

Configuration

User config:

```
$ git config --global user.name "name"
$ git config --global user.email "email"
$ git config --global color.ui true
```

Print config options:

```
$ git config --list
```

Remote commands

Download new commits from remote repo:

```
$ git fetch [remote-name]
```

Download new commits from remote and merge:

```
$ git pull [remote-name]
```

Push to remote repo:

```
$ git push [remote-name] [branch-name]
```

Maintenance

List remote repos:

```
$ git remote -v
```

What happened at the remote?

```
$ git remote show origin
```

Ignoring files:

```
$ vi .gitignore
```

Add a remote repo:

```
$ git remote add <name> <url>
```

Branching

List branches, the one you have checked out is prefixed with a *:

```
$ git branch [-v]
$ git branch -a # list remotes
```

Create new branch, which points to current position:

```
$ git branch <branch-name>
```

Switch to a branch:

```
$ git checkout <branch-name>
```

To create the branch on github, just push:

```
$ git push origin <branch-name>
```

Checkout specific remote branch locally:

```
$ git checkout -b <bn> origin/<bn>
```

Delete a branch, just deletes a pointer, not commits:

```
$ git branch -d <branch-name>
```

Merging

List branches which have not been merged:

```
$ git branch --no-merged
```

Suppose you have a master branch and a feature branch. You want to merge feature back into master:

```
# finish up working on feature,
# commit everything, test
$ git checkout master
$ git merge feature
```

This will create a special merge commit, which points to the two parents commits in master and feature. If there are conflicts, they will be put in the working area. Once they're resolved, commit them. After the merge is complete, the feature branch may be deleted:

```
$ git branch -d feature
$ git push origin :feature # push delete
```

List branches which have been merged, these are usually to be deleted:

```
$ git branch --merged
```

Tagging

Tagging:

```
$ git tag -a <tag> <commit-hash>
```

By default tags are not pushed:

```
$ git push origin v1.5
```

Local commands

Add all tracked files:

```
$ git add -u
```

Commit with message:

```
$ git commit -m '<commit-message>'
```

Amend previous commit:

```
$ git commit --amend -m '<commit-message>'
```

IMPORTANT: Do not amend commits that you have pushed to a public repository!

Stop tracking the file (leaves file alone):

```
$ git rm --cached <file>
```

Unstage file (leaves file alone):

```
$ git reset HEAD <file>
```

Revert file (changes file):

```
$ git checkout -- <file>
```

Delete last commit (changes files):

```
$ git reset --hard HEAD~1
```

IMPORTANT: Do not delete commits that you have pushed to a public repository!

Rebasing

Rebasing is flattening the commit graph instead of creating a merge commit that points to the two parent commits. It takes the diffs of the commits in a feature branch and creates new commits for them on the master branch:

```
$ git checkout feature
$ git rebase master
```

The endresult will be new commits put at the end of master, containing equivalent but different commits than the ones on feature. The old feature commits are deleted, the feature pointer points to HEAD, the master pointer is unchanged. Then you can fast-forward master:

```
$ git checkout master
$ git merge feature # fast-forward
$ git branch -d feature # delete feature branch
```

IMPORTANT: Do not rebase commits that you have pushed to a public repository!

Stashing

List stashes:

```
$ git stash list
```

Clear all stashes:

```
$ git stash clear
```

Drop specific stashes:

```
$ git stash drop <stash-name>
```

Save your local modifications to a new stash (saves tracked files and the staging area):

```
$ git stash # auto named like stash@{0}
# or
$ git stash save <stash-name>
```

Restore stash (working area only):

```
$ git stash apply
$ git stash pop # apply and remove
```

Restore stash (also re-stage):