An Introduction to Git and GitHub

A Practical Guide for Busy Researchers

Callum Arnold

3/20/23

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 mirror much 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 more in-depth 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!

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 without a bias and focus on R and R-studio, 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.

Pre-requisites

There are some pre-requisite tasks to get set up ahead of the workshop.

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

TOC

After getting setup, the workshop will cover the following sections:

1. An overview of how Git works
2. Making your first Git repository

3. Git mechanics - how to actually use Git
4. Git branching
5. Recommended practices

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, 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.

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

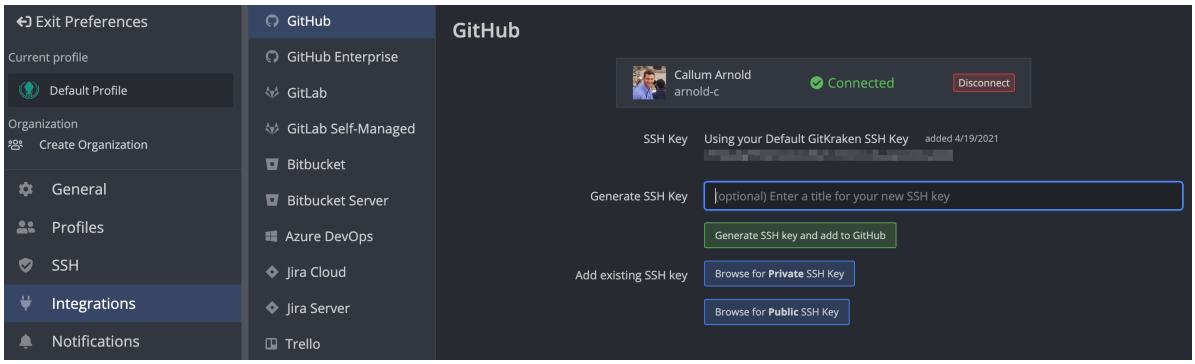
Assuming you use want to use GitKraken, you can connect it to GitHub to make for a much smoother experience. The full details are laid out in the GitKraken documentation, but in summary:

- 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

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 the changes in a repository 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.

> **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

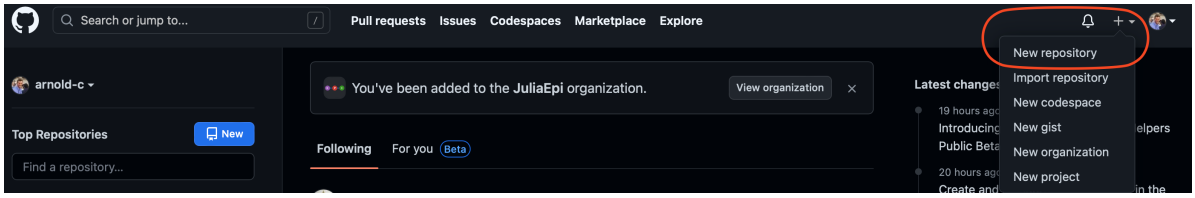
In general, there are two ways to make a Git repo. You can start with a local repo and **push** it to GitHub, or you can start with a GitHub repo and **clone** it to your local machine. If you don't have any existing code, it's marginally easier to start with GitHub, which is why we'll start with this 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.

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.

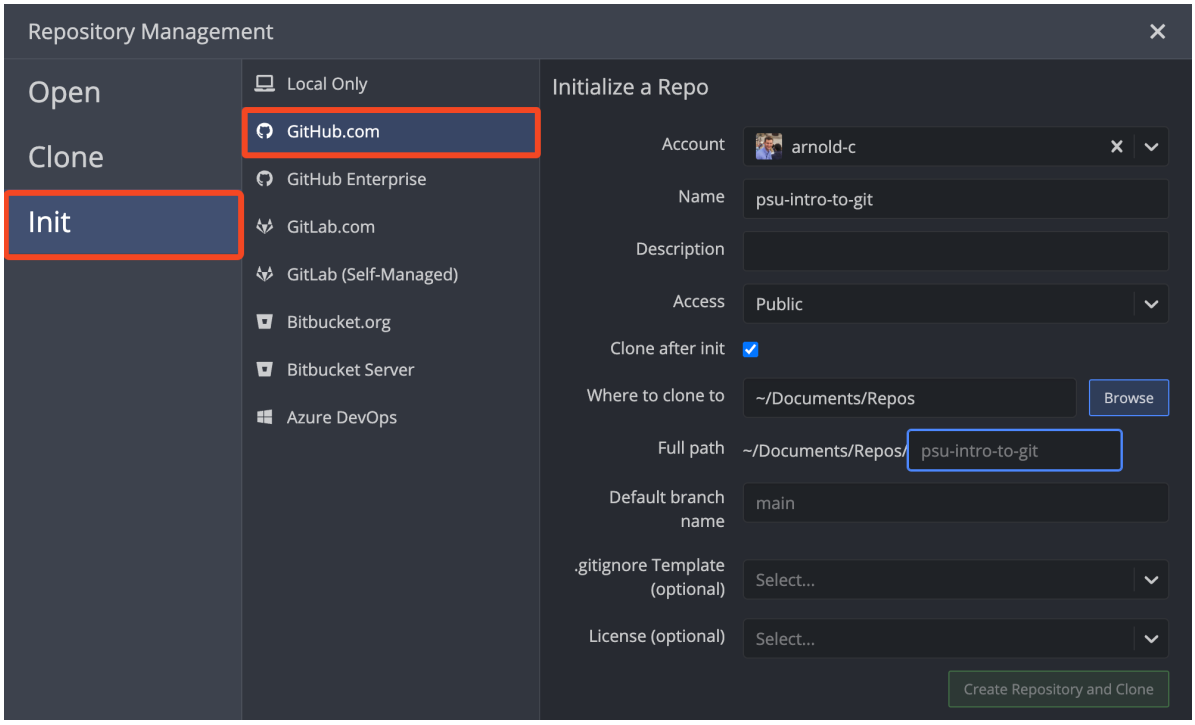
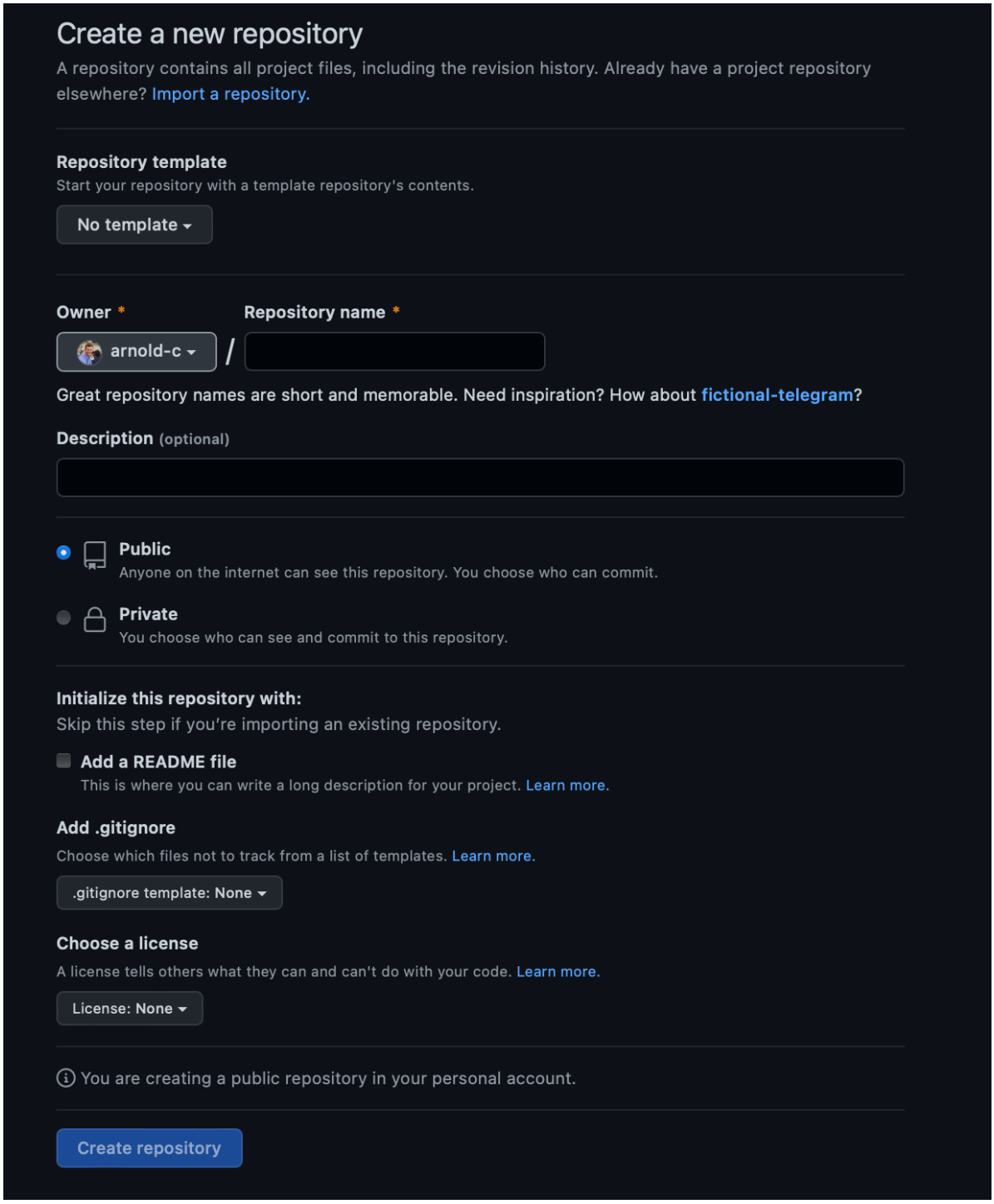


Figure 6.1: Pressing **cmd+I/ctrl+I** brings up the *"Init"* window


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

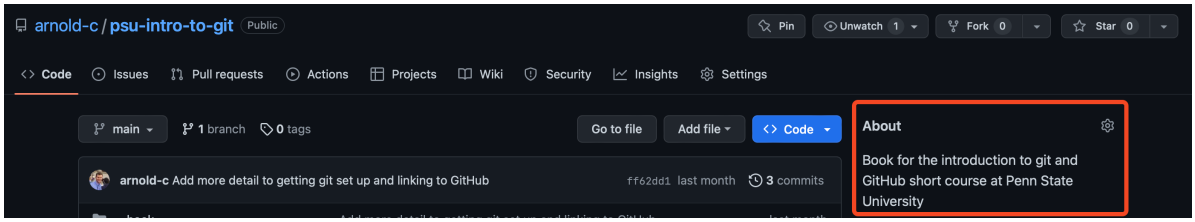
When choosing a repository name, you'll want something short, memorable, and descriptive of the project that you're working on. Ideally, you should follow Jenny Bryan's excellent guidelines when it comes to file and repository naming (after all, a repository is just a project folder), but as an overview:

- 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.

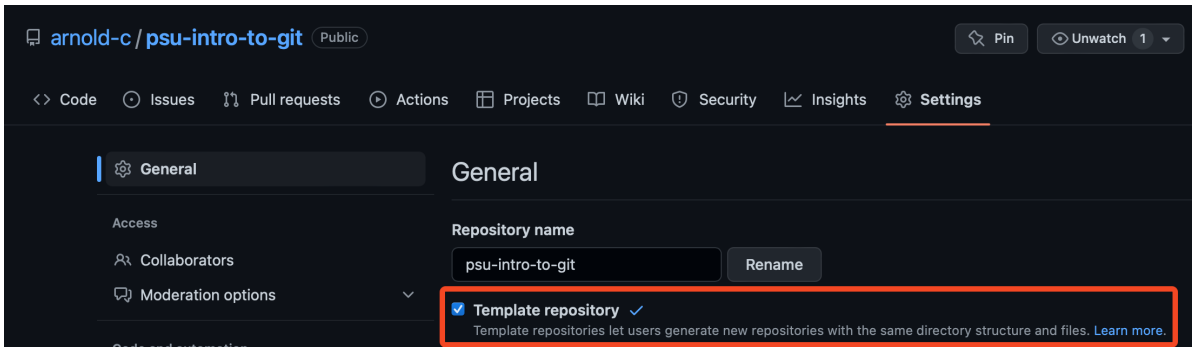
6.1.7 License

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

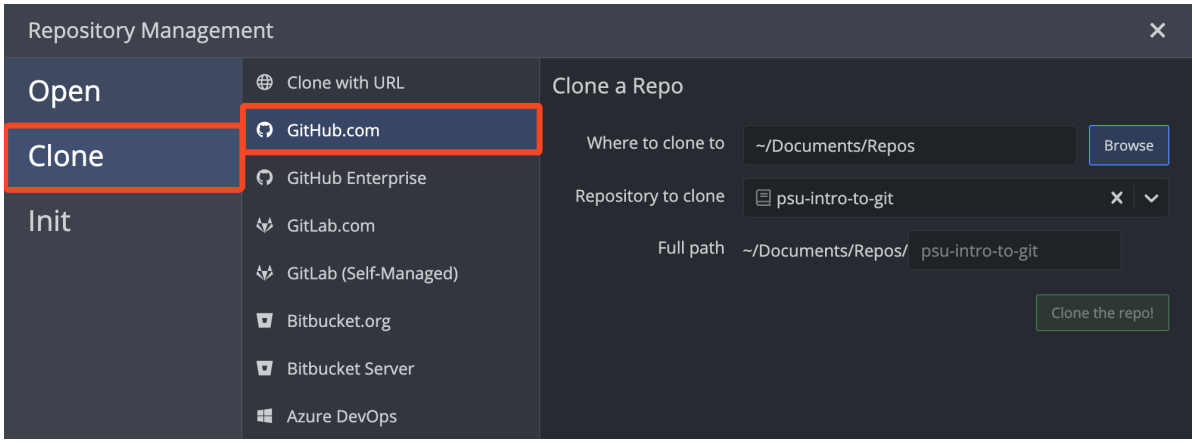
Now comes the fun part - getting your project set up on your computer where you can do your work. Thankfully, GitHub makes this incredibly easy to do, particularly if you have a Git client like GitKraken installed.

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.

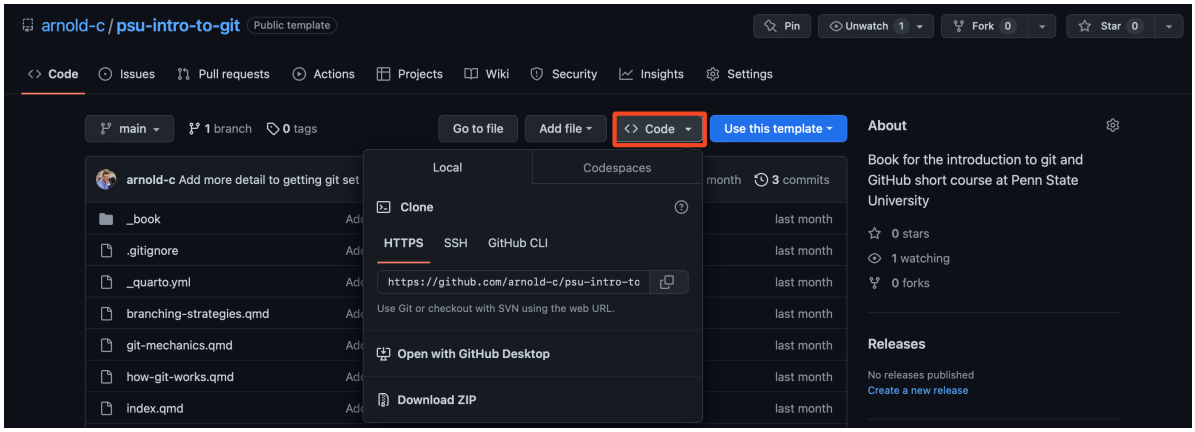


If you have connected GitKraken to GitHub, you can search through your GitHub repositories, otherwise you can paste in the HTTPS or SSH urls in the *"Clone with URL"* section.

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!

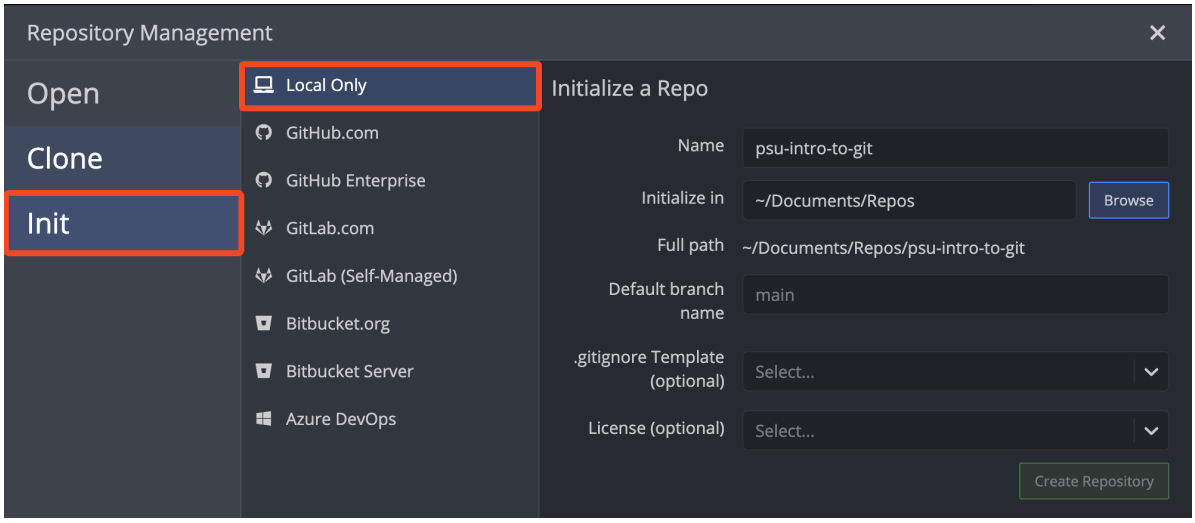
6.2 Local First

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

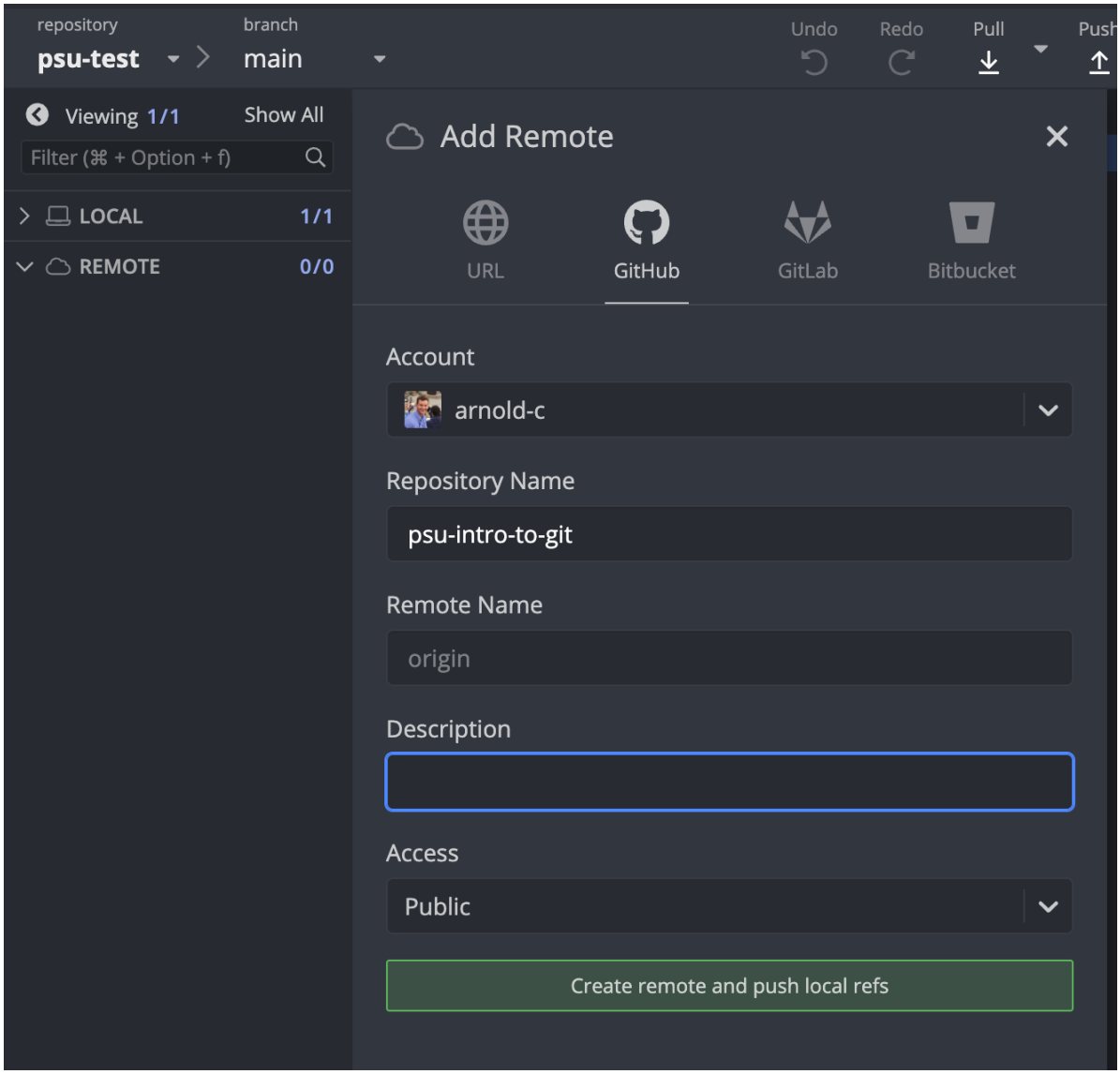
If you have a Git client installed, such as GitKraken, it's not too hard to turn an existing directory into a Git repository, before `pushing` to GitHub. First, open GitKraken and enter **cmd+I/ctrl+I** to open up the *"Init"* window. From here, navigate to *"Local Only"*, which tells GitKraken that 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**
- **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.
- **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**!
- **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 this is 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.

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**.

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 detai

### Contact

### Acknowledgements
```

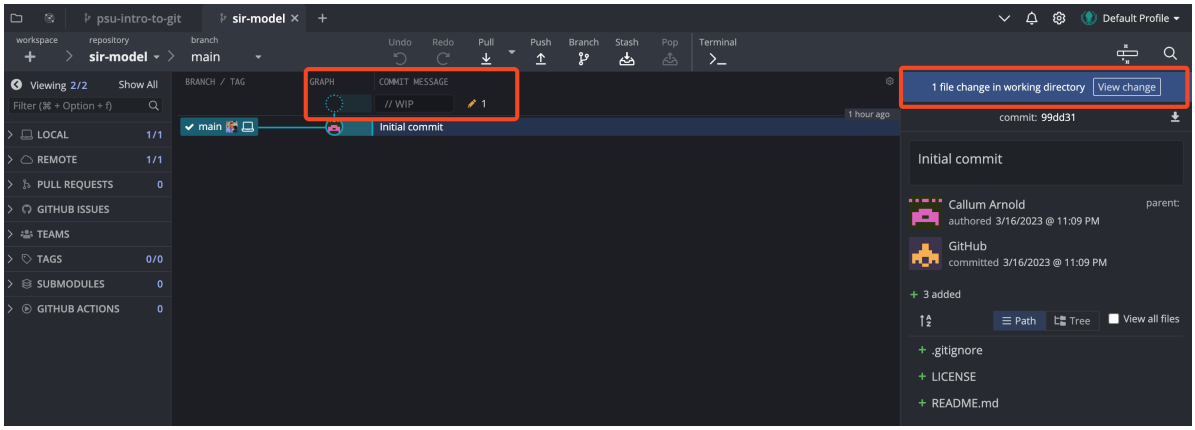
> **Note**
>
> The `##` and `###` are used to denote different levels of headings in Markdown. I prefer to use a level-2 heading for the title of the repository, and level-3 headings for the different sections, as I find it makes the README easier to read; level-1 headings are quite large.

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.

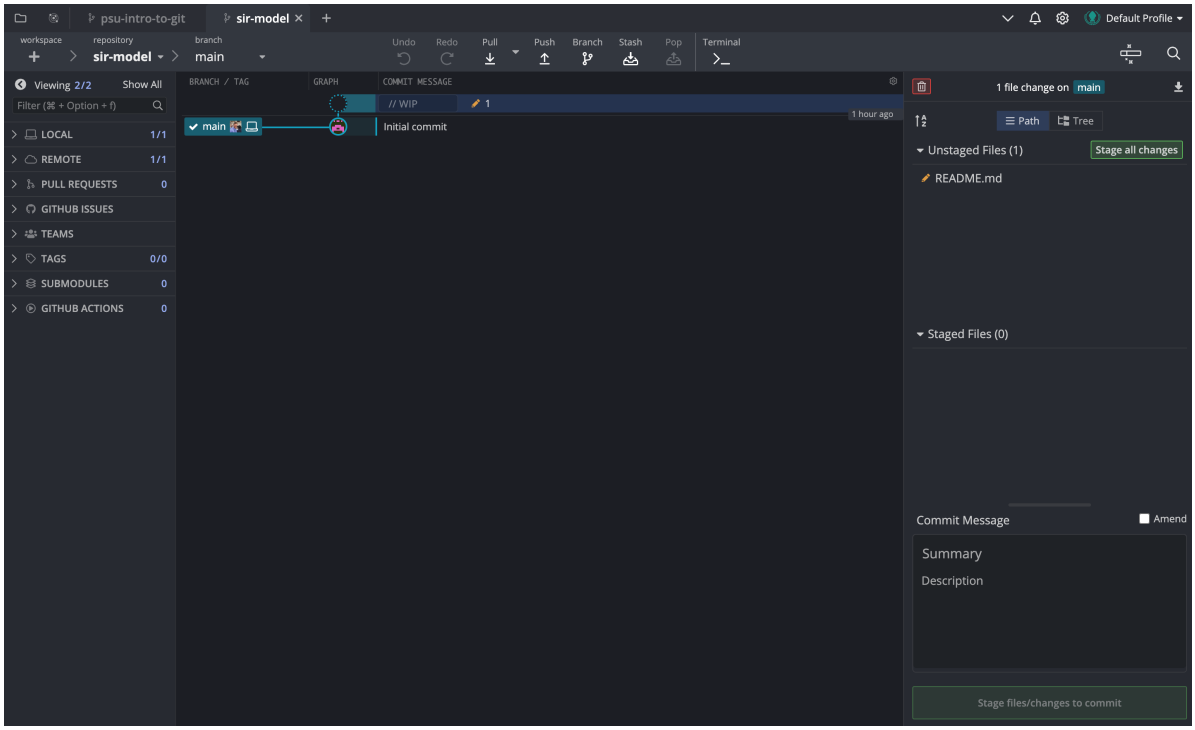
8.2.1 Creating The First Commit

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"*.

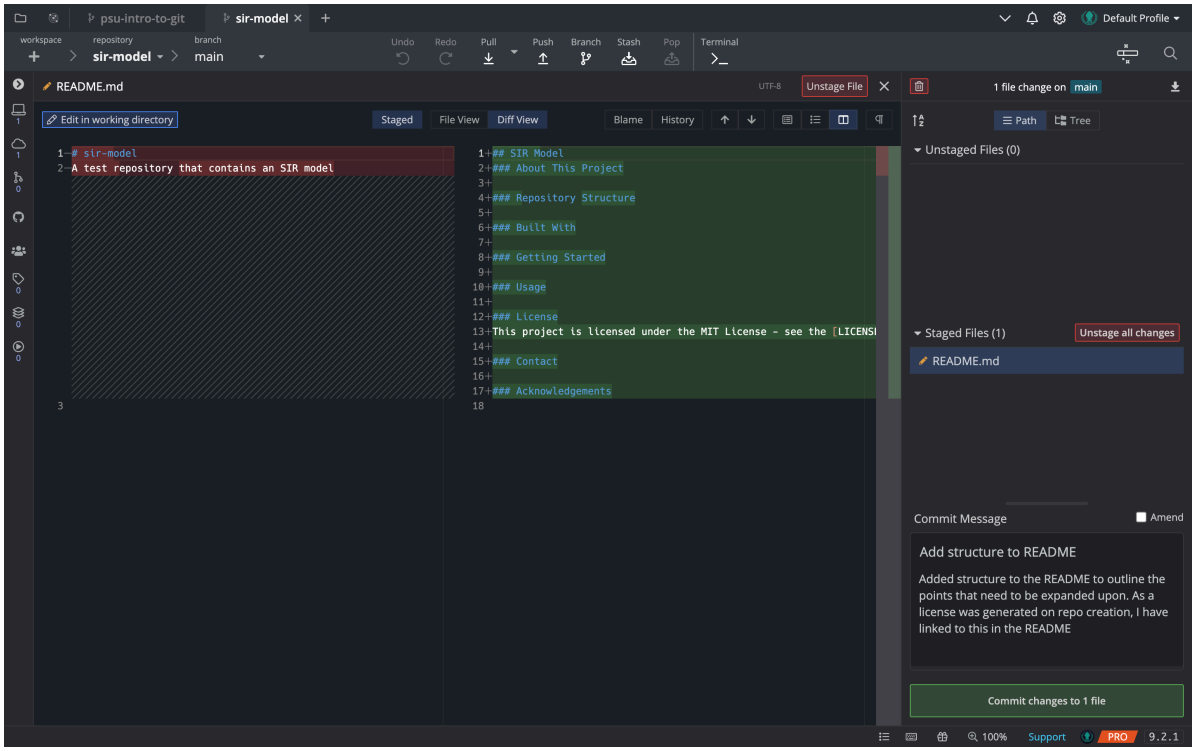


Clicking on either of these sections will present you will the below window, which shows you the changes that have been made to the file. You will also notice that the changes are **unstaged**, which means they will not be included in the next commit.



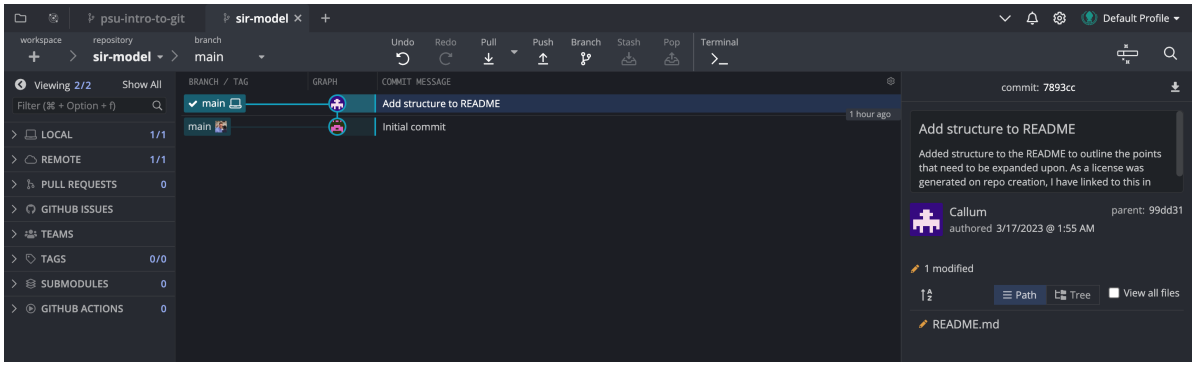
Clicking on the ***README.md*** file will open up the file **diff** in the GitKraken editor, and

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.

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 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*** in that folder. Copy the below code into that file.

```
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
  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")
```

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

8.4 Updating the README

8.4.1 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.

8.4.2 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.

```
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

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 usua
```

8.4.3 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.

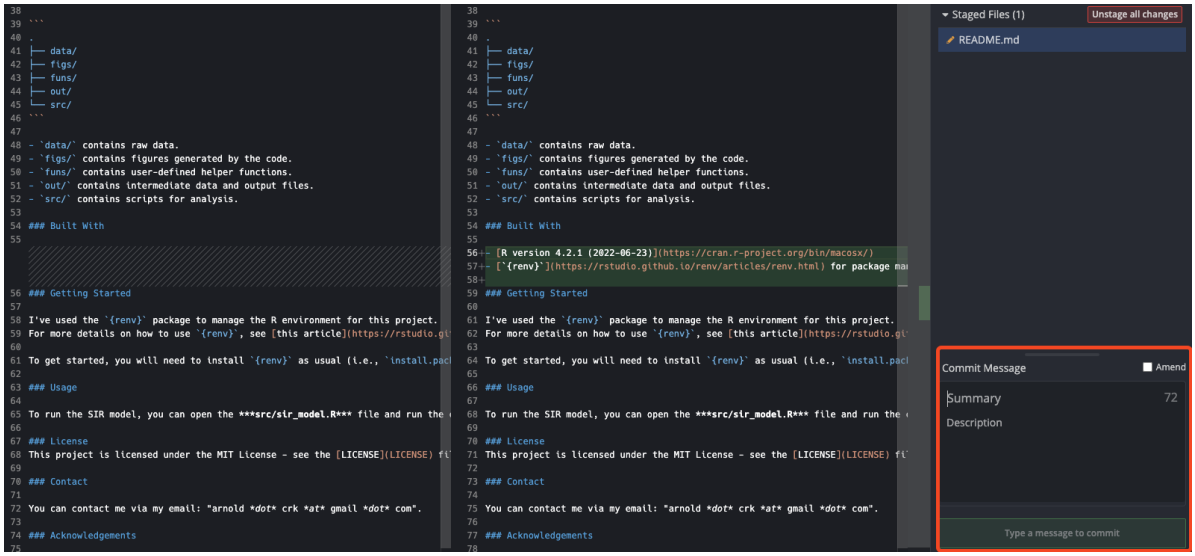
We want to add the following text to the "Built With" section of the README.

```
- [R 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
```

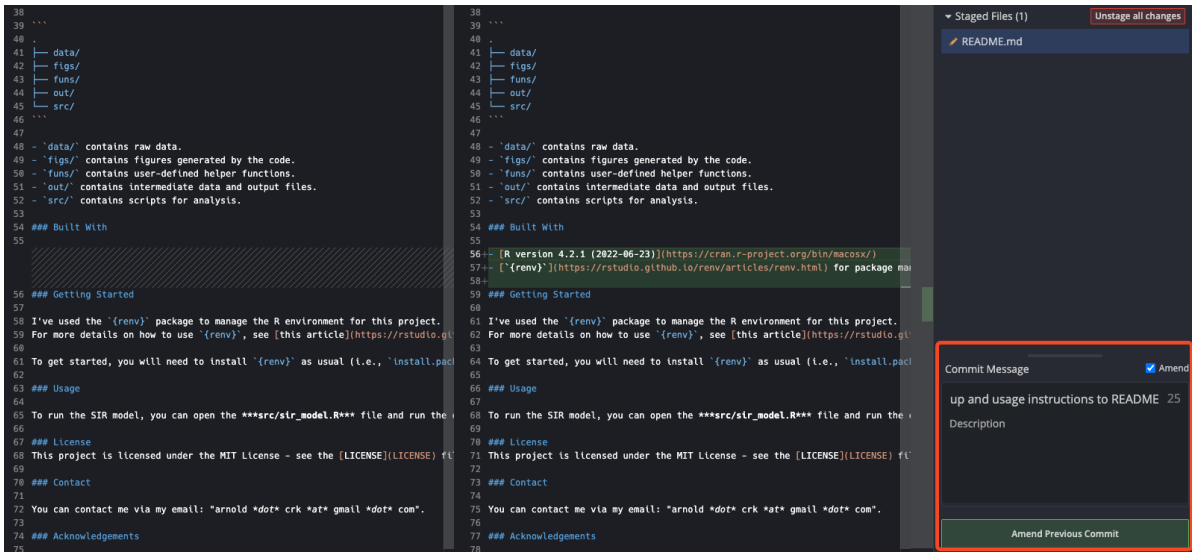
> ⚠ Warning
>
> We can only **amend staged** changes to the most recent **commit**, and only if we haven't **pushed** it to GitHub yet. If we've already **pushed** the **commit**, we'll need to make a new **commit**. If you do not make a new **commit** and try and just **amend** it, you'll get an error 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**.

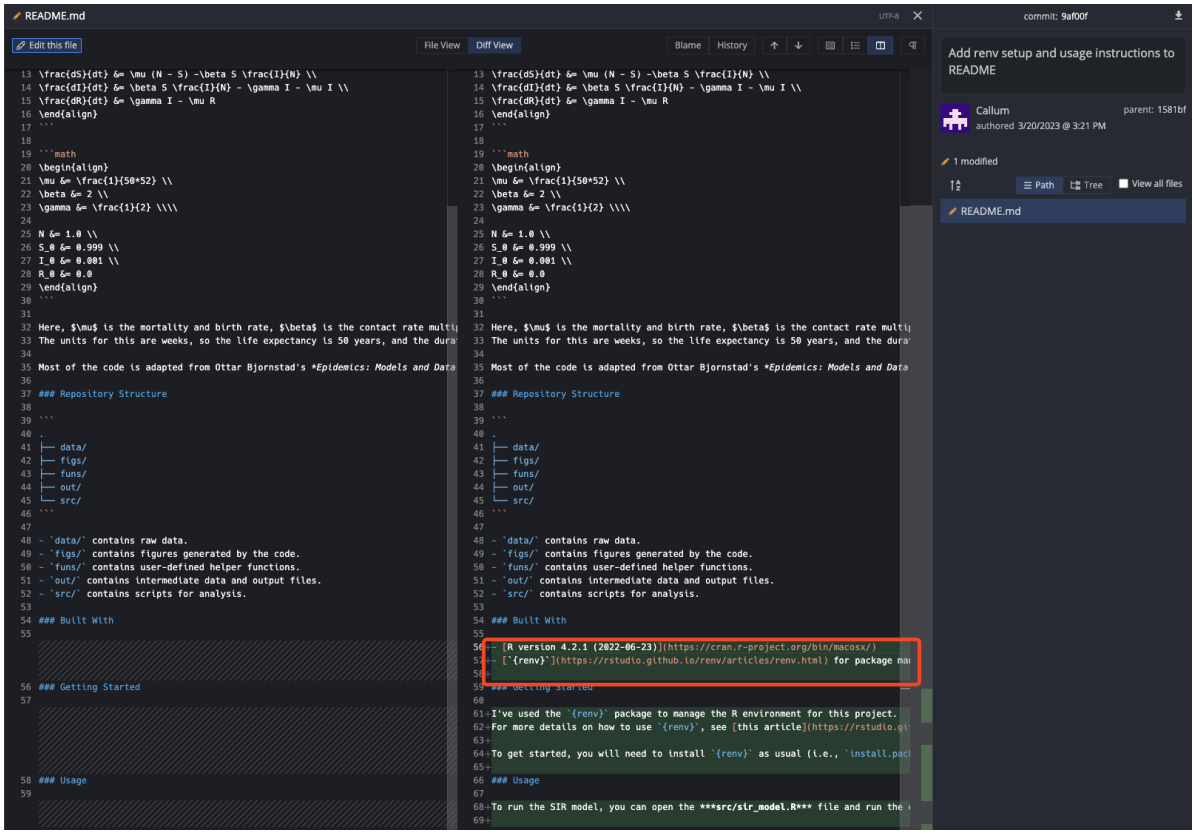
As per usual, when we save and **stage** the changes, we'll see the **diff** in the GitKraken interface, as below.



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.



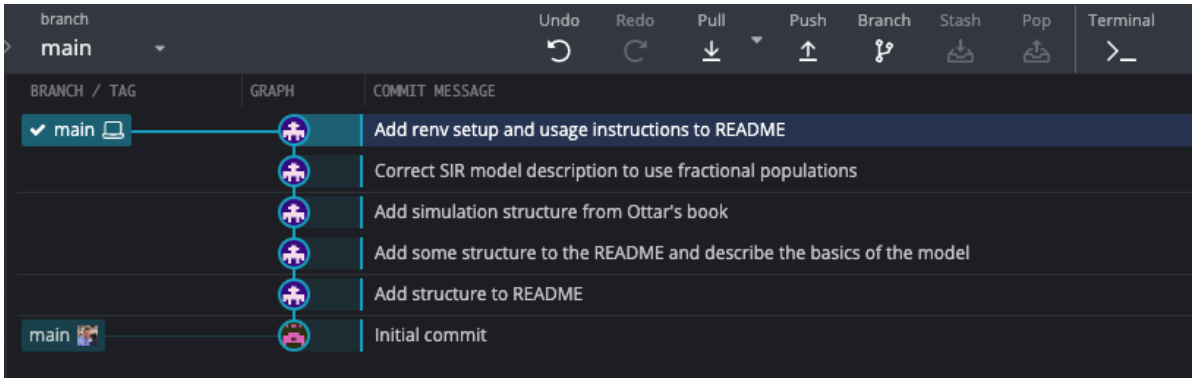
Once we've **amended** the **commit**, can examine the Git history and see that there is only one **commit** that refers to the changes we've made to the ***README.md*** file. Equally, if we look at the **diff** of this **commit**, we can see that our **amended** changes are included.



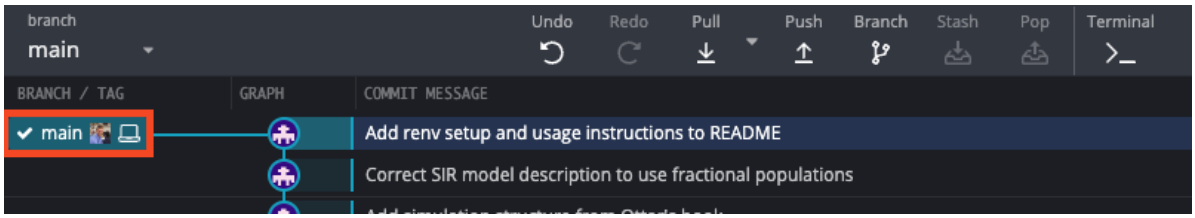
8.5 Our First Push

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.

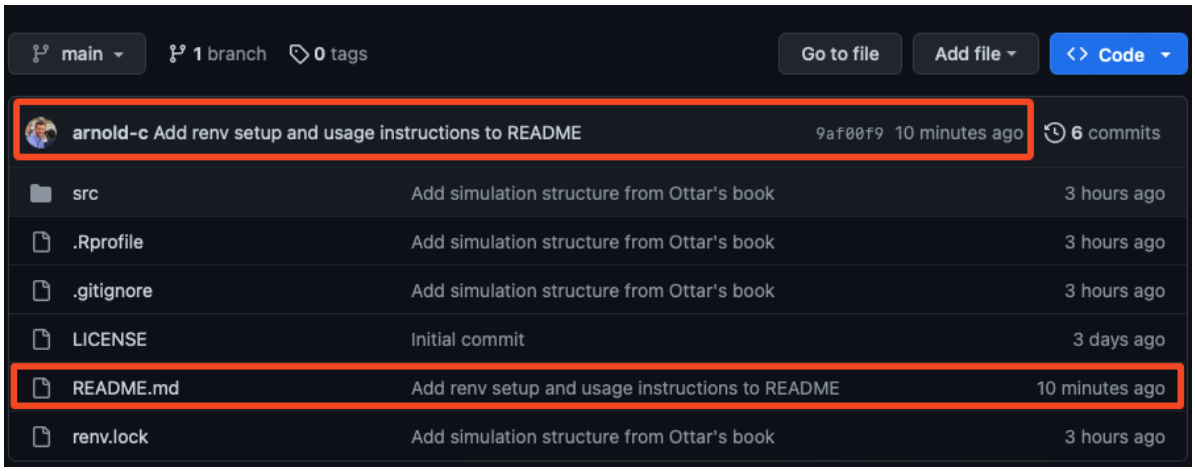
On the GitKraken client, there is a button in the top toolbar that is an up arrow, with the description "Push". When we hover over it, it shows the text "Push to origin/main".



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.

8.6 Collaborating on a Project

- Add me as a collaborator
- I'll make a change
- See how the changes are reflected in the Git history in GitHub, but not in your local repository
- Pull the changes

9 Branching Strategies

10 Recommended Practices

References