Assignment_2

DHRUV GAL - 35425822

WELCOME

This is a demonstration file for Assignment 2.

Git Branch Demo

This section will change depending on the branch

Step 2: Initialise Git and Push to GitHub

After setting up the RStudio project and writing this `example.qmd` file, I used the terminal to initialise version control using Git and connect it to a GitHub repository.

Why this step matters:

Initialising Git allows version tracking of all changes made in the project. Connecting to GitHub enables cloud backup and collaboration.

Bash Commands Used:

```
# Initialise git repository
git init
```

Stage all current files

```
git add .
```

Make the first commit

```
git commit -m "Initial commit: setup project and added example.qmd"
```

Add remote GitHub repository

```
git remote add origin git@github.com:DhruvGal/Git_Project_Guide-.git
```

Rename branch to 'main'

```
git branch -M main
```

Push changes to GitHub and set upstream

```
git push -u origin main
```

This line is added in the main branch

Step 3: Create a New Branch and Make Changes

To simulate collaborative development, I created a new branch called `testbranch`, made a change to the `example.qmd` file, committed the update, and pushed the branch to GitHub.

Why this step matters:

Using branches allows developers to experiment or work on features in isolation without affecting the main project. It supports safer collaboration and better version tracking.

Bash Commands Used:

```
# Create and switch to a new branch
git checkout -b testbranch
```

Edit assingment_2.qmd (e.g. adding a new line from another branch)

After saving the file, stage and commit the changes

```
git add example.qmd
git commit -m "Added new Git section in testbranch"
```

Push the new branch to GitHub

```
git push -u origin testbranch
```

This is a line added in the test branch

Step 4: Add Data Folder and Amend Previous Commit

To track the dataset used in Assignment 1, I created a `data/` folder and included relevant files. Instead of making a new commit, I amended the previous one to keep the commit history clean.

Why this step matters:

Amending commits is useful for minor corrections to the most recent commit — such as adding forgotten files — without cluttering the history.

Actions Taken:

1. Created a folder called `data/` in the RStudio project directory.
2. Copied the Assignment 1 dataset(s) into the `data/` folder (e.g., `dataset.csv`).
3. Amended the previous commit to include this new folder.

Bash Commands Used:

Create the data folder and add files manually in RStudio

Stage the new folder

```
git add data/
```

Amend the most recent commit to include the data

```
git commit --amend --no-edit
```

Push the amended commit (force is required)

```
git push --force
```

Step 5: Modify `main` to Cause a Conflict with `testbranch`

After pushing changes from `testbranch`, I switched back to the `main` branch and made a conflicting edit to the same section of `example.qmd`. This ensures a conflict when merging later.

Why this step matters:

Conflicts naturally occur in collaborative workflows. This step demonstrates how to simulate and later resolve a conflict — a key Git skill.

Bash Commands Used:

```
# Switch back to main branch
git checkout main
```

Edit example.qmd in the same section modified in testbranch

(e.g.editing the same line on the main branch)

Save the file, then stage and commit

```
git add example.qmd
git commit -m "Conflicting edit in example.qmd on main"
```

Push the change to remote

```
git push
```

Step 6: Merge `testbranch` into `main` and Resolve Conflict

Now that both `main` and `testbranch` have conflicting changes to the same part of `example.qmd`, I attempted to merge `testbranch` into `main`. As expected, Git detected a conflict.

Why this step matters:

Conflicts are a natural part of collaborative development. Resolving them cleanly is crucial to maintaining a functioning codebase.

Bash Commands Used:

```
# Ensure we are on main
git checkout main
```

```
# Pull latest changes to main (if needed)
git pull
```

```
# Merge testbranch into main|
git merge testbranch
```

Step 7: Tag the Final Commit as v1.0

After resolving the merge conflict and committing the result, I created an annotated tag named `v1.0` to mark this as the first stable version of the project.

Why this step matters:

Tags are used to capture important points in a project's history (e.g., releases). Annotated tags include metadata like tagger name, date, and a message.

Bash Commands Used:

```
# Create an annotated tag
git tag -a v1.0 -m "Version 1.0 after resolving merge conflict"
```

```
# Push the tag to GitHub
git push origin v1.0
```

Step 8: Delete `testbranch` Locally and Remotely

Now that `testbranch` has been successfully merged into `main`, I deleted the branch both from my local system and from the GitHub remote. This keeps the repository clean and avoids confusion.

Why this step matters:

After merging, feature branches are no longer needed. Removing them avoids clutter and reduces the chance of accidental edits to outdated branches.

Bash Commands Used:

```
# Delete testbranch locally
git branch -d testbranch
```

```
# Delete testbranch on GitHub
git push origin --delete testbranch
```

Step 9: Show Condensed Git Commit Log

To view the project's Git history in a compact and readable format, I used the `--oneline` flag with `git log`. This shows each commit as a single line with its hash and message.

Why this step matters:

A condensed commit log provides a clear summary of the project's development. It also helps validate that the correct changes were made in a structured sequence.

Bash Command Used:

```
git log --oneline
```

Step 10: Add a Plot and Undo the Commit (Keep Changes)

To demonstrate how to undo a commit while retaining changes locally, I added a simple plot to the `example.qmd` file, committed the update, and then used a Git reset to roll back the commit without discarding the work.

Why this step matters:

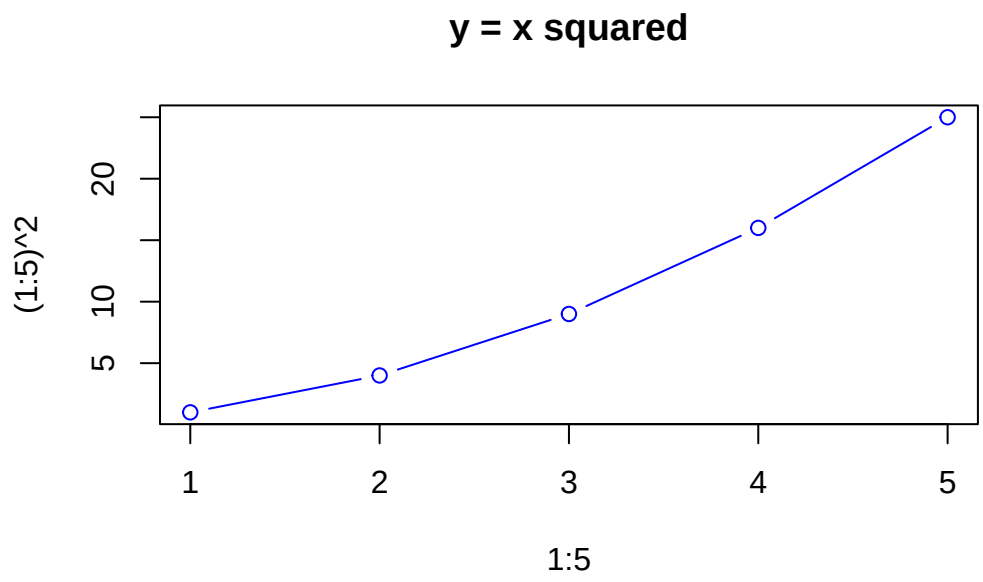
This workflow is useful when you've made a premature commit but still want to keep your edits for reworking or recommitting later. `git reset --soft` moves the HEAD pointer without touching your working directory.

What I added in `example.qmd`:

````
## Simple R Plot

::: {.cell}

```{.r .cell-code}
plot(1:5, (1:5)^2, type = "b", col = "blue", main = "y = x squared")
````
````


y = x squared
(1:5)^2
5
10
20
1
2
3
4
5
1:5

:::