**Git vs GitHub**

| **S.No.** | **Git** | **GitHub** |
| --- | --- | --- |
| 1. | Git is a software. | GitHub is a service. |
| 2. | Git is a command-line tool | GitHub is a graphical user interface |
| 3. | Git is installed locally on the system | GitHub is hosted on the web |
| 4. | Git is maintained by linux. | GitHub is maintained by microsoft. |
| 5. | Git is focused on version control and code sharing. | GitHub is focused on centralized source code hosting. |
| 6. | Git is a version control system to manage source code history. | GitHub is a hosting service for Git repositories. |
| 7. | Git was first released in 2005. | GitHub was launched in 2008. |
| 8. | Git has no user management feature. | GitHub has built-in user management feature. |

**Configuring user name and email:**

* git config --global user.name "user name" --- configuring the user name
* git config user.name --- to check if user is configured
* git config user.email --- to if email configured
* git config --global user.email --- to configure email

|  |  |
| --- | --- |
| Git Command | Description |
| 1. git init 2. git status 3. git add –flag 4. git commit -m 'message' 5. git log --<flag> | 1.  2. checking the untracked/stages files  3.  4.  5. Checking the logs/history of all commits |

**Repository:**

A git repo is workspace which tracks and manages files.

We can have any number of repo in our machines as needed, all with seperate histories and contents

**Initializing the repository**

git init

Create a new git repo in the path where we ran this command.

**Keeping git commit atomic**

We should try to make our git commit atomic. It means a commit should encompass a single feature, change or fix. In other words, try to keep each commit focused on a single thing.

**# .git hidden folder#**

It is hidden directory which contains all the logs about the repository.

if we delete this hidden directory then it will delete all logs

Note:

Don't initialize any folder or repo inside another repo, i.e don't know nesting of git init

**#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.

Files which are not added are called untracked file.

git add -A or git add . ----- to add all files from repository into stating/index are

git reset <filename> ------- to undo the git add

git add <filename(s)> ----- to add file into staging area  
It add(git add .) all untracked file to staging area or repo doesn’t matter from which branch it ran.

**git status:**

It is used to check for untracked/stages files. When we run(from any branch), it shows all untracked/staged files in repo.

**#git commit#**

this basically updates .git folder, add all message or info.

git commit -m "my message"

git commit -amend ---- to Change commit message. if ran then it prompt for entering message

Git Reset ---- Git undo commit

Git Revert -- git revert commit

Git Rm ---- git remove commit

Working Area ----(git add)-----> Staging area----(git commit)-->repository

**Git Ignore**

This is used to tell git to INGORE/untrack some file from repo.

Create a file by name '.gitignore' and add all files/folder that you want to untract/ignore in '.gitignore'.

'.gitignore' should be created in project/repo root directed.

**Main branch vs Master branch**

In 2020, GitHub renamed the default branch from master to main. The default git brac=nch name is still master, though the git team is exploring a potential change.

**What is HEAD in any branch name?**

It refers to the branch name and commit point where you are currently.

**Operations on Branches**

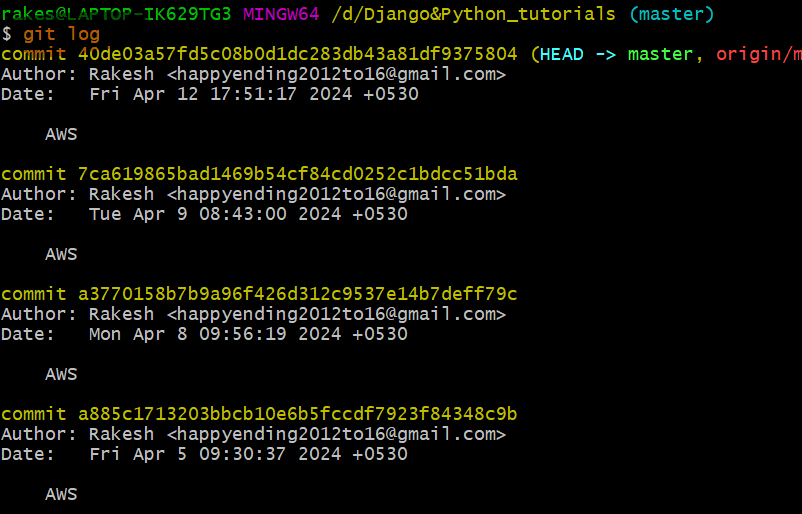
We can perform various operations on Git branches. The git branch command allows you to create, list, rename and delete branches

**git log --<flag>**

The git log command displays all of the commits in a repository’s history(commit hash-id, author, date when commit was made and commit message).

Enter 'q' to exit from the 'git log –flag' window.

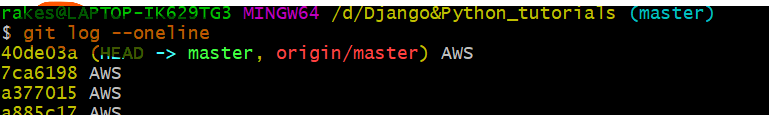
Example.



We have below all possible flag values for log

1. **–oneline**

* The --oneline flag causes git log to display one commit per line
* the first seven characters of the SHA
* the commit message



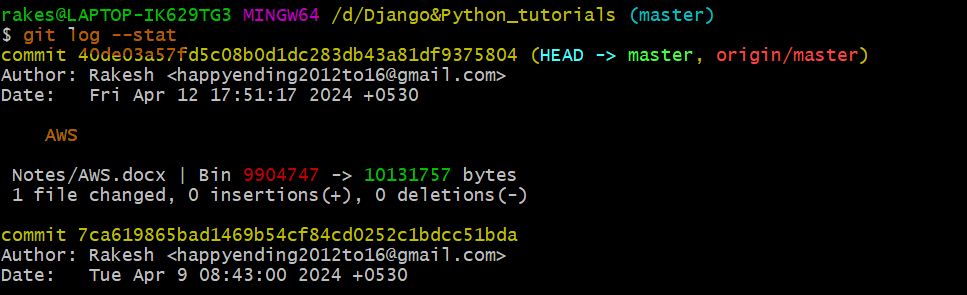
1. **–stat**

the --stat flag causes git log to display

* the files that were modified in each commit
* the number of lines added or removed
* a summary line with the total number of files and lines changed

**Example:**

In below command it shows that AWS file was changes.



1. **–graph**

The --graph flag enables you to view your git log as a graph

**git revert <--flag>**

The git revert command can be considered an 'undo' type command, however, it is not a traditional undo operation. Instead of removing the commit from the project history.

**git revert -n <commit 7 first char hash\_code>**

revert the commit to given hash\_code commit

**git reset <--flag>**

The git reset command is a complex and versatile tool for undoing changes. It has three primary forms of invocation. These forms correspond to command line

**git reset <commit 7 first char hash\_code>**

to go to back the given commit but id do not removes the commits/changes made in repo.

**git reset <commit 7 first char hash\_code> --hard**

to go to back the given commit and it remove the changes made in repo after that commit.

Check below videos for revert and rest

<https://www.youtube.com/watch?v=OGk5rvYw8c0>

<https://www.youtube.com/watch?v=hA3vYDyaZQQ>

Note:

We can combine flags also. e.g. -- git log --graph --oneline

**list all branch**

We can run below command to list out all branch from current repo.

**git branch** **/ git branch --list**

------ list all branch name. where ever it shows \*, that refers the branch you are.

$ git branch

\* master -- \* indicates that we are on master branch and there are two master and oldies branch

oldies

**Create new branch**

We can use below command to create new branch.

git branch <branch name> ---- to create new branch by <branch name>

rakes@LAPTOP-IK629TG3 MINGW64 /d/gitlearn (master)

$ git branch ---- only master branch here

\* master

rakes@LAPTOP-IK629TG3 MINGW64 /d/gitlearn (master)

$ git branch oldies --- created oldies branch

rakes@LAPTOP-IK629TG3 MINGW64 /d/gitlearn (master)

$ *git branch -------- now showing two branch*

*\* master*

*Oldies*

**Delete branch**

We can use below command to delete branch

git branch -d <branch name> --- to delete branch <branch name>

rakes@LAPTOP-IK629TG3 MINGW64 /d/gitlearn (master)

$ git branch --- two branch here

\* master

oldies

rakes@LAPTOP-IK629TG3 MINGW64 /d/gitlearn (master)

$ git branch -d oldies

Deleted branch oldies (was 81496ed).

rakes@LAPTOP-IK629TG3 MINGW64 /d/gitlearn (master)

$ git branch --- now only master branch b/c deleted oldies branch above

\* master

rakes@LAPTOP-IK629TG3 MINGW64 /d/gitlearn (master)

**Delete a Remote Branch**

We can delete a remote branch from Git desktop application. Below command is used to delete a remote branch:

git push origin -delete <branch name>

**Switching to any branch**

We can use switch/checkout command to switch the branches.

git branch <branch name> --- switch to branch name

git checkout <branch name> --- switch to new branch <branch name>

Note:

Checkout is old command for switching newer is switch

**Rename branch**

We can use below command to rename branch

git branch -m <old branch name> <new branch name> --- rename old branch name to new branch name

$ git branch

master

\* oldies

rakes@LAPTOP-IK629TG3 MINGW64 /d/gitlearn (oldies)

$ git branch -m oldies retro --- renaming oldies to retro

rakes@LAPTOP-IK629TG3 MINGW64 /d/gitlearn (retro)

$ git branch

master

\* retro

**Merge Branch**

Git allows you to merge the other branch(<branch name>) with the currently active(where you are) branch. You can merge two branches with the help of git merge command.

git merge <branch name>

**Git diff**

It's a multiuse Git command. When it is executed, it runs a diff function on Git data sources.

It list all the changes in our working directory that are NOT staged for the next commit.

Example –

Scenerio1: Track the changes that have not been staged. ---- git diff

Scenerio2: Track the changes that have staged but not committed --- git diff --staged

Scenerio3: Track all the changes stagged/unstagged in a file --- git diff HEAD

Scererio4: Trach changes b/w two commits --- git diff <commit1-sha> <commit2-sha>

Scenerio5:Comparing branched --- git diff branch1..branch2

Scenerios6: Diff-ing specific file --- git diff –staged filename (diffing stagged <filename>)

git diff ----- compared Staging area and working directory

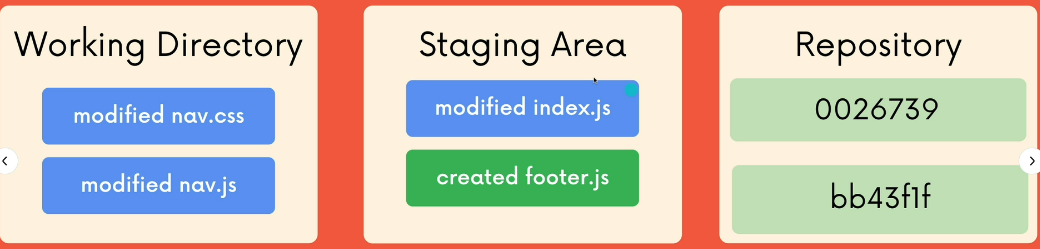
**Stashing / git stash**

Stashing is a Git function that allows you to temporarily save your file changes without applying them to your project.

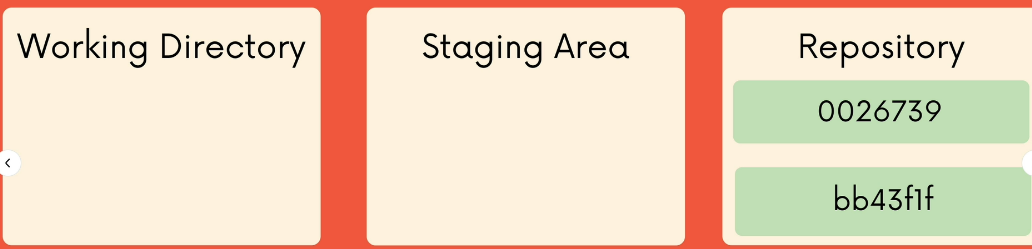
Let’s take an example: -

Suppose we are working on branch 'b1' which is incomplete and there is need to switch to other branch, as b1 is incomplete so don’t want to commit b1. Here we can use stashing to stash branch b1 and switch to another branch.

Let say we are on one branch which have some data in difference are before stashing

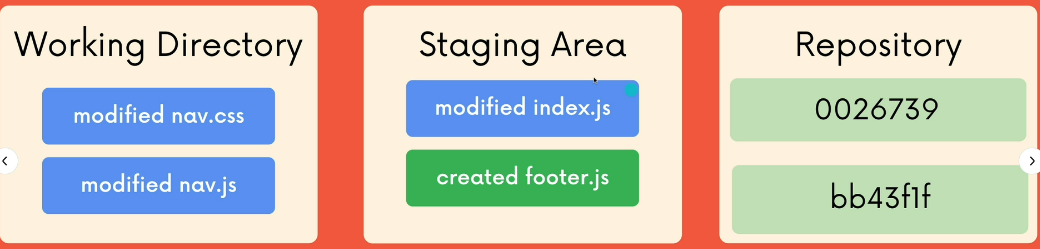


After stashing **( git stash**)



Here all data of staging are, working directory will be moved to stash and we can get those changes back

After popping the stash (**git stash pop**), all data will be retrived



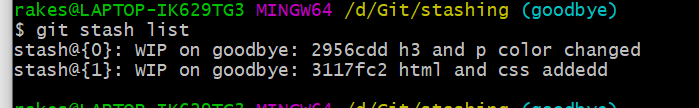
git stash / git stash save ------------- to stash the branch or working directory

git stash pop --------------------------- to pop out the stashed data

**Other stashing commands: -**

1. git stash list ---- to check all stashes that we have done on current branch”

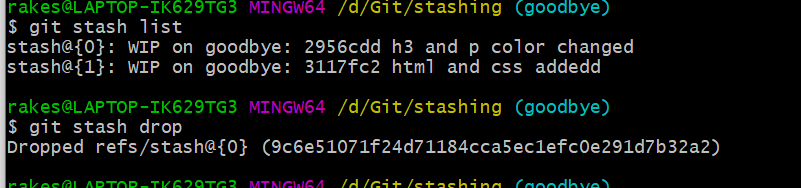
let say we I have done stashing two times in goodbye branch, then we will get below –



stash@{0} , stash@{1} ---- there are stash ids

1. git stash drop / git stash drop <stash id> -- to drop the most recent stash or using stash id

If we drop stash then only stash id are deleted but there will be no changes in data on that branch.



1. git stash clear ------- deleting all stashes or stash ids at once

**git checkout /detaching head**

We can use checkout for many work , e.g- time travel, detaching any commit, deleting commit, switching to branches

git checkout <commit hash>

this command allows us to see the data it had before that commit. If we runt his commit then we get message like 'detached HEAD'.

git checkout HEAD~1 ------ another way to go to one(immediate0 commit back

Note:

It doesn’t delete the commit data but it take us the state what it was before that commit hash.

**Questions:**

Suppose we on a branch we have done 10 commits then then was to create a new branch from where second commit was run.

**Solution**

git checkout <commit hash> # commit has of second commit

git branch <new branch name> --- #here new\_branch will be created and we can work

git switch <new\_branch> -------- # Switch to new branch

**Reattaching detached head**

Just switch to branch it HEAD will be attached.

**Discarding changes**

We can discard the changes that are made to any file which are made after the most recent commit.

git checkout HEAD <file\_name> --- to discard the changes

git checkout -- <file(s)> ---- alternate cmd to discard the changes

**git restore <file\_name>** ---- alternate cmd to discard the changes, **restore to recent commit**

git restore --source HEAD~n <file\_name> --- restore to to nth commit from current commit for file

Example:

git restore –source HEAD~1 check.txt --- restore to one commit back from current commit.

git restore check.txt ---- restore to most recent commit

**Note:**

Above commands will not cause loose of any commit data, it’s just to switch to particular commit

Example

Let say our working tree is clean and nothing to add or commit in branch goodbye

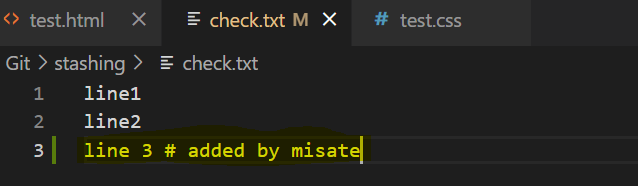
rakes@LAPTOP-IK629TG3 MINGW64 /d/Git/stashing (goodbye)

$ git status

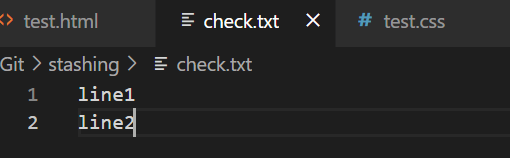
On branch goodbye

nothing to commit, working tree clean

Now accidently added/messed up some data into check.txt file but not commited



We can discard that data using checkout, after running cmd - git checkout HEAD checkout.txt



**git reset <commit hash>**

This is used to go back to particular commit level.

It carried all commit that were made even after that commit had but git restore/checkout doesn’t carry commits that were made after that commit.

**git reset**

It is used to resetting the commits, this cause data chages.

git reset <commit hash>

this is used to go back to particular commit hash. This causes the loss of data.

All committed data after that commit hash will be lost.