Reproducible research software development using Python

Big-picture goal

This is a **hands-on course on research software engineering.** In this workshop we assume that most workshop participants use Python in their work or a leading a group which uses Python. Therefore, some of the examples will use Python as the example language.

We will work with an example project and go through all important steps of a typical software project. Once we have seen the building blocks, we will try to **apply them to own projects**.

Prerequisites

⚙ Preparation

1. Get a **GitHub account** following these instructions.
2. You will need a **text editor**. If you don't have a favorite one, we recommend VS Code.
3. **If you prefer to work in the terminal** and not in VS Code, set up these two (skip this if you use VS Code):
   - Git in the terminal
   - SSH or HTTPS connection to GitHub from terminal
4. **One of these two software environments** (if you are not sure which one to choose or have no preference, choose Conda):
   - Conda environment
   - Virtual environment (Snakemake is not available in this environment)
5. **Optional** and only on Linux: Apptainer following these instructions.

Schedule

Day 1

- 13:00-13:30 - **Welcome and introduction**
  - Practical information (tools, communication, breaks, etc.)
  - Motivation (reproducibility, robustness, distribution, improvement, trust, etc.)
  - Example project: Simulating the motion of planets
- 13:30-14:45 - Introduction to version control with Git and GitHub (1/2)
  - Motivation (15 min)

    - Forking, cloning, and browsing (30 min)
    - Creating branches and commits (30 min)
- 15:00-16:30 - Introduction to version control with Git and GitHub (2/2)
    - Merging changes and contributing to the project (40 min)
    - Conflict resolution (30 min)
    - Practical advice: How much Git is necessary? (20 min)
    - Optional: How to turn your project to a Git repo and share it (30 min)
- 16:45-18:00 - Code documentation

Day 2

- 09:00-10:30 - Collaborative version control and code review (1/2)
    - Concepts around collaboration (10 min)
    - Collaborating within the same repository (40 min)
    - Practicing code review (40 min)
- 10:45-12:15 - Collaborative version control and code review (2/2)
    - How to contribute changes to repositories that belong to others (50 min)
    - Conflict resolution, rebasing, and organizational strategies (40 min)
- 16:45-18:00 - **Debriefing and Q&A**
    - Participants work on their projects
    - Together we study actual codes that participants wrote or work on
    - Constructively we discuss possible improvements
    - Give individual feedback on code projects

Day 3

- 09:00-10:30 - Automated testing
- 10:45-12:15 - How to make the project more reusable
- 13:00-14:45 - Code quality and good practices
    - Concepts in refactoring and modular code design (15 min)
    - Demo: From a script towards a workflow (90 min)
- 15:00-16:30 - How to release and publish your code
    - Choosing a software license (30 min)
    - How to publish your code (15 min)
    - Creating a Python package and deploying it to PyPI (45 min)
- 16:45-18:00 - **Debriefing and Q&A**
    - Participants work on their projects
    - Together we study actual codes that participants wrote or work on
    - Constructively we discuss possible improvements
    - Give individual feedback on code projects

Conda environment

A Conda environment is an isolated software environment that is used to manage dependencies for a project and you decide where it is located.

You will need a `environment.yml` file that documents the dependencies:

```yaml
name: coderefinery
channels:
  - conda-forge
  - bioconda
dependencies:
  - python >= 3.10
  - black
  - click
  - flit
  - ipywidgets
  - isort
  - jupyterlab
  - jupyterlab_code_formatter
  - jupyterlab-git
  - matplotlib
  - myst-parser
  - nbdime
  - numpy
  - pandas
  - pytest
  - pytest-cov
  - scalene
  - seaborn
  - snakemake-minimal
  - sphinx
  - sphinx-autoapi
  - sphinx-autobuild
  - sphinx_rtd_theme >= 2.0
  - vulture
  - scikit-image
```

Before you create a virtual environment

1. Create a new directory for this course.
2. In this directory, create an `environment.yml` file and copy-paste the dependencies above into it.

Choose the tool to manage the environment

If you are already using one of these tools, please continue using the tool that you like and know. If you are new to this, **we recommend using Miniconda or Miniforge**.

- Anaconda
  - Advantages: easy to install, easy to use, good for beginners
  - Disadvantages: large download, installs more than we will need, license restrictions
- Miniconda
  - Advantages: small size, installs only what you need
  - Disadvantages: no graphical interface, license restrictions
- Miniforge
  - Advantages: small size, no license restrictions

    - Disadvantages: no graphical interface
- Micromamba
    - Advantages: fast, small size
    - Disadvantages: no graphical interface
- Pixi
    - Advantages: fast and new
    - Disadvantages: new and less tested and not documented here

Creating the virtual environment

1. Open your terminal shell (e.g. Bash or Zsh).
2. Activate `conda` using `conda activate` or `source ~/miniconda3/bin/activate`.
3. Run the following command:

    ```
    $ conda env create --file environment.yml
    ```

4. Make sure that you see “coderefinery” in the output when you ask for a list of all available environments:

    ```
    $ conda env list
    ```

Virtual environment

A virtual environment is an isolated software environment that is used to manage dependencies for a project and you decide where it is located.

You will need a `requirements.txt` file that documents the dependencies:

```
black
click
flit
ipywidgets
isort
jupyterlab
jupyterlab-code-formatter
jupyterlab-git
matplotlib
myst-parser
nbdime
numpy
pandas
pytest
pytest-cov
scalene
seaborn
sphinx
sphinx-autoapi
sphinx-autobuild
sphinx_rtd_theme >= 2.0
vulture
scikit-image
```

Before you create a virtual environment

1. Create a new directory for this course.
2. In this directory, create a `requirements.txt` file and copy-paste the dependencies above into it.

Creating the virtual environment

Now create a virtual environment in this directory either using pip and venv (more traditional and safer) or using uv (more modern but also less tested):

pip and venv | uv

Create a new virtual environment and activate it:

```
$ python3 -m venv coderefinery-environment
$ source coderefinery-environment/bin/activate
```

Install the dependencies into the environment:

```
$ python3 -m pip install -r requirements.txt
```

Example project: Simulating the motion of planets

The example code that we will study is a hopefully simple N-body simulation written in Python. It is not important or expected that we understand the code in any detail.

The **big picture** is that the code simulates the motion of a number of planets:

- We can choose the number of planets.
- Each planet starts with a random position, velocity, and mass.
- At each time step, the code calculates the gravitational force between each pair of planets.
- The forces accelerate each planet, the acceleration modifies the velocity, the velocity modifies the position of each planet.
- We can choose the number of time steps.
- The units were chosen to make numbers easy to read.

Example run

Instructor note

The instructor demonstrates running the code on their computer.

The code is written to accept **command-line arguments** to specify the number of planets and the number of time steps.

We first generate starting data:

```
$ python generate-data.py --num-planets 10 --output-file initial.csv
```

The generated file (initial.csv) could look like this:

```
px,py,pz,vx,vy,vz,mass
-46.88,-42.51,88.33,-0.86,-0.18,0.55,6.70
-5.29,17.09,-96.13,0.66,0.45,-0.17,3.51
83.53,-92.83,-68.77,-0.26,-0.48,0.24,6.84
-36.31,25.48,64.16,0.85,0.75,-0.56,1.53
-68.38,-17.21,-97.07,0.60,0.26,0.69,6.63
-48.37,-48.74,3.92,-0.92,-0.33,-0.93,8.60
40.53,-75.50,44.18,-0.62,-0.31,-0.53,8.04
-27.21,10.78,-78.82,-0.09,-0.55,-0.03,5.35
88.42,-74.95,-45.85,0.81,0.68,0.56,5.36
39.09,53.12,-59.54,-0.54,0.56,0.07,8.98
```

Then we can simulate their motion (in this case for 20 steps):

```
$ python simulate.py --num-steps 20 \
                     --input-file initial.csv \
                     --output-file final.csv
```

The `--output-file` (final.csv) is again a CSV file (comma-separated values) and contains the final positions of all planets.

It is possible to run on **multiple cores** and to **animate** the result. Here is an example with 100 planets:

```
$ python generate-data.py --num-planets 100 --output-file initial.csv

$ python simulate.py --num-steps 50 \
                     --input-file initial.csv \
                     --output-file final.csv \
                     --trajectories-file trajectories.npz \
                     --num-cores 8

$ python animate.py --initial-file initial.csv \
                    --trajectories-file trajectories.npz \
                    --output-file animation.mp4
```

Learning goals

- What are the most important steps to make this code **reusable by others** and **our future selves**?
- Be able to apply these techniques to your own code/script.

We will not focus on …

- … how the code works internally in detail.
- … whether this is the most efficient algorithm.
- … whether the code is numerically stable.
- … how to code scales with the number of cores.
- … whether it is portable to other operating systems (we will discuss this later).

Introduction to version control with Git and GitHub

Motivation

Objectives

- Browse **commits** and **branches** of a Git repository.
- Remember that commits are like **snapshots** of the repository at a certain point in time.
- Know the difference between **Git** (something that tracks changes) and **GitHub/GitLab** (a web platform to host Git repositories).

Why do we need to keep track of versions?

Version control is an answer to the following questions (do you recognize some of them?):

- “It broke ... hopefully I have a working version somewhere?”
- “Can you please send me the latest version?”
- “Where is the latest version?”
- “Which version are you using?”
- “Which version have the authors used in the paper I am trying to reproduce?”
- “Found a bug! Since when was it there?”
- “I am sure it used to work. When did it change?”
- “My laptop is gone. Is my thesis now gone?”

Demonstration

- Example repository: https://github.com/workshop-material/planets
- Commits are like **snapshots** and if we break something we can go back to a previous snapshot.
- Commits carry **metadata** about changes: author, date, commit message, and a checksum.
- **Branches** are like parallel universes where you can experiment with changes without affecting the default branch: https://github.com/workshop-material/planets/network (“Insights” -> “Network”)
- With version control we can **annotate code** (example).
- **Collaboration**: We can fork (make a copy on GitHub), clone (make a copy to our computer), review, compare, share, and discuss.
- **Code review**: Others can suggest changes using pull requests or merge requests. These can be reviewed and discussed before they are merged. Conceptually, they are similar to “suggesting changes” in Google Docs.

Features: roll-back, branching, merging, collaboration

- **Roll-back**: you can always go back to a previous version and compare
- **Branching and merging**:
  - Work on different ideas at the same time
  - You can experiment with an idea and discard it if it turns out to be a bad idea
  - Different people can work on the same code/project without interfering

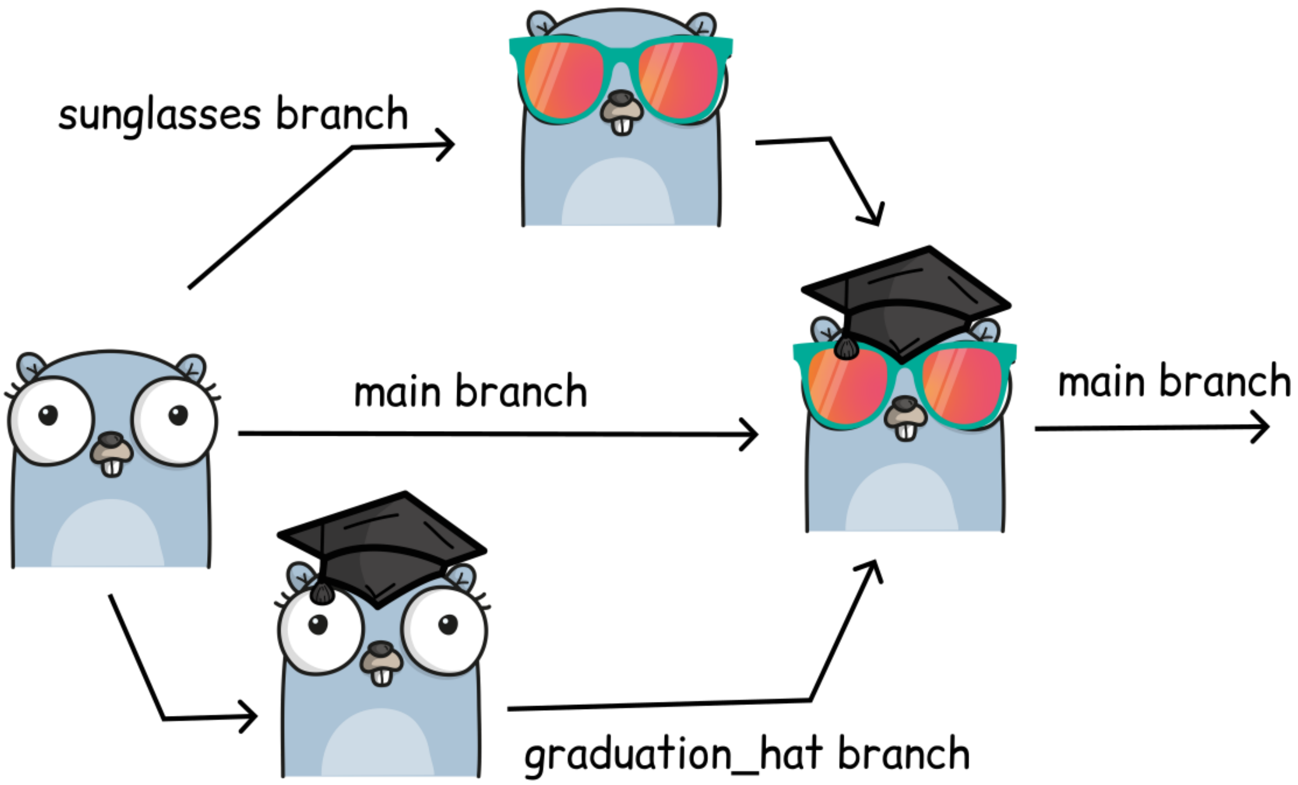


Image created using https://gopherize.me/ (inspiration).

- **Collaboration**: review, compare, share, discuss
- Example network graph

Talking about code

Which of these two is more practical?

1. "Clone the code, go to the file 'simulate.py', and search for 'force_between_planets'. Oh! But make sure you use the version from September 2024."
2. Or I can send you a permalink: https://github.com/workshop-material/planets/blob/1343ac0/simulate.py#L31C5-L39

What we typically like to snapshot

- Software (this is how it started but Git/GitHub can track a lot more)
- Scripts
- Documents (plain text files much better suitable than Word documents)
- Manuscripts (Git is great for collaborating/sharing LaTeX or Quarto manuscripts)
- Configuration files
- Website sources
- Data

💬 Discussion

In this example somebody tried to keep track of versions without a version control system tool like Git. Discuss the following directory listing. What possible problems do you anticipate with this kind of “version control”:

```
myproject-2019.zip
myproject-2020-february.zip
myproject-2021-august.zip
myproject-2023-09-19-working.zip
myproject-2023-09-21.zip
myproject-2023-09-21-test.zip
myproject-2023-09-21-myversion.zip
myproject-2023-09-21-newfeature.zip
...
(100 more files like these)
```

✔ Solution

- Giving a version to a collaborator and merging changes later with own changes sounds like lots of work.
- What if you discover a bug and want to know since when the bug existed?

Forking, cloning, and browsing

In this episode, we will look at an **existing repository** to understand how all the pieces work together. Along the way, we will make a copy (by **forking** and/or **cloning**) of the repository for us, which will be used for our own changes.

❗ Objectives

- See a real Git repository and understand what is inside of it.
- Understand how version control allows advanced inspection of a repository.
- See how Git allows multiple people to work on the same project at the same time.
- **See the big picture** instead of remembering a bunch of commands.

GitHub, VS Code, or command line

We offer **three different paths** for this exercise:

- **GitHub** (this is the one we will demonstrate)
- **VS Code** (if you prefer to follow along using an editor)
- **Command line** (for people comfortable with the command line)

Creating a copy of the repository by “forking” or “cloning”

A **repository** is a collection of files in one directory tracked by Git. A GitHub repository is GitHub's copy, which adds things like access control, issue tracking, and discussions. Each GitHub repository is owned by a user or organization, who controls access.

First, we need to make **our own copy** of the exercise repository. This will become important later, when we make our own changes.

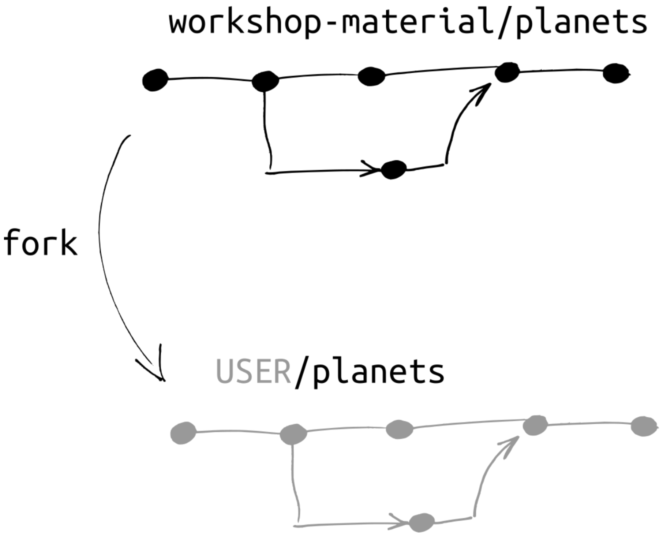


Illustration of forking a repository on GitHub.

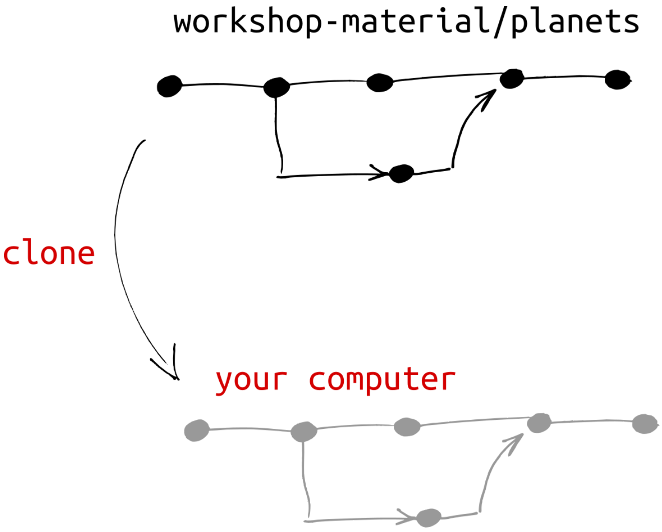
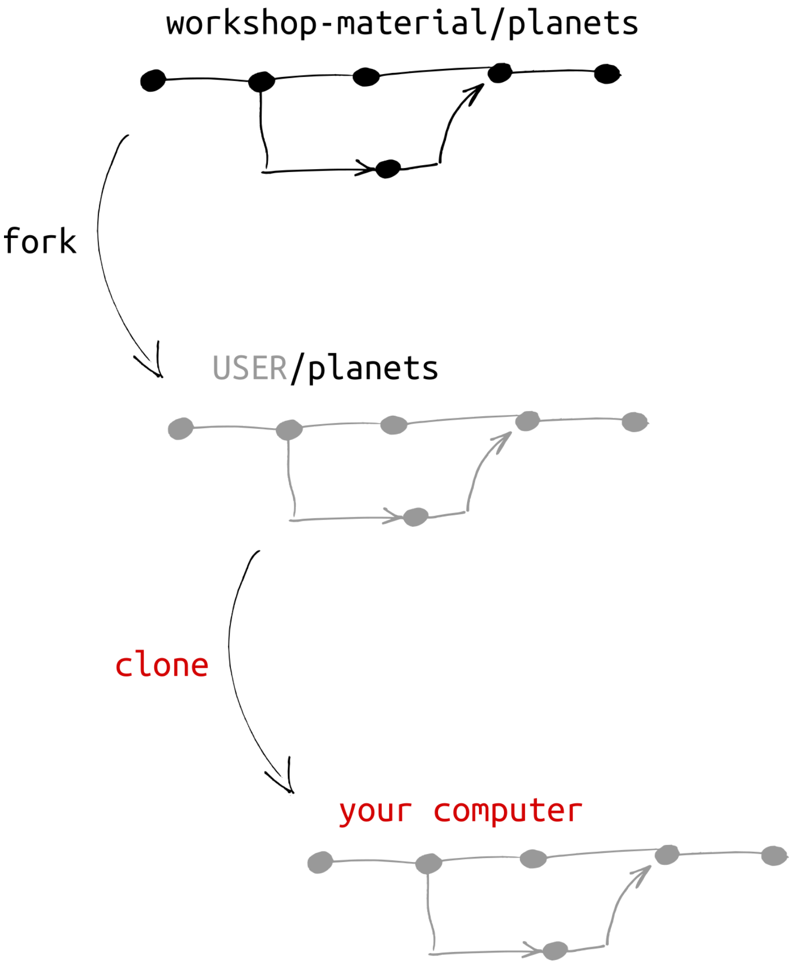


Illustration of cloning a repository to your computer.



It is also possible to do this: to clone a forked repository to your computer.

At all times you should be aware of if you are looking at **your repository** or the **upstream repository** (original repository):

- Your repository: https://github.com/**USER**/planets
- Upstream repository: https://github.com/**workshop-material**/planets

❗ How to create a fork

1. Go to the repository view on GitHub: https://github.com/workshop-material/planets
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 **USER/planets**.

Instructor note

Before starting the exercise session show how to fork the repository to own account (above).

Exercise: Copy and browse an existing project

Work on this by yourself or in pairs.

⚙ Exercise preparation

GitHub VS Code Command line

In this case you will work on a fork.

You only need to open your own view, as described above. The browser URL should look like https://github.com/**USER**/planets, where USER is your GitHub username.

✍️ Exercise: Browsing an existing project (20 min)

Browse the example project and explore commits and branches, either on a fork or on a clone. Take notes and prepare questions. The hints are for the GitHub path in the browser.

1. Browse the **commit history**: Are commit messages understandable? (Hint: “Commit history”, the timeline symbol, above the file list)
2. Compare the commit history with the **network graph** (“Insights” -> “Network”). Can you find the branches?
3. Try to find the **history of commits for a single file**, e.g. `simulate.py`. (Hint: “History” button in the file view)
4. **Which files include the word “position”**? (Hint: the GitHub search on top of the repository view)
5. In the `simulate.py` file, find out who modified the “time step” last and **in which commit**. (Hint: “Blame” view in the file view)
6. Can you use this code yourself? **Are you allowed to share modifications**? (Hint: look for a license file)

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

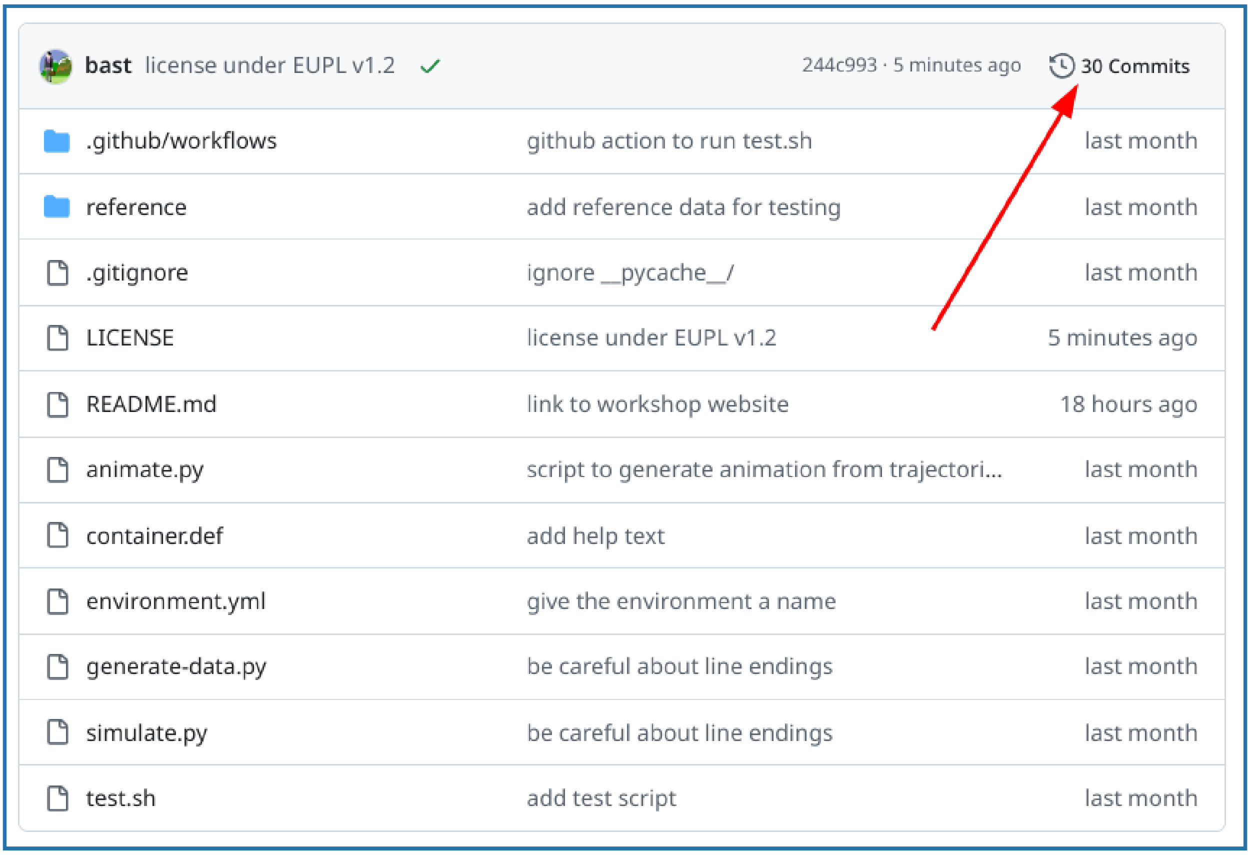
(1) Basic browsing

The most basic thing to look at is the history of commits.

- This is visible from a button in the repository view. We see every change, when, and who has committed.
- Every change has a unique identifier, such as `244c993`. This can be used to identify both this change, and the whole project’s version as of that change.
- Clicking on a change in the view shows more.

GitHub VS Code Command line

Click on the timeline symbol in the repository view:



bast license under EUPL v1.2 ✓ 244c993 · 5 minutes ago 30 Commits

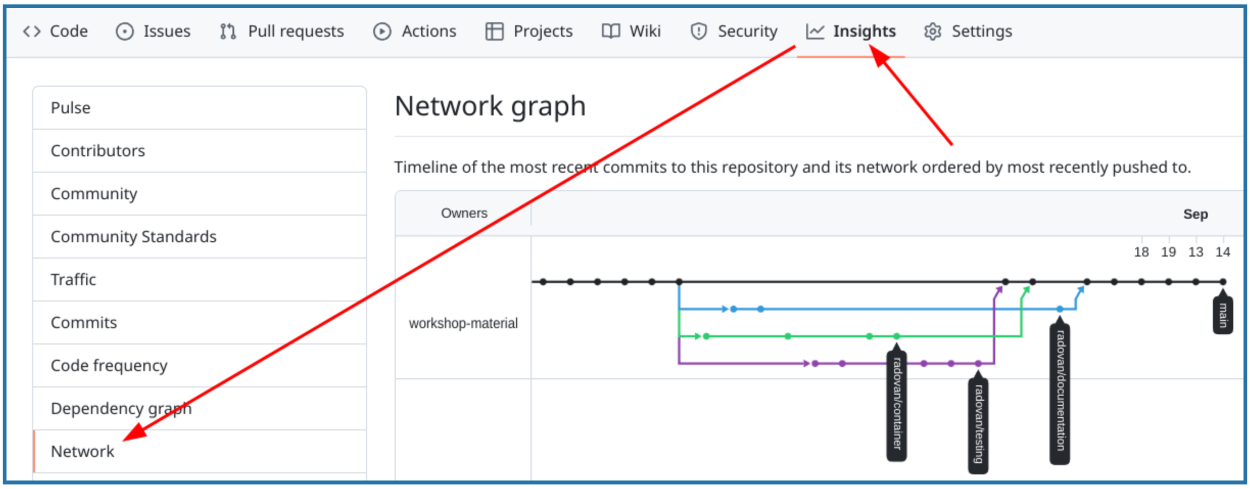
.github/workflows	github action to run test.sh	last month
reference	add reference data for testing	last month
.gitignore	ignore __pycache__/	last month
LICENSE	license under EUPL v1.2	5 minutes ago
README.md	link to workshop website	18 hours ago
animate.py	script to generate animation from trajectori...	last month
container.def	add help text	last month
environment.yml	give the environment a name	last month
generate-data.py	be careful about line endings	last month
simulate.py	be careful about line endings	last month
test.sh	add test script	last month

(2) Compare commit history with network graph

The commit history we saw above looks linear: one commit after another. But if we look at the network view, we see some branching and merging points. We'll see how to do these later. This is another one of the basic Git views.

GitHub VS Code Command line

In a new browser tab, open the "Insights" tab, and click on "Network". You can hover over the commit dots to see the person who committed and how they correspond with the commits in the other view:



Network graph

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

(3) How can you browse the history of a single file?

We see the history for the whole repository, but we can also see it for a single file.

GitHub VS Code Command line

Navigate to the file view: Main page → simulate.py. Click the “History” button near the top right.

(4) Which files include the word “position”?

Version control makes it very easy to find all occurrences of a word or pattern. This is useful for things like finding where functions or variables are defined or used.

GitHub VS Code Command line

We go to the main file view. We click the Search magnifying class at the very top, type “position”, and click enter. We see every instance, including the context.

! Searching in a forked repository will not work instantaneously!

It usually takes a few minutes before one can search for keywords in a forked repository since it first needs to build the search index the very first time we search. Start it, continue with other steps, then come back to this.

(5) Who modified a particular line last and when?

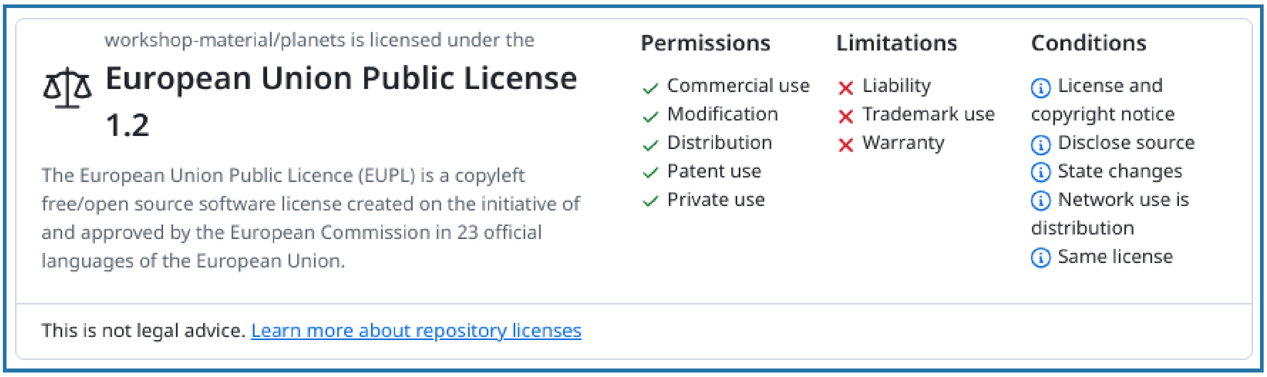
This is called the “annotate” or “blame” view. The name “blame” is very unfortunate, but it is the standard term for historical reasons for this functionality and it is not meant to blame anyone.

GitHub | VS Code | Command line

From a file view, change preview to “Blame” towards the top-left. To get the actual commit, click on the commit message next to the code line that you are interested in.

(6) Can you use this code yourself? Are you allowed to share modifications?

- Look at the file `LICENSE`.
- On GitHub, click on the file to see a nice summary of what we can do with this:



workshop-material/planets is licensed under the

European Union Public License 1.2

The European Union Public Licence (EUPL) is a copyleft free/open source software license created on the initiative of and approved by the European Commission in 23 official languages of the European Union.

Permissions	Limitations	Conditions
✓ Commercial use	✗ Liability	ⓘ License and copyright notice
✓ Modification	✗ Trademark use	ⓘ Disclose source
✓ Distribution	✗ Warranty	ⓘ State changes
✓ Patent use		ⓘ Network use is distribution
✓ Private use		ⓘ Same license

This is not legal advice. Learn more about repository licenses

Summary

- Git allowed us to understand this simple project much better than we could, if it was just a few files on our own computer.
- It was easy to share the project with the course.
- By forking the repository, we created our own copy. This is important for the following, where we will make changes to our copy.

Creating branches and commits

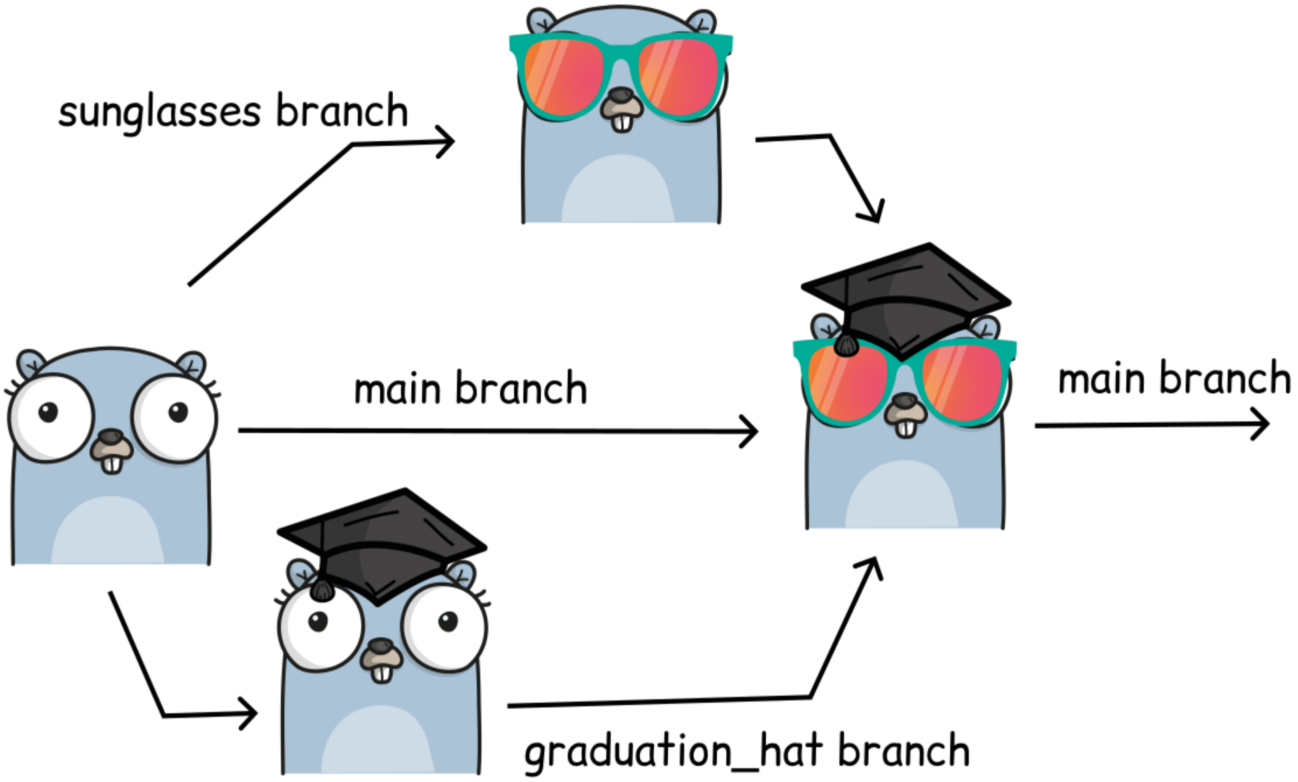
The first and most basic task to do in Git is **record changes** using commits. In this part, we will record changes in two ways: on a **new branch** (which supports multiple lines of work at once), and directly on the “main” branch (which happens to be the default branch here).

! Objectives

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

Background

- In the previous episode we have browsed an existing **repository** and saw **commits** and **branches**.
- Each **commit** is a snapshot of the entire project at a certain point in time and has a unique identifier (**hash**) .
- A **branch** is a line of development, and the `main` branch or `master` branch are often the default branch in Git.
- A branch in Git is like a **sticky note that is attached to a commit**. When we add new commits to a branch, the sticky note moves to the new commit.
- **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.



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: Creating branches and commits

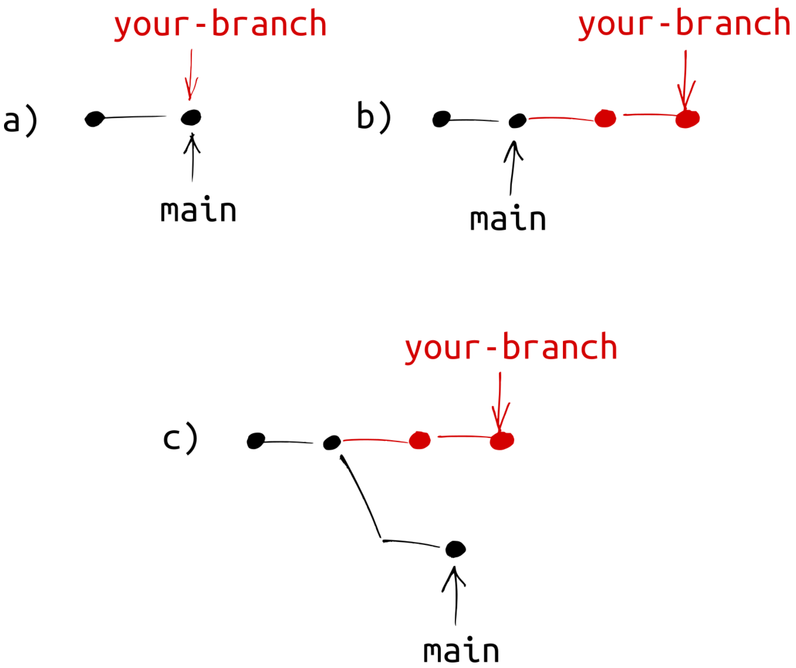


Illustration of what we want to achieve in this exercise.

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

1. First create a new branch and then either add a new file or modify an existing file and commit the change. Make sure that you now work **on your copy** of the example repository. In your new commit you can share a Git or programming trick you like.
2. In a new commit, modify the file again.
3. Switch to the `main` branch and create a commit there.
4. Browse the network and locate the commits that you just created (“Insights” -> “Network”).
5. Compare the branch that you created with the `main` branch. Can you find an easy way to see the differences?
6. Can you find a way to compare versions between two arbitrary commits in the repository?
7. Try to rename the branch that you created and then browse the network again.
8. 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) Create a new branch and a new commit

GitHub VS Code Command line

1. Where it says “main” at the top left, click, enter a new branch name (e.g. `new-tutorial`), then click on “Create branch … from main”.

2. Make sure you are still on the `new-tutorial` branch (it should say it at the top), and click “Add file” → “Create new file” from the upper right.
3. Enter a filename where it says “Name your file…”.
4. Share some Git or programming trick you like.
5. Click “Commit changes”
6. Enter a commit message. Then click “Commit changes”.

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

(2) Modify the file again 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 the file you added or modified previously.
2. Click the edit button, the pencil icon at top-right.
3. Follow the “Commit changes” instructions as in the previous step.

(3) Switch to the main branch and create a commit 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 file that already exists, following the pattern from above.

(4) Browse the commits you just made

Let’s look at what we did. Now, the `main` and the new 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).

(5) 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

A nice way to compare branches is to add `/compare` to the URL of the repository, for example (replace USER): https://github.com/**USER**/planets/compare

(6) Compare two arbitrary commits

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

GitHub VS Code Command line

Following the `/compare`-trick above, one can compare commits on GitHub by adjusting the following URL: https://github.com/**USER**/planets/compare/**VERSION1**..**VERSION2**

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

(7) Renaming a branch

GitHub VS Code Command line

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

(8) 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

On the right side, below "Releases", click on "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.

Summary

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.
- In Git, commits form a so-called "graph". Branches are tags in Git function like sticky notes that stick to specific commits. What this means for us is that it does not cost any significant disk space to create new branches.
- Not all files should be added to Git. For example, temporary files or files with sensitive information or files which are generated as part of the build process should not be added to Git. For this we use `.gitignore` (more about this later: Practical advice: How much Git is necessary?).
- Unsure on which branch you are or what state the repository is in? On the command line, use `git status` frequently to get a quick overview.

Merging changes and contributing to the project

Git allows us to have different development lines where we can try things out. It also allows different people to work on the same project at the same. This means that we have to somehow combine the changes later. In this part we will practice this: **merging**.

! Objectives

- Understand that on GitHub merging is done through a **pull request** (on GitLab: "merge request"). Think of it as a **change proposal**.
- Create and merge a pull request within your own repository.
- Understand (and optionally) do the same across repositories, to contribute to the upstream public repository.

Exercise

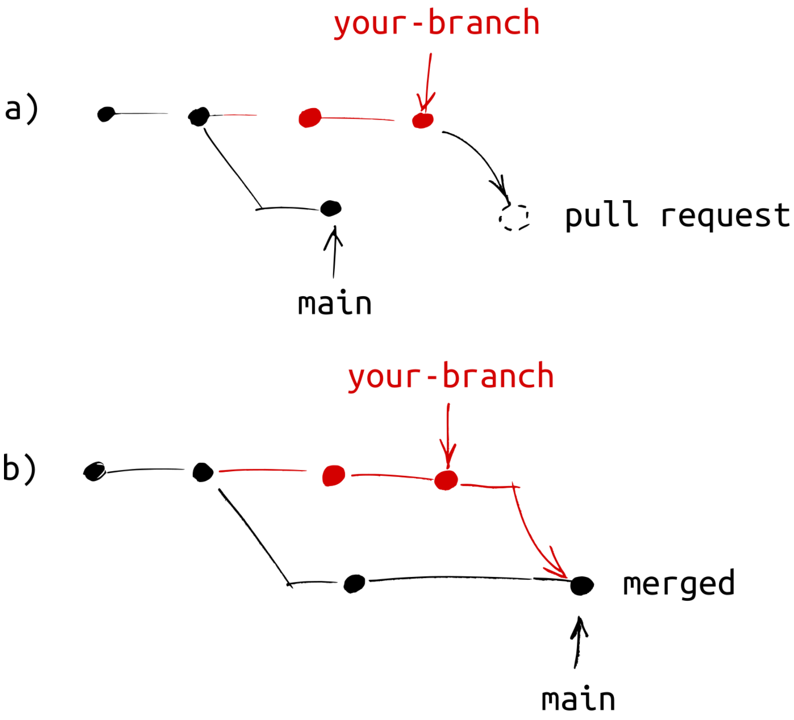


Illustration of what we want to achieve in this exercise.

✍️ Exercise: Merging branches

GitHub Local (VS Code, Command line)

First, we make something called a pull request, which allows review and commenting before the actual merge.

We assume that in the previous exercise you have created a new branch with one or few new commits. We provide basic hints. You should refer to the solution as needed.

1. Navigate to your branch from the previous episode (hint: the same branch view we used last time).
2. Begin the pull request process (hint: There is a "Contribute" button in the branch view).
3. Add or modify the pull request title and description, and verify the other data. In the pull request verify the target repository and the target branch. Make sure that you are merging within your own repository. **GitHub: By default, it will offer to make the change to the upstream repository, `workshop-material`. You should change this**, you shouldn't contribute your commit(s) upstream yet. Where it says `base repository`, select your own repository.
4. Create the pull request by clicking "Create pull request". Browse the network view to see if anything has changed yet.
5. Merge the pull request, or if you are not on GitHub you can merge the branch locally. Browse the network again. What has changed?

6. Find out which branches are merged and thus safe to delete. Then remove them and verify that the commits are still there, only the branch labels are gone (hint: you can delete branches that have been merged into `main`).
7. Optional: Try to create a new branch with a new change, then open a pull request but towards the original (upstream) repository. We will later merge few of those.

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.

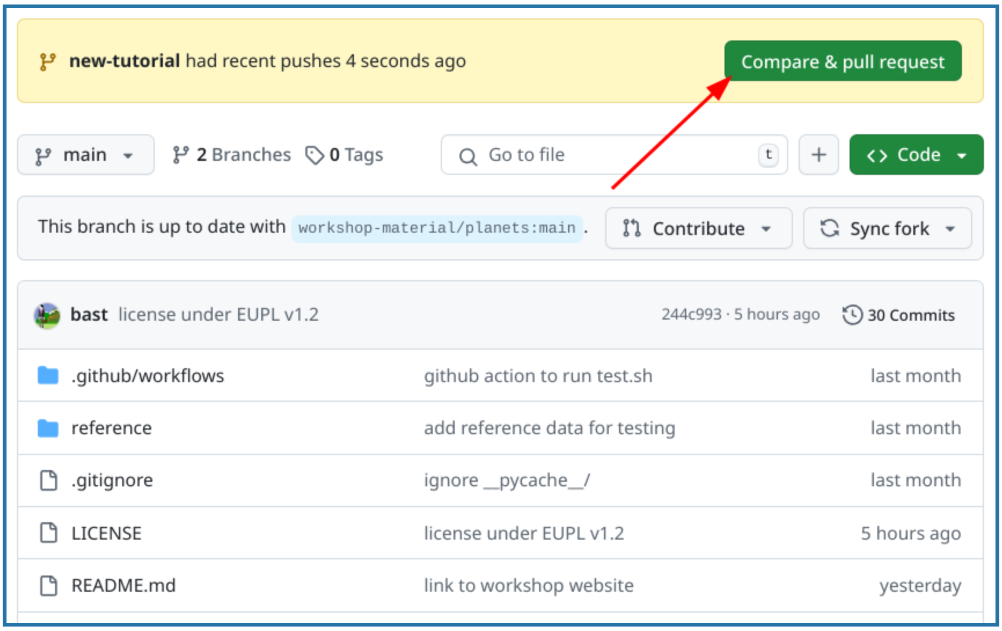
Solution and walk-through

(1) Navigate to your branch

Before making the pull request, or doing a merge, it's important to make sure that you are on the right branch. Many people have been frustrated because they forgot this!

GitHub VS Code Command line

GitHub will notice a recently changed branch and offer to make a pull request (clicking there will bring you to step 3):



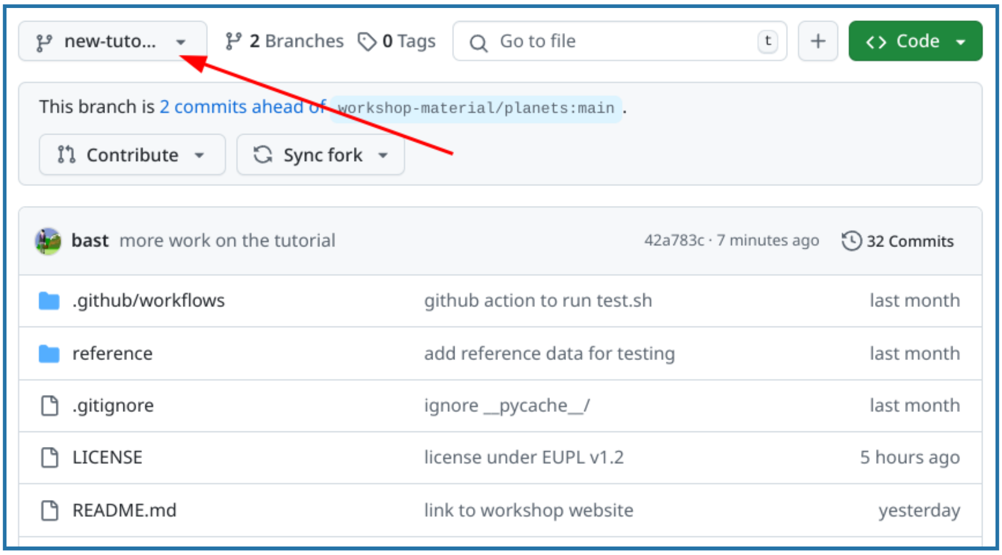
new-tutorial had recent pushes 4 seconds ago

This branch is up to date with `workshop-material/planets:main`.

bast license under EUPL v1.2 244c993 · 5 hours ago 30 Commits

.github/workflows	github action to run test.sh	last month
reference	add reference data for testing	last month
.gitignore	ignore __pycache__/	last month
LICENSE	license under EUPL v1.2	5 hours ago
README.md	link to workshop website	yesterday

If the yellow box is not there, make sure you are on the branch you want to merge **from**:



new-tuto... · 2 Branches · 0 Tags · Go to file · t · + · Code

This branch is 2 commits ahead of `workshop-material/planets:main` .

Contribute · Sync fork

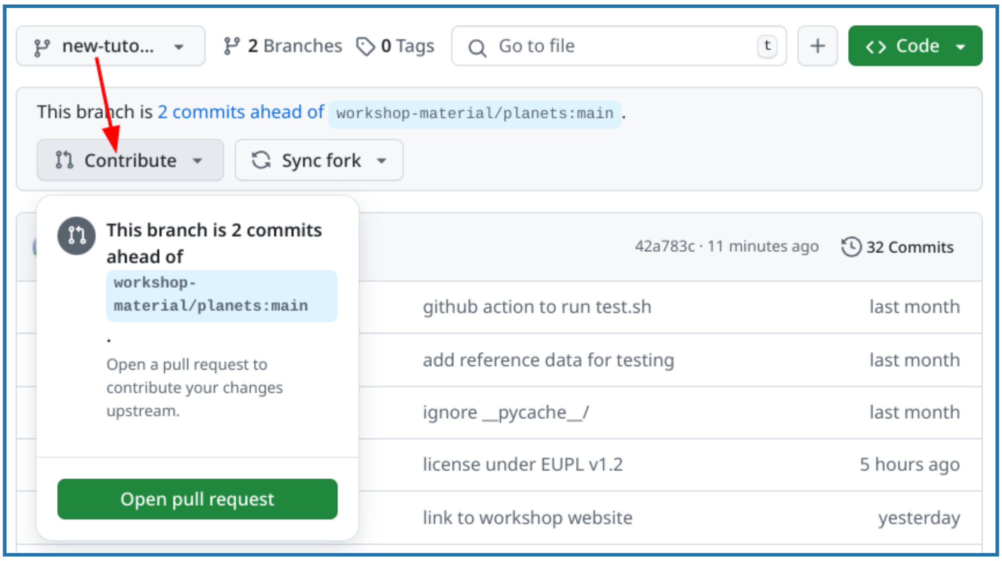
bast more work on the tutorial · 42a783c · 7 minutes ago · 32 Commits

.github/workflows	github action to run test.sh	last month
reference	add reference data for testing	last month
.gitignore	ignore __pycache__/	last month
LICENSE	license under EUPL v1.2	5 hours ago
README.md	link to workshop website	yesterday

(2) Begin the pull request process

In GitHub, the pull request is the way we propose to merge two branches together. We start the process of making one.

GitHub VS Code Command line



new-tuto... · 2 Branches · 0 Tags · Go to file · t · + · Code

This branch is 2 commits ahead of `workshop-material/planets:main` .

Contribute · Sync fork

This branch is 2 commits ahead of `workshop-material/planets:main` .

Open a pull request to contribute your changes upstream.

Open pull request

42a783c · 11 minutes ago · 32 Commits

github action to run test.sh	last month
add reference data for testing	last month
ignore __pycache__/	last month
license under EUPL v1.2	5 hours ago
link to workshop website	yesterday

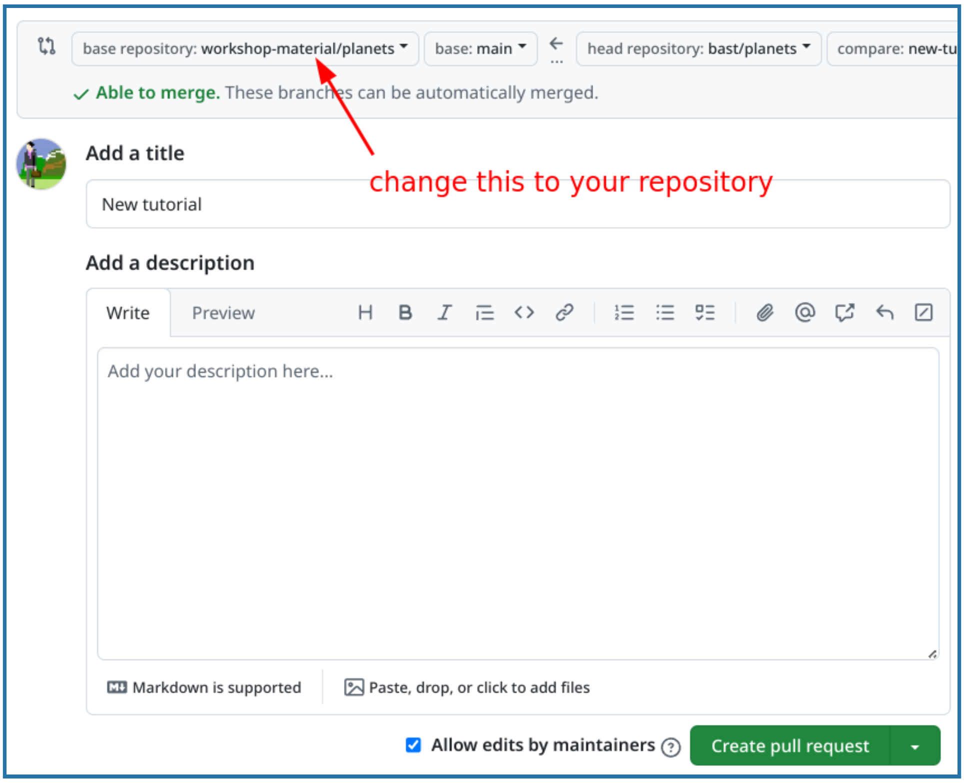
(3) Fill out and verify the pull request

Check that the pull request is directed to the right repository and branch and that it contains the changes that you meant to merge.

GitHub VS Code Command line

Things to check:

- Base repository: this should be your own
- Title: make it descriptive
- Description: make it informative
- Scroll down to see commits: are these the ones you want to merge?
- Scroll down to see the changes: are these the ones you want to merge?



base repository: workshop-material/planets base: main ← head repository: bast/planets compare: new-tu

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

change this to your repository

Add a title

New tutorial

Add a description

Add your description here...

Write
Preview
Markdown is supported
Paste, drop, or click to add files
☑ Allow edits by maintainers
Create pull request

This screenshot only shows the top part. If you scroll down, you can see the commits and the changes. We recommend to do this before clicking on "Create pull request".

(4) Create the pull request

We actually create the pull request. Don't forget to navigate to the Network view after opening the pull request. Note that the changes proposed in the pull request are not yet merged.

GitHub VS Code Command line

Click on the green button "Create pull request".

If you click on the little arrow next to "Create pull request", you can also see the option to "Create draft pull request". This will be interesting later when collaborating with others. It allows you to open a pull request that is not ready to be merged yet, but you want to show it to others and get feedback.

(5) Merge the pull request

Now, we do the actual merging. We see some effects now.

GitHub VS Code Command line

Review it again (commits and changes), and then click “Merge pull request”.

After merging, verify the network view. Also navigate then to your “main” branch and check that your change is there.

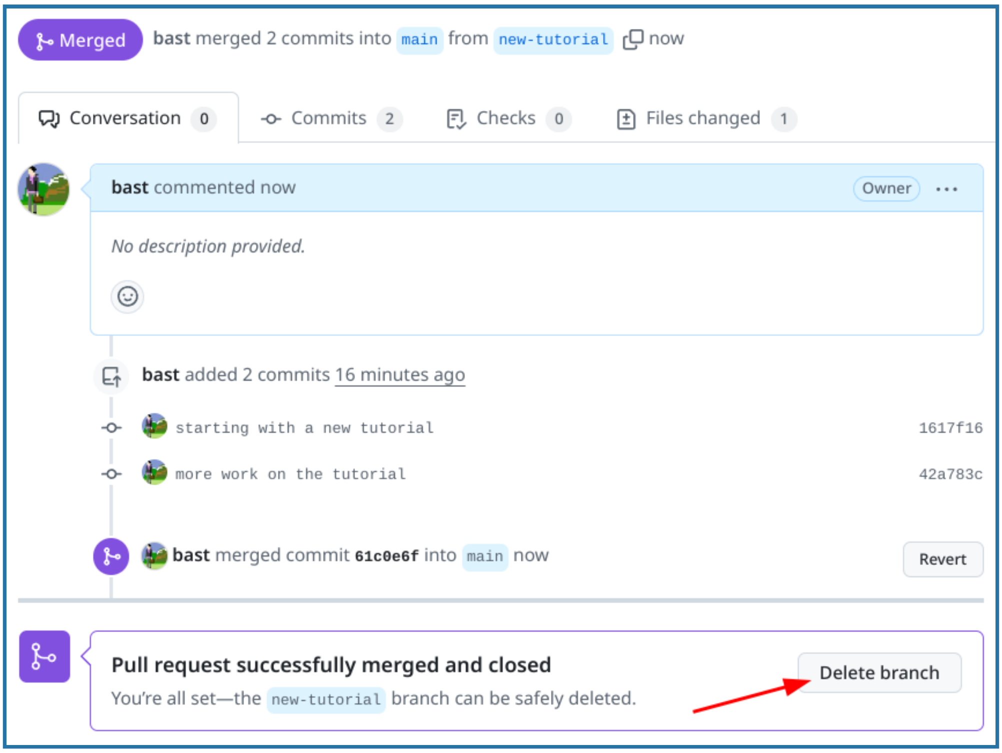
(6) Delete merged branches

Before deleting branches, first check whether they are merged.

If you delete an un-merged branch, it will be difficult to find the commits that were on that branch. If you delete a merged branch, the commits are now also part of the branch where we have merged to.

GitHub VS Code Command line

One way to delete the branch is to click on the “Delete branch” button after the pull request is merged:



Merged bast merged 2 commits into `main` from `new-tutorial` now

Conversation 0 Commits 2 Checks 0 Files changed 1

bast commented now Owner

No description provided.

bast added 2 commits 16 minutes ago

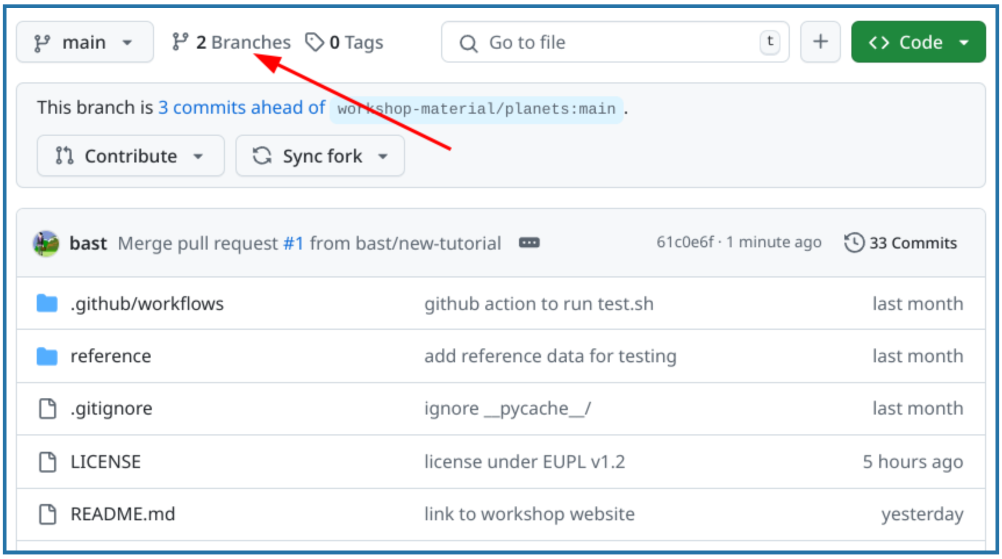
```
starting with a new tutorial    1617f16
more work on the tutorial       42a783c
```

bast merged commit `61c0e6f` into `main` now Revert

Pull request successfully merged and closed

You’re all set—the `new-tutorial` branch can be safely deleted.

But what if we forgot? Then navigate to the branch view:



This branch is 3 commits ahead of `workshop-material/planets:main` .

bast Merge pull request #1 from bast/new-tutorial 61c0e6f · 1 minute ago 33 Commits

.github/workflows	github action to run test.sh	last month
reference	add reference data for testing	last month
.gitignore	ignore __pycache__/	last month
LICENSE	license under EUPL v1.2	5 hours ago
README.md	link to workshop website	yesterday

In the overview we can see that it has been merged and we can delete it.

(7) Contribute to the original repository with a pull request

This is an advanced step. We will practice this tomorrow and it is OK to skip this at this stage.

GitHub VS Code Command line

Now that you know how to create branches and opening a pull request, try to open a new pull request with a new change but this time the base repository should be the upstream one.

In other words, you now send a pull request **across repositories**: from your fork to the original repository.

Another thing that is different now is that you might not have permissions to merge the pull request. We can then together review and browse the pull request.

Summary

- We learned how to merge two branches together.
- When is this useful? This is not only useful to combine development lines in your own work. Being able to merge branches also **forms the basis for collaboration**.
- Branches which are merged to other branches are safe to delete, since we only delete the “sticky note” next to a commit, not the commits themselves.

Conflict resolution

Resolving a conflict (demonstration)

A conflict is when Git asks humans to decide during a merge which of two changes to keep **if the same portion of a file has been changed in two different ways on two different branches**.

We will practice conflict resolution in the collaborative Git lesson (next day).

Here we will only demonstrate how to create a conflict and how to resolve it, **all on GitHub**. Once we understand how this works, we will be more confident to resolve conflicts also in the **command line** (we can demonstrate this if we have time).

How to create a conflict (please try this in your own time *and just watch now*):

- Create a new branch from `main` and on it make a change to a file.
- On `main`, make a different change to the same part of the same file.
- Now try to merge the new branch to `main`. You will get a conflict.

How to resolve conflicts:

- On GitHub, you can resolve conflicts by clicking on the "Resolve conflicts" button. This will open a text editor where you can choose which changes to keep. Make sure to remove the conflict markers. After resolving the conflict, you can commit the changes and merge the pull request.
- Sometimes a conflict is between your change and somebody else's change. In that case, you might have to discuss with the other person which changes to keep.

Avoiding conflicts

💬 The human side of conflicts

- What does it mean if two people do the same thing in two different ways?
- What if you work on the same file but do two different things in the different sections?
- What if you do something, don't tell someone from 6 months, and then try to combine it with other people's work?
- How are conflicts avoided in other work? (Only one person working at once? Declaring what you are doing before you start, if there is any chance someone else might do the same thing, helps.)

- Human measures
  - Think and plan to which branch you will commit to.
  - Do not put unrelated changes on the same branch.
- Collaboration measures
  - Open an issue and discuss with collaborators before starting a long-living branch.
- Project layout measures
  - Modifying global data often causes conflicts.
  - Modular programming reduces this risk.

- Technical measures
    - **Share your changes early and often**: This is one of the happy, rare circumstances when everyone doing the selfish thing (publishing your changes as early as practical) results in best case for everyone!
    - Pull/rebase often to keep up to date with upstream.
    - Resolve conflicts early.

Practical advice: How much Git is necessary?

Writing useful commit messages

Useful commit messages **summarize the change and provide context**.

If you need a commit message that is longer than one line, then the convention is: one line summarizing the commit, then one empty line, then paragraph(s) with more details in free form, if necessary.

Good example:

```
increase alpha to 2.0 for faster convergence

the motivation for this change is
to enable ...
...
(more context)
...
this is based on a discussion in #123
```

- **Why something was changed is more important than what has changed.**
- Cross-reference to issues and discussions if possible/relevant.
- Bad commit messages: “fix”, “oops”, “save work”
- Bad examples: http://whatthecommit.com
- Write commit messages that will be understood 15 years from now by someone else than you. Or by your future you.
- Many projects start out as projects “just for me” and end up to be successful projects that are developed by 50 people over decades.
- Commits with multiple authors are possible.

Good references:

- “My favourite Git commit”
- “On commit messages”
- “How to Write a Git Commit Message”

Note

A great way to learn how to write commit messages and to get inspired by their style choices: **browse repositories of codes that you use/like**:

Some examples (but there are so many good examples):

- SciPy
- NumPy
- Pandas
- Julia
- ggplot2, compare with their release notes
- Flask, compare with their release notes

When designing commit message styles consider also these:

- How will you easily generate a changelog or release notes?
- During code review, you can help each other improving commit messages.

But remember: it is better to make any commit, than no commit. Especially in small projects. **Let not the perfect be the enemy of the good enough**.

What level of branching complexity is necessary for each project?

Simple personal projects:

- Typically start with just the `main` branch.
- Use branches for unfinished/untested ideas.
- Use branches when you are not sure about a change.
- Use tags to mark important milestones.
- If you are unsure what to do with unfinished and not working code, commit it to a branch.

Projects with few persons: you accept things breaking sometimes

- It might be reasonable to commit to the `main` branch and feature branches.

Projects with few persons: changes are reviewed by others

- You create new feature branches for changes.
- Changes are reviewed before they are merged to the `main` branch.
- Consider to write-protect the `main` branch so that it can only be changed with pull requests or merge requests.

How large should a commit be?

- Better too small than too large (easier to combine than to split).

- Often I make a commit at the end of the day (this is a unit I would not like to lose).
- Smaller sized commits may be easier to review for others than huge commits.
- A commit should not contain unrelated changes to simplify review and possible repair/adjustments/undo later (but again: imperfect commits are better than no commits).
- Imperfect commits are better than no commits.

Working on the command line? Use "git status" all the time

The `git status` command is one of the most useful commands in Git to inform about which branch we are on, what we are about to commit, which files might not be tracked, etc.

How about staging and committing?

- Commit early and often: rather create too many commits than too few. You can always combine commits later.
- Once you commit, it is very, very hard to really lose your code.
- Always fully commit (or stash) before you do dangerous things, so that you know you are safe. Otherwise it can be hard to recover.
- Later you can start using the staging area (where you first stage and then commit in a second step).
- Later start using `git add -p` and/or `git commit -p`.

What to avoid

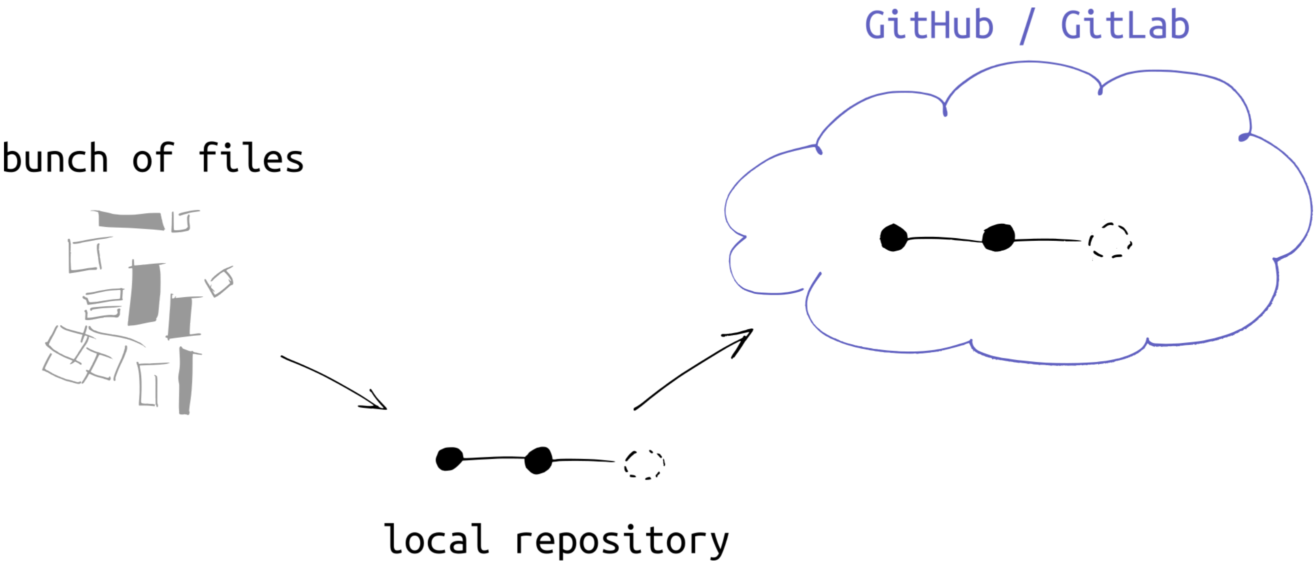
- Committing **generated files/directories** (example: `__pycache__`, `*.pyc`) -> use .gitignore files (collection of .gitignore templates).
- Committing **huge files** -> use code review to detect this.
- Committing **unrelated changes** together.
- Postponing commits because the changes are "unfinished"/"ugly" -> better ugly commits than no commits.
- When working with branches:
  - Working on unrelated things on the same branch.
  - Not updating your branch before starting new work.
  - Too ambitious branch which risks to never get completed.
  - Over-engineering the branch layout and safeguards in small projects -> can turn people away from your project.

Optional: How to turn your project to a Git repo and share it

Objectives

- Turn our own coding project (small or large, finished or unfinished) into a Git repository.
- Be able to share a repository on the web to have a backup or so that others can reuse and collaborate or even just find it.

Exercise



From a bunch of files to a local repository which we then share on GitHub.

✍️ Exercise: Turn your project to a Git repo and share it (20 min)

1. Create a new directory called **myproject** (or a different name) with one or few files in it. This represents our own project. It is not yet a Git repository. You can try that with your own project or use a simple placeholder example.
2. Turn this new directory into a Git repository.
3. Share this repository on GitHub (or GitLab, since it really works the same).

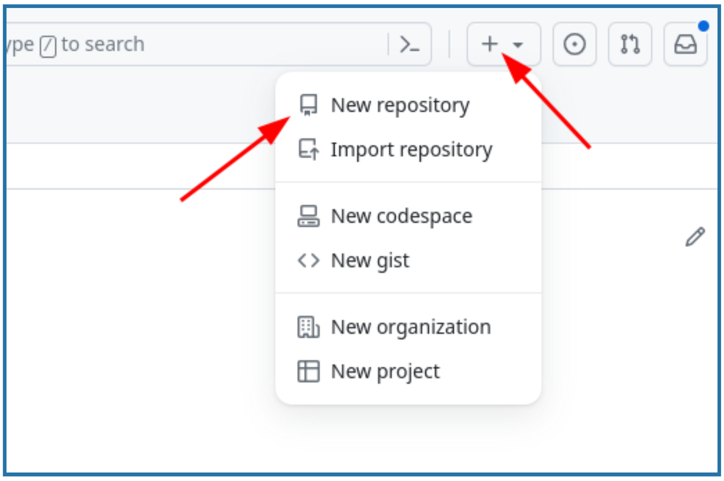
We offer **three different paths** of how to do this exercise.

- Via **GitHub web interface**: easy and can be a good starting point if you are completely new to Git.
- **VS Code** is quite easy, since VS Code can offer to create the GitHub repositories for you.
- **Command line**: you need to create the repository on GitHub and link it yourself.

Only using GitHub | VS Code | Command line | RStudio

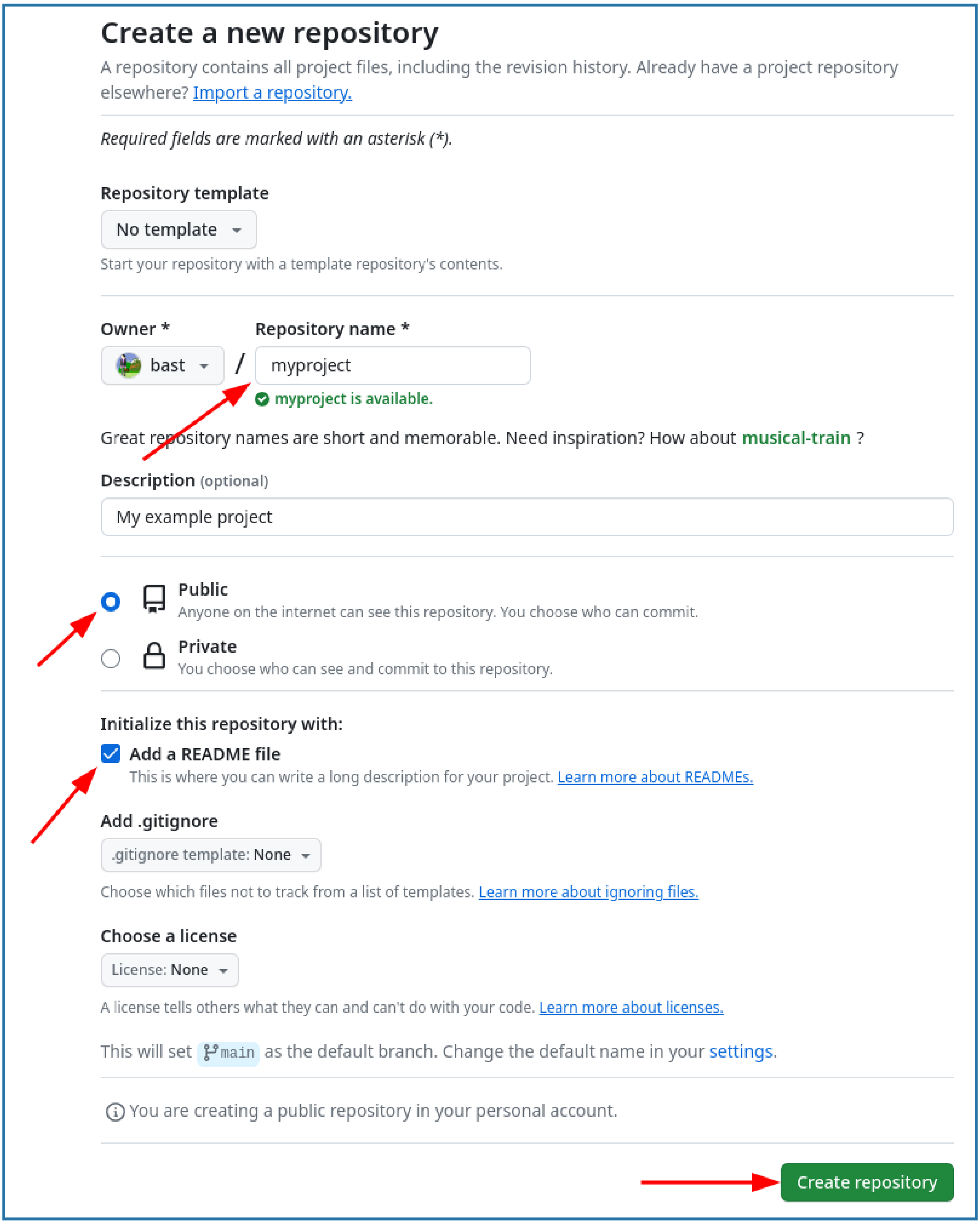
Create an repository on GitHub

First log into GitHub, then follow the screenshots and descriptions below.



Click on the "plus" symbol on top right, then on "New repository".

Then:



Create a new repository

A repository contains all project files, including the revision history. Already have a project repository elsewhere? Import a repository.

Required fields are marked with an asterisk (*).

Repository template

No template

Start your repository with a template repository's contents.

Owner * bast / **Repository name *** myproject

myproject is available.

Great repository names are short and memorable. Need inspiration? How about musical-train ?

Description (optional)

My example project

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

○ **Private**
You choose who can see and commit to this repository.

Initialize this repository with:

☑ **Add a README file**
This is where you can write a long description for your project. Learn more about READMEs.

Add .gitignore

.gitignore template: None

Choose which files not to track from a list of templates. Learn more about ignoring files.

Choose a license

License: None

A license tells others what they can and can't do with your code. Learn more about licenses.

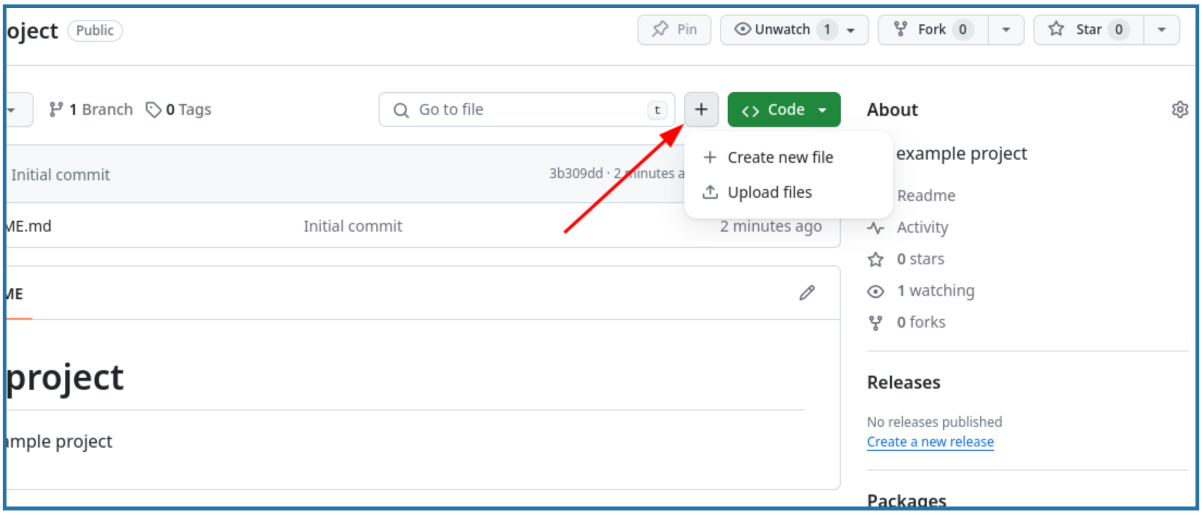
This will set `main` as the default branch. Change the default name in your settings.

You are creating a public repository in your personal account.

Choose a repository name, add a short description, and in this case ***make sure to check*** *“Add a README file”. Finally “Create repository”.*

Upload your files

Now that the repository is created, you can upload your files:



Click on the “+” symbol and then on “Upload files”.

Is putting software on GitHub/GitLab/… publishing?

It is a good first step but to make your code truly **findable and accessible**, consider making your code **citable and persistent**: Get a persistent identifier (PID) such as DOI in addition to sharing the code publicly, by using services like Zenodo or similar services.

Code documentation

❗ Objectives

- Discuss what makes good documentation.
- Improve the README of your project or our example project.
- Explore Sphinx which is a popular tool to build documentation websites.
- Learn how to leverage GitHub Actions and GitHub Pages to build and deploy documentation.

Instructor note

- (30 min) Discussion
- (30 min) Exercise: Set up a Sphinx documentation and add API documentation
- (15 min) Demo: Building documentation with GitHub Actions

Why? 💗✉️ to your future self

- You will probably use your code in the future and may forget details.

- You may want others to use your code or contribute (almost impossible without documentation).

In-code documentation

Not very useful (more commentary than comment):

```
# now we check if temperature is below -50
if temperature < -50:
    print("ERROR: temperature is too low")
```

More useful (explaining **why**):

```
# we regard temperatures below -50 degrees as measurement errors
if temperature < -50:
    print("ERROR: temperature is too low")
```

Keeping zombie code “just in case” (rather use version control):

```
# do not run this code!
# if temperature > 0:
#     print("It is warm")
```

Emulating version control:

```
# John Doe: threshold changed from 0 to 15 on August 5, 2013
if temperature > 15:
    print("It is warm")
```

Many languages allow “docstrings”

Example (Python):

```
def kelvin_to_celsius(temp_k: float) -> float:
    """
    Converts temperature in Kelvin to Celsius.

    Parameters
    ----------
    temp_k : float
        temperature in Kelvin

    Returns
    -------
    temp_c : float
        temperature in Celsius
    """
    assert temp_k >= 0.0, "ERROR: negative T_K"

    temp_c = temp_k - 273.15

    return temp_c
```

Keypoints

- Documentation which is only in the source code is not enough.
- Often a README is enough.
- Documentation needs to be kept **in the same Git repository** as the code since we want it to evolve with the code.

Often a README is enough - checklist

- **Purpose**
- Requirements
- Installation instructions
- **Copy-paste-able example to get started**
- Tutorials covering key functionality
- Reference documentation (e.g. API) covering all functionality
- Authors and **recommended citation**
- License
- Contribution guide

See also the JOSS review checklist.

Diátaxis

Diátaxis is a systematic approach to technical documentation authoring.

- Overview: https://diataxis.fr/
- How to use Diátaxis **as a guide** to work: https://diataxis.fr/how-to-use-diataxis/

What if you need more than a README?

- Write documentation in Markdown (.md) or reStructuredText (.rst) or R Markdown (.Rmd)
- In the **same repository** as the code -> version control and **reproducibility**
- Use one of many tools to build HTML out of md/rst/Rmd: Sphinx, MkDocs, Zola, Jekyll, Hugo, RStudio, knitr, bookdown, blogdown, ...
- Deploy the generated HTML to GitHub Pages or GitLab Pages

Exercise: Set up a Sphinx documentation

⚙ Preparation

In this episode we will use the following 5 packages which we installed previously as part of the Conda environment or Virtual environment:

```
myst-parser
sphinx
sphinx-rtd-theme
sphinx-autoapi
sphinx-autobuild
```

Which repository to use? You have 3 options:

- Clone **your fork** of the planets example repository.
- If you don’t have that, you can clone the original repository itself: https://github.com/workshop-material/planets
- You can try this with **your own project** and the project does not have to be a Python project.

There are at least two ways to get started with Sphinx:

1. Use `sphinx-quickstart` to create a new Sphinx project.
2. **This is what we will do instead**: Create three files (`doc/conf.py` , `doc/index.md` , and `doc/about.md`) as starting point and improve from there.

✍️ Exercise: Set up a Sphinx documentation

1. Create the following three files in your project:

   ```
   planets/      <- or your own project
   ├── doc/
   │   ├── conf.py
   │   ├── index.md
   │   └── about.md
   └── ...
   ```

   This is `conf.py` :

```python
project = "planets"
copyright = "2024, Authors"
author = "Authors"
release = "0.1"

exclude_patterns = ["_build", "Thumbs.db", ".DS_Store"]

extensions = [
    "myst_parser",  # in order to use markdown
]

myst_enable_extensions = [
    "colon_fence",  # ::: can be used instead of ``` for better rendering
]

html_theme = "sphinx_rtd_theme"
```

This is `index.md` (feel free to change the example text):

```markdown
# Our code documentation

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor
incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis
nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat.
Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu
fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in
culpa qui officia deserunt mollit anim id est laborum.

:::{toctree}
:maxdepth: 2
:caption: Some caption

about.md
:::
```

This is `about.md` (feel free to adjust):

```markdown
# About this code

Work in progress ...
```

2. Run `sphinx-build` to build the HTML documentation:

```console
$ sphinx-build doc _build

... lots of output ...
The HTML pages are in _build.
```

3. Try to open `_build/index.html` in your browser.
4. Experiment with adding more content, images, equations, code blocks, ...

   - typography

- images
- math and equations
- code blocks

There is a lot more you can do:

- This is useful if you want to check the integrity of all internal and external links:

  ```
  $ sphinx-build doc -W -b linkcheck _build
  ```

- sphinx-autobuild provides a local web server that will automatically refresh your view every time you save a file - which makes writing with live-preview much easier.

Demo: Building documentation with GitHub Actions

Instructor note

- Instructor presents.
- Learners are encouraged to try this later on their own.

First we need to extend the `environment.yml` file to include the necessary packages:

```
name: planets
channels:
  - conda-forge
dependencies:
  - python=3.12
  - numpy
  - click
  - matplotlib
  - myst-parser
  - sphinx
  - sphinx-rtd-theme
  - sphinx-autoapi
```

Then we add a GitHub Actions workflow `.github/workflow/sphinx.yml` to build the documentation:

```yaml
name: Build documentation

on:
  push:
    branches: [ main ]
  pull_request:
    branches: [ main ]

permissions:
  contents: write

jobs:
  docs:
    runs-on: ubuntu-latest

    steps:
    - name: Checkout
      uses: actions/checkout@v4

    - uses: mamba-org/setup-micromamba@v1
      with:
        micromamba-version: '1.5.8-0' # any version from https://github.com/mamba-org/micromamba-releases
        environment-file: environment.yml
        init-shell: bash
        cache-environment: true
        post-cleanup: 'all'
        generate-run-shell: false

    - name: Sphinx build
      run: |
        sphinx-build doc _build
      shell: bash -el {0}

    - name: Deploy to GitHub Pages
      uses: peaceiris/actions-gh-pages@v4
      if: ${{ github.event_name == 'push' && github.ref == 'refs/heads/main' }}
      with:
        publish_branch: gh-pages
        github_token: ${{ secrets.GITHUB_TOKEN }}
        publish_dir: _build/
        force_orphan: true
```

Now:

- Add these two changes to the GitHub repository.
- Go to “Settings” -> “Pages” -> “Branch” -> `gh-pages` -> “Save”.
- Look at “Actions” tab and observe the workflow running and hopefully deploying the website.
- Finally visit the generated site. You can find it by clicking the About wheel icon on top right of your repository. There, select “Use your GitHub Pages website”.
- **This is how we build almost all of our lesson websites**, including this one!
- Another popular place to deploy Sphinx documentation is ReadTheDocs.

Optional: How to auto-generate API documentation in Python

Add three tiny modifications (highlighted) to `doc/conf.py` to auto-generate API documentation (this requires the `sphinx-autoapi` package):

```python
project = "planets"
copyright = "2024, Authors"
author = "Authors"
release = "0.1"

exclude_patterns = ["_build", "Thumbs.db", ".DS_Store"]

extensions = [
    "myst_parser",  # in order to use markdown
    "autoapi.extension",  # in order to use markdown
]

# search this directory for Python files
autoapi_dirs = [".."]

# ignore this file when generating API documentation
autoapi_ignore = ["*/conf.py"]

myst_enable_extensions = [
    "colon_fence",  # ::: can be used instead of ``` for better rendering
]

html_theme = "sphinx_rtd_theme"
```

Then rebuild the documentation (or push the changes and let GitHub rebuild it) and you should see a new section “API Reference”.

Confused about reStructuredText vs. Markdown vs. MyST?

- At the beginning there was reStructuredText and Sphinx was built for reStructuredText.
- Independently, Markdown was invented and evolved into a couple of flavors.
- Markdown became more and more popular but was limited compared to reStructuredText.
- Later, MyST was invented to be able to write something that looks like Markdown but in addition can do everything that reStructuredText can do with extra directives.

Where to read more

- CodeRefinery documentation lesson
- Sphinx documentation
- Sphinx + ReadTheDocs guide
- For more Markdown functionality, see the Markdown guide.
- For Sphinx additions, see Sphinx Markup Constructs.
- An opinionated guide on documentation in Python

Collaborative version control and code review

Concepts around collaboration

Objectives

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

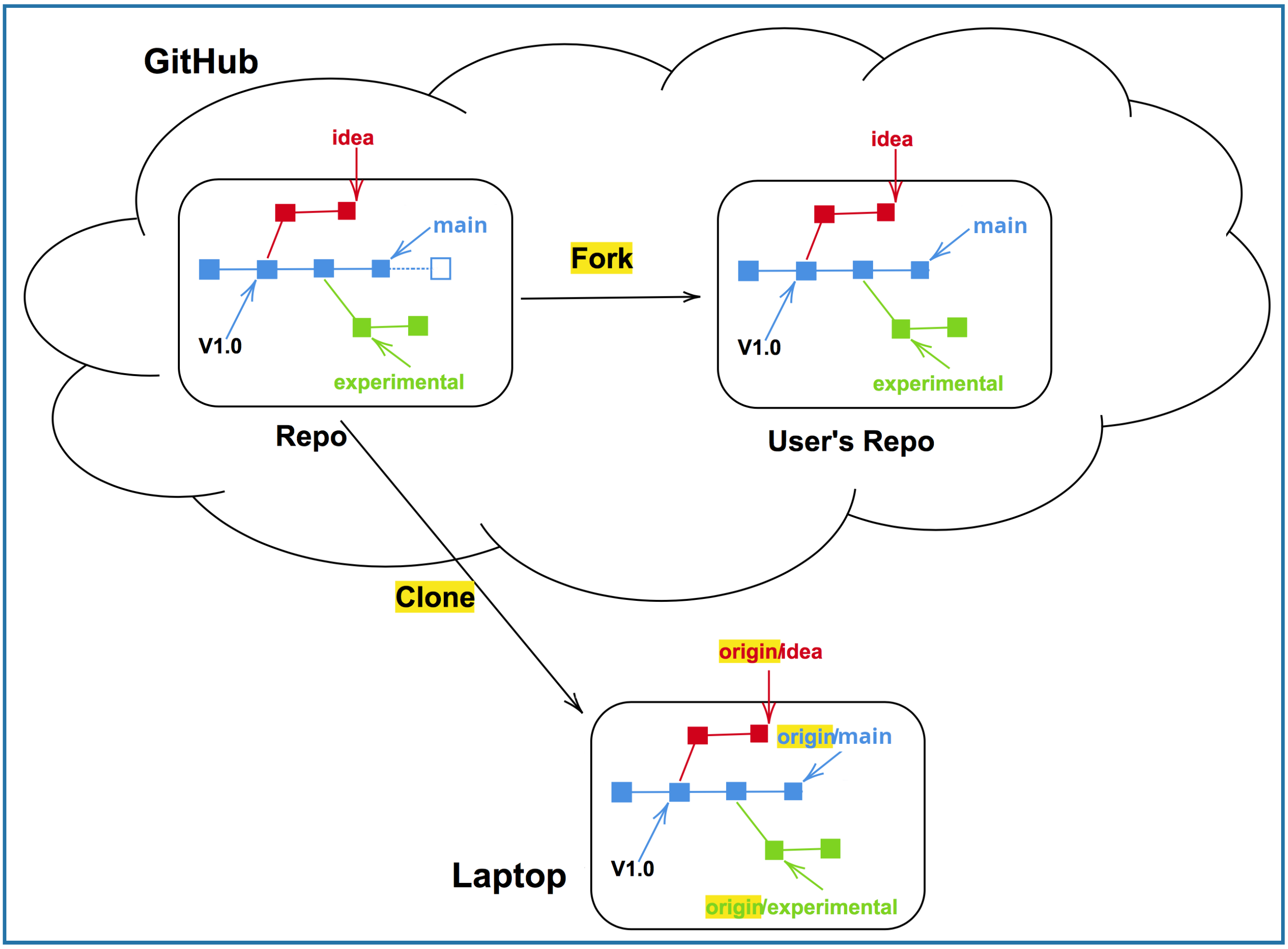
Commits, branches, repositories, forks, clones

- **repository**: The project, contains all data and history (commits, branches, tags).
- **commit**: Snapshot of the project, gets a unique identifier (e.g. `c7f0e8bfc718be04525847fc7ac237f470add76e`).
- **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`).
- **cloning**: Copying the whole repository to your laptop - the first time. It is not necessary to download each file one by one.
- **forking**: Taking a copy of a repository (which is typically not yours) - your copy (fork) stays on GitHub/GitLab and you can make changes to your copy.

Cloning a repository

In order to make a complete copy a whole repository, the `git clone` command can be used. When cloning, all the files, of all or selected branches, of a repository are copied in one operation. Cloning of a repository is of relevance in a few different situations:

- Working on your own, cloning is the operation that you can use to create multiple instances of a repository on, for instance, a personal computer, a server, and a supercomputer.
- The parent 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.



Forking and cloning

Forking a repository

When a fork is made on GitHub/GitLab a complete copy, of all or selected branches, of the repository is made. The copy will reside under a different account on GitHub/GitLab. Forking of a repository is of high relevance when working with a git repository to which you do not have write access.

- In the fork repository commits can be made to the base branch (`main` or `master`), and to other branches.
- 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.

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 GitHub 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 the former is a general operation for generating copies of a repository to different computers, whereas forking is a particular operation implemented on GitHub/GitLab.

Collaborating within the same repository

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.

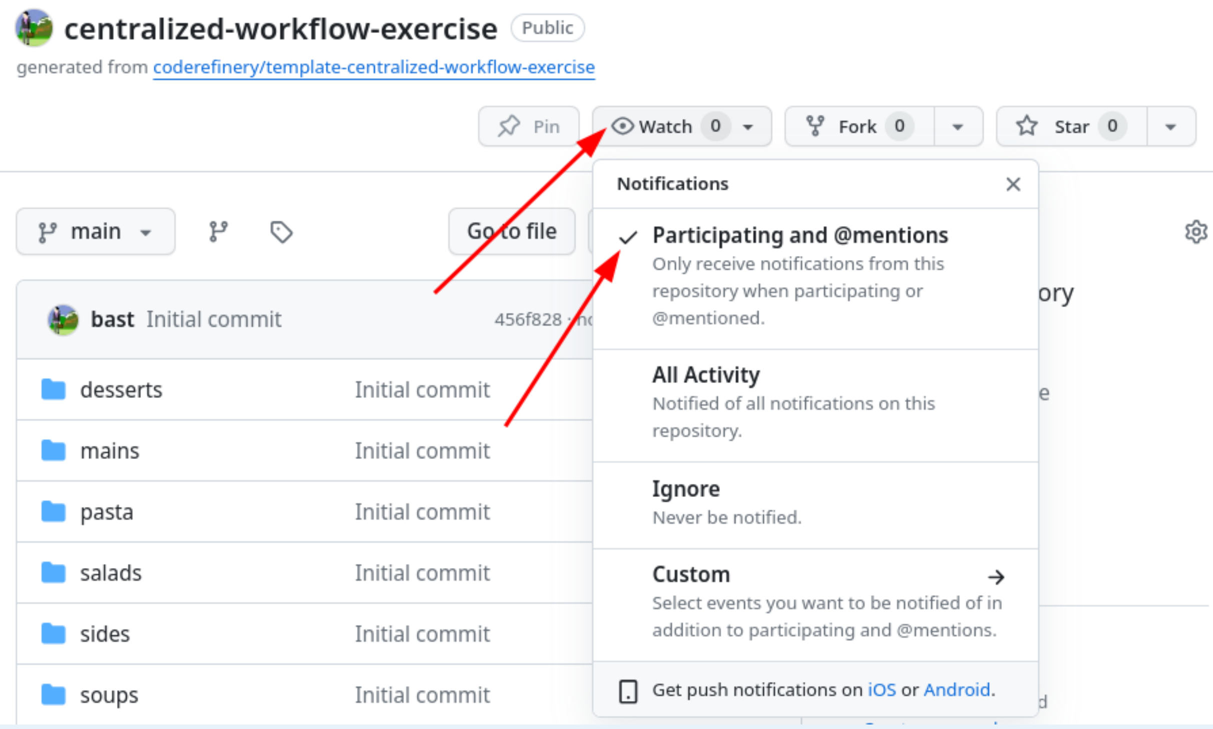
This exercise will form a good basis for collaboration that is suitable for most research groups.

> ❗ Note
>
> When you read or hear **pull request**, please think of a **change proposal**.

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 the exercise repository to which we will contribute.
>   - Instructor collects GitHub usernames from learners and adds them as collaborators to the exercise repository (Settings -> Collaborators and teams -> Manage access -> Add people).
> - **Don't forget to accept the invitation**
>   - Check https://github.com/settings/organizations/
>   - Alternatively check the inbox for the email account you registered with GitHub. 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 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.



Notifications

✓ **Participating and @mentions**
Only receive notifications from this repository when participating or @mentioned.

All Activity
Notified of all notifications on this repository.

Ignore
Never be notified.

Custom
Select events you want to be notified of in addition to participating and @mentions.

Get push notifications on iOS or Android.

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 (25 min)

Technical requirements (from installation instructions):

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

What is familiar from the previous workshop day (not repeated here):

- Cloning a repository.
- Creating a branch.
- Committing a change on the new branch.
- Submit a pull request towards the main branch.

What will be new in this exercise:

- 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. Start in the exercise repository and 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 GitHub web interface. For example, you could give the name of the recipe you want to add (so that others don't add the same one). It is the "Issues" tab.

(2) Create a new branch.

If on GitHub, you can make the branch in the web interface.

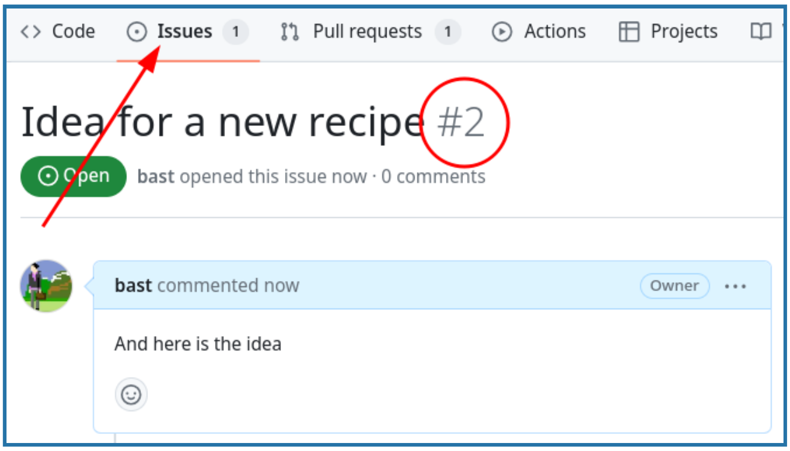
(3) Make a change adding the recipe

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`):



Idea for a new recipe #2

bast opened this issue now · 0 comments

bast commented now

And here is the idea

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).

Here are all the keywords that GitHub recognizes: https://help.github.com/en/articles/closing-issues-using-keywords

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.

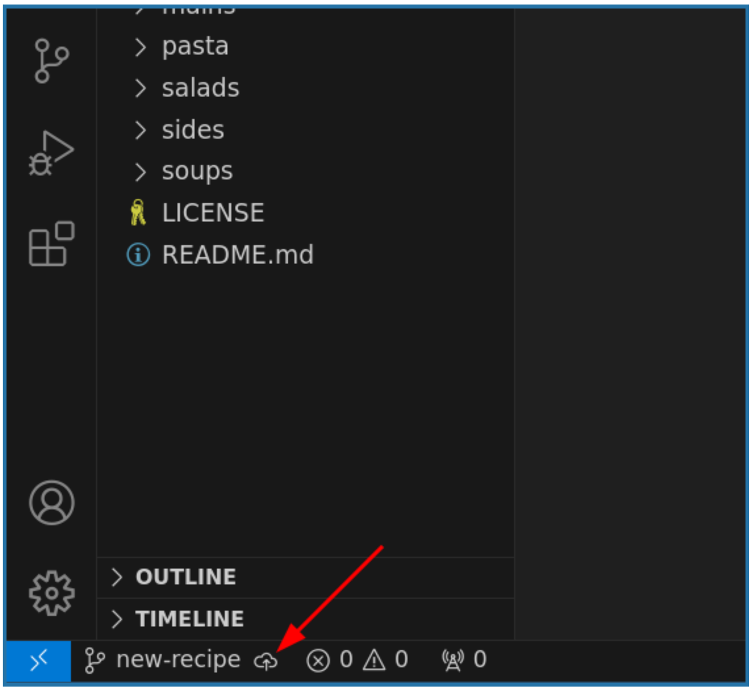
(4) Push to GitHub as a new branch

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

This is only necessary if you created the changes locally. If you created the changes directly on GitHub, 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:



(5) Open a pull request towards the main branch

This is done through the GitHub web interface.

(6) Reviewing pull requests

You review through the GitHub web interface.

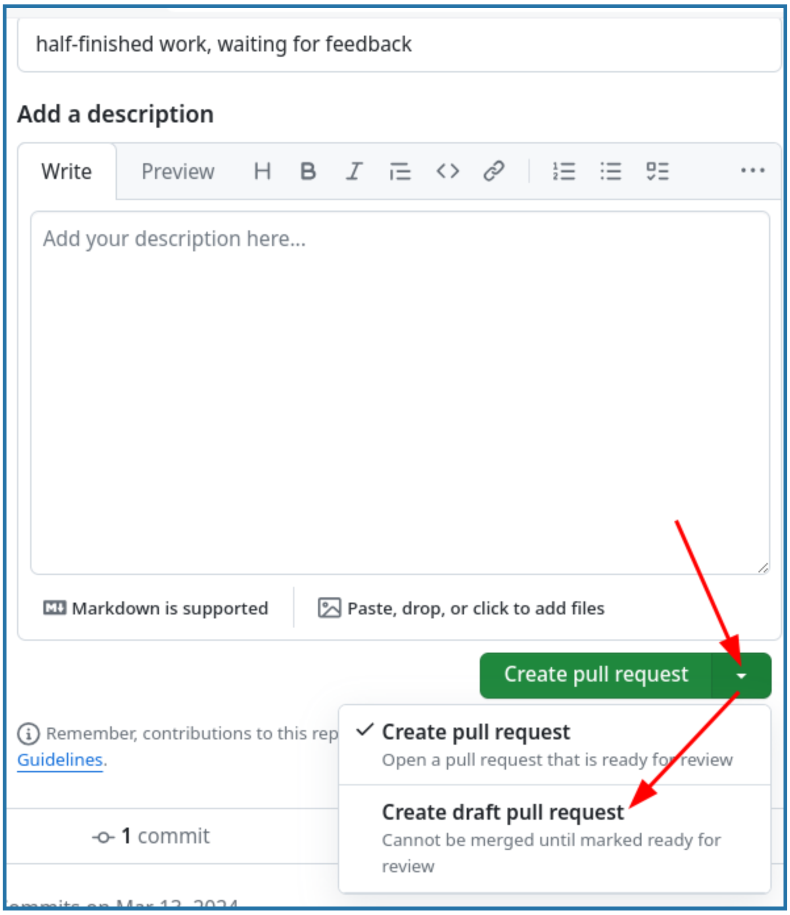
Checklist for reviewing a pull request:

- 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 pull requests

Try to create a draft pull request:



half-finished work, waiting for feedback

Add a description

Write
Preview
Add your description here...
Markdown is supported
Paste, drop, or click to add files
Create pull request

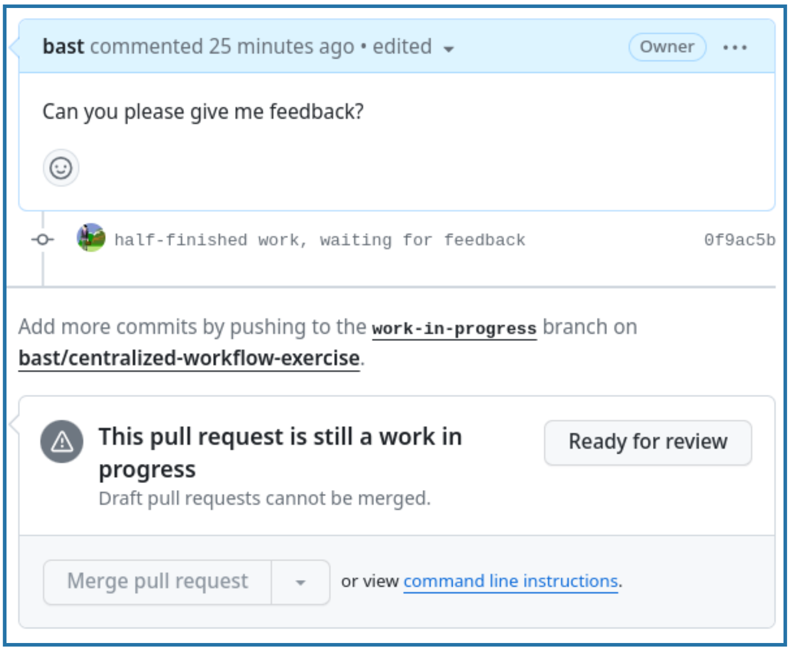
Remember, contributions to this rep
Guidelines.

Open a pull request that is ready for review

Create draft pull request

Cannot be merged until marked ready for review

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



bast commented 25 minutes ago • edited

Can you please give me feedback?

```
half-finished work, waiting for feedback                0f9ac5b
```

Add more commits by pushing to the `work-in-progress` branch on bast/centralized-workflow-exercise.

This pull request is still a work in progress

Draft pull requests cannot be merged.

or view command line instructions.

Draft 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.

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

A protected branch on GitHub or GitLab is a branch that 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 request.

To protect a branch in your own repository, go to “Settings” -> “Branches”.

Summary

- We used all the same pieces that we’ve learned previously.
- But we successfully contributed to a **collaborative project**!
- The pull request allowed us to contribute without changing directly: this is very good when it’s not mainly our project.

Practicing code review

In this episode 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, and how to modify a pull request.

This will enable research groups to work more collaboratively and to not only improve the code quality but also to **learn from each other**.

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 GitHub

What is familiar from previous lessons:

- Creating a branch.
- Committing a change on the new branch.
- Opening and merging pull requests.

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.

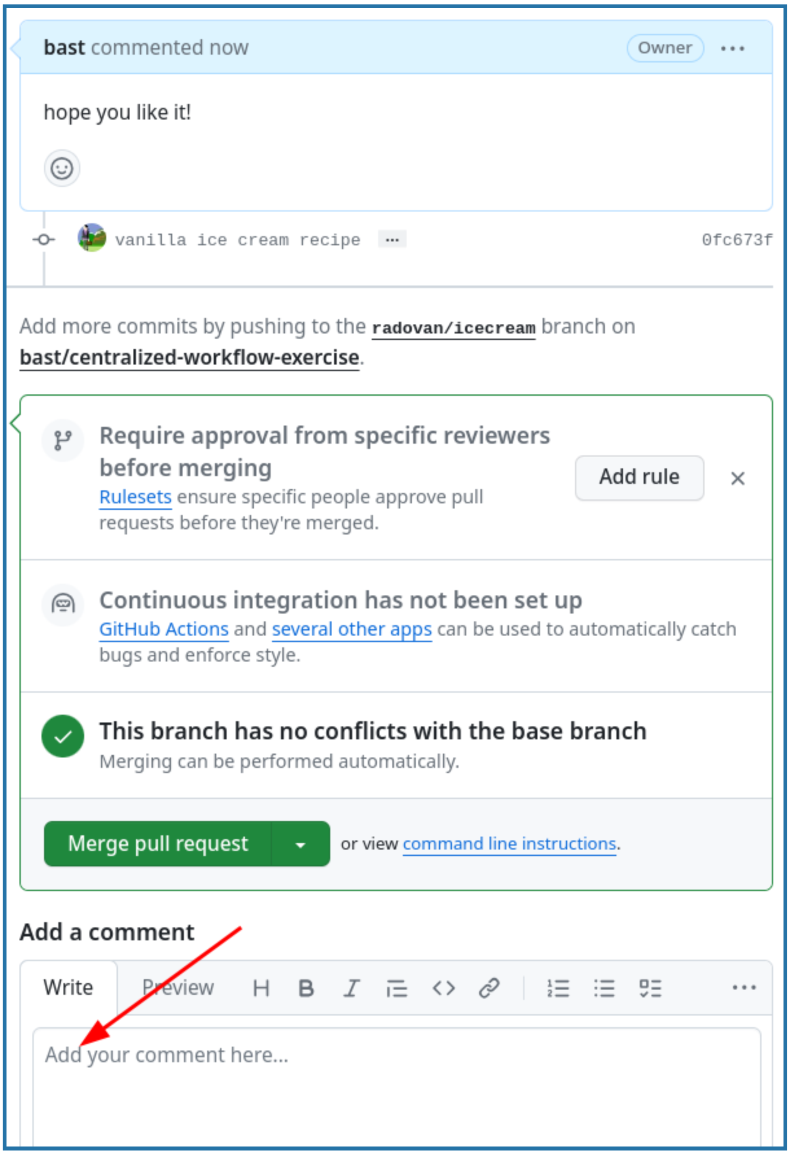
Help and discussion

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.

How to ask for changes in a pull request

Technically, there are at least two common ways to ask for changes in a pull request.

Either in the comment field of the pull request:



hope you like it!

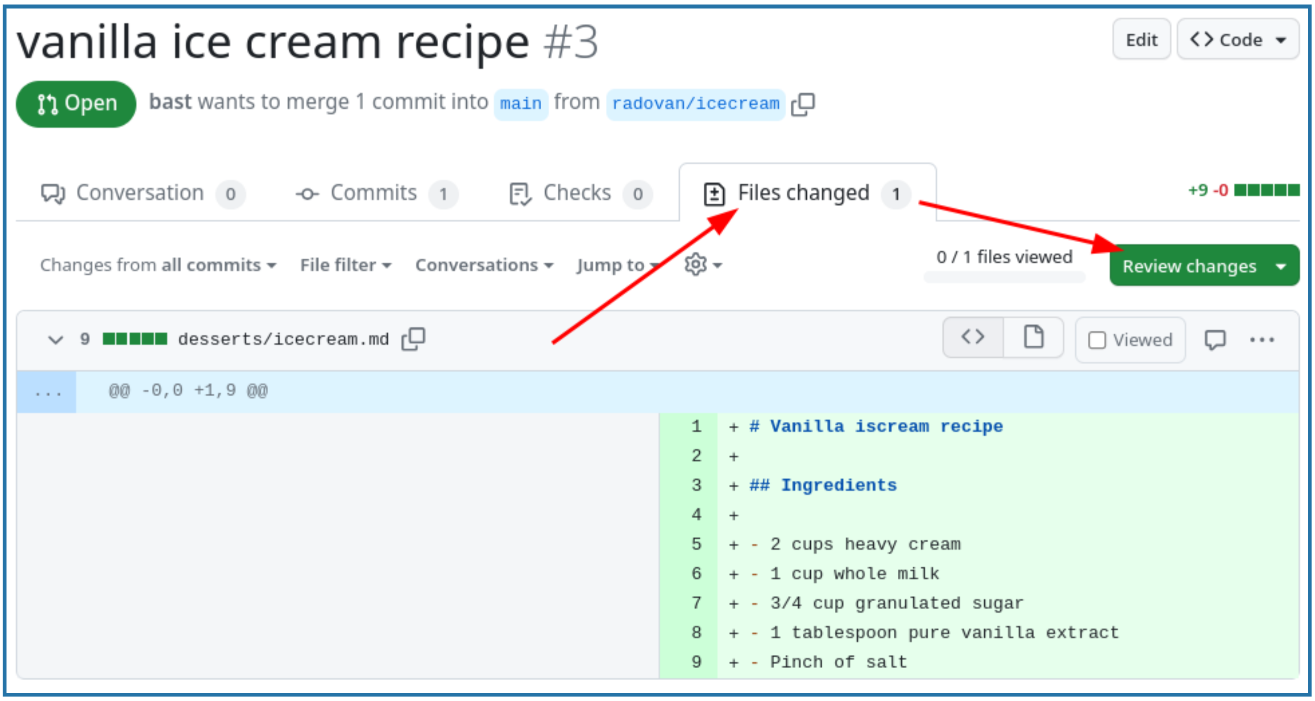
Add more commits by pushing to the **radovan/icecream** branch on **bast/centralized-workflow-exercise**.

Require approval from specific reviewers before merging
Rulesets ensure specific people approve pull requests before they're merged.

Continuous integration has not been set up
GitHub Actions and several other apps can be used to automatically catch bugs and enforce style.

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

Or by using the “Review changes”:


vanilla ice cream recipe #3

bast wants to merge 1 commit into `main` from `radovan/icecream`

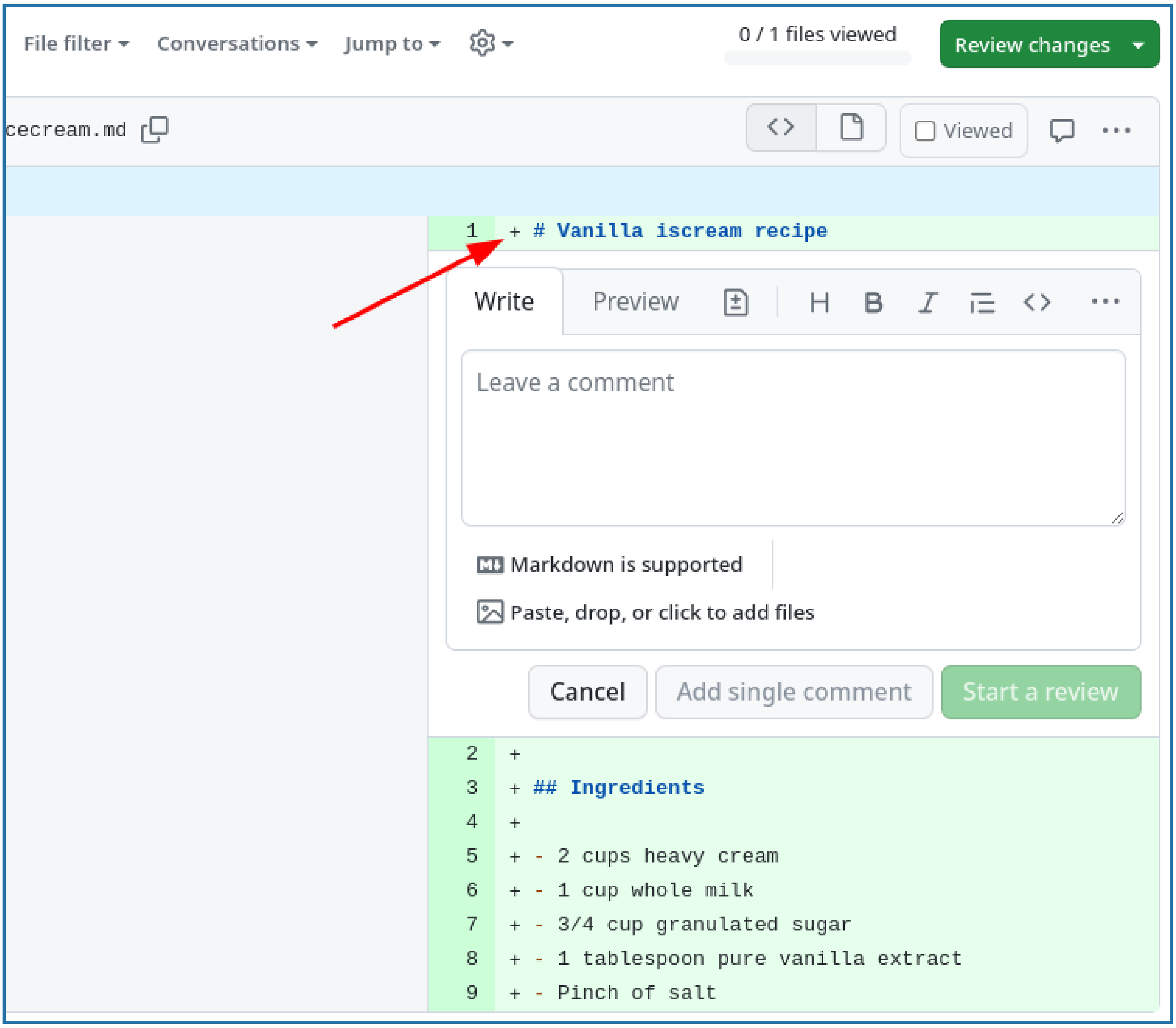
```
@@ -0,0 +1,9 @@
+ # Vanilla iscream recipe
+
+ ## Ingredients
+
+ - 2 cups heavy cream
+ - 1 cup whole milk
+ - 3/4 cup granulated sugar
+ - 1 tablespoon pure vanilla extract
+ - Pinch of salt
```

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

How to suggest a change in a pull request as a reviewer

If you see a very small problem that is easy to fix, you can suggest a change as a reviewer.

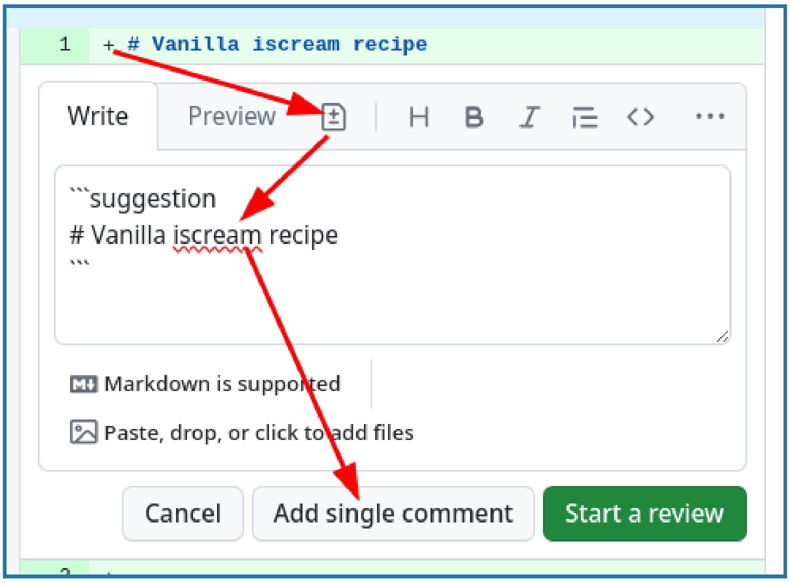
Instead of asking the submitter to tiny problem, you can suggest a change by clicking on the plus sign next to the line number in the “Files changed” tab:



```
+ # Vanilla iscream recipe
+
+ ## Ingredients
+
+ - 2 cups heavy cream
+ - 1 cup whole milk
+ - 3/4 cup granulated sugar
+ - 1 tablespoon pure vanilla extract
+ - Pinch of salt
```

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

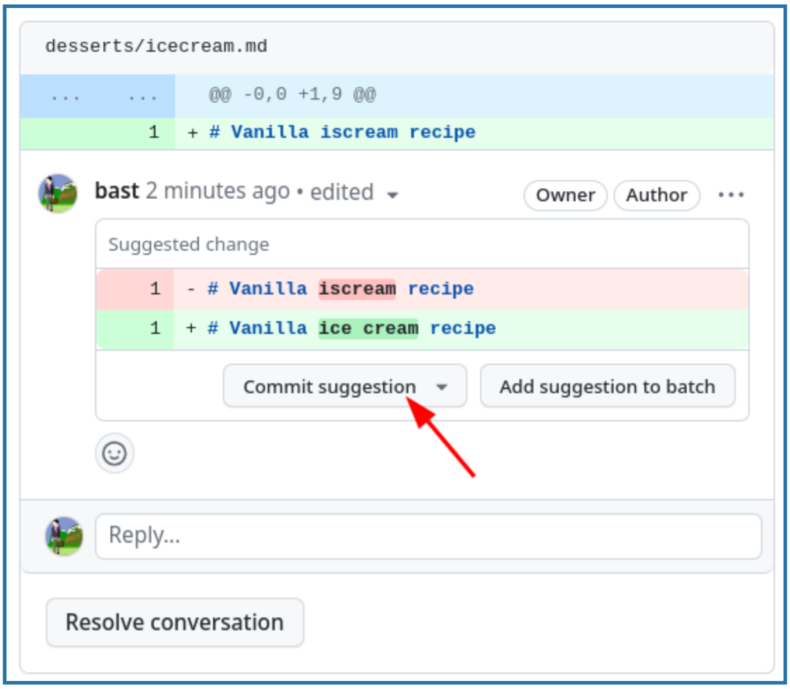
But now the interesting part is to click on the “Add a suggestion” symbol (the one that looks like plus and minus). Now you can fix the tiny problem (in this case a typo) and then click on the “Add single comment” button:



```
1  + # Vanilla iscream recipe
```

~~~
```suggestion
# Vanilla iscream recipe
```
~~~

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



```
desserts/icecream.md
...   ...   @@ -0,0 +1,9 @@
         1  + # Vanilla iscream recipe
```

bast 2 minutes ago • edited

Suggested change

```
1  - # Vanilla iscream recipe
1  + # Vanilla ice cream recipe
```

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 is 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 (or GitLab) discussions and reviewing are 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

- The exercise repository is now different: https://github.com/workshop-material/recipe-book-forking-exercise (note the **-forking-exercise**).
- First **fork** the exercise repository to your GitHub account.
- Then **clone your fork** to your computer (if you wish to work locally).
- Double-check that you have forked the correct repository.

✍ 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 previous lessons:

- Forking a repository.
- Creating a branch.
- Committing a change on the new branch.
- Opening and merging pull requests.

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. The instructor 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

Help! I don't have permissions to push my local changes

Maybe you see an error like this one:

```
Please make sure you have the correct access rights
and the repository exists.
```

Or like this one:

```
failed to push some refs to workshop-material/recipe-book-forking-exercise.git
```

In this case you probably try to push the changes not to your fork but to the original repository and in this exercise you do not have write access to the original repository.

The simpler solution is to clone again but this time your fork.

✓ Recovery

But if you want to keep your local changes, you can change the remote URL to point to your fork. Check where your remote points to with `git remote --verbose`.

It should look like this (replace `USER` with your GitHub username):

```
$ git remote --verbose

origin  git@github.com:USER/recipe-book-forking-exercise.git (fetch)
origin  git@github.com:USER/recipe-book-forking-exercise.git (push)
```

It should **not** look like this:

```
$ git remote --verbose

origin  git@github.com:workshop-material/recipe-book-forking-exercise.git (fetch)
origin  git@github.com:workshop-material/recipe-book-forking-exercise.git (push)
```

In this case you can adjust "origin" to point to your fork with:

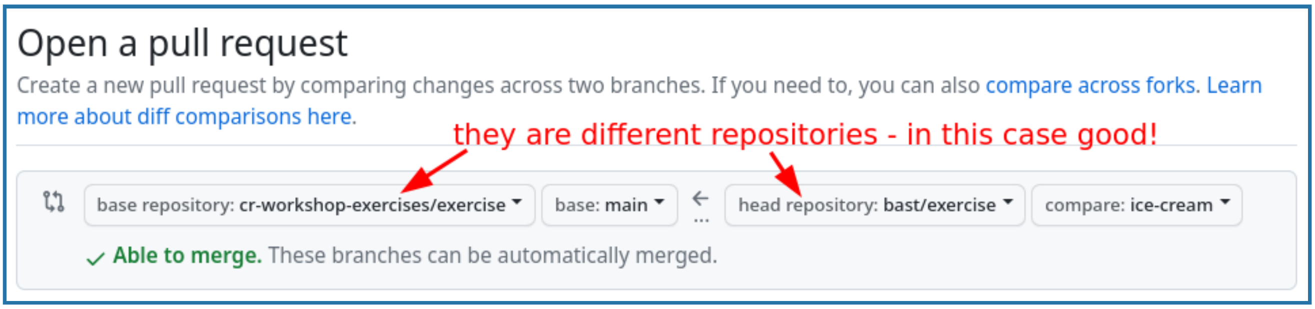
```
$ git remote set-url origin git@github.com:USER/recipe-book-forking-exercise.git
```

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:



Open a pull request

Create a new pull request by comparing changes across two branches. If you need to, you can also compare across forks. Learn more about diff comparisons here.

they are different repositories - in this case good!

base repository: cr-workshop-exercises/exercise
base: main
head repository: bast/exercise
compare: ice-cream

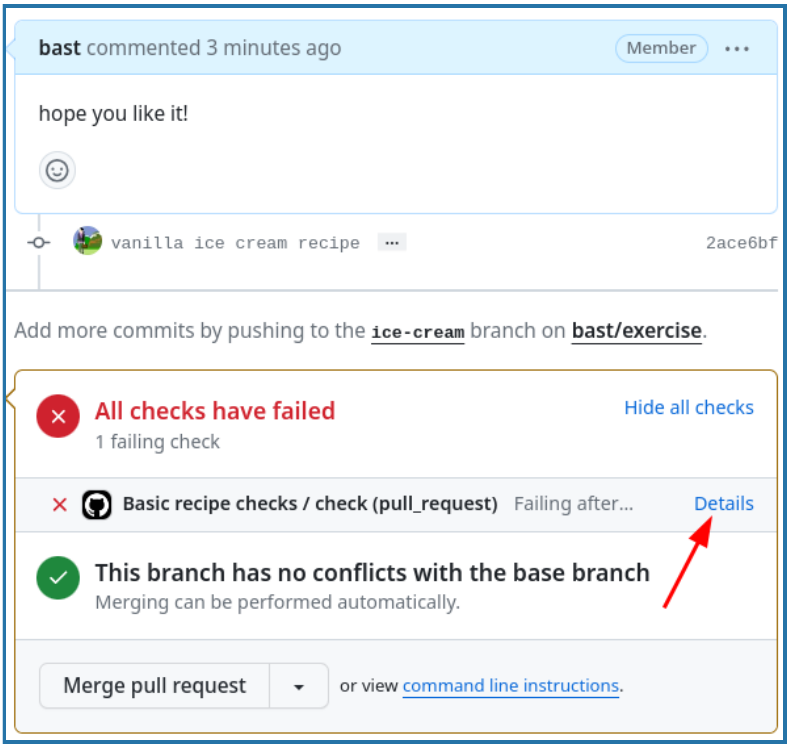
✓ Able to merge. These branches can be automatically merged.

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:



bast commented 3 minutes ago · Member

hope you like it!

vanilla ice cream recipe · 2ace6bf

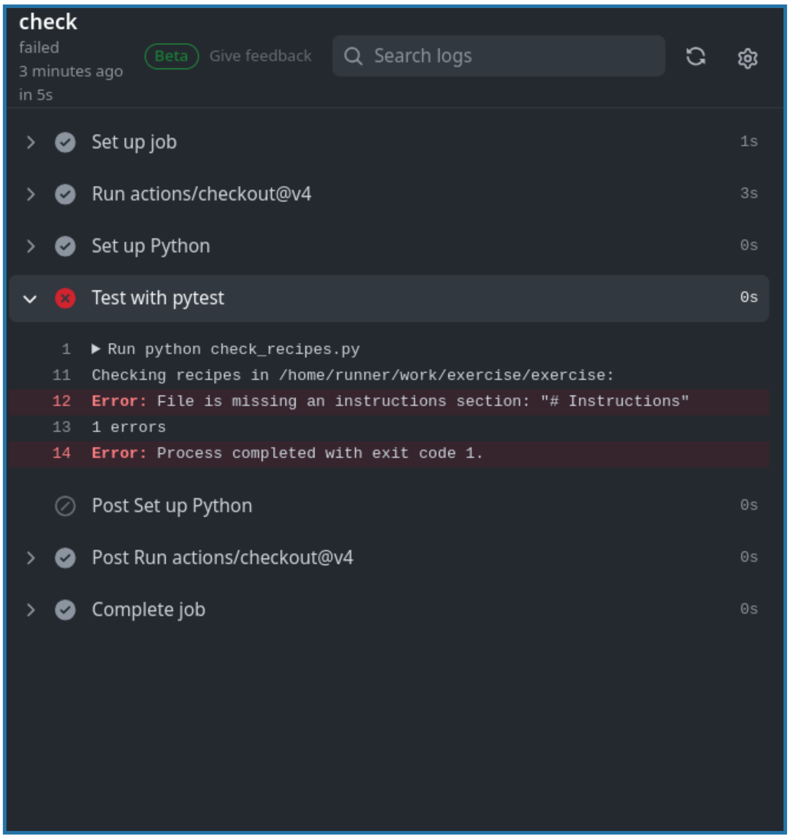
Add more commits by pushing to the `ice-cream` branch on **bast/exercise**.

All checks have failed
1 failing check

✕ **Basic recipe checks / check (pull_request)** Failing after…

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

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



check
failed
3 minutes ago
in 5s

```
Set up job                      1s
Run actions/checkout@v4         3s
Set up Python                   0s
Test with pytest                0s
  ▶ Run python check_recipes.py
  Checking recipes in /home/runner/work/exercise/exercise:
  Error: File is missing an instructions section: "# Instructions"
  1 errors
  Error: Process completed with exit code 1.
Post Set up Python              0s
Post Run actions/checkout@v4    0s
Complete job                    0s
```

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?

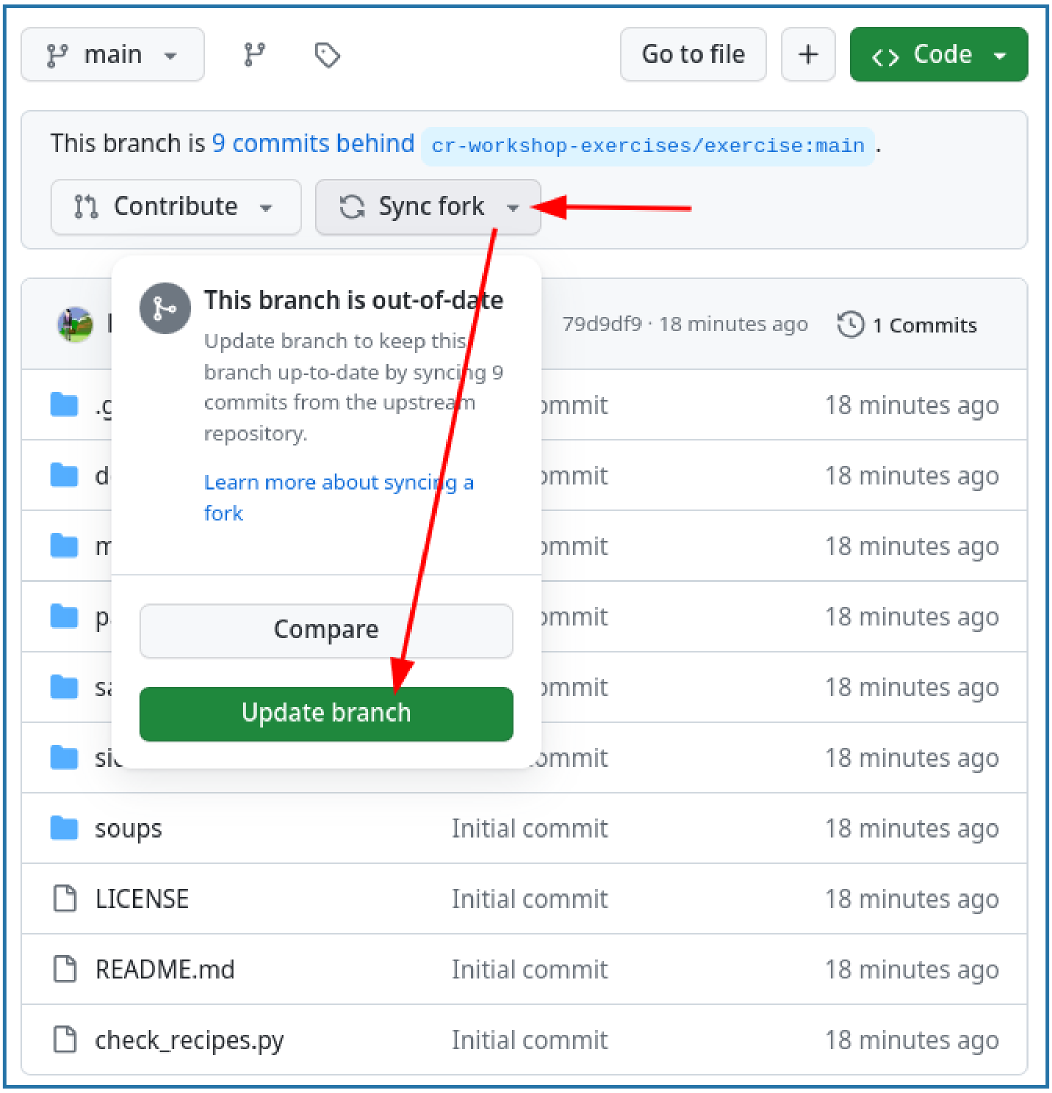
- We added a GitHub Actions workflow to automatically run on each push or pull request towards the `main` branch.

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”:



This branch is 9 commits behind `cr-workshop-exercises/exercise:main`.

This branch is out-of-date

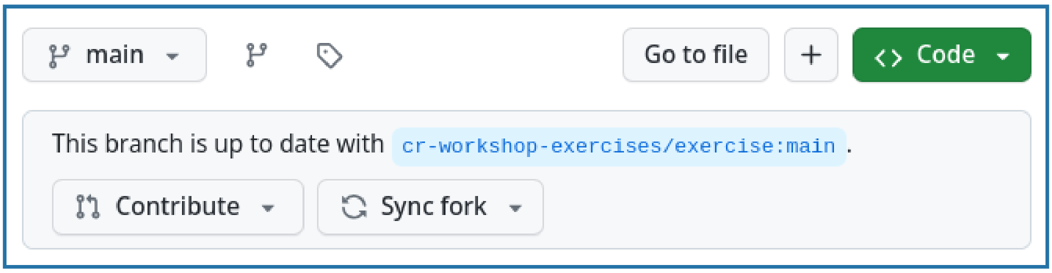
Update branch to keep this branch up-to-date by syncing 9 commits from the upstream repository.

Learn more about syncing a fork

79d9df9 · 18 minutes ago 1 Commits

soups	Initial commit	18 minutes ago
LICENSE	Initial commit	18 minutes ago
README.md	Initial commit	18 minutes ago
check_recipes.py	Initial commit	18 minutes ago

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



This branch is up to date with `cr-workshop-exercises/exercise:main`.

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 write access**.
- This is the way that much modern open-source software works.
- You can now contribute to any project you can view.

Conflict resolution, rebasing, and organizational strategies

(40 min demo and discussion)

Automated testing

Objectives

- Know **where to start** in your own project.
- Have an example for how to make the **testing part of code review**.

Instructor note

- (15 min) Motivation
- (15 min) End-to-end tests
- (15 min) Pytest
- (15 min) Adding the unit test to GitHub Actions
- (10 min) What else is possible
- (20 min) Exercise

Motivation

Testing is a way to check that the code does what it is expected to.

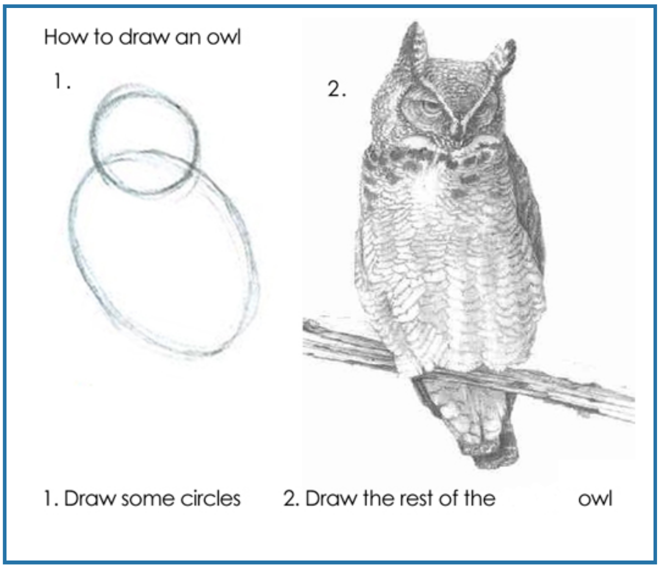
- **Less scary to change code**: tests will tell you whether something broke.
- **Easier for new people** to join.
- Easier for somebody to **revive an old code**.
- **End-to-end test**: run the whole code and compare result to a reference.
- **Unit tests**: test one unit (function or module). Can guide towards better structured code: complicated code is more difficult to test.

How testing is often taught

```python
def add(a, b):
    return a + b


def test_add():
    assert add(1, 2) == 3
```

How this feels:



[Citation needed]

Instead, we will look at and **discuss a real example** where we test components from our example project.

Where to start

Short answer: **Start with an end-to-end test.**

✔ Longer answer

- A simple script or notebook probably does not need an automated test.

If you have nothing yet:

- Start with an end-to-end test.
- Describe in words how *you* check whether the code still works.
- Translate the words into a script (any language).
- Run the script automatically on every code change.

If you want to start with unit-testing:

- You want to rewrite a function? Start adding a unit test right there first.

End-to-end tests

- This is our end-to-end test: https://github.com/workshop-material/planets/blob/main/test.sh
- Note how we can run it on GitHub automatically.
- Also browse https://github.com/workshop-material/planets/actions.
- If we have time, we can try to create a pull request which would break the code and see how the test fails.

💬 Discussion

Is the end-to-end test perfect? No. But it’s a good starting point. Discuss its limitations.

Pytest

First we need to add a test function, for instance for this function:

```python
def force_between_planets(position1, mass1, position2, mass2):
    G = 1.0  # gravitational constant

    r = position2 - position1
    distance = (r[0] ** 2 + r[1] ** 2 + r[2] ** 2) ** 0.5
    force_magnitude = G * mass1 * mass2 / distance**2
    force = (r / distance) * force_magnitude

    return force


def test_force_between_planets():
    position1 = np.array([0.0, 0.0, 0.0])
    mass1 = 1.0
    position2 = np.array([1.0, 0.0, 0.0])
    mass2 = 2.0

    force = force_between_planets(position1, mass1, position2, mass2)

    assert np.allclose(force, [2.0, 0.0, 0.0])
```

Let us run the test with:

```
$ pytest simulate.py
```

Explanation: `pytest` will look for functions starting with `test_` in files and directories given as arguments. It will run them and report the results.

Now let us try this:

- Commit the test.
- Break the function on purpose and run the test.
- Does the test fail as expected?

Adding the unit test to GitHub Actions

Our next goal is that we want GitHub to run the unit test automatically on every change.

First we need to extend our environment.yml:

```yaml
name: planets
channels:
  - conda-forge
dependencies:
  - python=3.12
  - numpy
  - click
  - matplotlib
  - pytest
```

We also need to extend `.github/workflows/test.yml` (highlighted line):

```yaml
name: Test

on:
  push:
    branches: [ main ]
  pull_request:
    branches: [ main ]

jobs:
  build:
    runs-on: ubuntu-latest

    steps:
    - name: Checkout
      uses: actions/checkout@v4

    - uses: mamba-org/setup-micromamba@v1
      with:
        micromamba-version: '1.5.8-0' # any version from https://github.com/mamba-org/micromamba-releases
        environment-file: environment.yml
        init-shell: bash
        cache-environment: true
        post-cleanup: 'all'
        generate-run-shell: false

    - name: Run tests
      run: |
        ./test.sh
        pytest simulate.py
      shell: bash -el {0}
```

If we have time, we can try to create a pull request which would break the code and see how the test fails.

What else is possible

- The testing above used **example-based** testing.
- **Test coverage**: how much of the code is traversed by tests?
  - Python: pytest-cov
  - Result can be deployed to services like Codecov or Coveralls.

- **Property-based** testing: generates arbitrary data matching your specification and checks that your guarantee still holds in that case.
    - Python: hypothesis
- **Snapshot-based** testing: makes it easier to generate snapshots for regression tests.
    - Python: syrupy
- **Mutation testing**: tests pass -> change a line of code (make a mutant) -> test again and check whether all mutants get “killed”.
    - Python: mutmut

Exercises

Experiment with the example project and what we learned above or try it on your own project:

- Add a unit test.
- Try to run it locally.
- Check whether it fails when you break the corresponding function.
- Try to run it on GitHub Actions.
- Create a pull request which would break the code and see whether the automatic test would catch it.
- Try to design an end-to-end test for your project.

How to make the project more reusable

❗ Objectives

There are not many codes that have no dependencies. How should we **deal with dependencies**?

How to avoid: “It works on my machine 🤷”

Use a **standard way** to list dependencies in your project:

- Python: `requirements.txt` or `environment.yml`
- R: `DESCRIPTION` or `renv.lock`
- Rust: `Cargo.lock`
- Julia: `Project.toml`
- C/C++/Fortran: `CMakeLists.txt` or `Makefile` or `spack.yaml` or the module system on clusters or containers
- Other languages: …

Tools and what problems they try to solve

Conda, Anaconda, mamba, pip, virtualenv, Pipenv, pyenv, Poetry, requirements.txt, environment.yml, renv, …, these tools try to solve the following problems:

- **Defining a specific set of dependencies**, possibly with well defined versions
- **Installing those dependencies** mostly automatically
- **Recording the versions** for all dependencies
- **Isolate environments**
  - On your computer for projects so they can use different software
  - Isolate environments on computers with many users (and allow self-installations)
- Using **different Python/R versions** per project
- Provide tools and services to **share packages**

Essential XKCD comics:

- xkcd - dependency
- xkcd - superfund

Best practices

Install dependencies into **isolated environments**:

- For each project, create a new environment.
- Don't install dependencies globally for all projects.
- Install them **from a file** which documents them at the same time.

Keypoints

If somebody asks you what dependencies you have in your project, you should be able to answer this question **with a file**.

In Python, the two most common ways to do this are:

- **requirements.txt** (for pip and virtual environments)
- **environment.yml** (for conda and similar)

You can export the dependencies from your current environment into these files:

```
# inside a conda environment
$ conda env export --from-history > environment.yml

# inside a virtual environment
$ pip freeze > requirements.txt
```

Discussion

- The dependencies in our example project are listed in a environment.yml file.
- Shouldn't the dependencies be pinned to specific versions?
- When is a good time to pin them?

Exercise

✍️ Exercise: Time-capsule of dependencies

Situation: 5 students (A, B, C, D, E) wrote a code that depends on a couple of libraries. They uploaded their projects to GitHub. We now travel 3 years into the future and find their GitHub repositories and try to re-run their code before adapting it.

- Which version do you expect to be easiest to re-run? Why?
- What problems do you anticipate in each solution?

Conda Python virtualenv

A: You find a couple of library imports across the code but that's it.

B: The README file lists which libraries were used but does not mention any versions.

C: You find a `environment.yml` file with:

```
name: student-project
channels:
  - conda-forge
dependencies:
  - scipy
  - numpy
  - sympy
  - click
  - python
  - pip
  - pip:
    - git+https://github.com/someuser/someproject.git@master
    - git+https://github.com/anotheruser/anotherproject.git@master
```

D: You find a `environment.yml` file with:

```
name: student-project
channels:
  - conda-forge
dependencies:
  - scipy=1.3.1
  - numpy=1.16.4
  - sympy=1.4
  - click=7.0
  - python=3.8
  - pip
  - pip:
    - git+https://github.com/someuser/someproject.git@d7b2c7e
    - git+https://github.com/anotheruser/anotherproject.git@sometag
```

E: You find a `environment.yml` file with:

```
name: student-project
channels:
  - conda-forge
dependencies:
  - scipy=1.3.1
  - numpy=1.16.4
  - sympy=1.4
  - click=7.0
  - python=3.8
  - someproject=1.2.3
  - anotherproject=2.3.4
```

✔ Solution

A: It will be tedious to collect the dependencies one by one. And after the tedious process you will still not know which versions they have used.

B: If there is no standard file to look for and look at and it might become very difficult for to create the software environment required to run the software. But at least we know the list of libraries. But we don't know the versions.

C: Having a standard file listing dependencies is definitely better than nothing. However, if the versions are not specified, you or someone else might run into problems with dependencies, deprecated features, changes in package APIs, etc.

D and **E**: In both these cases exact versions of all dependencies are specified and one can recreate the software environment required for the project. One problem with the dependencies that come from GitHub is that they might have disappeared (what if their authors deleted these repositories?).

E is slightly preferable because version numbers are easier to understand than Git commit hashes or Git tags.

Containers

- A container is like an **operating system inside a file**.
- "Building a container": Container definition file (recipe) -> Container image
- Let us explore and discuss the container definition file in our example project.
- This can be used with Apptainer/ SingularityCE.

Containers offer the following advantages:

- **Reproducibility**: The same software environment can be recreated on different computers. They force you to know and **document all your dependencies**.
- **Portability**: The same software environment can be run on different computers.
- **Isolation**: The software environment is isolated from the host system.
- “**Time travel**”:
  - You can run old/unmaintained software on new systems.
  - Code that needs new dependencies which are not available on old systems can still be run on old systems.

Demonstration: Building a container

✍️ Demo: Build a container and run it on a cluster

Here we will try to build a container from the definition file of our example project.

Requirements:

1. Linux (it is possible to build them on a macOS or Windows computer but it is more complicated).
2. An installation of Apptainer (e.g. following the quick installation). Alternatively, SingularityCE should also work.

Now you can build the container image from the container definition file. Depending on the configuration you might need to run the command with `sudo` or with `--fakeroot`.

Hopefully one of these four will work:

```
$ sudo apptainer build container.sif container.def
$ apptainer build --fakeroot container.sif container.def

$ sudo singularity build container.sif container.def
$ singularity build --fakeroot container.sif container.def
```

Once you have the `container.sif`, copy it to a cluster and try to run it there.

Here are two job script examples:

Dardel (Sweden) Saga (Norway)

```
#!/usr/bin/env bash

# the SBATCH directives and the module load below are only relevant for the
# Dardel cluster and the PDC Summer School; adapt them for your cluster

#SBATCH --account=edu24.summer
#SBATCH --job-name='container'
#SBATCH --time=0-00:05:00

#SBATCH --partition=shared

#SBATCH --nodes=1
#SBATCH --tasks-per-node=1
#SBATCH --cpus-per-task=16


module load PDC singularity


# catch common shell script errors
set -euf -o pipefail


echo
echo "what is the operating system on the host?"
cat /etc/os-release


echo
echo "what is the operating system in the container?"
singularity exec container.sif cat /etc/os-release


# 1000 planets, 20 steps
time ./container.sif 1000 20 ${SLURM_CPUS_PER_TASK} results
```

Where to explore more

- Reproducible research
- The Turing Way: Guide for Reproducible Research
- Ten simple rules for writing Dockerfiles for reproducible data science
- Computing environment reproducibility
- Carpentries incubator lesson on Docker
- Carpentries incubator lesson on Singularity/Apptainer

Concepts in refactoring and modular code design

Starting questions for the collaborative document

1. What does “modular code development” mean for you?
2. What best practices can you recommend to arrive at well structured, modular code in your favourite programming language?

3. What do you know now about programming that you wish somebody told you earlier?
4. Do you design a new code project on paper before coding? Discuss pros and cons.
5. Do you build your code top-down (starting from the big picture) or bottom-up (starting from components)? Discuss pros and cons.
6. Would you prefer your code to be 2x slower if it was easier to read and understand?

Pure functions

- Pure functions have no notion of state: They take input values and return values
- **Given the same input, a pure function always returns the same value**
- Function calls can be optimized away
- Pure function == data

a) pure: no side effects

```
def fahrenheit_to_celsius(temp_f):
    temp_c = (temp_f - 32.0) * (5.0/9.0)
    return temp_c

temp_c = fahrenheit_to_celsius(temp_f=100.0)
print(temp_c)
```

b) stateful: side effects

```
f_to_c_offset = 32.0
f_to_c_factor = 0.555555555
temp_c = 0.0

def fahrenheit_to_celsius_bad(temp_f):
    global temp_c
    temp_c = (temp_f - f_to_c_offset) * f_to_c_factor

fahrenheit_to_celsius_bad(temp_f=100.0)
print(temp_c)
```

Pure functions are easier to:

- Test
- Understand
- Reuse
- Parallelize
- Simplify
- Optimize
- Compose

Mathematical functions are pure:

$$f(x, y) = x - x^2 + x^3 + y^2 + xy$$

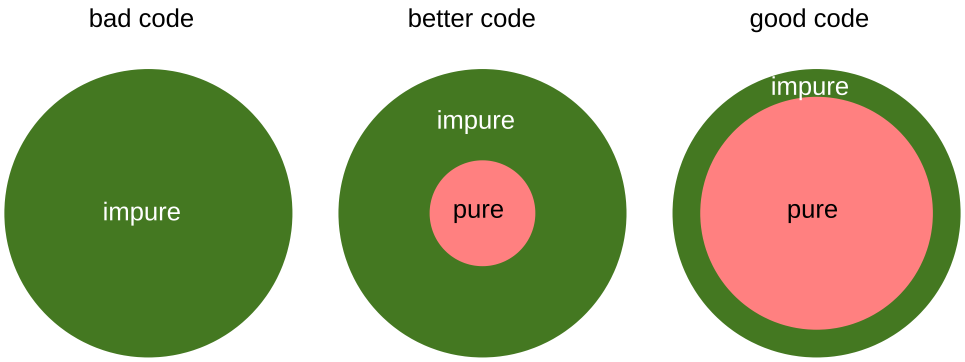
$$(f \circ g)(x) = f(g(x))$$

Unix shell commands are stateless:

```
$ cat somefile | grep somestring | sort | uniq | ...
```

But I/O and network and disk and databases are not pure!

- I/O is impure
- Keep I/O on the “outside” of your code
- Keep the “inside” of your code pure/stateless



From classes to functions

Object-oriented programming and functional programming **both have their place and value**.

Here is an example of expressing the same thing in Python in 4 different ways. Which one do you prefer?

1. As a **class**:

```python
import math


class Moon:
    def __init__(self, name, radius, contains_water=False):
        self.name = name
        self.radius = radius  # in kilometers
        self.contains_water = contains_water

    def surface_area(self) -> float:
        """Calculate the surface area of the moon assuming a spherical shape."""
        return 4.0 * math.pi * self.radius**2

    def __repr__(self):
        return f"Moon(name={self.name!r}, radius={self.radius}, contains_water={self.contains_water})"


europa = Moon(name="Europa", radius=1560.8, contains_water=True)

print(europa)
print(f"Surface area (km^2) of {europa.name}: {europa.surface_area()}")
```

2. As a **dataclass**:

```python
from dataclasses import dataclass
import math


@dataclass
class Moon:
    name: str
    radius: float  # in kilometers
    contains_water: bool = False

    def surface_area(self) -> float:
        """Calculate the surface area of the moon assuming a spherical shape."""
        return 4.0 * math.pi * self.radius**2


europa = Moon(name="Europa", radius=1560.8, contains_water=True)

print(europa)
print(f"Surface area (km^2) of {europa.name}: {europa.surface_area()}")
```

3. As a **named tuple**:

```python
import math
from collections import namedtuple


def surface_area(radius: float) -> float:
    return 4.0 * math.pi * radius**2


Moon = namedtuple("Moon", ["name", "radius", "contains_water"])


europa = Moon(name="Europa", radius=1560.8, contains_water=True)

print(europa)
print(f"Surface area (km^2) of {europa.name}: {surface_area(europa.radius)}")
```

4. As a **dict**:

```python
import math


def surface_area(radius: float) -> float:
    return 4.0 * math.pi * radius**2


europa = {"name": "Europa", "radius": 1560.8, "contains_water": True}


print(europa)
print(f"Surface area (km^2) of {europa['name']}: {surface_area(europa['radius'])}")
```

How to design your code before writing it

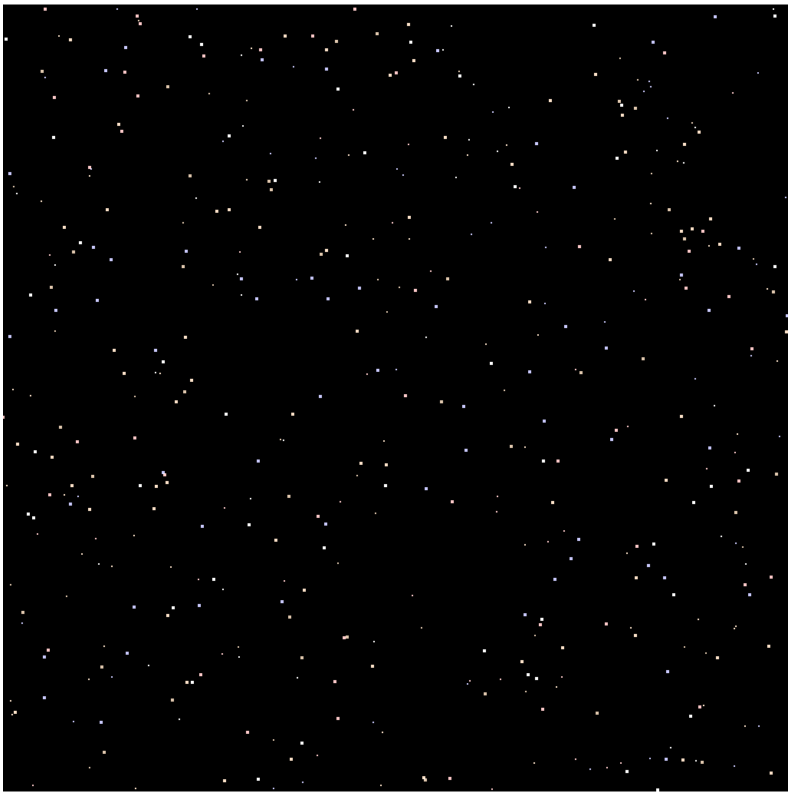
- Document-driven development can be a nice approach:
  - Write the documentation/tutorial first
  - Write the code to make the documentation true
  - Refactor the code to make it clean and maintainable
- But also it’s almost impossible to design everything correctly from the start -> make it easy to change -> keep it simple

Demo: From a script towards a workflow

In this episode we will explore code quality and good practices in Python using a hands-on approach. We will together build up a small project and improve it step by step.

We will start from a relatively simple image processing script which can read a telescope image of stars and our goal is to **count the number of stars** in the image. Later we will want to be able to process many such images.

The (fictional) telescope images look like the one below here (in this repository we can find more):



Generated image representing a telescope image of stars.

Rough plan for this demo

- (15 min) Discuss how we would solve the problem, run example code, and make it work (as part of a Jupyter notebook)?
- (15 min) Refactor the positioning code into a function and a module
- (15 min) Now we wish to process many images - discuss how we would approach this
- (15 min) Introduce CLI and discuss the benefits

- (30 min) From a script to a workflow (using Snakemake)

✔ Starting point (spoiler alert)

We can imagine that we pieced together the following code based on some examples we found online:

```python
import matplotlib.pyplot as plt
from skimage import io, filters, color
from skimage.measure import label, regionprops


image = io.imread("stars.png")
sigma = 0.5

# if there is a fourth channel (alpha channel), ignore it
rgb_image = image[:, :, :3]
gray_image = color.rgb2gray(rgb_image)

# apply a gaussian filter to reduce noise
image_smooth = filters.gaussian(gray_image, sigma)

# threshold the image to create a binary image (bright stars will be white, background black)
thresh = filters.threshold_otsu(image_smooth)
binary_image = image_smooth > thresh

# label connected regions (stars) in the binary image
labeled_image = label(binary_image)

# get properties of labeled regions
regions = regionprops(labeled_image)

# extract star positions (centroids)
star_positions = [region.centroid for region in regions]

# plot the original image
plt.figure(figsize=(8, 8))
plt.imshow(image, cmap="gray")

# overlay star positions with crosses
for star in star_positions:
    plt.plot(star[1], star[0], "rx", markersize=5, markeredgewidth=0.1)

plt.savefig("detected-stars.png", dpi=300)

print(f"number of stars detected: {len(star_positions)}")
```

Plan

Topics we wish to show and discuss:

- Naming (and other) conventions, project organization, modularity
- The value of pure functions and immutability

- Refactoring (explained through examples)
- Auto-formatting and linting with tools like black, vulture, ruff
- Moving a project under Git
- How to document dependencies
- Structuring larger software projects in a modular way
- Command-line interfaces
- Workflows with Snakemake

We will work together on the code on the big screen, and participants will be encouraged to give suggestions and ask questions. We will **end up with a Git repository** which will be shared with workshop participants.

Possible solutions

✔ Script after some work, with command-line interface (spoiler alert)

This is one possible solution (`countstars.py`):

```python
import click
import matplotlib.pyplot as plt
from skimage import io, filters, color
from skimage.measure import label, regionprops


def convert_to_gray(image):
    # if there is a fourth channel (alpha channel), ignore it
    rgb_image = image[:, :, :3]
    return color.rgb2gray(rgb_image)


def locate_positions(image):
    gray_image = convert_to_gray(image)

    # apply a gaussian filter to reduce noise
    image_smooth = filters.gaussian(gray_image, sigma=0.5)

    # threshold the image to create a binary image (bright objects will be white,
background black)
    thresh = filters.threshold_otsu(image_smooth)
    binary_image = image_smooth > thresh

    # label connected regions in the binary image
    labeled_image = label(binary_image)

    # get properties of labeled regions
    regions = regionprops(labeled_image)

    # extract positions (centroids)
    positions = [region.centroid for region in regions]

    return positions


def plot_positions(image, positions, file_name):
    # plot the original image
    plt.figure(figsize=(8, 8))
    plt.imshow(image, cmap="gray")

    # overlay positions with crosses
    for y, x in positions:
        plt.plot(y, x, "rx", markersize=5, markeredgewidth=0.1)

    plt.savefig(file_name, dpi=300)


@click.command()
@click.option(
    "--image-file", type=click.Path(exists=True), help="Path to the input image"
)
@click.option("--output-file", type=click.Path(), help="Path to the output file")
@click.option("--generate-plot", is_flag=True, default=False)
def main(image_file, output_file, generate_plot):
    image = io.imread(image_file)

    star_positions = locate_positions(image)

    if generate_plot:
        plot_positions(image, star_positions, f"detected-{image_file}")

    with open(output_file, "w") as f:
        f.write(f"number of stars detected: {len(star_positions)}\n")
```

```
if __name__ == "__main__":
    main()
```

✔ Snakemake rules which define a workflow (spoiler alert)

This is one possible solution (`snakefile`):

```
# the comma is there because glob_wildcards returns a named tuple
numbers, = glob_wildcards("input-images/stars-{number}.png")


# rule that collects the target files
rule all:
    input:
        expand("results/{number}.txt", number=numbers)


rule process_data:
    input:
        "input-images/stars-{number}.png"
    output:
        "results/{number}.txt"
    log:
        "logs/{number}.txt"
    shell:
        """
        python countstars.py --image-file {input} --output-file {output}
        """
```

We can process as many images as we like by running:

```
$ snakemake --cores 4  # adjust to the number of available cores
```

Choosing a software license

Objectives

- Knowing about what derivative work is and whether we can share it.
- Get familiar with terminology around licensing.
- We will add a license to our example project.

Copyright and derivative work: Sampling/remixing

[Midjourney, CC-BY-NC 4.0]

[Midjourney, CC-BY-NC 4.0]

- Copyright controls whether and how we can distribute the original work or the **derivative work**.
- In the **context of software** it is more about being able to change and **distribute changes**.
- Changing and distributing software is similar to changing and distributing music
- You can do almost anything if you don't distribute it

Often we don't have the choice:

- We are expected to publish software
- Sharing can be good insurance against being locked out

Can we distribute our changes with the research community or our future selves?

Why software licenses matter

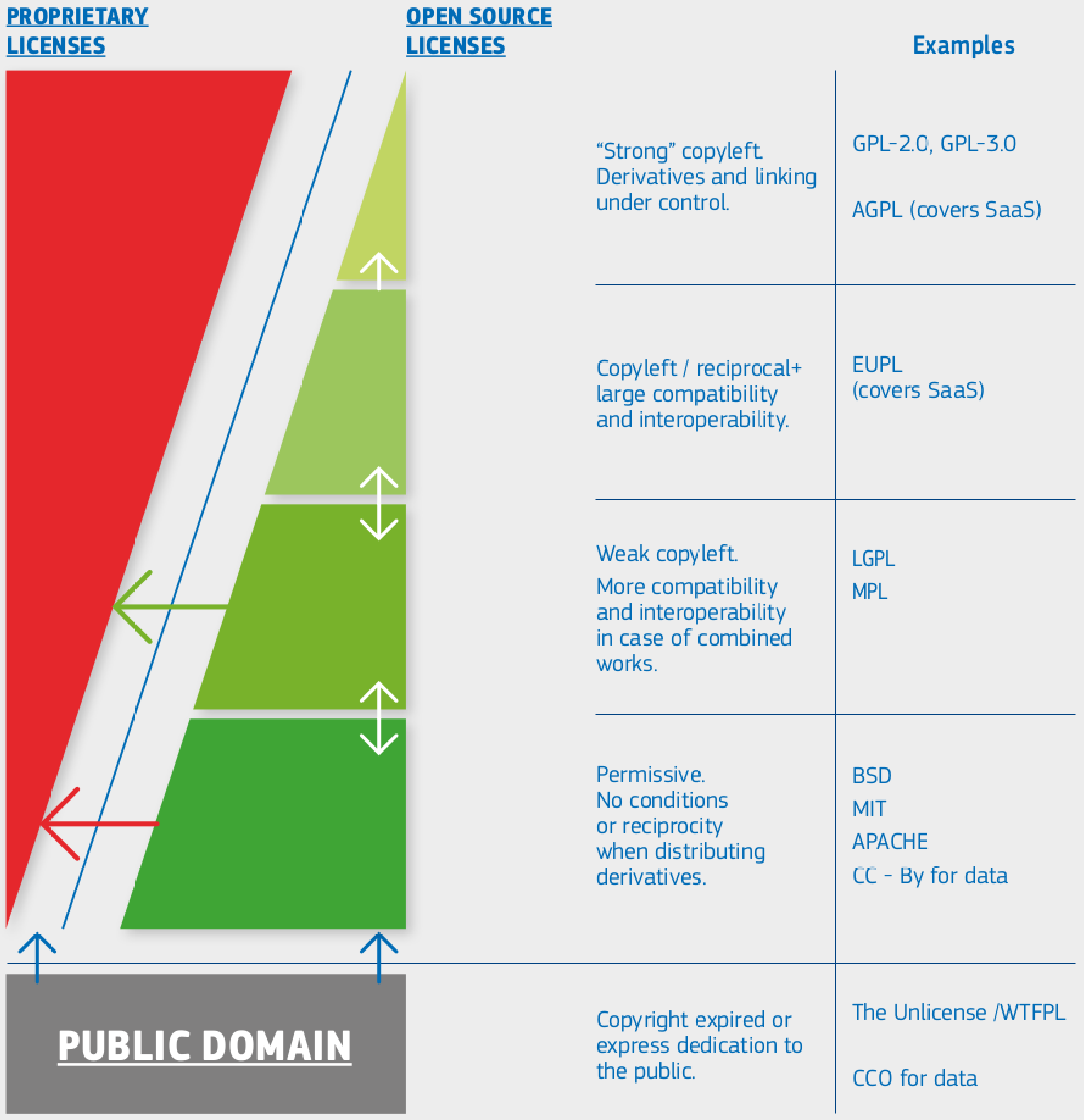
You find some great code that you want to reuse for your own publication.

- This is good for the original author - you will cite them. Maybe other people who cite you will cite them.
- You modify and remix the code.
- Two years later … ⌛
- Time to publish: You realize **there is no license to the original work** 😱

Now we have a problem:

- 😬 “Best” case: You manage to publish the paper without the software/data. Others cannot build on your software and data.
- 😱 Worst case: You cannot publish it at all. Journal requires that papers should come with data and software so that they are reproducible.

Taxonomy of software licenses



	Examples
"Strong" copyleft. Derivatives and linking under control.	GPL-2.0, GPL-3.0 AGPL (covers SaaS)
Copyleft / reciprocal+ large compatibility and interoperability.	EUPL (covers SaaS)
Weak copyleft. More compatibility and interoperability in case of combined works.	LGPL MPL
Permissive. No conditions or reciprocity when distributing derivatives.	BSD MIT APACHE CC - By for data
Copyright expired or express dedication to the public.	The Unlicense /WTFPL CCO for data

European Commission, Directorate-General for Informatics, Schmitz, P., European Union Public Licence (EUPL): guidelines July 2021, Publications Office, 2021, *https://data.europa.eu/doi/10.2799/77160*

Comments:

- Arrows represent compatibility (A -> B: B can reuse A)
- Proprietary/custom: Derivative work typically not possible (no arrow goes from proprietary to open)
- Permissive: Derivative work does not have to be shared
- Copyleft/reciprocal: Derivative work must be made available under the same license terms
- NC (non-commercial) and ND (non-derivative) exist for data licenses but not really for software licenses

Great resource for comparing software licenses: Joinup Licensing Assistant

- Provides comments on licenses
- Easy to compare licenses (example)
- Joinup Licensing Assistant - Compatibility Checker
- Not biased by some company agenda

Exercise/demo

✍ Exercise

- Let us choose a license for our example project.
- We will add a LICENSE to the repository.

💬 Discussion

- What if my code uses libraries like `scikit-image`, `scikit-learn`, `numpy`, `matplotlib`, etc. Do we need to look at their licenses? In other words, **is our project derivative work** of something else?

More resources

- Presentation slides “Practical software licensing” (R. Bast): https://doi.org/10.5281/zenodo.11554001
- Social coding lesson material
- UiT research software licensing guide (draft)
- Research institution policies to support research software (compiled by the Research Software Alliance)
- More reading material

More exercises

✍ Exercise: What constitutes derivative work?

Which of these are derivative works? Also reflect/discuss how this affects the choice of license.

- A. Download some code from a website and add on to it
- B. Download some code and use one of the functions in your code
- C. Changing code you got from somewhere
- D. Extending code you got from somewhere
- E. Completely rewriting code you got from somewhere
- F. Rewriting code to a different programming language
- G. Linking to libraries (static or dynamic), plug-ins, and drivers
- H. Clean room design (somebody explains you the code but you have never seen it)
- I. You read a paper, understand algorithm, write own code

✔ Solution

- Derivative work: A-F
- Not derivative work: G-I
- E and F: This depends on how you do it, see “clean room design”.

✍️ Exercise: Licensing situations

Consider some common licensing situations. If you are part of an exercise group, discuss these with others:

1. What is the StackOverflow license for code you copy and paste?
2. A journal requests that you release your software during publication. You have copied a portion of the code from another package, which you have forgotten. Can you satisfy the journal’s request?
3. You want to fix a bug in a project someone else has released, but there is no license. What risks are there?
4. How would you ask someone to add a license?
5. You incorporate MIT, GPL, and BSD3 licensed code into your project. What possible licenses can you pick for your project?
6. You do the same as above but add in another license that looks strong copyleft. What possible licenses can you use now?
7. Do licenses apply if you don’t distribute your code? Why or why not?
8. Which licenses are most/least attractive for companies with proprietary software?

✔ Solution

1. As indicated here, all publicly accessible user contributions are licensed under Creative Commons Attribution-ShareAlike license. See Stackoverflow Terms of service for more detailed information.
2. “Standard” licensing rules apply. So in this case, you would need to remove the portion of code you have copied from another package before being able to release your software.
3. By default you are no authorized to use the content of a repository when there is no license. And derivative work is also not possible by default. Other risks: it may not be clear whether you can use and distribute (publish) the bugfixed code. For the repo owners it may not be clear whether they can use and distributed the bugfixed code. However, the authors may have forgotten to add a license so we suggest you to contact the authors (e.g. make an issue) and ask whether they are willing to add a license.
4. As mentionned in 3., the easiest is to fill an issue and explain the reasons why you would like to use this software (or update it).
5. Combining software with different licenses can be tricky and it is important to understand compatibilities (or lack of compatibilities) of the various licenses. GPL license is the most protective (BSD and MIT are quite permissive) so for the

resulting combined software you could use a GPL license. However, re-licensing may not be necessary.

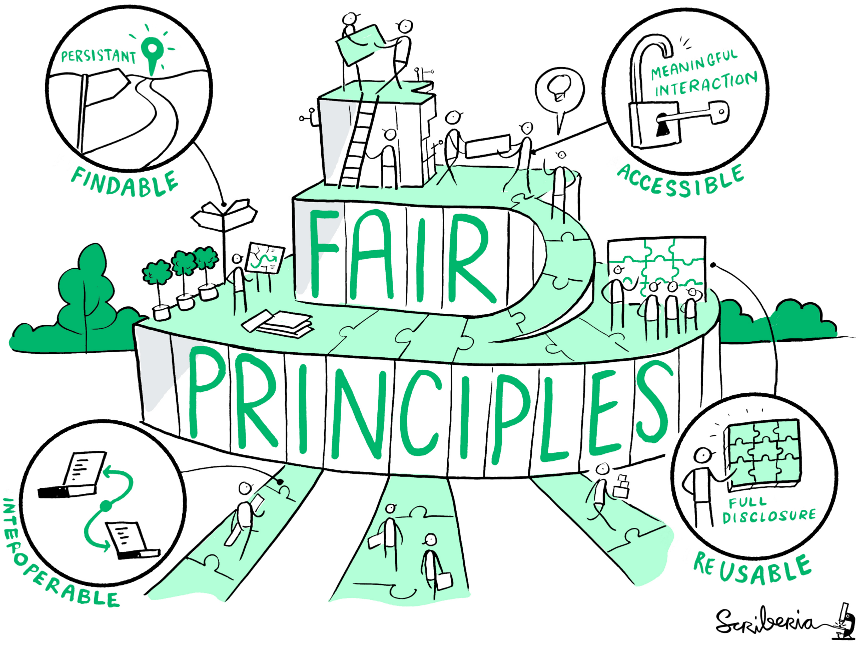
6. Derivative work would need to be shared under this strong copyleft license (e.g. AGPL or GPL), unless the components are only plugins or libraries.
7. If you keep your code for yourself, you may think you do not need a license. However, remember that in most companies/universities, your employer is “owning” your work and when you leave you may not be allowed to “distribute your code to your future self”. So the best is always to add a license!
8. The least attractive licenses for companies with proprietary software are licenses where you would need to keep an open license when creating derivative work. For instance GPL and and AGPL. The most attractive licenses are permissive licenses where they can reuse, modify and relicense with no conditions. For instance MIT, BSD and Apache License.

How to publish your code

❗ Objectives

- Make our code citable and persistent.
- Make our Notebook reusable and persistent.

Is putting software on GitHub/GitLab/… publishing?



FAIR principles. (c) Scriberia for The Turing Way, CC-BY.

Is it enough to make the code public for the code to remain **findable and accessible**?

- No. Because nothing prevents me from deleting my GitHub repository or rewriting the Git history and we have no guarantee that GitHub will still be around in 10 years.

- **Make your code citable and persistent**: Get a persistent identifier (PID) such as DOI in addition to sharing the code publicly, by using services like Zenodo or similar services.

How to make your software citable

💬 Discussion (Citation-1): Explain how you currently cite software

- Do you cite software that you use? How?
- If I wanted to cite your code/scripts, what would I need to do?

Checklist for making a release of your software citable:

- Assigned an appropriate license
- Described the software using an appropriate metadata format
- Clear version number
- Authors credited
- Procured a persistent identifier
- Added a recommended citation to the software documentation

This checklist is adapted from: N. P. Chue Hong, A. Allen, A. Gonzalez-Beltran, et al., Software Citation Checklist for Developers (Version 0.9.0). Zenodo. 2019b. (DOI)

Our practical recommendations:

- Add a file called CITATION.cff (example).
- Get a digital object identifier (DOI) for your code on Zenodo (example).
- Make it as easy as possible: clearly say what you want cited.

This is an example of a simple `CITATION.cff` file:

```yaml
cff-version: 1.2.0
message: "If you use this software, please cite it as below."
authors:
  - family-names: Doe
    given-names: Jane
    orcid: https://orcid.org/1234-5678-9101-1121
title: "My Research Software"
version: 2.0.4
doi: 10.5281/zenodo.1234
date-released: 2021-08-11
```

More about `CITATION.cff` files:

- GitHub now supports CITATION.cff files
- Web form to create, edit, and validate CITATION.cff files
- Video: “How to create a CITATION.cff using cffinit”

Papers with focus on scientific software

Where can I publish papers which are primarily focused on my scientific software? Great list/summary is provided in this blog post: “In which journals should I publish my software?” (Neil P. Chue Hong)

How to cite software

❗ Great resources

- A. M. Smith, D. S. Katz, K. E. Niemeyer, and FORCE11 Software Citation Working Group, “Software citation principles,” PeerJ Comput. Sci., vol. 2, no. e86, 2016 (DOI)
- D. S. Katz, N. P. Chue Hong, T. Clark, et al., Recognizing the value of software: a software citation guide [version 2; peer review: 2 approved]. F1000Research 2021, 9:1257 (DOI)
- N. P. Chue Hong, A. Allen, A. Gonzalez-Beltran, et al., Software Citation Checklist for Authors (Version 0.9.0). Zenodo. 2019a. (DOI)
- N. P. Chue Hong, A. Allen, A. Gonzalez-Beltran, et al., Software Citation Checklist for Developers (Version 0.9.0). Zenodo. 2019b. (DOI)

Recommended format for software citation is to ensure the following information is provided as part of the reference (from Katz, Chue Hong, Clark, 2021 which also contains software citation examples):

- Creator
- Title
- Publication venue
- Date
- Identifier
- Version
- Type

Exercise/demo

✍ Exercise

- We will add a `CITATION.cff` file to our example repository.
- We will get a DOI using the Zenodo sandbox:
  - We will log into the Zenodo sandbox using GitHub.
  - We will follow these steps and finally create a GitHub release and get a DOI.
- We will use the Binder badge on our example repository to run the Notebook in the cloud and discuss how we could make it persistent and citable.

💬 Discussion

- Why did we use the Zenodo sandbox and not the “real” Zenodo for our exercise?

More resources

- Social coding lesson material
- Sharing Jupiter Notebooks

Creating a Python package and deploying it to PyPI

Objectives

In this episode we will create a pip-installable Python package and learn how to deploy it to PyPI. As example, we will use the star counting script which we created in the previous episode.

Creating a Python package with the help of flit

There are unfortunately many ways to package a Python project:

- `setuptools` is the most common way to package a Python project. It is very powerful and flexible, but also can get complex.
- `flit` is a simpler alternative to `setuptools`. It is less powerful, but also easier to use.
- `poetry` is a modern packaging tool which is more powerful than `flit` and also easier to use than `setuptools`.
- `twine` is another tool to upload packages to PyPI.
- …

This will be a demo

- We will try to package the code together on the big screen.
- We will share the result on GitHub so that you can retrace the steps.
- In this demo, we will use Flit to package the code. From Why use Flit?:

  > Make the easy things easy and the hard things possible is an old motto from the Perl community. Flit is entirely focused on the easy things part of that, and leaves the hard things up to other tools.

Step 1: Initialize the package metadata and try a local install

1. Our starting point is that we have a Python script called `countstars.py` which we want to package.
2. Now we follow the flit quick-start usage and add a docstring to the script and a `__version__`.
3. We then run `flit init` to create a `pyproject.toml` file and answer few questions. I obtained:

```
[build-system]
requires = ["flit_core >=3.2,<4"]
build-backend = "flit_core.buildapi"

[project]
name = "countstars"
authors = [{name = "Radovan Bast", email = "radovan.bast@uit.no"}]
license = {file = "LICENSE"}
classifiers = ["License :: OSI Approved :: MIT License"]
dynamic = ["version", "description"]

[project.urls]
Home = "https://github.com/workshop-material/countstars"
```

4. We now add dependencies and also an entry point for the script:

```
[build-system]
requires = ["flit_core >=3.2,<4"]
build-backend = "flit_core.buildapi"

[project]
name = "countstars"
authors = [{name = "Radovan Bast", email = "radovan.bast@uit.no"}]
license = {file = "LICENSE"}
classifiers = ["License :: OSI Approved :: MIT License"]
dynamic = ["version", "description"]
dependencies = [
    "click",
    "matplotlib",
    "scikit-image",
]

[project.urls]
Home = "https://github.com/workshop-material/countstars"

[project.scripts]
count-stars = "countstars:main"
```

5. Before moving on, try a local install:

```
$ flit install --symlink
```

💬 What if you have more than just one script?

- Create a directory with the name of the package.
- Put the scripts in the directory.
- Add a `__init__.py` file to the directory which contains the module docstring and the version and re-exports the functions from the scripts.

Step 2: Testing an install from GitHub

If a local install worked, push the `pyproject.toml` to GitHub and try to install the package from GitHub.

In a `requirements.txt` file, you can specify the GitHub repository and the branch:

```
git+https://github.com/workshop-material/countstars.git@main
```

A corresponding `envionment.yml` file for conda would look like this:

```yaml
name: countstars
channels:
  - conda-forge
dependencies:
  - python=3.12
  - pip
  - pip:
      - git+https://github.com/workshop-material/countstars.git@main
```

Does it install and run? If yes, move on to the next step (test-PyPI and later PyPI).

Step 3: Deploy the package to test-PyPI using GitHub Actions

Our final step is to create a GitHub Actions workflow which will run when we create a new release.

> **❗ Danger**
>
> I recommend to first practice this on the test-PyPI before deploying to the real PyPI.

Here is the workflow (`.github/workflows/package.yml`):

```yaml
name: Package

on:
  release:
    types: [created]

jobs:
  build:
    permissions: write-all
    runs-on: ubuntu-latest

    steps:
    - name: Switch branch
      uses: actions/checkout@v4
    - name: Set up Python
      uses: actions/setup-python@v5
      with:
        python-version: "3.12"
    - name: Install Flit
      run: |
        pip install flit
    - name: Flit publish
      run:
        flit publish
      env:
        FLIT_USERNAME: __token__
        FLIT_PASSWORD: ${{ secrets.PYPI_TOKEN }}
        # uncomment the following line if you are using test.pypi.org:
#       FLIT_INDEX_URL: https://test.pypi.org/legacy/
        # this is how you can test the installation from test.pypi.org:
#       pip install --index-url https://test.pypi.org/simple/ package_name
```

About the `FLIT_PASSWORD: ${{ secrets.PYPI_TOKEN }}`:

- You obtain the token from PyPI by going to your account settings and creating a new token.
- You add the token to your GitHub repository as a secret (here with the name `PYPI_TOKEN`).

Credit

The following material (all CC-BY) was reused to create this workshop material:

- https://coderefinery.github.io/research-software-engineering/
- https://coderefinery.github.io/mini-workshop/
- https://coderefinery.github.io/git-collaborative/
- https://coderefinery.github.io/git-intro/
- https://coderefinery.github.io/documentation/
- https://coderefinery.github.io/reproducible-research/