Introduction to Programming for Data Science

In this introductory-level workshop, we will learn the very basics of programming for data science. What does it mean to code? To “run a script”? Variables and functions?

We will work in Jupyter Notebooks for its simplicity to get started writing code even without any previous experience. We will learn how to produce some **reproducible plots** using Vega-Altair and in general how to find your way with programming in Python or R.

Who is the course for?

- Somebody starting with Python or curious about Python.
- Somebody who needs to read, process, and plot data for their work or studies and would like to try it out with Python.

Preparations

- No programming language experience needed, we will start from zero and learn the basics together
- Computer with network access
- Software install instructions

What is not taught?

- Version control. Although super useful it is outside of this workshop.
- Python outside a Jupyter Notebook (e.g. on terminal, or VScode, or spyder)
- Python sets and tuples are only mentioned.
- File input/output is only used via libraries and doing “own” file-I/O is only part of optional material.
- How to choose the right visualization format for the data at hand.
- Python object oriented design.
- Python packaging.
- NumPy arrays.
- Managing environments and installing Python packages.

Episode overview

Introductory workshop (3h):

- Basics of Programming
- Jupyter Notebooks view on nbviewer Open in Colab launch binder
- Python basics
- Plotting with Vega-Altair view on nbviewer Open in Colab launch binder
- Plotting with ggplot2

More advanced topics:

- More Plotting with Vega-Altair
- Tidy data and dealing with messy data
- Customizing plots view on nbviewer Open in Colab launch binder
- Learning how to adapt existing gallery examples
- Sharing plots and notebooks

Other:

- Software install instructions

Basics of Programming

Objectives

- Understand what programming is
- Learn about variables and data types
- Understand conditional statements (`if` statements)
- Explore arrays and collections
- Understand loops and iteration
- Learn about functions
- Familiarize with basic operators

What is Programming?

Programming is the process of writing instructions for a computer to execute. These instructions are written in programming languages, which computers interpret and perform specific tasks. Programming languages allow us to create software, applications, and websites.

Programming involves tasks such as:

- Creating algorithms (step-by-step instructions)
- Managing data
- Automating tasks

Variables

Variables are containers used to store data values. They have names, and values can change during program execution.

Example:

Python R Julia

```
name = "Alice"
age = 30
print(name)
print(age)
```

Variable names:

- Should be descriptive
- Can include letters, numbers, and underscores (`_`)
- Cannot begin with a number

Data Types

Data types define the type of data stored in variables. Common data types include:

- **Integer**: Whole numbers

  Python R Julia

  ```
  age = 30
  ```

- **Float**: Numbers with decimal points

  Python R Julia

  ```
  price = 9.99
  ```

- **String**: Text or characters

Python R Julia

```
greeting = "Hello World"
```

- **Boolean**: True or False values

Python R Julia

```
is_student = True
```

Operators

Operators are used to perform operations on variables and values.

Common types of operators:

- **Arithmetic Operators**: `+`, `-`, `*`, `/`, `%`

Python R Julia

```
sum = 5 + 3
```

- **Assignment Operators**: `=`, `+=`, `-=`

Python R Julia

```
x = 10
x += 5
```

- **Comparison Operators**: `==`, `!=`, `<`, `>`

  Python R Julia

  ```
  print(10 > 5)
  ```

- **Logical Operators**: `and`, `or`, `not`

  Python R Julia

  ```
  print(10 > 5 and 5 < 10)
  ```

Conditional Statements

Conditional statements perform different actions based on conditions. The `if` statement is the most common:

Python R Julia

```
age = 18

if age >= 18:
    print("You are an adult.")
else:
    print("You are not an adult.")
```

Arrays

Arrays (or lists in Python) store multiple values in a single variable:

Python R Julia

```
colors = ["red", "green", "blue"]

print(colors[0])  # Output: red
```

Arrays are useful for:

- Storing multiple related values
- Iterating through values
- Sorting and modifying collections

Loops

Loops are used to repeat actions multiple times.

For loop:

Python R Julia

```
colors = ["red", "green", "blue"]

for color in colors:
    print(color)
```

While loop:

Python R Julia

```
count = 0

while count < 3:
    print(count)
    count += 1
```

Loops help in iterating over collections and automating repetitive tasks.

Functions

Functions are reusable blocks of code that perform specific tasks:

Python R Julia

```python
def greet(name):
    print("Hello, " + name + "!")

greet("Alice")
```

Functions:

- Improve readability
- Promote code reuse
- Simplify debugging

✍️ Exercise: Running code on the terminal

Open a terminal (in a computer where Python/R/Julia are installed) and try some of the commands above.

Python R Julia

```
# in your terminal, start python by typing the command "python"
# it will look like this:

(base) enrico@computername:~$ python
Python 3.11.9 | packaged by conda-forge | (main, Apr 19 2024, 18:36:13) [GCC
12.3.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Then run some code for example with

```
(base) enrico@computername:~$ python
Python 3.11.9 | packaged by conda-forge | (main, Apr 19 2024, 18:36:13) [GCC
12.3.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> print("Hello world!")
Hello world!
>>>
```

Why is this approach unpractical? Consider the usability perspective but also the reproducibility point of view…

Learn More

Explore these resources for further learning:

- W3Schools Programming Tutorials

Python:

- W3Schools Python Tutorial
- Python Documentation
- Learn Python

R:

- R for Data Science (Book)
- RStudio Cheatsheets
- Swirl - Learn R in R

Julia:

- Julia Documentation
- JuliaAcademy Courses
- Learn Julia in Y Minutes

Keypoints

- Programming means instructing computers to perform tasks.
- Variables store data, and each variable has a data type.
- Conditional statements allow programs to make decisions.
- Arrays and lists store collections of data.
- Loops repeat actions efficiently.
- Functions encapsulate reusable code.
- Operators manipulate data.

Jupyter Notebooks

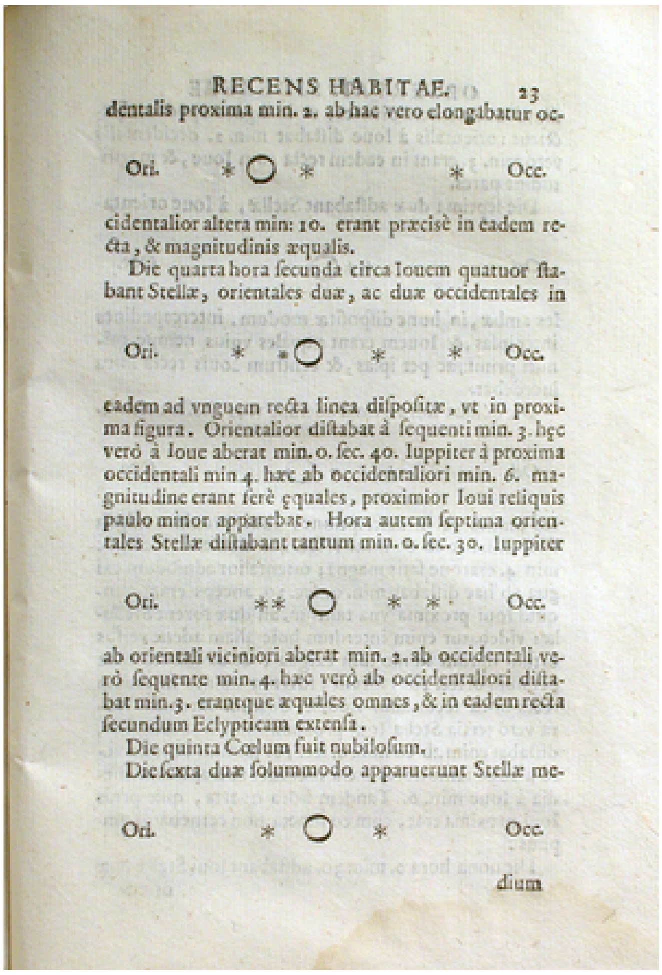
Objectives

- 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-ctą, & 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. hęc verò à Ioue aberat min. 0. ſec. 40. Iuppiter à proxima occidentali min 4. hæc ab occidentaliori min. 6. magnitudine erant ferè ęquales, 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 Jupyter-1: Create a notebook (15 min)

- Open a new notebook (on Windows: open Anaconda Navigator, then launch JupyterLab; on macOS/Linux: you can open JupyterLab from the terminal by typing `jupyter-lab`)
- Rename the notebook
- Create a **markdown cell** with a section title, a short text, an image, and an equation

```
# Title of my notebook

Some text.

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

$E = mc^2$
```

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

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

- In a different cell, call the function:

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

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

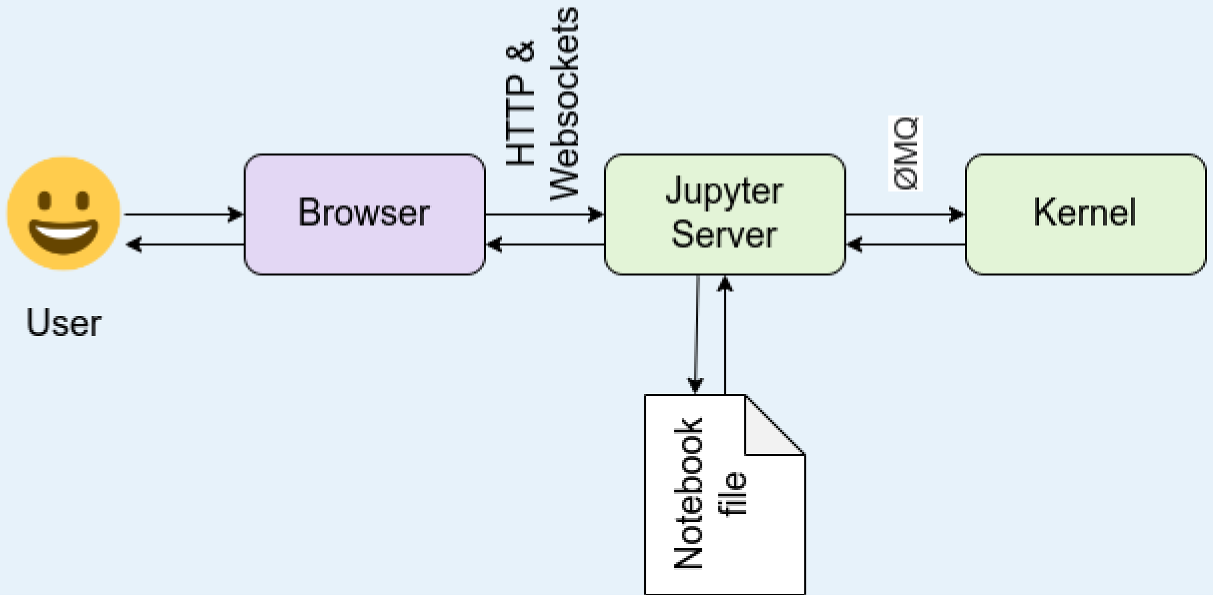
```python
# this example is from https://altair-
viz.github.io/gallery/layered_histogram.html

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.

Note

Understanding the Jupyter architecture

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



Think of it like a kitchen:

- **You (the user)** are the **chef** — deciding what you want to cook (analyze), and writing down the recipe.
- The **browser** is your fancy **recipe notebook interface**, where you write, update, and view your instructions.
- The **Jupyter server** is the **kitchen manager** — it organizes everything, coordinates requests, and keeps track of the pots, pans, and ingredients.
- The **kernel** is a **robotic cook** operating the **stove**. It follows your instructions, grabs the ingredients, prepares the dishes, and sends them back to your notebook — this is the part that actually runs your code.
- The **notebook file (`.ipynb`)** is the **recipe sheet**, storing everything: code, instructions, outputs, and notes — so you (or others) can reproduce the results later.

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

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

Use cases for notebooks

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

Good practices

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

This can be demonstrated with the following example:

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

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

Limitations of notebooks

(Read more at Pitfalls of Jupyter Notebooks.)

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

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

Keypoints

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

Python basics

Objectives

- Knowing what types exist
- Knowing the most common data structures: 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

```python
# 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

```python
# 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:

```python
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?):

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

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

Iterating over a dictionary:

```python
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.

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

- Together we write a function which sums all elements in a list:

```python
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:

```python
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:

```python
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 but also simplify reading and understanding code.

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:

```python
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):

```python
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)

Exercises

✍️ Exercise: checking file names before publishing a dataset

You have received a dataset (a folder with many files) but some files do not follow good naming conventions (no spaces, no strange characters) which could make the dataset less FAIR.

```
files = [
    "Final Report V2!.xlsx",
    "data summary 2023 .csv",
    "RESULTS-supplementary,materials.csv",
    "clean_data.csv",
    "code%python.py",
    "code#for#%plots.py",
    "figure1.eps",
    "figure2.eps",
    "figure2&.eps",
    "figure2_.eps"
]
```

The number of files is so large that manually reviewing each one would be time-consuming and error-prone.

You are given the following Python script to check if a file name contains non-recommended characters:

```
def check_filename(filename):
    bad_chars = [' ', ',', '&', '!']
    for char in filename:
        if char in bad_chars:
            print(f'"{filename}" is bad because it contains one or more non-
recommended characters.')
            return
    print(f'"{filename}" is good (from a FAIR point of view).')

for name in files:
    check_filename(name)
```

Your tasks are:

1. Create a new jupyter notebook and make sure you can run the two code blocks above. Look at the output.
2. Add "#" and "%" to the list of non-recommended characters.
3. Modify the function so it also prints which bad character(s) were found.

4. (Advanced) Modify the function so that it suggests a better file name by replacing non-recommended characters with underscores. If the output of this function would be used to automatically rename files, what risks could we face?

✔ Solution

Task 1 Solution

You should be able to see something like:

```
"Final Report V2!.xlsx" is bad because it contains one or more non-recommended characters.
"data summary 2023 .csv" is bad because it contains one or more non-recommended characters.
"RESULTS-supplementary,materials.csv" is bad because it contains one or more non-recommended characters.
"clean_data.csv" is good (from a FAIR point of view).
"code%python.py" is good (from a FAIR point of view).
"code#for#%plots.py" is good (from a FAIR point of view).
"figure1.eps" is good (from a FAIR point of view).
"figure2.eps" is good (from a FAIR point of view).
"figure2&.eps" is bad because it contains one or more non-recommended characters.
"figure2_.eps" is good (from a FAIR point of view).
```

Task 2 Solution – Add `"#"` and `"%"` to the bad characters list

```python
def check_filename(filename):
    bad_chars = [' ', ',', '&', '!', '#', '%']  # added '#' and '%'
    for char in filename:
        if char in bad_chars:
            print(f'"{filename}" is bad because it contains one or more non-recommended characters.')
            return
    print(f'"{filename}" is a good filename.')
```

Task 3 Solution – Show which bad characters were found

```python
def check_filename(filename):
    bad_chars = [' ', ',', '&', '!', '#', '%']
    found_bad = []

    for char in filename:
        if char in bad_chars and char not in found_bad:
            found_bad.append(char)

    if found_bad:
        print(f'"{filename}" is bad because it contains the following non-recommended characters: {found_bad}')
    else:
        print(f'"{filename}" is a good filename.')
```

Task 4 (Advanced) – Suggest a better filename

Here, we'll suggest a new filename by replacing bad characters with underscores (`_`), using a character-by-character loop.

```python
def check_filename(filename):
    bad_chars = [' ', ',', '&', '!', '#', '%']
    found_bad = []
    suggested = ""

    for char in filename:
        if char in bad_chars:
            if char not in found_bad:
                found_bad.append(char)
            suggested += "_"  # replace bad character
        else:
            suggested += char  # keep good character

    if found_bad:
        print(f'"{filename}" is bad because it contains: {found_bad}')
        print(f'→ Suggested filename: "{suggested}"')
    else:
        print(f'"{filename}" is a good filename.')
```

What issue do you see in the output of this function, if the new name would be used to automatically rename files?

✍️ 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:

```python
# 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:

```python
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)

```python
# 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()`.

```python
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:

```python
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:

  ```python
  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)
```

R basics

Objectives

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

Motivation for R

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

Basic types

```r
# integer
num_measurements <- 13L

# numeric (float)
some_fraction <- 0.25

# character (string)
name <- "Bruce Wayne"

# logical (bool)
value_is_missing <- FALSE
skip_verification <- TRUE

# we can print values
print(name)

# arithmetic operations with integers and numeric values
print(5 * num_measurements)
print(1.0 - some_fraction)
```

- R is **dynamically typed**: We do not have to define that an integer is an `integer`, we can use it directly, and R will infer it.
- Comments in R use the `#` symbol.

Data structures for collections: vectors, lists, data frames, and sets

```r
# vectors (similar to Python lists, important for ordered elements)
scores <- c(13, 5, 2, 3, 4, 3)

# first element
print(scores[1])

# add items to vectors
scores <- c(scores, 4)

# sort vectors
scores <- sort(scores)
print(scores)

# lists are useful to store collections with named elements
experiment <- list(location = "Svalbard", date = "2021-03-23", num_measurements = 23)

print(experiment$date)

# add items to lists
experiment$instrument <- "a particular brand"
print(experiment)

if ("instrument" %in% names(experiment)) {
  print("yes, the list 'experiment' contains the element 'instrument'")
} else {
  print("no, it doesn't")
}
```

- **Vectors** are good when order matters and elements are homogeneous.
- **Lists** allow heterogeneous elements and are named collections.

- **Data frames** are like tables.
- **Sets** (`unique()` function or `sets` package) for collections without repetition.

You can nest:

- lists inside vectors
- vectors inside lists
- data frames inside lists
- lists inside data frames
- ...

Iterating over collections

Iterating over a vector:

```
scores <- c(13, 5, 2, 3, 4, 3)

for (score in scores) {
  print(score)
}

# example with string formatting
for (score in scores) {
  print(sprintf("the score is %s", score))
}
```

Iterating over a list:

```
experiment <- list(location = "Svalbard", date = "2021-03-23", num_measurements = 23)

# iterating over values
for (value in experiment) {
  print(value)
}

# iterating over names and values
for (key in names(experiment)) {
  print(paste(key, experiment[[key]]))
}
```

Functions

- Functions are like **reusable recipes**. They receive input arguments, perform operations, and return a result.

  ```
  add <- function(a, b) {
    result <- a + b
    return(result)
  }
  ```

- Function summing elements in a vector:

```r
add_all_elements <- function(sequence) {
  s <- 0.0
  for (element in sequence) {
    s <- s + element
  }
  return(s)
}

measurements <- c(1,2,3,4,5,6,7,8,9,10)
print(add_all_elements(measurements))
```

- Function computing the mean:

```r
arithmetic_mean <- function(sequence) {
  s <- add_all_elements(sequence)
  n <- length(sequence)
  return(s / n)
}

measurements <- c(1,2,3,4,5,6,7,8,9,10)
mean <- arithmetic_mean(measurements)
print(mean)
```

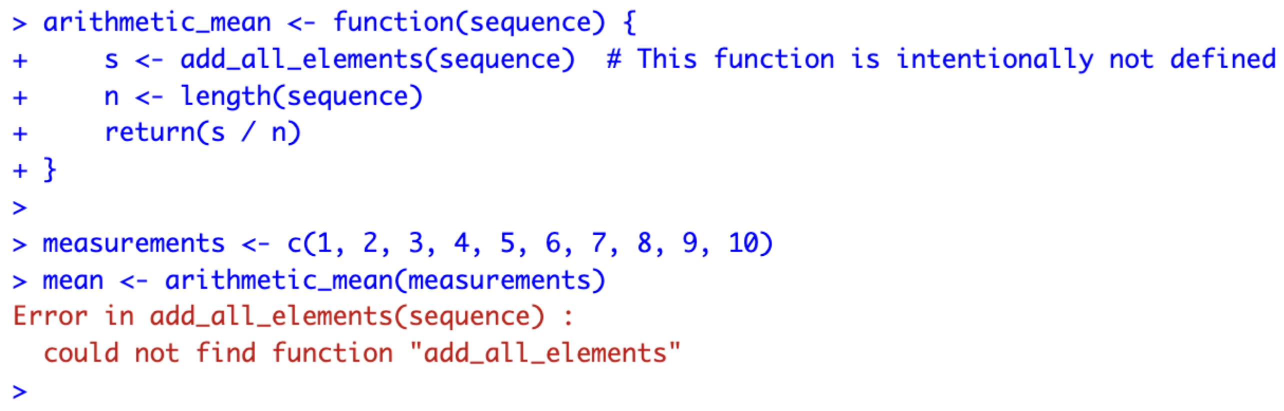
- Functions calling other functions and returning multiple values:

```r
uppercase_and_lowercase <- function(text) {
  u <- toupper(text)
  l <- tolower(text)
  return(list(upper = u, lower = l))
}

some_text <- "SequenceOfCharacters"
text_cases <- uppercase_and_lowercase(some_text)

print(text_cases$upper)
print(text_cases$lower)
```

Why functions? Reduce repetition and simplify understanding of the code.

Reading error messages

Here we introduce a mistake and try to understand the traceback:



```
> arithmetic_mean <- function(sequence) {
+     s <- add_all_elements(sequence)  # This function is intentionally not defined
+     n <- length(sequence)
+     return(s / n)
+ }
>
> measurements <- c(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
> mean <- arithmetic_mean(measurements)
Error in add_all_elements(sequence) :
  could not find function "add_all_elements"
>
```

Example error traceback. Can you explain the error?

Libraries

Libraries are collections of functions. We load libraries to reuse functions defined in them.

Try this:

```r
library(stats)

measurements <- c(1,2,3,4,5,6,7,8,9,10)
result <- sd(measurements)
print(result)
```

Often, libraries are loaded directly:

```r
library(dplyr)
```

You can create your own libraries (packages) with functions for reuse.

Great resources to learn more

- R for Data Science (great for beginners)
- The R Manuals (great for beginners)
- Advanced R (intermediate level)

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:

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

- Take this as starting point:

```r
arithmetic_mean <- function(sequence) {
  mean(sequence)
}

standard_deviation <- function(sequence) {
  # your code here
}
```

✔ Solution

```r
standard_deviation <- function(sequence) {
  mean <- arithmetic_mean(sequence)
  sqrt(mean((sequence - mean)^2))
}
```

✍ Exercise: working with a named list

- Add more names and grades:

```r
grades <- list(Alice = 80, Bob = 95)
```

- Print, modify, and explore the list.

✔ Solution

```r
grades$Craig <- 56
grades$Dave <- 28
grades$Eve <- 75
print(grades)

# Accessing a specific grade
print(grades$Eve)

# Names, values, and items
print(names(grades))
print(unlist(grades))
```

Working with data in Python

❗ Objectives

- Understanding how to structure data in your projects

- Reading data with Pandas from disk or a web resource

Note

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**.

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:

```python
import pandas as pd

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

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

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

# let us print the combined result
data_monthly
```

Data preprocessing

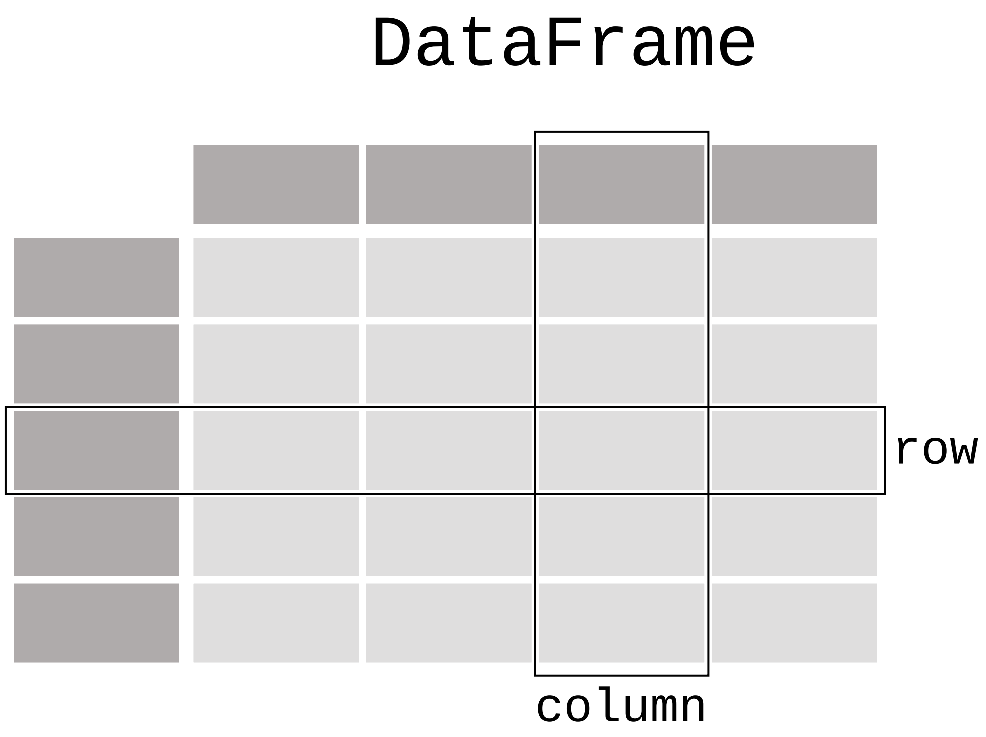
There is a problem which we may not see yet: Dates are not in a standard date format (YYYY-MM-DD). We can fix this:

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

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

What is in a dataframe?

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



A pandas dataframe object is composed of rows and columns.

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

```python
# 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]
```

What makes Python and pandas better than Excel?

- **Automation**: Python scripts can be reused and automated, reducing repetitive tasks.
- **Scalability**: Excel struggles with large datasets, while pandas handles millions of rows efficiently.
- **Version control**: Code and data processing steps can be tracked using Git.
- **Reproducibility**: Your workflow can be repeated or shared exactly by others.
- **Visualization and analysis**: Seamless integration with powerful libraries like matplotlib, seaborn, and statsmodels.
- **Error prevention**: Fewer manual steps mean fewer opportunities for copy-paste mistakes.
- **Integration**: Easily connects with databases, APIs, and other formats like JSON, HDF5, etc.

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)

✍ Exercise - Work with tabular data using Python

You might be an Excel expert, but how about doing the same things you do with Excel with python? What is in your opinion the biggest advantage compared to doing it with Excel?

1. Try running some of the commands above in a jupyter notebook and look at their output
2. Calculate the median of the column “max temperature”. What is its value? To know how to calculate the median, check the documentation of Pandas DataFrame class.
3. Sort the data by the column “max temperature”. Which city was the hottest? Use the Pandas documentation to know how to sort.
4. (Advanced) Compute the differences of max temperatures between Olso and Tromsø for each month (hint: you will need to make a pivot table…)

✔ Solution

2. **Calculate the median of the column “max temperature”.**

```
data_monthly["max temperature"].median()
```

3. **Sort the data by the column “max temperature”. Which city was the hottest?**

```
sorted_data = data_monthly.sort_values("max temperature", ascending=False)
sorted_data.head(1)
```

4. **(Advanced) Compute the differences of max temperatures between Oslo and Tromsø for each month**

```
pivot_df = data_monthly.pivot(index='date', columns='name', values='max
temperature')
pivot_df['Temperature_Difference'] = pivot_df['Oslo - Blindern'] - pivot_df['Tromso
- Langnes']
print(pivot_df)
```

Other Python libraries for tabular data

Besides pandas, here are a few other libraries useful for working with tabular data:

- **Polars**: Fast DataFrame library written in Rust, great for large datasets.
- **Dask**: Scales pandas workflows to larger-than-memory datasets and parallel computation.
- **Vaex**: Optimized for lazy evaluation and memory-efficient handling of huge tabular datasets.
- **PySpark**: Python API for Apache Spark, useful for distributed data processing.
- **datatable**: High-performance library for large data processing with R-style syntax.

Libraries for other types of data

Here are some popular Python libraries for working with non-tabular data:

- **JSON and structured text:** `json`, `orjson`, `pydantic`
- **Plain text and NLP:** `nltk`, `spacy`, `transformers`, `re` (regex)
- **Images:** `Pillow`, `OpenCV`, `imageio`, `scikit-image`
- **Videos:** `OpenCV`, `moviepy`, `ffmpeg-python`
- **Audio and sound:** `librosa`, `pydub`, `torchaudio`, `wave`
- **Scientific data formats:** `h5py`, `netCDF4`, `xarray`
- **Geospatial data:** `geopandas`, `shapely`, `rasterio`, `fiona`

These libraries can be combined with pandas or used standalone to handle more complex, real-world data types.

Keypoints

- Loading data with python is easy and through Jupyter notebooks one can interact and understand the data at hand
- Pandas are the most popular option when dealing with tabular data
- There are many more types of data and libraries available, please explore which data you would like to load.

Plotting with Vega-Altair

Objectives

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

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

Reading data into a dataframe

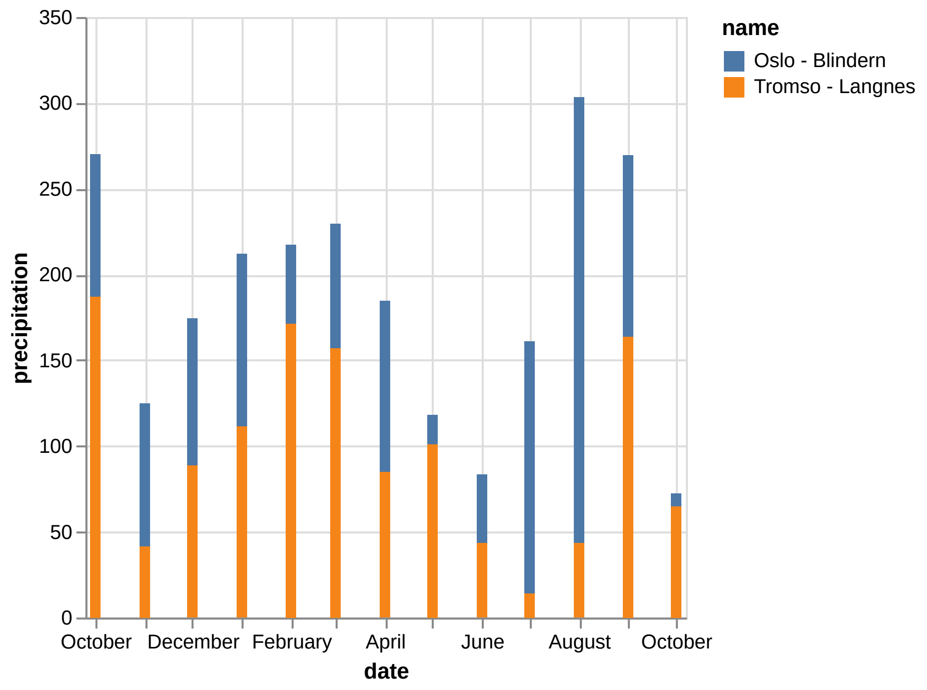
From the previous section, let’s load the data in our jupyter notebook and fix the dates.

```
import pandas as pd

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

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

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

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

# let us print the combined result
data_monthly
```

Plotting the data

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

```
import altair as alt

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



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

💬 Let us pause and explain the code

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

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

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

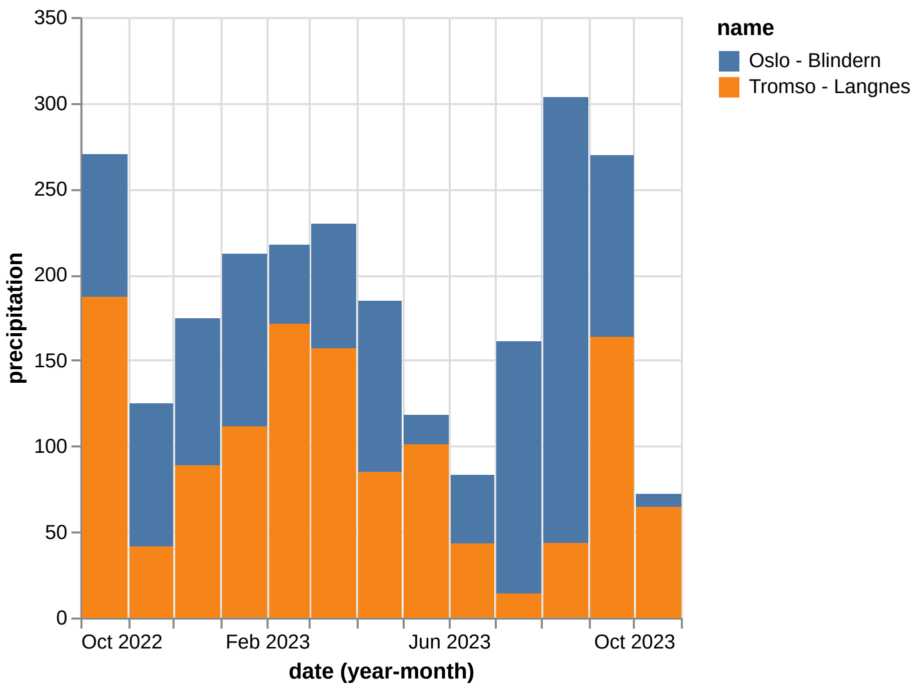
- x-coordinate with "date"
- y-coordinate with "precipitation"
- color with "name" (name of weather station; city)

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

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

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

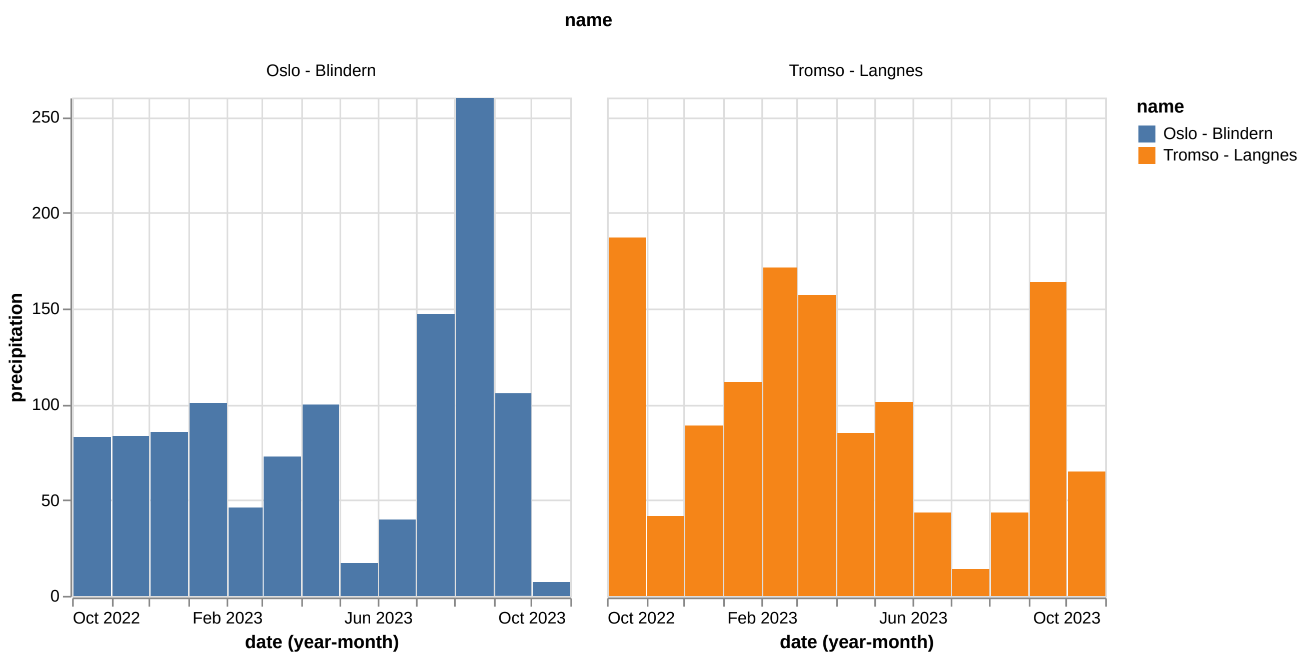
- Q (quantitative)
- O (ordinal)
- N (nominal)
- T (temporal)
- G (geojson)



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

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

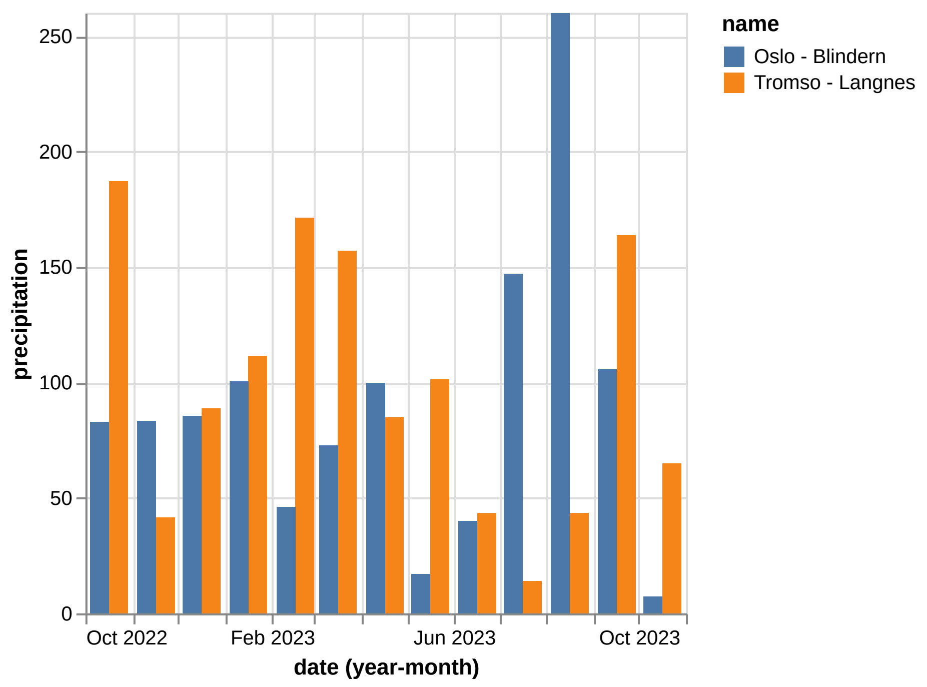
```
alt.Chart(data_monthly).mark_bar().encode(
    x="yearmonth(date):T",
    y="precipitation",
    color="name",
    column="name",
)
```



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

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

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

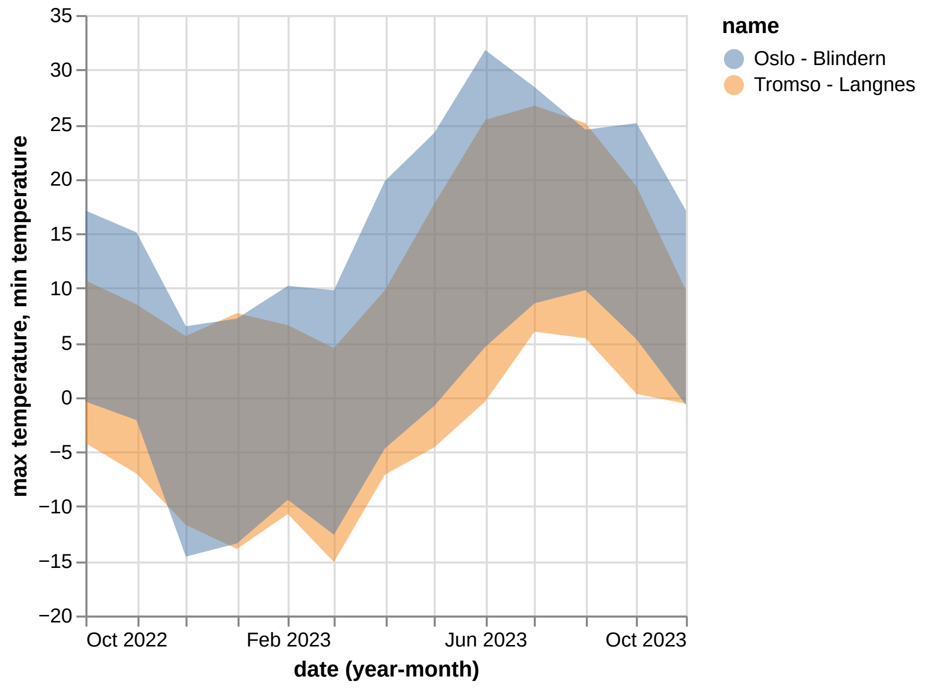


Monthly precipitation for the cities Oslo and Tromsø over the course of a year plotted as stacked bar chart.

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

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

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



Monthly temperature ranges for two cities in Norway.

💬 What other marks and other visual channels exist?

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

Themes

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

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

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

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

You can even define your own themes!

✍️ Discover the Vega-Altair gallery of examples

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

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

❗ Keypoints

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

Software install instructions

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

Choosing an installation method

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

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

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

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

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

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

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

❗ Python, conda, anaconda, miniforge, etc?

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

- **Python** is a programming language very commonly used in science, it's the topic of this course.
- **Conda** is a package manager: it allows distributing and installing packages, and is designed for complex scientific code.
- **Mamba** is a re-implementation of Conda to be much faster with resolving dependencies and installing things.

- An **Environment** is a self-contained collections of packages which can be installed separately from others. They are used so each project can install what it needs without affecting others.
- **Anaconda** is a commercial distribution of Python+Conda+many packages that all work together. It used to be freely usable for research, but since ~2023-2024 it's more limited. Thus, we don't recommend it (even though it has a nice graphical user interface).
- **conda-forge** is another channel of distributing packages that is maintained by the community, and thus can be used by anyone. (Anaconda's parent company also hosts conda-forge packages)
- **miniforge** is a distribution of conda pre-configured for conda-forge. It operates via the command line.
- **miniconda** is a distribution of conda pre-configured to use the Anaconda channels.

Installing Python via Miniforge

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

Installing and activating the software environment

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

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

We will call the environment `data-viz`.

Windows   Linux / MacOS

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

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

Starting JupyterLab

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

Windows | Linux / MacOS

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

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

Removing the software environment

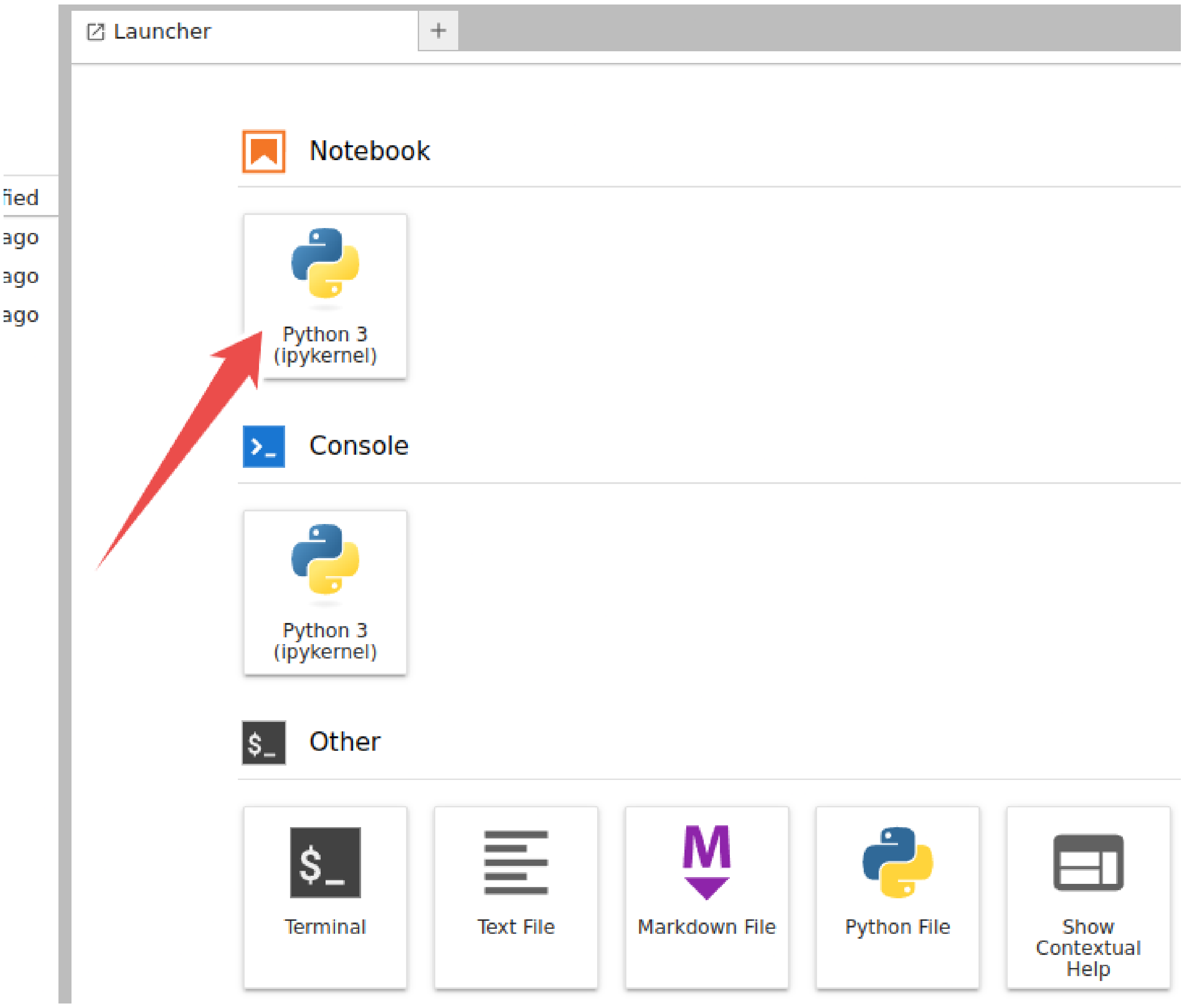
Windows | Linux / MacOS

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

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

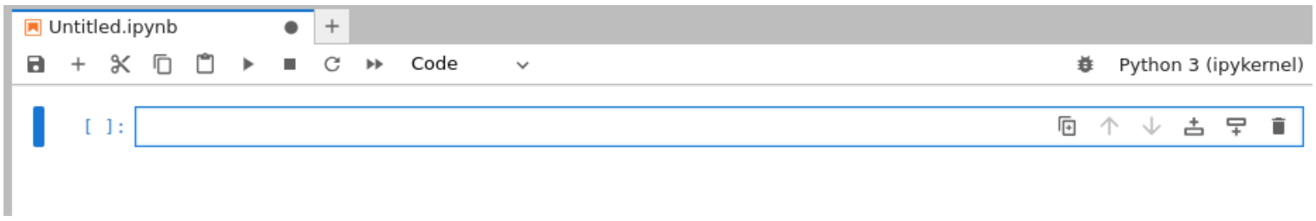
How to verify your installation

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



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

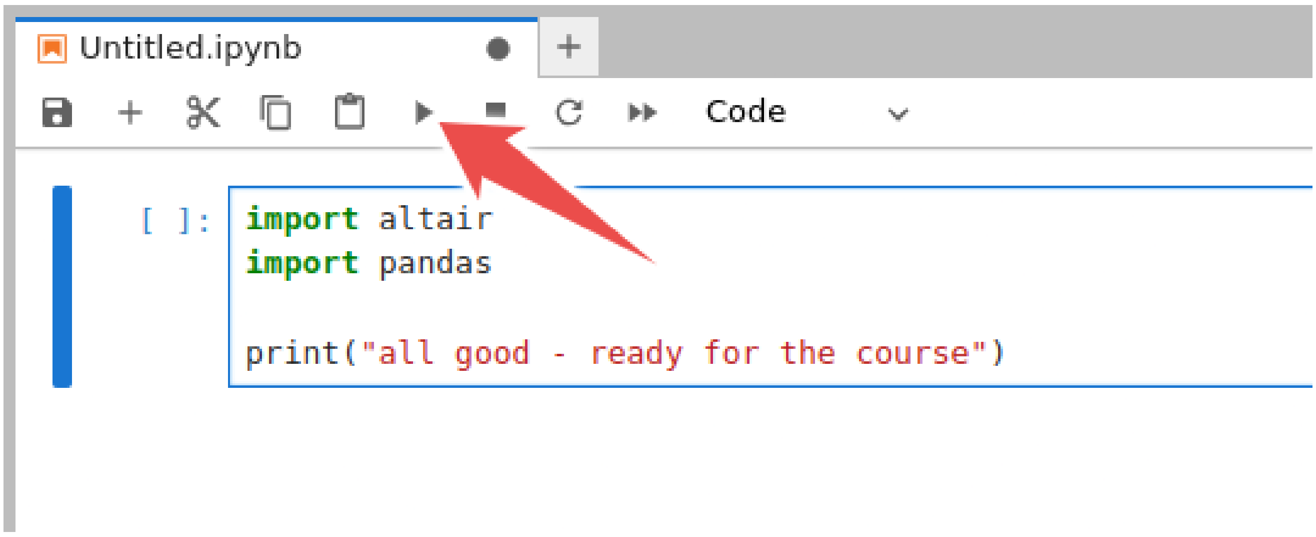
Once you clicked the Python 3 tile it should look like this:



Python 3 notebook started.

Into that blue "cell" please type the following:

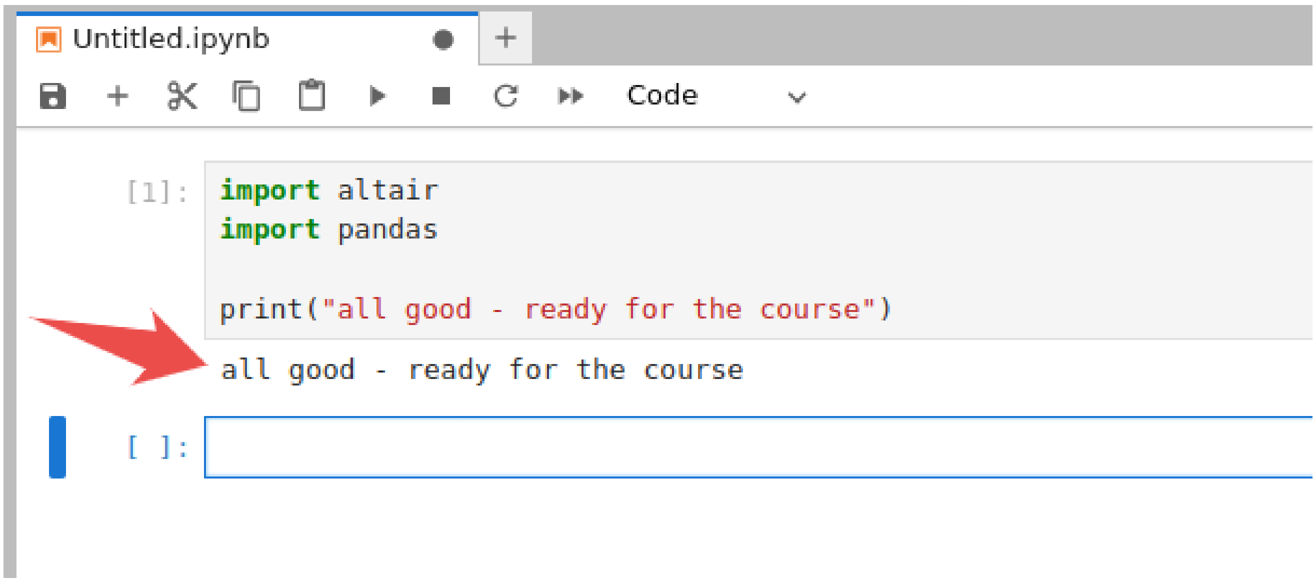
```
import altair
import pandas

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



```
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:



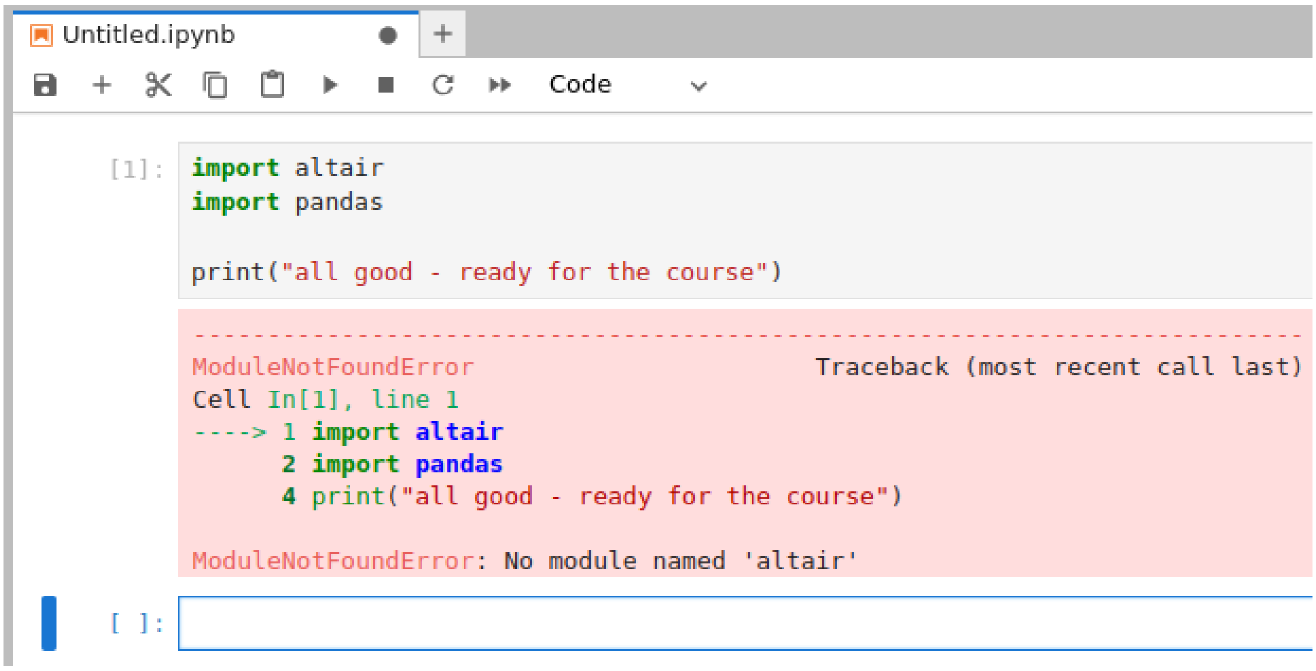
```
import altair
import pandas

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

```
all good - ready for the course
```

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:



```
[1]: import altair
     import pandas

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

```
---------------------------------------------------------------------------
ModuleNotFoundError                       Traceback (most recent call last)
Cell In[1], line 1
----> 1 import altair
      2 import pandas
      4 print("all good - ready for the course")

ModuleNotFoundError: No module named 'altair'
```

Error: required packages could not be found.

Going through the course using Google Colab

It is possible to go through the workshop examples using Google Colab.

However you may need the following cell on top of your notebook:

```
import altair as alt

# this is here for google colab to update altair
if not alt.__version__.startswith("5"):
    %pip install altair==5.5.0
```

And then you may need to click on “Runtime” -> “Restart session and run all”.

Tidy data and dealing with messy data

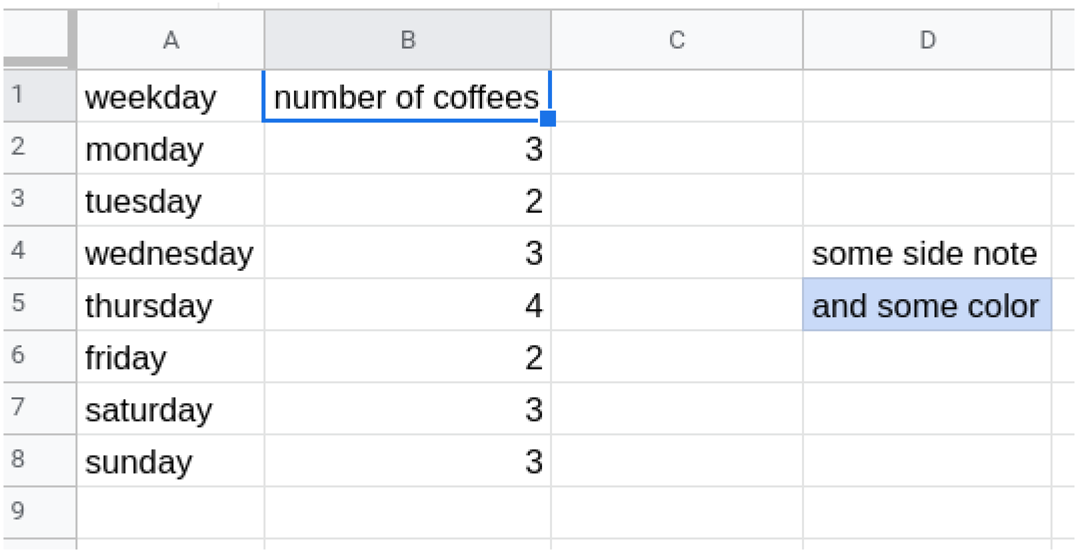
Objectives

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

In the previous episode we read data from 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 will 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`.

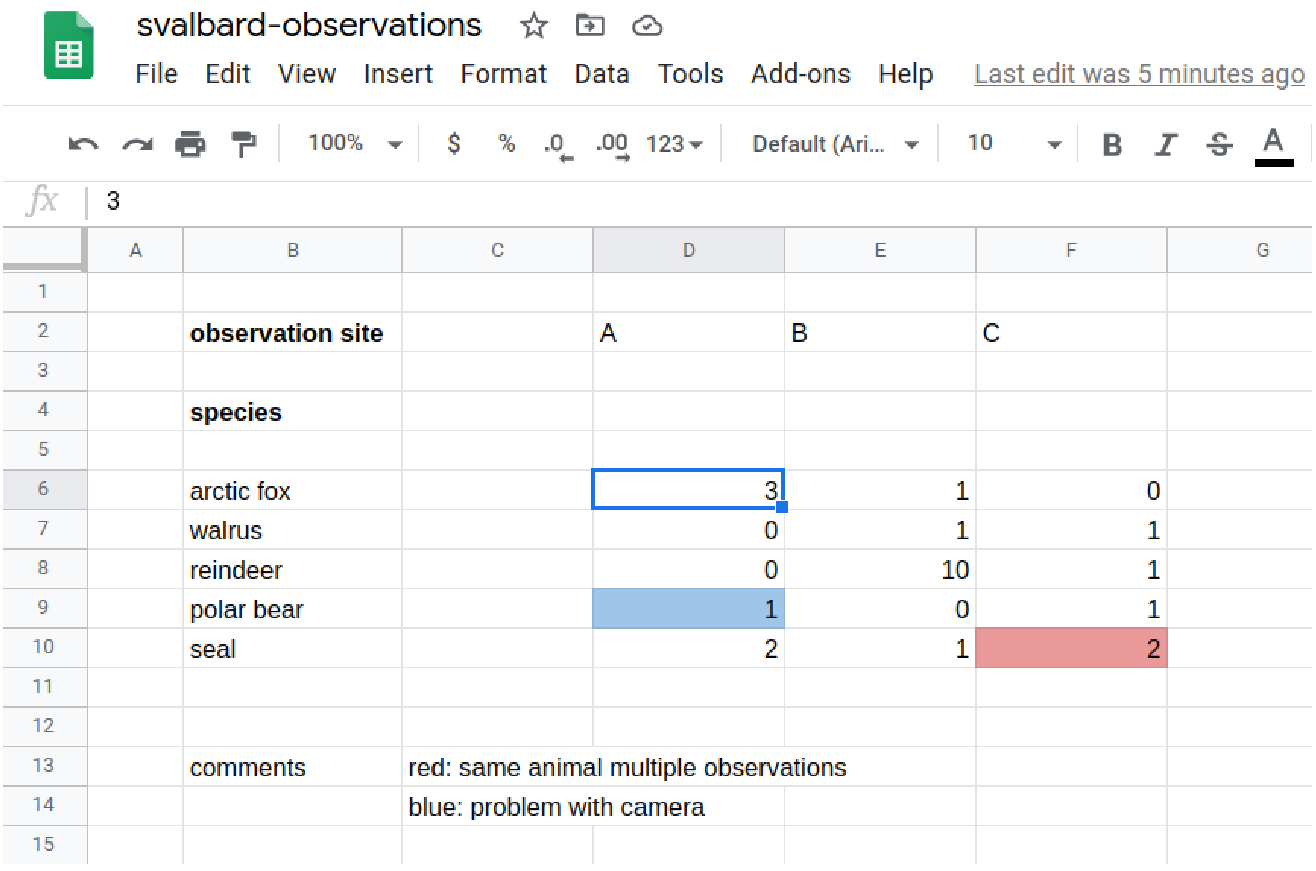
Now we will together try to read and inspect both sheets in the Jupyter Notebook:

```
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%
Default (Ari...
10

	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

Customizing plots

Objectives

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

Loading and plotting a dataset

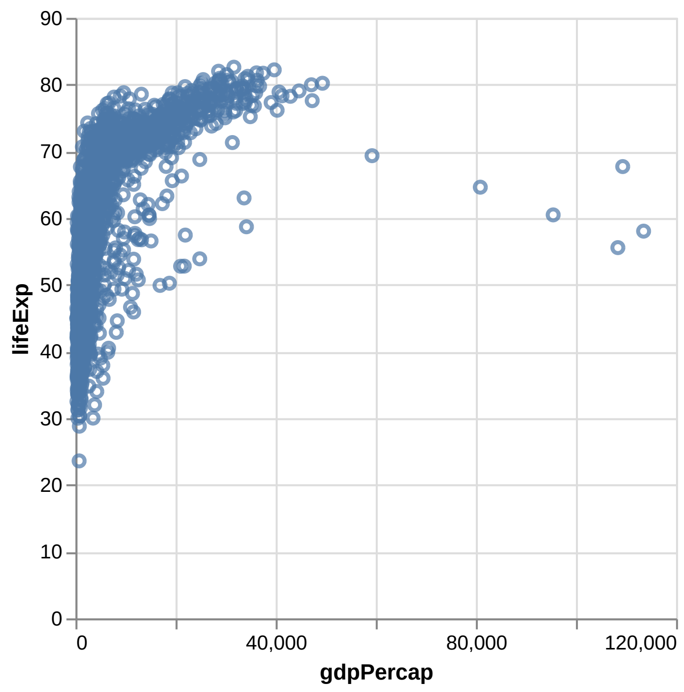
In this lesson will work with one of the Gapminder datasets.

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

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

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

# print overview of the dataset
data
```

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

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



First raw plot with all countries and all years.

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

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

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

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

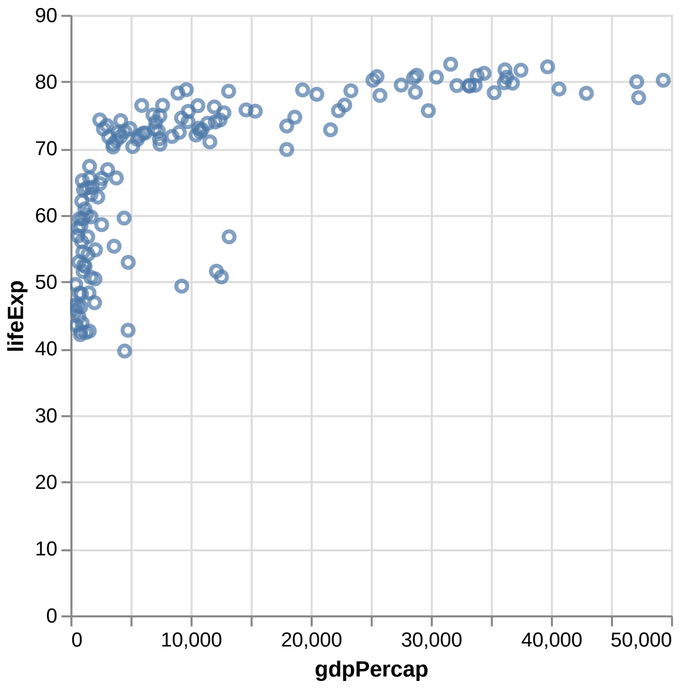
💬 **Let us pause and explain the code**

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

Filtering data

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

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



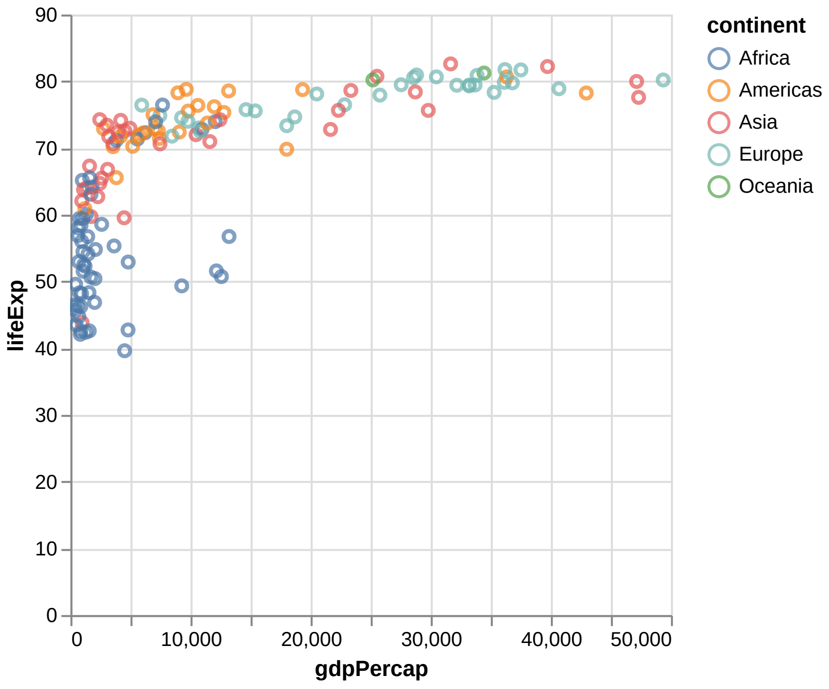
Now we only keep the year 2007.

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

Using color as additional channel

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

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

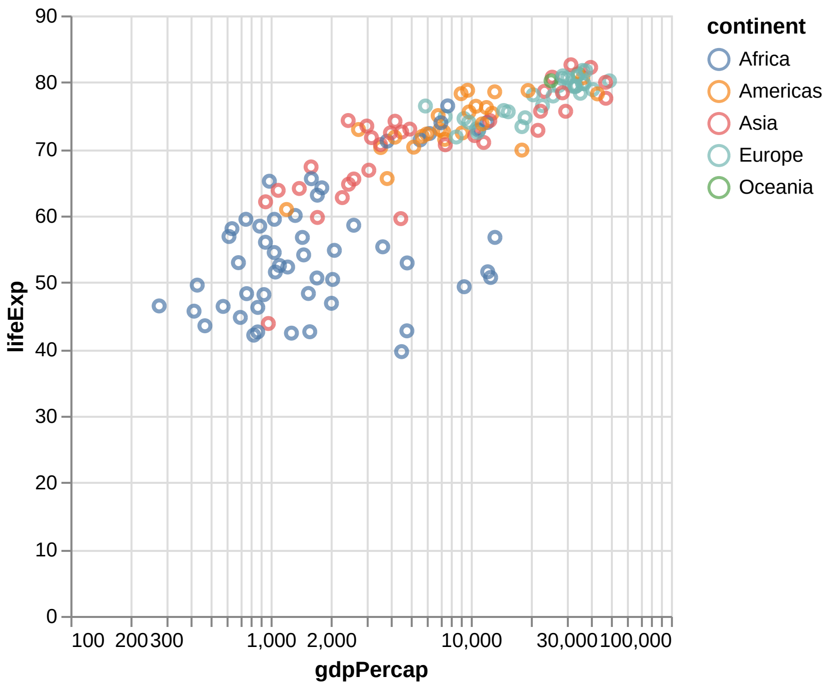


Using different colors for different continents.

Changing to log scale

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

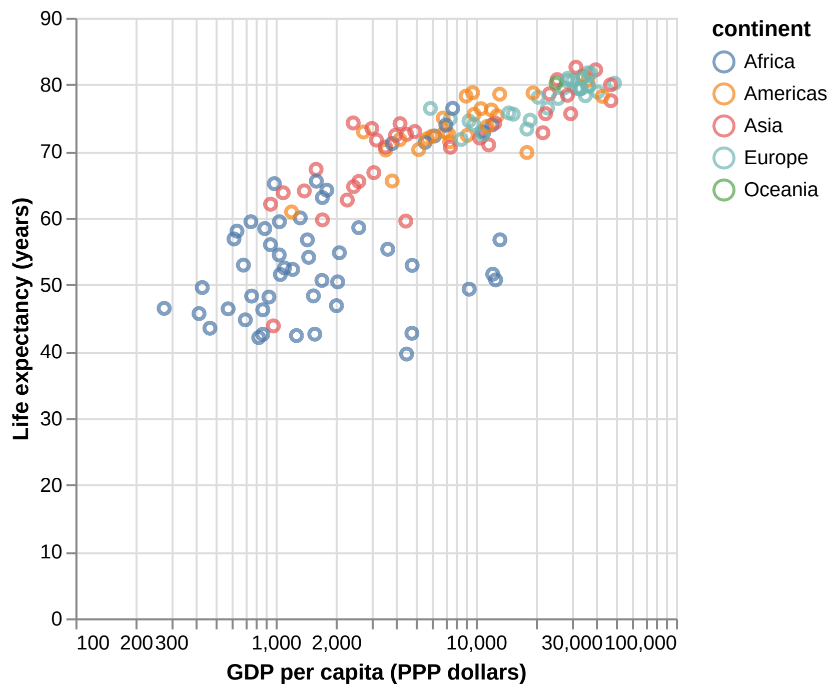
```
alt.Chart(data).mark_point().encode(
    x=alt.X("gdpPercap").scale(type="log"),
    y=alt.Y("lifeExp"),
    color="continent",
).transform_filter(alt.datum.year == 2007).interactive()
```



Changing the x axis to log scale.

Improving axis titles

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



Improving the axis titles.

Faceted charts

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

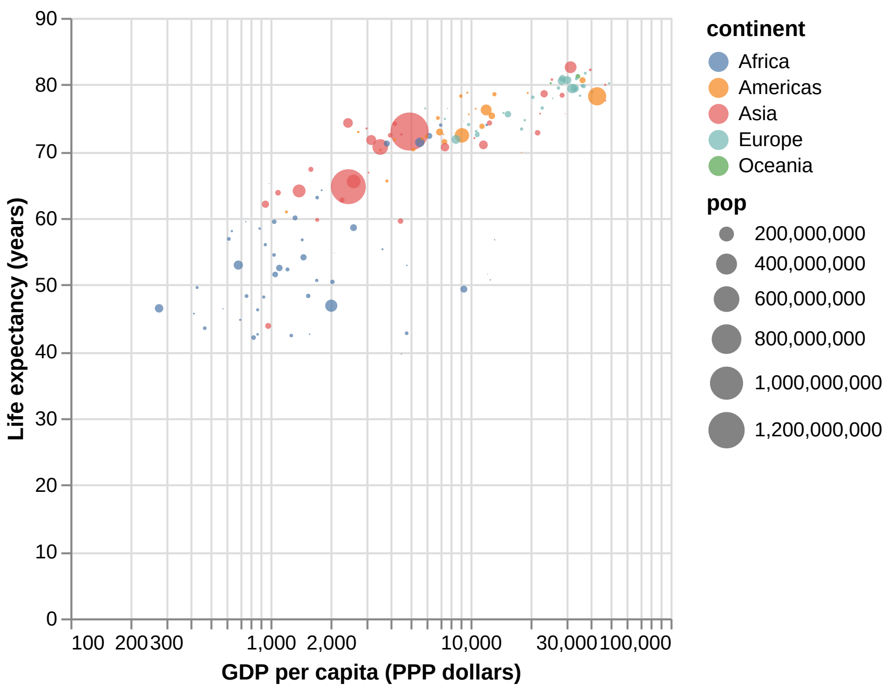
```
alt.Chart(data).mark_point().encode(
    x=alt.X("gdpPercap").scale(type="log").title("GDP per capita (PPP dollars)"),
    y=alt.Y("lifeExp").title("Life expectancy (years)"),
    color="continent",
    row="continent",
).transform_filter(alt.datum.year == 2007).interactive()
```

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

Changing from points to circles

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

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



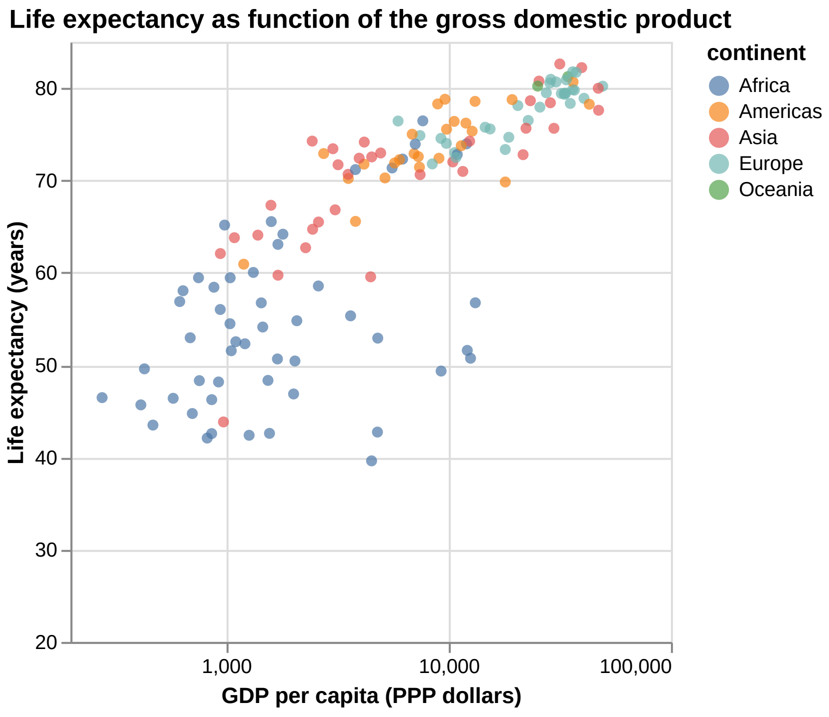
Circle sizes are proportional to population sizes.

Title and axis values

In the next step we modify a number of things:

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

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

chart.interactive()
```

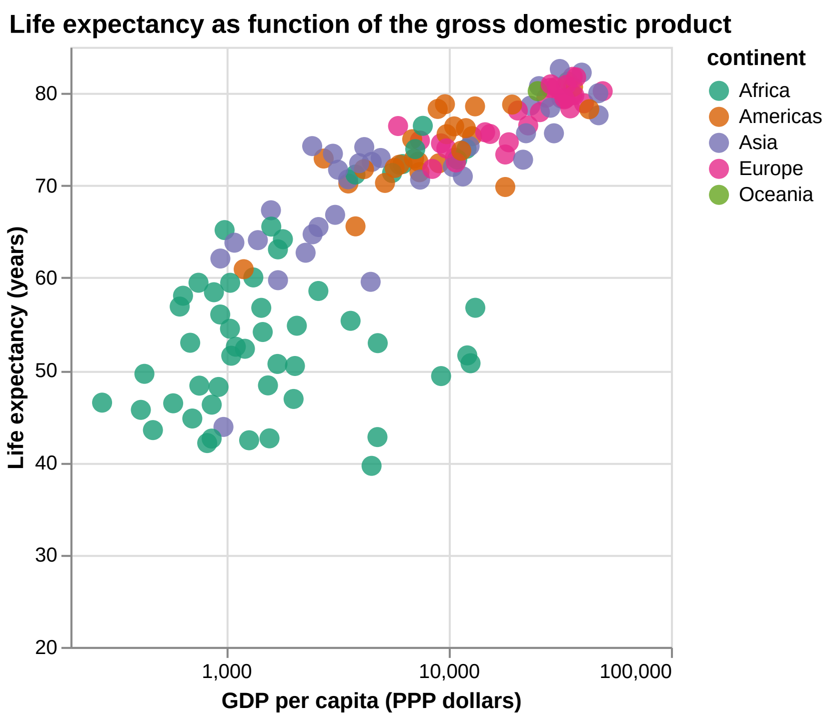


The plot is starting to look better!

Colors

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

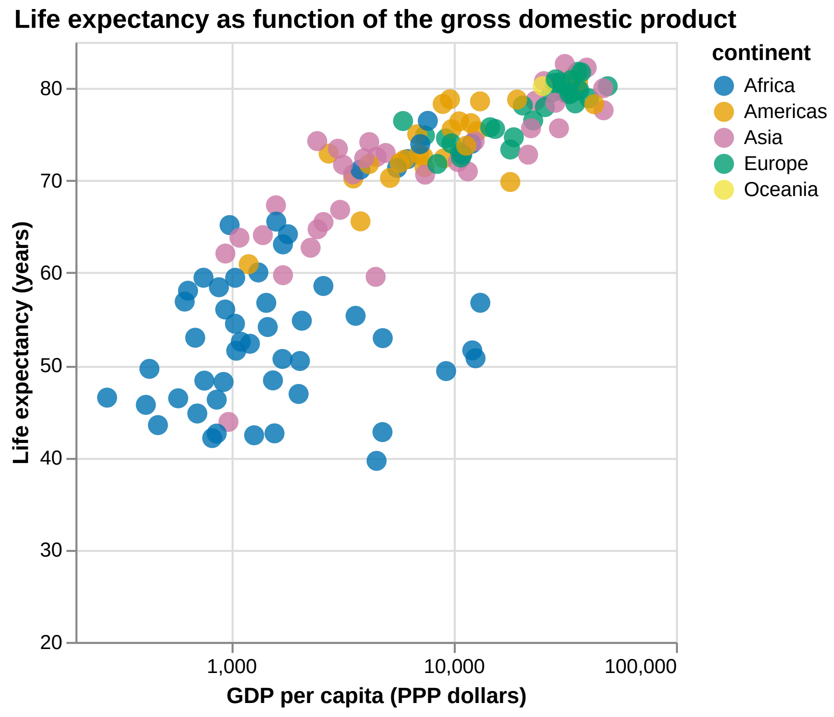
```
chart = (
    alt.Chart(
        data,
        title=alt.Title("Life expectancy as function of the gross domestic product"),
    )
    .mark_circle(opacity=0.8, size=100.0)
    .encode(
        x=alt.X("gdpPercap", axis=alt.Axis(values=[100, 1000, 10000, 100000]))
        .scale(type="log", domain=(200,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

chart.interactive()
```



The plot after adjusting circles and colors.

We can also define own colors:

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

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

chart.interactive()
```



Adjusting colors to those recommended by Okabe and Ito.

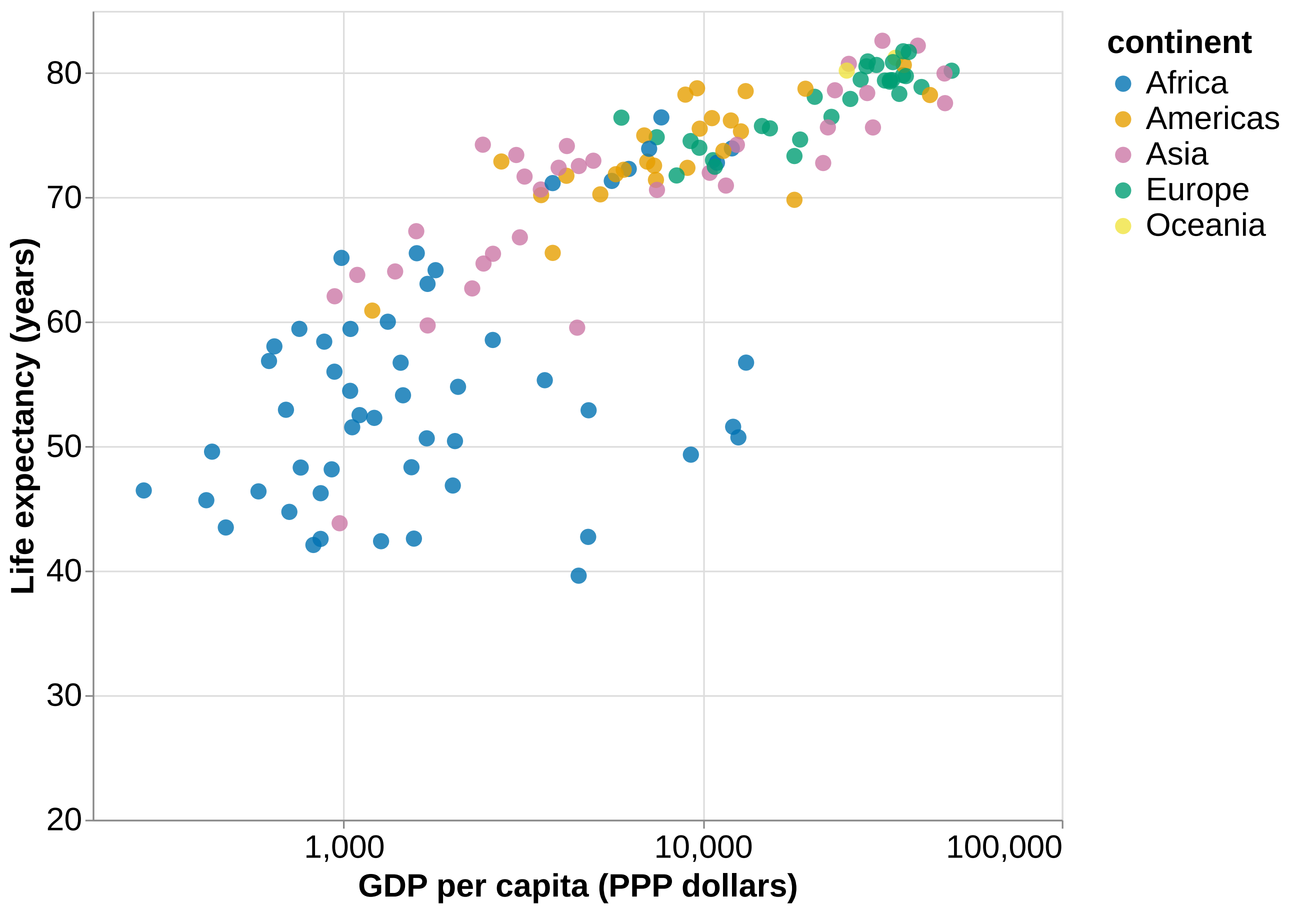
💬 Why these colors?

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

More tweaking towards a publication-ready figure

Let us add a subtitle and adjust sizing and positioning:

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

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

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

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

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

chart.interactive()
```



Life expectancy as function of the gross domestic product
Gross domestic product (GDP) per capita measures the value of everything produced in a country during a year, divided by the number of people.
The unit is in purchasing power parities (PPP dollars), fixed to 2017 prices.
Data is adjusted for inflation and differences in the cost of living between countries.

Interactive charts

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

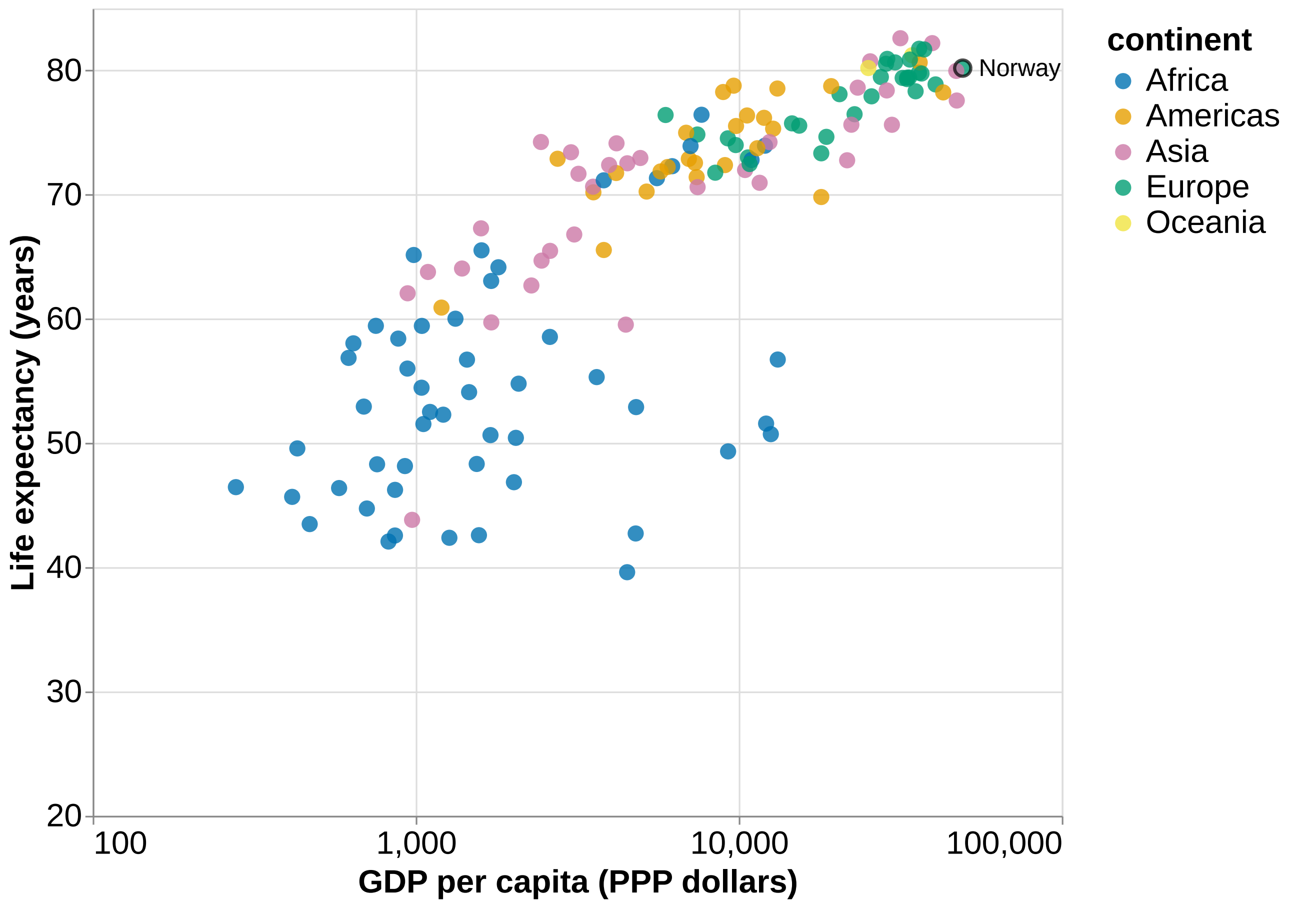
```python
year_slider = alt.binding_range(min=1952, max=2007, step=5, name="Year")
slider_selection = alt.selection_point(bind=year_slider, fields=["year"], value=2007)

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

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

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

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

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

chart.interactive()
```

Adding more annotation

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

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

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

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

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

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

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

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

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

chart.interactive()
```



How life expectancy and gross domestic product evolved over time
Gross domestic product (GDP) per capita measures the value of everything produced in a country during a year, divided by the number of people.
The unit is in purchasing power parities (PPP dollars), fixed to 2017 prices.
Data is adjusted for inflation and differences in the cost of living between countries.

Saving the chart as web page

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

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

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)`:

```python
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`:

```python
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):

```python
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.

Sharing plots and notebooks

Objectives

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

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

Document dependencies

If you import libraries into your notebook, note down their versions.

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

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

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

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

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

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

This is not only useful for people who will try to rerun this in future, it is also understood by some tools (e.g. Binder) which we will see later.

Different ways to share a Vega-Altair plot

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

Vega-Altair and notebooks containing sensitive data

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

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

> "Standard Altair rendering requires the entire dataset to be accessible to the viewer's browser: this is a fundamental design decision in Vega/Vega-Lite, in which a chart is equivalent to a dataset plus a specification of how to render it. In general, you should assume that the entire contents of any dataframe you pass to the alt.Chart() object will be saved in the notebook and be inspectable by the viewer."
>
> "One way to get around this would be to render the chart server-side, export a PNG, and display this png instead of the live chart. Incidentally, in the Jupyter notebook you can do this by running:"
>
> `alt.renderers.enable('png')`

> "This sets up Altair such that charts will be rendered to PNG within the kernel, and only that PNG rendering will be embedded in the notebook. Note this requires some extra dependencies, described here."
>
> "But even here, I wouldn't call your data "private" (for example, if you save a scatter plot to PNG, a user can straightforwardly read the data values off the chart!) So this makes me think you're actually doing some sort of aggregation of your data before plotting (e.g. showing a histogram). If this is the case, I would suggest doing those aggregations outside of Altair using e.g. pandas, and then passing the aggregated dataset to the chart. Then you get the normal interactive display of the Altair chart, and your data is just as private as it would have been in the equivalent static rendering – the user can only see the aggregated values you supplied to the chart."

Different ways to share a notebook

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

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

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

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

Sharing dynamic notebooks using Binder

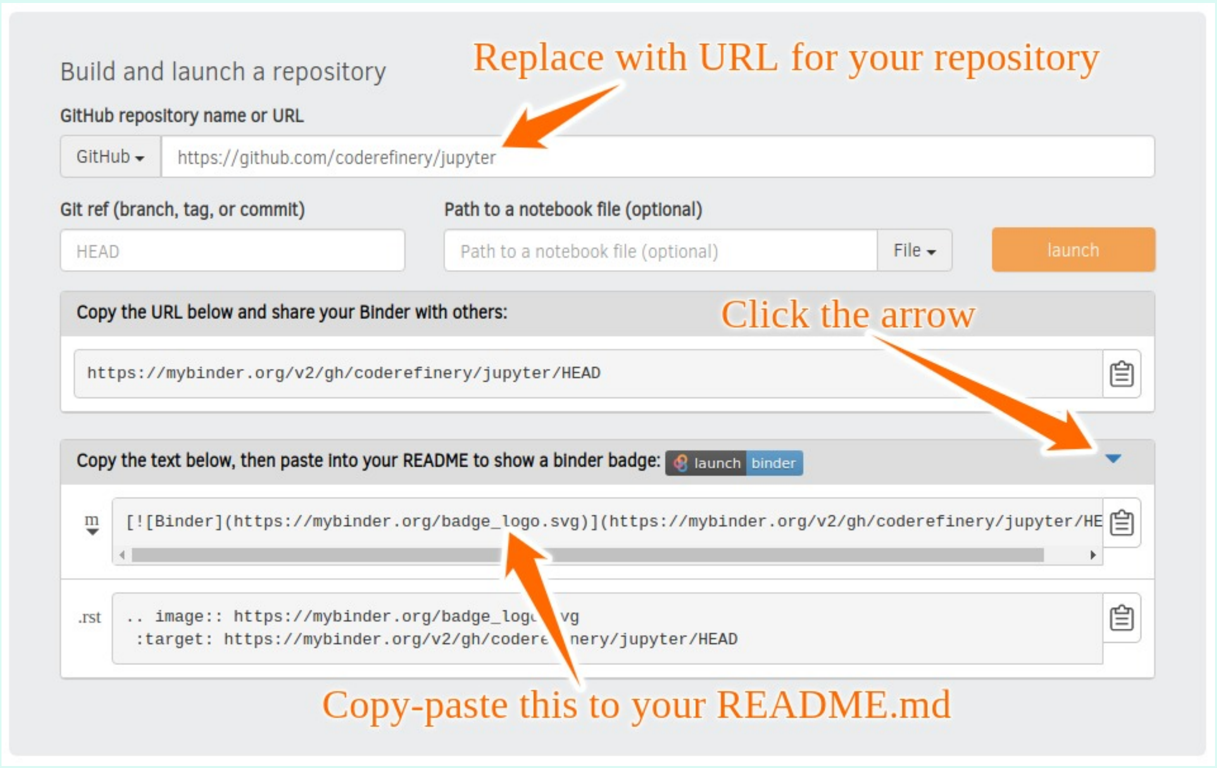
✍ Exercise/demo: Making your notebooks reproducible by anyone (15 min)

Instructor demonstrates this:

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

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

- Visit https://mybinder.org:


Replace with URL for your repository
Click the arrow
Copy-paste this to your README.md
Build and launch a repository
GitHub repository name or URL
GitHub
Git ref (branch, tag, or commit)
Path to a notebook file (optional)
File
launch
launch binder
md
.rst

```
https://github.com/coderefinery/jupyter
HEAD
Path to a notebook file (optional)
```

Copy the URL below and share your Binder with others:

```
https://mybinder.org/v2/gh/coderefinery/jupyter/HEAD
```

Copy the text below, then paste into your README to show a binder badge:

```
[![Binder](https://mybinder.org/badge_logo.svg)](https://mybinder.org/v2/gh/coderefinery/jupyter/HE
```

```
.. image:: https://mybinder.org/badge_logo.svg
 :target: https://mybinder.org/v2/gh/coderefinery/jupyter/HEAD
```

- Check that your notebook repository now has a “launch binder” badge in your `README.md` file on GitHub.
- Try clicking the button and see how your repository is launched on Binder (can take a minute or two). Your notebooks can now be explored and executed in the cloud.
- Enjoy being fully reproducible!

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

How to get a digital object identifier (DOI)

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

Reading data in custom format

From earlier episodes we know that storing data in standard formats can be convenient and we know about the tidy data format but **sometimes we don't have control over this**: we may get a text file from another program or an instrument and now we are expected to deal with it.

As an example, we got two text files from two different instruments. We wish to extract all frequencies and all intensities into two lists.

This is one, `example1.txt`:

```
result from instrument: R2-D2
------------------------------------------

  measurement   frequency   intensity

*     1           0.01       0.01
*     2           0.02       0.02
*     3           0.03       0.01
*     4           0.04       0.10
*     5           0.05       0.20
*     6           0.06       0.12
*     7           0.07       0.07
*     8           0.08       0.02
*     9           0.09       0.01
*    10           0.10       0.01

==========================================

timestamp: Sat Mar 27 03:30:34 PM CET 2021
```

And here is the other one, `example2.txt`:

```
result from instrument: C-3PO
=============================

  numbers we want: 10
      0.01   0.01
      0.02   0.02
      0.03   0.01
      0.04   0.10
      0.05   0.20
      0.06   0.12
      0.07   0.07
      0.08   0.02
      0.09   0.01
      0.10   0.01

  unrelated numbers:
      1.23   4.56
      7.89   0.12
```

At the end we want the code to produce these two lists:

```
frequencies = [0.01, 0.02, 0.03, 0.04, 0.05, 0.06, 0.07, 0.08, 0.09, 0.1]
intensities = [0.01, 0.02, 0.01, 0.1, 0.2, 0.12, 0.07, 0.02, 0.01, 0.01]
```

💬 **Discussion**

Before we discuss possible solutions, it can be a good exercise to **describe in words** what the code should do to extract the numbers.

Below you find possible solutions for these two examples. These are designed to be somehow understandable (which does not mean that the code is trivial) and are not as short and as elegant as they could be. One way to make them more elegant would be to use regular expressions.

One solution for `example1.txt`:

```python
def read_data(file_name):
    # we start with empty lists
    frequencies = []
    intensities = []

    # open the file with read permissions
    with open(file_name, "r") as f:
        # iterate over all lines in the file object "f"
        for line in f:
            # we are only interested in lines that start with "*"
            if line.startswith("*"):
                # we split the line on "whitespace"
                # the "_" means we are not interested in the first two entries
                _, _, frequency, intensity = line.split()

                # add the new numbers to the lists
                # we convert from string to float
                frequencies.append(float(frequency))
                intensities.append(float(intensity))

    # function returns the result
    return frequencies, intensities


# here we are outside the function
frequencies, intensities = read_data("example1.txt")

print(frequencies)
print(intensities)
```

One solution for `example2.txt`.

```python
def read_data(file_name):
    # we start with empty lists
    frequencies = []
    intensities = []

    # open the file with read permissions
    with open(file_name, "r") as f:
        # iterate over all lines in the file object "f"
        for line in f:
            # we are only interested in lines that start with "*"
            if "numbers we want" in line:
                # we split the line on "whitespace"
                words = line.split()
                # we are only interested in the last element
                # -1 means last element in that list
                last_element = words[-1]
                # convert from string to int
                num_measurements = int(last_element)

                # now we know the next 10 lines are the interesting ones
                # range(10) produces a list with 10 elements: [0, 1, 2, 3, 4, 5, 6, 7,
8, 9]
                # we use it to iterate exactly 10 times
                for _ in range(num_measurements):
                    # this advances f by one and we get one line at a time
                    next_line = next(f)
                    words = next_line.split()
                    frequency = float(words[0])
                    intensity = float(words[1])
                    # add the new numbers to the lists
                    frequencies.append(frequency)
                    intensities.append(intensity)

    # function returns the result
    return frequencies, intensities


# here we are outside the function
frequencies, intensities = read_data("example2.txt")

print(frequencies)
print(intensities)
```

Credit

When preparing this lesson, we have reused these resources:

- https://aaltoscicomp.github.io/python-for-scicomp/
- https://datacarpentry.org/python-ecology-lesson/
- https://swcarpentry.github.io/python-novice-inflammation/
- https://coderefinery.github.io/jupyter/
- https://coderefinery.github.io/data-visualization-python/