Git + Python Projects

Learn How to Version, Track, and Share Real Python Projects

Writing Python code is only one part of becoming a real developer.

A real Python project also needs to be:

- organized
- version controlled
- recoverable
- documented
- shareable
- easy for others to run
- easy to improve over time

This is where **Git and Python projects** come together.

Git tracks how your project changes over time. Python provides the code and project structure.

Together, they create a workflow you can use for everything from a small CLI script to a production application.

---

1. Why Git Matters in Python Projects

Imagine you build a Python application today.

Tomorrow you change several files.

The application stops working.

You remember that it worked yesterday, but you do not remember exactly what changed.

Without version control, you may have to manually search through your files and guess what went wrong.

With Git, you can inspect your history and identify what changed.

Git gives you a way to:

- track changes
- create checkpoints
- compare versions
- undo mistakes
- work on separate features
- collaborate with others
- share your project through platforms such as GitHub

- understand how a project evolved

Think of Git as a **history system for your codebase**.

---

2. Git vs GitHub

These are related but different.

Git

Git is a version control system that runs on your computer.

It tracks changes in your project.

GitHub

GitHub is a platform for hosting Git repositories online.

It can provide:

- remote repositories
- collaboration
- pull requests
- issue tracking
- code review
- project documentation
- public portfolios

You can use Git without GitHub.

You can also use Git with other hosting platforms.

A simple mental model:

```
Python
   ↓
Your project
   ↓
Git
   ↓
Version history
   ↓
GitHub
   ↓
Online repository
```

---

3. What Is a Git Repository?

A Git repository is a project whose history is tracked by Git.

Suppose your Python project looks like this:

```
expense_tracker/
│
├── app.py
├── expenses.py
├── README.md
└── requirements.txt
```

After initializing Git:

```
git init
```

Git creates a hidden `.git` directory.

```
expense_tracker/
│
├── .git/
├── app.py
├── expenses.py
├── README.md
└── requirements.txt
```

The `.git` directory contains the information Git needs to track the repository.

Do not manually modify it.

---

4. Create a Python Project Before Creating Git History

A useful workflow starts with a project directory.

```
mkdir expense_tracker
cd expense_tracker
```

Create your Python file:

```
touch app.py
```

On Windows PowerShell, you can use:

```
New-Item app.py
```

Then initialize Git:

```
git init
```

Now the directory is both:

- a Python project
- a Git repository

---

5. Check the Repository

Use:

```
git status
```

Git may show:

```
Untracked files:
    app.py
```

This means Git sees the file but is not currently tracking it in a commit.

---

6. The Git Workflow

The most important Git workflow is:

```
Edit
 ↓
Status
 ↓
Add
 ↓
Commit
 ↓
Push
```

For example:

```
git status
git add .
git commit -m "Add expense tracker"
git push
```

Each step has a different purpose.

---

7. Working Directory

The working directory contains the files you are currently editing.

For example:

```
app.py
expenses.py
README.md
```

When you change `app.py`, Git notices that the file is different from the last committed version.

---

8. Staging Area

Git has a staging area.

You can add a specific file:

```
git add app.py
```

Or multiple files:

```
git add app.py expenses.py
```

Or all changes:

```
git add .
```

The staging area lets you decide which changes should become part of the next commit.

Think of it as preparing a package before shipping it.

```
Working Directory
        ↓
   git add
        ↓
Staging Area
        ↓
  git commit
        ↓
Repository History
```

---

9. Commits

A commit is a saved point in your project's history.

Example:

```
git commit -m "Add expense creation"
```

A good commit message explains what changed.

Good:

```
Add expense creation
```

```
Validate expense amount
```

```
Add CSV export
```

Less useful:

```
update
```

```
changes
```

```
fix stuff
```

A commit should help your future self understand the project.

---

10. Small Commits Are Easier to Understand

Suppose you implement three unrelated features:

```
Add login
Add CSV export
Fix dashboard bug
```

Putting everything into one commit makes the history harder to understand.

Instead:

```
git commit -m "Add login"
git commit -m "Add CSV export"
git commit -m "Fix dashboard bug"
```

Each commit represents a meaningful change.

---

11. Viewing Git History

Use:

```
git log
```

You may see:

```
commit a81f...
Author: Hafsa
Date: ...

    Add CSV export
```

```
commit 72bc...
Author: Hafsa
Date: ...

    Add expense validation
```

For a shorter view:

```
git log --oneline
```

Example:

```
a81f2d1 Add CSV export
72bc9c0 Add expense validation
4fd3a21 Create expense model
```

This is useful for quickly understanding a project's history.

---

12. Inspecting Changes

Before committing, use:

```
git diff
```

It shows changes that have not been staged.

For staged changes:

```
git diff --staged
```

This is a useful habit:

```
Change code
   ↓
git diff
   ↓
Review
   ↓
git add
   ↓
git diff --staged
   ↓
Review again
   ↓
git commit
```

---

13. `.gitignore`

Python projects generate files that usually should not be committed.

Examples:

```
__pycache__/
*.pyc
.venv/
.env
.pytest_cache/
.coverage
```

Create:

```
.gitignore
```

Example:

```
__pycache__/
*.py[cod]

.venv/
venv/

.env

.pytest_cache/
.coverage
htmlcov/

.vscode/
.idea/
```

The purpose of `.gitignore` is to tell Git which files it should ignore.

---

14. Never Commit Secrets

A common mistake is committing:

```
.env
```

containing:

```
API_KEY=secret-value
DATABASE_PASSWORD=secret-value
```

Do not commit credentials.

Instead:

```
.env
```

should usually be ignored.

Create a safe example file:

```
.env.example
```

Example:

```
API_KEY=
DATABASE_URL=
```

This tells other developers which environment variables they need without exposing your actual secrets.

---

15. Python Virtual Environments and Git

Your virtual environment should normally not be committed.

For example:

```
.venv/
```

should be in `.gitignore`.

Why?

Because the virtual environment contains installed packages and environment-specific files.

Instead of committing the environment itself, commit the dependency information.

For example:

```
requirements.txt
```

Then another developer can recreate the environment:

```
python -m venv .venv
```

```
pip install -r requirements.txt
```

---

16. A Good Python Project Structure

A small Python project might start like this:

```
expense_tracker/
│
├── .gitignore
├── README.md
├── requirements.txt
├── app.py
│
└── src/
    └── expense_tracker/
        ├── __init__.py
        ├── models.py
        ├── services.py
        └── utils.py
```

The exact structure depends on project size.

Do not create complicated folders just to look professional.

Structure should solve a real problem.

---

17. README.md

A GitHub repository should explain what the project does.

A basic README can contain:

````
# Expense Tracker

A Python application for recording and analyzing expenses.

## Features

- Add expenses
- View expenses
- Calculate totals
- Export data

## Requirements

- Python 3.12+

## Installation

```bash
````
````

```
python -m venv .venv
pip install -r requirements.txt
```

Run

```
python app.py
```

```
A good README reduces the amount of explanation another developer needs.

---

# 18. Connect a Local Repository to GitHub

After creating an empty repository on GitHub, connect it to your local
project.

```bash
git remote add origin https://github.com/USERNAME/expense-tracker.git
```

Check:

```
git remote -v
```

Then push:

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

The `-u` establishes the upstream relationship so future pushes can usually use:

```
git push
```

---

19. Branches

Branches allow you to work on different versions of a project without changing the main branch directly.

Create a branch:

```
git switch -c add-csv-export
```

Now you can work on the feature.

```
git add .
git commit -m "Add CSV export"
```

Then switch back:
```

```
git switch main
```

---

20. Why Feature Branches Help

Suppose the main application works.

You want to add authentication.

Instead of experimenting directly on `main`:

```
main
```

create:

```
feature/authentication
```

Your repository becomes:

```
main
   \
    feature/authentication
```

You can develop and test the feature separately.

---

21. Merging a Branch

After finishing the feature:

```
git switch main
```

Then:

```
git merge feature/authentication
```

The feature becomes part of the main branch.

---

22. Pulling Changes

If a remote repository has newer changes:

```
git pull
```

Conceptually:

```
Remote repository
       ↓
     pull
```

```
        ↓
Local repository
```

Before starting work on an existing project, a common workflow is:

```
git pull
```

Then make your changes.

---

23. Push vs Pull

Remember:

`git push`

Sends your local commits to the remote repository.

```
Computer → GitHub
```

`git pull`

Gets remote changes into your local repository.

```
GitHub → Computer
```

---

24. Git Status as a Daily Tool

One of the most useful commands is:

```
git status
```

Use it frequently.

It answers questions such as:

- What changed?
- What is staged?
- What is not tracked?
- Which branch am I on?
- Are there conflicts?

Do not treat `git status` as a command you only run when something goes wrong.

Use it throughout development.

---

25. Undoing Mistakes

Git gives you several ways to recover from mistakes.

For example, if you changed a file but have not committed it, you can inspect the difference first:

```
git diff
```

For staged changes:

```
git diff --staged
```

To unstage a file:

```
git restore --staged app.py
```

To discard local changes to a file:

```
git restore app.py
```

Be careful with destructive commands.

Always understand what will be removed before running them.

---

26. Git Repositories Are Not Backups of Everything

Git tracks project history.

But Git is not a reason to commit:

```
.env
.venv/
large generated files
passwords
API keys
temporary files
```

Version control should contain the **source and configuration needed to understand and recreate the project**, not every file on your computer.

---

27. Git + Python Development Workflow

A practical workflow looks like this:

```
1. Create project
        ↓
2. Create virtual environment
        ↓
3. Create .gitignore
        ↓
4. Initialize Git
        ↓
```

```
5. Build a small feature
        ↓
6. Run the program
        ↓
7. Run tests
        ↓
8. Review git diff
        ↓
9. Commit
        ↓
10. Push
```

Repeat this cycle as the project grows.

---

28. Example: Building a Python Expense Tracker

Suppose you want to build:

```
Expense Tracker
```

Initial structure:

```
expense-tracker/
│
├── .gitignore
├── README.md
├── requirements.txt
└── app.py
```

Initialize Git:

```
git init
```

Create the initial application:

```
expenses = []

expenses.append({
    "name": "Lunch",
    "amount": 800
})

print(expenses)
```

Commit:

```
git add .
git commit -m "Create expense tracker"
```

---

29. Add a Feature

Next, create a function:

```
def add_expense(name, amount):
    expenses.append({
        "name": name,
        "amount": amount
    })
```

Test it:

```
add_expense("Transport", 500)

print(expenses)
```

Commit:

```
git add .
git commit -m "Add expense creation"
```

---

30. Add Validation

Now improve the function:

```
def add_expense(name, amount):
    if amount <= 0:
        raise ValueError("Amount must be greater than zero")

    expenses.append({
        "name": name,
        "amount": amount
    })
```

Commit:

```
git add .
git commit -m "Validate expense amounts"
```

Your history now tells a story:

```
Create expense tracker
        ↓
Add expense creation
        ↓
Validate expense amounts
```

This is much more useful than one giant commit.

---

31. Git Helps You Experiment

Suppose you want to completely redesign your application.

Create a branch:

```
git switch -c redesign-expense-tracker
```

Experiment there.

If the experiment works:

```
Merge it
```

If it does not:

```
Return to main
```

This makes experimentation safer.

---

32. GitHub as a Python Portfolio

A GitHub profile can demonstrate more than your ability to write code.

A Python project can show:

- project organization
- clean commits
- documentation
- testing
- dependency management
- debugging
- Git usage
- problem solving
- software design

Instead of uploading random `.py` files, create complete projects.

---

33. What Makes a Python Project Portfolio Ready?

A useful project should ideally have:

```
README.md
.gitignore
requirements.txt or pyproject.toml
clear project structure
meaningful commits
tests
example usage
configuration instructions
```

Depending on the project, also include:

```
screenshots
API documentation
architecture notes
sample data
deployment instructions
```

---

34. Commit Messages as Project Documentation

Consider these two histories.

Poor

```
update
fix
more changes
final
final2
working
```

Useful

```
Create resource model
Add resource validation
Add resource repository
Add search functionality
Add API endpoint
Add tests for resource search
```

The second history communicates development decisions.

---

35. Git Branch Naming

Use names that describe the purpose.

Examples:

```
feature/search
feature/authentication
feature/csv-export
fix/resource-validation
refactor/resource-service
docs/setup-guide
```

Avoid:

```
branch1
test
new
stuff
mybranch
```

---

36. Common Git Mistakes in Python Projects

Mistake 1: Committing `.venv`

Bad:

```
git add .
```

without a `.gitignore`.

Fix:

```
.venv/
```

Mistake 2: Committing `.env`

Never commit real credentials.

Use:

```
.env
```

Mistake 3: Huge commits

Avoid waiting until the entire application is finished.

Commit meaningful milestones.

Mistake 4: Vague commit messages

Avoid:

```
update
```

Prefer:

```
Add expense filtering
```

Mistake 5: Working directly on `main`

For larger features, use feature branches.

Mistake 6: Not reading `git diff`

You might accidentally commit unrelated changes.

Use:

```
git diff
```

before staging.

Mistake 7: Ignoring the README

A repository without setup instructions can be difficult for others to use.

Mistake 8: Treating GitHub as a dumping ground

Ten unfinished scripts do not communicate the same thing as a few complete, documented projects.

---

37. A Better Python + Git Mental Model

Think in four layers:

```
Python
What does the program do?

Project structure
Where does each responsibility live?

Git
How does the project change over time?
```

```
GitHub
How is the project shared and collaborated on?
```

You need all four for professional project development.

---

38. Git + Testing

Git becomes especially useful when combined with tests.

Example workflow:

```
Write feature
      ↓
Run tests
      ↓
Tests pass
      ↓
Review changes
      ↓
Commit
```

You can then make another change.

If something breaks:

```
New code
   ↓
Tests fail
   ↓
Inspect recent changes
   ↓
Fix
```

Version control and automated testing complement each other.

---

39. Git + Debugging

Suppose your application worked yesterday.

Today it fails.

You can inspect recent commits:

```
git log --oneline
```

Then inspect changes:

```
git show <commit>
```

This can help identify when the behavior changed.

Git does not automatically find every bug, but it gives you historical evidence.

---

40. Git + Environment Configuration

A Python application might require:

```
DATABASE_URL=
API_KEY=
DEBUG=
```

The actual values belong to the local environment.

The project should document the required variables.

For example:

```
.env.example
```

```
DATABASE_URL=
API_KEY=
DEBUG=False
```

This creates a useful separation:

```
Repository
   ↓
Configuration requirements

Local environment
   ↓
Actual secret values
```

---

41. Git + Dependency Management

Your project might use:

```
requests
pytest
python-dotenv
```

The repository should record its dependencies.

For a requirements-based project:

```
requirements.txt
```

For modern Python projects, dependency and project configuration can also live in:

```
pyproject.toml
```

The goal is reproducibility.

Another developer should be able to clone the repository and recreate the environment.

---

42. Git + Code Quality

A professional Python workflow can combine:

```
Git
+
Virtual environment
+
Dependency management
+
Formatting
+
Linting
+
Testing
+
Documentation
```

Example:

```
Edit
 ↓
Format
 ↓
Lint
 ↓
Test
 ↓
Review diff
 ↓
Commit
 ↓
Push
```

This creates a repeatable development process.

---

43. A Real haas.dev Project Workflow

Imagine haas.dev stores resources in Python.

A simplified project might contain:

```
haas-resource-api/
│
├── .gitignore
├── README.md
├── requirements.txt
├── .env.example
│
├── src/
│   └── haas_resources/
│       ├── models.py
│       ├── services.py
│       └── repositories.py
│
└── tests/
    ├── test_models.py
    └── test_services.py
```

A development task might be:

```
Add resource search
```

Create a branch:

```
git switch -c feature/resource-search
```

Implement the feature.

Run tests:

```
pytest
```

Format and lint:

```
ruff check .
ruff format .
```

Review:

```
git diff
```

Commit:

```
git add .
git commit -m "Add resource search"
```

Push:

```
git push -u origin feature/resource-search
```

Now the change is isolated, tested, documented through Git history, and ready for review.

---

44. The Professional Project Loop

A strong Python development loop is:

```
PLAN
 ↓
CREATE BRANCH
 ↓
IMPLEMENT
 ↓
TEST
 ↓
FORMAT
 ↓
LINT
 ↓
REVIEW DIFF
 ↓
COMMIT
 ↓
PUSH
 ↓
REVIEW / MERGE
```

This is more important than memorizing dozens of Git commands.

---

45. When Should You Commit?

Commit when you have a **meaningful, working change**.

Good examples:

```
Add user validation
Add resource search
Fix invalid date handling
```

```
Add CSV export
Add tests for authentication
Update setup documentation
```

Avoid committing every keystroke.

Also avoid waiting weeks before committing.

---

46. Git Commit Granularity

Think:

```
One commit = one understandable change
```

Not necessarily:

```
One commit = one file
```

A feature may require several files.

For example:

```
models.py
services.py
tests/test_services.py
```

could all belong in:

```
Add resource search
```

if they represent one coherent change.

---

47. Mini Project: Git-Tracked Python CLI

Build a simple task manager.

Requirements:

```
Add task
List tasks
Complete task
Delete task
Save tasks
Load tasks
```

Suggested structure:

```
task-manager/
│
├── .gitignore
├── README.md
├── requirements.txt
├── app.py
├── tasks.py
└── tests/
    └── test_tasks.py
```

Development sequence:

```
Commit 1
Create project structure

Commit 2
Add task creation

Commit 3
Add task listing

Commit 4
Add task completion

Commit 5
Add task deletion

Commit 6
Add file persistence

Commit 7
Add tests

Commit 8
Improve documentation
```

The final repository becomes both a working project and a demonstration of your development process.

---

48. 5 Minute Git + Python Challenge

Create a directory:

```
python-git-demo/
```

Inside it:

```
app.py
README.md
.gitignore
```

Initialize Git:

```
git init
```

Add:

```
def greet(name):
    return f"Hello, {name}!"
```

Commit:

```
git add .
git commit -m "Add greeting function"
```

Then modify the function.

Before committing again:

```
git diff
```

Your goal is to understand exactly what Git sees between two versions of your Python project.

---

49. Exercises

Exercise 1: Repository Setup

Create a Python project and:

- initialize Git
- create `.gitignore`
- create `README.md`
- create a virtual environment
- make the first commit

Exercise 2: Feature Branch

Create:

```
feature/calculator
```

Add calculator functionality.

Commit it separately.

Exercise 3: History

Use:

```
git log --oneline
```

Explain what each commit represents.

Exercise 4: Diff

Change a Python function.

Run:

```
git diff
```

Explain every changed line.

Exercise 5: GitHub

Push your project to GitHub.

Verify that:

- `.venv` is absent
- `.env` is absent
- README is visible
- source code is organized

---

50. Mini Quiz

1. What does Git do?

A. Runs Python code

B. Tracks changes to a project

C. Installs Python packages

D. Hosts websites

Answer: B

---

2. What does `git add` do?

A. Deletes files

B. Uploads files to GitHub

C. Stages changes for a commit

D. Creates a Python environment

Answer: C

---

3. What does `git commit` create?

A. A Python package

B. A saved point in Git history

C. A virtual environment

D. A GitHub account

Answer: B

---

4. Should `.venv/` normally be committed?

A. Yes

B. No

Answer: B

---

5. What is the difference between `git push` and `git pull`?

Answer:

```
push = local → remote
pull = remote → local
```

---

51. Interview Questions

Beginner

1. What is Git?
2. What is a Git repository?
3. What is GitHub?
4. What is a commit?
5. What does `git add` do?
6. What does `git status` show?
7. What is `.gitignore`?
8. Why should `.venv` not be committed?
9. What is the difference between Git and GitHub?

10. What is a branch?

Intermediate

11. What is the staging area?
12. Why use feature branches?
13. What is a merge?
14. What is a remote repository?
15. What is the difference between `git fetch` and `git pull`?
16. Why are meaningful commits important?
17. How should secrets be handled in a Python project?
18. What belongs in a Python repository?
19. How can Git help with debugging?
20. How does Git work with automated tests?

Practical

21. How would you structure a Git repository for a Python API?
22. How would you prevent `.env` from being committed?
23. How would you recover from a bad local change?
24. How would you develop a large feature without destabilizing `main`?
25. What steps would you take before pushing a Python feature?

---

52. Cheat Sheet

Start a Repository

```
git init
```

Check Status

```
git status
```

Stage

```
git add .
```

Commit

```
git commit -m "Add feature"
```

View History

```
git log --oneline
```

View Changes

```
git diff
```

View Staged Changes

```
git diff --staged
```

Create Branch

```
git switch -c feature/name
```

Switch Branch

```
git switch main
```

Merge

```
git merge feature/name
```

Add Remote

```
git remote add origin <repository-url>
```

Push

```
git push
```

Pull

```
git pull
```

Unstage

```
git restore --staged filename
```

Discard Local File Changes

```
git restore filename
```

---

53. The Core Mental Model

Remember this:

```
Python
→ builds the software

Project structure
→ organizes the software

Git
→ tracks the software's history

GitHub
→ shares and collaborates on the software

README
→ explains the software

.gitignore
→ keeps irrelevant and sensitive files out

Tests
→ verify the software

Branches
→ isolate changes
```

The goal is not to memorize Git commands.

The goal is to develop the habit of building Python projects that are:

```
Organized
+
Reproducible
+
Tested
+
Documented
+
Version controlled
```

That is the difference between writing a Python script and maintaining a Python project.

Key Takeaways

- Git tracks changes in your Python project.
- GitHub hosts Git repositories online.

- A commit represents a meaningful point in project history.
- The staging area lets you choose what goes into a commit.
- `git status` and `git diff` are essential daily commands.
- `.gitignore` prevents unnecessary and sensitive files from entering the repository.
- Never commit real API keys, passwords, or secrets.
- Do not commit `.venv`; recreate it from dependency information.
- Use meaningful commit messages.
- Use feature branches for isolated work.
- A README should explain how to understand and run the project.
- Git becomes more powerful when combined with testing, formatting, linting, and dependency management.
- A well-maintained GitHub repository can demonstrate real software development skills.

---

haas.dev

https://dev-roast-app.vercel.app

Visit haas.dev for more resources and guides.