**Session 01.**

Git is a useful tool for version control system (VCS) that is required to know if you want to be welcomed in the programming world.

Imagine you are developing a program and you are done with version 1 and you went further on v2. Now you found a bug in your code and you want to come back to older version, in this case, you better to have a version control system which takes care of your versions and is able to help you back to previous version. It is not only for programming world, but also for everything else you need to have a record of it.

To start learning, you need to download git and install it on your computer. If you did so, by typing ‘git’ in terminal (MacOS or Linux) or PowerShell (Windows), you must see something but error.

~git 🡪 output:

1. usage: git [--version] [--help] [-C <path>] [-c <name>=<value>]

2. [--exec-path[=<path>]] [--html-path] [--man-path] [--info-path]

3. [-p | --paginate | -P | --no-pager] [--no-replace-objects] [--bare]

4. [--git-dir=<path>] [--work-tree=<path>] [--namespace=<name>]

5. <command> [<args>]

6. These are common Git commands used in various situations:

7. start a working area (see also: git help tutorial)

8. clone Clone a repository into a new directory

9. init Create an empty Git repository or reinitialize an existing one

10. work on the current change (see also: git help everyday)

11. add Add file contents to the index

12. mv Move or rename a file, a directory, or a symlink

13. restore Restore working tree files

14. rm Remove files from the working tree and from the index

15. sparse-checkout Initialize and modify the sparse-checkout

16. examine the history and state (see also: git help revisions)

17. bisect Use binary search to find the commit that introduced a bug

18. diff Show changes between commits, commit and working tree, etc

19. grep Print lines matching a pattern

20. log Show commit logs

21. show Show various types of objects

22. status Show the working tree status

23. grow, mark and tweak your common history

24. branch List, create, or delete branches

25. commit Record changes to the repository

26. merge Join two or more development histories together

27. rebase Reapply commits on top of another base tip

28. reset Reset current HEAD to the specified state

29. switch Switch branches

30. tag Create, list, delete or verify a tag object signed with GPG

31. collaborate (see also: git help workflows)

32. fetch Download objects and refs from another repository

33. pull Fetch from and integrate with another repository or a local branch

34. push Update remote refs along with associated objects

35.

36. 'git help -a' and 'git help -g' list available subcommands and some

37. concept guides. See 'git help <command>' or 'git help <concept>'

38. to read about a specific subcommand or concept.

39. See 'git help git' for an overview of the system.

40.

**Session 02.**

Git is a software that every time you run it looks to the files in the current directory and decide which it should do based on its criteria.

To start, we can create a directory to investigate how git works.

~ mkdir test

~ cd test

Now, we want to initialize git for this directory, so we type:

~ git init

~ ls -ltrha (🡪 in Unix based OS)

By entering ls -ltrha, you can see a folder (“.git”) is created.

There is another important function in git which shows the latest status of our directory:

~ git status

If you type it here, you probably see the results below showing that you just initialized the git and there is no new thing.

On branch master

No commits yet

nothing to commit (create/copy files and use "git add" to track)

Let’s create an html file and see if this file is in git or not. By using ~ git status, you see this file is untracked and if you want to include it in your git, you must use ~ git add <your file>.

‼ A point needed to be considered is that your file after changing go to STAGE situation and then by committing, they become the original file we are working on them. Every change takes the file into stage mode and then they need to be committed to be stored in the *.git* folder settings.

After adding the file, if you desire to accept changes, you can enter git commit (-m ‘…’). If you are going to add a note to your commit you can use the parentheses part in your commit command. After committing, you do not see anything in the stage mode (when you are entering git status).

Now, if you put a change into your file, and entering git status, you just see it says one file is modified but it is not staged. To move it to stage you can enter:

~ git add <your file or -A> -A is for including all files in the directory

Above line of code, put our file(s) into stage mode and ready for commit by

~ git commit -m ‘…’

**-Lesson 3**

There is a command named log in git that shows the history of your work:

~ git log

This command shows you history as newer commits are on top and each commit has its own exclusive code and date.

Now imagine your project is going further and you are going to add three more files to it.

~ touch page1.html page2.html page3.html

Also, using a text editor you can add texts to these files.

You probably remember that if we type git status, we see it returns you are on master branch and you have 3 untracked files. In case, we have a change in the index.html file together with creating new files, git status returns you have a modification (for index.html) and untracked files (for 3 new files).

Till the end of lesson three, we have become familiar with below commands of git:

~git init

~git status

~git add <your file(s) or -A>

~git commit -m ‘. . .’

~git log

**Lesson 4.**

If you make a change in one of your files, and want to see the difference, you can use git diff <HEAD>. HEAD is showing our current situation and the command generally looks at the new changes and your current files in HEAD.

I changed text in page1.html and we see the results.

~git diff HEAD

**diff --git a/page1.html b/page1.html**

**index 3bf08a8..3c48890 100644**

**--- a/page1.html**

**+++ b/page1.html**

@@ -1 +1,18 @@

-you are on page 1

\ No newline at end of file

Now it is important to interpret the results. Diff is a Unix system command to compare two files and a/ and b/ show old and new versions, respectively. ‘---’ and ‘+++’ also show older and newer versions. In the blue line, we see it stated that it is comparing the first line of older version and 1 and 18 lines of newer versions.

You can use –-staged instead of HEAD to see the difference in files in stage mode. imagine you are not happy with the changes you have made in page1.html and you do not want to commit it, so you can use command below to eliminate it from stage step:

~ git reset page1.html

If you want to commit other changes but page1.html, you are good to use command commit.

Another command is using checkout. It is useful when we are not sure our changes are good and we want to back every changes in the file to the previous version (last commit or HEAD).

~ git checkout –- page3.html

To summary, we had commands below in this lesson:

~git diff HEAD

~git diff -–staged

~git reset <file name>

~git checkout -- <file name>

**Lesson 5: Branches**

You have seen many times that YOU ARE ON MASTER BRANCH. Imagine you are working on a project and you committed several times, and now you decided to add a part to an older commit and then after that you merge it with the current status of the project. We call this a branch.



~ git branch <let’s say fixpages>

Using the above command, you create a branch for your work and when you type git branch you can see all branches on your workspace. To move to newer branch, you can type command below:

~ git checkout fixpages/master

Now you have modified your files based on your desire and it is turn to merging thgis branch with master branch; first we come back to master branch and then merge:

~ git checkout master

~ git merge fixpages

!! if you have created a file mistakenly or you no longer need a file, you can remove it using command below:

~ git rm <your file>

If you removed a file in a branch, coming back to master branch that file is still existed, unless you merge that branch with master branch. After merging and making sure you do not need the branch, you can delete it using command below:

~git branch -d <your branch>

**Lesson 6: GitHub**

Git claims that it is distributed so we expect we can save our job in different places. Apart from your hard disk, you can use GitHub or GitLab to distribute your work there. To start learning GitHub, we go to the website and find a random repository, for this tutorial we use [git@github.com:Sinamahani/git\_tut.git](mailto:git@github.com:Sinamahani/git_tut.git). Then you type in your terminal/PowerShell command below:

~ git clone [git@github.com:Sinamahani/git\_tut.git](mailto:git@github.com:Sinamahani/git_tut.git)

So git create a directory named same as your repository and download all files.

After some modifications and changes, you decide to upload this repository to your GitHub, so you can use command below:

~ git push origin master (or main in PowerShell)

In the meanwhile, it may ask you security question like username and password. Another important point to mention is that origin is the name of our repository on the GitHub (the URL) and you see this name frequently when cloning a repo from GitHub and also when pulling (next paragraph).

In case you are working on a project that someone else is working on too, you can get updates of GitHub using command below:

~ git pull origin master (main)

If a file added or deleted or modified in the main/master on GitHub, you will be up-to-date using command above.

Now imagine, you started a project on your system and you have a repo on GitHub and you want to add your project to that repo. Since you have not cloned, you do not have an origin. In such a case, youi need to create a remote using command below:

~ git remote 🡪 showing all remote created already

~ git remote add origin <URL>

As you see above, we have a keyword named origin. This keyword can be anything else. In addition, URL is the address to your repo.

**Lesson 7: Conflicts**

Assume you and your colleague working on a file and one of you first push the file (or edit the file through GitHub) and now the other one wants to push. At first git tries to merge each part of your work, unless you both edited same part. In such a case, you face a conflict. The best course of action in this case is to pull the repo and edit it manually and then push it.

If you merge your files shortly, you probably won’t face such a difficulty.

**Git – Going Further**

**Lesson 8: Tag**

You have developed a code with many commits. It is obviously hard to go through all commit messages to find the relevant one. The best idea here is to tag you code by things like v.1 or v.0.1 etc. If in your terminal (PowerShell) you type git log, you probably see many logs but by git tag you won’t see anything because you have not done so.

To add your first tag:

~ git tag -a <version> -m ‘<your message>’ ; -a showing annotating the tag name

In addition, if you go through your commits and find out you for instance wants to have a version 0.1 for a commit, you can use code below:

~ git tag -a <version name> <commit code from logs>

Now if you use git show, you can see the detail about the tag

~ git show <version name>

Other point here is about pushing. If use git push, it doesn’t push your tags, to do so:

~ git push origin –tags

or

~ git push origin <version name>

**Lesson 0: Signing tags and commits**

In computer world we have a concept named coding that makes key showing that who signed that product. Pretty good privacy (pgp) is the most famous one so far; however, in Linux we have gpg which is open source. If it is installed on your system, you can type gpg –list-keys to see all your keys.

When you create a key, you have two keys; public and private. If something is signed by your public key, it shows you provided the key to others and it was not you who changed the files.

To create a new key:

~ gpg –gen-key

Then it asks your name and your email and confirmation of those. Finally it asks your password. If you list your keys now, you will see your public keys. To see secret keys (private keys):

~ gpg –list-secret-keys –keyid-format LONG

Now you can add your key to your git config.

~git config –global user.signingkey <key from previous step>

~git tag -s v2.1 -m <your message> ; -s indicating signing your version

The last line of code, signs your version based on you key. If you show and verify (-v) your tag, you can see the key showing that you are the signer.

For more reading:

~ git help <your function>

~ git bisect

~ got blame

**Special Characters on Windows**

Alt + 224 🡪 α

Alt + 225 🡪 ß

Alt + 226 🡪 Γ

Alt + 227 🡪 π

Alt + 228 🡪 Σ

Alt + 229 🡪 σ

Alt + 230 🡪 µ

Alt + 231 🡪 τ

Alt + 232 🡪 Φ

Alt + 233 🡪 Θ

Alt + 234 🡪 Ω

Alt + 235 🡪 δ

Alt + 236 🡪 ∞

Alt + 237 🡪 φ

Alt + 238 🡪 ε

Alt + 239 🡪 ∩

Alt + 240 🡪 ≡

Alt + 241 🡪 ±

Alt + 242 🡪 ≥

Alt + 243 🡪 ≤

Alt + 244 🡪 ⌠

Alt + 245 🡪 ⌡

Alt + 246 🡪 ÷

Alt + 247 🡪 ≈

Alt + 248 🡪 °

Alt + 249 🡪 ∙

Alt + 250 🡪 ·

Alt + 251 🡪 √

Alt + 252 🡪 ⁿ

Alt + 253 🡪 ²

Alt + 254 🡪 ■