

SIB  
Swiss Institute of  
Bioinformatics

[www.sib.swiss](http://www.sib.swiss)

# Version control with Git - first steps

Robin Engler

Vassilios Ioannidis

Lausanne, 28-29 Oct 2025

## Introduction to Git: course outline

- **Introduction** to Version Control Systems and Git.
- **Git basics:** creating a Git repository, making commits, and displaying a repository's history.
- **Git concepts:** commits, the HEAD pointer and the Git index.
- **Git branches:** introduction to branched workflows and collaborative workflow examples.
- **Branch management:** merge, rebase and cherry-pick.
- **Retrieving data from the Git database:** git checkout.
- **Working with remotes:** collaborating with Git and GitHub/GitLab.
- **GitHub/GitLab:** an overview.

## Course resources

**Course home page:** Slides, exercises, exercise solutions, command summary (cheat sheet), setting-up your environment guide.

<https://github.com/sib-swiss/git-training>

**Shared online doc:** Register for the collaborative exercises (and optionally for exam), ask questions. Link sent via email before the course.

**Questions:** feel free to interrupt at anytime to ask questions, or use the shared online document.

# Course slides

- 3 categories of slides:

**Regular slide  
[Red]**

Slide covered in detail  
during the course.

**Supplementary  
material  
[Blue]**

Material available for your interest, to read on your own.  
Not formally covered in the course.  
We are of course happy to discuss it with you if you have questions.

**Reminder slide  
[Green]**

Material we assume you know.  
Covered quickly during the course.

**GitHub-specific  
[Purple]**

**GitLab-specific  
[orange]**

Some slides are specific to **GitHub** or **GitLab**.

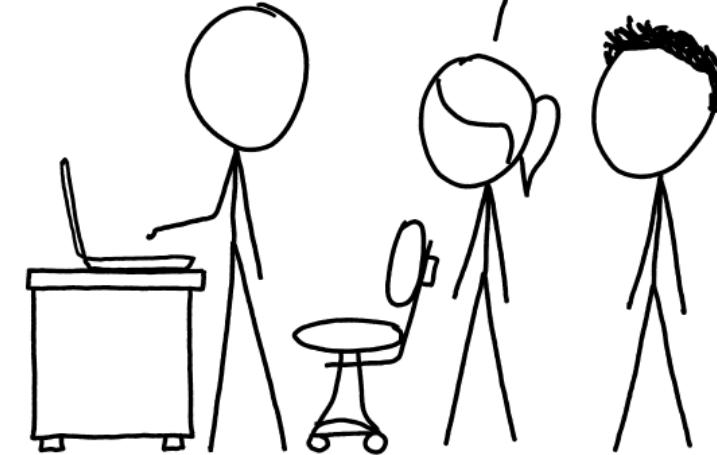
## Learning objectives

- Learn the concepts behind Git.
- Understand when and why to use each command.
- Collaborative workflows using GitHub/GitLab.
- Learn to re-write history (day 2).

THIS IS GIT. IT TRACKS COLLABORATIVE WORK ON PROJECTS THROUGH A BEAUTIFUL DISTRIBUTED GRAPH THEORY TREE MODEL.

COOL. HOW DO WE USE IT?

NO IDEA. JUST MEMORIZIZE THESE SHELL COMMANDS AND TYPE THEM TO SYNC UP. IF YOU GET ERRORS, SAVE YOUR WORK ELSEWHERE, DELETE THE PROJECT, AND DOWNLOAD A FRESH COPY.



## Command line vs. graphical interface (GUI)

- This course focuses exclusively on **Git concepts** and **command line** usage.
- Many GUI (graphical user interface) software are available for Git, often integrated with code or text editors (e.g. Rstudio, Visual Studio Code, PyCharm, ...).  
It will be easy for you to start using them (if you wish to) once you know the command line usage and the concepts of Git.

# version control

a (very) brief introduction

## Why use version control ?

**Version control systems** (VCS), sometimes also referred to as *source control/code managers* (SCM), are software designed to:

- **Keep a record of changes** made to (mostly) text-based content by recording specific states of a repository's content.
- **Associate metadata to changes**, such as author, date, description, tags (e.g. version).
- **Share files** among several people and **enable collaborative work**.
- **Backup strategy**:
  - Repositories under VCS can typically be mirrored to more than one location.
  - The database allows to retrieve older versions of a document: if you delete something and end-up regretting it, the VCS can restore past content for you.
- In the case of Git, entire ecosystems such as GitHub or GitLab have emerged to offer **additional functionality**:
  - **Distribute software and documentation**.
  - **Run automated pipelines** for code testing and deployment (CI/CD).
  - **Project management features** (e.g. issue tracking, continuous integration).

## A brief history of Git

- First release in 2005.
- Initially written by Linus Torvald (who also wrote the first Linux kernel in his spare time...).
- Created to support the development of the Linux kernel code (> 20 million lines of code).

The first commit of Git's own repository by Linus Torvalds in 2005.

```
commit e83c5163316f89bfbde7d9ab23ca2e25604af29
Author: Linus Torvalds <torvalds@ppc970.osdl.org>
Date: Thu Apr 7 15:13:13 2005 -0700
```

Initial revision of "git", the information manager  
from hell

## (some of) The principles that guided the development of Git

Linus wasn't satisfied with existing version control software, so he wrote his own...

He had the following objectives (among others) in mind:

- **Distributed development:** allow parallel, asynchronous work in independent repositories that do not require constant synchronization with a central database. **Each local Git repo is a full copy of the project** so users can work independently and offline.
- **Maintain integrity and trust:** since Git is a distributed VCS, maintaining integrity and trust between the different copies of a repositories is essential. **Git uses a blockchain-like approach to uniquely identify each change to a repository**, making it impossible to modify the history of a Git repo without other people noticing it.
- **Enforce documentation:** in Git, **each change to a repo must have an associated message**. This forces users to document their changes.
- **Easy branching/merging:** Git makes it **easy to create new branches** (i.e. lines of development) in a project. This encourages good working practices.
- **Free and open source:** users have the freedom to run, copy, distribute, study, change and improve the software.

## Part I

# Git basics

Working principle, definitions, and  
making your first commit

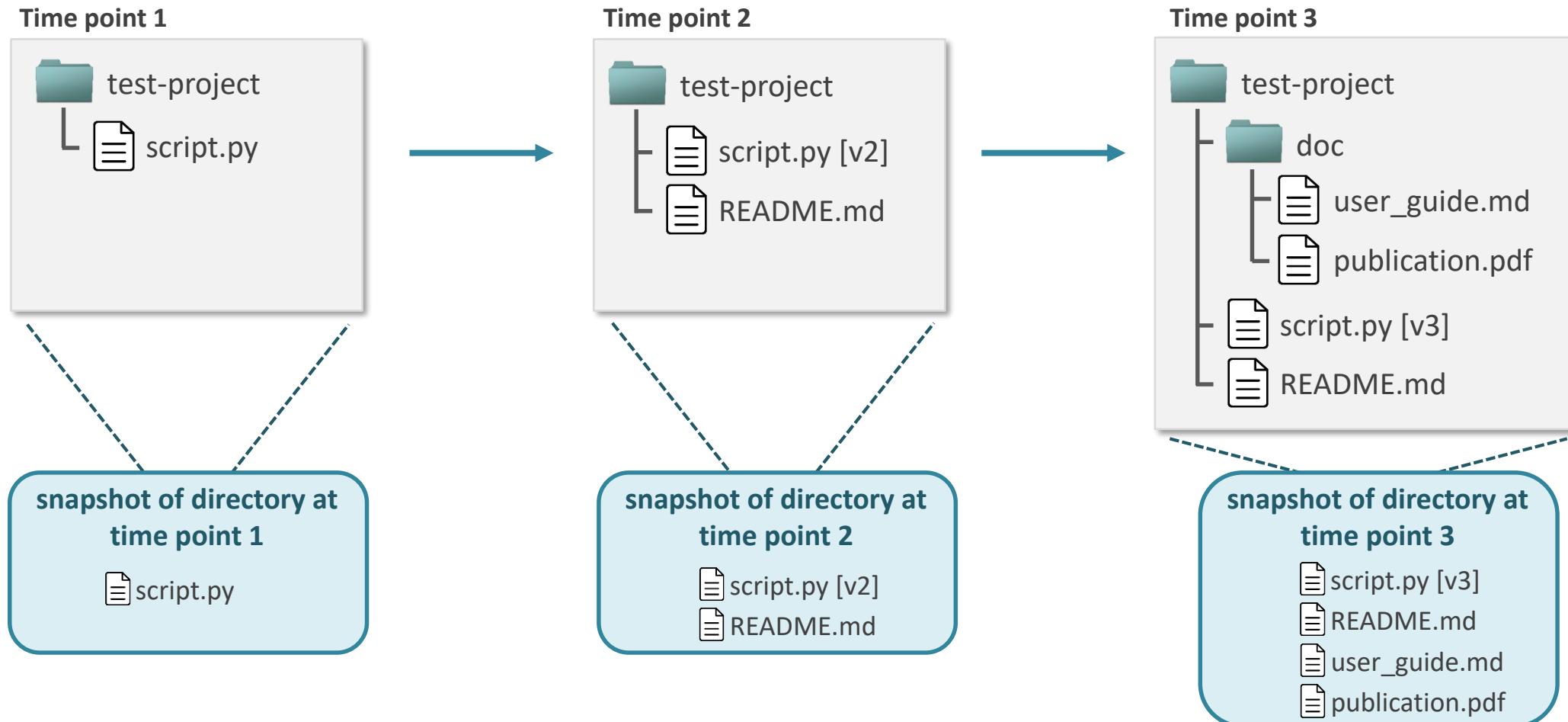
# Git working principles and definitions

## Basic principle of Git

Our objective: record the changes made to the content of a directory on our local machine.

How we proceed:

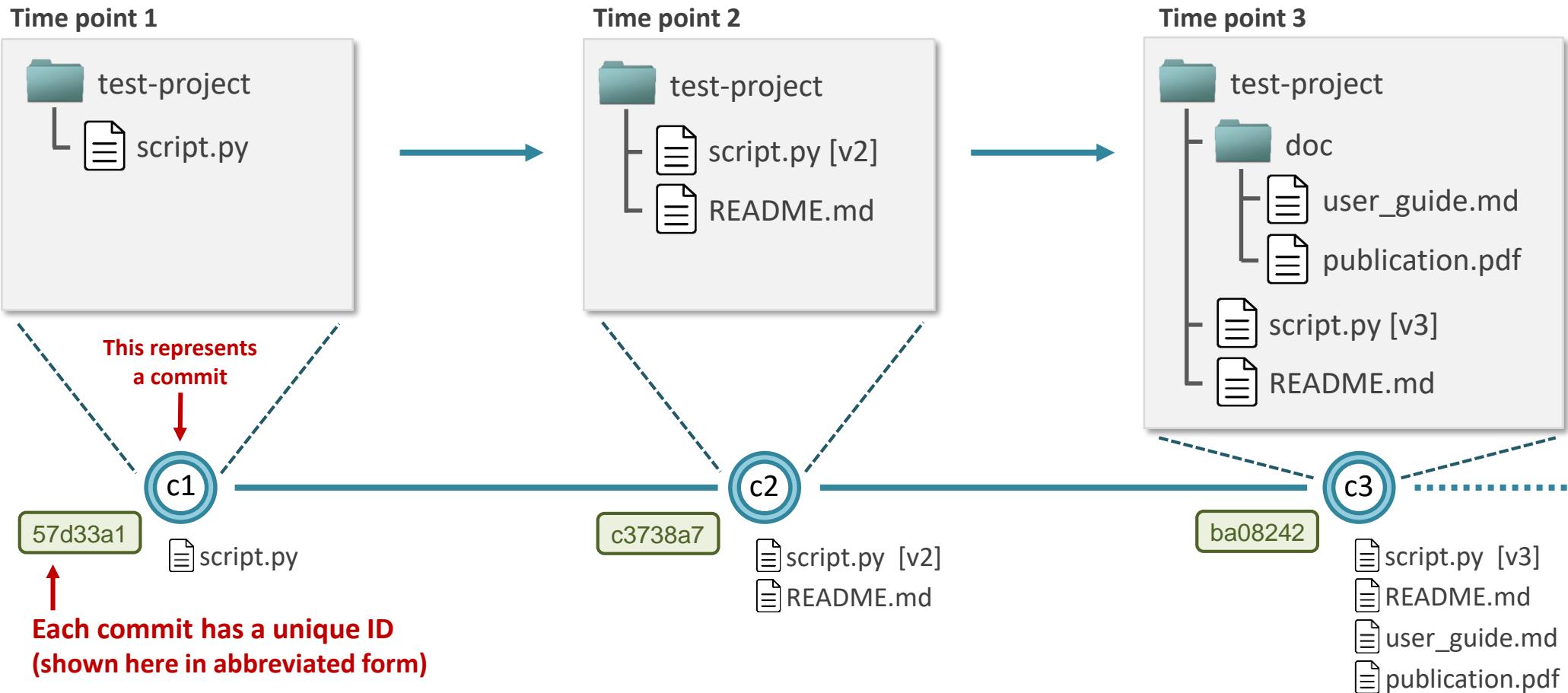
- Take **snapshots** (current content of files) at user defined time points – they are not taken automatically.
- Keep track of the order of snapshots (the relation between them) so their history can be recreated.
- Associate **metadata** with each snapshot: who made it, when, description, ...



Git can track any types of files (text or binary), but is optimized to work with not-too-large text files.

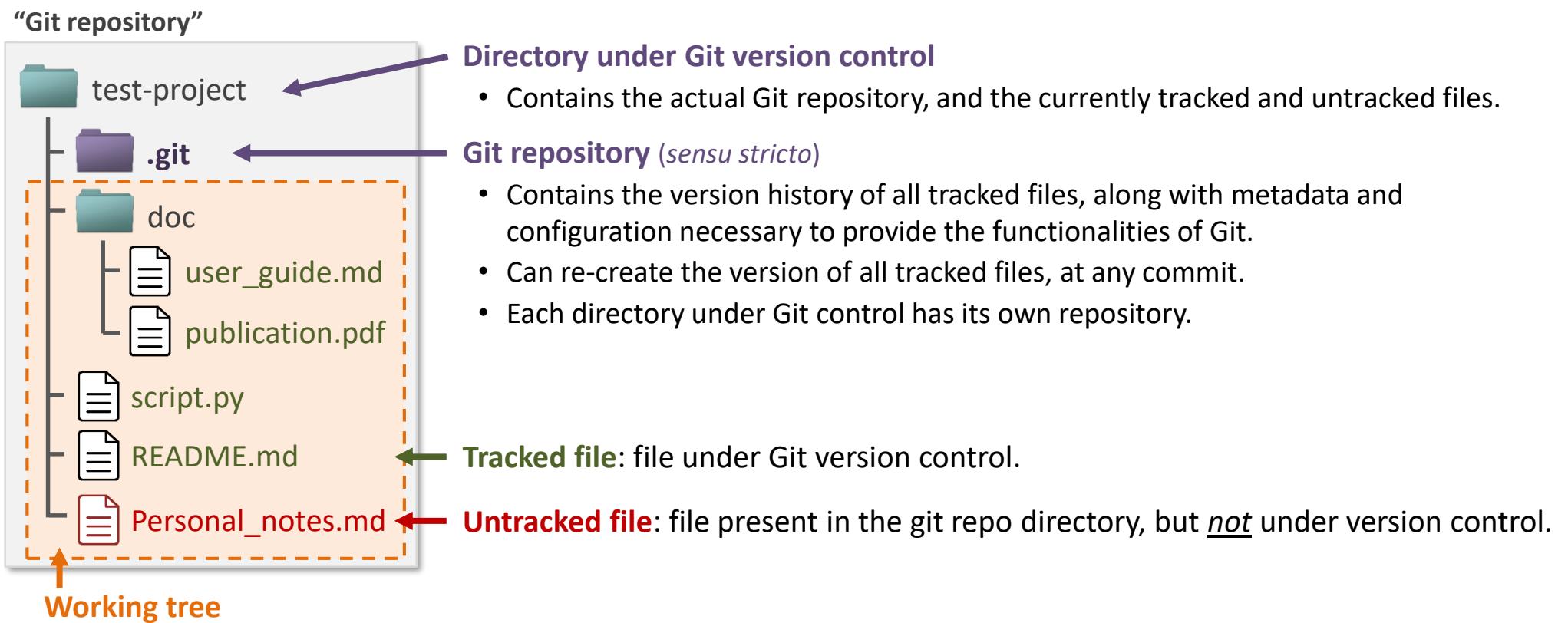
## Definition: snapshots are called "commits"

- **Commit = snapshot + metadata** (author, time, commit message, parent commit ID, etc. ...).
- Create a new commit = record a new state of the directory's content.  
Note: we will soon see that this statement is *not* 100% correct, but it is a good-enough approximation for now.
- Each commit has a unique **ID number / hash** (40 hexadecimal characters): 3c1bb0cd5d67dddc02fae50bf56d3a3a4cbc7204



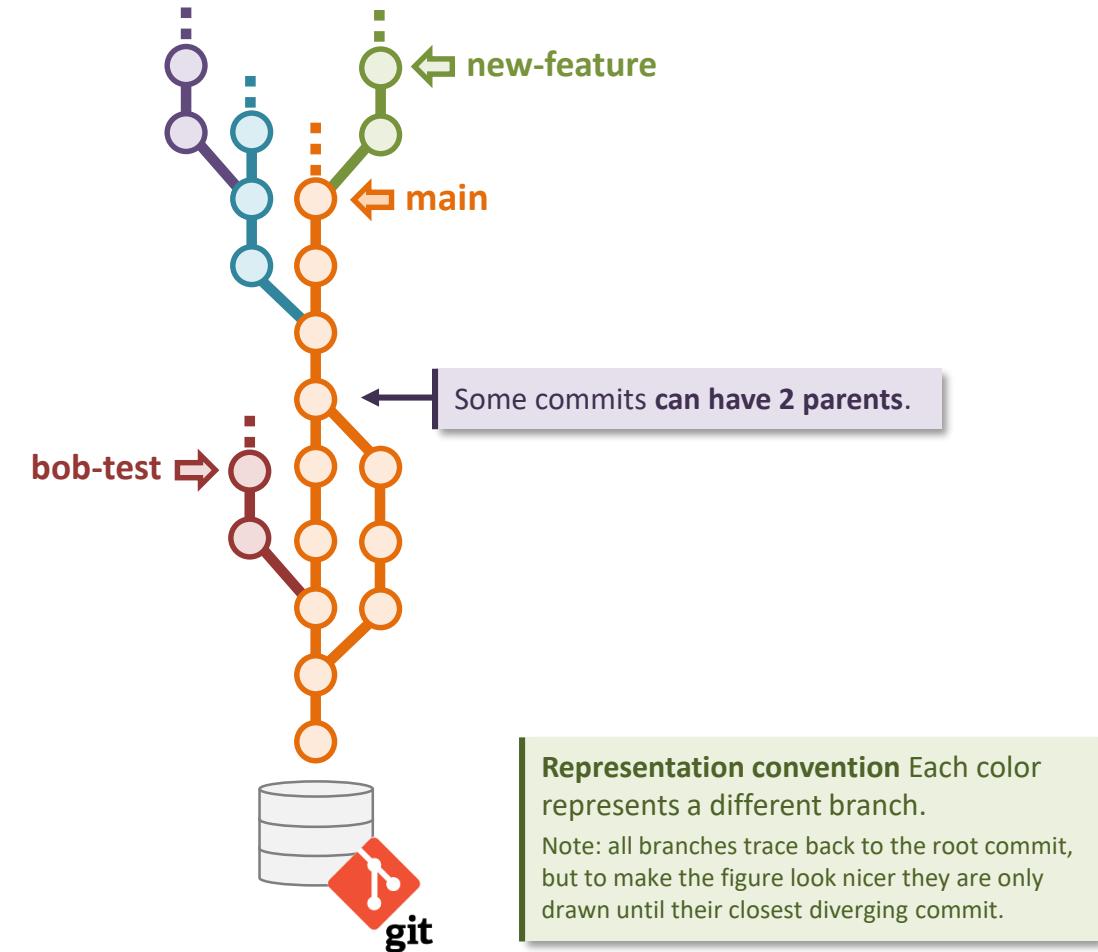
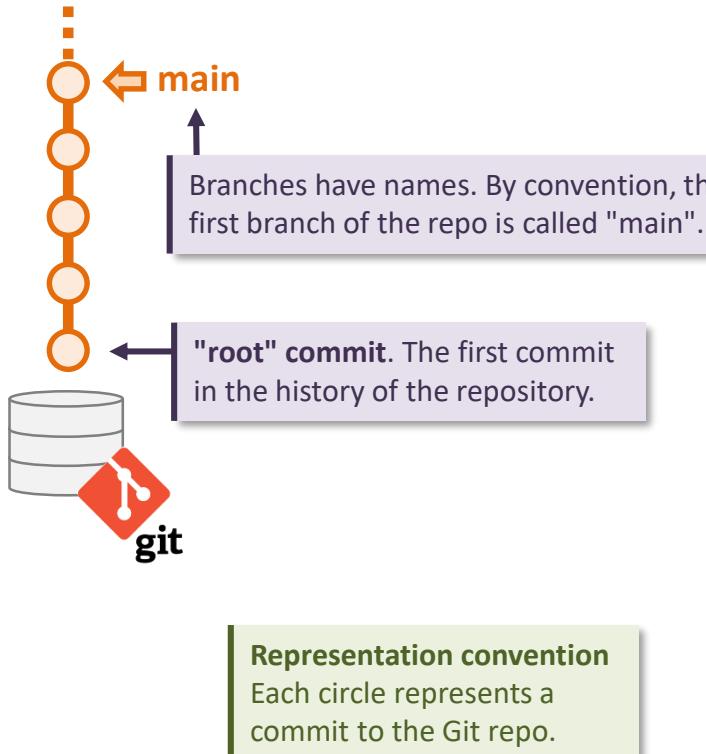
## Definition: commits are stored in a repository (or “repo”)

- **Git repository/repo:** version history of files in a directory under Git version control, along with metadata, and configurations necessary for version tracking and collaboration.
  - Technically, a Git repository is only the hidden “.git” directory (see figure below), but often the term is also used to refer to the entire directory under Git control (“test\_project” in the example below).
  - Not all files in a directory under Git control have to be tracked: there can be a mix of **tracked** and **untracked** files.
- **Working Tree:** current content (on your computer) of a directory under Git control.
  - More exhaustive definition: state of the project files corresponding to the branch/commit that is currently checked out, augmented with uncommitted changes made to files, as well as untracked files.



## Definition: branches

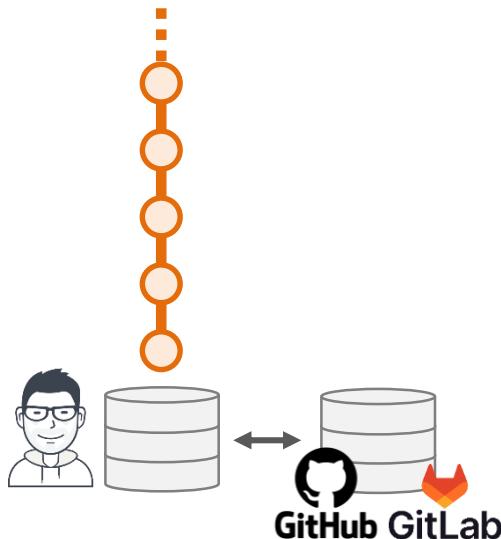
- **Branch:** refers to a “line of development” within the commit history, a lineage of commits starting from the latest (newest) commit of the branch and tracing back through the history of the repo.  
Technically, a branch is simply a pointer (reference) to a commit.
- **Repository history:** chronology of all of commits reachable from a branch (or tag).  
Technically, a directed acyclic graph (DAG) of commits that describe how the repository content evolved over time.



# Git use cases: examples

## Exercise 1

### Single repo, single branch



#### Use case

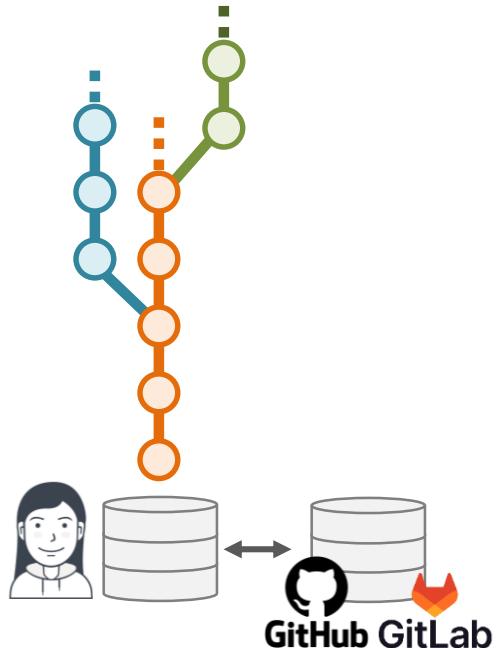
- Keep a documented log of your work.
- Go back and compare to earlier versions.
- See what changed since last commit.
- Backup (if a paired with a remote).
- Distribute your code (if paired with remote)



**The local repo must be associated to a remote repository to provide backup functionality (and new commits must be regularly pushed). Highly recommended.**

## Exercises 2 and 3

### Single repo, branched workflow (multiple development lines)

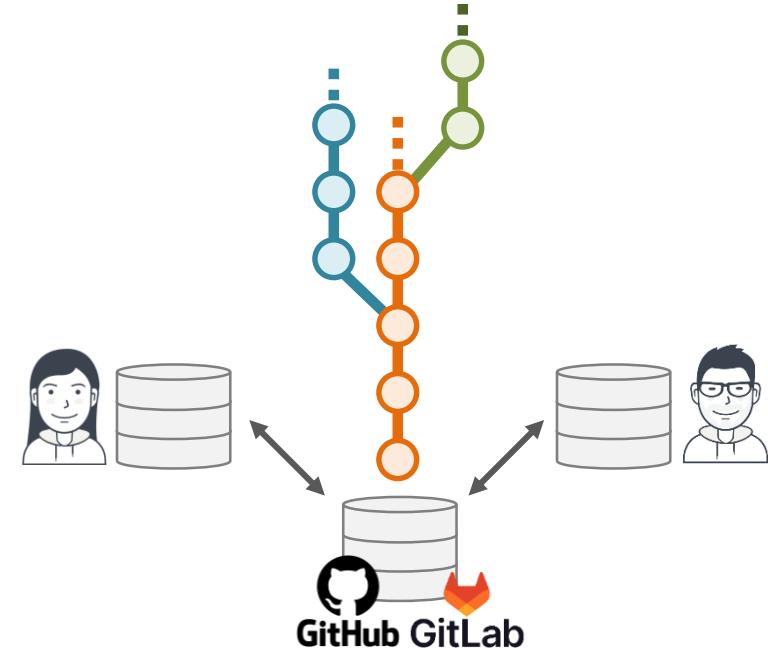


#### Use case

- Service in production with continued development in parallel (e.g. adding new feature).
- + all benefits of the previous use case.

## Exercise 4

### Collaboration with distributed and central repos.



#### Use case

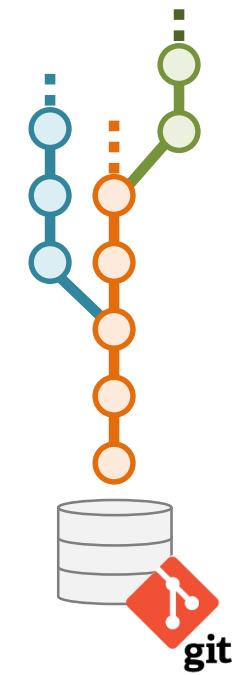
- Collaborate with others (distributed development).
- + all benefits of the previous use case.

**Each user has a full copy of the data \***

\* Provided they regularly sync their local repo.

## Local vs. Remote repository

- When creating a new Git repository on your computer, **everything is only local**.
- To get a copy of your repository online, you must take the active steps of:
  - **Creating** a new repository on a hosting service (e.g. GitHub, GitLab, Bitbucket, ...).
  - **Associate** the online repository with your local repo.
  - **Push** your local content to the remote.
- By design, Git **does not automatically synchronize** a local and remote repo. Download/upload of data must be triggered by the user.



## Using Git with large files: the problem

Git can store any type of file (“plain text” or binary).

**It would be nice if we could store data (large files) together with code ...**

Tracking large files together with code is an attractive proposition, e.g. in scientific applications:

- Data analysis/processing pipeline.
- Machine learning applications (training data and code in the same place).

**... but Git does not work well with large files**

- Git was designed for tracking code – i.e. relatively small text files.
- Adding large files to a Git repo is technically possible, however:
  - Since Git is a distributed VCS (version control system), **each local copy of a repository will contain a full copy of all versions of all tracked files**. Therefore, **adding large files will quickly inflate the size of everyone's repository**, resulting in higher disk space usage (on local hosts).
  - **Git's internal data compression** (i.e. packfiles) is **not optimized to work with binary data** (e.g. image or video files). Each change to a binary file will (more or less) add the full size of the file to the repo, taking disk space and slowing down operations such as repo cloning or update fetching.
  - Commercial **hosting platforms impose limits on the size of files** that can be pushed to hosted Git repos (GitHub: 100 MB, GitLab: no file limit but 10 GB repo limit).

## Using Git with large files: possible solutions

### Git LFS (Large File Storage)

Git LFS (Large File Storage) is an extension for Git, specifically **designed to handle large files**.

- Open source project: <https://git-lfs.github.com>

Basic principle: large files are not stored in the Git database (the `.git` directory), instead:

- Only a **reference/pointer to large files** is stored in the Git database.
- The actual **files are stored in a separate repository** or “object store”.



Not all hosting services support Git LFS, and when they do, storage space is limited (additional space may be purchased).

### DVC (Data Version Control)

DVC (Data Version Control) is a software that integrates with Git (a sort of layer used on top of Git) to allow versioning and storage of large files.

- Open source project: <https://dvc.org>

# Git configuration

```
git config
```

# Configuring Git

The minimum configuration is setting a **user name** and **email**. These will be used as default author for each commit.

- Setting user name and email:

```
git config --global user.name <user name>
git config --global user.email <email>
```

The **--global** option/flag tells Git to store the setting at the “global” (user wide) scope. Global settings apply to all Git repos on your machine. Without the **--global** option, the setting will only apply to the current Git repo.



Global settings are stored in the following config file:

- Linux: /home/\$USER/.gitconfig
- Mac OS: /Users/<user name>/.gitconfig
- Windows: C:/Users/<user name>/.gitconfig

- Config values can be retrieved by using the **--get** option.

```
# Set user name and email at the global (user-wide) scope.
> git config --global user.name "Alice"
> git config --global user.email alice@redqueen.org

# Retrieve setting values.
> git config --get user.name
Alice
> git config --get user.email
alice@redqueen.org
```

## Configuring Git: changing the default text editor

On most systems, the default editor that Git uses is “**vim**”.

However, this can be configured with the following **git config** command:

```
git config --global core.editor <editor cmd>
```

- Display the current default editor used by Git:

```
git config --global --get core.editor
```

- Example: changing the default editor to “nano” (another command line editor).

```
# Change the default editor to "nano".  
$ git config --global core.editor nano  
  
# Change the default editor to "VS Code".  
$ git config --global core.editor "code --wait"  
  
# Display the current default editor.  
$ git config --global --get core.editor  
nano
```

## Configuring Git: scopes and their config file locations

Depending on their scope, Git configurations apply to all Git repositories of a user, or only to a specific repository.

The main 3 scopes are:

- **Global (user wide):** settings apply to all Git repositories controlled by the user.
  - To save a setting at the global scope, add `--global` to the `git config` command: `git config -global ...`
  - Stored in `/home/<user name>/.gitconfig` (Linux), `C:\Users\<user name>\.gitconfig` (Windows) or `/Users/<user name>/ .gitconfig` (Mac OS).
- **Local (repo specific):** settings apply only to a specific Git repo.
  - Stored in the `.git/config` file of the repository.
- **System (system wide):** settings apply to all users and all repos on a given machine. This can only be modified by a system administrator.

To show the list of all Git configurations, along with their scope and the location of the file they are stored-in:

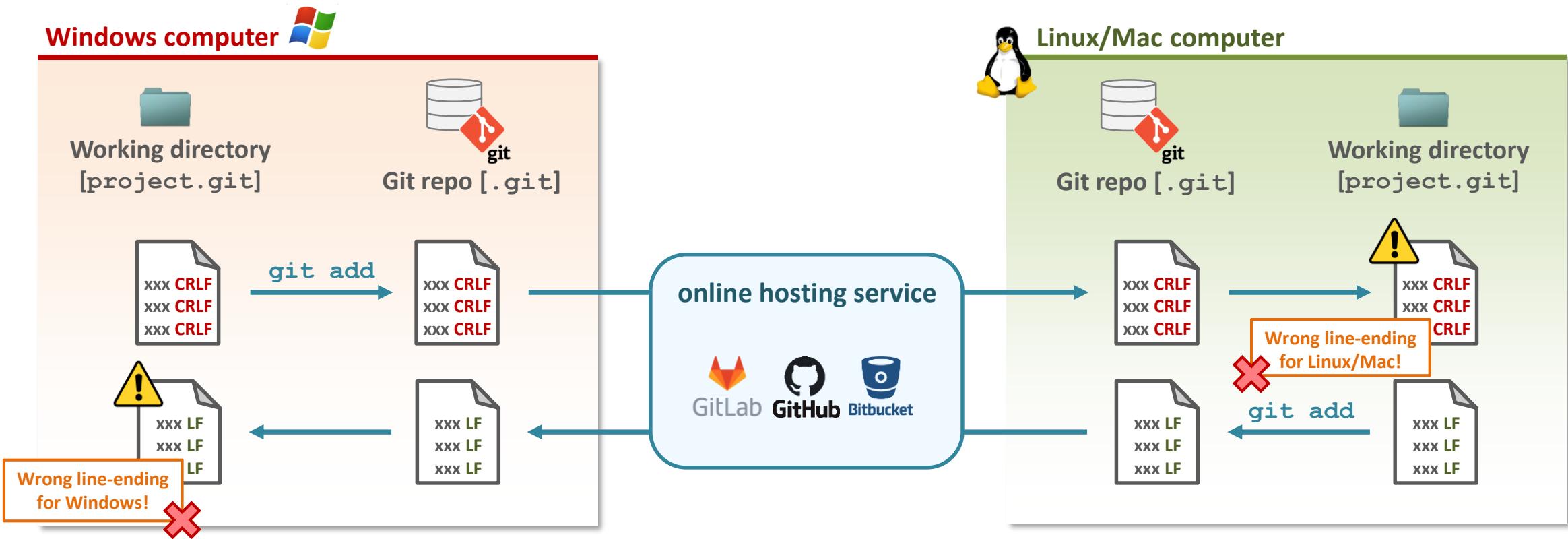
```
git config --list --show-origin --show-scope
```

## Cross-platform collaboration: the line-end problem

Linux/Mac and Windows do not use the same “line-end” characters: this can cause problems when collaborating with people who use a different operating system.

- **Linux/Mac:** uses **LF** (linefeed; `\n`) as line-ending character.
- **Windows:** uses **CRLF** (carriage-return + linefeed; `\r\n`) as line-ending character.

→ Problem: text files created on Windows will not work well on Linux/Mac and vice versa.



## Cross-platform collaboration: solution

The solution is to ask Git to automatically convert between LF and CRLF during add/checkout operations using the configuration option:

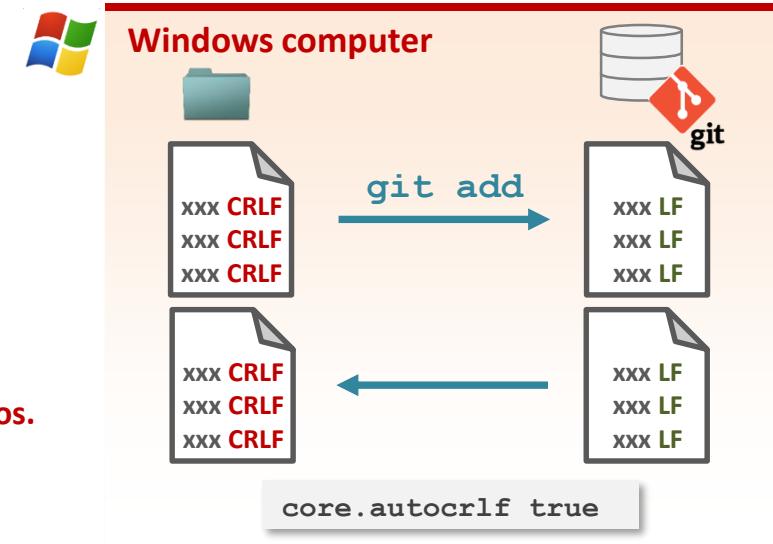
```
git config core.autocrlf
```

- On Windows computers: `core.autocrlf true` should be set so that LF are automatically changed to CRLF each time a file is checked-in or checked-out.

```
git config core.autocrlf true
git config --global core.autocrlf true
```

← Change setting for current repo.

← --global = change setting for all repos.

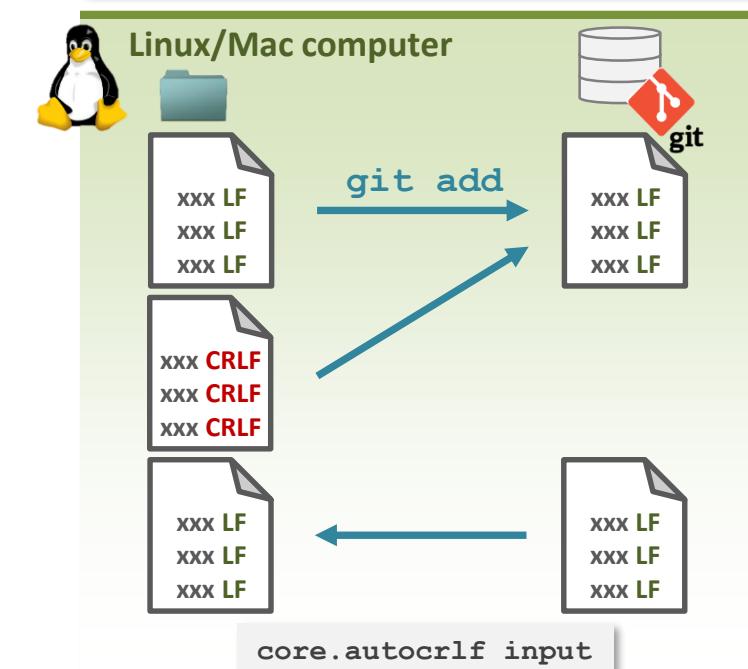


- On Linux/Mac computers: `core.autocrlf input` should be set so that LF line-endings (LF) are left untouched, and that CRLF are converted to LF when a file is added (this will only be useful in the rare cases when a file with CRLF ending is somehow present on the machine, e.g. because it was sent via email by a Windows user).

```
git config core.autocrlf input
git config --global core.autocrlf input
```

- `core.autocrlf false` to disable LF/CRLF auto-modifications (this is the default):

```
git config core.autocrlf false
git config --global core.autocrlf false
```



## core.autocrlf warnings

When `core.autocrlf` is set to `True` (this is in principle only for windows users), a warning is displayed when files are added/checked-out to/from the git repo:

```
> git add test_file.py
warning: LF will be replaced by CRLF in test_file.py
The file will have its original line endings in your working directory
```

Somehow the message is the same during adding and check-out of files... so when adding files to the index (`git add`), the message is actually the wrong way round: it should be something like “CRLF will be changed to LF in checked-in file”.



# Creating a **new repo**

```
git init
```

```
git clone
```

## Obtaining a new Git repo: two main ways...

### Turn a local directory into a Git repo (start from scratch)

Enter the directory to version-control, then run:

```
git init
```

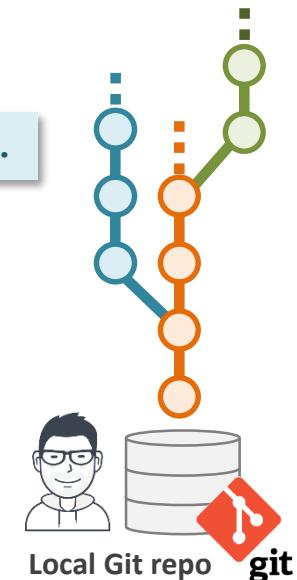


- A new, empty, Git repository is created in the current directory.
- Files present in the directory can now be version-controlled. However, version-control of files is not automatic – more on that later.
- At this point there is no online remote associated with the new repo. Everything is only local.

### Clone a repo from an online source (start from an existing repo)

```
git clone https://github.com/...
```

Remote copy of repo



- The entire content of the online Git repository is “cloned” (downloaded) to the local machine.
- The online repo is automatically linked (setup as a “remote”) for the local repo: we can push commits with no additional setup.
- Starting a new project on GitHub/GitLab and cloning it can also be a way to create a new empty local repository and immediately link it to a remote.

Cloning and working with remotes will be presented in more details later in these slides.

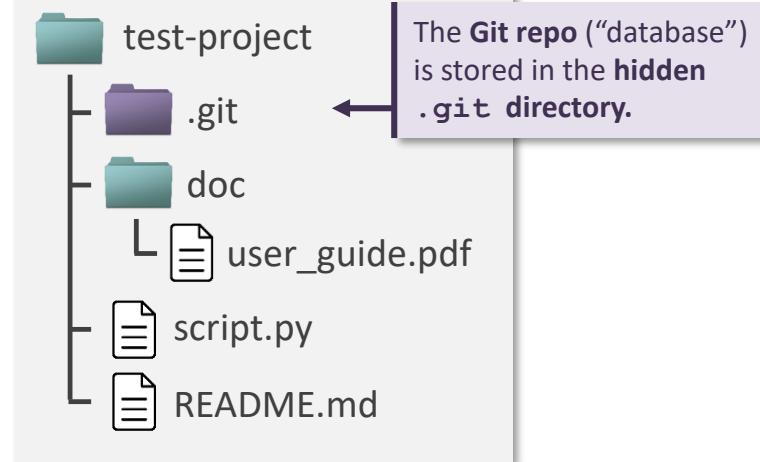
# Creating a new Git repository (from scratch)

**git init**

Initializes a Git repository in the current working directory, turning it into a version controlled directory.

**Example**

```
# Enter directory to version control.  
› cd /home/alice/test-project  
  
# Initialize a new Git repo.  
› git init  
Initialized empty Git repository in /home/alice/test-project/.git/  
  
# List the content of our directory: we now see a new .git subdirectory.  
› ls -a  
./ ../.git/ doc/ src/ README.md
```



- You must be located **at the root of the directory to version control** before typing **git init**
- **git init** creates a hidden **.git** directory at the root of the directory.
- **Everything** is stored in this single **.git** directory:
  - Complete version history of all tracked files.
  - All other data associated with the Git repository (e.g. branches, tags).
  - The content of **.git** can re-create the exact state of all your files at any versioned time, e.g. if you delete a file accidentally or want to go back to an earlier version.

**Never delete the .git directory**

Unless you intend to start again your repo from scratch, and accept to lose all its history.

## State of the working directory (here just after `git init`)

Useful commands to assess the current status of a Git repo:

- Show status of files in project directory (working tree).

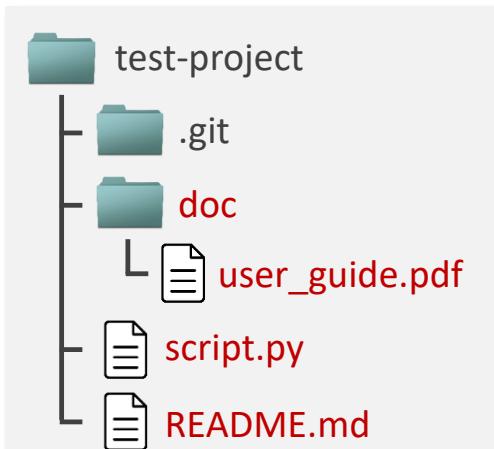
`git status`

```
> git status
On branch main
No commits yet
Untracked files:
  doc/
    README.md
    script.py
```

“main” is the default branch name.

Red = untracked files

How it looks in the file system



- Commit history: show log of commits, i.e. the history of the repo.

`git log`

```
> git log
fatal: your current branch 'main' does not have any commits yet
```

Since we just created a new repo there are no commits yet, which is why we get this error.

- List files currently tracked by Git (i.e. part of the Git index).

`git ls-files`

```
> git ls-files
<empty output>
```

By default, files are untracked. This is why initially there is no tracked file.

## A few more things about Git repos

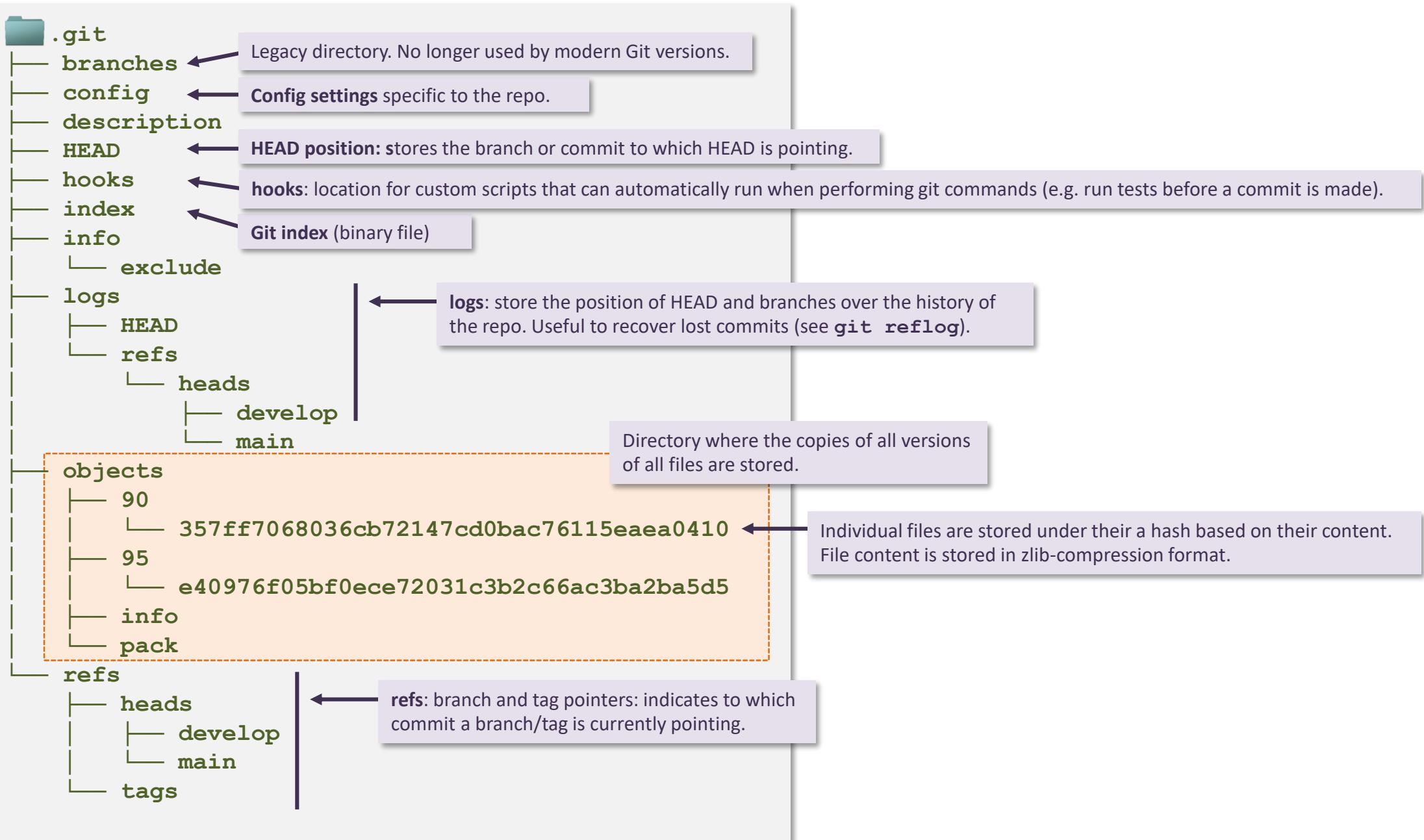
- When creating a new Git repo, it does not matter whether the directory is empty or already contains files/sub-directories.
- Files in the project directory (working tree) are not automatically tracked by Git: files are untracked by default.
- You can have both tracked and untracked files in a project directory.
- Only files located in the project directory – or one of its sub-directories – can be tracked.
- Project directories are self-contained – you can rename them or move them around in your file system.
- You can (should) have multiple Git repositories on your system – typically one per project or per code/script you develop \* - don't use a single Git repo to track the entire content of your computer!
- Nesting Git repositories (i.e. having one repo inside another) is technically possible, but should be avoided unless there is a clear use-case for it.

### Reminder

 **Never delete the `.git` directory**, you would lose the entire versioning history of your repository (along with all files not currently present in the working tree).

\* An exception is the case of multiple projects that are tightly linked to another: in such cases it can be useful to have them all in a single repo – this is known as a **monorepo**.

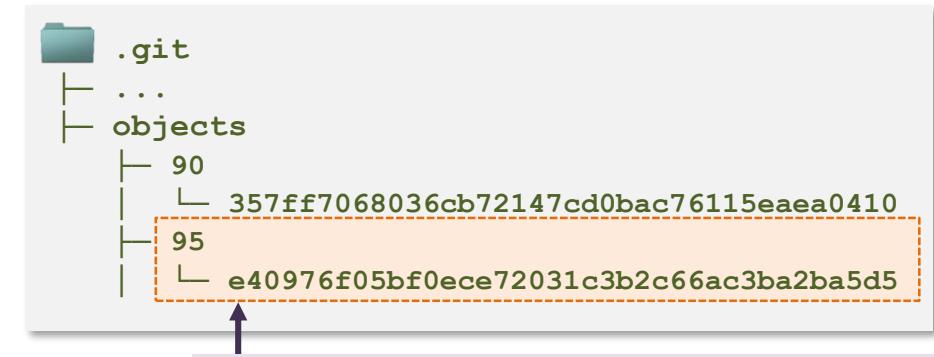
## Behind the scenes: the content of the `.git` directory



## Behind the scenes: how Git stores files in `.git/objects`

Each time a new file (or a new version of a file) is added to a Git repo, Git stores it in its object store (`.git/objects`) by:

- Computing a hash values for the file (see details below).
- Compressing the content of the file in zlib-format, and storing the output under the file's hash value. This is called a "blob".



### Hashing details

When computing the hash value of a file, Git does not directly hash the raw file content. It prepends a little metadata (object type and size) in the form:

```
blob <size>\0<file content>
```

and runs this content through a hashing function. Historically Git has used SHA-1 as hashing function, but starting with Git 2.51 it now uses SHA-256 (more secure).

To compute the hash of a file, as Git would:

```
git hash-object
```

```
# Compute the hash of the file "README.md".
> git hash-object README.md
90357ff7068036cb72147cd0bac76115eaea0410
```

To print the content of a "blob":

```
git cat-file
```

```
# Print the content of a "blob".
> git cat-file -p 90357ff7068036cb72147cd0bac76115eaea0410
...
```

## “Bare” Git repositories

A **bare repo** is a repo that has **no working tree**: it does not contain any instance of the files that are under Git version control, but only the content of the `.**git**` directory/database.

This type of repo is found on remote servers used to share and sync changes across multiple Git repositories. They can be initialized with the command:

```
git init --bare
```

# Making a **commit**

git add

git commit

# Making a commit

In Git, creating a commit is a 2-step process:

## Step 1 – Staging files

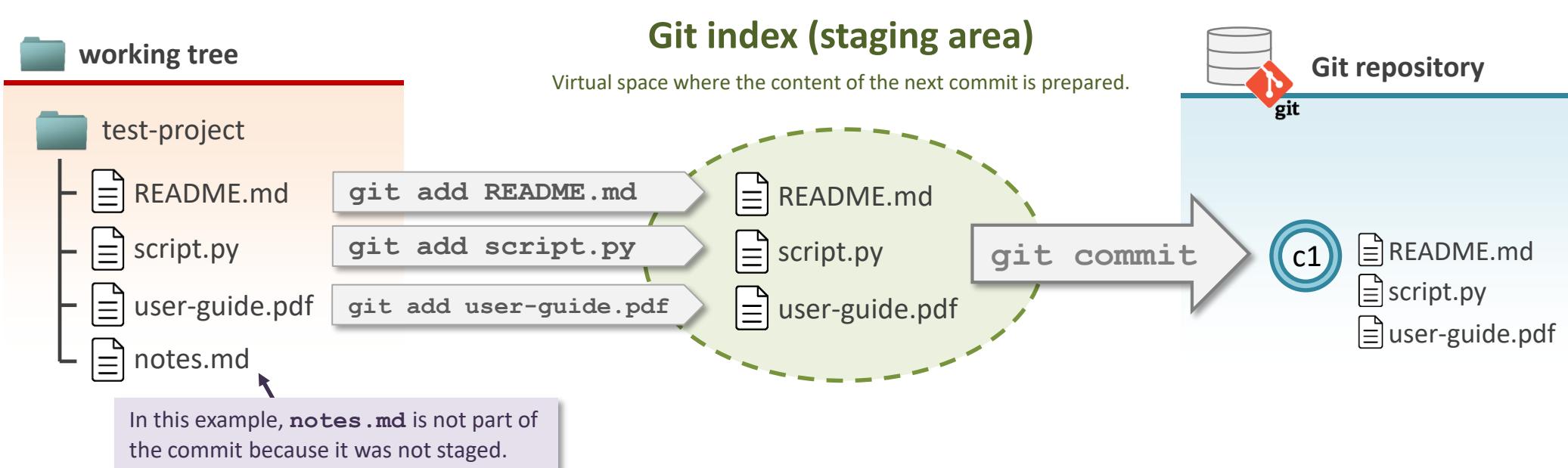
**Selection of files to commit.** To make a new or modified file part of the next commit, it must be added to the **Git index** (also known as the **staging area**).

```
git add <file or directory>
```

## Step 2 – Commit

**Create a commit with the current content of the Git index.** A new commit (containing the current content of the Git index) is added to the repository.

```
git commit -m "commit message..."
```



## Making a commit

- To commit a new/updated version of a file, the file (in its updated version) must be **staged again** (added to the Git index again).
- Once a file has been added to the index, it **remains in the index** and there is no need to add it again (unless it changes and we want to commit these changes).
- Files that are in the index are called **tracked files**.
- Files that are not in the index are called **untracked files**.
- **Removing files from the Index** is done with:

```
git rm --cached <file to remove>
```

← Removes the file from the Git index.

```
git rm <file to remove>
```

← Removes the file from the Git index and **deletes it** from your working tree (i.e. your local disk).

- Files can be **partially staged** (only add some of the changes to the index) by using the **--edit** or **--patch** options of **git add** .

## Definition: the Git index (staging area)

### Definition

**Git index (staging area):** “virtual space” where changes are collected before committing them to the repository, providing fine-grained control over which changes to include in the next commit.

#### *Technical notes:*

- In practice, the Git index is a file in Git’s database.
- It is also possible to stage only certain changes made to a file, instead of the entire file.



We can now update our **definition of a commit**.

### Definition

**Commit** = snapshot of the Git index at a given time.  
**Git index** = content of your next commit.

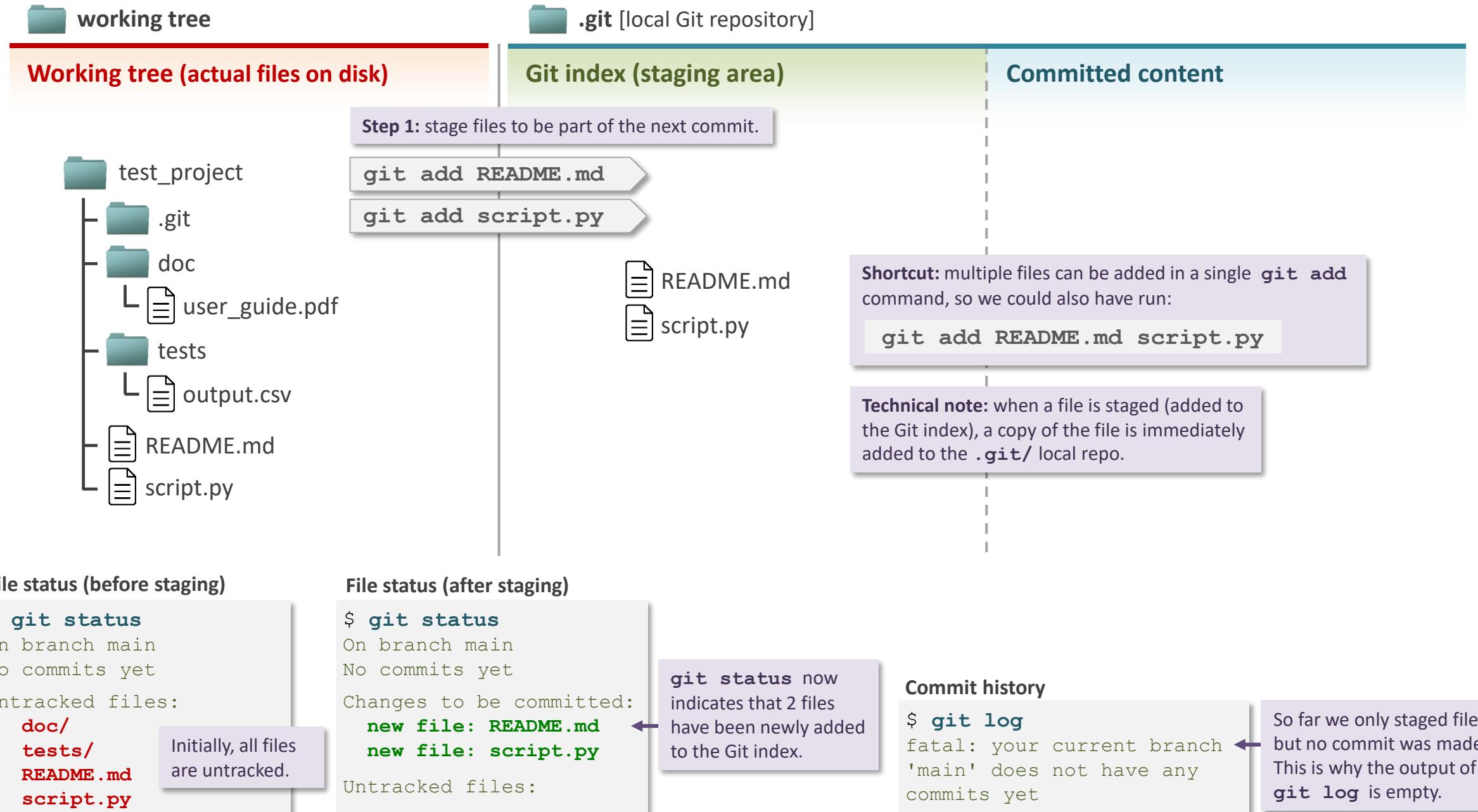
Why do we need the Git index and this 2-step process ? Why not always commit the entire content of a repo ?

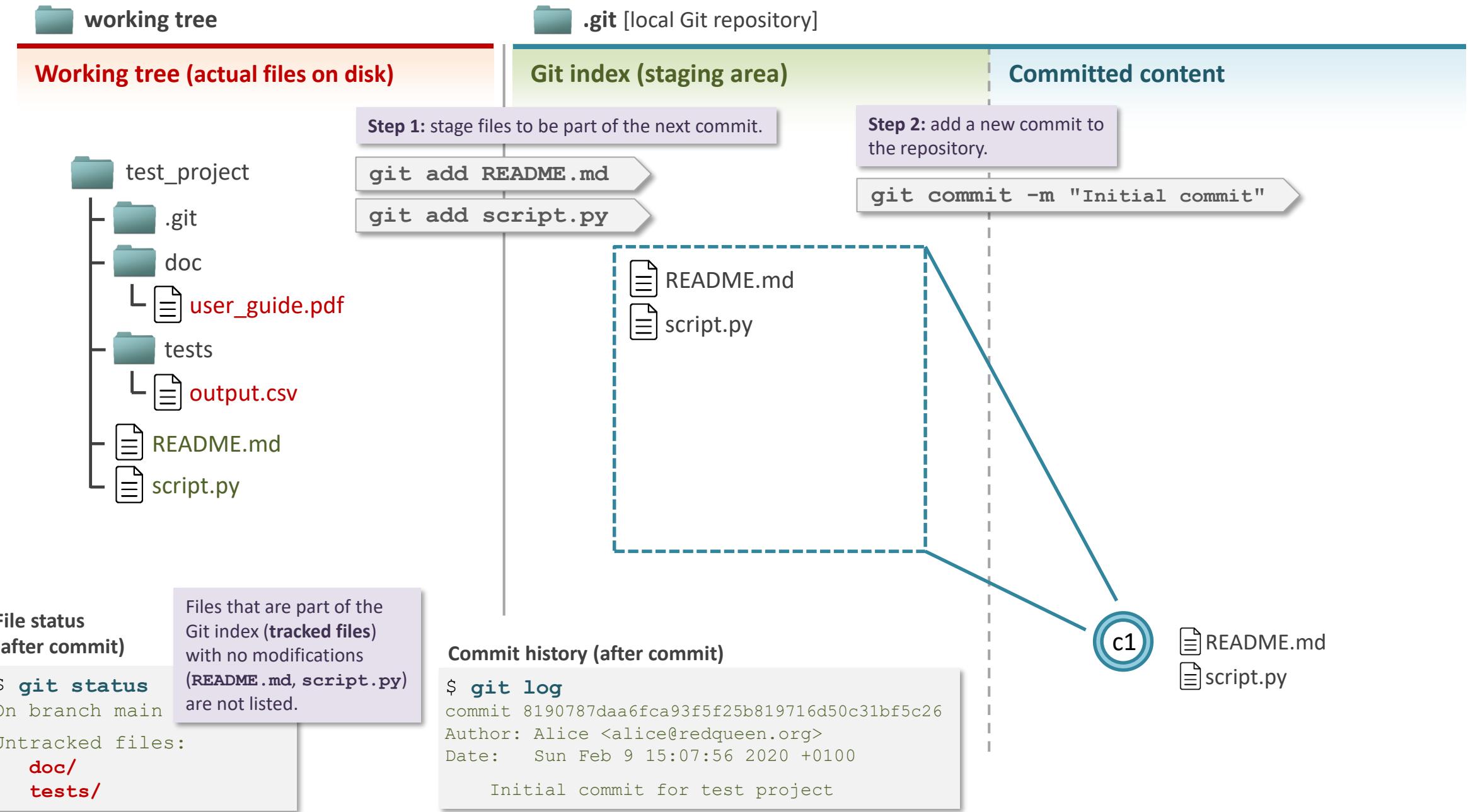
→ The objective of this 2-step procedure is to let users make “well thought” commits.

- Commits are meant to be **meaningful units of change** in your code base (or the content you track).
- Not all changes made to the working tree need to be part of the next commit.
- Well defined commits make it easier to track bugs or revert changes if needed.

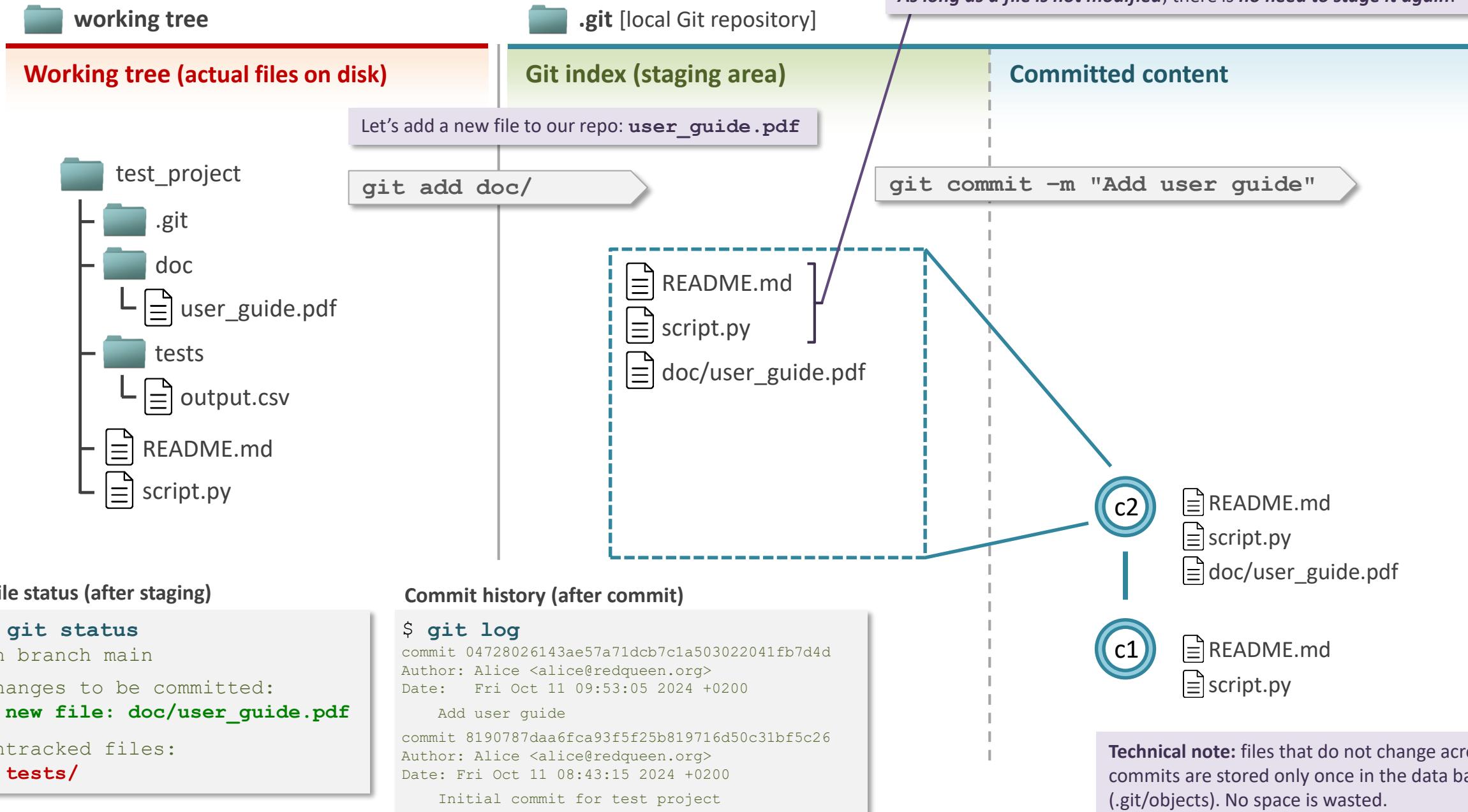


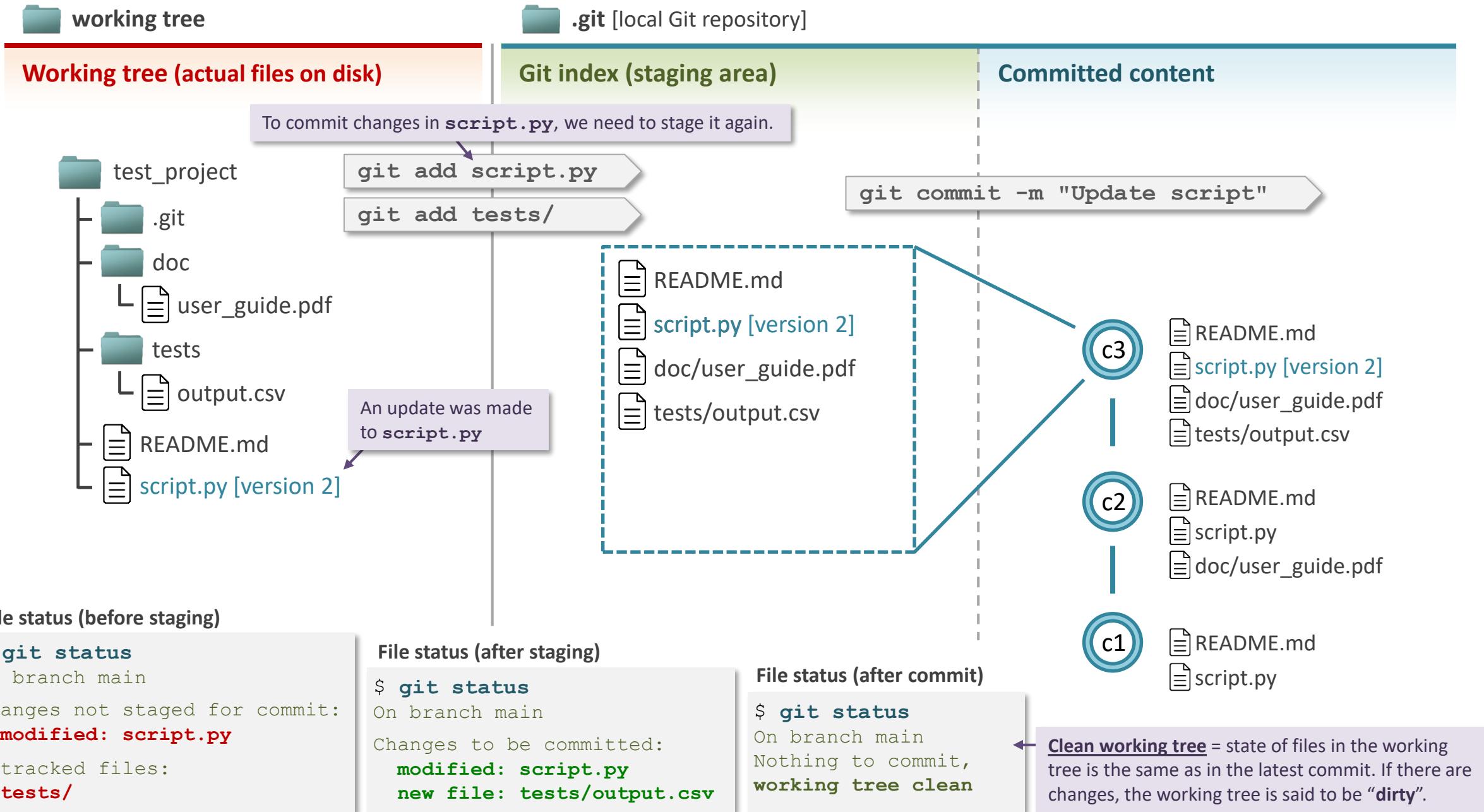
## Staging and making a commit: a step-by-step example





## Staging and making a commit: a step-by-step example





## Summary: staging (`git add`) and committing (`git commit`) files

- By default, files in a directory under Git control are **untracked**.
- To include a file (in its current state) – or a change in file content – in the next commit, the file **must be added to the Git index (staged)** with:

```
git add <file or directory>      # Add the specified files/directories to the Git index.
```

- Multiple files/directories can be added in a single command (by passing multiple file/directory names).
- By default, the entire content of a file is added.  
Adding only part of a file is possible with the `--edit` or `--patch` options.

- Staged files remain staged, unless explicitly removed (with `git rm` or `git rm --cached`).
- **Modified files must be staged (added to the index) again**, if the new content is to be added to the next commit.
- To create a new commit, the command is:

```
git commit -m/--message "your commit message"  
git commit
```



If no commit message is given, Git will open the default editor and ask you to enter a message **interactively**.

### Reminder

**commit = snapshot of the Git index**

The Git index (staging area) can be thought of as a “virtual space” where the content of the next commit is prepared.

## Making commits: some advice

Git does not impose any restrictions on what and when things can be committed. \*

However, it's best to:

- Make commits at *meaningful points* of your code/script development, for instance:
  - When a new feature was added (or a few related functions).
  - When a bug was fixed.
- Make *multiple small commits instead of a large one* if you are making changes that affect different functionalities of your code (this can make it easier to e.g. revert changes).
- *Don't commit broken code on your main/master branch*, as this is the branch that others might use to get the latest version of your code.

If you have partial work, you can commit it to a *temporary/feature* branch, and later merge it into *main/master* (more on branch management will follow later).

\* One exception being that, by default, commits with zero changes are not allowed, but they are possible by using the `--allow-empty` option: `git commit --allow-empty`

## Shortcuts: for staging (`git add`) and committing (`git commit`) files

### ▪ `git add` shortcuts

```
git add -u / --update      # Stages all already tracked files, but ignore untracked files.  
git add -A / --all        # Stages all files/directories in the working tree (except ignored files), including file deletions.  
git add .                 # Stages entire content of the current directory, except file deletions.
```

### ▪ `git commit` shortcuts

```
git commit -m "commit message" <files or dirs>    # Stage and commit the files/directories in a single command.  
git commit -a / -all -m "commit message"             # Stage and commit all modified tracked files in a single command.
```

This is a shortcut for:

```
git add <file or directory>  
git commit -m "commit message"
```

-a / --all is a shortcut for:

```
git add -u  
git commit -m "commit message"
```

It will not stage/commit untracked files.

## Detail: understanding the output of `git commit`

```
$ git commit -m "Initial commit for test-project"  
[main (root-commit) 8190787] Initial commit for test-project  
3 files changed, 6 insertions(+) ←  
create mode 100644 README.md  
create mode 100644 script.py  
create mode 100644 doc/quick_start.md
```

+ 4 (empty lines also count)

6 insertions = 6 lines added in total (across all files)

**README.md**

+ 1 | # Quick-start guide for the test-project software

**script.py**

+ 1 | #! /usr/bin/env python3

**doc/quick\_start.md**

# Test project: testing version control with Git

A small test project to illustrate the use of Git.  
Maybe I will add more content to it later.

## Committing content: interactive commit message with the “vim” editor

```
$ git commit
```

### Initial commit for test\_project

```
# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
#
# On branch main
# Changes to be committed:
#   new file: README.md
#   new file: script.py
#   new file: doc/quick_start.md
#
```



When no commit message is specified,  
Git automatically opens a text editor.  
By default, this editor is “vim”.

- In the “vim” editor, press on the key “i” to enter edit mode
- In edit mode, you can now type your commit message.

## Committing content: interactive commit message with the “vim” editor

```
Initial commit for test_project
```

```
This is the very first commit in this Git repo.
```

```
Way to go!
```

```
# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
#
# On branch main
# Changes to be committed:
#       modified: README.md
#       new file: script.py
#       new file: doc/quick_start.md
#
~
~
:wq
```

- Commit message can be entered over multiple lines.
- By convention, try to keep lines reasonably short (<= 80 chars)

- Press “**Esc**” to exit “edit” mode.
- Type “**:wq**” in the vim “command” mode.



Press “**Enter**” to exit vim and save your commit message.

- You are now back in the shell and your commit is done.

```
[main (root-commit) 8190787] Initial commit for test_project
3 files changed, 6 insertions(+)
create mode 100644 README.md
create mode 100644 script.py
create mode 100644 doc/quick_start.md
```

## Demo

- Initializing a new Git repo.
- Adding content to the Git repo.
- Making a commit with interactive commit message.

# exercise 1 – part A

Your first commit



This exercise has helper slides

## Exercise 1 help: bash (shell) commands you may need during this course

<b>cd &lt;directory&gt;</b>	Change into directory (enter directory).
<b>cd ..</b>	Change to parent directory.
<b>ls -l</b>	List content of current directory.
<b>ls -la</b>	List content of current directory including hidden files.
<b>pwd</b>	Print current working directory.
<b>cp &lt;file&gt; &lt;dest dir&gt;</b>	Copy a file to directory “dest dir”.
<b>mv &lt;file&gt; &lt;new name&gt;</b>	Rename a file to <new name>.
<b>mv &lt;file&gt; &lt;directory&gt;</b>	Move a file to a different directory.
<b>cat &lt;file&gt;</b>	Print a file to the terminal.
<b>less &lt;file&gt;</b>	Show the content of a file (type “q” to exit).
<b>vim &lt;file&gt;</b>	Open a file with the “vim” text editor.
<b>nano &lt;file&gt;</b>	Open a file with the “nano” text editor.

# Inspecting file status

git status

git diff

# Display file status

**git status**

Display the status of files in the working tree.

- \* **Modified files:** files with changes in content as compared to the latest commit.
- \*\* **Staged files that have not been modified since the last commit (unmodified files)** are not listed, but they are still in the index and will be part of the next commit.
- **Ignored files** are also not listed.

```
$ git status
```

On branch main

#### Changes to be committed:

(use "git restore --staged <file>..." to unstage)

<b>new file:</b> LICENSE.txt <b>modified:</b> README.md <b>modified:</b> script.py <b>deleted:</b> test/test_output.csv	← new file = file is not present in latest commit. ← modified = file is modified compared to latest commit. ← deleted = file is present in latest commit and will now be removed
--	--

Staged files \*\*

**Green = files with (changes in) content (compared to the latest commit) that has been staged and will be part of the next commit.**

**Red = files with (changes in) content (compared to the latest commit) that is not staged. These changes will not be part of the next commit.**

#### Changes not staged for commit:

(use "git add <file>..." to update what will be committed)  
 (use "git restore <file>..." to discard changes in working directory)

<b>modified:</b> README.md <b>modified:</b> doc/user_guide.md <b>deleted:</b> test/log.txt	← modified = file is modified compared Git index. ← deleted = file is deleted on disk, but is still present in the Git index (and the latest commit).
--	--

Unstaged files

#### Untracked files:

(use "git add <file>..." to include in what will be committed)

untracked\_file.txt

untracked files



**Note:** the (new) content of a file can be **partially staged**: some changes in the file are staged (added to the index), while some remain unstaged. This is the case in the example above for the **README .md** file (which is why it's listed in both the staged and unstaged sections). **Only the staged content will become part of the next commit.**

## File status in Git: summary

Possible statuses for files in Git:

- **Tracked** – file that is currently under version control. More specifically, it is currently part of the Git index (staging area) and therefore also generally part of the latest commit \*. Tracked files can be further categorized as:
  - **Unmodified** – the file is part of the latest commit \* (and the Git index), and no change was made to the file since then. In other words, the content of the file in the working directory (working tree) is the same as in the latest commit. Unmodified files are not listed by the `git status` command.
  - **Modified** – the content of the file in the working directory (working tree) differs from the latest commit \*. Modified files can be staged, unstaged, or partially staged.
    - **Staged**: the difference in content has been added to the Git index (staging area), and will therefore be committed with the next commit.
    - **Unstaged**: the difference in content has not been staged (not part of the Git index), and will therefore not be part of the next commit.
    - **Partially staged**: some differences (but not all) have been staged (added to the Git index). Only the staged differences will be part of the next commit.
- **Untracked** - file present in the project directory (working tree), but not currently under version control by Git. More specifically, the file is not currently present in the Git index – but could be part of an earlier commit.
- **Ignored** - untracked file that is part of the repository's "ignore list" (`.gitignore` or `.git/info/exclude` file). Ignored files are not listed by the `git status` command.

\* more precisely: the commit to which the HEAD pointer is currently pointing – this concept is explained later in the slides.

## How do I know what changed and which changes are staged ?

### git diff

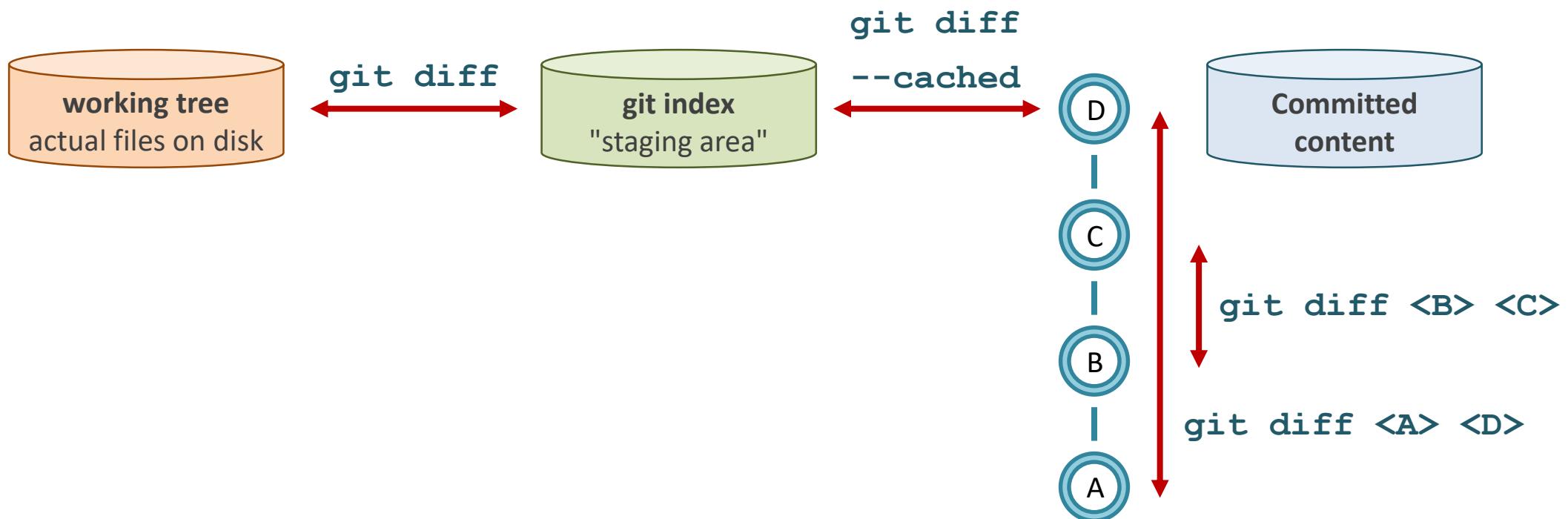
Show differences between two states of a Git repo.

```
git diff           # Show diff between working tree and Index.
git diff file     # Show diff for a specific file between working tree and index.
git diff --cached # Show diff between index and the last commit (HEAD).
git diff <older commit> <newer commit> # Show diff between 2 commits.

git diff --name-only # Add --name-only to show only file names, not the changes.
```

### Example output

```
> git diff
diff --git a/README.md b/README.md
index f5e333d..844d178 100644
--- a/README.md
+++ b/README.md
@@ -1,2 +1,3 @@
Project description:
-This is a test
+This is a demo project
+and it's pretty useless
```



# Inspecting **commits** and **history**

`git show`

`git log`

## Display the “content” of a commit

**git show** Display the changes in file content introduced by a commit.

**git show <commit reference>**

**git show**

with no argument, the latest commit on the current branch is shown (i.e. HEAD)

### Example:

```
$ git show 89d201f
```

```
commit 89d201fd01ead6a499a146bc6da5aa078c921ecf
Author: Alice <alice@redqueen.org>
Date:   Wed Feb 19 14:00:02 2020 +0100
```

```
    Add stripe color option to class CheshireCat

diff --git a/script.sh b/script.sh
index d7bfdcc8..fa99250 100755
--- a/script.sh
+++ b/script.sh
@@ -7,13 +7,28 @@
     class CheshireCat():
-     def __init__(self, name, owner="red queen"):
+     def __init__(self, name, owner="red queen", stripe_color="orange"):
+         self.stripe_color = stripe_color
```

### Examples of commit references:

- A commit ID (hash): **89d201f**
- A branch name: **develop**
- A tag name: **1.0.7**
- The **HEAD** pointer.
- A relative reference: **HEAD~3**

If no commit reference is given, **HEAD** is used as default.



The detail of changes can only be shown for plain text files.

**git show --name-only <ref>**

Only display file names (without the changes)

```
$ git show --name-only 89d201f
commit 89d201fd01ead6a499a146bc6da5aa078c921ecf
Author: Alice <alice@redqueen.org>
Date:   Wed Feb 19 14:00:02 2020 +0100
```

Add stripe color option to CheshireCat

script.sh

## Display commit history

Print the commit history of the repository (newest commit at the top)



**git log** has many options to format its output.  
See **git log --help**

```
git log  
git log --oneline  
git log --all --decorate --oneline --graph
```

### Example: default view

```
$ git log  
commit f6ceaac2cc74bd8c152e11b9c12ada725e06c8b9 (HEAD -> main, origin/main)  
Author: Alice alice@redqueen.org  
Date:   Wed Feb 19 14:13:30 2020 +0100  
  
        Add stripe color option to class CheshireCat  
  
commit f3d8e2280010525ba29b0df63de8b7c2cd7daeaf  
Author: Alice alice@redqueen.org  
Date:   Wed Feb 19 14:11:56 2020 +0100  
  
        Fix off_with_their_heads() so it now passes tests  
  
commit cfd30ce6e362bb4536f9d94ef0320f9bf8f81e69  
Author: Mad Hatter mad.hatter@wonder.net  
Date:   Wed Feb 19 13:31:32 2020 +0100  
  
        Add .gitignore file to ignore script output
```

## Example: compact view of current branch

```
$ git log --oneline
f6ceaac (HEAD -> main, origin/main) peak_sorter: add authors to script
f3d8e22 peak_sorter: display name of highest peak when script completes
cf30ce Add gitignore file to ignore script output
f8231ce Add README file to project
821bcf5 peak_sorter: add +x permission
40d5ad5 Add input table of peaks above 4000m in the Alps
a3e9ea6 peak_sorter: add first version of peak sorter script
```

## Example: compact view of entire repo (all branches)

```
$ git log --all --decorate --oneline --graph
* fc0b016 (origin/feature-dahu, feature-dahu) peak_sorter: display highest peak at end of script
* d29958d peak_sorter: add authors as comment to script
* 6c0d087 peak_sorter: improve code commenting
* 89d201f peak_sorter: add Dahu observation counts to output table
* 9da30be README: add more explanation about the added Dahu counts
* 58e6152 Add Dahu count table
| * f6ceaac (HEAD -> main, origin/main) peak_sorter: add authors to script
| * f3d8e22 peak_sorter: display name of highest peak when script completes
|/
* cf30ce Add gitignore file to ignore script output
* f8231ce Add README file to project
| * 1c695d9 (origin/dev-jimmy, dev-jimmy) peak_sorter: add check that input table has the ALTITUDE and PEAK columns
| * ff85686 Ran script and added output
|/
* 821bcf5 peak_sorter: add +x permission
* 40d5ad5 Add input table of peaks above 4000m in the Alps
* a3e9ea6 peak_sorter: add first version of peak sorter script
```



## Adding custom shortcuts to Git

Some git commands can be long and painful to type, especially when you need them often !  
To shorten a command, you can create **custom aliases**:

```
git config --global alias.<name of your alias> "command to associate to alias"
```

Example:

```
git config --global alias.adog "log --all --decorate --oneline --graph"
```

With the alias set, you can now simply type:

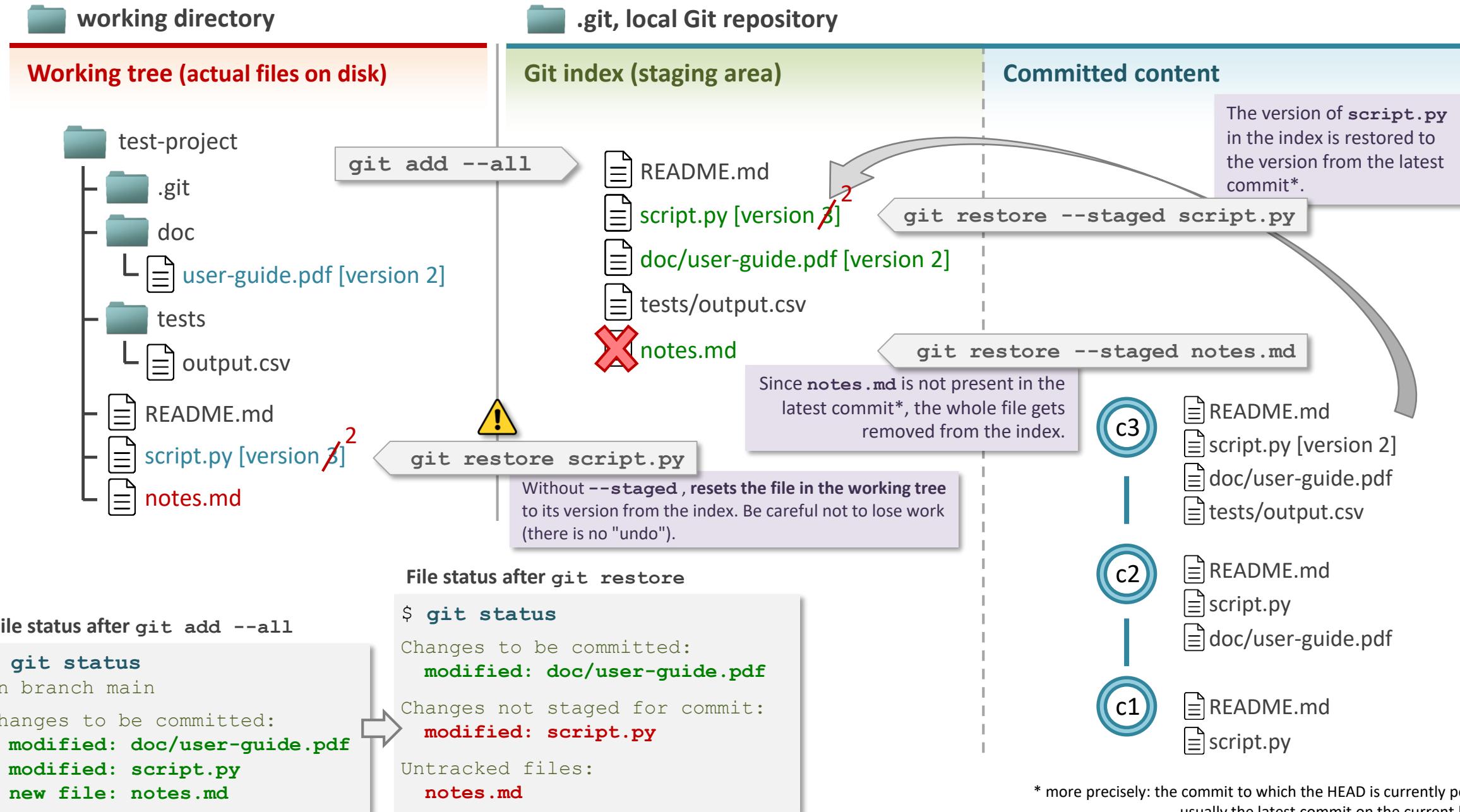
```
git adog
```

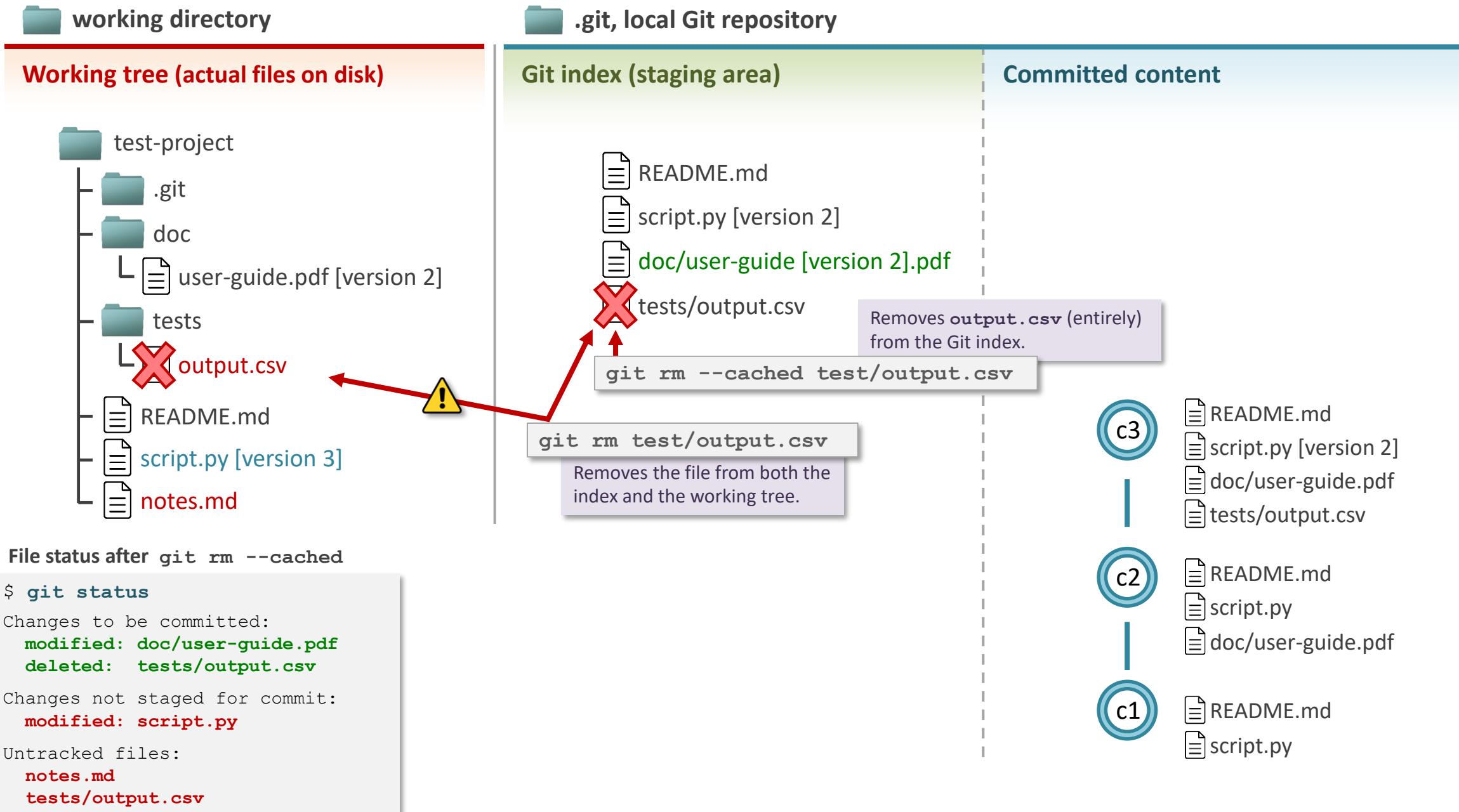


# Editing the Git index (staging area)

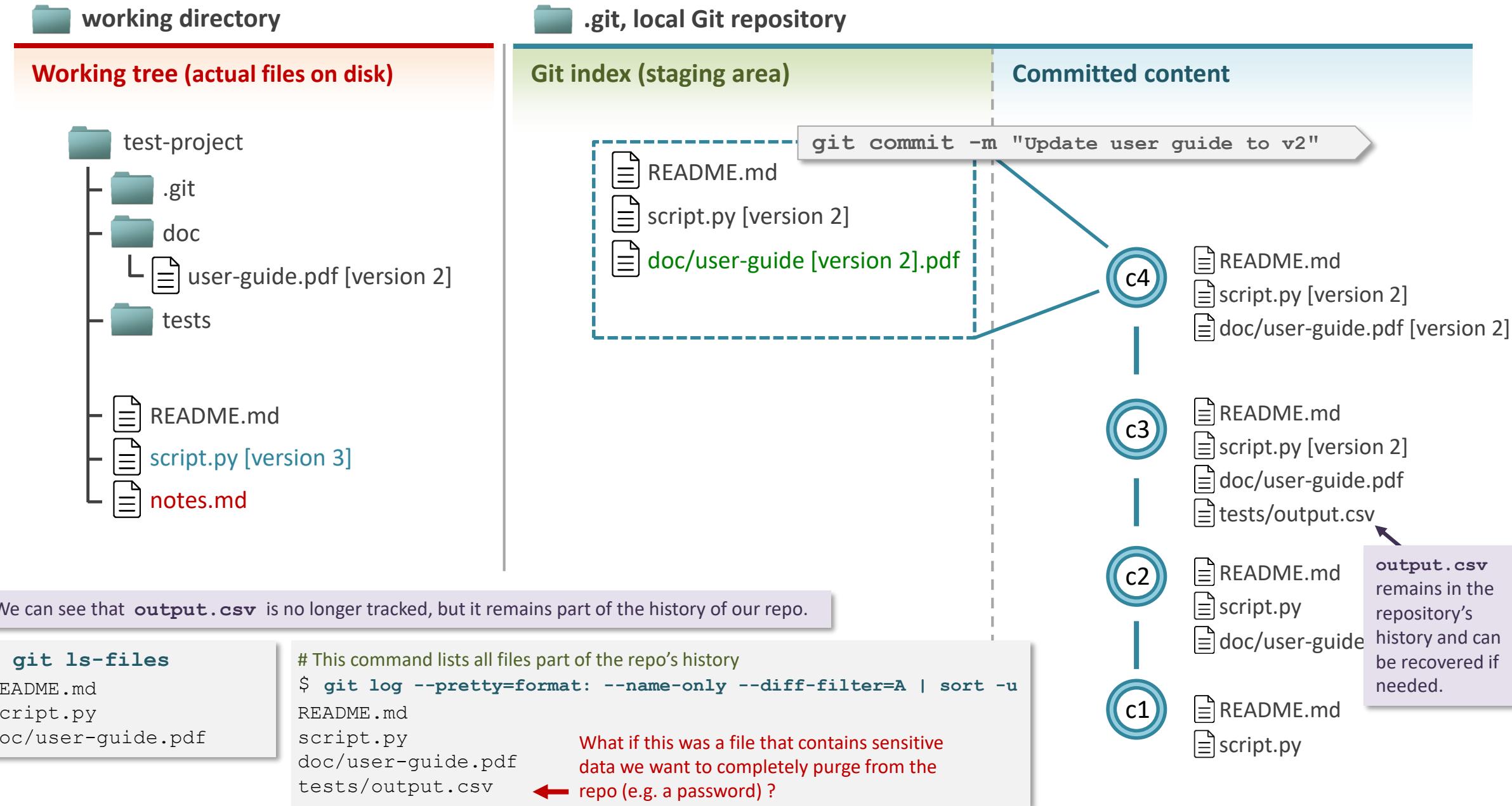
## Removing content from the Git index: example

Scenario: an update was made to `user-guide.pdf` and `script.py`. We want to commit the new version of `user-guide.pdf` (version 2), but not the changes to `script.py` and not `notes.md`.

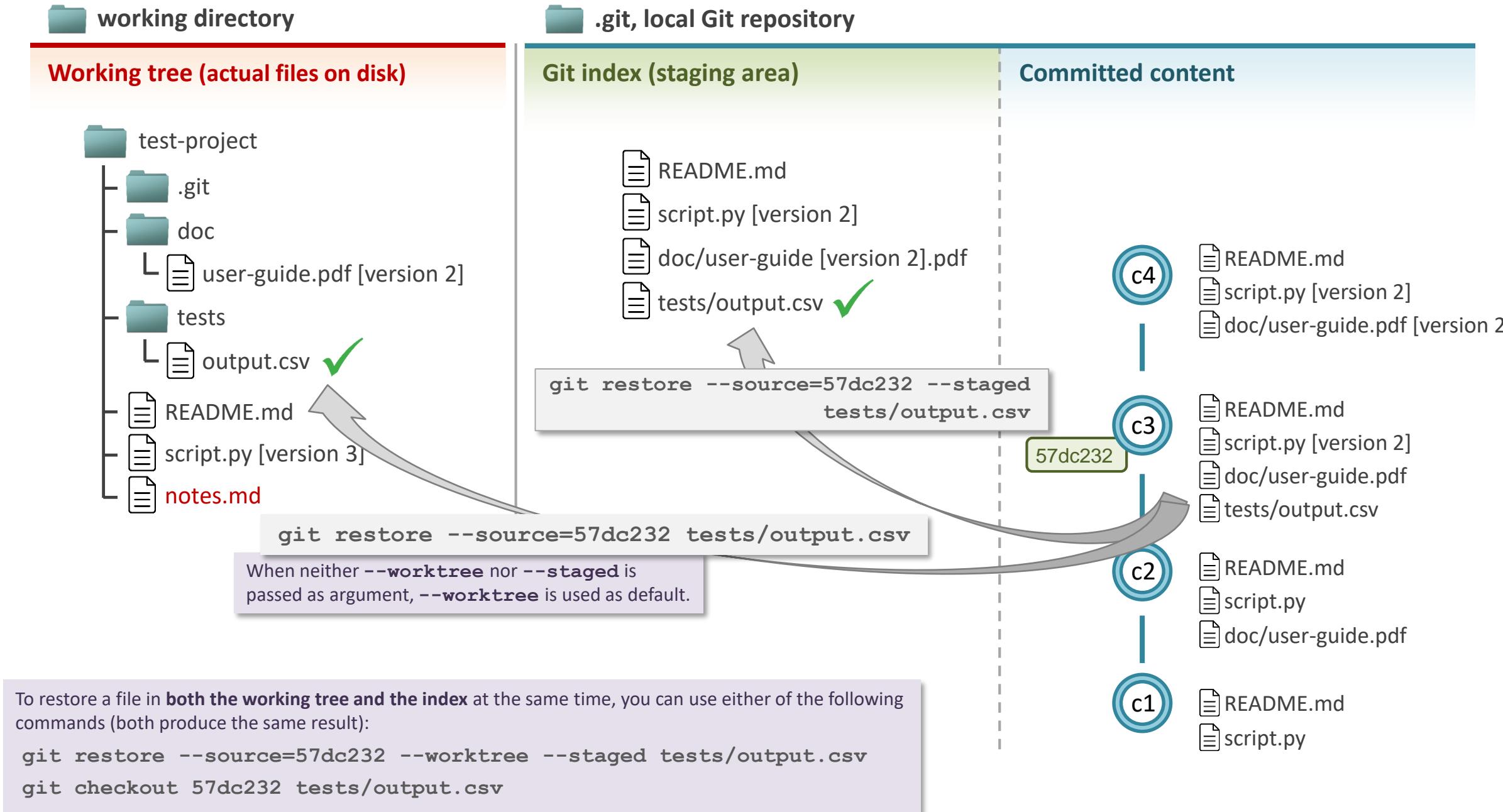




## Removing content from the Git index: example



I would now like to retrieve the file `output.csv` from the Git repository.





# OMG ! How will I remember all these fantastic commands ?

The `git status` command provides helpful hints on how to stage/unstage files.

```
$ git status
On branch main
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
    modified:   user-guide.pdf

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
    modified:   script.py

Untracked files:
  (use "git add <file>..." to include in what will be committed)
    notes.md
    tests/output.csv
```

**Warning:** without the `--staged` option, `git restore` will **reset (overwrite)** the file in the **working tree** with the version of the file from the Git index.

Only run it if you intend to delete the current version of your file – there is no "undo".



# Summary: removing/renaming files from/in the Git index

- Remove newly staged content from the index (one file at a time).

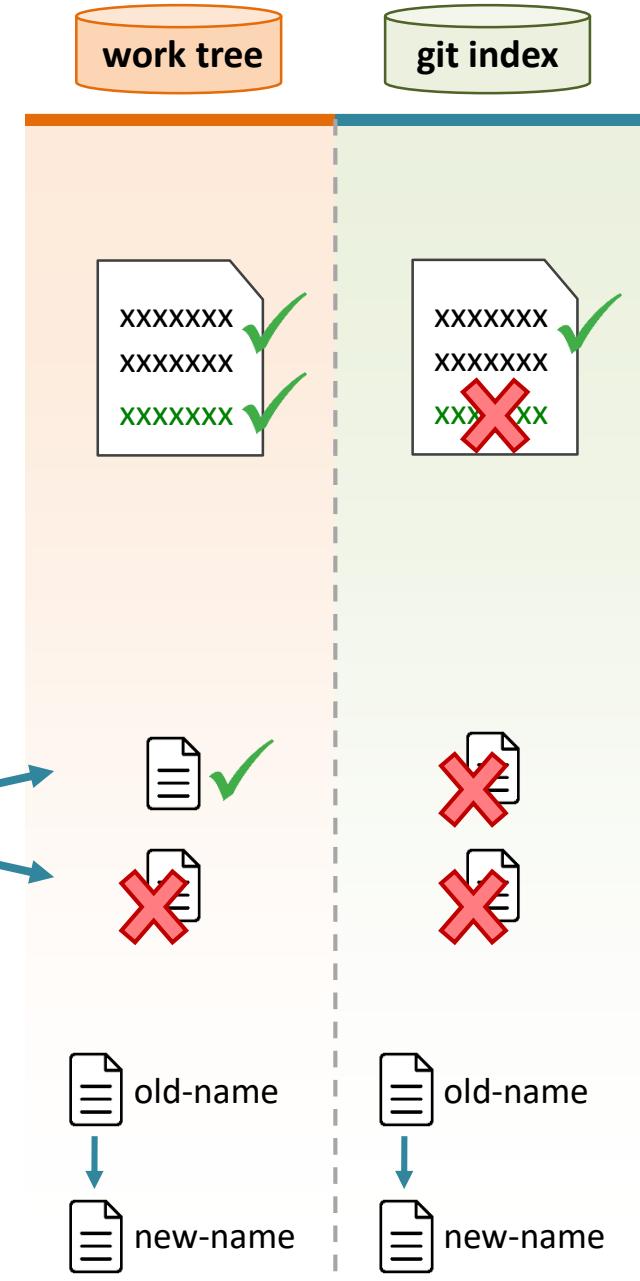
```
git restore --staged <file> # Remove newly staged content of the specified file.
```

⚠ Without the `--staged` option => resets file in work tree to the its version in the Git index.

The same can also be achieved using the `git reset` command. This is a specific use of the `reset` command, which has a wider scope.

```
git reset HEAD <file> # Remove newly staged content of a specific file.  
git reset HEAD # Remove all newly staged content (all files).
```

Useful to unstage all changes in a single command.



- Delete entire files from the index and the working tree.

```
git rm --cached <file> # Delete file from index only.  
git rm <file> # Delete file from both index and working tree.
```

⚠ Without the `--cached` option => deletes file in working tree (i.e. on disk) !

- Rename and/or move files both in the working tree and the Git index.

```
git mv <file> <new location/new name>
```

# ignoring untracked files

.gitignore

.git/info/exclude

# Ignoring files

- By default, files that are not added to a Git repo are considered **untracked**, and are always listed as such by `git status`.
- To stop Git from listing files as **untracked**, they can be added to one of the following "ignore" files:

## .gitignore

- For files to be **ignored by every copy of the repository**.
- Usually a single `.gitignore` file, placed at the root of the working tree is sufficient, but multiple files can be used (see next slide).
- `.gitignore` is meant to be shared, so track it in your repo.
- Examples of files to ignore:
  - outputs of tests
  - `.Rhistory`, `.RData`
  - `.pyc` (compiled version of python code)

Example of a `.gitignore` file

```
my_tests.py
.Rdata
.Rhistory
*.pyc
test_outputs/
```

Most of the time, this is  
the method you will want  
to use to ignore files.

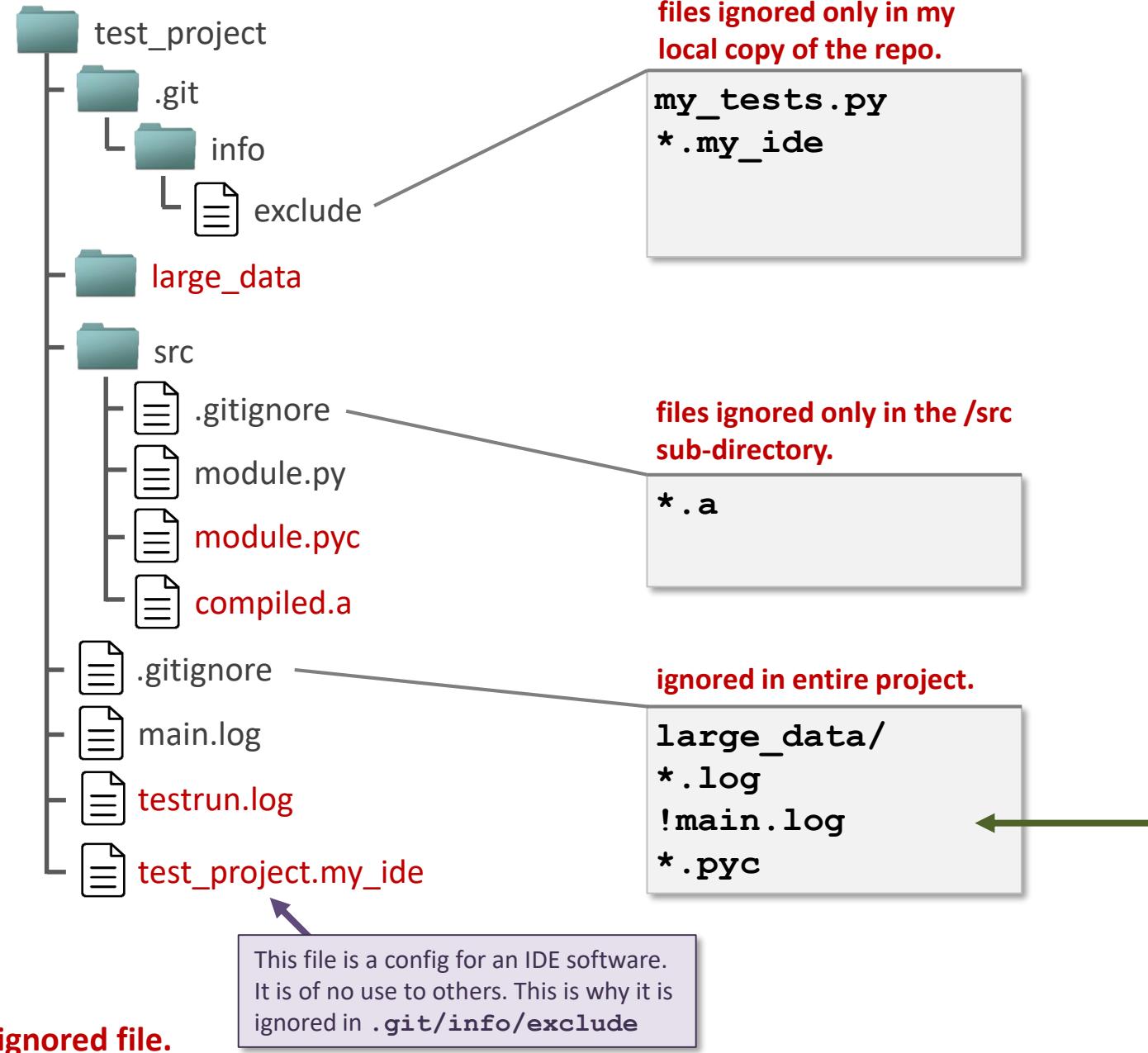
- Files to ignore are added by manually editing the two above-mentioned files.
- Files can be ignored based on their full name, or based on glob patterns (see next slide for examples).
  - `*.txt` ignore all files ending in ".txt"
  - `*.[oa]` ignore all files ending either in ".o" or ".a"
  - `logs/` appending a slash indicates a directory. The entire directory and all of its content are ignored.
  - `!dontignorethis.txt` adding a ! in front of a file name means it should not be ignored (exception to rule).

## .git/info/exclude

- For files that should be **ignored only by your own local copy of the repository**.
- Not versioned and not shared.
- Examples of files to ignore:
  - Files with some personal notes.
  - Files specific to your development environment (IDE).

Use this method for **special cases** where a file should **only be ignored in your local copy of the repo**.

# Ignoring files: example



- There can be multiple `.gitignore` files per project, to create custom per-directory ignore rules.
- Ignore rules in sub-directories are inherited from the `.gitignore` of their parent directory(ies).
- The `.gitignore` files themselves should not be ignored: add them to the Git repo so they are tracked.

- **Order (sometimes) matters:** here the rule to not ignore `main.log` must be placed after the general rule to ignore `*.log` files.

## Demo

- Ignoring files with `.gitignore`

# exercise 1 – part B and C

Your first commit

# A detailed look at **commits**

# hashing algorithms

Hashing algorithms, such as SHA-1 or SHA-256 (**Secure Hashing Algorithm**) turns a binary input into an (almost) unique hexadecimal **hash/checksum value**.

hexadecimal = base 16 number (0-9 + a-f)

SHA-1 hashes are 40 characters long. SHA-256 are 64 characters long.

**e83c5163316f89bfbdde7d9ab23ca2e25604af290**

Example of a SHA-1 hash value.

- Important: hashing a given input always outputs the exact same and (almost) unique hash.
- Example: running "This is a test" through the SHA-1 algorithm, will always produce the hash shown below:

`echo "This is a test" | openssl sha1`



**3c1bb0cd5d67dddc02fae50bf56d3a3a4cbc7204**

`echo "This is a Test" | openssl sha1`



**7500c6645cb9cdb20b32002cb82bbe067cc77d6e**

Notes:

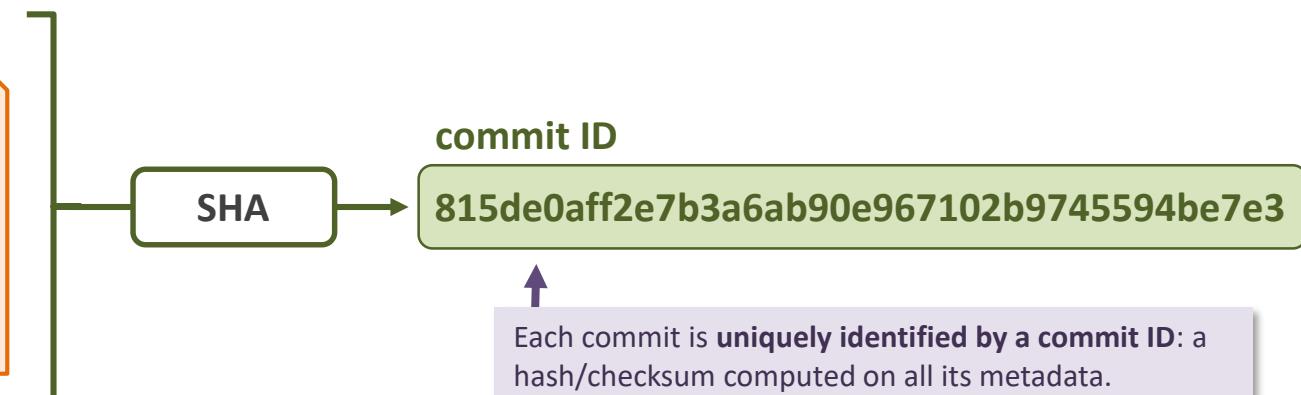
- With current hardware, SHA-1 collisions can be reasonably easily created. SHA-1 is no longer considered secure for cryptographic purposes, but is good enough for usage in Git. It is also fast to compute.
- Recent Git versions can also use SHA-256 instead of SHA-1 as hashing algorithm, and starting with version 2.51, Git uses SHA-256 by default.

# Commits: immutable snapshots of a repository's state

- A commit represents the **state of a repository at a given time** => **snapshot of Git index + metadata**.
- A commit is **the only way to enter a change** into a Git repository.  
This **enforces accountability** as you cannot have untraceable modifications.
- Each commit has an associated author, committer, commit message and date - this **enforces documentation**.
- Commits are **lightweight**:
  - They **do *not* contain the tracked files' data**, only a **reference to the data** (specifically, a **Tree\*** object that represents the state of the Git index at the time the commit was made).
- Commits contain a **reference to their parent** commit.

## Content of a commit

Author: Mad Hatter  
Committer: Alice  
Commit msg: Fix bug in CheshireCat()  
Date: 24.02.2020 10:43  
Tree: `e5d56fa`  
Parent: `57dc232`

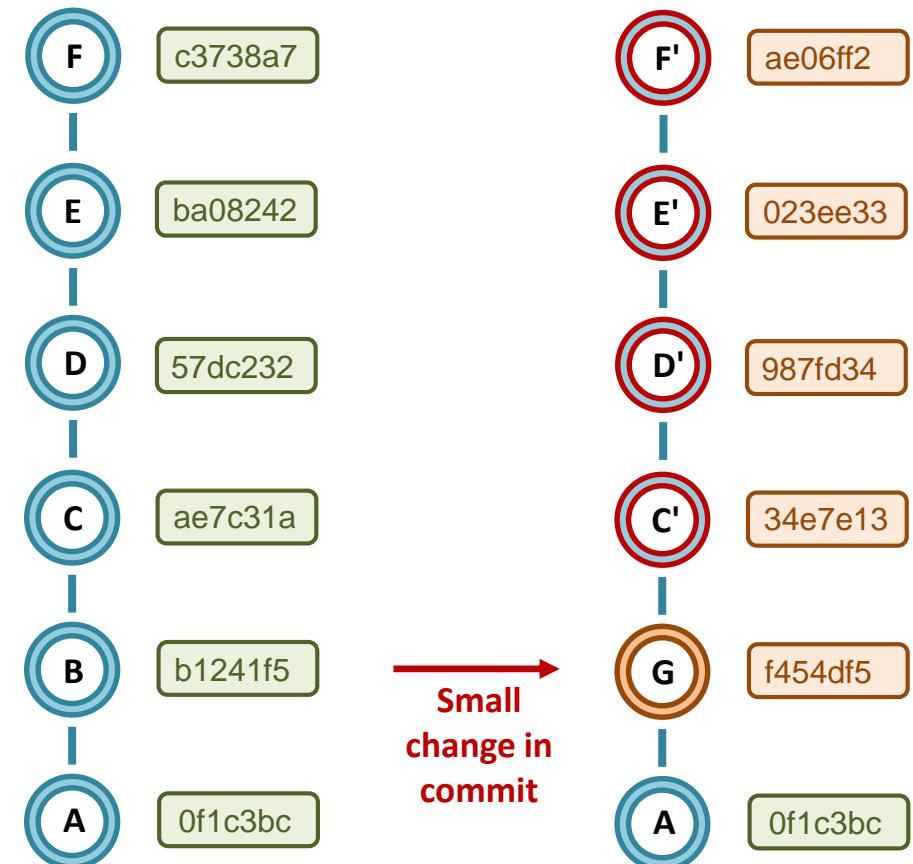


\* Tree = reference to the state of all files at a given time point = snapshot of repository state.

- Because of how a commit ID is computed, **commits are immutable**: once a commit is made, it cannot be modified without its commit ID being modified too - which would then make it a different commit !
- Modifying a commit** will modify all of its descendants. It **creates a completely new history** of the Git repo.
- This ensures the **integrity of a Git repository's history**, something that **is important due to the distributed nature of Git**. It can be seen as a sort of blockchain.

Examples of things that change a commit's ID:

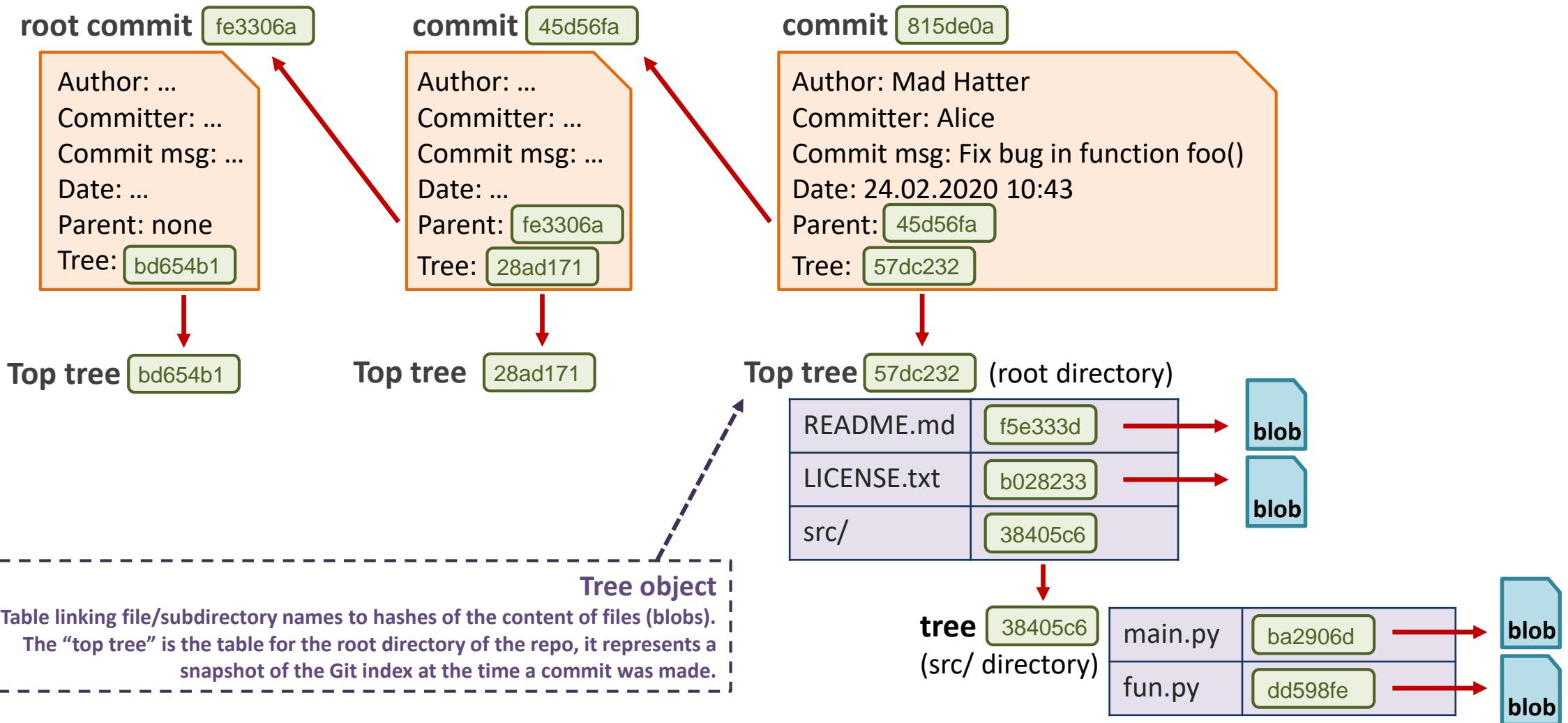
- Changing the content of a file.
- Changing the time a commit was made.
- Changing the parent commit of a commit.



- Commits contain a reference to the top “Tree object”** – a table linking file names and hashes of the Git index at the time the commit was made. This is a “snapshot” of the index, and is how Git can retrieve the state of every file at a given commit.
- Commits point to their direct parent** – forming a DAG (directed acyclic graph) where no commit can be modified without altering all of its descendants.

If two commits have the same ID, their content is identical !

If two commits have the same ID, their entire history is identical !



# Git versioning

- Git stores a complete copy of each file's version\*.
- Optimized for speed rather than disk space preservation.
- Sub-optimal for tracking large files, as they will quickly inflate the size of the `.git` repo.

As counter-intuitive as it may sound, Git stores a complete copy of each file version. Not just a diff.

What ??

Yes! It may not be space efficient, but it's fast :-)

## most VCS versioning

```
--- version2 diff
+++ version3 diff
+ Yes! It may not be space
+ efficient, but it's + fast :-)
```

```
--- version1 diff
+++ version2 diff
+ What ??
```

### version1

As counter-intuitive as it may sound, git stores a complete copy of each file version. Not just a diff.



## Git versioning

### version3

As counter-intuitive as it may sound, Git stores a complete copy of each file version. Not just a diff.

What ??

Yes! It may not be space efficient, but it's fast :-)

SHA1 – e78bf23...

### version2

As counter-intuitive as it may sound, Git stores a complete copy of each file version. Not just a diff.

What ??

SHA1 – 8fb24d3...

### version1

As counter-intuitive as it may sound, Git stores a complete copy of each file version. Not just a diff.

SHA1 – 27da79b...

\* At least for a while - at some point Git also stores things as diffs, see "packfiles".

## Git packfiles: compressing old history

- For older commits, Git uses a few tricks to decrease disk space usage:
  - Differences between similar files are stored as diffs.
  - Multiple files are compressed into a single “packfile” (`.pack` extension).
  - Each packfile has an associated packfile index (`.idx` extension), that associates filenames to blobs.

# The **HEAD** pointer

## HEAD: a pointer to the most recent commit on the currently active branch

Looking at the output of `git log`, we see a **HEAD ->** label: this shows the position of the **HEAD** pointer.

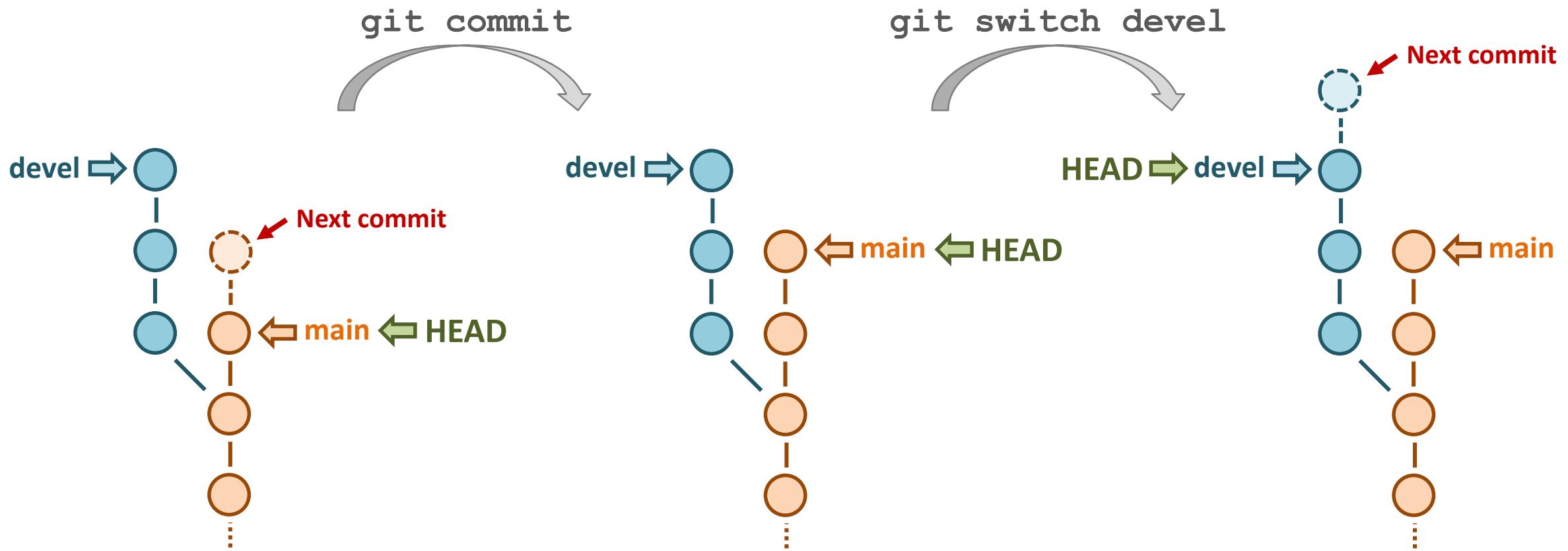
### Commit ID (hash)

Here shown in a shortened  
form (7 first chars).

```
git log --all --decorate --oneline --graph
* 351dca6 (HEAD -> main, origin/main, origin/HEAD) peak_sorter: added authors to script
* f3d8e22 peak_sorter: display name of highest peak when script completes
* 076aa80 (origin/feature-dahu, feature-dahu) peak_sorter: display highest peak at end of script
* d29958d peak_sorter: added authors as comment to script
* 6c0d087 peak_sorter: improved code commenting
* 89d201f peak_sorter: add Dahu observation counts to output table
* 9da30be README: add more explanation about the added Dahu counts
* 58e6152 Add Dahu count table
/
* cfd30ce Add gitignore file to ignore script output
* f8231ce Add README file to project
* 8e0d4fe (origin/dev-jimmy) peak_sorter: add check that input table has the ALTITUDE and PEAK columns
* ff85686 Ran script and added output
/
* 821bcf5 peak_sorter: add +x permission
* 40d5ad5 Add input table of peaks above 4000m in the Alps
* a3e9ea6 peak_sorter: add first version of peak sorter script
```

# HEAD: a pointer to the currently checked-out branch/commit

- **HEAD** is – most of the time – a pointer to the latest commit on your current branch.  
(Sometimes it is also described as a pointer to the current branch – which is itself a pointer to the latest commit on the branch)
- The **HEAD** position is how Git knows what is the currently “active” branch.
- New commits are always added “under” the current **HEAD** (the new commit is the “child” of the commit pointed-to by **HEAD**).
- When a new commit is added, **HEAD** (and the branch it points to) is automatically moved to point to that new commit.



Another way to look at it, is that **HEAD** always points to the parent of your next commit.

## Relative references to commits

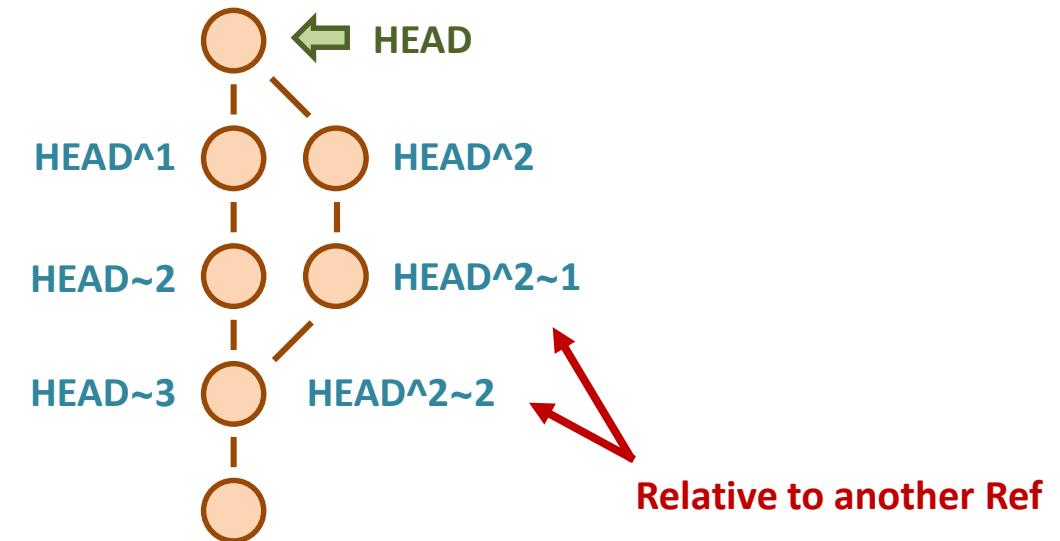
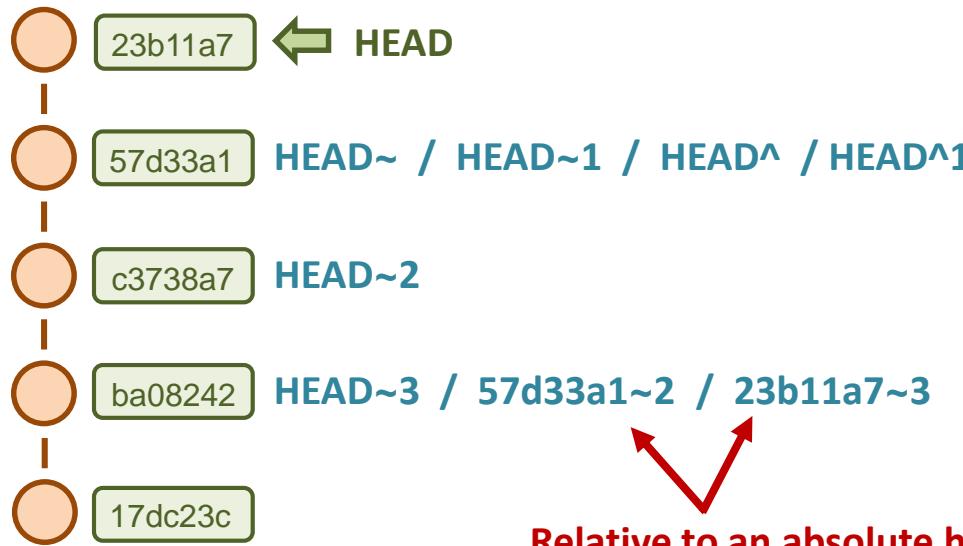
- Using `~` and `^` symbols, Git allows to refer to a commit by its position relative to another commit, rather than by its absolute hash.
- `Ref` can be any reference, such as `HEAD`, a commit hash, a branch name, or even another Ref.

`Ref~X` refers to the **Xth generation before** the commit: `~1` = parent, `~2` = grand-parent, etc.

`Ref~` is a shortcut for `Ref~1`

`Ref^X` refers to the **Xth direct parent** of the `HEAD` commit (but most commits have only a single parent).

`Ref^` is a shortcut for `Ref^1`



## Part II

# Git branches

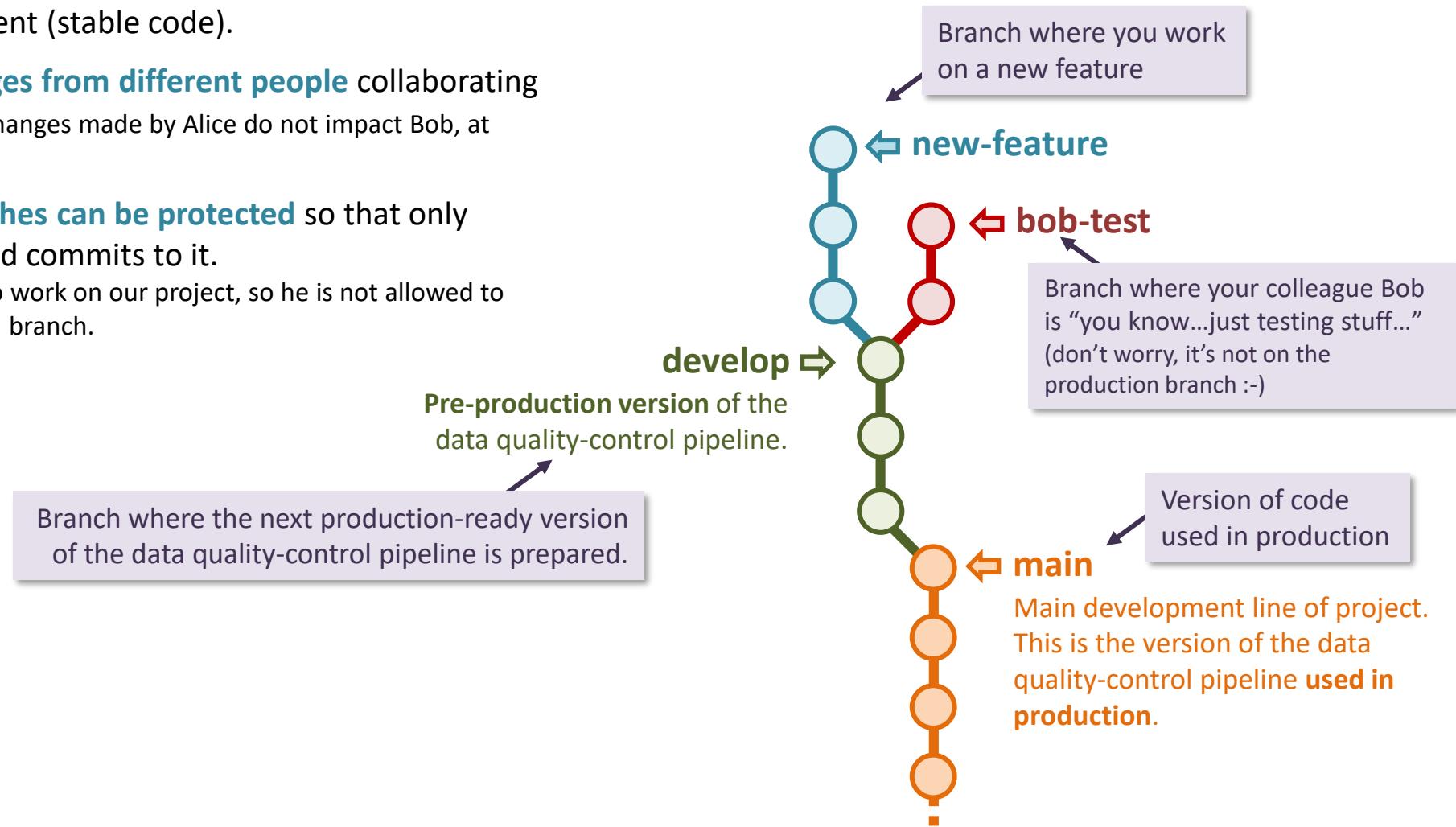
Managing multiple lines of development

# Why branches? An illustration with a data quality-control pipeline project

“Branching” means to **diverge from the main line of development**.

- Branches **isolate new changes** (work in progress) from the main line of development (stable code).
- Branches **isolate changes from different people** collaborating on a same project (so changes made by Alice do not impact Bob, at least not immediately).
- On online repos, **branches can be protected** so that only selected people can add commits to it.

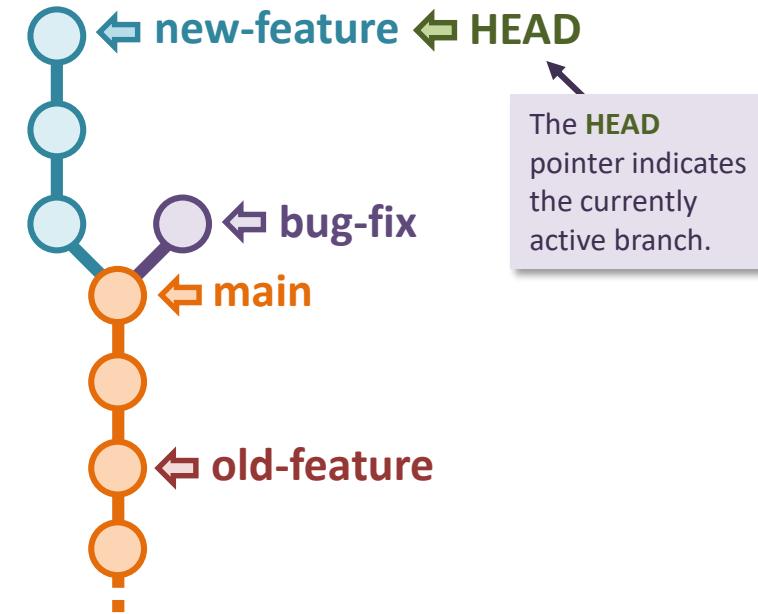
Use case: Bob just started to work on our project, so he is not allowed to make changes to the “main” branch.



Git is **designed to encourage branching**: branches are “cheap” (don’t take much disk space) and switching between them is fast.

# What are branches?

- A branch is just a pointer to a commit.
- A branch is **very lightweight** (41 bytes).
- By convention, the **main** branch is the branch representing the stable version of your work.
- To know which is the currently active branch, Git uses the **HEAD** pointer. The **HEAD** pointer always points to the currently active branch (except for the special case of “detached HEAD” mode, discussed later in the second part of this course).
- New commits are always added at the top of the currently active branch\* (except for the special case of “detached HEAD” mode).



The **main** branch is no special branch. It is simply the default name given to the branch created when initializing a new repo [`git init`]. It has become a convention to use this branch as the stable version of a project.

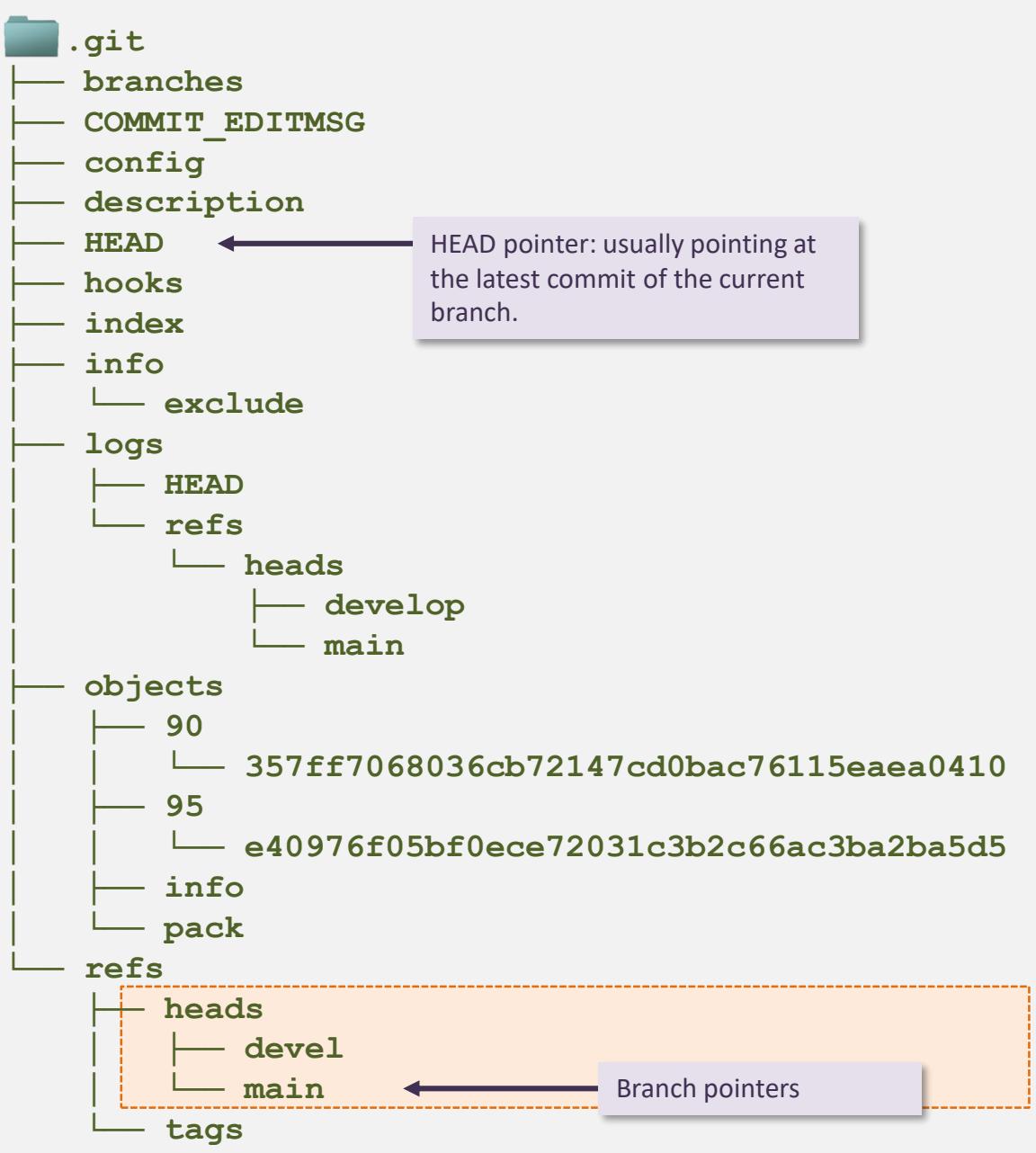
Note: in earlier versions, the “**main**” branch used to be called the “**master**” branch.



## Illegal characters in branch names

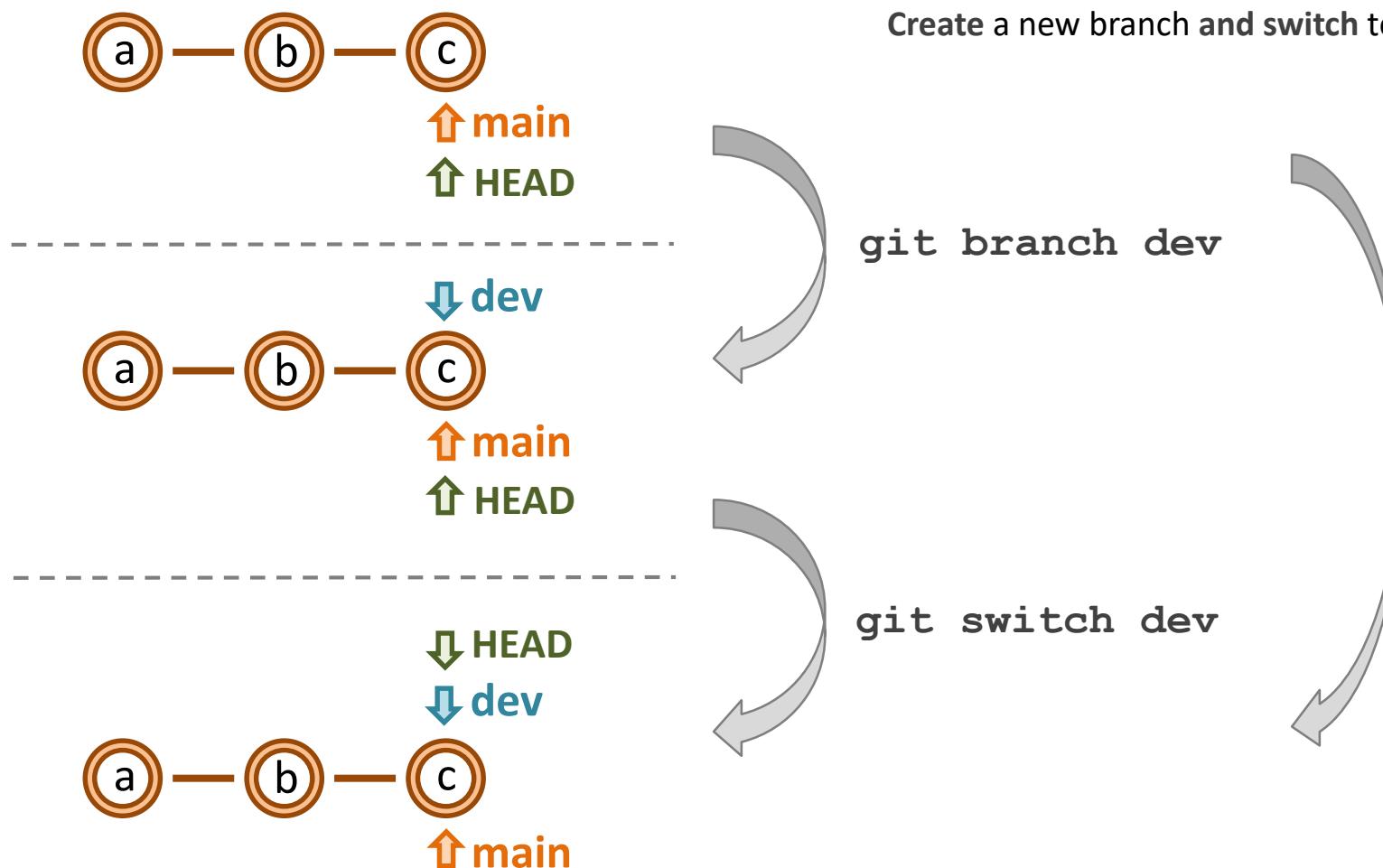
Spaces and some characters such as `, ~^: ?* [ ] \` are not allowed in branch names. It is strongly recommended to stick to lowercase letters, numbers and the “dash” character `[ - ]`.

## Behind the scenes: branches are lightweight pointers to commits



```
> cat .git/refs/heads/main  
8508bc698498861c036636dba40ac28b6c7f3a7a  
  
> cat .git/refs/heads/devel  
4aefde0735e0f95de9969fa660265f71d6a95ebd  
  
> cat .git/HEAD  
ref: refs/heads/main
```

## Create and switch branches



Create a new branch: `git branch <branch name>`

Switch to another branch: `git switch <branch name>`

Create a new branch and switch to it: `git switch -c <branch name>`

The **-c** option is to create and switch to the new branch immediately.

`git switch -c dev`

### switch vs. checkout

On older Git versions the `git switch` command does not exist.

Instead, `git checkout` is used to switch branches:

`git checkout <branch name>`  
`git checkout -b <branch name>`

The `git switch` command was introduced in Git version 2.23 as a replacement to `git checkout` for switching branches. This was done because the `checkout` command already has other uses (e.g. to extract older files from the Git database), and it was deemed confusing that a same command would have multiple usages. It remains nevertheless possible to switch branches with the `git checkout` command in recent Git versions.

## Create and switch branches (continued)

- By default new branches are created at the current position of the **HEAD** pointer (i.e. the current commit).
- But they can be created at any specified reference.

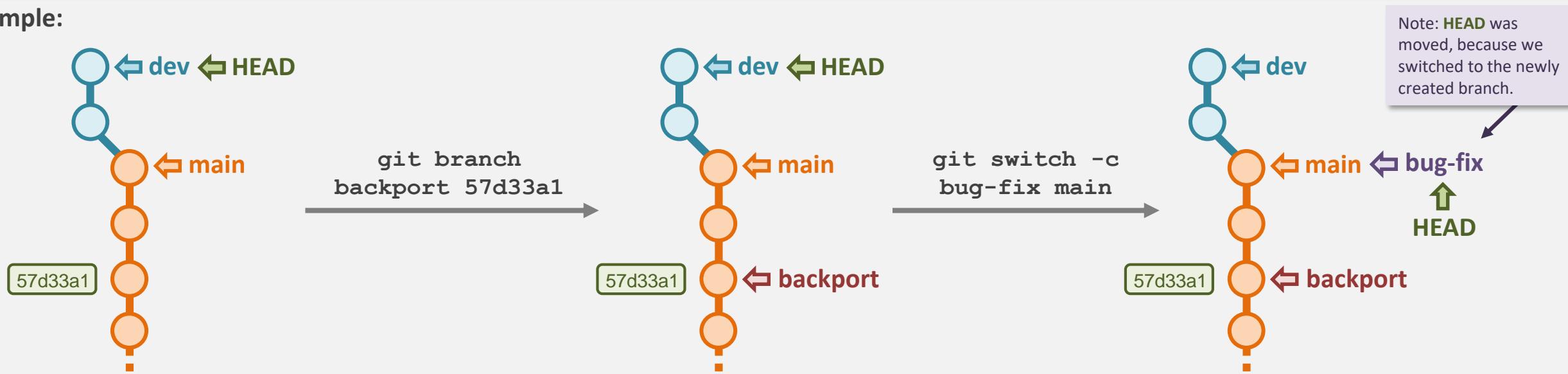
Reference to a commit, branch or tag.  
The default reference is **HEAD**.

↓

Create a new branch: `git branch <branch name> <reference>`

Create a new branch and switch to it: `git switch -c <branch name> <reference>`

### Example:



## List branches and identify the currently active branch

`git branch`

List local branches

`git branch -a`

List local and remote branches

### Examples

```
> git branch  
devel  
* main  
new-feature
```

The \* denotes the currently checked-out (active) branch. Generally displayed in green.

```
> git branch -a  
devel  
* main  
new-feature  
remotes/origin/main  
remotes/origin/devel
```

Remote branches (to be precise, pointers to remote branches) are shown in red and are named `remotes/<remote name>/<branch name>`

A convenient alternative: `git adog (git log --all --decorate --oneline --graph)` will also show all branches.

The currently active branch can be identified as it has the `HEAD` pointing to it.

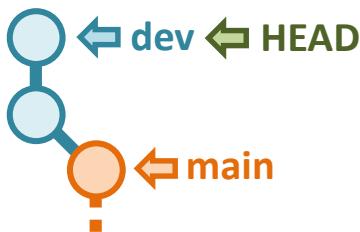
```
* 351dca6 (HEAD -> main, origin/main, origin/HEAD) peak_sorter: added authors to script  
* f3d8e22 peak_sorter: display name of highest peak when script completes  
| * 076aa80 (origin/feature-dahu, feature-dahu) peak_sorter: display highest peak at end of script  
| * d29958d peak_sorter: added authors as comment to script  
| * 6c0d087 peak_sorter: improved code commenting  
| * 89d201f peak_sorter: add Dahu observation counts to output table  
| * 9da30be README: add more explanation about the added Dahu counts  
| * 58e6152 Add Dahu count table  
/  
* cfd30ce Add gitignore file to ignore script output
```

## Demo: git switch

- What happens in the working directory when switching branches

## What happens in the working tree when switching branches

- When switching to different branch, the content of your working directory (working tree) is updated as to reflect the state of the commit the active branch (i.e. the branch you just switched to).
- This means that when switching branches, you can have files appear/disappear or be modified in your working directory.
- A copy of committed files is kept at all times in the `.git` repository, so they can be restored when switching branches.



```
> ls -l
4096 Jan 29 22:45 user_guide.md
108 Jan 29 22:30 personal_notes.md
53 Jan 29 22:30 README.md
77 Jan 29 22:45 script.py
```

#! /usr/bin/env python3  
print("Hello World")  
print("Git branches are great")

Untracked files (here in red) are unaffected by branch switches.



```
> ls -l
108 Jan 29 22:30 personal_notes.md
53 Jan 29 22:30 README.md
45 Jan 29 22:43 script.py
```

#! /usr/bin/env python3  
print("Hello World")

Size and last modified date have changed!

What has changed:  
• `user_guide.md` has disappeared...  
• `script.py` was reverted to the older version...

`git switch main`



```
> ls -l
4096 Jan 29 22:45 user_guide.md
108 Jan 29 22:30 personal_notes.md
53 Jan 29 22:30 README.md
77 Jan 29 22:45 script.py
```

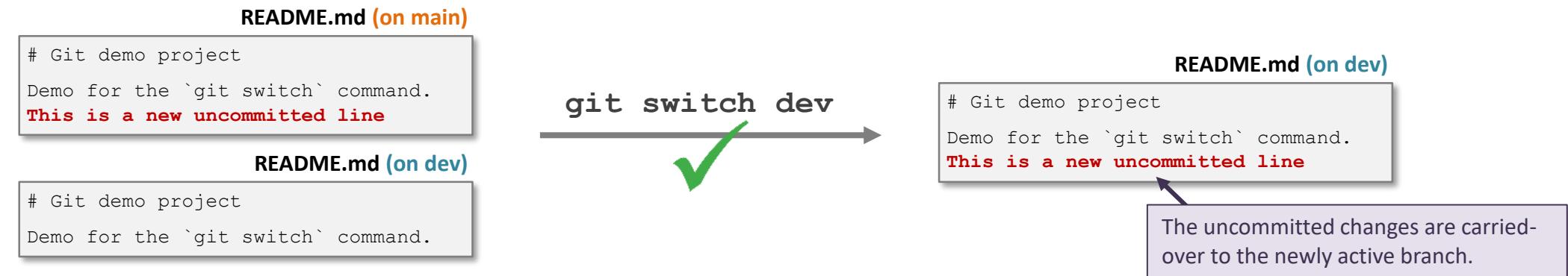
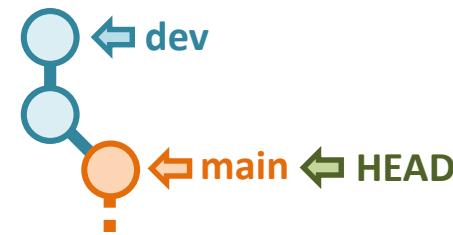
What has changed:  
• `user_guide.md` is back.  
• `script.py` reverted to newer version.

`git switch dev`

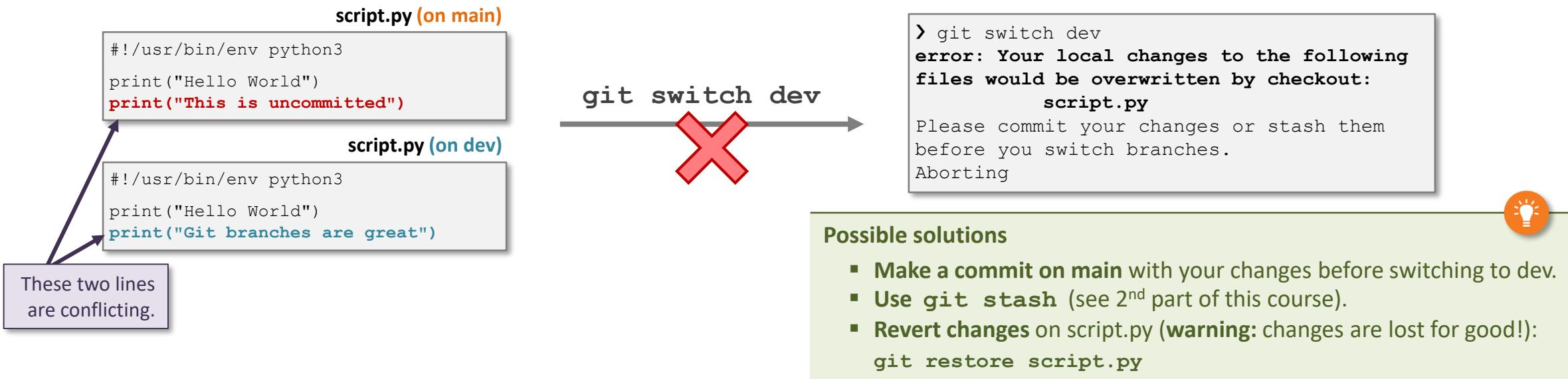
## What happens in the working tree when switching branches

What if you have uncommitted changes ?

- If the changes do not conflict between the branches, they are “carried-over” with the switch.



- If the changes conflict between the branches, Git will not allow you to switch.



# git merge

get branches back together

## Merging branches

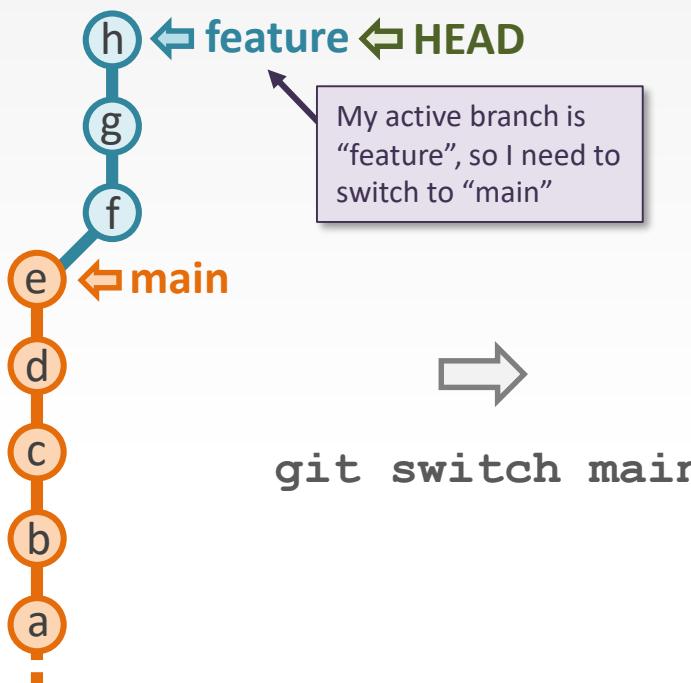
- **Merge:** incorporate changes from the specified branch into the currently active (checked-out) branch.

```
git merge <branch to merge into the current branch>
```

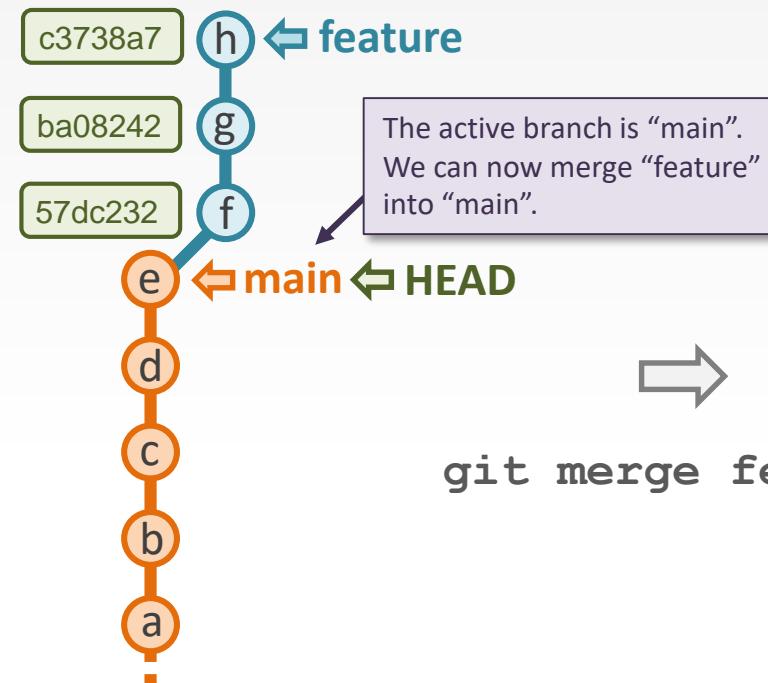


**Before** running the command, make sure that the branch into which the changes should be merged is the currently active branch. If not, use `git switch <branch>` to checkout the correct branch.

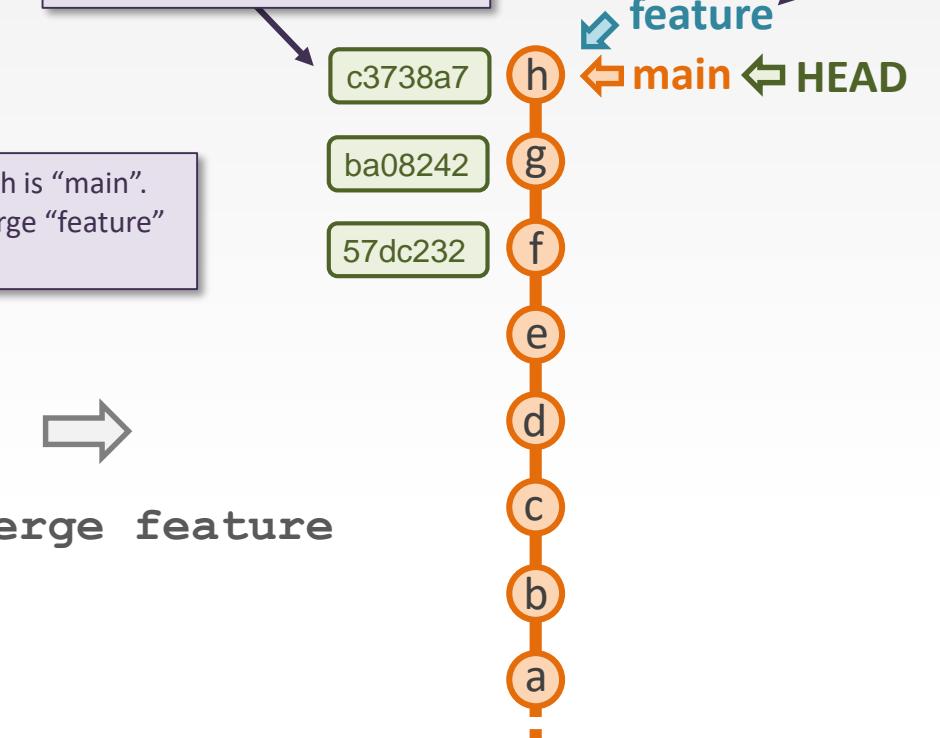
**Example:** merge changes made on branch **feature** into the branch **main**



```
git switch main
```



## git merge feature



Merging has not made any changes to my commit history. All my commits remain the same (no change in hash).

At this point, the "feature" branch can be deleted.  
**git branch -d feature**

## Two types of merges

- **Fast-forward merge:** when branches have not diverged.

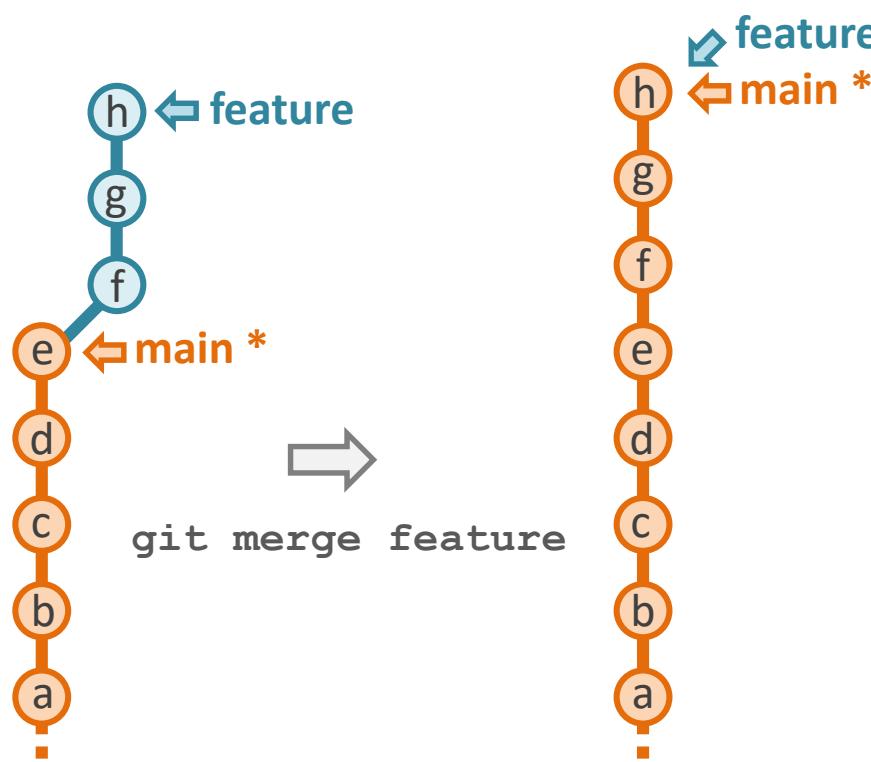
The branch that is being merged (here **feature**) is rooted on the latest commit of the branch that it is being merged into (here **main**).

- **3-way merge:** when branches **have diverged**. This introduces an extra “merge commit”.

The **common ancestor** of the 2 branches is not the last commit of the branch we merge into (here **main**).

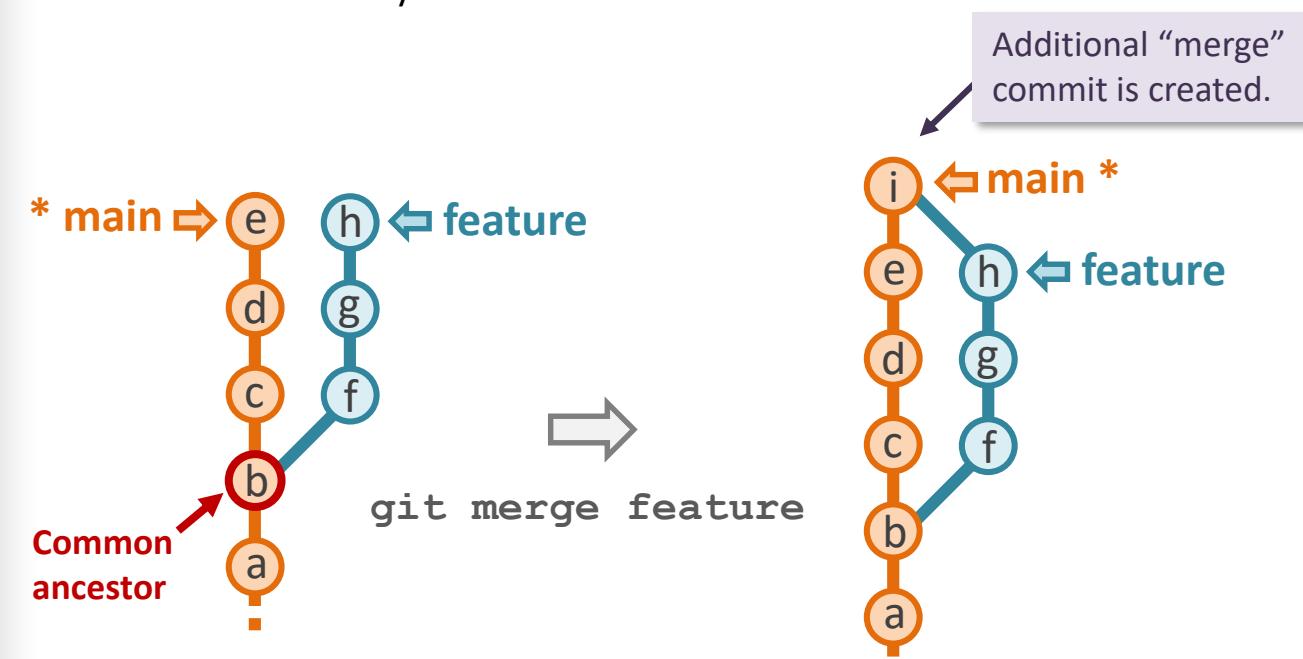
### Fast-forward merge

- Guaranteed to be conflict free.



### 3-way merge (non-fast-forward)

- Creates an additional “merge commit” (has 2 parents).
- Conflicts may occur.



\* denotes the currently active (checkout-out) branch.

## Conflicts in 3-way merges (non fast-forward)

If a same file is modified at (or around) the same place in the two branches being merged, Git cannot decide which version to keep. There is a conflict, and you need to manually resolve it.

### README.md version of main branch.

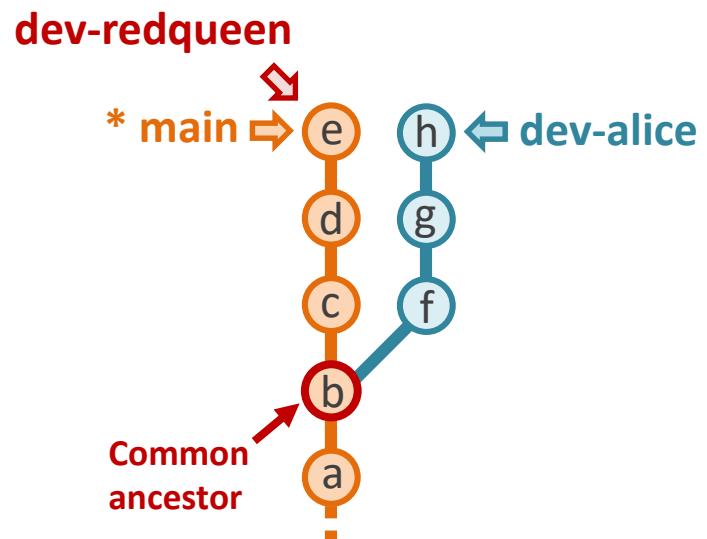
```
# Tea pot quality-control pipeline
Check and approve tea pots for use in
unbirthday parties.
```

**Authors:** Mad Hatter, Red Queen  
**Date modified:** 2022 Oct 10

```
## Step 1: physical integrity check
* Check exterior for cracks and uneven
  painting.
* Check for mice inside of pot.
* Verify the Mad Hatter is on time.
```

```
## Step 2: tea-brewing integration test
* Brew tea for 7 min.
* Add 2 cubes of sugar.
* Probe tea.
```

**Story background:** the Red Queen has just merged changes from her branch **dev-redqueen** into **main**.  
Now Alice wants to merge her branch **dev-alice** into **main**.



Let's merge dev-alice into main...

```
$ git merge dev-alice
```

Auto-merging README.md

**CONFLICT (content): Merge conflict in README.md**

Automatic merge failed; fix conflicts and then commit the result.

### README.md version of dev-alice branch.

```
# Tea pot quality-control pipeline
Check and approve tea pots for use in
unbirthday parties.
```

**Authors:** Mad Hatter, Alice  
**Date modified:** 2022 Oct 11

```
## Step 1: physical integrity check
* Check exterior for cracks and uneven
  painting.
* Check for mice inside of pot.
```

```
## Step 2: tea-brewing integration test
* Brew tea for 7 min.
* Add 2 cubes of sugar.
* Probe tea.
* Make sure we still have no idea why
  a raven is like a writing desk.
```

# Resolving conflicts

1. Open the conflicting files in the text editor of your choice.

2. Look for the text between <<<<< and >>>>> .

There can be more than one of such sections, if there is more than one conflict in the file.

- The text between <<<<< and ===== is the version of the current branch, i.e. the branch into which we merge (main, in this example).
- The text between ===== and >>>>> is the version from the branch we are merging (dev-alice, in this example).

Version from the current branch (here main).

Version from branch being merged into the current branch (here dev-alice).

Note: there is no conflict for these 2 lines, because the edits were made at different locations in the file. Git is able to auto-merge such changes.

```
# Tea pot quality-control pipeline
Check and approve tea pots for use in
unbirthday parties.
```

<<<<< HEAD

Authors: Mad Hatter, Red Queen  
Date modified: 2022 Oct 10

=====

Authors: Mad Hatter, Alice

Date modified: 2022 Oct 11

>>>>> dev-alice

## Step 1: physical integrity check  
\* Check for mice inside of pot.  
**\* Verify the Mad Hatter is on time.**

## Step 2: tea-brewing integration test  
\* Brew tea for 7 min.  
\* Add 2 cubes of sugar.  
\* Probe tea.  
**\* Make sure we still have no idea why a raven is like a writing desk.**

\$ git merge dev-alice

Auto-merging README.md

CONFLICT (content): Merge conflict in README.md  
Automatic merge failed; fix conflicts and then commit the result.

File with conflicts



3. Manually edits the file(s)...

```
# Tea pot quality-control pipeline
Check and approve tea pots for use in
unbirthday parties.
```

Authors: Mad Hatter, Red Queen, Alice

Date modified: 2022 Oct 11

## Step 1: physical integrity check  
\* Check for mice inside of pot.  
\* Verify the Mad Hatter is on time.

## Step 2: tea-brewing integration test  
\* Brew tea for 7 min.  
\* Add 2 cubes of sugar.  
\* Probe tea.  
\* Make sure we still have no idea why a raven is like a writing desk.



4. Stage the conflict-resolved file(s).  
5. Commit

Hash of the added "merge" commit.

\$ git add README.md

\$ git commit

[main a317d38] Merge branch 'dev-alice'

An editor will open with a pre-set commit message. You can accept it as is, or modify it.

## Resolving conflicts: if you get lost...

- If you are lost at some point, run `git status` and it will give you some hints and commands.
- A merge can be aborted at anytime with `git merge --abort`
- Completed merges can be reverted (with the `git reset` commands – see the “git advanced” slides).

### Examples

```
$ git status
On branch main
You have unmerged paths.
  (fix conflicts and run "git commit")
  (use "git merge --abort" to abort the merge)

Unmerged paths:
  (use "git add <file>..." to mark resolution)
    both modified: README.md
```

Git tells you what to do and reminds you of commands.

Output of `git status` before conflicts are resolved in the file.

```
$ git status
On branch main
All conflicts fixed but you are still merging.
  (use "git commit" to conclude merge)

Changes to be committed:
  modified: README.md
```

Git tells you what to do and reminds you of commands.

Output of `git status` after conflicts are resolved in the file and the file was staged.

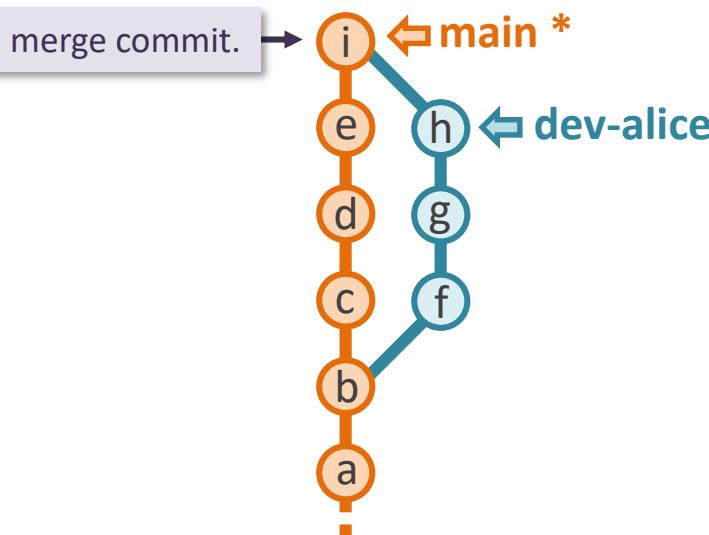
# What's in a merge commit ?

If there was no conflict, the **merge commit** contains nothing but the commit message (and other metadata).

```
$ git show HEAD
commit 10fa3ad505821b0ea628b811143af47343a4d8dc (HEAD -> main)
Merge: 7446b3e b4fb462
Author: Red Queen <off.with.their.heads@wonder.org>
Date:   Tue Oct 11 15:16:39 2022 +0200

Merge branch 'dev-redqueen'
```

If there was a conflict, the **merge commit** contains the conflict resolution changes made to the conflicted file(s).



```
$ git show HEAD
commit a317d38448dae4e6bd9b4862dcaccf4e416cc46c (HEAD -> main)
Merge: 10fa3ad 7999c7c
Author: Alice <alice@redqueen.org>
Date:   Tue Oct 11 15:27:35 2022 +0200

Merge branch 'dev-alice'

diff --cc README.md
index 647be0c,74edef5..3ce8aa7
--- a/README.md
+++ b/README.md
@@@ -1,8 -1,8 +1,8 @@@
# Tea pot quality-control pipeline
Check and approve tea pots for use in unbirthday parties.

- Authors: Mad-Hatter, Red Queen
- Date modified: 2022 Oct 10
- Authors: Mad-Hatter, Alice
++Authors: Mad-Hatter, Red Queen, Alice
+ Date modified: 2022 Oct 11

## Step 1: physical integrity check
* Check exterior for cracks and uneven
```

## Demo

- Merging branches (fast-forward and 3-way merge)

## Deleting branches

Branches that are merged and are not used anymore can (should) be deleted.

```
git branch -d <branch name>
```

← **safe option:** only lets you delete branches that are fully merged.

```
git branch -D <branch name>
```

← **YOLO option:** lets you delete any branch.

- Note: A currently active (checked-out) branch cannot be deleted. You must switch to another branch before deleting it.

### Examples

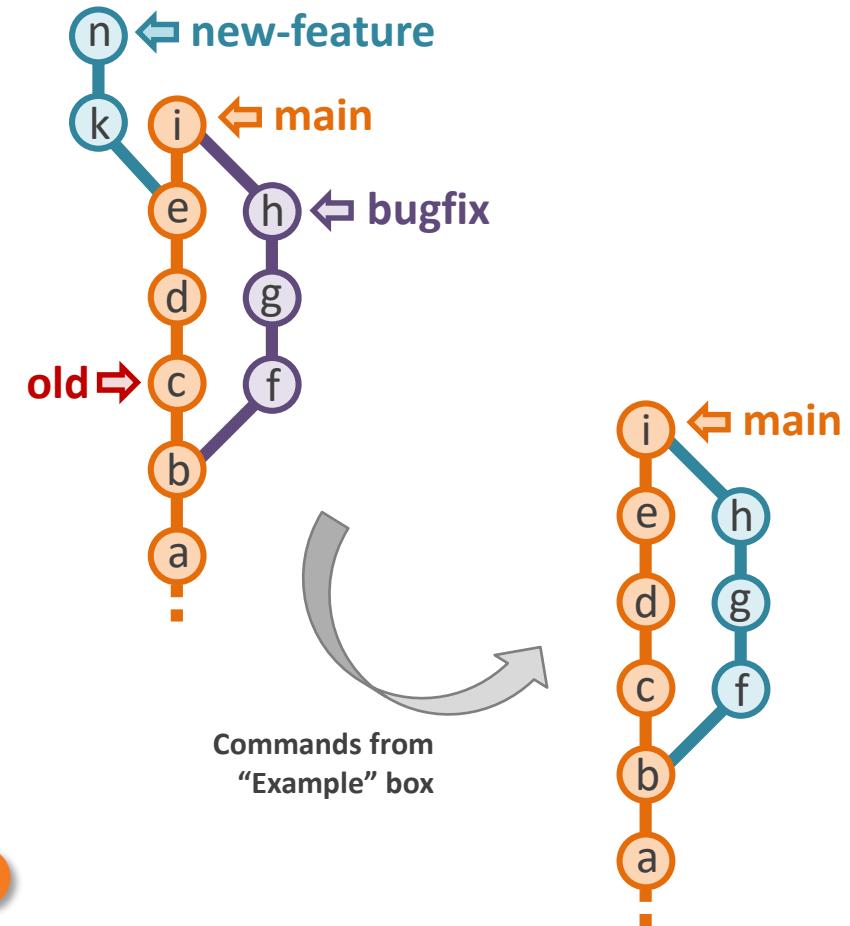
```
# The 'bugfix' and 'old' branches are fully merged. We can use -d.
> git branch -d bugfix
Deleted branch bugfix (was bd898dc)

> git branch -d old
Deleted branch old (was 75d3fed)

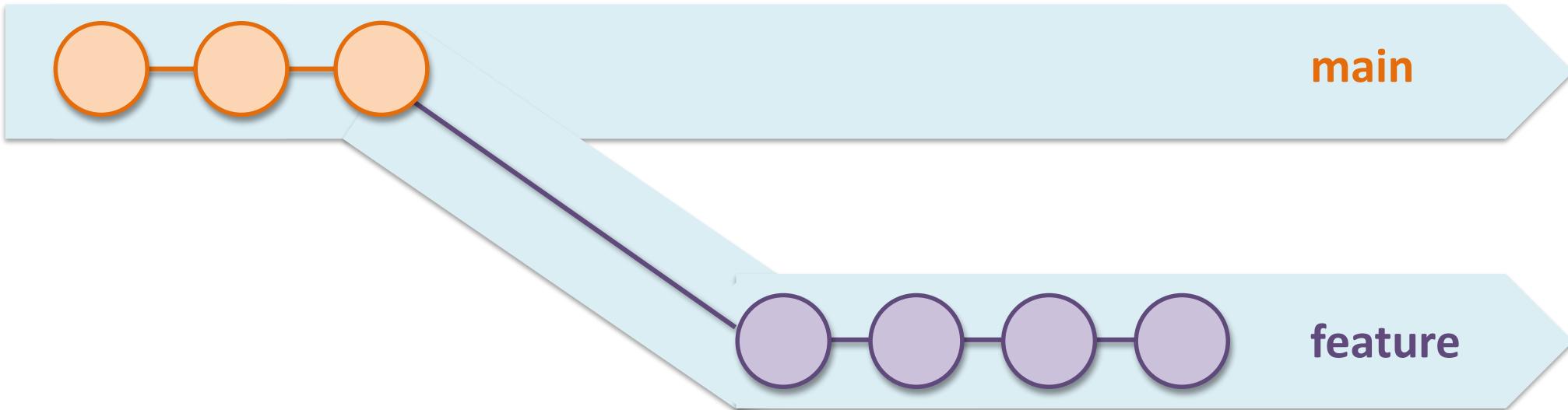
# Trying to delete a non-merged branch with -d will fail.
> git branch -d new-feature
error: The branch 'testing' is not fully merged.
If you are sure you want to delete it, run 'git branch -D testing'.

# -D allows deletion of a non-merged branch.
> git branch -D new-feature
Deleted branch new-feature (was f2a898b)
```

Deleted a branch by mistake ? – no panic !
This hash can be used to re-create it:
git branch new-feature f2a898b



## Branch management: best practices



- **Use branches** to develop and tests new changes to your code/scripts - don't test directly on main.
- **Don't hesitate to create branches**, they are “cheap” (they don't add much overhead to the git database).
- Delete branches that are no longer used.

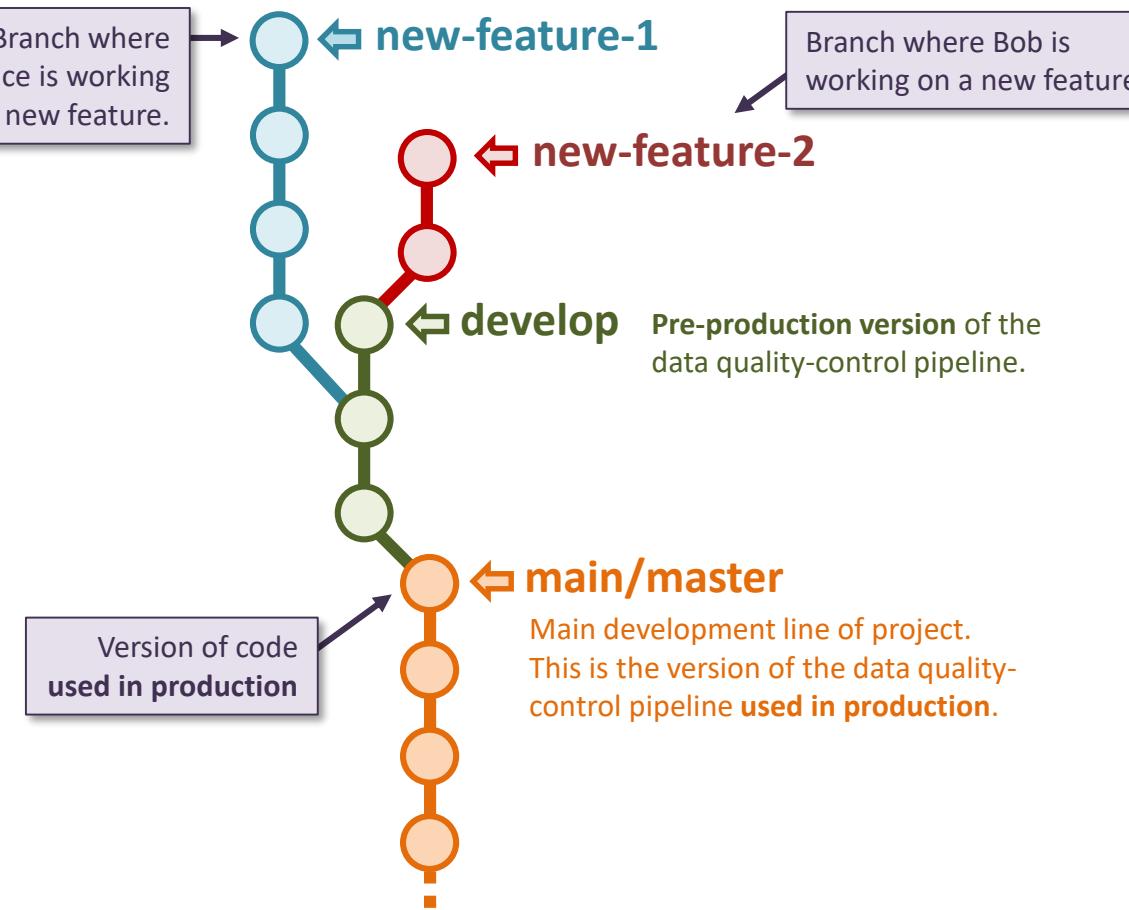


**Don't change the history on the main branch** if your project is used by others.

# Branch management strategies: GitFlow vs. Trunk-based development

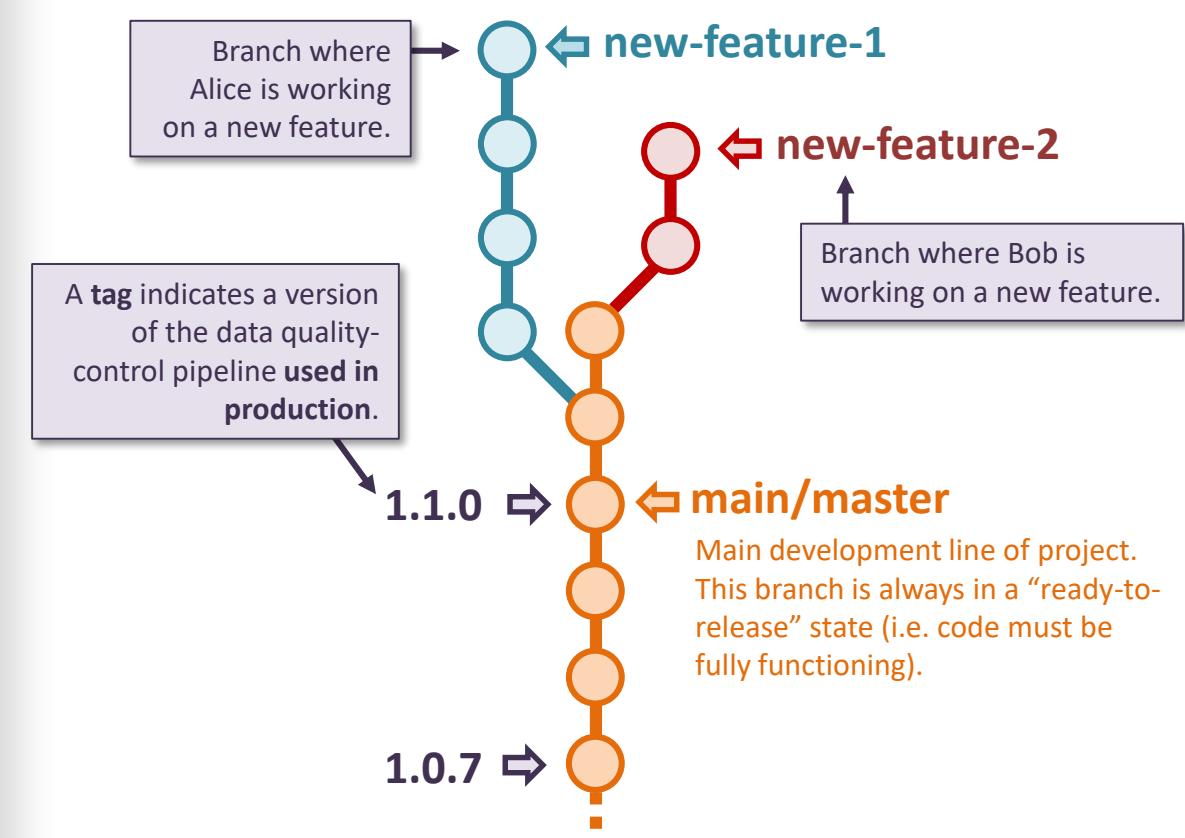
**GitFlow:** the idea is to have a long-lived **pre-production branch** (here “**develop**”), on which new features are added until ready for a new release, at which point the pre-production branch is merged into **main**.

- Useful if you distribute your code via the **main** branch of the Git repo, without making formal releases, i.e. your end-users use the latest version of **main** in production.

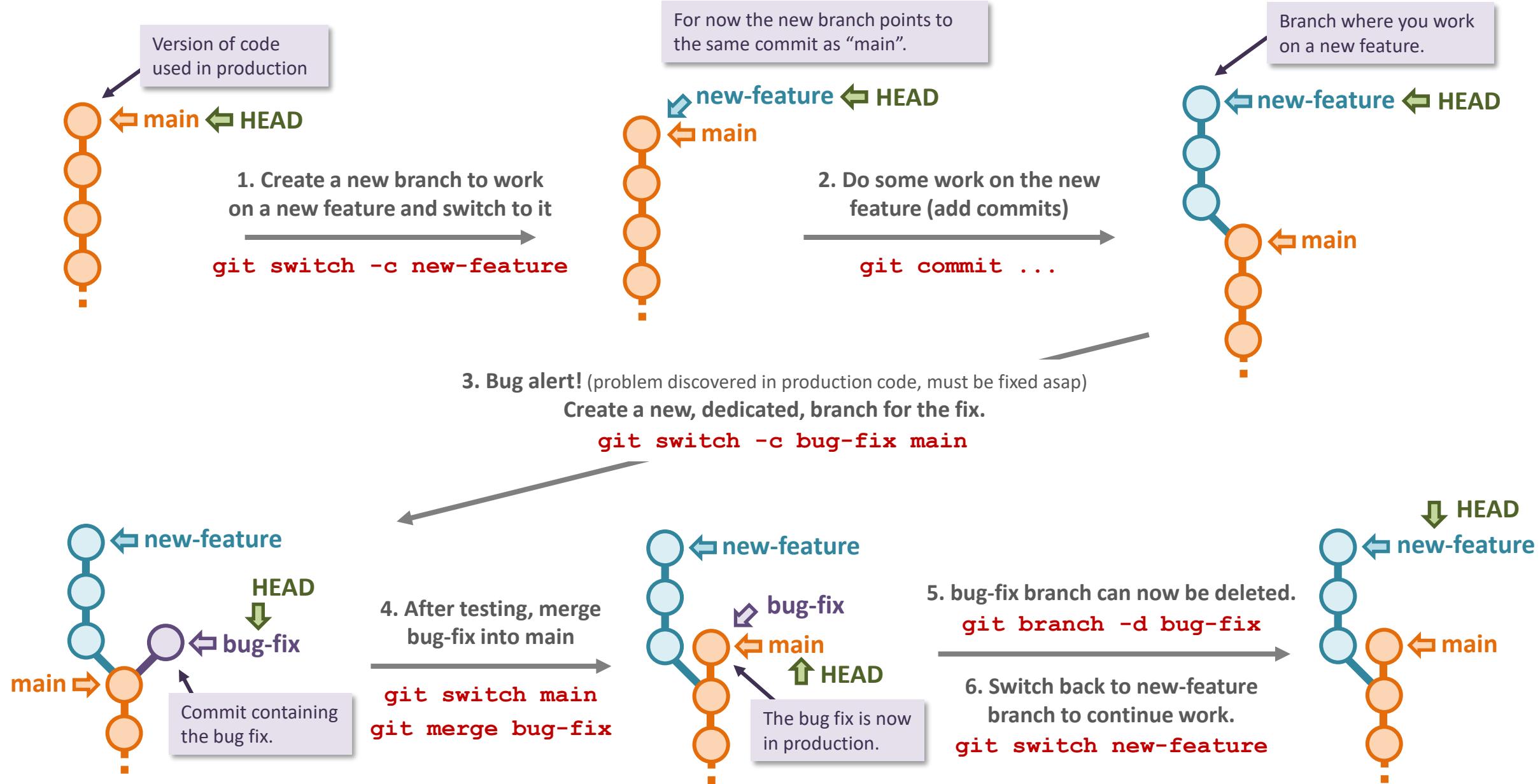


**Trunk-based development:** there is no long-lived branch outside of the **main** branch. All feature branches are directly merged into **main** once they are completed, and **main** should always be “production-ready”. **Tags** are generally added to denote commits corresponding to versions used in production.

- If you **distribute your code via formal releases**, then this strategy makes more sense as it avoids the overhead of managing an extra long-lived branch (the pre-release branch in GitFlow).



## Recap - example of branched workflow: adding a new feature to an application and fixing a bug



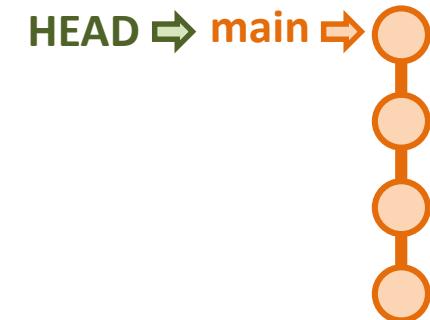
# exercise 2

The Git reference webpage

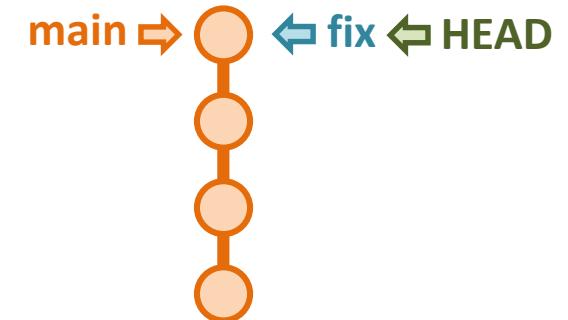


This exercise has helper slides

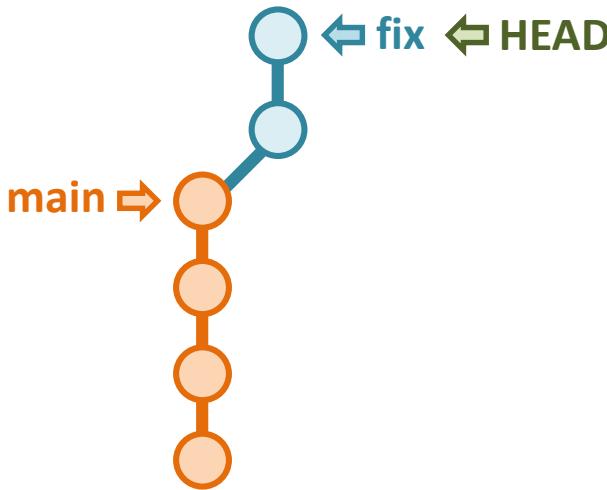
## Exercise 2 help: workflow summary



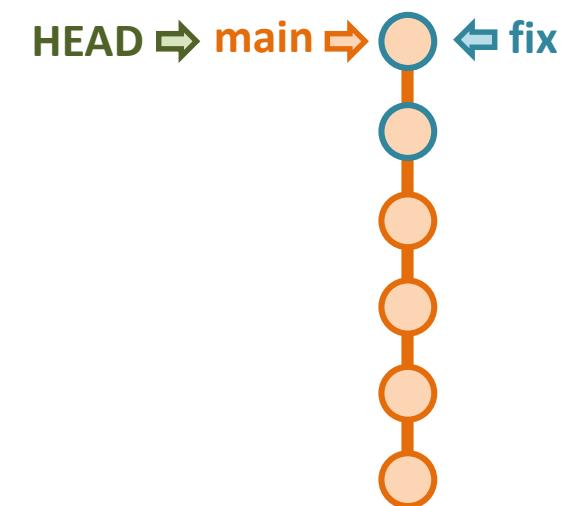
1. Create new branch *fix* and switch to it.



2. Do some work,  
add commits.



3. Test new feature, then merge  
branch *fix* into *main*.



# git rebase

## make a linear history

# git rebase: replay commits\* onto a different base

\* To be completely correct, we should say that we replay **differences between commits** (i.e. the changes that commits introduce to our code base), not the commits themselves (a commit is a state of the repo at a given time, it does not directly contain the information of changes to the codebase ).

- **git rebase**: move/re-root a branch onto a different base commit.
- **Important**: the rebase command must be executed when on the branch to rebase, not the branch you rebase on.

```
git rebase <branch to rebase on>
```

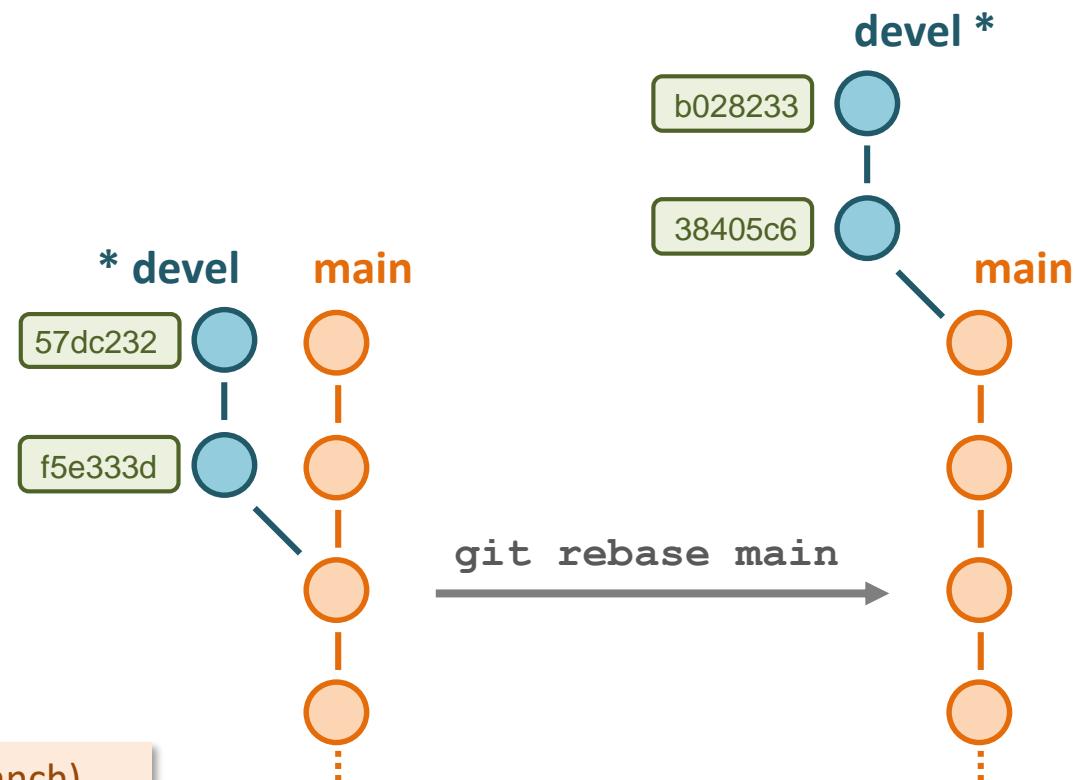
Example:

```
$ git branch
* devel      ← ! Make sure you are on the
main               branch you want to rebase !
$ git rebase main
```

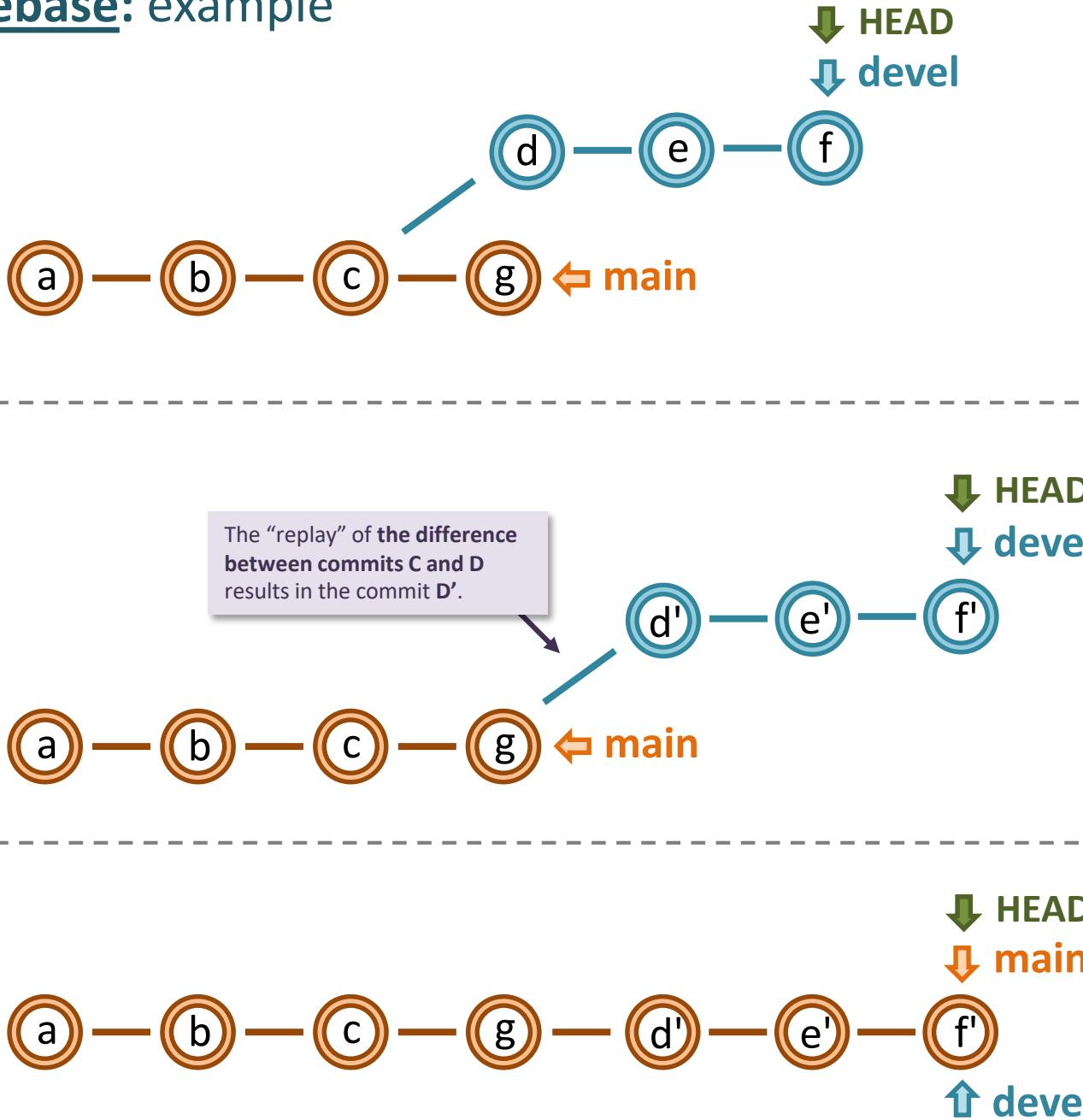
The branch you want to rebase on.



Rebase will modify your commit ID values (history of the rebased branch).  
It's best to only rebase commits that have never left your own computer.



## git rebase: example



Before starting the rebase: make sure you are on the branch to rebase!

In this case, if we were not already on branch **devel** we would run:

```
git switch devel
```

git rebase main

We can now fast-forward merge.  
Guaranteed to be conflict free :-)

git switch main  
git merge devel

## Resolving conflicts with rebase

- Rebase re-applies all commit to rebase sequentially: **at each step** there is a potential for conflict...
- To resolve conflicts, you will have to (same as for conflict resolution during merges):

**1. Edit the conflicting files**, choose the parts you want to keep, then remove all lines containing <<<<<, ===== and >>>>>.

**2. Mark the files as resolved with  
git add <file>**

**1. Continue the rebase with  
git rebase --continue**

**When a conflict arises, Git will provide guidance:**

```
$ git rebase main
```

First, rewinding head to replay your work on top of it...

Applying: first commit on new branch

Using index info to reconstruct a base tree...

M new.txt

Falling back to patching base and 3-way merge...

Auto-merging new.txt

**CONFLICT (content): Merge conflict in new.txt**

error: Failed to merge in the changes.

Patch failed at 0001 first commit on new branch

Use 'git am --show-current-patch' to see the failed patch

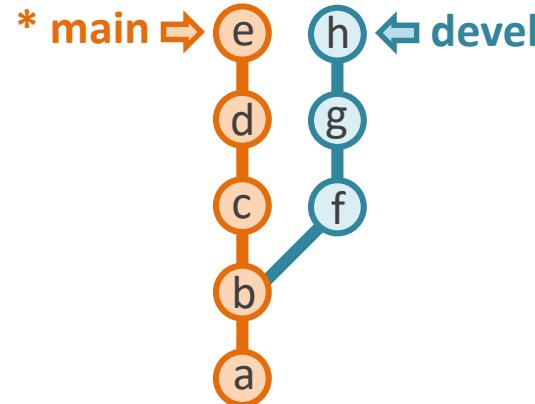
1. → **Resolve all conflicts manually,**
2. → **mark them as resolved with "git add/rm <conflicted\_files>"**
3. → **, then run "git rebase --continue".**

You can instead skip this commit: run "git rebase --skip".  
To abort and get back to the state before "git rebase",  
run "git rebase --abort".

# Branch reconciliation strategies when history has diverged: merge vs. rebase

## merge (3-way merge)

- + Preserves history perfectly.
- + Potential conflicts must be solved only once.
- Creates an additional merge commit.
- Often leads to a "messy" history.

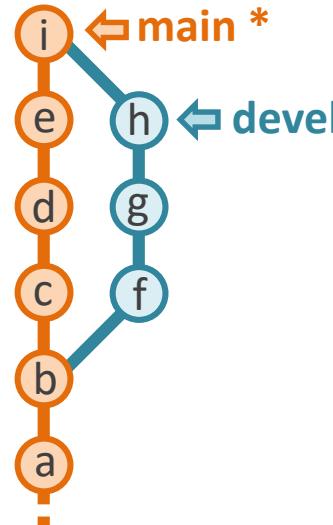


## rebase + fast-forward merge

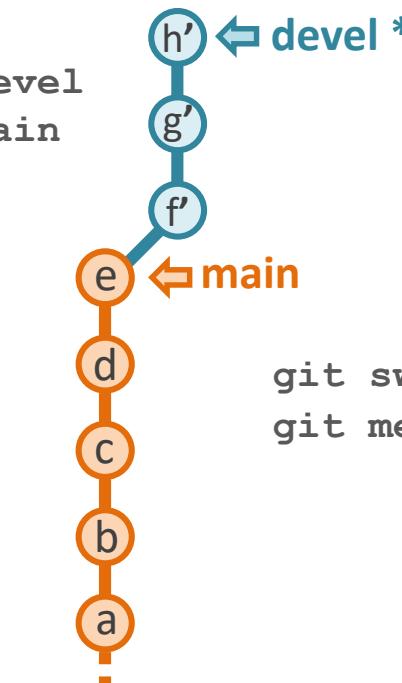
- + Cleaner history = easier to read and navigate.
  - Conflicts may have to be solved multiple times.
  - Loss of branching history.
- History of rebased branch is rewritten, not a problem in general.

Additional "merge commit".

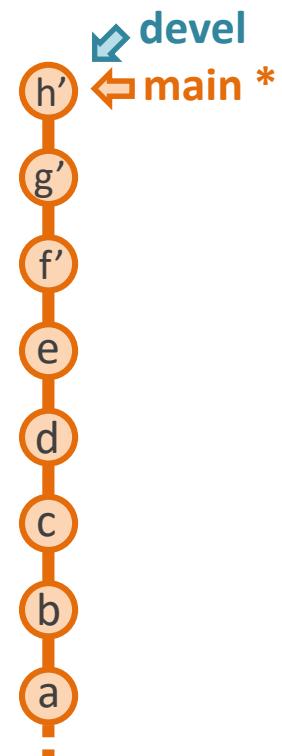
git merge devel



git switch devel  
git rebase main



git switch main  
git merge devel



Spoiler-alert: the end result is the same, **i** and **h'** have the same content.

## Readability vs. history preservation tradeoff

Screenshots of two versions of a **same repository** (in the sense that it contains the exact same content with mostly the same commits).

```
* a317d38 (HEAD -> main) Merge branch 'dev-alice'  
|  
| * 7999c7c (dev-alice) improvement: add success message to QC pipeline  
| * da96caa fix: update README  
| * ccea24b improvement: better tea-brewing checks  
| | 10fa3ad Merge branch 'dev-redqueen'  
| |  
| |  
| |  
| |  
| | * b4fb462 (dev-redqueen) update: add timing module  
| | * d37df05 improvement: check that Mad Hatter is on time  
| |  
| |  
* 7446b3e update: add tea-brew integration test  
* b82c9c9 update: add physical integrity check to pipeline  
* 96d19d4 Initial commit
```

← Here, history has been fully preserved, by always using merges and forcing extra merge commits (--no-ff) when needed.

Here, having a linear history has been prioritized (better readability), by rebasing branches before (fast-forward) merging them.

```
* 77d8354 (HEAD -> main, dev-alice) improvement: add success message to QC pipeline  
* e48c71a fix: update README  
* 51ae05e improvement: better tea-brewing checks  
* b4fb462 (dev-redqueen) update: add timing module  
* d37df05 improvement: check that Mad Hatter is on time  
* 7446b3e update: add tea-brew integration test  
* b82c9c9 update: add physical integrity check to pipeline  
* 96d19d4 Initial commit
```

## Supplementary material...



```
* 2501d8d417 (origin/test_node, test_node) Merge pull request #14830 from migueldiascosta/20220124105343_new_pr_EasyBuild452
| * a7f24f6f0c adding easyconfigs: EasyBuild-4.5.2.eb
| * aaa77532dc resume running test suite with Python 3.5 by using actions/setup-python@v2
| * 925fc73a7 add quotes to avoid that Python 3.10 is interpreted as Python 3.1 ...
| * d780bb7cae stop running easyconfigs test suite with Python 3.5, also test with Python 3.8-3.10
| * 2585d099b8 sync with main + bump version to 4.5.3dev
| * f1de981545 (tag: easybuild-easyconfigs-v4.5.2, origin/main, eb-source/main) Merge pull request #14829 from easybuilders/4.5.x
| * 4440893abe (eb-source/4.5.x) Merge pull request #14828 from migueldiascosta/eb452
| * de2a8651cc minor tweak release notes for v4.5.2
| * 0a205792b7 prepare release notes for EasyBuild v4.5.2 + bump version to 4.5.2
| * 5f7f1e103e Merge pull request #14821 from branfosj/20220121150125_new_pr_X1120210518
| * 9390faedad add libXfont2 patch to fix build when libbsd is present
| * f497a23162 (origin/scicore) Merge pull request #14743 from sib-swiss/20220117153155_new_pr_RDKit2021034
| * 33c378c1be Update RDKit-2021.03.4: update comic-neue-checksum patch checksum
| * 1c52b4bf3 Update RDKit-2021.03.4: add comic-neue-checksum patch description and author
| * cf0bacffd7 Add patch for hard-coded checksum value of downloaded source file in the source code
| * 73275792fb add missing binutils build dependency to namedlist easyconfig
| * 55ec7565aa adding easyconfigs: namedlist-1.8-GCCcore-11.2.0.eb
| * 0ea31891d9 Merge pull request #14806 from boegel/20220120190948_new_pr_R-bundle-Bioconductor314
| * bbfa623c8c add pathview extension to R-bundle-Bioconductor 3.14
| * 45acf59f55 Merge pull request #14711 from ItIsI-Orient/20220113183646_new_pr_Short-Pair20170125
| * 70bfefbaca Added required changes
| * be0006d48b Fixed error + edited patch desc
| * 6f85ffb535 adding easyconfigs: Short-Pair-20170125-foss-2021b.eb and patches: Short-Pair-20170125-Python3fix.patch
| * ab3099a9df Merge pull request #14792 from branfosj/20220119163605_new_pr_Pillow-SIMD832
| * c2f8c0ee7c the Pillow v8 patch also works for Pillow-SIMD v7
| * 9e333911ca fix CVE-2021-23437 in Pillow-SIMD v8 + add Pillow-SIMD v8.3.2 in easyconfigs using a 2021b toolchain
| * aa46b3ecf1 Merge pull request #14548 from shot0829/20211213195043_new_pr_elbencho203
```

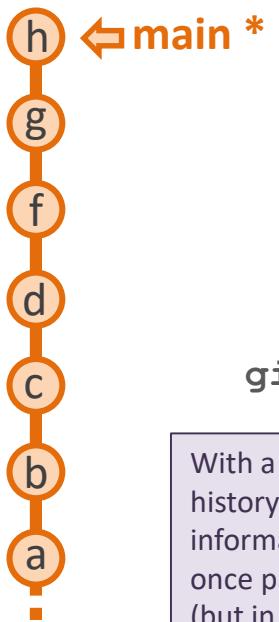
Never rebasing your changes before merging can lead to a hard to read history...

## Ultimate history preservation: force the addition of a merge commit with --no-ff

When keeping an **exact record** of how the history of a Git repo came into existence is of prime importance, some people like to **add a merge commit even if a fast-forward merge is possible**.

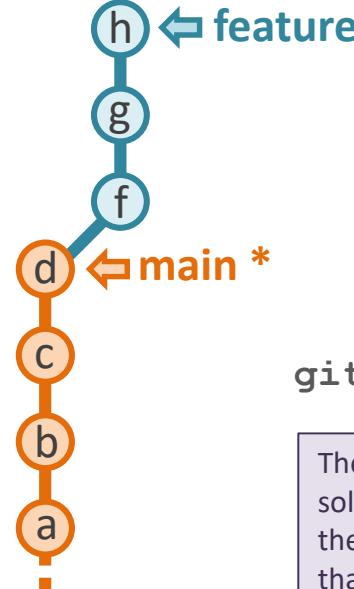
This is possible by adding the **--no-ff** option (“no fast-forward”) to git merge.

```
git merge --no-ff <branch to merge>
```



git merge feature

With a regular fast-forward merge, the history is cleaner. However, the information that “f”, “g” and “h” were once part of a different branch is lost (but in most cases this doesn’t matter).

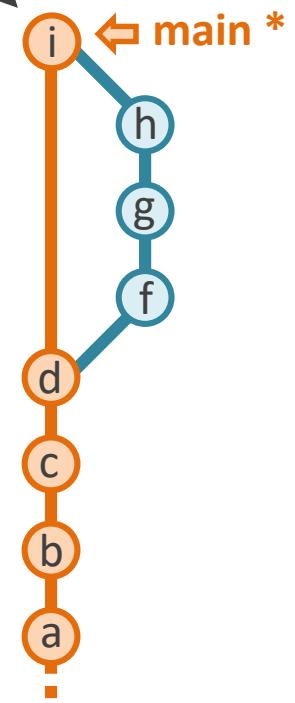


git merge --no-ff feature

The merge commit “i” is added for the sole purpose of allowing us to reconstruct the exact history of the repo: it tells us that commits “f”, “g” and “h” were once part of a different branch, which was then merged into “main”.

```
$ git show 10fa3ad
commit 10fa3ad505821b0ea628b8
Merge: 7446b3e b4fb462
Author: Alice <alice@redqueen.org>
Date: Tue Oct 11 15:16:39 2022 +0200
```

Merge branch ‘feature’



## Demo

- Rebasing a branch (feat. manual conflict resolution)

# Cherry-picking: copy-pasting commits

```
git cherry-pick
```

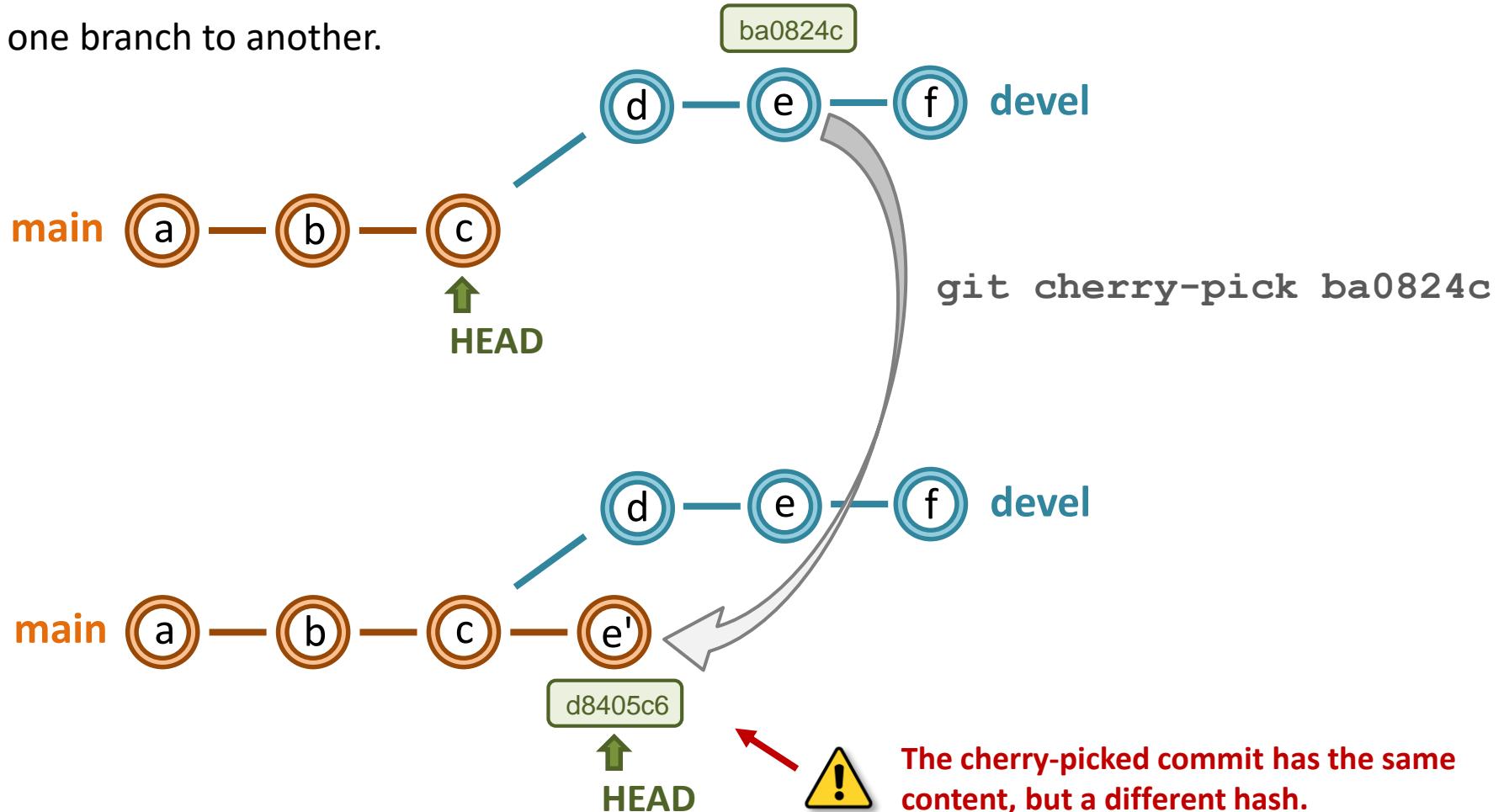
## Cherry-pick: merge a single commit into the current branch

- **git cherry-pick**: "copy" a commit (or several) to the current branch.

```
git cherry-pick <commit to pick>
```

### Example:

"copy" a fix from one branch to another.



## Cherry-pick use case: simplify history

```
* 0cc45cf (HEAD -> master) Update and fix bugs in exercises and slides
| * 4e08672 (origin/master, origin/HEAD) Add newline before list in Cleanup.
| * 5935b22 Merge branch 'master' of gitlab.sib.swiss:rengler/git_course
| \
| /
| *
* | 9164384 Update slides
* | 8672e4b Hopefully make the list in ex4 look good now. All we really need  
is this commit
| /
* 7fdb6b1 Update exercises
* 4ad684b Rename files
* b7a33b1 Add exercise for git stash
```

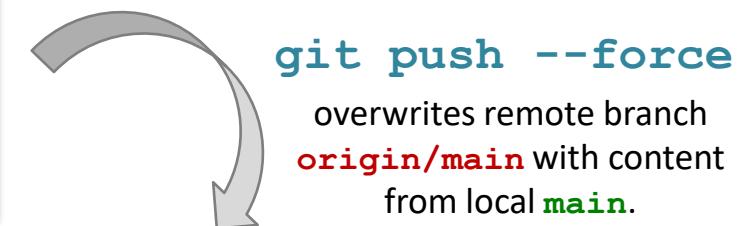
```
* 8473b64 (HEAD -> master) Hopefully make the list in ex4 look good now.
* 0cc45cf Update and fix bugs in exercises and slides
| * 4e08672 (origin/master, origin/HEAD) Add newline before list in Cleanup.
| * 5935b22 Merge branch 'master' of gitlab.sib.swiss:rengler/git_course
| \
| /
| *
* | 9164384 Update slides
* | 8672e4b Hopefully make the list in ex4 look good now.
| /
* 7fdb6b1 Update exercises
* 4ad684b Rename files
* b7a33b1 Add exercise for git stash
```

```
* 8473b64 (HEAD -> master, origin/master, origin/HEAD) Hopefully make the list in ex4 look good now.
* 0cc45cf Update and fix bugs in exercises and slides
* 9164384 Update slides
* 7fdb6b1 Update exercises
* 4ad684b Rename files
* b7a33b1 Add exercise for git stash
```

Supplementary material...

### Situation:

- The local and remote **main** branches have a complicated history.
- But in the end, all that is really needed from the **origin/main** branch is **8672eb4**.
- Should I merge my 2 branches ?



# Retrieve data from earlier commits

git restore

git checkout

## Un-stage file modifications (restore file in index)

```
git restore --staged <file name>
```

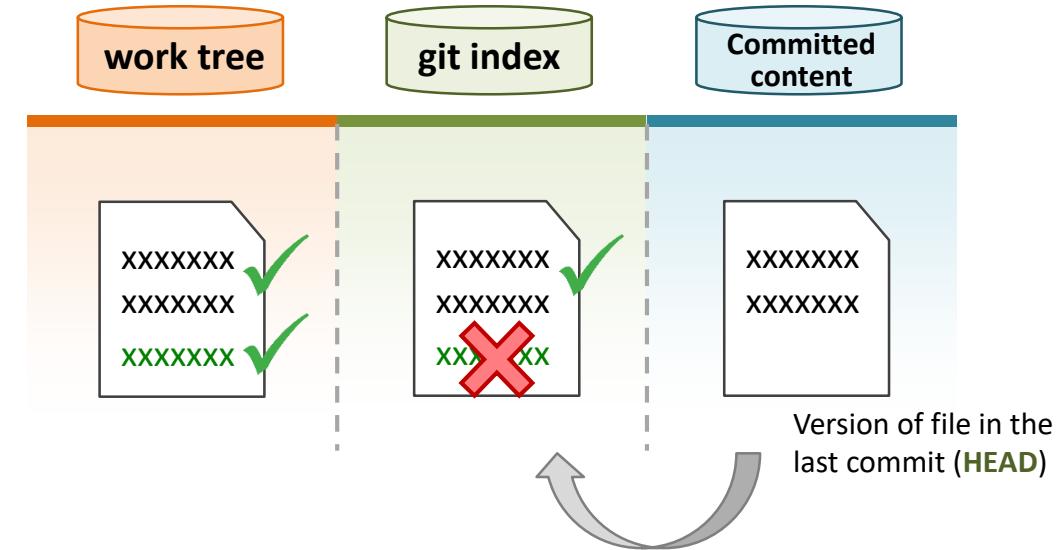
- Restores the content of a file in the Git index back to the latest commit (**HEAD** commit).
- Does not modify files in the working tree.

**Example:** un-stage changes to README.md file.

```
$ git status
On branch main
Changes to be committed:
(use "git restore --staged <file>..." to unstage)
    modified:   README.md
```

```
$ git restore --staged README.md
```

```
$ git status
On branch main
Changes not staged for commit:
(use "git add <file>..." to update what will be committed)
(use "git restore <file>..." to discard changes in working directory)
    modified:   README.md
```



```
git restore --staged README.md
```

The file is still modified in the working directory, but the changes are no longer staged.

# Restore / checkout of individual files

Retrieving the content of a file from an earlier commit can be done with either:

```
git restore -s/--source <commit reference> <file name>
```

or

```
git checkout <commit reference> <file name>
```

**Warning:** these commands will overwrite existing versions of the retrieved file in your working tree (without any sort of warning). Make sure you don't have uncommitted changes you want to keep.



➤ If no commit references is specified, the file is retrieved from the index.

Examples: the <commit reference> can be e.g. a commit ID, a relative reference, a tag or a branch name.

```
$ git restore -s ba08242 output.txt
$ git restore -s HEAD~10 output.txt
$ git restore -s v2.0.5 output.txt
$ git restore -s devel-branch output.txt
```

using a branch name, implicitly refers  
to the latest commit on the branch.

```
$ git checkout ba08242 output.txt
$ git checkout HEAD~10 output.txt
$ git checkout v2.0.5 output.txt
Updated 1 path from 2a7fac8
$ git checkout devel-branch output.txt
Updated 1 path from e55fa6f
```

A small difference between these two commands is that **restore** updates the file only in the working tree (i.e. the files in your working directory), while **checkout** updates both the working tree and the index.

```
$ git restore --source ad26560 README.md
$ git status
Changes not staged for commit:
(use "git restore <file>..." to discard changes
in working directory)
modified: README.md
```

```
$ git checkout ad26560 README.md
Updated 1 path from e55fa6f
$ git status
Changes to be committed:
(use "git restore --staged <file>..." to unstage)
modified: README.md
```



## Checkout of the entire repo state at an earlier commit

- Checking out a commit will restore both the working tree and the index to the exact state of the specified commit.
- It will also move the **HEAD** pointer to that commit.

```
git checkout <commit reference>
```

Examples:

```
$ git checkout ba08242  
$ git checkout HEAD~10  
$ git checkout v2.0.5
```

Make sure to have a clean working tree before doing a checkout!

```
$ git checkout ad26560  
error: Your local changes to the following files would be  
overwritten by checkout:  
        README.md  
Please commit your changes or stash them before you switch branches
```

- After a checkout, you enter a "detached HEAD" state....
- To get back to a “normal” state you should go back to a regular branch:

```
git switch <branch> or git checkout <branch>
```



```
$ git checkout ba08242  
Note: checking out 'ba08242'.
```

You are in 'detached HEAD' state. You can look around, make experimental changes and commit them, and you can discard any commits you make in this state without impacting any branches by performing another checkout.

# exercise 3

The crazy peak sorter script



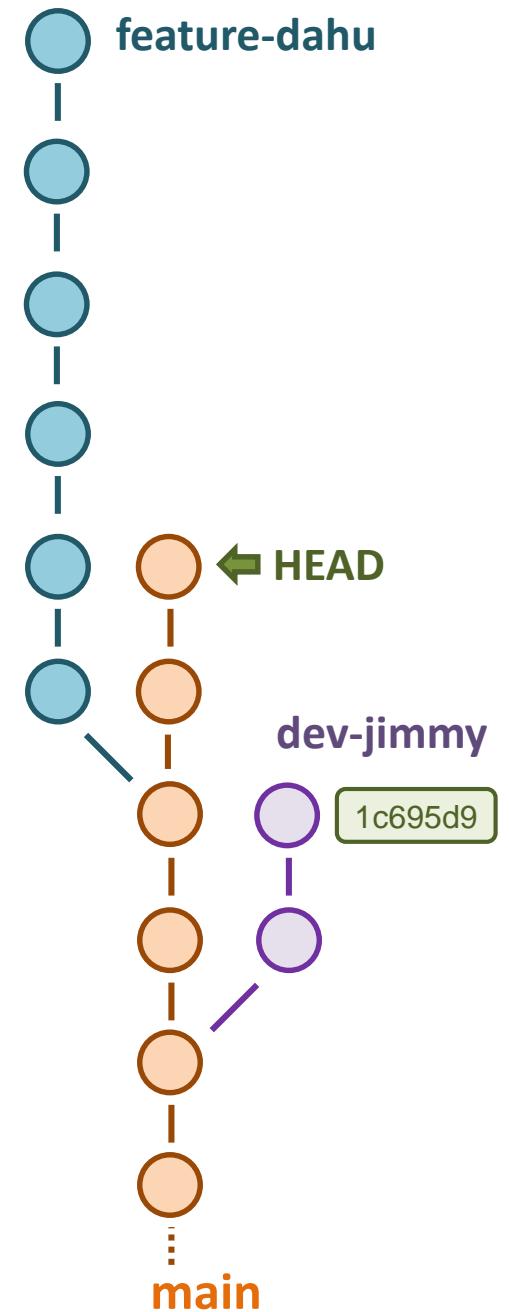
This exercise has helper slides

## Exercise 3 help: history of the peak-sorter repo

This slide shows the history of the repo for exercise 3, both as the command line output and as a schematic representation (on the right).

This can help you understand the command line representation of a repo's history.

```
[rengler@local peak_sorter]$ git log --all --decorate --oneline --graph
* fc0b016 (origin/feature-dahu) peak_sorter: display highest peak at end of script
* d29958d peak_sorter: added authors as comment to script
* 6c0d087 peak_sorter: improved code commenting
* 89d201f peak_sorter: add Dahu observation counts to output table
* 9da30be README: add more explanation about the added Dahu counts
* 58e6152 Add Dahu count table
| * f6ceaac (HEAD -> master, origin/master, origin/HEAD) peak_sorter: added authors to script
| * f3d8e22 peak_sorter: display name of highest peak when script completes
|
* cfd30ce Add gitignore file to ignore script output
* f8231ce Add README file to project
| * 1c695d9 (origin/dev-jimmy) peak_sorter: add check that input table has the ALTITUDE and PEAK columns
| * ff85686 Ran script and added output
|
* 821bcf5 peak_sorter: add +x permission
* 40d5ad5 Add input table of peaks above 4000m in the Alps
* a3e9ea6 peak_sorter: add first version of peak sorter script
```



## Part III

# Working with remotes

Linking your local repo with an  
online server

# What is a “remote” ?

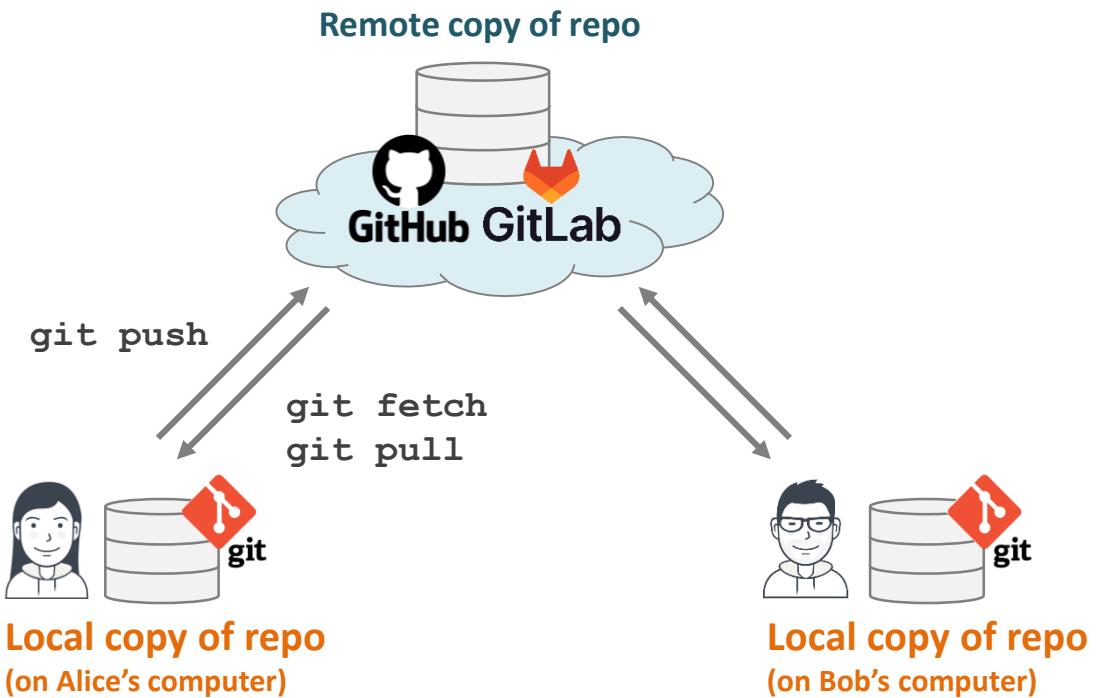
A **remote** is a copy of a Git repository that is stored on a server (i.e. online).

Remotes are very useful, as they allow you to:

- **Backup** your work.
- **Collaborate** and synchronize your repo with other team members.
- **Distribute** your work – i.e. let other people clone your repo (e.g. like the repo of this course).

## Good to know:

- Each copy of a Git repo (local or online) is a **full copy of the entire repo’s history** (provided it has been synced).
- Git does not perform any automatic sync between the local and remote repos. All **sync operations must be manually triggered**.



Remotes are generally hosted on dedicated servers/services, such as GitHub, GitLab (either gitlab.com or a self-hosted instance), BitBucket, ...

## Add a remote to an existing project (or update a remote's URL)

- **Case 1:** your local **repo** was cloned from a remote – *nothing to do* (the remote was automatically added by Git).
- **Case 2:** your local **repo** was created independently from the remote – it must be linked to it.

Add a new remote: `git remote add <remote name> <remote url>`

Change URL of remote: `git remote set-url <remote name> <remote url>`

Note: by convention, the `<remote name>` is generally set to `origin`.

### Examples

```
# Add a new remote (named origin) to the local repo:  
$ git remote add origin https://github.com/sibgit/test.git
```

```
# Update the URL of the existing origin remote.  
# In this example, the remote was moved GitLab.  
$ git remote set-url origin https://gitlab.sib.swiss/sibgit/test.git
```

## Example – part 1: creating a new remote and pushing new branches



Alice's computer

GitHub GitLab Remote



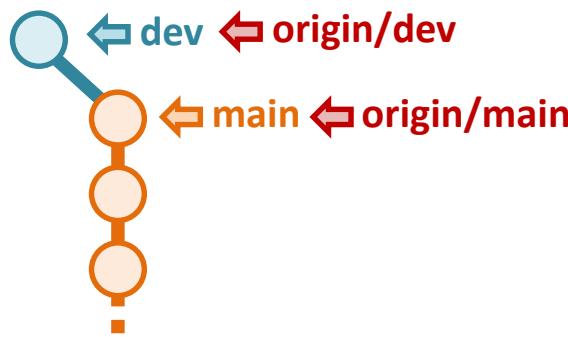
Alice has a Git repo with 2 branches: **main** and **dev**. She now wants to store her work on GitHub, to collaborate and have a backup.

1. She creates a remote on GitHub and links it to her local repo using `git remote add origin <URL of remote>`
2. She pushes her branch **main** to the remote using `git push -u origin <branch name>`  
(at this point the branch has no upstream, so the `-u/--set-upstream` option must be used).
3. She pushes her branch **dev** to the remote.

## Example – part 2: cloning a remote and checking-out branches

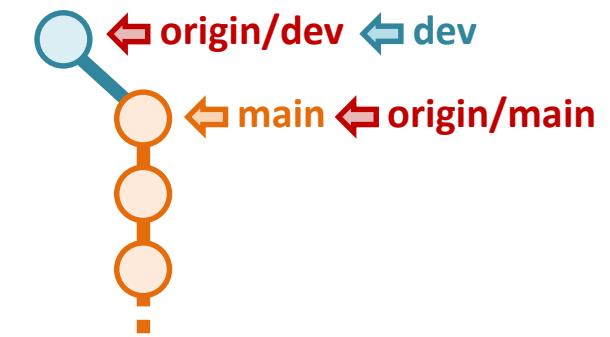
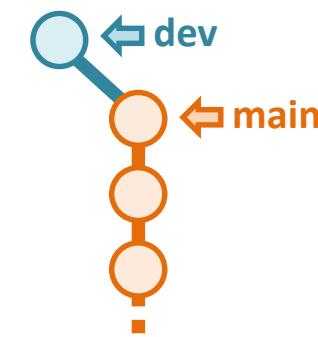


Alice's computer



Bob's computer

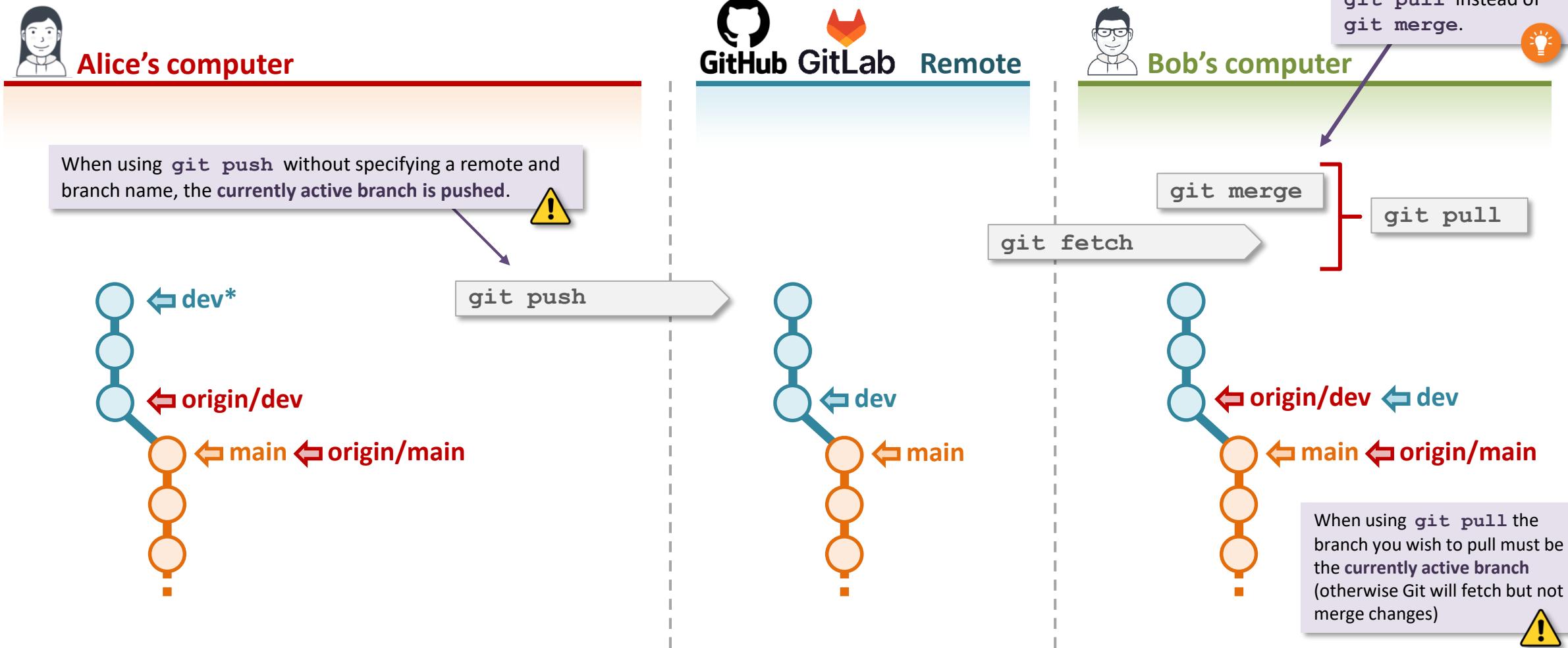
```
git clone https://github.com/...  
git switch dev
```



Bob has now joined the team to work with Alice.

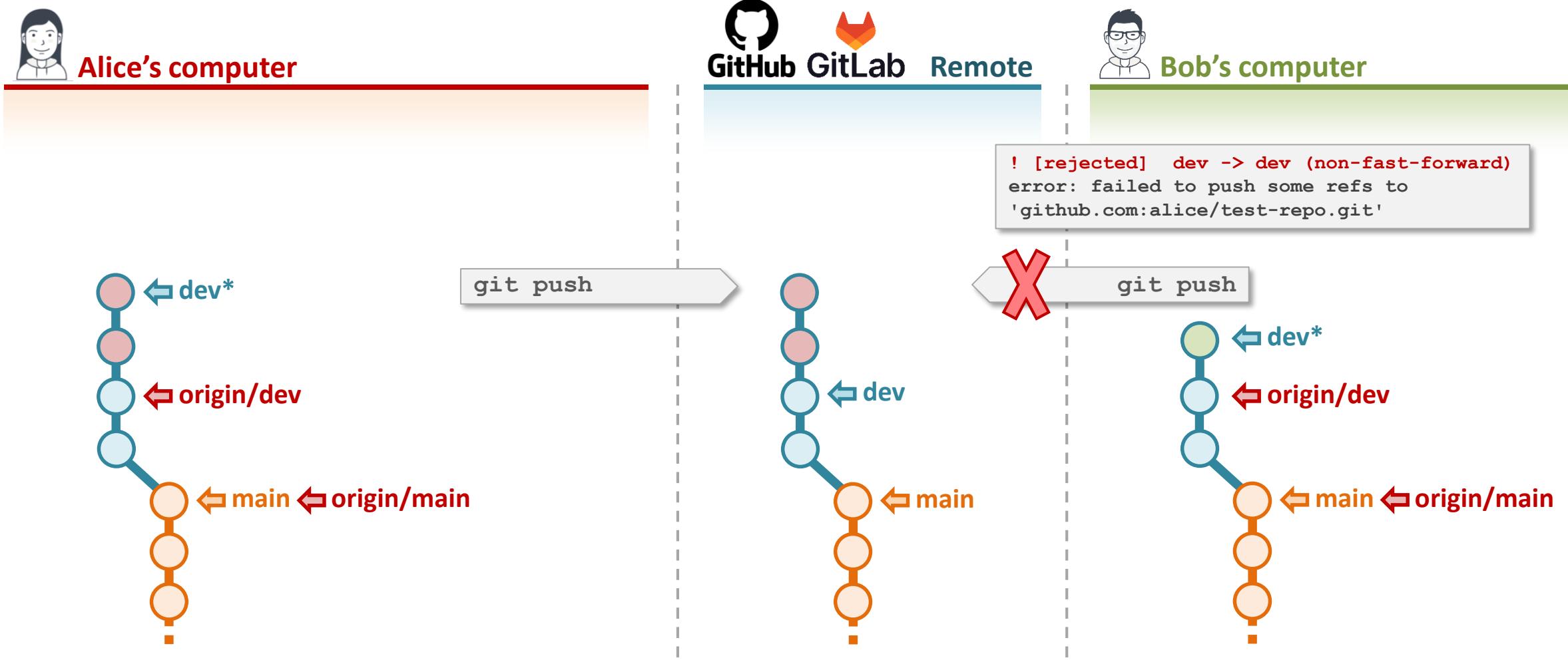
1. He **clones** the repo from GitHub using `git clone <URL of remote>`. At this point, Bob has no local **dev** branch - only a pointer to **origin/dev**.
2. Bob checks-out the **dev** branch to work on it. Because there is already a remote branch **origin/dev** present, Git automatically creates a new local branch **dev** with **origin/dev** as upstream (no need add the `--create/-c` option to `git switch`).

## Example – part 3: pushing and pulling changes



1. In the mean time, Alice added 2 new commits to `dev`. She pushes her changes to the remote using `git push` (since her `dev` branch already has an upstream, there is no need to add the `-u/--set-upstream` option this time).
  2. To get Alice's updates from the remote, Bob runs `git pull`, which is a combination of `git fetch + git merge`.  
**Important:** `git fetch` downloads all new changes/updates from the remote, but does not update your local branches.

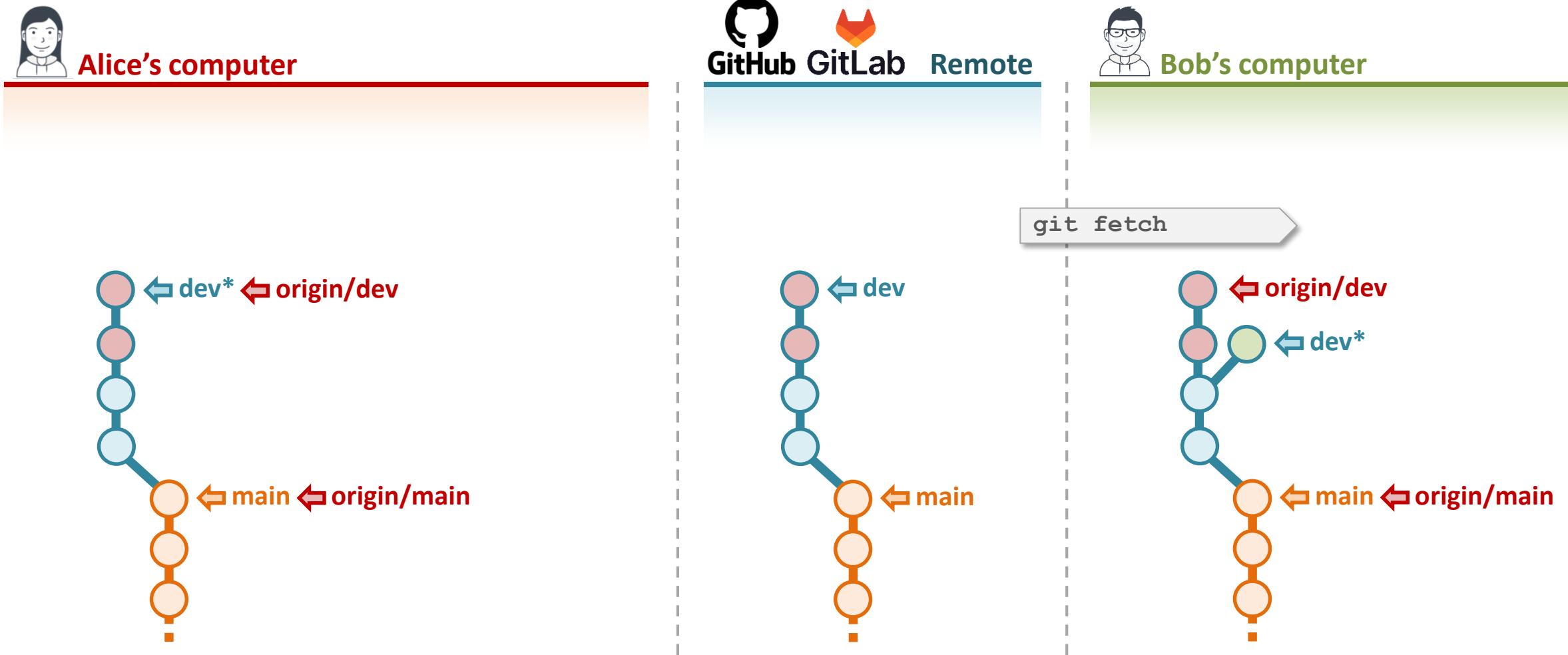
## Example – part 4: reconciliation of a diverging history



Both Alice and Bob have now both added some commits to their local `dev` branch. As a result, the history of their branches has diverged.

1. Alice pushes her changes to the remote with `git push`, as usual.
2. When Bob tries to `git push`, his changes are rejected because the history between his local `dev` branch and the remote have **diverged!**

## Example – part 4: reconciliation of a diverging history (continued)

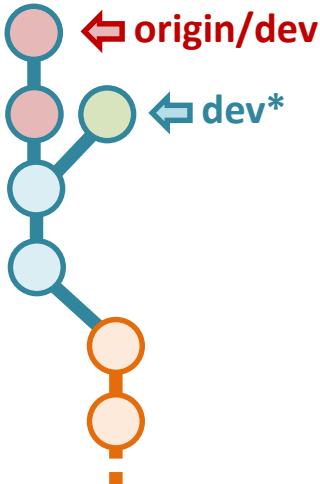


In order to be able to push his changes to the remote, Bob must first reconcile his local `dev` branch with the remote...

1. Bob starts by performing a `git fetch`, just to get the new commits from the remote and see how his local branch diverges from the remote (**important:** this operation does not impact/update his local `dev` branch).

## Example – part 4: reconciliation of a diverging history (continued)

To reconcile his local `dev` branch with the remote, Bob must decide to either perform a merge or a rebase.



In this situation, a regular pull raises an error \*

```
$ git pull
fatal: Need to specify how to
reconcile divergent branches
```

### Option 1 - reconciliation using `merge`.

This is equivalent to:

```
git fetch
git merge origin/dev
```

*git pull --no-rebase*

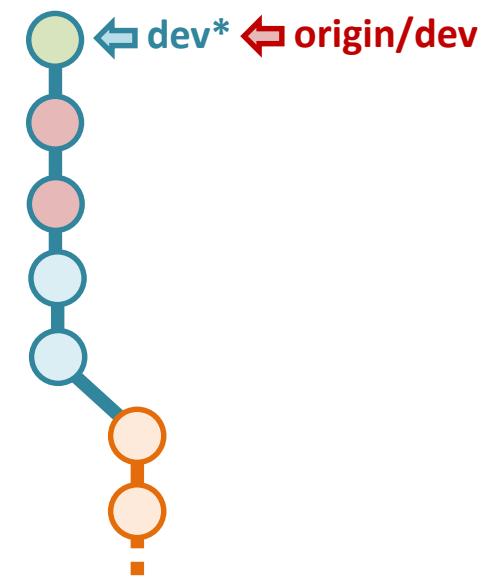


### Option 2 - reconciliation using `rebase`.

This is equivalent to:

```
git fetch
git rebase origin/dev
```

If you don't remember the `--no-rebase` and `--rebase` options of `git pull`, simply `fetch` and then `merge` or `rebase` on `origin/dev`.

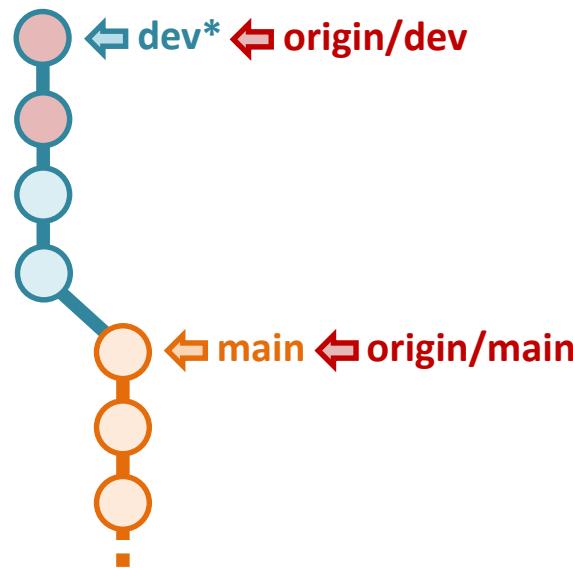


\* On recent Git versions (>= 2.33), the default pull behavior is to abort if history diverged. On older versions, the default behavior is to merge (as in `git pull --no-rebase`).

## Example – part 4: reconciliation of a diverging history (continued)

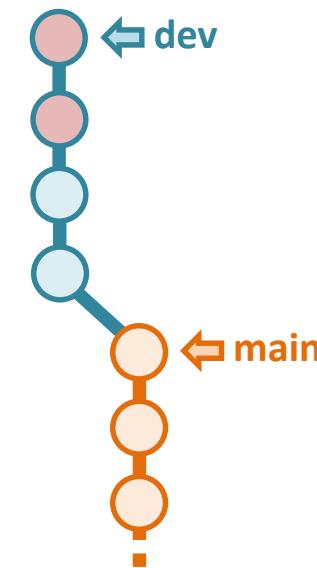


Alice's computer



Bob's computer

`git pull --no-rebase`



`dev* ← origin/dev`

Bob decides to merge without rebase and runs `git pull --no-rebase`.

Note: depending on the version of Git, the default behavior of `git pull` is different:

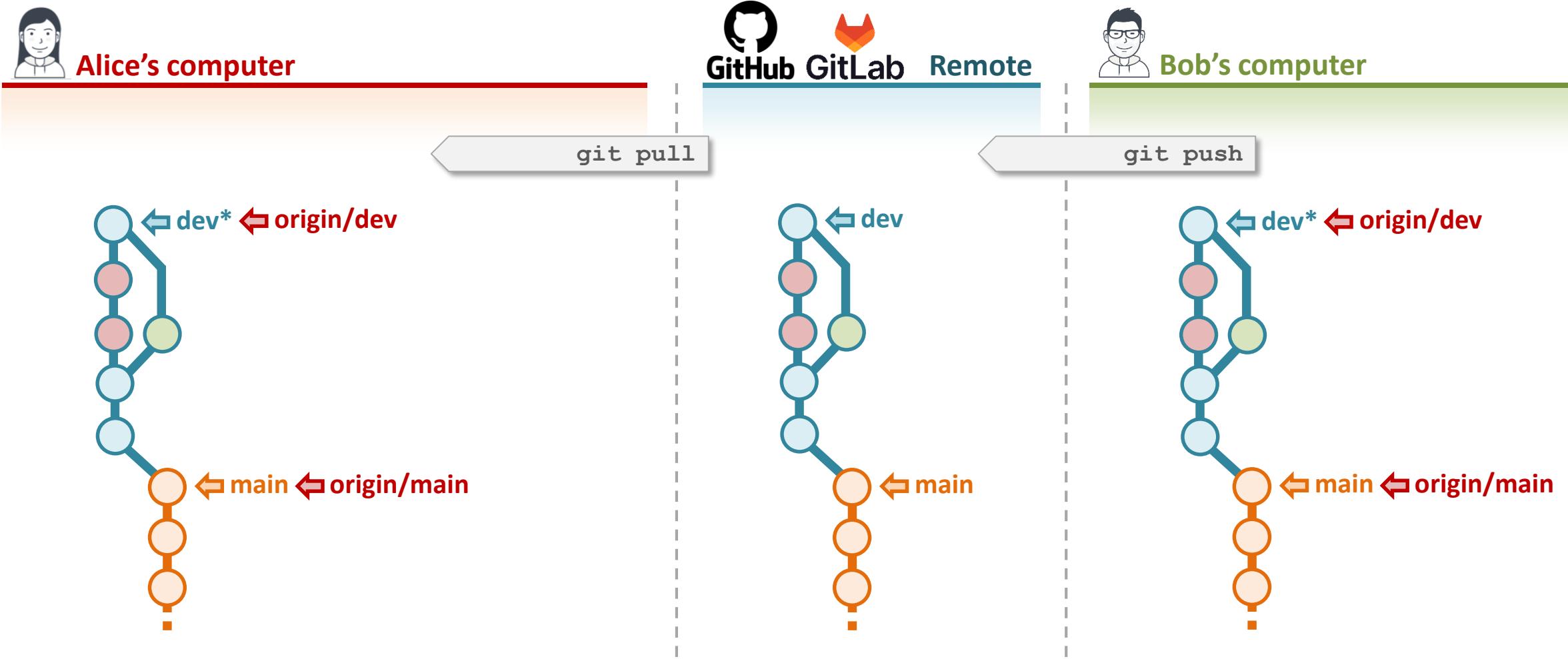
- Newer versions default to `git pull --ff-only` (i.e. raise an error if a fast-forward merge is not possible)
- Older versions default to `git pull --no-rebase` (i.e. the automatically merge)

The default behavior can be modified in the git config.

```

git config pull.rebase false      # merge
git config pull.rebase true       # rebase
git config pull.ff only          # fast-forward only
  
```

## Example – part 4: reconciliation of a diverging history (the end!)

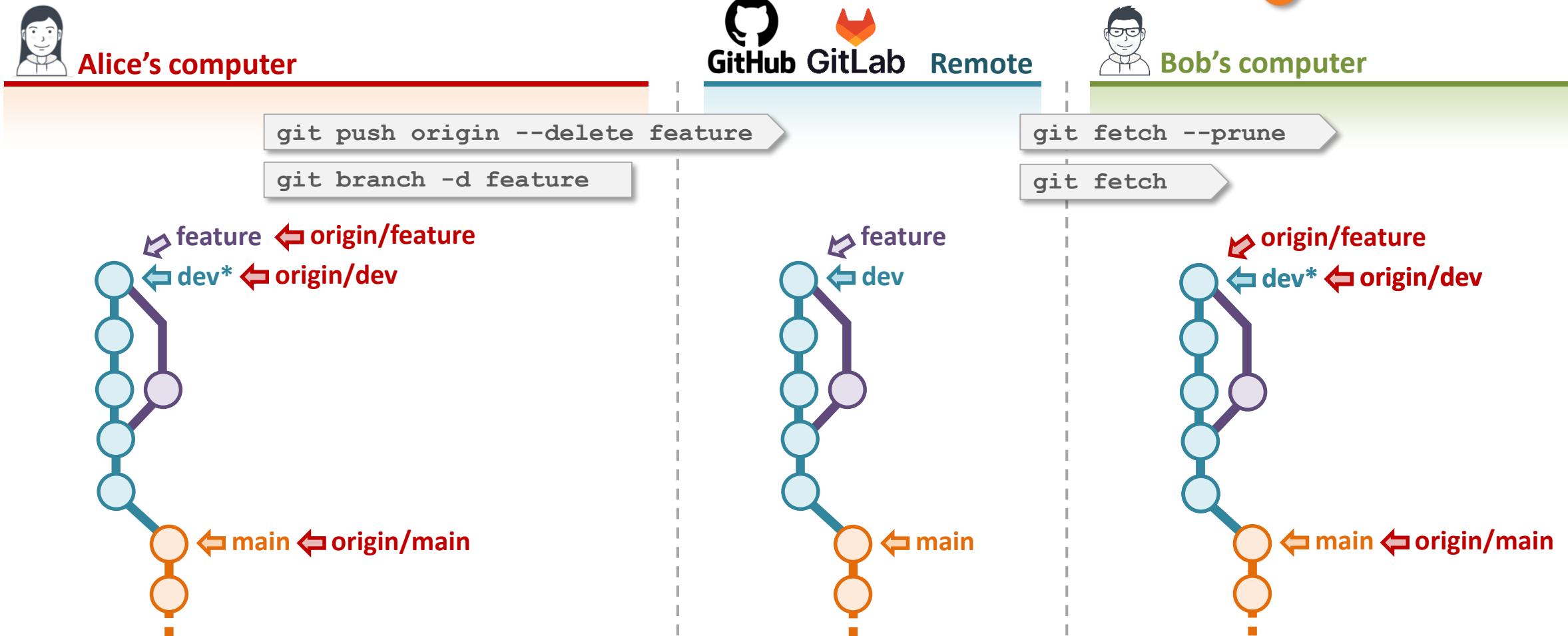


Finally, Bob can `git push` his changes to the remote - there are no more conflicts.

Alice can then `git pull` them.

## Example – part 5: deleting branches on the remote

The `--prune` option also works with `git pull --prune`.

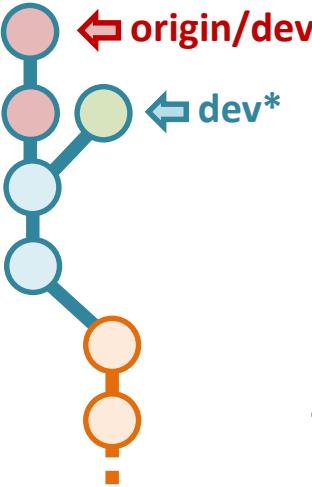


We are now at a later point in the development... Alice has just completed a new feature on her branch `feature`, and merged it into `dev`. She now wants to delete the `feature` branch both locally and on the remote.

1. Alice deletes her local branch with `git branch -d <branch name>`.
2. Alice deletes the feature branch on the remote with `git push origin --delete <branch name>`. This also deletes her `origin/feature` pointer.
3. Bob runs `git fetch`, but this does not delete references to remote branches, even if they no longer exist on the remote.
4. To delete his local reference to the remote feature branch (`origin/feature`), Bob has to use `git fetch --prune`.

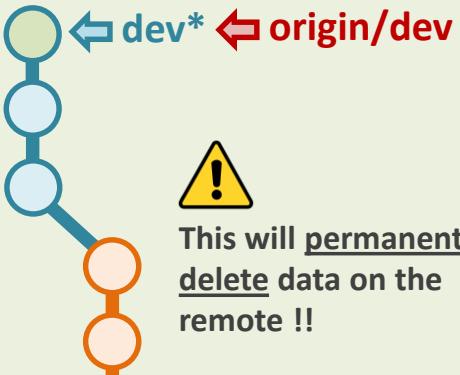
## Example – part 6: overwrite history on the remote

Example, if you made some history-rewriting change locally, typically a rebase of a branch.

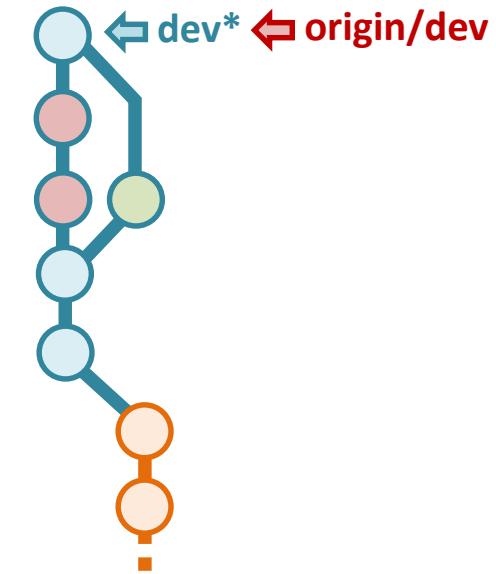


`git push  
--force`

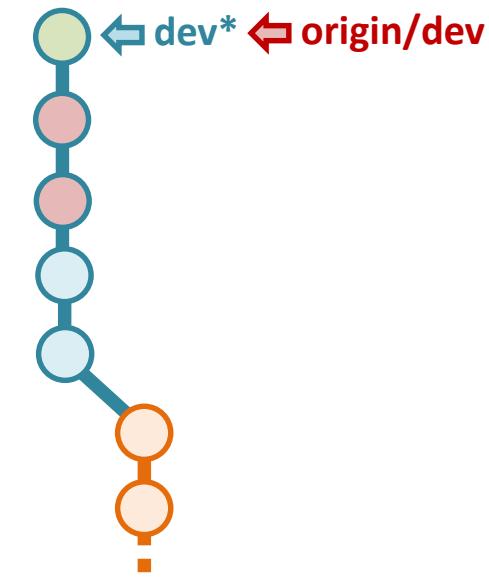
Option 3 – overwrite the remote  
with `git push --force`



`git pull --no-rebase`



`git pull --rebase`



# Interacting with remotes: commands summary

## Command      What it does

`git clone <URL>`

Create a local copy from an existing online repo. Git automatically adds the online repo as a remote.

By convention, the `<remote name>` is generally set to `origin`, but it could be anything.

`git remote add origin <remote url>`

Add a new remote to an existing local repo.

`git remote set-url origin <remote url>`

Change/update the URL of a remote associated to a local repo.

`git remote -v`

Display the remote(s) associated to a repo.

```
$ git remote -v
origin  https://github.com/alice/test-project.git (fetch)
origin  https://github.com/alice/test-project.git (push)
```

The fetch and push URLs should be the same.  
To use different URLs (different remotes) for push and fetch, add two different remotes.

`git branch -vvva`

List branches of repo and their associated upstream (if any).

```
$ git branch -vvva
manta-dev 18d8de0 [origin/manta-dev] manta ray: add animal name
main       6c8d731 [origin/main] Merge pull request #44 from sibgit/dahu-dev
* sunfish   18d8de0 manta ray: add animal name
```

We can see that the branches `main` and `manta-dev` have an upstream branch. The `sunfish` branch does not.

# Interacting with remotes: commands summary

## Command

## What it does

## Where to run and comments

### git push

push new commits on the current branch to the remote.

**Run on the branch that you wish to push.**

(only changes on the active branch are pushed)

`git push -u origin <branch-name>`

Same as git push, but additionally sets the upstream branch to **origin/branch-name**. Only needed if no upstream is set.

`-u` option is only needed when pushing a branch to the remote for the very first time. It is not needed if you initially created the local branch from a remote branch.

`git push origin <branch-name>`

Push new commits on the specified branch to the remote.

When the remote (here **origin**) and branch names are specified, the push command **can be run from anywhere**.

`git push --force`

Overwrite the branch on the remote with the local version.

**Warning:** this deletes data on the remote!

### git fetch

Download all updates from the remote to your local repo (even for non-active branches or branches for which there is no local version).

**Can be run from any branch.**

Does not update your local branch pointer to **origin/branch-name**.

### git pull

Download all updates and **merge changes** the upstream **origin/branch-name** into the active branch (i.e. update the active branch to its version on the remote).

**Run on the branch that you wish to update.**

`git pull` is a shortcut for  
`git fetch + git merge origin/branch-name`

`git pull --no-rebase`

Fetch + 3-way merge active branch with its upstream **origin/branch-name**.

On recent versions of Git (>= 2.33), the default pull behavior is to abort the pull if a branch and its upstream are diverging.

`git pull --rebase`

Fetch + rebase active branch on its upstream **origin/branch-name**.

On older versions, the default behavior is to merge them (same as `git pull --no-rebase`).

`git pull --ff-only`

Fetch + fast-forward merge active branch with its upstream **origin/branch-name**. If a fast-forward merge is not possible, an error is generated.



# git pull: a shortcut for fetch + merge

The `git pull` command is a shortcut for:

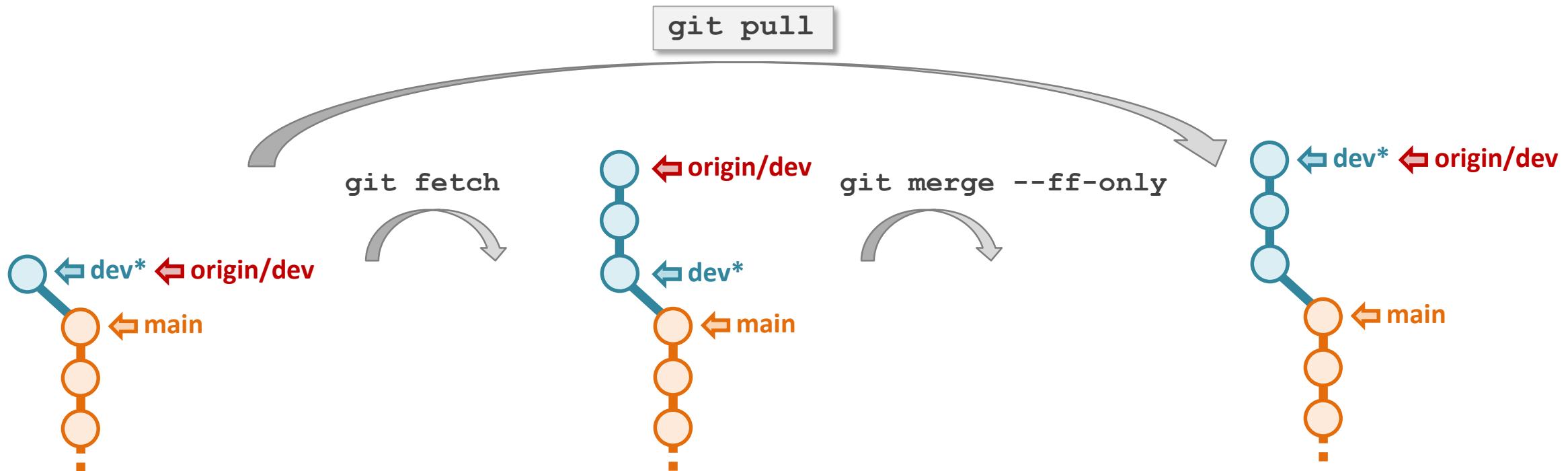
1. `git fetch` : fetches all updates from the remote.
2. `git merge -ff-only` : merge the currently active branch with its upstream branch (`origin/<branch>`).



Having the `git pull` command use `--ff-only` as default merge option is a recent behavior (Git >= 2.33). In older versions, to force `git pull` to only allow fast-forward merges, the following option must be set:

```
git config --global pull.ff only
```

↑ Fast-forward only -> any divergence in history will cause the command to fail and report an error.



By default, git merges a branch with its upstream branch, so `git merge` is the same as `git merge origin/<branch>`.

# GitHub / GitLab

collaborate and share your work



## GitHub / GitLab – an online home for Git repositories

- GitHub [[github.com](https://github.com)] and GitLab [[gitlab.com](https://gitlab.com)] are hosting platforms for Git repositories.
- Very popular to share/distribute software.
- Allows to host public (anybody can access) and private (restricted access) repos.
- Hosting of projects is free, with some paid features.
- Popular alternatives include:
  - A local instance of GitLab, the same as GitLab.com but hosted by someone else.
  - BitBucket [[bitbucket.org](https://bitbucket.org)].



# Project home page on GitHub

Example of the “home page” of a repository on GitHub

The screenshot shows the GitHub repository page for 'sibgit/test'. The top navigation bar includes 'Pull requests', 'Issues', 'Marketplace', 'Explore', and a search bar. Below the header, the repository name 'sibgit / test' is shown, along with a 'Code' tab (highlighted with an orange arrow), 'Issues (1)', 'Pull requests (1)', 'Actions', 'Projects (1)', 'Wiki', 'Security', 'Insights', and 'Settings'.

Annotations on the left side:

- Code tab: the “home” page of your repo.** (Orange arrow pointing to the 'Code' tab)
- Branch you are currently viewing** (Purple arrow pointing to the 'master' dropdown)
- List of files present in the repo.** (Purple arrow pointing to the file list)
- If you have a README.md file, it is displayed here (with markdown rendering).** (Purple arrow pointing to the 'README.md' content area)

Annotations on the right side:

- About** (Green button) (Purple arrow pointing to the 'About' section)
- No description, website, or topics provided.** (Dashed arrow from the 'About' section to the empty description area)
- Clone** (Purple box around the cloning options)
  - HTTPS** **SSH** **GitHub CLI** (Text)
  - To copy the repo’s URL.** (Text with a purple arrow pointing to the URL field)
  - https://github.com/sibgit/test.git** (URL field with a copy icon)
  - Use Git or checkout with SVN using the web URL.** (Text)
- Download ZIP** (Purple box around the download link)

## Repository settings (only available to project owner)

Here you can set diverse settings concerning your repository, e.g. :

- Invite **collaborators**.
- Setup **branch protection**.

### View with no collaborator added yet

Who has access

PRIVATE REPOSITORY  
Only those with access to this repository can view it.  
[Manage](#)

DIRECT ACCESS  
0 collaborators have access to this repository. Only you can contribute to this repository.

Manage access

You haven't invited any collaborators yet

 Click here to add a collaborator

[Add people](#)

Search or jump to... / Pull requests Issues Marketplace Explore

sibgit / sibgit.github.io Public

Code Issues Pull requests Actions Projects Wiki Security Insights Settings

General

Access

**Collaborators** (highlighted)

Moderation options

Code and automation

Branches

Tags

Actions

Webhooks

Environments

Pages

Security

Code security and analysis

Deploy keys

Secrets

Integrations

GitHub apps

Email notifications

Who has access

PUBLIC REPOSITORY

This repository is public and visible to anyone.

Manage

DIRECT ACCESS

26 have access to this repository.  
17 collaborators. 9 invitations.

Click here to add a collaborator

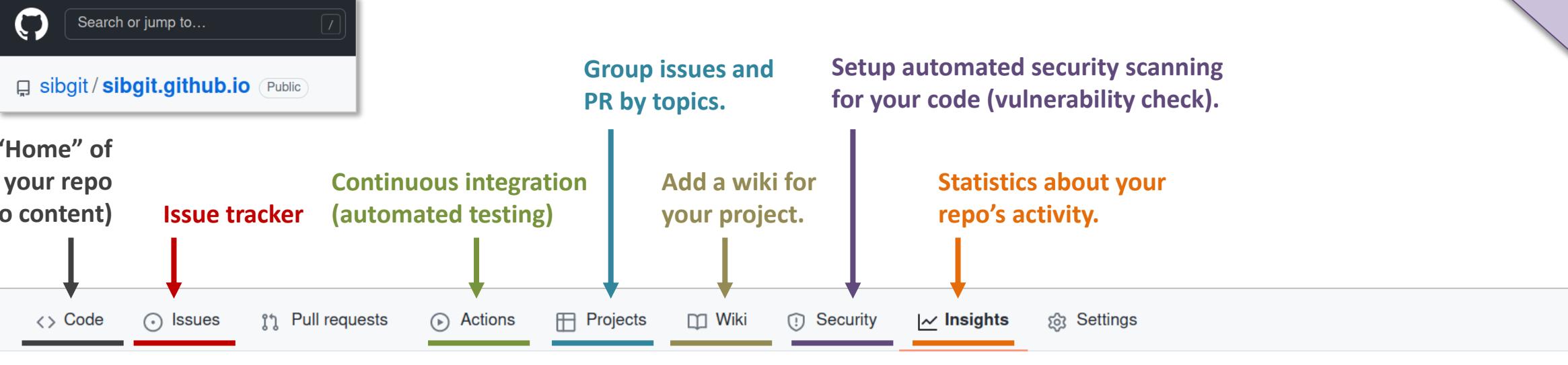
Add people

Select all

Find a collaborator...

alinefuchs	Awaiting alinefuchs's response	Pending Invite	
AmirKH	Awaiting AmirKhalilzadeh's response	Pending Invite	
AurelieLen	Awaiting AurelieLen's response	Pending Invite	
Burulca	burulca • Collaborator		
christec5	Awaiting christec5's response	Pending Invite	

## Other GitHub features (some of them)



The screenshot shows a GitHub repository page for `sibgit / sibgit.github.io`. The top navigation bar includes a search bar and a link to the repository's website. Below the navigation, there are several tabs: `Code`, `Issues` (highlighted in red), `Pull requests`, `Actions` (highlighted in green), `Projects` (highlighted in blue), `Wiki`, `Security`, `Insights` (highlighted in orange), and `Settings`.

Annotations with arrows point from specific features to their corresponding sections:

- “Home” of your repo (repo content)** points to the `Code` tab.
- Issue tracker** points to the `Issues` tab.
- Continuous integration (automated testing)** points to the `Actions` tab.
- Group issues and PR by topics.** points to the `Projects` tab.
- Add a wiki for your project.** points to the `Wiki` tab.
- Setup automated security scanning for your code (vulnerability check).** points to the `Security` tab.
- Statistics about your repo’s activity.** points to the `Insights` tab.

**Network graph**

Timeline of the most recent commits to this repository and its network ordered by most recently pushed to.

Owners	Feb	Mar
sibgit	19 17	10
		master
		manta-dev

**Network**

- Pulse
- Contributors
- Community
- Community Standards
- Traffic
- Commits
- Code frequency
- Dependency graph
- Network
- Forks



# Project home page on GitLab

Example of the “home page” of a repository on GitLab

The screenshot shows the GitLab interface for the repository 'awesome-animal-awareness'. On the left, a sidebar lists various project management and monitoring options. The main area displays a list of files in the repository, including 'img', '.gitlab-ci.yml', 'README.md', and several HTML files ('alpaca.html', 'blue\_whale.html', 'dahu.html', 'gorilla.html', 'index.html', 'kiwi\_bird.html', 'manta\_ray.html'). A red arrow points from the text 'Branch you are currently viewing' to the 'main' branch selection in the top navigation bar. Another red arrow points from the text 'List of files present in the repo.' to the file list. A third red arrow points from the text 'If you have a README.md file, it is displayed here (with markdown rendering).' to the content of the 'README.md' file, which is shown in a preview window at the bottom.

**Branch you are currently viewing**

**List of files present in the repo.**

If you have a **README.md** file, it is displayed here (with markdown rendering).

**A awesome-animal-awareness**

main / awesome-animal-awareness / +

web: add img subdirectory to store images  
Robin Engler authored 3 hours ago

Name	Last commit	Last update
img	web: add img subdirectory to store ima...	1 minute ago
.gitlab-ci.yml	cicd: add .gitlab-ci.yml file	1 minute ago
README.md	doc: add README.md	1 minute ago
alpaca.html	web: add animal page templates	1 minute ago
blue_whale.html	web: add animal page templates	1 minute ago
dahu.html	web: add animal page templates	1 minute ago
gorilla.html	web: add animal page templates	1 minute ago
index.html	web: rename home page to Awesome A...	1 minute ago
kiwi_bird.html	web: add animal page templates	1 minute ago
manta_ray.html	web: add animal page templates	1 minute ago

**README.md**

Awesome Animal Awareness Project

Welcome to the Awesome Animal Awareness Project.  
To visit our website, go to: <https://sib-git-training.gitlab.io/awesome-animal-awareness>

**Copy the project's URL (e.g. to git clone it)**

History Find file Edit Code

Project information

8 Commits 2 Branches 0 Tags 1.1 MiB Project Storage 1 Environment

Clone with SSH  
git@gitlab.com:sib-git-training/ [Copy URL](#)

Clone with HTTPS  
<https://gitlab.com/sib-git-train> [Copy URL](#)

**Copy the project's URL (e.g. to git clone it)**



# GitLab “project” menu



The screenshot shows the main project navigation menu. Several items are highlighted with dashed orange boxes:

- Manage**: Under Project
- Plan**: Under Project
- Code**: Under Project
- Build**: Under Project
- Secure**: Under Project
- Deploy**: Under Project
- Operate**: Under Project
- Monitor**: Under Project
- Analyze**: Under Project
- Settings**: Under Project

The screenshot shows the 'Manage' section of the project menu. It includes:

- Activity
- Members
- Labels

Add people to your project

The screenshot shows the 'Code' section of the project menu. It includes:

- Merge requests (0)
- Repository**
- Branches
- Commits**
- Tags
- Repository graph**
- Compare revisions
- Snippets

Repo home page

List of commits

History graph of your project

The screenshot shows the 'Activity' section of the project menu. It displays a list of recent commits:

- Jan 31, 2024:
  - main → manta-ray: add behavior information
  - manta-ray: add distribution and image
  - manta-ray: add animal name and diet
  - main → web: add img subdirectory to store images
  - cicd: add .gitlab-ci.yml file
  - web: rename home page to Awesome Animal Awareness Project
  - web: add animal page templates
  - styles: change paragraphs fonts
  - styles: add styles.css file
  - doc: add README.md
  - first commit

The screenshot shows the 'Repository graph' of the project. It displays a timeline from Jan 31 to Feb 19, 2024, showing the commit history:

- Jan 31: manta-ray → manta-ray: add behavior information
- manta-ray: add distribution and image
- manta-ray: add animal name and diet
- main → web: add img subdirectory to store images
- cicd: add .gitlab-ci.yml file
- web: rename home page to Awesome Animal Awareness Project
- web: add animal page templates
- styles: change paragraphs fonts
- styles: add styles.css file
- doc: add README.md
- first commit

All settings of your project

## Cloning a repo: HTTPS vs. SSH

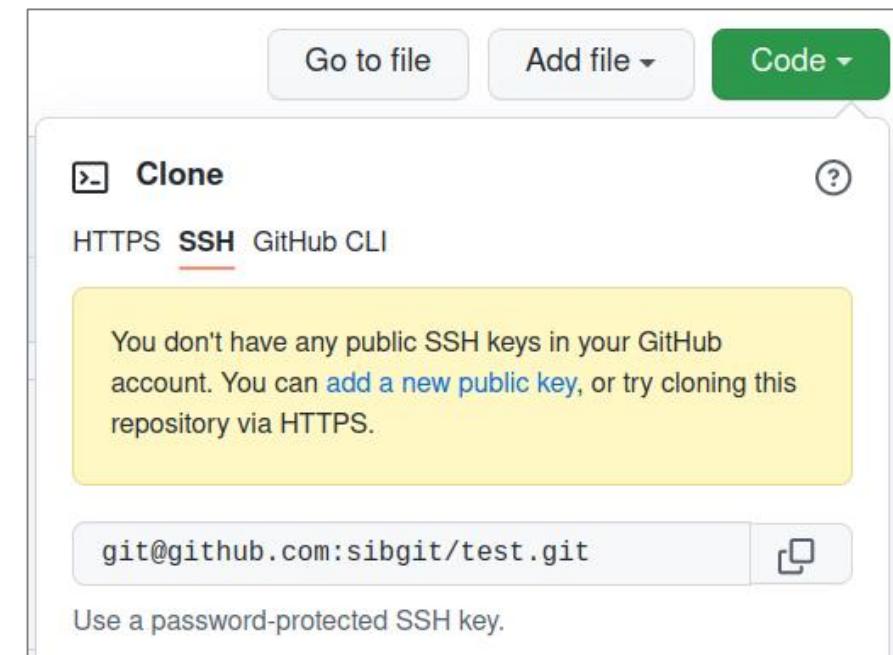
HTTPS and SSH are two different network protocols that machines can use to communicate.

When cloning (or adding a remote) via:

- **HTTPS**, you will need to provide a **personal access token (PAT)** as authentication credential.
  - If the repo is public, credentials are only needed to push data to the remote (not to pull).
  - Your local Git repo will in principle store the login credentials, so you need to provide them only once.
- **SSH**, you will need to add your **public SSH key** to your GitHub account.



The screenshot shows the GitHub 'Clone' interface for a repository at <https://github.com/sibgit/test.git>. The 'HTTPS' tab is selected. A note below says: 'Use Git or checkout with SVN using the web URL.'



The screenshot shows the GitHub 'Clone' interface for the same repository, but the 'SSH' tab is selected. A yellow callout box contains the message: 'You don't have any public SSH keys in your GitHub account. You can [add a new public key](#), or try cloning this repository via HTTPS.' Below this, the SSH URL `git@github.com:sibgit/test.git` is shown, along with the note: 'Use a password-protected SSH key.'

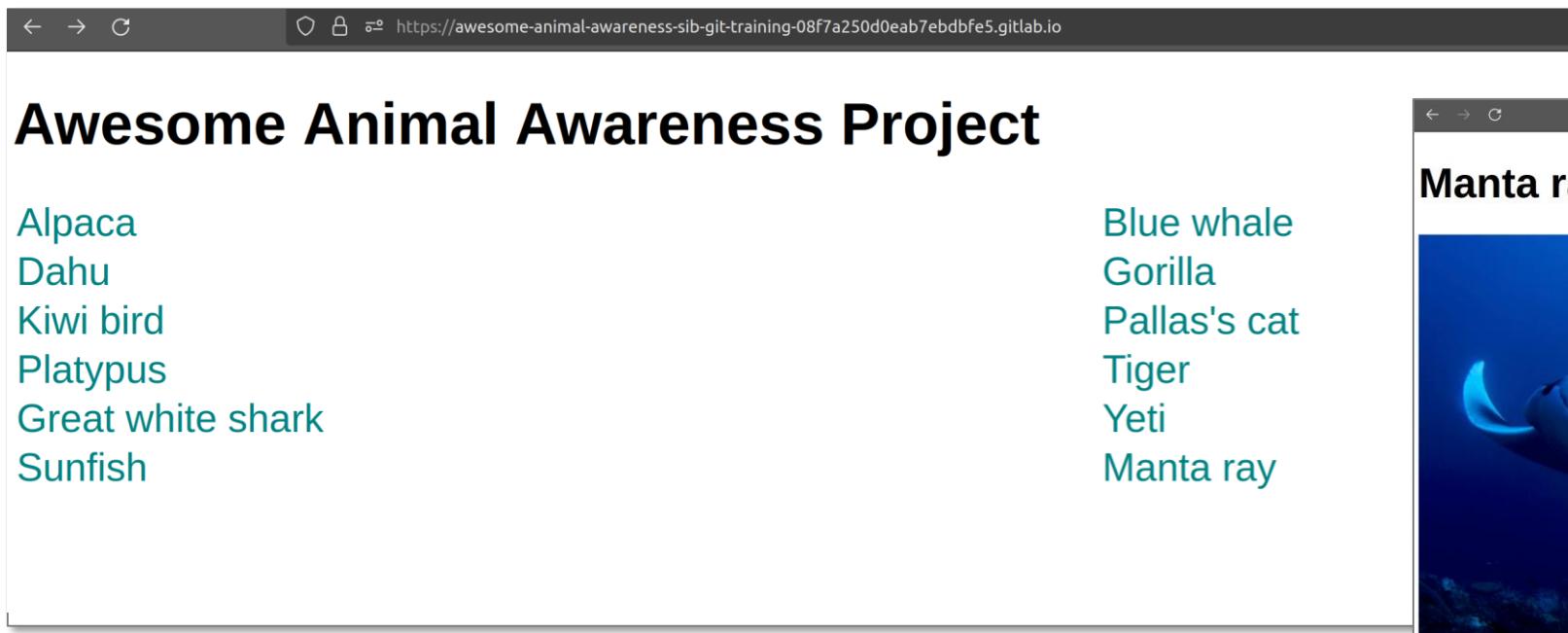
Reminder: command to clone a repo (here via https)

```
$ git clone https://github.com/sibgit/test.git
```

# Pull Requests (GitHub) and Merge Requests (GitLab)

## An introduction to the upcoming exercise 4...

In exercise 4, we will all work together on building a website for the **Awesome Animal Awareness project!**

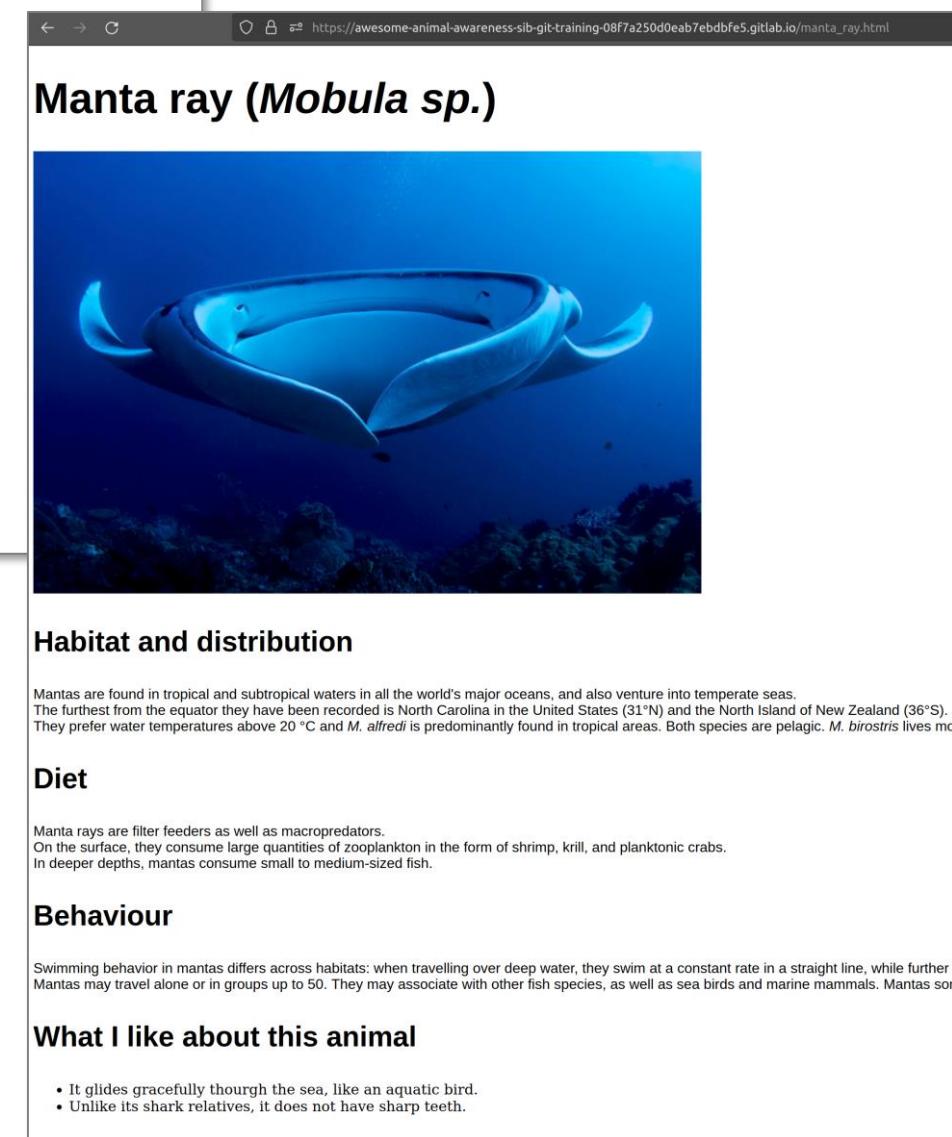


The screenshot shows a web browser window with the URL <https://awesome-animal-awareness-sib-git-training-08f7a250d0eab7ebdbfe5.gitlab.io>. The page title is "Awesome Animal Awareness Project". On the left, there is a list of animals: Alpaca, Dahu, Kiwi bird, Platypus, Great white shark, Sunfish. On the right, there is another list: Blue whale, Gorilla, Pallas's cat, Tiger, Yeti, Manta ray.

### How we will work:

- We will split into teams of 2-3 people.
- Each team will be responsible for creating the page of an (awesome!) animal \*.
- Within a team, each person will work on a different part of the animal's page (e.g. one person works on the "Habitat and distribution" section, while another works on the "Diet" or "Behavior").

\* Note: every animal in the list is awesome – you can't go wrong!



The screenshot shows a web browser window with the URL [https://awesome-animal-awareness-sib-git-training-08f7a250d0eab7ebdbfe5.gitlab.io/manta\\_ray.html](https://awesome-animal-awareness-sib-git-training-08f7a250d0eab7ebdbfe5.gitlab.io/manta_ray.html). The page title is "Manta ray (*Mobula* sp.)". It features a large, high-quality photograph of a manta ray swimming in deep blue water. Below the image, there are sections for "Habitat and distribution", "Diet", "Behaviour", and "What I like about this animal".

**Habitat and distribution**  
Mantas are found in tropical and subtropical waters in all the world's major oceans, and also venture into temperate seas. The furthest from the equator they have been recorded is North Carolina in the United States (31°N) and the North Island of New Zealand (36°S). They prefer water temperatures above 20 °C and *M. alfredi* is predominantly found in tropical areas. Both species are pelagic. *M. birostris* lives more

**Diet**  
Manta rays are filter feeders as well as macropredators. On the surface, they consume large quantities of zooplankton in the form of shrimp, krill, and planktonic crabs. In deeper depths, mantas consume small to medium-sized fish.

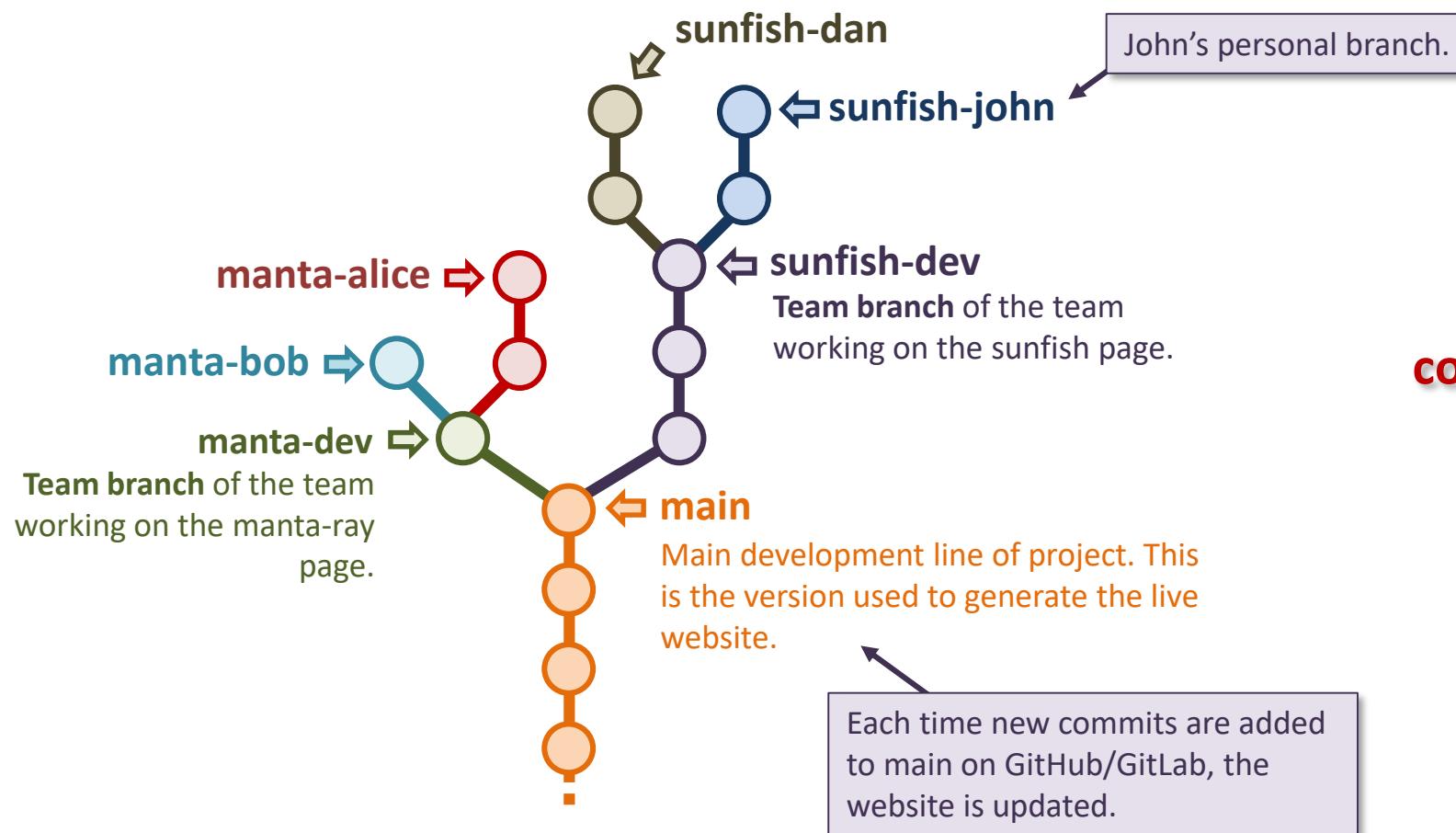
**Behaviour**  
Swimming behavior in mantas differs across habitats: when travelling over deep water, they swim at a constant rate in a straight line, while further Mantas may travel alone or in groups up to 50. They may associate with other fish species, as well as sea birds and marine mammals. Mantas son

**What I like about this animal**

- It glides gracefully through the sea, like an aquatic bird.
- Unlike its shark relatives, it does not have sharp teeth.

## An introduction to the upcoming exercise 4...

- This is how (more or less) our shared repository will look on GitHub/GitLab...
- Changes made to the **main** branch are directly reflected in the production website – so we don't want to mess-up **main** !!
- => You are **not allowed to push directly to main**.

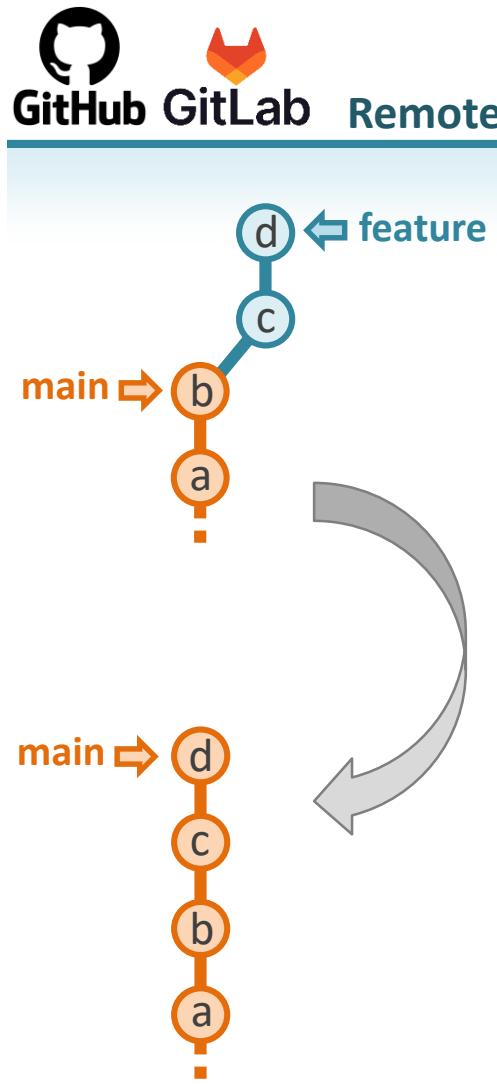
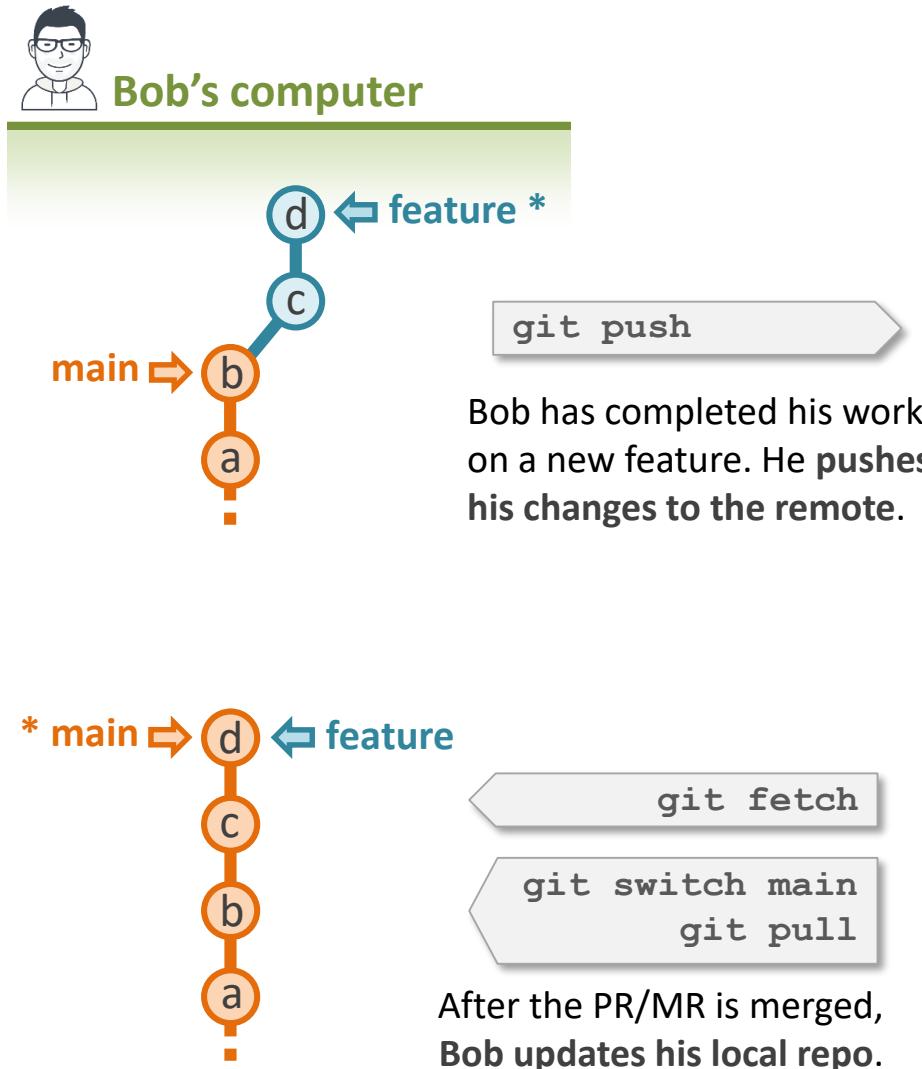


**How are we going to contribute changes from our team branches ?**

# Pull Requests (GitHub) / Merge Requests (GitLab)

**Pull Requests (PR) and Merge Requests (MR)** are a way to perform a merge operation on the remote (on GitHub/GitLab) instead of in your local copy of the repository.

PR/MR are the same thing, they just have different names on GitHub/GitLab.



Why use a PR/MR instead of a local merge (and push) ?

- The branch you want to merge into (e.g. **main**) is **protected** \*.
- Gives the opportunity to the repository owner(s) to **review changes** before merging them.
- Makes it easy to merge changes from a **forked** \*\* repository.

\* **Protected** branches are branches where push operations are limited to users with enough privileges.

\*\* A **fork** is a copy of an entire repository under a new ownership.

The PR/MR workflow:

- Bob opens a PR/MR on GitHub/GitLab.
- Alice reviews the changes made by Bob on branch **feature**.
- Alice approves the PR/MR.
- Bob (or Alice) merges the PR/MR.
- On the remote, the **feature** branch is now merged into **main**. Optionally, **feature** is then deleted.



After the PR/MR is merged, you can pull the changes from the remote to update your local repo (at this point the merge is only on the remote).

```
> git log --all --decorate --oneline --graph
* 9af6e4 (HEAD -> manta-dev, origin/manta-dev) manta-ray: add behavior information
* ba11531 manta-ray: add distribution and image
* 7b0516b manta-ray: add animal name and diet
* 65cb84f (origin/main, origin/HEAD, main) cicd: add .gitlab-ci.yml file
* 90c5dbe web: rename home page to Awesome Animal Awareness Project
* 4401bd7 web: add animal page templates
* 11a7390 styles: change paragraphs fonts
* 30b07cf styles: add styles.css file
* f1828a1 doc: add README.md
* 998ea08 first commit
```

```
> git log --all --decorate --oneline --graph
* 13c625f (origin/main, origin/HEAD) Merge branch 'manta-dev' into 'main'
|\ 
* 9af6e4 (HEAD -> manta-dev) manta-ray: add behavior information
* ba11531 manta-ray: add distribution and image
* 7b0516b manta-ray: add animal name and diet
|/
* 65cb84f (main) cicd: add .gitlab-ci.yml file
* 90c5dbe web: rename home page to Awesome Animal Awareness Project
* 4401bd7 web: add animal page templates
* 11a7390 styles: change paragraphs fonts
```

```
> git log --all --decorate --oneline --graph
* 13c625f (HEAD -> main, origin/main, origin/HEAD) Merge branch 'manta-dev'
|\ 
* 9af6e4 manta-ray: add behavior information
* ba11531 manta-ray: add distribution and image
* 7b0516b manta-ray: add animal name and diet
|/
* 65cb84f cicd: add .gitlab-ci.yml file
* 90c5dbe web: rename home page to Awesome Animal Awareness Project
* 4401bd7 web: add animal page templates
```

Using `git fetch` is optional, it's useful if you want to preview the position of `origin/main` before merging it into your local `main` with `git pull`.

`git fetch --prune`

--prune deletes local references to remote branches (`origin/manta-dev` has been deleted).

`git switch main`  
`git pull --prune`  
`git branch -d manta-dev`

`git switch main`  
`git pull`  
`git branch -d manta-dev`



# Open a Pull Request on GitHub: step-by-step

You will need to do this in exercise 4 !



1. On the project's page on GitHub, go to the **Pull requests** tab.

The screenshot shows a GitHub project page for 'sibgit / sibgit.github.io'. The 'Pull requests' tab is selected, indicated by a red arrow and a callout box labeled 'Pull requests tab'. A modal message 'Label issues and pull requests for new contributors' is displayed, with a 'Dismiss' button. Below the modal, a yellow banner shows a recent push from 'manta-dev'. The main area displays filters ('is:pr is:open'), a summary of 0 Open and 25 Closed pull requests, and a message stating 'There aren't any open pull requests.' A purple arrow points to the 'New pull request' button, which is highlighted with a callout box labeled '2. Click on New pull request.'. A purple box on the left says 'Pending pull requests will be listed here...'. The GitHub logo is in the top right corner.



### 3. Select the branches to merge:

The screenshot shows the GitHub interface for comparing branches. Two dropdown menus are highlighted with orange boxes and purple arrows pointing to them:

- base: master** (Branch to merge into)
- compare: manta-dev** (Branch to merge (your contribution))

Branch to merge into  
Branch to merge (your contribution)

List of commits that will be merged ➡

In this example, there are 2 commits on branch "manta-dev" that will be merged into "master".

Summary of changes introduced by the pull request.

Green lines = new content.  
Red lines = deleted content.

### 4. Click on **Create pull request**.

The screenshot shows the GitHub interface for comparing changes between "base: master" and "compare: manta-dev". The interface includes the following elements:

- Comparing changes** header
- base: master** and **compare: manta-dev** dropdown menus
- Able to merge** message (with a green checkmark)
- Create pull request** button (green)
- Commits on Mar 10, 2022** section:
  - Add info on habitat and behavior for manta ray** (commit by sibgit, 18 minutes ago)
  - Add image for manta ray** (commit by sibgit, 17 minutes ago)
- Showing 2 changed files with 14 additions and 6 deletions.**
- Detailed diff view** for `manta_ray.html`:
 

Line	Change Type	Content
4	+	<link rel="stylesheet" href="styles.css">
5	+	</head>
6	+	<body>
7	-	<h1>?? Animal name</h1>
7	+	<h1>Manta Ray - <i>Mobula sp.</i></h1>
8	+	
9	+	
10	+	<h3>Habitat and distribution</h3>
11	+	<p>
12	+	?? Replace this with a few lines on the animal's habitat and distribution.
13	+	Mantas are found in tropical and subtropical waters in all the world's major oceans,
14	+	and also venture into temperate seas.
15	+	 
16	+	The furthest from the equator they have been recorded is North Carolina in the
17	+	United States, and the North Island of New Zealand.
18	+	 
19	+	They prefer water temperatures above 68 °F (20 °C)
20	+	</p>
- If there are conflicts, you probably need to rebase your branch and resolve them.** (Note in orange text with an orange arrow)



5. Optionally, enter ➔  
a **message** for the  
people that will  
review your pull  
request.

## Open a pull request

Create a new pull request by comparing changes across two branches. If you need to, you can also [compare across forks](#).

base: master ▾ ← compare: manta-dev ▾ ✓ **Able to merge.** These branches can be automatically merged.

Manta dev

Write Preview

I worked hard to add these awesome changes to the manta ray page.  
Please merge :smiley\_cat:

Attach files by dragging & dropping, selecting or pasting them.

Create pull request ▾

Reviewers  
sibgit

Assignees  
No one—assign yourself

Labels  
None yet

Projects  
None yet

Milestone  
No milestone

Development  
Use [Closing keywords](#) in the description to automatically close issues

Remember, contributions to this repository should follow our [GitHub Community Guidelines](#).

6. Submit your pull request by clicking  
**Create pull request**.

The pull request is now **created**, and **awaiting approval** from an authorized person.  
(e.g. the repo owner or a colleague)

The screenshot shows a GitHub pull request page for a repository named 'Manta dev'. The pull request has been opened by 'robinengler' to merge two commits from the 'manta-dev' branch into the 'master' branch. The commit history shows two commits from 'sibgit': one adding info on habitat and behavior for manta ray (commit d0a01b1) and another adding an image for manta ray (commit 0677d8c). A comment from 'robinengler' encourages merging with a smiley face emoji. The status bar at the bottom indicates that merging is blocked because a review is required.

Manta dev #27

Open robinengler wants to merge 2 commits into master from manta-dev

Conversation 0 Commits 2 Checks 0 Files changed 2

robinengler commented now

I worked hard to add these awesome changes to the manta ray page.  
Please merge 😊

sibgit added 2 commits 31 minutes ago

Add info on habitat and behavior for manta ray  
Add image for manta ray

d0a01b1  
0677d8c

Add more commits by pushing to the manta-dev branch on sibgit/sibgit.github.io.

This branch has not been deployed  
No deployments

Review required  
At least 1 approving review is required by reviewers with write access. [Learn more](#).

Merging is blocked  
Merging can be performed automatically with 1 approving review.

Merge pull request or view command line instructions.

The **reviewer** of your PR will then have a look at your changes (the modifications introduced with your commits) and **approve them or request changes**.

The screenshot illustrates the GitHub pull request review process. At the top, the GitHub logo is visible, followed by a search bar and navigation links for Pull requests, Issues, Marketplace, and Explore. The repository sibgit/sibgit.github.io is selected.

The main area shows a list of pull requests. A purple dashed box highlights the first pull request, labeled "Manta dev". Below it, a purple arrow points down to a detailed view of the pull request #27. This view includes the author's comment, commit details, and deployment status. Another purple arrow points from the pull request list to the "Review changes" interface, which is shown in a separate window on the right. This interface allows the reviewer to write a review, attach files, and choose between Comment, Approve, or Request changes. A green "Submit review" button is at the bottom.

## Manta dev #27

[Open](#) robinengler wants to merge 2 commits into `master` from `manta-dev`

Conversation 1 Commits 2 Checks 0 Files changed 2

robinengler commented 7 minutes ago  
I worked hard to add these awesome changes to the manta ray page.  
Please merge 😊

sibgit added 2 commits 38 minutes ago

- Add info on habitat and behavior for manta ray d0a01b1
- Add image for manta ray 0677d8c

sibgit approved these changes 1 minute ago

sibgit left a comment  
Looking good, thanks for the contribution !

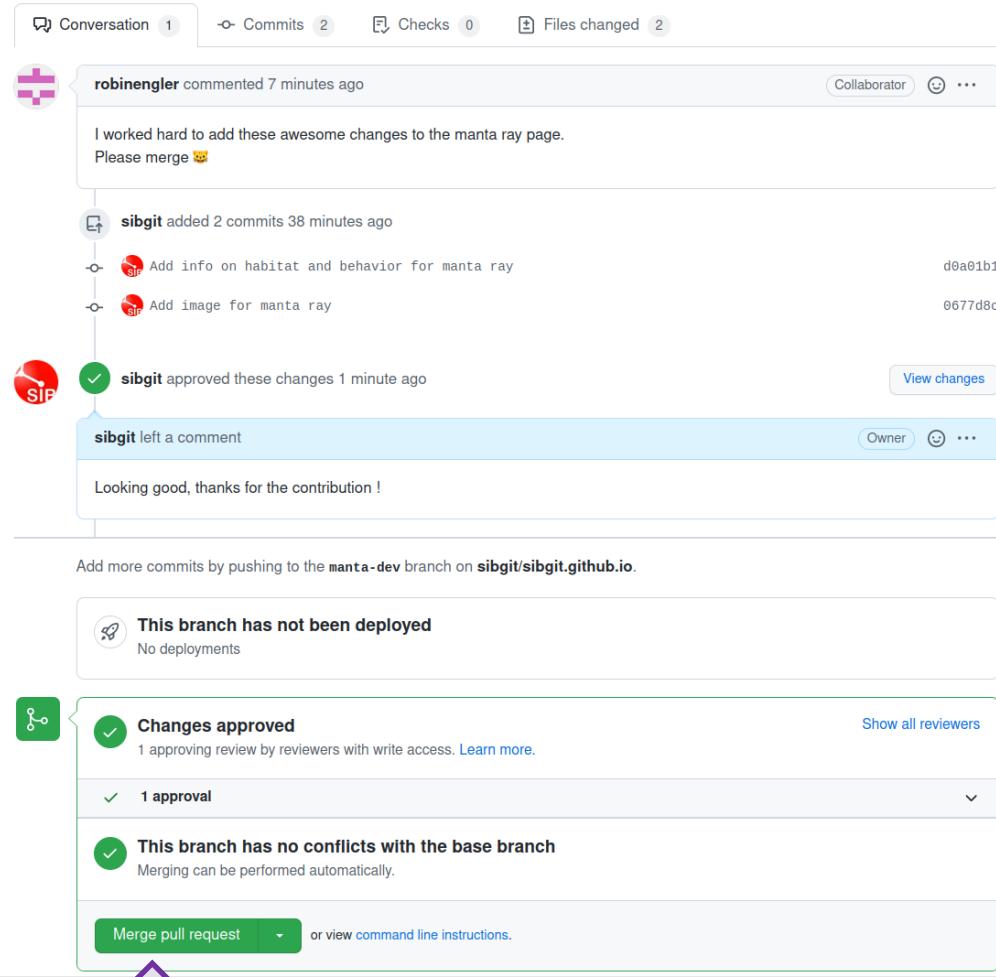
Add more commits by pushing to the `manta-dev` branch on [sibgit/sibgit.github.io](https://sibgit/sibgit.github.io).

This branch has not been deployed  
No deployments

Changes approved  
1 approving review by reviewers with write access. [Learn more](#).  
1 approval

This branch has no conflicts with the base branch  
Merging can be performed automatically.

[Merge pull request](#) or view [command line instructions](#).



Now that the pull request is approved, it can be merged (either by the reviewer or by you) by clicking **Merge pull request**.

## Manta dev #27

[Open](#) robinengler wants to merge 2 commits into `master` from `manta-dev`

Conversation 1 Commits 2 Checks 0 Files changed 2

robinengler commented 9 minutes ago  
I worked hard to add these awesome changes to the manta ray page.  
Please merge 😊

sibgit added 2 commits 40 minutes ago

- Add info on habitat and behavior for manta ray d0a01b1
- Add image for manta ray 0677d8c

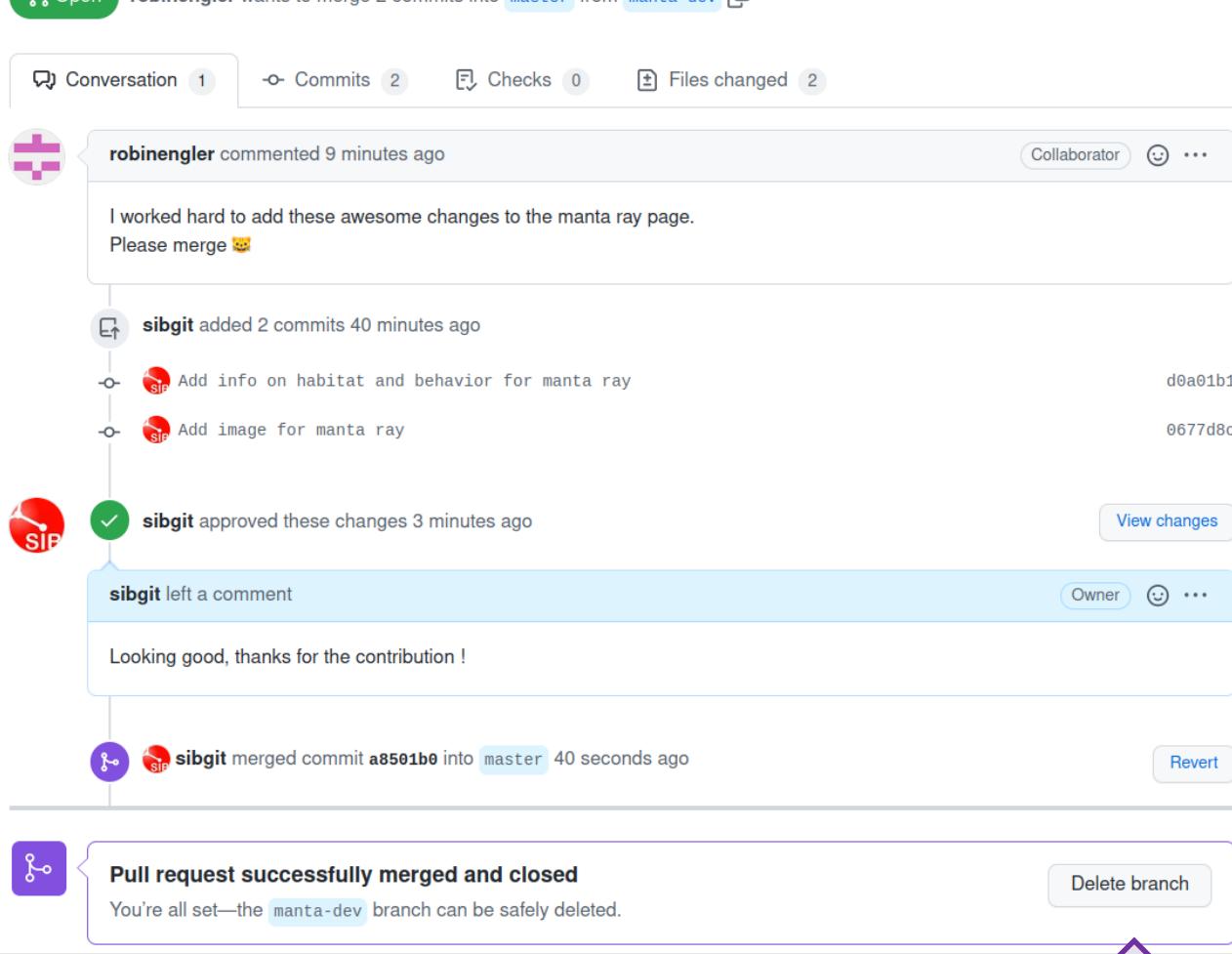
sibgit approved these changes 3 minutes ago

sibgit left a comment  
Looking good, thanks for the contribution !

sibgit merged commit `a8501b0` into `master` 40 seconds ago

Pull request successfully merged and closed  
You're all set—the `manta-dev` branch can be safely deleted.

Delete branch



Completed ! Optionally, you can **delete your branch** on the remote (this will not delete it locally).



# Open a Merge Request on GitLab: step-by-step

You will need to do this in exercise 4 !



1. On the project's page on GitLab, use the left-hand side menu to navigate to **Code > Merge requests**.
2. Click on **New merge request**, or on **Create merge request** if your branch is already listed (as is the case with “manta-dev” in the example).

The screenshot shows the GitLab interface for the project "awesome-animal-awareness".

- Left sidebar:** Shows the project navigation menu with "Merge requests" selected.
- Middle top:** Shows the "Merge requests" page with a message: "You pushed to `manta-dev` 3 minutes ago". A red arrow points to the "Create merge request" button, which is highlighted with a red dashed oval.
- Middle bottom:** Shows the "Merge requests" list with one item: "Manta dev" (1 merged, 0 closed). A red arrow points to the "New merge request" button in the top right corner of the list area, which is also highlighted with a red dashed oval.
- Right side:** A circular icon with two nodes connected by arrows, representing a workflow. Below it, text explains: "Merge requests are a place to propose changes you've made to a project and discuss those changes with others. Interested parties can even contribute by pushing commits if they want to." A red arrow points to the "New merge request" button on the right side of the text area, which is highlighted with a red dashed oval.

3. On the next screen, select the branch to merge (in exercise 4, this is your team branch branch) as **Source branch**, and “main” as **Target branch**.

Then click on **Compare branches and continue**.

Note: if you have clicked on **Create merge request** at step 2, this step will be skipped as the correct target and source branches will be automatically selected for you by GitLab.

The screenshot shows the "New merge request" form.

- Source branch:** Set to "sib-git-training/awesome-animal-aw...". A red arrow points to the dropdown menu.
- Target branch:** Set to "main". A red arrow points to the dropdown menu.
- Buttons:** The "Compare branches and continue" button is highlighted with a red dashed oval and a red arrow pointing to it.



## New merge request

From manta-dev into main Change branches

**Title (required)**

Manta dev

**Mark as draft**  
Drafts cannot be merged until marked ready.

## Description

Preview | B I  $\frac{1}{2}$  |  $\equiv$   $\langle\rangle$   $\wp$   $\equiv$   $\frac{1}{2}$   $\exists$   $\forall$  |  $\oplus$   $\otimes$  |  $\mathbb{C}$   $\mathbb{R}$

Please merge my changes for the manta ray into main

Switch to rich text editing

Add [description templates](#) to help your contributors to communicate effectively.

**Assignee**

Unassigned ▼

## Reviewer

Unassigned

Approvals are optional.

## > Approval rules

Milestone

Select m

## Merge options

Delete source branch when merge request is accepted.

Squash commits when merge request is accepted 

© 2010 Pearson Education, Inc., publishing as Pearson Addison Wesley

Create merge request

- 4. Give a **Title** to your merge request (MR). A default Title will be pre-set. Optionally you can enter a description.

5. At the bottom of the page, you can see the commits that are part of the MR (in this example, there are 3 commits).

Create merge request Cancel

Commits 3 Changes 3

Jan 30, 2024

-  manta-ray: add behavior information  
Robin Engler authored 22 minutes ago
-  manta-ray: add distribution and image  
Robin Engler authored 25 minutes ago
-  manta-ray: add animal name and diet  
Robin Engler authored 2 hours ago

6. Click on **Create merge request** to create the MR.



The pull request is now **created**, and awaiting approval from an authorized person (e.g. the repo owner or a colleague).

## Manta dev

Open Robin Engler requested to merge `manta-dev` into `main` 2 minutes ago

Overview 0 Commits 3 Pipelines 0 Changes 3

Please merge my changes for the manta ray into main.

0 0

8✓ Approve Approval is optional

Ready to merge!

Delete source branch  Edit commit message  
3 commits and 1 merge commit will be added to main.

**Merge**

### Activity

Preview |

Write a comment or drag your files here...

Switch to rich text editing

Make this an internal note

**Comment** **Close merge request**

SIB Git training / awesome-animal-awareness / Merge requests / !1

Open Manta dev `manta-dev` into `main`

Overview 0 Commits 3 Pipelines 0 Changes 2

Search (e.g. \*.vue) (Ctrl+P)

img

- `img_manta.jpg` +0 -0
- `manta_ray.html` +32 -6

**manta\_ray.html**

4	4	00 -4,28 +4,54 00	<link rel="stylesheet" href="styles.css">
5	5		</head>
6	6		<body>
7	-		<h1>?? Animal name</h1>
8	+		<h1>Manta ray (<em>Mobula sp.</em>)</h1>
9	-		
10	+		
11	11		<h3>Habitat and distribution</h3>
12	12		<p>
13	-		?? Replace this with a few lines on the animal's habitat and distribution.
14	+		Mantas are found in tropical and subtropical waters in all the world's major oceans, and also venture into temperate seas.
15	+		 
16	+		The furthest from the equator they have been recorded is North Carolina in the United States (31°N) and the North Island of New Zealand (36°S).
17	+		 
18	+		They prefer water temperatures above 20 °C and <em>M. alfredi</em> is predominantly found in tropical areas. Both species are pelagic.
19	+		<em>M. birostris</em> lives mostly in the open ocean, travelling with the currents and migrating to areas where upwellings of nutrient-rich water increase prey concentrations.
20	+		
21	+		
22	+		
23	+		
24			</p>

The **reviewer** of your PR will then have a look at your changes (the modifications introduced with your commits) and **approve them or request changes**.



When the merge request is **approved**, it can be merged by clicking on **Merge**.

## Manta dev

Open Robin Engler requested to merge [manta-dev](#) into [main](#) 7 minutes ago

Overview 0 Commits 3 Pipelines 0 Changes 2

Please merge my changes for the manta ray into main.

0 0

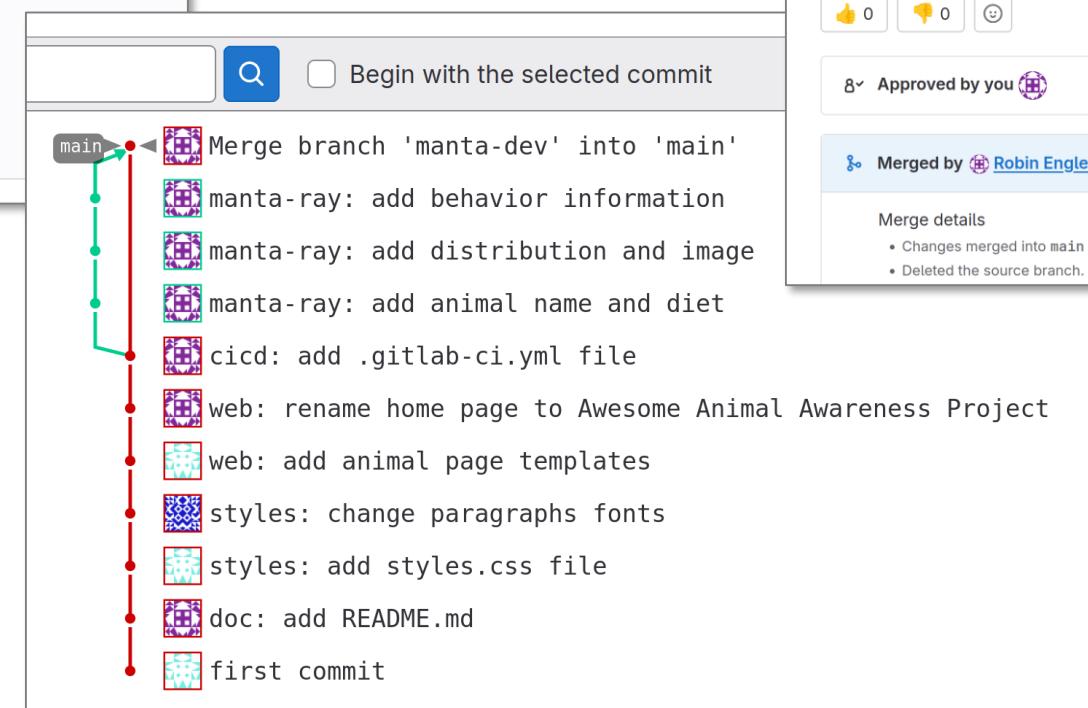
8✓ Revoke approval Approved by you

Ready to merge!

Delete source branch  Edit commit message

3 commits and 1 merge commit will be added to main.

**Merge**



## Manta dev

Merged Robin Engler requested to merge [manta-dev](#) into [main](#) 7 minutes ago

Overview 0 Commits 3 Pipelines 0 Changes 2

Please merge my changes for the manta ray into main.

0 0

8✓ Approved by you

Merged by [Robin Engler](#) just now

Merge details

- Changes merged into main with [13c625fe](#).
- Deleted the source branch.

**Done!** The MR is now merged, the changes from the branch are now part of the “main” branch of the repository.

# Personal Access Tokens (PAT)

## on GitHub or GitLab

# Personal access tokens (PAT) on GitHub/GitLab

Pushing data to a remote requires **some form of authentication...**  
... otherwise anyone could push anything to your remotes!

For security reasons, GitHub does not allow using your user name and password for authentication when running a git push command. Instead you need to use a **personal access token (PAT)**.

In **exercise 4** you will need a PAT to push commits to GitHub/GitLab \*

**Let's generate a PAT together now...**

\* Alternatively, you can also authenticate to GitHub/GitLab using SSH keys. If your account is already setup to use SSH keys, then you don't need a PAT.

## Select scopes

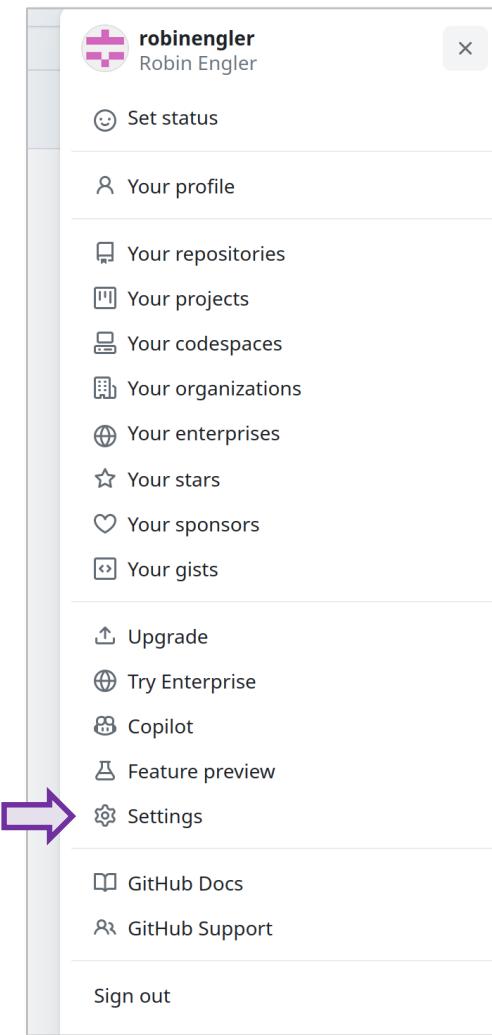
Scopes define the access for personal tokens. [Read more about OAuth scopes.](#)

<input checked="" type="checkbox"/> <b>repo</b>	Full control of private repositories
<input type="checkbox"/> repo:status	Access commit status
<input type="checkbox"/> repo_deployment	Access deployment status
<input type="checkbox"/> public_repo	Access public repositories
<input type="checkbox"/> repo:invite	Access repository invitations
<input type="checkbox"/> security_events	Read and write security events
<input type="checkbox"/> <b>workflow</b>	Update GitHub Action workflows
<input type="checkbox"/> <b>write:packages</b>	Upload packages to GitHub Package Registry
<input type="checkbox"/> read:packages	Download packages from GitHub Package Registry
<input type="checkbox"/> <b>delete:packages</b>	Delete packages from GitHub Package Registry
<input type="checkbox"/> <b>admin:org</b>	Full control of orgs and teams, read and write org projects
<input type="checkbox"/> write:org	Read and write org and team membership, read and write org projects
<input type="checkbox"/> read:org	Read org and team membership, read org projects
<input type="checkbox"/> manage_runners:org	Manage org runners and runner groups
<input type="checkbox"/> <b>admin:public_key</b>	Full control of user public keys
<input type="checkbox"/> write:public_key	Write user public keys
<input type="checkbox"/> read:public_key	Read user public keys
<input type="checkbox"/> <b>admin:repo_hook</b>	Full control of repository hooks
<input type="checkbox"/> write:repo_hook	Write repository hooks
<input type="checkbox"/> read:repo_hook	Read repository hooks
<input type="checkbox"/> <b>admin:org_hook</b>	Full control of organization hooks
<input type="checkbox"/> <b>gist</b>	Create gists
<input type="checkbox"/> <b>notifications</b>	Access notifications
<input type="checkbox"/> <b>user</b>	Update ALL user data
<input type="checkbox"/> read:user	Read ALL user profile data
<input type="checkbox"/> user:email	Access user email addresses (read-only)
<input type="checkbox"/> user:follow	Follow and unfollow users
<input type="checkbox"/> <b>delete_repo</b>	Delete repositories
<input type="checkbox"/> <b>write:discussion</b>	Read and write team discussions
<input type="checkbox"/> read:discussion	Read team discussions
<input type="checkbox"/> <b>admin:enterprise</b>	Full control of enterprises
<input type="checkbox"/> manage_runners:enterprise	Manage enterprise runners and runner groups
<input type="checkbox"/> manage_billing:enterprise	Read and write enterprise billing data
<input type="checkbox"/> read:enterprise	Read enterprise profile data

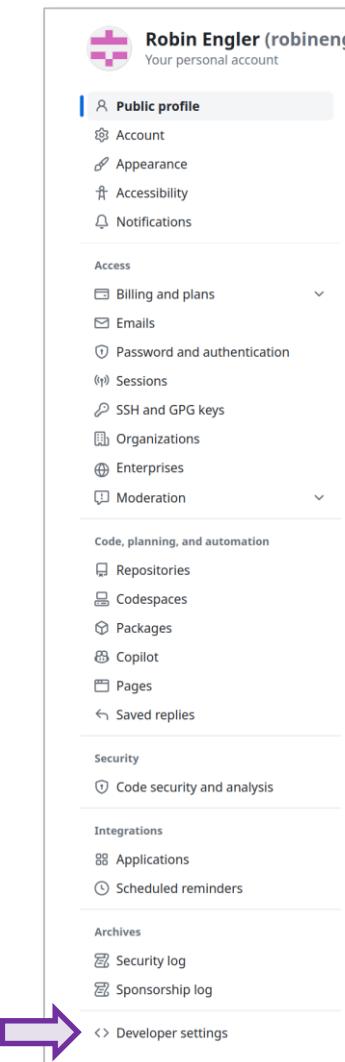
# Generating a “personal access token” (PAT) on GitHub

In order to push data (commits) to GitHub, you will need a **personal access token (PAT)**.

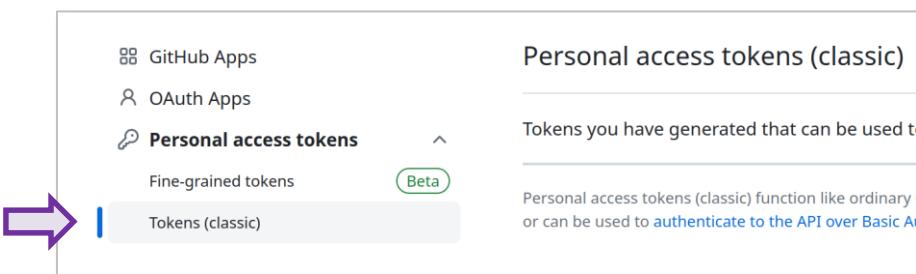
1. In your user profile (top right), click on **Settings**.



2. In your Account settings, click on **Developer settings** (at the very bottom of the list)



3. In **Developer settings**, click on **Personal access tokens**, and select **Tokens (classic)**.



4. Click on **Generate new token**, and select **(classic)**.



Go to next page 

5. Add a **Note** (description) to your token and select the **repo** scope checkbox. Then click **Generate token**.

New personal access token (classic)

Personal access tokens (classic) function like ordinary OAuth access tokens. They can be used instead of a password for Git over HTTPS, or can be used to [authenticate to the API over Basic Authentication](#).

**Note**

Repo access token

What's this token for?

**Expiration \***

30 days The token will expire on Thu, Nov 2 2023

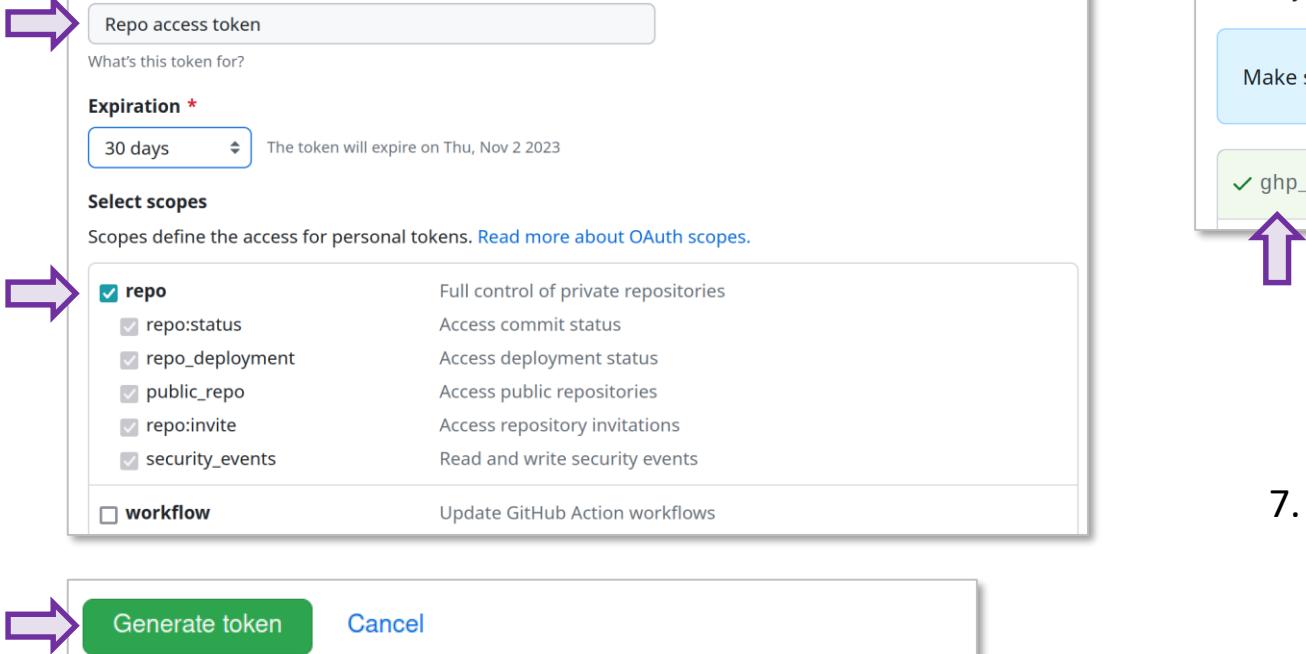
**Select scopes**

Scopes define the access for personal tokens. [Read more about OAuth scopes](#).

repo Full control of private repositories  
 repo:status Access commit status  
 repo\_deployment Access deployment status  
 public\_repo Access public repositories  
 repo:invite Access repository invitations  
 security\_events Read and write security events

workflow Update GitHub Action workflows

**Generate token** Cancel



6. **Copy the personal access token** to a safe locations (ideally in a password manager). You will not be able to access it again later.

Personal access tokens (classic)

Generate new token ▾ Revoke all

Tokens you have generated that can be used to access the [GitHub API](#).

Make sure to copy your personal access token now. You won't be able to see it again!

✓ ghp\_GY9IbuAsGDH4REh4tDc16CxicIWXJe0uMNpx 

Delete



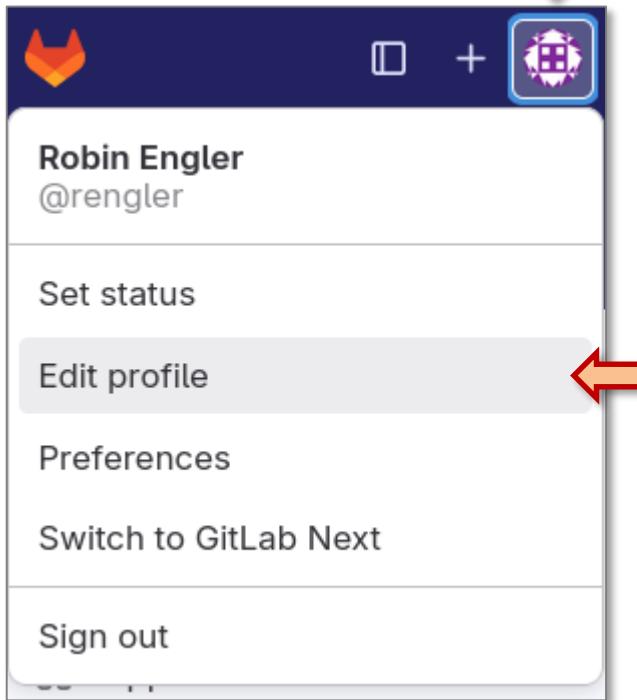
7. When you will push content to GitHub for the first time in the project, you will be asked for your user name and password. **Instead of the password**, enter the **personal access token** you just created.



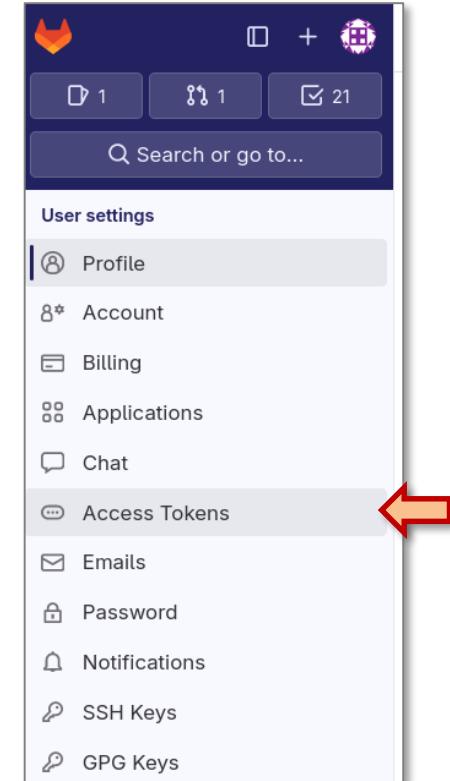
# Generating a “personal access token” (PAT) on GitLab

In order to push data (commits) to GitLab, you will need a **personal access token (PAT)**.

1. Click on your user icon (top left), and select **Edit profile**.



2. In your User settings menu (on the left side), click on **Access Tokens**.



3. On the Personal Access Tokens page, click on **Add new token**.

A screenshot of the Personal Access Tokens page. It shows a table of active tokens and a button labeled "Add new token" at the bottom right. A red arrow points to this button.

Active personal access tokens (2)						
Token name	Scopes	Created	Last Used	Expires	Action	
GitLab container registry read-write	read_registry, write_registry	Mar 27, 2023	10 months ago	in 3 months		
GitLab repo access token	read_repository, write_repository	Oct 08, 2023	3 months ago	in 8 months		



4. Give a **Token name** to your token. You can leave the **Expiration date** empty, so your token will be valid for 1 year.

5. Select **read\_repository** and **write\_repository** as scopes.

6. Click **Generate personal access token**.

**Personal Access Tokens**

Active personal access tokens 2

**Add a personal access token**

**Token name**  
GitLab repo read/write access

For example, the application using the token or the purpose of the token.

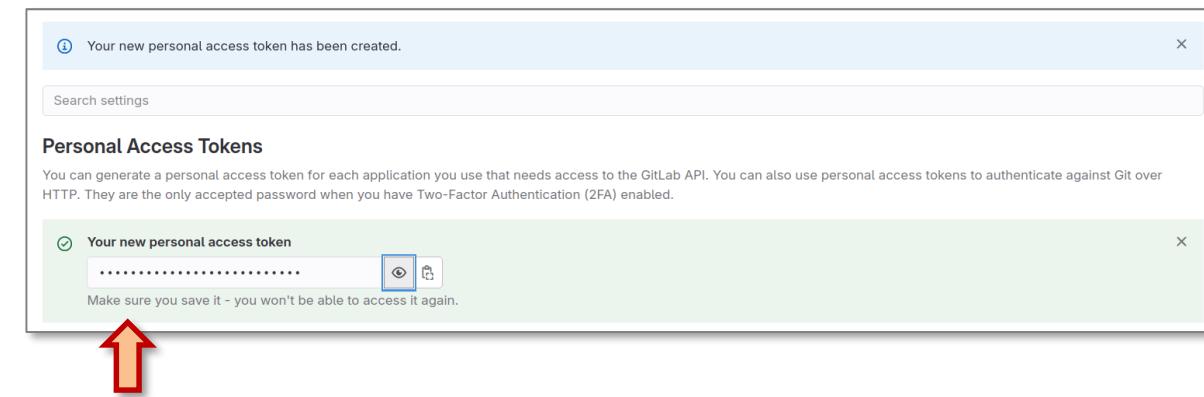
**Expiration date**  
YYYY-MM-DD

**Select scopes**  
Scopes set the permission levels granted to the token. [Learn more.](#)

- api**  
Grants complete read/write access to the API, including all groups and projects, the container registry, the dependency registry, and the package registry.
- read\_api**  
Grants read access to the API, including all groups and projects, the container registry, and the package registry.
- read\_user**  
Grants read-only access to the authenticated user's profile through the /user API endpoint, which includes user API endpoints under /users.
- create\_runner**  
Grants create access to the runners.
- k8s\_proxy**  
Grants permission to perform Kubernetes API calls using the agent for Kubernetes.
- read\_repository**  
Grants read-only access to repositories on private projects using Git-over-HTTP or the Repository Files API.
- write\_repository**  
Grants read-write access to repositories on private projects using Git-over-HTTP (not using the API).
- read\_registry**  
Grants read-only access to container registry images on private projects.
- write\_registry**  
Grants write access to container registry images on private projects.
- ai\_features**  
Grants access to GitLab Duo related API endpoints.

**Create personal access token** **Cancel**

7. Copy the personal access token to a safe locations (ideally in a password manager). You will **not be able to access it again** later.



8. When you will push content to GitLab for the first time in the project, you will be asked for your user name and password. Instead of the password, enter the **personal access token** you just created.

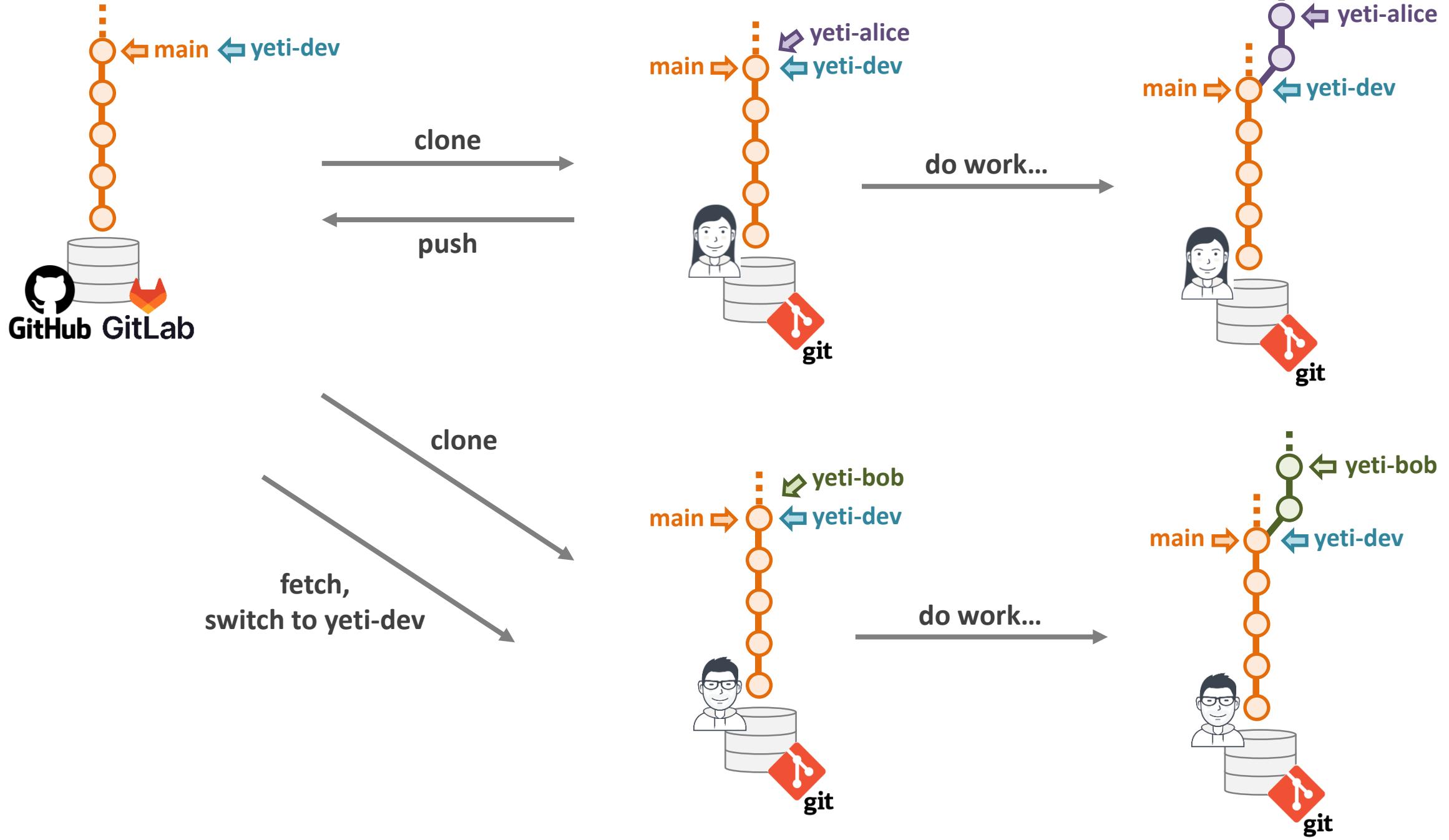
# exercise 4

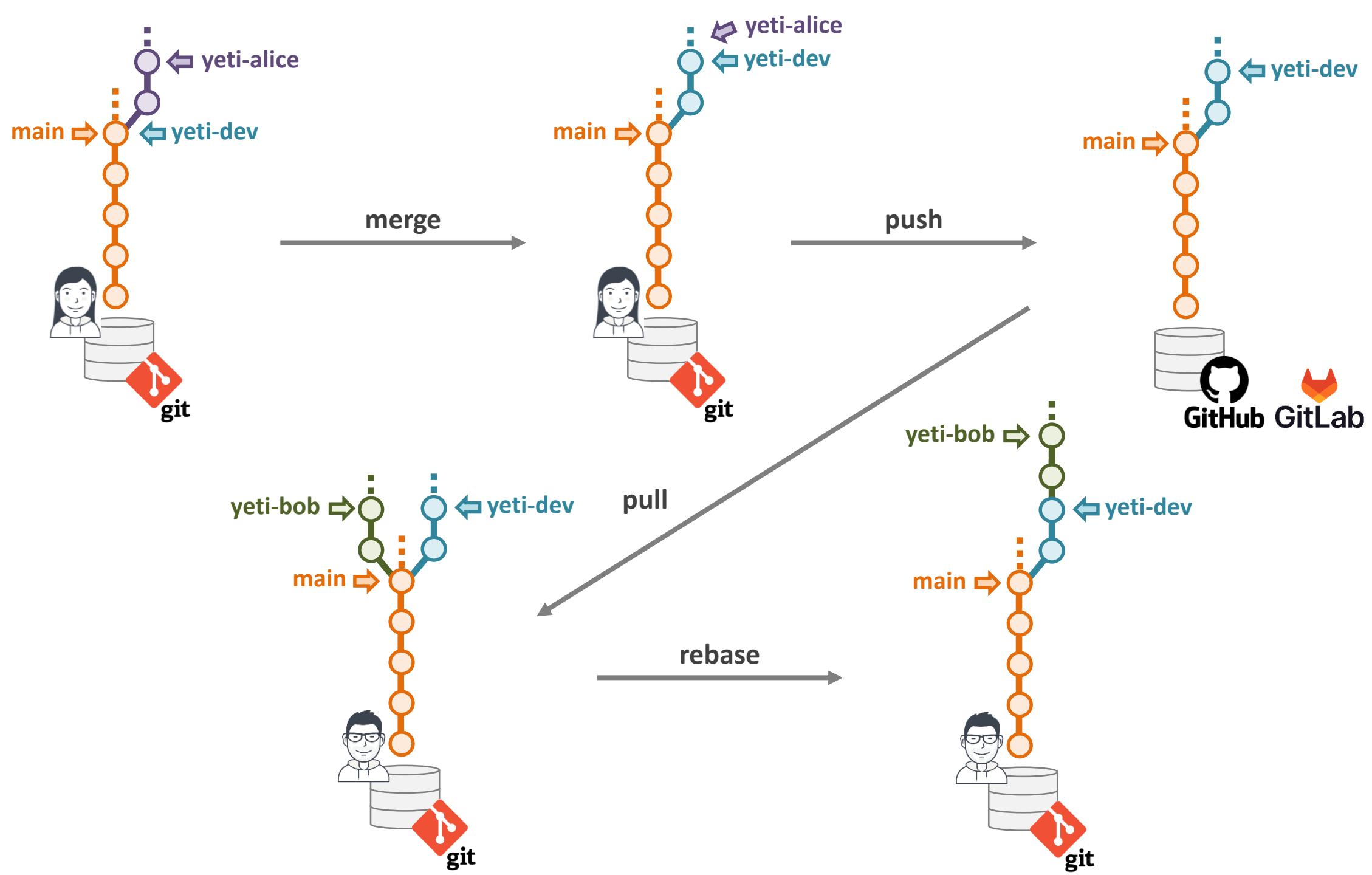
## The Awesome Animal Awareness Project

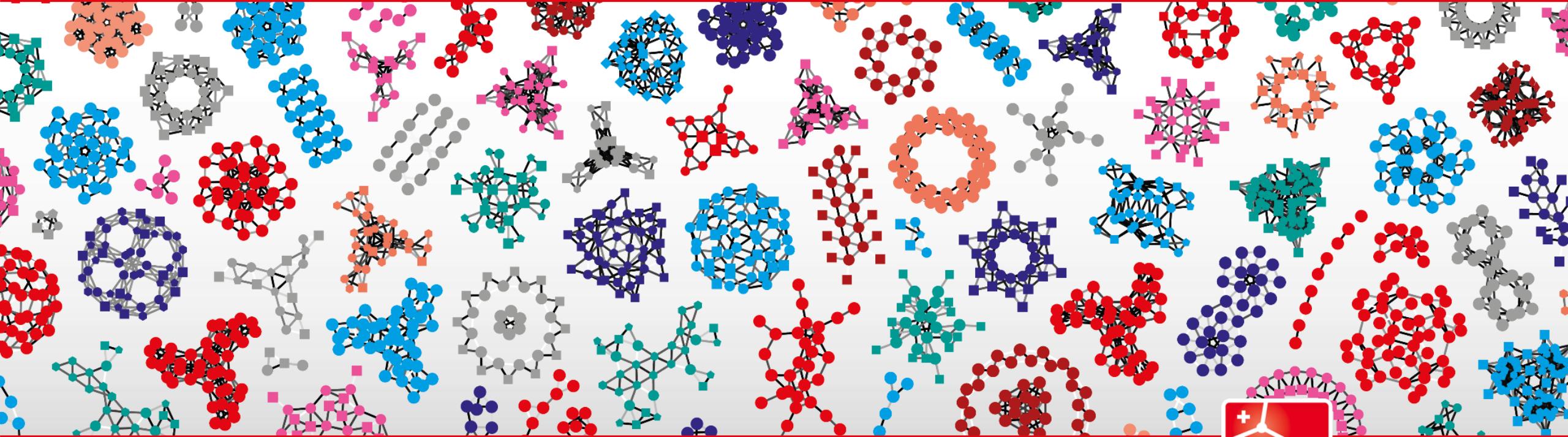


This exercise has helper slides

## Exercise 4 help: branch – rebase – merge sequence







SIB

Swiss Institute of  
Bioinformatics

Thank you for attending this course