Getting started with Git

In this train, weâ€™ll delve deeper into the practical aspects of using Git for version control. We'll cover the fundamental concepts, from initialising a repository to managing branches.

Learning objectives

By the end of this train, you should be able to:

- Set basic Git configurations to personalise the version control environment.
- Initialise a Git repository for a new project.
- Understand the concept of branches and demonstrate how to create, switch, and merge branches.

Outline

Using Git

In many cases, we issue Git commands through our built-in terminal or the Git Bash interface (which comes included in Git for Windows).

We can start by confirming whether Git is installed on our machine using the `git version` command.

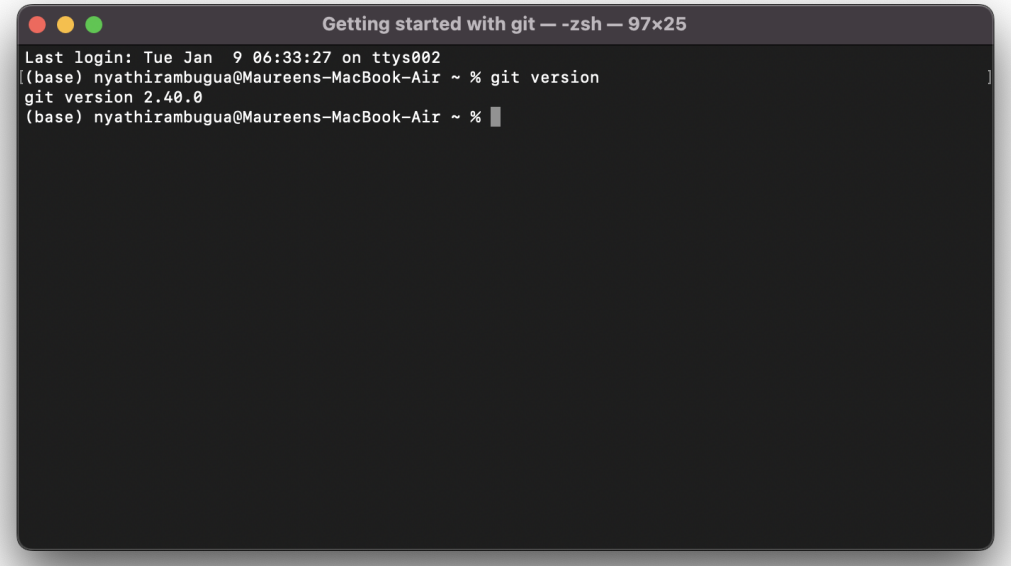

Figure 1: Check the Git version.

If Git is installed on our computer, the **Git version available will be returned**. If no version is returned, it means that Git has not yet been installed.

> Note that we clear the terminal after each step before entering a new command (using the `clear` command for Mac and `cls` for Windows).

Setting up Git

a) Installing Git

If Git is not already installed, we will proceed to **download and install Git** onto our local machine.

There are different ways to install Git **depending on our operating system**. Go to this link and choose the operating system that you wish to download for.

Then, follow the given instructions to install Git on your system.

Type in the `git version` command on the terminal once again to verify that Git has been successfully installed.

Git configuration

Git configuration refers to the process of **personalising our version control environment**.

This way, Git is able to identify us and associate us to the commits that we make. This is helpful when it comes to identifying and tracking contributors in a collaborative project.

We can add user information such as the username and email (you can use the same credentials as your GitHub account) using the `config` command:

- Configure user name: `git config --global user.name "Your Name"`
- Configure user email: `git config --global user.email "your.email@example.com"`
- The `git config --global --list` gives us a view of the current configuration.

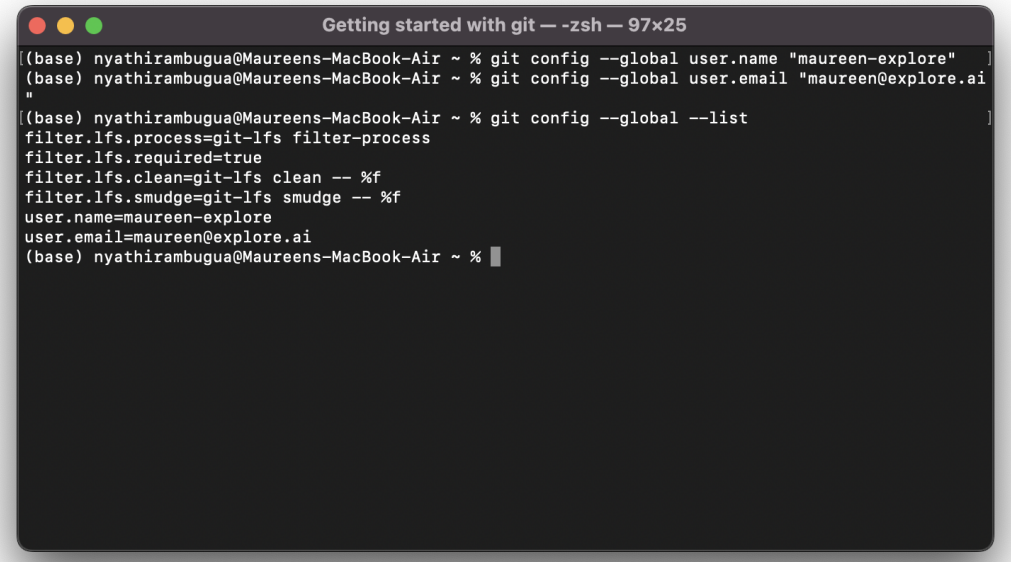

Figure 2: Configure Git.

Note: We use `global` to indicate that we want to set the username and email for **every repository on our computer**. Otherwise, we can remove it if we want to set the username and email only for the current repo.

Initialise a Git repository for a new project

a) Create the project folder

We start by **creating a new folder** where we will store our new project. You can also choose to work on an already existing folder.

We then **navigate to** the newly created/existing project folder on our terminal using the `cd` command.

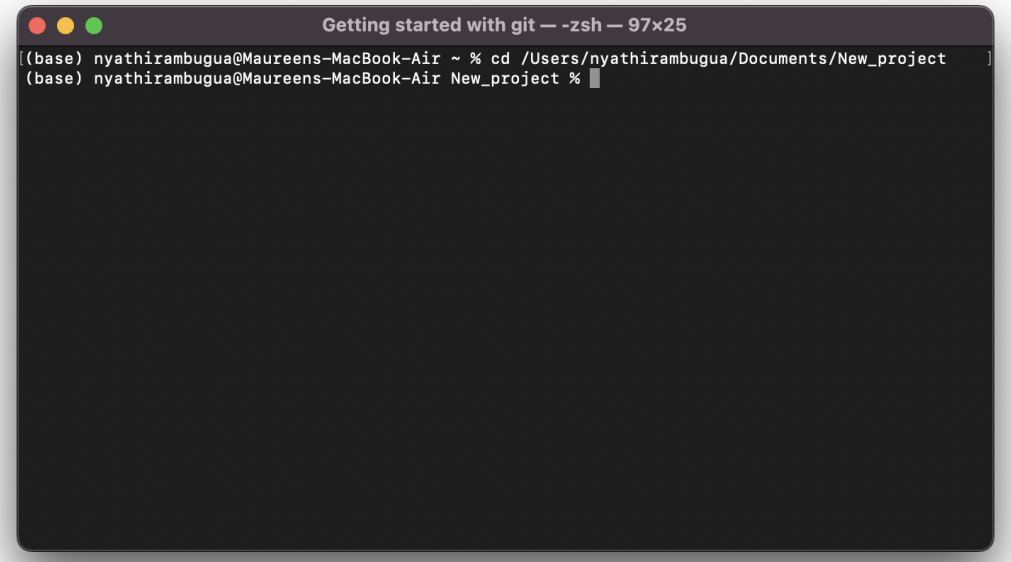

Figure 3: Navigate to the project folder.

Alternatively, if you are using Windows, you can open the folder in your file explorer then right-click on it and select â€œ**Git Bash Here**â€.

b) Initialise Git on the project folder

Once we are in the correct folder, we can **initialise Git on that folder** using the `git init` command.

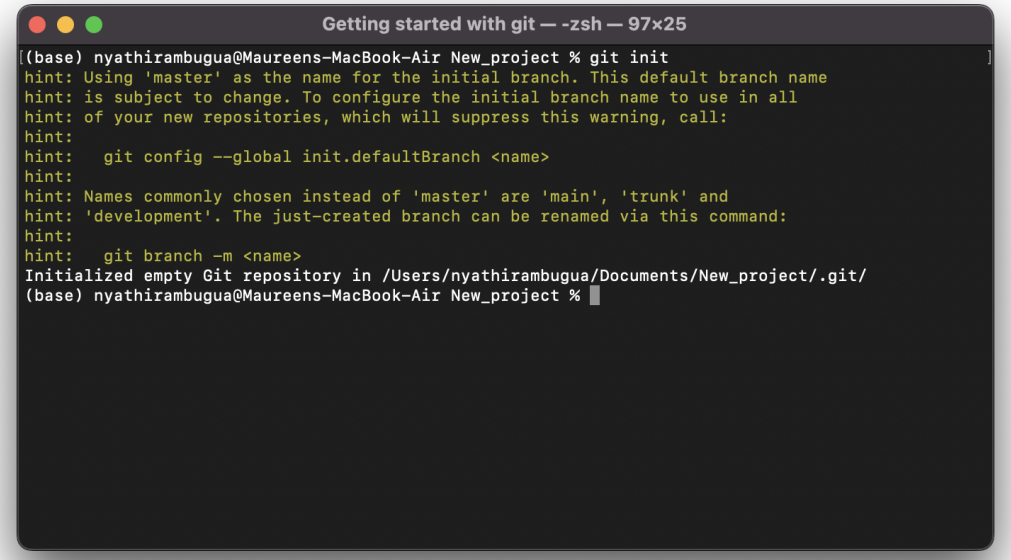

Figure 4: Initialise Git.

A new Git repository has now been created. Git will now track the `New_project` folder.

Adding new files to the Git repository folder

With our local Git repository set up, we can now **add files to our project**

a) Add a file to the project folder

Say we have created a notebook file named `Samples.ipynb` and we want it to be part of our newly created repo. We will now save this file in the project folder we created earlier.

Letâ€™s use the `ls` command to list the files available in our working directory (i.e. My_project).

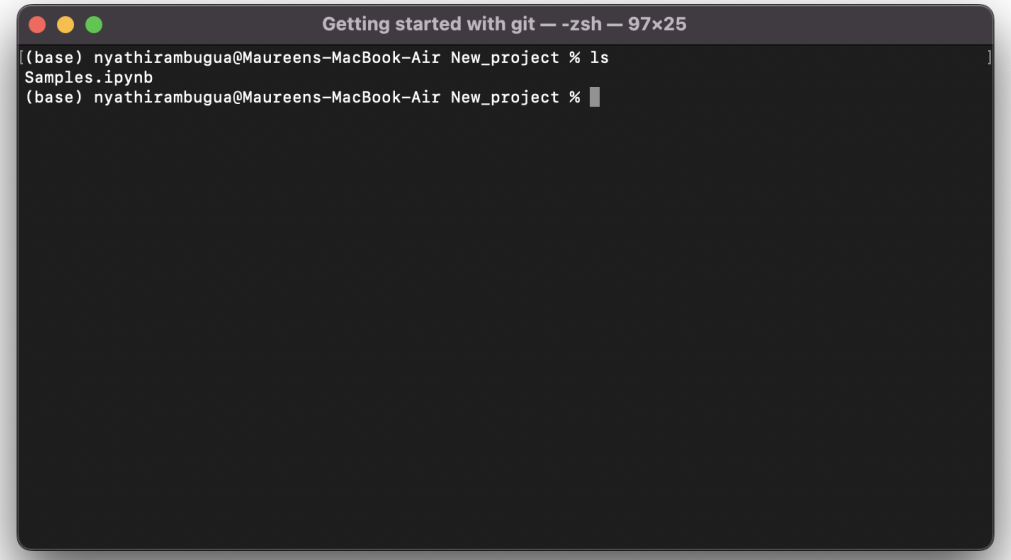

Figure 5: List files in the working directory.

We can see that `Samples.ipynb` is there.

Files in a Git repository folder can have either one of the following statuses:

- **Untracked:** Files that are present in the project directory but have not yet been added to the Git repository for monitoring.
- **Tracked:** Files that have been added to the Git repository and are being monitored for changes.

Letâ€™s use the `git status` command to confirm whether the new file, `Samples.ipynb`, is part of our repo.

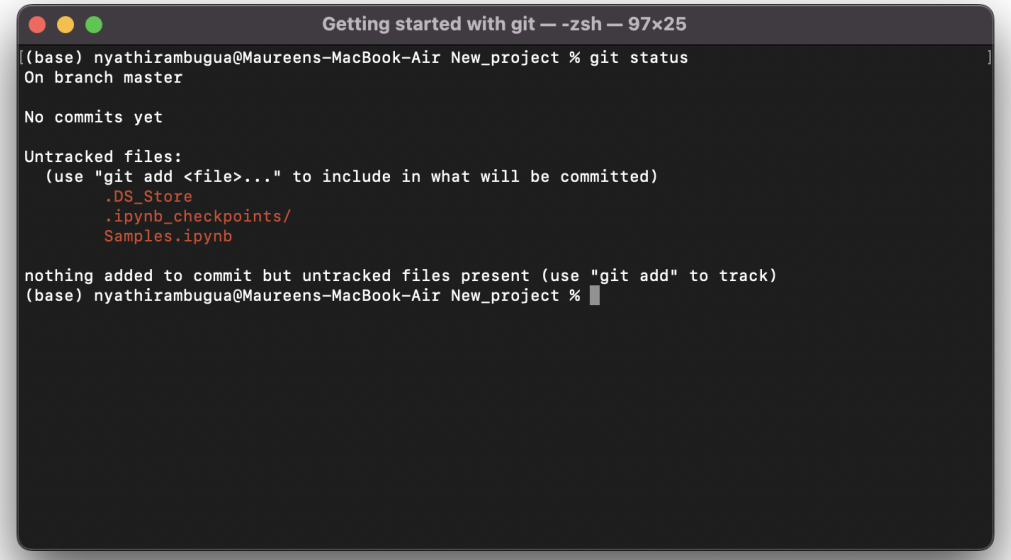

Figure 6: Check the new file status.

When we first **add new files** to a Git repository folder, they are**by default untracked**, which is the case for Samples.ipynb above. We have to explicitly add the new file to the Git repository for it to be included in the version history.

b) Add the new file to the staged environment

For our file to be tracked, we will**add it to the staged environment** using the git add command. When we stage a file, we are ready to commit it to our repository.

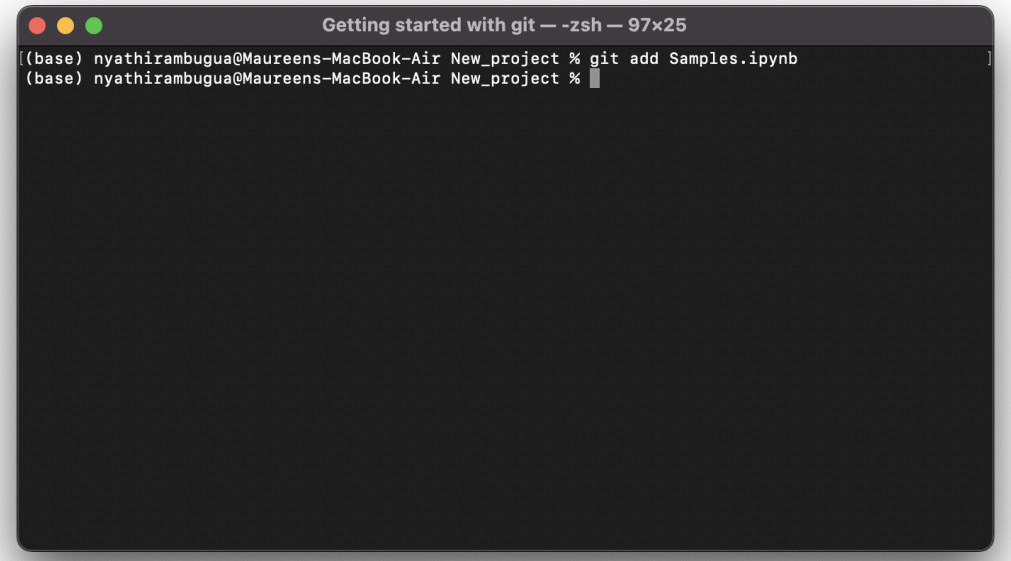

Figure 7: Add a new file to the repo.

Letâ€™s confirm the status of our Samples.ipynb file.

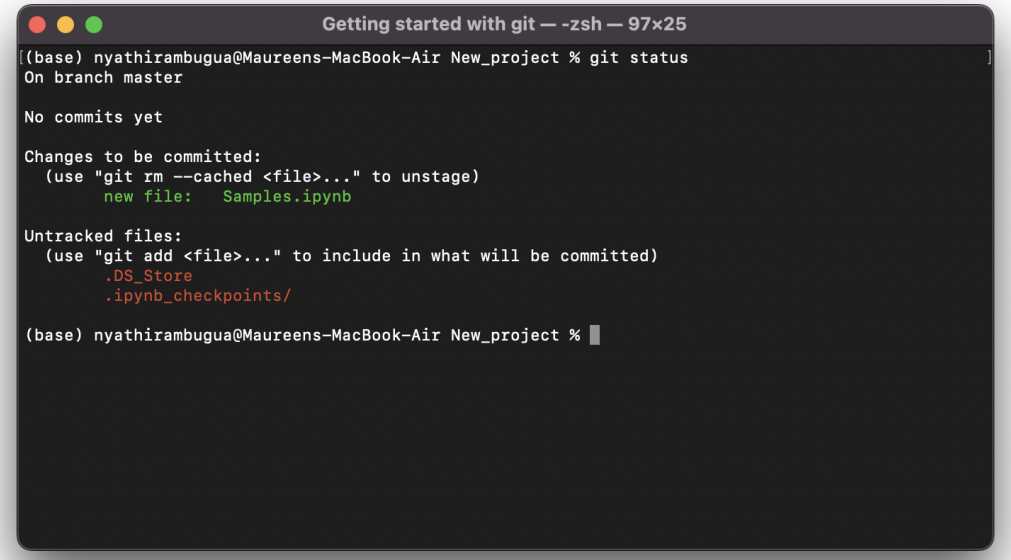

Figure 8: Confirm the status of the new file.

We can see that the file has now been staged.

Note: If we have more than one file, we use`git add .` or `git add --all` instead of individual file names to stage all of them:

- `git add .`: Stages new and modified files only in the current directory and its subdirectories.
- `git add --all`: Stages all changes (new, modified, and deleted files) across the entire repository.

Committing

We use the `commit` command to **commit all the changes that are in the staging area**to the current repository. This can also be accompanied by a commit message:`-m â€œmessageâ€`.

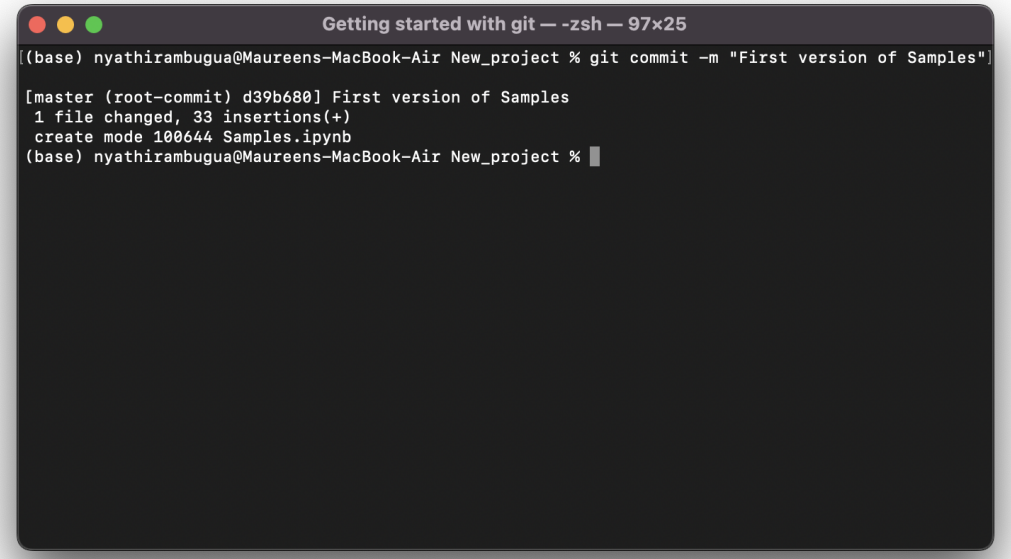

Figure 9: Commit staged changes.

We can use the `git log` command to **view all the commits** that have been made to a particular repository.

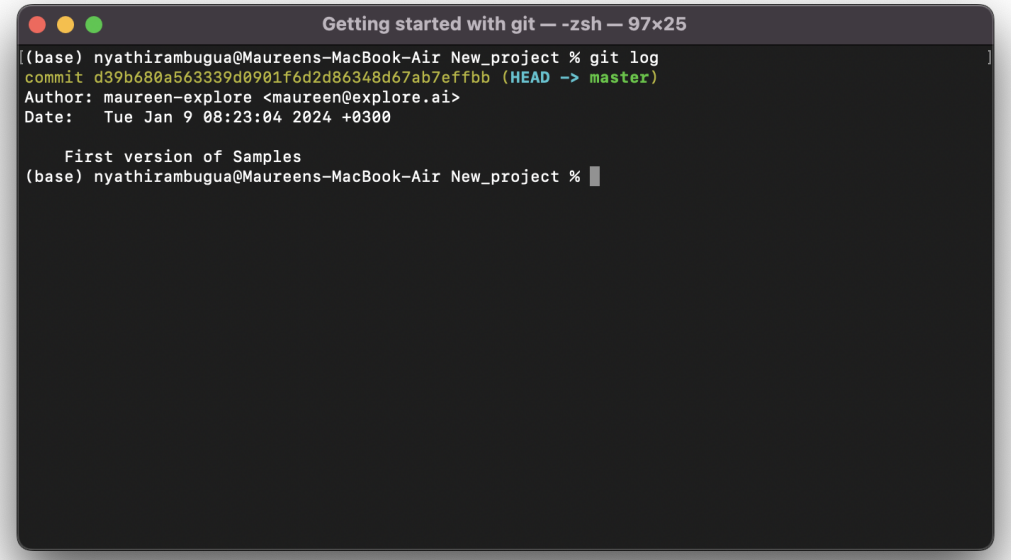

Figure 10: View all commits.

Branching

Suppose we want to update the `Samples.ipynb` file without interfering with the initial version. We will **create a branch** to isolate these changes, which we can later merge into the main branch.

a) Creating a branch

We use the `git branch` command, followed by the branch name to create a new branch.

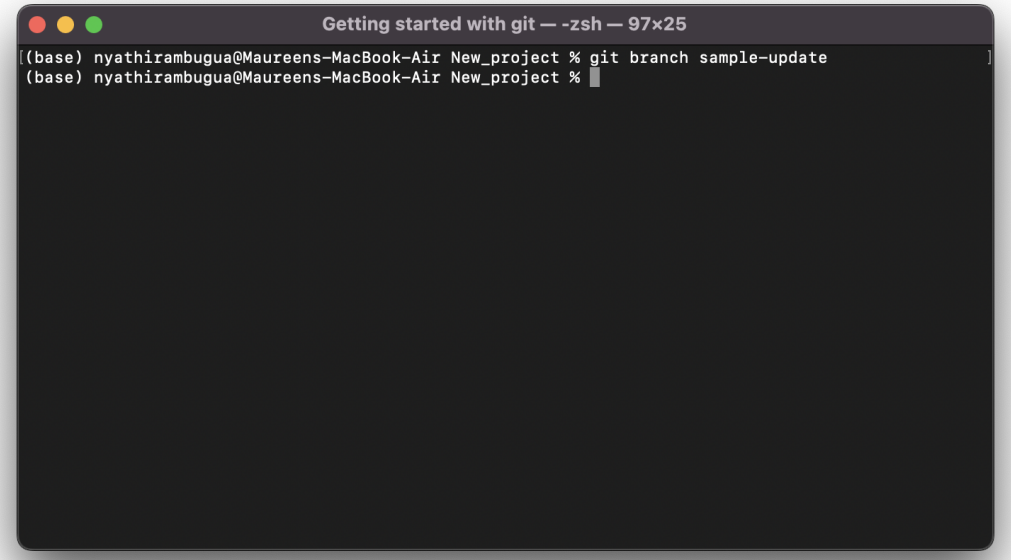

Figure 11: Create a new branch.

We can confirm that our branch has been successfully created by **listing all the branches**:

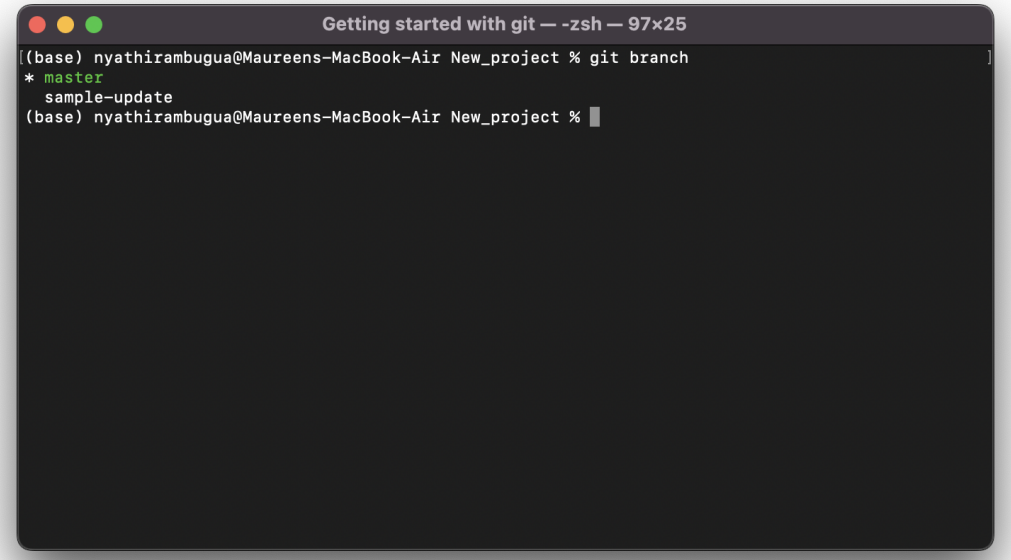

Figure 12: List all branches.

We can see the new branch, `sample-update`, that we have just created. However, we are still currently in the master branch, as shown by the `*` next to it.

b) Switching to a branch

We will switch to the new `sample-update` branch using the `checkout` command.

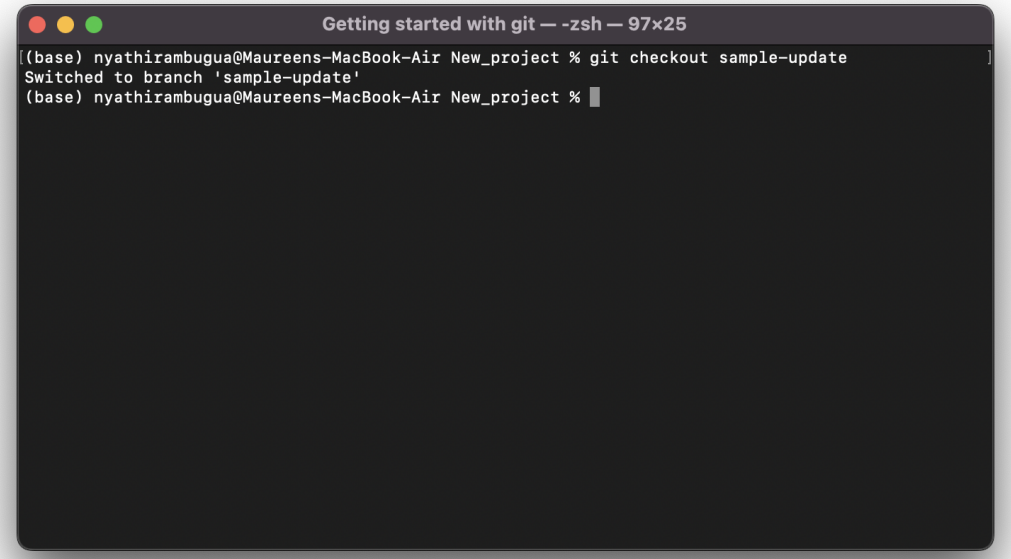

Figure 13: Switch to a new branch.

While in the new branch, we make the desired changes to the `Samples.ipynb` file. We also add a new file, `Country.csv`, containing a dataset.

Now, letâ€™s **check the status** of the current branch:

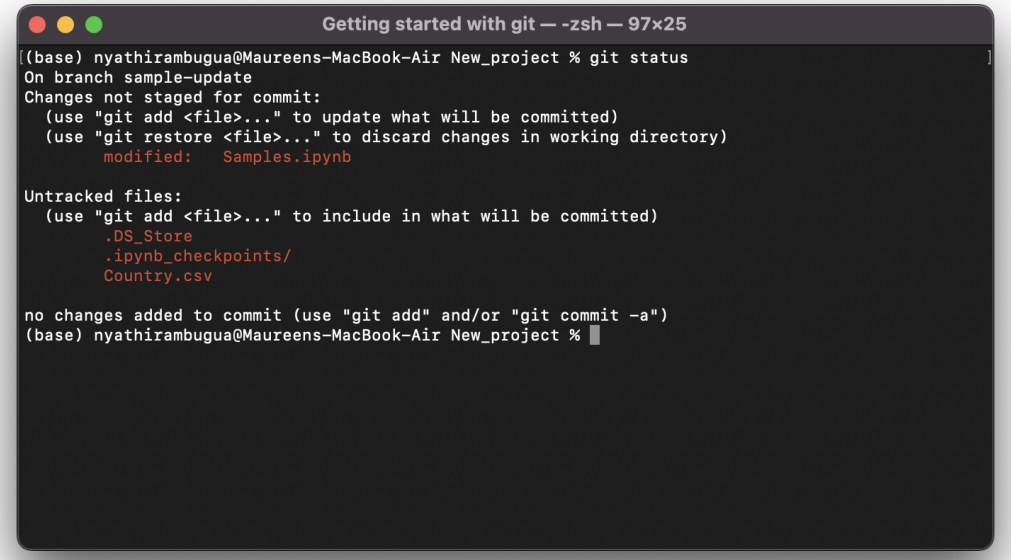

Figure 14: Check the status of the current branch.

We observe that:

- We have changes in the `Samples.ipynb` file that have not been staged for commit.
- We have a new file `Country.csv` that is untracked.

We therefore go ahead and **add the two files to the staging environment**. We will then **commit our changes to the current feature branch**.

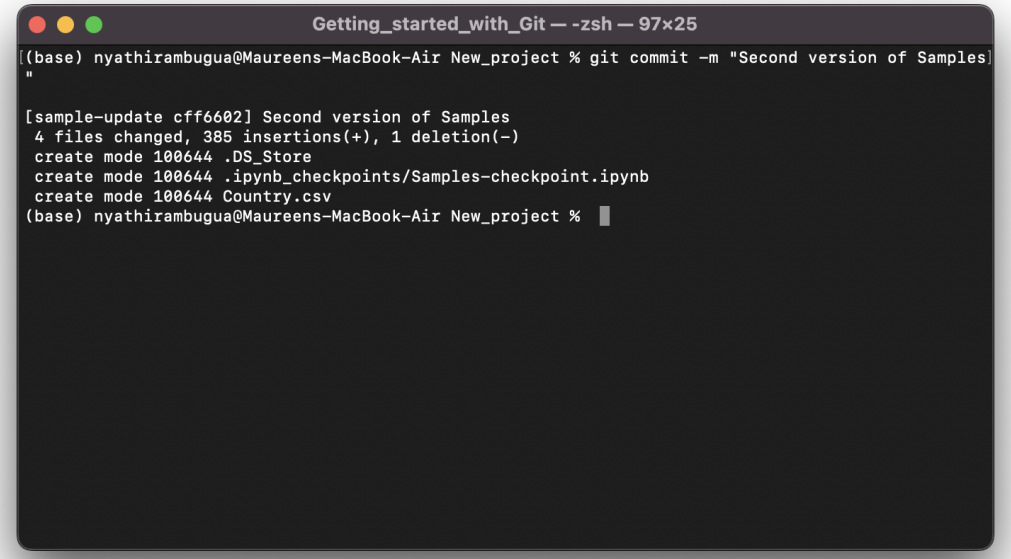

Figure 15: Commit changes to the current branch.

What would we find when we switch back to the main branch and list all the files in the repository?

c) Merging a branch

The changes we have just committed are available on the feature branch, but not on the main branch. That is, the updates made to the `Samples.ipynb` file and the addition of a new dataset file, `Country.csv`.

Therefore, we need to **merge the main branch and the feature branch** for the changes to also reflect on the main branch.

First, we need to switch to the main branch.

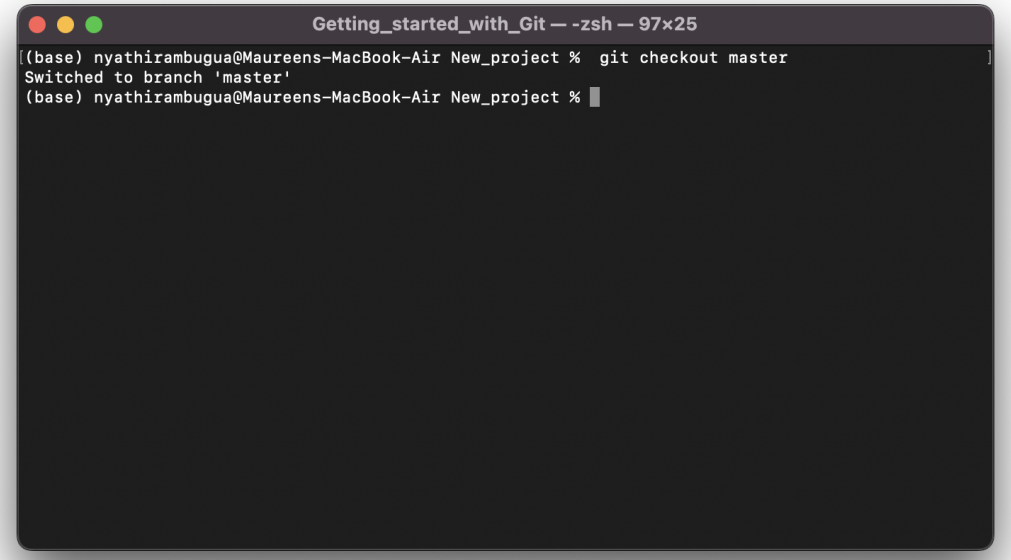

Figure 16: Switch to the main branch.

We then merge the current master branch with the`sample-update` branch.

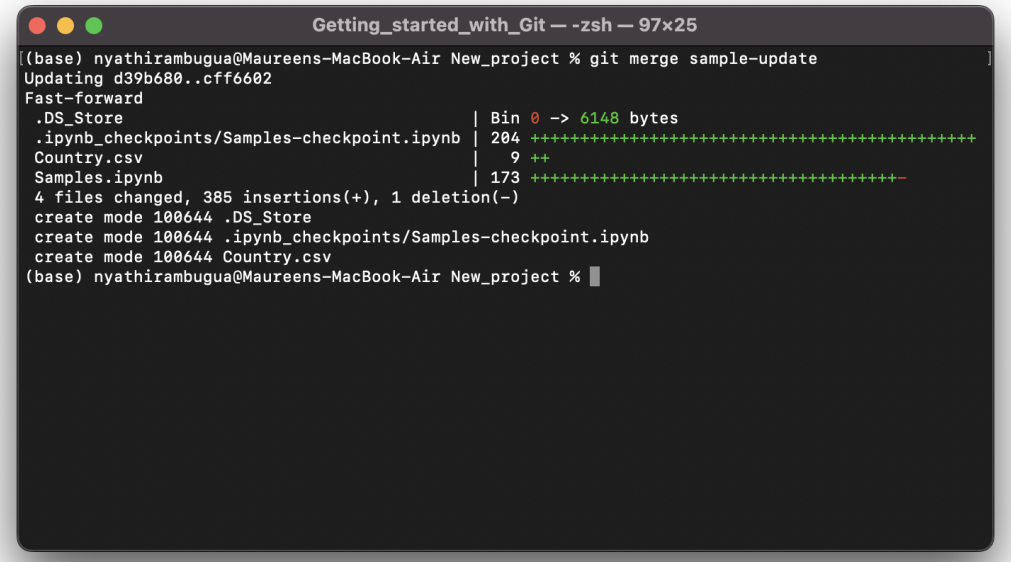

Figure 17: Merge branches.

We can now safely delete the feature branch as we do not need it anymore.

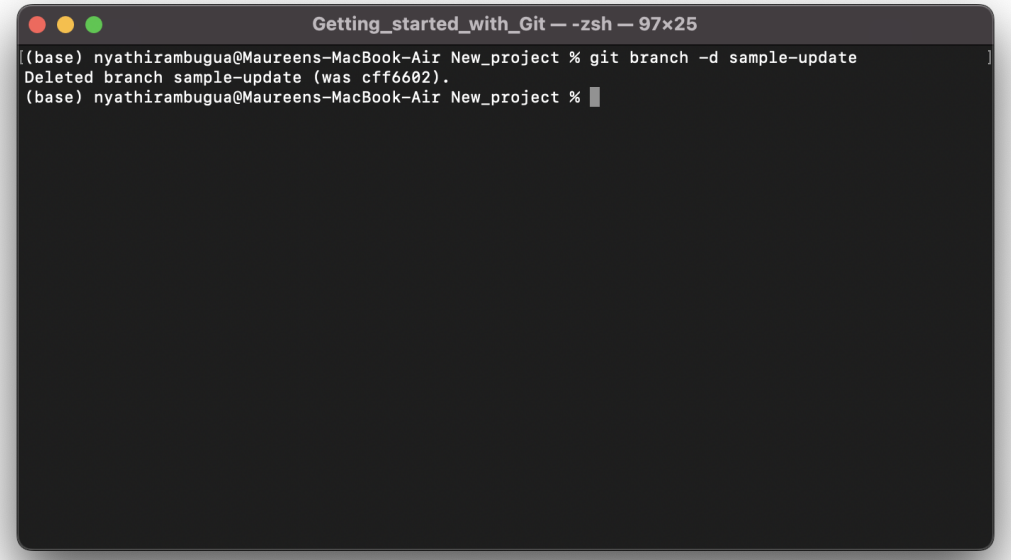

Figure 18: Delete a branch.