Python Progression: Moving Beyond the Basics

In this short intermediate-level workshop, we will learn how to write Python with a focus on reproducibility and re-usability.

We will start in Jupyter Notebooks and later move to running Python scripts from the command line.

Who is the course for?

- Somebody who has written Python scripts or notebooks and wants to improve their code style and code re-usability.
- Somebody who needs to read, process, and plot data for their work or studies.
- Persons who already use Python for this but want to learn about libraries to simplify common tasks and about how to share their workflow in a reproducible way.

⚙ Preparation

- We assume everybody follows on their computer with network access
- Software install instructions (but we will go through this together)

Episode overview

Warm-up:

- Python basics

Day 1 morning:

- 11:00 - 12:30
  - Software install instructions
  - Jupyter Notebooks
  - Data formats, tidy data, and data cleaning

Day 1 afternoon:

- 13:30 - 15:00
  - Plotting with Vega-Altair
  - Learning how to adapt existing gallery examples
- 15:30 - 17:00

    - From notebooks to scripts
    - Command-line interfaces (CLI)
    - Good practices and tools
- 17:00 - 18:00
    - Q&A
    - Working on own scripts/projects

Day 2 morning:

- 09:00 - 09:45
    - Version control (motivation)
    - Reproducible environments and dependencies
    - Where to start with documentation
- 09:45 - 10:20
    - Profiling
- 10:30 - 11:00
    - Automated testing

Software install instructions

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

Choosing an installation method

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

environment.yml | requirements.txt

```yaml
name: course
channels:
  - conda-forge
dependencies:
  - python <= 3.12
  - jupyterlab
  - altair-all
  - vega_datasets
  - pandas
  - numpy
  - pytest
  - scalene
  - ruff
  - icecream
  - myst-parser
  - sphinx
  - sphinx-rtd-theme
  - sphinx-autoapi
  - sphinx-autobuild
```

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

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

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

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

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

❗ Python, conda, anaconda, miniforge, etc?

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

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

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

Installing Python via Miniforge

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

Unsure what to download and what to do with it?

Windows | MacOS | Linux

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

Installing and activating the software environment

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

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

We will call the environment `course`.

Windows | Linux / MacOS

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

```
$ mamba env create -n course -f
https://raw.githubusercontent.com/coderefinery/python-
progression/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 `course` environment.

Windows | Linux / MacOS

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

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

Removing the software environment

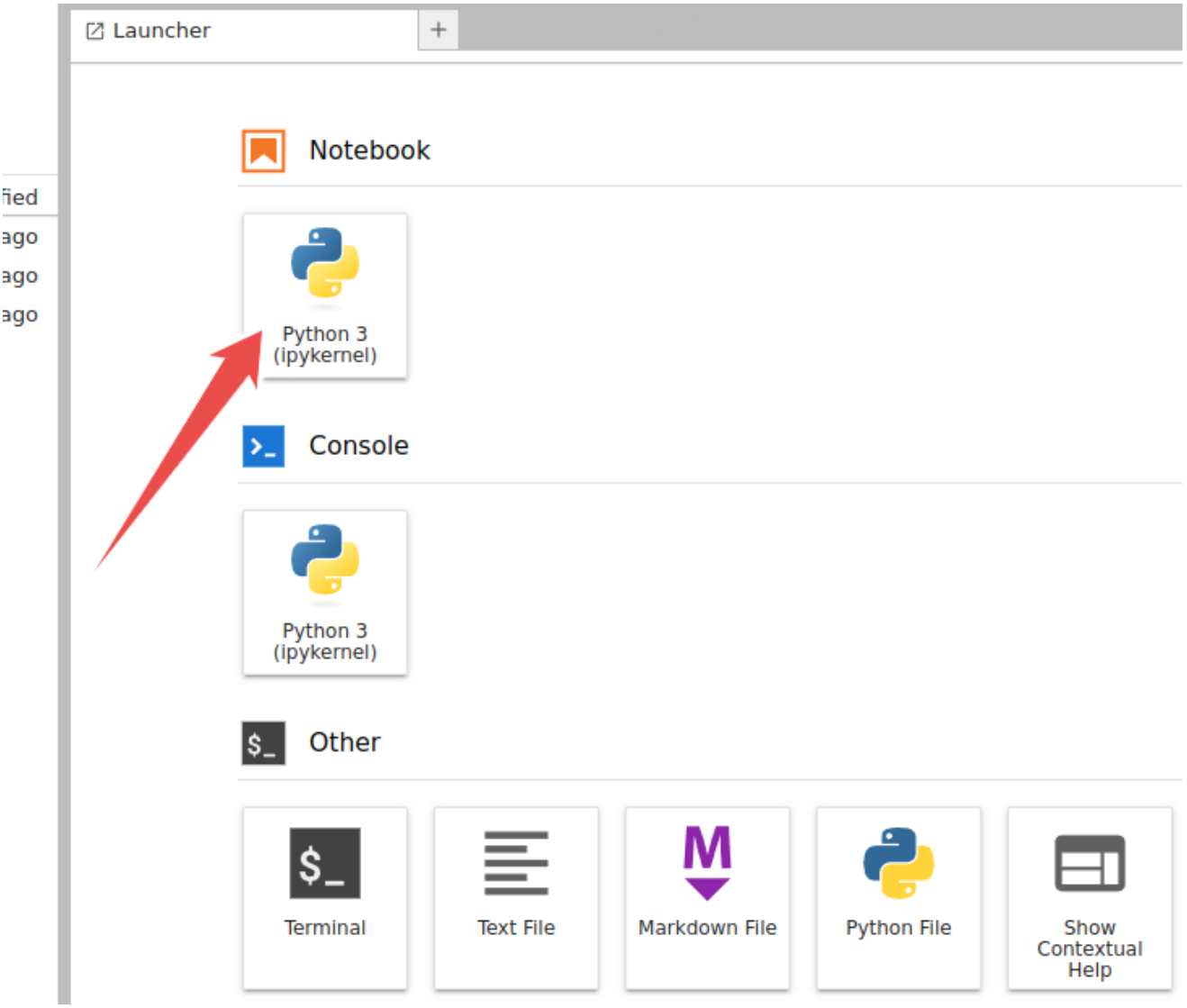
Windows | Linux / MacOS

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

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

How to verify your installation

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



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

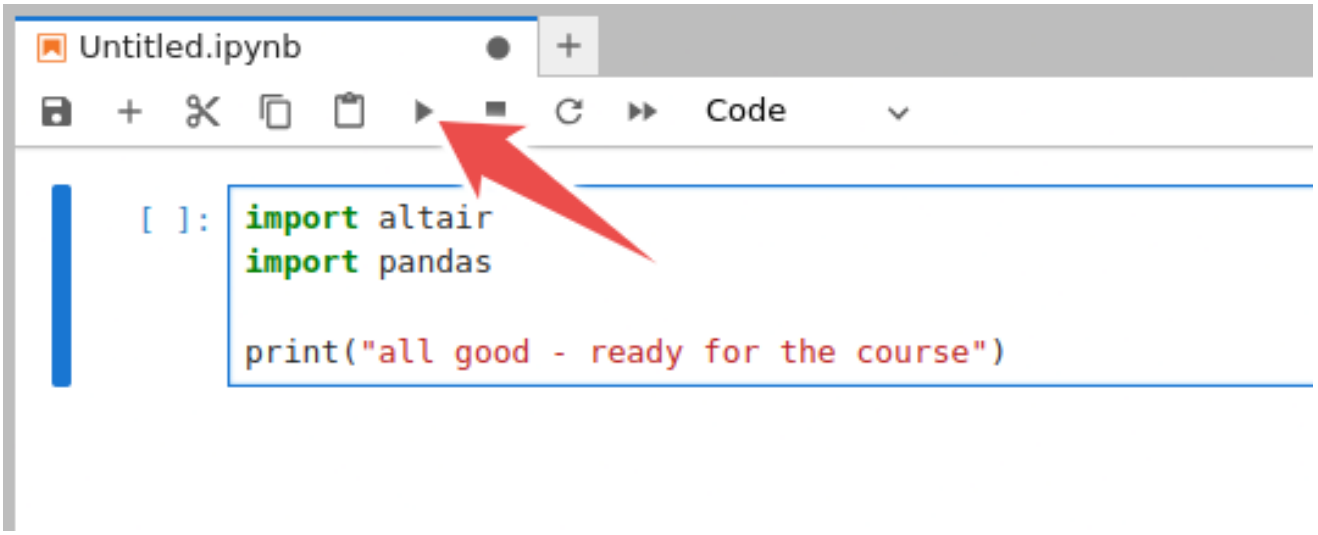
Once you clicked the Python 3 tile it should look like this:

Python 3 notebook started.

Into that blue “cell” please type the following:

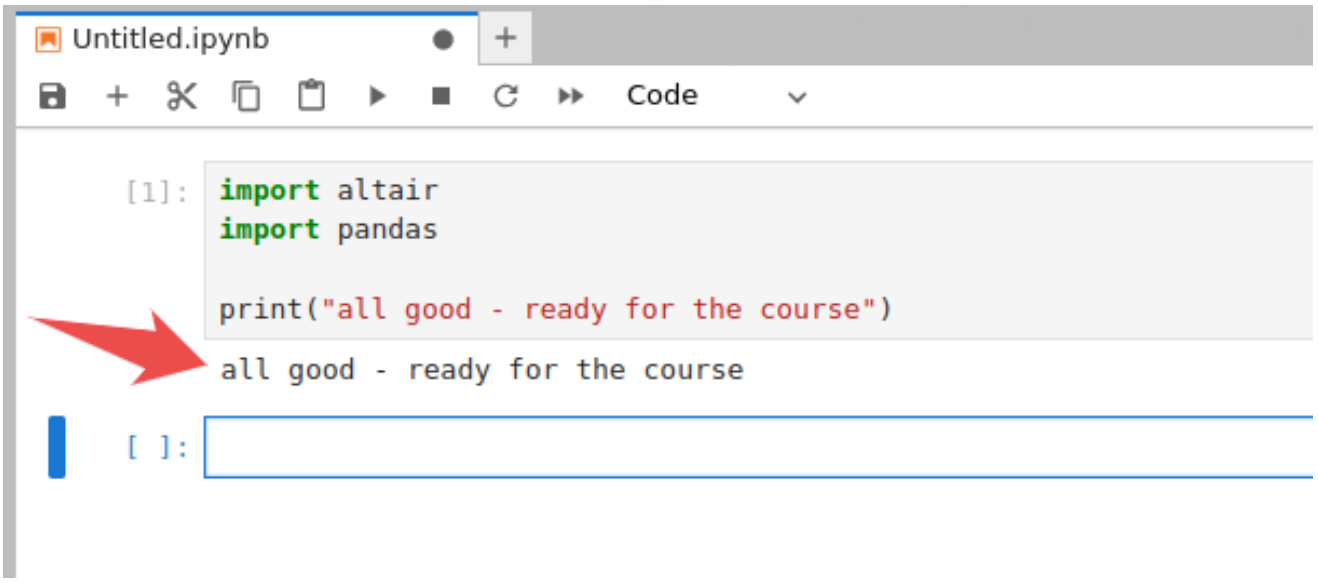
```
import altair
import pandas

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



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

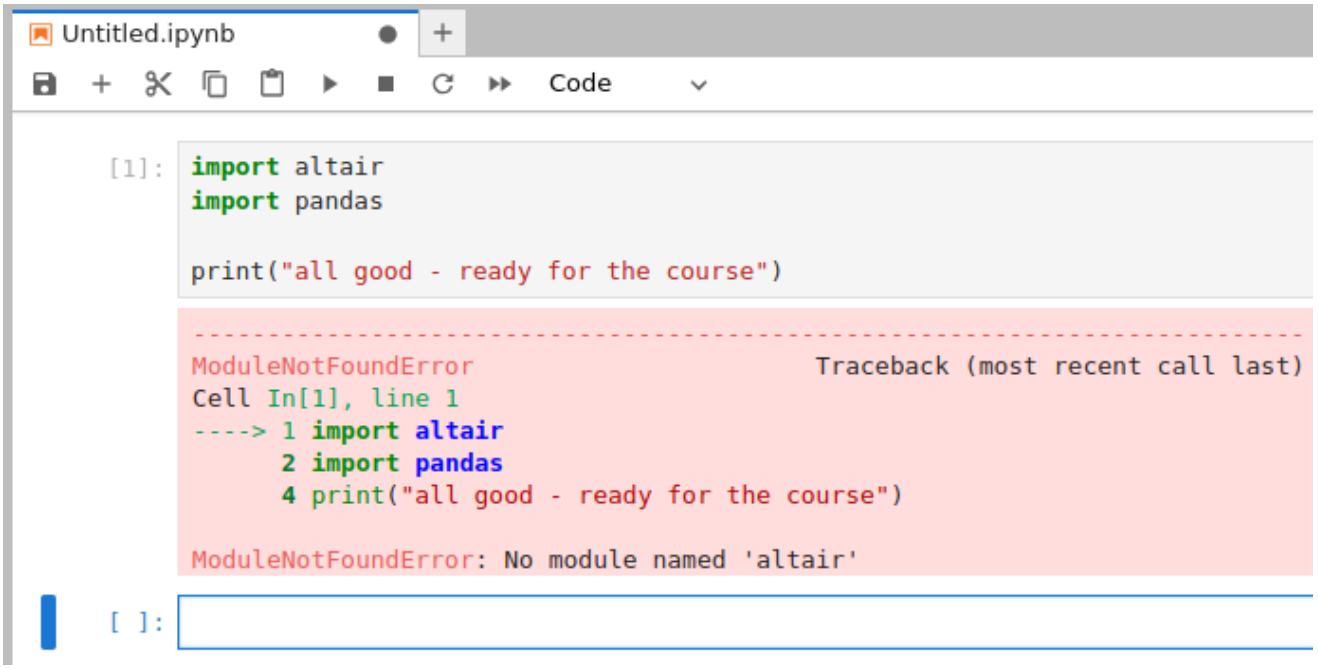
This is how it should look:



Screenshot after successful import.

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

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



Error: required packages could not be found.

Python basics

Objectives

- Knowing what types exist
- Knowing the most common data structures (collections): lists, tuples, dictionaries, and sets
- Creating and using functions
- Knowing what a library is
- Knowing what `import` does
- Being able to “read” an error

Motivation for Python

- **Free**
- Huge **ecosystem of examples, libraries, and tools**
- Relatively easy to read and understand
- Similar in scope and use cases to R, Julia, and Matlab

Basic types

```
# int
num_measurements = 13

# float
some_fraction = 0.25

# string
name = "Bruce Wayne"

# bool
value_is_missing = False
skip_verification = True

# we can print values
print(name)

# and we can do arithmetics with ints and floats
print(5 * num_measurements)
print(1.0 - some_fraction)
```

- Python is **dynamically typed**: We do not have to define that an integer is an `int`, we can use it this way and Python will infer it.
- However, one can use type annotations in Python (see also mypy).
- Now you also know that we can add `# comments` to our code.

Data structures for collections: lists, dictionaries, sets, and tuples

```
# lists are good when order is important
scores = [13, 5, 2, 3, 4, 3]

# first element
print(scores[0])

# we can add items to lists
scores.append(4)

# lists can be sorted
scores.sort()
print(scores)

# dictionaries are useful if you want to look up
# elements in a collection by something else than position
experiment = {"location": "Svalbard", "date": "2021-03-23", "num_measurements": 23}

print(experiment["date"])

# we can add items to dictionaries
experiment["instrument"] = "a particular brand"
print(experiment)

if "instrument" in experiment:
    print("yes, the dictionary 'experiment' contains the key 'instrument'")
else:
    print("no, it doesn't")
```

- `Lists` are good when order is important, and it needs to be changed

- `Dictionaries` are mappings key→value.
- `Sets` are useful for unordered collections where you want to make sure that there are no repetitions.
- There are also `tuples` that are similar to lists but their items cannot be modified.

You can put:

- dictionaries inside lists
- lists inside dictionaries
- dictionaries inside dictionaries
- lists inside lists
- tuples inside …
- …

Iterating over collections

Often we wish to iterate over collections.

Iterating over a list:

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

for score in scores:
    print(score)

# example with f-strings
for score in scores:
    print(f"the score is {score}")
```

We don't have to call the variable inside the for-loop "score". This is up to us. We can do this instead (but is this more understandable for humans?):

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

for x in scores:
    print(x)
```

Iterating over a dictionary:

```
experiment = {"location": "Svalbard", "date": "2021-03-23", "num_measurements": 23}

for key in experiment:
    print(experiment[key])

# another way to iterate
for (key, value) in experiment.items():
    print(key, value)
```

Functions

- Functions are like **reusable recipes**. They receive ingredients (input arguments), then inside the function we do/compute something with these arguments, and they return a result.

```
def add(a, b):
    result = a + b
    return result
```

- Together we can write a function which sums all elements in a list (this is not the most elegant way to do this but it is easier at this point):

```
def add_all_elements(sequence):
    """
    This function adds all elements.
    This here is a docstring, a documentation string for a function.
    """
    s = 0.0
    for element in sequence:
        s += element
    return s


measurements = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

print(add_all_elements(measurements))
```

- We reuse this function to write a function which computes the mean:

```
def arithmetic_mean(sequence):
    # we are reusing add_all_elements written above
    s = add_all_elements(sequence)
    n = len(sequence)
    return s / n


measurements = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

mean = arithmetic_mean(measurements)

print(mean)
```

- Functions can call other functions. Functions can also get other functions as input arguments.
- Functions can return more than one thing:

```
def uppercase_and_lowercase(text):
    u = text.upper()
    l = text.lower()
    return u, l


some_text = "SequenceOfCharacters"
uppercased_text, lowercased_text = uppercase_and_lowercase(some_text)

print(uppercased_text)
print(lowercased_text)
```

Why functions?

- Less repetition
- Simplify reading and understanding code
- Simpler re-use of code
- Make it easier for Python to deallocate memory

Reading error messages

Here we introduce a mistake and we together try to make sense of the traceback:

```
---------------------------------------------------------------------------
--
NameError                                 Traceback (most recent call las
t)
<ipython-input-10-8e6794a136dc> in <module>
      8 measurements = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
      9
---> 10 mean = arithmetic_mean(measurements)
     11
     12 print(mean)

<ipython-input-10-8e6794a136dc> in arithmetic_mean(sequence)
      1 def arithmetic_mean(sequence):
      2     # we are reusing add_all_elements written above
----> 3     s = add_all_element(sequence)
      4     n = len(sequence)
      5     return s / n

NameError: name 'add_all_element' is not defined
```

Example error traceback. Can you explain the error?

Libraries

We can look at libraries as collections of functions. We can import the libraries/modules and then reuse the functions defined inside these libraries.

Try this:

```
import numpy

measurements = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

result = numpy.std(measurements)

print(result)
```

This means `numpy` contains a function called `std` which apparently computes the standard deviation (check also its documentation).

Often you see this in tutorials (the module is imported and renamed to a shortcut):

```
import numpy as np

result = np.std([1, 2, 3, 4, 5, 6, 7, 8, 9, 10])
```

It is possible to create own modules to collect own functions for reuse.

Great resources to learn more

- Real Python Tutorials (great for beginners)
- The Python Tutorial (great for beginners)
- The Hitchhiker's Guide to Python! (intermediate level)
- Effective Python (once you know the basics and want to write better code)

Exercises

✍ Exercise: create a function that computes the standard deviation

- Arithmetic mean:

$$\bar{x} = \frac{1}{N} \sum_{i=1}^N x_i$$

- Standard deviation:

$$\sqrt{ \frac{1}{N} \sum_{i=1}^N (x_i - \bar{x})^2 }$$

- In other words the computation is similar but we need to sum over squares of differences and at the end take a square root.
- Take this as a starting point:

```
# we have written this one together previously
def arithmetic_mean(sequence):
    s = 0.0
    for element in sequence:
        s += element
    n = len(sequence)
    return s / n


def standard_deviation(sequence):
    # here we need to do some work:
    # mean = ?
    # s = ?
    n = len(sequence)
    return (s / n) ** 0.5
```

- If this is the input list:

```
measurements = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Then the result would be: 2.872...

✔ Solution 1 (longer but hopefully easier to understand)

```
# we have written this one together previously
def arithmetic_mean(sequence):
    s = 0.0
    for element in sequence:
        s += element
    n = len(sequence)
    return s / n


# notice how this function reuses the other
def standard_deviation(sequence):
    mean = arithmetic_mean(sequence)
    s = 0.0
    for element in sequence:
        s += (element - mean) ** 2
    n = len(sequence)
    return (s / n) ** 0.5
```

✔ Solution 2 (more compact)

```python
def arithmetic_mean(sequence):
    return sum(sequence) / len(sequence)


def standard_deviation(sequence):
    mean = arithmetic_mean(sequence)
    s = sum([(x - mean) ** 2 for x in sequence])
    n = len(sequence)
    return (s / n) ** 0.5
```

✍️ Exercise: working with a dictionary

- We have this dictionary as a starting point:

```python
grades = {"Alice": 80, "Bob": 95}
```

- Add the grades of few more (fictious) persons to this dictionary.
- Print the entire dictionary.
- What happens when you add a name which already exists (with a different grade)?
- Print the grade for one particular person only.
- What happens when you try to print the result for a person that wasn't there?
- Try also these:

```python
print(grades.keys())
print(grades.values())
print(grades.items())
```

✔ Solution

We can add more people like this:

```python
grades["Craig"] = 56
grades["Dave"] = 28
grades["Eve"] = 75
```

Print the entire dictionary with:

```python
print(grades)
```

We get:

```
{'Alice': 80, 'Bob': 95, 'Craig': 56, 'Dave': 28, 'Eve': 75}
```

Adding an entry which already exists updates the entry (please try it).

Printing the result for one particular person:

```
print(grades["Eve"])
```

Printing the result for a person which does not exists, gives a `KeyError`.

The outputs of these three:

```
print(grades.keys())
print(grades.values())
print(grades.items())
```

... are either the only the keys or only the values, or in the case of `items()`, key-value pairs (tuples):

```
dict_keys(['Alice', 'Bob', 'Craig', 'Dave', 'Eve'])
dict_values([80, 95, 56, 28, 75])
dict_items([('Alice', 80), ('Bob', 95), ('Craig', 56), ('Dave', 28), ('Eve', 75)])
```

The exercises below use if-statements.

✍ Optional exercise/ homework: removing duplicates

- This list contains duplicates:

```
measurements = [2, 2, 1, 17, 3, 3, 2, 1, 13, 14, 17, 14, 4]
```

- Write a function which removes duplicates from the list and sorts the list. In this case it would produce:

```
[1, 2, 3, 4, 13, 14, 17]
```

✔ Solution 1 (longer but hopefully easier to understand)

The function `sorted` sorts a sequence but it creates a new sequence. This is useful if you need a sorted result without changing the original sequence.

We could have achieved the same result with `list.sort()`.

```
def remove_duplicates_and_sort(sequence):
    new_sequence = []
    for element in sequence:
        if element not in new_sequence:
            new_sequence.append(element)
    return sorted(new_sequence)
```

✔ Solution 2 (more compact)

Converting to set removes duplicates. Then we convert back to list:

```
def remove_duplicates_and_sort(sequence):
    new_sequence = list(set(sequence))
    return sorted(new_sequence)
```

✍️ Optional exercise/ homework: counting how often an item appears

- Back to our list with duplicates:

```
measurements = [2, 2, 1, 17, 3, 3, 2, 1, 13, 14, 17, 14, 4]
```

- Your goal is to write a function which will return a dictionary mapping each number to how often it appears. In this case it would produce:

```
{2: 3, 1: 2, 17: 2, 3: 2, 13: 1, 14: 2, 4: 1}
```

✔ Solution 1 (longer but hopefully easier to understand)

```
def how_often(sequence):
    counts = {}
    for element in sequence:
        if element in counts:
            counts[element] += 1
        else:
            counts[element] = 1
    return counts
```

✔ Solution 2 (more compact)

The point of this solution is to show that for such common operations, ready-made functions and objects already exist and is is worth to check out the documentation about the collections module.

```
from collections import Counter, defaultdict


def how_often_alternative1(sequence):
    return dict(Counter(sequence))


def how_often_alternative2(sequence):
    counts = defaultdict(int)
    for element in sequence:
        counts[element] += 1
    return dict(counts)
```

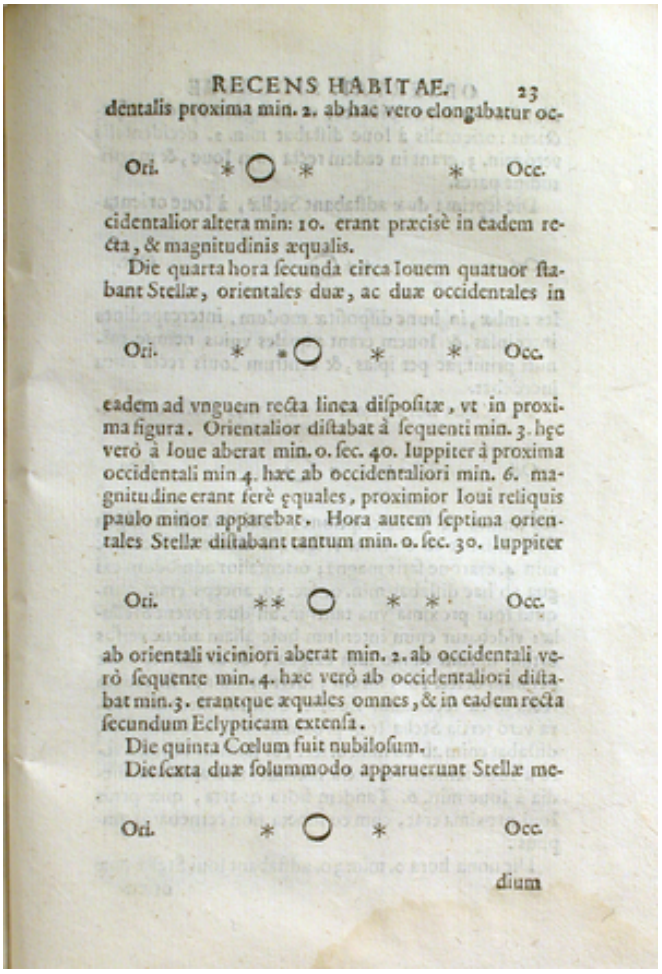
Jupyter Notebooks

❗ 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
- We will build up this notebook (spoiler alert!)

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

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-
ctа, & 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.

Our first notebook

✍ Exercise: Create a notebook (15 min)

- Open a new notebook (if you are unsure how, have a look at Software install instructions)
- 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:

```
# 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.

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**
- **Supporting information with published articles**

Situations where notebooks are less of a good fit:

- Code takes long to run
- It is so long and complex that I need to test it
- When I need a command-line interface
- When I want to process many similar files and each takes few minutes

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.

Data formats, tidy data, and data cleaning

Objectives

- Knowing about different storage formats
- Knowing about the tidy data format
- Be able to reformat tabular data into the tidy data format

Data is not always in nicely formatted “plain” text files. But sometimes the data is in a spreadsheet or in less nicely formatted text files. In this episode we will discuss strategies for how to work with these.

Importing data from spreadsheets

We can create a spreadsheet with the following content (only columns A and B; the actual content does not have to be exactly the same):

	A	B	C	D
1	weekday	number of coffees		
2	monday	3		
3	tuesday	2		
4	wednesday	3		some side note
5	thursday	4		and some color
6	friday	2		
7	saturday	3		
8	sunday	3		
9				

Example spreadsheet with a side note.

Copy this also to the second sheet and for demonstration purpose add some side-notes to the second sheet and also color one or two cells (some people like to give some meaning to cells using color).

Save the spreadsheet as `experiment.xls`.

Now we will together try to read and inspect both sheets in the Jupyter Notebook:

```
import pandas as pd

data = pd.read_excel('experiment.xls', sheet_name="Sheet1")
data

data = pd.read_excel('experiment.xls', sheet_name="Sheet2")
data
```

💬 Discussion

- We can import data from spreadsheets (more documentation)!
- “Side notes” in spreadsheets can be annoying in this context.
- Also encoding data in cell colors is a problem now. We will avoid those in future.

Tidy data

svalbard-observations

File Edit View Insert Format Data Tools Add-ons Help Last edit was 5 minutes ago

100% $ % .0 .00 123 Default (Ari... 10

fx 3

	A	B	C	D	E	F	G
1							
2		**observation site**		A	B	C	
3							
4		**species**					
5							
6		arctic fox		3	1	0	
7		walrus		0	1	1	
8		reindeer		0	10	1	
9		polar bear		1	0	1	
10		seal		2	1	2	
11							
12							
13		comments	red: same animal multiple observations				
14			blue: problem with camera				
15							

Example spreadsheet (this is a phantasy dataset, apologies to biology students/researchers - this is not my domain).

💬 What is the problem with storing data like this?

- Format: Limited interoperability with other programs
- Error prone (see e.g. this famous example
- Difficult to parse ("understand") by scripts: difficult to automate
- Not in *tidy format*: difficult to extend/modify

How should we arrange the data?

Species	Observation sites
arctic fox	A, B
walrus	B, C
reindeer	B, C
polar bear	A, C
seal	A, B, C

Attempt 1: Not great since we need to somehow divide at the comma. How should we deal with multiple sightings?

Species	Observation site A	Observation site B	Observation site C
arctic fox	3	1	0
walrus	0	1	1
reindeer	0	10	1
polar bear	1	0	1
seal	2	1	2

Attempt 2: Adding observation sites will force us to add columns.

Species	arctic fox	walrus	reindeer	polar bear	seal
Observation site A	3	0	0	1	2
Observation site B	1	1	10	0	1
Observation site C	0	1	1	1	2

Attempt 3: Adding species will force us to add columns.

Species	Observation site	Number of sightings
arctic fox	A	3
arctic fox	B	1
walrus	B	1
walrus	C	1
reindeer	B	10
reindeer	C	1
polar bear	A	1
polar bear	C	1
seal	A	2
seal	B	1
seal	C	2

Tidy data format: Columns are variables, rows are observations/measurements. Easy to add new species and sites.

Tidy data format

- Hadley Wickham: Tidy Data
- Columns are variables
- Rows are observations/measurements
- “Long form”
- Order does not matter
- **Easy to extend** with more species and more sites without modifying the code
- **Structure for storing data** - this does not mean that this is ideal for tables in presentations or publications
- It is possible to convert between wide form and long form and back (e.g. using `pandas.melt` or `pandas.pivot`), see this example notebook

Use a standard format

```
Species,Observation site,Number of sightings
arctic fox,A,3
arctic fox,B,1
walrus,B,1
walrus,C,1
reindeer,B,10
reindeer,C,1
polar bear,A,1
polar bear,C,1
seal,A,2
seal,B,1
seal,C,2
```

- **Use a format that is standard in your community, don't invent your own**
- CSV is often a good choice since most visualization tools can read CSV data

There are many more formats (adapted after Python for Scientific Computing):

Name:	Human readable:	Space efficiency:	Arbitrary data:	Tidy data:	Array data:	Long term storage/sharing:
CSV	✅	❌	❌	✅	🟨	✅
Feather	❌	✅	❌	✅	❌	❌
Parquet	❌	✅	🟨	✅	🟨	✅
NPY	❌	🟨	❌	❌	✅	❌
HDF5	❌	✅	❌	❌	✅	✅
NetCDF	❌	✅	❌	❌	✅	✅
JSON	✅	❌	🟨	❌	❌	✅
GeoJSON	✅	❌	🟨	❌	❌	✅
Excel	❌	❌	❌	🟨	❌	🟨
Graph formats	🟨	🟨	❌	❌	❌	✅
SQL	❌	🟨	❌	❌	❌	❌

Note

- ✅ : Good
- 🟨 : Ok / depends on a case
- ❌ : Bad

Data cleaning

Often we want to visualize data sets with inconsistent or missing entries:

```
Date,Organization,Number of participants
2020-09-27,UiT,20
Oct 10 2020,UiT Norges arktiske universitet,15
"Nov. 11, 2020",UiT The Arctic University of Norway,40
2020-12-12,UiT The Arctic University of Norway,-
```

Data cleaning is a bit outside the scope of this course (although we have done some of this in the pandas episode) but still good to know:

- There are tools to clean and merge inconsistent data sets (e.g. OpenRefine, see also this Data Carpentry lesson)
- This does not have to be done manually

Plotting with Vega-Altair

Objectives

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

Repeatability/reproducibility

From Claus O. Wilke: "Fundamentals of Data Visualization":

> *One thing I have learned over the years is that automation is your friend. I think figures should be autogenerated as part of the data analysis pipeline (which should also be automated), and they should come out of the pipeline ready to be sent to the printer, no manual post-processing needed.*

- **Try to minimize manual post-processing**. This could bite you when you need to regenerate 50 figures one day before submission deadline or regenerate a set of figures after the person who created them left the group.
- There is not the one perfect language and **not the one perfect library** for everything.
- Within Python, many libraries exist:
  - Vega-Altair: declarative visualization, statistics built in
  - Matplotlib: probably the most standard and most widely used
  - Seaborn: high-level interface to Matplotlib, statistical functions built in
  - Plotly: interactive graphs
  - Bokeh: also here good for interactivity
  - plotnine: implementation of a grammar of graphics in Python, it is based on ggplot2
  - ggplot: R users will be more at home
  - PyNGL: used in the weather forecast community

    - K3D: Jupyter Notebook extension for 3D visualization
    - Mayavi: 3D scientific data visualization and plotting in Python
    - ...
- Two main families of libraries: procedural (e.g. Matplotlib) and declarative (e.g. Vega-Altair).

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

Example data: Weather data from two Norwegian cities

We will experiment with some example weather data obtained from Norsk KlimaServiceSenter, Meteorologisk institutt (MET) (CC BY 4.0). The data is in CSV format (comma-separated values) and contains daily and monthly weather data for two cities in Norway: Oslo and Tromsø. You can browse the data here in the lesson repository.

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** and tabular data turns out to be a great input format for data visualization libraries. Vega-Altair understands Pandas dataframes and can plot them directly.

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/python-
progression/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
```

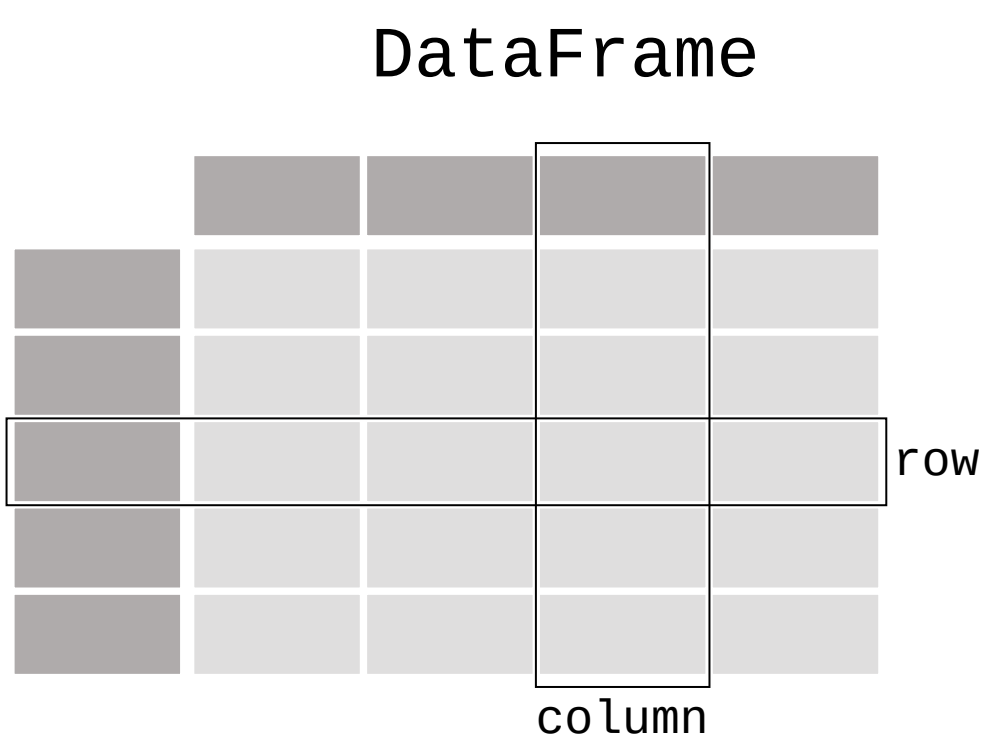
Before plotting the data, there is a problem which we may not see yet: Dates are not in a standard date format (YYYY-MM-DD). We can fix this:

```
# 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.

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]
```

Where to learn more about pandas

Pandas is extremely powerful and there is a lot that can be done and there are great resources to explore more:

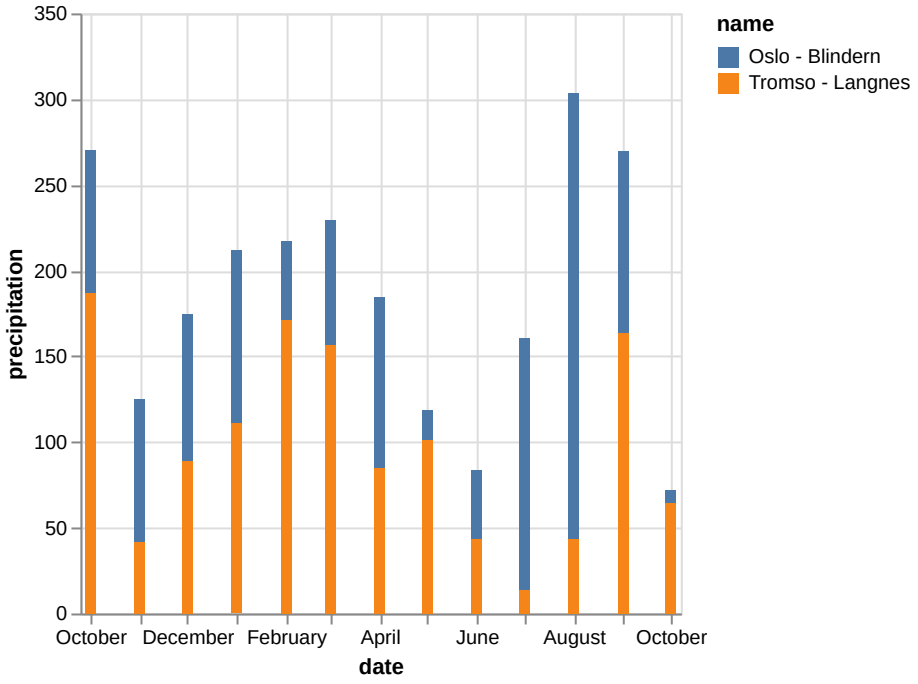
- Getting started guide (including tutorials and a 10 minute flash intro)
- 10 minutes to pandas tutorial
- Pandas documentation
- Cheatsheet
- Cookbook
- Data Carpentry lesson "Data Analysis and Visualization in Python for Ecologists" (useful not only for ecologists)

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:

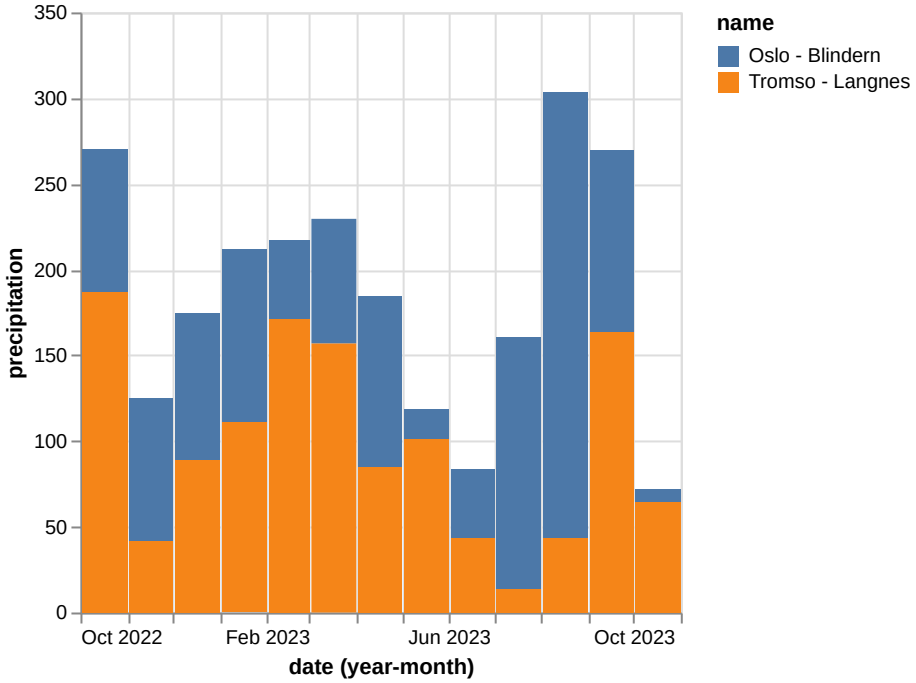
- x-coordinate with “date”
- y-coordinate with “precipitation”
- color with “name” (name of weather station; city)

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:

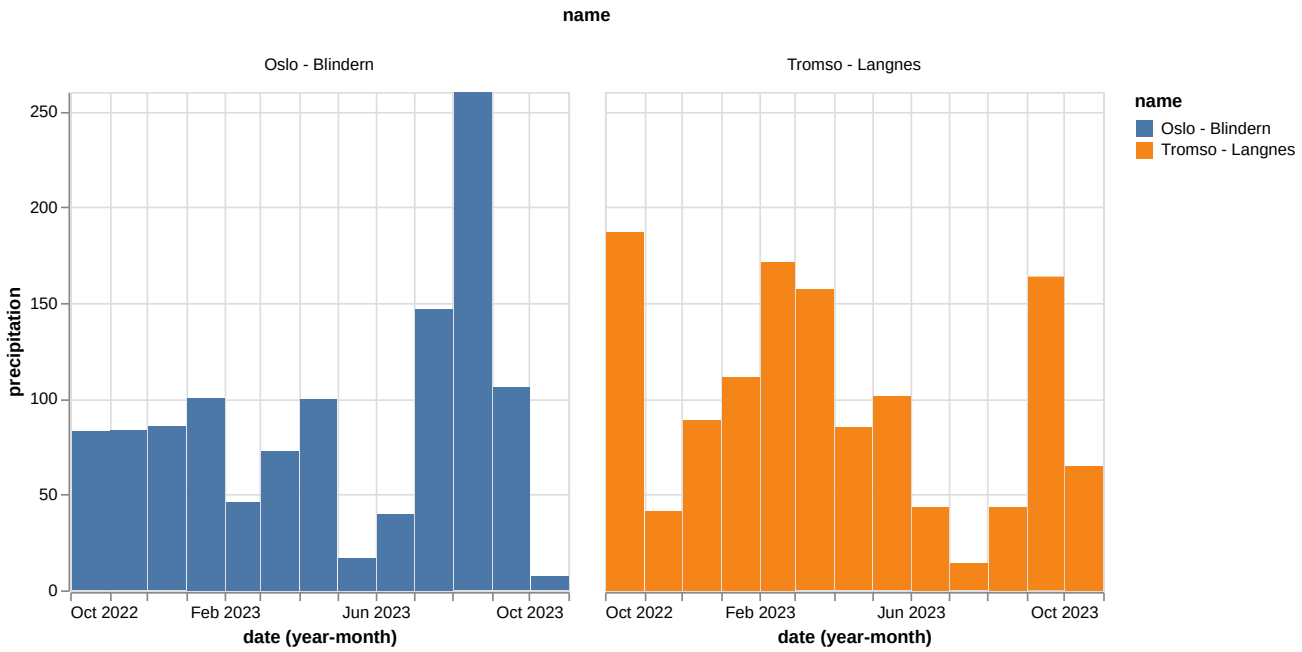
- Q (quantitative)
- O (ordinal)
- N (nominal)
- T (temporal)
- G (geojson)



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

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

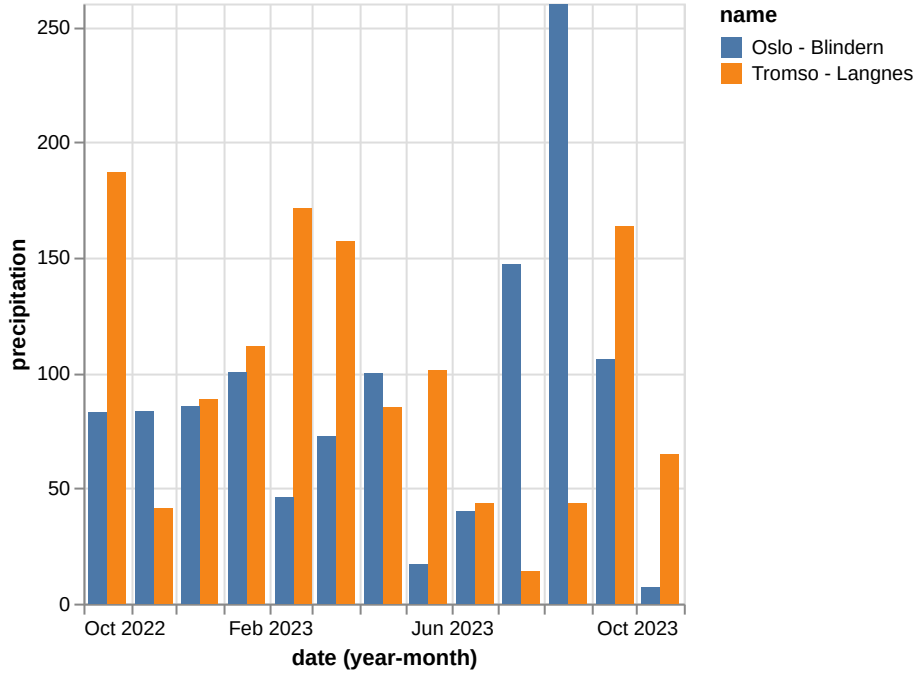
```
alt.Chart(data_monthly).mark_bar().encode(
    x="yearmonth(date):T",
    y="precipitation",
    color="name",
    column="name",
)
```



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",
)
```

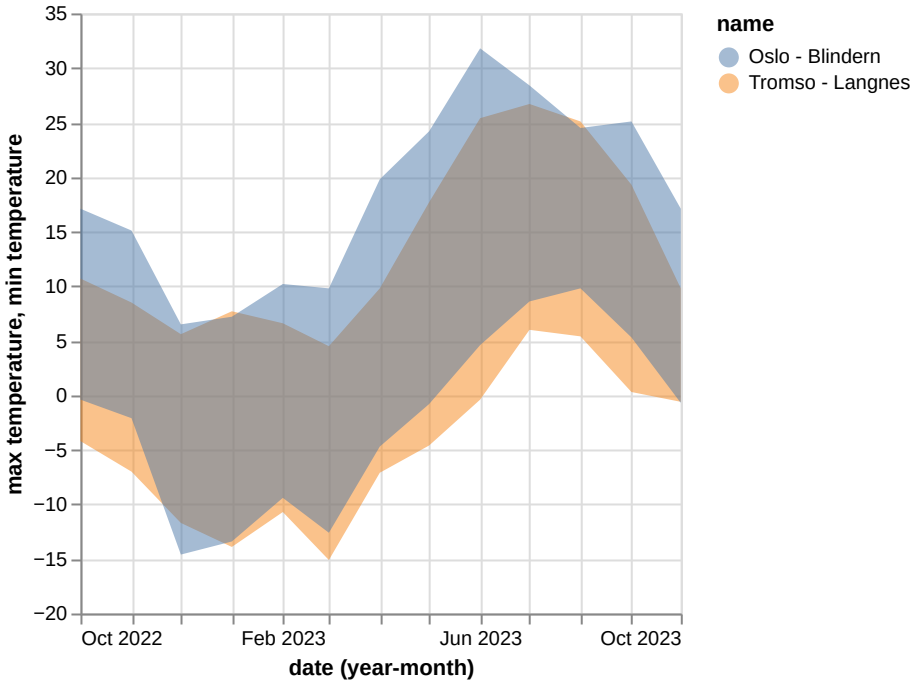


Monthly precipitation for the cities Oslo and Tromsø over the course of a year plotted as stacked bar chart.

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.

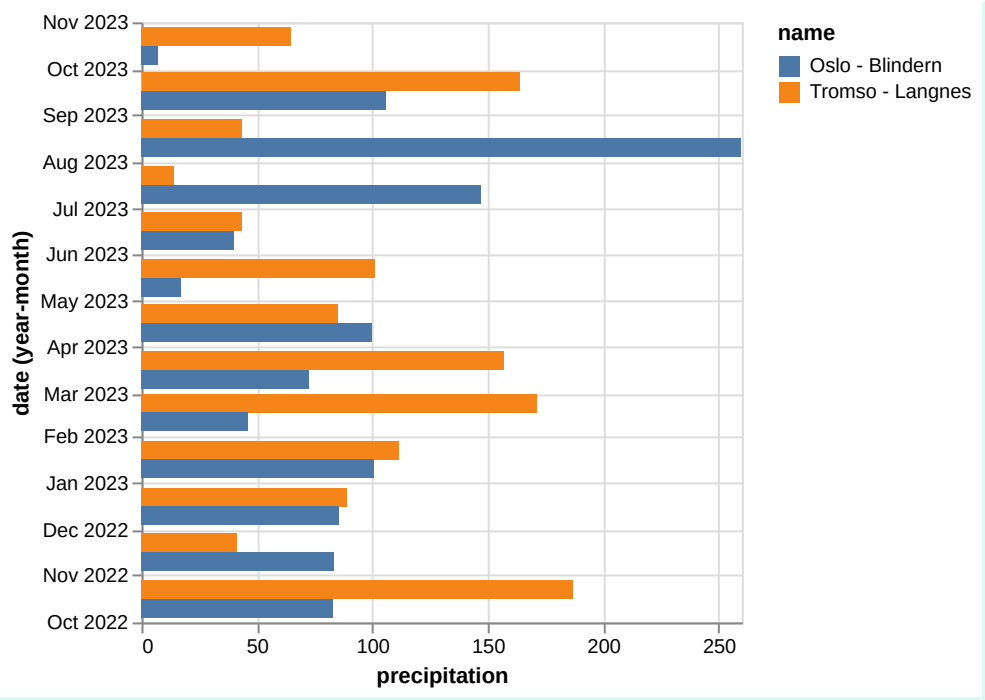
💬 What other marks and other visual channels exist?

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

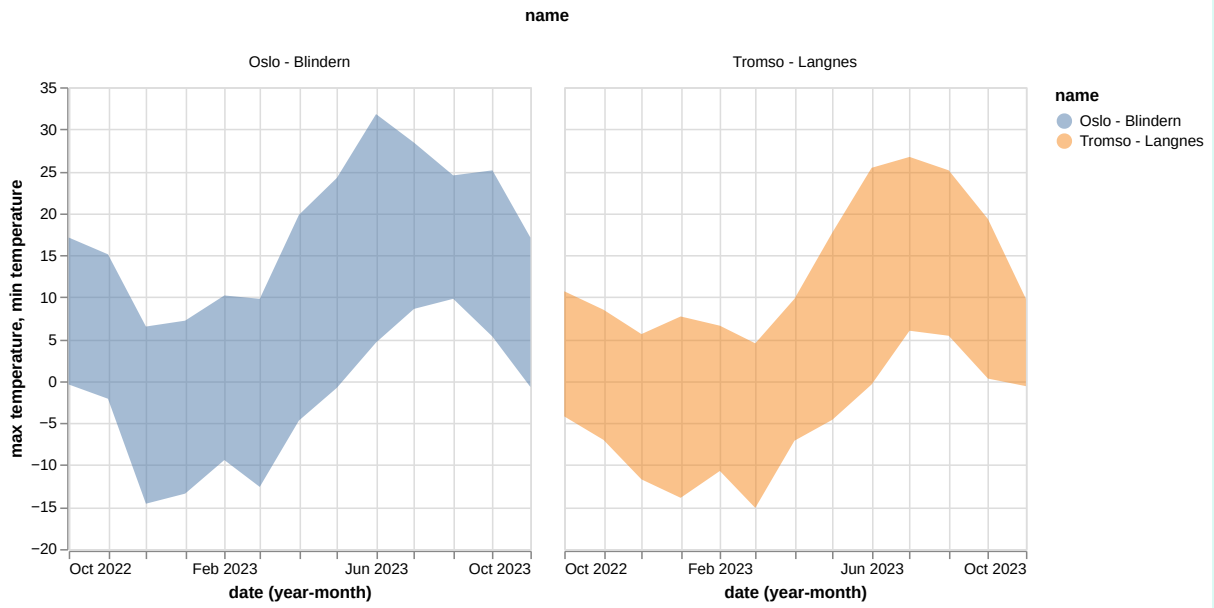
Exercise: Using visual channels to re-arrange plots

✍️ Exercise Plotting-1: Using visual channels to re-arrange plots

1. Try to reproduce the above plots if they are not already in your notebook.
2. Above we have plotted the monthly precipitation for two cities side by side using a stacked plot. Try to arrive at the following plot where months are along the y-axis and the precipitation amount is along the x-axis:



3. Ask the Internet or AI how to change the axis title from “precipitation” to “Precipitation (mm)”.
4. Modify the temperature range plot to show the temperature ranges for the two cities side by side like this:



✔ Solution

1. Copy-paste code blocks from above.
2. Basically we switched x and y:

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

3. This can be done with the following modification:

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

4. We added one line:

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

More fun with visual channels

Now we will try to **plot the daily data and look at snow depths**. We first read and concatenate two datasets:

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

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

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

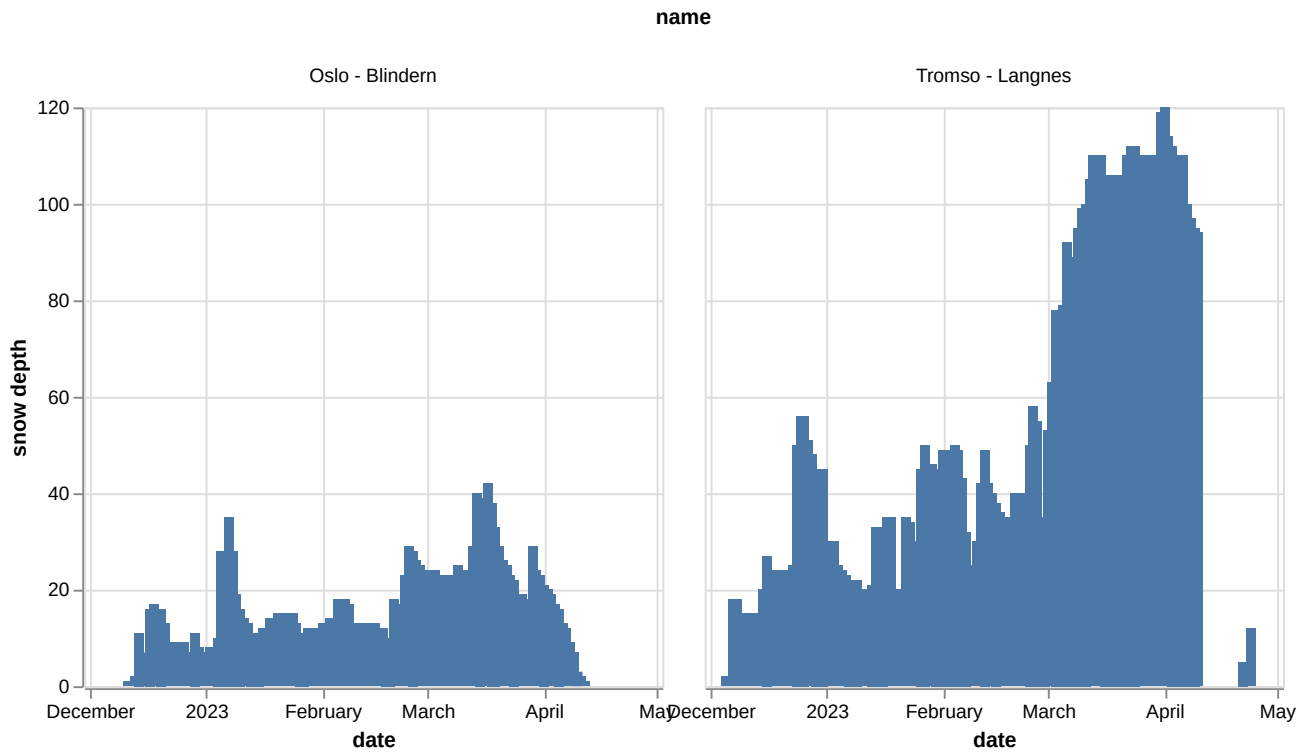
We adjust the data a bit:

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

# we are here only interested in the range december to may
data_daily = data_daily[
    (data_daily["date"] > "2022-12-01") & (data_daily["date"] < "2023-05-01")
]
```

Now we can plot the snow depths for the months December to May for the two cities:

```
alt.Chart(data_daily).mark_bar().encode(
    x="date",
    y="snow depth",
    column="name",
)
```

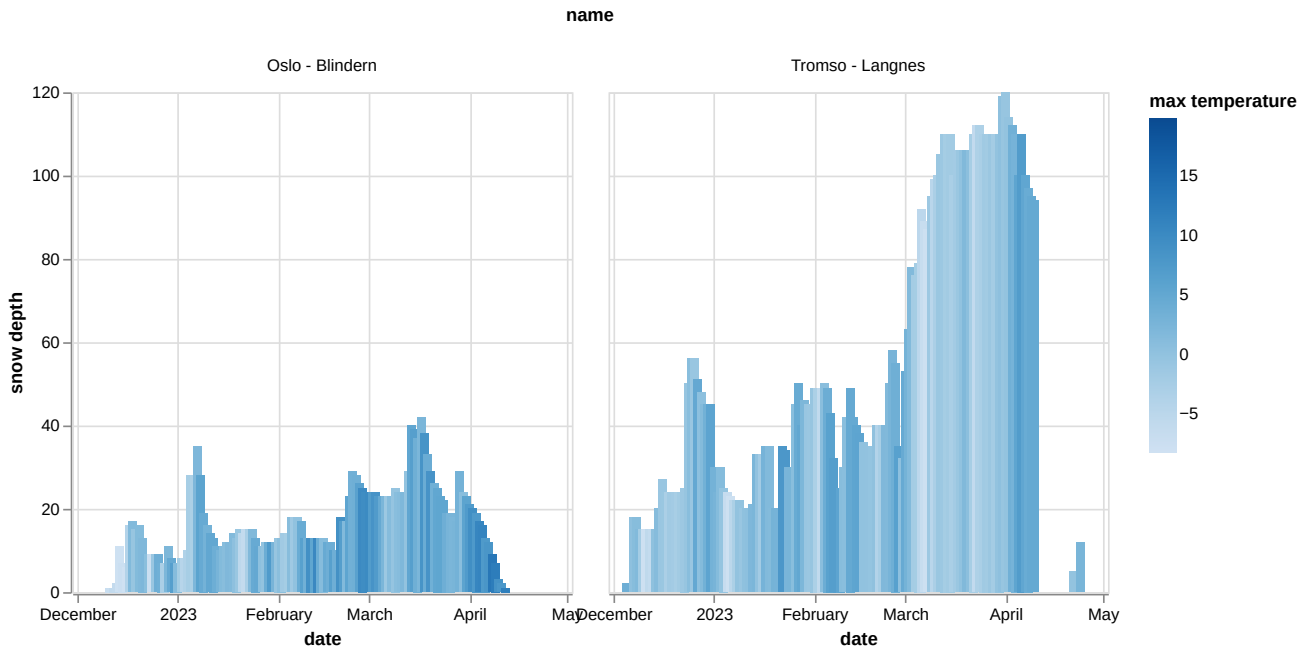


Snow depth (in cm) for the months December 2022 to May 2023 for two cities in Norway.

What happens if we try to color the plot by the “max temperature” values?

```
alt.Chart(data_daily).mark_bar().encode(
    x="date",
    y="snow depth",
    color="max temperature",
    column="name",
)
```

The result looks neat:

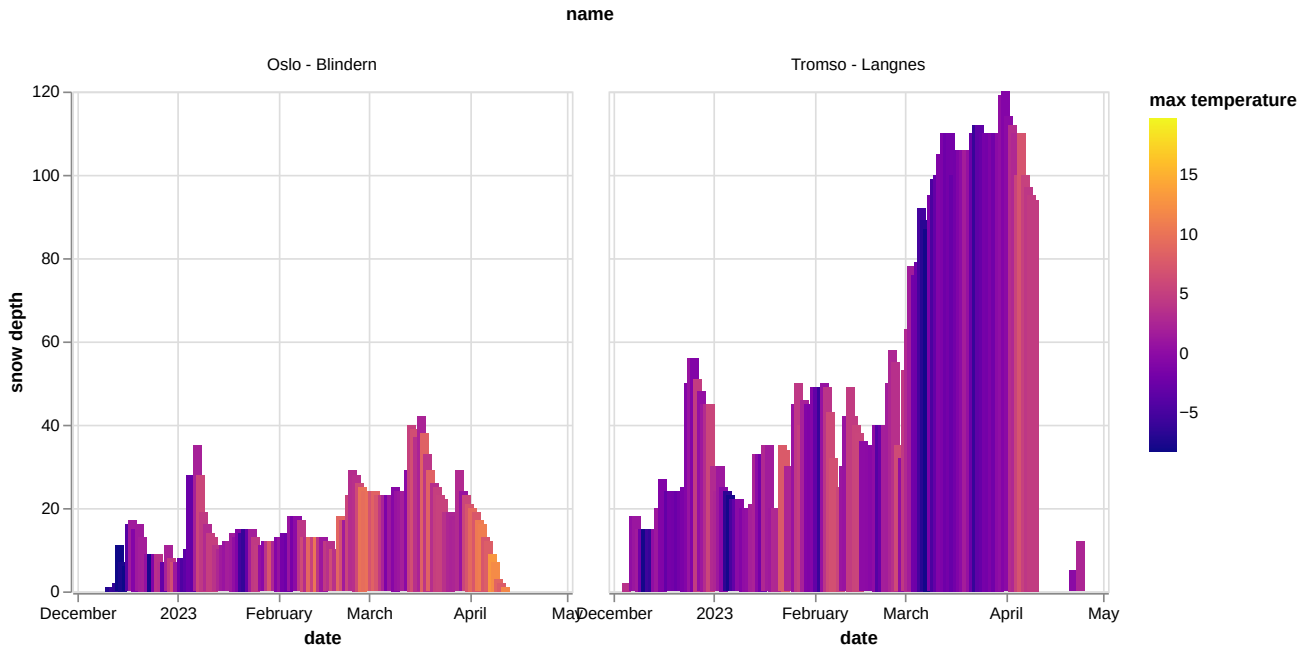


Snow depth (in cm) for the months December 2022 to May 2023 for two cities in Norway. Colored by daily max temperature.

We can change the color scheme (available color schemes):

```
alt.Chart(data_daily).mark_bar().encode(
    x="date",
    y="snow depth",
    color=alt.Color("max temperature").scale(scheme="plasma"),
    column="name",
)
```

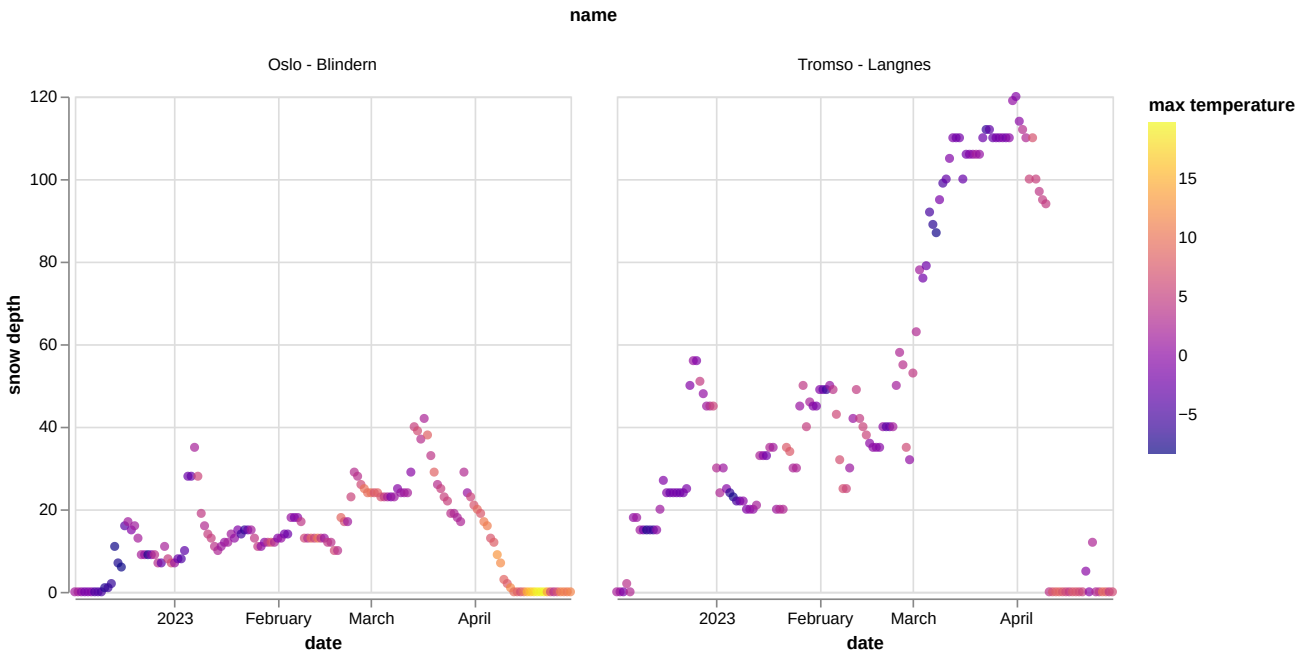
With the following result:



Snow depth (in cm) for the months December 2022 to May 2023 for two cities in Norway. Colored by daily max temperature. Warmer days are often followed by reduced snow depth.

Let's try one more change to show that we can experiment with different plot types by changing `mark_bar()` to something else, in this case `mark_circle()`:

```
alt.Chart(data_daily).mark_circle().encode(
    x="date",
    y="snow depth",
    color=alt.Color("max temperature").scale(scheme="plasma"),
    column="name",
)
```



Snow depth (in cm) for the months December 2022 to May 2023 for two cities in Norway. Colored by daily max temperature. Warmer days are often followed by reduced snow depth.

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!

Exercise: Anscombe's quartet

Save the following data as `example.csv` (you can do this directly from JupyterLab; this data is the Anscombe's quartet):

```
dataset,x,y
I,10.0,8.04
I,8.0,6.95
I,13.0,7.58
I,9.0,8.81
I,11.0,8.33
I,14.0,9.96
I,6.0,7.24
I,4.0,4.26
I,12.0,10.84
I,7.0,4.82
I,5.0,5.68
II,10.0,9.14
II,8.0,8.14
II,13.0,8.74
II,9.0,8.77
II,11.0,9.26
II,14.0,8.1
II,6.0,6.13
II,4.0,3.1
II,12.0,9.13
II,7.0,7.26
II,5.0,4.74
III,10.0,7.46
III,8.0,6.77
III,13.0,12.74
III,9.0,7.11
III,11.0,7.81
III,14.0,8.84
III,6.0,6.08
III,4.0,5.39
III,12.0,8.15
III,7.0,6.42
III,5.0,5.73
IV,8.0,6.58
IV,8.0,5.76
IV,8.0,7.71
IV,8.0,8.84
IV,8.0,8.47
IV,8.0,7.04
IV,8.0,5.25
IV,19.0,12.5
IV,8.0,5.56
IV,8.0,7.91
IV,8.0,6.89
```

✍ Exercise Plotting-2: Read and plot a CSV file

- Save the above CSV file to disk as `example.csv` in the same folder where you run JupyterLab. We recommend to create the file in the JupyterLab interface.
- Plot the data using `mark_point`.
- Your goal is to arrive at four plots for the four data sets, all side by side.
- If you have time, try to customize the plot.

✓ Solution

```
# we don't need to import again but just in case you started here
import pandas as pd

data = pd.read_csv("example.csv")

alt.Chart(data).mark_point().encode(
    x="x",
    y="y",
    color="dataset",
    column="dataset",
    )
```

Here is a more advanced example where the four plots are arranged in a 2 x 2 grid:

```
def create_chart(data, number):
    chart = (
        alt.Chart(data)
        .transform_filter(alt.datum.dataset == number)
        .mark_point()
        .encode(x="x", y="y")
    )
    return chart


chart1 = create_chart(data_example, "I")
chart2 = create_chart(data_example, "II")
chart3 = create_chart(data_example, "III")
chart4 = create_chart(data_example, "IV")

chart = alt.vconcat(
    alt.hconcat(chart1, chart2),
    alt.hconcat(chart3, chart4),
)

chart.display()
```

❗ 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.

Learning how to adapt existing gallery examples

In this exercise we can try to adapt existing scripts to either **tweak how the plot looks** or to **modify the input data**. This is very close to real life: there are so many options and possibilities and it is almost impossible to remember everything so this strategy is useful to practice:

- Select an example that is close to what you have in mind
- Being able to adapt it to your needs
- Being able to search for help

✍️ Exercise: Adapting a gallery example

This is a great exercise which is very close to real life.

- Browse the Vega-Altair example gallery.
- Select one example that is close to your current/recent visualization project or simply interests you.
- First try to reproduce this example, as-is, in the Jupyter Notebook.
- **If you get the error "ModuleNotFoundError: No module named 'vega_datasets'", then try one of these examples:** (they do not need the "vega_datasets" module)
  - Slider cutoff (**below you can find a walk-through for this example**)
  - Multi-Line tooltip
  - Heatmap
  - Layered histogram
- Then try to print out the data that is used in this example just before the call of the plotting function to learn about its structure. Consider writing the data to file before changing it.
- Then try to modify the data a bit.
- If you have time, try to feed it different, simplified data. **This will be key for adapting the examples to your projects.**

✓ Example walk-through for the slider cutoff example

In this walk-through I imagine browsing: https://altair-viz.github.io/gallery/index.html

Then this example caught my eye: https://altair-viz.github.io/gallery/slider_cutoff.html

I then copy-paste the example code into a notebook and try to run it and I get the same result.

Next, there is a lot of code that I don't (need to) understand yet but my eyes are trying to find `alt.Chart` which tells me that the data must be the "df" in `alt.Chart(df)`:

```
import altair as alt
import pandas as pd
import numpy as np

rand = np.random.RandomState(42)

df = pd.DataFrame({
    'xval': range(100),
    'yval': rand.randn(100).cumsum()
})

slider = alt.binding_range(min=0, max=100, step=1)
cutoff = alt.param(bind=slider, value=50)

alt.Chart(df).mark_point().encode(
    x='xval',
    y='yval',
    color=alt.condition(
        alt.datum.xval < cutoff,
        alt.value('red'), alt.value('blue')
    )
).add_params(
    cutoff
)
```

My next step will be to print out the data `df` just before the call to `alt.Chart`:

```
import altair as alt
import pandas as pd
import numpy as np

rand = np.random.RandomState(42)

df = pd.DataFrame({
    'xval': range(100),
    'yval': rand.randn(100).cumsum()
})

slider = alt.binding_range(min=0, max=100, step=1)
cutoff = alt.param(bind=slider, value=50)

print(df)

alt.Chart(df).mark_point().encode(
    x='xval',
    y='yval',
    color=alt.condition(
        alt.datum.xval < cutoff,
        alt.value('red'), alt.value('blue')
    )
).add_params(
    cutoff
    )
```

The print reveals that `df` is a dataframe which contains x and y values:

```
    xval       yval
0      0   0.496714
1      1   0.358450
2      2   1.006138
3      3   2.529168
4      4   2.295015
..   ...        ...
95    95 -10.712354
96    96 -10.416233
97    97 -10.155178
98    98 -10.150065
99    99 -10.384652

[100 rows x 2 columns]
```

The next thing that often helps me is to save the data to a comma-separated values (CSV) file:

```
import pandas as pd

df.to_csv("data.csv", index=False)
```

I then open the file in an editor and see that it contains 100 rows:

```
xval,yval
0,0.4967141530112327
1,0.358449851840048
2,1.0061383899407406
3,2.5291682463487657
4,2.2950148716254297
5,2.060877914676249
6,3.6400907301836405
7,4.407525459336549
8,3.938051073401597
9,4.4806111169875615
...
```

Saving the data to file often helps me to see the structure of the data and now I am in a position to replace this with my own data. I create a file called “mydata.csv” and there I use the maximum temperatures for months 1-10 from the Tromso monthly data which we used further up:

```
xval,yval
01,7.7
02,6.6
03,4.5
04,9.8
05,17.7
06,25.4
07,26.7
08,25.1
09,19.3
10,9.8
```

In the notebook I then verify that the reading of the data works:

```
mydata = pd.read_csv("mydata.csv")

mydata
```

Now I can replace the example with my own data (note how I now can comment out some code that I don't need any longer):

```
import altair as alt
import pandas as pd
# import numpy as np

# rand = np.random.RandomState(42)

# df = pd.DataFrame({
#     'xval': range(100),
#     'yval': rand.randn(100).cumsum()
# })

slider = alt.binding_range(min=0, max=100, step=1)
cutoff = alt.param(bind=slider, value=50)

# print(df)
df = pd.read_csv("mydata.csv")

alt.Chart(df).mark_point().encode(
    x='xval',
    y='yval',
    color=alt.condition(
        alt.datum.xval < cutoff,
        alt.value('red'), alt.value('blue')
    )
).add_params(
    cutoff
    )
```

Seems to work! I then make few more adjustments (I want the slider to work on the y-axis and have a more reasonable default):

```
import altair as alt
import pandas as pd

slider = alt.binding_range(min=0, max=30, step=1)
cutoff = alt.param(bind=slider, value=15)

df = pd.read_csv("mydata.csv")

alt.Chart(df).mark_point().encode(
    x='xval',
    y='yval',
    color=alt.condition(
        alt.datum.yval < cutoff,
        alt.value('red'), alt.value('blue')
    )
).add_params(
    cutoff
    )
```

My next steps would then be to change axis titles, display the month names, add a legend, and refine from here.

From notebooks to scripts

Objectives

- Understand when notebooks are not useful anymore and when to start using scripts
- Be able to use `nbconvert` or other tools to convert a notebook to a script
- Run a script from the command line
- Reflect on the advantages of scripts over notebooks

Why make the change?

Though we have seen that notebooks are very practical and useful for many tasks, such as testing out analyses, sharing results, teaching, and more, there are some cases where it is better to use scripts.

We will all at one point find out that notebooks are not the best tool for every use. Some cases where a script would do a better job are:

- You want to run an analysis at a specific time every day, with a scheduler of some kind
- Your notebook has exceeded your laptops capabilities and you want to migrate your workflow to a supercomputer with a scheduler such as SLURM
- You run a complex process in the notebook needs to be optimized and profiled with the relevant tools (We will look at this more later in this course)

Although scripts are not the perfect tool either, they are still a powerful tool and allow for more flexibility and control.

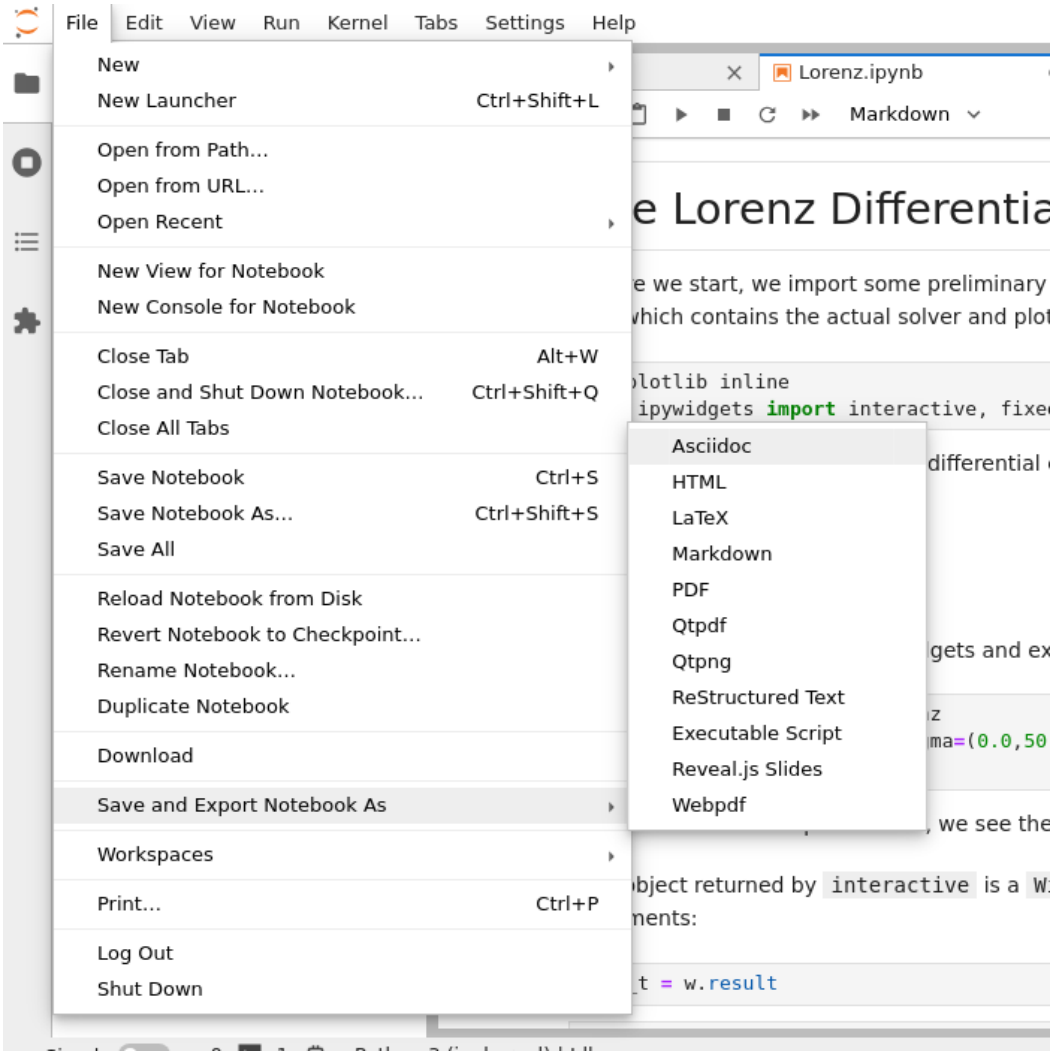
Converting a notebook to a script

There are several ways to make a notebook into a command line script. One such way is `nbconvert`, which is a tool that comes with Jupyter. Check the `JupyterLab documentation <https://jupyterlab.readthedocs.io/en/stable/user/export.html>` for more information.

You can get a command line in jupyter lab by (File → New Launcher → Terminal - if you go through New Launcher, your command line will be in the directory you are currently browsing), you can convert files in the terminal by running:

```
jupyter nbconvert --to script your_notebook_name.ipynb
```

If you are struggling with the command line, you can also convert a notebook to a script by going to the notebook and clicking on `File → Save and Export Notebook As → Executable Script`. This will save the notebook in your machine, so you will have to copy it to the relevant directory (if you want it on a remote machine such as a supercomputer).



*Press in the top menu **File → Save and Export Notebook As → Executable Script** to download the notebook as a script.*

✍ Exercise Scripts-1: Convert a notebook to a script (20 mins)

1. Download the following notebook by right-clicking on this link and selecting “Save Link As…” to save the file to your local machine.
2. Convert the notebook to a script using the terminal following the instructions above.

```
jupyter nbconvert --to script weather_observations.ipynb
```

3. Run the generated script in the terminal with:

```
python weather_observations.py
```

Discussion:

- What changed when you converted the notebook to the script?
- What would have happened if the notebook was not written with linear execution in mind?
- what would you have to do if you wanted to change the date range for the plotting?
- Is this something you think you can use in your work?

There are some other ways of executing your notebook as a script, such as papermill, though we will not go through these today.

We can now move on to working with command line arguments in scripts, which is where scripts really shine.

Command-line interfaces (CLI)

Objectives

- Understand the basics of command-line interfaces
- Learn how to write code that can change its behaviour based on command-line arguments

In the previous section, we learned how to convert a Jupyter notebook into a Python script that is executable from the command line. Though we can now run it in the command line, every time we want to change something in the analysis we would need to edit the script.

In this section we will learn how to change the behaviour of a script based on command-line arguments. This will allow us to run the script with different parameters without having to edit the script every time.

This is useful when you want to share your script with other people, as they wont necesarilly need to know what is inside the script to run it.

Command-line arguments with `argparse`

Command-line arguments are parameters that are passed to a script when running it in the command line. An example of this is:

```
python my_script.py arg1 arg2
```

In this case, `arg1` and `arg2` are the command-line arguments that are passed to the script `my_script.py`.

`argparse` is a Python module that makes it easy to write descriptive command-line arguments, and it also automatically writes useful `--help` sections for your script.

`argparse` defines two types of arguments:

- Positional arguments: these are required arguments that are passed to the script in a specific order.
- Optional arguments: these are arguments that are not required and can be passed in any order.

To use argparse you first set up a parser by calling `parser = argparse.ArgumentParser()` and then you add arguments using `parser.add_argument(args)`.

Lets write an example script that takes in a name and a birth date as arguments and prints them out.

```
import argparse

# Initialize the parser
parser = argparse.ArgumentParser()

# One positional and one optional argument
parser.add_argument('name', type=str, metavar="N",
                    help="The name of the subject")
parser.add_argument('-d', '--date', type=string, default="01/01/2000",
                    help="Birth date of the subject")

# Parse the arguments and collect them in args
args = parser.parse_args()

print(args.name + " was born on " + args.date)
```

when this is run you have to pass in the name and the date as arguments in this way:

```
python birthday.py John -d 01/01/1990
```

If you run the script without the date argument, it will default to the date specified in the `add_argument` function.

If you run the script with the `--help` flag, you will get a description of the arguments that the script takes in:

```
python birthday.py --help
```

which would return:

```
usage: birthday.py [-h] [-d DATE] N

positional arguments:
  N                     The name of the subject

optional arguments:
  -h, --help            show this help message and exit
  -d DATE, --date DATE  Birth date of the subject
```

Lets now try

✍ Exercise Scripts-2: Add command-line arguments to a script (15 mins)

Use the example above edit the script `weather_observations.py` from the previous exercise to take in the date range as arguments using `argparse`.

- Hint: The script should be able to be run like this:

  ```
  python weather_observations.py --start 2020-01-01 --end 2020-12-31
  ```

- Hint: try not to do it all at once, but add one or two arguments, test, then add more, and so on.
- Hint: The input and output filenames make sense as positional arguments, since they must always be given. Input is usually first, then output.
- Hint: The start and end dates should be optional parameters with the defaults as they are in the current script.

Discussion:

- What was the main challenge in adding the arguments?

- What would you have to do if you wanted to add more arguments? or a new analysis?
- How would you work with the arguments in for example a slurm submit script?

This is not the only way to add command-line arguments to a script. We encourage you to explore other ways to do this, such as using `sys.argv`, `doctopt`, `typer` or `click`.

Good practices and tools

Objectives

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

Follow the PEP 8 style guide

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

Linting and static type checking

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

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

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

💬 **Linters and formatters can be configured to your liking**

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

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

```
import re
import requests


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

Please try whether you can locate these problems using Ruff:

```
$ ruff check
```

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

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

Use an auto-formatter

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

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

Badly formatted | Auto-formatted

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

Other popular formatters:

- Black
- YAPF

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

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

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

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

Consider annotating your functions with type hints

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

Without type hints | With type hints

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

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

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

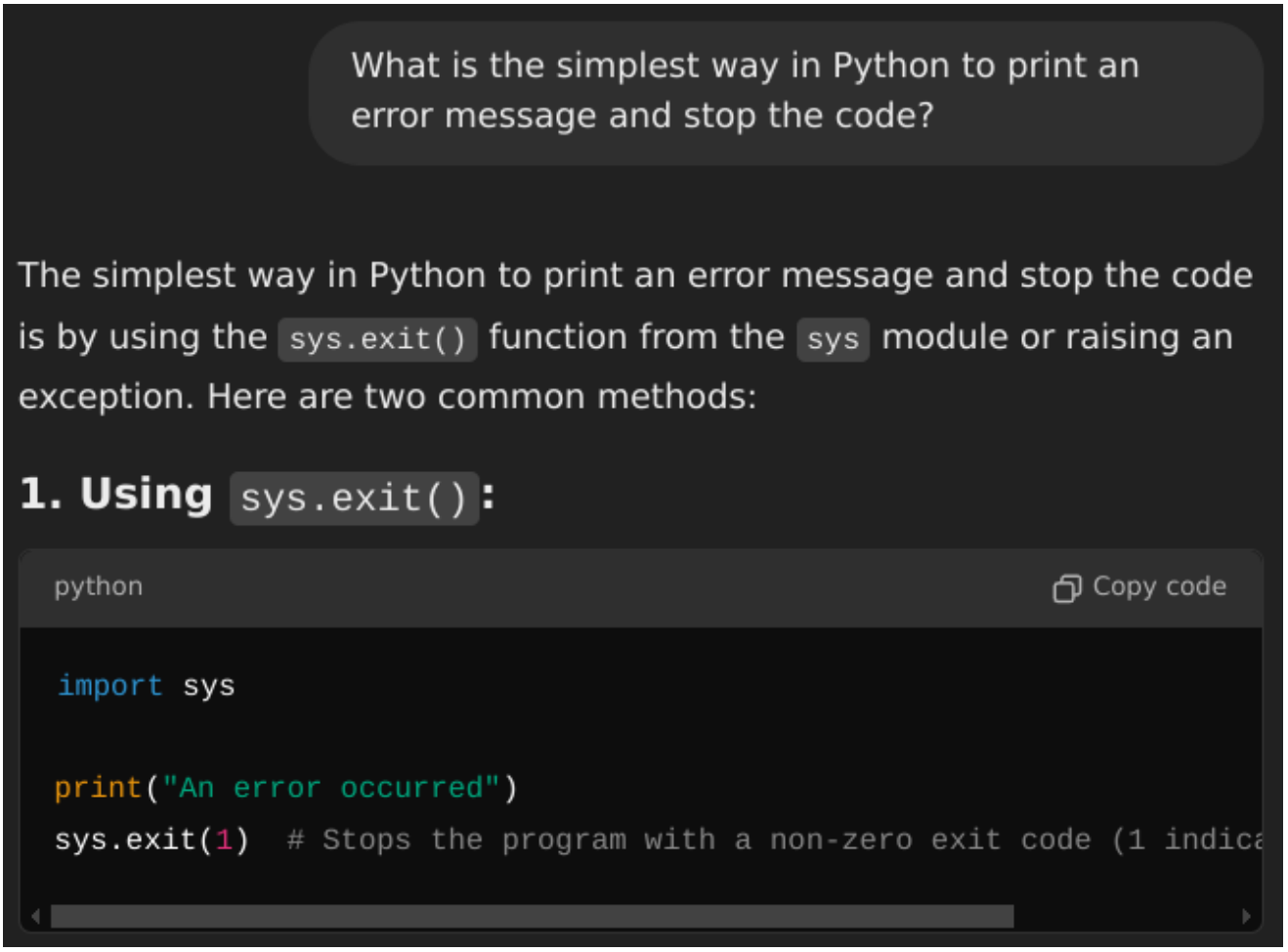
Consider using AI-assisted coding

We can use AI as an assistant/apprentice:

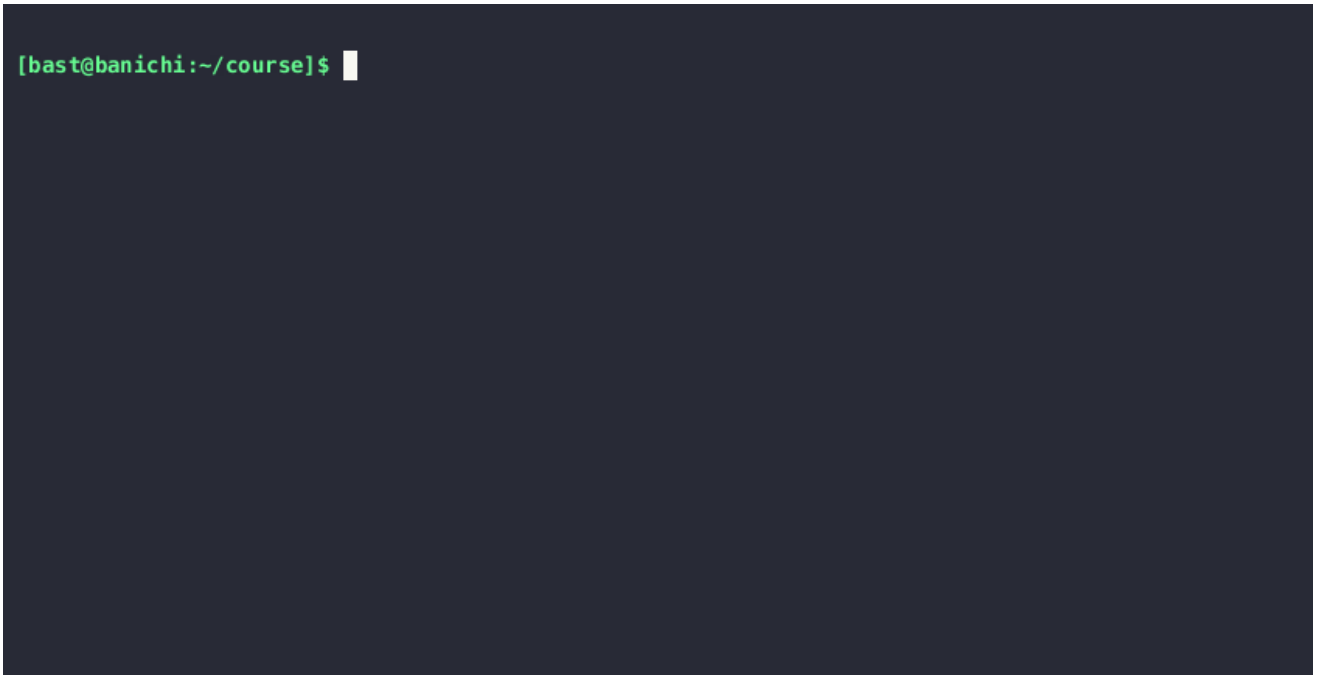
- Code completion
- Write a test based on an implementation
- Write an implementation based on a test

Or we can use AI as a mentor:

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



Example for using a chat-based AI tool.



Example for using AI to complete code in an editor.

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

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

Debugging with print statements

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

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

Or more elaborate:

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

But there can be better alternatives:

- Logging module

```
import logging

logging.basicConfig(level=logging.DEBUG)

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

- IceCream offers compact helper functions for print-debugging

```
from icecream import ic

ic(file_path)
```

Often you can avoid using indices

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

Iterating

Verbose and can be brittle | Better

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

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

Enumerate if you need the index

Verbose and can be brittle | Better

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

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

Zip if you need to iterate over two collections

Using an index can be brittle | Better

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

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

Unpacking

Verbose and can be brittle | Better

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

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

Prefer catch-all unpacking over indexing/slicing

Verbose and can be brittle | Better

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

sorted_scores = sorted(scores)

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

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

List comprehensions, map, and filter instead of loops

For-loop | List comprehension | Map

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

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

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

For-loop | List comprehension | Filter

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


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

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

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

Know your collections

How to choose the right collection type:

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

What to avoid:

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

Making functions more ergonomic

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

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

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

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

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

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

Iterating

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

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

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

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

Dataclasses

Dataclasses are often a good alternative to regular classes:

Regular class | Dataclass | Named tuple

```
class Point:
    def __init__(self, x, y, z):
        self.x = x
        self.y = y
        self.z = z

    def __repr__(self):
        return f"Point(x={self.x}, y={self.y}, z={self.z})"

    def __eq__(self, other):
        if not isinstance(other, Point):
            return NotImplemented
        return self.x == other.x and self.y == other.y and self.z == other.z
```

Use relative paths and pathlib

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

Project structure

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

```
def main():
    ...

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

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

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

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

Reading and writing files

- Good construct to know to read a file:

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

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

Use subprocess instead of os.system

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

Parallelizing

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

Version control (motivation)

Objectives

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

Why do we need to keep track of versions?

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

- "It broke ... hopefully I have a working version somewhere?"
- "Can you please send me the latest version?"
- "Where is the latest version?"

- “Which version are you using?”
- “Which version have the authors used in the paper I am trying to reproduce?”
- “Found a bug! Since when was it there?”
- “I am sure it used to work. When did it change?”
- “My laptop is gone. Is my thesis now gone?”

Demonstration

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

Features: roll-back, branching, merging, collaboration

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

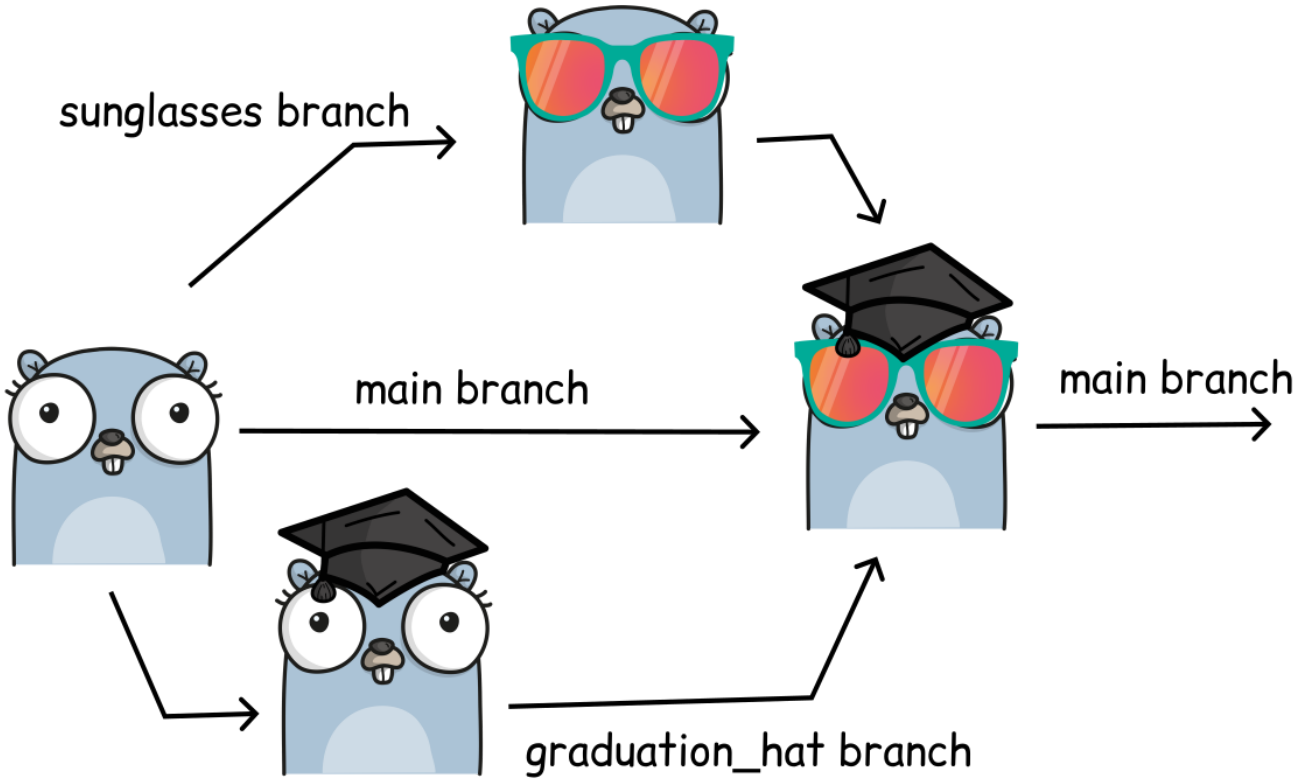


Image created using https://gopherize.me/ (inspiration).

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

Talking about code

Which of these two is more practical?

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

What we typically like to snapshot

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

💬 Discussion

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

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

✓ Solution

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

Where to learn more

CodeRefinery lessons with **focus on collaboration** and **not only for the command line**:

- Introduction to version control (day 1-2): **Why we want to track versions and how to go back in time to a working version.** This lesson brings you from zero to using Git and GitHub for own projects.
- Collaborative distributed version control (day 3): This lesson builds on "Introduction to version control" and we apply branching and learn about pull requests (merge requests), forks, and collaboration using Git and GitHub.

Reproducible environments and dependencies

Objectives

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

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

Essential XKCD comics:

- xkcd - dependency
- xkcd - superfund

How to avoid: "It works on my machine 🤷"

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

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

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

PyPI

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

Conda

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

Conda ecosystem explained

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

  Main points of the article are:

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

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

- Major repositories/channels:

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

Tools and distributions for dependency management in Python

- Poetry: Dependency management and packaging.
- Pipenv: Dependency management, alternative to Poetry.
- pyenv: If you need different Python versions for different projects.
- virtualenv: Tool to create isolated Python environments for PyPI packages.
- micropipenv: Lightweight tool to "rule them all".
- Conda: Package manager for Python and other languages maintained by Anaconda Inc.
- Miniconda: A "miniature" version of conda, maintained by Anaconda Inc. By default uses Anaconda's channels. Check licensing terms when using these packages.
- Mamba: A drop in replacement for conda. It used be much faster than conda due to better dependency solver but nowadays conda also uses the same solver. It still has some UI improvements.
- Micromamba: Tiny version of the Mamba package manager.
- Miniforge: Open-source Miniconda alternative with conda-forge as the default channel and optionally mamba as the default installer.
- Pixi: Modern, super fast tool which can manage conda environments.
- uv: Modern, super fast replacement for pip, poetry, pyenv, and virtualenv. You can also switch between Python versions.

Best practice: Install dependencies into isolated environments

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

Keypoints

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

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

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

You can export ("freeze") the dependencies from your current environment into these files:

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

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

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

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

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

Containers

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

Containers offer the following advantages:

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

How to install dependencies into environments

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

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

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

This command did two things:

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

For your own projects:

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

environment.yml | requirements.txt

```yaml
name: course
channels:
  - conda-forge
dependencies:
  - python <= 3.12
  - jupyterlab
  - altair-all
  - vega_datasets
  - pandas
  - numpy
  - pytest
  - scalene
  - ruff
  - icecream
  - myst-parser
  - sphinx
  - sphinx-rtd-theme
  - sphinx-autoapi
  - sphinx-autobuild
```

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

Miniforge | Pixi | Virtual environment | uv

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

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

  Or equivalently:

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

- Activate the environment:

  ```
  $ conda activate myenv
  ```

- Run your code inside the activated virtual environment.

  ```
  $ python example.py
  ```

Updating environments

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

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

Miniforge | Pixi | Virtual environment | uv

- Update the environment by running:

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

- Or equivalently:

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

Pinning package versions

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

- Both `pip` and `conda` ecosystems and all the tools that we have mentioned support pinning versions.
- It is possible to define a range of versions instead of precise versions.
- While your project is still in progress, I often use latest versions and do not pin them.
- When publishing the script or notebook, it is a good idea to pin the versions to ensure that the code can be run in the future.
- Remember that at some point in time you will face a situation where newer versions of the dependencies are no longer compatible with your software. At this point you'll have to update your software to use the newer versions or to lock it into a place in time.

Managing dependencies on a supercomputer

- Additional challenges:
  - Storage quotas: **Do not install dependencies in your home directory**. A conda environment can easily contain 100k files.
  - Network file systems struggle with many small files. Conda environments often contain many small files.
- Possible solutions:

- Try Pixi (modern take on managing Conda environments) and uv (modern take on managing virtual environments). Blog post: Using Pixi and uv on a supercomputer
- Install your environment on the fly into a scratch directory on local disk (**not** the network file system).
- Install your environment on the fly into a RAM disk/drive.
- Containerize your environment into a container image.

Keypoints

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

Where to start with documentation

Objectives

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

Why? 💗✉️ to your future self

- You will probably use your code in the future and may forget details.
- You may want others to use your code or contribute (almost impossible without documentation).

In-code documentation

Not very useful (more commentary than comment):

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

More useful (explaining **why**):

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

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

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

Emulating version control:

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

Many languages allow “docstrings”

Example (Python):

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

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

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

    temp_c = temp_k - 273.15

    return temp_c
```

Keypoints

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

Often a README is enough - checklist

- **Purpose**

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

See also the JOSS review checklist.

Diátaxis

Diátaxis is a systematic approach to technical documentation authoring.

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

What if you need more than a README?

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

Setting up a Sphinx documentation

⚙ Preparation

In this episode we will use the following 5 packages which we installed previously as part of the Software install instructions:

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

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

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

✍️ Exercise: Set up a Sphinx documentation

1. Create the following three files in your project:

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

This is `conf.py`:

```
project = "your-project"
copyright = "2025, Authors"
author = "Authors"
release = "0.1"

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

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

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

html_theme = "sphinx_rtd_theme"
```

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

```
# Our code documentation

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

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

about.md
:::
```

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

```
# About this code

Work in progress ...
```

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

```
$ sphinx-build doc _build

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

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

   - typography
   - images
   - math and equations
   - code blocks

There is a lot more you can do:

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

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

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

How to auto-generate API documentation in Python

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

```
project = "your-project"
copyright = "2025, Authors"
author = "Authors"
release = "0.1"

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

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

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

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

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

html_theme = "sphinx_rtd_theme"
```

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

Possibilities to host Sphinx documentation

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

Confused about reStructuredText vs. Markdown vs. MyST?

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

Where to read more

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

Profiling

Objectives

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

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

Should we even optimize the code?

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

Discussion

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

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

Depends. What does it depend on?

Measure instead of guessing

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

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

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

One of the simplest tools is to insert timers

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

```python
import time


# ...
# code before the function


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


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

Many tools exist

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

CPU profilers:

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

Memory profilers:

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

Both CPU and memory:

- Scalene

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

Tracing profilers vs. sampling profilers

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

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

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

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

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

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

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

Choosing the right system size

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

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

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

Exercises

✍️ Exercise: Practicing profiling

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

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

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

   ```
   $ scalene example.py
   ```

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

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

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

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

Example code (`example.py`):

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


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


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


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


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


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

✔ Solution

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

Results:

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

Explanation:

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

What we can learn from this exercise:

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

Additional resources

- Python performance workshop (by ENCCS)

Automated testing

Objectives

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

Motivation

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

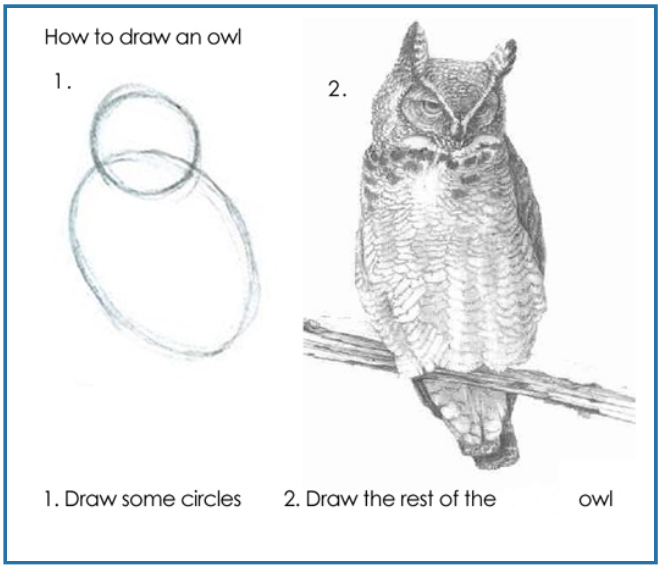
- **Less scary to change code**: tests will tell you whether something broke.
- **Easier for new people** to join.
- Easier for somebody to **revive an old code**.
- **End-to-end test**: run the whole code and compare result to a reference.
- **Unit tests**: test one unit (function or module). Can guide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

How testing is often taught

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


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

How this feels:



[Citation needed]

Where to start

Do I even need testing?:

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

If you have nothing yet:

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

If you want to start with unit-testing:

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

Pytest

Here is a simple example of a test:

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


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

To run the test(s):

```
$ pytest example.py
```

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

Good practice to add unit tests:

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

What else is possible

- Run the test set **automatically** on every code change:
  - GitHub Actions
  - GitLab CI
- The testing above used **example-based** testing.
- **Test coverage**: how much of the code is traversed by tests?
  - Python: pytest-cov
  - Result can be deployed to services like Codecov or Coveralls.
- **Property-based** testing: generates arbitrary data matching your specification and checks that your guarantee still holds in that case.
  - Python: hypothesis
- **Snapshot-based** testing: makes it easier to generate snapshots for regression tests.
  - Python: syrupy
- **Mutation testing**: tests pass -> change a line of code (make a mutant) -> test again and check whether all mutants get “killed”.
  - Python: mutmut

Further reading

- CodeRefinery lesson about automated testing

Credit

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

- https://aaltoscicomp.github.io/python-for-scicomp/
- https://coderefinery.github.io/data-visualization-python/
- https://coderefinery.github.io/reproducible-python/
- https://coderefinery.github.io/jupyter/