Center for HIGH PERFORMANCE COMPUTING
THE UNIVERSITY OF UTAH

Git reference

Written by Robben Migacz · Updated March 2025

Git is **version control software**, which means that it keeps track of changes to files and helps its users manage versions. Git can be a bit difficult to navigate as a new user; to help with this, we've put together a reference for some common commands. In this reference, we've made a few simplifications and left out a number of commands and options. Please consult the Git documentation (available online at https://git-scm.com/docs ⊖) for more information about each concept or command if you'd like to learn more.

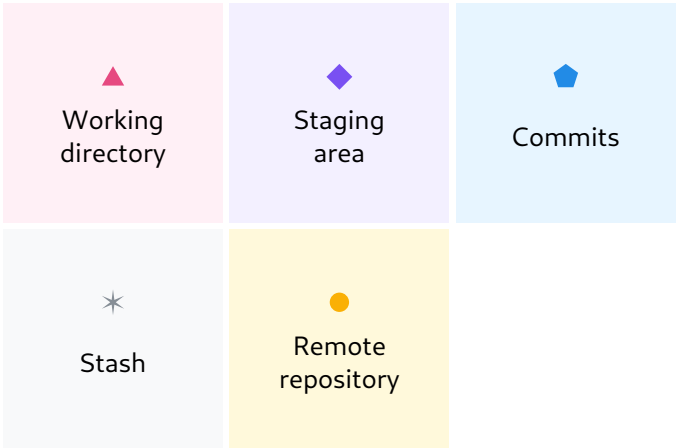
Contents

Conventions in this reference

Commands are shown in **bold, blue-green text**. Segments of text in *italics* are values you will want to change based on your own project or situation. Select commands have annotations that show how they affect projects, like

▲ → ◆,

which represents updating the staging area based on the contents of the working directory (see diagram below). Annotations are not present on all commands; please consult the Git documentation if you are not sure how a command will affect your project.



Getting help

Git comes with documentation and several tutorials in case you need an explanation of a command, a list of options, or an example.

git help commit
man git-commit

Get information about the command **git commit** (as an example; also applies to other commands)
Read documentation ⊜

git help tutorial
git help everyday
git help workflows
git help glossary

View tutorials created by the maintainers of Git
Read documentation ⊜

Inspecting objects

Git provides a tool to help you find more information about objects like commits and tags.

git show *object-identifier*

Get further details about *object-identifier*, which may be an identifier like a commit, a branch name, or a tag
Read documentation ⊜

Initializing a repository

You can obtain a Git repository by initializing one in a directory or cloning an existing project.

git init

Initialize a new Git repository in the current directory
Read documentation ⊜

git clone *remote-location*

Copy a repository from *remote-location* into a new directory within the current directory
Read documentation ⊜

git clone *remote-location directory-name*

As above, with *directory-name* as the name of the new directory
Read documentation ⊜

git clone --origin *origin-name remote-location directory-name*

As above, with *origin-name* as the name of the remote repository (used for **git push** and **git pull**)
Read documentation ⊜

Configuring Git

There are many options that affect the behavior of Git. These can be set with commands or by modifying configuration files. You can set defaults and make adjustments for specific repositories.

ⓘ Information like your name and email address is added to commits; sharing your repository with another person will expose this information. Changing the configuration does not affect prior commits.

Global configuration

The global configuration is the default; it is used when there are no repository-specific configurations.

git config --global user.name "*Your Name*"

Set the user's default name for commits
Read documentation ⊜

git config --global user.email "*your.name@your.domain*"

Set the user's default email for commits
Read documentation ⊜

git config --global --edit

Edit the global (default) configuration in a text editor (see below for notes on editor selection)
Read documentation ⊜

git config --global core.editor "nano"

Set the default text editor to nano (as an example; also applies to other editors); alternatively, set an environment variable, as in **(export GIT_EDITOR="nano"; git config --global edit)**

Read documentation ⊜

Local configuration

The local configuration applies to a specific repository and overrides the global configuration.

git config user.name "*Your Name*"

Set the user's name for commits in the current repository; other options can also be changed locally but are omitted here

Read documentation ⊜

git config edit

Edit the configuration for the current repository in a text editor

Read documentation ⊜

Viewing the configuration

Git provides a tool that shows the current configuration (including repository-specific configurations where applicable and global configurations otherwise).

git config list

List current configurations (global and local)

Read documentation ⊜

Viewing the status of a repository

Git provides a tool that allows you to see the status of the working directory and staging area.

git status

Show the status of the project, including a list of files with modifications since the last commit and files currently staged

Read documentation ⊜

Working with the staging area

Before you can make a commit, you must add your changes to the staging area.

Adding files

Changes that are added to the staging area are included in commits.

git add *example-filename*

▲ → ◆

Add *example-filename*, which may be a pattern like . or *.py, to the staging area

Read documentation ⊜

git add --all

▲ → ◆

Match the staging area to the working directory; will add files—**including new files** (compare to **git commit --all**)—or remove where appropriate

Read documentation ⊜

git add --dry-run *files-to-add*

See the effect that **git add *files-to-add*** would have, but don't actually **add** any files

Read documentation ⊜

Removing files

You can also remove files from the staging area if you decide you don't want to commit certain changes.

ⓘ Please read the documentation on **git rm** before using it. The command can remove files, which is permanent.

git rm --cached *files-to-remove-from-index*

◆ → ▲

Remove *files-to-remove-from-index*, which may be a pattern like . or *.py, from the staging area; don't delete the files from the filesystem

Read documentation ⊜

git rm --dry-run *files-to-remove*

See the effect **git rm *files-to-remove*** would have, but don't actually **rm** any files

Read documentation ⊜

git rm *file-to-remove*

Remove the file *file-to-remove* from the repository **and the filesystem**

Read documentation ⊜

git rm -r *directory-to-remove*

Remove the directory *directory-to-remove* from the repository **and the filesystem**

Read documentation ⊜

Comparing files

Git provides tools that can help you compare different versions of your project's files.

git diff

Compare uncommitted changes to the last commit

Read documentation ⊜

git diff --cached

Compare staged changes to the last commit

Read documentation ⊜

git diff *object-before object-after*

Compare *object-before* and *object-after*, which may be commits, branches, or tags

Read documentation ⊜

git diff --word-diff=color

Compare by words, not lines, with color to distinguish between words added and removed; by default, words are split at whitespace

Read documentation ⊜

Committing changes

Commits are snapshots of your project. Committing saves the state of your (staged) files and allows you to explore the history of your project.

git commit

◆ → ⬟

Create a new commit with the changes in the staging area

Read documentation ⊜

git commit --dry-run

See the effect **git commit** would have, but don't actually commit anything

Read documentation ⊜

git commit --all

▲ ◆ → ⬟

Skip the **git add** step by including **previously tracked** (not new; compare to **git add --all**) files that have been modified or deleted; shortened as **git commit -a**

Read documentation ⊜

git commit --message="*Message about the commit*"

◆ → ⬟

Skip the step of opening a text editor to specify a commit message; shortened as **git commit -m "*Message about the commit*"**

Read documentation ⊜

git commit -am "*Message about the commit*"

▲ ◆ → ⬟

Combine the effects of the previous two commands

Read documentation ⊜

git commit --amend

◆ → ⬟

Replace the previous commit; not recommended if you have already shared your project with others

Read documentation ⊜

Viewing the project history

Git provides a tool that allows you to view the history (commits) of a project.

git log

View information about commits

Read documentation ⊜

git log --graph --all --oneline

Show all commits with a visual representation of the associations between commits

Read documentation ⊜

git log --graph --all --abbrev-commit --date=relative

Similar to the previous command, but with additional information about each commit and a timestamp relative to the current time

Read documentation ⊜

Managing tags

Tags allow you to annotate specific commits and add information like version numbers.

ⓘ Tags must be transferred to remotes explicitly. Use **git push *your-remote-name some-tag-name*** or **git push *your-remote-name* --tags**. To delete, use **git push *your-remote-name* --delete *some-tag-name***.

git tag

List tags

Read documentation ⊜

git tag -a *some-tag-name* -m "*Message about the tag*"

Annotated tag; by convention, *some-tag-name* is often something like v1.0; includes tagger name and email, date, and message (compare to below command)

Read documentation ⊜

git tag *some-tag-name*

A "lightweight" tag *some-tag-name*, which does not include tagger information, date, or message (compare to above command)

Read documentation ⊜

git tag -d *some-tag-name*

Delete the tag *some-tag-name*

Read documentation ⊜

Working with branches

Branches allow projects to exist in multiple states at the same time and can facilitate collaboration when working with others.

git branch

List branches

Read documentation ⊜

git branch *some-new-branch*

Create branch *some-new-branch*, but don't switch to it (compare to **git switch --create *some-new-branch***)

Read documentation ⊜

git branch --delete *some-branch-to-delete*

Delete branch *some-branch-to-delete*

Read documentation ⊜

git switch *some-other-branch*

Change to branch *some-other-branch*

Read documentation ⊜

ⓘ Switching to a different branch may fail if doing so would

result in a loss of information (this is essentially the same as a merge conflict). You can perform a merge when switching by adding --merge.

git switch --create *some-new-branch*

Create—and switch to (compare to **git branch**)—a new branch *some-new-branch*; shortened as **git switch -c *some-new-branch***

Read documentation

git merge *branch-to-merge-into-current*

Merge *branch-to-merge-into-current* into the current branch

Read documentation

git mergetool

Start a tool to help with conflict resolution for merges; note that you may need to **configure** a specific tool first

Read documentation

ⓘ A merge may fail if there are conflicts. Conflicts occur when the same parts of a project have been modified on more than one branch involved in a merge. A merge conflict will require manual intervention; you will need to review files before making another **commit**. https://git-scm.com/docs/git-merge#_how_to_resolve_conflicts

Working with the stash

The stash allows you to quickly store your working directory and index. With the stash, you can save what you're working on without creating a commit. It's useful for quick tests.

git stash list

View entries in the stash list

Read documentation

git stash push

▲ ◆ → ✶

Move working directory to stash; equivalent to **git stash**

Read documentation

git stash pop *some-stash-entry*

✶ → ▲

Inverse of **git stash push**; moves entry from stash to working directory (and **removes entry** from stash list; compare to **git stash apply *some-stash-entry***)

Read documentation

git stash apply *some-stash-entry*

✶ → ▲

Inverse of **git stash push**; moves entry from stash to working directory (and **does not remove entry** from stash list; compare to **git stash pop *some-stash-entry***)

Read documentation

ⓘ **git stash pop** and **git stash apply** may fail if there are conflicts. This is similar to a merge conflict, though it applies to unstaged files.

git stash drop *some-stash-entry*

Remove the entry *some-stash-entry* from the stash list

Read documentation

Working with remotes

Remote repositories allow you to share your work with others and collaborate on projects.

git remote add *your-remote-name remote-location*

Add a remote named *your-remote-name* pointing to *remote-location*

Read documentation

ⓘ When using **git clone** to clone a remote repository, the remote is named origin by default (unless this has been changed in the configuration).

git pull *your-remote-name branch-name*

● → ⬟

Pull *branch-name* from *your-remote-name* into the local repository

Read documentation

git push *your-remote-name branch-name*

Push *branch-name* from the local repository to remote *your-remote-name*

Read documentation

ⓘ **git pull** and **git push** may fail if there are conflicts. A conflict with a **pull** is similar to a merge conflict. A conflict with a **push** can usually be fixed by pulling changes from the remote repository first. https://git-scm.com/docs/git-push/2.1.4#_note_about_fast_forwards

Reverting and resetting

Reverting and resetting both allow you to undo changes (commits) to your project, but they function very differently.

Reverting

Reverting is always an additive change; it does not change the project history. This is generally preferred when working with others.

ⓘ **git revert** will revert the changes in only one commit at a time. It must be done for each commit to be reverted. (A range can be used to simplify this process.)

git revert *some-commit*

Undo the changes in *some-commit* by creating a new commit

Read documentation

git revert *some-commit..some-other-commit*

Revert the changes in a range of commits from *some-commit* (exclusive; see note below) to *some-other-commit*

Read documentation

git revert *some-commit..*

Revert the changes in a range of commits beginning at *some-commit* (exclusive; see note below)

Read documentation

ⓘ The beginning of the range is not included in the commits that will be reverted. Use **git revert *some-commit^..some-other-commit*** to include the first commit in the reversion.

Using a range will result in many new commits if you are reverting many commits (one each). You may want to use **(git revert --no-commit *some-commit..some-other-commit* && git commit)**, which will result in one commit instead of several.

It is a little more difficult to revert a merge. Git does not know which of the two (or more) branches to keep; there's a fork in the road. Note that **git show *merge-commit*** will include a line like "Merge: *first-commit-in-merge second-commit-in-merge*." The command **git revert *merge-commit* -m 1** will revert to the state in *first-commit-in-merge*; **git revert *merge-commit* -m 2** will revert to the state in *second-commit-in-merge*.

Resetting

Resetting can change the history of your project. It is generally not recommended when working with others (if you have already shared your project files).

ⓘ Please read the documentation on **git reset** before using it. It changes the history of the project and may cause you to lose information. **git reset *some-object* affects the current branch reference** (compare to **git checkout *some-object***).

git reset --soft *some-object*

Move to *some-object*, but keep the working directory and staged changes as-is

Read documentation

git reset --mixed *some-object*

Move to *some-object* and move staged changes to the working directory

Read documentation

git reset --hard *some-object*

Move to *some-object* and discard working directory and staged changes
Read documentation ⊜

Rebasing

A rebase moves a commit to a new base (parent). In effect, it allows you to move commits around on the graph. It also allows you to combine commits, which can be helpful when you have a series of small changes and want to simplify your project history.

ⓘ Rebasing changes the project history. Generally, you should avoid this if you have already shared your commits with others (on a remote repository, for instance).

git rebase *branch-name*

Rebase the current branch on top of *branch-name*
Read documentation ⊜

ⓘ A rebase can fail if there are conflicts. It proceeds one commit at a time; if there is a conflict, you will need to resolve it and **commit** your changes. You can then use **git rebase --continue** to proceed with the rebase that was interrupted by the conflict.

git rebase --interactive *some-object*

Rebase starting at *some-object*, which is useful if you want to combine or amend commits; shortened as **git rebase -i *some-object***
Read documentation ⊜

Moving to a different project state

You can work "on top of" a specific commit (or branch or tag). (In effect, you are viewing the state of a project at a different commit.)

git checkout *some-object*

Move HEAD (the current view of the project) to *some-object* **without affecting the current branch reference** (compare to **git reset *some-object***)
Read documentation ⊜

ⓘ Performing a **checkout** on a specific commit (not the tip of a branch) yields a state called "detached HEAD." You can **checkout** a branch to return HEAD to the tip of a branch.