An Introduction to Git and GitHub

A Practical Guide for Busy Researchers

Callum Arnold

2025-09-22

Table of contents

Welcome, and What This Book is About

This book accompanies the Pennsylvania State University's Center for Infectious Disease Dynamics short workshop on using Git and GitHub as researchers. The content will form the basis of the workshop's syllabus, and act as a reference for attendees (and others), although I would highly recommend reading through the excellent book by Jenny Bryan and co., as well as the GitHub docs for additional information. I also strongly recommend looking at the Atlassian Git tutorials for excellent in-depth tutorials about Git, and Learn Git Branching for an interactive way to learn Git! As you become more familiar with Git, it's worth checkout out the official Git documentation and book, which provides a wealth of information about Git and its internals.

Motivation

As mentioned above, there are plenty of great resources out there for learning Git and GitHub. So why write another one? Well, in part, I wanted to try and consolidate the information out there into a book that doesn't have a bias and focus on R and R-Studio tools, which many of the more introductory resources do, and without getting too deep into the weeds like some of the resources aimed at software developers rather than busy researchers.

Who Should Read This Book?

This book is aimed at researchers who are interested in using Git and GitHub to manage their research projects, so really that should be everyone who does any computational work as part of their research! Because this book accompanies an introductory workshop, we will work from the fundamentals up, so you don't need to have any prior experience with Git or GitHub.

However, that does not mean that this won't be of any use to you if you already know the basics. The plan of this book is to keep updating it with new content as I my own research and experience with Git and GitHub progresses, and as new tools are developed, with the latter half of the book including more advanced topics that can act as a reference for attendees to follow up on after the workshop, as well as a resource for more experienced users who come across this.

In part, this latter section will be a place for me to keep track of the things I've learned, and to share them with others. As you use Git, you will inevitably put yourself in a position where you realized you've messed up and you need to try and back out of the situation you've created. I've been there, and will certainly experience this again, so I'll try and document the solutions I've found to these problems here. This second half of the book will also be where I outline more general research workflows that I've found useful, and how I've integrated them with Git and GitHub. Research project management is a large topic in its own right, so I won't be able to, and shouldn't, cover everything here, but if it relates to Git, it'll be included.

Workshop Pre-Requisites

There are some pre-requisite tasks to get set up ahead of the workshop. Because we only have 2 hours, there is not enough time to go over the installation of Git and GitHub if we want to get to a point where we can understand and troubleshoot our way through actually using these tools, so you'll need to do that ahead of time. Please set aside about an hour to try and get this set up. I'm hoping that you should be able to do this in about 30 minutes, but as with all things computational, it's worth including some buffer time in case you run into issues.

The pre-requisite sections cover the following topics and questions:

1. What are Git and GitHub, and why do I care?
2. How to install Git
3. Setting up a GitHub account
4. Connecting GitHub to my machine

Keywords, Code, and Other Formatting

Throughout the book, you'll see some keywords, code, and other points that I'll try to delineate with the following formatting:

> **Note**
>
> This will be a note, and will be used to highlight important points, or to provide additional information.

> **Tip**
>
> This will be used to highlight a useful tip.

> ⚠ Warning
>
> This will provide a warning that you may get an unexpected result if you're not careful.

- `code` will be used to highlight code.
- `{package::function()}` will be used to denote a specific package and function, e.g., `{dplyr::mutate()}` denotes the `mutate()` function from the `{dplyr}` package. I will use this for all languages for consistency, even though some (like Python or Julia) don't use the `::` notation to export functions.
- **keywords** will be used to highlight keywords and phrases, e.g., **Git** or **GitHub**.
  - **actions** will also be highlighted in this way, e.g., **commits** or **pushed** being the result of the code `git commit` or `git push`
- ***files*** will be used to highlight file names, e.g., ***README.md*** or ***LICENSE***.
- *italics* will be used for emphasis in certain circumstances, e.g., signifying a question from an interactive terminal command.

Part I

Prerequisites

1 Git & GitHub Overview

1.1 What is Git?

If you're in this workshop, or have stumbled across this book, there's a good chance you already know what Git is, or have at least heard of it. However, if you don't and you've been told by someone you should start using Git, but have no idea what that even means, then hopefully this subsection will help.

Git is a version-control system (VSC). Think of it like a better version of Microsoft Word tracked changes and Google version history that can track everything from code, to text, to pdf images. Much like how tracked changes is useful for both single and multi-user documents, Git can help us remember what we've done, when, and what version of the document(s) previously existed, as well as denoting which user made the changes. Where tracked changes can get unwieldy after multiple iterations, Git makes it easy to understand the whole file history without needing to use ***document_v3*** filenames - just keep changing the same original file for as long as you want! In addition to just being able to understand a file history, when working on a project, even if you're the only one coding, it's important to be able to go back to previous versions if you make a mistake. This is possible with Git! In fact, this book was created using Git and GitHub, so you can explore how it was put together and iterated by going to the GitHub page. Git isn't the only VCS available, but it's the most prevalent, has some advantages over the alternatives in the method of storing its data, and has a good support community, so is what will be the focus in this book.

1.2 What is GitHub?

Hopefully I've convinced you that Git is a useful addition to your research, so now let's turn our attention to GitHub. GitHub is a website and server system makes it easy to collaborate and share your code with the scientific community. The key feature of Git is that it's a distributed VCS. What this means is that users can make changes to a file on their own computer and then **push** the updated version to GitHub so that all collaborators can then use this version of the file. In Git terminology, GitHub is your **remote**. Later on we'll go through the mechanics of this, including what happens when two users make changes to the same lines of code and try to **push** to GitHub, but for the moment we can just appreciate that GitHub allows us to both work offline and collaborate. There are many different remote services that can be used

to host our remote code, such as Bitbucket or GitLab, but I'd strongly recommend you use GitHub over the alternatives for a number of reasons. Principally, GitHub has the largest user base, so more people will likely see your work. With GitHub, if you ever want to make your code open-source, you immediately have access to the largest community of programmers who can help you improve your code, as well as putting it to good use. And isn't that why we do research? As an academic or student, you can get a free **PRO** account, meaning unlimited collaborators on private repositories and a bunch of other useful things that we'll touch on more later. GitHub is also owned and backed by Microsoft. While this is a negative for some, it does result in tighter integration with Azure cloud computing, very active development of the platform with frequent improvements to the user experience (e.g. GitHub Actions that allow for easy continuous integration), and fewer data storage concerns with regards to university policies. If you really want to avoid all Microsoft based products, I'd recommend you look at GitLab.

2 Installing Git

To get started, you first need to install Git. There are many ways to get Git running on your computer, but the recommended steps depend on the operating system you have.

2.1 Mac OS and Linux

If you're on Mac OS or Linux, you likely already have Git pre-installed. However, you are unlikely to have the most up-to-date version and I'd recommend you install it manually. If you do not already use a package manager, I would suggest you download homebrew as it is the most widely used and therefore can download the most applications. Homebrew also now works for Linux (see here for more details), and is useful as its packages can often be more up-to-date than those through some Linux package managers.

1. Open the terminal and enter `/usr/bin/ruby -e "$(curl -fsSL https:/raw.githubusercontent.com/`
2. Enter `brew install git` into the terminal
3. Enter `which git` into the terminal
    a. You should see `/opt/homebrew/bin/git`, if not, you may need to edit the environment variables

Using homebrew to install packages makes it easy to update them (including Git). All you need to do is type `brew update` and all your brew-installed packages are updated in one command!

2.2 Windows

Getting set up on Windows requires a bit more work as Windows doesn't come with a good terminal (command prompt doesn't count!). If you want to explore using Windows Subsystem for Linux (WSL), then go for it as it'd probably make your life easier as you get into more advanced things like cloud computing and remote servers (see here for more details, or use homebrew for linux), but for the moment, you can use the following steps to get started.

1. Install Git for Windows

- This gives you Git Bash, which is a much nicer way of interfacing with Git than the command line.

> **Note**
>
> When asked about "*Adjusting your PATH environment*", be sure to select "*Git from the command line and also from 3rd-party software*". The other default options should be fine. For more details about the installation settings, please click here

2. Open up Git Bash and enter `which git`, or open up the command prompt and enter `where git`. Depending on whether you have administrator privileges, the outputs should look something like this, respectively
    1. `which git` : `/mingw64/bin/git`
    2. `where git` : `C:\Users\owner\AppData\Local\Programs\git\bin\git.exe` (User privileges)
        1. `where git` : `C:\Program Files\git\bin\git.exe` (administrator privileges)
    - If you see `cmd` instead of `bin`, then you need to edit the PATH in your environment variables.

You could also install Git using Chocolatey, as this would provide you with a package manager that you can use to install other useful software, much like homebrew on Mac OS.

2.3 Final Git set up steps

Now that you have Git running, you need to tell it who you are. This allows multiple people to make changes to code, and the correct names will be attached to the changes. We will also make sure that all Git repositories use the default branch name **main**.

Open up the Git Bash or the terminal and enter

```
Git config --global user.name 'Firstname Lastname'
Git config --global user.email 'my_email@domain.com'
Git config --global init.defaultBranch main
```

Typing in `Git config --global --list` is a way to check that your details have been saved correctly.

> **Note**
>
> The email you use with Git will be published when you **push** to GitHub, so if you don't want your email to be public, you can use the GitHub-provided no-reply email address instead. The key points are that you need to turn on email privacy in your GitHub settings, and then using that address in your Git config.
> On another note, if you would prefer to use a different user name than your GitHub user name you can. This would help show you which computer you completed the work on, but it is not important to most people.

2.4 Troubleshooting

2.4.1 Environment Variables

If you are not able to access Git appropriately (i.e., from the terminal/Git bash), you may need to edit the environment variables.

In Windows you do this by navigating to *Environment Variables* from the Windows key/Start prompt and editing the PATH in *User Variables*. To do this, scroll to the PATH section of User/System variables (depending on whether you have administrator privileges), and changing `cmd` to `bin` in the `git.exe` path.

In Mac, you should open the terminal and use `vi`/`touch`/`nano` command to edit the ~**/.zshrc** or ~**/.bashrc** file (depending on how old your Mac is - OSX switched to zsh in 2019) e.g., `vi ~/.zshrc`, which opens (or creates if missing) the file that stores your PATH. From here, type `export PATH="path-to-git-executable:$PATH"` to add the executable to the path. For me, using a Mac and homebrew, my Git path is `export PATH="/opt/homebrew/bin:$PATH"`. Now save and exit the text editor, source the file if on Mac (e.g., run `source ~/.zshrc` in your terminal), and you're good to go.

3 Setting up a GitHub Account

It's very easy to get set up on GitHub. Just click the link above and select the package you'd like. If you have an academic email address, consider making this your primary email address on the account, as it gives you a **PRO** account for free with access to more features. See here for more details about the differences. If you are a student, you should also sign up for the GitHub Student Developer Pack as this gives you free access to a bunch of useful tools, such as the GitKraken Git client mentioned earlier. If you are a teacher, you get access to a GitHub Teams account, which also comes with its own set of benefits (see here for more details).

Be sure to choose a user name that is easy to remember, and easy to find. I would suggest just using your name, or the username you have for other work-related accounts (e.g., Twitter). It's quite annoying to try and change this later, so spend a little time thinking about it now.

> **Note**
>
> You can choose a public-facing name that is different to your username, so you can just use your full name here if yours is long and you don't want to use it as your username.

Now you have a GitHub account set up, this is your **remote**. If you work on a project with collaborators, this can be shared with them. That way, collaborators can work on their own versions of the code on their **local** machine (computer), and when it's ready for other people to use/help write, they can `push` it to the **remote** where others can access it. Don't worry if you don't know what `push` is - we'll cover that soon.

3.1 Linking Git to GitHub

Now you have a GitHub account, you need link it to your local Git installation. There are a couple of different methods for doing this. The first is to use HTTPS and **Personal Access Tokens (PATs)**. This is somewhat easier to set up, but it's a little less secure and ultimately is more annoying to use as it often requires entering your username and password each time you connect to GitHub. The second is to use **SSH**, which is a little more complicated to set up, but is more secure and easier to use once it's set up. Unlike Jenny Bryan, I think it's worth the effort to set up SSH as you front load the work (though it's not too bad), and then you can just forget about it. Knowing SSH basics is also really useful as it's the basis for many

other things, such as connecting to remote servers and computing on clusters, so will serve you well moving forward.

Instead of trying to cover all eventualities for all operating systems, please work through the excellent GitHub docs on **PATs** and **SSH**. If you are having issues after working through the steps, then try reading Jenny Bryan's excellent Git guide, which covers both **PATs** and **SSH**. If nothing here works, let me know and I'll try to help!

> **Tip**
>
> If you use the GitKraken client (detailed here), you can use the instructions in their documentation to create and pass the SSH keys directly to GitHub. This assumes you have linked GitKraken to your GitHub account.

That being said, here's an outline of the steps you'll need to take to set up SSH (that are lightly edited from the GitHub documentation). If on Mac/Linux, this will be in the terminal, and if on Windows, this will be in Git Bash.

1. Create a new SSH key

```
# -t specifies the type of cryptography the key will use (ed25519 is more secure than the de
# -f specifies the file name to save the key to
# -C specifies the comment to add to the key i.e., your GitHub email address
ssh-keygen -t ed25519 -f ~/.ssh/github -C "24391445+arnold-c@users.noreply.github.com"
```

> **Note**
>
> If you're on MacOS, edit the *~/.ssh/config* file using the `open/vi/nano` commands e.g., `open ~/.ssh/config` to add the following lines:
>
> ```
> Host github.com
>   AddKeysToAgent yes
>   UseKeychain yes   # delete if you're not using a password for the SSH key
>   IdentityFile ~/.ssh/github
> ```
>
> If the file doesn't exist, you can create it using `touch ~/.ssh/config`.

2. Start the SSH agent:

```
eval "$(ssh-agent -s)"
```

3. Add the SSH key to the SSH agent:

```
## --apple-use-keychain tells the agent to use the macOS keychain to store the passphrase fo
# Omit if you're not using a password for the SSH key, or if you're using Windows/Linux
ssh-add --apple-use-keychain ~/.ssh/github
```

4. Copy the *public* key contents to your clipboard:

```
# print contents to terminal to manually copy
cat ~/.ssh/github.pub
```

5. Navigate to the *Settings* > *SSH and GPG Keys* section of your GitHub account and click the *New SSH key* button. Paste in the contents of the *public* key you copied earlier, and click save.

> **Note**
>
> It would be useful to add GitHub's public key fingerprints to your ***known_hosts*** file, so that you don't get a warning when you connect to GitHub. You can find the fingerprints here.
> Currently, you should edit the ***~/.ssh/known_hosts*** file to add the following lines:
>
> ```
> github.com ssh-ed25519 AAAAC3NzaC1lZDI1NTE5AAAAIOMqqnkVzrm0SdG6UOoqKLsabgH5C9okWi0dh2l9GKJ
> github.com ecdsa-sha2-nistp256 AAAAE2VjZHNhLXNoYTItbmlzdHAyNTYAAAAIbmlzdHAyNTYAAABBBEmKSEN
> github.com ssh-rsa AAAAB3NzaC1yc2EAAAADAQABAAABgQCj7ndNxQowgcQnjshcLrqPEiiphnt+VTTvDP6mHBL
> ```

6. Test you can connect to GitHub using SSH:

```
ssh -T git@github.com
```

> **Note**
>
> If you're on Windows, you will need to start the `ssh-agent` service before you can use SSH i.e., when you want to use GitHub. This can be done by running `eval$(ssh-agent -s)` in Git Bash. However, you can automate this every time you open Git Bash by adding the following line to your ***~/.bashrc*** file:

```
env=~/.ssh/agent.env

agent_load_env () { test -f "$env" && . "$env" >| /dev/null ; }

agent_start () {
    (umask 077; ssh-agent >| "$env")
    . "$env" >| /dev/null ; }

agent_load_env

# agent_run_state: 0=agent running w/ key; 1=agent w/o key; 2=agent not running
agent_run_state=$(ssh-add -l >| /dev/null 2>&1; echo $?)

if [ ! "$SSH_AUTH_SOCK" ] || [ $agent_run_state = 2 ]; then
    agent_start
    ssh-add
elif [ "$SSH_AUTH_SOCK" ] && [ $agent_run_state = 1 ]; then
    ssh-add
fi

unset env
```

4 Installing a Git Client

You've now installed Git, GitHub, and you're ready to get going! Much like most pieces of software, we can interact with Git via a command line or using a graphical user interface (GUI). There's a small vocal minority of people that proclaim that you can't learn Git with a GUI (aka Git client), but don't listen to them! There are plenty of good Git clients out there that make the basic commands simple, and provide a visual for more complicated ideas. I prefer to use the GitKraken client, which is free to use for students and academics if you sign up to the GitHub developer pack, but only allows access to a limited number of private repositories otherwise, so you may want to explore other options if that's you. GitHub for Desktop is made by the GitHub team, so as you can imagine it is tightly integrated with GitHub. Whether this is a pro or a con will depend on whether you think you'll explore other remote hosting services, but the reason I've avoided it is that it doesn't have a method of visualizing **branches**. You'll have to decide for yourself if this is a deal-breaker for you and your workflow. If you use VSCode as your development IDE, there's a built in Git client (the *Source Control panel*), and you can install the *Git Graph* extension to visual branches. The *GitLens VSCode extension* is also worth installing, and now offers paid features to further extend its use, although if you have a GitKraken Pro account you get these for free. In case you come across it in other recommendations, SourceTree was another good alternative, but I have had some issues connecting to some GitHub accounts, and it has limited support, so I have since moved away from it.

As you get a better understanding of Git, you may want to use the command line as it can be quicker to use, and is more powerful. If you decide to go this route, I'd recommend looking at the lazygit plugin which brings a GUI to the terminal, enabling you to get (most of) the best of both worlds. It's a little out of the scope of this workshop, but there are some useful video tutorials linked on lazygit's GitHub page.

4.1 Connecting GitKraken with GitHub

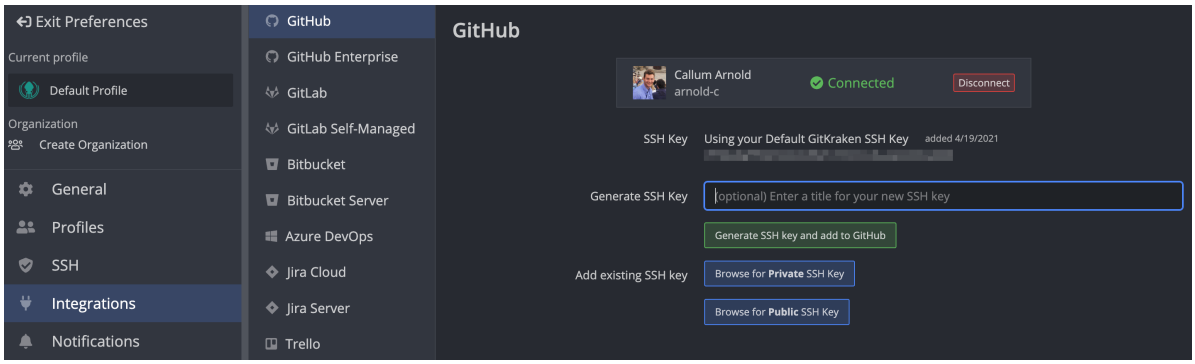
Assuming you use want to use GitKraken, you can connect it to GitHub to make for a much smoother experience. The full details are laid out in the GitKraken documentation, but in summary:

- When you set up GitKraken for the first time, choose *"Sign in with GitHub"* to use OAuth to log in to your GitHub account and create a link with the GitKraken application

- Open up the *"Preferences"* window and navigate to *"Integrations"* then *"GitHub"*
  - Make sure it shows your GitHub account as *"Connected"*

To connect your local Git installation with GitHub, you may want to use SSH keys. GitKraken makes this super easy. From the *"Preferences > Integrations > GitHub"* window, enter a name for your SSH key e.g., `github-ssh` and click on the *"Generate SSH key and add to GitHub"* button. It's as simple as that!

If you would like to manually create your SSH keys, you can do so using the instructions linked here, and then use the *"Browse for Private/Public SSH Key"* buttons to point to the location you saved your SSH keys. On a Mac and Linux, the default location is **~/.ssh/**. If you have not already added your public key to GitHub, GitKraken should detect this and provide you with a button to do so. You could also add these keys through the *"Preferences > SSH"* window where the *"Browse for Private/Public SSH Key"* buttons also appear.



Part II

Working with Git Fundamentals

5 How Git Works

Before we get too deep into how to use Git, it's a good idea to get a better understanding of how Git works. At a basic level, we know that it is a version control system, and we can think of it like track changes for our code. But it's a little more complicated than that, so we'll break it down into its component parts.

5.1 Repositories

I've touched on the idea of a repository, but what is a repository? A repository is just a different way of saying a folder that houses everything related to a project, AKA a project directory. You don't need to use Git to have a repository, per se, but it's a good idea to use Git to manage your repository for the reasons we've already discussed. Given you're using Git to manage your repository, Git will keep track of every file in the repository, and every change to those files. You can, however, tell Git to ignore certain files, and it will do so. This is useful for things like log files and html outputs, which are not part of the code, but are generated by the code and will take up a lot of space when you push to GitHub.

5.2 Commits

So how does Git keep track of all these changes? It does so by creating **commits**. A **commit** is a snapshot of every file in a repository, along with its changes since the last snapshot, at a given point in time. You can think of it like saving a file in a word processor, and is an action that has to be done manually. We'll talk in more detail later about how to do this, but a key idea is to **commit** often, and **commit** early.

5.3 Branches

Branching is a complex but powerful feature of Git. It allows you to make divergent copies of your repository, and then merge them back together. This is useful for a number of reasons, but the main one is that it allows you to work on different parts of the codebase at the same time, without having to worry about conflicting changes. We'll talk more about branching at the end of the workshop, but as a thought experiment, imagine you're working on a project

and have a new collaborator that is going to help out with some of the code. When you first set up a repository, you'll be on the **main** branch, so all of your code will be made here. Now your collaborator joins the project, and you're both working on different parts of the code. You're working on the code that generates the plots, and your colleague is working on the code that generates the tables. You both need to make changes to the same file (say, the manuscript Quarto file), but you don't want to have to wait for your colleague to finish their changes before you can start working on yours. You can both create a **feature** branch off **main** branch of the repository, make your changes, and then merge them back together when you're done. This is a very simplified example, but it gives you an idea of how branching works.

> **i** Note
>
> I used **main** to signify the default branch name, but did not for the example **feature** branches. This is because you should name your branches something meaningful, and not just **feature**. I'll talk more about this later.

5.4 Remotes

Up until now, everything we've talked about has been local to your computer. But an integral part of Git is that it is a **distributed** version control system. This means that you can have a copy of your repository on your computer, and also on a remote server (or multiple in some cases).

The key thing to remember is that the remote repository works via asynchronous communication. This means that you can make changes to your local repository, and then **push** those changes to the remote repository. Collaborators can then **pull** those changes from the remote repository to their local repository. If multiple collaborators are working on the same repository at the same time, it can be easy for their local versions to get out of sync with each other, so it's important to frequently check for updates and **pull** changes from the remote repository before you start working on your local repository.

6 Making Your First Git Repo

In general, there are two ways to make a Git repo. You can start with a local repo and **push** it to GitHub, or you can start with a GitHub repo and **clone** it to your local machine. If you don't have any existing code, it's marginally easier to start with GitHub, which is why we'll start with this workflow. But no problem if you have code on your local machine - you just need to follow slightly different steps to connect the two. You will have to make the exact same decisions about repository structure, regardless of the workflow you use, so it's worth reading through the GitHub First section as we'll go into the most detail here.

6.1 GitHub First

If you have the GitKraken client installed, and it is connected to your GitHub account, you can create a new repository on GitHub directly in the GitKraken client, and it takes care of cloning the remote repository directly to your local machine (e.g., the cloning section). This makes the set up a little bit easier, as you don't have to use your browser or clone the repository. Otherwise, it works through exactly the same steps as the browser-based workflow, which I'll walk through below in case you do not have GitKraken.

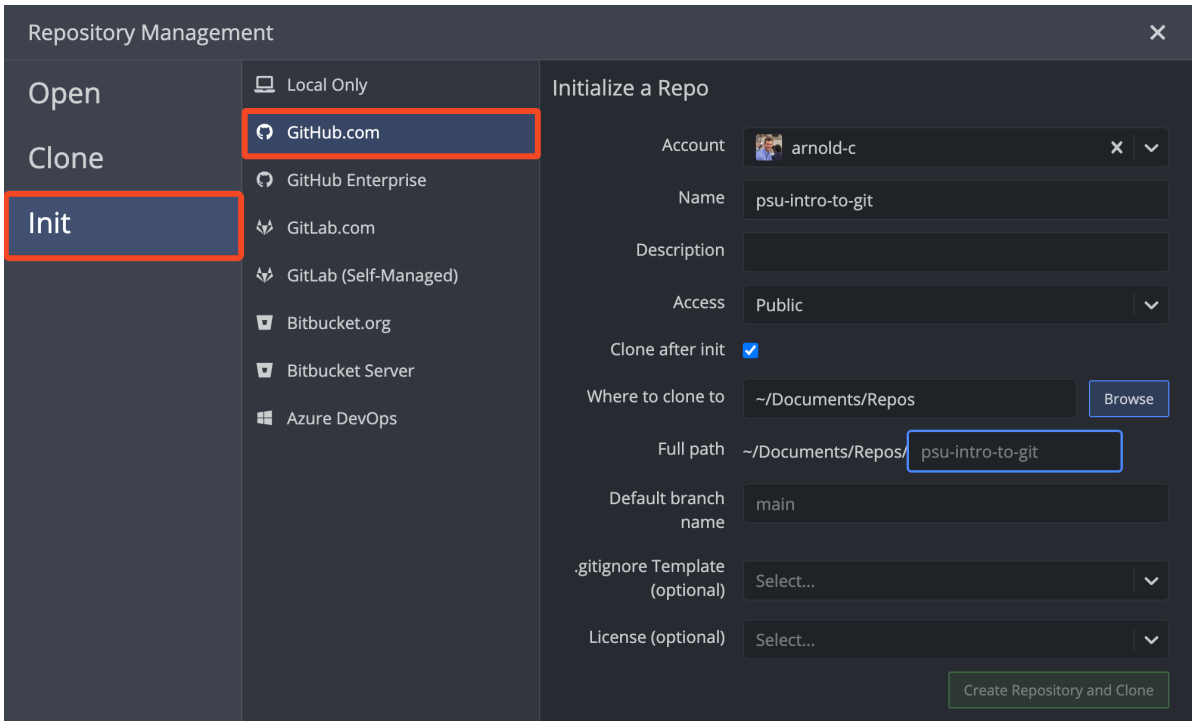
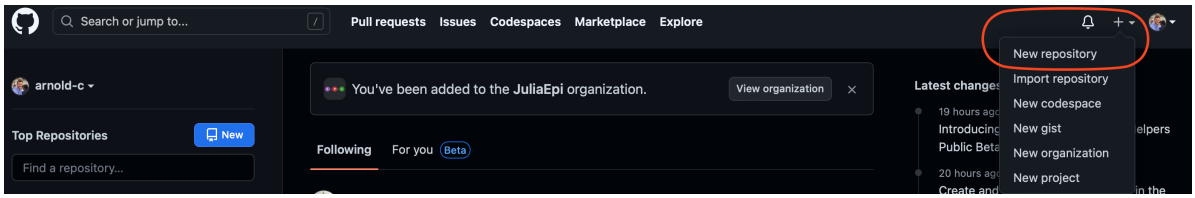


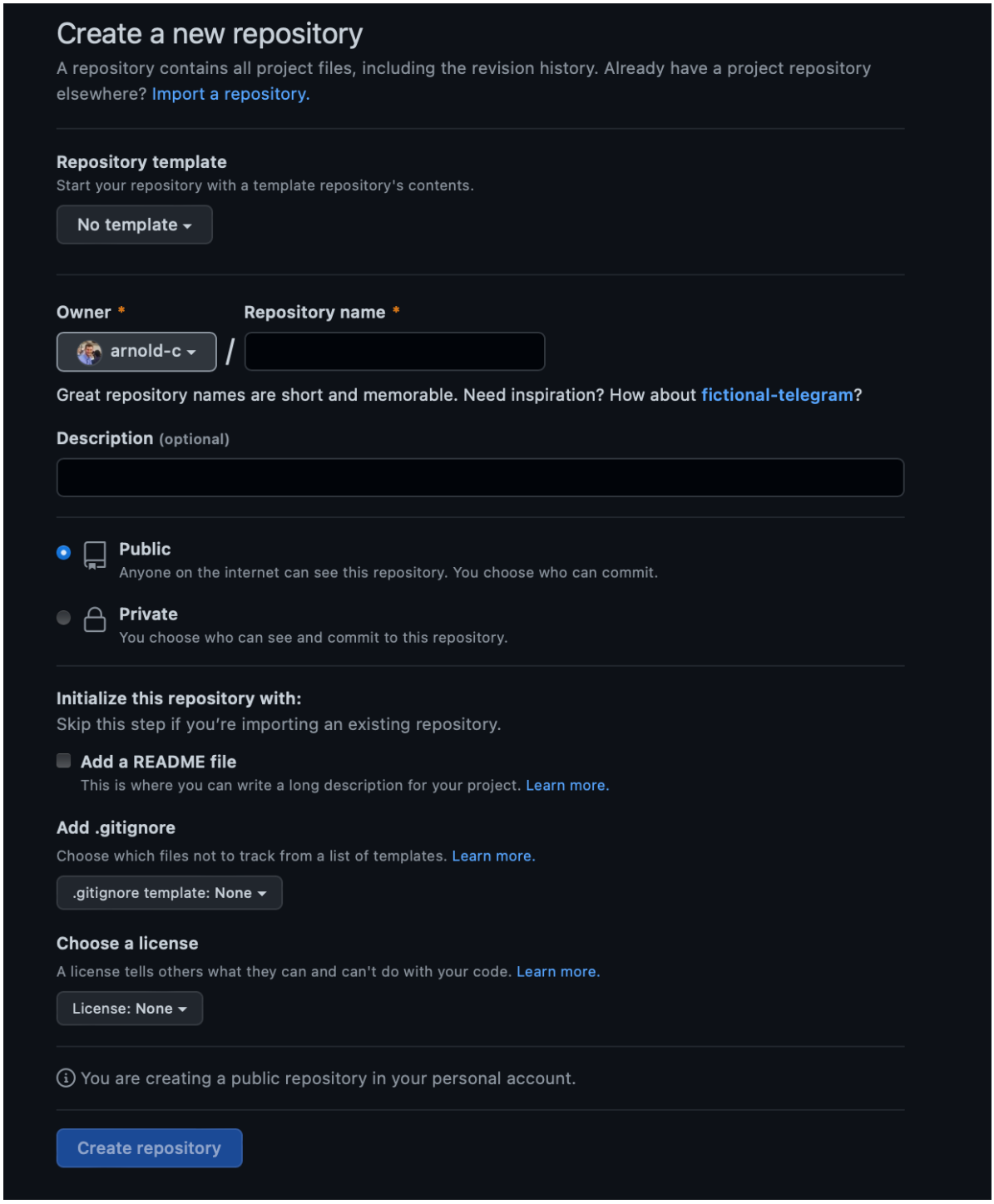
Figure 6.1: Pressing **cmd+I/ctrl+I** brings up the *"Init"* window

6.1.1 Creating the Repository

Log in to your GitHub account. From here, setting up a new repository is quick and simple - just click on the + button in the top right corner and then select "New repository".



From here, you're off to the races. You'll be presented with the following options, that we'll go through one-by-one.


Create a new repository
A repository contains all project files, including the revision history. Already have a project repository elsewhere? Import a repository.
Repository template
Start your repository with a template repository's contents.
No template
Owner *
Repository name *
arnold-c
/
Great repository names are short and memorable. Need inspiration? How about fictional-telegram?
Description (optional)
Public
Anyone on the internet can see this repository. You choose who can commit.
Private
You choose who can see and commit to this repository.
Initialize this repository with:
Skip this step if you're importing an existing repository.
Add a README file
This is where you can write a long description for your project. Learn more.
Add .gitignore
Choose which files not to track from a list of templates. Learn more.
.gitignore template: None
Choose a license
A license tells others what they can and can't do with your code. Learn more.
License: None
You are creating a public repository in your personal account.
Create repository

6.1.2 Repository Name

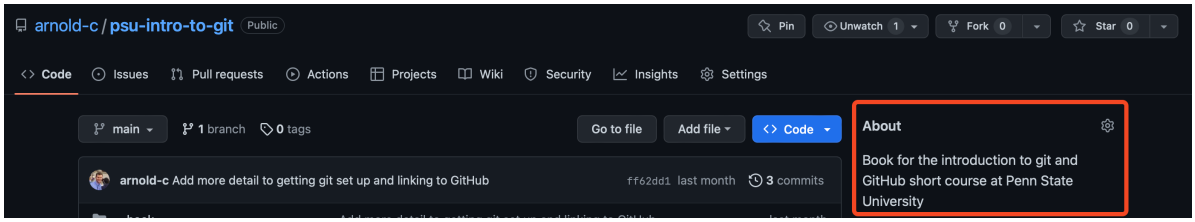
When choosing a repository name, you'll want something short, memorable, and descriptive of the project that you're working on. Ideally, you should follow Jenny Bryan's excellent guidelines when it comes to file and repository naming (after all, a repository is just a project folder), but as an overview:

- Use `-` to separate words
- Use `_` to separate metadata (e.g. dates from script description)
- Try to be consistent in how you use case e.g., `CamelCase` vs `camelCase` vs `snake_case` vs `everythingmashedtogether`

For example, the repository for this workshop book is **psu-into-to-git**. I'd highly recommend skimming these slides to learn more and save you some hassle in your digital life.

6.1.3 Description

The description is an optional part of the setup, but is worth completing. In a few short words, you should describe what you're trying to do with the project. For example, the description for this project is:



6.1.4 Public vs Private

Obviously this is a personal decision, but one of the benefits of learning and using Git is that it facilitates collaboration. While not all code can be open-sourced for legal, ethical, or other reasons, if you can, I would encourage you to try and publish the code as a *"Public"* repository. Not only is it in keeping with the principles of open and collaborative science, particularly when it comes to peer-review, but you also might get helpful feedback on your work from interested parties. If you've done something great that you want to share with the world, say, you've developed an awesome package or method of analyzing your data, it's cool if people can build off your work. But from a personal perspective, members of the community can, and often will, help you improve your code and move it forwards, giving it robustness and allowing you to do new things that you hadn't thought of.

You can always change the visibility of the repo later by going to the *"Danger Zone > Visibility"* option at the bottom of the *"Settings > General"* page.

6.1.5 README

The ***README.md*** acts as the landing page to your repository. You don't need it, but each repository should have one. The point of the README is to tell the reader what the repository is all about. As hinted by the file extension, the ***README.md*** is a markdown file. Markdown is very simple to use - you just type and let your syntax take care of the formatting. See this document from GitHub for an overview of how to use markdown.

At a minimum, start with these few items:

- Repository title
- About this project
  - Give a short (paragraph) overview of the project and what you hope to achieve with the work
- Repository structure
  - Tell the reader about the layout of the repository
    - What are your folder names, and what is contained in each folder
    - What do the key files do e.g. I have a file in one of my projects called ***student_missing-data-analysis.Rmd*** with the description "notebook that explore the missingness present in the data as a whole, but particularly among students with PSU samples. It examines the effects of imputation on the GLM ORs"
- Built with
  - What are the key packages that you used in the project?
    - I often use the `{targets}` package for automating R analysis pipelines and `{renv}` package for R package management
- Getting started
  - How to download the repository and get set up to run the analysis
  - Include the cloning commands
  - Tell the reader what packages to install, and how (some packages you use may not be standard and you may need to use special instructions e.g., `{JAGS}` often requires installation from SourceForge)
- Usage
  - Tell the reader how to re-run your analysis
  - Hopefully this will be fairly simple if you clearly describe your repository structure above

  - Because I often use `{targets}` when I'm working in R, I like to add a quick description about how to use `{targets}`, specifically that it is based around the functional programming concept, so it is a little unfamiliar to people used to writing and using scripts

- License
  - This can be a link to your ***LICENSE.txt*** file
  - It is particularly important if your code is public-facing
- Contact
- Acknowledgements

6.1.6 .gitignore

The ***.gitignore*** file is a special "dot" file that stays in your project root and tells Git to not track a file, or a group of files if you specify a folder or use the `*` wildcard. For example, we often do not want to html files as they are typically the outputs of our analysis e.g., rendered notebooks that we want to look at and share with collaborators. To exclude all html files, we simply add `*.html` to the ***.gitignore***, and html files will stop being tracked. To exclude a folder, we would add `my-folder/` to the ***.gitignore***.

If you have multiple files with the same name, but in different folders within the project, e.g. ***folder01/eda.Rmd*** and ***folder02/eda.Rmd***, you may only want to ignore one of them. In this situation, you should specify the path i.e., `folder02/eda.Rmd`. If you just add `eda.Rmd` to ***.gitignore***, both files will be ignored.

As you can see in the setup image, GitHub provides optional templates for the ***.gitignore***. It's worth taking advantage of this and using the one for your language of choice e.g., R, python, Julia etc. It will provide you with a good starting point that you can customize as necessary.

> ⚠ Warning
>
> ***.gitignore*** will not remove files from the Git history.

To do this, you would have to very carefully **cherry pick** and change past **commits**, particularly if you have already **pushed** your local changes to GitHub, and that is far beyond the scope of this workshop. So it's better to preemptively exclude a file or file type from Git tracking if you think you might not want it in the Git history later on e.g. put all sensitive data and outputs into folders that are ignored by Git.

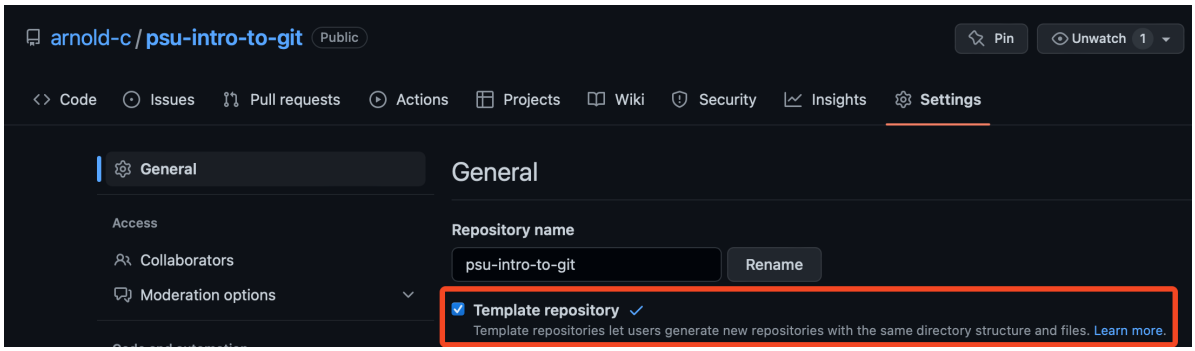
6.1.7 License

Most people place their license in a ***/LICENSE.txt*** file and a state what license type you want to use e.g., MIT. Then in the ***README.md***, you can just type in the location of the license e.g., ***/LICENSE.txt***, and it will provide a link that readers can click on for the full details. You can use this helpful website to decide what license is appropriate for you and your project.

6.1.8 Repository Template

Despite being first in the list, I've left this to the end for a reason. Firstly, when you get started with Git and GitHub, you won't have anything set up to use as a template. Secondly, it's important to understand how to use GitHub before you try and automate away the set up tedium.

Now we've got that out of the way, after you've created a repository with a structure you like (e.g., it has all of the components stated above), you can turn that into a template you can use for your next repository. To do so is very easy. Simply go to *"Settings > General"* and click on the *"Template repository"* button under your repository name.



The next time you go to create a repository, your previous repo will show up in the templates drop-down for you to use and then edit. If you felt so inclined, you could even create a separate repository that is only for your template, so you don't have to go through and delete parts of the ***README.md***.

6.1.9 Cloning to Your Local Computer

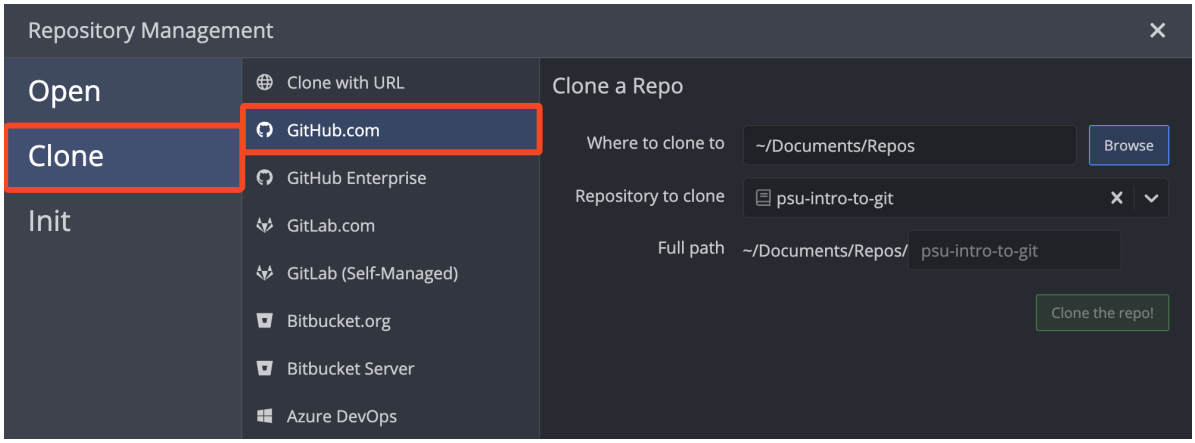
Now comes the fun part - getting your project set up on your computer where you can do your work. Thankfully, GitHub makes this incredibly easy to do, particularly if you have a Git client like GitKraken installed.

The first thing you need to do is create a local folder to **clone** your repository to. I like to have all of my repositories, work or otherwise, in one location on my computer, making it easy to find

and switch between them. For example, I keep my repositories in ***~/Documents/Repos/*** , therefore the repository for this project is at ***~/Documents/Repos/psu-intro-to-git***. Note that `~/` just means `/users/username/` on MacOS, which would translate to `/home/` on Linux, and `C:\Users\username\` on Windows.

It's not necessary to choose a directory name as the repository name on GitHub, but it's good practice and helps minimize confusion.

If you have a Git client, you can open it up, and assuming that it is connected to GitHub, there should be a "Clone button" you can click on. If you use GitKraken, pressing **cmd+N/ctrl+N** should bring up the clone window that looks like this below.

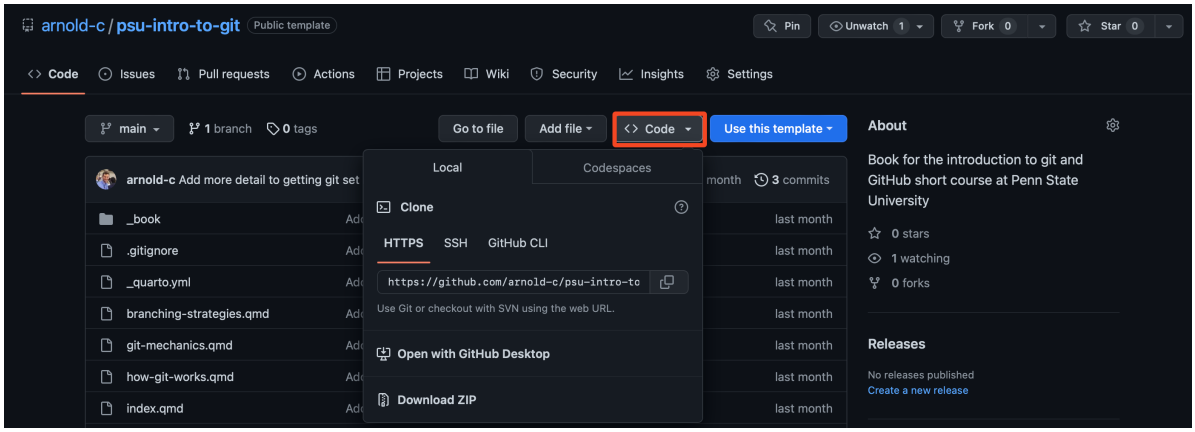


If you have connected GitKraken to GitHub, you can search through your GitHub repositories, otherwise you can paste in the HTTPS or SSH urls in the *"Clone with URL"* section.

To find your HTTPS/SSH urls, navigate to the repository, and click on the $<>$ *Code* button, before selecting either HTTPS or SSH, depending on your initial GitHub setup. These urls have a consistent pattern, so you don't always have to go to GitHub to find them.

For HTTPS, they are `https://github.com/GITHUB_USERNAME/REPOSITORY_NAME.git`.

For SSH, they are `git@github.com:GITHUB_USERNAME/REPOSITORY_NAME.git`.



If you do not have a Git client, navigate to the directory where you would like to keep your repositories; for me, this is *~/Documents/Repos/*. Then open up the terminal/Git Bash command line in this location. From here you have two options.

1. Create a folder for your repository (ideally use the same or a very similar name to the one on GitHub) and use the terminal command `Git clone git@github.com:GITHUB_USERNAME/REPOSITORY_` `.` (if you are using SSH) from within this folder to clone the contents of the GitHub repository into this folder.
2. Enter the command `Git clone git@github.com:GITHUB_USERNAME/REPOSITORY_NAME.git` (if you are using SSH) to create and clone the repository into *~/Documents/Repos/REPOSITORY_N* (note the lack of the `.` at the end of the command)

That's it, you're ready to start using Git!

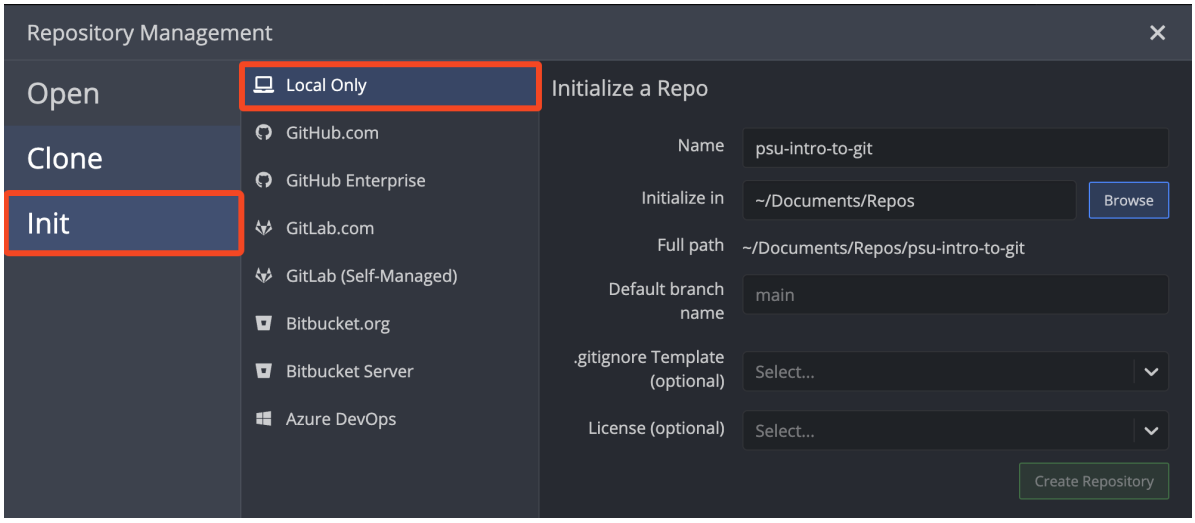
6.2 Local First

Sometimes you already have code on your computer that you want to turn into a Git project. As mentioned, this is a pretty easy problem to solve. You will still need to go through the same repository structure steps and decisions as above, so I recommend going back if you've just skipped ahead to here, but assuming you've already covered that material, the first thing you need to do is turn your local repository into a Git repository.

6.2.1 Git Client

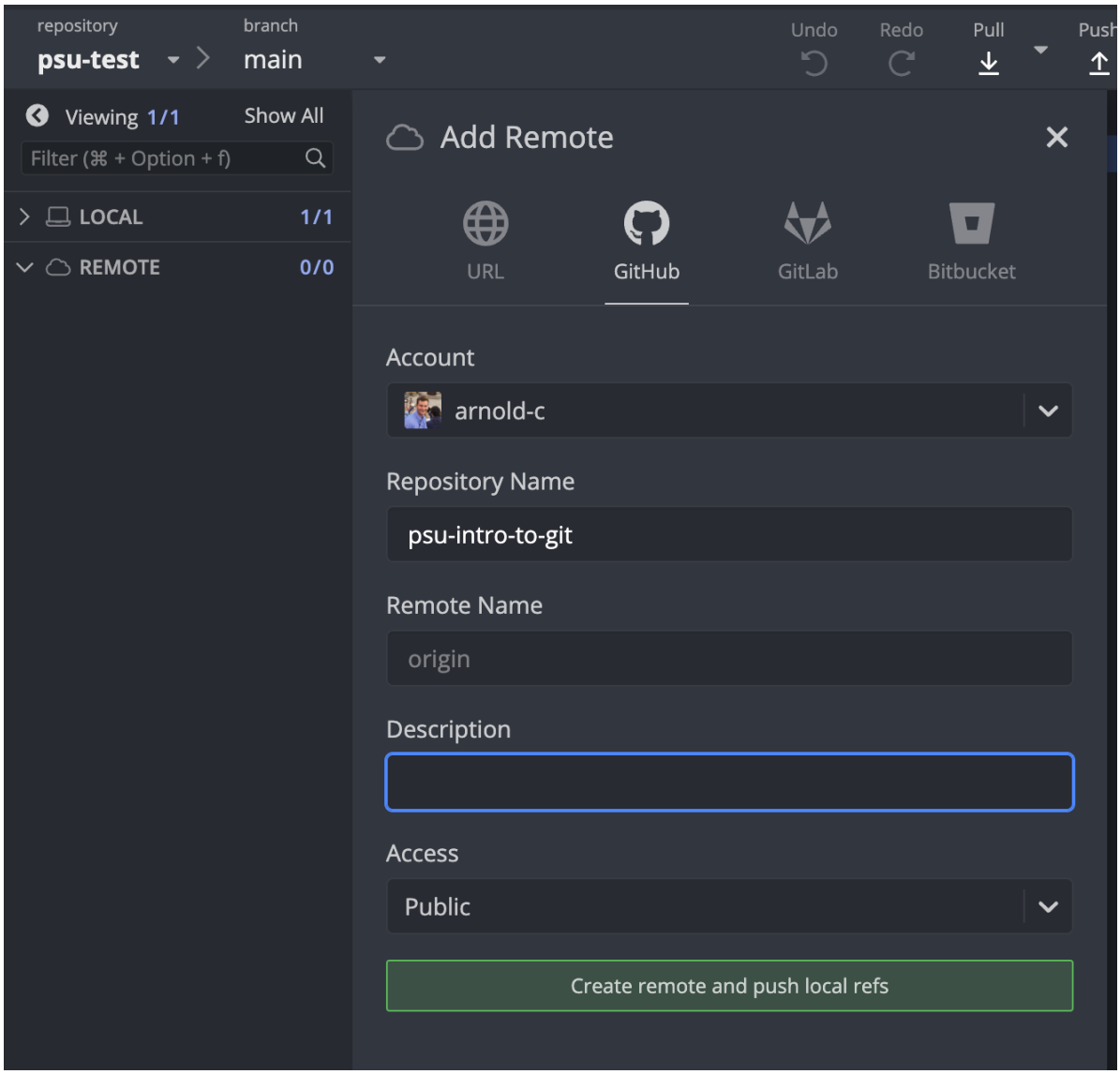
If you have a Git client installed, such as GitKraken, it's not too hard to turn an existing directory into a Git repository, before `pushing` to GitHub. First, open GitKraken and enter **cmd+I/ctrl+I** to open up the *"Init"* window. From here, navigate to *"Local Only"*, which tells GitKraken that you want to turn a directory local to your computer into a Git repository. Ensure that your "*Initalize in*" Path is in the correct parent directory (for me, I keep my

repositories in ***~/Documents/Repos***). Now just enter the name of the directory you would like to turn into the repository and click *"Create Repository"*.



This will not overwrite your file, just add the necessary Git components and add the ***.gitignore***, ***LICENSE.md*** (if selected), and a blank ***README.md*** (if you haven't already created one) as your first **commit**. All other files will be `added` (i.e., `git add .`), ready to be **staged** and **committed**, though we'll explain what this means in more detail shortly.

From here, you need to connect your local repository to GitHub. In GitKraken, open up the repository (if it didn't open automatically after creating), and click on the sidebar button labelled *"REMOTE"*.



From here:

- Select GitHub
- Choose your account (assuming you have linked GitKraken with GitHub)
- Choose the repository name you'd like for GitHub (I'd recommend the default, which is your local repository's name)
- Add a quick description
- Select the level of visibility you would like

> **Note**
>
> I would leave *"Remote Name"* as **origin** as lots of standard commands refer to **origin**, so it'll make your life easier if you have to troubleshoot things later on in your Git adventures

And that's it.

6.2.2 No Git Client

Firstly, go install a Git client if you are learning Git - you'll save yourself a lot of time and heartache when it comes to the day-to-day of using Git. But if you're insistent that you need to use the command line at this stage, or just are interested in understanding what commands your GUI is performing, carry on reading.

From here, your code will vary slightly depending on whether you set up Git using SSH or PAT token, but the structure is generally the same. If you already have code, you can ignore `echo ...` in step 2 as you have files Git can track (though you should add a ***README.md***!), and replace ***README.md*** with the file(s) you wish to track.

1. Open a terminal/Git Bash command line in your directory
    - All the following commands should be entered in the terminal from within the directory
2. You will need to turn your local directory into a git-tracked repository
    - Git will not track a completely empty directory, so you need to first add a file
    - `echo # PROJECT_NAME > README.md`
    - `git init`
3. `add` and **commit** the ***README.md*** file to the Git history so you have something to **push** to GitHub
    - `git add README.md`
        - Use `git add .` to track all the files in a repository with changes e.g. if you have existing code in a folder that you are converting into a Git repository
    - `git commit -m "Initialize project with README"`
4. Rename your default branch to **main**
    - `git branch -M main`
5. Connect your local repository to your GitHub repository
    - For PAT token setup
        - `git remote add origin https://github.com/arnold-c/psu-intro-to-git.git`

- For SSH setup
  - `git remote add origin git@github.com:arnold-c/psu-intro-to-git.git`

6. **push** your **commit** to GitHub
   - `git push -u origin main`
7. Navigate to GitHub, and edit the visibility of the repository if desired
   - By default, GitHub will publish the repository as public access

If you already have a Git repository on your local computer, but for some reason haven't connected it to a remote (e.g, GitHub) you can do so by following commands 4-7 inclusively.

7 Git Mechanics

7.1 An Overview of the Main Commands & Terms

There are many commands that you could learn in Git, but these are the basics, and will be sufficient for pretty much everything you'll need to do at the moment. I've added a few extras that you will likely come across, so it's worth having at least a rudimentary understanding of what they mean, and where they might be useful. As you get more advanced, you'll want to explore them in a little more detail.

7.1.1 Core Commands

- **add**
- **stage**
- **commit**
- **diff**
- **amend**
- **fetch**
- **pull**
- **push**
- **branch**
- **checkout**
- **merge**
- **pull request**

7.1.2 Useful to Know About

- **revert**
- **reset**
- **rebase**
- **rebase -i**
- **HEAD**
- **squash**
- **cherry pick**
- **reflog**

7.2 Term Descriptions

7.2.1 Core

- **add**: this takes all of the changes you have made to a file/set of files, and **stages** them, so they are ready to be committed
  - This is essential, as it allows you to work as you normally would, but save your changes in small chunks for more descriptive **commits** that are easier to understand and review.
  - `git add .` stages all files that have been modified, but you can specify specific files by explicitly naming them.
- **stage**: *this is the process of preparing a change/set of changes to be **committed**.
  - Think of it as putting a piece of code in a folder, ready to be saved.
  - You can **stage** as many changes as you like, including lines of code (AKA **hunks**) rather than whole files, and then **commit** them all at once.
  - If you have made changes to multiple files without writing any **commits**, you might not want to include all these changes in a single **commit** as the changes might be unrelated, or just too big to be useful so would be better served by splitting up into multiple distinct **commits**. **Staging** allows you to only prepare a subset of your changes for **committing**.
- **commit**: this standings for *committing* a change to your file in Git.
  - Think of it as saving a document, but instead of saving the whole document as-is, Git saves just the changes since the last version. This makes it very efficient, especially when it comes to backing up your work.

> **!** Important
>
> - **commit** often. By making and saving small changes, your code versions becomes more readable in case you need to go back and find out exactly what and where it went wrong.
> - Always write helpful messages - keep them succinct, but make sure they describe what the change you made was.

- **diff**: this command shows you what has changed in a file since the last **commit**
  - This is your "tracked changes" view!
- **amend**: this command add your changes to the most recent **commit**, rather than creating a new one.

  - This is useful when you forget to include something in a **commit**, i.e., it is a small change that belongs in the most recent **commit** and is not a substantial piece of work, even if the two are related
  - You never want to try to **amend** if your most recent **commit** has been **pushed** to the **remote**. You end up in a situation where collaborators might have already **pulled** your work so they are now out-of-sync with your rewritten git history, therefore git will not allow you to **push** these changes.
    * In this situation, just create a new **commit**!

> **Note**
>
> Technically, you actually **amend** the **commit** at the **HEAD**. Given that the **HEAD** is the most recent **commit** by default, this is something you shouldn't need to worry about for now. However, this distinction is important when it comes to **interactive rebases**. You can see this document for more information about **interactive rebases**, which highlights visually where the **HEAD** is in relation to the **commit** you are **amending**.

- **fetch**: checks the status of your **remote** and compares it to the version on your local machine, telling you if you are out of date i.e., need to **pull**
- **pull**: this command copies the version of the code from your remote to your local machine.
  - Use this when you want to get the most up-to-date version of your code to work on (assuming your local version isn't the most up-to-date)
- **push**: the opposite of **pull**. If your local version is the most up-to-date version, **push** your version to the remote.
  - You should try to do this a few times a day, but certainly less frequently than you **commit** to allow yourself some time to correct any mistakes before they are cemented into the git history
- **branch**: a branch is a specific version of your code that has its own git history, separate from the code and history of other branches.
  - This is useful for working on different features at the same time, as you can keep them separate until you are ready to merge them into the main code base.
  - See this section of the introduction for an overview, and the branching section for more details about how you can use branches to your advantage.
- **checkout**: changes the branch that you are working on
- **merge**: merges code changes from one branch into another i.e., keeps the git history separate for each branch, but at the merge point reconciles the differences
  - Most of the time this will work without issues, but occasionally if the two branches have made changes to the same line of code, you may get a merge conflict where

you need to tell git which version of the code it should keep in the final merged state.

- **pull request**: this is not a feature or command of git itself, but of GitHub (and other remote repositories). It is effectively a merge that takes place online to the **remote**, rather than to your local version
  - This is useful as it allows for mechanisms like code checks before changes are merged into a branch, helping to minimize merge conflicts that can happen when multiple people change the same file sections during the same period of time between **pushes** to the **remote**.

7.2.2 Useful Extras

- **revert**: creates a new **commit** that undoes the changes made during a specific **commit**
  - This is a useful and safe way of rolling back work as it does not delete any git history.
  - More applicable for public repositories that **reset**, as multiple collaborators rely on a shared git history, therefore it is critical this does not change unexpectedly.
- **reset**: this command set the current branch **HEAD** to whichever **commit** you are choosing to **reset** to i.e., moves the working state of the branch back
  - You do not need to specify a particular **commit** - this will just **reset** to the previous **commit**
  - There are 3 main types of **reset**:
    - `reset --soft`: Will not reset any files that have been **staged** but not **committed**. All changes in previous **commits** will be uncommitted, but will still exist and saved as **staged** changes, ready for you to **commit** them again.
    - `reset --mixed`: Will reset any files that have been **staged** but not **committed**. All changes in previous **commits** will be uncommitted, but will still exist. Unlike **reset** –**soft**, these changes are **unstaged** changes by default, so you will have to **add** (**stage**) them before you can **commit** them again. This is useful to **unstage** files you **staged** by accident, without deleting the code modifications you made.
    - `reset --hard`: Unstages all files and changes all files back to the version specified e.g., `git reset --hard` (without a commit specified) deletes all the **uncommitted** code changes since the last **commit**. If you specify a **commit** e.g., `git reset --hard 1a23b456` you delete every change after `commit 1a23b456`

> **Tip**
>
> I would recommend watching this video to get a better understanding of how **reset** works

- **rebase**: instead of **merge**, where the histories of each branch are retained, **rebase** moves all the **commits** from one branch onto the tip of the the other branch
  - When you are getting started, you rarely want to use **rebase** over **merge**
- **rebase -i**
  - There is a version of **rebase** called the interactive rebase that uses the command **rebase -i**
  - The interactive rebase allows you to completely rewrite the git history, including splitting up a **commit** into multiple smaller ones.
  - It is far beyond the scope of this workshop, and you should really think hard about whether it's necessary as it's easy to mess up your git history, but if you need to do this then you can find more information here
- **HEAD**: this is a description of which **commit** git points to
  - When you **checkout** a branch, the **HEAD** is set to the last **commit** in that branch, by default.
  - However, you can choose to move the **HEAD** back down the branch's history, i.e., **checkout** a specific **commit**.
    - This is called a **detached HEAD** state.
    - This does not delete the **commits** that have happened since, but it does mean any changes you now make will diverge from the state of the code present at **that commit** and can not be accessed as they are not created within a branch.
      - You should create a new **branch** if you intend to create new **commits**
- **squash**: this combines multiple **commits** into one
  - You will rarely want/need to do this, particularly when starting out, but sometimes it can clean up the git history when performing a pull request that targeted a distinct new feature, and after a code review, doesn't need all the changes to be recorded in separate **commits**.
  - Easiest to perform during a pull request on the GitHub interface.
- **cherry pick**: a command that allows selected **commits** to be appended to the current working **HEAD**.
  - This can be incredibly useful when you have local **commits** that you would like to move to a different branch, or if you would like to split up a **commit** into smaller ones.
- **reflog**: git records every command you make in the *reference log*, including **checking-out** branches, and the **git reflog** shows you this log
  - Normally, this is not necessary to reference, but it can be useful if you end up in a position where you've **reset** a branch, and realize you didn't mean to do that.

- You can reference the **reflog** to show which commands you want to roll back, and **checkout** that **detached HEAD** state, before carrying on as normal
  - `git checkout HEAD@{1}` would roll back one position (the end of the branch - the **attached HEAD** - sits at `HEAD@{0}`)

8 Basic Git Workflow

In this section, we'll put together the basic Git workflow, and show how all these many terms and commands actually fit together. We'll start completely from scratch, and work our way up to a full-fledged Git repository.

I find that the best way to learn anything is to actually do it on a real project, as it's hard to conceptualize what's going on when you're just reading about it, or even working through a toy example. And because we research infectious diseases in CIDD, we'll build up a repository that contains a notebook for an SIR model (Susceptible-Infected-Removed), and do it in both Python and R as that should allow most people to follow along with the code in a language they're familiar with. Look for the following drop-down sections to reveal the simulation code you need.

R Code

Python Code

Throughout this section, we'll be demonstrating the workflow using GitKraken, but the same principles apply to the command line. In fact, below each section that highlights a GitKraken feature, I've included a drop-down section that shows the equivalent command line commands. Just look for the following drop-down sections to reveal the commands.

Terminal Commands

8.1 Creating a New Repository

We'll create a new repository on GitHub, and then clone it to our local machine. We'll follow the steps outlined in the GitHub First section. So, in summary:

1. On your computer, decide where you want to store all your Git repositories, and create a folder for them
    - Mine has the path ***~/Documents/Repos/***, but you can put it wherever you want
2. Create a new repository on GitHub, and name it `sir-model`
    - Make sure you inialize it with a ***README.md***, Description, R/Python ***.gitignore*** template (depending on the language you'd like to follow along with), and an MIT license

3. Open GitKraken
4. Open the *"Clone"* window using **cmd+N/ctrl+N**
5. Click on the *"GitHub"* button and select the `sir-model` repository
    - Make sure the clone path is set to the folder you created in step 1
6. Click *"Clone"* and wait for the repository to be cloned to your computer

The repo should now be cloned to your computer, and you should be able to see it in the *"Open a repo"* section of GitKraken, which you can access using **cmd+O/ctrl+O**.

Terminal Commands

```
# Make the directory to store all your Git repos
# -p flag will create the directory and any parent directories that don't exist, but not ove
mkdir -p ~/Documents/Repos

# Change into the directory
cd ~/Documents/Repos

# Clone the repository
git@github.com:GITHUB_USERNAME/sir-model.git

# Change into the repository
cd sir-model
```

8.2 Giving Our Repository Some Structure

Now that we have a repository, we need to give it some structure. We'll start by fleshing out the ***README***, which will act as a guide for how we want to structure our repository. This is a good practice to get into as it will help you think carefully about how you want to organize your code, and will help you and others understand what's going on in your repository, as it's easy to skip this step and end up with a repository that's hard to decipher.

Let's first add the core components. Copy the following into the ***README.md*** file:

```
## Repository Title
### About This Project

### Repository Structure

### Built With
```

```
### Getting Started

### Usage

### License
This project is licensed under the MIT License - see the [LICENSE](LICENSE) file for details

### Contact

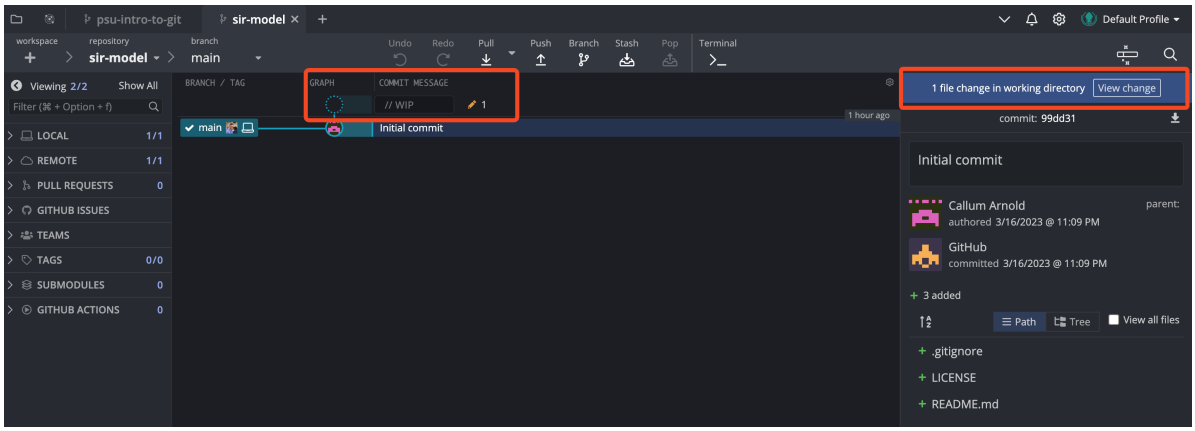
### Acknowledgements
```

> **Note**
>
> The `##` and `###` are used to denote different levels of headings in Markdown. I prefer to use a level-2 heading for the title of the repository, and level-3 headings for the different sections, as I find it makes the README easier to read; level-1 headings are quite large.

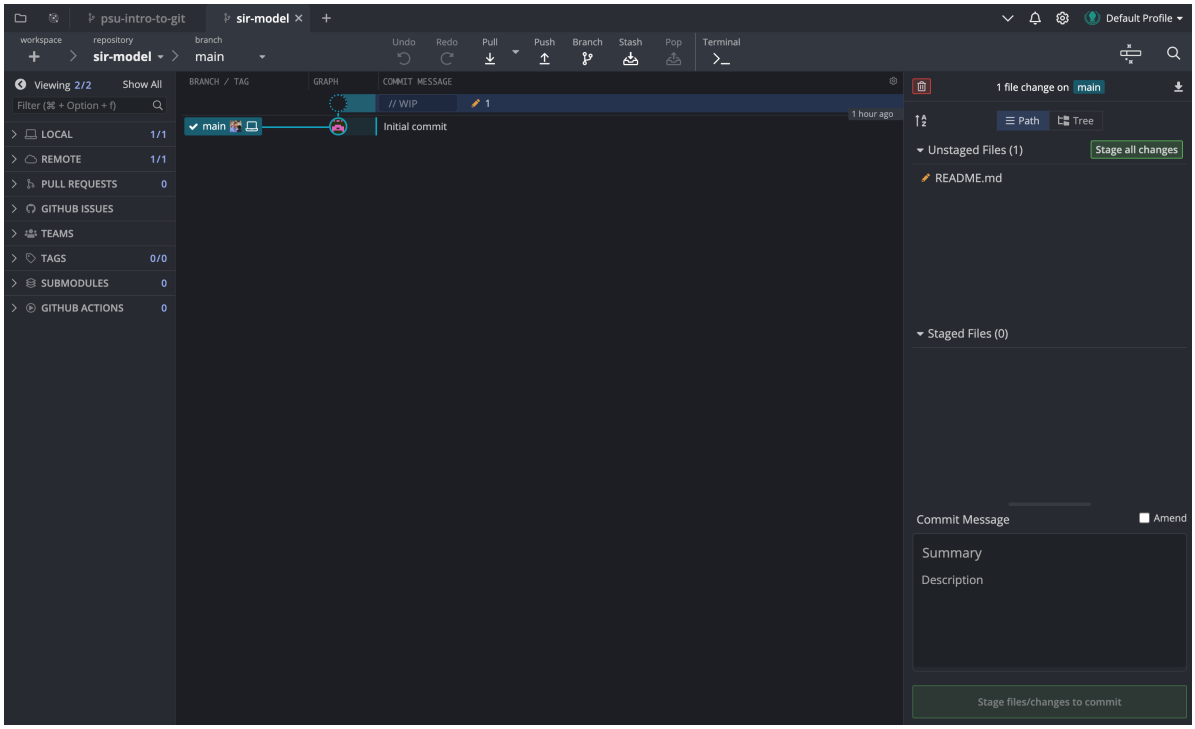
Now, rename the "Repository Title", and this is a good spot to create our first **commit** before we start to fill in a few of the sections.

8.2.1 Creating The First Commit

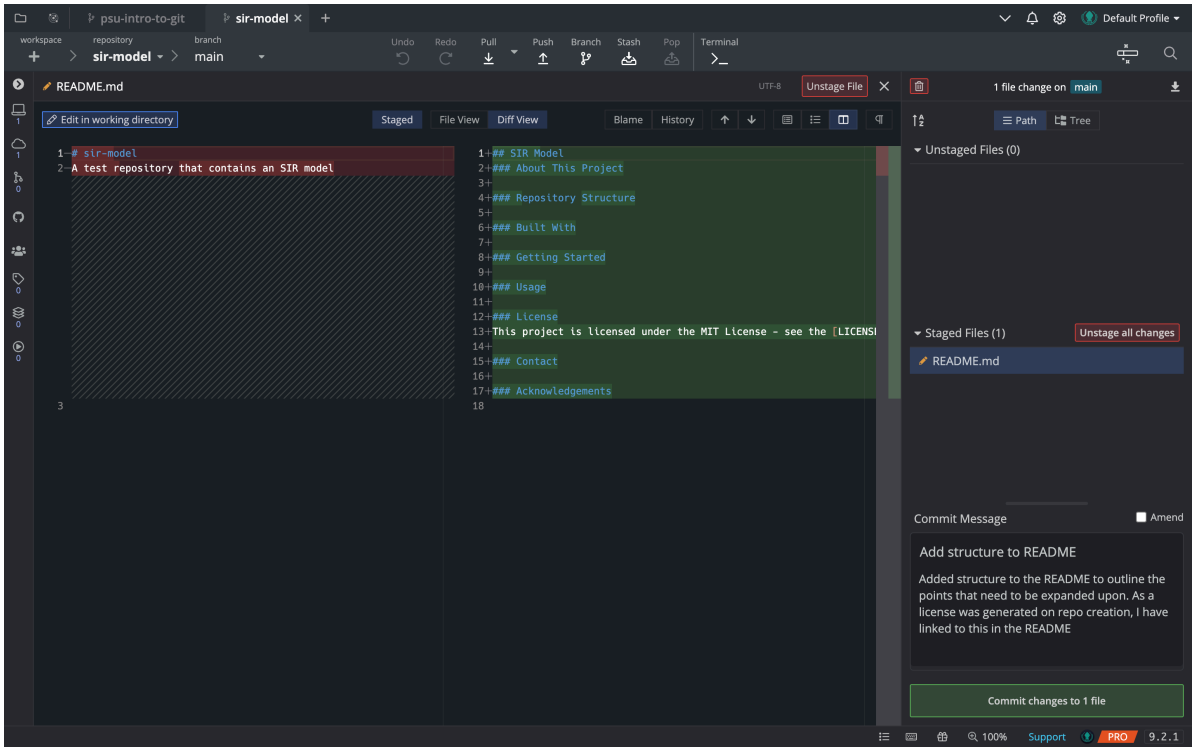
At this stage, we have a repository with a README, but we haven't actually saved any changes to the repository as far as Git is concerned. When we cloned our repository, the ***README.md*** only contained the text that GitHub added by default (the repo name and short description - you can check this on GitHub if you'd like). If you open up GitKraken, you'll see that it is showing that we have made changes to a file, indicated by the pencil icon in the commit history section, as well as the note above the commit message box that says *"1 file change in working directory"*.



Clicking on either of these sections will present you will the below window, which shows you the changes that have been made to the file. You will also notice that the changes are **unstaged**, which means they will not be included in the next commit.

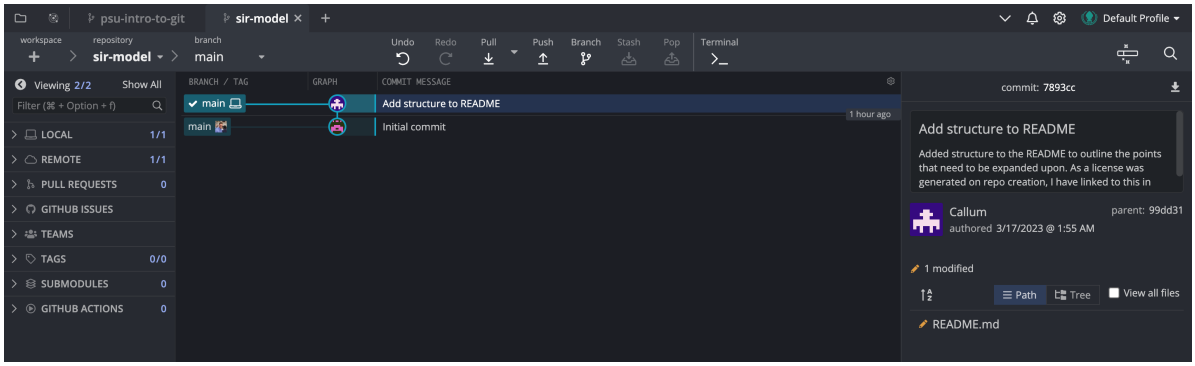


Clicking on the ***README.md*** file will open up the file **diff** in the GitKraken editor, and you can see that the changes are highlighted in green (for additions) and red (for deletions). Once we are happy with the changes, we can **stage** them and write the **commit** message. The **commit** message title should be short and descriptive, and the body should contain more details about the changes that were made, if necessary for clarity.



> **Note**
>
> Your **diff** view may look different than mine shown here e.g., it may default to lines being shown above each other, rather than side-by-side. You can change this if you would like, by clicking on the buttons in the top-right corner of the editor window, just below the button that says *"Unstage File"*.

Now that we've made our first **commit**, we will see that the pencil icon has disappeared and we have a similar view to what we started with when we cloned the repository. The key difference is now there are two `main` branches being shown in the Git history. The one with the computer that is highlighted is the **local** branch, which is the branch that is stored on our computer. The other branch is the **remote** branch, which is the branch that is stored on GitHub. Because we've just made a commit, the **local** branch is ahead of the **remote** branch by one commit as we haven't pushed our changes to GitHub yet. We'll do that later after we've added a few more things to the README.



So for now, let's continue to flesh out the README.

Terminal Commands

```
# Check what files have been changed
# -vv flag will show where the files have changed and display the diff
git status -vv

# Stage all changes
git add .

# Commit changes
git commit -m "Add structure to README" -m "Added structure to the README to outline the poi
```

8.2.2 About This Project

Our project is going to contain an SIR model with births and deaths, which is a simple model of infectious disease transmission. The SIR model is a compartmental model, which means that it divides the population into different compartments (Susceptible, Infected, and Removed), and tracks how people move between these compartments. Not everyone is familiar with this, so we'll probably want to include this information in the README. Similarly, we'll likely want to include the basic differential equations that govern the model, and the parameters that we'll use to run the model (as well as a short description of what they mean). As we're going to use some code from Ottar's Epidemics book (chapter 2), we'll also want to include a link to the book (and citation). To add this to a ***README.md*** file, we can use the following syntax:

````
```math
\begin{align}
\frac{dS}{dt} &= \mu (N - S) -\beta S \frac{I}{N} \\
\frac{dI}{dt} &= \beta S \frac{I}{N} - \gamma I - \mu I \\
\frac{dR}{dt} &= \gamma I - \mu R
````
````

````
\end{align}
```


```math
\begin{align}
\mu &= \frac{1}{50*52} \\
\beta &= 2 \\
\gamma &= \frac{1}{2} \\\\

N &= 1000 \\
S_0 &= 999.0 \\
I_0 &= 1.0 \\
R_0 &= 0.0
\end{align}
```
````

This will render as:

$$\frac{dS}{dt} = \mu(N - S) - \beta S \frac{I}{N} \tag{8.1}$$
$$\frac{dI}{dt} = \beta S \frac{I}{N} - \gamma I - \mu I \tag{8.2}$$
$$\frac{dR}{dt} = \gamma I - \mu R \tag{8.3}$$
$$\tag{8.4}$$
$$\mu = \frac{1}{50 * 52} \tag{8.5}$$
$$\beta = 2 \tag{8.6}$$
$$\gamma = \frac{1}{2} \tag{8.7}$$
$$\tag{8.8}$$
$$N = 1000 \tag{8.9}$$
$$S\_0 = 999.0 \tag{8.10}$$
$$I\_0 = 1.0 \tag{8.11}$$
$$R\_0 = 0.0 \tag{8.12}$$

Fill in these details, and then we'll move on to the next section we can complete.
````

8.2.3 Repository Structure

We can't fill in all of this section until we know exactly what the code is going to do, but we can at least give a rough outline.

To start, I like to use the following folders to help organize my code, but you are welcome to use whatever structure works best for you:

```
```
.
   data/
   figs/
   funs/
   out/
   src/
```

- `data/` contains ...
- `figs/` contains ...
- `funs/` contains ...
- `out/` contains ...
- `src/` contains ...
```

- ***data/*** will contain any raw data that we use in our analysis, and is not edited by hand
- ***figs/*** will contain any figures that we generate
- ***funs/*** will contain any functions that we write to help us with small, repeatable things e.g. functions to format tables in a notebook
- ***out/*** will contain any output from our analysis, e.g. intermediate datasets that have been cleaned and are ready for analysis, tables, etc
- ***src/*** will contain any code that we write to do our analysis, e.g. notebooks, scripts, etc
  - If you work with multiple languages, it might make sense to have subfolders for each language, e.g. ***src/python/*** and ***src/R/***

You will want to ensure that your descriptions of each of the folders includes enough detail that you (and anyone else reading your project) can understand what's going on, but not so much detail that it becomes hard to read. And remember, these folders will not appear in the Git history until they contain a file that Git can track.

It's quite nice to include an ASCII tree to visualize the structure of the repository, and there are a couple of ways to generate this. If you use VSCode, you can install the ASCII Tree Generator extension, which will allow you to right-click on a folder and select *"Generate Tree String"* to generate a tree for that folder. If you use R with RStudio, you can use the `fs::dir_tree()` function from the {fs} package. If you want to do it semi-manually, you can just use this website.

8.2.4 Contact and Acknowledgements

These are pretty self-explanatory, and easy to fill in now.

8.2.5 Making a Second Commit

Now we've added a bit more to the README, let's make another **commit**. The process is exactly the same as before, so nothing much to go through here.

8.3 Installing the Necessary Packages

To run the code, we'll need to install a few packages. In both R and Python I'm using packages to manage the environments to try and make the code more reproducible. In R, this is the `{renv}` package, and in Python, this is the `{poetry}` package.

If you use Python, you are likely already familiar with the `pip` package manager and the concept of virtual environments, and but R users may not be. This is too big of a topic to go into detail here, but the basic premise is that the package manager takes a snapshot of all the installed packages and their dependencies, along with their version numbers, and stores this information in a file called a *lockfile*. Doing this means that if someone **clones** your repository, and your repo has a *lockfile*, they can run just one command to install all of the packages that you used to run your code, and not worry about incompatible versions of packages being installed that would cause errors.

Here, I've provided the *lockfiles* for both R and Python, so you can just run the commands below to install the packages.

R Code

Install the `{renv}` package if you don't already have it installed:

```
install.packages("renv")
```

Copy the ***renv.lock*** file to the root of your repository. Then, run the following command in your R terminal:

```
renv::restore()
```

Python Code

Install the `{poetry}` package if you don't already have it installed. How you do this will depend on your operating system, but you can find instructions here. Make sure the poetry package is available in your PATH by running the following command in your terminal:

```
poetry --version
```

Copy the ***poetry.lock*** and ***pyproject.toml*** files to the root of your repository. Then, run the following command in your terminal:

```
poetry install
```

> **Note**
>
> This code was developed in a pyenv virtual environment, using Python 3.11.0.

8.4 Adding Simulation Code

Let's start writing our simulation. As mentioned, we'll adapt code from Ottar's book. We'll start by creating the ***src/*** folder, and then creating a new file called ***sir_model.R*** / ***sir_model.py*** in that folder. Copy the below code into that file.

R Code

```r
library(deSolve)
library(tidyverse)

theme_set(theme_minimal())

sirmod <- function(t, y, parms) {
  # Pull state variables from y vector
  S <- y[1]
  I <- y[2]
  R <- y[3]

  # Pull parameter values from parms vector
  beta <- parms["beta"]
  mu <- parms["mu"]
  gamma <- parms["gamma"]
  N <- parms["N"]

  # Define equations
  dS <- mu * (N  - S)  - beta * S * I / N
  dI <- beta * S * I / N - (mu + gamma) * I
  dR <- gamma * I - mu * R
```

```r
  res <- c(dS, dI, dR)

  # Return list of gradients
  list(res)
}


times <- seq(0, 26, by = 1/10)
parms <- c(mu = 0, N = 1, beta =  2, gamma = 1/2)
start <- c(S = 0.999, I = 0.001, R = 0)

out <- ode(y = start, times = times, func = sirmod, parms = parms)
out_df <- as_tibble(out) %>%
  pivot_longer(cols = -time, names_to = "state", values_to = "number") %>%
  mutate(
    time = as.numeric(time),
    number = as.numeric(number),
    state = factor(state, levels = c("S", "I", "R")),
    number = round(number, 6)
  )

ggplot(out_df, aes(x = time, y = number, color = state)) +
  geom_line(linewidth = 2) +
  labs(x = "Time", y = "Number", color = "State")
```

Python Code

```python
import numpy as np
import pandas as pd
from scipy.integrate import solve_ivp
from plotnine import *

def sirmod(t, y, beta, mu, gamma, N):
    # Unpack states
    S, I, R = y

    # Define equations
    dS = mu * (N - S) - beta * S * I / N
    dI = beta * S * I / N - (mu + gamma) * I
    dR = gamma * I - mu * R

    # Return list of gradients
```

```
    return dS, dI, dR

tmin = 0
tmax = 26
tstep = 1 / 10
times = np.arange(tmin, tmax, tstep)

beta = 2
mu = 0
gamma = 1 / 2
N = 1
parms = (beta, mu, gamma, N)

S0 = 0.999
I0 = 0.001
R0 = 0
start = (S0, I0, R0)

out = solve_ivp(
    sirmod, [tmin, tmax], np.array(start), args=parms, t_eval=times
)

out_df = (
    pd.DataFrame(out.y).transpose().rename(columns={0: 'S', 1: 'I', 2: 'R'})
)
out_df['time'] = out.t
out_df = out_df.melt(id_vars='time', value_vars=['S', 'I', 'R']).rename(
    columns={'variable': 'state', 'value': 'number'}
)

theme_set(theme_minimal())

(
    ggplot(out_df, aes(x='time', y='number', color='state'))
    + geom_line(size=2)
    + labs(x='Time', y='Number', color='State')
)
```

We've now made some substantial changes to the repository, so let's make a **commit**. The process is exactly the same as before.

8.5 Updating the README

8.5.1 Mistake in our Model Description

Now that we've added some code, let's go back and update the README. Looking at what we've written, we can see that the code is actually different from what we had in the README, so we'll need to update that. Examining the code, we can see that we've run the model on fractional populations, not whole numbers, so we'll need to update the description of the model to reflect that.

Go back and adjust the S_0 … values in the equations to represent fractions, and then update the description of the model to reflect that.

Because we've made some changes that are distinct from the other changes we've made, we'll again want to make a **commit** for these changes.

8.5.2 Expanding Upon the README

Now we have some code, we can expand upon sections of the README. The first thing we can do is to add some text to the "Getting Started" section. I've used the `{renv}` package to manage the R environment for this project, so I'll add some text to the README to reflect that.

R Code

```
I've used the `{renv}` package to manage the R environment for this project.
For more details on how to use `{renv}`, see [this article](https://rstudio.github.io/renv/ar

To get started, you will need to install `{renv}` as usual (i.e., `install.packages("renv")`
```

The following text can be added to the "Usage" section.

```
To run the SIR model, you can open the ***src/sir_model.R*** file and run the code as usual.
```

Python Code

```
I've used the `{poetry}` package to manage the R environment for this project.
For more details on how to use `{poetry}`, see [this page of the documentation](https://pytho

To get started, you will need to install `{poetry}` using system-dependent commands (see [the
```

The following text can be added to the "Usage" section.

```
To run the SIR model, you can open the ***src/sir_model.py*** file and run the code as usual
```

8.5.3 Amending the README Commit

We forgot to update the "Built With" section of the README, but we don't want to make a whole new **commit** just for that, as making tons of tiny small **commits** will make it harder to read and navigate the Git history if you need to reference past work. Thankfully, this change aligns with the previous **commit**, so we don't need to make a new **commit**. Instead, we can **amend** the previous **commit** to include the changes we've made.

We want to add the following text to the "Built With" section of the README.

R Code

```
- [R version 4.2.1 (2022-06-23)](https://cran.r-project.org/bin/macosx/)
- [`{renv}`](https://rstudio.github.io/renv/articles/renv.html) for package management
```

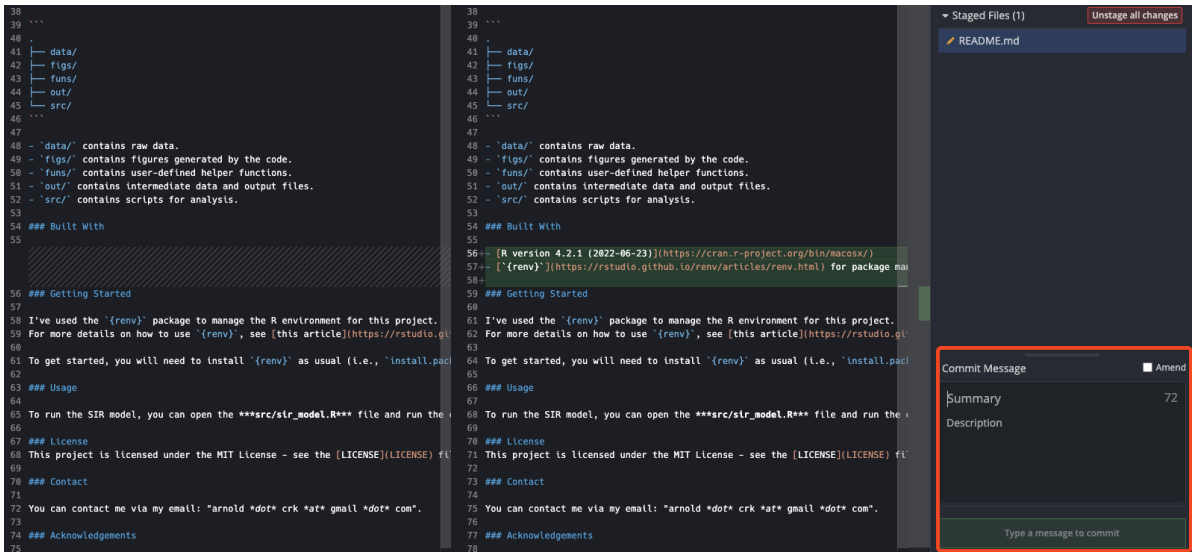
Python Code

```
- [Python version 3.11.0](https://www.python.org/downloads/)
- [`{poetry}`](https://python-poetry.org/docs/) for package management
```

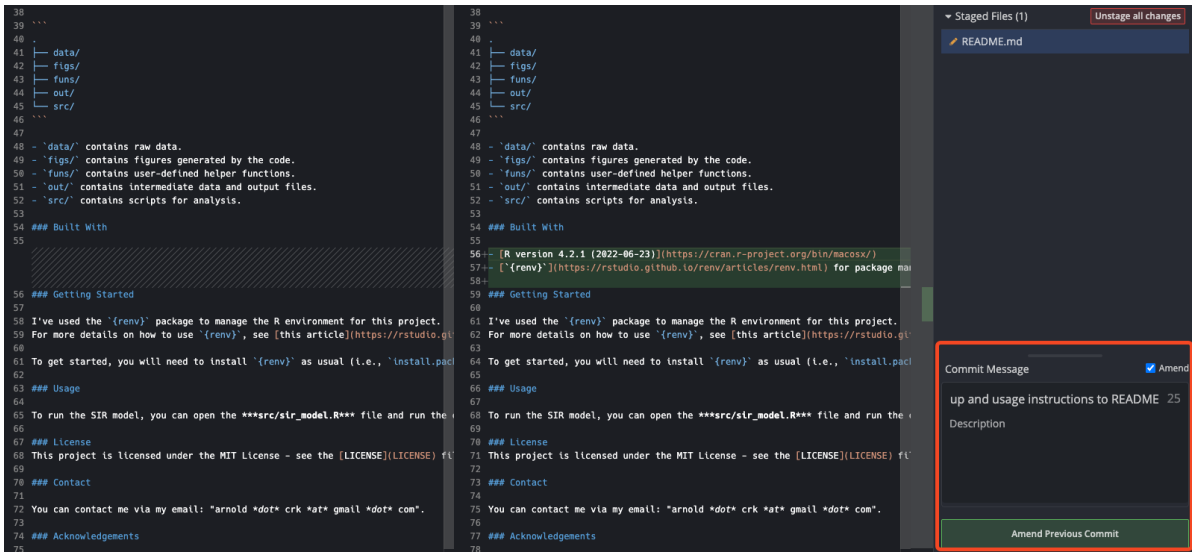
> ⚠ Warning
>
> We can only **amend staged** changes to the most recent **commit**, and only if we haven't **pushed** it to GitHub yet. If we've already **pushed** the **commit**, we'll need to make a new **commit**. If you do not make a new **commit** and try and just **amend** it, you'll get an error because Git wants to avoid rewriting history that might have been accessed by others, resulting in conflicting Git histories on different machines.

Once we've made and saved the changes to the ***README.md***, we can **stage** them and then **amend** the previous **commit**.

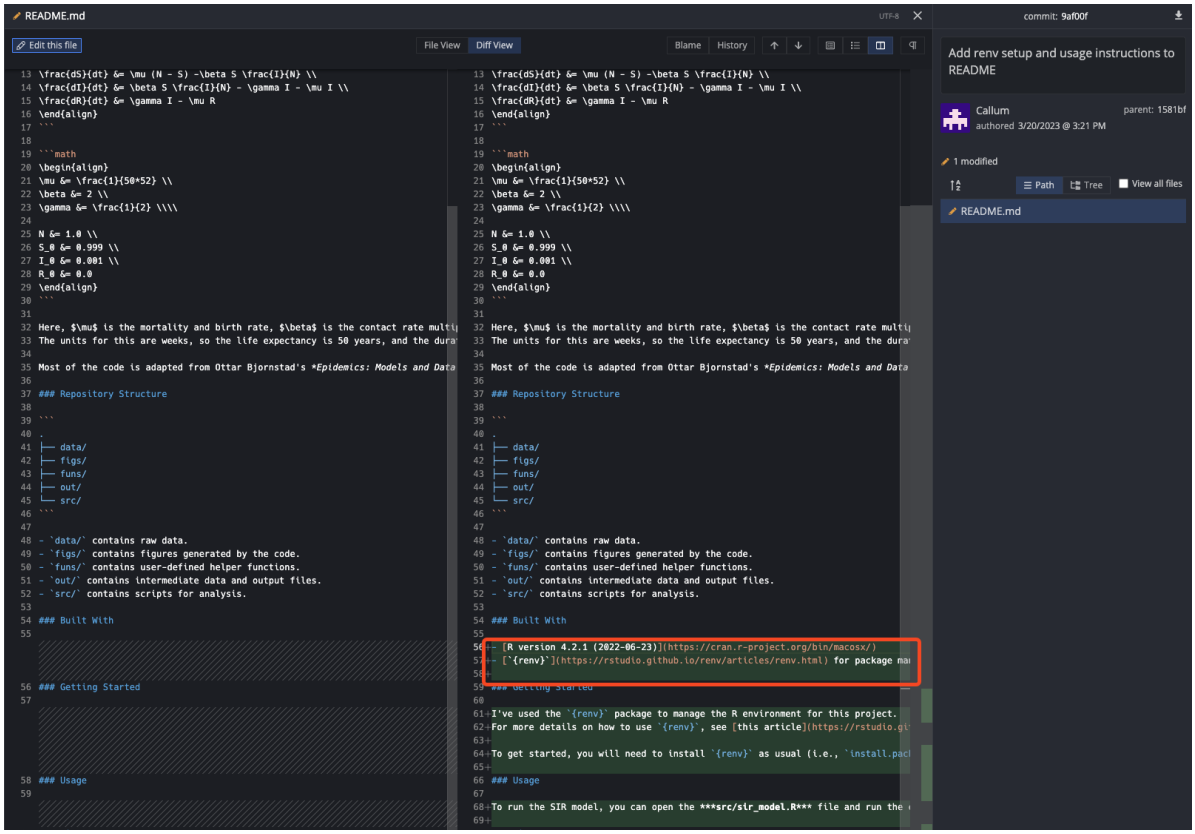
As per usual, when we save and **stage** the changes, we'll see the **diff** in the GitKraken interface, as below.



You can see that just above the "Commit Message" input section, there is a button for "Amend". Clicking this will **amend** the previous **commit** to include the changes we've made. As such, it will set the **commit** message to the same as the previous **commit**, so there is nothing else to do. During the **amend** process, GitKraken will also change the text of the "Commit" button from *"Commit changes to X file(s)"* to *"Amend Previous Commit"*, as below.



Once we've **amended** the **commit**, can examine the Git history and see that there is only one **commit** that refers to the changes we've made to the ***README.md*** file. Equally, if we look at the **diff** of this **commit**, we can see that our **amended** changes are included.



Terminal Commands

```
# Check and stage changes
git status -vv
git add .

# Amend changes
git commit --amend

# Examine the Git history and find the commit ID for the commit you want to diff
# --oneline: show each commit on a single line, rather than the default multi-line format whi
# --graph: show a graphical representation of the Git history, which is useful when you have
# --decorate: show the names of all references (branches, tags, etc.) that point to the commi
# --all: show the Git history for all branches
git log --oneline --graph --decorate --all

# Diff the commit (replace with your commit ID)
git diff 0de4674
```

> **Note**
>
> If you do not specify the **commit ID**, the **diff** of the most recent **commit** will be shown.

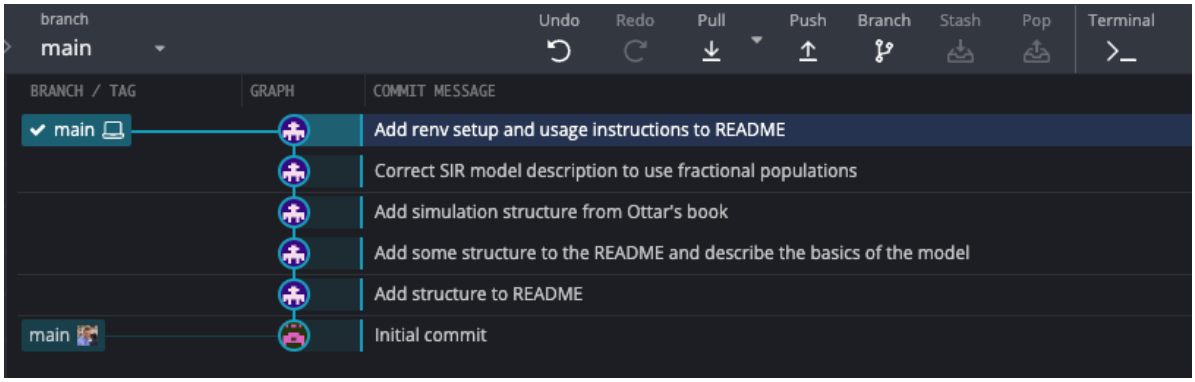
> **Warning**
>
> If you try to **diff** a **commit** that is particularly long i.e., has a lot of line changes, you will enter a special interactive mode that is based on Vim. There are a number of commands you'll need to know.
>
> - `d`: scroll the page down
> - `u`: scroll the page up
> - `j`: move the cursor down one line
> - `k`: move the cursor up one line
> - `q`: quit the interactive mode and return to the terminal
>
> To see a cheat sheet with more Vim commands, see here.

See here for some git aliases that can make your git log prettier.

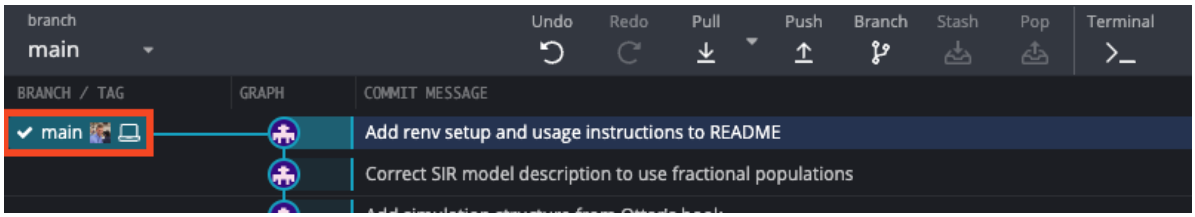
8.6 Our First Push

Now that we've made some substantial changes to the repository, we can **push** them to GitHub. This is pretty straightforward to do, and we can do it from the GitKraken interface.

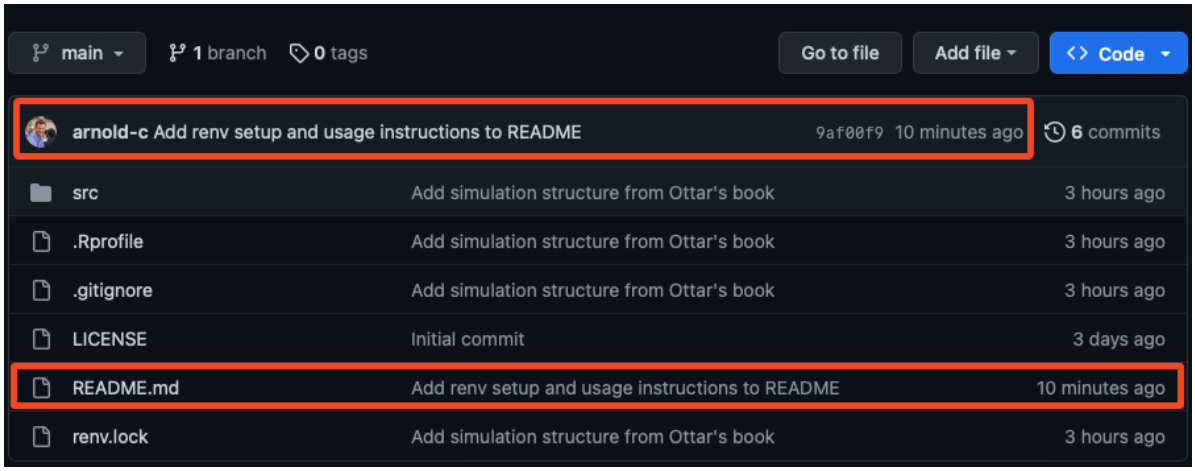
On the GitKraken client, there is a button in the top toolbar that is an up arrow, with the description "Push". When we hover over it, it shows the text "Push to origin/main".



Pushing the changes will bring the version of the code on GitHub up-to-date with the version of the code on our local machine. We can see that the **main** branch with out GitHub user photo is now in-line with the **main** branch of our local machine.



We can also confirm this is the case by navigating to our repository on GitHub and looking at the Git history.



> **Note**
>
> The Git history on GitHub will show the relative time that the **commit** was made, not **pushed** to GitHub.

Terminal Commands

```
# Push changes
git push
```

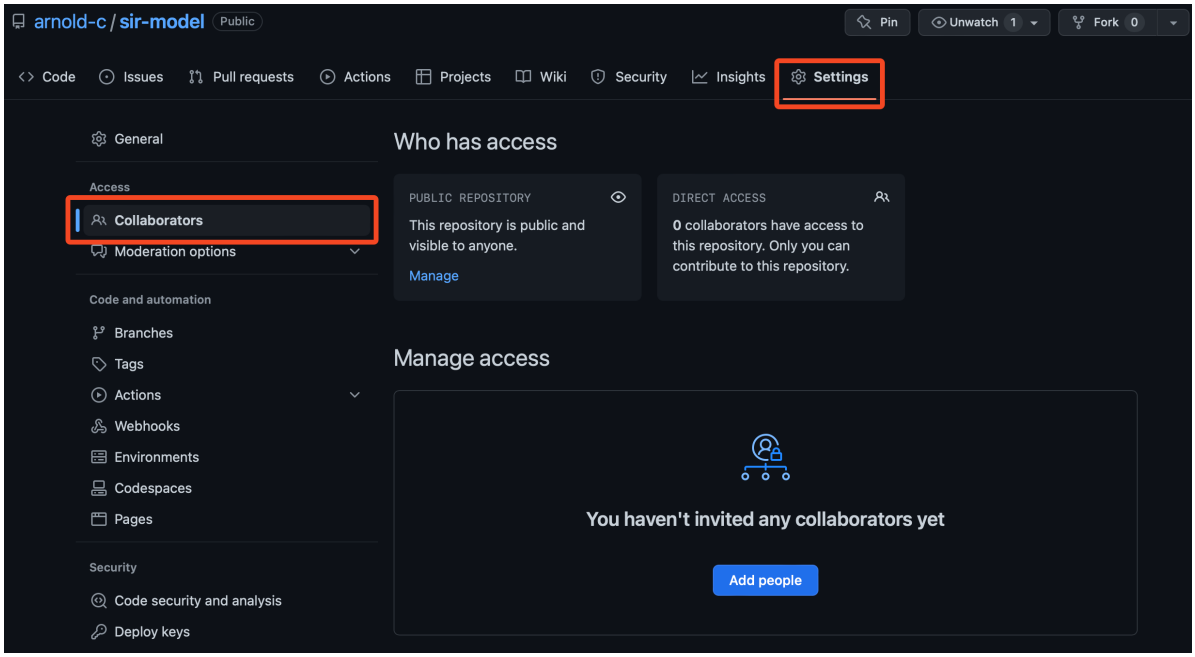
8.7 Collaborating on a Project

Up until now, we've just been working on our own, but what if we want to collaborate with others on a project? There are a couple of ways we can do this, but the easiest is to add collaborators to our repository on GitHub.

> **Note**
>
> Just for reference, you could set up a GitHub organization and then add collaborators to that, but that's a bit more complicated than what we're doing here.
> For open-source projects where you want to allow others to make suggestions for their contributions, without directly giving them access to the repository, you can use the "**fork** and **clone**" model, where people can **fork** the repository and then make a **pull request** to merge their changes into the original repository. This is far more complicated than what we're doing here, so we won't cover it, but you can learn more about it in Jenny Bryan's Happy Git with R book chapter, the Atlassian tutorial, and/or the official GitHub tutorial that demonstrates with a repository that you can practice on.

We'll practice where you can add me as a collaborator to your SIR repository.

First, go to *"Settings > Access > Collaborators"* on your repository on GitHub.



You can see that there are no collaborators at the moment, but click on the "Add people" button to add me as a collaborator. From here, you can type in my GitHub username, `arnold-c`, and then click on the *"Add arnold-c to this repository"* button.

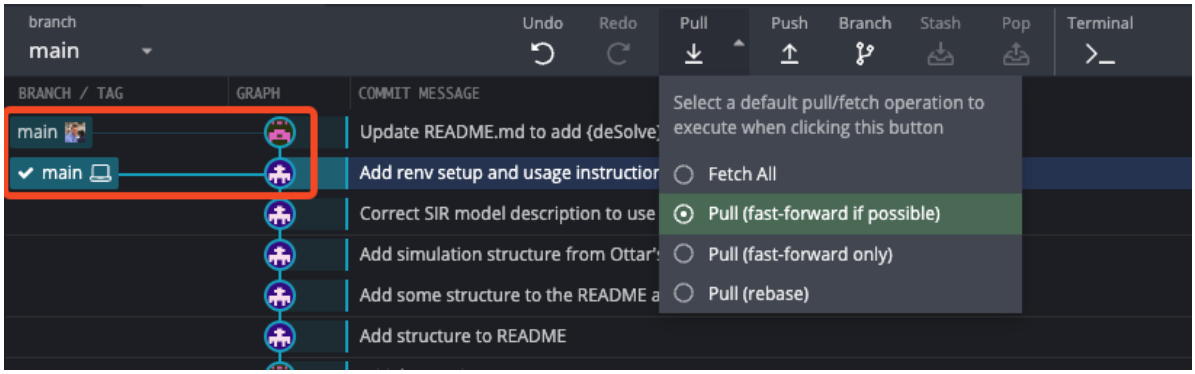
From here, I'll make the following change to the ***README.md*** file under the "Built With" section.

```
- [`{deSolve}`](https://cran.r-project.org/web/packages/deSolve/index.html) for solving diffe
```

What you will notice is that when I make this change, it will be reflected in the Git history on GitHub, but not in your local repository. However, this will discrepancy will not immediately be apparent, as our computer hasn't checked in with GitHub to see if there are any changes since we last **pushed**. To do this, we will use the **fetch** command.

> **Note**
>
> In this instance, I've just made the change myself directly in GitHub, which will have a similar effect to it being made by a collaborator.

To **fetch** the changes, we can use the GitKraken interface. Clicking on the *"Pull"* button will bring up a menu, where we can select the *"Fetch All"* option. Once the changes have been **fetched**, we can see that there is a new **commit** in the Git history, and our **remote** branch is now ahead of our **local** branch.



> **Note**
>
> In some cases, GitKraken will automatically **fetch** changes, but here's how you can do it manually (this is worth doing routinely if you're collaborating with others, just to be safe).

Pulling the changes will bring our **local** branch up-to-date with the **remote** branch. The default **pull** behavior is to *"**fast-forward** if possible"*, which means that the **local** branch will be moved to the **commit** that the **remote** branch is pointing to. It does this by performing a **merge**. There are some instances (that we are unlikely to run into for a while) where this can cause issues, which is why there are options to *"fast-forward only"* and *"rebase"*. If you want to learn more, you can read about it here and here. For the time-being, just stick with the default behavior and you'll be fine!

> **Tip**
>
> When working on a collaborative project, get into the habit of **fetching** routinely to see if there are any changes that you need to **pull**. If you leave it too long, you might end up with a lot of changes to **pull** and it might be difficult to figure out what's going on. Even worse, you might end up making changes to old versions of the code, which will cause conflicts when you try and **pull**/**push**/**merge** the changes.

Terminal Commands

```
# Fetch changes
git fetch

# Pull changes
git pull
```

9 Branching Strategies

OK, so we've covered quite a lot of ground here, and this should be enough to get you started with Git. But there's one more helpful feature of Git I'd like to talk about: **branching**.

As mentioned earlier, **branches** are a way to keep track of different versions of your code. They're also a way to keep your code organized and to keep your work separate from other people's work. For the purposes of this workshop, we're just going to look at how we can use **branching** to avoid making breaking changes to the code that others see and are basing our work on (that includes ourselves, too!).

9.1 When To Branch

There are a couple of different philosophies on when to **branch**, but the one that's now encouraged in software development is to make small, discrete **branches** for each feature, and regularly **merge** them back into the **main** branch. It is commonly referred to as the GitHub Flow - see this video for an example using the command line instructions. What you generally want to avoid are long-lived **branches** that are worked on for a long time, because they can get out of sync with the **main** branch and become difficult to merge back in, which was traditionally the workflow that was advocated for by the "git flow" philosophy. Fortunately for us, the short-lived **branch** workflow is well-suited to the pace and development style of scientific software.

We could just code directly on the **main** branch, but if we make a mistake and want to roll back changes, this could have a big impact on the work of others. Instead, we can create a new **branch** for this feature, and then **merge** it back into the **main** branch when we're done. This allows others to keep working off the code on **main** while we work on our new feature in isolation. When our code is working as we want it, we can **merge** it back into the **main** branch, and they will then be able to access it.

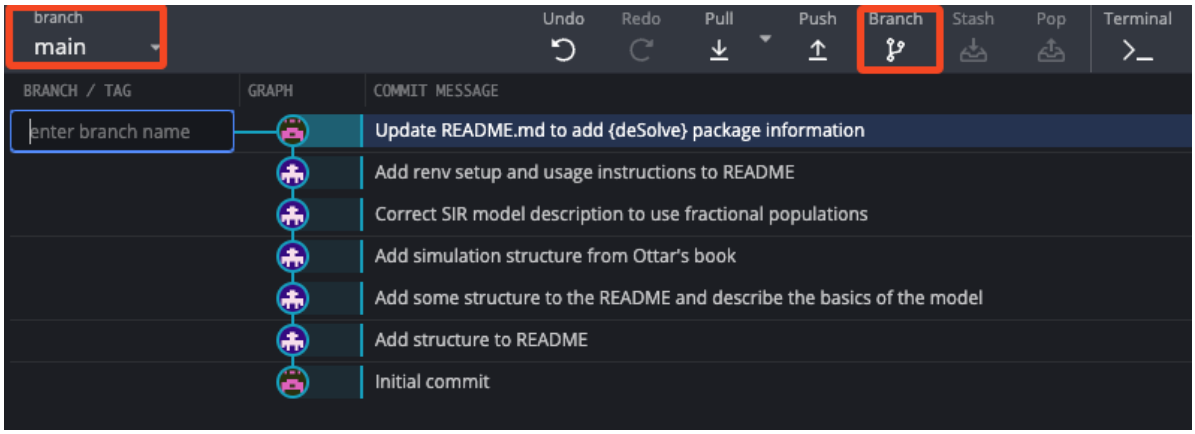
Let's go back to our SIR model example and see how we can use **branching**.

9.2 Branching Example

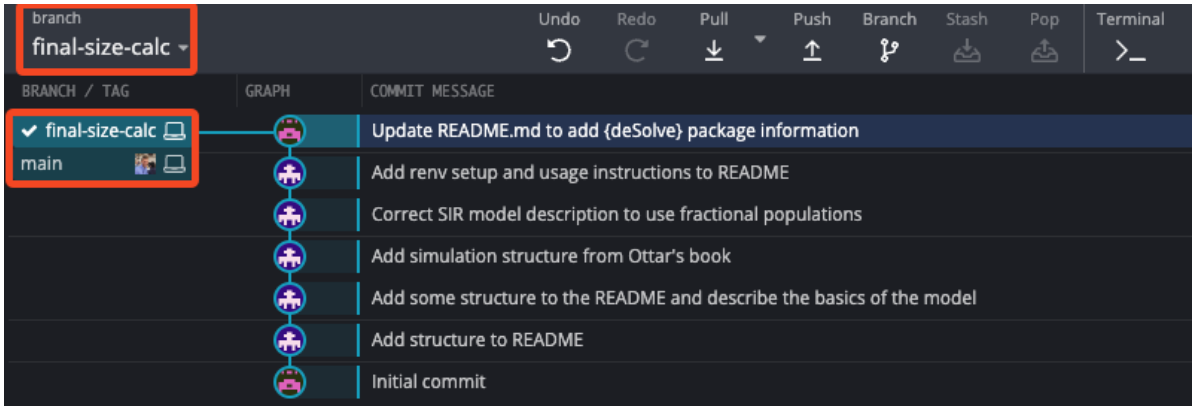
9.2.1 Creating a Branch

Let's say that we want to add some code that calculates the final size of the epidemic for a range of transmission parameters (i.e., varying R_0 to adjust β). Because our desired feature is a nice discrete piece of work that isn't dependent on other work, we can make a **branch** for it. Importantly, it's a relatively small and self-contained feature so it shouldn't proliferate into a long-lived **branch** as we work on it i.e., we shouldn't end up in a situation where there is a never-ending list of tasks to perform before we can consider the **feature** complete.

To create a new **branch**, we navigate to GitKraken and make sure our **local** machine is up to date with the **remote** repository i.e., perform a **pull** and resolve any conflicts. Making sure we're on the **main** branch (we should be as we haven't created any branches yet), we can create a new **branch** by clicking on the *"Branch"* button in the middle of the GitKraken toolbar, and then typing in a branch name.



After entering a name, a new **branch** will be created, and we can see that we're now on the new **branch**, **final-size-calc**. Hovering our mouse of the branch icon, we can see that both **main** and **final-size-calc** are listed, and we can switch between them by double-clicking on the branch we want to switch to. This also shows that both **branches** are currently at the same **commit**, as indicated by the horizontal marker pointing to the same **commit**.



Now we're ready to start adding code.

Terminal Commands

```
# checkout changes the branch
# -b creates a new branch with the given name
git checkout -b final-size-calc
```

9.2.2 Adding Code

Add the following R code to the end of the ***sir_model.R*** / ***sir_model.py*** file:

R Code

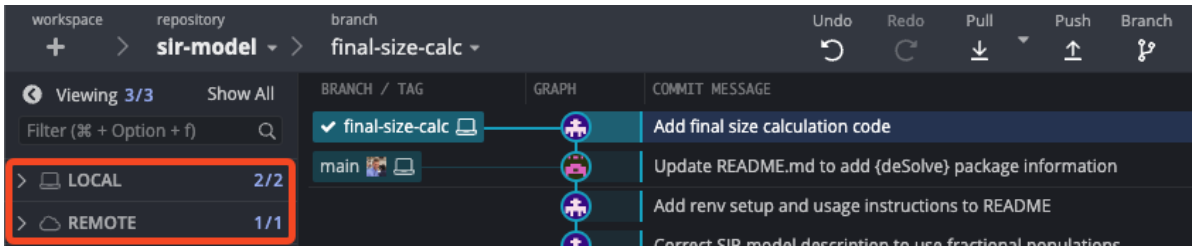
```
# Candidate values for R0 and beta
R0 <- seq(0.1, 5, length = 50)
betas <- R0 * 1/2

# Calculate proportion infected for each value of R0
# map2_dfr is a {purrr} function that applies a function to two vectors i.e., it is a vector
final_size_df <- map2_dfr(
  .x = betas,
  .y = R0,
  .f = function(.x, .y) {
    equil <- runsteady(
      y = c(S = 1 - 1E-5, I = 1E-5, R = 0),
      times = c(0, 1E5),
      func = sirmod,
      parms = c(mu = 0, N = 1, beta = .x, gamma = 1/2)
    )
```

```
    tibble(
      R0 = .y,
      final_size = equil$y["R"]
    )
  }
)

ggplot(final_size_df, aes(x = R0, y = final_size)) +
  geom_line(linewidth = 2) +
  labs(x = "R0", y = "Final size")
```

Python Code

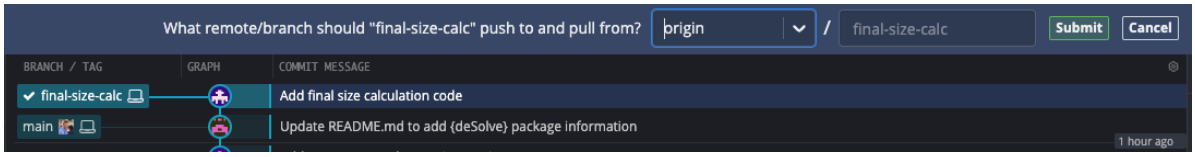
```
# Candidate values for R0 and beta
R0 = np.linspace(0.1, 5, 50)
betas = R0 * 1 / 2

solve_ivp(
    sirmod, [tmin, 1e5], start, args=parms
).y[2, -1]

final_size_df = pd.DataFrame({"R0": R0, "final_size": np.zeros(len(R0))})

for (index, beta) in enumerate(betas):
    p = (beta, mu, gamma, N)
    final_size_df.final_size[index] = solve_ivp(sirmod, [tmin, 1e5], start, args=p).y[2, -1]

(
    ggplot(final_size_df, aes(x='R0', y='final_size'))
    + geom_line(size=2)
    + labs(x='R0', y='Final size')
)
```

After **staging** and **committing** the changes, we'll see that our **final-size-calc** branch is now ahead of the **main** branch by one **commit**. We'll also notice that **final-size-calc** only exists locally, and not on the **remote** repository.



This is expected as we haven't **pushed** the **final-size-calc** branch to the **remote** repository yet, so GitHub doesn't know about it. So let's **push** the **commit**.

GitKraken will immediately ask us to name the **remote** branch. Click *"Submit"* with the default to avoid any confusion that may be caused by using a different name **locally** and **remotely** for the same **branch**.



We'll now see that both our **local** machine and GitHub are pointing to the same **commit** for the **final-size-calc** branch.

Terminal Commands

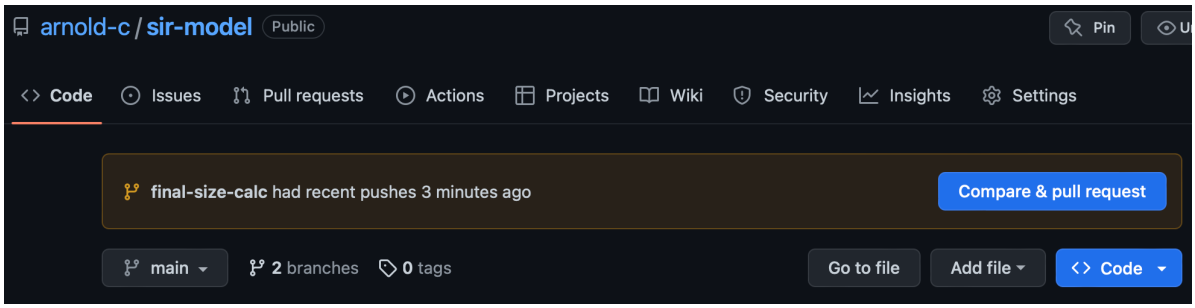
```
# Stage and commit the changes
git add .
git commit -m "Add final size calculation code"

# Show git tree
git log --graph --decorate --oneline --all

# Push the commit to the remote repository
# -u sets the upstream branch to the remote branch of the same name
# origin is the name of your remote (the default name)
# final-size-calc is the name of the local branch you want to push
git push -u origin final-size-calc
```

9.2.3 Merging the Branch

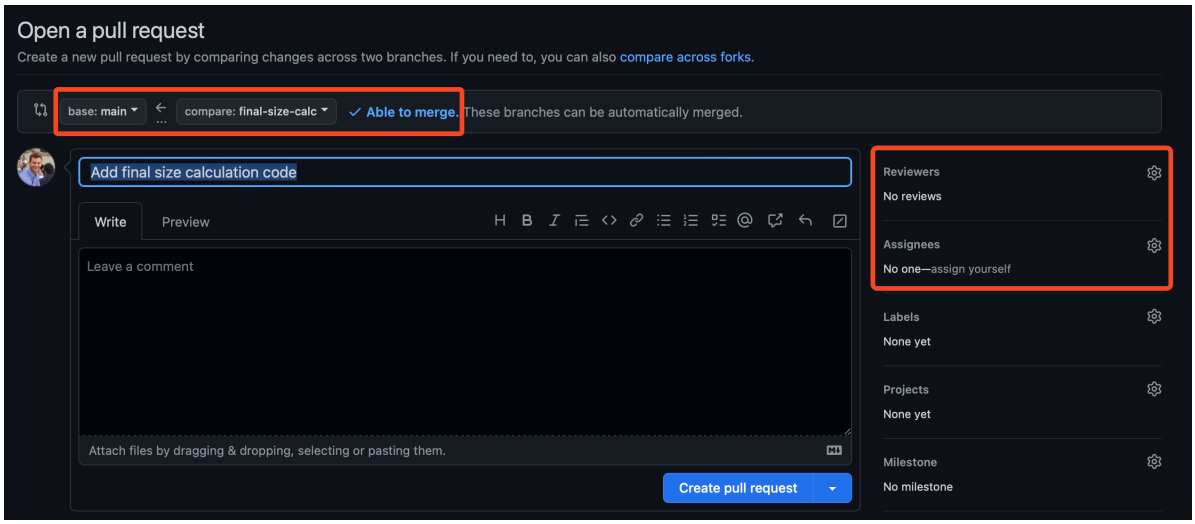
Now we've added code and **pushed** it to GitHub, we can keep going and add more code to the **final-size-calc** branch. But in this case, we're done, so we can **merge** the **final-size-calc** branch back into the **main** branch. You can do this on your **local** machine, but let's do it on

GitHub with a **pull request**, as this is more appropriate for collaborative work, and doesn't require any additional steps when we're working on our own, so is a better default workflow.

Navigating to GitHub, we see that **final-size-calc** has recent **pushes** and GitHub prompts us to start a **pull request**. Go ahead and click *"Compare & pull request"*.



Doing so, we are presented with a lot of information. The key points are that there are two buttons at the top of the page that allow you to select which **branches** are being compared in the **pull request**, i.e., you want to **merge** together. The **base** branch is the **branch** that you want to **merge** into, and the **compare** branch is the **branch** that you want to **merge** from. So in this example, our changes happened on the **final-size-calc** (compare) branch, and we want to **merge** them into the **main** (base) branch. Because there are no conflicting changes, e.g., changes to the same lines of code, GitHub can automatically **merge** the **branches** for us, hence the "Able to merge" message.



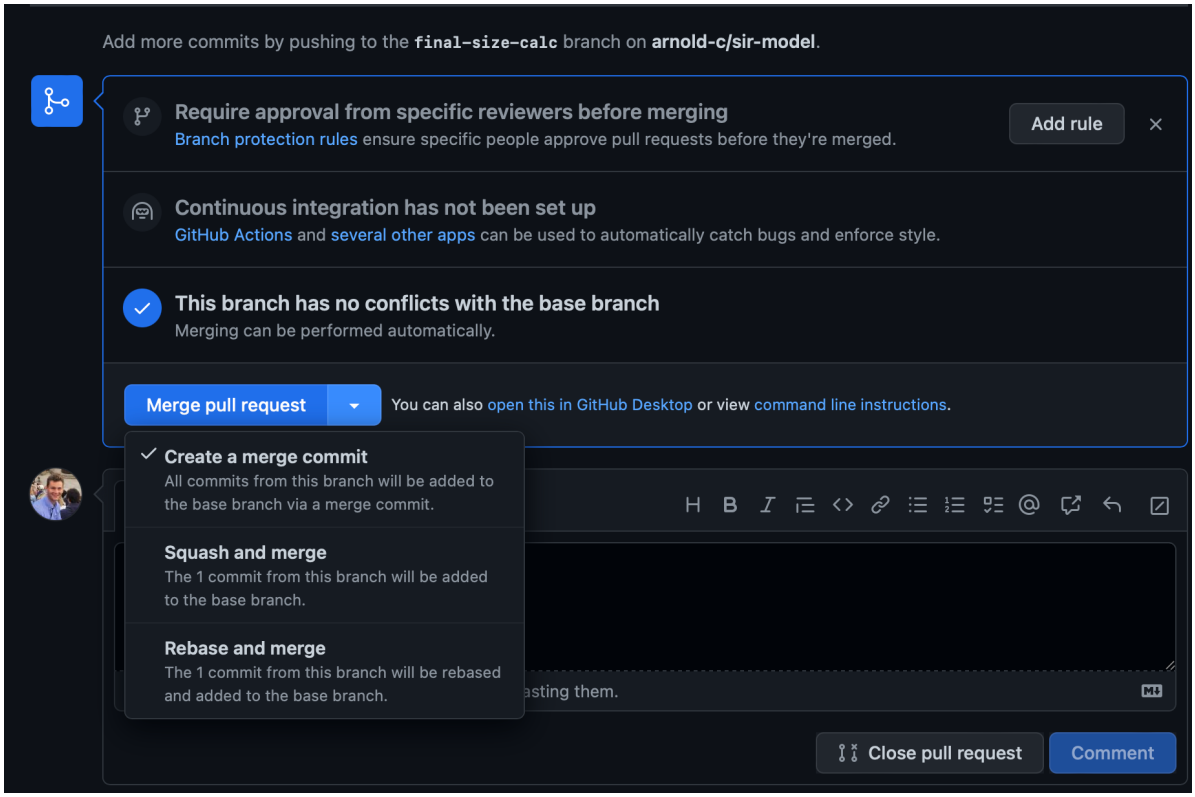
As we only made one **commit**, the **pull request** message defaults to the **commit** message. In instances where you've made multiple **commits**, you can edit the **pull request** message to provide a more detailed description of the changes you've made.

On the right hand side of the **pull request** page, you will see the option to add reviewers, assignees etc. This is useful for collaborative work where you want collaborators to check your

work for correctness, and is a good habit to get into. You can also use GitHub Projects to keep track of your work and upcoming tasks, so linking to a project can help you visualize the work you're doing and how it fits into the bigger picture.

After clicking *"Create pull request"*, GitHub will go through some checks to make sure that the **branches** can be **merged**, and notify any reviewers that you've added. It uses GitHub Actions to do this, which is a powerful tool for automating tasks, and worth exploring in more detail when you're comfortable with Git and GitHub.

Assuming everything checks out, you can click *"Merge pull request"* to **merge** the **branches**.



> **Note**
>
> Clicking on the drop-down menu of the *"Merge pull request"* button will allow you to choose how to **merge** the **branches**. The default retains all of the **commits** in the comparison **branch** separate, which is useful if you want to keep a record of all the changes you made.
> *"Squash and merge"* will combine all of the **commits** on the comparison **branch** into a single **commit**, which is useful if you want to don't need to keep a record of exactly how you ended up with the final version of the new code, and just want to show that you added the feature.

> *"Rebase and merge"* will add all the **commits** from the comparison **branch** to the end of the **base** branch, which is useful the files modified in the **pull request** are in conflict with changes in the **base** branch.
> You can find more information about the different **pull request merge** methods here.

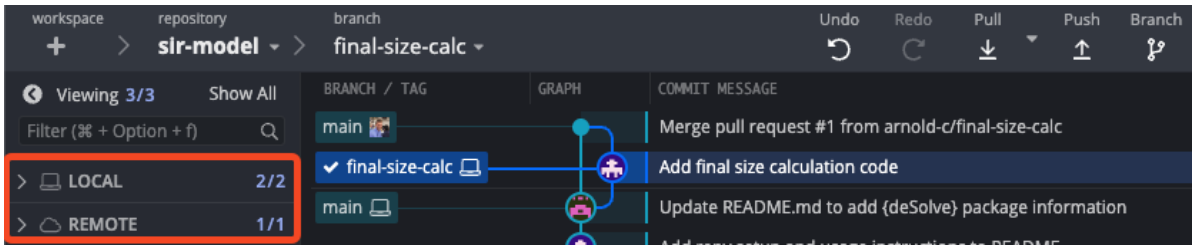
9.2.4 Cleaning Up Our Repository

After successfully **merging** the **branches**, we can delete the **final-size-calc** branch as it is no longer needed. Don't worry, you still have your full Git history, and the branch can always be restored if required, but this is a good way to keep your repository tidy and avoid accidentally making changes to an old **branch**.

To delete the **remote** branch, just click on the *"Delete branch"* button that appears after merging the **pull request**.

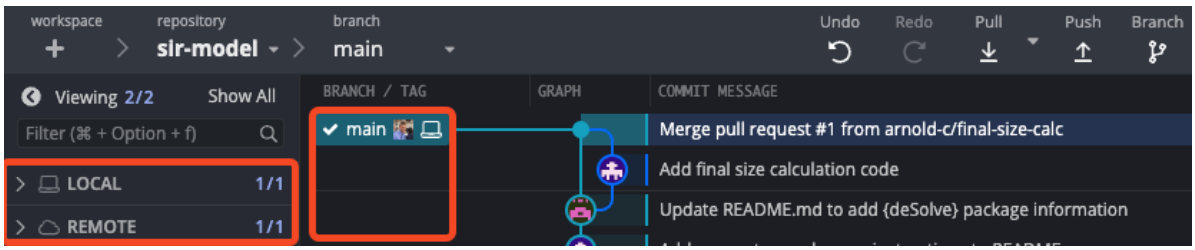
At this point, we can head back to GitKraken. One really nice feature of GitKraken is that it's easy to see what we've done as it shows us the Git tree. This is something that you can't do in the command line, or the GitHub Desktop application, and is a really useful way to keep track of your work, particularly if you have a lot of, or long-lived, **branches** (which you know you shouldn't!).

Here, we can see that there is now only one **branch remotely**, but two **locally**.



If we double-click on the **main local branch**, we will **checkout** that branch, i.e., we will switch to it. We then **pull** the changes from the **remote** and **main** will now be up to date on our **local** machine.

If we now right click on the **final-size-calc local branch**, we can delete it by selecting the option *"Delete final-size-calc"*. We will now see the updated Git history.



Terminal Commands

```
# Delete the remote branch
git push -d origin final-size-calc

# Delete the local branch
git branch -d final-size-calc
```

10 How to Collaborate

Often in science, we're not the only people who are working on a project. If we are working with others, it is worth learning about effective ways to use Git, allowing for asynchronous collaboration and minimizing the risk of **merge conflicts**. Fortunately for us, much of this revolves around the **branching** concept outlined in the previous chapter.

10.1 Feature Branches

The main principal is that every new feature/change, where that is a bug fix or new model, has its own branch. Ideally, each one will will only be a small change, but sometimes this gets away from us and results in 500 lines changes.

10.1.1 Why GitHub Issues

GitHub has a couple of useful features that make this easy to work with, as well as track progress in your todo list of features. The first key concept is the liberal use of **GitHub issues**. For every feature I want to add, I create an **issue** (I promise this is going somewhere). Each issue should be small enough that is only does one thing and is easy to understand and review (though it will likely be comprised of multiple **commits**).

For example, say I have just added my final size calculation code and I realize that I want update my model to be an SEIR model instead of an SIR model. I could just make a new branch and be done with it, but creating an **issue** allows us to track which **commits** are involved in the change. It also allows us to outline a number of changes we wish to make over the next short while and link them, making it easier to remember what we wanted to do when we come back to a project, as well as who was working on the changes.

10.1.2 Creating an Issue

To create an issue, simply navigate to your project in GitHub, click on the Issues tab, and then on the "New Issue" button.

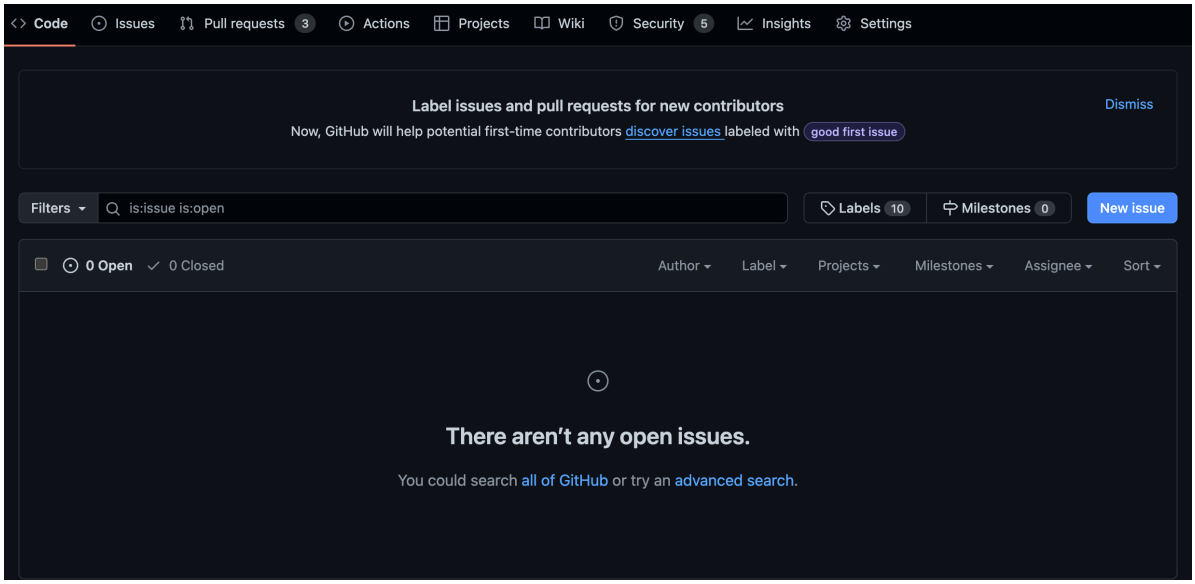


Figure 10.1: Creating an issue

From here, add a short and descriptive title, explanation, and assign the issue to someone (if you're the only person in the project, this obviously isn't necessary). You may also find it useful to add a label to the issue to distinguish it from other issues later on (e.g., use the default "enhancement" label for new features and "bug" for bugs).

> **Tip**
>
> You can also create a GitHub issue from directly within GitKraken if you are signed into your GitHub account.

Once you have created the issue, you will see an option on the right side that suggests you "Create a branch", under the "Development" heading. You can use this (I would recommend this approach as I find it a bit cleaner - it only requires a **fetch** and **checkout** to get working on it), or you can open up GitKraken and do the same thing there as GitKraken will create a local branch that you can get working in straight away. I'm demonstrating this way just in case you come across this feature and it doesn't seem to connect the two as you'd expect (as it did for me, which is why the two screenshots refer to different issues). Opening up your repository, you will see in the sidebar a heading "GitHub Issues". Clicking on that will reveal your issues, and upon selecting the correct one you will see an option to create a feature branch.

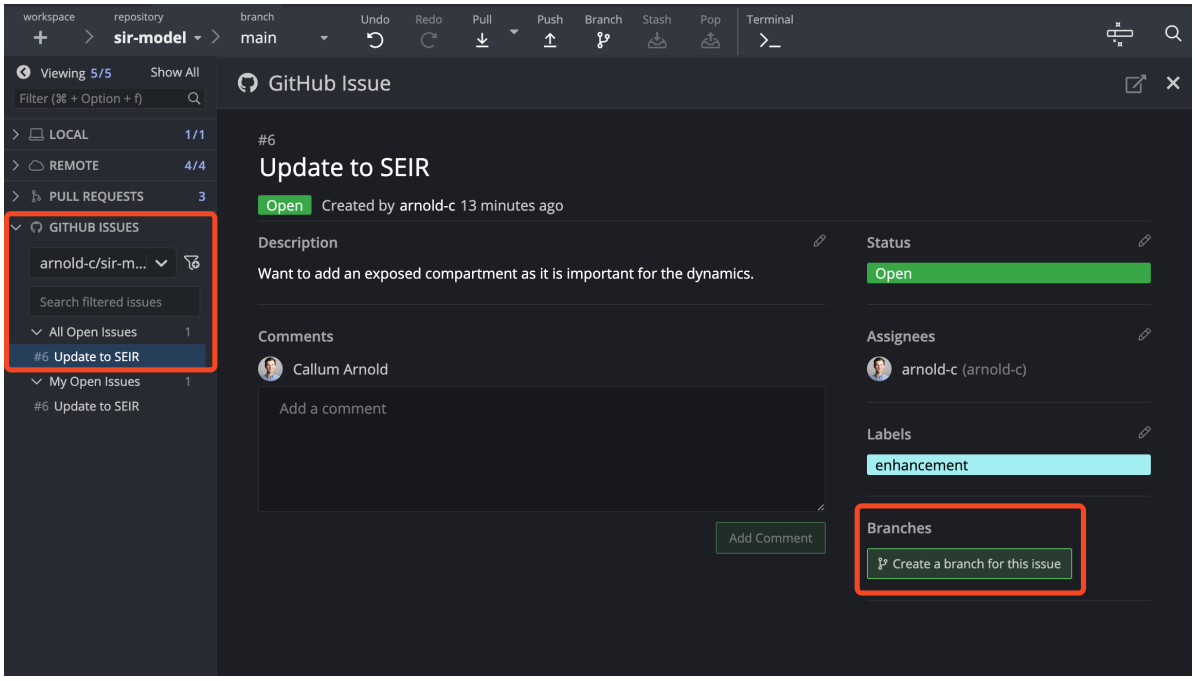


Figure 10.2: Creating an issue branch in GitKraken

Going through the next steps, you can name your branch whatever you like (though I tend to leave the default suggestion as they're not hanging around for long), and which branch you want to use for the base. If you're following the short-lived branch strategy, your base should probably be the **main** branch, an no others should be around for long enough that extra features can be branches off of them. Once you have your new **local** branch created, you should push it so there is a copy on your **remote** repository. There is just one final thing you need to do - connect your branch to your issue. Clicking on the "Development" heading of the issue will provide you with a menu to search all your repositories and then your branches that you can choose to link with the issue. At this point, you're ready to update your code.

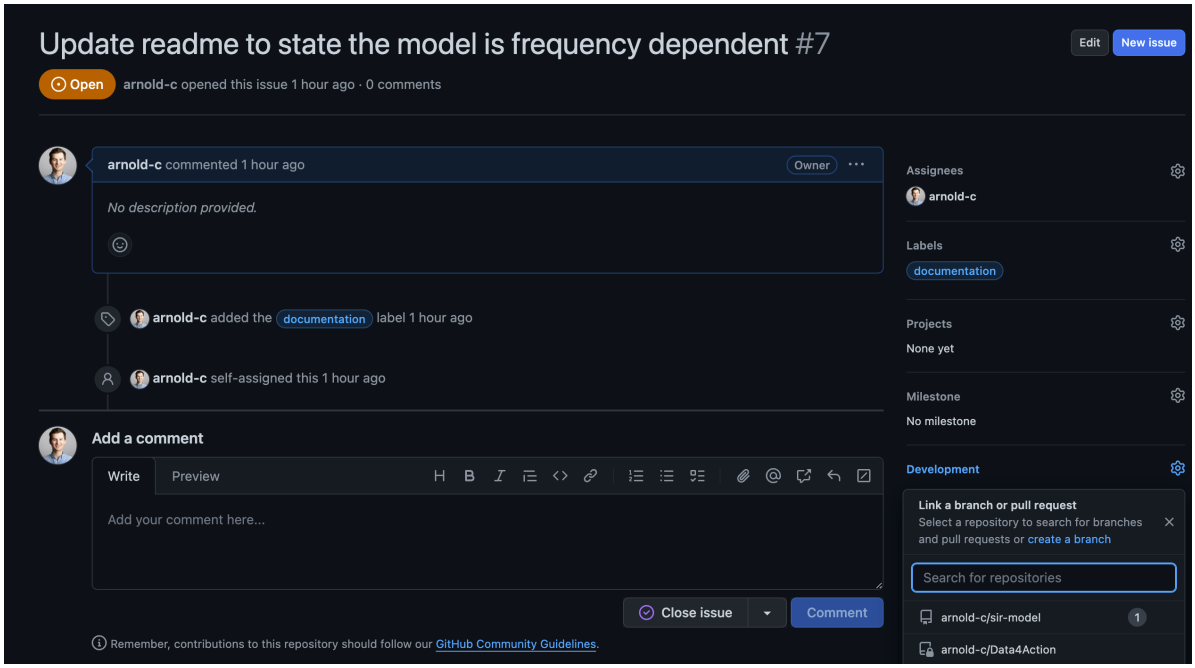


Figure 10.3: Linking your branch to your issue in GitHub

Below is code that updates the simulation files (*simulation.R* and *simulation.py*). Go ahead and update your code and then **commit** and **push** your changes to your remote repository.

R Code

```r
library(deSolve)
library(tidyverse)
library(rootSolve)

theme_set(theme_minimal())

seirmod <- function(t, y, parms) {
  # Pull state variables from y vector
  S <- y[1]
  E <- y[2]
  I <- y[3]
  R <- y[4]

  # Pull parameter values from parms vector
  beta <- parms["beta"]
  sigma <- parms["sigma"]
```

```r
  mu <- parms["mu"]
  gamma <- parms["gamma"]
  N <- parms["N"]

  # Define equations
  dS <- mu * (N - S) - beta * S * I / N
  dE <- beta * S * I / N - sigma * E
  dI <- sigma * E - (mu + gamma) * I
  dR <- gamma * I - mu * R
  res <- c(dS, dE, dI, dR)

  # Return list of gradients
  list(res)
}


times <- seq(0, 26, by = 1 / 10)
parms <- c(mu = 0, N = 1, beta = 2, sigma = 1, gamma = 1 / 2)
start <- c(S = 0.999, E = 0.0, I = 0.001, R = 0)

out <- ode(y = start, times = times, func = seirmod, parms = parms)
out_df <- as_tibble(out) %>%
  pivot_longer(cols = -time, names_to = "state", values_to = "number") %>%
  mutate(
    time = as.numeric(time),
    number = as.numeric(number),
    state = factor(state, levels = c("S", "E", "I", "R")),
    number = round(number, 6)
  )

ggplot(out_df, aes(x = time, y = number, color = state)) +
  geom_line(linewidth = 2) +
  labs(x = "Time", y = "Number", color = "State")


# Candidate values for R0 and beta
R0 <- seq(0.1, 5, length = 50)
betas <- R0 * 1 / 2

# Calculate proportion infected for each value of R0
# map2_dfr is a {purrr} function that applies a function to two vectors i.e., it is a vector
final_size_df <- map2_dfr(
```

```
  .x = betas,
  .y = R0,
  .f = function(.x, .y) {
    equil <- runsteady(
      y = c(S = 1 - 1E-5, E = 0.0, I = 1E-5, R = 0),
      times = c(0, 1E5),
      func = seirmod,
      parms = c(mu = 0, N = 1, beta = .x, sigma = 1, gamma = 1 / 2)
    )

    tibble(
      R0 = .y,
      final_size = equil$y["R"]
    )
  }
)

ggplot(final_size_df, aes(x = R0, y = final_size)) +
  geom_line(linewidth = 2) +
  labs(x = "R0", y = "Final size")
```

Python Code

```
# %%
import numpy as np
import pandas as pd
from scipy.integrate import solve_ivp
from plotnine import *


# %%
def seirmod(t, y, beta, mu, sigma, gamma, N):
    # Unpack states
    S, E, I, R = y

    # Define equations
    dS = mu * (N - S) - beta * S * I / N
    dE = beta * S * I / N - sigma * E
    dI = sigma * E - (mu + gamma) * I
    dR = gamma * I - mu * R

    # Return list of gradients
```

```
    return dS, dE, dI, dR


# %%
tmin = 0
tmax = 26
tstep = 1 / 10
times = np.arange(tmin, tmax, tstep)

beta = 2
mu = 0
sigma = 1
gamma = 1 / 2
N = 1
parms = (beta, mu, sigma, gamma, N)

S0 = 0.999
E0 = 0
I0 = 0.001
R0 = 0
start = (S0, E0, I0, R0)

# %%
out = solve_ivp(seirmod, [tmin, tmax], np.array(start), args=parms, t_eval=times)

# %%
out_df = (
    pd.DataFrame(out.y).transpose().rename(columns={0: "S", 1: "E", 2: "I", 3: "R"})
)
out_df["time"] = out.t
out_df = out_df.melt(id_vars="time", value_vars=["S", "E", "I", "R"]).rename(
    columns={"variable": "state", "value": "number"}
)

# %%
theme_set(theme_minimal())

(
    ggplot(out_df, aes(x="time", y="number", color="state"))
    + geom_line(size=2)
    + labs(x="Time", y="Number", color="State")
)
```

```
# %%
# Candidate values for R0 and beta
R0 = np.linspace(0.1, 5, 50)
betas = R0 * 1 / 2

# %%
solve_ivp(seirmod, [tmin, 1e5], start, args=parms).y[2, -1]

# %%
final_size_df = pd.DataFrame({"R0": R0, "final_size": np.zeros(len(R0))})

for index, beta in enumerate(betas):
    p = (beta, mu, sigma, gamma, N)
    final_size_df.final_size[index] = solve_ivp(seirmod, [tmin, 1e5], start, args=p).y[
        2, -1
    ]

# %%
(
    ggplot(final_size_df, aes(x="R0", y="final_size"))
    + geom_line(size=2)
    + labs(x="R0", y="Final size")
)
```

Now that you've updated the model structure, **committed** the changes, and **pushed** your changes to GitHub, you will also want to update the ***README.md*** file to indicate to readers the model that you are using.

10.2 Collaborating on the Same Feature

At the same time you're working on updating the model structure, one of your collaborators has decided to help you out and update the ***README.md*** file for the change you're working on. If all goes well, and you're not working on the same lines of the same file at the same time, you should be fine. In this case, the only thing to do is to regularly check GitHub for any changes that have been made to the project since you last **pushed** changes. Imagine the ***README.md*** file looks like this:

README.md

````
## SEIR Model
### About This Project

This is a test project to get used to using Git and GitHub.

The purpose of this project is to create a SEIR model in R.
An SEIR model is a model that describes the spread of a disease in a population, placing indi
The compartments are susceptible (S), exposed (E), infected (I), and recovered (R).
The model is described by the following equations:

```math
\begin{align}
\frac{dS}{dt} &= \mu (N - S) -\beta S \frac{I}{N} \\
\frac{dE}{dt} &= \beta S \frac{I}{N} - \sigma E \\
\frac{dI}{dt} &= \sigma E - \gamma I - \mu I \\
\frac{dR}{dt} &= \gamma I - \mu R
\end{align}
```

```math
\begin{align}
\mu &= \frac{1}{50*52} \\
\beta &= 2 \\
\sigma &= 1 \\
\gamma &= \frac{1}{2} \\\\

N &= 1.0 \\
S_0 &= 0.999 \\
E_0 &= 0 \\
I_0 &= 0.001 \\
R_0 &= 0.0
\end{align}
```

Here, $\mu$ is the mortality and birth rate, $\beta$ is the contact rate multiplied by the pe
The units for this are weeks, so the life expectancy is 50 years, and the duration of infecti
````

If you haven't made changes to the ***README.md***, pulling the changes won't cause any problems. Once you have finalized you changes, **push** them to GitHub and create a **pull request**, as we did in the branching chapter. If you have both made changes to the same lines of code, you will get a **merge conflict** as Git doesn't known which change to keep. This can be resolved, but we will demonstrate that in the troubleshooting examples using a slightly

different example where there are multiple branches changing the same lines of code, but the concept is exactly the same.

10.3 Collaborating on Different Features

More often than someone working on the same feature as you at the same time, you will split up the tasks between individuals. You may be working on updating the model to use the SEIR structure, and your collaborator will be working on a different feature, for example, editing the ***README.md*** file to be more descriptive about the model - it is a frequency dependent model as we are scaling the transmission terms by N.

> **Note**
>
> You don't need to follow along with this, but you are welcome to, to get a sense of how to deal with potential merge conflicts.

To do this, I'm first going to create a GitHub issue for the new change we want to make. This is simply going to be an issue with the "documentation" label that states the need to distinguish the model as frequency dependent in the ***README.md*** file. Then, I'm going to create a new branch for the feature we want to work on. However, before I create a new branch, I'm going to **checkout** the main branch, which is the one that contains the code we want to work on (your collaborator would not be creating a new branch off your short-lived branch).

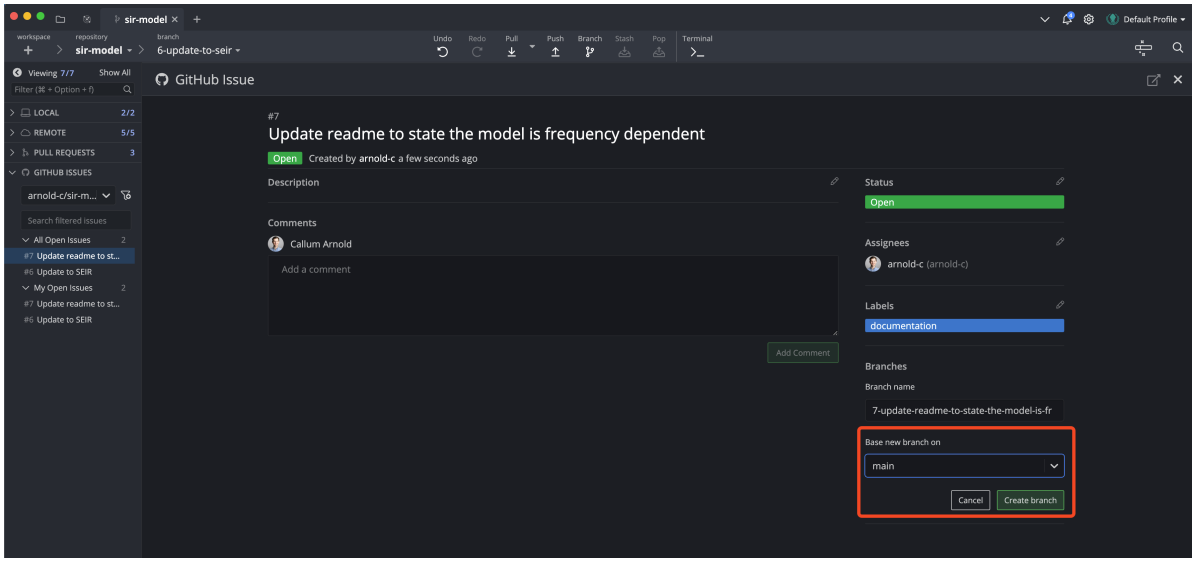


Figure 10.4: Creating a new feature branch off main

Once this branch has been created, you should see in GitKraken that your new feature branch exists in the same state as your **main** branch.

Now let's imagine our collaborator steps away from their computer for a while before they get a chance to modify the ***README.md*** file to indicate that we are working on a frequency dependent model.

Their changes might look like this:

README.md

````
## sir model
### about this project

this is a test project to get used to using git and github.

the purpose of this project is to create a frequency dependent sir model in r.
an sir model is a model that describes the spread of a disease in a population, placing indiv
the compartments are susceptible (s), infected (i), and recovered (r).
the model is described by the following equations:

```math
\begin{align}
\frac{ds}{dt} &= \mu (n - s) -\beta s \frac{i}{n} \\
\frac{di}{dt} &= \beta s \frac{i}{n} - \gamma i - \mu i \\
\frac{dr}{dt} &= \gamma i - \mu r
\end{align}
```

```math
\begin{align}
\mu &= \frac{1}{50*52} \\
\beta &= 2 \\
\gamma &= \frac{1}{2} \\\\

n &= 1.0 \\
s_0 &= 0.999 \\
i_0 &= 0.001 \\
r_0 &= 0.0
\end{align}
```

here, $\mu$ is the mortality and birth rate, $\beta$ is the contact rate multiplied by the pe
the units for this are weeks, so the life expectancy is 50 years, and the duration of infecti
````

Any time multiple people are working on a project and numerous features are being worked on, there is the potential for conflicts when the same file is edited simultaneously and Git doesn't know which is the "correct" version. To try and avoid that problem, our collaborator should be regularly checking to see if we have made changes to files they are working on (in this case the ***README.md*** file). Because we've committed and pushed regularly, they see we have changed the file and will want to pull down the changes into their branch. And given we're working in a short-lived branch, we will have just completed our changes and submitted them as a pull request, but the issue here is that the base of our collaborator's branch is now out of date - it refers to before you implemented that changes to model structure and ***README.md***. So how does our collaborator resolve this issue?

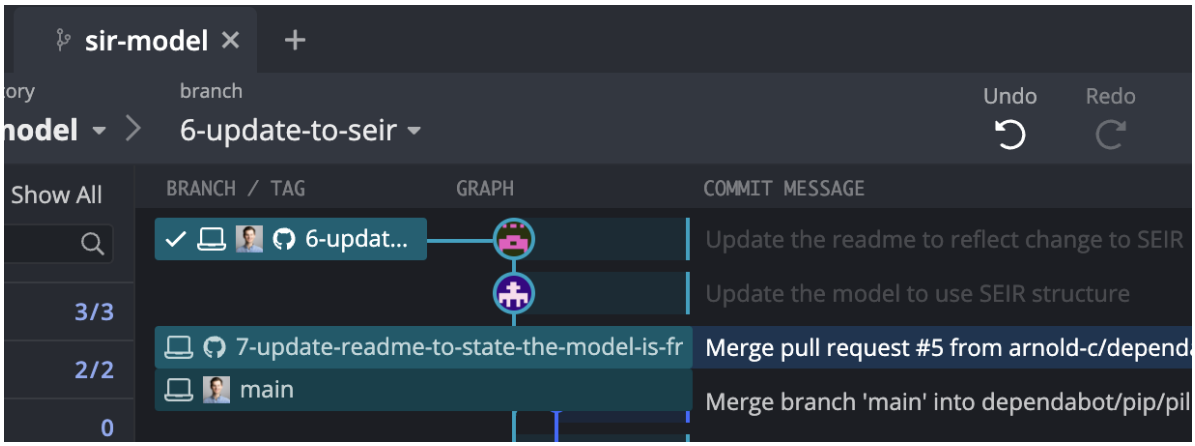


Figure 10.5: Ignore the merge pull request messages - that is from automatically updating package dependencies

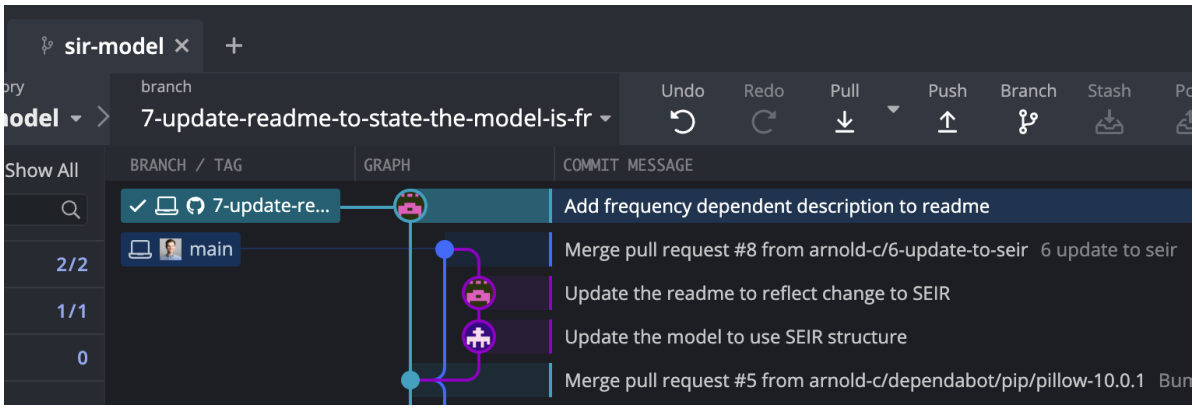


Figure 10.6: New feature branch is based of an out of date main branch

There are a number of methods to do this, but I'll just outline the common methods you may

see here.

10.3.1 Carry On as Usual & PR

In this strategy, your collaborator doesn't need to do anything different. They will simply finish their changes, **push** to their feature branch on GitHub, and then create a pull request to merge back into the **main** branch. Doing this, they will be faced with a merge conflict on trying to complete the pull request. They can go ahead and create the PR, but they will have to resolve the conflicts before they can be merged into the **main** branch.

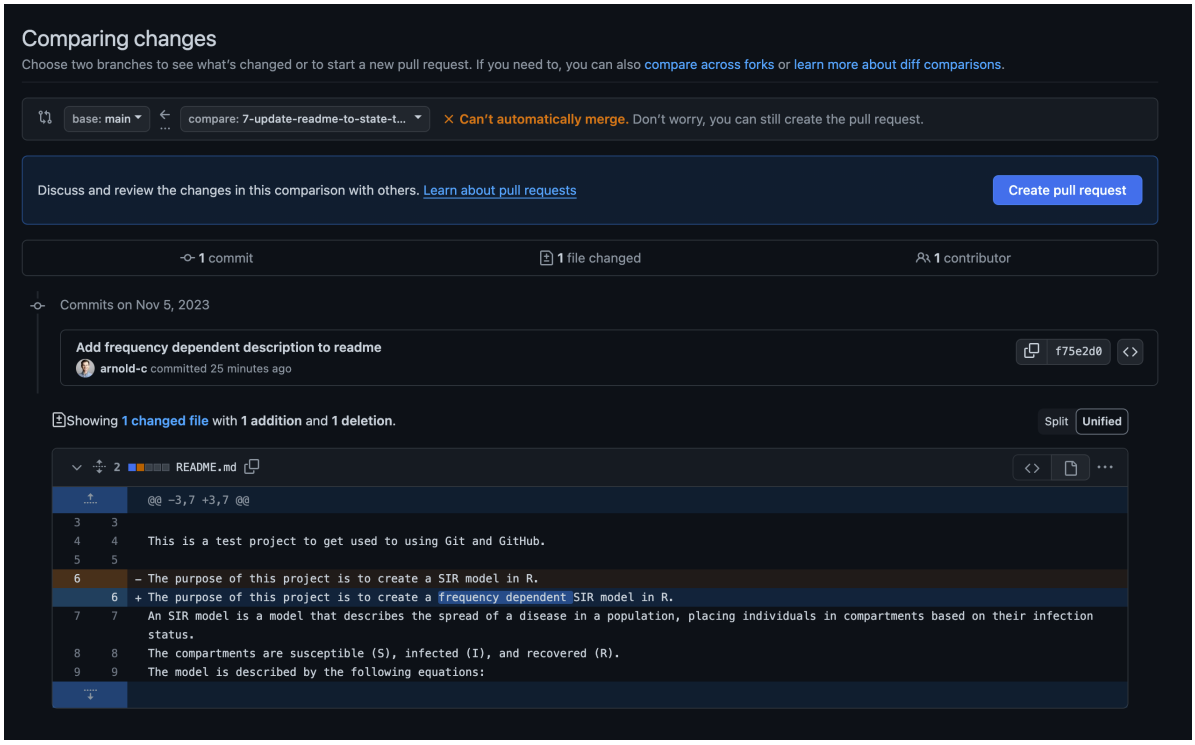


Figure 10.7: Pull request warning

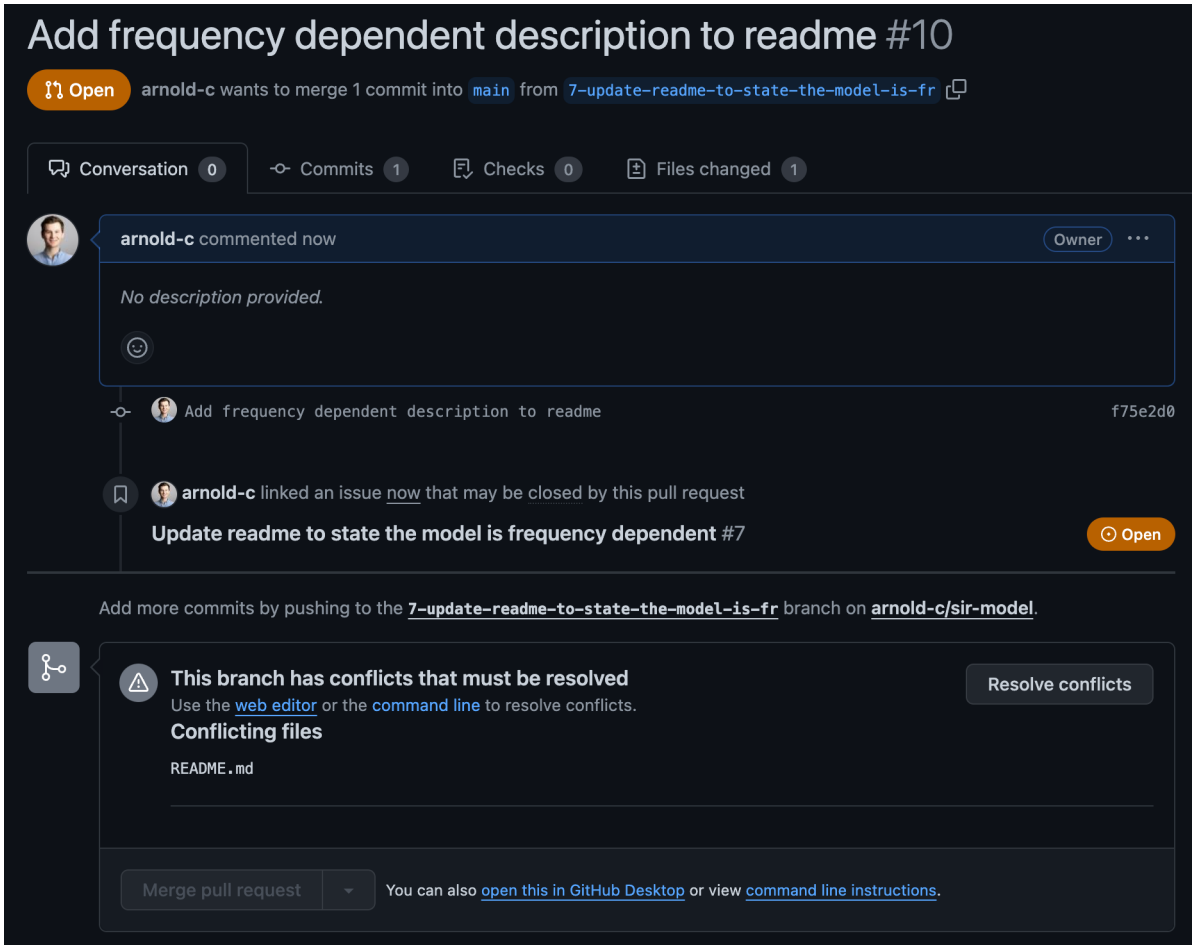


Figure 10.8: PR creation directs you the conflicts

Clicking on the "Resolve conflicts" button will bring up an online editor that will allow them to select the correct version of the code for each conflicting line. The sections that conflict will be between:

```
<<<< feature-branch
...
=======
...
>>>> main
```

Figure 10.9: GitHub Pull request conflict editor

After deleting the incorrect code and surrounding braces, the collaborator can just click on the "Mark as Resolved" button in the top right corner (not shown above) to save this update to the file (often there is more than one file that needs to be corrected), and then click on "Commit merge".

10.3.2 Rebase & PR

This method requires that your collaborator first updates their **main** branch to the most recent version that is on the **remote**. From here, there ensure they are checked out on their **feature branch**. Right clicking on **main** will bring up a menu that has the option "Rebase feature branch onto main".

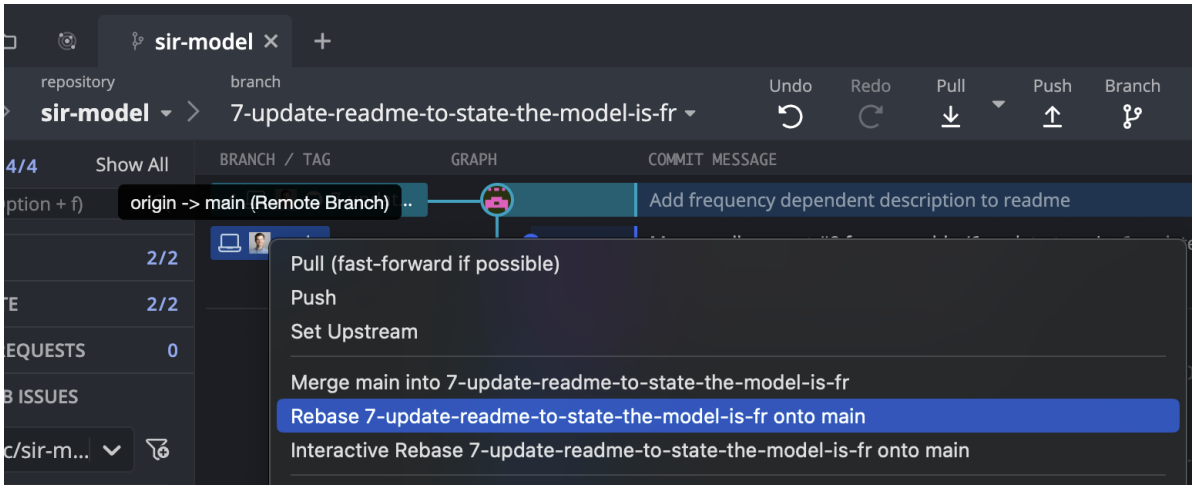


Figure 10.10: Rebase feature branch onto main

This will try to perform a **rebase**, and will fail due to a conflict, which it will ask your collaborator to resolve. The conflict can be fixed in GitKraken, or in a regular editor.

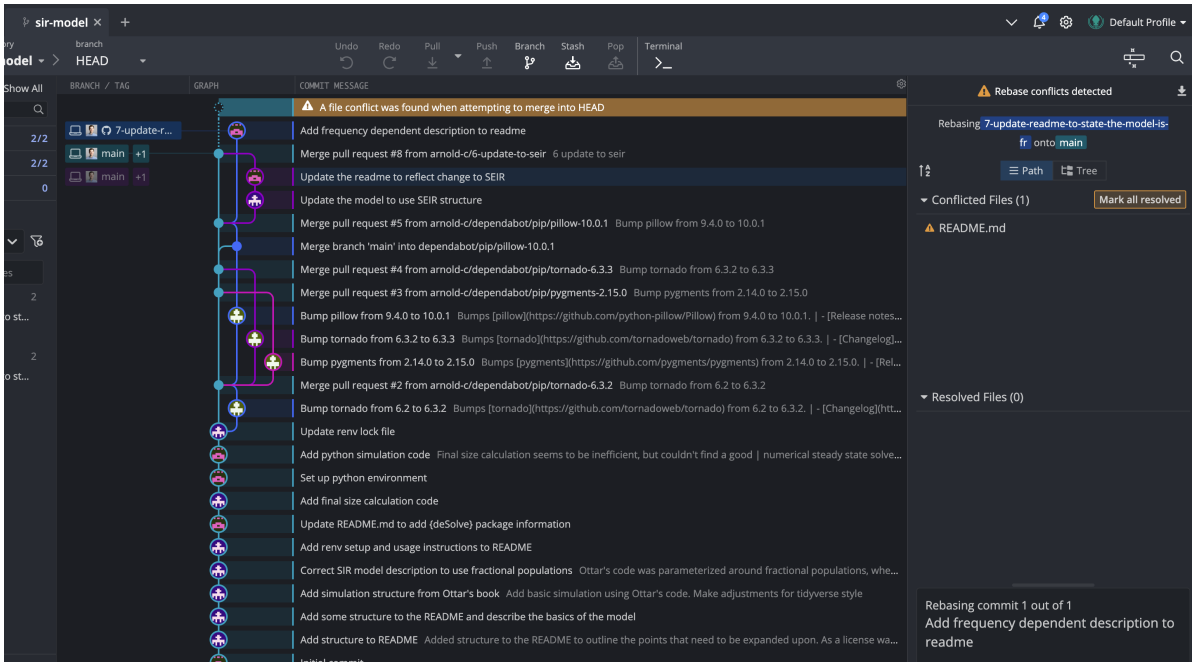


Figure 10.11: Rebase conflict warning

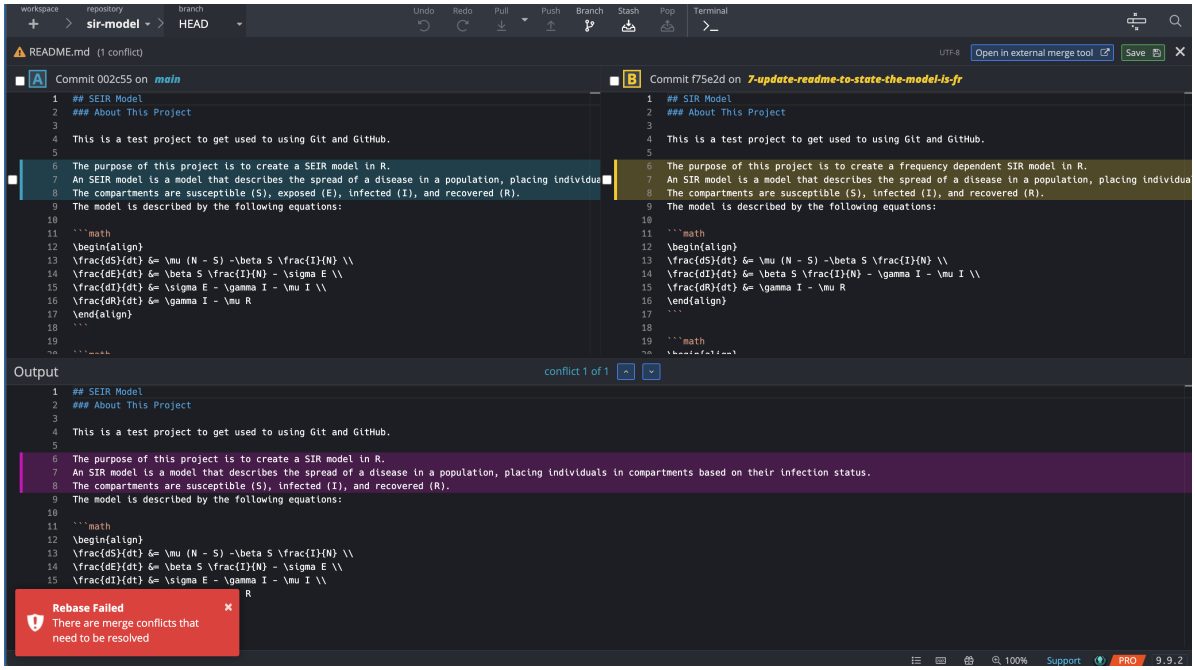


Figure 10.12: Rebase conflict editor

As you can see, the way **rebase** works is a little different than **merge**. Instead of creating a new commit to join the branches, **rebase** copies the commits made on the feature branch and then tries to sequentially apply then to your base branch (in this case, the **main** branch). As a result, after the **rebase** has been successfully completed, the base of the feature branch has been shifted up the Git tree. To visualize this, take a close look at where the feature branch joins **main** in the before and after (below) images.

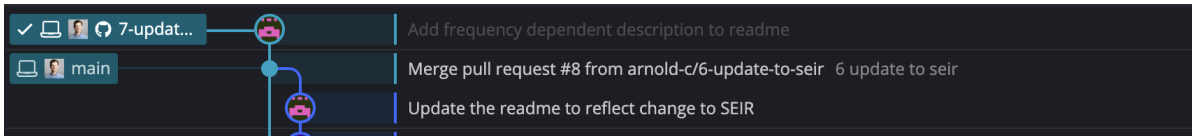


Figure 10.13: Rebase shifts the feature branch up

> **Warning**
>
> Because **rebase** creates a copy of your commits to reapply (it doesn't just move them), your commit history is technically different and changed. For this reason, you should be careful about rebasing when someone else has already checkout out the feature branch, as you will be altering Git history and no longer pointing to the same commits (even though the code is identical between the original and the copy **rebase** creates).

Now, your collaborator can **push** the corrected code to GitHub and create a pull request to merge back into the **main** branch.

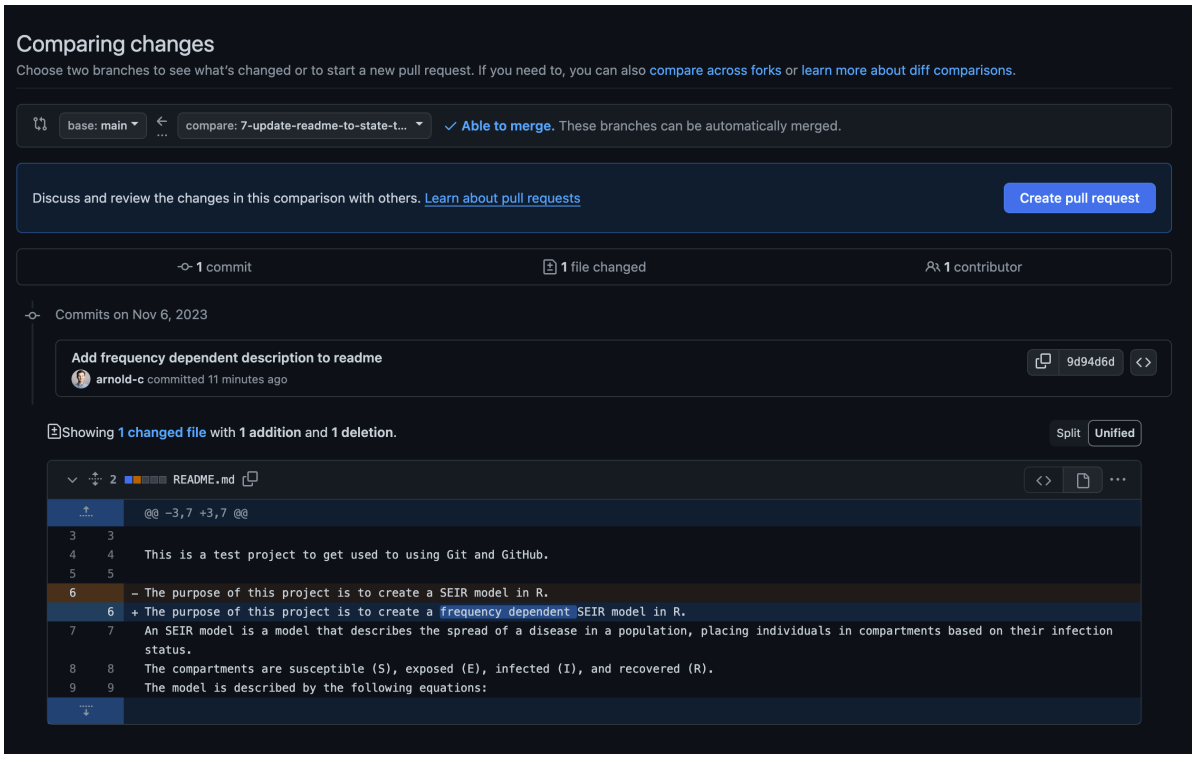


Figure 10.14: Pull request to end the rebase

> **Tip**
>
> To learn more about Git **rebase**, I would strongly recommend watching this short video by The Modern Coder, which does an amazing job of explaining the difference between **merge** and **rebase** and why one may be an advantage over the other in certain contexts.

10.3.3 Local Merge

First, **checkout** the feature branch, ensure your **main** branch is up to date, and then right click on **main** and select "merge feature branch into main". As before, you will be prompted to resolve the conflicts, and after doing so, can **push** the changes to GitHub where **main** will be updated.

> **Warning**
>
> If you are using **merge**, you should be careful about merging your changes into the **main** branch as this should represent the correct version of the code. For this reason, it is often

advised that you apply branch protection rules to the **main** branch to prevent accidental changes to the **main** branch that are not complete features. If you do this you will no longer be able to **push** directly to the **main** branch, so the **merge** method will not work, and will instead need to create a pull request. This has the added benefit of allowing your collaborators to review the changes before they are merged into the **main** branch.

10.3.4 Local Merge & PR

There is a second **merge** method - instead of merging **feature branch** into **main**, you can merge **main** into **feature branch**. This process is identical to the **rebase** method, except that you use **merge** instead of **rebase**. This has the advantage (relative to the other **merge** method) that you can **push** directly to the **feature branch** without needing worry about branch protection rules stopping your **push**. And relative to the rebase method, it doesn't copy and reapply the commits you have made, so you are not altering the Git history, making it safer when multiple individuals may be working on the same feature branch simultaneously. However, it has the disadvantage that it produces more merge commits, so your Git history is a little messier than with the rebase method, so you will need to decide how much that matters to you (probably shouldn't be much).

First, checkout the **main** branch this time, ensure your **main** branch is up to date, and then right click on the **feature branch** and select "merge main into feature branch". As before, **checkout** the feature branch where you will be prompted to resolve the conflicts, and after doing so, can **push** the changes to GitHub where **feature branch** will be updated. Finally, create a pull request to merge back into the **main** branch.

10.3.5 Pull Directly into the Branch

This is quite a nice and clean method to resolve conflicts. The only thing your coworker needs to do is to use the following command in the terminal (assuming they have already checked out the feature branch):

```
git pull origin main
```

This will result in exactly the same process as the PR version above, but this time it is within the **local** branch, so your collaborator can edit the files directly in the code editor rather than GitHub. It might also show `HEAD` instead of the feature branch name.

```
<<<< HEAD
...
=======
```

```
...
>>>> main
```

> **Tip**
>
> If you get a warning about `hint: You have divergent branches and need to specify how to reconcile them`, you might need to look into either using the flag `--ff-only` or `--no-ff`.

Here, it is also possible to view the conflict (and edit it) in GitKraken. It looks like this:

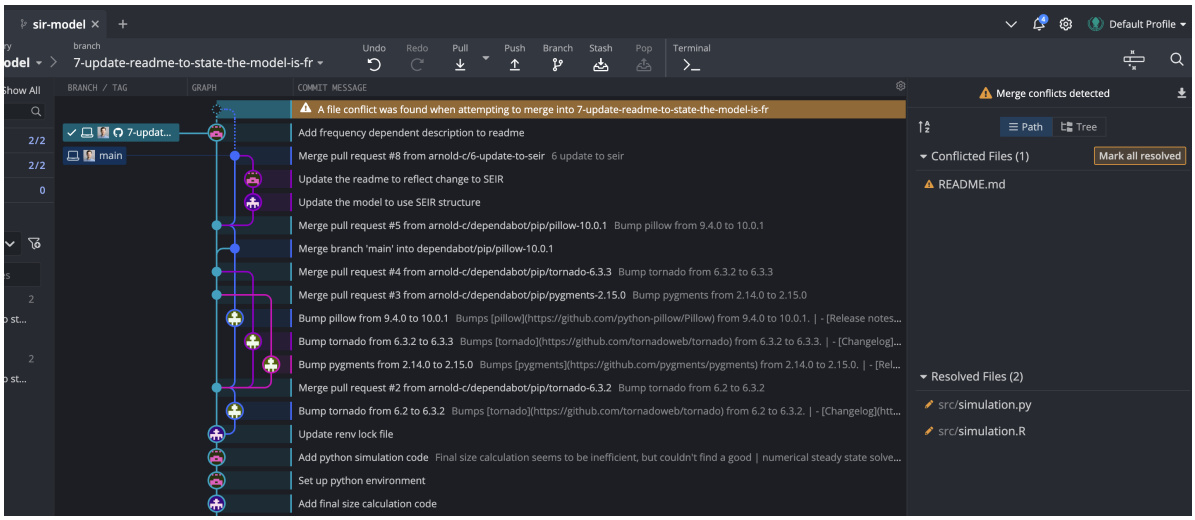


Figure 10.15: GitKraken merge conflict

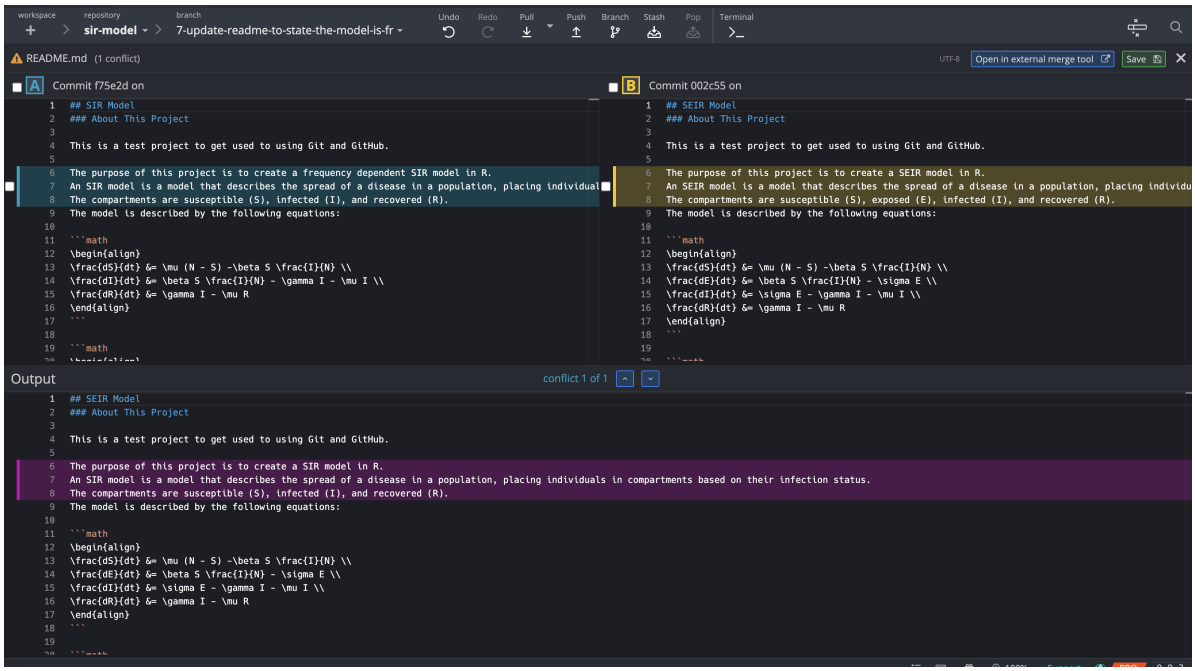


Figure 10.16: GitKraken conflict editor

Similar to the **rebase** method, your collaborator can then **push** the corrected code to GitHub and create a pull request to merge back into the **main** branch.

11 Recommended Practices

Congratulations for making it this far! You've learned enough to be able to start using Git and GitHub in your research workflow. This section will summarize the lessons learned so far and package them into a set of recommended practices.

- If you can, create your project on GitHub before **cloning** to your **local** machine
  - Use GitKraken to do this as it will take care of **cloning** the repository for you
  - Make sure you initialize the project with a ***README***, ***.gitignore***, and ***LICENSE***
- Add the default text to the ***README*** file
  - Create a blank repository on GitHub that contains the outlined ***README.md***, ***.gitignore***, and ***LICENSE*** files so you can use this as a template for future projects
  - This will be crucial for others to understand what your project is about, as well as future you!
- **Commit** often and **commit** early
  - Provide descriptive **commit** messages so you can quickly understand what you did when reading the Git history
  - It's better to have too many **commits** than too few
    - You can always **squash** them later
- Make sure your **commits** relate to specific and distinct code changes
  - This makes it easier to understand what you did and why you did it, as well as reverting changes if necessary
  - You can **stage** files line-by-line if necessary to ensure you are only **committing** the changes that are related to the goal of the **commit** and described in the **commit** message
- Make use of `git amend` to facilitate frequent **commits**
  - This will allow you to develop quickly and then decide what you want to keep in the **commit** later
  - Not every change needs to be a separate **commit**, so repeatedly **amending** to the previous **commit** is a good way to prototype an idea without cluttering the Git history or risking losing your work
- Create **short-lived feature branches** for each new feature

  - This will keep your **main** production branch clean and working for your collaborators, provide a clear descriptive structure to your development, and make it easier to roll back changes if necessary, without concern of breaking working code on the **main** branch

- Use **pull requests** to merge your **feature branches** into the **main** branch
  - This will allow you to get feedback from your collaborators before merging your changes into the **main** branch
  - It's necessary to get experience with **pull requests** as they are a common way to contribute to open source projects
  - Delete your **feature branches** after merging them into the **main** branch
- Use **GitHub issues** to track bugs and feature requests
  - We haven't covered this, but it's a good idea to get familiar with this feature and the documentation and general ideas are easy to understand
  - This will allow you to keep a self-contained project, rather than having to try and link to external issues in a different system
  - Learn how to reference issues in your **commit** messages and **pull requests** to automatically close issues when feature branches are **merged** after completion
- Routinely update the ***README.md*** file to ensure it is up to date with the current state of the project

Part III

Troubleshooting Examples

12 Deleting Branches

As described earlier, you should be using short-lived feature branches, which means that you will be deleting branches quite often. Sometimes, you will end up getting some error messages when you go through the process outlined here.

This page will help you navigate some of the errors you might encounter.

12.1 Fatal Reference

```
fatal: couldn't find remote ref refs/heads/branch-name
```

Sometimes you will delete a branch from both the remote and local repositories, but you will get an error message when you later try to pull or push to the remote repository. For Git to connect local and remote branches, it uses references that are stored in a number of files. The first place to check is in the ***.git/config*** file. If you see a reference to the branch you deleted, you can remove it from the file.

```
cat .git/config

>   ...
    [branch "old-branch-name"]
            remote = origin
            merge = refs/heads/old-branch-name
    ...
```

13 Miscellaneous

This section contains miscellaneous troubleshooting information that does not have a natural home elsewhere, at the moment.

13.1 Partial Push

Sometimes you make a lot of **commits** locally that you haven't pushed to the **remote**. If this happens, occasionally, you may want to push only a subset of the commits to the **remote**. This could be because you're not sure that all of the commits are complete enough to be shared, and you may want to edit them later on. It could also be that you tried a regular push, but it failed, and pushing only a subset of the commits is a way to troubleshoot the problem.

```
git push origin commit-id:branch-name
```

13.2 Forking/Duplicating Your Own Repository

It's not possible to fork your own repository on GitHub. However, you can create a copy of it. The steps are as follows:

- Create a new repository on GitHub.
- Clone your original repository to your local machine.
- Add the new repository as a **remote** (**origin**).
- Add the old repository as the **upstream** (i.e. the origin of the material)
  - This allows you to track changes to the original repository, and pull them into your new repository.
- **Push** the local copy of the new repository to the new **remote**.
- Check for other branches in the original repository
  - Checkout each branch in turn
  - Push each branch to the new **remote**
  - Repeat for all branches

```
git clone git@github.com:arnold-c/psu-intro-to-git.git psu-intro-to-git-copy
cd psu-intro-to-git-copy
git remote set-url origin git@github.com:arnold-c/psu-intro-to-git-copy.git
git remote add upstream git@github.com:arnold-c/psu-intro-to-git.git
git push -u origin main
git branch -a # list all branches in the original (remote) repository
# git checkout other-branch-name
# git push -u origin other-branch-name
```

14 TODOs

This is a list of topics I want to cover when I get the time. They will generally be more advanced, as most of the basics should be covered by now. If you have any suggestions, please let me know.

- ☐ Moving away from GitKraken
  - ☐ LazyGit TUI
  - ☐ GitHub CLI
  - ☐ Integrating Git into NeoVim
  - – I want to show alternative, faster, and importantly, free ways to interact with Git. GitKraken is a great tool, but learning to use the terminal and command line clients allows for faster interaction with Git. GitKraken is also only free while you are a student and have the GitHub student development plan.
- ☐ `git stash`
- ☐ `git squash`
- ☐ `git rebase -i`
  - – Interactive rebases are incredibly powerful mechanisms to correct past mistakes and conflicts, but they introduce many more advanced concepts.
- ☐ `git cherry-pick`
  - – Cherry picking commits is useful when you accidentally start work on the wrong branch and want to move it across to the correct feature branch.
- ☐ `git bisect`
  - – When a bug is introduced and discovered at a later time, and you need to find what commit caused the issue, `git bisect` can be useful.
- ☐ Alternative branching strategies
  - ☐ **tags** and **releases**
  - ☐ GitHub actions
  - – Sometimes it is useful to have a long running **dev** branch that is developed against (i.e., feature branches come off the **dev** branch), and **dev** is merged into the **main** branch for releases, usually coinciding with automated GitHub action tests. For science, a release could be when a block of work has been completed that represents

the initial paper submission, and subsequent submissions/updates reflect different **releases**

References