Reproducible research software development using Python (ML example)

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 are 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 (Example project: 2D classification task using a nearest-neighbor predictor) 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**.

⚙ 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. Follow Software install instructions (but we will do this together at the beginning of the workshop).

Schedule

Monday

- 09:00-10:00 - Getting started
  - Welcome and introduction
  - Software install instructions
  - Example project: 2D classification task using a nearest-neighbor predictor
- 10:15-11:30 - Introduction to version control with Git and GitHub (1/2)
  - Motivation
  - Forking, cloning, and browsing
  - Creating branches and commits
- 11:30-12:15 - Lunch break
- 12:15-13:30 - Introduction to version control with Git and GitHub (2/2)

    - Merging changes and contributing to the project
    - Conflict resolution
    - Practical advice: How much Git is necessary?
    - Optional: How to turn your project to a Git repo and share it
- 13:45-15:00 - Where to start with documentation

Tuesday

- 09:00-10:00 - Collaborative version control and code review (1/2)
    - Concepts around collaboration
    - Collaborating within the same repository
- 10:15-11:30 - Collaborative version control and code review (2/2)
    - Practicing code review
    - How to contribute changes to repositories that belong to others
- 11:30-12:15 - Lunch break
- 12:15-12:45 - Reproducible environments and dependencies
- 12:45-13:30 - Working with Notebooks
    - Notebooks and version control
    - Other useful tooling for notebooks
    - Sharing notebooks
- 13:30-14:15 - Other useful tools for Python development
    - Profiling
    - Tools and useful practices
- 14:15-15:00 - 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

Wednesday

- 09:00-10:00 - Automated testing
- 10:15-11:30 - Modular code development
    - Concepts in refactoring and modular code design
    - How to parallelize independent tasks using workflows (example: Snakemake)
- 11:30-12:15 - Lunch break
- 12:15-14:00 - How to release and publish your code
    - Choosing a software license
    - How to publish your code
    - Creating a Python package and deploying it to PyPI
- 14:15-15:00 - 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

Thursday

- 09:00-09:30 - Concepts of high-performance computing
- 09:30-10:15 - Running AI training jobs in HPC
- 10:30-11:30 - Using, building and extending containers in HPC
- 11:30-12:15 - Lunch break
- 12:15-13:00 - Scaling AI training to multiple GPUs
- 13:00-14:00 - Monitoring and profiling AI training jobs
- 14:00-15:00 - Debriefing and introduction to the exam

Software install instructions

[this page is adapted from https://aaltoscicomp.github.io/python-for-scicomp/installation/]

Choosing an installation method

For this course we will install an **isolated environment** with following dependencies:

environment.yml | requirements.txt

```yaml
name: course
channels:
  - conda-forge
  - bioconda
dependencies:
  - python <= 3.12
  - click
  - numpy
  - pandas
  - scipy
  - altair
  - vl-convert-python
  - jupyterlab
  - pytest
  - scalene
  - flit
  - ruff
  - icecream
  - snakemake-minimal
  - myst-parser
  - sphinx
  - sphinx-rtd-theme
  - sphinx-autoapi
  - sphinx-autobuild
  - black
  - isort
  - pip
  - pip:
    - jupyterlab-code-formatter
```

❗ If you have a tool/method that works for you, keep using it

If you are used to installing packages in Python and know what to do with the above files, please follow your own preferred installation method.

If you are new to Python or unsure how to create isolated environments in Python from files above, please follow the instructions below.

💬 There are many choices and we try to suggest a good compromise

There are very many ways to install Python and packages with pros and cons and in addition there are several operating systems with their own quirks. This can be a huge challenge for beginners to navigate. It can also difficult for instructors to give recommendations for something which will work everywhere and which everybody will like.

Below we will recommend **Miniforge** since it is free, open source, general, available on all operating systems, and provides a good basis for reproducible environments. However, it does not provide a graphical user interface during installation. This means that every time we want to start a JupyterLab session, we will have to go through the command line.

❗ Python, conda, anaconda, miniforge, etc?

Unfortunately there are many options and a lot of jargon. Here is a crash course:

- **Python** is a programming language very commonly used in science, it's the topic of this course.
- **Conda** is a package manager: it allows distributing and installing packages, and is designed for complex scientific code.
- **Mamba** is a re-implementation of Conda to be much faster with resolving dependencies and installing things.
- An **environment** is a self-contained collections of packages which can be installed separately from others. They are used so each project can install what it needs without affecting others.
- **Anaconda** is a commercial distribution of Python+Conda+many packages that all work together. It used to be freely usable for research, but since ~2023-2024 it's more limited. Thus, we don't recommend it (even though it has a nice graphical user interface).
- **conda-forge** is another channel of distributing packages that is maintained by the community, and thus can be used by anyone. (Anaconda's parent company also hosts conda-forge packages)
- **Miniforge** is a distribution of conda pre-configured for conda-forge. It operates via the command line.
- **Miniconda** is a distribution of conda pre-configured to use the Anaconda channels.

We will gain a better background and overview in the section Reproducible environments and dependencies.

Installing Python via Miniforge

Follow the instructions on the miniforge web page. This installs the base, and from here other packages can be installed.

Unsure what to download and what to do with it?

Windows | MacOS | Linux

You want to download and run `Miniforge3-Windows-x86_64.exe`.

Installing and activating the software environment

First we will start Python in a way that activates conda/mamba. Then we will install the software environment from this environment.yml file.

An **environment** is a self-contained set of extra libraries - different projects can use different environments to not interfere with each other. This environment will have all of the software needed for this particular course.

We will call the environment `course`.

Windows | Linux / MacOS

Use the "Miniforge Prompt" to start Miniforge. This will set up everything so that `conda` and `mamba` are available. Then type (without the `$`):

```
$ mamba env create -n course -f
https://raw.githubusercontent.com/coderefinery/reproducible-python-
ml/main/software/environment.yml
```

If this throws an error you can download the `environmet.yml` file first and set up the environment via

```
$ curl -O https://raw.githubusercontent.com/coderefinery/reproducible-python-
ml/main/software/environment.yml
$ mamba env create -n course -f environment.yml
```

Starting JupyterLab

Every time we want to start a JupyterLab session, we will have to go through the command line and first activate the `course` environment.

Windows | Linux / MacOS

Start the Miniforge Prompt. Then type (without the `$`):

```
$ conda activate course
$ jupyter-lab
```

Removing the software environment

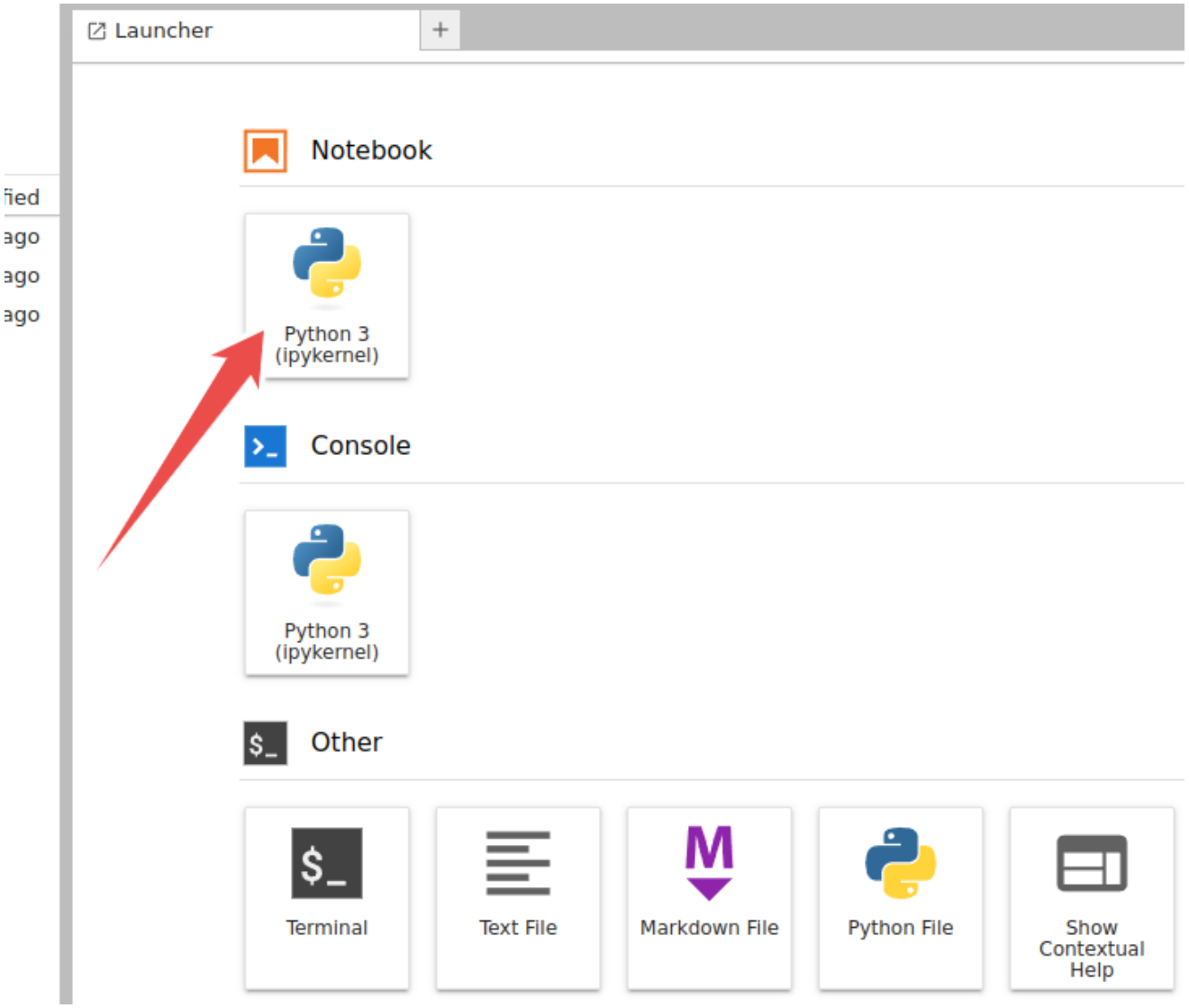
Windows | Linux / MacOS

In the Miniforge Prompt, type (without the `$`):

```
$ conda env list
$ conda env remove --name course
$ conda env list
```

How to verify your installation

Start JupyterLab (as described above). It will hopefully open up your browser and look like this:



JupyterLab opened in the browser. Click on the Python 3 tile.

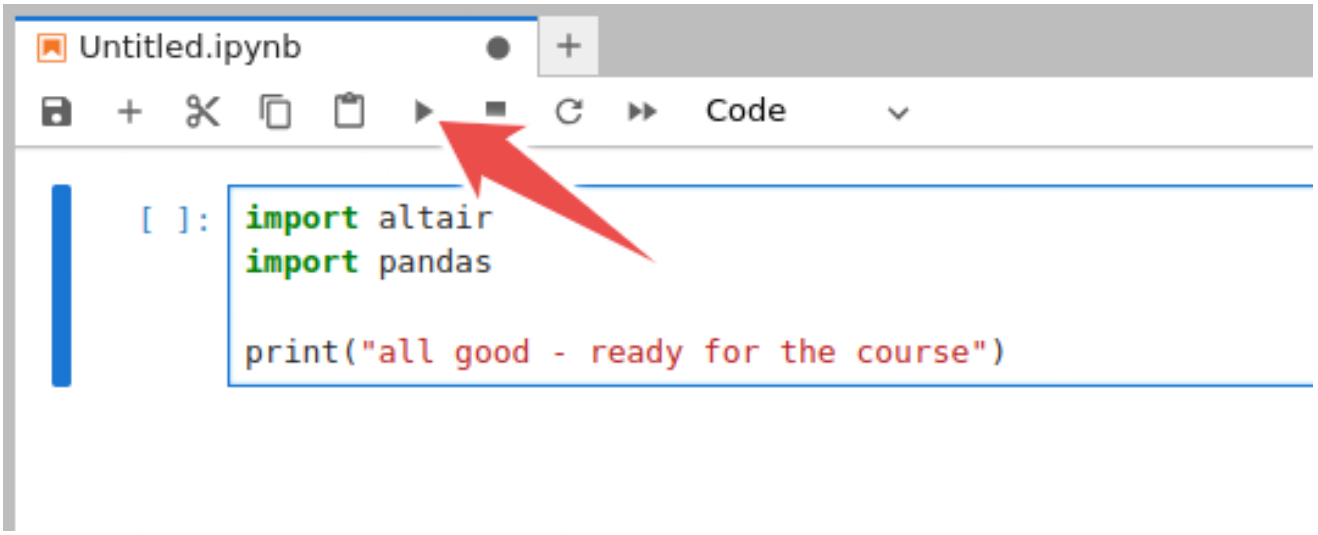
Once you clicked the Python 3 tile it should look like this:

Python 3 notebook started.

Into that blue “cell” please type the following:

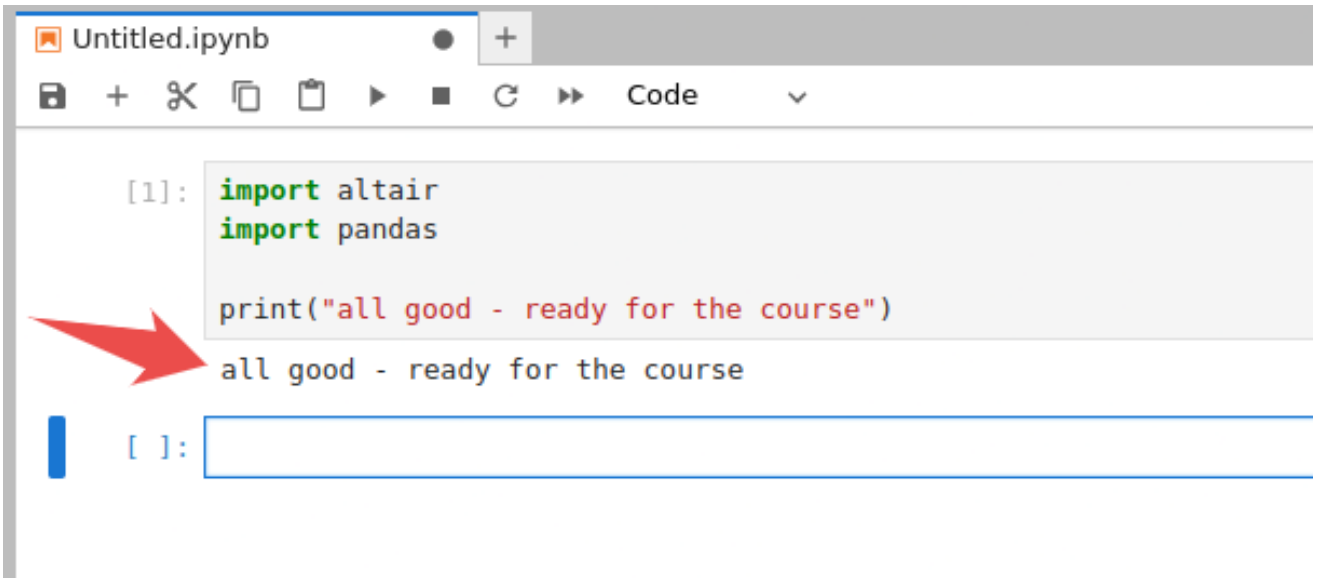
```
import altair
import pandas

print("all good - ready for the course")
```



Please copy these lines and click on the "play"/"run" icon.

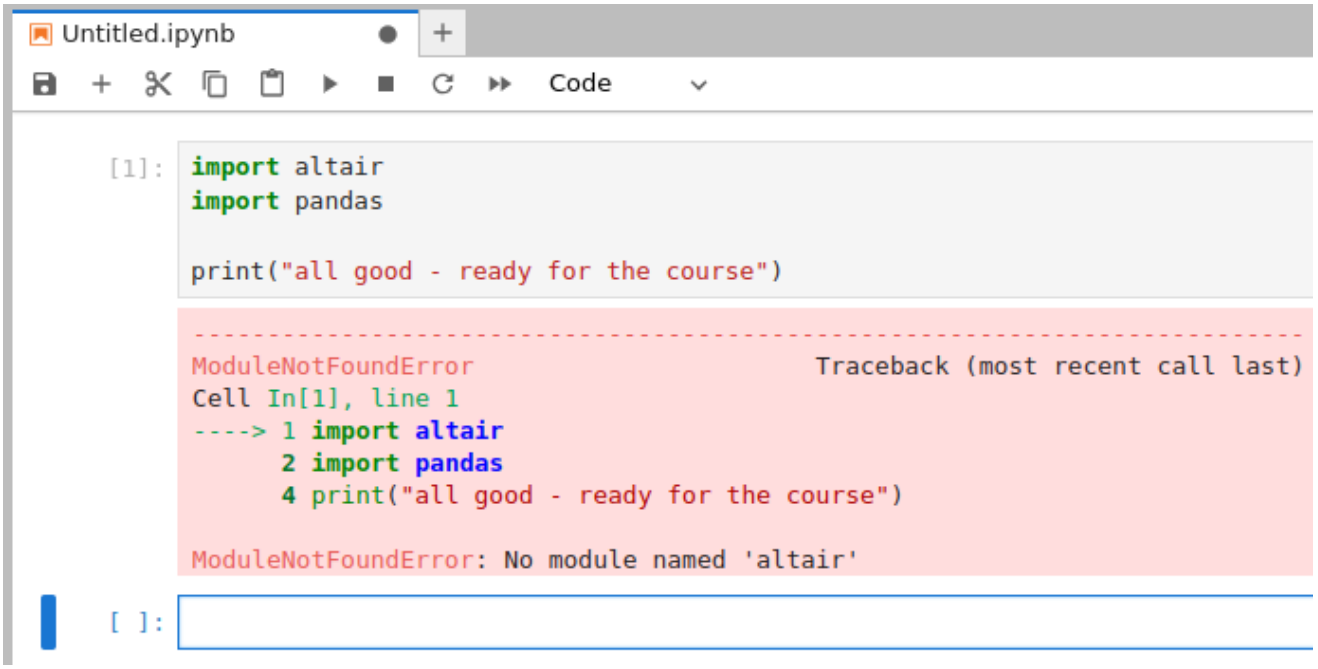
This is how it should look:



Screenshot after successful import.

If this worked, you are all set and can close JupyterLab (no need to save these changes).

This is how it **should not** look:

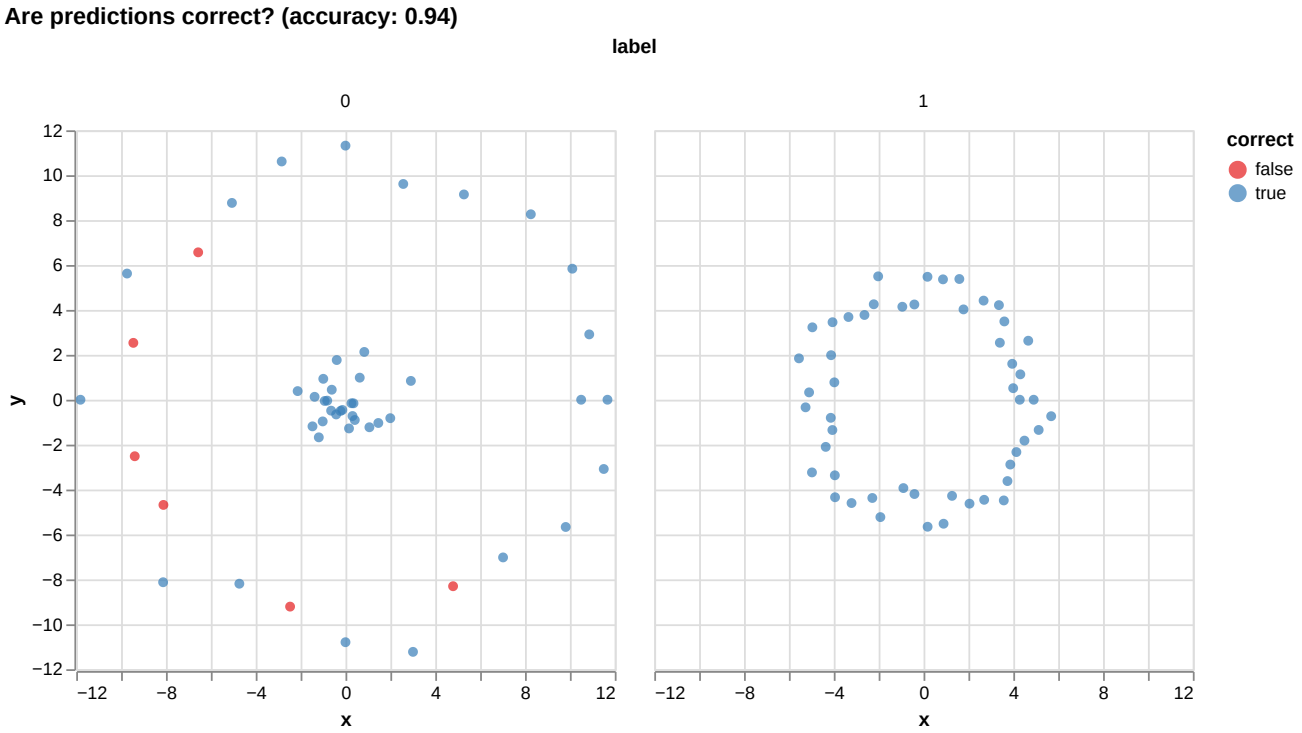


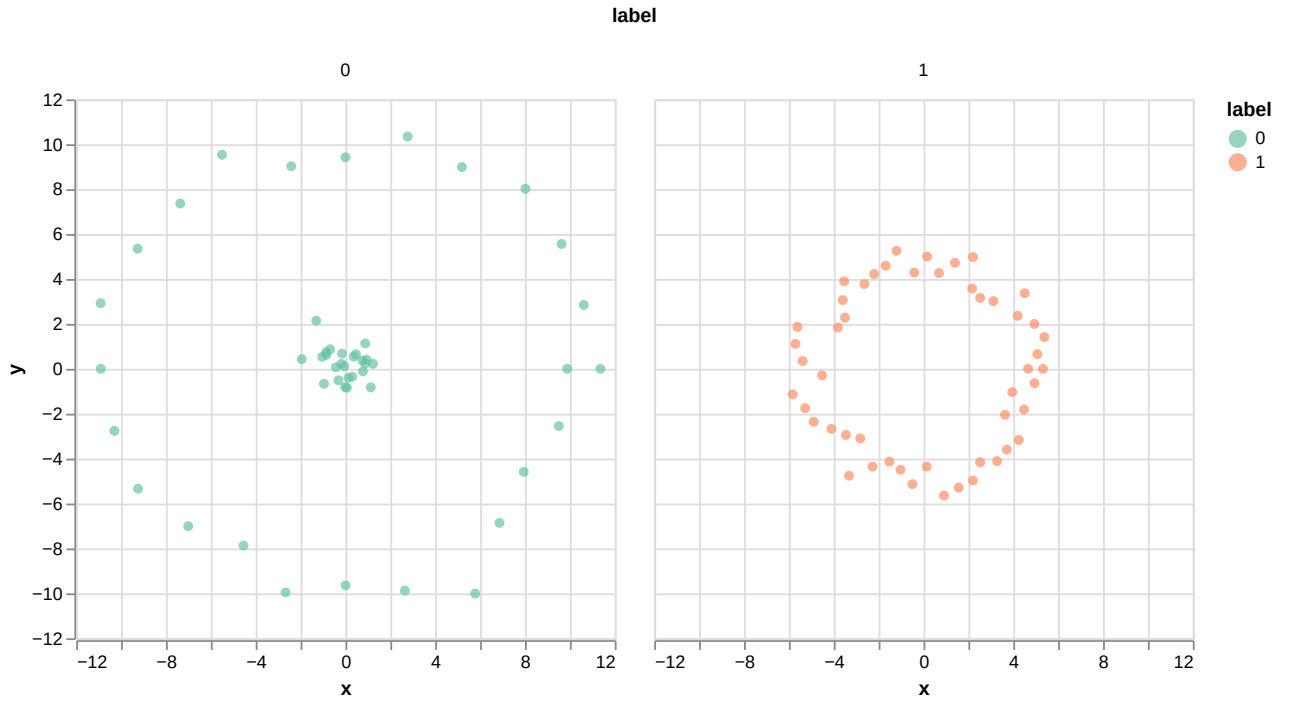
Error: required packages could not be found.

Example project: 2D classification task using a nearest-neighbor predictor

The example code that we will study is a relatively simple nearest-neighbor predictor written in Python. It is not important or expected that we understand the code in detail.

The code will produce something like this:





The bottom row shows the training data (two labels) and the top row shows the test data and whether the nearest-neighbor predictor classified their labels correctly.

The **big picture** of the code is as follows:

- We can choose the number of samples (the example above has 50 samples).
- The code will generate samples with two labels (0 and 1) in a 2D space.
- One of the labels has a normal distribution and a circular distribution with some minimum and maximum radius.
- The second label only has a circular distribution with a different radius.
- Then we try to predict whether the test samples belong to label 0 or 1 based on the nearest neighbors in the training data. The number of neighbors can be adjusted and the code will take label of the majority of the neighbors.

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 samples and file names. Later we will discuss advantages of this approach.

Let us try to get the help text:

```
$ python generate_data.py --help

Usage: generate_data.py [OPTIONS]

  Program that generates a set of training and test samples for a non-linear
  classification task.

Options:
  --num-samples INTEGER  Number of samples for each class.  [required]
  --training-data TEXT   Training data is written to this file.  [required]
  --test-data TEXT       Test data is written to this file.  [required]
  --help                 Show this message and exit.
```

We first generate the training and test data:

```
$ python generate_data.py --num-samples 50 --training-data train.csv --test-data
test.csv

Generated 50 training samples (train.csv) and test samples (test.csv).
```

In a second step we generate predictions for the test data:

```
$ python generate_predictions.py --num-neighbors 7 --training-data train.csv --test-
data test.csv --predictions predictions.csv

Predictions saved to predictions.csv
```

Finally, we can plot the results:

```
$ python plot_results.py --training-data train.csv --predictions predictions.csv --
output-chart chart.svg

Accuracy: 0.94
Saved chart to chart.svg
```

Discussion and goals

💬 Discussion

- Together we look at the generated files (train.csv, test.csv, predictions.csv, chart.svg).
- We browse and discuss the example code behind these scripts.

❗ 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 system size.
- ... 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/classification-task
- 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/classification-task/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

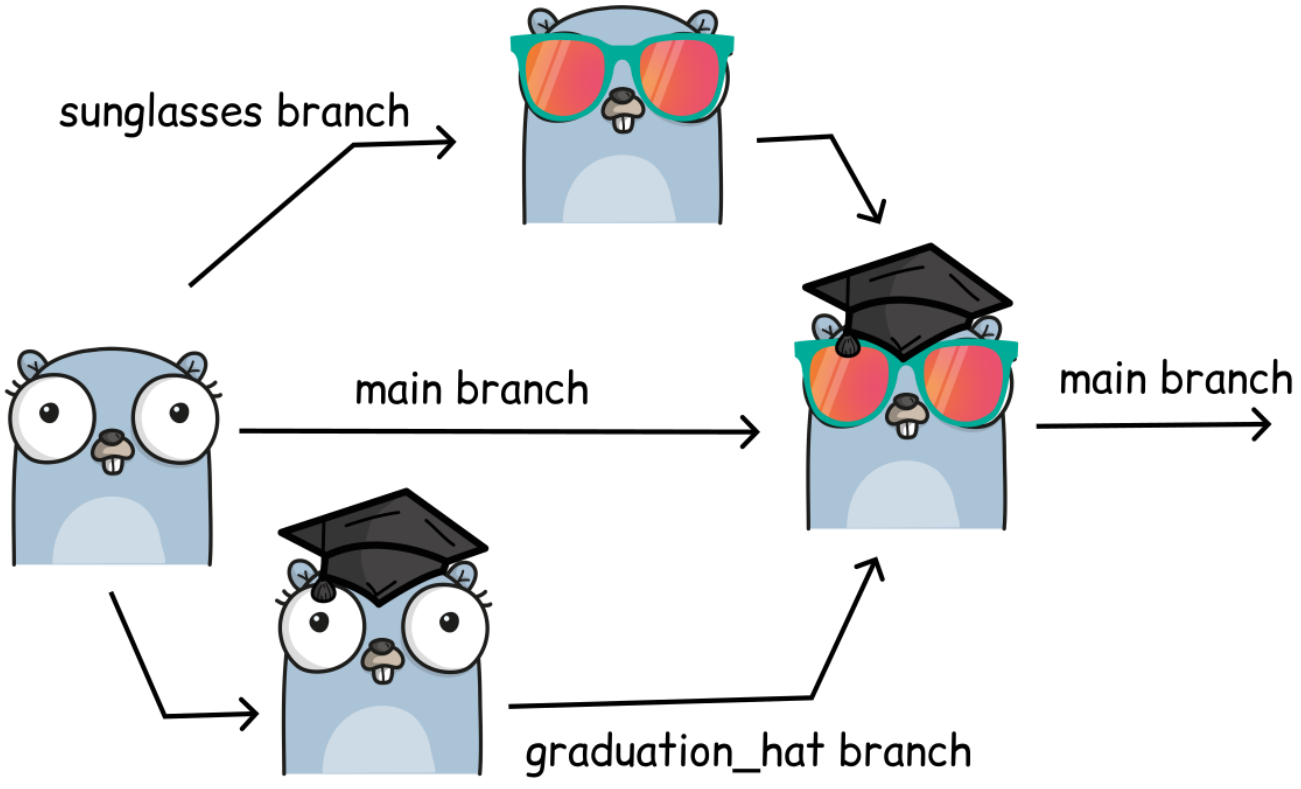


Image created using https://gopherize.me/ (inspiration).

- **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 'generate_predictions.py', and search for 'majority_index'. Oh! But make sure you use the version from January 2025."
2. Or I can send you a permalink: https://github.com/workshop-material/classification-task/blob/79ce3be8/generate_predictions.py#L25-L28

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.

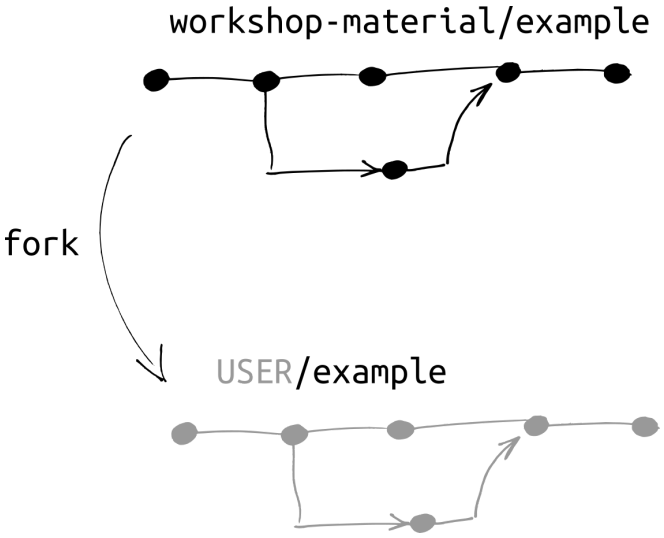


Illustration of forking a repository on GitHub.

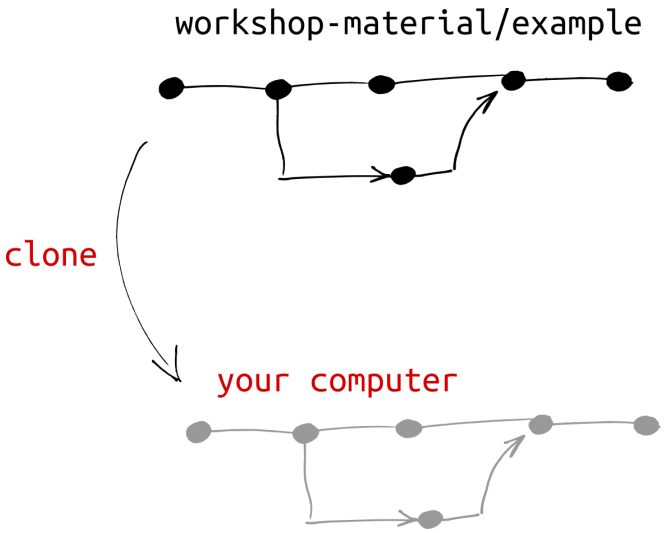
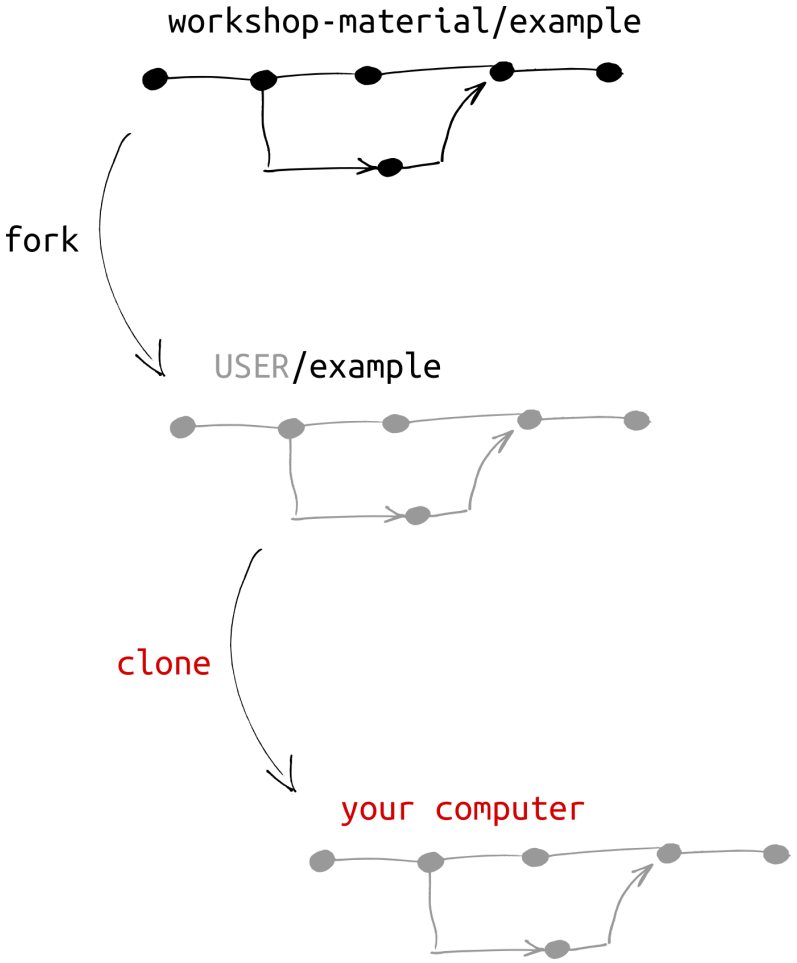


Illustration of cloning a repository to your computer.



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**/classification-task
- Upstream repository: https://github.com/**workshop-material**/classification-task

How to create a fork

1. Go to the repository view on GitHub: https://github.com/workshop-material/classification-task

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/classification-task**.

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**/classification-task, 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. `generate_predictions.py`. (Hint: “History” button in the file view)
4. **Which files include the word “training”**? (Hint: the GitHub search on top of the repository view)
5. In the `generate_predictions.py` file, find out who modified the evaluation of “majority_index” 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

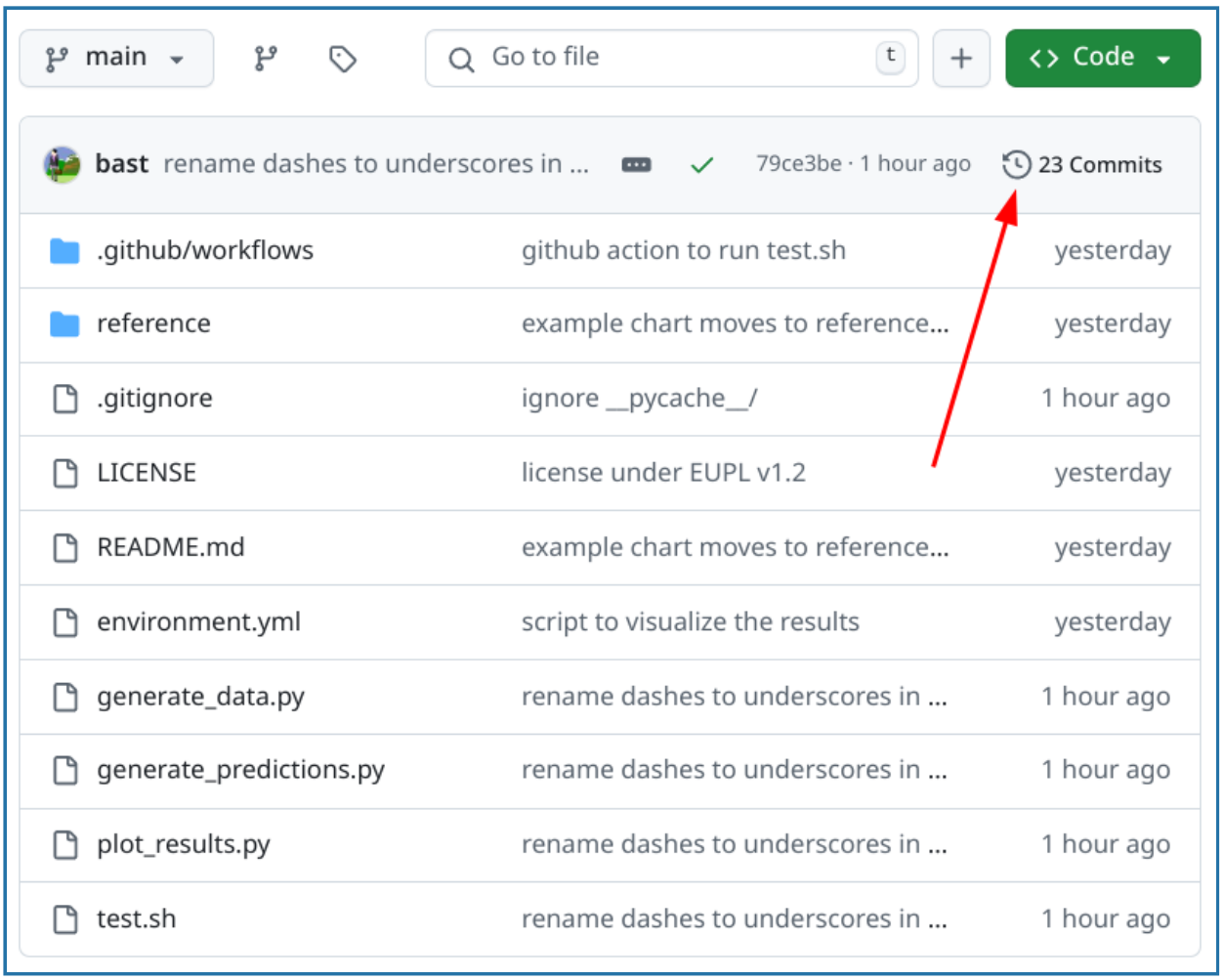
(1) Basic browsing

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 `79ce3be`. 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:

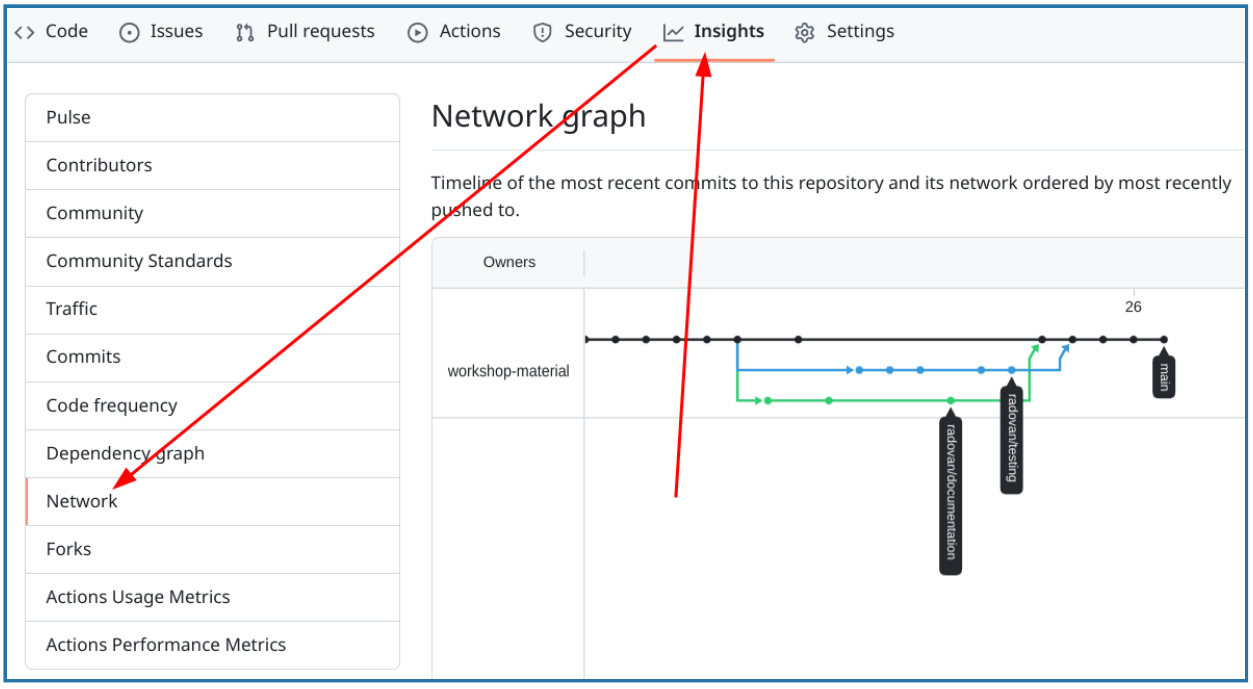


(2) Compare commit history with network graph

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:



(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 → generate_predictions.py. Click the “History” button near the top right.

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

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 “training”, 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?

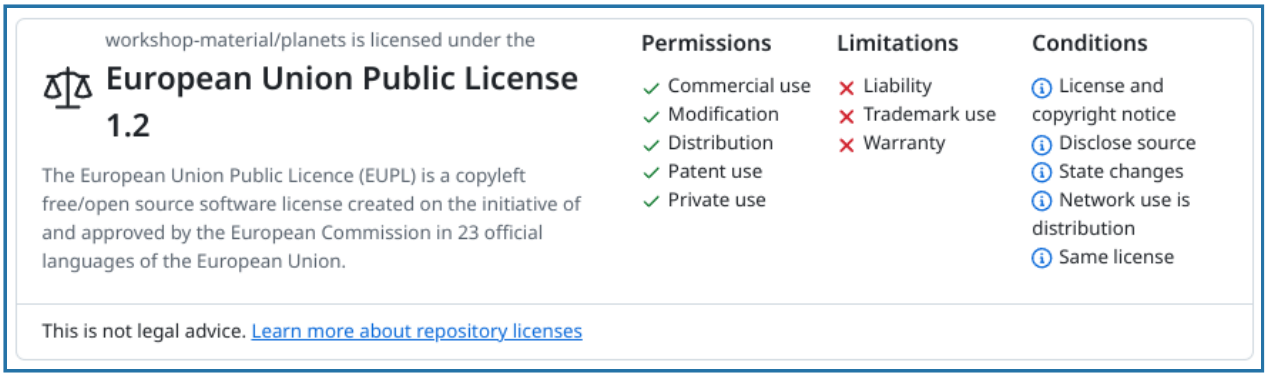
This is called the “annotate” or “blame” view. The name “blame” is very unfortunate, but it is the standard term for historical reasons for this functionality and it is not meant to blame anyone.

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:



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

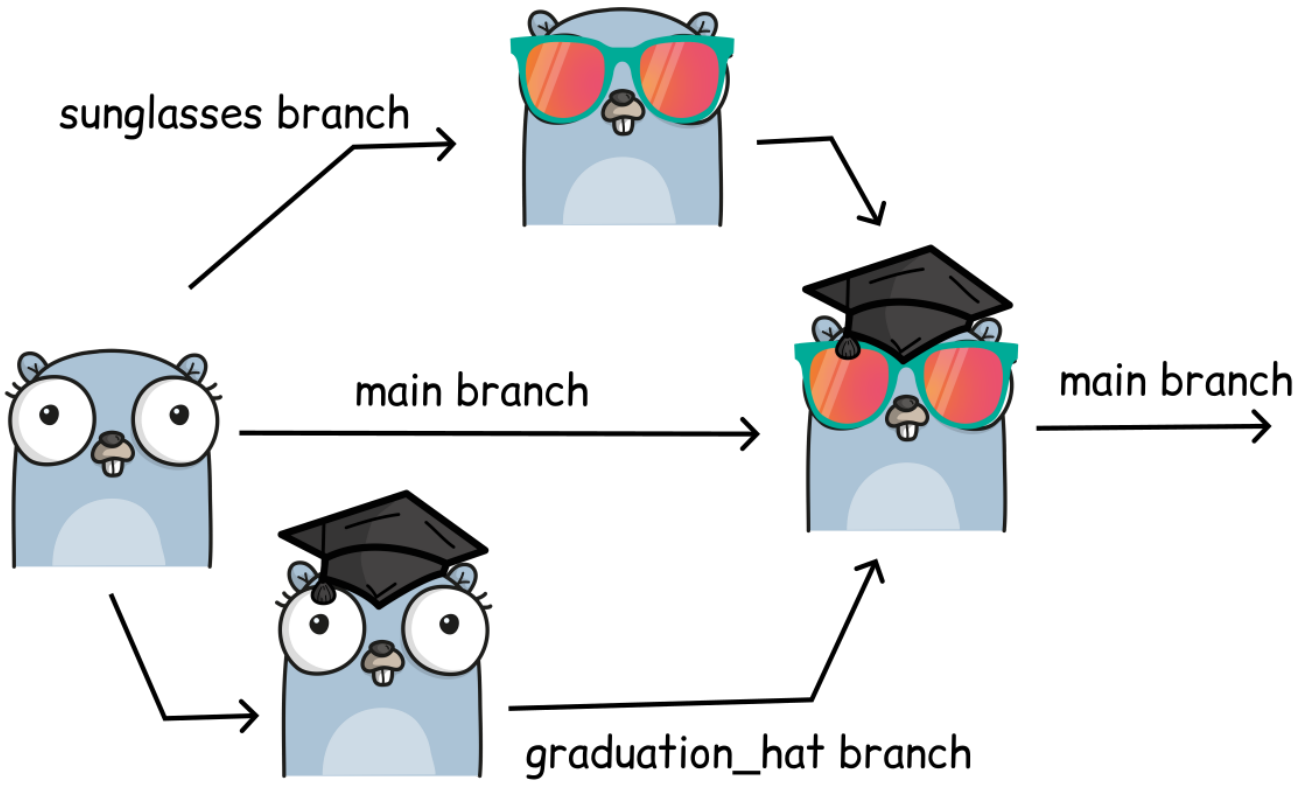
The first and most basic task to do in Git is **record changes** using commits. In this part, we will record changes in two ways: on a **new branch** (which supports multiple lines of work at once), and directly on the “main” branch (which happens to be the default branch here).

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

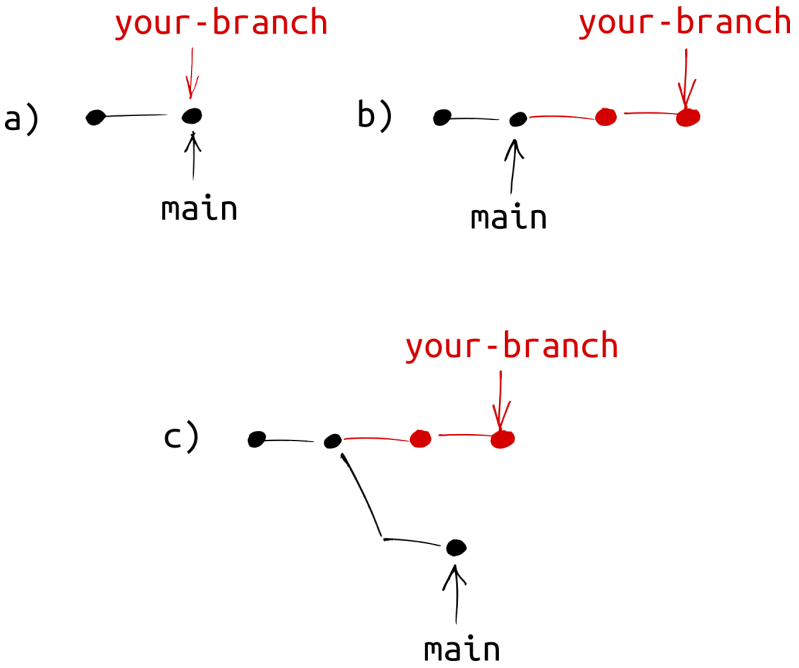


Illustration of what we want to achieve in this exercise.

✍ 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 or improve the documentation.
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").
9. Optional: If this was too easy, you can try to create a new branch and on this branch work on one of these new features:
   - The random seed is now set to a specific number (42). Make it possible to set the seed to another number or to turn off the seed setting via the command line interface.
   - Move the code that does the majority vote to an own function.
   - Write a test for the new majority vote function.
   - The `num_neighbors` in the code needs to be odd. Change the code so that it stops with an error message if an even number is given.
   - Add type annotations to some functions.
   - When calling the `scatter_plot` function, call the function with named arguments.
   - Add example usage to README.md.

- Add a Jupyter Notebook version of the example code.

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**/classification-task/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**/classification-task/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

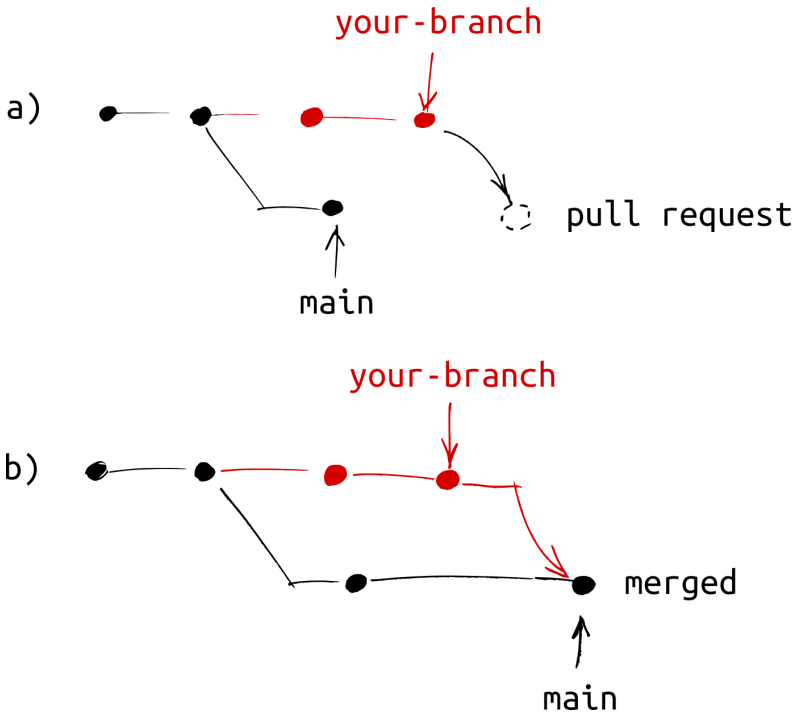


Illustration of what we want to achieve in this exercise.

✍ 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.

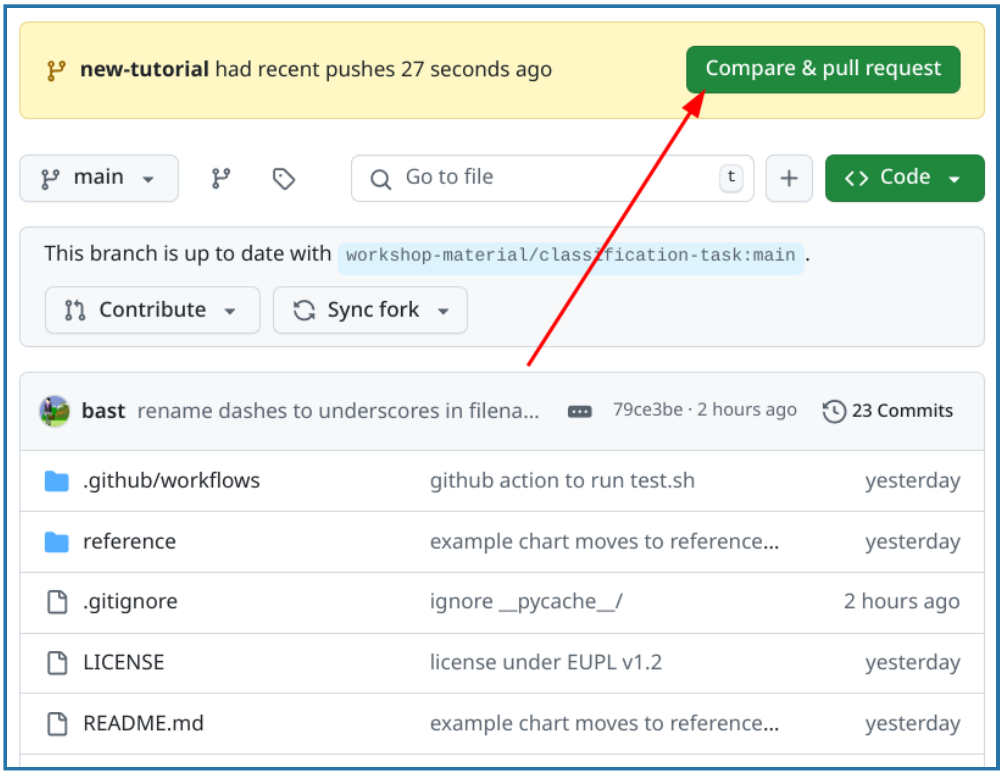
Solution and walk-through

(1) Navigate to your branch

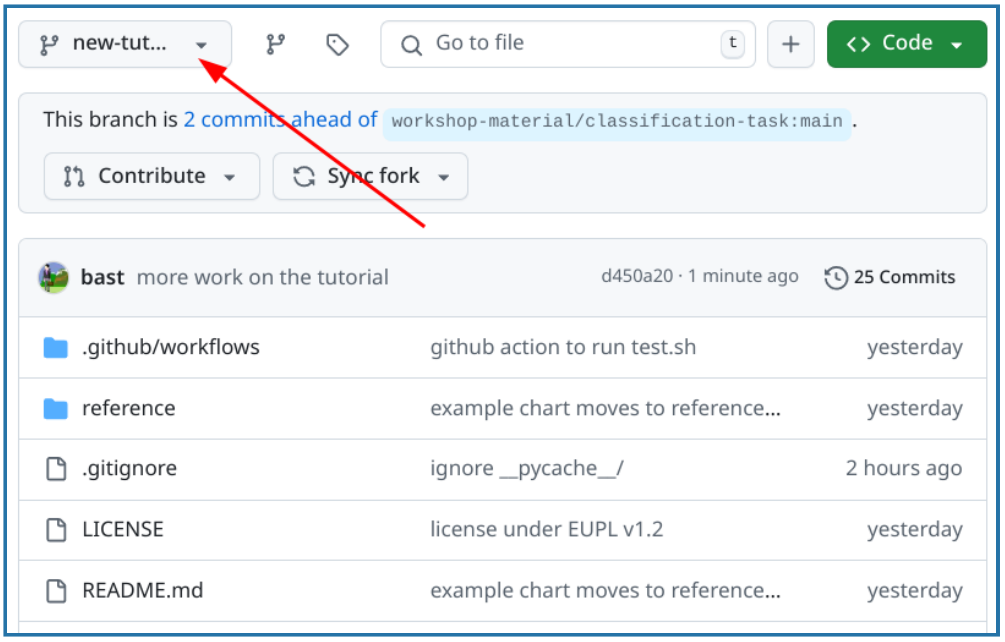
Before making the pull request, or doing a merge, it's important to make sure that you are on the right branch. Many people have been frustrated because they forgot this!

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):



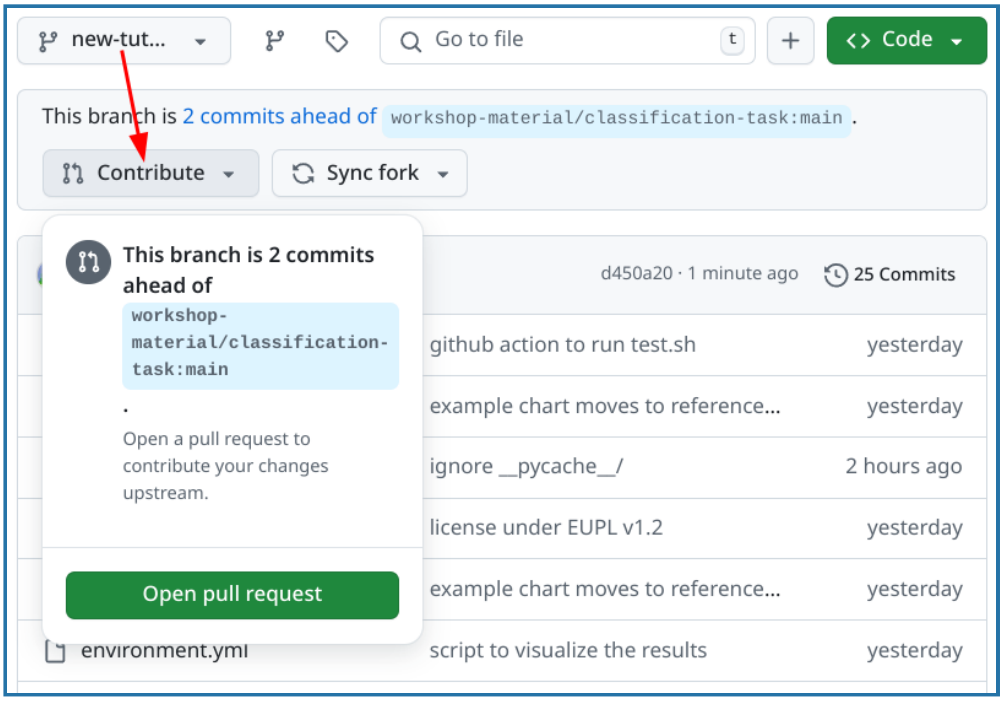
If the yellow box is not there, make sure you are on the branch you want to merge **from**:



(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



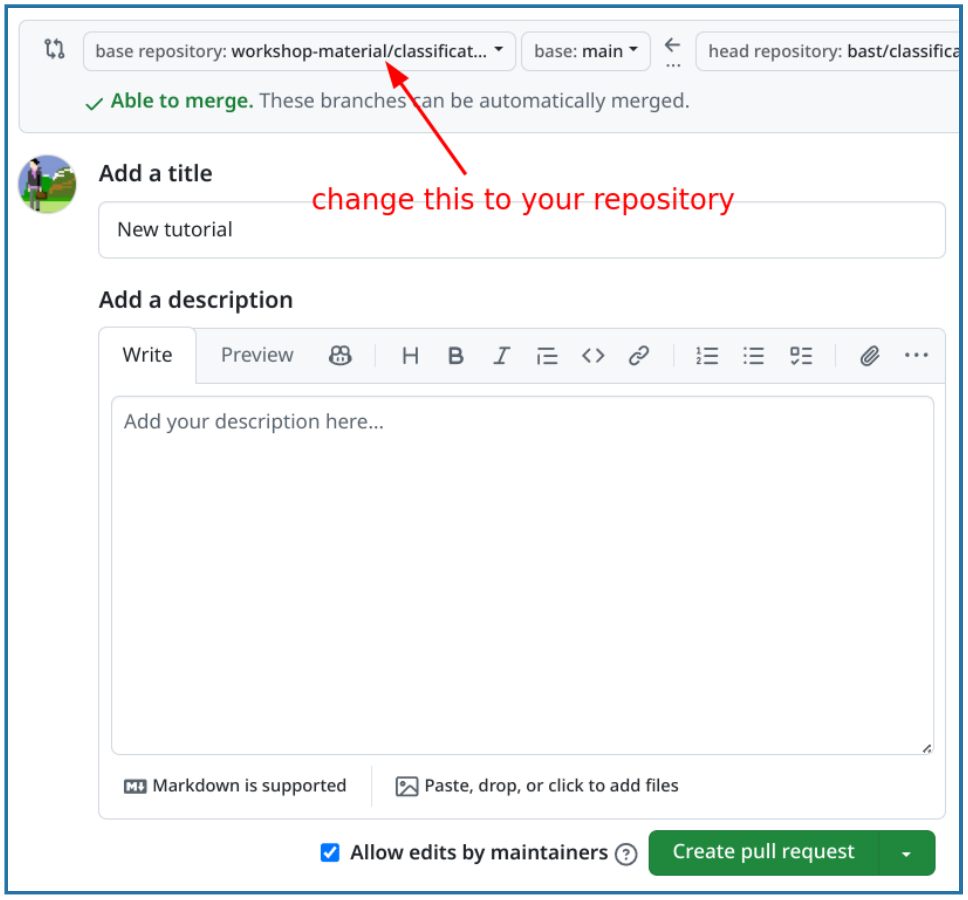
(3) Fill out and verify the pull request

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?



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.

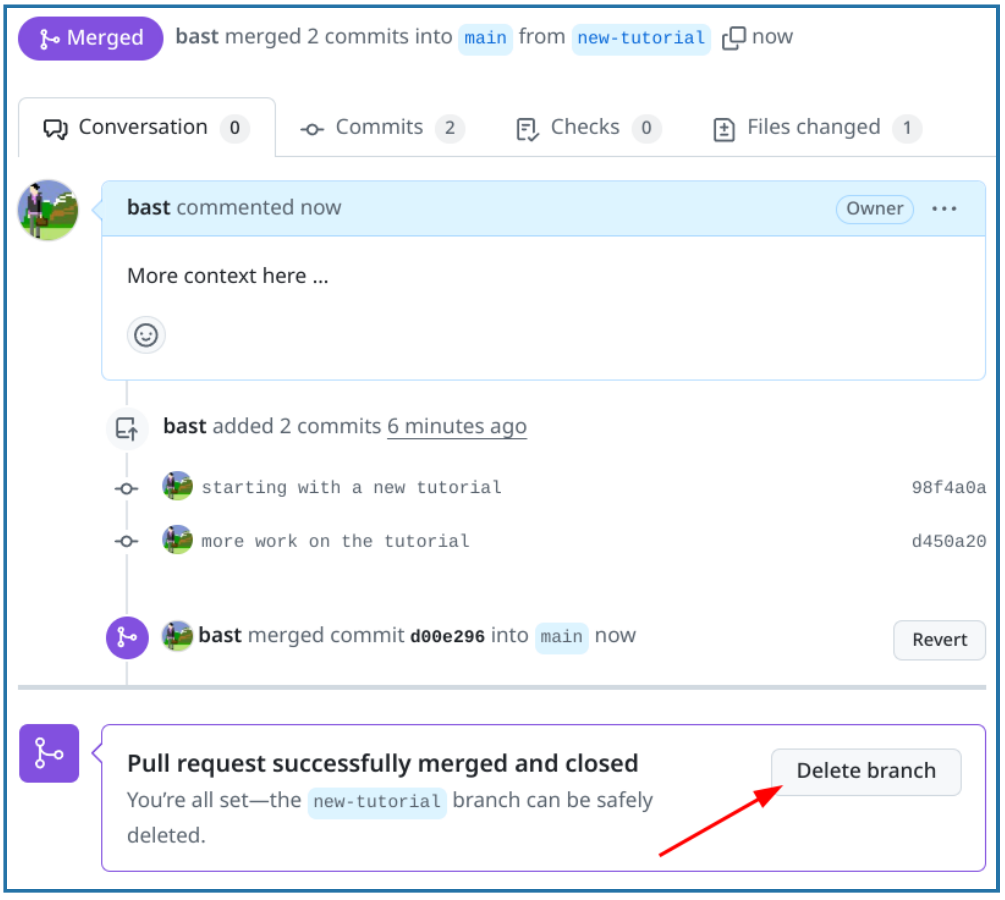
(6) Delete merged branches

Before deleting branches, first check whether they are merged.

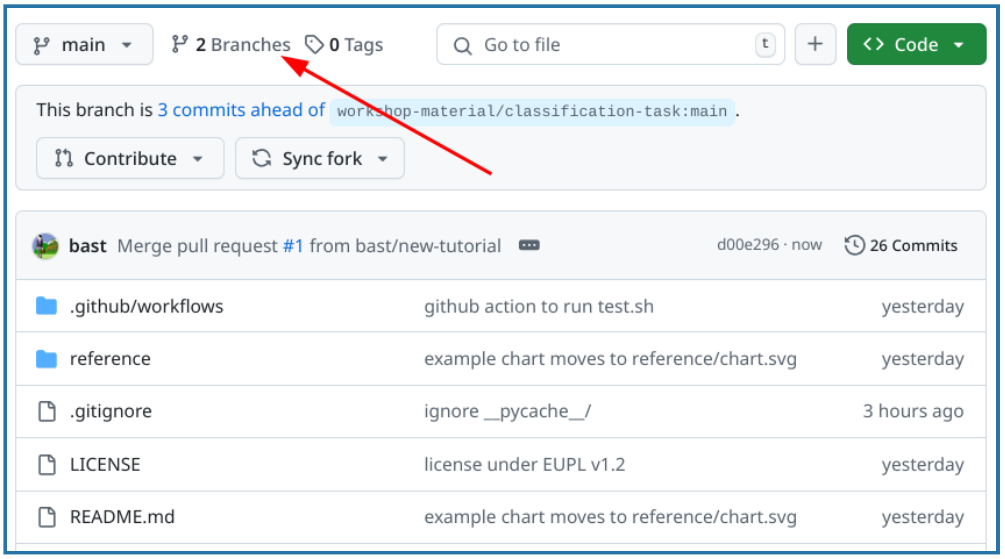
If you delete an un-merged branch, it will be difficult to find the commits that were on that branch. If you delete a merged branch, the commits are now also part of the branch where we have merged to.

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:



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



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”
- Just for fun, a page collecting 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

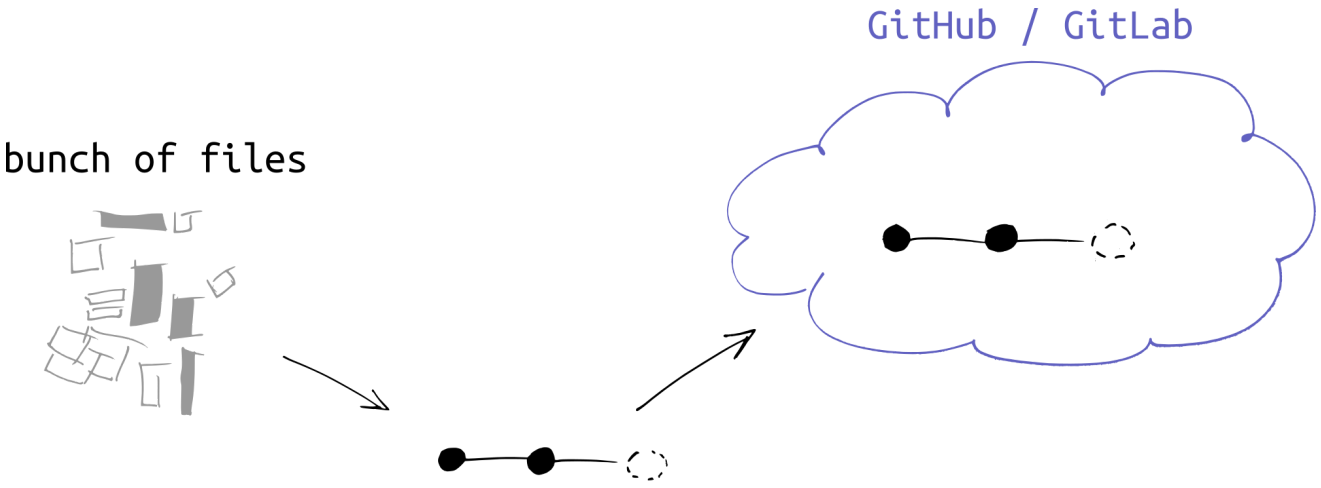
- Committing **generated files/directories** (example: `__pycache__`, `*.pyc`) -> use .gitignore files (collection of .gitignore templates).
- Committing **huge files** -> use code review to detect this.
- Committing **unrelated changes** together.
- Postponing commits because the changes are "unfinished"/"ugly" -> 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).

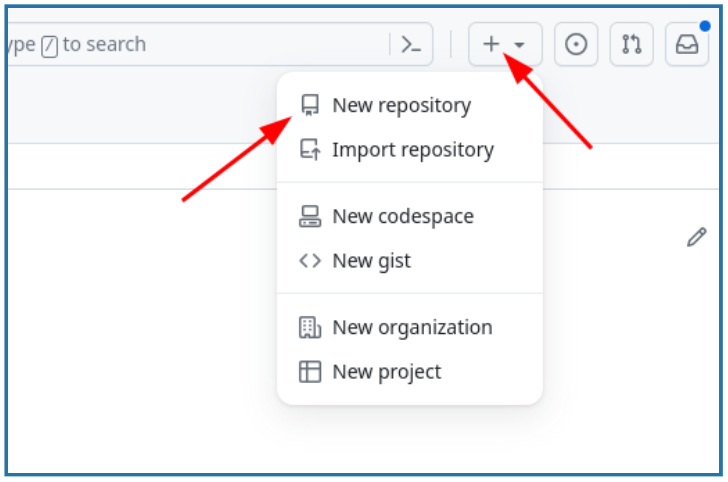
We offer **three different paths** of how to do this exercise.

- 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

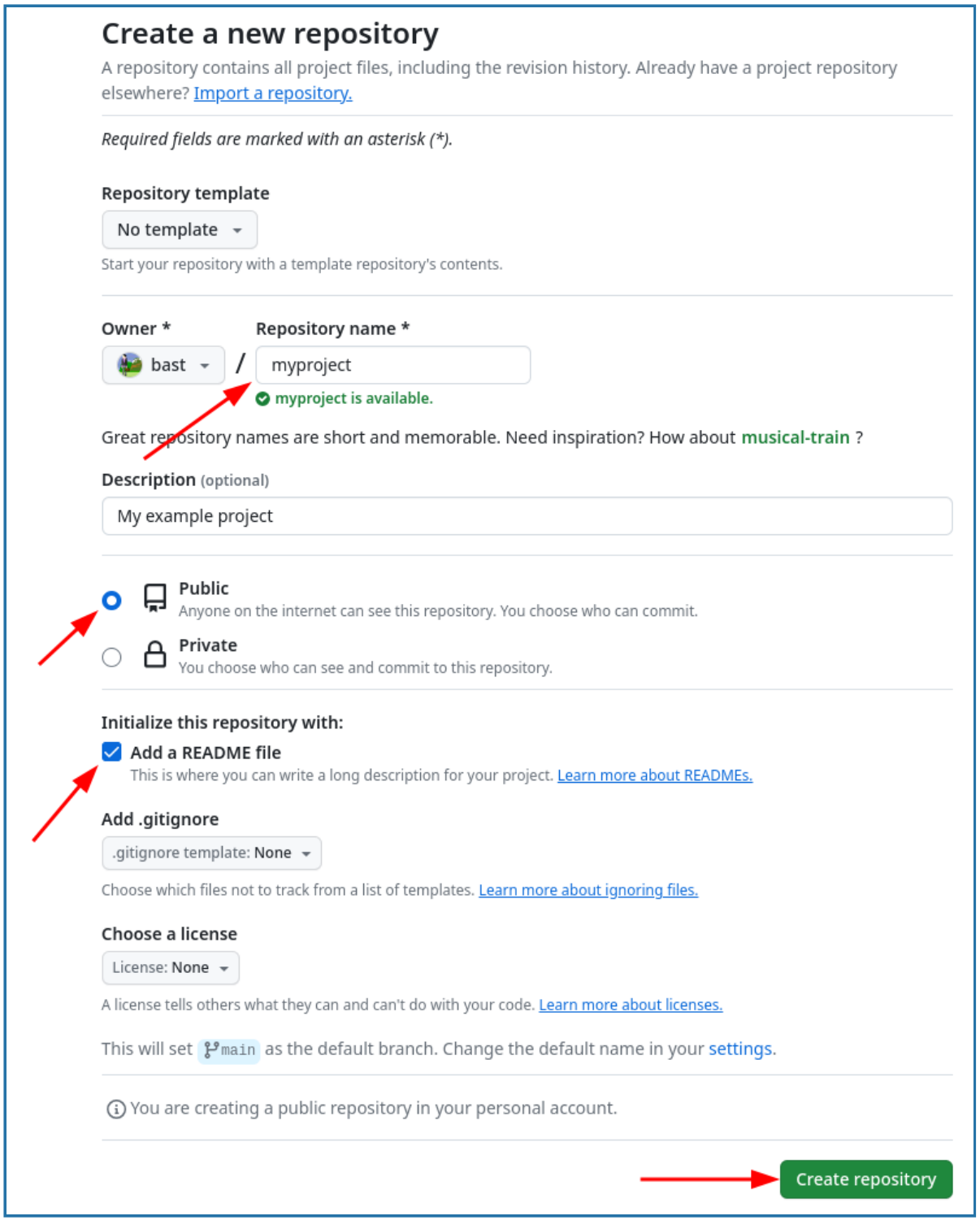
Create an repository on GitHub

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



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

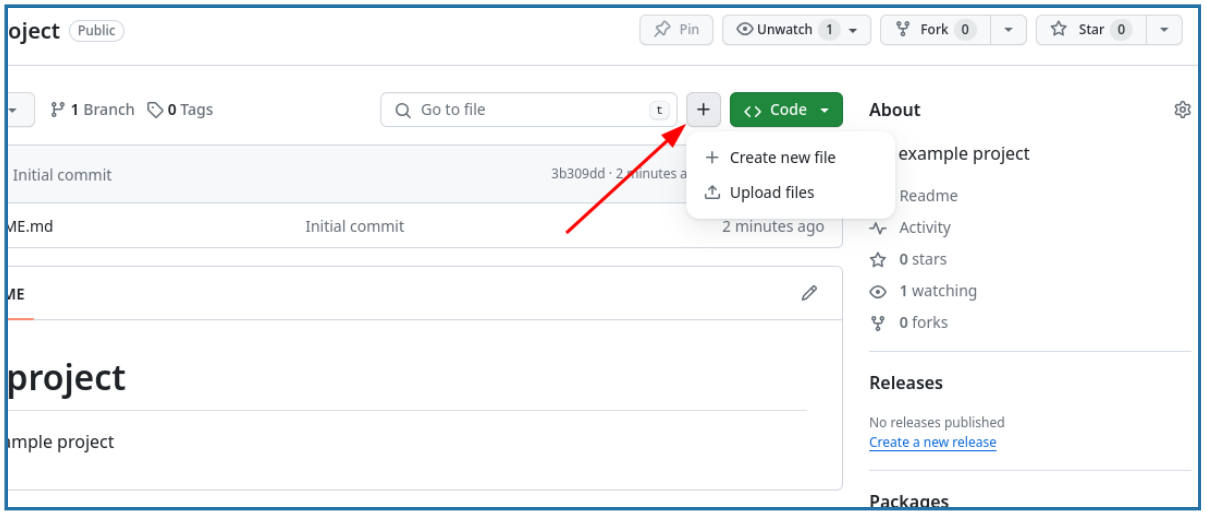
Then:



*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.

More about this in How to publish your code.

Where to start with 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 Software install instructions:

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

Which repository to use? You have 3 options:

- Clone **your fork** of the example repository.
- If you don't have that, you can clone the exercise repository itself.
- 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:

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

This is `conf.py`:

```
project = "your-project"
copyright = "2025, 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):

```
# 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):

```
# About this code

Work in progress ...
```

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

```
$ 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: classification-task
channels:
  - conda-forge
dependencies:
  - python <= 3.12
  - click
  - numpy
  - pandas
  - scipy
  - altair
  - vl-convert-python
  - 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-24.04

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

    - uses: mamba-org/setup-micromamba@v1
      with:
        micromamba-version: '2.0.5-0' # 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):

```
project = "your-project"
copyright = "2025, 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”.

Possibilities to host Sphinx documentation

- Build with GitHub Actions and deploy to GitHub Pages.
- Build with GitLab CI/CD and deploy to GitLab Pages.
- Build with Read the Docs and host there.

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.

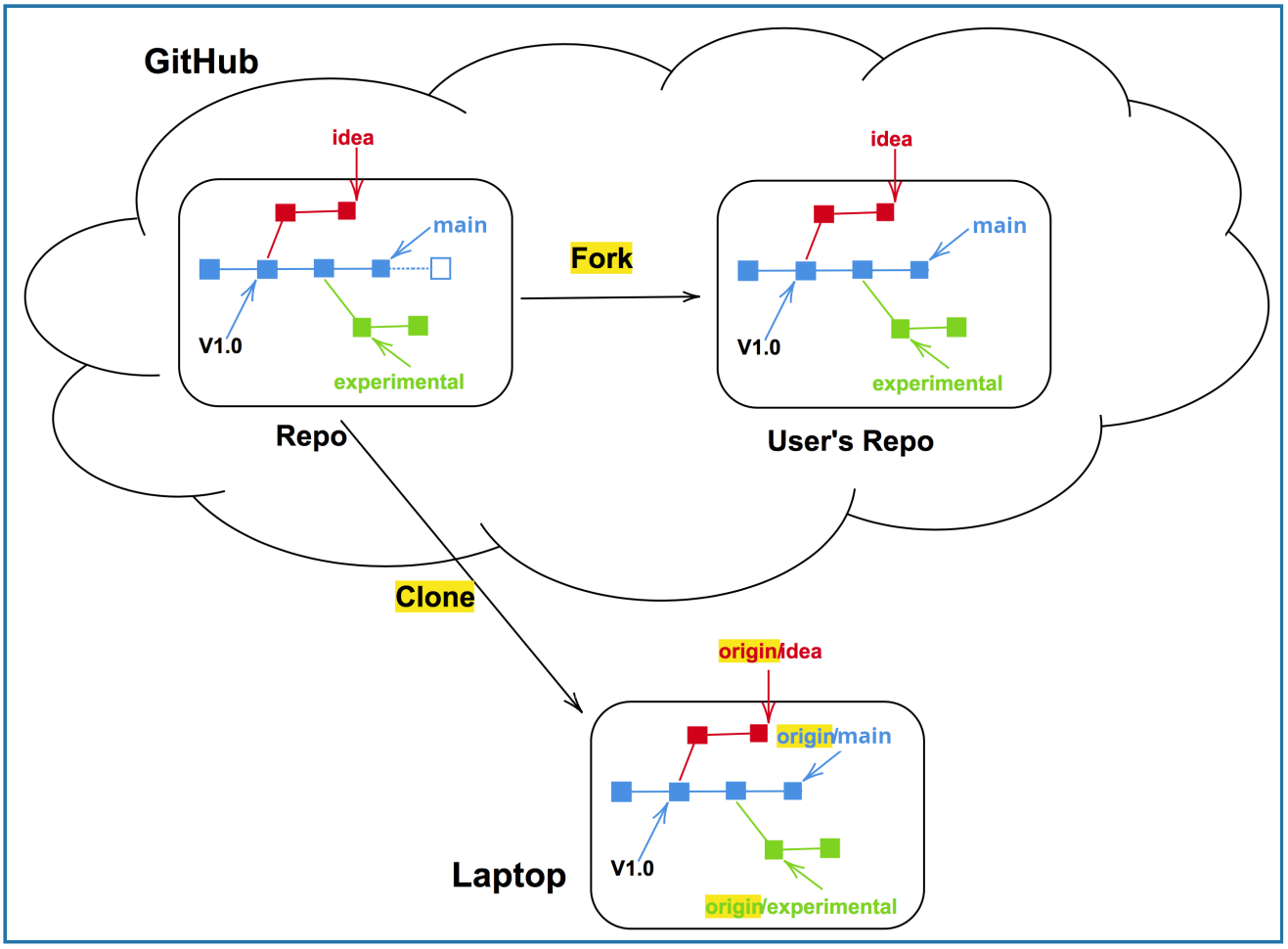
Commits, branches, repositories, forks, clones

- **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

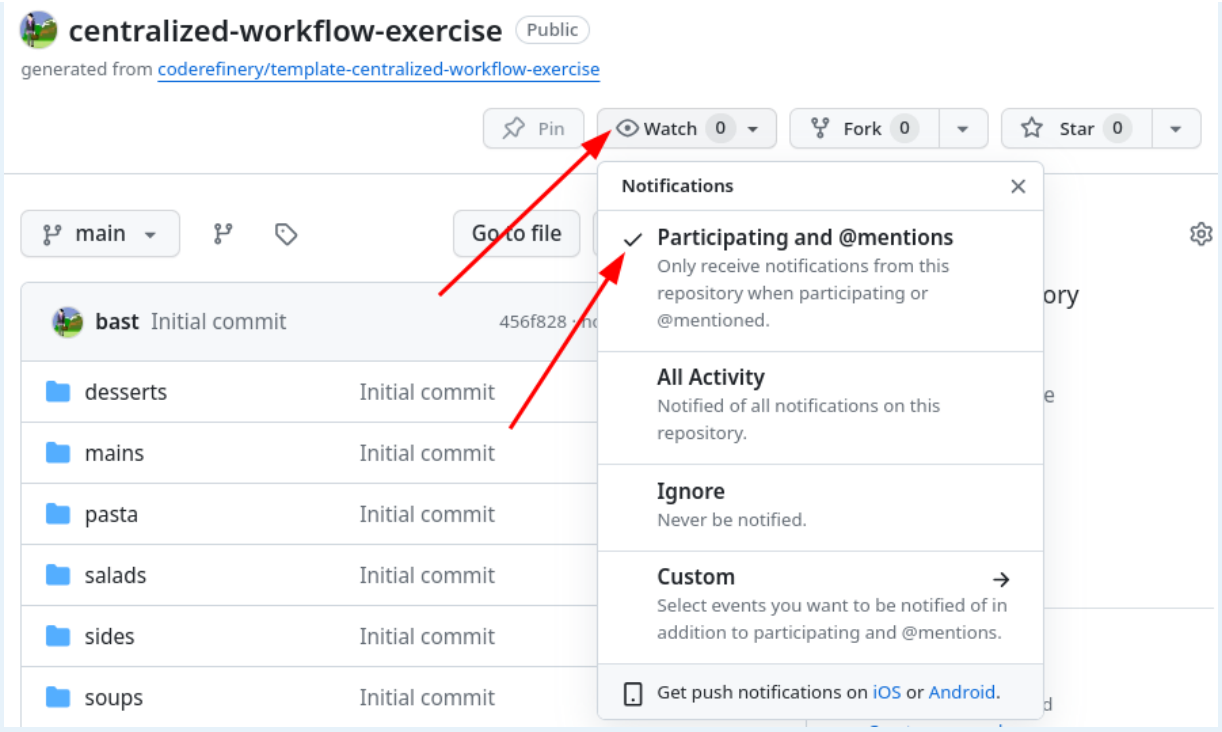
When you read or hear **pull request**, please think of a **change proposal**.

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.



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.

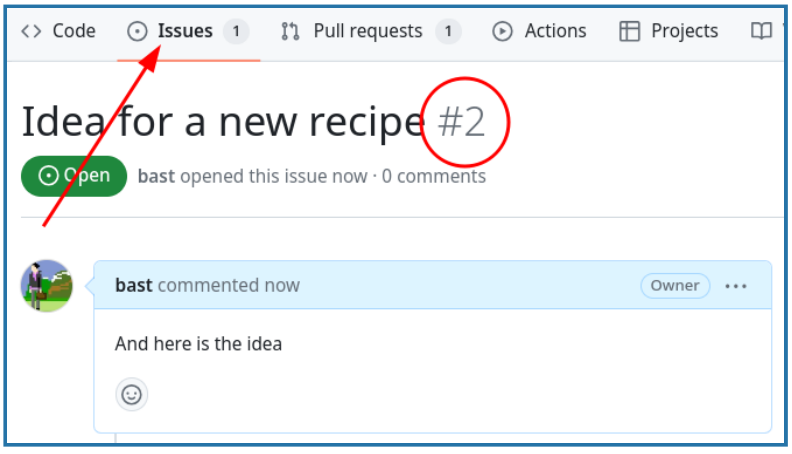
(3) Make a change adding the recipe

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`):



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.

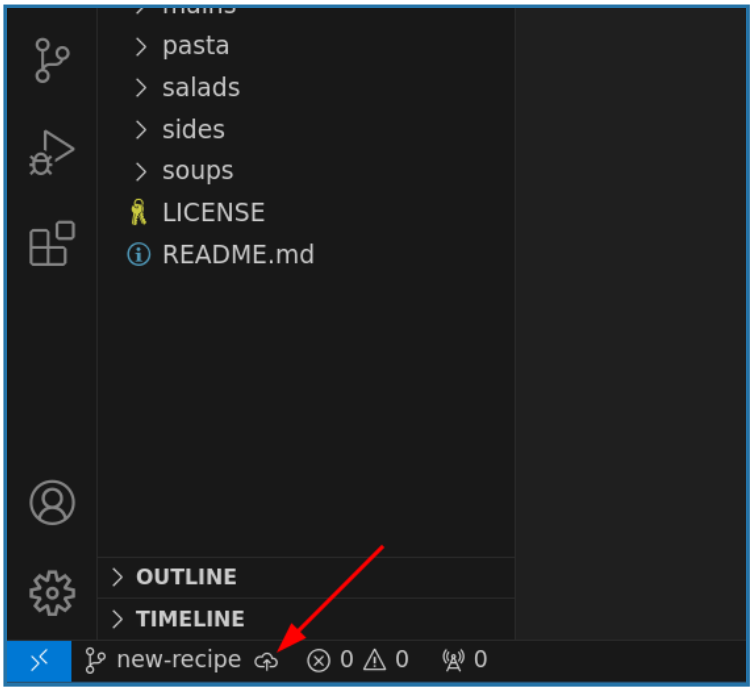
(4) Push to GitHub as a new branch

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.

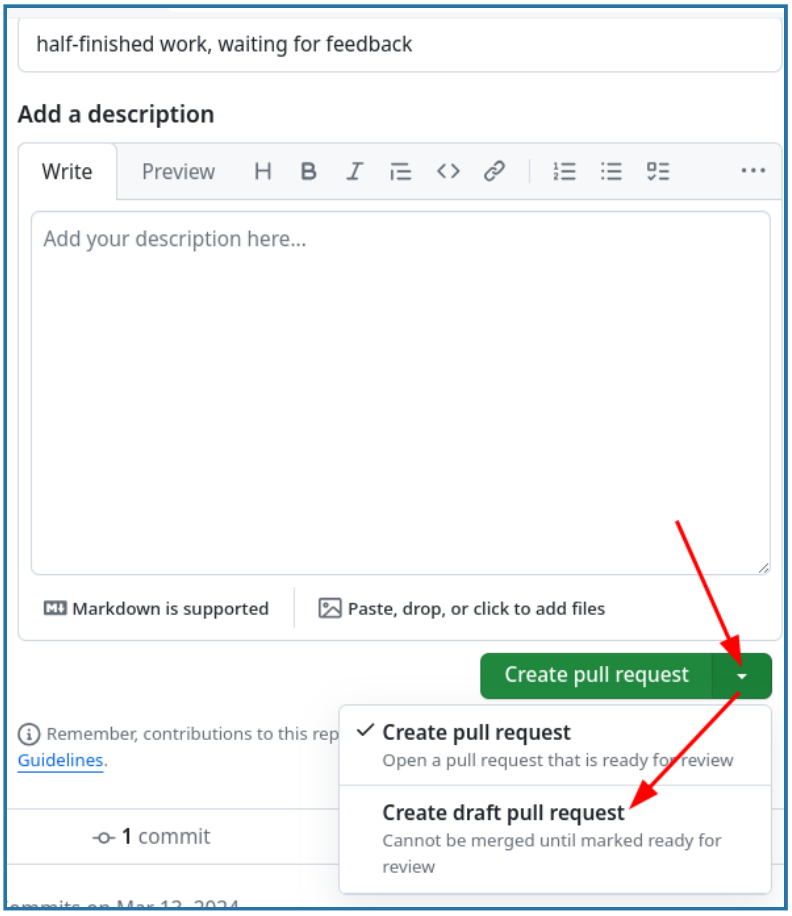
Checklist for reviewing a pull request:

- 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.

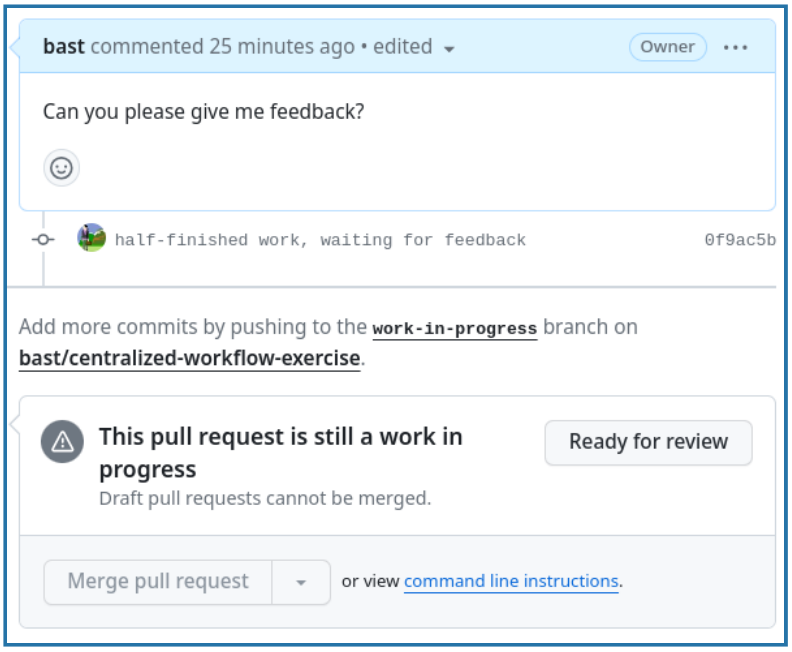
If someone is new, it's often nice to say something encouraging in the comments before merging (even if it's just "thanks"). If all is good and there's not much else to say, you could merge directly.

(7) Draft pull requests

Try to create a draft pull request:



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



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.

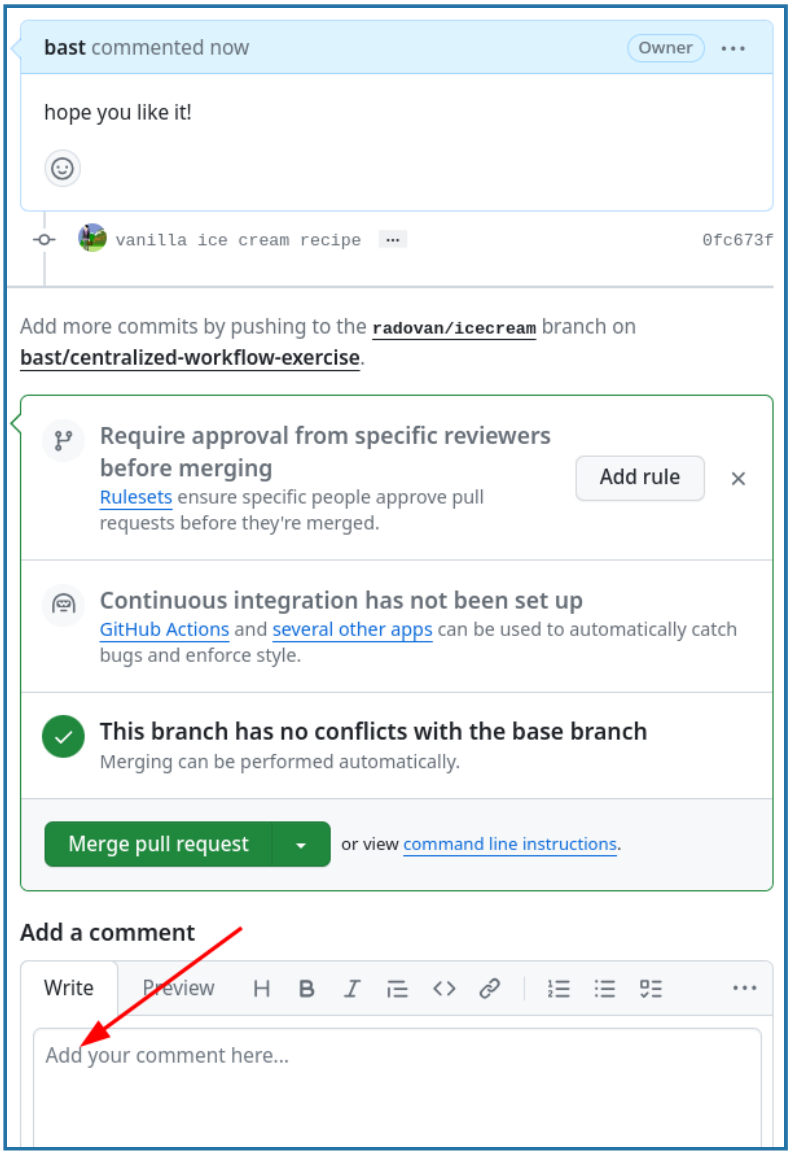
Help and discussion

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.

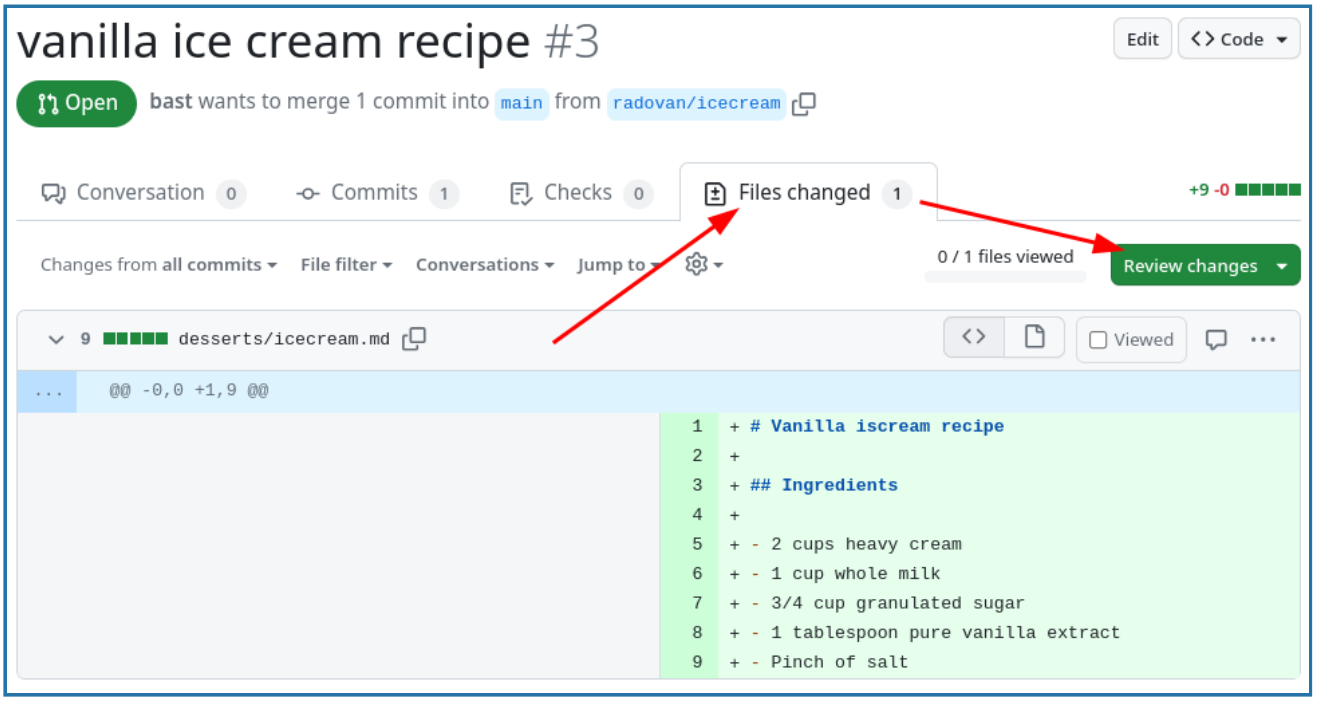
How to ask for changes in a pull request

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:



Or by using the “Review changes”:

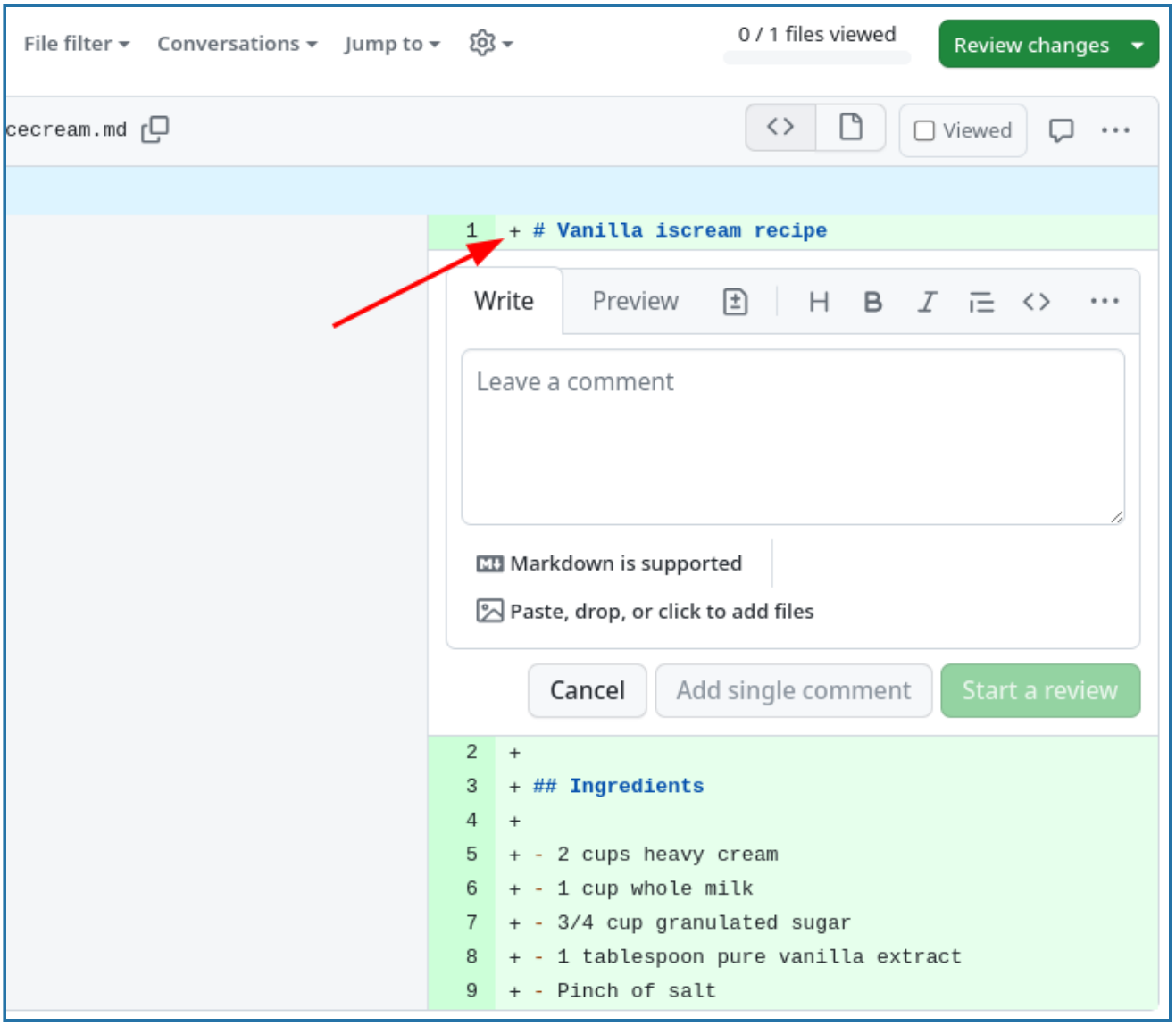


And always please be kind and constructive in your comments. Remember that the goal is not gate-keeping but **collaborative learning**.

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

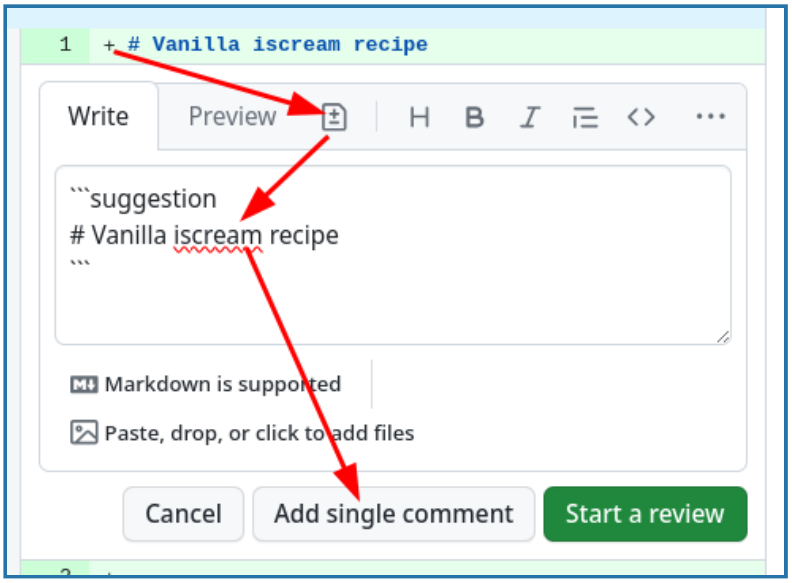
If you see a very small problem that is easy to fix, you can suggest a change as a reviewer.

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:

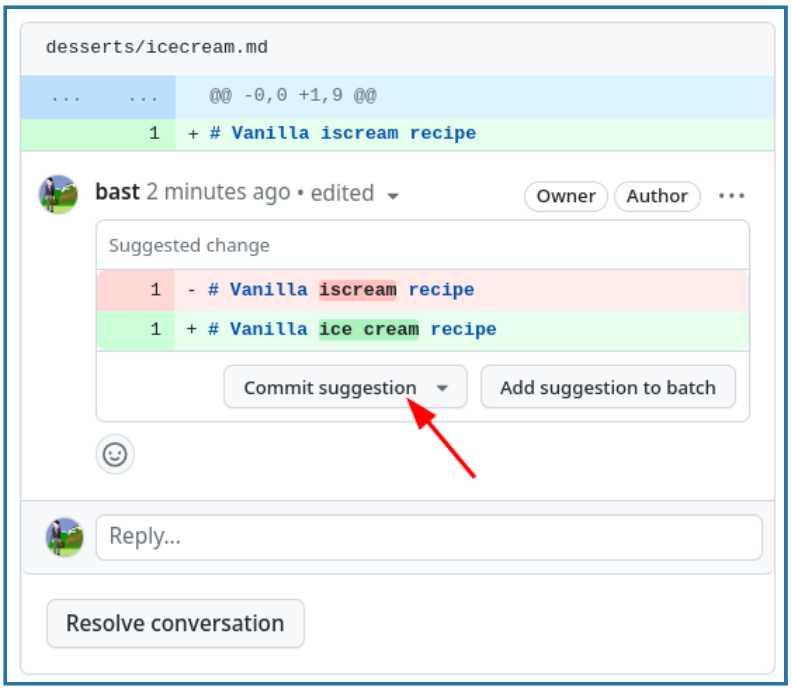


Here you can comment on specific lines or even line ranges.

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:



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



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

How to modify a pull request to address the review comments

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

A much better mechanism 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:

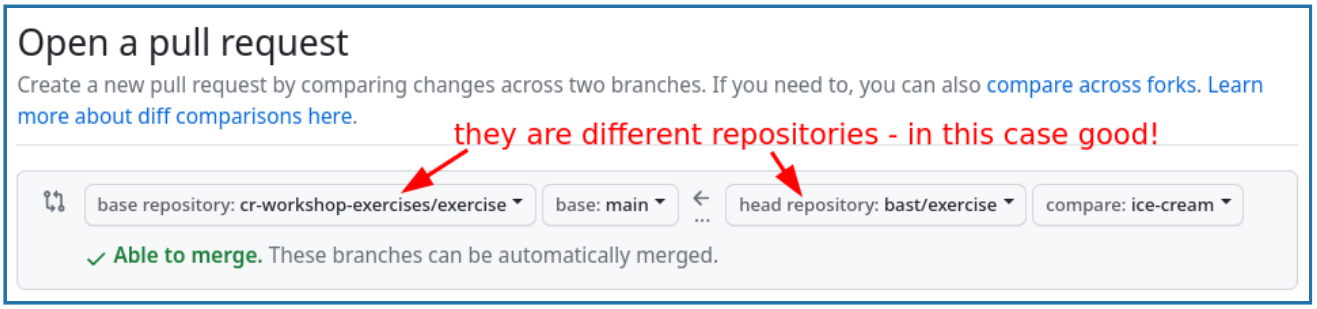
```
$ git remote set-url origin git@github.com:USER/recipe-book-forking-exercise.git
```

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:

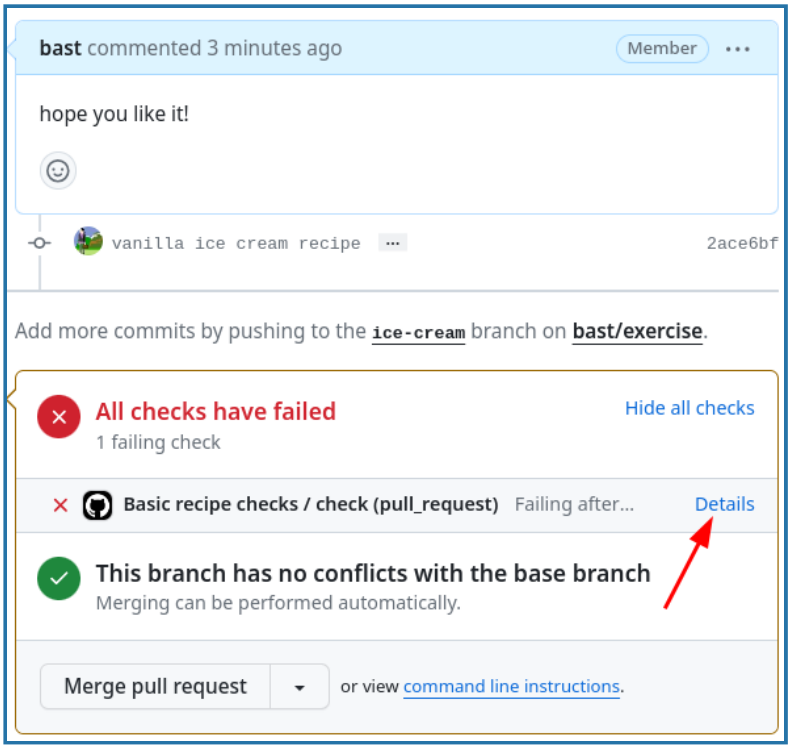


Pull requests can be coupled with automated testing

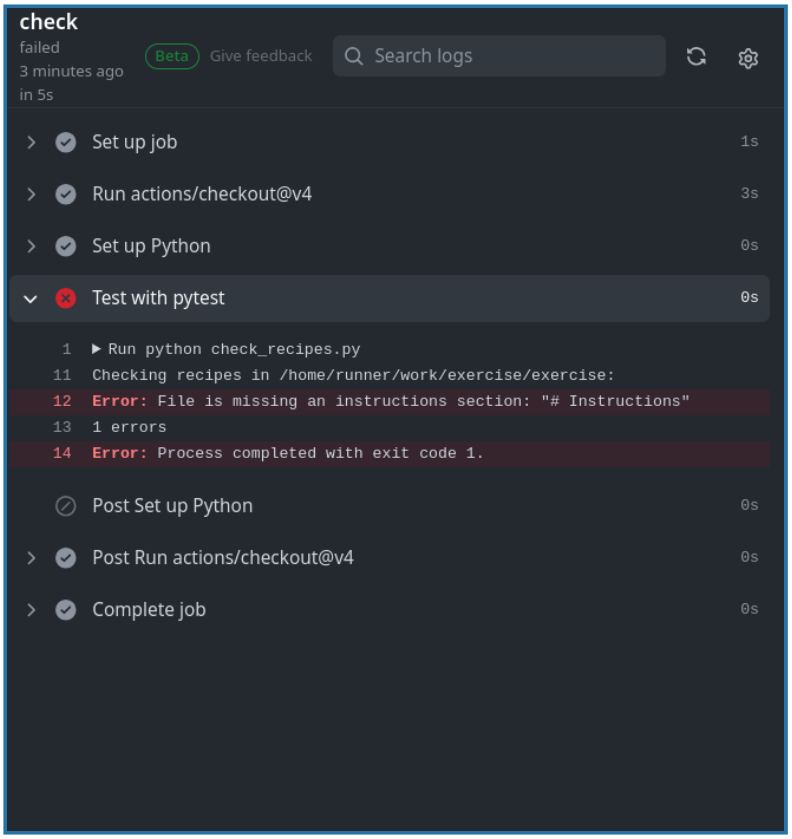
We added an automated test here just for fun and so that you see that this is possible to do.

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

In this example the test failed:



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



How can this be useful?

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

How does it work?

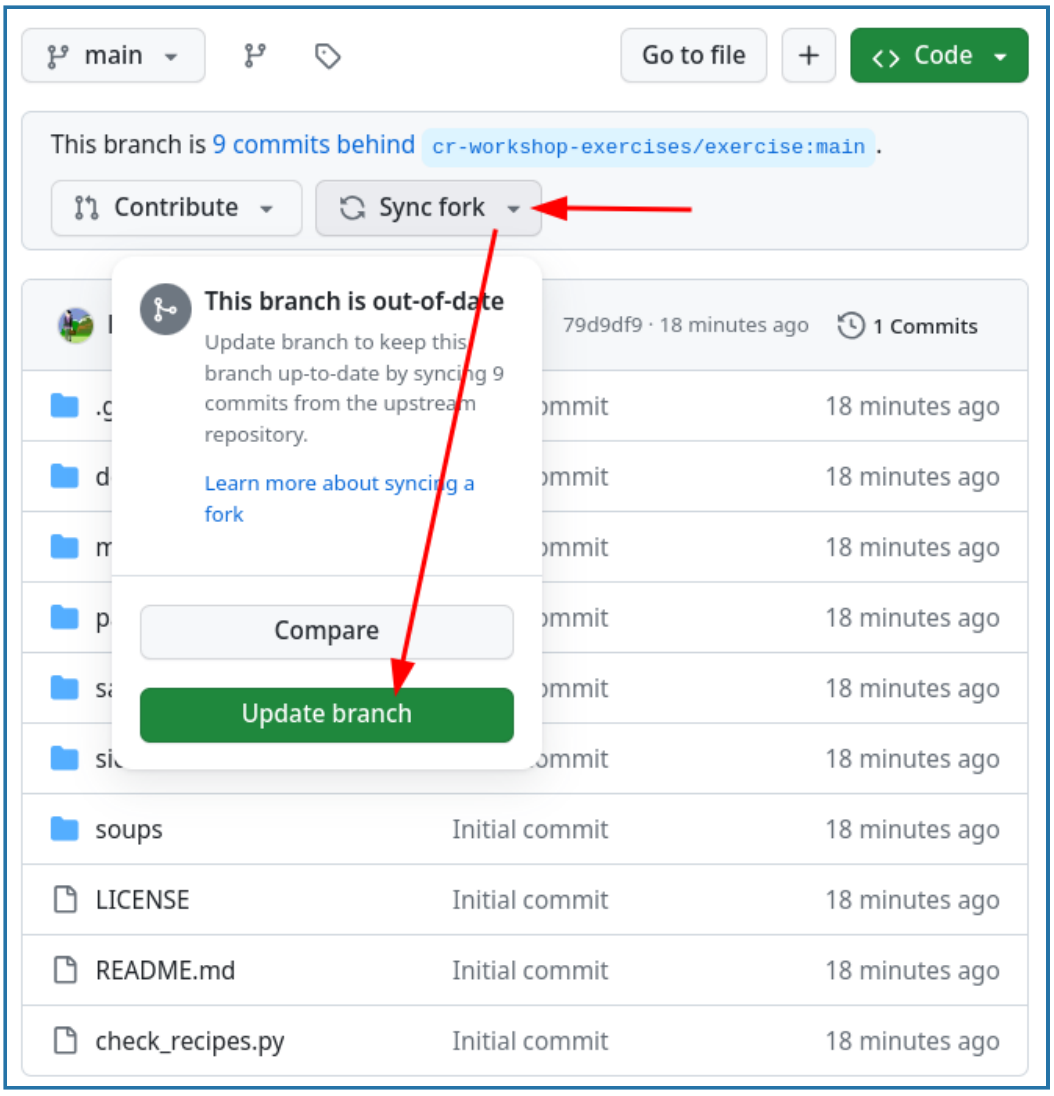
- We added a GitHub Actions workflow to automatically run on each push or pull request towards the `main` branch.

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

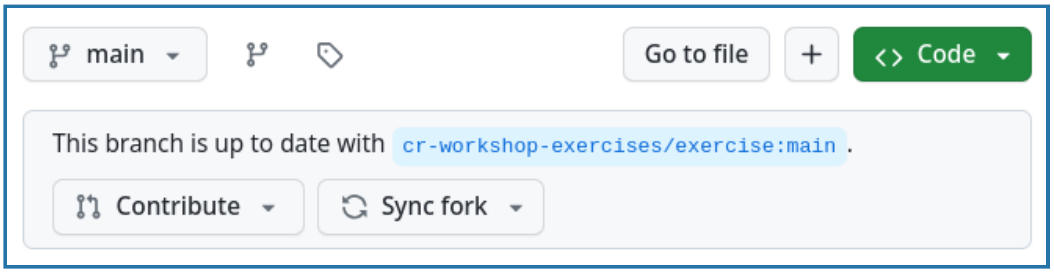
How to update your fork with changes from upstream

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

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



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



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

Contributing very minor changes

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

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

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

Motivation

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

If you have an idea for a new feature

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

Motivation

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

Summary

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

Reproducible environments and dependencies

Objectives

- There are not many codes that have no dependencies. How should we **deal with dependencies**?
- We will focus on installing and managing dependencies in Python when using packages from PyPI and Conda.
- We will not discuss how to distribute your code as a package.

[This episode borrows from https://coderefinery.github.io/reproducible-python/reusable/ and https://aaltoscicomp.github.io/python-for-scicomp/dependencies/]

Essential XKCD comics:

- xkcd - dependency
- xkcd - superfund

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: ...

Two ecosystems: PyPI (The Python Package Index) and Conda

PyPI

- **Installation tool:** `pip` or `uv` or similar
- Traditionally used for Python-only packages or for Python interfaces to external libraries. There are also packages that have bundled external libraries (such as numpy).
- **Pros:**
  - Easy to use
  - Package creation is easy
- **Cons:**
  - Installing packages that need external libraries can be complicated

Conda

- **Installation tool:** `conda` or `mamba` or similar
- Aims to be a more general package distribution tool and it tries to provide not only the Python packages, but also libraries and tools needed by the Python packages.
- **Pros:**
  - Quite easy to use
  - Easier to manage packages that need external libraries
  - Not only for Python
- **Cons:**
  - Package creation is harder

Conda ecosystem explained

- Anaconda is a distribution of conda packages made by Anaconda Inc. When using Anaconda remember to check that your situation abides with their licensing terms (see below).
- Anaconda has recently changed its **licensing terms**, which affects its use in a professional setting. This caused uproar among academia and Anaconda modified their position in this article.

  Main points of the article are:

  - conda (installation tool) and community channels (e.g. conda-forge) are free to use.
  - Anaconda repository and **Anaconda's channels in the community repository** are free for universities and companies with fewer than 200 employees. Non-university research institutions and national laboratories need licenses.
  - Miniconda is free, when it does not download Anaconda's packages.
  - Miniforge is not related to Anaconda, so it is free.

  For ease of use on sharing environment files, we recommend using Miniforge to create the environments and using conda-forge as the main channel that provides software.

- Major repositories/channels:

  - Anaconda Repository houses Anaconda's own proprietary software channels.
  - Anaconda's proprietary channels: `main`, `r`, `msys2` and `anaconda`. These are sometimes called `defaults`.
  - conda-forge is the largest open source community channel. It has over 28k packages that include open-source versions of packages in Anaconda's channels.

Tools and distributions for dependency management in Python

- Poetry: Dependency management and packaging.
- Pipenv: Dependency management, alternative to Poetry.
- pyenv: If you need different Python versions for different projects.
- virtualenv: Tool to create isolated Python environments for PyPI packages.
- micropipenv: Lightweight tool to "rule them all".
- Conda: Package manager for Python and other languages maintained by Anaconda Inc.

- Miniconda: A “miniature” version of conda, maintained by Anaconda Inc. By default uses Anaconda’s channels. Check licensing terms when using these packages.
- Mamba: A drop in replacement for conda. It used be much faster than conda due to better dependency solver but nowadays conda also uses the same solver. It still has some UI improvements.
- Micromamba: Tiny version of the Mamba package manager.
- Miniforge: Open-source Miniconda alternative with conda-forge as the default channel and optionally mamba as the default installer.
- Pixi: Modern, super fast tool which can manage conda environments.
- uv: Modern, super fast replacement for pip, poetry, pyenv, and virtualenv. You can also switch between Python versions.

Best practice: Install dependencies into isolated environments

- For each project, create a **separate environment**.
- Don’t install dependencies globally for all projects. Sooner or later, different projects will have conflicting dependencies.
- Install them **from a file** which documents them at the same time Install dependencies by first recording them in `requirements.txt` or `environment.yml` and install using these files, then you have a trace (we will practice this later below).

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)
- **pyproject.toml** (for uv, poetry and similar)

You can export (“freeze”) 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
```

How to communicate the dependencies as part of a report/thesis/publication

Each notebook or script or project which depends on libraries should come with either a `requirements.txt` or a `environment.yml`, unless you are creating and distributing this project as Python package.

- Attach a `requirements.txt` or a `environment.yml` to your thesis.
- Even better: Put `requirements.txt` or a `environment.yml` in your Git repository along your code.
- Even better: Also binderize your analysis pipeline.

Containers

- A container is like an **operating system inside a file**.
- "Building a container": Container definition file (recipe) -> Container image
- This can be used with Apptainer/ SingularityCE.

Containers offer the following advantages:

- **Reproducibility**: The same software environment can be recreated (most of the time) on different computers. They force you to know and **document all your dependencies**.
- **Portability**: The same software environment can be run (most of the time) 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.

How to install dependencies into environments

Now we understand a bit better why and how we installed dependencies for this course in the Software install instructions.

We have used **Miniforge** and the long command we have used was:

```
$ mamba env create -n course -f
https://raw.githubusercontent.com/coderefinery/reproducible-python-
ml/main/software/environment.yml
```

This command did two things:

- Create a new environment with name "course" (specified by `-n`).
- Installed all dependencies listed in the `environment.yml` file (specified by `-f`), which we fetched directly from the web. Here you can browse it.

For your own projects:

1. Start by writing an `environment.yml` of `requirements.txt` file. They look like this:

environment.yml | requirements.txt

```yaml
name: course
channels:
  - conda-forge
  - bioconda
dependencies:
  - python <= 3.12
  - click
  - numpy
  - pandas
  - scipy
  - altair
  - vl-convert-python
  - jupyterlab
  - pytest
  - scalene
  - flit
  - ruff
  - icecream
  - snakemake-minimal
  - myst-parser
  - sphinx
  - sphinx-rtd-theme
  - sphinx-autoapi
  - sphinx-autobuild
  - black
  - isort
  - pip
  - pip:
    - jupyterlab-code-formatter
```

2. Then set up an isolated environment and install the dependencies from the file into it:

Miniforge | Pixi | Virtual environment | uv

- Create a new environment with name “myenv” from `environment.yml`:

```
$ conda env create -n myenv -f environment.yml
```

Or equivalently:

```
$ mamba env create -n myenv -f environment.yml
```

- Activate the environment:

```
$ conda activate myenv
```

- Run your code inside the activated virtual environment.

```
$ python example.py
```

Updating environments

What if you forgot a dependency? Or during the development of your project you realize that you need a new dependency? Or you don't need some dependency anymore?

1. Modify the `environment.yml` or `requirements.txt` file.
2. Either remove your environment and create a new one, or update the existing one:

Miniforge | Pixi | Virtual environment | uv

- Update the environment by running:

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

- Or equivalently:

```
$ mamba env update --file environment.yml
```

Pinning package versions

Let us look at the environment.yml which we used to set up the environment for this course. Dependencies are listed without version numbers. Should we **pin the versions**?

- Both `pip` and `conda` ecosystems and all the tools that we have mentioned support pinning versions.
- It is possible to define a range of versions instead of precise versions.
- While your project is still in progress, I often use latest versions and do not pin them.
- When publishing the script or notebook, it is a good idea to pin the versions to ensure that the code can be run in the future.

- Remember that at some point in time you will face a situation where newer versions of the dependencies are no longer compatible with your software. At this point you'll have to update your software to use the newer versions or to lock it into a place in time.

Managing dependencies on a supercomputer

- Additional challenges:
  - Storage quotas: **Do not install dependencies in your home directory**. A conda environment can easily contain 100k files.
  - Network file systems struggle with many small files. Conda environments often contain many small files.
- Possible solutions:
  - Try Pixi (modern take on managing Conda environments) and uv (modern take on managing virtual environments). Blog post: Using Pixi and uv on a supercomputer
  - Install your environment on the fly into a scratch directory on local disk (**not** the network file system).
  - Install your environment on the fly into a RAM disk/drive.
  - Containerize your environment into a container image.

Keypoints

- Being able to communicate your dependencies is not only nice for others, but also for your future self or the next PhD student or post-doc.
- If you ask somebody to help you with your code, they will ask you for the dependencies.

Notebooks and version control

Objectives

- Demonstrate two tools which make version control of notebooks easier.

[this episode is adapted after https://coderefinery.github.io/jupyter/version-control/]

Jupyter Notebooks are stored in JSON format. With this format it can be a bit difficult to compare and merge changes which are introduced through the notebook interface.

Packages and JupyterLab extensions to simplify version control

Several packages and JupyterLab extensions have been developed to make it easier to interact with Git and GitHub:

- nbdime (notebook "diff" and "merge") provides "content-aware" diffing and merging.
  - Adds a Git button to the notebook interface.

    - `git diff` and `git merge` shell commands can use nbdime's diff and merge for notebook files, but leave Git's behavior unchanged for non-notebook files.
- jupyterlab-git is a JupyterLab extension for version control using Git.
    - Adds a Git tab to the left-side menu bar for version control inside JupyterLab.
- JupyterLab GitHub is a JupyterLab extension for accessing GitHub repositories.
    - Adds a GitHub tab to the left-side menu bar where you can browse and open notebooks from your GitHub repositories.

All three extensions can be used from within the JupyterLab interface and our Conda environment provides jupyterlab-git and nbdime. To install additional extensions, please consult the official documentation about installing and managing JupyterLab extensions.

Comparing Jupyter Notebooks on GitHub

For this you really want to enable Rich Jupyter Notebook Diffs on GitHub:

- On GitHub click on your avatar/image (top right).
- Click on "Feature preview".
- Enable "Rich Jupyter Notebook Diffs".

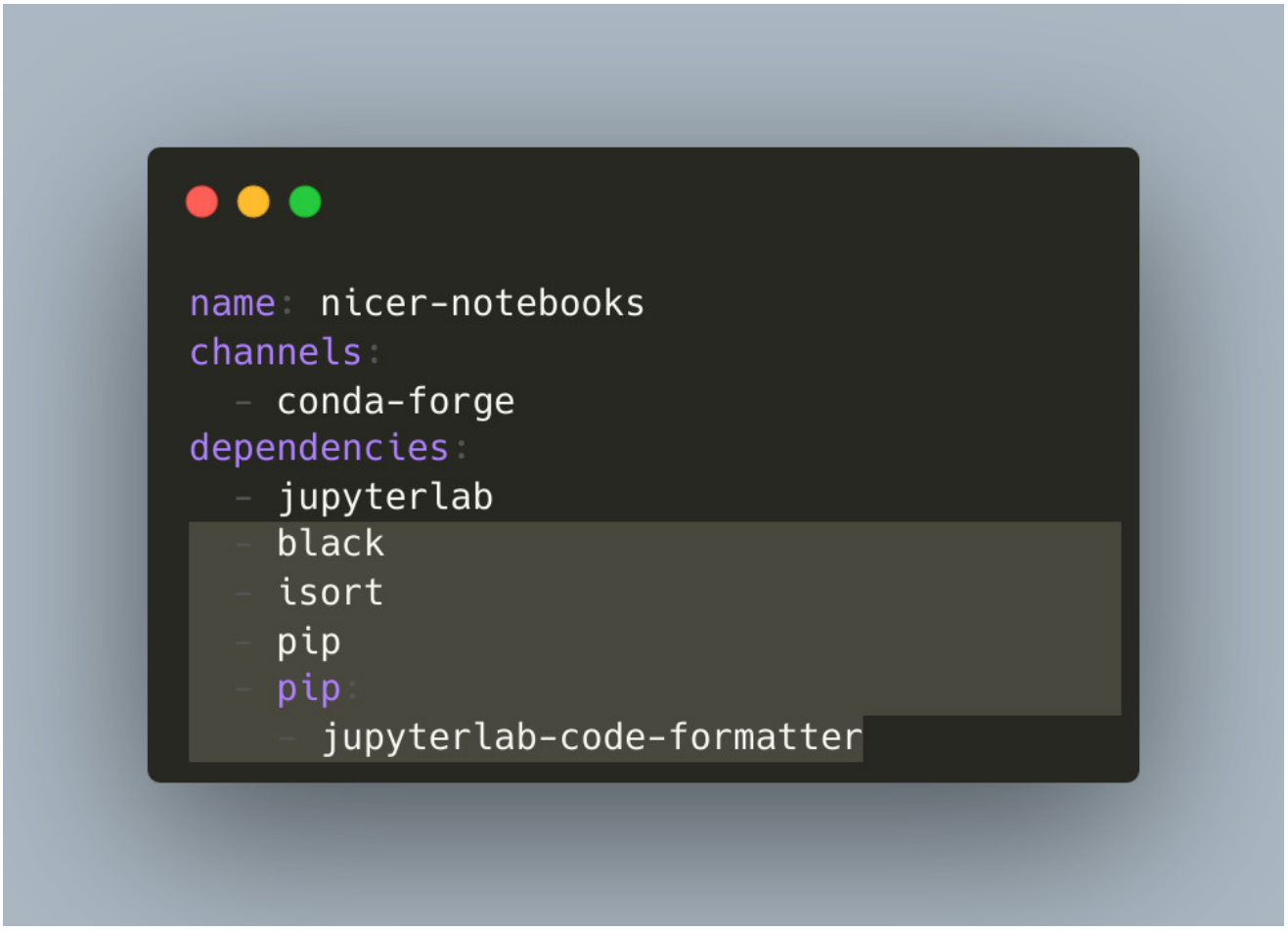
✍ Demonstration

- We can demonstrate this with a notebook that contains a Matplotlib plot (unfortunately the demonstration is less convincing with an Altair plot since the latter is generated on the fly and not stored as an image).
- We place the notebook in a GitHub repository and make a small change to it.
- We use https://github.com/**USER**/**REPO**/compare/**VERSION1**..**VERSION2** to compare the two versions of the notebook, once with "Rich Jupyter Notebook Diffs" enabled, and once without.

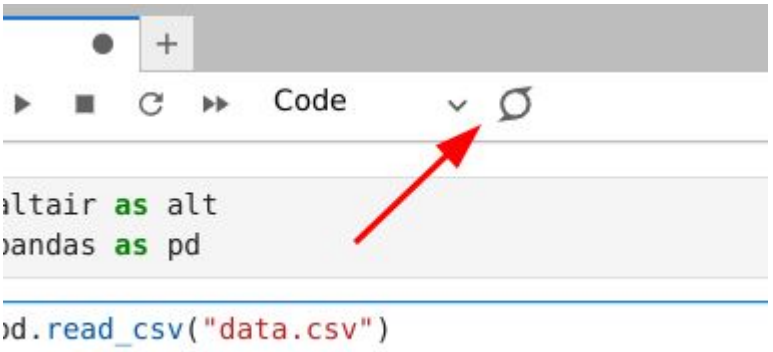
Other useful tooling for notebooks

Code formatting

https://jupyterlab-code-formatter.readthedocs.io/



We need three additional packages to format code in JupyterLab: `jupyterlab-code-formatter`, `black`, *and* `isort`.



This button will format the code in all cells of the notebook.

Instructor note

We test it out together on an example notebook.

Sharing notebooks

Objectives

- Know about good practices for notebooks to make them reusable
- Have a recipe to share a dynamic and reproducible visualization pipeline

[this lesson is adapted after https://coderefinery.github.io/jupyter/sharing/]

Document dependencies

- If you import libraries into your notebook, note down their versions. Document the dependencies as discussed in section Reproducible environments and dependencies.
- Place either `environment.yml` or `requirements.txt` in the same folder as the notebook(s).
- If you publish the notebook as part of a publication, it is probably a good idea to **pin the versions** of the libraries you use.
- This is not only useful for people who will try to rerun this in future, it is also understood by some tools (e.g. Binder) which we will see later.

Different ways to share a notebook

We need to learn how to share notebooks. At the minimum we need to share them with our future selves (backup and reproducibility).

- You can enter a URL, GitHub repo or username, or GIST ID in nbviewer and view a rendered Jupyter notebook
- Read the Docs can render Jupyter Notebooks via the nbsphinx package
- Binder creates live notebooks based on a GitHub repository
- EGI Notebooks (see also https://egi-notebooks.readthedocs.io)
- JupyterLab supports sharing and collaborative editing of notebooks via Google Drive. Recently it also added support for Shared editing with collaborative notebook model.
- JupyterLite creates a Jupyterlab environment in the browser and can be hosted as a GitHub page.
- Notedown, Jupinx and DocOnce can take Markdown or Sphinx files and generate Jupyter Notebooks
- Voilà allows you to convert a Jupyter Notebook into an interactive dashboard
- The `jupyter nbconvert` tool can convert a (`.ipynb`) notebook file to:
  - python code (`.py` file)
  - an HTML file
  - a LaTeX file
  - a PDF file
  - a slide-show in the browser

The following platforms can be used free of charge but have **paid subscriptions** for faster access to cloud resources:

- CoCalc (formerly SageMathCloud) allows collaborative editing of notebooks in the cloud
- Google Colab lets you work on notebooks in the cloud, and you can read and write to notebook files on Drive
- Microsoft Azure Notebooks also offers free notebooks in the cloud
- Deepnote allows real-time collaboration

Sharing dynamic notebooks on Binder

✍️ Exercise (20 min): Making your notebooks reproducible by anyone via Binder

- Create a new GitHub repository and **click on "Add a README file"**: https://github.com/new
- This exercise can be done entirely through the GitHub web interface (but using the terminal is of course also OK). You can use the "Add file" button to upload files.

Jupyter Notebooks | R Markdown/R Studio project

- Download this notebook (right-click, "Save as ...") and upload it to this repository. You can also try this with a different notebook.
- Add also a `requirements.txt` file which contains (adapt this if your notebook has other dependencies):

```
numpy==1.26.4
matplotlib==3.4.1
```

- Visit https://mybinder.org:

Screenshot of Binder web interface.

- Copy-paste the markdown text for the mybinder badge into a README.md file in your notebook repository.
- Check that your notebook repository now has a "launch binder" badge in your `README.md` file on GitHub.
- Try clicking the button and see how your repository is launched on Binder (can take a minute or two). Your notebooks can now be expored and executed in the cloud.

- Enjoy being fully reproducible!

More examples with Binder:

- Binder documentation
- Collection of example repositories

How to get a digital object identifier (DOI)

- Zenodo is a great service to get a DOI for a notebook (but **first practice** with the Zenodo sandbox).
- Binder can also run notebooks from Zenodo.
- In the supporting information of your paper you can refer to its DOI.

Tools and useful practices

Objectives

- How does good Python code look like? And if we only had 30 minutes, which good practices should we highlight?
- Some of the points are inspired by the excellent Effective Python book by Brett Slatkin.

Follow the PEP 8 style guide

- Please browse the PEP 8 style guide so that you are familiar with the most important rules.
- Using a consistent style makes your code easier to read and understand for others.
- You don't have to check and adjust your code manually. There are tools that can do this for you (see below).

Linting and static type checking

A **linter** is a tool that analyzes source code to detect potential errors, unused imports, unused variables, code style violations, and to improve readability.

- Popular linters:
  - Autoflake
  - Flake8
  - Pyflakes
  - Pycodestyle
  - Pylint
  - Ruff

We recommend Ruff since it can do **both checking and formatting** and you don't have to switch between multiple tools.

💬 Linters and formatters can be configured to your liking

These tools typically have good defaults. But if you don't like the defaults, you can configure what they should ignore or how they should format or not format.

This code example (which we possibly recognize from the previous section about Profiling) has few problems (highlighted):

```
import re
import requests


def count_unique_words(file_path: str) -> int:
    unique_words = set()
    forgotten_variable = 13
    with open(file_path, "r", encoding="utf-8") as file:
        for line in file:
            words = re.findall(r"\b\w+\b", line.lower())
            for word in words:
                unique_words.add(word)
    return len(unique_words)
```

Please try whether you can locate these problems using Ruff:

```
$ ruff check
```

If you use version control and like to have your code checked or formatted **before you commit the change**, you can use tools like pre-commit.

Many editors can be configured to automatically check your code as you type. Ruff can also be used as a **language server**.

Use an auto-formatter

Ruff is one of the best tools to automatically format your code according to a consistent style.

To demonstrate how it works, let us try to auto-format a code example which is badly formatted and also difficult to read:

Badly formatted | Auto-formatted

```
import re
def  count_unique_words (file_path : str)->int:
  unique_words=set()
  with open(file_path,"r",encoding="utf-8") as file:
    for line in file:
      words=re.findall(r"\b\w+\b",line.lower())
      for word in words:
        unique_words.add(word)
  return len(   unique_words   )
```

Other popular formatters:

- Black
- YAPF

Many editors can be configured to automatically format for you when you save the file.

It is possible to automatically format your code in Jupyter notebooks! For this to work you need the following three dependencies installed:

```
jupyterlab-code-formatter
black
isort
```

More information and a screen-cast of how this works can be found at https://jupyterlab-code-formatter.readthedocs.io/.

Consider annotating your functions with type hints

Compare these two versions of the same function and discuss how the type hints can help you and the Python interpreter to understand the function better:

Without type hints | With type hints

```
def count_unique_words(file_path):
    unique_words = set()
    with open(file_path, "r", encoding="utf-8") as file:
        for line in file:
            words = re.findall(r"\b\w+\b", line.lower())
            for word in words:
                unique_words.add(word)
    return len(unique_words)
```

A (static) type checker is a tool that checks whether the types of variables in your code match the types that you have specified. Popular tools:

- Mypy
- Pyright (Microsoft)
- Pyre (Meta)

Consider using AI-assisted coding

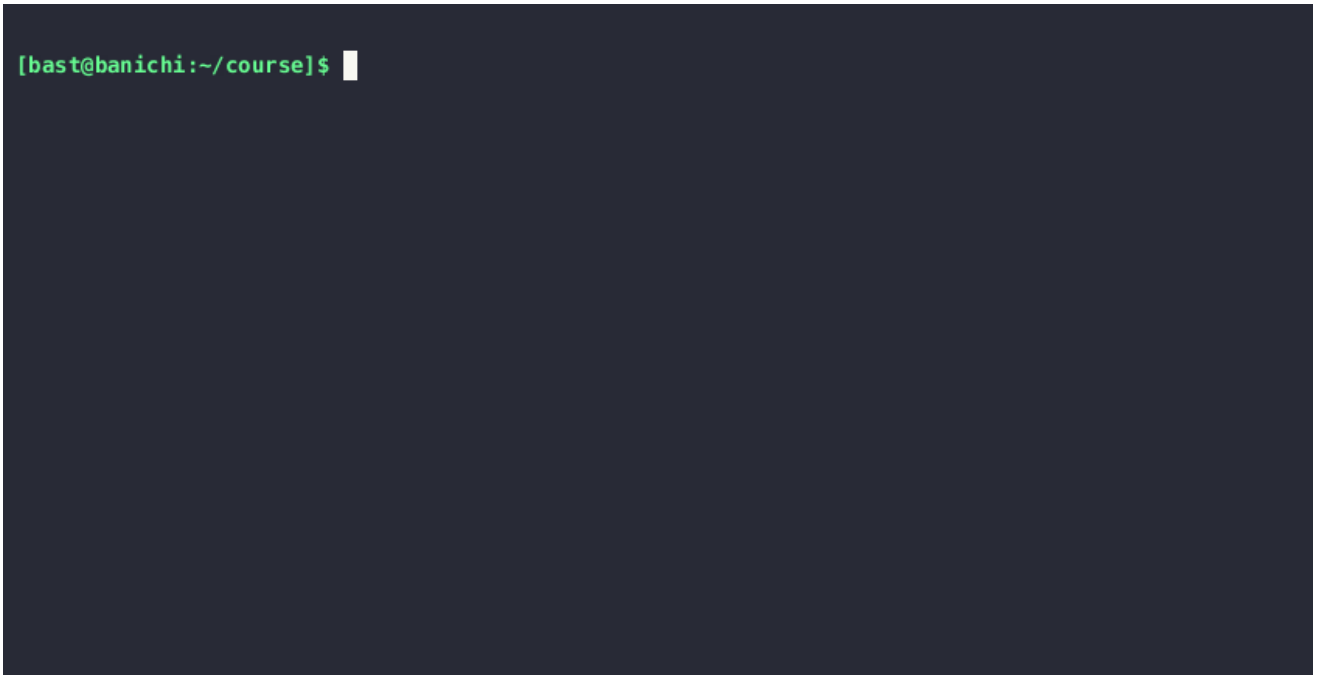
We can use AI as an assistant/apprentice:

- Code completion
- Write a test based on an implementation
- Write an implementation based on a test

Or we can use AI as a mentor:

- Explain a concept
- Improve code
- Show a different (possibly better) way of implementing the same thing

Example for using a chat-based AI tool.



Example for using AI to complete code in an editor.

AI tools open up a box of questions which are beyond our scope here

- Legal
- Ethical
- Privacy
- Lock-in/ monopolies
- Lack of diversity
- Will we still need to learn programming?
- How will it affect learning and teaching programming?

Debugging with print statements

Print-debugging is a simple, effective, and popular way to debug your code like this:

```
print(f"file_path: {file_path}")
```

Or more elaborate:

```
print(f"I am in function count_unique_words and the value of file_path is {file_path}")
```

But there can be better alternatives:

- Logging module

```
import logging

logging.basicConfig(level=logging.DEBUG)

logging.debug("This is a debug message")
logging.info("This is an info message")
```

- IceCream offers compact helper functions for print-debugging

```
from icecream import ic

ic(file_path)
```

Often you can avoid using indices

Especially people coming to Python from other languages tend to use indices where they are not needed. Indices can be error-prone (off-by-one errors and reading/writing past the end of the collection).

Iterating

Verbose and can be brittle | Better

```
scores = [13, 5, 2, 3, 4, 3]

for i in range(len(scores)):
    print(scores[i])
```

Enumerate if you need the index

Verbose and can be brittle | Better

```
particle_masses = [7.0, 2.2, 1.4, 8.1, 0.9]

for i in range(len(particle_masses)):
    print(f"Particle {i} has mass {particle_masses[i]}")
```

Zip if you need to iterate over two collections

Using an index can be brittle | Better

```
persons = ["Alice", "Bob", "Charlie", "David", "Eve"]
favorite_ice_creams = ["vanilla", "chocolate", "strawberry", "mint", "chocolate"]

for i in range(len(persons)):
    print(f"{persons[i]} likes {favorite_ice_creams[i]} ice cream")
```

Unpacking

Verbose and can be brittle | Better

```
coordinates = (0.1, 0.2, 0.3)

x = coordinates[0]
y = coordinates[1]
z = coordinates[2]
```

Prefer catch-all unpacking over indexing/slicing

Verbose and can be brittle | Better

```
scores = [13, 5, 2, 3, 4, 3]

sorted_scores = sorted(scores)

smallest = sorted_scores[0]
rest = sorted_scores[1:-1]
largest = sorted_scores[-1]

print(smallest, rest, largest)
# Output: 2 [3, 3, 4, 5] 13
```

List comprehensions, map, and filter instead of loops

For-loop | List comprehension | Map

```
string_numbers = ["1", "2", "3", "4", "5"]

integer_numbers = []
for element in string_numbers:
    integer_numbers.append(int(element))

print(integer_numbers)
# Output: [1, 2, 3, 4, 5]
```

For-loop | List comprehension | Filter

```
def is_even(number: int) -> bool:
    return number % 2 == 0


numbers = [1, 2, 3, 4, 5, 6]

even_numbers = []
for number in numbers:
    if is_even(number):
        even_numbers.append(number)

print(even_numbers)
# Output: [2, 4, 6]
```

Know your collections

How to choose the right collection type:

- Ordered and modifiable: `list`
- Fixed and (rather) immutable: `tuple`
- Key-value pairs: `dict`
- Dictionary with default values: `defaultdict` from `collections`
- Members are unique, no duplicates: `set`
- Optimized operations at both ends: `deque` from `collections`
- Cyclical iteration: `cycle` from `itertools`
- Adding/removing elements in the middle: Create a linked list (e.g. using a dictionary or a dataclass)
- Priority queue: `heapq` library
- Search in sorted collections: `bisect` library

What to avoid:

- Need to add/remove elements at the beginning or in the middle? Don't use a list.
- Need to make sure that elements are unique? Don't use a list.

Making functions more ergonomic

- Less error-prone API functions and fewer backwards-incompatible changes by enforcing keyword-only arguments:

```
def send_message(*, message: str, recipient: str) -> None:
    print(f"Sending to {recipient}: {message}")
```

- Use dataclasses or named tuples or dictionaries instead of too many input or output arguments.
- Docstrings instead of comments:

```
def send_message(*, message: str, recipient: str) -> None:
    """
    Sends a message to a recipient.

    Parameters:
    - message (str): The content of the message.
    - recipient (str): The name of the person receiving the message.
    """
    print(f"Sending to {recipient}: {message}")
```

- Consider using `DeprecationWarning` from the `warnings` module for deprecating functions or arguments.

Iterating

- When working with large lists or large data sets, consider using generators or iterators instead of lists. Discuss and compare these two:

```
even_numbers1 = [number for number in range(10000000) if number % 2 == 0]

even_numbers2 = (number for number in range(10000000) if number % 2 == 0)
```

- Beware of functions which iterate over the same collection multiple times. With generators, you can iterate only once.
- Know about `itertools` which provides a lot of functions for working with iterators.

Use relative paths and pathlib

- Scripts that read data from absolute paths are not portable and typically break when shared with a colleague or support help desk or reused by the next student/PhD student/postdoc.
- `pathlib` is a modern and portable way to handle paths in Python.

Project structure

- As your project grows from a simple script, you should consider organizing your code into modules and packages.
- Function too long? Consider splitting it into multiple functions.
- File too long? Consider splitting it into multiple files.
- Difficult to name a function or file? It might be doing too much or unrelated things.
- If your script can be imported into other scripts, wrap your main function in a `if __name__ == "__main__":` block:

```
def main():
    ...

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

- Why this construct? You can try to either import or run the following script:

```
if __name__ == "__main__":
    print("I am being run as a script")  # importing will not run this part
else:
    print("I am being imported")
```

- Try to have all code inside some function. This can make it easier to understand, test, and reuse. It can also help Python to free up memory when the function is done.

Reading and writing files

- Good construct to know to read a file:

```
with open("input.txt", "r") as file:
    for line in file:
        print(line)
```

- Reading a huge data file? Read and process it in chunks or buffered or use a library which does it for you.
- On supercomputers, avoid reading and writing thousands of small files.
- For input files, consider using standard formats like CSV, YAML, or TOML - then you don't need to write a parser.

Use subprocess instead of os.system

- Many things can go wrong when launching external processes from Python. The `subprocess` module is the recommended way to do this.
- `os.system` is not portable and not secure enough.

Parallelizing

- Use one of the many libraries: `multiprocessing`, `mpi4py`, Dask, Parsl, ...
- Identify independent tasks.
- More often than not, you can convert an expensive loop into a command-line tool and parallelize it using workflow management tools like Snakemake.

Profiling

Objectives

- Understand when improving code performance is worth the time and effort.
- Knowing how to find performance bottlenecks in Python code.
- Try Scalene as one of many tools to profile Python code.

[This page is adapted after https://aaltoscicomp.github.io/python-for-scicomp/profiling/]

Should we even optimize the code?

Classic quote to keep in mind: “Premature optimization is the root of all evil.” [Donald Knuth]

Discussion

It is important to ask ourselves whether it is worth it.

- Is it worth spending e.g. 2 days to make a program run 20% faster?
- Is it worth optimizing the code so that it spends 90% less memory?

Depends. What does it depend on?

Measure instead of guessing

Before doing code surgery to optimize the run time or lower the memory usage, we should **measure** where the bottlenecks are. This is called **profiling**.

Analogy: Medical doctors don’t start surgery based on guessing. They first measure (X-ray, MRI, ...) to know precisely where the problem is.

Not only programming beginners can otherwise guess wrong, but also experienced programmers can be surprised by the results of profiling.

One of the simplest tools is to insert timers

Below we will list some tools that can be used to profile Python code. But even without these tools you can find **time-consuming parts** of your code by inserting timers:

```
import time


# ...
# code before the function


start = time.time()
result = some_function()
print(f"some_function took {time.time() - start} seconds")


# code after the function
# ...
```

Many tools exist

The list below here is probably not complete, but it gives an overview of the different tools available for profiling Python code.

CPU profilers:

- cProfile and profile
- line_profiler
- py-spy
- Yappi
- pyinstrument
- Perfetto

Memory profilers:

- memory_profiler (not actively maintained)
- Pympler
- tracemalloc
- guppy/heapy

Both CPU and memory:

- Scalene

In the exercise below, we will use Scalene to profile a Python program. Scalene is a sampling profiler that can profile CPU, memory, and GPU usage of Python.

Tracing profilers vs. sampling profilers

Tracing profilers record every function call and event in the program, logging the exact sequence and duration of events.

- **Pros:**
  - Provides detailed information on the program's execution.
  - Deterministic: Captures exact call sequences and timings.
- **Cons:**
  - Higher overhead, slowing down the program.
  - Can generate larger amount of data.

Sampling profilers periodically samples the program's state (where it is and how much memory is used), providing a statistical view of where time is spent.

- **Pros:**
  - Lower overhead, as it doesn't track every event.
  - Scales better with larger programs.
- **Cons:**
  - Less precise, potentially missing infrequent or short calls.
  - Provides an approximation rather than exact timing.

💬 Analogy: Imagine we want to optimize the London Underground (subway) system

We wish to detect bottlenecks in the system to improve the service and for this we have asked few passengers to help us by tracking their journey.

- **Tracing**: We follow every train and passenger, recording every stop and delay. When passengers enter and exit the train, we record the exact time and location.
- **Sampling**: Every 5 minutes the phone notifies the passenger to note down their current location. We then use this information to estimate the most crowded stations and trains.

Choosing the right system size

Sometimes we can configure the system size (for instance the time step in a simulation or the number of time steps or the matrix dimensions) to make the program finish sooner.

For profiling, we should choose a system size that is **representative of the real-world** use case. If we profile a program with a small input size, we might not see the same bottlenecks as when running the program with a larger input size.

Often, when we scale up the system size, or scale the number of processors, new bottlenecks might appear which we didn't see before. This brings us back to: "measure instead of guessing".

Exercises

✍️ Exercise: Practicing profiling

In this exercise we will use the Scalene profiler to find out where most of the time is spent and most of the memory is used in a given code example.

Please try to go through the exercise in the following steps:

1. Make sure `scalene` is installed in your environment (if you have followed this course from the start and installed the recommended software environment, then it is).
2. Download Leo Tolstoy's "War and Peace" from the following link (the text is provided by Project Gutenberg): https://www.gutenberg.org/cache/epub/2600/pg2600.txt (right-click and "save as" to download the file and **save it as "book.txt"**).
3. **Before** you run the profiler, try to predict in which function the code (the example code is below) will spend most of the time and in which function it will use most of the memory.
4. Save the example code as `example.py` and run the `scalene` profiler on the following code example and browse the generated HTML report to find out where most of the time is spent and where most of the memory is used:

   ```
   $ scalene example.py
   ```

   Alternatively you can do this (and then open the generated file in a browser):

   ```
   $ scalene example.py --html > profile.html
   ```

   You can find an example of the generated HTML report in the solution below.

5. Does the result match your prediction? Can you explain the results?

Example code (`example.py`):

```python
"""
The code below reads a text file and counts the number of unique words in it
(case-insensitive).
"""
import re


def count_unique_words1(file_path: str) -> int:
    with open(file_path, "r", encoding="utf-8") as file:
        text = file.read()
    words = re.findall(r"\b\w+\b", text.lower())
    return len(set(words))


def count_unique_words2(file_path: str) -> int:
    unique_words = []
    with open(file_path, "r", encoding="utf-8") as file:
        for line in file:
            words = re.findall(r"\b\w+\b", line.lower())
            for word in words:
                if word not in unique_words:
                    unique_words.append(word)
    return len(unique_words)


def count_unique_words3(file_path: str) -> int:
    unique_words = set()
    with open(file_path, "r", encoding="utf-8") as file:
        for line in file:
            words = re.findall(r"\b\w+\b", line.lower())
            for word in words:
                unique_words.add(word)
    return len(unique_words)


def main():
    # book.txt is downloaded from
https://www.gutenberg.org/cache/epub/2600/pg2600.txt
    _result = count_unique_words1("book.txt")
    _result = count_unique_words2("book.txt")
    _result = count_unique_words3("book.txt")


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

✔ Solution

Result of the profiling run for the above code example. You can click on the image to make it larger.

Results:

- Most time is spent in the `count_unique_words2` function.
- Most memory is used in the `count_unique_words1` function.

Explanation:

- The `count_unique_words2` function is the slowest because it **uses a list** to store unique words and checks if a word is already in the list before adding it. Checking whether a list contains an element might require traversing the whole list, which is an O(n) operation. As the list grows in size, the lookup time increases with the size of the list.
- The `count_unique_words1` and `count_unique_words3` functions are faster because they **use a set** to store unique words. Checking whether a set contains an element is an O(1) operation.
- The `count_unique_words1` function uses the most memory because it **creates a list of all words** in the text file and then **creates a set** from that list.
- The `count_unique_words3` function uses less memory because it traverses the text file line by line instead of reading the whole file into memory.

What we can learn from this exercise:

- When processing large files, it can be good to read them line by line or in batches instead of reading the whole file into memory.
- It is good to get an overview over standard data structures and their advantages and disadvantages (e.g. adding an element to a list is fast but checking whether it already contains the element can be slow).

Additional resources

- Python performance workshop (by ENCCS)

Automated testing

Objectives

- Know **where to start** in your own project.
- Know what possibilities and techniques are available in the Python world.
- 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.

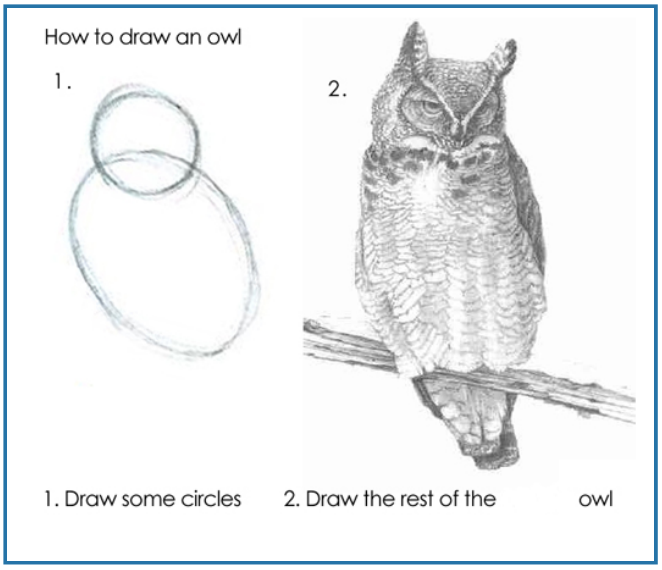
- **Less scary to change code**: tests will tell you whether something broke.
- **Easier for new people** to join.
- Easier for somebody to **revive an old code**.
- **End-to-end test**: run the whole code and compare result to a reference.
- **Unit tests**: test one unit (function or module). Can guide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

Do I even need testing?:

- 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 (GitHub Actions or GitLab CI).

If you want to start with unit-testing:

- You want to rewrite a function? Start adding a unit test right there first.
- You spend few days chasing a bug? Once you fix it, add a test to make sure it does not come back.

End-to-end tests

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

Pytest

Here is a simple example of a test:

```
def fahrenheit_to_celsius(temp_f):
    """Converts temperature in Fahrenheit
    to Celsius.
    """
    temp_c = (temp_f - 32.0) * (5.0/9.0)
    return temp_c


# this is the test function
def test_fahrenheit_to_celsius():
    temp_c = fahrenheit_to_celsius(temp_f=100.0)
    expected_result = 37.777777
    # assert raises an error if the condition is not met
    assert abs(temp_c - expected_result) < 1.0e-6
```

To run the test(s):

```
$ pytest example.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.

Good practice to add unit tests:

- Add the test function and run it.
- 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:

```
name: classification-task
channels:
  - conda-forge
dependencies:
  - python <= 3.12
  - click
  - numpy
  - pandas
  - scipy
  - altair
  - vl-convert-python
  - pytest
```

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

```
name: Test

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

jobs:
  build:
    runs-on: ubuntu-24.04

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

    - uses: mamba-org/setup-micromamba@v1
      with:
        micromamba-version: '2.0.5-0' # 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 generate-predictions.py
      shell: bash -el {0}
```

In the above example, we assume that we added a test function to `generate-predictions.py`.

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

- Run the test set **automatically** on every code change:

    - GitHub Actions
    - GitLab CI
- 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

✍ Exercise

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

- Add a unit test. **If you are unsure where to start**, you can try to move the majority vote into a separate function and write a test function for it.
- Try to run pytest 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. Already the thought process can be very helpful.

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:

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

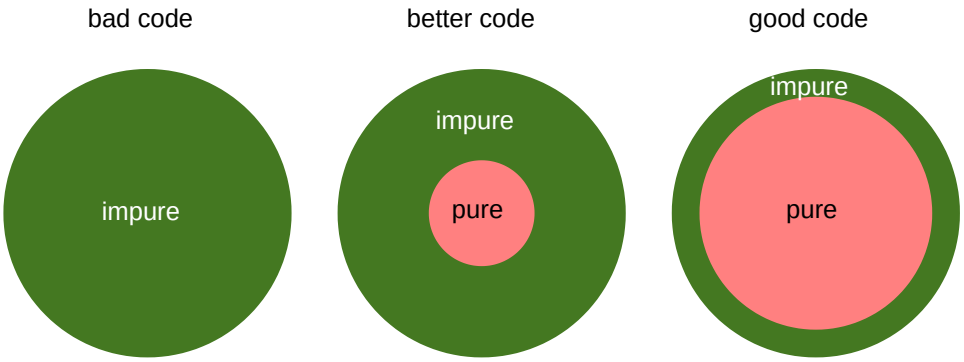
$$(f \circ g)(x) = f(g(x))$$

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**:

```
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

How to parallelize independent tasks using workflows (example: Snakemake)

Objectives

- Understand the concept of a workflow management tool.
- Instead of thinking in terms of individual step-by-step commands, think in terms of **dependencies** (**rules**).
- Try to port our computational pipeline to Snakemake.
- See how Snakemake can identify independent steps and run them in parallel.
- It is not our goal to remember all the details of Snakemake.

The problem

Imagine we want to process a large number of similar input data.

This could be one way to do it:

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

num_rounds=10

for i in $(seq -w 1 ${num_rounds}); do
    python generate_data.py \
            --num-samples 2000 \
            --training-data data/train_${i}.csv \
            --test-data data/test_${i}.csv

    python generate_predictions.py \
            --num-neighbors 7 \
            --training-data data/train_${i}.csv \
            --test-data data/test_${i}.csv \
            --predictions results/predictions_${i}.csv

    python plot_results.py \
            --training-data data/train_${i}.csv \
            --predictions results/predictions_${i}.csv \
            --output-chart results/chart_${i}.png
done
```

Discuss possible problems with this approach.

Thinking in terms of dependencies

For the following we will assume that we have the input data available:

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

num_rounds=10

for i in $(seq -w 1 ${num_rounds}); do
    python generate_data.py \
            --num-samples 2000 \
            --training-data data/train_${i}.csv \
            --test-data data/test_${i}.csv
done
```

From here on we will **focus on the processing part**.

The central file in Snakemake is the `snakefile`. This is how we can express the pipeline in Snakemake (below we will explain it):

```
# the comma is there because glob_wildcards returns a named tuple
numbers, = glob_wildcards("data/train_{number}.csv")


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


rule chart:
    input:
        script="plot_results.py",
        predictions="results/predictions_{number}.csv",
        training="data/train_{number}.csv"
    output:
        "results/chart_{number}.svg"
    log:
        "logs/chart_{number}.txt"
    shell:
        """
        python {input.script} --training-data {input.training} --predictions
{input.predictions} --output-chart {output}
        """


rule predictions:
    input:
        script="generate_predictions.py",
        training="data/train_{number}.csv",
        test="data/test_{number}.csv"
    output:
        "results/predictions_{number}.csv"
    log:
        "logs/predictions_{number}.txt"
    shell:
        """
        python {input.script} --num-neighbors 7 --training-data {input.training} --
test-data {input.test} --predictions {output}
        """
```

Explanation:

- The `snakefile` contains 3 **rules** and it will run the “all” rule by default unless we ask it to produce a different “target”:
  - “all”
  - “chart”
  - “predictions”
- Rules “predictions” and “chart” depend on **input** and produce **output**.
- Note how “all” depends on the output of the “chart” rule and how “chart” depends on the output of the “predictions” rule.
- The **shell** part of the rule shows how to produce the output from the input.
- We ask Snakemake to collect all files that match `"data/train_{number}.csv"` and from this to infer the **wildcards** `{number}`.

- Later we can refer to `{number}` throughout the `snakefile`.
- This part defines what we want to have at the end:

```
rule all:
    input:
        expand("results/chart_{number}.svg", number=numbers)
```

- Rules correspond to steps and parameter scanning can be done with wildcards.

Exercise

✍ Exercise: Practicing with Snakemake

1. Create a `snakefile` (above) and run it with Snakemake (adjust number of cores):

```
$ snakemake --cores 8
```

2. Check the output. Did it use all available cores? How did it know which steps it can start in parallel?
3. Run Snakemake again. Now it should finish almost immediately because all the results are already there. Aha! Snakemake does not repeat steps that are already done. You can force it to re-run all with `snakemake --delete-all-output`.
4. Remove few files from `results/` and run it again. Snakemake should now only re-run the steps that are necessary to get the deleted files.
5. Modify `generate_predictions.py` which is used in the rule "predictions". Now Snakemake will only re-run the steps that depend on this script.
6. It is possible to only process one file with (useful for testing):

```
$ snakemake results/predictions_09.csv
```

7. Add few more data files to the input directory `data/` (for instance by copying some existing ones) and observe how Snakemake will pick them up next time you run the workflow, without changing the `snakefile`.

What else is possible?

- With the option `--keep-going` we can tell Snakemake to not give up on first failure.
- The option `--restart-times 3` would tell Snakemake to try to restart a rule up to 3 times if it fails.
- It is possible to tell Snakemake to use a **locally mounted file system** instead of the default network file system for certain rules (documentation).

- Sometimes you need to run different rules inside different software environments (e.g. **Conda environments**) and this is possible.
- A lot **more is possible**:
  - Snakemake documentation
  - Snakemake tutorial

More elaborate example

In this example below we also scan over a range of numbers for the `--num-neighbors` parameter:

```
# the comma is there because glob_wildcards returns a named tuple
numbers, = glob_wildcards("data/train_{number}.csv")


# define the parameter scan for num-neighbors
neighbor_values = [1, 3, 5, 7, 9, 11]


# rule that collects all target files
rule all:
    input:
        expand("results/chart_{number}_{num_neighbors}.svg", number=numbers,
num_neighbors=neighbor_values)


rule chart:
    input:
        script="plot_results.py",
        predictions="results/predictions_{number}_{num_neighbors}.csv",
        training="data/train_{number}.csv"
    output:
        "results/chart_{number}_{num_neighbors}.svg"
    log:
        "logs/chart_{number}_{num_neighbors}.txt"
    shell:
        """
        python {input.script} --training-data {input.training} --predictions
{input.predictions} --output-chart {output}
        """


rule predictions:
    input:
        script="generate_predictions.py",
        training="data/train_{number}.csv",
        test="data/test_{number}.csv"
    output:
        "results/predictions_{number}_{num_neighbors}.csv"
    log:
        "logs/predictions_{number}_{num_neighbors}.txt"
    params:
        num_neighbors=lambda wildcards: wildcards.num_neighbors
    shell:
        """
        python {input.script} --num-neighbors {params.num_neighbors} --training-data
{input.training} --test-data {input.test} --predictions {output}
        """
```

Other workflow management tools

- Another very popular one is Nextflow.
- Many workflow management tools exist (overview).

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

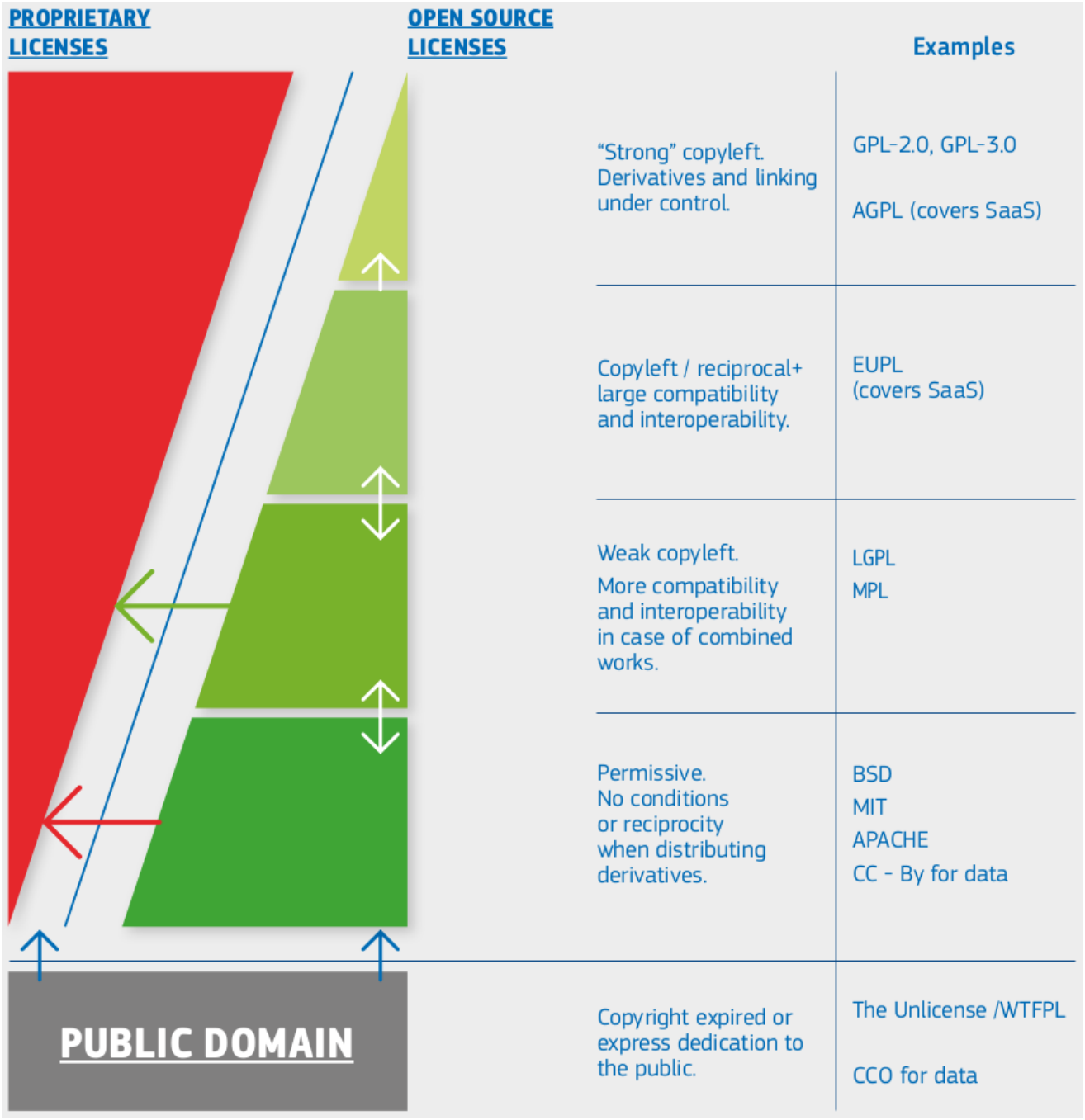
You find some great code that you want to reuse for your own publication.

- 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



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 `numpy`, `pandas`, `scipy`, `altair`, 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

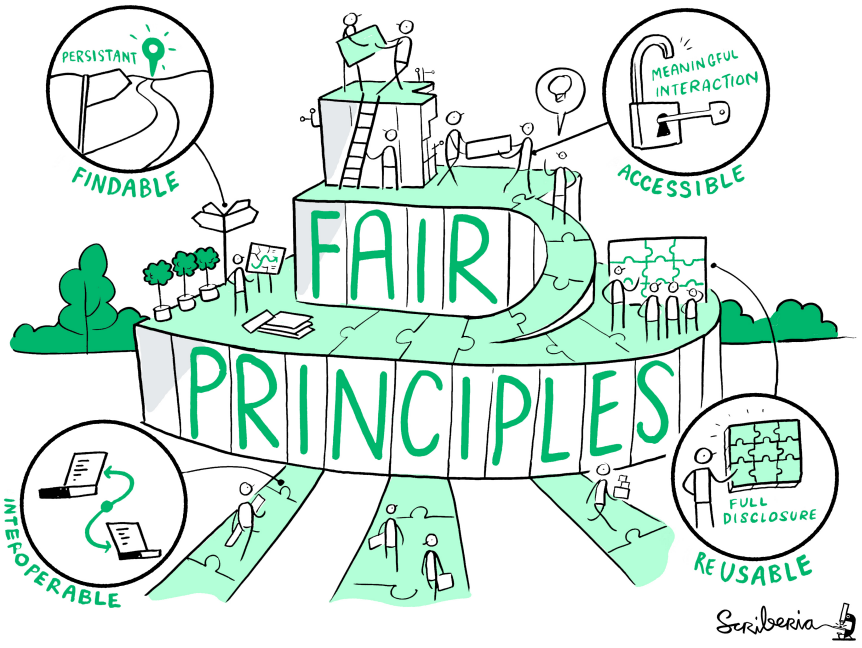
resulting combined software you could use a GPL license. However, re-licensing may not be necessary.
6. Derivative work would need to be shared under this strong copyleft license (e.g. AGPL or GPL), unless the components are only plugins or libraries.
7. If you keep your code for yourself, you may think 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: 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:

```
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)

Have a look at The Journal of Open Source Software (JOSS). JOSS is an open access journal with zero article processing charges where you can submit your GitHub repository directly (it requires writing a short markdown document as the "paper"). Example publication: Blobmodel: A Python package for generating superpositions of pulses in one and two dimensions.

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 can try to create an example repository with a Jupyter Notebook and run it through Binder to 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 can use one of the Python scripts from our example repository.

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 versatile, but also easier to use.
- `poetry` is a modern packaging tool which is more versatile 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 `example.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 = "example"
authors = [{name = "Firstname Lastname", email = "first.last@example.org"}]
license = {file = "LICENSE"}
classifiers = ["License :: OSI Approved :: European Union Public Licence 1.2 (EUPL
1.2)"]
dynamic = ["version", "description"]

[project.urls]
Home = "https://example.org"
```

   To have a more concrete example, if we package the `generate_data.py` script from the example repository, then replace the name `example` with `generate_data`.

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 = "example"
authors = [{name = "Firstname Lastname", email = "first.last@example.org"}]
license = {file = "LICENSE"}
classifiers = ["License :: OSI Approved :: European Union Public Licence 1.2 (EUPL
1.2)"]
dynamic = ["version", "description"]
dependencies = [
    "click",
    "numpy",
    "pandas",
]

[project.urls]
Home = "https://example.org"

[project.scripts]
example = "example: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 (adapt the names):

```
git+https://github.com/ORGANIZATION/REPOSITORY.git@main
```

A corresponding `envionment.yml` file for conda would look like this (adapt the names):

```yaml
name: experiment
channels:
  - conda-forge
dependencies:
  - python <= 3.12
  - pip
  - pip:
      - git+https://github.com/ORGANIZATION/REPOSITORY.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-24.04

    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

This lesson is a mashup of the following sources (all CC-BY):

- https://github.com/coderefinery/reproducible-python (shares common history with this lesson)
- https://github.com/coderefinery/python-progression

The lesson uses the following example repository:

- https://github.com/workshop-material/classification-task

The classification task has replaced the “planets” example repository used in the original lesson.