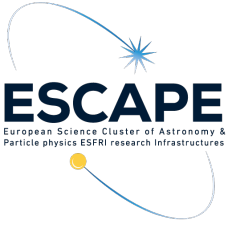
Version Control using `git`

Maximilian Nöthe

Astroparticle Physics, TU Dortmund



Summer School – 2022-06-21

Overview

whoami

What is version control and why do we need it?

Git

Git Hosting Providers

Advanced Git

whoami

- maxnoe
- PostDoc @ TU Dortmund
- PhD in astroparticle physics, also @ TU Dortmund
- Gamma-ray astronomy with Imaging Atmospheric Cherenkov Telescopes (CTA, FACT, MAGIC)
- Data Analysis, Statistics, Machine Learning, Software Development
- , C++, LaTeX, (neo)vim, zsh, astropy, matplotlib, ...
- FOSS, Open Science, Best Practices

Warning

Copying commands or code from PDF files is **dangerous**

Copy from the example files in the repository or type by hand.

Typing by hand is best for learning.

What is version control and why do we need it?

What is Version Control?

- → Version Control tracks changes of a (collection of) document(s)
- → This can basically be anything:
  - → software
  - → legal documents
  - → documentation
  - → scientific paper
  - → images
  - → ...
- → We will call a snapshot of such a collection a “revision”.
- → Revisions are the complete history of our projects

Why Use Version Control?

- → Allows us to go back to arbitrary revisions
- → Shows differences between revisions
- → Enables collaborative working
- → Acts as backup

Why Use Version Control?

Most Version Control Systems (VCS) make answering the following questions easy:

What? What changed from revision *A* to revision *B*?

Who? Who made a change? Who contributed?

Why? VCS usually encourage or even force adding explanations to changes.

When? In which revision was a bug introduced or fixed?

Why Use Version Control?

Most Version Control Systems (VCS) make answering the following questions easy:

What? What changed from revision *A* to revision *B*?

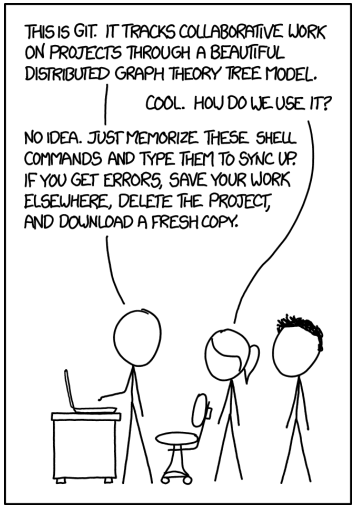
Who? Who made a change? Who contributed?

Why? VCS usually encourage or even force adding explanations to changes.

When? In which revision was a bug introduced or fixed?

Version Control is a basic requirement for reproducible science


git



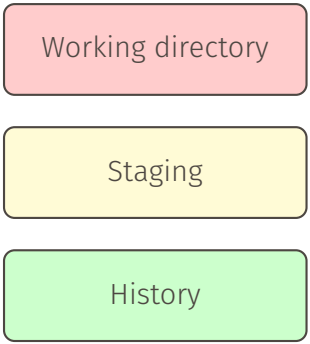
R. Munroe, xkcd.com/1597

- → Created by Linus Torvalds in 2005 for the Linux Kernel
- → Most widely used VCS in FOSS
- → Distributed, allows offline usage
- → Much better branching model than precursors like SVN

The Git Repository

Central Concept: Repository

- → `git init` creates a git repository in the current working directory
- → All git data is stored in the `.git` directory.
- → Git has three different areas, data can reside in:

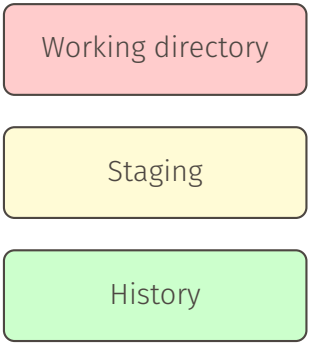


What actually is on disk in the current working directory.

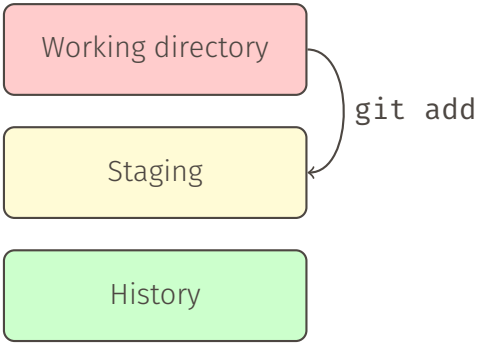
Changes that are saved to go into the next commit.

The history of the project. All changes ever made. A *Directed Acyclic Graph* of commits.

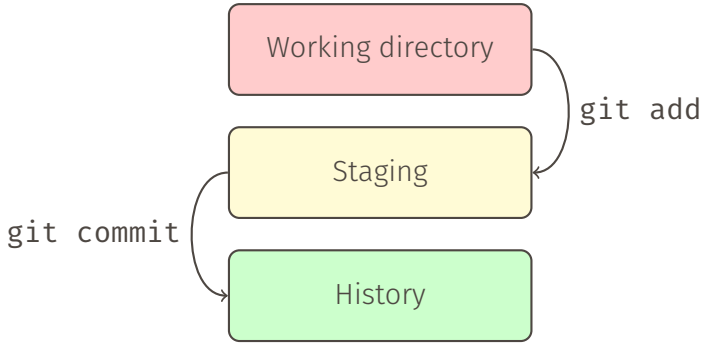
Central Concept: Repository



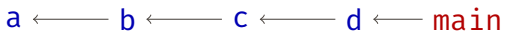
Central Concept: Repository



Central Concept: Repository

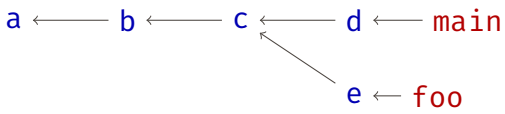


History



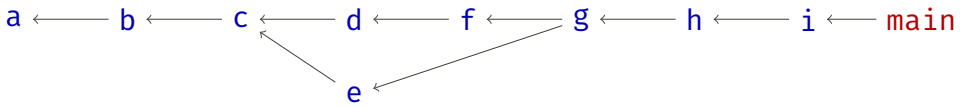
- Commit: State/Content at a given time
  - Contains a commit message to describe the changes
  - Always points to its parent(s)
  - Is identified by a hash of the content, message, author, parent(s), time

History



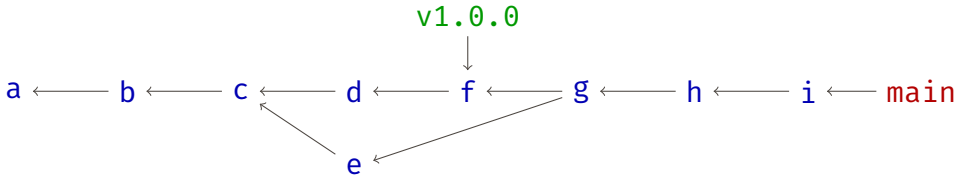
- → Commit: State/Content at a given time
  - → Contains a commit message to describe the changes
  - → Always points to its parent(s)
  - → Is identified by a hash of the content, message, author, parent(s), time
- → Branch: A named pointer to a commit
  - → Development branches
  - → Default branch: `master` or `main`
  - → Moves to the next child if a commit is added

History



- → Commit: State/Content at a given time
  - → Contains a commit message to describe the changes
  - → Always points to its parent(s)
  - → Is identified by a hash of the content, message, author, parent(s), time
- → Branch: A named pointer to a commit
  - → Development branches
  - → Default branch: `master` or `main`
  - → Moves to the next child if a commit is added

History



- → Commit: State/Content at a given time
  - → Contains a commit message to describe the changes
  - → Always points to its parent(s)
  - → Is identified by a hash of the content, message, author, parent(s), time
- → Branch: A named pointer to a commit
  - → Development branches
  - → Default branch: `master` or `main`
  - → Moves to the next child if a commit is added
- → Tag: Fixed, named pointer to a commit
  - → For important revisions, e.g. release versions or version used for a certain paper

Initial Setup

→ The first thing you should do after installing git is tell it who you are:

⚠ Fill your own information! ⚠

```
$ git config --global user.name "Maximilian Nöthe"
$ git config --global user.email "maximilian.noethe@tu-dortmund.de"
```

→ This information is required as it will be added to each commit you make

→ Hosting providers map commits to users by the commit email

Creating or Cloning a Repository

→ Create a new git repository in the current directory

```
$ git init
```

→ Clone (download) a repository from a server, e. g. GitHub

```
$ git clone <url>
```

→ Remove all traces of git from a repository

```
$ rm -rf .git
```

⚠ This is not recoverable locally ⚠

git status

- Shows current branch and new, modified, added files
- Make a habit of calling `git status` often
- Concise version with `git status -s`

git add, git mv, git rm, git reset

`git add <file> …`	Add files to the staging
`git mv`	like `mv`, stages automatically
`git rm`	like `rm`, stages automatically
`git reset <file>`	Removes changes/files from the staging area

Creating a new commit

→ Create a new commit from the changes in the staging area.
This will open editor for the commit message, most likely *vim*

```
$ git commit
```

→ You can also give the message directly on the command line:

```
$ git commit -m "Fix critical bug in flight control system"
```

→ If you are not familiar with vim, you might want to change the editor.
The exact settings depend on the editor, to use VS Code or nano:

```
$ git config --global core.editor "code --wait"
$ git config --global core.editor nano
```

What is a good commit?

- Commits should be small, logical units
- “Commit early, commit often”
- It is common convention to formulate the commit subject as imperative:

```
Fix sub-optimal value of foo
```

- Style guide for commit messages:

```
Subject line, short description, best < 60 characters

After one empty line, a detailed description of the commit.
Explain **why** the change was necessary and give details.
* Use bullet point lists for stuff
* Link releveant issues, #2

Give credit to other people when working together:
Co-authored-by: Thomas Vuillaume <thomas.vuillaume@lapp.in2p3.fr>
```

- Great example on Github

Accessing the log

- Shows author information, date, hash, message

```
$ git log
```

- Supports ranges:

```
$ git log <a>..<b>
```

(All commits reachable from `<b>` but not from `<a>`)

- More concise log, helpful for use with `grep`

```
$ git log --oneline
```

Accessing the log

→ Showing branches in “ASCII art”:

```
$ git log --all --decorate --graph
```

→ Show (number of) commits by author

```
$ git shortlog [-sne]
```

→ Get contributors between two versions

```
$ git shortlog [-sne] v4.0..v5.0
```

git diff

Shows the differences between versions.

- Show diff between the current working directory and the staging area:

```
$ git diff
```

- Show diff between the staging area and the last commit:

```
$ git diff --staged
```

  This is *very* useful, since this will become be the next commit.
  Run before `git commit` and check for mistakes.

- Show diff between two files, commits, branches, ...

```
$ git diff <arg1> <arg2>
```

Loading commits / restoring files

- Load a certain commit from the history into the CWD (check with `git log`)

```
$ git checkout <commit>
```

- Restore a file to the version from the last commit (throwing any changes away)

```
git restore <file>
```

- Restore a file to a version from a specific commit or branch

```
git restore --source=<source> <file>
```

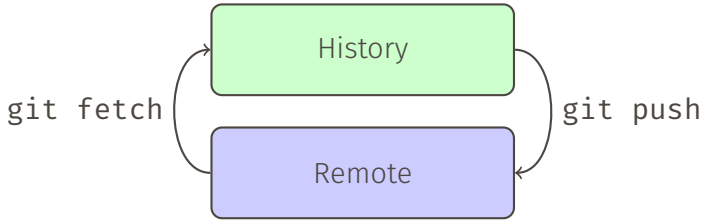
Note on older versions of git

- In older versions of git, `git checkout` had many different tasks, which is confusing
  - Loading commits into the working directory
  - Restoring files to another version
  - Switching branches
- Recent versions of git added `git restore` for restoring files
- Iff you have an old version of git and cannot upgrade easily, use

```
$ git checkout <source> -- <file>
```

Remotes

Remotes are central places, e. g. servers, where repositories can be saved and which can be used to synchronize between different clients.



Syncing with remotes

- Download (clone) a repository from a remote

```
$ git clone <url>
```

- Add a new remote to the repository

```
$ git remote add <name> <url>
```

  - The default remote is called “origin” by convention
  - When you clone a repository, the “origin” remote will already be setup

- Download changes from the default or a specific remote:

```
$ git fetch [remote]
```

- Download changes from the default remote and merge them into your local branch

```
$ git pull
```

Syncing with remotes

→ Upload your current branch to its default remote

```
$ git push
```

→ Set the default remote for branch and push (needed when pushing a new branch for the first time)

```
$ git push -u origin branch
```

Typical single-branch workflow

0. Get / create / update the repository

If new Create or clone repository : `git init`, `git clone <url>`

If exists `git pull`

1. Work

1.1 Edit files and build/test

1.2 Add changes to the next commit: `git add`

1.3 Save added changes in the history as *commit*: `git commit`

2. Download commits that happend in the meantime: `git pull`

3. Upload your own: `git push`

4. Go back to 1

Working using multiple branches – GitHub Workflow

There are multiple models of working together with git using branches

Simplest and most popular: “GitHub-Workflow”

- → Nobody directly commits into the main branch
- → A new branch is created for each new feature / change / bug-fix
- → Branches should be rather short-lived
- → Merge into the main branch as soon as possible, then delete the feature branch
- → The main branch should always contain a working version

Note: this workflow is only named after GitHub, you can and should also use it on GitLab or whatever other platform you are using.

Branches

→ Create a new branch pointing to the current commit

```
$ git branch <name>
```

→ Switch to branch `<name>`

```
$ git switch <name>
```

→ Create a new branch and change to it

```
$ git switch -c <name>
```

→ Merge the changes of branch `<other>` into the current branch

```
$ git merge <other>
```

Note: As with `git restore`, `git switch` is a relatively new addition to git. Use `git checkout [-b] <branch>` for older versions

Default branch name

- Recently, a political correctness discussion happened around master/slave terminology in tech
- Many software projects have since replaced those terms
- While not directly related to master/slave, git also enabled to change the name of the default branch
- GitHub and GitLab now use `main` as default for new repositories
- Currently, you will encounter both `master` and `main`
- Use `main` locally (when using `git init`)

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

Beware: Merge conflicts

Happens when git can't merge automatically, e. g. two people edited the same line.

1. Open the files with conflicts
2. Find the lines with conflicts and resolve by manually editing them

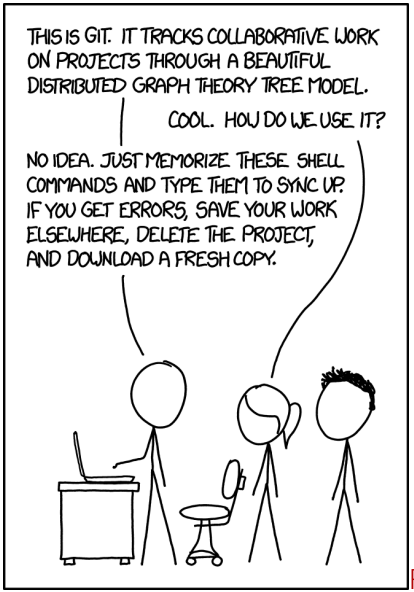
```
<<<<<<< HEAD
foo
||||||| merged common ancestors
bar
=======
baz
>>>>>>> Commit-Message
```

3. Commit merged changes:

```
$ git add …
$ git commit
```

useful: `git config --global merge.conflictstyle diff3`

Relevant XKCD



R. Munroe, xkcd.com/1597

.gitignore

- Many files or filetypes should not be put under version control
  - Compilation results
  - Files reproducibly created by scripts
  - Config files containing credentials
  - ...
- Solution: `.gitignore` in the base of a repository
- One file or glob pattern per line for files that git should ignore
- Hosting providers have default `.gitignore` for most programming languages: github.com/github/gitignore

Example `.gitignore`

```
build/
*.so
__pycache__/
```

Global .gitignore

- Some files should be ignored globally for all repositories of a user
  - OS specific files
  - Editor / tool specific files
  - ...
- `git config --global core.excludesfile $HOME/.gitignore`

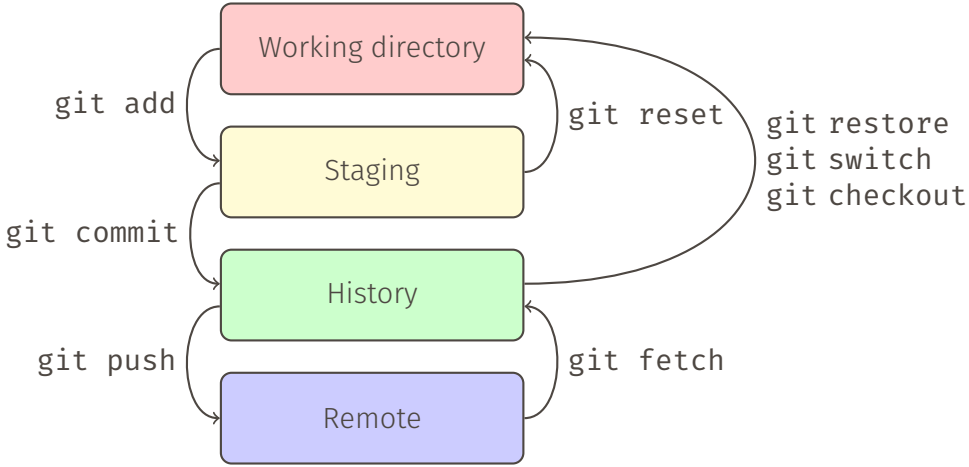
$HOME/.gitignore

```
__MACOSX     # weird mac directory
.DS_STORE    # mac finder metadata file
*.swp        # vim backup files
*~           # nano / gedit / emacs backup files
desktop.ini # Windows explorer metadata file
```

Some Limitations of git

- Git is only designed to work well with text files
- In general, git does not handle binary files well, this includes:
  - Images
  - Document formats like odp, docx, pdf, ...
  - Binary data files
- Git cannot efficiently store these files in the history
- Repository size will grow quickly when often changing binary files
- Although being text (JSON), jupyter notebooks are also hard to use well with git
  - Graphics embedded into the notebook
  - Merge conflict resolution hard to get right
  - Diffs not really telling what changed
  - Tooling to improve this: nbdime, reviewnb
  - Recommendation: only commit notebooks after running “Clear all outputs”

Most common commands




```
git pull = git fetch && git merge <default remote>/<branch>
```

Questions?

Git Hosting Providers

Git Hosting Providers

- → Several Providers and self-hosted server solutions are available
- → Usually provide much more than just hosting the repositories
  - → Issue tracking
  - → Code review using pull requests
  - → Wiki
  - → Project Management, e.g. Canban boards
  - → Continuous integration
  - → Releases

Git Hosting Providers

GitHub

- Largest Hoster
- Many Open Source Projects, e.g. Python
- Unlimited private repositories
- Free CI service for public repositories
- Github Pro free for students / teachers / researchers

GitLab

- open-source community edition
- paid enterprise edition with more features
- unlimited private repositories
- Self hosted or as service at gitlab.com

Gitea

- Lightweight solution for self-hosting
- Repositories, Issues, Pull Requests, Wiki, etc.

Login for Remote Servers in the CLI

Git can communicate using two ways with a remote:

HTTPS
- → Works for cloning of public repositories out of the box and for GitLab
- → As of 2021, Github requires a *token*, not your password, for CLI login
- → Use git credential helper to have to enter credentials every time:
  `https://git-scm.com/book/en/v2/Git-Tools-Credential-Storage`

SSH
- → Using private/public keys.
- → Keys can be protected with a password
- → Using an ssh-agent, you only need to decrypt a key once per session.
- → Many systems come with ssh-agent by default

Using ssh

- → To use ssh, you need to add at least one public key to your profile.
- → It is considered best practice to use unique keys per machine and service

1. Create the key using: (choose a new password when asked)

Works in Powershell (Windows) and Unix Systems

```
$ ssh-keygen -t ed25519 -C "GitHub Key for <username> at <machine>" -f
 ↪  "$HOME/.ssh/id_ed25519.github"
```

2. Copy the content of the public key, on unix with `cat ~/.ssh/id_rsa.github.pub` or by using a text editor to open the file
3. Add the public key to your profile
4. Use the ssh URL for cloning. To change the URL of an already cloned repository, do

```
$ git remote set-url origin git@github.com:escape2020/school2022
```

Pull / Merge Requests

- Pull Requests (GitHub) / Merge Requests (GitLab) are a feature on top of git provided by several platforms
- Used to propose changes by pushing a new branch and then requesting it to be merged into the main branch
- Usually, projects using the GitHub Workflow only allow changes to the main branch via Pull Requests
- Pull Requests are used for Code Review, project maintainers and co-developers can look at your code and ask for changes
- Usually, a Continuous Integration (CI) system runs checks for the changes proposed in a Pull Request

Code Reviews

- → Code reviews are among the most essential parts of software development
- → Similar to the peer-review process in science
- → Get feedback and advice for improvements
- → Prevent easy-to-find mistakes
- → Ensure quality, performance, documentation, clarity of the software
- → Developers can learn from each other immensely during code reviews
- → You should require code reviews for pull requests

How should you review code?

- → Automatize as much as possible before the actual human review
  - → Static code checks
  - → Unit tests / CI
  - → Coverage
  - → Code style checks
- → Focus on (in order):
  - ✅ Are enough unit tests there?
  - ✅ Are code and tests clear / explained in comments / following best practices?
  - ✅ Any obvious performance improvements?[1]
  - ✅ Is the code documented?
- → Stay friendly but be concise

[1] "Premature optimization is the root of all evil" – Donald Knuth

Forking

- Using git and hosting providers, it's easy to contribute to projects you do not have write access to.
- This is arguably the most important reason for git's success.
- Forking means to create a copy of the main repository in your namespace, e.g. `http://github.com/matplotlib/matplotlib` to `http://github.com/maxnoe/matplotlib`
- You can then make changes and create a pull request in the main repository!
- To keep your fork up to date, you should add both your fork and the main repo as remotes.

Forks

We'll use the school repository for this example

- Click the "Fork" button on GitHub
- Clone your fork

```
$ git clone git@github.com:maxnoe/school2022
```

- Add the main repository as second remote. The name "upstream" is convention.

```
$ git remote add upstream git@github.com:escape2020/school2022
```

- Download content also from upstream

```
$ git fetch upstream
```

Making a Pull Request from a forked Repository

- When starting the new branch, make sure to start from the up-to-date upstream main/master:

```
$ git fetch upstream
$ git switch -c new_branch upstream/master
```

- Make changes and commit
- When pushing the branch, specify your fork (origin):

```
$ git push -u origin new_branch
```

- Go to GitHub or click on the link in the push message to open the Pull Request

Issue Tracking

- Issue Trackers are an important part of every software project
  - Report bugs
  - Feature requests
  - Project planning
  - Ask for help
- Issues can be linked to commits and pull requests

Commit Integration with Issue Tracking

Start working on fixing a bug, that was documented in issue 42.

```
$ git checkout -b fix_42

... do stuff to fix bug ...

$ git add src/foo.cxx
$ git commit -m "Fix segmentation fault when doing stuff, fixes #42"
$ git push -u origin fix_42
```

If this commit get's merged into master, issue 42 will automatically be closed.

Continuous Integration

- → Strictly interpreted, continous integration means integrating current work “often” into the main version
- → Usually, this means running automated builds and checks on a dedicated server
- → Ideally, these are run for each push event
- → For git projects, checks for pull requests should run on the merged result, not the branch itself
- → You should require passing CI system for Pull Requests

Common Features of CI systems

- → Build your application / library
- → Run the test suite
- → Do that for multiple OSes and software / compiler versions
- → Build documentation and packages
- → Upload and publish results / build products

Example python workflow

`https://github.com/maxnoe/pyfibonacci/blob/main/.github/workflows/ci.yml`

More details during the testing lecture.

Advanced Git

Partial Adding

- Commits should be small, logically contained units
- Sometimes, we implement multiple things in one go
- Go through all changes interactively, select what we want to add with

```
$ git add -p
```

Changing the git history (aka the danger zone)

Disclaimers

- → The main/main branch's history should only be modified under severe circumstances
  - → Sensitive data in the history[2]
  - → Large files in the history that need to be removed
- → Not-yet-pushed commits can be freely modified
- → Feature branches can usually be modified
- → Most large projects will even ask you to cleanup the history of your Pull Request to have a "nice" history
- → Modifying already-pushed commits requires pushing with the `--force` option
- → The main/main branch should be protected against force pushes (Github/Gitlab settings)

[2] Under most circumstances I wouldn't recomend to change the history. Change the passwords.

Fixing the last commit

- Just changing the last commit is one of the most common use cases
  - Fix a typo in the commit message
  - Add a forgotten file
  - Remove an accidentally commit file
- Make and add the changes you want to include / fix in the last commit
- Execute

```
$ git commit --amend
```

  - Adds the current staging area to the last commit (optional)
  - Opens the editor for editing the commit message
  - Overwrites the last commit (will change the hash)

Rebase - rewriting the git history

Rebase is a very powerful tool to rewrite the git history.

It can

- → Change commit order
- → Drop / edit single commits
- → Merge multiple commits into one

Merge vs. Rebase

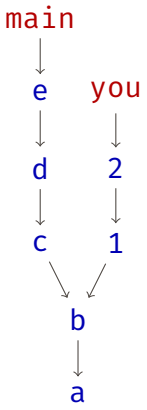
- Default behaviour of `git pull` is equivalent to
  `git fetch && git merge <remote>/<branch>`
- This results in a non-linear history with many merge conflicts like “Merging remote tracking branch...”
- `git pull --rebase` is equivalent to
  `git fetch && git rebase <remote>/<branch>`
- It makes the history equal to the remote history, and then tries to apply the local commits in order

Can be made the default with `git config --global pull.rebase true`

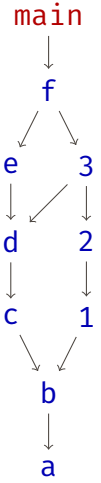
Changes how conflicts are resolved: Instead of creating a single merge commit that contains the fixes to make the two parents compatible, each commit that is rebased is adapted so the conflicts never existed.

git pull --rebase vs. git pull --merge

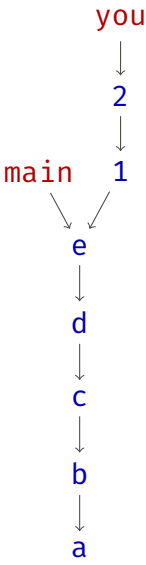
Situation at start



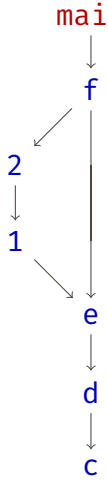
With merging



After rebase



After merging the rebased branch



Interactive Rebase

- → Very powerful tool to change commits
- → Joining / dropping / reordering / changing commits

```
$ git rebase -i <target commit>
```

Submodules

- Git discourages mono-repositories with many projects or just adding other projects to a repository
- Useful for
  - external source dependencies (e. g. Google Test for C++ projects)
  - meta-repositories joining multiple repositores at specific versions
- Submodules add a reference to another repository at a certain commit:

```
$ git submodule add <url> <path>
```

- Cloning does not include submodules by default, needs

```
$ git clone <url> --recursive
```

- Update submodules (e. g. if changed on the remote)

```
$ git submodule update --init --recursive
```

Questions?