😊 Git

Git is a distributed version control system (DVCS) that is widely used for tracking changes in source code during software development. Developed by Linus Torvalds in 2005, Git has become the standard for version control in many industries and open-source projects. Here's a brief overview of Git:

1. **Distributed Version Control:** Git is a distributed version control system, which means that every developer working on a project has a complete copy of the repository, including its full history. This allows for offline work and provides redundancy and scalability.
2. **Tracking Changes:** Git tracks changes to files in a project by recording snapshots of the files' states over time. Each snapshot represents a commit, which includes metadata such as the author, timestamp, and a unique identifier (hash).
3. **Branching and Merging:** Git allows developers to create lightweight branches to work on new features or experiments independently of the main development line (master branch). Branches can be easily created, merged, and deleted, facilitating parallel development workflows.
4. **Collaboration:** Git supports collaborative development by providing mechanisms for sharing changes between developers. Developers can push their changes to a shared repository and pull changes made by others, enabling seamless collaboration on a shared codebase.
5. **Remote Repositories:** Git enables developers to work with remote repositories hosted on platforms like GitHub, GitLab, or Bitbucket. Remote repositories serve as central hubs for sharing code and collaborating with other developers.
6. **Workflow Flexibility:** Git supports various workflows, including centralized, feature branch, git flow, and forking workflows. Teams can choose the workflow that best suits their development process and project requirements.
7. **Open Source and Community:** Git is open source software distributed under the GNU General Public License (GPL), and it has a large and active community of developers contributing to its development and maintenance. This community-driven approach ensures ongoing support and innovation.
8. **Extensibility:** Git is highly extensible and customizable through its rich ecosystem of plugins, tools, and integrations. Developers can enhance Git's functionality by integrating it with other development tools, such as issue trackers, continuous integration systems, and code review platforms.

😄 Use cases of Git:

Git, as a distributed version control system, offers a wide range of use cases across various industries and software development scenarios. Here are some common use cases of Git:

1. **Collaborative Software Development:** Git enables multiple developers to work on the same codebase simultaneously. It facilitates collaboration by allowing developers to

clone repositories, create branches for feature development or bug fixes, and merge changes back into the main branch.

2. **Code Versioning and History:** Git tracks changes to source code files over time, providing a complete history of modifications. Developers can view the commit history, compare different versions, and revert to previous states if needed, ensuring code stability and accountability.
3. **Branching Strategies:** Git's branching and merging capabilities support various development workflows, such as feature branching, git flow, and trunk-based development. Teams can adopt branching strategies that suit their project requirements and release management processes.
4. **Continuous Integration and Deployment (CI/CD):** Git integrates with CI/CD pipelines to automate build, test, and deployment processes. CI/CD systems monitor Git repositories for changes, trigger automated builds and tests, and deploy changes to production environments, streamlining the software delivery pipeline.
5. **Code Review and Collaboration:** Git facilitates code review processes by providing tools for reviewing changes, commenting on code, and suggesting improvements. Code review helps maintain code quality, identify bugs, and share knowledge among team members, fostering collaboration and continuous improvement.
6. **Open Source Contribution:** Git plays a vital role in open-source projects, allowing contributors from around the world to collaborate on codebases. Contributors can fork repositories, make changes, and submit pull requests for review and inclusion in the main project, promoting community-driven development.
7. **Project Management Integration:** Git integrates with project management tools, issue trackers, and collaboration platforms to provide a unified environment for software development. Integration with tools like Jira, Trello, or GitHub Issues enables teams to track tasks, bugs, and feature requests alongside code changes.
8. **Documentation and Compliance:** Git captures detailed information about changes, including commit messages, authorship, and timestamps. This documentation serves as an audit trail for compliance purposes and helps teams understand the context and rationale behind specific modifications.
9. **Experimentation and Prototyping:** Git's lightweight branching model allows developers to create experimental branches for testing new ideas or prototypes. Developers can work on experimental features without affecting the main codebase, facilitating innovation and risk-free experimentation.
10. **Backup and Disaster Recovery:** Git serves as a backup mechanism, preserving a copy of code repositories in distributed environments. In case of data loss or system failures, developers can recover code from local or remote repositories, ensuring data integrity and business continuity.

😄 Version Control System:

A version control system (VCS) is a software tool that helps manage changes to source code, documents, or any other set of files. It enables collaboration among multiple users working

on the same files and keeps track of changes made over time. Here's a brief summary of key points about version control systems:

1. **Tracking Changes:** VCS tracks changes to files by recording modifications, additions, and deletions made by users. Each change is accompanied by metadata such as the user who made the change, the timestamp, and a descriptive message.
2. **Versioning:** VCS maintains multiple versions of files, allowing users to revert to previous states or compare different versions. This enables users to track the history of changes and identify when and why specific modifications were made.
3. **Branching and Merging:** VCS supports branching, allowing users to create separate lines of development to work on features or fixes independently. Branches can later be merged back into the main line of development, incorporating changes from multiple contributors.
4. **Collaboration:** VCS facilitates collaboration among distributed teams by providing mechanisms for sharing changes, resolving conflicts, and coordinating work on shared files. Users can work asynchronously and merge their changes seamlessly.
5. **Backup and Recovery:** VCS serves as a backup mechanism, preserving a copy of files in a centralized repository. This helps prevent data loss in case of accidental deletions or file corruption and enables recovery to a known good state.
6. **Code Review:** VCS supports code review processes by providing tools for reviewing changes, commenting on code, and suggesting improvements. This promotes code quality, knowledge sharing, and collaboration among team members.
7. **Documentation and Auditing:** VCS captures detailed information about changes, including who made each change and why. This serves as documentation of the development process and provides an audit trail for compliance and accountability purposes.
8. **Integration with Tools:** VCS integrates with other development tools and services, such as issue trackers, continuous integration (CI) systems, and project management platforms. This allows for seamless workflow automation and enhances productivity.

Common examples of version control systems include Git, Subversion (SVN), Mercurial, and Perforce. Among these, Git is one of the most widely used VCS due to its distributed nature, flexibility, and extensive ecosystem of tools and services.

😄 Use cases of Version Control System:

Version control systems (VCS) are essential tools for managing changes to source code and other files in software development projects. Here are some common use cases for version control systems:

1. **Code Management:** Version control systems are primarily used to manage changes to source code. Developers can track modifications, additions, and deletions made to files over time, enabling them to collaborate on codebases effectively.
2. **Collaborative Development:** VCS facilitates collaborative development by providing mechanisms for multiple developers to work on the same codebase simultaneously.

Developers can create branches to work on new features or bug fixes independently and merge their changes back into the main codebase when ready.

3. **Code Review:** VCS supports code review processes by allowing developers to submit changes for review, comment on code, and suggest improvements. Code review helps maintain code quality, identify bugs, and share knowledge among team members.
4. **Versioning and History:** Version control systems maintain a history of changes to files, enabling users to revert to previous versions if needed. This versioning capability provides a safety net against accidental deletions or modifications and helps track the evolution of the codebase over time.
5. **Branching Strategies:** VCS supports various branching strategies, such as feature branching, git flow, and trunk-based development. These strategies enable teams to organize their development workflow and manage the release process effectively.
6. **Continuous Integration and Deployment (CI/CD):** Version control systems integrate with CI/CD pipelines to automate build, test, and deployment processes. CI/CD systems monitor VCS repositories for changes, trigger automated builds and tests, and deploy changes to production environments.
7. **Rollback and Recovery:** In case of errors or issues introduced by new changes, version control systems allow users to rollback to a previous known good state. This rollback capability helps minimize downtime and mitigate risks during deployments.
8. **Documentation and Compliance:** VCS captures detailed information about changes, including who made each change, when it was made, and why. This documentation serves as an audit trail for compliance purposes and helps teams understand the rationale behind specific modifications.
9. **Project Management Integration:** Version control systems integrate with project management tools, issue trackers, and collaboration platforms to provide a unified environment for software development. This integration streamlines communication, task tracking, and project planning within development teams.
10. **Open Source Contribution:** Version control systems play a crucial role in open-source projects by enabling contributors from around the world to collaborate on codebases. Contributors can fork repositories, make changes, and submit pull requests for review and inclusion in the main project.

🤔 What are we doing in this Lab?

In this lab, you're diving into the world of version control using Git and GitHub. The primary objective is to grasp the essential concepts and workflows involved in managing software development projects collaboratively. Here's a brief overview of what you're doing and the end goal:

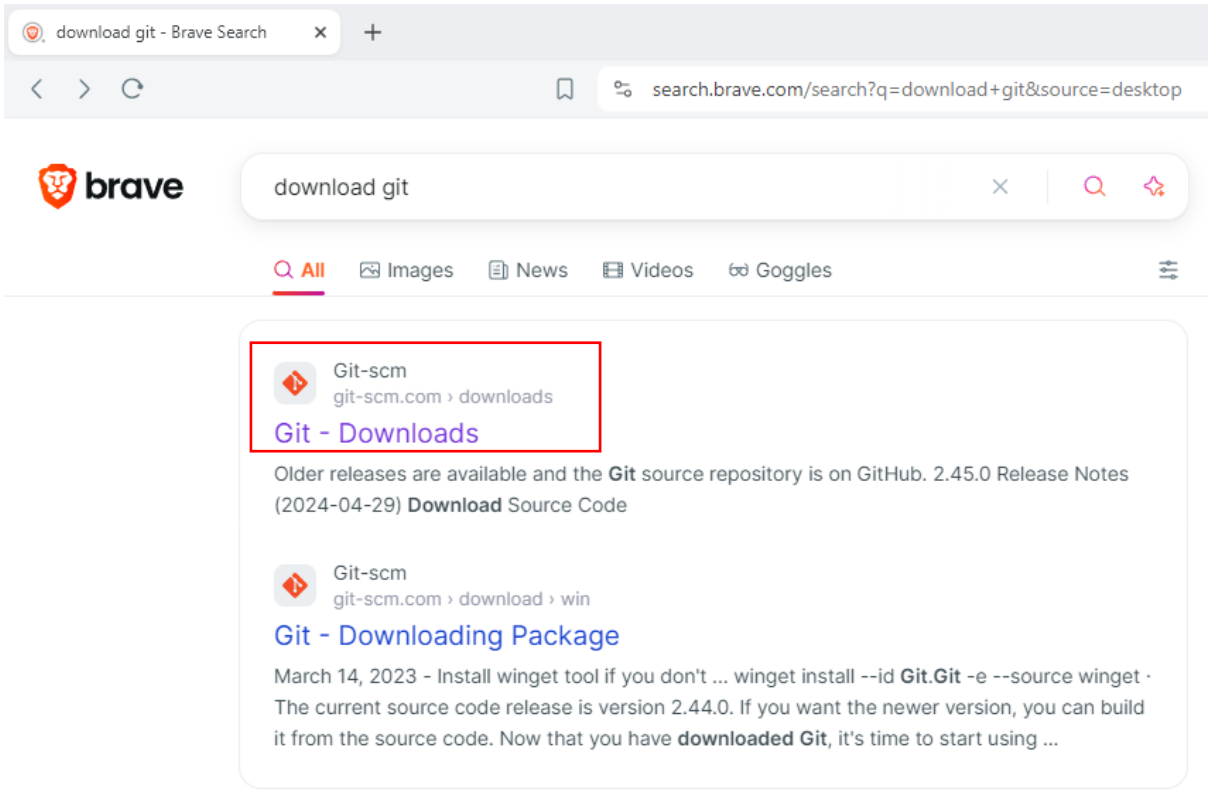
1. **Setting Up Environment:** You're starting by setting up your development environment, installing Git on your local machine, and creating a GitHub account. This step ensures you have the necessary tools to work with Git and collaborate on projects hosted on GitHub.

2. **Creating a Repository:** You're creating a new Git repository, both locally on your machine and remotely on GitHub. This repository serves as a centralized location to store your project files and track changes over time.
3. **Tracking Changes:** You're learning how to track changes to files using Git. By adding files to the staging area and committing them, you're creating snapshots of your project's state at different points in time. This enables you to review changes, revert to previous versions if needed, and maintain a detailed history of your project.
4. **Collaborating on GitHub:** You're exploring GitHub's collaboration features by pushing your local repository to a remote repository hosted on GitHub. This allows you to share your project with others, collaborate on code, and coordinate development efforts seamlessly.
5. **Managing Project History:** You're using Git commands to view the project's commit history, examine differences between file versions, and understand how your project has evolved over time. This helps you track progress, identify issues, and maintain code quality throughout the development process.
6. **Real-world Scenarios:** Through practical exercises, you're simulating real-world development scenarios such as making changes locally, pushing them to GitHub, pulling changes from GitHub, and resolving conflicts. This hands-on experience prepares you to handle common challenges encountered in collaborative software development environments.

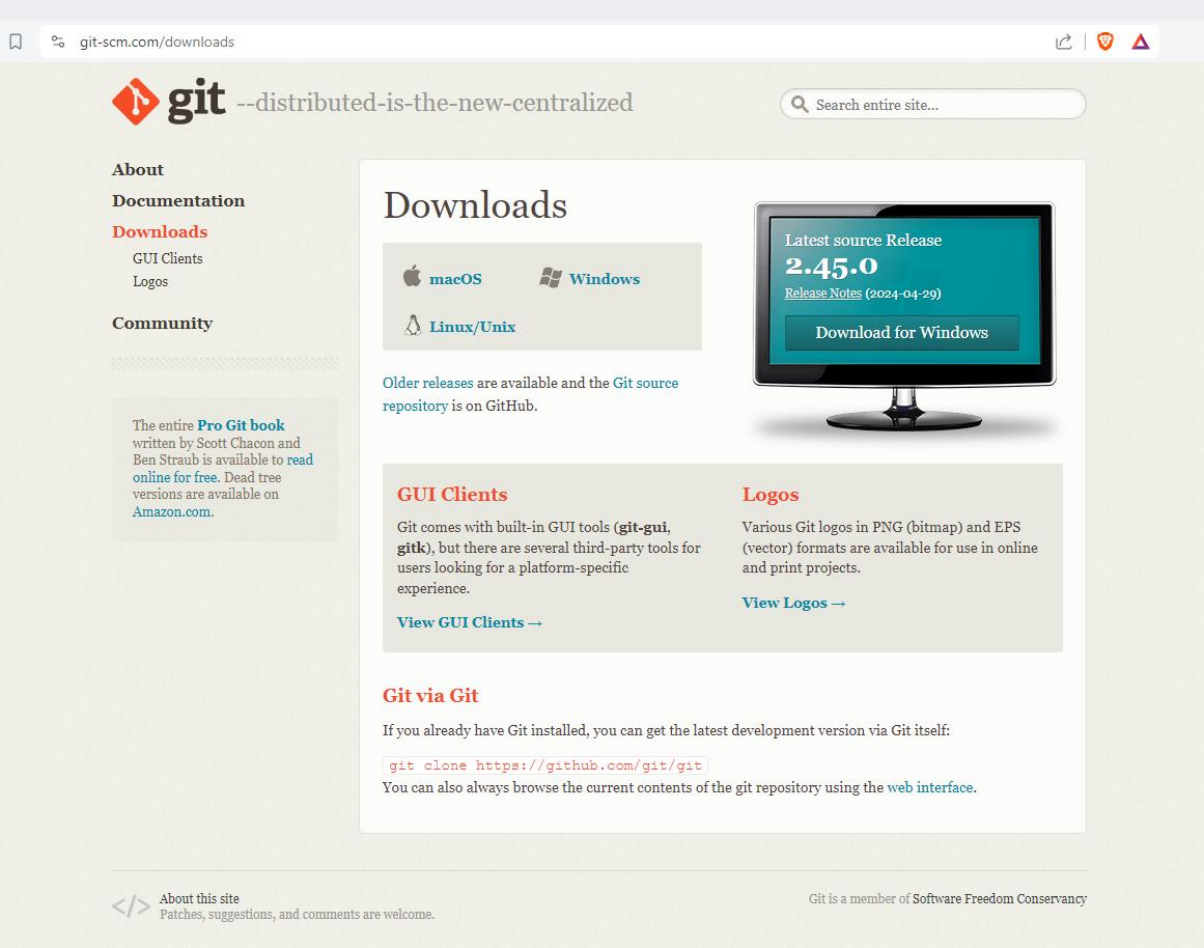
By the end of the lab, you'll have a solid understanding of version control concepts, Git workflows, and GitHub collaboration practices. You'll be equipped with the skills to manage projects effectively, collaborate with team members, and contribute to open-source projects with confidence.

😊 To begin with the Lab:

1. Now to start with this lab you need to have a GitHub account in place. For that go to github.com and create a new account if you haven’t already.
2. After that you need to have gitbash in your local machine. For that search download git on Google and open the first website that you will see.



3. Then according to your operating system choose from the downloads options and download git. After that, you need to install it.



4. Once git has been installed in your system, then you need to open GitBash terminal from any of your local disks. After choosing to go into any folder that you wish to.
5. Then we are going to create a new directory. Once it is created then we need to go into that directory.

```
PULKIT@LAPTOP-G2CAKBK8 MINGW64 ~/Desktop
$ mkdir gitrepository

PULKIT@LAPTOP-G2CAKBK8 MINGW64 ~/Desktop
$ cd gitrepository
```

6. After that we will be creating a repository which will be our work repository.
7. So, we created a repository, and we get inside of it. After that we issue the command git init.

```
PULKIT@LAPTOP-G2CAKBK8 MINGW64 ~/Desktop/gitrepository
$ mkdir sattelitework

PULKIT@LAPTOP-G2CAKBK8 MINGW64 ~/Desktop/gitrepository
$ cd sattelitework

PULKIT@LAPTOP-G2CAKBK8 MINGW64 ~/Desktop/gitrepository/sattelitework
$ git init
Initialized empty Git repository in C:/Users/PULKIT/Desktop/gitrepository/sattelitework/.git/
```

8. So, this git init command it should create a .git directory in your current directory, so ls -a, if you do, you should see a .git directory and this is going to maintain all the visions, all the history, all the configuration of your git repository.

```
$ ls -a
./   ../   .git/
```

9. Then if you will see it is calling our work directory as master.

```
PULKIT@LAPTOP-G2CAKBK8 MINGW64 ~/Desktop/gitrepository/sattelitework (master)
$
```

10. After that we created 3 directories and 10 files as you can see down below.

```
PULKIT@LAPTOP-G2CAKBK8 MINGW64 ~/Desktop/gitrepository/sattelitework (master)
$ mkdir insat1a

PULKIT@LAPTOP-G2CAKBK8 MINGW64 ~/Desktop/gitrepository/sattelitework (master)
$ mkdir scross1

PULKIT@LAPTOP-G2CAKBK8 MINGW64 ~/Desktop/gitrepository/sattelitework (master)
$ mkdir gsat9

PULKIT@LAPTOP-G2CAKBK8 MINGW64 ~/Desktop/gitrepository/sattelitework (master)
$ ls
gsat9/  insat1a/  scross1/

PULKIT@LAPTOP-G2CAKBK8 MINGW64 ~/Desktop/gitrepository/sattelitework (master)
$ touch moon{1..10}.py

PULKIT@LAPTOP-G2CAKBK8 MINGW64 ~/Desktop/gitrepository/sattelitework (master)
$ ls
gsat9/  insat1a/  moon1.py  moon10.py  moon2.py  moon3.py  moon4.py  moon5.py  moon6.py  moon7.py  moon8.py  moon9.py  scross1/
```

11. Then we created some empty files in our 3 directories.

```
PULKIT@LAPTOP-G2CAKBK8 MINGW64 ~/Desktop/gitrepository/sattelitework (master)
$ touch insat1a/test1

PULKIT@LAPTOP-G2CAKBK8 MINGW64 ~/Desktop/gitrepository/sattelitework (master)
$ touch scross1/test2

PULKIT@LAPTOP-G2CAKBK8 MINGW64 ~/Desktop/gitrepository/sattelitework (master)
$ touch gsat9/test3
```

12. Now we have made some changes. So, we are going to run this command to check for the status.
13. So, if you run this command it should tell you what the current state of your repository is and here, we can see that no commits have been done yet and they are untracked files. That is why these files have red color as a symbol.

```
PULKIT@LAPTOP-G2CAKBK8 MINGW64 ~/Desktop/gitrepository/sattelitework (master)
$ git status
On branch master

No commits yet

Untracked files:
  (use "git add <file>..." to include in what will be committed)
        gsat9/
        insat1a/
        moon1.py
        moon10.py
        moon2.py
        moon3.py
        moon4.py
        moon5.py
        moon6.py
        moon7.py
        moon8.py
        moon9.py
        scross1/

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

14. Now we will run this command below, this dot here means the current working directory.

    git add .

15. Then you have to run the status command and you will see that the color has changed to green which means that these files are now tracked. This is also called as staging.

```
PULKIT@LAPTOP-G2CAKBK8 MINGW64 ~/Desktop/gitrepository/sattelitework (master)
$ git add .

PULKIT@LAPTOP-G2CAKBK8 MINGW64 ~/Desktop/gitrepository/sattelitework (master)
$ git status
On branch master

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
        new file:   gsat9/test3
        new file:   insat1a/test1
        new file:   moon1.py
        new file:   moon10.py
        new file:   moon2.py
        new file:   moon3.py
        new file:   moon4.py
        new file:   moon5.py
        new file:   moon6.py
        new file:   moon7.py
        new file:   moon8.py
        new file:   moon9.py
        new file:   scross1/test2
```

16. Then you are going to commit these files. Below you can see that we have successfully committed all the files. This -m means message and you have to give it mostly every time you commit your files.

    git commit -m “new files commit”

```
PULKIT@LAPTOP-G2CAKBK8 MINGW64 ~/Desktop/gitrepository/sattelitework (master)
$ git commit -m "new files commit"
[master (root-commit) 10cf38b] new files commit
 13 files changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 gsat9/test3
 create mode 100644 insat1a/test1
 create mode 100644 moon1.py
 create mode 100644 moon10.py
 create mode 100644 moon2.py
 create mode 100644 moon3.py
 create mode 100644 moon4.py
 create mode 100644 moon5.py
 create mode 100644 moon6.py
 create mode 100644 moon7.py
 create mode 100644 moon8.py
 create mode 100644 moon9.py
 create mode 100644 scross1/test2
```

17. Now usually when you are using git for the first time it generally throws an error message like this.
18. So, when git is tracking, it will also track who made the change. So, for that, who, do you have to set who is the user and what is the email address of it?
19. Below you can see that in the highlighted area it has given you the command to set your email and username.
20. Here you have to write your email address and the username of your GitHub repository. And also, these are your global settings and it is for all the repositories.
21. After that run git commit command again and then you will see that it was successful.

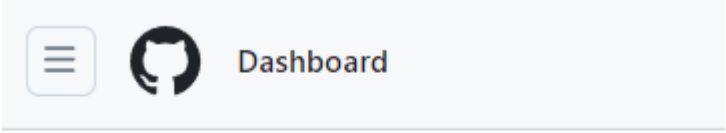
    git config --global user.email "your@email"
    git config --global user.name "Your name"

```
$ git commit -m "new files commited"
Author identity unknown

*** Please tell me who you are.

Run

  git config --global user.email "you@example.com"
  git config --global user.name "Your Name"

to set your account's default identity.
Omit --global to set the identity only in this repository.
```

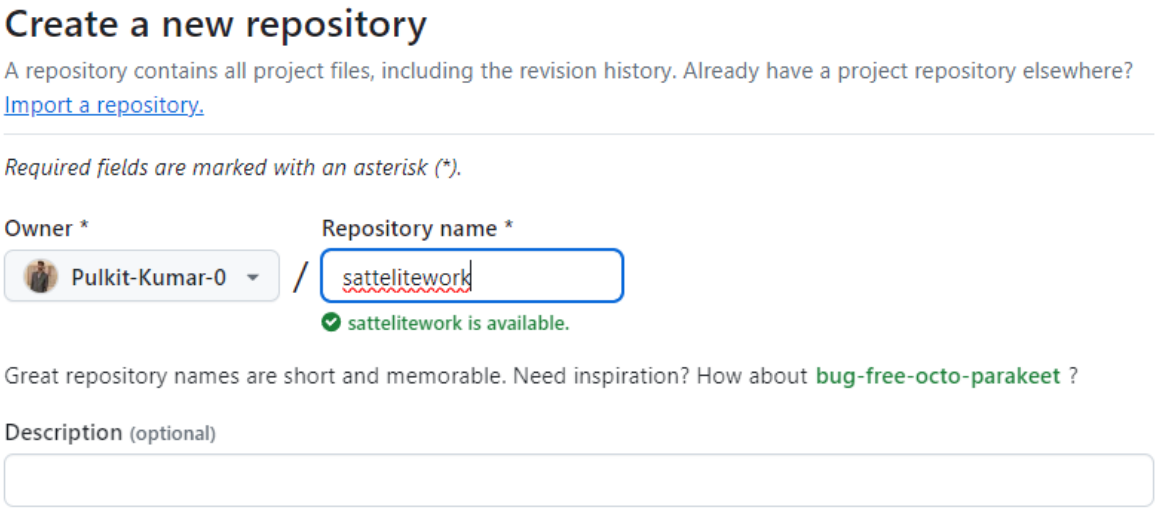
22. Now if you run the git status command then you will see that is gives you message that nothing to commit.

```
$ git status
On branch master
nothing to commit, working tree clean
```

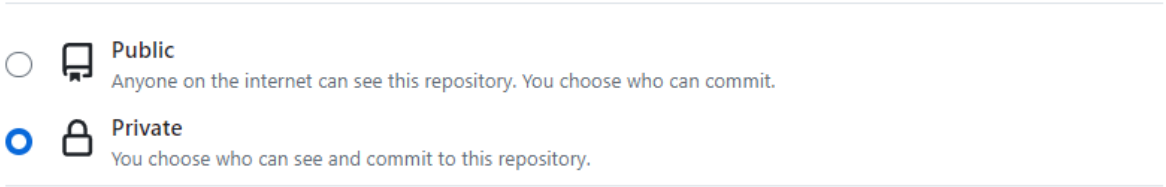
23. Now you need to navigate to your browser, search github.com, and open your GitHub account.
24. Then we are going to create a new repository there. Now from the dashboard of your GitHub click on this highlighted icon, it means that we are creating a new repository.



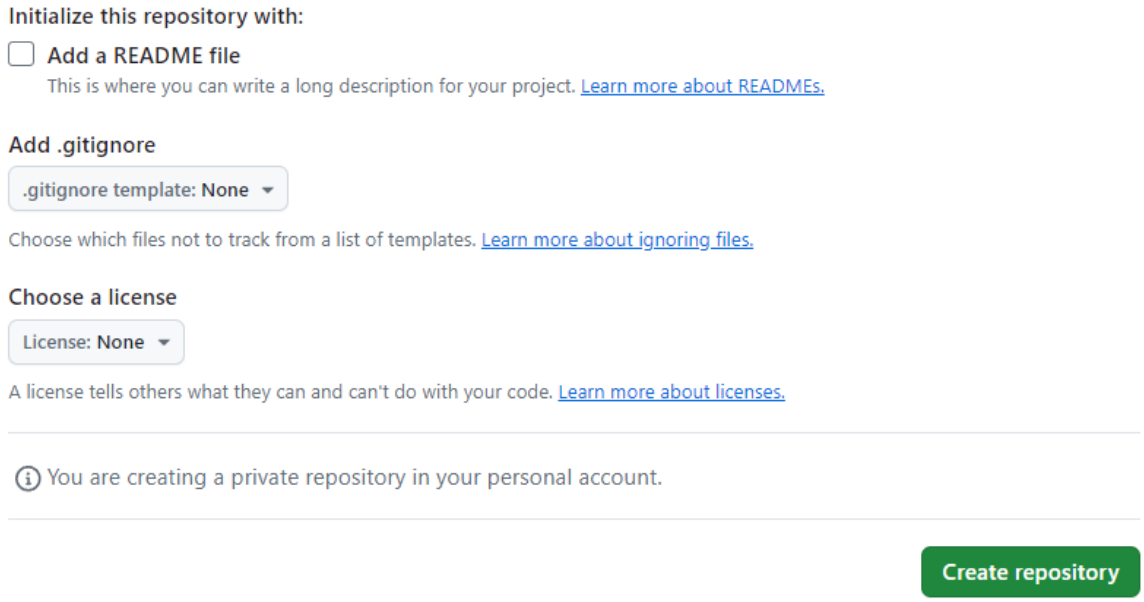
25. The first thing you need to do is give your repository the same name because we are going to sync both of the repositories.
26. Here you will see that it is saying to me that the name I have chosen is available which means GitHub repositories are global and require a unique name.



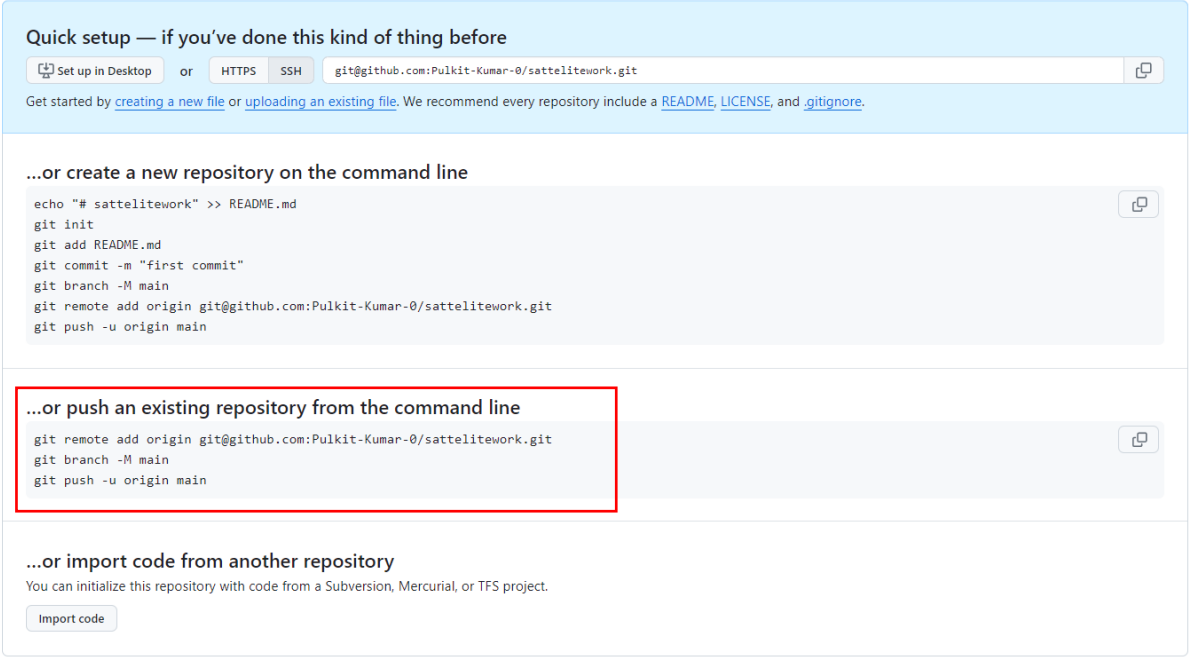
27. Then we have two options, either we can choose public or private. Public means that it will be visible to everyone and private means that it will be accessible to those to whom we have given access.
28. For this lab we are going to keep our repository private.



29. After that we will keep everything to default and create our first repository. So hit on create.



30. So, once our repository is created it will give us two option that we can create a new repository on the command line which could be PowerShell, or Gitbash.
31. Or we can push an existing repository. So, we have a repository and we will be pushing it to GitHub.
32. Now you need to copy the highlighted commands from your repository and paste them one by one on your gitbash terminal.



33. Now once we run the first command to add origin. Then there is a git config file, if you will see the contents of this file in then you will notice that it got a remote origin path.

```
PULKIT@LAPTOP-G2CAKBK8 MINGW64 ~/Desktop/gitrepository/sattelitework (master)
$ git remote add origin git@github.com:Pulkit-Kumar-0/sattelitework.git

PULKIT@LAPTOP-G2CAKBK8 MINGW64 ~/Desktop/gitrepository/sattelitework (master)
$ cat .git/config
[core]
        repositoryformatversion = 0
        filemode = false
        bare = false
        logallrefupdates = true
        symlinks = false
        ignorecase = true
[remote "origin"]
        url = git@github.com:Pulkit-Kumar-0/sattelitework.git
        fetch = +refs/heads/*:refs/remotes/origin/*
```

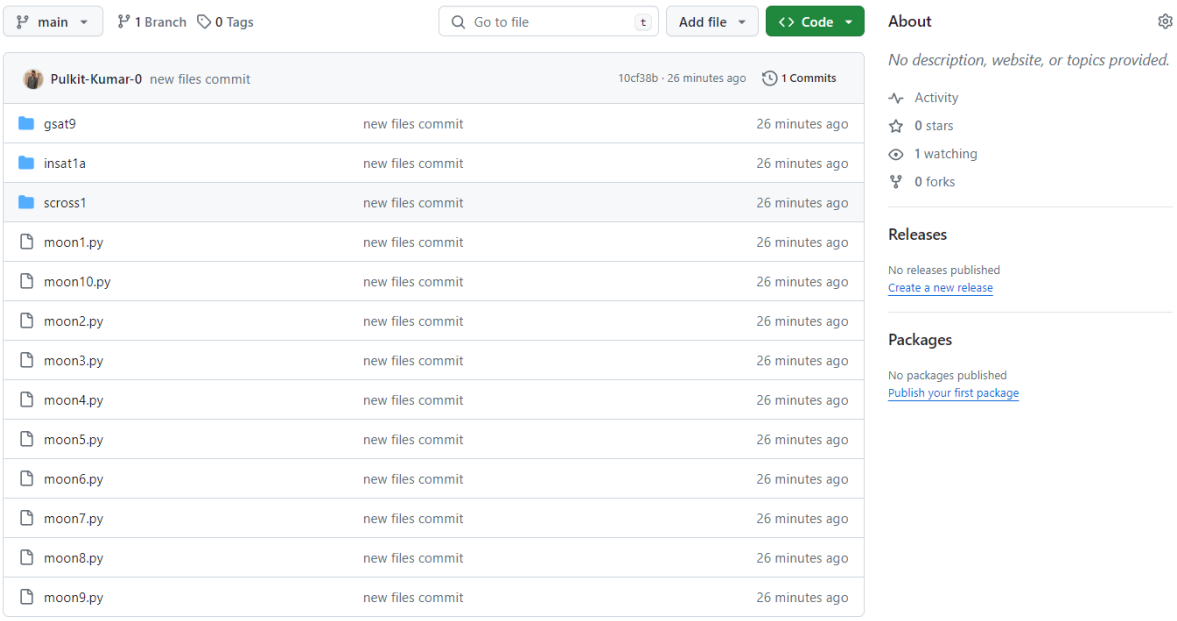
34. Then the next command will change your branch name from master to main.

```
PULKIT@LAPTOP-G2CAKBK8 MINGW64 ~/Desktop/gitrepository/sattelitework (master)
$ git branch -M main

PULKIT@LAPTOP-G2CAKBK8 MINGW64 ~/Desktop/gitrepository/sattelitework (main)
$
```

35. Then the last command is to push all the data to your GitHub repository.

```
PULKIT@LAPTOP-G2CAKBK8 MINGW64 ~/Desktop/gitrepository/sattelitework (main)
$ git push -u origin main
Enumerating objects: 6, done.
Counting objects: 100% (6/6), done.
Delta compression using up to 12 threads
Compressing objects: 100% (2/2), done.
Writing objects: 100% (6/6), 480 bytes | 480.00 KiB/s, done.
Total 6 (delta 0), reused 0 (delta 0), pack-reused 0
To github.com:Pulkit-Kumar-0/sattelitework.git
 * [new branch]      main -> main
branch 'main' set up to track 'origin/main'.
```

36. Again, when you will run the last command then it will ask you to authenticate so you need to authenticate yourself. This only happens when you are using git for the first time.
37. Now the changes have been saved to the remote repository, you need to refresh your GitHub on the browser, and below you can see all the data in place.



38. Now we are going to make some text-based changes in our files. First, do the listing of files in your directory then choose a file of your choice.
39. Then run the vim command to get inside of it and write some text.

```
PULKIT@LAPTOP-G2CAKBK8 MINGW64 ~/Desktop/gitrepository/sattelitework (main)
$ ls
gsat9/  insat1a/  moon1.py  moon10.py  moon2.py  moon3.py  moon4.py  moon5.py  moon6.py  moon7.py  moon8.py  moon9.py  scross1/

PULKIT@LAPTOP-G2CAKBK8 MINGW64 ~/Desktop/gitrepository/sattelitework (main)
$ vim moon1.py
```

40. Then write some text in it. After that save it and quit. Now if you are wondering how to write anything here, for that you need to hit **insert key** on your keyboard, then you will be able to write something here. And to save your file first you need to click on **Esc** button on your keyboard and write **:wq** then press enter.

```
INSAT-1A, launched in 1982, became India's first operational multipurpose communication and meteorology satellite, rev
olutionizing telecommunication.
~
```

41. Now run the git status command and you will see which file has been modified.

```
$ git status
On branch main
Your branch is up to date with 'origin/main'.

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   moon1.py

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

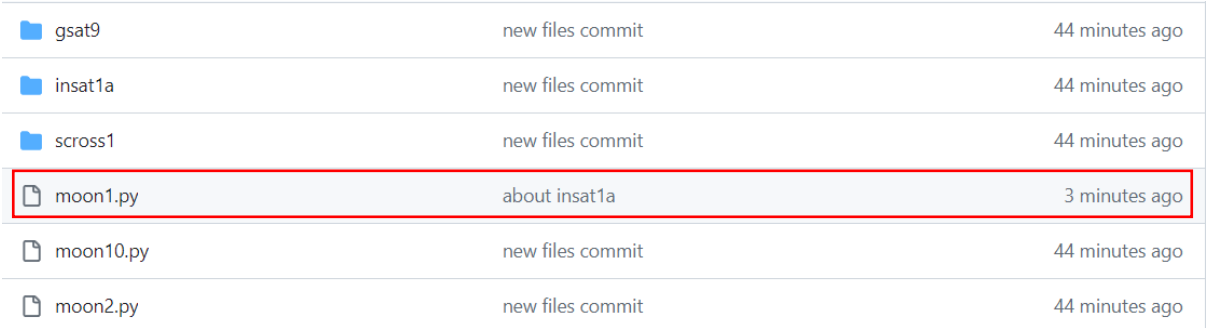
42. Now if you need to run the git add command with the file name or give dot instead of the file name.
43. After that we are going to give the commit command. So, this commit command worked locally not globally which means that we need to run push command in order to push changes on our GitHub repository.

```
$ git add moon1.py
warning: in the working copy of 'moon1.py', LF will be replaced by CRLF the next time Git touches it

PULKIT@LAPTOP-G2CAKBK8 MINGW64 ~/Desktop/gitrepository/sattelitework (main)
$ git commit -m "about insat1a"
[main bccee09] about insat1a
 1 file changed, 1 insertion(+)
```

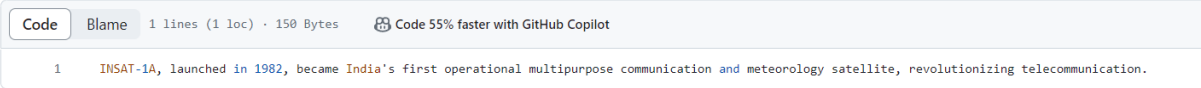
44. Below you can see that we run the push command to push the changes we made to our files.

```
PULKIT@LAPTOP-G2CAKBK8 MINGW64 ~/Desktop/gitrepository/sattelitework (main)
$ git push origin main
Enumerating objects: 5, done.
Counting objects: 100% (5/5), done.
Delta compression using up to 12 threads
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 381 bytes | 381.00 KiB/s, done.
Total 3 (delta 1), reused 0 (delta 0), pack-reused 0
remote: Resolving deltas: 100% (1/1), completed with 1 local object.
To github.com:Pulkit-Kumar-0/sattelitework.git
   10cf38b..bccee09  main -> main
```

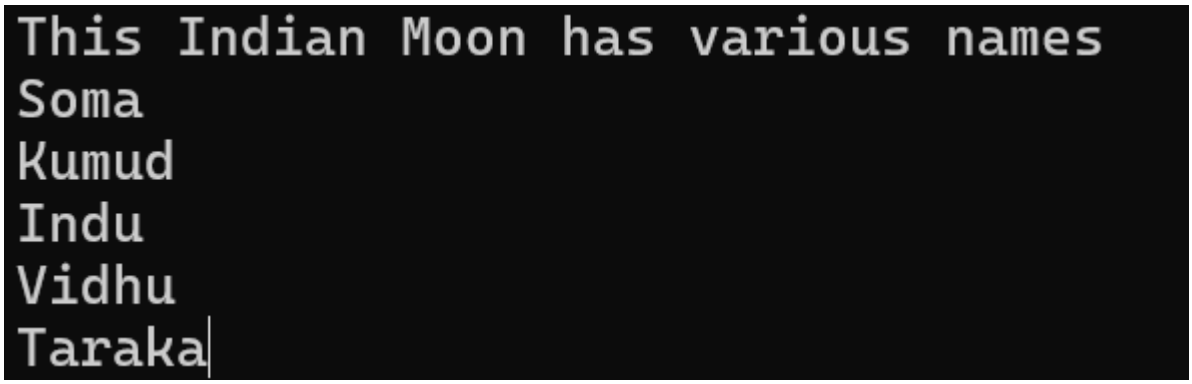
45. Now come back to your GitHub repository and refresh the page, here you will see that the commit message and the time has also changed.



46. And if you open the file, you can also see the changes made in it.



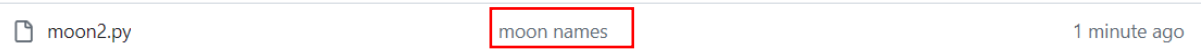
47. Let’s try to make some more changes to other of the files. Choose any file of your choice use the vim command and write something in it then just save and quit.

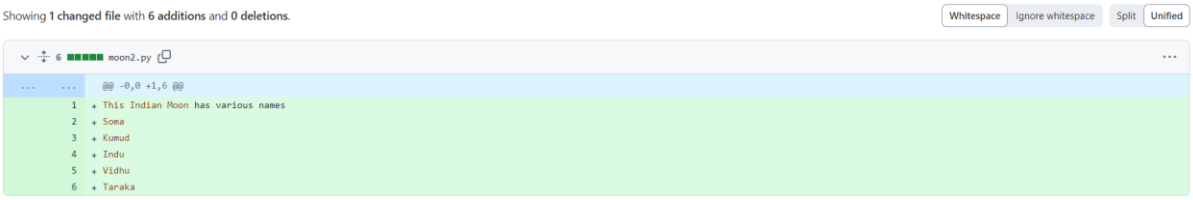


48. Then the process is same as before check its status, add this file then commit it and finally push it.

```
PULKIT@LAPTOP-G2CAKBK8 MINGW64 ~/Desktop/gitrepository/sattelitework (main)
$ git status
On branch main
Your branch is up to date with 'origin/main'.

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   moon2.py

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

PULKIT@LAPTOP-G2CAKBK8 MINGW64 ~/Desktop/gitrepository/sattelitework (main)
$ git add moon2.py
warning: in the working copy of 'moon2.py', LF will be replaced by CRLF the next time Git touches it

PULKIT@LAPTOP-G2CAKBK8 MINGW64 ~/Desktop/gitrepository/sattelitework (main)
$ git commit -m "moon names"
[main fa53ffb] moon names
 1 file changed, 6 insertions(+)

PULKIT@LAPTOP-G2CAKBK8 MINGW64 ~/Desktop/gitrepository/sattelitework (main)
$ git push origin main
Enumerating objects: 5, done.
Counting objects: 100% (5/5), done.
Delta compression using up to 12 threads
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 323 bytes | 323.00 KiB/s, done.
Total 3 (delta 1), reused 0 (delta 0), pack-reused 0
remote: Resolving deltas: 100% (1/1), completed with 1 local object.
To github.com:Pulkit-Kumar-0/sattelitework.git
   bccee09..fa53ffb  main -> main
```

49. Below you can see the file to which we had made changes. Now click on the commit message and you will be able to see changes.





50. You can also see the same from the command line.

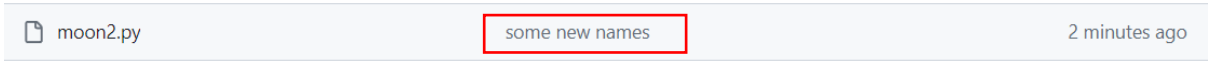
    git log –oneline
    git show fa53ffb

```
PULKIT@LAPTOP-G2CAKBK8 MINGW64 ~/Desktop/gitrepository/sattelitework (main)
$ git log --oneline
fa53ffb (HEAD -> main, origin/main) moon names
bccee09 about insat1a
10cf38b new files commit

PULKIT@LAPTOP-G2CAKBK8 MINGW64 ~/Desktop/gitrepository/sattelitework (main)
$ git show fa53ffb
commit fa53ffb506a9c58bd08938c6d1a8b289cd8155e6 (HEAD -> main, origin/main)
Author: Pulki-Kumar-0 <pulkitkumar2711@gmail.com>
Date:   Tue Apr 30 18:07:24 2024 +0530

    moon names

diff --git a/moon2.py b/moon2.py
index e69de29..e4176e5 100644
--- a/moon2.py
+++ b/moon2.py
@@ -0,0 +1,6 @@
+This Indian Moon has various names
+Soma
+Kumud
+Indu
+Vidhu
+Taraka
```

51. Again, we are going to make some changes, but this time in the same file. Here we will be using the file with more data which is moon2.py, open it using the vim command and make some changes according to you. Maybe add some files and remove some.
52. Once your changes are made then repeat the process and push the changes to GitHub.

```
$ git status
On branch main
Your branch is up to date with 'origin/main'.

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   moon2.py

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

PULKIT@LAPTOP-G2CAKBK8 MINGW64 ~/Desktop/gitrepository/sattelitework (main)
$ git add moon2.py
warning: in the working copy of 'moon2.py', LF will be replaced by CRLF the next time Git touches it

PULKIT@LAPTOP-G2CAKBK8 MINGW64 ~/Desktop/gitrepository/sattelitework (main)
$ git commit -m "some new names"
[main b3a4821] some new names
 1 file changed, 2 insertions(+), 2 deletions(-)

PULKIT@LAPTOP-G2CAKBK8 MINGW64 ~/Desktop/gitrepository/sattelitework (main)
$ git push origin main
Enumerating objects: 5, done.
Counting objects: 100% (5/5), done.
Delta compression using up to 12 threads
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 331 bytes | 331.00 KiB/s, done.
Total 3 (delta 1), reused 0 (delta 0), pack-reused 0
remote: Resolving deltas: 100% (1/1), completed with 1 local object.
To github.com:Pulkit-Kumar-0/sattelitework.git
   fa53ffb..b3a4821  main -> main
```

53. Below you can see that the changes have been made. Then click on the commit message. Also, you will see what are changes that have been made.
54. What is removed and what is added.



Showing **1 changed file** with **2 additions** and **2 deletions**.

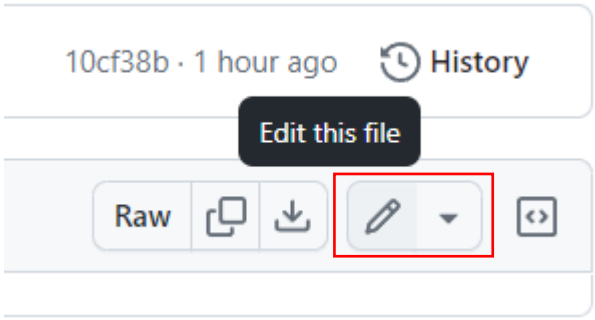
4 moon2.py

```
...  ...  @@ -1,6 +1,6 @@
1    1    This Indian Moon has various names
2    2    Soma
3    3    Kumud
4         - Indu
     4    + Shining
5    5    Vidhu
6         - Taraka
     6    + Chandrama
```

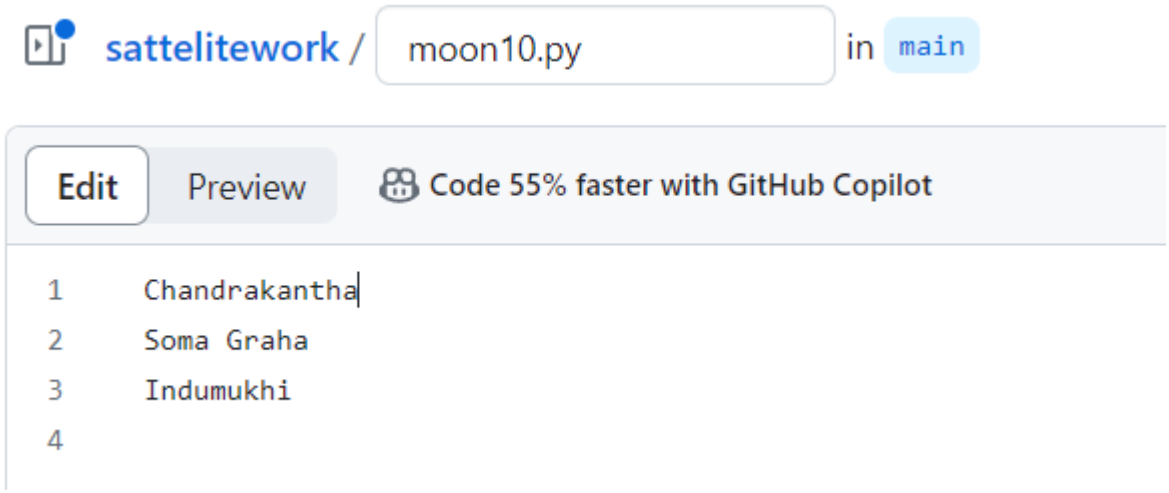
55. The same things can be seen on the command line.

```
PULKIT@LAPTOP-G2CAKBK8 MINGW64 ~/Desktop/gitrepository/sattelitework (main)
$ git log --oneline
b3a4821 (HEAD -> main, origin/main) some new names
fa53ffb moon names
bccee09 about insat1a
10cf38b new files commit

PULKIT@LAPTOP-G2CAKBK8 MINGW64 ~/Desktop/gitrepository/sattelitework (main)
$ git show b3a4821
commit b3a4821b7f8069615508ddcce2f5769fbd0a71fc (HEAD -> main, origin/main)
Author: Pulki-Kumar-0 <pulkitkumar2711@gmail.com>
Date:   Tue Apr 30 18:33:05 2024 +0530

    some new names

diff --git a/moon2.py b/moon2.py
index e4176e5..7219410 100644
--- a/moon2.py
+++ b/moon2.py
@@ -1,6 +1,6 @@
 This Indian Moon has various names
 Soma
 Kumud
-Indu
+Shining
 Vidhu
-Taraka
+Chandrama
```

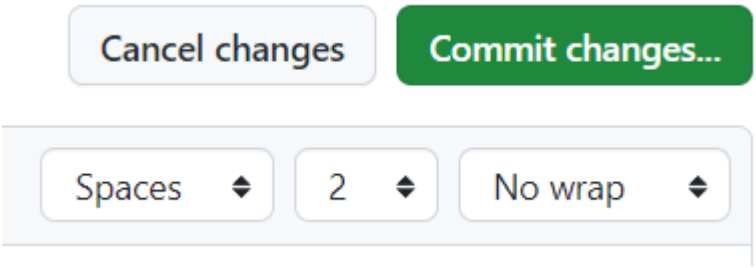
56. Now we are going to add some data in the files from GitHub itself. Go to the browser and open any file of your choice then click on the pencil icon to edit that file.



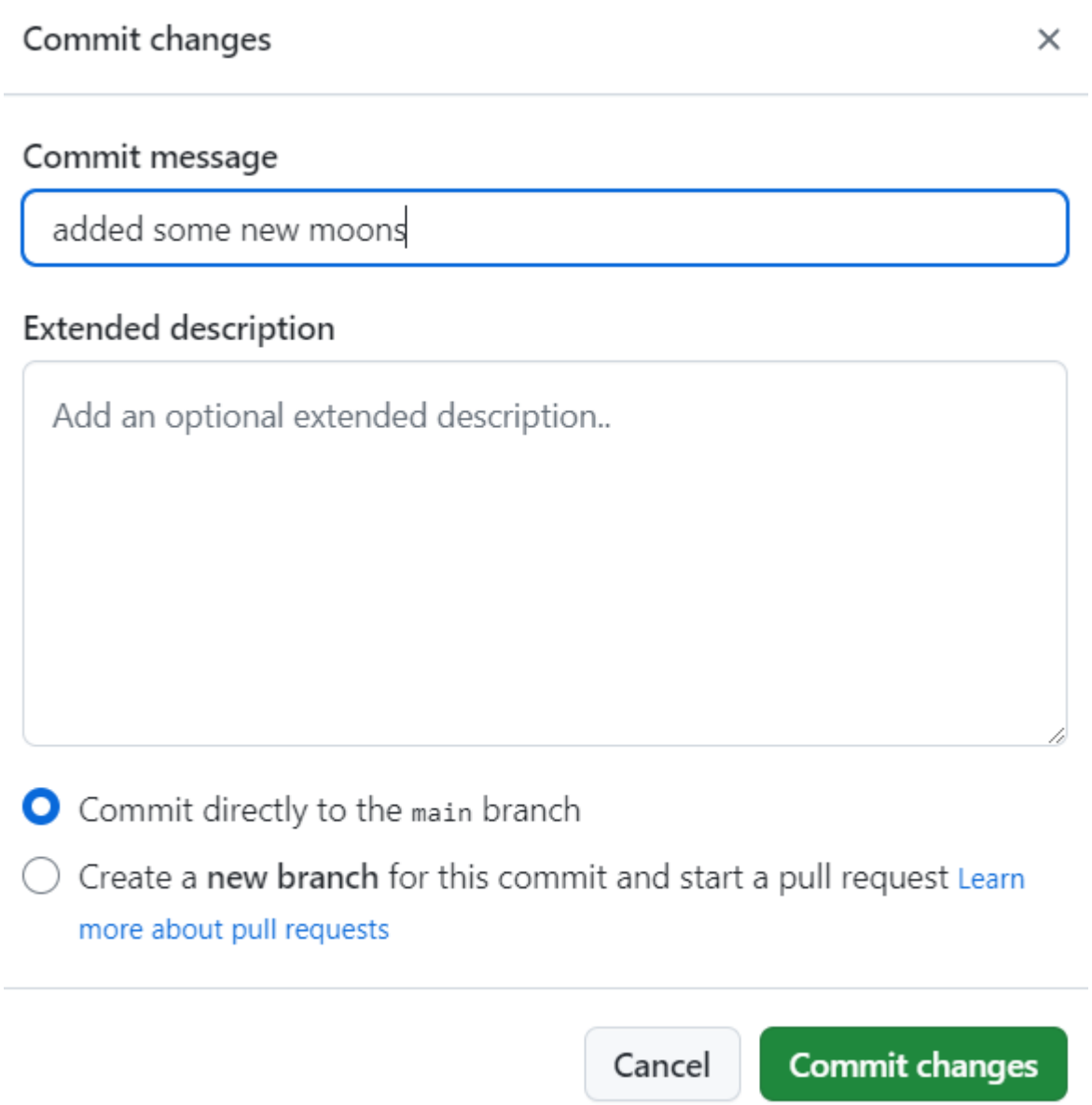
57. Below you can see that we have made some changes in this file.



58. Then from the right side of the web page click on commit changes.



59. Then it will ask you to write a commit message. Then click on commit changes.



60. So, these changes are made directly onto our repository. In real-time, this could be done by anybody who has access to your repository. So, to get these changes in your local repository you need to run the git pull command and it will pull all the changes made in your repository. Also, it will tell you how many files have been changed, how many insertions were done.

```
$ git pull
remote: Enumerating objects: 5, done.
remote: Counting objects: 100% (5/5), done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 3 (delta 1), reused 0 (delta 0), pack-reused 0
Unpacking objects: 100% (3/3), 951 bytes | 190.00 KiB/s, done.
From github.com:Pulkit-Kumar-0/sattelitework
   b3a4821..ccc886c  main       -> origin/main
Updating b3a4821..ccc886c
Fast-forward
 moon10.py | 3 +++
 1 file changed, 3 insertions(+)
```

61. Now if you want to check that file use the cat command with the file name. Then if you run the git log command, you will be able to see all the changes that were made.

```
$ cat moon10.py
Chandrakantha
Soma Graha
Indumukhi
```

```
PULKIT@LAPTOP-G2CAKBK8 MINGW64 ~/Desktop/gitrepository/sattelitework (main)
$ git log
commit ccc886cb97628ef9607615bb0ccc36e772e2eaed (HEAD -> main, origin/main)
Author: Pulkit Kumar <95499378+Pulkit-Kumar-0@users.noreply.github.com>
Date:   Tue Apr 30 18:44:22 2024 +0530

    added some new moons

commit b3a4821b7f8069615508ddcce2f5769fbd0a71fc
Author: Pulki-Kumar-0 <pulkitkumar2711@gmail.com>
Date:   Tue Apr 30 18:33:05 2024 +0530

    some new names

commit fa53ffb506a9c58bd08938c6d1a8b289cd8155e6
Author: Pulki-Kumar-0 <pulkitkumar2711@gmail.com>
Date:   Tue Apr 30 18:07:24 2024 +0530

    moon names

commit bccee099e4fb61a794a547b22e58f4dbe2242d54
Author: Pulki-Kumar-0 <pulkitkumar2711@gmail.com>
Date:   Tue Apr 30 17:56:31 2024 +0530

    about insat1a

commit 10cf38bd8c04c0ef8e84e58e0cde64b9cadd0999
Author: Pulki-Kumar-0 <pulkitkumar2711@gmail.com>
Date:   Tue Apr 30 17:15:21 2024 +0530

    new files commit
```