A Really Brief Introduction To Git

Git ready…

V. Markos
Mediterranean College

October 6, 2025

Abstract

In this really brief tutorial, we present the key features of Git, a (the?) code versioning tool you should use in all your coding assignments. We present a brief use case about an incredibly cool project you should know about.

1 Introduction

How often does one of your projects look like this?

```
project-X
├── first-version
│   ├── main.py
│   ├── utils.py
│   └── readme.md
├── better-version
│   ├── main.py
│   ├── utils.py
│   └── read.me
├── just-one-bug
│   ├── main.py
│   ├── main01.py
│   └── main02.py
└── sos
    └── all.py
```

If this is the case most of the times, then you are probably in need of a good version control tool, like Git. With Git you can have full control over what you do on a project, access to its full change history and a lot of cool stuff[1]. Moreover, Git facilitates project sharing, collaboration and much more you will get to know as you use and explore Git.

Git should not be confused with online code repositories like Github or Gitlab. All project management comes with Git. Online code repositories are just a cloud solution that allows your code to be shared there (plus whatever on top of that they have to offer).

2 Installation

Git is too easy to install not have it on your machines, even if you are using Windows. So, let's have a quick look over the various installation methods.

[1]Coolness is subjective, but, well, why should Git not be cool?

2.1 Linux and the like

In most Linux / Unix distributions, Git is accessible via the system's package manager, so you can install it via terminal directly. For instance, in Debian–based distributions, just run:

```
apt-get install git
```

Similarly, for Arch you would use:

```
pacman -S git
```

You can find more for other Linux distributions at the official manual page for Git: https://git-scm.com/downloads/linux.

2.2 macOS

Typically, Git is shipped alongside `Xcode`, so you might need to do nothing to get it. However, you can easily `brew` Git as with most packages with:

```
brew install git
```

For more: https://git-scm.com/downloads/mac.

2.3 Windows

On Windows, you can either use `winget` to install git on your machine (you have to get winget, as it is not shipped with Windows) or download one of the Windows installers available at[2]: https://git-scm.com/downloads/win.

3 Using Git

Having installed git, we shall now present a really brief tutorial on how to use it. We shall use the incredibly important `git-demo` public repository found at: https://github.com/vmarkos-mc/git-demo.

3.1 Pulling the project

Suppose you want to get this all–important project. To do so, create a directory, let's say `git-demo`, and move into it. Open a terminal session there and type[3]:

```
git init .
```

This command initialises a Git repository in the directory you are calling it, indicated here by that tiny `.`. This means that now Git is there, watching you and your files, ready to serve you at your request. You might notice that all this is accomplished by creating a hidden `.git` directory in your project directory, which is where Git actually lives for this project. Deleting this directory *removes Git from this project*, so, be wise with your choices.

You could think of Git as a friend that is there, keeping track of everything you do in a relatively clever way, much like a chief minister for your project.

So far, we have a Git repository set up in our project, but not the actual project. To do so, just run from your project directory:

```
git pull https://github.com/vmarkos-mc/git-demo.git
```

This will most probably print something like this:

[2]With Windows, you are also given the option to install any Linux distribution and let Windows evacuate your machine.

[3]On Windows, "terminal" is called "cmd" or "powershell" (thery are not the same), but we shall stick to Linux terminology here, just because.

```
remote: Enumerating objects: 11, done.
remote: Counting objects: 100% (11/11), done.
remote: Compressing objects: 100% (9/9), done.
remote: Total 11 (delta 0), reused 11 (delta 0), pack-reused 0 (
    from 0)
Unpacking objects: 100% (11/11), 3.34 KiB | 684.00 KiB/s, done.
From https://github.com/vmarkos-mc/git-demo
* branch            HEAD       -> FETCH_HEAD
```

This means that all stuff has been correctly copied into your local directory, so, congratulations, you have just made your first Git pull! Now you can browse around the files you have just downloaded and see the all–precious code included in this project.

3.2 Ignoring

After having the necessary amount of fun, you might observe that the files included are (run `ls -al` on Linux bash for this output):

```
total 28
drwxr-xr-x 3 bill bill 4096 Oct  6 09:33 .
drwxr-xr-x 4 bill bill 4096 Oct  6 09:41 ..
drwxr-xr-x 8 bill bill 4096 Oct  6 09:33 .git
-rw-r--r-- 1 bill bill   80 Oct  6 09:33 .gitignore
-rw-r--r-- 1 bill bill 1066 Oct  6 09:33 main.py
-rw-r--r-- 1 bill bill  120 Oct  6 09:33 requirements.txt
-rw-r--r-- 1 bill bill 1095 Oct  6 09:33 utils.py
```

Besides the first three entries, corresponding to directories, and the two python files, `main.py` and `utils.py`, containing, most probably, the main project entry point and project utilities, respectively, there are those two alarmingly named files:

- `requirements.txt`, and;
- `.gitignore`.

The contents of `requirements.txt` look as follows:

```
black==25.9.0
click==8.3.0
mypy_extensions==1.1.0
packaging==25.0
pathspec==0.12.1
platformdirs==4.4.0
pytokens==0.1.10
```

This file contains some project requirements, as one might suspect. You can run `pip[3] install -r requirements.txt` to install all these project dependencies but this will, most probably, install them system wide. While this might not concern you, it is, in general, a good practice to create local environments to install libraries you need for your projects. In the case of Python projects, you can use `venv` to do so.

But, WAIT! Doing so will create a local directory in your project, with tons of Python files and libraries that you would most probably not want to share with others, since they can just fetch those from Python's package manager locally. So, you need a way to tell Git about this, and the mysterious `.gitignore` serves exactly this purpose. A quick look into it yields[4]:

[4]There is nothing special in starting a directory exclusion rule with a forward slash or not, since all paths are interpreted relatively to Git's directory for this project. This has been used here just for demonstration purposes.

```
# Ignore local python venv
/demo-venv/

# Ignore __pycache__ files
__pycache__/
```

To begin with, lines starting with `#` correspond to comments, so can be safely ignored. Beyond comments, there are two lines included into this file, corresponding to two directories:

- `/demo-venv/` informs Git to ignore the entire directory `demo-venv`.
- `__pychache__/` informs Git to ignore the `__pycache__` directory.

All directories and file paths included in a `.gitignore` file are relative to the location of the file, which should be placed in the same directory as the `.git` directory, typically the project's main directory.

Git does not create ignore files, so you have to do it on your own. On Windows, this might require some tweak, since a file with just an extension and no file name might not be permitted by the operating system.

3.3 Changing

Let us make some minor changes in our project. To begin with, observe that the `utils.Segment` class offers a length computation feature, which the existing project does not utilise. So, why not add this to `main.py` for demonstration purposes? We make the following changes in it:

```python
# main.py

from argparse import ArgumentParser

from utils import Point, Segment


def prepare_parser() -> ArgumentParser:
    parser = ArgumentParser()
    parser.add_argument(
        "a", nargs=2, type=float, help="First edge of the line
    segment."
    )
    parser.add_argument(
        "b",
        nargs=2,
        type=float,
        help="Second edge of the line segment",
    )
    parser.add_argument(
        "p",
        nargs=2,
        type=float,
        help="Point to check whether it belongs to the line
    segment.",
    )
    parser.add_argument(
        "-tol",
        "--tolerance",
        default=0.0,
        type=float,
        help="Float arithmetic tolerance. Default value: 0.0",
    )
    return parser


```

```
35 def main() -> None:
36     parser = prepare_parser()
37     args = vars(parser.parse_args())
38     _a, _b, _p, _tol = args.values()
39     print(_a, _b)
40     a = Point(*_a)
41     b = Point(*_b)
42     p = Point(*_p)
43     segment = Segment(a, b, tol=_tol)
44     print(f"Is point {p} in {segment}? {p in segment}.")
45     print(f"Segment length: {segment.length()}")
46
47
48 if __name__ == "__main__":
49     main()
```

Listing 1: Just a minor change to `main.py`, by adding line 45.

Now, observe that there is also no `readme.md` file, informing users about the purpose and coolness of this project. So, why not create one? We can create it and add the following contents:

```
# README

Read this to learn how cool this project is.

## On Coolness

This project is really cool, because it is..., well, cool.
```

Listing 2: A simple readme file, demonstrating the coolness of this project.

Then, we have to somehow inform Git about the changes we have made, and then commit them. Prior to that, we should tell Git who we are, by configuring our user information:

```
git config user.name "Your user name"
git config user.email "your@email.com"
```

If you are confident enough that you will be using the same credentials on all projects, you can utillise the `--global` flag:

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

Now, since we have made changes to a third–party project, we should first create a new branch and add our new feature to it. To create a new branch, we can use:

```
git checkout -b add-length
```

This should print something like that:

```
Switched to a new branch 'add-length'
```

Now, you can add your changes by using:

```
git add .
```

This adds all changed files to Git, allowing it to track them. Next, you have to *commit the changes to the local repository:*

```
git commit -m "Add readme"
```

This should print something like that:

```
[add-length 8273e96] Add readme
3 files changed, 9 insertions(+), 1 deletion(-)
create mode 100644 readme.md
```

Git informs us that it has successfully updated any changes to the local repository, also providing the short version of the commit hash value (`8273e96`). If we wish to further work locally and then commit, we can repeat the same process — without creating a new branch each time — and then proceed to push our changes to the remote repository. To do so, we run:

```
git push -u origin add-length
```

This pushes any changes made to the remote repository (under the name of `origin` here) to the new branch `add-length`. The remote repository `origin` corresponds to the source url from which this repository has been cloned. In case you want to change it or add more remote resources, you can do so by using `git remote add <name> <source>` to add and `git remote remove <name>` to remove a remote resource.

4 Merging

Assume now that we want to merge changes from our new feature branch, we can do so by pulling that branch, e.g., from the remote repository directly to our local repository. Before doing so, we should first check the branch we are at:

```
git status
```

This should print something like the following:

```
On branch add-length
Your branch is up to date with 'origin/add-length'.

nothing to commit, working tree clean
```

Things are in place, except for our being on the wrong branch. To change back to the `main` branch, we can just checkout this branch:

The main branch on a git repository is typically named either `master` or `main`. The Git default is `master`, which you can change to your liking.

```
git checkout main
```

This should print:

```
Switched to branch 'main'
Your branch is up to date with 'origin/main'.
```

Now, we can merge changes from `add-length` by just running:

```
git pull origin add-length
```

Running now `git status` will will print the following:

```
On branch main
Your branch is ahead of 'origin/main' by 1 commit.
(use ``git push'' to publish your local commits)

nothing to commit, working tree clean
```

All things have been properly updated, so it remains to push changes to the `main` branch. To do so, we run:

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

You can verify that things are up to date by peeking at the logs Git keeps for you:

```
git log
```

This should print something like:

```
commit 8273e966c4579d4a64af2a4da302eec28a77afcb (HEAD -> main,
    origin/main, origin/add-length, add-length)
Author: vmarkos-mc <v.markos@mc-class.gr>
Date:   Mon Oct 6 11:46:02 2025 +0300

Add readme

commit 00901cd9d2f2a3b23101a2e98c1f5066db8a38b8
Author: bill <vassileiosmarkos@gmail.com>
Date:   Mon Oct 6 09:27:17 2025 +0300

Remove python cache files

commit 39e4ce99001bd12f9f6c362723b34559b31e4930
Author: bill <vassileiosmarkos@gmail.com>
Date:   Mon Oct 6 09:26:07 2025 +0300

Initial commit
```

So, all history is actually there, kept safely by your next best friend, Git.

5 Conclusions

Use Git. Just that.

References

[1] *Git - Learn — git-scm.com*. https://git-scm.com/learn. [Accessed 06-10-2025].

[2] *Manual — cli.github.com*. https://cli.github.com/manual/. [Accessed 06-10-2025].