

# **GIT INSTRUCTION MANUAL**

EVAN BERGERON  
SUNNY GAKHAR  
NISHAD GOTHOSKAR  
FREDERICK LEE  
ZIYANG WANG

## **CONTENTS**

Introduction . . . . .	2
What is Git . . . . .	2
Motivation . . . . .	2
Installing Git . . . . .	3
Windows . . . . .	3
Mac . . . . .	3
Linux . . . . .	3
Initializing Repository . . . . .	4
Adding files to version control . . . . .	5
Committing changes . . . . .	5
Reverting changes . . . . .	5
Branching . . . . .	5
Merging Branches . . . . .	5
Ignoring Files . . . . .	6
Rebasing . . . . .	7
Cherrypicking . . . . .	8
Collaboration with Github . . . . .	8
General workflows . . . . .	8
Easily Reordering Commits . . . . .	8
Adding partial files . . . . .	8
Git aliases . . . . .	8
Vim workflows . . . . .	9
Installing vim-fugitive . . . . .	9
Easy git blame . . . . .	9
Emacs workflows . . . . .	9
Installing Magit . . . . .	9

## INTRODUCTION

This is a manual on how to set up Git on your computer and set up a basic Git workflow. The document covers installing and setting up Git and how to work with Git. The majority of this document is primarily intended for users who want to learn Git to use it in a fast-paced setting such as a hackathon, this document is also useful as a reference for experienced Git users who want to refer to some specific concept or command which they need.

## WHAT IS GIT

Git is a version control system designed to be used for working on small and large projects.

## MOTIVATION

When you were in school and you had short homework assignments, you would just start them and finish them in a short span of time (not longer than a week). But when you move on to designing and working on bigger projects, there are multiple issues that come into play. Say you are participating in a hackathon and have finalized your idea and distribution of work among the teammates. How do you actually work on the project together. Having all of them work on one computer is not optimal. You might have each teammate work on his own piece independently, but how do you merge everyone's work? Moreover, what if two or more teammates work on the same file, but do different modifications unknown to the others? And what if someone wants to explore a different direction to work on, while keeping the original work intact? Enter Git.

## INSTALLING GIT

## WINDOWS

If you have Windows, one easy way of installing Git is from this website:

`https://git-scm.com/download/win`

Once you have downloaded the installation file, you can run it and proceed through the installation steps.

## MAC

Similar to Windows, one way of installing Git on is from this link

`https://git-scm.com/download/mac`

## LINUX

If you're working on Linux, you can install Git using a basic package management tool that comes with your distribution.

For example, for ubuntu, you have

`apt-get install git`

or for Arch Linux, you have

`pacman -S git`

Here

## INITIALIZING REPOSITORY

To make sure you have Git set up, type `git` into your console and the following should show up.

```
usage: git [--version] [--help] [-C <path>] [-c name=value]
       [--exec-path[=<path>]] [--html-path] [--man-path] [--info-path]
       [-p | --paginate | --no-pager] [--no-replace-objects] [--bare]
       [--git-dir=<path>] [--work-tree=<path>] [--namespace=<name>]
       <command> [<args>]
```

...

'`git help -a`' and '`git help -g`' list available subcommands and some concept guides. See '`git help <command>`' or '`git help <concept>`' to read about a specific subcommand or concept.

To start a new repository, type `git init` into the console. It should output the following

```
Initialized empty Git repository in <path to current directory>/.
```

ADDING FILES TO VERSION CONTROL

COMMITTING CHANGES

REVERTING CHANGES

BRANCHING

MERGING BRANCHES

## IGNORING FILES

The best way to ignore files in a repository is to create a `.gitignore` file at the base of your repository with the names of all files to ignore.

Often you will find that in your projects, you will have files in your repository that do not need to be tracked. These include automatically generated files, larger libraries that are really external dependencies, or system specific files such as the `.DS_Store` file in Macs.

For example, let us have a repository with two files: `code.py` and `generated.txt`. Here we only want to keep track of changes to `code.py` while ignoring changes to `generated.txt` which is automatically generated every time `code.py` is run.

Without a `.gitignore` file, running `git status` would get the following output:

```
$ git status
On branch master
Untracked files:
  (use "git add <file>..." to include in what will be committed)

    code.py
    generated.txt
```

```
nothing added to commit but untracked files present
(use "git add" to track)
```

We create a `.gitignore` file with the following content:

```
generated.txt
.gitignore
```

Running `git status` again, we see that the generated file and the `.gitignore` itself are ignored by Git as desired.

```
$ git status
On branch master
Untracked files:
  (use "git add <file>..." to include in what will be committed)

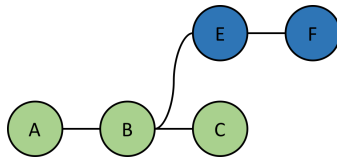
    code.py
```

```
nothing added to commit but untracked files present
(use "git add" to track)
```

## REBASING

The rebasing and merging are the two primary ways of combining changes from separate branches with Git. One primary advantage of rebasing is that the Git history is made more linear which makes it easier to track changes over time.

For example, say that you have been working on a separate branch called **feature** (shown in blue) on which you have two commits. At the same time, a friend of yours has committed change “C” to master (shown in green).

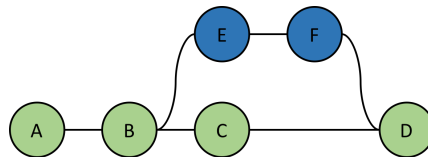


A regular merge command such as

```
$ git checkout master
```

```
$ git merge feature
```

would result in the following Git history:



However, if you want a more linear history, you would ideally want your commits “E” and “F” to be stacked on top of your friend’s commits. To do as such, simply run as follows:

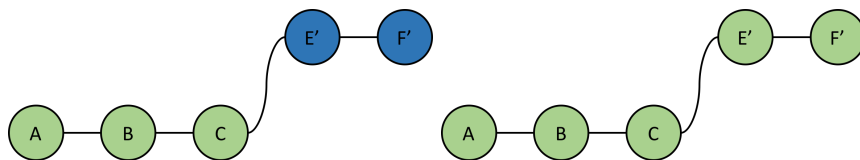
```
$ git checkout feature
```

```
$ git rebase master
```

```
$ git checkout master
```

```
$ git merge feature
```

The initial rebase command changes the root of your **feature** branch to your friend’s last commit as shown below on the left image.



The final merge of your feature onto master then forwards master’s history to match your feature branch as shown above on the right. As you can see, the history follows a much more linear pattern compared to the branching pattern of a regular merge alone.

## CHERRY-PICKING

### COLLABORATION WITH GITHUB

#### GENERAL WORKFLOWS

##### EASILY REORDERING COMMITS

We've all been there. You pull from master, take several minutes to clean up the various merge conflicts, and then are ready to push. You pull one last time before pushing, and what do you know - someone has pushed again in the last couple minutes.

Previously, my workflow for this situation would look something like

```
$ git checkout -b tmp
$ git checkout master
$ git reset --hard HEAD~
$ git cherry-pick tmp
```

This works fine, but there's a much easier way – one that involves very little typing. We can simply say

```
git rebase -i HEAD 2
```

This will open an interactive git rebasing session (the `-i` stands for interactive). The window will display something along the lines of

```
pick 370e221 Commit one
pick c342396 Commit two
```

In whichever text editor we're in, we may simply reorder these lines to reorder the commits. Much shorter!

##### ADDING PARTIAL FILES

I just used this, actually. Suppose you've changed a single file `foo.c` in different sections, and each of these changes are logically different. For instance, maybe you refactor some function `foo`, while at the same time fixing a bug in function `bar`. Rather than create a separate branch and manually edit the files, we can simply say

```
git add -p foo.c
```

This will bring up an interactive prompt. It will automatically cycle through all the different areas of the diff, asking you if you want to stage each section. You may hit `y` or `n` for yes or no.

Once you're done adding the subset of changes you want to commit, you can double-check you have the right changes staged by saying

```
git diff --cached
```

Once you're sure that you're good to go, just commit your changes as normal. You can repeat this process for the remaining changes. (Or just do a normal `git add` at this point).

##### GIT ALIASES

Git aliases are a good way to save yourself a lot of typing. I frequently want to see the git commit history, but don't especially care about the body of each commit.



Heres the command Ive added to my configuration:

```
git config --global alias.l "log --oneline"
```

I can then just type `git l` to see a one line log of this commit history. If there are several long commands you use frequently, this can be a great way to save yourself some time.

## VIM WORKFLOWS

At the time of writing, perhaps the most feature complete vim-git plugin is Tim Popes vim-fugitive. Consequently, we will assume usage of this plugin throughout the entire vim workflows tutorial.

### INSTALLING VIM-FUGITIVE

There are a number of ways to install vim-fugitive. The one suggested by Tim Pope is as follows:

```
$ cd ~/.vim/bundle
$ git clone git://github.com/tpope/vim-fugitive.git
$ vim -u NONE -c "helptags vim-fugitive/doc" -c q
```

Vundle is a great plugin manager for vim – if you use this, you may simply add the line

```
Plugin 'tpope/vim-fugitive'
```

to your vimrc and run the `PluginInstall` command.

### EASY GIT BLAME

Youre browsing some file and discover a horrible bug written by one of your coworkers. Youre about to storm over to someones desk and verbally abuse them for producing incorrect code. Before you deliver your diatribe, you need to know who to blame.

Before using vim-fugitive, you would have to exit vim, manually type `git blame <filename>`, and then search for the relevant line in the output. Now, you can simply type `:Gblame` in your vim prompt, and a vertical split will open up right next to the line in question. You could even establish a keybinding to do this for you! What was once several lines of typing is now a single keystroke away! Your coworkers have never been so scared...

## EMACS WORKFLOWS

Similarly to vim-fugitive for vim, Magit is (at the time of writing), the most feature-complete git wrapper for emacs. We will thus assume usage of this package.

### INSTALLING MAGIT