# Scientific Python

Boris Nerot [1]
LabOratoire proCédés énergIe bâtimEnt

November, 2024
v0.2



[1]boris.nerot@univ-smb.fr

# Contents

# Part I

# General programming knowledge

# Chapter 1

# Computer basics

## 1.1 Hardware

### 1.1.1 Main componenents

#### Memory

**Description** Used to store the data related to running softwares. Can be described by:

1. Capacity (GB): amount of stored data
2. Frequency (MHz) / generation: transfer speed
3. Format: tower/laptop
4. ...


#### Photos

#### CPU

**Description** Performs base operations (sum, division, etc...) using data stored in memory. Can be described by:

1. Number of cores
2. Operating frequency, generation, engraving width, supported instructions

3. *TDP* (W)
4. Cached memory
5. ...

***Multithreading*** Idea: perform many tasks simulatneously on a same physical cores
Chez *Intel*: *Hyper-Threading*.

$$\#\text{logical cores} = \#\text{physical cores} \times \frac{\#\text{threads}}{\#\text{physical core}}$$

**Photos**


CPU

CPU cooler

**Storage**

**Description** Save data on the long term. Can be described by:

1. Capacity (GB)
2. Reading/Writing speed, latency
3. Type
4. ...

**Photos**

**GPU**

**Description** Similar to GPU. Performs graphical operations and produces an image to display
More broadly, performs some highly parallelizable tasks. Can be described by:

1. Memory: capacity (GB) and generation (i.e.: frequency)
2. External connectors

**GPU and graphic card** The word "GPU" describes the computation unit only (not the fans, memory, ...).


HDD


SSD (old technology)


SSD (new technology)

**Integrated GPU** For computers having small graphics needs, the GPU is a small unit dedicated to graphics integrated into the CPU.


**Photos**

### 1.1.2 Typical configurations

| Component | Typical hardware properties | |
|---|---|---|
| | Personnal computer | Shared working station |
| Memory | 8 GB | 64-256 GB |
| CPU | 4-8 logical cores<br>2GHz | 16-128 logical cores<br>2-4 GHz |
| Disk | SSD 500 GB | HDD 10 TB<br>SSD 1 TB |
| GPU | Integrated | 2-8 GB memory |

### 1.1.3 Howto: compare two computers

Before any comparison, first ask yourself about the software you want to run:

- Can it be parallelizable?
- Is it eunning on GPU or CPU?

- Does it need to write or read a lot of data from the disk?

Two computers can be compared using one of these two methods:

- Compare the date they were bought together with the price at that times
- Compare main characteristics:
  1. CPU: number of logical cores
     (if a GPU exists: generation, memory capacity)
  2. CPU: frequency
  3. RAM: capacity

  Some websites host a component comparator; typical result is an averaged score built from differents categorical scores (ex: number of cores for a CPU).

## 1.2 Software

### 1.2.1 Operating systems (*OS*)

An operating system is a software that makes hardware ressources availablethrough interfaces.
We usually make a distinction between low-level component of the OS (e.g.: kernel, handles the hardware) and those that provide applications the user can interact with.

#### Many OS

Main OS are:

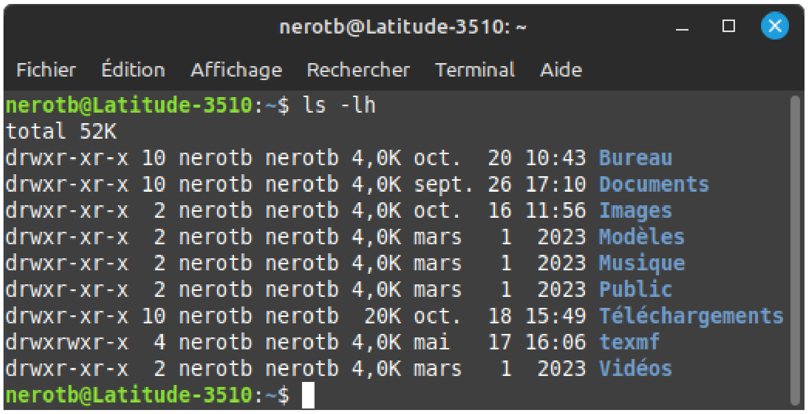
- Windows (Microsoft)
- OS X (Apple)
- Linux

#### GUI and CLI

***GUI*** : *Graphic User Interface*
Interface that describes software components using drawings. One can interact with a GUI mainly using a mouse.

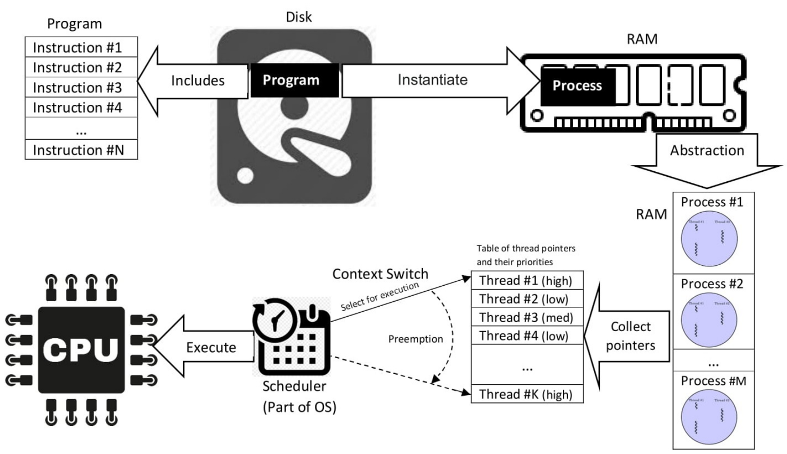
***CLI*** : *Command Line Interface*
Interface that describes software components using text. One can interact with a CLI using a keyboard.

**GUI is built on the top of CLI** The visual aspect rendered by a GUI is often a simplified version of what can be achieved using the CLI. In the background, most of GUI-related actions are translated to CLI commands at run time.




### 1.2.2 Processes

A process is a set of instructions processed by the hardware as asked by a running software. Some processes create several threads that use all the available logical cores. Processes are identified using a *PID* (Process IDentifier).




### 1.2.3 Programing language

> [...] A programming language is a system of notation for writing computer programs. A programming language is described by its syntax (form) and semantics (meaning). It gets its basis from formal languages.
> source: Wikipedia

#### Compiled vs interpretated

**Compiled language** A software written using a compiled language is directly translated into something the OS can handle. Examples of compiled languages: Fortran, C++

**interpretated language** A software written using an interpreted language is split into pieces that make a sense for a master language which handles the execution Examples of interpretaed languages: Python, Matlab

# Chapter 2

# Development basics

## 2.1 Variables types

Programming (often) consists in the definition of functions that handle variables. The type of a variable defines the operations one can perform on this variable. Hence, the choice of each variable type has some important consequences for the entire program. One must think about proper variable types *before* writing any code.

### 2.1.1 Primitive variable types

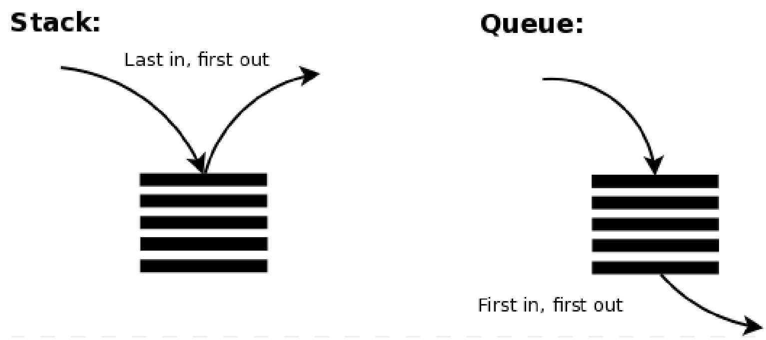
Primitive variable types are types already implemented by the language itself. Most common are:

- Int: integers can be signed (i.e. possibly negative) or unsigned (greater that only). In the second case they are often called UInt, for Unsigned Integer.
- Float: similare to integers, float values exist as different flavours: Float16, Float32, Float64 (sometimes also called Double). A float can store numbers with a decimal part.
- Char: character, a type than can store 1 byte.
- String: sequence of characters used to store text
- Bool: True/False

### 2.1.2 Containers

A container is a data structure that unites different variables for an easy reading and writing access. Most commons are:

- vectors/arrays: index-like access
- Stacks: LIFO access (*Last In First Out*)
- Queues: FIFO access (*First In First Out*)
- Dictionaries: key/values access. Each key is unique.
- set: set of unique values




### 2.1.3 Object oriented programming

#### Definition

Some data types can handle references to different primitive variables in a common data structure. These variables are usually called **attributes**. **Object Oriented Programming** makes possible to define these types using **classes**. Each variable built from a class is called an **object**. It interacts with other variables through **methods**. **Inheritance** makes it possible for such a custom data type (*T2*) to have the properties (attributes, methods) of another one (*T1*).

#### Examples

**First** We can think about an **Official identity** type that stores the following information:

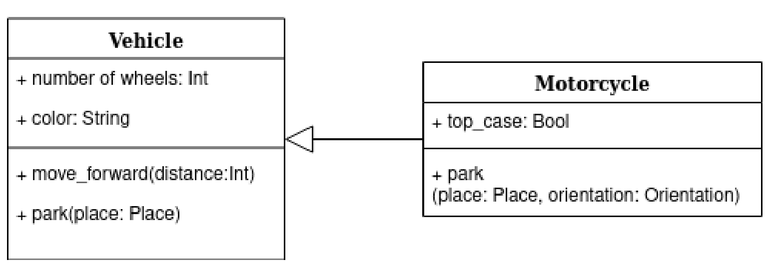
- Unique identifier: *Int*
- Name: *String*
- Height: *Float16*
- Parent(s) and siblings: containers of objects of type *Official identity*

One could also partly define **Official identity** from a type **Identity** (inheritance):

- Name: *String*

In that case, the "Name" attributed would be inherited from the parent object of type **Identity**. Then **Official identity** specifies **Identity**: every property of **Identity** is a property of **Official identity** (but not the other way).

**Second** In case of onheritance, one can **overload** some methods of the parent (see *park* in the example below)



## 2.2 Variable scope

### 2.2.1 Definition

Each language defines different variable scope rules. These rules state whether a variable created in a scope A can be accessed from a scope B. These rules are related to:

- Control flow structures
- Functions
- Local code imports
- ...

These rules makes it possible to use a common variable name in different places in the code, to refer to different variables. **Shadowing** refers to the fact of reusing for a different purpose an already defined variable name

### 2.2.2 Static and dynamic typing

Static typing consists in the manual assignment of a type to a variable. This variable will keep this type until the end of its use. Dynamic typing let the compiler (or interpreter) chose the variable type during run time.
In both cases, the variable has a known type during execution.

## 2.3 Mutability

The mutability of an object refers to the possibility of modifying its properties.

### 2.3.1 Copy and assignment

The assignment of a mutable object to another variable might lead to unexpected results. Indeed:

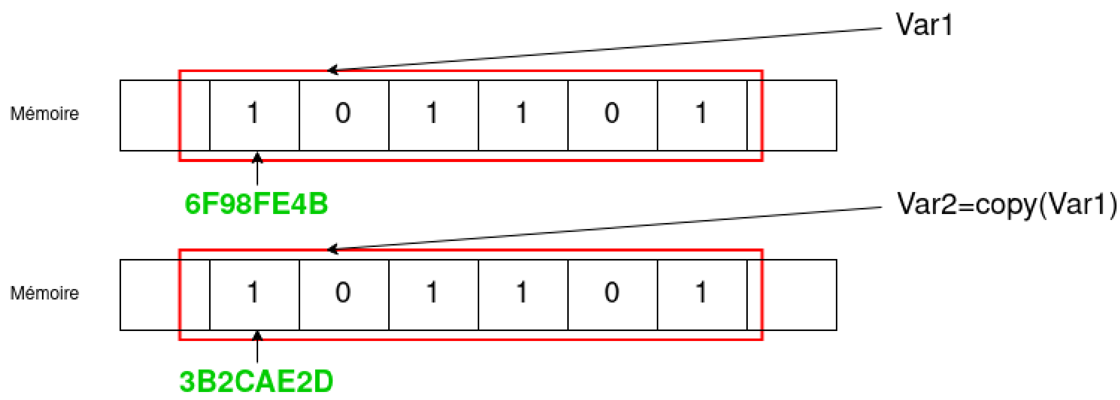
- The in-depth copy of an object makes a full duplicate of this object. (Figure 2.3).
- Conversely, an assignment of a mutable object to a variable makes the variable reference the memory content of the object (Figure 2.4).

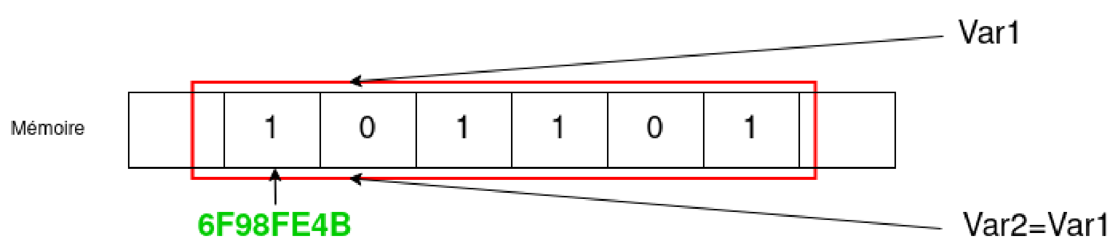
In the first case, a modification on the first variable has no effect on the second. In the second case, if first is modified then second is modified too.
**There exists an intermediate copy level between assignment and in-depth copy**: shallow copy. If 'b' is a shallow copy of 'a', 'b' has a different memory address than 'a' (different objects), yet references of 'b' to mutables objects are the one of 'a'.

### 2.3.2 Equality and identity

Some objects can be equal - i.e. same properties - withouth being the same - i.e. different memory address. The identity test makes it possible to check if 2 objets have the same identity. Using pseudo-code:

```
MyType var1 = MyType.constructor()
MyType var2 = InDepthCopy(var1)
```




```
AreIdentical(var1, var2)  # False
AreEqual(var1, var2)      # True
```

# Chapter 3

# Development steps

## 3.1 Identify your problem

### 3.1.1 Define the scientific goal

*Before* writing any code you must be able to answer the following questions:

- What is my code suppose to do?
  If the code serves multiple puroses, split them in small independant units.
- What are the different ways to achieve this/theses goals?
  Temporal dynamic simulation, mathmatical optimisation, formal mathematical models, etc ...
- How reliable is my input data?
  Another way to put it: to what extend the code errors depend on things I have no control over.
- Is there a way for me to control:
  - the exactness of the results produced by the code
  - the determinism of the code (if any)

### 3.1.2 Choose the language

Various criteria must be taken into account in the choice of the programming language.

#### Open-source aspect

An open-source language is a language for which the content needed to improve or modify the language is available freely. Every one can contribute and use such a language.

These languages benefit from:

- A large community that can provide help
- The tracking of different versions and modifications of the language
- A pretty fast development cycle (time between two releases of the language)

Proprietary languages (in contrast with open-source languages) have the following drawbacks)

- Often very costly
- Code sharing made difficult because one must have bought the right tools
- Sometimles, partial documentation in order to protect an intellectual property

#### Running speed

Some languages are slower than other (sometimes up to 100 times slower). In many cases, slowness is the price to pay for a simpler syntax and fewer code lines. Regarding this running speed:

- It can often be **largely improved** using dedicated libraries.
- It depends on what you intend to do: different languages perfom best in different problems.
- It depends on the compiler/interpreter

Without using dedicated libraries, popular languages can be sorted out according to their typical running speed **approximately**:

1. C/C++
2. Julia
3. Matlab
4. Python
5. R

#### Versatility

| Do you need to ... | Then: search for |
|---|---|
| Read/write files, manipulate strings | simple syntax |
| Interact with existing codes and softwares | dedicated libraries, large community |
| Store and retrieve easily a lot of data into memory | easy type specification, simple syntax |
| Perform a intensive computation | native language speed, dedicated libraries |

#### Understandable by other people

Your code must be understandle by people:

- Who work in the same field
- Whose training is similar to yours

note: using a rare language can prevent people from trying to help you.

### 3.1.3 Define a development plan

Software development is a full time activity. Software development on "spare time" between two other activites (for instance: experimental activites) is error prone. **It is less time consuming to develop in an organize way rather than correct lack-of-attention errors afterwards**.

How to get organized:

- **Evaluate prior to any development** the time need for each development step.
  If you exceed this time, this might mean that you are out of the scope of the step.

| Step | Prior thoughts | Actions | Test |
|---|---|---|---|
| 1. Meet the the primary purpose of the code. | Data structure. | Reading/Declaration of input parameters.<br>Main content of the code.<br>Simplified presentation of results. | Using a test case:<br>Are the results correct?<br>Does the running time seem compatible with all the cases to be treated? |
| 2. Structure the code | Possible interactions with the code.<br>Aspect of results. | Definition of functions/classes/structures<br>Errors handling, progress log. | Resilience to parameters change, easy results exploitation. |
| 3. Optimize the code | I/O importance in the problem, memory independance. | Benchmark<br>Speed up: algorithmic, variable types. | Better ressources use, with same code functionalities. |
| 4. Document the code for your own interest | Sections of the code subjected to change | 1. Short comments for difficult sections<br>2. Short description at the top of the file | Try to understand the code 2 weeks later. |
| 5. Document for other users of the code | Typical use cases of the code | Writing documentation in an iterative process:<br>1. Important content<br>2. Optionnal content<br>(see 3.3.3) | Ask somebody to try your code. |

- **Keep in mind the real aim of the code**
  This makes it possible to distinguish what is useful from what is detail.

## 3.2 Develop

### 3.2.1 Choose an IDE

#### Definition

An *Integrated development environment* is a GUI software that facilitates software development by:

- Communicating with the compiler/interpreter.
- Highlighting some of the development errors before compilation.
- Speeding up code writing using plugins and keyboard shortcuts.

Support for advanced features (version control (*git, svn*), code suggestion) are sometimes available.

#### Examples

Some IDE are language specific, other are multi-languages. Some famous IDE:

- Eclipse
- Netbeans
- Visual Studio
- PyCharm
- Spyder

Les languages fermés (ex: Matlab, VBA) viennent avec leur propre IDE.

#### Notes

Don't lose times on optionnal time consuming tasks! For instance, proper formatting of a file is done automatically by the IDE.

### 3.2.2 Organize your working space

A working space is made of:
- Source files and compiled files
- Input data: in your case, all the content with a scientific meaning, used by the program
- Data produced by the program
- Documentation: automatically generated files (html, texte, tex, ...)
- Configuration files: every fotware-related parameters needed for execution on your computer

Some important rules:
- Store all the source files together. Do not duplicate them.
- Separate data from source.
- Avoid file paths with accentuation and special characters.

```
├── data
│   ├── input
│   │   ├── in_1.csv
│   │   └── in_2.csv
│   └── output
│       ├── out_1.png
│       ├── out_2.png
│       ├── out_3.txt
│       └── out_4.txt
├── doc
├── LICENSE
├── README
└── src
    ├── dir1
    │   ├── file1
    │   └── file2
    ├── dir2
    │   ├── file3
    │   └── file4
    └── main
```

Suggestion of working space

### 3.2.3 Universal development rules

Each language has some specific mandatory rules. Yet there exists come common good practices:

#### Language

Code must be written in English language since:

- The syntax of the language is English.
- Scientific community communicates using English.
- Development problems are described in English on dedicated forums.

#### Naming conventions

Some rules must be respected:
- In most of the languages, do not use any accentuation or special character.
- CamelCase or snake_case
  Same everywhere in the code.

camelCase

snake_case

nocase

#### Variables

In the common case, a variable name must be:

- Short (≤ 25 characters)
- Explicit
- Without any references to its type
- Using plural form if it's a container-like

Global variables are written in UPPER_SNAKE_CASE.

| Incorrect | Correct |
|---|---|
| myvar | dyn_viscosity |
| a_very_long_var | input_paths |
| temporaryArraySTRING | initialConditions |

In a software development process, an explicit variable name tells the user about the physical (or mathematical, etc ...) meaning of the variable. Each wisely-defined variable name represents a lot of saved time when using the code as it is understood with less effort. lit le code.

**Functions and methods**
The name of a function:

- **Starts with a verb** à l'impératif at the imperativ form.
- Does not mention the type of input or output data.

**Indentation and spacing**

Some languages have few constraints regarding indentation and spacing. In these cases:

- Do not exceed 100 characters as a length of code lines.
- Indent the same way instructions blocs that share the same structure, or let the IDE do it for you.

**Nested code**

Nested code is difficult to read. For instance, do not nest multiple calls to functions in a one-liner as follows:

```
Float32 solar_power = compute_solar_power(get_area(solar_panel),
            get_irradiance(
                        download_weather(get_actual_time())))
```

Instead, write as many lines as needed:

```
Float32 solar_area = get_area(solar_panel)
                     Time current_time = get_actual_time()
WeatherData Geneve_weather = download_weather(current_time)
Float32 Geneve_irradiance = get_irradiance(Geneve_weather)
Float32 solar_power = compute_solar_power(solar_area, Geneve_irradiance)
delete(solar_area, Geneve_weather, Geneve_irradiance)               # optional in most la
```

**Comments**

Comments start with a special character or set of characters that is defined for each language. Yet, comment characters must not be inserted manually as:

- It is less readable (unwanted spaces at wrong places)
- It creates indentations error whenever you remove manually this comment sign
- Automatic removal by the IDE might not be possible

Instead, you must use the IDE shortcut for defining either line comments or block comments.

## 3.3 Comment and documentation

### 3.3.1 Differences

A comment is a short explanation in the code that helps any **developer** to understand a few instructions. A comment may be only temporary. A commented code is mush simpler to understand.

A documentation is an exhaustive explanation for the **user** of the code. The documented parts of the code are the one that constitute its API.

### 3.3.2 Comment a code

A comment must be inserted at least in the following cases:

1. Bug to be corrected
2. Local code improvement needed
3. The way a difficult bug was solved Exemple:

   ```
   UInt64 simulation_timestep = get_timestep()  // UInt64 prevents overflow
   ```

4. Unusual instructions, ye tneeded. Exemple:

   ```
   parse_request(parser=Parser.NO_DEFAULT, timeout=NULL) // default timeout (3 min)
                                                         // is too short
   ```

5. Reference to a scientific article or an important scientific methodology or result Exemple:

   ```
   Array[Float32] fourier_coefs = adapted_fourier_transform() //cf DOI 04.52/j.fake.206.1
   ```

   This type of comments does not intend to replace an exhaustive scientific description of the code in a dedicated document.

6. Some parameters are specified or a section of code is made unavailable (i.e.: "commented out") Exemple:

   ```
   Int8 thermal_diffusivity = 9 // get_thermal_properties() has to be fixed
   ```

Each IDE comes with specific keywords to make comments more readable. It is often the case of **fixme** (see 1) and **todo** (see 2).

### 3.3.3 Document a code

#### Content of a documentation

The documentation section of a function must give some information about the following points, from most to less important:

1. What the function does
2. Input parameters
   - types
   - what they describe
   - the set of expected values, if any
   - the default value, if any
   - physical unit, if any
3. Same for output parameters
4. Related functions
5. Detail of the operations achieved by the function
6. Complete scientific references
7. Use case examples of the function

**Points 1 to 3 are mandatory**. The other are optional yet recommended, in particular 7.
Note that writting the documentation of a code is an iterative process: going through the entire code reveals some bug to fix, before going further into the documentation.

#### Documentation: howto

**Documentation writing** The documentation is written into the code. It is often located:

- Either at the line before the function definition
- Or in the function body

A particular syntax is adoptéed to reference a component of the code (variable, function, ...) or the language (types, ...).

**Documentation generation** Each documentation block is then rendered in an easy-to-read format:

- HTML: if documentation is hosted on the web `https://about.readthedocs.com/`
- Latex/PDF: if documentation is shared locally

The built document is called the "API reference" because it makes it possible to entirely use the software according to its API. Some famous documentation builders are:

- Javadoc (Java)
- Doxygen (C, C++)

- Sphinx (Python)

```python
def norm(x, ord=None, axis=None, keepdims=False):
    """
    Matrix or vector norm.

    This function is able to return one of eight different matrix norms,
    or one of an infinite number of vector norms (described below), depending
    on the value of the ``ord`` parameter.

    Parameters
    ----------
    x : array_like
        Input array.  If `axis` is None, `x` must be 1-D or 2-D, unless `ord`
        is None. If both `axis` and `ord` are None, the 2-norm of
        ``x.ravel`` will be returned.
    ord : {non-zero int, inf, -inf, 'fro', 'nuc'}, optional
        Order of the norm (see table under ``Notes``). inf means numpy's
        `inf` object. The default is None.
```

`linalg.norm(x, ord=None, axis=None, keepdims=False)` [source]

Matrix or vector norm.

This function is able to return one of eight different matrix norms, or one of an infinite number of vector norms (described below), depending on the value of the `ord` parameter.

**Parameters:** **x** : *array_like*

Input array. If *axis* is None, *x* must be 1-D or 2-D, unless *ord* is None. If both *axis* and *ord* are None, the 2-norm of `x.ravel` will be returned.

**ord** : *{non-zero int, inf, -inf, 'fro', 'nuc'}, optional*

Order of the norm (see table under `Notes`). inf means numpy's `inf` object. The default is None.

Other than the API reference, the user can find on the web some special documentations to learn a programming language or a specific library:

- Getting started : suggestions of code and important concepts that cover most of development needs
- Tutorials: achieve very specific things with the tool you learn

## 3.4 Example

This part shows the different development steps applied to a fictive math problem (parts 3.1, 3.2, 3.3) Let's solve the following differential equation:

$$(E) \quad \ddot{\theta} + w_0^2 \sin\theta = 0 \tag{3.1}$$

Given:

- $\theta$ a time dependant function, with $t \in [0, 10]$
- initial conditions:

$$\theta_0 = \theta(0) = \frac{\pi}{8} \tag{3.2}$$

$$\theta_0 = \theta'(0) = 0 \tag{3.3}$$

- $w_0^2$ takes one of the following values:

$$w_0^2 \in \{0.05, 0.5, 5, 50, 500\} \tag{3.4}$$

### 3.4.1 Identify one's problem

**Define the scientific goal**

Our code must be able to:

1. Read a parameter file
2. Solve a differential equation using the parameters specified by the file
3. Plot and save a diagram presenting the solution to the equation

A discretization on $t$ makes it possible to get a solution close to the formal one. Conversely, a formal resolution would be difficult with the sin.

The input data ($w_0^2$) is reliable since these are independant parameters of typical amplitude regarding this type of equations.

The exactness of the result will be evaluated usng the approximation for small $\theta_0$ values:

$$\theta : x \to \theta_0 \cos(w_0 x)$$

This problem is deterministic, and we shall check we get the same solution from one run to another either graphically or numerically.

**Choose the language**

This problem does not represent a heavy CPU task. Yet, discretization and numerically stable solving both require a dedicated library. *Julia* language is adaptated to analytic maths problems, using dedicated libraries such as *SciML* (solving) and *Plots* (plotting). This language is a high-level language, hence dealing with data input/output will be easy.

#### Define a development plan

Table 3.3 is a suggestion of development plan.
note: Parameters reading from the disk is not prioritary and could take place at the second step only.

| Step | Actions | Duration |
|---|---|---|
| 1. meet the the primary purpose of the code. | read and store parameters values<br>Define and solve the equation.<br>Perform aquick draft plot of the solution. | 1 hour |
| 2. Structure the code | Build an API.<br>Display some progress information for the user.<br>Customizz and save the results plot. | 1 hour |
| 3. Optimize the code | Profile and analyse the running time.<br>Evaluate potential time savings.<br>Speed up the code. | 30 min |
| 4. Document the code for your own interest | Description of every difficult instruction. | 10 min |
| 5. Document the code for other users of the code | Write documentation block of each function.<br>Build as html. | 30 min |

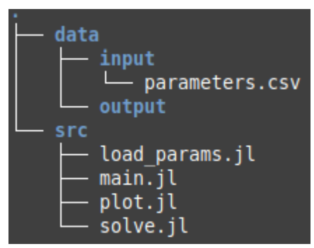
### 3.4.2 Develop

#### Preparation

**Choose an IDE** Regarding Julia language, the largest community IDE is Visual Studio Code with the dedicated plugin.

#### Organize your working space

For instance, let's create 3 source files, one for each basic functionality. All of them are supervised by the file *main.jl*.
Yet, at step 1, (see Table 3.3), all of the code is in *main.jl*. Code structuration befins at step 2.




Actual structure of the working dir.

### Development

**Step 1: meet the the primary purpose** Equation 3.1 is transformed into 2 equations of first order:

$$\dot{\theta}(t) = \omega(t)$$
$$\dot{\omega}(t) = -w_0^2\theta(t)$$

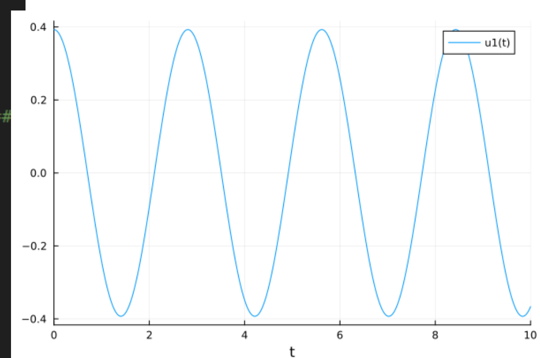
Let's call $u$ the result vector:

$$u = \begin{bmatrix} \theta \\ \omega \end{bmatrix}$$

Below, a first version of the code for step 1.

```
src > main.jl > ...
#= reading parameters
helper doc: https://csv.juliadata.org/stable/reading.html#CSV.read =#
using CSV
using DataFrames
parameters_path = joinpath(pwd(), "data", "input", "parameters.csv")

all_params = CSV.read(parameters_path, DataFrame, header=false)
all_params = all_params[!, "Column1"]


#= solving equation for one value of w_0_2
helper doc: https://docs.sciml.ai/DiffEqDocs/stable/types/ode_types/ =#
using DifferentialEquations
using Plots

w_0_2 = all_params[1]

function eq!(du, u, p, t)
    du[1] = u[2]
    du[2] = - p * u[1]
end
u0 = [π/8, 0]
tspan = (0.0, 10)
prob = ODEProblem(eq!, u0, tspan, w_0_2)

sol = solve(prob, Tsit5())


#= Plotting result =#
plot(sol, vars=1)
```

Code - step 1




$\theta$ as a function of time - step 1

**Step 2: structure the code** The following taks are achieved:

- Split up of the code in dedicated files.
- Use of functions
- Check of parameters compliance
- Display of log messages
- Custom of the plot, disk save

See figures 3.2 and 3.3.

```
src > main.jl > ...
include("load_params.jl")
include("solve.jl")
include("plot.jl")


all_params = load_from_disk("parameters.csv")

results = Dict()
println()
for w_0_2 in all_params
    sol = solve_inplace(w_0_2)
    results[w_0_2] = sol
end

println("")
plot_custom(results, "first_parameters_set.png");
```

*main.jl*

```
src > load_params.jl > ...
#= reading parameters
helper doc: https://csv.juliadata.org/stable/reading.html#CSV.read =#
using CSV
using DataFrames

function load_from_disk(file_name::String)
    @info "Loading file" file_name
    parameters_path = joinpath(pwd(), "data", "input", file_name)
    all_params = CSV.read(parameters_path, DataFrame, header=false)
    if isempty(all_params)
        error("\nBad parameters: no values")
    end
    all_params = all_params[!, "Column1"]
    if typeof(all_params)!=Vector{Float64}
        error("\nBad parameters: type error, expected floats")
    end
    all_params
end
```

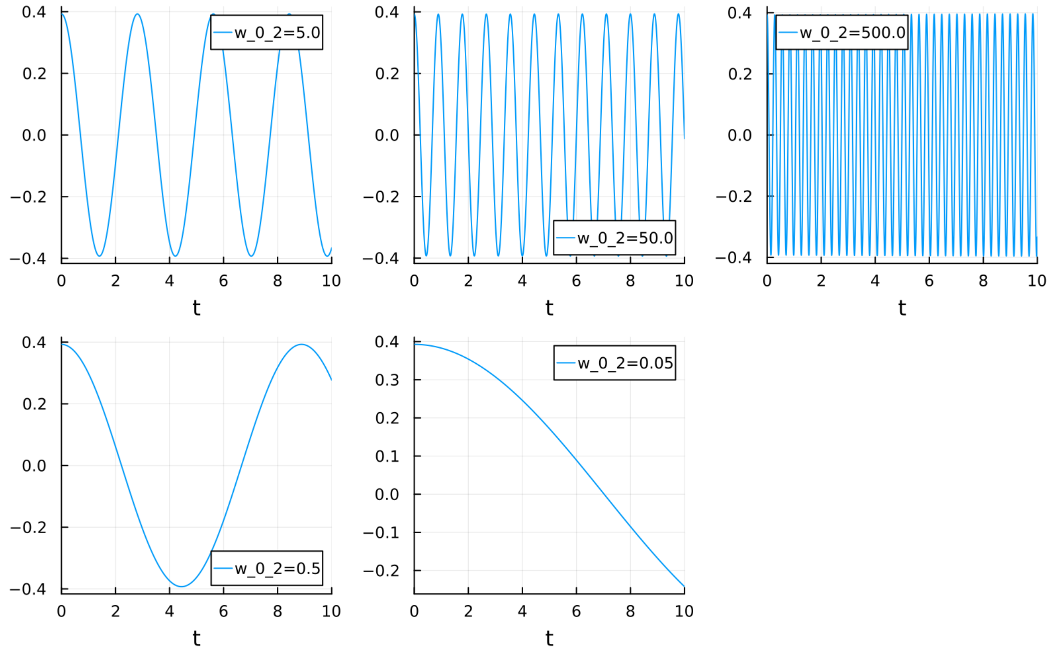
*load_params.jl*

```
src > solve.jl > ...
#= solving equation
helper doc: https://docs.sciml.ai/DiffEqDocs/
            stable/types/ode_types/ =#

using DifferentialEquations
using Plots


function solve_inplace(w_0_2::Float64)
    function eq!(du, u, p, t)
        du[1] = u[2]
        du[2] = - p * u[1]
    end
    u0 = [π/8, 0]
    tspan = (0.0, 10)
    @info "Solving problem" w_0_2
    prob = ODEProblem(eq!, u0, tspan, w_0_2)
    sol = solve(prob, Tsit5())
    return sol
end
```

*solve.jl*

```
src > plot.jl > ...
#= Plotting result =#
using Plots

function plot_custom(results, plot_name)
    @info "Building figures"
    results = collect(results)
    nbr_solutions = length(results)
    plots = []
    for (w_0_2, sol) in results
        pl = plot(sol, vars=1, label="w_0_2=$w_0_2", linewidth=1)
        push!(plots, pl)
    end
    nbr_cols = div(nbr_solutions, 2) + 1
    plot(plots..., layout=(2, nbr_cols), size=(800, 500), dpi=200)
    save_path = joinpath(pwd(), "data", "output", plot_name)
    savefig(save_path)
end
```

*plot.jl*




**Step 3: optimize the code**

**Measure the running time: overview**
Every language, IDE and operating system comes with **benchmarking and profiling tools**. For this problem, the macro *@benchmark* is used. it returns a total execution duration and an estimation of memory usage.

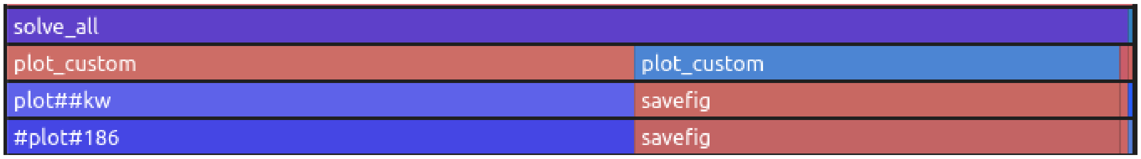
```
BenchmarkTools.Trial: 15 samples with 1 evaluation.
 Range (min … max):  315.696 ms … 405.419 ms  ┊ GC (min … max): 0.00% … 0.00%
 Time  (median):     352.485 ms               ┊ GC (median):    0.00%
 Time  (mean ± σ):   355.361 ms ±  29.252 ms  ┊ GC (mean ± σ):  0.00% ± 0.00%

 316 ms           Histogram: frequency by time          405 ms <

 Memory estimate: 13.92 MiB, allocs estimate: 351244.
```

Results:

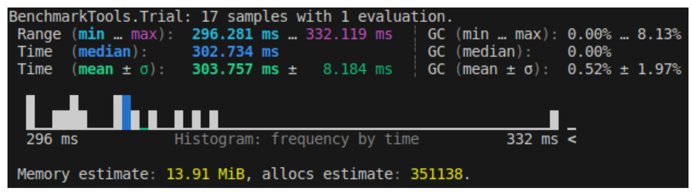
1. On average, the code runs takes 0.36s to run and uses 14 MiB of memory (*MiB=mebibyte*)
   → **Is-it too much?**

2. The standard deviation of running time is low.

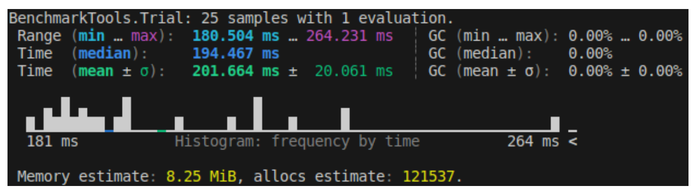
**Profile the running time**
The macro *@profile* makes it possible to focus on time consuming sections of the code. Figure 3.4 shows a *FlameGraph* of this analysis.




Most of the running time comes from plotting and saving the results. This is an important part of the code that must be sped up. Two choices:




1. Specify some variable types to speed up code interpretation.
   - This is specific to Julia language
   - Saved time: 15%.
   - **does not alter functionalities of the code**.
2. Do not save the plot on disk.
   - Saved time: 30%.
   - **is a degradation of functionalities**.




**Step 4: document the code for your own interest** Will be seen on a Python case.

**Step 5: document the code for other users of the code** Will be seen on a Python case.

# Part II

# Best of Python

# Chapter 4

# Notebooks

## 4.1 Jupyter code [easy]

### 4.1.1 Presentation

A **notebook** is a sort of interactive Python console where the code is contained within **cells**. each celle can be ran alone, but the memory (i.e. variables) is shared by all cells.

Output of each cell is printed below the cell. By default, the last evaluated statement of the cell is an output.

There are several pros of using **notebooks** at a development step:

- split heavy computation into small units that make sense
- access easily to variable values
- describe the scientific flow of your program using *markdown* (discussed hereafter)

### 4.1.2 Howto

[ ]:
```
area = 100
speed = 30

print(area, speed)
```

[6]:
```
flowrate = area * speed
print(flowrate)
```

```
3000
```

Above: `flowrate` is displayed as `print` is used.

Below: `flowrate` is returned, and thus displayed, as it is the last statement of the cell

[7]:
```
flowrate
```

[7]: 3000

One can run nearly every Python code in a **notebook**:

- in script mode: global variables declared directly in any cell
- in function mode: functions are defined and called later on
- in object oriented mode: class and methods are defined

[8]:
```
def compute_flowrate(area, speed):
    return area * speed
compute_flowrate(1000, 30)
```

[8]:
```
30000
```

[9]:
```
class Flowrate():
    def __init__(self, area, speed):
        self._area = area
        self._speed = speed
        self.compute()

    def compute(self):
        self._flowrate =  self._area * self._speed

    def __repr__(self):
        return f"Instance 'Flowrate' with value: {self._flowrate:d}"

Flowrate(100, 30)
```

[9]:
```
Instance 'Flowrate' with value: 3000
```

Note: during the execution of a cell:

- the cell itself shows an asterisk **[*]**
- the black circle on the right hand side of the page is colored with black

### 4.1.3 Configuration

#### Files

**notebooks** files have an extension *.ipynb*: these files cannot be manually read (JSON format) and cannot be ran the same way a Python file (*.py*) is ran. Yet, most IDE have a support plugin for these files.

#### Kernel

The *Python* version used by the **notebook** depends either on the environment it is running in or on the configuration of the system. If several Python versions are available, one can choose which to use in the notebook by going to Kernel > Change kernel. The *nb_ conda* plugins facilitates the discovery of existing *conda* environments.

Other **kernels** can be installed to code in other languages (full list).

Some helper functionalitiesin the development process are not natively available in a **notebook**:

Some of them are available using *plugins*.

## 4.2 Jupyter markdown [medium]

### 4.2.1 Introduction

A notebook cell can also contained text formatted in **markdown**. **markdown** is a language that makes it easy to structure a text. **markdown** has fewer features than *html* or *Latex* yet it is very adapted to a scientific context.

*markdown* benefits from a large community. A documentation lies here.

### 4.2.2 Main functionalities

A new line is inserted only if a blank line is added: line 1

line 2

A new line is inserted only if a blank line is added: line 1

line 2

**bold** and *italic*

Same with: **bold** and *italic*

**bold** and *italic*

Same with: **bold** and *italic*

Items list:

- item 1
- item 2

Items list:

- item 1
- item 2

[ ]:
```
Numbered list:

1. item 1
2. item 2
3. item 3
```

Numbered list:

1. item 1
2. item 2

3. item 3

[ ]:
```
titles:
# Level 1
## Level 2
### Level 3
```

```
#### Etc...
```

titles:

# Level 1

## Level 2

### Level 3

#### Etc...

[ ]:
```
URL: [search engine](www.google.fr)
```

URL: search engine

[ ]:
```
Image:  ![some elephants](figures/elephants.png)
```

Image:

[ ]:
```
Reference to a software component, for instance the `matplotlib` library.
```

Reference to a software component, for instance the `matplotlib` library.

[ ]:
```
Mathematical formulas are (mainly) written using the Latex commands :

- In-line mode: $a_{3,4}=\sum_{j}{b^{j}_{3}\times c^{j}_{4}}$

- Block mode:
  $$ a_{3,4}=\sum_{j}{b^{j}_{3}\times c^{j}_{4}} $$
```

Mathematical formulas are (mainly) written using the Latex commands :

- In-line mode: $a_{3,4}=\sum_{j}b^{j}_{3}\times c^{j}_{4}$
- Block mode:

$$a_{3,4}=\sum_{j}b^{j}_{3}\times c^{j}_{4}$$

### 4.2.3 Make the best of markdown

One can combine in the same **notebook** some cells of **markdown** and some cells of code. In a ascientific approach, it is useful to give short explanations regarding what is computed.

For instance:

"[...] after the fit I compute the quadratic error:

$$\epsilon = \sum_i (\hat{y}_i - \bar{y}_i)^2$$

"

[1]:
```
from numpy import sum, array

def sum_square(y_predicted, y_mean):
    return sum((y_predicted - y_mean)**2)

y_predicted = array([1,2,3])
sum_square(y_predicted, 0.5)
```

[1]: 8.75

**One can easily convert a notebook into a Latex or PDF file. This is very handy to produce a scientific report where code has a major importance.**

### 4.2.4 See also

**markdown** is one out of many languages of a similar type: the **markup languages**.

An interesting library is `pandoc` (doc): it converts content from a markup language to another.

# Chapter 5

# Python basics

## 5.1 Control flow [easy]

### 5.1.1 while loop

Code runs while a condition holds True.

[1]:
```
i = 10
j = 0
while (j<5) or (i>6):
    print(f"i={i:<2}, j={j}")
    j += 1
    i -= 1 # i = i -1
```

```
i=10, j=0
i=9 , j=1
i=8 , j=2
i=7 , j=3
i=6 , j=4
```

note: beware of `while` loops that never come to an end!

Operators `any` and `all` are used to process iterables of boolean values:

- `any` returns `True` when at least one value is `True`
- `all` returns `True` when all values are `True`

note: if `all` returns `True` then `any` returns also `True`.

[2]:
```
data = [1, 2, 3, 4, 5]
data_cond = [e > 3 for e in data]
print(any(data_cond))
print(all(data_cond))
```

```
True
False
```

[3]:
```
data = [1, 2, 3, 4, 5]
data_cond = [e > 0 for e in data]
print(any(data_cond))
print(all(data_cond))
```

```
True
True
```

### 5.1.2 `for` loop

`for` makes it possible to go through the values of any **iterable** object.

#### Lists

[4]:
```
for k in [0, 1, 2, 3, 4]:
    print(k)
```

```
0
1
2
3
4
```

#### Generators-like

[5]:
```
for k in range(5):
    print(k)
```

```
0
1
2
3
4
```

[6]:
```
gen = (k//2 for k in range(0, 10, 2))
for k in gen:
    print(k)
```

```
0
1
2
3
4
```

#### Dictionaries

Iteration is done on the keys of the dictionary:

[7]:
```
dic = {"key1": "value1", "key2": "value2"}
for k in dic:
```

```
    print(k)
```

```
key1
key2
```

If both keys and values are needed, one must use the `items()` method:

[8]:
```
dic = {"key1": "value1", "key2": "value2"}
for k, v in dic.items():
    print(k, v)
```

```
key1 value1
key2 value2
```

#### Indexes and values

The `enumerate` function applies to any object that can be iterated over. It returns both the index, and the value. In Python, **the first index is always 0**.

[9]:
```
for idx, k in enumerate(["A", "B", "C"]):
    print(idx, k)
```

```
0 A
1 B
2 C
```

#### Group using `zip`

`zip` makes tuples by extracting an element from each iterable it is given, as long as at least one iterable has no more elements.

[2]:
```
iter_1 = [1,2,3,4]      # longueur: 4
iter_2 = "abcdefgh"     # longueur: 8
for i, j in zip(iter_1, iter_2):
    print(i, j)
```

```
1 a
2 b
3 c
4 d
```

#### `break`

Get out of a control flow structure:

[11]:
```
for k in range(5):
    print(f"k={k}")
    if k == 3:
        break
```

```
k=0
k=1
k=2
k=3
```

If multiple control flow structure are ensted, `break` only exits the deepest level (innermost):

[3]:
```python
j = 0
while (j < 5):
    print(f"\nj={j}", end="")
    for k in range(5):
        print(f", k={k}", end="")
        if k == 3:
            break
    j += 1
```

```
j=0, k=0, k=1, k=2, k=3
```

#### `continue`

Interrupt current iteration and go the next one.

[4]:
```python
j = 0
while j < 5:
    j += 1
    if (j%3 == 0):
        continue
    print(f"j={j}")
```

```
j=1
j=2
j=4
j=5
```

#### `else` statement

The content of `else` is ran only and only if the previously ran control flow structure never met a `break`.

[14]:
```python
for k in range(5):
    if k > 15:
        break
else:
    print("Values lower than 15")
```

```
Values lower than 15
```

### 5.1.3 `if` conditions

Keywords are:

- `if`: test whether a condition holds `True`, independantly from previous tests
- `elif`: (optional) test whether a condition holds `True` if and only if previous `if` and `elif` did not hold `True`
- `else`: (optional) code that mus

[15]:

```
word = "abracadra"
if "x" in word:
    print("'x' in word")
if "a" in word:
    print("'a' in word")
if "y" in word:
    print("'y' in word")
elif len(word) < 5:
    print("Small word")
else:
    print("Fallback to 'else'")
```

```
'a' in word
Fallback to 'else'
```

## 5.2 Print formatting [easy]

### 5.2.1 Introduction

There exists two ways to print a variable content:

1. Use it as an argument of `print`. `var = 5      print("Var: ", var)` Then, the `__print__` method of instances is called.
2. Insert it in a string with special formatting option, and print this string.

### 5.2.2 Method 1: no formatting

This method is fully compliant with the **unpacking** technic:

[4]:
```python
data = "abcdefg"
print(*data)
print(*data, sep="/")
```

```
a b c d e f g
a/b/c/d/e/f/g
```

[2]:
```python
data = [1, 2, 3, 4]
print(*data)
print(*data, end="/")
```

```
1 2 3 4
1 2 3 4/
hello
```

### 5.2.3 Method 2: advanced formatting

Advanced formatting is interesting in a scientific approach because it presents the variable value according to its type. One can see this tutorial for specific use cases. Most of them are covered hereafter.

#### Key idea

Variable name is enclosed in curly brackets `{}` and a prefix `f` is added in front of the string.

[3]:
```python
var = 35.123456
sentence = f"Value of 'var' is {var}"
print(sentence)
```

```
Value of 'var' is {var}
```

Symbols `<`, `>` and `^` produces text-alignment: left, right and center. If any character precedes one of these symbols, the character is used in a **padding way**.

#### Floats

If the variable is to be printed as a float, an additional formatting using the 'f' letter is used inside the curly brackets. One can specify:

- Number of decimal figures
- Minimal number of caracters

[5]:
```
var = 35.123456
print(f"{var:<9.2f}")
print(f"{var:_<9.2f}")    # padding
print(f"{var:_>9.2f}")    # padding
print(f"{var:_^9.2f}")    # padding
```

```
35.12
35.12____
SSSS35.12
__35.12__
```

#### Scientific notation

Similar to float values representation, yet with letter 'e':

[8]:
```
var = 35.123456
print(f"{var:_>9.2e}")
```

```
_3.51e+01
```

#### Percentages

Percentage mode involves the symbol '%': what is printed is the product of the variable by 100.

[9]:
```
var = 0.35123456
print(f"{var:_>9.2%}")
```

```
___35.12%
```

#### Int

The total number of characters is specified.

[10]:
```
var = 12356
print(f"{var:0>9}")
```

```
000012356
```

Beware! For Python an integer that ends with `.` is a float!

#### Other ways to specify arguments

#### Positional arguments

[11]:
```
data = range(5)
print("third value = {2}".format(*data))
```

```
third value = 2
```

#### Named arguments

[12]:
```
data = {"key1": 0, "key2": 1}
print("'key1' = {key1}".format(**data))
```

```
'key1' = 0
```

**Advanced: call `str` or `repr`**

By default, the `__format__` method of instances is called. One can change to use:

- `str`: `{var!s}` (overload of `__str__`)
- `repr`: `{var!r}` (overload of `__repr__`)

[13]:
```
class Fake(float):

    def __repr__(self):
        return "This is my Float (repr)\n"

    def __str__(self):
        return "This is my Float (str)\n"

    def __format__(self, *args, **kwargs):
        return super().__format__(f"{123456.32145:2.3f}")

var = Fake()
print(f"This is formatted: {var}")
print(f"This is printed: {var!s}")
print(f"This is represented: {var!r}")
```

```
This is formatted:
0.0
This is printed: This is my Float (str)

This is represented: This is my Float (repr)
```

## 5.3 Variable types [easy]

### 5.3.1 `list`

`list` are the most common data containers. They are:

- **mutable**: one can change their content by adding new elements or changing existing elements
- **indexable**: one can access the content of a list using an index, starting from **0**

All the operations done on a list are done **in place**. That means no other list is returned but the current instance is modified.

#### Appending content

**1 at a time at the end of the list**

[1]:
```
var = [34, 23]
var.append(1)
var.append(2)
var.append("b")
print(var)
```

```
[34, 23, 1, 2, 'b']
```

**1 at a time by position**

[ ]:
```
var.insert(3, "new")
print(var)
```

**Already stored in a list**

[3]:
```
var1 = [1, 2]
var2 = [3, 4]
var1.extend(var2)
print(var1)
```

```
[1, 2, 3, 4]
```

#### Modifying content

[4]:
```
var[0] = 3
print(var)
```

```
[3, 23, 1, 'new', 2, 'b']
```

One can specify a **negative index**: -1 is the latest value of the list, -2 is the second to last, etc…

[5]:
```
var = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
print(var[-1])
print(var[-3])
```

```
10
8
```

Note than you can go from negative to positive index using this simple trick:

[6]:
```
positive_index  = 3
negative_index  = -(len(var)-positive_index)
positive_index2 =  len(var)+negative_index
print(var[positive_index])
print(var[negative_index])
print(var[positive_index2])
```

```
3
3
3
```

The `index` method returns the index (positive) of the first occurence of an element. A lower and upper index can also be specified to look for a value at a particular location.

[7]:
```
var.append(1)
print(var)
print(f"Index of first '1' value starting from index 0: {var.index(1)}")
print(f"Index of first '1' value starting from index 0: {var.index(1, 4,␣
 ↪len(var))}")
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 1]
Index of first '1' value starting from index 0: 1
Index of first '1' value starting from index 0: 11
```

#### Deleting content

**At the end of the list**

[8]:
```
last = var.pop()
print(f"Last value: {last}")
print(var)
```

```
Last value: 1
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

**Positionnal**

[9]:
```
var.remove(1)
print(var)
```

```
[0, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

#### Sorting

**With homogeneous data types** Beware that sorting the variable is done in place.

[10]:
```
var = [1, -6.7, 189]
var.sort(reverse=True)
print(var)
```

```
[189, 1, -6.7]
```

**Using a custom sorting key** A custom sorting key is a function that returns, for all elements of the list to sort, a value that can be compared to the other.

[11]:
```
var = [56, "98", -102, "102.45"]
var.sort(key=lambda key:float(key))
print(var)
```

```
[-102, 56, '98', '102.45']
```

**Advanced** Sorting is done internally by a call to methods `__lt__` and `__gt__` of an instance. One can redefine these methods to sort elements on some relevant properties.

Below is an example of a custom object who internal value (used for the sorting process) depends on the order of creation of the instance (see `COUNTER`). The `__lt__` method is redefined using this value.

[12]:
```
class Custom():
    COUNTER = 10
    def __init__(self):
        self.value = Custom.COUNTER
        Custom.COUNTER -= 1

    def __lt__(self, other):
        if not isinstance(other, Custom):
            raise NotImplementedError()
        return self.value < other.value

    def __repr__(self):
        return f"Custom ({self.value})"

var1 = Custom()
var2 = Custom()
var3 = Custom()
var = [var1, var2, var3]
print(var)
var.sort()
print(var)
```

```
[Custom (10), Custom (9), Custom (8)]
[Custom (8), Custom (9), Custom (10)]
```

Another way to sort data containers is the `sorted` function. Differently from method `.sort()`, it returns a new sorted list.

[13]:
```
var = (3, 7, 2, 9, -4)
sorted(var)
```

[13]:
```
[-4, 2, 3, 7, 9]
```

#### Concatenation

Two lists can be concatenated using `+`.

[2]:
```
var1 = [1, 2]
var2 = [3, 4]

print(var1)
print(var2)
print(var1 + var2)
```

```
[1, 2]
[3, 4]
[1, 2, 3, 4]
```

#### Conclusion

`list` are commonly used in Python. Yet they are neither always fitted to all use cases, nor the the most powerful solution.

### 5.3.2 `set`

`set` are data containers that can store a given value at most one time. They are not **mutable** and not **indexable**. Thus, **there is no guarantee for the insertion order to be preserved.**

[15]:
```
var = {1, 2, 3}
var.add(4)
print(var)
var.add(3)
print(var)
```

```
{1, 2, 3, 4}
{1, 2, 3, 4}
```

One can perform unions, intersections and differences of `set` instances.

[16]:
```
var1 = {1, 2, 3}
var2 = {2, 3, 4}
print(f"{'Union':<15}: {var1&var2}")
print(f"{'Intersection':<15}: {var1|var2}")
print(f"{'Difference 1':<15}: {var1-var2}")
print(f"{'Difference 2':<15}: {var2-var1}")
```

```
Union          : {2, 3}
Intersection   : {1, 2, 3, 4}
Difference 1   : {1}
Difference 2   : {4}
```

Above, we used operators `&`, `|`, `-`: these are shortcuts for the dedicated methods. These methods can be showed using `dir`.

[17]:
```python
attrs = dir(var1)
[attr for attr in attrs if not attr.startswith("__")]
```

[17]:
```
['add',
 'clear',
 'copy',
 'difference',
 'difference_update',
 'discard',
 'intersection',
 'intersection_update',
 'isdisjoint',
 'issubset',
 'issuperset',
 'pop',
 'remove',
 'symmetric_difference',
 'symmetric_difference_update',
 'union',
 'update']
```

### 5.3.3 tuple

`tuple` are similar to `list`, yet they are **immutable** (not **mutable**). Whenever it's possible `tuple` must be prefered over `list`.

[18]:
```python
var = (1, 2, 3)
print(var)
```

```
(1, 2, 3)
```

Reminder, `tuple`s are immutable:

[19]:
```python
var[2] = 5
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[19], line 1
----> 1 var[2] = 5

TypeError: 'tuple' object does not support item assignment
```

**Important**: what defines a `tuple` is not the parenthesis `(...)` but **the comma** `,`.

[20]:
```python
var = (1)
print(type(var))
var = 1,
print(type(var))
```

```
var = 1, 2, 3
print(type(var))
```

```
<class 'int'>
<class 'tuple'>
<class 'tuple'>
```

### 5.3.4 dict

Dictionaries makes it possile to store a **value** with a unique **key** as identifier.

[21]:
```
var = {"one": 1, "two": 2, "three": 3}
var
```

[21]:
```
{'one': 1, 'two': 2, 'three': 3}
```

Keys and values are retrieved using dedicated methods. These methods return something similar to a list (but different):

[22]:
```
keys = var.keys()
print(keys)
values = var.values()
print(values)
```

```
dict_keys(['one', 'two', 'three'])
dict_values([1, 2, 3])
```

A call to `items` extract couples of (key, value).

[23]:
```
items = var.items()
print(items)
```

```
dict_items([('one', 1), ('two', 2), ('three', 3)])
```

One can read and modify a dictionary using any key.

[24]:
```
var["one"] = "first"
print(var)
print(var["two"])
```

```
{'one': 'first', 'two': 2, 'three': 3}
2
```

Retriving an element using brackets `[]` is internally done by a call to the `get` method. One can call explicitely the `get` method with a default value in case the key does not exist.

[25]:
```
print(var.get("three", "Unknown!"))
print(var.get("four", "Unknown!"))
```

```
3
Unknown!
```

Other methods are presented using `dir(dict)`.

[26]:
```
attrs = dir(dict)
[attr for attr in attrs if not attr.startswith("__")]
```

[26]:
```
['clear',
 'copy',
 'fromkeys',
 'get',
 'items',
 'keys',
 'pop',
 'popitem',
 'setdefault',
 'update',
 'values']
```

Recall that help is available on any Python object using `help`:

[38]:
```
help(dict.popitem)
```

```
Help on method_descriptor:

popitem(self, /)
    Remove and return a (key, value) pair as a 2-tuple.

    Pairs are returned in LIFO (last-in, first-out) order.
    Raises KeyError if the dict is empty.
```

#### Notes

As for values of a `set`, keys of a dictionary must be **hashable**. Hence, a list cannot be a dictionary key.

In practice, most immutable objects are hashable.

[27]:
```
var = {[1,2]: 5}
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[27], line 1
----> 1 var = {[1,2]: 5}

TypeError: unhashable type: 'list'
```

### 5.3.5 deque

A **deque** is kind of `list` that is optimized for fast values appending at both ends (beginning and end of the container). The speed up is observed maily for very large containers.

`deque` thus have the following methods `appendleft`, `popleft` et `extendleft`.

[28]:
```
from collections import deque
d = deque([1, 2, 3 ,4])
d.popleft()
print(d)
d.appendleft(0)
print(d)
d.extendleft([-5, 63])
print(d)
```

```
deque([2, 3, 4])
deque([0, 2, 3, 4])
deque([63, -5, 0, 2, 3, 4])
```

### 5.3.6 About slicing

**slicing** is a way to extract part of an indexable object (ex: `list`, `tuple`, `str`).

Slincing is done using brackets with a specific notation: `start:end+1:step` (last index is excluded). `step` is not mandatory (if none, it is assumed `step=1`) but if `step` is given then the two other must be given too.

Hereafter, a use case using a `list`.

[29]:
```
var = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
print(var[:5])     # the first 5 values
print(var[5:])     # values from index 5 to the end
print(var[1:5])    # values from index 1 to index 4
print(var[1:5:2])  # values from index 1 to index 4, every 2 values
```

```
[0, 1, 2, 3, 4]
[5, 6, 7, 8, 9, 10]
[1, 2, 3, 4]
[1, 3]
```

Some negative indexers are also possible here, as long as `start` references an element locarted before `end`.

[30]:
```
print(var[-3:])    # the last three values
print(var[-6:8:2])  # values from index -6 to index 8, every 2 values
```

```
[8, 9, 10]
[5, 7]
```

This small trick reverse the container:

[31]:
```
var = var[::-1]
print(var)
```

```
[10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
```

**slicing** returns a **copy** of its argument, even when the argument is mutable.

[4]:
```
var = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
var2 = var[:5]
print(f"var: {var}\t\t var2: {var2}")
var[1] = 1000                                # modification
print(f"var: {var} \t var2: {var2}")
```

```
var: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]          var2: [0, 1, 2, 3, 4]
var: [0, 1000, 2, 3, 4, 5, 6, 7, 8, 9, 10]       var2: [0, 1, 2, 3, 4]
```

### 5.3.7 Booleans

Some values are interpreted as booleans in a **boolean context**, for instance an `if`.

Example: an empty data container has a value of `False` (`True` if it contains at least one element)

[33]:
```
for var in ([], 0, None):
    if var:
        print(f"{var!s:<5} is interpreted as True")
    else:
        print(f"{var!s:<5} is interpreted as False")
```

```
[]    is interpreted as False
0     is interpreted as False
None  is interpreted as False
```

This is not the case outside of these contexts, unless using **type casting** toward a **bool** type.

[34]:
```
print([] is False)         # False, since a list is not a bool, even empty
print(bool([]) is False)   # True
```

```
False
True
```

## 5.4 List comprehensions [medium]

### 5.4.1 Introduction

**list comprehensions** are a way to define lists. Pros of using list comprehensions include:

1. does not pollute the current scope wwith unwanted variables
2. takes only one line of code

Regarding 1, consider the following example.

[1]:
```
var = []
for k in range(5):
    var.append(k**2)

print(k)
```

```
4
```

Variable `k` whose only purpose is to build the list still exists after the loop. This is dangerous in case the name `k` is used elsewhere with no initialization.

List comprehensions are made of:

- one or several `for` loops
- a value to fill the container with, possibly dependant from the indexes of the `for` loops
- (optional) a condition `if`/`then`/`else`

### 5.4.2 Simple examples

Even integers

[2]:
```
var = [2*k for k in range(10)]
var
```

[2]:
```
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
```

Even integers that are multiple of 4.

[3]:
```
var = [2*k for k in range(10) if not 2*k%4]
var
```

[3]:
```
[0, 4, 8, 12, 16]
```

Note that with a `else`, location of `if` is also modified.

[4]:
```
var = [2*k if not 2*k%4 else 0 for k in range(10) ]
var
```

[4]:
```
[0, 0, 4, 0, 8, 0, 12, 0, 16, 0]
```

One can use existing other variables:

[5]:
```
reference = {0: "val_2", 1: "val_1", 2: "val_1", 3: "val_2", 4: "val_1"}
var = [reference[k] for k in range(5)]
var
```

[5]:
```
['val_2', 'val_1', 'val_1', 'val_2', 'val_1']
```

### 5.4.3 With other data containers

**list comprehensions** can be used with other data containers:

- `generateur`
- `set`
- `dict`
- etc...

#### Generators

A generator is a data container whose content is not stored into memory until it has to be retrieved. Retrieval is done either using a classical `for`, or the `next` method.

[6]:
```
var = (letter.upper() for letter in "test" if letter != "e")
var
```

[6]:
```
<generator object <genexpr> at 0x7f77e76625e0>
```

[7]:
```
print(next(var))
print(next(var))
print(next(var))
```

```
T
S
T
```

#### Sets

sets cannot contain duplicate values:

[8]:
```
var = {k%5 for k in range(100)}
var
```

[8]:
```
{0, 1, 2, 3, 4}
```

#### Dictionaries

[9]:
```
var = {k: 2*k+1 for k in range(5)}
var
```

[9]:
```
{0: 1, 1: 3, 2: 5, 3: 7, 4: 9}
```

## 5.5 Function signature [medium]

### 5.5.1 unpacking

#### Key idea

**unpacking** is a way to extract values from a data container into separate variables.

[1]:
```
a, b, c = [1, 2, 3]
print(a)
print(b)
print(c)
```

```
1
2
3
```

If the number of variables on left side of = is not the number of elements of the container, an error is raised:

[2]:
```
a, b = [1, 2, 3]
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Cell In[2], line 1
----> 1 a, b = [1, 2, 3]

ValueError: too many values to unpack (expected 2)
```

One can discard specific elements using *.

[3]:
```
var1, var2, var3, *unwanted, var4 = "abcdefg"
print(var1, var2, var3, var4)
```

```
a b c g
```

Yet, in this case, there must be at most one unknown variable (one *)

[4]:
```
a, *unwanted, c, *unwanted, g = "abcdefg"
```

```
  Cell In[4], line 1
    a, *unwanted, c, *unwanted, g = "abcdefg"
    ^
SyntaxError: multiple starred expressions in assignment
```

#### Use cases

Unpacking can be used in the following situations:

- `for` loops
- permutationw with **no intermediate values**
- arguments passed to a function (see hereafter)

`for` loops:

[5]:
```
for a, b, *_ in [(1, 2, 30),
                 (4, 5, 60, 42)]:
    print(a, b)
```

```
1 2
4 5
```

Permutations:

[6]:
```
a = 5
b = 6
a, b = b, a
print(a, b)
```

```
6 5
```

### 5.5.2 Function signature

#### Simple

A function signature presents the name and expected order of every argument of this function.

**In a function call**, these arguments are of two types:

- positional: their role is defined by the place they take in the arguments order
- named (*keyword arguments*, i.e. `kwargs`)

[7]:
```
def f(a, b, c):
    print(f"`a`: {a}        `b`: {b}        `c`: {c}")

f(1, 2, 3)         # all positional
f(1, 2, c=5)       # some positional, some named
f(c=5, a=1, b=2)   # all named, order does not matter
```

```
`a`: 1        `b`: 2        `c`: 3
`a`: 1        `b`: 2        `c`: 5
`a`: 1        `b`: 2        `c`: 5
```

Named arguments are always placed **after** positional arguments.

[8]:
```
f(1, b=2, 3)
```

```
  Cell In[8], line 1
    f(1, b=2, 3)
              ^
```

```
SyntaxError: positional argument follows keyword argument
```

An argument cannot be specified both as positional and named:

[9]:
```python
f(1, a=1, b=2, c=5)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[9], line 1
----> 1 f(1, a=1, b=2, c=5)

TypeError: f() got multiple values for argument 'a'
```

#### Default value

An argument can be absent from a function call if the function signature defines for this argument a default value. If this argument is given, default value is not taken into account.

[10]:
```python
def f2(a, b, c=3):
    print(f"`a`: {a}        `b`: {b}        `c`: {c}")

f2(1, 2)
f2(1, 2, 5)
```

```
`a`: 1        `b`: 2        `c`: 3
`a`: 1        `b`: 2        `c`: 5
```

It is common to assign a `None` value to optional arguments. Then the body of the function must contain a special treatment for this argument.

[11]:
```python
def f3(a, b, c=None):
    if c is None:
        c = 0
    return a + b + c

print(f3(1, 2))
print(f3(1, 2, 5))
```

```
3
8
```

Beware: default argument is defined only once (when the function is defined): if it **mutable, it will be modified from a call to another**

[12]:
```python
def f4(a, c=[0]):
    c.append(a)
    print(c)
```

```
f4(1)
f4(2)
f4(3)
```

```
[0, 1]
[0, 1, 2]
[0, 1, 2, 3]
```

#### Undeterminated positional arguments: `*args`

A function can take an undeterminated number of positional arguments with the syntax `*args`.

During a function call, all positional arguments undescribed by the function signature are gathered into a `tuple` and passed to the function using the name `args`.

[1]:
```
def f5(a, b, *args):
    print(a, b, args)

f5(1, 2, 3, 4, 5)
f5(1, 2)
```

```
1 2 (3, 4, 5)
1 2 ()
```

Note that word “`args`” is only a convention.

[2]:
```
def f6(a, b, *other_values):
    print(a, b, other_values)
```

Every argument following `*args` must be **named**.

[1]:
```
def f7(a, *args, b):
    print(a, b, args)

f7(1, 3, 4, 5, b=2)
# f7(1, 3, 4, 5, 2)
```

```
1 2 (3, 4, 5)
```

That explains why only `**kwargs` comes after `*args`.

#### Undeterminated keyword arguments: `**kwargs`

Similar to `*args`, `**kwargs` contains named arguments that are not defined by the function signature. Beware that the keys of `kwargs` are of type `str` (and the values are the passed variables).

[5]:
```
def f8(a, b, **kwargs):
    print(kwargs)
    print(a, "  ", b, end="   ")
```

```
    print(kwargs.get("c", 0), end="   ")     # if 'c' does not exist as a dict␣
↪key, take 0
    print(kwargs.get("d", 0))
```

[6]:
```
f8(1, 2)
```

```
{}
1    2    0    0
```

[7]:
```
f8(1, 2, c=3, d=5)
```

```
{'c': 3, 'd': 5}
1    2    3    5
```

[8]:
```
f8(1, 2, d=5)
```

```
{'d': 5}
1    2    0    5
```

One can also provide named argument using a **dictionary unpacking**..

In the example below, unpacking is used to deconstruct the dictionary into a group of named arguments:

- some are explicitely named
- other go to the `kwarg` variable

[10]:
```
f8(1, 2)
f8(1, **{"b": 9, "c": 3, "d": 5})
f8(1, 2, **{"d": 5})
```

```
{}
1    2    0    0
{'c': 3, 'd': 5}
1    9    3    5
{'d': 5}
1    2    0    5
```

### Advanced examples

[18]:
```
def custom_sum(a, b, *args, **kwargs):
    weight = kwargs.get("weight", 1)                  # get the value of␣
↪'weight' if exists, else 1
    print(a + b + sum([arg * weight for arg in args]))

custom_sum(1, 2)
custom_sum(1, 2, 3, 4)
custom_sum(1, 2, 3, 4, weight=2)
custom_sum(1, 2, weight=2)
```

```
3
10
17
3
```

Note: a force of `**kwargs` (and `*args`) is that it can be easily passed from a function call to another.

[19]:
```
def advanced_function(a, b, *args, **kwargs):
    if kwargs.get("advanced") > 10:
        kwargs["weight"] = 1
    else:
        kwargs["weight"] = 0
    custom_sum(a, b, *args, **kwargs)

advanced_function(1, 2, 3, 4, advanced=11, useless_kwarg=1000)
advanced_function(1, 2, 3, 4, advanced=9, useless_kwarg=1000)
advanced_function(*[1, 2, 3, 4], advanced=9, useless_kwarg=1000)
```

```
10
3
3
```

## 5.6 OS interactions [medium]

### 5.6.1 File reading/writing

#### Examples

Python is able to read and modify text files (and binary files, too) using the `open` function. In the example below, a file 'file.txt' was previously created on disk.

[1]:
```
with open("file.txt") as f:
    line_1 = f.readline()
    line_2 = f.readline()

print(line_1, line_2, sep="")
```

```
A new line
Another new line
```

By default, `open` opens the file:

- in **text mode**
- in **read only** mode: modifying the file is not possible

[2]:
```
with open("file.txt") as f:
    new_line = f.write("A new line\nAnother new line\n")
```

```
---------------------------------------------------------------------------
UnsupportedOperation                      Traceback (most recent call last)
Cell In[2], line 2
      1 with open("file.txt") as f:
----> 2     new_line = f.write("A new line\nAnother new line\n")

UnsupportedOperation: not writable
```

To edit the file, one can use one of the following options:

- `'w'`: write to the beginning of the file and existing content is removed!
- `'a'`: write at the end of the file, existing content is kept

[3]:
```
with open("file.txt", "a") as f:
    new_line = f.write("A new line\nAnother new line\n")

with open("file.txt") as f:
    print("Appending to existing file: ", end="")
    print(f.readlines())

with open("file.txt", "w") as f:
    new_line = f.write("A new line\nAnother new line\n")
```

```
with open("file.txt") as f:
    print("Replacing content of existing file: ", end="")
    print(f.readlines())
```

```
Appending to existing file: ['A new line\n', 'Another new line\n', 'A new
line\n', 'Another new line\n']
Replacing content of existing file: ['A new line\n', 'Another new line\n']
```

Note the following methods:

- `readline`: read a single line of a file. If several calls to `readline` are done, lines are displayed one after another.
- `readlines`: read all the lines of the file and store them into a list
- `write`: write a string in the file

#### Notes

`\n` is the universal character to describe a line break:

[4]:
```
s = "This is a sentence.\nA"
print(s[-3:])    # the last three caracters are 'A',
                 # a new line and a dot '.':
                 # the new line is not made of 2 caracters!
```

```
.
A
```

The `with` bloc is important: it makes sure file is open and closed in a clean way.

The other way to manage files is described here after: it is **depreciated** because if `f.close()` is never called then the file might be corrupted or damage the operating system.

[5]:
```
f = open("file.txt")
line_1 = f.readline()
line_2 = f.readline()
f.close()    # never forget this one!
print(line_1, line_2, sep="")
```

```
A new line
Another new line
```

One can create an empty text file:

[6]:
```
with open("my_empty_file.txt", "w") as f:
    pass
```

### 5.6.2 Files management

#### Key ideas

A file path is the adress of a file on the disk. In Python, the preferred way to handle file paths is to use the `pathlib` library (built in). It handles perfectly the differences of separators ('/' or '') between different operating systems. `pathlib.Path` instances can handle both files **and** directories.

[7]:
```
from pathlib import Path
path = Path("/this/is/my/path/a_file.txt")
print(path.name)     # file
print(path.parent)   # directory
print(path.suffix)   # file extension
```

```
a_file.txt
/this/is/my/path
.txt
```

The creation of a `Path` instance does not mean the corresponding path exists:

[8]:
```
path.exists()
```

[8]:
```
False
```

Yet, it can be used to create it:

[9]:
```
if False:
    path.parent.mkdir(
                        parents=True,     # if parents do not exist, they are␣
 ↪created ('this', 'is', 'my')
                        exist_ok=True     # if the path already exists, it does␣
 ↪nothing and does not throw an error
                        )
```

### Absolute and relative file paths

An absolute path is a complete adress of a file (or directory) on the disk. Using Linux, these paths start with '/' (root), using Windows they start with the drive name ('c:/', 'd:/', etc. . . ).

[10]:
```
path
```

[10]:
```
PosixPath('/this/is/my/path/a_file.txt')
```

[11]:
```
path.is_absolute()
```

[11]:
```
True
```

[12]:
```
relative_path = Path("path/relative/to/current/directory")
print(relative_path.is_absolute())
```

```
False
```

Conversely, relative paths are path defined starting from the current directory, which can be obtained using `Path.cwd()`. This directory is also called `'.'`. Th e parent of this directory is called '`..`'.

[13]:
```
path1 = Path("./dir1/dir2/dir3/../..")
path2 = Path("./dir1/")
```

```
print(path1)
print(path2)
```

```
dir1/dir2/dir3/../..
dir1
```

Two relative paths cannot be compared. One must first call the `resolve` method that returns an absolute path.

[14]:
```
print(path1==path2)
print(path1.resolve()==path2.resolve())
```

```
False
True
```

Some libraries do not accept `Path` instances. . . in this case one must use `str`.

[15]:
```
if False:
    print(str(path1.resolve()))
```

### Define complex paths

The / operator creates a single path from two paths. It can be used several times in a row:

[16]:
```
base_path = Path("/my/project/is/in/a/very/deep/dir")
data_path = base_path / "data" / "case_study"
src_path = data_path / "../../src"
file_path = data_path / "a_file.txt"
print(base_path.resolve())
print(data_path.resolve())
print(src_path.resolve())
print(file_path.resolve())
```

```
/my/project/is/in/a/very/deep/dir
/my/project/is/in/a/very/deep/dir/data/case_study
/my/project/is/in/a/very/deep/dir/src
/my/project/is/in/a/very/deep/dir/data/case_study/a_file.txt
```

### Browse your files

Let's create a fictive files structure:

```
A/
    1/
        a/
            file_1.txt
            file_2.txt
        b/
            file_1.txt
            file_2.txt
    2/
```

```
        a/
            file_1.txt
            file_2.txt
        b/
            file_1.txt
            file_2.txt
    useless_file.txt
```

A list of the files of a specific directory is available using the `iterdir` method of a `Path` instance describing this directory.

`iterdir` returns a generator. Below, it is transformed into a list for easier handling:

[17]:
```
p = Path('A')
print(p.iterdir())
print(list(p.iterdir()))
```

```
<generator object Path.iterdir at 0x7f3fd41a46d0>
[PosixPath('A/2'), PosixPath('A/useless_file.txt'), PosixPath('A/1')]
```

One can also browse sub directories using the `walk` function of library `os` (built in).

[18]:
```
import os
for current_dir, subdirs, files in os.walk(p):
    print(f"{current_dir:<8}", subdirs, files)
```

```
A        ['2', '1'] ['useless_file.txt']
A/2      ['b', 'a'] []
A/2/b    [] ['file_1', 'file_2']
A/2/a    [] ['file_1', 'file_2']
A/1      ['b', 'a'] []
A/1/b    [] ['file_1', 'file_2']
A/1/a    [] ['file_1', 'file_2']
```

Note that:

- `os.walk` returns strings
- a method `Path.walk` exists for very recent version of python, and should be prefered over `os.walk` if available

### Operations on files

**Removal** A file can be removed using `Path.unlink`. if it's a directory, then use `Path.rmdir`.

[19]:
```
if False:
    p = Path('A/useless_file.txt')
    print("Existing files: ", list(p.parent.iterdir()))
    p.unlink()
    print("A file was removed: ", list(p.parent.iterdir()))
```

**Move/copy** To move or copy/paste a file, the `shutil` library must be used (built in).

Below, a file is moved to the same directory, but its name is changed:

[20]:
```
import shutil
source = Path('A/useless_file.txt')
destination = Path('A/useless_file_new_name.txt')

shutil.move(source, destination)
```

[20]:
```
PosixPath('A/useless_file_new_name.txt')
```

Copy/paste:

[24]:
```
shutil.move(destination, source)
source = Path('A/useless_file.txt')
destination = Path('A/useless_file_new_name.txt')

shutil.copy2(source, destination)   #  source file still exists
```

[24]:
```
PosixPath('A/useless_file_new_name.txt')
```

**Note**: the copy of metadata (owner of the file, permissions, dates, etc…) might fail!

#### Take away

3 libraries can handle files:

- browse the disk, delete files and directories: use `pathlib` (documentation) in priority, else `os` (documentation).
- move, copy files and directories: use `shutil` (documentation)

### 5.6.3 Run a system call

#### Introduction

The call to an external program from Python makes it possible to build complex scripts that involve several different software components.

The key idea is to define a command the same way one would define it in a terminal (Linux, OS X) or a *cmd* command line (Windows).

#### Example

**One must use the `run` function of library `subprocess` (built in).**

[22]:
```
import subprocess
name = 'something'
result = subprocess.run(args=f"mkdir {name}",
                        shell=True,
                        capture_output=True,
                        check=True)
```

```
print(type(result))
print(result)
```

```
<class 'subprocess.CompletedProcess'>
CompletedProcess(args='mkdir a_new_dir', returncode=0, stdout=b'', stderr=b'')
```

Some explanations:

- `args` describes the command to run
- `shell=True` allows to specify `args` as a `str`. if `shell=False`, then `args` must be set to `['mkdir', 'a_new_dir']`
- `capture_output=True` stores the outputs of the command in the attributes `stdout` (standard output) and `stderr` (error output) of the instance returned by `run` (here this instance is ‘results)
- `check=True` makes sure an error is raised if the system command (described by `args`) fails
- attribute `returncode` of `results` is 0 when the command **succeeds**

## 5.7 Decorators [advanced]

### 5.7.1 Introduction

A **decorator** is a function that takes as input a function and returns a function. Using decorators is done in a generic manner: it can be applied quickly to existing functions to have them behave a bit differently.

### 5.7.2 Simple example

#### Complete syntax

**Code**

[1]:
```
def custom_decorator(function):
    def new_function(*args, **kwargs):
        print(f"This message is printed because `{function.__name__}` was␣
 ↪decorated using `custom_decorator`.")
        result = function(*args, **kwargs)
        return result
    return new_function

def basic_function(a, b):
    print(f"We are in `basic_function`: {a}, {b}")


new_function = custom_decorator(basic_function)
new_function(3, 4)
```

```
This message is printed because `basic_function` was decorated using
`custom_decorator`.
We are in `basic_function`: 3, 4
```

**Explanation** A new function `new_function` is defined in the body of `custom_decorator`. It achieves some work (`print`) and then call the function passed as an argument (`basic_function`). This new funtion is returned and can be stored in a variable.

**Notes** A decorator cannot easily modified what happens **inside** the function. Yet, it can modify the arguments it takes as input and the output it returns

#### Lighter syntax

The same can be achievd without using an intermediate `new_function` variable: the instruction `@decorator_name` is placed the line preceding the `def` keywork of a function to decorate.

[2]:
```
@custom_decorator
def basic_function(a, b):
    print(f"We are in `basic_function`: {a}, {b}")

basic_function(3, 4)
```

```
This message is printed because `basic_function` was decorated using
`custom_decorator`.
We are in `basic_function`: 3, 4
```

Internally, Python replaces `basic_function` by its decorated version.

### 5.7.3 Fictive use case

In the example below, a **prod**, a **sum** and a **pow** functions are defined. They perform the sum/product/iterative power of elements of the iterable they receive.

[3]:
```python
def sum_(var):
    return sum(var)

def prod_(var):
    p = 1
    for e in var:
        p *= e
    return p

def pow_(var):
    if var:
        p = var[0]
        for e in var[1:]:
            p = p ** e
        return p
    else:
        return 1
```

Now, we want these functions to return 0 whenever the result is negative. The common way is to modify these functions directly:

[4]:
```python
def sum2_(var):
    s = sum(var)
    return max(s, 0)

def prod2_(var):
    p = 1
    for e in var:
        p *= e
    return max(p, 0)

def pow2_(var):
    if var:
        p = var[0]
        for e in var[1:]:
            p = p ** e
        return max(p, 0)
    else:
```

```
        return 1
```

But with these modifications, the body of the functions is modified and the functions do not do anymore what they were first intended to do. Instead, one could use an decorator:

[5]:
```
def only_positive(func):
    def new_func(*args, **kwargs):
        result = func(*args, **kwargs)
        return max(result, 0)
    return new_func
```

[6]:
```
@only_positive
def sum_p(var):
    return sum_(var)

@only_positive
def prod_p(var):
    return prod_(var)

@only_positive
def pow_p(var):
    return pow_(var)
```

[7]:
```
var = [-3, 5, 3]
print(sum_(var), prod_(var), pow_(var))
print(sum_p(var), prod_p(var), pow_p(var))
```

```
5 -45 -14348907
5 0 0
```

### Advanced

This is handy, but the functions are still modified a bit and we have to comment the decorator line to remove the special behavior.

Instead, we will define a **decorator with argument** to decide whether we want this special behavior to apply. To this purpose, `set_only_positive` is a function that returns a decorator, depending on whether we want (`positive=True`) or do not want (`positive=False`) to apply the special behaviour on the to-be-decorated function.

[14]:
```
def set_only_positive(positive):
    if positive:
        return only_positive
    else:
        def no_decorator(func):
            return func
        return no_decorator

positive_results = True
```

```python
@set_only_positive(positive_results)
def sum_p(var):
    return sum_(var)

@set_only_positive(positive_results)
def prod_p(var):
    return prod_(var)

@set_only_positive(positive_results)
def pow_p(var):
    return pow_(var)
```

[15]:
```python
var = [-3, 5, 3]
print(sum_(var), prod_(var), pow_(var))
print(sum_p(var), prod_p(var), pow_p(var))
```

```
5 -45 -14348907
5 0 0
```

### 5.7.4 Conclusion

Decorators are advanced features of the python language. In most cases, it can be replaced by (more ugly) simpler solutions. But they are very common in some common libraries, thus it is important to understand the meaning of the `@decorator` syntax.

[ ]:
```python

```

## 5.8 Date/time [medium]

### 5.8.1 Introduction

Handling date, time and delays can be required in a scientific progress, especially in an experimental work. For instance:

- schedule data acquisition
- handle experimental databases

*Python* comes with the `datetime` library to address these issues. Scientific oriented libraries such as `numpy` and `pandas` have a different, **but compatible**, implementation of date/time management.

### 5.8.2 Timestamp

A timestamp is an accurate description of a moment in time.

`date`

The `date` library (built in) is used to describe accurately a date: year, month, day of month.

[1]:
```
from datetime import date
d1 = date(2023, 3, 1)
print(d1.day, d1.month, d1.year)
```

```
1 3 2023
```

The current date is obtained using `date.today()`.

[2]:
```
d2 = date.today()
print(d2.day, d2.month, d2.year)
```

```
21 2 2024
```

As many Python objects, dates can be **compared**:

[3]:
```
if d1 < d2:
    print(f"{d1} is anterior of {d2}")
else:
    print(f"{d1} is posterior of {d2}")
```

```
2023-03-01 is anterior of 2024-02-21
```

Yet, there is no meaning to do the sum of two dates:

[4]:
```
d1 + d2
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[4], line 1
----> 1 d1 + d2
```

```
TypeError: unsupported operand type(s) for +: 'datetime.date' and 'datetime.date'
```

A `datetime` instance comes with its own representation:

[5]:
```
d1
```

[5]:
```
datetime.date(2023, 3, 1)
```

To get a string representation, one must use `date.strftime`, i.e. 'string from time'. This method takes as an argument the wanted **format**. This format must be specified following the special characters described here.

[6]:
```
print(d1.strftime("%d %B, %Y (%A)"))      # custom representation
print(d1.strftime("%x"))                   # official representation for your␣
↪country
```

```
01 March, 2023 (Wednesday)
03/01/23
```

Without using `strftime`, the previously introduced formatting methods (using `f'{var}'`) can be used:

[7]:
```
print(f"The event happened on the {d1:%d}th of {d1:%B} {d1:%Y}.")
```

```
The event happened on the 01th of March 2023.
```

`time`

Following the same idea, python can describe an exact hour: from hour to microseconds:

[8]:
```
from datetime import time
t1 = time(13, 34, 28, microsecond=156545)
print(t1)
print(t1.second)
```

```
13:34:28.156545
28
```

Comparison of `time` instances is also possible:

[9]:
```
t2 = time(11, 14, 54)
print(t2.microsecond)
print(t2 < t1)
```

```
0
True
```

But won't work between date and time instances:

[10]:
```
d1 < t2
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[10], line 1
----> 1 d1 < t2

TypeError: '<' not supported between instances of 'datetime.date' and 'datetime.
↪time'
```

Formatting is possible too:

[11]:
```
print(t2.strftime('%I:%M %p, %S seconds'))
print(f'It is currently {t2:%I}:{t2:%M} {t2:%p} (and {t2:%S} seconds)')
```

```
11:14 AM, 54 seconds
It is currently 11:14 AM (and 54 seconds)
```

`datetime`

`datetime` objects bring together the functionalities of both `date` and `time` objects (still with a microsecond resolution). Beware of not mistaking the `datetime` module (the one of `date` and `time`) and its submodule `datetime` (siblings of `date` and `time`).

[1]:
```
from datetime import datetime

now = datetime.now()
print(now.strftime("%H:%M:%S:%f in %B %Y"))
```

```
16:44:10:142280 in February 2024
```

Feature: a datetime instance can be built from a `str` using the `strptime` function (which is **not** `strftime`):

[13]:
```
var = datetime.strptime("16:50:24:194724 in January 2029", "%H:%M:%S:%f in %B␣
↪%Y")
var
```

[13]:
```
datetime.datetime(2029, 1, 1, 16, 50, 24, 194724)
```

Note that there are some limits to the creation of dates:

[14]:
```
from datetime import MINYEAR, MAXYEAR
print(MINYEAR, MAXYEAR)
```

```
1 9999
```

**Thus, be careful when your experimental data acquisition may last longer than 8000 years.**

### 5.8.3 Time periods: `timedelta`

A time period describes the temporal length of an event. It can be represented by objects of type `timedelta`:

[15]:
```
from datetime import timedelta
dt = timedelta(days=1, seconds=2, microseconds=3, milliseconds=4, minutes=5,␣
↪hours=6, weeks=7)
dt
```

[15]:
```
datetime.timedelta(days=50, seconds=21902, microseconds=4003)
```

[16]:
```
print('Total number of seconds: ', dt.total_seconds())
```

```
Total number of seconds:  4341902.004003
```

Parameters can be **negative**. In the example below, a duration of 55 minutes is equal to a duration of 1 hour minus 5 minutes.

[17]:
```
dt1 = timedelta(minutes=55)
dt2 = timedelta(hours=1, minutes=-5)
dt1 == dt2
```

[17]:
```
True
```

One can substract, sum and even divide these instances:

[18]:
```
dt1 = timedelta(days=1, seconds=2, microseconds=3, milliseconds=4, minutes=5,␣
↪hours=6, weeks=7)
dt2 = timedelta(days=41, seconds=265, microseconds=123, milliseconds=41,␣
↪minutes=51, hours=6, weeks=7)
print('Sum:         ', dt1 + dt2)
print('Difference: ', dt1 - dt2)
ratio = dt2 / dt1
print(f'Division:    {ratio:.2f} ({type(ratio)})')
```

```
Sum:         140 days, 13:00:27.045126
Difference:  -41 days, 23:09:36.962880
Division:    1.80 (<class 'float'>)
```

Important: a `date` (or `datetime`) instance can be added to a `timedelta` instance to get a new `date` (or `datetime`).

[19]:
```
print(d1, dt1, d1 - dt1, type(d1 - dt1), sep="\n")
```

```
2023-03-01
50 days, 6:05:02.004003
2023-01-10
<class 'datetime.date'>
```

[20]:
```
print(var, dt1, var + dt1, type(var + dt1), sep="\n")
```

```
2029-01-01 16:50:24.194724
50 days, 6:05:02.004003
2029-02-20 22:55:26.198727
<class 'datetime.datetime'>
```

# Chapter 6

# Handling exceptions

## 6.1 Exceptions [medium]

### 6.1.1 Introduction

Some run time errors can be anticipated. They must be dealt with using dedicated code instructions: that is called **exceptions handling**.

An exception is a Python object that tells the user about an error occuring at a specific instruction. These exceptions are of several types since they describe several different problems: type errors (`TypeError`), index errors (`IndexError`), . . .

[1]:
```
s = "a string"
s / 5
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[1], line 2
      1 s = "a string"
----> 2 s / 5

TypeError: unsupported operand type(s) for /: 'str' and 'int'
```

[2]:
```
var = [0, 1, 2, 3]
var[4]
```

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
Cell In[2], line 2
      1 var = [0, 1, 2, 3]
----> 2 var[4]

IndexError: list index out of range
```

A message exhibits the problematic instruction in what is called a `Traceback`. A `Traceback` is a list of code instructions concerned by the exception, going from most recent call to a function (the problematic one) to oldest call.

[3]:
```python
def f1(a, b):
    return a / b

def f2(a, b):
    f1(a, b)

def f3(a, b):
    f2(a, b)
```

[4]:
```python
f3(1, 0)
print("Not executed")
```

```
---------------------------------------------------------------------------
ZeroDivisionError                         Traceback (most recent call last)
Cell In[4], line 1
----> 1 f3(1, 0)
      2 print("Not executed")

Cell In[3], line 8, in f3(a, b)
      7 def f3(a, b):
----> 8     f2(a, b)

Cell In[3], line 5, in f2(a, b)
      4 def f2(a, b):
----> 5     f1(a, b)

Cell In[3], line 2, in f1(a, b)
      1 def f1(a, b):
----> 2     return a / b

ZeroDivisionError: division by zero
```

### 6.1.2 Handle an exception

If nothing particular is done, an exception is **blocking** for the running code and the process is terminated. To prevent this termination, one must use a `try` code block:

If an instruction fails in `try`, an exception is raised (as usual) but is not blocking: the content of `except` is ran instead:

[5]:
```python
def f1(a, b):
    try:
        return a / b
```

```
    except ZeroDivisionError as e:
        print("`b` was 0, met the following exception: ", e)


def f2(a, b):
    f1(a, b)

def f3(a, b):
    f2(a, b)

f3(1, 0)
print("Executed")
```

```
`b` was 0, met the following exception:  division by zero
Executed
```

Notes:

- **Whenever it's possible**, one must specify the type of exception to **'catch'** (here `ZeroDivisionError`). Yet, it is also possible to only write `except:` in order to catch all types of exceptions.
- The syntax `as e` store the exception instance (an instance of type `ZeroDivisionError`) in variable `e`.
- Several `except` can follow each other to catch different types of exceptions or define several exceptions types in one `except` (see examples below).

[6]:
```
def f1(a, b):
    try:
        a / b
        b[5]
    except ZeroDivisionError as e:
        print("`b` was 0, met the following exception: ", e)
    except TypeError as e:
        print(f"`b` was of type {type(b)}, met the following exception: ", e)


def f2(a, b):
    f1(a, b)

def f3(a, b):
    f2(a, b)
```

[7]:
```
f3(1, 0)
f3(0, 1)
```

```
`b` was 0, met the following exception:  division by zero
`b` was of type <class 'int'>, met the following exception:  'int' object is not
subscriptable
```

[8]:
```
def f1(a, b):
    try:
        a / b
        b[5]
    except (ZeroDivisionError, TypeError) as e:
        print(e)
f1(0, 1)
```

```
'int' object is not subscriptable
```

**The `try` code section must include as few instructions as possible** (so that unpredicted errors won't be covered by `except`).

Thus, the `else` section is used: instructions in `else` are executed if risky code in `try` runs with no exception. Put differently: `else` is not ran if `except` is ran.

[9]:
```
def f1(a, b):
    try:
        x = a / b
    except (ZeroDivisionError, TypeError) as e:
        print(e)
    else:
        y = 2 * x + 5
        return y
f1(5, 2)
```

[9]:
```
10.0
```

Note: another clause exists: `finally`. Instructions in `finally` will be executed just before `try` terminates (i.e. before an exception is raised within `try`, or after the very last instruction of `try`). It is useful for some advanced cases.

### 6.1.3 Raise an exception

It is possible to create a code interruption depending on certain conditions. In this case, with use the `raise` statement. In the example below, a `ValueError` is raised whenever the acquired value is negative. This error is catched in using a dedicated `except`.

[10]:
```
from random import randint

def sensor_reading(only_valid_data=True):
    # fake real-time data acquisition
    new_value = randint(-1, 10)
    if new_value < 0 and only_valid_data:
        raise ValueError(f"Sensor default, got negative value {new_value}.")
    return new_value


def data_acquisition(replacement_value):
    data = []
```

```
    for k in range(20):
        try:
            new_value = sensor_reading()
        except ValueError as e:
            print(f"Time step {k}: ", e)
            new_value = replacement_value
        data.append(new_value)
    return data

data_acquisition(0)
```

```
Time step 2:  Sensor default, got negative value -1.
Time step 19:  Sensor default, got negative value -1.
```

```
[10]: [2, 8, 0, 9, 6, 9, 9, 4, 0, 6, 2, 3, 4, 9, 5, 5, 10, 0, 8, 0]
```

### 6.1.4 Conclusion

`try`/`except` is a powerful way to handle expected errors, i.e. errors that will likely happen and that need a special treatment.

## 6.2 Debugger [medium]

### 6.2.1 Introduction

#### Key idea

In a development process, the developper can uncounter some unpredictable and unwanted errors. These can be:

- classical Python exceptions, as introduced previously
- inconsistent scientific results that sugget a code error

The ***debugger*** is a tool that makes solving these problems an easier task. Using the debugger, one can pause running code at some given instructions, called ***breakpoints***. Whenever a breakpoint is encountered, the user can:

- observe the some variable values (local or global variables)
- evaluate some new expressions using these variables
- enter the details of the breakpoint and run these instructions one after another
- resume the execution until a new breakpoint is met

Execution is **always** paused **before** the breakpoint, as if breakpoint occured at the end of the previous instruction.

#### Howto

The native debugger of Python is a library called `pdb`. Whenever the `set_trace` method is used to declare breakpoints, execution falls back to debug mode.

Yet, `pdb` is not handy, and a much better debugger integration is done within recent IDE. In this part, the debugger of *Visual Studio Code* is introduced.

### 6.2.2 Debugging using VSCode

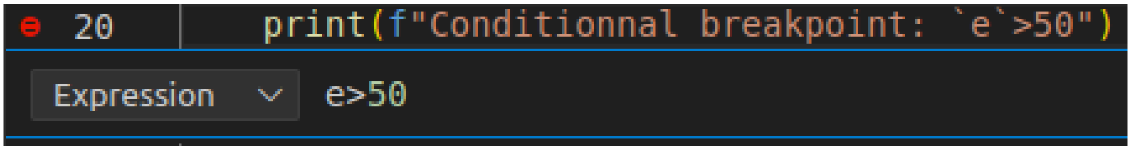
#### Set some *breakpoints*

Let's focus on the following code:

to_debug.py > ...

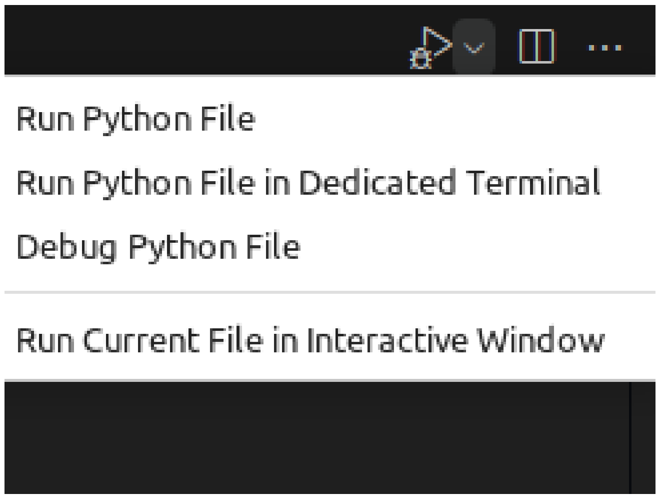
```
1
2   def f1(a, b, c):
3       print("Entering f1")
4       a = min(a, 5)
5       print("After first instruction")
6       d = a * b + c
7       print("After second instruction")
8       idx = 0
9       while d < 100:
10          d += 1
11          f2(a, d)
12          idx += 1
13      print("Exiting f1")
14
15
16  def f2(a, d):
17      print("Entering f2")
18      e = a ** (d%5)
19      f3()
20      print(f"Conditionnal breakpoint: `e`>50")
21      print("Exiting f2")
22
23
24  def f3():
25      print("Entering f3")
26      print("Still in f3")
27      print("Exiting f3")
28
29
30  f1(7, 8, 4)
```

By a left clic in the margin, 4 breakpoints were defined (red dots). At lines 4, 9 and 19 are normal breakpoints. Line 20 is a conditional breakpoint: a breakpoint that is activated is and only if `e>50`.




### Launch the debugger

Unfold the menu at the right hand side of the page and clic on 'Debug Python file':




A new toolbar is printed at the top:

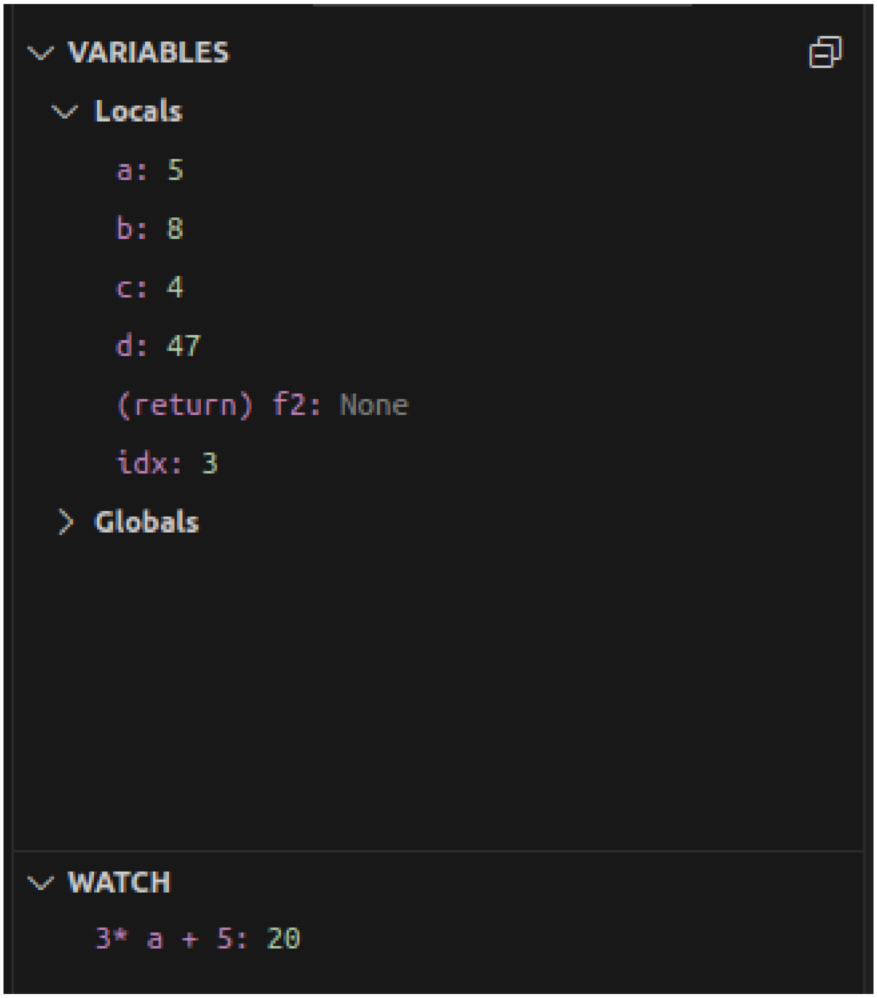
From left to right:

- *Continue*: resume execution until next breakpoint
- *Step over*: run one instruction after another. If the instruction is a function call, execution is paused only at breakpoints of the called function.
- *Step into*: run one instruction after another. If the instruction is a function call, execution is paused at every instruction of this call, no matter the presence of breakpoints or not.
- *Step out*: get out of a previously issued *step into* by executing every instruction without pausing, until the end of the function
- *Stop*: exit debug mode

### Debug tools

At every moment, local **variables** (for instance, the variables defined in the function currently inspected) are showed in a pannel on the left hand side. Their value change in real time as execution goes on.

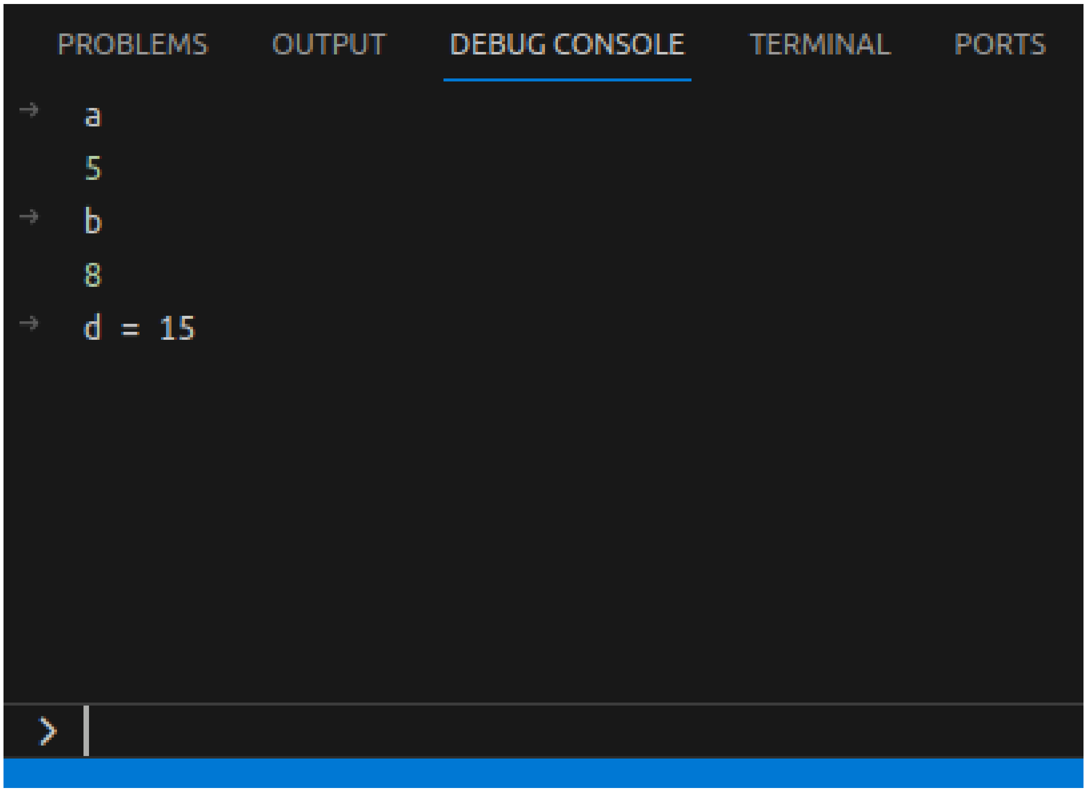
Similar to panel **variables** is panel **watch**: one can define custom Python expressions involving variables. These expressions are also updated as the execution goes on.




A right clic on a variable shows options to:

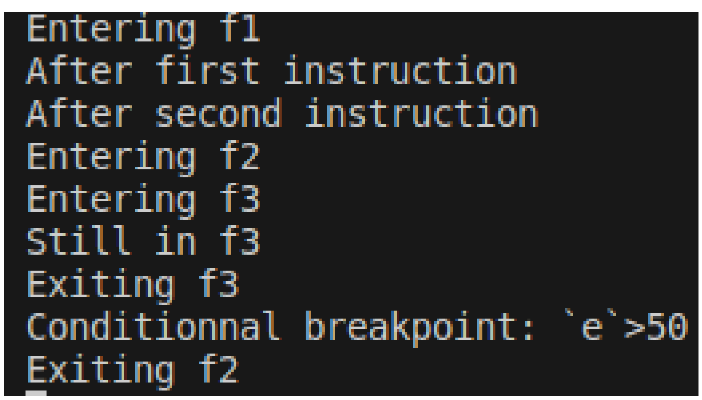
- copy the variable value
- explicitely set this value

Yet, these options are more accessible in the **DEBUG CONSOLE**, at the bottom of the window:




### Execution

As usual, outputs of the execution are visible in the **TERMINAL** tab, next to **DEBUG CONSOLE**.




### Important notes

**There is not return trip in debug mode: the execution of an instruction cannot be undone.**

For this reason, breakpoints must be carefully placed **before** the problematic section. Usually, setting less than 5 breakpoints is enough to debug, as functions *Step over* and *Step into* are pretty powerful.

## 6.3 Logging [advanced]

### 6.3.1 About this notebook

Avoid running again the cells of this notebook.

### 6.3.2 Introduction

A running code can be monitored using carefully placed `print` statements. Yet, this solution does not allow to choose:

- Whether or not the messages must be printed during an execution. To do this, one must comment out or uncomment the `print` statements.
- Where are the messages printed: standard output (terminal) and/or files(s) on disk.

The `logging` library solves these problems.

### 6.3.3 Exemple

#### Setting up a logger

[1]:
```
import logging

def define_logger(base_level):
    logger = logging.getLogger()
    logger.setLevel(base_level)

    overview_handler = logging.StreamHandler()
    overview_handler.setLevel(logging.INFO)
    logger.addHandler(overview_handler)

    problem_handler = logging.StreamHandler()
    problem_handler.setLevel(logging.WARNING)
    logger.addHandler(problem_handler)

    return logger

logger = define_logger(base_level=logging.INFO)
```

The `logger` is the base object used to write messages. When it receives a message whose level is higher than `base_level`, the `logger` shares it with its `handler` instances. here, 2 handlers are defined:

- `overview_handler` will write messages whose level is at least `INFO`. `INFO` messages describe simply what is going on in the code.
- `problem_handler` will write messages whose level is at least `WARNING`. `WARNING` messages tell the user about a small problem.

Both these handlers publish their messages in the output console ("`StreamHandler`").

Then, what is the levels hierarchy? It is given in the documentation page of the logging library:

| | |
|---|---|
| `logging.DEBUG` | 10 |
| `logging.INFO` | 20 |
| `logging.WARNING` | 30 |
| `logging.ERROR` | 40 |
| `logging.CRITICAL` | 50 |

These levels are internally represented as integers, but one must use the syntax `logging.[...]` instead.

### Using a logger

Using the previous example regarding data acquisition:

[2]:
```
from random import randint
from time import sleep

def sensor_reading(only_valid_data=True):
    # fake real-time data acquisition
    sleep(1)
    new_value = randint(-10, 10)
    if new_value < 0 and only_valid_data:
        raise ValueError(f"Sensor default, got negative value {new_value}.")
    return new_value

def data_acquisition(logger, replacement_value=0):
    data = []
    for k in range(5):
        try:
            new_value = sensor_reading()
            logger.info(f"Time step {k}: good value: {new_value}")
```

```
        except ValueError as e:
            new_value = replacement_value
            logger.warning(f"Time step {k}: bad value was replaced with␣
↪{replacement_value}")

        data.append(new_value)
    return data
```

[3]:
```
data_acquisition(logger, 0)
```

```
Time step 0: good value: 9
Time step 1: bad value was replaced with 0
Time step 1: bad value was replaced with 0
Time step 2: good value: 1
Time step 3: good value: 5
Time step 4: good value: 4
```

[3]:
```
[9, 0, 1, 5, 4]
```

In the output console ('*Stream*'), we can notice two different behaviours:

- whenever acquisition succeeds (`value`>=0), a message with level `logging.INFO` is printed. Among the two defined handlers, only `overview_handler` prints this message since the level of `problem_handler` (`logging.WARNING`) is higher than `logging.INFO`.
- whenever acquisition fails, both handlers print the failure message because it is published with a level `logging.WARNING`. Thus this message is printed two times.

### Advanced features

Defining two handlers having the same output is not that interesting. hereafter, the `overview_handler` is modified so that its messages are written to a file.

Moreover, some information are added to the logged messages. This is done using a `__Formatter__` object:

- The time stamp of message production
- The level of the message

The information that can be added to each message are described in the documentation. Note that the `style` argument tells that the formatting syntax is `{}` (see `fmt` argument).

[2]:
```
import logging


def define_logger2(base_level):
    logger = logging.getLogger()
    logger.setLevel(base_level)
    formatter = logging.Formatter(fmt='{asctime} :: {levelname} :: {message}',
                                  datefmt='%H:%M',
```

```
                                  style='{')

    overview_handler = logging.FileHandler("overview_log.txt", mode="w")
    overview_handler.setLevel(logging.INFO)
    overview_handler.setFormatter(formatter)
    logger.addHandler(overview_handler)

    problem_handler = logging.StreamHandler()
    problem_handler.setLevel(logging.WARNING)
    problem_handler.setFormatter(formatter)
    logger.addHandler(problem_handler)

    return logger

logger2 = define_logger2(base_level=logging.INFO)
```

[3]:
```
data_acquisition(logger2, 0)
```

```
19:38 :: WARNING :: Time step 0: bad value was replaced with 0
```

[3]:
```
[0, 1, 2, 6, 8]
```

[6]:
```
with open("overview_log.txt", "r") as file:
    content = file.read()
    print(content)
```

```
19:38 :: WARNING :: Time step 0: bad value was replaced with 0
19:38 :: INFO :: Time step 1: good value: 1
19:38 :: INFO :: Time step 2: good value: 2
19:38 :: INFO :: Time step 3: good value: 6
19:38 :: INFO :: Time step 4: good value: 8
```

We can notice that:

- The console output is limited to messages having a level `logging.WARNING`.
- The file *overview_log.txt* contains both levels of messages.

# Part III

# Share one's code

# Chapter 7

# Environments [easy]

## 7.1 Introduction

A *Python* installation basically consists in:

- the *Python* executable itself: `bin` directory
- the default packages included within Python
- the third party packages installed by the user: `lib/*/site-packages` directory

Whenever one installs a new package, others are updated first and then it is downloaded and installed.

Thus, there exists a risk to break the compatibility with previous packages. Moreover, default Python installation is - regarding Linux - a system-wide installation: if some components are modified there exists some instability risks.

For these reasons, the preferred way is to create an environment as soon as a new Python project is started. The simplest way is to use environments managers such as ***Anaconda***.

## 7.2 Anaconda

### 7.2.1 Introduction

Anaconda is a software that manages Python environments. The software comes with a graphical interface (Anaconda Navigator) but the preferred way is to use the command line: faster, more stable. The document is accessible here.

Hereafter, all commands must be ran in a command line.

### 7.2.2 Create an environment

the first step is to install Anaconda (or Miniconda, a smaller alternative). Once installed, an environment can be created using a command similar to this one:

```
conda create -n myenv python=3.11 scipy=0.17.3 astroid babel
```

We can notice :

- the environment name ('myenv')
- the Python version (3.11)
- packages to be installed, with some specified versions

Instead of specifying a name, one can specify a location of the new environment:

```
conda create --prefix ./envs python=3.11
```

### 7.2.3 Activate an environment

Activating an environment is telling the computer that everything that relates to Python must be ran within the environment.

#### Using the command line

In a command prompt,`conda info --envs` gives a list of available environments.

A star `*` shows the currently activated environment. By default, it is the 'base' environment, i.e the one that comes with Anaconda installation. The 'base' environment must **never be used**.

Let's activate `myenv`:

```
conda activate myenv
```

'myenv' is showed in parenthesis (instead of 'base').

Once activated, we can manage this environment:

- list current packages: `conda list`.
- install new packages: `conda install` (complete documentation).

We can also:

- start a new interactive session:
  - `python` is the standard Python interpreter
  - `ipython` if the magic Python interpreter, with extended commands and friendly interface (install needed)
- run a Pyhon file: `python my_program.py`.

#### Using Vscode

If the environment is not automatically detected, it can be chosen using the `Select interpreter` interpreter tool (using `Ctrl`+`Maj`+`P`). In this tool, one can specify the executable Python path (`bin` directory of the environment).

#### Duplicate an environment

Environment duplication makes it possible to work in the same conditions on several different computers (for instance, a personnal computer and a working station). Yet, it works best when all computers have the same operating system.

**From computer 1 . . .** Once the environment is activated:

```
conda list --explicit > spec-file.txt
```

This command exports to a file (`spec-file.txt`) all the details of the environment.

**. . . to computer 2**

```
conda create --name same_env_computer_2 --file spec-file.txt
```

This command install an environment following specifications of `spec-file.txt`.

### 7.2.4 Notes

- Anaconda is not only a Python packages manager: a conda environment can handle other softwares and successfully isolate them from the remaining of the operating system.

- Regarding Python, only some packages can be installed using Anaconda:

  1. some packages exist only on the official Python package repository, called PyPi. Those must be installed using `pip`.

     **In a conda environment**, it is strongly unrecommend to mix conda and pip packages (though it's possible). The prefered procedure is:

     – whenever it's possible, install the conda version of the package
     – install the pip version when there is no other choice
     – try to avoid conda packages after some pip packages were installed

  2. some packages exist only as source code that can be retrieved from shared plateforms such as GitHub or GitLab

- Anaconda is not the only way to handle Python environments. Another way is using the `venv` module. Pros include:

  – compatible with pip
  – lighter than conda

  Yet, managing venv environments is slightly less intuitive than conda environments.

## Chapter 8

# Structuration

## 8.1 Imports [easy]

### 8.1.1 Introduction

The code of large Python projects is spread among several files. One must be able to use within a file the software components stored in another file.

Python handles imports by replacing the idea of file by the idea of **module**. A set of modules constitutes a **package**.

In this part, the `numpy` package is used to demonstrate the various import methods.

### 8.1.2 Full import

With the keywork `import`, a package (group of subpackages and modules) or a module is placed into the local memory space.

We can make use of the module. For instance, list its attributes using the `dir` function.

[1]:
```
import numpy
dir(numpy)[:10]
```

[1]:
```
['ALLOW_THREADS',
 'AxisError',
 'BUFSIZE',
 'CLIP',
 'ComplexWarning',
 'DataSource',
 'ERR_CALL',
 'ERR_DEFAULT',
 'ERR_IGNORE',
 'ERR_LOG']
```

It is handy to associate a shorter name to the imported module: this is called an **alias**.

[2]:
```
import numpy as np
dir(np)[:10]
```

[2]:
```
['ALLOW_THREADS',
 'AxisError',
 'BUFSIZE',
 'CLIP',
 'ComplexWarning',
 'DataSource',
 'ERR_CALL',
 'ERR_DEFAULT',
 'ERR_IGNORE',
 'ERR_LOG']
```

The components that can be imported are available as a hierarchy from the root package (here: `numpy`):

[7]:
```
import numpy.random.bit_generator
type(numpy.random.bit_generator)
```

[7]:
```
module
```

### 8.1.3 Relative import

Using `from ... import ...`, some specific components are placed in the local memory space. These components can be:

- variables
- classes
- functions
- modules

[2]:
```
from numpy.random import randint
```

### 8.1.4 Good practices

#### Import only the needed content

Importing Python objects can be unnecessarily time-consuming. Thus, relative imports mist be preferred over absolute imports so that only needed components are imported.

[5]:
```
from numpy import log, sqrt
from numpy.random import rand
```

#### Choose the import name wisely

If a chosen alias is also an existing variable name, the variable reference will be lost (shadowing):

[6]:
```
rd = 5
print(rd)
from numpy.random import randint as rd
print(rd)
```

```
5
<built-in method randint of numpy.random.mtrand.RandomState object at
0x7f1ea0513b40>
```

#### Move all imports to the beginning of the file

For clarity purpose, in an ideal world, all imports must be placed at the beginning of the file and be sorted:

1. By origin:
    1. Built-in packages: `os`, `sys`, `pathlib`, etc…
    2. Third-party packages from internet: `pandas`, `numpy`, `matplotlib`, etc…
    3. Your local packages or modules

2. By alphabetical order

n.b.: some IDE order the imports automatically.

Example of sorted imports:

[7]:
```
from os import getcwd, lstat
from time import sleep

from pandas import DataFrame, Interval

# from mypackage.mymodule import a, b, c
```

## 8.2 Package structuration [medium]

### 8.2.1 Introduction

To import some content, Python must know:

- which directory is a package
- where to find such directories

Hereafter is presented a simple methodology to access some Python code that is stored elsewhere on the disk.

### 8.2.2 Example

```
- dir
  ├── package_parent
  │   ├── src
  │   │   ├── __init__.py
  │   │   ├── subpackage_1
  │   │   │   ├── __init__.py
  │   │   │   ├── main_subpackage_1.py
  │   │   │   ├── subpackage_1_A
  │   │   │   │   ├── __init__.py
  │   │   │   │   ├── module_1_A_1.py
  │   │   │   │   └── module_1_A_2.py
  │   │   │   └── subpackage_1_B
  │   │   │       ├── __init__.py
  │   │   │       ├── module_1_B_1.py
  │   │   │       └── module_1_B_2.py
  │   │   └── subpackage_2
  │   │       ├── __init__.py
  │   │       ├── main_subpackage_2.py
  │   │       ├── subpackage_2_A
  │   │       │   ├── __init__.py
  │   │       │   ├── module_2_A_1.py
  │   │       │   └── module_2_A_2.py
  │   │       └── subpackage_2_B
  │   │           ├── __init__.py
  │   │           ├── module_2_B_1.py
  │   │           └── module_2_B_2.py
  │   └── test.py
  └── project
      └── script.py
```

There are two directories sharing the same parent:

- `package_parent`: hosts the `src` directory which contains a Python package.

  In `src`, the first `__init__.py` file tells Python that directory `src` is a **package**. The other `__init__.py` files define a subpackages whose name are the directory they are in (`subpackage_1`, `subpackage_1_A`, etc...). The other `*.py` files are those containing useful code. They are called **modules**. Each of them is filled with:

    – a `print` that tells about the file name
    – a variable `var`

- `project`: a fictive Python project that contains a Python file

### 8.2.3 Search for packages

**When importing a package, Python looks for it**:

- in the current directory
- in files described in the `sys.path` list

Thus, the import of `src` from the `project` directory (for instance in `script.py`) will fail as current directory (`project`) is not the one of the package (`package_parent`).

[1]:
```
import os
os.chdir(r'dir/project')
import src
```

```
---------------------------------------------------------------------------
ModuleNotFoundError                       Traceback (most recent call last)
Cell In[1], line 3
      1 import os
      2 os.chdir(r'dir/project')
----> 3 import src

ModuleNotFoundError: No module named 'src'
```

Let's move to `package_parent`:

[2]:
```
os.chdir(r'../package_parent/')
import src
```

```
I am __init__ of package
```

Import succeeded.

Yet, changing current directory is not a good solution to access packages. **The preferred way is to modify the `sys.path` variable**. The path is inserted at first position:

[3]:
```
os.chdir(r'../project')    # back to a dir different than the one compatbile␣
 ↪with import of `src`
from sys import path
from pathlib import Path
path.insert(0, str(Path('../package_parent/').resolve()))
```

Then the import is running smoothly:

[4]:
```
%reset -f --aggressive
import src
```

```
culling sys module...
I am __init__ of package
```

**Note**: in Python console, packages/modules are never imported twice. Thus, for explanation purpose, the magic function `reset -f` is used here to erase all the memory content, included imports, so that we can experience a second import. Usually, this is not necessary (and `--agressive` must not be used).

### 8.2.4 Add some content to __init__

When importing a module, one can access its components:

[5]:
```
%reset -f --aggressive
from src.subpackage_1.subpackage_1_A import module_1_A_1
module_1_A_1.var
```

```
culling sys module...
I am __init__ of package
I am __init__ of subpackage_1
I am __init__ of subpackage_1_A
I am module_1_A_1
```

[5]: 1

Conversely, when importing a package 'my_package', with an empty `__init__.py` file, modules or subpackages that are contained in 'my_package' are not imported. In the code snippet below, access to a module of 'package' ('module_1_B_1') is impossible:

[6]:
```
%reset -f --aggressive
import src.subpackage_1.subpackage_1_B as package
package.module_1_B_1.var
```

```
culling sys module...
I am __init__ of package
I am __init__ of subpackage_1
I am __init__ of subpackage_1_B
I am __init__ of subpackage_1_A
I am module_1_A_1
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
Cell In[6], line 3
      1 get_ipython().run_line_magic('reset', '-f --aggressive')
      2 import src.subpackage_1.subpackage_1_B as package
----> 3 package.module_1_B_1.var

AttributeError: module 'src.subpackage_1.subpackage_1_B' has no attribute
 ↪'module_1_B_1'
```

Suppose we add some additionnal content to `module_1_A_1`:

```
print("I am module_1_A_1")
var = 1

def new_func():
    print("I am `new_func`")
```

To import this content when importing `subpackage_1_A`, the file `src/subpackage_1/subpackage_1_A/__init__.py` is completed with:

```
print("I am __init__ of subpackage_1_A")
useless_var = 10
from .module_1_A_1 import var
from .module_1_A_1 import new_func as imported_func
```

The syntax `.[module_name]` tells Python to search for a module in the current package.

Then, back to our project, the import of module content does not fail:

[7]:
```
%reset -f --aggressive
import src.subpackage_1.subpackage_1_A as package
print(package.useless_var)
print(package.var)
package.imported_func()
```

```
culling sys module...
I am __init__ of package
I am __init__ of subpackage_1
I am __init__ of subpackage_1_A
I am module_1_A_1
10
1
I am `new_func`
```

Explanation: `import src.subpackage_1.subpackage_1_A as package` executed the instructions contained in the `__init__` file of `subpackage_1_A`. These instructions make avilable some content of `module_1_A_1`.

The search is possible in siblings packages or parent packages using the `..` syntax. For instance, let's import the same components but from `module_1_B_1.py` by modifying `src/subpackage_1/subpackage_1_B/__init__.py`:

```
print("I am __init__ of subpackage_1_B")
from ...subpackage_1.subpackage_1_A import imported_func
```

[8]:
```
%reset -f --aggressive
import src.subpackage_1.subpackage_1_B as package
package.imported_func()
```

```
culling sys module...
I am __init__ of package
I am __init__ of subpackage_1
I am __init__ of subpackage_1_B
```

```
I am __init__ of subpackage_1_A
I am module_1_A_1
I am `new_func`
```

**Note**: one can also use the `from module import *` syntax to import everything from a module:

```
print("I am __init__ of subpackage_1_B")
from ...subpackage_1.subpackage_1_A import *
```

Yet, this syntax must be avoided.

### 8.2.5 Take away

**Never copy/paste package directories to make import easier!**

#### Packages and modules

- a module is a `*.py` file within a package
- a package is a directory containing an `__init__.py` file
- file `__init__.py` is ran when the package is imported
- some content to be imported can be added to `__init__.py` files using the `.`, `..`, `...` (etc…) syntax to browse sibling subpackages
- a content is never imported twice

#### `sys.path`

- Modifying `sys.path` is a way to access some Python code stored elsewhere on the disk.
- One must **never** copy/paste package directories to make import easier.

# Chapter 9

# Documentation

## 9.1 Documentation writing [medium]

### 9.1.1 Introduction

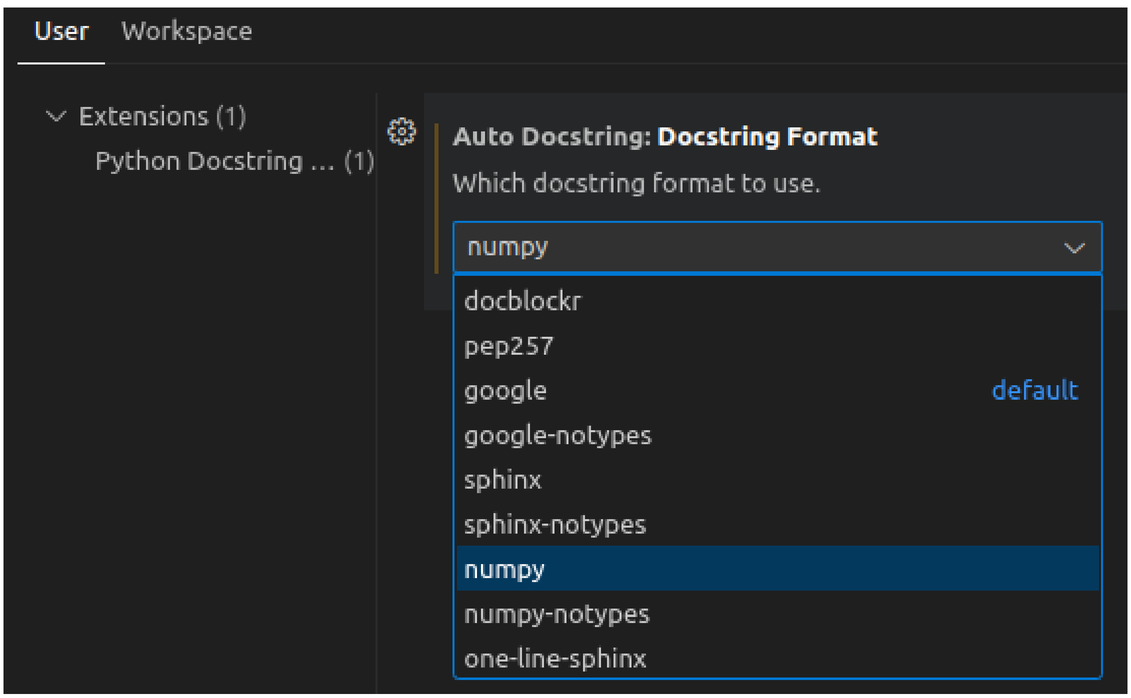
Documenting a code consists in writing ***docstrings***. *Docstrings* are special strings attached to a code section that a have special meaning for the interpreter. They are what is displayed using the `help` function.

In this part, the `src` package introduced previsouly is documented using VSCode (see file `module_2_A_1.py`).

### 9.1.2 Setting up the needed tools

A plugin is needed to write proper docstrings: **autoDocstring** (Nils Werner).

A docstring contains some important information (attributes, types, explanations) that must be formatted in a consistent way. Several formatting mode exist, but a common one is the `numpy` formatting mode.




This format follows these rules. A quick overview is shown here after:

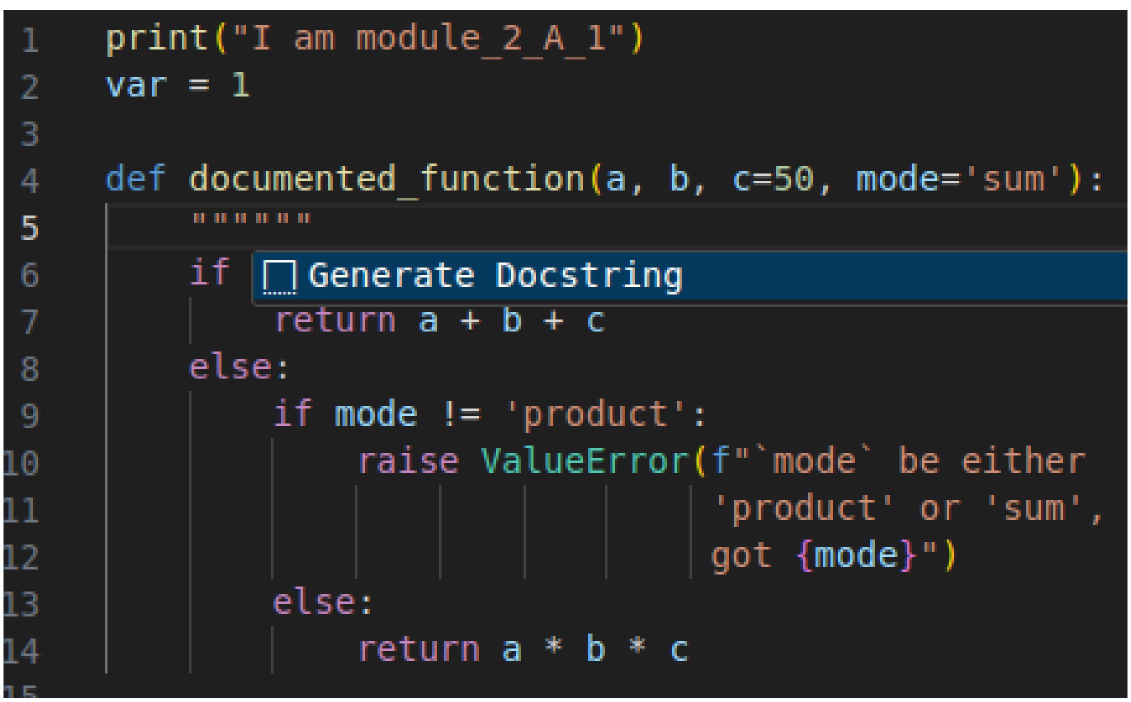
[1]:
```
def abc(a: int, c = [1,2]):
    """_summary_

    Parameters
    ----------
    a : int
        _description_
    c : list, optional
```

```
        _description_, by default [1,2]

    Returns
    -------
    _type_
        _description_

    Raises
    ------
    AssertionError
        _description_
    """
    if a > 10:
        raise AssertionError("a is more than 10")

    return c
```

### 9.1.3 Create a docstring

One can create docstrings for **modules, functions, classes and methods**:

1. Place the carret immediately after the definition line (ex: `def` or `class`)
2. write `"""`
3. press `enter`



### 9.1.4 Add some content to a docstring

#### Key idea

Prioritary information is given first:

1. Purpose of the function
2. Input parameters
3. Returned parameters

Some reminders:

- functionalities of the code are first described using a software point of view.

  Then, a scientific explanation is added if needed.

- imperative mood must be used

```python
def documented_function(a, b, c=50, mode='sum'):
    """Compute either the sum or the product of its arguments,
    depending on parameter `mode`.

    Parameters
    ----------
    a : float
        first parameter of the operation
    b : float
        second parameter of the operation
    c : float, optional
        third parameter of the operation, by default 50
    mode : {'sum', 'product'}
        operation to run on `a`, `b` and `c`

    Returns
    -------
    float
        The result of operation described by `mode`

    Raises
    ------
    ValueError
        If mode is not one of 'sum' or 'product'
    """
    if mode == 'sum':
        return a + b + c
    else:
        if mode != 'product':
            raise ValueError(f"`mode` be either
                                 'product' or 'sum',
                                 got {mode}")
        else:
            return a * b * c
```

Notes: all references to a software element (variable, module, function and classes) must be quoted with **backticks**: ‘ (`Alt Gr + 7` on a french keyboard).

### An iterative process

It is very common to discover weaknesses in the code while writing docstrings:

- possibility of erroneous scientific results
- unconsistent code from one component to another
- instability risk
- ...

For these reasons, the preferred way of writing documentation is first documenting all the components without any detail, and then go further when additional information is needed.

## 9.2 Documentation generation [advanced]

### 9.2.1 Introduction

#### What is doc generation for?

Once docstrings are written, documentation can be read in two ways:

1. In interactive mode using the `help` function.
2. In a dedicated document making simpler the large scale diffusion of the code.

This part presents the second way. Such a dedicated document is built from the docstrings in `*.py` files.

#### When to generate documentation?

Documentation generation must be done when the code API is stable. Recall that the API is all the functions, classes and methods that makes it possible to use the code without caring about its internal behaviour.

### 9.2.2 Overview

There are 3 main steps to doc generation.

1. Analysis of Python code to extract the docstrings. Conversion of these docstrings in documentation files with extension `rst`.

2. Definition of a structure for the final documentation: table of contents, sections, etc...

3. Documentation generation in 2 common formats:

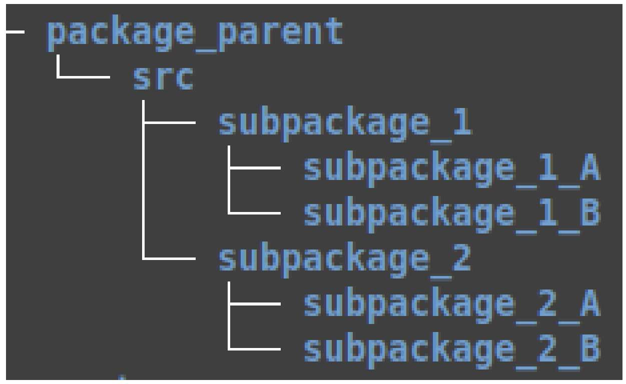
   - html
   - pdf

### 9.2.3 Details

#### Setting up the tools

All the doc generation process can be done using `sphinx`, which is itself a Python package that can be installed using `conda` or `pip`. Be careful to install `sphinx` in the environment used by the Python project.

Let's document the following package:




Here is the documentation environment set up process:

1. Open a commond prompt in directory 'package_parent'
2. Move to a new 'doc' directory
3. Run `sphinx-quickstart`

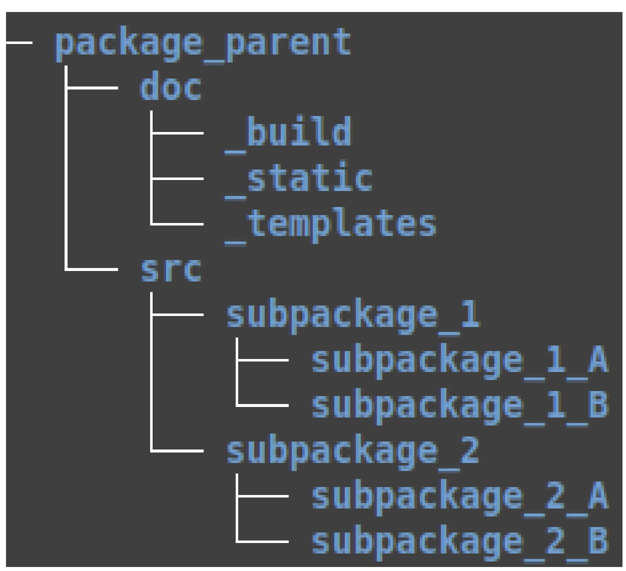
Some files are created in 'doc':

- `conf.py`: contains `sphinx` configuration for the project (step 1 et 3 décrites described in part 'Overview').

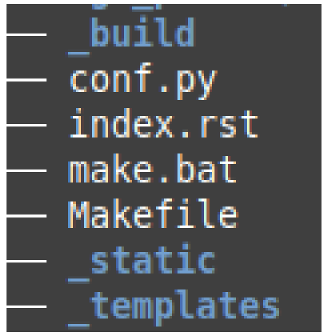
- `index.rst`: structure of the final documentation (step 2).

  This is a *reStructuredText* file (*markup language*).

Working directory is now:




And 'doc' directory:




#### Step 1: Converting docstrings

To have Sphinx understood the *numpy* docstring format, the ***napoleon*** plugin is needed. This is configured in `conf.py`:

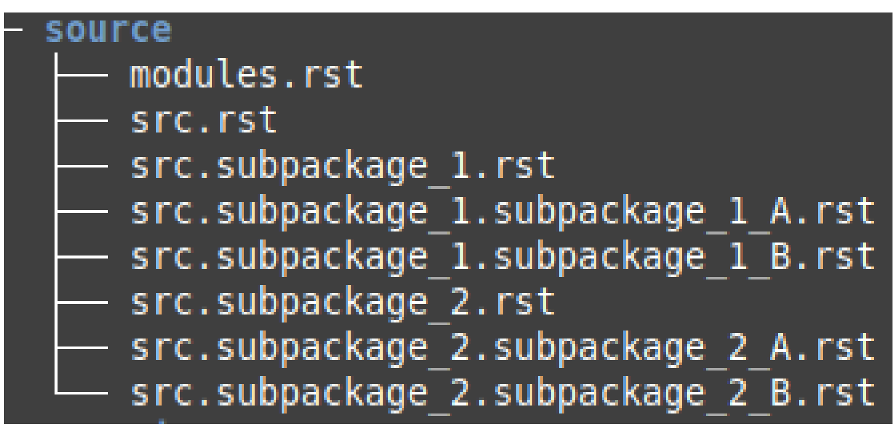
[1]:
```
extensions = ['sphinx.ext.napoleon']
```

Then conversion can be done. Let's move to `doc` directory and run:

```
sphinx-apidoc -o source/ ../src/
```

What happens:

- directory `../src/` is the one that contains our code: the first `__init__.py` telling Sphinx where to look for the package.
- directory `source` is created and contains `rst` files.




#### Step 2: Defining a structure

Let's order the `rst` file using the `index.rst` file:

```
   You can adapt this file completely to your liking, but it should at least
   contain the root `toctree` directive.

Welcome to my_package's documentation!
======================================

.. toctree::
   :maxdepth: 2
   :caption: Contents:
    Here is a short description for our doc page.

   source/src.subpackage_2.subpackage_2_A.rst
```

**note**: an introduction to `rst` language is available here.

### Step 3: Generating documentation

Documentation is generated using the *html* format (the one that descibes web pages).

When generating the documentation, `sphinx` must run the code. Indeed, as stated in the `rst` files of the `source` directory, `sphinx` will look for a package entitled `src`.

Thus a modification of `conf.py` is needed:

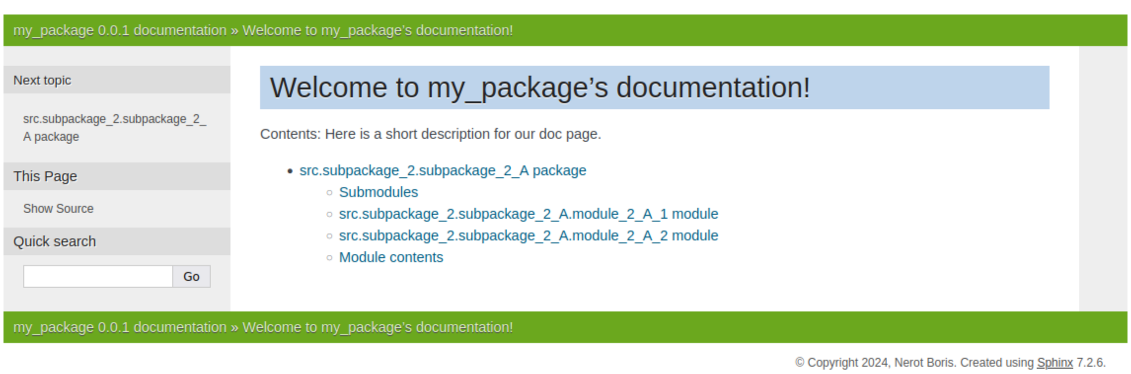
```
from sys import path
path.insert(0,  r'/absolute/path/to/dir/package_parent')
```

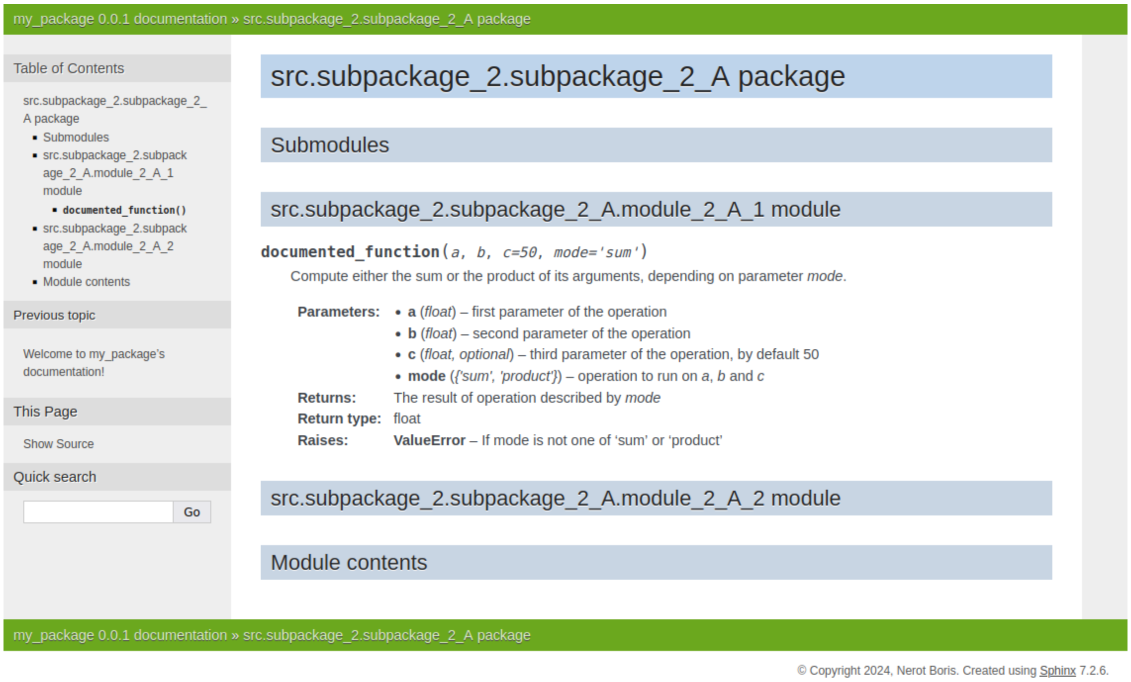
Then, back to commond prompt in the `doc` directory:

```
sphinx-build  -M html . _build/.
```

## 9.2.4 Result

HTML documentation is opened using `doc/_build/html/index.html`.






### 9.2.5 Notes

#### HTML customization

HTML documentation can be customized. For instance, one can change the theme by modifying the `conf.py` file:

```
html_theme = 'nature'.
```

All customisation options are described here.

#### PDF production

A PDF generation is also possible using `latex`:

- `sphinx-build  -M pdf . _build/`
- `cd _build/latex/`
- `make`

That process requires a valid Latex installation.

# Part IV

# Scientific Python

# Chapter 10

# Numpy

## 10.1 Introduction [easy]

### 10.1.1 Presentation

`numpy` is a Python package used in scientific computing. It handles numerical values with a mathematical approach.

`numpy` is **much faster** than native Python since most of `numpy` code is written in C (10 to 100 times faster).

`numpy` is stable, it benefits from a large online community.

### 10.1.2 When to use `numpy`?

You must use `numpy` for projects relying on large homogeneous data (i.e. same type data) or specific mathematical operations.

### 10.1.3 What not to do with `numpy`?

`numpy` is fast. Yet, one must avoid looping using `numpy`.In this part, some major `numpy` functions are presented, part of them make looping unnecessary.

## 10.2 Array [easy]

### 10.2.1 Introduction

An *array* is a numpy object (whose type is `numpy.ndarray`). It is similar to Python lists but suited for mathematical operations.

[1]:
```
import numpy as np
var = [1, 2, 3]
print(type(var))
arr = np.array([1, 2, 3])
print(type(arr))
```

```
<class 'list'>
<class 'numpy.ndarray'>
```

### 10.2.2 Create an array

One can create an array from an iterable (ex: list, see above) or use dedicated `numpy` functions:

[2]:
```
print(np.ones(5))
print(np.arange(2, 42, 5))      # similar to `range`
print(np.zeros(5))
```

```
[1. 1. 1. 1. 1.]
[ 2  7 12 17 22 27 32 37]
[0. 0. 0. 0. 0.]
```

Functions `linspace` and `logspace` define regularly spaced values in a linear space or logarthmic space.

Using `linspace`: the difference between two consecutive elements is constant:

[3]:
```
arr = np.linspace(1, 10, 5)
print(arr)
print(arr[1]-arr[0])
print(arr[2]-arr[1])
```

```
[ 1.    3.25  5.5   7.75 10.  ]
2.25
2.25
```

Using `logspace`: the ratio of two consecutive elements is constant:

[4]:
```
arr = np.logspace(1, 10, 5)
print(arr)
print(arr[1]/arr[0])
print(arr[2]/arr[1])
```

```
[1.00000000e+01 1.77827941e+03 3.16227766e+05 5.62341325e+07
 1.00000000e+10]
```

```
177.82794100389228
177.82794100389225
```

### 10.2.3 Important ideas: axis and dimension

#### Introduction

Usually, the length of an iterable is it's number of elements, given by the length function `len`. For an array, there may be **more than one dimension**. Below is a 2-dimensionnal array:

- 2 lines
- 3 columns

[5]:
```python
arr = np.array([[1, 2, 3], [4, 5, 6]])
print(arr)
print(arr.ndim)
```

```
[[1 2 3]
 [4 5 6]]
2
```

The shape of this array is (2, 3) because:

- it has 2 elements along dimension 1
- it has 3 elements along dimension 2

[6]:
```python
print(arr.shape)
```

```
(2, 3)
```

Instead of "dimension", `numpy` uses the term **"axis"**.

#### Modify the shape

`shape` gives the actual size of an `array`. But one can change this shape using `reshape`:

[7]:
```python
arr = arr.reshape((3, 2))    # 3 rows, 2 columns: still 6 elements,
                             # hence reshape is possible
print(arr)
```

```
[[1 2]
 [3 4]
 [5 6]]
```

If you don't want several dimensions, the `flatten` method will return all elements along one dimension.

[8]:
```python
arr2 = arr.flatten()
print(arr2)            # all values along a single dimension
```

```
[1 2 3 4 5 6]
```

Note that `flatten` returns a copy, hence if `arr2` is modified `arr` will remain the same.

[9]:
```
arr2[3] = 100            # resulting array is modified
print(arr)               # this does not change original array
```

```
[[1 2]
 [3 4]
 [5 6]]
```

Conversely, `ravel` performs the same than `flatten` but uses the same underlying data:

[10]:
```
arr2 = arr.ravel()
print(arr2)              # all values along a single dimension
arr2[3] = 100            # resulting array is modified
print(arr)               # this changes the original array because data is shared␣
 ↪in memory
```

```
[1 2 3 4 5 6]
[[  1   2]
 [  3 100]
 [  5   6]]
```

### Handle axes

Axes of a multidimensionnal array start from 0 (and goes to `ndim-1`). One can index an array following a specific axis the same way it is done for lists.

[23]:
```
arr = np.zeros((4, 4))
print(arr)
arr[1:3, :2] = 99                # same value for some indexes
print(arr)
arr[2:, 3:4] = [[34], [35]]   # an iterable of the same shape as the modified␣
 ↪subarray (shape (2, 1))
print(arr)
```

```
[[0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]]
[[ 0.  0.  0.  0.]
 [99. 99.  0.  0.]
 [99. 99.  0.  0.]
 [ 0.  0.  0.  0.]]
[[ 0.  0.  0.  0.]
 [99. 99.  0.  0.]
 [99. 99.  0. 34.]
 [ 0.  0.  0. 35.]]
```

Let's create a 3-dimensionnal array:

- `axis` 0 has 4 éléments
- `axis` 1 has 3 éléments
- `axis` 2 has 2 éléments

[12]:
```
arr = np.arange(4 * 3 * 2).reshape((4, 3, 2))
print(arr)
```

```
[[[ 0  1]
  [ 2  3]
  [ 4  5]]

 [[ 6  7]
  [ 8  9]
  [10 11]]

 [[12 13]
  [14 15]
  [16 17]]

 [[18 19]
  [20 21]
  [22 23]]]
```

Let's extract the elements whose coordinates match:

- 0, 1 or 3 on first axis
- 2 on second axis
- whatever on third axis

[13]:
```
arr[[0, 1, 3], 2, :]
```

[13]:
```
array([[ 4,  5],
       [10, 11],
       [22, 23]])
```

**Many `numpy` methods** take an optional argument `axis` to specify where the mathematical operation must be performed.

For instance, let's compute a mean over axis 1.

[15]:
```
result = arr.mean(axis=1)
result
```

[15]:
```
array([[ 2.,  3.],
       [ 8.,  9.],
       [14., 15.],
       [20., 21.]])
```

Above, **using `axis=1`, `numpy` takes the mean of all elements whose coordinates are all equal, except for axis 1**.

Hence, `result[2, 1]` (3rd row, 1st column) is the mean of:

- `arr[2, 0, 1]` (13)
- `arr[2, 1, 1]` (15)
- `arr[2, 2, 1]` (17)

### Concatenate some arrays

`np.concatenate` gather different arrays into a single instance. Their dimensions must be compatible:

[14]:
```
arr1 = np.arange(16).reshape((2, 8))
arr2 = np.arange(16, 32).reshape((2, 8))
print(arr1)
print(arr2)
```

```
[[ 0  1  2  3  4  5  6  7]
 [ 8  9 10 11 12 13 14 15]]
[[16 17 18 19 20 21 22 23]
 [24 25 26 27 28 29 30 31]]
```

[15]:
```
print(np.concatenate([arr1, arr2], axis=0))    # shape of axis 0 is increased (i.
 ↪e: more elements)
```

```
[[ 0  1  2  3  4  5  6  7]
 [ 8  9 10 11 12 13 14 15]
 [16 17 18 19 20 21 22 23]
 [24 25 26 27 28 29 30 31]]
```

[16]:
```
print(np.concatenate([arr1, arr2], axis=1))    # shape of axis 1 is increased (i.
 ↪e: more elements)
```

```
[[ 0  1  2  3  4  5  6  7 16 17 18 19 20 21 22 23]
 [ 8  9 10 11 12 13 14 15 24 25 26 27 28 29 30 31]]
```

One can also gather arrays along a new dimension, using `np.stack`.

[14]:
```
arr1 = np.arange(16)
arr2 = np.arange(16, 32)
print(arr1)
print(arr2)
```

```
[ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15]
[16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31]
```

[15]:
```
print(np.stack([arr1, arr2], axis=0))       # `arr1` and `arr2` can be accessed␣
 ↪along axis 0
```

```
[[ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15]
 [16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31]]
```

[16]:
```
print(np.stack([arr1, arr2], axis=1))       # `arr1` and `arr2` can be accessed␣
 ↪along axis 1
```

```
[[ 0 16]
 [ 1 17]
 [ 2 18]
 [ 3 19]
 [ 4 20]
```

```
 [ 5 21]
 [ 6 22]
 [ 7 23]
 [ 8 24]
 [ 9 25]
 [10 26]
 [11 27]
 [12 28]
 [13 29]
 [14 30]
 [15 31]]
```

**Split some arrays**

You want to define different variables for som parts of an array? Use `np.split`.

[22]:
```
arr = np.arange(16).reshape((2, 8))
print(arr)
```

```
[[ 0  1  2  3  4  5  6  7]
 [ 8  9 10 11 12 13 14 15]]
```

Let's split `arr` along axis 1. `np.split` takes a list of indexes to specify where to split. By giving `[2, 4, 5]`, 4 sub arrays are defined, having these coordinates along axis 1:

- [0, 2[
- [2, 4[
- [4, 5[
- [5, 8[

[29]:
```
sub1, sub2, sub3, sub4  = np.split(arr, [2, 4, 5], axis=1)
print(sub1, sub2, sub3, sub4, sep='\n')
```

```
[[0 1]
 [8 9]]
[[ 2  3]
 [10 11]]
[[ 4]
 [12]]
[[ 5  6  7]
 [13 14 15]]
```

A different way is to ask for a fixed number of sub arrays, for instance 4.

[30]:
```
sub1, sub2, sub3, sub4  = np.split(arr, 4, axis=1)
print(sub1, sub2, sub3, sub4, sep='\n')
```

```
[[0 1]
 [8 9]]
[[ 2  3]
 [10 11]]
```

```
[[ 4  5]
 [12 13]]
[[ 6  7]
 [14 15]]
```

#### Modify dimensions

It may happen that an array has only one element along a specific axis. One can delete this dimensions using `np.squeeze`.

[24]:
```python
arr = np.arange(24).reshape((6, 1, 4))
print(arr)
print(arr.shape)
```

```
[[[ 0  1  2  3]]

 [[ 4  5  6  7]]

 [[ 8  9 10 11]]

 [[12 13 14 15]]

 [[16 17 18 19]]

 [[20 21 22 23]]]
(6, 1, 4)
```

[25]:
```python
arr = np.squeeze(arr, axis=1)
print(arr)
print(arr.shape)
```

```
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]
 [12 13 14 15]
 [16 17 18 19]
 [20 21 22 23]]
(6, 4)
```

Conversely, one can add dimensions using `np.expand_dims`.

[ ]:
```python
arr = np.arange(24).reshape((6, 4))
print(arr)
print(arr.shape)
```

[7]:
```python
arr = np.expand_dims(arr, axis=1)
print(arr)
print(arr.shape)
```

```
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]
 [12 13 14 15]
 [16 17 18 19]
 [20 21 22 23]]
(6, 4)
[[[ 0  1  2  3]]

 [[ 4  5  6  7]]

 [[ 8  9 10 11]]

 [[12 13 14 15]]

 [[16 17 18 19]]

 [[20 21 22 23]]]
(6, 1, 4)
```

### 10.2.4 Common operations

Contrary to lists, numpy arrays handle mathematical operations in a simple way.

[18]:
```
arr = np.arange(5)
```

[19]:
```
arr + arr
```

[19]:
```
array([0, 2, 4, 6, 8])
```

Operations on arrays are done ***element wise***.

[13]:
```
print(arr * arr)         # product
print(arr ** 2)          # power
print(np.exp(arr))       # some function
print((5*arr+9) % 14)    # modulo
```

```
[1 4 9]
[1 4 9]
[ 2.71828183  7.3890561  20.08553692]
[ 0  5 10]
[ True  True False]
```

Note that you can perform matric multiplication using `@`

[22]:
```
arr = np.array([[1, 2, 3], [4, 5, 6]])                              # shape is␣
 ↪(2, 3)
arr2 = np.array([[5, 6, 7, 8], [7, 8, 9, 10], [8, 9, 10, 11]])       # shape is␣
 ↪(3, 4)
```

```
arr @ arr2                                                     # shape is␣
↪(2, 4)
```

```
[22]: array([[ 43,  49,  55,  61],
             [103, 118, 133, 148]])
```

## 10.3 Basic operations [easy]

### 10.3.1 Conditions

Numpy arrays can be created according to a (boolean) condition.

For instance, let's define a set of temperature values (°C).

[3]:
```
import numpy as np
T = np.array([25, 27, 29, 24, 26, 18, 32])
```

Let's extract temperatures above 25°C:

[4]:
```
print(T > 25)
print(T[T > 25])
```

```
[False  True  True False  True False  True]
[27 29 26 32]
```

One can replace these values using `np.where`:

- whenever the condition holds True, replace the value with 30
- whenever the condition holds False, keep the value

[5]:
```
np.where(T>25, 30, T)
```

[5]:
```
array([25, 30, 30, 24, 30, 18, 30])
```

Advanced: note that `np.where` returns a copy of the object

Let's suppose now there are several temperature series:

[6]:
```
T = np.stack([T, T+1, T-3])
T
```

[6]:
```
array([[25, 27, 29, 24, 26, 18, 32],
       [26, 28, 30, 25, 27, 19, 33],
       [22, 24, 26, 21, 23, 15, 29]])
```

Let's replace the values of each serie for which the maximum is not 33°C. Hence, maximum of `T` along axis 1 is calculated (since series are stacked along axis 0).

[7]:
```
max_ = T.max(axis=1)
print(max_)
```

```
[32 33 29]
```

Then the condition is defined:

[8]:
```
cond = max_ == 33
print(cond)
```

```
[False  True False]
```

And it is used in `np.where`.

[37]:
```
np.where(cond, T, 0)
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Cell In[37], line 1
----> 1 np.where(cond, T, 0)

File <__array_function__ internals>:200, in where(*args, **kwargs)

ValueError: operands could not be broadcast together with shapes (3,) (3,7) ()
```

**Error! The condition `cond` does not have the same shape than replacing values (`T` and `0`)** Hence `numpy` cannot handle the condition.

This is because the maximum calculation removed one dimension. But this can be prevented using `keepdims=True`:

[9]:
```
max_ = T.max(axis=1, keepdims=True)
print(max_)                          # shape of dimension 1 is 1 instead of 7␣
 ↪(because max value is kept),
                                     # but at least this dimension is kept
```

```
[[32]
 [33]
 [29]]
```

[10]:
```
cond = max_ == 33
np.where(cond, T, 0)
```

[10]:
```
array([[ 0,  0,  0,  0,  0,  0,  0],
       [26, 28, 30, 25, 27, 19, 33],
       [ 0,  0,  0,  0,  0,  0,  0]])
```

Similarly: using `np.any`, one can look for series for which **at least one** value meets a criterium:

[11]:
```
cond = (T >= 32).any(axis=1, keepdims=True)
np.where(cond, T, 0)
```

[11]:
```
array([[25, 27, 29, 24, 26, 18, 32],
       [26, 28, 30, 25, 27, 19, 33],
       [ 0,  0,  0,  0,  0,  0,  0]])
```

With `np.all`, all values must satisfy the criterium:

[55]:
```
cond = (T < 32).all(axis=1, keepdims=True)
np.where(cond, T, 0)
```

```
[55]: array([[ 0,  0,  0,  0,  0,  0,  0],
             [ 0,  0,  0,  0,  0,  0,  0],
             [22, 24, 26, 21, 23, 15, 29]])
```

### 10.3.2 Find out duplicate values

`np.unique` eliminates values that occur several times:

```
[12]: rng = np.random.default_rng(42)
      arr = rng.integers(0, 2, (12, 3))
      arr
```

```
[12]: array([[0, 1, 1],
             [0, 0, 1],
             [0, 1, 0],
             [0, 1, 1],
             [1, 1, 1],
             [1, 1, 0],
             [1, 0, 1],
             [0, 0, 1],
             [1, 1, 0],
             [1, 1, 0],
             [0, 0, 0],
             [1, 1, 0]])
```

```
[13]: np.unique(arr, axis=0)      # there exists duplicated values along axis 0
```

```
[13]: array([[0, 0, 0],
             [0, 0, 1],
             [0, 1, 0],
             [0, 1, 1],
             [1, 0, 1],
             [1, 1, 0],
             [1, 1, 1]])
```

```
[14]: np.unique(arr, axis=1)      # no duplicated values along axis 1
```

```
[14]: array([[0, 1, 1],
             [0, 0, 1],
             [0, 1, 0],
             [0, 1, 1],
             [1, 1, 1],
             [1, 1, 0],
             [1, 0, 1],
             [0, 0, 1],
             [1, 1, 0],
             [1, 1, 0],
             [0, 0, 0],
```

```
       [1, 1, 0]])
```

[88]:
```
np.unique(arr)              # no other values than 0 and 1 in the array
```

[88]:
```
array([0, 1])
```

### 10.3.3 Element-wise maximum

`np.maximum` can calculate the maximum of several arrays **element-wise**.

note: this is very different from `np.max` that takes the maximum value of a single array.

[15]:
```
v1 = np.array([1, 2, 3, 4])
v2 = np.array([3, 2, 1, 0])
```

[16]:
```
_ = np.maximum(v1, v2)              # a new array is created to store the result
print(_)
```

```
[3 2 3 4]
```

**Advanced**: `np.maximum` is one of the `numpy` function that can allocate memory in an already-defined variable. This is done using parameter `out`. This is useful to modify the content of a variable that has already been passed to several different functions.

[18]:
```
out = np.full(4, 1000)       # an array of shape (4,) with constant value 1000
out
```

[18]:
```
array([1000, 1000, 1000, 1000])
```

[19]:
```
_ = np.maximum(v1, v2, out=out)
print(_ is out)                      # the result of np.maximum is stored in the␣
 ↪already defined `out`
out
```

```
True
```

[19]:
```
array([3, 2, 3, 4])
```

### 10.3.4 Index of maximum

Let's suppose `T` is an array with 1000 values. One can determine the index of the maximum value of `T` using `np.argmax`.

[20]:
```
T = np.sin(np.linspace(0, np.pi, 1000)) * 10
```

On peut trouver le pas de temps pour lequels la température est maximale.

[21]:
```
np.argmax(T)
```

[21]:
```
499
```

### 10.3.5 Vectorization of Python functions

Some Python operations have no meaning for `numpy`.

For instance, the function below is running fine for usual Python scalars:

[23]:
```
def f(a, b):
    if a < b:
        return a + b
    else:
        return a * b

f(3, 5)
```

[23]:
```
8
```

But it does not work with `numpy` arrays as comparison of 2 arrays using `<` is unclear for `numpy`: it does not know if the condition must be met for all elements (`numpy.all`) or at least one (`numpy.any`):

[24]:
```
arr1, arr2 = np.split(np.random.randint(1, 5, 16), 2)
print(arr1)
print(arr2)
f(arr1, arr2)
```

```
[3 2 3 2 4 3 3 2]
[2 3 3 3 2 3 2 2]
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Cell In[24], line 4
      2 print(arr1)
      3 print(arr2)
----> 4 f(arr1, arr2)

Cell In[23], line 2, in f(a, b)
      1 def f(a, b):
----> 2     if a < b:
      3         return a + b
      4     else:

ValueError: The truth value of an array with more than one element is ambiguous.
 ↪Use a.any() or a.all()
```

In this case, we want `f` to be applied element-wise. Thus best choice is to use `np.where`:

[26]:
```
np.where(arr1 < arr2,
         arr1 + arr2 ,
         arr1 * arr2)
```

[26]:
```
array([6, 5, 9, 5, 8, 9, 6, 4])
```

Another way is to use the `np.vectorize` function that make `f` element-wise for `numpy` arrays:

[27]:
```
vectorized_f = np.vectorize(f)
vectorized_f(arr1, arr2)
```

[27]:
```
array([6, 5, 9, 5, 8, 9, 6, 4])
```

**Advanced**: beware, `np.vectorize`:

- is a bad choice for performance
- define a function that makes hypotheses regarding data type on the first call

### 10.3.6 Multi-variables functions

Let's define a 3-variables function.

[39]:
```
def f(a, b, c):
    return (a + b) ** c

f(2, 3, 4)
```

[39]:
```
625
```

The function must be evaluated on the following values:

[40]:
```
A = (2, 3, 4)
B = (4, 5)
C = (5, 6)
```

#### Method 1

First choice is define several loops:

[41]:
```
results = {}
for a in A:
    for b in B:
        for c in C:
            results[(a, b, c)] = f(a, b, c)
print(results)
```

```
{(2, 4, 5): 7776, (2, 4, 6): 46656, (2, 5, 5): 16807, (2, 5, 6): 117649, (3, 4,
5): 16807, (3, 4, 6): 117649, (3, 5, 5): 32768, (3, 5, 6): 262144, (4, 4, 5):
32768, (4, 4, 6): 262144, (4, 5, 5): 59049, (4, 5, 6): 531441}
```

This solution is **very slow**.

#### Method 2

Instead, let's use `np.meshgrid` to create some evaluation grid:

[42]:
```
aa, bb, cc = np.meshgrid(A, B, C)
```

Then vectorize the `f` function and apply the vectorized version on the grid:

[43]:
```
vectorized_f = np.vectorize(f)
results = vectorized_f(aa, bb, cc)
print(results)
```

```
[[[  7776  46656]
  [ 16807 117649]
  [ 32768 262144]]

 [[ 16807 117649]
  [ 32768 262144]
  [ 59049 531441]]]
```

Results are the same as with method 1 but presented in a different way:

- along axis 2 (`axis=2`), values change according to `C`, with `a` and `b` being constant
- along axis 1 (`axis=1`), values change according to `B`, with `a` and `c` being constant
- along axis 0 (`axis=0`), values change according to `A`, with `b` and `c` being constant

Let's understand what is performed by `np.meshgrid` using a simpler 2D example:

[47]:
```
aa, bb = np.meshgrid(A, B)
print(aa)
print(bb)
```

```
[[2 3 4]
 [2 3 4]]
[[4 4 4]
 [5 5 5]]
```

`aa` and `bb` are two arrays of shape (`len(B)`, `len(A)`). `aa` contains the values of `A`, repeated as many times as needed (`len(B)` times). Same for `bb`.

Using values of both `aa` and `bb` gives an exhaustive grid to evaluate a 2D function.

[48]:
```
np.stack([aa, bb], axis=-1)
```

[48]:
```
array([[[2, 4],
        [3, 4],
        [4, 4]],

       [[2, 5],
        [3, 5],
        [4, 5]]])
```

## 10.4 Random [easy]

### 10.4.1 Create random numbers

The `random` subpackage of `numpy` can be used to generate random numbers:

- float values in $[0, 1[$
- integers

[2]:
```
import numpy.random as npr
```

[6]:
```
npr.random((2, 5))
```

[6]:
```
array([[0.81766694, 0.14319621, 0.24420405, 0.79485374, 0.36446199],
       [0.62801431, 0.16768836, 0.31869047, 0.91741838, 0.48900763]])
```

[7]:
```
npr.randint(1, 10, (8, 3))   # (8, 3) integers in [1, 10[ (10 excluded)
```

[7]:
```
array([[2, 1, 8],
       [4, 8, 4],
       [2, 6, 2],
       [1, 9, 2],
       [7, 7, 9],
       [1, 1, 8],
       [9, 9, 1],
       [4, 4, 2]])
```

Random numbers can also be generated following specific statistical distribution:

[8]:
```
print(npr.uniform(0, 5, (3, 4)))                       # uniform probability of a␣
↪number in [0, 5],
                                                       # with shape (3, 4)
print(npr.normal(loc=0, scale=5, size=(3, 4)))         # normal probability with mu=0␣
↪and std=5,
                                                       # with shape=(3, 4)
```

```
[[1.99681708 2.79904736 2.17633328 1.49729344]
 [4.11716777 4.92349861 0.21887788 0.75476233]
 [2.885842   1.27164751 3.66103527 1.56154697]]
[[ 7.26234311 -4.04165132 -0.50045676  1.06735817]
 [-3.21492431  0.59875201  4.56122662 -5.31195884]
 [-1.54579796 -5.27946634  4.55081793  1.30889338]]
```

### 10.4.2 Create deterministic random

The scientific approach needs reproducible computation steps. Whenever these steps imply random number generation, this can leads to problems: values differ from one execution to another.

[9]:
```
my_physical_variable = npr.random(4)
print(my_physical_variable)
```

```
[0.49496843 0.075648   0.99441966 0.66990802]
```

[10]:
```
my_physical_variable = npr.random(4)
print(my_physical_variable)
```

```
[0.44128815 0.48055089 0.03025358 0.06241068]
```

Fortunately, **one can create reproducible random**, i.e. a way to get the same random values whenever the code is ran.

To this purpose, one must define a random number generator and **initialize it** with the same initial *state* for all executions:

[11]:
```
rng = npr.default_rng(42)
my_physical_variable = rng.random(4)
print(my_physical_variable)
```

```
[0.77395605 0.43887844 0.85859792 0.69736803]
```

[12]:
```
rng_new = npr.default_rng(42)
my_physical_variable = rng_new.random(4)
print(my_physical_variable)
```

```
[0.77395605 0.43887844 0.85859792 0.69736803]
```

[66]:
```
rng_new = npr.default_rng(65)      # different seed
my_physical_variable = rng_new.random(4)
print(my_physical_variable)
```

```
[0.04739149 0.51822218 0.37485856 0.22867852]
```

Note that:

- defining the initial state returns a new instance that must be used to generate random numbers
- using this instance provides reproducible random numbers generation
- a different initial state gives different random numbers

## 10.5 Data types [medium]

### 10.5.1 Data types

#### What is a data type

The data type of an array gives `numpy` some information on how to deal with this array. Most common data types are:

- `int_`
- `float_`
- `str_`
- `bool_`

These types are a bit different from the ones of Python. They can be accessed using the `dtype` attribute (whereas Python type is given by `type(...)`)

A numpy array is always of type `numpy.ndarray`, but the dtype depends on its content:

[1]:
```
import numpy as np
arr = np.array([1, 2, 3])
print(type(arr))
print(arr.dtype)
```

```
<class 'numpy.ndarray'>
int64
```

#### Use data types

**Automatically assigned data type** In most cases, numpy will choose a dtype automatically. The chosen dtype is the one compatible with all the elements of the array.

[2]:
```
np.array([1, 2, 3]).dtype
```

[2]:
```
dtype('int64')
```

If one of the integer has a '.', Python thinks it's a float (even though decimal part is 0):

[3]:
```
np.array([1, 2, 3.]).dtype
```

[3]:
```
dtype('float64')
```

If some non-numeric values exist, the dtype is non-numeric and mathematical operations are impossible:

[4]:
```
arr = np.array(['azerty', 45, 98])
print(arr.dtype)
arr.sum()
```

```
<U21
```

```
---------------------------------------------------------------------------
UFuncTypeError                            Traceback (most recent call last)
```

```
Cell In[4], line 3
      1 arr = np.array(['azerty', 45, 98])
      2 print(arr.dtype)
----> 3 arr.sum()

File ~/Documents/5-Donnees_techniques/Python/310_base/lib/python3.10/site-packages/
 ↪numpy/core/_methods.py:49, in _sum(a, axis, dtype, out, keepdims, initial,␣
 ↪where)
     47 def _sum(a, axis=None, dtype=None, out=None, keepdims=False,
     48          initial=_NoValue, where=True):
---> 49     return umr_sum(a, axis, dtype, out, keepdims, initial, where)

UFuncTypeError: ufunc 'add' did not contain a loop with signature matching types␣
 ↪(dtype('<U21'), dtype('<U21')) -> None
```

**Change data type** One can change the data type using `astype`, by specifying one of these:

- a numpy dtype: object or string
- a Python type for which equivalent dtype exists in `numpy`

[ ]:
```
arr = np.array([1, 2, 3])
print(arr.dtype)
arr = arr.astype(np.float_)   # numpy dtype, specified as an object
print(arr.dtype)
```

[ ]:
```
arr = arr.astype(int)         # python type
print(arr.dtype)
```

[ ]:
```
arr = arr.astype('complex')   # numpy dtype, specified as a string
print(arr.dtype)
```

Modifying the *dtype* can change the data:

[ ]:
```
np.array([1, 2, 3.65]).astype(int)
```

*casting* is sometimes possible, for instance regarding boolean values:

[6]:
```
np.array([1, 2, 0]).astype(bool)
```

[6]:
```
array([ True,  True, False])
```

### 10.5.2 Working with `nan`

#### Definition

`nan` means 'not a number'. A `nan` value (`np.nan`) is used to describe:

- a missing or unknown value
- the result of an impossible mathematical operation

You must never deal with `np.nan` using equality tests (`==`): the preferred way is to use dedicated functions of `numpy`.

#### `nan` propagation

As `np.inf` (infinite), `nan` values propagate in mathematical operations:

[7]:
```
arr = np.arange(16).reshape((4,4)).astype(float)
arr[1, 2] = np.nan
arr
```

[7]:
```
array([[ 0.,  1.,  2.,  3.],
       [ 4.,  5., nan,  7.],
       [ 8.,  9., 10., 11.],
       [12., 13., 14., 15.]])
```

[8]:
```
arr.sum(axis=1)
```

[8]:
```
array([ 6., nan, 38., 54.])
```

`numpy.isnan()` returns a boolean describing which value is a nan. With `numpy.where` replacement is possible:

[9]:
```
cond = np.isnan(arr)
arr[cond] = 0
arr.sum(axis=1)
```

[9]:
```
array([ 6., 16., 38., 54.])
```

# Chapter 11

# Pandas

## 11.1 Introduction [easy]

### 11.1.1 Introduction

#### `numpy` limitations

`numpy` does not allow to:

- assign custom labels to data
- perform common database-like operations
- import/export easily data from the disk

#### About `pandas`

`pandas` is a "data analysis and manipulation tool". In the backend, `pandas` relies on `numpy`, which makes it fast for many operations.

`pandas` presents 2 different data containers:

- `DataFrame`: similar to a 2D numpy array with:
  - rows and columns labels
  - possibly heterogeneous data
- `Series`: similar to a 1D numpy array

#### `pandas` and notebooks

`pandas` objets have a pretty representation in notebooks: instead of printing them, just call them as the last statement of the cell.

### 11.1.2 Dataframe

[1]:
```
import pandas as pd
import numpy as np
```

Dataframes can be built in several ways. For instance, using a dictionary:

[2]:
```
data = {'Some integers': (1, 2, 3), 'Some booleans': (True, False, True), 'Some␣
↪strings': ('a', 'b', 'c')}
pd.DataFrame(data=data)
```

[2]:
```
   Some integers  Some booleans Some strings
0              1           True            a
1              2          False            b
2              3           True            c
```

Or an iterable:

[3]:
```
data = ((1, True, 'a'), (2, False, 'c'), (3, True, 'c'))
columns = ('Some integers', 'Some booleans', 'Some strings')
pd.DataFrame(data=data, columns=columns)
```

[3]:
```
   Some integers  Some booleans Some strings
0              1           True            a
1              2          False            c
2              3           True            c
```

One can notice an additional columns on the left side: this is **the index along axis 0**, or more simply “index”. Index along axis 1 is also called “columns”.

One can specify the index values:

[4]:
```
data = ((1, True, 'a'), (2, False, 'b'), (3, True, 'c'))
columns = ('Some integers', 'Some booleans', 'Some strings')
index = ('first row', 'second row', 'third row')
df = pd.DataFrame(data=data, columns=columns, index=index)
df
```

[4]:
```
            Some integers  Some booleans Some strings
first row               1           True            a
second row              2          False            b
third row               3           True            c
```

### 11.1.3 Series

A `Series` object is similar to a `DataFrame` with one column. Instead of having ‘columns’, a `Series` has a `name` attribute.

[5]:
```
data = (True, False, True)
pd.Series(data=data, name='Some booleans', index=index)
```

[5]:
```
first row      True
second row    False
third row      True
Name: Some booleans, dtype: bool
```

### 11.1.4 Access data

*note: explanations below are for `DataFrame`, but similar behaviour is observed for `Series`.*

One can access data in a `DataFrame` in 2 ways:

- indexing: the same way one would do with a numpy array
- using labels

#### Using indexing

The important method is `iloc`, which is used using brackets `[]`:

- first value selects along axis 0
- second value selects along axis 1

[6]:
```
df
```

[6]:
```
            Some integers  Some booleans Some strings
first row               1           True            a
second row              2          False            b
third row               3           True            c
```

[7]:
```
df.iloc[1, 2]
```

[7]:
```
'b'
```

*slicing* is also possible. Let's extract:

- One row out of two from 0 to 3
- The last two columns

[8]:
```
df.iloc[0:3:2, -2:]
```

[8]:
```
           Some booleans Some strings
first row           True            a
third row           True            c
```

Another way is to specify directly a **list** of indexes:

[9]:
```
df.iloc[[0, 2], [-2, -1]]
```

[9]:
```
           Some booleans Some strings
first row           True            a
third row           True            c
```

Or a boolean indexer:

[10]:
```
df.iloc[[True, False, True], [False, True, True]]
```

[10]:
```
           Some booleans Some strings
first row           True            a
third row           True            c
```

Beware! The type of returned object depend on the way indexing is done:

[11]:
```
print(type(df.iloc[0:3:2, -1:]))    # every column from -1 to the end -->␣
↪DataFrame
```

```
<class 'pandas.core.frame.DataFrame'>
```

[12]:
```
print(type(df.iloc[0:3:2, -1]))    # specifically the last column --> Series
```

```
<class 'pandas.core.series.Series'>
```

[13]:
```
print(type(df.iloc[0:3:2, [-1]]))    # the columns specified by one-element list␣
↪--> DataFrame
```

```
<class 'pandas.core.frame.DataFrame'>
```

As with `numpy`, `:` is used to get all the data along a specific axis:

[14]:
```
df.iloc[:, [-2, -1]]
```

[14]:
```
            Some booleans Some strings
first row            True            a
second row          False            b
third row            True            c
```

For columns (axis 1), one can also undefine the index in order to get all data.

[15]:
```
df.iloc[[0, 2]]
```

[15]:
```
           Some integers  Some booleans Some strings
first row              1           True            a
third row              3           True            c
```

### Using labels

Similarly to `iloc`, `loc` allows to access elements using their labels:

[16]:
```
df
```

[16]:
```
            Some integers  Some booleans Some strings
first row               1           True            a
second row              2          False            b
third row               3           True            c
```

[17]:
```
df.loc['first row', 'Some strings']
```

[17]:
```
'a'
```

[18]:
```
df.loc[['first row', 'third row'], 'Some strings']
```

[18]:
```
first row     a
third row     c
Name: Some strings, dtype: object
```

If axis=0 does not matter, simple brackets [] can be used to access columns.

Hereafter, the two solutions are equivalent:

[19]:
```python
df.loc[:, ['Some booleans', 'Some strings']]
df[['Some booleans', 'Some strings']]
```

[19]:
```
            Some booleans Some strings
first row            True            a
second row          False            b
third row            True            c
```

Note: to modify a value, one must **always** :

- use loc or iloc
- specify the 2 coordinates in a single call.

[20]:
```python
df.loc['first row', 'Some integers'] = 42
df
```

[20]:
```
            Some integers  Some booleans Some strings
first row              42           True            a
second row              2          False            b
third row               3           True            c
```

Else, a _warning_ is raised.

[21]:
```python
df.loc['third row'].iloc[2] = 'new_string'   # warning is raised
df
```

```
/tmp/ipykernel_19950/3522855576.py:1: FutureWarning: ChainedAssignmentError:
behaviour will change in pandas 3.0!
You are setting values through chained assignment. Currently this works in
certain cases, but when using Copy-on-Write (which will become the default
behaviour in pandas 3.0) this will never work to update the original DataFrame
or Series, because the intermediate object on which we are setting values will
behave as a copy.
A typical example is when you are setting values in a column of a DataFrame,
like:

df["col"][row_indexer] = value

Use `df.loc[row_indexer, "col"] = values` instead, to perform the assignment in
a single step and ensure this keeps updating the original `df`.

See the caveats in the documentation: https://pandas.pydata.org/pandas-
docs/stable/user_guide/indexing.html#returning-a-view-versus-a-copy
```

```
  df.loc['third row'].iloc[2] = 'new_string'   # warning is raised
/tmp/ipykernel_19950/3522855576.py:1: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame

See the caveats in the documentation: https://pandas.pydata.org/pandas-
docs/stable/user_guide/indexing.html#returning-a-view-versus-a-copy
  df.loc['third row'].iloc[2] = 'new_string'   # warning is raised
```

[21]:

```
            Some integers  Some booleans Some strings
first row              42           True            a
second row              2          False            b
third row               3           True            c
```

### 11.1.5 Modify index

#### From existing data

`set_index` method replaces the current index with a columns of the DataFrame and then:

- deletes the column if `drop` = True (defaut)
- keeps the column if `drop` = False

[22]:

```
df
```

[22]:

```
            Some integers  Some booleans Some strings
first row              42           True            a
second row              2          False            b
third row               3           True            c
```

[23]:

```
df.set_index('Some booleans', drop=False)
```

[23]:

```
               Some integers  Some booleans Some strings
Some booleans
True                      42           True            a
False                      2          False            b
True                       3           True            c
```

[24]:

```
df.set_index('Some booleans')
```

[24]:

```
               Some integers Some strings
Some booleans
True                      42            a
False                      2            b
True                       3            c
```

#### Using integers

`reset_index` method replaces the current index by integers. It:

- deletes the current index if `drop` = True (defaut)
- keeps the current index as a column if `drop` = False

[25]:
```python
df.reset_index(drop=False)
```

[25]:
```
        index  Some integers  Some booleans Some strings
0   first row             42           True            a
1  second row              2          False            b
2   third row              3           True            c
```

If the index had a name, the resulting column keeps this name.

[26]:
```python
df.index.name = 'my_index'
df.reset_index(drop=False)
```

[26]:
```
     my_index  Some integers  Some booleans Some strings
0   first row             42           True            a
1  second row              2          False            b
2   third row              3           True            c
```

### 11.1.6 Iteration

Iteration over a dataframe is pretty slow but still made possible using dedicated methods:

#### Over rows

The `iterrows` method is used to iterate over rows, including index.

[27]:
```python
for index_value, (integer, string) in df[['Some integers', 'Some strings']].
↪iterrows():
    print(index_value, integer, string)
```

```
first row 42 a
second row 2 b
third row 3 c
```

#### Over columns

Let's use `items`.At each iteration, this method returns:

- the column name
- the column data as a `Series` instance

[28]:
```python
for column_name, column in df[['Some integers', 'Some strings']].items():
    print(column_name, '\n\n', column)
```

```
Some integers

 my_index
first row     42
second row     2
```

```
third row      3
Name: Some integers, dtype: int64
Some strings

 my_index
first row     a
second row    b
third row     c
Name: Some strings, dtype: object
```

### 11.1.7 Basic operations

**Similar to numpy**

**Many numpy operations look similar with pandas.**

[29]:
```
data = np.arange(16).reshape((4, 4))
df = pd.DataFrame(data=data, columns=['A', 'B', 'C', 'D'])
df
```

[29]:

| | A | B | C | D |
|---|---|---|---|---|
| 0 | 0 | 1 | 2 | 3 |
| 1 | 4 | 5 | 6 | 7 |
| 2 | 8 | 9 | 10 | 11 |
| 3 | 12 | 13 | 14 | 15 |

The agg method perform element-wise operations:

[30]:
```
df['E'] = df['B'].agg(lambda x: x if x%5==0 else 42)
df
```

```
/tmp/ipykernel_19950/1286485336.py:1: FutureWarning: using <function <lambda> at
0x7975a0084220> in Series.agg cannot aggregate and has been deprecated. Use
Series.transform to keep behavior unchanged.
  df['E'] = df['B'].agg(lambda x: x if x%5==0 else 42)
```

[30]:

| | A | B | C | D | E |
|---|---|---|---|---|---|
| 0 | 0 | 1 | 2 | 3 | 42 |
| 1 | 4 | 5 | 6 | 7 | 5 |
| 2 | 8 | 9 | 10 | 11 | 42 |
| 3 | 12 | 13 | 14 | 15 | 42 |

Yet, the same result could be achieved using where.

[31]:
```
cond = df['B']%5==0
df['E'] = df['B']                    # values of B if cond is True
df['E'] = df['E'].where(cond, 42)    # 42 if cond is False
                                     # (opposite to numpy where
                                     # first arg corresponds to positive cond)
df
```

[31]:

| | A | B | C | D | E |
|---|---|---|---|---|---|
| 0 | 0 | 1 | 2 | 3 | 42 |
| 1 | 4 | 5 | 6 | 7 | 5 |
| 2 | 8 | 9 | 10 | 11 | 42 |
| 3 | 12 | 13 | 14 | 15 | 42 |

### Sorting data

One can use the `sort_values` and `sort_index` methods:

[32]:
```
data = np.random.randint(0, 10, (5, 5))
df = pd.DataFrame(data=data, columns=('c1', 'c2', 'c3', 'c4', 'c5'), index=('e',␣
↪'b', 'c', 'a', 'd'))
df
```

[32]:

| | c1 | c2 | c3 | c4 | c5 |
|---|---|---|---|---|---|
| e | 8 | 9 | 6 | 9 | 7 |
| b | 3 | 7 | 6 | 7 | 2 |
| c | 5 | 0 | 7 | 3 | 4 |
| a | 8 | 4 | 3 | 3 | 2 |
| d | 9 | 7 | 7 | 8 | 2 |

[33]:
```
df.sort_index()
```

[33]:

| | c1 | c2 | c3 | c4 | c5 |
|---|---|---|---|---|---|
| a | 8 | 4 | 3 | 3 | 2 |
| b | 3 | 7 | 6 | 7 | 2 |
| c | 5 | 0 | 7 | 3 | 4 |
| d | 9 | 7 | 7 | 8 | 2 |
| e | 8 | 9 | 6 | 9 | 7 |

Let's sort `df` according to:

- column `c2`
- descending order

[34]:
```
df.sort_values(by='c2', ascending=False)
```

[34]:

| | c1 | c2 | c3 | c4 | c5 |
|---|---|---|---|---|---|
| e | 8 | 9 | 6 | 9 | 7 |
| b | 3 | 7 | 6 | 7 | 2 |
| d | 9 | 7 | 7 | 8 | 2 |
| a | 8 | 4 | 3 | 3 | 2 |
| c | 5 | 0 | 7 | 3 | 4 |

Let's sort following axis 1!

- row `b`
- custom key

[36]:
```
# using `key`, the closest the values are to 5 the sooner
# they come in the DataFrame
df.sort_values(by='b', axis=1, key=lambda x: abs(x-5))
```

[36]:
```
   c1  c3  c4  c2  c5
e   6   4   3   5   8
b   6   8   8   1   9
c   3   4   0   2   9
a   9   2   4   7   4
d   9   8   0   9   5
```

### 11.1.8 Manage *dtype*

You must **always** check that the data type is the expected one, because it defines the possible operations. The `astype` method makes it possible to change the data type of a column:

[37]:
```
data = (('01', True, 'a'), ('02', False, 'c'), ('03', True, 'c'))
columns = ('Some integers', 'Some booleans', 'Some strings')
df = pd.DataFrame(data=data, columns=columns, index=('first row', 'second row',␣
 ↪'third row'))
print(df.dtypes)
df
```

```
Some integers    object
Some booleans      bool
Some strings     object
dtype: object
```

[37]:
```
           Some integers  Some booleans Some strings
first row             01           True            a
second row            02          False            c
third row             03           True            c
```

[38]:
```
df['Some integers'] = df['Some integers'].astype(int)
print(df.dtypes)
df
```

```
Some integers     int64
Some booleans      bool
Some strings     object
dtype: object
```

[38]:
```
            Some integers  Some booleans Some strings
first row               1           True            a
second row              2          False            c
third row               3           True            c
```

## 11.2 IO [easy]

### 11.2.1 Introduction

`pandas` can write and read data files from the disk.

A very common file format is CSV. A CSV file is a text file whose columns are separated by `,`. Sometimes, this is not a comma but another character: `;`, `/`, etc…

When reading a file, `pandas` try to guess the data types.

### 11.2.2 Read a CSV file

Let's define a CSV file `data.csv` and use the `read_csv` function:

```
Integer;Some value;Another column;Date
1;3.45;True;21/02/2026 12:56:54
5;2.75;False;21/02/2026 14:25:08
2;4.15;False;21/02/2026 16:56:41
```

[7]:
```python
import pandas as pd
df = pd.read_csv('data.csv', index_col=0, sep=';')
df
```

[7]:
```
         Some value  Another column                 Date
Integer
1              3.45            True  21/02/2026 12:56:54
5              2.75           False  21/02/2026 14:25:08
2              4.15           False  21/02/2026 16:56:41
```

A quick dtype check:

[8]:
```python
df.dtypes
```

[8]:
```
Some value        float64
Another column       bool
Date               object
dtype: object
```

The dates have not been interprated as dates but as strings.

We need an additional argument in `read_csv` to specify the date format:

[17]:
```python
df['Date'] = pd.to_datetime(df['Date'], format='%d/%m/%Y %H:%M:%S')
df
```

[17]:
```
         Some value  Another column                Date
Integer
1              3.45            True 2026-02-21 12:56:54
5              2.75           False 2026-02-21 14:25:08
2              4.15           False 2026-02-21 16:56:41
```

[19]:
```
df.dtypes
```

[19]:
```
Some value                  float64
Another column                 bool
Date             datetime64[ns]
dtype: object
```

Many other arguments exist for `read_csv`:

- `names`: define new names for the columns, instead of those of the file
- `index_col`: select the column to use as index (axis 0)
- `comment`: tell pandas no to care about rows starting with a specific character: these are not data
- `skiprows`, `skipfooter`, `nrows`: read only specific rows

### 11.2.3 Write a CSV file

The `to_csv` method of `DataFrame` and `Serie` is used hereafter.

[24]:
```
df['Last column'] = df['Some value'] / 8
df.to_csv('data_write.csv', sep=',',
          index=False) # do not write index
```

Result:

```
Some value,Another column,Date,Last column
3.45,True,2026-02-21 12:56:54,0.43125
2.75,False,2026-02-21 14:25:08,0.34375
4.15,False,2026-02-21 16:56:41,0.51875
```

### 11.2.4 Other file formats

`pandas` handle a lot of IO file formats: Excel, json, html, hdf, etc…

The dedicated methods are called `read_[...]` (data import) and `to_[...]` (data export).

## 11.3 Dates [medium]

### 11.3.1 Introduction

`pandas` can handle dates in several ways:

- comparison
- extraction of day, hour, etc…
- addition of delays
- etc…

Date objects in `pandas` are the one of `numpy`, which differ a bit from those of native Python but are nonetheless compatible with them.

### 11.3.2 Useful functions

#### Create a dates series

`pd.date_range` expects 3 out of 4 of the following parameters to be specified:

- `start`: (date) first date
- `end`: (date) last date
- `periods` (integer): number of values
- `freq`: elapsed time between 2 consecutive dates. Accepted values are given here.

[1]:
```
import pandas as pd
# 7 values, once every 2 hours, from 21st of January 2023 at 30s past midnight
pd.date_range(start='21/01/2023 00:00:30', freq='2h', periods=7)
```

[1]:
```
DatetimeIndex(['2023-01-21 00:00:30', '2023-01-21 02:00:30',
               '2023-01-21 04:00:30', '2023-01-21 06:00:30',
               '2023-01-21 08:00:30', '2023-01-21 10:00:30',
               '2023-01-21 12:00:30'],
              dtype='datetime64[ns]', freq='2H')
```

[2]:
```
# a value every 3 days, from 21st of January 2023 to 4 of February
pd.date_range(start='21/01/2023 00:00:00', end='04/02/2023 00:00:00', freq='3D')
```

[2]:
```
DatetimeIndex(['2023-01-21', '2023-01-24', '2023-01-27', '2023-01-30',
               '2023-02-02', '2023-02-05', '2023-02-08', '2023-02-11',
               '2023-02-14', '2023-02-17', '2023-02-20', '2023-02-23',
               '2023-02-26', '2023-03-01', '2023-03-04', '2023-03-07',
               '2023-03-10', '2023-03-13', '2023-03-16', '2023-03-19',
               '2023-03-22', '2023-03-25', '2023-03-28', '2023-03-31'],
              dtype='datetime64[ns]', freq='3D')
```

[3]:
```
# 3 values from 21st of January 2023 to 4 of February
from datetime import datetime, timedelta
start = datetime(day=21, month=1, year=2023)
end = datetime(day=4, month=2, year=2023)
pd.date_range(start=start, end=end, periods=3)
```

```
[3]: DatetimeIndex(['2023-01-21', '2023-01-28', '2023-02-04'],
     dtype='datetime64[ns]', freq=None)
```

With `pandas`, one can specify dates in 3 different ways:

- A string

  Very handy but beware of bad interpreation done by `pandas`. In particular, the English way of processing date is to write the month before the day (ex: 21th june –> 06/21).

- Python date-like objects: `datetime.datetime`, `datetime.timedelta`

- `numpy` date-like objects

- `pandas` date-like objects: `pd.TimeStamp`, `pd.TimeDelta`

### Resample temporal data

Whenever temporal data comes with a too high or too low frequency, the `pd.resample` functions can modify this frequency to ease processing. There exists two cases:

- *upsampling*: additional values are added between current values, hence frequency is increased
- *downsampling*: existing values are aggragated following a specific rule, hence frequency is decreased

**Upsampling** Here after, a DataFrame that has a datetime index: six values, one every 2 hours, from 15th of August.

```
[4]: df = pd.DataFrame({'A': range(6), 'B': range(6, 12)},
                       index=pd.date_range(start='15/08/2024 00:00:00', freq='2h',␣
      ↪periods=6))
     df
```

```
[4]:                      A   B
     2024-08-15 00:00:00  0   6
     2024-08-15 02:00:00  1   7
     2024-08-15 04:00:00  2   8
     2024-08-15 06:00:00  3   9
     2024-08-15 08:00:00  4  10
     2024-08-15 10:00:00  5  11
```

Let's resample the DataFrame with one value every 50 min:

```
[5]: rs = df.resample('50min')
```

`rs` is an instance of type `Resampler`. It must be used with a rule qui doit être appelé avec une règle that defines what to do with unexistant data. For instance `ffill`, for *'forward fill'*, fill missing values with the previous existing value.

```
[6]: rs.ffill()
```

[6]:
```
                     A   B
2024-08-15 00:00:00  0   6
2024-08-15 00:50:00  0   6
2024-08-15 01:40:00  0   6
2024-08-15 02:30:00  1   7
2024-08-15 03:20:00  1   7
2024-08-15 04:10:00  2   8
2024-08-15 05:00:00  2   8
2024-08-15 05:50:00  2   8
2024-08-15 06:40:00  3   9
2024-08-15 07:30:00  3   9
2024-08-15 08:20:00  4  10
2024-08-15 09:10:00  4  10
2024-08-15 10:00:00  5  11
```

**Downsampling** Let's reduce the frequency from 2h to 4h. Values are averaged.

[8]:
```
rs = df.resample('4h')
rs.mean()
```

[8]:
```
                       A     B
2024-08-15 00:00:00  0.5   6.5
2024-08-15 04:00:00  2.5   8.5
2024-08-15 08:00:00  4.5  10.5
```

### 11.3.3 Indexing

With a temporal index, one can use slicing methods but with date-like instances:

[7]:
```
start = datetime(day=15, month=8, year=2024, hour=3)
end = start + timedelta(hours=5)
print(start, end, sep='\n')
```

```
2024-08-15 03:00:00
2024-08-15 08:00:00
```

[10]:
```
df.loc[start:end]    # everything from start to end
                     # both are included since `loc` is label-based
```

[10]:
```
                     A   B
2024-08-15 04:00:00  2   8
2024-08-15 06:00:00  3   9
2024-08-15 08:00:00  4  10
```

### 11.3.4 `dt` accessor

When temporal data is stored in a column, the **`dt` accessor** provides date-specific methods:

[8]:
```
df.index.name = 'date'
df = df.reset_index()
df
```

[8]:
```
                 date  A   B
0 2024-08-15 00:00:00  0   6
1 2024-08-15 02:00:00  1   7
2 2024-08-15 04:00:00  2   8
3 2024-08-15 06:00:00  3   9
4 2024-08-15 08:00:00  4  10
5 2024-08-15 10:00:00  5  11
```

Let's have a look to all the methods and attributes of the `dt` object:

[12]:
```
# listing of attributes and methods of object dt
for attr in dir(df['date'].dt):
    if not attr.startswith('_'):
        print(attr, end=' / ')
```

```
ceil / date / day / day_name / day_of_week / day_of_year / dayofweek / dayofyear
/ days_in_month / daysinmonth / floor / freq / hour / is_leap_year /
is_month_end / is_month_start / is_quarter_end / is_quarter_start / is_year_end
/ is_year_start / isocalendar / microsecond / minute / month / month_name /
nanosecond / normalize / quarter / round / second / strftime / time / timetz /
to_period / to_pydatetime / tz / tz_convert / tz_localize / week / weekday /
weekofyear / year /
```

For instance, one can get the hour of the day of a date-like column:

[13]:
```
df['Hour'] = df['date'].dt.hour
df
```

[13]:
```
                 date  A   B  Hour
0 2024-08-15 00:00:00  0   6     0
1 2024-08-15 02:00:00  1   7     2
2 2024-08-15 04:00:00  2   8     4
3 2024-08-15 06:00:00  3   9     6
4 2024-08-15 08:00:00  4  10     8
5 2024-08-15 10:00:00  5  11    10
```

### 11.3.5 TimeDelta

`pandas` can also handle date differences.

Hereafter, the first date is substracted from the other. This operation returns the durations from this initial date, for all elements of `df['date']` (because `df['date']` is chronologically sorted).

[9]:
```
df['date'] - df.loc[0, 'date']
```

[9]:
```
0   0 days 00:00:00
1   0 days 02:00:00
2   0 days 04:00:00
3   0 days 06:00:00
4   0 days 08:00:00
5   0 days 10:00:00
Name: date, dtype: timedelta64[ns]
```

## 11.4 Strings [medium]

### 11.4.1 Introduction

Similarly to the `dt` accessor that can handle dates, a `Series` containing strings can be managed using the `str` accessor.

[1]:
```
import pandas as pd
df = pd.DataFrame({'text': ['aCag', '53Bc^', 'cc', '/c_8cd45', 'F98', '__m'],
                   'other column': range(6)})
df
```

[1]:

| | text | other column |
|---|---|---|
| 0 | aCag | 0 |
| 1 | 53Bc^ | 1 |
| 2 | cc | 2 |
| 3 | /c_8cd45 | 3 |
| 4 | F98 | 4 |
| 5 | __m | 5 |

### 11.4.2 Existing methods

Below are presented methods and attributes of the `str` accessor

[2]:
```
# listing of attributes and methods of object dt
for attr in dir(df['text'].str):
    if not attr.startswith('_'):
        print(attr, end=' / ')
```

```
capitalize / casefold / cat / center / contains / count / decode / encode /
endswith / extract / extractall / find / findall / fullmatch / get / get_dummies
/ index / isalnum / isalpha / isdecimal / isdigit / islower / isnumeric /
isspace / istitle / isupper / join / len / ljust / lower / lstrip / match /
normalize / pad / partition / removeprefix / removesuffix / repeat / replace /
rfind / rindex / rjust / rpartition / rsplit / rstrip / slice / slice_replace /
split / startswith / strip / swapcase / title / translate / upper / wrap / zfill
/
```

Many of them also exist with the native Python `str` type. In other words, what you can do with a Python string can be done at large scale on a `pandas` Series containing strings:

[3]:
```
print(*[attr for attr in dir(str) if not attr.startswith('_')], sep=' / ')
```

```
capitalize / casefold / center / count / encode / endswith / expandtabs / find /
format / format_map / index / isalnum / isalpha / isascii / isdecimal / isdigit
/ isidentifier / islower / isnumeric / isprintable / isspace / istitle / isupper
/ join / ljust / lower / lstrip / maketrans / partition / removeprefix /
removesuffix / replace / rfind / rindex / rjust / rpartition / rsplit / rstrip /
split / splitlines / startswith / strip / swapcase / title / translate / upper /
zfill
```

### 11.4.3 Simple examples

[4]:
```
df['text'].str.lower()   # lower case
```

[4]:
```
0        acag
1       53bc^
2          cc
3    /c_8cd45
4         f98
5         __m
Name: text, dtype: object
```

[5]:
```
df['text'].str.len()   # length
```

[5]:
```
0    4
1    5
2    2
3    8
4    3
5    3
Name: text, dtype: int64
```

#### Indexing

[6]:
```
df['text'].str[2:4]
```

[6]:
```
0    ag
1    Bc
2
3    _8
4     8
5     m
Name: text, dtype: object
```

#### Splitting

Splitting means building different strings by cutting the original one at the location of a special character. Below, the splitting operation results in the substrings being stored in a list, for each row of the Series.

[7]:
```
df['text'].str.split('c')
```

[7]:
```
0          [aCag]
1        [53B, ^]
2          [, , ]
3    [/, _8, d45]
4           [F98]
5           [__m]
```

```
Name: text, dtype: object
```

The `expand` argument makes it possible to get distinct columns.

[8]:
```
df['text'].str.split('c', expand=True)
```

[8]:
```
      0     1     2
0  aCag  None  None
1   53B     ^  None
2
3     /    _8   d45
4   F98  None  None
5   __m  None  None
```

#### Suffixes and prefixes

[9]:
```
df['text'].str.startswith('53')
```

[9]:
```
0    False
1     True
2    False
3    False
4    False
5    False
Name: text, dtype: bool
```

[10]:
```
df['text'].str.endswith('m')
```

[10]:
```
0    False
1    False
2    False
3    False
4    False
5     True
Name: text, dtype: bool
```

### 11.4.4 Advanced: *regex*

#### Introduction

The ***regex*** word means *regular expression*. A regex is a group of characters built in a very specific order in order to describe a generic type of strings.

Regex can be used on very large amount of data to detect some strings with a particular meaning.

Documentation of the Python version of regex is accessible here.

**Regex is a difficult notion of computer engineering**. A very simple case is presented here after.

### Case study definition

Let's suppose one has some experimental values coming from different sensors.

[11]:
```
import numpy as np
npr = np.random.default_rng(42)
# `npr.choice` randomly takes 10 values from a certain iterable
sensors = npr.choice(('AB-45-PL', 'AB-46-KL', 'AB-47-KL', 'AB-48-KL',␣
↪'ZB-76-PM', '87-PA-98'), 10)
values = range(len(sensors))
df = pd.DataFrame({'sensors': sensors, 'values': values})
df
```

[11]:

| | sensors | values |
|---|---|---|
| 0 | AB-45-PL | 0 |
| 1 | ZB-76-PM | 1 |
| 2 | AB-48-KL | 2 |
| 3 | AB-47-KL | 3 |
| 4 | AB-47-KL | 4 |
| 5 | 87-PA-98 | 5 |
| 6 | AB-45-PL | 6 |
| 7 | ZB-76-PM | 7 |
| 8 | AB-46-KL | 8 |
| 9 | AB-45-PL | 9 |

**Question 1** How to access all sensors whose name contains both an 'A' and a '7'?

**Solution 1** Let's use the `str.contains` method, 2 times. We use the `&` (and) operator to assemble the two conditions.

[12]:
```
cond1 = df['sensors'].str.contains('A', regex=False)
cond2 = df['sensors'].str.contains('7', regex=False)
df[cond1 & cond2]
```

[12]:

| | sensors | values |
|---|---|---|
| 3 | AB-47-KL | 3 |
| 4 | AB-47-KL | 4 |
| 5 | 87-PA-98 | 5 |

**Question 2** How to get all sensors whose name contains:

- either 'A' or 'Z'
- then '7'

**Solution 2** Let's build a regex pattern:

[13]:
```
pattern = '.*(A|Z).*7.*[a-zA-Z]'
df[df['sensors'].str.contains(pattern, regex=True)]
```

```
/tmp/ipykernel_116620/2985439971.py:2: UserWarning: This pattern is interpreted
as a regular expression, and has match groups. To actually get the groups, use
str.extract.
  df[df['sensors'].str.contains(pattern, regex=True)]
```

[13]:

| | sensors | values |
|---|---|---|
| 1 | ZB-76-PM | 1 |
| 3 | AB-47-KL | 3 |
| 4 | AB-47-KL | 4 |
| 7 | ZB-76-PM | 7 |

Some explanations about the pattern:

- `.*` means: look for every possible characters
- `(A|Z)` means: look for either an ‘A’ or a ‘Z’
- 7 means: look for a ‘7’

Note that order matters: the ‘7’ must come after the ‘A’ or ‘Z’

## 11.5 Basic operations [medium]

### 11.5.1 Long and wide data format

There are several ways to store the same data.

#### Long format

[1]:
```
import pandas as pd
df_long = pd.DataFrame({'Animal': ('cat', 'cat', 'dog', 'dog', 'cow', 'cow'),
                        'Feature': ('Age', 'Mass', 'Age', 'Mass', 'Age', 'Mass'),
                        'Value': (11, 5, 8, 17, 4, 650)})
```

[2]:
```
df_long
```

[2]:

| | Animal | Feature | Value |
|---|---|---|---|
| 0 | cat | Age | 11 |
| 1 | cat | Mass | 5 |
| 2 | dog | Age | 8 |
| 3 | dog | Mass | 17 |
| 4 | cow | Age | 4 |
| 5 | cow | Mass | 650 |

Above, some data is stored using the **long format** relatively to column `Animal`. This means several rows have the sale 'Animal' value.

The **long format** :

- makes DataFrame having few comumns but many rows
- makes it difficult to work on specific values. For instance, how to perform calculation on the mass of all animals?

#### From long to wide format

Thus, let's transform the data to have it in **large format**. This is done using `pivot_table`.

[3]:
```
df_wide = df_long.pivot_table(index='Animal', columns='Feature', values='Value')
df_wide
```

[3]:

| Feature | Age | Mass |
|---|---|---|
| Animal | | |
| cat | 11.0 | 5.0 |
| cow | 4.0 | 650.0 |
| dog | 8.0 | 17.0 |

Above, `df_wide` has as many columns as there are different elements in the `Feature` column of `df_long`. The name 'Feature' is given to the index along axis 1, i.e. the columns.

[4]:
```
df_wide.columns.name
```

[4]: 'Feature'

#### From wide to long format

Conversely, the `melt` function makes it possible to transform data from a wide to a long format:

[5]:
```
df_wide = df_wide.reset_index()
df_wide.melt(id_vars='Animal', value_vars=['Age', 'Mass'],
             var_name='Feature', value_name='Value')
```

[5]:

| | Animal | Feature | Value |
|---|---|---|---|
| 0 | cat | Age | 11.0 |
| 1 | cow | Age | 4.0 |
| 2 | dog | Age | 8.0 |
| 3 | cat | Mass | 5.0 |
| 4 | cow | Mass | 650.0 |
| 5 | dog | Mass | 17.0 |

#### Advanced

In the previous example `df_long` has only one value `Value` for each (`Animal`, `Feature`) pair. Whenever it's not the case, a `aggfunc` must be specified when going from long to wide format.

Another DataFrame definition:

[6]:
```
df_long = pd.DataFrame({'Animal': ('cat', 'cat', 'cat', 'dog', 'dog', 'dog',␣
 ↪'cow', 'cow'),
                   'Feature': ('Age', 'Mass', 'Mass', 'Age', 'Mass', 'Mass',␣
 ↪'Age', 'Mass'),
                   'Value': (11, 5, 9, 8, 17, 11, 4, 650)})
df_long
```

[6]:

| | Animal | Feature | Value |
|---|---|---|---|
| 0 | cat | Age | 11 |
| 1 | cat | Mass | 5 |
| 2 | cat | Mass | 9 |
| 3 | dog | Age | 8 |
| 4 | dog | Mass | 17 |
| 5 | dog | Mass | 11 |
| 6 | cow | Age | 4 |
| 7 | cow | Mass | 650 |

`df_long` has now 2 masses for the cat and the dog. Let's define `aggfunc`:

[7]:
```
df_long.pivot_table(index='Animal', columns='Feature', values='Value',␣
 ↪aggfunc='mean')  # mean
```

[7]:

| Feature | Age | Mass |
|---|---|---|
| Animal | | |
| cat | 11.0 | 7.0 |
| cow | 4.0 | 650.0 |
| dog | 8.0 | 14.0 |

[8]:
```
df_long.pivot_table(index='Animal', columns='Feature', values='Value',␣
↪aggfunc=list)  # keep all values
```

[8]:
```
Feature    Age       Mass
Animal
cat       [11]     [5, 9]
cow        [4]      [650]
dog        [8]   [17, 11]
```

### 11.5.2 Index swapping

A DataFrame has two indexes:

- along rows (axis 0): can be accessed using `.index`
- along columns (axis 1): can be accessed using `.columns`

The `stack` method can append the column index to rows. `unstack` do the opposite.

stack

[9]:
```
df_wide
```

[9]:
```
Feature Animal   Age   Mass
0          cat  11.0    5.0
1          cow   4.0  650.0
2          dog   8.0   17.0
```

[10]:
```
df_wide_stacked = df_wide.stack()
df_wide_stacked
```

[10]:
```
   Feature
0  Animal       cat
   Age         11.0
   Mass         5.0
1  Animal       cow
   Age          4.0
   Mass       650.0
2  Animal       dog
   Age          8.0
   Mass        17.0
dtype: object
```

Since there was only one level of columns, the call to `stack` returns a `Serie`.

[11]:
```
type(df_wide_stacked)
```

[11]:
```
pandas.core.series.Series
```

And since there already was an index, there are now 2 of them (multi index):

[12]:
```
df_wide_stacked.index
```

[12]:
```
MultiIndex([(0, 'Animal'),
            (0,    'Age'),
            (0,   'Mass'),
            (1, 'Animal'),
            (1,    'Age'),
            (1,   'Mass'),
            (2, 'Animal'),
            (2,    'Age'),
            (2,   'Mass')],
           names=[None, 'Feature'])
```

Multi index can be accessed this way:

[13]:
```
df_wide_stacked.loc[(1, 'Age')]
```

[13]:
```
4.0
```

`unstack`

Using `unstack`, the row index becomes a columns index. Thus, the multi index is now at the column level and there is no more index at the row level:

[14]:
```
df_wide
```

[14]:
```
Feature Animal   Age   Mass
0          cat  11.0    5.0
1          cow   4.0  650.0
2          dog   8.0   17.0
```

[15]:
```
df_wide_unstacked = df_wide.unstack()
df_wide_unstacked
```

[15]:
```
Feature
Animal   0      cat
         1      cow
         2      dog
Age      0     11.0
         1      4.0
         2      8.0
Mass     0      5.0
         1    650.0
         2     17.0
dtype: object
```

#### Use case

`stack` and `unstack` are very powerful whenever the DataFrame has an index (rows or columns) with more than one level.

#### DataFrame definition

[16]:
```
import numpy as np
index_data_rows = [[1, 1, 2, 2, 3], ['x', 'y', 'x', 'y', 'z']]
index_rows = pd.MultiIndex.from_arrays(index_data_rows,
                                       names=('level_0_rows', 'level_1_rows'))

index_data_cols = [['a', 'a', 'b'], ['A', 'B', 'B']]
index_cols = pd.MultiIndex.from_arrays(index_data_cols, names=('level_0_cols',␣
 ↪'level_1_cols'))
df = pd.DataFrame(data=np.arange(15).reshape((5, 3)),
                  columns=index_cols,
                  index=index_rows)
```

[17]:
```
df
```

[17]:
```
level_0_cols                a       b
level_1_cols                A   B   B
level_0_rows level_1_rows
1            x              0   1   2
             y              3   4   5
2            x              6   7   8
             y              9  10  11
3            z             12  13  14
```

[18]:
```
df.index
```

[18]:
```
MultiIndex([(1, 'x'),
            (1, 'y'),
            (2, 'x'),
            (2, 'y'),
            (3, 'z')],
           names=['level_0_rows', 'level_1_rows'])
```

[19]:
```
df.columns
```

[19]:
```
MultiIndex([('a', 'A'),
            ('a', 'B'),
            ('b', 'B')],
           names=['level_0_cols', 'level_1_cols'])
```

#### unstack

[20]:
```
unstacked = df.unstack()
```

[21]:
```
unstacked
```

[21]:
```
level_0_cols     a                          b
level_1_cols     A              B           B
level_1_rows     x    y     z    x     y     z    x     y     z
level_0_rows
1              0.0  3.0   NaN  1.0   4.0   NaN  2.0   5.0   NaN
2              6.0  9.0   NaN  7.0  10.0   NaN  8.0  11.0   NaN
3              NaN  NaN  12.0  NaN   NaN  13.0  NaN   NaN  14.0
```

[22]:
```
unstacked.loc[2, ('a', 'B', 'y')]
```

[22]:
```
10.0
```

[23]:
```
unstacked.index
```

[23]:
```
Index([1, 2, 3], dtype='int64', name='level_0_rows')
```

[24]:
```
unstacked.columns
```

[24]:
```
MultiIndex([('a', 'A', 'x'),
            ('a', 'A', 'y'),
            ('a', 'A', 'z'),
            ('a', 'B', 'x'),
            ('a', 'B', 'y'),
            ('a', 'B', 'z'),
            ('b', 'B', 'x'),
            ('b', 'B', 'y'),
            ('b', 'B', 'z')],
           names=['level_0_cols', 'level_1_cols', 'level_1_rows'])
```

stack

[25]:
```
stacked = df.stack()
```

```
/tmp/ipykernel_21859/924736501.py:1: FutureWarning: The previous implementation
of stack is deprecated and will be removed in a future version of pandas. See
the What's New notes for pandas 2.1.0 for details. Specify future_stack=True to
adopt the new implementation and silence this warning.
  stacked = df.stack()
```

[26]:
```
stacked
```

[26]:
```
level_0_cols                              a    b
level_0_rows level_1_rows level_1_cols
1            x            A               0  NaN
                          B               1  2.0
             y            A               3  NaN
                          B               4  5.0
```

```
2           x           A           6   NaN
                        B           7   8.0
            y           A           9   NaN
                        B          10  11.0
3           z           A          12   NaN
                        B          13  14.0
```

[27]:
```
stacked.index
```

[27]:
```
MultiIndex([(1, 'x', 'A'),
            (1, 'x', 'B'),
            (1, 'y', 'A'),
            (1, 'y', 'B'),
            (2, 'x', 'A'),
            (2, 'x', 'B'),
            (2, 'y', 'A'),
            (2, 'y', 'B'),
            (3, 'z', 'A'),
            (3, 'z', 'B')],
           names=['level_0_rows', 'level_1_rows', 'level_1_cols'])
```

[28]:
```
stacked.columns
```

[28]:
```
Index(['a', 'b'], dtype='object', name='level_0_cols')
```

### 11.5.3 Grouping data

When dealing with multi dimensional data, you may need to extract global trendlines regarding some specific attributes. This can be done using `groupby`.

[29]:
```
df_long
```

[29]:
```
  Animal Feature  Value
0    cat     Age     11
1    cat    Mass      5
2    cat    Mass      9
3    dog     Age      8
4    dog    Mass     17
5    dog    Mass     11
6    cow     Age      4
7    cow    Mass    650
```

#### Unique function

Below, let's compute the average of values 'Value' for every pair (`Animal`, `Feature`).

[30]:
```
df_long.groupby(by=['Animal', 'Feature'])['Value'].mean()
```

[30]:
```
Animal  Feature
cat     Age         11.0
        Mass         7.0
cow     Age          4.0
        Mass       650.0
dog     Age          8.0
        Mass        14.0
Name: Value, dtype: float64
```

note: in this particular case, the result is very similar to what would be returned by `melt`.

If several columns exist, the agregate is done everywhere:

[31]:
```
df_long['Other value'] = range(10, 18)
df_long
```

[31]:
```
  Animal Feature  Value  Other value
0    cat     Age     11           10
1    cat    Mass      5           11
2    cat    Mass      9           12
3    dog     Age      8           13
4    dog    Mass     17           14
5    dog    Mass     11           15
6    cow     Age      4           16
7    cow    Mass    650           17
```

[32]:
```
df_long.groupby(by=['Animal', 'Feature']).mean()
```

[32]:
```
                Value  Other value
Animal Feature
cat    Age       11.0         10.0
       Mass       7.0         11.5
cow    Age        4.0         16.0
       Mass     650.0         17.0
dog    Age        8.0         13.0
       Mass      14.0         14.5
```

### Multiple functions

But one can specify a different aggregate function depending on the column. This is done passing a dictionary to agg:

[33]:
```
df_long.groupby(by=['Animal', 'Feature']).agg({'Value': 'mean', 'Other value':␣
↪list})
```

[33]:
```
                Value Other value
Animal Feature
cat    Age       11.0        [10]
       Mass       7.0    [11, 12]
```

```
cow     Age        4.0          [16]
        Mass     650.0          [17]
dog     Age        8.0          [13]
        Mass      14.0      [14, 15]
```

### Iterating over groups

Without aggregating, one can **iterate over groups**.

[34]:
```
groupby_object = df_long.groupby(by=['Animal', 'Feature'])
```

[36]:
```
for tuple_, dataframe in groupby_object:
    print(tuple_)
    print(dataframe, end='\n\n')
    if tuple_==('cow', 'Mass'):
        break # stop displaying values
```

```
('cat', 'Age')
  Animal Feature  Value  Other value
0    cat     Age     11           10

('cat', 'Mass')
  Animal Feature  Value  Other value
1    cat    Mass      5           11
2    cat    Mass      9           12

('cow', 'Age')
  Animal Feature  Value  Other value
6    cow     Age      4           16

('cow', 'Mass')
  Animal Feature  Value  Other value
7    cow    Mass    650           17
```

### Merging data

**Case study** Merging data is needed to work on a unified instance that contains all the relevant information. For instance, here are some datasets having similar features:

[40]:
```
df1 = pd.DataFrame({'Name': ('Laura', 'Bob', 'Sarah', 'Li'),
                    'Age': (45, 15, 41, 23),
                    'Address': ('Annecy', 'Turin', 'Annecy', 'Chambéry')})
df2 = pd.DataFrame({'Name': ('Sarah', 'Li', 'Pierre', 'David'),
                    'Age': (41, 23, 26, 45),
                    'Address': ('Annecy', 'Paris', 'Geneva', 'Annecy')})
```

[41]:
```
df1
```

[41]:
```
    Name  Age   Address
0  Laura   45    Annecy
1    Bob   15     Turin
2  Sarah   41    Annecy
3     Li   23  Chambéry
```

[42]:
```python
df2
```

[42]:
```
     Name  Age Address
0   Sarah   41  Annecy
1      Li   23   Paris
2  Pierre   26  Geneva
3   David   45  Annecy
```

Note that:

- A row is common to `df1` and `df2`: the one with name `Sarah`
- A row is common to `df1` and `df2` yet has a different value for column `Address`: the one with name `Li`
- Some rows exist only in `df1`, or only in `df2`.

**Outer merge** Let's use `merge` to gather these datasets in one instance:

[43]:
```python
pd.merge(df1, df2, how='outer', on=['Name', 'Age'], suffixes=('_df1', '_df2'))
```

[43]:
```
     Name  Age Address_df1 Address_df2
0     Bob   15       Turin         NaN
1   David   45         NaN      Annecy
2   Laura   45      Annecy         NaN
3      Li   23    Chambéry       Paris
4  Pierre   26         NaN      Geneva
5   Sarah   41      Annecy      Annecy
```

Some explanations:

- `on` tells `pandas` where to look for different tuples of values. These columns must exist in both dataframes.

- `suffixes` makes it possible to assign different names to columns that have the same name in both dataframes.

- `how='outer'` creates one row for every `('Name', 'Age')` pair in `df1` **or** in `df2`.
  - Specifying `how='inner'` would create a row for every pair that exists in `df1` **and** in `df2`
  - `how='left'` only takes pairs of `df1`.
  - `how='right'` only takes pairs of `df2`.

**Inner merge** Here after, using `how='inner'`.

[44]:
```python
pd.merge(df1, df2, how='inner', on=['Name', 'Age'], suffixes=('_df1', '_df2'))
```

[44]:
```
    Name  Age Address_df1 Address_df2
0  Sarah   41      Annecy      Annecy
1     Li   23    Chambéry       Paris
```

If on is set to 'Address' how='inner' only ‘Annecy’ which is in both df1 and df2 is kept:

[45]:
```
pd.merge(df1, df2, how='inner', on=['Address'], suffixes=('_df1', '_df2'))
```

[45]:
```
  Name_df1  Age_df1 Address Name_df2  Age_df2
0    Laura       45  Annecy    Sarah       41
1    Sarah       41  Annecy    Sarah       41
2    Laura       45  Annecy    David       45
3    Sarah       41  Annecy    David       45
```

**Left/Right merge** Here after, using how='left'.

[46]:
```
pd.merge(df1, df2, how='left', on=['Name', 'Age'], suffixes=('_df1', '_df2'))
```

[46]:
```
    Name  Age Address_df1 Address_df2
0  Laura   45      Annecy         NaN
1    Bob   15       Turin         NaN
2  Sarah   41      Annecy      Annecy
3     Li   23    Chambéry       Paris
```

### 11.5.4 Applying a rolling function

Suppose we have some experimental data. How can we compute a rolling mean?

[83]:
```
sr = pd.Series(range(6, 0, -1), index=list('abcdef'))
sr
```

[83]:
```
a    6
b    5
c    4
d    3
e    2
f    1
dtype: int64
```

Let’s use the rolling method:

[84]:
```
sr.rolling(window=3).mean()
```

[84]:
```
a    NaN
b    NaN
c    5.0
d    4.0
e    3.0
f    2.0
```

```
dtype: float64
```

The **default behaviour makes the window flushed to the right**: the output value at index $k$ is computed using the input values from $k - windows + 1$ to $k$.

This baheviour can be changed using `center=True`:

[85]:
```python
sr.rolling(window=3, center=True).mean()
```

[85]:
```
a    NaN
b    5.0
c    4.0
d    3.0
e    2.0
f    NaN
dtype: float64
```

Similarly to `groupby` and `resample` objects, one can iterate over what is returned by the `rolling` method:

[89]:
```python
rolling_object = sr.rolling(window=3)
for k in rolling_object:
    print(k)
```

```
a    6
dtype: int64
a    6
b    5
dtype: int64
a    6
b    5
c    4
dtype: int64
b    5
c    4
d    3
dtype: int64
c    4
d    3
e    2
dtype: int64
d    3
e    2
f    1
dtype: int64
```

# Chapter 12

# Matplotlib

## 12.1 Simle plot [easy]

### 12.1.1 Introduction

`matplotlib` is a Python library that creates charts.

#### Pros

- Adapted to scientific publications: straight-forward charts
- Unlimited customization of charts
- Large online community

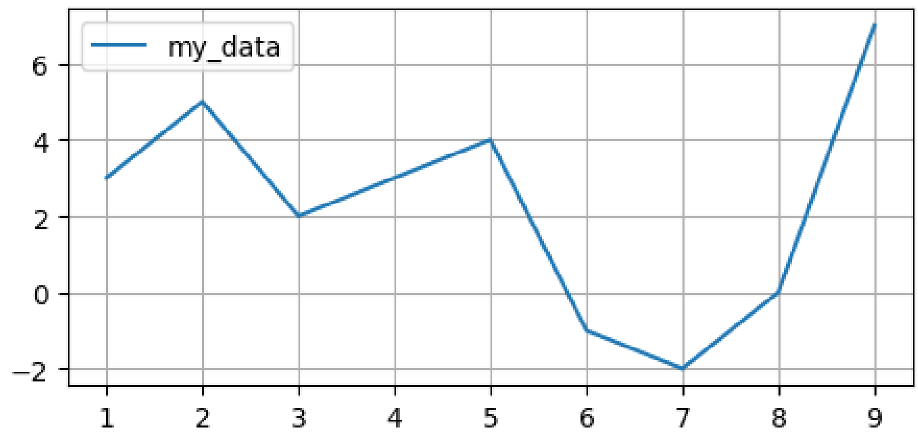
#### Cons

- No interactive plotting

### 12.1.2 Simple plot

[2]:
```
import matplotlib.pyplot as plt
```

[4]:
```
x = range(1, 10)
y = [3, 5, 2, 3, 4, -1, -2, 0, 7]
fig, ax = plt.subplots(figsize=(5, 2.5))    # figsize is given in inches: 1 inch␣
↪= 2.54 cm
ax.plot(x, y, label="my_data")
_ = ax.legend()
```



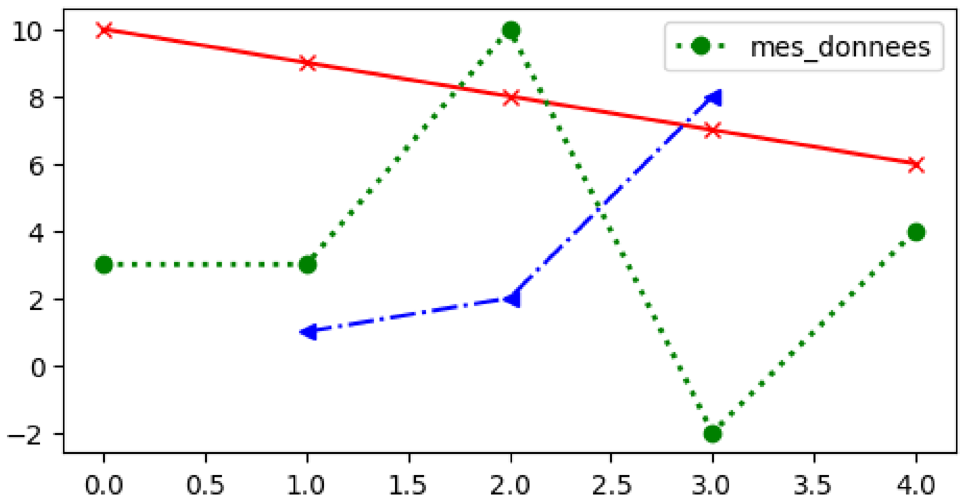

Above, a simple full blue line is used. Yet, plotting style can be fully configured when calling `plot`.

Here after, 3 lines are plotted on the same chart. The first two plots set the line style in a short format, whereas the third one use named keyword argument (`marker`, `linestyle`, `color`). Note that:

- `'x-r'` tells that:
  - marker is an 'x'
  - line must be full
  - color is red
- In the `'<-.b'` string, `-.` stands for a 'dash-dot line style'

[16]:
```
x1 = range(5)
x2 = range(1, 4)

y1 = range(10, 5, -1)
y2 = [1, 2, 8]

x3 = range(5)
y3 = [3, 3, 10, -2, 4]

fig, ax = plt.subplots(figsize=(6, 3))
ax.plot(x1, y1, 'x-r')
ax.plot(x2, y2, '<-.b')
ax.plot(x3, y3, linewidth=2, marker='o', linestyle=':', color="green",␣
 ↪label="mes_donnees")
ax.legend()
```

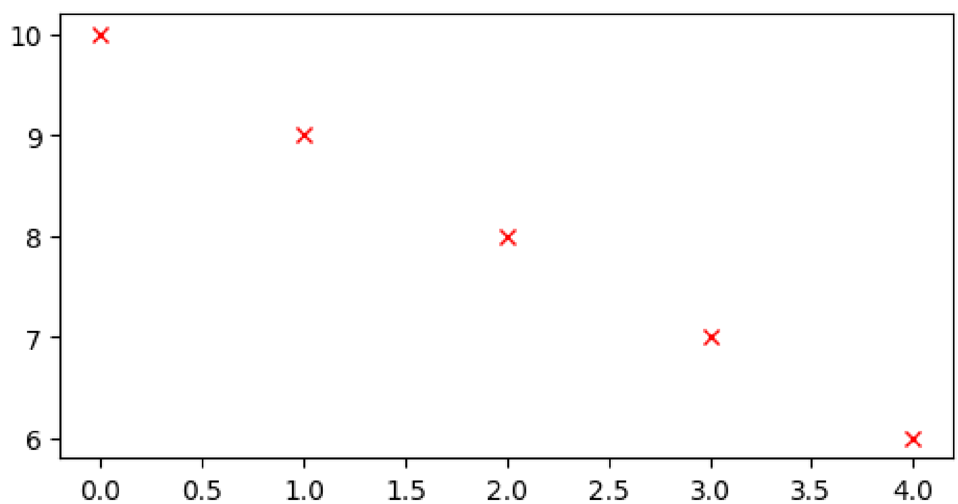
[16]:
```
<matplotlib.legend.Legend at 0x7fca2c274ac0>
```




Thus it is possible to plot only the markers, without any line:

[17]:
```
fig, ax = plt.subplots(figsize=(6, 3))
ax.plot(x1, y1, 'xr')
```

[17]: `[<matplotlib.lines.Line2D at 0x7fca2c1216f0>]`



### 12.1.3 3D plots

#### Case study definition

Let's define values of a 2-variables function:

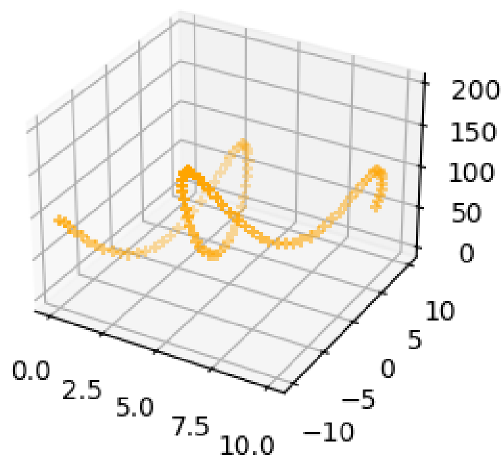
[18]:
```
import numpy as np
x = np.linspace(0, 10, 100)
y = 10 * np.sin(np.linspace(5, 15, 100))
z = (x-y)**2 + x*y
```

### Points

The `scatter` function with a 3d projection can plot the data with only markers:

[19]:
```
fig, ax = plt.subplots(figsize=(6, 3), subplot_kw={'projection': '3d'})
ax.scatter(x, y, z, marker='+', color='orange')
```

[19]: `<mpl_toolkits.mplot3d.art3d.Path3DCollection at 0x7fca2c21ef20>`



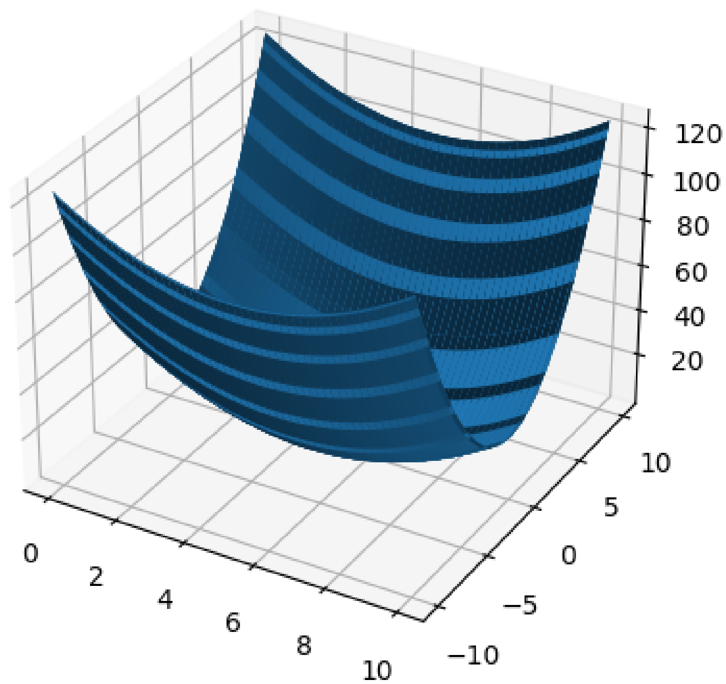

### Surface

The previous example used specific values for `x` and `y` data. But if a 2-variable function must be completely described, a grid evaluation (`np.meshgrid`) and a surface plot (`ax.plot_surface`) are needed.

$ f: (x, y) \rightarrow (x - 5)^{2} + y^{2}$

[20]:
```
xx, yy = np.meshgrid(x, y)
zz = (xx - 5)**2 + yy**2

fig, ax = plt.subplots(subplot_kw={"projection": "3d"})
ax.plot_surface(xx, yy, zz)
```

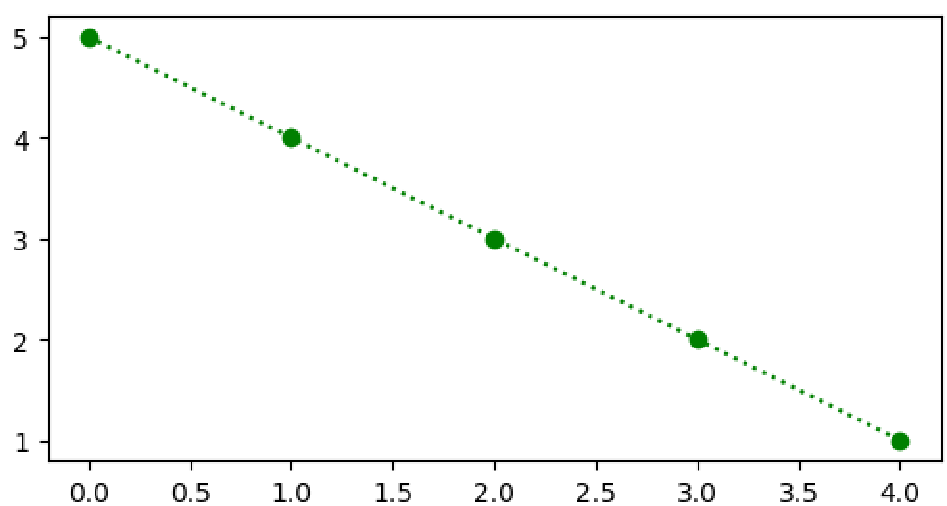
[20]: `<mpl_toolkits.mplot3d.art3d.Poly3DCollection at 0x7fca2c014e20>`




### 12.1.4 Saving a figure

When using `matplotlib`, figures are shown in an interactive way. Whenever saving to the disk is required, one must use the `savefig` method of the figure:

[21]:
```
fig, ax = plt.subplots(figsize=(6, 3))
ax.plot(range(5), range(5, 0, -1), 'o:g')
fig.savefig('figures/super_figure.png', dpi=200, bbox_inches='tight')
```



Note the arguments:

- `bbox_inches='tight'`: use all available space in the window
- `dpi`: set the quality of the figure in **pixels per inch**.

  Most scientific journals require a dpi higher than 200. Yet, a larger dpi comes with:

  - a larger disk space
  - a longer writing time

  Hence, prefer a not to high dpi in your every-day life.

## 12.2 Complex figures [medium]

### 12.2.1 Introduction

A figure is made of several `matplotlib.axes` objects, also called 'ax'. These can be placed in the figure in a special order. In this part, two approaches of axes layout are presented:

1. Simple approach: create a grid of `axes` objects, where all have the same dimensions
2. Advanced approach: define custom dimensions for each `axe`

### 12.2.2 Simple approach

#### Using an iterable

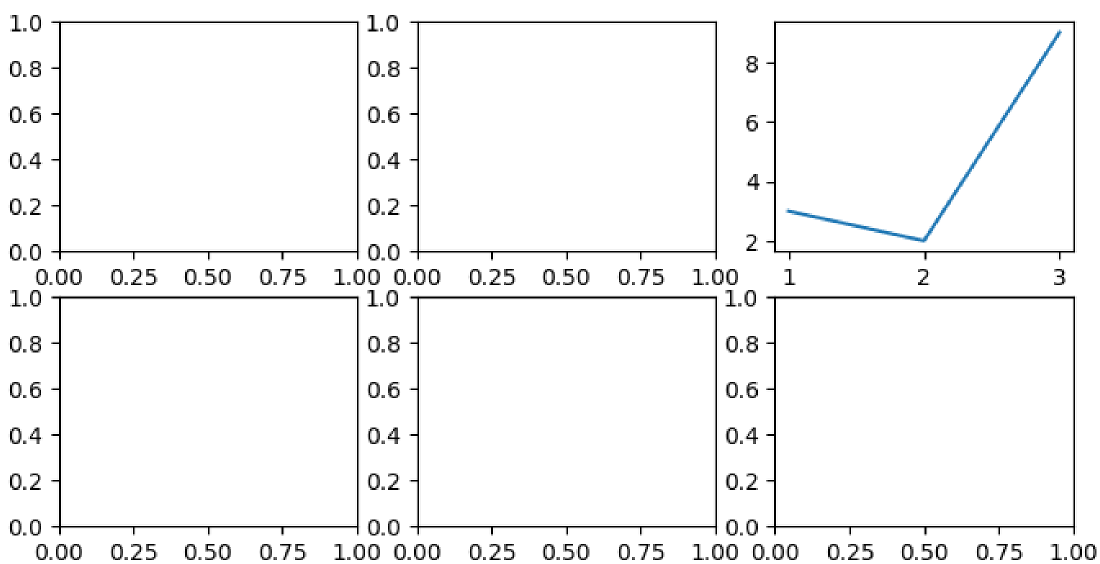
Using `plt.subplots`, axes are available in a 2D `numpy` array.

Let's:

- create 3 axes along columns and 2 along rows, i.e. an array of shape (2, 3)
- use of the axes to plot something.

[17]:
```
import matplotlib.pyplot as plt
fig, axes = plt.subplots(nrows=2, ncols=3, figsize=(8, 4))
ax = axes[0, 2]            # access the first row and third column
ax.plot([1,2,3], [3,2,9])
```

[17]:
```
[<matplotlib.lines.Line2D at 0x7f0cd30c2080>]
```

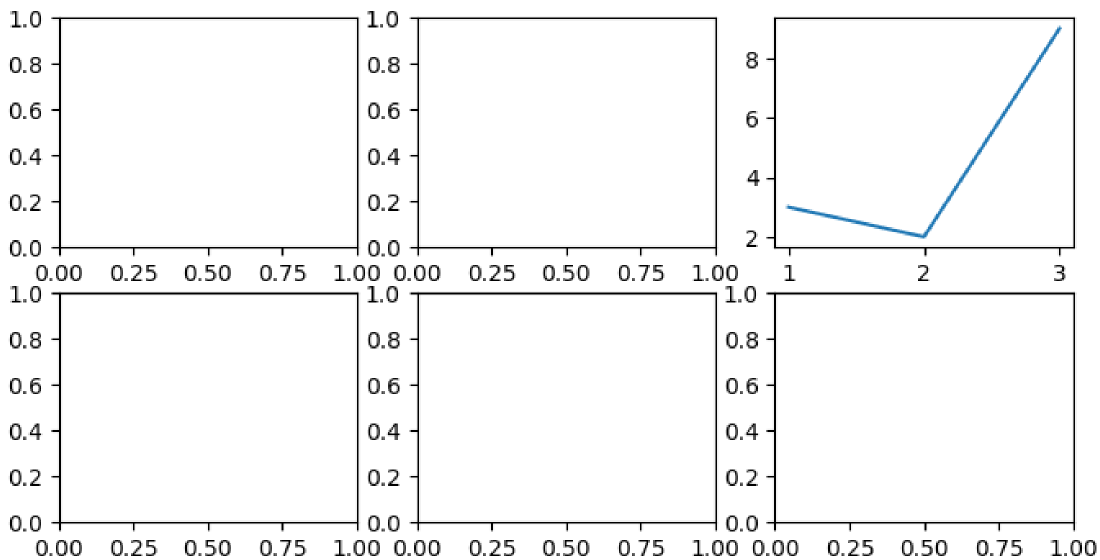



#### Using a dictionary

`plt.subplot_mosaic` is similar to `plt.subplots` yet it returns a dictionary whose keys are the names used in the passed iterable.

[18]:
```
wanted_layout = [['ax1', 'ax2', 'ax3'],
                 ['ax4', 'ax5', 'ax6']]
fig, axes = plt.subplot_mosaic(wanted_layout,
                               figsize=(8,4))
ax = axes['ax3']
ax.plot([1,2,3], [3,2,9])
```

[18]:
```
[<matplotlib.lines.Line2D at 0x7f0cd2f9f400>]
```



### 12.2.3 Advanced approach

In the previous example, all axes had equal sizes. Let's create some axes with different sizes.

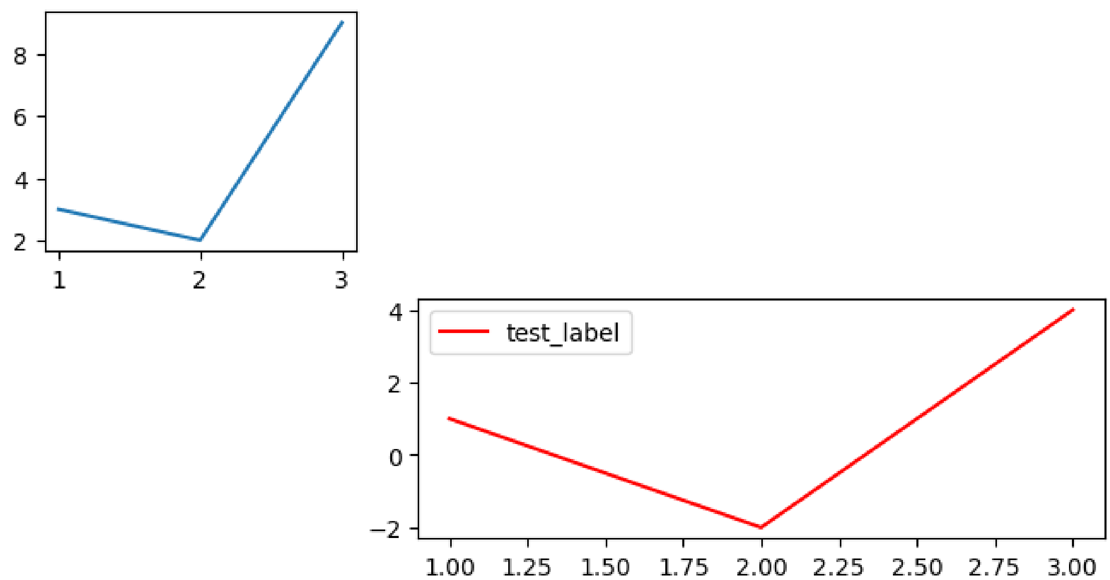
#### Using an iterable

There are 3 steps:

1. create the figure
2. add to the figure a `GridSpec` instance using `add_gridspec`
3. add axes one after another, specifying the rows and columns of the grid that axes must occupy. This is done using the `add_subplot` function, which returns an `axe` object.

[19]:
```
fig = plt.figure(figsize=(8, 4))
spec = fig.add_gridspec(2, 3)         # 2 rows, 3 columns:
                                      # no more than 6 axes can be defined
                                      # but possibly less if some spans several
                                      # rows/columns


ax1 = fig.add_subplot(spec[0, 0])
```

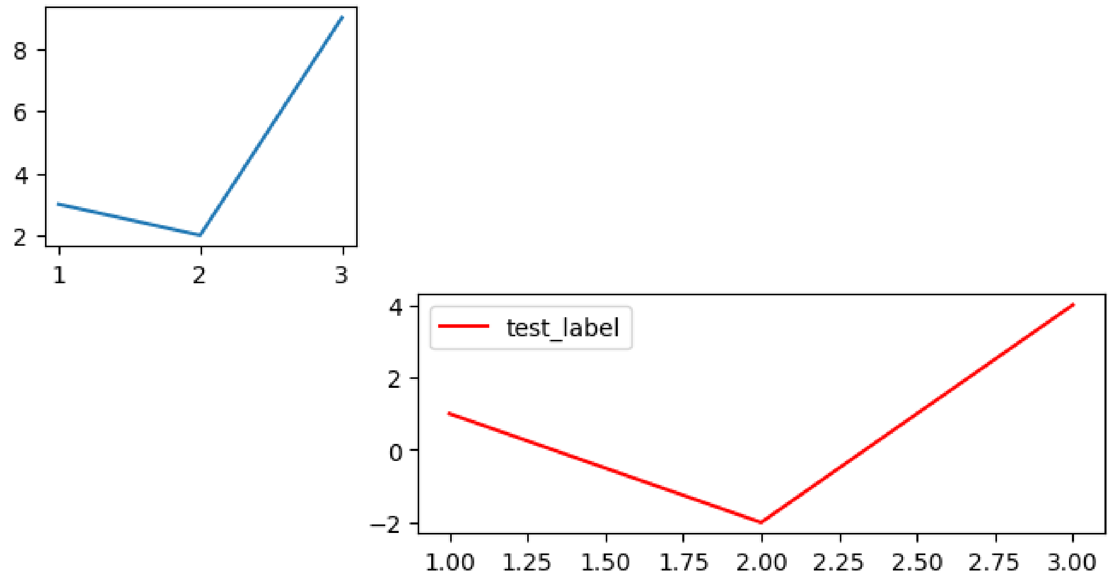
```
_ = ax1.plot([1,2,3], [3,2,9])

ax2 = fig.add_subplot(spec[1, 1:3])  # 2nd row, from 2nd to 3rd column
ax2.plot([1,2,3], [1,-2,4], label='test_label', color='red')
_ = ax2.legend()
```




**Using a dictionary**

In a dictionary approach (`subplot_mosaic`), one must set the same name in different places to have an axe span several rows/columns.

[20]:
```
wanted_layout = [['ax1', '.', '.'],      # the dot '.' tells matplotlib
                 ['.', 'ax6', 'ax6']]    # not to define axes at these locations
fig, axes = plt.subplot_mosaic(wanted_layout,
                               figsize=(8,4))
ax1 = axes['ax1']
_ = ax1.plot([1,2,3], [3,2,9])

ax2 = axes['ax6']
ax2.plot([1,2,3], [1,-2,4], label='test_label', color='red')
_ = ax2.legend()
```

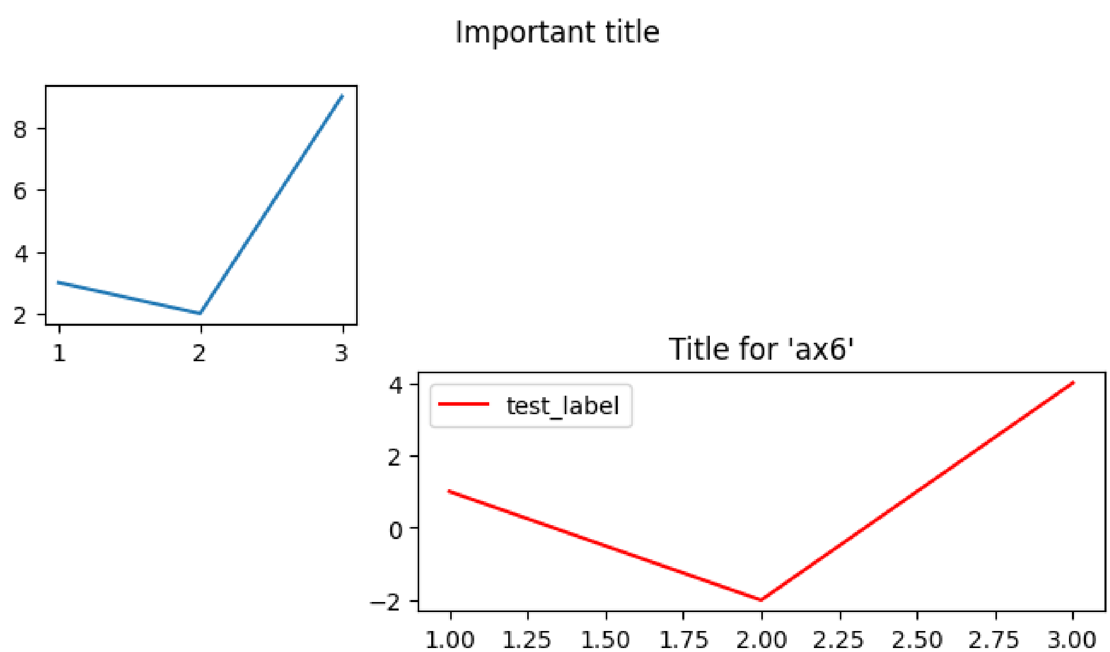



### 12.2.4 Notes

The figure have some similar properties to those of the axes. For instance, let's define a title at the figure level using `suptitle`:

[21]:
```
fig.suptitle('Important title')
ax = axes['ax6']
ax.set_title("Title for 'ax6'")
fig
```

[21]:



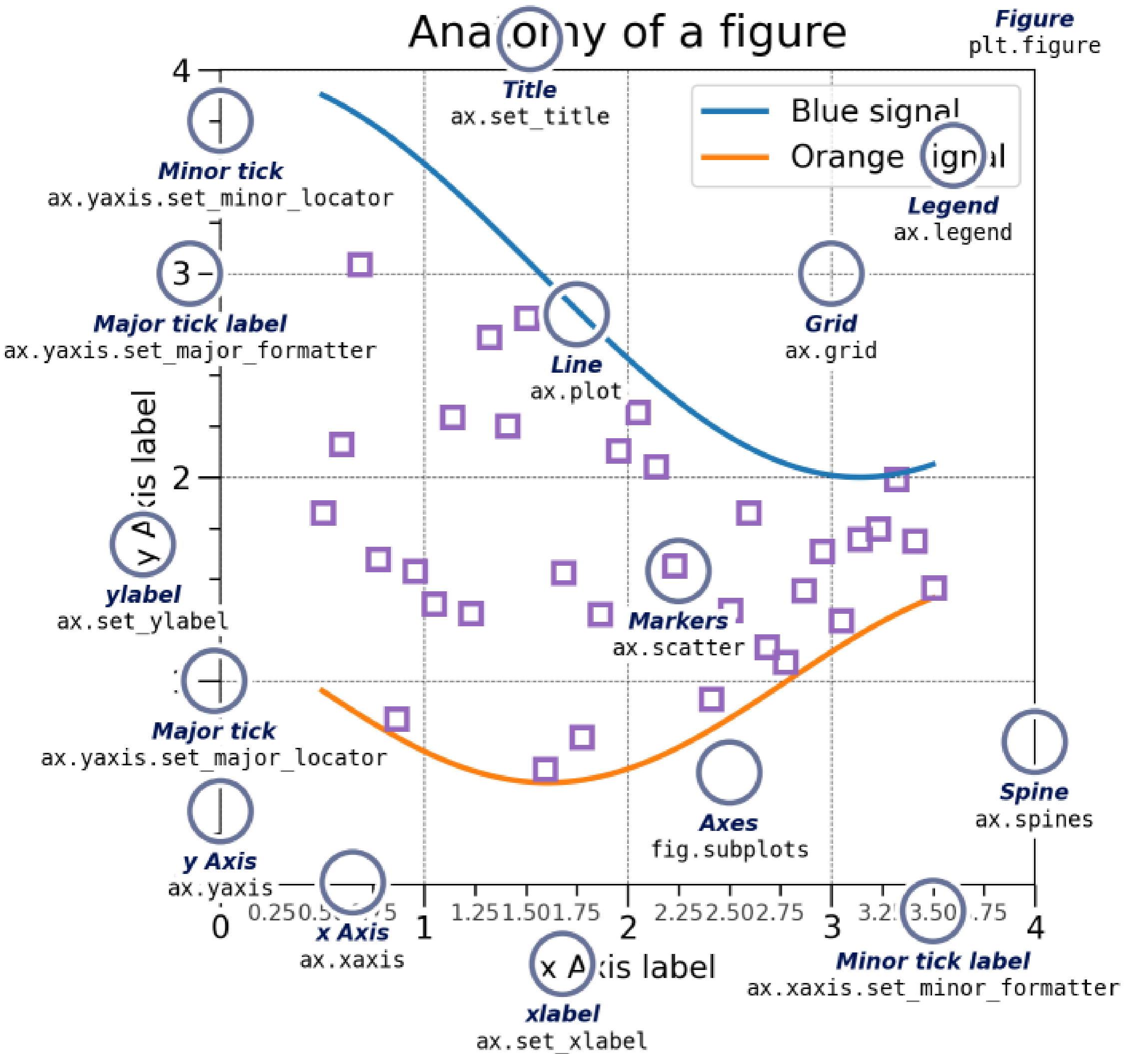
## 12.3 Customization [medium]

### 12.3.1 Introduction

A figure is made of one or several `axes` objects. In practice, an ax is the zone where the chart is drawn.

The properties of an ax can be modified before after the data is plotted.

Here is a description of the different customizable features of an ax:




### 12.3.2 Adding titles, labels, annotations, ...

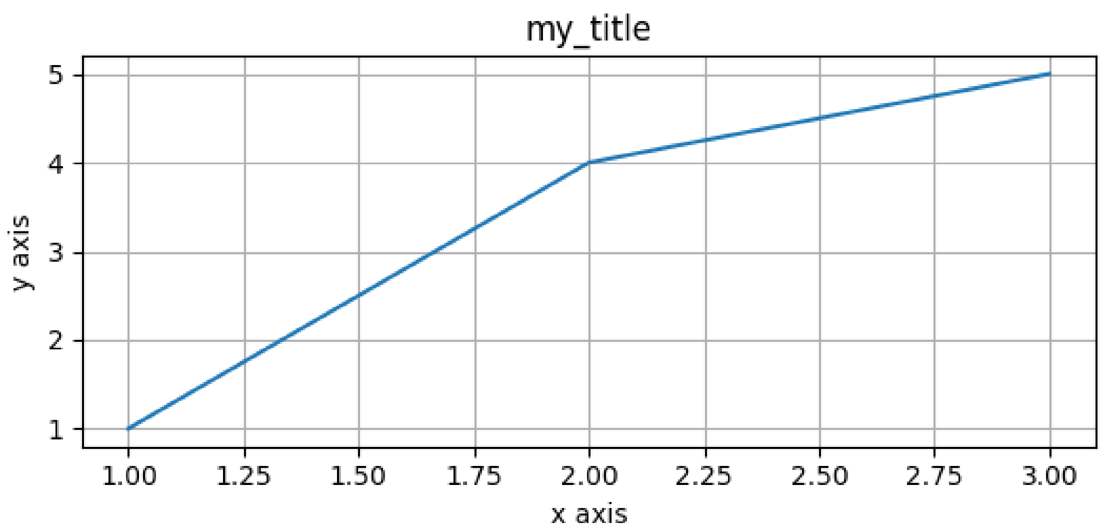
#### Basics

A couple of `set_...` methods can be used. Note that ax and axis are different things.

[1]:
```
import matplotlib.pyplot as plt
fig, ax = plt.subplots(figsize=(6, 3))
```

```
ax.plot([1,2,3], [1, 4, 5])
ax.set_xlabel('x axis')     # a label to the x axis
ax.set_ylabel('y axis')     # a label to the y axis
ax.set_title('my_title')    # a title to the ax
```

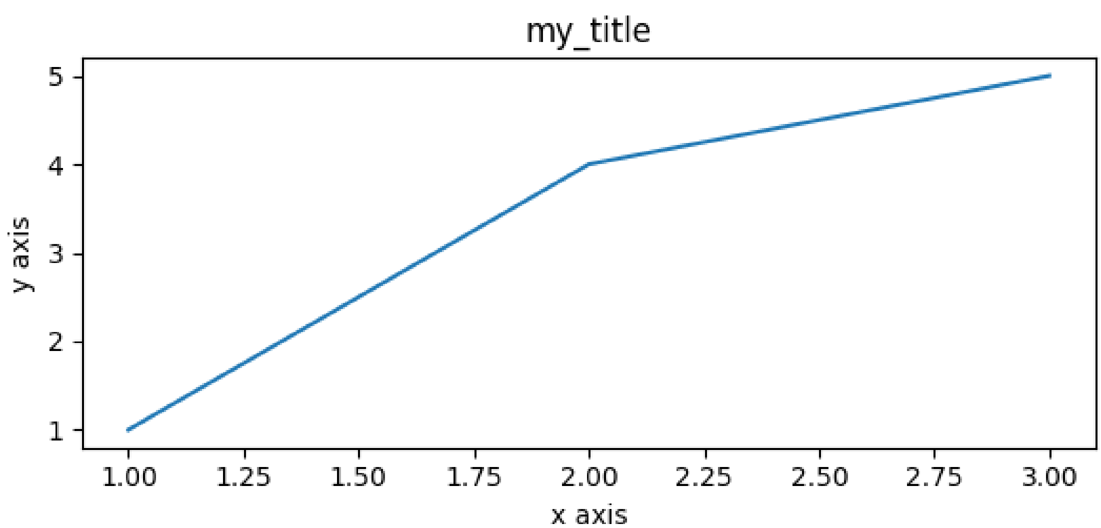
[1]: Text(0.5, 1.0, 'my_title')




A grid is set using `ax.grid()`:

[2]:
```
ax.grid()
fig
```

[2]:



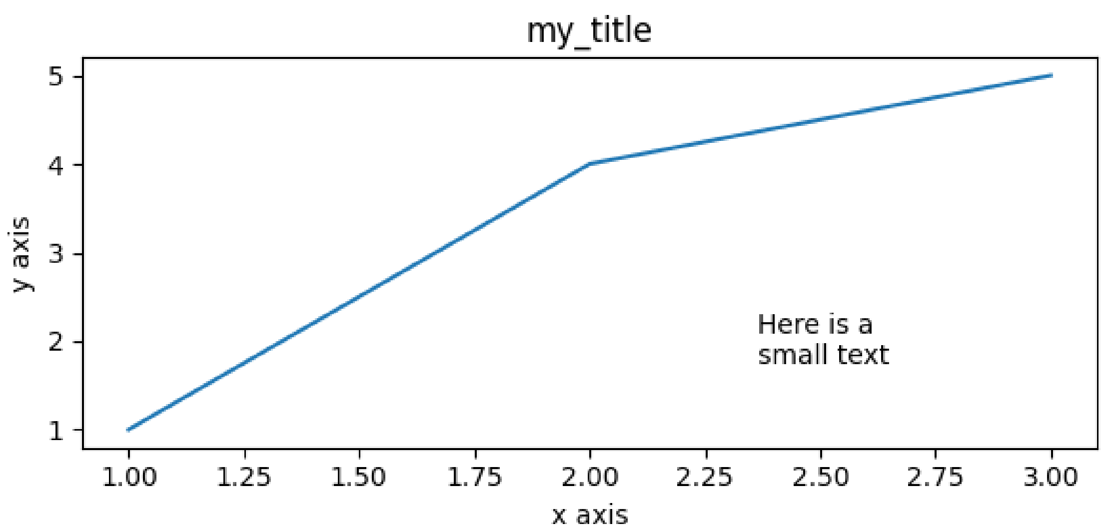
*Note*: in a Jupyter notebook, figures are automatically displayed after some content is plotted. Yet, one can display them again (in another cell) using the `fig.show()` method.

### Annotations

Let's add a small text near the plotted area, using `annotate`.

[3]:
```
ax.annotate('Here is a \n small text', xy=(2.5, 2), xytext=(2.5, 2),
            va="center", # vertical alignment of the text
            ha="center"  # horizontal alignment
           )
fig
```

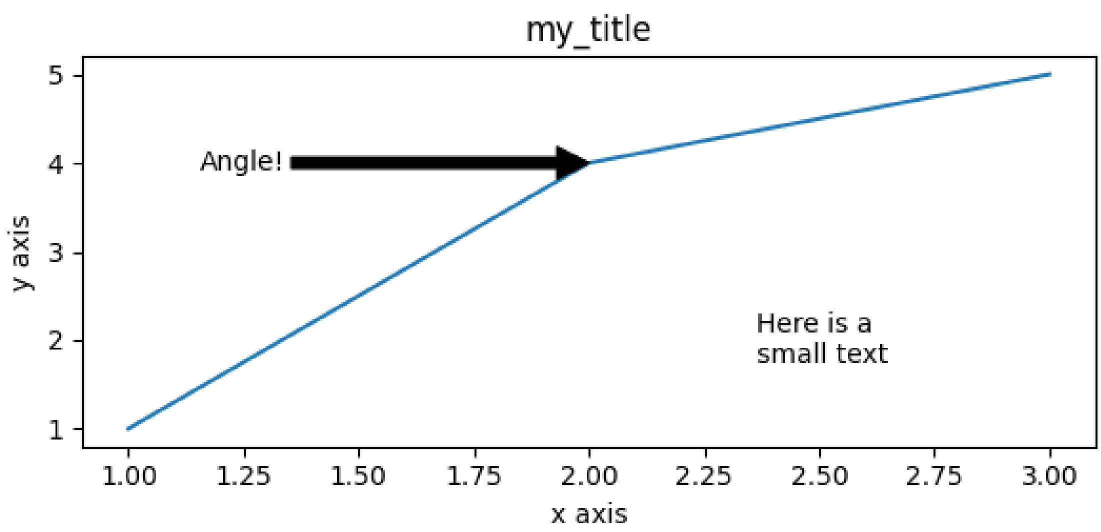
[3]:




An arrow might be needed to describe some specific features of the plot!

[4]:
```
ax.annotate('Angle!', xy=(2, 4), xytext=(1.25, 4),
            arrowprops={"facecolor":'black'},
           va="center", ha="center")
fig
```

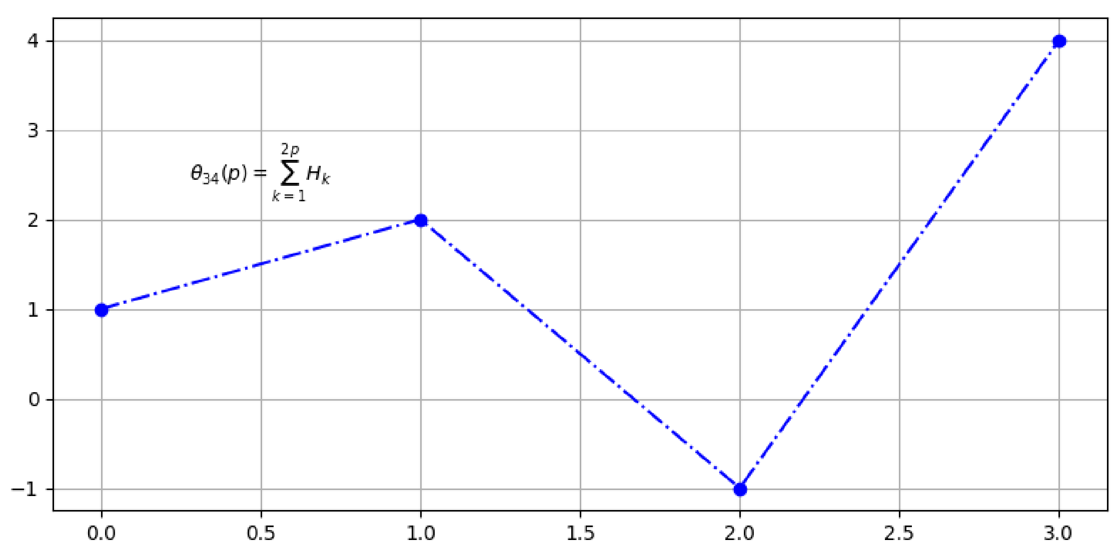
[4]:




### Mathematical content

Mathematical formulas can be displayed too: a **Latex-style mathematical content** must be inserted between $ signs, with a r in front of the strings:

[5]:
```
fig, ax = plt.subplots(figsize=(8, 4))
ax.plot([1,2,-1, 4], 'o-.b')      # no x values specified:
                                  # matplotlib assumes they are `range(len(y))`
formula = r'$\theta_{34}(p)=\sum_{k=1}^{2p}{H_k}$'
_ = ax.annotate(formula,
                xy=(0.5, 2.5), xytext=(0.5, 2.5),
                va="center", ha="center")
```



### 12.3.3 Other customizations

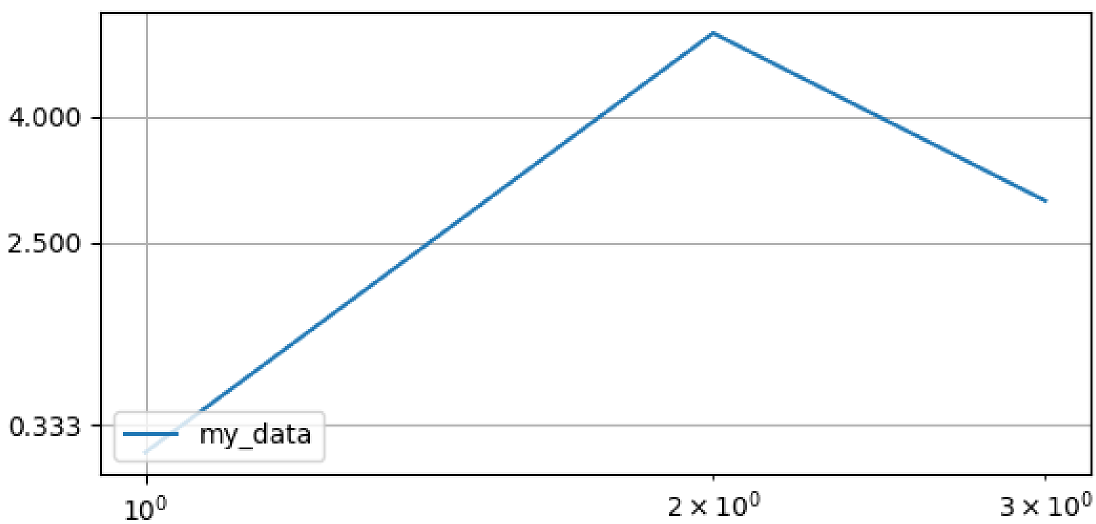
Many other tuning options exist. For instance:

- axis scale: linear, logarithmic, etc. . .
- ticks positions and labels

Below is a more advanced example:

[6]:
```
from matplotlib.ticker import FormatStrFormatter

fig, ax = plt.subplots(figsize=(6, 3))
ax.plot([1,2,3], [0, 5, 3], label="my_data")
ax.set_xscale("log")           # set log scale for x axis
ax.set_yticks([1/3, 2.5, 4])   # set custom position for y ticks
ax.yaxis.set_minor_formatter(FormatStrFormatter("%.5f"))  # show labels with␣
↪more precision for y axis
ax.legend(loc='lower left')    # choose the location of legend
```

[6]: <matplotlib.legend.Legend at 0x78f47831dbb0>




### 12.3.4 Default parameters

Other appearance settings of `matplotlib` are numerous:

- font size and family
- colors
- . . .

These parameters can be modified:

- at the script level only
- at system-wide level: will be applied for all future figures

#### Local modification

One need to mofify the `rcParams` attribute **before** importing `matplotlib.pyplot` everything else that uses `matplotlib`.

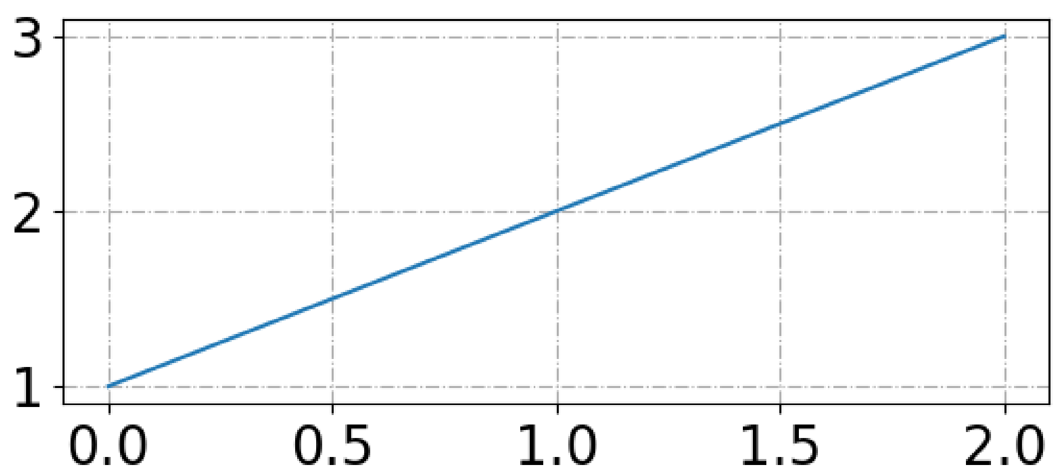
Below is an example of modification:

- a dash grid is forced
- font size is set to 20

All plots in **this script** of will use these parameters.

[7]:
```
import matplotlib as mpl
mpl.rcParams["font.size"] = 20
mpl.rcParams["axes.grid"] = True
mpl.rcParams["grid.linestyle"] = "-."


fig, ax = plt.subplots(figsize=(6, 3))
ax.plot([1,2,3])
```

[7]:
```
[<matplotlib.lines.Line2D at 0x78f46f57d6d0>]
```




If you mess up with `mpl.rcParams`, original settings can be reloaded using :

[8]:
```
mpl.rcParams.update(mpl.rcParamsDefault)
```

#### System level modification

One can modify the file where `matplotlib` store the default parameters. This is the simplest solution to set parameters once for all.

1. Find where is the parameters file using `mpl.matplotlib_fname()`
2. Save it elsewhere so that you can revert your changes if needed
3. Edit this file

[14]:
```
print('On my computer, the file lies in: \n...', mpl.matplotlib_fname()[12:],␣
↪sep='')
```

```
On my computer, the file lies in:
.../Python/3.12/lib/python3.12/site-packages/matplotlib/mpl-data/matplotlibrc
```

### 12.3.5 Advice

Customizing a plot can be very time-consuming. You must do it **at the last time**, for instance when the figure is shared with other people (article , poster, presentation, . . . ).

## 12.4 Matplotlib backend [medium]

### 12.4.1 Introduction

Many chart types can be built with `matplotlib`: histograms, bars, polar, ... One can look at the **examples presented in the documentation and gallery.**.

Yet, a lot of code lines are needed to build only a simple graph. In some cases, the prefered way is this one:

1. Use higher-level packages **built on top of** `matplotlib`
2. Use `matplotlib` to customize the plot if needed

### 12.4.2 Case study definition

A dataset about a pengouins population is downloaded on seaborn_data and stored on disk as 'data.csv'.

[1]:
```
import matplotlib.pyplot as plt
import pandas as pd

df = pd.read_csv('data.csv')
df
```

[1]:
```
    species     island  bill_length_mm  bill_depth_mm  flipper_length_mm  \
0    Adelie  Torgersen            39.1           18.7              181.0
1    Adelie  Torgersen            39.5           17.4              186.0
2    Adelie  Torgersen            40.3           18.0              195.0
3    Adelie  Torgersen             NaN            NaN                NaN
4    Adelie  Torgersen            36.7           19.3              193.0
..      ...        ...             ...            ...                ...
339  Gentoo     Biscoe             NaN            NaN                NaN
340  Gentoo     Biscoe            46.8           14.3              215.0
341  Gentoo     Biscoe            50.4           15.7              222.0
342  Gentoo     Biscoe            45.2           14.8              212.0
343  Gentoo     Biscoe            49.9           16.1              213.0

     body_mass_g     sex
0         3750.0    Male
1         3800.0  Female
2         3250.0  Female
3            NaN     NaN
4         3450.0  Female
..           ...     ...
339          NaN     NaN
340       4850.0  Female
341       5750.0    Male
342       5200.0  Female
343       5400.0    Male
```

```
[344 rows x 7 columns]
```

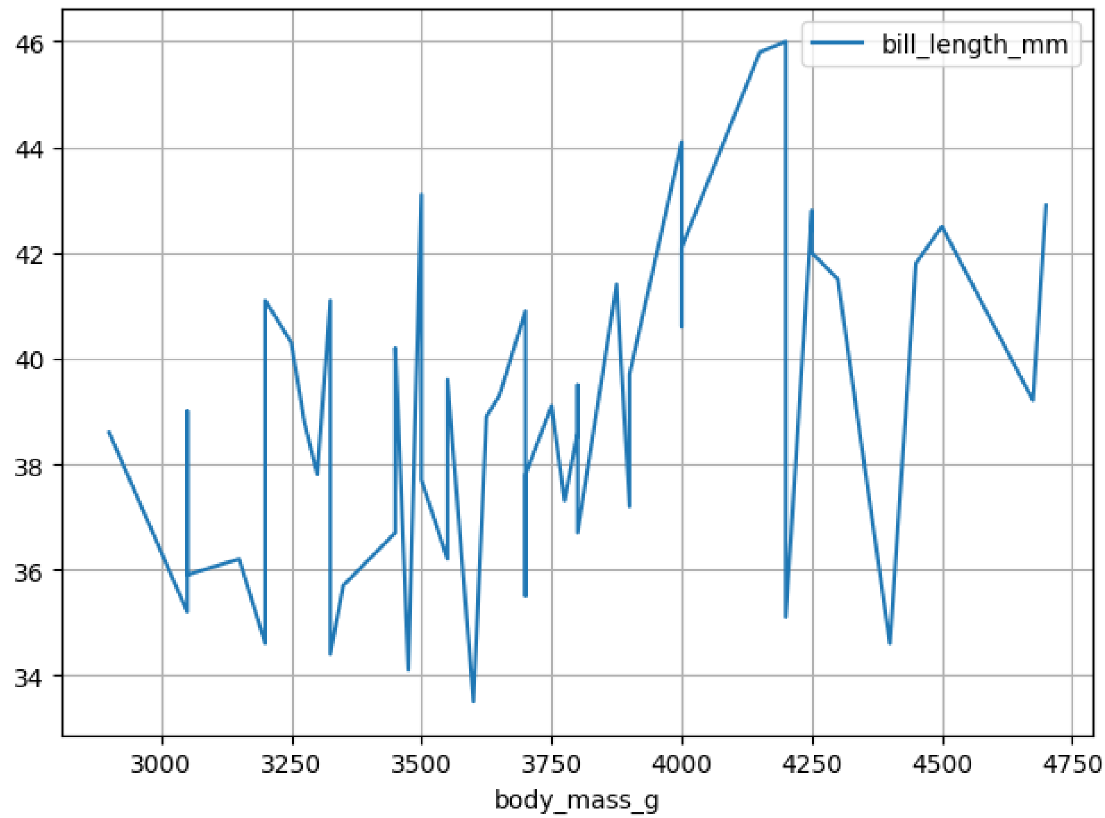
### 12.4.3 pandas

`pandas` can perform quick plot of data stored in either a `DataFrame` or a `Series`.

**Example 1:** ***line plot***

Let's plot the length of the bill of pengouns from Torgersen island, as a function of their mass:

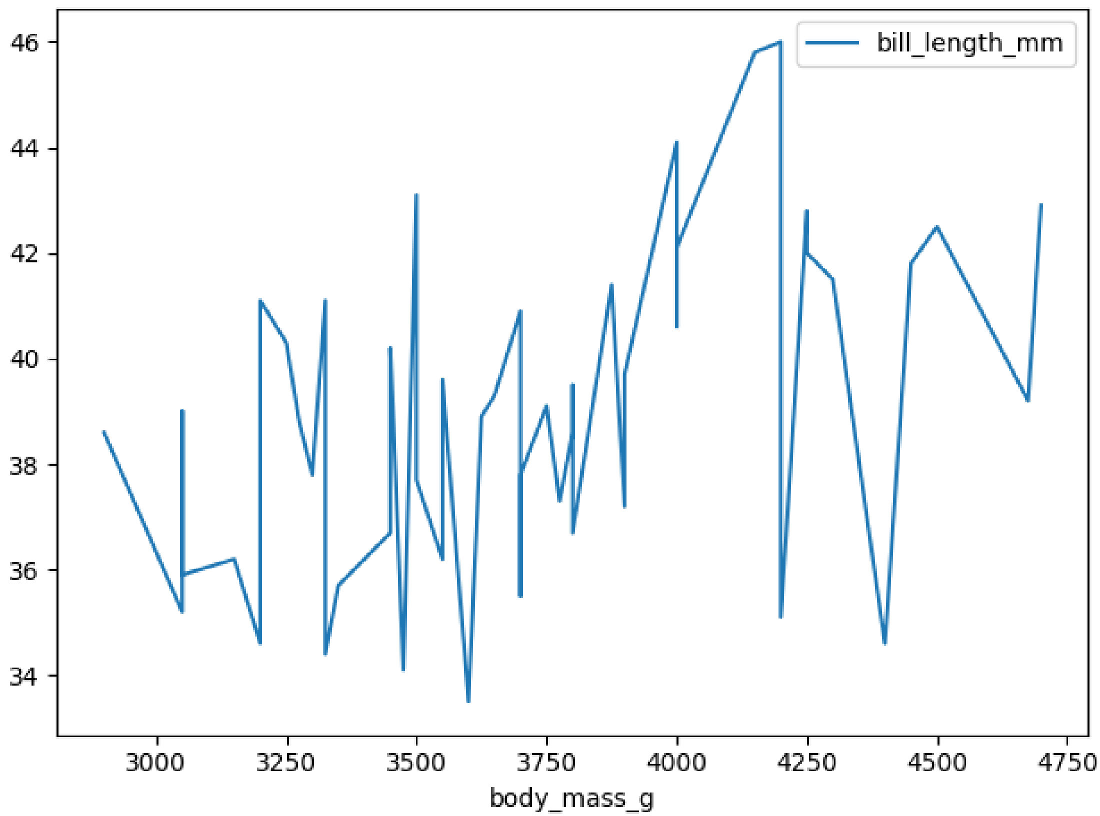
[2]:
```
df1 = df[df['island']=='Torgersen']
df1 = df1.sort_values('body_mass_g')
ax = df1.plot(x='body_mass_g', y='bill_length_mm', kind='line')
```




The call to the `plot` method of `pandas` returned a `matplotlib axes` object: **let's modify it**.

[3]:
```
ax.grid()
ax.get_figure()    # needed to display once again
                   # the figure in Jupyter Notebook.
```

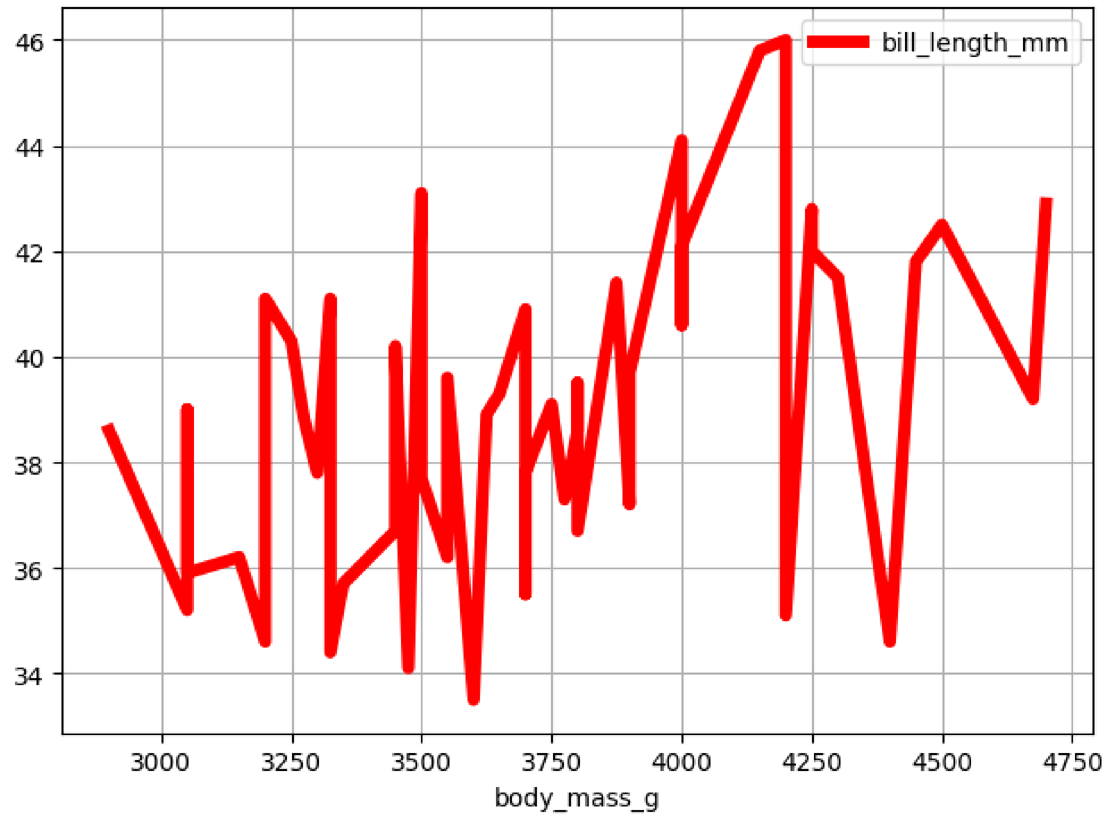
[3]:




Another way is to modify the plot when created. For this purpose, some *keywords arguments* can be passed to the `plot` method of the dataframe. These are the same than the plot function of `matplotlib`.

[4]:
```
df1.plot(x='body_mass_g', y='bill_length_mm', kind='line',
         color='red', linewidth=5)     # these arguments are passed to matplotlib
```

[4]: `<Axes: xlabel='body_mass_g'>`




**Example 2:** ***barplot***

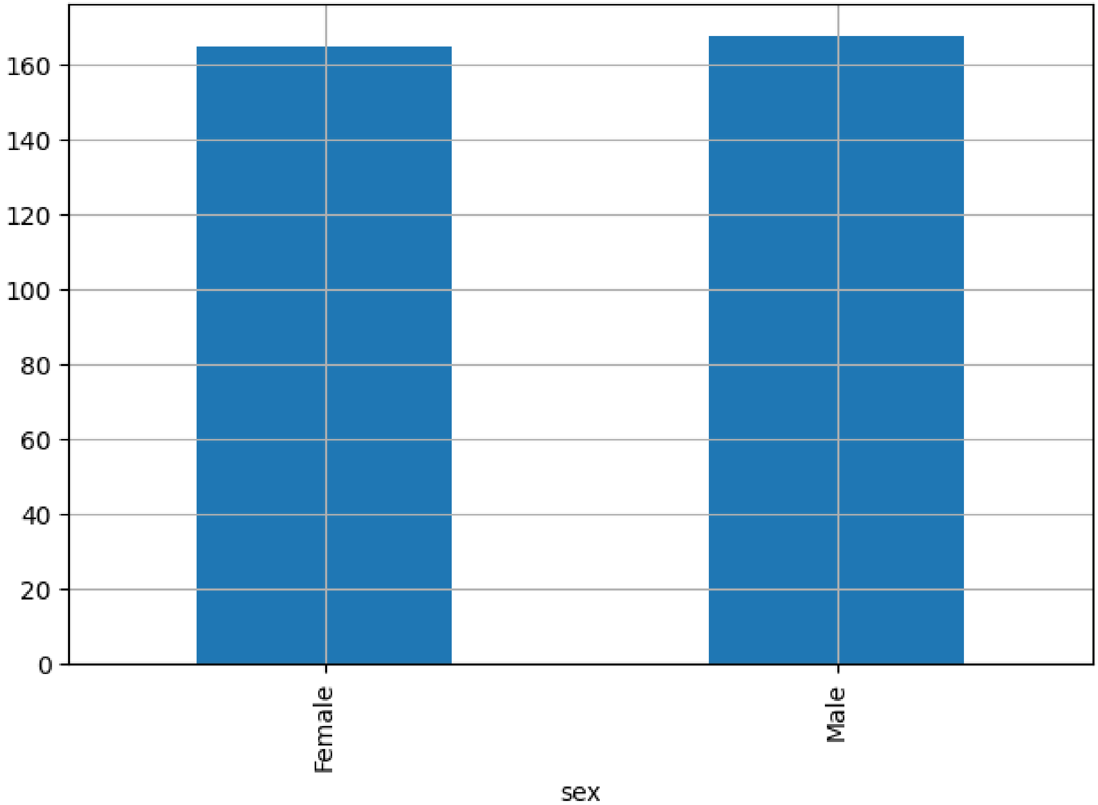
Let's plot the numbers of females and males on Torgersen island:

[5]:
```
df2 = df.groupby('sex')['bill_length_mm'].count()  # a random column is needed
                                                   # for rows to be counted
df2
```

[5]:
```
sex
Female    165
Male      168
Name: bill_length_mm, dtype: int64
```

[6]:
```
df2.plot(kind='bar')
```

[6]: <Axes: xlabel='sex'>




**Conclusion**

The `plot` method is an easy way to quickly inspect the content of a DataFrame. Yet, it is unadapted to advanced statistical plots

In that case, the preferred tool is `seaborn`, which produces clear an pretty charts.

### 12.4.4 `seaborn`

`seaborn` is a plotting library that is built on top of `matplotlib`.

**The strength of `seaborn` is that it can directly be used with `pandas` DataFrames and Series**.

Learning `seaborn` mainly consists in understanding the arguments it takes:

- `x`: abscissa data
- `y`: ordinate data
- `hue`: data differenciated according to a **color** code
- `style`: data differenciated according to the line/marker **style**
- `size`: data differenciated according to the line/marker **size**

`seaborn` comes with 2 kinds of funtions:

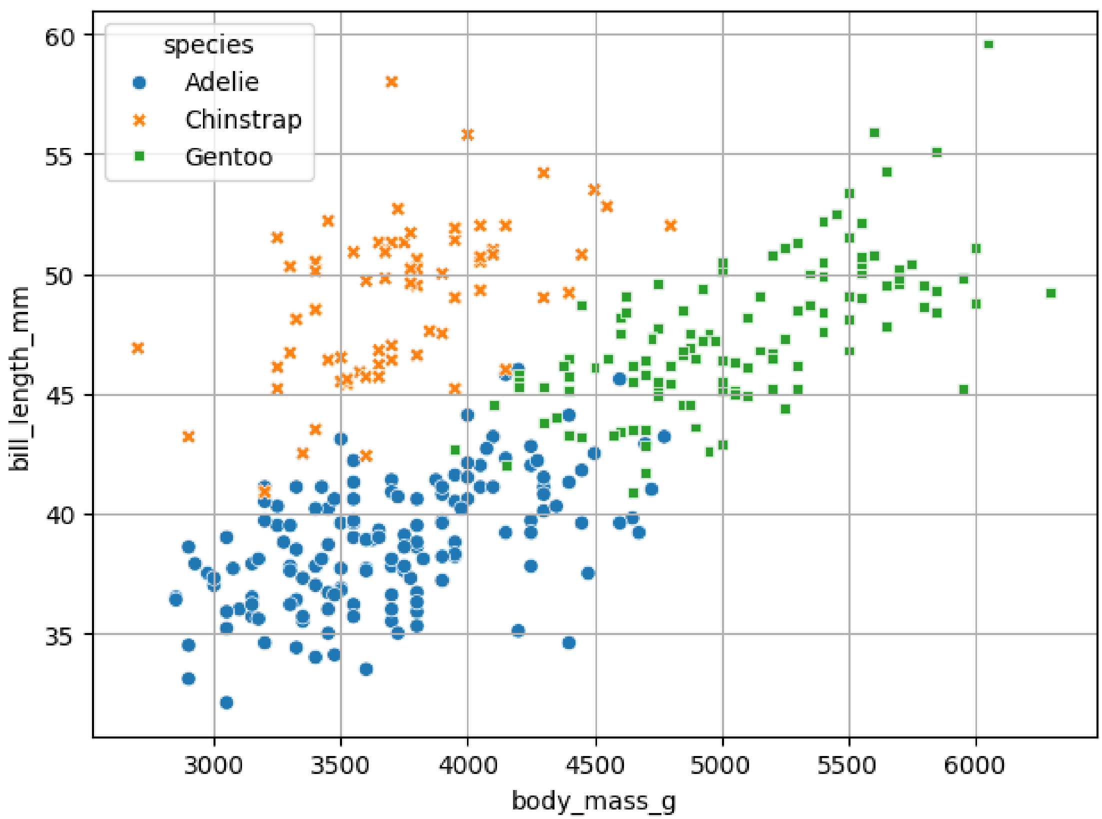
1. Functions that create only one ax to plot the data
2. Functions that create several axes using arguments `row` and `col` (related to a `FacetGrid` ojects):
   - `row`: data differenciated according to the **row** of the ax object in the figure
   - `col`: data differenciated according to the **col** of the ax object in the figure

#### One `axes`

Let's go back to our pengouins. The bill length is plotted as a function of body mass, with a color code and style differenciation for species. The returned object is an ax object: it can be customized if needed.

```
[7]: import seaborn as sns
     sns.scatterplot(df, x='body_mass_g', y='bill_length_mm', hue='species',␣
      ↪style='species')
```

```
[7]: <Axes: xlabel='body_mass_g', ylabel='bill_length_mm'>
```

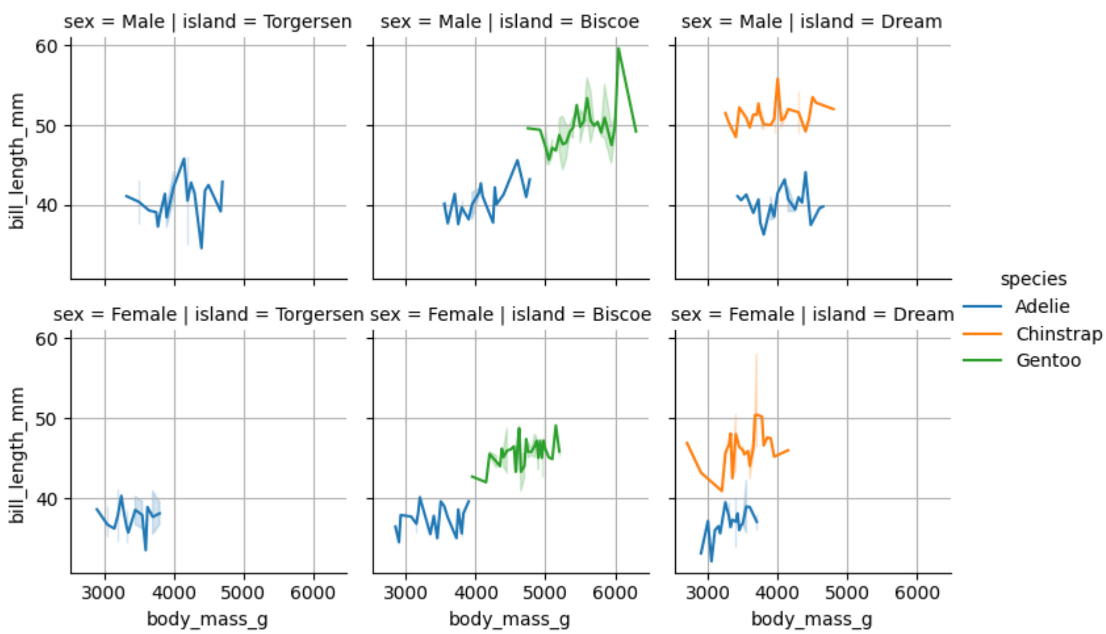



#### Several `axes`

**Complete example** Here after, the bill length is plotted:

- as a function of body mass
- with a color code for species
- with sex differenciation along rows
- with island differenciation along columns

[8]:
```
import seaborn as sns
import matplotlib
fg = sns.relplot(df, kind='line', x='body_mass_g', y='bill_length_mm',␣
↪hue='species', row='sex', col='island',
                 height=2.5, aspect=1)
```




Notes:

- `relplot` needs `kind='line'` to behave as `lineplot`
- `height` is the height of each *subplot*, i.e. each ax.

  `aspect` is the width/height ratio.
- Light color zones represent uncertainties. Indeed, given a tuple of (species, sex, island, mass), there are several individuals.

Let's investigate the uncertainties zones:

[9]:
```
df_ = df.groupby(['body_mass_g', 'species', 'sex', 'island']).count()
df_[(df_['bill_depth_mm']!=1)|(df_['flipper_length_mm']!=1)].head(3)
```

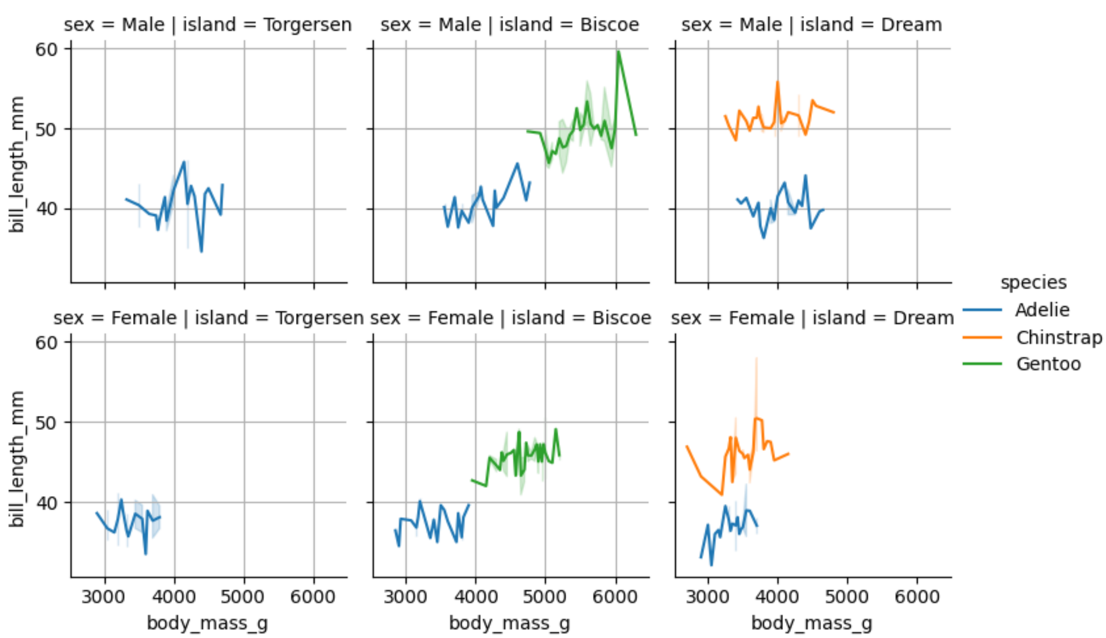
[9]:
```
                                            bill_length_mm  bill_depth_mm  \
body_mass_g species sex    island
```

```
2850.0      Adelie  Female Biscoe                   2             2
3000.0      Adelie  Female Dream                    2             2
3050.0      Adelie  Female Torgersen                3             3

                                         flipper_length_mm
body_mass_g species sex    island
2850.0      Adelie  Female Biscoe                        2
3000.0      Adelie  Female Dream                         2
3050.0      Adelie  Female Torgersen                     3
```

Finally, note that `relplot` returned a `FacetGrid` instance. It has an `axes` attribute (numpy array) that can be used to customize plots.

[9]:
```
axes = fg.axes
axes[1, 2].grid()
axes[1, 2].get_figure()  # needed for Jupyter
```

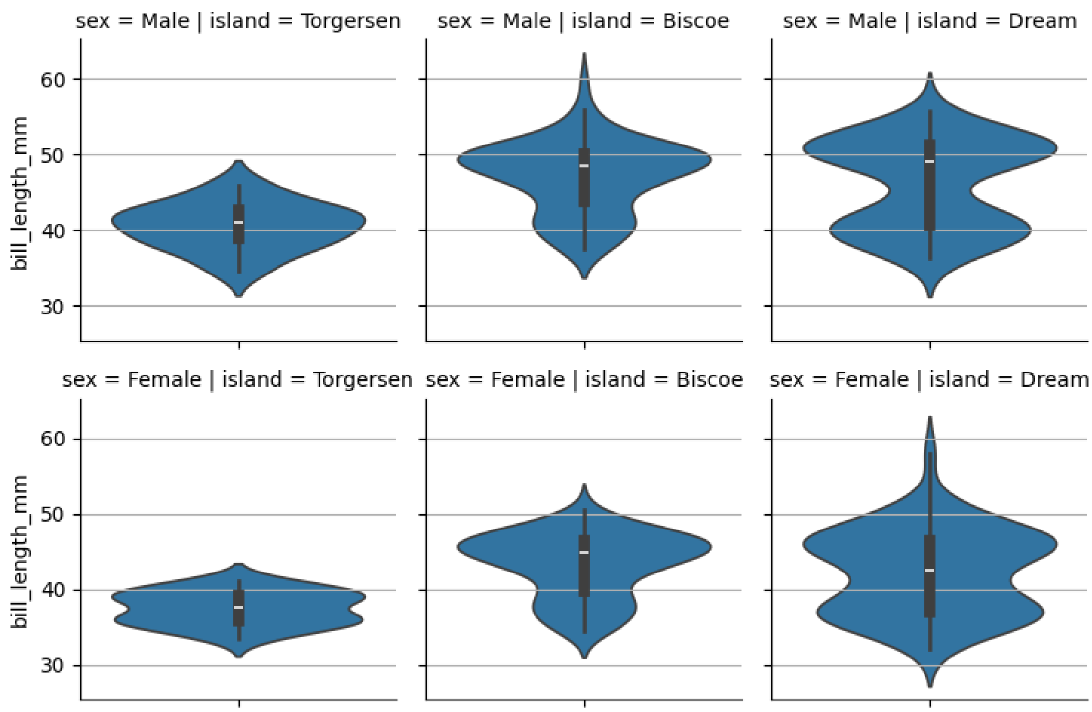
[9]:




**Other chart types** Many charts types can be created using `seaborn` (see the gallery).

For instance, let's create a `violinplot` to get a statistcial approach of data:

[10]:
```
fg = sns.catplot(df, kind='violin', y='bill_length_mm',  row='sex', col='island',
                height=2.5)
```


sex = Male | island = Torgersen
sex = Male | island = Biscoe
sex = Male | island = Dream
sex = Female | island = Torgersen
sex = Female | island = Biscoe
sex = Female | island = Dream
bill_length_mm
bill_length_mm
30
40
50
60

# Chapter 13

# Typical problems

## 13.1 Introduction [easy]

### 13.1.1 Many kinds of scientific problems

The scientific questions that arise in biology, mecanics, etc... are all different from each other. Moreover, some work are more experimental-based and others have a strong numerical dimension:

- equations solving
- search for functions optimum
- statistical analysis
- real time data acquisition
- etc...

### 13.1.2 Some dedicated libraries

Python is suitable for scientific computing:

1. Everything you interact with in Python an object with methods and attributes
2. To solve a problem:
    1. one define some objects that represent some mathematical or physical properties.
    2. these objects interacts with each other using a well documented API

The definition of suitable objects can be difficult (step 2.A). Thus, hundreds of open source dedicated packages did it for you. `numpy`, `pandas` and `matplotlib` are particular examples of these since many libraries are built on top of them.

For instance, a `numpy` array is of type `np.ndarray`: it can store some temperature values which average can be calculated using `.mean()`.

### 13.1.3 Some very common problems

Some packages are very famous in scientific computing. Let's focus on:

- `scipy`: typically used in optimization problems
- `scikit-learn`: typically used in machine learning problems

- `sympy`: designed for natural mathematical processing

## 13.2 Function minimization [easy]

### 13.2.1 Case study definition

Here is a 2-variable function:

$$f : (x, y) \rightarrow xy + (x - 4)^2 + (y + 3)^2$$

Let's determine the arguments that minimize this function.

### 13.2.2 Code

#### Simple example

The relevant function is `minimize` from package `scipy.optimize`.

[1]:
```
from scipy.optimize import minimize
import numpy as np
```

It takes 2 mandatory arguments:

- function to minimize
- initial values for the minimisation

Here after, function `f` takes as input a `numpy` array of size 2.

[2]:
```
def f(X):
    x, y = X
    return x*y + (x-4)**2 + (y+3)**2

initial = np.array([4, -3])
```

Then `minimize` is called:

[3]:
```
result = minimize(f, initial)
```

It returns a `OptimizeResult` object:

[4]:
```
print(type(result))
result
```

```
<class 'scipy.optimize._optimize.OptimizeResult'>
```

[4]:
```
  message: Optimization terminated successfully.
  success: True
   status: 0
      fun: -24.333333333333265
        x: [ 7.333e+00 -6.667e+00]
      nit: 5
      jac: [ 0.000e+00 -4.768e-07]
 hess_inv: [[ 6.667e-01 -3.333e-01]
            [-3.333e-01  6.667e-01]]
```

```
    nfev: 21
    njev: 7
```

Let's extract the *argmin* values:

[5]:
```
x_min, y_min = result.x
x_min, y_min
```

[5]:
```
(7.333333584713669, -6.666666929228556)
```

### Additional arguments

It may happen that function `f` has more arguments than the one we want to minimize along, i.e. that some arguments are fixed. For instance, let's add the $m$ variable:

$$f : (x, y, m) \rightarrow \left(xy + (x-4)^2\right)^m + (y+3)^2$$

When defining `f`, $m$ must not be given in `X` (that holds $x$ and $y$) but in a separate variable:

[6]:
```
def f(X, m):
    x, y = X
    return (x*y + (x-4)**2)**m + (y+3)**2
```

Then an argument `args` is used in `minimize` to set $m$ value. Note that a `tuple` is required even if there is only one fixed variable:

[7]:
```
minimize(f, initial, args=(1, )).x
```

[7]:
```
array([ 7.33333358, -6.66666693])
```

[8]:
```
minimize(f, initial, args=(2, )).x
```

[8]:
```
array([ 1.72508278, -3.00000009])
```

### Specify the jacobian

Specifying the jacobian is a way to fasten/improve minimization in **some cases**.

This is done by modifying `f`. `f` now returns:

- the function value itself (same than previously)
- the partial derivatives that define the jacobian

Let's compute these derivatives (without the $m$ case).

With respect to $x$:

$$\frac{\partial f}{\partial x} : (x, y) \rightarrow y + 2(x-4)$$

With respect to $y$:

$$\frac{\partial f}{\partial y} : (x, y) \to x + 2(y + 3)$$

[9]:
```python
def f(X):
    x, y = X
    value = x*y + (x-4)**2 + (y+3)**2
    jac = np.array([y + 2 *(x-4), x+ 2 *(y+3)])
    return value, jac

x_min, y_min = minimize(f, initial, jac=True).x
x_min, y_min
```

[9]: (7.333333333333333, -6.666666666666667)

Note that the order in `jac` is the same as in `X` ($x$ first, then $y$).

### 13.2.3 Plot the result

[10]:
```python
import matplotlib.pyplot as plt
```

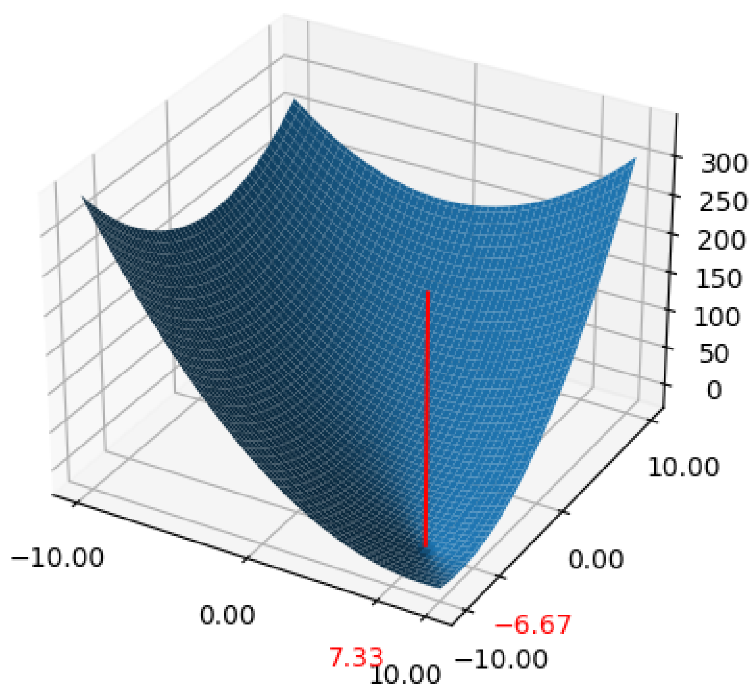
#### Data definition

[11]:
```python
x = np.linspace(-10, 10, 100)
y = np.linspace(-10, 10, 100)
xx, yy = np.meshgrid(x, y)
zz, _ = f([xx, yy])    # gradient is not needed here
```

#### Plotting

[12]:
```python
# surface only
fig, ax = plt.subplots(subplot_kw={'projection':'3d'})
ax.plot_surface(xx, yy, zz)



# red tick label for x_min and y_min
x_ticks = [-10, 0, 10] + [x_min]
ax.set_xticks(x_ticks)
x_ticklabels = ax.get_xticklabels()
for tick, label in zip(x_ticks, x_ticklabels):
    if tick == x_min:
        label.set_color('red')

y_ticks = [-10, 0, 10] + [y_min]
ax.set_yticks(y_ticks)
y_ticklabels = ax.get_yticklabels()
for tick, label in zip(y_ticks, y_ticklabels):
    if tick == y_min:
```

```
        label.set_color('red')

# vertical line at solution point
z, _ = f([x_min, y_min])
marker, line, _ = ax.stem([x_min], [y_min], [z], bottom=300)
line.set_color('red')
```



## 13.3 Function differenciation [medium]

### 13.3.1 Case study definition

Let's define a function with a complicated expression:

\$ f: (x, y) $\rightarrow \cos\left[(xy + (x-4)^2 + \arctan\left((y+3)^2\right))x\right]$\$

With $x > 0$ and $y > 0$.

`sympy` wil be used to compute partial derivatives of this function.

### 13.3.2 Key idea of `sympy`

`sympy` can handle mathematical objects in a formal way, i.e. **without using numerical discretization but relying on the properties of the defined objects**.

With `sympy`, objects are defined with a mathematical meaning that goes beyond the software meaning of traditional Python code. Then some operations are performed on these objects:

- differenciation
- integration
- limits
- etc…

### 13.3.3 Code

#### Mathematical objects declaration

Let's define two mathematical variables `x` and `y`. `sympy` is told these variables must take positive real values, using `positive=True`.

[1]:
```
import sympy as sym

x = sym.Symbol('x', positive=True)
y = sym.Symbol('y', positive=True)
```

Then the function is defined:

[2]:
```
f = sym.Lambda((x, y),
               sym.cos((x*y + (x-4)**2 + sym.atan((y+3)**2))*x))
f
```

[2]: $\left((x,\ y) \mapsto \cos\left(x\left(xy + (x-4)^2 + \operatorname{atan}\left((y+3)^2\right)\right)\right)\right)$

**Be careful!** Functions cos, atan and log are the one of `sympy`!

The function can be evaluated at a specific point:

[3]:
```
function_call = f(2, 4)
```

Yet, `evalf` must be used to get a numerical approximation. `evalf` returns a `sympy`-related type, it can be converted using `float`:

[4]:
```
print(type(function_call.evalf()))
print(float(function_call.evalf()))
```

```
<class 'sympy.core.numbers.Float'>
-0.3868788252007259
```

#### Operations on defined objects

Let's compute the partial derivative of `f` with respect to $x$, using `diff`:

[5]:
```
f(x, y).diff(x)
```

[5]:
$$-\left(xy + x\left(2x + y - 8\right) + \left(x - 4\right)^{2} + \operatorname{atan}\left(\left(y + 3\right)^{2}\right)\right)\sin\left(x\left(xy + \left(x - 4\right)^{2} + \operatorname{atan}\left(\left(y + 3\right)^{2}\right)\right)\right)$$

Or compute the second derivative with respect to $x$ and then the first derivative with respect to $y$:

[6]:
```
der = f(x, y).diff(x, 2, y)
der
```

[6]:
$$-2x\left(x + \frac{2\left(y + 3\right)}{\left(y + 3\right)^{4} + 1}\right)\left(3x + y - 8\right)\cos\left(x\left(xy + \left(x - 4\right)^{2} + \operatorname{atan}\left(\left(y + 3\right)^{2}\right)\right)\right) +$$
$$x\left(x + \frac{2\left(y + 3\right)}{\left(y + 3\right)^{4} + 1}\right)\left(xy + x\left(2x + y - 8\right) + \left(x - 4\right)^{2} + \operatorname{atan}\left(\left(y + 3\right)^{2}\right)\right)^{2}\sin\left(x\left(xy + \left(x - 4\right)^{2} + \operatorname{atan}\left(\left(y + 3\right)^{2}\right)\right.\right.$$
$$4\left(x + \frac{y + 3}{\left(y + 3\right)^{4} + 1}\right)\left(xy + x\left(2x + y - 8\right) + \left(x - 4\right)^{2} + \operatorname{atan}\left(\left(y + 3\right)^{2}\right)\right)\cos\left(x\left(xy + \left(x - 4\right)^{2} + \operatorname{atan}\left(\left(y + 3\right)^{2}\right)\right.\right.$$
$$2\sin\left(x\left(xy + \left(x - 4\right)^{2} + \operatorname{atan}\left(\left(y + 3\right)^{2}\right)\right)\right)$$

In this expression, $x$ and $y$ are still unknown. Let's replace $x$ using `subs`:

[8]:
```
der = der.subs({x: 5})
der
```

[8]:
$$-10\left(y + 7\right)\left(2y + 5\left(y + 3\right)^{4} + 11\right)\cos\left(25y + 5\operatorname{atan}\left(\left(y + 3\right)^{2}\right) + 5\right) - 2\left(\left(y + 3\right)^{4} + 1\right)\sin\left(25y + 5\operatorname{atan}\left(\left(y + 3\right)^{2}\right.\right.$$

#### `numpy` conversion

The evaluation of `der` is not fast. Thus it can be converted to a `numpy` function using `lambdify`. Then evaluation on arrays is possible:

[9]:
```
import numpy as np
numpy_func = sym.lambdify(y, der)
Y = np.linspace(0, 10, 5)
numpy_func(Y)
```

[9]:
```
array([  -1618.36732557,   -8543.34409677,  -44619.86174987,
        -132966.53918376, -279318.28704865])
```

### 13.3.4 When to use `sympy`

`sympy` can provide exact mathematical solutions **for simple problems only**. Thus, it can be used to check the results of some very specific calculation steps performed in a more numerical way.

## 13.4 Linear regression [easy]

### 13.4.1 Case study definition

Here is our pengouins database:

[1]:
```
import pandas as pd
df = pd.read_csv(r'../3__matplotlib/data.csv')
df.head(5)                    # show only the first five rows
```

[1]:
```
  species     island  bill_length_mm  bill_depth_mm  flipper_length_mm  \
0  Adelie  Torgersen            39.1           18.7              181.0
1  Adelie  Torgersen            39.5           17.4              186.0
2  Adelie  Torgersen            40.3           18.0              195.0
3  Adelie  Torgersen             NaN            NaN                NaN
4  Adelie  Torgersen            36.7           19.3              193.0

   body_mass_g     sex
0       3750.0    Male
1       3800.0  Female
2       3250.0  Female
3          NaN     NaN
4       3450.0  Female
```

Let's search for a linear dependance of the pengouin mass (`'body_mass_g'`, M) in the following variables:

- `'bill_length_mm'`, L
- `'bill_depth_mm'`, D
- `flipper_length_mm'`, F

The linear expression is the following:

$$M = \alpha_1 L + \alpha_2 D + \alpha_3 F + \beta$$

[2]:
```
M = 'body_mass_g'
L = 'bill_length_mm'
D = 'bill_depth_mm'
F = 'flipper_length_mm'
```

### 13.4.2 Code

The `scikit-learn` package includes a `LinearRegression` class to solve this problem;

#### Removing `nan` values

[3]:
```
cond = df[[M, L, D, F]].isna().any(axis=1)    # every row where
                                              # at least one value is nan
df = df[~cond]                                # keep the other rows,
                                              # '~' does the negation of `cond`
```

**Performing the regression**

A `LinearRegression` instance is first created. Then it's `fit` method is called directly using the columns of the 'DataFrame:

[11]:
```
from sklearn.linear_model import LinearRegression

model = LinearRegression()
_ = model.fit(df[[L, D, F]], df[M])
```

Results are **stored in the `model` instance**. Let's read the coefficients:

[12]:
```
alpha_1, alpha_2, alpha_3 = model.coef_.round(1)
beta = model.intercept_.round(1)
print(alpha_1, alpha_2, alpha_3, beta)
```

```
4.2 20.0 50.3 -6424.8
```

The regression quality (determination coefficient) is given by `score`:

[13]:
```
model.score(df[[L, D, F]], df[M])
```

[13]:
```
0.7614704841272493
```

**Using as a predictor** Using the fit coefficient, one can now go the reverse way: define the mass $M$ from the 3 other variables $(L, D, F)$. This is done using `predict`:

[14]:
```
import numpy as np
L_predict = [40, 45]
D_predict = [15, 20]
F_predict = [200, 275]

# `T` takes the transpose of the array, because data must be column-wise
to_predict = np.array([L_predict, D_predict, F_predict]).T
model.predict(to_predict)
```

```
/home/nerotb/Documents/5-Donnees_techniques/Python/310_base/lib/python3.10/site-
packages/sklearn/base.py:465: UserWarning: X does not have valid feature names,
but LinearRegression was fitted with feature names
  warnings.warn(
```

[14]:
```
array([4096.29544534, 7987.54383621])
```

# Part V

# Algorithmic complexity

# Chapter 14

# Problem size estimation

## 14.1 Theoretical analysis [advanced]

### 14.1.1 Basics of algorithmic complexity

Each code instruction consists in several elementary operations that involve different components of the computer:

- CPU/GPU: the fastest
- memory: quite fast
- disk: very slow

The running time of each operation is variable. **Minimizing the algorithmic complexity** is chosing the instructions that:

- run fast
- do what we want to do

### 14.1.2 Overhead and variable running times

#### Key idea

In a typical scientific problem, the running time increases depending on some parameters but a constant time always exists.

#### Simple example

In this part, a fake problem is built and it's code is analysed in order to understand how each instruction contributes to the overall algorithmic complexity.

**Case study definition** Let's define a **naive** function that sums the first `n` integers.

[69]:
```
def sum_integers(n):
    print('Summing the n first integers')
    total = 0
    for k in range(n):
        total += k
```

```
    print(f'The sum is: {total}')
```

### Time complexity decomposition

**Intuitive approach** What is the time complexity of this code? There are two types of operations:

- operations that do not depend on `n`, i.e. the **size of the problem**
  - calls to `print` at the beginning and at the end
  - the creation of a variable `total`
  - the creation of a `range` instance
- `n` iterations in the `for` loop. Each iteration includes:
  - an incrementation of `k`
  - the update of `total`

Thus one understands that running time will never be near-zero even for small `n` values. Conversely, the overhead run time is negligible for large `n` values.

This shows that the optimization of a code depends on the typical usage one intend to have of this code.

**Math approach** The `sum_integers` function performs:

- $C_1$ operations
- $C_2$ opérations for each iteration of the `for` loop

Eventually time complexity can be formulated this way:

$$C = C_1 + C_2 \times n$$

What is of interest in most cases is the evolution of the complexity with `n`, thus $C$ becomes $C(n)$. We note that, asymptotically, for large values of `n`:

$$C(n) = O(n)$$

### Advanced example

Let's add to the previous example a second `for` loop inside the first one:

[1]:
```
def sum_integers_difficult(n):
    total = 0
    total2 = 0
    for k in range(n):
        total += k
        for j in range(k+1):
            total2 += j
    return total
```

The following operations are performed:

- $C_1$ operations (overhead)
- for each iteration in the first loop (`n` iterations):
  - $C_2$ operations (incrementation of `total`)
  - for each iteration in the second loop (`k+1` iterations): $C_3$ operations (incrementation of `total2`)

Thus time complexity becomes:

$$C(n) = C_1 + \sum_{k=0}^{n-1} \left[ C_2(k) + \sum_{j=0}^{k} C_3(k,j) \right]$$

But $C_2(k)$ is almost independant from $k$ ($C_2(k) \simeq C_2$) and similarly $C_3(k,j) \simeq C_3$. Thus:

$$C(n) = C_1 + n \times C_2 + \frac{n(n+1)}{2} \times C_3$$

$$C(n) = C_1 + \left( C_2 + \frac{C_3}{2} \right) \times n + \frac{C_3}{2} \times n^2$$

Thus **asymptotically**:

$$C(n) = O(n^2)$$

**When `n` is doubled, running time is multiplied by 4.**

### Hidden operations

In the previous examples all instructions were rather explicit: simple loops and incrementations, no advanced function calls.

A complex code can include several calls to unknown functions. Thus the preferred way is to work at **the function level**:

- **User defined functions**: think about the time complexity of your function: overhead and asymptotic behaviour.
- **Other functions**: if the documentation says nothing about time complexity, you can make simple hypothesis to define lower and upper bounds of this complexity. For instance, the sum of a `numpy` array of `n` elements using `array.sum()` implies as many operations as there are elements in the array (`n`), thus one can expect a time complexity $O(n)$.

## 14.2 Typical limitations [advanced]

### 14.2.1 Introduction

Elementary operations can be grouped in 2 categories:

1. those that rely on 'external' resources, which are slow:
   - hard drives
   - network
2. thos that rely on 'internal' resources, which are fast
   - CPU/GPU
   - memory

The codes that run slowly due to the first family of causes are called **IO bound** problems. The other one are **CPU bound**.

### 14.2.2 Example of a *IO bound* problem

Let `f` be defined as:

[1]:
```
import pandas as pd
import numpy as np

def f_IO(n, k):
    # CPU-like tasks
    arr1 = np.random.rand(n)
    arr1 = arr1**2 + arr1 * 5 + np.exp(arr1-10)

    arr2 = np.random.rand(k)
    sr = pd.Series(arr2)

    # IO-like tasks
    sr.to_csv('data.csv')
```

This function creates 2 arrays of size `n` and `k`, do some mathematical operations on the first array and write the second one on disk.

#### Theorical time complexity

In theory, the function has the following time complexity:

$$C(n, k) = C_1 + C_2 \times n + C_3 \times k$$

With:

- $C_1$ overhead run time
- $C_2$ operations proportional to `n`: creation of `arr1`, various calculations using `arr1`
- $C_3$ operations proportional to `k`: creation of `arr2`, creation of the `Serie` object, export to disk

Asymptotically, time complexity grows the same way with respect to `k` or `n`. This is described by:

$$C(n,k) = O(n) + O(k) = O(n+k)$$

#### Real time complexity

Yet, the $C_3$ coefficient is much bigger than $C_2$ since **it is related to disk operations.** Thus **the asymptotical behaviour corresponds `n` values that are too large to correspond to any practical use**. Hence the real time complexity is much more something like:

$$C(n,k) = O(k)$$

#### Experimental running time

Let's measure the real running times of this function for:

- $n \in [10^3, 10^6]$
- $k \in [10^3, 10^6]$

Results, **in milliseconds**, are presented hereafter:

[5]:
```
pd.DataFrame(data = {'$k=10^3$': [4.320, 48.6], '$k=10^6$': [2330, 2360]},
             index=['$n=10^3$', '$n=10^6$'])
```

[5]:

| | $k=10^3$ | $k=10^6$ |
|---|---|---|
| $n=10^3$ | 4.32 | 2330 |
| $n=10^6$ | 48.60 | 2360 |

Interpretation: the **contribution of `n` is negligible compared to the one of `k`**. A factor 1000 increase of `n` adds only a few milliseconds to the running time.

### 14.2.3 Conclusion

The theoretical estimation of a problem time complexity can be very difficult.

1. Some simple estimators are:

   - search for overhead run times
   - search for disk/network operations, in opposition to CPU/RAM

2. Sometimes, experimental measurements are a better a choice.

# Chapter 15

# Profile one's code

## 15.1 Time complexity [medium]

### 15.1.1 Optimisation methodology

#### Theory

The following methodology can be used to reduce the time complexity of any code:

1. Evaluate the total running time (timing)
2. Sort each function $f$ by running time (profiling)

$$t(f)$$

3. Estimate for each function the possible room for improvement

$$\Delta t(f)$$

4. Optimize the function $f$ for which $t(f) \times \Delta t(f)$ is the smallest.
5. Check that the code is still operating properly
6. Go back to step 1 if running time is still too high

If running time is still not satisfactory and no more room exists for improvement, **you must think of another formulation of your scientific problem**. This is very time consuming, that's why you must wonder whether your formulation is adaptated **before** writing any code line.

#### Example

Let's assume we have some code that calls 3 functions:

- the first function call is 90% of total running time
- the second function call is 1%
- the third is 9%

**Achieving a 10% time reduction on the first function call (which seems possible) will have more effect than a 90% time reduction on the third call (which seems difficult)**

### 15.1.2 Tools

#### Presentation

Python comes with some tools to measure the running time of some code instructions. There are 2 steps:

1. Measuring the total running time: **timing**.

   This can be done using the `timeit` package.

2. Measuring the running time of each function call: **profiling**.

   This can be done using the `cProfile` package.

#### *iPython*

Instead of directly using `timeit` and `cProfile`, it is recommended to use the magick commands `%timeit` and `%prun`.

These commands require the Python shell to be a ***iPython*** shell.

***iPython*** can be used:

- by installing the `ipython` package and by opening a shell using the `ipython` command
- by installing either Jupyter notebook or Jupyter lab

In addition to the magic commands, ***iPython*** shells come with a better color code and indentation handling.

### 15.1.3 Example

#### Case study definition

Let's define a very naive function and analyse its algorithmic complexity. **This function is designed specifically not to be efficient**, it must not be used.

[13]:
```
def build_list(k):
    result = []
    for j in range(1, k):
        result.append(j)
    return result

def sum_list(var):
    s = 0
    for e in var:
        s += e
    return s

def f(n):
    assert n >= 3
```

```
    result = []
    for j in range(3, n):
        var = build_list(j)
        s = sum_list(var)
        result.append(s)
    prod = 1
    for e in result:
        prod *= e
```

What does this function do? It computes the product of the sum of $q-1$ first non zero integers, for $q \in [3, n-1]$:

$$f : n \to \prod_{q=3}^{n-1} \sum_{p=1}^{q-1} p$$

**Running time: timing**

Let's measure the total running time using `%timeit`. The percent sign `'%'` decribes ***iPython*** magic commands: it tells the Python interpreter that this is not a common variable.

`timeit` measures the running time several times in a row in order to remove statistical noise.

`f` function expects a value for variable `n`. Let's suppose `n=10000` is a typical value for the problem we want to solve.

[14]:
```
%timeit f(10000)
```

```
5.84 s ± 404 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

The command described above performs 7 runs (`-r 7`) of calling `f(10000)` 1 time (`-n 1`). Average is done and the mean and std of running time among the different runs is returned. `%timeit` decides alone what are the good values for `-r` and `-n`, but it can also be specified:

[16]:
```
%timeit -r 2 -n 2 f(10000)
```

```
5.7 s ± 195 ms per loop (mean ± std. dev. of 2 runs, 2 loops each)
```

An object mode also exists: instead of printing the results, they are stored in a dedicated instance:

[17]:
```
running_time = %timeit -o -r 2 -n 2 f(10000)
print(running_time.all_runs)
print(running_time.best)
print(running_time.worst)
```

```
5.75 s ± 54.6 ms per loop (mean ± std. dev. of 2 runs, 2 loops each)
[11.390800094988663, 11.609388401004253]
5.695400047494331
5.8046942005021265
```

**Running time: profiling**

*Profiling* is done using `%prun`. The result is saved in a file (`-T` option).

```
[ ]: %prun -T prun_results.txt f(10000)
```

```
         50014995 function calls in 10.999 seconds

   Ordered by: internal time

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
     9997    5.946    0.001    8.356    0.001 3227873083.py:1(build_list)
 49994997    2.411    0.000    2.411    0.000 {method 'append' of 'list' objects}
     9997    2.353    0.000    2.353    0.000 3227873083.py:7(sum_list)
        1    0.289    0.289   10.999   10.999 3227873083.py:13(f)
        1    0.000    0.000   10.999   10.999 <string>:1(<module>)
        1    0.000    0.000   10.999   10.999 {built-in method builtins.exec}
        1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler' objects}
```

There are several columns:

- on the right, the executed function (file name:line number)
- 1st column on the left (`ncalls`): number of times this function is called
- 2nd column (`tottime`): the time spent in this function (excluding the time spent in subfunctions)
- 3rd column (`percall`): the sum of `tottime` divided by `ncalls`

**Optimize the relevant code section**

The profiling results show that the most time-consuming function is `build_list`. And this part seems easy to optimize. Thus, optimization of time complexity starts here.

### 15.1.4 Reminder

Optimization is always done at **the very last moment**, once the code is stable and no functionnalities are to be added.

## 15.2 Memory usage [advanced]

### 15.2.1 Size of numpy arrays

#### Introduction

Different Python instances occupy different amounts of memory.

Regarding `numpy` arrays, the memory footprint mainly depends on the data type of the data (`dtype`). The knowledge of typical memory use of some dtypes can help estimate the memory use of any scientific problem.

#### `nbytes` attribute

The memory footprint of the data of `np.ndarray` objects is determined using the `nbytes` attribute.

#### Example: array of integers

Let's define an array with dtype `uint8`:

- 'u': *unsigned*
- 'int': *integers*
- 8: stored on 8 bits (8 bits=1 byte)

This data type can store integers from 0 to 255 (included) since 8 bits can store $2^8 = 256$ values.

[1]:
```
import numpy as np


arr = np.arange(1000, dtype='uint8')
arr.nbytes
```

[1]:
```
1000
```

`arr` has 1000 values each occupying 1 byte, thus the total is 1000 bytes.

#### Example: array of floating values

The default behaviour of `numpy` is storing data using a `np.float64` dtype. This data type takes 8 bytes (hence 64 bits) per value.

[52]:
```
arr = np.random.rand(1000)
print(arr.dtype)
print(arr.nbytes)
```

```
float64
8000
```

**Using this data type for an array of 125 million values would use 1 GB of memory**.

Recall that a recent computer has typically 8 GB of memory yet **you must always keep some memory left to store intermediate results during computation**.

125 millions is a large number that can be obtained using multidimensionnal data: a 4 dimensions array with shape (100, 100, 100, 100) has 100 millions values.

### Specifying *dtype* to reduce memory use?

What about changing the dtype to gain some memory? In most cases, given our skills in computer sciences, this is often a bad idea!

Let's inspect some reasons for this.

**Smaller range of possible values** Let's see what are the numbers that can be represented using a specific dtype. For float values, this is done using `numpy.finfo`:

[53]:
```
finfo = np.finfo(np.float64)
finfo
```

[53]:
```
finfo(resolution=1e-15, min=-1.7976931348623157e+308,
max=1.7976931348623157e+308, dtype=float64)
```

`float64` dtype can handle values from `min` to `max`. Let's see what happens in other cases:

[54]:
```
np.array([1.79e308], dtype=np.float64)  # value is smaller than max possible␣
↪value
```

[54]:
```
array([1.79e+308])
```

[55]:
```
np.array([1.80e308], dtype=np.float64)  # value is larger than max possible␣
↪value
                                        # seen as infinite
```

[55]:
```
array([inf])
```

**Advanced**: Python floats are float64, and passing values in a float format have them evaluated by Python before `numpy`. Thus, a way to build arrays with more complete dtypes than float64 is to pass the values as strings:

[56]:
```
np.array(['1.80e308'], dtype=np.float128)  # a float128 can handle this value...
```

[56]:
```
array([1.8e+308], dtype=float128)
```

[57]:
```
np.array([1.80e308], dtype=np.float128)  # but beware of prior evaluation to inf␣
↪by Python
```

[57]:
```
array([inf], dtype=float128)
```

**Lower numerical resolution** Storing a float value in a low memory footprint dtype comes with **a resolution loss**. Thus, going from `np.float64` (default) to `np.float32` divides the memory footprint by 2 but strongly deteriorates the resolution.

[58]:
```
print(np.finfo(np.float64).resolution)
print(np.finfo(np.float32).resolution)
```

```
1e-15
1e-06
```

[59]:
```
np.array([1, 1e-15]).sum(dtype=np.float64)    # result of the operation
                                               # can be understood
                                               # with the resolution of float64
```

[59]:
```
1.000000000000001
```

[60]:
```
np.array([1, 1e-15]).sum(dtype=np.float32)    # result of the operation
                                               # comes with a loss of
                                               # resolution if float32 is used
```

[60]:
```
1.0
```

**Poorer CPU performances** Modern CPU are not optimized to work with unconventionnal floating points resolution. Thus, computation time can increase with memory efficient dtypes.

**Unexpected casting** `numpy` casting rules may be complex and may lead in unexpected results and having a fine control over all dtypes in a large problem is very time consuming during the development phase.

**Conclusion** For all the reasons presented above, the preferred way is **not to change the dtype**. In some cases, however, the `astype` method can be used.

### 15.2.2 References of Python instances (advanced)

#### Theory

A variable is only a name that is associated with a content in memory.

Sometimes this content is shared by several variables. All of these are references. Thus the two following operations are very different:

- copying all the memory content (deep copy)
- adding a reference to some memory content (reference)

Many functions and methods rely on references up to a point. For instance, `np.ravel(my_array)` will build a new array with some attributes different from the one of `my_array` (e.g. shape…) but keep the same memory content, i.e. a reference to the data of `my_array`.

That makes it pretty difficult to assign some memory use to a specific function call.

#### Howto

The `getrefcount` function of package `sys` **returns the number of Python instances that share the same memory content**.

Let's create an object stored in `var`.

The memory content of `var` is 2: one for `var` and one created during the call of `getrefcount`.

[61]:
```python
from sys import getrefcount
var = [(5, 4)]
getrefcount(var)
```

[61]:
```
2
```

Let's add a reference to `var`. Its ref count is incremented since `var2` now redirects to the memory content of `var`.

[62]:
```python
var2 = var
getrefcount(var)
```

[62]:
```
3
```

What if a deep copy is performed? The `var` refcount does not change.

[63]:
```python
var3 = var.copy()
print(getrefcount(var))
print(getrefcount(var3))
```

```
3
2
```

### Note

When creating `var3`, a new list is created (different memory address than the one of `var`). Yet, this list shares the content of `var` (the tuple)!

[64]:
```python
print(getrefcount(var[0]))
print(getrefcount(var3[0]))
```

```
3
3
```

### Garbage collection

Whenever no reference exists for a memory content, Python makes this space available for other use; This is called garbage collection.

[65]:
```python
print(getrefcount(var))
del var2
print(getrefcount(var))
del var
# memory content of `var` no longer exists
```

```
3
2
```

In the every day life, **scoping rules** of Python are such that a variable defined inside a function is detached from its memory content when exiting the function, thus there are not many cases where `del` should be called.

# Chapter 16

# Simple code sped up

## 16.1 Numpy [easy]

### 16.1.1 Introduction

`numpy` is much faster than native Python code relying on loops. The sped up factor can be up to 10 or 100.

### 16.1.2 Example

Let's compare the running speed of native Python and `numpy`. The defined function perform the following operations:

- create a list (or array) fill with `1`, of length `n`
- sum the values of this data container

[1]:
```
import numpy as np

def sum_python(n):
    var = [1 for _ in range(n)]
    sum(var)     # Python `sum` is different from
                 # `np.sum`

def sum_numpy(n):
    var = np.ones(n)
    np.sum(var)
```

Let's compare running times:

[2]:
```
%timeit sum_python(10000)
```

```
299 µs ± 8.65 µs per loop (mean ± std. dev. of 7 runs, 1,000 loops each)
```

[3]:
```
%timeit sum_numpy(10000)
```

```
9.41 µs ± 138 ns per loop (mean ± std. dev. of 7 runs, 100,000 loops each)
```

The `numpy` implementation is rougly 30 times faster than the Python one for `n=10000`.

### 16.1.3 Caution

Every code comes with a constant overhead run time.

**The use of `numpy` arrays comes with a pretty large overhead that is balanced only for large number of elements**

The code below makes this overhead explicit.

**Timing**

[ ]:
```
import pandas as pd

N = np.logspace(1, 8, 8).astype(int)  # n=10, n=100, ..., n=100 000 000
results = []
for n in N:
    timeit_python = %timeit -q -o sum_python(n)
    timeit_numpy = %timeit -q -o sum_numpy(n)
    results.append({'N': n,
                    'Python': timeit_python.average,
                    'Numpy': timeit_numpy.average})
```

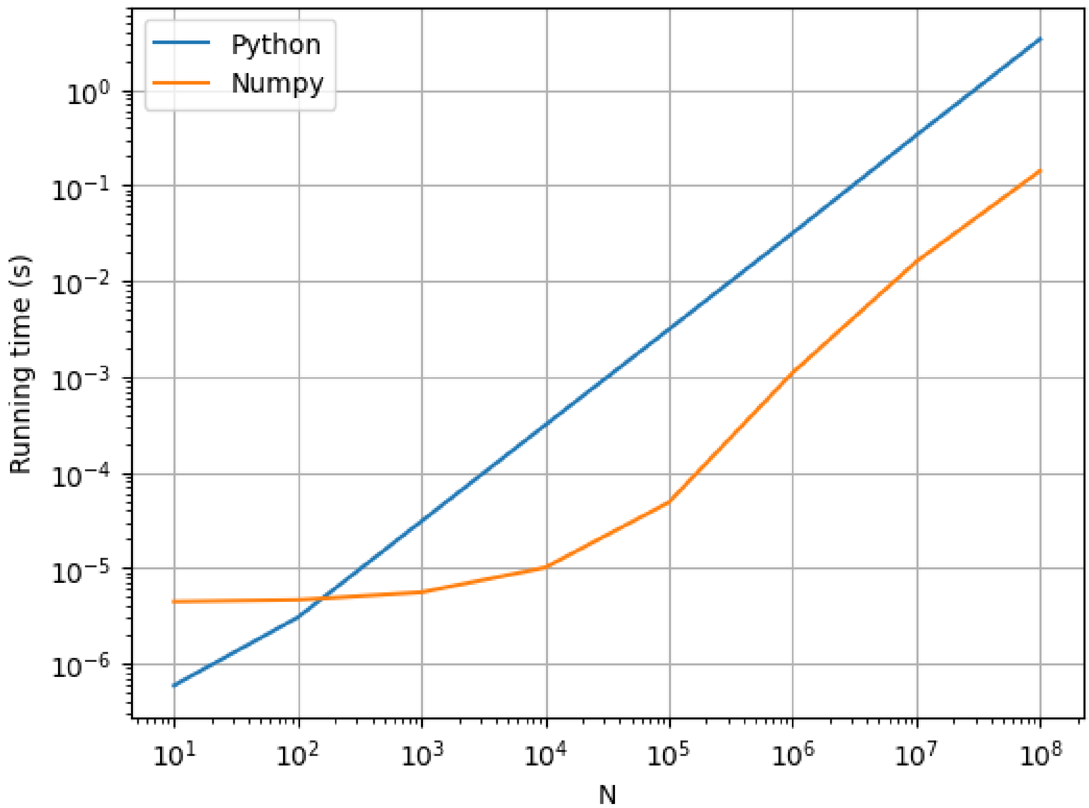
[8]:
```
df = pd.DataFrame(results).set_index('N')
df
```

[8]:

| N | Python | Numpy |
|---|---|---|
| 10 | 5.945477e-07 | 0.000004 |
| 100 | 3.047114e-06 | 0.000005 |
| 1000 | 3.083205e-05 | 0.000006 |
| 10000 | 3.113286e-04 | 0.000010 |
| 100000 | 3.114604e-03 | 0.000049 |
| 1000000 | 3.154092e-02 | 0.001097 |
| 10000000 | 3.319341e-01 | 0.015903 |
| 100000000 | 3.358813e+00 | 0.140852 |

**Plotting**

[9]:
```
ax = df.plot(logx=True, logy=True, grid=True, ylabel='Running time (s)')
```




**Analysis** Regarding the `numpy` version:

- it is 10 to 100 times faster than the Python version, **starting from** `n=300`
- for $n < 300$, the `numpy` running time is constant, which denotes a large overhead of `numpy` operations

## 16.2 Caching [medium]

### 16.2.1 Introduction

**Memoization** methods consist in storing an intermediate result and returning it whenever this is needed.

These methods reduce the time complexity but deteriorates memory use.

### 16.2.2 Implementation

#### Theory

In Python, the `lru_cache` decorator of the `functools` package is a way to do memoization. Recall that a decorator can be applied using `@` before the line of a function/class definition.

In the background, `lru_cache` stores the results of each function call and returns them when the function is called one more time with the same arguments.

`lru_cache` takes a `maxsize` argument so that only a certain number of function calls results are stored ('lru' = 'least recently used'). Whenever the function result has a low memory footprint, one can set `maxsize=None` to leverage the limit.

#### Example

**Code** Let's build a classical case study: the Fibonacci sequence. Let's define 2 versions:

- without memoization
- with memoization, using `lru_cache`

[1]:
```python
from functools import lru_cache

def fib_simple(n):
    if n < 2:
        return n
    return fib_simple(n-1) + fib_simple(n-2)

@lru_cache(maxsize=2**20)  # maxsize must be specified as a power of 2
def fib_memoized(n):
    if n < 2:
        return n
    return fib_memoized(n-1) + fib_memoized(n-2)
```

**Results**

[2]:
```python
%timeit fib_simple(10)
```

```
19.5 µs ± 1.36 µs per loop (mean ± std. dev. of 7 runs, 100,000 loops each)
```

[3]:
```python
%timeit fib_memoized(10)
```

```
88.1 ns ± 10.6 ns per loop (mean ± std. dev. of 7 runs, 10,000,000 loops each)
```

There is a time sped up of approximately a factor 220!

**Details**

the decorated function has a `cache_info()` method that gives some information regarding function calls:

[4]:
```
fib_memoized.cache_info()
```

[4]:
```
CacheInfo(hits=81111118, misses=11, maxsize=1048576, currsize=11)
```

Explanations:

- `hits`: number of function calls that were handled by the cache, i.e. for which the function result was already known and stored. This number is very large since `%imeit` performs several runs.
- `hits`: number of function calls that were not led by the cache. This typically corresponds to the first calls of the function, when the cache is empty.
- `currsize`: number of stored call results

### 16.2.3 Caution

`lru_cache` relies on a dictionary to store the results of a function call. Yet, keys of a dictionary must be hashable.

Thus, most of **mutable objects cannot be arguments of a function decorated with** `lru_cache`.

Below is a comparison of a `tuple` argument (immutable) and `list` argument (mutable, unhashable).

[5]:
```
import numpy as np

@lru_cache()
def memoized_function(arg):
    print(arg)
```

[6]:
```
memoized_function((1, 2, 3))
```

```
(1, 2, 3)
```

[7]:
```
memoized_function.cache_info()
```

[7]:
```
CacheInfo(hits=0, misses=1, maxsize=128, currsize=1)
```

[8]:
```
memoized_function([1, 2, 3])
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[8], line 1
----> 1 memoized_function([1, 2, 3])

TypeError: unhashable type: 'list'
```

### 16.2.4 When to use memoization

The `lru_cache` decorator must be used on functions that **meets the 3 following criterias**:

- it takes a long time to run
- it is frequently called with the same arguments (non mutables)
- it returns intermediate results that have no particular scientific meaning

# Chapter 17

# Advanced code sped up

## 17.1 Introduction [easy]

### 17.1.1 Sequential execution

In the the general case, Python instructions are ran **one after another**. If an instruction 'A' follows a 'B' instruction, 'B' must wait for 'A' to finish in order to start. For instance:

[1]:
```
from datetime import datetime
from time import sleep

def A():
    current_time = datetime.now().strftime('%H:%M:%S')
    print(f"[{current_time}] I am 'A', and I will sleep for 3 seconds")
    sleep(3)
    current_time = datetime.now().strftime('%H:%M:%S')
    print(f"[{current_time}] I am 'A', and my sleeping time was so good!")


def B():
    current_time = datetime.now().strftime('%H:%M:%S')
    print(f"[{current_time}] I am 'B'")

A()
B()
```

```
[19:16:58] I am 'A', and I will sleep for 3 seconds
[19:17:01] I am 'A', and my sleeping time was so good!
[19:17:01] I am 'B'
```

**Explanation**: 'B' must wait for the end of 'A' to start.

But 'A' is mainly useless! In fact, 'A' is doing something that **does not require the CPU** (`sleep`). Yet, **'A' won't let 'B' use the CPU** while it is sleeping.

### 17.1.2 Other execution modes

#### Threading

In the example above, it would be really cool to use the CPU while 'A' does not need it, though 'A' is running.

This is one side of what is called **concurrency**: several instructions are 'racing' to get some CPU computations as soon as they finished their non-CPU activity. In Python, the `threading` package can be used to write concurrent code.

A `thread` is some tasks that make sense to group together. One can start several threads that will access the CPU whenever it is available without having to wait for the other thread to be terminated.

This execution mode **provides speed up capabilities mainly for IO bound problems**. Indeed, while a thread is using the disk or waiting for the network, another one can use the CPU.

#### Multi-processing

Sometimes one may want some instructions to be ran in a completely independant way. The idea, in the example above, is that it would not be a problem for 'A' to occupy the CPU since another part of the CPU could be used to process 'B' at the same time. This is called **parallelism**. These 'parts' are the CPU cores. In Python, this is done using the `multiprocessing` package.

This execution mode can speed up several CPU intensive problems, i.e. **CPU bound**.

#### Limitations

**Memory** `threading` and `multiprocessing` are interesting but may lead to memory errors:

- using `threading`, 2 threads might need to access almost at the same time the same variable, which can lead to data corruption.
- using `multiprocessing`, 2 processes might want to share some memory content, but this is not a native behaviour since a process is not allowed to access the memory content of another one.

Some functions were defined to share variables at no risk. But using them can be difficult. Moreover, the risk of making a mistake is rather high and may lead to inacurrate scientific results.

For this reason, **concurrency and parallelism must be the last resort methods to speed up a code**.

**Overhead** Overhead running time associated with `multiprocessing` are **very large, especially for codes dealing with very large amount of data**. Consequently, multiprocessing is really useful for CPU-bound tasks that typically takes a few minutes to run. Conversely, threading comes with a lower overhead.

#### Advanced note

Python is a particular case regarding parallelism/concurrency. Indeed, the true limitation of Python is that a single Python process cannot run different instructions simultaneously on the

CPU, due to an internal limitation. This explains the limitations of `threading` and the need for `multiprocessing`.

In other languages, threads can run simultaneously on different cores of the CPU. Then the difference between threads and processes is more related to memory management.

## 17.2 Threading [advanced]

### 17.2.1 Simple example

#### Sequential version

The `sleeper` function below sleeps for `n` seconds. It is described by the variable `identifier`. Recall that during sleep time, the CPU is not used.

[1]:
```
from datetime import datetime
from time import sleep

def gt():                                        # get_time
    return datetime.now().strftime('%H:%M:%S')

def sleeper(n, identifier):
    print(f"[{gt()}] I am '{identifier:^8}', and I will sleep for {n} seconds")
    sleep(n)
    print(f"[{gt()}] I am '{identifier:^8}', and my sleeping time was so good!")
```

Then the `f_sequential` calls `sleeper` using differents arguments.

No surprise here, everything runs sequentially:

[2]:
```
args = [(3, 'First'), (2, 'Second'), (4, 'Third')]

def f_sequential():
    for arg in args:
        sleeper(*arg)

f_sequential()
```

```
[19:24:31] I am ' First  ', and I will sleep for 3 seconds
[19:24:34] I am ' First  ', and my sleeping time was so good!
[19:24:34] I am ' Second ', and I will sleep for 2 seconds
[19:24:36] I am ' Second ', and my sleeping time was so good!
[19:24:36] I am ' Third  ', and I will sleep for 4 seconds
[19:24:40] I am ' Third  ', and my sleeping time was so good!
```

#### Threaded version

**Code** Let's define a function `f_threading` that separate each call to `sleeper` in a dedicated thread.

The use of threads in Python is made simple by the `ThreadPoolExecutor` **class** of the `concurrent` package (rather than the `threading` package).

[3]:
```
from concurrent.futures import ThreadPoolExecutor

def f_threading(max_workers=2):
    with ThreadPoolExecutor(max_workers=max_workers) as executor:
```

```
        for arg in args:
            executor.submit(sleeper, *arg) # unpacking is needed here
                                           # because `arg` is a tuple
```

Notice `max_workers=2`. This means that at every time no more than 2 threads can be running (no matter they use the CPU or not).

The executor submits the call to `sleeper` for each argument tuple.

Here is the result:

[4]:
```
f_threading()
```

```
[19:24:40] I am ' First  ', and I will sleep for 3 seconds[19:24:40] I am '
Second ', and I will sleep for 2 seconds

[19:24:42] I am ' Second ', and my sleeping time was so good!
[19:24:42] I am ' Third  ', and I will sleep for 4 seconds
[19:24:43] I am ' First  ', and my sleeping time was so good!
[19:24:46] I am ' Third  ', and my sleeping time was so good!
```

**Explanation**

- The first call to `sleeper` is with `(3, 'First')`. `sleeper` performs the first `print` and then go to sleep for 3 seconds.
- While the first call is sleeping, it does not need the CPU. Thus the second call (with `(2, 'Second')`) starts: first `print` statement and then go to sleep too.
- Neither the `sleep` of the first call nor the one of the second call came to an end. Yet, executor is not allowed to start a 3rd thread due to `max_worker=2`. Thus, the executor wait for the second call to end (the shortest one) t perform the third call.
- The first call terminates.
- The third call terminates too.

Eventually the execution of `f_threading` took only 6 seconds, which is lower than the 9 seconds of the sequential version.

What if `max_workers` is set to 3?

[5]:
```
f_threading(max_workers=3)
```

```
[19:24:46] I am ' First  ', and I will sleep for 3 seconds
[19:24:46] I am ' Second ', and I will sleep for 2 seconds
[19:24:46] I am ' Third  ', and I will sleep for 4 seconds
[19:24:48] I am ' Second ', and my sleeping time was so good!
[19:24:49] I am ' First  ', and my sleeping time was so good!
[19:24:50] I am ' Third  ', and my sleeping time was so good!
```

**Notes**

In a real code, this is not a call to the `sleep` function that monopolize the CPU. It may be:

- the writing of a big file on disk

- the time waiting for some internet server response

### 17.2.2 Sharing some variables

#### Sequential version

**Code** Let's define a function, similar to `sleeper`, that modifies a **mutable** variable passed as an argument. This variable is a one-integer list, and the modification consists in incrementing this integer.

[6]:
```
def sleeper(n, identifier, var):
    actual_value = var[0]
    print(f"[{gt()}] I am '{identifier:^8}', and I will sleep for {n} seconds")
    sleep(n)
    var[0] = actual_value + 1
    print(f"[{gt()}] I am '{identifier:^8}', and my sleeping time was so good!")
    print(f"[{gt()}] I am '{identifier:^8}', and I modified var[0]: it was␣
 ↪{actual_value}, it is now {var[0]}!")
```

Let's assume that `var` is shared among all calls. Thus it is defined once before `args` and a reference is passed to `args` (this is not a copy).

[7]:
```
var = [10]
args = [(3, 'First', var), (2, 'Second', var), (4, 'Third', var)]
f_sequential()
```

```
[19:24:50] I am ' First  ', and I will sleep for 3 seconds
[19:24:53] I am ' First  ', and my sleeping time was so good!
[19:24:53] I am ' First  ', and I modified var[0]: it was 10, it is now 11!
[19:24:53] I am ' Second ', and I will sleep for 2 seconds
[19:24:55] I am ' Second ', and my sleeping time was so good!
[19:24:55] I am ' Second ', and I modified var[0]: it was 11, it is now 12!
[19:24:55] I am ' Third  ', and I will sleep for 4 seconds
[19:24:59] I am ' Third  ', and my sleeping time was so good!
[19:24:59] I am ' Third  ', and I modified var[0]: it was 12, it is now 13!
```

**Explanation** The sequential version is ok. There are 3 function calls, each call adds 1 to var[0] which is initialized with 10, thus we end up with $13 = 10 + 3$.

#### Naïve threading

**Code** Let's call `f_threading` using these new arguments.

[8]:
```
var[0] = 10
f_threading()
```

```
[19:24:59] I am ' First  ', and I will sleep for 3 seconds
[19:24:59] I am ' Second ', and I will sleep for 2 seconds
[19:25:01] I am ' Second ', and my sleeping time was so good!
[19:25:01] I am ' Second ', and I modified var[0]: it was 10, it is now 11!
```

```
[19:25:01] I am ' Third  ', and I will sleep for 4 seconds
[19:25:02] I am ' First  ', and my sleeping time was so good!
[19:25:02] I am ' First  ', and I modified var[0]: it was 10, it is now 11!
[19:25:05] I am ' Third  ', and my sleeping time was so good!
[19:25:05] I am ' Third  ', and I modified var[0]: it was 11, it is now 12!
```

**Explanation** What is going on?

- `'First'` stores the value `var[0]` in `actual_value`.
- While `'First'` is sleeping, `'Second'` modifies `var[0]`. `var[0]` now redirects to another integer, which is different from what `'First'` stored in `actual_value`.
- `First` wakes up and modifies `var[0]` according to the obsolete `actual_value`. Thus the incrementation process is broken.
- etc …

### Protected threading

The `var` variable must be protected, for instance using a `Lock` object (or similarly, a `Semaphore`).

A `Lock` object can be used only by one thread at a time. A thread must 'acquire' the permission and 'release' it.

Let's redefine `sleeper`.

[9]:
```python
from threading import Lock

var[0] = 10

def sleeper(lock, n, identifier, var):
    with lock:
        actual_value = var[0]
        print(f"[{gt()}] I am '{identifier:^8}'",
              f" and I will sleep for {n} seconds")
        sleep(n)
        var[0] = actual_value + 1
        print(f"[{gt()}] I am '{identifier:^8}",
              " and my sleeping time was so good!")
        print(f"[{gt()}] I am '{identifier:^8}', and I modified var[0]:",
              f" it was {actual_value}, it is now {var[0]}!")

def f_threading_lock(max_workers=2):
    lock = Lock()
    with ThreadPoolExecutor(max_workers=max_workers) as executor:
        for arg in args:
            executor.submit(sleeper, lock, *arg)

f_threading_lock()
```

```
[19:25:05] I am ' First  '  and I will sleep for 3 seconds
[19:25:08] I am ' First    and my sleeping time was so good!
```

```
[19:25:08] I am ' First  ', and I modified var[0]:  it was 10, it is now 11!
[19:25:08] I am ' Third  '  and I will sleep for 4 seconds
[19:25:12] I am ' Third    and my sleeping time was so good!
[19:25:12] I am ' Third  ', and I modified var[0]:  it was 11, it is now 12!
[19:25:12] I am ' Second '  and I will sleep for 2 seconds
[19:25:14] I am ' Second   and my sleeping time was so good!
[19:25:14] I am ' Second ', and I modified var[0]:  it was 12, it is now 13!
```

This is working! By asking the acquisition of `lock`, everything that is inside the `with` bloc of `sleeper` **must be ran in one go**.

**Notes**

- In this example, the protection using `Lock` is a particular extreme case because all the instructions of `sleeper` are protected. Thus execution is sequential.

- Only mutable variables must be protected.

  Similarly, whenever some data is written to the disk, the writing process must be protected too in order to prevent data corruption.

## 17.3 Multiprocessing [advanced]

### 17.3.1 Case study definition

Let's focus on the following function, defined for any complex number $z$:

$$f_c : z \to \sqrt{z^2 + c}$$

With $c$ a complex number.

[1]:
```
import numpy as np

def f(z, c):
    return np.sqrt(z**2 + c)
```

This function can be called recursively $n$ times:

$$g_{c,n} : z \to f_c^n(z) = f_c(f_c(f_c(\ldots)))$$

[2]:
```
def g(z, c, n):
    for _ in range(n):
        z = f(z, c)
    return z
```

$z$ is a complex number: a limited section of the complex plan is needed for evaluation of $g_{c,n}$.

[3]:
```
x, y = np.meshgrid(np.linspace(-1, 1, 2000), np.linspace(-1, 1, 2000))
mesh = x + 1j * y   # imaginary part is defined using `j`
```

And here is a function that defines some values for `c`:

[7]:
```
from numpy.random import uniform
def create_c_values(k):
    c_values = uniform(-1, 1, k) + 1j * uniform(-1, 1, k)
    return c_values
```

### 17.3.2 Sequential version

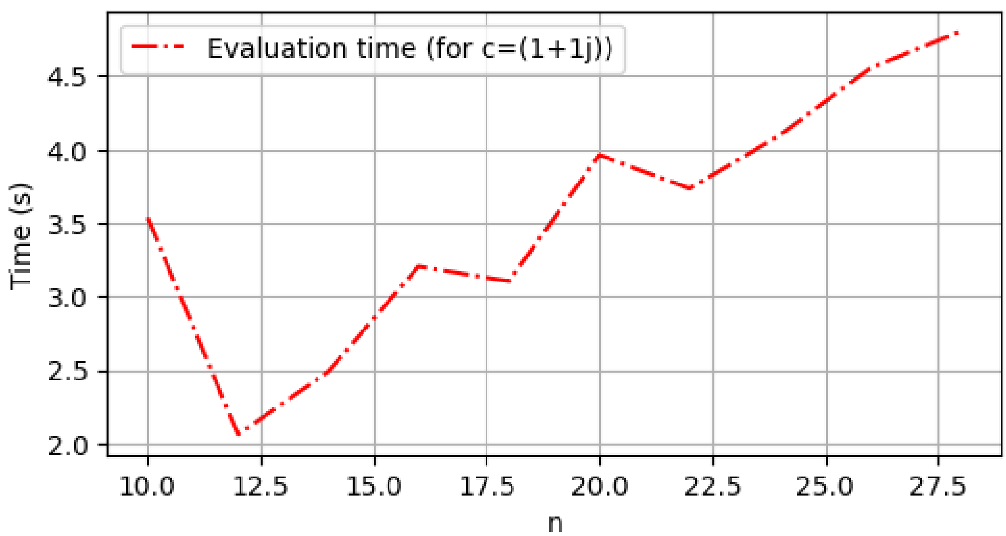
The sequential version of a function that would evaluate `g` given some `c` values is presented here after:

[5]:
```
def evaluate_sequential(mesh, c_values, n=50):
    for c in c_values:
        g(mesh, c, n)
```

### Some benchmarks

**With respect to n** Let's observe the dependance of the total running time in n (number of evaluations of f):

[6]:
```
import matplotlib.pyplot as plt

n = range(10, 30, 2)
running_times = []
c = 1 + 1j
for k in n:
    timeit = %timeit -q -o g(mesh, c, k)
    running_times.append(timeit.average)

fig, ax = plt.subplots(figsize=(6, 3))
ax.plot(n, running_times, 'r-.', label=f'Evaluation time (for c={c})')
_ = ax.set_xlabel('n'), ax.set_ylabel('Time (s)'), ax.legend(), ax.grid()
```

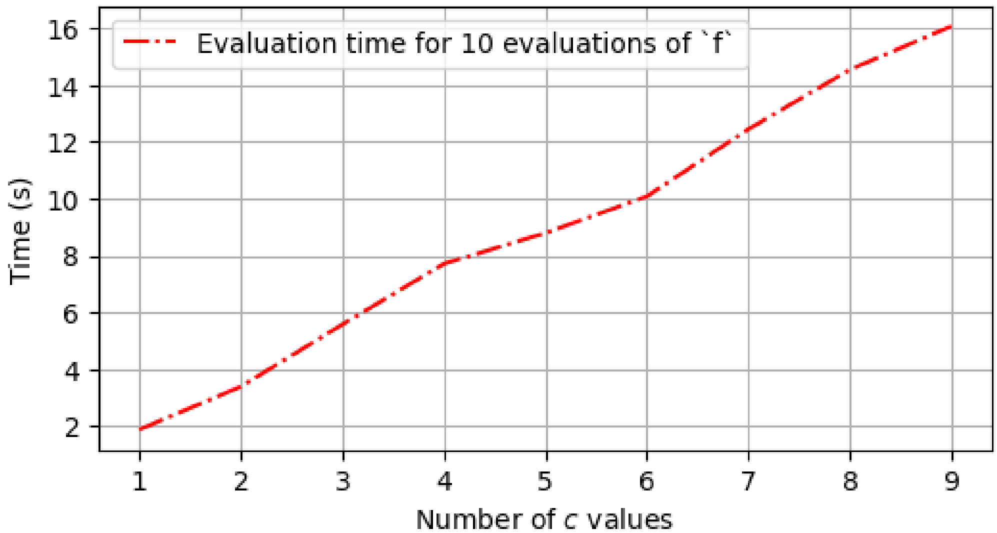



The running time grows **linearly** with n.

**With respect to the number of c values**

[7]:
```
import matplotlib.pyplot as plt

running_times = []
n = 10
nbr_c_values = range(1, 10)
for k in nbr_c_values:
    c_values = create_c_values(k)
    timeit = %timeit -q -o evaluate_sequential(mesh, c_values, n=n)
```

```
    running_times.append(timeit.average)


fig, ax = plt.subplots(figsize=(6, 3))
ax.plot(nbr_c_values, running_times, 'r-.', label=f'Evaluation time for {n}␣
↪evaluations of `f`')
_ = ax.set_xlabel('Number of $c$ values'), ax.set_ylabel('Time (s)'), ax.
↪legend(), ax.grid()
```




same: solving twice more problems requires twice more time.

#### Typical problem

Let's assume we need to run $g_{c,n}$ for 16 different $c$ values and $n = 30$.

[8]:
```
c_values = create_c_values(16)
n = 30
```

How long does it take in a sequential mode?

[9]:
```
%timeit evaluate_sequential(mesh, c_values, n=n)
```

```
1min 22s ± 4.92 s per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

### 17.3.3 Multiprocessing version

#### Code

Similarly to the threading case, `ProcessPoolExecutor` cab be used to manage multiprocessing easily.

The problem is split with respect to $c$ values: elements from the `c_values` variable are dealt with in parallel, with at most `max_workers` simultaneous running processes. This is possible since **no $c$ value is shared between 2 calls of `g`**.

[9]:
```
from concurrent.futures import ProcessPoolExecutor

def evaluate_multiprocessing(c_values, n, max_workers=2):
    with ProcessPoolExecutor(max_workers=max_workers) as executor:
        for c in c_values:
            executor.submit(g, mesh, c, n)
```

Let's time the execution:

[10]:
```
%timeit evaluate_multiprocessing(c_values, n)
```

```
51.1 s ± 2.45 s per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

**Notes**

- Using **two simultaneous processes makes the resolution faster** than in the sequential mode
- **Running time is not divided by 2** because constant overhead times associated with the use of `ProcessPoolExecutor` are very large