BASICS OF GIT

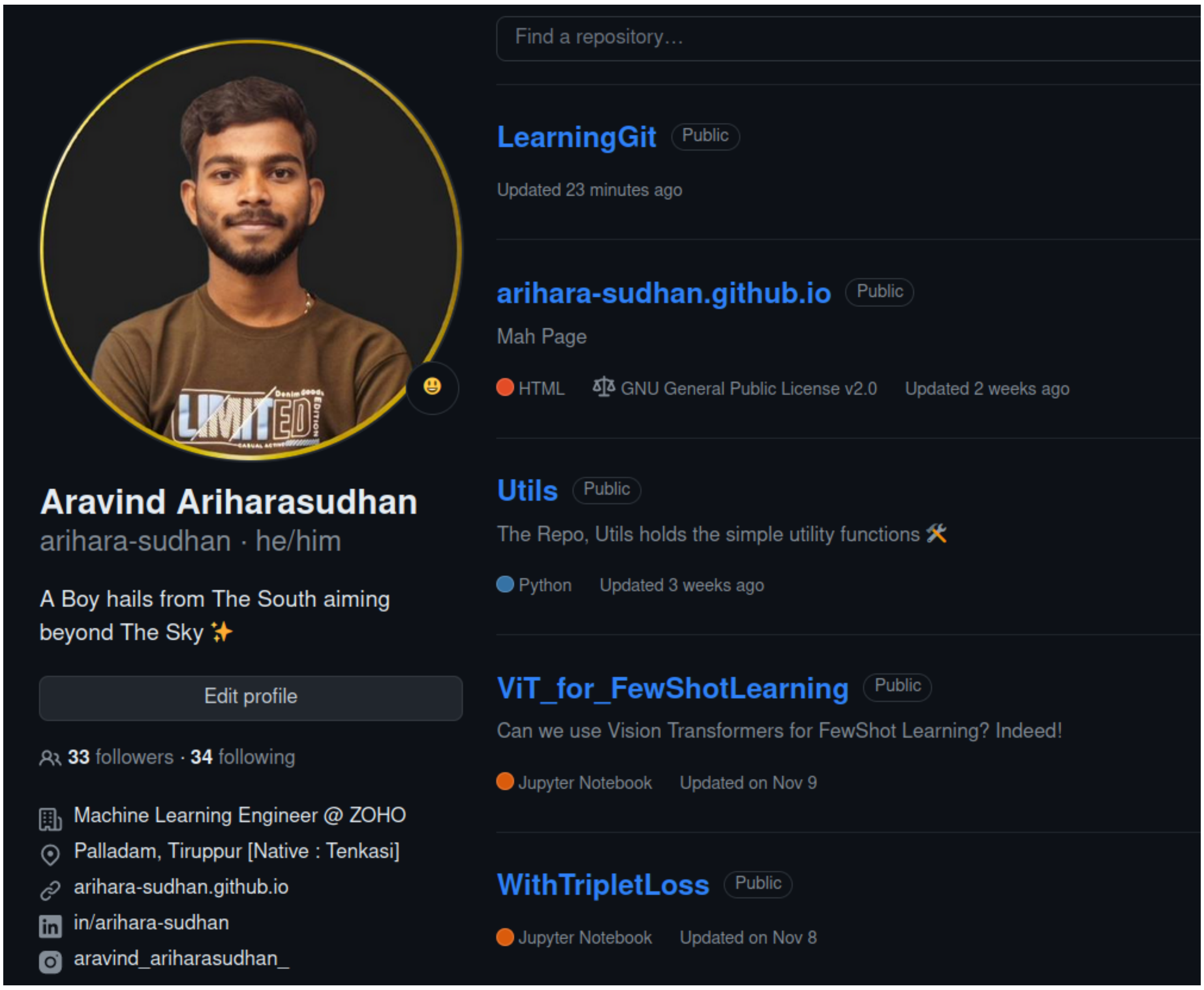
ARIHARASUDHAN

GIT BASICS

Git is a distributed version control system (DVCS) designed to track changes in source code during software development. Git helps manage and track changes to source code over time. It allows multiple developers to collaborate on a project and keeps a record of every change made to the codebase. Unlike centralized version control systems, Git is distributed. Each developer has a complete copy of the repository, including the entire history of changes. This makes Git more robust and flexible.

Repository

A Git repository is a collection of files and the entire history of changes to those files. It can be local (on a developer's machine) or remote (on a server), allowing for collaboration between developers. You can check out my repositories.


Find a repository…

Aravind Ariharasudhan
arihara-sudhan · he/him

A Boy hails from The South aiming beyond The Sky ✨

Edit profile

33 followers · 34 following

Machine Learning Engineer @ ZOHO
Palladam, Tiruppur [Native : Tenkasi]
arihara-sudhan.github.io
in/arihara-sudhan
aravind_ariharasudhan_

LearningGit Public
Updated 23 minutes ago

arihara-sudhan.github.io Public
Mah Page
● HTML GNU General Public License v2.0 Updated 2 weeks ago

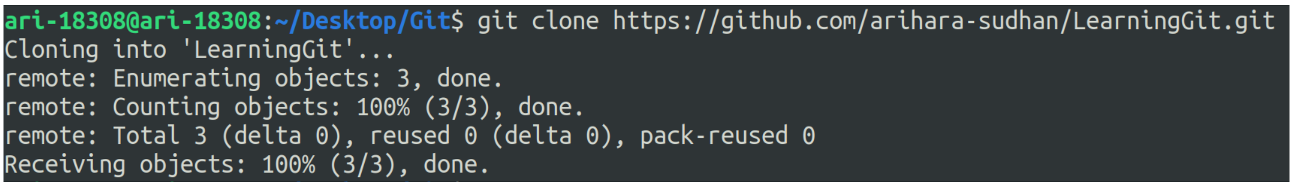
Utils Public
The Repo, Utils holds the simple utility functions 🛠
● Python Updated 3 weeks ago

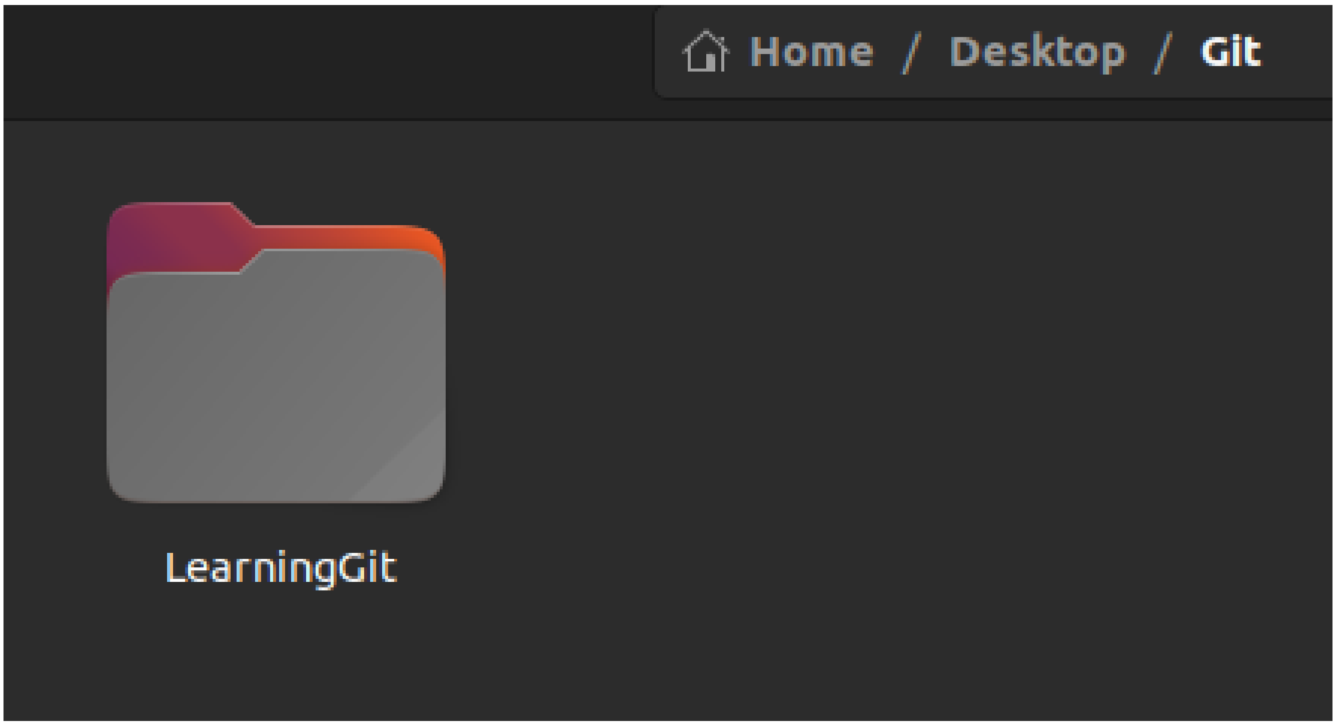
ViT_for_FewShotLearning Public
Can we use Vision Transformers for FewShot Learning? Indeed!
● Jupyter Notebook Updated on Nov 9

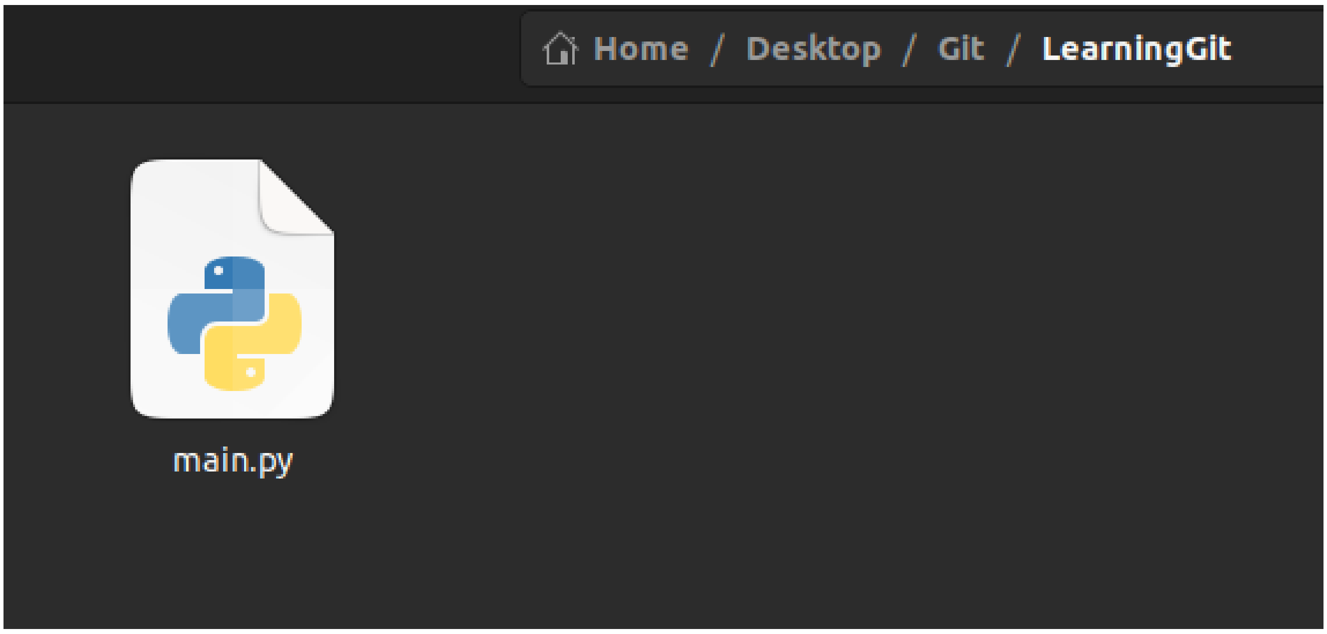
WithTripletLoss Public
● Jupyter Notebook Updated on Nov 8

Clone

Cloning is the process of creating a copy of a Git repository on a local machine. This is typically done to work on a project or contribute to it. The below example is the clone request to clone the repository LearningGit into local machine.


```
ari-18308@ari-18308:~/Desktop/Git$ git clone https://github.com/arihara-sudhan/LearningGit.git
Cloning into 'LearningGit'...
remote: Enumerating objects: 3, done.
remote: Counting objects: 100% (3/3), done.
remote: Total 3 (delta 0), reused 0 (delta 0), pack-reused 0
Receiving objects: 100% (3/3), done.
```

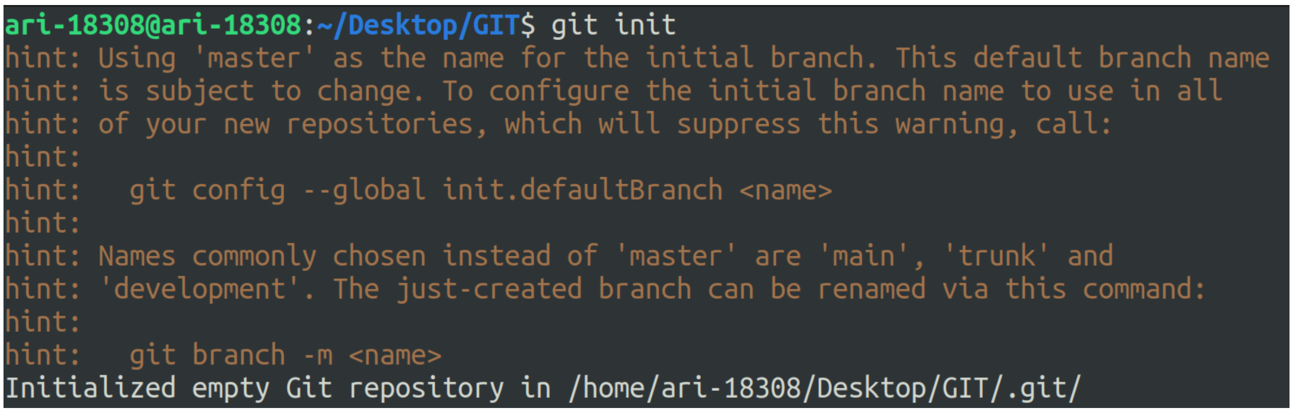




Add

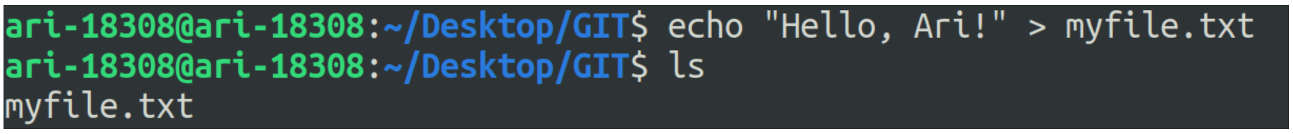
The git add command is used to stage changes for the next commit in Git. Staging changes means you've marked them as ready to be included in the next commit.

To demonstrate this, let's create an empty folder and make it a local git repository using git init.



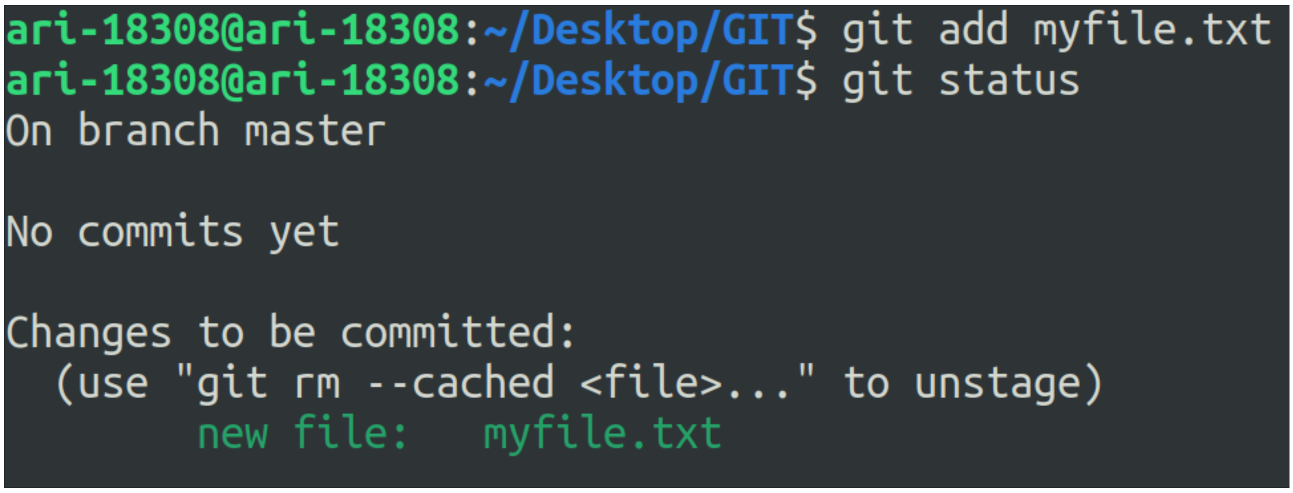
```
ari-18308@ari-18308:~/Desktop/GIT$ git init
hint: Using 'master' as the name for the initial branch. This default branch name
hint: is subject to change. To configure the initial branch name to use in all
hint: of your new repositories, which will suppress this warning, call:
hint:
hint:   git config --global init.defaultBranch <name>
hint:
hint: Names commonly chosen instead of 'master' are 'main', 'trunk' and
hint: 'development'. The just-created branch can be renamed via this command:
hint:
hint:   git branch -m <name>
Initialized empty Git repository in /home/ari-18308/Desktop/GIT/.git/
```

Now, let's create a text file myfile.txt inside this repository.



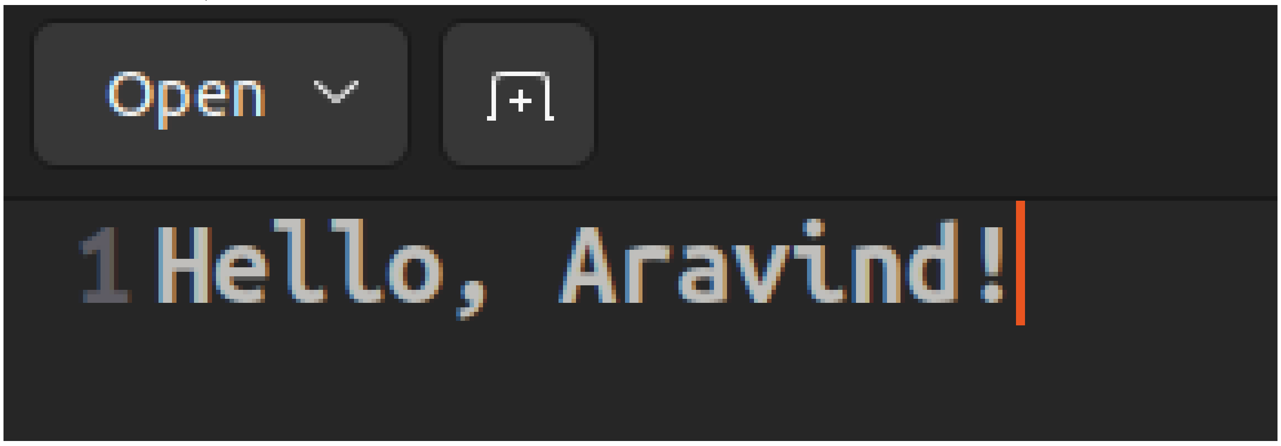
```
ari-18308@ari-18308:~/Desktop/GIT$ echo "Hello, Ari!" > myfile.txt
ari-18308@ari-18308:~/Desktop/GIT$ ls
myfile.txt
```

This file is not added to changes tracking. To do that, we use git add.



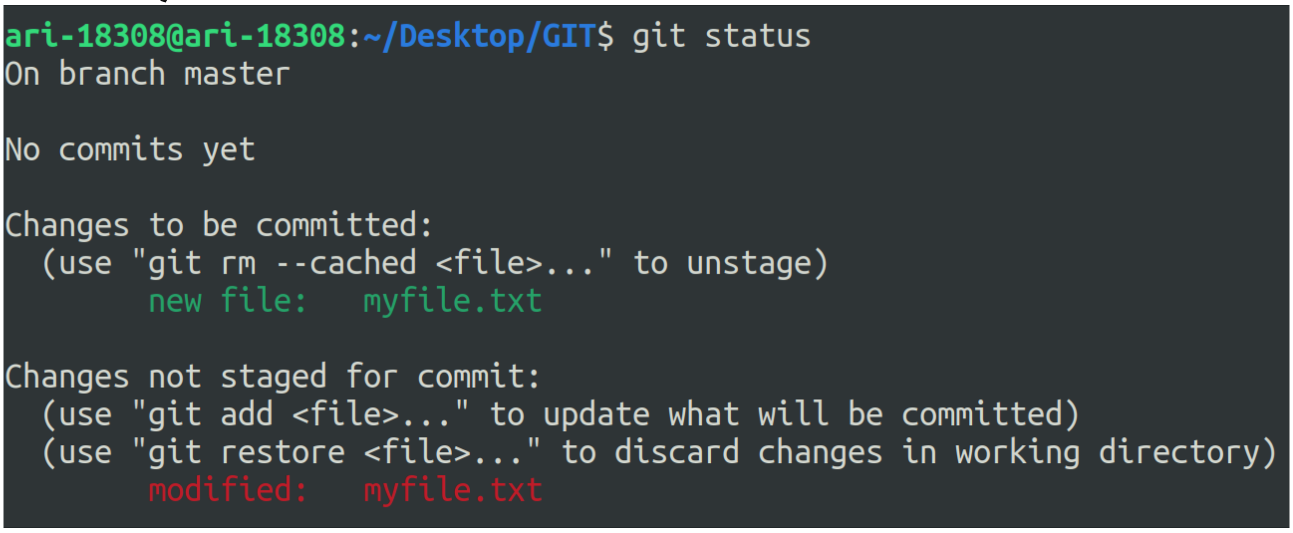
```
ari-18308@ari-18308:~/Desktop/GIT$ git add myfile.txt
ari-18308@ari-18308:~/Desktop/GIT$ git status
On branch master

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
        new file:   myfile.txt
```

Now, let's change the contents of the file and check the status. Ari is changes to Aravind (using A Text Editor).



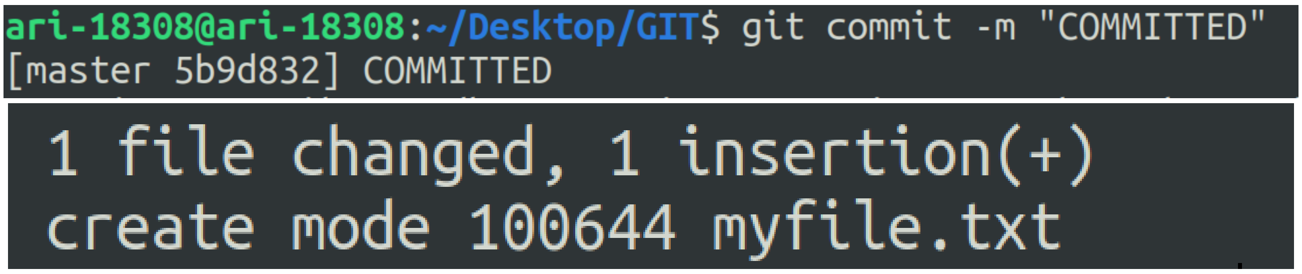
```
1 Hello, Aravind!
```

Now,

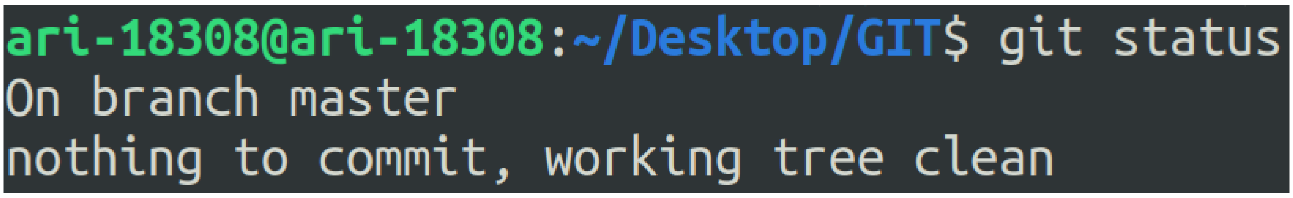


```
ari-18308@ari-18308:~/Desktop/GIT$ git status
On branch master

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
        new file:   myfile.txt

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:   myfile.txt
```

Our file is under changes to be committed which means it is in staging area and ready to be committed.



```
ari-18308@ari-18308:~/Desktop/GIT$ git commit -m "COMMITTED"
[master 5b9d832] COMMITTED
 1 file changed, 1 insertion(+)
 create mode 100644 myfile.txt
```

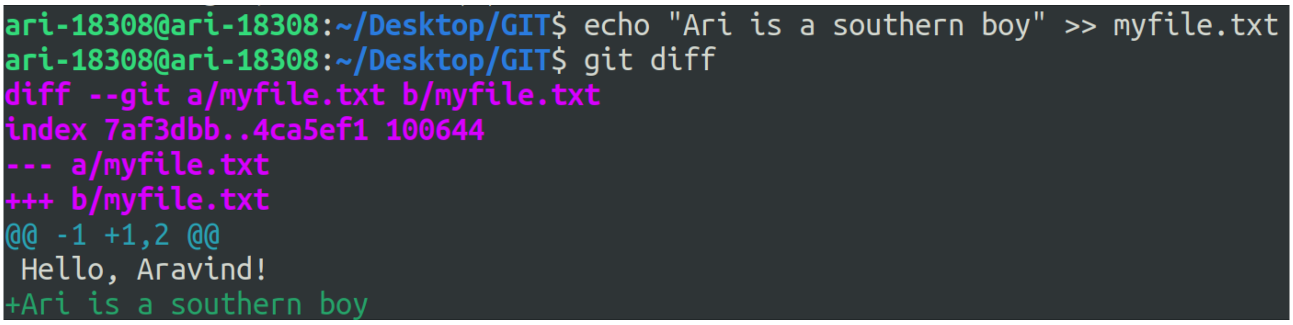


```
ari-18308@ari-18308:~/Desktop/GIT$ git status
On branch master
nothing to commit, working tree clean
```

Similarly, we can use git add <directory> (for staging specific directory).

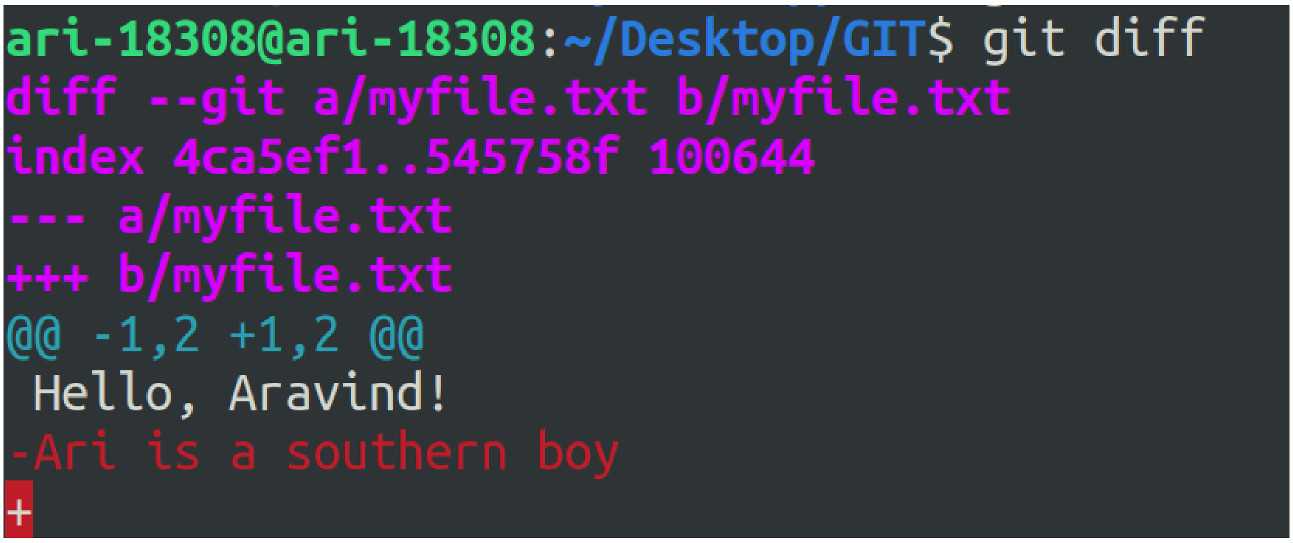
Diff

Now, let's add a text of "Ari is a southern boy" in myfile.txt. After doing that, just type git diff.



```
ari-18308@ari-18308:~/Desktop/GIT$ echo "Ari is a southern boy" >> myfile.txt
ari-18308@ari-18308:~/Desktop/GIT$ git diff
diff --git a/myfile.txt b/myfile.txt
index 7af3dbb..4ca5ef1 100644
--- a/myfile.txt
+++ b/myfile.txt
@@ -1 +1,2 @@
 Hello, Aravind!
+Ari is a southern boy
```

Here, git diff has pointed out that the text "Ari is a southern boy" is added (+) to myfile.txt. Now, let me remove the last line and check what diff gets.

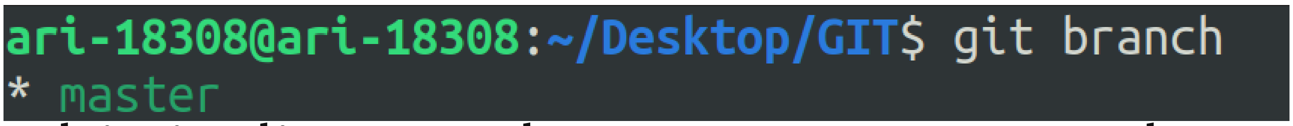


```
ari-18308@ari-18308:~/Desktop/GIT$ git diff
diff --git a/myfile.txt b/myfile.txt
index 4ca5ef1..545758f 100644
--- a/myfile.txt
+++ b/myfile.txt
@@ -1,2 +1,2 @@
 Hello, Aravind!
-Ari is a southern boy
+
```

git diff is simply a Git command that allows us to see the differences between different states of our Git repository. It can be used to compare changes between commits, branches, working directory, and more. Don’t forget to commit each time you change the file.

Branch

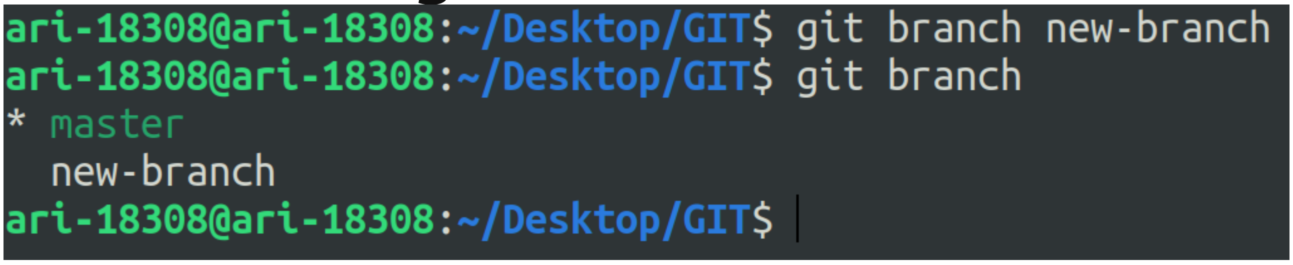
A branch is a lightweight movable pointer to a specific commit. It allows to work on different features, bug fixes, or experiments independently of each other. Let's say we have a Git repository and we want to add a new feature. We decide to create a branch for this feature. (Who wants to touch the working product?) Let’s open a terminal and navigate to our Git repository. Use the following command to see the current branch.



```
ari-18308@ari-18308:~/Desktop/GIT$ git branch
* master
```

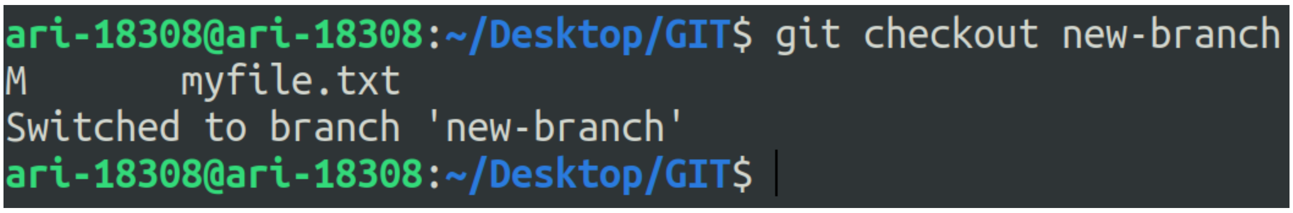
This indicates that we are currently on the master branch.

To create a new branch, we can use the following command.



```
ari-18308@ari-18308:~/Desktop/GIT$ git branch new-branch
ari-18308@ari-18308:~/Desktop/GIT$ git branch
* master
  new-branch
ari-18308@ari-18308:~/Desktop/GIT$
```

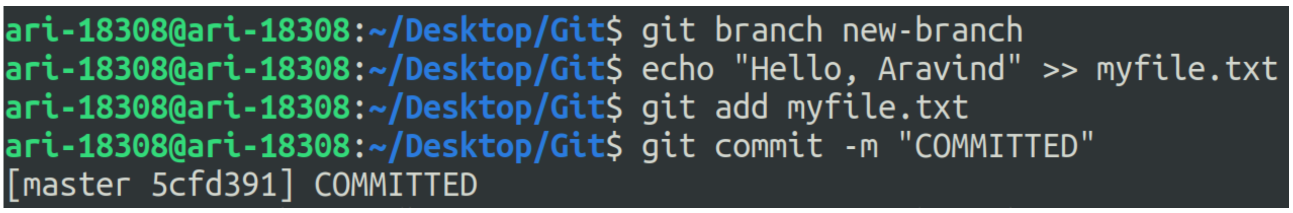
The * indicates that we are currently on master branch. We are yet to switch to new-branch. To switch to the new-branch, use the check out command as shown below.



```
ari-18308@ari-18308:~/Desktop/GIT$ git checkout new-branch
M       myfile.txt
Switched to branch 'new-branch'
ari-18308@ari-18308:~/Desktop/GIT$
```

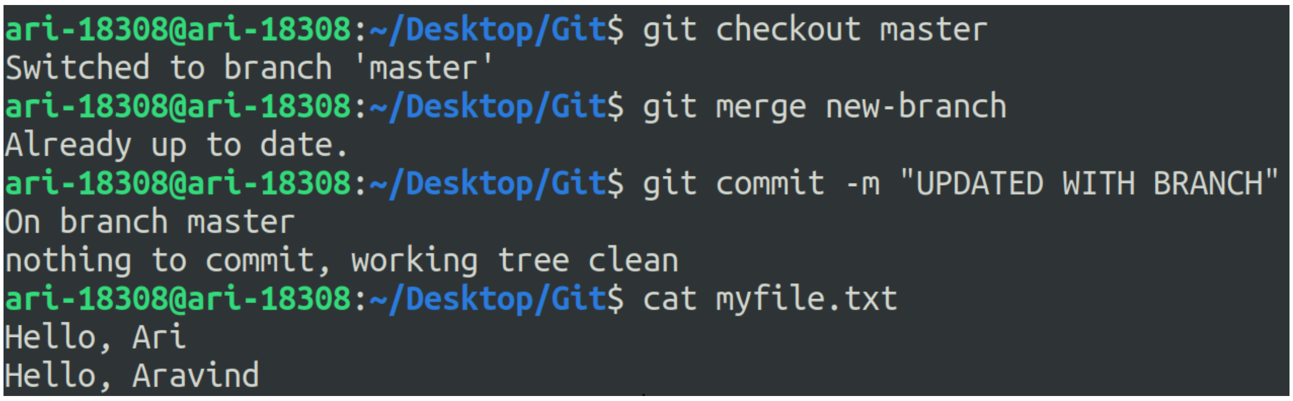
We can also combine the branch creation and checkout in one command, git checkout -b new-branch

Now, we are on the new-branch. Let’s make changes to our text file as needed for the new feature. I just append a text of “Hello, Aravind”. After making changes, commit them to the branch.



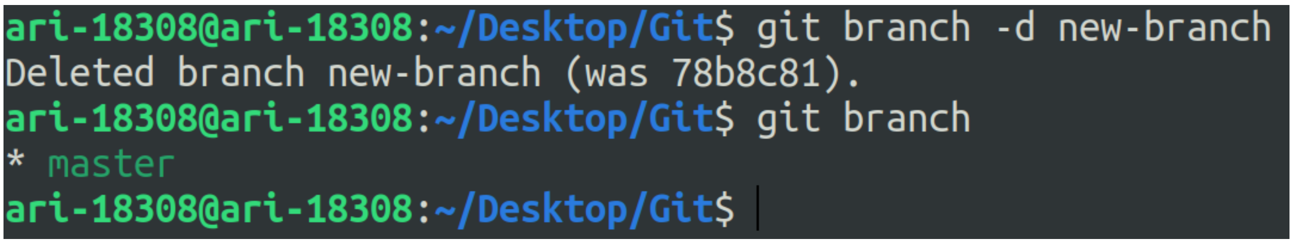
```
ari-18308@ari-18308:~/Desktop/Git$ git branch new-branch
ari-18308@ari-18308:~/Desktop/Git$ echo "Hello, Aravind" >> myfile.txt
ari-18308@ari-18308:~/Desktop/Git$ git add myfile.txt
ari-18308@ari-18308:~/Desktop/Git$ git commit -m "COMMITTED"
[master 5cfd391] COMMITTED
```

Once the feature is implemented and committed, we can switch back to the master branch. To incorporate the changes from the new-branch into master, we need to merge them using git merge command. This combines the changes from new-branch into the master branch.



```
ari-18308@ari-18308:~/Desktop/Git$ git checkout master
Switched to branch 'master'
ari-18308@ari-18308:~/Desktop/Git$ git merge new-branch
Already up to date.
ari-18308@ari-18308:~/Desktop/Git$ git commit -m "UPDATED WITH BRANCH"
On branch master
nothing to commit, working tree clean
ari-18308@ari-18308:~/Desktop/Git$ cat myfile.txt
Hello, Ari
Hello, Aravind
```

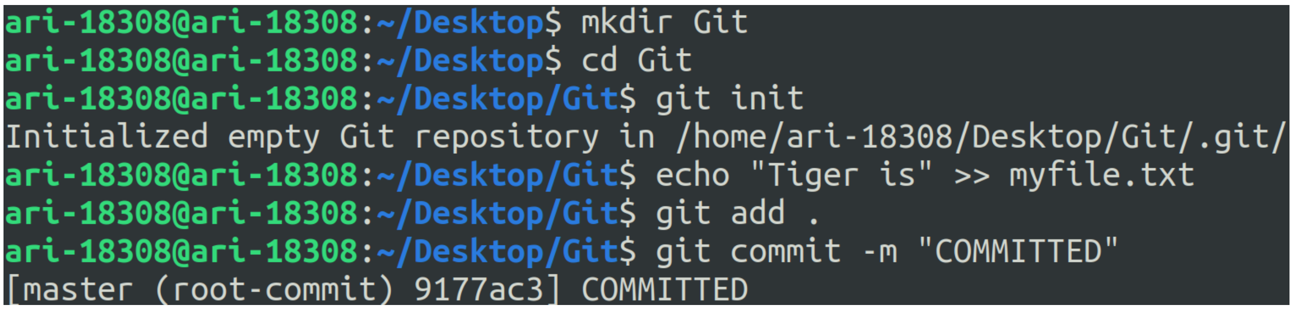
After merging, if we don't need the new-branch anymore, we can delete it using git branch -d.



```
ari-18308@ari-18308:~/Desktop/Git$ git branch -d new-branch
Deleted branch new-branch (was 78b8c81).
ari-18308@ari-18308:~/Desktop/Git$ git branch
* master
ari-18308@ari-18308:~/Desktop/Git$
```

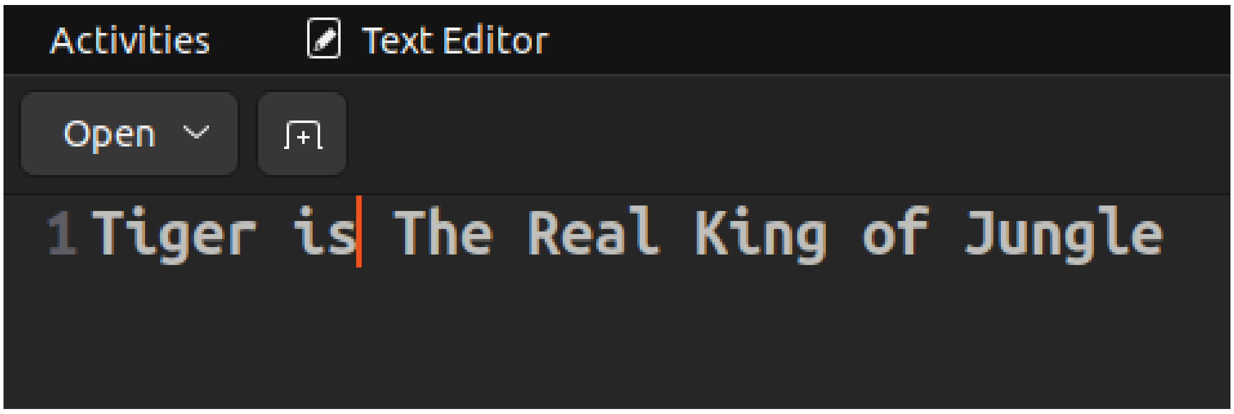
Stash

The git stash command is used to temporarily store changes that have not yet been committed to a permanent branch. It allows developers to switch to another branch or perform other actions without losing their changes. To understand this, let's create an empty folder Git and git init it. Write a text of "Tiger is " inside a text file named file.txt. Commit it (after adding).



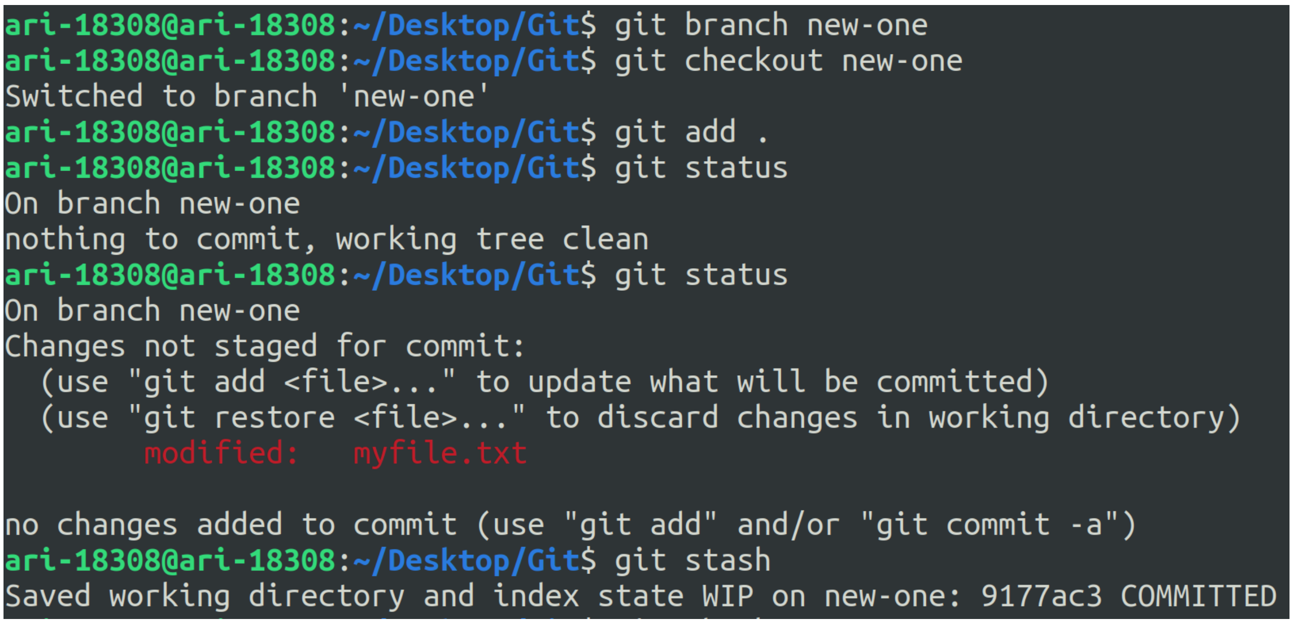
```
ari-18308@ari-18308:~/Desktop$ mkdir Git
ari-18308@ari-18308:~/Desktop$ cd Git
ari-18308@ari-18308:~/Desktop/Git$ git init
Initialized empty Git repository in /home/ari-18308/Desktop/Git/.git/
ari-18308@ari-18308:~/Desktop/Git$ echo "Tiger is" >> myfile.txt
ari-18308@ari-18308:~/Desktop/Git$ git add .
ari-18308@ari-18308:~/Desktop/Git$ git commit -m "COMMITTED"
[master (root-commit) 9177ac3] COMMITTED
```

Now, let's create a new branch named "new-one" and checkout to that. In new-one branch, I change the text as "Tiger is The Real King of The Jungle". I don't commit here.



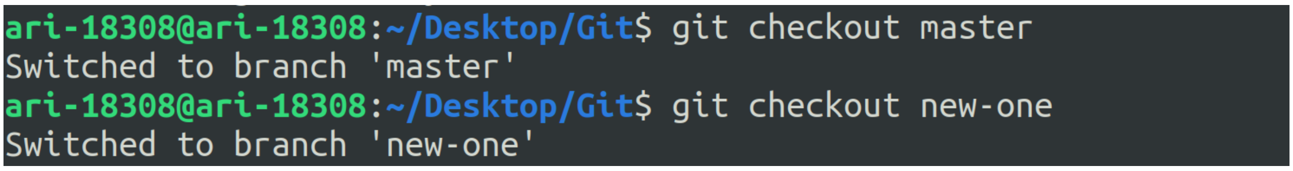
```
Tiger is The Real King of Jungle
```

If we don’t commit, we may lose our changes. Here, I simply stash this.



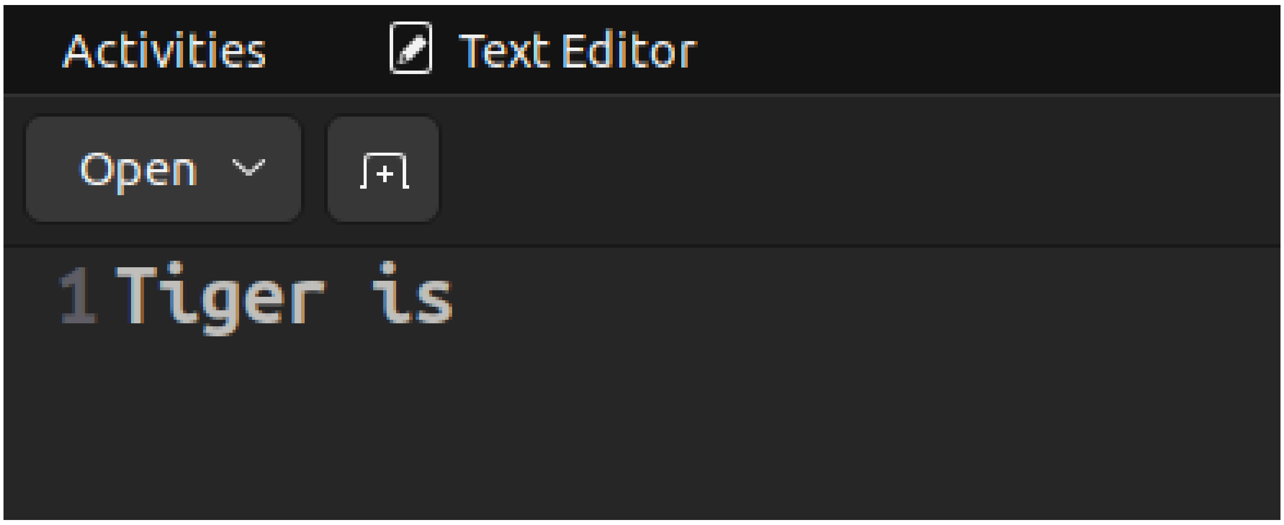
```
ari-18308@ari-18308:~/Desktop/Git$ git branch new-one
ari-18308@ari-18308:~/Desktop/Git$ git checkout new-one
Switched to branch 'new-one'
ari-18308@ari-18308:~/Desktop/Git$ git add .
ari-18308@ari-18308:~/Desktop/Git$ git status
On branch new-one
nothing to commit, working tree clean
ari-18308@ari-18308:~/Desktop/Git$ git status
On branch new-one
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:   myfile.txt

no changes added to commit (use "git add" and/or "git commit -a")
ari-18308@ari-18308:~/Desktop/Git$ git stash
Saved working directory and index state WIP on new-one: 9177ac3 COMMITTED
```

Let’s check out to master branch and come back to the new-one branch.



```
ari-18308@ari-18308:~/Desktop/Git$ git checkout master
Switched to branch 'master'
ari-18308@ari-18308:~/Desktop/Git$ git checkout new-one
Switched to branch 'new-one'
```

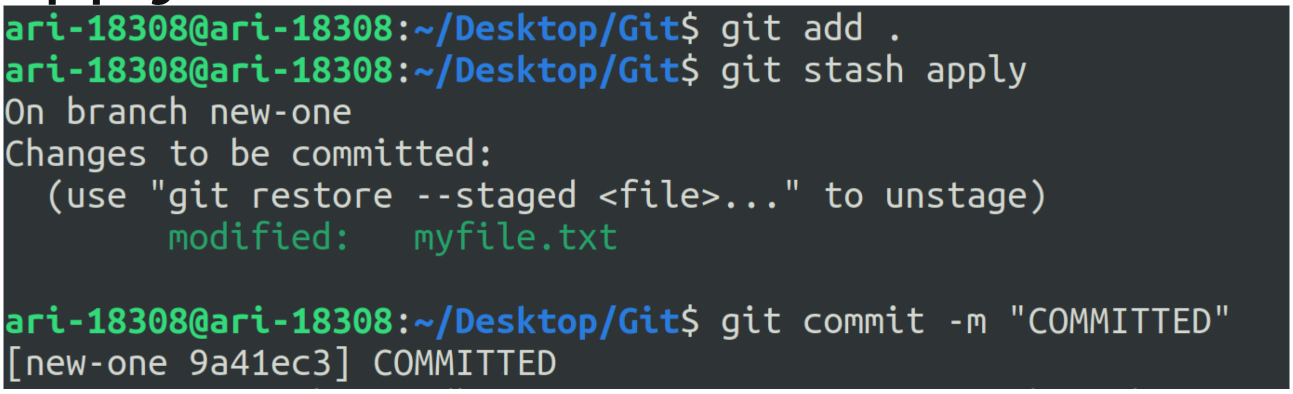
What will be there in the text file?



```
Tiger is
```

But, we did something called stash.

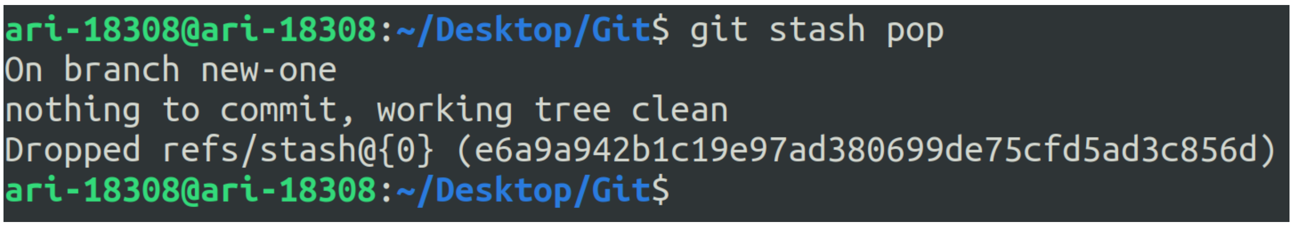
So, where are my contents? Stashing is just saving the working directory. We need to apply the save state back to our new-one branch. The changes can be reapplied using the git stash apply command.



```
ari-18308@ari-18308:~/Desktop/Git$ git add .
ari-18308@ari-18308:~/Desktop/Git$ git stash apply
On branch new-one
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        modified:   myfile.txt

ari-18308@ari-18308:~/Desktop/Git$ git commit -m "COMMITTED"
[new-one 9a41ec3] COMMITTED
```

The git stash command is useful in scenarios where we want to switch to another branch to work on something else, but don't want to commit the changes we've made to our current branch. Additionally, git stash can also be used to store multiple sets of changes, which can be listed using the git stash list command and reapplied using git stash apply. If we have multiple stashes, we can specify a stash by name or reference, like stash@{2}.

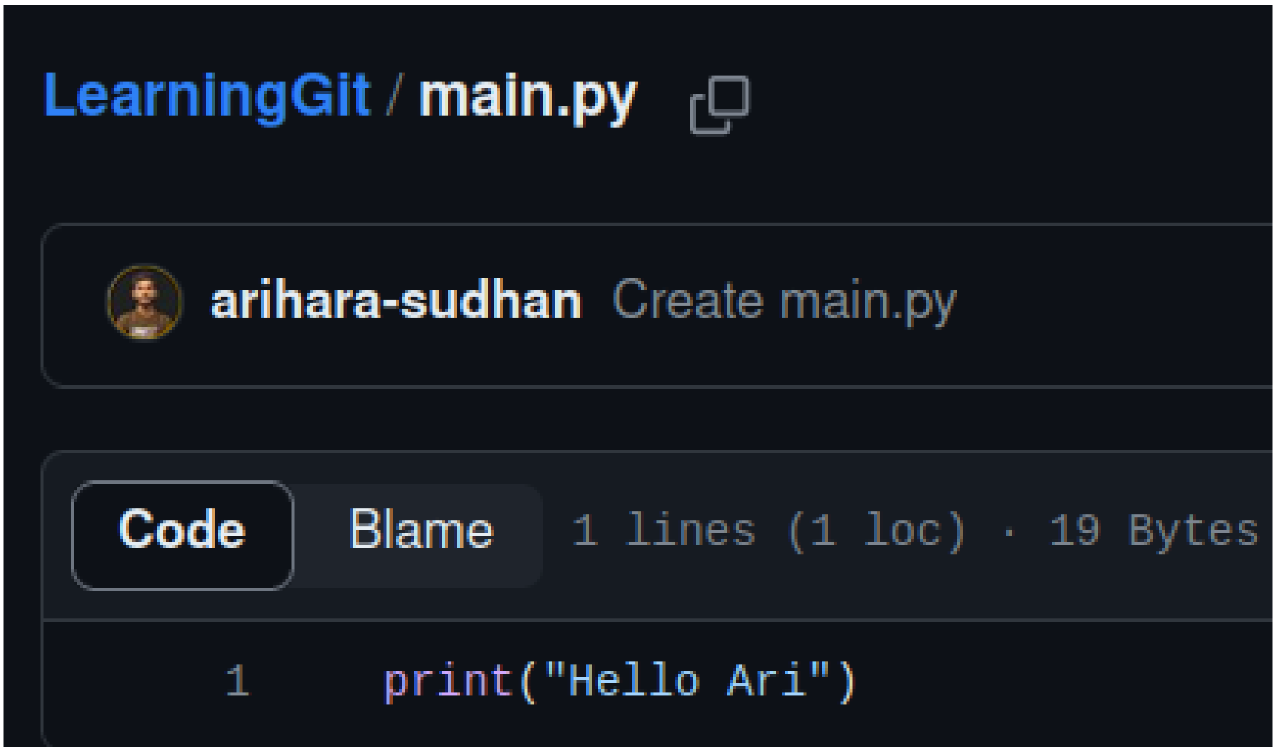
git stash pop removes the most recent stash from the stash list.



```
ari-18308@ari-18308:~/Desktop/Git$ git stash pop
On branch new-one
nothing to commit, working tree clean
Dropped refs/stash@{0} (e6a9a942b1c19e97ad380699de75cfd5ad3c856d)
ari-18308@ari-18308:~/Desktop/Git$
```

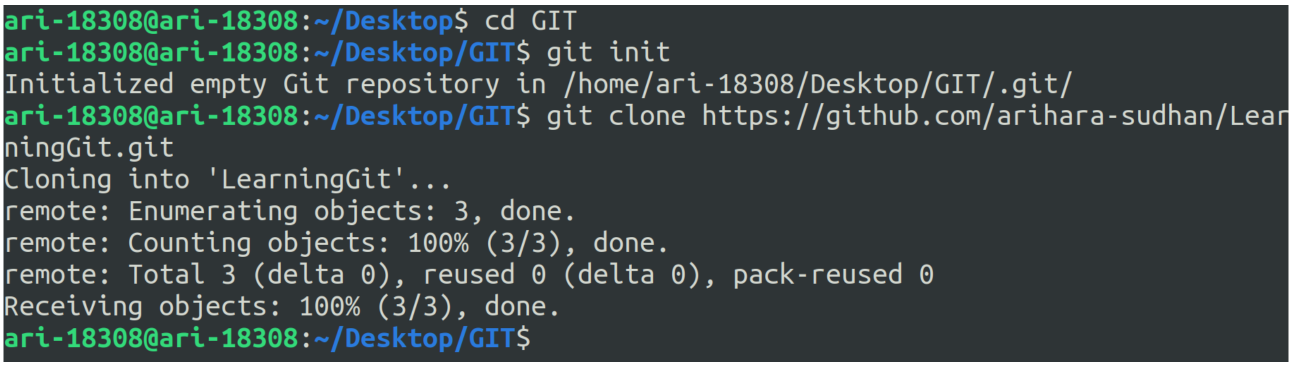
Push & Pull

Let’s clone one repository from my github. The repository has main.py as shown below.



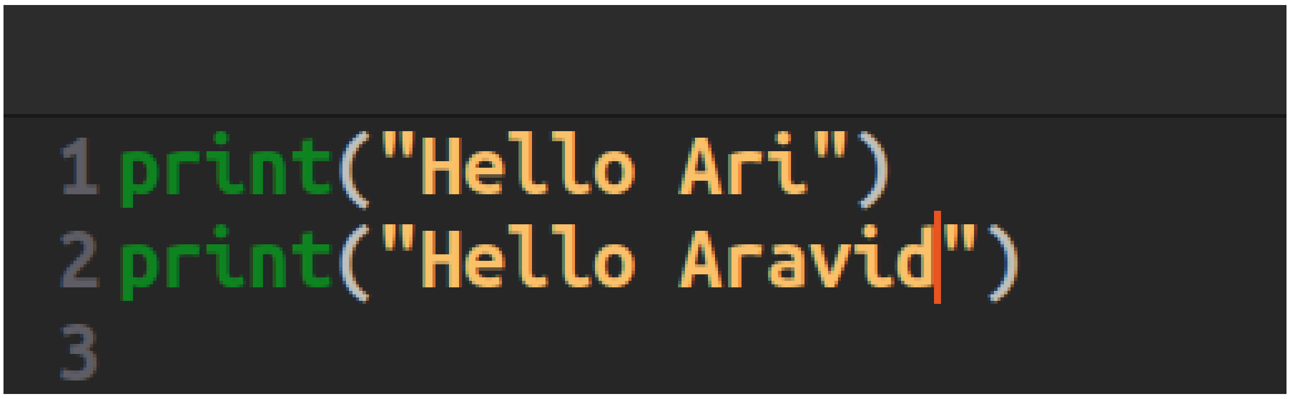
```
print("Hello Ari")
```

Let’s clone it.

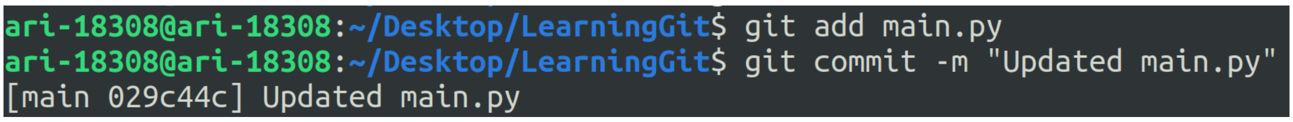


```
ari-18308@ari-18308:~/Desktop$ cd GIT
ari-18308@ari-18308:~/Desktop/GIT$ git init
Initialized empty Git repository in /home/ari-18308/Desktop/GIT/.git/
ari-18308@ari-18308:~/Desktop/GIT$ git clone https://github.com/arihara-sudhan/Lear
ningGit.git
Cloning into 'LearningGit'...
remote: Enumerating objects: 3, done.
remote: Counting objects: 100% (3/3), done.
remote: Total 3 (delta 0), reused 0 (delta 0), pack-reused 0
Receiving objects: 100% (3/3), done.
ari-18308@ari-18308:~/Desktop/GIT$
```

Once we have cloned it, let's change the main.py in the local repo. I would like to add another print statement in the code.

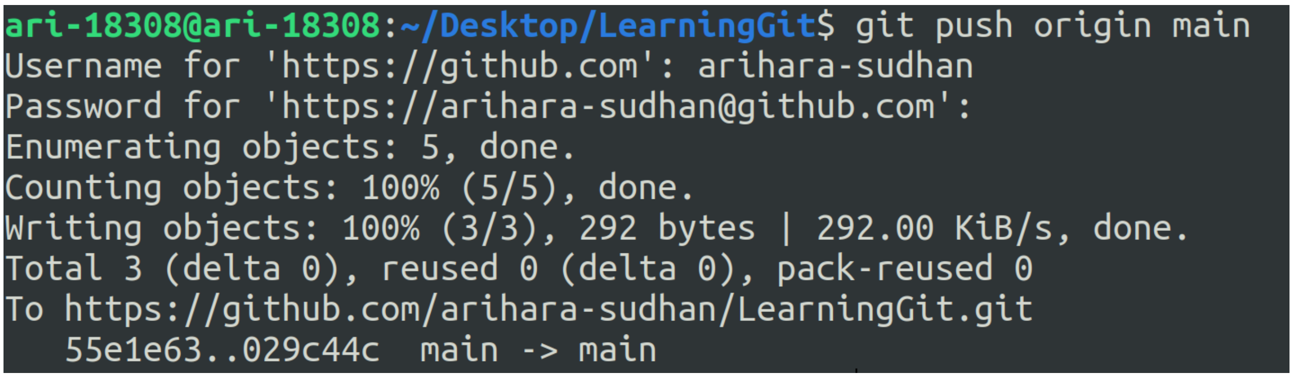


```
print("Hello Ari")
print("Hello Aravid")
```



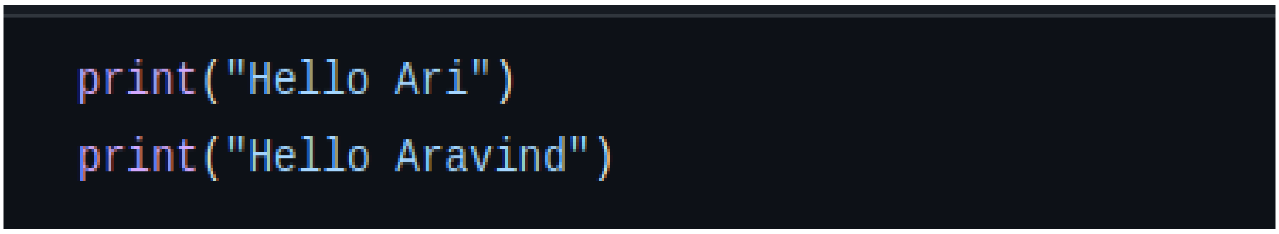
```
ari-18308@ari-18308:~/Desktop/LearningGit$ git add main.py
ari-18308@ari-18308:~/Desktop/LearningGit$ git commit -m "Updated main.py"
[main 029c44c] Updated main.py
```

How to apply this change to main.py in the remote repository? The answer is through push command. We can use git push origin branch-name.



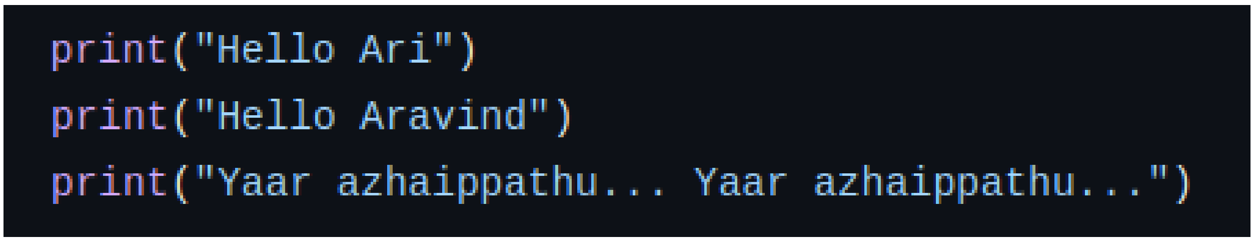
```
ari-18308@ari-18308:~/Desktop/LearningGit$ git push origin main
Username for 'https://github.com': arihara-sudhan
Password for 'https://arihara-sudhan@github.com':
Enumerating objects: 5, done.
Counting objects: 100% (5/5), done.
Writing objects: 100% (3/3), 292 bytes | 292.00 KiB/s, done.
Total 3 (delta 0), reused 0 (delta 0), pack-reused 0
To https://github.com/arihara-sudhan/LearningGit.git
   55e1e63..029c44c  main -> main
```

In remote :



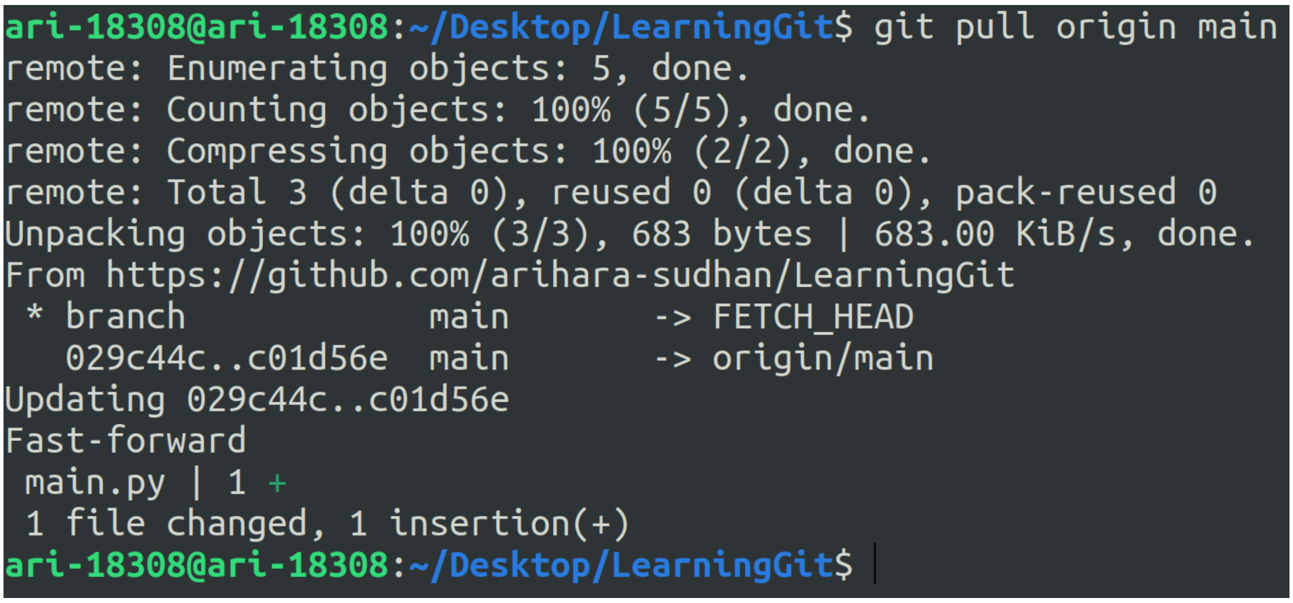
```
print("Hello Ari")
print("Hello Aravind")
```

In Git, the term "origin" is commonly used as a default remote repository name. When you clone a repository, Git automatically creates a remote called "origin" that points to the repository you cloned from. It's a convention, but you can technically rename the remote if you want. Now, I would like to add another print statement but in remote.



```
print("Hello Ari")
print("Hello Aravind")
print("Yaar azhaippathu... Yaar azhaippathu...")
```

We need this changes to be reflected in local repository. What can we do for that? We can use pull for that. (eg : git pull origin main)

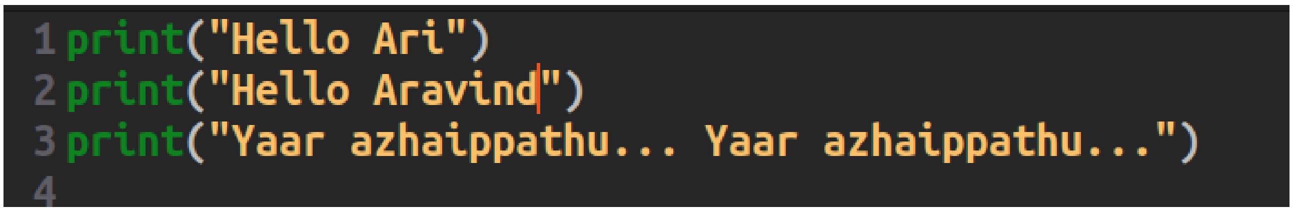


```
ari-18308@ari-18308:~/Desktop/LearningGit$ git pull origin main
remote: Enumerating objects: 5, done.
remote: Counting objects: 100% (5/5), done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 3 (delta 0), reused 0 (delta 0), pack-reused 0
Unpacking objects: 100% (3/3), 683 bytes | 683.00 KiB/s, done.
From https://github.com/arihara-sudhan/LearningGit
 * branch            main       -> FETCH_HEAD
   029c44c..c01d56e  main       -> origin/main
Updating 029c44c..c01d56e
Fast-forward
 main.py | 1 +
 1 file changed, 1 insertion(+)
ari-18308@ari-18308:~/Desktop/LearningGit$
```

In local :

We can also use the fetch. But, after fetching the remote contents, we have to manually merge them to a branch. It is advantageous while we want to merge to a specific branch instead of to the active branch.

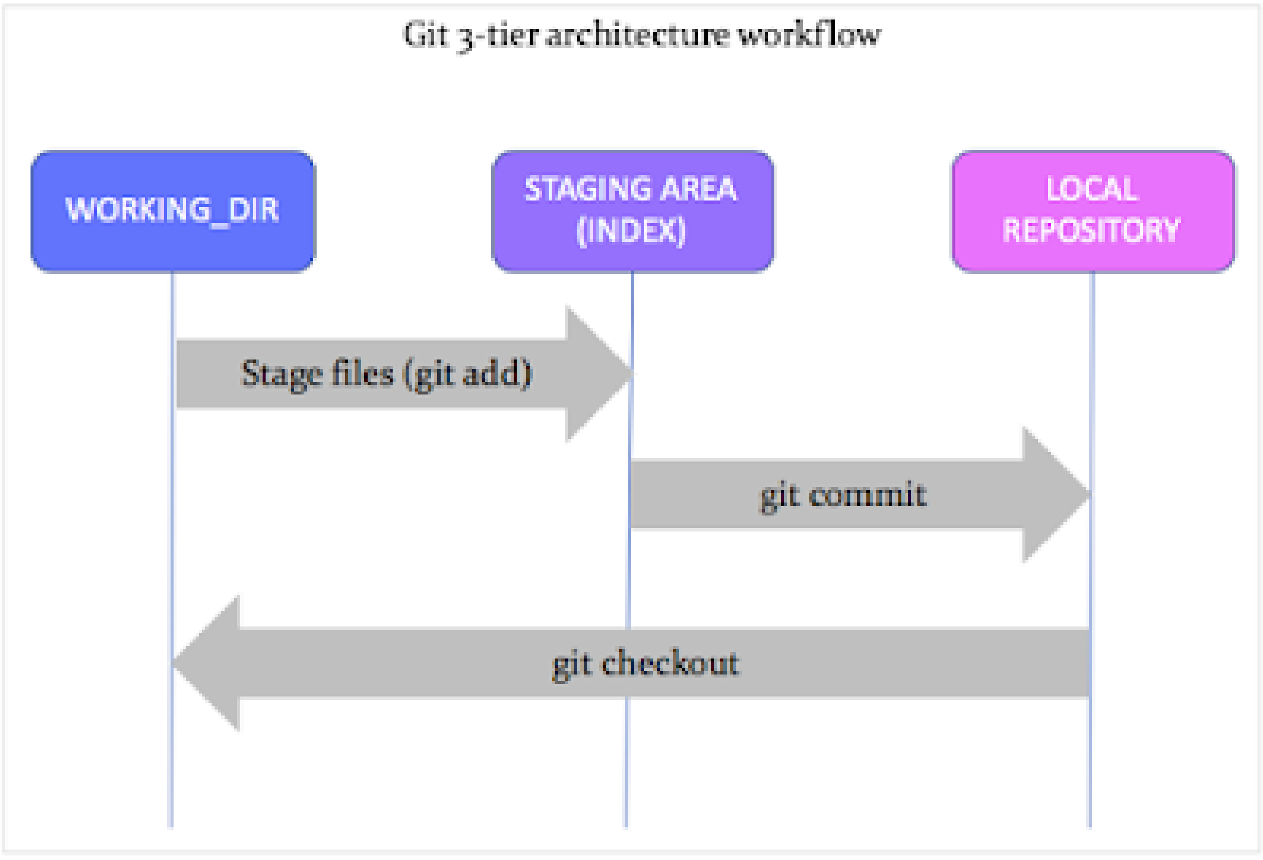
# Fetch changes from the remote git
fetch origin main



```
print("Hello Ari")
print("Hello Aravind")
print("Yaar azhaippathu... Yaar azhaippathu...")
```

# Merge the changes into local git
merge origin/main

The Three Tier Architecture

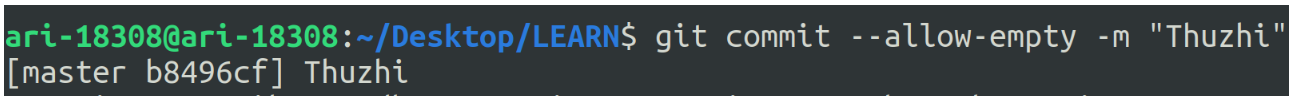


The Working Directory is where we have all our files and folders, and we're actively making changes to our code. When we modify or create new files, we're doing it in our working directory. The staging area is like a pre-commit zone. Before we actually save our changes in the project's history (repository), we can selectively choose which changes to include in the next commit. Imagine it as a place where we decide, "I want to include these specific changes in my next save point." It allows us to organize and review our modifications before they become a permanent part of our project history The repository is like a database that keeps track of all the changes to our project over time. It's where our project's history is stored. When we make a commit, we're taking a snapshot of our code at that point in time and saving it to the repository.

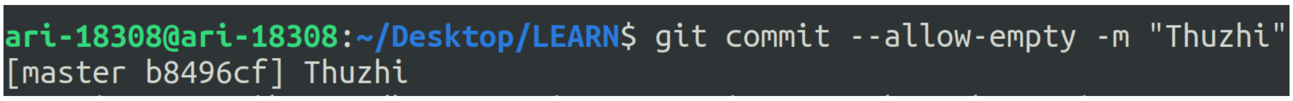
Rebase

Our project is a series of commits in Git. These snapshots capture the state of our project at different points in time. We might be working on a particular branch. Rebase is like rearranging these snapshots to create a cleaner, more straightforward history. Instead of adding our changes on top of the existing snapshots (as with merging), rebase moves or combines our changes with the latest state of the project. We start working on a feature branch, but in the meantime, others made changes to the main branch. If we perform rebase, it would result in a cleaner, linear history without unnecessary merge commits.

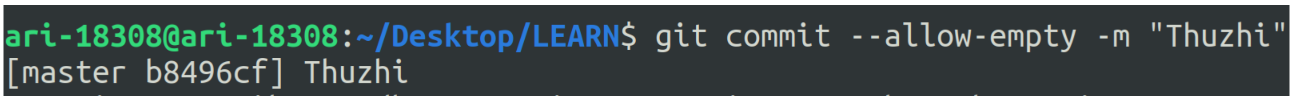
I do three empty commits in my master branch.



```
ari-18308@ari-18308:~/Desktop/LEARN$ git commit --allow-empty -m "Thuzhi"
[master b8496cf] Thuzhi
```

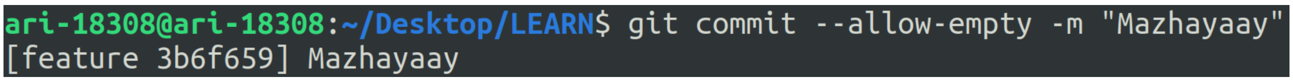


```
ari-18308@ari-18308:~/Desktop/LEARN$ git commit --allow-empty -m "Thuzhi"
[master b8496cf] Thuzhi
```

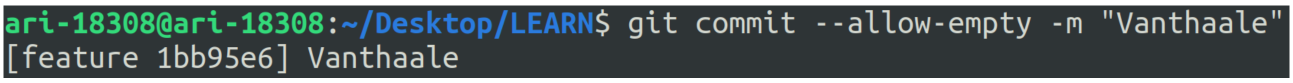


```
ari-18308@ari-18308:~/Desktop/LEARN$ git commit --allow-empty -m "Thuzhi"
[master b8496cf] Thuzhi
```

Now, I do two empty commits in feature branch.

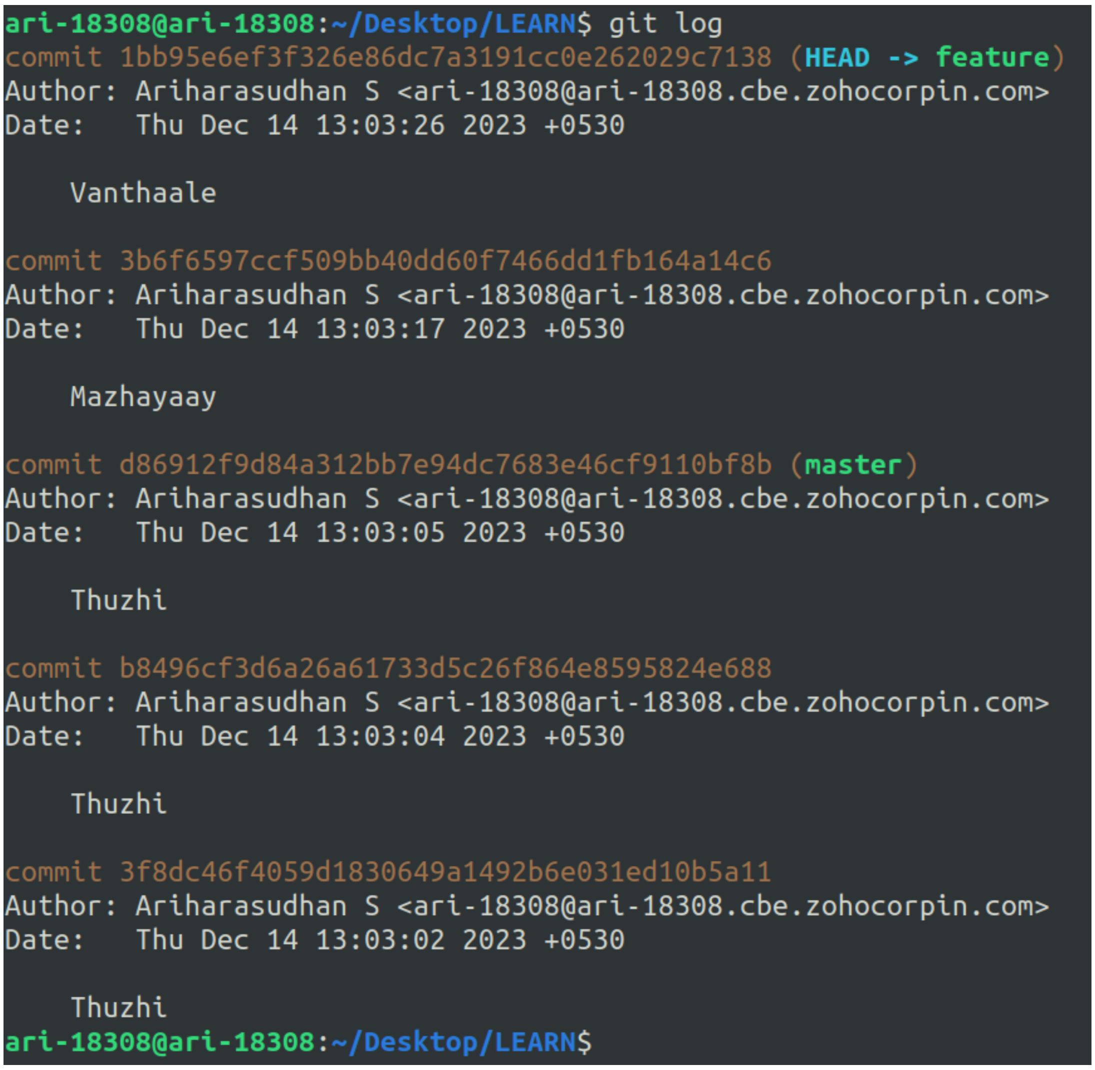


```
ari-18308@ari-18308:~/Desktop/LEARN$ git commit --allow-empty -m "Mazhayaay"
[feature 3b6f659] Mazhayaay
```



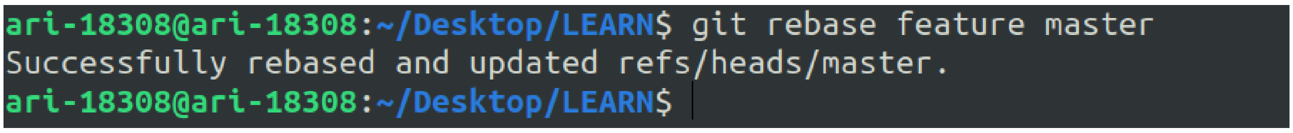
```
ari-18308@ari-18308:~/Desktop/LEARN$ git commit --allow-empty -m "Vanthaale"
[feature 1bb95e6] Vanthaale
```

We can view the log using git log.



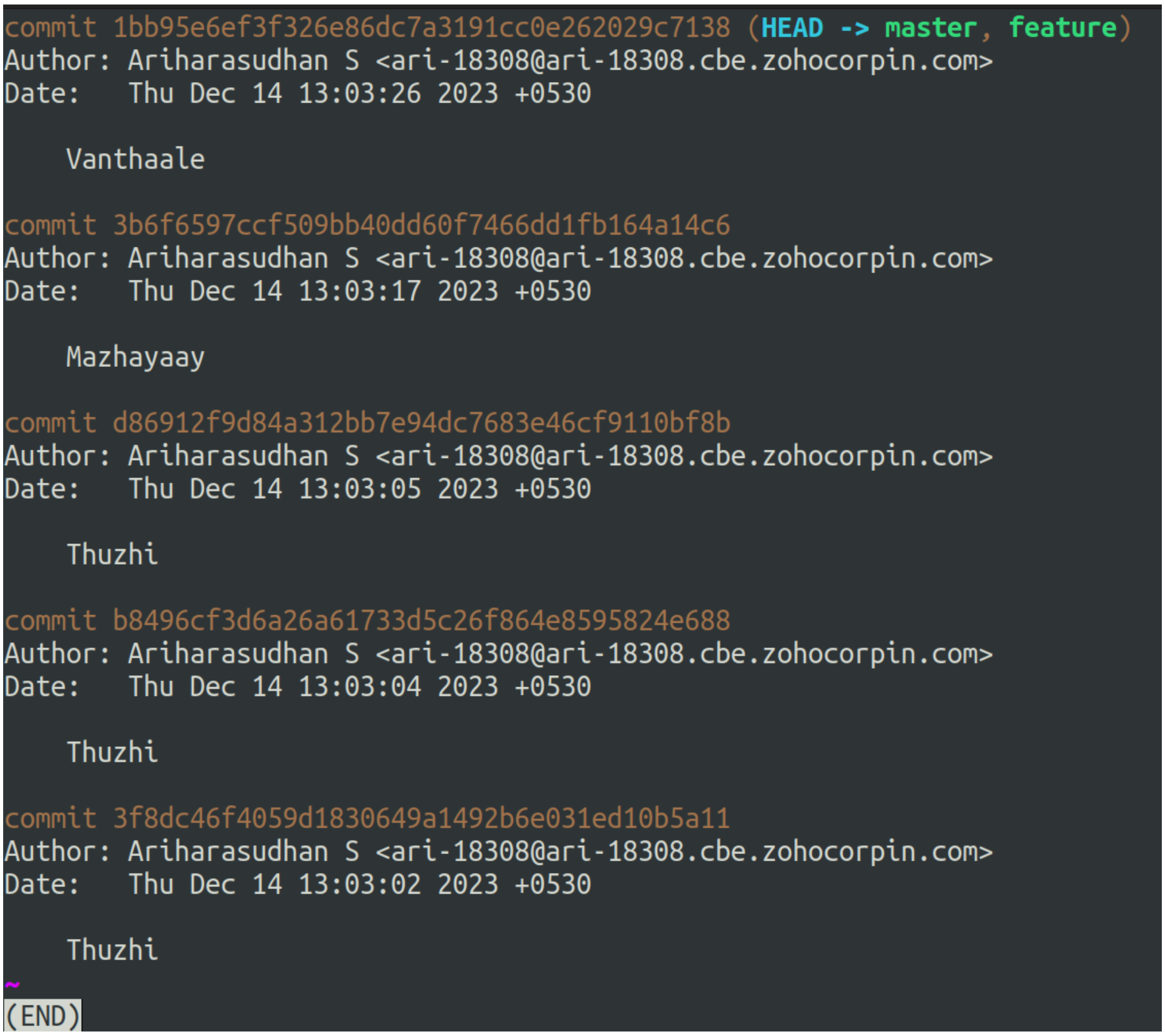
```
ari-18308@ari-18308:~/Desktop/LEARN$ git log
commit 1bb95e6ef3f326e86dc7a3191cc0e262029c7138 (HEAD -> feature)
Author: Ariharasudhan S <ari-18308@ari-18308.cbe.zohocorpin.com>
Date:   Thu Dec 14 13:03:26 2023 +0530

    Vanthaale

commit 3b6f6597ccf509bb40dd60f7466dd1fb164a14c6
Author: Ariharasudhan S <ari-18308@ari-18308.cbe.zohocorpin.com>
Date:   Thu Dec 14 13:03:17 2023 +0530

    Mazhayaay

commit d86912f9d84a312bb7e94dc7683e46cf9110bf8b (master)
Author: Ariharasudhan S <ari-18308@ari-18308.cbe.zohocorpin.com>
Date:   Thu Dec 14 13:03:05 2023 +0530

    Thuzhi

commit b8496cf3d6a26a61733d5c26f864e8595824e688
Author: Ariharasudhan S <ari-18308@ari-18308.cbe.zohocorpin.com>
Date:   Thu Dec 14 13:03:04 2023 +0530

    Thuzhi

commit 3f8dc46f4059d1830649a1492b6e031ed10b5a11
Author: Ariharasudhan S <ari-18308@ari-18308.cbe.zohocorpin.com>
Date:   Thu Dec 14 13:03:02 2023 +0530

    Thuzhi
ari-18308@ari-18308:~/Desktop/LEARN$
```

I would like to rebase the feature branch with the master branch.



```
ari-18308@ari-18308:~/Desktop/LEARN$ git rebase feature master
Successfully rebased and updated refs/heads/master.
ari-18308@ari-18308:~/Desktop/LEARN$
```

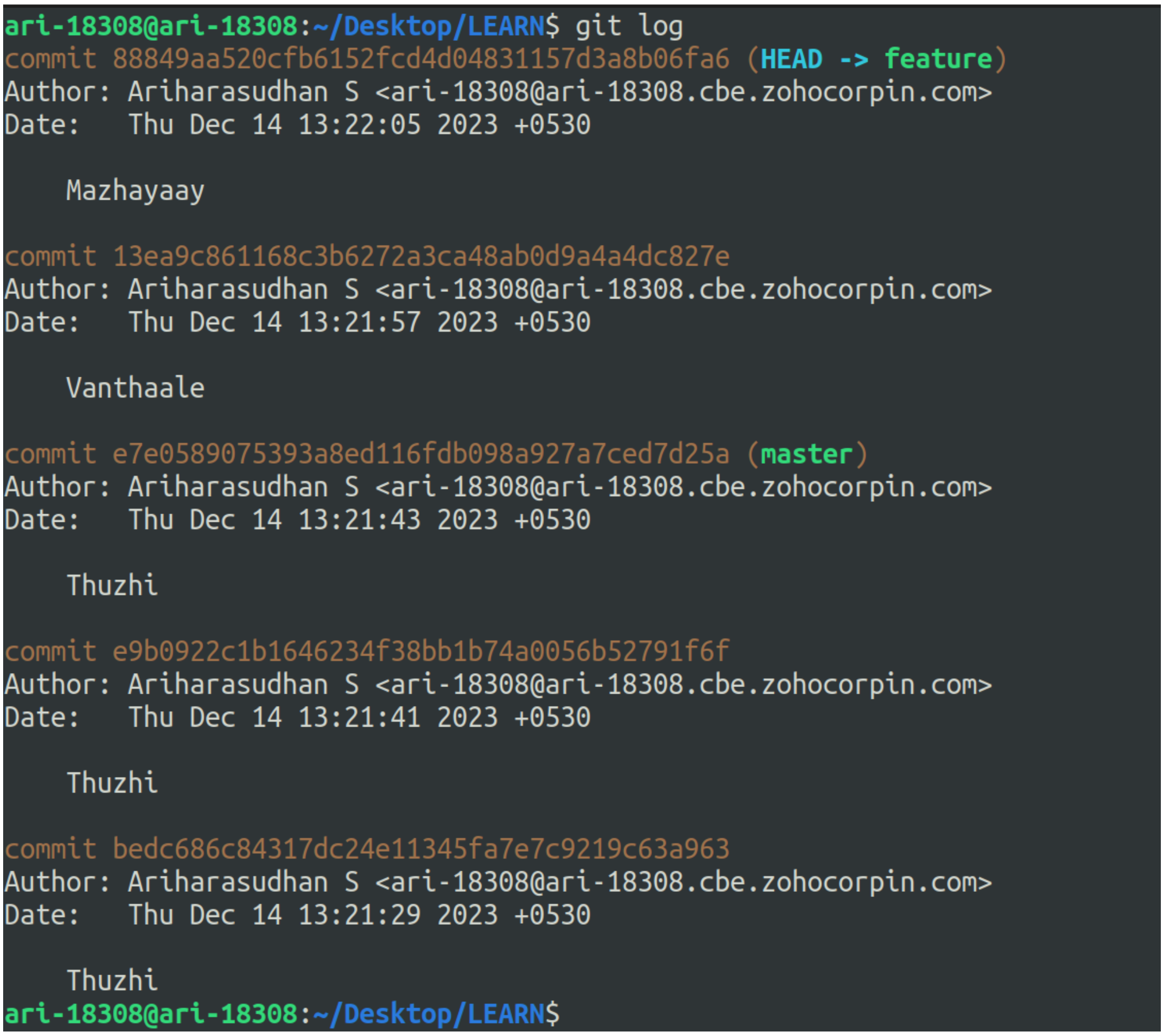
Now, if you log,



```
commit 1bb95e6ef3f326e86dc7a3191cc0e262029c7138 (HEAD -> master, feature)
Author: Ariharasudhan S <ari-18308@ari-18308.cbe.zohocorpin.com>
Date:   Thu Dec 14 13:03:26 2023 +0530

    Vanthaale

commit 3b6f6597ccf509bb40dd60f7466dd1fb164a14c6
Author: Ariharasudhan S <ari-18308@ari-18308.cbe.zohocorpin.com>
Date:   Thu Dec 14 13:03:17 2023 +0530

    Mazhayaay

commit d86912f9d84a312bb7e94dc7683e46cf9110bf8b
Author: Ariharasudhan S <ari-18308@ari-18308.cbe.zohocorpin.com>
Date:   Thu Dec 14 13:03:05 2023 +0530

    Thuzhi

commit b8496cf3d6a26a61733d5c26f864e8595824e688
Author: Ariharasudhan S <ari-18308@ari-18308.cbe.zohocorpin.com>
Date:   Thu Dec 14 13:03:04 2023 +0530

    Thuzhi

commit 3f8dc46f4059d1830649a1492b6e031ed10b5a11
Author: Ariharasudhan S <ari-18308@ari-18308.cbe.zohocorpin.com>
Date:   Thu Dec 14 13:03:02 2023 +0530

    Thuzhi
~
(END)
```

Cherry Pick

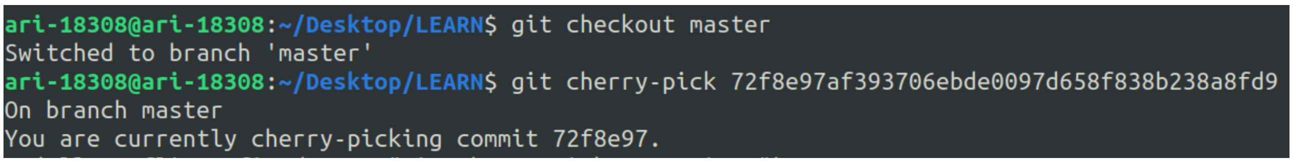
We can specifically select (pick) a commit and merge to a branch. We do the same empty commits to understand how cherry pick works.



```
ari-18308@ari-18308:~/Desktop/LEARN$ git log
commit 88849aa520cfb6152fcd4d04831157d3a8b06fa6 (HEAD -> feature)
Author: Ariharasudhan S <ari-18308@ari-18308.cbe.zohocorpin.com>
Date:   Thu Dec 14 13:22:05 2023 +0530

    Mazhayaay

commit 13ea9c861168c3b6272a3ca48ab0d9a4a4dc827e
Author: Ariharasudhan S <ari-18308@ari-18308.cbe.zohocorpin.com>
Date:   Thu Dec 14 13:21:57 2023 +0530

    Vanthaale

commit e7e0589075393a8ed116fdb098a927a7ced7d25a (master)
Author: Ariharasudhan S <ari-18308@ari-18308.cbe.zohocorpin.com>
Date:   Thu Dec 14 13:21:43 2023 +0530

    Thuzhi

commit e9b0922c1b1646234f38bb1b74a0056b52791f6f
Author: Ariharasudhan S <ari-18308@ari-18308.cbe.zohocorpin.com>
Date:   Thu Dec 14 13:21:41 2023 +0530

    Thuzhi

commit bedc686c84317dc24e11345fa7e7c9219c63a963
Author: Ariharasudhan S <ari-18308@ari-18308.cbe.zohocorpin.com>
Date:   Thu Dec 14 13:21:29 2023 +0530

    Thuzhi
ari-18308@ari-18308:~/Desktop/LEARN$
```

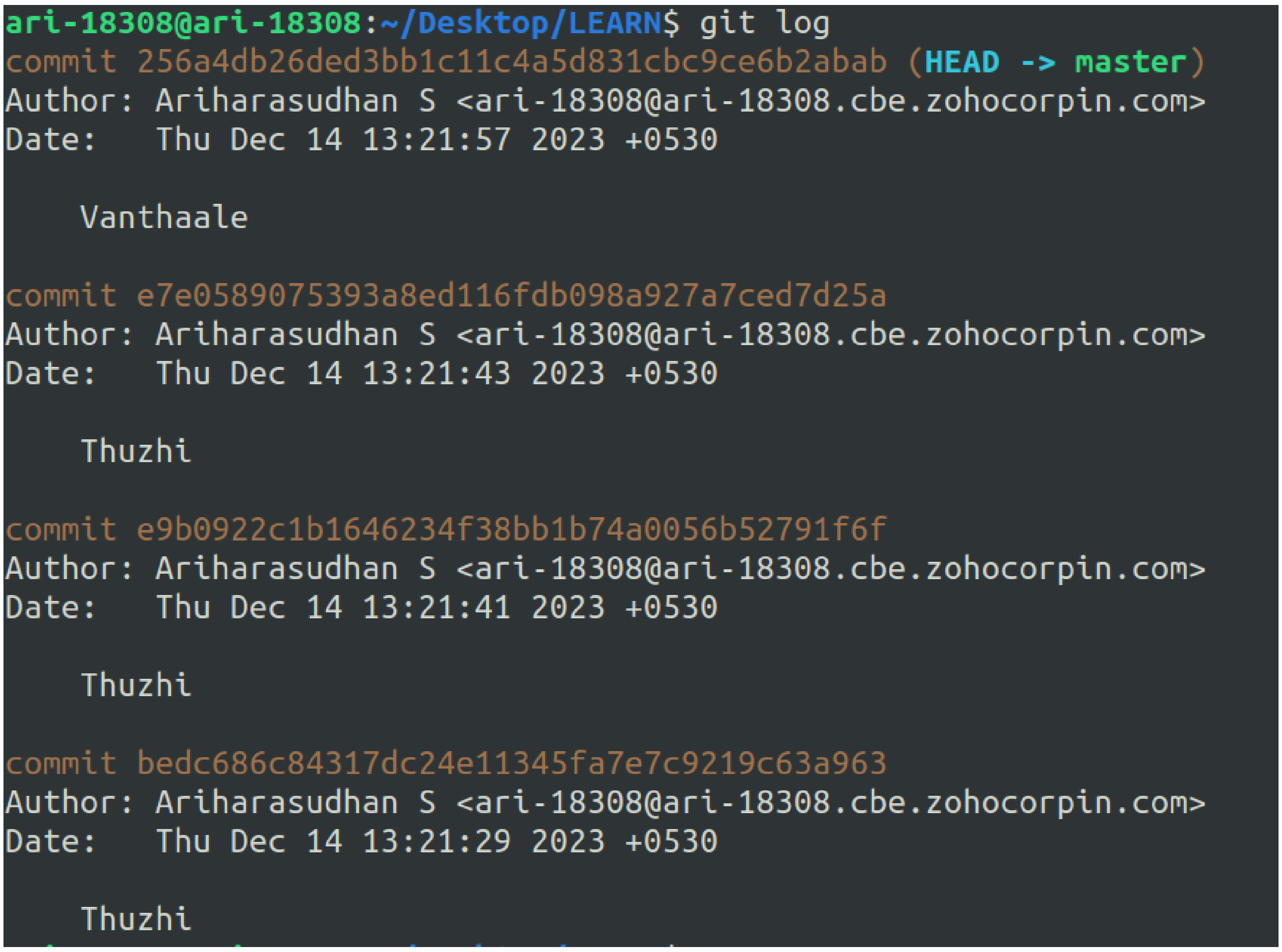
I would like to cherry pick the commit with the message “Vanthaale” to master branch. The target branch is where you are currently.

To do that, let’s checkout to master first. Then, use git cherrypick as shown below.



```
ari-18308@ari-18308:~/Desktop/LEARN$ git checkout master
Switched to branch 'master'
ari-18308@ari-18308:~/Desktop/LEARN$ git cherry-pick 72f8e97af393706ebde0097d658f838b238a8fd9
On branch master
You are currently cherry-picking commit 72f8e97.
```

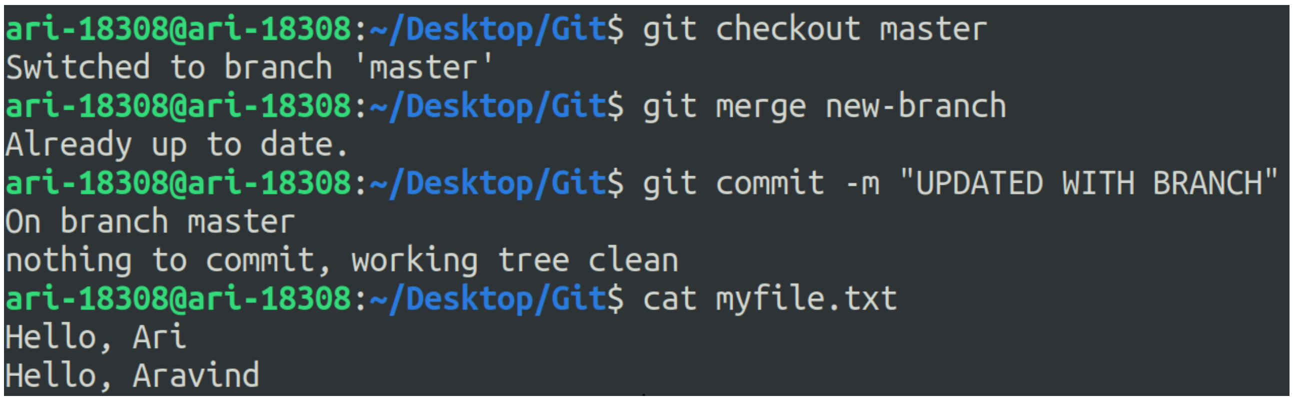
The command git cherrypick is followed by the hash of a specific commit.



```
ari-18308@ari-18308:~/Desktop/LEARN$ git log
commit 256a4db26ded3bb1c11c4a5d831cbc9ce6b2abab (HEAD -> master)
Author: Ariharasudhan S <ari-18308@ari-18308.cbe.zohocorpin.com>
Date:   Thu Dec 14 13:21:57 2023 +0530

    Vanthaale

commit e7e0589075393a8ed116fdb098a927a7ced7d25a
Author: Ariharasudhan S <ari-18308@ari-18308.cbe.zohocorpin.com>
Date:   Thu Dec 14 13:21:43 2023 +0530

    Thuzhi

commit e9b0922c1b1646234f38bb1b74a0056b52791f6f
Author: Ariharasudhan S <ari-18308@ari-18308.cbe.zohocorpin.com>
Date:   Thu Dec 14 13:21:41 2023 +0530

    Thuzhi

commit bedc686c84317dc24e11345fa7e7c9219c63a963
Author: Ariharasudhan S <ari-18308@ari-18308.cbe.zohocorpin.com>
Date:   Thu Dec 14 13:21:29 2023 +0530

    Thuzhi
```

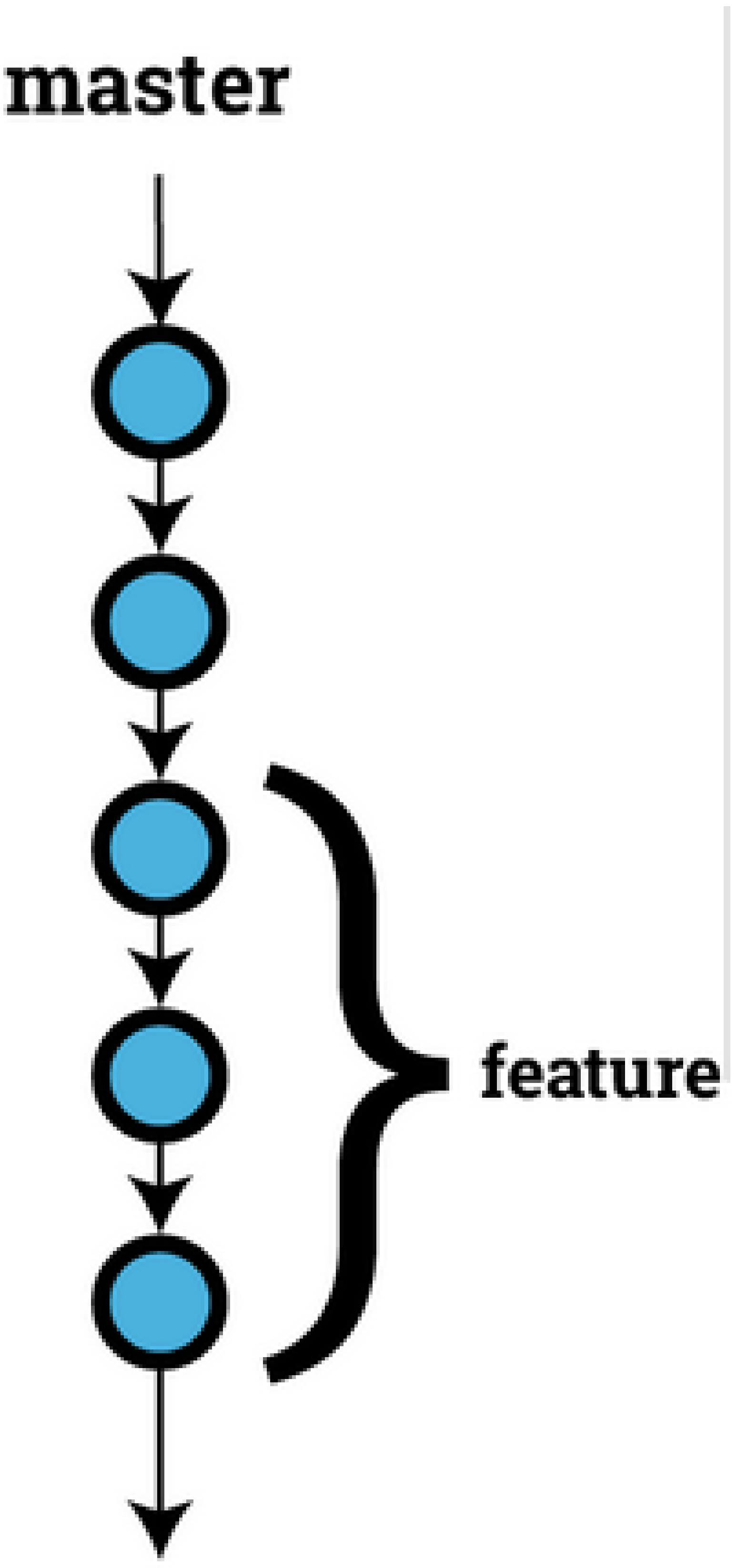
FF Merge

We have already performed merging operations in our previous examples. It is a process of incorporating the changes in one branch to the currently checked-out branch. Now, let's learn some interesting stuffs related to merge.



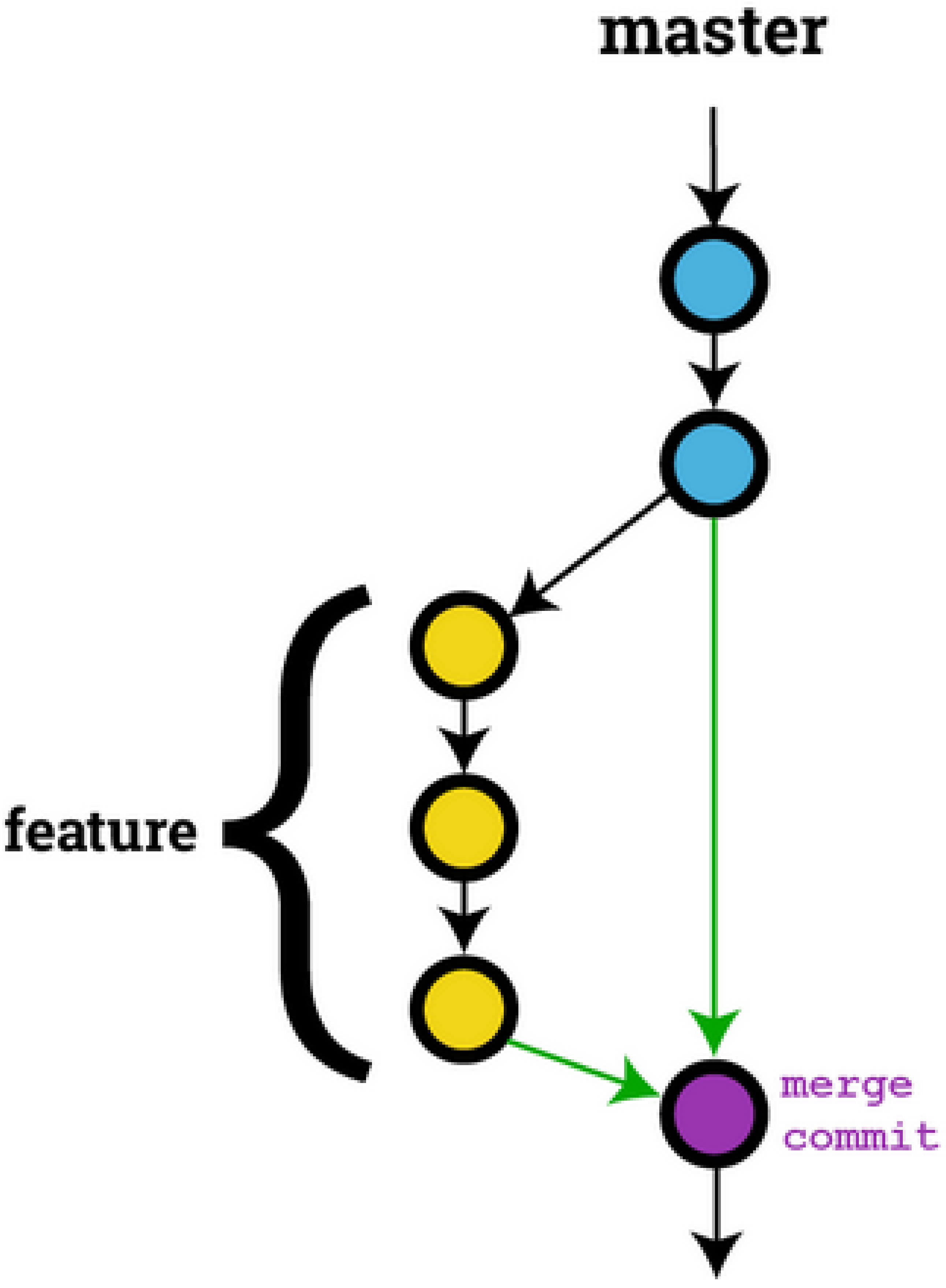
```
ari-18308@ari-18308:~/Desktop/Git$ git checkout master
Switched to branch 'master'
ari-18308@ari-18308:~/Desktop/Git$ git merge new-branch
Already up to date.
ari-18308@ari-18308:~/Desktop/Git$ git commit -m "UPDATED WITH BRANCH"
On branch master
nothing to commit, working tree clean
ari-18308@ari-18308:~/Desktop/Git$ cat myfile.txt
Hello, Ari
Hello, Aravind
```

There are some interesting types of merges we can learn. If we create a new branch, make changes on that branch, and during that time, no one else makes changes to the main branch, then when we merge our branch back into the main branch, it will be a fast-forward merge. Fast-forward merges result in a linear history, where each commit follows the previous one in a straightforward sequence. This makes it easier to understand the chronological order of changes.



`git merge`

If we don’t do fast forward merge, the history won’t be linear.



If it is not a fast forward merge,
# Switch to the main branch

```
> git checkout main
```

# Merge the feature branch

```
git merge --no-ff feature-branch
```

If we're confident that a fast-forward merge is possible, the process is simpler :

# Switch to the main branch

```
git checkout main
```

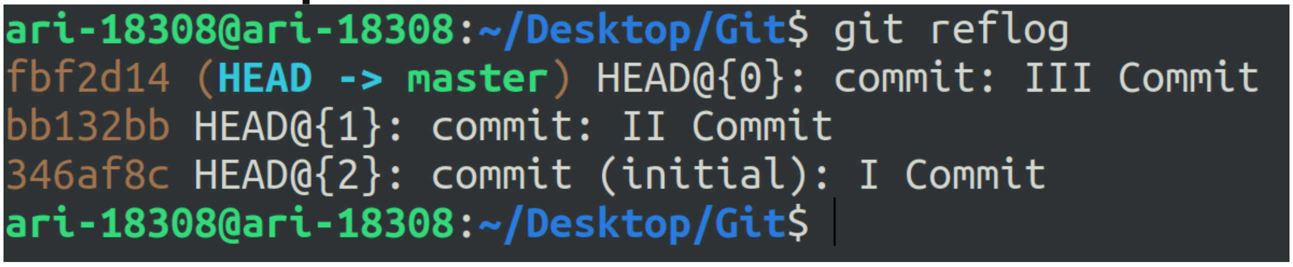
# Merge the feature branch

```
git merge feature-branch
```

Revert

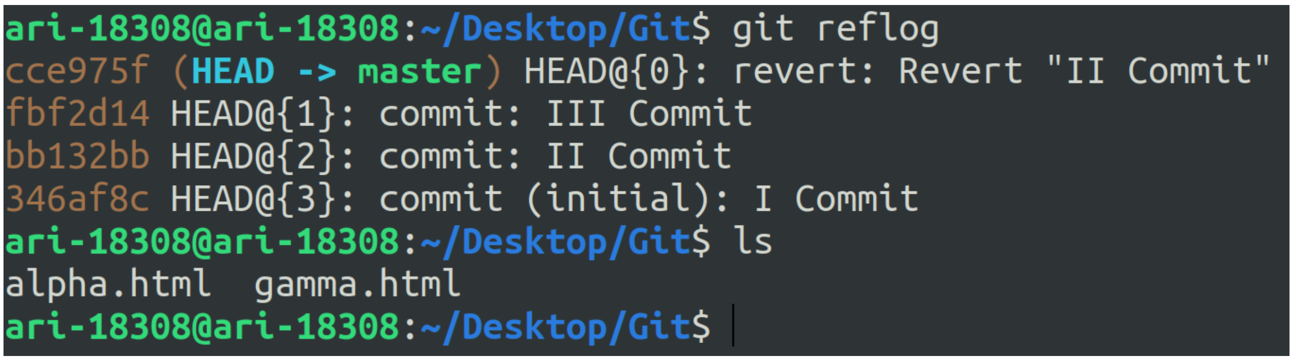
Reverting is used to create a new commit that undoes the changes made in a specific previous commit. This is useful when we want to reverse the effects of a particular commit without removing it from the commit history. Let’s understand this with the example below.

Let’s use reflog to view the log. reflog is a private, workspace-specific record of the repo's local commits.



```
ari-18308@ari-18308:~/Desktop/Git$ git reflog
fbf2d14 (HEAD -> master) HEAD@{0}: commit: III Commit
bb132bb HEAD@{1}: commit: II Commit
346af8c HEAD@{2}: commit (initial): I Commit
ari-18308@ari-18308:~/Desktop/Git$
```

In the I commit, I created alpha.html. In the II commit, I created beta.html and in the III commit, I created gamma.html. Let’s revert to II Commit. (Will it remove the changes made by III commit?)

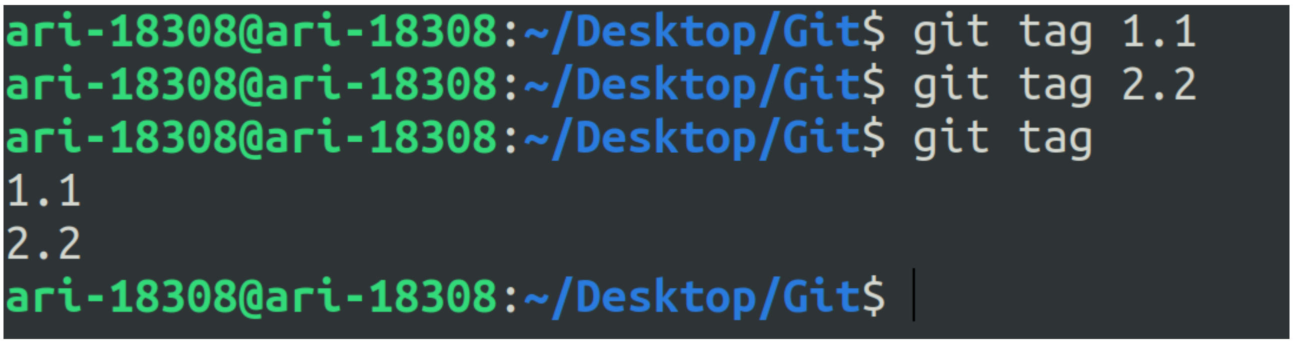


```
ari-18308@ari-18308:~/Desktop/Git$ git reflog
cce975f (HEAD -> master) HEAD@{0}: revert: Revert "II Commit"
fbf2d14 HEAD@{1}: commit: III Commit
bb132bb HEAD@{2}: commit: II Commit
346af8c HEAD@{3}: commit (initial): I Commit
ari-18308@ari-18308:~/Desktop/Git$ ls
alpha.html  gamma.html
ari-18308@ari-18308:~/Desktop/Git$
```

Yes! It did so! We don’t have beta.html.

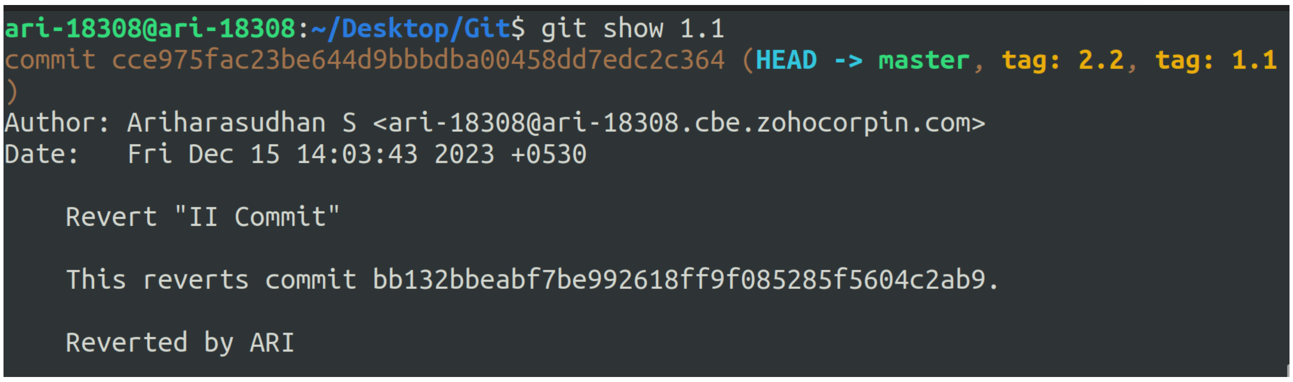
Tag

Git has the ability to tag specific points in a repository's history as being important. Typically, people use this functionality to mark release points (`v1.0`, `v2.0` and so on). We can create tags using git tag.



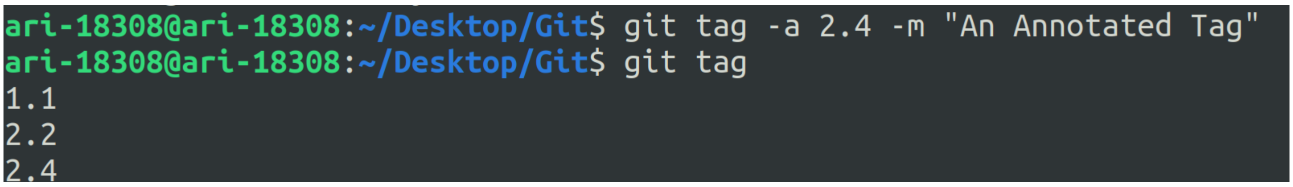
```
ari-18308@ari-18308:~/Desktop/Git$ git tag 1.1
ari-18308@ari-18308:~/Desktop/Git$ git tag 2.2
ari-18308@ari-18308:~/Desktop/Git$ git tag
1.1
2.2
ari-18308@ari-18308:~/Desktop/Git$ 
```

These are simple tags (light weighted).



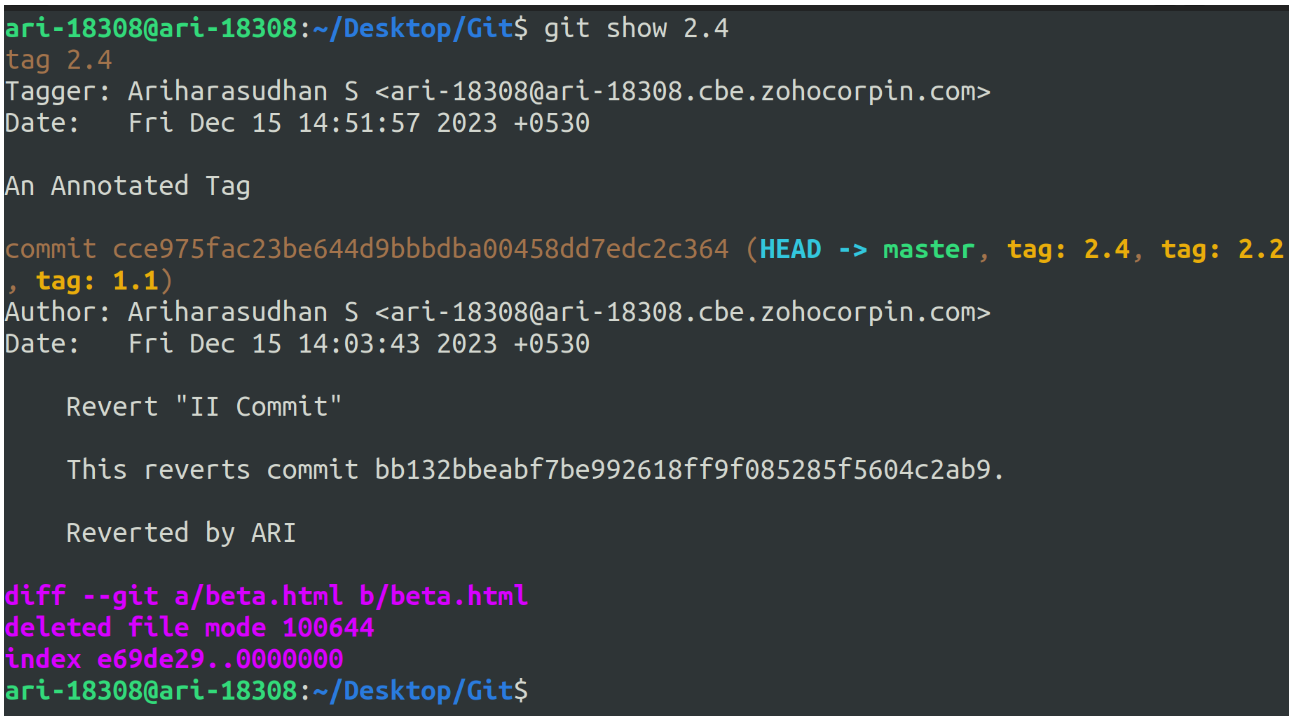
```
ari-18308@ari-18308:~/Desktop/Git$ git show 1.1
commit cce975fac23be644d9bbbdba00458dd7edc2c364 (HEAD -> master, tag: 2.2, tag: 1.1
)
Author: Ariharasudhan S <ari-18308@ari-18308.cbe.zohocorpin.com>
Date:   Fri Dec 15 14:03:43 2023 +0530

    Revert "II Commit"

    This reverts commit bb132bbeabf7be992618ff9f085285f5604c2ab9.

    Reverted by ARI
```

We can also create annotated tags.



```
ari-18308@ari-18308:~/Desktop/Git$ git tag -a 2.4 -m "An Annotated Tag"
ari-18308@ari-18308:~/Desktop/Git$ git tag
1.1
2.2
2.4
```

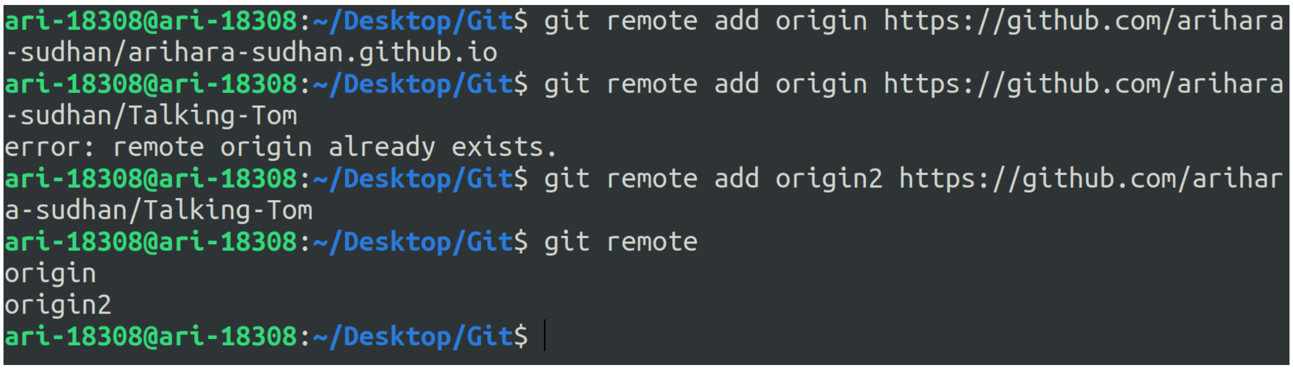
-a stands for annotated tag, which creates a full tag object in the Git database that contains a tagger name, email, date, and a tagging message.



```
ari-18308@ari-18308:~/Desktop/Git$ git show 2.4
tag 2.4
Tagger: Ariharasudhan S <ari-18308@ari-18308.cbe.zohocorpin.com>
Date:   Fri Dec 15 14:51:57 2023 +0530

An Annotated Tag

commit cce975fac23be644d9bbbdba00458dd7edc2c364 (HEAD -> master, tag: 2.4, tag: 2.2
, tag: 1.1)
Author: Ariharasudhan S <ari-18308@ari-18308.cbe.zohocorpin.com>
Date:   Fri Dec 15 14:03:43 2023 +0530

    Revert "II Commit"

    This reverts commit bb132bbeabf7be992618ff9f085285f5604c2ab9.

    Reverted by ARI

diff --git a/beta.html b/beta.html
deleted file mode 100644
index e69de29..0000000
ari-18308@ari-18308:~/Desktop/Git$
```

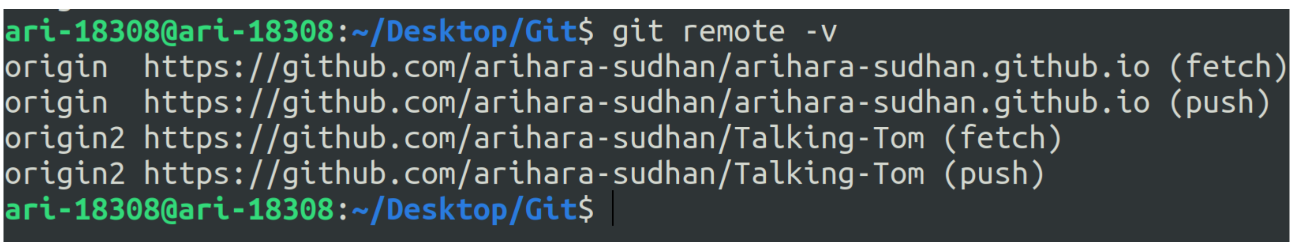
Remotes

A remote is a reference to a repository on a remote server. It allows to interact with repositories hosted on platforms like GitHub. Remotes are used to fetch and push changes to and from remote repositories. git remote add is the command is used to add a new remote. For example, to add a remote named "origin" pointing to a GitHub repository, we can do, git remote add origin https://github.com/username/repo.git This command establishes a connection between your local repository and the remote repository.



```
ari-18308@ari-18308:~/Desktop/Git$ git remote add origin https://github.com/arihara
-sudhan/arihara-sudhan.github.io
ari-18308@ari-18308:~/Desktop/Git$ git remote add origin https://github.com/arihara
-sudhan/Talking-Tom
error: remote origin already exists.
ari-18308@ari-18308:~/Desktop/Git$ git remote add origin2 https://github.com/arihar
a-sudhan/Talking-Tom
ari-18308@ari-18308:~/Desktop/Git$ git remote
origin
origin2
ari-18308@ari-18308:~/Desktop/Git$
```

git remote -v command lists all remotes associated with the repository, along with their URLs.



```
ari-18308@ari-18308:~/Desktop/Git$ git remote -v
origin  https://github.com/arihara-sudhan/arihara-sudhan.github.io (fetch)
origin  https://github.com/arihara-sudhan/arihara-sudhan.github.io (push)
origin2 https://github.com/arihara-sudhan/Talking-Tom (fetch)
origin2 https://github.com/arihara-sudhan/Talking-Tom (push)
ari-18308@ari-18308:~/Desktop/Git$
```

The -v flag stands for verbose and shows the URLs.

Local & Remote Branches

Local branches are branches that exist only on our local machine, while remote branches are references to the state of branches on our remote repositories. When we clone a repository, our local machine creates a copy of the remote branches. Local Branches exist only on local machine. It allows to work on features, bug fixes, or experiments locally. Local operations are fast and do not require interaction with a remote repository. Remote Branches exist on a remote repository (e.g., GitHub, GitLab). It facilitate collaboration by allowing multiple people to work on the same codebase.

If local and remote branches are not in sync, it means that changes have occurred in one branch and not in the other. This can happen when, someone else has pushed changes to the remote branch; we have pushed changes to the remote branch from another location; We have pulled changes from the remote branch. To synchronize local and remote branches, we can use fetch, pull and push as we have already discussed.

git branch -d branch_name is the command to delete a branch. If we delete a local branch, the branch will be deleted from your local machine. Deleting a local branch does not affect the remote branch. The command, git push origin --delete branch_name is used to delete remote branch. That branch will be deleted from the remote repository. Other team members need to fetch the changes to update their local repositories.

Local branches tracking the deleted remote branch will still exist, but they will be out of sync. We can delete the local tracking branch using `git branch -dr origin/branch_name` to remove the reference.

MR and PR

Merge Requests and Pull Requests are used to propose and discuss changes before they are merged into a target branch. A Merge Request is a request to merge changes from a source branch into a target branch. Developers create Merge Requests to propose changes, and these requests typically go through a review process before being merged. A Pull Request is a request to pull changes from a source branch into a target branch. Developers create Pull Requests to propose changes, and these requests also go through a review process before being merged.

Merging when Target Ahead

When we attempt to merge a source branch into a target branch, but the target branch is ahead (meaning it has additional commits that the source branch doesn't have), Git will perform a merge operation. If the source branch's changes can be applied directly on top of the target branch without conflicts and without creating a new commit (i.e., a linear history), Git will perform a "fast-forward" merge as we have already discussed earlier in this book. The target branch pointer simply moves forward to the latest commit of the source branch, and no new commit is created. If there are changes in both the source and target branches that cannot be applied cleanly in a linear fashion, Git will perform a "three-way merge.” This involves finding the common ancestor commit of the two branches and then merging the changes from both branches into a new commit. If there are conflicting changes in the source and target branches (i.e., changes in the same lines of the same file), Git will identify these conflicts and mark the files as conflicted. We will need to manually resolve these conflicts by editing the files, marking them as resolved, and then committing the changes.

HEAD State

HEAD refers to the currently checked-out branch's latest commit. The HEAD moves automatically whenever we create a new commit, indicating the latest state of your working directory. HEAD points to the latest commit in the currently active branch. It is a symbolic reference rather than a direct pointer to a commit. If we check out a specific commit (rather than a branch), Git enters a detached HEAD state. In this state, HEAD is pointing directly to a commit rather than a branch.

Commits made in this state won't be associated with any branch, and they might be lost if we switch branches.

```
# Checking out a specific commit
git checkout <commit_hash>
```

When we switch branches using git checkout or git switch, HEAD is updated to point to the latest commit of the newly checked-out branch. When we make a new commit, HEAD is moved to point to that new commit. The new commit becomes the latest commit on the active branch. We can reference previous commits relative to HEAD using commit notation. For example, `HEAD~1` refers to the commit before the latest commit, and `HEAD~2` refers to the commit two steps before the latest commit. It can be used like, `git show HEAD~1.` Commands like git reset and git revert can be used with HEAD to undo or modify the latest commit.

```
# Undo last commit but keep changes
> git reset HEAD~1
#Commit that undoes changes of last commit
> git revert HEAD
```