Git Introduction

Git Foundations

- **The Problem:** Saving files as `script_final.py`, `script_final_v2.py`, or `script_really_final.py` is messy, error-prone, and impossible to track cleanly.
- **The Solution—Git:** A version control system that tracks history, acts as a time machine, and records exact snapshots of your project.
- **The Goal:** Master the core local workflow: initializing, staging, committing, and inspecting history.

What is Version Control?

- **Not Just Backups:** Git doesn't just record file differences line by line; it takes a compressed **snapshot** of your entire project workspace whenever you tell it to.
- **The Working Directory:** Where you write and edit your files.
- **The Local Repository (.git folder):** Git's hidden database where your snapshots and history live safely.

Starting a Repository

- To turn any ordinary folder on your computer into a Git-tracked project:

```
cd my_project_folder
git init
```

- **What this does:** Creates a hidden `.git` subdirectory that begins tracking your files.
- *Note:* You only ever need to run `git init` **once** per project.

Checking Your Status

- Your primary command-line dashboard. Use it constantly to see what Git is doing:

  ```
  git status
  ```

- **What it tells you:**
  - Which branch you are on.
  - Which files have been modified (shown in red).
  - Which files are ready to be saved (shown in green).

Staging Changes

- Git doesn't automatically save every file you change. You must choose *what* goes into the next snapshot.
- To stage a specific file:

  ```
  git add order_processor.py
  ```

- To stage **all** modified files in your directory:

  ```
  git add .
  ```

- *Why stage?* It lets you curate a clean, intentional bundle of changes before saving.

Saving a Snapshot

- Once your files are staged, you permanently record the snapshot into your project's history with a message:

  ```
  git commit -m "Implement initial order processing logic
  ```

- **The `-m` flag:** Allows you to type an inline commit message explaining *what* changed and *why*.
- **Golden Rule:** Write messages in the imperative mood (*"Add login validation"*, not *"Added login validation"*).

The Core Local Workflow Loop

1. **Code:** Write or modify code in your files.
2. **Status:** Run `git status` to see what changed.
3. **Add:** Run `git add <filename>` to stage your files.
4. **Commit:** Run `git commit -m "Message"` to save the snapshot.
5. *Repeat!*

Viewing Your Project History

- To look back in time and see all the commits you've made:

  ```
  git log
  ```

- **What it displays:**
  - The unique Commit Hash (e.g., `a1b2c3d`).
  - The author and timestamp.
  - Your commit message.
- *Tip:* Press `q` on your keyboard to exit the log view if it fills up your terminal screen.

A Cleaner View of History

- When your project grows, standard `git log` can take up too much screen space.
- To see a compact, single-line summary of your history:

  ```
  git log --oneline
  ```

- This gives you a clean list of recent commits and their short hashes at a glance.

Command Cheat Sheet

- `git init` — Turn a folder into a Git repository (run once)
- `git status` — Check the current state of your files and staging area
- `git add <file>` or `git add .` — Stage changes for the next commit
- `git commit -m "msg"` — Permanently save staged changes with a message
- `git log` — View full project commit history
- `git log --oneline` — View a compact summary of project history