# GIT:

https://raybo.org/slides_practicalactions/#/

https://github.com/LinkedInLearning/learning-git-github-2421501/tree/01_02b

## Git Config:

Configure the user name and email for global git

git config --global user.name “your name”

git config --global user.email “your email”

## initialize git on folder:

git init

## Staging Files:

Area where files are temporary stored

git add FILENAME

### To add all files to the project

git add --all

git add -A

git add .

commit the files, always include a message with it.

git commit  -m “message”

### View the actions or logs:

git log

In the log command we see a HEAD pointer pointing to current branch which we are working on

## Git environment:

- Working – here the files look like when they were in last commit
- Staging - add files here using add command, queue the changes before commit
- Commit – a new log entry is created

File states:

- Tracked: existed when previous commit was done
    - Unmodified:
    - Modified
    - Staged

  Check by using command

  git status

- Untracked:not present when previous commit was done

## Git Restore:

To undo the changes and bring the file to last committed state,

NOTE: it will not automatically change an untracked file

git restore FILENAME

to remove the file from staging state

git restore --staged FILENAME

git restore .

git checkout .

## ignore the files:

for sensitive data or notes we can choose to not update them on git, in order to do so we create a ".gitignore" file at root level of the project and in that we can add the file names or pattern of files names we need to ignore


Also it can be done at a global level using

git config --global core.excludesfile [file]

if the ignore is not picked up immediately clear the cache:

git rm -r --cached .

## Deleting files:

Manually deleting files, will put them in working folder

Another way is to use

git rm FILENAME

this will automatically move the file to staging state

to restore use first restore with staging flag and then normal restore

## Git RENAME:

When you rename a file through explorer, git register it as 2 actions deletion of the renamed file and then creating a new file with the contents in it.

If you try to restore after this step it will undo the delete of the file and the renamed file will still exist as new file. You have to manually delete it.

Other option is:

git mv OldFileName NewFileName

## Difference:

In order to check what is changed we can use the below command

git diff <diff hash in logs>

but if there are too many changed it is hard to see what is changed, here we can use visual studio code tool – “source control”

you can also use “git Lense” for diffing purpose

## Logging:

For seeing all the change logs use command

git log

if the logs are too long you can use

git log –oneline

## Amending:

Adding things to last commit, if you don’t want to change the commit history too much

git commit -amend

git commit -am ‘new commit message’

git commit -amend --no-edit

## RESET:

git reset <hash>

it will reset the commit flag to another commit.

git reset --hard <hash>

it will reset the commit and also delete and modify the files to that commit

## Rebase:

To  get commits from one branch and apply to another branch, or change the commit order

git rebase <branch>/<commit>

git rebase --interactive <branch>/<commit>

git rebase –i HEAD~#

git rebase --interactive –root

there are many other options in this rebase and you can see once interactive window is open

## Branches:

Look at all the branchs in the repository

git branch

make a copy of existing branch:

git switch -c NAME

get checkout -b NAME

## Merging:

In order to make changes to main we need to merge the branches

git merge <branch>

first we switch from other branch to main branch using

git switch main

and using the merge command

## deleting branch:

also after the feature is done we delete the branch after merging

using command

git branch –delete Name

git  branch -d name

for force deletion

git branch -D name

## Merge Conflicts:

You can do it manually and select what changes you want to keep and then do the merge

## Stashing:

It is useful when you want to just stash away your changes and then later use them or work on them instead of committing them to branch

Revert the branch to last commit and save the changes done in a stash.

git stash

list the stashed changes:

git stash list

apply the stashed changes:

git stash apply <no in list you need to apply>

to apply the last change in the stash

git stash pop

## Clean:

For removing any untracked files which restore cannot do.

git clean -n #dry run

git clean -d #directories

git clean -f #force

# GITHUB:

## Remotes:

1. git remote add NAME URL:

- Explanation: Adds a new remote repository under the specified name (alias). The remote repository is a location where your local repository can push or pull code.

- Example: If you want to add a new remote named origin that points to your GitHub repository URL:

bash

git remote add origin https://github.com/your-username/your-repo.git

- Now, you can push to this remote using git push origin branch-name.

2. git remote remove NAME:

- Explanation: Removes an existing remote from your repository configuration. This won't affect the actual remote repository but will remove your connection to it.

- Example: If you need to remove a remote named origin:

bash

git remote remove origin

- This deletes the reference to the origin remote.

3. git remote rename OldName NewName:

- Explanation: Renames an existing remote from OldName to NewName. This is helpful if you want to reorganize or clarify the naming of your remotes.

- Example: If you want to rename a remote from origin to main-remote:

bash

```
git remote rename origin main-remote
```

- Now, the remote can be referred to as main-remote.

4. git remote -v:

- Explanation: Lists all currently configured remotes and their URLs, displaying which remotes are set up for fetching and pushing.

- Example: To view your configured remotes:

```
bash

git remote -v
```

- The output will look like this:

```
origin  https://github.com/your-username/your-repo.git (fetch)

origin  https://github.com/your-username/your-repo.git (push)
```

These commands will help you efficiently manage remotes in your Git workflow!

## Push:

1. git push -u origin main:

- Explanation: Pushes the main branch to the remote named origin and sets it as the default upstream branch. This means future git push commands will default to pushing to origin/main.

- Example: If you've committed changes to the local main branch and want to push it to the remote origin:

```
bash

git push -u origin main
```

- Now, the upstream for main is set, and subsequent pushes can be done just with git push.

2. git push --all:

- Explanation: Pushes all branches from your local repository to the configured remote. This is useful if you have multiple branches that need to be synchronized.

- Example: To push all branches to the origin remote:

  bash

  git push --all origin

- All branches from the local repository will now be available on the origin remote.

3. git branch --set-upstream-to <origin/remote-branch>:

- Explanation: Sets the upstream tracking branch for a local branch, linking it to a specific remote branch. This allows git pull and git push to work seamlessly with that remote branch.

- Example: If you have a local branch feature and want it to track a remote branch origin/feature:

  bash

  git branch --set-upstream-to origin/feature feature

- Now, when you're on the feature branch, commands like git pull will fetch changes directly from origin/feature.