# Comparing files

In terminal:

* Windows: fc (file compare)
* Mac/Linux: diff (difference)

On Mac:

1. Go to terminal
2. Navigate to directory with file > cd [FolderName]
3. Compare two files using 'diff -u' > diff -u [File1] [File2]

This will give you a report with just the sections of the file that are different.

-/+ signs refer to lines that were added or removed.

# Commits

Commits are a user generated 'checkpoint' of a version change of a file.

Each commit represents a version of the content at a specific point in time.

When creating a commit, the user submits a 'commit message' which describes what change has been made to that version of the file.

Commits have:

* a unique ID (serial #)
* author name
* date
* message

To compare two different commits > git diff [Serial #1] [Serial #2]

Rule of thumb: make one commit per logical change (ie. purpose).

# Repository

A repository is a collection of files that are tracked by Git as a cohesive unit - that is, all file are tracked at the same time.

This is useful if two files reference or are related to each other. When you make a commit in git, you save a version of every file in your repository.

A commit is a snapshot of every file in your repository at the time you choose to make a commit.

Example of a Repository:

|  |  |  |
| --- | --- | --- |
| **Commit 1 @ 1pm** | **Commit 2 @ 1:30pm** | **Commit 3 @ 4pm** |
| File 1 | File 1\* | File 1$ |
| File 2 | File 2 | File 2\* |

Use git log --stat to see a summary of which files were changed and the number of insertions/additions to each file.

Use git clone [url] to copy an entire repository (that is, both the files themselves and the full history of each file).

Use git checkout [commit serial#] to restore all files in the repository to the version a previous commit. This may be useful if there is a bug and you want to figure out which commit caused it.

## Creating a repository

1. Go to directory where you want to create a repository > cd [directory]
2. Create a .git directory > git init (note: when you initialise the repository, git does NOT create any commits for you).
3. Check that the directory is a git repository and see which files have changed since last commit > git status
4. Add all files you want to keep track of to the staging area > git add [file name]
5. Commit files that are in the staging area > git commit
6. Write a commit message – remember to do this as a command (ie. Use ‘Add’ instead of ‘Added’.

# Your working spaces

When using Git and repositories to track your changes, you have three working spaces:

|  |  |  |
| --- | --- | --- |
| **Working Directory** | **Staging Area** | **Repository** |
| File 1 🡪 | Add file to this area 🡪 when you are ready to commit changes  ( git add [file name] )  Once you’ve added all the files you want to include in your commit, you can commit to the repository  ( git commit ) | File 1 Commit |

You can check what changes you have made to a file (before commiting them) by comparing files in your Working Directory (which has your current changes) against the files in the Staging Area (which has the most recent commit files, if you haven’t added any files to the staging area). This can be useful if you have been working on changes in a particular file over separate sittings and you can’t remember all the changes you have already made.

To do this use > git diff (with no arguments)

|  |  |  |
| --- | --- | --- |
| **Working Directory** | **Staging Area** | **Repository** |
| File 1$  File 2 | File 1\*  File 2  When no files have been added to it, the staging area has most recent commit (Commit 2) | Commit 2:  File 1\*  File 2  Commit 1:  File 1  File 2 |
| git diff will show the difference between File 1$ and File 1\* | |  |

You can also check the difference between files in the Staging Area against your most recent commit (after you’ve added files to the Staging Area). This can be helpful before submitting your commit, to double-check that the changes you are committing are the changes you actually want to include in the commit, and that the correct files have been added.

To do this use > git diff --staged

|  |  |  |
| --- | --- | --- |
| **Working Directory** | **Staging Area** | **Repository** |
| File 1$  File 2 | File 1$  File 2  (After you have added files to the Staging Area) | Commit 2:  File 1\*  File 2  Commit 1:  File 1  File 2 |
|  | git diff --staged will show the difference between File 1$ and File 1\* | |

To discard any changes in your Working Directory or Staging Area, use git reset -- hard. **Be careful as this is irreversible.** Before doing this, be sure to run git diff and git diff –staged to see exactly what you will be erasing.

# Branches

Commits that build on each other make sense to stay on the same, linear branch:

O 🡪Fix bug

|

O 🡪 New feature

|

O 🡪 Update docs

(Master branch)

You can use branches to experiment with new changes, investigate and fix bugs, add and test new features, and so on. It allows you to use and play with the real, live code while still keeping a working version of it separately.

O 🡪Fix bug

|

O 🡪 New feature

|

O 🡪 Update docs

| \

| \

| \

| \

Bug fix 🡨 O O 🡪 New feature

(master branch) (experimental branch)

This way, while you are experimenting with code or fixing a bug, you can do it separate from the master without worrying about breaking something, and when the code is finalised and ready, you can merge it to the master branch.

* To see what branches you have > git branch

A \* indicates which branch you have checked out.

* To create a new branch > git branch [new branch name]
* To switch to a different branch > git check out [branch name]
* To visualise branches > git log --graph --online
* To delete branch name (when no longer needed) > git branch -d [branch name].

To merge separate branches and include all changes to both branches (for example, if you want to merge X branch into master branch), use git merge [branch name1] [branch name 2].

Note that when you are merging, you may have conflicts. This will happen when the same line has been changed in different ways. Git won't know which version to keep, so you will have to look at the changes made, decide how they should ultimately be merged and manually make changes to the file to combine the two changes.

# GitHub & Remotes

GitHub makes it easy to share entire git repositories with other people, and for people to collaborate on a project.

When you create a copy of your local repository in GitHub (which hosts repositories in the cloud), you will need to decide when to sync your local repository to the remote verion (that is, just like with commits, the syncing is not automatic).

A remote repository in GitHub is also referred to as a "Remote". Unlike your local repository, the "remote" does not have a working director and staging area - it only keeps the history of commits and branches.

To send (push) and receive (pull) commits, you will have to specify the branch. When you push or pull commits, Git will only push new commits (that is, it won't re-send old commits that are already in the repository).

To create a remote, first go to GitHub and create the repository there, then add the remote using git remote add [Name] [URL]. If there is only one remote, the standard name is 'Origin'. The URL is taken from the repository you created on GitHub

* To see a list of all your remotes set up and their URLs, use git remote -v
* Send changes in your local repository to the remote - git push [remote name] [branch name]

You (or someone else) can edit files directly on GitHub, or you/someone can also push changes from a different computer, so that your GitHub remote ends up with more recent commits that aren't in your local drive.

To sync your local with the remote Github version, you need to PULL the remote branch into your local repository, using git pull [remote name] [branch name] - usually this would be git pull origin master.

If there is a project on GitHub that you like and want to copy and make changes to, then you can make a copy of it to your GitHub and then clone it to your computer. This is called 'forking' (that is, cloning within GitHub). A fork is a clone that GitHub keeps track of. This means your repository will always be linked to the original.

Git pull is a combination of git fetch (which takes changes from the remote and creates a new branch on local drives) and git merge (which then merges the new branch together with the master branch on the local repository.

The local repository also keeps track of where on the branch the remote repository is at:

|  |  |
| --- | --- |
| **Local** | **GitHub (Remote)** |
| O 🡪 A origin/master  |  O 🡪 B  |  O 🡪 C (current master) | O 🡪 A master |

'origin/master' above reflects where the remote branch is at compared to the local branch. In the example above, commit B is made on the local repository, so that the branch master is now at point B. However, the position of 'origin/master' tells us that commit B has not yet been pushed to the remote. Thus, the local branch is "ahead of origin/master ".

Running git fetch will update the local copy of the remote branch (that is, bring in any new commits that are in the remote but not in the local repository) without affecting the local master branch -- it does this by creating a new branch on your remote with any new commits from the remote:

|  |  |
| --- | --- |
| **Local** | **GitHub (Remote)** |
| O 🡪 A  |  O 🡪 B origin/master  |  O 🡪 C (new commit) master | O 🡪 A  |  O 🡪 B  |  O 🡪 D (new commit) master |

Above, origin/master on the local tells us that the last commit that has been pushed to the Remote was B, and that commit C is not yet in the remote. That is, up to commit B, the local and remote are sync'd. We can see then that there is commit C in the local that needs to be pushed, but also that there is commit D in the remote that has not been pulled into the local.

Running git fetch would create a new branch on the local repository which brings in the commits in the remote that are not already in the local (that is, commit D) without affecting the master branch.

|  |  |
| --- | --- |
| **Local** | **GitHub (Remote)** |
| O 🡪 A  |  O 🡪 B  | \  | \  | \  | O D origin/master  |  O 🡪 C master | O 🡪 A  |  O 🡪 B  |  O 🡪 D master |

If we wanted to combine the changes from commit C and D into the master branch, we would run git merge which would result in:

|  |  |
| --- | --- |
| **Local** | **GitHub (Remote)** |
| O 🡪 A  |  O 🡪 B  | \  | \  | \  | \  | \  | O D origin/master  | \  O 🡪 C \  **\ /**  **\ /**  **\ /**  **\ /**  O 🡪 E master | O 🡪 A  |  O 🡪 B  |  O 🡪 D master |

Commit E would contain the changes in both commits D and C. The effect of the git fetch and git merge is the equivalent of git pull.

Pull requests (aka "merge request) are a feature of GitHub which is used to submit changes (commits) to someone else for review before they are merged into the master branch of a repository.