`git-intro`

Introduction to version control, collaborative coding, and continuous integration

Dieter Werthmüller

6th November 2022

Outline

Introduction to `git`

`git` platforms and collaborative coding

Reinforcement and advanced topics

Continuous Integration (CI)

Introduction to `git`

At the end of the first module you should know...

- what `git` is and what it is not;
- why you should use `git`;
- how to initiate a `git` repo;
- how to work with `git` locally (`add`, `commit`, `branch`, `merge`, ...);
- how to see changes and to read the log.

Basic rules of code development & the power of CI

1. If you write code, **you are a developer!** (like it or not)
2. Code that is not under **version control** does not exist / is not reproducible
3. Code that is not **properly documented** is not useful
4. Code that has no **tests** is broken

Remember:

- Is it code? $\Rightarrow$ Version Control!
- Document, document, document, . . . , and document some more!
- Test, test, test, . . . , and test again! Ideally with benchmarks.

A lot of work $\Rightarrow$ Continuous Integration!

Version Control: What is it and why should I bother?

Why?	**Track your changes and retain entire history**; Reproducibility; Collaboration & parallel development; Try stuff; . . .
What?	Important with **any** (text) file (do not comment out large junks!)
Watch out	**Version Control** *vs* **Backup** *vs* **Synchronization**
Systems	**Centralized** CVS ('86), SVN ('00), . . . **Distributed** git ('05, Linus), Mercurial (hg '05), . . .
Hostings	GitHub, GitLab, Bitbucket, LaunchPad

git $\neq$ GitHub/GitLab!

Useful Links: Git in general

- The Git Book: git-scm.com/book/en/v2
- Version Control for Scientists: youtu.be/S4uqsbV-gxY
- Good, simple primer: rogerdudler.github.io/git-guide
- GUIs (`gitk`): There are GUIs for any OS, just search the web; e.g., hostinger.com/tutorials/best-git-gui-clients (show GitHub Desk./VS Code/TortoiseGit)
- bash-git-prompt: github.com/magicmonty/bash-git-prompt
- If you screwed up, this might help: sethrobertson.github.io/GitFixUm
- Some visualization sites:
  - dev.to/lydiahallie/cs-visualized-useful-git-commands-37p1
  - onlywei.github.io/explain-git-with-d3
  - marklodato.github.io/visual-git-guide

Licensing

- Why is it important?
  - Putting your code on the web does *NOT* make it open
- Copyleft (e.g., GPL) vs permissive (e.g., Apache/MIT/BSD)
- Software license $\neq$ creative common licenses (e.g., CC-BY)
- **Free as in speech** vs free as in beer vs open-source
  (freedom to run, copy, distribute, study, change and improve the software)
- How to choose a license?
  - choosealicense.com
  - FSF license list
  - Agile Blog: Choose a license; Open-Source wish list
- Free Software Foundation fsf.org (35 years; Linux 29 years)
- Electronic Frontier Foundation eff.org (30 years)
- Creative Commons creativecommons.org (20 years)

Before we get started... tell me about you

- Operating System(s)
- Programming Language(s)
- Editor(s)
- How much knowledge / experience in version control / git / CI

Any other questions before we start?

Configuration

Check it is installed, and version.

```
$ git
$ git --version
```

Configuration

```
$ git config --list # --local / --global
$ git config --global user.name "Your Name"
$ git config --global user.email "your@email.com"
$ git config --list
```

Location of config file

- `~/.gitconfig` / `$HOME\.gitconfig`
- `local` lives in `.git/config`; there is also a system-wide configuration file

First git repository

```
$ mkdir testgit
$ cd testgit
$ git status
$ git init
$ git status
```

⇒ git is very well documented; use `--help` on any command.

`master` vs `main` vs `trunk` vs `...`

git version >= 2.28.0

```
$ git config --global init.defaultBranch main
```

git version < 2.28.0

```
$ git init
$ git checkout -b main # Empty repo
$ git branch -m main # Non-empty repo
```

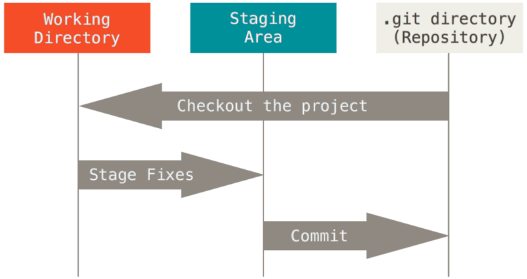
Adding and committing – The git states

```
$ echo "Some text" > myfile.txt
$ git status
$ git add myfile.txt
$ git status
$ git commit -m "First commit"
$ git status
```

Look at

```
$ git rm --cached myfile.txt
$ git commit -am "Message"
```

Three states of git



Source: git-scm.com/book/en/v2/Getting-Started-What-is-Git%3F

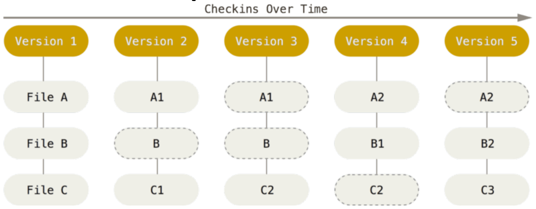
What should be in git, what not, ignored files

- Yes: Text files
- No: Binaries
- *Many* exceptions to this

To ignore files and directories, there is `.gitignore`.

⇒ Look at the `.git`-directory!

Stream of snapshots



Source: git-scm.com/book/en/v2/Getting-Started-What-is-Git%3F

Exercise I: Getting started

1. Create some files.
2. Make changes to files.
3. Commit frequently.
4. Use `git status` to observe what happens.

Wrap the exercise up by tagging the current version!

```
$ git tag -a v0.1 -m "Initial version" # Annotated tag
$ git tag just-to-remember # Lightweight tag
```

Branching and merging

Create branches

```
$ git switch -c featureA
$ git status
```

Switch between branches

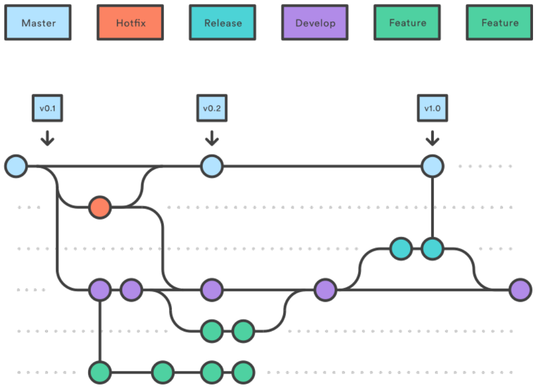
```
$ git switch featureA
```

Merge branche into current one

```
$ git merge featureA
```

Advanced

- Resolving merge conflicts
- Rebasing, squashing, . . .



Source: https://www.vippng.com/maxp/hbmmiRx/

Exercise II: Branching and merging

- Create some branches
- Make changes
- Merge them back to main

If that is too easy...

- Make some conflicting changes in another branch
- Try to resolve the merge conflicts
- (We look at it together afterwards)

`diff` and `log`

`git diff` and `git log` can be used to compare branches, tags, or hashes in general. In the following a few commands, using `diff`, but you can also replace it by `log`.

```
$ git diff branch1..branch2
$ git diff tag1..tag2
$ git diff hash1..hash1
$ git diff branch1..branch2 -- filename
```

Graphical log

```
$ git log --pretty=oneline
$ git log --graph --pretty
$ git log --graph --oneline --decorate --all
```

Here some commands I copied from ma.ttias.be/pretty-git-log-in-one-line:

```
git log --graph --pretty=format:'%Cred%h%Creset -%C(yellow)%d%Creset %s
%Cgreen(%cr) %C(bold blue)<%an>%Creset' --abbrev-commit
```

To not type that every time, you can add a new command `logline`:

```
git config --global alias.logline "log --graph
--pretty=format:'%Cred%h%Creset -%C(yellow)%d%Creset %s %Cgreen(%cr)
%C(bold blue)<%an>%Creset' --abbrev-commit"
```

Then you can run simply `git logline`

Recap – Initializing and basic working

Initializing

`git config`	Get and set repository or global options
`git init`	Create an empty Git repository
`git status`	Show the working tree status
`git <command> --help`	Help for every git command

Adding, committing

`git add`	Add file contents to the index
`git commit`	Record changes to the repository
`git commit --amend`	Amend something to previous commit

Recap – File management and history

File management

`git mv`	Move or rename a files
`git rm`	Remove files
`git restore`	Restore working tree files
`git restore --staged`	Restore staged files

History

`git diff`	Show changes between commits (see extra slide)
`git log`	Show commit logs (see extra slide)
`git grep`	Print lines matching a pattern

Recap – Branches and advance features

Branches

`git branch`	List, create, or delete branches
`git switch`	Switch branches (`-c` flag to create)
`git checkout`	Switch branches or restore files
`git merge`	Join two or more development histories together
`git tag`	Create, list or delete a tag

Wrap-up! Now it should be clear...

- what `git` is and what it is not;
- why you should use `git`;
- how to initiate a `git` repo;
- how to work with `git` locally (`add`, `commit`, `branch`, `merge`, . . .);
- how to see changes and to read the log.

Common question: **How frequently should I commit?**

git platforms and collaborative coding

At the end of the second module you should know...

- the difference between `git` and `git` platforms;
- how to suggest a change to any repo on GitHub/GitLab;
- how to create a project on GitLab, and use it locally;
- how to add an existing repository to GitLab;
- how to work and collaborate with others.

Collaborative coding

We start by looking at GitHub (dominant platform).

- Look at some GitHub repos:
  - github.com/scipy/scipy
  - github.com/simpeg/simpeg

  ⇒ Code; Issues; Pull Requests; . . .

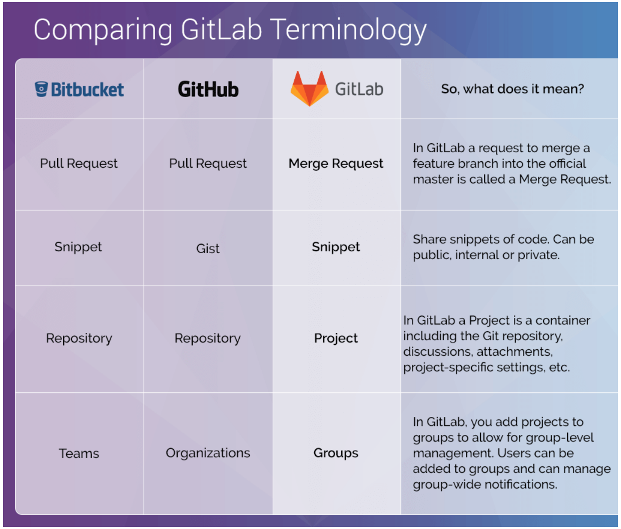
  ⇒ Insights; Releases; Used by; . . .

  ⇒ compare; blame; raw; history; . . .

  ⇒ press the .
- Make an edit on GitHub! github.com/prisae/learning2git

  Make it a habit: learn by giving back to the community!

Vocabulary of different platforms



Comparing GitLab Terminology

Bitbucket	GitHub	GitLab	So, what does it mean?
Pull Request	Pull Request	Merge Request	In GitLab a request to merge a feature branch into the official master is called a Merge Request.
Snippet	Gist	Snippet	Share snippets of code. Can be public, internal or private.
Repository	Repository	Project	In GitLab a Project is a container including the Git repository, discussions, attachments, project-specific settings, etc.
Teams	Organizations	Groups	In GitLab, you add projects to groups to allow for group-level management. Users can be added to groups and can manage group-wide notifications.

Source: about.gitlab.com/blog/2016/01/27/comparing-terms-gitlab-github-bitbucket

- Look at GitLab interface
  - Personal
  - Groups
  - Members
- Most important settings/features

Useful Links: SSH access and documenting

- Setting up password-free access:
  - PASSWORD FREE ACCESS IS CRUCIAL!
  - docs.gitlab.com/ee/user/ssh.html
  - docs.github.com/en/github/authenticating-to-github/keeping-your-account-and-data-secure/about-authentication-to-github
  - docs.github.com/en/github/authenticating-to-github/connecting-to-github-with-ssh/adding-a-new-ssh-key-to-your-github-account

- Markdown: guides.github.com/features/mastering-markdown

- reStructuredText: writethedocs.org/guide/writing/reStructuredText

Exercise III: Create a project in your GitLab; collaborate with others

1. Go to gitlab.tudelft.nl/<your-username>.
2. Hit **New project** ⇒ **Create blank project**.
3. Give it a name and a short description.
4. Choose visibility and hit **Create project**.
5. Add some stuff using the Web IDE.
6. **Search projects of other participants, suggest some changes!**
   `adaniilidis`; `amaalfaraj`; `acuestacano`; `andreashadjigeorgiou`; `akarimzadanzab`; `bareveloobando`; `dzhang2`; `edreveloobando`; `dverschuur`; `eslob`; `jthorbecke`; `mikhaildavydenko`; `sabolhassani`
7. Comment/merge/reject/deal with other's suggestions.
8. At the end, also clone it locally:

```
$ git clone git@gitlab.tudelft.nl:<username>/<projectname>
```

Exercise IV-a: Add an existing project to GitLab

1. Take an existing, local repository.
2. Create a project on your GitLab as above;
   untick the *Initialize repository with a README*.
3. We look together at the *Command line instructions*.
4. ⇒ Push your existing git repository following the instructions.
5. Afterwards:
   - Refresh your browser and check your GitLab repo.
   - Run in your local repository: `git remote -v`

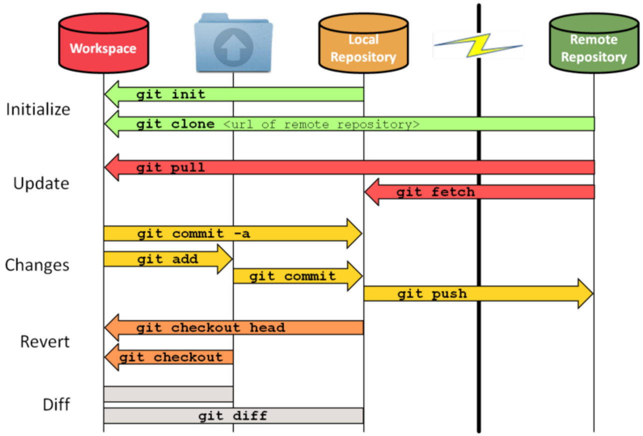
- Note: local repo-folder and remote project folder can have different names!

Terminology – repository (local and remote)

```
$ git clone
$ git pull
$ git fetch
$ git push
$ git remote -v
```

Advanced

- (soft/hard) fork; upstream



Source: https://www.vippng.com/maxp/hbmmhhh/

Exercise IV-b: Working with the local-remote setup

1. Local-to-remote
   - Make some changes in your local repo.
   - Commit them and push them to remote.
2. Remote-to-local
   - Make some changes in the web interface.
   - Fetch the changes to local.
   - Pull the changes to local.
3. Pushing/pulling branches and tags.
   ($ `git push --set-upstream origin <newbranchname>`)
   ($ `git push origin --tags`)

Exercise V: Collaborating with the Research Group

Precursor: Look at branch-setup together.

1. $ `git clone git@gitlab.tudelft.nl:research-group/dummy`
2. Create a branch (from `dev`, *not* from `main`).
3. Make a change.
4. Commit and push to remote.
5. Create a merge request to `dev` in the web interface.

Recap – Working with remotes

Remotes

`git clone`	Clone a repository into a new directory
`git push`	Update remote refs along with associated objects
`git fetch`	Download objects from another repository
`git pull`	Fetch from and integrate with another repository
`git remote`	Manage set of tracked repositories

Wrap-up! Now it should be clear...

- the difference between `git` and `git` platforms;
- how to suggest a change to any repo on GitHub/GitLab;
- how to create a project on GitLab, and use it locally;
- how to add an existing repository to GitLab;
- how to work and collaborate with others.

Reinforcement and advanced topics

Today's program

- Repeat I: walk-through example, using all commands and learning some new
- Repeat II: making a collaborative release on gitlab.tudelft.nl

Exercise VI-a: Walk-through --local

config, init, status, log, diff, add, commit, restore, branch, switch, merge, mv, rm, tag, clone, fetch, pull, push, remote

(`git config`)

Start

```
mkdir sample
cd sample
git init
git switch -c main
echo "My Code" > README.md
git add README.md
git commit -m "Initial"
```

Branch dev

```
git switch -c dev
echo "# My code" > code.py
echo "# My test" > test.py
git add code.py test.py
git commit -am "Add code"
git switch main
echo "A sample repo" >> README.md
git commit -am "More info"
```

Branch featureA

```
git switch -c featureA
echo "New feature" > code2.py
echo "# Changelog" > CHANGELOG.md
echo "- New featureA" >> CHANGELOG.md
git add code2.py CHANGELOG.md
git commit -am "Feature A"
git switch main
```

Merge and check

```
git branch -a
git merge featureA
git branch -d featureA
git status
git log
git diff dev..main
```

A change that we regret and undo

```
echo "bla" >> README.md
git add README.md
git status
git restore --staged README.md
git status
git restore README.md
git status
```

Update dev, keep working

```
git switch dev
git merge main
echo "- Code&Test" >> CHANGELOG.md
git commit -am "Update log"
git switch main
echo "More docs" >> README.md
git commit -am "More docs"
git merge dev
```

Move and remove

```
git mv CHANGELOG.md OLD.md
git commit -am "Rename log"
git rm OLD.md
git commit -am "Delete log"
git status
```

Tag

```
git tag -a v1.0 -m "Initial Release"
```

Notable ones not used

```
# git rebase
# git checkout => switch; restore
```

we didn't look at `.gitignore` here

Exercise VI-b: Walk-through --remote

config, init, status, log, diff, add, commit, restore, branch, switch, merge, mv, rm, tag, clone, fetch, pull, push, remote

On your GitLab or GitHub create an *empty* repo (no README)

```
git remote add origin git@gitlab.tudelft.nl:dieterwerthmul/sample.git
git push -u origin --all
git push -u origin --tags
```

Create a 2nd folder of the same

```
cd ..
git clone git@gitlab.tudelft.nl:dieterwerthmul/sample.git twin
ls
```

Change in «twin»

```
cd twin
git remote -v
echo "Started using remote" >> README.md
git commit -am "Info remote info"
git status
git push
```

Update «sample»

```
cd ../sample
git fetch
git status
git pull
git status
cat README.md
```

Push a new branch

```
git switch -c new
echo "Yet another file" > config.cfg
git add config.cfg
git commit -am "Add config file"
git push --set-upstream origin new
```

«add» and «set-upstream» only first time

```
echo "More config" >> config.cfg
git commit -am "More config"
git push
```

Comments

- Fetching/pushing new/deleted branches/tags
- All these are probably 98 % of `git`

If something went awry:

- Internet & Colleagues
- Duplicate & diff
- Clone again & diff

Exercise VII: Creating release `v2.0` for `git2code`

We want to create v2.0 of our code. For this, everyone has to add his name to the contributor file, and add his file in the source folder. After merging all into dev, the owner can merge to main and tag the release.

```
$ git clone git@gitlab.tudelft.nl:research-group/git2code
```

1. Checkout `dev` (is default branch)
2. Checkout a branch with your name
3. Add your name to `AUTHORS.md`
4. Add a file `yourname.xyz` with any content in the directory `src/`
5. Push it to remote, and create a merge request (to `dev`!)
6. Try to merge each others merge requests into the `dev` branch
7. In the end, we merge it to `main` and tag a release

Miscellaneous git

- git is very actively developed: github.com/git/git/graphs/contributors
- git help online (nice to read): git-scm.com/docs

```
$ git config --global core.editor vim
```

(nano, vim, nvim, emacs, subl -n -w, atom --wait, code --wait)

Not shown

`git fetch upstream`	Keep a forked repo in sync
`git cherry-pick`	Apply a particular commit
`git rebase -i`	Interactive rebasing
`git stash`	Stash dirty working directory away
`git bisect`	To search/find the commit that introduced a bug

Continuous Integration (CI)

Today's program

- What is it, when is it used
- «*Show and tell*» on my codes `empymod` & `emg3d`
- Make a release of `emg3d`

Continuous Integration (CI) – What/Why

In its original sense

The practices of collaborating and *merging frequently* to a shared version of the code (see en.wikipedia.org/wiki/Continuous_integration).

- This is a lot of work.
- Automate the test-review-QC-build-deploy workflow as much as possible.
- Goes well with *Version control*; Test-driven development; Release early, release often philosophy; and other ideas.

Continuous Integration (CI) – Advantages

- No last minute chaos on «integration day»!
- Helps to produce more modular code (refactor; less complexity).
- Reduces maintainer load when accepting contributions.
- Much more likely to refactor.

It is very easy to write complex code. It is hard to write simple code.

CI & Automation on the example of `empymod`

- Commit to GitHub
- GitHub Actions - run all tests
  - Linux, MacOS, Windows
  - Different Python versions
  - Report to coveralls (code coverage)
  - Check all hyperlinks
  - Deploy to PyPi and conda-forge (*if tag*)
- Codacy (code quality)
- ReadTheDocs and Gallery
- Mint DOI at Zenodo (*if tag*)
- Benchmarks (`asv`, *manual*)

All of them are *free* (for open-source projects)

CI: Fast/automatic deployment

We create a release of emg3d.