



Git

 `git commit -a -m "<commit-message>"` `git log / git log --oneline`


Displays the last few commits

 `git --amend`

If commit is already done, and there is another file to be added, `git add <file-name>`, then `--amend` (reverts the last commit), to commit all files together

 `git branch`


Overview of all branches

 `git switch <branch-name> / git switch -c <branch-name>`

First one switches between existing branches, second one creates new branch and moves HEAD to that branch

 `git branch -d <branch-name> / git branch -D <branch-name>`

Deletes the branch (HEAD must be on another branch)

 `git merge <branch-name>`

Merging two diverged branches


- Fast Forward merge: the parent branch does not change after the branching, while the child is ahead a few commits. With merging child to parent we sync all commits (HEAD must be at the parent/master, merge is done on the child branch)
- Merge with no conflicts: after branching changes are made both on child as well as the parent, but while merging there are no conflicts, as changes do not overlap
- Merge with conflicts: changes, made on both branches, overlap and must be resolved manually

 `git diff / git diff HEAD / git diff --staged [file-name]`

Lists all the changes in working directory that are not staged for commit


HEAD: lists all changes, staged or un-staged

`--staged`: lists the changes between staging area and last commit (shows what will be committed, if committed now)

 `git diff branch1..branch2 / git diff commit1..commit2`


Branch: comparison between two branches

Commit: comparison between two commits (commit hashes)

 `git diff HEAD HEAD~1`

Comparing the current HEAD to the previous commit


Changes made on two different branches: if un-committed, the changes transfer to the other branch upon switching (if committed, they stay on their original branch)

 `git stash / git stash pop / git stash apply`

Saves the uncommitted changes, without having to make a commit: undoes un-staged/staged changes in the working directory, so they do not transfer upon branch switching

pop: re-applies stashed changes to the working directory


apply: similar to pop, but enables to apply the stashed changes more times (pop can only re-apply it once)

 `git stash list / git stash apply stash@{1}`

First one lists all the stashed changes, while the other one pops the second stashed change, if there are more


 `git stash drop stash@{2} / git stash clear`

Deletes a particular or whole stash


 `git checkout <commit-hash> → git switch master HEAD`

Travels back to the commit-hash (detached HEAD state): HEAD points to the referenced commit, enables to check the file few commit ago

Switch master HEAD then returns to the latest master branch reference


 `git checkout <commit-hash> → git switch -c <new-branch-name>`

Branching off of a certain commit hash in the past (branch is based upon the HEAD, so all the changes after commit hash are not visible here)


 `git checkout HEAD~1`

~1: refers to the commit before HEAD, ~2: would refer to 2 commits before HEAD

Referencing the commit without the hash id

 `git checkout HEAD <file> / git checkout — <file> / git restore <file-name>`

Reverts the chosen file to the stage of the last commit, therefore it deletes all the changes made after the last commit


 `git restore —source HEAD~1 <file-name> → git restore <file-name>`

Restoring the file back two commits prior to HEAD, changes, if uncommitted, are lost

Using 'restore' reverts the file back to the last commit, where HEAD is, while uncommitted changes are lost


 `git restore —staged <file-name>`

If unwanted file was added with 'git add', using 'restore' removes it from staging, file is not tracked anymore


 `git reset <commit-hash> / git reset —hard <commit>`

'Reset' undoes/deletes the commits after the commit hash, while not losing the changes in working directory


—hard: while reverting/deleting the commits, it also deletes the changes from working directory

 `git revert <commit-hash>`

Similar to reset, but instead of deleting the commits, it makes a new commit, reverting/undoing the unwanted commits, therefore does not delete the commits

 `git remote add origin <GitHub-URL>`


Naming the GitHub repository URL 'origin'

 `git remote -v`

Lists the given name of GitHub repository and its URL


'Git push' works for the entire branch, not a single commit

'Origin/master' shows the master branch at the point of cloning a repository, 'master' can be ahead of it, if committed (and un-pushed) changes are made


 `git branch -r / git checkout origin/master`

First one lists all the remote branches, while the second one enables to view 'origin/master' state, without any later changes

'Git push' then updates the 'origin/main' and branches are not diverted anymore


 `git switch <remote-branch-name>`

Git sets up the new local branch, while it sets up to track the remote branch `origin/<remote-branch-name>`: it connects the cloned branches, which do not show before 'git switch', only 'origin/master' is visible

 `git fetch <remote> <branch>`

Downloads the changes from remote GitHub repository, but those changes are not automatically integrated into working files (enables access to the changes, but without merging into local repository)

'git fetch origin master': updated origin master branch will be created and changes, made to GitHub repository, will be visible, while local master branch does not change

 `git pull <branch> = git fetch + git merge`

Updates the current branch with the latest changes from local GitHub repository, merging them to working directory

Conflicts: resolve the conflict in VSCode, then add, commit and push ('git push origin <branch>') the changes to GitHub

 `git remote add upstream <GitHub-URL>`


Setting up the remote After forking a repository (fork makes a copy in time of the original repository) and cloning it to our machine, for keeping up with changes, made to the original repository, the remote upstream must be added (URL is url from the original repository, while naming a remote 'upstream' is a convention)

 `git pull upstream`

Downloads the latest changes made to the original repository

Git rebase is an alternative to merge, also used for cleanup

Rebasing is done from feature branch, as we want to merge latest changes on master branch into feature branch


 `git rebase master`

Instead of merge commit to feature, rebasing feature branch onto the master branch moves the entire feature branch so that it begins at the tip of master branch (rewriting history)


After rebasing new commits are made, but it cleans up the commit history as it is linear versus merging, which has two paths, which merge multiple times

Never rebase master branch that more people use

Interactive rebase: editing commits, adding files; rebasing commits onto the HEAD they are currently based on, while changing the commit hashes


 `git rebase -i HEAD~8`

Allowing to go back 8 commits: opens a text editor with selected commits in reversed order


 pick, edit, fixup, reword, drop ...

 `git tag / git tag -l "**beta**"`

Lists all tags in a repository / lists tag taht include 'beta' (* on both sides mean that there could be other optional characters on both sides)

 `git checkout <tag-name>`

Detached HEAD state (tag is only a pointer to a certain commit)


 `git tag <tag-name> / git tag -a <tag-name>`

Creating a lightweight tag: points to whatever the head is pointing at at that moment in time

Creating an annotated tag: holds more metadata about the tag

 `git tag <tag-name> <git-commit-hash>`

Tagging an earlier commit

 `git tag -d <tag-name>`

Deleting a tag

Reflog = reference log (local stored changes on HEAD)


 `git reflog show HEAD / git reflog show <branch-name>`

Shows all reference logs, with their identifiers (HEAD@{<number>}).

 `name@{qualifier}: git checkout HEAD@{2}`

Two moves in the past

Reflog is useful in case of hard reset; while everything to a certain commit hash is deleted locally (git log removes it), git reflogs are still there

 `git reset --hard <commit-hash> → git reset --hard master@{1}`

Hard reset can still be restored with reflog; using commit hash of the lost commit (still visible on 'git reflog show HEAD') or relative position of the move (master@{<number>})

After wrong rebase git reflog still shows all the commit hashes, so using 'git reset --hard <commit-hash>' works and restores the lost commit