**Git and Github:**

1. **ls**

List

1. **ls –a**

List all including hidden

1. **git init**

Initialize the .git file that contains the history of the project

1. **git add <filename>**

To add a file on stage

1. **git add** (dot)

To add all files in the present working directory on stage

1. **git reset <filename>** or **git rm - -cached <filename>**

To remove a file from stage

1. **git commit –m “<message>”**

To commit the changes in the repository

1. **git status**

To check the status of file that whether it is on stage or not or whether the changes that you’ve done are being putted on stage or not

1. **git log**

To see all the commits that were made in the pwd(present working directory) with the time zones

1. **touch <filename with extension>**

To create a file with given extension

1. **vi <filename.txt>**

To open the text file in vim editor

1. **:wq**

This is done when inside the vim editor press **Esc** then type :wq (write quit). This is done to write the changes inside the file on the disc and it writes everytime on the disc even if the changes are made or not. Modification time will be changed everytime.

1. **:x**

It just exits the buffer. It only writes on the disc if changes are made and do not write if change are not made. Modification time will be untouched if the file is not changed.

1. **cat <filename.txt>**

To show the content of text file

1. **git rm –rf <filename>**

rm is short for “remove” and –r is short for recursive to delete the subdirectories and content inside too. Whereas, -f is short for force to forcefully delete the files.

1. **git stash**

When you want on stage files to like “Go back to the back stage” And when I want I will make you bring back on stage. But I don’t want to commit and make history. So this is where it is used

1. **git stash pop**

All the files who were at the back stage, come on stage.

1. **git stash clear**

When you want all the files on the back stage to go away and you don’t want any of them to come on stage.

1. **git remote add origin <link>**

“git” is basically a git command. “remote” is basically that you are working with remote files. “add” means that you are gonna add the current file. “origin” means that the local repo where you’re gonna fetch files will be named origin. And the “link” is the link you get from github when you create a new repository. “Repository” is nothing but a file containing the history of your project.

1. **git push origin master**

Used to push the origin into the master branch of the repository.

1. **git branch <name of branch>**

First one is the main branch. When you work try to work on another branch not on the main branch as your code is not finalized yet. So to create a new branch you write this command.

1. **git checkout <name of branch you wanna move to>**

Lets say you have two branches “main” and “feature” and you are currently inside the feature branch and your commits are going directly on the feature branch. And you wanna move to the main branch you’ll simply write “**git checkout main**”. This will make the head to move to the next branch

1. **git merge <name of branch you wanna merge>**

Lets say your code is finalized and now ready to move to the main branch. To do this you simply merge the main branch with your current branch. So here in this example, if you wanna merge the feature branch with main branch you simply write “**git merge feature**”

1. **git clone <url to clone>**

This is done to clone a url to your present working directory in order to make desired changes on the project.

**Forking:**

Creating a new copy of existing repository of source code. You cannot directly make changes to the main branch of any other person’s repository. In order to make these changes you need to fork the project so you can work on the main branch of another person’s repository.

1. **git remote add upstream <Link of the upstream>**

“upstream” is basically the link from where you fork the repository

1. **git remote –v**

This command is used to show the list of remote repositories that are currently associated with a Git repository. The "**-v**" option shows the URL of each remote repository next to its name, which can be helpful when working with multiple remote repositories.

**Notes:**

* For every new feature you should create a new branch. This is because imagine if one feature has 1000 LOC(Lines of code) and there are 10 such features. Imagine how difficult it would be to manage such files. So it is recommended to create a new branch for a new feature.
* One more thing, one branch can only have one pull request. So if you have made a pull request to the upstream and then you make the commits then that commits will be shown or represented in the same pull request. This thing shows the importance of creating new branches for new features.
* Also, do not make changes in the main branch. Create a new branch for features that you want to add later.

**Pull Request**

* Now in order to make a pull request, first you have to fork the repository from the upstream and then you create a new branch and add a new feature and after adding that new feature a button pops on the screen to **make a pull request**. This allows you to request the upstream to accept the changes you made. Then the discussion session will consider your code and the upstream owner can now merge the code.
* Imagine you have two commits in the single branch and you want the previous commit to be deleted so you have to do “**git log**” and then copy the **commit id**, and then you have to write the command “**git reset <commit id>**” then “**git add .**” this will add the file on the stage and then “**git stash**” this will send it on the back stage area and then “**git add .**”. Now when you push this then you have to force push it, because the commits folder in the repository has more files then the present working directory. So in order to force push it you have to write the following command:
* **git push origin <branch name> -f**

**Fetch Upstream**

**IMPORTANT:** The purpose is to balance the main branch of the upstream and the main branch of your fork. So you have to do the “**git checkout main**” first and then do the fetching.

* Now imagine someone has made a pull request and the upstream owner has merged it with the main file. Now imagine some other people have the fork of the main file too. Now as the main file has now been changed, all the forks from that main file needs to get changed and in order to change them. Let’s say a third person has made a pull request and the owner of the upstream has accepted and merged that request to the main file. Now you are the person who has forked that main file before. Your fork needs to get updated so in order to update your fork you need to **fetch upstream.**
* “**git fetch - -all - -prune**”
* Here prune means that remove the files from my repository that no longer exist in the upstream repository
* Now you have to reset the main branch of my origin to the main branch of upstream and in order to do that we have to write the command: **git reset - -hard upstream/main**
* Now you can do “**git log**” to look at the commits
* “**git push origin main**” to push the changes to your local repository
* **IMPORTANT:** Same thing can be done with “**git pull upstream main**” Then“**git push origin main**”

**Merging Commits**

Let’s say you have four commits and you want to merge them to a single commit then it has two approaches.

1. First approach is to reset the higher commit so that the lower commits gets unstaged and when they are unstaged make them on stage all at once so that the commit will be only one.
2. The next approach is by using **Pick** and **Squash**, you need to write the following command:

* “**git rebase –i <commit id of the head commit i-e upper commit>**”
* Here –i means **interactive environment**,there will be some options of pick and squash… ***s*** for squash and ***p*** for pick. Now the commits that you write s with will all be merged with the commit having ***p*** at the above of the ***s*** commits
* Now do “**git push origin <branch name>**”

**Merging Conflicts**

If two people make a change on the same line and same file and the owner has clicked on the merge commits on the both of the files then the git will get confused which to make visible on the main file. So it asks the owner to resolve the conflict. Then owner deletes one person’s code and make the other person’s code to be merged in the main file to resolve the conflict.