GIT

- [ ] **What is GIT-**
  - Git is a modern and widely used **distributed version control** system in the world. It is developed to manage projects with high speed and efficiency. The *version control system* allows us to monitor and work together with our team members at the same workspace.
  - Git is foundation of many services like **GitHub** and **GitLab**, but we can use Git without using any other Git services. Git can be used **privately** and **publicly**.
  - Git was created by **Linus Torvalds** in **2005** to develop Linux Kernel. It is also used as an important distributed version-control tool for **the DevOps**.
  - Git is easy to learn, and has fast performance. It is superior to other *Source Code Management* tools like Subversion, CVS, Perforce, and ClearCase.
  - Local Repository of Project is placed at Git which is on Developer Machine.
- [ ] **Features Of GIT-**
  - **Open Source**
    Git is an **open-source tool**. It is released under the **GPL** (General Public License) license.
  - **Scalable**
    Git is **scalable**, which means when the number of users increases, the Git can easily handle such situations.
  - **Distributed**
    One of Git's great features is that it is **distributed**. Distributed means that instead of switching the project to another machine, we can create a "clone" of the entire repository. Also, instead of just having one central repository that you send changes to, every user has their own repository that contains the entire commit history of the project. We do not need to connect to the remote repository; the change is just stored on our local repository. If necessary, we can push these changes to a remote repository.
  - **Git Integrity**
    Git is **developed to ensure** the **security** and **integrity** of content being version controlled. It uses checksum during transit or tampering with the file system to confirm that information is not lost. Internally it creates a checksum value from the contents of the file and then verifies it when transmitting or storing data.
  - **Trendy Version Control System**
    Git is the **most widely used version control system**. It has **maximum projects** among all the version control systems. Due to its **amazing workflow** and features, it is a preferred choice of developers.
  - **Everything is Local**
    Almost All operations of Git can be performed locally; this is a significant reason for the use of Git. We will not have to ensure internet connectivity.
  - **Collaborate to Public Projects**
    There are many public projects available on the GitHub. We can collaborate on those projects and show our creativity to the world. Many developers are collaborating on public projects.

The collaboration allows us to stand with experienced developers and learn a lot from them; thus, it takes our programming skills to the next level.

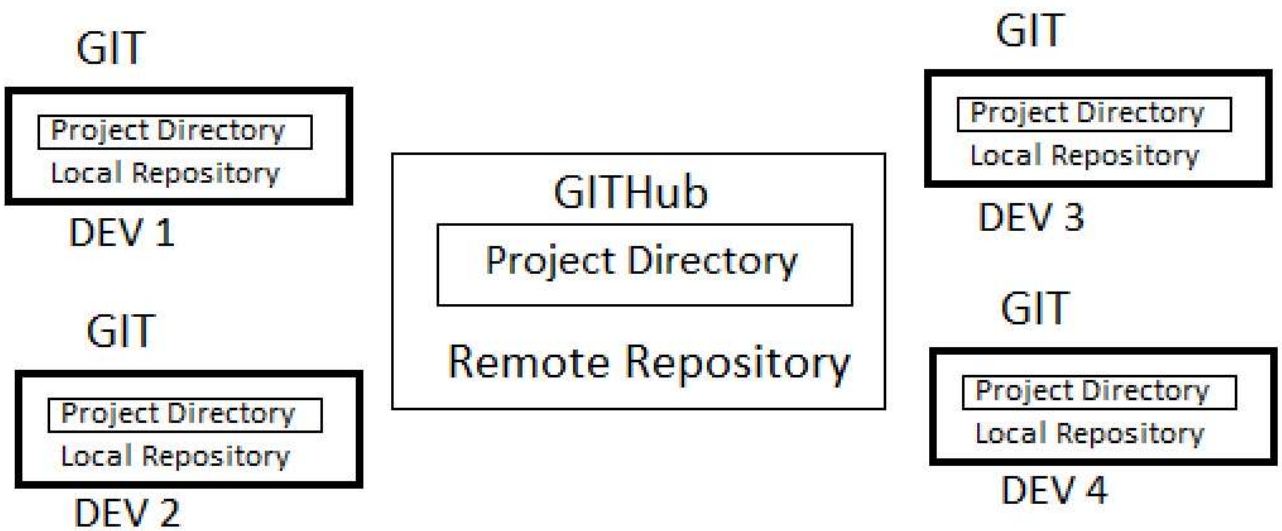
- **Impress Recruiters**
  We can impress recruiters by mentioning the Git and GitHub on our resume. Send your GitHub profile link to the HR of the organization you want to join. Show your skills and influence them through your work. It increases the chances of getting hired.

- [ ] **What is GitHub..??**

- GitHub is a Git repository hosting service. GitHub also facilitates with many of its features, such as access control and collaboration. It provides a Web-based graphical interface.
- GitHub is an American company. It hosts source code of your project in the form of different programming languages and keeps track of the various changes made by programmers.
- It offers both **distributed version control and source code management (SCM)** functionality of Git. It also facilitates with some collaboration features such as bug tracking, feature requests, task management for every project.

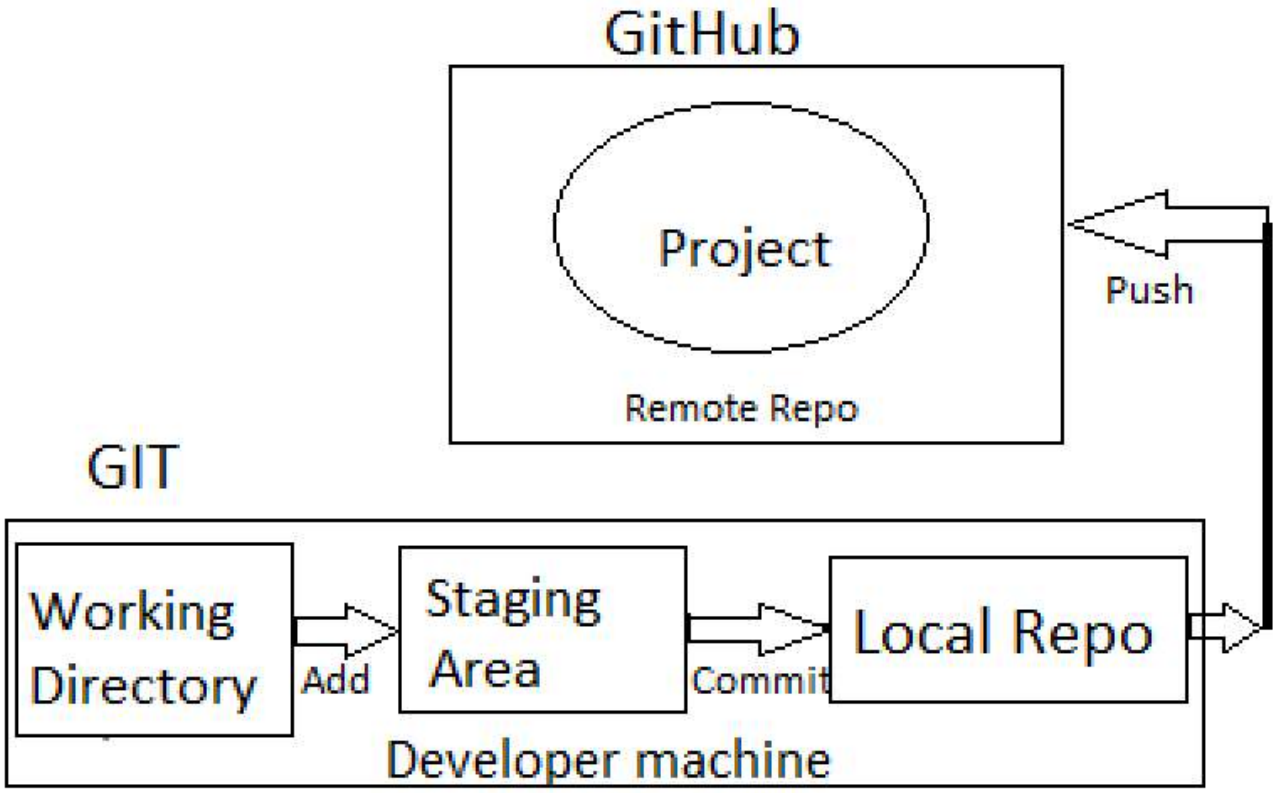
- [ ] **Benefits Of GitHub-**

- GitHub can be separated as the Git and the Hub. GitHub service includes access controls as well as collaboration features like task management, repository hosting, and team management.
  The key benefits of GitHub are as follows.
  - It is easy to contribute to open source projects via GitHub.
  - It helps to create an excellent document.
  - You can attract recruiter by showing off your work. If you have a profile on GitHub, you will have a higher chance of being recruited.
  - It allows your work to get out there in front of the public.
  - You can track changes in your code across versions.
  - There two types of account on GitHub,1) Private Account 2) Public Account
  - Remote Repository of Project is placed at GitHub.



- [ ] **Combined Scenario for GIT and GitHub-**

- There is staging area at developer end for final check and then-after developer can push the code at remote repo.



- [ ] **Git Tools-**

- To explore the robust functionality of Git, we need some tools. Git comes with some of its tools like Git Bash, Git GUI to provide the interface between machine and user. It supports inbuilt as well as third-party tools.
- Git comes with built-in GUI tools like **git bash**, **git-gui**, and **gitk** for committing and browsing. It also supports several third-party tools for users looking for platform-specific experience.

1.Git Bash-

- Git Bash is an application for the Windows environment. It is used as Git command line for windows. Git Bash provides an emulation layer for a Git command-line experience. Bash is an abbreviation of **Bourne Again Shell**. Git package installer contains Bash, bash utilities, and Git on a Windows operating system.
- Bash is a standard default shell on Linux and macOS. A shell is a terminal application which is used to create an interface with an operating system through commands.
- By default, Git Windows package contains the Git Bash tool. We can access it by right-click on a folder in Windows Explorer.

2.Git GUI-

- Git GUI is a powerful alternative to Git BASH. It offers a graphical version of the Git command line function, as well as comprehensive visual diff tools. We can access it by simply right click on a folder or location in windows explorer. Also, we can access it through the command line by typing below command.

  $ git gui

$ git gui

3.Gitk-

- gitk is a graphical history viewer tool. It's a robust GUI shell over **git log** and **git grep**. This tool is used to find something that happened in the past or visualize your project's history.
- Gitk can invoke from the command-line. Just change directory into a Git repository, and type: $gitk

---

- [ ] **Git Terminologies-**

- Git is a tool that covered vast terminology and jargon, which can often be difficult for new users, or those who know Git basics but want to become Git masters. So, we need a little explanation of the terminology behind the tools. Let's have a look at the commonly used terms.

1.Branch-

- A branch is a version of the repository that diverges from the main working project. It is an essential feature available in most modern version control systems. A Git project can have more than one branch. We can perform many operations on Git branch-like rename, list, delete, etc.

2.Checkout-

- In Git, the term checkout is used for the act of switching between different versions of a target entity. The **git checkout** command is used to switch between branches in a repository.

3.Clone-

- The **git clone** is a Git command-line utility. It is used to make a copy of the target repository or clone it. If I want a local copy of my repository from GitHub, this tool allows creating a local copy of that repository on your local directory from the repository URL.

4.Fetch-

- It is used to fetch branches and tags from one or more other repositories, along with the objects necessary to complete their histories. It updates the remote-tracking branches.

5.HEAD-

- It is used to fetch branches and tags from one or more other repositories, along with the objects necessary to complete their histories. It updates the remote-tracking branches.

6.Staging/Index Area-

- The Git index is a staging area between the working directory and local repository. It is used as the index to build up a set of changes that you want to commit together.
- The staging area can be described as a preview of your next commit. When you create a git commit, Git takes changes that are in the staging area and make them as a new commit. You are allowed to add and remove changes from the staging area. The staging area can be considered as a real area where git stores the changes.

7.Master-

- Master is a naming convention for Git branch. It's a default branch of Git. After cloning a project from a remote server, the resulting local repository contains only a single local branch. This branch is called a "master" branch. It means that "master" is a repository's "default" branch.

8.Merge-

- Merging is a process to put a forked history back together. The git merge command facilitates

you to take the data created by git branch and integrate them into a single branch.

9.Origin-

- In Git, "origin" is a reference to the remote repository from a project was initially cloned. More precisely, it is used instead of that original repository URL to make referencing much easier.

10.Pull/Pull Request-

- The term Pull is used to receive data from GitHub. It fetches and merges changes on the remote server to your working directory. The **git pull command** is used to make a Git pull.
- Pull requests are a process for a developer to notify team members that they have completed a feature. Once their feature branch is ready, the developer files a pull request via their remote server account. Pull request announces all the team members that they need to review the code and merge it into the master branch.

11.Push-

- The push term refers to upload local repository content to a remote repository. Pushing is an act of transfer commits from your local repository to a remote repository. Pushing is capable of overwriting changes; caution should be taken when pushing.

12.Rebase-

- In Git, the term rebase is referred to as the process of moving or combining a sequence of commits to a new base commit. Rebasing is very beneficial and visualized the process in the environment of a feature branching workflow.
- From a content perception, rebasing is a technique of changing the base of your branch from one commit to another.

13.Remote-

- In Git, the term remote is concerned with the remote repository. It is a shared repository that all team members use to exchange their changes. A remote repository is stored on a code hosting service like an internal server, GitHub, Subversion and more.
- In case of a local repository, a remote typically does not provide a file tree of the project's current state, as an alternative it only consists of the .git versioning data.

14.Repository-

- In Git, Repository is like a data structure used by VCS to store metadata for a set of files and directories. It contains the collection of the file as well as the history of changes made to those files. Repositories in Git is considered as your project folder. A repository has all the project-related data. Distinct projects have distinct repositories.

15.Stashing-

- Sometimes you want to switch the branches, but you are working on an incomplete part of your current project. You don't want to make a commit of half-done work. Git stashing allows you to do so. The **git stash command** enables you to switch branch without committing the current branch.

16.Git Revert-

- In Git, the term revert is used to revert some commit. To revert a commit, **git revert** command is used. It is an undo type command. However, it is not a traditional undo alternative.

17.Git Reset-

- In Git, the term reset stands for undoing changes. The **git reset** command is used to reset the

changes. The git reset command has three core forms of invocation. These forms are as follows.

1.Soft

2.Mixed

3.Hard

18.Git Diff-

- Git diff is a command-line utility. It's a multiuse Git command. When it is executed, it runs a diff function on Git data sources. These data sources can be files, branches, commits, and more. It is used to show changes between commits, commit, and working tree, etc.

19.Git Squash-

- In Git, the term squash is used to squash previous commits into one. Git squash is an excellent technique to group-specific changes before forwarding them to others. You can merge several commits into a single commit with the powerful interactive rebase command.

20.Git Rm-

- In Git, the term rm stands for **remove**. It is used to remove individual files or a collection of files. The key function of git rm is to remove tracked files from the Git index. Additionally, it can be used to remove files from both the working directory and staging index.

How to set Git username and Password in GitBash..??

- Git is usually bundled as one of the elevated GUI applications in Windows settings. Bash is a popular Linux and Mac OS default shell. On a Windows operating system, Git Bash is a package that installs Bash, some standard bash utilities, and Git. The fundamental version control system primitives may be abstracted and hidden by Git GUI s. Git Bash is a Microsoft Windows software that functions as an abstraction shell for the Git command-line experience. A shell is a console application that allows you to interact with your operating system by command prompt.
- Let set Username and Email in Git Bash ,follow below commands-

1. git config --global user.name ''<User_Name>"
2. git config --global user.email "<Email_Address>"

Stagging and Commits-

1.Git Init-

- The git init command is the first command that you will run on Git. The git init command is used to create a new blank repository. It is used to make an existing project as a Git project. Several Git commands run inside the repository, but init command can be run outside of the repository.
- The git init command creates a .git subdirectory in the current working directory. This newly created subdirectory contains all of the necessary metadata. These metadata can be categorized into objects, refs, and temp files. It also initializes a HEAD pointer for the master branch of the repository.

```
Abhimanyu Devadhe@LAPTOP-BCK0BG0F MINGW64 ~/Desktop/Git Demo
$ git init
Initialized empty Git repository in C:/Users/Abhimanyu Devadhe/Desktop/Git Demo/.git/
```

- The above command will initialize a **.git** repository on the Git Demo. Now we can create and add files on this repository for version control.
- To create a file, run the cat or touch command as follows:

  $touch <filename>

```
Abhimanyu Devadhe@LAPTOP-BCK0BG0F MINGW64 ~/Desktop/Git Demo (master)
$ touch Gitdemo.txt
```

- We can list all the untracked files by git status command.

```
Abhimanyu Devadhe@LAPTOP-BCK0BG0F MINGW64 ~/Desktop/Accenture/accenture (develop4)
$ git status
On branch develop4
Your branch is up to date with 'origin/develop4'.

Untracked files:
  (use "git add <file>..." to include in what will be committed)
        dem1.py
        demo2.py
        demo3.py

nothing added to commit but untracked files present (use "git add" to track)
```

2.Git Add-

- The git add command is used to add file contents to the Index (Staging Area).This command updates the current content of the working tree to the staging area. It also prepares the staged content for the next commit. Every time we add or update any file in our project, it is required to forward updates to the staging area.
- **Git add** command is a straight forward command. It adds files to the staging area. We can add single or multiple files at once in the staging area. It will be run as:

  $ git add <File name>

```
Abhimanyu Devadhe@LAPTOP-BCK0BG0F MINGW64 ~/Desktop/Accenture/accenture (develop4)
$ git add dem1.py

Abhimanyu Devadhe@LAPTOP-BCK0BG0F MINGW64 ~/Desktop/Accenture/accenture (develop4)
$ git status
On branch develop4
Your branch is up to date with 'origin/develop4'.

Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        new file:   dem1.py

Untracked files:
  (use "git add <file>..." to include in what will be committed)
        demo2.py
        demo3.py
```

- With the help of git add <filename> command we only add specific file to staging area.
- **Git Add All-**We can add more than one files in Git, but we have to run the add command repeatedly. Git facilitates us with a unique option of the add command by which we can add all the available files at once. To add all the files from the repository, run the add command with **-A** option. We can use '.' Instead of **-A** option. This command will stage all the files at a

time. It will run as follows:

$git add .

```
Abhimanyu Devadhe@LAPTOP-BCK0BG0F MINGW64 ~/Desktop/Accenture/accenture (develop4)
$ git add .

Abhimanyu Devadhe@LAPTOP-BCK0BG0F MINGW64 ~/Desktop/Accenture/accenture (develop4)
$ git status
On branch develop4
Your branch is up to date with 'origin/develop4'.

Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        new file:   dem1.py
        new file:   demo2.py
        new file:   demo3.py
```

- With the help of git add . command we can add all files which is untracked to staging area.

3.Git Commit-

- It is used to record the changes in the repository. It is the next command after the git add. Every commit contains the index data and the commit message. Every commit forms a parent-child relationship. When we add a file in Git, it will take place in the staging area. A commit command is used to fetch updates from the staging area to the repository.
- **Git Commit -m " msg"-**The -m option of commit command lets you to write the commit message on the command line. This command will not prompt the text editor. It will run as follows:

  $ git commit -m "Default Message"
- The above command will make a commit with the given commit message.

```
Abhimanyu Devadhe@LAPTOP-BCK0BG0F MINGW64 ~/Desktop/Accenture/accenture (develop4)
$ git status
On branch develop4
Your branch is up to date with 'origin/develop4'.

Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        new file:   dem1.py
        new file:   demo2.py
        new file:   demo3.py


Abhimanyu Devadhe@LAPTOP-BCK0BG0F MINGW64 ~/Desktop/Accenture/accenture (develop4)
$ git commit -m"3 Files are created "
[develop4 c8a80f0] 3 Files are created
 3 files changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 dem1.py
 create mode 100644 demo2.py
 create mode 100644 demo3.py
```

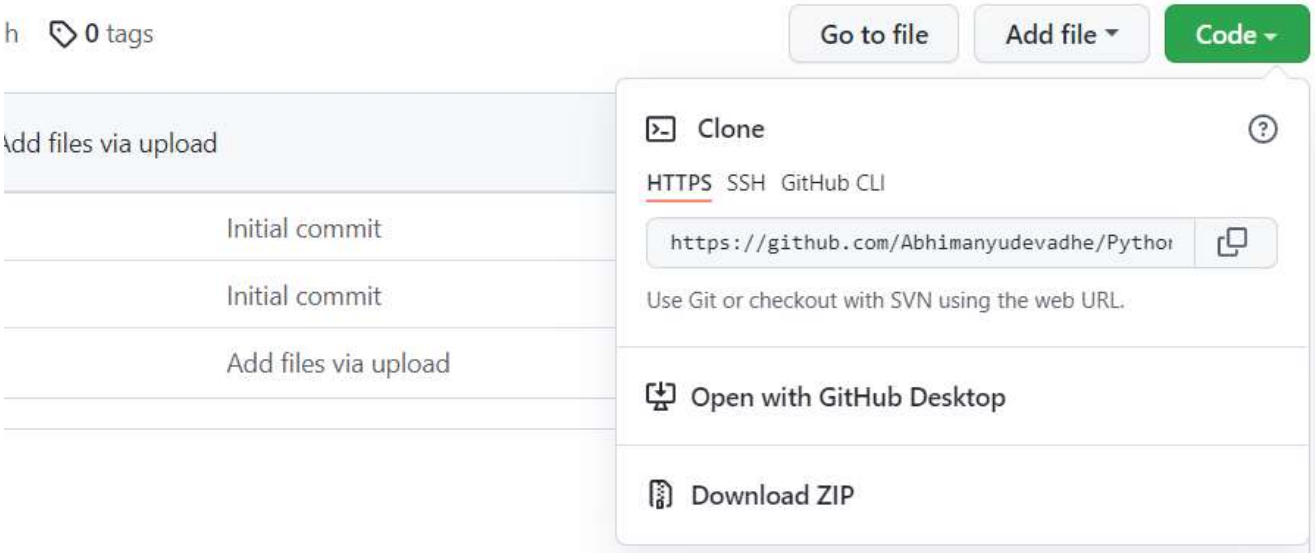
- This command will move all the files at staged area to Local Repository.

4.Git Clone-

- In Git, cloning is the act of making a copy of any target repository. The target repository can be remote or local. You can clone your repository from the remote repository to create a local copy on your system. Also, you can sync between the two locations.
- **git clone** is a command-line utility which is used to make a local copy of a remote repository. It accesses the repository through a remote URL.Usually, the original repository is located on

a remote server, often from a Git service like GitHub, Bitbucket, or GitLab. The remote repository URL is referred to the **origin**.

- Firstly create new folder in your machine and open git bash from there.
- Copy Remote repository url from Git hub.



Syntax= git clone <repository Url>

```
Abhimanyu Devadhe@LAPTOP-BCKOBGOF MINGW64 ~/Desktop/Accenture 1
$ git clone https://github.com/Abhimanyudevadhe/accenture.git
Cloning into 'accenture'...
remote: Enumerating objects: 23, done.
remote: Counting objects: 100% (23/23), done.
remote: Compressing objects: 100% (17/17), done.
remote: Total 23 (delta 6), reused 13 (delta 3), pack-reused 0
Receiving objects: 100% (23/23), done.
Resolving deltas: 100% (6/6), done.
```

- After using this command all files from Remote repository will be in your Local Machine.
- **Git Clone branch-**Git allows making a copy of only a particular branch from a repository. You can make a directory for the individual branch by using the git clone command. To make a clone branch, you need to specify the branch name with -b command. Below is the syntax of the command to clone the specific git branch:
  $ git clone -b <branch name><Repository Url>

```
HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/new folder(2) (master)
$ git clone -b master https://github.com/ImDwivedi1/Git-Example.git
Cloning into 'Git-Example'...
remote: Enumerating objects: 9, done.
remote: Counting objects: 100% (9/9), done.
remote: Compressing objects: 100% (6/6), done.
remote: Total 9 (delta 1), reused 0 (delta 0), pack-reused 0
Unpacking objects: 100% (9/9), done.

HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/new folder(2) (master)
$
```

- In the given output, only the master branch is cloned from the principal repository Git-Example.

5.Git Stash-

- Sometimes you want to switch the branches, but you are working on an incomplete part of your current project. You don't want to make a commit of half-done work. Git stashing allows you to do so. The **git stash command** enables you to switch branches without committing the current branch.
- Generally, the stash's meaning is "**store something safely in a hidden place**." The sense in Git is also the same for stash; Git temporarily saves your data safely without committing.

Syntax = $ git stash

6.Git Head-

- The **HEAD** points out the last commit in the current checkout branch. It is like a pointer to any reference. The HEAD can be understood as the "**current branch**." When you switch branches with 'checkout,' the HEAD is transferred to the new branch.
- **Git Head-**The **git show head** is used to check the status of the Head. This command will show the location of the Head.

  Syntax= $ git show head

```
Abhimanyu Devadhe@LAPTOP-BCK0BG0F MINGW64 ~/Desktop/Accenture 1/accenture (main)
$ git show head
commit 46181ff6347315edd745f30a370d5b8609b41b7b (HEAD -> main, origin/main, origir
Author: Abhimanyu Devadhe <abhimanyudevadhe625@gmail.com>
Date:   Fri Feb 11 17:17:49 2022 +0530

    File is added

diff --git a/main.txt b/main.txt
new file mode 100644
index 0000000..e69de29
```

- In the above output, you can see that the commit id for the Head is given. It means the Head is on the given commit.
- Now, check the commit history of the project. You can use the git log command to check the commit history. See the below output:

```
Abhimanyu Devadhe@LAPTOP-BCK0BG0F MINGW64 ~/Desktop/Accenture 1/accenture (main)
$ git log
commit 46181ff6347315edd745f30a370d5b8609b41b7b (HEAD -> main, origin/main, origi
Author: Abhimanyu Devadhe <abhimanyudevadhe625@gmail.com>
Date:   Fri Feb 11 17:17:49 2022 +0530

    File is added
```

- As we can see in the above output, the commit id for most recent commit and Head is the same. So, it is clear that the last commit has the Head.

☐ Undoing Changes-

1.Git Revert-

- In Git, the term revert is used to revert some changes. The git revert command is used to apply revert operation. It is an undo type command. However, it is not a traditional undo alternative. It does not delete any data in this process; instead, it will create a new change with the opposite effect and thereby undo the specified commit. Generally, git revert is a

commit.

- It can be useful for tracking bugs in the project. If you want to remove something from history then git revert is a wrong choice.
- **Git Revert Previous Commit-**Suppose you have made a change to a file say **test.txt** of your project. And later, you remind that you have made a wrong commit in the wrong file or wrong branch. Now, you want to undo the changes you can do so. Git allows you to correct your mistakes.
- I have made changes in test.txt. We can undo it by git revert command. To undo the changes, we will need the commit-ish. To check the commit-ish, run the below command:

$git log

```
HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/GitExample2 (new_branchV1.1)
$ git log
commit 099a8b4c8d92f4e4f1ecb5d52e09906747420814 (HEAD -> new_branchV1.
Author: ImDwivedi1 <himanshudubey481@gmail.com>
Date:   Sat Oct 19 16:54:14 2019 +0530

    new file2 is edited

commit 56afce0ea387ab840819686ec9682bb07d72add6 (tag: -d, tag: --delet
d, tag: projectv1.1, origin/master, testing, master)
Author: ImDwivedi1 <himanshudubey481@gmail.com>
Date:   Wed Oct 9 12:27:43 2019 +0530

    Added an empty newfile2

commit 0d5191fe05e4377abef613d2758ee0dbab7e8d95
Author: ImDwivedi1 <himanshudubey481@gmail.com>
Date:   Sun Oct 6 17:37:09 2019 +0530

    added a new image to prject

commit 828b9628a873091ee26ba53c0fcfc0f2a943c544 (tag: olderversion)
Author: ImDwivedi1 <52317024+ImDwivedi1@users.noreply.github.com>
Date:   Thu Oct 3 11:17:25 2019 +0530
```

- In the above output, I have copied the most recent commit-ish to revert. Now, I will perform the revert operation on this commit. It will operate as:

$ git revert <commit id>

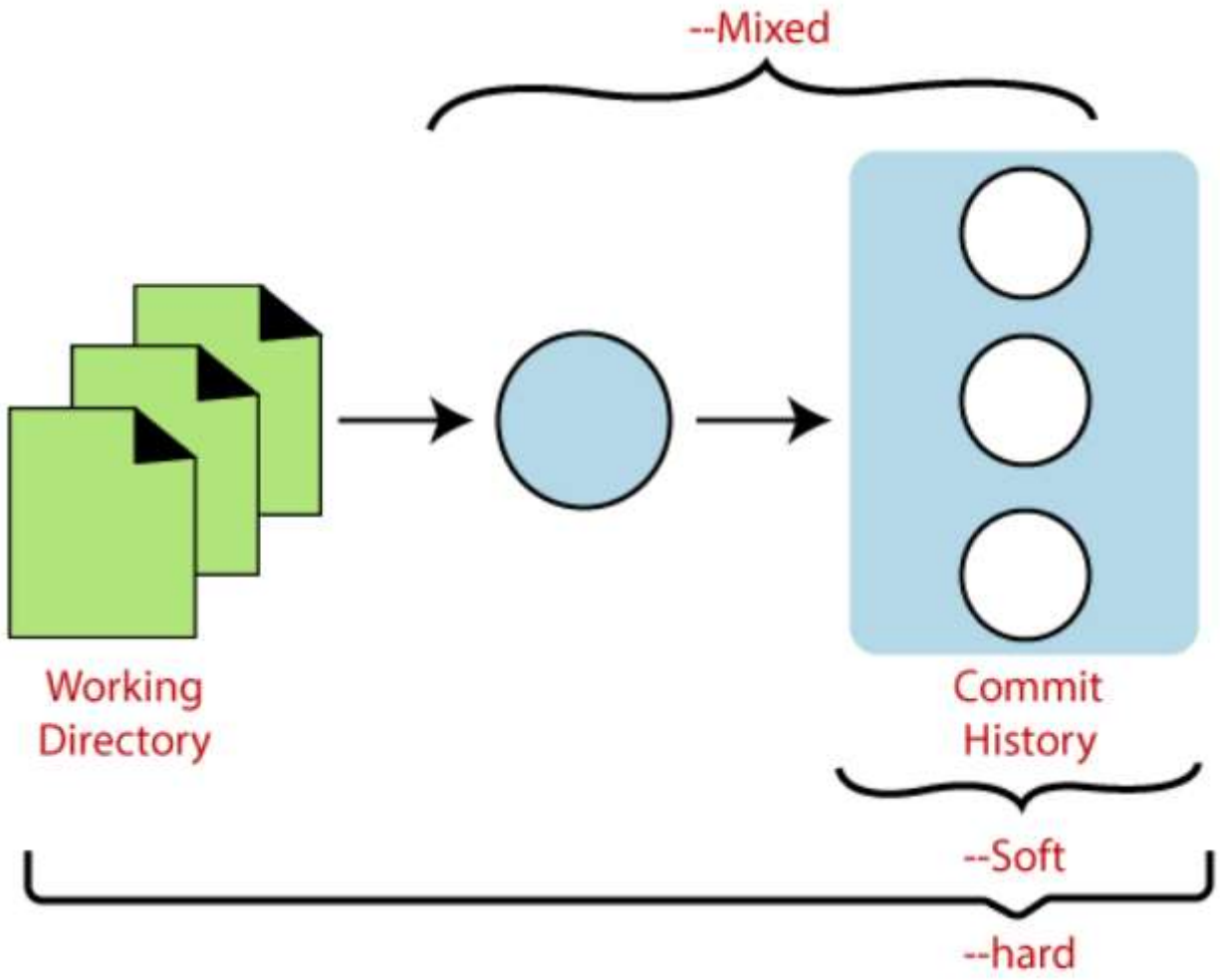
```
HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/GitExample2 (new_branchV1.1)
$ git revert  099a8b4c8d92f4e4f1ecb5d52e09906747420814
[new_branchV1.1 1778984] Revert "new file2 is edited"
 1 file changed, 1 deletion(-)
```

2.Git Reset-

- The term reset stands for undoing changes. The git reset command is used to reset the changes. The git reset command has three core forms of invocation. These forms are as follows.
  - **Soft**

- **Mixed**
- **Hard**

- If we say in terms of Git, then Git is a tool that resets the current state of HEAD to a specified state. It is a sophisticated and versatile tool for undoing changes. It acts as a **time machine for Git**. You can jump up and forth between the various commits. Each of these reset variations affects specific trees that git uses to handle your file in its content.
- Additionally, git reset can operate on whole commits objects or at an individual file level. Each of these reset variations affects specific trees that git uses to handle your file and its contents.



- The working directory lets you change the file, and you can stage into the index. The staging area enables you to select what you want to put into your next commit. A commit object is a cryptographically hashed version of the content. It has some Metadata and points which are used to switch on the previous commits.

A.Git Reset Hard-

- It will first move the Head and update the index with the contents of the commits. It is the most direct, unsafe, and frequently used option. The --hard option changes the Commit History, and ref pointers are updated to the specified commit. Then, the Staging Index and Working Directory need to reset to match that of the specified commit. Any previously pending commits to the Staging Index and the Working Directory gets reset to match Commit Tree. It means any awaiting work will be lost.
- Let's understand the --hard option with an example. Suppose I have added a new file to my existing repository. To add a new file to the repository, run the below command:

```
Abhimanyu Devadhe@LAPTOP-BCK0BG0F MINGW64 ~/Desktop/Accenture 1/accenture (develop5)
$ git add test.py

Abhimanyu Devadhe@LAPTOP-BCK0BG0F MINGW64 ~/Desktop/Accenture 1/accenture (develop5)
$ git status
On branch develop5
Your branch is ahead of 'origin/develop5' by 2 commits.
  (use "git push" to publish your local commits)

Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        new file:   test.py


Abhimanyu Devadhe@LAPTOP-BCK0BG0F MINGW64 ~/Desktop/Accenture 1/accenture (develop5)
$ git log
commit fcc0b35811aaf0400b86e74ea52806a85b9a491f (HEAD -> develop5)
Author: Abhimanyu Devadhe <abhimanyudevadhe625@gmail.com>
Date:   Sat Feb 12 11:24:42 2022 +0530

    Revert "Changes done"
```

- In the above output, I have added a file named **test.py**. I have checked the status of the repository. We can see that the current head position yet not changed because I have not committed the changes. Now, I am going to perform the **reset --hard** option. The git reset hard command will be performed as:

$git reset --hard

```
Abhimanyu Devadhe@LAPTOP-BCK0BG0F MINGW64 ~/Desktop/Accenture 1/accenture (develop5)
$ git reset --hard
HEAD is now at fcc0b35 Revert "Changes done"
```

- As you can see in the above output, the -hard option is operated on the available repository. This option will reset the changes and match the position of the Head before the last changes. It will remove the available changes from the staging area. Consider the below output:

```
Abhimanyu Devadhe@LAPTOP-BCK0BG0F MINGW64 ~/Desktop/Accenture 1/accenture (develop5)
$ git status
On branch develop5
Your branch is ahead of 'origin/develop5' by 2 commits.
  (use "git push" to publish your local commits)

nothing to commit, working tree clean
```

- The above output is displaying the status of the repository after the hard reset. We can see there is nothing to commit in my repository because all the changes removed by the reset hard option to match the status of the current Head with the previous one. So the file **test.py** has been removed from the repository.
- Generally, the reset hard mode performs below operations:

1. It will move the HEAD pointer.
2. It will update the staging Area with the content that the HEAD is pointing.
3. It will update the working directory to match the Staging Area.

2.Git Reset Mixed-

- A mixed option is a default option of the git reset command. If we would not pass any argument, then the git reset command considered as **--mixed** as default option. A mixed option updates the ref pointers. The staging area also reset to the state of a specified commit. The undone changes transferred to the working directory. Let's understand it with an

The undone changes transferred to the working directory. Let's understand it with an example.

- Lets use below commands-

```
Abhimanyu Devadhe@LAPTOP-BCK0BG0F MINGW64 ~/Desktop/Accenture 1/accenture (develop5)
$ touch test1.txt

Abhimanyu Devadhe@LAPTOP-BCK0BG0F MINGW64 ~/Desktop/Accenture 1/accenture (develop5)
$ git status
On branch develop5
Your branch is ahead of 'origin/develop5' by 2 commits.
  (use "git push" to publish your local commits)

Untracked files:
  (use "git add <file>..." to include in what will be committed)
        test1.txt

nothing added to commit but untracked files present (use "git add" to track)

Abhimanyu Devadhe@LAPTOP-BCK0BG0F MINGW64 ~/Desktop/Accenture 1/accenture (develop5)
$ git add test1.txt
```

- In the above output, I have added a **test1.txt** to my local repository. Now, we will perform the reset mixed command on this repository. It will operate as:

$git reset --mixed or git reset

- The above command will reset the status of the Head, and it will not delete any data from the staging area to match the position of the Head. Consider the below output:

```
Abhimanyu Devadhe@LAPTOP-BCK0BG0F MINGW64 ~/Desktop/Accenture 1/accenture (develop5)
$ git reset --mixed

Abhimanyu Devadhe@LAPTOP-BCK0BG0F MINGW64 ~/Desktop/Accenture 1/accenture (develop5)
$ git status
On branch develop5
Your branch is ahead of 'origin/develop5' by 2 commits.
  (use "git push" to publish your local commits)

Untracked files:
  (use "git add <file>..." to include in what will be committed)
        test1.txt

nothing added to commit but untracked files present (use "git add" to track)
```

- From the above output, we can see that we have reset the position of the Head by performing the git reset -mixed command. Also, we have checked the status of the repository. As we can see that the status of the repository has not been changed by this command. So it is clear that the mixed-mode does not clear any data from the staging area. Generally, the reset mixed mode performs the below operations:
  - It will move the HEAD pointer
  - It will update the Staging Area with the content that the HEAD is pointing to.
- It will not update the working directory as git hard mode does. It will only reset the index but not the working tree, then it generates the report of the files which have not been updated.

3.Git Reset Soft-

- The soft option does not touch the index file or working tree at all, but it resets the Head as all options do. When the soft mode runs, the refs pointers updated, and the resets stop there. It will act as git amend command. It is not an authoritative command. Sometimes developers considered it as a waste of time.
- Generally, it is used to change the position of the Head. Let's understand how it will change

the position of the Head. It will use as:

```
Abhimanyu Devadhe@LAPTOP-BCK0BG0F MINGW64 ~/Desktop/Accenture 1/accenture (develop5)
$ git commit -m"test1.txt added"
[develop5 f1d2b75] test1.txt added
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 test1.txt

Abhimanyu Devadhe@LAPTOP-BCK0BG0F MINGW64 ~/Desktop/Accenture 1/accenture (develop5)
$ git reset --soft

Abhimanyu Devadhe@LAPTOP-BCK0BG0F MINGW64 ~/Desktop/Accenture 1/accenture (develop5)
$ git log
commit f1d2b75b03f58a359b7b859e92a86d178528adf7 (HEAD -> develop5)
Author: Abhimanyu Devadhe <abhimanyudevadhe625@gmail.com>
Date:   Sat Feb 12 16:44:48 2022 +0530

    test1.txt added

commit fcc0b35811aaf0400b86e74ea52806a85b9a491f
Author: Abhimanyu Devadhe <abhimanyudevadhe625@gmail.com>
Date:   Sat Feb 12 11:24:42 2022 +0530

    Revert "Changes done"

    This reverts commit 6a1dc8f2ecc7f881d976b908f9d8772dd305fce1.
```

- From the above output, you can see that the current position of the HEAD is on f1d2b75b03f58a359b7b859e92a86d178528adf7 commit. But, I want to switch it on my older commit fcc0b35811aaf0400b86e74ea52806a85b9a491f. Since the commit-sha number is a unique number that is provided by sha algorithm. To switch the HEAD, run the below command:

$ git reset --soft fcc0b35811aaf0400b86e74ea52806a85b9a491f

```
Abhimanyu Devadhe@LAPTOP-BCK0BG0F MINGW64 ~/Desktop/Accenture 1/accenture (develop5)
$ git reset --soft fcc0b35811aaf0400b86e74ea52806a85b9a491f

Abhimanyu Devadhe@LAPTOP-BCK0BG0F MINGW64 ~/Desktop/Accenture 1/accenture (develop5)
$ git log
commit fcc0b35811aaf0400b86e74ea52806a85b9a491f (HEAD -> develop5)
Author: Abhimanyu Devadhe <abhimanyudevadhe625@gmail.com>
Date:   Sat Feb 12 11:24:42 2022 +0530

    Revert "Changes done"
```

- As you can see from the above output, the HEAD has been shifted to a particular commit by git reset --soft mode.

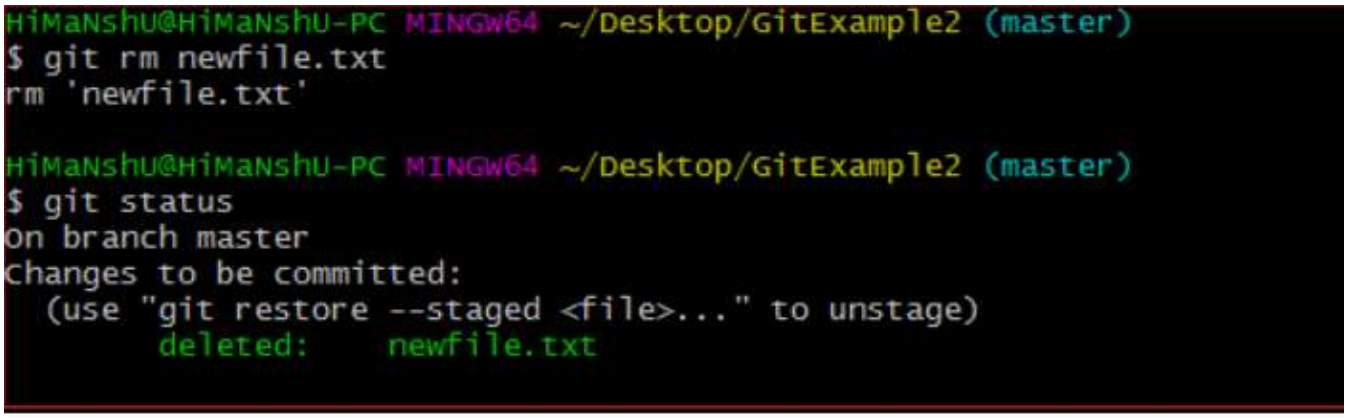
4.Git Rm-

- In Git, the term rm stands for remove. It is used to remove individual files or a collection of files. The key function of git rm is to remove tracked files from the Git index. Additionally, it can be used to remove files from both the working directory and staging index.
- The Git Rm Command-is used to remove the files from the working tree and the index.
- If we want to remove the file from our repository. Then it can be done by the git rm command. Let's take a file say newfile.txt to test the rm command. The git rm command will be operated as:

$git rm <filename>

- The above command will remove the file from the Git and repository. The git rm command

removes the file not only from the repository but also from the staging area. If we check the status of the repository, then it will show as deleted. Consider the below output:



- In the above output, the file **newfile.txt** has been removed from the version control system. So the repository and the status are shown as deleted.

Inspecting Changes-

1.Git Diff-

- Git diff is a command-line utility. It's a multiuse Git command. When it is executed, it runs a diff function on Git data sources. These data sources can be files, branches, commits, and more. It is used to show changes between commits, commit, and working tree, etc.
- It compares the different versions of data sources. The version control system stands for working with a modified version of files. So, the diff command is a useful tool for working with Git.
- Let's understand different scenarios where we can utilize the git diff command.

Scenerio1: Track the changes that have not been staged.

- The usual use of git diff command that we can track the changes that have not been staged.
- Suppose we have edited the test.txt file. Now, we want to track what changes are not staged yet. Then we can do so from the git diff command. Consider the below output:

- From the above output, we can see that the changes made on test.txt are displayed by git diff command. As we have edited it as "Hello Abhimanyu Devadhe." So, the output is displaying the changes with its content.

Scenerio2: Track the changes that have staged but not committed:

- The git diff command allows us to track the changes that are staged but not committed. We can track the changes in the staging area. To check the already staged changes, use the --staged option along with git diff command.
- To check the untracked file, run the git status command as:
- use below command-

```
Abhimanyu Devadhe@LAPTOP-BCK0BGOF MINGW64 ~/Desktop/Accenture 1/accenture (develop5)
$ git status
On branch develop5
Your branch is ahead of 'origin/develop5' by 3 commits.
  (use "git push" to publish your local commits)

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   test.txt

no changes added to commit (use "git add" and/or "git commit -a")

Abhimanyu Devadhe@LAPTOP-BCK0BGOF MINGW64 ~/Desktop/Accenture 1/accenture (develop5)
$ git add test.txt
```

- Now, the file is added to the staging area, but it is not committed yet. So, we can track the changes in the staging area also. To check the staged changes, run the git diff command along with **--staged** option. It will be used as:

$git diff --staged

- The above command will display the changes of already staged files. Consider the below output:

```
Abhimanyu Devadhe@LAPTOP-BCK0BGOF MINGW64 ~/Desktop/Accenture 1/accenture (develop5)
$ git diff --staged
diff --git a/test.txt b/test.txt
index e69de29..5a3e68b 100644
--- a/test.txt
+++ b/test.txt
@@ -0,0 +1 @@
+Hello Abhimanyu Devadhe
\ No newline at end of file
```

- The given output is displaying the changes of test.txt, which is already staged.

Scenerio3: Track the changes after committing a file:

- Git, let us track the changes after committing a file. Suppose we have committed a file for the repository and made some additional changes after the commit. So we can track the file on this stage also.
- In the below output, we have committed the changes that we made on our newfile1.txt. Consider the below output:

```
Abhimanyu Devadhe@LAPTOP-BCK0BGOF MINGW64 ~/Desktop/Accenture 1/accenture (develop5)
$ git commit -m"Changes Done with test1.txt"
[develop5 766c04e] Changes Done with test1.txt
 1 file changed, 1 insertion(+)
```

- Now, we have changed the test1.txt file again as "Changes are made after committing the file." To track the changes of this file, run the git diff command with **HEAD** argument. It will run as follows:

$git diff head

Scenerio4: Track the changes between two branches:

- Git allows comparing the branches. If you are a master in branching, then you can understand the importance of analyzing the branches before merging. Many conflicts can arise if you merge the branch without comparing it. So to avoid these conflicts, Git allows many handy commands to preview, compare, and edit the changes.

- The git diff command allows us to compare different versions of branches and repository. To get the difference between branches, run the git diff command as follows:

git diff <branch1> <branch2>

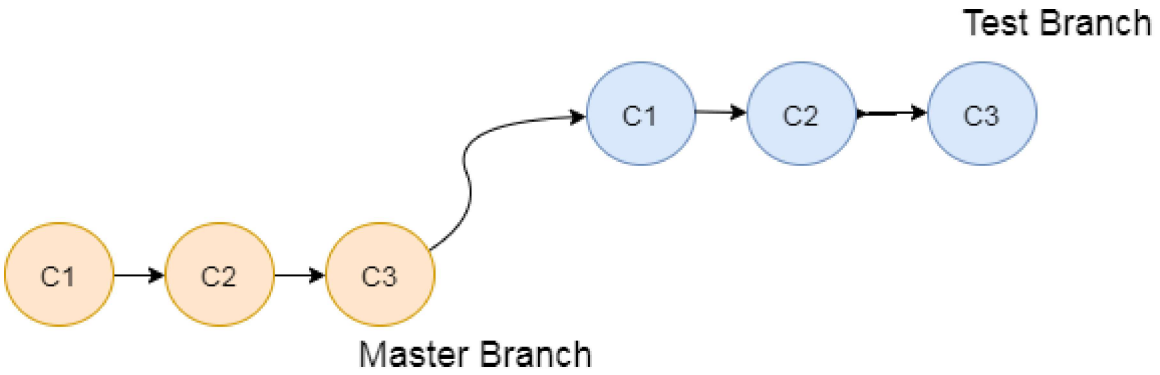
- The above command will display the differences between branch 1 and branch 2. So that you can decide whether you want to merge the branch or not. Consider the below output:

```
Abhimanyu Devadhe@LAPTOP-BCK0BG0F MINGW64 ~/Desktop/Accenture 1/accenture (develop5)
$ git diff main develop5
diff --git a/test.txt b/test.txt
new file mode 100644
index 0000000..5a3e68b
--- /dev/null
+++ b/test.txt
@@ -0,0 +1 @@
+Hello Abhimanyu Devadhe
\ No newline at end of file
diff --git a/test1.txt b/test1.txt
new file mode 100644
index 0000000..e69de29
```

- The above output is displaying the differences between my repository branches **develop5** and **main**. The git diff command is giving a preview of both branches. So, it will be helpful to perform any operation on branches.

- [ ] **Git Rebase-**

- Rebasing is a process to reapply commits on top of another base trip. It is used to apply a sequence of commits from distinct branches into a final commit. It is an alternative of git merge command. It is a linear process of merging.
- In Git, the term rebase is referred to as the process of moving or combining a sequence of commits to a new base commit. Rebasing is very beneficial and it visualized the process in the environment of a feature branching workflow.
- It is good to rebase your branch before merging it.
- Generally, it is an alternative of git merge command. Merge is always a forward changing record. Comparatively, rebase is a compelling history rewriting tool in git. It merges the different commits one by one.
- Suppose you have made three commits in your master branch and three in your other branch named test. If you merge this, then it will merge all commits in a time. But if you rebase it, then it will be merged in a linear manner. Consider the below image:



- The above image describes how git rebase works. The three commits of the master branch are merged linearly with the commits of the test branch.
- Merging is the most straightforward way to integrate the branches. It performs a three-way merge between the two latest branch commits.

- [ ] **How to Rebase branches..??**

- When you made some commits on a feature branch (develop5 branch) and some in the master branch. You can rebase any of these branches. Use the git log command to track the changes (commit history).
- If we have many commits from distinct branches and want to merge it in one. To do so, we have two choices either we can merge it or rebase it. It is good to rebase your branch.
- Let's see the below commands:

$git checkout develop5

- This command will switch you on the test2 branch from the master.
- You are on the Develop5 branch. Hence, you can rebase the Develop5 branch with the master branch. See the below command:

$git rebase master

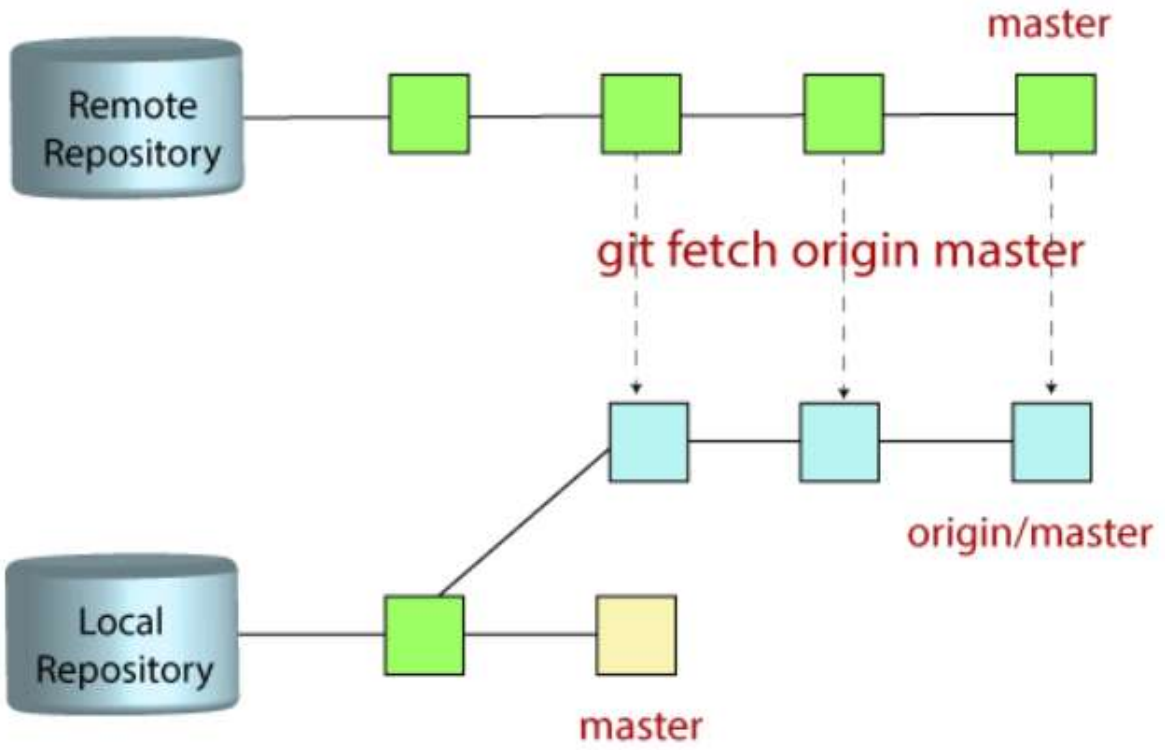
- This command will rebase the develop5 branch and will show as **Applying: new commit on develop5 branch**. Consider the below output:

```
HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/GitExample2 (test)
$ git rebase master
First, rewinding head to replay your work on top of it...
Fast-forwarded test to master.

HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/GitExample2 (test)
$
```

- [ ] **Git Fetch-**

- Git "fetch" Downloads commits, objects and refs from another repository. It fetches branches and tags from one or more repositories. It holds repositories along with the objects that are necessary to complete their histories to keep updated remote-tracking branches.



- **The "git fetch" command**-The "**git fetch**" **command** is used to pull the updates from remote-tracking branches. Additionally, we can get the updates that have been pushed to our remote branches to our local machines. As we know, a branch is a variation of our repositories main code, so the remote-tracking branches are branches that have been set up to pull and push from remote repository.

Scenario : To fetch the remote repository:

- We can fetch the complete repository with the help of fetch command from a repository URL like a pull command does. See the below output:

$ git fetch <repository url>

```
Abhimanyu Devadhe@LAPTOP-BCK0BG0F MINGW64 ~/Desktop/Accenture 1/accenture (main)
$ git fetch https://github.com/Abhimanyudevadhe/Pyhton-Code.git
remote: Enumerating objects: 7, done.
remote: Counting objects: 100% (7/7), done.
remote: Compressing objects: 100% (6/6), done.
remote: Total 7 (delta 1), reused 0 (delta 0), pack-reused 0
Unpacking objects: 100% (7/7), 3.06 KiB | 10.00 KiB/s, done.
From https://github.com/Abhimanyudevadhe/Pyhton-Code
 * branch            HEAD       -> FETCH_HEAD
```

- In the above output, the complete repository has fetched from a remote URL.

- [ ] **Difference Between Git Fetch and Git Pull-**

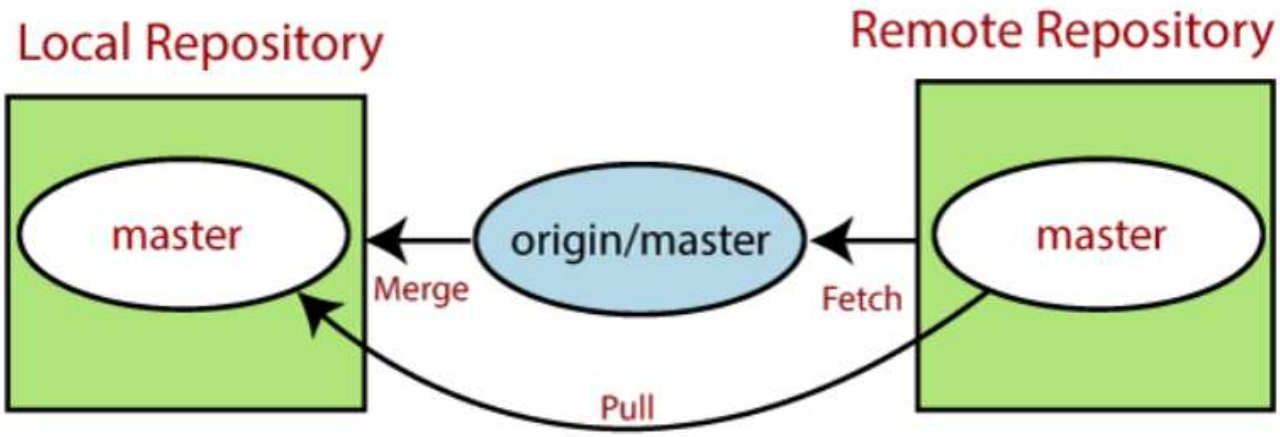
- To understand the differences between fetch and pull, let's know the similarities between both of these commands. Both commands are used to download the data from a remote repository. But both of these commands work differently. Like when you do a git pull, it gets all the changes from the remote or central repository and makes it available to your corresponding branch in your local repository. When you do a git fetch, it fetches all the changes from the remote repository and stores it in a separate branch in your local repository. You can reflect those changes in your corresponding branches by merging.So basically,

git pull =git fetch +git merge

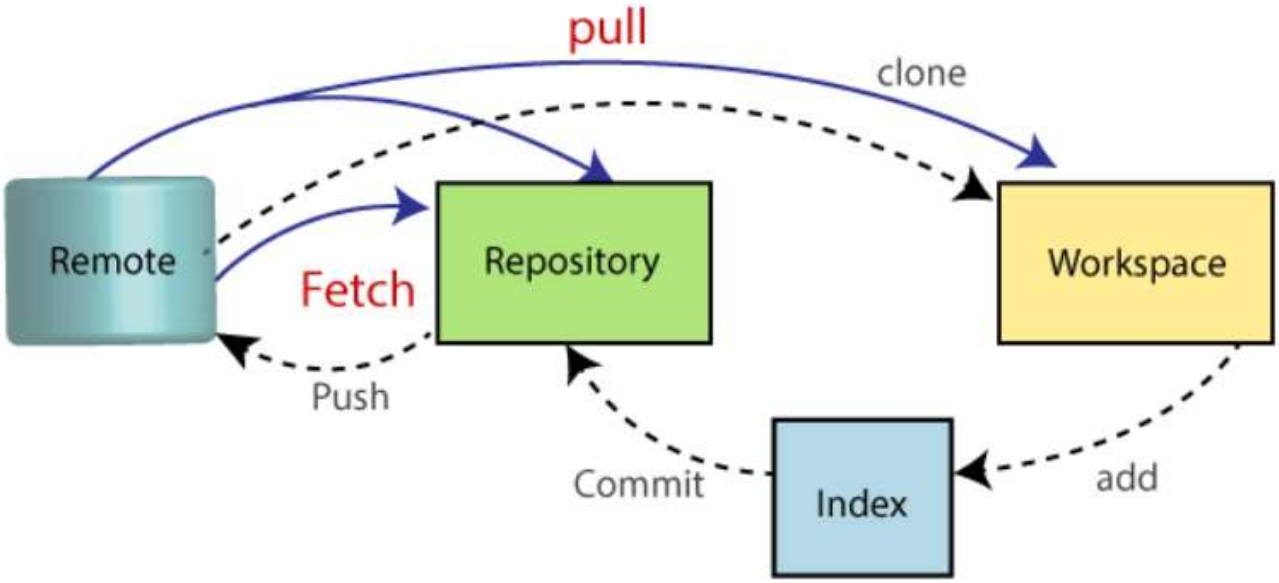
git fetch	git pull
Fetch downloads only new data from a remote repository.	Pull is used to update your current HEAD branch with the latest changes from the remote server.
Fetch is used to get a new view of all the things that happened in a remote repository.	Pull downloads new data and directly integrates it into your current working copy files.
Fetch never manipulates or spoils data.	Pull downloads the data and integrates it with the current working file.
It protects your code from merge conflict.	In git pull, there are more chances to create the **merge conflict**.
It is better to use git fetch command with git merge command on a pulled repository.	It is not an excellent choice to use git pull if you already pulled any repository.

- [ ] **Git Pull/Pull Request-**

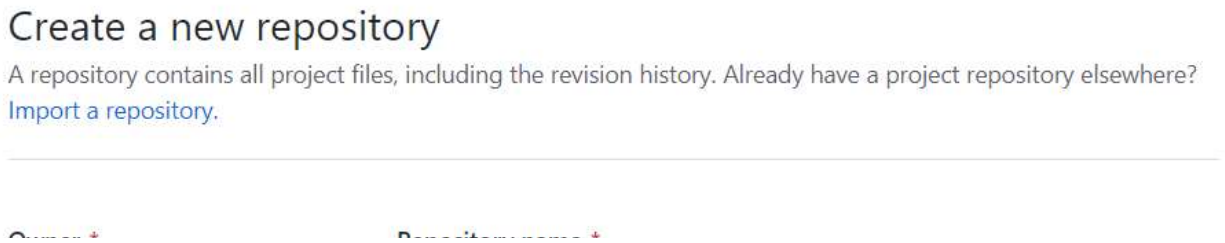
- The term pull is used to receive data from GitHub. It fetches and merges changes from the remote server to your working directory. The **git pull command** is used to pull a repository.



- Pull request is a process for a developer to notify team members that they have completed a feature. Once their feature branch is ready, the developer files a pull request via their remote server account. Pull request announces all the team members that they need to review the code and merge it into the master branch.
- The below figure demonstrates how pull acts between different locations and how it is similar or dissimilar to other related commands.



- **Create new Repository as below-**



Owner * Repository name *

Abhimanyudevadhe / Python12345

Great repository names are short and memorable. Need inspiration? How about vigilant-octo-carnival?

Description (optional)

This repo is belongs to Abhimanyu Devadhe

Public
Anyone on the internet can see this repository. You choose who can commit.

Private
You choose who can see and commit to this repository.

Initialize this repository with:
Skip this step if you're importing an existing repository.

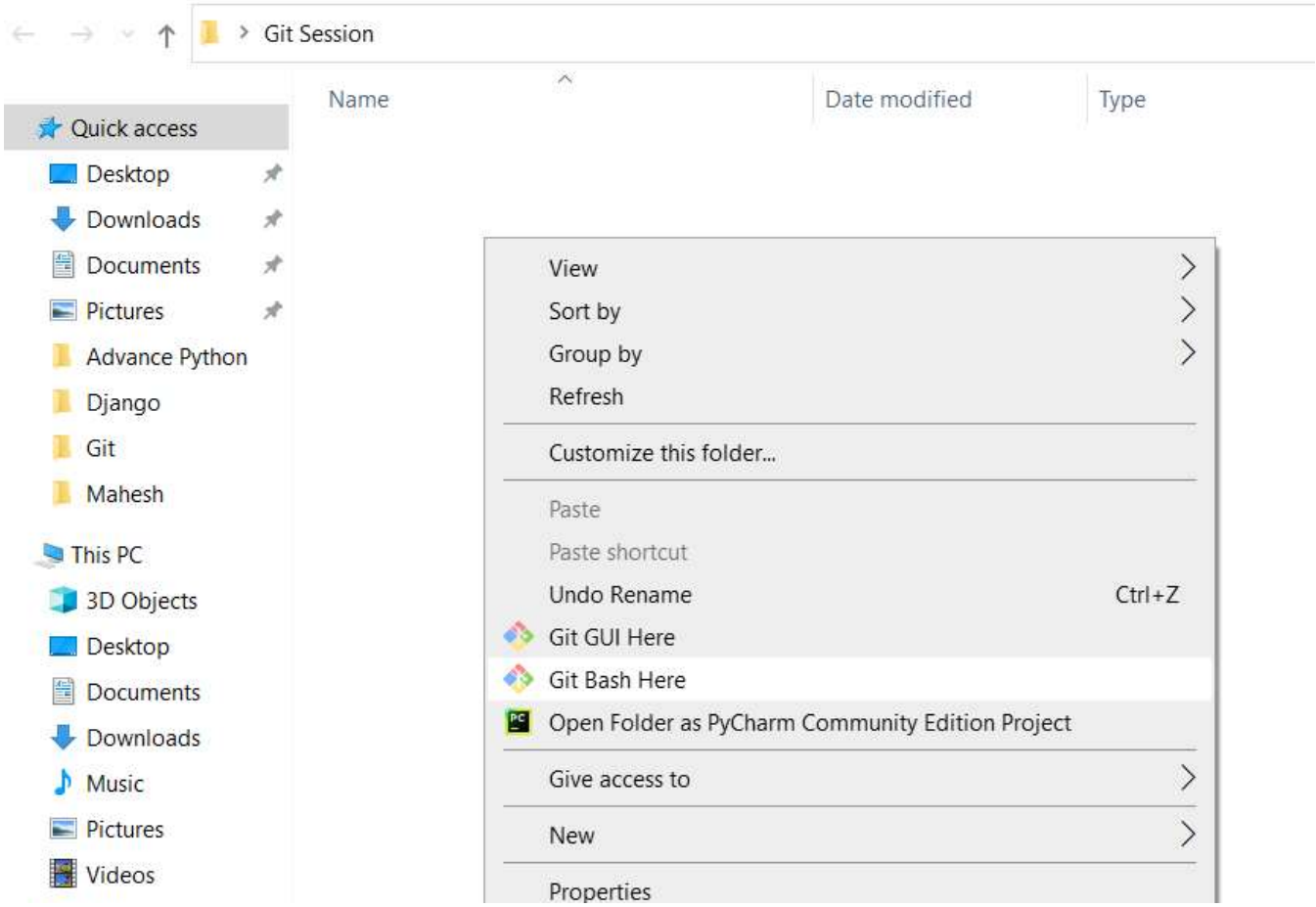
Add a README file
This is where you can write a long description for your project. Learn more.

Add .gitignore
Choose which files not to track from a list of templates. Learn more.

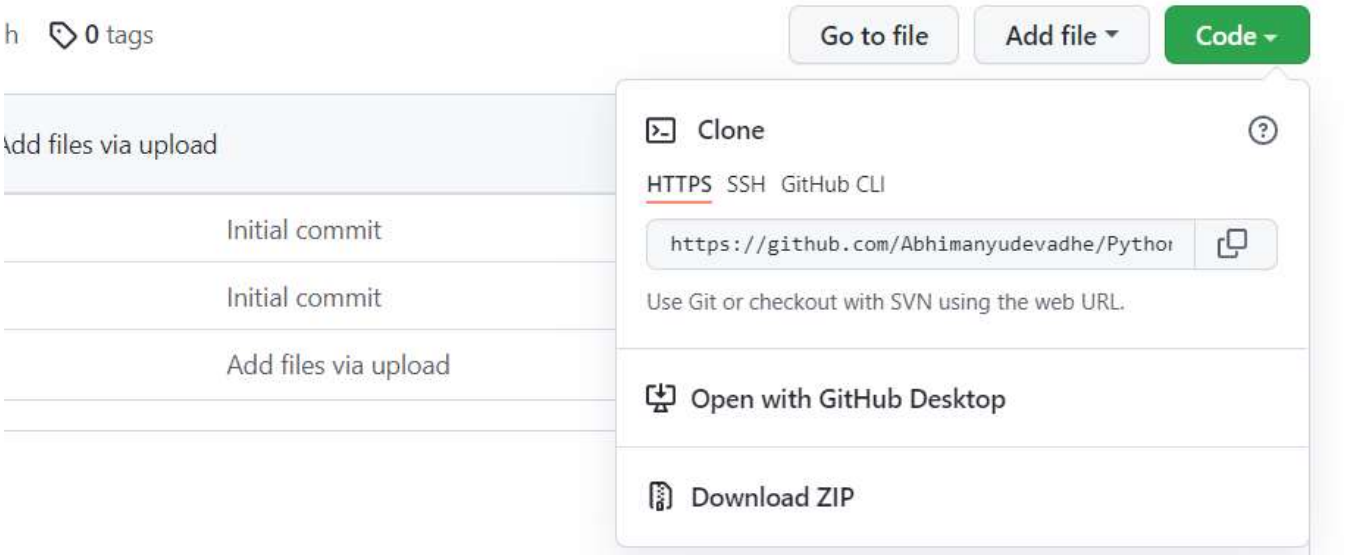
Choose a license
A license tells others what they can and can't do with your code. Learn more.

License: MIT License

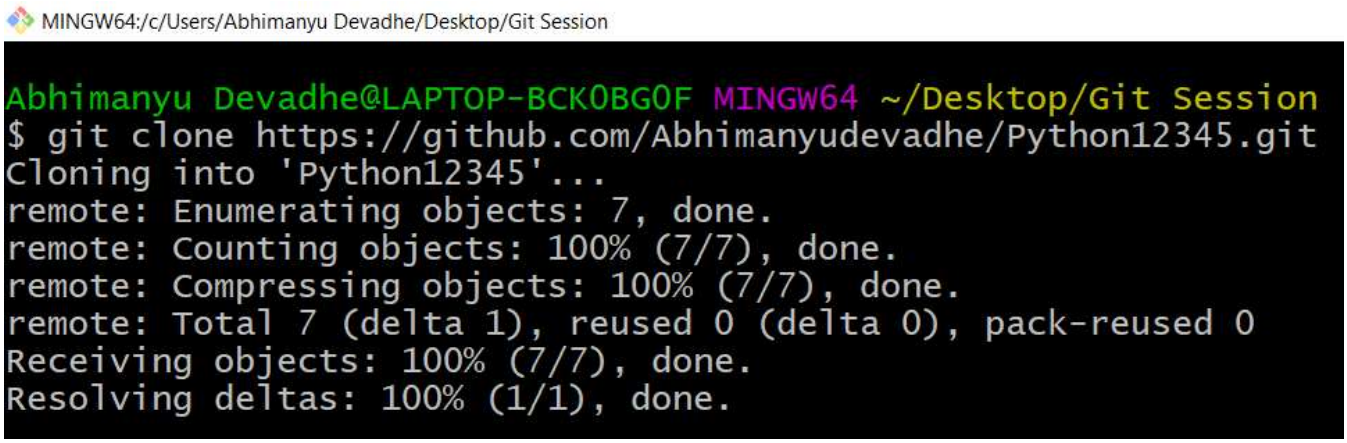
- **Upload New files to Repository-**
- **Create Local Repository as new created folder at desktop and Open With Git bash-**



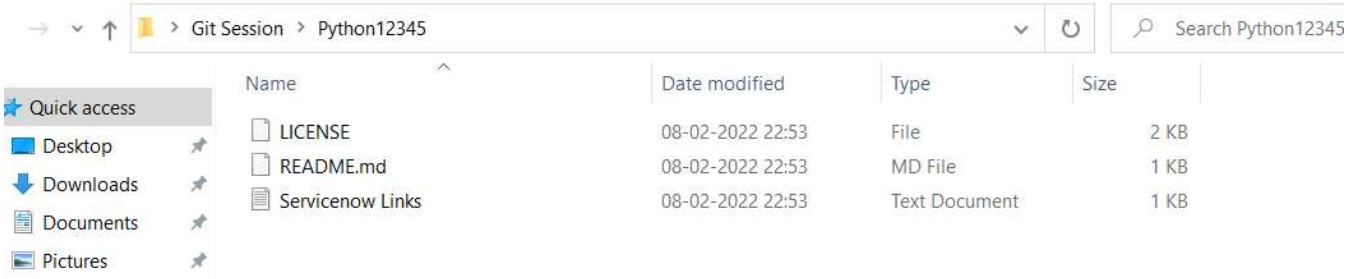
- **Create Clone copy of Project-**
- **Copy link from copy-->HTTPS**



- Use command git clone 'copied url'-



- **Clone copy will like as below-**



- **Check file modified status-**

1. Change working directory as python12345
2. Modify file.

```
Abhimanyu Devadhe@LAPTOP-BCK0BG0F MINGW64 ~/Desktop/Git Session/python12345 (main)
$
```

3.Check status using command - git status

```
Abhimanyu Devadhe@LAPTOP-BCK0BG0F MINGW64 ~/Desktop/Git Session/python12345 (main)
$ git status
On branch main
Your branch is up to date with 'origin/main'.

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   Servicenow Links.txt

no changes added to commit (use "git add" and/or "git commit -a")
```

- [ ] **Move modified file in Staging area-**
  - Use command - git add 'file name'
  - Use backward slash.
  - For Multiple file- git add filename1 filename2
  - For All files - git add .

```
Abhimanyu Devadhe@LAPTOP-BCK0BG0F MINGW64 ~/Desktop/Git Session/python12345 (main)
$ git add Servicenow\ Links.txt

Abhimanyu Devadhe@LAPTOP-BCK0BG0F MINGW64 ~/Desktop/Git Session/python12345 (main)
$
```

  - Now check git status- Filename now reflected in green colour i.e modified file is now added in staging area.

```
Abhimanyu Devadhe@LAPTOP-BCK0BG0F MINGW64 ~/Desktop/Git Session/python12345 (main)
$ git status
On branch main
Your branch is up to date with 'origin/main'.

Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        modified:   Servicenow Links.txt
```

- [ ] **Commit files from Staging Area to move into Local Repo-**
  - Use Commit command for move files from stagging area to Local repo-

```
Abhimanyu Devadhe@LAPTOP-BCK0BG0F MINGW64 ~/Desktop/Git Session/python12345 (main)
$ git commit -m "Changes done in servicenow Links"
[main 6c6b05f] Changes done in servicenow Links
 1 file changed, 1 insertion(+), 9 deletions(-)
```

- [ ] **Push Files from Local Repo to Remote Repo-**
  - Use command git push personal Access token
  - Create Personal Access Token from GitHub-

settings-->Devloper Settings-->Personal Access Settings-->Generate new Token

- Use token as password-

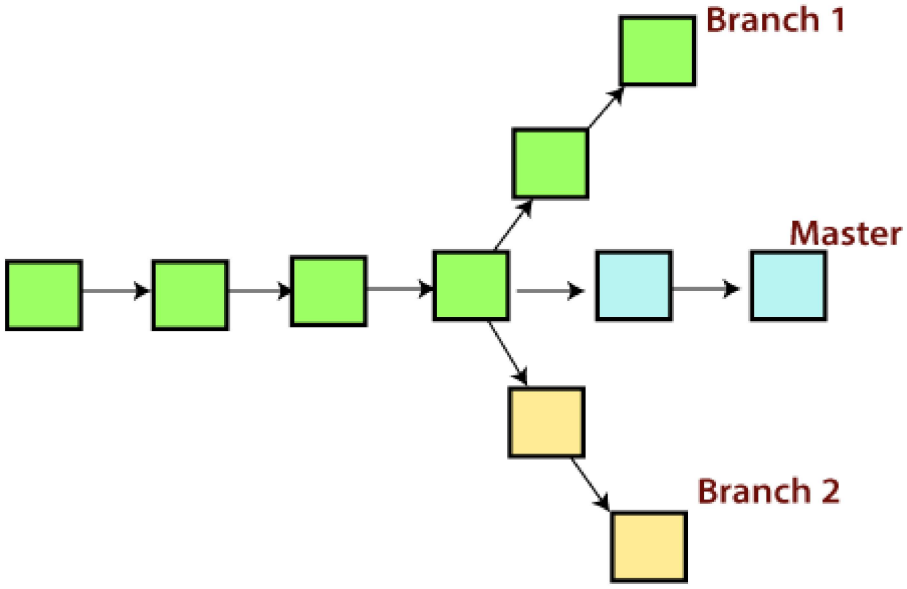
```
Abhimanyu Devadhe@LAPTOP-BCK0BG0F MINGW64 ~/Desktop/Git Session/python12345 (main)
$ git push
Enumerating objects: 5, done.
Counting objects: 100% (5/5), done.
Delta compression using up to 8 threads
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 338 bytes | 338.00 KiB/s, done.
Total 3 (delta 2), reused 0 (delta 0), pack-reused 0
remote: Resolving deltas: 100% (2/2), completed with 2 local objects.
To https://github.com/Abhimanyudevadhe/Python12345.git
   3027038..6c6b05f  main -> main
```

- [ ] **What is Difference Between Git clone and Git Pull method-..??**

- Git Clone- Git clone is used for create local Repo, It is an one time activity.
- Git Pull- Git Pull is used for pull updated changes to Local repository.

- [ ] **Git Branch-**

- A branch is a version of the repository that diverges from the main working project. It is a feature available in most modern version control systems. A Git project can have more than one branch. These branches are a pointer to a snapshot of your changes.
- When you want to add a new feature or fix a bug, you spawn a new branch to summarize your changes. So, it is complex to merge the unstable code with the main code base and also facilitates you to clean up your future history before merging with the main branch.



- **Git Master Branch-**

- The master branch is a default branch in Git. It is instantiated when first commit made on the project. When you make the first commit, you're given a master branch to the starting commit point. When you start making a commit, then master branch pointer automatically moves forward. A repository can have only one master branch.
- Master branch is the branch in which all the changes eventually get merged back. *It can be called as an official working version of your project.*

- [ ] **Operations On Branch-**

- We can perform various operations on Git branches. The **git branch command** allows you to **create**, **list**, **rename** and **delete** branches. Many operations on branches are applied by git checkout and git merge command. So, the git branch is tightly integrated with the **git checkout** and **git merge commands**.

1.Create Branch-

- Use command= git branch <branch name>

```
Abhimanyu Devadhe@LAPTOP-BCK0BG0F MINGW64 ~/Desktop/Accenture/accenture (main)
$ git branch Develop

Abhimanyu Devadhe@LAPTOP-BCK0BG0F MINGW64 ~/Desktop/Accenture/accenture (main)
$ git
```

- This command will create the **branch Develop** locally in Git directory.

2.List Branch-

- You can List all of the available branches in your repository by using the following command.
- Either we can use **git branch - list** or **git branch** command to list the available branches in the repository.

```
Abhimanyu Devadhe@LAPTOP-BCK0BG0F MINGW64 ~/Desktop/Accenture/accenture (main)
$ git branch --list
  Develop
* main

Abhimanyu Devadhe@LAPTOP-BCK0BG0F MINGW64 ~/Desktop/Accenture/accenture (main)
$ git branch
  Develop
* main
```

- Here, both commands are listing the available branches in the repository*. The symbol * is representing currently active branch.*

3.Delete Branch-

- You can delete the specified branch. It is a safe operation. In this command, Git prevents you from deleting the branch if it has unmerged changes. Below is the command to do this.

```
Abhimanyu Devadhe@LAPTOP-BCK0BG0F MINGW64 ~/Desktop/Accenture/accenture (main)
$ git branch -d develop
Deleted branch develop (was e146804).
```

- This command will delete the existing branch B1 from the repository.
- The **git branch d** command can be used in two formats. Another format of this command is **git branch D**. The '**git branch D**' command is used to delete the specified branch.

4.Delete A Remote Branch-

- You can delete a remote branch from Git desktop application. Below command is used to delete a remote branch:
- **Syntax:** $ git push origin -delete **<branch** name**>**

```
HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/GitExample2 (master)
$ git push origin --delete branch2
To https://github.com/ImDwivedi1/GitExample2
```

```
 - [deleted]         branch2

HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/GitExample2 (master)
$
```

- As you can see in the above output, the remote branch named **branch2** from GitHub account is deleted.

5.Switch Branch-

- Git allows you to switch between the branches without making a commit. You can switch between two branches with the **git checkout** command. To switch between the branches, below command is used:

  $ git checkout**<branch** name**>**

```
Abhimanyu Devadhe@LAPTOP-BCK0BG0F MINGW64 ~/Desktop/Accenture/accenture (main)
$ git checkout develop2
Switched to branch 'develop2'
```

- Switch to master branch-You can switch to the master branch from any other branch with the help of below command.

  $ git branch -m master

```
Abhimanyu Devadhe@LAPTOP-BCK0BG0F MINGW64 ~/Desktop/Accenture/accenture (develop2)
$ git branch -m master

Abhimanyu Devadhe@LAPTOP-BCK0BG0F MINGW64 ~/Desktop/Accenture/accenture (master)
$
```

- As you can see in the above output, branches are switched from **develop2 to master** without making any commit.

6.Rename Branch-

- We can rename the branch with the help of the **git branch** command. To rename a branch, use the below command:

$ git branch -m **<old** branch name**><new** branch name**>**

```
Abhimanyu Devadhe@LAPTOP-BCK0BG0F MINGW64 ~/Desktop/Accenture/accenture (master)
$ git branch -m develop3 develop4

Abhimanyu Devadhe@LAPTOP-BCK0BG0F MINGW64 ~/Desktop/Accenture/accenture (master)
$ git branch
  develop4
  main
* master
```

- As you can see in the above output, **Develop3** renamed as **Develop4**.

7.Merge Branch-

- Git allows you to merge the other branch with the currently active branch. You can merge two branches with the help of **git merge** command. Below command is used to merge the branches:

$ git merge **<branch** name**>**

```
Abhimanyu Devadhe@LAPTOP-BCKOBGOF MINGW64 ~/Desktop/Accenture/accenture (master)
$ git merge develop4
Already up to date.
```

- From the above output, you can see that **the master** branch **merged** with **develop4**. Since I have made no-commit before merging, so the output is showing as already up to date.
- *If you have make changes in Develop4 and you have to add these changes in main branch,then checkout in main branch and then use this command git merge develop4.*

- [ ] **Git Cheatsheet-**

1. Git configuration

- **Git config**

  Get and set configuration variables that control all facets of how Git looks and operates.

  Set the name:

  $ git config --global user.name "User name"

  Set the email:

  $ git config --global user.email "himanshudubey481@gmail.com"

  Set the default editor:

  $ git config --global core.editor Vim

  Check the setting:

  $ git config -list

- **Git alias**

  Set up an alias for each command:

  $ git config --global alias.co checkout

  $ git config --global alias.br branch

  $ git config --global alias.ci commit

  $ git config --global alias.st status

2. Starting a project

- **Git init**

  Create a local repository:

  $ git init

- **Git clone**

  Make a local copy of the server repository.

  $ git clone

3. Local changes

- **Git add**

  Add a file to staging (Index) area:

  $ git add Filename

  Add all files of a repo to staging (Index) area:

  $ git add*

- **Git commit**

  Record or snapshots the file permanently in the version history **with a message**.

  $ git commit -m " Commit Message"

4. Track changes

- **Git diff**

Git diff

Track the changes that have not been staged: $ git diff

Track the changes that have staged but not committed:

$ git diff --staged

Track the changes after committing a file:

$ git diff HEAD

Track the changes between two commits:

$ git diff Git Diff Branches:

$ git diff < branch 2>

- **Git status**

  Display the state of the working directory and the staging area.

  $ git status

- **Git show Shows objects:**

  $ git show

5. Commit History

- **Git log**

  Display the most recent commits and the status of the head:

  $ git log

  Display the output as one commit per line:

  $ git log -oneline

  Displays the files that have been modified:

  $ git log -stat

  Display the modified files with location:

  $ git log -p

- **Git blame**

  Display the modification on each line of a file:

  $ git blame <file name>

6. Ignoring files

- **.gitignore**

  Specify intentionally untracked files that Git should ignore. Create .gitignore:

  $ touch .gitignore List the ignored files:

  $ git ls-files -i --exclude-standard

7. Branching

- **Git branch Create branch:**

  $ git branch List Branch:

  $ git branch --list Delete a Branch:

  $ git branch -d Delete a remote Branch:

  $ git push origin -delete Rename Branch:

  $ git branch -m

- **Git checkout**

  Switch between branches in a repository.

  Switch to a particular branch:

  $ git checkout

  Create a new branch and switch to it:

$ git checkout -b Checkout a Remote branch:
$ git checkout

- **Git stash**
  Switch branches without committing the current branch. Stash current work:
  $ git stash
  Saving stashes with a message:
  $ git stash save ""
  Check the stored stashes:
  $ git stash list
  Re-apply the changes that you just stashed:
  $ git stash apply
  Track the stashes and their changes:
  $ git stash show
  Re-apply the previous commits:
  $ git stash pop
  Delete a most recent stash from the queue:
  $ git stash drop
  Delete all the available stashes at once:
  $ git stash clear
  Stash work on a separate branch:
  $ git stash branch
- **Git cherry pic**
  Apply the changes introduced by some existing commit:
  $ git cherry-pick

8. Merging

- **Git merge**
  Merge the branches:
  $ git merge
  Merge the specified commit to currently active branch:
  $ git merge
- **Git rebase**
  Apply a sequence of commits from distinct branches into a final commit.
  $ git rebase
  Continue the rebasing process:
  $ git rebase -continue Abort the rebasing process:
  $ git rebase --skip
- **Git interactive rebase**
  Allow various operations like edit, rewrite, reorder, and more on existing commits.
  $ git rebase -i

9. Remote

- **Git remote**
  Check the configuration of the remote server:
  $ git remote -v
  Add a remote for the repository:

$ git remote add Fetch the data from the remote server:
$ git fetch
Remove a remote connection from the repository:
$ git remote rm
Rename remote server:
$ git remote rename
Show additional information about a particular remote:
$ git remote show
Change remote:
$ git remote set-url

- **Git origin master**
  Push data to the remote server:
  $ git push origin master Pull data from remote server:
  $ git pull origin master

10. Pushing Updates

- **Git push**
  Transfer the commits from your local repository to a remote server. Push data to the remote server:
  $ git push origin master Force push data:
  $ git push -f
  Delete a remote branch by push command:
  $ git push origin -delete edited

11. Pulling updates

- **Git pull**
  Pull the data from the server:
  $ git pull origin master
  Pull a remote branch:
  $ git pull
- **Git fetch**
  Download branches and tags from one or more repositories. Fetch the remote repository:
  $ git fetch< repository Url> Fetch a specific branch:
  $ git fetch
  Fetch all the branches simultaneously:
  $ git fetch -all
  Synchronize the local repository:
  $ git fetch origin

12. Undo changes

- **Git revert**
  Undo the changes:
  $ git revert
  Revert a particular commit:
  $ git revert
- **Git reset**
  Reset the changes:

Reset the changes.

$ git reset -hard

$ git reset -soft:

$ git reset --mixed

13. Removing files

- **Git rm**

  Remove the files from the working tree and from the index:

  $ git rm <file Name>

  Remove files from the Git But keep the files in your local repository:

  $ git rm --cached

- [ ] **Interview Notes-**