Seeing File Differences

- We can use ***diff -u*** command from the terminal to see the differences in 2 files.
  diff -u file1.py file2.py
- We can use ***wdiff*** to see the words which have changed in a file.
- Some graphical tools for this are- ***meld, kdiff3, vimdiff***

Applying File Differences

- To apply the differences in the file, we use ***patch*** command. We pass the diff file as the second parameter.
  patch file1.py < file1.diff
- To create a diff file, we use the following command
  diff -u original.py fixed.py > repair.diff

Using Git

Git repo initialisation

- The command to initialise a git repo is ***git init.***
- The working tree has the current working version.
- Staging area (index)- A file maintained by Git that contains all of the information about what files and changes are going to go into your next commit.

Adding new files/ Staging the changes

- We use ***git add file.py*** to add a file to the staging area.
- By using ***git add ****, we stage all the changes to all the files.

Checking current status

- We can check the current working tree and pending changes by using ***git status***.

Committing files

- We commit the file using command ***git commit*** and specify the commit message in the editor. To specify the commit message in the command itself, we use ***-m flag***. Example- ***git commit -m "Message"***

Git Terminology

- Any git project will have 3 components- the Git directory, the working tree and the staging area.
  The Git directory contains the history of all the files and changes.

The working tree contains the current state of the project, including any changes that we've made.
The staging area contains the changes that have to be made in the next commit.

- There are two types of files- tracked and untracked.
  Tracked files are present in the snapshot whereas untracked files are not.
- The files can be either modified, staged or committed.
  Modified files mean that we have made changes to it but haven't committed them yet.
  The next step is to stage the files and then they become ready for commit.
  After the commit, the changes made to it are safely stored in a snapshot in a Git directory.

Checking log and the changes

- We use ***git log*** to review the history of all the commits.
- We can commit the changes in files directly without the staging step by using ***-a flag*** with ***git commit*** like ***git commit -a***. This works only for modified files and not for new files as they are untracked files.
- HEAD is used to indicate which snapshot is currently checked out.
- Using ***-p flag*** with ***git log***, we can see the actual lines which changed in each commit. The p is for patch and it shows the differences the same way.
- If we want to see the changes because of a particular commit, we can use ***git show commit_id***. The commit_id is the id which occurs after the word commit when we run ***git log***.
- We can also use ***--stat flag*** with ***git log*** command which shows statistics related to the changes.
- We can use ***git diff*** if we want to see differences in a file when it is not yet committed. This works like the ***diff -u*** talked above. It shows only the unstaged changes by default. If we want to see the staged changes too, we use the ***--staged flag***.
- If we want to see the changes that will be committed during the staging step, we use the ***-p flag*** with ***git add***. This will also confirm from us if we want to stage the changes or not.

Removing files

- We use ***git rm*** to remove a file from the Git directory. We also need to commit after removing a file as it is also a change. It is automatically staged.

Renaming/Moving files

- We use ***git mv old_name.py new_name.py*** to rename the files in a Git directory. In this too, it is automatically staged but we need to commit.

.gitignore file

- We use .gitignore file to ignore the unnecessary files which we don't want to show up in our directory. In this file, we specify the rules to tell Git which files to skip.

Undoing changes before committing

- We can use ***git checkout filename.py*** to undo the changes made to a file after the last commit. This basically checks out the version of the file from the latest snapshot. This effectively refers to switching branches.
- To remove the changes from the staging area, we can use ***git reset HEAD filename.py***.

Undoing changes after committing

- We use ***git commit --amend*** to make changes to modify and add changes to the most recent commit. Example- Incomplete commits.

Rollback a commit

- We can use ***git revert commit_id/HEAD*** (for latest one) to create a commit that contains the inverse of all the changes made in the bad commit in order to cancel them out. This will perform a new commit to provide the previous working version and thus will be a record in the history too.

Branches

- A branch is a pointer to a particular commit.
- We can create a new branch, work in it till the code works successfully and then merge the final changes with the master branch.
- We can see the current branches using ***git branch***.

Creating and Switching Branches

- We can create a new branch using ***git branch branch_name***.
- We use ***git checkout*** to checkout the latest snapshot for both files and for branches.
- To switch to a particular branch, we use ***git checkout branch_name***.

- To create a new branch and also switch to it in a single command, we use ***git checkout -b branch_name***.

Committing to a branch

- When we commit to a branch, it is committed only to the history of that branch.

Deleting a branch

- To delete a branch, we use ***git branch -d branch_name***.

Merging Branches

- We can merge a branch to the master branch using ***git merge branch_name***.
- If there are merge conflicts and we want to revert the merge, we can use ***git merge --abort***.
- Merge conflicts arise when one version of a file is committed in the branch and another version of the file is committed in the master. We have to see what changes to keep.
- After we make changes to the file to resolve conflicts, we have to add it and then commit it separately.
- To solve conflicts, Git uses 2 algorithms, fast forward merge and three way merge. Fast forward merge is the one in which the master branch has no commits after the new branch. To merge the branches, only the HEAD for the master branch has to move forward.

GitHub

- We can clone repo from GitHub to our local machine by using ***git clone "url"***.
- We can then make all the necessary changes in the repo locally.
- We can list the remote repos using ***git remote***.
- To push the local changes to the remote repo, we use ***git push***.
- To not enter credentials again and again, we can keep a SSH key-pair in our laptop or cache it for a particular time using ***git config --global credential_helper cache***.
- We can pull new changes from the remote repo by using ***git pull***.
- We can look at the configuration for a remote by using ***git remote -v***. One URL is used to fetch data and the other is used to push it.
- To get more information about the remote repo/branch, we can use ***git remote show origin***.
- We could have a look at the remote branches that our Git repo is currently tracking by running ***git branch -r***.

- To fetch the changes in the remote branch for our local branch, we use ***git fetch***. We have to merge these changes separately using ***git merge***.
- To combine the above operations, we can simply use ***git pull***.
- If we want to get the contents of remote branches without automatically merging any contents into the local branches, we can call ***git remote update***. This will fetch the contents of all remote branches, so that we can just call checkout or merge as needed.
- To push a branch to the remote repo, we have to add info to the ***git push***. We use it as ***git push -u origin branch_name***.
- To delete a remote branch, we use ***git push --delete origin branch_name***.

Rebasing

- It means changing the base commit that is used for our branch.
- When we try to merge a file which has been updated in both the versions, there is a three way merge. To prevent the issues in this, we can use rebasing to replay the new commits on top of the new base (the updated version of file not in branch). This makes the tree linear.
- The command for this is ***git rebase branch_name***. This moves the current branch on top of the specified branch_name.
- Squashing refers to squash all the changes of multiple commits into a single commit. We use ***git rebase -i master*** for this purpose.
- We can combine the commits in two ways- squash and fix up. When we choose squash, the commit messages are added together and an editor opens up to let us make any necessary changes. When we choose fix up, the commit message for that commit is discarded.

Git commands

- git init
- git add -p, *
- git status
- git commit -a -m " ", --amend
- git log -p --stat
- git show commit_id
- git diff --staged
- git rm
- git mv
- git checkout
- git reset