Git Basics

Jan Rathfelder
Bayer: Data Science & Advanced Analytics, Powerhouse AI

February 14, 2022

Contents

1 Add, change, delete, rename

- **add + commit**: git commit -a -m 'description'. → combines git add and git commit in one line
- **git diff**: shows differences in code between last commit and current file (which has not been staged already!)
- **git diff --staged**: shows differences in code between last and current file when already staged (used git add already)
- **git rm uselessfile.py**: removes/deletes file from repository and your machine for unstaged files only. Commit after rm is mendatory
- **git rm -f uselessfile.py**: deletes file from repository and your machine even when in staging area
- **git rm --uselessfile.py**: deletes file only from repository, but file being kept on your machine
- **git mv example.py example_v1.py**: renames the file example.py to example_v1.py. Needs commit afterwards to be applied!
- **git restore example.txt**: Reverts changes and brings example.txt back to changes of last commit.
- **git restore --staged example.txt**: New and replaces reset HEAD. Reverts changes and brings file back to last commit.
- **git commit --amend -m 'forgot to add a file in the commit'**: Can be used to change a commit, if it has not been pushed already. You do: 'git add, git commit' and then realise you forgot something. Then you can do another 'git add' followed by 'git commit --amend', which combines the two commits and now you can push that.

2 Git history

2.1 Show previous commits

- **git log**: Shows complete history of all commits

- **git log -3**: Shows last three commits
- **git log -patch**: Shows changes between different commits. Can probably be combined with git log -3, but I have not tried it until now.
- **git log --stat**: Shows which files have been changed and number of lines added and deleted
- **git log --author="name of author"**: Shows only history of that specific author
- With --since, --after, --until, --before in combination with git log followed by a date filters in history

2.2 Restore old versions

- **git checkout 'Head-Hash-Value'**: Goes back to that commit.
- **git checkout 'Head-Hash-Value' example.py**: Goes back to the example file of that specific commit
- **git checkcout HEAD~3**: Also goes back in commits without using the head hash value. In this example, we go back 3 commits.

Now you can view the file you want to. If you want to continue from here and manipulate a file and save this, then you need to create a new branch. Also note that going back to the HEAD where you have been coming from is not possible using git log, because this only shows history below the commit you are on. If you want to go back to the latest commit (because remember git checkout Head-Hash-Value let's you have a look at previous commits, it is not actually saving this view unless you create a new branch) where you were coming from, then use: git checkout branchname.

- **git reset --hard 'Head-Hash-Value'**: Goes back to that commit while losing all commits after that commit you specified using the head hash

3 Tag commits

There is a difference between lightweight-tags and annotated tags. Lightweight-tags is just the name and the commit info, while annotated tags include information about the author, her/his e-mail address and the date. Lightweight-tags are mostly used if importance is low, while annotated tags are used for bigger projects and more important use cases in general.

- **git tag version1.0**: Lightweight-tag
- **git tag -a version1.1 -m "Version 1.1 of Impact Attribution"**: Annotated tag
- **git tag**: shows all tags
- **git tag -list version1.***: shows all tags that start with project1.
- **git show version1.1**: shows commit if tag version1.1

4 Branches

- **git branch**: shows all branches
- **git branch --merged**: shows all branches that have been merged (to the branch you are on)
- **git branch --no-merged**: shows all branches that have not been merged (to the branch you are on)
- **git merge example_branch**: merges branch example_branch into the branch you are on
- **git rebase master**: pulls the commits from master into the branch you are on (difference between merge and rebase is the difference in structure that happens. While merge is really integrating the different commits and keeping the right order and creating a real commit, rebase just puts the commits of the branch you are on on-top of the branch you are pulling. Recommended to use when you are on a branch and want to get commits from master. But after a rebase it might be tricky or even impossible to go back to your previous state, because no commit is created and therefore you can not go back in commit history)

5 Advanced functions

5.1 Advanced search

- **git grep elasticity**: searches in all files for the term elasticity
- **git grep --count elasticity**: searches in all files for the term elasticity and counts how often used (so it returns just a number)
- **git grep --line-number elasticity**: searches in all files for the term elasticity and returns the line and the line number (so you also see the code)
- **git grep --show-function elasticity**: searches in all files for the term elasticity and returns the context for the keyword. Likely it returns some part of the code
- **git grep elasticity counterfactual.py**: searches in the file counterfactual.py for the term elasticity → all commands above can be used file specific
- **git grep -e elasticity --and -e profit**: searches in all files for the term elasticity AND profit. Needs to be in the same line (as far as I understood it)
- **git log -S elasticity**: Looks for commits where the word elasticity can be found. Returns Head hash, author, date and commit message
- **git log -L :elasticity:counterfactual.py**: Finds all commits where the function elasticity in the file counterfactual.py has been changed (only works for functions as I understand it)
- **git log --all --graph**: produces a visualisation of the commit flow to understand different branches
- **git refLog**: returns a history of all actions you did in git (merges, commits...). Also returns a head ID (e.g.: HEAD@{1}), which can be used to go back in commits (e.g.: git checkout HEAD@{1})
- **git show HEAD~**: points to the previous commit for orientation (and information about Head hash etc.)

- **git show HEAD^2**: points to the commit before the previous for the current commit you have done (not so clear to me what the difference between ~ and ^ is)
- **git rebase i HEAD~3**: opens text editor and allows you to change the last 3 commits. Now you can decide on different options like pick reword, edit or drop (which would be a delete)

5.2 GPG key

Create a GPG-key to tag your commits with a crypto generated key no one can fake:

- **gpg --gen-key**: generates gpg key (for windows you need to be in bash)
- **gpg --list-key**: shows all available gpg keys
- **gpg config --user.signingkey 'yourkeynumber'**: activates the key for tagging
- **git commit -a -S -m 'compare linear vs constant elasticity'**: adds the signature to the commit, which is shown using git log or git show

5.3 Interactive staging

Interactive staging let's you break down a possible big commit for multiple files into smaller pieces. Let us see how:

- **git add -i**: shows you all files that have not been added
- **u**: once in the staging area, you have different options. Here we go with u for update. Next step see below
- **1, 3**: 1 and 3 means you want to update the files 1 and 3 (from the list of all possible files in the staging area)

Instead of u you can remove files from the staging area with r. There is a list of different actions. First you decide on the action and then pick the files where you want to perform that action on.

You can add files to the staging area that have not been added to the repository before (in previous examples we were performing changes to files that exist already in the repository) by selecting 4 or a (add untracked).

When you pick 5 or p (patched) then you can even slice a big file into different parts you want to commit. Git gives you pieces automatically. But even from these pieces you can create smaller chunks by typing s when git asks you if you want to stage the hunk (not so easy to describe without pictures, probably needs to be tried out to be fully understood)

5.4 stash and clean

Stashing:

- **git stash**: saves your work without creating a commit. Good practice if you are interrupted in between (and need to change branches) or just want to save your work in progress without creating a commit
- **git stash --include-untracked**: also included files that have not been staged yet
- **git stash list**: shows you the stashed files
- **git stash apply**: if you had to change the branch and are now back on the branch where you stashed, this brings the files to the saved status (for me it is a bit counterintuitive that we need to apply the stashing now...)
- **git stash apply@{1})**: goes back to the one before the last stashed version (and etc...)
- **git stash apply --index**: now also files that have been added already are preserved (and can be found in the staging area once you return to the branch). This also allows you to change a branch and use the file you added before on the other branch by running git stash apply on that new branch.

Cleaning:

- **git clean -n**: Does a dry-run cleaning. Before you apply actual cleaning have a look first which files will be deleted

- **git clean**: Cleans the repo and deletes all untracked files (no turning back!) in master branch item **git clean -d -f**: Cleans the repo and deletes all untracked files (no turning back!) in all branches

5.5 Advanced merging

There are different strategies embedded in Git, which you can apply by using -s. Below are some examples not the full list:

- **git merge -s octopus**: Merges more than two branches automatically
- **git merge -s ours**: Ignores changes from the other branch completely

Additionally there are different merge options, which can be used together with merge strategies. Below one work around for solving conflicts:

- **git merge -Xours branch2**: Tries to merges branch you are on and branch2. If there are conflicts, then the changes from branch2 are discarded
- **git merge -Xtheirs branch2**: Exact opposite of -Xours. Tries to merges branch you are on and branch2. If there are conflicts, then the changes from branch2 are used
- **git merge -Xpatience branch2**: Applies a different kind of algorithm for merging. Is slower than regular merging, but might be beneficial for branches that have diverged a lot

If there are conflicts which you cannot or don't want to resolve immediately then it might make sense to abort the merge:

- **git merge --abort**: Aborts a merge

Sometimes it might make sense to revert a merge. There are different ways to revert a merge, depending how many commits you have made after the merge:

- **git reset --hard HEAD~**: Could be used directly after a merge to revert the merge, as it goes back one commit, which includes the previous merge. But if there have been commits after the merge they are lost

- **git merge -m revert 1 HEAD**: Reverts a last commit by creating a new commit and achieving the old unmerged status. The -m 1 refers to the branch you are on, which means you can create the exact status as before. By applying HEAD~ and specification of commits, you can go to any previous commit you want to (but again: a new commit will be created)

5.6 rerere

rerere stands for reuse recorded resolution and can help to resolve conflicts. If there is a conflict you can record the solution and apply this solution again at a later stage. Here are the steps you need to follow:

1. **git config --global rerere.enabled true**: rerere recording enabled. If there are conflicts, recording starts immediately

2. If there is a conflict, solve that manually to show rerere how you want it to be solved in the future

3. But even though the conflict solving strategy is applied, there will be an error message that conflicts. But if you go into the files, you will see that the files actually have been changed