5.1 Getting Started

5.1.1 Configuring Git

Git very may already be installed on your system. You can test this by running the following command, which will tell you the version if it is installed. If it is not, you will get a command not found error.

```
git --version
```

Git is provided by package managers such as apt-get, yum, and Homebrew and can be installed it using standard commands. Conda-forge, the Python scientific repository also contains it, which can be useful if you do not have root access. In installing it, you might be missing some dependencies but the package managers should handle this appropriately.

Now, that you have installed Git, you need to do some simple configuration. Effectively, you have to tell Git who you are so that it can keep track of who is changing the files. Note, we will be setting configuration options for the machine as a whole. Most of Git's options can also be done at the repository (repo) level, but we will cover that.

```
git config --gobal user.name "Your Name" git config --
global user.email "Your Email"
```

Since Windows and Linux handle line endings differently on their files, and Git tracks changes line by line, you want to tell Git how to handle the line endings. It can also be helpful to tell Git to colorize its output appropriately.

```
git config --global cor.autocrlf input git config --
global color.ui "auto"
```

You will also want to set a text editor to use in your code. Nano is a good choice for Linux if you are not used to a text editors generally. Sublime can be a good choice for macOS. Many experienced programmers use either Vim or Emacs. They are both very powerful, but come with a steep learning curve.

```
git config --global core.editor "nano -w" git config --
global core.editor "subl -n -w" git config --global
core.editor "emacs" git config --global core.editor "vim"
```

5.1.2 Creating a Repository

Now that Git is setup we can get to the fun part, and create a repository. First, you need to make a directory and enter it.

```
mkdir example cd
planets
```

Now, we need initialize the directory as a Git repository.

```
git init
```

This command just tells git to setup the minimal structure to make the repository function as a repository. If we run the ls command and tell it to show everything, we can see that Git created a .git subdirectory that contains the information regarding the repository.

```
ls -a
```

We can now also check the status of the repository, and you can see that it has nothing to save.

```
On branch master No
commits yet
nothing to commit (create/copy files and use "git add" to track)
```

Note, the observant reader might have noticed most Git commands so far are of the form *git verb options*, where examples of verb include *config* and *init*. This is true generally. Although, there are many verbs and options we have not and will not cover.

The first file we will add is a README. Having such a file is not required, but is recommended. Create a file titled README in your favorite text editor that contains a description of your project.

Having done that we run *git status* again, now you can see that there is an untracked file in your repository.

```
On branch master No
commits yet
Untracked files:


   (use "git add <file>..." to include in what will be committed)
            README


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

Now we see that Git realizes that there is a file *README* in the directory, but it is not tracking it. In other words, it will not do anything with that file until we tell it to.

5.1.3 Ignoring Things

Before, we tell Git to track *README*, we will tell Git which files we want it to ignore. This is useful for temporary and generated files. It can also be useful for very large files such as datasets. (Very large (100's of mebibyte files or larger), do not work well in version control because they are often duplicated in the history.)

The way that we do this is by creating a *.gitignore* file. This file is just a list of files to ignore. It can also recognize wildcard characters.

For example, say that want to ignore our datasets which are saved as *csv* files and we also have some temporary files that end in *.tmp*, then our *.gitignore* file is as follows.

```
*.csv
*.tmp
```

Github also has a collection of useful gitignore files for projects in various programming languages at https://github.com/github/gitignore. If there are multiple different file formats in your projects, you can combine each of the languages' gitignore files into one big one.

5.1.4 Tracking Changes

If we run *git status* again, we see that we now have two untracked files.

```
On branch master No
commits yet
Untracked files:

  (use "git add <file>..." to include in what will be committed)
        .gitignore
        README
nothing added to commit but untracked files present (use "git add" to track)
```

We want to track both the README and the .gitignore files. To add the .gitignore file, we can run the following command.

```
git add .gitignore
```

If we want to add all of the untracked files, we can do that as well.

```
git add --all
```

Now, that we have added the files, we need to commit the changes. The commit command tells Git to save a snapshot of all the files we added before.

Definition 1 *commit A commit is a permanent snapshot of the files that Git tracks stored in the .git directory. It is associated with a unique identifier (hash). Also known as a revision.*

The *m* option tells Git that to associate this commit with the message that follows it. A good commit message contains a brief (< 50 characters) description of what was changed in the commit. You are informing your future self or your collaborators what changes you made in the commit.

```
git commit -m "Created the Repository."
```

Now that we have a current, we can view the git log.

```
commit 8809817caae8b7abff13d482d86817b5774a28ea (HEAD -> master)
Author: Your Name <Your Email>
Date:       Todays Date Current Time -0500

    Created the Repository.
```

The *git log* is a list of past commits, it shows the git hash, which is how git keeps track of the commits, who made the changes, the date and time, and the associated message. When we try to undo commits later, we can look at the log to figure out which commit we want to come back to. The hash in this case is 8809817caae8b7abff13d482d86817b5774a28ea. HEAD always refers to current commit. In other words, in commit 8809817caae8b7abff13d482d86817b5774a28ea at the current date, you took the current changes in HEAD and saved them to the master branch.

If you have already some files, and you want to add their changes and save them, you can combine the *git add* and *git commit* commands into one, by adding the *-a* option to *git commit*. This command does not add any new files though, like *git add* does.

```
git commit -a -m "Your message."
```

As long as you are working linearly, and do not want to return to a previous revision or incorporate changes from your coauthors, this is essentially all there is to it. In 5.3, we will discuss how to push your changes to the cloud, but for now you know enough to get started.

5.2 Git File Structure

Before we discuss how to resolve issues, go back in time, and other powerful and complicated ways of dealing with your history, we must understand how Git structures this history. There are two main classes of locations where Git places files. The first, which we just discussed ends with a commit, and the second starts with the commit. We will will consider the first now.

5.2.1 Staging Area

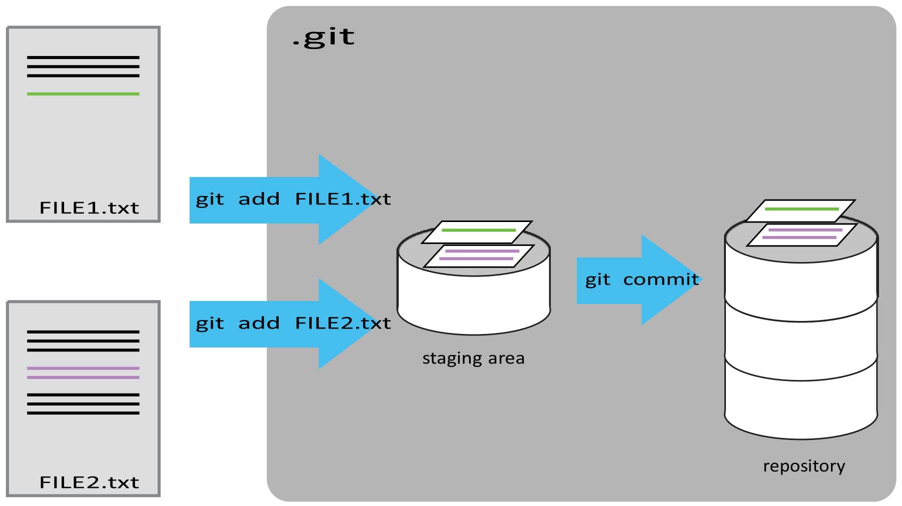


Figure 5.1: Git Staging [1]

The workflow I just described for Git can be broken into three locations. On the left, (where FILE1.txt and FILE2.txt are), we have the working directory. This is where you make changes, and is outside of the .git directory.

The second place is the staging area. *git add* moves stuff from the working directory to the staging area, and then *git commit* moves it from the staging area to the permanent repository. So the first question we have is what is the inverse of git add. How can remove files from the staging area?

Say you have modified the README and added the changes. The output of *git status* is as follows.

```
On branch master
Changes to be committed:
```

.

```
  (use "git reset HEAD <file>..." to unstage)


          modified:    README
```

As we can see, if want to remove changes from the staging area, we can use *git reset*. Recall that HEAD is the name of the current commit. Let’s run the suggested command and see what happens. Using *git reset* with the HEAD commit and without any options does not change the history at all nor does it change what is going on the working directory, and so it is a rather safe command.[2]

Once you have reset the changes in the stating area, *git status* returns the following.

```
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
           (use "git checkout -- <file>..." to discard changes in working directory)


          modified:    README


no changes added to commit (use "git add" and/or "git commit -a")
```

Sometimes you need to change either the repository itself or the current working directory. It is helpful to understand how a Repository is setup when everything is nice and linear. In Git lingo, when we have only one branch.

Let’s say that I want to throw out the changes to README in the working directory, and go back to what was in HEAD. Not only do I want to remove the changes from the staging area like I did above, I want to throw them out complete. Warning! When you do this will lose any unsaved work!

```
git checkout HEAD -- README
```

The *git checkout* command checks out the specified file from a previous commit. The code above checks out the README from the HEAD commit. If we wanted to checkout two files, we could do it as follows. The -- here tells Git that that what follows it are filenames. This is true generally. For example, we could reset certain files using similar syntax.

```
git checkout COMMIT_IDENTIFIER -- file1, file2
```

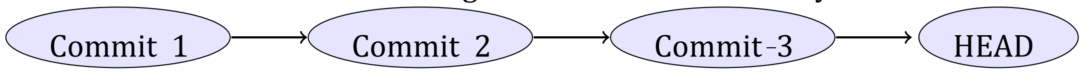
[2] Warning!!! Here be Dragons! *git reset* does have options --*hard* and --*soft* that can be used to rewrite history, and so it can be used to rewrite history as well and to throw out commits that you no longer want.

5.2.2 Git Branches and Merging

Now we turn to the most confusing part of Git—How to keep track of multiple versions of the same file in history. In other words, we will discuss how a repository is laid out in terms of commits.

A Git repository is a history of commits and how they relate. As long no one brings in any changes from collaborators and does not go back in time, everything is nice an linear, and the repository looks as follows, and the repository looks like 5.2.

Figure 5.2: Linear Git History



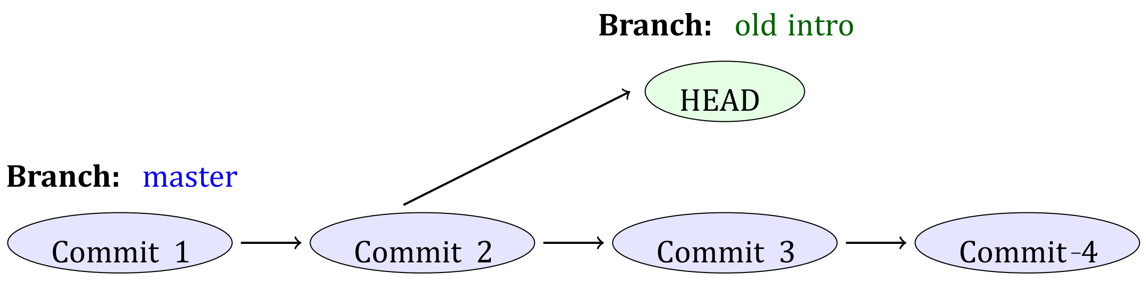
Lets consider the following. You have a history like you did in 5.2, but you decide you want to go back in time to Commit 2. However, you do not want to throw out the changes afterwards. To do this, you can run the following commands.

```
git add --all
git commit -m "Rewrote the introduction to the README." git checkout
COMMIT_3 -b old_intro
```

The first two commands above, you should recognize. They save all that you have done so far to the repository. The third command checks out an old commit for changes. In doing so it rewrites the current working directory, as the checkout command did before. The last two options though you probably do not recognize. The *-b* option tells Git to create a new branch and call it *old intro*.

What is a branch, you ask? A branch is a sequence of commits. Perhaps, this is clearly explained through a picture. The command *git checkout Commit 2 -b -old intro* created the branch old intro and changed the files in the working directory so they align with that branch.

Figure 5.3: Git History with Branches



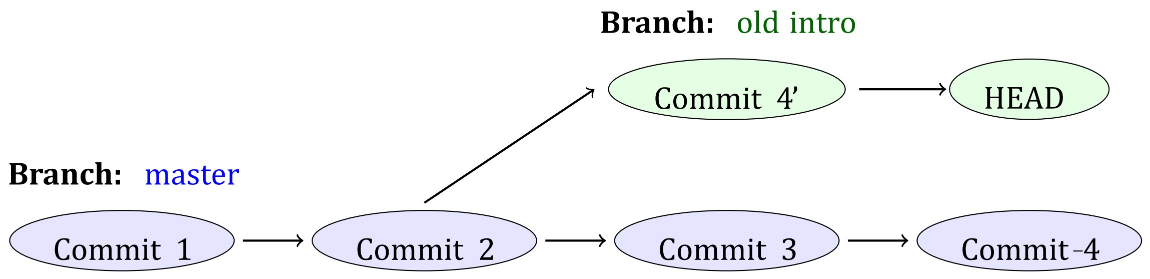
Now we are on branch *old intro*, and can act as normal. To further see what is going on, consider the output of the *branch* command. (The *-a* option, tells Git to list all of the branches.

```
git branch -a
```

```
   master
* <@ \textcolor{green}{old\_intro} @>
```

As we can see, there are two branches *master*, which was the default branch we created, and a new branch *old intro*. Now, if you make changes and commit them they will be added to the new branch, as we can see in 5.4.

Figure 5.4: Git History with Branches



To merge the changes on *old intro* back into the master, we make sure we have commited all of the changes we want in *old intro*. Then we can switch to the master branch and merge the changes back into it.

```
git checkout master git
merge old_intro
```

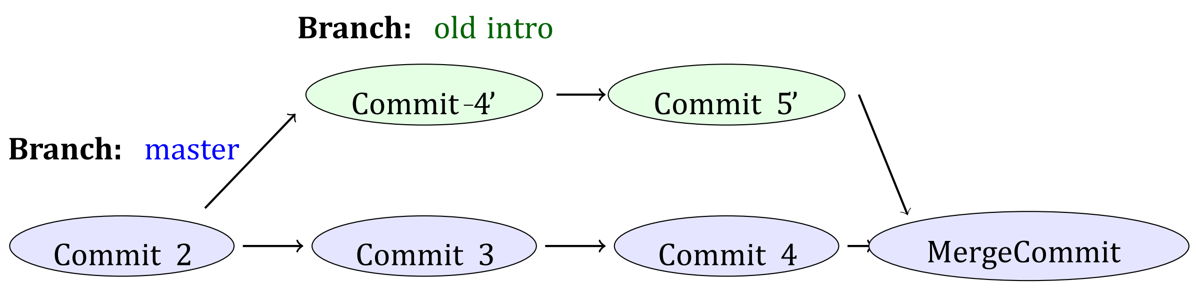
If there are no conflicts, that is all there is to it. Git will take all of the changes and compress them into one commit on the top of the master branch.

5.2.3 Dealing with Conflicts

Sadly, collaboration is like marriage. There will be conflicts. For example, both files might have changed the same line in *README*. Then when we try to merge Git will give us some output that looks the following.

```
Auto-merging README
CONFLICT (content): Merge conflict in README Automatic merge failed; fix
conflicts and then commit the result.
```

Figure 5.5: Git Merge



If we run *git status*, we get the following output, and to resolve the conflict we must do what it says.

```
On branch master
You have unmerged paths.
   (fix conflicts and run "git commit")
   (use "git merge --abort" to abort the merge)

Unmerged paths:
   (use "git add <file>..." to mark resolution)

            <@ \textcolor{red}{both modified: README} @>

no changes added to commit (use "git add" and/or "git commit -a")
```

We need to open *README* with your favorite editor. If you do that you will see some sections that are surrounded by ¿¿¿¿¿¿ and === characters. Git is telling you where the conflicts are and showing the two versions next to each other.

```
<<<<<<< HEAD
Changes in the master (current) branch.
=======
Changes in the old_intro branch.
>>>>>>> old_intro
```

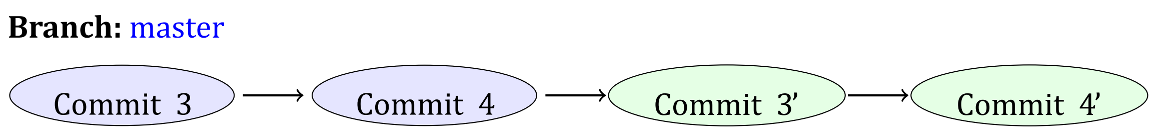
So we need to choose what we want, and delete all of the delimiters, and then add the file back. We can then add the files to mark resolution and finish the *git merge*.

```
git add --all git merge --
continue
```

5.2.4 Rebase, Revert, & Remove

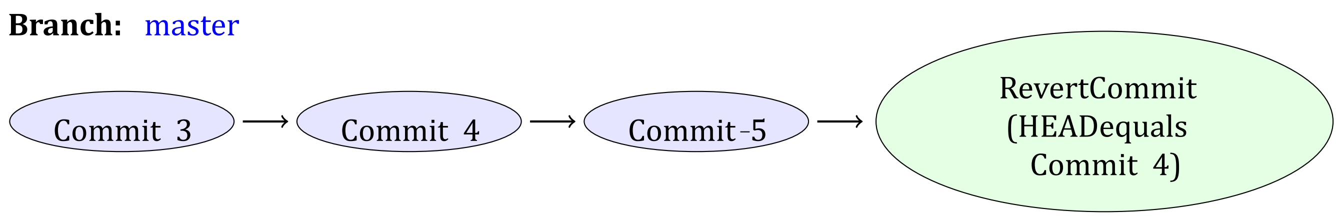
The other way to combine files is by using *git rebase*. This command moves all of your commits in the old intro branch to the front of master. In other words, it takes 5.4 and converts it into 5.6. The advantage of this is it maintains a nice linear history. The disadvantage is that it requires rewriting history. The changes that Git stores in Commit 3' are not the same as the changes it stored when it was on branch old intro because the commit immediately preceding it is different. If there are conflicts, they can be resolved in a similar fashion to how the are resolved with *git merge*.

Figure 5.6: Git Merge



If you want to go back to a previous commit, say go from Commit 4' in 5.6 back to Commit 4, you could check out Commit 4 as a new branch and then merge it. However, since this is such a common thing to want to do, there is a special command for it—*git revert*. Git revert creates a merge commit, that undoes the work of the previous commits.

Figure 5.7: Git Merge



The last command that is quite useful for normal Git use on your own computer is *git rm*, which can be used to remove files. If you want to remove files from both the staging area and the working directory, we can use it as follows.

```
git rm -- file_to_remove
```

If you just want to remove it from the staging area, but not form the working directory—Perhaps, you accidentally added it—we can do the following.)

```
git rm --cached -- file_to_remove
```

5.3 Hosting

Because all work that we care about should be backed up and to facilitate collaboration with your coauthors, you probably want to host your repositories on the cloud. By far the largest host of open-source repositories is GitHub, and it has worked very hard to make its interface easy to use. Gitlab can also be a good option. It lets you host an unlimited number of free repositories.

To upload our example project, you first want to create a blank repository on the host. Once you have done that, you can return to your terminal and tell your local machine about your host.

```
git remote add origin <remote repository URL> git push -u
origin/master
```

In the future, you will treem the remote branch *origin/master* as a branch that you want to keep *master* roughly up to date with. To retrieve changes from the remote to the local, you need to first fetch them (download them) and then merge them into the current branch.

```
git fetch origin git merge
origin master
```

To save keystrokes, Git provides the shortcut *git pull* which runs these two commands. To push changes from the local to the remote, you can use *git push*, but you must pull first. So to get the two branches back in sync, we can run these commands.

```
git pull git
push
```

5.4.1 Collaboration

Once you understand how to do handle interactions between Git branches, collaboration is quite simple. Both people setup local master branches and connect them to same remote branch. The *git clone branch url* command can be used to download a repository that already is hosted somewhere on a new computer.

A basic collaborative workflow goes as follows.

1. *git pull* your coauthor's changes from the remote repository.
2. Make you own changes.
3. Use *git -am "message"* to add your changes.
4. *git push* your changes to the remote.
5. Iterate.

Since it is easy to handle conflicts using this workflow, the authors do not need to be in sync when they make changes, reducing the difficulty of coordinating between the two people. In addition, since everything is stored on a remote server, you automatically get backups of all of you work. This sort of workflow can even be useful if you are working alone but have a version on a server somewhere.

Implement Basic Version Control with Git

1. Install Git

If you don't have Git installed, you can download it from git-scm.com.

2. Set Up Git

Open your terminal (or Git Bash if you're on Windows) and configure your Git username and email:

```
git config --global user.name "Your Name"
git config --global user.email "your.email@example.com"
```

3. Initialize a Git Repository

Navigate to your project directory:

```
cd path/to/your/project
```

Initialize a Git repository:

```
git init
```

4. Create Initial Project Files

Create some initial files for the project. For example:

```
mkdir example-project
cd example-project
echo "# Example Project" > README.md
echo "print('Hello, World!')" > main.py
```

5. Add Files to the Repository

Add the files to the staging area:

```
git add README.md main.py
```

6. Commit the Changes

Commit the changes to the repository:

```
git commit -m "Initial commit with README and main.py"
```

7. Making Changes and Tracking Versions

Let’s make some changes to `main.py` and track these changes.

Edit `main.py`:

```
# main.py
print('Hello, World!')
print('This is a version-controlled project.')
```

Check the status to see the changes:

```
git status
```

You should see something like:

```
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

    modified:   main.py

no changes added to commit (use "git add" and/or "git commit -a")
```

Add the modified file to the staging area and commit the changes:

```
git add main.py
git commit -m "Added a new print statement to main.py"
```

8. Viewing Commit History

View the commit history:

```
git log
```

This will display something like:

```
commit 1234567890abcdef1234567890abcdef12345678
Author: Your Name <your.email@example.com>
Date:   Sat May 25 14:53:07 2024 +0000

    Added a new print statement to main.py

commit abcdef1234567890abcdef1234567890abcdef12
Author: Your Name <your.email@example.com>
Date:   Sat May 25 14:50:50 2024 +0000

    Initial commit with README and main.py
```

9. Creating and Switching Branches

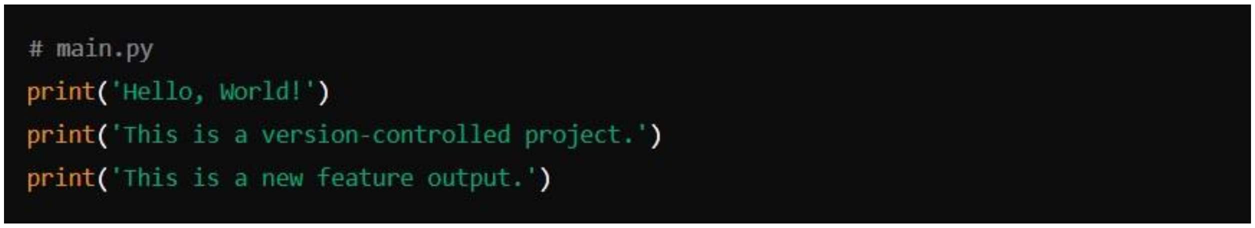
Create a new branch for developing a new feature:

```
git branch feature-new-output
```

Switch to the new branch:

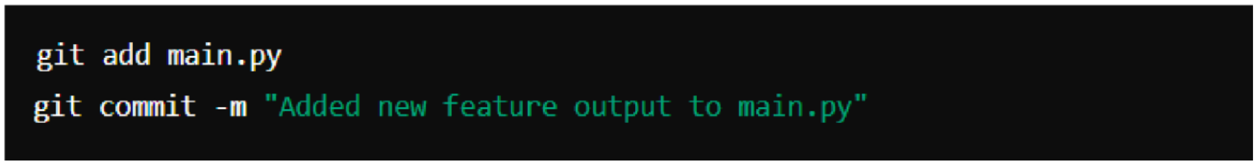
```
git checkout feature-new-output
```

Make changes in the new branch:



```
# main.py
print('Hello, World!')
print('This is a version-controlled project.')
print('This is a new feature output.')
```

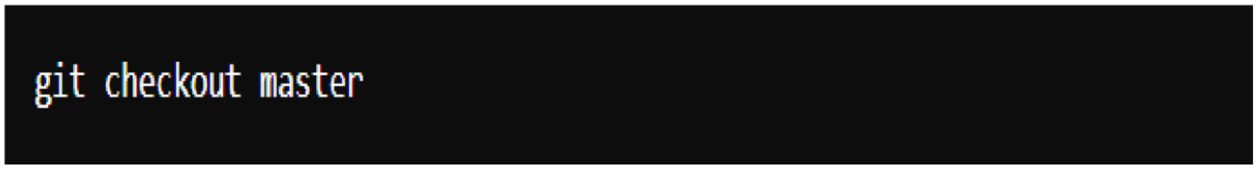
Add and commit the changes:



```
git add main.py
git commit -m "Added new feature output to main.py"
```

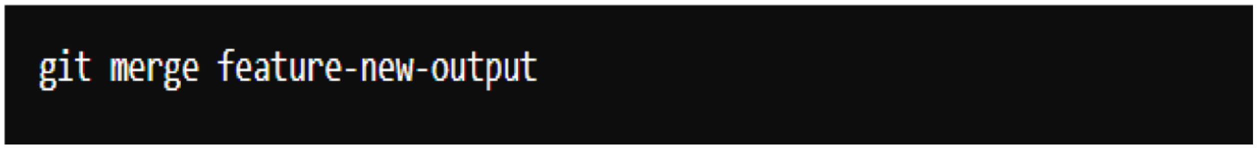
10. Merging Branches

Switch back to the `master` branch:



```
git checkout master
```

Merge the new feature branch into the `master` branch:

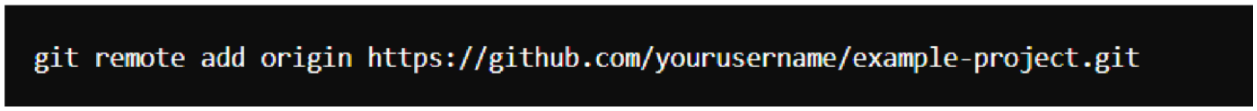


```
git merge feature-new-output
```

Resolve any conflicts if they arise, then add and commit the resolved files.

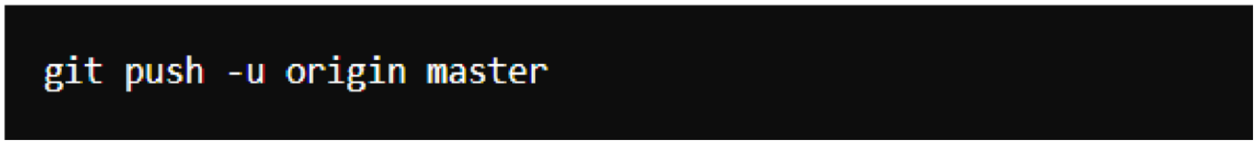
11. Pushing to a Remote Repository

If you have a remote repository (e.g., on GitHub), add it as a remote:



```
git remote add origin https://github.com/yourusername/example-project.git
```

Push your changes to the remote repository:



```
git push -u origin master
```

5.5 Conclusion

In conclusion, Git provides a way of keeping track of past versions of software and papers, making collaboration between various authors easy, and provides backup for your software. It has proven very useful to the open-source community and in academia as well. Git also has several other commands that can be useful to perform less standard actions such as remove sensitive data, clean up past history to save space, and so on, but you should know enough to get started with Git. An extension to Git, known as Git LFS (Git large file system) can be useful if you want to store large files in your repository such as datasets or generated results.

Additional useful resources include Software Carpentry and Stack Overflow, which both contain a vast store of knowledge on how to use the software. Git's documentation is rather exhaustive, but uses a lot of jargon. Once you have a rough idea of what the various commands do, it can be quite useful especially to better understand the various options.