Collaborative distributed version control and troubleshooting @ KIT

This material was originally developed by CodeRefinery. The original material is visible here and here. Pull requests and fixes are welcome!

The content of this workshop can be roughly divided in two parts.

Effective collaborative software development

How can we share work on a repository of files with others on the internet?

- Share an archive of the directory using email or using some file sharing service: This would lead to many back and forth emails and would be difficult keep all copies synchronized.
- One person's repository on the web: allows one person to keep track of more projects, gain visibility, feedback, and recognition.
- Common repository for a group: everyone can directly update the same repository. Good for small groups.
- Forks or copies with different owners: anyone can suggest changes, even without advance permission. Maintainers approve what they agree with.

Being able to share more easily (going down the above list) is *transformative*, because it allows projects to scale to a new level. **This can't be done without proper tools.**

We will discuss the centralized as well as the forking workflows.
During the workshop, you will collaborate in small groups using the same **forge**.
Some of the details might not apply to the forge you are using, please focus on the general ideas.

Fixing problems using Git, and fixing Git problems

Version Control has been sometimes described as "*an unlimited undo button*", that offers important ways to tackle problems and gain insight during development (typically, software development), by inspecting the history and the state of all the files involved.

Of course, using a new tool can introduce additional complexity on top of an already complex workflow. Being in control of the tool can guarantee a much productive and stress-free experience, especially when collaborating with other people.

Expected learning outcomes

❗ Objectives

1. Be able to collaborate with others on remote repositories hosted on Git Forges (e.g., GitHub, GitLab and other similar services);
2. Be able to use git tools to diagnose and fix problems in code or documents (the content of the repository);
3. Be able to fix common issues encountered when deviating from the simplest workflow (*pull - add - commit - push*);
4. Bonus point: fix issues in this repository

⚙ Prerequisites

1. Basic understanding of Git.
2. You need an account on a "Forge", e.g.
   - github.com
   - gitlab.com
   - gitlab.kit.edu
   - codeberg.org

Quick recap on Git Basics: Commits and Branches

The first and most basic task to do in Git is *record changes* using commits.
We will record changes in two ways:

- on a new branch (which supports multiple lines of work at once)
- directly on the "main" branch (which happens to be the default branch here).

❗ Objectives

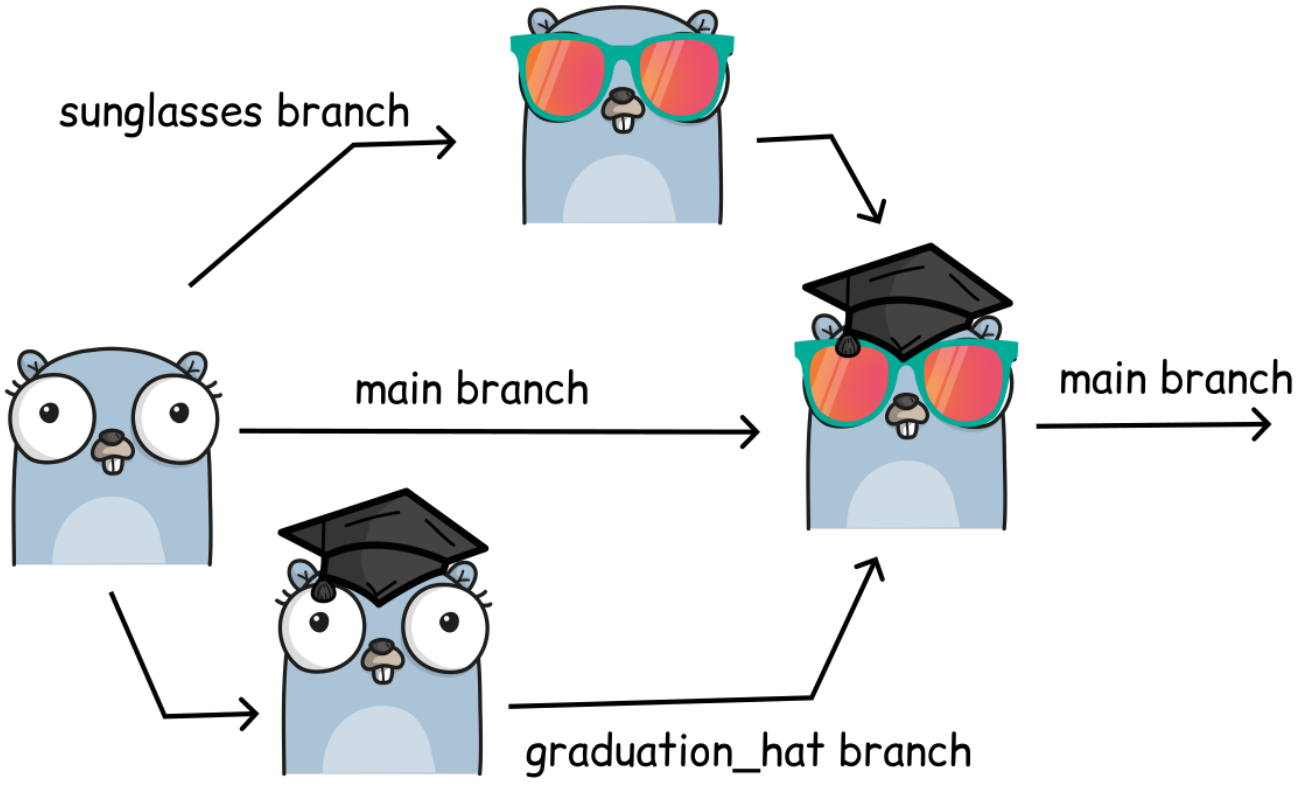
- Record new changes to our own copy of the project.
- Understand adding changes in two separate branches.
- See how to compare different versions.

Glossary

- **commit**: Snapshot of the project at a certain point in time, gets a unique identifier (called a **hash**, e.g. `c7f0e8bfc718be04525847fc7ac237f470add76e`). Usually you can be lazy and use only the first 4 characters wherever a commit hash is needed.
- **branch**: Independent development line. The main development line is often called `main`.
- **tag**: A pointer to one commit, to be able to refer to it later. Like a "commemorative plaque" that you attach to a particular commit (e.g. `phd-printed` or `paper-submitted`).

- **repository**: A copy of the project, contains all data and history (commits, branches, tags).
- **forge**: *a web-based collaborative software platform for both developing and sharing code* (from wikipedia), e.g. GitHub or GitLab
- **cloning**: Copying the whole repository - the first time, e.g. downloading it on your computer. It is not necessary to download each file one by one.
- **forking**: Cloning a repository (which is typically not yours) on a forge - your copy (fork) stays on the forge and you can make changes to your copy.

Merging



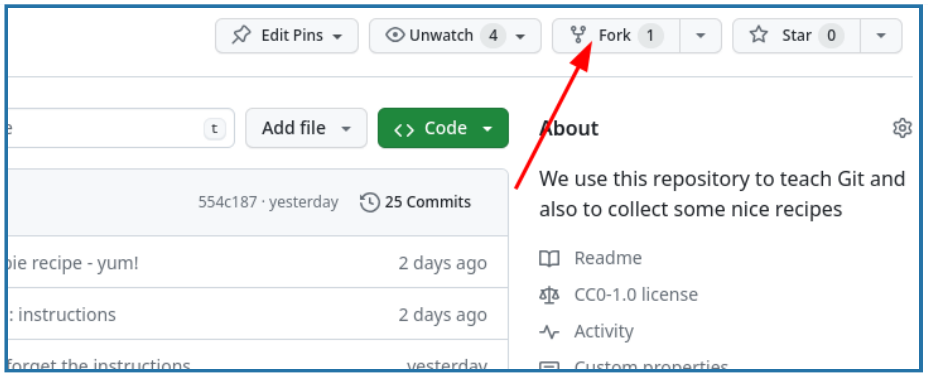
What if two people, at the same time, make two different changes? Git can merge them together easily. Image created using https://gopherize.me/ (inspiration).

Exercise: Practice creating commits and branches

⚙ How to prepare the repository

Fork on github.com | Clone and push to new repository

1. Go to the repository view on GitHub https://github.com/coderefinery/recipe-book
2. First, on GitHub, click the button that says “Fork”. It is towards the top-right of the screen:



3. You should shortly be redirected to your copy of the repository **YOUR_USER_NAME/recipe-book**.

At all times you should be aware of if you looking at *your* repository or the *CodeRefinery* ***upstream*** repository.

- Your repository: https://github.com/**USERNAME**/recipe-book
- CodeRefinery upstream repository: https://github.com/**coderefinery**/recipe-book

We offer **three different paths** of how to do this exercise:

- on GitHub
- using VSCode
- using the command line

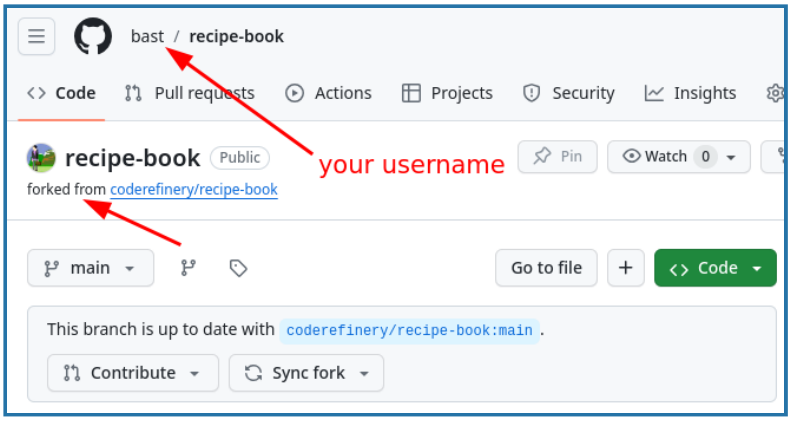
✍️ Exercise: Practice creating commits and branches (20 min)

1. Make sure that you now work **on your fork** of the recipe-book repository (`USER/recipe-book`, *not* `coderefinery/recipe-book`)
2. First create a new branch and then add a recipe to the branch and commit the change.
3. In a new commit, modify the recipe you just added.
4. Switch to the `main` branch and modify a recipe there.
5. Browse the network and locate the commits that you just created ("Insights" -> "Network").
6. Compare the branch that you created with the `main` branch. Can you find an easy way to see the differences?
7. Can you find a way to compare versions between two arbitrary commits in the repository?
8. Try to rename the branch that you created and then browse the network again.
9. Try to create a tag for one of the commits that you created (on GitHub, create a "release").

The solution below goes over most of the answers, and you are encouraged to use it when the hints aren't enough - this is by design.

Solution and walk-through

(1) Make sure you are on your fork



You want to see your username in the URL and you want to see the "forked from ..." part.

(2) Create a branch and add a recipe to the branch

A recipe template is below. This format is called "Markdown", but it doesn't matter right now. You don't have to use this particular template.

```
# Recipe name

## Ingredients

- Ingredient 1
- Ingredient 2

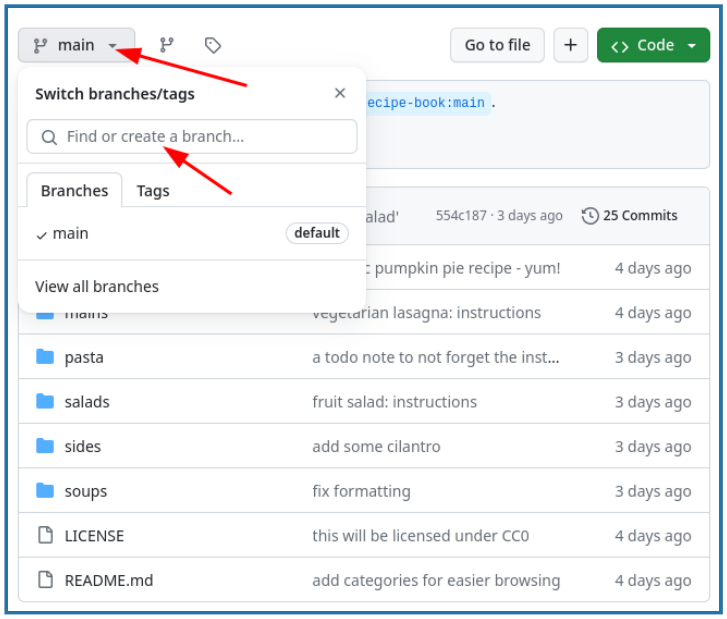

## Instructions

- Step 1
- Step 2
```

There is a **main** branch that is default. We want to create a *different* **branch** for our new commit, because we will *merge* it later. **Commit** is the verb to describe recording more changes, and also the name of the thing you make. A commit is identified by something such as `554c187`.

GitHub | VS Code | Command line

1. Where it says "main" at the top left, click, enter a new branch name `new-recipe`, click on the offer to create the new branch ("Create branch new-recipe from main").



2. Change to some sub-directory, for example `sides`
3. Make sure you are still on the `new-recipe` branch (it should say it at the top), and click “Add file” → “Create new file” from the upper right.
4. Enter a filename where it says “Name your file…”, with a `.md` at the end. Example: `mixed-nuts.md`.
5. Enter the recipe. You can use the template above.
6. Click “Commit changes”
7. Enter a commit message. Then click “Commit changes”.

You should appear back at the file browser view, and see your new recipe there.

(3) Modify the recipe with a new commit

GitHub | VS Code | Command line

This is similar to before, but we click on the existing file to modify.

1. Click on your new recipe, for example `mixed-nuts.md`.
2. Click the edit button, the pencil icon at top-right.
3. Follow the “Commit changes” instructions as in the previous step.

(4) Switch to the main branch and modify a recipe there

GitHub | VS Code | Command line

1. Go back to the main repository page (your user’s page).
2. In the branch switch view (top left above the file view), switch to `main`.

3. Modify another recipe that already exists, following the pattern from above. Don't modify the one you just created (but it shouldn't even be visible on the `main` branch).

(5) Browse the commits you just made

Let's look at what we did. Now, the `main` and `new-recipe` branches have diverged: both have some modifications. Try to find the commits you created.

GitHub | VS Code | Command line

Insights tab → Network view (just like we have done before).

(6) Compare the branches

Comparing changes is an important thing we need to do. When using the GitHub view only, this may not be so common, but we'll show it so that it makes sense later on.

GitHub | VS Code | Command line

Next to the branch name switcher, click on "Branches" to get an overview.

Another way to compare branches or commits on GitHub is to adjust the following URL: `https://github.com/USER/recipe-book/compare/VERSION1..VERSION2`

Replace `USER` with your username and `VERSION1` and `VERSION2` with a commit hash or branch name. Please try it out.

(7) Compare two arbitrary commits

This is similar to above, but not only between branches.

GitHub | VS Code | Command line

Like above, one can compare commits on GitHub by adjusting the following URL: `https://github.com/USER/recipe-book/compare/VERSION1..VERSION2`

Replace `USER` with your username and `VERSION1` and `VERSION2` with a commit hash or branch name. Please try it out.

(8) Renaming a branch

GitHub | VS Code | Command line

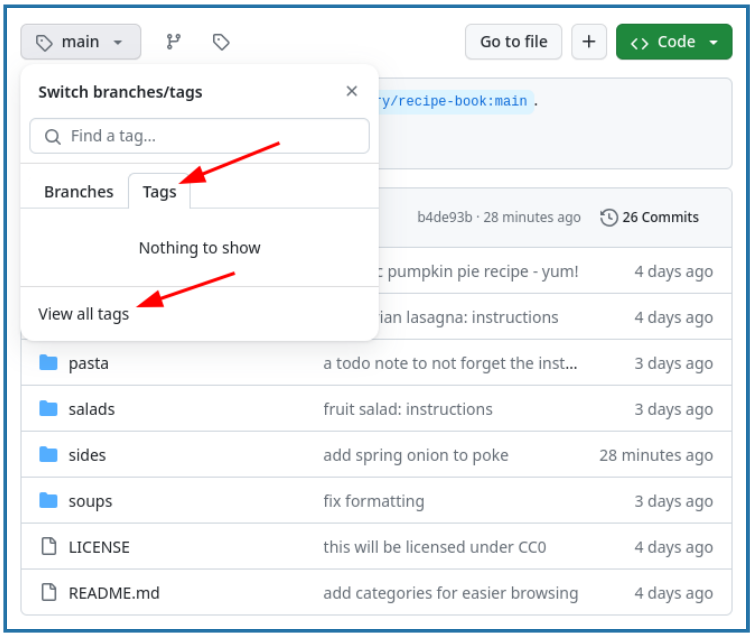
Branch button → View all branches → three dots at right side → Rename branch.

(9) Creating a tag

Tags are a way to mark a specific commit as important, for example a release version. They are also like a sticky note, but they don't move when new commits are added.

GitHub | VS Code | Command line

Click on the branch switcher, and then on "Tags", then on "View all tags", then "Create a new release":



What GitHub calls releases are actually tags in Git with additional metadata. For the purpose of this exercise we can use them interchangeably.

Discussion

In this part, we saw how we can make changes to our files. With branches, we can track several lines of work at once, and can compare their differences.

- You could commit directly to `main` if there is only one single line of work and it's only you.
- You could commit to branches if there are multiple lines of work at once, and you don't want them to interfere with each other.
- Tags are useful to mark a specific commit as important, for example a release version.

Beyond add and commit: undoing mistakes

Most git users will typically use `pull`, `add`, `commit` and `push` and these 4 commands will perform >95% of the operations needed.

To be in control of Git, it is beneficial to know at least how to undo these commands.

Objectives

- Undo `git add`
- Undo `git commit`
- Understand that there are many states, and the complexity that this entails

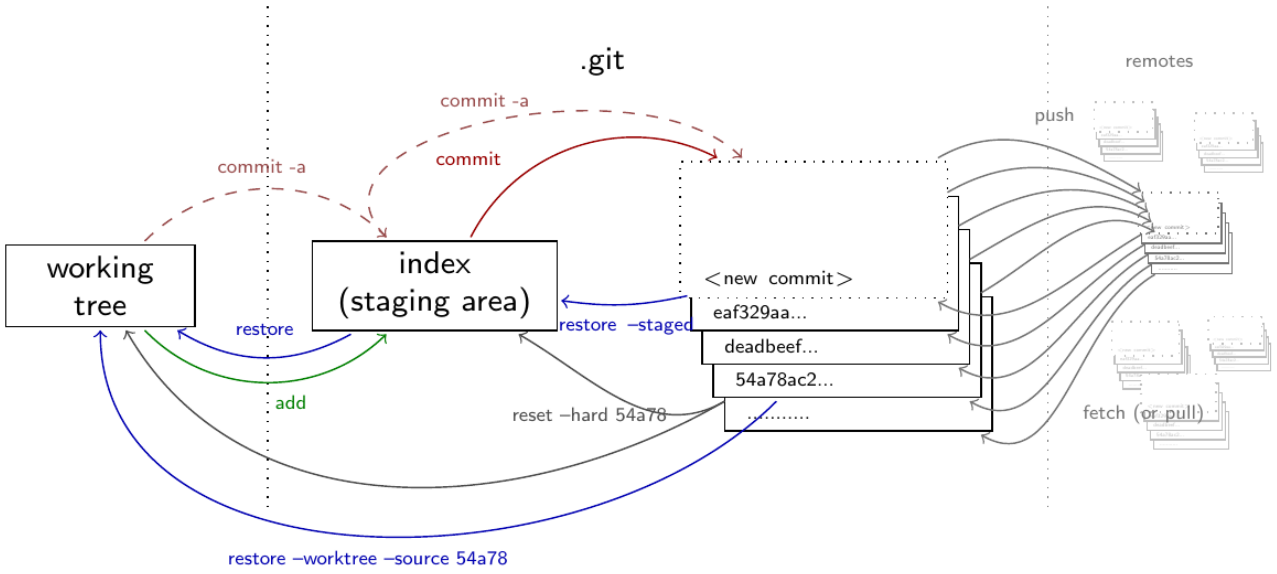
Git offers you **many versions** of any file in the repository:

- the one in the **working tree**
- the one in the **index** (the index is also called "staging area")
- as many versions as there are commits.

The committed versions cannot be changed, but new commits can be created, and the versions in the **working tree** and in the **index** can be overwritten.

As a result, there can be many commands that are used to copy one version of a file into another.

The 3 kinds of states of a file in Git



The different versions of a file in Git and the commands that can be used to copy them into each other.

The 3 Kinds of State of a file in Git - Table version

from \ to	Working Tree	Index	HEAD
Working Tree		commit -a **add**	commit -a
Index	**restore**		**commit**
`<commit>`	checkout reset –hard **restore –source <...>**	checkout reset –hard **reset (–mixed)** **restore (–source <...>) –staged**	checkout reset –hard reset (–mixed) **reset –soft**

Undoing `git add`

Undoing `git add <path>` typically means either to remove the file from the index completely, but leaving it in the working tree or to recover the previous state of a file in the index . This might not be always possible, but in most cases the previous state is the one in the last commit (`HEAD`).

Undoing `git add`

Problem	Solution
File should not be tracked at all	`git rm --cached <filename>`
Changes to file should not go in the next commit	`git restore --staged <filename>`

Undoing `git commit`

The `commit` command cannot actually be undone *completely*, since it created another immutable **object** in the git repository.

Practically, there are some different effects of `git commit` that we might want to undo:

Undoing `git commit`

Problem	Solution	Comments
Change to the files in the repository	`git revert <commit>`	Creates another commit with the opposite changes
Forgot to add a file before committing	`git commit --amend --no-edit`	You can add the file, and then run this command. **Don't do it after** `git push`
Wrong commit message	`git commit --amend`	You can change the commit message. **Don't do it after** `git push`
Branch has moved to new commit	`git reset --soft <last-good-commit>`	Moves the current branch to the chosen commit. **Don't do it after** `git push`

Avoid rewriting published history

If you have already published your changes to a branch with `git push` and someone has already seen them (and perhaps started working on them) using `git reset` or `git commit --amend` could be considered **very rude**!

Why?

With those commands you do, in other words, *rewrite the history* of the branch. This means that the tracking information of the branches in the other repositories might be inconsistent.

If someone works on a history that has been later rewritten, it might result in introduction of undesired modifications to the repository. Moreover, it might be hard to spot that such changes occurred.

In that case, better to use

Patching: Partial commands

Most of the commands listed above accept a `--patch` (or `-p`) option that allows to interactively select the parts (*hunks*) of a file that will be copied, very useful when some additional finesse is required.

Take-home messages

- To undo `git add`, typically you need to copy the content of `HEAD` into the index. `git restore --staged <file-path>` or `git reset` will do it.
- To undo a commit, typically you want to use `git revert`. If you have not used `git push` yet, you have fancier options.
- A convenient guide to get out of unpleasant situations can be found here.
- An alternative explanation of many useful "life-saving" commands is available here

Inspecting history

Objectives

- Be able find a line of code, find out why it was introduced and when.
- Be able to quickly find the commit that changed a behavior.

Instructor note

- 30 min teaching/type-along
- 20 min exercise

Command line, GitHub, and VS Code

As usual, we offer ways to do this with the command line, VS Code, and GitHub.

- **Command line** is most powerful and relatively easy with this. You may also use it along with other things. If you haven't tried it yet, we'd recommend you to give it a try.
- The **GitHub** web interface allows many things to be done, but not everything.
- **VS Code** allows some of these, but for some it's easier to open the VS Code terminal and run git there.

Our toolbox for history inspection

Instructor note

First the instructor demonstrates few commands on a real life example repository https://github.com/networkx/networkx (mentioned in the amazing site The Programming Historian). Later we will practice these in an archaeology exercise (below).

Warm-up: "Git History" browser

As a warm-up we can try the "Git History" browser on the README.rst file of the networkx repository:

- Visit and browse https://github.githistory.xyz/networkx/networkx/blob/main/README.rst (use left/right keys).
- You can try this on some of your GitHub repositories, too!

Searching text patterns in the repository

With `git grep` you can find all lines in a repository which contain some string or regular expression. This is useful to find out where in the code some variable is used or some error message printed.

Command line | GitHub | VS Code | RStudio

The Git command is as described above:

```
$ git grep TEXT
$ git grep "some text with spaces"
```

In the networkx repository you can try:

```
$ git clone https://github.com/networkx/networkx
$ cd networkx
$ git grep -i fixme
```

While `git grep` searches the **current state** of the repository, it is also possible to search through all changes with `git log -S sometext` which can be useful to find where something got removed.

Inspecting individual commits

Command line | GitHub | VS Code | RStudio

We have seen this one before already. Using `git show` we can inspect an individual commit if we know its hash:

```
$ git show HASH
```

For instance:

```
$ git show 759d589bdfa61aff99e0535938f14f67b01c83f7
```

Line-by-line code annotation with metadata

With `git annotate` you can see line by line who and **when** the line was modified last. It also prints the precise hash of the last change which modified each line. Incredibly useful for reproducibility.

Command line | GitHub | VS Code | RStudio

```
$ git annotate FILE
```

Example:

```
$ git annotate networkx/convert_matrix.py
```

If you annotate in a terminal and the file is longer than the screen, Git by default uses the program `less` to scroll the output. Use `/sometext` `<ENTER>` to find "sometext" and you can cycle through the results with `n` (next) and `N` (last). You can also use page up/down to scroll. You can quit with `q`.

💬 Discussion

Discuss how these relatively trivial changes affect the annotation:

- Wrapping long lines of text/code into shorter lines
- Auto-formatting tools such as `black`
- Editors that automatically remove trailing whitespace

Inspecting code in the past

Command line | GitHub | VS Code | RStudio

We can create branches pointing to a commit in the past. This is the recommended mechanism to inspect old code:

```
$ git switch --create BRANCHNAME HASH
```

Example (lines starting with “#” are only comments):

```
$ # create branch called "older-code" from hash 347e6292419b
$ git switch --create older-code 347e6292419bd0e4bff077fe971f983932d7a0e9

$ # now you can navigate and inspect the code as it was back then
$ # ...

$ # after we are done we can switch back to "main"
$ git switch main

$ # if we like we can delete the "older-code" branch
$ git branch -d older-code
```

On old Git versions which do not know the `switch` command (before 2.23), you need to use this instead:

```
$ git checkout -b BRANCHNAME SOMEHASH
```

Exercise

This is described with the command line method, but by looking above you can translate to the other options.

✍ Exercise: Explore basic archaeology commands (20 min)

Let us explore the value of these commands in an exercise. Future exercises do not depend on this, so it is OK if you do not complete it fully.

Exercise steps:

- **Make sure you are not inside another Git repository** when running this exercise. If you are, first step “outside” of it. We want to avoid creating a Git repository inside another Git repository.

  Command line | GitHub | VS Code | RStudio

  You can check if you are inside a Git repository with:

```
$ git status

fatal: not a git repository (or any of the parent directories): .git
```

You want to see the above message which tells us that this is not a Git repository.

- Clone this repository: https://github.com/networkx/networkx.git.

  Command line | GitHub | VS Code | RStudio

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

- Then let us all **make sure we are working on a well-defined version** of the repository.

  Command line | GitHub | VS Code | RStudio

  Step into the new directory and create an exercise branch from the networkx-2.6.3 tag/release:

  ```
  $ cd networkx
  $ git switch --create exercise networkx-2.6.3
  ```

  On old Git versions which do not know the `switch` command (before 2.23), you need to use this instead:

  ```
  $ git checkout -b exercise networkx-2.6.3
  ```

Then using the above toolbox try to:

1. Find the code line which contains `"Logic error in degree_correlation"`.
2. Find out when this line was last modified or added. Find the actual commit which modified that line.
3. Inspect that commit with `git show`.
4. Create a branch pointing to the past when that commit was created to be able to browse and use the code as it was back then.

5. How would you bring the code to the version of the code right before that line was last modified?

✓ Solution

We provide here a solution for the command line but we also encourage you to try to solve this in the browser.

1. We use `git grep`:

   ```
   $ git grep "Logic error in degree_correlation"
   ```

   This gives the output:

   ```
   networkx/algorithms/threshold.py:                print("Logic error in
   degree_correlation", i, rdi)
   ```

   Maybe you also want to know the line number:

   ```
   $ git grep -n "Logic error in degree_correlation"
   ```

2. We use `git annotate`:

   ```
   $ git annotate networkx/algorithms/threshold.py
   ```

   Then search for “Logic error” by typing “/Logic error” followed by Enter. The last commit that modified it was `90544b4fa` (unless that line changed since).

3. We use `git show`:

   ```
   $ git show 90544b4fa
   ```

4. Create a branch pointing to that commit (here we called the branch “past-code”):

   ```
   $ git branch past-code 90544b4fa
   ```

5. This is a compact way to access the first parent of `90544b4fa` (here we called the branch “just-before”):

   ```
   $ git switch --create just-before 90544b4fa~1
   ```

Finding out when something broke/changed with `git bisect`

This only works with the command line.

> *"But I am sure it used to work! Strange."*

Sometimes you realize that something broke. You know that it used to work. You do not know when it broke.

💬 How would you solve this?

Before we go on first discuss how you would solve this problem: You know that it worked 500 commits ago but it does not work now.

- How would you find the commit which changed it?
- Why could it be useful to know the commit that changed it?

We will probably arrive at a solution which is similar to `git bisect`:

- First find out a commit in past when it worked.

```
$ git bisect start
$ git bisect good f0ea950  # this is a commit that worked
$ git bisect bad main      # last commit is broken
```

- Now compile and/or run and/or test and decide whether "good" or "bad".
- This is how you can tell Git that this was a working commit:

```
$ git bisect good
```

- And this is how you can tell Git that this was not a working commit:

```
$ git bisect bad
```

- Then bisect/iterate your way until you find the commit that broke it.
- If you want to go back to start, type `git bisect reset`.

- This can even be automatized with `git bisect run SCRIPT`. For this you write a script that returns zero/non-zero (success/failure).

Optional exercise: Git bisect

This only works with the command line.

✍ (optional) History-2: Use git bisect to find the bad commit

In this exercise, we use `git bisect` on an example repository. It is OK if you do not complete this exercise fully.

Begin by cloning https://github.com/coderefinery/git-bisect-exercise.

Motivation

The motivation for this exercise is to be able to do archaeology with Git on a source code where the bug is difficult to see visually. **Finding the offending commit is often more than half the debugging**.

Background

The script `get_pi.py` approximates pi using terms of the Nilakantha series. It should produce 3.14 but it does not. The script broke at some point and produces 3.57 using the last commit:

```
$ python get_pi.py

3.57
```

At some point within the 500 first commits, an error was introduced. The only thing we know is that the first commit worked correctly.

Your task

- Clone this repository and use `git bisect` to find the commit which broke the computation (solution - spoiler alert!).
- Once you have found the offending commit, also practice navigating to the last good commit.
- Bonus exercise: Write a script that checks for a correct result and use `git bisect run` to find the offending commit automatically (solution - spoiler alert!).

Hints

Finding the first commit:

```
$ git log --oneline | tail -n 1
```

How to navigate to the parent of a commit with hash SOMEHASH:

```
$ git switch --create BRANCHNAME SOMEHASH~1
```

Instead of a tilde you can also use this:

```
$ git switch --create BRANCHNAME SOMEHASH^
```

Summary

- git log/grep/annotate/show/bisect is a powerful combination when doing archaeology in a project on the command line.
- `git switch --create NAME HASH` is the recommended mechanism to inspect old code on the command line.
- Most of these commands can be used in the GitHub web interface (except `git bisect`).

Optional: Git Internals

❗ Objectives

- Understand what is easy to do with git, and what is not easy

Instructor note

- 15 min teaching/type-along
- 15 min exercise

Down the rabbit hole

When usually working with Git, **you will never need to go inside .git**, but in this exercise we will in order to learn about

- how branches are implemented in Git, and how to use them freely
- how you can avoid losing data with Git.

⚙ Prerequisites

For this exercise create a new repository and commit a couple of changes. You can also clone this repository:

```
$ git clone https://github.com/mmesiti/merge-fu.git
```

Now that we've made a couple of commits let us look at what is happening under the hood.

```
$ cd .git
$ ls -l

drwxr-xr-x   - user 25 Aug 15:51 branches
.rw-r--r-- 499 user 25 Aug 15:52 COMMIT_EDITMSG
.rw-r--r--  92 user 25 Aug 15:51 config
.rw-r--r--  73 user 25 Aug 15:51 description
.rw-r--r--  21 user 25 Aug 15:51 HEAD
drwxr-xr-x   - user 25 Aug 15:51 hooks
.rw-r--r-- 137 user 25 Aug 15:52 index
drwxr-xr-x   - user 25 Aug 15:51 info
drwxr-xr-x   - user 25 Aug 15:52 logs
drwxr-xr-x   - user 25 Aug 15:52 objects
drwxr-xr-x   - user 25 Aug 15:51 refs
```

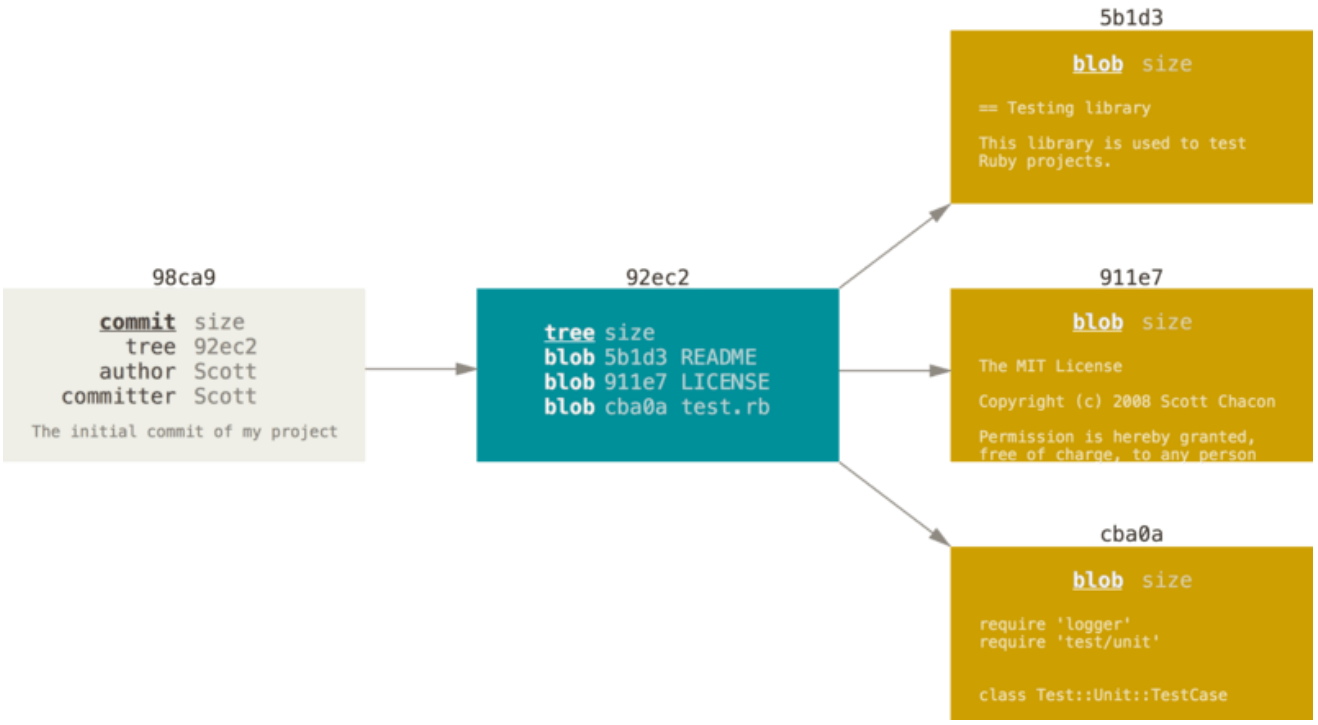
Git stores everything under the .git folder in your repository.
We will have a look at the `objects` and the `refs` directories.

In the **objects** directory we find, among others, 3 kinds of objects:

- `commit`s: These represent the commits we have made with `git commit`
- `blob`s: These represent snapshots of all the files we have ever added to the repo with `git add`.
- `tree`s: These represent directories containing the files we have added, and reference other `tree`s (subdirectories) and `blob`s (files that we have added).

`commit` objects contain information about the author and the commit message, and every `commit` object references a single `tree` object.

All objects are named as the SHA-1 hash (a 40-character hexadecimal string) that is computed on their content.
This means that all objects are *immutable*.



States of a Git repository. Image from the Pro Git book. License CC BY 3.0.

✍ Changes and their effect: files and commits

Refer to the figure above, and discuss: which SHA-1 hashes would change in the diagram if:

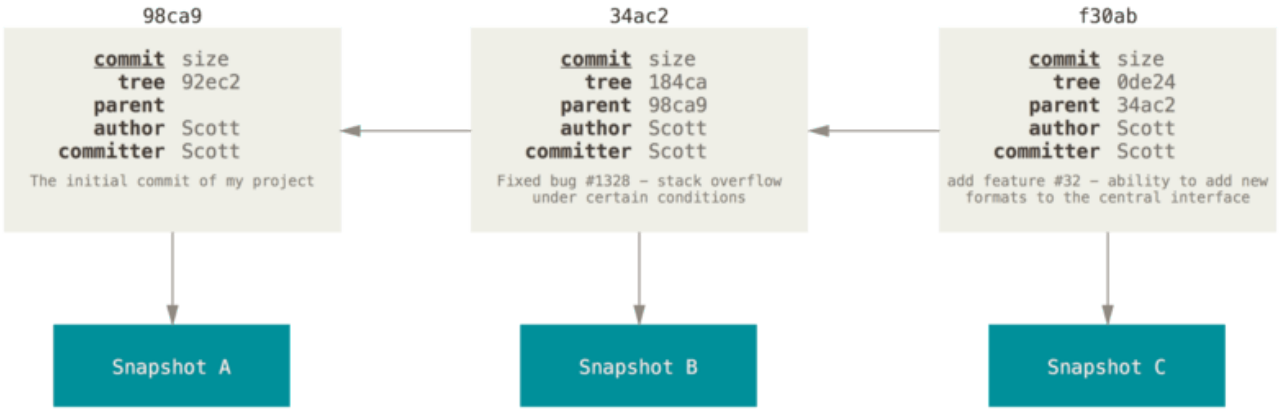
- the content of the first file is changed,
- we recreate a commit with another message or author
- we recreate a commit with the same message or author

Is it possibe to have multiple commits refer to the same tree? What happens when you use `git revert`?

✔ Solution

When reverting a commit B that happens after a commit A, the new commit will point at the same tree as A.

Once you have several commits, each commit object also links to the hash of the previous commit(s) (there is more than one previous commit for for merge commits). The commits form a directed acyclic graph (do not worry if the term is not familiar).



A commit and its parents. Image from the Pro Git book. License CC BY 3.0.

💬 Changes and their effect: changing history

Refer to the figure above, and discuss: which SHA-1 hashes would change in the diagram if:

- The the 3rd commit were changed
- The 2nd commit were changed

Git is at its core a content-addressed storage system

- CAS: "mechanism for storing information that can be retrieved based on its content, not its storage location"
- Content address is the content digest (SHA-1 checksum)
- Stored data does not change, so when we modify commits or add new version of the files, we always create new objects. Git doesn't delete the old objects right away, which is why it is *very hard to lose data if you commit it once*.

✍️ A look at the objects

Let us poke a bit into raw objects! Start with:

```
$ git cat-file -p HEAD
```

Then explore the `tree` object, then the `file` object, etc. recursively using the hashes you see.

Demo: If you add it, you don't lose it (for a while)

A common way to (apparently) lose work is to use `git add` indiscriminately.

You make some changes to a file, (let us call this version A) you `git add` them, then you make some other changes (let us call this version B) and you `git add` those again.

Now version A is apparently lost, and if we realize that we need it back we typically click nervously on the “undo” arrow of our editor.

But fear not! Try this.

1. Create a file named `test-add` with the following command:

```
echo 'Once a file has been git added, it is hard to lose!' > test-add
```

2. Add it to the repository

```
$ git add test-add
```

3. Now change the content of the file to be

```
Ops
```

4. And repeat the add command

```
$ git add test-add
```

5. Apparently we have lost the previous version of the file. But it is actually there, stored in a *dangling* blob object (which is not referenced, even indirectly, by any `ref`) We can see this with the command `fsck`:

```
$ git fsck
Checking object directories: 100% (256/256), done.
dangling blob dc3b15f60045eea7a87639436ed75021130579e0
```

   We can see the content of that blob by passing its hash (shortened for convenience) to the `git cat-file -p` command:

```
$ git cat-file -p dc3b
Once a file has been git added, it is hard to lose!
```

Deletion of dangling objects is done by a *garbage collector* that might be triggered automatically by some commands.

💬 Discussion

Discuss the findings with other course participants.

Operating on Branches

A **branch** represents independent line of work. Git internal structure makes it very easy to implement branches as a thin layer of abstraction on its *Object database*.

Objectives

- Understand that branches are just pointers to commits
- Manipulate local branches

In this exercise we will look inside the `.git` directory in order to learn about how branches are implemented in Git, and how to use them freely.

Demonstration: experimenting with branches

Branches are pointers to commits that move over time.

We are starting from the `main` branch and create an `idea` branch:

```
$ git status

On branch main
nothing to commit, working tree clean
$ git branch idea
$ git switch idea
Switched to a new branch 'idea'
```

(Creating a branch and switching to it can be done in a single command with `git switch --create <branch-name>`)

```
$ git branch

* idea
  main
```

Let us lift the hood and create few branches “manually”, without using the `git branch` command.

Let us have a look at the `refs` directory:

```
$ ls -l .git/refs/heads

.rw-r--r-- 41 user 25 Aug 15:54 idea
.rw-r--r-- 41 user 25 Aug 15:52 main
```

Let us check what the `idea` file looks like (do not worry if the hash is different):

```
$ cat .git/ref/heads/idea

045e3db14740c60684d745e5fb891ae71e335611
```

Now let us replicate this file:

```
$ cp .git/refs/heads/idea .git/refs/heads/idea-2
$ cp .git/refs/heads/idea .git/refs/heads/idea-3
```

Let us go up two levels and inspect the file `HEAD`:

```
$ cat .git/HEAD

ref: refs/heads/idea
```

Let us open this file and change it to:

```
ref: refs/heads/idea-3
```

Now - on which branch are we?

```
$ git branch

  idea
  idea-2
* idea-3
  main
```

✍ Exercise

By changing the content of `.git/HEAD` we have manually "switched" from a branch to another one that actually points to the same commit.

What would have happened if we changed HEAD to point to a branch that does not point to the same commit as the one we were on before? What would we see with `git status`?

✍️ Branches on different repositories

How are branches on different repositories related to each other?

✔ Solution

After creating a branch, one can use the `--set-upstream-to` options

```
$ git branch <new-branch>
$ git branch <new-branch> --set-upstream-to=<remote>/<branch>
```

to set the default *upstream* branch.

When pushing, it is possible to use the verbose command:

```
$ git push <repository> <local-branch>:<remote-branch>
```

Typically `<local-branch>` and `<remote-branch>` are the same, and `:<remote-branch>` is omitted it is assumed to be equal to `<local-branch>`.

`git push` can also use the default upstream branch if configured correctly:

```
$ git config --local push.default upstream
```

But typically there is no need for such complex setups.

Deleting branches (also by mistake - and undoing it)

Let us add some work on the branch `idea-3`, and create some additional commits.

Let's assume we want to remove the branch `idea-2`, to tidy up our repository.

We first switch to `main`, then try to remove the useless branch

```
$ git switch main
$ git branch -d idea-3
error: The branch 'idea-3' is not fully merged.
```

We are sure we want to delete, so we use the `-D` option.

```
$ git branch -D idea-3
```

We then get distracted and go doing something else.

```
$ clear
```

Wait a moment! We deleted the wrong branch. Is our work lost? Using

```
$ git reflog
```

we can see all the last commits pointed at by HEAD, and among them there will be the one which was referenced by `idea-3` before we deleted it. We can check it out and recreate our branch.

Moving branches back to where they pointed

When using many commands, we move forward the branch we are on.

We can make a branch point back to where it pointed before by switching to it and using `git reset --soft`.

If we do not exactly remember where it pointed, we can use `git reflog <branch name>` to get an idea of where it was moved.

Visualizing branches efficiently

When working with branches on the command line, it is useful to look at the log with the following command (or something similar):

```
$ git log --oneline --graph --all
```

It is inconvenient to type such a long message every time. Git allows us to configure an alias for it, in this case it will be called `graph`:

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

After this configuration, we will be able to use `graph` as a git command with the same effect as the original, longer command.

Concepts around collaboration

Objectives

- Be able to decide whether to divide work at the branch level or at the repository level.

Instructor note

- 15 min teaching

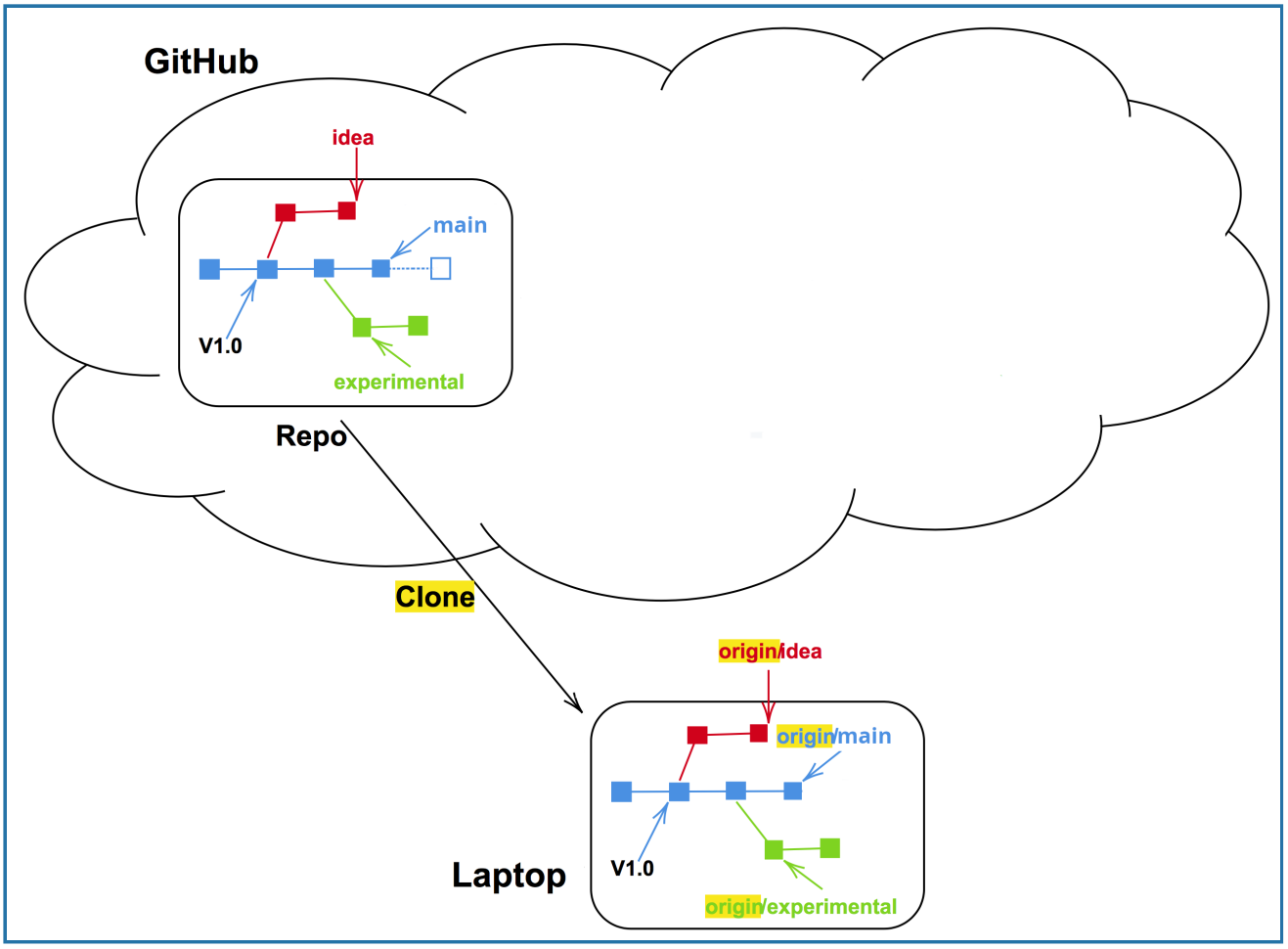
Motivation

- Someone has given you access to a repository online and you want to contribute?
- We will review how to make a copy and send changes back.
- Then, we make a "pull request" that allows a review.
- Once we know how code review works, we will be able to:
  - propose changes to repositories of others
  - review changes submitted by external contributors.

Cloning a repository

In order to make a copy a repository (a **clone**), the `git clone` command can be used. Cloning of a repository is of relevance in a few different situations:

- Working on your own, cloning is the way to copy a repository on, e.g., a personal computer, a server, and a supercomputer.
- The original repository could be a repository that you or your colleague own. A common use case for cloning is when working together within a smaller team where everyone has read and write access to the same git repository.
- Alternatively, cloning can be made from a public repository of a code that you would like to use. Perhaps you have no intention to work on the code, but would like to stay in tune with the latest developments, also in-between releases of new versions of the code.
- Your work is not visible to others, because it is on your computer.

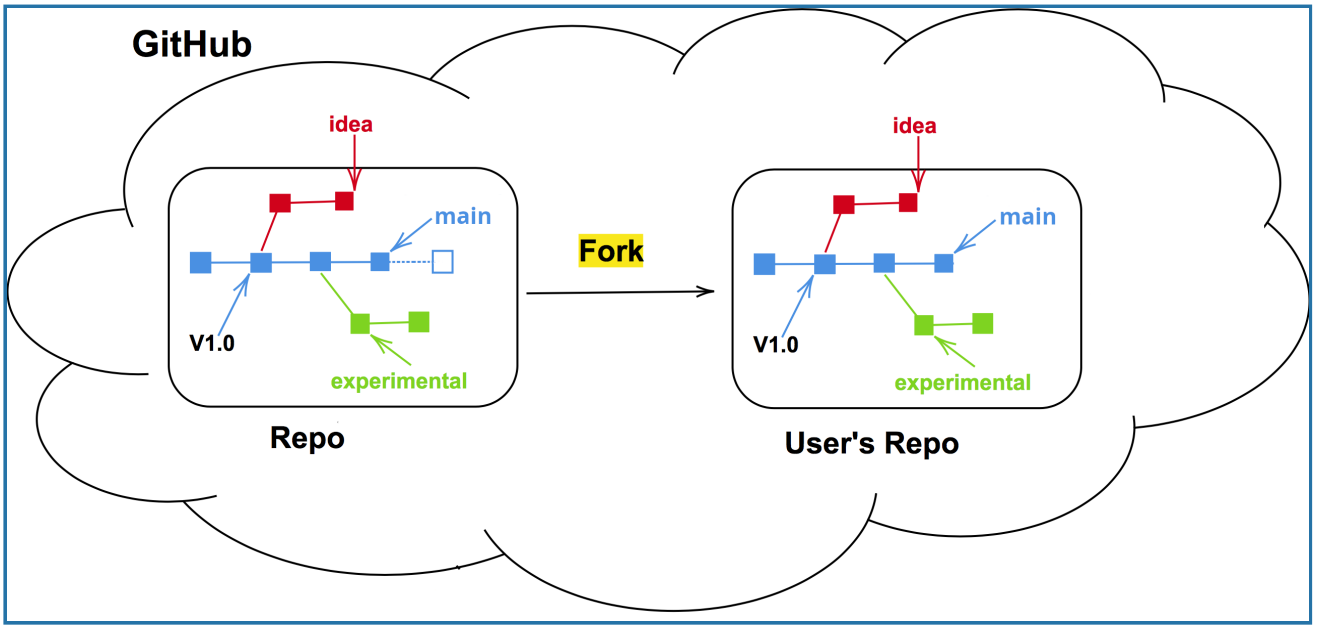


Cloning

Forking a repository

Forking a repository on a forge creates a clone that reside under a different account on the same forge (a **fork**).
It is typically done to work on a git repository you cannot write to.

- Your work is visible to others, because it is on the web
- commits in the fork can be made to any branch (including `main` or `master`)
- The commits that are made within the branches of the fork repository can be contributed back to the parent repository by means of pull (or merge) requests.



✍ Exercise

What is the difference between forking and then cloning (your fork, to your computer) vs cloning (to your computer) and then pushing to a brand new repository?

✔ Solution

1. Forking on a forge and then cloning creates links:
   - from your fork to the original repository;
   - from clone to your fork.
2. When cloning and then pushing to a new repository, you will create links:
   - from your clone to the original repository;
   - from your clone to the new repository.

   Your repository on the forge will not have a link to the original repository and will not be listed as a fork of the original repository.

Generating from templates and importing

There are two more ways to create "copies" of repositories into your user space:

- A repository can be marked as **template** and new repositories can be **generated** from it like using a cookie-cutter. The newly created repository will start with a new history.
- You can **import** a repository from another hosting service or web address. This will preserve the history of the imported project and features like Wikis, issues and the like.

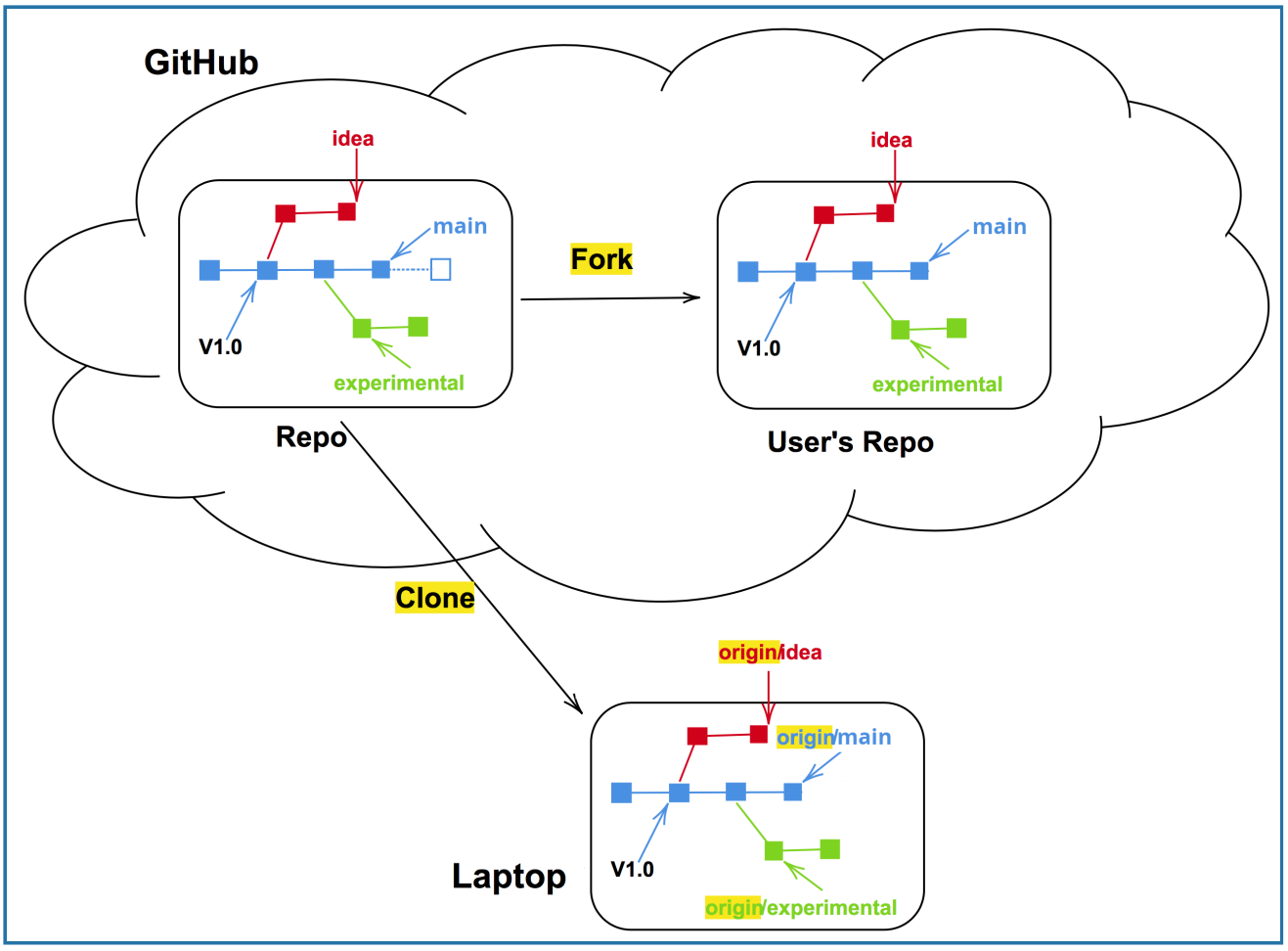
💬 Discussion

- Visit one of the repositories/projects that you have used recently and try to find out how many forks exist and where they are.
- In which situations could it be useful to start from a "template" repository by generating?

Synchronizing changes between repositories

- We need a mechanism to communicate changes between the repositories.
- We will **pull** or **fetch** updates **from** remote repositories (we will soon discuss the difference between pull and fetch).
- We will **push** updates **to** remote repositories.
- We will learn how to suggest changes within repositories on a forge and across repositories (**pull request**).

- Repositories that are forked or cloned do not automatically synchronize themselves: We will learn how to update forks (by pulling from the “central” repository).
- A main difference between cloning a repository and forking a repository is that
  - **cloning** is a general operation for generating copies of a repository to different computers
  - **forking** is a particular operation implemented on **forges** (that includes cloning)



Forking and cloning

Authentication: connecting to the repository from your computer

There are mainly two ways to do authentication:

- SSH keys
- HTTPS

Please have a look at this guide by CodeRefinery for a general introduction to authentication options.

We suggest setting up and using an **SSH key**, since it is a form of authentication that is also used on other services (e.g., to access HPC systems). For a step-by-step guide look at this walkthrough by Software Carpentry.

Authentication via HTTPS might require less set up, if password authentication is allowed. If not, you can use a **personal access token** as a drop-in replacement, which can be configured at these pages:

- gitlab.com
- gitlab.kit.edu

Problems in Collaborative Software development

Merging can be a difficult moment in the life cycle of a software.

Git will try to do *reasonable* operations when merging two different lines of work, but:

- There might be an detectable ambiguity in the way that two different lines of work can be reconciled (this leads to a conflict)
- the results are not guaranteed to give you working software all the times (i.e., you don't get a conflict, but the result is not correct either - *this is scarier*).

Contributing to the main branch as often as possible, to make the changes as small as possible, is a possible approach to reduce the difficulty related to merging.

In the following chapters we will focus on tools that ease the *communication* aspect of collaborative software development.

Collaborating within the same repository: issues and pull requests

In this episode, we will learn how to collaborate within the same repository. We will learn how to cross-reference issues and pull requests, how to review pull requests, and how to use draft pull requests.

This exercise will form a good basis for collaboration that is suitable for most research groups.

Exercise

In this exercise, we will contribute to a repository via a **pull request**. This means that you propose some change, and then it is accepted (or not).

⚙ Exercise preparation

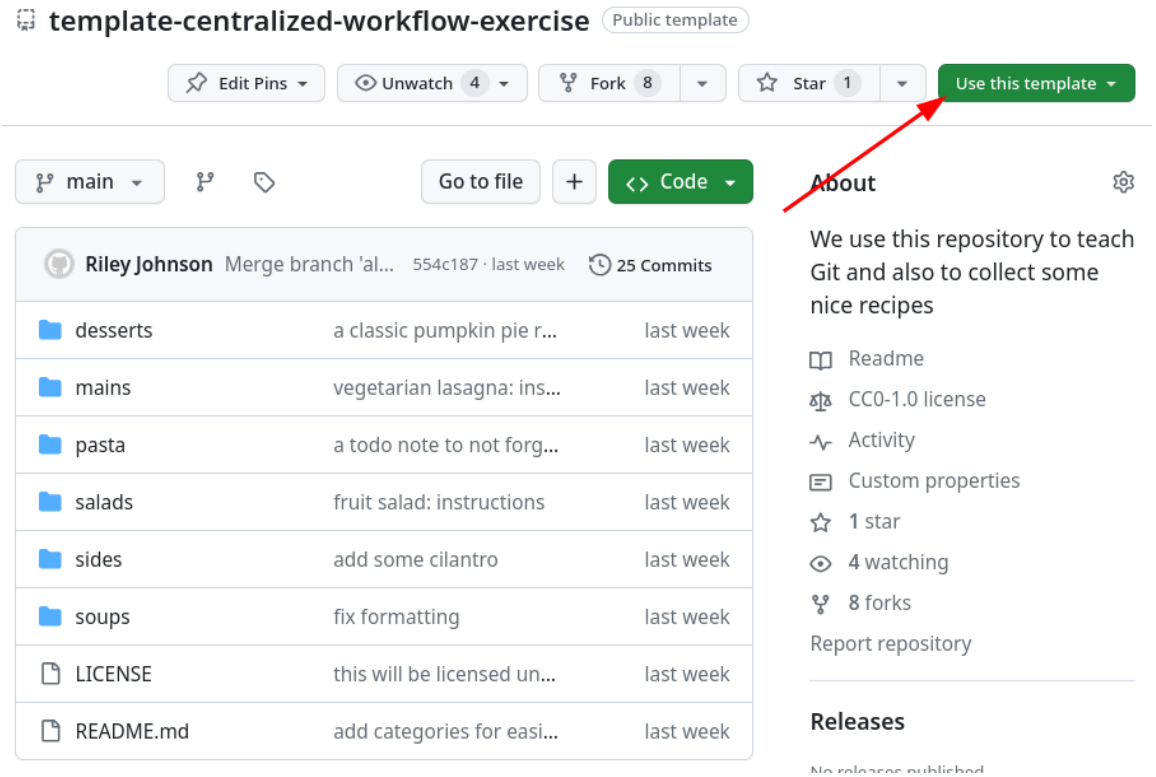
First, we need to get access to some repository to which we will contribute.

1. Form not too large groups (4-5 persons), which have accounts on the same **forge**.
2. Each group needs to appoint someone who will host the shared repository: the *maintainer*. This is typically the exercise lead (if available). Everyone else is a *collaborator*.
3. The **maintainer** (one person per group) generates a new repository called `centralized-workflow-exercise` on the selected **forge**:

⚙ How to prepare the repository

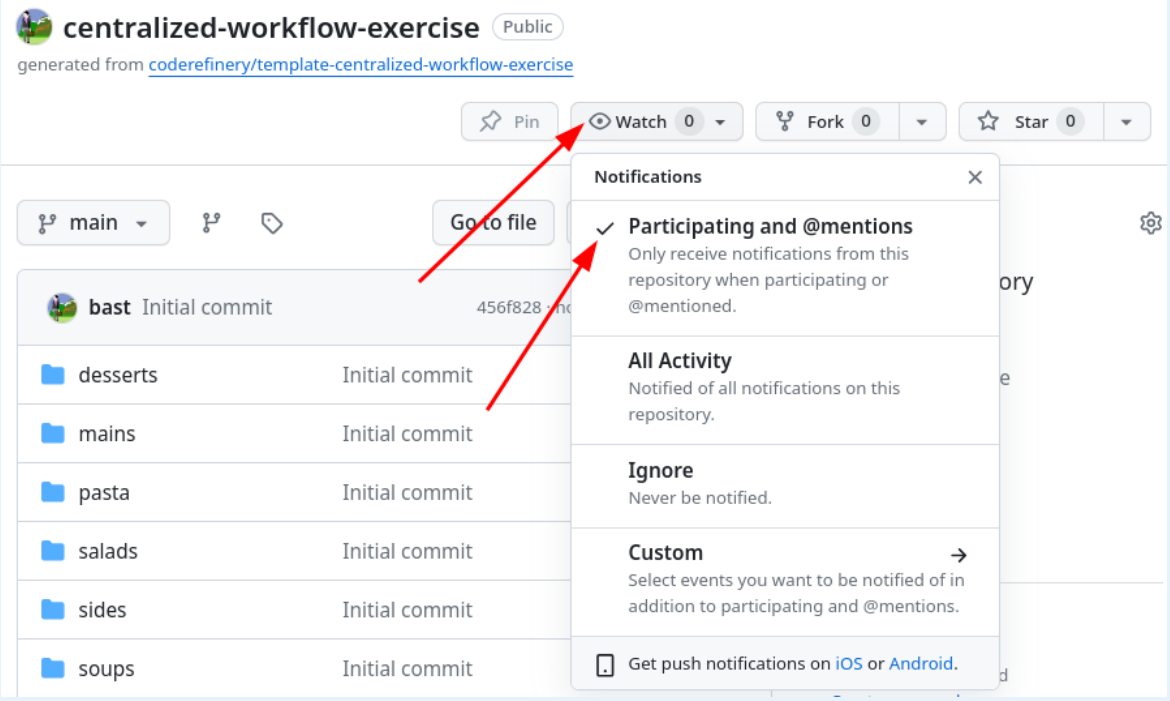
On github.com | On other Forges: Clone and Push to new repository

The repository can be generated from the template https://github.com/coderefinery/template-centralized-workflow-exercise (There is no need to tick "*Include all branches*" for this exercise):



- Then **everyone in your group** needs their account on the forge to be added as collaborator to the exercise repository:
  - Collaborators give their usernames on the forge to their chosen maintainer.
  - Maintainer gives the other group members the newly created repository URL.
  - Maintainer adds participants as collaborators to their project.
    - on github.com: Settings -> Collaborators and teams -> Manage access -> Add people.
    - on GitLab: Manage -> Members -> Invite Members. Choose at least the Maintainer role for this exercise
    - on Codeberg: Settings -> Collaborators
- **Don't forget to accept the invitation**
  - Check your personal area on the forge of choice (look for your notifications)
  - Alternatively check the inbox for the email account you registered with the forge.
    - GitHub emails you an invitation link, but if you don't receive it you can go to your GitHub notifications in the top right corner. The maintainer can also "copy invite link" and share it within the group.

- **Watching and unwatching repositories**
  - Now that you are a collaborator, you get notified about new issues and pull/merge requests via email.
  - If you do not wish this, you can "unwatch" a repository (top of the project page).
  - However, we recommend watching repositories you are interested in. You can learn things from experts just by watching the activity that come through a popular project.



Unwatch a repository by clicking "Unwatch" in the repository view, then "Participating and @mentions" - this way, you will get notifications about your own interactions.

✍ Exercise: Collaborating within the same repository (45 min)

Technical requirements (from installation instructions):

- If you create the commits locally: Being able to authenticate to GitHub

Skills that you will practice:

- Cloning a repository (CodeRefinery lesson)
- Creating a branch (CodeRefinery lesson)
- Committing a change on the new branch (CodeRefinery lesson)
- Submit a pull request towards the main branch (CodeRefinery lesson)
- If you create the changes locally, you will need to **push** them to the remote repository.
- Learning what a protected branch is and how to modify a protected branch: using a pull request.
- Cross-referencing issues and pull requests.
- Practice to review a pull request.

- Learn about the value of draft pull requests.

Exercise tasks:

1. Open an issue where you describe the change you want to make. Note down the issue number since you will need it later.
2. Create a new branch.
3. Make a change to the recipe book on the new branch and in the commit cross-reference the issue you opened (see the walk-through below for how to do that).
4. Push your new branch (with the new commit) to the repository you are working on.
5. Open a pull request towards the main branch.
6. Review somebody else's pull request and give constructive feedback. Merge their pull request.
7. Try to create a new branch with some half-finished work and open a draft pull request. Verify that the draft pull request cannot be merged since it is not meant to be merged yet.

Solution and hints

(1) Opening an issue

This is done through the web interface of your preferred **forge**. For example, you could give the name of the recipe you want to add (so that others don't add the same one).

- On github.com and codeberg.org: Top row -> the "Issues" tab.
- On GitLab: Left side -> Plan -> Issues

(2) Create a new branch.

You have two options:

- make the branch in the web interface (CodeRefinery lesson - refresher, for GitHub: Committing changes)
- If working locally, you need to know how to work locally.

Note: on GitLab, it is possible to create a merge request (and a branch) directly from an issue.

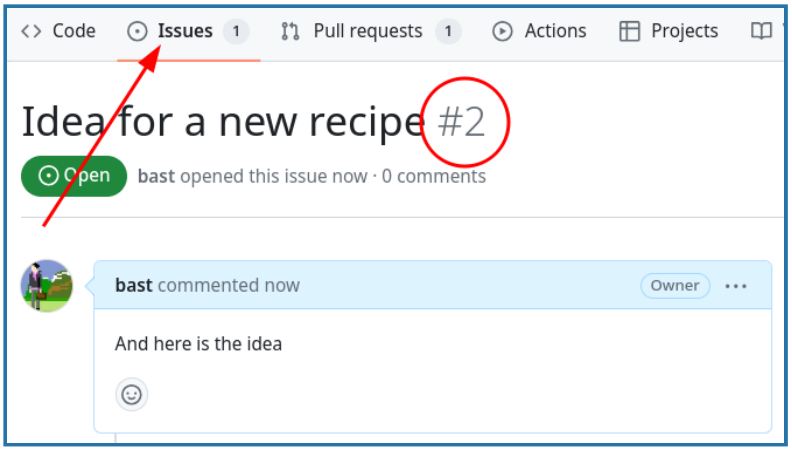
(3) Make a change adding the recipe

Add a new file with the recipe in it. Commit the file. In the commit message, include the note about the issue number, saying that this will close that issue.

Cross-referencing issues and pull requests

Each issue and each pull request gets a number and you can cross-reference them.

When you open an issue, note down the issue number (in this case it is `#2`):



You can reference this issue number in a commit message or in a pull request, like in this commit message:

```
this is the new recipe; fixes #2
```

If you forget to do that in your commit message, you can also reference the issue in the pull request description. And instead of `fixes` you can also use `closes` or `resolves` or `fix` or `close` or `resolve` (case insensitive).

Then observe what happens in the issue once your commit gets merged: it will automatically close the issue and create a link between the issue and the commit. This is very useful for tracking what changes were made in response to which issue and to know from when until when precisely the issue was open.

(4) Push to your forge as a new branch

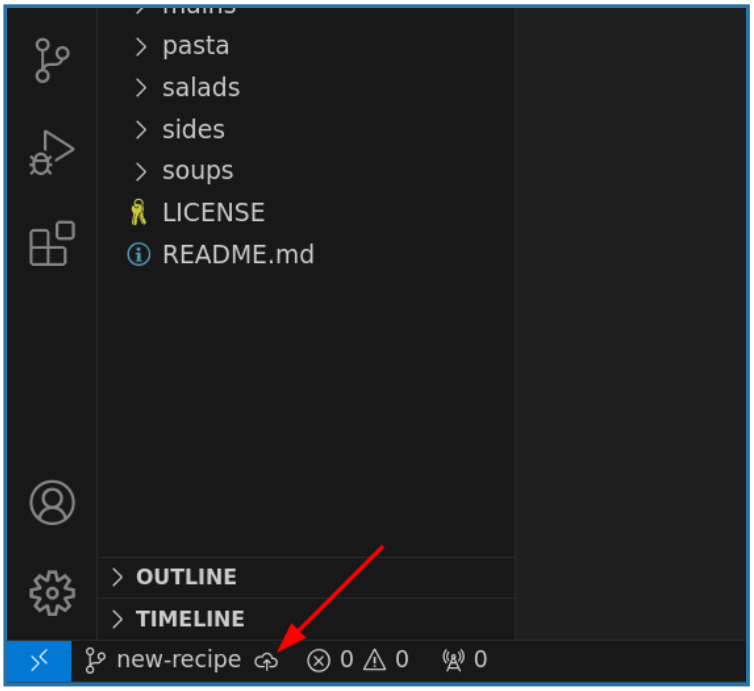
Covered in Cloning a Git repository and working locally.

Push the branch to the repository. You should end up with a branch visible in the web view of your forge.

This is only necessary if you created the changes locally. If you created the changes directly on the web interface of the forge, you can skip this step.

VS Code | Command line

In VS Code, you can “publish the branch” to the remote repository by clicking the cloud icon in the bottom left corner of the window:



If the remote points to the wrong place, you can change it with:

```
$ git remote set-url origin NEWADDRESS
```

(5) Open a pull request towards the main branch

This is done through the GitHub web interface. We saw this in, for example, in a previous lesson.

(6) Reviewing pull requests

You review through the web interface.

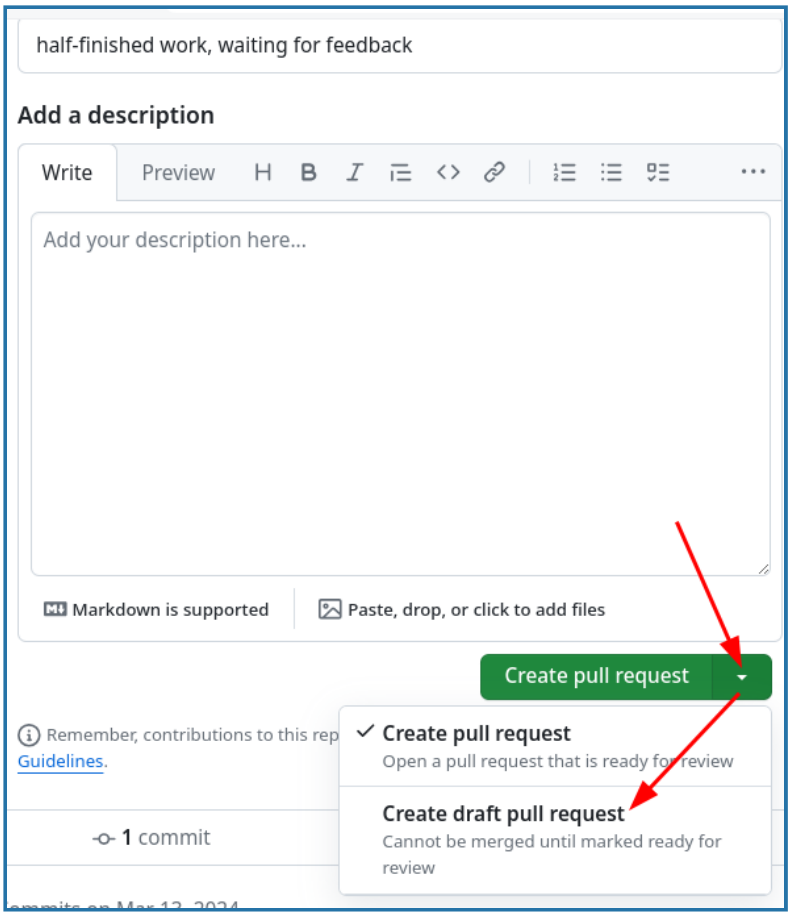
Checklist for reviewing a pull request:

- Be kind, on the other side is a human who has put effort into this.
- Be constructive: if you see a problem, suggest a solution.
- Towards which branch is this directed?
- Is the title descriptive?
- Is the description informative?
- Scroll down to see commits.
- Scroll down to see the changes.
- If you get incredibly many changes, also consider the license or copyright and ask where all that code is coming from.
- Again, be kind and constructive.
- Later we will learn how to suggest changes directly in the pull request.

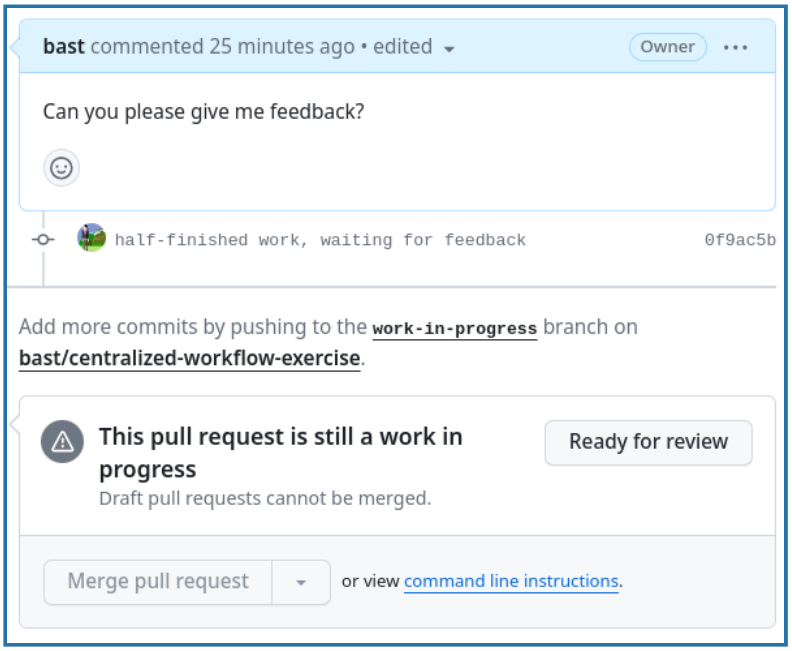
If someone is new, it's often nice to say something encouraging in the comments before merging (even if it's just "thanks"). If all is good and there's not much else to say, you could merge directly.

(7) Draft/WIP pull requests

Try to create a draft pull request:



Verify that the draft pull request cannot be merged until it is marked as ready for review:



Draft/WIP pull requests can be useful for:

- **Feedback**: You can open a pull request early to get feedback on your work without signaling that it is ready to merge.
- **Information**: They can help communicating to others that a change is coming up and in progress.
- **Discussion**: while an issue can be used to discuss a problem, a draft pull request can be used to show a possible solution

What is a protected branch? And how to modify it?

A protected branch is a branch that has some restrictions. For example, it cannot (accidentally) deleted or force-pushed to. It is also possible to require that a branch cannot be directly pushed to or modified, but that changes must be submitted via a pull or merge request (that can be accepted or rejected by the owners or maintainers of the repository).

To protect a branch in your own repository:

- on github.com and codeberg.org: “Settings” -> “Branches”.
- on GitLab: “Settings” -> Repository -> Branches

Summary

- Issue/bug tracking is a very important part of the code development process.
- We practiced working with issues and pull requests, and how they can be related
- The pull request allowed us to contribute to a repository without directly changing its content, but ask for permission. This is appropriate in many collaborative development scenarios.

Code review demo

Here we will practice the code review process. We will learn:

- how to ask for changes in a pull request;
- how to suggest a change in a pull request;
- how to modify a pull request.

This will enable research groups to work more collaboratively, which should help to

- improve the code quality
- learn from each other.

Note that *pair programming* is usually seen as possible alternative to code review. Compared to the practice of code review, it has its own pros and cons.

Exercise

⚙ Exercise preparation

We can continue in the same exercise repository which we have used in the previous episode.

✍ Exercise: Practicing code review (25 min)

Technical requirements:

- If you create the commits locally: Being able to authenticate to your preferred forge

What should be familiar:

- Creating a branch (lesson from CodeRefinery)
- Committing a change on the new branch (lesson from CodeRefinery)
- Opening and merging pull requests (lesson from CodeRefinery)

What will be new in this exercise:

- As a reviewer, we will learn how to ask for changes in a pull request.
- As a reviewer, we will learn how to suggest a change in a pull request.
- As a submitter, we will learn how to modify a pull request without closing the incomplete one and opening a new one.

Exercise tasks:

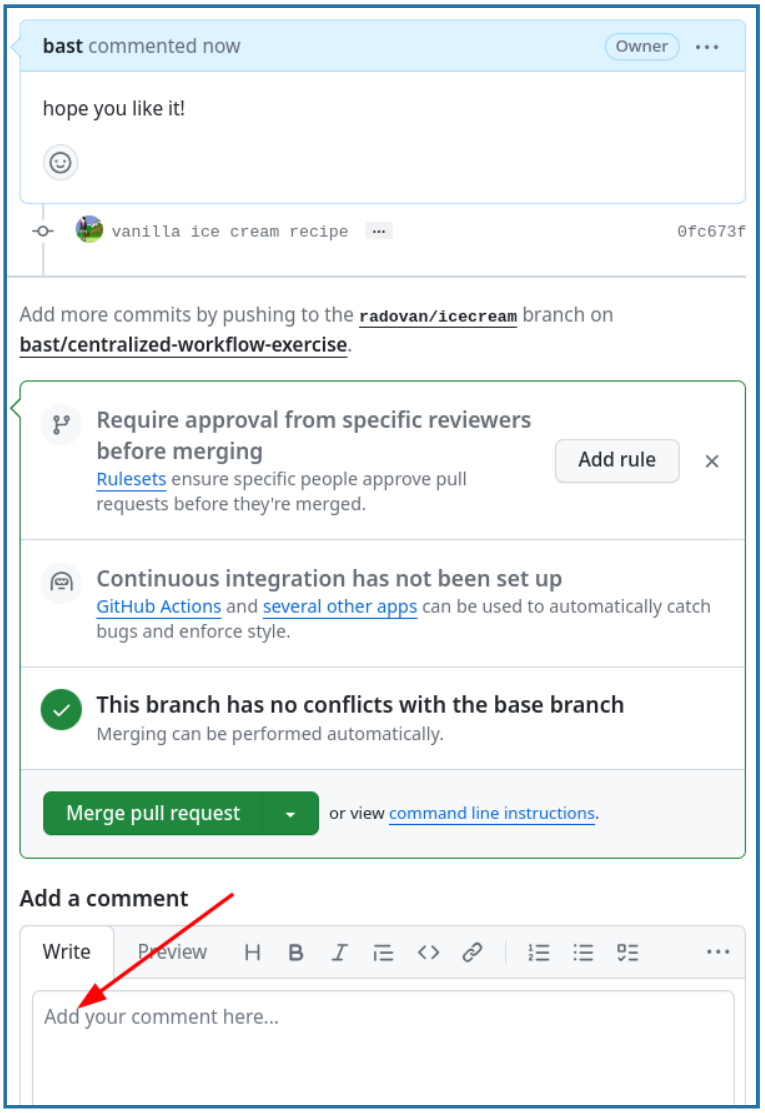
1. Create a new branch and one or few commits: in these improve something but also deliberately introduce a typo and also a larger mistake which we will want to fix during the code review.
2. Open a pull request towards the main branch.
3. As a reviewer to somebody else's pull request, ask for an improvement and also directly suggest a change for the small typo. (Hint: suggestions are possible through the GitHub web interface, view of a pull request, "Files changed" view, after selecting some lines. Look for the "±" button.)
4. As the submitter, learn how to accept the suggested change. (Hint: GitHub web interface, "Files Changed" view.)
5. As the submitter, improve the pull request without having to close and open a new one: by adding a new commit to the same branch. (Hint: push to the branch again.)
6. Once the changes are addressed, merge the pull request.

Help and discussion

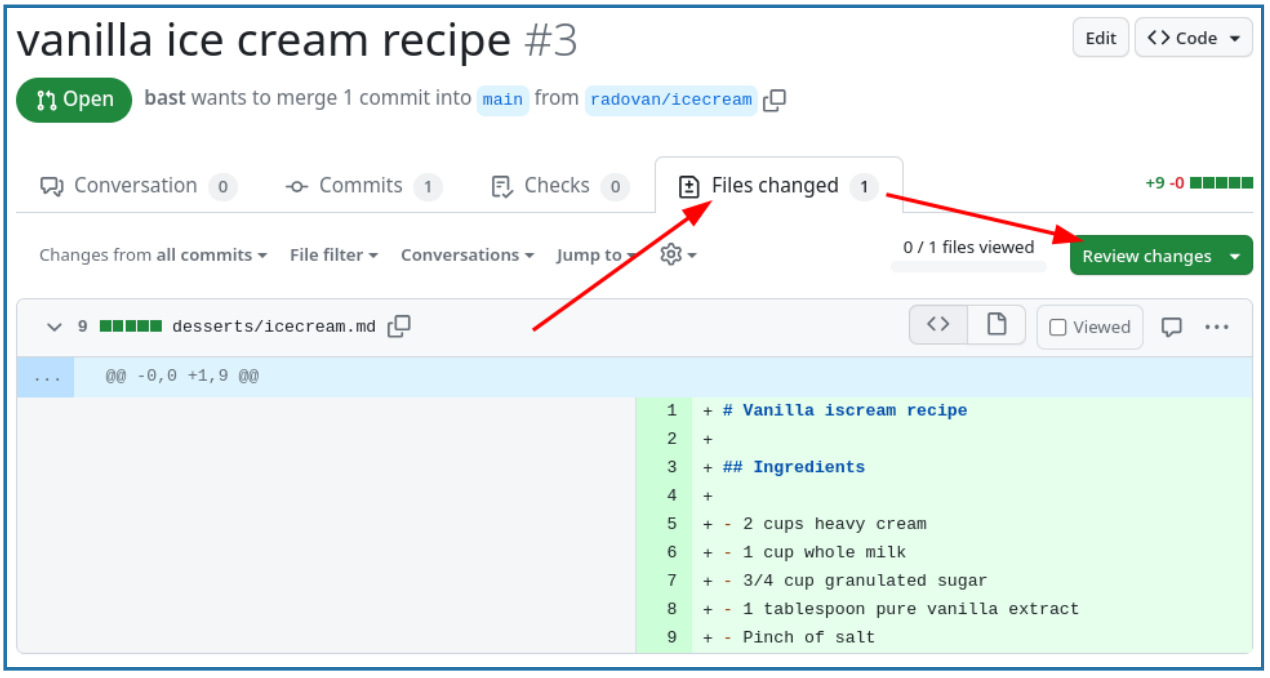
From here on out, we don't give detailed steps to the solution. You need to combine what you know, and the extra info below, in order to solve the above.

Asking for changes in a pull request: 2 ways

1. Either in the comment field of the pull request:



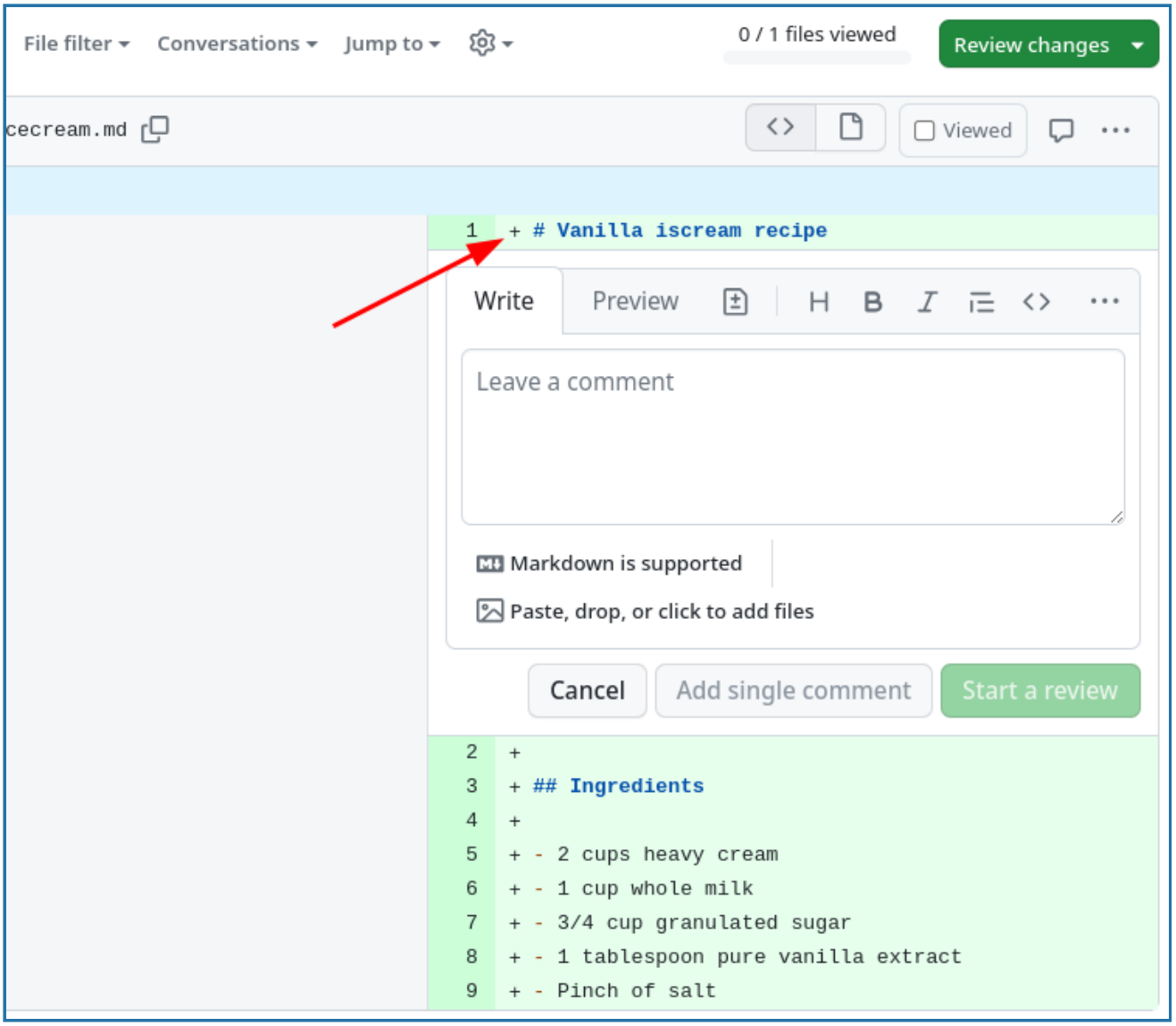
2. Or by using the “Review changes”:



And always please be kind and constructive in your comments. Remember that the goal is not gate-keeping but **collaborative learning**.

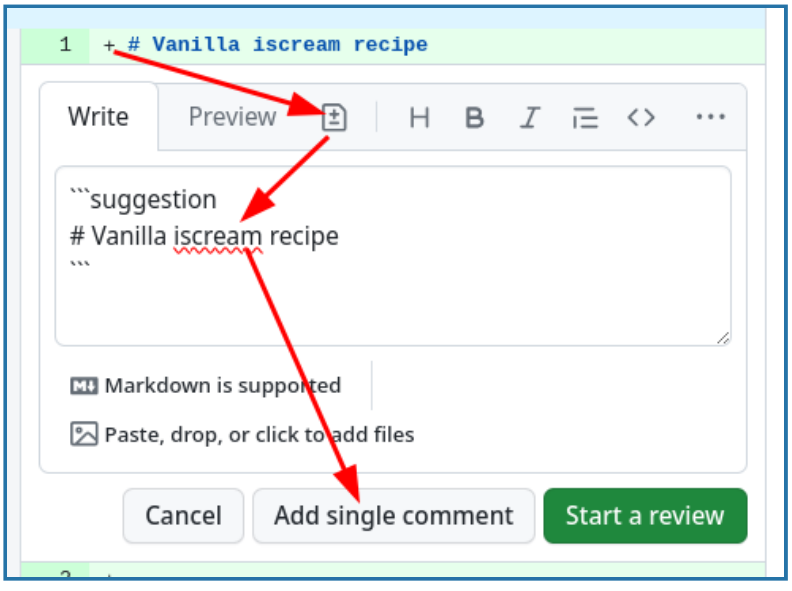
Suggest a change in a pull request as a reviewer

If you see a very small problem that is easy to fix, add a comment by clicking on the sign next to the line number in the tab that shows the changes:

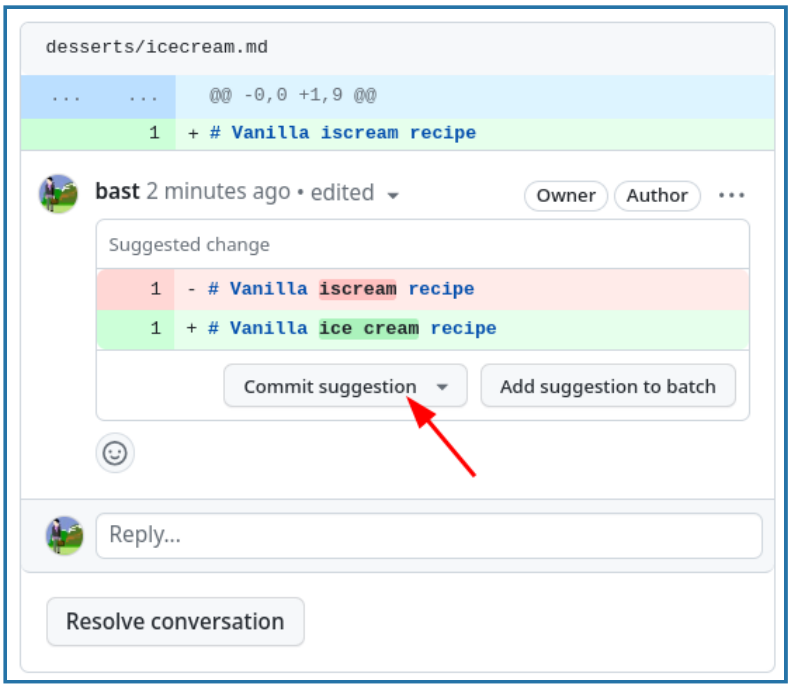


Here you can comment on specific lines or even line ranges.

Click on the “Add suggestion/Insert suggestion” symbol. Now you can fix the tiny problem (in this case a typo) and then click on the “Add single comment” button:



The result is this and the submitter can accept the change with a single click:



After accepting with “Commit suggestion”, the improvement gets added to the pull request.

How to modify a pull request to address the review comments

If the reviewer asks for changes, it is not necessary to close the pull request and later open a new one. It can even be counter-productive to do so: This can fragment the discussion and the history of the pull request and can make it harder to understand the context of the changes.

A much better mechanism to recognize that pull requests are not implemented from a specific commit to a specific branch, but **always from a branch to a branch**.

This means that you can make amendments to the pull request by adding new commits to the same source branch. This way the pull request will be updated automatically and the reviewer can see the new changes and comment on them.

The fact that pull requests are from branch to branch also strongly suggests that it is a good practice to create a new branch for each pull request. Otherwise you could accidentally modify an open pull request by adding new commits to the source branch.

Summary

- Our process isn’t just about code now. It’s about discussion and working together to make the whole process better.
- GitHub discussions and reviewing is quite powerful and can make small changes easy.

How to contribute changes to repositories that belong to others

In this episode we prepare you to suggest and contribute changes to repositories that belong to others. These might be open source projects that you use in your work.

We will see how Git and services like GitHub or GitLab can be used to suggest modification without having to ask for write access to the repository and accept modifications without having to grant write access to others.

Exercise

⚙ Exercise preparation

Part of team/exercise room | Following on your own

Maintainer (team lead):

- Create an exercise repository by generating from a template using this template: https://github.com/coderefinery/template-forking-workflow-exercise called `forking-workflow-exercise`
- In this case we **do not add collaborators** to the repository (this is the point of this example).
- Share the link to the newly created repository with your group.

Learners in exercise team: Fork the newly created repository (not the "coderefinery" one) and then **clone your fork** (if you wish to work locally).

✍ Exercise: Collaborating within the same repository (25 min)

Technical requirements:

- If you create the commits locally: Being able to authenticate to GitHub

What is familiar from the previous workshop days:

- Forking a repository (previous lesson)
- Creating a branch (previous lesson)
- Committing a change on the new branch (previous lesson)
- Opening and merging pull requests (previous lesson)

What will be new in this exercise:

- Opening a pull request towards the upstream repository.
- Pull requests can be coupled with automated testing.
- Learning that your fork can get out of date.
- After the pull requests are merged, updating your fork with the changes.
- Learn how to approach other people's repositories with ideas, changes, and requests.

Exercise tasks:

1. Open an issue in the upstream exercise repository where you describe the change you want to make. Take note of the issue number.
2. Create a new branch in your fork of the repository.
3. Make a change to the recipe book on the new branch and in the commit cross-reference the issue you opened. See the walk-through below for how to do this.
4. Open a pull request towards the upstream repository.
5. Team leaders will merge the pull requests. For individual participants, the instructors and workshop organizers will review and merge the pull requests. During the review, pay attention to the automated test step (here for demonstration purposes, we test whether the recipe contains an ingredients and an instructions sections).
6. After few pull requests are merged, update your fork with the changes.
7. Check that in your fork you can see changes from other people's pull requests.

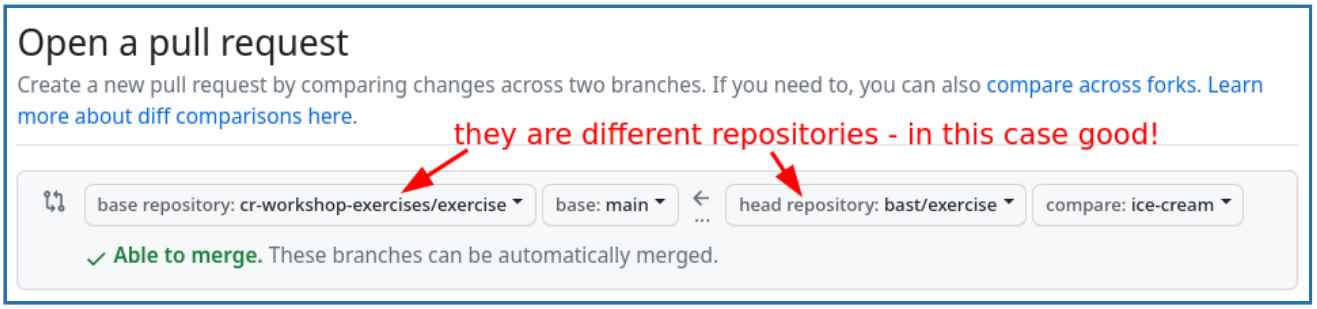
Help and discussion

Opening a pull request towards the upstream repository

We have learned in the previous episode that pull requests are always from branch to branch. But the branch can be in a different repository.

When you open a pull request in a fork, by default GitHub will suggest to direct it towards the default branch of the upstream repository.

This can be changed and it should always be verified, but in this case this is exactly what we want to do, from fork towards upstream:

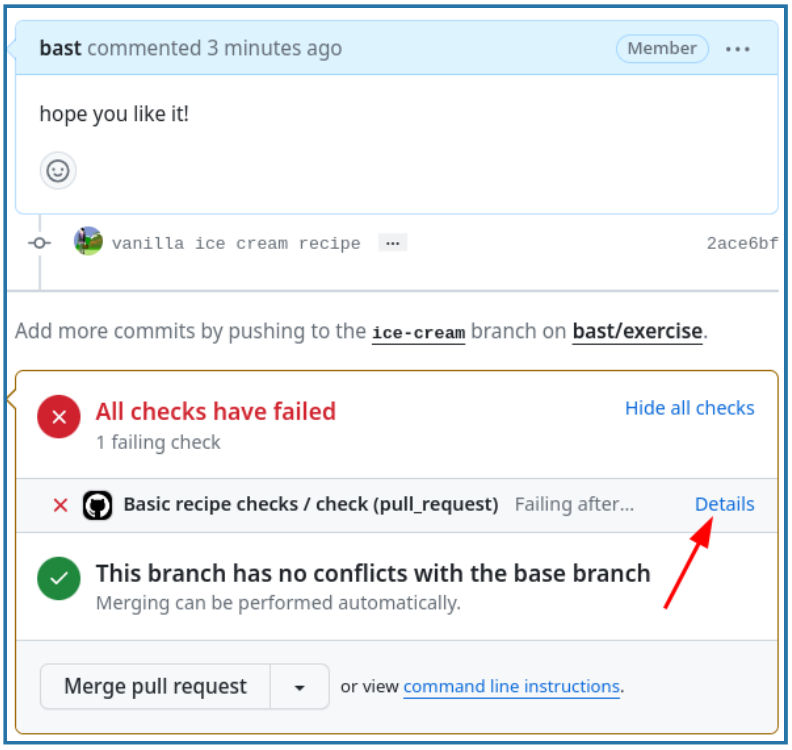


Pull requests can be coupled with automated testing

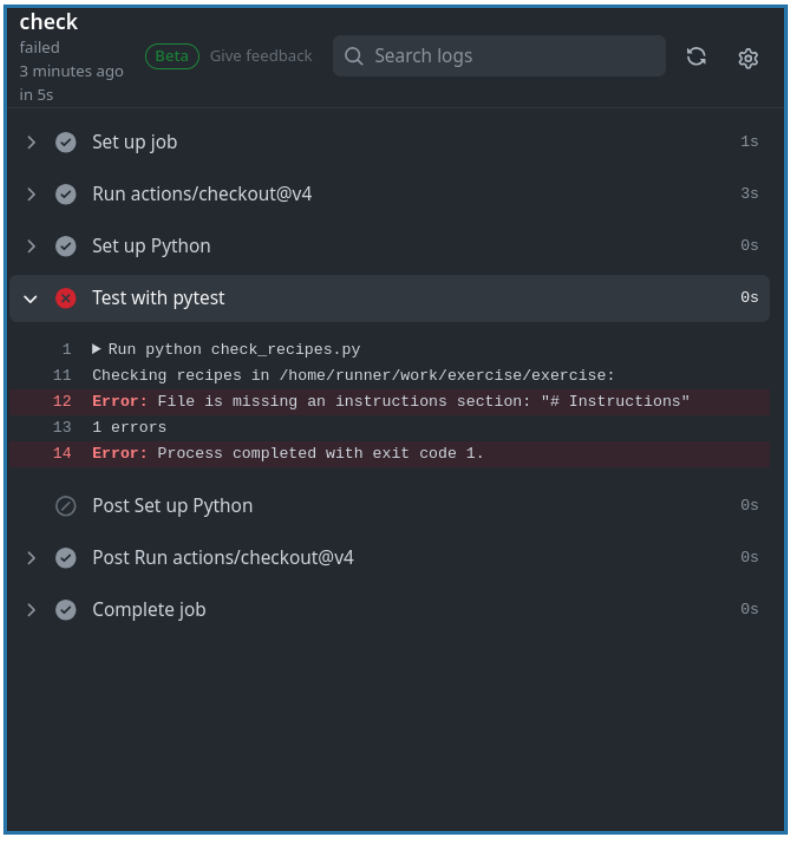
We added an automated test here just for fun and so that you see that this is possible to do.

In this exercise, the test is silly. It will check whether the recipe contains both an ingredients and an instructions section.

In this example the test failed:



Click on the “Details” link to see the details of the failed test:



How can this be useful?

- The project can define what kind of tests are expected to pass before a pull request can be merged.
- The reviewer can see the results of the tests, without having to run them locally.

How does it work?

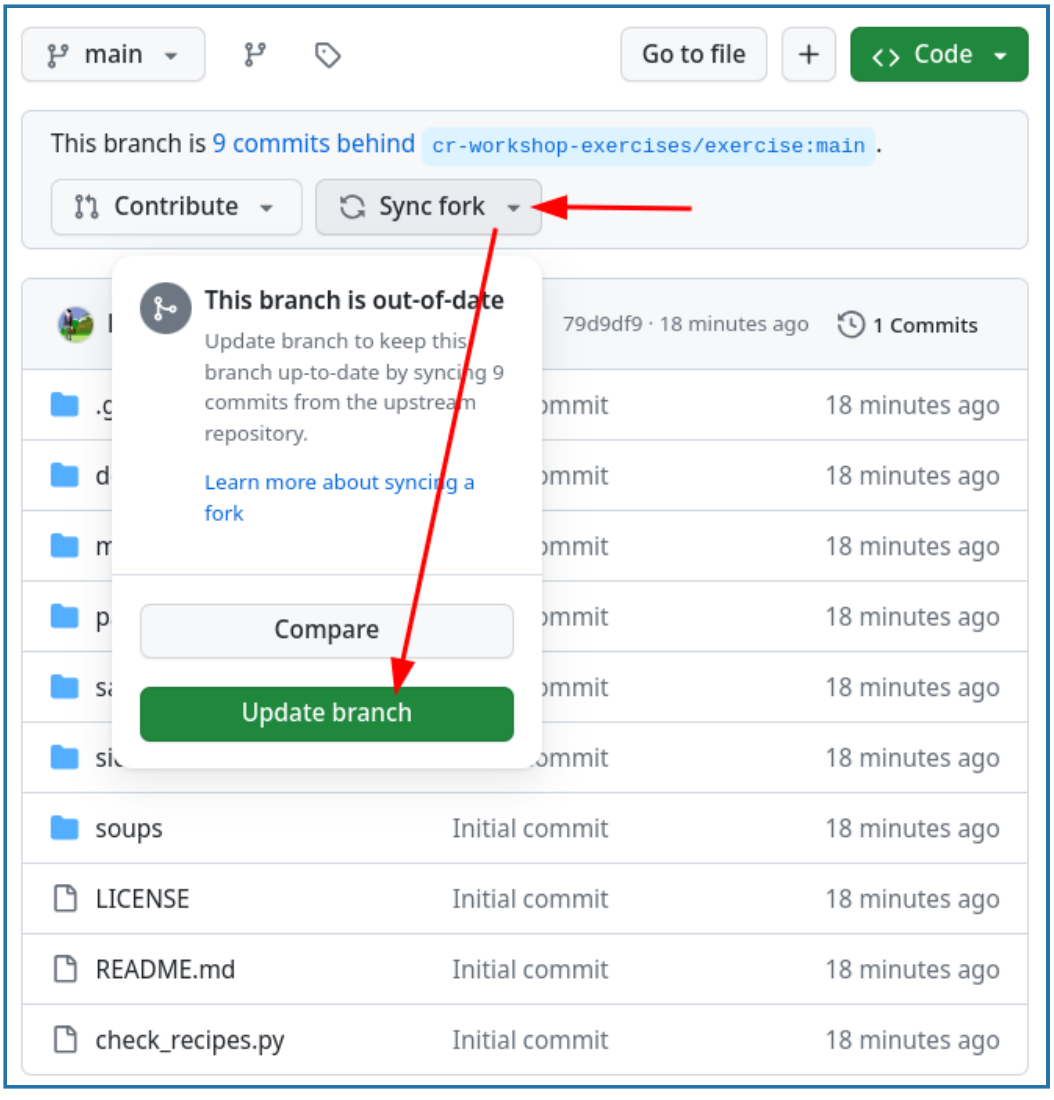
- We added a GitHub Actions workflow to automatically run on each push or pull request towards the `main` branch.

What tests or steps can you image for your project to run automatically with each pull request?

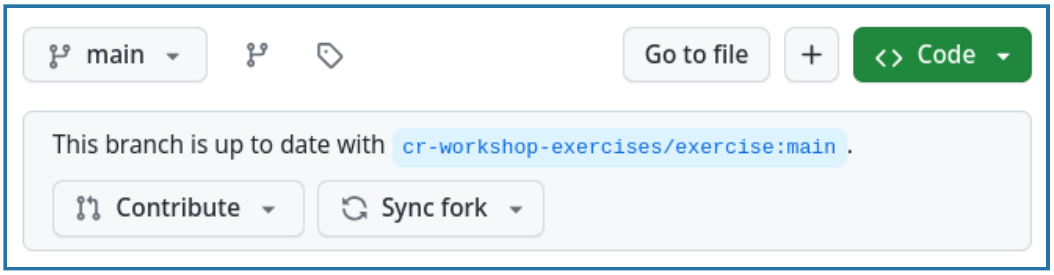
How to update your fork with changes from upstream

This used to be difficult but now it is two mouse clicks.

Navigate to your fork and notice how GitHub tells you that your fork is behind. In my case, it is 9 commits behind upstream. To fix this, click on “Sync fork” and then “Update branch”:



After the update my “branch is up to date” with the upstream repository:



How to approach other people's repositories with ideas, changes, and requests

Contributing very minor changes

- Clone or fork+clone repository
- Create a branch
- Commit and push change
- Open a pull request or merge request

If you observe an issue and have an idea how to fix it

- Open an issue in the repository you wish to contribute to
- Describe the problem
- If you have a suggestion on how to fix it, describe your suggestion
- Possibly **discuss and get feedback**
- If you are working on the fix, indicate it in the issue so that others know that somebody is working on it and who is working on it
- Submit your fix as pull request or merge request which references/closes the issue

Motivation

- **Inform others about an observed problem**
- Make it clear whether this issue is up for grabs or already being worked on

If you have an idea for a new feature

- Open an issue in the repository you wish to contribute to
- In the issue, write a short proposal for your suggested change or new feature
- Motivate why and how you wish to do this
- Also indicate where you are unsure and where you would like feedback
- **Discuss and get feedback before you code**
- Once you start coding, indicate that you are working on it
- Once you are done, submit your new feature as pull request or merge request which references/closes the issue/proposal

Motivation

- **Get agreement and feedback before writing 5000 lines of code** which might be rejected
- If we later wonder why something was done, we have the issue/proposal as reference and can read up on the reasoning behind a code change

Summary

- This forking workflow lets you propose changes to repositories for which *you have no access*.
- This is the way that much modern open-source software works.
- You can now contribute to any project you can view.

Interrupted work

Objectives

- Learn to switch context or abort work without panicking.

Instructor note

- 10 min teaching/type-along
- 15 min exercise

Keypoints

- There is almost never reason to clone a fresh copy to complete a task that you have in mind.
- Sometimes Git suggests to “stash your changes”. What is this about?

Frequent situation: interrupted work

We all wish that we could write beautiful perfect code. But the real world is much more chaotic:

- You are in the middle of a “Jackson-Pollock-style” debugging spree with 27 modified files and debugging prints everywhere.
- Your colleague comes in and wants you to fix/commit something right now.
- What to do?

Git provides lots of ways to switch tasks without ruining everything.

💬 Ways to switch context

What strategies have you used in the past?

Have you created a new clone of the repository to leave your original directory intact? Have you used `git worktree`?

Option 1: Stashing

The **stash** is the first and easiest place to temporarily "stash" things.

- `git stash push` will put working directory and staging area changes away. Your code will be same as last commit.
- `git stash pop` will return to the state you were before.
- `git stash list` will list the current stashes.
- `git stash push -m "message"` is like the first, but will give it a message. Useful if it might last a while.
- `git stash push [-p] [filename]` will stash certain files files and/or by patches.
- `git stash drop` will drop the most recent stash (or whichever stash you give).
- The stashes form a stack, so you can stash several batches of modifications.

Exercise: Stashing

✍ Interrupted-1: Stash some uncommitted work

1. Make a change.
2. Check status/diff, stash the change with `git stash`, check status/diff again.
3. Make a separate, unrelated change which doesn't touch the same lines. Commit this change.
4. Pop off the stash you saved with `git stash pop`, and check status/diff.
5. Optional: Do the same but stash twice. Also check `git stash list`. Can you pop the stashes in the opposite order?
6. Advanced: What happens if stashes conflict with other changes? Make a change and stash it. Modify the same line or one right above or below. Pop the stash back. Resolve the conflict. Note there is no extra commit.
7. Advanced: what does `git graph` show when you have something stashed?

✔ Solution

5: Yes you can. With `git stash pop INDEX` you can decie which stash index to pop.

6: In this case Git will ask us to resolve the conflict the same way when resolving conflicts between two branches.

7: It shows an additional commit hash with `refs/stash`.

✍ Stashing all

Sometimes we want to stash files that are not yet tracked by git (i.e., have not been `add`ed). How would we do that? Look at the man page using `git help stash`.

✔ Solution

By passing the option `-a`, we are telling `git stash` to take every file in our working tree, including untracked and ignored files.

✍ Comments

The option `-m` to add a message is optional. Why use it?

✔ Solution

By looking at the output of `git stash list`, it will be much easier to determine which stash we are interested in.

✍ Stash vs commit

In what sense are stashes similar to commits?

✔ Solution

```
Stashing is roughly equivalent to
```console
git switch -c tempbranch; git add -u; git commit -m 'temp commit'}).
```
In particular, stashes are identified as `commit` objects in the object
database,
and they are referenced by `refs/stash` and the reflog of the "stash" reference.
```

Option 2: Create branches

You can use branches almost like you have already been doing if you need to save some work. You need to do something else for a bit? Sounds like a good time to make a feature branch.

You basically know how to do this:

```
$ git switch --create temporary  # create a branch and switch to it
$ git add PATHS                  # stage changes
$ git commit                     # commit them
$ git switch main                # back to main, continue your work there ...
$ git switch temporary           # continue again on "temporary" where you left off
```

Later you can merge it to main or rebase it on top of main and resume work.

Storing various junk you don't need but don't want to get rid of

It happens often that you do something and don't need it, but you don't want to lose it right away. You can use either of the above strategies to stash/branch it away: using branches is probably better because branches are less easily overlooked if you come back to the repository in few weeks. Note that if you try to use a branch after a long time, conflicts might get really bad but at least you have the data still.

Tooling and practices that you might find useful

Objectives

- A bird's eye view of git-related tooling
- Install and configure 1 tool of your choice so that you can start using it

Difftools and merge tools.

There are many file types for which the usual output of `git diff` can be from difficult to read to just completely impossible to understand.

- For Jupyter notebooks: nbdime
- For Latex: Latexdiff (not git-related) and git-latexdiff. Latex is still a text-based format, but a PDF-rendered view of the differences can be more readable.
- There are also tools for images (e.g., git-diff-image)

Automation: Git Hooks

Git can be configured to perform some tasks automatically when some events happen.

Most notable tasks:

- auto-formatting: is your code properly formatted? This is important because:
  - proper formatting improves readability
  - consistently using automatic formatting makes the output of `git diff` much more informative (for easier code reviews) There are tools for every language you use:
  - for Python: black
  - for C/C++: clang-format
- Linting: there are automated tools that can spot bad practices in writing code
  - for Python: pylint
  - for C/C++: clang-tidy
  - for bash shell scripts: shellcheck

- Spellchecking (useful for documentation)
- compiling/building, deploying services or documentation
- Launch a test suite

Such tasks can be performed as part of a *git hook*. Git hooks are executable programs in the `.git/hooks/` directory.

The most commonly used is the *pre-commit* hook, which runs when you call `git commit`, before the commit message is created. Auto-formatting, linting tools and anything that is quick enough can be run here.

✍ Try them out!

Install and/or configure some of the mentioned tooling that can be helpful for your daily workflow.

Another hook typically used is *post-receive*. When it is configured on a remote repository, it runs after a push. The *post-receive* hook is typically used to start the run of a test suite, or to notify other services that the push happened.

Automation: GitHub Actions, GitLab CI/CD (et similia)

Automation platforms like, e.g. GitHub actions and GitLab CI/CD build on top of the idea of the post-receive hook, and are commonly used for (including but not limited to):

- run a test suite and present the results in a web interface;
- build the software and make it available for download;
- build and deploy documentation.

Merge and beyond

Git exposes many commands that can be used to add the work done in a branch into another branch.

For the following demonstration, you can clone a toy repository created on purpose:

```
$ git clone https://github.com/mmesiti/merge-fu.git
$ cd merge-fu
```

Have a look at the branch structure:

```
$ git graph # alias for git log --oneline --all --graph --decorate
```

For what follows, we do not want to see all the branches all the time, so we want to define a `git gl` alias *locally* without the `--all` flag:

```
$ git config alias.gl --oneline --graph --decorate
```

so that we can use it like this:

```
$ git gl branch-1 branch-2
```

Merge

`git merge` is the classic command that is used join (potentially more than 2) branches together.

It will create an additional commit, called the *merge commit*.

When Git cannot determine unambiguously how to merge two versions of a given file, it will produce a *conflict*.

Solving conflicts requires some practice and typically some thought.

Complex conflicts can be made easier to understand by configuring git to show also the version in the *merge base* in addition to the two conflicting versions:

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

Using a *merge tool* can also help when there are large change sets to merge. Please refer to the documentation for more information.

Merge Conflict and abort

Let us now try to merge `branch-1` into `branch-2`. We first need to create the local branches that match the remote ones:

```
$ git branch branch-1 origin/branch-1
$ git branch branch-2 origin/branch-2
```

We can check what are the differences between the two branches:

```
$ git diff branch-1 branch-2
diff --git a/text-file.txt b/text-file.txt
index b7062f5..ce1f6b8 100644
--- a/text-file.txt
+++ b/text-file.txt
@@ -1,4 +1,4 @@
 1st line
 2nd line
 3rd line
-4th line on branch 1
+4th line on branch 2
```

✍️ Predict conflicts

Will the merge between branch-1 and branch-2 cause a conflict? Why?

✔ Solution

Unfortunately, yes. Both versions have appended lines at the end, and Git cannot determine in which order they need to be.

First of all, to do the merge we need to switch to `branch-2`:

```
$ git switch branch-2
```

We will get a conflict:

```
$ git merge branch-1
Auto-merging text-file.txt
CONFLICT (content): Merge conflict in text-file.txt
Automatic merge failed; fix conflicts and then commit the result.
```

We can check the content of `text-file.txt`:

```
$ cat text-file.txt
1st line
2nd line
3rd line
<<<<<<< HEAD
4th line on branch 2
||||||| 874ebe0
4th line
=======
4th line on branch 1
>>>>>>> branch-1
```

If we know how to solve it, we can modify the file, stage it and commit.

But what to do in the unhappy situation where we are not sure how to proceed? We stop the merge with the command

```
$ git merge --abort
```

The `--abort` option is a useful “handbrake” that works also with other commands.

No conflicts, but still wrong

There are cases where a conflictless `git merge` can introduce a bug.

For example, switch to the branch `python-example`:

```
$ git switch python-example
```

Check the content of the `example.py` file:

```
$ cat example.py
def add1(n):
    res = n

    print("This function adds 1 to the input")

    return res
```

This is obviously wrong: the function is not adding 1.

Fortunately, we have already two possible fixes, by Alice and Bob. One is on branch `python-example-fix-1`:

```
$ git switch python-example-fix-1
$ cat example.py
def add1(n):
    res = n + 1

    print("This function adds 1 to the input")

    return res
```

And another is on branch `pythyon-example-fix-2`:

```
$ git switch python-example-fix-2
$ cat example.py
def add1(n):
    res = n

    print("This function adds 1 to the input")

    return res + 1
```

They are just one commit away from `python-example`:

```
$ git gl python-example-fix-1 python-example-fix-2
* 4d8b65f (origin/python-example-fix-2, python-example-fix-2) fix add1
| * 0392b16 (origin/python-example-fix-1, python-example-fix-1) fix add1
|/
* ff35a6e (HEAD -> python-example, origin/python-example) add python example
* 874ebe0 (origin/main, origin/HEAD, main) First commit
```

Excellent! We will merge them both into `python-example`, to make everybody feel like their work is appreciated. We switch to the `python-example` branch:

```
$ git switch python-example
Switched to branch 'python-example'
Your branch is up to date with 'origin/python-example'.
```

We merge first `python-example-fix-1`:

```
$ git merge python-example-fix-1
Updating ff35a6e..0392b16
Fast-forward
 example.py | 2 +-
 1 file changed, 1 insertion(+), 1 deletion(-)
```

This merge is a *fast forward*: `python-example-fix-1` is a direct descendant of the `python-example` so `python-example` can be just moved forward without too much thinking.

We then merge `python-example-fix-2`:

```
$ git merge python-example-fix-2
Auto-merging example.py
Merge made by the 'ort' strategy.
 example.py | 2 +-
 1 file changed, 1 insertion(+), 1 deletion(-)
```

The file is now **wrong**, though:

```
$ cat example.py
def add1(n):
    res = n + 1

    print("This function adds 1 to the input")

    return res + 1
```

We are adding 1 twice!

This is obviously a contrived example. But it shows that:

1. conflicts might be annoying, but are actually a good thing;
2. merges should always be checked in some way, by a human and/or with an automatic test suite.

To fix this, we can undo the commit in one of the ways we have already seen.

✍ Optional: `revert` a merge commit

When reverting a merge commit, it is not clear which is the parent commit to which we want to revert.

Use the `-m` option (`--mainline`) to select the version you want to revert to. See the documentation for `git revert`.

Cherry-Pick

There might be a commit in a branch that we want to use, without merging the whole branch on which it was created.

For this we will consider the branches `proverbs` and 'good-and-bad-commits':

```
$ git gl proverbs good-and-bad-commits
```

The branch that contains our work is `proverbs`, where we started a collection of popular pieces of wisdom. Perhaps the branch `good-and-bad-commits` contains some useful work? We can check it with `git log -p`, which will show all the changes along with the commit messages:

```
$ git log --oneline -p proverbs..good-and-bad-commits | cat
9396ffd git is hard!!!!!
diff --git a/wisdom.txt b/wisdom.txt
index b3c8fac..ec51263 100644
--- a/wisdom.txt
+++ b/wisdom.txt
@@ -4,3 +4,6 @@ Early to bed,
 early to rise,
 makes a man wealthy,
 healthy, and wise.
+
+
+I HATE VERSION CONTROL!
82cfb15 Add proverb
diff --git a/wisdom.txt b/wisdom.txt
index c343ccb..b3c8fac 100644
--- a/wisdom.txt
+++ b/wisdom.txt
@@ -1 +1,6 @@
 # Old Proverbs
+
+Early to bed,
+early to rise,
+makes a man wealthy,
+healthy, and wise.
```

here `proverbs..good-and-bad-commits` is a way of specifying the range of commits above *merge base* on the branch `good-and-bad-commits`.

Once we see the content of each commit, we become interested in applying the second-last commit on `good-and-bad-commits` to the `proverb` branch.

To do so, we switch to the `proverb` branch

```
$ git switch proverb
```

and use `git cherry-pick` with the commit we want to apply

```
$ git cherry-pick  good-and-bad-commits~
Auto-merging wisdom.txt
CONFLICT (content): Merge conflict in wisdom.txt
error: could not apply 82cfb15... Add proverb
hint: After resolving the conflicts, mark them with
hint: "git add/rm <pathspec>", then run
hint: "git cherry-pick --continue".
hint: You can instead skip this commit with "git cherry-pick --skip".
hint: To abort and get back to the state before "git cherry-pick",
hint: run "git cherry-pick --abort".
```

We have a conflict, but the resolution in this case is trivial.

Rebase

`git rebase` is an alternative to `git merge` that typically leads to a clearer commit history.

In particular:

- an additional merge commit is not necessary
- the commit graph has no bifurcations

The `rebase` command will try to reapply all the commits on the current branch on top of another branch (which will be left untouched), and then point the current branch at the last commit.

The Golden Rule of Rebase

Do not be rude: `git rebase` *rewrites history*. Be very careful when rebasing public branches!

Rebase demo

For this demo we will switch on branch `rebase-me`

```
$ git switch rebase-me
```

and try to rebase it onto the branch `rebase-onto-this`, which we need to create locally from the remote branch, with this command:

```
$ git branch rebase-onto-this origin/rebase-onto-this
```

We can have a look at the branch structure:

```
$ git gl rebase-me rebase-onto-this
* 3b514df (rebase-onto-this) Add line at end
* e459dcd Add an intermezzo
* a4cc39e (origin/branch-1, branch-1) 2nd commit - on branch-1
| * d30163f (HEAD -> rebase-me) 3rd commit - on branch rebase-me
| * 98f36f0 2nd commit - on branch rebase-me
|/
* 874ebe0 (origin/main, origin/HEAD, main) First commit
```

We see that:

- there is a bifurcation at `874ebe0`
- our current branch (`rebase-me`) has 2 commits above the merge base
- the branch we want to rebase on (`rebase-onto-this`) has 3 commits above the merge base.

To be able to compare the end result with the initial situation, we create a “backup branch” as a bookmark:

```
$ git branch rebase-me-original rebase-me
```

We now can do the proper rebase. Make sure we are on the `rebase-me` branch:

```
$ git branch
  branch-1
  branch-2
  good-and-bad-commits
  main
  proverbs
* rebase-me
  rebase-me-original
  rebase-onto-this
```

then we invoke the rebase command to rebase the current branch (`rebase-me`) onto `rebase-onto-this` :

```
$ git rebase rebase-onto-this
```

This command will try to apply all the commits on the current branch (`rebase-me`) onto the branch `rebase-onto-this` , one at a time. For each commit we might get a conflict, which is the first thing

```
Auto-merging text-file.txt
CONFLICT (content): Merge conflict in text-file.txt
error: could not apply 98f36f0... 2nd commit - on branch rebase-me
hint: Resolve all conflicts manually, mark them as resolved with
hint: "git add/rm <conflicted_files>", then run "git rebase --continue".
hint: You can instead skip this commit: run "git rebase --skip".
hint: To abort and get back to the state before "git rebase", run "git rebase --abort".
Could not apply 98f36f0... 2nd commit - on branch rebase-me
```

We can resolve this conflict in the way we please.

```
$ # edit text-file.txt
```

Once we are done, we can add our changes:

```
$ git add text-file.txt
```

and tell rebase to continue to the next commit:

```
$ git rebase --continue
```

When rebase can automatically merge without commits, it will not ask for our intervention, but when there are conflicts it will stop and ask us to solve them, `git add` the results and then use `git rebase --continue`.

After all the commits on the current branch are processed, we will get a linear commit history for the current branch:

```
$ git log --oneline
f5b0417 (HEAD -> rebase-me) 3rd commit - on branch rebase-me
3b514df (rebase-onto-this) Add line at end
e459dcd Add an intermezzo
a4cc39e (origin/branch-1, branch-1) 2nd commit - on branch-1
874ebe0 (origin/main, origin/HEAD, main) First commit
```

We can compare the new commit history with the original position of the branch:

```
$ git gl rebase-me rebase-me-original
* f5b0417 (HEAD -> rebase-me) 3rd commit - on branch rebase-me
* 3b514df (rebase-onto-this) Add line at end
* e459dcd Add an intermezzo
* a4cc39e (origin/branch-1, branch-1) 2nd commit - on branch-1
| * d30163f (rebase-me-original) 3rd commit - on branch rebase-me
| * 98f36f0 2nd commit - on branch rebase-me
|/
* 874ebe0 (origin/main, origin/HEAD, main) First commit
```

As `merge` and `cherry-pick`, `rebase` has a `--abort` option.

If we are not satisfied by the result of the rebase after it completed, we can use `git reflog rebase-me` to determine the last satisfactory commit, and use `git reset` to move the branch to point there again.

Sometimes, the same conflicts will need to be solved over and over in the same way. In such a situation, the `rerere` command (`re`use `re`corded `re`solution) may come in handy.

Interactive rebase

`git rebase` has an interactive mode (that can be entered using the `-i` flag, or `--interactive`) that can be used to perform complex manipulations of the commit history in a range.

It is very powerful, and it is also used to clean the commit history of a feature branch before making a pull request (in this case, there are lower chances for conflicts because we rebase on a commit that is already an ancestor of the current branch, e.g. `git rebase -i HEAD~3`).

It is possible to perform the following actions on any commit in the range:

- **pick**: keep it in the history;
- **drop**: dropt it from the history;
- **reword**: change only the commit message
- **squash**: remove the commit, but attribute its changes to the previous picked commit.
- **edit**: change the files and the commit message (even create new commits in the meantime - the opposite of squashing)
- **exec**: pause the rebasing and run a command there (e.g., a test suite)

More information can be read from the manual.

✍ Interactive rebase on another branch

You can practice interactive rebase with

```
$ git switch rebase-me
$ git reset reset --hard d30163f
$ git rebase -i rebase-onto-this
```

Quick reference

Other cheatsheets

See the git-intro cheatsheet for the basics.

- Interactive git cheatsheet

- Very detailed 2-page git cheatsheet

Glossary

forge

A web-based collaborative software platform for both developing and sharing code (from wikipedia). Common example of forges are github.com, gitlab.com, codeberg.org, and self-hosted instances of GitLab or Forgejo.

remote

Roughly, another git repository on another computer. A repository can be linked to several other remotes.

push

Send a branch from your current repository to another repository

fetch

Update your view of another repository

pull

Fetch (above) and then merge

origin

Default name for a remote repository.

origin/NAME

A branch name which represents a remote branch.

main

Default name for main branch.

merge

Combine the changes on two branches.

conflict

When a merge has changes that affect the same lines, git can not automatically figure out what to do. It presents the conflict to the user to resolve.

issue

Feature of web repositories that allows discussion related to a repository.

pull request

A GitHub/Gitlab feature that allows you to send a code suggestion using a branch, which allows one-button merging. In Gitlab, called "**merge request**".

git hook

Code that can run before or after certain actions, for example to do tests before allowing you to commit.

bare repository

A copy of a repository that only is only the `.git` directory: there are no files actually checked out. Directory names usually like `something.git`

working tree

The directory where the files of your project live, excluding the `.git` subdirectory. It represents all that non git-aware applications can interact with, and it exists independently of git, but it can be manipulated by git

index

Also called sometimes "staging area". A version of a file is added to it with `git add`.

object

A git object is one of 4 kinds: a commit (representing a commit or a stash), a tree (representing a directory), a blob (representing a file) or a tag.

Commands we use

This excludes most introduced in the git-intro cheatsheet.

Setup:

- `git clone URL [TARGET-DIRECTORY]`: Make a copy of existing repository at <url>, containing all history.

Status:

- `git status`: Same as in basic git, list status
- `git remote [-v]`: List all remotes
- `git graph`: see a detailed graph of commits. Create this command with `git config --global alias.graph "log --all --graph --decorate --oneline"`

General work:

- `git switch BRANCH-NAME`: Make a branch active.
- `git push [REMOTE-NAME] [BRANCH:BRANCH]`: Send commits and update the branch on the remote.
- `git pull [REMOTE-NAME] [BRANCH-NAME]`: Fetch and then merge automatically. Can be convenient, but to be careful you can fetch and merge separately.
- `git fetch [REMOTE-NAME]`: Get commits from the remote. Doesn't update local branches, but updates the remote tracking branches (like origin/NAME).
- `git merge [BRANCH-NAME]`: Updates your current branch with changes from another branch. By default, merges to the branch is is tracking by default.
- `git remote add REMOTE-NAME URL`: Adds a new remote with a certain name.

List of exercises

Full list

This is a list of all exercises and solutions in this lesson, mainly as a reference for helpers and instructors. This list is automatically generated from all of the other pages in the lesson. Any single teaching event will probably cover only a subset of these, depending on their interests.

Instructor guide

Schedule

- 08:50: Soft start
- 09:00: Necessary introductions, forming groups.
- 09:15: Recap on Git basics
- 09:40: Beyond add and commit: undoing mistakes
- 10:00: Inspecting history
- 10:30: Break
- 10:50: Concepts around collaboration
- 11:00: Collaborating within the same repository
- 11:30: Demo: Code review
- 11:40: Demo: How to contribute changes to repositories that belong to others
- 11:50: Tooling and practices
- 12:00: Merge and Beyond
- 12:10: Free practice and discussion

Why I modified this lesson

The main change is that I am adding a little more content.

My impression is that in CodeRefinery workshops the pace is a bit relaxed, and more could be done in a in-person setting (which is the plan here).

I would not let attendees work on their own on an exercise for 30 minutes without feedback, so I changed the times of the exercise lessons to also include instructor feedback.

Another change is that I try not to be **forge**-specific. One reason is that the expected audience typically has access to GitLab instances, and also someone might be concerned about handing all their code to a Microsoft-owned platform, so I am adding codeberg.org as an alternative. Another reason is that actually the features of every forge typically evolve over time. Looking at a variety of forges at a single time point might give an idea of the distribution of features of a single forge at different points in time.

Audience intended for this course

There seems to be quite a demand for an intermediate Git course at KIT.

The audience of this course is expected to be slightly more familiar with git than target of the original CodeRefinery workshops (which start on day 1 assuming no git knowledge - so I think), and definitely more advanced than the audience that would attend a Software Carpentry lesson on Git.

At KIT, we do offer the software carpentry curriculum already twice a year, complete beginners should attend those courses instead of coming to this course.

Intended prerequisites

At the beginning of this lesson, learners:

- Must know the basic git commands (status/diff/add/commit)
- Should be comfortable with the command line
- Should Be familiar with the usual git workflow (pull/add/commit/push)
- Probably already have used github or similar
- **Should have a way to authenticate to the chosen forge**

Intended learning outcomes

By the end of this lesson, learners should:

- Understand the concept of remotes
- Be able to describe the difference between local and remote branches
- Be able to describe the difference between centralized and forking workflows
- Be able to work efficiently with forges:
  - Know how to use pull requests or merge requests to submit changes to another projects
  - Know how to reference issues in commits or pull/merge requests and how to auto-close issues
  - Know how to update a fork
- **Be able to contribute in code review as submitter or reviewer**
- Know the difference between merge and rebase, in particular:
  - Know the golden rule of rebase
  - What force push means (what are the consequences)
  - What are the advantages of a linear commit history
- Choose the right tool for fixing common problems with Git *(I know this is a little vague)*/ This includes:
  - issues with lost data when using git add/checkout/restore
  - cleaning their commit history if they so wish (rebase)
  - deal with large binary files (lfs/annex)
  - deal with large repositories (partial cloning)
  - using long complex commands efficiently (aliases)
  - use git to analyse development history (pickaxe, blame, bisect)

    - re-discover which branches they had been working on before
    - dealing with nested repositories (existence of submodules)
    - when you do not want to add/commit part of the changes you made to a file, without having to undo (potentially big) changes in your editor (-p)
    - collaboration between windows and *nix users (line ending issues)
    -

Instructor guide - original

Schedule - Original

- 08:50 - 09:00: Soft start and icebreaker question
- 09:00 - 09:15: Recap Git, any HedgeDoc questions to highlight
- 09:15 - 09:30: Concepts around collaboration
    - Explain terms: Pull, push, clone, fork. Focus on pull and not fetch.
    - Focus more on clone and less on generating from templates and importing.
- 09:30 - 11:00: Centralized workflow
    - 9:30 - 9:45: Explain concepts
    - 9:45 - 9:55: Break
    - 9:55 - 10:00: Inform clearly what is expected outcome
    - 10:00 - 10:30: Exercise
    - 10:30 - 11:00: Instructors go through the exercise. Discussion and answering questions
- 11:00 - 12:00: Lunch Break
- 12:00 - 13:10: Distributed version control and forking workflow
    - 12:00 - 12:15: Concepts and what are exercise outcomes
    - 12:15 - 12:45: Exercise
    - 12:45 - 12:55 Break
    - 12:55 - 13:10: Instructors go through excercises. Discussion and answering questions
- 13:10 - 13:30: How to contribute changes to somebody else's project and Q&A

Why we teach this lesson - original

In order to collaborate efficiently using Git, it's essential to have a solid understanding of how remotes work, and how to contribute changes through pull requests or merge requests. The git-intro lesson teaches participants how to work efficiently with Git when there is only one developer (more precisely: how to work when there are no remote Git repositories yet in the picture). This lesson dives into the collaborative aspects of Git and focuses on the possible collaborative workflows enabled by web-based repository hosting platforms like GitHub.

This lesson is meant to directly benefit workshop participants who have prior experience with Git, enabling them to put collaborative workflows involving code review directly into practice when they return to their normal work.

For novice Git users (who may have learned a lot in the git-intro lesson) this lesson is somewhat challenging, but the lesson aims to introduce them to the concepts and give them confidence to start using these workflows later when they have gained some further experience in working with Git.

Intended learning outcomes

By the end of this lesson, learners should:

- Understand the concept of remotes
- Be able to describe the difference between local and remote branches
- Be able to describe the difference between centralized and forking workflows
- Know how to use pull requests or merge requests to submit changes to another projects
- Know how to reference issues in commits or pull/merge requests and how to auto-close issues
- Know how to update a fork
- **Be able to contribute in code review as submitter or reviewer**

Interesting questions you might get

- If participants run `git graph` they might notice `origin/HEAD`. This has been omitted from the figures to not overload the presentation. This pointer represents the default branch of the remote repository.

Timing

- The centralized collaboration episode is densest and introduces many new concepts, so at least an hour is required for it.
- The forking-workflow exercise repeats familiar concepts (only introduces forking and distributed workflows), and it takes maybe half the time of the first episode.
- The “How to contribute changes to somebody else’s project” episode can be covered relatively quickly and offers room for discussion if you have time left. However, this should not be skipped as this is perhaps the key learning outcome.

Preparing exercises

Exercise leads typically prepare exercise repositories for the exercise group (although the material speaks about “maintainer” who can also be one of the learners). Preparing the first exercise (centralized workflow) will take more time than preparing the second (forking workflow). Most preparation time is not the generating part but will go into communicating the URL to the exercise group, communicating their usernames, adding them as collaborators, and waiting until everybody accepts the GitHub invitation to join the newly created exercise repository.

Live stream:

- Create the centralized exercises **in an organization** (not under your username) so that you can give others admin access to add collaborators. Also this way you can then fork yourself if needed.
- For CR workshops, the exercises were placed under https://github.com/cr-workshop-exercises. The instructors or team leads need to have owner status in the organization in order to invite people.
- We have created two versions of each **a day in advance** to signal which one might end up being discussed on recording/stream:
    - `centralized-workflow-exercise-recorded`
    - `centralized-workflow-exercise`
    - `forking-workflow-exercise-recorded`
    - `forking-workflow-exercise`
- Protect the default branch of the two `centralized-*` repositories.
- We create a organization team, `stream-exercise-participants`. The *centralized* workflow exercise repos have this team added as a collaborator (*not* forking - they fork so they don't need write access there).
- We have collected usernames of people who want to contribute via issues on GitHub. Make a fifth repository, `access-requests`, create a sample access request issue there, and have learners make a new issue in that repository. The day/morning before the day of the lesson the instructor or team leader now has to invite the learners to the team. Three steps: 1. copy the learners GitHub username from the issue 2. go to team member page, example linked here and invite that username to the team (this means first clicking invite and then scrolling down to click the "add username to ..." button. This sends an email to that users email that is connected to their GitHub account. 3. In the issue, copy following text (or similar) to the issue and "close with comment":

```
We have added you to the CodeRefinery exercise repository.

What you should do before the exercise starts:

You will get an invitation from GitHub to your email address (that GitHub knows
about). Please accept that invitation so that you can participate in the
collaborative exercise.
To make sure you don't get too many emails during the exercise, don't forget to
"unwatch" both https://github.com/cr-workshop-exercises/centralized-workflow-
exercise and https://github.com/cr-workshop-exercises/centralized-workflow-
exercise-recorded.
To "unwatch", go to the repository and click the "Unwatch" button (top middle of
the screen) and then select "Participating and @mentions".
```

``

- Why a fifth repository? So that learners don't get emails from all other access requests once they get added to the team
- Example email requesting learners to join
- Example issue comment

Typical pitfalls

Difference between pull and pull requests

The difference between pull and pull requests can be confusing, explain clearly that pull requests or merge requests are a different mechanism specific to GitHub, GitLab, etc.

Pull requests are from branch to branch, not from commit to branch

The behavior that additional commits to a branch from which a pull request has been created get appended to the pull request needs to be explained.

Other practical aspects

- In in-person workshops participants really have to sit next to someone, so that they can see the screens. From the beginning.
- Emphasize use of `git graph` a lot, just like in the git-solo lesson.