Git Basic Commands

**Your Identity**

The first thing you should do when you install Git is to set your user name and email address. This is important because every Git commit uses this information, and it’s immutably baked into the commits you start creating:

1. git config --global user.name "VAIBHAV GUPTA"
2. git config --global user.email [guptavaibhav.05086@gmail.com](mailto:guptavaibhav.05086@gmail.com)
3. git config --list

### Initializing a Repository in an Existing Directory

If you have a project directory that is currently not under version control and you want to start controlling it with Git, you first need to go to that project’s directory. If you’ve never done this, it looks a little different depending on which system you’re running:

cd /c/user/my\_project

git init

This creates a new subdirectory named .git that contains all of your necessary repository files — a Git repository skeleton. At this point, nothing in your project is tracked yetIf you want to start version-controlling existing files (as opposed to an empty directory), you should probably begin tracking those files and do an initial commit. You can accomplish that with a few git add commands that specify the files you want to track, followed by a git commit:

cd "C:\Users\Praveen\Desktop\HTMLTutorial\GitDemoTriaining"

$ git add \*.c

$ git add LICENSE

$ git commit -m 'initial project version'

### Checking the Status of Your Files

The main tool you use to determine which files are in which state is the git status command. If you run this command directly after a clone, you should see something like this:

Let’s say you add a new file to your project, a simple README file. If the file didn’t exist before, and you run git status, you see your untracked file like so:

git status

You can see that your new README file is untracked, because it’s under the “Untracked files” heading in your status output. Untracked basically means that Git sees a file you didn’t have in the previous snapshot (commit); Git won’t start including it in your commit snapshots until you explicitly tell it to do so. It does this so you don’t accidentally begin including generated binary files or other files that you did not mean to include

### Tracking New Files

In order to begin tracking a new file, you use the command git add. To begin tracking the README file, you can run this:

git add README

git add --all

If you run your status command again, you can see that your README file is now tracked and staged to be committed:

### Staging Modified Files

Let’s change a file that was already tracked. If you change a previously tracked file called CONTRIBUTING.md and then run your git status command again, you get something that looks like this:

The CONTRIBUTING.md file appears under a section named “Changes not staged for commit” — which means that a file that is tracked has been modified in the working directory but not yet staged. To stage it, you run the git add command. git add is a multipurpose command — you use it to begin tracking new files, to stage files

### Short Status

While the git status output is pretty comprehensive, it’s also quite wordy. Git also has a short status flag so you can see your changes in a more compact way. If you run git status -s or git status --short you get a far more simplified output from the command:

### Viewing Your Staged and Unstaged Changes

If the git status command is too vague for you — you want to know exactly what you changed, not just which files were changed — you can use the git diff command. We’ll cover git diff in more detail later, but you’ll probably use it most often to answer these two questions: What have you changed but not yet staged? And what have you staged that you are about to commit? Although git status answers those questions very generally by listing the file names, git diff shows you the exact lines added and removed — the patch, as it were.

To see what you’ve changed but not yet staged, type git diff with no other arguments:

### Committing Your Changes

Now that your staging area is set up the way you want it, you can commit your changes. Remember that anything that is still unstaged — any files you have created or modified that you haven’t run git add on since you edited them — won’t go into this commit. They will stay as modified files on your disk. In this case, let’s say that the last time you ran git status, you saw that everything was staged, so you’re ready to commit your changes. The simplest way to commit is to type git commit:

$ git commit

### Skipping the Staging Area

Although it can be amazingly useful for crafting commits exactly how you want them, the staging area is sometimes a bit more complex than you need in your workflow. 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'

Notice how you don’t have to run git add on the CONTRIBUTING.md file in this case before you commit. That’s because the -a flag includes all changed files. This is convenient, but be careful; sometimes this flag will cause you to include unwanted changes.