

Git is a powerful tool, but it has a reputation of baffling newcomers. It doesn't help that most people are simply thrown in at the deep end and expected to swim.

With the right knowledge, anyone can master git. Once you start to understand it, the terminology will make more sense and you'll (eventually) learn to love it. Stay strong 🙏

Why another guide?

There are plenty of “git tutorials” out there already, but most of them simply tell you to copy/paste specific things to do one-off tasks. Anyone with a keyboard can copy/paste; to really understand how git works and what it can do for you, you need a slightly deeper understanding.

They also tend to throw vocabulary at you without explaining what the words really mean.

This guide aims to give you a workable understanding of the basic phrases and commands you'll use frequently. It's impossible to learn such a powerful and complex tool in just one sitting, so I encourage you to take your time and enjoy the journey.

What is git?

First of all, **GitHub** is *not* git. Many people understandably confuse the two. **GitHub** is a website for hosting projects that *use* git.

Git is a type of **version control system** (VCS) that makes it easier to track changes to files. For example, when you edit a file, git can help you determine exactly *what* changed, *who* changed it, and *why*.

It's useful for coordinating work among multiple people on a project, and for tracking progress over time by saving “checkpoints”. You could use it while writing an essay, or to track changes to artwork and design files.

Git isn't the only version control system out there, but it's by far the most popular. Many software developers use git daily, and understanding how to use it can give a major boost to your resume.

In complex projects, where multiple people might be making changes to the same files simultaneously, it's easy to get into a weird state. Anyone who's dealt with “merge conflicts” and those baffling >>>>>> ===== <<<<<< symbols can attest to this.

If you start to understand how git works, you'll see why conflicts occur and how to recover from these situations easily.

How to get git

Git comes installed by default on many systems. If you don't already have it:

- You can download git's **command-line interface (CLI)** [here](#). I recommended this for both beginners and advanced users.

- If you prefer to use a fancy graphical user interface (GUI) instead, try **GitHub Desktop** (for Windows and Mac). This will be simpler to use, but will make it more difficult to really see what's going on.

The examples below all assume you're using the CLI.

❗ You use the command-line interface by typing in a **terminal**. If you're not familiar with terminals, that's okay—try **this** first (or search Google for help).

Common commands

Below is a series of basic commands with descriptions of what they each do. This section is intended to be interactive. As you read, feel free to try out the commands yourself before moving on. At the end there will also be a “real life” example with a list of commands you can try all at once.

Notes:

- Any **jargon** will be highlighted in bold the first time it's described. Feel free to look up those terms in the official **git glossary** or **reference guide** for more details.
- This is a simplified guide. It attempts to be as accurate as possible while avoiding some of the messier details. There are links at the end for more depth.
- 🌟 There will be some new insights in the descriptions, so make sure to keep reading.

Here are some of the most common commands, roughly in the order you will encounter them:

*Start your own **repository** from scratch (in any existing folder on your computer):*


```
git init
```

This will create a hidden `.git` folder inside your current folder—this is the "repository" (or **repo**) where git stores all of its internal tracking data. Any changes you make to any files within the original folder will now be possible to track.

✧ The original folder is now referred to as your **working directory**, as opposed to the repository (the `.git` folder) that tracks your changes. You *work* in the working directory. Simple!

Clone an existing repo:

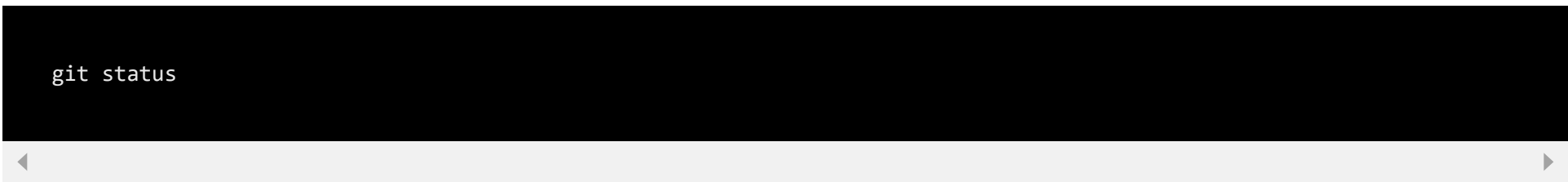
```
git clone https://github.com/cooperka/emoji-commit-messages.git
```



This will download a `.git` repository from the internet (GitHub) to your computer and extract the latest **snapshot** of the repo (all the files) to your working directory. By default it will all be saved in a folder with the same name as the repo (in this case `emoji-commit-messages`).

✨ The URL you specify here is called the **remote origin** (the place where the files were *originally* downloaded from). This term will be used later on.

*View the current **status** of your project:*



```
git status
```

This will print some basic information, such as which files have recently been modified.

You should check your status anytime you're confused. Git will print additional information depending on what's currently going on in order to help you out.

Create a new **branch** name:

```
git branch <new-branch-name>
```

You can think of this like creating a local “checkpoint” (technically called a **reference**) and giving it a name. It’s similar to doing **File > Save as...** in a text editor; the new branch that gets created is a reference to the current state of your repo. The branch name can then be used in various other commands as you’ll soon see.

Similar to branching, more commonly you will save each checkpoint as you go along in the form of **commits** (see `git commit` below soon).

Commits are a particular type of checkpoint called a **revision**. The name will be a random-looking **hash** of numbers and letters such as e093542. This hash can then be used in various other commands just like branch names.

✨ *That’s really the core function of git: To save checkpoints (revisions) and share them with other people.*

Everything revolves around this concept.

If you've ever created a checkpoint to something, you'll be able to get back to it later as long as your `.git` folder is intact. It's magical. See `git reflog` if you're interested in learning more.

Branching is a huge and complex topic. I'll write more about it soon; for now you can read more [here](#) if you want to.

Check out a particular branch:

```
git checkout <existing-branch-name>
```

You can think of this like “resuming” from an existing checkpoint. All your files will be reset to whatever state they were in on that particular branch.

⚠ Keep in mind that any changes in your working directory will be kept around. See `git stash` if you're interested in a simple way to avoid headaches.

😊 You can use the `-b` flag as a shortcut to *create* a new branch and then *check it out* all in one step. This is quite common:

```
git checkout -b <new-branch-name>
```

View the **differences** between checkpoints:

```
git diff <branch-name> <other-branch-name>
```

After editing some files, you can simply type `git diff` to view a list of the changes you've made. This is a good way to double-check your work before committing it.

For each group of changes, you'll see what the file *used to* look like (prefixed with `-` and colored red), followed by what it looks

like now (prefixed with + and colored green).

See further down for more advanced examples of this command.

Stage your changes to prepare for committing them:

```
git add <files>
```

After editing some files, this command will mark any changes you've made as “staged” (or “ready to be committed”).

⚠ If you then go and make more changes, those new changes will *not* automatically be staged, even if you've changed the same files as before. This is useful for controlling exactly what you commit, but also a major source of confusion for newcomers.

If you're ever unsure, just type `git status` again to see what's going on. You'll see “Changes to be committed:” followed by file names in green. Below that you'll see “Changes not staged for commit:” followed by file names in red. These are not yet staged.

🧐 As a shortcut, you can use **wildcards** just like with any other terminal command. For example:

```
git add README.md app/*.txt
```

This will add the file `README.md`, as well as every file in the `app` folder that ends in `.txt`. Typically you can just type `git add -a` to add everything that's changed.

***Commit** your staged changes:*

```
git commit
```

This will open your default command-line text editor and ask you to type in a **commit message**. As soon as you save and quit, your commit will be saved locally.

The commit message is important to help other people understand what was changed and why you changed it. There's a brief guide [here](#) explaining how to write useful commit messages.

😁 You can use the `-m` flag as a shortcut to write a message. For example:

```
git commit -m "Add a new feature"
```

***Push** your branch to upload it somewhere else:*

```
git push origin <branch-name>
```

This will upload your branch to the **remote** named **origin** (remember, that's the URL defined initially during `clone`).

After a successful push, your teammates will then be able to `pull` your branch to view your commits (see `git pull` below).

🤖 As a shortcut, you can type the word `HEAD` instead of `branch-name` to automatically use the branch you're currently on. `HEAD` always refers to your latest checkpoint, that is, the latest commit on your current branch.

✨ As mentioned earlier, everything in git can be thought of as a checkpoint. Here's a list of the types of checkpoint you know about now (again, these are technically called “references” and “revisions”):

- `HEAD`
- `<branch-name>`, e.g. `master`
- `<commit-hash>`, e.g. `e093542d01d11c917c316bfaffd6c4e5633aba58` (or `e093542` for short)

There's also:

- `<tag-name>`, e.g. `v1.0.0`
- `stash`

Finally, special characters like `^`, `~`, and `@{ }` can be used to modify references. They're quite useful; learn more [here](#).

Fetch the latest info about a repo:

```
git fetch
```

This will download the latest info about the repo from `origin` (such as all the different branches stored on GitHub).

It doesn't change any of your local files—just updates the tracking data stored in the `.git` folder.

Merge in changes from somebody else:

```
git merge <other-branch-name>
```

This will take all commits that exist on the `other-branch-name` branch and integrate them into your own current branch.

⚠ This uses whatever branch data is stored locally, so make sure you've run `git fetch` first to download the latest info.

For example, if someone else adds a few commits to the master branch of origin, you can do the following to download their changes and update your own local master branch:

```
git checkout master    # Make sure you're on the right branch.
git fetch              # Download any new info from origin.
git merge origin/master # Merge the 'origin/master' branch
                       into your current branch.
```

✧ The name `origin/master` here literally means the `origin/master` checkpoint on your computer. Git uses this notation to differentiate branches of the same name (e.g. `master`) located in different places (e.g. your own branches vs. `origin`'s branches).

😎 As a shortcut, you can use the **pull** command to both *fetch* and *merge* all in one step. This is more common than merging manually like above:

```
git pull origin master
```

Here we separate the words `origin` and `master` (without a slash like above). We don't want to use the `origin/master` checkpoint on our own computer, because that's stored offline and is probably out of date. We instead want to fetch directly from the `master` branch of the remote endpoint called `origin`. Keep an eye out; the difference is important!

For a deeper understanding of how merging works and how **conflicts** are resolved (with fun pictures and graphs), see the official [merge docs](#). This will also be covered extensively in part 3 of this series.

Real life example

Here's a series of commands that could hypothetically be executed while developing a real feature. See if you can figure out what they would each do, then try it out yourself and check.

```
git clone https://github.com/cooperka/emoji-commit-messages.git
cd emoji-commit-messages
git status
git checkout -b my-new-feature
echo "This is a cool new file" > my-file.txt
git status
git add --all
git status
git diff HEAD
git commit -m "Add my-file.txt"
git status
git log
git push origin HEAD
```

```
git checkout master  
git pull
```

Most of these commands have additional parameters you can pass to customize them, and shorthand versions for brevity, all of which make everything more interesting (read: confusing). There's usually more than one way to accomplish a given task.

Now what?

Now you know everything there is to know about git!

...just kidding, this is only the tip of the iceberg. There's a lot more you need to understand before you can claim to be an expert, but for now, you at least have the basics to be able to work with a team and understand the lingo.

Once you understand these basic commands, you can and should continue to learn more from [git's website](#) or this [more advanced guide](#).

If you found this useful, please help by **tapping the 🤝 button as many times as you'd like** so others can find it too.

Thank you, and stay tuned for [part 2](#). Let me know in the comments if there are any specific topics you're interested in.

