**Learning Git**

Well, I think I know Git now. Haha, now, that's the overstatement of the year. Git is so damn vast, that I'd be surprised if even Linus Torvalds knows everything about it.

What I do know about Git is enough for me to set it up and use it fairly effectively. I think I've a more intuitive understanding of Git now and most importantly, I'm not afraid of Git anymore.

Which is why I thought why not blog about it? Like, ooh I have got hard earned knowledge about it. Haha, turns out it's the ticket punching stuff, the price of entry into the club. The club of professional-level programmers. And there are plenty of resources on the web to get started off with Git.

Having said that though, Git is a very non-intuitive, scary beast for people who've never dealt with source-control systems. The previous sentence does seem a little untruthful now, because it's not scary for me anymore. So that's the key, the moment you understood the key concepts of Git and have played with and broken a few things in Git, your understanding improves tremendously.

Since there are a lot of free resources on the internet, I'm not going to reinvent the wheel and type in explanations of all those Git concepts. What I intend to do here is create a short cheat sheet of what the most important concepts in Git are and point to the best explanations (in my view) of those concepts.

So let's start.

Also, you can use this blog to come up with a list of starter questions to help you get started with Git. Like just Google the questions in bold which follow. I'm sure I'd have missed other good questions, so please let me know in the comments about those.

**What is Git?**

I think listening to this dude is really enlightening: [Git Video-What is git?](http://git-scm.com/video/what-is-git" \t "_blank)

**Installing and Setting up Git**

To help you get start:

Git Download page: [Git-Download](http://git-scm.com/downloads" \t "_blank)

Once you've installed Git, you can choose from any of the free UI tools which layer on top of the Git application that you installed in the previous step.

Git UIs: [Git Guis Download](http://git-scm.com/downloads/guis" \t "_blank)

You can use any of the UIs provided there. I use SourceTree now, while earlier I was using Git Extensions.

Be advised, Git by itself only provides a command line interface for control. As a beginner, I found this to be a little overwhelming and found the UI interface much easier to use.

Software set-ups are oftentimes pesky. And that is best solved by Googling around.

**What is a Git repo?**

The concept of a Git repo is central to how Git is designed. You can think of a repo or repository as that set of files which comprise a unique project. Though very simply, a Git repo is just a folder with the .git folder in it.

A repo is the means by which you organize your files and the contents thereof-- mainly computer code within Git.

These are some of the links I like:

[What's a Git repo-more details](http://www.sbf5.com/~cduan/technical/git/git-1.shtml)

[How to get started with GitHub--touches upon git repos](http://readwrite.com/2013/09/30/understanding-github-a-journey-for-beginners-part-1)

So if your read the first link above, you'll see that my explanation and the explanation in the link differ. Mainly, '.git' folder is the actual Git repo, while the folder in which Git repo is located is the **local working directory**.

**What is the local working directory?**

[Difference between local repo and working directory](http://stackoverflow.com/questions/21692155/whats-the-difference-between-working-directory-and-local-repository)

So if you read the first answer in the above SO question, you'll see that the working directory is the directory or folder in which the '.git' folder which is the actual local repo resides.

How git works is by tracking the differences between the files on your system in the working directory and the local repo. And keeps track of remote changes by comparing the local git repo or .git folder with the .git folder of a remote repo.

[Mainly diff between SVN and Git terms](http://stackoverflow.com/questions/5284449/where-is-the-git-working-directory)

[A simple guide to Git](http://rogerdudler.github.io/git-guide/)

**What is the local Git repo and remote Git repo?**

Alright, let's put some background to what I mean by local vs remote.

Simply, **local** means on your computer, the computer you're probably reading this on. And it is probably also the computer where you'd be making changes to your Git managed files.

Your local system or computer is where your local Git repo would be.

**Remote: so what does remote mean?** **And why do we need a remote?**

So Remote means any computer other than yours. A repo which resides on a central server or even someone else's personal computer would be a remote repo to you.

**Why do you need a remote version of your repo?**

Simply put you **don't need**a remote repo. Say, if you're working on a single person project and just want to manage your code or files locally, you can just have a local Git set-up.

Now, usually coding is done in teams and people use other people's code all the time. So you need an effective means of collaboration and having a Git remote repo offers the same.

For example if you use GitHub, when you create a GitHub account what you're doing is setting up some space on GitHub's servers to create remote versions of your local Git repos. And then this code becomes available to the whole wide world and anyone on GitHub can collaborate with you on your projects and you can do the same.

**Alright, now how do you use Git?**

***Most basic use case.***

So the most basic use case when you're using Git is:

1. Clone (**What is cloning a repo?**) the remote Git repo of the project you'll be contributing to onto your local system. OR create a new Git repo on your local system.

2. Add the files you want to manage to the newly created Git repo on your system, if it's a brand new empty repo. If it's a cloned repo then it'd have some files added to it by someone else.

3. Do the initial commit. This will make the first entry to the local .git folder or the local Git repo. Again,this is assuming it's a brand new repo you have created.

4. Make changes to your files in the local working directory.

5. Then pick the changes you want to commit and **stage** them.

6. **Commit** the staged changes.

7. If you have a remote repo added to your Git project, **push** your local commits to the remote repo.

**Making changes to files in the local Git repo:**

**What does Stage or staging mean?**

[the-staging-area](http://gitready.com/beginner/2009/01/18/the-staging-area.html)

[What does stage mean?](http://programmers.stackexchange.com/questions/119782/what-does-stage-mean-in-git)

[Getting-Started-Git-Basics](http://git-scm.com/book/en/v2/Getting-Started-Git-Basics)

**What is a Commit?**

[Git commit Vs Git push](http://stackoverflow.com/questions/2745076/what-are-the-differences-between-git-commit-and-git-push)

The first answer in the above question has one of the best graphics for the Git versioning process.

[Saving-changes/Git-commit](https://www.atlassian.com/git/tutorials/saving-changes/git-commit)

**What is a Push?**

[Pushing-and-pulling](http://gitready.com/beginner/2009/01/21/pushing-and-pulling.html)

The above link along with the first link for Commit should clarify the basics of Push and Commit.

**What is a Pull Request**(Optional, though standard practise on many platforms now)?

*Would need to understand branching before you can understand this well.*

Pull Request eh? It's basically a quality control and code education step. It's usually created so that other developers who're working on the same project can take a look at your code before you merge your stuff into the common branch off which all developers work.

This process is called code review.

**Git Branching: Most hard to wrap your head around and most useful.**

**What is branching in Git?**

[Branching](http://gitref.org/branching/)

[Gitmagic-branching](http://www-cs-students.stanford.edu/~blynn/gitmagic/ch04.html)

[Git-branching-2](http://www.sbf5.com/~cduan/technical/git/git-2.shtml)

[How do Git branches work?](http://superuser.com/questions/80379/how-do-git-branches-work-can-i-remove-the-master-branch)

[Atlassian-using Git branches](https://www.atlassian.com/git/tutorials/using-branches)

The reason why I've included these many links to this topic is because this is an inherently very non intuitive concept and this is where I think the power and flexibility of Git lies. Of course it's distributed and all. But branching is the real magic.

Just **checkout** on another branch and you enter a new world, which doesn't give a damn to the world it came from. All the files in the new branch can be as different from each other as you want them to be and they are all in the same folder. Trippy eh?

You can also think of branching as creating multiple universes within a single multiverse. Where your repo/project/local working directory forms the multiverse and the branches form the universes.

So when you create a repo or project in Git you get a default branch called Master. Or rather when you create a multiverse, it starts off with a single universe. You can just work off the single multiverse or as necessary whenever you want to, you can create new multiverses off the default multiverse. You can branch off the default multiverse and then branch off other multiverses and keep branching as necessary.

*And*you can merge from one branch to another at any time. You can transfer matter and energy across the multiverses you create. That is where **Merge** comes in.

If you keep branching and creating multiverses indiscriminately, then you'll end up with a mess which probably you'd need a PhD in Git to solve.

That's the reason why you have an optimum number of branches and different branching strategies. And this leads us into Git flow:

Before that:

**What is Checkout?**

Checkout is the operation which allows you to shift from one branch to another. From one multiverse to another. The magic command which allows you to shape shift from one form of the project to another.

Shifting between branches by Checking out a branch.

**What is Git Merge?**

[Git Merging](http://www.sbf5.com/~cduan/technical/git/git-3.shtml)

[Git-merge](https://www.atlassian.com/git/tutorials/using-branches/git-merge)

[Merging Vs Rebasing](https://www.atlassian.com/git/tutorials/merging-vs-rebasing/)

*Follows Branching.*

Basically you can pick some particular commit you want to on a branch and merge it into any other branch.

**What is GitFlow?**

There is no single definitive Git flow. By Git Flow I mean a branching model or strategy which is more commonly adopted by a team of developers working on a project. And the link I'm posting to is more or less the branching model or Git workflow that is adopted by most teams.

[Gitflow-workflow](https://www.atlassian.com/git/tutorials/comparing-workflows/gitflow-workflow)

The rest of the page the above link links to is pretty informative too.

**What is GitHub?**

[What is Github?](http://www.howtogeek.com/180167/htg-explains-what-is-github-and-what-do-geeks-use-it-for/)

Well, GitHub can be thought of as a way for collaborating with any developer in the world. All you need is a GitHub account. Once, you create a GitHub account you can put up your projects on GitHub and ask people to review it and etc. Similarly you can look into other peoples projects and review the code, use the code or suggest changes to the code or even create a new branch for yourself and make it yours, this process is called Forking in GitHub. Once you have forked a project to your account, you can make whatever changes you like to it. If you want the original developer to incorporate those changes into his project which is the main version of the project, just create a Pull request and if the original guy is happy with the suggested changes, he'll accept the Pull request and Merge the changes into the main project.

It can also be thought of as free code backup for your code. If you are a single developer with no closed source stuff then this works.

**Closing thoughts:**

I think these are the most important and commonly used features of Git. There are a lot of things which can be done in many ways that I'm not aware of in Git land.

If you're new to Git, then it's better that you pick up a good UI for your local Git such as SourceTree or GitHub's local UI or any other such. I prefer SourceTree now, because of the ease with which I can manage branching.

So, the idea is that you use a UI to first get upto speed with Git operations. And most of the times you'd be only using the UI, even after you become familiar with Git operations. Later, you'll be able to confidently use the Git bash to carry out Git operations much like giving instructions on the Windows command prompt or PowerShell.

Apart from two occasions, I've never used the Git command line interface. Most of the usual work and some more can be done through a good Git UI. Though of course you do need to understand what all those fancy buttons are doing for your under the hood.

The best part about Git is that it's very hard to lose data or changes in code, which is very helpful in teams as team members are liable to fuck up each others' code once in a while through an improper use or improper ordering of Git operations.

*UPDATE:*

*I've recently started working on Linux exclusively and since most GIT UIs for Linux suck, I've been using the linux terminal for using Git. Not bad after you get used to it. This cheat sheet has been useful:*[Git cheatsheet](https://training.github.com/kit/downloads/github-git-cheat-sheet.pdf" \t "_blank)

**Extra: Some other useful Stack Overflow questions:**

Undoing a commit:

[Undoing the last commit](http://stackoverflow.com/questions/927358/how-to-undo-the-last-commit)

Keeping your feature branch or ancillary branches up to date:

[Merging feature branch into master-best practice](http://stackoverflow.com/questions/5601931/best-and-safest-way-to-merge-a-git-branch-into-master)

Fixing merge conflicts:

[Fixing merge conflicts.](http://stackoverflow.com/questions/161813/fix-merge-conflicts-in-git)