ENSF400: Software Engineering Industry Practices and Communication

Tutorial 2: Version Control with Git and GitHub

Winter 2026

Department of Electrical and Software Engineering

Schulich School of Engineering

University of Calgary

Git

Git is a **distributed** version control system that **tracks changes to files** over time so you can undo mistakes, compare versions, and collaborate with others by merging work from multiple people.

Installing Git

Visit the official `git` install website (https://git-scm.com/install), and follow the instructions for your operating system

GitHub

GitHub is a web-based platform for hosting Git repositories and collaborating on software projects, providing tools like pull requests, issues, code review, and releases.

Foundational Git Terminology

- **Working tree**: The set of nested directories and files that contain the project that is being worked on.

- **Repository (repo)**: A project tracked by Git. The Git database (history, objects, refs, config) lives in the `.git/` **directory** inside the project folder.
- **Hash**: A number produced by a hash function that represents the contents of a file or another object as a fixed number of characters. Git traditionally uses SHA-1 (160-bit) to create hashes.
- **Commit:** A commit is a saved snapshot of one or more file changes on your branch. Git gives each commit a unique identifier (a SHA/hash) that ties together what changed, who made the change, and when the change was made.
- **Branch:** A branch is a named series of linked commits. The default branch, which is created when you initialize a repository, is usually called main.
- **HEAD**: A special pointer to the commit you currently have checked out (usually the current branch).

Case Study: Python Calculator Library

To illustrate the capabilities of Git, we will work through a Git-tracked demo project. We will start by initializing a local repository, then make it public on GitHub, and use it to demonstrate common real-world workflows. Along the way, we will learn how to:

- track versions of the project over time,
- revert to known-good states when something breaks,
- collaborate effectively with others once the project is public.

Preliminary Steps: Configuring Git

A. Git config (identity and defaults)

```
git config --global user.name "myname"
git config --global user.email "name@example.com"
git config --global init.defaultBranch main #The default from
```

```
Git version 3
git config --global --list
```

B. Repository Setup

- Navigate to a new folder
- run the following command: `git init`
- Add a `README.md` file

File States in Git

In Git, a file is in one of the following states:

- **untracked**: new files Git doesn't know about yet
- **Modified**: You have changed the file, but those changes are not recorded in Git yet.
  - View changes: `git diff`
  - Move **Modified** → **Clean** (discard uncommitted changes): `git restore <file>`
- **Staged**: You have marked the current version of the file to be included in the next commit.
  - Move **Modified** → **Staged**: `git add <file>` (or `git add .`)
  - Move **Staged** → **Modified** (unstage, keep changes): `git restore --staged <file>`
- **Committed**: The file's changes are saved in the repository history as part of a commit.
  - Move **Staged** → **Committed**: `git commit -m "message"`
  - Undoing Commits: We undo commits using the `git revert` or `git reset` commands.
  - As a general rule of thumb `reset` can be used safely when work has not been shared with others, while `revert` is better when others already depend on our commit (to avoid rewriting history)

C. Move `README.md` to the Staging Area

Add some content to `README.md`:

```
# A Python Calculator

The best calculator in the world!
```

Then execute the following command to add it to the staging area:

```
git add README.md
```

`README.md` is now in the staging area. We can `commit` files that we have placed in the staging area. First a note on commits.

Commits

A commit records a snapshot of your project at a point in time. It includes the changes Git is tracking, metadata like author and timestamp, and links to parent commit(s). Each commit should represent a single, logical change, and it should include a clear message describing what and why. This has a few key advantages:

- **Clear history:** It's easy to understand what changed and why.
- **Easier debugging:** You can pinpoint when a bug was introduced.
- **Safer rollbacks:** You can revert specific changes without undoing unrelated work.

D. Commit: Add a README

With `README.md` in the staging area, run the following command:

```
git commit -m "chore: add README"
```

E. Add a .gitignore file

`.gitignore` prevents accidental commits of local artifacts. It does **not** affect files that are already tracked. We might want to exclude files containing sensitive information like API keys. Let's add some files to the `.gitignore` to be extra safe.

1. Create a file called `.gitignore`
2. Add the following content to the file

```
.venv/
.env
```

3. Commit it:

```
git add .gitignore
git commit -m "chore: add .gitignore"
```

Git workflow

We will begin by implementing the calculator in a local Git repository on your computer. In the first part of this section, we will work entirely locally, making changes, staging them, and creating a sequence of commits to examine how Git records and navigates project history. Later in this section, we will connect the repository to GitHub and publish it for sharing and collaboration.

A. Implement addition

In `calculator.py`, we implement addition

```
def add(a, b):
    return a + b
```

In your terminal, run the following commands. We move `calculator.py` to the staging area, then we make a commit after we make meaningful change to the repository

```
git add calculator.py
```

Before committing the change, in your terminal, run:

```
git status
```

This command shows the state of the working directory and staging area.

Finally, run

```
git commit -m "feat: implement addition"
```

B. Implement subtraction

In `calculator.py`:

```
def add(a, b):
    return a + b

def sub(a, b):
    return a - b
```

In the terminal:

```
git add calculator.py
git commit -m "feat: implement subtraction"
```

C. Implement multiplication

In `calculator.py`:

```
def add(a, b):
    return a + b

def sub(a, b):
    return a - b
```

```
def mul(a, b):
    return a * b
```

In the terminal:

```
git add calculator.py
git commit -m "feat: implement multiplication"
```

D. Implement division

In `calculator.py`:

```
def add(a, b):
    return a + b

def sub(a, b):
    return a - b

def mul(a, b):
    return a * b

def div(a, b):
    return a // b
```

In the terminal:

```
git add calculator.py
git commit -m "feat: implement division"
```

E. View Created Commits

In your terminal, run:

```
git log
```

`git log` shows the commits we have created so far in reverse chronological order, including the commit hash, commit message, author's name/email and date/time of the commit.

Creating a GitHub Repository

At this point, the project is in a good enough state to share, so we create a repository on GitHub. It hosts Git repositories and provides collaboration features like **Issues**, **Pull Requests**, and **Releases** to help teams discuss changes, review code, and publish stable versions.

A. Create a repository on GitHub

To create a repository on GitHub:

1. Sign in to GitHub.
2. Click **New** (or **+** → **New repository**).
3. Enter a repository name (and optional description).
4. Choose **Public** or **Private**.
5. Click **Create repository**.

B. Add the GitHub Repo as a `Remote` for the Local Repo

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

C. Create a tag for the current version

Tags

Git tags are names you attach to specific commits in a repository so you can easily refer back to important points in history, like a version you shipped. They're most often used to mark releases using version-like names such as `v1.2.0`. A tag can be a simple pointer to a commit or an "annotated" tag that also stores extra information like who created it, when, and a short message. Tags are part of Git itself, so they exist independently of any hosting platform. We can tag the current version of our repository and push it.

Tag the current commit. Tags should be stable and not moved later.

```
git tag -a v0.0.1 -m "Initial release"
```

```
git push -u origin main
git push origin v0.0.1
```

D. Create a release on GitHub

Releases on GitHub

GitHub releases are a GitHub feature built on top of tags. A release typically corresponds to a tag and adds a human-friendly page with a title and release notes, and it can include attached build artifacts like compiled binaries or install packages. In other words, the tag marks the exact commit in Git, and the GitHub release is the published announcement and packaging of that tagged version on GitHub.

To create a release on GitHub:

1. Open the repository on GitHub.
2. Go to **Releases**.
3. Click **Draft a new release**.
4. Choose an existing tag or create a new one.
5. Select the target branch/commit.
6. Add a release title and notes.
7. Click **Publish release**.

Branches

Now that the repository is public, others may want to contribute. We want the public `main` branch to remain a reliable, working version that anyone can clone and run without errors. We could keep a separate copy of the project for ongoing work, but that would become cumbersome quickly. Git makes this easy with **branches**, which let us develop new features and fixes in isolation and merge them into `main` only when they are ready.

A branch is a separate line of development in Git that points to a specific commit and moves forward as you make new commits. Branches allow you to work on new features, fixes, or experiments without affecting `main`, keeping the stable version clean for anyone cloning the repository. Git makes creating branches easy, and more importantly, makes it easy to merge work done on branches back into the main project when it's ready.

We want to implement trigonometric functions (sin, cos, tan). To keep the work organized and avoid destabilizing `main`, we create a separate feature branch.

A. Create a branch for trigonometric functions

```
git checkout -b feat/trig_fns
```

B. Implement `sin`

Add the following import at the top of `calculator.py`

```
import math
```

Then append the following to `calculator.py`

```
def sin(x):
    return math.sin(x)
```

Move `calculator.py` to the staging area and commit.

```
git add calculator.py
git commit -m "feat: add sin"
```

C. Implement `cos`

```
#calculator.py
def cos(a):
    return math.cos(a)
```

Finally, move `calculator.py` to the staging area and commit.

```
git add calculator.py
git commit -m "feat: add cos"
```

While we are implementing the trig functions on the `feat/trig_fns` branch, we receive a bug report. Since our work is isolated on a branch, we can switch back to `main` (or the appropriate release branch), fix the bug without mixing in unfinished trig work, and then merge the fix where needed.

!!! Bug: Float Division !!!!

Someone found the calculator and started using it in their personal project almost immediately. However, they soon discovered a bug: **division uses integer division**, so results are truncated (for example, `5 / 2` returns `2` instead of `2.5`). They need to inform the maintainer that division behaves incorrectly in release `v0.0.1`. GitHub provides a convenient mechanism for doing this: **Issues**.

Fixing the Bug

`main` has moved on from `v0.0.1`, so we don't want to rewind it and disrupt ongoing work. Instead, we create a branch starting at the `v0.0.1` tag, apply the fix there, and cut a new patch release from that branch. After fixing the bug, we tag and release `v0.0.2`. Under Semantic Versioning, this is a **patch** bump because it fixes a bug without introducing breaking changes.

A. Create a `Hotfix` branch from the release tag

```
git checkout -b hotfix/v0.0.1-div v0.0.1
```

B. Implement the fix and commit it

In `calculator.py` edit the definition of the `div` function to match the following:

```
def div(a, b):
    if b == 0:
        return float('inf')
    return a / b
```

```
git add calculator.py
git commit -m "fix: handle division correctly"
```

C. Tag the patched release

```
git tag -a v0.0.2 -m "Hotfix buggy division"
git push -u origin hotfix/v0.0.1-div
git push origin v0.0.2
```

We also need to fix the problem on the current `main` branch so the bug does not carry forward into future versions of the project.

D. Port the fix to `main`

Merge the hotfix branch:

```
git switch main
git merge hotfix/v0.0.1-div
git push
```

Merging

A merge is a Git operation that combines changes from one branch into another. Git offers the following merge strategies:

- **Merge commit** (`git merge --no-ff <branch>`)
- **Fast-forward merge** (`git merge --ff-only <branch>`)
- **Squash merge** (`git merge --squash <branch>`)

For now, we will use a **merge commit**, because it works even when branches have diverged and it clearly shows where a feature branch was merged. We will explore the other options in a later section. Before we do that, we shall switch back to the `feat/trig_fns` branch to complete our implementation of trigonometric functions.

Rebasing

- A rebase 'replays commits' from a base branch on a target branch.
- Instead of creating a merge commit, it creates a sequence of new (but semantically equivalent commits)
- **Rebase then merge** (`git rebase <target>` then `git merge`)

A. Checkout the `feat/trig_fns` branch

```
git checkout feat/trig_fns
```

B. Implement `tan` and commit it

```
#calculator.py
def tan(x):
    return math.tan(x)
```

```
git add calculator.py
git commit -m "feat: add tan"
```

C. Upstream our changes back to the main branch

Now that we are confident the trigonometric functions are implemented correctly, we `merge` the completed work from our feature branch into `main` so the changes become part of the main codebase. First, a note on merging.

```
git checkout main
git merge --no-ff feat/trig_fns
```

This can create a merge conflict because `main` has new commits (the division fix) that were made after we branched off to create `feat/trig_fns`. If both branches changed the same lines or nearby code, Git cannot automatically combine them and will require manual conflict resolution.

D. Resolve merge conflicts

Merge conflicts happen when `git` cannot automatically combine changes because the same lines (or nearby code) were modified in different branches. To resolve them, open the conflicted files, choose what the final code should be, remove the conflict markers, then stage the fixes and complete the merge with a commit. Modern code editors (e.g., VS Code) provide a relatively simple interface for doing this.

```
```

Contributing to Public Repositories

Someone needs logarithm functions for their use case. Rather than wait for the maintainer to implement them, they decide to do it themselves. They create a fork of the calculator project and implement the features they need. In this section, we shall work through the typical workflow for implementing new features, all the way from creating a `fork` to creating a `pull request` , and getting it accepted.

Repository Fork

A repository fork is a copy of a repository (often belonging to another user) under your own GitHub account that you can modify independently and use to propose changes back to the original project (typically via a pull request).

A. Fork the `calculator` Repository on GitHub

To fork a repository on GitHub:

1. Open the repository on GitHub.
2. Click **Fork** (top-right).
3. Choose your account (or an organization) as the destination.

B. Create a feature branch

```
git checkout -b feat/logarithm
git push -u origin feat/logarithm
```

C. Implement the desired feature (logarithms)

Include a general `log(value, base)`.

```
#calculator.py
def log(value, base):
    return math.log(value, base)

def log10(value):
    return math.log(value)
```

D. Commit the changes to `calculator.py`

```


git add calculator.py
git commit -m "feat: add log functions"

git push
```

Pull Request (PR)

A pull request is a notification to the maintainers of a project that you have done work you would like them to incorporate into the project. While Git provides a utility for drafting pull requests (the `git request-pull` command), GitHub provides a more convenient mechanism for creating pull requests.

Drafting a Good Pull Request

A good pull request should tell a clear story: what problem it solves, what changed, and how someone can verify it. Keep the scope focused, and structure the work as a small number of logical commits with descriptive messages so a reviewer can follow the progression. This reduces the amount of strain that reviewing a pull request places on a maintainer, and greatly increases the chances that a pull request gets reviewed in a timely fashion.

E. Create a Pull Request on GitHub

1. Push your branch to GitHub (in your fork or the main repo).
2. Open the repository on GitHub.
3. Click **Pull requests** → **New pull request**.
4. Select the base repo/branch and compare repo/branch.
5. Review the diff to confirm the changes.
6. Add a title and description (what changed, why, how to test).
7. Click **Create pull request**.

Reviewing Pull Requests

It is important to review pull requests carefully. Diligent review helps catch errors before they reach the codebase and ensures incoming code adheres to established standards. When reviewing pull requests, provide actionable feedback and explain the reason behind requested changes. Focus on correctness, readability, and maintainability, and comment on specific lines where possible. If something looks good, say so; if changes are needed, suggest concrete improvements or examples.

A. Give feedback on the PR

There is a mistake in the `log10` function: it assumes the default base is 10, but the current implementation computes the natural logarithm instead. We can leave a comment on the pull request to point out the error and suggest the correct approach. For example:

```
Thanks for the PR — the implementation looks good overall. On
e issue: `log10(x)` currently calls `math.log(x)`, which comp
utes the natural log (base e), not base 10. For example, `log
10(100)` should return `2`, but the current implementation re
turns ~`4.605`.

Could you switch to `math.log10(x)` (or `math.log(x, 10)`) so
it computes the base-10 logarithm as intended.
```

B. Fix the wrongly implemented function

Modifying a PR Under Review

1. Read the review comments and identify what needs to change.
2. Make updates on the **same branch** used for the pull request.
3. Commit the fixes with clear messages (or amend/squash if appropriate).
4. Push the branch again (`git push`), and GitHub will update the PR automatically.
5. Reply to comments explaining what you changed and request another review.

Checkout the `feat/logarithm` branch

```
git checkout "feat/logarithm"
```

Edit `calculator.py` to correctly implement log10.

```
def log10(value):
    return math.log(value, 10)
```

C. Create a commit and push (while still on the same branch)

```
git add calculator.py
git commit -m "fix wrong log10 implementation"
git push
```

D. Merge the PR

Now the PR is ready to be merged. To merge a PR on GitHub:

- Click **Merge pull request** (or choose **Squash and merge** / **Rebase and merge**, depending on the project's preferred strategy).

More on Merge Strategies

- **Merge commit** (`git merge --no-ff <branch>`): creates a new commit that ties the two histories together. Preserves the full branch history and makes it obvious a feature branch was merged.
- **Fast-forward merge** (`git merge --ff-only <branch>`): moves the target branch pointer forward with no new merge commit, but only works if the target branch has no new commits since the branch point.
- **Squash merge** (`git merge --squash <branch>`): combines all commits from the feature branch into one new commit on the target branch. Keeps history linear and concise, but loses individual commit history from the feature branch.
- **Rebase then merge** (`git rebase <target>` then `git merge`): rewrites the feature branch commits onto the latest target branch to create a linear history, then merges (often as a fast-forward). Useful for clean history, but changes commit hashes and should be avoided on shared branches.