Git Cheatsheet

[Raw](https://gist.github.com/hofmannsven/6814451/raw/d919c3c387ea1e01e18acc88299e16bcb79fcc74/README.md)

[**README.md**](https://gist.github.com/hofmannsven/6814451#file-readme-md)

**Git**

**Global Settings**

* Related Setup: <https://gist.github.com/hofmannsven/6814278>
* Related Pro Tips: <https://ochronus.com/git-tips-from-the-trenches/>
* Interactive Beginners Tutorial: <http://try.github.io/>
* Git Cheatsheet by GitHub: <https://services.github.com/on-demand/downloads/github-git-cheat-sheet/>

**Reminder**

Press minus + shift + s and return to chop/fold long lines!

Show folder content: ls -la

**Notes**

Do not put (external) dependencies in version control!

**Setup**

See where Git is located: which git

Get the version of Git: git --version

Create an alias (shortcut) for git status: git config --global alias.st status

Help: git help

**General**

Initialize Git: git init

Get everything ready to commit: git add .

Get custom file ready to commit: git add index.html

Commit changes: git commit -m "Message"

Commit changes with title and description: git commit -m "Title" -m "Description..."

Add and commit in one step: git commit -am "Message"

Remove files from Git: git rm index.html

Update all changes: git add -u

Remove file but do not track anymore: git rm --cached index.html

Move or rename files: git mv index.html dir/index\_new.html

Undo modifications (restore files from latest commited version): git checkout -- index.html

Restore file from a custom commit (in current branch): git checkout 6eb715d -- index.html

**Reset**

Go back to commit: git revert 073791e7dd71b90daa853b2c5acc2c925f02dbc6

Soft reset (move HEAD only; neither staging nor working dir is changed): git reset --soft 073791e7dd71b90daa853b2c5acc2c925f02dbc6

Undo latest commit: git reset --soft HEAD~

Mixed reset (move HEAD and change staging to match repo; does not affect working dir): git reset --mixed 073791e7dd71b90daa853b2c5acc2c925f02dbc6

Hard reset (move HEAD and change staging dir and working dir to match repo): git reset --hard 073791e7dd71b90daa853b2c5acc2c925f02dbc6

Hard reset of a single file (@ is short for HEAD): git checkout @ -- index.html

**Update & Delete**

Test-Delete untracked files: git clean -n

Delete untracked files (not staging): git clean -f

Unstage (undo adds): git reset HEAD index.html

Update most recent commit (also update the commit message): git commit --amend -m "New Message"

**Branch**

Show branches: git branch

Create branch: git branch branchname

Change to branch: git checkout branchname

Create and change to new branch: git checkout -b branchname

Rename branch: git branch -m branchname new\_branchname or: git branch --move branchname new\_branchname

Show all completely merged branches with current branch: git branch --merged

Delete merged branch (only possible if not HEAD): git branch -d branchname or: git branch --delete branchname

Delete not merged branch: git branch -D branch\_to\_delete

**Merge**

True merge (fast forward): git merge branchname

Merge to master (only if fast forward): git merge --ff-only branchname

Merge to master (force a new commit): git merge --no-ff branchname

Stop merge (in case of conflicts): git merge --abort

Stop merge (in case of conflicts): git reset --merge // prior to v1.7.4

Undo local merge that hasn't been pushed yet: git reset --hard origin/master

Merge only one specific commit: git cherry-pick 073791e7

Rebase: git checkout branchname » git rebase master or: git merge master branchname (The rebase moves all of the commits in master onto the tip of branchname.)

Cancel rebase: git rebase --abort

Squash multiple commits into one: git rebase -i HEAD~3 ([source](https://www.devroom.io/2011/07/05/git-squash-your-latests-commits-into-one/))

Squash-merge a feature branch (as one commit): git merge --squash branchname (commit afterwards)

**Stash**

Put in stash: git stash save "Message"

Show stash: git stash list

Show stash stats: git stash show stash@{0}

Show stash changes: git stash show -p stash@{0}

Use custom stash item and drop it: git stash pop stash@{0}

Use custom stash item and do not drop it: git stash apply stash@{0}

Use custom stash item and index: git stash apply --index

Create branch from stash: git stash branch new\_branch

Delete custom stash item: git stash drop stash@{0}

Delete complete stash: git stash clear

**Gitignore & Gitkeep**

About: <https://help.github.com/articles/ignoring-files>

Useful templates: <https://github.com/github/gitignore>

Add or edit gitignore: nano .gitignore

Track empty dir: touch dir/.gitkeep

**Log**

Show commits: git log

Show oneline-summary of commits: git log --oneline

Show oneline-summary of commits with full SHA-1: git log --format=oneline

Show oneline-summary of the last three commits: git log --oneline -3

Show only custom commits: git log --author="Sven" git log --grep="Message" git log --until=2013-01-01 git log --since=2013-01-01

Show only custom data of commit: git log --format=short git log --format=full git log --format=fuller git log --format=email git log --format=raw

Show changes: git log -p

Show every commit since special commit for custom file only: git log 6eb715d.. index.html

Show changes of every commit since special commit for custom file only: git log -p 6eb715d.. index.html

Show stats and summary of commits: git log --stat --summary

Show history of commits as graph: git log --graph

Show history of commits as graph-summary: git log --oneline --graph --all --decorate

**Compare**

Compare modified files: git diff

Compare modified files and highlight changes only: git diff --color-words index.html

Compare modified files within the staging area: git diff --staged

Compare branches: git diff master..branchname

Compare branches like above: git diff --color-words master..branchname^

Compare commits: git diff 6eb715d git diff 6eb715d..HEAD git diff 6eb715d..537a09f

Compare commits of file: git diff 6eb715d index.html git diff 6eb715d..537a09f index.html

Compare without caring about spaces: git diff -b 6eb715d..HEAD or: git diff --ignore-space-change 6eb715d..HEAD

Compare without caring about all spaces: git diff -w 6eb715d..HEAD or: git diff --ignore-all-space 6eb715d..HEAD

Useful comparings: git diff --stat --summary 6eb715d..HEAD

Blame: git blame -L10,+1 index.html

**Releases & Version Tags**

Show all released versions: git tag

Show all released versions with comments: git tag -l -n1

Create release version: git tag v1.0.0

Create release version with comment: git tag -a v1.0.0 -m 'Message'

Checkout a specific release version: git checkout v1.0.0

**Collaborate**

Show remote: git remote

Show remote details: git remote -v

Add remote upstream from GitHub project: git remote add upstream https://github.com/user/project.git

Add remote upstream from existing empty project on server: git remote add upstream ssh://root@123.123.123.123/path/to/repository/.git

Fetch: git fetch upstream

Fetch a custom branch: git fetch upstream branchname:local\_branchname

Merge fetched commits: git merge upstream/master

Remove origin: git remote rm origin

Show remote branches: git branch -r

Show all branches (remote and local): git branch -a

Create and checkout branch from a remote branch: git checkout -b local\_branchname upstream/remote\_branchname

Compare: git diff origin/master..master

Push (set default with -u): git push -u origin master

Push: git push origin master

Force-Push: `git push origin master --force

Pull: git pull

Pull specific branch: git pull origin branchname

Fetch a pull request on GitHub by its ID and create a new branch: git fetch upstream pull/ID/head:new-pr-branch

Clone to localhost: git clone https://github.com/user/project.git or: git clone ssh://user@domain.com/~/dir/.git

Clone to localhost folder: git clone https://github.com/user/project.git ~/dir/folder

Clone specific branch to localhost: git clone -b branchname https://github.com/user/project.git

Clone with token authentication (in CI environment): git clone https://oauth2:<token>@gitlab.com/username/repo.git

Delete remote branch (push nothing): git push origin :branchname or: git push origin --delete branchname

**Archive**

Create a zip-archive: git archive --format zip --output filename.zip master

Export/write custom log to a file: git log --author=sven --all > log.txt

**Troubleshooting**

Ignore files that have already been committed to a Git repository: <http://stackoverflow.com/a/1139797/1815847>

**Security**

Hide Git on the web via .htaccess: RedirectMatch 404 /\.git (more info here: <http://stackoverflow.com/a/17916515/1815847>)

**Large File Storage**

Website: <https://git-lfs.github.com/>

Install: brew install git-lfs

Track \*.psd files: git lfs track "\*.psd" (init, add, commit and push as written above)

# Git Commands

### A Summary of Commonly Used Commands

There are hundreds of different Git commands, but to get started you only need to remember a handful of them. Here is a summary of the commands you’ll use most often:

* git init initializes your local directory as a new git repository. You must run this before you can commit any of your work.
* git status shows the current status of your repo. It will show you if you have any work that is unstaged, what branch you are on, how many commits you are ahead of the master remote on github, and other useful things.
* git diff shows you the changes in your unstaged code.
* git remote -v shows you all the remotes for your repo. The v stands for verbose, which shows you the URL of the repository on github, if any, that your local repository is pointing to rather than just the name of the remote repo.
* git add . takes all unstaged work and stages it, making it ready to be committed. You can also specify a particular file to stage with git add file-path/name-of-file
* git commit -m "write commit message here" commits all staged work. It’s important to write a brief, clear commit message so you know what each commit is for. “Final commit” is not the commit message you’re looking for exactly 100% of the time.
* git pull once you’ve committed all your local work and running git status shows that you have nothing to commit, you pull down any changes from your remote. By default, this will pull from the origin remote’s master branch. To be specific about which remote and branch to pull from, you can use: git pull name-of-remote name-of-branch
* git push pushes your local changes up to your remote. By default, this will push to the origin remote’s master branch. Like pull, you can push to a specific remote and branch with: git push name-of-remote name-of-branch. This is useful if you are using [branches](https://git-scm.com/book/en/v2/Git-Branching-Basic-Branching-and-Merging) and [pull requests](https://git-scm.com/book/en/v2/GitHub-Contributing-to-a-Project#The-GitHub-Flow). If you get an error message, it’s probably because you haven’t pushed your local branch up to github yet. Try git push -u name-of-remote name-of-branch.
* git branch shows you all your local branches and indicates which branch you are currently on.
* git checkout -b name-of-new-branch makes a new branch and switches to that branch.
* git merge name-of-branch will merge the specified branch into the branch you are currently on.
* git branch -d name-of-branch-to-delete deletes the specified branch
* git log will show you the full list of commits and authors for your repo
* history will show you your past git commands
* git stash stashes any unstaged changes in your repository. They will not be present in your codebase, but they are not deleted.
* git stash pop gives you back the last staged changes you stashed
* git blame file-path/name-of-file shows you line-by-line who wrote the code in the specified file. Useful when you have a question about how something works and want to figure out who to ask, and also great source of shame when you realize you wrote the chunk of code you’ve been swearing at for the last hour.

**Git Cheat Sheet 📄 (50 commands + PDF and poster)**

[#git](https://dev.to/t/git)[#github](https://dev.to/t/github)[#programming](https://dev.to/t/programming)[#productivity](https://dev.to/t/productivity)

I was tired of looking up the same common Git commands - so I made a cheat sheet that I could print and put on my office wall.

This cheat sheet contains 50 commonly used Git commands on the following topics:

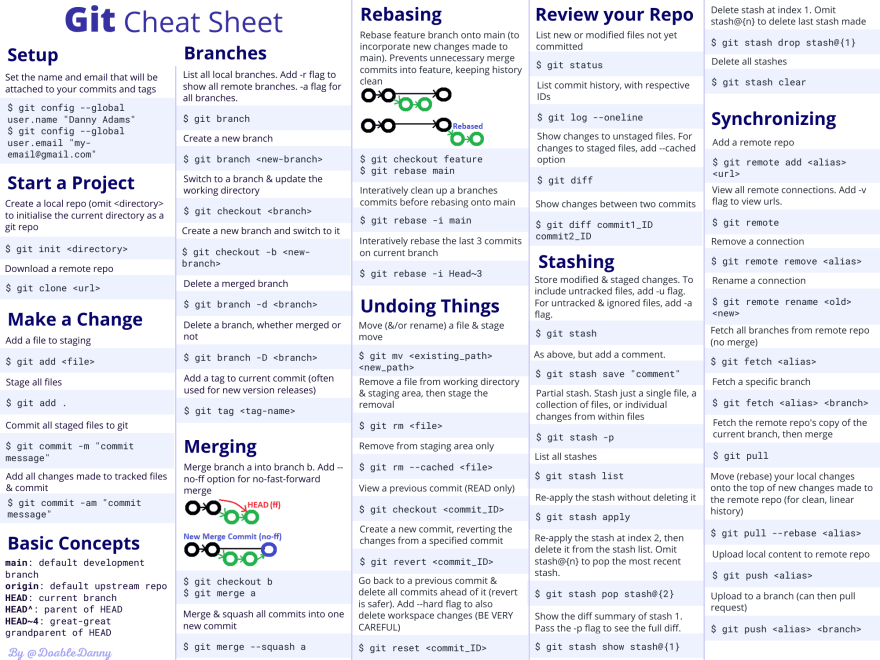
* Setting up Git
* Starting a project
* Making a change
* Basic concepts
* Branching
* Merging
* Rebasing
* Undoing things
* Reviewing your repo
* Stashing
* Synchronising local and remote repositories

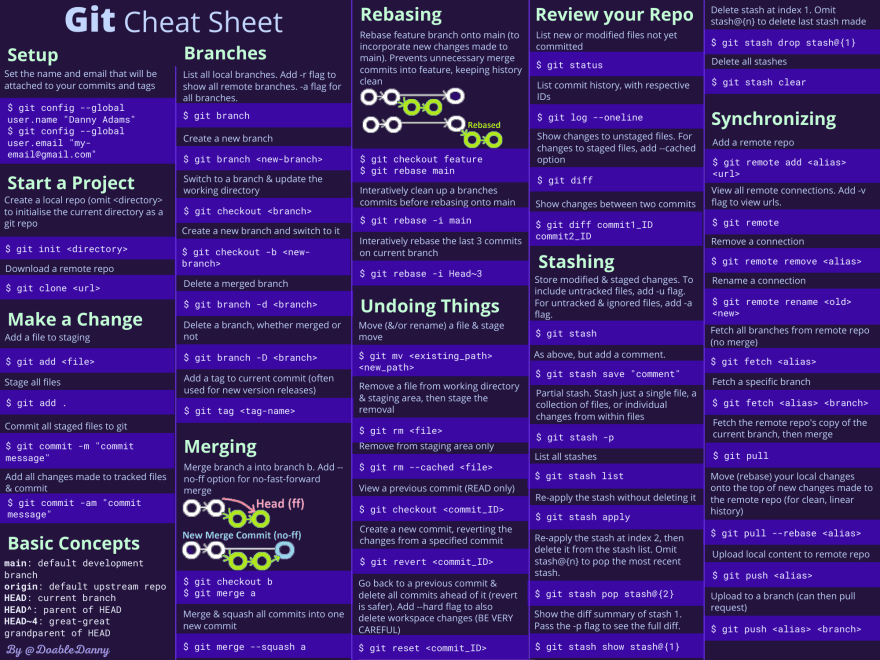
**Git Commands Cheat Sheet PDF**

One page PDF to make it easy to copy and paste in commands.

[Download the Git Commands Cheat Sheet PDF here](https://doabledanny.gumroad.com/l/git-commands-cheat-sheet-pdf)

Both PDF and poster are available in Light Mode and Dark Mode:

[](https://res.cloudinary.com/practicaldev/image/fetch/s--Zib71Fgv--/c_limit%2Cf_auto%2Cfl_progressive%2Cq_auto%2Cw_880/https:/dev-to-uploads.s3.amazonaws.com/uploads/articles/n082uxea33j6zq3mca7u.png)

[](https://res.cloudinary.com/practicaldev/image/fetch/s--w-UEE2WQ--/c_limit%2Cf_auto%2Cfl_progressive%2Cq_auto%2Cw_880/https:/dev-to-uploads.s3.amazonaws.com/uploads/articles/8cl1lotlaikqp8vhhr6i.png)

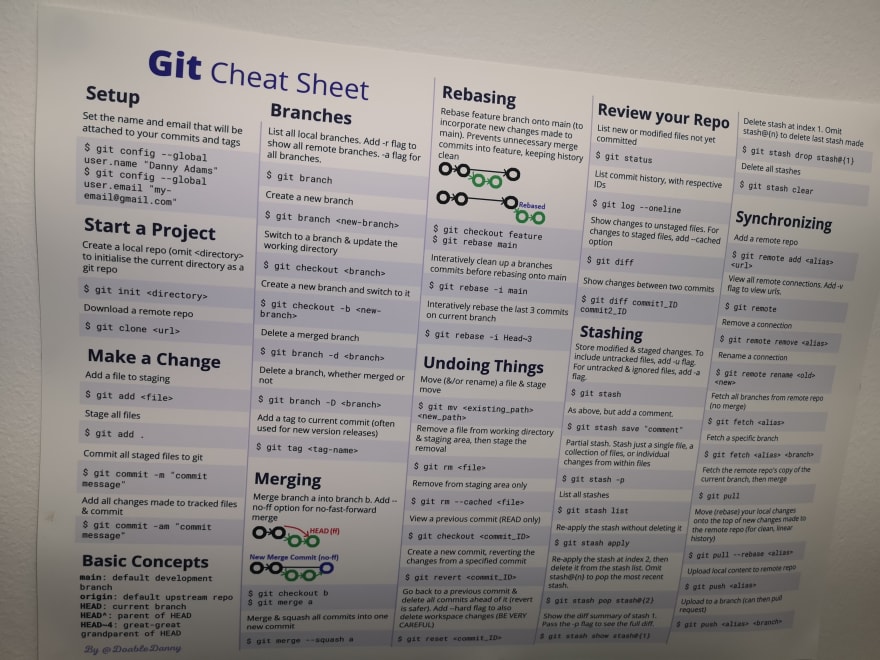
**Git Cheat Sheet Poster**

Order a physical A3 poster for your office wall - so you can quickly look up commands, and keep them at the top of your head.

It comes in thick durable paper, and a matte, light-absorbing finish.

[Order a Git Cheat Sheet Poster here](https://doabledanny.gumroad.com/l/git-cheat-sheet-poster)

Here's mine on my office wall:

[](https://res.cloudinary.com/practicaldev/image/fetch/s--XUn-4X_T--/c_limit%2Cf_auto%2Cfl_progressive%2Cq_auto%2Cw_880/https:/dev-to-uploads.s3.amazonaws.com/uploads/articles/60mjbucqvp3ojsq0fowy.jpg)

Here are all of the commands from the cheat sheet:

**Setup**

Set the name and email that will be attached to your commits and tags

$ git config --global user.name "Danny Adams"

$ git config --global user.email "myemail@gmail.com"

**Starting a Project with Git**

Create a local repo (omit <directory> to initialise the current directory as a git repo)

$ git init <directory>

Download a remote repo

$ git clone <url>

**Make a Change**

Add a file to staging

$ git add <file>

Stage all files

$ git add .

Commit all staged files to git

$ git commit -m "commit message"

Add all changes made to tracked files & commit

$ git commit -am "commit message"

**Basic Git Concepts**

**main**: default development branch  
**origin**: default upstream repo  
**HEAD**: current branch  
**HEAD^**: parent of HEAD  
**HEAD~4**: great-great grandparent of HEAD

**Branches**

List all local branches. Add -r flag to show all remote branches. -a flag for all branches.

$ git branch

Create a new branch

$ git branch <new-branch>

Switch to a branch & update the working directory

$ git checkout <branch>

Create a new branch and switch to it

$ git checkout -b <newbranch>

Delete a merged branch

$ git branch -d <branch>

Delete a branch, whether merged or  
not

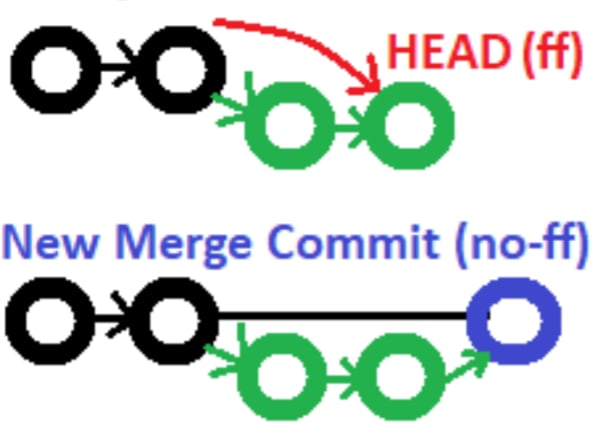
$ git branch -D <branch>

Add a tag to current commit (often used for new version releases)

$ git tag <tag-name>

**Merging**

Merge branch a into branch b. Add --no-ff option for no-fast-forward merge

[](https://res.cloudinary.com/practicaldev/image/fetch/s--P-4I6vZM--/c_limit%2Cf_auto%2Cfl_progressive%2Cq_auto%2Cw_880/https:/dev-to-uploads.s3.amazonaws.com/uploads/articles/2zpi27oxqws7j39iaxs2.PNG)

$ git checkout b

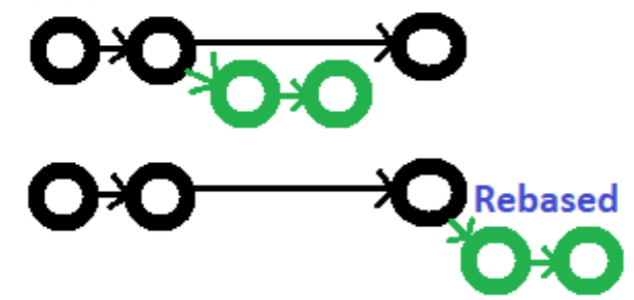
$ git merge a

Merge & squash all commits into one new commit

$ git merge --squash a

**Rebasing**

Rebase feature branch onto main (to incorporate new changes made to main). Prevents unnecessary merge commits into feature, keeping history clean

[](https://res.cloudinary.com/practicaldev/image/fetch/s--6pTlAeTV--/c_limit%2Cf_auto%2Cfl_progressive%2Cq_auto%2Cw_880/https:/dev-to-uploads.s3.amazonaws.com/uploads/articles/99lln9qgsiiwz8w18vdp.PNG)

$ git checkout feature

$ git rebase main

Interactively clean up a branches commits before rebasing onto main

$ git rebase -i main

Interactively rebase the last 3 commits on current branch

$ git rebase -i Head~3

**Undoing Things**

Move (&/or rename) a file & stage move

$ git mv <existing\_path> <new\_path>

Remove a file from working directory & staging area, then stage the removal

$ git rm <file>

Remove from staging area only

$ git rm --cached <file>

View a previous commit (READ only)

$ git checkout <commit\_ID>

Create a new commit, reverting the changes from a specified commit

$ git revert <commit\_ID>

Go back to a previous commit & delete all commits ahead of it (revert is safer). Add --hard flag to also delete workspace changes (BE VERY CAREFUL)

$ git reset <commit\_ID>

**Review your Repo**

List new or modified files not yet committed

$ git status

List commit history, with respective IDs

$ git log --oneline

Show changes to unstaged files. For changes to staged files, add --cached option

$ git diff

Show changes between two commits

$ git diff commit1\_ID commit2\_ID

**Stashing**

Store modified & staged changes. To include untracked files, add -u flag. For untracked & ignored files, add -a flag.

$ git stash

As above, but add a comment.

$ git stash save "comment"

Partial stash. Stash just a single file, a collection of files, or individual changes from within files

$ git stash -p

List all stashes

$ git stash list

Re-apply the stash without deleting it

$ git stash apply

Re-apply the stash at index 2, then delete it from the stash list. Omit stash@{n} to pop the most recent stash.

$ git stash pop stash@{2}

Show the diff summary of stash 1. Pass the -p flag to see the full diff.

$ git stash show stash@{1}

Delete stash at index 1. Omit stash@{n} to delete last stash made

$ git stash drop stash@{1}

Delete all stashes

$ git stash clear

**Synchronizing**

Add a remote repo

$ git remote add <alias> <url>

View all remote connections. Add -v flag to view urls.

$ git remote

Remove a connection

$ git remote remove <alias>

Rename a connection

$ git remote rename <old> <new>

Fetch all branches from remote repo (no merge)

$ git fetch <alias>

Fetch a specific branch

$ git fetch <alias> <branch>

Fetch the remote repo's copy of the current branch, then merge

$ git pull

Move (rebase) your local changes onto the top of new changes made to the remote repo (for clean, linear history)

$ git pull --rebase <alias>

Upload local content to remote repo

$ git push <alias>

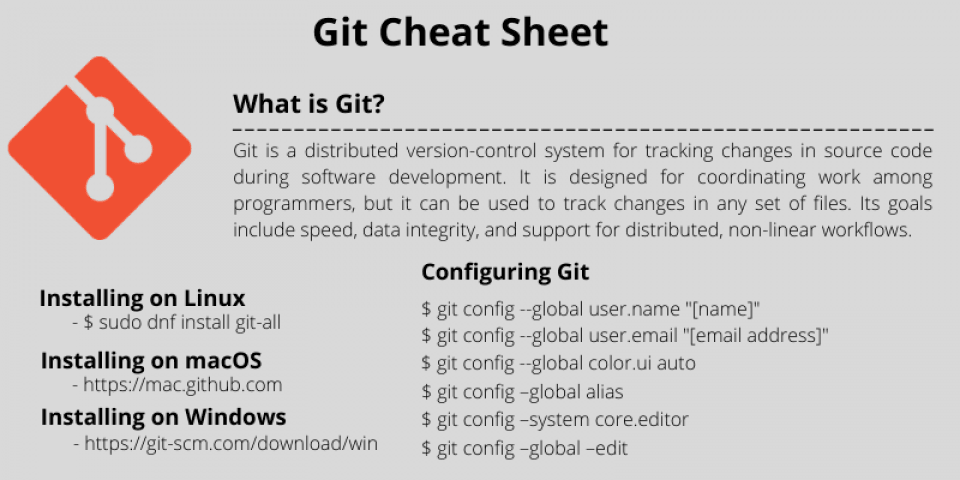
Upload to a branch (can then pull request)

$ git push <alias> <branch>

**Thanks for reading**

# Git Cheat Sheet: Download PDF for Quick Reference

Posted in [Git](https://hackr.io/blog/category/git), [Cheat Sheet](https://hackr.io/blog/category/cheat-sheet)



A picture containing person, outdoor, smiling, posing

Description automatically generated

[Sindhuja Hari](https://hackr.io/blog/author/sindhuja-hari)

Last Updated 15 Dec, 2022

**Share:**  

[**1 Comment**](https://hackr.io/blog/git-cheat-sheet#comment-link)

## Table of Contents

Version control is one of the significant challenges faced by developers since the genesis of coding. Pushing source code into a central server is what developers usually do as part of version control. Whether it is Java, .Net, Python, or Database code, when an application under development, there is minimum communication among developers. Hence, there are significant conflicts over code version controls, tracking, and retrieval, which could potentially lead to delays in project or product release. Version Control Software (VCS) are tools that help in software configuration management and help to keep track of the source code and builds application components or modules.

## ****What is Git?****

‘’Git’’ is a go-to-version control tool that allows developers to access all of the code and efficiently manage their source code and track file changes be it small, medium, or massive application development. It remains the most widely used [open-source distributed version control system](https://en.wikipedia.org/wiki/Distributed_version_control) (DVCS) till date and has been in use for over a decade after its initial release. Unlike other version control systems that store a project’s full version history in one place, Git gives each developer their repository locally containing the entire history of changes and the entire application.

* Git is used to tracking changes in the source code hence tracks history
* Git is a distributed version control tool for source code management. It is free and open-source.
* Git creates backup automatically, as the developer has a version of the code on the local repository.
* Git allows multiple developers to work together; hence is scalable and supports collaboration.
* Git supports non-linear development because of its thousands of parallel branches.

This Git cheat sheet can serve as a ready reckoner to [essential git commands](https://hackr.io/blog/basic-git-commands-with-examples) that you may need to use in your coding career.

### **Installing Git**

Before you start using Git, you have to make it available on your computer. Even Though if it is installed, it is probably a good idea to update to the latest version. You can either install it as a package or via another installer or download the source code and compile it yourself.

### **Installing on Linux**

If you want to install the essential Git tools on Linux via a binary installer, you can generally do so through the package management tool that comes with your distribution. If you are on operating systems such as Fedora, you can use dnf.

3

## Git Cheat Sheet: Quick Guide for Reference

|  |  |  |
| --- | --- | --- |
| Installing on Linux | If you want to install the basic Git tools on Linux via a binary installer, you can generally do so through the package management tool that comes with your distribution. If you are on operating systems such as Fedora you can use dnf. | $ sudo dnf install git-all |
|  | If you are on a Debian-based distribution, such as Ubuntu use the following command | $ sudo apt install git-all |
| Installing on macOS | There are several ways to install Git on a Mac. The easiest way is to install it from the Github website | https://mac.github.com |
| Installing on Windows | There are also a few ways to install Git on Windows. The most official build is available for download on the Git website. | <https://git-scm.com/download/win> |

### **Configuring Git**

Configure user information for all local repositories on your computer

|  |  |
| --- | --- |
| $ git config --global user.name "[name]" | Sets the name you want to be attached to your commit transactions. |
| $ git config --global user.email "[email address]" | Sets the email you want to beattached to your commit transactions. |
| $ git config --global color.ui auto | Enables helpful colorization of the command line output |
| $ git config –global alias | Creates a Git command shortcut |
| $ git config –system core.editor | Sets the preferred text editor |
| $ git config –global –edit | Open and edit the global configuration file in the text editor |

### **Setting Up Git Repositories**

|  |  |
| --- | --- |
| $ git init [project-name] | Creates an empty repository in the project folder with the specified name |
| $ git clone (repo URL) | Downloads a project from a remote service such as Github and its entire version history |
| $ git clone (repo URL) (folder) | Clones a repository to a specific folder |
| $ git remote -v | Displays a list of remote repositories with URLs |
| $ git remote rm (remote repo name) | Removes a remote repository |
| $ git fetch | Fetching from a repository grabs all the new remote-tracking branches and tags without merging those changes into your own branches. |
| $ git pull | Retrieve the most recent changes from origin and merge |

### **Managing File Changes**

|  |  |
| --- | --- |
| $ git add (file name) | Adds file changes to staging. Snapshots the file in preparation for versioning. |
| $ git add | Adds all directory changes to staging |
| $ git add -A | Adds new and modified files to staging |
| $ git rm (file\_name) | Removes a file and stops tracking it. Deletes the file from the working directory and stages the deletion |
| $ git rm –cached (file\_name) | Removes the file from version control but preserves the file locally |
| $ git checkout <deleted file name> | Recovers a deleted file and prepares it for commit |
| $ git status | Displays the status of modified files. Lists all new or modified files to be committed |
| $ git diff | Displays all unstaged changes in the index and the current directory. Shows file differences that are not yet staged |
| $ git diff --staged | Shows file differences between staging and the last file version. |
| $ git reset [file] | Unstages the file, but preserve its contents |
| $ git commit -m "[descriptive message]" | Records file snapshots permanently in version history |
| $ git mv [file-original] [file-renamed] | Changes the file name and prepares it for commit |

### **REDO COMMITS**

Erase mistakes: You would typically want to UNDO/REDO when you commit some changes to Git and realize that the changes need to be removed/reverted.

|  |  |
| --- | --- |
| $ git reset [commit] | Undo all commits after [commit], preserving changes locally |
| $ git reset --hard [commit] | Discards all history and changes back to the specified commit |

#### **GROUP CHANGES: Commands for Git branching**

You can decide how to group the changes to create meaningful commits.

|  |  |
| --- | --- |
| $ git branch | Lists all local branches in the current repository |
| $ git branch [branch-name] | Creates a new branch |
| $ git checkout [branch-name] | Switches to the specified branch and updates the working directory |
| $ git merge [branch] | Combines the specified branch’s history into your current branch |
| $ git branch -d [branch-name] | Deletes the specified branch |
| $ git fetch remote <branchname> | Fetches a branch from the repository |
| $ git push –all | Pushes all local branches to a designated remote repository |

### **SAVE FRAGMENTS**

The Git stash command removes changes from your index and “stashes” them away for later. It is useful if you wish to pause what you are doing and work on something else for a while. You cannot stash more than one set of changes at a time.

|  |  |
| --- | --- |
| $ git stash | Temporarily stores all modified tracked files |
| $ git stash pop | Restores the most recently stashed files |
| $ git stash list | Lists all stashed changesets |
| $ git stash drop | Discards the most recently stashed changeset |

### **Review History**

Browse and view the version history of your project files.

|  |  |
| --- | --- |
| $ git log | Lists version history for the current branch |
| $ git log --follow [file] | Lists version history for a file, including renames |
| $ git diff [first-branch]...[second-branch] | Shows content differences and conflicts between two branches |
| $ git show [commit] | Outputs metadata and content changes of the specified commit |

### **Git Glossary**

|  |  |
| --- | --- |
| Branch | Branches represent specific versions of a repository that “branch out” from your main project. Branches allow you to keep track of experimental changes you make to repositories and revert to older versions |
| Commit | Each time you save/commit the state of your project in Git, it basically takes a picture of what all your files look like at that moment and stores a reference to that snapshot. To be efficient, if the files have not changed, Git does not store the file again but has a link to the previous identical file it has already stored. |
| Checkout | The git checkout command switches between branches or restores working tree files. It is used to undo the effects of changes to your repository. |
| Fetch | The Git fetch command copies and downloads all of a branch’s files to your device. Use it to save the latest changes to your repositories. It is possible to fetch multiple branches simultaneously |
| Index | Whenever you add, delete or alter a file, it remains in the index until you are ready to commit the changes. It is like a staging area for Git. Use the Git status command to see the contents of your index. |
| Repositories | Git repositories hold all of your project’s files including branches, tags, and commits |
| Pull | Pull requests represent suggestions for changes to the master branch. The Git pull command is used to add changes to the master branch |
| Push | The git push command is used to update remote branches with the latest changes you have committed |
| Stash | Stashing takes the state of your working directory — that is, your modified tracked files and staged changes — and saves it on a stack of unfinished changes that you can reapply at any time. |
| Staging | The staging area is a simple file, generally contained in your Git directory that stores information about what will go into your next commit. It's sometimes referred to as the index. |

[**The Git & Github Bootcamp**](https://click.linksynergy.com/link?id=jU79Zysihs4&offerid=1045023.3105206&type=2&murl=https%3A%2F%2Fwww.udemy.com%2Fcourse%2Fgithub-git%2F&u1=blog%2Fgit-cheat-sheet_amcid-A3VAil7QzIVNe6YIpgUzi)

## ****Summary****

Git cheat sheet is a broad set of command guidelines for version control. As a coder, it is just next to impossible to keep all the commands inside the head. Hence this Git cheat sheet can be your pocket ready reckoner when you need to know a specific command.

## How do I change directory permissions in Linux?

To change directory permissions in Linux, use the following:

* chmod +rwx filename to add permissions
* chmod -rwx directoryname to remove permissions.
* chmod +x filename to allow executable permissions.
* chmod -wx filename to take out write and executable permissions.

Note that “r” is for read, “w” is for write, and “x” is for execute.

This only changes the permissions for the owner of the file.

## What are the three permission groups?

There are three options for permission groups available to you in Linux. These are

* owners: these permissions will only apply to owners and will not affect other groups.
* groups: you can assign a group of users specific permissions, which will only impact users within the group.
* all users: these permissions will apply to all users, and as a result, they present the greatest security risk and should be assigned with caution.

## What are the three kinds of file permissions in Linux?

There are three kinds of file permissions in Linux:

* Read (r): Allows a user or group to view a file.
* Write (w): Permits the user to write or modify a file or directory.
* Execute (x): A user or grup with execute permissions can execute a file or view a directory.

## More ways to manage permissions

Here's a more comprehensive list of ways you can manage file permissions, groups, and ownership beyond the basic commands listed at the top of this guide.

### **How to Change Directory Permissions in Linux for the Group Owners and Others**

The command for changing directory permissions for group owners is similar, but add a “g” for group or “o” for users:

* chmod g+w filename
* chmod g-wx filename
* chmod o+w filename
* chmod o-rwx foldername

To change directory permissions for everyone, use “u” for users, “g” for group, “o” for others, and “ugo” or “a” (for all).

* chmod ugo+rwx foldername to give read, write, and execute to everyone.
* chmod a=r foldername to give only read permission for everyone.

### **How to Change Groups of Files and Directories in Linux**

By issuing these commands, you can change groups of files and directories in Linux.

* chgrp groupname filename
* chgrp groupname foldername

Note that the group must exit before you can assign groups to files and directories.

### **Changing ownership in Linux**

Another helpful command is changing ownerships of files and directories in Linux:

* chown name filename
* chown name foldername

These commands will give ownership to someone, but all sub files and directories still belong to the original owner.

You can also combine the group and ownership command by using:

* chown -R name:filename /home/name/directoryname

### **Changing Linux permissions in numeric code**

You may need to know how to change permissions in numeric code in Linux, so to do this you use numbers instead of “r”, “w”, or “x”.

* 0 = No Permission
* 1 = Execute
* 2 = Write
* 4 = Read

Basically, you add up the numbers depending on the level of permission you want to give.

Permission numbers are:

* 0 = ---
* 1 = --x
* 2 = -w-
* 3 = -wx
* 4 = r-
* 5 = r-x
* 6 = rw-
* 7 = rwx

For example:

* chmod 777 foldername will give read, write, and execute permissions for everyone.
* chmod 700 foldername will give read, write, and execute permissions for the user only.
* chmod 327 foldername will give write and execute (3) permission for the user, w (2) for the group, and read, write, and execute for the users.

As you can see, there are several options when it comes to permissions. You have the capability to dictate usability among users. While it may be easier to just give all permission to everyone, it may end up biting you in the end. So choose wisely.

# Permissions in Linux

* Difficulty Level : [Easy](https://www.geeksforgeeks.org/easy/)
* Last Updated : 01 Jun, 2022

 Read

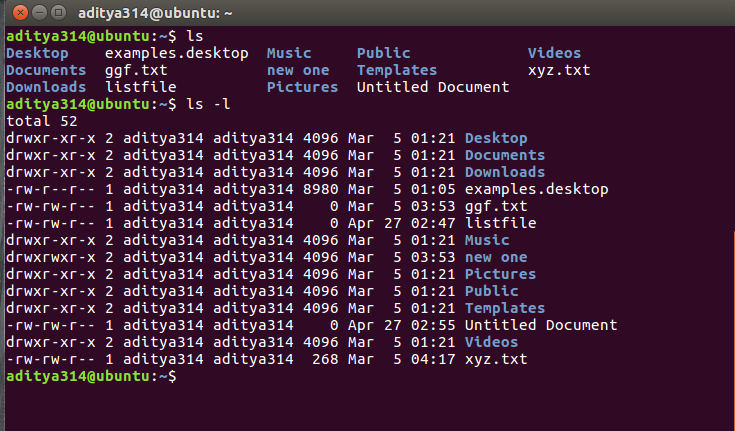
 Discuss

Linux is a multi-user operating system, so it has security to prevent people from accessing each other’s confidential files.

## Introduction

When you execute an “ls” command, you are not given any information about the security of the files, because by default “ls” only lists the names of files. You can get more information by using an “option” with the “ls” command. All options start with a ‘-‘. For example, to execute “ls” with the “long listing” option, you would type ls -l

When you do so, each file will be listed on a separate line in long format. There is an example in the window below. 



There’s a lot of information in those lines. 

1. The first character will almost always be either a ‘-‘, which means it’s a file, or a ‘d’, which means it’s a directory.
2. The next nine characters (rw-r–r–) show the security; we’ll talk about them later.
3. The next column shows the owner of the file. In this case it is me, my userID is “aditya314”.
4. The next column shows the group owner of the file. In my case I want to give the “aditya314” group of people special access to these files.
5. The next column shows the size of the file in bytes.
6. The next column shows the date and time the file was last modified.
7. And, of course, the final column gives the filename.

Deciphering the security characters will take a bit more work.

## Understanding the security permissions

First, you must think of those nine characters as three sets of three characters (see the box at the bottom). Each of the three “rwx” characters refers to a different operation you can perform on the file. 

--- --- ---

rwx rwx rwx

user group other

### Read, write, execute and –

The ‘r’ means you can “read” the file’s contents.   
The ‘w’ means you can “write”, or modify, the file’s contents.   
The ‘x’ means you can “execute” the file. This permission is given only if the file is a program.   
If any of the “rwx” characters is replaced by a ‘-‘, then that permission has been revoked.

### User, group and others

user – The user permissions apply only the owner of the file or directory, they will not impact the actions of other users.   
group – The group permissions apply only to the group that has been assigned to the file or directory, they will not effect the actions of other users.   
others – The others permissions apply to all other users on the system, this is the permission group that you want to watch the most.

### Reading the security permissions

For example, consider that the user’s permissions for some files is “rw-” as the first three characters. This means that the owner of the file (“aditya314”, i.e. me) can “read” it (look at its contents) and “write” it (modify its contents). I cannot execute it because it is not a program; it is a text file.

If “r-x” is the second set of 3 characters it means that the members of the group “aditya314” can only read and execute the files.

The final three characters show the permissions allowed to anyone who has a UserID on this Linux system. Let us say we have the permission (“r–“). This means anyone in our Linux world can read, but they cannot modify the contents of the files or execute it.

## Changing security permissions

The command you use to change the security permissions on files is called “chmod”, which stands for “change mode”, because the nine security characters are collectively called the security “mode” of the file.

1. The first argument you give to the “chmod” command is ‘u’, ‘g’, ‘o’. We use:   
   u for user   
   g for group   
   o for others,   
   you can also use a combination of them (u,g,o).   
   This specifies which of the three groups you want to modify.
2. After this use   
   a ‘+’ for adding   
   a ‘-‘ for removing   
   and a “=” for assigning a permission.
3. Then specify the permission r,w or x you want to change.   
   Here also you can use a combination of r,w,x.   
   This specifies which of the three permissions “rwx” you want to modify
4. use can use commas to modify more permissions
5. Finally, the name of the file whose permission you are changing

An example will make this clearer.   
For example, if you want to give “execute” permission to the world (“other”) for file “xyz.txt”, you would start by typing 

chmod o

Now you would type a ‘+’ to say that you are “adding” a permission. 

chmod o+

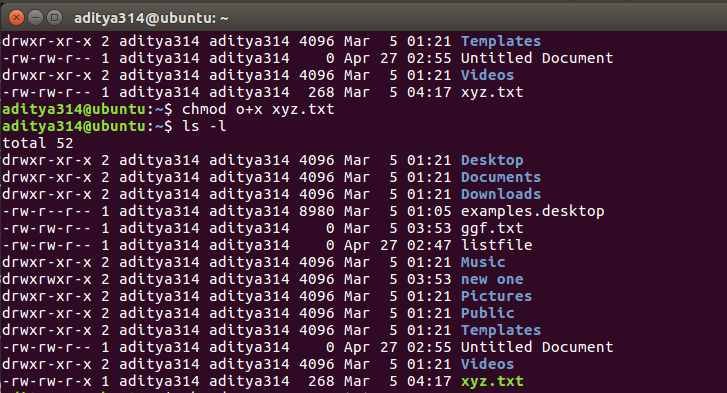
Then you would type an ‘x’ to say that you are adding “execute” permission. 

chmod o+x

Finally, specify which file you are changing. 

chmod o+x xyz.txt

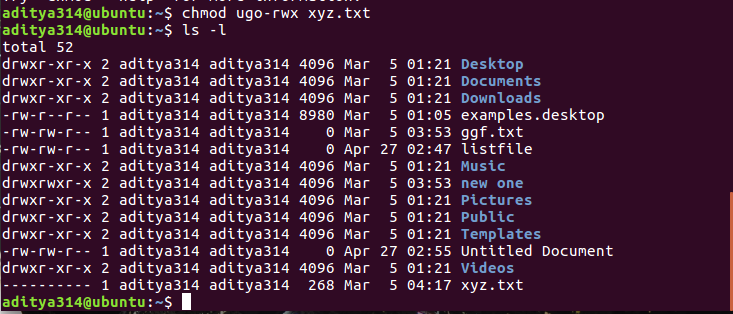
You can see the change in the picture below. 



You can also change multiple permissions at once. For example, if you want to take all permissions away from everyone, you would type 

chmod ugo-rwx xyz.txt

The code above revokes all the read(r), write(w) and execute(x) permission from all user(u), group(g) and others(o) for the file xyz.txt which results to this. 



Another example can be this: 

chmod ug+rw,o-x abc.mp4

The code above adds read(r) and write(w) permission to both user(u) and group(g) and revoke execute(x) permission from others(o) for the file abc.mp4.

Something like this: 

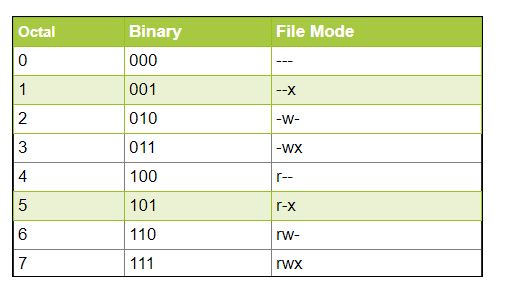
chmod ug=rx,o+r abc.c

assigns read(r) and execute(x) permission to both user(u) and group(g) and add read permission to others for the file abc.c.

There can be numerous combinations of file permissions you can invoke, revoke and assign. You can try some in your linux system.

## The octal notations

You can also use octal notations like this. 



Using the octal notations table instead of ‘r’, ‘w’ and ‘x’. Each digit octal notation can be used of either of the group ‘u’,’g’,’o’.

So, the following work the same. 

chmod ugo+rwx [file\_name]

chmod 777 [file\_name]

Both of them provides full read write and execute permission (code=7) to all the group.

Same is the case with this.. 

chmod u=r,g=wx,o=rx [file\_name]

chmod 435 [file\_name]

Both the codes give read (code=4) permission to user, write and execute (code=3) for group and read and execute (code=5) for others.

And even this… 

chmod 775 [file\_name]

chmod ug+rwx,o=rx [file\_name]

Both the commands give all permissions (code=7) to user and group, read and execute (code=5) for others.

## Further learning

The default Linux security model is a bit inflexible. To give special access (such as modification privileges) to a group of people, you have to get your system administrator to create a group with those people in it. Furthermore, if you would like to give a different set of access privileges (such as read access) to another group of people, you can’t do it because you can only assign one group owner per file or directory. To solve this problem, you can use ACLs (Access Control Lists). You can learn more about them from this link: [ACLs](https://en.wikipedia.org/wiki/Access_control_list)

References: [askubuntu](https://askubuntu.com/questions/518259/understanding-chmod-symbolic-notation-and-use-of-octal" \t "_blank)   
[linuxcommand.org](http://linuxcommand.org/)

This article is contributed by **Aditya Nihal Kumar Singh**. If you like GeeksforGeeks and would like to contribute, you can also write an article using [write.geeksforgeeks.org](http://write.geeksforgeeks.org/) or mail your article to review-team@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

# File Permissions in Linux / Unix: How to Read, Write & Change?

By[Mary Brent](https://www.guru99.com/author/mary)UpdatedJanuary 7, 2023

Linux is a clone of UNIX, the multi-user operating system which can be accessed by many users simultaneously. Linux can also be used in mainframes and servers without any modifications. But this raises security concerns as an unsolicited or malign user can corrupt, change or remove crucial data. For effective security, Linux divides authorization into 2 levels.

1. Ownership
2. Permission

In this Linux file commands tutorial, you will learn-

* [Linux File Ownership](https://www.guru99.com/file-permissions.html#linux_file_ownership)
* [Linux File Permissions](https://www.guru99.com/file-permissions.html#linux_file_permissions)
* [Changing file/directory permissions in Linux Using ‘chmod’ command](https://www.guru99.com/file-permissions.html#changing_file_permissions_in_linux_using_chmod_command)
* [Absolute(Numeric) Mode in Linux](https://www.guru99.com/file-permissions.html#absolute_mode_in_linux)
* [Symbolic Mode in Linux](https://www.guru99.com/file-permissions.html#symbolic_mode_in_linux)
* [Changing Ownership and Group in Linux](https://www.guru99.com/file-permissions.html#changing_ownership_and_group_in_linux)

The concept of Linux File **permission** and **ownership** is crucial in Linux. Here, we will explain Linux permissions and ownership and will discuss both of them. Let us start with the **Ownership.**

Click [here](https://www.guru99.com/faq#faq1) if the video is not accessible

## Linux File Ownership

Every file and directory on your Unix/Linux system is assigned 3 types of owner, given below.

### User

A user is the owner of the file. By default, the person who created a file becomes its owner. Hence, a user is also sometimes called an owner.

Logo

Description automatically generated with low confidenceEXPLORE MOREShape

Description automatically generated with low confidenceLearn Java Programming with Beginners Tutorial08:32A picture containing shape

Description automatically generatedLinux Tutorial for Beginners: Introduction to Linux Operating...01:35Graphical user interface, application

Description automatically generatedWhat is Integration Testing Software Testing Tutorial03:04A picture containing application

Description automatically generatedWhat is JVM (Java Virtual Machine) with Architecture JAVA...02:24How to write a TEST CASE Software Testing Tutorial01:08Graphical user interface, application

Description automatically generatedSeven Testing Principles Software Testing05:01Graphical user interface, application

Description automatically generatedLinux File Permissions Commands with Examples13:29Graphical user interface, application

Description automatically generatedHow to use Text tool in Photoshop CC Tutorial08:32A picture containing background pattern

Description automatically generatedWhat is NoSQL Database Tutorial02:00A picture containing shape

Description automatically generatedImportant Linux Commands for Beginners Linux Tutorial15:03

### Group

A user- group can contain multiple users. All users belonging to a group will have the same Linux group permissions access to the file. Suppose you have a project where a number of people require access to a file. Instead of manually assigning permissions to each user, you could add all users to a group, and assign group permission to file such that only this group members and no one else can read or modify the files.

### Other

Any other user who has access to a file. This person has neither created the file, nor he belongs to a usergroup who could own the file. Practically, it means everybody else. Hence, when you set the permission for others, it is also referred as set permissions for the world.

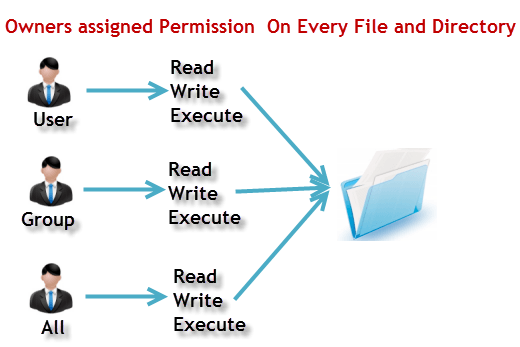
Now, the big question arises how does **Linux distinguish**between these three user types so that a user ‘A’ cannot affect a file which contains some other user ‘B’s’ vital information/data. It is like you do not want your colleague, who works on your [Linux computer](https://www.guru99.com/unix-linux-tutorial.html), to view your images. This is where **Permissions** set in, and they define **user behavior**.

Let us understand the **Permission system** on Linux.

## Linux File Permissions

Every file and directory in your UNIX/Linux system has following 3 permissions defined for all the 3 owners discussed above.

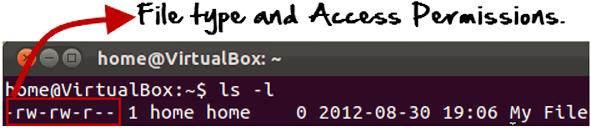
* **Read:** This permission give you the authority to open and read a file. Read permission on a directory gives you the ability to lists its content.
* **Write:**The write permission gives you the authority to modify the contents of a file. The write permission on a directory gives you the authority to add, remove and rename files stored in the directory. Consider a scenario where you have to write permission on file but do not have write permission on the directory where the file is stored. You will be able to modify the file contents. But you will not be able to rename, move or remove the file from the directory.
* **Execute:**In Windows, an executable program usually has an extension “.exe” and which you can easily run. In Unix/Linux, you cannot run a program unless the execute permission is set. If the execute permission is not set, you might still be able to see/modify the program code(provided read & write permissions are set), but not run it.

File Permissions in Linux/Unix

Let’s see file permissions in Linux with examples:

**ls – l** on terminal gives

ls - l



Here, we have highlighted **‘-rw-rw-r–‘**and this weird looking code is the one that tells us about the Unix permissions given to the owner, user group and the world.

Here, the first ‘**–**‘ implies that we have selected a file.p>



Else, if it were a directory, **d**would have been shown.

File Permissions in Linux/Unix

The characters are pretty easy to remember.

**r** = read permission  
**w** = write permission  
**x** = execute permission  
**–** = no permission

Let us look at it this way.

The first part of the code is **‘rw-‘**. This suggests that the owner ‘Home’ can:



* Read the file
* Write or edit the file
* He cannot execute the file since the execute bit is set to ‘-‘.

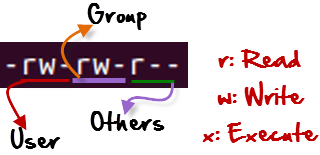
By design, many Linux distributions like Fedora, CentOS, Ubuntu, etc. will add users to a group of the same group name as the user name. Thus, a user ‘tom’ is added to a group named ‘tom’.

The second part is **‘rw-‘.** It for the user group ‘Home’ and group-members can:

* Read the file
* Write or edit the file

The third part is for the world which means any user. It says **‘r–‘.** This means the user can only:

* Read the file



## Changing file/directory permissions in Linux Using ‘chmod’ command

Say you do not want your colleague to see your personal images. This can be achieved by changing file permissions.

We can use the ‘**chmod’** command which stands for ‘change mode’. Using the command, we can set permissions (read, write, execute) on a file/directory for the owner, group and the world.

**Syntax:**

chmod permissions filename

There are 2 ways to use the command –

1. **Absolute mode**
2. **Symbolic mode**

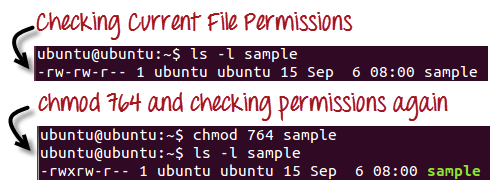
## Absolute(Numeric) Mode in Linux

In this mode, file **permissions are not represented as characters but a three-digit octal number**.

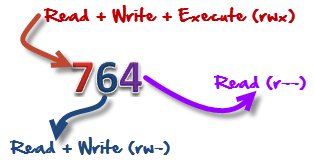
The table below gives numbers for all for permissions types.

| **Number** | **Permission Type** | **Symbol** |
| --- | --- | --- |
| 0 | No Permission | — |
| 1 | Execute | –x |
| 2 | Write | -w- |
| 3 | Execute + Write | -wx |
| 4 | Read | r– |
| 5 | Read + Execute | r-x |
| 6 | Read +Write | rw- |
| 7 | Read + Write +Execute | rwx |

Let’s see the chmod permissions command in action.



In the above-given terminal window, we have changed the permissions of the file ‘sample to ‘764’.



‘764’ absolute code says the following:

* Owner can read, write and execute
* Usergroup can read and write
* World can only read

**This is shown as ‘-rwxrw-r–**

This is how you can change user permissions in Linux on file by assigning an absolute number.

## Symbolic Mode in Linux

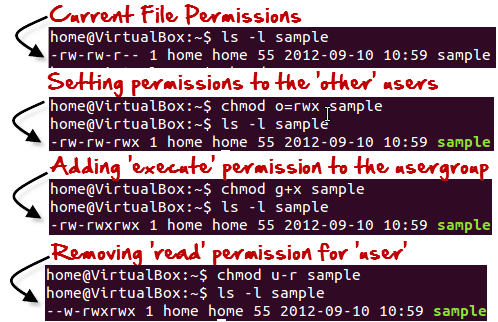
In the Absolute mode, you change permissions for all 3 owners. In the symbolic mode, you can modify permissions of a specific owner. It makes use of mathematical symbols to modify the Unix file permissions.

| **Operator** | **Description** |
| --- | --- |
| **+** | Adds a permission to a file or directory |
| **–** | Removes the permission |
| **=** | Sets the permission and overrides the permissions set earlier. |

The various owners are represented as –

| **User Denotations** | |
| --- | --- |
| u | user/owner |
| g | group |
| o | other |
| a | all |

We will not be using permissions in numbers like 755 but characters like rwx. Let’s look into an example



## Changing Ownership and Group in Linux

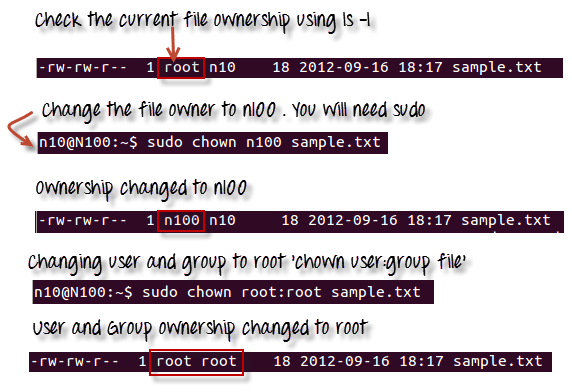
For changing the ownership of a file/directory, you can use the following command:

chown user filename

In case you want to change the user as well as group for a file or directory use the command

chown user:group filename

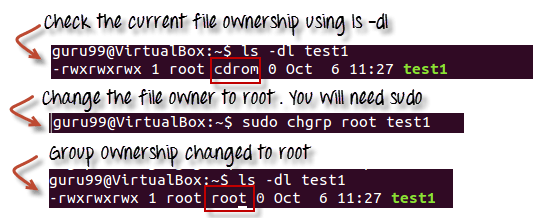
Let’s see this in action



In case you want to change group-owner only, use the command

chgrp group\_name filename

‘**chgrp’** stands for change group.

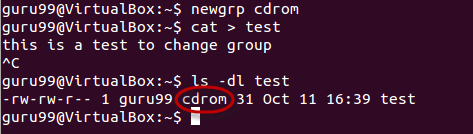


### Tip

* The file /etc/group contains all the groups defined in the system
* You can use the command “groups” to find all the groups you are a member of

File Permissions in Linux/Unix

* You can use the command newgrp to work as a member a group other than your default group



* You cannot have 2 groups owning the same file.
* You do not have nested groups in Linux. One group cannot be sub-group of other
* x- eXecuting a directory means Being allowed to “enter” a dir and gain possible access to sub-dirs
* There are other permissions that you can set on Files and Directories which will be covered in a later advanced tutorial

## Summary:

* Linux being a multi-user system uses permissions and ownership for security.
* There are three user types on a Linux system viz. User, Group and Other
* Linux divides the file permissions into read, write and execute denoted by r,w, and x
* The permissions on a file can be changed by ‘chmod’ command which can be further divided into Absolute and Symbolic mode
* The ‘chown’ command can change the ownership of a file/directory. Use the following commands: chown user file or chown user:group file
* The ‘chgrp’ command can change the group ownership **chrgrp group filename**
* What does x – eXecuting a directory mean? A: Being allowed to “enter” a dir and gain possible access to sub-dirs.

[Home](https://www.informit.com/) > [Articles](https://www.informit.com/articles/index.aspx)

# [Linux Command-Line for Beginners](https://www.informit.com/articles/article.aspx?p=2928189)

* Sep 5, 2018

[📄 Contents](https://www.informit.com/articles/article.aspx?p=2928189&seqNum=7&gclid=Cj0KCQiAic6eBhCoARIsANlox847BKUg-fKgErYPsluCSkz8o_PqNANW8GKrpPmPSySvw9UNUwMvIa4aAu7EEALw_wcB)

[␡](https://www.informit.com/articles/article.aspx?p=2928189&seqNum=7&gclid=Cj0KCQiAic6eBhCoARIsANlox847BKUg-fKgErYPsluCSkz8o_PqNANW8GKrpPmPSySvw9UNUwMvIa4aAu7EEALw_wcB)

1. [What Is the Command Line?](https://www.informit.com/articles/article.aspx?p=2928189)
2. [Accessing the Command Line](https://www.informit.com/articles/article.aspx?p=2928189&seqNum=2)
3. [User Accounts](https://www.informit.com/articles/article.aspx?p=2928189&seqNum=3)
4. [Reading Documentation](https://www.informit.com/articles/article.aspx?p=2928189&seqNum=4)
5. [Understanding the Linux File System Hierarchy](https://www.informit.com/articles/article.aspx?p=2928189&seqNum=5)
6. [Navigating the Linux File System](https://www.informit.com/articles/article.aspx?p=2928189&seqNum=6)
7. Working with Permissions
8. [Working with Files](https://www.informit.com/articles/article.aspx?p=2928189&seqNum=8)
9. [Working as Root](https://www.informit.com/articles/article.aspx?p=2928189&seqNum=9)
10. [Commonly Used Commands and Programs](https://www.informit.com/articles/article.aspx?p=2928189&seqNum=10)
11. [References](https://www.informit.com/articles/article.aspx?p=2928189&seqNum=11)

* [⎙ Print](https://www.informit.com/articles/printerfriendly/2928189)
* [+ Share This](http://www.addthis.com/bookmark.php)

[< Back](https://www.informit.com/articles/article.aspx?p=2928189&seqNum=6) **Page 7** of 11 [Next >](https://www.informit.com/articles/article.aspx?p=2928189&seqNum=8)

## Working with Permissions

Under Linux (and UNIX), everything in the file system, including directories and devices, is a file. And every file on your system has an accompanying set of permissions based on ownership. These permissions provide data security by giving specific permission settings to every single item denoting who may read, write, or execute the file. These permissions are set individually for the file’s owner, for members of the group the file belongs to, and for all others on the system.

You can examine the default permissions for a file you create by using the umask command, which lists default permissions using the number system explained next, or by using the touch command and then the ls command’s long-format listing, like this:

matthew@seymour:~$ **touch file**

matthew@seymour:~$ **ls -l file**

-rw-r--r-- 1 matthew matthew 0 2015-06-30 13:06 file

In this example, the touch command quickly creates a file. The ls command then reports on the file, displaying the following (from left to right):

* **The type of file created**—Common indicators of the type of file are in the leading letter in the output. A blank (which is represented by a dash, as in the preceding example) designates a plain file, d designates a directory, c designates a character device (such as /dev/ttyS0), and b is used for a block device (such as /dev/sda).
* **Permissions**—Read, write, and execute permissions may be assigned for the owner, group, and all others on the system. (You learn more about these permissions later in this section.)
* **Number of links to the file**—The number 1 designates that there is only one file, and any other number indicates that there might be one or more hard-linked files. Links are created with the ln command. A hard-linked file is an exact copy of the file, but it might be located elsewhere on the system. Symbolic links of directories can also be created, but only the root operator can create a hard link of a directory.
* **The owner**—This is the account that owns the file; it is originally the file creator, but you can change this designation by using the chown command.
* **The group**—This is the group of users allowed to access the file; it is originally the file creator’s main group, but you can change this designation by using the chgrp command.
* **File size and creation/modification date**—The last two elements indicate the size of the file in bytes and the date the file was created or last modified.

### Assigning Permissions

Under Linux, permissions are grouped by owner, group, and others, with read, write, and execute permission assigned to each, as follows:

Owner Group Others

rwx rwx rxw

Permissions can be indicated by mnemonic or octal characters. Mnemonic characters are listed here:

* r indicates permission for an owner, a member of the owner’s group, or others to open and read the file.
* w indicates permission for an owner, a member of the owner’s group, or others to open and write to the file.
* x indicates permission for an owner, a member of the owner’s group, or others to execute the file (or read a directory).

In the previous example for the file named file, the owner, matthew, has read and write permission. Any member of the group named matthew may only read the file. All other users may only read the file. Also note that default permissions for files created by the root operator (while using sudo or a root account) will differ because of umask settings assigned by the shell.

Many users prefer to use numeric codes, based on octal (base 8) values, to represent permissions. Here’s what these values mean:

* 4 indicates read permission
* 2 indicates write permission
* 1 indicates execute permission

In octal notation, the previous example file has a permission setting of 644 (read + write or 4 + 2, read-only or 4, read-only or 4). Although you can use either form of permissions notation, octal is easy to use quickly when you visualize and understand how permissions are numbered.

**NOTE**

In Linux, you can create groups to assign a number of users access to common directories and files, based on permissions. You might assign everyone in accounting to a group named accounting and allow that group access to accounts payable files while disallowing access by other departments. Defined groups are maintained by the root operator, but you can use the newgrp command to temporarily join other groups to access files (as long as the root operator has added you to the other groups). You can also allow or deny other groups’ access to your files by modifying the group permissions of your files.

### Directory Permissions

Directories are also files under Linux. For example, again use the ls command to show permissions, like this:

matthew@seymour:~$ **mkdir directory**

matthew@seymour:~$ **ls -ld directory**

drwxr-xr-x 2 matthew matthew 4096 2015-06-30 13:23 directory

In this example, the mkdir command is used to create a directory. The ls command, with its -ld option, is used to show the permissions and other information about the directory (not its contents). Here you can see that the directory has permission values of 755 (read + write + execute or 4 + 2 + 1, read + execute or 4 + 1, and read + execute or 4 + 1).

This shows that the owner can read and write to the directory and, because of execute permission, also list the directory’s contents. Group members and all other users can list only the directory contents. Note that directories require execute permission for anyone to be able to view their contents.

You should also notice that the ls command’s output shows a leading d in the permissions field. This letter specifies that this file is a directory; normal files have a blank field in its place. Other files, such as those specifying a block or character device, have a different letter.

For example, if you examine the device file for a Linux serial port, you see the following:

matthew@seymour:~$ **ls -l /dev/ttyS0**

crw-rw---- 1 root dialout 4, 64 2015-06-30 08:13 /dev/ttyS0

Here, /dev/ttyS0 is a character device (such as a serial communications port and designated by a c) owned by root and available to anyone in the dialout group. The device has permissions of 660 (read + write, read + write, no permission).

On the other hand, if you examine the device file for an IDE hard drive, you see this:

matthew@seymour:~$ **ls -l /dev/sda**

brw-rw-- -- 1 root disk 8, 0 2015-06-30 08:13 /dev/sda

In this example, b designates a block device (a device that transfers and caches data in blocks) with similar permissions. Other device entries you will run across on your Linux system include symbolic links, designated by s.

### Altering File Permissions with chmod

You can use the chmod command to alter a file’s permissions. This command uses various forms of command syntax, including octal or a mnemonic form (such as u, g, o, or a and rwx, and so on) to specify a desired change. You can use the chmod command to add, remove, or modify file or directory permissions to protect, hide, or open up access to a file by other users (except for the root account or a user with super user permission and using sudo, either of which can access any file or directory on a Linux system).

The mnemonic forms of chmod’s options are (when used with a plus character, +, to add, or a minus sign, -, to remove):

* **u**—Adds or removes user (owner) read, write, or execute permission
* **g**—Adds or removes group read, write, or execute permission
* **o**—Adds or removes read, write, or execute permission for others not in a file’s group
* **a**—Adds or removes read, write, or execute permission for all users
* **r**—Adds or removes read permission
* **w**—Adds or removes write permission
* **x**—Adds or removes execution permission

For example, if you create a file, such as a readme.txt, the file has the following default permissions (set by the umask setting in /etc/bashrc, covered in the next section):

-rw-r--r-- 1 matthew matthew 0 2015-06-30 13:33 readme.txt

As you can see, you can read and write the file. Anyone else can only read the file (and only if it is outside your home directory, which will have read, write, and execute permission set only for you, the owner). You can remove all write permission for anyone by using chmod, the minus sign (-), and aw, as follows:

matthew@seymour:~$ **chmod a-w readme.txt**

matthew@seymour:~$ **ls -l readme.txt**

-r--r--r-- 1 matthew matthew 0 2015-06-30 13:33 readme.txt

Now, no one can write to the file (except you, if the file is in your /home or /tmp directory because of directory permissions). To restore read and write permission for only you as the owner, use the plus sign (+) and the u and rw options, like so:

matthew@seymour:~$ **chmod u+rw readme.txt**

matthew@seymour:~$ **ls -l readme.txt**

-rw-r--r-- 1 matthew matthew 0 2015-06-30 13:33 readme.txt

You can also use the octal form of the chmod command (for example, to modify a file’s permissions so that only you, the owner, can read and write a file). Use the chmod command and a file permission of 600, like this:

matthew@seymour:~$ **chmod 600 readme.txt**

matthew@seymour:~$ **ls -l readme.txt**

-rw------- 1 matthew matthew 0 2015-06-30 13:33 readme.txt

If you take away execution permission for a directory, files might be hidden inside and may not be listed or accessed by anyone else (except the root operator, of course, who has access to any file on your system). By using various combinations of permission settings, you can quickly and easily set up a more secure environment, even as a normal user in your /home directory.

### File Permissions with umask

When you create a file, it is created with a default set of permissions. You can view and modify the default permissions for files with umask, which works like a filter. When a file is created by a user account, whether that account is owned by a human like matthew or a process like init, the file will be created using specific permissions.

The numbers we used above when discussing file permissions are also used with umask, but with an interesting change. Now, the numbers defined in umask are subtracted from the ultimate file permissions. So, if you wanted all new files to be created with a default permission of 777, you would type this:

matthew@seymour:~$ **umask 000**

Of course, you would never want to have all your files accessible by default because that would be incredibly insecure and unsafe. The default umask is 022, which means that files are created by default with 755 permissions, except in your /home directory where the umask is 002 and files are created with 775.

To find the current umask setting, use this:

matthew@seymour:~$ **umask**

This may list four digits instead of three. If so, don’t be confused. The additional digit is the first one; it is explained later in this chapter, in the section “Understanding Set User ID, Set Group ID, and Sticky Bit Permissions.”

To change the umask setting—for example, if you wanted the default to be 740—use the following:

matthew@seymour:~$ **umask 037**

### File Permissions with chgrp

You can use the chgrp command to change the group to which a file belongs:

matthew@seymour:~$ **chgrp wheel filename**

### Changing File Permissions with chown

You can use the chown command to change the owner of a file:

matthew@seymour:~$ **chown matthew filename**

You can also use the chown command to change the group of a file at the same time:

matthew@seymour:~$ **chown matthew:wheel filename**

### Understanding Set User ID, Set Group ID, and Sticky Bit Permissions

The first two of the three listed types of permission are “set user ID,” known as suid, and “set group ID,” or sgid. These settings, when used in a program, enable any user running that program to have program owner or group owner permissions for that program. These settings enable the program to be run effectively by anyone, without requiring that each user’s permissions be altered to include specific permissions for that program.

One commonly used program with suid permissions is the passwd command:

matthew@seymour:~$ **ls -l /usr/bin/passwd**

-rwsr-xr-x 1 root root 42856 2015-01-26 10:09 /usr/bin/passwd

This setting allows normal users to execute the command (as root) to make changes to a root-only-accessible file /etc/passwd.

By default, suid and sgid are turned off on files. To set them, add an extra digit to the beginning of a number in a chmod command. Suid uses 4. Sgid uses 2. You can set both at the same time by using 6 (4 + 2). For example, for a file owned by root with current 711 permissions allowing anyone to run it, you can make it run as root with the following:

matthew@seymour:~$ **chmod 4711 filename**

**NOTE**

Other files that might have suid or guid permissions include at, rcp, rlogin, rsh, chage, chsh, ssh, crontab, sudo, sendmail, ping, mount, and several UNIX-to-UNIX Copy (UUCP) utilities. Many programs (such as games) might also have this type of permission to access a sound device.

Files or programs that have suid or guid permissions can sometimes present security holes because they bypass normal permissions. This problem is compounded if the permission extends to an executable binary (a command) with an inherent security flaw because it could lead to any system user or intruder gaining root access. In past exploits, this typically happened when a user fed a vulnerable command with unexpected input (such as a long pathname or option); the command would fail, and the user would be presented with a root prompt. Although Linux developers are constantly on the lookout for poor programming practices, new exploits are found all the time and can crop up unexpectedly, especially in newer software packages that haven’t had the benefit of peer developer review.

Savvy Linux system administrators keep the number of suid or guid files present on a system to a minimum. The find command can be used to display all such files on a system:

matthew@seymour:~$ **sudo find / -type f -perm /6000 -exec ls -l {} \;**

**NOTE**

The find command is quite helpful and can be used for many purposes, such as before or during backup operations.

Note that the programs do not necessarily have to be removed from your system. If your users really do not need to use the program, you can remove a program’s execute permission for anyone. As the root operator, you have to decide whether your users are allowed, for example, to mount and unmount CD-ROMs or other media on your system. Although Linux-based operating systems can be set up to accommodate ease of use and convenience, allowing programs such as mount to be suid might not be the best security policy. Other candidates for suid permission change could include the chsh, at, or chage commands.

An additional setting called the sticky bit is available using this same additional first digit. A sticky bit limits who may rename or delete files within a directory. When it is set, files in that directory may be unlinked or renamed only by a super user, the directory owner, or the file owner. Set the sticky bit to on by using a 1, like this for a directory with 755 permissions:

matthew@seymour:~$ **chmod 1755 directoryname**

You can set the sticky bit concurrently with suid and sgid, like this (4 + 2 + 1):

matthew@seymour:~$ **chmod 7755 directoryname**

### Setting Permissions with Access Control Lists

POSIX is a family of standards created to maintain stability and consistency across operating systems for UNIX and UNIX-like systems, such as Linux. One important feature of POSIX is the access control list (ACL; often pronounced “AK-el”). ACLs permit even more fine-grained control over access permissions.

By default, all files have an ACL. To view the ACL for a file, use this:

matthew@seymour:~$ **getfacl filename**

Typical getfacl output includes multiple lines, like this for filename.txt:

# file: filename.txt

# owner: matthew

# group: matthew

user::rw-

group::rw-

other::r--

The information listed here is standard and clear, based on what you already know. The real power of ACLs is that you can add to them. You are not restricted to the standard set of user, group, other. You can add multiple users and groups with permissions specific to each.

To add the user sandra with read, write, and execute permissions to the ACL for a file named secrets.txt, use the following:

matthew@seymour:~$ **setfacl -m u:sandra:rwx secrets.txt**

To remove and reset sandra’s permissions on the file to the file’s defaults, use the following:

matthew@seymour:~$ **setfacl -r u:sandra: secrets.txt**

From these two examples, you can see that -m is for modify and -r is for remove.

ACLs permit similar actions with groups and others as with a user. Instead of the u: before the name, use a g: for groups and an o: for others, like this:

matthew@seymour:~$ **setfacl -m g:groupname:rwx secrets.txt**

matthew@seymour:~$ **setfacl -m o:r secrets.txt**

Notice that with others, there is no username or group name to include in the commands.

A useful feature is masking, which allows you to list only the permissions that are available, as in this example:

matthew@seymour:~$ **setfacl -m m:rx secrets.txt**

This limits everyone, regardless of any other settings. So, in this case, a group may have rwx settings on the file, but the mask here says to only permit rx, so rx will be the only settings that are available.

As an exercise, see if you can figure out the meaning of this output from getfacl for a file named coffeecup.conf:

# file: coffeecup.conf

# owner: matthew

# group: yirgacheffe

user::rw-

group::rw-

other::r--

group:qa:rwx

group:uat:rwx

mask::rwx

* repo -> repository
* clone -> bring a repo down from the internet (remote repository like Github) to your local machine
* add -> track your files and changes with Git
* commit -> save your changes into Git
* push -> push your changes to your remote repo on Github (or another website)
* pull -> pull changes down from the remote repo to your local machine
* status -> check to see which files are being tracked or need to be commited
* init -> use this command inside of your project to turn it into a Git repository and start using Git with that codebase

# Classic SysAdmin: Understanding Linux File Permissions

###### **THE LINUX FOUNDATION | 06 JANUARY 2022**

This is a classic article written by [*Jack Wallen*](http://www.twitter.com/jackofalltech1) from the [*Linux.com*](http://linux.com/) archives. For more great SysAdmin tips and techniques check out our free [*intro to Linux course*](https://www.edx.org/course/introduction-to-linux?utm_medium=partner-marketing&utm_source=affiliate&utm_campaign=linuxfoundation&utm_content=blog-lfs101).

Although there are already a lot of good security features built into Linux-based systems, one very important potential vulnerability can exist when local access is granted – – that is file permission-based issues resulting from a user not assigning the correct permissions to files and directories. So based upon the need for proper permissions, I will go over the ways to assign permissions and show you some examples where modification may be necessary.

## ****Permission Groups****

Each file and directory has three user based permission groups:

* **owner** – The Owner permissions apply only to the owner of the file or directory, they will not impact the actions of other users.
* **group** – The Group permissions apply only to the group that has been assigned to the file or directory, they will not affect the actions of other users.
* **all users** – The All Users permissions apply to all other users on the system, this is the permission group that you want to watch the most.

## Permission Types

Each file or directory has three basic permission types:

* **read** – The Read permission refers to a user’s capability to read the contents of the file.
* **write** – The Write permissions refer to a user’s capability to write or modify a file or directory.
* **execute** – The Execute permission affects a user’s capability to execute a file or view the contents of a directory.

## Viewing the Permissions

You can view the permissions by checking the file or directory permissions in your favorite GUI File Manager (which I will not cover here) or by reviewing the output of the ***“ls -l”*** command while in the terminal and while working in the directory which contains the file or folder.

The permission in the command line is displayed as: ***\_rwxrwxrwx 1 owner:group***

1. User rights/Permissions
   1. The first character that I marked with an underscore is the special permission flag that can vary.
   2. The following set of three characters (rwx) is for the owner permissions.
   3. The second set of three characters (rwx) is for the Group permissions.
   4. The third set of three characters (rwx) is for the All Users permissions.
2. Following that grouping since the integer/number displays the number of hardlinks to the file.
3. The last piece is the Owner and Group assignment formatted as Owner:Group.

## Modifying the Permissions

When in the command line, the permissions are edited by using the command **chmod**. You can assign the permissions explicitly or by using a binary reference as described below.

## Explicitly Defining Permissions

To explicitly define permissions you will need to reference the Permission Group and Permission Types.

The Permission Groups used are:

* **u**– Owner
* **g** – Group
* **o** – Others
* **a** – All users

The potential Assignment Operators are + (plus) and – (minus); these are used to tell the system whether to add or remove the specific permissions.

The Permission Types that are used are:

* **r** – Read
* **w** – Write
* **x** – Execute

So for example, let’s say I have a file named file1 that currently has the permissions set to \_rw\_rw\_rw, which means that the owner, group, and all users have read and write permission. Now we want to remove the read and write permissions from the all users group.

To make this modification you would invoke the command: **chmod a-rw file1**  
To add the permissions above you would invoke the command: ***chmod a+rw file1***

As you can see, if you want to grant those permissions you would change the minus character to a plus to add those permissions.

## Using Binary References to Set permissions

Now that you understand the permissions groups and types this one should feel natural. To set the permission using binary references you must first understand that the input is done by entering three integers/numbers.

A sample permission string would be **chmod 640 file1**, which means that the owner has read and write permissions, the group has read permissions, and all other user have no rights to the file.

The first number represents the Owner permission; the second represents the Group permissions; and the last number represents the permissions for all other users. The numbers are a binary representation of the rwx string.

* ***r*** = 4
* ***w*** = 2
* ***x*** = 1

You add the numbers to get the integer/number representing the permissions you wish to set. You will need to include the binary permissions for each of the three permission groups.

So to set a file to permissions on file1 to read **\_rwxr\_\_\_\_\_**, you would enter ***chmod 740 file1***.

## ****Owners and Groups****

I have made several references to Owners and Groups above, but have not yet told you how to assign or change the Owner and Group assigned to a file or directory.

You use the chown command to change owner and group assignments, the syntax is simple

**chown owner:group filename**,

so to change the owner of file1 to user1 and the group to family you would enter **chown user1:family file1**.

## ****Advanced Permissions****

The special permissions flag can be marked with any of the following:

* **\_** – no special permissions
* ***d*** – directory
* ***l***– The file or directory is a symbolic link
* ***s*** – This indicated the setuid/setgid permissions. This is not set displayed in the special permission part of the permissions display, but is represented as a **s** in the read portion of the owner or group permissions.
* ***t*** – This indicates the sticky bit permissions. This is not set displayed in the special permission part of the permissions display, but is represented as a **t** in the executable portion of the all users permissions

## ****Setuid/Setgid Special Permissions****

The setuid/setguid permissions are used to tell the system to run an executable as the owner with the owner’s permissions.

Be careful using setuid/setgid bits in permissions. If you incorrectly assign permissions to a file owned by root with the setuid/setgid bit set, then you can open your system to intrusion.

You can only assign the setuid/setgid bit by explicitly defining permissions. The character for the setuid/setguid bit is **s**.

So do set the setuid/setguid bit on file2.sh you would issue the command **chmod g+s file2.sh**.

## Sticky Bit Special Permissions

The sticky bit can be very useful in shared environment because when it has been assigned to the permissions on a directory it sets it so only file owner can rename or delete the said file.

You can only assign the sticky bit by explicitly defining permissions. The character for the sticky bit is **t**.

To set the sticky bit on a directory named dir1 you would issue the command **chmod +t dir1**.

## When Permissions Are Important

To some users of Mac- or Windows-based computers, you don’t think about permissions, but those environments don’t focus so aggressively on user-based rights on files unless you are in a corporate environment. But now you are running a Linux-based system and permission-based security is simplified and can be easily used to restrict access as you please.

So I will show you some documents and folders that you want to focus on and show you how the optimal permissions should be set.

* ***home directories***– The users’ home directories are important because you do not want other users to be able to view and modify the files in another user’s documents of desktop. To remedy this you will want the directory to have the **drwx\_\_\_\_\_\_ (700)** permissions, so lets say we want to enforce the correct permissions on the user user1’s home directory that can be done by issuing the command **chmod 700 /home/user1**.
* ***bootloader configuration files***– If you decide to implement password to boot specific operating systems then you will want to remove read and write permissions from the configuration file from all users but root. To do you can change the permissions of the file to 700.
* ***system and daemon configuration files***– It is very important to restrict rights to system and daemon configuration files to restrict users from editing the contents, it may not be advisable to restrict read permissions, but restricting write permissions is a must. In these cases it may be best to modify the rights to 644.
* ***firewall scripts*** – It may not always be necessary to block all users from reading the firewall file, but it is advisable to restrict the users from writing to the file. In this case the firewall script is run by the root user automatically on boot, so all other users need no rights, so you can assign the 700 permissions.

Other examples can be given, but this article is already very lengthy, so if you want to share other examples of needed restrictions please do so in the comments