

Assignment-1

Software Tools and Techniques for CSE

Shardul Junagade

September 5, 2025

Contents

1	Lab 1: Introduction to Version Controlling, Git Workflows, and Actions	2
1.1	Introduction, Setup, and Tools	2
1.1.1	Introduction	2
1.1.2	Setup and Tools	2
1.2	Methodology and Execution	3
1.2.1	Initializing a Local Repository	3
1.2.2	Local Git Configuration	3
1.2.3	Adding and Committing Files	3
1.2.4	Working with Remote Repository	4
1.2.5	GitHub Actions – Pylint Workflow	7
1.3	Results and Analysis	11
1.4	Discussion and Conclusion	12
1.5	References (1 Mark for Format + Style)	12

1 Lab 1: Introduction to Version Controlling, Git Workflows, and Actions

1.1 Introduction, Setup, and Tools

1.1.1 Introduction

The objective of this lab was to gain hands-on experience with Version Control Systems (VCS), specifically Git, and to explore the integration of GitHub with automated workflows such as GitHub Actions. My aim was to understand fundamental concepts like repositories, commits, branches, and remotes, and to perform essential Git operations. Additionally, I set up a continuous integration workflow using GitHub Actions and Pylint to ensure code quality.

1.1.2 Setup and Tools

I logged in to my GitHub account in the web browser – ShardulJunagade – for version control. I installed Git and Visual Studio Code by downloading the installers from their official websites and following the installation instructions. After installation, I verified the installation by checking the versions of Git and Visual Studio Code.

The following is a list of the tools and technologies used in this lab:

- Operating System: Windows 11
- Terminal: PowerShell 7
- Git version: 2.42.0
- Code Editor: Visual Studio Code
- Python version: 3.13.7
- GitHub for remote repository hosting
- GitHub Actions for CI/CD pipelines

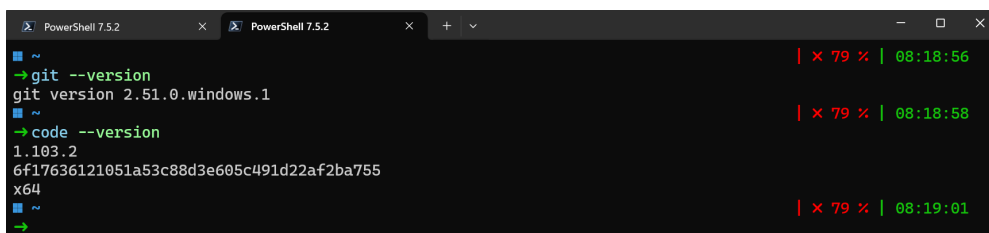

A screenshot of a PowerShell 7.5.2 terminal window. The terminal shows the command 'git --version' being executed, resulting in 'git version 2.51.0.windows.1'. Then, the command 'code --version' is executed, resulting in '1.103.2' followed by a long alphanumeric string '6f17636121051a53c88d3e605c491d22af2ba755' and 'x64'. The terminal has a dark background with green text for commands and white text for output. On the right side of the terminal, there are red and green status indicators and timestamps: '| x 79 x | 08:18:56', '| x 79 x | 08:18:58', and '| x 79 x | 08:19:01'.

Figure 1: Verifying Git installation and version.

1.2 Methodology and Execution

1.2.1 Initializing a Local Repository

I created a new folder for the lab, named `Any_Name` using the command `mkdir Any_Name` and initialized the folder as a git repository using the command `git init`.



```
PowerShell 7.5.2
~\Desktop\sem-5\cs202-stt\lab1_git :: $main = 75
→ mkdir "Any_Name"

Directory: C:\Users\shardul\Desktop\sem-5\cs202-stt\lab1_git

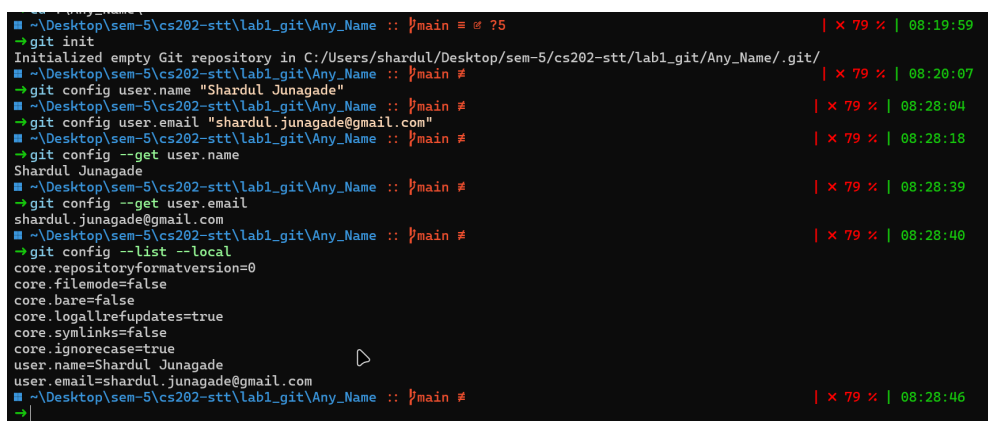
Mode                LastWriteTime         Length Name
----                -
d-----          29-08-2025    08:19             Any_Name

~\Desktop\sem-5\cs202-stt\lab1_git :: $main = 75
→ cd .\Any_Name\
~\Desktop\sem-5\cs202-stt\lab1_git\Any_Name :: $main = 75
→ git init
Initialized empty Git repository in C:/Users/shardul/Desktop/sem-5/cs202-stt/lab1_git/Any_Name/.git/
~\Desktop\sem-5\cs202-stt\lab1_git\Any_Name :: $main = 75
```

Figure 2: Initializing a new Git repository.

1.2.2 Local Git Configuration

Since my global Git configuration was already set up from previous work, I decided to configure my name and email specifically for this repository using the local configuration commands `git config user.name` and `git config user.email`. This ensures that all commits in this repository are attributed with the correct identity, regardless of the global settings.



```
~\Desktop\sem-5\cs202-stt\lab1_git\Any_Name :: $main = 75
→ git init
Initialized empty Git repository in C:/Users/shardul/Desktop/sem-5/cs202-stt/lab1_git/Any_Name/.git/
~\Desktop\sem-5\cs202-stt\lab1_git\Any_Name :: $main = 75
→ git config user.name "Shardul Junagade"
~\Desktop\sem-5\cs202-stt\lab1_git\Any_Name :: $main = 75
→ git config user.email "shardul.junagade@gmail.com"
~\Desktop\sem-5\cs202-stt\lab1_git\Any_Name :: $main = 75
→ git config --get user.name
Shardul Junagade
~\Desktop\sem-5\cs202-stt\lab1_git\Any_Name :: $main = 75
→ git config --get user.email
shardul.junagade@gmail.com
~\Desktop\sem-5\cs202-stt\lab1_git\Any_Name :: $main = 75
→ git config --list --local
core.repositoryformatversion=0
core.filemode=false
core.bare=false
core.logallrefupdates=true
core.symlinks=false
core.ignorecase=true
user.name=Shardul Junagade
user.email=shardul.junagade@gmail.com
~\Desktop\sem-5\cs202-stt\lab1_git\Any_Name :: $main = 75
```

Figure 3: Configuring local Git user details.

1.2.3 Adding and Committing Files

I created a new file named `README.md` and used the command `git status` to inspect the repository. The output indicated one untracked file. I then staged the file with `git add README.md` and confirmed the change with another `git status`, which now showed the file as staged.

```

PowerShell 7.5.2
~\Desktop\sem-5\cs202-stt\lab1_git\Any_Name :: | x 79 % | 08:29:53
→ echo "# STT Lab Assignment-1" > README.md | x 79 % | 08:30:22
~\Desktop\sem-5\cs202-stt\lab1_git\Any_Name :: | x 79 % | 08:30:22
→ ls
Directory: C:\Users\shardul\Desktop\sem-5\cs202-stt\lab1_git\Any_Name

Mode                LastWriteTime         Length Name
----                -
-a---             29-08-2025      08:30           24 README.md

~\Desktop\sem-5\cs202-stt\lab1_git\Any_Name :: | x 79 % | 08:30:25
→ git status
On branch main

No commits yet

Untracked files:
  (use "git add <file> ..." to include in what will be committed)
        README.md

nothing added to commit but untracked files present (use "git add" to track)
~\Desktop\sem-5\cs202-stt\lab1_git\Any_Name :: | x 79 % | 08:30:29
→ git add .\README.md | x 79 % | 08:30:39
~\Desktop\sem-5\cs202-stt\lab1_git\Any_Name :: | x 79 % | 08:30:39
→ git status
On branch main

No commits yet

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

~\Desktop\sem-5\cs202-stt\lab1_git\Any_Name :: | x 79 % | 08:30:42
→

```

Figure 4: Staging the README.md file for commit.

Next, I committed the staged file using the command `git commit -m "Initial commit: Added README.md"`. To verify, I executed `git log`, which displayed the commit details along with the message and metadata.

```

PowerShell 7.5.2
~\Desktop\sem-5\cs202-stt\lab1_git\Any_Name :: | x 79 % | 08:30:42
→ git commit -m "Initial commit: added README.md"
[main (root-commit) d8e3b8d] Initial commit: added README.md
1 file changed, 1 insertion(+)
create mode 100644 README.md
~\Desktop\sem-5\cs202-stt\lab1_git\Any_Name :: | x 79 % | 08:31:38
→ git log
commit d8e3b8d9b0d2174baed175eff5463bcd2c6ad43e (HEAD → main)
Author: Shardul Junagade <shardul.junagade@gmail.com>
Date:   Fri Aug 29 08:31:38 2025 +0530

    Initial commit: added README.md

~\Desktop\sem-5\cs202-stt\lab1_git\Any_Name :: | x 79 % | 08:31:45
→

```

Figure 5: Committing the file and viewing the commit history.

1.2.4 Working with Remote Repository

I created a remote repository on GitHub named Any_Name.

Repository Link: https://github.com/ShardulJunagade/Any_Name

Create a new repository

Repositories contain a project's files and version history. Have a project elsewhere? [Import a repository](#).
Required fields are marked with an asterisk (*).

1 General

Owner * ShardulJunagade / **Repository name *** Any_Name
✓ Any_Name is available.

Great repository names are short and memorable. How about **turbo-telegram**?

Description
0 / 350 characters

2 Configuration

Choose visibility * Public
Choose who can see and commit to this repository

Start with a template No template
Templates pre-configure your repository with files.

Add README Off
READMEs can be used as longer descriptions. [About READMEs](#)

Add .gitignore No .gitignore
.gitignore tells git which files not to track. [About ignoring files](#)

Add license No license
Licenses explain how others can use your code. [About licenses](#)

Create repository

Figure 6: Creating a new repository on GitHub.

After that, I linked the remote repository to my local repository using the command `git remote add origin <URL>`. Once the connection was established, I pushed my local commits to GitHub with `git push -u origin main`.

```
PowerShell 7.5.2
~\Desktop\sem-5\cs202-stt\lab1_git\Any_Name :: $main #
→ git remote add origin https://github.com/ShardulJunagade/Any_Name.git
~\Desktop\sem-5\cs202-stt\lab1_git\Any_Name :: $main #
→ git remote -v
origin https://github.com/ShardulJunagade/Any_Name.git (fetch)
origin https://github.com/ShardulJunagade/Any_Name.git (push)
~\Desktop\sem-5\cs202-stt\lab1_git\Any_Name :: $main #
→ git branch -M main
~\Desktop\sem-5\cs202-stt\lab1_git\Any_Name :: $main #
→ git push -u origin main
Enumerating objects: 3, done.
Counting objects: 100% (3/3), done.
Writing objects: 100% (3/3), 258 bytes | 86.00 KiB/s, done.
Total 3 (delta 0), reused 0 (delta 0), pack-reused 0 (from 0)
To https://github.com/ShardulJunagade/Any_Name.git
 * [new branch] main -> main
branch 'main' set up to track 'origin/main'.
~\Desktop\sem-5\cs202-stt\lab1_git\Any_Name :: $main #
→
```

Figure 7: Adding the remote and pushing local commits to GitHub.

To confirm that everything worked correctly, I opened the repository on GitHub in my browser and verified that all the changes were successfully reflected online. We can see that the README.md file is present.

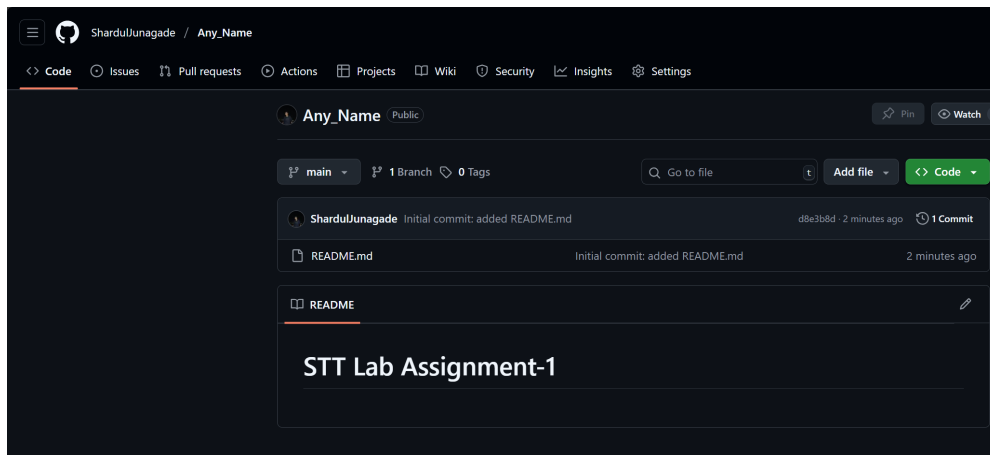


Figure 8: Verifying updates on GitHub.

For the next step, I cloned the repository using the command `git clone <URL>`, and we can see that the README.md file is present in the freshly cloned repo.

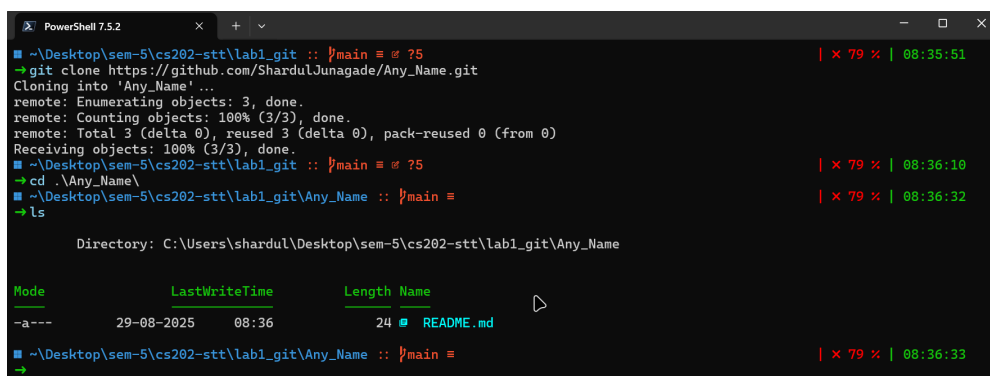


Figure 9: Cloning a repository from GitHub.

Then, I moved to GitHub in my web browser and added a new file named `main.py` that contains a simple Python program. I committed the changes directly on GitHub to the remote repository. This file was now visible in the remote repository on GitHub but not in my local repository.

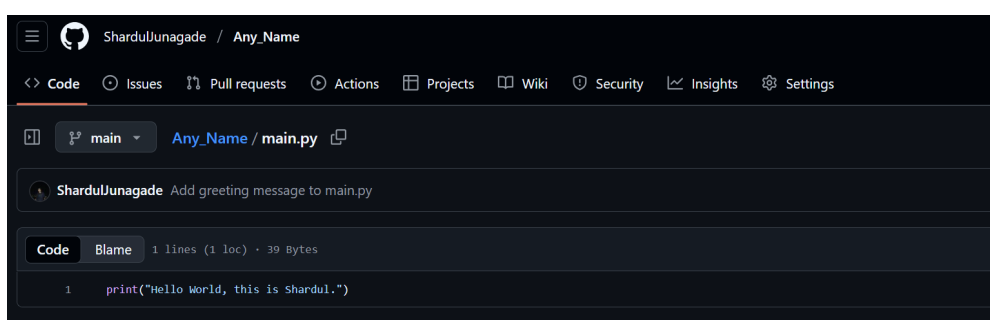


Figure 10: Adding a greeting message in main.py via GitHub.

Now, I had to pull these changes/commits from the remote repository to my local repository. To do this I executed the command `git pull origin main` to fetch and merge the changes. This pulled the new file into my local repository, keeping both versions in sync.

```

PowerShell 7.5.2
~\Desktop\sem-5\cs202-stt\lab1_git\Any_Name :: $main =
git pull
remote: Enumerating objects: 4, done.
remote: Counting objects: 100% (4/4), done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 3 (delta 0), reused 0 (delta 0), pack-reused 0 (from 0)
Unpacking objects: 100% (3/3), 980 bytes | 49.00 KiB/s, done.
From https://github.com/ShardulJunagade/Any_Name
d8e3b8d..9f2e282 main    -> origin/main
Updating d8e3b8d..9f2e282
Fast-forward
 main.py | 1 +
 1 file changed, 1 insertion(+)
 create mode 100644 main.py
~\Desktop\sem-5\cs202-stt\lab1_git\Any_Name :: $main =

```

Figure 11: Pulling the latest changes from GitHub.

1.2.5 GitHub Actions – Pylint Workflow

The next task was to automate code quality checks using Pylint.

For this task, I wrote a Python file called `app.py` that contained a simple calculator program, which you can see below:

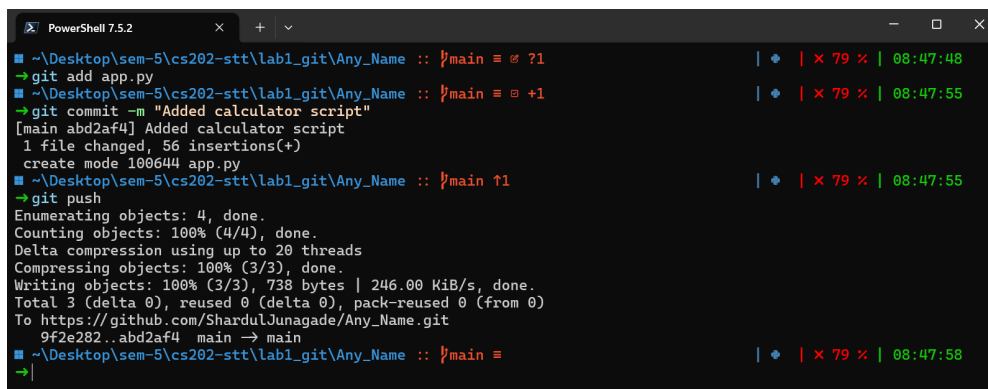
```

EXPLORER
ANY_NAME
  app.py
  main.py
  README.md
app.py
1  ## A simple calculator application
2
3  def add(x, y):
4      return x + y
5
6  def subtract(x, y):
7      return x - y
8
9  def multiply(x, y):
10     return x * y
11
12 def divide(x, y):
13     if y == 0:
14         return "Error: Division by zero"
15     return x / y
16
17 def modulus(x, y):
18     return x % y
19
20 def power(x, y):
21     return x ** y
22
23 def calculator():
24     print("Simple Calculator")
25     print("Select operation:")
26     print("1. Add")
27     print("2. Subtract")
28     print("3. Multiply")
29     print("4. Divide")
30     print("5. Modulus")
31     print("6. Power")
32
33     choice = input("Enter choice (1-6): ")
34
35     if choice not in ['1', '2', '3', '4', '5', '6']:
36         print("Invalid input")
37         return
38
39     num1 = float(input("Enter first number: "))
40     num2 = float(input("Enter second number: "))
41
42     if choice == '1':
43         print("Result:", add(num1, num2))
44     elif choice == '2':
45         print("Result:", subtract(num1, num2))
46     elif choice == '3':
47         print("Result:", multiply(num1, num2))
48     elif choice == '4':
49         print("Result:", divide(num1, num2))
50     elif choice == '5':
51         print("Result:", modulus(num1, num2))
52     elif choice == '6':
53         print("Result:", power(num1, num2))
54
55 if __name__ == "__main__":
56     calculator()
57

```

Figure 12: Initial version of `app.py` (calculator code).

After saving the file, I staged, committed, and pushed it to GitHub.



```
PowerShell 7.5.2
~\Desktop\sem-5\cs202-stt\lab1_git\Any_Name :: $main = ?1
git add app.py
~\Desktop\sem-5\cs202-stt\lab1_git\Any_Name :: $main = +1
git commit -m "Added calculator script"
[main abd2af4] Added calculator script
1 file changed, 56 insertions(+)
create mode 100644 app.py
~\Desktop\sem-5\cs202-stt\lab1_git\Any_Name :: $main ↑1
git push
Enumerating objects: 4, done.
Counting objects: 100% (4/4), done.
Delta compression using up to 20 threads
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 738 bytes | 246.00 KiB/s, done.
Total 3 (delta 0), reused 0 (delta 0), pack-reused 0 (from 0)
To https://github.com/ShardulJunagade/Any_Name.git
9f2e282..abd2af4 main -> main
~\Desktop\sem-5\cs202-stt\lab1_git\Any_Name :: $main =
```

Figure 13: Pushing the calculator code to GitHub.

On GitHub, there was a section of suggested workflows for GitHub Actions. Here, I selected the Pylint workflow and clicked on **Configure**.

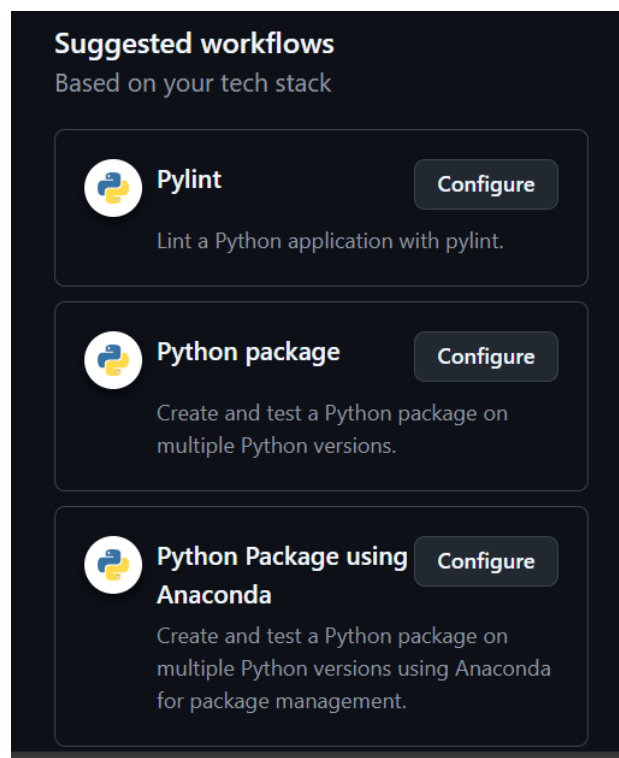
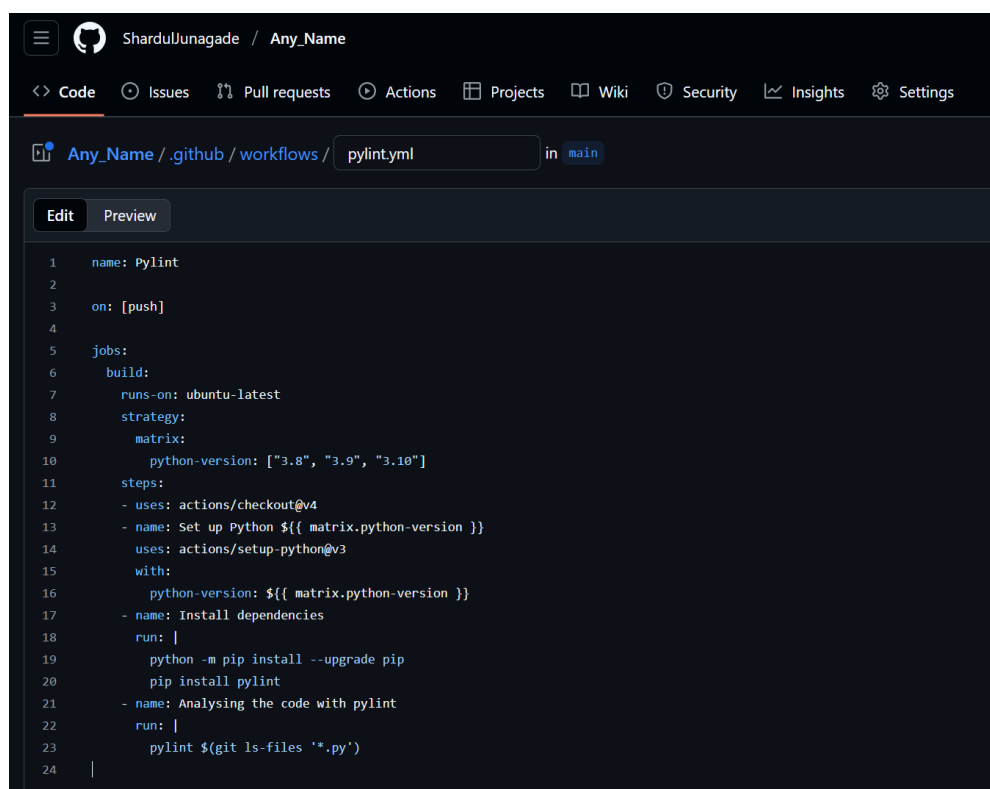


Figure 14: Suggested workflows for GitHub Actions.

Then, I created the `.github/workflows/Pylint.yml` file to enable linting on every push, as shown below:



The screenshot shows the GitHub Actions workflow editor for a file named `pylint.yml` in the `main` branch. The workflow is configured with the following steps:

```

1  name: Pylint
2
3  on: [push]
4
5  jobs:
6    build:
7      runs-on: ubuntu-latest
8      strategy:
9        matrix:
10         python-version: ["3.8", "3.9", "3.10"]
11      steps:
12        - uses: actions/checkout@v4
13        - name: Set up Python ${ matrix.python-version }
14          uses: actions/setup-python@v3
15          with:
16            python-version: ${ matrix.python-version }
17        - name: Install dependencies
18          run: |
19            python -m pip install --upgrade pip
20            pip install pylint
21        - name: Analysing the code with pylint
22          run: |
23            pylint $(git ls-files '*.py')
24

```

Figure 15: Creating the Pylint.yml workflow file.

As soon as I committed the workflow file, GitHub Actions automatically triggered the workflow. The first run failed because Pylint reported several errors in my code, which can be seen below:

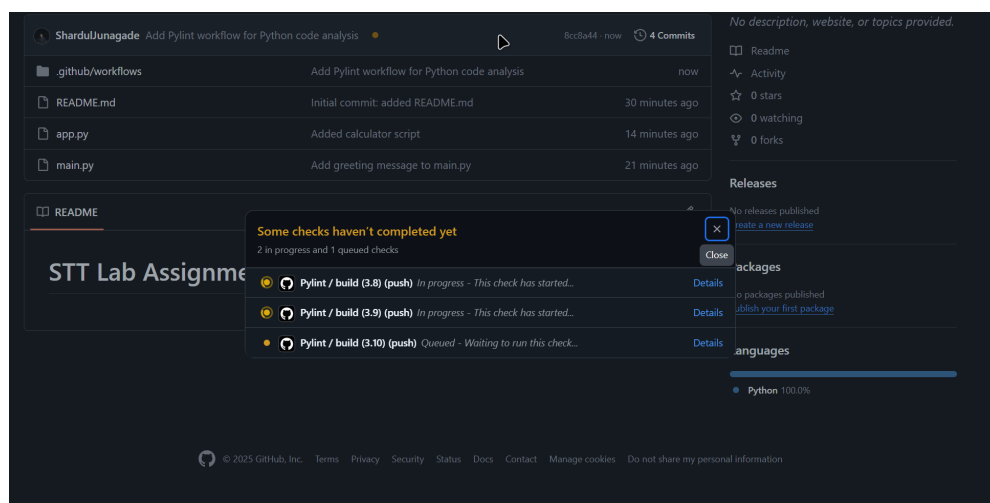


Figure 16: GitHub Actions running Pylint workflow.

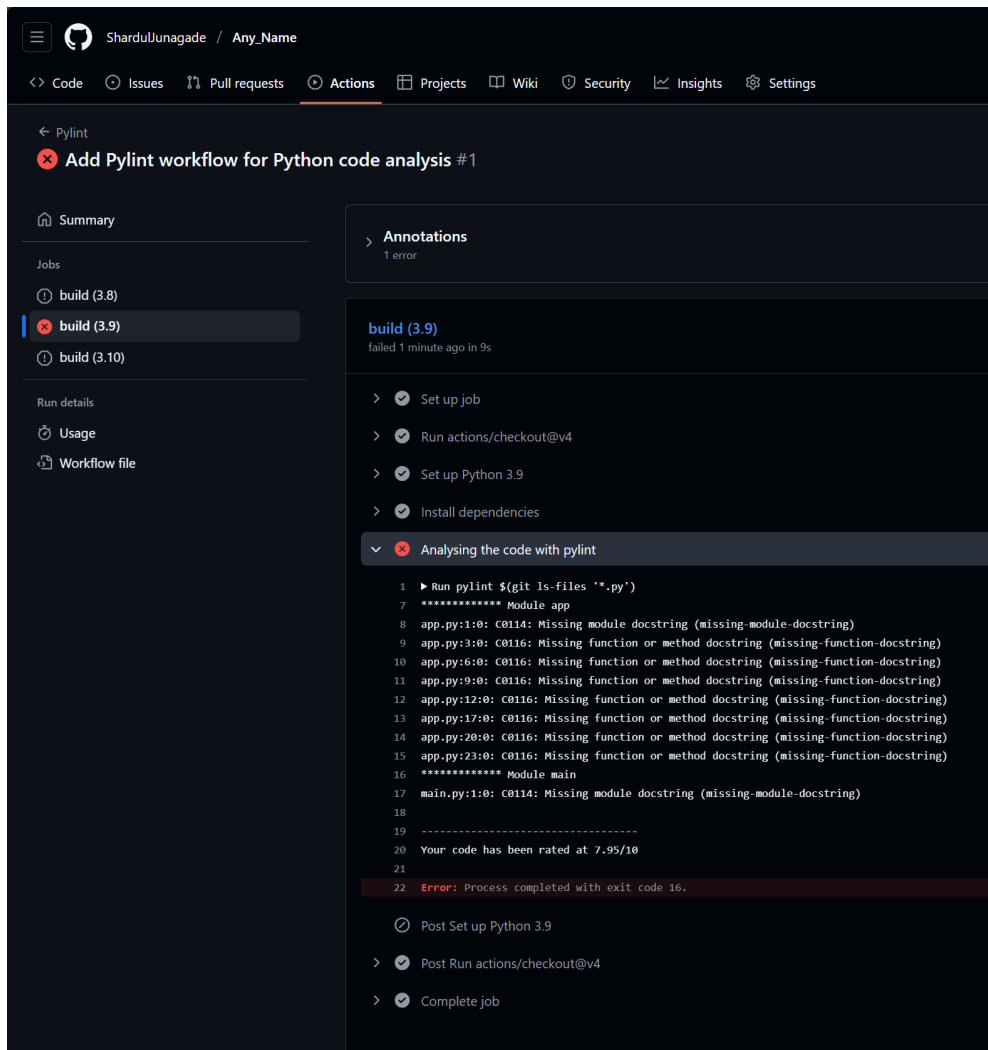


Figure 17: Initial Pylint errors detected.

After reviewing the error messages, I fixed the issues by adding proper docstrings and correcting other bugs. I committed and pushed the corrected code to the repository.

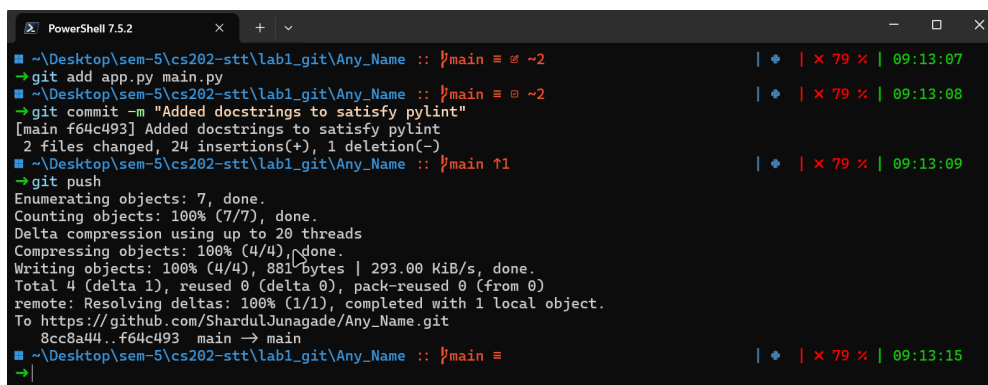


Figure 19: Pushing the corrected code.

This time, the workflow passed successfully, and GitHub showed a green tick, which confirmed that my code met the required standards.

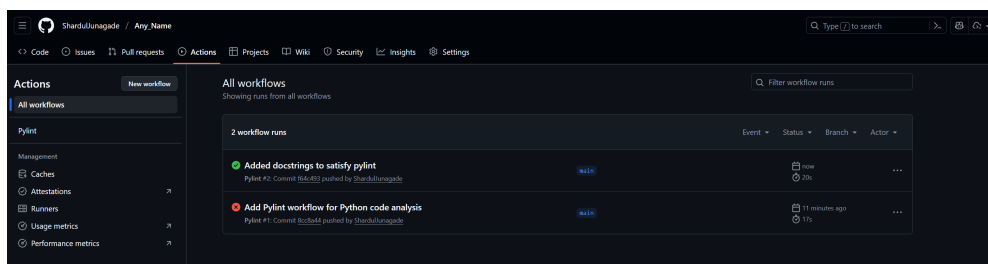


Figure 20: Successful Pylint workflow run.

All three workflow runs completed successfully, as indicated by the green tick on each build in the GitHub Actions tab.

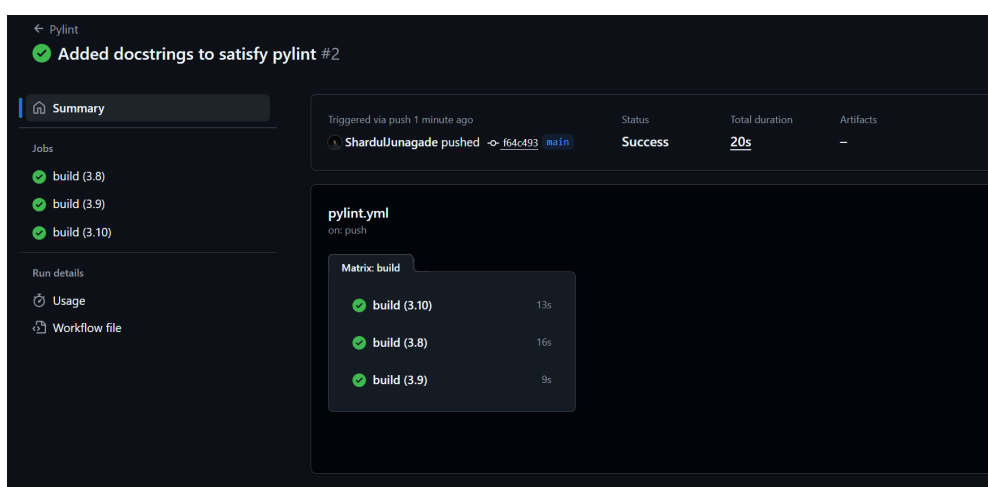


Figure 21: Workflow completion with a green tick.

1.3 Results and Analysis

Through this lab, I successfully created and managed a Git repository, synchronized changes with GitHub, and demonstrated cloning and pulling operations. The GitHub Actions workflow validated my Python code using Pylint, ensuring it followed coding standards and confirming the results with a green tick on GitHub.

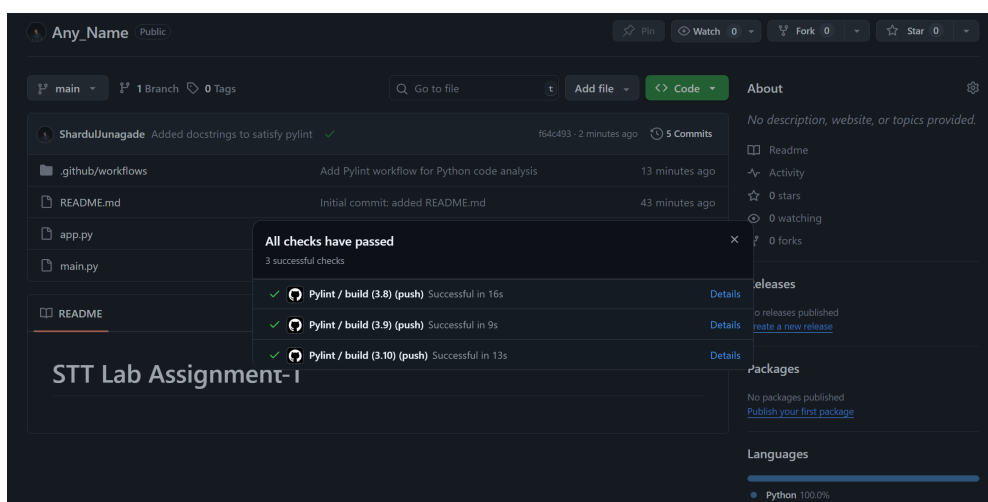


Figure 22: Successful Pylint validation and workflow status.

The final commit history below shows the whole process – from the first commit to fixing errors and reaching the successful workflow run.

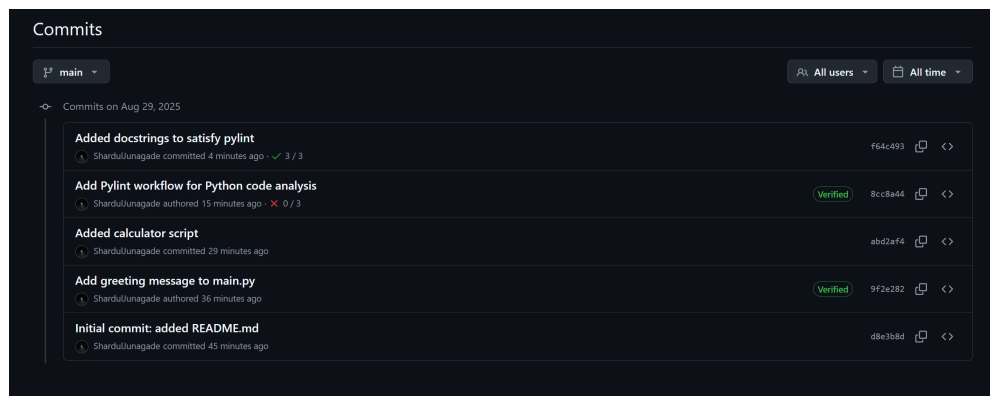


Figure 23: Final commit history showing all changes.

Task	Outcome
Initialize repository	Repository created successfully (Figure 2-3)
Push to GitHub	Changes pushed and visible online (Figures 4–8)
Clone and Pull	Repository cloned and synced correctly (Figures 9–11)
GitHub Actions (Pylint)	Failed first due to docstring issues, later passed with green tick (Figures 12–21)

1.4 Discussion and Conclusion

This lab gave me hands-on practice with Git, starting from creating an empty repository to adding files, staging, committing, pushing to GitHub, cloning repositories, and pulling updates. One of the main challenges I faced was the branch mismatch issue (main vs. master), which caused errors during my first push.

I also found interpreting the Pylint workflow's yaml file slightly tricky at first and even though the Python code was running fine, the workflow still failed due to docstring and formatting issues which were caught by Pylint. Going through the error messages and fixing them was a valuable learning experience. The most satisfying moment was finally seeing the green tick on GitHub after the corrected code passed Pylint.

Overall, this lab gave me a nice overview of how Git and GitHub work together and showed me how CI/CD pipelines can give instant feedback on code quality, something that is very useful in real-world projects where many developers work together.

1.5 References (1 Mark for Format + Style)

- Git Documentation
- GitHub Guides
- Git Cheat Sheet
- Pylint Documentation
- Lab Document (Shared on Google Classroom)