Chapter 5

Optimizing Your Reproducible Neuroimaging Workflow with Git

Mélanie Garcia and Clare Kelly

Abstract

As a neuroimager working with open-source software and tools, you will quickly become familiar with the website GitHub, which is a (for profit) platform for storing, managing, and sharing code, software, and projects. Many of the open-source tools discussed in this book are hosted on GitHub. Although people are generally very familiar with GitHub (or GitLab), they are often less familiar with its foundation—Git. Better understanding Git will help you better manage your own projects *and* will also help you to better understand GitHub and how to use it optimally. In this chapter, we will first explain Git and how to use it, then we will turn to GitHub. A worked example—a clustering analysis—with open access neuroimaging data is provided, demonstrating utilities such as version control, branching, and conflict resolution.

Key words Neuroimaging, Git, GitHub, Version control, Reproducibility

1 What Is Git?

> "Git is a free and open source distributed version control system designed to handle everything from small to very large projects with speed and efficiency." (https://git-scm.com/)

To understand what Git [*see* Resources] is, we first need to understand the concept of *version control* [*see* Glossary]. At its most basic, *version control* means tracking and managing changes to your code (scripts) and projects. Git will help you to do this in a structured and systematic way, which helps you to manage and share your code with others. The Git version control system means that parts of your project can be developed or edited, while your work is also protected from unwise or undesirable changes through the possibility of reverting to an earlier version. These features of Git facilitate the collaborative development of tools. Git is widely used in academia but also in most business environ-

Robert Whelan and Hervé Lemaître (eds.), *Methods for Analyzing Large Neuroimaging Datasets*, Neuromethods, vol. 218, https://doi.org/10.1007/978-1-0716-4260-3_5,

ments that involve IT programming. While using Git may seem cumbersome or complicated at first, the habits that it establishes, the version control system it implements, and the ease of collaboration it enables will, in the long run, make the investment worth it.

2 Version Control with Git

Git and version control are best explained through an example. As a prerequisite, you need to install Git (https://git-scm.com/book/en/v2/Getting-Started-Installing-Git) on your machine. Below, we work through one example, providing command lines you can type into a terminal on your machine. In this example, we will perform a clustering analysis on a file from the derivatives folder in the AOMIC-PIOP2 dataset (https://openneuro.org/datasets/ds002790/versions/2.0.0), using Python. To follow the tutorial, you will need to download the file *derivatives/fs_stats/data-cortical_type-aparc_measure-area_hemi-lh.tsv* from the Open Neuro repository [*see* Chapter 2].

2.1 Create a Git Repository for Your Project

If you want to version control a project, you will need to create a repository [*see* Glossary] (which is simply a directory or folder; often called a "repo") that will contain all the code related to the project.

Let's call this folder "my-analysis". Note that when using the command line, it is best to avoid spaces or special characters when naming folders.

```
mkdir my-analysis
```

Next, change directory into this folder.

```
cd my-analysis
```

Now, initialize a Git repository associated with this folder, by running the command:

```
git init
```

This creates a .git directory inside the folder "my-analysis". The .git directory is the only difference between a Git repository and an ordinary folder.

Note
Be very careful! Deleting the .git folder will mean that all the files in the folder "my-analysis" become *unversioned*—you will lose the history of all changes to those files.

2.2 Basic Configuration of the Git Repository

First, configure Git by telling it who you are:

```
git config --global user.name "Your Name"

git config --global user.email your.email@example.com
```

Note
In the first command, the inverted commas are required. For the second command, you should use the same email address used to set up your github account (this will be explained later). These steps have a very practical benefit in that they allow you to track *who* coded each part of a collaborative project.

2.3 Check the Status of the Git Repository

You should check the status of the Git repository frequently, using the command:

```
git status
```

In this example, this command should return:

```
On branch master


No commits yet


nothing to commit (create/copy files and use "git add" to track)
```

The first line indicates that you are on the branch "master". We will explain the notion of a branch later in this tutorial. The second line tells us that we have not yet begun to track ("commit") any files. The final line tells us that we do not yet have any files in the repository that we can track—there is "nothing to commit".

Users often prefer to rename the "master" branch as "main", in recognition of the fact that technology terminology such as "master" has its origins in slavery and colonialism. GitHub recently made the same change across its platform (https://github.com/github/renaming). This change cannot be done until you have made your first commit, however, so we will return to this below.

2.4 Create a File

Let's create a notebook to visualize the data, called visualization.ipynb. The notebook is available here for download (https://github.com/garciaml/my-analysis). Download the whole folder as a zipped folder (*see* Fig. 1)—then unzip it and move the file **visualization.ipynb** into the "my-analysis" folder. To run this notebook, you will need to have installed python and the libraries pandas, matplotlib and seaborn [*see* Resources].

You also need to move the data downloaded from the AOMIC-PIOP2 ("derivatives fs_stats data-cortical_type-aparc_measure-area_hemi-lh.tsv") into your "my-analysis" folder.

Next, verify the status of the Git repository again:

```
git status
```

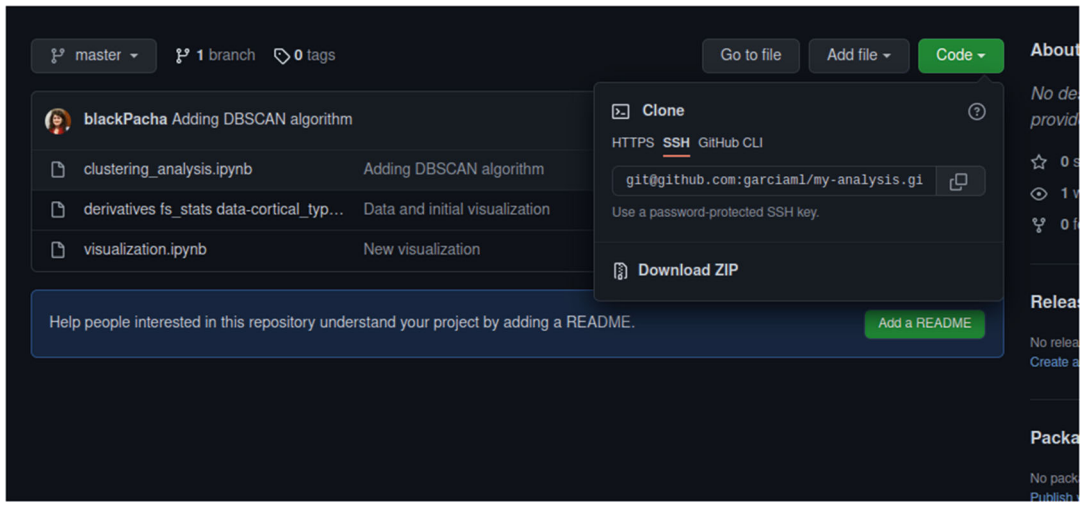

Fig. 1 Repository containing the files needed for this tutorial. You can download these files as a zipped folder

This command should return:

```
On branch master

No commits yet

Untracked files:
  (use "git add <file>..." to include in what will be committed)
    .ipynb_checkpoints/
    derivatives fs_stats data-cortical_type-aparc_measure-area_hemi-lh.tsv
    visualization.ipynb

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

Now that you have some files inside your my-analysis folder, the *git status* command returns a lot more information. It now has a section, "Untracked files", which lists two files:

derivatives fs_stats data-cortical_type-aparc_measure-area_hemi-lh.tsv

and visualization.ipynb. These files are untracked, which means that no versioning has yet been applied to these files. The directory ".ipynb_checkpoints/" is a hidden folder containing cache memory related to the notebook. It should not be tracked.

2.5 Staging Files

The first step in versioning files is called *staging*—which involves letting Git know which files we intend to version. Stage the files "derivatives fs_stats data-cortical_type-aparc_measure-area_hemi-lh.tsv" and "visualization.ipynb" using the following command:

```
git add derivatives fs_stats data-cortical_type-aparc_measure-area_hemi-lh.tsv visualization.ipynb
```

This operation means that you intend to track these files, but have not yet started to save versions of them. If needed, you can go back and remove these files from being staged/tracked with no impact on the Git repository, using the command *git restore --staged <file>*.

You can verify the status with:

```
git status
```

The outcome should have changed now, and you should see the two stage files now included in the list of files to be committed.

2.6 Committing Files

In Git, a "commit" is a snapshot of the staged changes to the project. Each time you perform a "commit", you create a *version* of your project. Git keeps these versions safe, so we can always return to an earlier version of our project if needed.

Let's create your first commit. First, check the history of commits of the project.

```
git log
```

Because this is a new project and no commits have yet been made, it should return:

```
fatal: your current branch 'main' does not have any commits yet
```

One way to commit is to run:

```
git commit
```

This command will lead to a window where you can type a brief message related to the commit. The message should capture what the commit does—what changes the commit makes to your repository (here's a guide to writing Git commit messages: https://cbea.ms/git-commit/). Next, save the changes and exit. This will have created a new commit in the history of our Git repository.

Another way to commit is to use the command-line:

```
git commit -m "Data and initial visualization"
```

The part inside the inverted commas (quotes) is the commit message.

We can visualize this commit and obtain its identifier by running:

```
git log
```

Now that you have made your first commit, you can rename the "master" branch as "main" using the command:

```
git branch -m master main
```

2.7 Create and Track a New File in Your Project

Let's create a notebook that will contain our clustering analysis, called "clustering_analysis.ipynb". You'll find this notebook in the folder you previously downloaded from the repository. Move it into the "my-analysis" folder.

Stage and commit it:

```
git add clustering_analysis.ipynb

git commit clustering_analysis.ipynb -m "First clustering: PCA + k-Means"
```

Note
Your commit command is different this time, in that you specified the name of the file you wanted to commit. This may be necessary when you have several files that are staged but you only want to commit one of these.

Now you can make changes and edits to your project files, staging and committing those changes as often as needed (*see* Fig. 2).

2.8 Remove Modifications in a File Before Committing

Sometimes you will want to undo changes you have made to a file. For example, let's modify the file "visualization.ipynb" by adding a cell and writing some code inside. Save the changes. Next, run:

```
git status
```

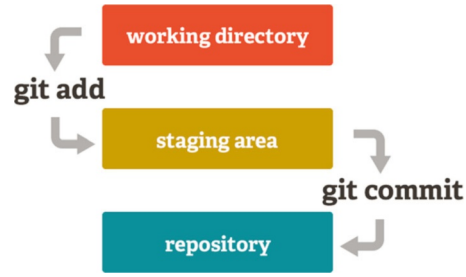

Fig. 2 Visualizing the staging and committing process

The name of the file should be in the section "Changes not staged for commit":

Now, let's say you regret the change you made—they didn't produce the output expected and you want to revert to an earlier version of your notebook.

To do this, run:

```
git restore visualization.ipynb
```

Open the notebook in Jupyter or another interface. You should find that you have restored the last version of the file before you made those modifications—the changes you made to the file have disappeared.

2.9 Check a Specific Old Version of a File in the Git Repository

Let's say you have developed a project quite a bit, but you'd like to look at a specific version of one of your files—for example, the first version of the file "visualization.ipynb". You can do this by:

- Finding the ID of the commit corresponding to the version of the project you want to check, by running: git log --oneline.
- The need for informative commit messages should now be clear! You need to be able to identify the commit containing the version of the project that you want. Here, you want the first commit, "Data and initial visualization", which the log indicates is commit 62901c0.
- To access this version, use the command `git checkout 62901c0`.
- You can now open the notebook "visualization.ipynb" in Jupyter or another editor. This is the old version of your file. Now, things can get complicated if you want to modify *this* version of the file. In that case, it is recommended that you start a *new branch* of your project, and commit any new changes within this

new branch. Otherwise you will run into trouble when you return to the most recent version of the project in the branch main. We will look at how to build a new branch in the next section.

- Finally, to return to the most recent version of your project in the branch main, run the command line: git checkout main.
- You can check that all the recent changes in "visualization.ipynb" are still there by opening that notebook.

2.10 Developing in Branches

Let's say you want to add a new type of algorithm to our analysis but are not sure it will work. One way to do this safely is to create a new "branch" of your project where you can develop this new analysis feature, and make commits, but without modifying the stable version of your project (which is by default on a branch named "main").

First, let's see what branches there currently are in your project:

```
git branch
```

Since we have not yet created any branches, we should see just one listed:

```
* main
```

You will now create a new branch named "dbscan" because this is the new feature to add to the notebook "clustering_analysis.ipynb".

```
git branch dbscan
```

Now, you need to go to this branch, so the work you do is staged and committed to the dbscan branch and not the main:

```
git checkout dbscan
```

You can verify that you are in the dbscan branch by running *git branch* (you will see an asterisk in front of the current branch).

```
* dbscan      main
```

Modify the file clustering_analysis.ipynb. You can add a cell with these lines of code:

```
# Perform a k-Median

clustering = cluster.DBSCAN(eps=10000, min_samples=3).fit(X)

print(clustering.labels_)

# Let's visualize the clusters

plt.figure(figsize=(16,9))

plt.scatter(X[:, 0], X[:, 1], c=clustering.labels_)

plt.figure()
```

Save your changes in the Git repository within the current branch:

```
git add clustering_analysis.ipynb

git commit -m "Adding DBSCAN algorithm"
```

By working within a branch, you can make sure that the changes you make are what you want. Once you have a version of a new feature that works and that you are fully satisfied with, you can *merge* the branch with the branch main to integrate our feature in the stable version of our project.

First, go back to the branch master:

```
git checkout main
```

Next, merge the branch dbscan:

```
git merge dbscan
```

Finally, in order to keep our Git repository clean and refined, you should delete the branch dbscan since we will not use it later in the project:

```
git branch -d dbscan
```

Fig. 3 Branches in Git projects

Sometimes you will need to manage conflicts while merging branches. Conflicts arise due to the line-by-line differences between two versions of the same file. To resolve conflicts, you can edit files in their respective branches and make the different versions compatible. Git will help in this task by showing us the conflicting parts of the files. For example:

```
<<<<<<< HEAD     What is written in the version in the branch you want to merge files into.
=======
    What is written in the version in the branch where you want to get the version from.
>>>>>>> dbscan
```

Branches (*see* Fig. 3) are particularly useful when you want to work in a collaborative way. Every contributor can create new branches where they can develop new features, and, when agreed, can later merge their branches with the stable version of the project. When you code alone, it is also a safe way to add new features to your projects, without breaking the stable version.

2.11 Link Your Project with a GitHub Repository

GitHub is a platform for storing projects in public or private repositories. It is widely used within the scientific community and uses Git as backbone. It makes it possible to access your projects from anywhere, and to share your projects in an open-source way, thanks to the "public repository" mode.

To work remotely on your laptop on a project that is stored in GitHub, and to keep the state of the project synchronized between your laptop and GitHub, you will need to perform several steps.

First, you'll need a GitHub account. Next, create an ssh connection that will serve as your secure authentication when synchronizing your project with your GitHub repository. You can follow this GitHub tutorial: https://docs.github.com/en/authentication/connecting-to-github-with-ssh/about-ssh

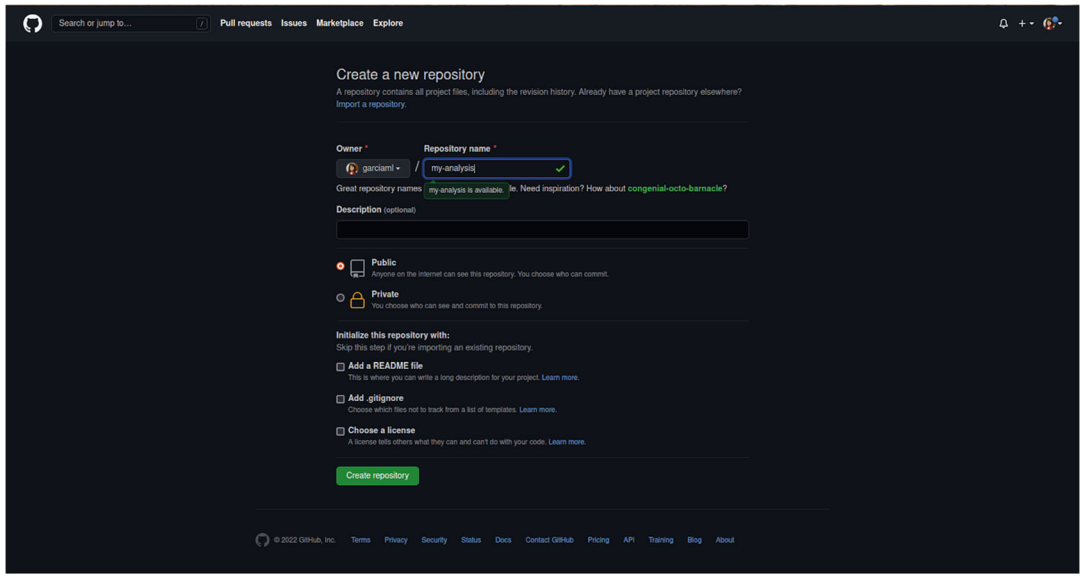

Fig. 4 Creating a GitHub repository

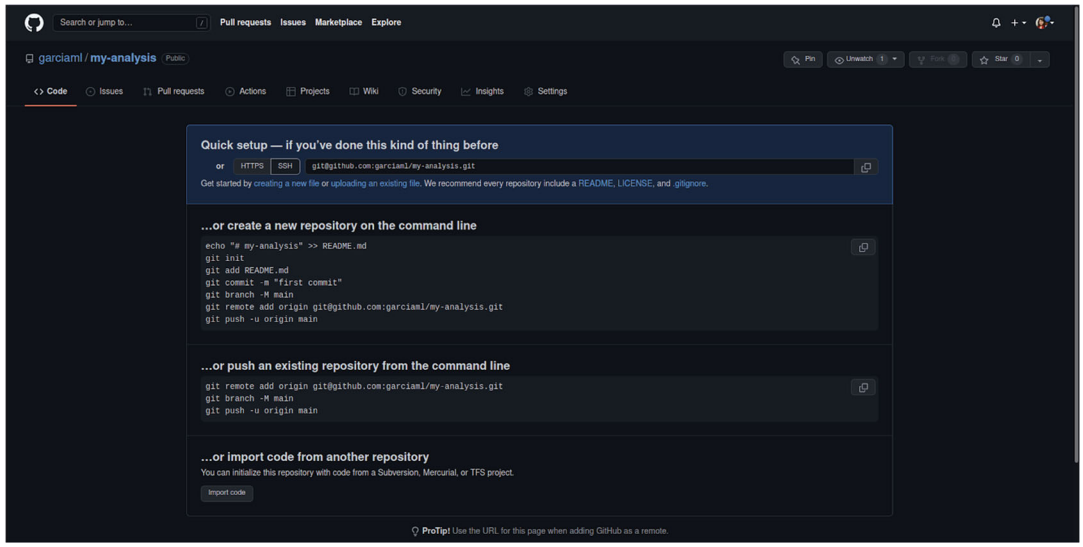

Fig. 5 Quick setup after creating your new repository on GitHub

Next, you can create a new GitHub repository, for instance with the same name: my-analysis (*see* Fig. 4).

To be able to connect your GitHub repository via ssh, select the ssh button. This will display the commands you'll need to launch (*see* Fig. 5).

Once you have run those setup commands, from the my-analysis folder, run:

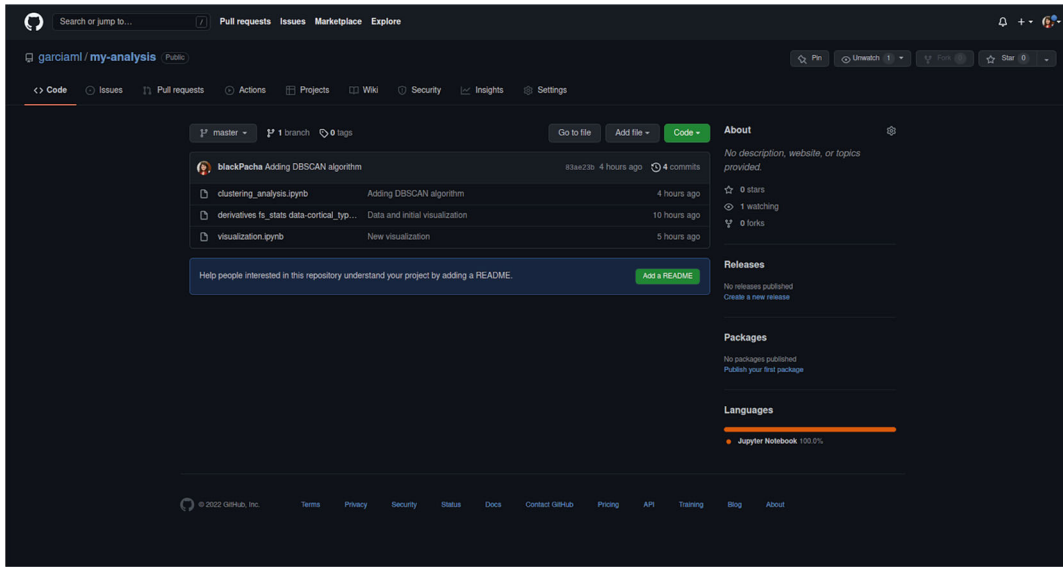

Fig. 6 GitHub repository after *git push*

```
git remote add origin git@github.com:garciaml/my-analysis.
git      git branch -M main
```

Note
Be sure to change git@github.com:garciaml/my-analysis.git to the name of your own GitHub repository!

Next, you can push everything you have coded and tracked in your Git repository to the GitHub repository by running:

```
git push -u origin main
```

If you update your GitHub webpage, you will see your files uploaded on the platform (*see* Fig. 6).

It is very important to regularly push your commits so that you keep an updated version of your project in your GitHub repository.

When you develop a project with other people or when you develop alone but using different machines, you will need to synchronize the version of the project on your machine with the one in

the GitHub repository each time you connect to your machine to keep on working on the project. Do this using the command:

```
git pull
```

2.12 Collaborating with Git and GitHub

Part of what makes Git and GitHub so popular is that they are important tools for collaborative projects. In GitHub, you can add collaborators to your project. These collaborators will then be allowed to directly push their changes to the GitHub repository and to pull the project regularly to stay synchronized with changes committed by others.

Generally, collaborators develop on their own branches before merging with the main branch which is usually the branch of the "deployed" project (i.e., the code on the main branch should run well and correspond to a stable version of the project).

Note
It is useful to have a "develop" branch (*see* Fig. 2) that will allow the collaborators to develop new features of a project until the next stable version that will be merged with the main branch. It also prevents anyone from breaking a stable version of a project with a bad merge (because the stable version remains on the main branch!).

Each collaborator can contribute to developing new features in a project or solving issues, using separate branches. This can make the division of tasks amongst collaborators structured and clear.

It is important to commit often (code that runs well) and push these changes to the GitHub repository to make the evolution of the project visible for all and to avoid overlap or duplication of effort between contributors. There are many ways to establish a productive and efficient work environment. For instance, GitHub Flow [*see* Resources] can help with setting up "*a lightweight, branch-based workflow*" for collaborative projects in neuroscience. Trunk-based-development [*see* Resources] is another type of workflow that can help teams of developers to build a project or a software in an efficient way.

What we have just described is known as the "shared repository model", when team members directly share and collaborate on the source repository with others. Another way to contribute to projects is through the "Fork and pull model". Here, you can create a "fork" from any repository you have access to (e.g., public

repositories, the authors of which you might not know or formally collaborate with). The "fork" is a copy of that repository, which exists on your own GitHub. Now you can develop and push your changes on the fork, *without the permission of the owner of the source repository.* If you want to suggest your changes to the owner of the source repository, you can create a "pull request" and the owner will be able to accept or decline your suggestions for changes to the source repository. The "fork and pull model" is often used for big collaborative projects, when the code is made public and open to be developed by the wider community. You can learn more about collaborative development models from the GitHub documentation (https://docs.github.com/en/pull-requests/collaborating-with-pull-requests/getting-started/about-collaborative-development-models).

2.13 Some Other Practical Tips for Git and GitHub Users

- You can tag specific versions of your project (i.e., commits) with tag names using the command *git tag -a <tag-name> -m "<description>"*.
- You can create a .gitignore file that will contain the names of the untracked files or folders that Git should ignore.
- You can easily clone a GitHub project on your machine by using git clone.
- You can undo committed changes with the command git revert.
- Using the GitHub CLI might be useful (https://cli.github.com/).
- Our example used command-lines directly typed into a terminal. Another option is to use a Graphical User Interface like Sourcetree or GitHub Desktop [*see* Resources]. Here's a list of good Git GUIs (https://dev.to/theme_selection/best-git-gui-clients-for-developers-5023).

2.14 Tutorials to Further Develop Your Git Skills

Kirstie Whitaker's intro: https://kirstiejane.github.io/friendly-github-intro/

The GitHub quickstart: https://docs.github.com/en/get-started/quickstart/hello-world

The official Git tutorial: https://git-scm.com/docs/gittutorial

Videos: https://git-scm.com/videos; https://johnmathews.is/rys-git-tutorial.html

Git Cheat Sheets: https://training.github.com/

2.15 Ten Simple Rules for Taking Advantage of Git and GitHub

Reproduced verbatim here from [1]:

Rule 1: Use GitHub to Track Your Projects

Rule 2: GitHub for Single Users, Teams, and Organizations

Rule 3: Developing and Collaborating on New Features: Branching and Forking

Rule 4: Naming Branches and Commits: Tags and Semantic Versions

Rule 5: Let GitHub Do Some Tasks for You: Integrate

Rule 6: Let GitHub Do More Tasks for You: Automate

Rule 7: Use GitHub to Openly and Collaboratively Discuss, Address, and Close Issues

Rule 8: Make Your Code Easily Citable, and Cite Source Code!

Rule 9: Promote and Discuss Your Projects: Web Page and More

Rule 10: Use GitHub to Be Social: Follow and Watch

Reference

1. Perez-Riverol Y, Gatto L, Wang R, Sachsenberg T, Uszkoreit J, da Leprevost FV, Fufezan C, Ternent T, Eglen SJ, Katz DS, Pollard TJ, Konovalov A, Flight RM, Blin K, Vizcaíno JA (2016) Ten simple rules for taking advantage of git and GitHub. PLoS Comput Biol 12:e1004947. https://doi.org/10.1371/journal.pcbi.1004947