

Git_Commands.md

Description:

Introduction:

Github is an open-source code platform that allows us to collaborate with others developers through [Git](#), the most popular decentralized VCS (Version Control System), although, there are other VCS like [Subversion](#) for a more centralized approach.

This is **quick guide to collaborate** on Github through Git but mainly as a reference for **basic/other commands** which are meant to be used on a git terminal.

While not so user friendly, using the **terminal** allows us to use *more* commands, to *modify* their *syntax* and to provide them with many different *options, arguments or objects* and to *chain several commands successively* on **unix-like** systems.

The most popular ones are MAC OS and Linux, but Windows users can also use the [Gitbash](#) emulator provided by Git with its download.

Table of Contents

- [Github Contributions](#)
 - [Fork vs Clone](#)
 - [Contributions without permissions](#)
 - [Contributions with permissions](#)
- [Git Commands](#)
 - [Definitions](#)
- [Other Commands](#)
 - [Contributors](#)
 - [References](#)

Github Contributions



Fork vs Clone:

- **Fork**: Merge with original repo is possible with a pull request.
 - **Clone**: Merge with original repo is only achieved by pushing to fork and then a pull request.
-

Contributions without permissions:

Note: It is better to fork a repository before cloning it due to [copyrights](#) when the *user is NOT declared as a contributor*.

General steps:

1. [Fork](#) repository.
 2. [Clone](#) forked repository.
 3. Make Changes in Local.
 4. [Push](#) to Personal Remote.
 5. [Pull Request](#) to Original Remote.
-

Contributions with permissions:

Note: It is a faster option to clone the original repository without a previous fork of the project if the *user IS declared as a contributor*.

General steps:

1. [Clone](#)
2. Make Changes in Local.
3. [Push](#) to Personal Remote.

Refer to Github official [documentation](#) for more information related to contributions.

Git Basic Commands



The following is a list of common git commands based on the [Git Documentation](#).

Note: if you don't understand a term, check out the *definitons* section below.

Basic	Command	Description
1. help	1. git help 2. git help <code><command></code> 3. git help -a	1. List common commands . 2. Display help on git command . 3. List all available commands .
2. init	1. git init 2. git init -b <code><branch></code> 3. git init <code><subdir.></code> 4. git init --bare <code><subdir.></code> 5. git init --template= <code><template-dir.></code> 6. git init --shared <code>[=(-options)]</code>	1. Initialize git repo in folder. 2. Override branch name (<i>config. default set if none</i>). 3. Initialize a git repo inside a new subdir. 4. Initialize a git bare repo inside new subdir. 5. Specify dir. from which templates will be used. 6. Make git readable/writable by users (see options).
3. clone	1. git clone <code><URL></code> 2. git clone --no-hardlinks <code><dir.></code> 3. git clone <code><URL></code> <code><dir.></code> 4. git clone <code><URL></code> --branch <code><branch></code> --single-branch	1. Clone remote default branch with URL . 2. Clone local repo for <i>backup</i> purposes.

Basic	Command	Description
	5. git clone --bare 6. git clone --mirror 7. git clone --template= <temp_dir.> <dir.> 8. git clone --depth= <depth>	3. Clone remote default branch in dir . 4. Clone remote single branch with repo URL . 5. Clone remote with no remote-tracking & config . 6. Clone --bare with remote tracking & config.. 7. Clone set template in dir. (see 2.git init). 8. Clone truncated to a number of revisions .
4. config	1. git config 2. git config --global pull.rebase true 3. git config --global ff no 4. git config ff no 5. git config remote.origin.prune true 6. git config --global fetch.prune true 7. git config --global user.name <username> 8. git config --global user.email <e-mail> 9. git config --system user.name <project> 10. git config --get user.name 11. git config -l 12. git config -e	1. Display git global config (create if none). 2. Set the pull command as rebase globally. 3. Disable fast-forward merge for local repos. 4. Disable fast-forward merge in local repo. 5. Set auto-prune with fetch & pull . 6. Set auto-prune w/ fetch for local repos.

Basic	Command	Description
		<p>7. Set author to commits for local repos.</p> <p>8. Set email to commits for local repos.</p> <p>9. Set author for all git users.</p> <p>10. Get author/email from global/system.</p> <p>11. List all variables set in config. file.</p> <p>12. Edit config files from global/system</p>
5. checkout	<p>1. git checkout <branch></p> <p>2. git checkout -b <feature></p> <p>3. git checkout -b <branch> <origin/branch></p> <p>4. git checkout -- <file></p> <p>5. git checkout -</p> <p>6. git checkout <branch>~n <file></p>	<p>1. Switch to branch in working tree.</p> <p>2. Create and switch to feature (or any) branch.</p> <p>3. Clone remote branch and switch.</p> <p>4. Discard changes in file to match current branch.</p> <p>5. Switch to last checkout.</p> <p>6. Reverts local file in branch n commits (e.g. $n=2$).</p>
6. fetch	<p>1. git fetch <origin></p> <p>2. git fetch <origin> <branch></p> <p>3. git fetch --all</p> <p>4. git fetch --dry-run</p> <p>5. git fetch --append</p>	<p>1. Fetch all.</p> <p>2. Fetch branch.</p> <p>3. Fetch all branches in repo.</p> <p>4. Show output but without fetching.</p>

Basic	Command	Description
	6. git fetch --depth= <code><depth></code> 7. git fetch -f 8. git fetch --prune	5. Fetch <i>without overwriting</i> (.git/FETCH_HEAD). 6. Limit fetching to <i>n depth</i> commits (e.g. <i>n=3</i>). 7. Fetch even if it's <i>not descendant</i> of remote branch. 8. Remove <i>unexistant remote-tracking</i> branches.
7.merge	1. git merge <code><branch></code> 2. git merge <code><branch></code> <code><target_branch></code> 3. git merge --no-ff <code><branch></code> 4. git merge --continue 5. git merge --allow-unrelated-histories 6. git merge -base [-a] <code><commit_id></code> <code><commit_id></code> 7. git merge -s resolve <code><branch-1></code> <code><branch-2></code> 8. git merge -s recursive -X ours OR theirs <code><branch></code> 9. git merge -s octopus <code><branch-1></code> <code><branch-n></code> 10. git merge -s ours <code><branch-1></code> <code><branch-n></code> 11. git merge -s subtree <code><branch-1></code> <code><branch-2></code>	1. Fast-forward merge branch with <i>HEAD (linear)</i> . 2. Fast-forward merge branch to tip of <i>target</i> . 3. Maintain commit <i>history</i> , may not fast-fwd. 4. Conclude <i>conflicting merge</i> . 5. Merge <i>indep. projects</i> by overriding safeties. 6. Find ancestor on <i>n commits</i> for a 3-way merge. 7. 3-way merge 2 branch HEADs. 8. 3-way merge >1 <i>common ancestors</i> for tree. 9. Merges <i>more than 2</i> branch HEADs.

Basic	Command	Description
		<p>10. Merges <i>multiple branches tip</i> in HEAD.</p> <p>11. Reflect B tree <i>structure</i> as <i>subtree</i> of A.</p>
8. rebase	<ol style="list-style-type: none"> 1. git rebase <base> 2. git rebase -i <base> 3. git rebase --continue 4. git rebase --abort 5. git rebase -i HEAD~n 6. git rebase --onto <newbase> 7. git rebase --allow-empty 8. git rebase --edit-todo 9. git rebase --stat 10. git rebase -p 11. git rebase <branch> -s <strategy> 12. git rebase <branch> -s <recursive> -X <option> 	<ol style="list-style-type: none"> 1. Rebase branch into <i>base</i>. 2. Rebase branch <i>interactively</i> on <i>base</i>. 3. Continue rebase after <i>resolving merge conflict</i>. 4. Abort & <i>return HEAD</i> to original position. 5. Interactive rebase of <i>last n commits</i>. 6. Rebase branch into <i>base</i> other than upstream. 7. Allow empty msg <i>commits</i> to be created. 8. Edit the <i>list of commits</i> to be <i>rebased</i>. 9. Show diffstat of what changed upstream. 10. Recreate commits instead of flattening. 11. Use the given merge strategy.

Basic	Command	Description
		12. Use recursive merge with a valid <i>option</i> .
9. pull	<ol style="list-style-type: none"> 1. git pull 2. git pull <URL> 3. git pull <origin> <branch> 4. git pull --rebase <origin> <branch> 5. git pull --ff-only 6. git pull --no-ff 7. git pull -s <strategy> -X <option> 	<ol style="list-style-type: none"> 1. Fetch & merge <i>remote-tracking</i> with local. 2. Clone, fetch & merge remote's <i>URL</i> with local. 3. Fetch & merge <i>remote branch</i> with local. 4. Fetch & rebase <i>branch</i>. 5. Update <i>branch</i> without a merge commit. 6. Pull & commit even for <i>fast-forwards (linear)</i>. 7. Same strategies and options as for merge last 5.
10. add	<ol style="list-style-type: none"> 1. git add -A 2. git add . 3. git add <file> 4. git add -n <file> 5. git add --v 6. git add -force 7. git add -p 8. git add -i 9. git add -e 	<ol style="list-style-type: none"> 1. Add <i>all changes</i> in files to stage. 2. Add <i>changes</i> without <i>deletions</i> for stage. 3. Add <i>file</i> to stage. 4. Show if <i>file</i> is <i>unexistent</i>. 5. Ignore indexing <i>errors</i> for git add. 6. Allows to add <i>ignored</i> files.

Basic	Command	Description
		<p>7. Patch hunks <i>interactively</i> from <i>index to tree</i>¹.</p> <p>8. Patch changes <i>interactively</i> from <i>index to tree</i>.</p> <p>9. Interactive patch mode vs diff editor.</p>
11. commit	<ol style="list-style-type: none"> 1. git commit -m <code><msg></code> 2. git commit --date= <code><date></code> 3. commit -i <code><msg></code> 4. git commit --dry-run 5. git commit -v 6. git commit --amend 7. git commit -s 	<ol style="list-style-type: none"> 1. Overwrite commit <i>msg</i> . 2. Override author's <i>date</i> in commit. 3. Commit <i>changes</i> & <i>unstaged</i> content. 4. List only <i>committed</i>, <i>uncommitted</i> & <i>untracked</i> paths. 5. Show differences between <i>HEAD</i> and <i>commit</i>. 6. Modify the most <i>recent</i> commit <i>msg</i>. 7. Add <i>author signature</i> at the <i>end</i> of <i>commit msg</i>.
12.push	<ol style="list-style-type: none"> 1. git push 2. git push -u <code><origin></code> <code><branch></code> 3. git push --all 4. git push <code><origin></code> --delete <code><branch></code> 5. git push --force 6. git push --force-with-lease 7. git push --prune <code><origin refs/heads/*></code> 8. git push --mirror 	<ol style="list-style-type: none"> 1. Push <i>commits</i>. 2. Push <i>commits</i> and set as <i>upstream</i>. 3. Push <i>all</i> <i>commits</i>. 4. Delete <i>remote-tracking</i> branch. 5. Push commits and <i>destroy all unmerged</i> changes. 6. Push and <i>destroy personal unmerged</i>

Basic	Command	Description
		<p>changes.</p> <p>7. Remove <i>remote</i> <i>without local counterpart</i>.</p> <p>8. Overwrite <i>remote</i> <i>with local</i> branches.</p>
13.pull request	1. git request-pull <branch> <URL> <feature>	1. Pull request for changes between tag and feature.
14. branch	1. git branch 2. git branch -r 3. git branch -a 4. git branch <branch> 5. git branch -d <branch> 6. git branch -D <branch> 7. git branch -f <branch> <feature> 8. git branch --show-current 9. git branch --set-upstream-to 10. git branch / grep -v <branch(es)> / xargs git branch -D	1. See <i>local</i> branches. 2. See <i>remote</i> branches. 3. See <i>local and remote</i> branches. 4. Create <i>branch</i> and <i>name it</i> . 5. Delete <i>unmerged</i> branch. 6. Delete <i>merged & unmerged</i> branches. 7. Rewrite local <i>branch</i> with <i>feature</i> branch. 8. Show <i>current</i> branch in local. 9. Make an existing git branch track a <i>remote</i> . 10. Delete <i>all</i> branches <i>excepting</i> selected.
15. diff	1. git diff 2. git diff --staged 3. git diff HEAD	1. Check for differences in <i>local</i> & <i>remote-tracking</i> .

Basic	Command	Description
	4. git diff --color-words 5. git diff <branch> <feature> <file> 6. git diff <commit_id> <commit_id> <file> 7. git diff --stats 8. git diff-files 9. git diff stash@{n} <branch>	2. Check for differences in <i>local</i> & <i>staged</i> changes. 3. Check for differences in <i>work dir.</i> & <i>last commit</i> . 4. Highlight <i>changes</i> with <i>color</i> granularity. 5. Check for differences in a file between <i>two branches</i> . 6. Check for differences in a file between <i>two commits</i> . 7. Show <i>insertions</i> & <i>deletions</i> in <i>staged</i> and <i>local</i> . 8. Compare <i>files</i> in the <i>working tree</i> ² . 9. Compare <i>stash n</i> with <i>branch</i> .
16. log	1. git log -n 2. git log --oneline 3. git log -p --follow -- <file> 4. git log --oneline --decorate 5. git log --stats 6. git shortlog 7. git log --pretty=format:"%cn committed %h on %cd" 8. git log --after= <yyyy-m-d> --before= <yyyy-m-d> 9. git log --author= <username>	1. Display <i>logs</i> from last 1,2,... <i>n</i> commits. 2. Show <i>IDs</i> from commits. 3. Show <i>commits</i> on a <i>file</i> . 4. Display <i>commits~branches</i> . 5. Show <i>insertions</i> & <i>deletions</i> . 6. Display <i>commits</i> first coding line by

Basic	Command	Description
		<p>author.</p> <p>7. Customized log (show author, hash & date).</p> <p>8. Search for commits in range.</p> <p>9. Search for commits by author.</p>
17. revert	<ol style="list-style-type: none"> 1. git revert <commit_id> 2. git revert <commit_id> --no-edit 3. git revert -n <commit_id> 4. git revert -n <HEAD>~n 5. git revert -n <HEAD>~n .. <HEAD>~m 	<ol style="list-style-type: none"> 1. Invert commit & commit undone changes. 2. Reverts without a new commit msg. 3. Invert changes & stage only. 4. Revert n commits. 5. Revert from n→m commits [n,m].
18. reset	<ol style="list-style-type: none"> 1. git reset <file> 2. git reset --mixed <HEAD>~n 3. git reset --mixed <commit-id> 4. git reset --hard <HEAD>~n 5. git reset --soft <HEAD>~n 6. git reset -p 	<ol style="list-style-type: none"> 1. Untrack file. 2. Unmerge & uncommit but don't unstage (default). 3. Mixed with commit hash (default). 4. Undo all n changes. 5. Hard reset but able to recover changes with git commit. 6. Patch interactively (git add -p inverse).
19. stash	<ol style="list-style-type: none"> 1. git stash 2. git stash push -m <msg> 3. git stash list 4. git stash list --stat 	<ol style="list-style-type: none"> 1. Saves work dir. from local & hard reset.

Basic	Command	Description
	5. git stash apply 6. git stash pop -n 7. git stash drop -n	2. Saves work dir. from local <i>with msg</i> & <i>hard reset</i> . 3. List <i>stashed</i> changes as an <i>index [n]</i> 4. Show summary of <i>changes</i> in <i>stash list</i> 5. Recover <i>stash[0]</i> from <i>work dir</i> . 6. Recover <i>stash n</i> & <i>delete it</i> from <i>stash list</i> . 7. Delete <i>stash n</i> from <i>stash list</i> .
20. status	1. git status 2. git status -s 3. git status -b	1. List (un)staged, (un)tracked changes (work dir., stage & modif.). 2. Status in <i>short format</i> . 3. Status on a <i>branch</i> .
21. touch	1. git touch <name.ext>	1. Create file with <i>extension</i> (e.g <i>test.txt</i>).
22. switch	1. git switch <branch> 2. git switch -c <branch> 3. git switch -c <branch> <commit_id>	1. Switch to <i>branch</i> . 2. Create a new <i>branch</i> and <i>switch</i> . 3. Grow branch from <i>commit</i> .
23. cd	1. cd ~/ <home> 2. cd ~/ <home> / <dir.> 3. cd ~/ <home> / <dir.> / <subdir.>	1. Change dir. to <i>home</i> (e.g <i>~/Desktop</i>) 2. Change dir. to a <i>folder</i> in <i>home</i> .

Basic	Command	Description
		3. Change dir. to n sub-folders in <i>home</i> .
24. ls	1. ls 2. ls -la	1. List subfolders in dir . 2. List subfolders in dir with <i>hidden files</i> .
25. rm	1. git rm <file> 2. rm <file>	1. Remove file from git tracking & local . 2. Remove file from local only .
26. mv	1. git mv <file.ext> <new-filename.ext> 2. git mv <file.ext> ~/ <home> / <dir.1> / <subdir.>	1. Rename file with the same extension . 2. Move file from dir.1 to subdir. (inside dir.1)
27. mkdir	1. mkdir ~/ <home> / <dir.> / <subdir.>/<new_dir.>	1. Create dir. in path .
28. remote	1. git remote 2. git remote -v 3. git remote rename <old-name> <new-name> 4. git remote add <URL>	1. List remote branches . 2. List remote branches with URL. 3. Rename remote . 4. Connection with repo with URL.
29. gitk	1. gitk 2. gitk HEAD...FETCH_HEAD	1. Show Git GUI for commits . 2. Show Git GUI for all users since last push.

Installation Required:

Basic	Command	Description
30. grip	1. grip --export 2. grip --export <file-name> .ipynb <file-name> .html 3. grip --export <name> .ipynb <name> .pdf	1. Export <i>README.md</i> to <i>README.html</i> . 2. Export <i>file.ipynb</i> to <i>file.html</i> . 3. Export <i>file.ext</i> to <i>file.ext2</i>

```
!pip install grip
```

Note: Branches should be called by their names even if they weren't specified (see 14. branch).

Tip: <main> is the default name for remote repositories as <master> is for local.

Definitions:

- **Origin:** Primary *working dir. of remote* repositories by *default*.
- **Fetch:** Fetch is a *safe pull* version because local *files aren't merged* until they are reviewed, checked out & merged.
- **Revert:** Revert is *safer than reset*, checkout to *discard* (see 5.4 checkout), etc., because commit *history isn't erased* but an inverted commit is appended.
- **Feature:** Feature represents a *branch of developments* in progress with their descriptions.
- **Rebase:** Rebase is a *rewritten branch* from another but keep in mind it is *not a good practice to rewrite public commits history (remote repositories)*.
Creating a backup branch is a good idea. This would allow us to perform a hard reset if the resulting rebase is unexpected.
- **Base:** It is a commit *id, branch, tag*, or a relative *reference* to HEAD (e.g. HEAD~3).

See Also:

[Glossary](#)

If you are interested in learning more about git commands you can check out the list below and refer to [git documentation](#).

Other Commands:

- **git am** ~ Splits patches from a mailbox into commit msg, author and patches to apply them to branch.
e.g: `git am --keep-cr --signoff < a_file.patch` *to apply patch as commit.*
- **git apply** ~ Apply a patch to files and add them to the index.
e.g: `git apply < a_file.patch` *to apply patch to files.*
- **git archive** ~ Combine multiple files in a single file but removes git data.
e.g: `git archive --format=zip --output=archive.zip HEAD` *to create a zip file with all files in HEAD.*
- **git bisect** ~ Binary search algorithm to find commit in project history which caused a bug.
e.g: `git bisect start` *to start the search.*
- **git blame** ~ Show what revision and author last modified each line of a file.
e.g: `git blame <file>` *to show the last author of each line in file.*
- **git bugreport** ~ Create a report to send to git mailing list.
e.g: `git bugreport -o report.txt` *to create a report and save it to report.txt.*
- **git bundle** ~ Move objects and refs by archive.
e.g: `git bundle create <file> <branch>` *to create a bundle with branch.*
- **git cat-file** ~ Provide content or type and size information for repository objects.
e.g: `git cat-file -p <commit>` *to show the content of commit.*
- **git check-attr** ~ Display git attributes.
e.g: `git ls-files | xargs git check-attr myAttr` *to show if an attribute is set for all the files in repo & overcome limit of 1024 files.*
- **git check-mailmap** ~ Show canonical names and email addresses of contacts.
e.g: `git check-mailmap user1 <user1@domain.com>` *to show the canonical name and email address of user1.*
- **git check-ref-format** ~ Ensure that a reference name is well formed.
e.g: `git check-ref-format --branch @{-1}` *print the name of the previous branch.*
- **git check-ignore** ~ Debug gitignore files.
e.g: `git check-ignore -v <file>` *to show the gitignore file that ignores file.*
- **git cherry** ~ Find commits not merged upstream.
e.g: `git cherry -v <branch>` *to show the commits not merged in branch.*

- **git cherry-pick** ~ Apply the changes introduced by some existing commits.
e.g: `git cherry-pick <commit_id>` *to apply the changes of commit to current branch.*
- **git citool** ~ Graphical alternative to git-commit.
e.g: `git citool` *to open the graphical commit tool.*
- **git clean** ~ Remove untracked files from the working tree.
e.g: `git clean -i` *to interactively remove untracked files.*
- **git clone** ~ Clone a repository into a new directory.
e.g: `git clone <URL> <dir.>` *to clone a repo with URL into directory.*
- **git column** ~ Display data in columns.
e.g: `git column --mode=html <file>` *to display file in html columns.*
- **git commit** ~ Record changes to the repository.
e.g: `git commit -m <msg>` *to commit with msg.*
- **git commit-graph** ~ Write and verify a commit-graph file.
e.g: `git show-ref -s | git commit-graph write --stdin-commits` *to write a commit-graph file for reachable commits.*
- **git commit-tree** ~ Create a new commit object.
e.g: `git commit-tree <tree> -m <msg>` *to create a commit with tree and msg.*
- **git config** ~ Get and set repository or global options.
e.g: `git config --global user.name <name>` *to set the global user name.*
- **git count-objects** ~ Count unpacked number of objects and their disk consumption.
e.g: `git count-objects -v` *to show the number of objects and their size.*
- **git credential** ~ Retrieve and store user credentials.
e.g: `git credential fill` *attempt to add "username" and "password" attributes by reading config credential helpers.*
- **git credential-cache** ~ Helper to temporarily store passwords in memory.
e.g: `git config credential.helper cache` *to set credentials automatic authentication & returns username/password blanks to fill.*
- **git credential-store** ~ Helper to store credentials on disk to reduce time to fill.
e.g: `git config --global credential.helper store` *to save credentials in plaintext PC disk, everyone in PC can read it (warning).*
- **git cvsexportcommit** ~ Export a single commit to a CVS checkout.
e.g: `git cvsexportcommit <commit_id>` *to export commit to a CVS directory.*
- **git cvsimport** ~ Create a new git repository from a CVS checkout.
e.g: `git cvsimport -v -d <cvsroot> <module> <project>` *to create a new git repository from a CVS checkout.*
- **git cvsserver** ~ Server for CVS clients to connect to and use Git repositories.
e.g `git cvsserver --base-path=<path> <repo>` *to start the git cvsserver.*

- **git daemon** ~ A really simple server for Git repositories.
e.g: `git daemon --reuseaddr --base-path=<dir.> --export-all` to restart server & look for repos in dir. to export.
- **git describe** ~ Describe specific commits with their hash.
e.g: `git describe <commit_id>` to describe commit (HEAD by default).
- **git diff** ~ Show changes between commits, commit and working tree, etc.
e.g: `git diff --stat` to show the summary of the changed files.
- **git diff-files** ~ Show changes between index and working tree.
e.g: `--diff-algorithm={minimal}` to include the smallest possible diff are included.
- **git diff-index** ~ Compare a tree to the working tree or index.
e.g: `git diff-index --compact-summary HEAD` to show the summary of the changed files in HEAD.
- **git diff-tree** ~ Compares the content and mode of the blobs found via two tree objects.
e.g: `git diff-tree --s7hortstat HEAD` to show the summary of the changed files in HEAD.
- **git difftool** ~ Show changes using common diff tools.
e.g: `git difftool --tool-help` to show the list of available tools.
- **git fast-export** ~ Dumps the given revisions in a form suitable to be piped with fast-import.
e.g: `git fast-export --all` to export all data.
- **git fast-import** ~ Reads data stream from std. input and writes it into one or more packfiles.
e.g: `git fast-import --max-pack-size=1G` to import data into a packfile of size 1G (default is unlimited)
- **git fetch** ~ Download objects and refs from another repository.
e.g: `git fetch --dry-run` to show output without making any changes.
- **git fetch-pack** ~ Receive missing objects from another repository.
e.g: `git fetch-pack --prune --all` to fetch all objects and prune refs that are missing on the remote.
- **git filter-branch** ~ Rewrite branches.
e.g: `git filter-branch --tree-filter 'rm -f *.txt' HEAD` to remove all .txt files.
- **git filter-repo** ~ Quickly rewrite Git repository history.
e.g: `git filter-repo --invert-paths --path 'README.md'` to remove all files except README.md.
- **git fmt-merge-msg** ~ Produce a merge commit message.
e.g: `git fmt-merge-msg -m` Use msg instead of branch names for the first line of the log message.

- **git for-each-ref** ~ Iterate over references.
e.g: `git for-each-ref --format='%(%(refname))' refs/heads` to list all branches.
- **git format-patch** ~ Prepare patches for e-mail submission.
e.g: `git format-patch -root <commit>` to format everything up from start until commit.
- **git fsck** ~ Verifies the connectivity and validity of the objects in the database.
e.g: `git fsck --cache` to check the connectivity and validity of the objects in the cache.
- **git gc** ~ Cleanup unnecessary files and optimize the local repository.
e.g: `git gc --force` to force garbage collection.
- **git get-tar-commit-id** ~ Extract commit ID from an archive created using git-archive.
e.g: `git get-tar-commit-id <file>` to extract most recent commit ID from file.
- **git grep** ~ Print lines matching a pattern.
e.g: `git grep -n 'print' <file>` to print lines containing 'print' and their line numbers.
- **git gui** ~ A portable graphical interface to Git.
e.g: `git gui citool --nocommit` Checks for unmerged entries on index and exits gui without committing.
- **git hash-object** ~ Compute object ID and optionally creates a blob from a file.
e.g: `git hash-object -w --path <file>` to write the blob to the object database and print its hash.
- **git help** ~ Display help information about Git.
e.g: `git help -all` to display all git commands.
- **git http-fetch** ~ Download objects and refs from another repository via HTTP.
e.g: `git http-fetch -v <[URL]/refs>` to report all refs downloaded in repo with URL.
- **git http-backend** ~ Server side implementation of Git over HTTP.
e.g: `git http-backend --help` to display help for http-backend.
- **git imap-send** ~ Send a collection of patches from stdin to an IMAP folder.
e.g: `git imap-send git format-patch --cover-letter -M --stdout origin/master | git imap-send` to send patches from origin/master to IMAP folder once the commits are ready to send.
- **git index-pack** ~ Build pack index file for an existing packed archive.
e.g: `git index-pack --max-input-size=1G` to build pack index file and die if the pack is larger than 1G (or any).
- **git init** ~ Create an empty Git repository or reinitialize an existing one.
e.g: `git init -b <branch-name>` to create an empty local Git repository with given branch name.
- **git init-db** ~ Create an empty Git repository or reinitialize an existing one.
e.g: `git init-db --config <config-file>` to create an empty local Git repository with given config file.

- **git instaweb** ~ Instantly browse your working repository in gitweb.
e.g: `git instaweb --httpd=python --port=8080` to start a python web server on port 8080.
- **git interpret-trailers** ~ Parse trailer lines from text.
e.g: `git interpret-trailers --check <file>` to check if file contains trailer lines (similar to RFC 822 e-mail headers)
- **git log** ~ Show commit logs.
e.g: `git log --oneline --decorate --graph --all` to display all commits in a nice format.
- **git ls-files** ~ Show information about files in the index and the working tree.
e.g: `git ls-files -u` to show unmerged files.
- **git ls-remote** ~ List references in a remote repository.
e.g: `git ls-remote <[URL]/refs>` to display references in a remote repository URL associated with commits IDs.
- **git ls-tree** ~ List the contents of a tree object.
e.g: `git ls-tree -d <tree>` to list the named tree only, without its children.
- **git mailinfo** ~ Extracts patch and authorship from a single e-mail message.
e.g: `git mailinfo -k <msg> <patch>` Removes unnecessary headers from msg and writes the result to patch.
- **git mailsplit** ~ Splits a single mailbox into a list of files.
e.g: `git mailsplit -o<directory> <mbox>` to split given mbox file in directory as individual msg files.
- **git merge** ~ Join two or more development histories together.
e.g: `git merge --allow-unrelated-histories <branch>` override the check for unrelated histories with common ancestors and merge.
- **git merge-base** ~ Find as good common ancestors as possible for a merge.
e.g: `git merge-base --is-ancestor <commit_id> <commit_id>` to check if first commit_id is an ancestor of the second and return 0 if true and 1 if not.*
- **git merge-file** ~ Run a three-way file merge.
e.g: `git merge-file <current_file> <base_file> <other_file>` incorporate changes from other_file into current_file, using base_file as common base
- **git merge-index** ~ Run a merge for files in the index.
e.g: `git merge-index -o -a <file>` to run a merge for all files in index that need it & write result to file.
- **git merge-tree** ~ Show three-way merge without touching index.
e.g: `git merge-tree <base-tree> <branch1> <branch2>` Reads the trees & outputs the result of merge without storing results in index.*

- **git mergetool** ~ Run merge conflict resolution tools to resolve merge conflicts.
e.g: `git mergetool --tool-help` to list available tools.
- **merge-index** ~ Run a merge for files in the index.
e.g: `git merge-index -o <file>` to run a merge for files in the index that need merging and write the result to file.
- **git mktag** ~ Create a tag object.
e.g: `git mktag <mytag>` *to create a tag object with given tag name and die if the connection to the object store fails.
- **git mktree** ~ Build a tree-object from ls-tree formatted text.
e.g: `git mktree --batch <file>` to create more than one tree object from a file.
- **git mv** ~ Move or rename a file, a directory, or a symlink.
e.g: `git mv -v <source> <destination>` to move source to destination and display the result of the move.
- **git name-rev** ~ Find symbolic names for given revs.
e.g: `git log | git name-rev --annotate-stdin` to retrieve author, date and commit hash from the logs.
- **git notes** ~ Add or inspect object notes.
e.g: `git notes add -m <msg> <commit>` to add a note/msg to commit.
- **git pack-objects** ~ Create a packed set of objects from one or more packed archives compressed
e.g: `git pack-object --all-progress-implied` to create a packed set of objects from one or more packed archives compressed.
- **git pack-redundant** ~ Find redundant pack files for piping to xargs rm.
e.g: `git pack-redundant --all --i-still-use-this` to find all redundant pack files in repo (nominated for removal).
- **git pack-refs** ~ Pack heads and tags for efficient repository access.
e.g: `git pack-refs --all` to pack heads and tags that are already packed
- **git patch-id** ~ Compute unique ID for a patch.
e.g: `git patch-id <file>` to compute unique ID for a patch.
- **git prune** ~ Prune all unreachable objects from the object database.
e.g: `git prune --expire <time>` to prune all unreachable objects from the object database that are older than time.
- **git prune-packed** ~ Prune loose objects that are already in pack files.
e.g: `git prune-packed -n` to prune loose objects that are already in pack files and display what would be done.
- **git pull** ~ Fetch from and integrate with another repository or a local branch.
e.g: `git pull <remote> <local>` to fetch from and integrate with local branch.

- **git push** ~ Update remote refs along with associated objects.
e.g: `git push` to update remote refs along with associated objects.
- **git range-diff** ~ Show changes between two commit ranges.
e.g: `git range-diff <commit_1> <commit_2>` to show changes between two commit ranges
- **git read-tree** ~ Reads tree information into the index.
e.g: `git read-tree -m <tree-ish1> <tree-ish2> <tree-ish3>` to read tree information into the index and merge the trees.
- **git rebase** ~ Reapply commits on top of another base tip.
e.g: `git rebase -i <base> <branch>` to rebase interactively a branch on base.
- **git receive-pack** ~ Receive what is pushed into the repository.
Note: This command is not meant to be invoked directly.
- **git reflog** ~ Manage reflog information.
e.g: `git reflog show` to show the reflog for the current branch like log.
- **git remote** ~ Manage set of tracked repositories.
e.g: `git remote add <remote> <URL>` to add a remote named remote with URL.
- **git remote-ext** ~ External helper to communicate with a remote, used by default with clone, push, remote add & where.
Note: This command is not used normally by end users but it is instead invoked when interacting with remote repos.
- **git remote-fd** ~ Helper to communicate with a remote repository when calling git fetch, push or archive.
Note: This command is not invoked by end users but scripts calling commands to setup a bidirectional socket with remotes.
- **git repack** ~ Pack unpacked objects in a repository or for pack reorganization.
e.g: `git repack -a -d -f --depth=250 --window=250` Single pack repo by removing redundant packs & reusing existing deltas. Set up 250mb depth and window (default=10,50).
- **git replace** ~ Create, list, delete refs to replace objects.
e.g: `git replace --graft <commit_id> <new-parent>` to create a new commit with commit content but by replacing its parent with new-parent.
- **git request-pull** ~ Request upstream to pull changes into their tree.
e.g: `git request-pull <upstream_commit-id> <URL>` to make a pull-request starting from commit to repo URL to be pulled from.
- **git rerere** ~ Reuse recorded resolution of conflicting merges.
e.g: `git rerere diff` to show the recorded state of resolution, what you've started with and what you've ended up with.

- **git reset** ~ Reset current HEAD to the specified state.
e.g: `git reset --soft HEAD~n` *to make a hard reset n commits back but able to recover changes with git commit.
- **git rev-list** ~ Lists commits by building commit ancestry graphs. *e.g:* `git rev-list <commit_id> ^ HEAD --count` to count the number of commits between commit_id and HEAD.
- **git rev-parse** ~ Ancillary plumbing command for parameters.
e.g: `git rev-parse --short HEAD` to get the short version hash of HEAD.
- **git revert** ~ Revert some existing commits.
e.g: `git revert HEAD~n` to revert the last n commits.
- **git rm** ~ Remove files from the working tree and from the index.
e.g: `git rm <file>` to remove file from remote and local.
- **git send-email** ~ Send a collection of patches as emails.
e.g: `git send-email --from=<sender> --to=<recipient> --compose` to send email from sender address to recipient by invoking a text editor.
- **git shortlog** ~ Summarize 'git log' output.
e.g: `git shortlog -s -n` to show the number of commits per author.
- **git show** ~ Show various types of objects.
e.g: `git show --expand-tabs=n` to show repository with tabs expanded to n.
- **git show-branch** ~ Show branches and their commits.
e.g: `git show-branch--all` to show all branches and their commits.
- **git stage** ~ Stage file contents for the next commit.
e.g: `git stage--clear` to clear the staging area.
- **git stash** ~ Stash the changes in a dirty working directory away.
e.g: `git stash--keep-index` to stash the changes in a dirty working directory away but keep the index.
- **git status** ~ Show the working tree status.
e.g: `git status--short` to show the working tree status in short format.
- **git strip-space** ~ Remove unnecessary whitespace.
e.g: `git strip-space--comment-lines` to remove unnecessary whitespace from comment lines.
- **git submodule** ~ Initialize, update or inspect submodules.
e.g: `git submodule--depth=1` to initialize, update or inspect submodules with depth 1.
- **git tag** ~ Create, list, delete or verify a tag object signed with GPG.
e.g: `git tag --annotate` to create, list, delete or verify a tag object signed with GPG.
- **git unpack-file** ~ Unpack a packed archive.
e.g: `git unpack-file --list` to list the contents of a packed archive.

- **git unpack-objects** ~ Unpack objects from a packed archive.
e.g: `git unpack-objects --all` to unpack all objects from a packed archive.
- **git update-index** ~ Register file contents in the working tree to the index.
e.g: `git update-index --refresh` to register file contents in the working tree to the index.
- **git update-ref** ~ Update the object name stored in a ref safely.
e.g: `git update-ref --no-deref` to update the object name stored in a ref safely.
- **git update-server-info** ~ Update auxiliary info file to help dumb servers.
e.g: `git update-server-info --force` to update the file even if it is not necessary.
- **git upload-archive** ~ Send archive back to git-upload-archive on the other end.
e.g: `git upload-archive` to send archive back to git-upload-archive on the other end.
- **git upload-pack** ~ Send objects packed back to git-upload-pack on the other end.
e.g: `git upload-pack` to send objects packed back to git-upload-pack on the other end.
- **git var** ~ Show a Git logical variable.
e.g: `git var -l` to show a Git logical variable.
- **git verify-commit** ~ Check the GPG signature of commits.
e.g: `git verify-commit <commit>` to check the GPG signature of commits.
- **git verify-pack** ~ Check the GPG signature of packed objects.
e.g: `git verify-pack` to check the GPG signature of packed objects.
- **git verify-tag** ~ Check the GPG signature of tags.
e.g: `git verify-tag <tag>` to check the GPG signature of tags.
- **git web--browse** ~ Show a file or directory from web browser.
e.g: `git web--browse <URL>` to show a file or directory from a web browser.
- **git whatchanged** ~ Show logs with difference each commit introduces.
e.g: `git whatchanged --stat` to show logs with difference each commit introduces.
- **git write-tree** ~ Create a tree object from the current index.
e.g: `git write-tree --missing-ok` to create a tree object from the current index.

References:

1. [Git](#)
2. [Linux Man](#)
3. [Ubuntu Manuals](#)
4. Official Git Pro [ebook](#). Chacon, S and Straub, B. (2022).
5. [\(Github\) Render](#) (Command 30.)

Collaborations:



Contributors:



EstebanMqz 

Contact:



¿Just Thanks 