Lecture 2

Introduction to `git`

Development Tools for Scientific Computing - SISSA, 2024-2025

Pasquale Claudio Africa, Dario Coscia

11 Feb 2025

Outline

1. Introduction to `git`:
    - Local vs. remote.
    - Branching and collaborative working.
    - Sync the course material with your computer.
2. `git` internals.
3. Hands on `git`.

Introduction to `git`

Version control

Version control, also known as source control, is the practice of tracking and managing changes to software code. Version control systems are software tools that help software teams manage changes to source code over time.

`git` is a free and open-source version control system, originally created by Linus Torvalds in 2005. Unlike older centralized version control systems such as SVN and CVS, Git is distributed: every developer has the full history of their code repository locally. This makes the initial clone of the repository slower, but subsequent operations dramatically faster.

How does `git` work?

1. Create (or find) a repository with a git hosting tool (an online platform that hosts you project, like GitHub or Gitlab).
2. `git clone` (download) the repository.
3. `git add` a file to your local repo.
4. `git commit` (save) the changes, this is a local action, the remote repository (the one in the cloud) is still unchanged.
5. `git push` your changes, this action synchronizes your version with the one in the hosting platform.

How does `git` work? (Collaborative)

If you and your teammates work on different files the workflow is the same as before, you just have to remember to `pull` the changes that your colleagues made.

If you have to work on the same files, the best practice is to create a new `branch`, which is a particular version of the code that branches form the main one. After you have finished working on your feature you `merge` the branch into the main.

Other useful `git` commands

- `git diff` shows the differences between your code and the last commit.
- `git status` lists the status of all the files (e.g. which files have been changed, which are new, which are deleted and which have been added).
- `git log` shows the history of commits.
- `git checkout` switches to a specific commit or brach.
- `git stash` temporarily hides all the modified tracked files.

An excellent visual cheatsheet can be found here .

SSH authentication

1. Sign up for a GitHub account.
2. Create a SSH key.
3. Add it to your account.
4. Configure your machine:

```
git config --global user.name "Name Surname"
git config --global user.email "name.surname@email.com"
```

See here for more details on SSH authentication.

The course repository

Clone the course repository:

```
git clone git@github.com:pcafrica/devtools_scicomp.git
```

Before every lecture, download the latest updates by running:

```
git pull origin main
```

from inside the cloned folder.

Advanced `git` usage

Working with branches (1/4)

Branching is a core `git` feature that allows developers to work on different tasks simultaneously without affecting the main codebase. Branches enable parallel development and help in organizing features, bug fixes, and experimental changes. A branch should be created for each new feature or fix, ensuring an isolated development environment.

Creating and switching branches

```
# Create a new branch.
git branch feature-branch

# Switch to a branch.
git checkout feature-branch

# Create and switch to a new branch in one step.
git checkout -b new-feature
```

Working with branches: merging vs. rebasing (2/4)

Working with branches (3/4)

Merging branches

Merging integrates changes from one branch into another. This is commonly used when a feature is complete and needs to be incorporated into the main development line.

```
# Merge a branch into the current branch.
git merge feature-branch

# Merge while keeping history.
git merge --no-ff feature-branch
```

Using `--no-ff` ensures a merge commit is created, maintaining the historical context of the branch.

Working with branches (4/4)

Rebasing

Rebasing is an alternative to merging that results in a cleaner, linear commit history.

```
# Rebase the current branch onto main.
git rebase main

# Interactive rebase (modify, squash, reorder commits).
git rebase -i HEAD~3
```

During an interactive rebase, you can squash commits, edit messages, or reorder commits to keep the history tidy.

Managing commits (1/2)

Undoing changes

```
# Reset staged files.
git reset HEAD <file>

# Undo last commit (keep changes staged).
git reset --soft HEAD~1

# Undo last commit (discard changes).
git reset --hard HEAD~1
```

Be cautious with `--hard` as it discards changes permanently.

Managing commits (2/2)

Amending commits

```
# Edit the last commit message.
git commit --amend -m "New commit message"

# Amend the last commit by adding new files/changes.
git add <file>
git commit --amend --no-edit
```

Working with remote repositories

Remote repositories facilitate collaboration by allowing multiple developers to work on a shared codebase.

Pushing and pulling changes

```
# Push changes to a remote branch.
git push origin feature-branch

# Pull latest changes from remote.
git pull origin main
```

Force pushing (not recommended)

```
# Force push (overwrites remote history).
git push --force

# Force push with safety (does not overwrite if history diverged).
git push --force-with-lease
```

Stashing changes

When switching branches or pulling updates, you may need to temporarily save uncommitted changes.

```
# Save uncommitted changes.
git stash

# Apply the last stashed changes.
git stash apply

# Apply and remove the last stash.
git stash pop

# List all stashed changes.
git stash list

# Stash with a message.
git stash save "WIP: fixing bug"
```

Working with tags

Tags mark specific commits, often for versioning releases.

```
# Create an annotated tag.
git tag -a v1.0 -m "Version 1.0 release"

# List tags.
git tag

# Push tags to remote.
git push origin --tags
```

Debugging with `git`

Viewing commit history

```
# Pretty log format.
git log --oneline --graph --decorate --all

# Show commit differences.
git log -p
```

Finding bugs

```
# Show line-by-line changes in a file.
git blame <file>

# Find when a bug was introduced.
git bisect start
git bisect bad # Mark current commit as bad.
git bisect good <commit>
```

Cleaning up

```
# Remove untracked files.
git clean -f

# Remove untracked directories.
git clean -fd

# Remove all local branches except main.
git branch | grep -v "main" | xargs git branch -D
```

Advanced configuration

```
# Set global ignore file.
git config --global core.excludesfile ~/.gitignore_global

# Set default editor.
git config --global core.editor "vim"
```

Working with submodules

```
# Add a submodule.
git submodule add <repository-url> path/to/submodule

# Clone a repo with submodules.
git clone --recurse-submodules <repo-url>

# Update submodules.
git submodule update --init --recursive
```

Automating `git` with aliases

```
# Set up an alias for a pretty log view
git config --global alias.lg "log --oneline --graph --decorate --all"

# Set up an alias for rebasing interactively
git config --global alias.ri "rebase -i"
```

`git` internals

`git` objects (1/2)

Git is fundamentally a content-addressable filesystem with a unique way of storing data. Instead of tracking files and directories directly, Git stores four main types of objects:

- Blobs (Binary Large Objects)
  - Store file contents (not filenames or metadata).
  - Identified by a SHA-1 hash.
  - Example: Running `git hash-object -w myfile.txt` creates a blob.
- Trees
  - Store file names and directory structure.
  - A tree object points to multiple blobs (files) and other trees (subdirectories).
  - Example: Running `git ls-tree HEAD` shows the tree for the latest commit.

`git` objects (2/2)

Git is fundamentally a content-addressable filesystem with a unique way of storing data. Instead of tracking files and directories directly, Git stores four main types of objects:

- Commits
  - Contain metadata (author, timestamp, message) and point to a tree.
  - Link to a parent commit, forming a history.
  - Example: Running `git cat-file -p HEAD` reveals a commit's details.
- Tags
  - Used to label specific commits (e.g., software releases).
  - Lightweight (pointer to a commit) or Annotated (stores extra metadata).
  - Example: `git tag -a v1.0 -m "First release"` creates an annotated tag.

The `.git` directory structure

```
.git/
|-- objects/    # Stores commits, trees, and blobs.
|-- refs/       # Stores pointers to branches and tags.
|-- logs/       # Records operations history.
|-- config      # Repository settings.
|-- index       # Changes that have been staged but not yet committed.
|-- HEAD        # Points to the current branch.
```

Key commands to explore:

- `ls .git/objects/`: Shows stored `git` objects.
- `cat .git/HEAD`: Shows which branch you are currently on.
- `git reflog`: Uses logs to recover lost commits.

How `git` handles diffs and compression

- Delta compression:
    - Git doesn't store multiple versions of a file. Instead, it stores the differences (deltas) to save space.
    - `git gc` (garbage collection) optimizes storage by compressing deltas.
- Packfiles (`.pack` and `.idx` files in `.git/objects/pack/`):
    - Large repositories use packfiles to group objects and speed up cloning/pulling.
    - `git verify-pack -v .git/objects/pack/*.idx1` shows compressed object details.

Plumbing vs. porcelain commands

Porcelain (user-friendly)	**Plumbing (internal mechanisms)**
`git commit`	`git hash-object`
`git branch`	`git update-ref`
`git log`	`git cat-file`
`git checkout`	`git read-tree`

Manually creating a commit

Challenge: Instead of using git commit, you may manually create a commit using plumbing commands:

```
echo "Hello, git internals" > file.txt
git init
git add file.txt
git hash-object -w file.txt                               # Store the blob manually.
git write-tree                                            # Create a tree object.
git commit-tree TREE_HASH -p PARENT_HASH -m "Manual commit"  # Create a commit manually.
git update-ref refs/heads/main COMMIT_HASH
```

Hands on `git`

Collaborative `git`

`git` flow with forking is a lightweight, branch-based workflow designed for fast and continuous collaboration. It works well for both team projects and open-source contributions. When contributing to repositories you don't have direct access to (e.g., open-source projects), **forking** is required.

Cloning vs. forking

- **Cloning**: Used when you have write access to a repository.
- **Forking**: Used when contributing to a repository you **don’t** have write access to.

How to fork a repository

1. Navigate to the repository on GitHub.
2. Click the **Fork** button (top right).
3. This creates a copy of the repository under your GitHub account.
4. Clone the forked repository to your local machine:
   ```
   git clone git@github.com:your-username/repo-name.git
   ```
5. Add the original repository (upstream) as a *remote*:
   ```
   git remote add upstream git@github.com:original-owner/repo-name.git
   ```

`git` flow with forking (1/4)

Once you have forked a repository, follow these steps to contribute:

Step 1: create a branch

Never work directly on `main`. Instead, create a new feature branch:

```
git checkout -b feature-branch
```

Step 2: make and commit changes

Write your code and commit frequently:

```
git add .
git commit -m "Added feature X"
```

`git` flow with forking (2/4)

Step 3: push to your fork

Push the branch to **your fork** (not the upstream repo):

```
git push origin feature-branch
```

Step 4: open a pull request (PR)

1. Navigate to the **original repository** on GitHub.
2. Click **Compare & pull request**.
3. Select your branch and explain the changes.
4. Request reviews from maintainers.

`git` flow with forking (3/4)

Step 5: keep your fork updated

Before making further changes, always sync your fork with the upstream repository:

```
git checkout main
git pull upstream main
git push origin main
```

`git` flow with forking (4/4)

Step 6: merge and delete the branch

Once the PR is approved and merged:

1. Delete your local branch:
   ```
   git branch -d feature-branch
   ```
2. Delete it on GitHub:
   ```
   git push origin --delete feature-branch
   ```

Exercise 1: your first pull request

1. Fork the course repo.
2. Create a branch.
3. Make some changes, commit and push them to your fork.
4. Open a PR on the original repo.

Exercise 2: collaborative file management (1/3)

1. Form groups of 2-3 members.
2. Designate one member to create a new repository (visit https://github.com/ and click the `+` button in the top right corner), and ensure everyone clones it.
3. In a sequential manner, each group member should create a file with a distinct name and push it to the online repository while the remaining members pull the changes.
4. Repeat step 3, but this time, each participant should modify a different file than the ones modified by the other members of the group.

Exercise 2: collaborative file management (2/3)

Now, let's work on the same file, `main.py`. Each person should create a hello world `main.py` that includes a personalized greeting with your name. To prevent conflicts, follow these steps:

1. Create a unique branch using the command: `git checkout -b [new_branch]`.
2. Develop your code and push your branch to the online repository: `git push origin [new_branch]`.
3. Once everyone has finished their work, merge your branch into the `main` branch using the following commands:

```
git checkout main
git pull origin main
git merge [new_branch]
git push origin main
```

How to deal with `git` conflicts

The first person to complete this process will experience no issues. However, subsequent participants may encounter merge conflicts.

`git` will mark the conflicting sections in the file. You'll see these sections surrounded by `<<<<<<<`, `=======`, and `>>>>>>>` markers.

Carefully review the conflicting sections and decide which changes to keep. Remove the conflict markers (`<<<<<<<`, `=======`, `>>>>>>>`) and make the necessary adjustments to the code to integrate both sets of changes correctly.

After resolving the conflict, commit your changes and push your resolution to the repository.

➡️ Python for scientific computing. CI/CD