

SIB
Swiss Institute of
Bioinformatics

www.sib.swiss

Version control with Git - first steps

Robin Engler
Vassilios Ioannidis
Lausanne, 14 Mar 2022

First steps with Git: course outline

- **Introduction** to Version Control Systems and Git.
- **Git basics:** your first commit.
- **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.
- **GitHub:** an overview.

Course resources

Course home page: slides, exercises, exercise solutions (available at end of day), command summary (cheat sheet), feedback.

Google doc: ask questions.

Questions: feel free to interrupt at anytime to ask questions, or use the Google doc.

Command line vs. graphical interface (GUI)

- This course focuses exclusively on **Git concepts** and **command line** usage.
- Many GUI are available for Git, often integrated with code or text editors (e.g. Rstudio, Visual Studio Code, PyCharm, ...), and 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.

Course slides

- 2 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.

version control

a brief introduction

Why use version control ?

Version control systems (VCS), often 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 allow **collaborative, simultaneous, work** on the repository's content.
- **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.
 - Team and product management tool (e.g. issue tracking, continuous integration).

A (very brief) history of Git

- Created 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).
- **First release in 2005** - in a self-hosting Git repository... of course :-).

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

```
commit e83c5163316f89bfde7d9ab23ca2e25604af29
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...

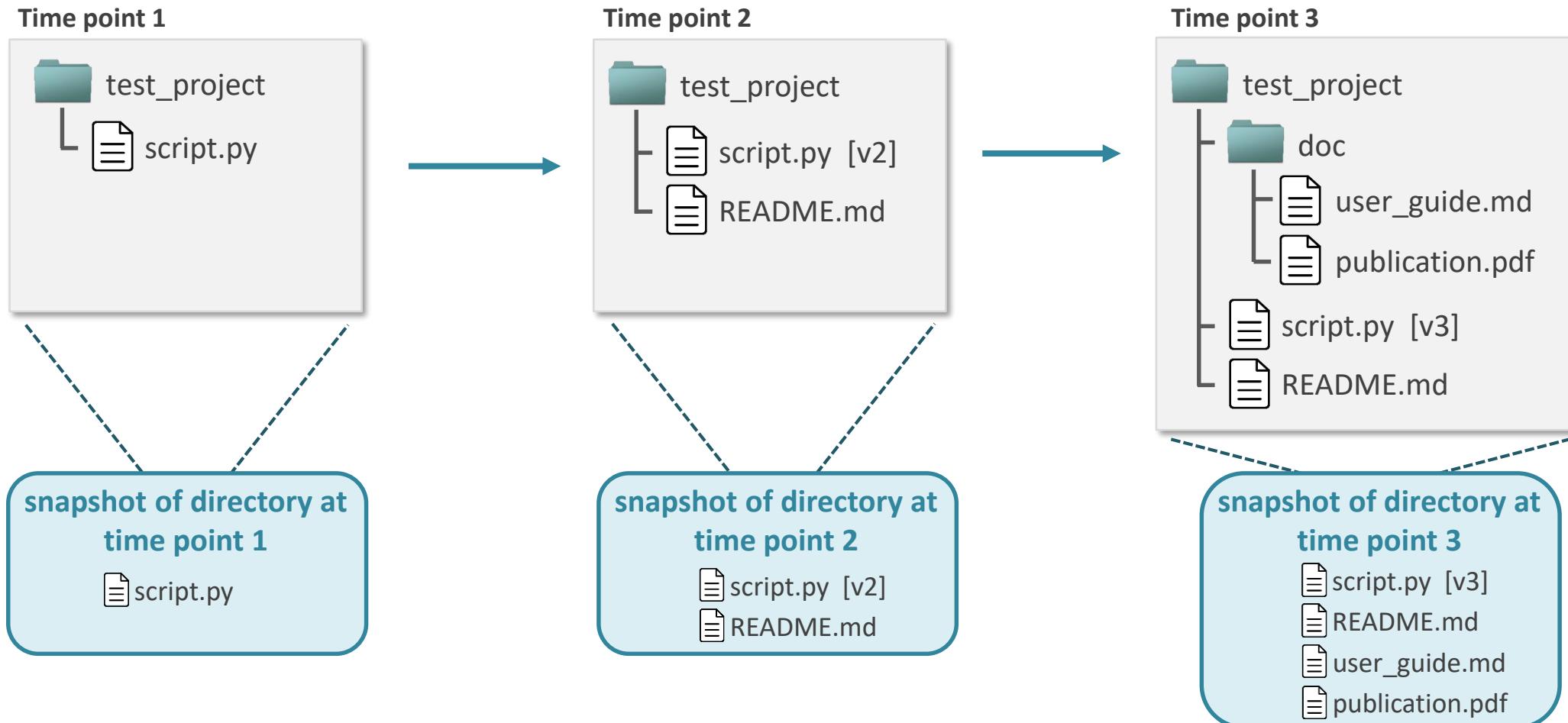
He had the following objectives (among others) in mind:

- **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 "lines of development" (a.k.a. branches) 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.

Basic principle of Git (and VCS in general)

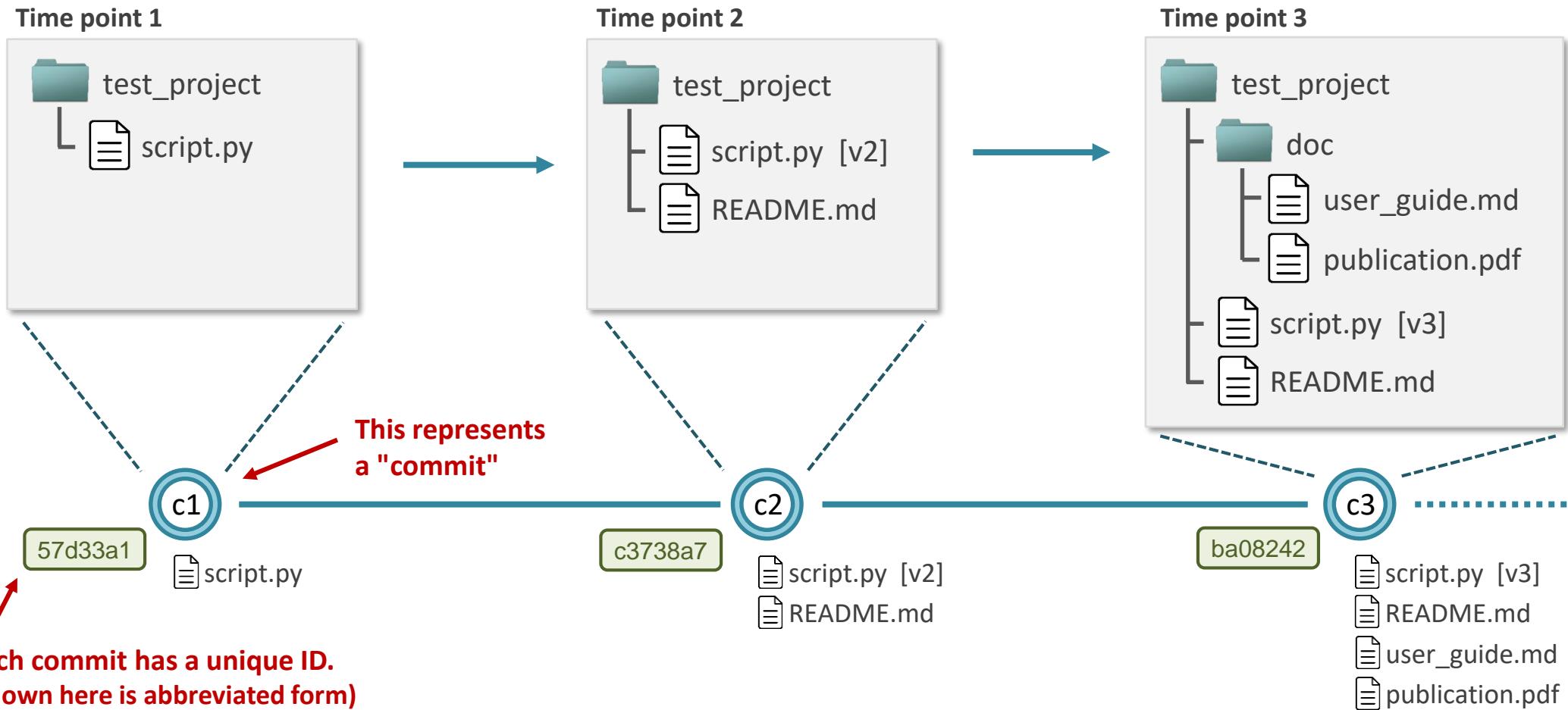
Our objective: version control the content of a directory on our **local** machine. For this we:

- Take **snapshots** (current content of files) **at user defined time points**.
- Keep track of **links between snapshots** so their history can be recreated.



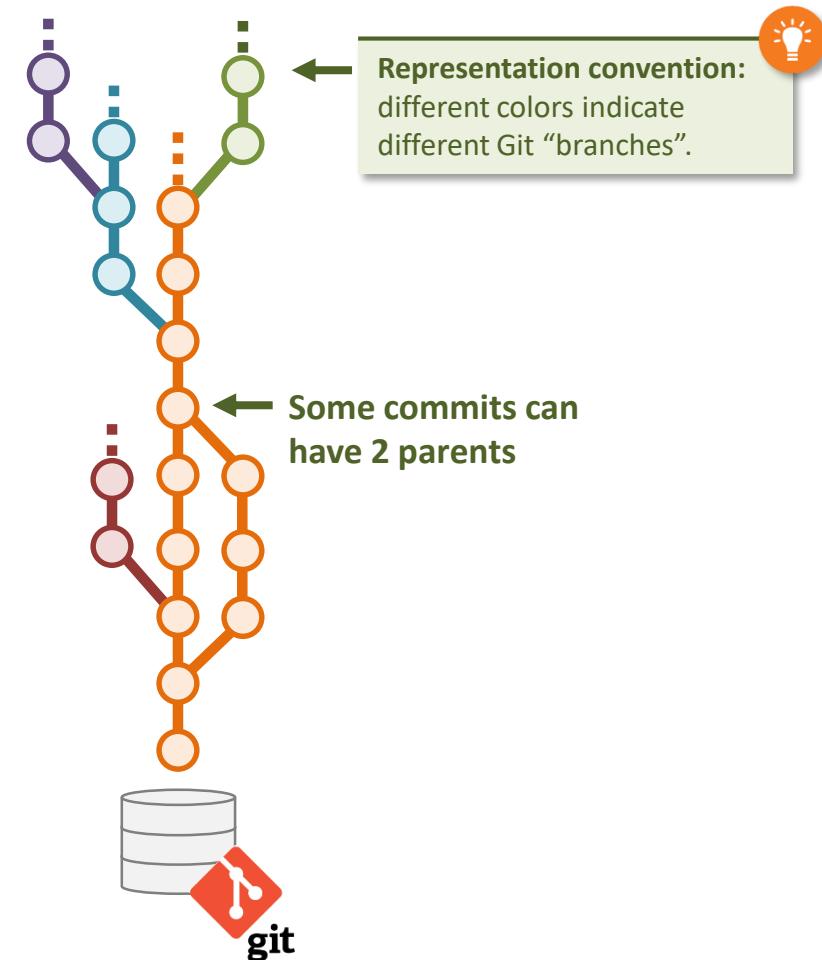
In Git these snapshots are called “commits”

- Commit = snapshot + metadata (author, time, commit message, parent commit ID, etc. ...).
- Create a new commit = record a new state of the directory’s content.
- Each commit has a unique ID number (40 hexadecimal characters): **3c1bb0cd5d67dddc02fae50bf56d3a3a4cbc7204** commit ID



Commits are linked to their parent(s)

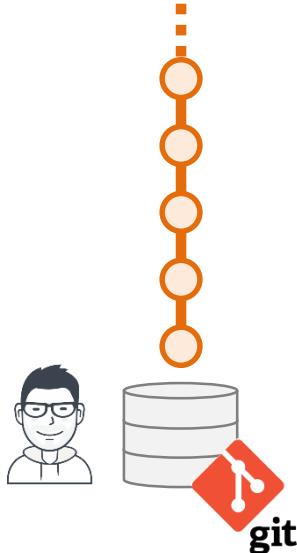
- Git **repository history** = links between commits.
- Multiple “**branches**”(parallel lines of development) can exist within a repo.



Examples of Git use cases

Exercise 1

Local repo, single branch

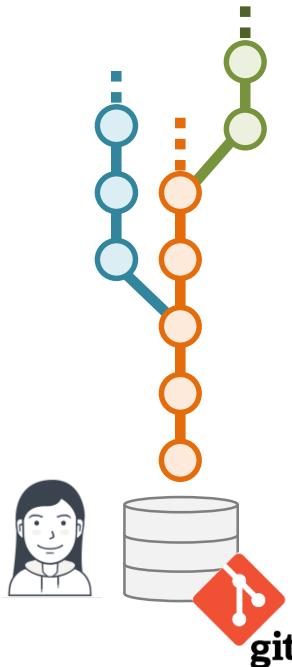


Use case

- Keep a documented log of your work.
- Go back to earlier versions.

Exercises 2 and 3

Local repo, branched workflow
(multiple development lines)

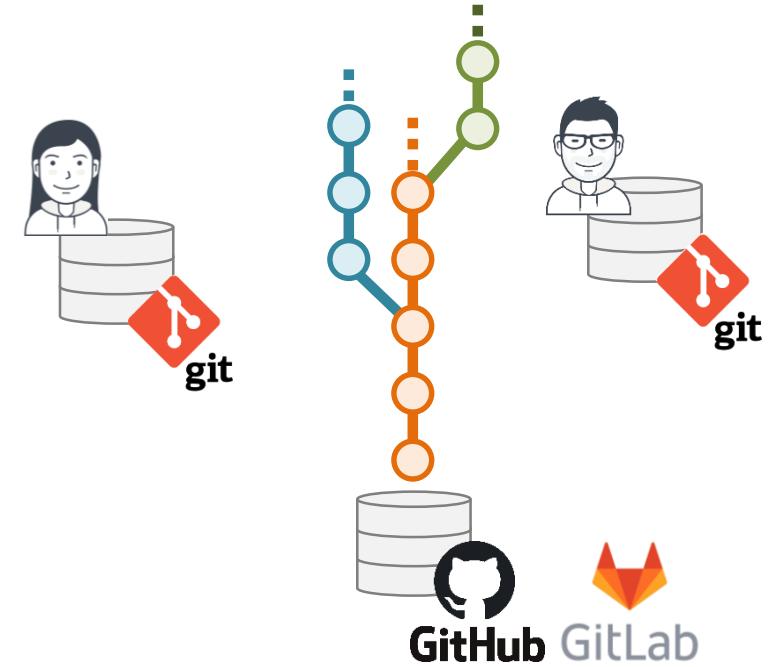


Use case

- Service in production with continuing development in parallel (e.g. new feature).

Exercise 4

Collaboration with distributed and central repos.



Use case

- Collaborate with others.
- Distributed development.



These two cases provide no backup !! only versioning.

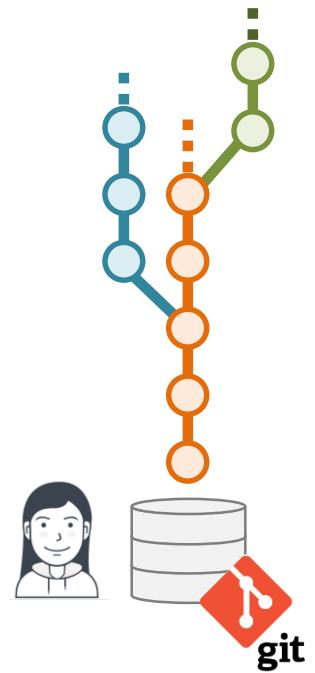
Each user has a full copy of the data*

* 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).
 - Associating 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.

By default, everything stays local.



Git basics

your first commit

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>
```

- Config values can be retrieved by adding the **--get** option.
- Examples:

```
[alice@local ~]$ git config --global user.name "Alice"  
[alice@local ~]$ git config --global user.email alice@redqueen.org  
[alice@local ~]$ git config --global --get user.name  
Alice  
[alice@login1 ~]$ git config --global --get user.email  
alice@redqueen.org
```

- User related settings are stored in:
 - Linux: `/home/$USER/.gitconfig`
 - Windows: `C:/Users/<user name>/ .gitconfig`
 - Mac OS: `/Users/<user name>/ .gitconfig`

Git config: changing the default text editor

- On most systems, the default editor that Git uses is “**vim**”.
However, this can be configured with the following config command:

```
git config --global core.editor <editor name>  
git config --global --get core.editor
```

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

```
[alice@local ~]$ git config --global core.editor nano  
[alice@local ~]$ git config --global --get core.editor  
nano
```

Git config: scopes and 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 as part of the global scope, add the `--global` flag 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
```

Creating a new Git repository

- Typing `git init` in any directory will initialize a Git database in the directory, and thereby turn it into a “Git repository”.
- This creates a hidden `.git` directory - i.e. an empty Git database - at the root of the directory.

```
$ cd /home/alice/test_project
$ git init
Initialized empty Git repository in /home/alice/test_project/.git/
$ ls -a
./ ../ .git/ doc/ src/ README.md
```

The Git database is stored in this “hidden” directory.



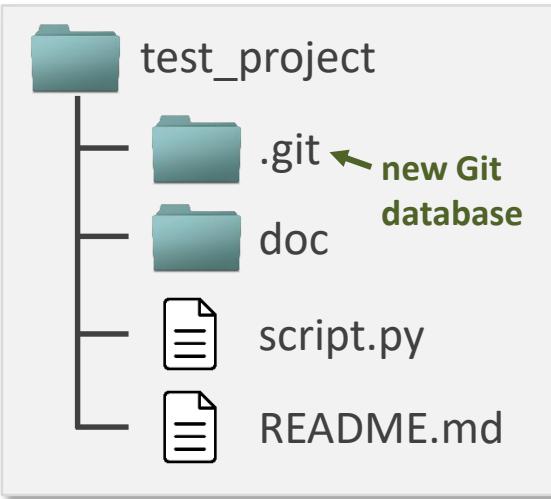
- **Everything** is stored in this single `.git` directory:
 - Content of all tracked files.
 - Complete versioning history.
 - All other data associated to the Git repository (e.g. branches, tags).
- The content of the `.git` database can re-create the exact state of all your files at any versioned time - e.g. if you delete a file accidentally or want to go back to an earlier version.



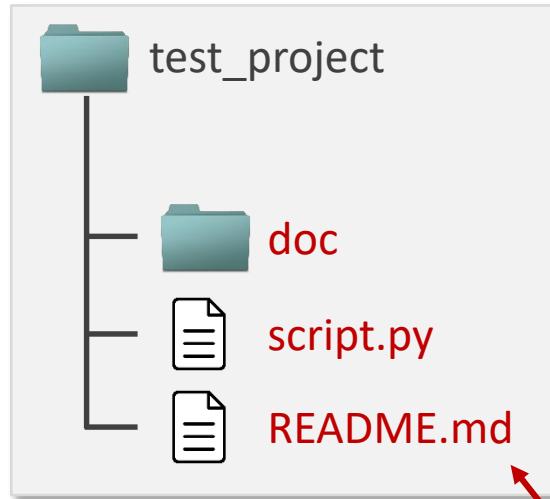
Never delete the ` `.git` directory

State of the working directory just after `git init`

How it look on your filesystem:



How Git sees it:



List of files tracked by Git:

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

`git status`

show status of files in project directory.

`git ls-files`

show files tracked by Git.

`git log`

Show log of commits (i.e. history of repo).

Commit history.

```
$ git log  
fatal: your current branch  
'master' does not have any  
commits yet
```

```
$ git status  
On branch master ← default branch  
No commits yet  
Untracked files:  
  doc/ ← name  
  README.md  
  script.py
```

red = untracked file

Summary: when creating new Git repo...

- It does not matter whether the directory is empty or already contains files/sub-directories.
- Files in your git repo (project directory) are **not automatically tracked** by Git. They must be manually added.
- Only files located in the git repo (or one of its sub-directories) can be tracked.
- You can have both tracked and untracked files in a project directory.
- You can have multiple Git repositories on your system – e.g. one per project or one per code/script you develop.
- Git repos are self-contained – you can rename them or move them around on your file system.
- The ensemble of all files that are under Git control in a given git repository is generally referred to as the repository's **working tree**.



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).

“Bare” repositories

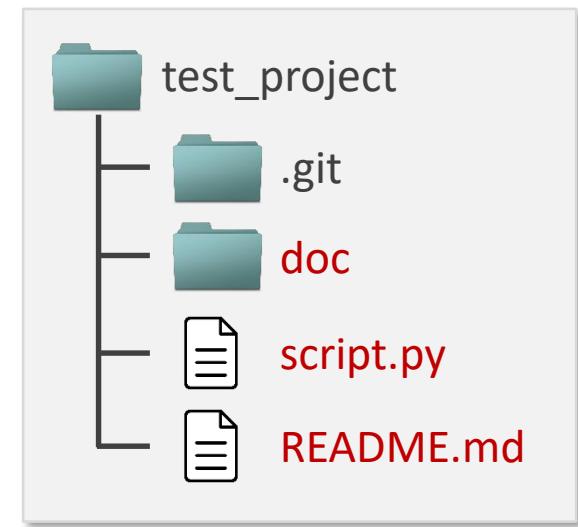


A **bare repo** is a repo that has **no working tree**: it does not contain any instance of the files that are under Git version control, but only the content of the `.`git`` directory/database.

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`

Adding content to a Git repository

- By default, files in the working area are **untracked**.
- To add a new file – or a change in file content – to the Git repository, the file **must be explicitly added** with the `git add` command.
- This allows to **separate important files** of your project - that you want to be tracked by Git - **from unimportant ones** that should not be tracked or shared (e.g. a test file of your own).
- After a file has been added once, it is considered as **tracked** by Git (unless you manually remove it).
- **Each time a file is modified, it must be added again** for the new content to get added to the repo.

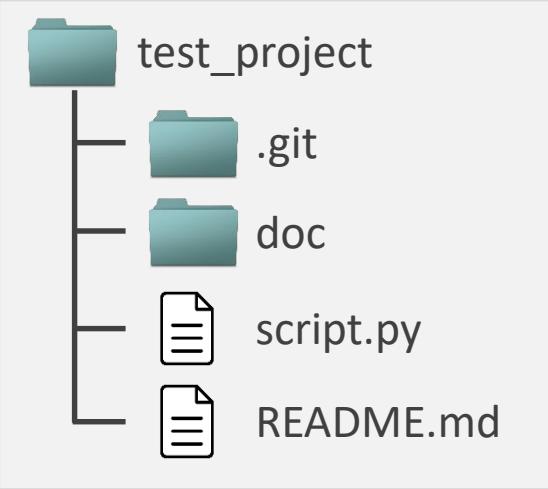


Only files/directories located inside the project's directory can be added.

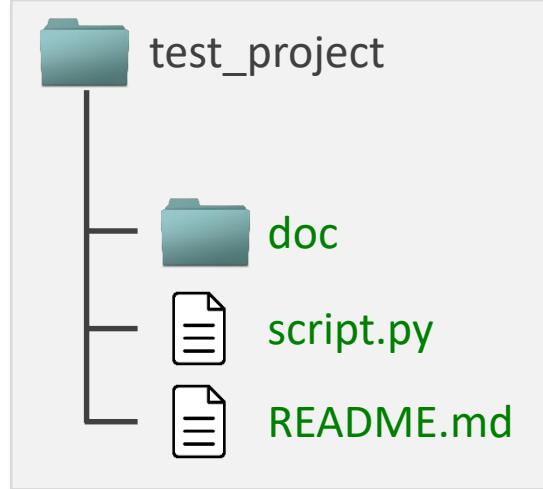
State of the project directory after content is added with `git add`

```
$ git add script.py README.md doc
```

How it look on your filesystem:



How Git sees it:



List of files tracked by Git:

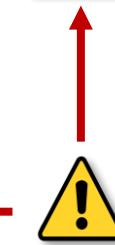
```
$ git ls-files
README.md
script.py
doc/quick_start.md
```

Commit history.

```
$ git log
fatal: your current branch 'master' does not have any commits yet
```

green = tracked and modified file

```
$ git status
On branch master
No commits yet
Changes to be committed:
  new file: README.md
  new file: script.py
  new file: doc/quick_start.md
```



Files/changes are added,
but not committed yet.

Committing content

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

If no commit message is given, Git will open its default editor and ask you to enter it interactively.



Example

```
$ git commit -m "Initial commit for test_project"  
[master (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
```

README.md

```
# Quick-start guide for the test_project software
```

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

+ 1

script.py

```
#!/usr/bin/env python3
```

+ 1

doc/quick_start.md

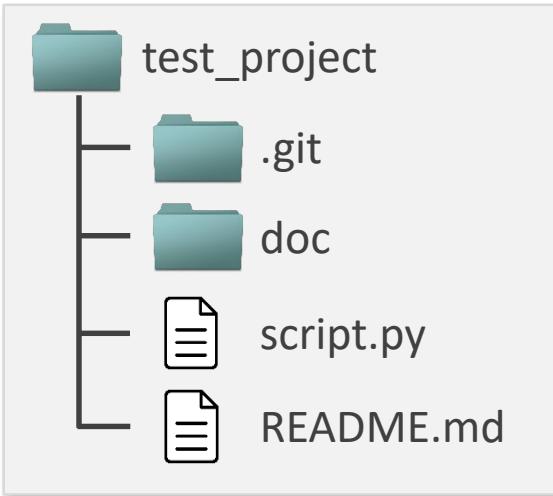
```
# Test project: a project to test version control with git
```

+ 4 (empty lines also count)

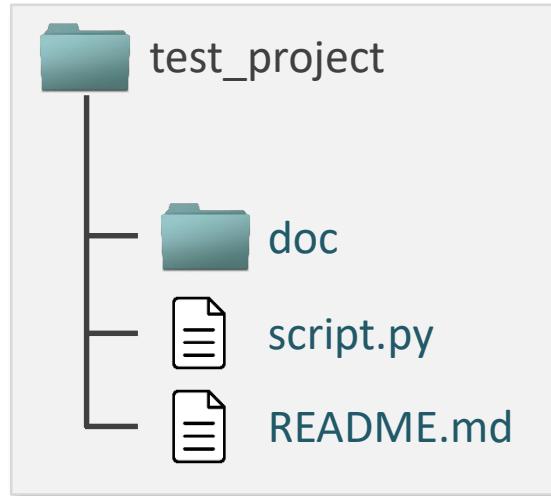
```
This is a small test project to illustrate the use of git.  
Maybe I will add more content to it later.
```

State of the project directory after `git commit`

How it look on your filesystem:



How git sees it:



```
$ git status
On branch master
Nothing to commit, working
tree clean
```

Clean working tree = current state of working tree
matches exactly with the latest commit.

List of files tracked by Git:

```
$ git ls-files
README.md
script.py
doc/quick_start.md
```

Commit history.

```
$ git log
commit 8190787daa6fca93f5f25b819716d50c31bf5c26
Author: Alice <alice@redqueen.org>
Date:   Sun Feb 9 15:07:56 2020 +0100
```

Initial commit for test_project

Now `git log` has finally something
to display (just 1 commit, for now).

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 master
# 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 enter 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 master
# Changes to be committed:
#       modified: README.md
#       new file: script.py
#       new file: doc/quick_start.md
#
~
~
:wq
```

```
[master (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
```

- 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.

Live demo

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

Making commits: some basic advice.

Git does not impose any restrictions on what and when things can be committed.
(the only exception being you cannot commit zero changes)

However, it's best if you:

- Make **commits at meaningful points of your code/script development.**
For instance:
 - a new function/feature was added (or a few related functions)
 - a bug was fixed.
- **Don't commit broken code on your main/master branch** (i.e. the main branch).
You can commit them to a *devel / feature* branch, and later consolidate them before merging with main/master (more on branch management later).

Ignoring files

- By default, files that are not added to the Git repo are considered by Git as "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" lists:

.gitignore

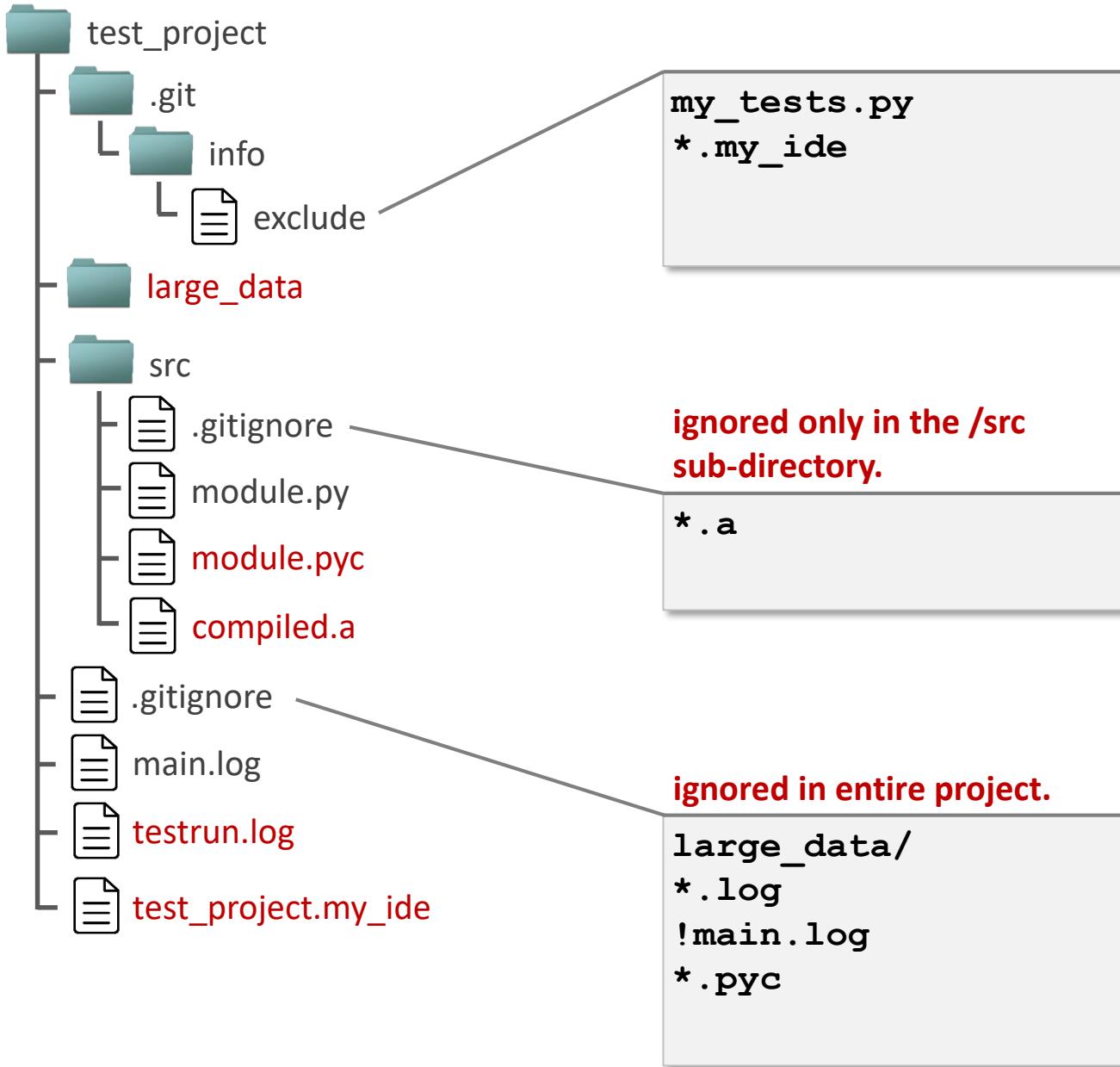
- For files to be **ignored by every copy of the repository**.
- `.gitignore` is meant to be tracked: `git add .gitignore`
- Examples:
 - outputs of tests
 - `.Rhistory`, `.RData`
 - `.pyc`
 - `.o`, `.a`

.git/info/exclude

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

- Files are added by manually editing the two above-mentioned files.
- Files can be ignored based on their full name, or based on glob patterns.
 - `*.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).

Ignoring files: example



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

Live demo

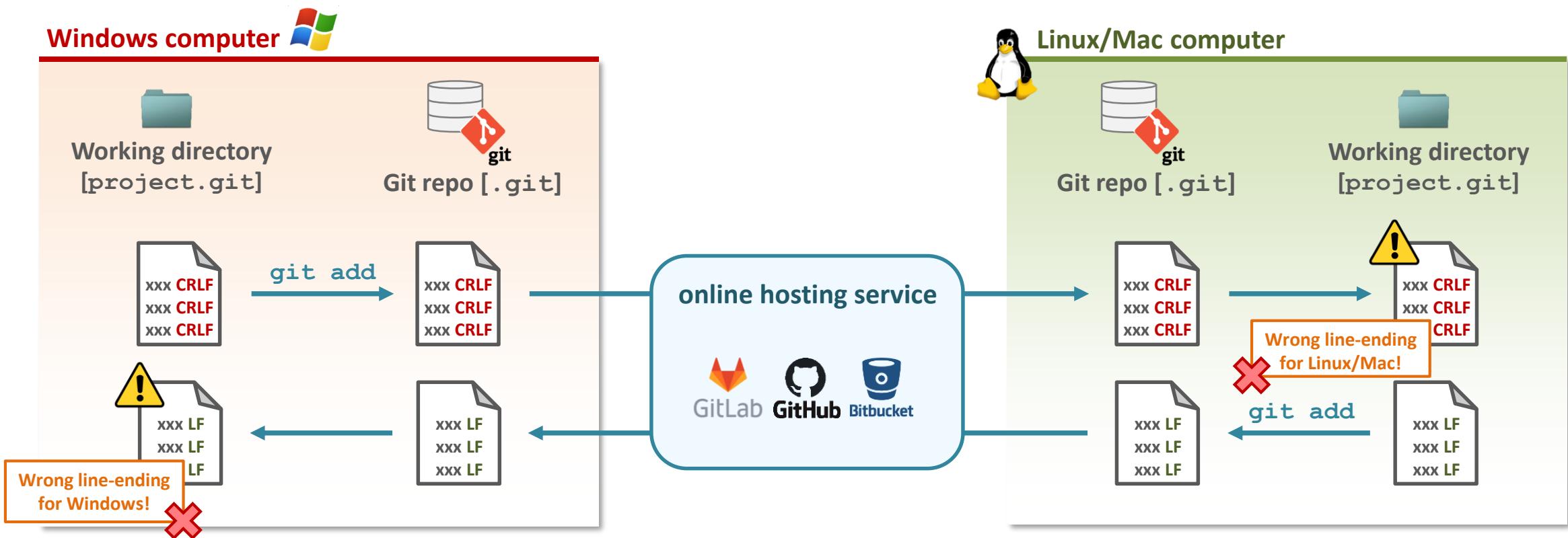
- Adding files to `.gitignore`

Cross-platform collaboration: the line-ending problem

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

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

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



Cross-platform collaboration: solution -> setting git config core.autocrlf

The solution is to ask Git to automatically convert between LF and CRLF during add/checkout operations.

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

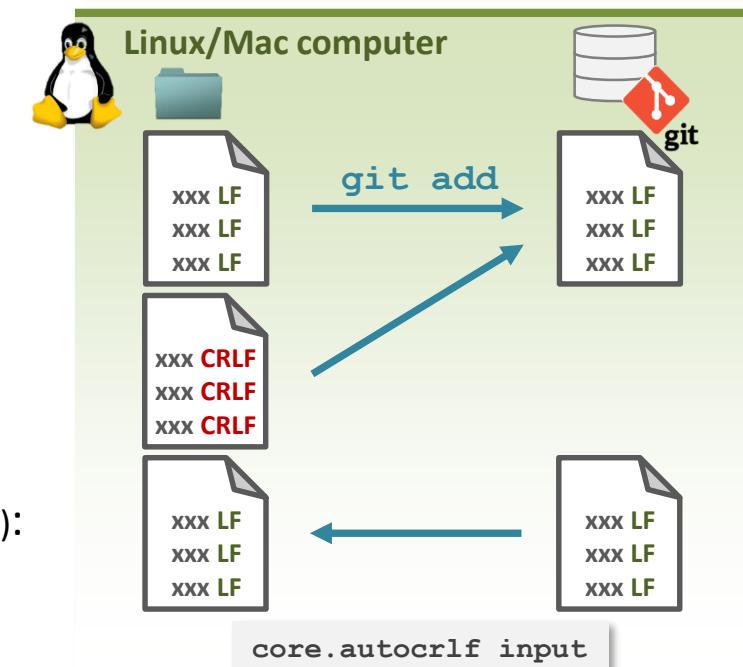
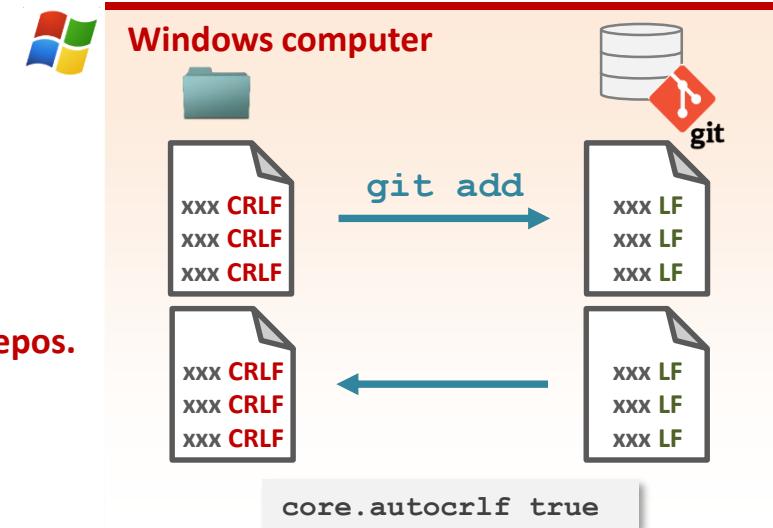
```
git config core.autocrlf true           ← Change setting for current repo.  
git config --global core.autocrlf true ← --global = change setting for all repos.
```

- 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` (so 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 check-in/check-out of files... so when checking-in files (`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”.

Displaying a repository's state and history

git status, **git show** and **git log**

git status

- Display the status of files in the working directory.

git status

Green = New content in this
file will be part of the next
commit.

Red = New content in this
file will not be part of the
next commit.

```
$ git status  
On branch master  
Changes to be committed:  
(use "git reset HEAD <file>..." to unstage)
```

modified: LICENSE.txt

```
Changes not staged for commit:
```

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

modified: README.md

```
Untracked files:
```

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

untracked_file.txt

traced
files.

untraced
files.

File status in Git

There are 4 possible statuses for files in Git:

- **Tracked**

File that is currently under version control by Git (i.e. it is in the Git index).

- **unmodified** - the content of the file is the same as in latest commit.

more precisely: the content is the same as in the commit to which HEAD is currently pointing.

- **modified** - the content of the file differs from the latest commit.

more precisely: it differs from the commit to which HEAD is currently pointing.

- **Untracked**

File that is in the working directory, but not under version control by Git.

- **Ignored**

Untracked file, but where Git is aware it should not be tracked.

git show

- Show the change in file content introduced by a commit.

```
git show <commit>
```

```
git show → with no argument, the latest commit on the current branch is shown
```

Example:

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

    Fix function so it now passes tests

diff --git a/script.sh b/script.sh
index d7bfcdc8..fa99250 100755
--- a/script.sh
+++ b/script.sh
@@ -7,13 +7,28 @@
 
 # Sort peak list by summit elevation, from highest to lowest.
-cat <(< head -n1 $INPUT_FILE ) <(< tail -n+2 $INPUT_FILE | sort -nr -k $COL_NB) > $OUTPUT_FILE
+cat <(< head -n1 ${OUTPUT_FILE}.tmp ) \
+  <(< tail -n+2 ${OUTPUT_FILE}.tmp | sort -nr -k $COL_NB) > $OUTPUT_FILE
+rm "${OUTPUT_FILE}.tmp"
  echo "### Completed peak sorter script. Output file is: $OUTPUT_FILE"
+DAHU_COUNT=$(head -n2 $OUTPUT_FILE | tail -n1 | cut -f5)
+echo "### The number of Dahus on the Alps' highest peak is: $DAHU_COUNT"
```

git log: display the commit history of a Git repo

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

+ loads of other options (see git log --help)

Example: default view (detailed commits of current branch).

```
$ git log  
commit f6ceaac2cc74bd8c152e11b9c12ada725e06c8b9 (HEAD -> master, origin/master)  
Author: Alice alice@redqueen.org  
Date:   Wed Feb 19 14:13:30 2020 +0100  
  
        Add stripe color option to class Cheshire_cat.  
  
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 -> master, origin/master) 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 -> master, origin/master) 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!

But Git developers have you covered, allowing you to set 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
```



Git versioning

- Git stores a complete version 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 extention), that associates filenames to blobs.

exercise 1

Your first commit



This exercise has helper slides

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

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

Git concepts

commits, the **HEAD** pointer and the **git index**

Git commits

Git's immutable, atomic, units of change

Introducing SHA-1

- SHA-1 stands for **Secure Hashing Algorithm 1**.
- Turns any binary input into an (almost*) unique 40 character hexadecimal hash/checksum value.
hexadecimal = base 16 number (0-9 + a-f)

```
e83c5163316f89bfbd7d9ab23ca2e25604af290
```

- Important: for a given input, SHA-1 always computes 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
```

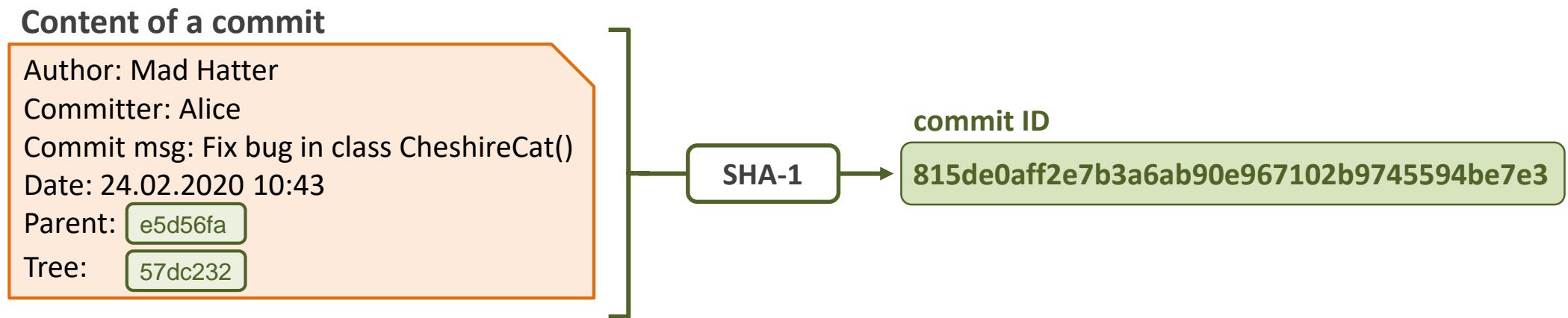


```
3c1bb0cd5d67dddc02fae50bf56d3a3a4cbc7204
```

* as of Jan 2020, SHA-1 collisions can be created for 45'000 USD worth of CPU time.

Commits: Git's atomic, immutable, units of change

- A commit is the **smallest unit of change** in a Git repository.
- A commit is **the only way to enter a change** into a Git repository.
(enforces accountability as you cannot have untraceable modifications)
- Each commit has an associated author, committer, commit message and date.
(enforces documentation)



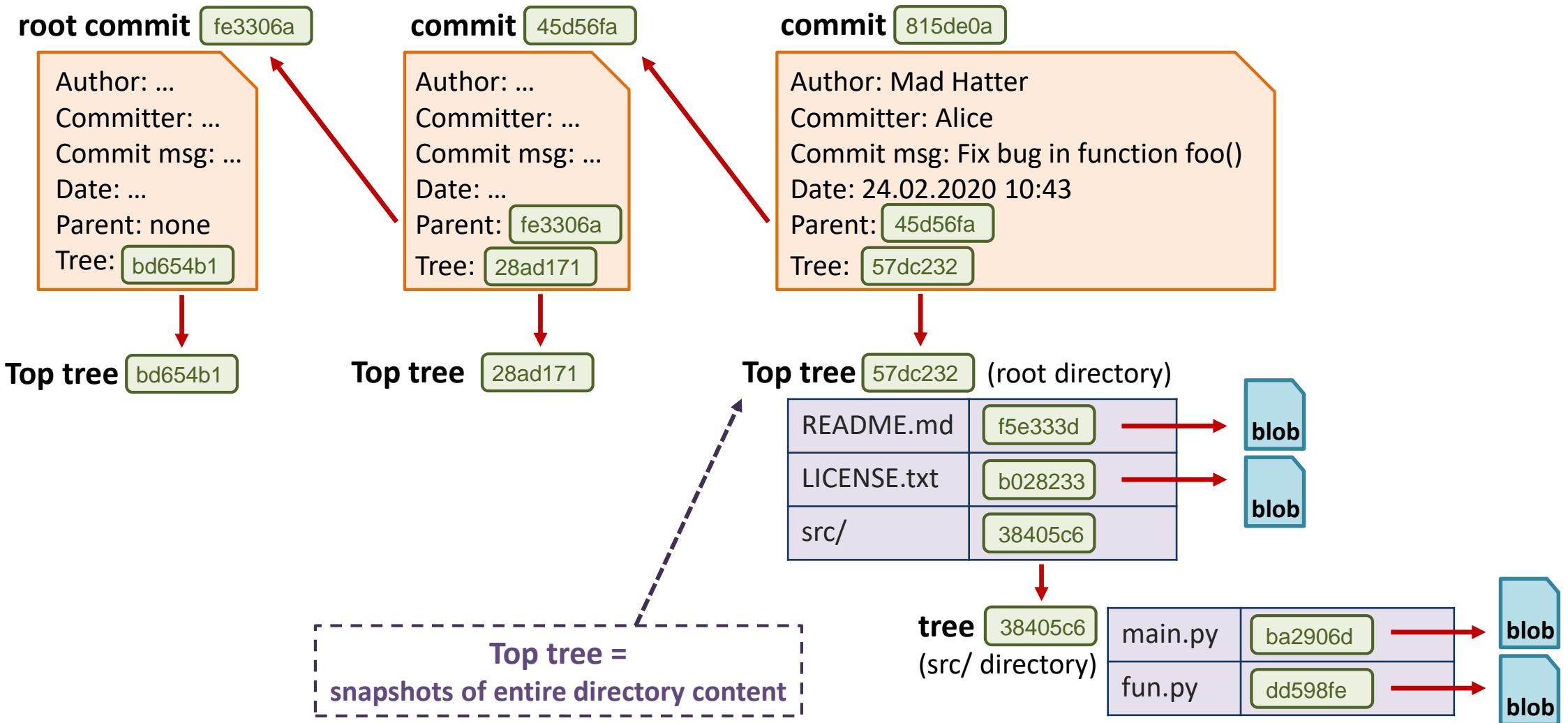
- Commits are lightweight: they do not contain the tracked files' data, only a **reference to the data**.
(a tree object* that represents the content of the Git index at the time the commit was made).
- Commits contain a **reference to their parent** commit.
- Each commit is uniquely identified by a **commit ID**: a SHA-1 hash/checksum computed on its metadata

* Tree = reference to the content of all files at a given time point.

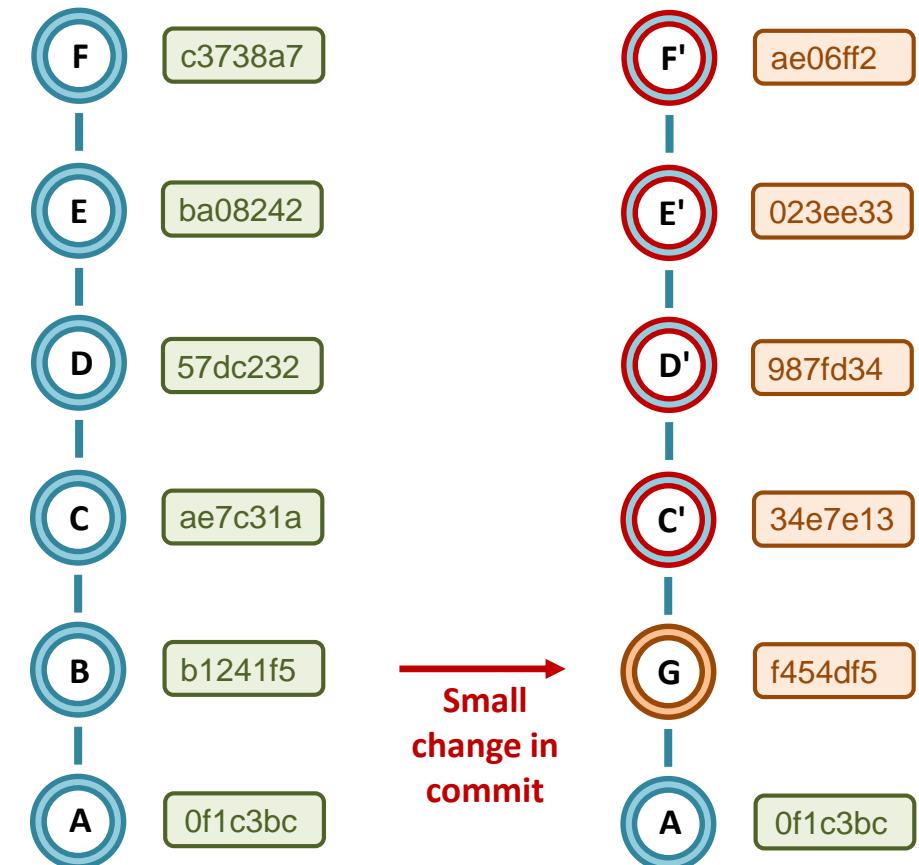
- Commits contain a reference to the top “Tree object”** – a table linking file names and hashes of the Git index at the time the commit was made. This is how Git can retrieve the state of every file at a given commit.
- Commits point to their direct parent** – forming a DAG (directed acyclic graph) 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 !



- Because of how their commit ID (SHA-1 hash) is computed, **commits are immutable**: once a commit is made, it cannot be modified without its commit ID being modified too - which would then makes 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.



the **HEAD** pointer

The tip of your current branch

HEAD: a pointer to the most recent commit on the current branch.

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

```
[rengler@pc-robin exercise_1]$ git log  
commit 30e657bc31de70de260fdcfa3d90f350db69942e (HEAD -> master)  
Author: Robin Engler <robin.engler@sib.swiss>  
Date:   Tue Oct 5 11:22:39 2021 +0200  
  
        Update DESCRIPTION and README
```

Commit ID (SHA1 hash)

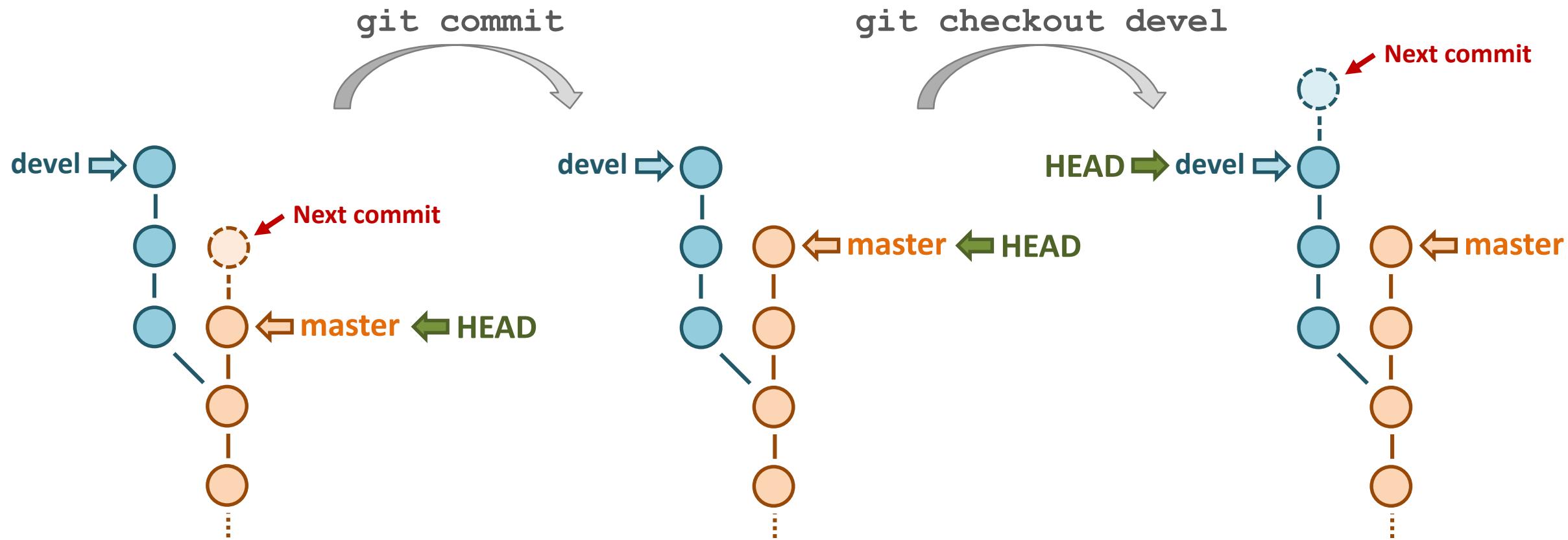
HEAD pointer

branch name

```
[rengler@pc-robin exercise_1]$ git log --all --decorate --oneline --graph  
* 30e657b (HEAD -> master) Update DESCRIPTION and README  
* b6d778e README: add author and URL  
* fd570c5 Add .gitignore file  
* e50b5cc Initial commit for fake stringr package
```

HEAD: a pointer to the most recent commit on the current branch.

- **HEAD** is – most of the time – a pointer to the most recent commit on your current branch
Sometimes it's also described as a pointer to the current branch – which is itself a pointer to the most recent commit.
- When a new commit is added, **HEAD** is automatically moved by Git 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

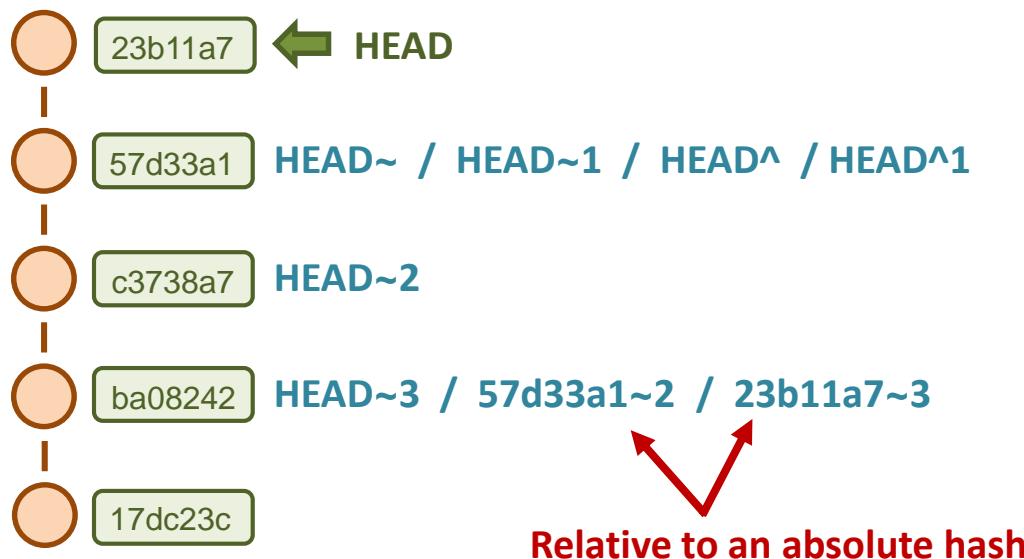
- Using `~` and `^` symbols, Git allows to refer to a commit by its position relative to another commit, rather than by its absolute hash.
- Ref** can be any reference, such as **HEAD**, a commit hash, a branch name, or even another Ref.

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

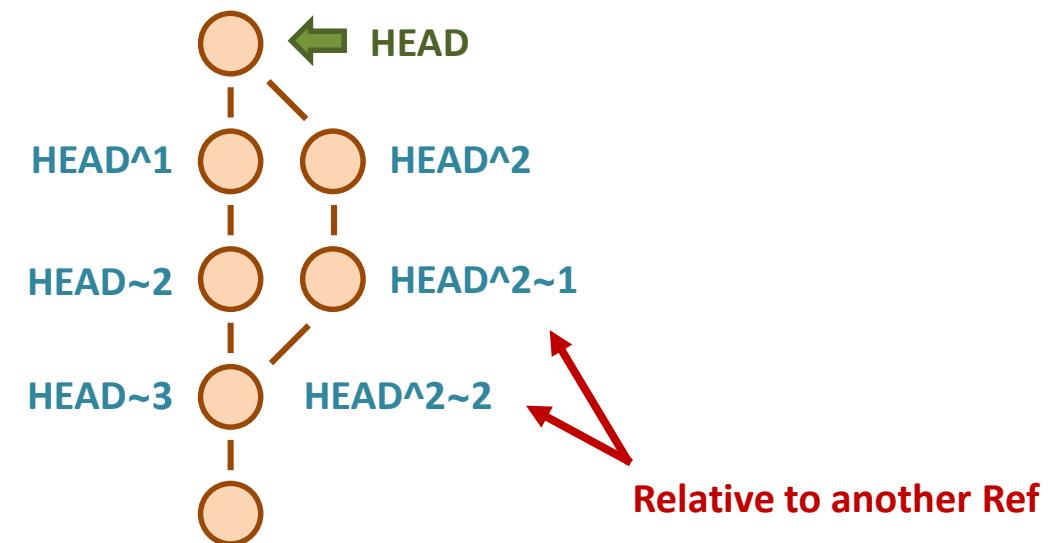
Ref[~] is a shortcut for **Ref^{~1}**

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}**



Relative to an absolute hash



Relative to another Ref

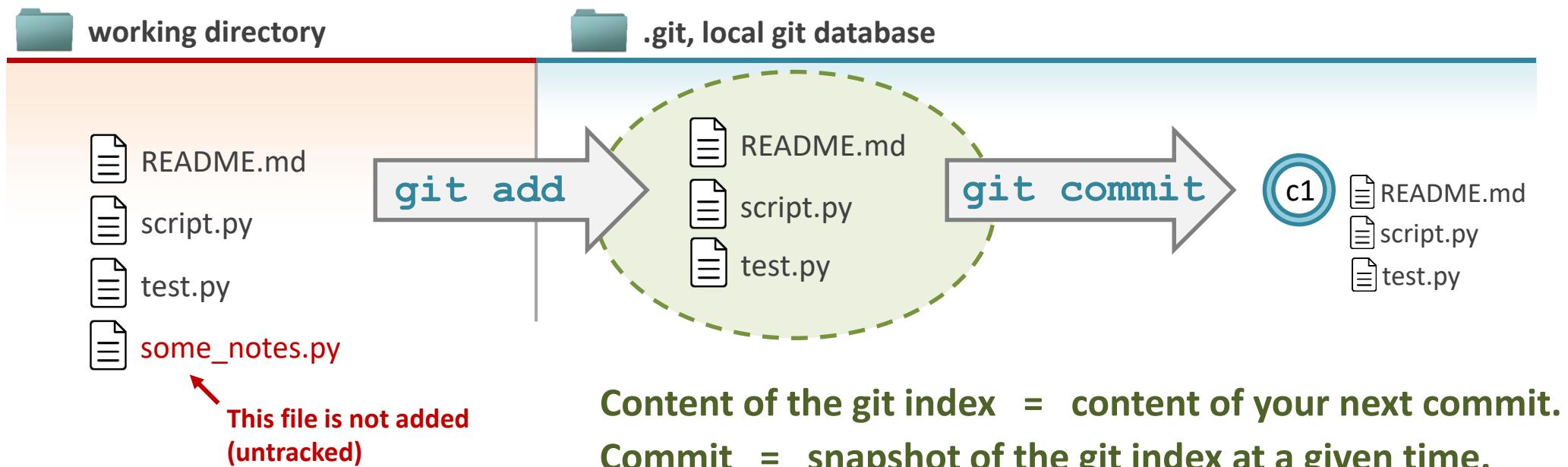
the Git index

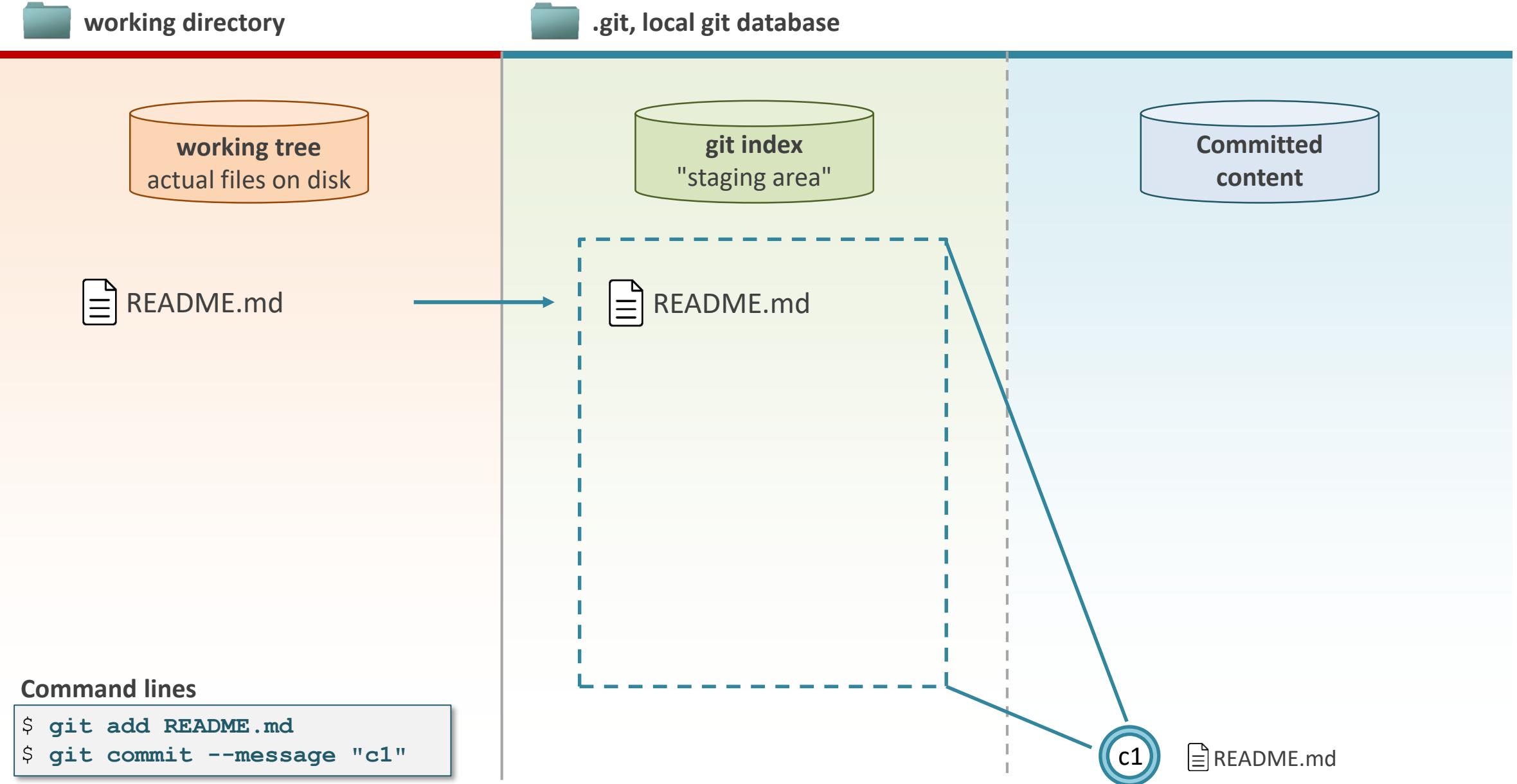
(staging area)

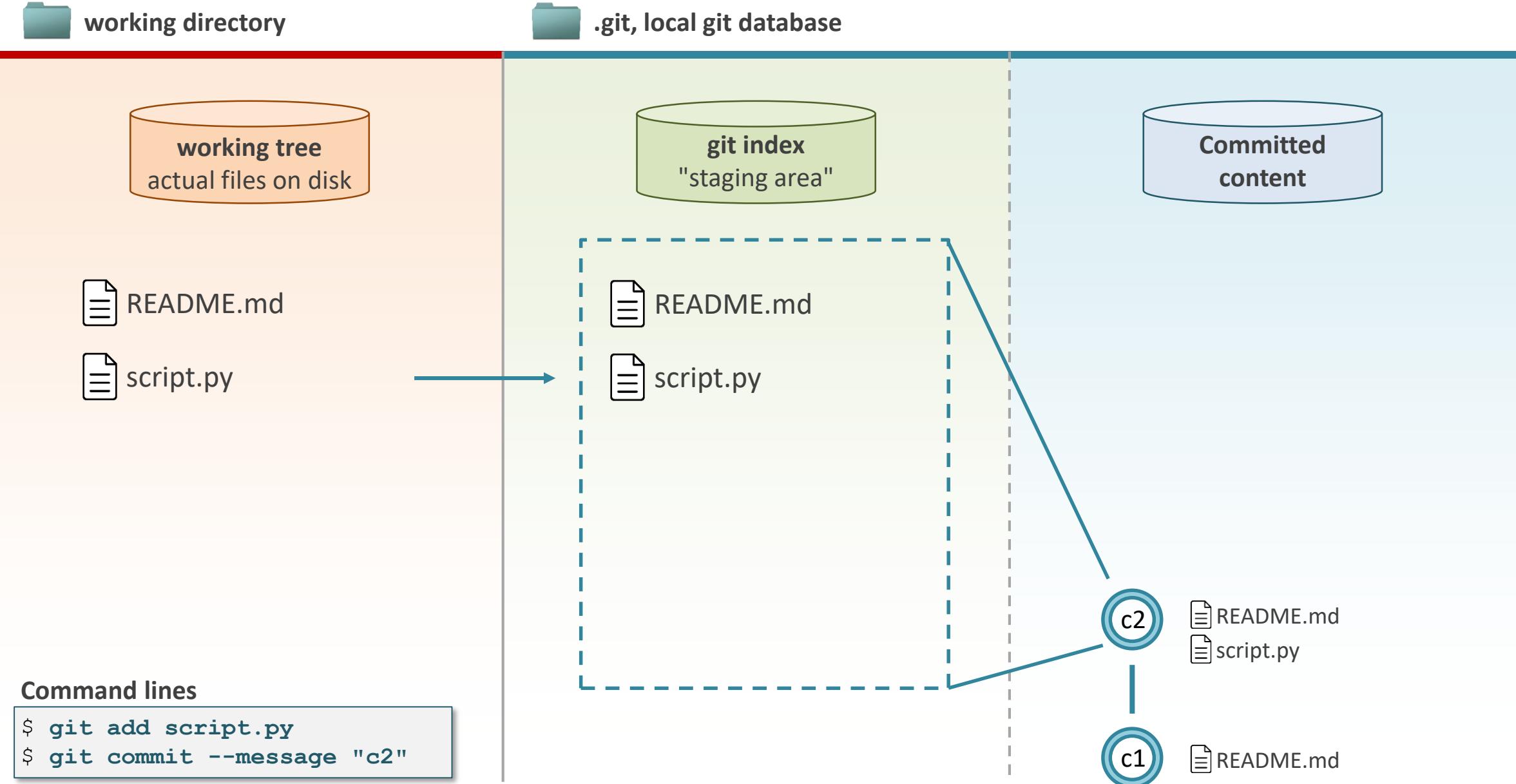
A preview of your next commit

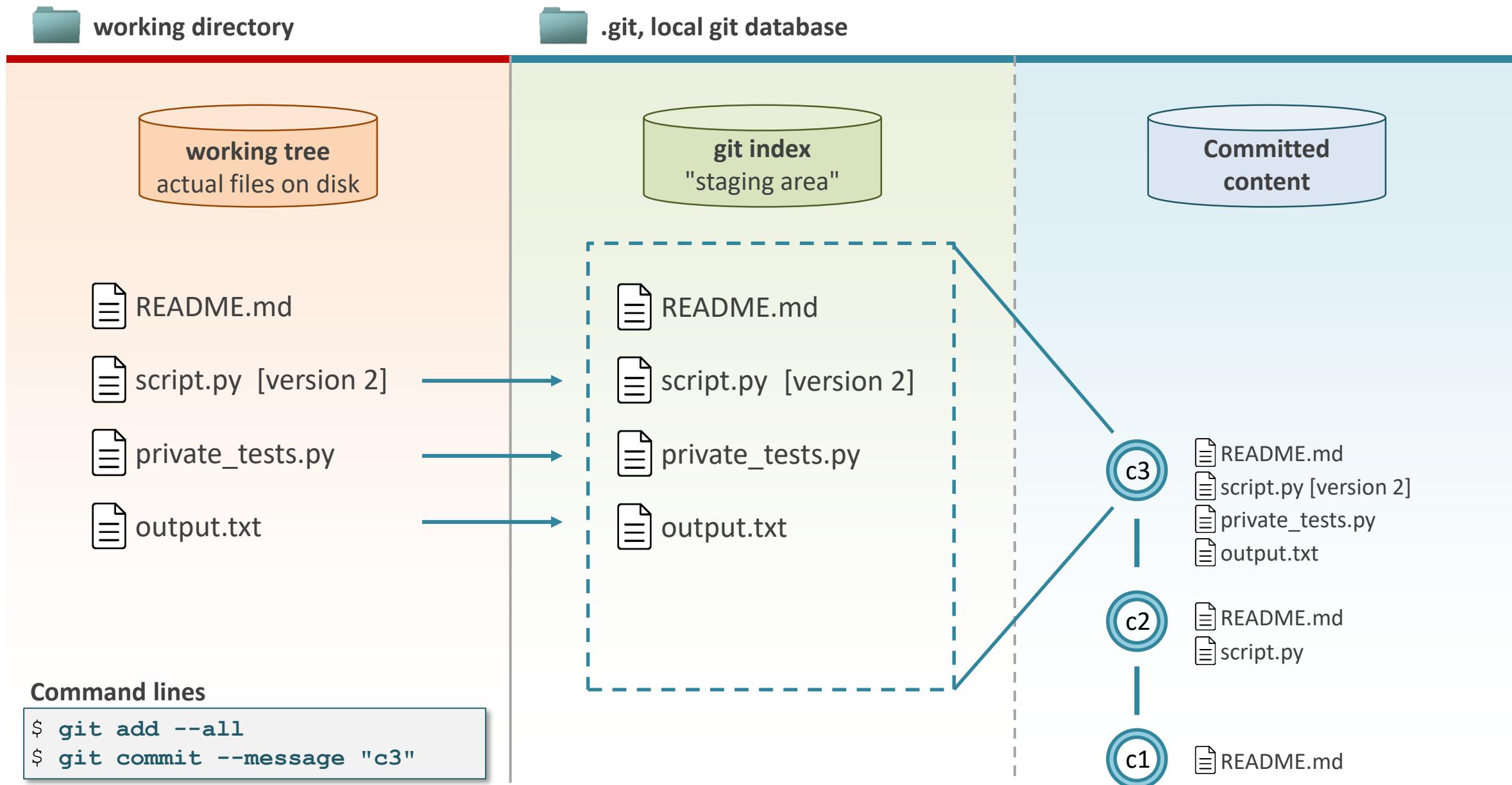
Committing new content: a two-step process...

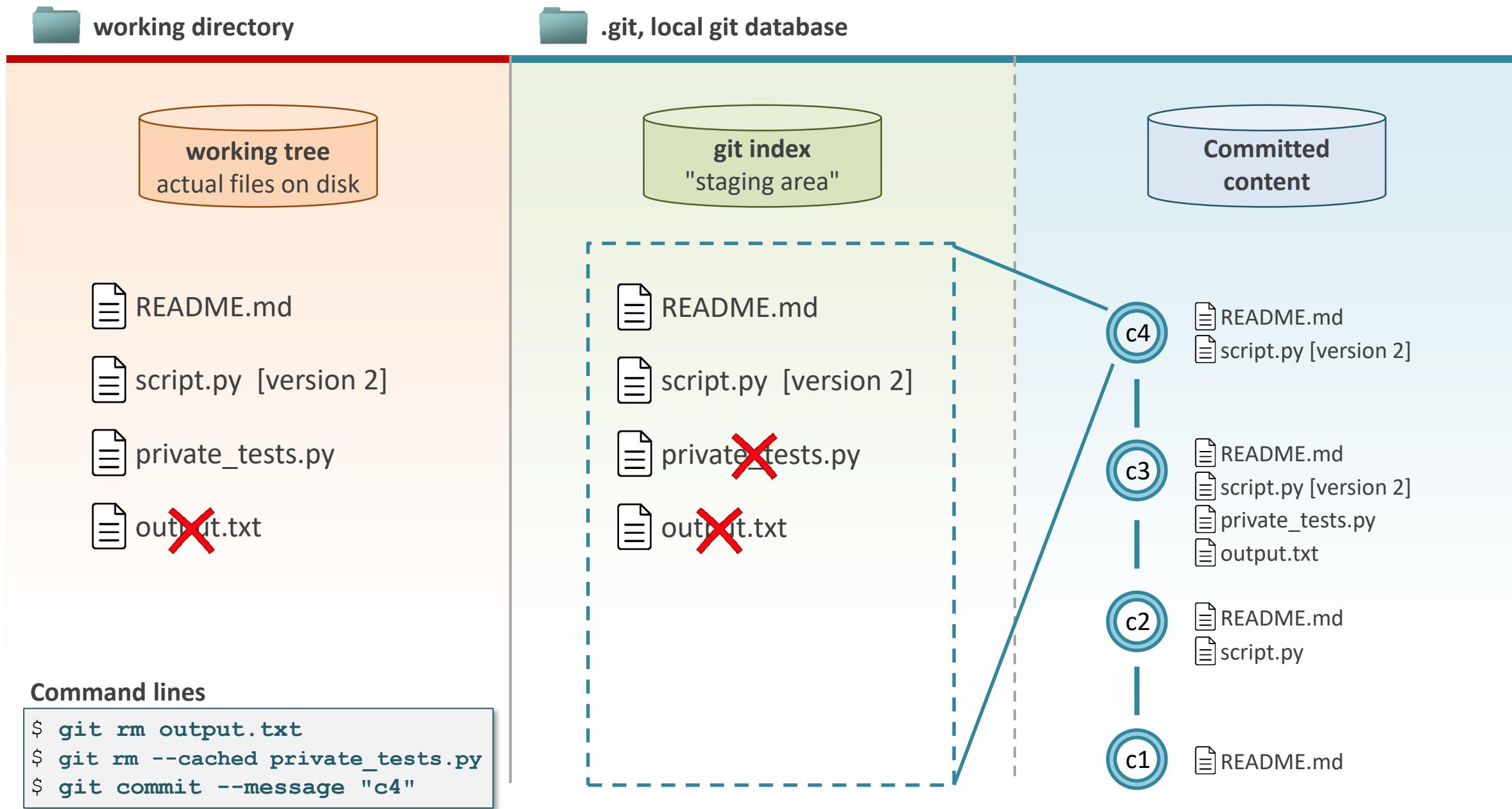
- Any new content to be committed must first be added to the **git index**, or **staging area**. This process is referred to as **staging**.
- **New commit = snapshot of the Git index.** The Git index can be thought of as a sort of “virtual stage” where the content of the next commit is prepared.
- **Staged files remain staged**, unless removed or overwritten by a newer version.
- When files are added to the git index (staged), their content is already copied to the git database.
- The objective of this 2-step procedure is to help create “well defined” commits: not all changes to files in the working directory need to be part of the next commit.

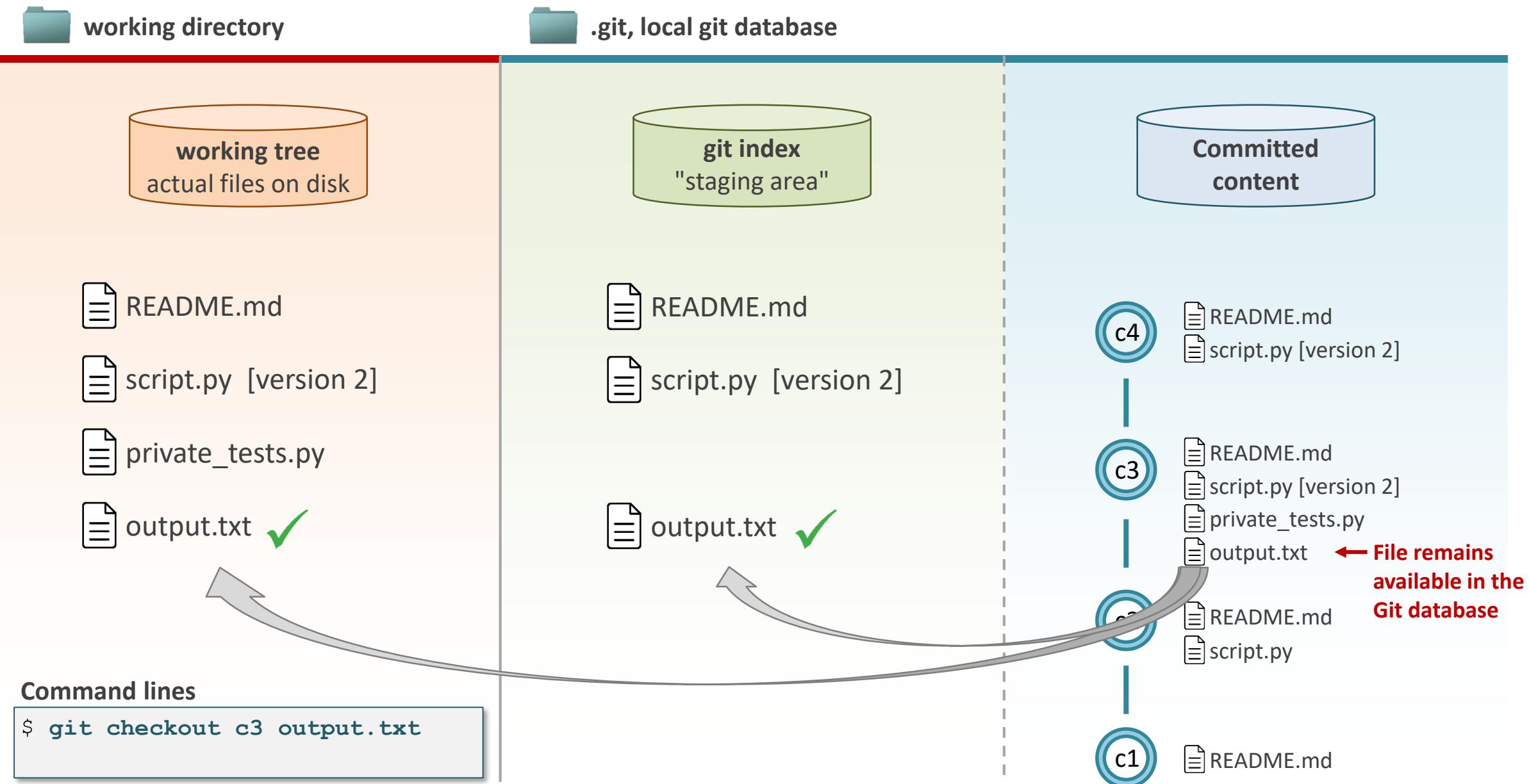












Adding content to the index (staging content)

```
git add <file/directory>      # add the selected file/directory to the git index.
```

- Adds the file content to the Git index (“stages” a file).
 - By default, the entire content of a file is added.
(adding only part of a file is possible with --edit or --patch options)
 - **Each time a file is modified, it must be added again** so that the new version of the file gets added to the git index.
-
- Useful `git add` options

```
git add <file(s) or directory(ies)> # Stages selected files/directories.  
git add -u/--update                # Stages all already tracked files, but ignore untracked files.  
git add -A/--all                   # Stages all files/directories in the working directory (except  
# ignored files). Also stages file deletions.  
git add .                          # Stages entire content of working directory, except file deletions.
```

Removing content from the index

- git restore --staged / git reset HEAD: remove newly staged content from the index.

```
git restore --staged <file> # remove newly staged content of specific file.
```

Note: this is a specific use of the reset command, which have a wider scope.

```
git reset HEAD <file> # remove newly staged content of specific file.  
git reset HEAD # remove all newly staged content.
```

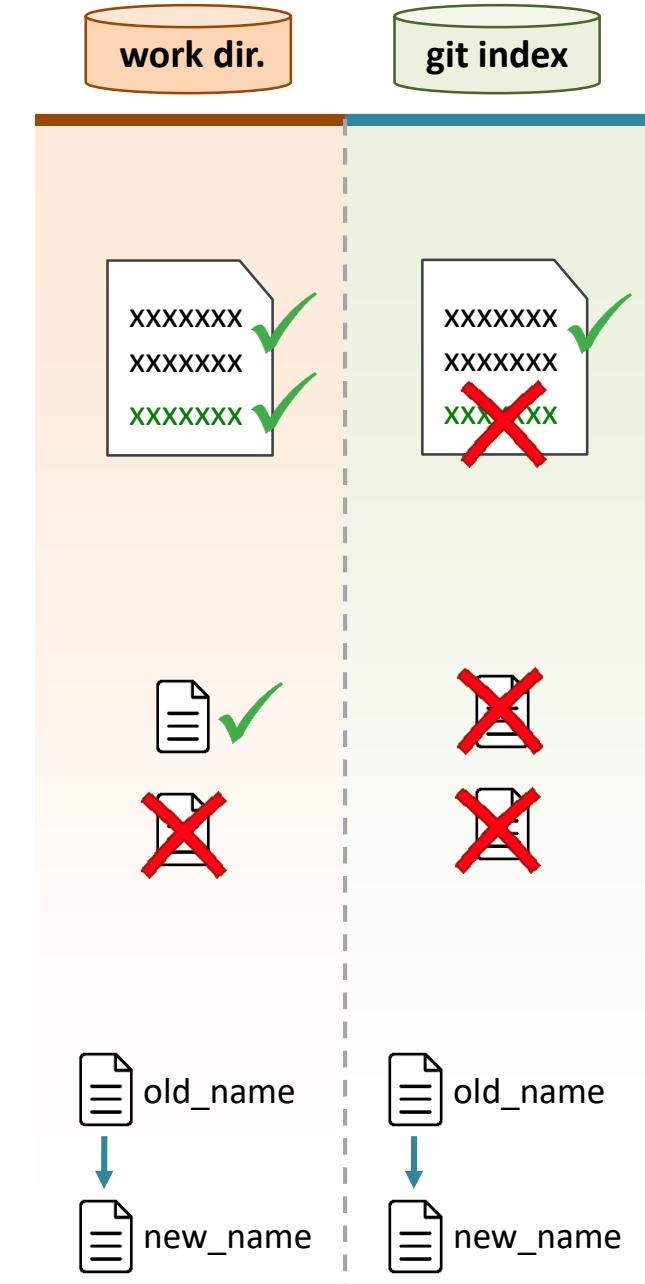
- git rm: remove entire files from the index and the working tree.

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

 ! No --cached option = deletes file on disk !

- git mv: rename and/or move files both in the working tree and the index.

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



How do I know which files are staged? use `git status!`

Green = New content in this
file will be part of the next
commit.

Red = New content in this
file will not be part of the
next commit.

git status

```
$ git status
```

On branch master

Changes to be committed:

(use "git reset HEAD <file>..." to unstage)

modified: LICENSE.txt

Changes not staged for commit:

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

(use "git checkout -- <file>..." to discard changes in working
directory)

modified: README.md

Untracked files:

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

untracked_file.txt

traced
files.

untraced
files.

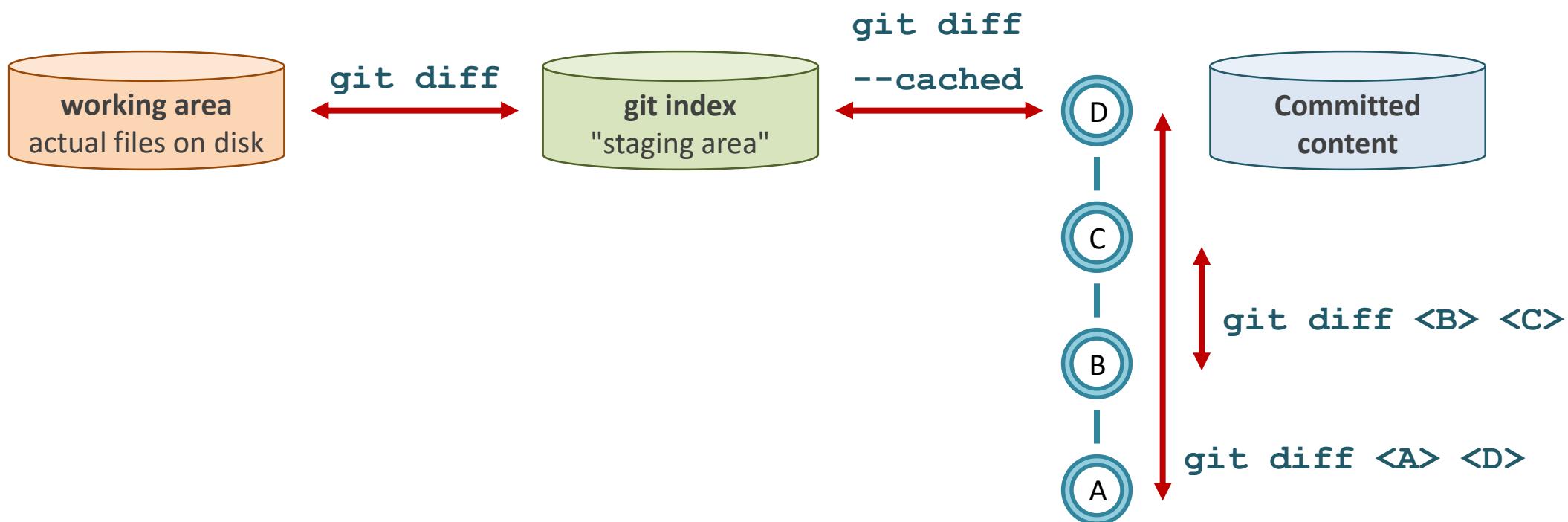
How do I know which changes are staged? use `git diff`!

- Show differences between two states of the git repo.

```
git diff
git diff <file>          # show diff only for a specific file
git diff --cached
git diff <commit 1 (older)> <commit 2 (newer)>
```

Example:

```
$ 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
```



Shortcuts: add + commit in a single command

- Stage + commit all changes in the specified files:

```
git commit -m "log message" <file to commit>
```

This only works for files
that are already tracked.



Example

```
$ git commit -m "README: updates project description" README.md
```

Is the same as:

```
$ git add README.md
$ git commit -m "README: updates project description"
```

- Stage all changes in *tracked* files and commit them.

This will not commit untracked files (unlike `git add --all` that also adds untracked files).

```
git commit --all --message "log message"
```

```
git commit -am "log message"
```

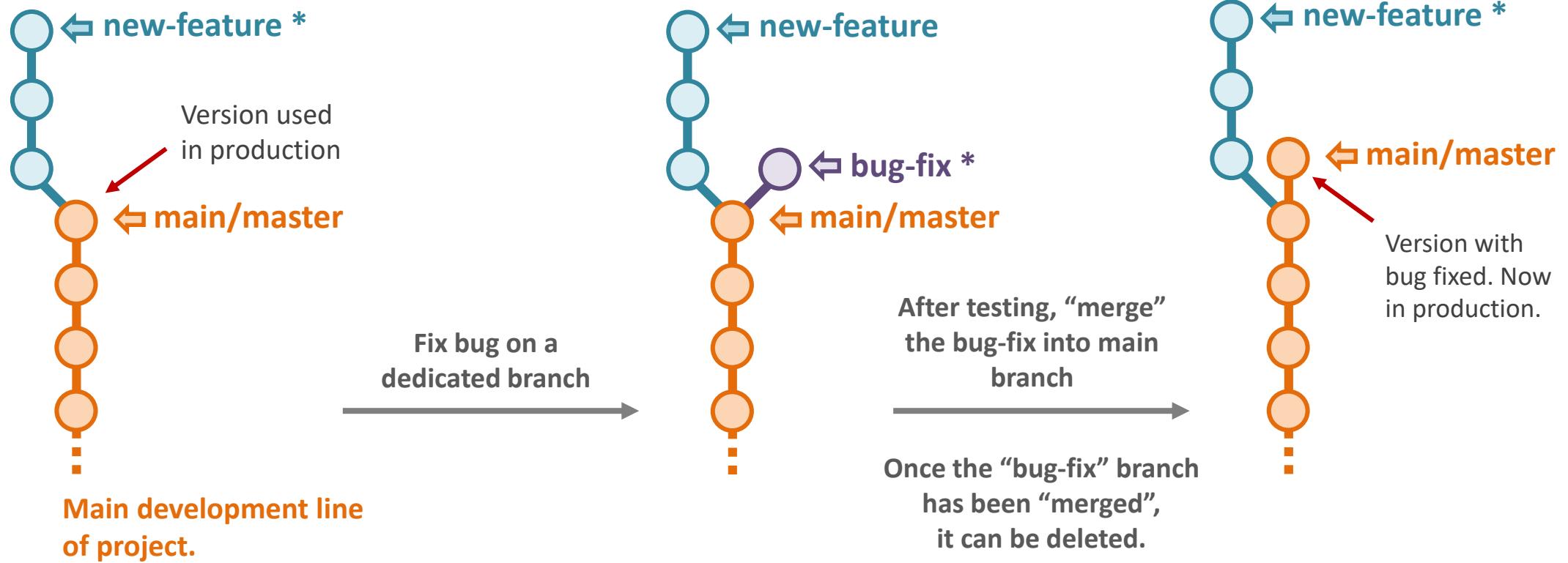
short options version.

Git branches

the killer feature

Why branches?

- Branches are **a great way to isolate new changes** from the main line of development.
- Example: adding a new feature to an application used in production.



What are branches?

- A branch is just a pointer.
- A branch is **very lightweight** (41 bytes).
- Usually the **master branch** is the main branch representing your work.



The master branch

The master branch **is no special branch**. It is just commonly created by `git init`.



Master gets main

Since October 2020 any new repository created on GitHub uses **main** instead of **master** as its default branch. This is done to remove unnecessary references to slavery.

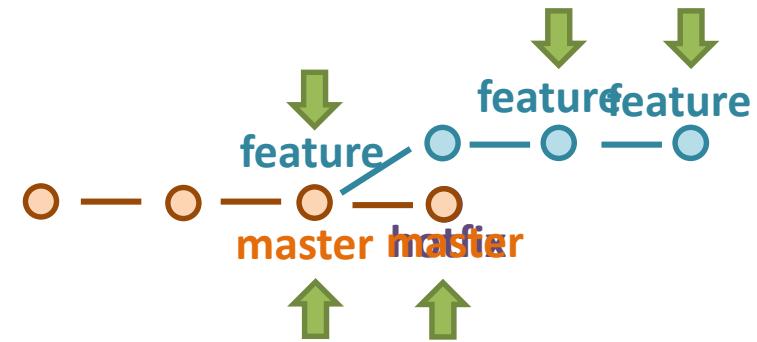


Illegal characters within branch names

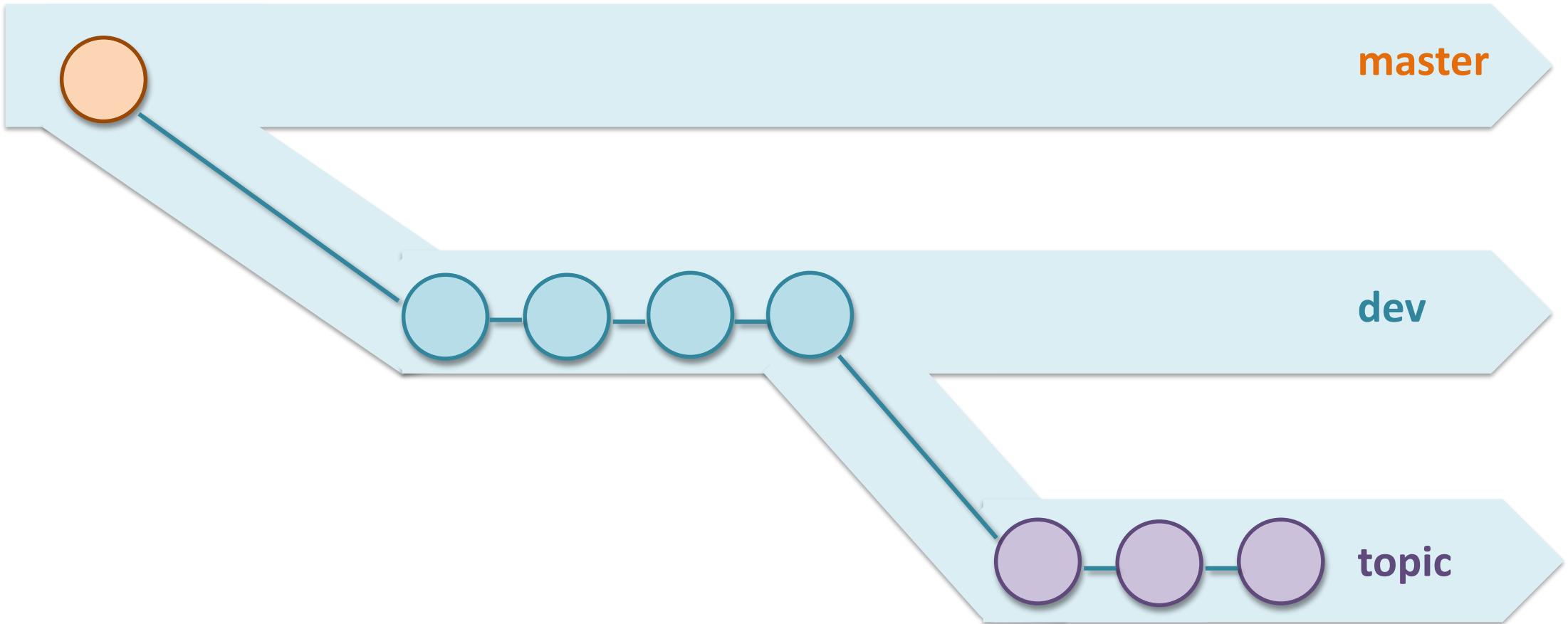
Spaces and some characters such as `, ~^:?:*[]\` aren't allowed for branch names, others must be escaped. Generally it is recommended to stay with lowercase letters, numbers and `-`.

A typical branch workflow

1. Do work on a website.
2. Create a branch for a new story you're working on.
3. Do some work on that branch.
4. Your boss wants a hotfix! Switch to your master branch.
5. Create a branch to add a hotfix.
6. After testing add hotfix to the master branch, push and delete the hotfix branch.
7. Switch back to your new story and continue working on it.

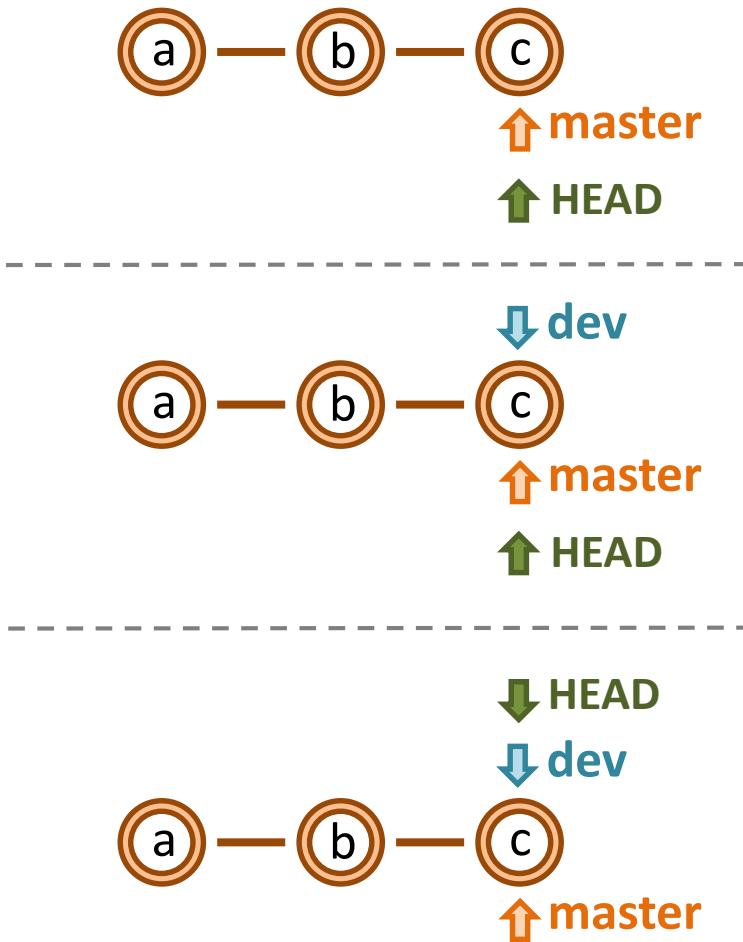


Branch management: best practices



Don't change history on the master branch.

Create a local branch



```
git checkout -b <branchname>
```

```
git switch -c <branchname>
```

git branch dev

git checkout -b dev

git checkout dev
or
git switch dev

List your branches
(* indicates where your HEAD is):

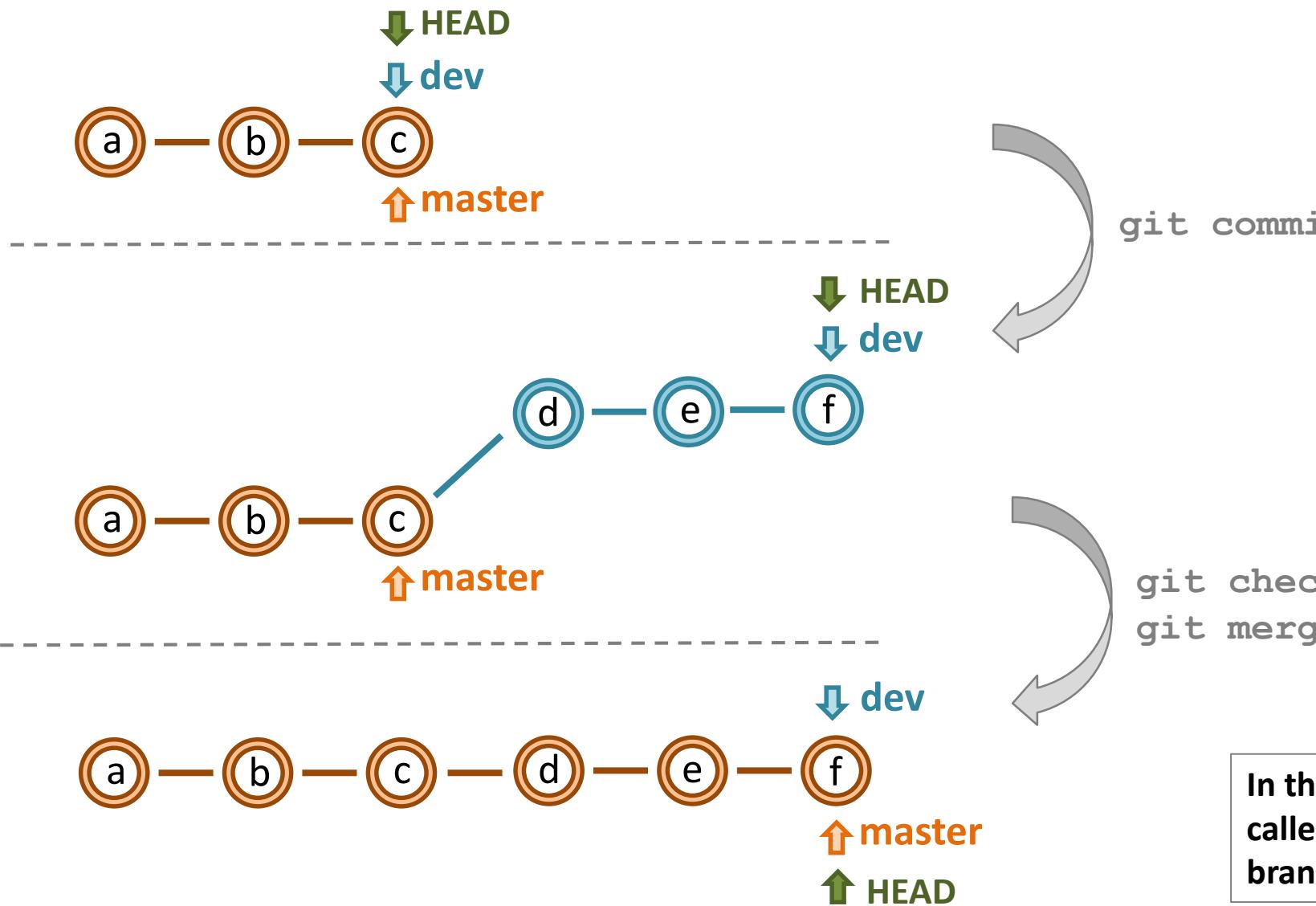
```
$ git branch  
master  
* dev
```



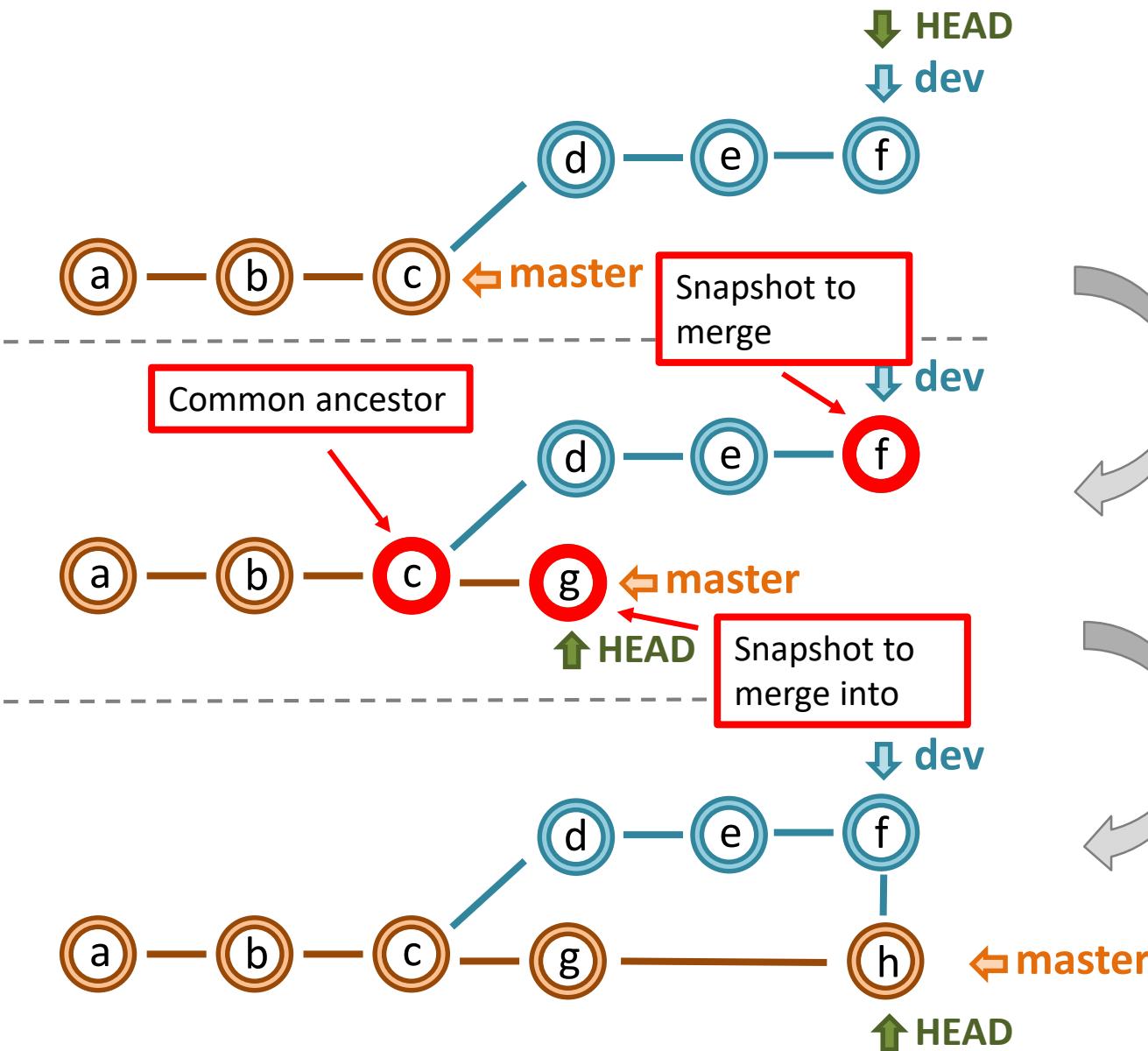
The **checkout** command is a bit confusing, since in another context it is used to reset files to a certain version. This is why **switch** was introduced (Git 2.23) and can alternatively be used to checkout a branch. But it is currently still marked as experimental and its behavior might change.

git merge
get back together

Make commits on the dev branch and merge them into master



Merge branches that diverged



`git checkout master`
`git commit`

`git merge dev`

Branches diverged, so git cannot make a “fast forward” anymore. It will perform a so called “3-way merge”. There might occur conflicts that have to be resolved manually.

Resolving conflicts

- If the same file got changed in both branches, git might not be able to do an automatic merge. It will tell you that there is a conflict.

```
$ git merge dev
Auto-merging index.html
CONFLICT (content): Merge conflict in index.html
Automatic merge failed; fix conflicts and then commit the result.
```

- git status** will tell you what you have to do:

```
$ git status
On branch master
You have unmerged paths.
  (fix conflicts and run "git commit")

Unmerged paths:
  (use "git add <file>..." to mark resolution)

    both modified:      index.html

no changes added to commit (use "git add" and/or "git commit -a")
```

Resolving conflicts

- Open the conflicting files in your file editor of choice.
- Look for text in between <<<<< and >>>>>. The text between <<<<< and ===== corresponds to the version you currently in. In between ===== and >>>>> you have the version from the file you want to merge in.

```
<<<<< HEAD:index.html
<div id="footer">contact : email.support@github.com</div>
=====
<div id="footer">
  please contact us at support@github.com
</div>
>>>>> dev:index.html
```

- You have to **manually edit** the conflicting file and choose the parts you want. Make sure to remove all lines containing <<<<<, ===== and >>>>>.

```
<div id="footer">
  please contact us at email.support@github.com
</div>
```

Resolving conflicts continued

- Now you can `git add` resolved files.

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

Changes to be committed:

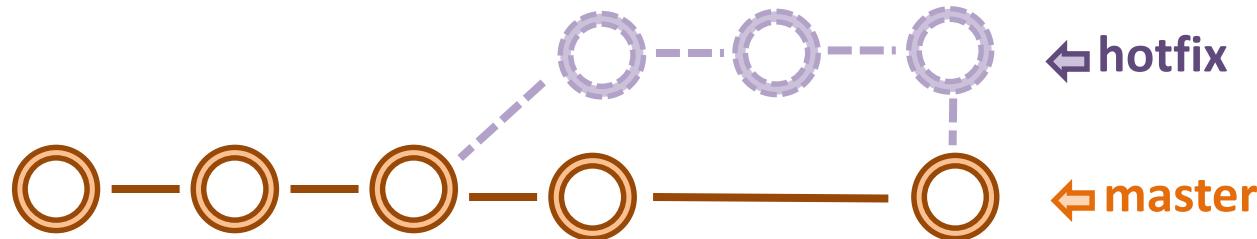
modified:   index.html
```

- The fixed merged can now be committed.

```
$ git commit
```

Deleting branches

- Branches that are merged and aren't used anymore should be deleted.



```
$ git branch -d hotfix
```

- Deleting a branch with `git branch -d` is a safe operation. Git only let you delete branches that are fully merged.

```
$ git branch -d testing
error: The branch 'testing' is not fully merged.
If you are sure you want to delete it, run 'git branch -D testing'.
```

- To force the deletion of a branch you can use `-D` instead.

```
$ git branch -D testing
```

exercise 2

The Git reference webpage

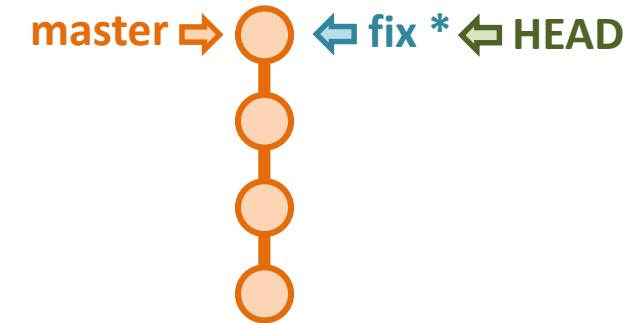


This exercise has helper slides

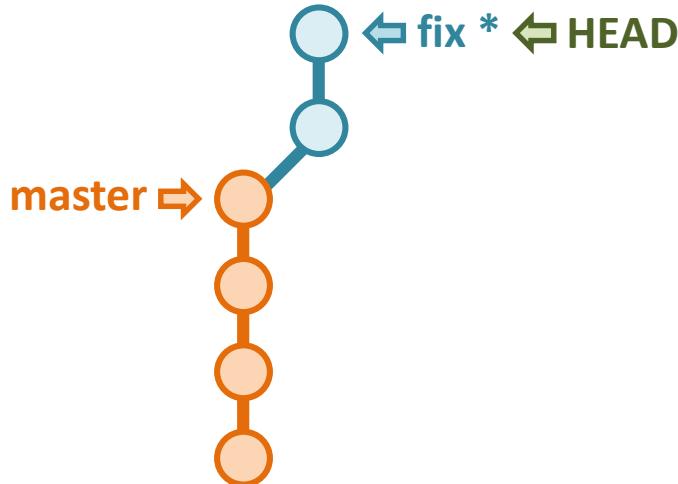
Exercise 2 help: workflow example



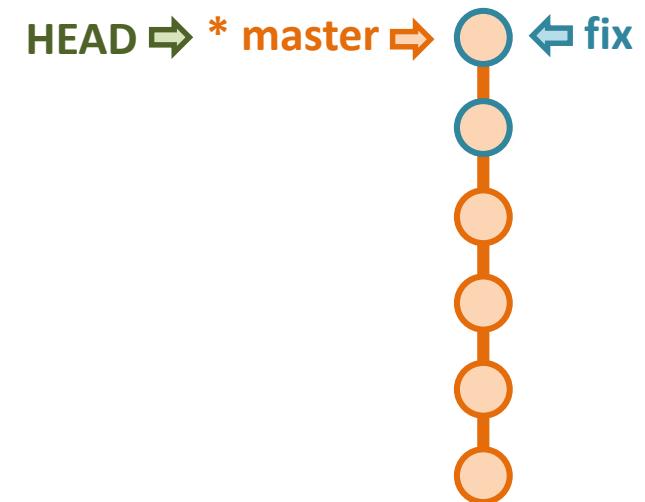
1. Create new branch "fix"



2. Do some work,
add commits



3. Test new feature, then merge
branch "fix" into "master"



git rebase

make a linear history

git rebase: replay commits onto a different base

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

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

Example:

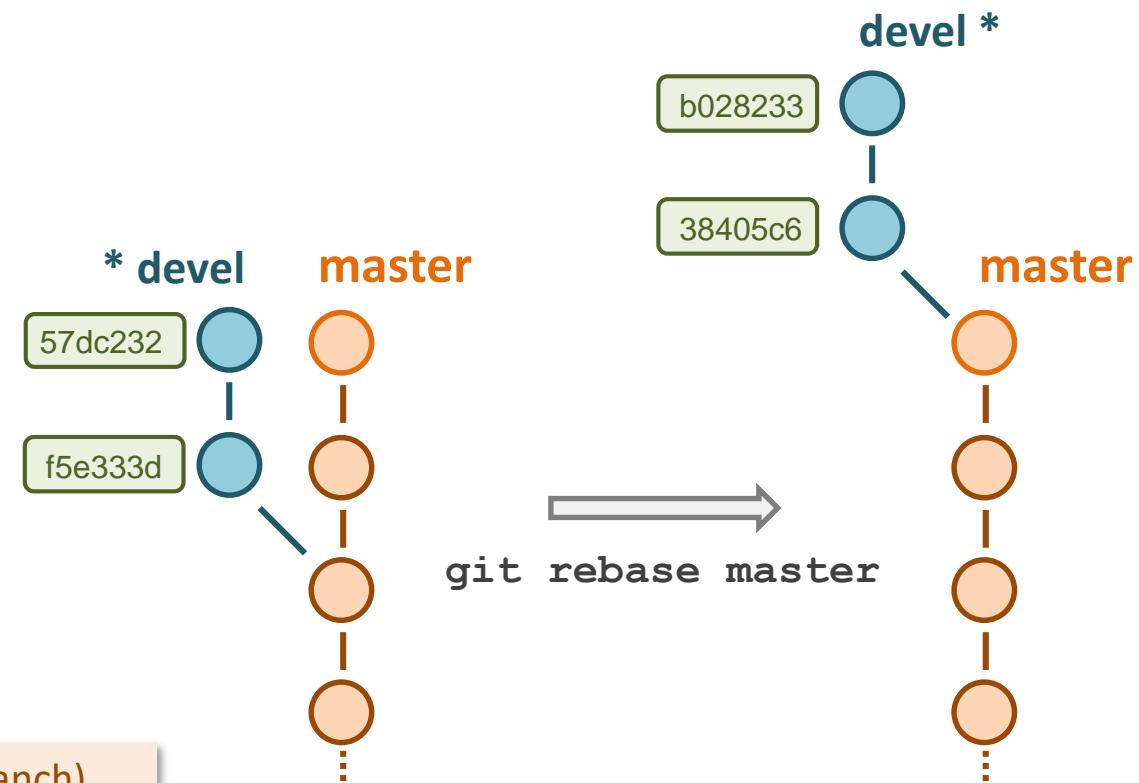
```
$ git branch
* devel      ← Make sure you are on the
master               branch you want to rebase !
```

```
$ git rebase master
```

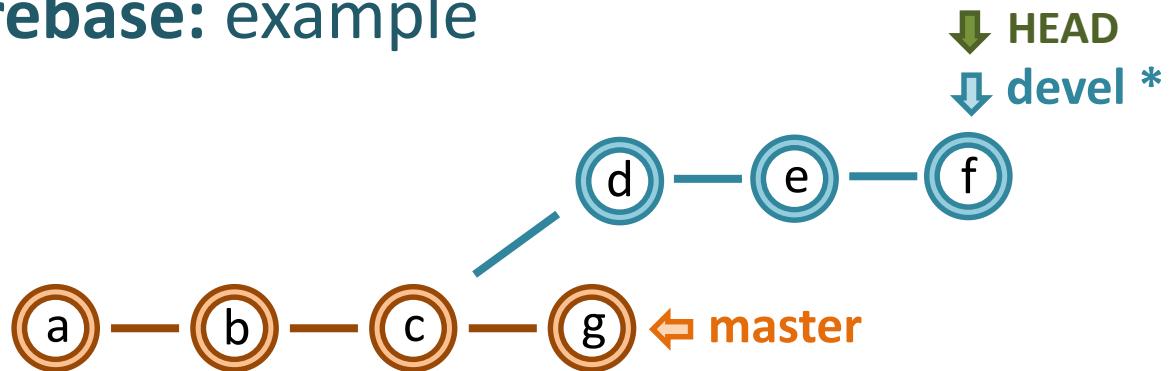
The branch you want to
rebase on.



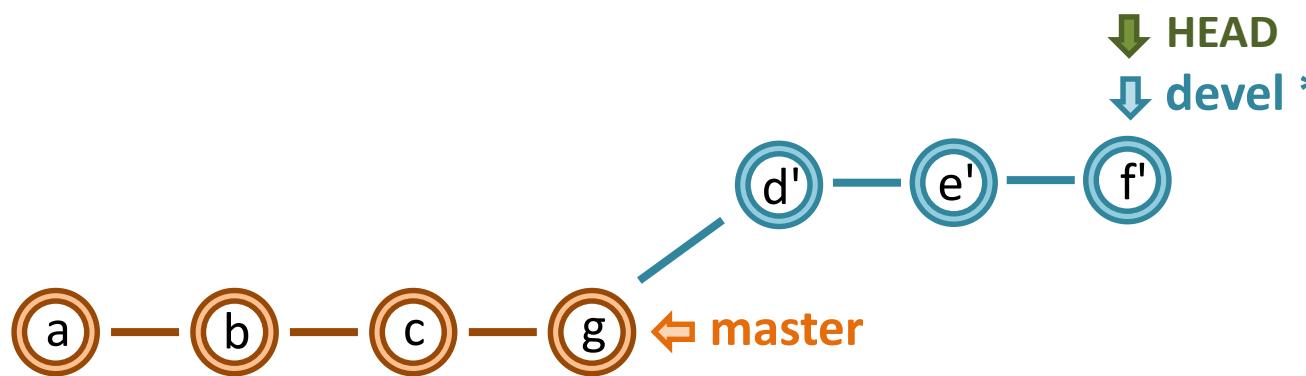
Rebase will modify your commit ID values (history of the rebased branch).
It's best to **only** rebase commits that have never left your own computer.



git rebase: example

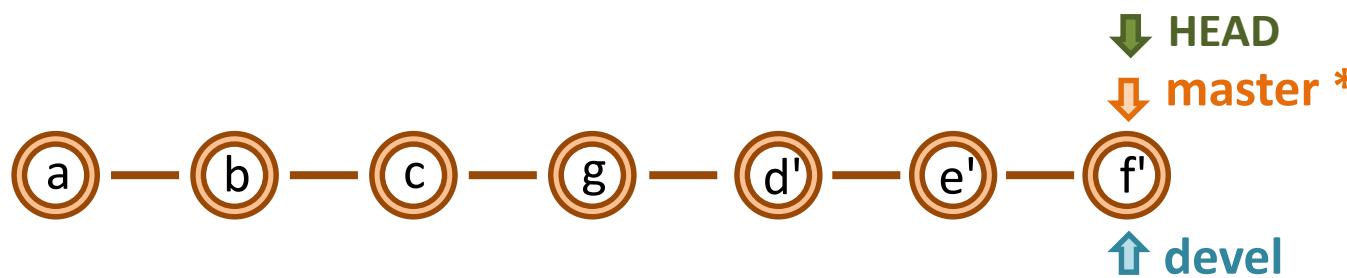


Make sure you are on branch devel !
git branch
If not on devel: git switch devel



git rebase master

We can now fast-forward merge !
Guaranteed conflict free :-)



git switch master
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:

1. Edit the conflicting files, choose the parts you want and 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 master
```

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>" , then run "git rebase --continue".

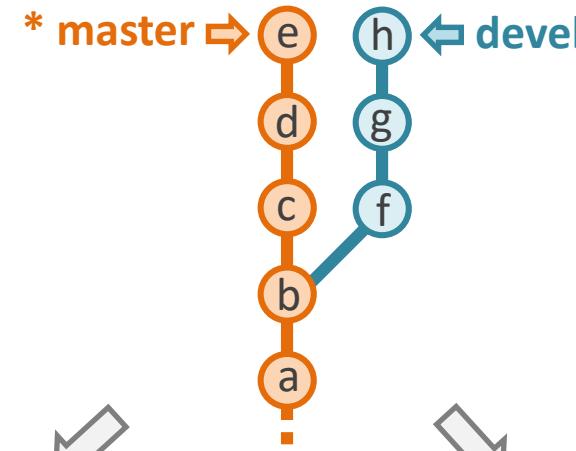
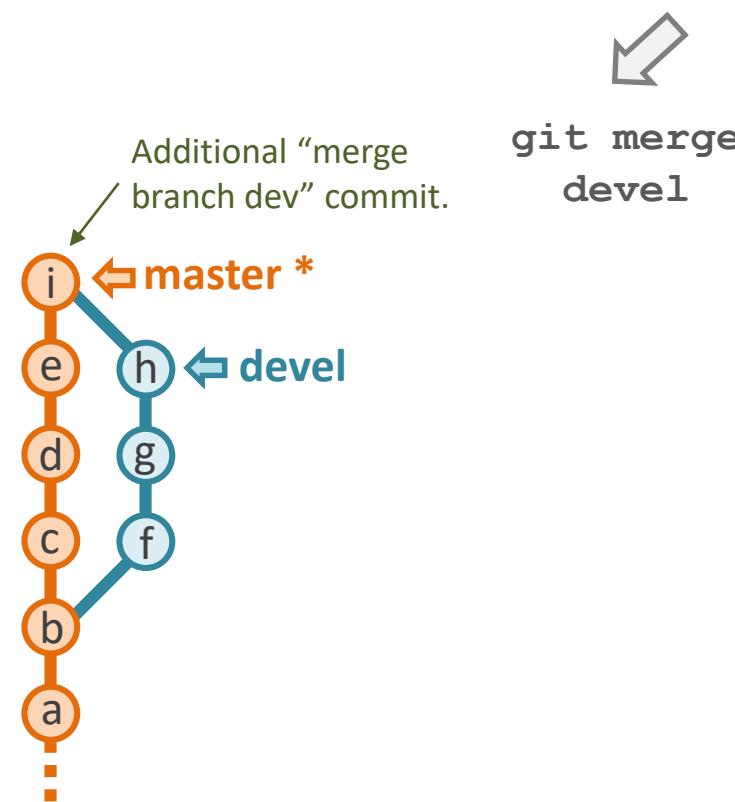
You can instead skip this commit: run "git rebase --skip".
To abort and get back to the state before "git rebase",
run "git rebase --abort".

Branch reconciliation: merge vs. rebase

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

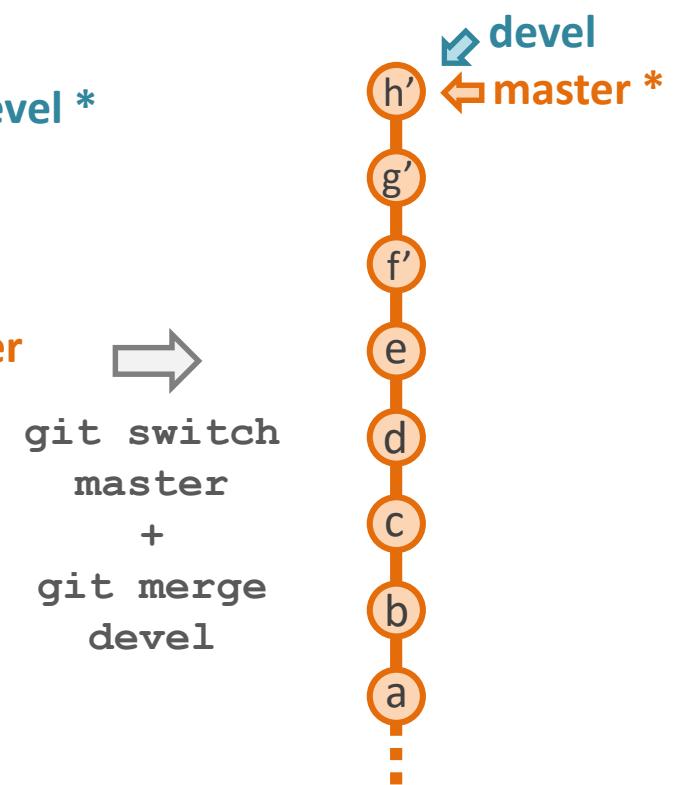
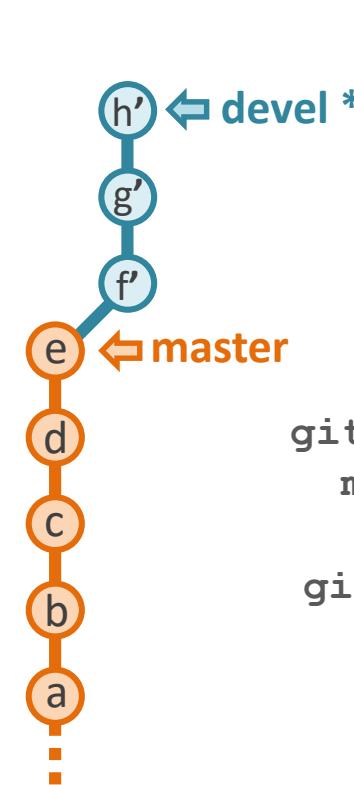
merge strategy (non-fast-forward)

- + Preserves history.
- + Potentials conflicts must be solved only once.
- Creates an additional “merge commit”.
- Often leads to “messy” history.



rebase + merge strategy

- + 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.



demo: branch rebase

feat. manual conflict resolution

git cherry-pick

the "copy/paste" for commits

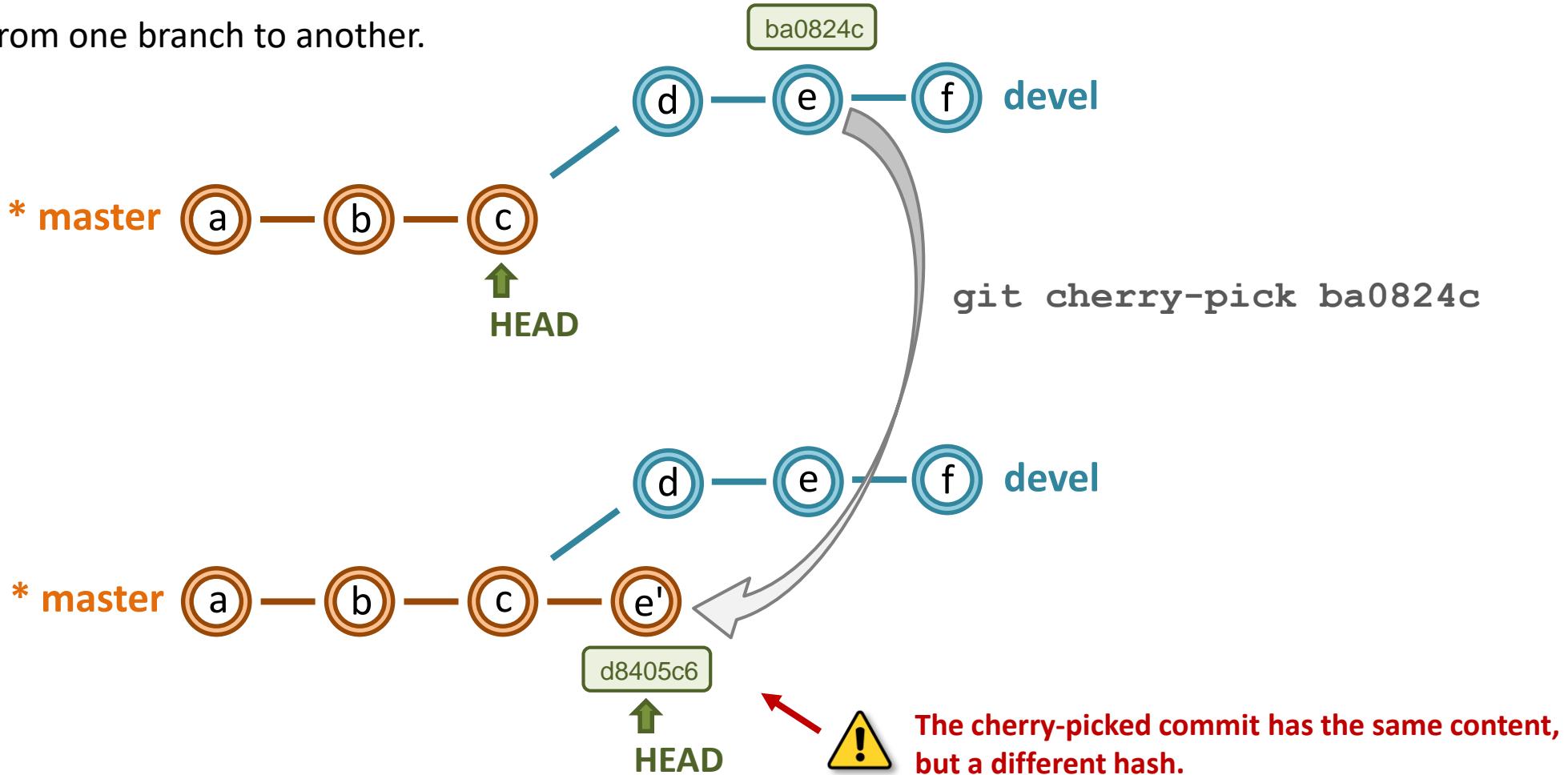
Cherry-pick: merge a single commit into the current branch

- `git cherry-pick` allows to "copy" a single commit to the current branch.

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

Example:

"copy" a fix from one branch to another.



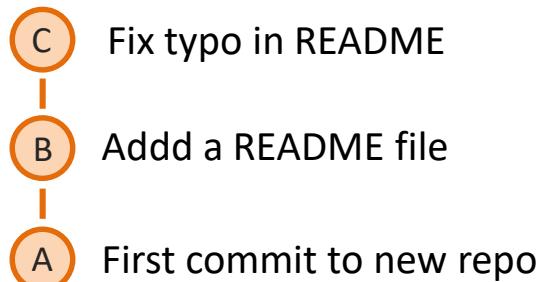
git commit --amend

Overwrite (re-write) the latest commit of a branch

Amending the latest commit of a branch

- Assume that we realize we made a mistake in a file, after a new commit was made.
- In addition, there is also a typo in the commit message...

`git commit -m "Fix typo in README"`



b1241f5 B Add a README.md file

0f1c3bc A First commit to new repo

`git add README.md`

Symbolizes the “staged” corrected README.md file

B Add a README file

A First commit to new repo

Possible but **not ideal**:

- ✗ New commit just to fix a typo !
- ✗ Typo still present in the second commit message !

`git commit --amend -m "Add a README file"`

✓ Cleaner solution



Commit ID is modified !

57dc232 B' Add a README file

0f1c3bc A First commit to new repo

Re-writing the latest commit (amending)

To amend the latest commit of a branch:

1. Stage the changes you want to make to your commit, or, if you just want to modify the commit message, don't stage anything.

2. Run `git commit --amend` (possibly with some added options).

This will open an editor where you can modify the commit message interactively.

```
git commit --amend
```

This is to enter the new commit message directly in the command.

```
git commit --amend -m "new message"
```

This is to keep the commit message unchanged (only edit the content of the commit).

```
git commit --amend --no-edit
```

demo: commit amending

git checkout

retrieving data from earlier commits

Checkout of individual files

- Checking out an individual file updates both the working tree and the index.

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

Example:

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

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

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



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 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
```

- But you will enter a "detached HEAD" state.... →
- To get back to a “normal” state:
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



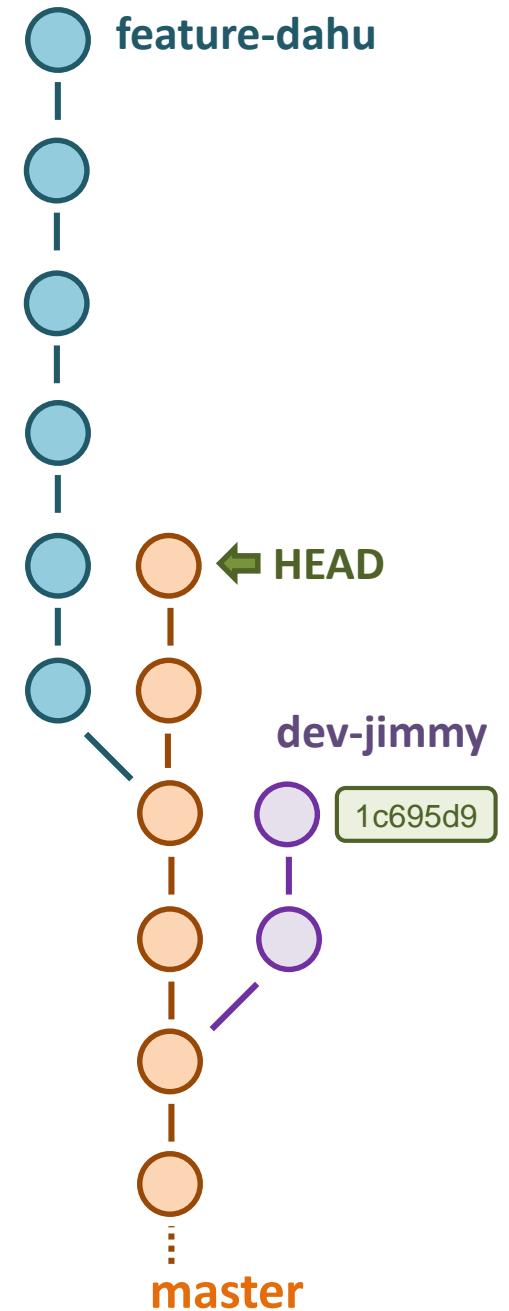
This exercise has helper slides

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
```



Working with **remotes**

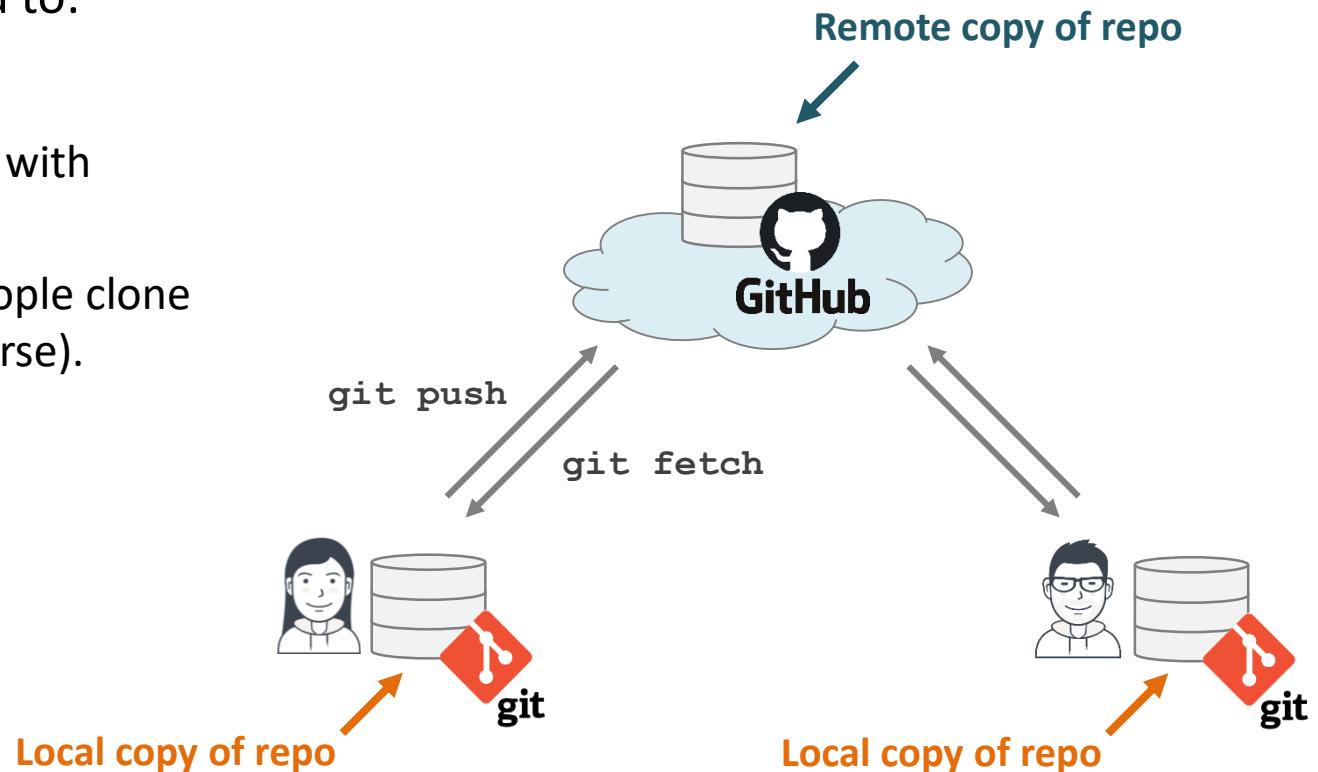
Linking your local repo with an online server

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:

- Create a backup/copy of 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).



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>
```

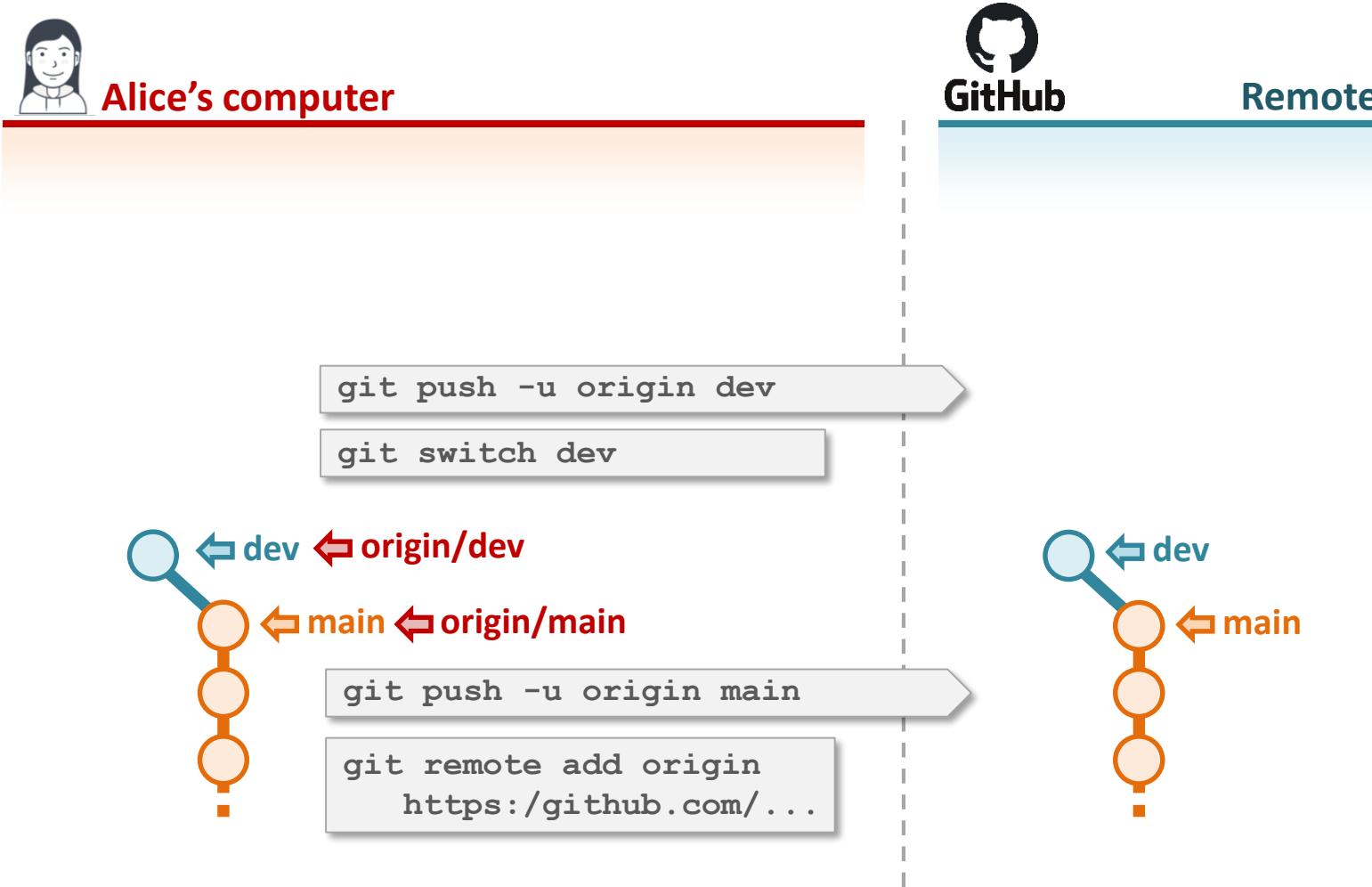
Note: by convention, the `<remote name>` is generally set to `origin`.

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
```

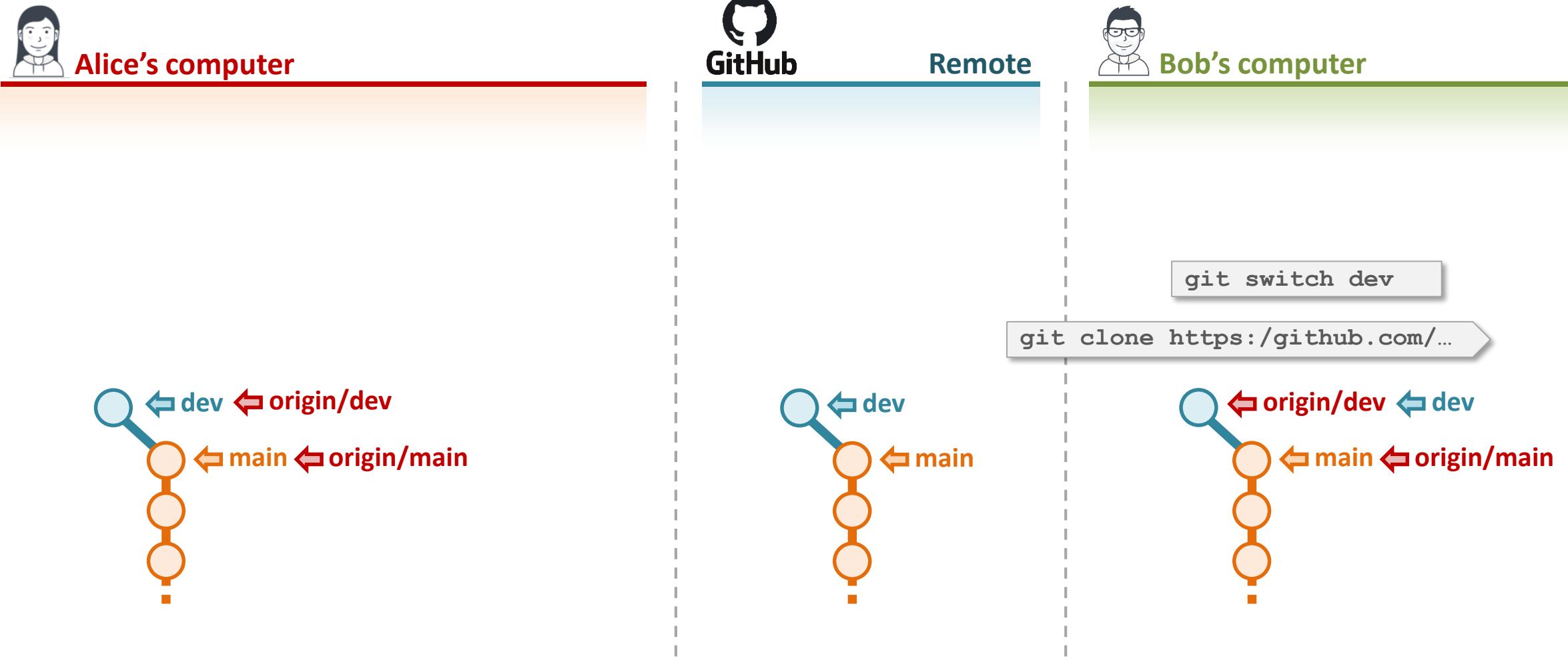
Example – part 1: creating a new remote and pushing new branches.



Alice 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`.
2. She pushes her branch `main` to the remote using `git push -u origin main` (the branch has no upstream, so `-u/--set-upstream` must be used).
3. She pushes her branch `dev` to the remote (**Important:** you must switch-to/checkout the branch before pushing).

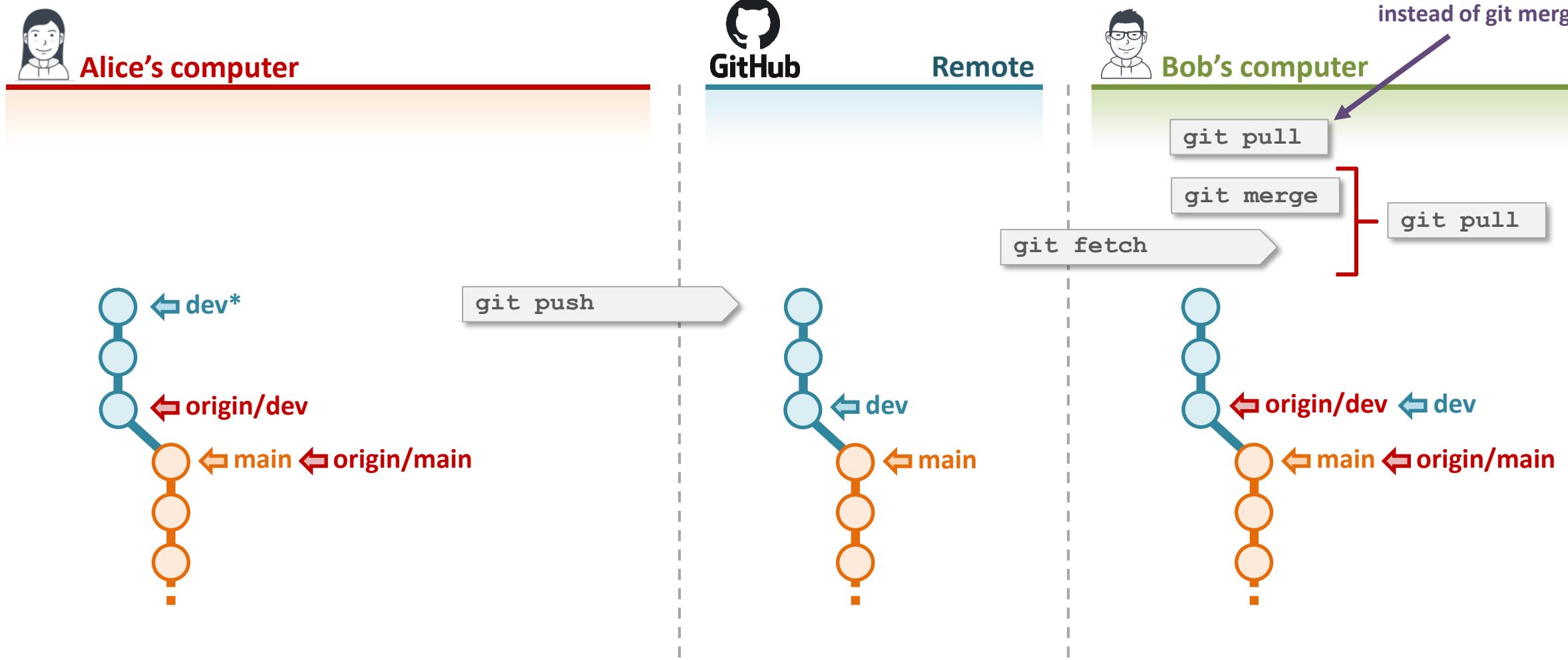
Example – part 2: cloning a remote and checking-out branches.



Bob has now joined the team to work with Alice.

1. He **clones** the repo from GitHub (note: at this point, Bob has no local `dev` branch - but he has 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 of `git switch`).

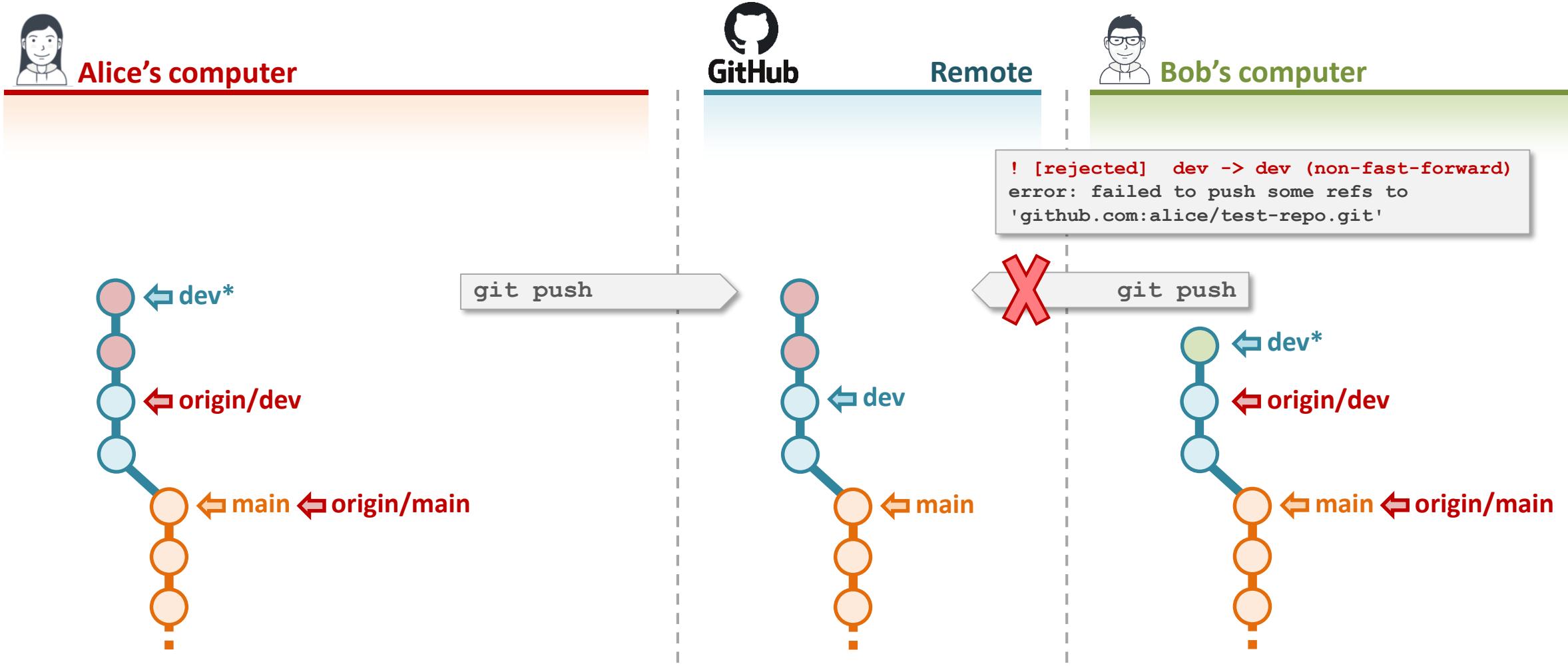
Example – part 3: pushing and pulling changes.



1. Alice adds 2 new commits to `dev`. She then 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` download all new changes/updates from the remote, but does not modify your local branches.

To merge, you can also simply run `git pull` instead of `git merge`.

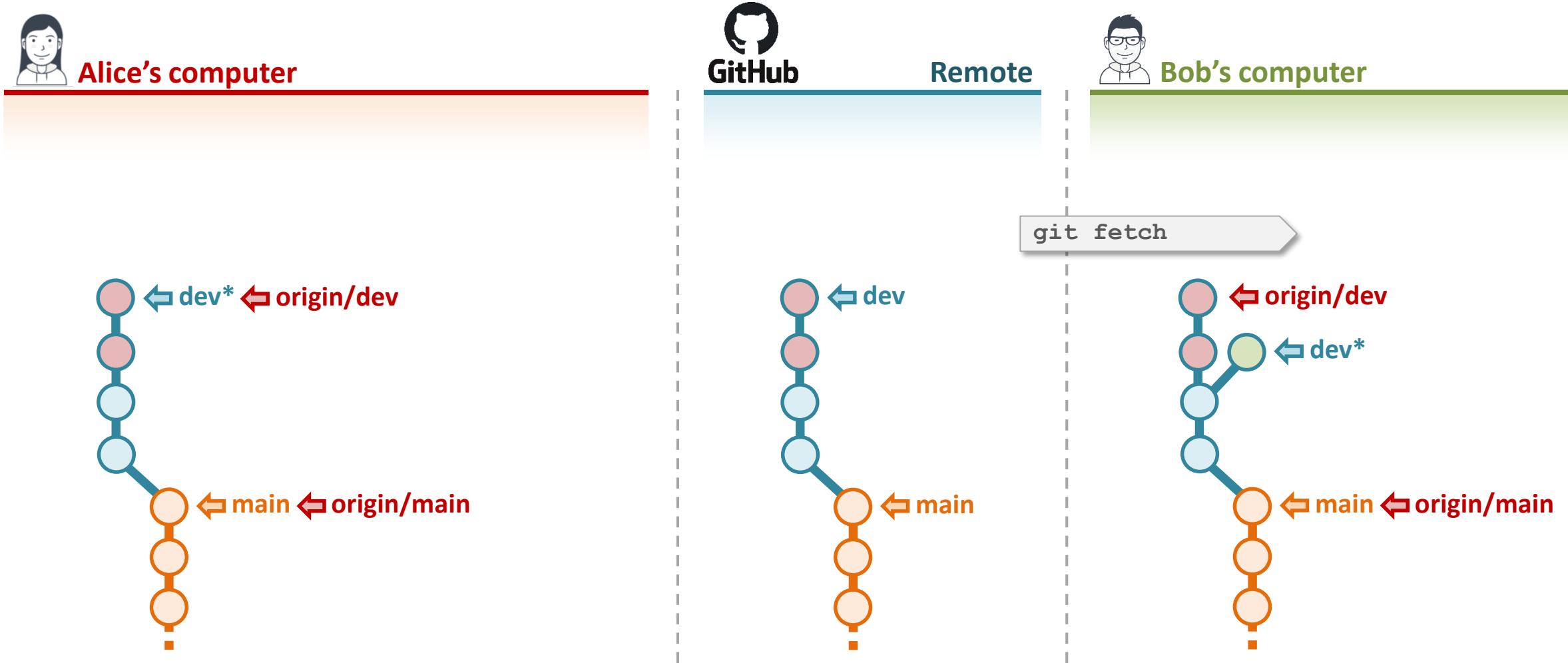
Example – part 4: reconciliation of a diverging history.



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

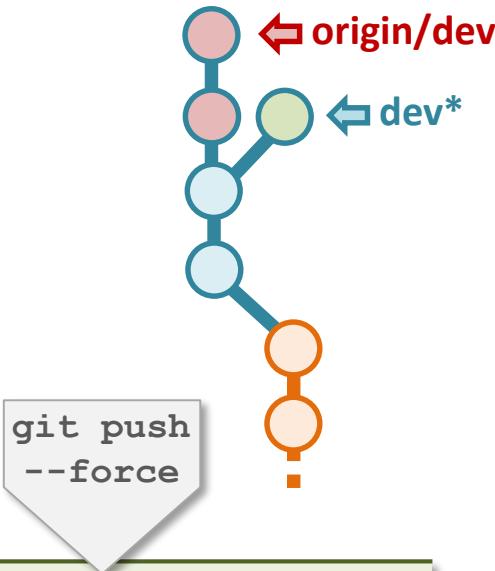


In order to be able to push his changes to the remote, Bob must first reconcile his local `dev` branch with the remote...

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.

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:



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



Option 1 - reconciliations using `merge`.

This is equivalent to:

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

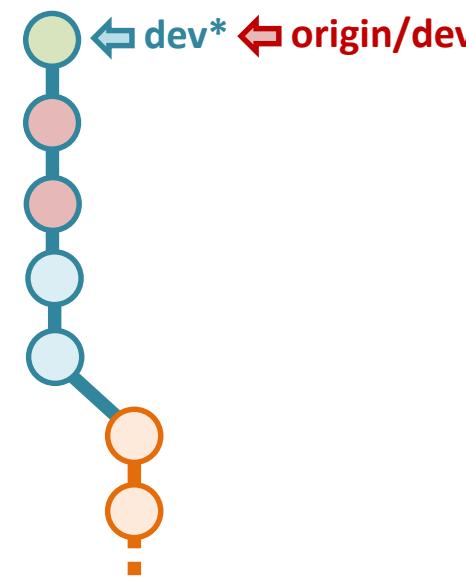
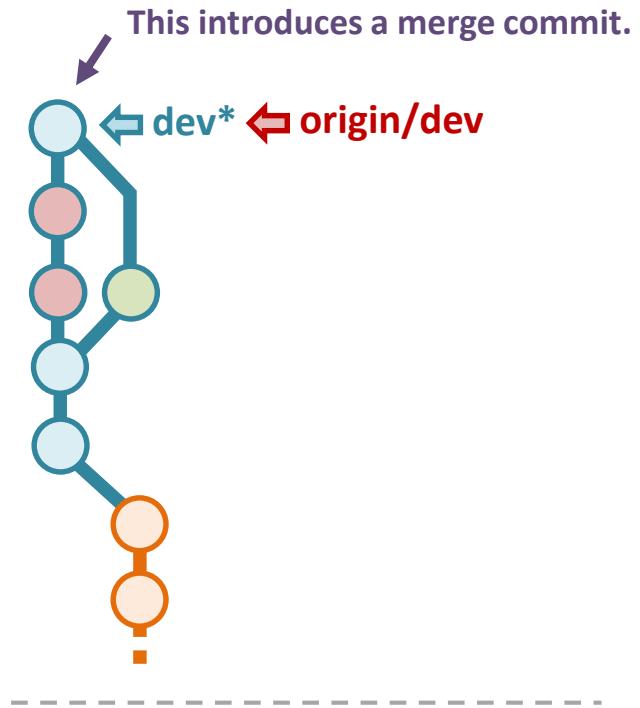
```
git pull --no-rebase
```

`git pull --rebase`

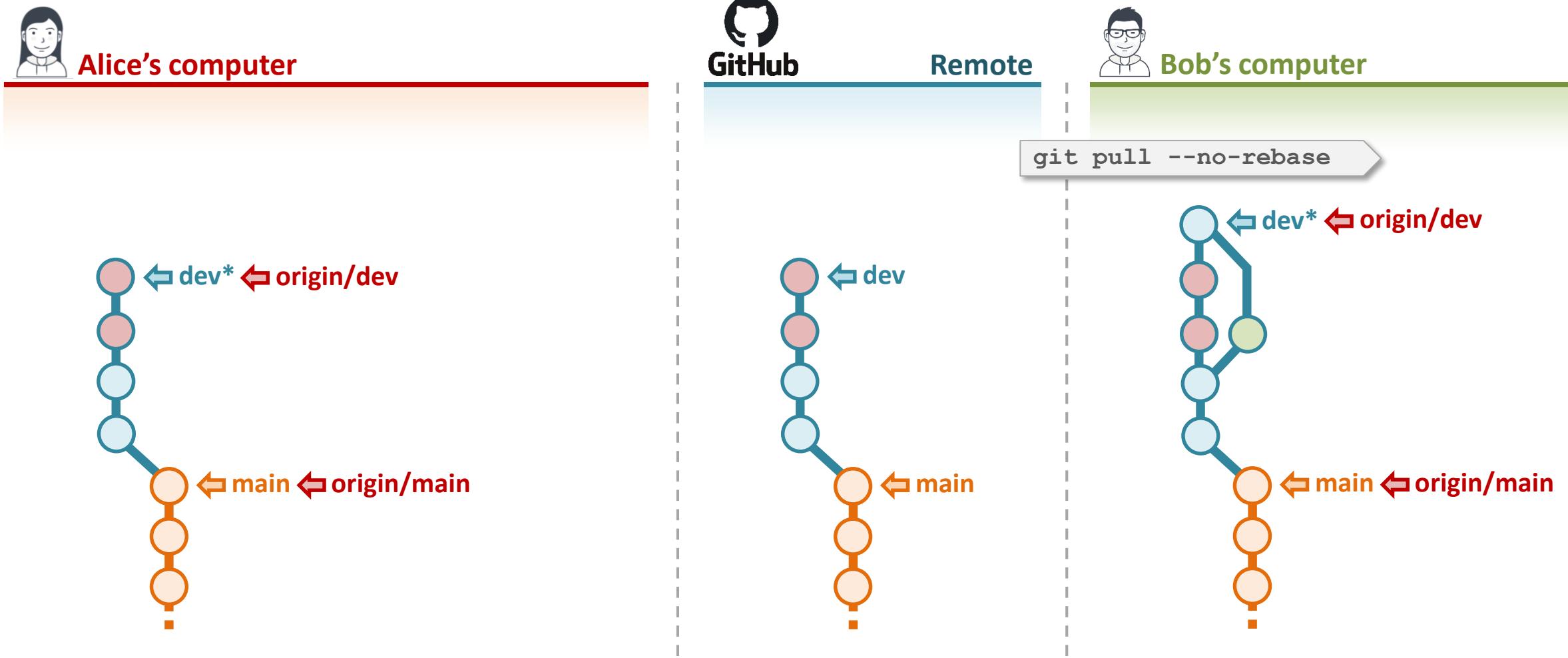
Option 2 - reconciliations using rebase.

This is equivalent to:

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



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



Bob decides to follow a “merge” strategy and runs `git pull --no-rebase`.

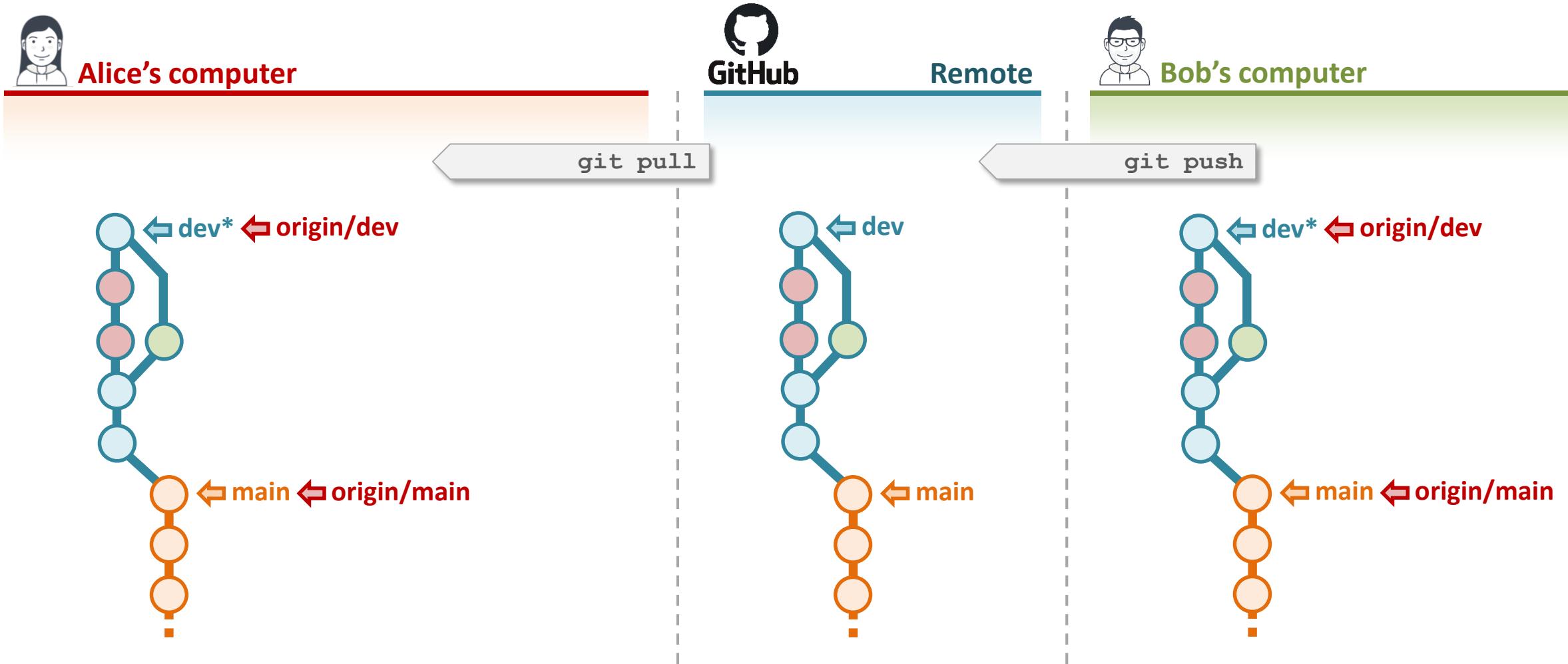
Note: depending on the version of Git, the default behavior of `git pull` is different:

- Newer versions default to `git pull --ff-only`.
- Older versions default to `git pull --no-rebase`.

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
```

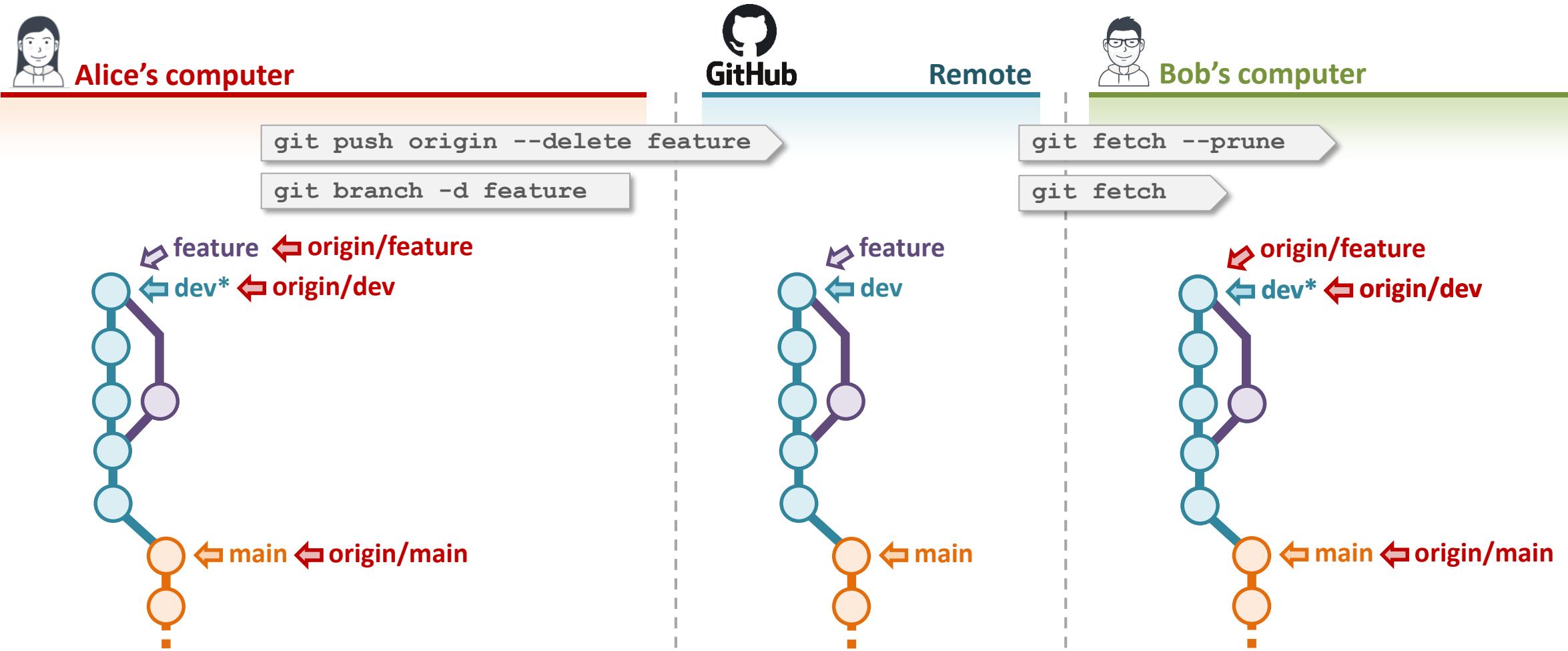
Example – part 4: reconciliation of a diverging history (the end!).



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

Alice can then **pull** them.

Example – part 5: deleting branches on the remote.



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 feature`.
2. Alice deletes the feature branch on the remote with `git push origin --delete feature`.
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), Bon has to use `git fetch --prune`.

GitHub

collaborate and share your work

GitHub – an online home for your Git repos

- GitHub [github.com] is a hosting platform for Git repositories.
- 73+ million users, 200+ million repositories (as of 2022).
- Very popular to share/distribute open source 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:
 - **GitLab** [gitlab.com], which can also be installed as a local instance: e.g. `gitlab.sib.swiss`.
 - **BitBucket** [bitbucket.org].



Creating a new project on GitHub

To create a new repo, click on



... or click on your user icon (top right), then Your profile ...

Either on the welcome screen at <https://github.com> (after signing-in)...

The image shows three screenshots of the GitHub interface. The top-left screenshot shows the main GitHub welcome screen with a red dashed circle around the green 'New' button in the top navigation bar. A purple arrow points from the text 'To create a new repo, click on' to this button. The top-right screenshot shows the user profile dropdown menu with 'Your profile' selected, indicated by a purple arrow. The bottom screenshot shows the repository list with the 'Repositories' tab selected, also indicated by a purple arrow. All screenshots include purple arrows pointing to specific UI elements.

... and the **Repositories** tab.

Create a new repository

A repository contains all project files, including the revision history. Already have a project repository elsewhere?
[Import a repository.](#)

Owner * Repository name *

 sibgit / my-new-project ✓

Great repository names are short and memorable. Need inspiration? How about [super-duper-guacamole?](#)

Description (optional)

A first test project on GitHub

 Public

Anyone on the internet can see this repository. You choose who can commit.

 Private

You choose who can see and commit to this repository.

Initialize this repository with:

Skip this step if you're importing an existing repository.

Add a README file

This is where you can write a long description for your project. [Learn more.](#)

Add .gitignore

Choose which files not to track from a list of templates. [Learn more.](#)

Choose a license

A license tells others what they can and can't do with your code. [Learn more.](#)

Create repository

Enter a name for your new repository.

Project description.

Select whether your repo should be:

- **Public** - anyone can access it (read from it).
- **Private** - only people you authorize.

Note: even if a repo is public, only authorized members can push changes to it.

Pre-fill the repository with some files:

- **README** – A text file that is displayed on the homepage of your repo (with markdown rendering).
- A **.gitignore** file selected from a list of templates.
- A **license** file selected from a set of standard licenses (e.g. GPL, MIT, ...).

Click Create repository.

The home page of an empty repository provides instructions to get started...

The screenshot shows the GitHub landing page for an empty repository. The top navigation bar includes links for Code, Issues, Pull requests, Actions, Projects, Wiki, Security, Insights, and Settings. A prominent callout box titled "Quick setup — if you've done this kind of thing before" provides instructions for setting up a new repository. It offers options to "Set up in Desktop" or "HTTPS" or "SSH", with the URL <https://github.com/sibgit/test-project.git> displayed. Below this, text encourages creating a new file or uploading an existing one, and recommends including a README, LICENSE, and .gitignore. Another section, "...or create a new repository on the command line", lists Git commands: echo "# test-project" >> README.md, git init, git add README.md, git commit -m "first commit", git branch -M main, git remote add origin https://github.com/sibgit/test-project.git, and git push -u origin main. A dashed orange box highlights the last two commands. To the right, purple arrows point from these commands to explanatory text: "Add remote to your local repo." and "Push a branch (here “main”) to the remote.". A third section, "...or push an existing repository from the command line", also highlights the same set of commands with a dashed orange box, and a purple arrow points to the text "Same commands as above...".

Code Issues Pull requests Actions Projects Wiki Security Insights Settings

Quick setup — if you've done this kind of thing before

Set up in Desktop or HTTPS SSH https://github.com/sibgit/test-project.git

Get started by [creating a new file](#) or [uploading an existing file](#). We recommend every repository include a [README](#), [LICENSE](#), and [.gitignore](#).

...or create a new repository on the command line

```
echo "# test-project" >> README.md  
git init  
git add README.md  
git commit -m "first commit"  
git branch -M main  
git remote add origin https://github.com/sibgit/test-project.git  
git push -u origin main
```

Add remote to your local repo.
Push a branch (here “main”) to the remote.

...or push an existing repository from the command line

```
git remote add origin https://github.com/sibgit/test-project.git  
git branch -M main  
git push -u origin main
```

Same commands as above...

When at least 1 file is present in the repo, the **home page** of your Git repo looks like this:

The screenshot shows the GitHub repository page for 'sibgit/test'. The 'Code' tab is selected. The top navigation bar includes 'Pull requests', 'Issues', 'Marketplace', 'Explore', and a user icon. Below the navigation, the repository name 'sibgit / test' is shown, along with 'Public', 'Pin', 'Unwatch', 'Fork', and 'Star' buttons.

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 content area)

Annotations on the right side:

- About** (Green button with a purple arrow) - A box contains 'No description, website, or topics provided.' and links for 'Readme' and '0 stars'.
- Clone** (Green button) - A box contains 'HTTPS' (underlined), 'SSH', 'GitHub CLI', and a URL field: 'https://github.com/sibgit/test.git'. A purple box labeled 'To copy the repo's URL.' has an arrow pointing to the URL field.
- Download ZIP** (Green button)

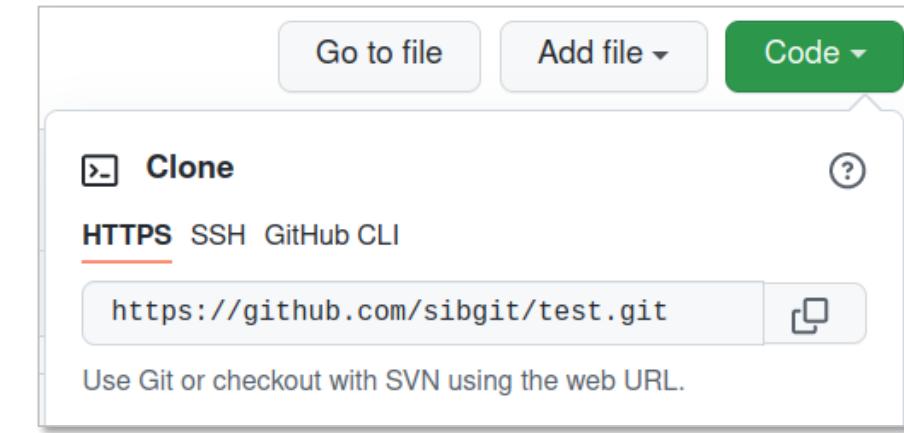
Cloning a repo: HTTPS vs. SSH

When cloning (or adding a remote) via:

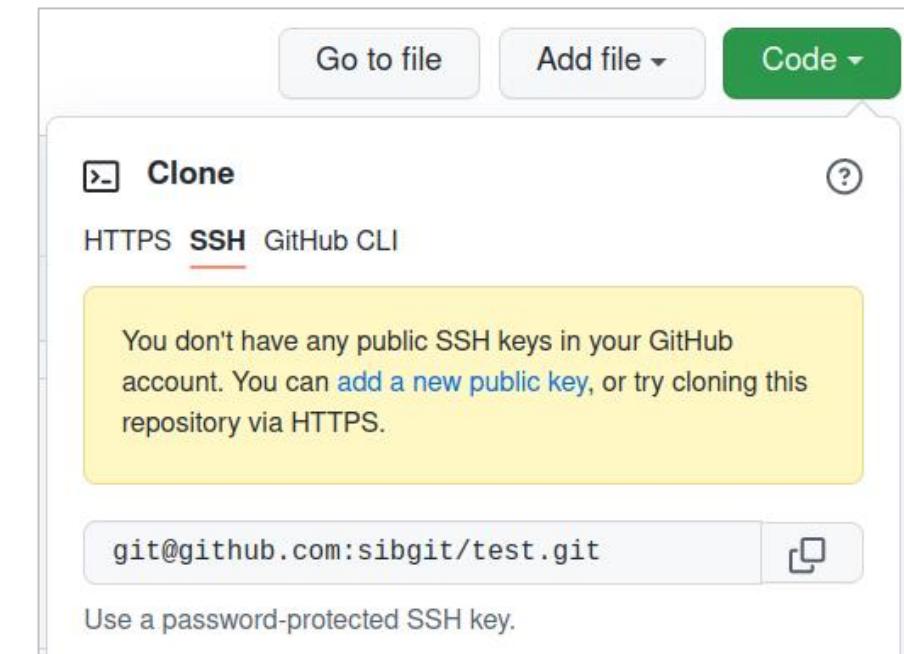
- **HTTPS**, you will need to provide a **personal access token** (PAT) as authentication credential.
 - Credentials are only needed to push data to the remote.
 - Your local Git repo will in principle store the login credentials, so you need to provide them only once.
 - Instructions on **how to generate a PAT** can be found in the *helper slides of exercise 4*.
- **SSH**, you will need to add a **public SSH key** to your GitHub account.

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

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



The screenshot shows the GitHub interface with a repository cloned. The URL `https://github.com/sibgit/test.git` is highlighted in a red box. A tooltip below it says: "Use Git or checkout with SVN using the web URL."



The screenshot shows the GitHub interface with a repository cloned. The URL `git@github.com:sibgit/test.git` is highlighted in a red box. A tooltip below it says: "Use a password-protected SSH key."

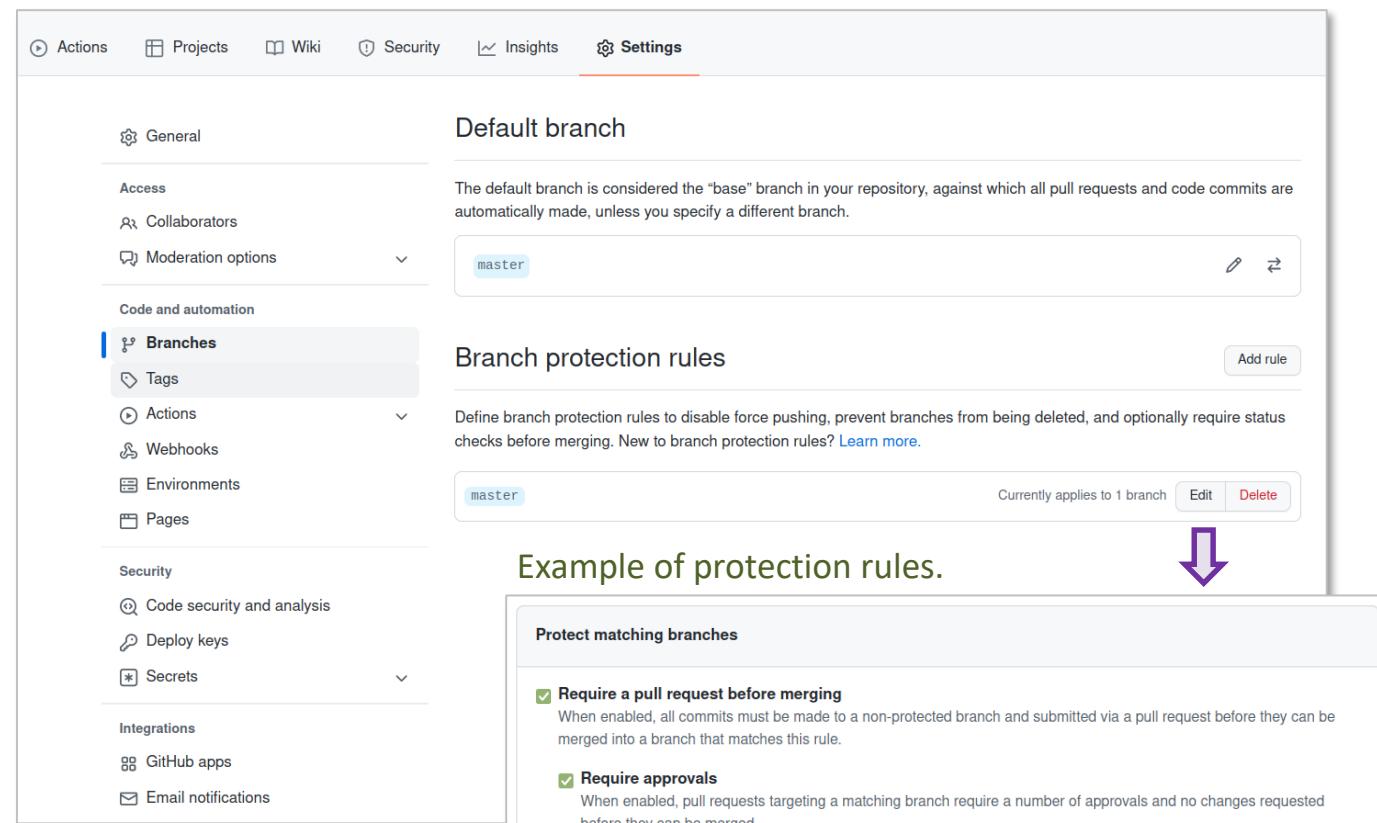
GitHub Pull Requests (PR)

Pull Requests * (PR) are a way to ask someone to integrate your changes (i.e. merge your branch) into another branch.

- PRs perform a branch merge operation on the GitHub remote (rather than on your local copy).
- Typically, a PR is created to merge a feature branch into the main/master branch.

Why use PRs instead of a local merge (and push)?

- The branch you want to merge into (e.g. main/master) 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.



The screenshot shows the GitHub repository settings page for a repository named "Default branch". The "Settings" tab is selected. On the left, there's a sidebar with sections like General, Access, and Code and automation. Under "Code and automation", the "Branches" section is highlighted. In the main area, under "Default branch", it says: "The default branch is considered the "base" branch in your repository, against which all pull requests and code commits are automatically made, unless you specify a different branch." Below this, there's a dropdown menu set to "master". Under "Branch protection rules", it says: "Define branch protection rules to disable force pushing, prevent branches from being deleted, and optionally require status checks before merging. New to branch protection rules? [Learn more](#)." There's a dropdown menu also set to "master". At the bottom right of this section, there are "Edit" and "Delete" buttons. A purple arrow points down to the "Protect matching branches" section in the "Example of protection rules" box.

Default branch

The default branch is considered the "base" branch in your repository, against which all pull requests and code commits are automatically made, unless you specify a different branch.

master

Branch protection rules

Define branch protection rules to disable force pushing, prevent branches from being deleted, and optionally require status checks before merging. New to branch protection rules? [Learn more](#).

master

Currently applies to 1 branch [Edit](#) [Delete](#)

Example of protection rules.

Protect matching branches

- Require a pull request before merging**
When enabled, all commits must be made to a non-protected branch and submitted via a pull request before they can be merged into a branch that matches this rule.
- Require approvals**
When enabled, pull requests targeting a matching branch require a number of approvals and no changes requested before they can be merged.
Required number of approvals before merging: 1 ▾
- Dismiss stale pull request approvals when new commits are pushed**
New reviewable commits pushed to a matching branch will dismiss pull request review approvals.
- Require review from Code Owners**
Require an approved review in pull requests including files with a designated code owner.

* On GitLab, pull requests are called **Merge Requests** (MR), but it's the exact same thing.

** **Protected** branches are branches where push operations are limited to some users.

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

How to open a Pull Request on GitHub: step-by-step

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

You will need to do this in exercise 4!

Pull requests tab

New pull request

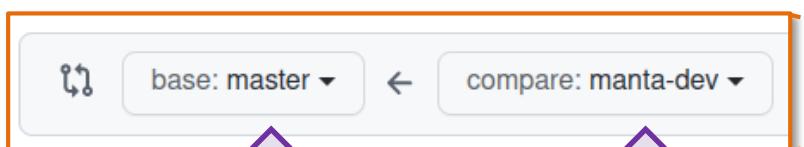
Pending pull requests will be listed here...

There aren't any open pull requests.

You could search [all of GitHub](#) or try an [advanced search](#).

2. Click on **New pull request**.

3. Select the branches to merge:



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**.

If there are conflicts, you probably need to rebase your branch and resolve them.

The screenshot shows the GitHub 'Comparing changes' interface. At the top, it says 'base: master' and 'compare: manta-dev'. A green checkmark indicates 'Able to merge'. Below this, it says 'Discuss and review the changes in this comparison with others. Learn about pull requests'. It shows '2 commits', '2 files changed', and '1 contributor'. The commit details are listed as follows:

- o 2 commits
- o Commits on Mar 10, 2022
 - Add info on habitat and behavior for manta ray
sibgit committed 18 minutes ago
 - Add image for manta ray
sibgit committed 17 minutes ago

Showing 2 changed files with 14 additions and 6 deletions.

Detailed Diff View:

```

  ✓ 20 manta_ray.html
  @@ -4,28 +4,36 @@
  4   4           <link rel="stylesheet" href="styles.css">
  5   5           </head>
  6   6           <body>
  7   -           <h1>?? Animal name</h1>
  7   +           <h1>Manta Ray - <i>Mobula sp.</i></h1>
  8   8
  9   -           
  9   +           
  10  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.
  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  +
  17  +
  18  +
  19  +
  20  </p>
  
```



[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.

2

base: master ▾

con

pare: manta-dev ▾

✓ **Able to merge.** These branches can be automatically merged.



Manta dev

Write | Preview

Preview

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

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

Please merge :smiley_cat

Attach files by dragging & dropping, selecting or pasting them

Create pull request

 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)

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

-o  Add info on habitat and behavior for manta ray d0a01b1
-o  Add image for manta ray 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 workflow for a pull request (PR). It shows three main views:

- Top View (Pull Requests):** Shows a list of pull requests. One pull request, labeled "Manta dev #27", is highlighted with a purple dashed box. A purple arrow points from this box down to the detailed view of the PR.
- Middle View (Pull Request Detail):** Shows the details of pull request #27. It includes a conversation with a comment from "robinengler" asking for merge. Below the conversation, there are commit details and deployment information. A purple dashed box highlights the "Review required" status at the bottom, which is also indicated by a purple arrow pointing to the review interface.
- Bottom View (Review Interface):** A modal window titled "Finish your review". It contains a "Write" tab with a rich text editor, a message "Looking good, thanks for the contribution!", and a "Review changes" button. Below the message, there are three options: "Comment", "Approve", and "Request changes".



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 · Collaborator

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

sibgit added 2 commits 38 minutes ago

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

sibgit approved these changes 1 minute ago

sibgit left a comment · Owner

Looking good, thanks for the contribution !

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

This branch has not been deployed · No deployments

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

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 · Collaborator

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

sibgit added 2 commits 40 minutes ago

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

sibgit approved these changes 3 minutes ago

sibgit left a comment · Owner

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).

Repository settings (only available if you are the owner).

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

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

Click here to add a collaborator

Who has access

PUBLIC REPOSITORY This repository is public and visible to anyone.

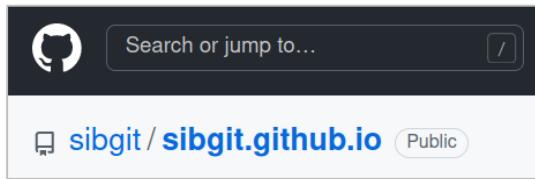
DIRECT ACCESS 26 have access to this repository. 17 collaborators. 9 invitations.

Add people

Manage access

Actions	Collaborators	Moderation options
Code and automation	Branches	
	Tags	
	Actions	
	Webhooks	
	Environments	
	Pages	
Security		
	Code security and analysis	
	Deploy keys	
	Secrets	
Integrations		
	GitHub apps	
	Email notifications	

Other GitHub features (some of them)



“Home” of
your repo
(repo content)

Issue tracker

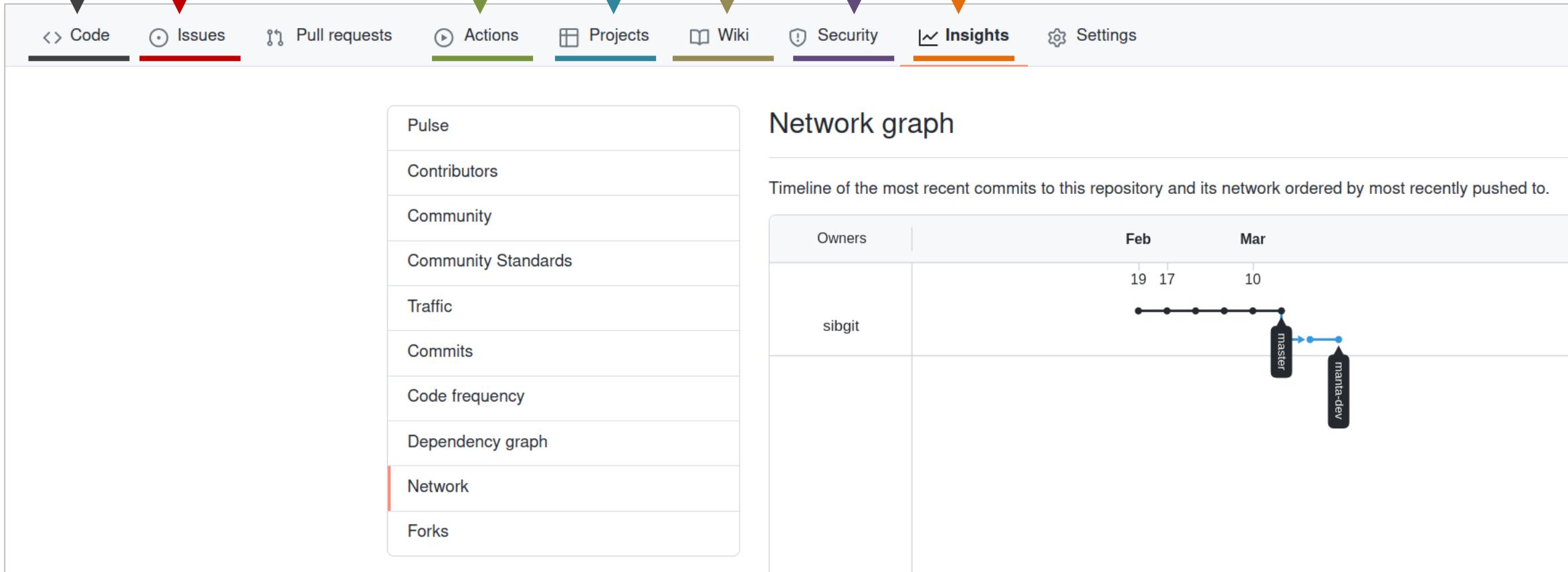
**Continuous integration
(automated testing)**

**Group issues and
PR by topics.**

**Add a wiki for
your project.**

**Setup automated security scanning
for your code (vulnerability check).**

**Statistics about your
repo’s activity.**



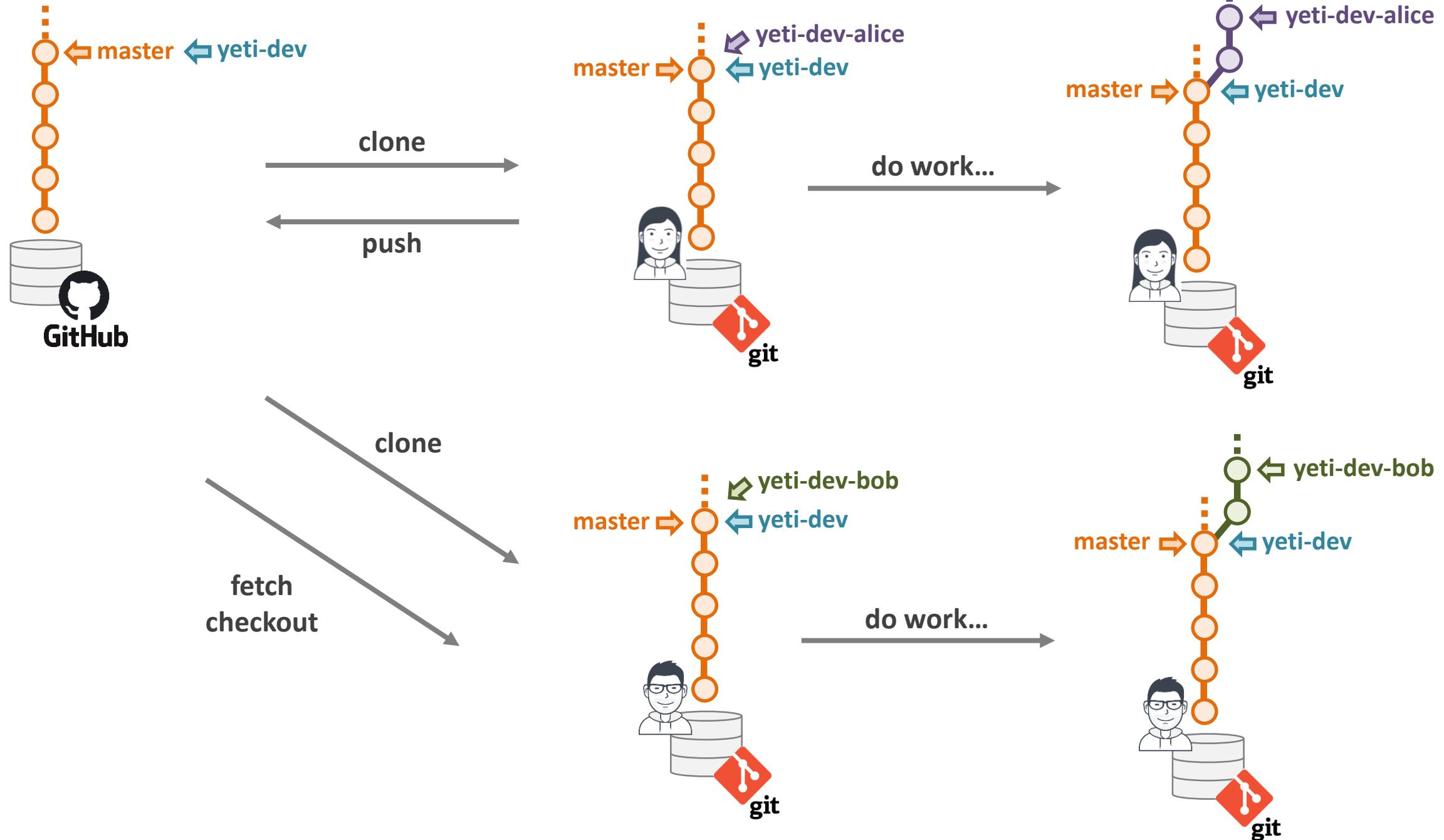
exercise 4

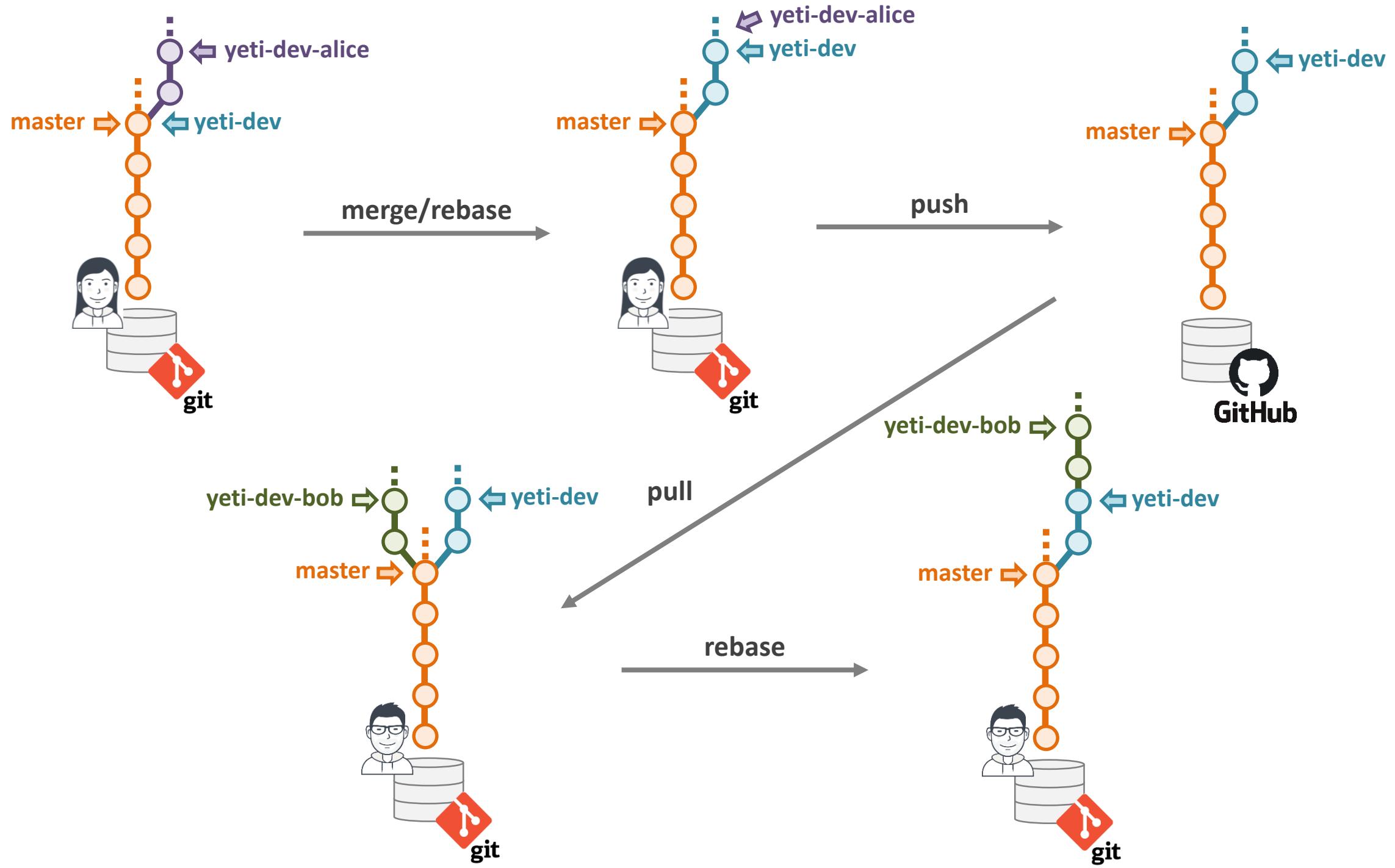
The Awesome Animal Awareness Project



This exercise has helper slides

Exercise 4 help: branch – rebase – merge sequence

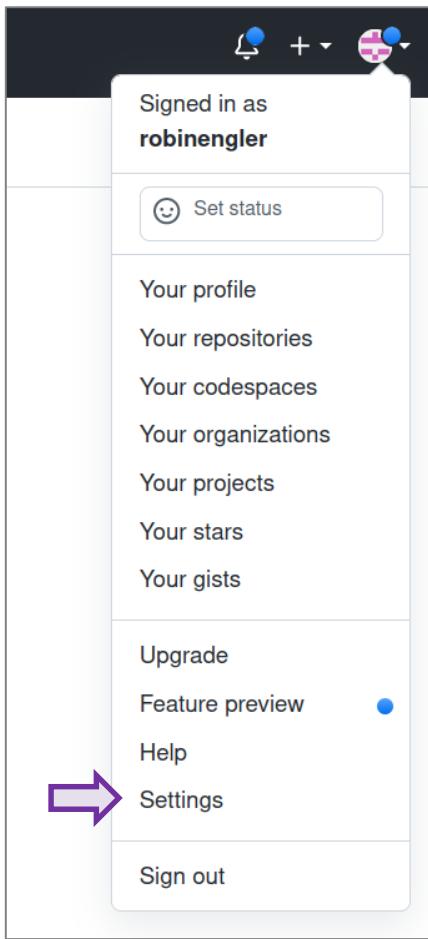




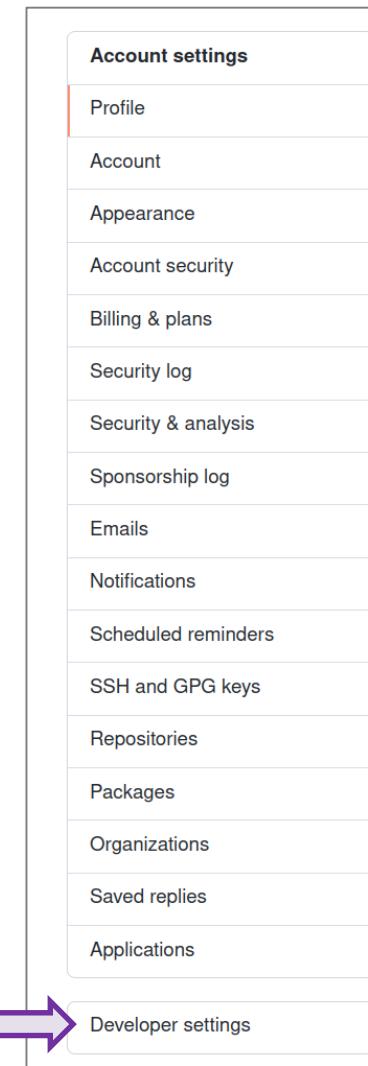
Exercise 4 help: generating a “personal access token” on GitHub

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

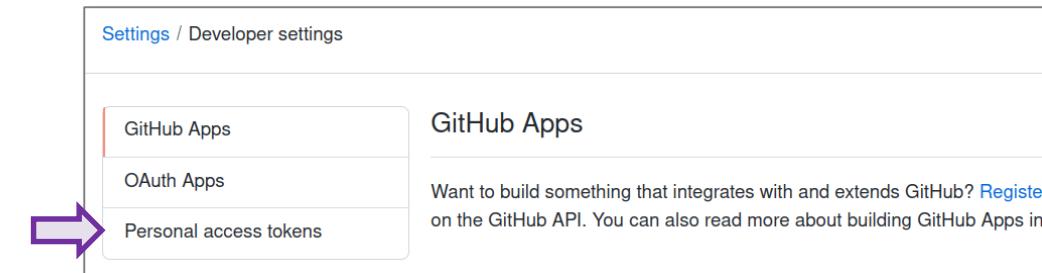
1. In your user profile (top right), click on **Settings**.



2. In your Account settings, click on **Developer settings**.



3. In **Developer settings**, click on **Personal access tokens**.



Go to next page

Exercise 4 help: generating a “personal access token” on GitHub

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

New personal access token

Personal access tokens function like ordinary OAuth access tokens. They can be used 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 Fri, Nov 5 2021

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

Generate token **Cancel**

5. **Copy the personal access token** to a safe location (for now maybe in a text file, but ideally in a password manager). You will not be able to access it again later.

Personal access tokens

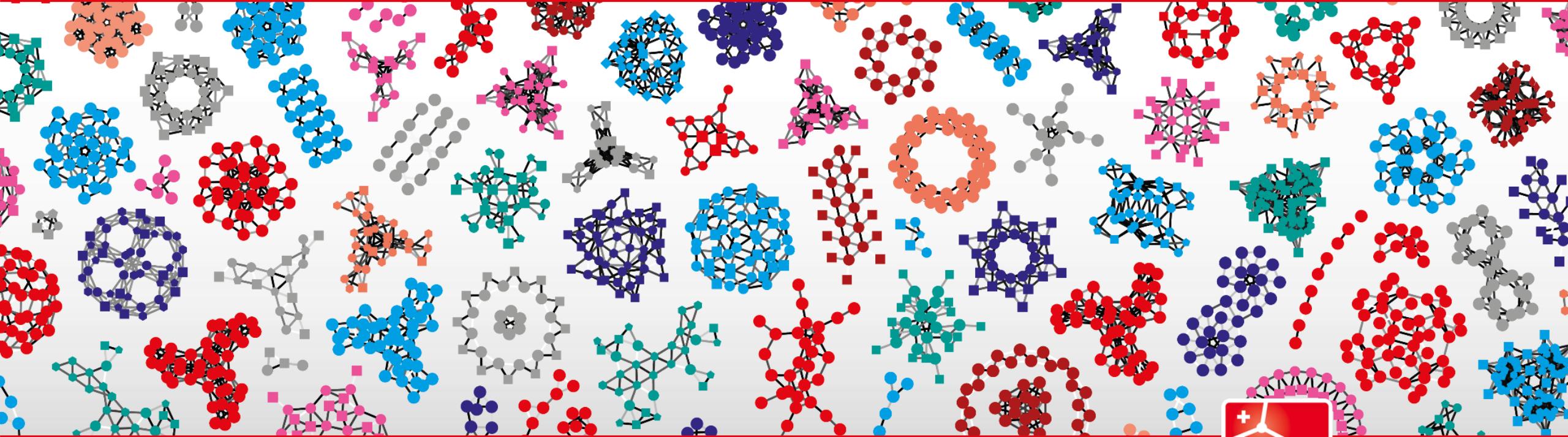
Generate new token Revoke all

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_9sy...  Delete

6. 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.



SIB

Swiss Institute of
Bioinformatics

Thank you for attending this course