Git for normal people, the cheatsteet v1.1

More info and courses from CodeRefinery, a Nordic project: https://coderefinery.org/lessons/

Git is a version control system (VCS) used to track changes to files. By using git, you can go back to any previous version of your files. Once something is committed once, it is very, very hard to ever lose it. Without a VCS, work is not reproducible, not easily shared, not recoverable, and it is not easy to find out when a bug got introduced.

The first side of this cheatsheet shows the most basic use, which should be used in *every* project. Each level can be used without knowing anything of the later levels. If you only use exactly the commands here, you will never get to a state you can't recover. Some commands do very different things depending on options, so check carefully first.

Terminology: *working directory* is the actual files you see right now. *Branch* is a distinct line of work (*main* or *master* is usually the default branch). A *commit (verb)* saves a version, and *commit (noun)* also refers to that distinct version. Versions are identified by *hashes* (like f3415b). The *staging area* is a place to collect changes before committing.

Basic principles to prevent problems:

- Commit often
- Always commit before merging, pushing, pulling, etc.
- Check info commands often. You can see what is happening and you will understand git better.
- Push and pull early and often. Less divergence is less chance of conflict.
- All files in a repository should either be under version control or ignored using *.gitignore*.

Basics

- **git init** - Create repository in current dir
- **git clone $url [$directory]** - Create a linked copy of other repo.
- **git help $cmd** - Help on any subcommand
- **git config** - Set common configuration
  - **git config --global user.name "Your Name"**
  - **git config --global user.email your@email.net**
  - **git config --global core.editor nano** - or your text editor
  - **git config --global alias.graph "log --all --oneline --decorate --graph"**

Info commands

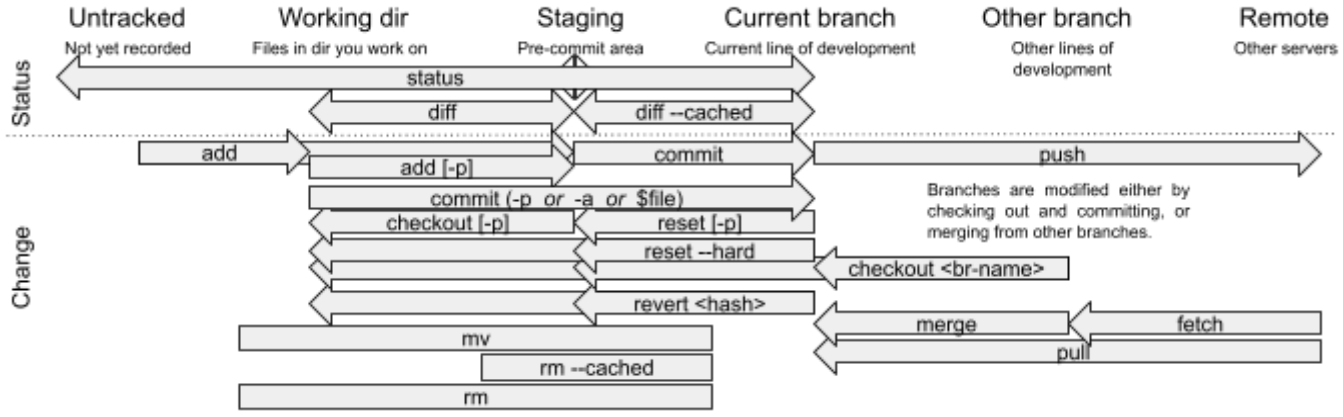
Before and after every Git command, run these info commands to understand what is going on. ★=important

Info about what is currently changed:

- **git status** ★ - What is uncommitted and about to be committed? Shows working, staging, and already committed.
- **git diff** ★ - What changes are currently unstaged?
- **git diff --cached** - What is already *staged* (about to be committed, see level 1) with *git commit*.

Info about entire history:

- **git graph** ★ - Show current commit graph. Alias defined above.
- **git log** - More detailed log of changes. Also **--stat** to show what changed and **-p** to show exact changes.
- **git log $file** - Log of changes to single file.
- **git show $hash** - Show changes and msg in a commit.
- **git show $hash:$filename** - Show old contents of file.
- **git diff $hash** - Show changes between hash and now.
- **git diff $hash..$hash** - Show changes between commits.
- **git blame $file** - show last edit of every line in file.



Level 0: commit everything

The simplest usage is to commit everything. Your cycle is: **add** (first time), **[status]**, **[diff]**, **commit**.

Start by telling files to track:

- **git add $file** ★ - Tell git a file is tracked. Run once on every file.

Then you can get info about current changes:

- **git status** ★ - What is not yet committed or ignored?
- **git diff** ★ - What exactly are those changes?

Then you can commit things:

- **git commit $file** ★ - Commit all changes in a single file.
- **git commit -a** - Commit all changes in tracked files.
- **git commit -m "some message"** - Give message on command line; use with either of the above.

Then you can browse history:

- **git graph** - Show full concise history graph (see col 2 "Basics")
- **git log** - Verbose log.
- **git show $hash** - Show changes in any one version.
- **git show $hash:$file** - Show old version of file.

Doing other management:

- **git mv $a $b** - Move $a to $b and stage the change (commit after).
- **git rm $a** - Remove $a from working copy and stage (commit after).
- **git rm --cached $file** - Remove file from staging and git, leave working copy (commit after).
- **git revert $hash** - Undo a change by applying opposite change. You must commit everything first.

Level 0.5: selective commits

Similar to level 1, but you can commit only some lines at a time instead of whole files. This allows you to have "one commit, one purpose" which makes your code more accurate. From now on, **-p** always means "ask me line by line".

- **git commit -p** ★ - View all changes and decide which to commit
- **git commit -p $file** - Like above, but single files.

.gitignore

Put a file **.gitignore** in the repository: each line is one file pattern which should never be committed. Then, your **git status** output will be useful. All files should be tracked or ignored, so keep .gitignore up to date. The .gitignore file should be part of the repository to make sure all collaborators have the same **git status** output.

http://version.aalto.fi http://scicomp.aalto.fi

Level 1: staging

Git has a "staging" (pre-commit) area which you can use to *stage* files before committing. This provides more organization and reduces chances of mistakes. Your cycle becomes **add** (first time), (edit), **[status]**, **[diff]**, **add**, **[diff --cached]**, **commit**.

Get info:

- **git status** ★ - Shows both changes and staging area. More important now!
- **git diff** - With staging, show diff of *unstaged* changes.
- **git diff --cached** - View difference between last commit and staging area.

Manage staging:

- **git add [-p] $file** ★ - Stage a file. With **-p**, decide on each change individually.
- **git commit** ★ - With no arguments, commit what is staged.
- **git checkout [-p] $file** - Revert working dir to last staged.
- **git reset [-p] [$file]** - Discard changes to staging area (go back to last commit). Don't touch working dir files.

Level 2: remotes, pushing, and pulling

Up to now, you have just tracked history locally. You can add remotes and share with others. You *push* changes to a server, and *pull* changes from server. Others do the same and you share code. If people modify same line, there is a *conflict* and you should check next column. For password-less pushing and pulling, use ssh keys (see “Advanced”).

Get info and setup:

- **git clone $url** - Create a new repo automatically linked to $url.
- **git remote add $name $url** - Add a remote to existing repo.
- **git remote [-v]** - List remotes.
- **git graph** ★ - Full commit graph, remotes labeled (see "Basics").
- Make SSH keys for automatic login: see "advanced".

Send and receive code:

- **git fetch** ★ - Check what is new on remote side (**git graph** to see). After fetch, you can merge as if it was a branch using **git merge $remote/$name**, see next section.
- **git pull** ★ - Gets remote changes and merges them immediately. Equivalent to *fetch* then *merge*, and for complex work it is recommended to *fetch*, check changes, then *merge*.
- **git push [$remote] [$branch]** ★ - Send changes to remote. Use **-u** first time to set default upstream. Pull before you push. In fact, pull before you start new work.

Level 3: branching

Branches are no different than remotes, and in fact branching should come before remotes (but we do it second since you will probably push and pull your own projects first).

Get info:

- **git graph** ★ - Full commit graph, all branches labeled (see "Basics")
- **git branch [-a] [-v]** - List all branches.

Manage branches:

- **git branch $name [$hash]** ★ - Create branch here or at some hash. Don't switch to the new branch yet.
- **git checkout $name** ★ - Switch to branch $name. Adjusts your current working files. Recommendation: commit first.
- **git checkout -b $name** - Create branch $name and checkout immediately.
- **git merge $name** ★ - Merge other branch changes into current branch
- **git branch -d $name** ★ - Delete a branch. Use **-D** if to force if it is not merged yet.

Conflict resolution

A conflict occurs when you try to *merge* changes that modify the same lines differently than modifications on the current branch. A conflict may seem scary, but by following these steps *resolving* it is easy. *Minimize conflicts by making small, frequent commits once something works and pushing/pulling often.* Commit everything before merging/pulling.

Get info (run these repeatedly while resolving!):

- **git status** ★ - Shows unmerged files.
- **git diff** ★ - Shows changes on both sides.

Doing the merge:

- Edit the conflicting files and look for the "==", "<<", ">>". Remove them and make the surrounding area look OK.
- **git add** ★ - Mark file resolved and add to staging.
- Double-check status and diff commands again.
- **git commit** ★ - Create a commit and finish up the resolution.
- **git mergetool** provides graphical help, first search web and set it up.

Giving up:

- **git checkout --theirs $file** - Revert to other version of file.
- **git merge --abort** - Give up merge, back to previous state.

If you get stuck and need to go back

If you do get to a place where you can't recover, you can always revert back to a previous good state:

- **git checkout -f $branch** - Return to other branch and discard all changes, e.g. main/master.
- **git reset --hard $hash** - Discard all changes, revert whole current branch to past hash. Use **HEAD** for $hash for the last committed version.
- **git reset --mixed $hash** - Revert whole branch to given hash, but leave all working files the same (so you don't lose your work).
- **git checkout [-p] $hash $file** - Change work dir file to old version.

Stashing

If you need to hide some changes away without doing something as permanent as a commit, you can *stash* them.

Get info:

- **git stash list** - List previous stashes.
- **git stash show [-p]** - Show contents of last stash.

Use the stash:

- **git stash** ★ - Stash changes.
- **git stash save $name** - Stash with a name to remind you of what it was.
- **git stash pop [$id]** ★ - Apply stash. If conflicts, merge them using "conflict resolution" section, then remove later with **git stash drop**.

Advanced

- Git isn't good for large binary files. Use .gitignore and manage separately, or check out git-LFS or git-annex.
- **git commit --amend** can change the last commit. Use with care and only if you haven't pushed or merged yet. Also takes **-p**.
- Use aliases to save time: **git config --global alias.X "yy --z"**.
- **git tag -a $name [$hash]** - add an annotated tag at present point or given hash. Names important revisions.
- **git rebase** allows you to powerfully move code and branches around. Modifies history, don't use this unless you know what you are doing. Most people can live without.
- Committed to the wrong branch? **git cherry-pick** commit to the right branch, then “rewind” the other branch back with **git reset --hard**.
- SSH keys: generate with **ssh-keygen**. Public key is in *~/.ssh/id_rsa.pub*.For a step-by-step guide see https://help.github.com/articles/connecting-to-github-with-ssh/.