Command Line + Git

Benjamin Walker

January 21, 2026

Why Git?

Git is a version control tool that helps you:

- **Undo mistakes**: go back to a working version of your code.
- **Track progress**: see what changed, when, and why.
- **Experiment safely**: make a new branch to try something without breaking main.
- **Collaborate**: share changes and merge work cleanly, especially when using a remote platform like GitHub.

Why the Command Line?

So that you can work on any operating system, even those that don't have a graphical user interface.

What we will do

1. Command line essentials
2. Git locally: commits, history, branching, merging
3. GitHub: push to a remote and open a pull request

Step 1: Open a terminal

- **Windows:** open **Git Bash** (recommended for this tutorial).
- **macOS:** open **Terminal**.
- **Linux:** open your usual terminal.

You can also use the terminal inside VS Code, PyCharm or JupyterLab, as long as `git --version` works.

Part A: Command line essentials

A1. Where am I? What files are here?

Run:

```
$ pwd
$ ls
```

- `pwd` prints your current folder (the "working directory").
- `ls` lists files in the current folder.
- Tip: press **Tab** to auto-complete names.

A2. Move around folders

Try these:

```
$ cd ..
$ pwd
$ cd ~
$ pwd
```

Useful shortcuts:

- `..` means “parent directory”.
- `~` means “home directory”.

A3. Create a new folder for today

We will work in a fresh folder:

```
$ mkdir cli-git-demo
$ cd cli-git-demo
$ pwd
```

A4. Create and view a file

Create a README file and print it:

```
$ echo "Hello from the command line" > README.md
$ cat README.md
```

Append another line:

```
$ echo "Another line" >> README.md
$ cat README.md
```

You will usually edit files using your favourite text editor or IDE (for example VS Code, PyCharm, or Vim). It is good practice to keep a terminal open so you can run code and Git commands as you work. After a short time, this workflow will start to feel natural, and you will learn Git faster.

A5. Commands you will use a lot

- `mkdir name` (make directory)
- `cd name` (change directory)
- `ls` (list files)
- `cat file` (print a file)
- `echo "text" > file` (write a file)

Part B: Git

B1. Create a Git repository

Inside your demo folder, run:

```
$ git init
$ git status
```

- `git init` turns the folder into a Git repo.
- `git status` shows what Git sees right now.

B2. Stage and commit your first change

Stage the README and commit it:

```
$ git add README.md
$ git status
$ git commit -m "Initial commit"
```

B3. Make another commit and inspect changes

Add one more line, inspect the diff, then commit:

```
$ echo "A third line for Git" >> README.md
$ git diff
$ git add README.md
$ git diff --cached
$ git commit -m "Add another line"
```

B4. View history

List commits:

```
$ git log
```

More concise:

```
$ git log --oneline
```

B5. Branching: experiment safely

Create a branch and switch to it:

```
$ git switch -c title
```

Create a new file on the branch and commit it:

```
$ echo "# Demo Repo" > TITLE.md
$ git add TITLE.md
$ git commit -m "Add a title file"
```

See your branches:

```
$ git branch
```

B6. Merging: bring your changes back to main

Switch back to `main` and merge the feature branch:

```
$ git switch main
$ git merge title
```

A compact history view:

```
$ git log --oneline --graph --decorate --all
```

Optional: delete the branch after merging:

```
$ git branch -d title
```

B7. Example: creating a merge conflict

A merge conflict happens when Git cannot automatically combine changes because the same part of a file was edited in two places.

First, create a branch and edit the same file in two different ways.

Create a new branch and change the README:

```
$ git switch -c conflict-demo
$ echo "This line was added on the branch." >> README.md
$ git add README.md
$ git commit -m "Edit README on branch"
```

Switch back to `main` and make a conflicting change:

```
$ git switch main
$ echo "This line was added on main." >> README.md
$ git add README.md
$ git commit -m "Edit README on main"
```

Now try to merge the branch into `main`:

```
$ git merge conflict-demo
```

Git will report a conflict. Check the status:

```
$ git status
```

Open `README.md` in your editor and you will see conflict markers like:

```
<<<<<<< HEAD
This line was added on main.
=======
This line was added on the branch.
>>>>>>> conflict-demo
```

Resolve the conflict by editing the file so it looks how you want, then stage and commit:

```
$ git add README.md
$ git commit -m "Resolve merge conflict"
```

Optional: delete the branch after resolving the conflict:

```
$ git branch -d conflict-demo
```

B8. Commit messages

A good commit message is short and specific:

- "Fix off-by-one error in indexing"
- "Add README usage example"
- "Refactor data loader"

B9. Undo tools

If you accidentally edited a file and want to discard changes:

```
$ git status
$ git restore README.md
```

If you staged the file but want to unstage it:

```
$ git restore --staged README.md
```

Part C: GitHub

C1. Create a GitHub repository and add a remote

On GitHub, create a new **empty** repository (no README, no license).

Then add it as your remote (replace the URL):

```
$ git remote add origin https://github.com/YOUR_USERNAME/YOUR_REPO.git
$ git remote -v
```

Push your local `main` to GitHub:

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

C2. Push a branch and open a pull request

Create a new branch locally:

```
$ git switch -c notes
```

Make a small change and commit it:

```
$ echo "Some notes for the repo." > NOTES.md
$ git add NOTES.md
$ git commit -m "Add notes"
```

Push the branch:

```
$ git push -u origin notes
```

Now open GitHub in the browser and create a pull request:

- GitHub will usually show **Compare & pull request**.
- Create the PR, then click **Merge pull request**.

Update your local main branch after merging on GitHub:

```
$ git switch main
$ git pull
```

Quick command cheat sheet

Command line

```
# print the current directory (where you are)
$ pwd

# list files and folders in the current directory
$ ls

# change directory into "folder"
$ cd folder

# move up to the parent directory
```

```
$ cd ..

# go to your home directory
$ cd ~

# create a new folder called "new-folder"
$ mkdir new-folder

# print the contents of file.txt
$ cat file.txt
```

Git

```
# create a new Git repository in this folder
$ git init

# show the current Git state (changes, staged files, branch)
$ git status

# stage <file> so it will be included in the next commit
$ git add <file>

# save a snapshot (commit) with a short message
$ git commit -m "message"

# show unstaged changes since the last commit
$ git diff

# show commit history as a compact graph
$ git log --oneline --graph --decorate --all

# create and switch to a new branch called "new-branch"
$ git switch -c new-branch

# merge branch-name into your current branch
$ git merge branch-name

# discard unstaged changes to <file> (revert to last committed version)
$ git restore <file>

# unstage <file> (keep the changes, but remove from the staging area)
$ git restore --staged <file>
```

GitHub

```
# connect this local repo to a GitHub repo at <url>
$ git remote add origin <url>

# push main to GitHub and set origin/main as the default upstream
$ git push -u origin main

# push <branch> to GitHub and set origin/<branch> as upstream
$ git push -u origin <branch>

# pull the latest changes from the upstream remote branch
$ git pull
```

Troubleshooting

- `fatal: not a git repository`: you are not inside a Git repo.

```
$ git init
```

- `nothing to commit`: Git sees no new changes, or you already committed them.

```
$ git status
```

- **Push fails the first time**: make sure you set the upstream branch.

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

- **Merge conflict**: Git will tell you which files conflict. Open them, fix the markers, then run:

```
$ git add <file>
$ git commit
```