Enhancing Data Support: Practical Reproducibility

💬 WIP

This material is **work in progress**, except for this index page and the setup pages, please do not trust anything, yet.

This is part 2/2 of the webinar/workshop series on “Enhancing data support: Reproducibility” held at CSC in October 2025.

Part 1 introduced the concept of reproducibility in research and highlighted its importance.

This workshop will discuss two tools often involved when discussing reproducible computational reasearch: GitHub and Jupyter.

However, the time in this workshop is too short to give a full practical introduction of the tools.

We mainly want to provide a safe environment to experience these tools and give you some pointers as to when and where they are useful to inftroduce researchers to, as well as how they might be useful in your own work, even if you are not programming yourself.

Prerequisites

This is a lesson targeted at beginners, no prior knowledge of GitHub or Jupyter is needed.

If you would like to get your hands in the dirt and play around with the tools of this webinar, please come prepared with:

- A GitHub account (https://samumantha.github.io/github-jupyter-4-ds/github-account/)
- Access to our Noppe workspace (https://samumantha.github.io/github-jupyter-4-ds/noppe/); If you have HAKA, VIRTU or CSC account, use those methods to log in, if you do not know what these are or do not have one already, you do not have to do anything about it. We will fix during the workshop, if you want to try it.

Learning outcomes

- Discover best practices and understand the basic principles of version control and interactive computing platforms to support computational reproducibility.
- Utilize the basic features of GitHub and Jupyter.
- Know where those tools can be run and found.

Agenda for the workshop

10 min	Intro and practicalities
20 min	Basics and motivation
15 min	creating-using-web
5 min	Contributing to existing repositories using pull requests
5 min	How to organize a group's work
10 min	**break**
20 min	Jupyter Notebooks
25 min	Our first notebook
10 min	<no title>

Content

GitHub account

In this workshop, we use the public GitHub service and you need an account at https://github.com and a supported web browser. Basic GitHub accounts are free.

> **Why GitHub**
>
> We will do this exercise on GitHub but also GitLab and Bitbucket allow similar workflows and basically everything that we will discuss is transferable. With this material and these exercises we do not endorse the company GitHub. We have chosen to demonstrate a number of concepts using examples with GitHub because it is currently the most popular web platform for hosting Git repositories and the chance is high that you will interact with GitHub-based repositories even if you choose to host your Git repository on another platform.
>
> We also encourage course participants to use our new Nordic research software repository platform hosted in Denmark, for more information see https://coderefinery.org/repository/.

If you are concerned about the personal information to reveal to GitHub, for example how to keep your email address private, please review these instructions for keeping your email address private provided at GitHub.

Create a GitHub account

1. Go to https://github.com.
2. Click on the "Sign up" at the right-top corner.
3. Enter your username of your choice (if it is already used, you will get some suggestions), email address, and password.
4. Follow further instruction and verify your account.

GitHub may require you to enable multi-factor authentication (MFA). This is generally a good thing, but may take some time to set up. Luckily, you probably don't have to do this immediately. If you are prompted to enable MFA before the end of the workshop, follow GitHub's instructions since they are usually pretty good.

How to verify that this worked

If you can log in to https://github.com, you should be good to go.

Basics and motivation

❓ Questions

- What is version control and why?
- What are common terms used around version control?

❗ Objectives

- Get an idea of why version control can be useful.
- Understand the difference between Git and GitHub.
- Get a mental representation for commits and branches.

💬 What we will not cover

- Command line interface
- GitHub Desktop
- Cloning using SSH protocol and SSH keys
- Rebasing and squashing
- Many Git tricks which can be explored later

Version control

Why version control?

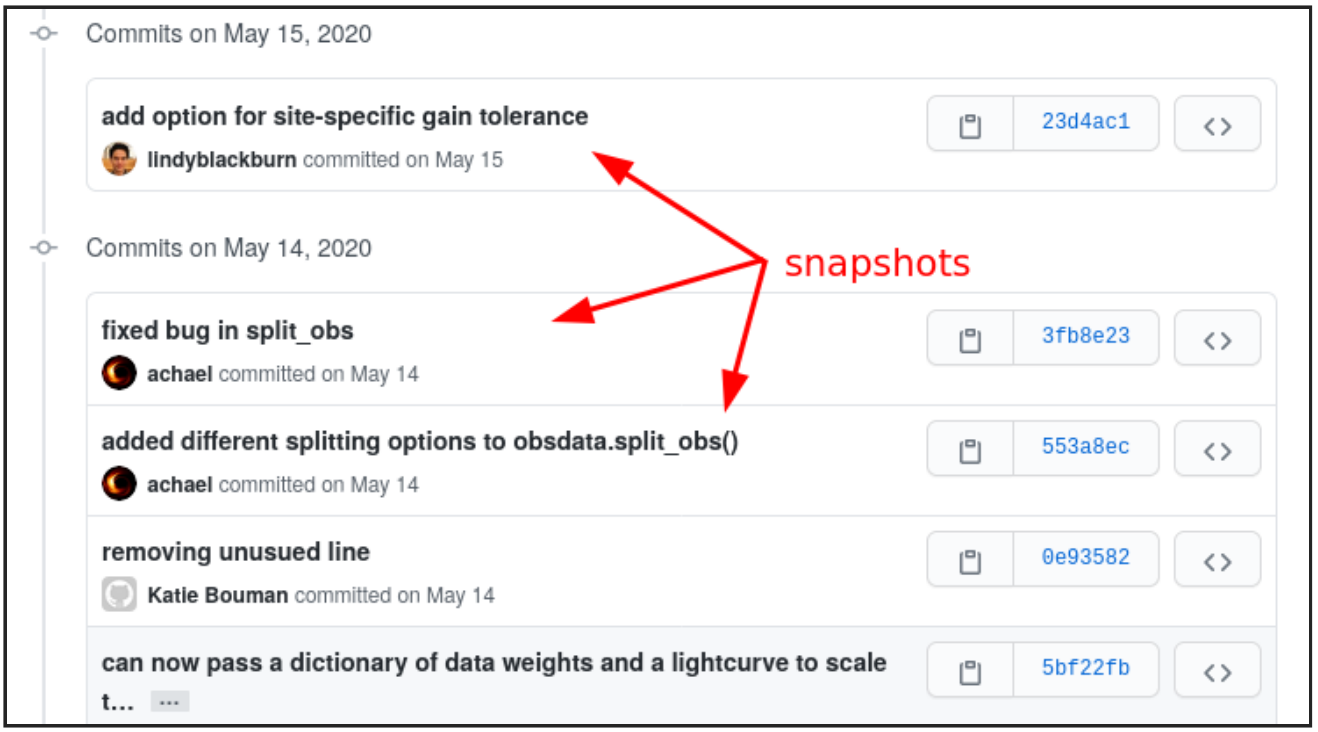
Version control is the answer to these questions:

- “It broke… hopefully I have a working version somewhere?”
- “Where is the latest version, and which one should I trust?”
- “I am sure it used to work. What changed, when, and why?”
- “When did this problem appear?”
- “Something looks different — what was updated, and who did it?”

-> Version control is simply a reliable way to remember, explain, and reproduce the evolution of digital research materials.

What are version control tools?

- Version control is a tool that can **record snapshots of a project**.
- You can think of version control like regularly taking a photo of your work.

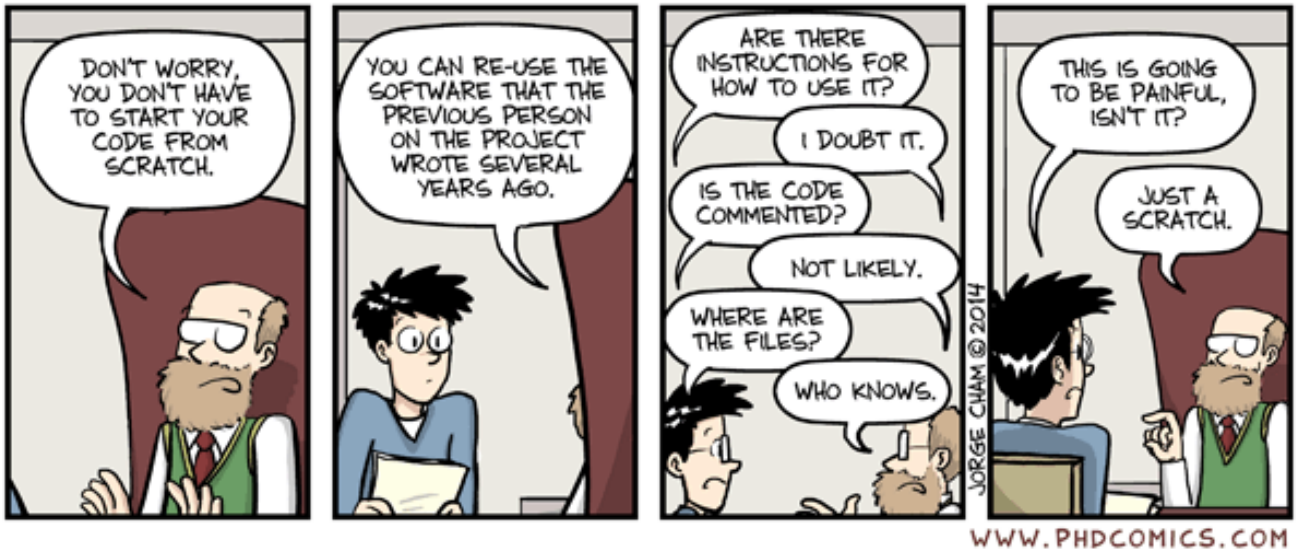


*Snapshots (**commits**) in the EHT-imaging repository.*

What we typically like to version control (or “snapshot”)?

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

Why are snapshots valuable? Reproducibility!



- We can always go back if we make a mistake.
- We can test new ideas without editing the working version
- If we discover a problem, we can find out when it was introduced.
- We have the means to refer to a well-defined version of a project when sharing, collaborating, and publishing.

Difference between Git and GitHub

Git

- Tool that can record and synchronize snapshots.
- Not the only tool that can record snapshots (other popular tools are Subversion and Mercurial).
- Not only a tool but also a format that can be read by many different tools.

GitHub

- Service that provides hosting for Git repositories with a nice web interface.
- Not the only service that provides this (other popular services are GitLab and Bitbucket).

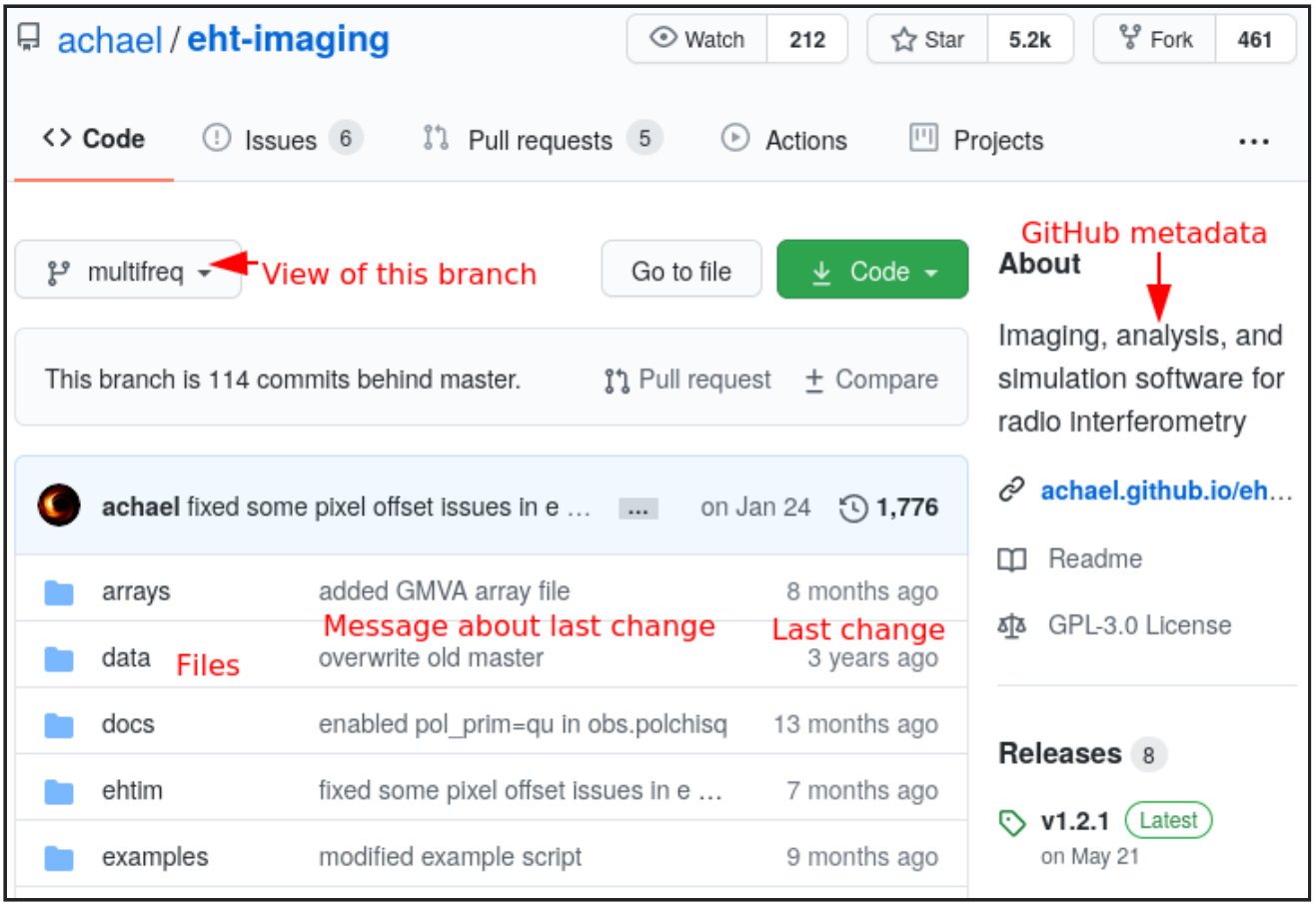
Git integration

- Many other tool also provide a git integration. Commonly used by researchers e.g. VSCode, RStudio, GitHub Desktop, JupyterLab, Overleaf mostly hiding git complexity, while providing same benefit.

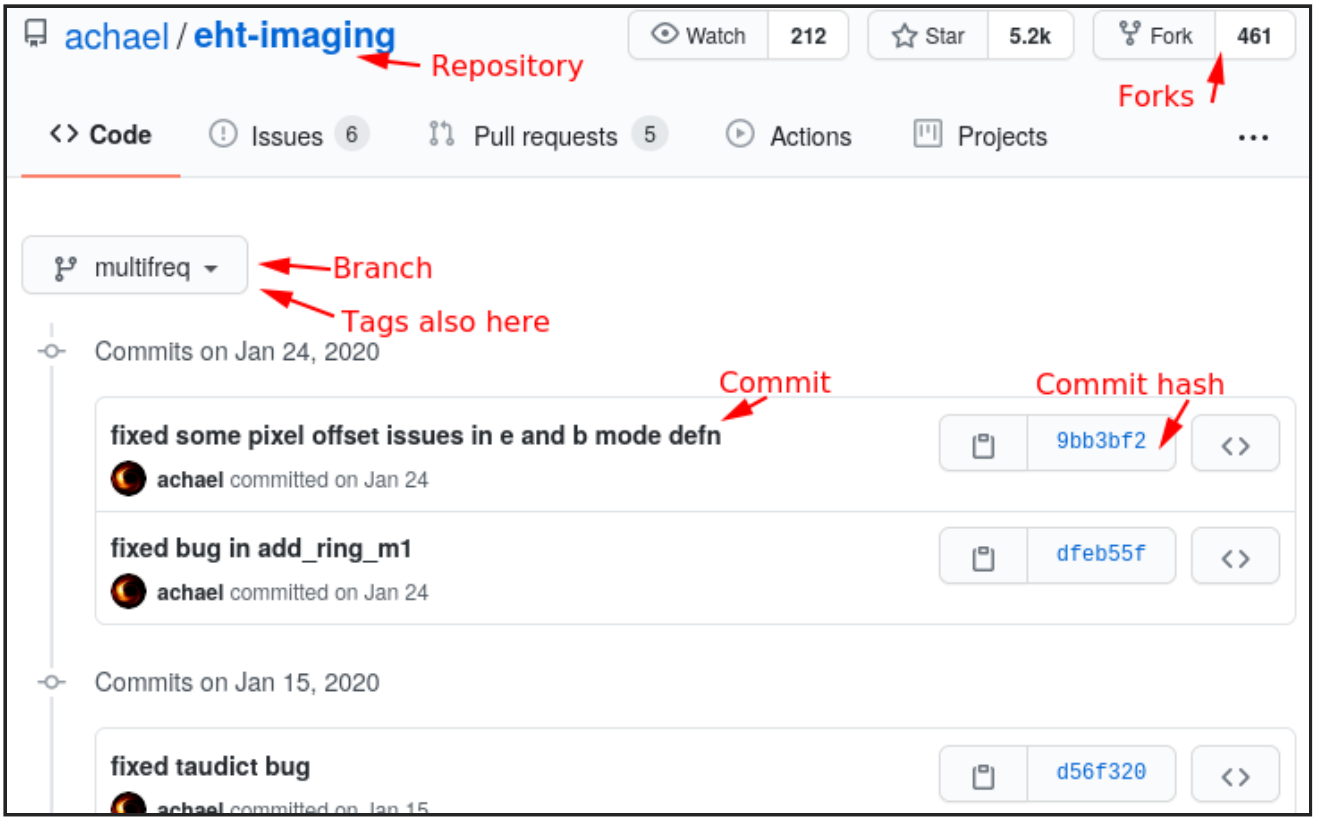
  You may have seen Git(Hub) many tutorials for git on the command line. We will stick to GitHub website. Why? Because for many cases, it is enough. Especially if you are contributing to existing non-code projects, this may be the fastest, easiest way to do it. Git and GitHub provide collaboration tools to all kinds of projects, and there are all kinds of good ways to use it.

Commits, branches, repositories, forks, clones

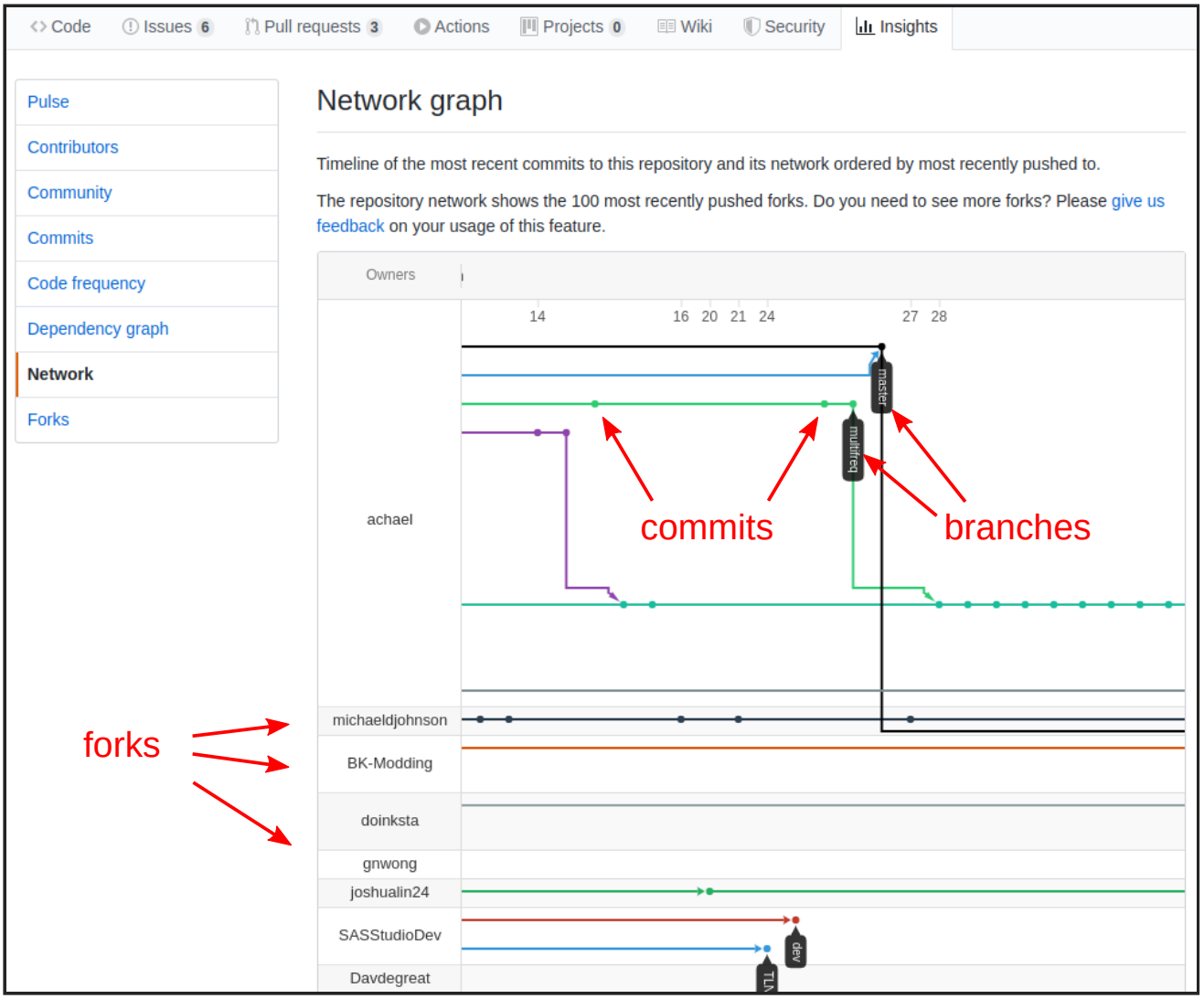
- **repository**: The project, contains all data and history (commits, branches, tags).
- **branch**: Independent development line, often we call the main development line `master`.
- **commit**: Snapshot of the project, gets a unique identifier (e.g. `c7f0e8bfc718be04525847fc7ac237f470add76e`).
- **tag**: A pointer to one commit, to be able to refer to it later. Like a sticky note 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 and you can make changes to your copy.
- **Pull request**: XX
- **Issue**: XX



GitHub file view of the EHT-imaging repository. This is the version of all files at a single point in time.



*Github history view of the EHT-imaging repository. This is the progression of the repository (with the **commit message** over time).*



*Network graph of all commits in the EHT-imaging repository. This shows the relationship between different **forks** of people who are contributing and sharing code.*

Interesting repositories to explore these concepts

- Activity inequality study
  - Contains data and code necessary to create figures from their article.
  - Data: https://github.com/timalthoff/activityinequality/tree/master/data
- Entire books are written using Git/GitHub:
  - https://github.com/alan-turing-institute/the-turing-way
- Papers under open review:
  - https://github.com/openjournals/joss-reviews/issues

Why use repositories? Think of your usecases for the following:

- All changes are recorded.
- We do not have to send changes via email.
- We can experiment with several ideas which might not work out (using branches).
- Several people can work on the same project at the same time (using branches).
- We do not have to wait for others to send us "the latest version" over email.
- We do not have to merge parallel developments by hand.
- Group-based access model where shared access is the default, instead of everything fundamentally owned by individuals who manage sharing as-needed: with Git you can easily have collaboration be the default.
- It is possible to serve websites directly from a repository.

{: .discussion}

💬 Discussion: workflows without version control

- How have you solved these in the past without version control?

Contributing to existing repositories using pull requests

TODO: Turn around: Non collaborator PR first then mention that one can also add collaborators. Maybe even put this as first exercise. Use stuff from this text:

Exercise

- We first open an issue via the web interface and describe our idea. In the issue we can collect feedback
- We clone the exercise repository using GitHub Desktop
- Create a new branch
- Add the new file to the local repository
- Commit and refer to the issue (e.g. here closing issue number 12: "this is the commit message, closes #12")
- Try to publish that branch (you may not have write permissions to the repository on GitHub)

- If you don't have write permissions: "Fork this repository", then try to publish the branch to the fork
- "Create Pull Request"
- If you forgot to refer to the issue in the commit, you can refer to the issue in the pull request

In this session we will learn how to contribute to repositories which either belong to a group that you are part of or belong to others.

We will do this in a progression from a small trivial fix to a change proposal and discuss the pros and cons.

💬 Different methods to "download" a repository

- Download all files in a repository as ZIP file (green button "Clone or download") if you do not plan to change files and if you are sure that you don't need to browse the history of file changes.
- **Possibly better: you clone the repository** (green button on the web or through GitHub Desktop or using command line) so that you can apply and track changes and possibly share them later.
- Cloning copies not only the latest version but all snapshots and all branches and tags: entire history.

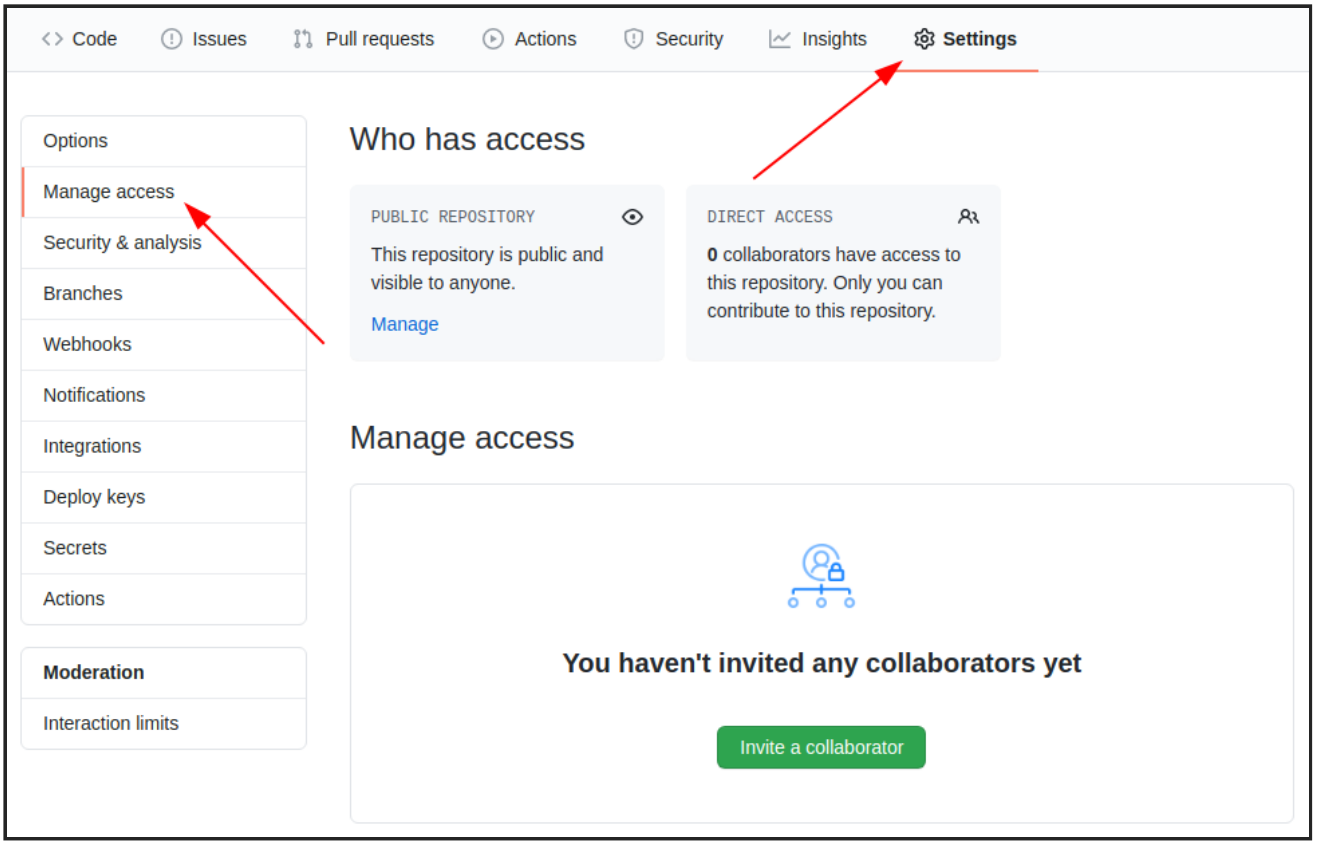
Instructor note

- The instructor will prepare an example repository and share the link with the participants (see instructor guide for more details).
- This is a good example text: https://www.gutenberg.org/ebooks/24542
- When showing how to add a collaborator, also show how to unwatch a repo.

Step 1: Learn how to add collaborators to your repository

Now we know how to share repositories and the first step to allow changes by others would be to add your group members or collaborators as "collaborators" under GitHub. This allows them to change things directly (but we'll actually do it with review).

- Instructor adds one or two learners who volunteer to later contribute a change via screen sharing.
- You can try this with one of the repositories which we created in the earlier episodes.
- Click on "Settings" (top right), then "Manage access" (left), then "Invite a collaborator" (green button).



From here on the collaborators can push changes in the same way as we have done in a single-person repository in the previous episodes.

💬 Discussion

- Discuss the advantages and possible disadvantages of this setup.

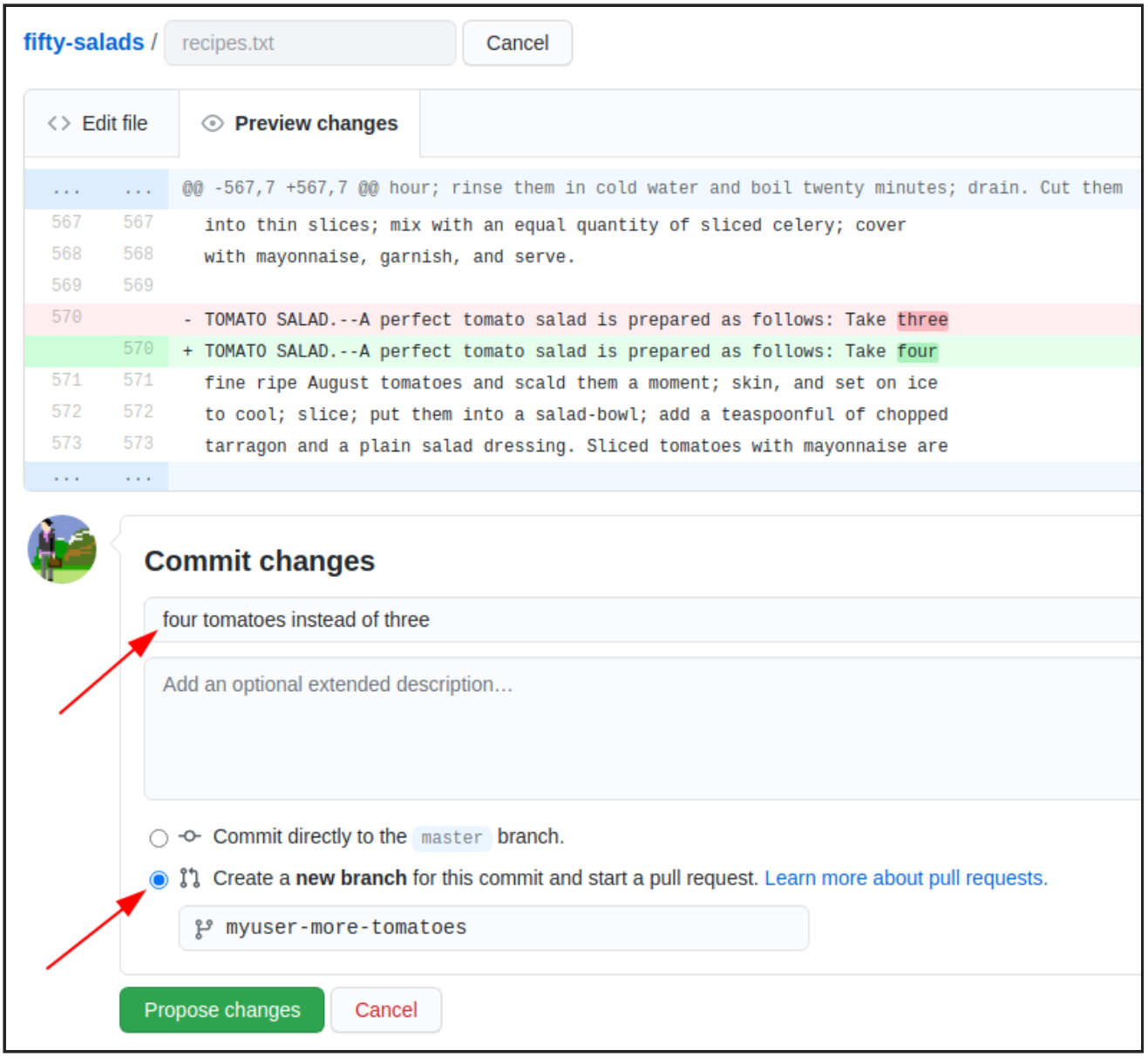
Step 2: Submit a small change via the web interface as collaborator

In the last episodes we learned how to directly commit changes either via web or via the desktop and you need to be a collaborator (have write permissions) to be able to do that.

In this exercise we will not change the `main` branch directly but we will submit a "pull request" (a **change proposal**) towards the `main` branch for **code review**.

Exercise: We will practice this by suggesting a change in a recipe book:

- Instructor shares an example repository and adds a volunteer learner as collaborator
- Learner shares screen and edits `recipes.txt` via web (click on the edit pen)
- We modify or extend one recipe from the example repository together by guiding the volunteer learner
- We do not commit directly to `main` but rather "Create a new branch for this commit and start a pull request."
- We choose a meaningful branch name (it can be useful to prefix it with your name so that we know who this branch belongs to)



- After we click “Propose file change” we are taken to this form:

Open a pull request

The change you just made was written to a new branch named `myuser-more-tomatoes`. Create a pull request bel

base: master ← compare: myuser-more-tomatoes ✓ **Able to merge.** These branches can be auto

four tomatoes instead of three

Write Preview

Leave a comment

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

Create pull request

✓ **Create pull request**
Open a pull request that is ready for review

Create draft pull request
Cannot be merged until marked ready for review

Remember, contributions to this repository should follow ou

1 commit **1** file changed **0** comments

Commits on Aug 03, 2020

bast four tomatoes instead of three

Showing **1 changed file** with **1 addition** and **1 deletion**.

2 recipes.txt

```
@@ -567,7 +567,7 @@ hour; rinse them in cold water and boil twenty minutes; drain. Cu
  into thin slices; mix with an equal quantity of sliced celery; cover
  with mayonnaise, garnish, and serve.

- TOMATO SALAD.--A perfect tomato salad is prepared as follows: Take three
+ TOMATO SALAD.--A perfect tomato salad is prepared as follows: Take four
  fine ripe August tomatoes and scald them a moment; skin, and set on ice
  to cool; slice; put them into a salad-bowl; add a teaspoonful of chopped
  tarragon and a plain salad dressing. Sliced tomatoes with mayonnaise are
```

- In there we verify the **source and target branch**, verify the **file changes**, can edit the **title** and **description** of the “pull request” (change proposal)
- After we have submitted the “pull request”, one of our collaborators can review it
- We can discuss and ask for changes before merging the changes “Merge pull request”

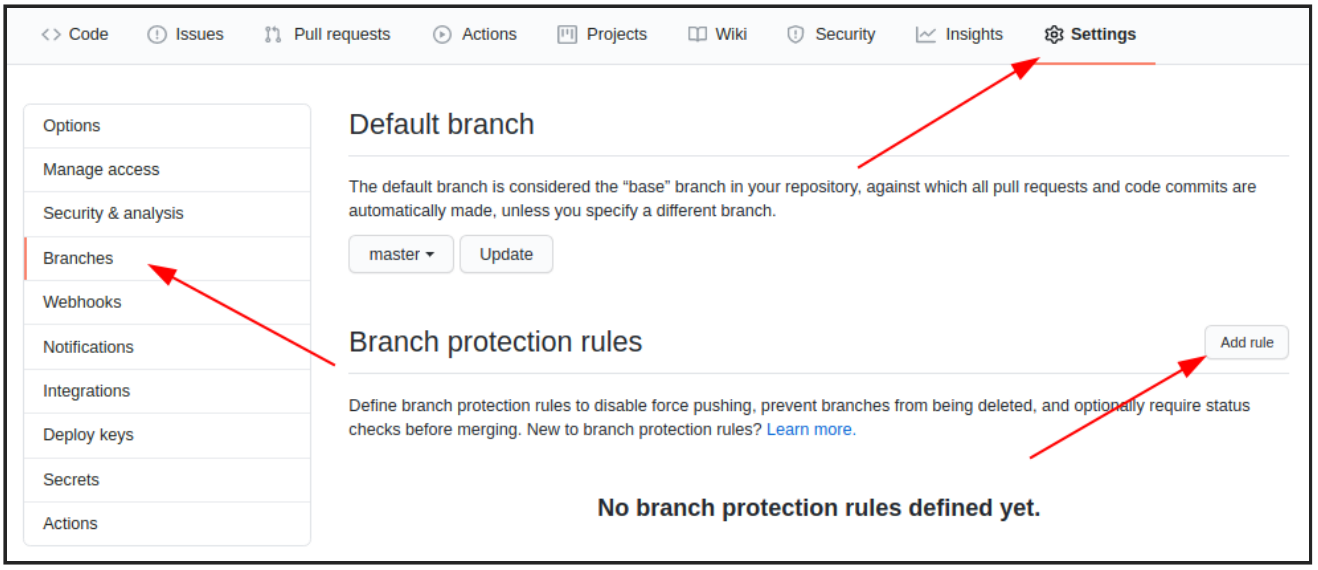
💬 Discussion

- Ideally submitter and reviewer should be two different persons. When is this best? When not?
- You can modify an open “pull request” by committing new changes to the branch
- Review is not only to assure quality but also to enhance learning and **knowledge transfer** within the group

---

To make sure that *all* changes of the `main` branch are reviewed and nobody can push commits to it directly, it can be useful to “protect” branches.

- “Settings”, then “Branches”, then “Add rule”:



Protecting the `main` branch “forces” all changes to it to be reviewed first. **We recommend this for group repositories**.

Step 3: Submit a small change via the web interface as external contributor

Submitting a change proposal as external contributor (we assume you are not added as “collaborator” and thus have no write-permissions to a repository) looks very similar to submitting a “pull request” to a repository with a protected `main` branch. Only this time you have no other choice than “Propose file change”.

Let’s try this with one participant who has not been added as collaborator sharing screen:

- Edit a file with the “pen” button
- Edit the commit message and click green button “Propose file change”
- This creates a **fork** of the repository (GitHub makes a copy of the original repository to your user space)
- You can now still review the change before submitting it, green button “Create pull request”
- Later you can remove the fork if you like

Step 4: Resolving a conflict

Instructor note

- “Non-talking instructor” prepares a conflicting commit during session (check what the first PR does).
- Conflict can be shown as demo.

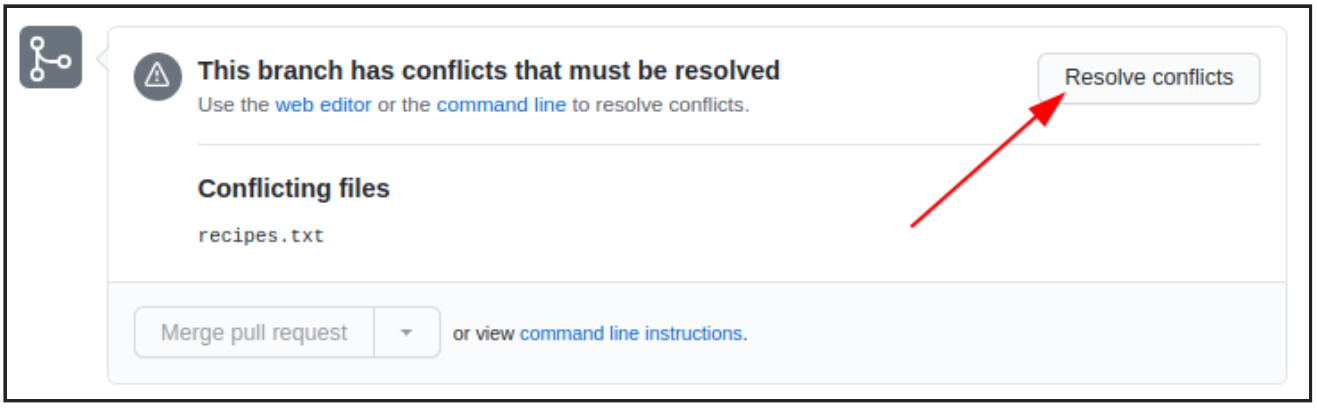
Exercise/demo: let us experience a conflict

When merging two branches a conflict can arise when the same file **portion** has been modified in two **different** ways on the two branches.

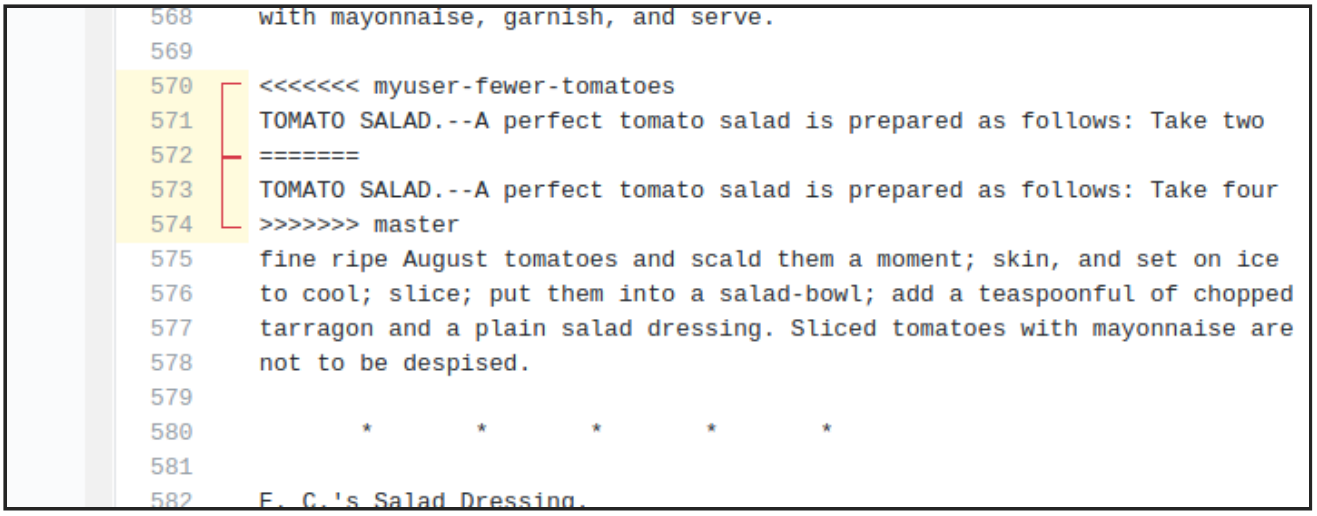
We can practice how a conflict looks and how to resolve it:

- Two participants should send two “pull requests” (change proposals) branching from `master` changing the same line in two different ways

- We merge together one of the pull requests (this will work)
- Then we try to merge the other and we see a conflict:



- We try to resolve the conflict via web
- Choose the version that you wish to keep, remove conflict markers, “Mark as resolved” and commit the change



💬 Discussion

- Compare with Google Docs: can you get conflicts there? What are the advantages and disadvantages?
- What can we do to avoid conflicts?

Bonus exercise

- Send a pull request with a typo/mistake in it and adjust the pull request with a subsequent commit. Discuss how adjusting pull requests can be a useful mechanism.

💬 Summary

- In this episode we learned how to propose changes and submit changes via “pull requests”.
- If you track and collaborate on text files it can be useful to wrap lines. If the entire paragraph is one long line, if will be more difficult to see what changed, and you risk seeing more conflicts.

- Protecting the `main` branch and insisting on every change going through a pull request can be useful to get feedback on your changes and to improve knowledge transfer.
- For controversial changes it can be useful to first discuss in an issue before submitting the changes.
- Note that you can mark pull requests as draft to collect feedback on unfinished work.
- Now that you know how to send improvements, we welcome improvements to this material also.

How to organize a group's work

GitHub organizations

Should I start a repository under my account or open a new organization?

- Single-person projects often start under own account.
- It is no problem to move a project from own namespace to an organization later.
- When starting a larger project with several people, possibly several affiliations, an organization may be a better start.
- If this is a GitHub pages project, then it will matter for the URL: `myuser.github.io/myproject/` vs. `ourorg.github.io/ourproject/`.

Should I add everybody as collaborator?

- If you are a handful of project collaborators it probably makes sense to add everybody as collaborators.
- But one does not have to be a *collaborator* to contribute (anybody can send contributions to public projects).
- External contributors don't have to be added.

Organizational permissions

- Organizations have **owners** and **members**.
- Owners can add additional members and delete repositories.
- Members can also be organized into teams.
- We recommend to write-protect the default branch and protect it against force-pushes and accidental deletions.

GitHub or GitLab?

- GitHub: probably better integrations (with services like Zenodo), probably more visibility (more users).
- GitLab: more features, you can also self-host, more advanced continuous integration.

- Your own university's GitLab: most control, local support, but limited visibility and you might lose access when you move on.
- Nordic GitLab: easier collaboration across organizations and national borders, visibility within Nordics.

Direct commits or pull requests?

- For single-person projects: direct pushes.
- If you have somebody who can help you with code review: use pull requests.
- For projects with 2 or more persons: agree on applying all changes via pull requests and create a new branch for every change.

Small vs. large changes

- Small changes or agreed upon improvements can be worked on directly.
- For larger changes first open an issue and describe your idea and collect feedback.
- Alternative: if you already have a larger change half-finished but you are unsure whether you are on the right track, open a **draft pull request**. These are meant to share unfinished drafts and collect suggestions.

Getting access to CSC Noppe

1. Go to https://noppe.csc.fi/
2. Accounts: * If you are from Finnish university and research institute, login with Haka, VIRTU or CSC account. Continue with 3) * Others do not need to do anything at this point, we will provide a separate username and password during the course, (You will then use "Special Login" for using these accounts)
3. In the Notebooks dashboard click the "Join workspace" button in the upper left corner, and copy-paste there the join code `byr14q9xlpph`
4. You should now see `Enhancing data support` Jupyter Notebook appear in the list.

If you see it, you are ready for the workshop!

Jupyter Notebooks

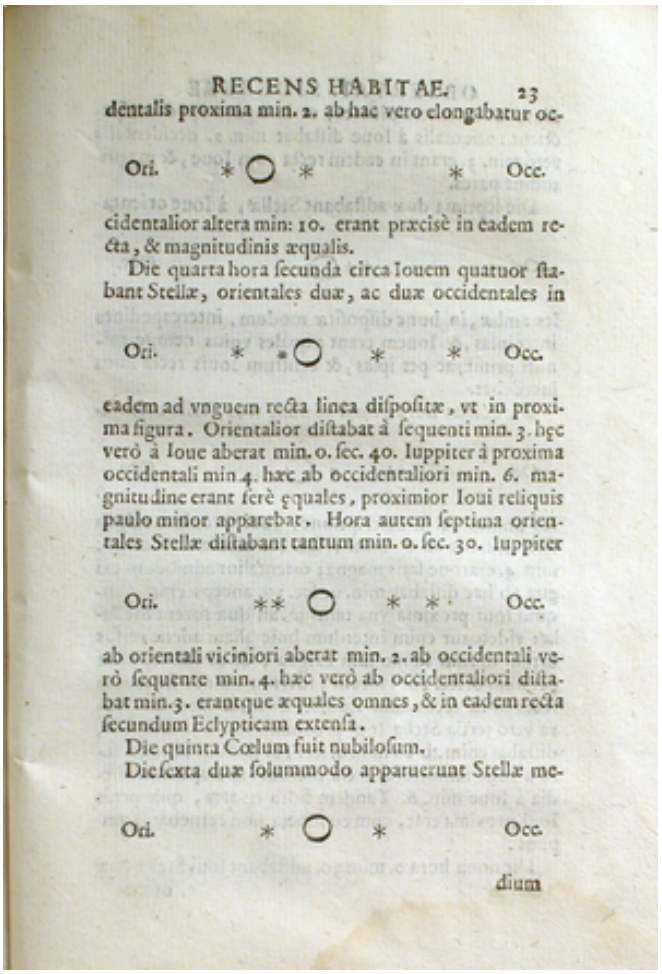
Objectives

- Know what it is
- Create a new notebook and save it
- Open existing notebooks from the web
- Be able to create text/markdown cells, code cells, images, and equations
- Know when to use a Jupyter Notebook for a Python project and when perhaps not to

[this lesson is adapted from https://coderefinery.github.io/jupyter/motivation/]

Jupyter is one of multiple options for executable notebooks (other RMarkdown, Pluto). Often used in teaching.

Motivation for Jupyter Notebooks



RECENS HABITAE. 23

dentalis proxima min. 2. ab hac vero elongabatur oc-

Ori. * ○ * * Occ.

cidentalior altera min: 10. erant præcisè in eadem re-
ctâ, & magnitudinis æqualis.

Die quarta hora secunda circa Iouem quatuor stabant Stellæ, orientales duæ, ac duæ occidentales in

Ori. * * ○ * * Occ.

eadem ad vnguem recta linea dispositæ, vt in proxima figura. Orientalior distabat à sequenti min. 3. hæc verò à Ioue aberat min. 0. sec. 40. Iuppiter à proxima occidentali min. 4. hæc ab occidentaliori min. 6. magnitudine erant ferè æquales, proximior Ioui reliquis paulo minor apparebat. Hora autem septima orientales Stellæ distabant tantum min. 0. sec. 30. Iuppiter

Ori. ** ○ * * · Occ.

ab orientali viciniori aberat min. 2. ab occidentali verò sequente min. 4. hæc verò ab occidentaliori distabat min. 3. erantque æquales omnes, & in eadem recta secundum Eclypticam extensa.

Die quinta Cœlum fuit nubilosum.

Die sexta duæ solummodo apparuerunt Stellæ me-

Ori. * ○ * Occ.

dium

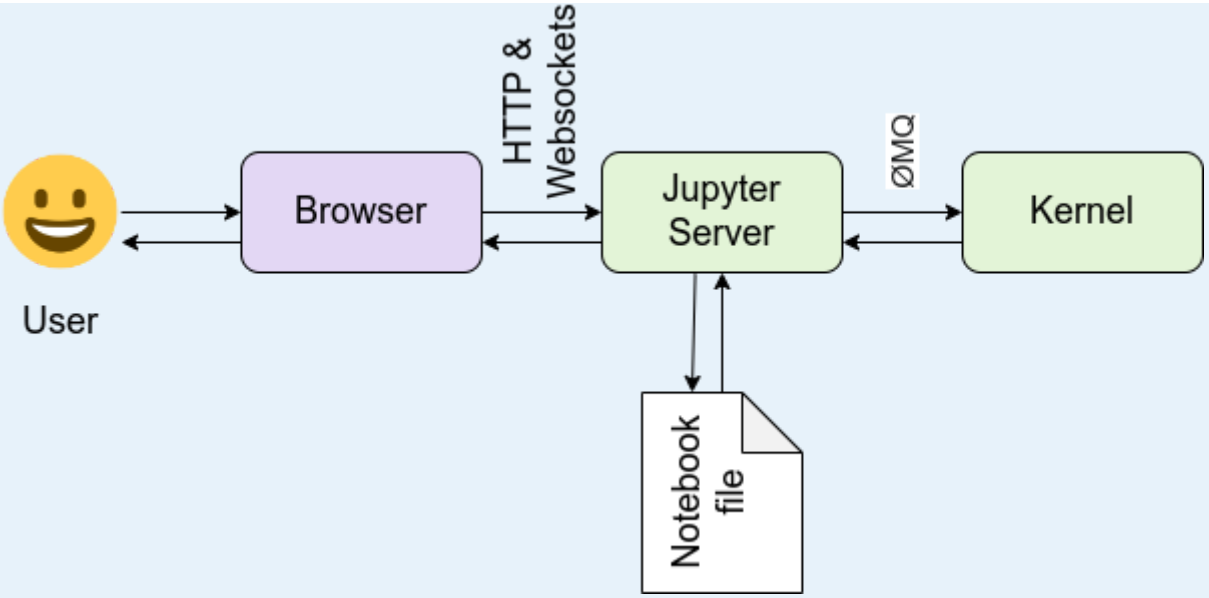
One of the first notebooks: Galileo's drawings of Jupiter and its Medicean Stars *from Sidereus Nuncius. Image courtesy of the History of Science Collections, University of Oklahoma Libraries (CC-BY).*

- **Code, text, equations, figures, plots**, etc. are interleaved, creating a *computational narrative*.
- *"an environment in which users execute code, see what happens, modify and repeat in a kind of iterative conversation between researcher and data"*
- The name "Jupyter" derives from Julia+Python+R, but today Jupyter kernels exist for dozens of programming languages.
- Gallery of interesting Jupyter Notebooks.

Note

Understanding the Jupyter architecture

It is important to understand how Jupyter Notebooks work behind the scenes. Here's a simple way to think about it:



Together, this setup allows you to write code, execute it, and see results — all in one place!

This is a simplified view — in practice, the Jupyter architecture can be extended with multiple users, remote kernels, and more. See the full architecture documentation at https://docs.jupyter.org/en/latest/projects/architecture/content-architecture.html

Jupyter Notebook: One notebook file in your browser Jupyter Lab: Workspace for notebooks, terminal other files, extensions etc

Both of above you can install and run on your own computer or in the cloud.

Jupyter Hub: Multi user self-hosting option (your institution might host their own)

What is Noppe?

Noppe (previously CSC Notebooks) offers web applications (RStudio and Jupyter) for self-learning, hosting courses and collaboration. The applications are accessed through a web browser and run in CSC cloud.

TODO: picture of Noppe

Use cases for notebooks

- Really good for step-by-step recipes (e.g. read data, filter data, do some statistics, plot the results)
- Experimenting with new ideas, testing new libraries/databases
- As an *interactive* development environment for code, data analysis, and visualization
- Keeping track of interactive sessions, like a **digital lab notebook**
- **Supplementary information with published articles**

Good practices

Run all cells or even **Restart Kernel and Run All Cells** before sharing/saving to verify that the results you see on your computer were not due to cells being run out of order.

This can be demonstrated with the following example:

```
numbers = [1, 2, 3, 4, 5]
arithmetic_mean(numbers)
```

We can first split this code into two cells and then re-define `numbers` further down in the notebook. If we run the cells out of order, the result will be different.

Limitations of notebooks

(Read more at Pitfalls of Jupyter Notebooks.)

- You can run cells in any order, which may lead to **confusing or incorrect results** if you're not careful.
- **Variables stay in memory** even if you delete the cell that created them — this can lead to surprises!
- Hard to know what's been run and in what order unless you **restart the kernel and run all cells**.
- Notebooks don't by default work well with **version control (like Git)** because output cells can make the files messy.
- Mixing too much **text, code, and results** can make notebooks hard to read or maintain if they get too long.
- Notebooks aren't ideal for building reusable **scripts, libraries, or production code** — they are more for exploration.

If you're using notebooks for more than quick experiments or interactive work, it's good to be aware of these limitations — and consider moving parts of your workflow to regular `.py` files when needed.

Keypoints

- Jupyter Notebooks combine code, text, plots, and results in one document: ideal for interactive data exploration.
- Use **markdown cells** for documentation and **code cells** for execution.
- Always **restart and run all cells** before sharing or saving to avoid confusion from out-of-order execution.
- Notebooks are great for exploration, teaching, and prototyping, but not the best tool for large software projects or version-controlled pipelines.
- Watch out for **common pitfalls** like hidden variables, confusing outputs, and version control issues.

Our first notebook

✍ Exercise Jupyter-1: Create a notebook (15 min)

- Open a new notebook
- Rename the notebook
- Create a **markdown cell** with a section title, a short text, an image, and an equation

```
# Title of my notebook

Some text.

![Photo of Galilei's manuscript]
(https://upload.wikimedia.org/wikipedia/commons/b/b3/Galileo_Galilei_%281564_-_1642%29_-_Serenissimo_Principe_-_manuscript_with_observations_of_Jupiter_and_four_of_its_moons%2C_1610.png)

$E = mc^2$
```

- Most important shortcut: **Shift + Enter**, to run current cell and create a new one below.
- Create a **code cell** where you define the `arithmetic_mean` function:

```
def arithmetic_mean(sequence):
    s = 0.0
    for element in sequence:
        s += element
    n = len(sequence)
    return s / n
```

- In a different cell, call the function:

```
arithmetic_mean([1, 2, 3, 4, 5])
```

- In a new cell, let us try to plot a layered histogram. **Note**: as a beginner you might be struggling to understand what is going on at this stage compared to the previous simple examples. This is to show you that a good starting point is to copy pasting code from tutorials to see if you can re-run them locally, and then proceed to understand the code and try some changes.

```
# this example is from https://altair-
viz.github.io/gallery/layered_histogram.html

import pandas as pd
import altair as alt
import numpy as np
np.random.seed(42)

# Generating Data
source = pd.DataFrame({
    'Trial A': np.random.normal(0, 0.8, 1000),
    'Trial B': np.random.normal(-2, 1, 1000),
    'Trial C': np.random.normal(3, 2, 1000)
})

alt.Chart(source).transform_fold(
    ['Trial A', 'Trial B', 'Trial C'],
    as_=['Experiment', 'Measurement']
).mark_bar(
    opacity=0.3,
    binSpacing=0
).encode(
    alt.X('Measurement:Q').bin(maxbins=100),
    alt.Y('count()').stack(None),
    alt.Color('Experiment:N')
)
```

- Run all cells.
- Save the notebook.
- Observe that a "#" character has a different meaning in a code cell (code comment) than in a markdown cell (heading).
- Your notebook should look like this one.

Plotting with Vega-Altair

Objectives

- Be able to create simple plots with Vega-Altair and tweak them
- Know how to look for help
- Know how to tweak example plots from a gallery for your own purpose
- We will build up this notebook (spoiler alert!)

We will experiment with some example weather data obtained from FMI (CC BY 4.0). The data is in CSV format (comma-separated values) and contains daily and monthly weather data for two cities in Finland: Espoo and Kajaani.

We will use the Pandas library to read the data into a dataframe.

Pandas can read from and write to a large set of formats (overview of input/output functions and formats). We will load a CSV file directly from the web. Instead of using a web URL we could use a local file name instead.

Pandas dataframes are a great data structure for **tabular data**.

Reading data into a dataframe

We can try this together in a notebook: Using Pandas we can **merge, join, concatenate, and compare** dataframes, see https://pandas.pydata.org/pandas-docs/stable/user_guide/merging.html.

Let us try to **concatenate** two dataframes: one for Tromsø weather data (we will now load monthly values) and one for Oslo:

```
import pandas as pd

url_prefix = "https://raw.githubusercontent.com/coderefinery/data-visualization-python/main/data/"

data_tromso = pd.read_csv(url_prefix + "tromso-monthly.csv")
data_oslo = pd.read_csv(url_prefix + "oslo-monthly.csv")

data_monthly = pd.concat([data_tromso, data_oslo], axis=0)

# let us print the combined result
data_monthly
```

Data preprocessing

There is a problem which we may not see yet: Dates are not in a standard date format (YYYY-MM-DD). We can fix this:

```
# replace mm.yyyy to date format
data_monthly["date"] = pd.to_datetime(list(data_monthly["date"]), format="%m.%Y")
```

With Pandas it is possible to do a lot more (adjusting missing values, fixing inconsistencies, changing format).

What is in a dataframe?

The name pandas is derived from the term "**pan**el **da**ta".


A pandas dataframe object is composed of rows and columns.
A pandas dataframe object is composed of rows and columns.

Let us explore these together in the notebook (run these in separate cells):

```
# print an overview of the dataset
data_monthly

# print the first 5 rows
data_monthly.head()

# print the last 5 rows
data_monthly.tail()

# print all column titles - no parentheses here
data_monthly.columns

# show which data types were detected
data_monthly.dtypes

# print table dimensions - no parentheses here
data_monthly.shape

# print one column
data_monthly["max temperature"]

# get some statistics
data_monthly["max temperature"].describe()

# what was the maximum temperature?
data_monthly["max temperature"].max()

# print all rows where max temperature was above 20
data_monthly[data_monthly["max temperature"] > 20.0]
```

Why are we starting with Vega-Altair?

- Concise and powerful
- “Simple, friendly and consistent API” allows us to focus on the data visualization part and get started without too much Python knowledge
- The way it **combines visual channels with data columns** can feel intuitive
- Interfaces very nicely with Pandas
- Easy to change figures
- Good documentation
- Open source
- Makes it easy to save figures in a number of formats (svg, png, html)
- Easy to save interactive visualizations to be used in websites

Reading data into a dataframe

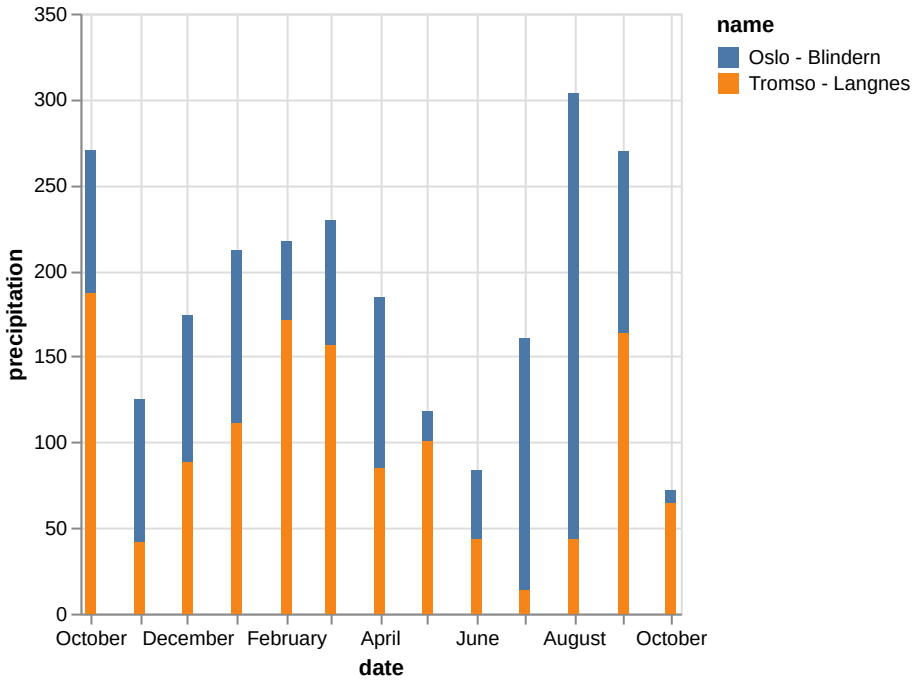
From the previous section, let’s load the data in our jupyter notebook and fix the dates.

```
import pandas as pd

url_prefix = "https://raw.githubusercontent.com/coderefinery/data-visualization-python/main/data/"

data_tromso = pd.read_csv(url_prefix + "tromso-monthly.csv")
data_oslo = pd.read_csv(url_prefix + "oslo-monthly.csv")

data_monthly = pd.concat([data_tromso, data_oslo], axis=0)

# replace mm.yyyy to date format
data_monthly["date"] = pd.to_datetime(list(data_monthly["date"]), format="%m.%Y")

# let us print the combined result
data_monthly
```

Plotting the data

Now let's plot the data. We will start with a plot that is not optimal and then we will explore and improve a bit as we go:

```
import altair as alt

alt.Chart(data_monthly).mark_bar().encode(
    x="date",
    y="precipitation",
    color="name",
)
```



Monthly precipitation for the cities Oslo and Tromsø over the course of a year.

💬 Let us pause and explain the code

- `alt` is a short-hand for `altair` which we imported on top of the notebook

- `Chart()` is a function defined inside `altair` which takes the data as argument
- `mark_bar()` is a function that produces bar charts
- `encode()` is a function which encodes data columns to **visual channels**

Observe how we connect (encode) **visual channels** to data columns:

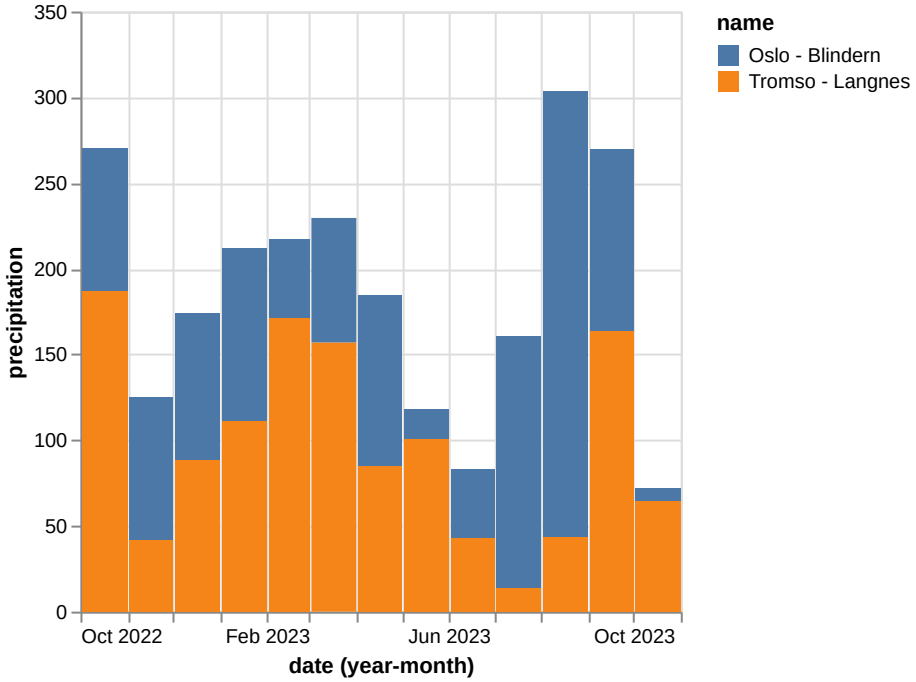
- x-coordinate with “date”
- y-coordinate with “precipitation”
- color with “name” (name of weather station; city)

We can improve the plot by giving Vega-Altair a bit more information that the x-axis is **temporal** (T) and that we would like to see the year and month (yearmonth):

```
alt.Chart(data_monthly).mark_bar().encode(
    x="yearmonth(date):T",
    y="precipitation",
    color="name",
)
```

Apart from T (temporal), there are other encoding data types:

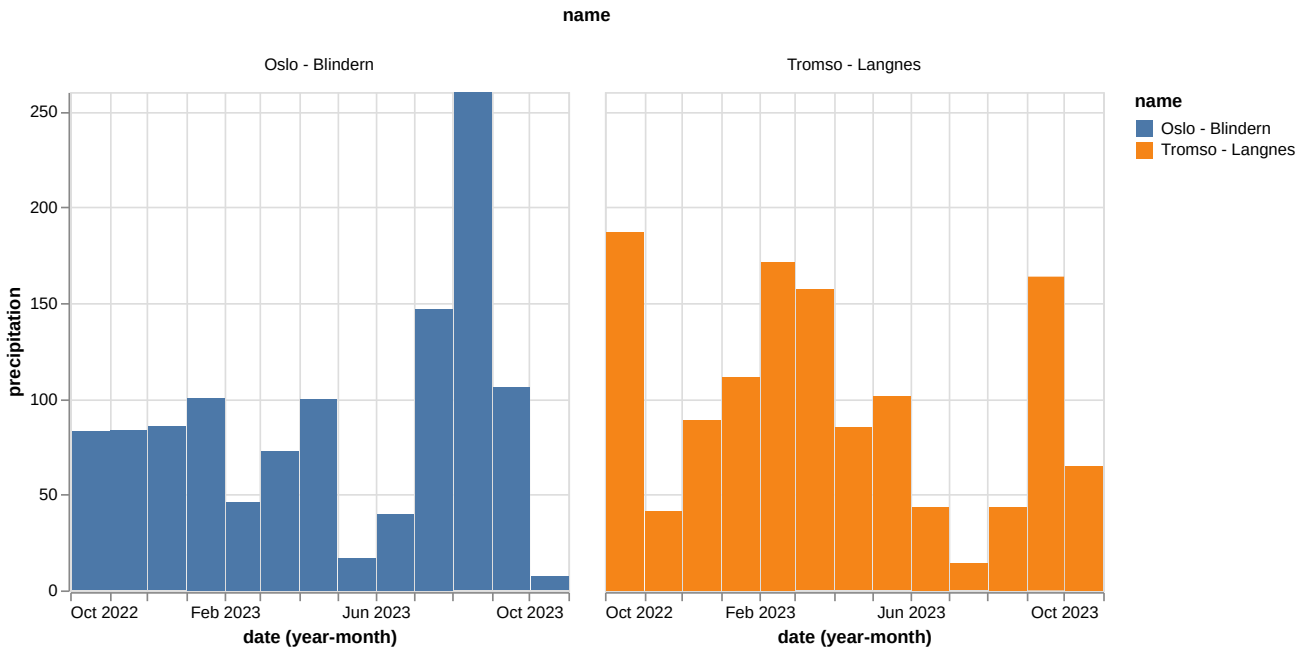
- Q (quantitative)
- O (ordinal)
- N (nominal)
- T (temporal)
- G (geojson)



Monthly precipitation for the cities Oslo and Tromsø over the course of a year.

Let us improve the plot with another one-line change:

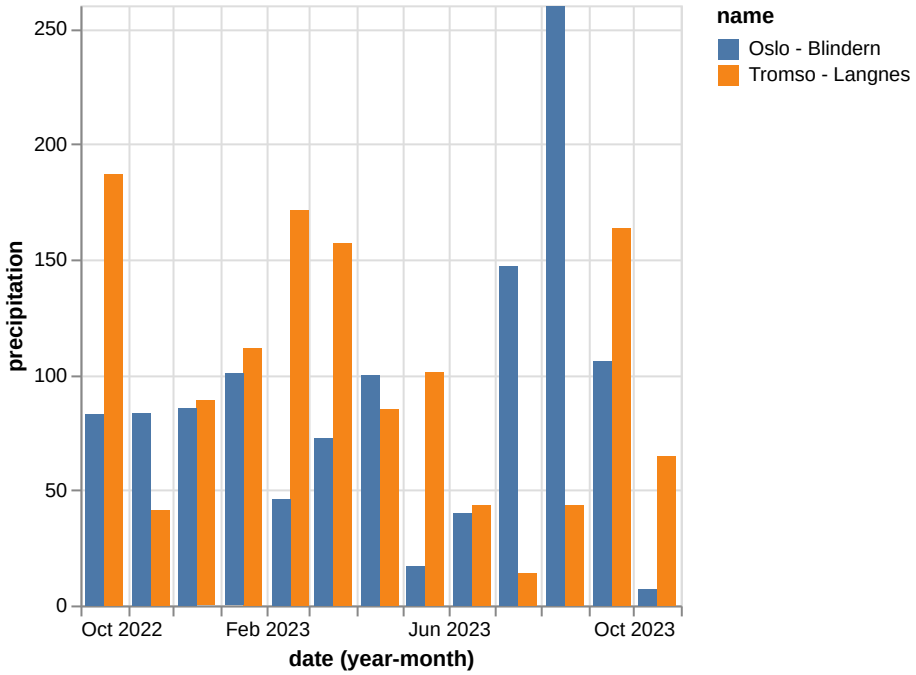
```
alt.Chart(data_monthly).mark_bar().encode(
    x="yearmonth(date):T",
    y="precipitation",
    color="name",
    column="name",
)
```



Monthly precipitation for the cities Oslo and Tromsø over the course of a year with with both cities plotted side by side.

With another one-line change we can make the bar chart stacked:

```
alt.Chart(data_monthly).mark_bar().encode(
    x="yearmonth(date):T",
    y="precipitation",
    color="name",
    xOffset="name",
)
```

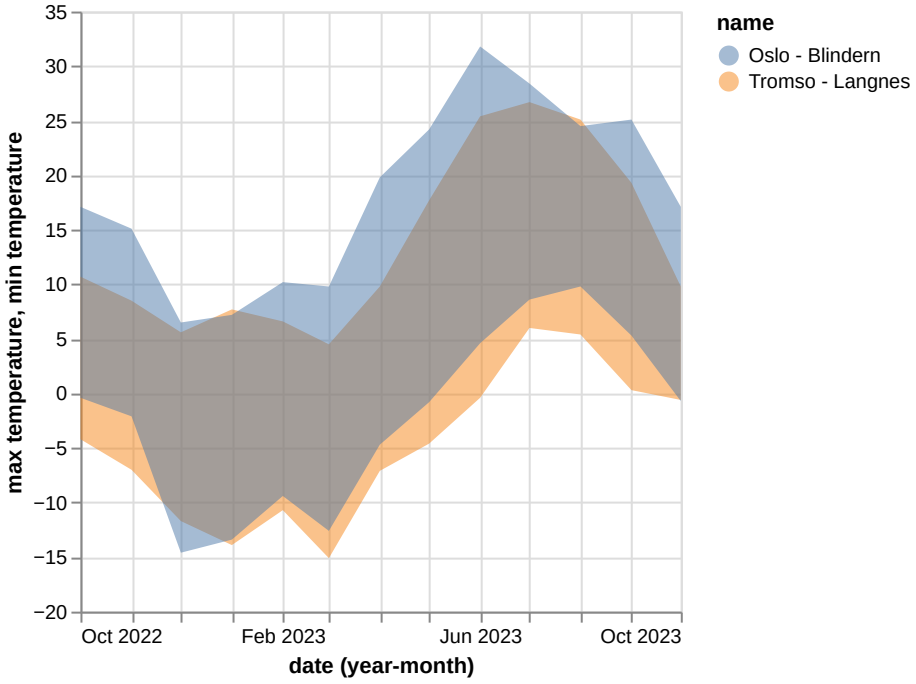


Monthly precipitation for the cities Oslo and Tromsø over the course of a year plotted as stacked bar chart.

This is not publication-quality yet but a really good start!

Let us try one more example where we can nicely see how Vega-Altair is able to map visual channels to data columns:

```
alt.Chart(data_monthly).mark_area(opacity=0.5).encode(
    x="yearmonth(date):T",
    y="max temperature",
    y2="min temperature",
    color="name",
)
```



Monthly temperature ranges for two cities in Norway.

💬 What other marks and other visual channels exist?

- Overview of available marks
- Overview of available visual channels
- Gallery of examples

Themes

In Vega-Altair you can change the theme and select from a long list of themes. On top of your notebook try to add:

```
alt.themes.enable('dark')
```

Then re-run all cells. Later you can try some other themes such as:

- `fivethirtyeight`
- `latimes`
- `urbaninstitute`

You can even define your own themes!

✍ Discover the Vega-Altair gallery of examples

Try to rerun some examples from the Gallery of examples. Which one did you choose? Were you able to reproduce the figures? Did you try `alt.themes.enable('dark')?

Note: you will need to first run in a cell the command `!pip install vega_datasets` to make the demo data available.

❗ Keypoints

- Browse a number of example galleries to help you choose the library that fits best your work/style.
- Minimize manual post-processing and try to script all steps.
- CSV (comma-separated values) files are often a good format to store the data that we wish to plot.
- Read the data into a Pandas dataframe and then plot it with Vega-Altair where you connect data columns to visual channels.

Additional:

Customizing plots

Objectives

- Know where to look to find out how to tweak plots
- Start with a relatively simple example and build up more and more features
- See the process of going from a raw plot towards a publication-ready plot
- We will build up this notebook (spoiler alert!)

Loading and plotting a dataset

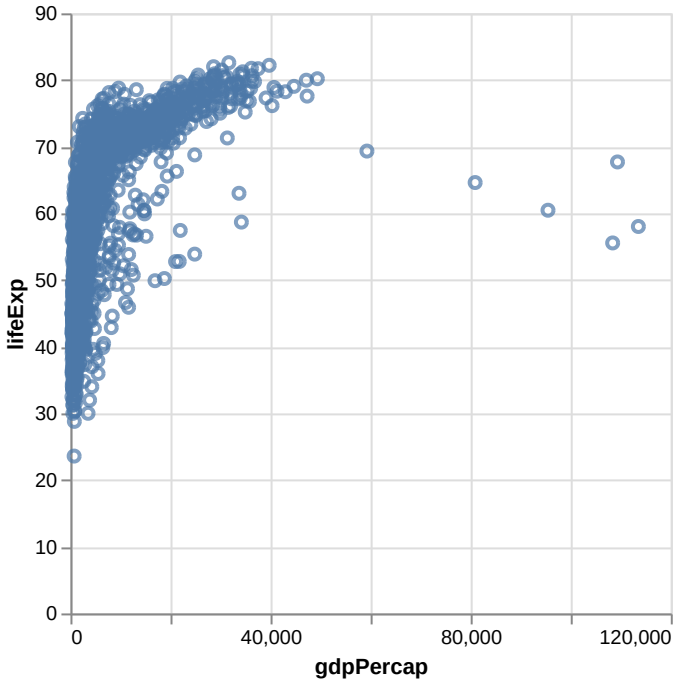
In this lesson will work with one of the Gapminder datasets.

Let us together read and plot the data and then we explain what is happening and we will improve the figure together. First we read and inspect the data:

```
# import necessary libraries
import altair as alt
import pandas as pd

# read the data
url_prefix = "https://raw.githubusercontent.com/plotly/datasets/master/"
data = pd.read_csv(url_prefix + "gapminder_with_codes.csv")

# print overview of the dataset
data
```

With very few lines we can get the first raw plot:

```
alt.Chart(data).mark_point().encode(
    x="gdpPercap",
    y="lifeExp",
)
```



First raw plot with all countries and all years.

Observe how we connect (encode) **visual channels** to data columns:

- x-coordinate with “gdpPercap”
- y-coordinate with “lifeExp”

The following code would have the same effect but the above version might be easier to read:

```
alt.Chart(data).mark_point().encode(x="gdpPercap", y="lifeExp")
```

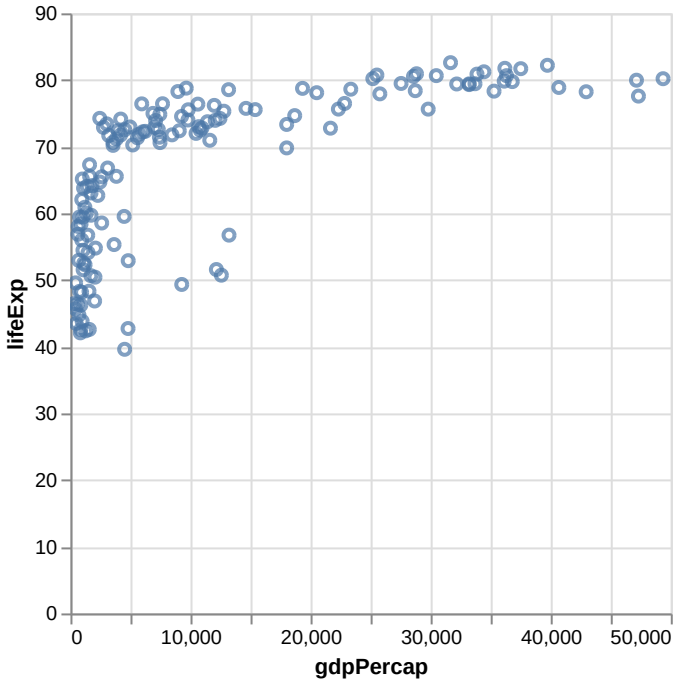
💬 **Let us pause and explain the code**

- `alt` is a short-hand for `altair` which we imported on top of the notebook
- `Chart()` is a function defined inside `altair` which takes the data as argument
- `mark_point()` is a function that produces scatter plots
- `encode()` is a function which encodes data columns to visual channels

Filtering data

In Vega-Altair we can chain functions. Let us add two more functions: The first will apply a filter, the second will make the plot interactive:

```
alt.Chart(data).mark_point().encode(
    x="gdpPercap",
    y="lifeExp",
).transform_filter(alt.datum.year == 2007).interactive()
```



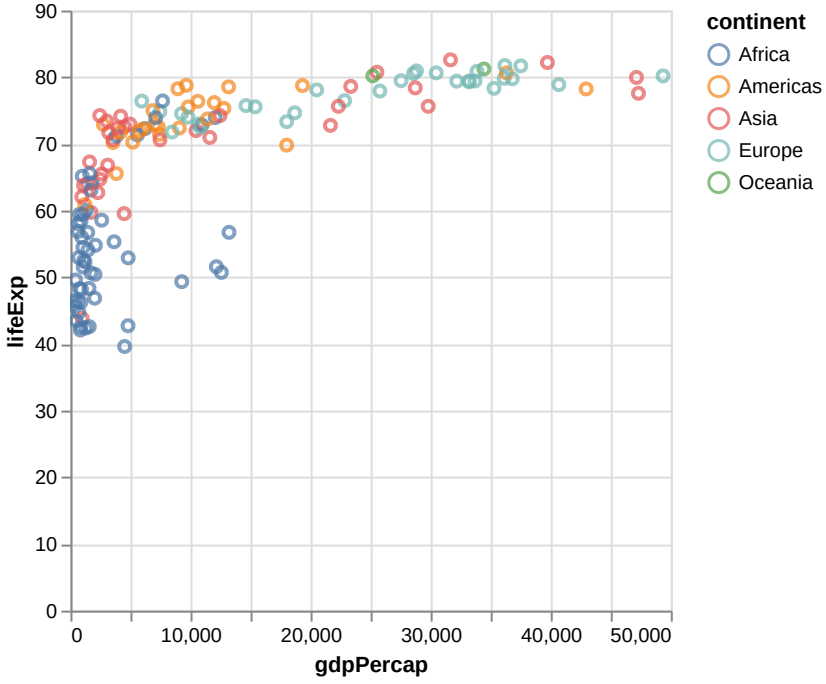
Now we only keep the year 2007.

Alternatively, we could have filtered the data before plotting using pandas.

Using color as additional channel

A very neat feature of Vega-Altair is that it is easy to add and modify visual channels. Let us try to add one more so that we do something with the "continent" data column:

```
alt.Chart(data).mark_point().encode(
    x="gdpPercap",
    y="lifeExp",
    color="continent",
).transform_filter(alt.datum.year == 2007).interactive()
```

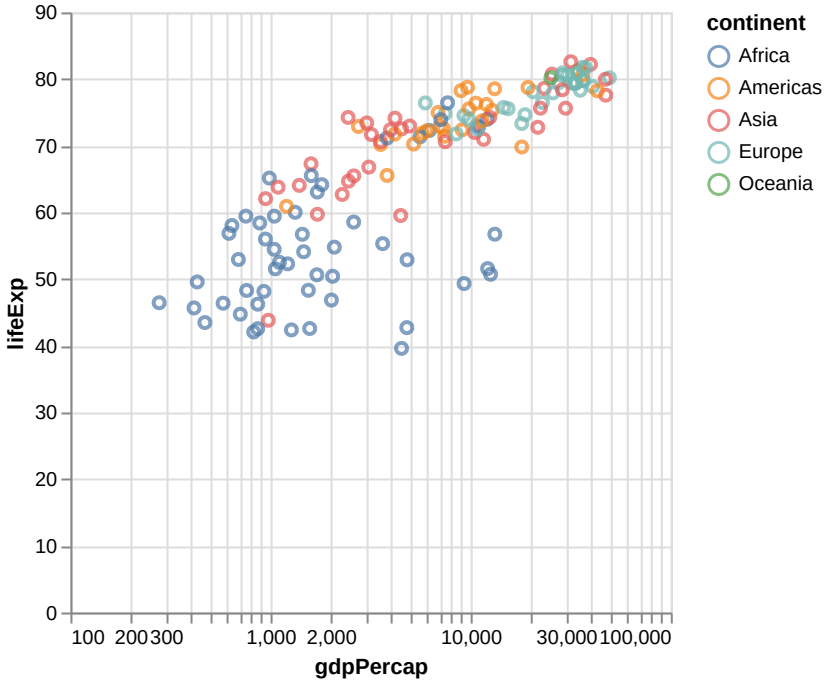


Using different colors for different continents.

Changing to log scale

For this data set we will get a better insight when switching the x-axis from linear to log scale:

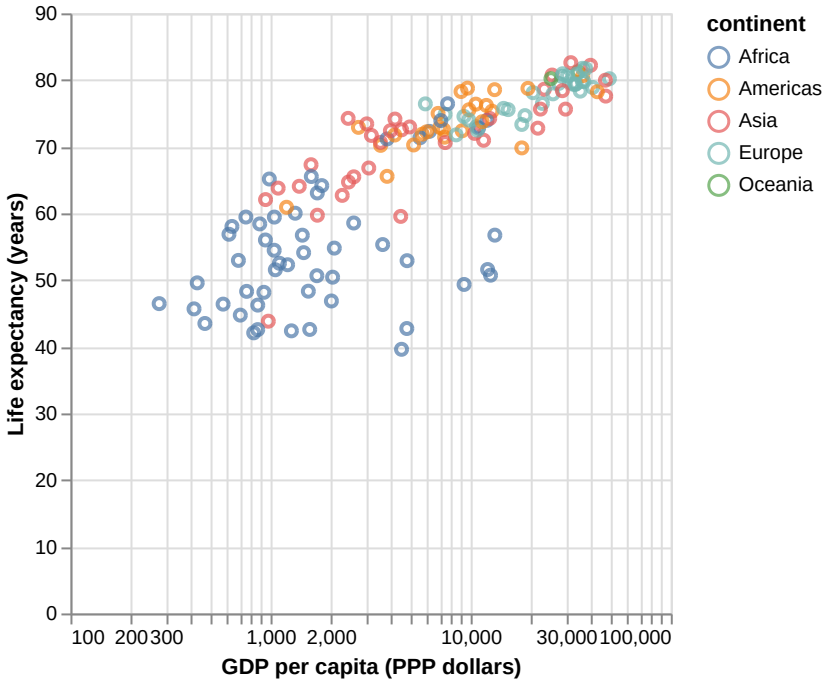
```
alt.Chart(data).mark_point().encode(
    x=alt.X("gdpPercap").scale(type="log"),
    y=alt.Y("lifeExp"),
    color="continent",
).transform_filter(alt.datum.year == 2007).interactive()
```



Changing the x axis to log scale.

Improving axis titles

```
alt.Chart(data).mark_point().encode(
    x=alt.X("gdpPercap").scale(type="log").title("GDP per capita (PPP dollars)"),
    y=alt.Y("lifeExp").title("Life expectancy (years)"),
    color="continent",
).transform_filter(alt.datum.year == 2007).interactive()
```



Improving the axis titles.

Faceted charts

To see what faceted charts are and how easy it is to do this, add the following line:

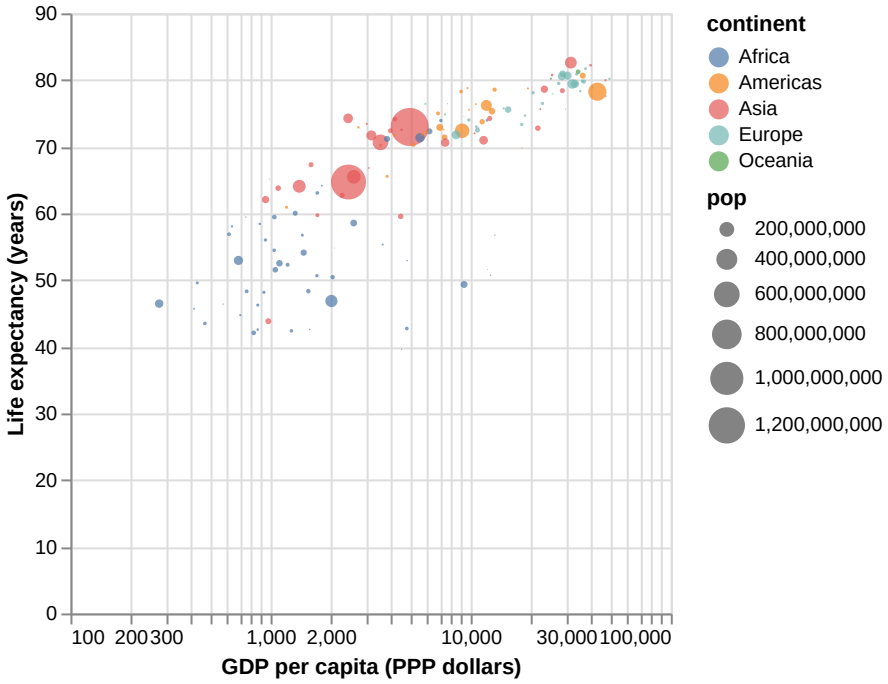
```
alt.Chart(data).mark_point().encode(
    x=alt.X("gdpPercap").scale(type="log").title("GDP per capita (PPP dollars)"),
    y=alt.Y("lifeExp").title("Life expectancy (years)"),
    color="continent",
    row="continent",
).transform_filter(alt.datum.year == 2007).interactive()
```

Guess what happens when you change `row="continent"` to `column="continent"`?

Changing from points to circles

Let us add one more visual channel, mapping size of the circle to the population size of a country:

```
alt.Chart(data).mark_circle().encode(
    x=alt.X("gdpPercap").scale(type="log").title("GDP per capita (PPP dollars)"),
    y=alt.Y("lifeExp").title("Life expectancy (years)"),
    color="continent",
    size="pop",
).transform_filter(alt.datum.year == 2007).interactive()
```



Circle sizes are proportional to population sizes.

Title and axis values

In the next step we modify a number of things:

- We go back to the version where all circles have the same size
- Add figure title
- Modify axis domains to “zoom into” the interesting part of the plot
- Set axis values
- Change from `mark_point()` to `mark_circle()`
- Invoke `interactive()` in a separate step

```
chart = (
    alt.Chart(
        data,
        title=alt.Title("Life expectancy as function of the gross domestic product"),
    )
    .mark_circle()
    .encode(
        x=alt.X("gdpPercap", axis=alt.Axis(values=[100, 1000, 10000, 100000]))
        .scale(type="log", domain=(200, 100000))
        .title("GDP per capita (PPP dollars)"),
        y=alt.Y("lifeExp", axis=alt.Axis(values=[20, 30, 40, 50, 60, 70, 80]))
        .title("Life expectancy (years)")
        .scale(domain=(20, 85)),
        color="continent",
    )
    .transform_filter(alt.datum.year == 2007)
)

chart.interactive()
```

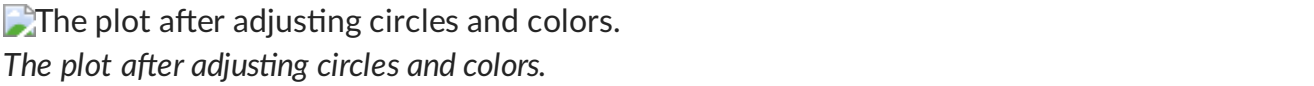
Plot after adding a figure title.

The plot is starting to look better!

Colors

In the next step we change the color scheme (list of all schemes), make the circles larger and slightly transparent:

```
chart = (
    alt.Chart(
        data,
        title=alt.Title("Life expectancy as function of the gross domestic product"),
    )
    .mark_circle(opacity=0.8, size=100.0)
    .encode(
        x=alt.X("gdpPercap", axis=alt.Axis(values=[100, 1000, 10000, 100000]))
        .scale(type="log", domain=(200, 100000))
        .title("GDP per capita (PPP dollars)"),
        y=alt.Y("lifeExp", axis=alt.Axis(values=[20, 30, 40, 50, 60, 70, 80]))
        .title("Life expectancy (years)")
        .scale(domain=(20, 85)),
        color=alt.Color("continent").scale(scheme="dark2"),
    )
    .transform_filter(alt.datum.year == 2007)
)

chart.interactive()
```


The plot after adjusting circles and colors.
The plot after adjusting circles and colors.

We can also define own colors:

```
okabe_ito = [
    "#0072b2",
    "#e69f00",
    "#cc79a7",
    "#009e73",
    "#f0e442",
    "#000000",
    "#d55e00",
    "#56b4e9",
]
```

```
chart = (
    alt.Chart(
        data,
        title=alt.Title("Life expectancy as function of the gross domestic product"),
    )
    .mark_circle(opacity=0.8, size=100.0)
    .encode(
        x=alt.X("gdpPercap", axis=alt.Axis(values=[100, 1000, 10000, 100000]))
        .scale(type="log", domain=(200, 100000))
        .title("GDP per capita (PPP dollars)"),
        y=alt.Y("lifeExp", axis=alt.Axis(values=[20, 30, 40, 50, 60, 70, 80]))
        .title("Life expectancy (years)")
        .scale(domain=(20, 85)),
        color=alt.Color("continent").scale(range=okabe_ito),
    )
    .transform_filter(alt.datum.year == 2007)
)

chart.interactive()
```


Adjusting colors to those recommended by Okabe and Ito.

💬 Why these colors?

This qualitative color palette is optimized for all color-vision deficiencies, see https://clauswilke.com/dataviz/color-pitfalls.html and Okabe, M., and K. Ito. 2008. “Color Universal Design (CUD): How to Make Figures and Presentations That Are Friendly to Colorblind People.”.

More tweaking towards a publication-ready figure

Let us add a subtitle and adjust sizing and positioning:

```
chart = (
    alt.Chart(
        data,
        title=alt.Title(
            "Life expectancy as function of the gross domestic product",
            subtitle=[
                "Gross domestic product (GDP) per capita measures the value of
everything",
                "produced in a country during a year, divided by the number of
people.",
                "The unit is in purchasing power parities (PPP dollars), fixed to 2017
prices.",
                "Data is adjusted for inflation and differences in the cost of living
between countries.",
            ],
        ),
    )
    .mark_circle(opacity=0.8, size=100.0)
    .encode(
        x=alt.X("gdpPercap", axis=alt.Axis(values=[100, 1000, 10000, 100000]))
        .scale(type="log", domain=(200, 100000))
        .title("GDP per capita (PPP dollars)"),
        y=alt.Y("lifeExp", axis=alt.Axis(values=[20, 30, 40, 50, 60, 70, 80]))
        .title("Life expectancy (years)")
        .scale(domain=(20, 85)),
        color=alt.Color("continent").scale(range=okabe_ito),
    )
    .transform_filter(alt.datum.year == 2007)
)

chart = chart.configure_axis(labelFontSize=20, titleFontSize=20)

chart = chart.properties(width=600, height=500)

chart = chart.configure_title(
    fontSize=20,
    subtitleFontSize=20,
    anchor="start",
    orient="bottom",
    offset=20,
    subtitleColor="gray",
)

chart = chart.configure_legend(
    titleFontSize=20,
    labelFontSize=20,
    padding=10,
)

chart.interactive()
```

 More tweaking towards a publication-ready figure.

Interactive charts

With not too many changes we can make the chart interactive and add a slider for the year (please try this in this notebook):

```
year_slider = alt.binding_range(min=1952, max=2007, step=5, name="Year")
slider_selection = alt.selection_point(bind=year_slider, fields=["year"], value=2007)

chart = (
    alt.Chart(
        data,
        title=alt.Title(
            "How life expectancy and gross domestic product evolved over time",
            subtitle=[
                "Gross domestic product (GDP) per capita measures the value of
everything",
                "produced in a country during a year, divided by the number of
people.",
                "The unit is in purchasing power parities (PPP dollars), fixed to 2017
prices.",
                "Data is adjusted for inflation and differences in the cost of living
between countries.",
            ],
        ),
    )
    .mark_circle(opacity=0.8, size=100.0)
    .encode(
        x=alt.X("gdpPercap", axis=alt.Axis(values=[100, 1000, 10000, 100000]))
        .scale(type="log", domain=(200, 100000))
        .title("GDP per capita (PPP dollars)"),
        y=alt.Y("lifeExp", axis=alt.Axis(values=[20, 30, 40, 50, 60, 70, 80]))
        .title("Life expectancy (years)")
        .scale(domain=(20, 85)),
        color=alt.Color("continent").scale(range=okabe_ito),
    )
    .add_params(slider_selection)
    .transform_filter(slider_selection)
)

chart = chart.configure_axis(labelFontSize=20, titleFontSize=20)

chart = chart.properties(width=600, height=500)

chart = chart.configure_title(
    fontSize=20,
    subtitleFontSize=20,
    anchor="start",
    orient="bottom",
    offset=20,
    subtitleColor="gray",
)

chart = chart.configure_legend(
    titleFontSize=20,
    labelFontSize=20,
    padding=10,
)

chart.interactive()
```

Adding more annotation

With few more lines we can add extra annotation that can help to highlight certain aspects of the plot and to tell a story:

```
year_slider = alt.binding_range(min=1952, max=2007, step=5, name="Year")
slider_selection = alt.selection_point(bind=year_slider, fields=["year"], value=2007)

chart = (
    alt.Chart(
        data,
        title=alt.Title(
            "How life expectancy and gross domestic product evolved over time",
            subtitle=[
                "Gross domestic product (GDP) per capita measures the value of
everything",
                "produced in a country during a year, divided by the number of
people.",
                "The unit is in purchasing power parities (PPP dollars), fixed to 2017
prices.",
                "Data is adjusted for inflation and differences in the cost of living
between countries.",
            ],
        ),
    )
    .mark_circle(opacity=0.8, size=100.0)
    .encode(
        x=alt.X("gdpPercap", axis=alt.Axis(values=[100, 1000, 10000, 100000]))
        .scale(type="log", domain=(200, 100000))
        .title("GDP per capita (PPP dollars)"),
        y=alt.Y("lifeExp", axis=alt.Axis(values=[20, 30, 40, 50, 60, 70, 80]))
        .title("Life expectancy (years)")
        .scale(domain=(20, 85)),
        color=alt.Color("continent").scale(range=okabe_ito),
    )
    .add_params(slider_selection)
    .transform_filter(slider_selection)
)

annotation = (
    alt.Chart(data)
    .encode(
        x="gdpPercap",
        y="lifeExp",
        text="country",
        color=alt.value("black"),
    )
    .transform_filter((slider_selection) & (alt.datum.country == "Norway"))
)

chart = (
    chart
    + annotation.mark_point(size=100.0)
    + annotation.mark_text(size=15, xOffset=10, align="left", baseline="middle")
)

chart = chart.configure_axis(labelFontSize=20, titleFontSize=20)

chart = chart.properties(width=600, height=500)

chart = chart.configure_title(
    fontSize=20,
    subtitleFontSize=20,
    anchor="start",
    orient="bottom",
    offset=20,
    subtitleColor="gray",
)
```

```
chart = chart.configure_legend(
    titleFontSize=20,
    labelFontSize=20,
    padding=10,
)

chart.interactive()
```

 Chart with extra annotation.

Saving the chart as web page

You can save the chart as a web site and try to open it in a separate browser tab and put it on your home page or research group website:

```
chart.save("chart.html")
```

TODO:

Link to reproducibility How you might want to use by yourself CodeRefinery workshop

This serves as an introduction to Git. After this lesson, you will both be able to use Git, and feel much more confident taking a command-line course Git course such as CodeRefinery's git-intro course. Check out our other upcoming workshops.

Reading:

- Quickly get started with common tasks on GitHub

Making your project citable

💬 Discussion: is depositing your data/code on GitHub enough?

- Consider the aspect of findability 5 or 10 years from now.
- What could go wrong?

There are many services where you can share or archive your code and data: See for instance our lesson on reproducible research.

In this present lesson we will discuss one of the many options to get a **digital object identifier (DOI)** for your dataset or code: Zenodo, A general-purpose open access repository created by OpenAIRE and CERN. Zenodo has nice **integration with GitHub**, and allows researchers to upload files up to 50 GB.

💬 We will exercise in the Zenodo sandbox

We will practice on https://sandbox.zenodo.org/ and not on the “real” https://zenodo.org/ to make sure we do not create “real” DOIs which we cannot remove.

The sandbox service is useful to calibrate your setup until you are happy with the result and then you can go for the real service. Once a dataset is uploaded to the “real” service, it cannot be easily removed or modified again (and this is good, otherwise DOIs would not make much sense).

Step 1: Prepare an example repository

Through web:

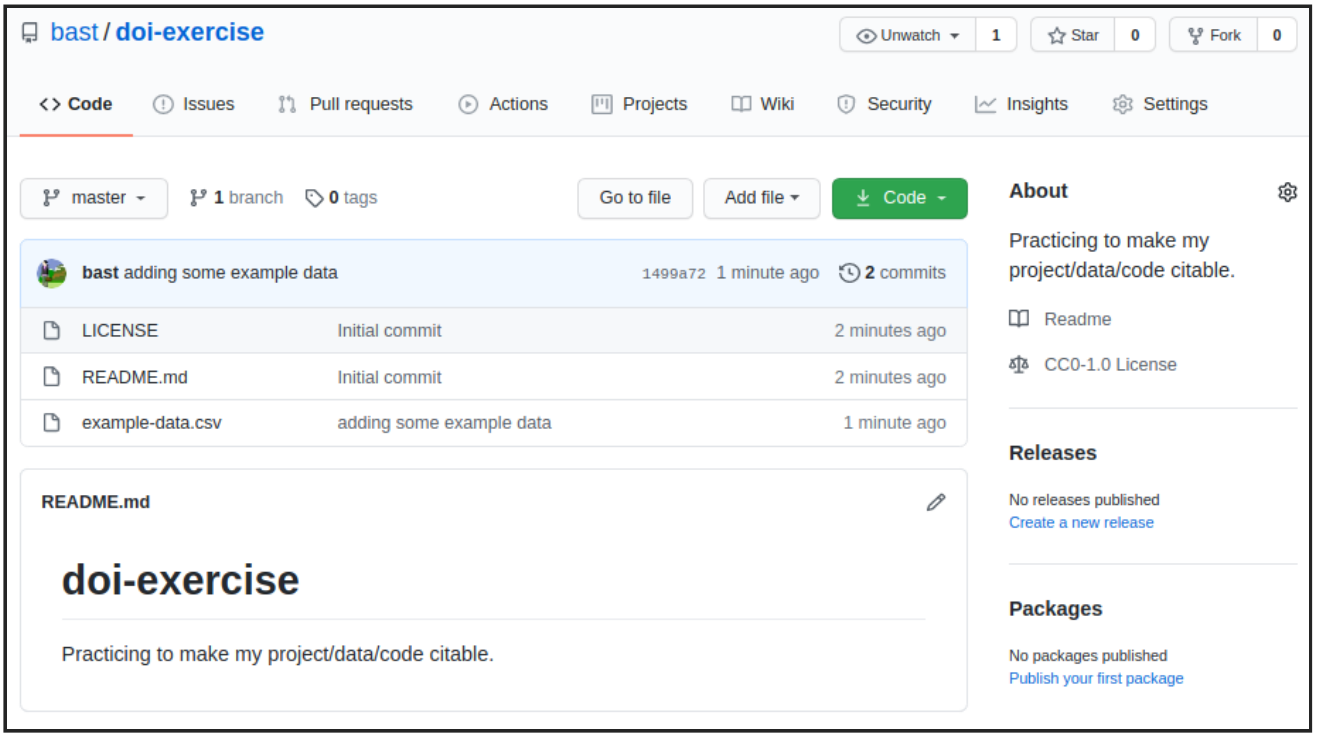
- Create a new repository on GitHub
- Upload some example data into it

Or using GitHub Desktop:

- Create a new repository
- Commit some example data
- Publish your example repository to GitHub

Alternatively we can also practice this with one of the repositories we created earlier in this lesson.

Here I just created a new one:

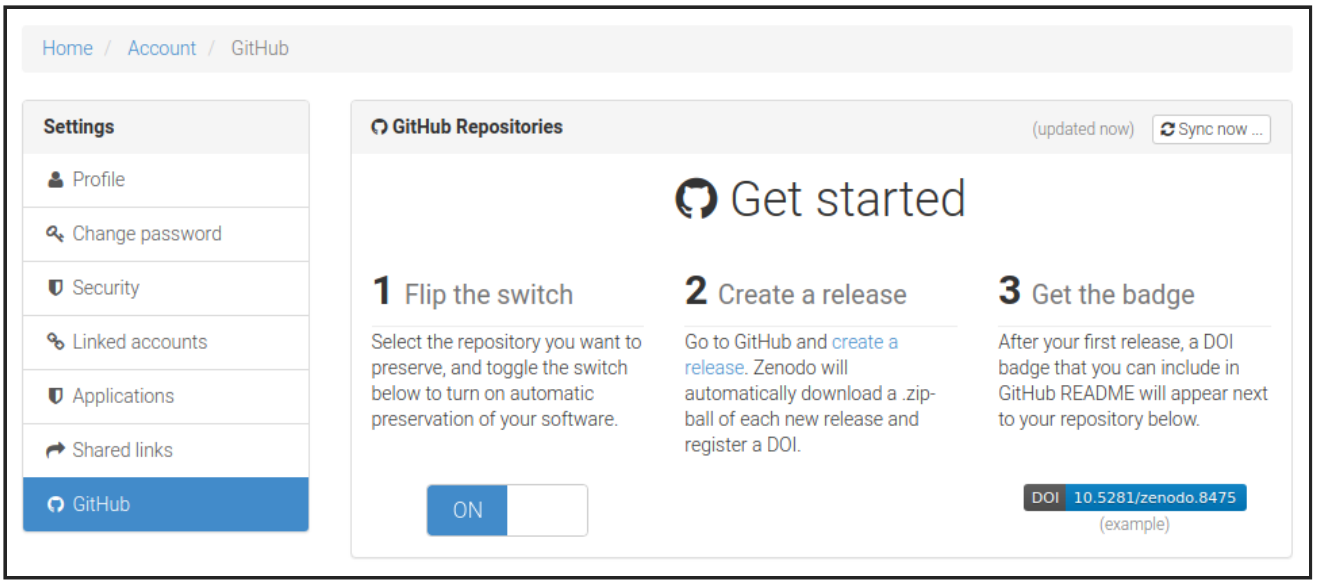


Step 2: Activate the repository on Zenodo (sandbox)

💬 We will exercise in the Zenodo sandbox

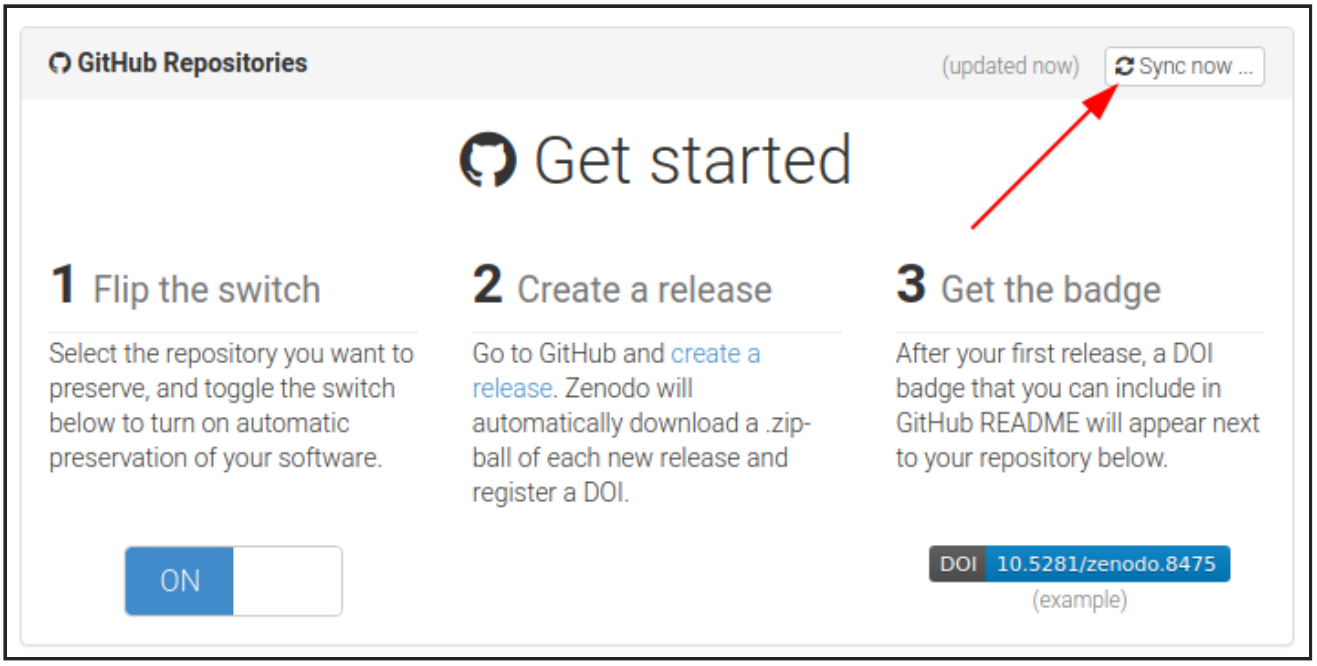
We will practice on https://sandbox.zenodo.org/ and not on the "real" https://zenodo.org/ to make sure we do not create "real" DOIs which we cannot remove.

- Visit https://sandbox.zenodo.org/account/settings/github/:



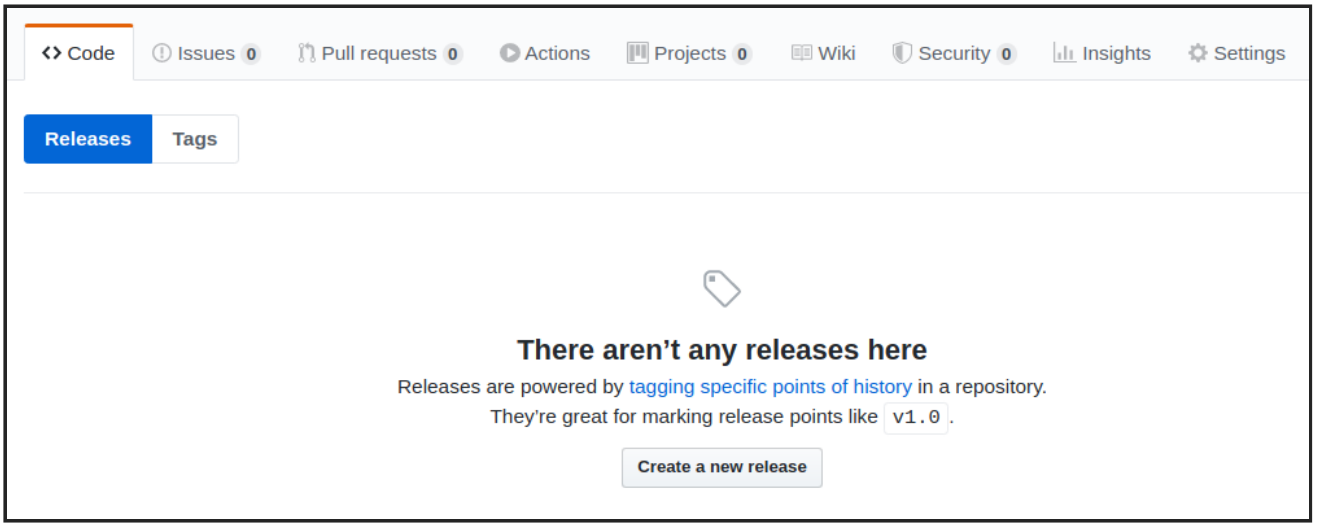
- Select the repository you wish to preserve:

- If it is not there, you may need to "Sync now ...":

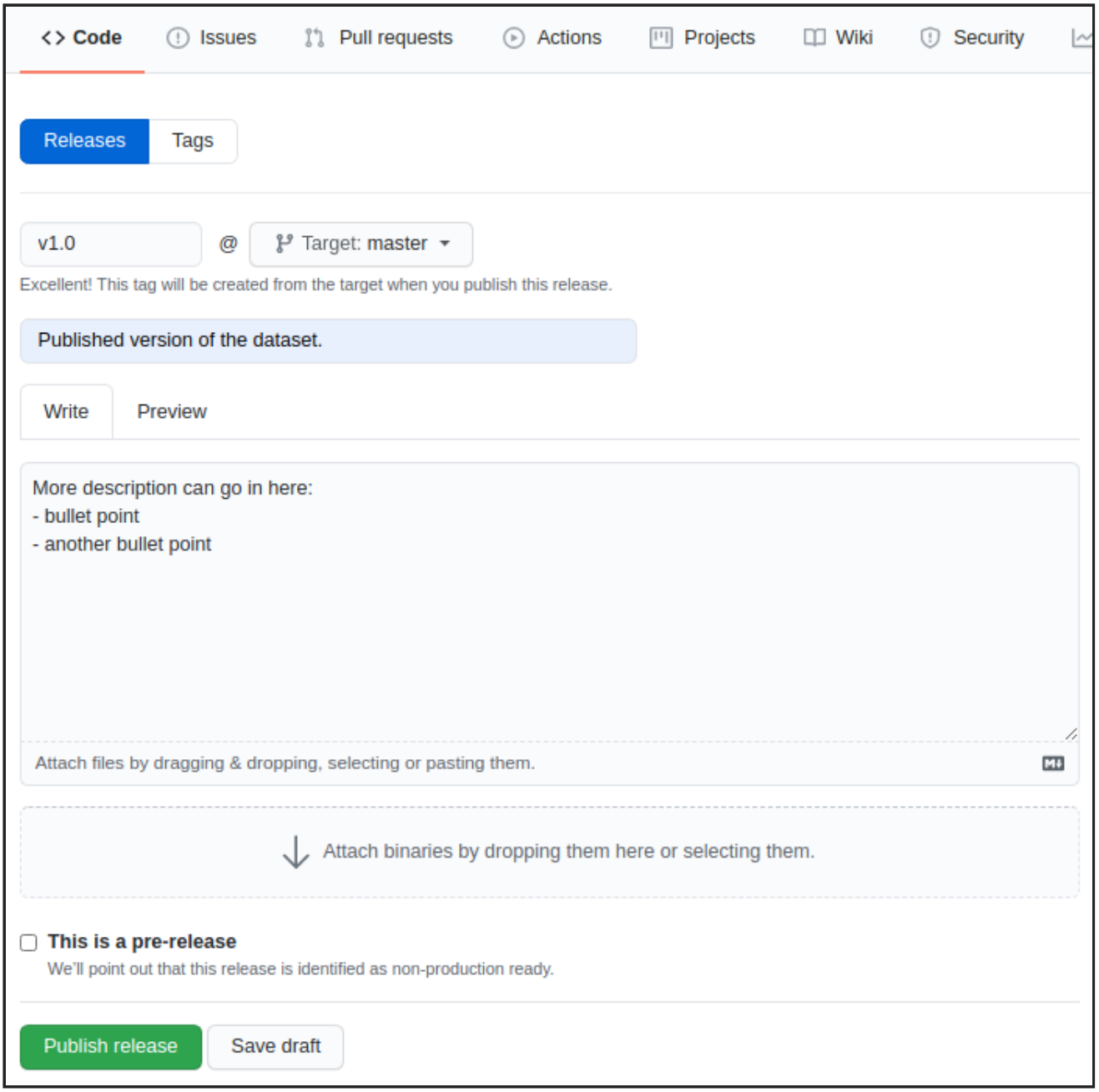


Step 3: Create a "release" and get a DOI

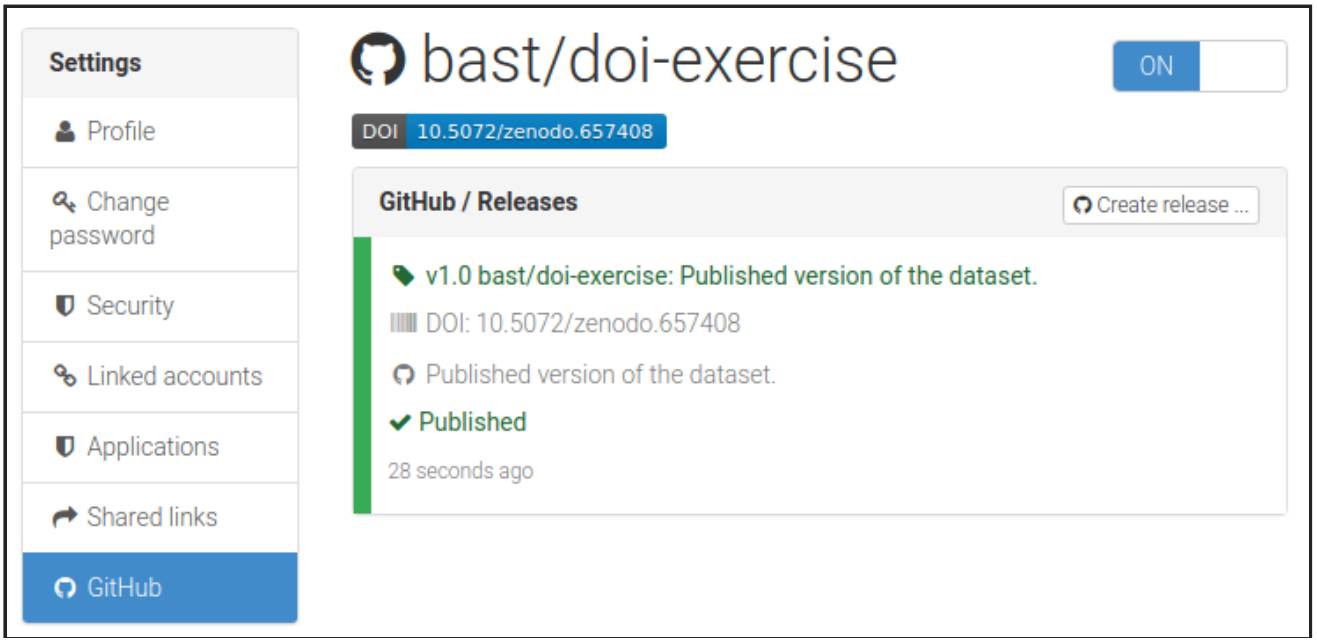
- Go back to the webpage of your GitHub repository
- Click on "releases":



- Fill out the form and click on "Publish release":



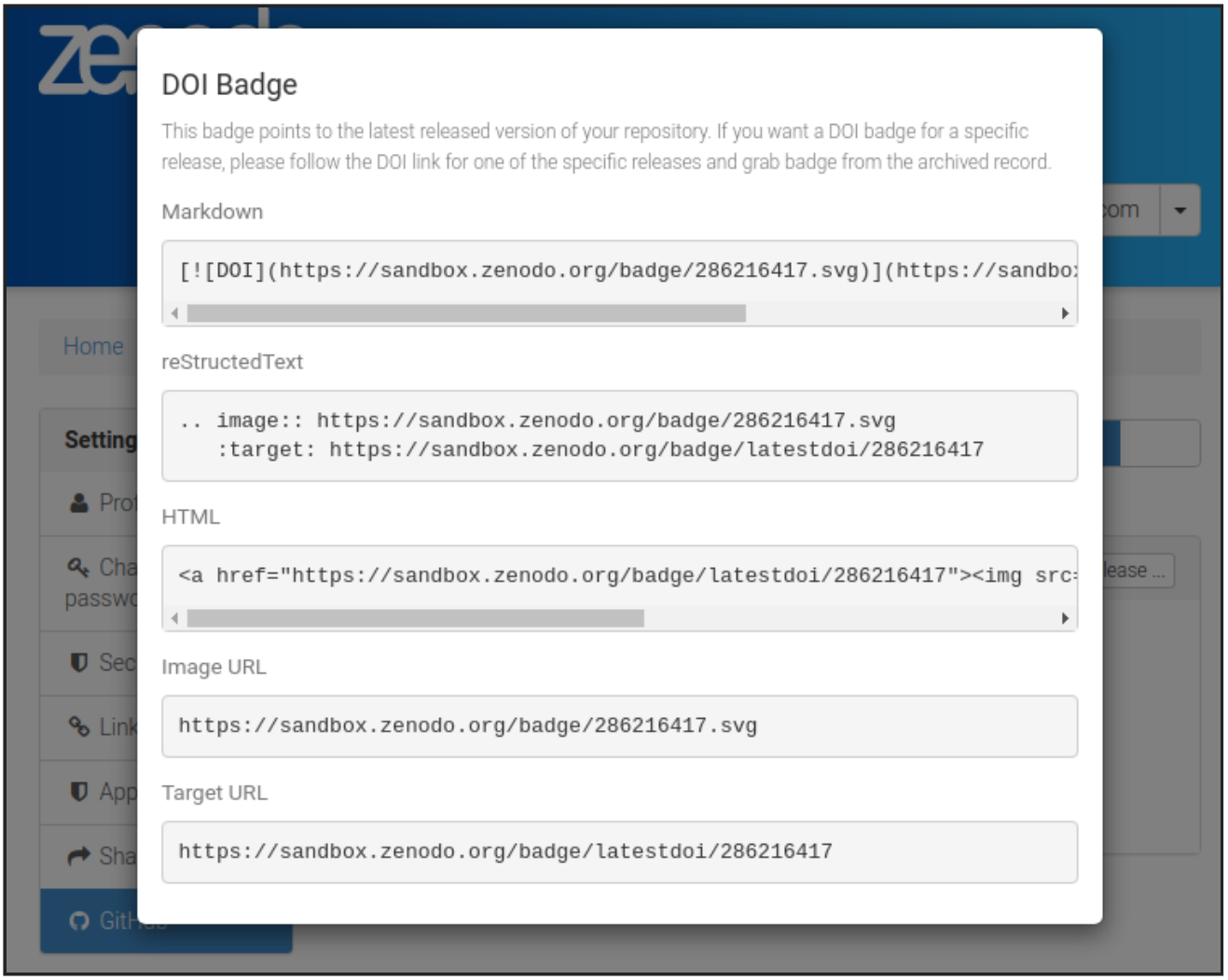
- Finally reload https://sandbox.zenodo.org/account/settings/github/
- It can take few moments for the project to be deposited
- After a while it turns to this:



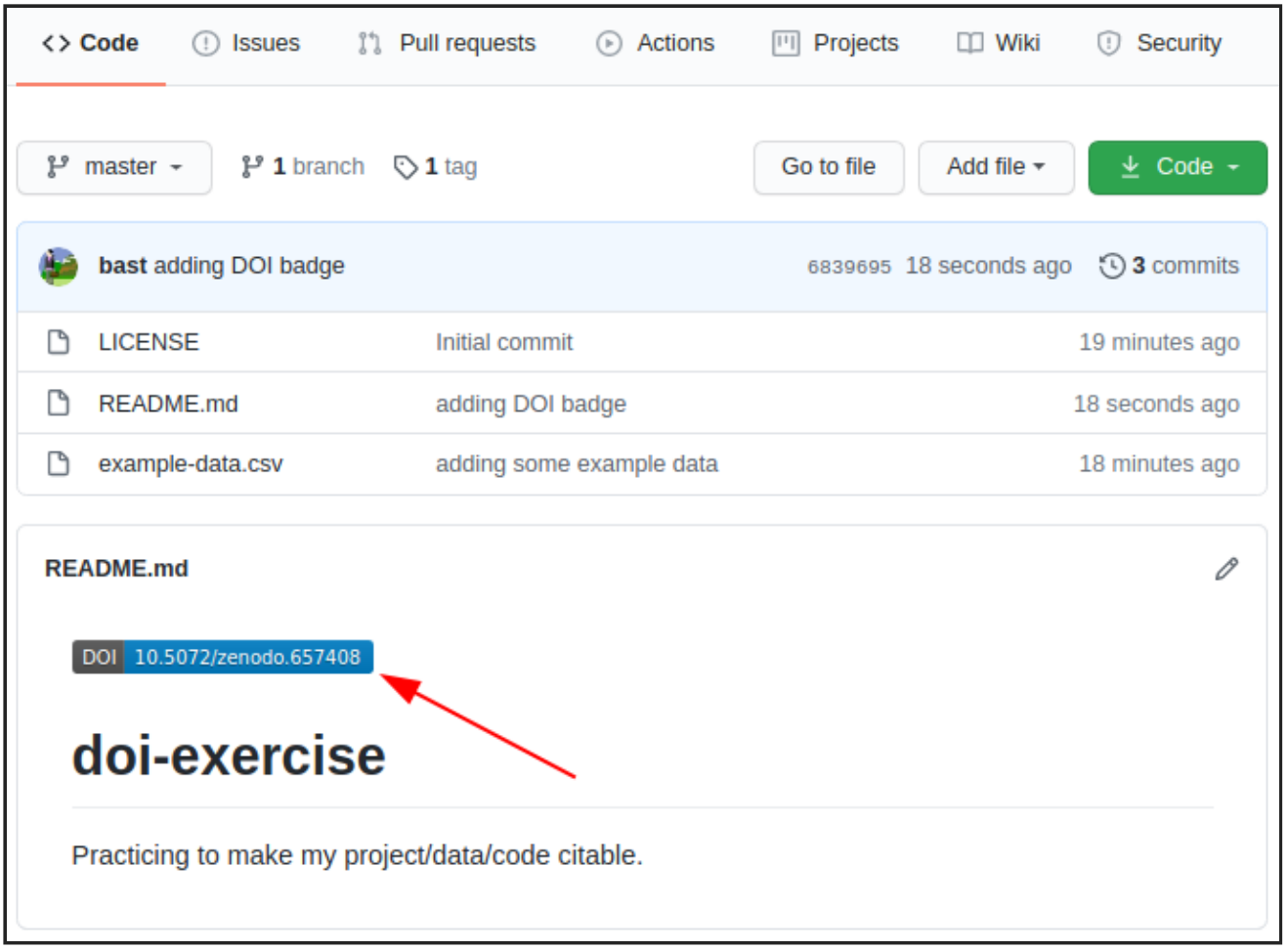
Step 4: Add a DOI badge to your repository

This is bonus but for visitors of your GiHub repository it can be nice to find a badge in your README that informs them about and links to the preserved dataset/code on Zenodo.

- On Zenodo, click on the badge (last figure) which opens up:



- Try to add such a badge to your README for the exercise repository:



Hosting websites/homepages on GitHub Pages

You can host your personal homepage or group webpage or project website on GitHub using GitHub Pages.

GitLab and Bitbucket also offer a very similar solution.

Unless you need user authentication or a sophisticated database behind your website, GitHub Pages can be a very nice alternative to running your own web servers.

This is how all https://coderefinery.org material is hosted.

Exercise

- Deploy own website reusing a template
- Make a change to the website after it has been deployed for the first time
- Verify that the change shows up on the website a minute or two later

The documentation for GitHub Pages is very good so no need for us to duplicate screenshots: https://pages.github.com/

💬 Discussion

- You can use HTML directly or another static site generator if you prefer to not use the default Jekyll.

- It is no problem to use a custom domain instead of `*.github.io`.

Sharing plots and 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.

In Python, it is customary to do this either in a `requirements.txt` file (example):

```
jupyterlab
altair == 5.5.0
vega_datasets
pandas == 2.2.3
numpy == 2.1.2
```

... or in an `environment.yml` file (example):

```yaml
name: data-viz
channels:
  - conda-forge
dependencies:
  - python <= 3.12
  - jupyterlab
  - altair-all = 5.5.0
  - vega_datasets
  - pandas = 2.2.3
  - numpy = 2.1.2
```

By the way, this is almost the same `environment.yml` file that we used to install the local software environment in the Software install instructions (the latter did not pin versions).

Place either `requirements.txt` or `environment.yml` in the same folder as the notebook(s).

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 Vega-Altair plot

- Save it in SVG format (vector graphics, “maximum resolution”)
- Save it in PNG format (raster graphics)
- Share it as notebook (more about it below)
- Save it a web page with `chart.save("chart.html")` and share the HTML file
- You can also get a shareable URL to a chart (example)
- With **sensitive data**, you need to be careful with sharing (see next section)

Vega-Altair and notebooks containing sensitive data

If you plot **sensitive data** in a notebook with Vega-Altair, you need to be careful.

The author of Vega-Altair provided a good summary in this GitHub comment:

> “Standard Altair rendering requires the entire dataset to be accessible to the viewer’s browser: this is a fundamental design decision in Vega/Vega-Lite, in which a chart is equivalent to a dataset plus a specification of how to render it. In general, you should assume that the entire contents of any dataframe you pass to the alt.Chart() object will be saved in the notebook and be inspectable by the viewer.”
>
> “One way to get around this would be to render the chart server-side, export a PNG, and display this png instead of the live chart. Incidentally, in the Jupyter notebook you can do this by running:”
>
> ```
> alt.renderers.enable('png')
> ```
>
> “This sets up Altair such that charts will be rendered to PNG within the kernel, and only that PNG rendering will be embedded in the notebook. Note this requires some extra dependencies, described here.”
>
> “But even here, I wouldn’t call your data “private” (for example, if you save a scatter plot to PNG, a user can straightforwardly read the data values off the chart!) So this makes me think you’re actually doing some sort of aggregation of your data before plotting (e.g. showing a histogram). If this is the case, I would suggest doing those aggregations outside of Altair using e.g. pandas, and then passing the aggregated dataset to the chart. Then you get the normal interactive display of the Altair chart, and your data is just as private as it would have been in the equivalent static rendering – the user can only see the aggregated values you supplied to the chart.”

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 using Binder

✍ Exercise/demo: Making your notebooks reproducible by anyone (15 min)

Instructor demonstrates this:

- Instructor creates a GitHub repository.
- Uploads a notebook file that we created in earlier episodes.
- Then we look at the statically rendered version of the notebook on GitHub and also nbviewer.
- Add a file `requirements.txt` which contains:

```
altair == 5.5.0
vega_datasets
pandas == 2.2.3
numpy == 2.1.2
```

- Visit https://mybinder.org:

- 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 explored and executed in the cloud.
- Enjoy being fully reproducible!

Also please see how we share the notebooks from this lesson in the episode-overview.

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.

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:

```yaml
name: data-viz
channels:
  - conda-forge
dependencies:
  - python <= 3.12
  - jupyterlab
  - altair-all
  - vega_datasets
  - pandas
  - numpy
```

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

If you are new to Python or unsure how to create isolated environments in Python from files like the `environment.yml` 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.

Installing Python via Miniforge

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

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 `data-viz`.

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 data-viz -f
https://raw.githubusercontent.com/coderefinery/data-visualization-
python/main/software/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 `data-viz` environment.

Windows | Linux / MacOS

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

```
$ conda activate data-viz
$ jupyter-lab
```

Removing the software environment

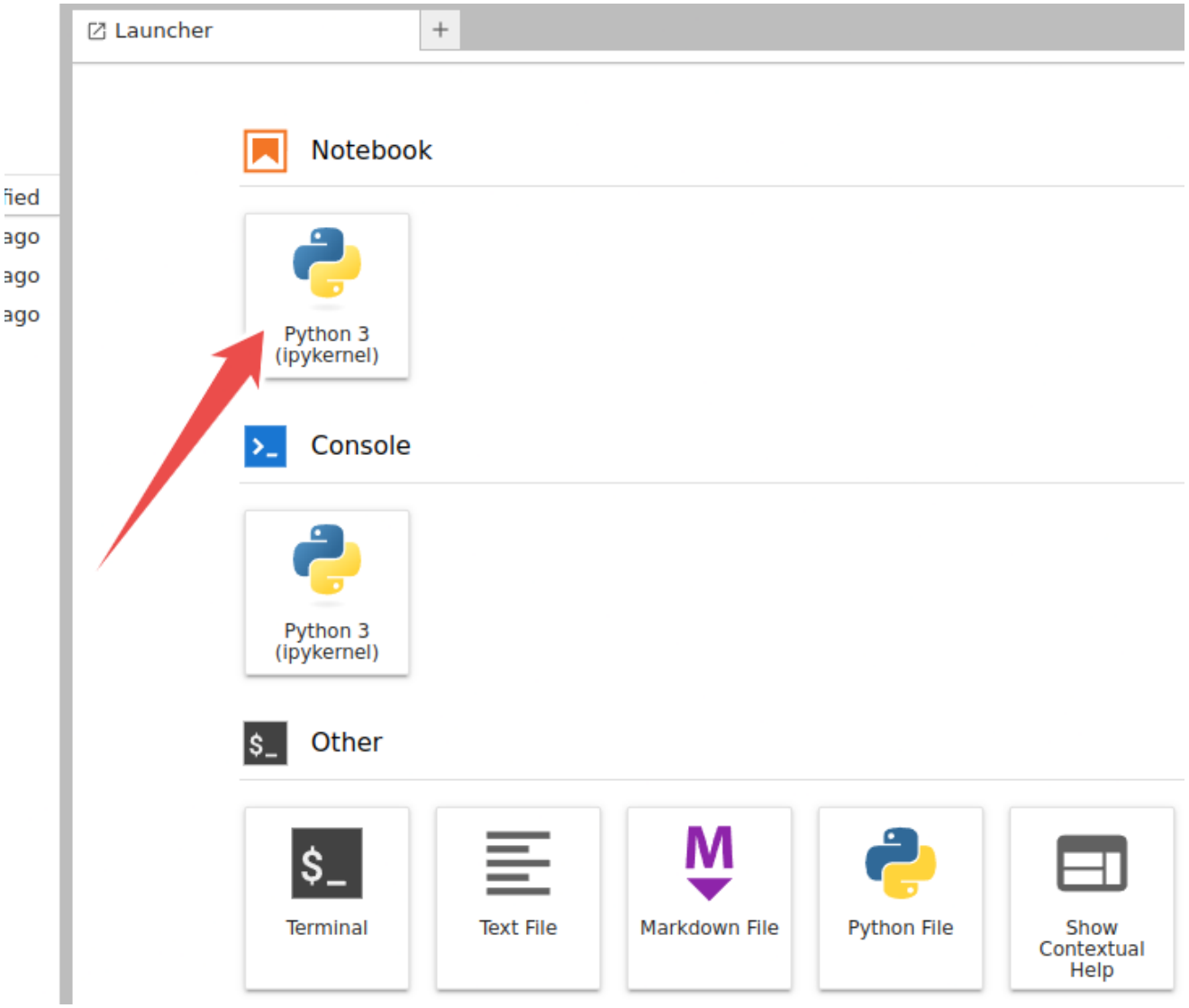
Windows | Linux / MacOS

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

```
$ conda env list
$ conda env remove --name data-viz
$ conda env list
```

How to verify your installation

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



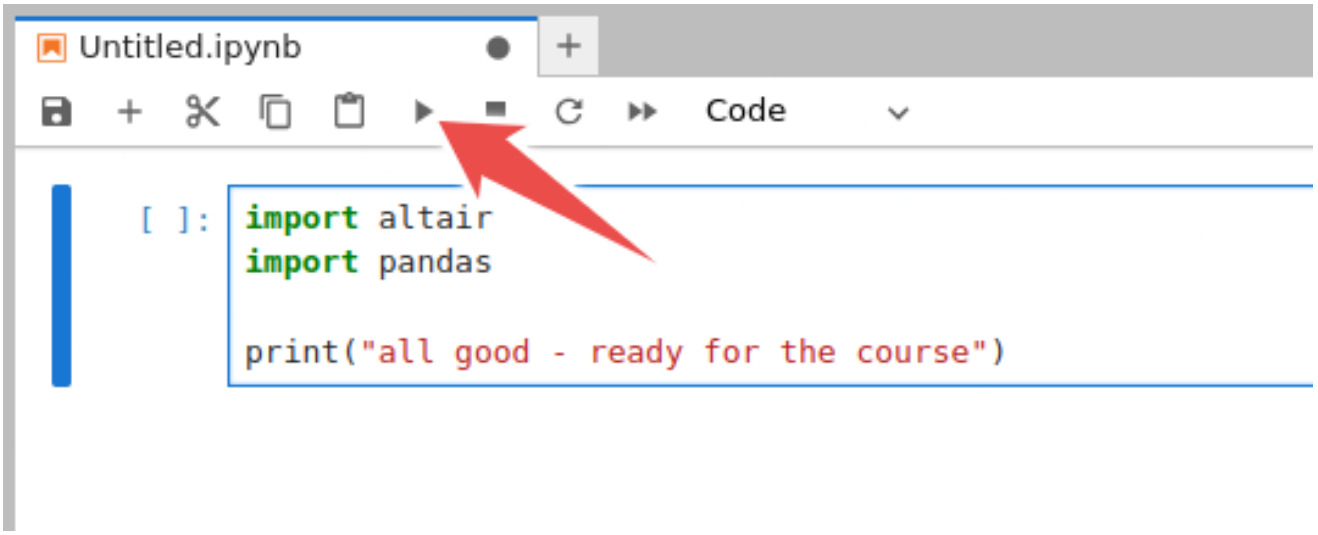
JupyterLab opened in the browser. Click on the Python 3 tile.

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

Python 3 notebook started.

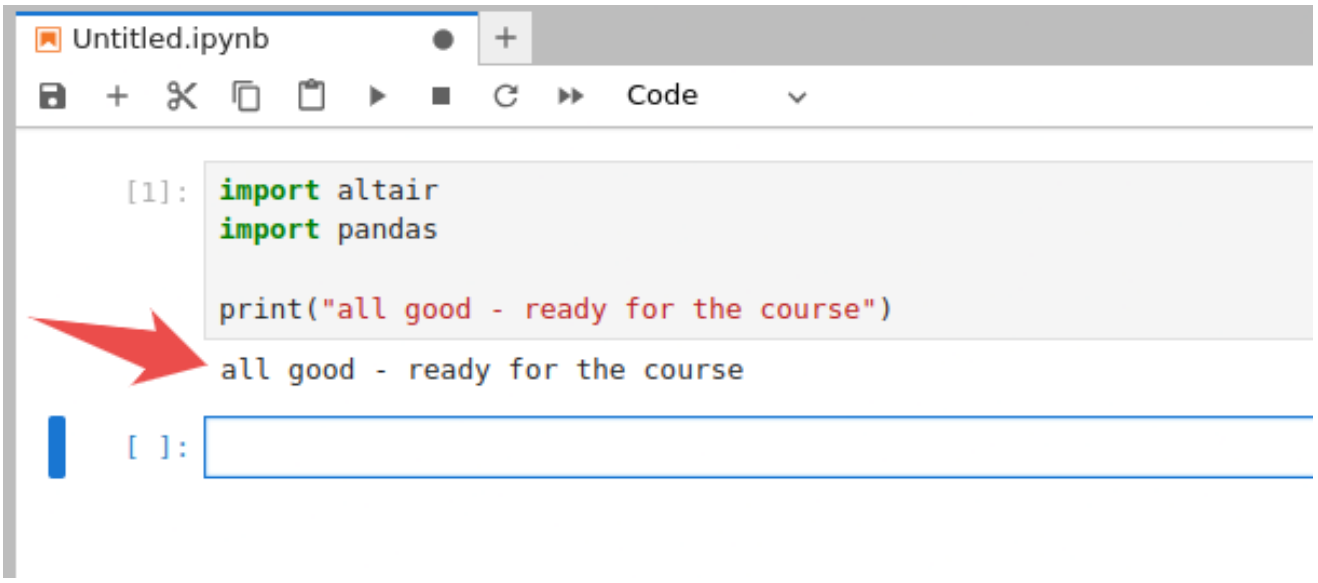
Into that blue “cell” please type the following:

```
import altair
import pandas

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



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

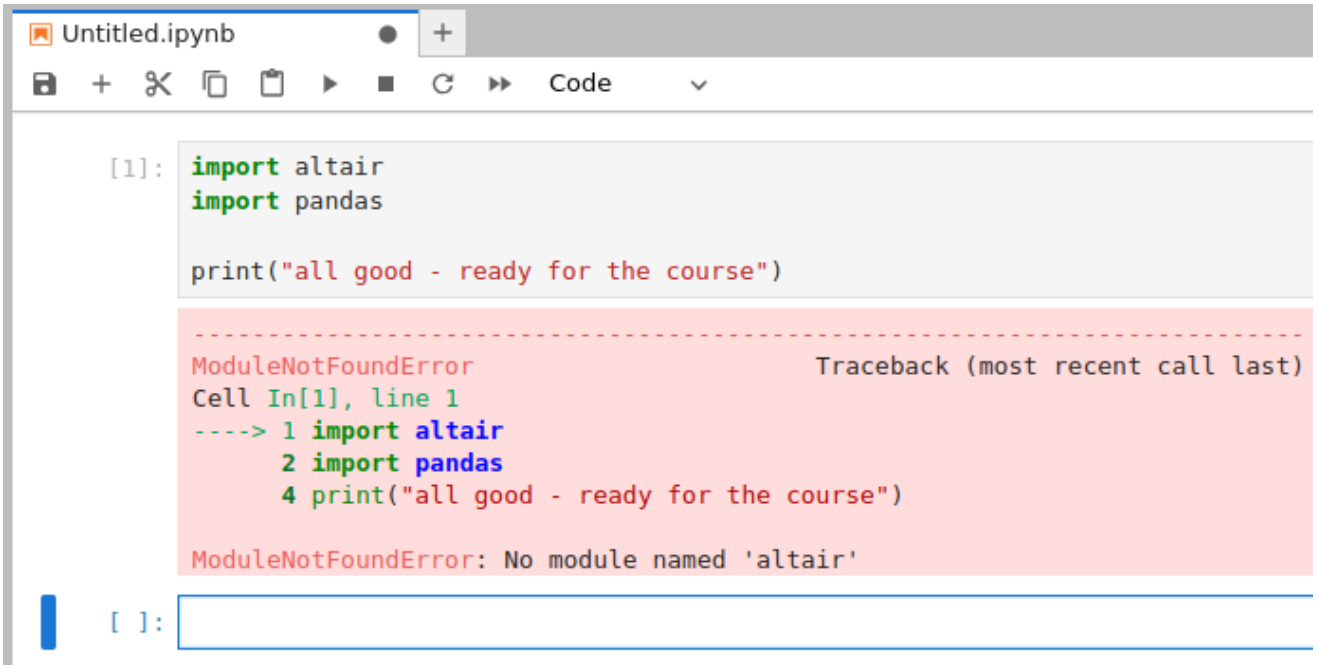
This is how it should look:



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.

Inspiration for this lesson and further reading

- TODO: Link to CodeRefinery lesson DOIs
- TODO: Link to Skills4EOSC curriculum link