# 1. What is Git?

Git is a **version control system** that tracks changes you make to your project files over time. Think of it like a **'history buffer’** you can jump back to any earlier point.

# 2. Why use Git for data science?

Git helps you:

* **Keep track** of every change (even small tweaks in code or analysis).
* **Revert mistakes** easily.
* Collaborate with others **without overwriting** each other’s work.

# 3. Key Concepts

**Repository** (repo): The folder you’re working in, now under Git’s watch. Created with `**git init**` or **cloned** from GitHub.

**Commit**: A snapshot of your code at a point in time. Created with `**git add**` and `**git commit -m "message"**`

**Branch**: A parallel version of your project. Created with **`git checkout -b branch-name**`

**Merge**: Combine changes from one branch into another. May cause conflicts if overlapping edits exist. Sometimes, if two people changed the same part, Git may ask you to resolve a conflict, and you must manually choose or combine the changes, then save and commit.

**Remote**: A copy of your project on GitHub. Use `**git remote add**`, `**git push**`, and `**git pull**`

# 4. Initialization

**Case 1: Create a Repository from Scratch (git init)**

1. Open Terminal or Command Prompt

2. Go to the folder you want to work in:

- **cd Folder path**

3. Create a new folder for your project:

- **mkdir Folder name**

- **cd Folder name**

or

- **mkdir Folder name && cd Folder name**

4. Initialize the Git repo:

- **git init**

Now you have a local repository. Git has created a hidden folder named **`.git`.**

5. Add a file and make your first commit:

When you initialize a new Git repository, it's standard practice to add a **README.md** file that tells **what the project is about**. It's like taping a **label** to a physical folder so others know what’s inside without opening every file.

* To add README file with description:  
   **echo "Description" > README.md**
* To addthe Readme file into the repository:  
   **git add README.md**
* To save the changes into Git with a **message:**

(The message describes what you did)  
 **git commit -m "Message"**

**Case 2: Clone an Existing Repository from GitHub (git clone)**

Let’s say your teammate shared a GitHub repo link with you.

1. Copy the GitHub URL:  
 **Example**: https://github.com/<username>/<repository-name>.git

2. Open Terminal / Command Prompt and go to your workspace folder:

**cd Folder path**

3. Clone the repository:  
 **git clone https://github.com/<username>/<repository-name>.git**

This will:  
 - Download the project folder`  
 - Set up the `.git` folder inside  
 - Connect to GitHub so you can pull/push changes

4. Go inside the cloned folder:  
 **cd** **Folder name**

# 5. Git Push and Pull Commands

**Git push**

The `git push` command is used to **upload your local commits** to a remote repository (like GitHub).  
  
- **Format**:  
git push origin <branch-name>  
  
- **Example**:  
git push origin main  
  
- **This means:** Send my latest changes from my local main branch to the GitHub remote repo.

**Git pull**

The `git pull` command is used to **download the latest changes** from the remote GitHub repository to your local machine.  
  
- **Format**:  
git pull origin <branch-name>  
  
- **Example**:  
git pull origin main  
  
- **This means** Fetch and merge changes from GitHub's main branch into my local main branch.

# 6. Things to remember

**Before git pull**

1**.** Make sure your local work is committed:  
 - Run `**git status**` to check for uncommitted changes.  
 - **Commit or stash** them before pulling.

**Example**:

A screen shot of a computer program

AI-generated content may be incorrect.

**This means:**

* You edited analysis.py but haven’t added it.
* You created a new file new\_plot.png that Git doesn’t know about yet.
* **To** **commit**:
  + - git add . # Stages all changes
    - git commit -m "Updated analysis and added plot"
* **To stash:**

Sometimes you’re not ready to commit, maybe the changes are incomplete. You can "stash" them.

* + - git stash

2. Check which branch you're on:  
 - Use `**git branch**` to verify you're on the correct one.

3. Pull before you push:  
 - Run `**git pull origin main**` to ensure you're working with the latest code.

**Before git push**

1. Check your changes:  
 - Use `**git status**` to see what will be pushed.

2. Make sure your branch is up to date:  
 - Run `**git pull origin main**` to avoid conflicts.

3. Make your changes inside this folder  
 For example, add or edit files like script.py, README.md, etc.

4. Stage your changes

* git add . *(or specify files like git add file.py)*

5. Write meaningful commit messages:  
 - **Example**: `git commit -m "Fix: Updated chart logic"`

6. Push to the correct branch:  
 - Double-check with `**git branch**` before pushing.

# 7. Workflow for Data Science Projects

**Step by step:**

1. Create and initialize your repo (or clone an existing one).
2. Work in branches (Create a new path to try your changes safely.).
3. Commit your work frequently with clear messages.
4. Push your progress to GitHub.
5. Create a pull request to merge your branch into main.
6. After merging, delete the feature branch that you created.

# 8. Collaboration Made Simple

* Everyone works in branches, not directly on main.
* Pull requests allow review before merging.
* GitHub tracks who changed what, when, and why.

# 9. Bonus Tips

**VIEWING CHANGE HISTORY**

You can always see past work with:  
- **git log** # Full detailed history  
- **git log --oneline** # Short summary view

**USING STATUS & DIFF**

- **git status** # See what’s modified, staged, or untracked.  
- **git diff** # View exactly what changed before you commit.

**OPTIONAL: GRAPHICAL TOOLS FOR GIT**

Don’t you like command line? No problem! You can use:  
- **GitHub Desktop**: Easy visual Git tool.  
- **VS Code Git Panel**: Click buttons to commit, push, pull, and more.

**HOW TO RESOLVE MERGE CONFLICTS (IN SIMPLE TERMS)**

If Git shows a **conflict** **during merging**:  
1. Open the conflicted file — Git will mark conflicting parts with <<< >>>.  
2. Choose the correct version, delete the markers, and save.  
3. Run:  
 **git add conflicted\_file.py** **git commit**  
Done! Conflict resolved.

**WHAT IS A PULL REQUEST?**

A **Pull Request (PR)** is how you ask to merge your changes into the main project. It allows teammates to review your code before it becomes part of the main branch. You create PR on GitHub after you’ve pushed your branch.

**HOW TO CREATE A PULL REQUEST (STEP-BY-STEP):**

**Step 1: Push Your Branch to GitHub**Make sure your changes are committed, then push:  
 **git push origin your-branch-name**  
**Step 2: Go to GitHub**- Open your repository on GitHub.  
- You'll see a message like: **'Compare & pull request'**.  
- Click that, **or** go to the 'Pull Requests' tab → **'New Pull Request'**.  
  
**Step 3: Choose Branches**  
- **Base**: usually 'main' (where you want your changes to go).  
- **Compare**: your feature branch (where your changes are).  
  
**Step 4: Add Title and Description**  
- **Title**: What did you change?  
- **Description**: Why did you make the change? Any details for review.

**Example**:  
Title: Add data cleaning script  
Description: Added script to remove nulls and filter bad data.  
  
**Step 5: Submit**- Click **'Create Pull Request'**.  
- Teammates can now **review, comment, or approve** your changes.  
- **Once approved,** click **'Merge'** to add it to the main branch.

**HOW TO CREATE AND USE BRANCHES**

Branches let you work on different versions of your project without affecting the main version.  
  
**Step 1: Create a New Branch**  
Use this command to create a branch and switch to it:  
 **git checkout -b your-branch-name**

**Step 2: Do Your Work**  
- Make changes in your files (edit, add, delete).  
- Save and commit your work regularly.  
  
**Step 3: Switch Between Branches**  
To move to another branch (like main):  
 **git checkout main**  
  
**Step 4: Push the Branch to GitHub**  
Once you're ready to share your branch:  
 **git push origin your-branch-name**  
**Tip**:  
- Use `**git branch**` to see all branches.  
- To delete a branch

* **git branch -d <branch-name>** # delete locally
* **git push origin --delete <branch-name>** # delete from GitHub (if needed)

# 10. Bonus Tips for Data Scientists

* **Use `.gitignore`** to exclude big files (data/models).

1. **Locate or create the .gitignore file** in your project’s root directory.
2. **Open it** using any text editor (e.g., VS Code, Notepad, etc.).
3. **Add the following lines** to ignore CSV files, the model’s folder, and any environment files:
   * + \*.csv
     + models/
     + .env
4. Save & commit the changes:

**git commit -m "Updated .gitignore to exclude large files"**

1. Check if any of these files were already tracked by Git. If so, you’ll need to **untrack** them:

* git rm --cached \*.csv
* git rm --cached -r models/
* git rm --cached .env

Keeps your project clean and faster to work with.

* **Set up SSH** keys to avoid re-typing credentials.

SSH keys let you **connect securely without typing credentials every time**. Safer and more convenient than HTTPS login.

* **Generate** SSH key:  
   ssh-keygen -t ed25519 -C [your-email@example.com](mailto:your-email@example.com)
* **Copy** the key and **add** it to GitHub (Settings > SSH and GPG Keys).

- **Clone** repos using SSH for secure, hassle-free authentication:  
 [git@github.com:username/repo-name.git](mailto:git@github.com:username/repo-name.git)

* **Use** **Git** **Large File Storage** (Git LFS) if you need to version large files  
  - It helps keep your repo clean and efficient.

**Step-by-Step with example:**  
1. Install Git LFS:  
 git lfs install

2. Tell Git LFS **which file types** to track:  
 git lfs track "\*.csv"  
 git lfs track "\*.pkl"  
  
3. Add and commit files as usual:

Git LFS automatically creates or updates a file called **.gitattributes** in your project.  
This file **tells Git which file types should be handled by Git LFS**.

* git add .gitattributes
* git add filename
* git commit -m "Track large data file with Git LFS"
* git push origin main

**Tip:**

You can use:

* git add . → adds **all** changed files in the folder
* git add \*.py → adds **all Python files**
* git add foldername/ → adds **everything in that folder**
* git add \*.csv → Adds **all CSV files**
* For Jupyter Notebook users, try `**nbdime**` to get better diff and merge views specifically for **.ipynb files.**

**Step-by-Step with example:**  
1. Install nbdime:  
 pip install nbdime  
  
2. Set it up with Git:  
 nbdime config-git --enable  
  
3. View notebook differences:  
 nbdiff-web old\_notebook.ipynb new\_notebook.ipynb  
  
4. Resolve merge conflicts visually:  
 nbdime merge-web conflicted\_notebook.ipynb

**What this command does:**

It opens an **easy-to-use graphical interface** in your browser where:

* You see **your version**, **their version**, and a **base version**
* You can choose which changes to keep
* You click "Save" to finish merging

# 11. PRACTICE FLOW (TRY Yourself)

**Practice these steps to get comfortable with Git:**1. Create a Git repo (git init)  
2. Add and commit a file  
3. Create a branch and make changes  
4. Commit those changes  
5. Merge the branch back to main  
6. Push your work to GitHub  
7. Create a Pull Request  
8. Try creating and resolving a merge conflict

# 12. Summary

Git is like saving multiple versions of your project—like a game with save points.  
GitHub is the cloud where you store those saves and optionally share them with teammates.  
Working with branches is like testing ideas in new worlds before merging them back into the main universe.