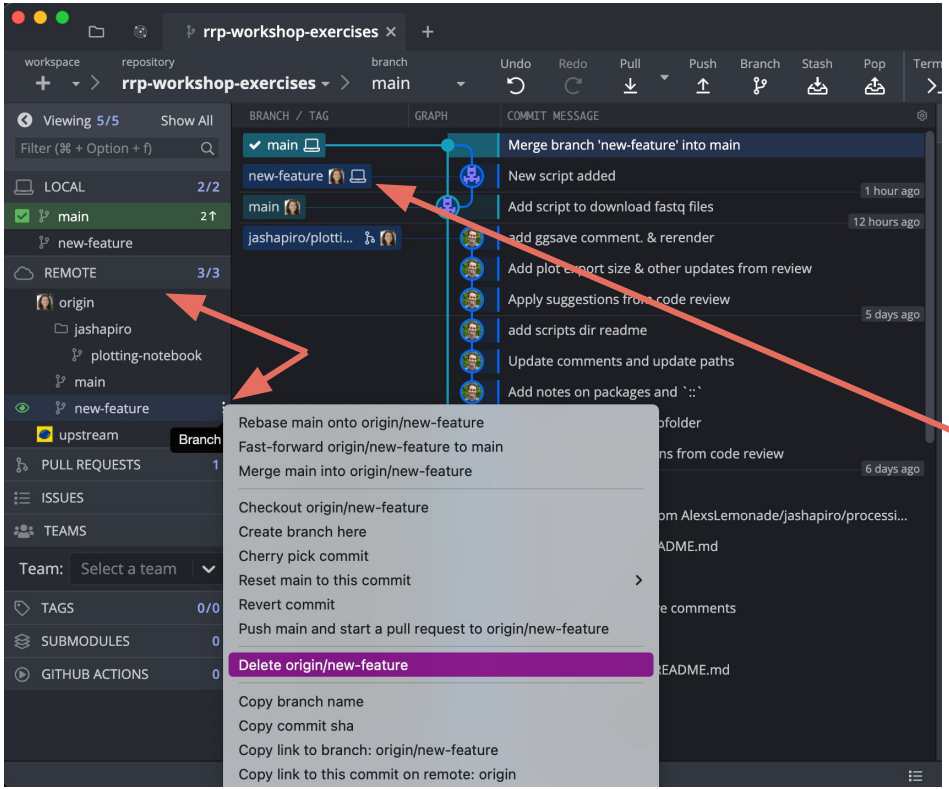
The merge itself is a commit to the `main` branch, with an automatic commit message

You can now safely delete your branch



Importantly, deleting branches that have been merged *does not* delete their commits! Those commits are part of the `main` branch's history now.

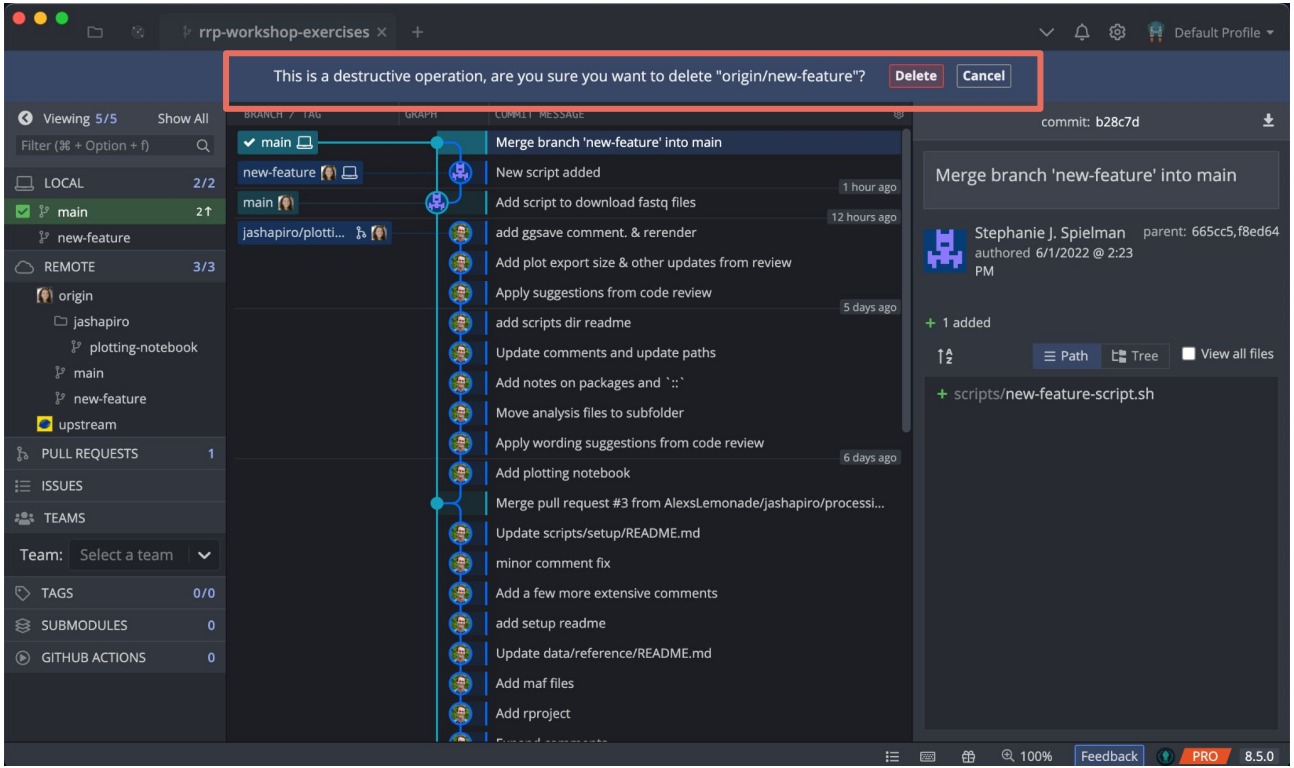
You can now safely delete your branch



Importantly, deleting branches that have been merged *does not* delete their commits! Those commits are part of the `main` branch's history now.

You can also conveniently delete *both* remote and local branches at once by right-clicking the joint label in the *source graph*.

Are you sure?? If you've merged: yes, you're sure.



Branches and merging on the command line

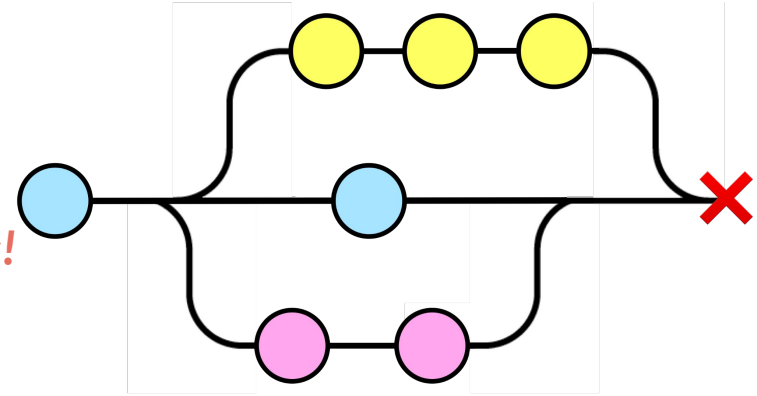
```
##### Making new branches

# Step 1: Create a new branch called `new-feature`
git branch new-feature

# Step 2: Switch to the new branch
git checkout new-feature

# Alternatively, create and switch in a single command
# Single step option:
git checkout -b new-feature
```

```
#### Merge new-feature into main:

# Step 1: make sure you are in the main branch!
git checkout main

# Step 2: Merge new-feature in
git merge new-feature
```

Merge conflicts can happen!

- If a given file has been *heavily* modified, `git` may not be able to merge the different file versions across branches together automatically
  - This is especially a problem if there are many branches floating around getting merged into each other
  - Imagine if each of these branches modified the same file in drastically incompatible ways……..

→ You will have to *manually fix the* ***merge conflicts!***



histologies_metadata.tsv

```
1  Biospecimen_ID  primary_site
2  BS_HZV4WDTB Frontal Lobe;Parietal Lobe
3  BS_E1SWA20C Peripheral Whole Blood
4  BS_KB9GJDCS Cerebellum/Posterior Fossa;Ventricles
5  BS_2EJWS3SD Peripheral Whole Blood
6  BS_6YMJ621P Skull
```

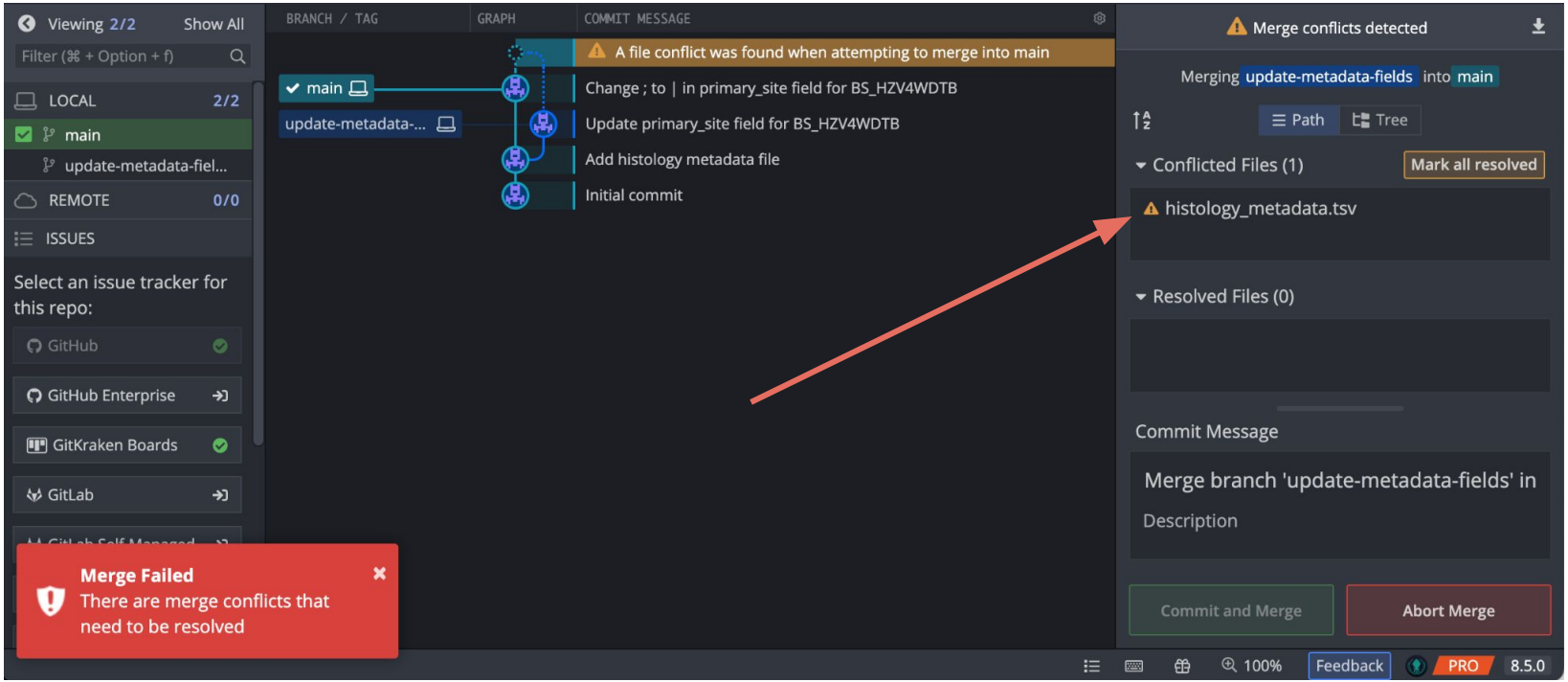
main

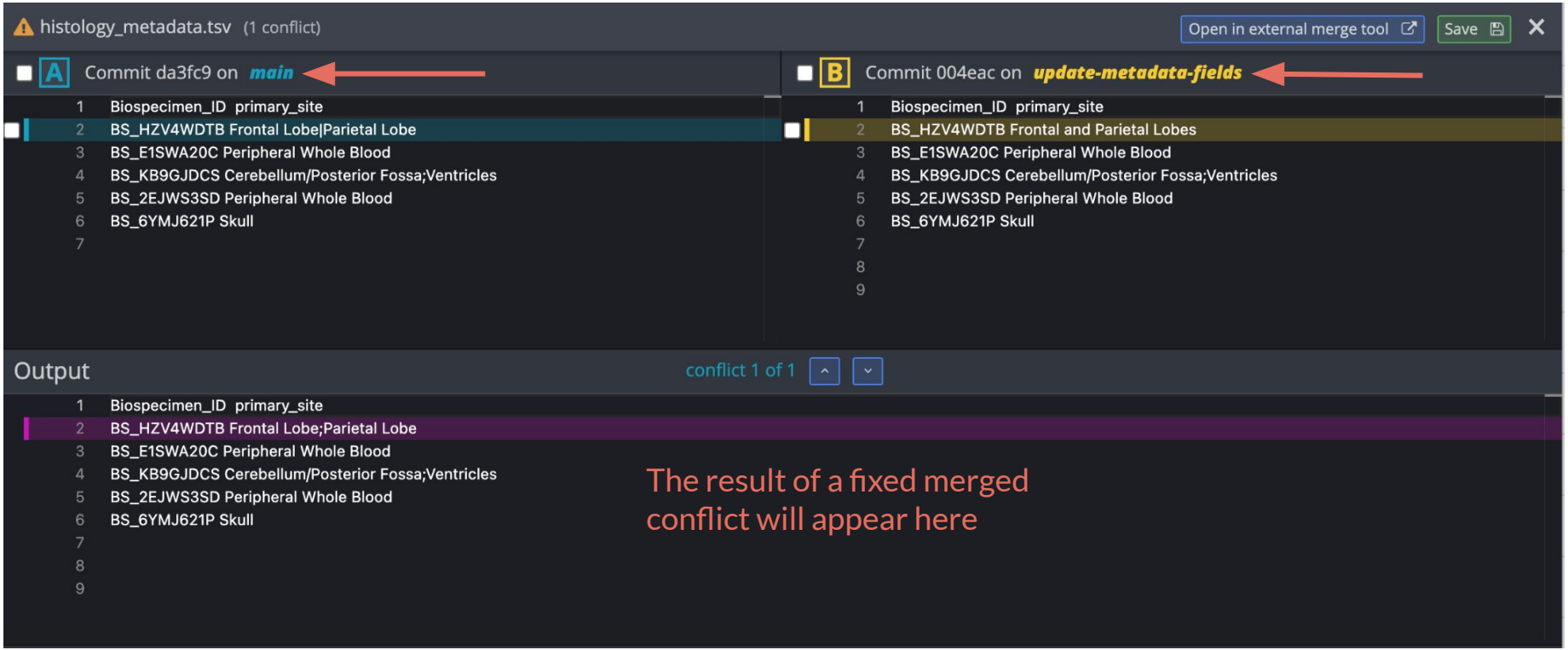
```
1  Biospecimen_ID  primary_site
2  BS_HZV4WDTB Frontal Lobe|Parietal Lobe
3  BS_E1SWA20C Peripheral Whole Blood
4  BS_KB9GJDCS Cerebellum/Posterior Fossa;Ventricles
5  BS_2EJWS3SD Peripheral Whole Blood
6  BS_6YMJ621P Skull
```

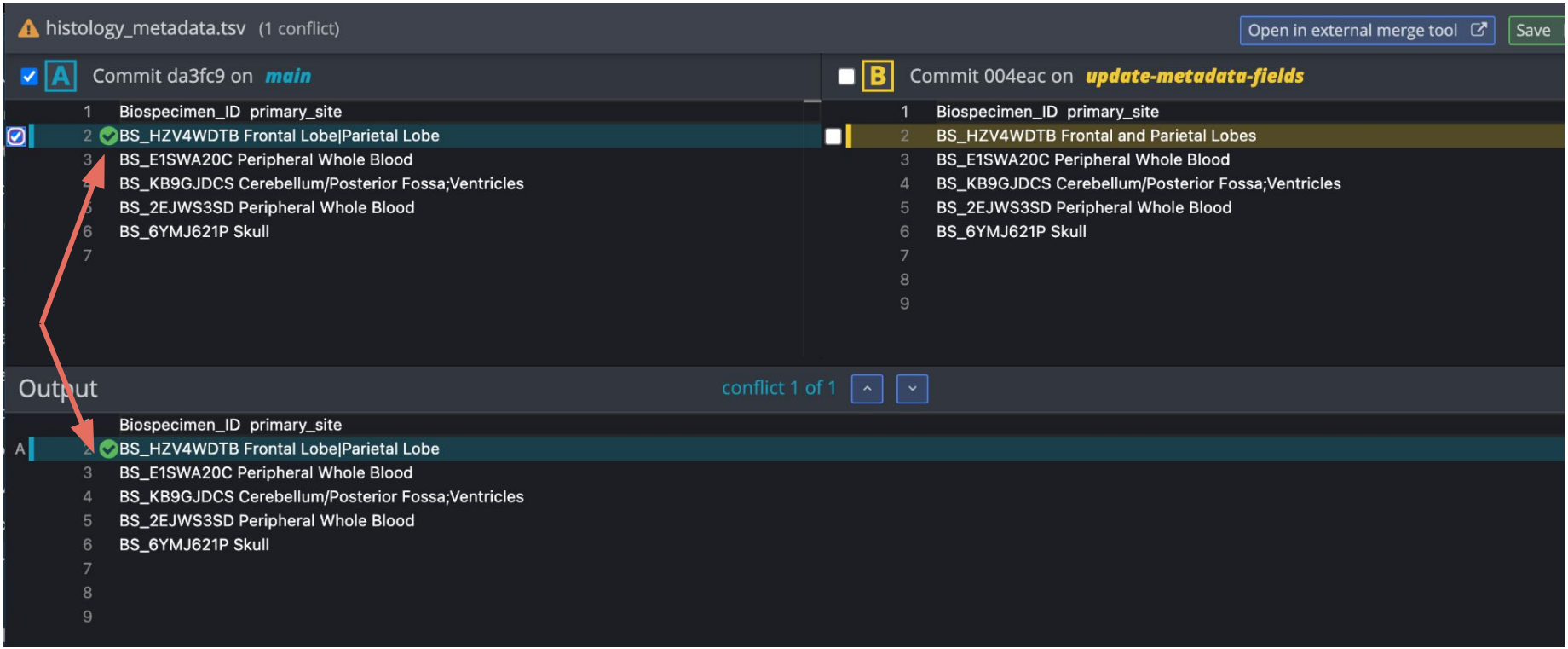
update-metadata-fields

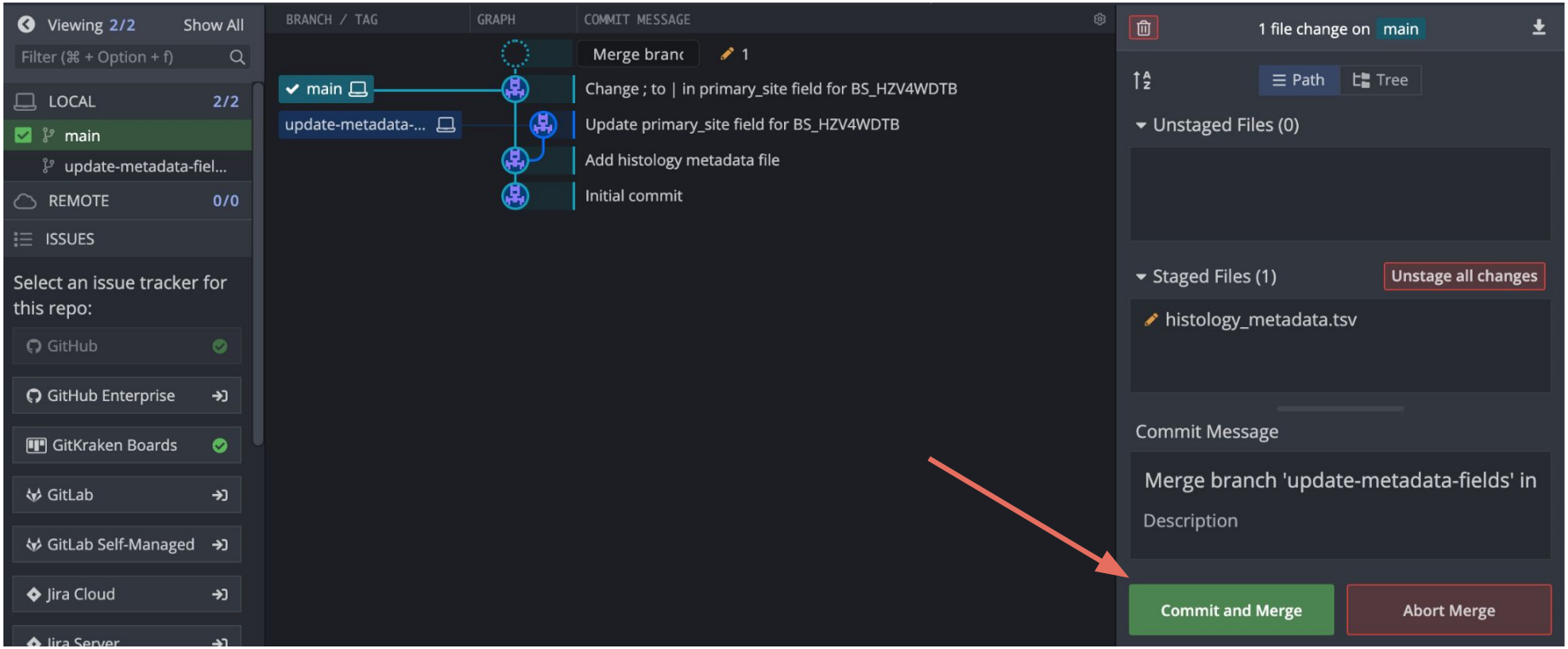
```
1  Biospecimen_ID  primary_site
2  BS_HZV4WDTB Frontal and Parietal Lobes
3  BS_E1SWA20C Peripheral Whole Blood
4  BS_KB9GJDCS Cerebellum/Posterior Fossa;Ventricles
5  BS_2EJWS3SD Peripheral Whole Blood
6  BS_6YMJ621P Skull
```

Merging `update-metadata-fields` into `main` causes a **merge conflict**




histology_metadata.tsv (1 conflict)
Open in external merge tool
Save
A
Commit da3fc9 on main
B
Commit 004eac on update-metadata-fields
1 Biospecimen_ID primary_site
2 BS_HZV4WDTB Frontal Lobe|Parietal Lobe
3 BS_E1SWA20C Peripheral Whole Blood
4 BS_KB9GJDCS Cerebellum/Posterior Fossa;Ventricles
5 BS_2EJWS3SD Peripheral Whole Blood
6 BS_6YMJ621P Skull
1 Biospecimen_ID primary_site
2 BS_HZV4WDTB Frontal and Parietal Lobes
3 BS_E1SWA20C Peripheral Whole Blood
4 BS_KB9GJDCS Cerebellum/Posterior Fossa;Ventricles
5 BS_2EJWS3SD Peripheral Whole Blood
6 BS_6YMJ621P Skull
Output
conflict 1 of 1
1 Biospecimen_ID primary_site
2 BS_HZV4WDTB Frontal Lobe;Parietal Lobe
3 BS_E1SWA20C Peripheral Whole Blood
4 BS_KB9GJDCS Cerebellum/Posterior Fossa;Ventricles
5 BS_2EJWS3SD Peripheral Whole Blood
6 BS_6YMJ621P Skull
The result of a fixed merged conflict will appear here


histology_metadata.tsv (1 conflict)
Open in external merge tool
Save
A
Commit da3fc9 on main
B
Commit 004eac on update-metadata-fields
1 Biospecimen_ID primary_site
2 BS_HZV4WDTB Frontal Lobe|Parietal Lobe
3 BS_E1SWA20C Peripheral Whole Blood
4 BS_KB9GJDCS Cerebellum/Posterior Fossa;Ventricles
5 BS_2EJWS3SD Peripheral Whole Blood
6 BS_6YMJ621P Skull
7
1 Biospecimen_ID primary_site
2 BS_HZV4WDTB Frontal and Parietal Lobes
3 BS_E1SWA20C Peripheral Whole Blood
4 BS_KB9GJDCS Cerebellum/Posterior Fossa;Ventricles
5 BS_2EJWS3SD Peripheral Whole Blood
6 BS_6YMJ621P Skull
7
8
9
Output
conflict 1 of 1
1 Biospecimen_ID primary_site
A 2 BS_HZV4WDTB Frontal Lobe|Parietal Lobe
3 BS_E1SWA20C Peripheral Whole Blood
4 BS_KB9GJDCS Cerebellum/Posterior Fossa;Ventricles
5 BS_2EJWS3SD Peripheral Whole Blood
6 BS_6YMJ621P Skull
7
8
9


Viewing 2/2
Show All
Filter (⌘ + Option + f)
LOCAL 2/2
main
update-metadata-fiel...
REMOTE 0/0
ISSUES
Select an issue tracker for this repo:
GitHub
GitHub Enterprise
GitKraken Boards
GitLab
GitLab Self-Managed
Jira Cloud
BRANCH / TAG
GRAPH
COMMIT MESSAGE
Merge branc
1
main
update-metadata-...
Change ; to | in primary_site field for BS_HZV4WDTB
Update primary_site field for BS_HZV4WDTB
Add histology metadata file
Initial commit
1 file change on main
Path
Tree
Unstaged Files (0)
Staged Files (1)
Unstage all changes
histology_metadata.tsv
Commit Message
Merge branch 'update-metadata-fields' in
Description
Commit and Merge
Abort Merge

VS Code also has helpful git integration!

```
1   Biospecimen_ID  primary_site
    Accept Current Change | Accept Incoming Change | Accept Both Changes | Compare Changes
2   <<<<<<< HEAD (Current Change)
3   BS_HZV4WDTB Frontal Lobe|Parietal Lobe
4   =======
5   BS_HZV4WDTB Frontal and Parietal Lobes
6   >>>>>>> update-metadata-fields (Incoming Change)
7   BS_E1SWA20C Peripheral Whole Blood
8   BS_KB9GJDCS Cerebellum/Posterior Fossa;Ventricles
9   BS_2EJWS3SD Peripheral Whole Blood
10  BS_6YMJ621P Skull
```

A reminder: git can be tricky!!

- You will make mistakes
  - *That's ok! So do we, and so does everyone else!*
  - Git and GitKraken error messages will try to help you
- But sometimes you will just want to curse

Dangit, Git!?!

Git is hard: messing up is easy, and figuring out how to fix your mistakes is impossible. Git documentation has this chicken and egg problem where you can't search for how to get yourself out of a mess, *unless you already know the name of the thing you need to know about* in order to fix your problem.

So here are some bad situations I've gotten myself into, and how I eventually got myself out of them *in plain english*.

https://dangitgit.com/en

(and its other version, https://ohs**tgit.com, but with those letters filled in!)

Many thanks to everyone who has volunteered to translate the site into new languages, you rock! Michael Botha (af) · Khaja Md Sher E Alam (bn) · Eduard Tomek (cs) · Moritz Stückler (de) · Franco Fantini (es) · Hamid Moheb (fa) · Senja Jarva (fi) · Michel (fr) · Alex Tzimas (gr) · Elad Leev (he) · Aryan Sarkar (hi) · Ricky Gultom (id) · fedemcmac (it) · Meiko Hori (ja) · Zhunisali Shanabek (kk) · Gyeongjae Choi (ko) · Rahul Dahal (ne) · Martijn ten Heuvel (nl) · Łukasz Wójcik (pl) · Davi Alexandre (pt_BR) · Catalina Focsa (ro) · Daniil Golubev (ru) · Nemanja Vasić (sr) · Björn Söderqvist (sv) · Kitt Tientanopajai (th) · Taha Paksu (tr) · Andriy Sultanov (ua) · Tao Jiayuan (zh) . With additional help from Allie Jones · Artem Vorotnikov · David Fyffe · Frank Taillandier · Iain Murray · Lucas Larson · Myrzabek Azil

Translated into over 20 languages, and counting!