Python for Data Analysis

Table of contents

1 Python for Data Analysis

Download as PDF.

These are the materials which are used for a course on *Python for Data Analysis* at the University of Freiburg in summer 2026. It does not assume any prior knowledge of Python (although this might help). The goal of the course is some literacy in Python, including basic knowledge such as data types and control flow, but also using libraries and APIs, reading and dealing with data, up to some methods such as PCA and regression.

Basically, the course has three parts:

1. Python basics: data types, functions, loops, conditions, libraries (Chapters 2–5)
2. Tools for data analysis: numpy, pandas, matplotlib, scipy.stats (Chapters 6–9)
3. Project: choose data that interests you and create an analysis (Chapter 10)

Within the first two parts, you will be assigned homeworks, which you can find at the end of each section. For the third part, you will have to find a project by yourself. Chapter 11 collects miscellaneous topics (error messages, type hints, git, etc.) that are useful throughout the course.

Let us discuss some preliminaries.

Your Python environment: We will assume that Python and Jupyter is installed on your system. These course notes are written with Python 3.13, but other versions might work as well. In addition to plain `python`, we will need some libraries. Which ones we need (including their versions) is written in the file `requirements.txt`. Some Jupyter notebooks might not work if these are not installed on your system. (To be more precise, they must be installed on the Jupyter server which runs the notebooks.) It is recommended to use a virtual environment for managing your Python packages — see Section 11.6 for details.

General note: We avoid repeating the distinction between the commands `python` *and* `python3`*, and will simply write Python in the sequel.*

Starting a Jupyter notebook:

There are basically three ways Python commands can be run: 1. In interactive mode on the command line by typing `python3` in your shell, 2. In Python-scripts, usually ending with `myScript.py`. You can run these using `python3 myScript.py`. 3. Jupyter notebooks have the opportunity to mix usual text with Python commands.

In this course, we will only use option 3. For such Jupyter notebooks, there are at least two ways to run them:

1. Start the Jupyter server using `jupyter lab` (hopefully, it is already installed), which opens your browser where you can edit and save what you have done;
2. Use `vscode` as you IDE and work from there; see below.

Integrated Development Environment (IDE): When programming, we need to type code. This is possible in a code editor, and best done in a more sophisticated environment such as vscode or pycharm. It is best to activate your virtual environment (see Section 11.6) using `source venv/bin/activate` before starting your IDE. In order to start vscode, go to the main folder of the course notes, and type `code .` E.g., vscode can be used for all three ways to write Python commands:

1. Use `Terminal -> New Terminal` in order to open a terminal, type `python3` and you are in interactive mode,
2. Write a Python script and run it using the opened terminal,
3. Click on some Jupyter notebook, and vscode starts the Jupyter server for you.

A first Python cell: Here is the first cell of Python-code:

```
x = "Hello, "
y = "world!"
print(x+y)
```

```
Hello, world!
```

If you are working within the Jupyter notebook (and not just in the html-file), you can execute this piece of code: Press the play button on the left (or hit `Control + Enter` within the Python-cell).

Using AI (ChatGPT, Gemini etc, Copilot, Claude, etc):

During the last years, AI has increasingly improved in code writing. You are encouraged to use this new technology! As usual when learning new things, this new tool can help a lot, and will be able to solve the exercises within the Jupyter notebooks. However, you still have to undestand the code, and what might go wrong during execution. As a first course in Python, you will end up having up to a few hundred lines of code, which has to be coherent. You must know the structure of that code, and in order to extend it, fix bugs etc. If you are ever involved in bigger projects, this is even more important.

For these course notes, I have also used AI (ChatGPT, Gemini) in order to ask the AI which topics I should cover, and make concrete suggestions for a workflow. (While I have been programming in Python for some years now, my experience with `numpy` and `matplotlib` was

still limited when I started to write the course, since I have used `R` for such topics in the past.) Not all suggestions from the AI were great. However, I think that the end result for this course is much better than without the help of AI. I hope this also helps the learning experience. In particular, an earlier version of Chapter 11 (generated with ChatGPT) served as a source of material: its content on `numpy`, `pandas`, and `scipy.stats` was integrated into Chapters 06, 07, 08, and 09 with the help of Claude Code. A detailed log of all AI-assisted edits (including the prompts used) is available in `CLAUDE.md`.

2 Simple Data types

Download notebook.

2.1 Overview

Every **variable** or other object in Python has a **type**. This is the kind of information which is stored insice the variable or object Here is an overview over the most important simple data types.

Type	Example	Description
`int`	`5`	Integer
`float`	`3.14`	Floating-point number
`bool`	`True`	Boolean value
`str`	`"hello"`	Text string
`NoneType`	`None`	No value / null value

- You can find about the type of any expression `x` in Python using `type(x)`.
- In fact, `type(x)` is already the first example of a built-in-function. The function `type` takes a single value as input, and returns its type. We will encounter various more functions already in this section. We will learn more about functions (built-in and self-written) in Section 3.4.
- There are conversions between some types. For example, `"0"` is a string, but `int("0")` (again a function, taking a single value, and returning its integer value) the integer `0`. Conversely, `str(0)` is the string `"0"`. We will have more examples below.
- You can optionally annotate variables with their type using **type hints**, e.g. `x: int = 5`. Python does not enforce these annotations, but they help readability. See Section 11.5 for more details.
- We will use `print()` throughout to display output. It takes one or more values and prints them to the screen.

```
x: int = 5
y: float = 3.14
name: str = "Alice"
print(type(x), type(y), type(name))
```

```
<class 'int'> <class 'float'> <class 'str'>
```

2.2 Integers (`int`)

With **int**s (numbers ...,-2,-1,0,1,2,...), you can compute as usual.

```
x = 7
y = 2
print("x + y = ", x + y)
print("x * y =  ", x * y)
```

```
x + y =  9
x * y =   14
```

```
# x to the power of y
print("x ** y = ", x ** y)
```

```
x ** y =  49
```

```
# x divided by y (which gives a `float`)
print("x / y = ", x / y)
```

```
x / y =  3.5
```

```
# integer-valued division of x by y
print("x // y = ", x // y)
```

```
x // y =  3
```

```
# x modulo y
print("x % y = ", x % y)
```

```
x % y =  1
```

In most programming languages (e.g. `C++`), there is a minimal and a maximal possible value for an int. Python, however, does not have this limitation. Only the amount of available memory restricts the numbers which can be stored in an `int`.

2.3 Floating point numbers (`float`)

Generally speaking, **float**s are decimal numbers, which come with an **exponent** and a **mantissa**, e.g. $1.2 \cdot 10^2$. Python can do the same computations as with ints, but the following should be kept in mind:

- If you type `1`, Python thinks this is an int, but if you type `1.0` or `1.`, Python thinks this is a float.
- Unlike ints, floats do have a maximal and minimal element. The reason is that every float is stored in 64 bits, one for the sign, 11 for the exponent and 52 for the mantissa. As a result, floats range from about 10^{-324} to 10^{308}.
- When doing mixed computations with floats and int, e.g. `1.0 + 1`, the int is changed into a float.
- It is not possible to perform exact computations with some floats; see the example below.

```
x = 1.0
y = 1
print("x + y = ", x + y)
```

```
x + y =  2.0
```

```
print("0.1 + 0.2 = ", 0.1 + 0.2)
```

```
0.1 + 0.2 =  0.30000000000000004
```

Special floats are `inf`, `-inf` and `nan` (not a number). Let us see them in action:

```
print((10. ** 308) * 10)
```

```
inf
```

```
print((10 ** 308) * 10)
```

```
100000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
```

```
print((10 ** (-323)) / 10)
```

```
0.0
```

```
print((10. ** 308) * 10 - (10. ** 308) * 10)
```

```
nan
```

You can transform ints to floats and vice versa by using `float()` and `int()`.

```
print(float(0))
```

```
0.0
```

```
print(int(0.0))
```

```
0
```

```
print(int(0.5))
```

```
0
```

If you deliberately want to introduce `nan` or `inf`, you can do this:

```
print(float("inf"))
print(float("nan"))
```

```
inf
nan
```

On `float`, you can use many functions. We only name two at the moment, for some float `x`:

- `abs(x)`: The absolute value of `x`.
- `round(x)` and `round(x,n)`: Round `x` to the next `int`, or after `n` digits.

In Section 5.2, we will encounter that many more usual functions (like `sqrt`) can be used on floats, but we need to `import` the mathematical library for this.

2.4 True-False variables (`bool`)

There are exactly two **boolean** values, known as `True` and `False`. You can compute with them using `not`, `and`, `or` and 'xor'. In addition, `True` and `False` can be the result of a condition, such as the expression `1 == 2`.

```
x = True
y = False
print("not x =", not x)
```

```
not x = False
```

```
print("x and y =", x and y)
```

```
x and y = False
```

```
print("x or y =", x or y)
```

```
x or y = True
```

```
# xor (exclusive or, written as ^)
print("x xor y =", x ^ y)
```

```
x xor y = True
```

```
print("((not x) == y) =", (not x) == y)
```

```
((not x) == y) = True
```

```
print("(1 == 2) =", 1 == 2)
```

```
(1 == 2) = False
```

```
# Python converts `bool` into `int`
print("x + y =", x + y)
```

```
x + y = 1
```

```
# != means "not equal"
print("(1 != 2) =", 1 != 2)
```

```
(1 != 2) = True
```

```
# < refers to less than
print("(1 < 1) =", 1 < 1)
```

```
(1 < 1) = False
```

```
# <= refers to less than or equal
print("(1 <= 1) =", 1 <= 1)
```

```
(1 <= 1) = True
```

```
z = float("nan")
print("(z == z) =", z == z)
```

```
(z == z) = False
```

2.5 Text (`string`)

In many cases, **strings** are objects in quotation markes, such as in "'Hello, world!".

- **Operations on strings**: You can concencatenate strings using `+` as in "'Python" + "for" + "Data" + "Analysis" (but you cannot subtract strings). You can also repeat a string by multiplying it with an int.
- **f-strings**: They were introduced in Python 3.6. They are the way to go if you want to mix description and variables. They are written as `f"Hello, {name}"`. The `f` introduces the f-string. Within curly braces, there is a variable, which should be printed instead of the variable name.
- **Special characters**: There are many of them, we only mention `\t` (tabulator) and `\n` (new line) here.
- **Multiline strings**: Usually, Python interprets a new line such that all commands are finished. However, if you want to use a sttring spanning multiple lines, you don't want this. Multiline strings start and end with `"""` (a triple quotation mark).
- **Quotation marks**: In various cases, you can use `'` instead of `"`. Sometimes this is useful when nesting quotation marks.

- **Substrings**: We will learn more about substrings, but here is already a way to see if a substring occurs within a string or not. The value of `"th" in "Python"` amount to `True`.

```
x = "Hello,"
y = "world"
# Concatenating two (or more) strings
print(x + y + "!")
```

```
Hello,world!
```

```
# repeating a string
z = "bla"
print(z*3)
```

```
blablabla
```

```
# a special character
print("\U0001F60E")
```

```
# an example of an f-string
name = "Alice"
print(f"Hello {name}  ")
```

```
Hello Alice
```

```
# a multiline string
text = """This is a multiline string.
It can span several lines.
Each line break is preserved."""
print(text)
```

```
This is a multiline string.
It can span several lines.
Each line break is preserved.
```

```
# determine if a given substring occurs within a string
print("Is 'th' a substring of 'Python'? ", 'th' in 'Python')
print("Is 'the' a substring of 'Python'? ", 'the' in 'Python')
```

```
Is 'th' a substring of 'Python'?  True
Is 'the' a substring of 'Python'?  False
```

For strings, there are various build-in functions. We mention some of them. Here, `s` is a string.

- `str.strip(s)`: Deletes spaces in the front and back.
- `len(s)`: gives the length of the string.
- `str.replace(s, "x", "y")`: Replaces each occurrence of `"x"` in `s` by `"y"`.
- `str.split(s, "x")`: Splits `s` at each occurrence of `"x"`. The result is a `list`, which we will encounter in Section 4.1.
- `", ".join(list)`: Joins the elements of a list into a single string, separated by the given string.
- `str.lower(s)`: Return the same string, but all letters are lowercase.
- `str.upper(s)`: Return the same string, but all letters are uppercase.
- `str.isalpha(s)`: Returns `True` if the string consists only of alphabetic letters, and `False` otherwise.
- `str.isdigit(s)`: Returns `True` if the string is actually a number, and `False` otherwise.
- `str.isalnum(s)`: Returns `True` if the string consists only of letters and numbers, and `False` otherwise.
- `str.isspace(s)`: Returns `True` if the string consists only empty spaces, and `False` otherwise.
- `str.islower(s)`: Returns `True` if the string consists only of lower case letters, and `False` otherwise.
- `str.isupper(s)`: Returns `True` if the string consists only of upper case letters, and `False` otherwise.
- `str.count(s, "x")`: Counts how ofter `"x"` appears in `s`.

Note that for a function `str.something(s,...)`, you can also write `s.something()`. For example, `s.strip()` is the same as `str.strip(s)`. This is called dot-notation and is explained in Section 11.4.

2.5.1 Formatting numbers in f-strings

When printing numerical results, we often want to control the number of decimal places, alignment, or padding. Inside an f-string, a **format specifier** follows the variable after a colon: `f"{x:.4f}"`.

- `f"{x:.4f}"`: float with 4 decimal places.
- `f"{x:.2e}"`: scientific notation with 2 decimal places.
- `f"{x:>10}"`: right-align in a field of width 10.
- `f"{x:<10}"`: left-align in a field of width 10.
- `f"{n:,}"`: integer with thousands separator.
- `f"{x:.2%}"`: format as percentage with 2 decimal places.

```
pi = 3.141592653589793
print(f"pi = {pi}")
print(f"pi = {pi:.2f}")
print(f"pi = {pi:.2e}")
print(f"123456 = {123456:,}")
print(f"pi = {pi:.2%}")
```

```
pi = 3.141592653589793
pi = 3.14
pi = 3.14e+00
123456 = 123,456
pi = 314.16%
```

2.6 Missing things (`NoneType`)

When analysing data, it is common to have **missing values**. These usually come as `None`. This is different from anything else we encountered so far (including `nan` and `False`).

```
x = None
print(x)
```

```
None
```

The `is` operator checks whether two variables refer to the same object in memory, as opposed to `==` which checks whether they have the same value. For `None`, always use `is` rather than `==`.

```
print("(x is None) =", x is None)
```

```
(x is None) = True
```

```
print("(None == False) =", None == False)
```

```
(None == False) = False
```

```
print("(None == nan) =", None == float("nan"))
```

```
(None == nan) = False
```

2.7 Exercises

Exercise 1 Find out how to use scientific notation in Python. (If you don't know what this is, please find out!)

```
# Exercise 1
```

Exercise 2 Is there a Python package which can compute `0.1 + 0.2 == 0.3` without error?

```
# Exercise 2
```

Exercise 3 Print `"Hello, world!"` including the quotation marks in the output.

```
# Exercise 3
```

Exercise 4 If you are on Windows, it is annoying that file paths come with `\`, which is interpreted as the start of a special character. Find out two ways how to give the string of the file path `C:\Users\Alice\Documents\file.txt`.

```
# Exercise 4
```

Exercise 5 What is `bool(None)`? What is `int(None)`?

```
# Exercise 5
```

Exercise 6 In tables, we often have `""` for a missing value. Is this the same as `None`?

```
# Exercise 6
```

Exercise 7 Given a float `x`, write a one-liner that rounds `x` to 3 decimal places without using `round`. (Hint: use arithmetic with `int()`.)

```
# Exercise 7
```

3 Control flow – conditions, loops, functions, etc

Download notebook.

Coding starts to become interesting when we can control how variables change, and repeat operations.

3.1 Conditions (`if ... elif ... else`)

As a simple (mathematical) example, let us compute the minimum of two numbers., which works by an **`if`-condition**:

```
a = 2
b = 5
if a < b:
    res = a
else:
    res = b
print(f"The minimum of {a} and {b} is:", res)
```

```
The minimum of 2 and 5 is: 2
```

In general, the structure of an `if`-statement reads:

```
if _condition1_:
    case1
elif _condition2_:
    case2
elif _condition3_:
    case3
...
else:
    case_else
```

Here, read `elif` alse `else if`, which means that the code jumps into this case if the respective condition is the first to hold. The `else` case only applied if both the `if` and all `elif` conditions do not apply.

In fact, there are abbreviations in Python which are very handy. In particular, look at this piece of code.

```
res = a if a < b else b
print(f"The minimum of {a} and {b} is:", res)
```

```
The minimum of 2 and 5 is: 2
```

Let us make another example, also using `elif`. Given some `n`, the task is to print `Fizz` if `n` is a multiple of 3, `Buzz`, if it is a multiple of 5, and `FizzBuzz` if both conditions are satisfied.

```
n = 165
res = ""
if n % 3 == 0 and n % 5 == 0:
    res = "FizzBuzz"
elif n % 3 == 0:
    res = "Fizz"
elif n % 5 == 0:
    res = "Buzz"
print(res)
```

```
FizzBuzz
```

3.2 Loops with fixed length (`for`)

There are **`for`-loops** in Python, as discussed in this section, and `while`-loops in the next section. The basic use-case of a `for`-loop is repeating a task several times. As a simple example, adding `b` a number of `a` times, results in `a * b`.

```
a = 3
b = 7
res = 0
for i in range(a):
    res = res + b
print(f"{a} * {b} = {res}")
```

```
3 * 7 = 21
```

Importantly, `range(a)` is a synonym for the numbers `0,..., a-1`. (In particular, note that Python usually starts counting at `0` and not at `1`, which will also be important in the next chapter.)

The general structure of a `for`-loop is as follows:

```
for _var_ in _iterable:
    _sequence of instructions_
```

3.2.1 Greatest common divisor I

Let us look at a more interesting example, computing the greatest common divisor (gcd) of two numbers, `a` and `b`. A very simple algorithm is going through all numbers `1,...,` in order, and update result each time we encounter a number which divides both, `a` and `b`. (Here, `range(1,a+1)` is a synoym for the numbers `1,...,a`.)

```
# Basic algorithm for finding the gcd of two numbers
a = 105
b = 33

res = 1
for i in range(1,a+1):
    if a % i == 0 and b % i == 0:
        res = i
print(f"The gcd of {a} and {b} is {res}.")
```

```
The gcd of 105 and 33 is 3.
```

Let us make some remarks on `for`-loops: * In the last example, note that the instructios within the `for`-loop depend on `i`. * Within a `for`-loop, you can use `continue` in order to stop the execution of the current iteration, and immediately jump to the next `i`. * Within a `for`-loop, you can use `break` in order to stop the whole execution of the `for`-loop.

```
# continue skips the rest of the current iteration
for i in range(5):
    if i == 2:
        continue
    print(i, end=" ")
print()
```

```
0 1 3 4
```

```
# break stops the loop entirely
for i in range(10):
    if i == 5:
        break
    print(i, end=" ")
print()
```

```
0 1 2 3 4
```

3.3 Loops with unknown length (`while`)

The `for`-loop directly iterates over some variable (`i` above). The **`while`-loop** instead comes with a condition, which is checked everytime it is started new. The general structure is:

```
while _condition_:
    instructions which might change the value of _condition_
```

Let us make two examples: finding $\sqrt{2}$ and computing the gcd using Euclid's algorithm.

3.3.1 Finding $\sqrt{2}$

For finding $\sqrt{2}$, note that this is a fixed point for the iteration

$$x_{n+1} = \frac{x_n}{2} - \frac{1}{x_n}.$$

(In order to see this, assume that $x = x_n = x_{n+1}$, and multiply the recursion by x.) We can use this as follows. Here, `abs()` is a built-in function returning the absolute value of a number.

```
x = 1
eps = 1e-10
x_prev = 0

while abs(x - x_prev) > eps:
    x_prev = x
    x = x/2 + 1 / x
print(x)
```

```
1.414213562373095
```

3.3.2 Finding the greatest common divisor II

The above algorithm (using `for`-loops) for finding the gcd were not particulary efficient. (You might want to test them with some large numbers.) Euclid's algorithm is much more efficient. It is based on the observation that `c % a == 0` (`c` divides `a`) and `c % b == 0` iff `c % a == 0` and `c % (b % a)`.

In order to see this, note that `c % a == 0` and `c % b == 0` iff `c % a == 0` and `c % (b - da) == 0` for any `d`. Then, the result follows from choosing `d = b \\ a` since `b - (b \\ a) * a = b % a`. Moreover, every number divides `0`, so if `b % a == 0`, the gcd of `a` and `b` is `a`.

We can use these insights, taking `a < b` for simplicity. Starting with `a` and `b`, we compute if `b % a == 0`. If yes, we are done, and return `a`. If not, we compute the gcd of `a` and `b % a` instead.

```
# Euclid's algorithm for finding the gcd of two numbers
a_input = 105
b_input = 33
# Copy the input to new variables
a = a_input
b = b_input
# Run Euclid's algorithm
while a != 0:
    r = b % a
    print(f"b = {b}, a = {a}, remainder = {r}")
    b = a
    a = r
print(f"The gcd of {a_input} and {b_input} is {b}.")
```

```
b = 33, a = 105, remainder = 33
b = 105, a = 33, remainder = 6
b = 33, a = 6, remainder = 3
b = 6, a = 3, remainder = 0
The gcd of 105 and 33 is 3.
```

3.4 Functions (`def`)

Once we implement something, we wan to reuse it without re-writing the code. This means, we want to pack our code into its own **function** for reusing it. For Euclid's algorithm, this would look as follows:

```python
a = 105
b = 33

def gcd(a,b):
    """ Compute the greatest common divisor of a and b."""
    while a != 0:
        r = b % a
        b = a
        a = r
    return b

print(f"The gcd of {a} and {b} is {gcd(a,b)}.")
```

```
The gcd of 105 and 33 is 3.
```

The general structure is

```
def name_of_the_funcion(arg1, arg2, arg3 = default3, arg4=default4):
    some instructions
    return something
```

Not all functions return something. Those who don't usually produce some output. Here is an example

```python
def greeting(name, prefix="Hello"):
    """ Return a greeting for name."""
    print(f"{prefix}, {name}!")

greeting("Alice")
greeting("Bob", prefix="Hi")
greeting(prefix="Welcome", name="Charlie")
```

```
Hello, Alice!
Hi, Bob!
Welcome, Charlie!
```

Here are some important remarks on functions:

- Variables within functions are private to them. (There will be exceptions to this) In the gcd example, variable names `a` and `b` occur both, within and outside of the function. Although the variables `a` and `b`, which are private to the function, are changed within the function, the `print`-statement still knowns their value from before the function definition.
- Variables which are given to functions might have **default values**, as the `prefix = "Hello"` in the `greeting` example.
- For giving variables, there are two options:
  - you can rely on the position they have in the definition of the function. (Example: `greeting("Bob", prefix="Hi")`, where you rely on `"Bob"` being in the `name` position.) These are called **positional variables**.
  - you can give it the explicite variable name it has in the function definition. In this case, the order of variable names does not matter. (Example: In `greeting(prefix="Welcome", name="Charlie")`, the output was correct although the order of the two variables differs from the function definition.)
- All positional arguments must come before **keyword aguments**.
- Both functions from above have a **docstring**, which is the multiline string after the `def`-line. They indicate what this function is doing and are displayed in various places, e.g. when you hover over a function in your jupyter notebook.
- Sometimes, it is too much to define an own function for a very simple task. In this case, Python comes with **`lambda`-functions**, which are very quick: `lambda x: x * x`. Most use cases are with container data types, and we will have one in Section 4.1.

3.5 Exception handling (`raise`, `try ... except`)

In code, various things happen, and some go wrong. **Raising** (throwing) errors, **catching** them, and dealing with them is the topic of this section. There are many error types implemented. Let us look at some of them:

Exception	When it happens	Example
`ValueError`	Right type, wrong value	`int("abc")`
`TypeError`	Wrong type used	`"2" + 3`
`IndexError`	Index out of range	`[1,2][5]`
`KeyError`	Dictionary key missing	`{"a":1}["b"]`
`ZeroDivisionError`	Division by zero	`1 / 0`
`FileNotFoundError`	File does not exist	`open("x.txt")`
`PermissionError`	No access rights	opening protected file
`AttributeError`	Object has no attribute	`"abc".foo`
`NameError`	Variable not defined	`print(x)`
`ImportError`	Import fails	`import nonexisting`
`ModuleNotFoundError`	Module not found	`import xyz`

Exception	When it happens	Example
`AssertionError`	`assert` fails	`assert False`

We obtain an error here:

```
print("This will not work")
print(f"1/0 = {1/0}")
print("If this print works, the program has continued past 1/0")
```

```
This will not work

ZeroDivisionError: division by zero
```

```
---------------------------------------------------------------------------
ZeroDivisionError                         Traceback (most recent call last)
Cell In[12], line 2
      1 print("This will not work")
----> 2 print(f"1/0 = {1/0}")
      3 print("If this print works, the program has continued past 1/0")
ZeroDivisionError: division by zero
```

However, if we *catch* the error, the code can run past it:

```
try:
    print("This will not work")
    print(f"1/0 = {1/0}")
    print("This never runs")
except ZeroDivisionError as e:
    print("You must not divide by zero!")
    print("This is the error message:", e)
print("The program continues!")
```

```
This will not work
You must not divide by zero!
This is the error message: division by zero
The program continues!
```

Now we know how to *catch* errors, but often, we need to *throw* them. This is actually simple using `raise`:

```
def compute_sth(x):
    if type(x) != int:
        raise TypeError("x must be an int!")
    y = x ** x
    return y

try:
    print(compute_sth(2))
    print(compute_sth("abc"))
except TypeError:
    print("Calculation not possible, sorry")
```

```
4
Calculation not possible, sorry
```

3.6 File input and output, context management (`open`, `read`, `write`, `with`)

Reading and writing files is one of the most common tasks in data analysis. Python's built-in `open` function, combined with the `with` statement, provides a clean way to handle files. The `with` **context manager** handles opening and closing files for you. More generall, our Python file (or notebook) must interact with other files or ressources (databases, other connections). In order to do this safely, we are using `with`, which guarantees that the ressource is entered (opened) and exited (closed) correctly. The general structure is as follows:

```
with something as name:
    do_something()
```

Specifically for input and output from and to files, let `f` be an open file object. Then, the following are useful: * `open(path, mode)`: open a file. See below for some details on `mode`. * `f.read()`: read the entire file as a single string. * `f.readline()`: read a single line. * `f.readlines()`: read all lines into a list of strings. * `f.write(s)`: write string `s` to the file. * `f.writelines(lines)`: write a list of strings to the file.

```
with open("hello.txt", "w") as f:
    f.write("Hello")
    f.write("This is a new line in the file!\n")
    print("File written.")
```

```
File written.
```

```
with open("hello.txt", "r") as f:
    text = f.read()
print("This is the content of the file:", text)
```

```
This is the content of the file: HelloThis is a new line in the file!
```

```
# reading line by line (useful for large files)
with open("hello.txt", "r") as f:
    for line in f:
        print(line.strip())
with open("hello.txt", "r") as f:
    print(f.read())
```

```
HelloThis is a new line in the file!
HelloThis is a new line in the file!
```

Here, `"hello.txt"` is closed at the end of the `with` block. The second argument of `open` comes with the following modes: * `r`: read (text mode) * `w`: (over-)write (text mode) * `a`: append (text mode) * `x`: create new (i.e. throw error if already exists) * `rb`: read (binary mode) * `wb`: (over-)write (binary mode) * `ab`: append (binary mode) * `xb`: create new (i.e. throw error if already exists)

3.7 Exercises

Exercise 1 Do newlines and tabs `\n`, `\t` count in `str.isspace()`? How do numbers interact with `str.islower()`? If `s` is the string of a number, what is `s.lower()`? Play around a bit with these functions and edge cases you might see in data. Then write a function `classify_char(c)` that takes a single character and returns `"letter"`, `"digit"`, `"whitespace"`, or `"other"`.

```
# Exercise 1
```

Exercise 2 Compute $\sum_{i=1}^{100} i$ using a `for`-loop.

```
# Exercise 2
```

Exercise 3 Find the largest n such that $\sum_{i=1}^{n} i < 1000$.

```
# Exercise 3
```

Exercise 4 `for`-loops can also become nested. Compute $\sum_{i=1}^{100} \sum_{j=1}^{i} j$. (Note the double indentation for the inner `for`-loop!)

```
# Exercise 4
```

Exercise 5 Write a function (without using imports) which computes the sum of all digits (cross sum) of an int.

```
# Exercise 5
```

Exercise 6 Can you code the `FizzBuzz` example in a single line of code? So, dependent on some `n`, give one line of code, which gives `Fizz` if `n` is a multiple of 3, `Buzz`, if it is a multiple of 5, and `FizzBuzz` if both conditions are satisfied.

```
# Exercise 6
```

Exercise 7 Python (since version 3.10) comes with `match`, which is useful for branching on specific values. Write a function `season(month)` that takes a month number (1–12) and returns the season (`"Winter"`, `"Spring"`, `"Summer"`, `"Fall"`). Use `match` with grouped patterns (e.g. `case 3 | 4 | 5:`) and a wildcard `_` for invalid inputs. Find out how `match` differs from a chain of `elif`s.

```
# Exercise 7
```

Exercise 8 The following function is supposed to compute the average of a list of numbers, but it contains a subtle bug. Find and fix it. (Hint: try it with a few inputs of different lengths and compare the result to what you expect. Debugging with `print` statements or the built-in debugger can help — see Section 11.7.)

```
def average(numbers):
    total = 0
    for i in range(1, len(numbers)):
        total += numbers[i]
    return total / len(numbers)

print(average([10, 20, 30]))    # expected: 20.0
print(average([1, 2, 3, 4]))    # expected: 2.5
print(average([7, 8]))          # expected: 7.5
```

```
16.666666666666668
2.25
4.0
```

```
# Exercise 8
```

Exercise 9 Write a function `collatz(n)` that returns the number of steps it takes for the Collatz sequence starting at `n` to reach `1`. (The rule is: if `n` is even, divide by 2; if odd, compute `3n+1`.)

```
# Exercise 9
```

Exercise 10 Write a function `newton_sqrt(a, tol=1e-10)` that computes $\sqrt{a}$ using Newton's method, i.e. the iteration $x_{k+1} = \frac{1}{2}\left(x_k + \frac{a}{x_k}\right)$, starting from $x_0 = a$, and stopping when $|x_{k+1} - x_k| < \text{tol}$.

```
# Exercise 10
```

Exercise 11 Write a function `primes_up_to(n)` that returns a list of all prime numbers up to `n`, using the Sieve of Eratosthenes.

```
# Exercise 11
```

Exercise 12 Write a function `bisection(f, a, b, tol=1e-10)` that finds a root of a continuous function f on $[a, b]$ using the bisection method. The function should raise a `ValueError` if $f(a)$ and $f(b)$ have the same sign. Test it on $f(x) = x^3 - 2$.

```
# Exercise 12
```

Exercise 13 Write a function `caesar(s, k)` that shifts every letter in the string `s` by `k` positions in the alphabet (wrapping around from `z` to `a`). Non-letter characters should remain unchanged. (Hint: `ord(c)` returns the integer code of a character, e.g. `ord("a") == 97`, and `chr(n)` converts an integer back to a character, e.g. `chr(97) == "a"`.)

```
# Exercise 13
```

4 Container data types

Download notebook.

When working with data, we usually not only have *one* int or *one* str, but many of them. For this situation, there are the **container data types**, `list`, `tuple`, `set`, and `dict`.

4.1 Lists (`list`)

A **list** is the most common container data type in Python. Two fundamental operations on lists are **indexing** and **slicing**. Indexing means accessing a single element by its position (starting at 0), and slicing means extracting a sublist using `start:stop` notation (where the element at position `stop` is excluded).

```
# Indexing: access a single element by position
names = ["Alice", "Bob", "Charlie", "Donna"]
print(names[0])
print(names[2])
```

```
Alice
Charlie
```

```
# Slicing: extract a sublist
names = ["Alice", "Bob", "Charlie", "Donna"]
print(names[1:3])
print(names[:2])
```

```
['Bob', 'Charlie']
['Alice', 'Bob']
```

The notation for a list comes with square brackets, where the entries are separated by a comma. The list of numbers 1, 2, 3 is `[1,2,3]`. A list of names is e.g. `["Alice", "Bob", "Charlie", "Donna"]`. So, for a `list`, the entries have simple (or again container) data types.

- We can access the entries of a list using square brackets. However, we have to be careful if the length of the list is too short.
- We can access sublists by using `i:j` for the entries `i,...,j-1`. If we want to have the beginning until `j-1`, we use `:j`. If we want everything after `i`, we use `i:`.
- Elements of a list can be lists (or other container data types) themselves.
- The types of elements of a list can be mixed.
- When talking about `for`-loops, we already had `range(n)`, which stands for `0,...,n-1`. This is not exactly a list, but can be turned into one by `list(range(n))`. The `range` function also accepts `range(start, stop, step)`, e.g. `range(0, 10, 2)` gives `0, 2, 4, 6, 8`. In general, `range(start, stop, step)` produces `start, start + step, start + 2*step, ...`, stopping before `stop`.

```
# a list of ints
x = [1,2,3]
print(x[0])
# len(x) gives the length of x
print(x[3] if len(x) > 3 else None)
```

```
1
None
```

Slicing refers to sublists.

```
x = [1,2,3]
print(f"The first and second element: {x[0:2]}")
print(f"The first two elements: {x[:2]}")
print(f"All elements after the first element: {x[1:]}")
```

```
The first and second element: [1, 2]
The first two elements: [1, 2]
All elements after the first element: [2, 3]
```

Playing around with `range`.

```
print(f"The first and second odd number: {list(range(1, 5, 2))}")
print(f"The first and second odd number: {list(range(1, 100, 2))[:2]}")
```

```
The first and second odd number: [1, 3]
The first and second odd number: [1, 3]
```

The elements of a list can be lists themselves.

```
y = [[1,2], [3,4,5], 6, 7, 8]
print(y[1])
print(y[1][0])
```

```
[3, 4, 5]
3
```

A list can have mixed types.

```
z = [1, "two", [3,4], 5.0]
print(z[1])
```

```
two
```

Here are some functions on lists.

- `len(l)`: length of `l`.
- `sum(l)`: sums all elements of `l`, if possible.
- `min(l)`, `max(l)`: computes the minimum/maximum of `l`.
- `sorted(l)`: returns a sorted version of `l`.
- `filter(fun, l)`: returns a filtered list, which only contains elements `x` of `l` with `fun x == True`.
- `list.append(l, x)` (or `l.append(x)`) (where `x` is any object) appends the element `x` to the list `l`.
- `list.extend(l, x)` (or `l.extend(x)`) (where `x` is a list) extends `l` by all elements of `x`. (You can also write `l + x` for this.)
- `list.insert(l, i, x)` (or `l.insert(i,x)`) inserts `x` at position `min(i, len(l))` in `l`.
- `list.remove(l, x)` (or `l.remove(x)`) removes the first occurrence of `x` in `l`.
- `list.pop(l)` (or `l.pop()`) removes the last element from `l`.
- `list.index(l, x)` (or `l.index(x)`) gives the index of the first occurrence of `x` in `l`.
- `list.count(x)` (or `l.count(x)`) counts the number of occurrences of `x` in `l`.
- `list.sort(l)` (or `l.sort()`) sorts the list.
- `list.reverse(l)` (or `l.reverse()`) reverses the list.

For the functions starting with `list`, there is a catch: While we mentioned that a function takes its input (here the list), and does not alter it, `lists` are **mutable** objects, which means that they are changed when calling such a function on them. So, e.g. after `l.sort()`, the list `l` itself is sorted, and there is no return value of that function (more precisely, the return value is `None`.) This can be counterintuitive:

```
l = [3,2,1]
# This gives none since the return value of l.sort() is None
print(l.sort())

l = [3,2,1]
# This changes l and makes it sorted.
l.sort()
# Now we can print it
print(f"l = {l}")

# In contrast, sorted() does not change the list but returns a new sorted list.
l = [3,2,1]
print(sorted(l))
```

```
None
l = [1, 2, 3]
[1, 2, 3]
```

Assume you want to sort a `list` not as usual. In this situation, you will find out about the `key`-parameter in `sorted`. It takes a function, we sorts the list according to the values of this function. Here, we use `lambda` to define short anonymous functions inline: `lambda x: expression` creates a function that takes `x` and returns `expression` (see also Section 3.4 in Chapter 3). As an example, assume we want to sort a list of `int`s according to their abolute value:

```
l = [5, -4, 0, 7, -3, 2]
print(sorted(l, key = lambda x: abs(x)))
```

```
[0, 2, -3, -4, 5, 7]
```

Similarly, you can e.g. filter the list by only taking numbers which are at least one:

```
l = [5, -4, 0, 7, -3, 2]
print(list(filter(lambda x: x >= 1, l)))
```

```
[5, 7, 2]
```

Let us shortly describe two functions on a list of `bool`s `l`.

- `all(l)` gives `True` if all entries of `l` are 'True.
- `any(l)` gives `True` if at least one entry of `l` is 'True.

4.2 Tuples (`tuple`)

At first glance, a **tuple** is like a `list`, but comes with normal brackets `()` instead of square brackets `[]`. So, `(1,2,3)` is the tuple of these three numbers, and `("Alice", "Bob", "Charlie")` is a tuple of three names. For the tuple`l`, still,`l[0]`gives the first entry etc. The main difference between`tuple`and`list`is that`list`s are more flexible when adding or changing entries, and`tuple`s are fixed (**immutable**) once they have been created.

```
l = ("Alice", "Bob", "Charlie")
# The next line does not work, since l is immutable
# l[0] = "Anna"
l = ["Alice", "Bob", "Charlie"]
# The next line works, since l is mutable.
l[0] = "Anna"
print(f"l = {l}")
```

```
l = ['Anna', 'Bob', 'Charlie']
```

An important aspect of tuples is that single values can easily be packed into a tuple. An important example is to use a tuple in order to assign two or more values at the same time:

```
def min_max(a,b):
    (mi, ma) = (a if a < b else b, b if a < b else a)
    return mi, ma

a = 20
b = 10
res = min_max(a,b)
print(f"Minimum and maximum are {res[0]} and {res[1]}.")
```

```
Minimum and maximum are 10 and 20.
```

A special note applies to tuples of length 1. Here, note that e.g. `(1)` does not create a `tuple` of length 1, but an `int`. Instead, use `(1,)` for the `tuple` of length one.

```
thisisnotuple = (1)
print(type(thisisnotuple))
print(thisisnotuple[0])
thisisatuple = (1,)
print(type(thisisatuple))
print(f"Its first and only element is {thisisatuple[0]}.")
```

```
<class 'int'>

TypeError: 'int' object is not subscriptable
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[13], line 3
      1 thisisnotuple = (1)
      2 print(type(thisisnotuple))
----> 3 print(thisisnotuple[0])
      4 thisisatuple = (1,)
      5 print(type(thisisatuple))
TypeError: 'int' object is not subscriptable
```

4.3 Sets (`set`)

As in mathematics, **set**s have no special order, and there are noduplicates within a set. In Python, they can be created using curly brackets, so e.g. `{1,2,3}` as well as `{1,3,2,3,1}` is the set of the numbers `1,2,3`. Although there are functions for updating sets (as for lists), the most frequent application is for removing duplicates. For example, if you have a `list`, which can contain duplicates, `list(set(l))` is the same list (maybe the order changed) but with duplicates removed.

Here are some functions on some `s` and `t`: * `len(s)`: returns the number of unique elements in `s`. * `s | t`: The union of `s` and `t`. * `s & t`: The intersection of `s` and `t`. * `s - t`: The difference of `s` and `t`.

```
l = [1,3,2,4,3,2]
print(f"There are {len(set(l))} unique elements in {l}.")
```

Unions, intersection, and set differences work as expected:

```
s = {1,2,3}
t = {5,4,3}
print(f"The union of {s} and {t} is {s | t}.")
print(f"The intersection of {s} and {t} is {s & t}.")
print(f"The difference of {s} and {t} is {s - t}.")
```

4.4 Dictioanries (`dict`)

Dictionaries (`dict`) are nothing but a collection of **key-value pairs**. Here is an example:

```
d = {
        "first_name": "Alice",
        "last_name": "Müller",
        "age": 29,
        "hobbies": ["reading", "cycling", "chess"]
    }
print(f"""{d['first_name']} {d['last_name']} is {d['age']} and likes
    {', '.join(d['hobbies'])}.""")
```

```
Alice Müller is 29 and likes
    reading, cycling, chess.
```

- As you see in the example, the f-string uses nested quotation marks, `"...'...'..."`. For this reason, we use single quotation marks `''` as inner separators and double quotation marks `""` for the outer quotation marks. If there is no nesting, there is no difference in using single/double quotation marks.
- keys in a dict can be any immutable data type (like `int`, `str`, `float`, `tuple`).

As you see, accessing field `'age'` in the `dict p` is done using `p['age']`. However, assume `'age'` is not a field in `p`. Then, `p['age']` gives an error. For this reason, it is often better to use the `get`-function:

```
print(f"{d['first_name']} {d['last_name']} lives in {d.get('address')}.")
# This version uses a default field in get
print(f"""{d['first_name']} {d['last_name']} lives
    in {d.get('address', 'unknown')}.""")
# We can also add "address" to the dict
d["address"] = "Freiburg, Germany"
print(f"""{d['first_name']} {d['last_name']} lives
    in {d.get('address', 'unknown')}.""")
```

```
Alice Müller lives in None.
Alice Müller lives
    in unknown.
Alice Müller lives
    in Freiburg, Germany.
```

Some of the most important functions on `d` as a `dicts` are: * `dict.keys(d)` (or `d.keys()`): Returns the list of all keys. * `dict.values(d)` (or `d.values()`): Returns the list of all values. * `dict.items(d)` (or `d.items()`): Returns the list of tuples (key, value).

```
print(f"The keys are {', '.join(d.keys())}.")
print(f"The values are:")
for v in d.values():
    print(str(v))
print(f"The key-value-pairs are:")
for key, value in d.items():
    print((key, str(value)))
```

```
The keys are first_name, last_name, age, hobbies, address.
The values are:
Alice
Müller
29
['reading', 'cycling', 'chess']
Freiburg, Germany
The key-value-pairs are:
('first_name', 'Alice')
('last_name', 'Müller')
('age', '29')
('hobbies', "['reading', 'cycling', 'chess']")
('address', 'Freiburg, Germany')
```

Let us mention some more functions on `dict`s:

- `dict.update(d, d')` (or `d.update(d')`): extends `d` by the keys in `d'`, and overwrites entries in `d` by the values of `d'`. Another possibility is `d |= d'` (similar to a set-union).
- `dict.pop(d, key, default)` (or `d.pop(key, default)`): removes the key `key` from `d` and returs its value (or `default`, if the key isn't found).
- `del d[key]`: Deletes the entry with key `key` from `d`.

```
d |= {
    "profession" : "student",
    "subject" : "Mathematics"
}
# del removes an entry from the dict
del d["age"]
for key, value in d.items():
    print((key, str(value)))
```

```
('first_name', 'Alice')
('last_name', 'Müller')
```

```
('hobbies', "['reading', 'cycling', 'chess']")
('address', 'Freiburg, Germany')
('profession', 'student')
('subject', 'Mathematics')
```

Here is a last example for a list of dicts.

```
people = [
    {
        "first_name": "Alice",
        "last_name": "Müller",
        "age": 29,
        "hobbies": ["reading", "cycling", "chess"]
    },
    {
        "first_name": "Bob",
        "last_name": "Schmidt",
        "age": 35,
        "hobbies": ["hiking", "photography", "cooking"]
    },
    {
        "first_name": "Clara",
        "last_name": "Weber",
        "age": 22,
        "hobbies": ["painting", "piano", "running"]
    },
    {
        "first_name": "David",
        "last_name": "Fischer",
        "age": 41,
        "hobbies": ["gardening", "woodworking", "cycling"]
    }
]
for p in people:
    print(f"{p["first_name"]} {p["last_name"]} is {p["age"]} and likes "
        + ", ".join(p["hobbies"]) + ".")
```

```
Alice Müller is 29 and likes reading, cycling, chess.
Bob Schmidt is 35 and likes hiking, photography, cooking.
Clara Weber is 22 and likes painting, piano, running.
David Fischer is 41 and likes gardening, woodworking, cycling.
```

4.5 Combining lists (`zip`, `enumerate`)

When working with multiple lists in parallel, `zip` and `enumerate` are very useful built-in functions.

- `zip(a, b)`: combines two (or more) lists element-wise into pairs (tuples). Stops at the shorter list.
- `enumerate(a)`: yields pairs `(index, element)` for each element of `a`.

```
names = ["Alice", "Bob", "Charlie"]
scores = [85, 92, 78]

# zip: iterate over two lists in parallel
for name, score in zip(names, scores):
    print(f"{name} scored {score}")
```

```
Alice scored 85
Bob scored 92
Charlie scored 78
```

```
# enumerate: get index and value
for i, name in enumerate(names):
    print(f"{i}: {name}")
```

```
0: Alice
1: Bob
2: Charlie
```

```
# zip in a list comprehension
diffs = [a - b for a, b in zip([10, 20, 30], [3, 7, 12])]
print(diffs)
```

```
[7, 13, 18]
```

4.6 Comprehensions

Python has very expressive (i.e. short) ways to alter container data types, called **comprehensions**. The general form is (where "iterable" is a container data type):

```
[expression for variable in list if condition]
[expression_if_true if condition else expression_if_false for variable in list]
{key: value for item in dict if condition}
```

Let us make some examples, and you will see how powerful these complehensions are!

```
# The squares of all even numbers below 10
print([x*x for x in range(10) if x % 2 == 0])
# The names of people who like cycling
print("People who like cycling are " +
    ', '.join([f'{p["first_name"]} {p["last_name"]}' for p in people
        if "cycling" in p['hobbies']]) + ".")
```

```
[0, 4, 16, 36, 64]
People who like cycling are Alice Müller, David Fischer.
```

4.7 Exercises

Exercise 1 Find out how you can sort a list descendingly.

```
# Exercise 1
```

Exercise 2 Write a function `is_palindrome(s)` that returns `True` if the string `s` reads the same forwards and backwards (e.g. `"racecar"`), and `False` otherwise. Ignore upper/lower case. (Hint: you can reverse a string using slicing.)

```
# Exercise 2
```

Exercise 3 Given a list of names [“ALICE”, “BOB”, “CHARLIE”], convert them all to lowercase.

```
# Exercise 3
```

Exercise 4 From a list of fruits [“apple”, “banana”, “kiwi”, “mango”, “pear”], create a list of fruits that have exactly 5 letters.

```
# Exercise 4
```

Exercise 5 Create a list of tuples representing all possible `(x, y)` coordinates where `x` is from `[1, 2, 3]` and `y` is from `[4, 5]`.

```
# Exercise 5
```

Exercise 6 Given the dict `{"a": 1, "b": 2, "c": 3}`, create a new dictionary that swaps keys and values `{1: "a", ...}`. This is best done using a comprehension.

```
# Exercise 6
```

Exercise 7 For two strings `s` and `t`, write a function `myCount(s, t)`, which counts the numbers of occurrences of `t` in `s`. Unlike `count`, it should count the number of positions in `s` starting with `t`. For example, `"bababab".count("bab")` gives 2, but `"bababab".myCount("bab")` should give 3.

```
# Exercise 7
```

Exercise 8 Write a function which removes duplicates from a list (you might want to use `set` here), but keeps the order of the list.

```
# Exercise 8
```

Exercise 9 Assume you have `a = [10, 20, 30, 40]` and `mask = [True, False, True, False]`, you might want obtain a sublist of `a` which only contains the positions where `mask` has `True` (without using other libraries). (You will probably need the `enumerate`- or `zip`-function here.)

```
# Exercise 9
```

Exercise 10 Given a list of integers, write a function `most_frequent(lst)` that returns the element that appears most often. If there is a tie, return any of the most frequent ones. (Hint: use a `dict` to count.)

```
# Exercise 10
```

Exercise 11 Write a function `flatten(lst)` that takes a nested list (e.g. `[[1, 2], [3, [4, 5]], 6]`) and returns a flat list `[1, 2, 3, 4, 5, 6]`. Use recursion.

```
# Exercise 11
```

Exercise 12 Write a function `invert_dict(d)` that swaps keys and values of a dictionary. If multiple keys have the same value, collect the keys in a list. For example, `{"a": 1, "b": 2, "c": 1}` becomes `{1: ["a", "c"], 2: ["b"]}`.

```
# Exercise 12
```

Exercise 13 A sparse vector can be represented as a `dict` mapping indices to nonzero values, e.g. `{0: 3.0, 5: -1.0, 99: 2.0}` for a vector that is zero everywhere except at positions 0, 5, 99. Write functions `sparse_add(v, w)` and `sparse_dot(v, w)` that compute the sum and inner product of two sparse vectors.

```
# Exercise 13
```

5 Some useful libraries/packages/modules

Download notebook.

In this chapter, we take a little break. Although there are lots of things to learn, the material covered here has the role of helping to learn about the Python ecosystem and what is possible, rather than learn the details of various labraries. The first section, however, is central and will be needed in any Python project you will start.

5.1 Loading modules (`import`)

As in all programming languages, not all Python programs come in a single file. If a Python project spans multiple files, an `import` becomes necessary. In Python, there is syntax for `import`s as follows:

Syntax	Meaning
`import module`	full module
`import module as name`	alias
`from module import x`	specific objects
`from module import *`	everything (avoid)
`import module.submodule`	submodule

Generally, there are **packages** (folders), **modules** (a `.py`-file within a package, or a single `py`-file), and **submodules**. Every submodule is a module. In any case, the import loads the (sub-)module, and executes all top-level code inside it (i.e. everything which is not inside a function).

There are three cases:

1. You wrote some code in file `one.py` (e.g. some functions) which you want to use in file `two.py`. (Note, btw, that the first digit in `one.py` must be a letter.) Assume `one.py` is as follows:

```
# File: one.py

def myConst():
    return 42
```

Then, the import in `two.py` is either `import one` (both files must be in the same folder), or `from one import myConst` or (not recommended) `from one import *`. In the first case, you must prefix `myconst()` by `one`:

```
# File: two.py

import one
print(f"My constant is {one.myConst()}.")
```

In the second case, there is no prefix:

```
from one import myConst
# or
from one import *
print(f"My constant is {myConst()}.")
```

Observe that the latter option comes with the disadvantage that you cannot see from the call of `myconst()`, if it is defined in `two.py` or in`one.py`, so it becomes harder to keep track of your code if things become more complicated.

If you have a folder `myPackage`, which holds `one.py`, you can must write `import myPackage`, `import myPackage.one` or `import myPackage.one as myNameOne` with differing ways to use `myConst()`:

```
import myPackage
print(f"My constant is {myPackage.one.myConst()}.")
```

```
import myPackage.one
print(f"My constant is {one.myConst()}.")
```

```
import myPackage.one as myName
print(f"My constant is {myName.myConst()}.")
```

2. You want to use functions from a built-in library. Then, a simple `import module` is often the best choice. We will see examples in Section 5.2.

3. You want to use functions from a library, which needs to be installed. Here, before the `import module` even works, you need to install the module in your Python environment. In a terminal, within your Python environment, use

```
# Uncomment the next line and run the cell for installation.
# This only needs to be done once for every package.
# If pip3 does not work, use pip instead.
# !pip3 install numpy
```

and then, `import numpy` will work. We will see examples for this e.g. in Section 7.8.

5.2 Mathematics (`math`)

The mathematical library in Python is used by `import math`. Then, you have e.g. the following functions, which are self-explanatory: `math.sqrt()`, `math.exp()`, `math.log()`, `math.sin()`, `math.cos()`, and constants like `math.pi` and `math.e`.

5.3 Dates and times of day (`datetime`)

In data science, we often have to handle **dates**. Their disadvantage is that there is no standard way to write these as strings. In Python, this and various other problems around dates and **times** is solved by the `datetime`-library.

Submodules (in fact, these are `class`es in this case) of this package are

- `date`: handling dates
- `time`: handling times
- `datetime.datetime`: handling `datetimes`, i.e. dates and times together
- `timedelta`: time differences
- `tzinfo` and `timezone`: handling timezones

We will here use

```
import datetime
```

at the cost that `datetime`-objects are only accessible by saying `datetime.datetime` (the first `datetime` being the package, the second `datetime` being the class/submodule).

Let us create a `datetime`-object‘:

```
dt = datetime.datetime(2026, 4, 21, 12, 15)
print(dt)
print(f"year: {dt.year}")
print(f"month: {dt.month}")
print(f"day: {dt.day}")
print(f"hour: {dt.hour}")
print(f"minutes: {dt.minute}")
print(f"second: {dt.second}")
print(f"microsecond: {dt.microsecond}")
print(f"date: {dt.date()}")
print(f"time: {dt.time()}")
```

```
2026-04-21 12:15:00
year: 2026
month: 4
day: 21
hour: 12
minutes: 15
second: 0
microsecond: 0
date: 2026-04-21
time: 12:15:00
```

which is the starting time of the first session in this course. As we see here, we can extract year,..., microseconds from the `datetime`-object.

Often, we are given strings such as `"21.4.26, 12:15"`, and want to convert this into a `datetime` object. For this, we have `datetime.datetime.strptime`:

```
s = "21.4.26, 12:15"
dt = datetime.datetime.strptime(s, "%y.%m.%d, %H:%M")
print(s + f" is converted to {dt}.")
```

```
21.4.26, 12:15 is converted to 2021-04-26 12:15:00.
```

(For the correct way to place %, y, m, d etc., it is in fact best to consult your favourite AI.)

Reversely, we want to print a `datetime`-object in a certain way. For this, we have `datetime.datetime.strftime`:

```
print(dt.strftime("%-m/%-d/%y, %H:%M"))
```

```
4/26/21, 12:15
```

which is the US-style output.

Next, assume we want to compute the same time 12 weeks from now. For this, we use `datetime.timedelta`.

```
print(dt + datetime.timedelta(days=84))
```

```
2021-07-19 12:15:00
```

Here are some more remarks:

- `datetime.date` is not used very often. The reason is that you can recover a `date` from a `datetime`, but not the other way round.
- `datetime.time` has the same limitations.
- We will not cover time-zones here, but if you have global data, they will become important!

5.4 Decimal numbers (`decimal`)

We already saw that:

```
print(f"Rounding error: 0.2 + 0.1 = {0.2 + 0.1}.")
```

```
Rounding error: 0.2 + 0.1 = 0.30000000000000004.
```

The reason is that computers in general use a base of 2 for all computations, and 0.3 has a bad representation in this system. While this small **rounding error** is acceptable often, it is not in some contexts (above all, finance). In this case, we can use the `decimal` library. Within `decimal`, a base of 10 is used, but calculations are slower than for `floats`.

```
from decimal import Decimal
print(f"Exact calculation: 0.2 + 0.1 = {Decimal('0.2') + Decimal('0.1')}.")
```

```
Exact calculation: 0.2 + 0.1 = 0.3.
```

5.5 Regular expressions (`re`)

Sometimes, you need to find **patterns** in a string. In data science, this happens if some field contains unstructured text, and you want to extract information. As an example, assume you load data from a csv-file, and one columns contains all email-adresses of the corresponding item. If you read this, you have something like `"muller@gmail.com; muller@yahoo.com"`. However, you want to separate the two addresses, and store them in a `list`. One way to approach this is using a **regular expression** (which in this case reads `"\S+@\S+\.\S+"`):

```
import re

text = "muller@gmail.com; muller@yahoo.com"

emails = re.findall(r"\S+@\S+\.\S+", text)
print(emails)
```

```
['muller@gmail.com;', 'muller@yahoo.com']
```

Let us quickly break this down to some extent, looking at `"\S+@\S+\.\S+"`: * `\S`: any non-whitespace character * `+`: one or more of the previous thing * `@`: exactly the `@` symbol * `\.`: exactly the dot `.` character (separating e.g. gmail from com)

We can agree that this exactly what an email address looks like. The `re.finall()` finds all places in `text` which match this pattern and extracts them, putting them into a list. Note that there are more patterns to find than `\S`, `+` and exact matches. For this reason, it is advised to use AI when it comes to regular expressions.

5.6 Storing and restoring objects (`pickle`)

Sometimes, you want to store intermediate objects. One way is to write them out in files, but you can also use the `pickle`-library in order to **pickle** (i.e. serialize) Python objects to disc. Here is an example:

```
import pickle

data = {"name": "Alice", "age": 30}

# Save to file (pickle)
with open("data.pkl", "wb") as f:
    pickle.dump(data, f)
```

```
# Load from file (unpickle)
with open("data.pkl", "rb") as f:
    loaded_data = pickle.load(f)

print(loaded_data)
```

```
{'name': 'Alice', 'age': 30}
```

5.7 Testing code (`pytest`)

```
# Uncomment the next line and run the cell for installation.
# If pip3 does not work, use pip instead.
# !pip3 install pytest
```

When starting to code, producing reliable programs is both important and difficult. One – not the best – way to approach this is to rely on the program until something breaks. It is actually better to *test* the code, i.e. to write **tests**. This means that you tell your program that you expect a certain behavior.

A simple and powerful library for this is `pytest`. When working with `.py`-files, you usually separate such tests in a separate file. Every test-case is within a function with name starting with `test_`. In a terminal, you can run `pytest` on that file, which performs all tests, and reports any error.

However, we are working with Jupyter notebooks here, so we cannot run `pytest` on separate files. Although there are workarounds (the package `ipytest`), we will only go through some basics here, without importing `pytest`. This means we are introducing `assert`.

```
x = 5
y = -1
assert 2 + 2 == 4
assert 2 + 2 == 5
```

```
AssertionError:
---------------------------------------------------------------------------
AssertionError                            Traceback (most recent call last)
Cell In[12], line 4
      2 y = -1
      3 assert 2 + 2 == 4
----> 4 assert 2 + 2 == 5
AssertionError:
```

5.8 Using the operating system (os)

For the kind of projects we envision, we often must access files, which happens via the **operating system**. We might also want to see all files in some directory, access them one by one, etc. Here, `folder` is the string of the name of a folder, and `file` is the string for the name of a file. * `os.getcwd()`: returns the current working directory. * `os.chdir(folder)`: this is a Python command for `cd`. * `os.path.join(folder", file)`: gives a complete path from the folder and file name. * `os.listdir(folder)`: returns a list of files within the folder. * `os.path.exists(file)`: returns true or false if the file exists or not. * `os.walk(folder)`: gives the triple `(root, dirs, files)`, where `root` is the absolute path of the folder, `dirs` are all direct subfolders in that folder, and `files` are the files in that folder. If folder has subfolders, it returns a list of triples for all subfolders.

The string method `.endswith(suffix)` checks whether a string ends with the given suffix and returns `True` or `False`.

```
import os
# Print all ipynb-files in the current folder
for file in os.listdir("."):
    if file.endswith(".ipynb"):
        print(file)
```

The built-in function `enumerate(iterable)` returns pairs `(index, element)` for each element, which is useful when you need both the position and the value during iteration.

```
# Print folders and files (only first 5 entries of os.walk)
for i, (root, dirs, files) in enumerate(os.walk(".")):
    if i >= 5:
        print("... (truncated)")
        break
    print("ROOT:", root)
    print("DIRS:", dirs[:5], "..." if len(dirs) > 5 else "")
    print("FILES:", files[:5], "..." if len(files) > 5 else "")
    print()
```

```
ROOT: .
DIRS: ['latex', 'misc', '05_libraries_files', 'site_libs', '.claude'] ...
FILES: ['data.pkl', 'index.qmd', '06_numerics.qmd', '05_libraries.html', '.gitignore'] ...

ROOT: ./latex
DIRS: []
FILES: ['gcd.log', 'gcd.aux', 'gcd.pdf', 'gcd.tex']
```

```
ROOT: ./misc
DIRS: []
FILES: ['fake_data.py', 'person.csv', 'person.db', 'person.json']

ROOT: ./05_libraries_files
DIRS: ['figure-html', 'mediabag']
FILES: []

ROOT: ./05_libraries_files/figure-html
DIRS: []
FILES: []

... (truncated)
```

5.9 Starting other programs (`subprocess`)

Sometimes, you want to start some piece of software (a command line tool) from within your Python program, and collect the result. You could use a script for this, which starts the command line tool, and rus your Python script afterwards, but you can as well use Python for the whole setting. We only give a basic example:

```
import subprocess
filename = "file.tex"
try:
    result = subprocess.run(
        ["pdflatex", "-interaction=nonstopmode", filename],
        capture_output=True, # capture stdout and stderr output instead of printing
        text=True, # returns output as string, not binary
        check=True # raises an error if the command fails
    )
    print(result.stdout)
except FileNotFoundError:
    print(f"File {filename} not found.")
```

5.10 Exercises

Exercise 1 What is your age in seconds/minutes/hours/days/months?

```
# Exercise 1
```

Exercise 2 Write a function `days_until_birthday(birthday_str)` that takes a date string like `"1990-03-15"` and returns the number of days until the next occurrence of that birthday. (Hint: use `datetime.date`.)

```
# Exercise 2
```

Exercise 3 Write a function `extract_emails(text)` that uses a regular expression to find all email addresses in a string. Test it on `"Contact us at info@example.com or support@example.org, but not at wrong-email."`. (Hint: a simple pattern like `r"[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}"` is a good starting point.)

```
# Exercise 3
```

Exercise 4 Create some `txt`-files in the current working directory. Then, write a Python program that iterates through all such files, creates `all.txt`, where the content of all your `txt`-files is appended. If `all.txt` already exists, the program should give a warning that this file is overridden. (Hint: you can generate fake text files using the `faker` library, e.g. `from faker import Faker; fake = Faker(); fake.text()`.)

```
# Exercise 4
```

Exercise 5 Below are some sales dates and quantities.

- Write a function, with `start` and `end` as `datetime` as input, which gives you the total amount of sales during this time. (You will need to convert `date` as given in the data to `datetime`. Note that `<`, `>`, `<=`, `>=`, `==` work for `datetime` as expected.)
- Write a function, with a date `date` and a number days `days` as input, which returns the average amount of sales per day during the last `days` since `date`.

```
data = [
        ("2026-01-01", 120),
        ("2026-01-02", 135),
        ("2026-01-03", 128),
        ("2026-01-08", 150),
        ("2026-02-01", 210),
        ("2026-02-03", 215)]
```

```
# Exercise 5
```

Exercise 6 Write a function `find_large_files(directory, min_size_mb=10)` that uses `os.walk` to find all files in `directory` (and subdirectories) that are larger than `min_size_mb` megabytes. Return a list of `(path, size_mb)` tuples, sorted by size descending.

```
# Exercise 6
```

Exercise 7 A bank manager has an idea: Interests should be computed daily (as usual), but rounded to cents (round up if last digit is 5 or above) at the end of the day. (So, sub-cents are not kept for computing the account balance.) Assume someone has invested 1 Euro on January 1st, 0 (birth of christ) with an interest rate of 5% per year. * Compute the account balance *as usual*, i.e. keeping sub-cents at all days. * Compute the account balance using the above strategy, i.e. ignoring sub-cents at the end of each day.

```
# Exercise 7
```

6 Numerical mathematics

Download notebook.

We will be using `numpy` and some `scipy`. These packages provide powerful and fast computations for numerical mathematics. (Most of it is implemented in `C`.) We will only learn about some parts of it, some linear algebra (using `numpy`), optimization (using `scipy`) and statistics (using `scipy`). Together with `pandas` (see Section 7.1) for data handling, this is the basic framework for data analysis in Python.

6.1 Arrays, vectors, matrices (`np.array`)

Let us start by importing `numpy`.

```
# uncomment the following line for installing numpy
# !pip3 install numpy
import numpy as np
```

(Then, the functions from `numpy` are called `np.something`.)

A vector or matrix is an `np.array`. `numpy` interprets a vector as a column vector. Here, `x` is an `np.array` of type `float64`:

- `x.ndim`: number of dimensions of `x`.
- `x.shape`: tuple of length `x.ndim` which gives the length of dimension `i` as entry `i`. (Recall that e.g. `(3,)` is a tuple of length `1`.)
- `x.dtype`: type of entries in `x`.
- `x.reshape(m, n)`: returns a view of `x` with shape `(m, n)`. The total number of entries must match. Using `-1` for one dimension lets NumPy infer it.
- `x.T`: the transpose of `x`. For a 1d array, this has no effect.
- `x.copy()`: returns an independent copy of `x`.
- `x.flatten()`: returns a 1d copy of `x`.

```
x = np.array([1.0, 2.0, 3.0])
A = np.array([
    [1.0, 0.0, 2.0],
    [0.0, 1.0, 1.0]
])
print("x =", x)
print("A =")
print(A)
```

```
x = [1. 2. 3.]
A =
[[1. 0. 2.]
 [0. 1. 1.]]
```

```
print("x.shape =", x.shape)
print("A.shape =", A.shape)
print("x.ndim =", x.ndim)
print("A.ndim =", A.ndim)
print("x.dtype =", x.dtype)
```

```
x.shape = (3,)
A.shape = (2, 3)
x.ndim = 1
A.ndim = 2
x.dtype = float64
```

```
# reshape and transpose return views, not copies
print("A.reshape(3,2) =")
print(A.reshape(3,2))
print("A.reshape(-1) =", A.reshape(-1))
print("A.T =")
print(A.T)
```

```
A.reshape(3,2) =
[[1. 0.]
 [2. 0.]
 [1. 1.]]
A.reshape(-1) = [1. 0. 2. 0. 1. 1.]
A.T =
[[1. 0.]
```

```
 [0. 1.]
 [2. 1.]]
```

```
# column vector vs row vector vs 1d array
x_col = x.reshape(-1, 1)
x_row = x.reshape(1, -1)
print("x.shape =", x.shape)
print("x_col.shape =", x_col.shape)
print("x_row.shape =", x_row.shape)
```

```
x.shape = (3,)
x_col.shape = (3, 1)
x_row.shape = (1, 3)
```

An important observation is that the shape of a vector in NumPy is `(3,)`, not `(3, 1)`. This is mathematically convenient in some places and dangerous in others. Much of good NumPy programming consists in keeping track of such distinctions.

6.2 Creating arrays (`np.zeros`, `np.ones`, `np.arange`, `np.linspace`, `np.eye`)

There are several convenient ways to create arrays:

- `np.zeros(n)`: array of `n` zeros.
- `np.ones(n)`: array of `n` ones.
- `np.arange(n)`: array `[0, 1, ..., n-1]`. Also `np.arange(start, stop, step)`.
- `np.linspace(a, b, n)`: `n` equally spaced points from `a` to `b`.
- `np.eye(n)`: the $n \times n$ identity matrix.
- `np.empty(n)`: uninitialized array of length `n` (fast, but entries are garbage).
- `np.column_stack([a, b, ...])`: stack 1d arrays as columns of a 2d array.
- `np.vstack([A, B])` / `np.hstack([A, B])`: stack arrays vertically / horizontally.

```
print("np.zeros(4) =", np.zeros(4))
print("np.ones(3) =", np.ones(3))
print("np.arange(10) =", np.arange(10))
print("np.linspace(0, 1, 5) =", np.linspace(0, 1, 5))
```

```
np.zeros(4) = [0. 0. 0. 0.]
np.ones(3) = [1. 1. 1.]
np.arange(10) = [0 1 2 3 4 5 6 7 8 9]
np.linspace(0, 1, 5) = [0.   0.25 0.5  0.75 1.  ]
```

```
print("np.eye(3) =")
print(np.eye(3))
```

```
np.eye(3) =
[[1. 0. 0.]
 [0. 1. 0.]
 [0. 0. 1.]]
```

```
# stacking arrays
a = np.array([1.0, 2.0, 3.0])
b = np.array([10.0, 20.0, 30.0])
print("np.column_stack([a, b]) =")
print(np.column_stack([a, b]))
```

```
np.column_stack([a, b]) =
[[ 1. 10.]
 [ 2. 20.]
 [ 3. 30.]]
```

6.3 Basic arithmetic

Arithmetic operations on arrays are **elementwise**. This is different from matrix multiplication.

- `x + y`, `x - y`: elementwise addition / subtraction.
- `x * y`: elementwise multiplication (not matrix multiplication!).
- `x / y`: elementwise division.
- `x ** n`: elementwise power.
- `np.abs(x)`: elementwise absolute value. Also `np.sqrt(x)`, `np.exp(x)`, `np.log(x)`, `np.sin(x)`, etc.

```
x = np.array([1.0, -2.0, 4.0])
y = np.array([3.0, 5.0, -1.0])
print("x + y =", x + y)
print("x * y =", x * y)
print("2 * x =", 2 * x)
print("x ** 2 =", x ** 2)
```

```
x + y = [4. 3. 3.]
x * y = [  3. -10.  -4.]
2 * x = [ 2. -4.  8.]
x ** 2 = [ 1.  4. 16.]
```

6.4 Indexing, slicing, views, copies

Indexing and slicing work similarly to lists, but NumPy arrays support multi-dimensional indexing. An important difference: slicing an array creates a **view**, not a copy. Modifying the slice modifies the original.

- `a[i]`: element at index `i`.
- `a[i:j]`: slice from `i` to `j-1`.
- `a[::k]`: every `k`-th element.
- `A[i, j]`: element at row `i`, column `j`.
- `A[:, j]`: column `j`.
- `A[i, :]`: row `i`.
- `a[mask]`: elements where boolean array `mask` is `True`.

```
a = np.arange(10)
print("a =", a)
print("a[2:7] =", a[2:7])
print("a[::2] =", a[::2])
```

```
a = [0 1 2 3 4 5 6 7 8 9]
a[2:7] = [2 3 4 5 6]
a[::2] = [0 2 4 6 8]
```

```
A = np.arange(1, 13).reshape(3, 4)
print("A =")
print(A)
print("A[:, 1] =", A[:, 1])
print("A[1, :] =", A[1, :])
print("A[:2, 1:3] =")
print(A[:2, 1:3])
```

```
A =
[[ 1  2  3  4]
 [ 5  6  7  8]
 [ 9 10 11 12]]
```

```
A[:, 1] = [ 2  6 10]
A[1, :] = [5 6 7 8]
A[:2, 1:3] =
[[2 3]
 [6 7]]
```

```python
# views vs copies: modifying a slice modifies the original!
a = np.arange(10)
b = a[2:6]
b[0] = 99
print("a =", a)
```

```
a = [ 0  1 99  3  4  5  6  7  8  9]
```

```python
# to avoid this, use .copy()
a = np.arange(10)
c = a[2:6].copy()
c[0] = -7
print("a =", a)
print("c =", c)
```

```
a = [0 1 2 3 4 5 6 7 8 9]
c = [-7  3  4  5]
```

```python
# boolean masks
x = np.array([3.0, -1.0, 2.5, -4.0, 0.0, 7.0])
mask = x > 0
print("x[mask] =", x[mask])
```

```
x[mask] = [3.  2.5 7. ]
```

6.5 Vectorization and broadcasting

In numerical computing, we want to avoid Python loops whenever possible and instead work with full arrays at once. This is called **vectorization**. It is both faster and more readable.

```python
x = np.linspace(0.0, 1.0, 6)
# vectorized: looks like the mathematical formula
y = 3 * x + 2
print("y =", y)
```

```
y = [2.  2.6 3.2 3.8 4.4 5. ]
```

Broadcasting allows NumPy to combine arrays of compatible but different shapes. The smaller array is "broadcast" across the larger one.

```
X = np.array([
    [1.0, 2.0],
    [3.0, 4.0],
    [5.0, 6.0]
])
b = np.array([10.0, 20.0])
# b is broadcast across all rows of X
print("X + b =")
print(X + b)
```

```
X + b =
[[11. 22.]
 [13. 24.]
 [15. 26.]]
```

```
# pairwise operations via broadcasting
x = np.array([0.0, 1.5, 3.0, 5.0])
D = np.abs(x[:, None] - x[None, :])
print("distance matrix =")
print(D)
```

```
distance matrix =
[[0.  1.5 3.  5. ]
 [1.5 0.  1.5 3.5]
 [3.  1.5 0.  2. ]
 [5.  3.5 2.  0. ]]
```

The expression `x[:, None]` turns `x` into a column vector (shape `(4,1)`), and `x[None, :]` into a row vector (shape `(1,4)`). Broadcasting then produces the full 4×4 distance matrix without any loop.

6.6 Linear algebra (`np.linalg`)

NumPy provides a comprehensive linear algebra module. Here, `x` and `y` are vectors, `A` and `B` are matrices.

- `A @ B` or `np.dot(A, B)`: matrix multiplication. For two vectors, `x @ y` computes the inner product $\langle x, y \rangle = \sum_j x_j y_j$. For a matrix and a vector, `A @ x` is the matrix-vector product.
- `np.linalg.norm(x)`: Euclidean norm $\|x\|_2$.
- `np.linalg.solve(A, b)`: solve $Ax = b$. Preferred over computing $A^{-1}b$.
- `np.linalg.inv(A)`: matrix inverse. Usually `solve` is preferred.
- `np.linalg.det(A)`: determinant.
- `np.linalg.cond(A)`: condition number. Large values indicate numerical instability.
- `np.linalg.eig(A)`: eigenvalues and eigenvectors.

```
x = np.array([1.0, -2.0, 2.0])
y = np.array([3.0, 1.0, -1.0])
print("<x,y> =", x @ y)
print("||x||_2 =", np.linalg.norm(x))
# np.abs(), np.sum(), np.max() are elementwise/aggregation functions
# described in detail in the aggregation section below
print("||x||_1 =", np.sum(np.abs(x)))
print("||x||_inf =", np.max(np.abs(x)))
```

```
<x,y> = -1.0
||x||_2 = 3.0
||x||_1 = 5.0
||x||_inf = 2.0
```

```
A = np.array([[1.0, 2.0], [3.0, 4.0]])
x = np.array([1.0, -1.0])
print("A @ x =", A @ x)
```

```
A @ x = [-1. -1.]
```

```
# solving Ax = b
A = np.array([[3.0, 1.0], [1.0, 2.0]])
b = np.array([9.0, 8.0])
x = np.linalg.solve(A, b)
print("solution x =", x)
print("A @ x =", A @ x)
```

```
solution x = [2. 3.]
A @ x = [9. 8.]
```

```
# eigenvalues
A = np.array([[2.0, 1.0], [1.0, 2.0]])
eigenvalues, eigenvectors = np.linalg.eig(A)
print("eigenvalues =", eigenvalues)
```

```
eigenvalues = [3. 1.]
```

One often sees formulas like $x = A^{-1}b$, but numerically it is usually better to solve the system directly using `np.linalg.solve` rather than computing the inverse explicitly.

6.7 Aggregation and the `axis` parameter

NumPy provides functions for aggregating arrays. When applied to 2d arrays, the `axis` parameter controls the direction.

- `np.sum(x)`, `np.mean(x)`, `np.median(x)`, `np.var(x)`, `np.std(x)`: sum, mean, median, variance, standard deviation.
- `np.min(x)`, `np.max(x)`: minimum, maximum.
- `np.cumsum(x)`: cumulative sum.
- `axis=0`: aggregate down the rows (result: one entry per column).
- `axis=1`: aggregate across the columns (result: one entry per row).

```
data = np.array([
    [1.0, 2.0, 3.0],
    [4.0, 5.0, 6.0],
    [7.0, 8.0, 9.0]
])
print("sum over all =", np.sum(data))
print("column means =", np.mean(data, axis=0))
print("row means =", np.mean(data, axis=1))
```

```
sum over all = 45.0
column means = [4. 5. 6.]
row means = [2. 5. 8.]
```

```
# standardization: subtract column mean, divide by column std
X = np.array([[1.0, 10.0], [2.0, 20.0], [3.0, 30.0], [4.0, 40.0]])
mu = np.mean(X, axis=0)
sigma = np.std(X, axis=0)
```

```
X_std = (X - mu) / sigma
print("column means after standardization =", np.mean(X_std, axis=0))
print("column stds after standardization =", np.std(X_std, axis=0))
```

```
column means after standardization = [0. 0.]
column stds after standardization = [1. 1.]
```

Standardization is a common preprocessing step. It ensures that all features are on comparable scales, which is important e.g. for gradient-based optimization.

6.8 Random numbers (`np.random`)

Randomness enters data analysis in many places: synthetic data generation, random initialization of parameters, train-test splitting, and randomized algorithms. The recommended interface is `np.random.default_rng`, which creates a random number generator with a given seed.

- `rng = np.random.default_rng(seed)`: create a random number generator.
- `rng.random(n)`: `n` uniform random numbers in $[0, 1)$.
- `rng.standard_normal(n)`: `n` standard normal random numbers.
- `rng.normal(loc, scale, size)`: normal with given mean and standard deviation.
- `rng.integers(low, high, size)`: random integers in `[low, high)`.
- `rng.choice(a, size, replace)`: random sample from array `a`.
- `rng.permutation(n)`: a random permutation of `0, ..., n-1`.

```
rng = np.random.default_rng(42)
print("uniform =", rng.random(5))
print("normal =", rng.standard_normal(5))
print("integers =", rng.integers(0, 10, size=5))
```

```
uniform = [0.77395605 0.43887844 0.85859792 0.69736803 0.09417735]
normal = [-1.30217951  0.1278404  -0.31624259 -0.01680116 -0.85304393]
integers = [5 3 1 9 7]
```

```
# reproducibility: same seed gives same numbers
rng1 = np.random.default_rng(123)
rng2 = np.random.default_rng(123)
print(rng1.standard_normal(4))
print(rng2.standard_normal(4))
```

```
[-0.98912135 -0.36778665  1.28792526  0.19397442]
[-0.98912135 -0.36778665  1.28792526  0.19397442]
```

```
# rng.choice: draw random samples from an array
rng = np.random.default_rng(42)
names = np.array(["Alice", "Bob", "Charlie", "Diana", "Eve"])
print("pick 3 with replacement =", rng.choice(names, size=3, replace=True))
print("pick 3 without replacement =", rng.choice(names, size=3, replace=False))
```

```
pick 3 with replacement = ['Alice' 'Diana' 'Diana']
pick 3 without replacement = ['Eve' 'Diana' 'Bob']
```

```
# rng.permutation: shuffle an array (returns a new array)
rng = np.random.default_rng(7)
cards = np.array(["7", "8", "9", "10", "J", "Q", "K", "A"])
shuffled = rng.permutation(cards)
print("original =", cards)
print("shuffled =", shuffled)
# permutation of integers
print("random order of 0..4 =", rng.permutation(5))
```

```
original = ['7' '8' '9' '10' 'J' 'Q' 'K' 'A']
shuffled = ['7' 'K' 'A' '9' 'J' 'Q' '8' '10']
random order of 0..4 = [4 2 3 0 1]
```

6.9 Optimization (`scipy.optimize`)

The `scipy.optimize` module provides tools for finding minima of functions and roots of equations. This is essential in many areas of mathematics and data analysis.

- `optimize.minimize_scalar(f)`: find the minimum of a scalar function $f : \mathbb{R} \to \mathbb{R}$. Returns a result object with `.x` (minimizer), `.fun` (minimum value), `.nfev` (number of function evaluations).
- `optimize.minimize(f, x0)`: find the minimum of $f : \mathbb{R}^d \to \mathbb{R}$, starting from `x0`. Returns a result object with attributes `.x` (minimizer), `.fun` (minimum value), `.success`, `.nit` (number of iterations).
- `optimize.minimize(f, x0, jac=grad_f)`: same, but with analytically supplied gradient for faster convergence.
- `optimize.minimize(f, x0, method='...')`: choose a specific algorithm, e.g. `'BFGS'`, `'Nelder-Mead'`, `'L-BFGS-B'`.

- `optimize.root_scalar(f, bracket=[a, b])`: find a root of f in the interval $[a, b]$. Returns a result object with `.root` (the root location).

```
from scipy import optimize
```

```
# minimizing a function of one variable
def f(x):
    return (x - 3)**2 + 1

result = optimize.minimize_scalar(f)
print("minimum at x =", result.x)
print("minimum value =", result.fun)
print("function evaluations =", result.nfev)
```

```
minimum at x = 3.0
minimum value = 1.0
function evaluations = 9
```

```
# minimizing a function of several variables (Rosenbrock function)
def rosenbrock(x):
    return (1 - x[0])**2 + 100 * (x[1] - x[0]**2)**2

x0 = np.array([0.0, 0.0])
result = optimize.minimize(rosenbrock, x0)
print("minimum at x =", result.x)
print("success =", result.success)
```

```
minimum at x = [0.99999467 0.99998932]
success = True
```

```
# providing the gradient for faster convergence
def rosenbrock_grad(x):
    dfdx0 = -2 * (1 - x[0]) - 400 * x[0] * (x[1] - x[0]**2)
    dfdx1 = 200 * (x[1] - x[0]**2)
    return np.array([dfdx0, dfdx1])

result = optimize.minimize(rosenbrock, x0, jac=rosenbrock_grad, method='BFGS')
print("minimum at x =", result.x)
print("iterations =", result.nit)
```

```
minimum at x = [0.99999913 0.99999825]
iterations = 19
```

```python
# root finding
def g(x):
    return x**3 - 2*x - 5

result = optimize.root_scalar(g, bracket=[2, 3])
print("root at x =", result.root)
print("g(root) =", g(result.root))
```

```
root at x = 2.094551481542327
g(root) = 3.552713678800501e-15
```

6.10 Exercises

Exercise 1 Find out about the singular value decomposition in `np.linalg.svd`. Compute the SVD of the matrix $A = \begin{pmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{pmatrix}$ and verify that $A = U\Sigma V^\top$ by reconstructing A from the three factors.

```python
# Exercise 1
```

Exercise 2 Why is it better to use `np.linalg.solve` rather than `np.linalg.inv`? Demonstrate with a concrete example: create a 10×10 random matrix A and a vector b. Solve $Ax = b$ both ways and compare the residuals $\|Ax - b\|$. Then try with a nearly singular matrix (e.g., add a row that is almost a copy of another) and compare again.

```python
# Exercise 2
```

Exercise 3 Write a function `power_method(A, num_iter=100)` that approximates the largest eigenvalue of a matrix A using the power iteration: start with a random vector v, repeatedly compute $v \leftarrow Av/\|Av\|$, and return $v^\top Av$. Compare the result with `np.linalg.eig`.

```python
# Exercise 3
```

Exercise 4 Create a 1d array of 20 equally spaced values between 0 and 2π using `np.linspace`. Compute $\sin(x)$ and $\cos(x)$ for these values (vectorized, no loop). Then use boolean masking to select only those entries where $\sin(x) > 0.5$.

```
# Exercise 4
```

Exercise 5 Use `np.random.default_rng(seed=0)` to generate a 100×4 matrix of standard normal random numbers. Compute the mean and standard deviation of each column using the `axis` parameter. Then standardize the matrix (subtract column means, divide by column standard deviations) and verify that the result has column means ≈ 0 and column standard deviations ≈ 1.

```
# Exercise 5
```

Exercise 6 Broadcasting exercise: create a column vector $a = (1, 2, 3, 4)^\top$ and a row vector $b = (10, 20, 30)$. Compute their outer product using broadcasting (i.e., `a * b` with appropriate shapes). Verify the result by comparing with `np.outer(a, b)`.

```
# Exercise 6
```

Exercise 7 Given the array `temps = np.array([15.2, 18.5, 22.1, 25.3, 19.8])` of temperatures in Celsius, convert all values to Fahrenheit using the formula $F = C \cdot 9/5 + 32$ — without using a loop. Then find all temperatures above $70°F$ using boolean masking.

```
# Exercise 7
```

Exercise 8 Create two arrays: `prices = np.array([10.0, 20.0, 30.0])` and `quantities = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])`, where each row represents a customer's order. Compute the total cost per customer using broadcasting and `np.sum` with the `axis` parameter.

```
# Exercise 8
```

Exercise 9 Create a 5×5 matrix where entry (i, j) contains the value $|i - j|$ (the distance between row and column index). Do this without loops, using broadcasting with `np.arange` and `np.abs`.

```
# Exercise 9
```

7 Loading and handling data

Download notebook.

In this chapter we learn how to load data from various sources and how to handle tabular data using `pandas`. Before we start, we describe what a dataframe is, since it is the building block of many data science routines.

7.1 Dataframes (`pandas`)

`pandas` is the standard library for handling tabular data in Python. Its central data structure is the **`DataFrame`**, which is a two-dimensional, tabular data structure — essentially a table with rows and columns. Each column has a name and a data type (e.g., integers, floats, strings), and each row represents one observation or record. You can think of a dataframe as an in-memory spreadsheet that comes with powerful methods for selecting, filtering, transforming, and aggregating data. A single column of a DataFrame is a **`Series`** — a one-dimensional array with an index. A `Series` behaves like a list but supports vectorized operations (e.g., `series + 1` adds 1 to every element) and has many built-in methods such as `.mean()`, `.sum()`, or `.value_counts()`.

```
# Execute once for installing pandas
# !pip3 install pandas
import pandas as pd
import numpy as np
```

Here are the most important routines, where `df` is a `DataFrame`:

- `pd.DataFrame(dict)`: create a DataFrame from a dictionary (keys become column names, values must be lists of the same length).
- `df["col"]`: select a single column (returns a `Series`, which is similar to a `list`).
- `df[["col1", "col2"]]`: select multiple columns.
- `df.iloc[i]` or `df.iloc[i:j]`: select row(s) `i: int` (up to `j-1`).
- `df.loc[i, "col"]` or `df.loc[i:j, ["col1", "col2"]]`: select row(s) `i` (up to `j-1`) for columns `"col"` (resp `"col1", "col"`).
- `df[df["col"] > value]`: filter rows by a condition (here `df["col"] > value`).
- `df.head(n)` / `df.tail(n)`: first / last `n` rows.

- `df.describe()`: summary statistics (count, mean, std, min, quartiles, max) for all numeric columns.
- `df["col"].count()`, `df["col"].mean()`, `df["col"].std()`, `df["col"].min()`, `df["col"].max()`: individual summary statistics for a column.
- `df["col"].median()`: median of a column.
- `df["col"].quantile(q)`: the q-th quantile (e.g., `q=0.25` for the first quartile).
- `df["col"].sum()`: sum of all values in a column.
- `df.sort_values("col")`: sort by a column.
- `df.groupby("col").mean()`: group by a column and compute means.
- `df.isna()`: boolean mask of missing values.
- `df.dropna()`: drop rows with missing values.
- `df.fillna(value)`: fill missing values.
- `df.drop(columns=["col"])`: remove a column.
- `df.to_numpy()`: convert to a NumPy array (but column names get lost).

```
df = pd.DataFrame({
    "age": [22, 27, 31, 45, 38, 29],
    "income": [42000, 51000, 64000, 83000, 72000, 56000],
    "hours_studied": [10, 12, 7, 4, 6, 9],
    "passed": [1, 1, 0, 0, 1, 1]
})
df
```

	age	income	hours_studied	passed
0	22	42000	10	1
1	27	51000	12	1
2	31	64000	7	0
3	45	83000	4	0
4	38	72000	6	1
5	29	56000	9	1

```
# selecting columns
print(df["age"])
```

```
0    22
1    27
2    31
3    45
4    38
5    29
Name: age, dtype: int64
```

```
print(df[["age", "income"]])
```

```
   age  income
0   22   42000
1   27   51000
2   31   64000
3   45   83000
4   38   72000
5   29   56000
```

```
# selecting rows
print(df.iloc[0])
```

```
age                22
income          42000
hours_studied      10
passed              1
Name: 0, dtype: int64
```

```
print(df.loc[0:2, ["age", "passed"]])
```

```
   age  passed
0   22       1
1   27       1
2   31       0
```

```
# filtering
filtered = df[df["income"] > 55000]
filtered
```

	age	income	hours_studied	passed
2	31	64000	7	0
3	45	83000	4	0
4	38	72000	6	1
5	29	56000	9	1

```python
# summary statistics
print(df.describe())
```

```
             age        income  hours_studied    passed
count   6.000000      6.000000       6.000000  6.000000
mean   32.000000  61333.333333       8.000000  0.666667
std     8.246211  14827.901627       2.898275  0.516398
min    22.000000  42000.000000       4.000000  0.000000
25%    27.500000  52250.000000       6.250000  0.250000
50%    30.000000  60000.000000       8.000000  1.000000
75%    36.250000  70000.000000       9.750000  1.000000
max    45.000000  83000.000000      12.000000  1.000000
```

```python
# individual statistics for a column
print("mean income:", df["income"].mean())
print("std income:", df["income"].std())
print("median income:", df["income"].median())
print("25th percentile:", df["income"].quantile(0.25))
print("total hours studied:", df["hours_studied"].sum())
```

```
mean income: 61333.333333333336
std income: 14827.901627225163
median income: 60000.0
25th percentile: 52250.0
total hours studied: 48
```

```python
# sorting
df_sorted = df.sort_values("income", ascending=False)
print(df_sorted)
```

```
   age  income  hours_studied  passed
3   45   83000              4       0
4   38   72000              6       1
2   31   64000              7       0
5   29   56000              9       1
1   27   51000             12       1
0   22   42000             10       1
```

```python
# groupby - numeric_only=True restricts the computation to numeric columns,
# ignoring string columns (which cannot be averaged)
grouped = df.groupby("passed").mean(numeric_only=True)
print(grouped)
```

```
          age   income  hours_studied
passed
0       38.0  73500.0           5.50
1       29.0  55250.0           9.25
```

```python
# missing values
df_missing = df.copy()
df_missing.loc[2, "income"] = np.nan
df_missing.loc[4, "hours_studied"] = np.nan

print(df_missing)
print()
print(df_missing.isna().sum())
print()
print(df_missing.dropna())
```

```
   age   income  hours_studied  passed
0   22  42000.0           10.0       1
1   27  51000.0           12.0       1
2   31      NaN            7.0       0
3   45  83000.0            4.0       0
4   38  72000.0            NaN       1
5   29  56000.0            9.0       1

age              0
income           1
hours_studied    1
passed           0
dtype: int64

   age   income  hours_studied  passed
0   22  42000.0           10.0       1
1   27  51000.0           12.0       1
3   45  83000.0            4.0       0
5   29  56000.0            9.0       1
```

```
# fill missing values with column means
df_filled = df_missing.fillna(df_missing.mean(numeric_only=True))
print(df_filled)
```

```
   age   income  hours_studied  passed
0   22  42000.0           10.0       1
1   27  51000.0           12.0       1
2   31  60800.0            7.0       0
3   45  83000.0            4.0       0
4   38  72000.0            8.4       1
5   29  56000.0            9.0       1
```

```
# from pandas to numpy
clean = df_missing.dropna()
X = clean[["age", "income", "hours_studied"]].to_numpy(dtype=float)
y = clean["passed"].to_numpy(dtype=float)
print("X.shape =", X.shape)
print("y =", y)
```

```
X.shape = (4, 3)
y = [1. 1. 0. 1.]
```

7.2 Adding and renaming columns (`assign`, `rename`)

Adding new columns and renaming existing ones are among the most frequent pandas operations.

- `df["new_col"] = values`: add a new column (or overwrite an existing one) by direct assignment.
- `df.assign(col=expr)`: return a new DataFrame with an additional column (does not modify the original).
- `df.rename(columns={"old": "new"})`: rename one or more columns.
- `df.insert(loc, column, value)`: insert a column at a specific position.

```
df = pd.DataFrame({
    "name": ["Alice", "Bob", "Charlie"],
    "weight_kg": [62, 85, 74],
    "height_m": [1.65, 1.80, 1.75]
})
df
```

	name	weight_kg	height_m
0	Alice	62	1.65
1	Bob	85	1.80
2	Charlie	74	1.75

```
# adding a computed column
df["bmi"] = df["weight_kg"] / df["height_m"] ** 2
df
```

	name	weight_kg	height_m	bmi
0	Alice	62	1.65	22.773186
1	Bob	85	1.80	26.234568
2	Charlie	74	1.75	24.163265

```
# assign returns a new DataFrame, leaving df unchanged
df2 = df.assign(height_cm=df["height_m"] * 100)
df2
```

	name	weight_kg	height_m	bmi	height_cm
0	Alice	62	1.65	22.773186	165.0
1	Bob	85	1.80	26.234568	180.0
2	Charlie	74	1.75	24.163265	175.0

```
# renaming columns
df_renamed = df.rename(columns={"weight_kg": "weight", "height_m": "height"})
df_renamed
```

	name	weight	height	bmi
0	Alice	62	1.65	22.773186
1	Bob	85	1.80	26.234568
2	Charlie	74	1.75	24.163265

Note that direct assignment (`df["col"] = ...`) modifies the DataFrame in place, while `assign` and `rename` return new DataFrames by default.

7.3 Merging and joining DataFrames (`merge`, `concat`)

Assume you have two DataFrames, which share one column (best used with unique entries, called `key` below). Then, `pd.merge` joins the two DataFrames into one by using the joint column as anchor. However, note that we have the keys from the left and right DataFrame, and there are keys appearing in the left, in the right, in both, or in any of the two DataFrames. This leads to the four cases, how `pd.merge(left, right, on="key", how="left/right/inner/outer")` operates:

how	meaning
`"left"`	keep all rows from `left`
`"right"`	keep all rows from `right`
`"inner"`	keep only joint keys
`"outer"`	keep all keys from both DataFrames

In contrast, in `pd.concat`, you have two DataFrames which do not need to share a column, and you simply stack them vertically or horizontally.

- `pd.merge(left, right, on="key")`: inner join on a shared column `key`.
- `pd.merge(left, right, on="key", how="left")`: left join (keep all rows from the left DataFrame). Other options: `"right"`, `"inner"`, `"outer"`.
- `pd.concat([df1, df2])`: stack DataFrames vertically (row-wise).
- `pd.concat([df1, df2], axis=1)`: stack DataFrames horizontally (column-wise).

```
students = pd.DataFrame({
    "student_id": [1, 2, 3, 4],
    "name": ["Alice", "Bob", "Charlie", "Diana"]
})

grades = pd.DataFrame({
    "student_id": [1, 2, 3, 5],
    "grade": [1.3, 2.0, 1.7, 2.7]
})

print(students)
print()
print(grades)
```

```
   student_id     name
0           1    Alice
1           2      Bob
```

```
2           3  Charlie
3           4    Diana

   student_id  grade
0           1    1.3
1           2    2.0
2           3    1.7
3           5    2.7
```

```python
# inner join: only students present in both tables
merged = pd.merge(students, grades, on="student_id")
# same as merged = pd.merge(students, grades, on="student_id", how="inner")
print(merged)
```

```
   student_id     name  grade
0           1    Alice    1.3
1           2      Bob    2.0
2           3  Charlie    1.7
```

```python
# left join: keep all students, NaN where no grade exists
merged_left = pd.merge(students, grades, on="student_id", how="left")
print(merged_left)
```

```
   student_id     name  grade
0           1    Alice    1.3
1           2      Bob    2.0
2           3  Charlie    1.7
3           4    Diana    NaN
```

```python
# concatenating vertically
df_a = pd.DataFrame({"name": ["Alice", "Bob"], "score": [85, 92]})
df_b = pd.DataFrame({"name": ["Charlie", "Diana"], "score": [78, 91]})
combined = pd.concat([df_a, df_b], ignore_index=True)
print(combined)
```

```
      name  score
0    Alice     85
1      Bob     92
2  Charlie     78
3    Diana     91
```

When concatenating vertically, use `ignore_index=True` to reset the index; otherwise both DataFrames keep their original indices, which may lead to duplicate index values.

7.4 Applying custom functions (`apply`, `map`)

Built-in methods like `.mean()` or `.sum()` cover many cases, but sometimes you need to apply an arbitrary Python function to each row, column, or element.

- `df.apply(func)`: apply `func` to each column (default) or each row (`axis=1`).
- `series.map(func)`: apply `func` element-wise to a Series. (This is usually applied to a column of a DataFrame.)
- `df.map(func)`: apply `func` element-wise to every cell of a DataFrame.

```
df = pd.DataFrame({
    "name": ["Alice", "Bob", "Charlie"],
    "score": [85, 92, 78],
    "bonus": [5, 3, 8]
})
df
```

	name	score	bonus
0	Alice	85	5
1	Bob	92	3
2	Charlie	78	8

```
# apply a function to each column (default axis=0)
print(df[["score", "bonus"]].apply(np.mean))
```

```
score    85.000000
bonus     5.333333
dtype: float64
```

```
# apply a function row-wise (axis=1)
df["total"] = df.apply(lambda row: row["score"] + row["bonus"], axis=1)
df
```

	name	score	bonus	total
0	Alice	85	5	90
1	Bob	92	3	95
2	Charlie	78	8	86

```
# map: element-wise transformation of a Series
df["grade"] = df["total"].map(lambda x: "pass" if x >= 85 else "fail")
df
```

	name	score	bonus	total	grade
0	Alice	85	5	90	pass
1	Bob	92	3	95	pass
2	Charlie	78	8	86	pass

Use `apply` with `axis=1` when the computation depends on multiple columns in the same row. For simple element-wise transformations on a single column, `map` is more readable. For best performance on large DataFrames, prefer vectorized operations (e.g., `df["score"] + df["bonus"]`) over `apply` whenever possible.

7.5 Iterating through a DataFrame (`iterrows`, `itertuples`)

Sometimes you need to loop over the rows of a DataFrame, for example to perform row-wise logic that is hard to express as a vectorized operation.

- `df.iterrows()`: iterate over rows as `(index, Series)` pairs.
- `df.itertuples()`: iterate over rows as named tuples (faster than `iterrows`).

```
df = pd.DataFrame({
    "name": ["Alice", "Bob", "Charlie"],
    "score": [85, 92, 78],
    "bonus": [5, 3, 8]
})

# iterrows: each row is a Series
for idx, row in df.iterrows():
    print(f"Row {idx}: {row['name']} scored {row['score'] + row['bonus']} in total")
```

```
Row 0: Alice scored 90 in total
Row 1: Bob scored 95 in total
Row 2: Charlie scored 86 in total
```

```python
# itertuples: each row is a named tuple (faster)
for row in df.itertuples():
    print(f"{row.name}: score={row.score}, bonus={row.bonus}")
```

```
Alice: score=85, bonus=5
Bob: score=92, bonus=3
Charlie: score=78, bonus=8
```

Prefer `itertuples` over `iterrows` when performance matters, since it avoids creating a `Series` for each row. However, for most tasks, vectorized operations (e.g., `df["score"] + df["bonus"]`) or `apply` are faster than explicit loops and should be preferred whenever possible.

7.6 Value counts and unique values (`value_counts`, `unique`)

When exploring a dataset, you often want to know which values occur in a column and how frequently. This is especially useful for categorical data.

- `df["col"].value_counts()`: count how often each value appears, sorted by frequency. (`df["col"].value_counts(normalize=True)` returns relative frequencies instead of absolute counts instead.)
- `df["col"].unique()`: return an array of all distinct values.
- `df["col"].nunique()`: return the number of distinct values.

```python
df = pd.DataFrame({
    "name": ["Alice", "Bob", "Charlie", "Diana", "Eve", "Frank"],
    "major": ["Math", "CS", "Math", "CS", "Physics", "Math"],
    "level": ["BSc", "MSc", "MSc", "BSc", "BSc", "MSc"]
})
df
```

	name	major	level
0	Alice	Math	BSc
1	Bob	CS	MSc
2	Charlie	Math	MSc

	name	major	level
3	Diana	CS	BSc
4	Eve	Physics	BSc
5	Frank	Math	MSc

```
# how often does each major appear?
print(df["major"].value_counts())
```

```
major
Math       3
CS         2
Physics    1
Name: count, dtype: int64
```

```
# relative frequencies
print(df["major"].value_counts(normalize=True))
```

```
major
Math       0.500000
CS         0.333333
Physics    0.166667
Name: proportion, dtype: float64
```

```
# unique values and their count
print("Unique majors:", df["major"].unique())
print("Number of majors:", df["major"].nunique())
```

```
Unique majors: <StringArray>
['Math', 'CS', 'Physics']
Length: 3, dtype: str
Number of majors: 3
```

`value_counts()` is one of the first things to try when getting to know a new dataset. For numeric columns, consider using `df["col"].describe()` instead.

7.7 Handling duplicates (`duplicated`, `drop_duplicates`)

Real-world data often contains duplicate rows — for example from repeated data imports or logging errors. `pandas` makes it easy to detect and remove them.

- `df.duplicated()`: return a boolean Series that is `True` for duplicate rows.
- `df.duplicated(subset=["col1", "col2"])`: check for duplicates based on specific columns only.
- `df.drop_duplicates()`: remove duplicate rows (keeps the first occurrence by default).
- `df.drop_duplicates(subset=["col"], keep="last")`: keep the last occurrence instead.

```
df = pd.DataFrame({
    "name": ["Alice", "Bob", "Alice", "Charlie", "Bob"],
    "score": [85, 92, 85, 78, 95]
})
df
```

	name	score
0	Alice	85
1	Bob	92
2	Alice	85
3	Charlie	78
4	Bob	95

```
# which rows are duplicates?
print(df.duplicated())
```

```
0    False
1    False
2     True
3    False
4    False
dtype: bool
```

```
# remove exact duplicate rows
print(df.drop_duplicates())
```

```
      name  score
0    Alice     85
1      Bob     92
3  Charlie     78
4      Bob     95
```

```
# duplicates based on a subset of columns
print(df.duplicated(subset=["name"]))
```

```
0    False
1    False
2     True
3    False
4     True
dtype: bool
```

```
# keep only the last entry per name
print(df.drop_duplicates(subset=["name"], keep="last"))
```

```
      name  score
2    Alice     85
3  Charlie     78
4      Bob     95
```

Note that `drop_duplicates()` returns a new DataFrame by default. To modify in place, pass `inplace=True`. When checking duplicates on a subset of columns, the first (or last) complete row is kept, not just the key columns.

7.8 Loading data from files (`csv`, `xlsx`, `json`)

Data often comes as files. The most common formats are CSV (comma-separated values), Excel (`.xlsx`), and JSON (JavaScript Object Notation). A CSV file is like an Excel table, where contents of cells are separated by a comma (or some other delimiter). A JSON file has the same structure as a Python dictionary. We use the following example files throughout this section: person.csv, person.json, and person.db. These contain data on 50 fictional persons with names, addresses, dates of birth, and hobbies.

Download notebook.

- `pd.read_csv(path)`: reads a CSV file into a DataFrame.

- `df.to_csv(path, index=False)`: writes a DataFrame to a CSV file.
- `pd.read_json(path)`: reads a JSON file into a DataFrame. (So the JSON file is interpreted as a dict in `pd.DataFrame(dict)`.)
- `df.to_json(orient="records")`: converts a DataFrame to a JSON string. The `orient` parameter controls the format; `"records"` produces a list of dictionaries.
- `pd.read_excel(path)`: reads an Excel file into a DataFrame (requires the `openpyxl` package).

```
# Reading a CSV file
df_csv = pd.read_csv("misc/person.csv")
df_csv.head()
```

	first_name	last_name	street	town	zip	country
0	André	Pieters	Paseo Chucho Roca 1	Rey	86379	Kingdom of the
1	Ilaria	Jäggi	Veerlepad 1	Naters	5078	France
2	Melissa	Pollak	Urbanización María Teresa Mancebo 4	Lopesdan	3553	Kingdom of the
3	Vittorio	Cárdenas	Rua de Sousa, 40	Lobenstein	4395PF	Switzerland
4	Gioffre	van Dam	Eichenbergerstrasse 1	Techer	07045	France

```
# The hobbies column is a semicolon-separated string - we can inspect it
print(df_csv["hobbies"].head())
```

```
0                            reading
1          chess;writing;photography
2                             skiing
3                             baking
4    cycling;reading;hiking;swimming
Name: hobbies, dtype: str
```

```
# Reading a JSON file
df_json = pd.read_json("misc/person.json")
df_json.head()
```

	first_name	last_name	street	town	zip	country
0	André	Pieters	Paseo Chucho Roca 1	Rey	86379	Kingdom of the
1	Ilaria	Jäggi	Veerlepad 1	Naters	5078	France
2	Melissa	Pollak	Urbanización María Teresa Mancebo 4	Lopesdan	3553	Kingdom of the
3	Vittorio	Cárdenas	Rua de Sousa, 40	Lobenstein	4395PF	Switzerland

	first_name	last_name	street	town	zip	country
4	Gioffre	van Dam	Eichenbergerstrasse 1	Techer	07045	France

```
# In the JSON version, hobbies is a list
print(df_json["hobbies"].head())
```

```
0                                 [reading]
1           [chess, writing, photography]
2                                  [skiing]
3                                  [baking]
4    [cycling, reading, hiking, swimming]
Name: hobbies, dtype: object
```

```
# Converting a DataFrame back to JSON
print(df_csv.head(3).to_json(orient="records", indent=2))
```


```
[
  {
    "first_name":"Andr\u00e9",
    "last_name":"Pieters",
    "street":"Paseo Chucho Roca 1",
    "town":"Rey",
    "zip":"86379",
    "country":"Kingdom of the Netherlands",
    "date_of_birth":"1993-01-12",
    "hobbies":"reading"
  },
  {
    "first_name":"Ilaria",
    "last_name":"J\u00e4ggi",
    "street":"Veerlepad 1",
    "town":"Naters",
    "zip":"5078",
    "country":"France",
    "date_of_birth":"1950-03-12",
    "hobbies":"chess;writing;photography"
  },
  {
    "first_name":"Melissa",
    "last_name":"Pollak",
```

```
    "street":"Urbanizaci\u00f3n Mar\u00eda Teresa Mancebo 4",
    "town":"Lopesdan",
    "zip":"3553",
    "country":"Kingdom of the Netherlands",
    "date_of_birth":"1983-08-09",
    "hobbies":"skiing"
  }
]
```

Note how CSV and JSON handle the `hobbies` column differently: CSV stores it as a flat string (here semicolon-separated), while JSON preserves it as a list.

7.9 Loading data from online resources (`requests`)

The `requests` library allows downloading data from the internet.

- `requests.get(url)`: sends a GET request (i.e. tries to download a website) and returns a response object.
- `response.status_code`: HTTP status code (200 means success).
- `response.text`: the response body as a string.
- `response.json()`: parse the response body as JSON.

```
import requests

url = "https://jsonplaceholder.typicode.com/todos/1"
response = requests.get(url)
print("status code:", response.status_code)
print("content:", response.json())
```

```
status code: 200
content: {'userId': 1, 'id': 1, 'title': 'delectus aut autem', 'completed': False}
```

As a concrete example, the World Bank Indicators API provides demographic and economic data. The API offers thousands of indicators; you can search them programmatically:

```
# search for population-related indicators
url = "https://api.worldbank.org/v2/indicator?format=json&per_page=20000"
indicators = requests.get(url).json()[1]
df_ind = pd.DataFrame(indicators)
df_ind[df_ind["name"].str.contains("population", case=False)][["id", "name"]].head()
```

	id	name
24	1.1_ACCESS.ELECTRICITY.TOT	Access to electricity (% of total population)
39	1.2_ACCESS.ELECTRICITY.RURAL	Access to electricity (% of rural population)
40	1.3_ACCESS.ELECTRICITY.URBAN	Access to electricity (% of urban population)
161	2.1_ACCESS.CFT.TOT	Access to Clean Fuels and Technologies for coo...
1551	allsa.cov_pop	Coverage of social safety net programs (% of p...

Here we fetch total population data (`SP.POP.TOTL`) for all countries and display the ten most populous ones. The API returns a list with two elements: metadata (page info) at index 0 and the actual data records at index 1. Since the response also includes regional aggregates (like "Arab World" or "Euro area"), we first fetch country metadata to identify and exclude them.

```
# step 1: find out which codes are regional aggregates (not actual countries)
meta = requests.get(
    "https://api.worldbank.org/v2/country",
    params={"format": "json", "per_page": 300}
).json()[1]
aggregates = {e["iso2Code"] for e in meta if e["region"]["id"] == "NA"}

# step 2: fetch population data and keep only actual countries
response = requests.get(
    "https://api.worldbank.org/v2/country/all/indicator/SP.POP.TOTL",
    params={"date": "2023", "format": "json", "per_page": 300}
)
data = response.json()[1]  # index 0 is metadata

df_pop = pd.DataFrame([{
    "country": entry["country"]["value"],
    "population": entry["value"]
} for entry in data
    if entry["country"]["id"] not in aggregates and entry["value"] is not None])

df_pop.sort_values("population", ascending=False).head(10)
```

	country	population
89	India	1438069596
41	China	1410710000
206	United States	336806231
90	Indonesia	281190067
149	Pakistan	247504495

	country	population
144	Nigeria	227882945
26	Brazil	211140729
15	Bangladesh	171466990
161	Russian Federation	143826130
127	Mexico	129739759

7.10 Working with dates and times (`to_datetime`, `Timestamp`)

`pandas` can parse and work with dates. We use the person dataset from Section 7.8 to illustrate this.

- `pd.to_datetime(...)`: convert strings to datetime. The format codes are the same as for the `datetime` module, see Section 5.3.
- `series.dt.year` / `.dt.month` / `.dt.day_of_week`: extract date components.
- `pd.Timestamp(...)`: create a single datetime value.
- `dt.strftime(format)`: format a datetime as a string. The format codes (e.g., `%Y`, `%m`, `%d`, `%H`, `%M`) are the same as for `datetime.datetime.strftime` from the `datetime` module covered in Section 5.3.

```
df = pd.read_csv("misc/person.csv")
df["date_of_birth"] = pd.to_datetime(df["date_of_birth"])
df["date_of_birth"].head()
```

```
0   1993-01-12
1   1950-03-12
2   1983-08-09
3   1954-03-25
4   1969-04-09
Name: date_of_birth, dtype: datetime64[us]
```

```
# extract year and month of birth
df["birth_year"] = df["date_of_birth"].dt.year
df["birth_month"] = df["date_of_birth"].dt.month
df[["first_name", "last_name", "birth_year", "birth_month"]].head()
```

	first_name	last_name	birth_year	birth_month
0	André	Pieters	1993	1

	first_name	last_name	birth_year	birth_month
1	Ilaria	Jäggi	1950	3
2	Melissa	Pollak	1983	8
3	Vittorio	Cárdenas	1954	3
4	Gioffre	van Dam	1969	4

```
# compute age in completed years
today = pd.Timestamp.now()
print(today)
# Subtracting two datetimes gives a timedelta; .dt.days extracts the number of days from it
df["age"] = (today - df["date_of_birth"]).dt.days // 365
df[["first_name", "last_name", "date_of_birth", "age"]].head()
```

```
2026-04-06 13:42:38.555066
```

	first_name	last_name	date_of_birth	age
0	André	Pieters	1993-01-12	33
1	Ilaria	Jäggi	1950-03-12	76
2	Melissa	Pollak	1983-08-09	42
3	Vittorio	Cárdenas	1954-03-25	72
4	Gioffre	van Dam	1969-04-09	57

```
# format dates as strings
df["dob_formatted"] = df["date_of_birth"].dt.strftime("%-d %B %Y")
df[["first_name", "last_name", "dob_formatted"]].head()
```

	first_name	last_name	dob_formatted
0	André	Pieters	12 January 1993
1	Ilaria	Jäggi	12 March 1950
2	Melissa	Pollak	9 August 1983
3	Vittorio	Cárdenas	25 March 1954
4	Gioffre	van Dam	9 April 1969

7.11 SQL databases (`sqlite3`)

Python has built-in support for SQLite databases via the `sqlite3` module. This is useful for working with structured data that lives in a database.

- `sqlite3.connect(path)`: connect to a database (use `":memory:"` for an in-memory database).
- `cursor.execute(sql)`: execute an SQL statement.
- `cursor.fetchall()`: fetch all results.
- `pd.read_sql_query(sql, conn)`: run an SQL query and return the result as a DataFrame.
- `df.to_sql("table_name", conn, index=False)`: write a DataFrame into an SQL table.

```
import sqlite3

# connect to the person SQLite database
conn = sqlite3.connect("misc/person.db")
```

```
# querying with raw SQL
cursor = conn.cursor()
cursor.execute("SELECT first_name, last_name, country FROM persons WHERE country = 'Germany'")
for row in cursor.fetchall():
    print(row)
```

```
('Natalie', 'Burtscher', 'Germany')
('Katherine', 'Gehringer', 'Germany')
('Raimondo', 'van der Heijden', 'Germany')
('Idriz', 'Kitzmann', 'Germany')
('Denis', 'Egli', 'Germany')
('Ekaterina', 'Ledoux', 'Germany')
('Isaac', 'Käfer', 'Germany')
```

```
# reading SQL results directly into a pandas DataFrame
df_french = pd.read_sql_query(
    "SELECT first_name, last_name, town FROM persons WHERE country = 'France'",
    conn
)
df_french
```

	first_name	last_name	town
0	Ilaria	Jäggi	Naters
1	Gioffre	van Dam	Techer
2	Filipa	Backer	Nördlingen
3	Leyre	de Vroege	Rudolstadt

	first_name	last_name	town
4	Tygo	Seiwald	Benoit-la-Forêt
5	Roswitha	Göschl	Alcara Li Fusi
6	Tito	Mader	Groß-Enzersdorf

```
# aggregation in SQL
df_counts = pd.read_sql_query(
    "SELECT country, COUNT(*) AS n_persons FROM persons GROUP BY country ORDER BY n_persons D
    conn
)
df_counts
```

	country	n_persons
0	Switzerland	12
1	Kingdom of the Netherlands	10
2	Germany	7
3	France	7
4	Austria	5
5	Portugal	4
6	Italy	3
7	Spain	2

```
conn.close()
```

7.12 Exercises

All exercises use the person dataset from Section 7.8. Start by loading it:

```
df = pd.read_csv("misc/person.csv")
```

Exercise 1 Add a column `age` to `df` that contains each person's age in completed years (as an integer). Hint: use `pd.to_datetime` to convert `date_of_birth`, then compute the difference to today's date. Then use `map` to add a column `age_group` that maps each age to `"young"` (under 30), `"middle"` (30–59), or `"senior"` (60+).

```
# Exercise 1
```

Exercise 2 Add a column `birth_year` extracted from `date_of_birth`. Then use `df.describe()` and `df["birth_year"].value_counts()` to explore the distribution of birth years.

```
# Exercise 2
```

Exercise 3 Using the `generations` dictionary from Exercise 6, write a function `get_generation(year)` that returns the generation name (or `"Other"`). Add a column `generation` to `df` using this function. Then write a function `filter_generation(df, gen_name)` that returns only the rows belonging to a given generation. Use it to display all millennials. How many are there?

```
# Exercise 3
```

Exercise 4 Consider the following two DataFrames where the `city` column contains duplicate values:

```
shops = pd.DataFrame({
    "city": ["Berlin", "Berlin", "Paris", "Paris"],
    "shop": ["BookWorld", "TechStore", "Café Lune", "ModeParis"]
})

deliveries = pd.DataFrame({
    "city": ["Berlin", "Berlin", "Paris"],
    "item": ["books", "laptops", "croissants"]
})
```

What does `pd.merge(shops, deliveries, on="city")` return? Run it and explain the result. How many rows does the output have, and why?

```
# Exercise 4
```

Exercise 5 Create a DataFrame with columns `name`, `exam1`, `exam2`, `exam3` for 10 students (you can invent the data). Add a column `average` that contains the mean of the three exams. Filter to show only students with an average above 70.

```
# Exercise 5
```

Exercise 6 The following dictionary defines generational cohorts by birth year ranges:

```
generations = {
    "Silent Generation": (1928, 1945),
    "Baby Boomer": (1946, 1964),
    "Generation X": (1965, 1980),
    "Millennial": (1981, 1996),
    "Generation Z": (1997, 2012),
}
```

Write a function `get_generation(year)` that takes a birth year and returns the generation name (or `"Other"` if the year does not fall into any range). Then use `map` or `apply` to add a column `generation` to `df`. How many persons belong to each generation?

```
# Exercise 6
```

Exercise 7 The `hobbies` column in the CSV is a semicolon-separated string. The `.str` accessor on a pandas Series gives access to string methods, so `df["hobbies"].str.split(";")` splits each entry into a Python list. Use this to split the hobbies. Then find out: what is the most common hobby across all persons? Hint: `series.explode()` turns a Series of lists into individual rows (one row per list element); then use `value_counts`.

```
# Exercise 7
```

Exercise 8 Download a CSV file from the internet using `pd.read_csv(url)` (e.g. from this collection). Use `describe()`, `groupby()`, and a histogram to explore the data. Write a short summary of what you find.

```
# Exercise 8
```

Exercise 9 Create a DataFrame with some missing values (`np.nan`). Compare three strategies for handling them: (a) drop rows with any missing value, (b) fill with the column mean, (c) fill with the column median. How do the summary statistics (`describe()`) change in each case?

```
# Exercise 9
```

Exercise 10 Write a function `csv_to_sqlite(csv_path, db_path, table_name)` that reads a CSV file into a pandas DataFrame and writes it into an SQLite database. Then write a function `query_db(db_path, sql)` that runs an SQL query and returns the result as a DataFrame. Test with a CSV file of your choice.

```
# Exercise 10
```

Exercise 11 Fetch Germany's total population from the World Bank API for the years 1960–2023 (see Section 7.9 for how to use the API; the country code is `DEU`, the indicator is `SP.POP.TOTL`). Convert the year column to a proper datetime using `pd.to_datetime(df["year"], format="%Y")`. Then: (a) Add a column `decade` that contains the decade of each year (e.g. 1960, 1970, …). Hint: use integer division `// 10 * 10`. (b) Compute the mean population per decade using `groupby`. (c) Add a column `growth` that contains the year-over-year population change. Hint: `df["population"].diff()`. (d) In which year was the population growth the highest? In which year was it negative for the first time? (e) Resample the data to 5-year intervals using `df.set_index("date").resample("5YE").mean()` and plot the result.

```
# Exercise 11
```

8 Visualizing data

Download notebook.

In this chapter, we first discuss the general principles behind choosing the right chart type, and then learn how to create plots in Python using `matplotlib`.

8.1 Choosing the right chart

Before writing any code, it is worth thinking about what kind of chart best conveys your message. A useful framework (following Zelazny, *Say It with Charts*) proceeds in three steps:

1. **What is your message?** The chart form should not be chosen based on the data alone, but on what you want to say about it. The same table of numbers can lead to very different charts depending on the message. A good chart title states the message, not just the topic — for example, "Pasta recipes doubled on the blog since March" instead of "Recipe overview".

2. **What type of comparison does your message involve?** Every message implies one of five basic comparison types:

   - **Structure**: What share does each part have of the whole? Keywords: *share, percentage, proportion.* Example: "Italian food accounts for 40% of restaurant visits."
   - **Ranking**: How do items compare in size or order? Keywords: *larger than, smaller than, equal.* Example: "Spain is the most popular travel destination."
   - **Time series**: How does a quantity change over time? Keywords: *increase, decrease, trend, fluctuation.* Example: "Library visits have grown steadily since January."
   - **Frequency distribution**: How are values distributed across ranges? Keywords: *concentration, distribution, range.* Example: "Most hiking trails in the region are between 8 and 15 km long."
   - **Correlation**: Is there a relationship between two variables? Keywords: *relates to, follows, independent of.* Example: "There is no connection between cooking time and taste rating."

3. **Which chart form fits the comparison type?** There are five basic chart forms, and each is best suited for certain comparison types:

- **Pie chart**: Shows parts of a whole. Best for **structure** comparisons, but use sparingly — bar charts are almost always more readable.
- **Box plot**: Best for comparing the **distribution** of multiple groups side by side (medians, quartiles, outliers). For a single distribution, a histogram (which is a column chart of binned data) is more informative. **Violin plots** are a refined version of box plots that additionally show the shape of the distribution.
- **Column chart** (vertical bars): The workhorse for **time series** and **frequency distributions** when the number of data points is small (up to about six).
- **Line chart**: Best for **time series** and **frequency distributions** with many data points. Together with column charts, these should cover about half of all charts.
- **Bar chart** (horizontal bars): Extremely versatile. Best for **ranking** comparisons.
- **Scatter plot**: Best for **correlation** comparisons. Also works for time series and frequency distributions with very many data points.

Beyond these basic forms, additional chart types are frequently used in data analysis, such as **heatmaps** (colored grids for correlation matrices or similar two-dimensional data).

The following matrix (adapted from Zelazny, *Say It with Charts*) summarizes which chart form to use for which comparison type. Each cell contains a small example of the recommended chart form.

8.2 Plotting with `matplotlib`

The standard plotting library in Python is `matplotlib`. Its `pyplot` module provides a MATLAB-like interface. Here, `plt` refers to `matplotlib.pyplot`.

- `plt.figure(figsize=(w, h))`: create a new figure with given size in inches.
- `plt.xlabel("name")`, `plt.ylabel("name")`, `plt.title("name")`: add axis labels and title.
- `plt.xticks(positions, labels)`, `plt.yticks(positions, labels)`: set tick positions and labels on the x/y axis.
- `plt.legend()`: show legend. Options include `loc` for positioning (e.g., `"upper left"`), `fontsize`, and `title` for the legend title.
- `plt.show()`: display the figure.
- `plt.axvline(x)` / `plt.axhline(y)`: draw a vertical / horizontal line across the plot.

```
import numpy as np
import matplotlib.pyplot as plt
# We will generate some random data here, so we need a random number generator.
rng = np.random.default_rng(0)
```

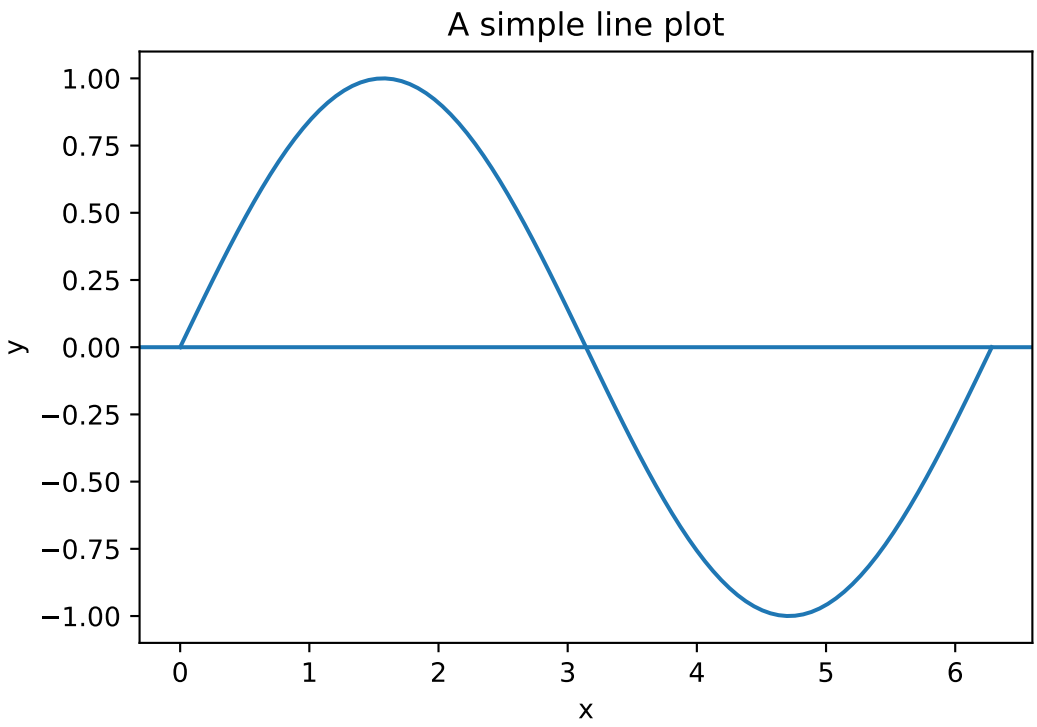
Figure 8.1: Chart forms vs. comparison types (after Zelazny, *Say It with Charts*)

8.3 Line plots (`plot`)

A line plot connects data points with straight line segments. It is the standard chart for **time series** and continuous functions.

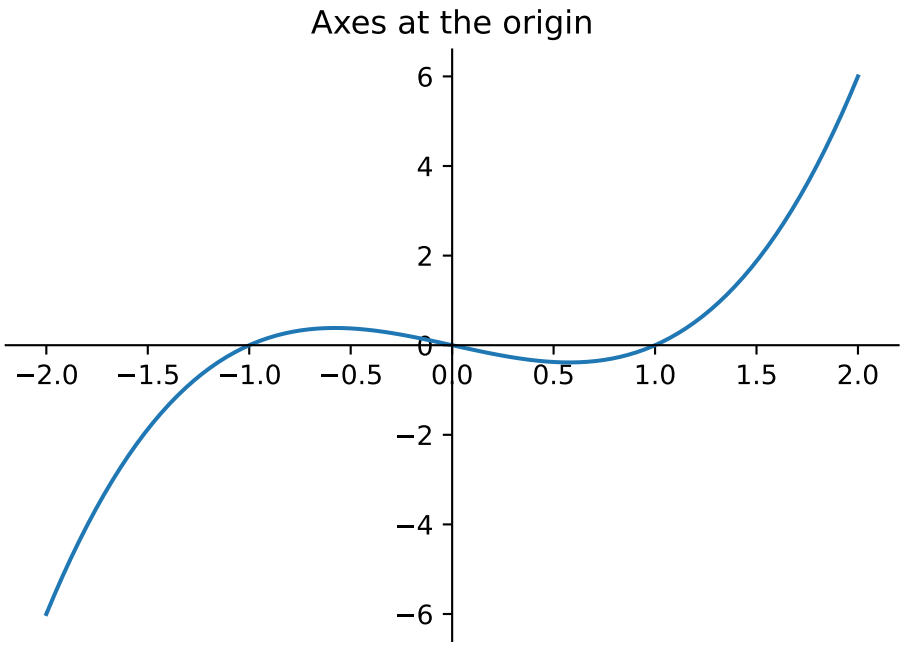
- `plt.plot(x, y, fmt, label=..., linestyle=...)`: line plot. The optional `fmt` is a shorthand format string combining color, marker, and line style, e.g. `"r*"` (red stars), `"b--"` (blue dashed), `"go-"` (green circles with solid line). Colors: `"r"` red, `"b"` blue, `"g"` green, `"k"` black. Markers: `"o"` circle, `"*"` star, `"s"` square, `"^"` triangle. The `linestyle` (or `ls`) option controls the line appearance: `"-"` solid (default), `"--"` dashed, `"-."` dash-dot, `":"` dotted.

```
x = np.linspace(0, 2 * np.pi, 100)
y = np.sin(x)

plt.figure(figsize=(6, 4))
plt.plot(x, y, label="sin(x)")
plt.axhline(0)
plt.xlabel("x")
plt.ylabel("y")
plt.title("A simple line plot")
plt.show()
```



By default, `matplotlib` draws a rectangular box around the plot. For mathematical function plots, it is often cleaner to place the x- and y-axes at the origin and remove the box. This requires `fig, ax = plt.subplots(...)` to access the `ax.spines` object (see also Section 8.8). Each **spine** is one of the four border lines of the plot area (`"left"`, `"right"`, `"top"`, `"bottom"`).

```
x = np.linspace(-2, 2, 100)

fig, ax = plt.subplots(figsize=(6, 4))
ax.plot(x, x**3 - x, label="$x^3 - x$")

# move the left and bottom spines to x=0 and y=0
ax.spines["left"].set_position(("data", 0))
ax.spines["bottom"].set_position(("data", 0))

# hide the top and right spines
ax.spines["top"].set_visible(False)
ax.spines["right"].set_visible(False)

ax.set_title("Axes at the origin")
plt.show()
```

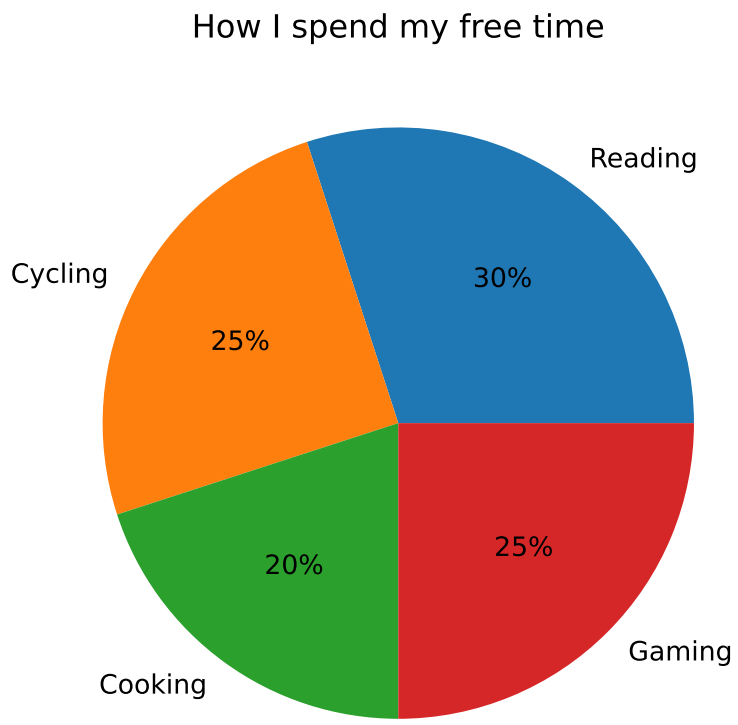


8.4 Pie charts (`pie`)

A pie chart shows parts of a whole. Use sparingly — bar charts are almost always more readable.

- `plt.pie(values, labels=...)`: pie chart. Use `autopct` to display percentages.

```
labels = ["Reading", "Cycling", "Cooking", "Gaming"]
values = [30, 25, 20, 25]

plt.figure(figsize=(5, 5))
plt.pie(values, labels=labels, autopct="%1.0f%%")
plt.title("How I spend my free time")
plt.show()
```



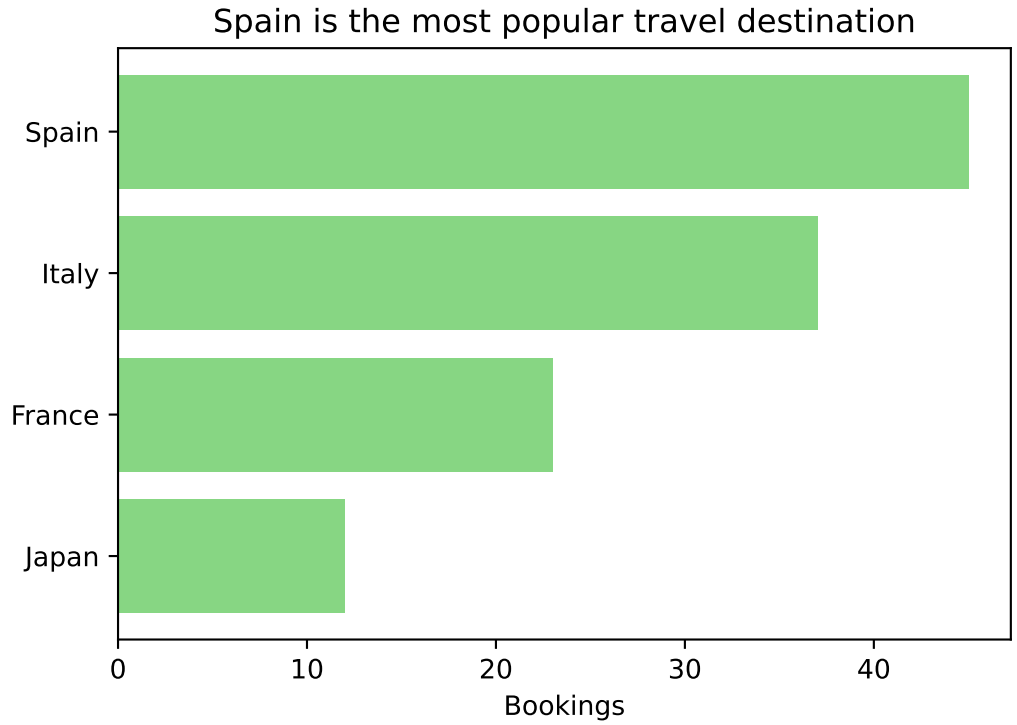
8.5 Bar charts (`bar`, `barh`)

Bar charts are the most versatile chart form, suitable for **structure**, **ranking**, and **comparison** tasks. Vertical bars are created with `plt.bar`, horizontal bars with `plt.barh`.

- `plt.bar(categories, values)`: vertical bar plot. Use `bottom=...` to stack bars on top of previous ones, `yerr=...` to add error bars, and `capsize=...` to set the width of the error bar caps. Use `color=...` to set the bar color, and `alpha=...` (0 to 1) to control its transparency.
- `plt.barh(categories, values)`: horizontal bar plot.

```
# horizontal bar chart (ranking)
categories = ["Japan", "France", "Italy", "Spain"]
values = [12, 23, 37, 45]

plt.figure(figsize=(6, 4))
plt.barh(categories, values, color="#6acc65", alpha=0.8)
plt.xlabel("Bookings")
```

```
plt.title("Spain is the most popular travel destination")
plt.show()
```

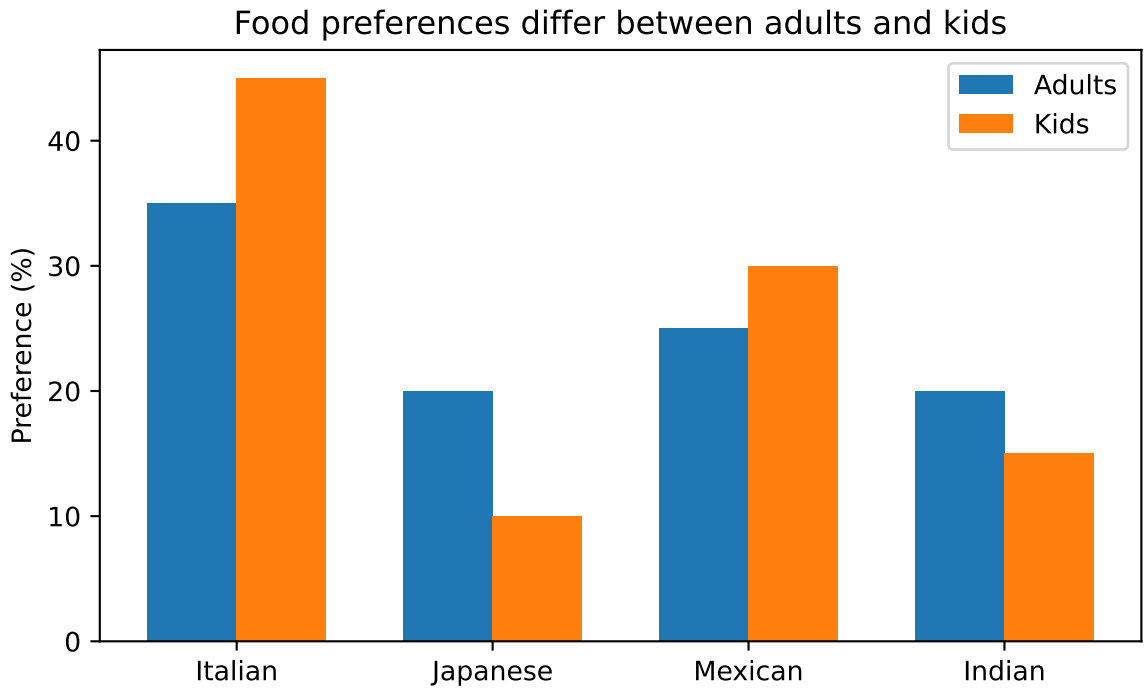


Bar charts become even more useful when comparing multiple series side by side, stacking them, showing differences, or adding error bars. To place two series next to each other, shift the bar positions by the bar width using `np.arange`.

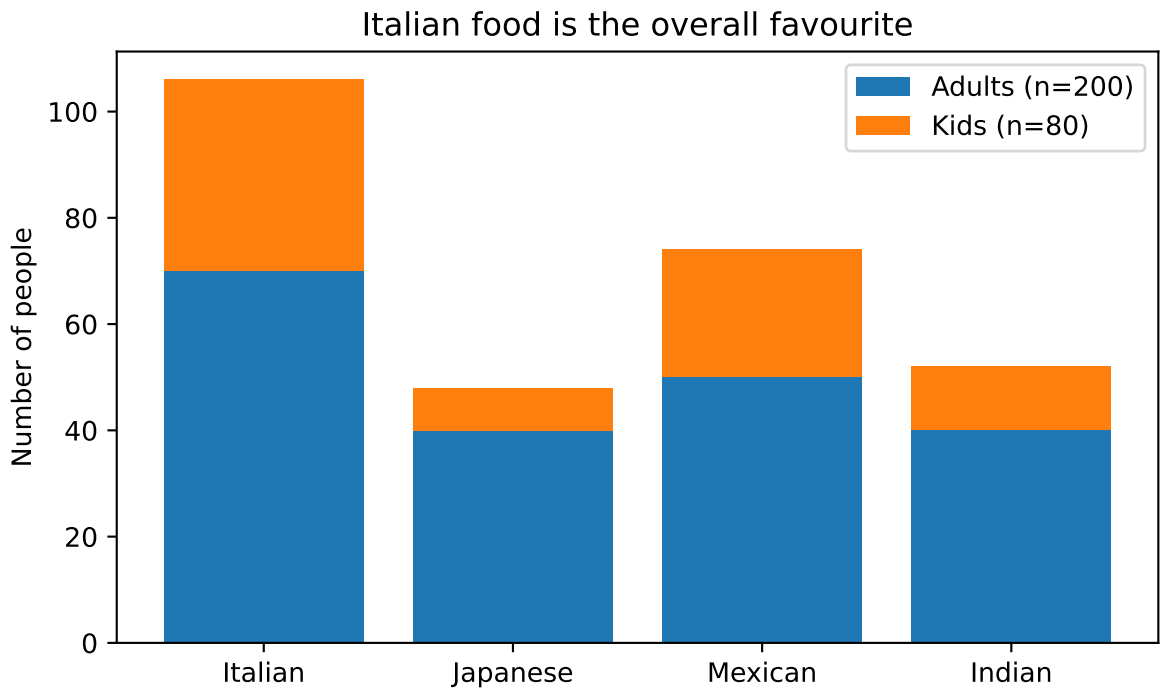
```
# survey data: absolute number of people who named each cuisine as favourite
cuisines = ["Italian", "Japanese", "Mexican", "Indian"]
adults_abs = [70, 40, 50, 40]   # 200 adults surveyed
kids_abs = [36, 8, 24, 12]     # 80 kids surveyed
```

```
# grouped bar chart: compare percentages
# (so group sizes don't distort the picture)
adults_pct = [a / sum(adults_abs) * 100 for a in adults_abs]
kids_pct = [k / sum(kids_abs) * 100 for k in kids_abs]

x = np.arange(len(cuisines))
width = 0.35
```

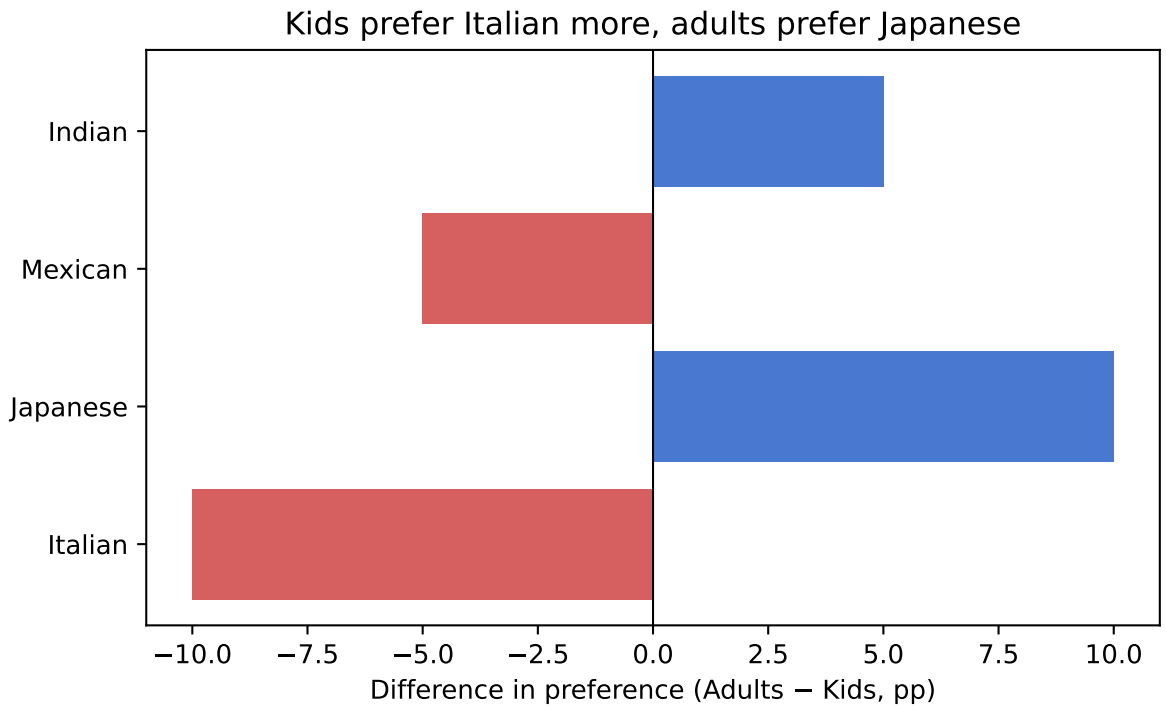
```
plt.figure(figsize=(7, 4))
plt.bar(x - width/2, adults_pct, width, label="Adults")
plt.bar(x + width/2, kids_pct, width, label="Kids")
plt.xticks(x, cuisines)
plt.ylabel("Preference (%)")
plt.title("Food preferences differ between adults and kids")
plt.legend()
plt.show()
```




```
# stacked bar chart: absolute numbers show the total count per cuisine
plt.figure(figsize=(7, 4))
plt.bar(cuisines, adults_abs, label="Adults (n=200)")
plt.bar(cuisines, kids_abs, bottom=adults_abs, label="Kids (n=80)")
plt.ylabel("Number of people")
plt.title("Italian food is the overall favourite")
plt.legend()
plt.show()
```

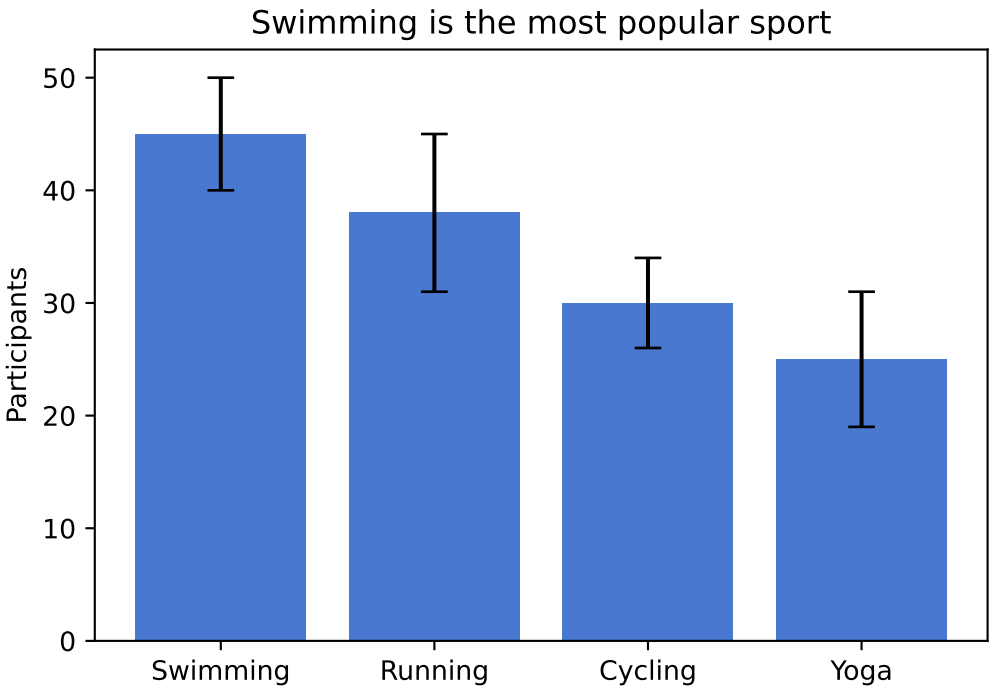



```
# difference chart: horizontal bars going left and right from a center axis
diff = [a - b for a, b in zip(adults_pct, kids_pct)]
colors = ["#4878cf" if d >= 0 else "#d65f5f" for d in diff]

plt.figure(figsize=(7, 4))
plt.barh(cuisines, diff, color=colors)
plt.axvline(0, color="black", linewidth=0.8)
plt.xlabel("Difference in preference (Adults - Kids, pp)")
plt.title("Kids prefer Italian more, adults prefer Japanese")
plt.show()
```



You can add **error bars** to a bar chart to show uncertainty (e.g., standard deviations or confidence intervals) using the `yerr` parameter.

```
# bar chart with error bars
sports = ["Swimming", "Running", "Cycling", "Yoga"]
participants = [45, 38, 30, 25]
std_dev = [5, 7, 4, 6]

plt.figure(figsize=(6, 4))
plt.bar(sports, participants, yerr=std_dev, capsize=5, color="#4878cf")
plt.ylabel("Participants")
plt.title("Swimming is the most popular sport")
plt.show()
```

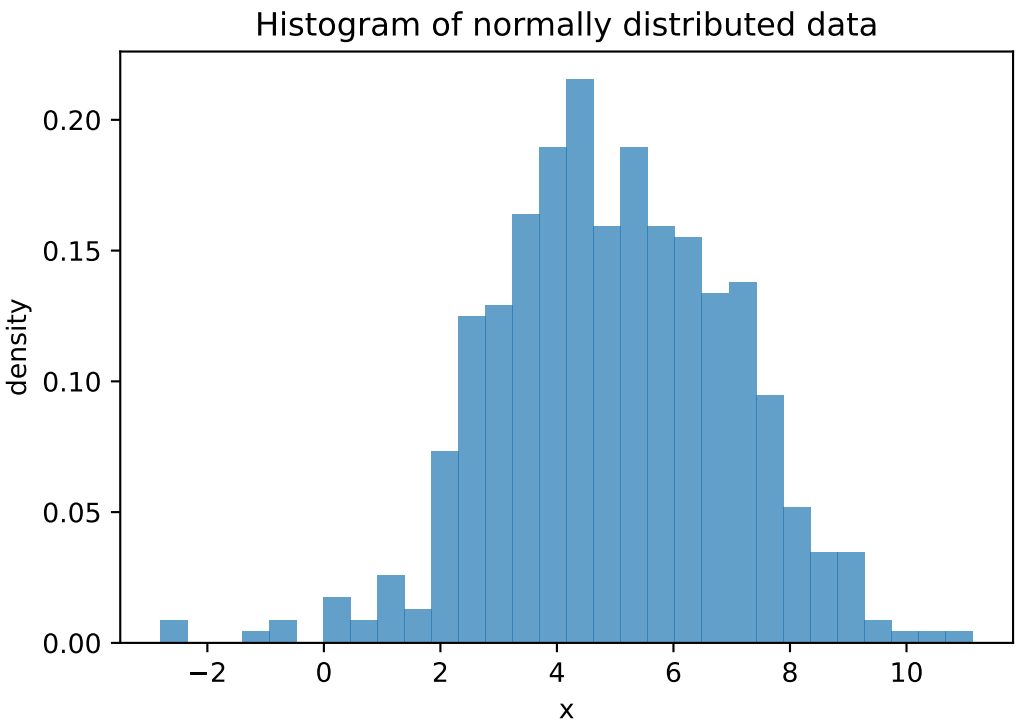


8.6 Histograms (`hist`)

A histogram groups numeric data into bins and shows the frequency or density of each bin. It is the standard chart for **frequency distributions**.

- `plt.hist(data, bins=..., density=...)`: histogram. Use `density=True` to normalize to a probability density, `alpha=...` (0 to 1) to make the bars semi-transparent, and `rwidth=...` (0 to 1) to control the bar width relative to the bin width (default 1.0, i.e. no gaps).

```
data = rng.normal(loc=5, scale=2, size=500)

plt.figure(figsize=(6, 4))
plt.hist(data, bins=30, density=True, alpha=0.7)
plt.xlabel("x")
plt.ylabel("density")
plt.title("Histogram of normally distributed data")
plt.show()
```

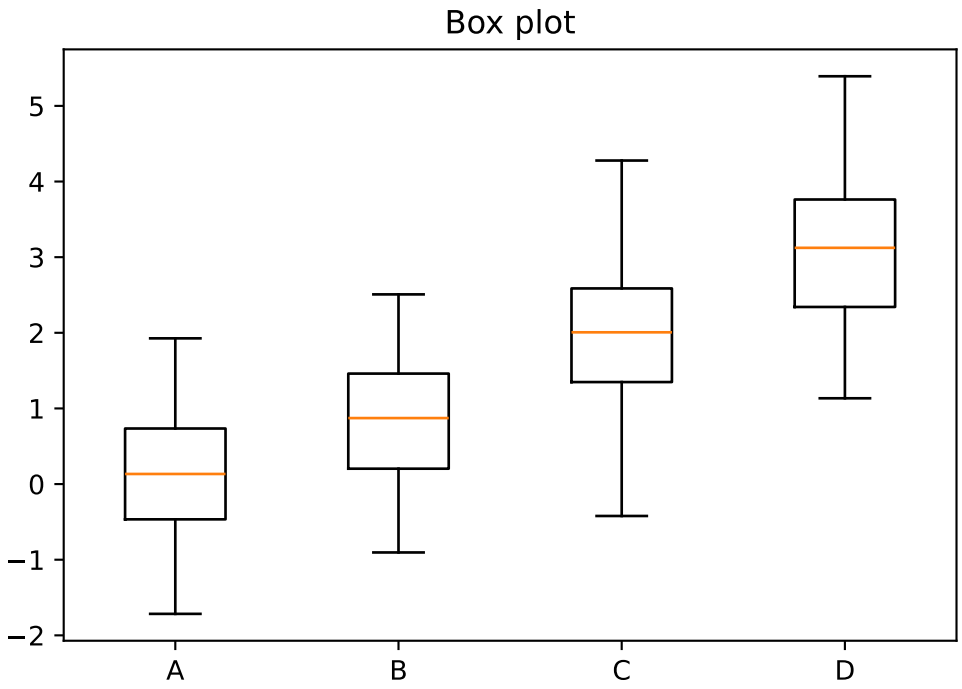


8.7 Box plots (`boxplot`)

A box plot summarizes the distribution of a dataset. The **box** spans from the first quartile (Q1, 25th percentile) to the third quartile (Q3, 75th percentile), so it contains the middle 50% of the data. The line inside the box marks the **median** (Q2, 50th percentile). The **whiskers** extend from the box to the most extreme data points that are still within 1.5 times the interquartile range (IQR = Q3 − Q1) from the box edges. Points beyond the whiskers are shown as individual dots and considered **outliers**. Box plots are especially useful for comparing distributions across groups.

- `plt.boxplot(data_list, tick_labels=...)`: box plot. Pass a list of arrays to compare multiple groups, and `tick_labels` to label them.

```
categories = ["A", "B", "C", "D"]
data_groups = [rng.normal(loc=m, size=50) for m in [0, 1, 2, 3]]

plt.figure(figsize=(6, 4))
plt.boxplot(data_groups, tick_labels=categories)
plt.title("Box plot")
plt.show()
```



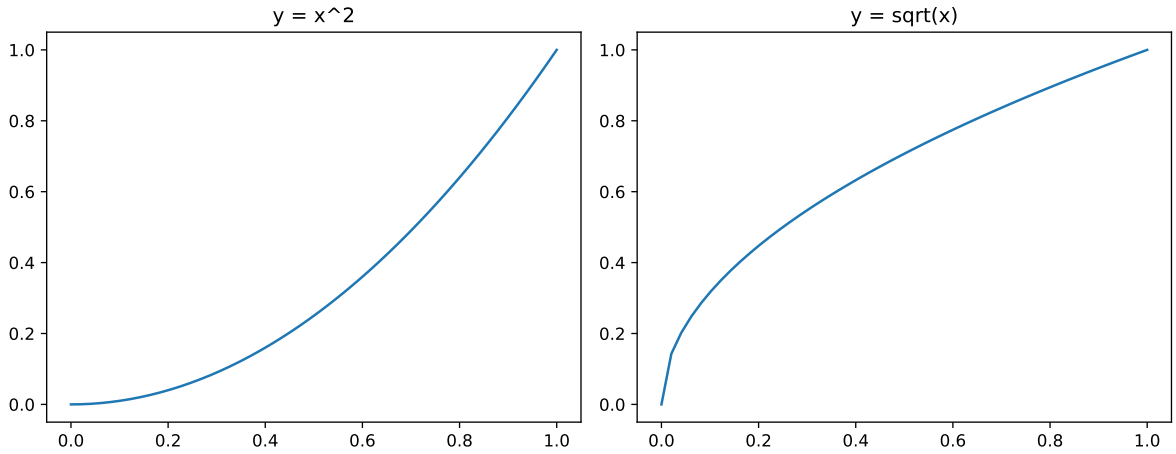
8.8 Subplots (`subplots`)

Subplots allow placing multiple plots in a single figure. Each subplot is an independent `axes` object that supports the same methods as `plt` but prefixed with `ax.` or `axes[i].` (e.g., `axes[0].plot(...)`, `axes[0].set_title(...)`).

- `fig, axes = plt.subplots(nrows, ncols, figsize=...)`: create a grid of subplots. For a single subplot, use `fig, ax = plt.subplots(figsize=...)`.
- `plt.tight_layout()`: automatically adjust spacing so labels do not overlap.
- `ax.spines["side"].set_position(("data", value))`: move a spine (the border lines of the plot area) so that it crosses the axis at `value`. The `"side"` can be `"left"`, `"right"`, `"top"`, or `"bottom"`.
- `ax.spines["side"].set_visible(False)`: hide a spine entirely.

```
fig, axes = plt.subplots(1, 2, figsize=(10, 4))

axes[0].plot(np.linspace(0, 1, 50), np.linspace(0, 1, 50)**2)
axes[0].set_title("y = x^2")
```

```
axes[1].plot(np.linspace(0, 1, 50), np.sqrt(np.linspace(0, 1, 50)))
axes[1].set_title("y = sqrt(x)")

plt.tight_layout()
plt.show()
```

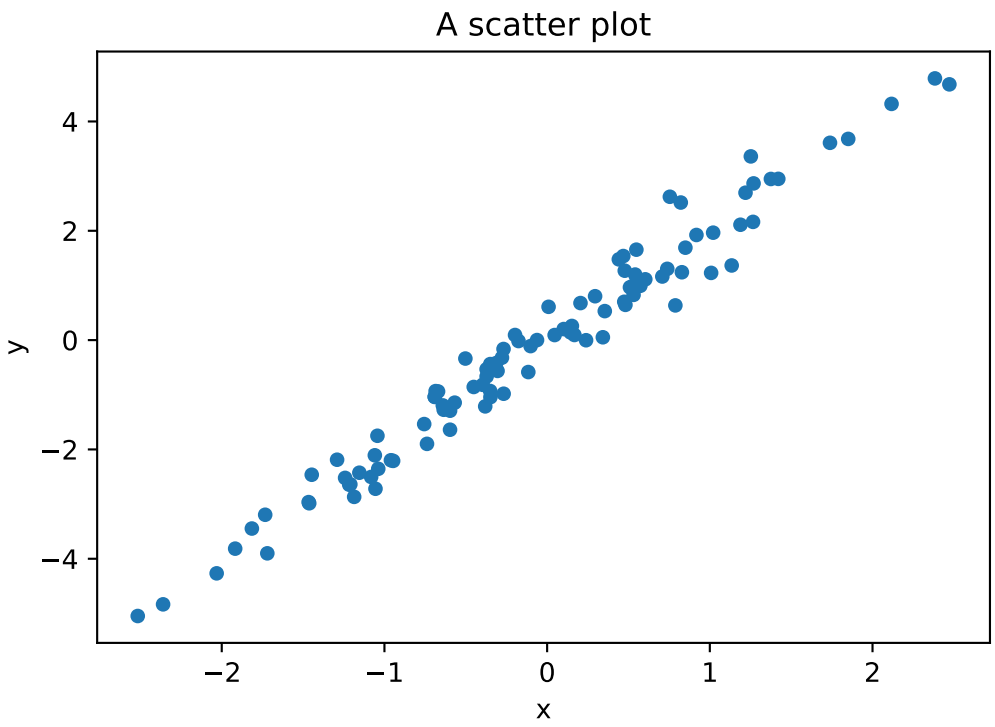


8.9 Scatter plots (`scatter`)

A scatter plot shows individual data points without connecting them. It is the primary chart for **correlation** comparisons.

- `plt.scatter(x, y, s=..., label=...)`: scatter plot (`s` is marker size).

```
x = rng.normal(size=100)
y = 2 * x + 0.5 * rng.normal(size=100)

plt.figure(figsize=(6, 4))
plt.scatter(x, y, s=20)
plt.xlabel("x")
plt.ylabel("y")
plt.title("A scatter plot")
plt.show()
```



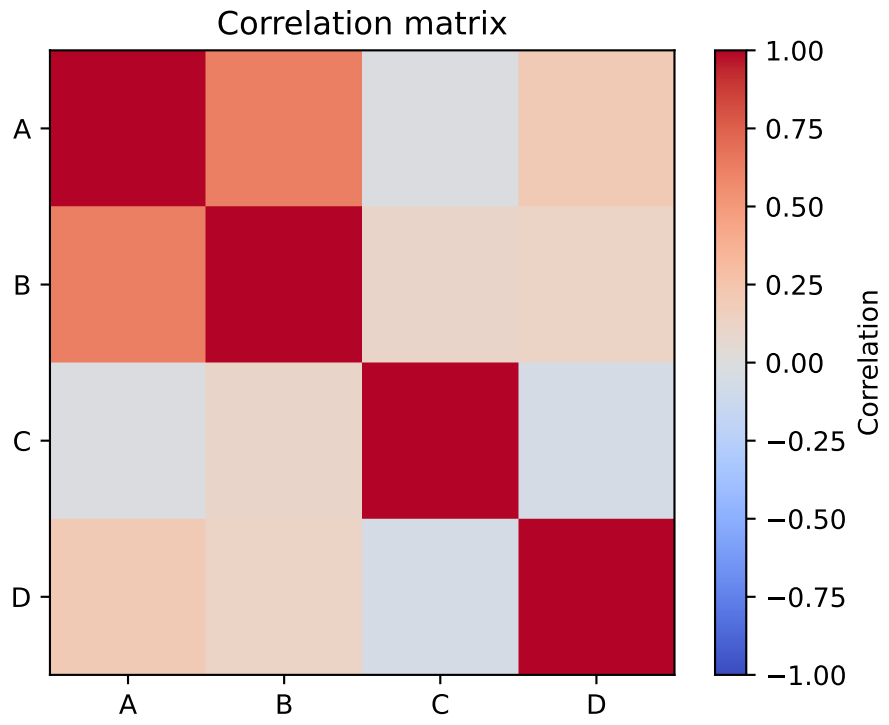
8.10 Heatmaps (`imshow`)

A heatmap displays a matrix of values as a grid of colored cells. It is useful for visualizing correlation matrices, confusion matrices, or any two-dimensional data where patterns emerge from color. A related function is `plt.contourf`, which draws filled contour lines for data given on a 2d grid (see Section 9.4 for an example).

- `plt.imshow(matrix, cmap=...)`: display a matrix as a colored grid. Use `vmin`/`vmax` to fix the color range. Common colormaps: `"viridis"` (default), `"coolwarm"` (diverging, blue to red), `"hot"`, `"gray"`, `"Blues"`, `"RdYlGn"`.
- `plt.contourf(X, Y, Z, levels=..., cmap=...)`: draw filled contour lines for values `Z` on a grid defined by `X` and `Y`. Use `levels` to control the number of contour levels.
- `plt.colorbar(label=...)`: add a color bar to the current plot, with an optional label.
- `np.corrcoef(data, rowvar=False)`: compute the correlation matrix of the columns of `data` (see Section 9.1 for more on correlation).

```
# heatmap of a correlation matrix
data = rng.normal(size=(100, 4))
data[:, 1] += 0.8 * data[:, 0] # introduce correlation between columns 0 and 1
```

```
labels = ["A", "B", "C", "D"]
corr = np.corrcoef(data, rowvar=False)

plt.figure(figsize=(5, 4))
plt.imshow(corr, cmap="coolwarm", vmin=-1, vmax=1)
plt.xticks(range(4), labels)
plt.yticks(range(4), labels)
plt.colorbar(label="Correlation")
plt.title("Correlation matrix")
plt.tight_layout()
plt.show()
```



8.11 Exercises

All exercises in this section use the World Bank Indicators API (see Section 7.9). The following helper code fetches population data for a list of countries over a range of years. You may reuse and adapt it.

```
import requests

def wb_population(countries, date_range="1960:2023"):
    """Fetch total population from the World Bank API.

    countries: list of ISO3 codes, e.g. ["DEU", "FRA", "ITA"]
    date_range: year range as "start:end"
    """
    codes = ";".join(countries)
    url = f"https://api.worldbank.org/v2/country/{codes}/indicator/SP.POP.TOTL"
    response = requests.get(url, params={
        "date": date_range, "format": "json", "per_page": 5000
    })
    data = response.json()[1]
    df = pd.DataFrame([{
        "country": e["country"]["value"],
        "iso3": e["countryiso3code"],
        "year": int(e["date"]),
        "population": e["value"]
    } for e in data if e["value"] is not None])
    return df.sort_values(["iso3", "year"])
```

Exercise 1 Use `wb_population` to fetch the world population from 1960 to 2023 (ISO3 code `"WLD"`). Plot the result as a **line chart** with the year on the x-axis and population on the y-axis. Add a title that states the trend you observe.

```
# Exercise 1
```

Exercise 2 Fetch the 2023 population for these countries: Germany (`DEU`), France (`FRA`), Italy (`ITA`), Spain (`ESP`), Poland (`POL`). Display the result as a **horizontal bar chart**, sorted by population. Which country is the most populous?

```
# Exercise 2
```

Exercise 3 Fetch population data from 1960 to 2023 for Germany, France, and Italy. Plot all three as **line charts in the same figure** with a legend. Convert the year column to a datetime using `pd.to_datetime(df["year"], format="%Y")` and use it as the x-axis.

```
# Exercise 3
```

Exercise 4 A survey asked 80 students about their favourite season. The results: Spring 18, Summer 30, Autumn 20, Winter 12. Display the results as a **pie chart** with percentages. Then create a second version as a **horizontal bar chart**. Which chart is easier to read?

```
# Exercise 4
```

Exercise 5 Generate four groups of random data: `rng.normal(loc=m, scale=s, size=50)` with `(m, s)` = (5, 1), (6, 2), (5.5, 0.5), (7, 1.5). Display them as a **box plot**. Which group has the largest spread? Are there outliers?

```
# Exercise 5
```

Exercise 6 Generate 500 samples from a normal distribution with mean 170 and standard deviation 10 (think: heights in cm). Plot a **histogram** with 25 bins and `density=True`. Overlay the theoretical density curve using `np.linspace` and `stats.norm.pdf` (you may import `from scipy import stats`). Experiment with `rwidth=0.9` and `alpha=0.7`.

```
# Exercise 6
```

Exercise 7 Using the data from Exercise 3, compute the population growth per decade (1960s, 1970s, …, 2010s) for each country. Display the result as a **grouped bar chart** with decades on the x-axis and one bar per country.

```
# Exercise 7
```

Exercise 8 Fetch a second indicator — GDP per capita (`NY.GDP.PCAP.CD`) — for at least 20 European countries for the year 2022. Also fetch their population. Create a **scatter plot** of GDP per capita (x-axis) vs. population (y-axis). What do you observe? Hint: adapt the `wb_population` function by changing the indicator code.

```
# Exercise 8
```

Exercise 9 Using the population data from Exercise 3, compute the **correlation matrix** of the three countries' populations (hint: use `pivot` to reshape the DataFrame so that each country is a column, then use `np.corrcoef` or `df.corr()`). Display the result as a **heatmap**. Why are the correlations so high?

```
# Exercise 9
```

9 Analyzing data

Download notebook.

In this chapter, we learn about statistical analysis with `scipy.stats`, including descriptive statistics, probability distributions, statistical tests, and maximum likelihood estimation.

```
import numpy as np
import matplotlib.pyplot as plt
from scipy import stats
```

9.1 Descriptive statistics (`scipy.stats`)

The `scipy.stats` module provides functions for descriptive statistics and statistical tests. Here, `data` is a 1d array or list. Note that `stats.describe(data)` is different from the pandas method `df.describe()` (see Section 7.1): `stats.describe` works on a single 1d array and additionally returns skewness and kurtosis, while `df.describe()` gives summary statistics (count, mean, std, min, quartiles, max) for all numeric columns of a DataFrame.

- `stats.describe(data)`: returns count, min, max, mean, variance, skewness, kurtosis.
- `stats.tmean(data, limits=(low, high))`: trimmed mean — the mean computed only over values in the interval `[low, high]`. Without `limits`, it returns the ordinary mean. This is useful to reduce the influence of outliers.
- `stats.tstd(data, limits=(low, high))`: trimmed standard deviation (same idea).
- `stats.tvar(data, limits=(low, high))`: trimmed variance (same idea).
- `stats.pearsonr(x, y)`: Pearson correlation coefficient and p-value.
- `stats.spearmanr(x, y)`: Spearman rank correlation and p-value.

```
data = [1, 2, 3, 4, 5, 6, 7, 8]
result = stats.describe(data)
print("count =", result.nobs)
print("min, max =", result.minmax)
print("mean =", result.mean)
print("variance =", result.variance)
print("skewness =", result.skewness)
print("kurtosis =", result.kurtosis)
```

```
count = 8
min, max = (np.int64(1), np.int64(8))
mean = 4.5
variance = 6.0
skewness = 0.0
kurtosis = -1.2380952380952381
```

```
data = np.array([2.3, 4.1, 3.7, 5.2, 4.8, 3.3])
print("trimmed mean =", stats.tmean(data))
print("std =", stats.tstd(data))
print("median =", np.median(data))
```

```
trimmed mean = 3.9000000000000004
std = 1.0488088481701514
median = 3.9
```

```
# correlation
rng = np.random.default_rng(0)
x = rng.normal(size=50)
y = 2 * x + 0.5 * rng.normal(size=50)

r, p_value = stats.pearsonr(x, y)
print(f"Pearson r = {r:.4f}, p-value = {p_value:.6f}")
```

```
Pearson r = 0.9621, p-value = 0.000000
```

9.2 Probability distributions (`scipy.stats`)

`scipy.stats` provides a large number of probability distributions. Each distribution object has a common interface. Here, `dist` is a distribution object (e.g. `stats.norm`, `stats.expon`, `stats.poisson`).

- `dist.rvs(size=n, random_state=seed)`: draw `n` random samples. `size` can be an integer or a tuple for multi-dimensional output (e.g. `size=(3, 4)` for a 3×4 array). `random_state` accepts an integer seed or a `np.random.Generator` for reproducibility.
- `dist.pdf(x)` / `dist.pmf(x)`: density / probability mass function.
- `dist.cdf(x)`: cumulative distribution function.
- `dist.ppf(q)`: quantile function (inverse of `cdf`).
- `dist.mean()`, `dist.var()`, `dist.std()`: theoretical moments.
- `dist.fit(data)`: estimate parameters from data (maximum likelihood).

Each distribution takes its own parameters, e.g. `stats.norm(loc=mu, scale=sigma)`, `stats.poisson(mu=lam)`, `stats.binom(n=n, p=p)`. See the `scipy.stats` documentation for the full list.

For continuous distributions, the location-scale convention is used: `stats.norm(loc=mu, scale=sigma)` gives $\mathcal{N}(\mu, \sigma^2)$.

```
# standard normal
print("mean =", stats.norm.mean())
print("pdf(0) =", stats.norm.pdf(0))
print("cdf(1.96) =", stats.norm.cdf(1.96))
print("ppf(0.975) =", stats.norm.ppf(0.975))
```

```
mean = 0.0
pdf(0) = 0.3989422804014327
cdf(1.96) = 0.9750021048517795
ppf(0.975) = 1.959963984540054
```

```
# normal with given parameters
dist = stats.norm(loc=3, scale=2)
samples = dist.rvs(size=1000, random_state=42)
print("first 5 samples:", samples[:5])
print("theoretical mean =", dist.mean())
print("theoretical variance =", dist.var())
print("empirical mean =", np.mean(samples))
print("empirical variance =", np.var(samples))
```

```
first 5 samples: [3.99342831 2.7234714  4.29537708 6.04605971 2.53169325]
theoretical mean = 3.0
theoretical variance = 4.0
empirical mean = 3.0386641116446507
empirical variance = 3.8316199589260687
```

```
# exponential
print("Exp(1) mean =", stats.expon.mean())
print("Exp(1) samples:", stats.expon.rvs(size=5, random_state=0))
```

```
Exp(1) mean = 1.0
Exp(1) samples: [0.79587451 1.25593076 0.92322315 0.78720115 0.55104849]
```

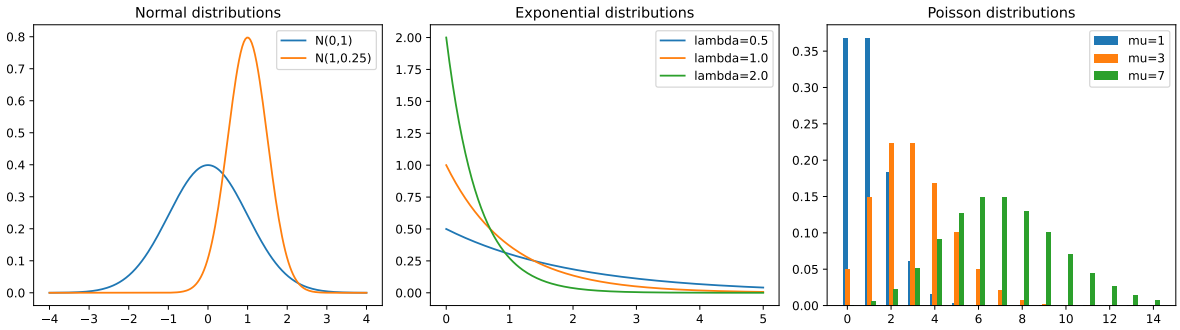
```
# Poisson
print("Poisson(3) pmf(2) =", stats.poisson.pmf(2, mu=3))
```

```
Poisson(3) pmf(2) = 0.22404180765538775
```

```
# binomial
print("Binom(10, 0.3) pmf(3) =", stats.binom.pmf(3, n=10, p=0.3))
print("Binom(10, 0.3) mean =", stats.binom.mean(n=10, p=0.3))
```

```
Binom(10, 0.3) pmf(3) = 0.2668279319999998
Binom(10, 0.3) mean = 3.0
```

```
# visualizing distributions
fig, axes = plt.subplots(1, 3, figsize=(14, 4))

x = np.linspace(-4, 4, 200)
axes[0].plot(x, stats.norm.pdf(x), label="N(0,1)")
axes[0].plot(x, stats.norm.pdf(x, loc=1, scale=0.5), label="N(1,0.25)")
axes[0].set_title("Normal distributions")
axes[0].legend()

x = np.linspace(0, 5, 200)
for lam in [0.5, 1.0, 2.0]:
    axes[1].plot(x, stats.expon.pdf(x, scale=1/lam), label=f"lambda={lam}")
axes[1].set_title("Exponential distributions")
axes[1].legend()

k = np.arange(0, 15)
for mu in [1, 3, 7]:
    axes[2].bar(k + 0.2*(mu-3)/4, stats.poisson.pmf(k, mu=mu), width=0.2, label=f"mu={mu}")
axes[2].set_title("Poisson distributions")
axes[2].legend()

plt.tight_layout()
plt.show()
```



9.3 Statistical tests

`scipy.stats` provides many standard statistical tests. Each of the tests below returns a named tuple with two entries: a test statistic (`.statistic`) and a p-value (`.pvalue`). The p-value is the probability of observing a test statistic at least as extreme as the one computed, assuming the null hypothesis is true. A small p-value (typically below 0.05) suggests rejecting the null hypothesis.

- `stats.ttest_1samp(data, popmean)`: one-sample t-test (is the mean equal to `popmean`?).
- `stats.ttest_ind(sample1, sample2)`: two-sample t-test (do two independent samples have the same mean?).
- `stats.chisquare(observed)`: chi-squared test (do observed frequencies match expected frequencies?).
- `stats.kstest(data, 'norm')`: Kolmogorov-Smirnov test (does the data come from the given distribution?).

```
rng = np.random.default_rng(0)
data = rng.normal(loc=5.2, scale=1.0, size=30)

# one-sample t-test
t_stat, p_value = stats.ttest_1samp(data, popmean=5.0)
print(f"t-statistic = {t_stat:.4f}, p-value = {p_value:.4f}")
```

```
t-statistic = 0.5225, p-value = 0.6053
```

```
# two-sample t-test
sample1 = rng.normal(loc=5.0, scale=1.0, size=40)
sample2 = rng.normal(loc=5.5, scale=1.0, size=40)
```

```
t_stat, p_value = stats.ttest_ind(sample1, sample2)
print(f"t-statistic = {t_stat:.4f}, p-value = {p_value:.4f}")
```

```
t-statistic = -1.2000, p-value = 0.2338
```

```
# chi-squared test
observed = np.array([18, 22, 20, 25, 15])
stat, p_value = stats.chisquare(observed)
print(f"chi2-statistic = {stat:.4f}, p-value = {p_value:.4f}")
```

```
chi2-statistic = 2.9000, p-value = 0.5747
```

```
# Kolmogorov-Smirnov test
data = rng.normal(loc=0, scale=1, size=100)
stat, p_value = stats.kstest(data, 'norm')
print(f"KS statistic = {stat:.4f}, p-value = {p_value:.4f}")
```

```
KS statistic = 0.0477, p-value = 0.9689
```

9.4 Maximum likelihood estimation

Maximum likelihood estimation (MLE) is one of the most important methods in statistics. Given i.i.d. observations $x_1, \ldots, x_n$ from a distribution with density $f(x;\theta)$, the likelihood function is

$$L(\theta) = \prod_{i=1}^{n} f(x_i;\theta),$$

and the log-likelihood is

$$\ell(\theta) = \sum_{i=1}^{n} \log f(x_i;\theta).$$

The MLE is the parameter value that maximizes $\ell(\theta)$, or equivalently minimizes $-\ell(\theta)$.

In `scipy.stats`, the `fit` method computes the MLE for any distribution:

- `dist.fit(data)`: returns the MLE of the distribution parameters. Note that `dist.fit` returns only point estimates, not standard errors.
- `dist.logpdf(x, ...)`: log-density, useful for computing the log-likelihood.

For cases where no closed-form MLE exists, we use numerical optimization via `scipy.optimize.minimize` (see Section 6.9). When using `method='BFGS'`, the result includes an approximate inverse Hessian (`result.hess_inv`), whose diagonal entries are the asymptotic variances of the parameter estimates. Taking square roots gives the standard errors.

```
# MLE for the normal distribution
rng = np.random.default_rng(0)
true_mu = 3.0
true_sigma = 1.5
data = rng.normal(loc=true_mu, scale=true_sigma, size=200)

# analytical MLE
mu_hat = np.mean(data)
sigma_hat = np.std(data)  # np.std uses 1/n, which is the MLE
print(f"true mu = {true_mu}, MLE mu = {mu_hat:.4f}")
print(f"true sigma = {true_sigma}, MLE sigma = {sigma_hat:.4f}")
```

```
true mu = 3.0, MLE mu = 3.0229
true sigma = 1.5, MLE sigma = 1.4418
```

```
# using scipy's fit method
mu_fit, sigma_fit = stats.norm.fit(data)
print(f"scipy fit: mu = {mu_fit:.4f}, sigma = {sigma_fit:.4f}")
```

```
scipy fit: mu = 3.0229, sigma = 1.4418
```

```
# MLE for the exponential distribution
# For Exp(lambda) with density lambda * exp(-lambda * x), the MLE is
# lambda_hat = 1 / x_bar
true_lambda = 2.0
# rng.exponential(scale, size) draws from an exponential distribution
data_exp = rng.exponential(scale=1/true_lambda, size=150)

lambda_hat = 1 / np.mean(data_exp)
print(f"true lambda = {true_lambda}, MLE lambda = {lambda_hat:.4f}")
```

```
true lambda = 2.0, MLE lambda = 1.7986
```

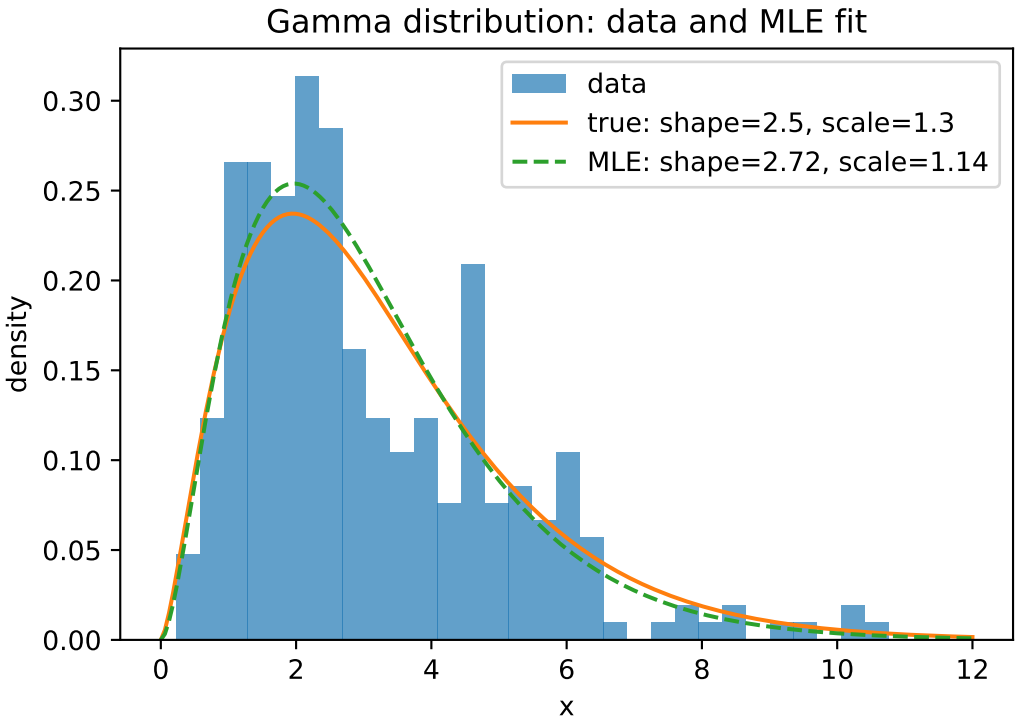
```
# MLE via numerical optimization (when no closed form exists)
# Example: Gamma distribution
from scipy import optimize

true_shape = 2.5
true_scale = 1.3
# rng.gamma(shape, scale, size) draws from a gamma distribution
data_gamma = rng.gamma(shape=true_shape, scale=true_scale, size=300)

def neg_log_likelihood_gamma(params):
    a, scale = params
    if a <= 0 or scale <= 0:
        return np.inf  # np.inf represents positive infinity
    return -np.sum(stats.gamma.logpdf(data_gamma, a=a, scale=scale))

# BFGS provides an approximate inverse Hessian, from which we get
# standard errors
result = optimize.minimize(neg_log_likelihood_gamma, x0=[1.0, 1.0],
    method='BFGS')
a_hat, scale_hat = result.x
print(f"true shape = {true_shape}, MLE shape = {a_hat:.4f}")
print(f"true scale = {true_scale}, MLE scale = {scale_hat:.4f}")

# standard errors from the inverse Hessian (asymptotic covariance matrix)
se = np.sqrt(np.diag(result.hess_inv))
print(f"std error of shape = {se[0]:.4f}")
print(f"std error of scale = {se[1]:.4f}")
```

```
true shape = 2.5, MLE shape = 2.7227
true scale = 1.3, MLE scale = 1.1411
std error of shape = 0.2061
std error of scale = 0.0987
```

```
# compare with scipy's built-in fit
# floc=0 fixes the location parameter to 0
a_fit, loc_fit, scale_fit = stats.gamma.fit(data_gamma, floc=0)
print(f"scipy fit: shape = {a_fit:.4f}, scale = {scale_fit:.4f}")
```

```
scipy fit: shape = 2.7227, scale = 1.1411
```

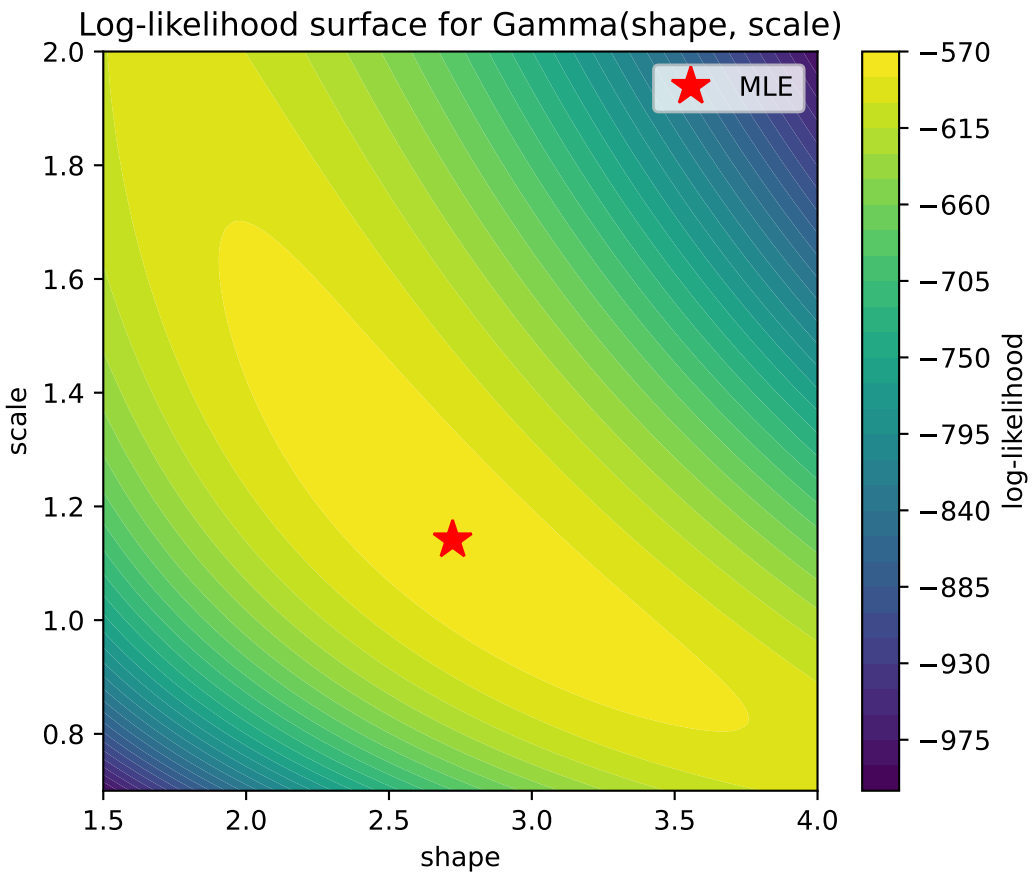
```
# visualizing the MLE fit
x_grid = np.linspace(0, 12, 200)

plt.figure(figsize=(6, 4))
plt.hist(data_gamma, bins=30, density=True, alpha=0.7, label="data")
plt.plot(x_grid, stats.gamma.pdf(x_grid, a=true_shape, scale=true_scale),
         label=f"true: shape={true_shape}, scale={true_scale}")
plt.plot(x_grid, stats.gamma.pdf(x_grid, a=a_hat, scale=scale_hat),
         label=f"MLE: shape={a_hat:.2f}, scale={scale_hat:.2f}",
         linestyle="--")
plt.xlabel("x")
plt.ylabel("density")
plt.title("Gamma distribution: data and MLE fit")
plt.legend()
plt.show()
```



The following plot shows the log-likelihood as a function of the two parameters shape and scale (for the gamma distribution fitted above). To evaluate a function on a 2d grid of parameter values, we use `np.meshgrid`: given two 1d arrays `shape_grid` and `scale_grid` of lengths m and n, `np.meshgrid(shape_grid, scale_grid)` returns two $n \times m$ arrays `SHAPE` and `SCALE` such that `(SHAPE[i,j], SCALE[i,j])` is the point `(shape_grid[j], scale_grid[i])`. This

allows evaluating the log-likelihood at every grid point with a simple loop (or vectorized). We store the results in an array created by `np.zeros_like(SHAPE)`, which returns an array of zeros with the same shape as `SHAPE`. The result is visualized with `plt.contourf`, which draws filled contour lines.

```
# log-likelihood surface for the Gamma distribution
shape_grid = np.linspace(1.5, 4.0, 100)
scale_grid = np.linspace(0.7, 2.0, 100)
SHAPE, SCALE = np.meshgrid(shape_grid, scale_grid)

# option 1: explicit loop (clear, but slow for large grids)
log_lik = np.zeros_like(SHAPE)
for i in range(len(scale_grid)):
    for j in range(len(shape_grid)):
        log_lik[i, j] = np.sum(stats.gamma.logpdf(data_gamma, a=SHAPE[i, j],
                scale=SCALE[i, j]))

# option 2: vectorized via broadcasting (much faster)
# data_gamma has shape (300,), SHAPE and SCALE have shape (100, 100)
# by reshaping data to (300, 1, 1), logpdf broadcasts to shape (300, 100, 100)
log_lik = np.sum(stats.gamma.logpdf(data_gamma[:, None, None],
        a=SHAPE[None, :, :], scale=SCALE[None, :, :]), axis=0)

plt.figure(figsize=(6, 5))
plt.contourf(SHAPE, SCALE, log_lik, levels=30, cmap="viridis")
plt.colorbar(label="log-likelihood")
plt.xlabel("shape")
plt.ylabel("scale")
plt.title("Log-likelihood surface for Gamma(shape, scale)")
# "r*" is a format string: "r" = red color, "*" = star marker
plt.plot(a_hat, scale_hat, "r*", markersize=15, label="MLE")
plt.legend()
plt.show()
```



Plotting the log-likelihood surface helps build geometric intuition: the MLE is the peak of the surface. For well-behaved models, the surface is concave near the maximum, and the curvature encodes the precision of the estimate (related to the Fisher information).

9.5 Linear regression

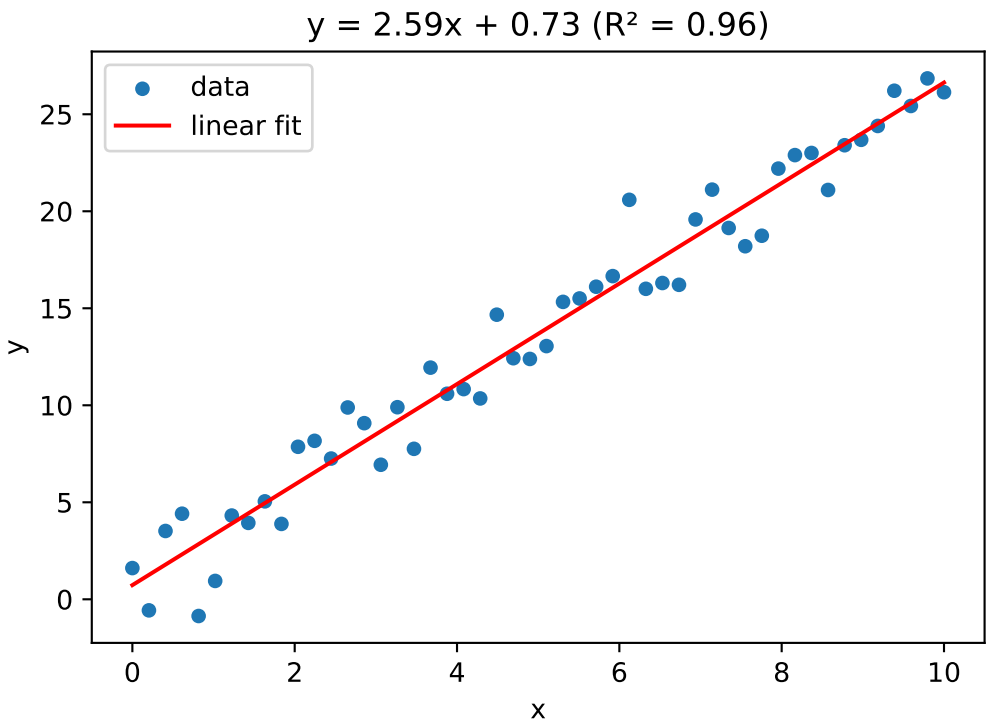
Linear regression fits a linear model $y = \beta_0 + \beta_1 x_1 + \cdots + \beta_p x_p + \varepsilon$ to data, where ε is the error term. The goal is to find the coefficients $\beta_0, \ldots, \beta_p$ that minimize the sum of squared residuals $\sum_i (y_i - \hat{y}_i)^2$. This is called **ordinary least squares** (OLS).

- `stats.linregress(x, y)`: simple linear regression ($y = \beta_0 + \beta_1 x$). Returns a named tuple with `slope`, `intercept`, `rvalue` (correlation coefficient), `pvalue` (for the null hypothesis that the slope is zero), and `stderr` (standard error of the slope).
- `np.polyfit(x, y, deg)`: fit a polynomial of degree `deg` to the data. For `deg=1`, this gives the same result as linear regression.
- `np.polyval(coeffs, x)`: evaluate a polynomial with the given coefficients at `x`.

```
# simple linear regression
rng = np.random.default_rng(42)
x = np.linspace(0, 10, 50)
y = 2.5 * x + 1.0 + rng.normal(scale=2.0, size=50)

result = stats.linregress(x, y)
print(f"slope = {result.slope:.4f}")
print(f"intercept = {result.intercept:.4f}")
print(f"R-squared = {result.rvalue**2:.4f}")
print(f"p-value = {result.pvalue:.6f}")
print(f"std error of slope = {result.stderr:.4f}")
```

```
slope = 2.5905
intercept = 0.7301
R-squared = 0.9629
p-value = 0.000000
std error of slope = 0.0734
```

```
# visualizing the fit
plt.figure(figsize=(6, 4))
plt.scatter(x, y, s=20, label="data")
plt.plot(x, result.slope * x + result.intercept, color="red",
    label="linear fit")
plt.xlabel("x")
plt.ylabel("y")
title = (f"y = {result.slope:.2f}x + {result.intercept:.2f}"
         f" (R² = {result.rvalue**2:.2f})")
plt.title(title)
plt.legend()
plt.show()
```



For multiple linear regression ($p > 1$ predictors), `scipy.stats.linregress` does not suffice. Instead, we can solve the normal equations directly using `np.linalg.lstsq`, which computes the least squares solution of an overdetermined system.

- `np.linalg.lstsq(A, b, rcond=None)`: solve $A\beta \approx b$ in the least squares sense. Returns a tuple of four values: the coefficient vector β, the sum of squared residuals (empty if `A` has fewer rows than columns), the rank of `A`, and its singular values.

```
# multiple linear regression: y = beta0 + beta1 * x1 + beta2 * x2
rng = np.random.default_rng(7)
n = 100
x1 = rng.normal(size=n)
x2 = rng.normal(size=n)
y = 3.0 + 1.5 * x1 - 2.0 * x2 + rng.normal(scale=0.5, size=n)

# design matrix with intercept column
A = np.column_stack([np.ones(n), x1, x2])
beta, residuals, rank, sv = np.linalg.lstsq(A, y, rcond=None)
print(f"intercept = {beta[0]:.4f} (true: 3.0)")
print(f"beta1 = {beta[1]:.4f} (true: 1.5)")
print(f"beta2 = {beta[2]:.4f} (true: -2.0)")
```

```
intercept = 2.9429 (true: 3.0)
beta1 = 1.5215 (true: 1.5)
beta2 = -1.9444 (true: -2.0)
```

```python
# R-squared for the multiple regression
y_pred = A @ beta
ss_res = np.sum((y - y_pred)**2)
ss_tot = np.sum((y - np.mean(y))**2)
r_squared = 1 - ss_res / ss_tot
print(f"R-squared = {r_squared:.4f}")
```

```
R-squared = 0.9469
```

The R-squared value ($R^2 = 1 - \mathrm{SS}_{\mathrm{res}}/\mathrm{SS}_{\mathrm{tot}}$) measures the fraction of variance in y explained by the model. A value close to 1 indicates a good fit.

9.6 Principal component analysis (PCA)

Principal component analysis (PCA) is a technique for reducing the dimensionality of a dataset while retaining as much variance as possible. Given n observations with p features, PCA finds new orthogonal axes (called **principal components**) along which the data varies most. The first principal component captures the most variance, the second the most remaining variance (orthogonal to the first), and so on. This is useful for visualization (projecting high-dimensional data to 2d), noise reduction, and understanding the structure of a dataset.

Mathematically, let $X \in \mathbb{R}^{n\times p}$ be the data matrix with n observations (rows) and p features (columns). Write the rows as $x_1, \dots, x_n \in \mathbb{R}^p$. First, center the data by subtracting the column-wise mean $\bar{x} = \frac{1}{n}\sum_{i=1}^{n} x_i$:

$$\tilde{x}_i = x_i - \bar{x}, \qquad \tilde{X} = \begin{pmatrix} \tilde{x}_1^\top \\ \vdots \\ \tilde{x}_n^\top \end{pmatrix} \in \mathbb{R}^{n\times p}.$$

The **sample covariance matrix** $C \in \mathbb{R}^{p\times p}$ has entries

$$C_{jk} = \frac{1}{n-1}\sum_{i=1}^{n} \tilde{x}_{ij}\,\tilde{x}_{ik}, \qquad j,k = 1,\dots,p,$$

or in matrix form $C = \frac{1}{n-1}\tilde{X}^\top\tilde{X}$. The diagonal entry C_{jj} is the sample variance of feature j, and C_{jk} ($j \neq k$) measures how features j and k co-vary: positive if they tend to increase together, negative if one increases when the other decreases, and zero if they are uncorrelated.

Since C is symmetric and positive semi-definite, it has an eigendecomposition $C = V\Lambda V^\top$, where $V = (v_1, \dots, v_p)$ is an orthogonal matrix of eigenvectors and $\Lambda = \text{diag}(\lambda_1, \dots, \lambda_p)$ contains the eigenvalues $\lambda_1 \geq \lambda_2 \geq \cdots \geq \lambda_p \geq 0$. The eigenvector v_j is the j-th **principal component direction**, and λ_j is the variance of the data when projected onto v_j, i.e. $\lambda_j = \text{Var}(\tilde{X}v_j) = \frac{1}{n-1}\|\tilde{X}v_j\|^2$. The fraction of total variance explained by the first k components is $\sum_{j=1}^{k} \lambda_j \,/\, \sum_{j=1}^{p} \lambda_j$.

To reduce the data from p dimensions to k, we project: $X_k = \tilde{X}V_k$, where $V_k = (v_1, \dots, v_k) \in \mathbb{R}^{p \times k}$ contains the first k eigenvectors. The resulting matrix $X_k \in \mathbb{R}^{n \times k}$ gives the coordinates of each observation in the new k-dimensional space.

- `np.cov(X, rowvar=False)`: compute $C = \frac{1}{n-1}\tilde{X}^\top\tilde{X}$, where rows of `X` are observations and columns are features. The option `rowvar=False` tells NumPy that rows are observations (the default assumes the opposite).
- `np.linalg.eigh(C)`: eigendecomposition of a symmetric matrix C. This is a variant of `np.linalg.eig` (see Section 6.6) specialized for symmetric (Hermitian) matrices: it guarantees real eigenvalues and returns them sorted in ascending order. Returns eigenvalues and eigenvectors as columns.
- Projection: `X_centered @ V[:, :k]` computes $X_k = \tilde{X}V_k$, i.e. projects the data onto the first `k` principal components (after reversing the order so that the largest eigenvalue comes first).

```
# generate correlated 3d data
rng = np.random.default_rng(42)
n = 200
z1 = rng.normal(size=n)
z2 = rng.normal(size=n)
z3 = rng.normal(size=n) * 0.1  # very little variance in this direction

# the three features are mixtures of z1, z2, z3
X = np.column_stack([
    2 * z1 + 0.5 * z2,
    z1 + z2,
    0.5 * z1 + 0.3 * z2 + z3
])
```

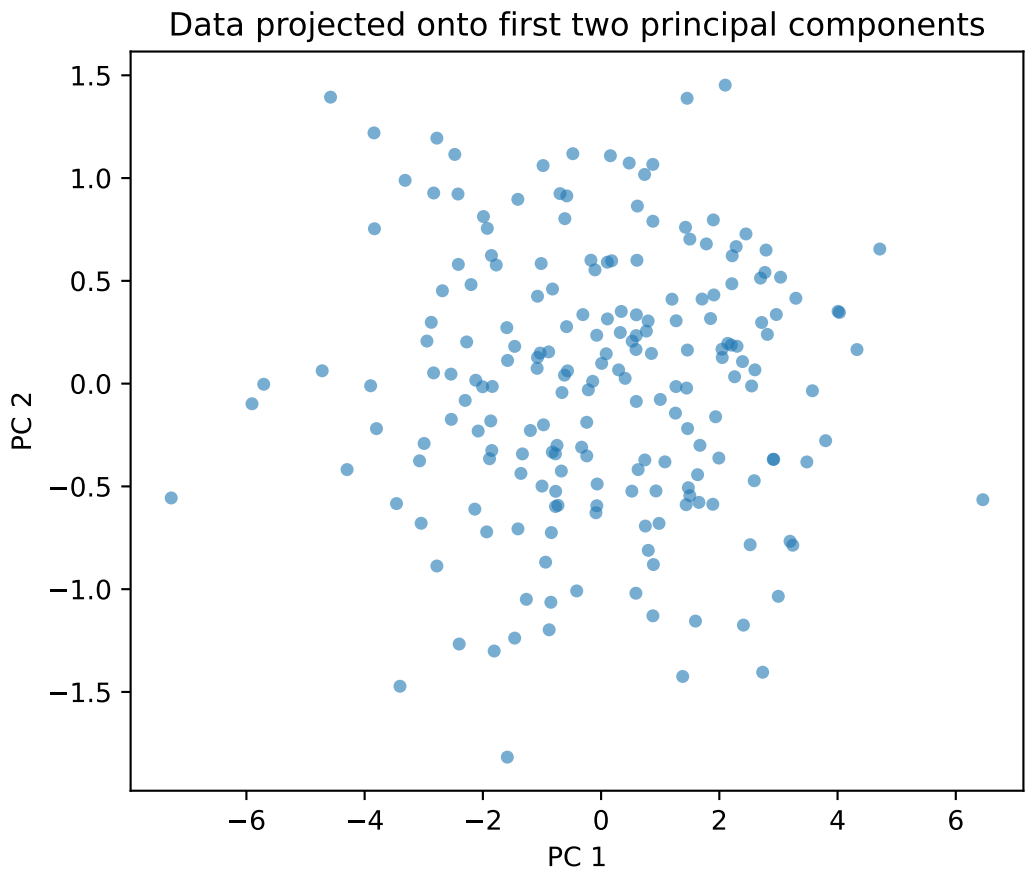
```
# step 1: center the data
X_centered = X - X.mean(axis=0)

# step 2: compute the covariance matrix
C = np.cov(X_centered, rowvar=False)
print("Covariance matrix:")
print(np.round(C, 3))
```

```
Covariance matrix:
[[3.245 1.917 0.873]
 [1.917 1.691 0.65 ]
 [0.873 0.65  0.277]]
```

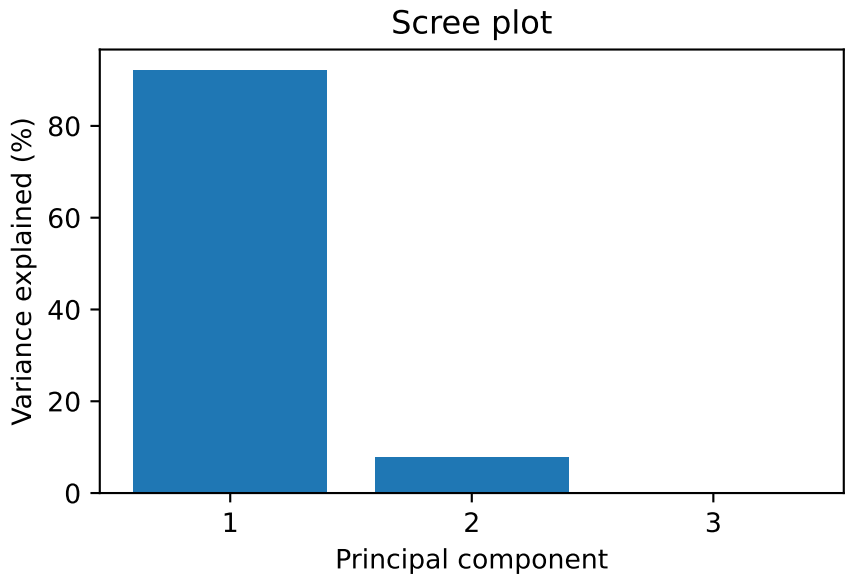
```python
# step 3: eigendecomposition
eigenvalues, eigenvectors = np.linalg.eigh(C)

# eigh returns eigenvalues in ascending order - reverse for descending
eigenvalues = eigenvalues[::-1]
eigenvectors = eigenvectors[:, ::-1]

print("Eigenvalues:", np.round(eigenvalues, 3))
print("Variance explained (%):", np.round(eigenvalues / eigenvalues.sum() * 100, 1))
```

```
Eigenvalues: [4.798 0.405 0.01 ]
Variance explained (%): [92.   7.8  0.2]
```

```python
# step 4: project onto the first 2 principal components
X_pca = X_centered @ eigenvectors[:, :2]

plt.figure(figsize=(6, 5))
plt.scatter(X_pca[:, 0], X_pca[:, 1], s=15, alpha=0.6)
plt.xlabel("PC 1")
plt.ylabel("PC 2")
plt.title("Data projected onto first two principal components")
plt.show()
```




```
# scree plot: how much variance does each component explain?
plt.figure(figsize=(5, 3))
plt.bar(range(1, len(eigenvalues) + 1), eigenvalues / eigenvalues.sum() * 100)
plt.xlabel("Principal component")
plt.ylabel("Variance explained (%)")
plt.title("Scree plot")
plt.xticks(range(1, len(eigenvalues) + 1))
plt.show()
```



The **scree plot** shows the variance explained by each principal component. A sharp drop suggests that only the first few components carry meaningful information — the rest is mostly noise. In this example, the third component explains very little variance, so projecting from 3d to 2d loses almost no information.

9.7 Exercises

Exercise 1 Load the Auto MPG dataset (originally from the UCI Machine Learning Repository) using `pd.read_csv`. Use `stats.describe` on the `mpg` column and compare the output with `df["mpg"].describe()`. Compute the Pearson and Spearman correlation between `weight` and `mpg`. Which is stronger? Why might they differ?

```
# Exercise 1
```

Exercise 2 Generate 1000 samples from a $\mathcal{N}(2, 3^2)$ distribution. Plot a histogram together with the true density, the density of the MLE fit (`stats.norm.fit`), and the density of a fit where you deliberately set $\mu = 0$ (wrong mean). Add a legend.

```
# Exercise 2
```

Exercise 3 Load the Advertising dataset (from the textbook *An Introduction to Statistical Learning* by James, Witten, Hastie, and Tibshirani) using `pd.read_csv`. Use `stats.linregress` to fit a simple linear regression of `sales` on `TV`. Report the slope, intercept,

R-squared, and p-value. Plot the data as a scatter plot with the regression line. Then fit a multiple regression of `sales` on `TV`, `radio`, and `newspaper` using `np.linalg.lstsq`. Compare the R-squared values. Which predictor contributes least?

```
# Exercise 3
```

Exercise 4 Load the Auto MPG dataset (same URL as Exercise 1). Select the numeric columns `mpg`, `cylinders`, `displacement`, `horsepower`, `weight`, `acceleration`. Drop rows with missing values. Perform PCA on the standardized data. How much variance do the first two components explain? Plot the data projected onto the first two principal components, coloring points by `cylinders`. What structure do you see?

```
# Exercise 4
```

Exercise 5 Generate two samples of size 50: one from an exponential distribution with rate 1, the other from a normal distribution with mean 1 and standard deviation 1. Use a Kolmogorov-Smirnov test (`stats.kstest`) to test whether each sample could come from a normal distribution. Interpret the p-values. Then use a chi-squared test on binned data to test the same hypothesis.

```
# Exercise 5
```

Exercise 6 A **QQ-plot** (quantile-quantile plot) compares the quantiles of a sample against the theoretical quantiles of a reference distribution. If the data follow the reference distribution, the points lie on a straight line. Use `stats.probplot(data, dist="norm", plot=plt)` to create QQ-plots for: (a) 200 samples from a standard normal, (b) 200 samples from an exponential distribution, (c) 200 samples from a t-distribution with 3 degrees of freedom (`stats.t.rvs(df=3, size=200)`). Which sample deviates most from normality, and how does the deviation appear in the QQ-plot?

```
# Exercise 6
```

Exercise 7 Generate two samples: `sample_a` from $\mathcal{N}(5, 1)$ (size 30) and `sample_b` from $\mathcal{N}(5.5, 1)$ (size 30). Perform a two-sample t-test. Then repeat this experiment 1000 times and count how often the test rejects at level $\alpha = 0.05$. This is the empirical power of the test. How does it change if you increase the sample size to 100?

```
# Exercise 6
```

Exercise 8 Using the World Bank API (see Section 7.9), fetch GDP per capita (`NY.GDP.PCAP.CD`) and life expectancy (`SP.DYN.LE00.IN`) for all countries for the year 2022.

Drop missing values. Use `stats.pearsonr` and `stats.spearmanr` to compute both correlation coefficients. Create a scatter plot of GDP per capita vs. life expectancy. Why does the Spearman correlation differ from the Pearson correlation? Hint: try plotting GDP on a log scale (`plt.xscale("log")`).

```
# Exercise 8
```

Exercise 9 Using the World Bank API, fetch Germany's total population (`SP.POP.TOTL`) for each year from 1960 to 2023. Use `stats.linregress` to fit a linear trend to the data. Plot the population over time together with the regression line. Compute the residuals and plot them. Does a linear model fit well? What happens around 1990?

```
# Exercise 9
```

Exercise 10 Using the World Bank API, fetch GDP per capita (`NY.GDP.PCAP.CD`) for all countries for the years 2010 and 2020. Merge the two years into a single DataFrame so that each country has columns `gdp_2010` and `gdp_2020`. Drop rows with missing values. Use a paired t-test (`stats.ttest_rel`) to test whether GDP per capita has changed significantly. Then fit a normal distribution to the differences `gdp_2020 - gdp_2010` using `stats.norm.fit` and plot the histogram together with the fitted density.

```
# Exercise 10
```

Exercise 11 Generate 200 samples from a mixture of two normals: with probability 0.4 draw from $\mathcal{N}(-2, 1)$, otherwise from $\mathcal{N}(3, 0.5^2)$. Write a function that computes the negative log-likelihood of a two-component Gaussian mixture (with 5 parameters: $\mu_1, \sigma_1, \mu_2, \sigma_2, p$). Use `scipy.optimize.minimize` to find the MLE. Plot the histogram and the fitted density.

```
# Exercise 11
```

Exercise 12 Using the World Bank API, fetch life expectancy (`SP.DYN.LE00.IN`) and GDP per capita (`NY.GDP.PCAP.CD`) for all countries for the year 2022. Drop missing values. Standardize both variables. Perform PCA on the two-dimensional data. What fraction of the variance is explained by the first principal component? Plot the data in the original and in the PCA coordinate system. Does PCA help here, given that there are only two features?

```
# Exercise 12
```

10 Your project

The third part of this course is a small data analysis project that you carry out in groups of 2. The goal is to apply the skills from chapters 02–09 to a topic of your choice. Use `git` and either GitHub or the university GitLab to share code within your group (see Section 11.8 for an introduction). The repository should be public (or shared with the instructor) so that the instructors can follow your progress. Here is the timeline:

1. **Find a project.** Choose a dataset and a question you want to answer. The dataset can come from any source — public APIs (e.g. World Bank, Eurostat, OpenStreetMap), CSV files from the web, or data you collect yourself. The question should be concrete enough that you can answer it with several plots, tables, and statistical tests.
2. **Hand in your project idea (by Pentecost – Pfingsten – break).** Write a short description (one to two pages are enough) containing:
   - What data will you use, and where does it come from?
   - What question(s) do you want to answer?
3. **Implement and present your project (last two or three sessions).** Build a Jupyter notebook that loads, cleans, analyzes, and visualizes your data. Each project will be presented briefly in class. Your presentation should cover:
   - The question and the data source.
   - The main steps of your analysis.
   - Your results (plots, tables, statistical findings).
   - Any difficulties you encountered and how you solved them.

11 Miscellaneous

We collect here some stuff, which does not really fit anywhere else. We use the material in this section without notice in the whole course.

11.1 Comments

In Python, the key for a comment is #.

```
# This line is a comment.
print("This line is not a comment.")
```

```
This line is not a comment.
```

11.2 Two equality signs

In Python (and many other programming languages), there are assignments such as `x = 0`, which says that `x` should from now on have the value `0`, and `x == 0`. The latter is a condition, which results in `True` if `x` is indeed `0`, and `False` otherwise. There is also the `is` keyword, which checks whether two variables refer to the same object in memory (not just the same value). For example, `x is None` is the recommended way to check for `None`, but for comparing values you should always use `==`.

11.3 User input

Sometimes, when operating on the command line, one wants to ask for user input. This works as follows:

```
name = input("What is your name?")
print(f"Hello, {name}!")
```

However, in the jupyter notebooks we are using in this course, one rather adds input direct to the code blocks.

11.4 Dot-notation

Here is another very useful notation convention, known as dot-notation: For a function `str.something(s,...)`, where `s` is of type `str`, you can as well write `s.something()`. For example, `s.strip()` is the same as `str.strip(s)`.

11.5 Type hints and assertions (`assert`, type annotations)

Python does not enforce that you say which type a variable has (in contrast to C, say), but you can annotate them. This is not enforced by Python, but helps readability.

- `x: int = 5`: tell Python that `x` is an int.
- `def f(x: float) -> float:`: declare that `f` takes a float and returns a float.

```
def square(x: float) -> float:
    """Return x squared."""
    return x * x

print(square(3.0))
# This also works, since Python does not enforce the type hint:
print(square(5))
```

```
9.0
25
```

11.6 Virtual environments and `pip` (`venv`, `pip`)

When working on different Python projects, you may need different versions of the same library. Virtual environments solve this by creating an isolated Python installation per project. The `pip` tool installs packages into the active environment.

- `python3 -m venv venv`: create a virtual environment in the folder `venv`.
- `source venv/bin/activate`: activate the virtual environment (Linux/Mac). On Windows: `venv\Scripts\activate`. When the environment is active, your terminal prompt will be prefixed with `(venv)`, e.g. `(venv) user@host:~$`. You can also verify by running `which python`, which should point to `venv/bin/python` instead of the system Python.
- `deactivate`: leave the virtual environment.
- `pip install package`: install a package into the active environment.
- `pip install -r requirements.txt`: install all packages listed in `requirements.txt`.

- `pip freeze`: list all installed packages and their versions.

```
# typical workflow when starting a new project
python3 -m venv venv
source venv/bin/activate
pip install numpy pandas matplotlib scipy
pip freeze > requirements.txt
```

```
# typical workflow when cloning an existing project
git clone https://github.com/someone/some_project
cd some_project
python3 -m venv venv
source venv/bin/activate
pip install -r requirements.txt
```

Without a virtual environment, installing packages changes your system-wide Python installation, which can break other projects. It is good practice to always work inside a virtual environment.

11.7 Reading error messages and debugging using `print`

When your code raises an error, Python prints a **traceback** — a summary of what went wrong and where. Learning to read tracebacks is an important skills in programming. Here is an example:

```
Traceback (most recent call last):
  File "example.py", line 5, in <module>
    result = divide(10, 0)
  File "example.py", line 2, in divide
    return a / b
ZeroDivisionError: division by zero
```

Read a traceback from **bottom to top**:

1. The **last line** tells you the error type and a short description (`ZeroDivisionError: division by zero`).
2. The lines above show the **call stack**: which functions were called, in which file, and at which line number. The most recent call is at the bottom.
3. The line of code that caused the error is shown directly below each location.

For a list of common error types and how to handle them with `try`/`except`, see Section 3.5.

When the error message alone is not enough to find the bug, you need to **debug** — i.e. inspect what your code is actually doing step by step.

- `print(x)`: the simplest debugging tool. Print the value (and possibly the type) of a variable at a critical point.
- `print(f"{x = }")`: a shorthand (since Python 3.8) that prints both the variable name and its value.
- `type(x)`: check the type of a variable. Many bugs come from unexpected types. For more complex issues, Python has a built-in interactive debugger (`pdb`). Placing `breakpoint()` in your code pauses execution at that point and lets you inspect variables interactively. See the Python debugger documentation for details.

11.8 git and github

`git` is a version control system: it tracks changes to files over time, so you can go back to earlier versions, collaborate with others, and see who changed what and when. `github` is a website that hosts `git` repositories online.

These course notes are organized using `git`. You need to install `git` on your system. Here are the most important `git` commands:

Getting started:

- `git clone url`: download a repository from github to your computer.
- `git pull`: update your local copy with the latest changes from github.
- `git status`: show which files have been changed, added, or deleted since the last commit.
- `git log`: show the history of commits (press `q` to quit).
- `git log --oneline`: same, but one line per commit.

Making changes (for your own projects):

- `git init`: turn the current folder into a new git repository.
- `git add file`: stage a file for the next commit (i.e. mark it to be included).
- `git add .`: stage all changed files.
- `git commit -m "message"`: save the staged changes as a new commit with a description.
- `git diff`: show what has changed since the last commit.
- `git diff --staged`: show what has been staged for the next commit.

Working with github:

- `git push`: upload your commits to github.
- `git remote -v`: show which github repository your local copy is connected to.

Undoing things:

- `git checkout -- file`: discard local changes to a file (revert to last commit).
- `git reset HEAD file`: unstage a file (undo `git add`).

A typical workflow looks like this:

```
# 1. make some changes to files
# 2. check what changed
git status
git diff
# 3. stage and commit
git add file1.py file2.py
git commit -m "add data loading function"
# 4. upload to github
git push
```

The `.gitignore` file lists files and folders that git should ignore (e.g. `venv/`, `__pycache__/`, `.ipynb_checkpoints/`, data files). This prevents large or private files from being tracked.

Getting the course materials:

```
git clone https://github.com/pfaffelh/python_for_data
cd python_for_data
python3 -m venv venv
source venv/bin/activate
pip install -r requirements.txt
```

The course notes will be updated during the semester. To get the latest version, run `git pull`. There is one catch: if you have changed a course file locally, `git pull` will refuse to overwrite your changes. The best way to avoid this is to do all your work in the folder `myFiles` or only change the `.ipynb` files — these are excluded from updates via `.gitignore`.