**GIT & GITHUB**

**1. What is Git?**

* Git is a Distributed Version Control system (DVCS)
* It can track changes to a file and allows you to revert to any change.

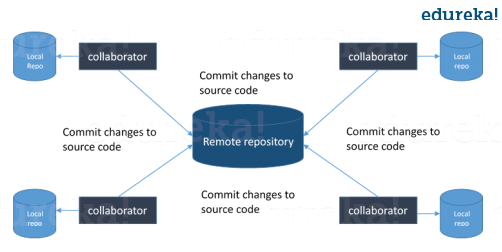
**2. What is the advantage of Git over other Version Control Systems (VCS)?**

**- Git is a Distributed Version Control System, whereas others are a Centralized Version Control System. They have a direct dependency on the Centralized Servers like CVS, SVN, Serena etc.**

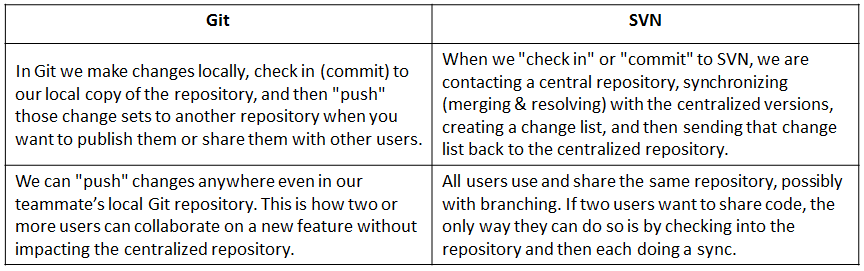
**- In Centralized Version Control System (CVS), if the central server is down, then it is difficult to maintain versioning in the local. If the server is down for too long, it would impact heavily, whereas in the DVCS, if the remote server is down, still version control and maintaining is possible through local repository.**

**-** Major advantage is that it does not rely on a central server to store all the versions of a project’s files

- Instead, every developer “clones” a copy of a repository I have shown in the diagram with “Local repository” and has the full history of the project on his hard drive so when there is a server outage all you need for recovery is one of your teammate’s local Git repository. There is a central cloud repository as well where developers can commit changes and share it with other teammates as you can see in the diagram where all collaborators are committing changes “Remote repository”.

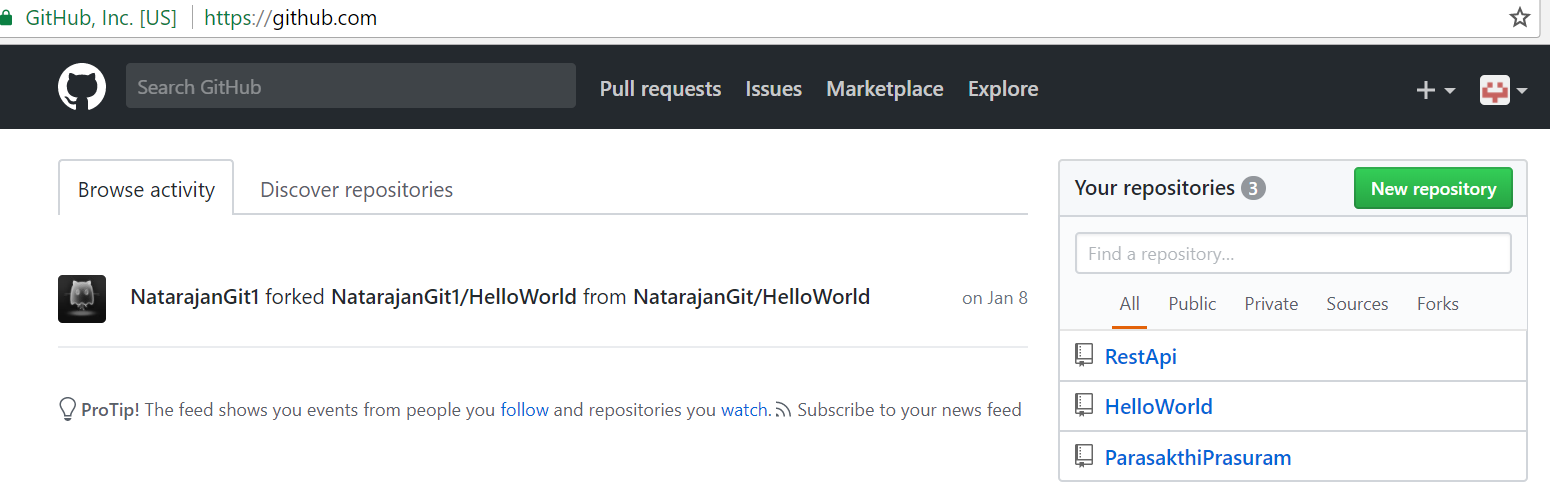


**3. What is the difference between Git and SVN**



**4. What is GitHub?**

**- GitHub is a web hosting service, which has a collective version of all Git repositories.**

****

**5. What is the command to write a commit message in Git?**

* **git commit -a**

-a instructs GIT to commit all the new content in the modified files.

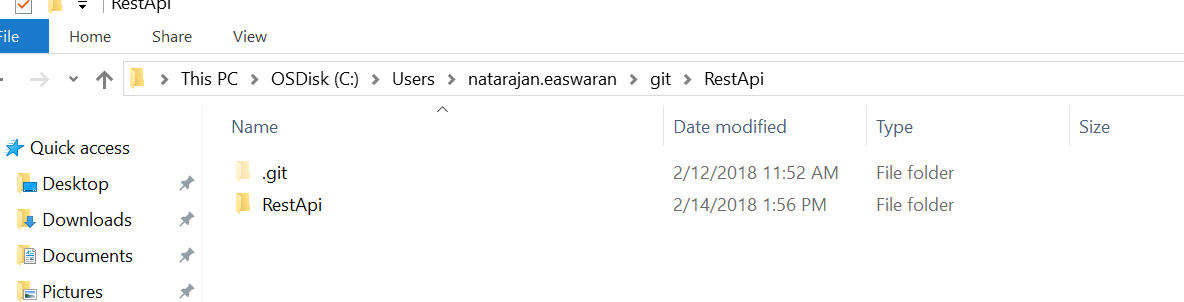
- **git add<file>**

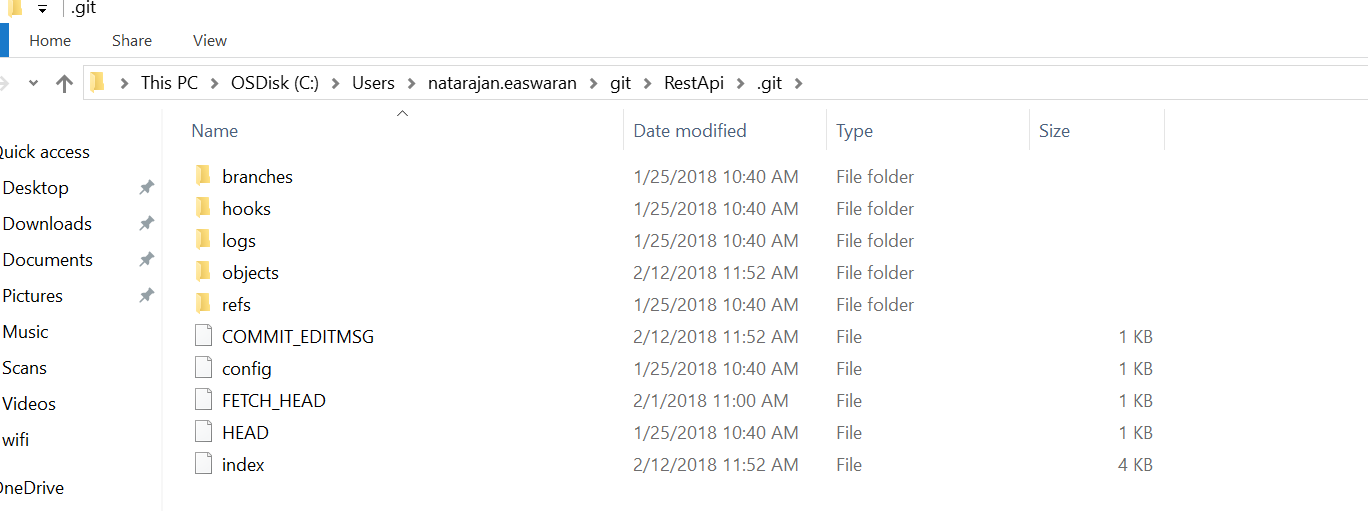
**- Adds new file to commit for first time.**

**6. What is a working directory in GIT?**

**- A working directory contains**

* A .git subdirectory with all the Git related revision history of your repo.
* A working tree, or checked out copies of your project files.







**7. What is a bare repository?**

- Bare repository contain no working or checked out copy of your source files.

- Bare repos store git revision history of your repo in the root folder of your repository instead of in a .git subfolder. Bare repositories are customarily given a .git extension. (something like RestApi.git)

- These are the repositories checked in GitHub.com for sharing. Because git is a distributed version control system, no one will directly edit files in the shared centralized repository. Instead developers will clone the shared bare repo, make changes locally in their working copies of the repo, then push back to the shared bare repo to make their changes available to other users. Because no one ever makes edits directly to files in the shared bare repo, a working tree is not needed.

**8. What language is used in GIT?**

- Git uses ‘C’ programming language.

**9. Difference between Git Pull and Git Merge?**

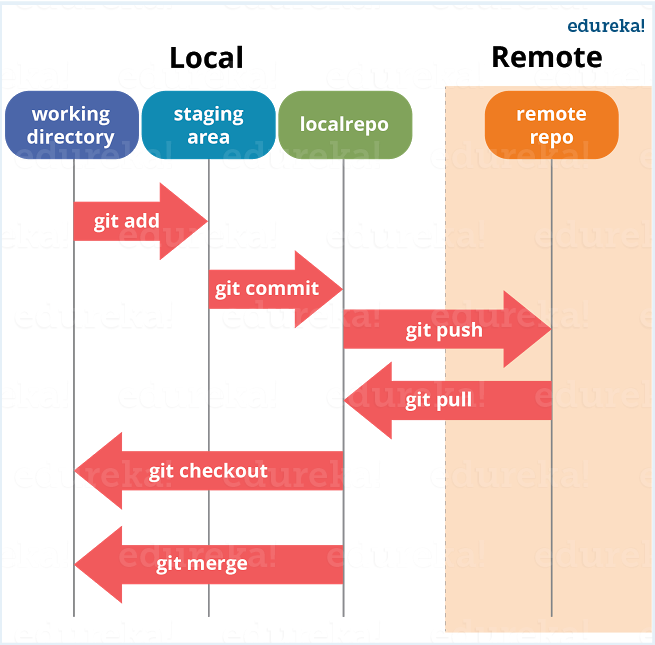
**-** Git pull command pulls new changes or commits from a particular branch from your central repository and updates your target branch in your local repository.

Git fetch is also used for the same purpose but it works in a slightly different way. When you perform a git fetch, it pulls all new commits from the desired branch and stores it in a new branch in your local repository. If you want to reflect these changes in your target branch, git fetch must be followed with a git merge. Your target branch will only be updated after merging the target branch and fetched branch. Just to make it easy for you, remember the equation below:

**Git pull = git fetch + git merge**

**10. What is ‘staging area’ or ‘index’ in Git?**

**-** Every file before committing goes through an intermediate staged area for review and verification called “Staged Area” or “Index”



**11. What is forking a repository?**

**-** In a large organization, developers are not directly committing or modifying the main code, instead they are being asked to fork a repository and then make changes on the fork and later merged into the main code through a pull request. This way, the main code is not altered by the developers, but only goes through pull request, which required Compile success and Peer review approval to get merged.

A fork is a copy of a **repository**. **Forking a repository** allows you to freely experiment with changes without affecting the original project. Most commonly, forks are used to either propose changes to someone else's project or to use someone else's project as a starting point for your own idea.

**12. Difference between Forking a repository and cloning a repository?**

- Forking a repository is still a remote repository under the User Name.

- Cloning a repository is getting the cloned copy of the repository in the local machine. Once we work on the local copy, then it is checked into the local repository and then gets committed to the remote forking repository.

- After the code is checked into Forking repo, then it is merged with the base code through pull request.

**13. In Git how do you revert a commit that has already been pushed and made public?**

**- git revert <name of bad commit>**

**14. What is Git stash?**

**-** Often, when you’ve been working on part of your project, things are in a messy state and you want to switch branches for some time to work on something else. The problem is, you don’t want to do a commit of half-done work just so you can get back to this point later. The answer to this issue is Git stash.

Stashing takes your working directory that is, your modified tracked files and staged changes and saves it on a stack of unfinished changes that you can reapply at any time.

**15. What is Git stash drop?**

**-** Git ‘stash drop’ command is used to remove the stashed item. It will remove the last added stash item by default, and it can also remove a specific item if you include it as an argument.

If you want to remove a stash item from the list of stashed items you can use the below commands:

**git stash list:**It will display the list of stashed items like:

stash@{0}: WIP on master: 049d078 added the index file  
stash@{1}: WIP on master: c264051 Revert “added file\_size”  
stash@{2}: WIP on master: 21d80a5 added number to log

If you want to remove an item named stash@{0} use command **git stash drop stash@{0}**.

**16.** **What is the function of ‘git config’?**

**-** Git uses your username to associate commits with an identity. The git config command can be used to change your Git configuration, including your username.

Suppose you want to give a username and email id to associate commit with an identity so that you can know who has made a particular commit. For that I will use:

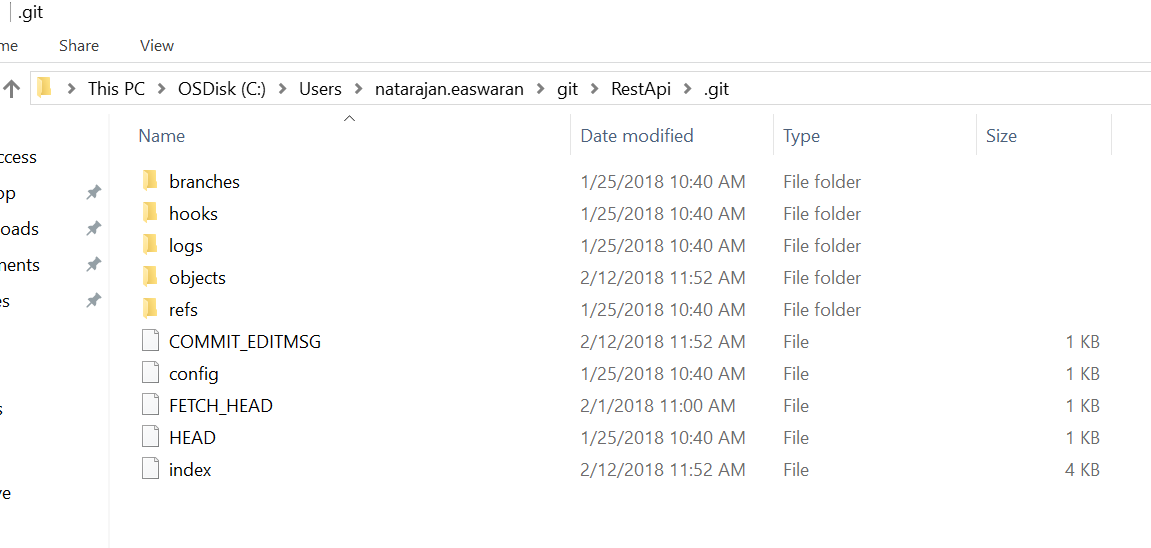
**git config –global user.name “Your Name”:**This command will add username.  
**git config –global user.email “Your E-mail Address”:**This command will add email id.

**17. What does commit object contains?**

* A set of files, representing the state of a project at a given point of time
* Reference to parent commit objects
* An SHAI name, a 40-character string that uniquely identifies the commit object.

18. **How can you create a repository in Git?**

**-** To create a repository, create a directory for the project if it does not exist, then run command “**git init**”. By running this command .git directory will be created in the project directory.



**19.** **What is Git bisect? How can you use it to determine the source of a (regression) bug?**

**-** Git bisect is used to find the commit that introduced a bug by using binary search. Command for Git bisect is

**git bisect <subcommand> <options>**

This command uses a binary search algorithm to find which commit in your project’s history introduced a bug. You use it by first telling it a “bad” commit that is known to contain the bug, and a “good” commit that is known to be before the bug was introduced. Then Git bisect picks a commit between those two endpoints and asks you whether the selected commit is “good” or “bad”. It continues narrowing down the range until it finds the exact commit that introduced the change.

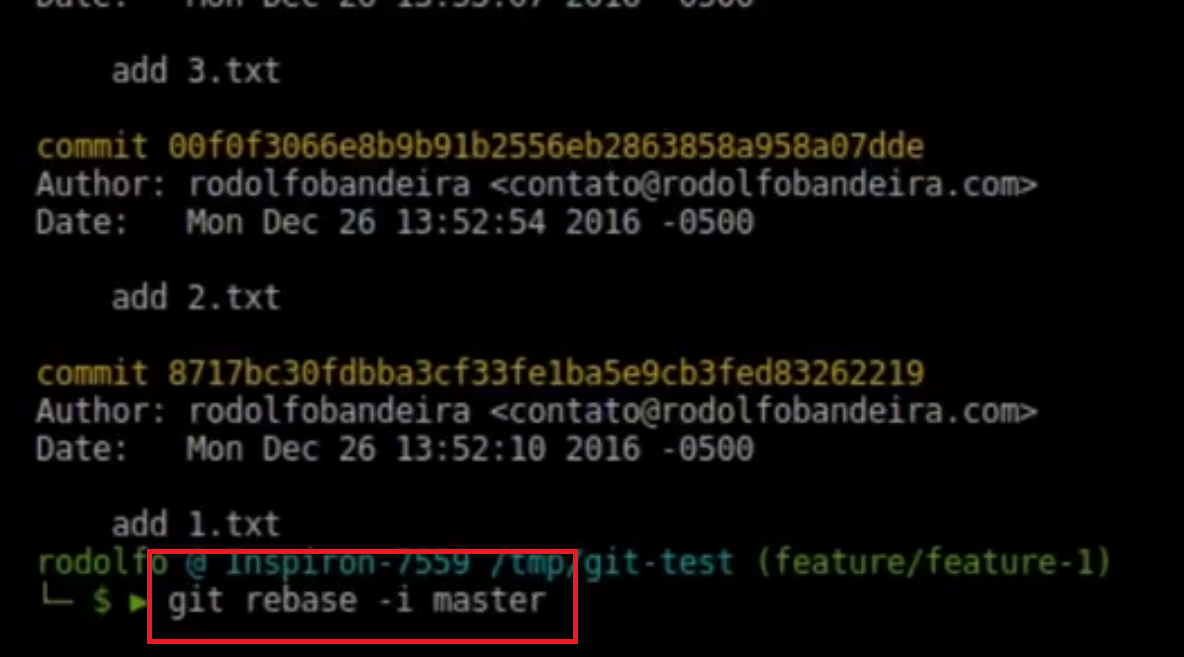
**20. What is squashing last N commits into a single commit?**

**-** Weare doing several commits for a develop branch and then we are merging them into Master. While merging we don’t want several branches to be merged into master, but instead squash the last N commits (3 or 4 or whatever number of commits) into one single commit and merge into Master, so that the Master gets the single commit from branch while merging.

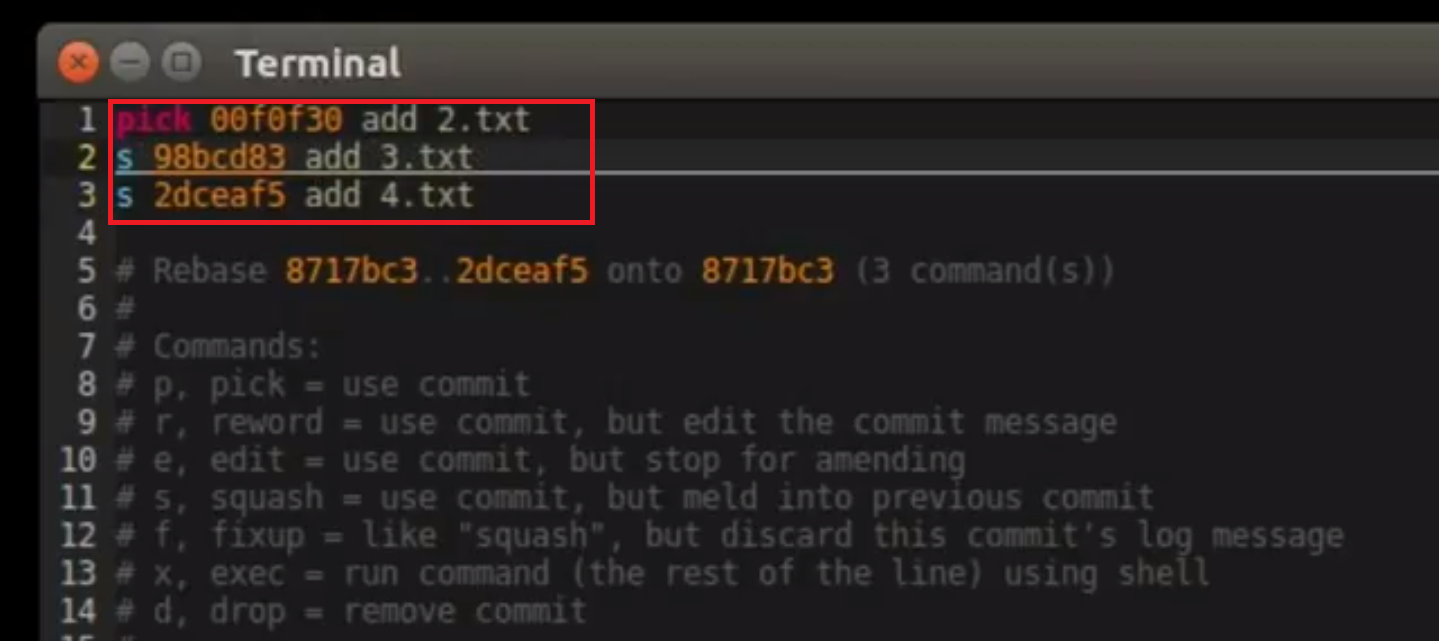
**21. How to do the squashing of last N commits into single one?**

- After doing N commits, use the command

**git rebase -i master**

****

After the command is executed, it will open the following window. Replace the pick into ‘s’, meaning we are squashing the commits into a single commit.

****

And then use “**git log**“ command to see the commits are squashed

**22. What are the branching strategies followed**

* Feature branching

A feature branch model keeps all of the changes for a particular feature inside of a branch. When the feature is fully tested and validated by automated tests, the branch is then merged into master.

* Task branching

In this model each task is implemented on its own branch with the task key included in the branch name. It is easy to see which code implements which task, just look for the task key in the branch name.

* Release branching

Once the develop branch has acquired enough features for a release, you can clone that branch to form a Release branch. Creating this branch starts the next release cycle, so no new features can be added after this point, only bug fixes, documentation generation, and other release-oriented tasks should go in this branch. Once it is ready to ship, the release gets merged into master and tagged with a version number. In addition, it should be merged back into develop branch, which may have progressed since the release was initiated.

**23.** **How will you know in Git if a branch has already been merged into master?**

- To know if a branch has been merged into master or not you can use the below commands:

**git branch –merged**: It lists the branches that have been merged into the current branch.  
**git branch –no-merged:**  It lists the branches that have not been merged.

**24. What is a Git Rebase? What is the difference between Git Merge and Git Rebase?**

**-** Git rebase is a command which will merge another branch into the branch where you are currently working, and move all the local commits that are ahead of the rebased branch to the top of the history on that branch.

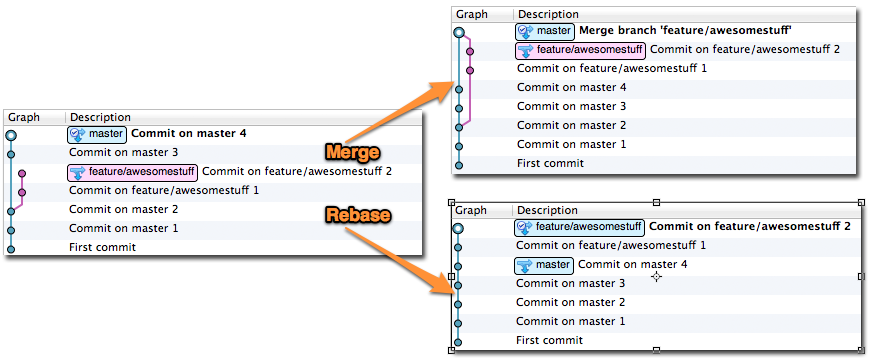
**25. What is Sub Git?**

- Sub Git is a tool for SVN to Git migration. It creates a writable Git mirror of a local or remote Subversion repository and uses both Subversion and Git as long as you like.

**26. Difference between Git Commit and Git Push?**

**-** Git Commit push the code to local repository, and git push will push the code to the remote repository

Difference between Merge and Rebase

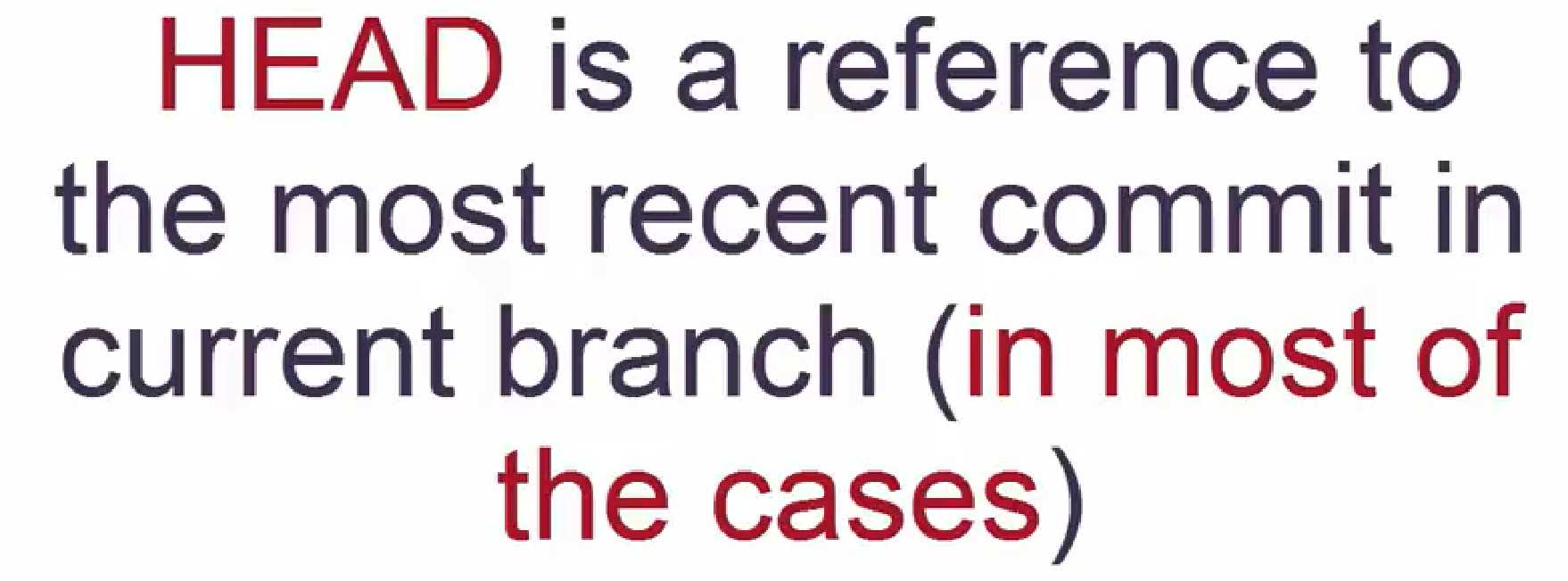


In above fig Commits 1,2,3 and 4 are done on master and after commit on master 2, branch is created for feature. awesomestuff 1,2 are committed in feature branch.

In case of Merge, the feature/awesomestuff is merged into master and created a separate commit for the merge, whereas in the rebase, the commits in the feature branch are added to the top of the master branch making it linear and making it a proper working history.

**27. What is ‘head’ in git?**

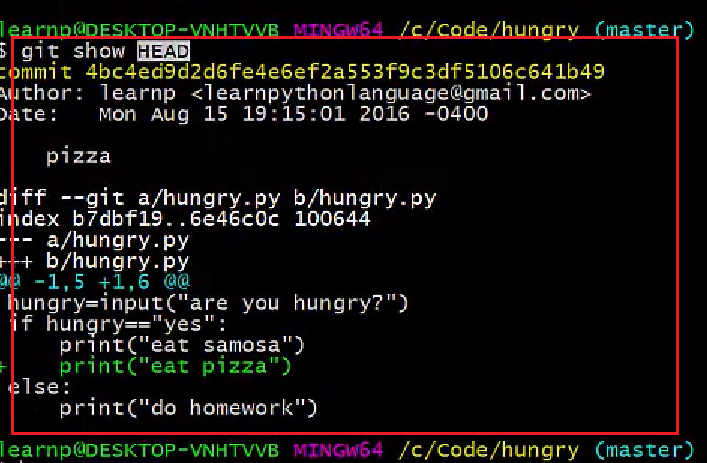
**-** HEAD is the most recent commit for a particular branch

****

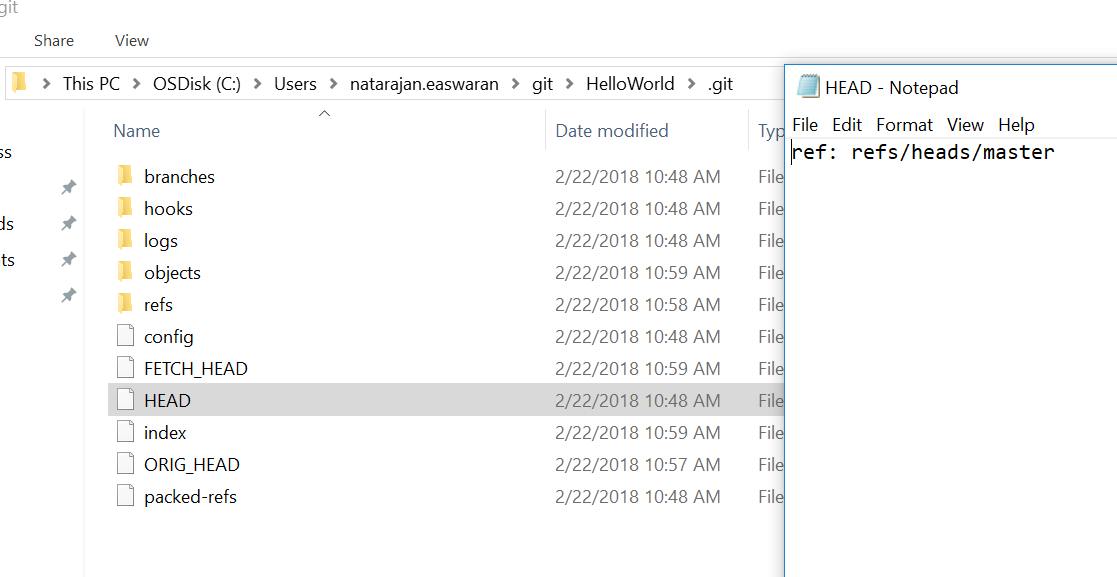
For example



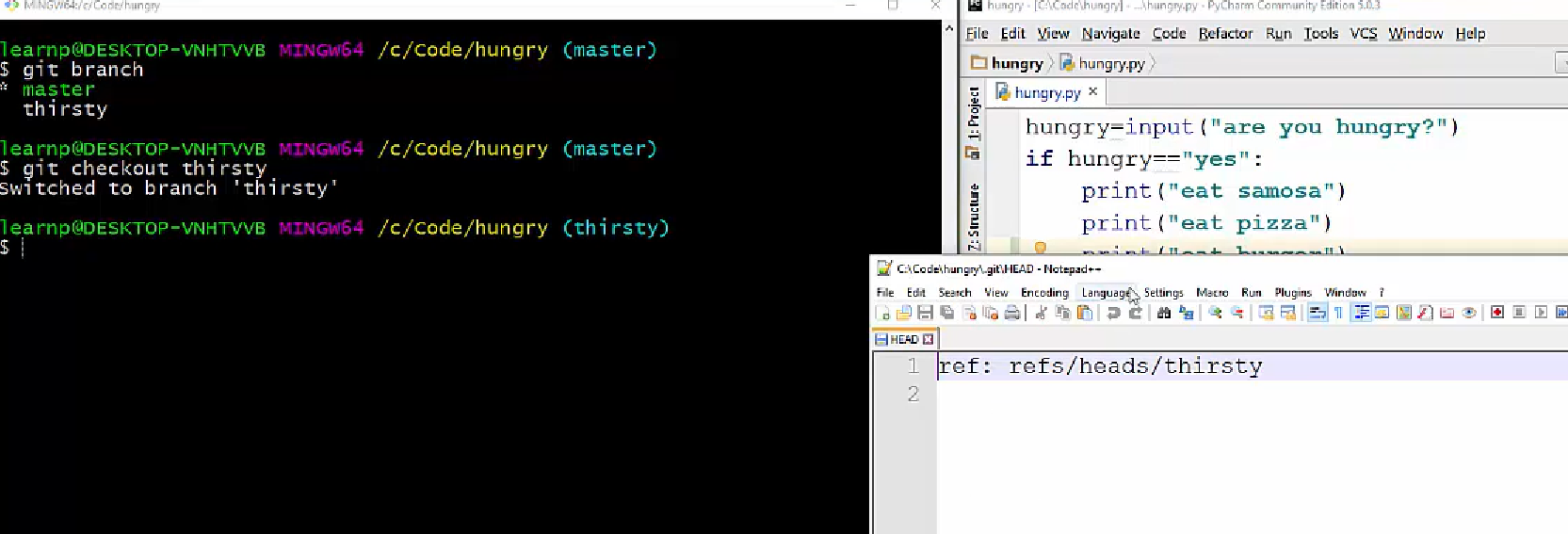
The last commit is the head (commit for pizza is the head in the ex above). When we execute command “git show HEAD”, it gives the most recent commit.



If we go to the project and open HEAD, we see it is referred to master. If we switch branch, the HEAD becomes the new branch.

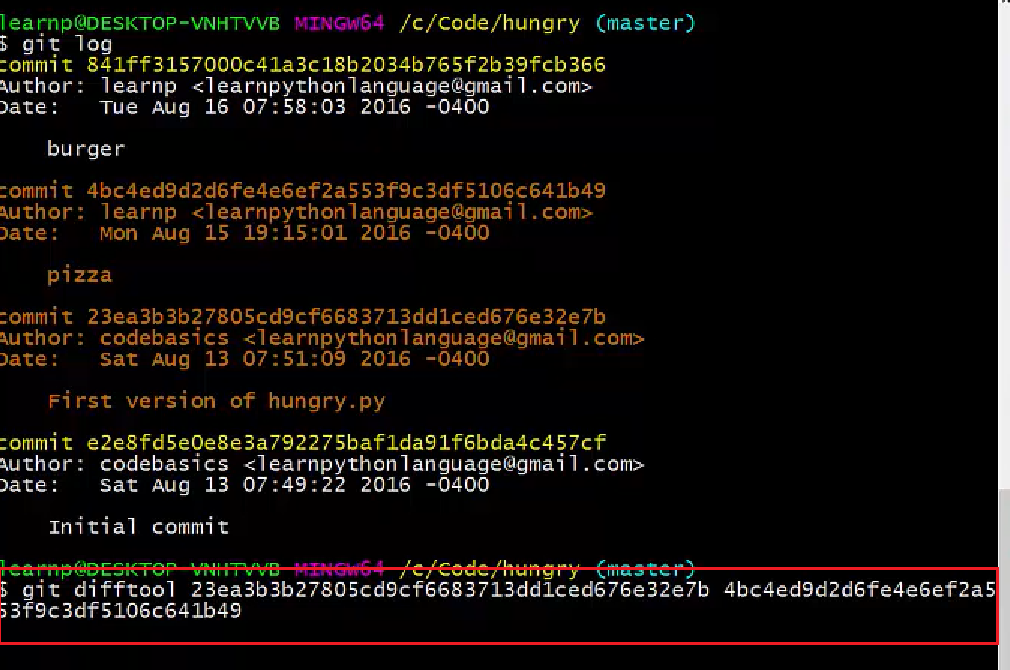


If we switch to a branch called ‘thirsty’, then HEAD refers to thirsty

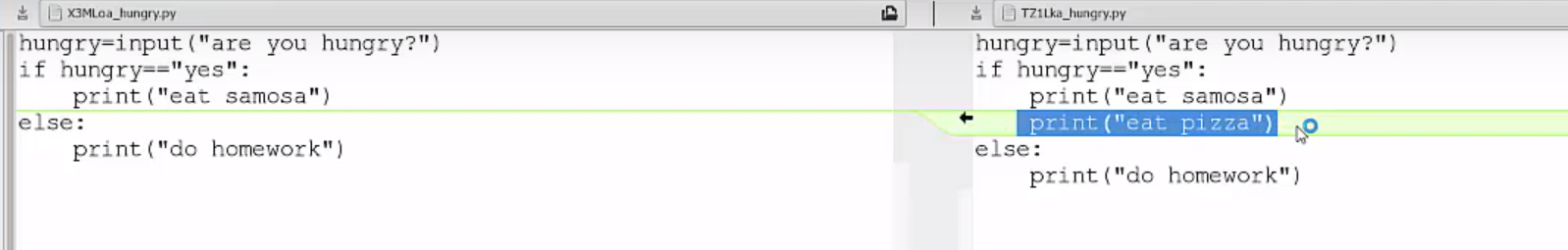


**28. Which command is giving difference between 2 commits?**

- git difftool commit 1 commit 2

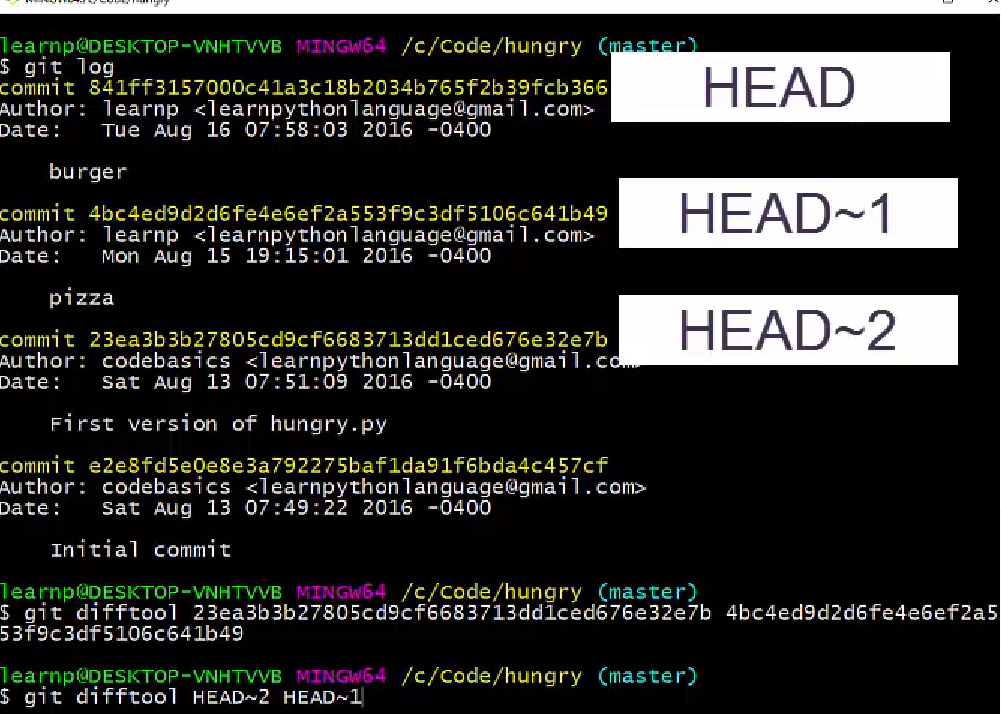


It will open a difftool and show the difference

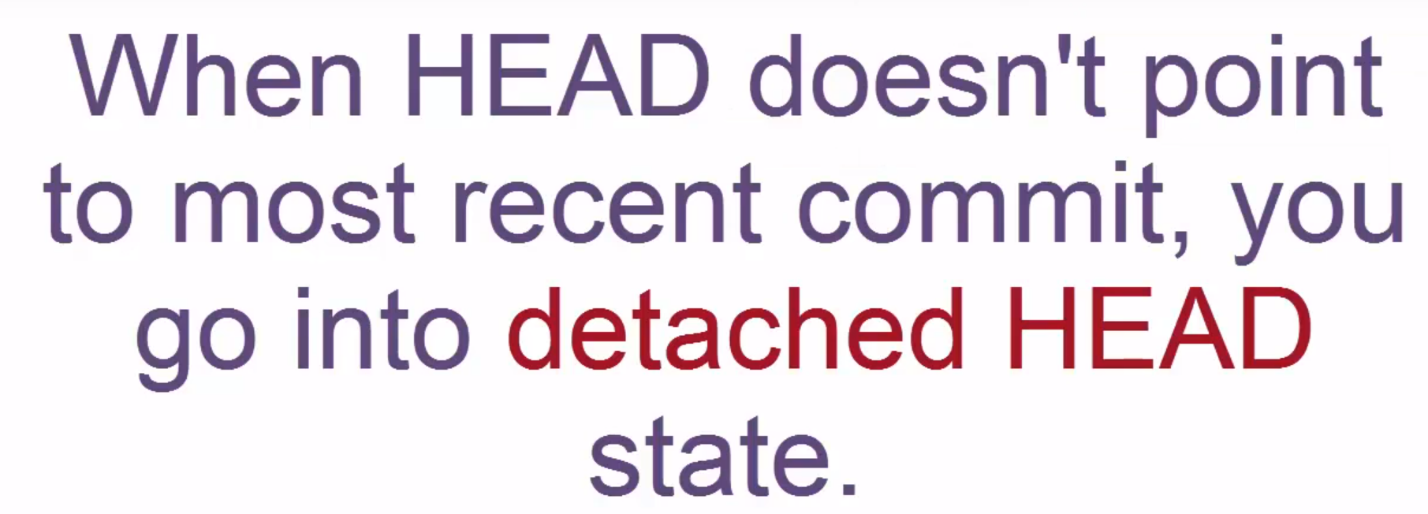


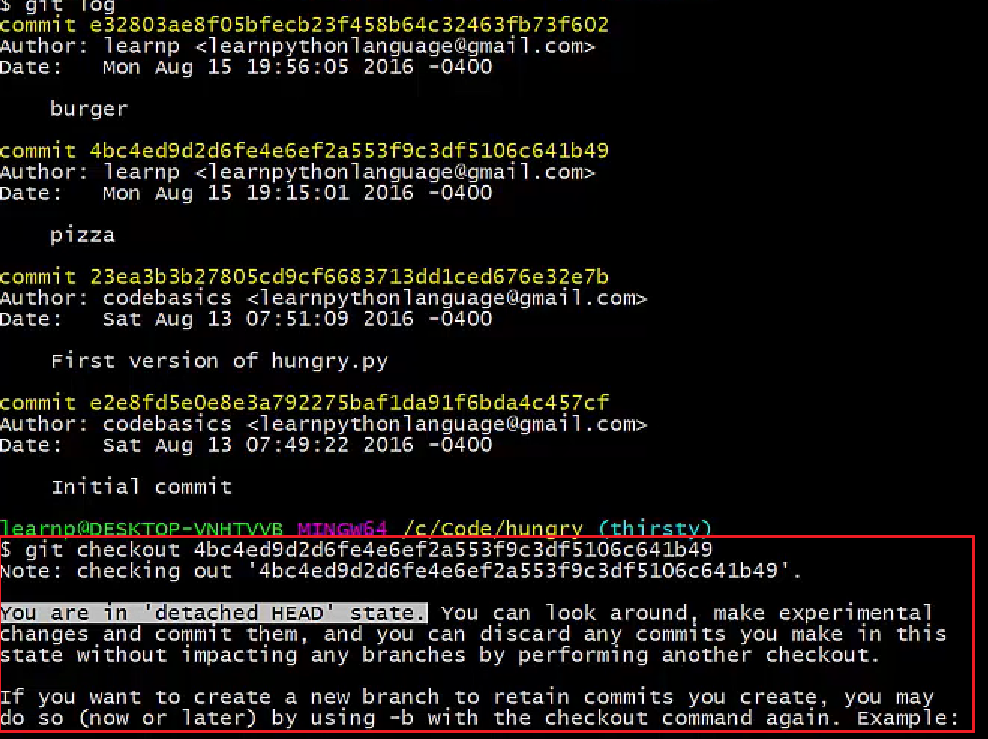
Same can be done through command

Git difftool HEAD~2 HEAD ~1, because (fig below)



**29. What is called detached HEAD?**

****

****

**30. What is the purpose of branching in git?**

**-** Branch is used for working on things parallelly. For example, there is a branch for a particular release, we can create a branch from them and work on codes for another release for which the code for the previous release is unaffected. Once the other release is done, we can merge the code for this release into the other one

**31. What is the common branching pattern in Git?**

**-** The common way of creating branch in GIT is to maintain one as “Main“

branch and create another branch to implement new features.

**32. What is a conflict in GIT and how to resolve the conflict?**

**-** A conflict occurs when a file has been committed, and there is a change happened on top of the commit and that has been committed before your changes has been committed. So, there is a difference in the file in your local vs the committed changes. This creates a conflict.

In order to resolve, we need to edit and merge the files in repo and your local, and then do a commit.

**33. How to delete a branch locally and remotely in GIT?**

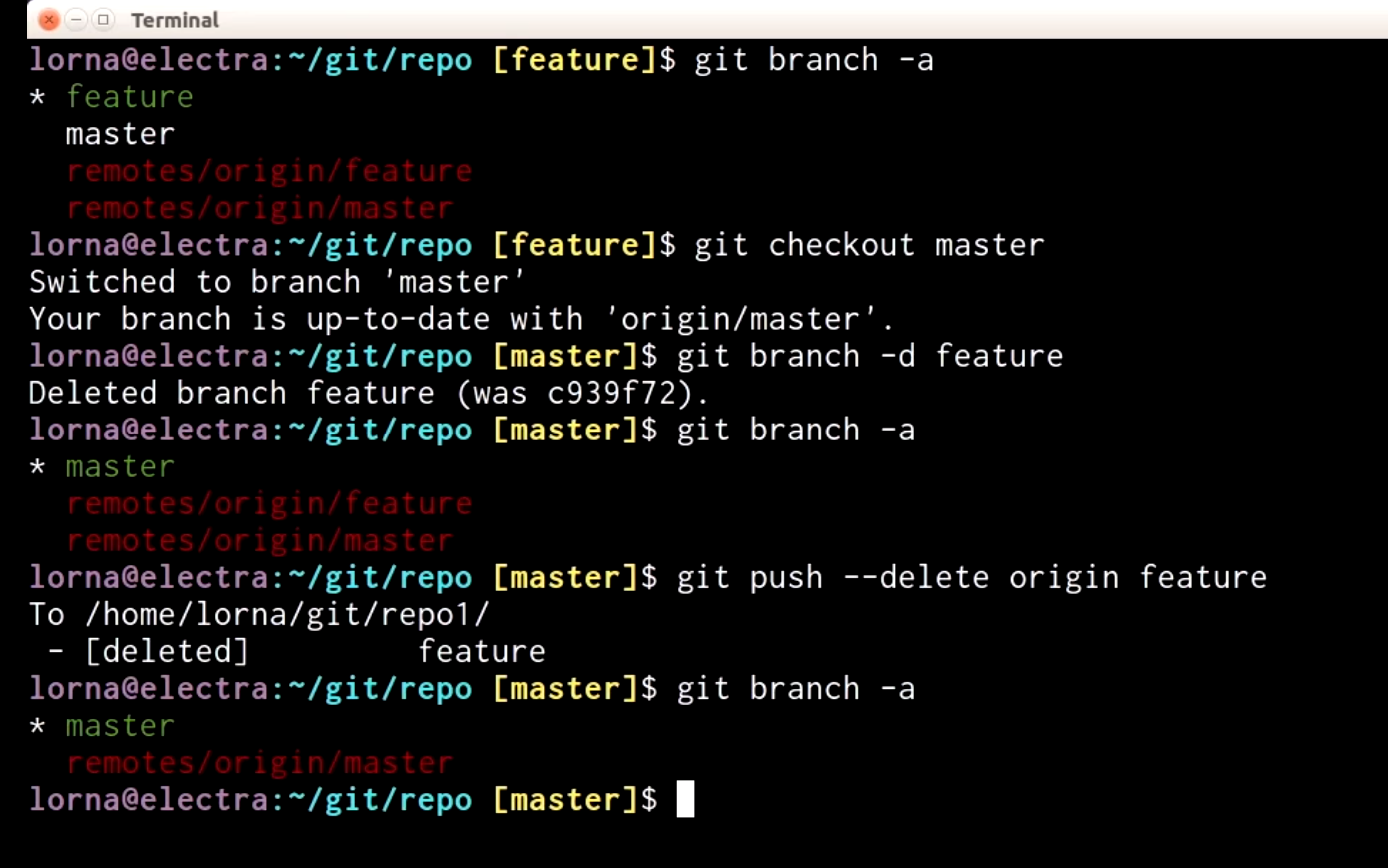
Consider, we are deleting a feature branch.

**git branch -a**: lists all branches locally and remotely

**git checkout master**: Since we cannot delete feature branch from feature, changing it into master

**git branch -d feature:** Deletes feature branch locally

**git push –delete origin feature**: deletes feature branch remotely

****

**34. What is git status?**

- List which files are staged, unstaged, and untracked.

The git status command displays the state of the working directory and the staging area. It lets you see which changes have been staged, which haven’t, and which files aren’t being tracked by Git. Status output does *not* show you any information regarding the committed project history. For this, you need to use [git log](https://www.atlassian.com/git/tutorials/inspecting-a-repository/git-log).

Usage

git status

List which files are staged, unstaged, and untracked.

Discussion

The git status command is a relatively straightforward command. It simply shows you what's been going on with git add and git commit. Status messages also include relevant instructions for staging/unstaging files. Sample output showing the three main categories of a git status call is included below:

# On branch master

# Changes to be committed:

# (use "git reset HEAD <file>..." to unstage)

#

#modified: hello.py

#

# Changes not staged for commit:

# (use "git add <file>..." to update what will be committed)

# (use "git checkout -- <file>..." to discard changes in working directory)

#

#modified: main.py

#

# Untracked files:

# (use "git add <file>..." to include in what will be committed)

#

#hello.pyc

**Ignoring Files**

Untracked files typically fall into two categories. They're either files that have just been added to the project and haven't been committed yet, or they're compiled binaries like .pyc, .obj, .exe, etc. While it's definitely beneficial to include the former in the git status output, the latter can make it hard to see what’s actually going on in your repository.

For this reason, Git lets you completely ignore files by placing paths in a special file called [.gitignore](https://www.atlassian.com/git/tutorials/gitignore). Any files that you'd like to ignore should be included on a separate line, and the \* symbol can be used as a wildcard. For example, adding the following to a .gitignore file in your project root will prevent compiled Python modules from appearing in git status:

\*.pyc

Example

It's good practice to check the state of your repository before committing changes so that you don't accidentally commit something you don't mean to. This example displays the repository status before and after staging and committing a snapshot:

# Edit hello.py

git status

# hello.py is listed under "Changes not staged for commit"

git add hello.py

git status

# hello.py is listed under "Changes to be committed"

git commit

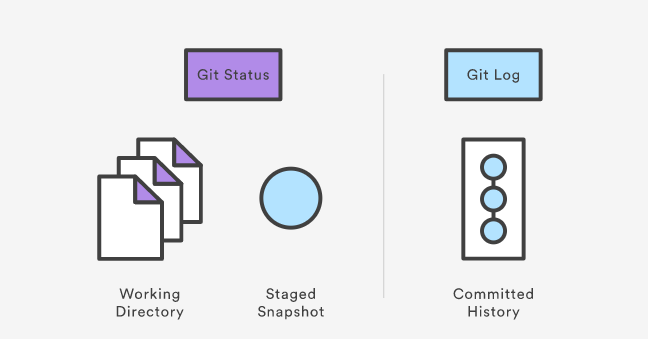
git status

# nothing to commit (working directory clean)

The first status output will show the file as unstaged. The git addaction will be reflected in the second git status, and the final status output will tell you that there is nothing to commit—the working directory matches the most recent commit. Some Git commands (e.g., [git merge](https://www.atlassian.com/git/tutorials/using-branches/git-merge)) require the working directory to be clean so that you don't accidentally overwrite changes.

**35. What is Git log?**

The git log command displays committed snapshots. It lets you list the project history, filter it, and search for specific changes. While git status lets you inspect the working directory and the staging area, git log only operates on the committed history.



Log output can be customized in several ways, from simply filtering commits to displaying them in a completely user-defined format. Some of the most common configurations of git log are presented below.

git log

Display the entire commit history using the default formatting. If the output takes up more than one screen, you can use Space to scroll and q to exit.

git log -n <limit>

Limit the number of commits by <limit>. For example, git log -n 3 will display only 3 commits.

git log --oneline

Condense each commit to a single line. This is useful for getting a high-level overview of the project history.

git log --stat

Along with the ordinary git log information, include which files were altered and the relative number of lines that were added or deleted from each of them.

git log -p

Display the patch representing each commit. This shows the full diff of each commit, which is the most detailed view you can have of your project history.

git log --author="<pattern>"

Search for commits by a particular author. The <pattern>argument can be a plain string or a regular expression.

git log --grep="<pattern>"

Search for commits with a commit message that matches <pattern>, which can be a plain string or a regular expression.

git log <since>..<until>

Show only commits that occur between <since> and <until>. Both arguments can be either a commit ID, a branch name, HEAD, or any other kind of [revision reference](http://www.kernel.org/pub/software/scm/git/docs/gitrevisions.html).

git log <file>

Only display commits that include the specified file. This is an easy way to see the history of a particular file.

git log --graph --decorate --oneline

A few useful options to consider. The --graph flag that will draw a text based graph of the commits on the left hand side of the commit messages. --decorate adds the names of branches or tags of the commits that are shown. --oneline shows the commit information on a single line making it easier to browse through commits at-a-glance.

### Discussion

The git log command is Git's basic tool for exploring a repository’s history. It’s what you use when you need to find a specific version of a project or figure out what changes will be introduced by merging in a feature branch.

commit 3157ee3718e180a9476bf2e5cab8e3f1e78a73b7

Author: John Smith

Most of this is pretty straightforward; however, the first line warrants some explanation. The 40-character string after commitis an SHA-1 checksum of the commit’s contents. This serves two purposes. First, it ensures the integrity of the commit—if it was ever corrupted, the commit would generate a different checksum. Second, it serves as a unique ID for the commit.

This ID can be used in commands like git log <since>..<until> to refer to specific commits. For instance, git log 3157e..5ab91 will display everything between the commits with ID's 3157e and 5ab91. Aside from checksums, branch names (discussed in the [Branch Module](https://www.atlassian.com/git/tutorials/using-branches)) and the HEAD keyword are other common methods for referring to individual commits. HEAD always refers to the current commit, be it a branch or a specific commit.

The ~ character is useful for making relative references to the parent of a commit. For example, 3157e~1 refers to the commit before 3157e, and HEAD~3 is the great-grandparent of the current commit.

The idea behind all of these identification methods is to let you perform actions based on specific commits. The git logcommand is typically the starting point for these interactions, as it lets you find the commits you want to work with.

### Example

The Usage section provides many examples of git log, but keep in mind that several options can be combined into a single command:

git log --author="John Smith" -p hello.py

This will display a full diff of all the changes John Smith has made to the file hello.py.

The .. syntax is a very useful tool for comparing branches. The next example displays a brief overview of all the commits that are in some-feature that are not in master.

git log --oneline master..some-feature

**36. What is git rm?**

- Removes the file from Staging area.

**37. What is Git reset?**

- The git reset command is a complex and versatile tool for undoing changes

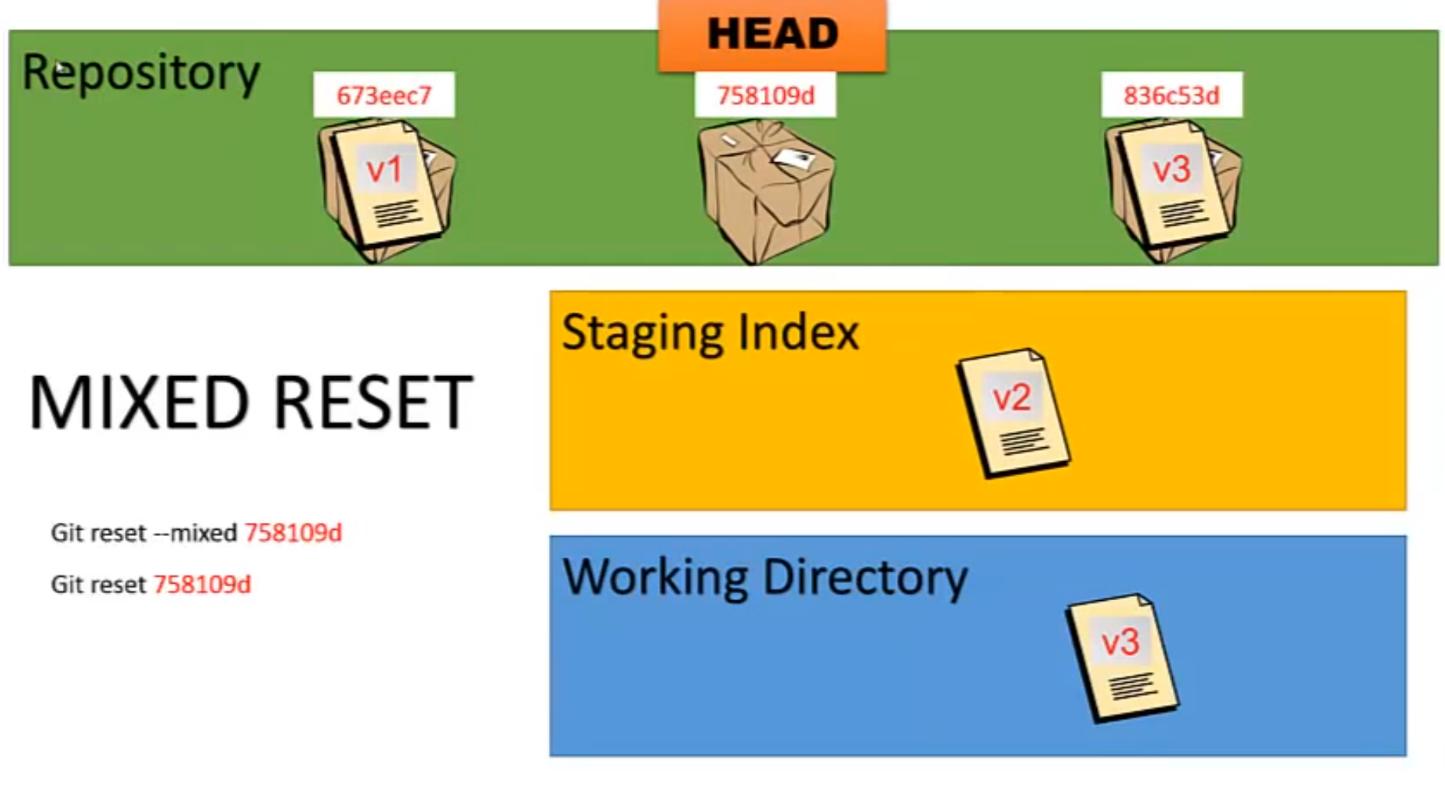
**37. Types of Git reset?**

****

**SOFT RESET**

****

**MIXED RESET**

****

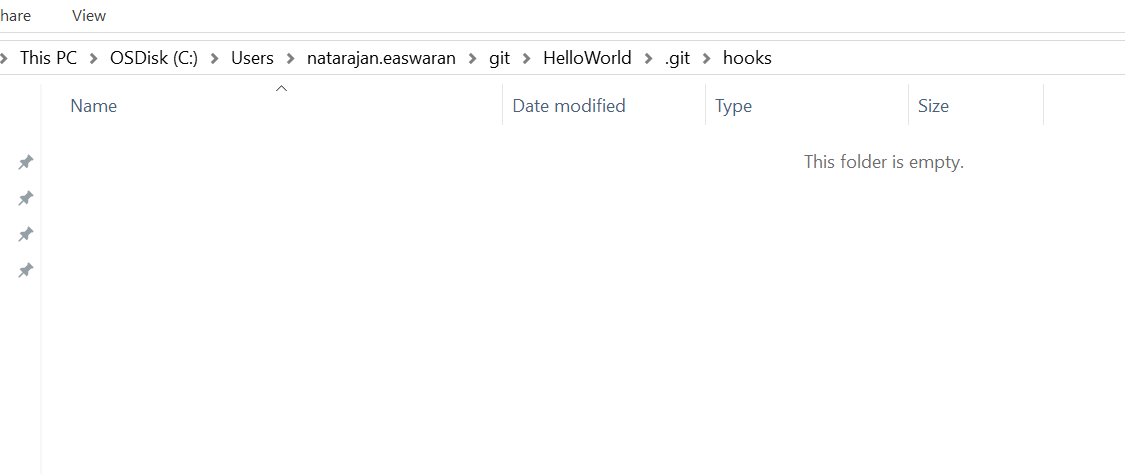
**HARD RESET**

****

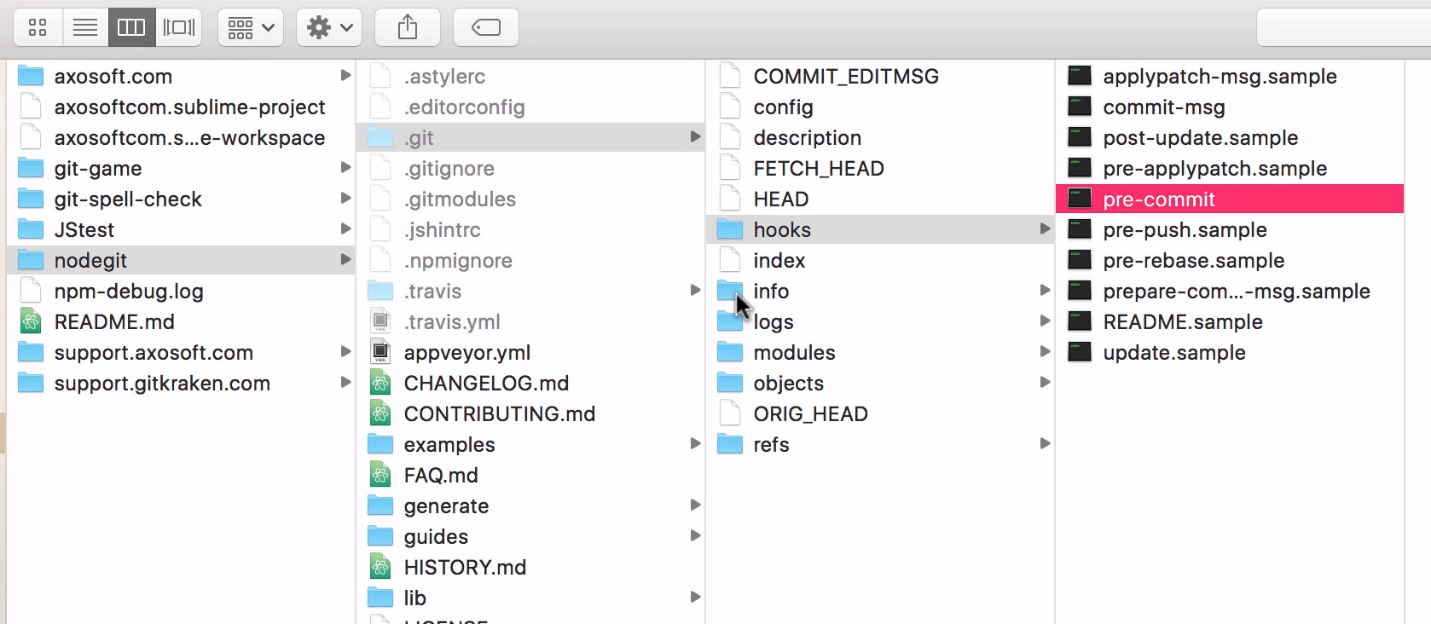
**38. What does ‘hooks’ consist of in git?**

**- Git hooks** are scripts that **Git** executes before or after events such as: commit, push, and receive. **Git hooks** are a built-in feature - no need to download anything. **Git hooks** are run locally.

For example, if you don’t want to commit with a problem, you can activate a git hook to set, then if there is a commit with a problem, once we execute git commit, then the shell script executes from git hook and prevents it from committing.

****

Git Hook scripts can be configured here. Something like below



**39. What does git commit—amend do?**

- This is used to edit the last commit message and allows to add new modified message.

Also, Changing the Last **Commit**: **git commit** --**amend**. The **git commit** --**amend** command is a convenient way to modify the most recent **commit**. It lets you combine staged changes with the previous **commit** instead of creating an entirely new**commit**.

If you have modified files, it will commit your modified files to your last commit. If not you can use this command to modify the last committed message.

**40**. **Why is it advisable to create an additional commit rather than amending an existing commit?**

There are couple of reason

a)      The amend operation will destroy the state that was previously saved in a commit.  If it’s just the commit message being changed then that’s not an issue.  But if the contents are being amended then chances of eliminating something important remains more.

b)      Abusing “git commit- amend” can cause a small commit to grow and acquire unrelated changes.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*