

# Git Information

## Getting git installed

- On Linux it will probably be installed already
- On university computers it may be installed to cmd
- On your own machines the best way is [gitforwindows.org](https://gitforwindows.org), which will allow you to install **git bash** (a built in Linux bash shell emulation) and **git GUI**, a graphical version. It can also add git to your cmd path.

## Creating a Repository


This is easily done on whatever git host you're using. We use [github.com](https://github.com), where you can create an account and create a repository linked to your account with the click of a button. You can also add a **README** in markup to provide details about the project as an automatic first commit.

### Create a new repository

A repository contains all project files, including the revision history. Already have a project repository elsewhere? [Import a repository](#).

Owner

Repository name \*

 Scorpuss3 ▾ /

Great repository names are short and memorable. Need inspiration? How about **didactic-eureka**?

Description (optional)



Public

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



Private

You choose who can see and commit to this repository.

Skip this step if you're importing an existing repository.

☐ Initialize this repository with a README

This will let you immediately clone the repository to your computer.

Add .gitignore: **None** ▾

Add a license: **None** ▾



Create repository

# Common Commands

## Configuring your identity

The local version of git must know who you are (your email and username for github). With git bash, this will popup when you first push, but to set permanent ID in the command line:

**git config --global user.email “[you@example.com](#)”**

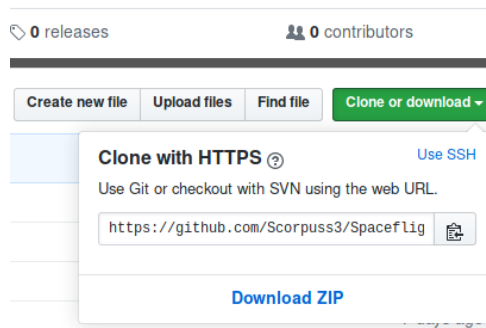
**git config --global user.name “username”**

To use an ID for only this repository, omit the ‘--global’

## Cloning a Repository

Once a repository has been created, you will want a local copy of it to work on.

**git clone <url>**



## Pulling changes

Locally, git will keep track of changes to the code in the repository in its cache. When there are changes to the code on the server, make sure you have no uncommitted changes and run:

**git pull**

## Creating & Switching Branches

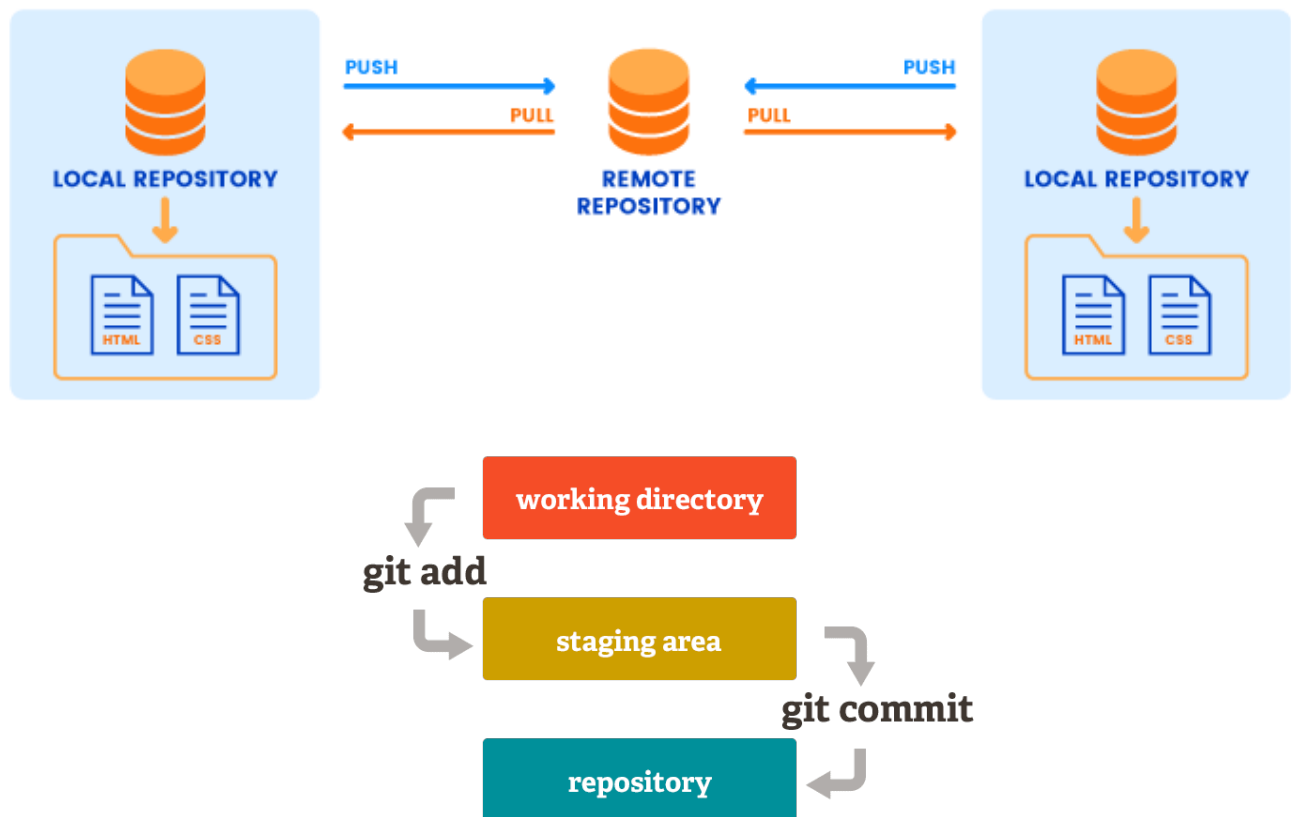
<b>git checkout -b &lt;branchname&gt;</b>	Create a branch called ‘branchname’ <b>locally</b> , and switch to it.
<b>git checkout &lt;branchname&gt;</b>	Switch to branch ‘branchname’

To check what branches your local git knows about, run ‘**git branch**’. If your remote branch is not there (when you first clone a repo, you will probably just get the master branch), use the following:

<b>git branch</b>	Check branches tracked locally
<b>git checkout --track origin/&lt;branchname&gt;</b>	Switch to a local branch tracking the remote branch ‘branchname’.

Note that branches may only be switched while you have no uncommitted changes. If you want to store your changes in your branch to switch without committing, you can do the following:

<b>git stash</b>	Store uncommitted changes
<b>git stash pop</b>	Restore uncommitted changes after returning to your branch



## Committing Changes

To update the repository with your changes, you must first select which files you wish to update. This is done by moving files to the 'staging area' ready for commit with the **git add** command, or in the GUI . This will also start tracking a new file if it is not yet tracked by the repository.

Once files have been staged, you can **git commit** the changes with a message to update the local repository's version of the code.

At any point after a commit, pushing the changes (next) section will update the remote branch.

**Git status** is a useful command that can be used at any time to show the current staging state of files.

<b>git add &lt;filename&gt;</b>	Stage the file with name <filename> only
<b>git add --all</b>	Stage any changed or new files
<b>git reset &lt;filename&gt;</b>	Remove a file from the staging area (undo add)

<b>git commit -m "&lt;message&gt;"</b>	Commit changes in staging area to local repository with a message. This will allow them to be pushed to the remote branch.
<b>git status</b>	Get general information about file changes and the staging process, along with current branch info. This is your go-to to find out what's going on.

## Pushing Changes

Pushing is what updates the remote repository with the changes committed to your local repository.

The command 'git push' can be run on its own once the remote branch is tracked. If it is not yet tracked, you can update the tracking as below or just manually specify the remote branch your local branch should push to.

If a push doesn't work, git will usually give a message with why, along with suggested command to try.

<b>git push</b>	Push commits to remote branch
<b>git push --set-upstream origin &lt;remotebranch&gt;</b>	Push an untracked local branch to a specified branch on the remote server (creating it if necessary). From this command on, the local branch will track remote and you will only have to type ' <b>git push</b> ' in future.

## Investigating Changes

To check changes between your files and the last commit, you can use git diff on its own. You can also specify particular files and other branches if you wish.

<b>git diff</b>	Show differences between last commit and unstaged files.
<b>git diff --staged</b>	Show differences between current files and the staged versions.
<b>git diff &lt;branchname&gt;</b>	Show changes between this branch and <branchname>

## Merging Branches

When changes on the project a branch was created for are complete and checked, the branch needs to be merged back into the main master branch so that the changes can be used in the main code base (and pulled into other branches as needed).

<b>git merge &lt;sourcebranch&gt;</b>	Merge branch <sourcebranch> into the current branch.
<b>git merge &lt;dstbranch&gt; &lt;srcbranch&gt;</b>	Merge <srcbranch> into <dstbranch>

Be careful about what you merge into master. It's a branch that should only be updated with code that *definitely* works, and it's good practice to ensure that at any stage the master branch can be built to a working system.

## Sub-modules

Sub-modules are often used in open source software to link libraries across github. They allow you to link a **certain commit** of another repository into a folder within your own repository. Note the extra '**git submodule init**' step needed when cloning a repository that uses submodules.

<b>git submodule add &lt;subrepo&gt;</b>	Add a remote repository as a submodule of our own repository. This will be included in its own directory.
<b>git submodule init</b>	This is needed when you have cloned a repository which includes submodules. Execute this in the submodule's root directory.

## Help

What is included here is only a brief summary. The git command line itself is actually very helpful in giving you commands to use if they fail.

There are also numerous guides and cheat-sheets online which will give you a good description of how to use specific commands and what to do when there are issues.