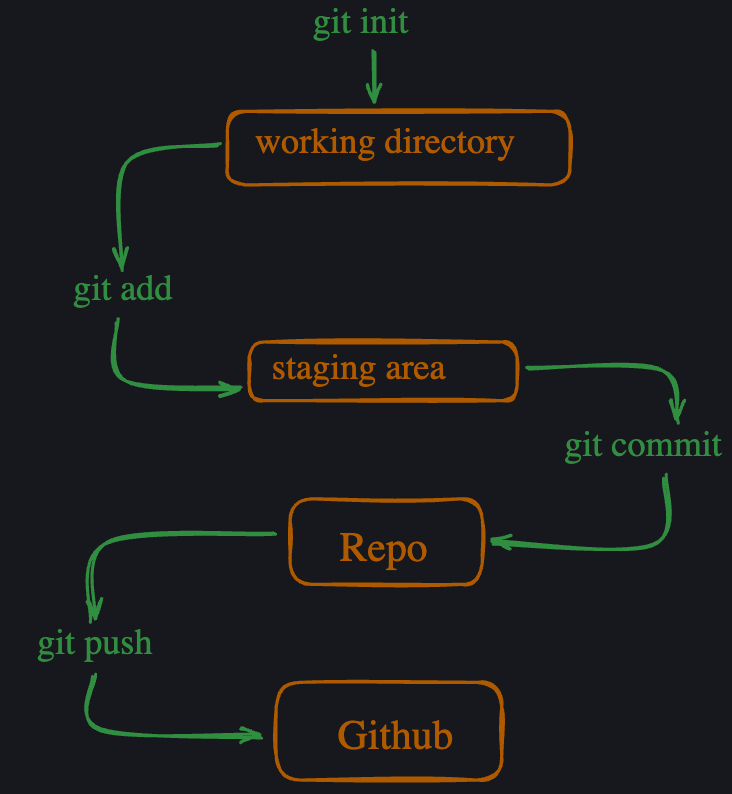
**git init-**

* Convert an existing, unversioned project to a Git repository. Run only once for a project.
* It creates a hidden .git folder which keep track of all files and sub-folders.
* To initialize git project with some other branch names ishan: git init -b ishan



**Head**

We can see all the heads inside .git/refs/heads, it will show names of all branches. Hence every branch head is the tip of that branch

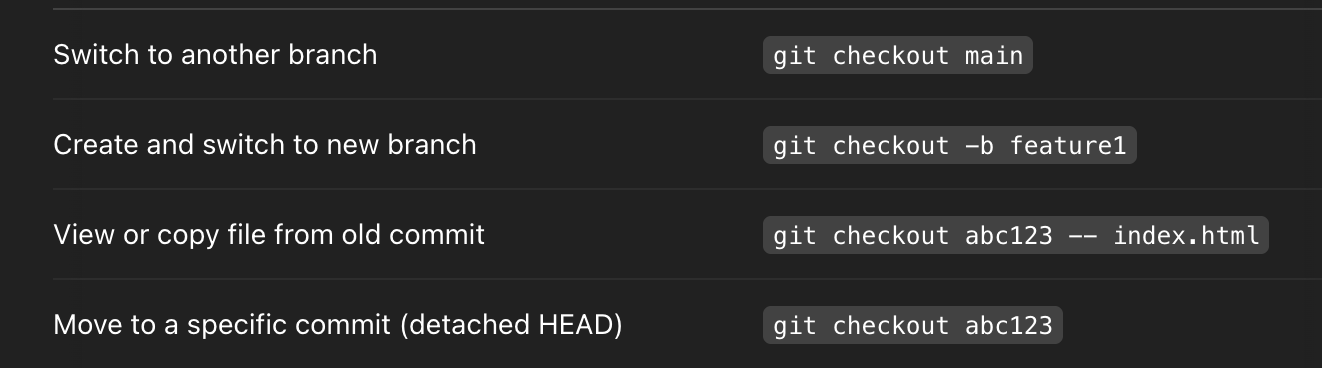
Current head always points to the tip of current branch, it can also be seen in Head file inside .git

**git add**

It does 2 things - make file tracked, stages the changes in the file

Now if I make any further changes in file - the file is tracked but new changes are not staged. To stage new changed, I need to run “git add” again.  
  
Untracked → git add → Tracked + Staged  
Tracked + Changed → Modified  
Modified → git add → Staged

**git checkout**



“A detached HEAD occurs when you check out a specific commit or tag instead of a branch. If you commit changes in detached HEAD state, the commits are not saved to any branch. To keep these changes, you must create a new branch using "git checkout -b new-branch"

Q1. What is .gitkeep file?

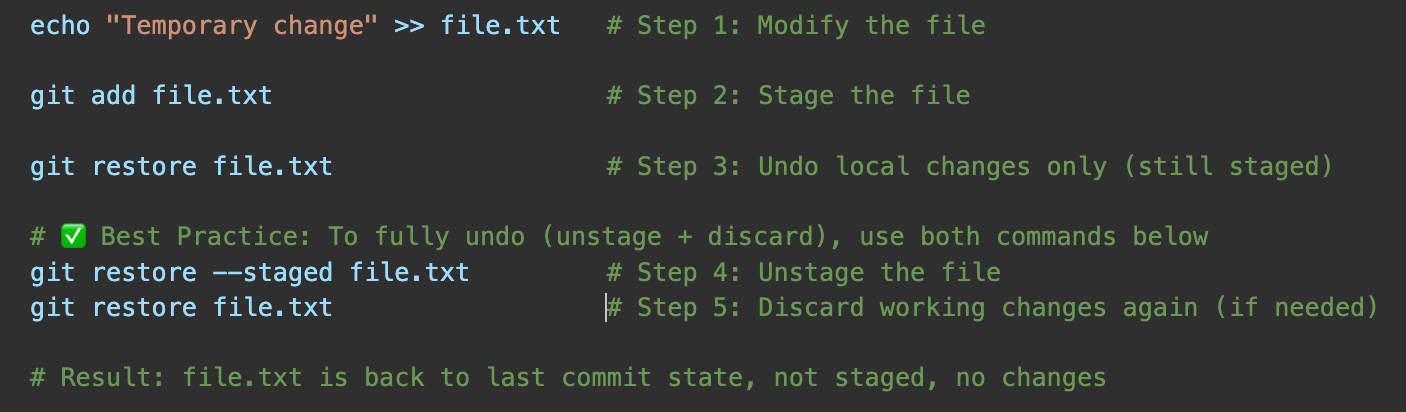
Ans- Git cannot add a completely empty directory. People who want to track empty directories in Git have created the convention of putting files called .gitkeep in these directories.

Q2. What is git restore & git restore --staged v/s git add?

Ans-

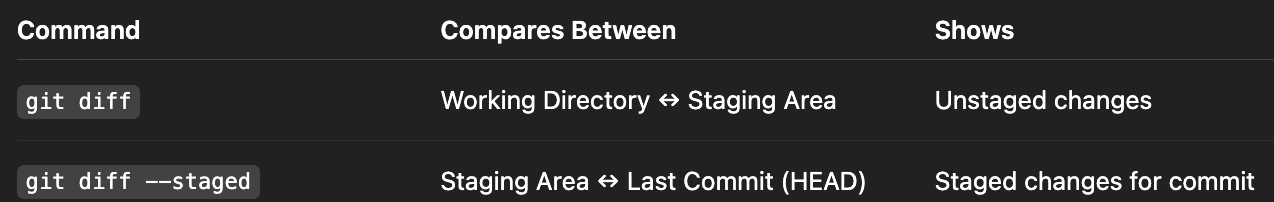
git restore : Reset file to the last committed state, hence it undo the changes you made in file, takes back to last committed version in local.

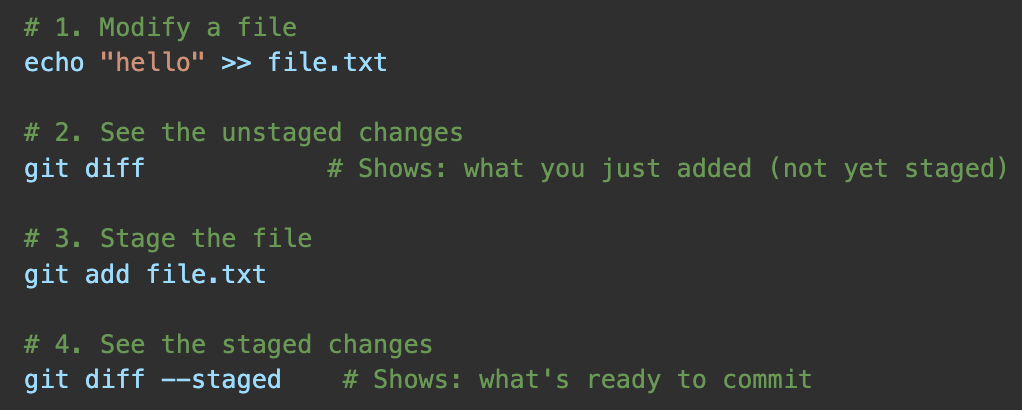
git restore --staged : It is completely opposite of git add. Hence after doing git add, if we want to unstaged a file, we use git restore --staged. It does not undo the changes you made in the file, it only unstages it.



**git diff** : Shows the changes you made in your working directory compared to the last staged. You can see what is modified but not yet staged.

git diff --staged : Shows the staged changes, yet to be committed.





git diff main feature1 : shows changes done in feature1 branch that are not in main branch (used before merging feature1 to main branch)

git diff 45347 987864 : compare 2 commits

**git log**

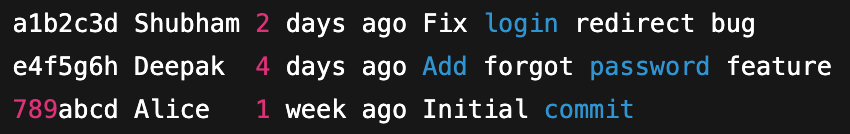
git log -p : shows the complete commit history with code changes (diffs) for every commit. Displays what code was added/removed in each commit.

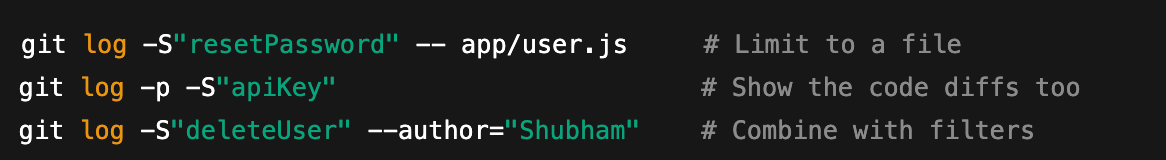
git log -p -2 : shows last 2 commit history with code changes.

git log --stat : shows summary of the changes with file name (+2 insertions, -3 deletion). it doesn’t show the actual changes done in those file like git log -p does

git log --oneline : shows all hash + commitMsg, each in one line

git log --pretty=format:”%h %an %ar %s”



git log -S "search\_term" : shows commits where a given string was added or deleted

git log --since="2025-05-01"

git log --author="deepak mishra"

git log --since="1 week ago" --author="shubham"

**Git branch and merge**

1. What does origin/master and master mean?

- origin = Git's name for the remote version of the repo (like GitHub, GitLab, etc.)

origin/master = The master branch on the remote repo.

master = This is local branch of our git on local system.

1. What are git conflicts and how to resolve it?

- If I want to merge feature1 branch to main branch. Then do “git checkout main” -> “git merge feature1”

- git merge combines branches. If both change the same line, Git can't auto-merge — it shows a conflict. I manually fix the file, add it (git add fileX), then commit the merge to complete the process."

1. How to delete a branch from local and remote?

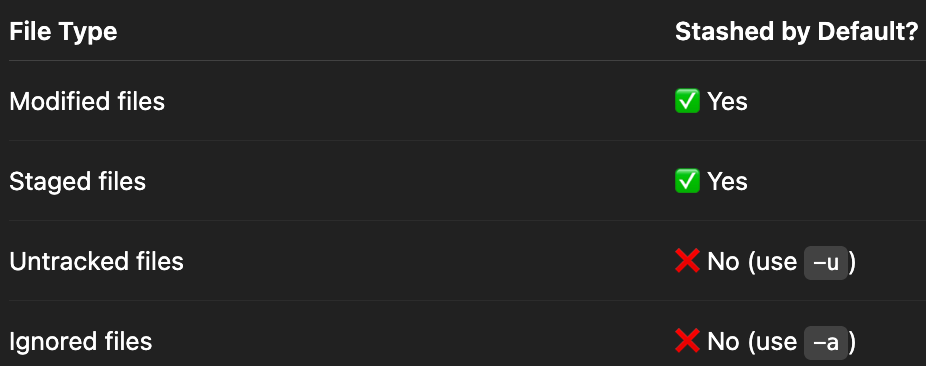
- git branch -d feature1 #Delete locally (git allows this only if feature1 is merged to any branch before deleting, otherwise do force delete by “git branch -D feature1”)

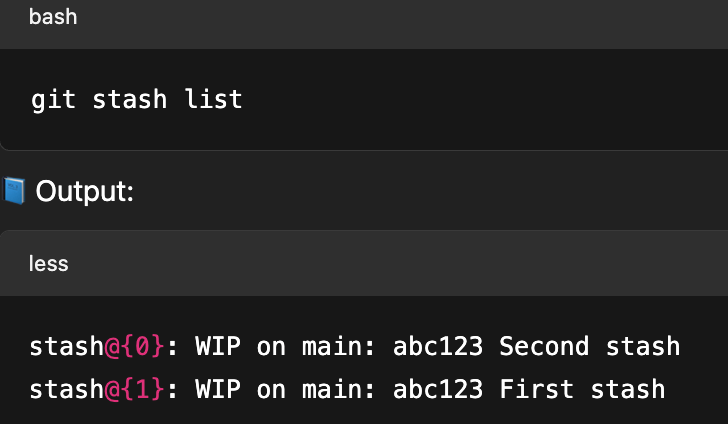
- git push origin --delete feature1 # Delete from remotegit branch -d feature1

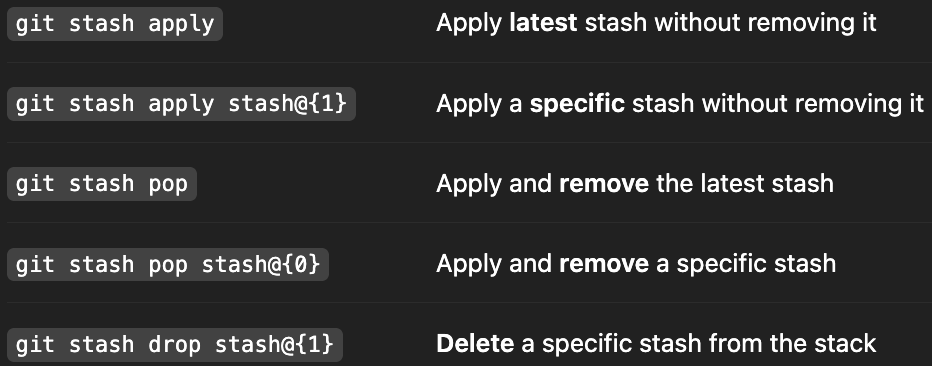
1. git push origin feature1 #push my local feature1 branch to remote

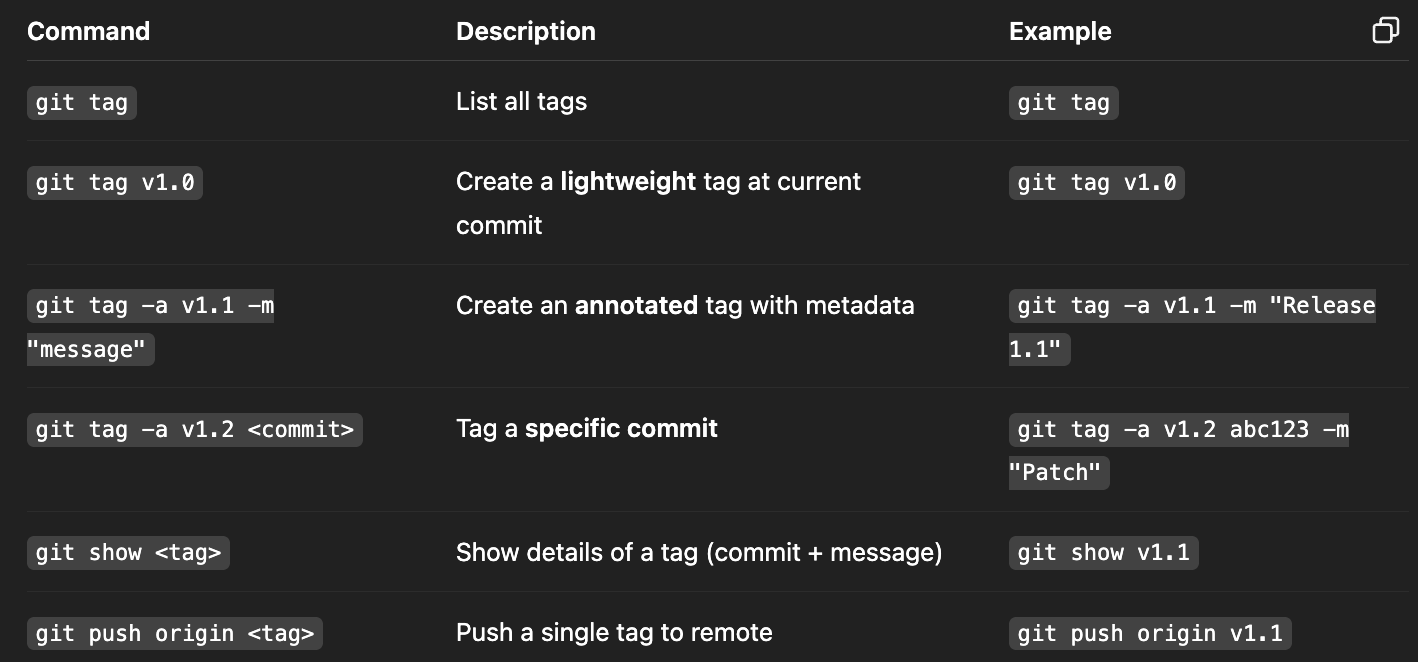
**git stash**

git stash temporarily saves your uncommitted changes (both staged and unstaged) to a stash stack, so you can work on something else (like switching branches) without committing them.







**git tag** : used to mark specific commits in Git history — commonly for releases, like v1.0, v2.3  
  


**git rebase** : Used to move a series of commits from one base to another, creating a cleaner, linear commit history. It’s especially helpful for incorporating the latest changes from the main branch into a feature branch without creating merge commits.

- Rebase rewrites commit history.

- I usually rebase my local feature branches before pushing to remote, but for remote branches (when that feature branch is already pushed to remote), I prefer merging to avoid git force push.

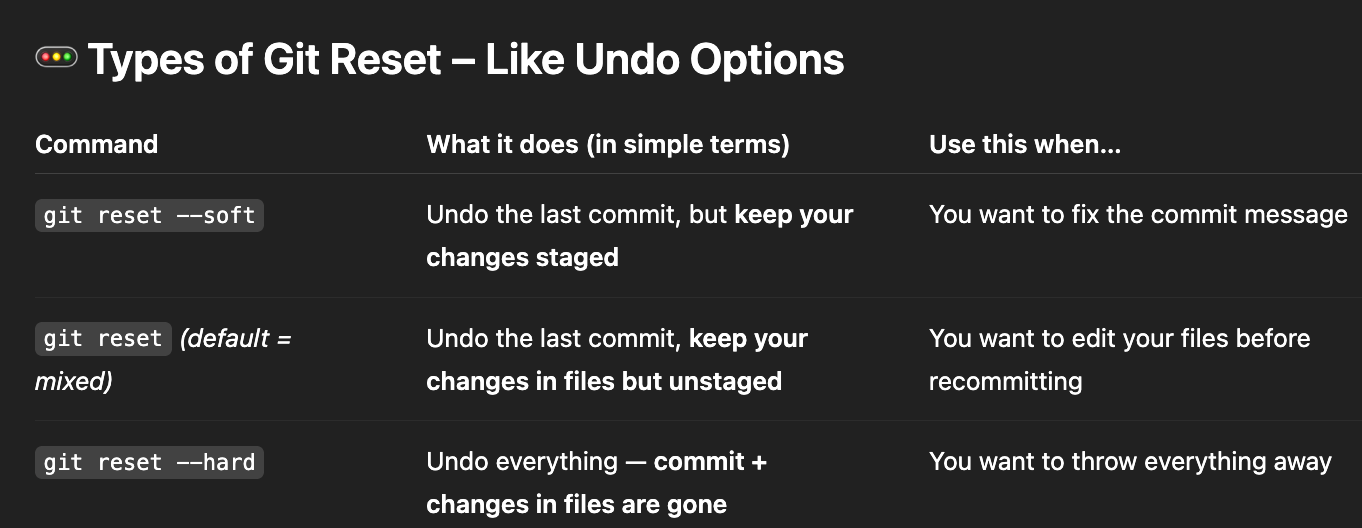
- To rebease a branch, go to that branch, then do “git rebase main”

**git push --force** : It overwrites the remote branch with your local branch, even if the commit history is different. Usually used after git rebase, git commit --amend when git push fails.

- While working in shared branches, safer option is using “git push --force-with-lease”. It check “Has the remote changed since I last pulled it?” If yes, it refuses to overwrite.

1. What is fast forward merge in git ?

- A fast-forward merge in Git occurs when the current branch can simply be moved to the target branch's latest commit, hence no merge commit is created. It keeps history clean but can hide the fact that a feature branch existed. To enforce clarity, teams often use --no-ff to preserve merge history.

**git reset** : changes the head of a branch to any previous commit id  




**git fetch** (*git fetch origin feature1)*: download changes from remote to local. then we can check diff using “git diff main origin/main” and then merger using “git merge origin/main”.

**git pull** (*git pull origin feature1*): fetch + merge.   
(git pull --rebase origin feature1): fetch + rebase