Modular code development - Making reusing parts of your code easier

Type-along/demo where we discuss and experience aspects of (un)modular code development. **We will focus on the "why", not on the "how".**

Image to get started: The curse of bad code design (Ten simple rules for quick and dirty scientific programming. PLoS Comput Biol 17(3): e1008549. https://doi.org/10.1371/journal.pcbi.1008549)

Timing overview

20 min	Starting with an example code in a Notebook
20 min	Exercise
10 min	Discussion
10 min	Break
50 min	Live-coding with suggestions via notes (One possible solution)
10 min	Break
15 min	Discussion

Starting with an example code in a Notebook

Data

The file weather_data.csv (raw csv file) contains hourly weather measurements from the observation station "Vantaa Helsinki-Vantaan lentoasema" (Helsinki airport) during 2024.

Origin of the data

Data obtained from https://en.ilmatieteenlaitos.fi/download-observations#!/ on 2025-09-18.

Data has been provided by the Finnish Meteorological Institute under the Creative Commons Attribution 4.0 International license (CC BY 4.0): https://en.ilmatieteenlaitos.fi/open-data-licence

Our initial goal

Our initial goal for this exercise is to plot a series of temperatures and precipitations for **January** and to compute and plot the **mean temperature** averaged over the month. We imagine that we assemble a working script from various internet research/ AI chat recommendations and arrive at:

Python R

```python
import pandas as pd
import matplotlib.pyplot as plt


# read data
data = pd.read_csv("weather_data.csv")

# combine 'date' and 'time' into a single column 'recorded_at' as type datetime
data["recorded_at"] = pd.to_datetime(data["date"] + " " + data["time"])

# set 'recorded_at' as index for convenience
data = data.set_index("recorded_at")

# keep only january data using datetime period indexing
january = data.loc["2024-01"]

fig, ax = plt.subplots()

# temperature time series
ax.plot(
    january.index,
    january["air_temperature_celsius"],
    label="air temperature (C)",
    color="red",
)

ax.set_title("air temperature (C) at Helsinki airport")
ax.set_xlabel("date and time")
ax.set_ylabel("air temperature (C)")
ax.legend()
ax.grid(True)

# format x-axis for better date display
fig.autofmt_xdate()

fig.savefig("2024-01-temperature.png")
```

This example is in Python but we will try to see “through” the code and focus on the bigger picture and hopefully manage to imagine other languages in its place. For the Python experts: we will not see the most elegant Python.

Further goals

- Once we get this working for **January**, our task changes to also plot the **February** and the **March** in two additional plots.

- Later, we wish to generalize the code so that a user can compute and plot this for **any month**, **without changing the code** (with a command line interface).

How we plan to solve it

Before we attempt to do this, we discuss with workshop participants how they would tackle this problem.

Together we improve the code based on suggestions from learners towards more modularity and re-usability.

Instructor note

Participants give suggestions and ask questions via collaborative document and instructor(s) try to follow and answer. They can also roughly follow the ideas and steps in the One possible solution.

It is OK and good if mistakes happen and it is fun if the instructor(s) can convey a bit of "improv" feel to this lesson.

Learning outcomes

- Know about **pure functions** (functions without side effects, functions which given same input always return same output).
- Learn why and how to **limit side effects** of functions.
- Discuss why and how to limit side effects of data. Also discuss when mutable data may be preferable.
- The Zen of Python
- Discuss why **single-purpose functions** are often preferred over multi-purpose functions.
- **Split-apply-combine**, which lets you more easily parallelize. Make your code modular in a way that lets you split the steps and parallelize.
- Think about **global vs local** data structures. It is not easy to separate them right.
- Understand how a command line interface to a code can improve usability and also make the code more versatile (to be combined with workflow management tools).
- Connect modular code development to the remaining lessons (version control, testing, documentation, reproducibility).

Exercise

You can participate in two ways

- Either by discussing with others and writing your own thoughts via collaborative notes
- Or by coding
- Or a mix of the two that is most meaningful for you or your group

Discussion track

We share these questions in a common collaborative document:

```
A. What does "modular code development" mean for you?
B. What best practices can you recommend to arrive at well structured,
   modular code in your favourite programming language?
C. What do you know now about programming that you wish somebody told you earlier?
D. Do you design a new code project on paper before coding? Discuss pros
   and cons.
E. Do you build your code top-down (starting from the big picture) or bottom-up
   (starting from components)? Discuss pros and cons.
F. Would you prefer your code to be 2x slower if it was easier to read and understand?
```

They can be answered by individual learners but also discussed within an exercise group.

Coding track

You can practice on our exercise repository which contains:

- Data set
- Python notebook which works but is not super general

To run the notebook, you'll need the same conda environment used throughout this workshop. You can activate it and launch Jupyter Lab with these commands:

```
conda activate coderefinery
jupyter lab
```

How to contribute improvements:

- Open issue at the exercise repository and in your pull request refer to that issue.

Exercise ideas (sorted from basic to advanced):

- Improve the README
- Add example usage to the README
- Add a result image to the README
- Make it installable and document installation (requirements.txt or environment.yml, pip/conda/...)
- Improve error messages (e.g., input file does not exist or does not contain data we want to plot)
- Draw a call tree for one of your recent projects. Identify the functions in your call tree which are "pure" (which have no side-effects).

- Try out with your own data
- Contribute a notebook/script with weather data from your place **in your favorite programming language**
- Make the notebook (more) reproducible
- Add automated tests
- Add a command line interface
- If you use AI, any tricks you can share to get modular code from it?
- How would you restructure the code to make it more maintainable and collaborative?
- Add support for Snakemake or any other workflow management tool

Encapsulation

Encapsulation in software development refers to isolating a piece of software and its dependencies from the external environment so that it can function, be reused, or moved independently. It can happen at different levels — within code (e.g., functions, classes, or modules), within environments (e.g., virtual environments, containers), or even at the system level (e.g., portable workflows and data management).

💬 Types of encapsulation

Let's say you want to move a code from one system to another. What are the things that can go wrong?

- **In-language dependencies**, e.g. Python. Can they all be expressed *only* in `requirements.txt`? Do you wrap everything in a container?
- **Paths**: Can you always use relative paths?
- **Operating-system (OS) dependencies, libraries, etc.**: Can you eliminate them?
- **Data files**: Are they controlled or few that you can migrate, or if you wanted to move it, are you forced to copy the whole directory without regard to what the file is (thus creating a lot of duplicates, and a big mess?)
- Are you using something that is **OS-specific** (GNU/Linux vs BSD)?
- Do you use support programs only available on certain computers? Fewer external utilities you use = easier portability.

💬 What needs to be global vs what needs to be local?

- Global data can be "seen"/accessed in the entire code.
- Local data is only available in the local vicinity of its definition.
- Try to have as little global data as possible.
- Global data are often input parameters, configuration parameters, command-line arguments.
- But try to localize these to the "main" code/function.

One possible solution

Before we start

We **don't have to follow this line by line** but it's important to study this example well before demonstrating this.

Emphasize that the example is Python but we will try to see "through" the code and **focus on the bigger picture** and hopefully manage to imagine other languages in its place.

We **collect ideas and feedback in the collaborative document while coding** and the instructor tries to react to that without going into the rabbit hole.

Learners can also explore some of these steps in one of the exercise sessions.

Checklist

- Start with notebook
- Add statistics (mean temperature)
- Add precipitation
- Generalize from January to also February and March data
- Abstract code into functions
- Move from notebook to script
- From functions with side-effects towards stateless functions
- Initialize git
- Add `requirements.txt`
- Add test
- Add command line interface
- Show how a workflow solution could look
- Split into multiple files/modules

Our initial version

We imagine that we assemble a working script/code from various internet research/ AI chat recommendations and arrive at:

Python | R

```python
import pandas as pd
import matplotlib.pyplot as plt


# read data
data = pd.read_csv("weather_data.csv")

# combine 'date' and 'time' into a single column 'recorded_at' as type datetime
data["recorded_at"] = pd.to_datetime(data["date"] + " " + data["time"])

# set 'recorded_at' as index for convenience
data = data.set_index("recorded_at")

# keep only january data using datetime period indexing
january = data.loc["2024-01"]

fig, ax = plt.subplots()

# temperature time series
ax.plot(
    january.index,
    january["air_temperature_celsius"],
    label="air temperature (C)",
    color="red",
)

ax.set_title("air temperature (C) at Helsinki airport")
ax.set_xlabel("date and time")
ax.set_ylabel("air temperature (C)")
ax.legend()
ax.grid(True)

# format x-axis for better date display
fig.autofmt_xdate()

fig.savefig("2024-01-temperature.png")
```

- We test it out **in a notebook**.

We add a dashed line representing the mean temperature

This is still only the January data.

Python R

```python
import pandas as pd
import matplotlib.pyplot as plt


# read data
data = pd.read_csv("weather_data.csv")

# combine 'date' and 'time' into a single column 'recorded_at' as type datetime
data["recorded_at"] = pd.to_datetime(data["date"] + " " + data["time"])

# set 'recorded_at' as index for convenience
data = data.set_index("recorded_at")

# keep only january data using datetime period indexing
january = data.loc["2024-01"]

fig, ax = plt.subplots()

# temperature time series
ax.plot(
    january.index,
    january["air_temperature_celsius"],
    label="air temperature (C)",
    color="red",
)

values = january["air_temperature_celsius"].values
mean_temp = sum(values) / len(values)

# mean temperature (as horizontal dashed line)
ax.axhline(
    y=mean_temp,
    label=f"mean air temperature (C): {mean_temp:.1f}",
    color="red",
    linestyle="--",
)

ax.set_title("air temperature (C) at Helsinki airport")
ax.set_xlabel("date and time")
ax.set_ylabel("air temperature (C)")
ax.legend()
ax.grid(True)

# format x-axis for better date display
fig.autofmt_xdate()

fig.savefig("2024-01-temperature.png")
```

We add another plot for the precipitation

As a first go, we achieve this by copy pasting the existing code and adjusting it for the precipitation column.

Python R

```python
import pandas as pd
import matplotlib.pyplot as plt


# read data
data = pd.read_csv("weather_data.csv")

# combine 'date' and 'time' into a single column 'recorded_at' as type datetime
data["recorded_at"] = pd.to_datetime(data["date"] + " " + data["time"])

# set 'recorded_at' as index for convenience
data = data.set_index("recorded_at")

# keep only january data using datetime period indexing
january = data.loc["2024-01"]

fig, ax = plt.subplots()

# temperature time series
ax.plot(
    january.index,
    january["air_temperature_celsius"],
    label="air temperature (C)",
    color="red",
)

values = january["air_temperature_celsius"].values
mean_temp = sum(values) / len(values)

# mean temperature (as horizontal dashed line)
ax.axhline(
    y=mean_temp,
    label=f"mean air temperature (C): {mean_temp:.1f}",
    color="red",
    linestyle="--",
)

ax.set_title("air temperature (C) at Helsinki airport")
ax.set_xlabel("date and time")
ax.set_ylabel("air temperature (C)")
ax.legend()
ax.grid(True)

# format x-axis for better date display
fig.autofmt_xdate()

fig.savefig("2024-01-temperature.png")

fig, ax = plt.subplots()

# precipitation time series
ax.plot(
    january.index,
    january["precipitation_mm"],
    label="precipitation (mm)",
    color="blue",
)

ax.set_title("precipitation (mm) at Helsinki airport")
ax.set_xlabel("date and time")
ax.set_ylabel("precipitation (mm)")
ax.legend()
```

```
ax.grid(True)

# format x-axis for better date display
fig.autofmt_xdate()

fig.savefig("2024-01-precipitation.png")
```

Plotting also February and March data

- Copy-pasting very similar code 6 times would be too complicated to maintain.
- We avoid this by iterating over the first 3 months.
- Instead of reusing `data`, we introduce `data_month`.

Python | R

```python
import pandas as pd
import matplotlib.pyplot as plt


# read data
data = pd.read_csv("weather_data.csv")

# combine 'date' and 'time' into a single column 'recorded_at' as type datetime
data["recorded_at"] = pd.to_datetime(data["date"] + " " + data["time"])

# set 'recorded_at' as index for convenience
data = data.set_index("recorded_at")

for month in ["2024-01", "2024-02", "2024-03"]:
    data_month = data.loc[month]

    fig, ax = plt.subplots()

    # temperature time series
    ax.plot(
        data_month.index,
        data_month["air_temperature_celsius"],
        label="air temperature (C)",
        color="red",
    )

    values = data_month["air_temperature_celsius"].values
    mean_temp = sum(values) / len(values)

    # mean temperature (as horizontal dashed line)
    ax.axhline(
        y=mean_temp,
        label=f"mean air temperature (C): {mean_temp:.1f}",
        color="red",
        linestyle="--",
    )

    ax.set_title("air temperature (C) at Helsinki airport")
    ax.set_xlabel("date and time")
    ax.set_ylabel("air temperature (C)")
    ax.legend()
    ax.grid(True)

    # format x-axis for better date display
    fig.autofmt_xdate()

    fig.savefig(f"{month}-temperature.png")

    fig, ax = plt.subplots()

    # precipitation time series
    ax.plot(
        data_month.index,
        data_month["precipitation_mm"],
        label="precipitation (mm)",
        color="blue",
    )

    ax.set_title("precipitation (mm) at Helsinki airport")
    ax.set_xlabel("date and time")
    ax.set_ylabel("precipitation (mm)")
    ax.legend()
```

```
    ax.grid(True)

    # format x-axis for better date display
    fig.autofmt_xdate()

    fig.savefig(f"{month}-precipitation.png")
```

Abstracting the plotting part into a function

Python | R

```python
import pandas as pd
import matplotlib.pyplot as plt


def plot(column, label, location, color, compute_mean):
    fig, ax = plt.subplots()

    # time series
    ax.plot(
        data_month.index,
        data_month[column],
        label=label,
        color=color,
    )

    if compute_mean:
        values = data_month[column].values
        mean_value = sum(values) / len(values)

        # mean (as horizontal dashed line)
        ax.axhline(
            y=mean_value,
            label=f"mean {label}: {mean_value:.1f}",
            color=color,
            linestyle="--",
        )

    ax.set_title(f"{label} at {location}")
    ax.set_xlabel("date and time")
    ax.set_ylabel(label)
    ax.legend()
    ax.grid(True)

    # format x-axis for better date display
    fig.autofmt_xdate()

    fig.savefig(f"{month}-{column}.png")


# read data
data = pd.read_csv("weather_data.csv")

# combine 'date' and 'time' into a single column 'recorded_at' as type datetime
data["recorded_at"] = pd.to_datetime(data["date"] + " " + data["time"])

# set 'recorded_at' as index for convenience
data = data.set_index("recorded_at")


for month in ["2024-01", "2024-02", "2024-03"]:
    data_month = data.loc[month]

    plot(
        "air_temperature_celsius",
        "air temperature (C)",
        "Helsinki airport",
        "red",
        compute_mean=True,
    )
    plot(
        "precipitation_mm",
        "precipitation (mm)",
```

```
        "Helsinki airport",
        "blue",
        compute_mean=False,
    )
```

- Discuss the advantages of what we have done here.
- Discuss what we expect before running it (we might expect this not to work because `data_month` seems undefined inside the function).
- Then try it out (it actually works).
- Discuss problems with this solution (what if we copy-paste the function to a different file?).

The point of this step was that abstracting code into functions can be really good for re-usability but just the fact that we created a function does not mean that the function is reusable since in this case it depends on a variable defined outside the function and hence there are **side-effects**.

Small improvements

- Abstracting into more functions.
- Notice how some code comments got redundant:

Python | R

```python
import pandas as pd
import matplotlib.pyplot as plt


def read_data(file_name):
    data = pd.read_csv(file_name)

    # combine 'date' and 'time' into a single column 'recorded_at' as type
datetime
    data["recorded_at"] = pd.to_datetime(data["date"] + " " + data["time"])

    # set 'recorded_at' as index for convenience
    data = data.set_index("recorded_at")

    return data


def arithmetic_mean(values):
    mean_value = sum(values) / len(values)
    return mean_value


def plot(column, label, location, color, compute_mean):
    fig, ax = plt.subplots()

    # time series
    ax.plot(
        data_month.index,
        data_month[column],
        label=label,
        color=color,
    )

    if compute_mean:
        mean_value = arithmetic_mean(data_month[column].values)

        # mean (as horizontal dashed line)
        ax.axhline(
            y=mean_value,
            label=f"mean {label}: {mean_value:.1f}",
            color=color,
            linestyle="--",
        )

    ax.set_title(f"{label} at {location}")
    ax.set_xlabel("date and time")
    ax.set_ylabel(label)
    ax.legend()
    ax.grid(True)

    # format x-axis for better date display
    fig.autofmt_xdate()

    fig.savefig(f"{month}-{column}.png")


data = read_data("weather_data.csv")

for month in ["2024-01", "2024-02", "2024-03"]:
    data_month = data.loc[month]

    plot(
```

```
        "air_temperature_celsius",
        "air temperature (C)",
        "Helsinki airport",
        "red",
        compute_mean=True,
    )
    plot(
        "precipitation_mm",
        "precipitation (mm)",
        "Helsinki airport",
        "blue",
        compute_mean=False,
    )
```

Discuss what would happen if we copy-paste the functions to another project (these functions are stateful/time-dependent).

Emphasize how stateful functions and order of execution in Jupyter notebooks can produce unexpected results and explain why we motivate to rerun all cells before sharing the notebook.

Move from notebook to script

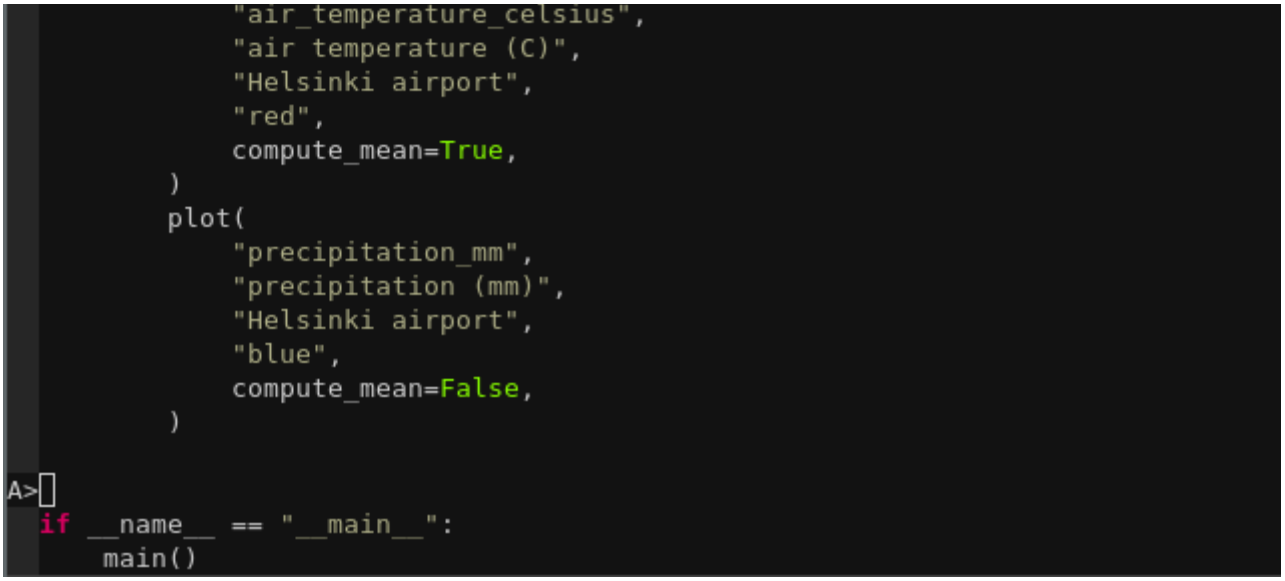
- "File" -> "Save and Export Notebook As ..." -> "Executable Script"
- `git init` and commit the working version.
- Add `requirements.txt` and motivate how that can be useful to have later.

As we continue from here, **create commits after meaningful changes** and later also share the repository with learners. This nicely connects to other lessons of the workshop.

Towards functions without side-effects

In Python we can detect problems by encapsulating all code into functions and when using a code editor with a static checker (instructor can demonstrate this by first introducing a main function, then detecting problems, then fixing the problems):

```
def read_data(file_name):
    data = pd.read_csv(file_name)

    # combine 'date' and 'time' into a single datetime column
    data["datetime"] = pd.to_datetime(data["date"] + " " + data["time"])

    # set datetime as index for convenience
    data = data.set_index("datetime")

    return data


def arithmetic_mean(values):
    mean_value = sum(values) / len(values)
    return mean_value


def plot(column, label, location, color, compute_mean):
    fig, ax = plt.subplots()

    # time series
    ax.plot(
E>      data_month.index,
 Undefined name `data_month`
E>      data_month[column],
 Undefined name `data_month`
        label=label,
        color=color,
    )

    if compute_mean:
E>      mean_value = arithmetic_mean(data_month[column].values)
 Undefined name `data_month`

        # mean (as horizontal dashed line)
        ax.axhline(
            y=mean_value,
            label=f"mean {label}: {mean_value:.1f}",
            color=color,
            linestyle="--",
        )

    ax.set_title(f"{label} at {location}")
    ax.set_xlabel("date and time")
    ax.set_ylabel(label)
    ax.legend()
    ax.grid(True)

    # format x-axis for better date display
    fig.autofmt_xdate()

E>  fig.savefig(f"{month}-{column}.png")
 Undefined name `month`


def main():
    data = read_data("weather_data.csv")

    for month in ["2024-01", "2024-02", "2024-03"]:
W>      data_month = data.loc[month]
 Local variable `data_month` is assigned to but never used

        plot(
```



After we have tucked the "main" code under a main function, an editor with linter/checker enabled highlights undefined names and variables which are assigned but never used. The screenshot was obtained from a vim editor with ruff language server enabled.

We then improve towards:

Python R

```python
import pandas as pd
import matplotlib.pyplot as plt


def read_data(file_name):
    data = pd.read_csv(file_name)

    # combine 'date' and 'time' into a single column 'recorded_at' as type
datetime
    data["recorded_at"] = pd.to_datetime(data["date"] + " " + data["time"])

    # set 'recorded_at' as index for convenience
    data = data.set_index("recorded_at")

    return data


def arithmetic_mean(values):
    mean_value = sum(values) / len(values)
    return mean_value


def plot(date_range, values, label, location, color, compute_mean, file_name):
    fig, ax = plt.subplots()

    # time series
    ax.plot(
        date_range,
        values,
        label=label,
        color=color,
    )

    if compute_mean:
        mean_value = arithmetic_mean(values)

        # mean (as horizontal dashed line)
        ax.axhline(
            y=mean_value,
            label=f"mean {label}: {mean_value:.1f}",
            color=color,
            linestyle="--",
        )

    ax.set_title(f"{label} at {location}")
    ax.set_xlabel("date and time")
    ax.set_ylabel(label)
    ax.legend()
    ax.grid(True)

    # format x-axis for better date display
    fig.autofmt_xdate()

    fig.savefig(file_name)


def main():
    data = read_data("weather_data.csv")

    for month in ["2024-01", "2024-02", "2024-03"]:
        data_month = data.loc[month]
        date_range = data_month.index
```

```
        plot(
            date_range,
            data_month["air_temperature_celsius"].values,
            "air temperature (C)",
            "Helsinki airport",
            "red",
            compute_mean=True,
            file_name=f"{month}-temperature.png",
        )
        plot(
            date_range,
            data_month["precipitation_mm"].values,
            "precipitation (mm)",
            "Helsinki airport",
            "blue",
            compute_mean=False,
            file_name=f"{month}-precipitation.png",
        )


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

These functions can now be copy-pasted to a different notebook or project and they will still work.

Unit tests (optional)

In this section we look at how to apply the testing lesson to this example.

- Discuss what one could mean with “design code for testing”.
- Discuss when to test and when not to test.
- Discuss where to add a test and add a test to the `arithmetic_mean` function:

Python | R

```python
import pandas as pd
import matplotlib.pyplot as plt
import pytest


def read_data(file_name):
    data = pd.read_csv(file_name)

    # combine 'date' and 'time' into a single column 'recorded_at' as type
datetime
    data["recorded_at"] = pd.to_datetime(data["date"] + " " + data["time"])

    # set 'recorded_at' as index for convenience
    data = data.set_index("recorded_at")

    return data


def arithmetic_mean(values):
    mean_value = sum(values) / len(values)
    return mean_value


def test_arithmetic_mean():
    result = arithmetic_mean([1.0, 2.0, 3.0, 4.0])
    assert result == pytest.approx(2.5)


def plot(date_range, values, label, location, color, compute_mean, file_name):
    fig, ax = plt.subplots()

    # time series
    ax.plot(
        date_range,
        values,
        label=label,
        color=color,
    )

    if compute_mean:
        mean_value = arithmetic_mean(values)

        # mean (as horizontal dashed line)
        ax.axhline(
            y=mean_value,
            label=f"mean {label}: {mean_value:.1f}",
            color=color,
            linestyle="--",
        )

    ax.set_title(f"{label} at {location}")
    ax.set_xlabel("date and time")
    ax.set_ylabel(label)
    ax.legend()
    ax.grid(True)

    # format x-axis for better date display
    fig.autofmt_xdate()

    fig.savefig(file_name)
```

```
def main():
    data = read_data("weather_data.csv")

    for month in ["2024-01", "2024-02", "2024-03"]:
        data_month = data.loc[month]
        date_range = data_month.index

        plot(
            date_range,
            data_month["air_temperature_celsius"].values,
            "air temperature (C)",
            "Helsinki airport",
            "red",
            compute_mean=True,
            file_name=f"{month}-temperature.png",
        )
        plot(
            date_range,
            data_month["precipitation_mm"].values,
            "precipitation (mm)",
            "Helsinki airport",
            "blue",
            compute_mean=False,
            file_name=f"{month}-precipitation.png",
        )


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

Command-line interface (CLI)

- Add a CLI for the input data file, the month, and the output folder.
- Instructor demonstrates it, for instance:

```
$ python example.py --month 2024-05 --data-file weather_data.csv --output-directory
/home/user/example/results
```

- Example here is using click but it can equally well be optparse, argparse, docopt, or Typer.
- Discuss the motivations for adding a CLI:
  - We are able to modify the behavior without changing (or needing to understand) the code
  - We can run many of such scripts as part of a workflow

Python | R

```python
from pathlib import Path


import pandas as pd
import matplotlib.pyplot as plt
import pytest
import click


def read_data(file_name):
    data = pd.read_csv(file_name)

    # combine 'date' and 'time' into a single column 'recorded_at' as type
datetime
    data["recorded_at"] = pd.to_datetime(data["date"] + " " + data["time"])

    # set 'recorded_at' as index for convenience
    data = data.set_index("recorded_at")

    return data


def arithmetic_mean(values):
    mean_value = sum(values) / len(values)
    return mean_value


def test_arithmetic_mean():
    result = arithmetic_mean([1.0, 2.0, 3.0, 4.0])
    assert result == pytest.approx(2.5)


def plot(date_range, values, label, location, color, compute_mean, file_name):
    fig, ax = plt.subplots()

    # time series
    ax.plot(
        date_range,
        values,
        label=label,
        color=color,
    )

    if compute_mean:
        mean_value = arithmetic_mean(values)

        # mean (as horizontal dashed line)
        ax.axhline(
            y=mean_value,
            label=f"mean {label}: {mean_value:.1f}",
            color=color,
            linestyle="--",
        )

    ax.set_title(f"{label} at {location}")
    ax.set_xlabel("date and time")
    ax.set_ylabel(label)
    ax.legend()
    ax.grid(True)

    # format x-axis for better date display
    fig.autofmt_xdate()
```

```
    fig.savefig(file_name)


@click.command()
@click.option("--month", required=True, type=str, help="Which month (YYYY-MM)?")
@click.option(
    "--data-file",
    required=True,
    type=click.Path(exists=True, path_type=Path),
    help="Data is read from this file.",
)
@click.option(
    "--output-directory",
    required=True,
    type=click.Path(exists=True, path_type=Path),
    help="Figures are written to this directory.",
)
def main(
    month,
    data_file,
    output_directory,
):
    data = read_data(data_file)

    data_month = data.loc[month]
    date_range = data_month.index

    plot(
        date_range,
        data_month["air_temperature_celsius"].values,
        "air temperature (C)",
        "Helsinki airport",
        "red",
        compute_mean=True,
        file_name=output_directory / f"{month}-temperature.png",
    )
    plot(
        date_range,
        data_month["precipitation_mm"].values,
        "precipitation (mm)",
        "Helsinki airport",
        "blue",
        compute_mean=False,
        file_name=output_directory / f"{month}-precipitation.png",
    )


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

Split long script into modules

- Discuss how you would move some functions out and organize them into separate modules which can be imported to other projects.
- Discuss naming.
- Discuss interface design.

Summarize in the collaborative document

- Now return to initial questions on the collaborative document and discuss questions and comments. If there is time left, there are additional questions and exercises.
- It is easier and more fun to teach this as a pair with somebody else where one person can type and the other person helps watching the questions and commends and relays them to the co-instructor.