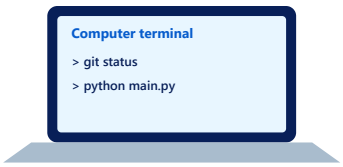
START HERE

Keep a way back

01

A Git checkpoint lets you compare a later experiment with a saved project. Your goal: save a small working program, make one change, then recover a selected file after inspecting it. Allow about **15 minutes**, including practice.

Recognize the tools



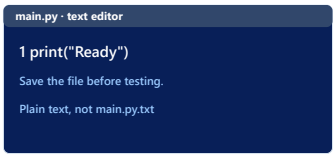
Computer + terminal

Runs Git commands and the small test program.



Project folder

Holds your files and Git's local history.



Plain text editor

Edits and saves `main.py`. Save before you test.

Before starting: install Git from the official Git site, and desktop Python. Open a terminal and check `git --version` and `python --version`.

A checkpoint has two kinds of evidence

1

Run the program

Observe the expected text

2

Inspect the exact change

Stage only intended files

3

Record the checkpoint

Find it again in history

Clean status means no pending Git changes. It is not a test result.

Git records file contents. You still need to test what those contents do.

Use a new practice folder. Keep this exercise separate from a real project. A local commit stays on this computer; a separate backup or approved remote copy protects against losing the computer.

No electronics, Git account, remote service or development board is needed. If your Python command is `python3` or `py`, use that command for the two program runs.

THE THREE PLACES

02

Know where the change is

A **repository** is a project with Git history. The **working folder** is what you edit. The **staging area**, also called the index, holds the contents selected for your next **commit**: a saved snapshot with a message.



Save in your editor → inspect → select → record

The file can have different contents in these three places.

git status

Shows new files and changes that are staged or still unstaged. Read the headings, not only the colors.

git diff / git diff --cached

The first compares working files with staging. The second reviews staging against the latest commit.

Adding is a snapshot

Staged at time of add

"Ready for the next step"

This is what commit records.

Edited again afterward

"Unreviewed experiment"

This stays unstaged.

git add is a snapshot, not a live link to future edits.

If you edit again after adding, review the new edit and add again only if you want it in the commit.

Predict: you add a file, then type one more sentence. Will that sentence be in the next ordinary commit?
Check: only if you add the new contents before committing.

PRACTICE · PREPARE

03

Create a safe practice project

Use a new folder called `git-practice` in your own working area. Do not run these setup commands inside a shared project or an existing repository.

1 · Initialize the folder

```
mkdir git-practice
cd git-practice
git init
git config user.name "Course Learner"
git config user.email "learner@example.invalid"
```

The two identity settings apply to **this repository** because these commands do not use `--global`. The example identity is for this throwaway exercise. Use your intended author identity in a real project.

2 · Make two plain text files

main.py

```
print("Ready")
```

One small program. Make sure the filename is not `main.py.txt`.

.gitignore

```
.venv/
__pycache__/
.env
private-notes.txt
```

One pattern on each line. Keep this file itself in Git.

PRACTICE · KEEP PRIVATE FILES OUT

04

Choose the files you will keep

Your source code helps you rebuild a project. Temporary dependencies and private information need different treatment. Inspect the file list before staging.

Track useful project files

main.py

.gitignore

short test / setup notes

Keep these out

.env / credentials

.venv/ / __pycache__/

private notes and generated clutter

Ignore rules affect untracked files; they do not erase history.

Choose useful source files deliberately. Do not put passwords in a practice file.

Read the four ignore patterns

Generated folders

`.venv/` holds a Python environment. `__pycache__/` holds automatically generated Python cache files. The ending slash identifies a folder pattern.

Private files

`.env` is often used for settings or credentials. `private-notes.txt` is an example personal note. Do not put real secrets into this exercise.

```
git status
```

Before you add: can you explain every file shown? Select the source and ignore file by name. If an unfamiliar or private file appears, leave it out and investigate.

Ignore is not a cleanup tool. Rules stop matching untracked files from being added normally. They do not remove already tracked files or secrets from earlier commits. If a real secret was committed, stop sharing it and get help replacing that credential.

PRACTICE · OBSERVE, SELECT, RECORD

Save the first checkpoint

05

Run the program first, then review exactly what will be saved. These commands all run inside `git-practice`.

1 · Observe a working result

```
python main.py
```

Expected reply: Ready. If you see a Python error, fix it and repeat this run before continuing.

2 · Inspect and select the two files

```
git status
git add -- main.py .gitignore
git diff --cached
git status
```

Read the staged patch: it should contain only the one-line program and the four ignore patterns. The `--` separates options from the file names. This lesson names files explicitly so you can see what you selected.

3 · Record and find the checkpoint

```
git commit -m "Save ready greeting"
git log --oneline -2
git status
```

Log observation: one entry should end with `Save ready greeting`. The characters before it identify your saved commit; they will differ on each computer.

Check: find `Save ready greeting` in the log. Status should report a clean working tree if there are no other files. The commit is local; it does not publish anything.

PRACTICE · COMPARE BEFORE SAVING

06

Make one small change

In your editor, replace the message in `main.py` with `Ready for the next step`. Save the file, then run it again.

```
python main.py
git diff -- main.py
```

git diff -- main.py

```
- print("Ready")
+ print("Ready for the next step")
```

- old line + new line

Read the words and the signs. Color is an extra cue.

A minus line removes the old text. A plus line adds its replacement. Do not type these patch signs into your program.

Stage and review the intended change

```
git add -- main.py
git diff --cached -- main.py
git status
```

Check that the new greeting is the staged version. If you edit the file again now, staging still holds the text from your last `git add`.

Need to unstage it?

```
git restore --staged -- main.py
```

After the first commit exists, this returns the staged entry to the latest commit. Your working edits stay in the file. Add again when ready.

Ready for the checkpoint?

```
git commit -m "Explain ready message"
git log --oneline -2
```

Use this after the intended file is staged. The message says what changed.

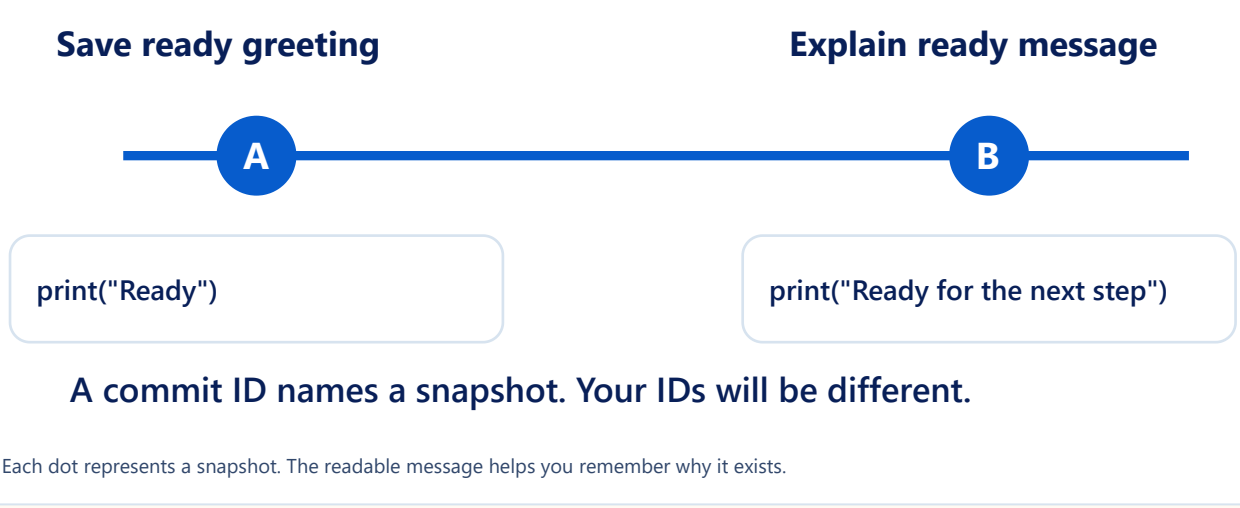
Check: the program prints your longer message, the log contains two checkpoints, and `git status` is clean. A clean status alone does not prove the program worked.

PRACTICE · INSPECT HISTORY

07

Read the saved versions

After the second checkpoint, the history has two saved project versions. You can read an earlier file without replacing your current one.



Each dot represents a snapshot. The readable message helps you remember why it exists.

```
git log --oneline -2
git show HEAD:main.py
```

The newest log entry appears first. **HEAD** names the latest commit at your current position. `HEAD:main.py` means that saved file inside that snapshot.

Inspect the earlier greeting

Copy the commit ID beside `Save ready greeting` from your own log. In the command below, replace `FIRST_ID` with that ID; do not type FIRST_ID literally.

```
git show FIRST_ID:main.py
```

Expected: the earlier snapshot shows `print("Ready")`. Your working file still contains the longer message. Reading a snapshot has not changed that file.

Point to the difference: identify the first commit, second commit and current working file. Explain which command only reads saved contents.

PRACTICE · INSPECT BEFORE REPLACING

Recover one file with care

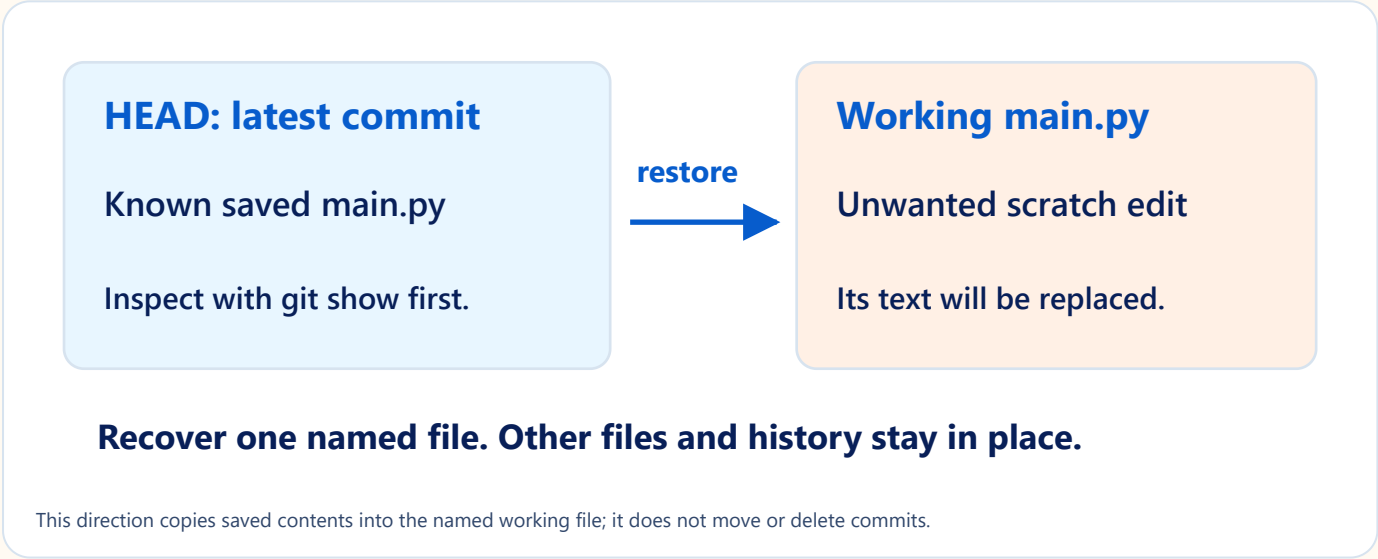
08

Now make a disposable scratch edit: change the message to `Unwanted scratch`, save it, and **do not add it**. We will replace only this working file with its latest committed version.

1 · Inspect both versions first

```
git status
git diff -- main.py
git show HEAD:main.py
```

HEAD means the latest commit at your current position. The show command displays its saved `main.py` without replacing your file.



This replaces uncommitted text in main.py. Continue only when the diff is exactly the scratch edit you want to discard. If the file contains anything valuable, save a separate copy outside the practice folder first. Keep the file unstaged for this exercise.

2 · Restore the named file, then test

```
git restore --source=HEAD --worktree -- main.py
python main.py
git status
```

Expected: the program prints `Ready for the next step` again. Status is clean because the working file matches its saved version. The other files and two commits remain in place.

For an older saved version, inspect it with `git show COMMIT_ID:main.py` first, replacing COMMIT_ID with an ID from your log. This exercise restores HEAD only.

PRACTICE AND NEXT STEPS

09

Use the checkpoint in a real build

Git stores your chosen files. A useful hardware checkpoint also says what you tested, with which setup and what you observed.

Software observation

"main.py prints Ready for the next step after I run it." That is the result tested in this practice.

Device observation

Record the actual board, firmware, connection and visible result when you test hardware. A successful commit does not prove a circuit works.

Your turn: a new small checkpoint

1. Change the greeting to your own short message, save and run it.
2. Use status and diff to inspect the intended change.
3. Add only `main.py`, review the staged diff and commit with a clear message.
4. Find the new message in the log and explain which version was saved.

Completion check

- ☐ Point to working files, staging and history in the diagram.
- ☐ Explain why an edit after adding needs another review and add.
- ☐ Find your checkpoints and show the saved file without changing it.
- ☐ Recover the disposable scratch edit in one named file.

Before sharing a repository: inspect the selected files for credentials and private information. Keep a separate backup. Ignoring a file now does not remove it from past history.

If a command fails: "not a git repository" → check your folder; "identity unknown" → set this practice repo's identity; "nothing to commit" → inspect status, save the editor file and review what is staged.