**Git Cheat Sheet**

**Initial Setup**

[**Initialize a repo**](https://git-scm.com/docs/git-init)

Create an empty git repo or reinitialize an existing one

git init

[**Fork a repo**](https://help.github.com/articles/fork-a-repo#step-1-fork-the-spoon-knife-repository)

Click the "Fork" button at the top-right of any repository's GitHub page.

[**Clone a repo**](https://git-scm.com/docs/git-clone)

Clone the foo repo into a new directory called foo:

git clone https://github.com/<username>/foo.git foo

[**Setup Remotes**](https://git-scm.com/docs/git-remote)

First, let's see a list of the repositories (remotes) whose branches you track:

git remote -v

Oh, it looks like we haven't setup upstream. Now is the time:

git remote add upstream https://github.com/<upstream\_username>/<repo\_name>.git

git fetch upstream

**Every-day Workflow**

[**Branching**](https://git-scm.com/docs/git-switch)

When working on a fork, you could be switching between different branches quite commonly. As such, you generally want to stay off the master branch and work on your own feature branches so that master is always clean and you can base new branches off of it.

git switch -c <new-branch-name>

If upstream has a special develop branch or something, you can checkout that branch separately, but setup tracking so you can sync it up from time to time. Like the master branch, don't work directly on this one. Try to keep it clean.

git switch -c <new-branch-name> --track upstream/<remote-branch-to-track>

Maybe you made some progress on a branch at work, but now you want to continue work at home. In that case, you're dealing with your own fork's branch, so you'll checkout from origin.

git switch -c <new-branch-name> --track origin/<remote-branch-to-track>

If you're using the same name as a remote branch name, this can be simplified:

git switch -c <branch-name>

Use the -C option flag to force it.

[**Switching Branches**](https://git-scm.com/docs/git-switch)

First, you'll want to know what branches are available in your working directory:

git branch

develop

feature\_x

master

Now, you can easily switch between branches with git switch:

git switch master

git switch develop

git switch feature\_x

git config --global alias.sw 'switch'

git sw master

**Switch to previous branch**

git switch @{-1}

*credit:*[*@MattNewbill*](https://gist.github.com/jedmao/5053440#gistcomment-3354581)

[**Status**](https://git-scm.com/docs/git-status)

Not sure if you're working on a clean branch? Want to see what files have changed? Git status will show you a report.

git status

[**Staging Changes**](https://git-scm.com/docs/git-add)

Now that you've added or modified some files, you need to stage those commits into "the staging area." Think of git commits like an array of airlock hatches on a space ship. On this space ship, you can only open the door to one airlock at a time. When you open the hatch, you can put stuff in or take stuff out at will. Not until you've closed the door have you committed those changes (git commit) and not until you hit the red button do all those hatches open up into space (git push).

You can stage inidividual files or all files at once.

git add foo.js

git add .

[**Unstaging Changes / Restoring Files**](https://git-scm.com/docs/git-restore)

Maybe you accidentally staged some files that you don't want to commit.

git restore foo.js

git restore .

[**Commits**](https://git-scm.com/docs/git-commit)

Commit often. You can always squash down your commits before a push.

git commit -m "Updated README"

Want to automatically stage files that have been modified and deleted, but new files you haven't told git about will be unaffected? Pass the -a or --all option flag:

git commit -am "Updated README"

[**Undoing Commits**](https://git-scm.com/docs/git-reset)

The following command will undo your most recent commit and put those changes back into staging, so you don't lose any work:

git reset --soft HEAD~1

The next one will completely delete the commit and throw away any changes. Be absolutely sure this is what you want:

git reset --hard HEAD~1

[**Squashing Commits**](https://git-scm.com/docs/git-rebase)

Maybe you have 4 commits, but you haven't pushed anything yet and you want to put everything into one commit so your boss doesn't have to read a bunch of garbage during code review.

git rebase -i HEAD~4

An interactive text file is displayed. You'll see the word "pick" to the left of each commit. Leave the one at the top alone and replace all the others with "s" for squash, save and close the file. This will display another interactive window where you can update your commit messages into one new commit message. I like to use "f" instead of "s", because I usually work in such a way that I name my first commit appropriately from the get-go. "f" just skips the 2nd interactive file and uses the first commit message.

[**Pushing**](https://git-scm.com/docs/git-push)

Push a local branch for the first time:

git push --set-upstream origin <branch>

git push

Configure Git to [always push using the current branch name](https://git-scm.com/docs/git-config#Documentation/git-config.txt-pushdefault):

git config --global push.default current

Push a local branch to a different remote branch:

git push origin <local\_branch>:<remote\_branch>

Use the -f option flag to force it.

[**Undo Last Push**](https://git-scm.com/docs/git-reset)

Some would say this is bad practice. Once you push something you shouldn't overwrite those changes. Instead, you're supposed to create a new commit that reverts the changes in the last one. So, technically, you shouldn't do this, but... you can?

git reset --hard HEAD~1 && git push -f origin master

[**Fetching**](https://git-scm.com/docs/git-fetch)

Fetch changes from upstream:

git fetch upstream

Fetch changes from both origin and upstream in the same shot:

git fetch --multiple origin upstream

[**Merging**](https://git-scm.com/docs/git-merge)

To be honest, I haven't used this command in quite some time. In my experience, it has created merge bubbles that have overwritten mine or others' code. For a better workflow, refer to rebasing, below.

Nonetheless, here's how you merge-in changes from origin's feature\_x branch:

git fetch origin

git merge origin/feature\_x

[**Pulling**](https://www.kernel.org/pub/software/scm/git/docs/git-pull.html)

Pulling is just doing a fetch followed by a merge. If you know your branch is clean (e.g., master branch), go ahead and get the latest changes. There will be no merge conflicts as long as your branch is clean.

git pull origin/feature\_x

[**Rebasing**](https://git-scm.com/docs/git-rebase)

Rebasing is a way of rewriting history. In place of merge, what this does is stacks your commits on top of commits that are already pushed up. In this case, you want to stack your commits on top of origin/feature\_x:

git fetch origin

git rebase origin/feature\_x

If you already have a local branch set to track feature\_x then just do:

git rebase feature\_x

Would you like to fetch, merge and then stack your changes on top, all in one shot? You can! If you have tracking setup on the current branch, just do:

git pull --rebase

Another great use of rebasing is just rewriting commit messages. To get an interactive text editor for the most recent commit, do:

git rebase -i HEAD~1

Now, you can replace "pick" with "r" and just change the commit message.

[**Manually Set Tracking**](https://git-scm.com/docs/git-config#Documentation/git-config.txt-branchltnamegtremote)

Perhaps you forgot to setup tracking when you pulled down a remote branch. No worries:

git config branch.<local\_branch>.remote origin

git config branch.<local\_branch>.merge refs/heads/<remote\_branch>

[**Deleting Branches**](https://git-scm.com/docs/git-branch#Documentation/git-branch.txt--d)

Delete a local branch:

git branch -d <local\_branch>

Use the -D option flag to force it.

Delete a remote branch on origin:

git push origin :<remote\_branch>

[**Stashing**](https://git-scm.com/docs/git-stash)

Sometimes you need to stash your changes so you can be on a clean branch or maybe because you want to go back and try something before you make a commit with these changes. Here's how you do a stash:

git stash

Now, if you want to unstash those changes and bring them back into your working directory:

git stash pop

Or maybe you want to unstash your changes without popping them off the stack. In other words, you might want to apply these stashed changes multiple times. To do this:

git stash apply

For a list of stashes:

git stash list

And to apply a specific stash from that list (e.g., stash@{3}):

git stash apply stash@{3}

**Tagging**

[**Fetch**](https://git-scm.com/docs/git-fetch#Documentation/git-fetch.txt--t)

git fetch --tags

[**Create**](https://git-scm.com/docs/git-tag)

git tag -a v1.0.0 9fceb02 -m "Initial release"

Where 9fceb02 represents the commit hash.

[**Push**](https://git-scm.com/docs/git-push#Documentation/git-push.txt---tags)

git push --tags

[**Delete**](https://git-scm.com/docs/git-tag#Documentation/git-tag.txt--d)

git tag -d v1.0.0

[**Remote**](https://git-scm.com/docs/git-push)

git push origin :refs/tags/v1.0.0