

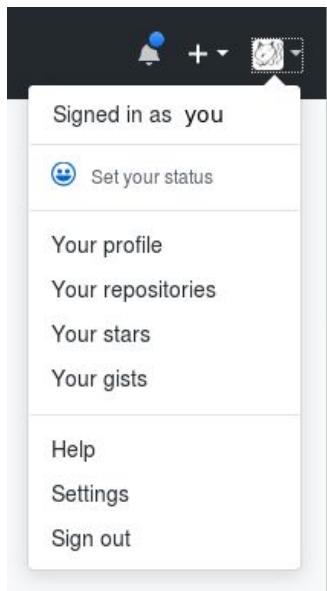
# Basic Git Commands

## Overview

Now that you know what Github is and how it interacts with Git, we can get started on creating a **repository**, and learning some basic terminal commands.

*Reminder: A repository (or "repo" for short) typically stores all files involved in a single project. They can include pretty much anything the project might need - folders, files, images, videos, spreadsheets...*

## Let's get started



Firstly, open Github and navigate to **Your profile** (under your picture in the top right).

Then, go to **Repositories** and then **New**. Give your repo a name - something like `infpals-versioncontrol` and a description if you'd like.

You will see an option which asks if you want to your repo to be **Public** or **Private**. Public repositories are for open source software that is built by multiple people, so choose this option for now. (Don't worry, others still won't be able to make changes without your permission.) On the other hand, Private repositories can't be seen or altered by anyone unless you allow them. Note that with the basic Github package you can only use Public channels.

Github also offers you shortcuts to creating three useful documents commonly found in repositories. Don't worry about **.gitignore** or **licence** for now. However, it's good practice to create a **readme**.


## Create a new repository

A repository contains all the files for your project, including the revision history.

---

Owner

Repository name \*

 you ▼

 / 


creative-floating-banana-frogs ✓

Great repository names are short and memorable. Need inspiration? How about **friendly-fiesta**.


Description (optional)

The awesomest git repo of all!

---

☒  **Public**

Anyone can see this repository. You choose who can commit.

☐  **Private**

You choose who can see and commit to this repository.


---

☒ **Initialize this repository with a README**

This will let you immediately clone the repository to your computer. Skip this step if you're importing an existing repository.

Add .gitignore: **None** ▼

 | 

Add a license: **None** ▼ 

Create repository

This is a markdown (.md) file which is automatically displayed whenever anyone opens your repo. Use this to tell new users anything you'd like them to know about the project - what it is, how it works and what they should do with it. The default contents are your repo name and description, which are good enough for now. If you want to change this file further, use [Markdown syntax](#).

Finally, click **Create repository** and you're done! You can access this from your profile by clicking **Repositories**. In addition, you can pin up to six public repositories to appear at the top of your profile if you want them to be the first thing you or anyone else sees when they visit.

## What's all included?

If you click on your repo name, you should see a number of tabs which provide a multitude of useful tools for your repo.

- The **Code** tab shows all the current files in your repo - as mentioned earlier, these can be of any types of files but in a real repo the majority of these will typically end up written in one or more programming languages.
- The **Issues** tab takes you to a discussion forum where you can track the progress of various features of the project. Issues are more useful when you have multiple people collaborating on the same project - unless you're in the habit of talking to yourself.
- The **Pull** requests tab tracks pull requests, which are what you'll receive if someone wants to add a new feature to your repo - or what you'll send if you want to modify one of their repos. We'll come back to pull requests later on in Activity 2, where we'll be looking at how multiple people can work simultaneously.
- The **Projects** tab allows you to create and manage sub-projects within your main repo. Needless to say, we won't be using this quite yet, but it's invaluable if your repo is so big you start to lose track of where everything is.
- The **Wiki** tab lets you create an entire wiki guide to your repo. Again, this is for large open-source projects so we won't be covering it here, although it's fairly self-explanatory. You can read more on creating a wiki [here](#).
- The **Insights** tab provides you with a number of features for tracking the progress of your project - who is contributing and when. Again, this won't be relevant quite yet.
- Finally, the **Settings** tab provides exactly what you think it does, including the option to delete your repo if you so wish.

## Cloning

Now, we can start working on your repo - and first things first, you'll need to create your own copy of it. (If you don't do this and do all your work in the central directory, your repo just becomes a kind of Google Drive for code, which is hardly the point of version control).

To download the contents of a git repo as a zip archive:

- Click **Clone or download** on the main page of your github repo and then click **Download ZIP**. Note that the downloaded files will not be synced with any changes online, so we'll be using the second method.

To clone the git repo (BETTER):

- Using the command `git clone`. To do this, click **Clone or download** and copy the URL. Open a terminal window, `cd` to where you want your repo clone to be, and then type `git clone` followed by the copied URL. It should look like this:

```
git clone https://github.com/your_account_name/your_repo_name
```

This will create a local repo, complete with a **working directory** for you to work in. This directory will contain:

- A `.git` folder containing configuration files
- All of the files that were in the remote repo - right now, this should just be `README.md`

*Note that all git commands follow the syntax of the word 'git' followed by the name of the command.*

## Your new best friend

Working from the command line can get confusing. It's powerful, for sure, but it's all too easy to lose track of where you are. That's why `git status` is your best friend. Like `ls` and `pwd` are helpful if you get lost when navigating your directory, `git status` will let you know what's going on with git at any given moment. Navigate into your `infpals` directory and run `git status` - you should see the following:

```
# On branch master
nothing to commit, working directory clean
```

The branch you are on and what's available to commit will change as you progress through the activities.

## Stage time

So (finally) it's time to see how version control actually works. First off, let's create a new file. The following command will create a new text file with the words "This is an example file" in it:

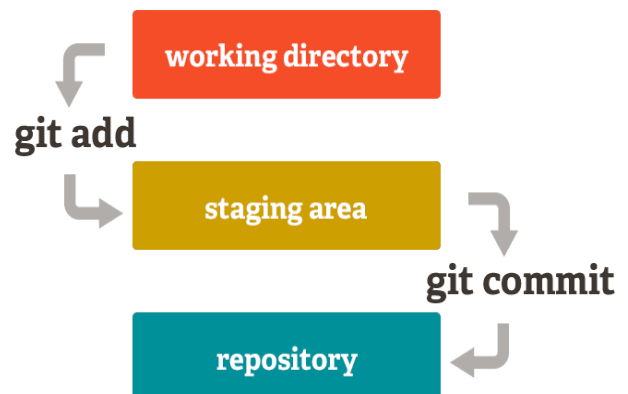
```
echo "This is an example file" > example.txt
```

Run `git status` and you'll see our text file is now an "untracked file" - sounds a lot more exciting than a five word file really is!<sup>1</sup>

Now let's add the file to the **staging area**. To do this, use

```
git add example.txt
```

The staging area is a collection of files with changes that are tracked by git.



<sup>1</sup> Image from <https://dev.to/sublimegeek/git-staging-area-explained-like-im-five-1anh>

Running `git status` again (it's your best friend, remember?) and you'll see our file has entered the second stage of the pipeline - it's a "change to be committed."

## Making a commitment

A commit, according to Wikipedia, “adds the latest changes to [part of] the source code to the repository”. Think of it as taking all of your changes currently sitting in the staging area, bundling them up, and carrying them out all at once. To commit the changes - in this case, the addition of a file - run the command

```
git commit -m 'your_commit_message'
```

Commit messages are optional, but they're a simple, easy way of telling your other collaborators what you've done for the project. It's customary to use an *imperative* mood to write commit messages - instead of 'added example.txt', a typical commit message would be 'add example.txt'. Think of it as you telling git to do what you want it to.

	COMMENT	DATE
○	CREATED MAIN LOOP & TIMING CONTROL	14 HOURS AGO
○	ENABLED CONFIG FILE PARSING	9 HOURS AGO
○	MISC BUGFIXES	5 HOURS AGO
○	CODE ADDITIONS/EDITS	4 HOURS AGO
○	MORE CODE	4 HOURS AGO
○	HERE HAVE CODE	4 HOURS AGO
○	AAAAA	3 HOURS AGO
○	ADKFJSLKDFJSDKLFJ	3 HOURS AGO
○	MY HANDS ARE TYPING WORDS	2 HOURS AGO
○	HAAAAAANDS	2 HOURS AGO

AS A PROJECT DRAGS ON, MY GIT COMMIT  
MESSAGES GET LESS AND LESS INFORMATIVE.

2

Now, run status again. You'll see that our local directory, or branch, is *ahead of origin/master* by 1 commit. What does this mean? It means that we've added one feature - in this case, the new file - that does **not** exist in the remote repo (the one stored on Github).

---

<sup>2</sup> Image source: [http://imgs.xkcd.com/comics/git\\_commit.png](http://imgs.xkcd.com/comics/git_commit.png)

## No pushover

Now we've actually done something for the project, it's time to transfer those changes to the remote repo. The most important thing to remember here is that this can only be done if there are no **conflicts** - for example, if someone else already added an `example.txt` to the remote repo, your push will be rejected. We'll go over conflicts in more depth in the second activity. Thankfully, we know that this isn't an issue now - your repo should be exactly as you left it (i.e. containing just `README.md`).

This means that we can push our changes. To do this, just run

```
git push
```

You can also run

```
git push origin master
```

`origin` refers to the remote repo (where your code is stored, in this case on GitHub)

`master` refers to the branch you are pushing to, but branching is covered later.

You'll have to provide your Github username and password - this is to ensure that only you can edit your repo. And voilà! You've now modified the central repo. Run `git status` again and you should see that your staging area and commits are now empty again. Go back to Github and check your repo, and you'll find that your `example.txt` has appeared within it, as if by magic. Now use the command line to:

- add some more lines to `example.txt`
- add the changes to the staging area
- commit your changes
- push your changes to the remote repo

All of this should be doable from what you've done already, although do ask if you get stuck!

## Another way to start a Git repository

This method may be more useful if you already have a project stored locally, and would only want to track changes locally - not remotely using GitHub or another platform.

You can initialise a local git repository by running

```
git init
```

in the desired directory. Stage and commit any files and continue this for local version control.

You can then push it to GitHub if required by creating a new repository on your GitHub account and running

```
git remote add origin <remote repository URL>
```

Then push the repository as usual using `git push`.

## Feeling the pull

What happens if something happens to the remote repo on Github? Say, some other user has added a new feature? You could `git clone` the entire thing, but you probably don't want to create more copies of the whole codebase. The solution is `git pull`. Pull is somewhat like the opposite of `git push`, in that it applies changes that someone else has made in the remote repo to your local repo.

For now, we'll test it out by making a change ourselves. Go to the remote repo (i.e. on Github) and create a file there using Create new file. Name it, write something in it and commit it, adding a description if you'd like. Now simply run

```
git pull
```

and you'll see that your file has now appeared in your working directory.

## We have to go back!

We all make mistakes from time to time. Thankfully, Git provides useful tools to account for the worst of these problems.

### Checkouts

The first is `git checkout`. This command undoes all unstaged changes that you've made, resetting your directory to how it was just after the last time you used `git push`. To try this out, use `echo` to add some text to `example.txt`, then run `git status`. You'll see that the only unstaged change you have is a modification of `example.txt`. Now run

```
git checkout example.txt
```

What happens? `status` will tell you that there are no adds or commits, and `cat example.txt` will show you that your file is as it was before! This is very handy for undoing accidental changes, although note that it will undo **all** changes - it's not just a substitute for Ctrl-Z.

## Resetting

The next useful command is `git reset`. You can think of reset as a kind of reverse add - while add moves a change from the working directory to the staging area, reset moves a change out of the staging area and back in to your working directory. This won't undo any of your changes - it simply moves it back one stage in the pipeline.

Try this out by making a change, using `git add example.txt` and then running

```
git reset HEAD example.txt
```

Running status will show you the status as if you never used add or reset. Then, you can `checkout example.txt` to take you back to where you were at the start of this section, or modify it and then add it again if you wish.

Note that you can also use `git reset` without any parameters to reset the entire staging area.

## Hard reset

If you want to combine checkout and reset into one command, you can hard reset. This will reset the staging area AND the working directory to match the most recent commit. If you want to try this, run

```
git reset --hard
```

Adding the `--hard` flag tells Git to overwrite all changes in both the staging area and working directory.

## Log

Like `git status`, `git log` is yet another useful tool to stop you getting lost. Simply running

```
git log
```

will tell you the history of all the commits you've made to this repo. Navigate through the history with the arrow keys and exit with Ctrl+Z.



You might be a little lost in all the text. Never fear - there are a number of flags that you can use with `log` as to simplify things. For now, let's run the following:

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

Now that's much prettier! Run this command anytime you want to see what your branches look like, to get a better feel of how they move around and interact.

**Pro tip:** set up an alias for fancy log by typing something like

```
alias gil="git log --graph --oneline --all --decorate"
```

From now on, you can just type `gil` instead of the long command. If you're interested in other flags you can use, you can check them out [here](#).

## Congratulations!

You've reached the end of Activity 1! If you're still not sure about anything here, how about going back and trying it again? Alternatively, if you're confident, feel free to move on to Activity 2.