**Git config –global user.name** -> before starting with commits

**Git config –global user.email** -> before starting with commits

**Git clone** -> download a git repository

Downloading a git folder from the website results in no presence of the (hidden) folder .git. If ‘git clone’ is used from bash the hidden folder is present.

**Git log** -> show all the changes for a certain project. Keep pressing the key ‘SPACE’ to go throught it till ‘END’ appears. At that point, to exit the log lo press either another time SPACE or ‘q’

**git log -- *file\_to\_log*** -> show all the committed versions of *file\_to\_log*

**git log –oneline** -> show a more concise log, only with a short description of the main changes

**git log –oneline –all** -> show the whole list of changes, including also those that occurred after the current version (one could be active on a version that has been superseeded)

**git log --oneline --all --graph** -> show the list of changes with some graphical representation of the changes (forks, braches, etc.)

**git checkout ##### -- *file\_to\_checkout***-> restore in the working tree and staging area the *file\_to\_checkout* from the commit with hash denoted by the first 5 characters #####. After this action the *file\_to\_checkout* is ready to be directly re-committed (on a new commit).

**git checkout -- *file\_name*** -> restore a previous version of a file that has been modified but not staged. This is how to undo an unstaged working tree change.

**git checkout -- <path>** -> remove unstaged changes so the files go back to before modification

**git checkout *fd7ec2a*** -> activate a previous version (commit) of the project (identified by its hash). It is also used to switch from one folder to another, for example from the master folder to a branch

**git checkout -b *branch\_name*** -> creates and activates *branch\_name* in one go. It combines git branch *branch\_name* + git checkout *branch\_name*.

**git diff** -> show difference between working tree and index (staging area)

**git diff -- staged** -> show differences of the latest staged versions and latest commit i.e. the differences that are ready to be committed

**git diff *fd7ec2a 055e29b*** -> show the differences between 2 commits

**git init** -> initialise a repository

**git status** -> get info about the status of the files in a certain repository

**git add *file\_to\_add*** -> put the file “file\_to\_add” to the staging area

**git add .** -> adds all files (excluding those specified to be ignored)

**git rm *file\_to\_remove*** -> remove a file both from the working tree (the folder) and from the staging area (index)

**git rm –cached *file\_to\_unstage*** -> remove file(s) from staging area. Example: git rm –cached index.html

**git rm --cached -r .** -> remove all files from the staging area (index). This can be used to change a commit. I have used it to solve this problem: I wanted to ignore .pot files and to do so I created a .gitignore in which the .pot extention was listed. However this works only if the .pot files had not been previously committed, which was the case. In other words, I created a .pot file first, committed the repo and then added the .gitignore (with .pot in it) to ignore the .pot files: this operation was unsuccessful because the .pot was already present and committed before the .gitignore. So I had to go back and remove all files from the commit with *“git rm --cached -r .”*

**git reset -- *file\_to\_unstage*** -> AS ABOVE remove file(s) from staging area. Example: git reset -- index.html

**git reset** -> unstage all files from the index

**git reset HEAD** ***file\_to\_unstage*** -> replace staged *file\_to\_unstage* with the latest committed version of that file. It changes the file on the index but not on the working tree.

**git commit** -> commit all files in the working folder. TO BE NOTED: if the files that are being committed are closely related they can be committed together (same commit) otherwise there should be separate commits (and readme) for every file.

|  |
| --- |
| Working directory |
| Staging area |
| Repository |

After the commit command is entered an editor will appear asking for a description of the commit. The description can be made on the first lines and does not need to be on quotes ‘ ’ or marked as a comment (for python with # at line start)

**git commit -m *“message\_to\_type”***-> commit with the possibility to directly type a message

**git commit –amend -m “*message\_to\_type*”** -> instead of creating a new commit

**git reset --soft <commit>** -> unstage commits. Using –soft argument changes are kept in th working directory and index (they are not in the Git repo anymore)

**git reset --hard <commit>** -> unstage commits on Git and discard all changes. All changes will be lost

**git add --patch** -> add only a certain part of a script. The reason is because one could want to keep separate commits for different meaningful additions. Once run the command Git will ask for each of the additional parts of the modified file if they have to be added or not. The user will choose.

**git branch *branch\_name*** -> create a branch for the existing project. The branch can be made with the scope of keeping separate a certain area of development, that could be a code section

**git branch -d *branch\_name*** -> delete branch named *branch\_name.* This deletes only branches that are merged

**git branch -D** ***branch\_name*** -> deletes branches that are not merged

**git checkout *branch\_name*** -> activate and go into a branch specified by branch\_name

**git checkout -b *branch\_name*** -> shortcut to create and check out a new branch called *branch\_name*

**git branch -- merged** -> shows the branches that have been merged. This can be a useful check to do before deleting a branch

**git merge *branch\_name*** -> merging a branch into the current branch (typically master (?)). This way the code in the branch is also included in the main code

**git merge --abort** -> abort branching, in particular when there are unresolved conflicts

to let git ignore files or folders a txt file called **.gitignore** has to be created, where the name of ignored items can be listed. If git status is called then it will not include in the list the files/folders that should be ignored

**git remote add <name> <url>** -> add a github (remote) repo to a local repo with a user-defined name <name>

**git remote add *origin* *url*** -> add the github (remote) repo to local repo. *Origin* is a conventional name to indicate the repo at that *url*

**git push *origin master*** -> push the commit from the local repository *master* to github repository *origin*

**git fetch *origin*** -> send modification from remote repo to local repo i.e. notifies on updates to the github repo that might not be yet implemented in the local repo

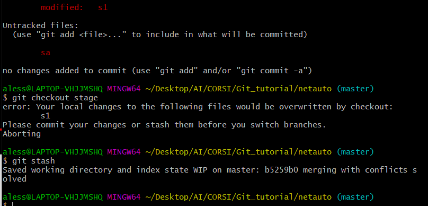
**git merge *origin/master*** -> merge the remote with the local repo

**git merge <name>/master** -> merge the remote with the local <name> repo. This could be in case the original github repo has been forked but a connection with the original needs to be maintained (and synchronised with the changes on the main one). For this reason a <name> local repo referring which can enable to implement the changes on the main(or original) repo to the forked version.

**git pull** would have done the same as git fetch origin + git merge origin/master. However is better to keep them separate to avoid unexpected merges.

**git push --follow-tags** -> push tags on Github

**git stash** -> save changes to a file thus allowing to switch to another branch without committing the changes on the current working branch. It’s sort of saving without staging or committing. The stashes are not deleted even after a commit so they can be reused.



**git stash list** -> list all stashes

**git stash list -p** -> shows the files with modification highlighted

**git stash apply** -> apply the most recent stash

**git stash save “*message*”** -> stash with message “message” that will be displayed in the list of stashes when ‘git stash list’ is called

**git remote -v** -> show the full location of a remote repository (on github)

**FAQ**

***How to know the status of files in the working tree and staging area?***

git status

***How to overwrite an un-staged working tree change?***

git checkout -- <filename>

***How to unstage a snapshot and replace it with the latest commit***

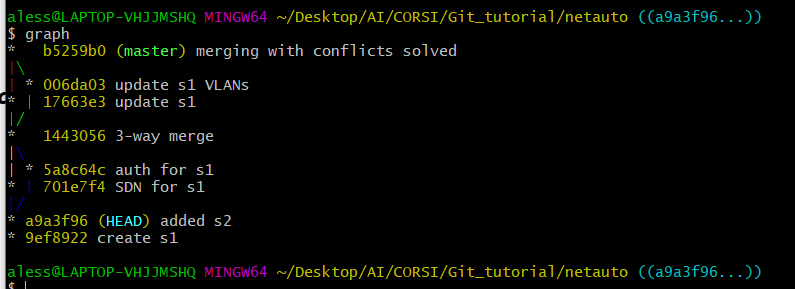
git reset HEAD <filename>

***How to remove a file from a repo (working tree and staging area/index)***

Git rm <filename>

**What is a ‘detached head’?**

When the pointer is referencing to a commit and not to a branch. Usually it happens after a ‘git checkout #####’



**How to get out of a detached head state?**

By checking out a new branch on the commit we are on

**What command is useful to deal with unfinished work that is not committed?**

git stash

**How to register a command with a name?**

Alias. Example: alias graph=’git log --all --decorate --oneline --graph’

**How to fetch from and integrate with another repository or a local branch? i.e. incorporate changes from a remote repository into the current branch**

git pull [<options>] [<repository> [<refspec>…]]

**Test questions**

List at least 3 different uses of git checkout

What is the command to fetch from and integrate with another repository or a local branch? Git pull

How do you

**TOPICS**

**Dealing with no clean working tree or staging area i.e. no modified files or stage changes not committed**

Git stash

Git remotes

After a clone a default name is assigned to a remote. To get the name:

**Git remotes**

To get the full location -> git remote -v

**Git fetch / merge**

Git does not automatically investigate what changes have been made to the remotes (GitHub). To get updated on what happens on the remote:

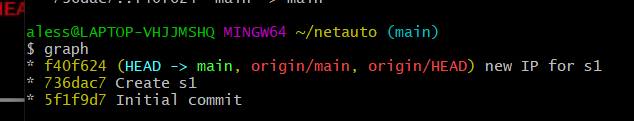
git fetch origin

to push a modification made locally to the github repo

git push origin main(or master)

|  |  |
| --- | --- |
| Before push | After push |
|  |  |

Forking github repos



**GLOSSARY**

*Branch:* labels that points to a commit

*HEAD*: pointer that indicates what is checked out

Detached head state: when a commit is checked out by its hash and the pointer is therefore not pointing to a branch anymore. However the current hash can be made branch. It is possible to get out of a detached head state by checking out a new branch on the commit we are on

Fast-forward merge: when there is a direct path between 2 branches

3-ways merge: when there is no direct path between 2 branches. It looks at the common base commit and

**RESOURCES**

Getting started with Git and GitHub: the complete beginner’s guide

<https://towardsdatascience.com/getting-started-with-git-and-github-6fcd0f2d4ac6>

Youtube tutorial VERY GOOD

<https://www.youtube.com/watch?v=uR6G2v_WsRA>