* To find the version of Git installed

$git --version

* To create global configuration in Git

$git config --global user.name “Root User”

$git config --global user.email “[ismailshaik1989@gmail.com](mailto:ismailshaik1989@gmail.com)”

Global configuration is useful to identify yourself to Git repository or other repositories when you checking files or making any changes to repository.

To view your global configuration in your system

$git config --list

* To initialize a git repository

$ git init

* To check the git status or our project status

$ git status

* To add files to staging area

$ git add <filename>

* To commit changes to the repository

$ git commit –m “Added file <filename>”

* You can also add same type of files at a time by using wildcards

$ git add ‘\*.txt’

* To view all the changes, we made (log)

$ git log

* To add remote repository to your git repository

$ git remote add <short name> <URL of remote repository>

* Cloning an existing repository

$ git clone <https://github.com/libgit2/libgit2>

That creates a directory named “libgit2”, initializes a .git directory inside it, pulls down all the data for that repository, and checks out a working copy of the latest version.

* If you want to clone the repository into a directory named something other than “libgit2”, you can specify that as the next command-line option

$ git clone <https://github.com/libgit2/libgit2> mylibgit

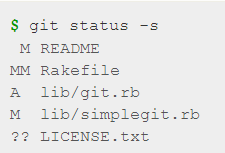
* Checking the status of your files

$ git status

If you would like to see the short status about changes

$ git status –s

$ git status –short



New files that aren’t tracked have a ?? next to them, new files that have been added to the staging area have an A, modified files have an M and so on. README file is modified in the working directory but not yet staged, while the lib/simplegit.rb file is modified and staged. The Rakefile was modified, staged and then modified again.

* To add files to the git

$ git add README

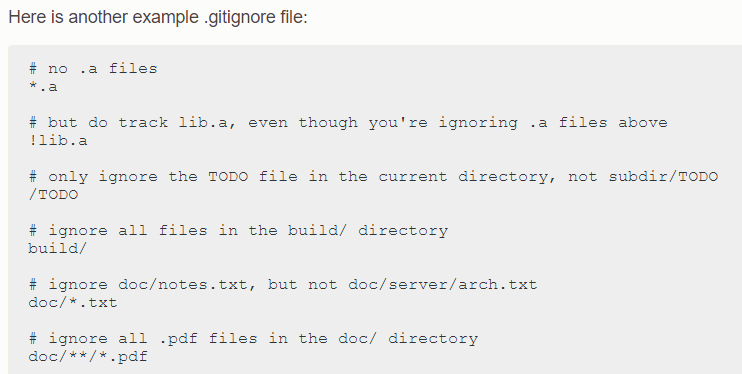
* Ignore files: Often, you’ll have a class of files that you don’t want git to automatically add or even show you as being untracked. In such cases, you can create a file listing to match them named .gitignore. Here is an example of .ignore file.

$ cat .ignore

\*.[oa]

\*~

The first line tells Git to ignore files ending in ”.o” or “.a” the second line tells Git to ignore all files whose names end with a tilde, which is used by many text editors. You can also include log, tmp, or pid directory.



* Viewing your staged and unstaged changes

If the git status command is too vague for you and you want to know exactly what you changed, not just which files were changed, you can use the git diff command

$ git diff

* If you want to see what you’ve staged that will go into your next commit, you can use git diff –staged. This command compares your staged changes to your last commit

$ git diff --staged

$ git diff --cached to see what you’ve staged so far (--staged and --cached are synonyms).

* It’s important to note that git diff by itself doesn’t show all changes made since your last commit – only changes that are still unstaged. This can be confusing, because if you’ve staged all of your changes, git diff will give you no output.
* Skipping the staging area: If you want to skip the staging area, Git provides a simple shortcut. Adding the –a option to the git commit command makes git automatically stage every file that is already tracked before doing the commit, letting you skip the git add part

$ git commit –a –m “added new benchmarks”

* To remove a file from Git, you have to remove it from your tracked files (more accurately, remove it from your staging area) and then commit. The git rm command does that, also removes the file from your working directory so you don’t see it as as untracked file the next time around.

$ git rm PROJECTS.md

$ git status

* If you want to keep the file in your working tree but remove it from your staging area. In other words, you may want to keep the file on your hard drive but not have Git track it anymore. This is particularly useful if you forgot to add something to your .gitignore file and accidentally staged it, like a large log file or a bunch of .a compiled files. To do this use the --cached option

$ git rm --cached README

* If you want to rename a file in Git, you can use

$ git mv file\_from file\_to

$ git status

* To see the commit history of Git

$ git log

It has so many options available to show exactly what you’re looking for.

One of the most helpful option is –p which shows the difference introduced in each commit. You can use -2, which limits the output to only the last two entries

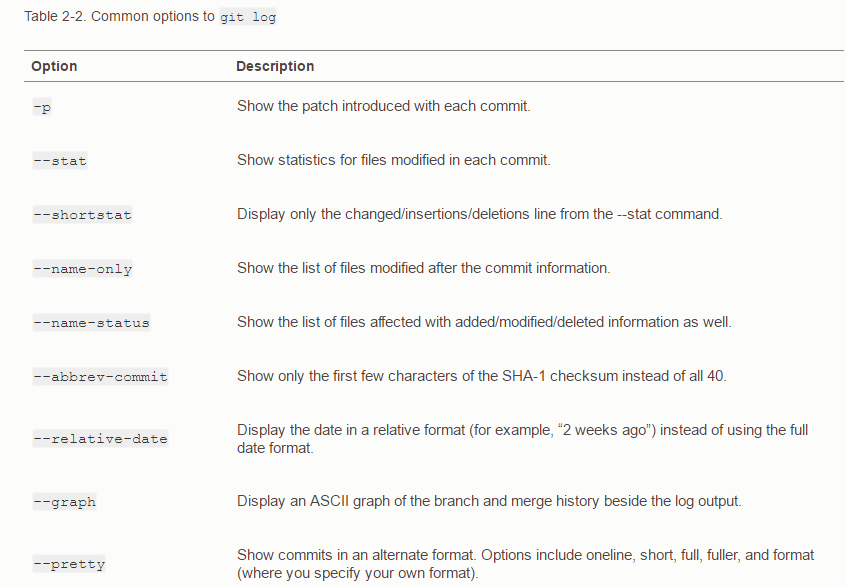
$ git log –p -2

You can also use a series of summarizing options with git log. For example, if you want to see some abbreviated stats for each commit, you can use the --stat option

$ git log --stat

Another really useful option is --pretty. This option changes the log output to formats other than the default. The online option prints each commit on a single line, which is useful if you’re looking at a lot of commits. In addition, the short, full, and fuller option show the output in roughly same format but with less or more info

$ git log --pretty=online



* To list the commits made in the last two weeks

$ git log --since=2.weeks

* One of the common undo takes place when you commit too early and possibly forget to add some files, or you mess up your commit messages. If you want to try that commit again, you can run commit with the --amend option

$ git commit --amend

* If we have two files and want to commit them as two separate changes, but you accidentally type git add \* and stage them both. To unstage one of the file

$ git reset HEAD <filename>

* To see which remote servers you have configured, you can run the git remote command. It lists the short names of each remote handle you’ve specified.

$ git remote

You can also specify –v option, which shows you the URLs that git has stored for the short names to be used when reading and writing to that remote

$ git remote –v

* Adding remote repositories: We learned how the clone command implicitly adds the origin remote for you. Here’s how to add a new remote explicitly. To add a new remote Git repository as a short name you can reference easily, run

$ git remote add <shortname> <URL>

* To fetch data from the remote repository

$ git fetch <remote-name>

Ex: git fetch pb

Now pb’s master branch is now accessible as pb/master - you can merge it into one of your branches, or you can check out a local branch at that point if you want to inspect it.

If you clone a repository, the command automatically adds that repository under the name “origin”. So, **git fetch origin** fetches any new work that has been pushed ot that server since you cloned it. It’s important to note that the **git fetch** command downloads the data to your local repository – It doesn’t automatically merge it with any of your work modify what you’re currently working on. You have to merge it manually into your work when you’re ready.

* When you have your project at a point that you want to share, you have to push it upstream. If you want to push your master branch to your origin server, then you can run this to push any commits you’ve done back up to the server

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

* If you want to see more information about a particular remote, you can use git remote show [remote-name] command.

$ git remote show [remote-name]

* You can run git remote rename to change remote’s shortname. For instance, if you want to rename pb to paul, you can do so with git remote rename

$ git remote rename pb paul

* If you want to remove a remote for some reason

$ git remote rm paul

* Listing the available tags in Git is straightforward. Just type git tag

$ git tag

You can also search for tags with a particular pattern.

$ git tag –l “v1.8.5\*”

* Git uses two main types of tags: lightweight and annotated

A lightweight tag is very much like a branch that doesn’t change. Annotated tags, are stored as full objects in the git database.

Creating an annotated tag in Git, use –a when you run the tag command

$ git tag –a v1.4 –m “my version 1.4”

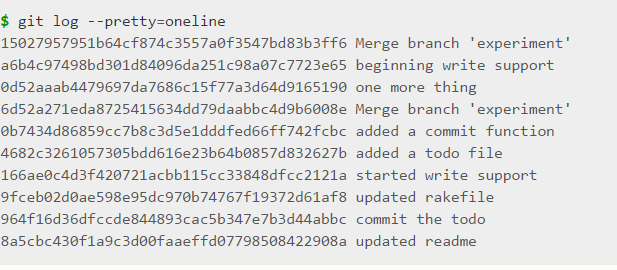
You can see the tag data along with the commit that was tagged by using the git show command

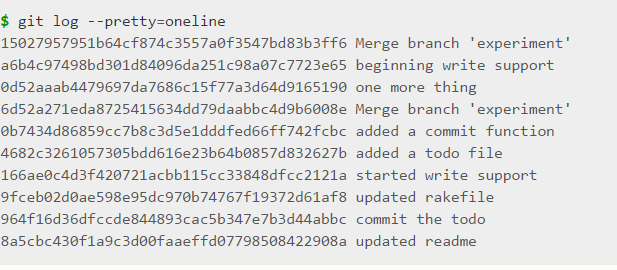
$ git show v1.4

* Another way to gat commits is with a lightweight tag. This is basically the commit checksum stored in a file. No other information is kept. To create a lightweight tag, don’t supply the –a, -s, or –m option:

$ git tag v1.4.5

* You can also tag commits after you’ve moved past them.





Now, suppose you forgot to tag the project at v1.2, which was the “updated rackfile” commit. You can add it after the fact. To tag that commit, you specify the commit checksum (or part of it) at the end of the command

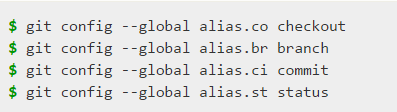
$ git tag –a v1.2 9fceb02

* Sharing tags: By default, the **git push** command doesn’t transfer tags to remote servers. You will have to explicitly push tags to a shared server after you have created them. This process is just like sharing remote branches – you can run

$ git push origin [tagname]

If you have a lot of tags that you want to push up at once, you can use the --tags option to the git push command. This will transfer all of your tags to the remote server that are not already there.

* If you don’t want to type the entire text of each of the Git commands, you can easily set up an alias for each command using **git config**



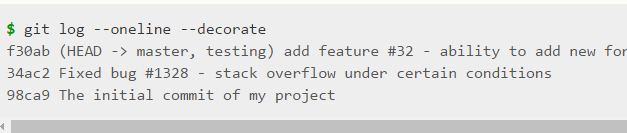
This means that, for example, instead of typing **git commit**, you need to type **git ci**.

* Git doesn’t store data as a series of changesets ro differences, but instead as a series of snapshots.
* To create a branch in Git

$ git branch <branch name>

Ex: git branch testing

The git branch command only creates a new branch – It didn’t switch to that branch. You can easily see this by running a simple git log command that shows you where the branch pointers are pointing. This option is called --decorate



* Switching branches: To switch to an existing branch, run the **git checkout**  command. Let’s switch to the new testing branch

$ git checkout testing

To create a branch and switch it to it at the same time, you can run the **git checkout**  command with **–b**  option

$ git checkout –b <branchname>

* To delete the branch

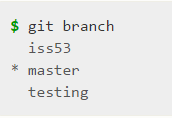
$ git branch –d <branchname>

* To merge your branch to the master

$ git checkout master

$ git merge <branchname>

* The **git branch**  command does more than just create and delete branches. If you run it with no arguments, you get a simple listing of your current branches.

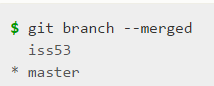


Notice the \* character that prefixes the master branch: it indicates branch that you currently have checked out (i.e., the branch that HEAD points to).

To see last commit on each branch, you can run

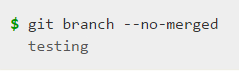
$ git branch –v

* The useful --merged and --no-merged options can filter this list to branches that you have or have not yet merged into the branch you’re currently on. To see which branches are already merged into the branch you’re on, you can run git branch –mreged:



Because you already merged in iss53 earlier, you see it in your list. Branches on this list without the \* in front of them are generally fine to delete with **git branch –d** ; you’ve already incorporated their work into another branch, so you’re not going to lose anything.

To see all branches that contain work you haven’t yet merged in, you can run **git branch --no-merged**



* If you do some work on your local master branch, and , in the meantime, someone else pushes to git.ourcompany.com and updates its master branch, then your histories move forward differently. Also, as long as you stay out of contact with your origin server, your origin/master pointer doesn’t move. To synchronize your work, you run a **git fetch origin** command. This command looks up which server “origin” is (in this case, it’s git.ourcompany.com), fetches any data from it that you don’t yet have, and updates your local database, moving your origin/master pointer to its new, more up-to-date position.
* If you have a branch named serverfix that you want to work on with others. You can push it up the same way you pushed your first branch. Run  **git push <remote> <branch>**.
* If you committed some files on one branch and replay them on another one

$ git checkout experiment

$ git rebase master

At this point, you can go back to the master branch and do a fast-forward merge

$ git checkout master

$ git merge experiment