Scientific Python

Boris Nerot [1]
LabOratoire proCédés énergIe bâtimEnt

February, 2024
v0.1

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

Contents

I General programming knowledge **7**

1 Computer basics **9**
- 1.1 Hardware . . . 9
  - 1.1.1 Main componenents . . . 9
  - 1.1.2 Typical configurations . . . 11
  - 1.1.3 Howto: compare two computers . . . 11
- 1.2 Software . . . 12
  - 1.2.1 Operating systems (OS) . . . 12
  - 1.2.2 Processes . . . 13
  - 1.2.3 Programing language . . . 13

2 Development basics **15**
- 2.1 Variables types . . . 15
  - 2.1.1 Primitive variable types . . . 15
  - 2.1.2 Containers . . . 15
  - 2.1.3 Object oriented programming . . . 16
- 2.2 Variable scope . . . 17
  - 2.2.1 Definition . . . 17
  - 2.2.2 Static and dynamic typing . . . 17
- 2.3 Mutability . . . 17
  - 2.3.1 Copy and assignment . . . 17
  - 2.3.2 Equality and identity . . . 17

3 Development steps **19**
- 3.1 Identify your problem . . . 19
  - 3.1.1 Define the scientific goal . . . 19
  - 3.1.2 Choose the language . . . 19
  - 3.1.3 Define a development plan . . . 20
- 3.2 Develop . . . 21
  - 3.2.1 Choose an IDE . . . 21
  - 3.2.2 Organize your working space . . . 22
  - 3.2.3 Universal development rules . . . 22
- 3.3 Comment and documentation . . . 24
  - 3.3.1 Differences . . . 24
  - 3.3.2 Comment a code . . . 24
  - 3.3.3 Document a code . . . 24

3.4 Example . . . 27
3.4.1 Identify one's problem . . . 27
3.4.2 Develop . . . 28

II Best of Python **33**

4 Notebooks **35**
4.1 Jupyter code . . . 35
4.1.1 Presentation . . . 35
4.1.2 Howto . . . 35
4.1.3 Configuration . . . 36
4.2 Jupyter markdown . . . 38
4.2.1 Introduction . . . 38
4.2.2 Main functionalities . . . 38
4.2.3 Make the best of markdown . . . 40
4.2.4 See also . . . 40

5 Python basics **41**
5.1 Control flow . . . 41
5.1.1 `while` loop . . . 41
5.1.2 `for` loop . . . 42
5.1.3 `if` conditions . . . 45
5.2 Print formatting . . . 46
5.2.1 Introduction . . . 46
5.2.2 Method 1: no formatting . . . 46
5.2.3 Method 2: advanced formatting . . . 46
5.3 Variable types . . . 49
5.3.1 `list` . . . 49
5.3.2 `set` . . . 52
5.3.3 `tuple` . . . 53
5.3.4 `dict` . . . 54
5.3.5 `deque` . . . 56
5.3.6 About **slicing** . . . 56
5.3.7 Booleans . . . 57
5.4 List comprehensions . . . 58
5.4.1 Introduction . . . 58
5.4.2 Exemples simples . . . 58
5.4.3 Autres types de données . . . 59
5.5 Function signature . . . 61
5.5.1 `unpacking` . . . 61
5.5.2 Function signature . . . 62
5.6 OS interactions . . . 67
5.6.1 File reading/writing . . . 67
5.6.2 Files management . . . 69
5.6.3 Run a system call . . . 72
5.7 Decorators . . . 74
5.7.1 Introduction . . . 74

5.7.2 Simple example . . . 74
5.7.3 Fictive use case . . . 75
5.7.4 Conclusion . . . 77
5.8 Date/time . . . 78
5.8.1 Introduction . . . 78
5.8.2 Timestamp . . . 78
5.8.3 Time periods: `timedelta` . . . 81

6 Handling exceptions **83**
6.1 Exceptions . . . 83
6.1.1 Introduction . . . 83
6.1.2 Handle an exception . . . 84
6.1.3 Raise an exception . . . 86
6.1.4 Conclusion . . . 87
6.2 Debugger . . . 88
6.2.1 Introduction . . . 88
6.2.2 Debugging using VSCode . . . 88
6.3 Logging . . . 93
6.3.1 About this notebook . . . 93
6.3.2 Introduction . . . 93
6.3.3 Exemple . . . 93

III Share one's code **97**

7 Environments **99**
7.1 Introduction . . . 99
7.2 Anaconda . . . 99
7.2.1 Introduction . . . 99
7.2.2 Create an environment . . . 99
7.2.3 Activate an environment . . . 100
7.2.4 Important notes . . . 101

8 Structuration **103**
8.1 Imports . . . 104
8.1.1 Introduction . . . 104
8.1.2 Full import . . . 104
8.1.3 Relative import . . . 105
8.1.4 Good practices . . . 105
8.2 Package structuration . . . 107
8.2.1 Introduction . . . 107
8.2.2 Example . . . 107
8.2.3 Search for packages . . . 109
8.2.4 Add some content to __init__ . . . 110
8.2.5 Take away . . . 112

9 Documentation **113**
9.1 Documentation writing . . . 114
9.1.1 Introduction . . . 114

9.1.2 Setting up the needed tools . . . 114
9.1.3 Create a docstring . . . 115
9.1.4 Add some content to a docstring . . . 116
9.2 Documentation generation . . . 119
9.2.1 Introduction . . . 119
9.2.2 Overview . . . 119
9.2.3 Details . . . 119
9.2.4 Result . . . 122
9.2.5 Notes . . . 123

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

Ventirad de CPU

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 2GHz	16-128 logical cores 2-4 GHz
Disk	SSD 500 GB	HDD 10 TB 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. Un système d'exploitation est un logiciel qui rend disponible les ressources matérielles de l'ordinateur à travers des interfaces (graphiques ou non).
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:

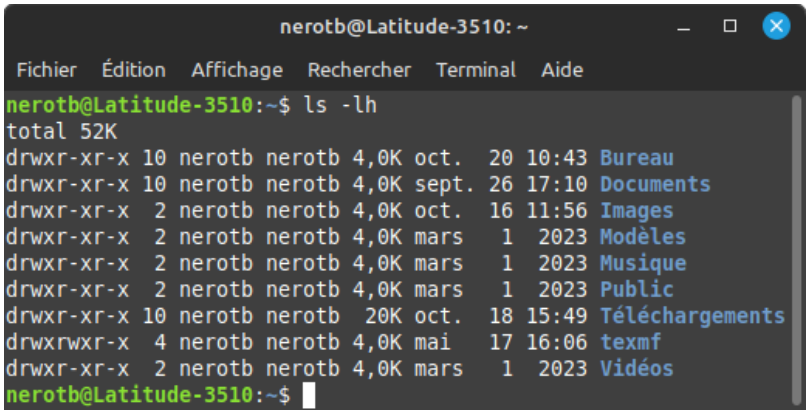
- Windows (Microsoft)
- OS X (Apple)
- Linux

GUI and CLI

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

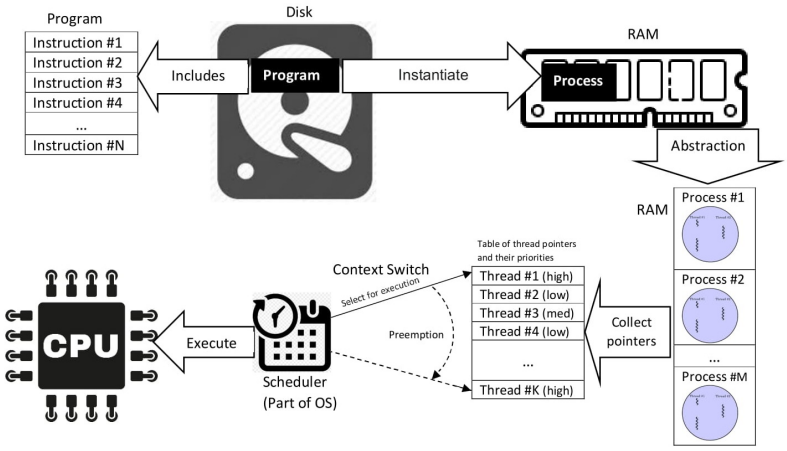
CLI : *Command Line Interface*
Interface that describes software components using text. One can interact with a CLI using a keyboard.

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

Compilated vs interpretated

Compilated 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

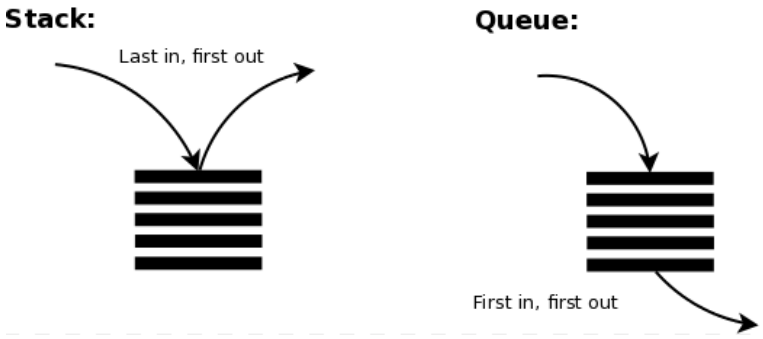
Primitive variable types are types already implemented by the language itself. Most common are:

- 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:

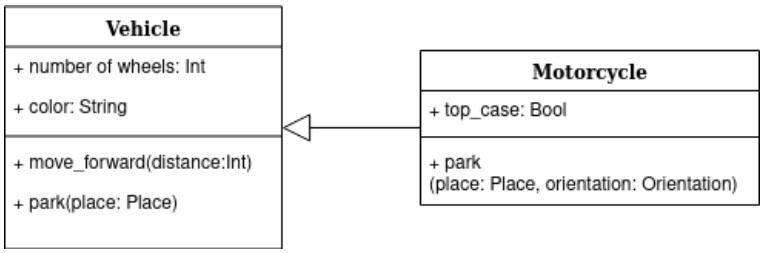
- Unique identifier: *Int*
- Name: *String*
- Height: *Float16*
- Parent(s) and siblings: containers of objects of type *Official identity*

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:

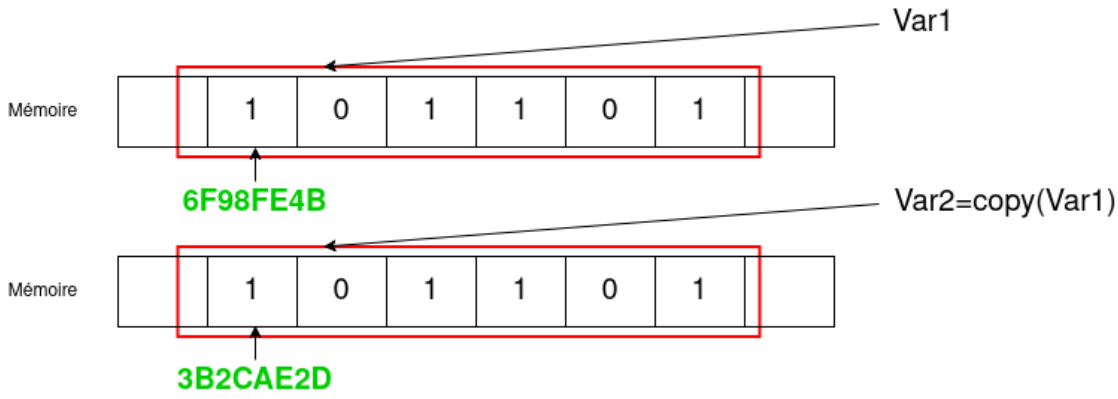
- The in-depth copy of an object makes a full duplicate of this object. (Figure 2.3).
- Conversely, an assignment of a mutable object to a variable makes the variable reference the memory content of the object (Figure 2.4).

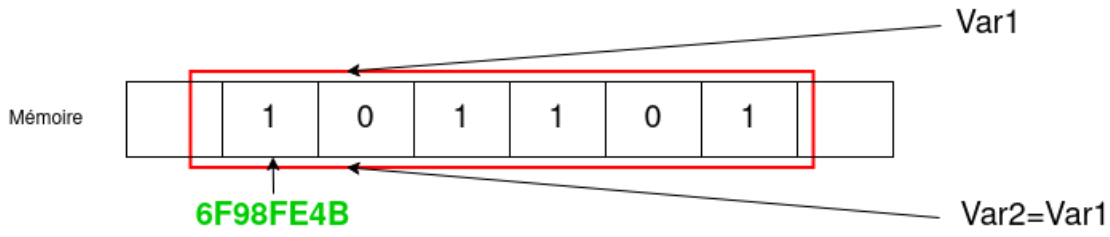
In the first case, a modification on the first variable has no effect on the second. In the second case, if first is modified then second is modified too.
There exists an intermediate copy level between assignment and in-depth copy: shallow copy. If 'b' is a shallow copy of 'a', 'b' has a different memory 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 can be due to 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. Main content of the code. Simplified presentation of results.	Using a test case: Are the results correct? Does the running time seem compatible with all the cases to be treated?
2. Structure the code	Possible interactions with the code. Aspect of results.	Definition of functions/classes/structures Errors handling, progress log.	Resilience to parameters change, easy results exploitation.
3. Optimize the code	I/O importance in the problem, memory independance.	Benchmark 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 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: 1. Important content 2. Optionnal content (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.

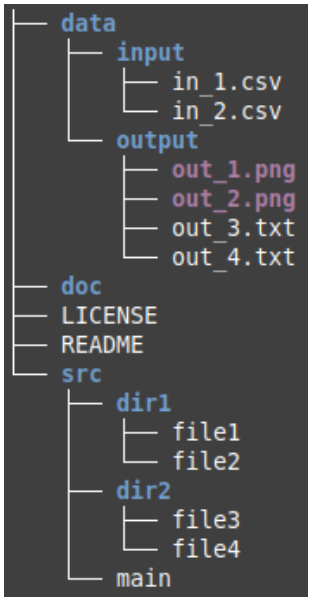
3.2.2 Organize your working space

A working space is made of:

- Source files and compilated 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.



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 ($\leq$ 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 languages
```

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.206.1815
   ```

   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

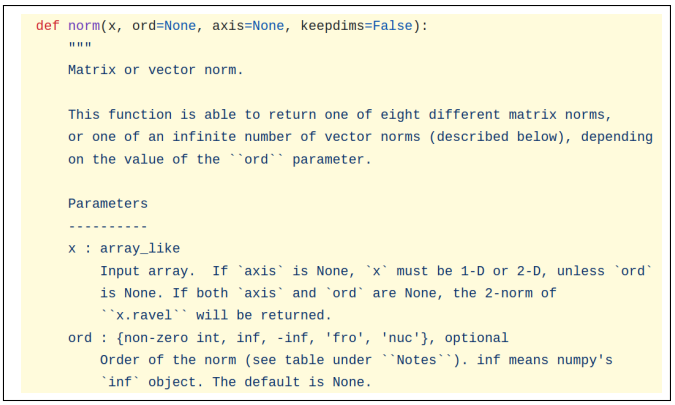
A particular syntax is adoptéed to reference a component of the code (variable, function, ...) or the language (types, ...).

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

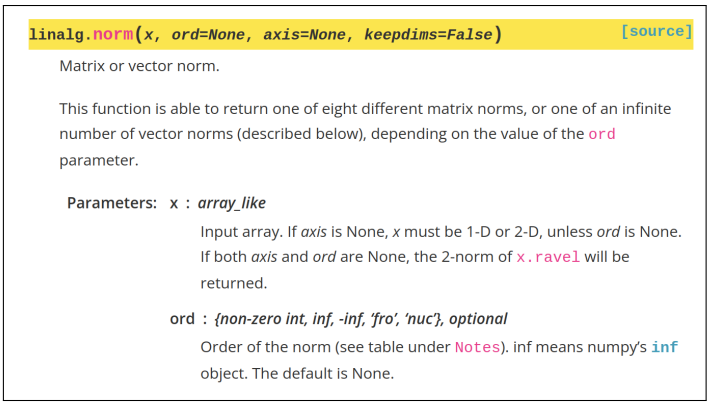
- HTML: if documentation is hosted on the web `https://about.readthedocs.com/`
- Latex/PDF: if documentation is shared locally

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)



```
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:

- θ 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 θ_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 Define and solve the equation. Perform aquick draft plot of the solution.	1 hour
2. Structure the code	Build an API. Display some progress information for the user. Customizz and save the results plot.	1 hour
3. Optimize the code	Profile and analyse the running time. Evaluate potential time savings. 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. Build as html.	30 min

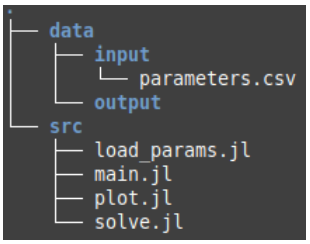
3.4.2 Develop

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:

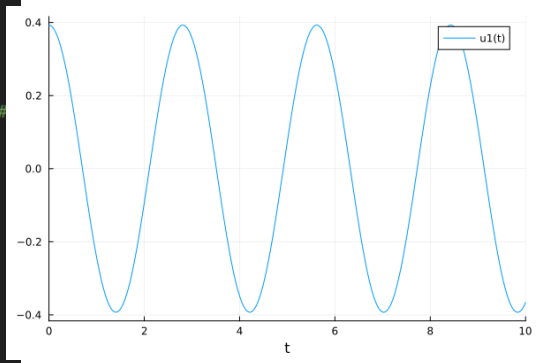
$$u = \begin{bmatrix} \theta \\ \omega \end{bmatrix}$$

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

src > main.jl > ...

```julia
#= 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



θ 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 > ...

```julia
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 > ...

```julia
#= 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 > ...

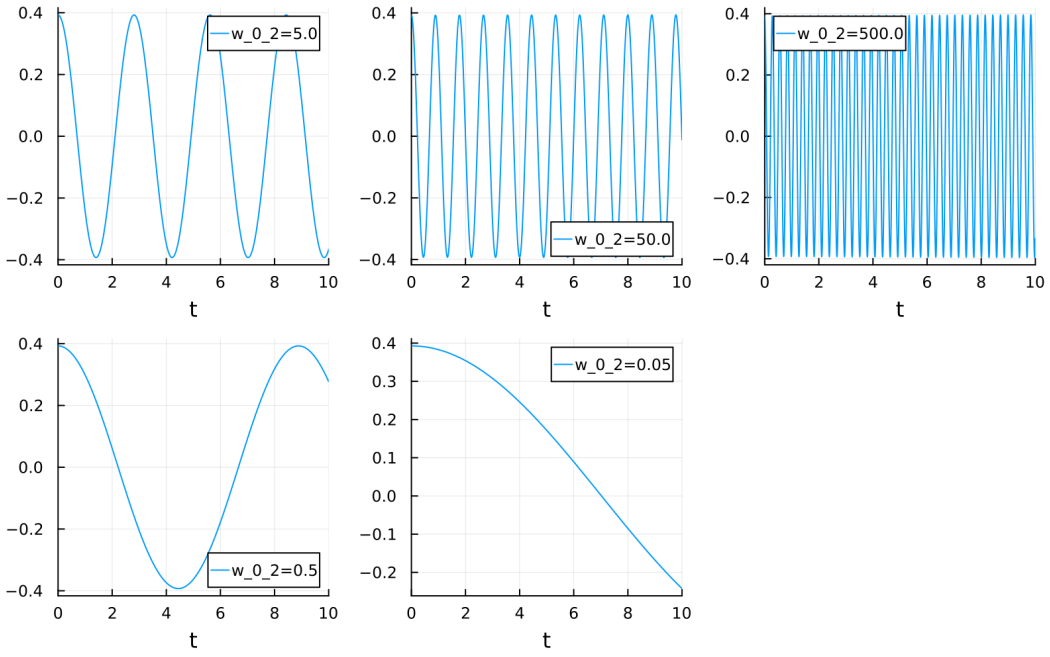
```julia
#= solving equation
helper doc: https://docs.sciml.ai/DiffEqDocs/
            stable/types/ode_types/ =#

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 > ...

```julia
#= 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.

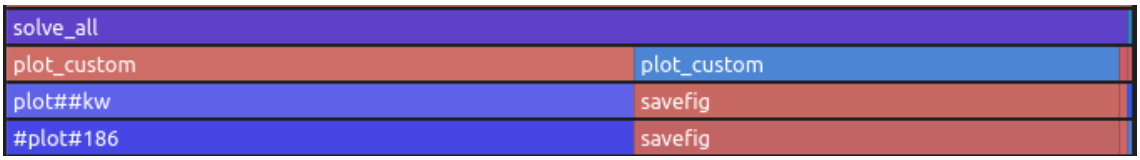
```
BenchmarkTools.Trial: 15 samples with 1 evaluation.
 Range (min … max):  315.696 ms … 405.419 ms  | GC (min … max): 0.00% … 0.00%
 Time  (median):     352.485 ms               | GC (median):    0.00%
 Time  (mean ± σ):   355.361 ms ±  29.252 ms  | GC (mean ± σ):  0.00% ± 0.00%

  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?**

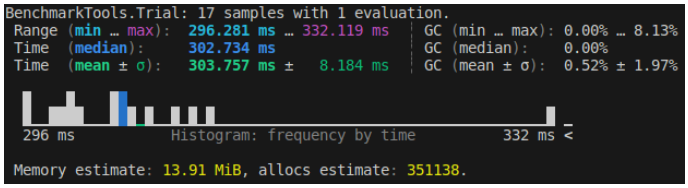
2. The standard deviation of running time is low.

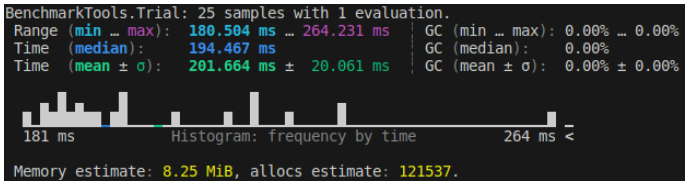
Profile the running time
The macro *@profile* makes it possible to focus on time consuming sections of the code. Figure 3.4 shows a *FlameGraph* of this analysis.



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

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

[5]:
```
area = 100
speed = 30
print(area, speed)
```

```
100 30
```

[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).

Functionalities

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

Some of them are available using *plugins*.

4.2 Jupyter markdown

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: $a_{3,4} = \sum_j b_3^j \times c_4^j$

- Block mode:

$$a_{3,4} = \sum_j b_3^j \times c_4^j$$

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:

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

"

[4]:
```
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)
```

[4]:
```
8.75
```

One can easily convert a notebook into a Latex or PDF file. This is very hand 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

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

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 `any` returns `True` then `all` 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
```

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

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.

[10]:
```
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):

[12]:
```
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
j=1, k=0, k=1, k=2, k=3
j=2, k=0, k=1, k=2, k=3
j=3, k=0, k=1, k=2, k=3
j=4, k=0, k=1, k=2, k=3
```

`continue`

Interrupt current iteration and go the next one.

[13]:
```
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]:
```
for k in range(5):
    if k > 15:
        break
```

```
else:
    print("Values lower than 15")
```

```
Values lower than 15
```

5.1.3 `if` conditions

Keyworks 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

Keyworks 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 is ran when the nearest `if` (and following `elseif` if any) did not hold `True`.

[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

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

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

[5]:
```
data = [1, 2, 3, 4]
print(*data)
print(*data, end="/")
```

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

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.

[6]:
```
var = 35.123456
sentence = f"Value of 'var' is {var}"
print(sentence)
```

```
Value of 'var' is 35.123456
```

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

[7]:
```
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____
____35.12
__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

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 `+`.

[14]:
```
print(var1)
print(var2)
print(var1 + var2)
```

```
Custom (10)
Custom (9)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[14], line 3
      1 print(var1)
      2 print(var2)
----> 3 print(var1 + var2)

TypeError: unsupported operand type(s) for +: 'Custom' and 'Custom'
```

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

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

Reminder, `tuple`s are immutable:

[19]:
```
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]:
```
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.

[32]:
```
var2 = var[:5]
print(f"var: {var}\t\t var2: {var2}")
var2[1] = 1000                          # modification
print(f"var: {var}\t\t var2: {var2}")
```

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

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

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 Exemples simples

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 Autres types de données

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

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 forget about some 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]:
```
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]:
```
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]:
```
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]:
```
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`.

[13]:
```
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.

[14]:
```
def f6(a, b, *other_values):
    print(a, b, other_values)
```

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

[15]:
```
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)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[15], line 5
      2     print(a, b, args)
      4 f7(1, 3, 4, 5, b=2)
----> 5 f7(1, 3, 4, 5, 2)

TypeError: f7() missing 1 required keyword-only argument: 'b'
```

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).

[16]:
```
def f8(a, b, **kwargs):
    print(a, "  ", b, end="   ")
    print(kwargs.get("c", 0), end="   ")    # if no 'c' exists as a dict key,␣
↪take 0
    print(kwargs.get("d", 0))


f8(1, 2)
f8(1, 2, c=3, d=5)
f8(1, 2, d=5)
```

```
1   2   0   0
1   2   3   5
1   2   0   5
```

One can also perform unpacking of dict-like variables.

In the example below, unpacking is used to deconstruct the dictionary into a group of named arguments, which are once again interprated by the function:

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

[17]:
```
def f9(a, b, **kwargs):
    print(a, "  ", b, end="   ")
    print(kwargs.get("c", 0), end="   ")
    print(kwargs.get("d", 0))

f9(1, 2)
f9(1, **{"b": 9, "c": 3, "d": 5})
f9(1, 2, **{"d": 5})
```

```
1   2   0   0
1   9   3   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

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 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
result = subprocess.run(args="mkdir a_new_dir",
                        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

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

[33]:
```
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.

[34]:
```
@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.

[35]:
```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:

[36]:
```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. Hence, one could use an decorator:

[37]:
```
def only_positive(func):
    def new_func(*args, **kwargs):
        result = func(*args, **kwargs)
        return max(result, 0)
    return new_func

@only_positive
def sum_(var):
    return sum(var)

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

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

[38]:
```
var1 = [2, 3, 5]
var2 = [-3, 5, 3]
print(sum_(var), prod_(var), pow_(var))
print(sum_(var2), prod_(var2), pow_(var2))
```

```
9 24 4096
5 0 0
```

Advanced

This is handy, but the functions are still modified a bit and we wave 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.

[39]:
```python
def set_only_positive(positive):
    if positive:
        return only_positive
    else:
        def no_decorator(func):
            return func
        return no_decorator

@set_only_positive(True)
def sum_(var):
    return sum(var)

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

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

[40]:
```python
var1 = [2, 3, 5]
var2 = [-3, 5, 3]
print(sum_(var), prod_(var), pow_(var))
print(sum_(var2), prod_(var2), pow_(var2))
```

```
9 24 4096
5 0 -14348907
```

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.

5.8 Date/time

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`).

[12]:
```
from datetime import datetime

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

```
10:25:24:498134 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

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`), ...

[3]:
```
s = "a string"
s / 5
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[3], line 2
      1 s = "a string"
----> 2 s / 5

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

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

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
Cell In[4], 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.

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

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

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

f3(1, 0)
print("Not executed")
```

```
---------------------------------------------------------------------------
ZeroDivisionError                         Traceback (most recent call last)
Cell In[5], line 10
      7 def f3(a, b):
      8     f2(a, b)
---> 10 f3(1, 0)
     11 print("Not executed")

Cell In[5], line 8, in f3(a, b)
      7 def f3(a, b):
----> 8     f2(a, b)

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

Cell In[5], 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:

[6]:
```
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).

[7]:
```
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)

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`.

[7]:
```
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 10:  Sensor default, got negative value -1.
Time step 15:  Sensor default, got negative value -1.
```

```
[7]: [9, 10, 0, 10, 2, 5, 9, 3, 7, 1, 0, 5, 10, 7, 9, 0, 1, 10, 2, 10]
```

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

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

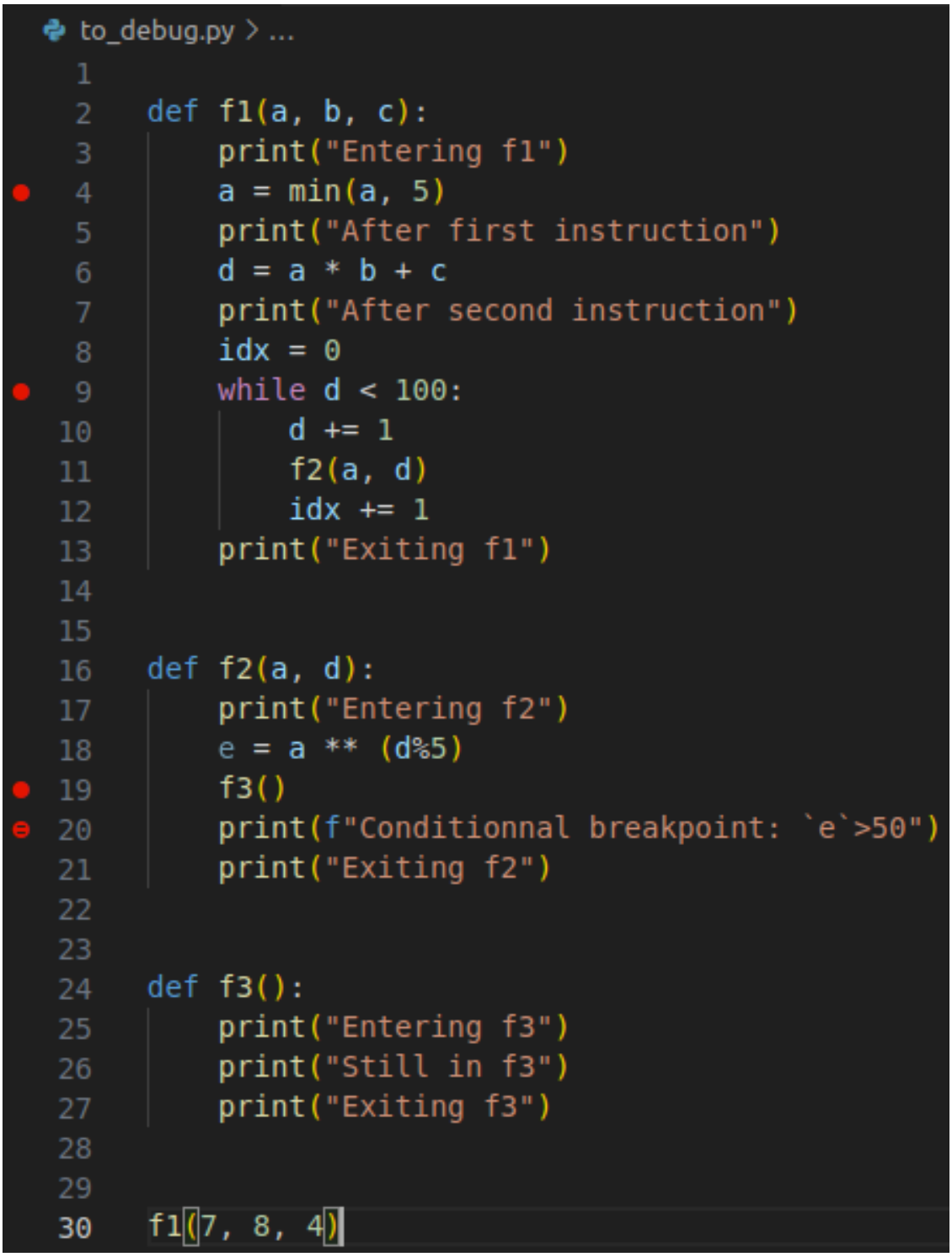
The native debugger of Python is a library called `pdb`. Whenever the `set_trace` method is used to declare breakpoints, execution falls back to debug mode.

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.

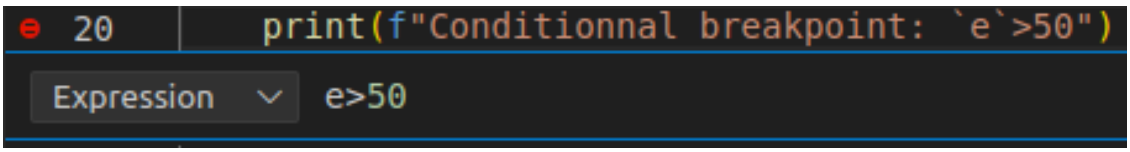
6.2.2 Debugging using VSCode

Set some *breakpoints*

Let's focus on the following code:

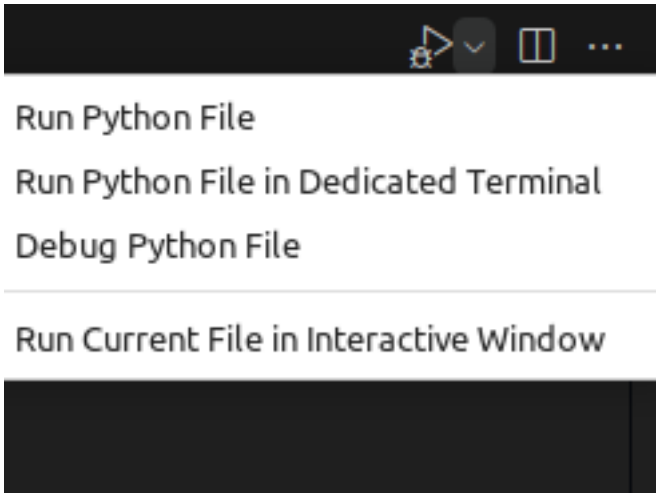


By a left clic in the margin, 4 breakpoints were defined (red dots). At lines 4, 9 and 19 are normal breakpoints. Line 20 is a conditional breakpoint: a breakpoint that is activated is and only if `e>50`.

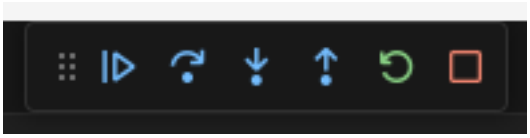


Launch the debugger

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:



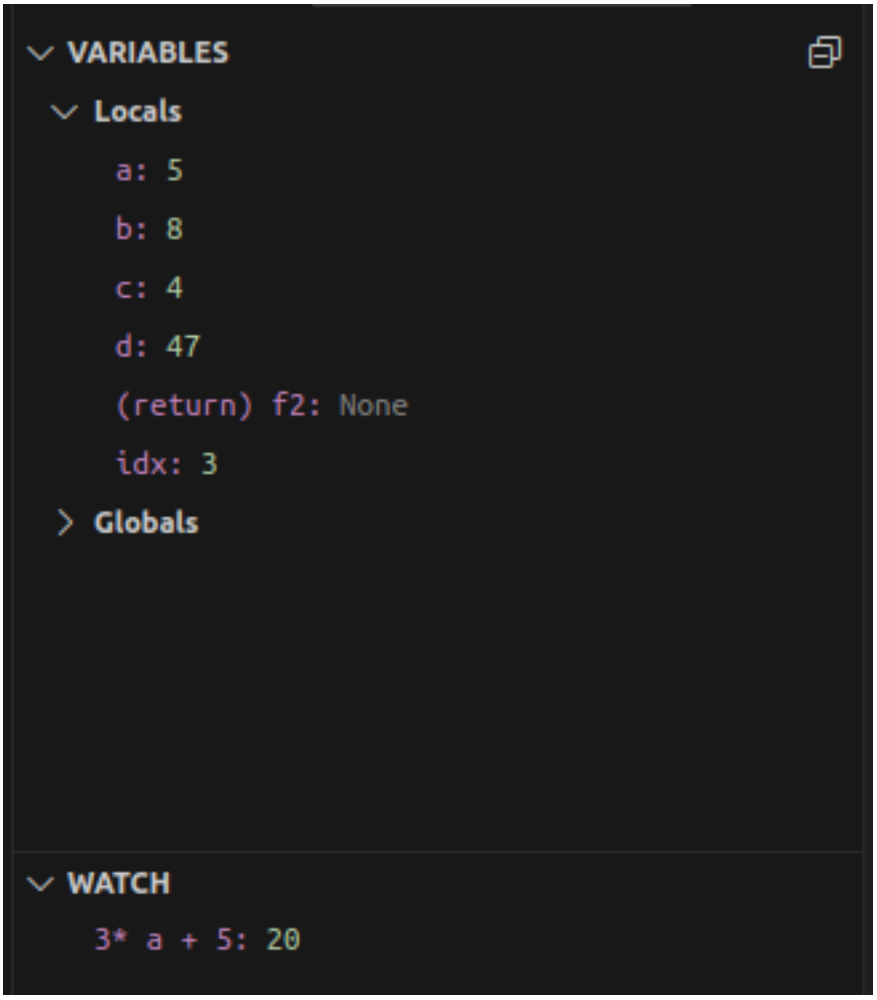
From left to right:

- *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.

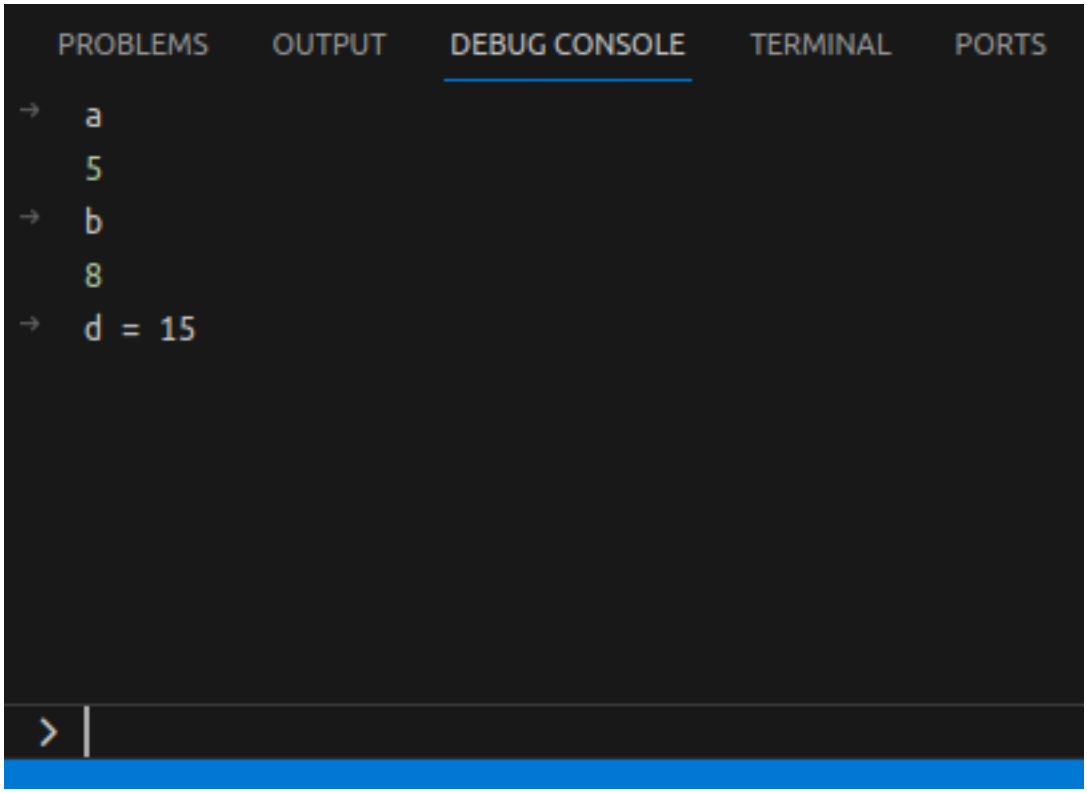
Similar to panel **variables** is panel **watch**: one can define custom Python expressions involving variables. These expressions are also updated as the execution goes on.



A right clic on a variable shows options to:

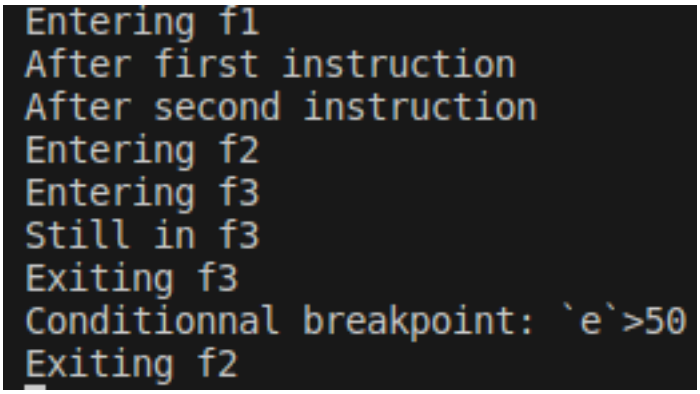
- copy the variable value
- explicitely set this value

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

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

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 availabl 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 Important 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

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 to 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

8.2.1 Introduction

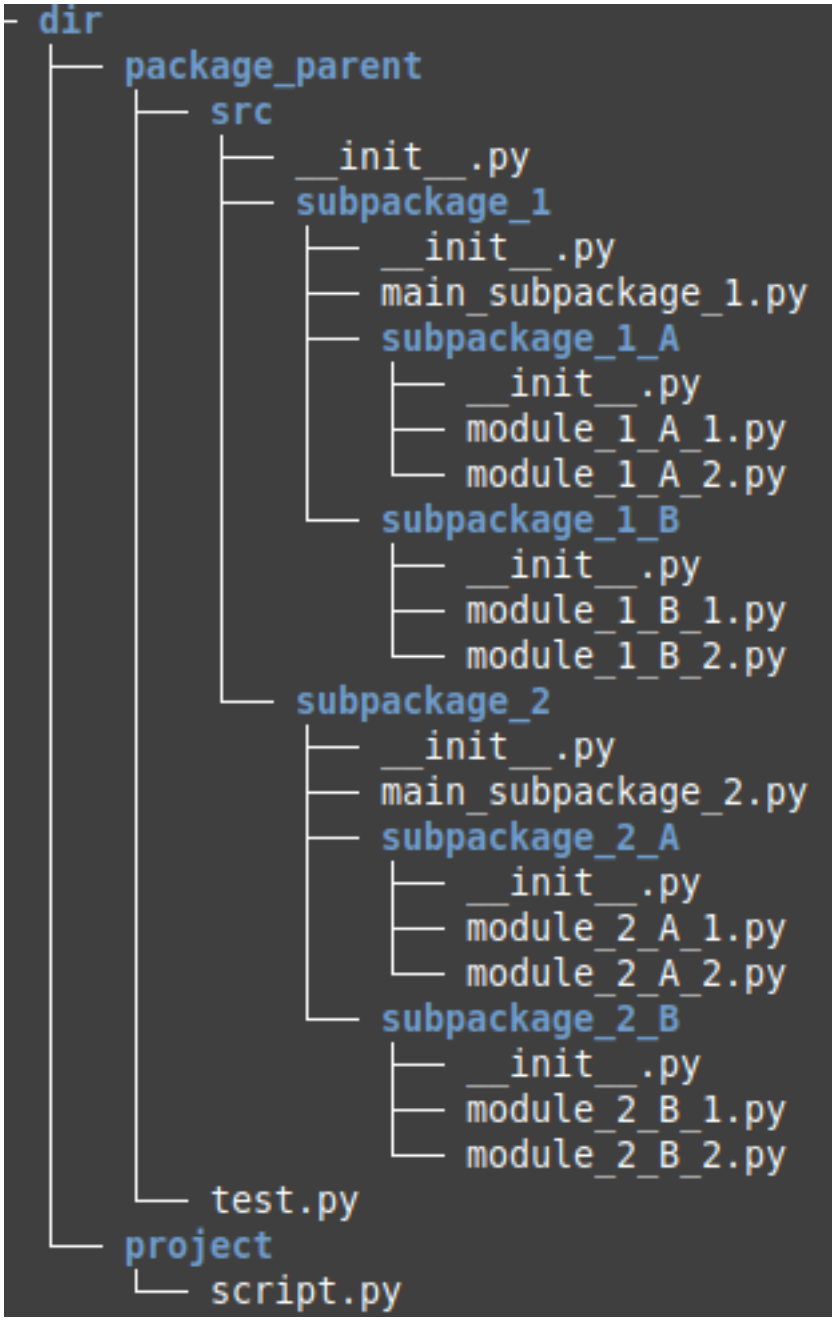
To import some content, Python must know:

- 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

Let's focus on the following files structure:



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'
```

Note: the term used in the exception is `module`, not packaged, but the principle remains similar

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
```

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 syntax `.[module_name]` tells Python to search for a module in the current package.

Yet, 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 pour importer tout le contenu du 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

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

9.1.1 Introduction

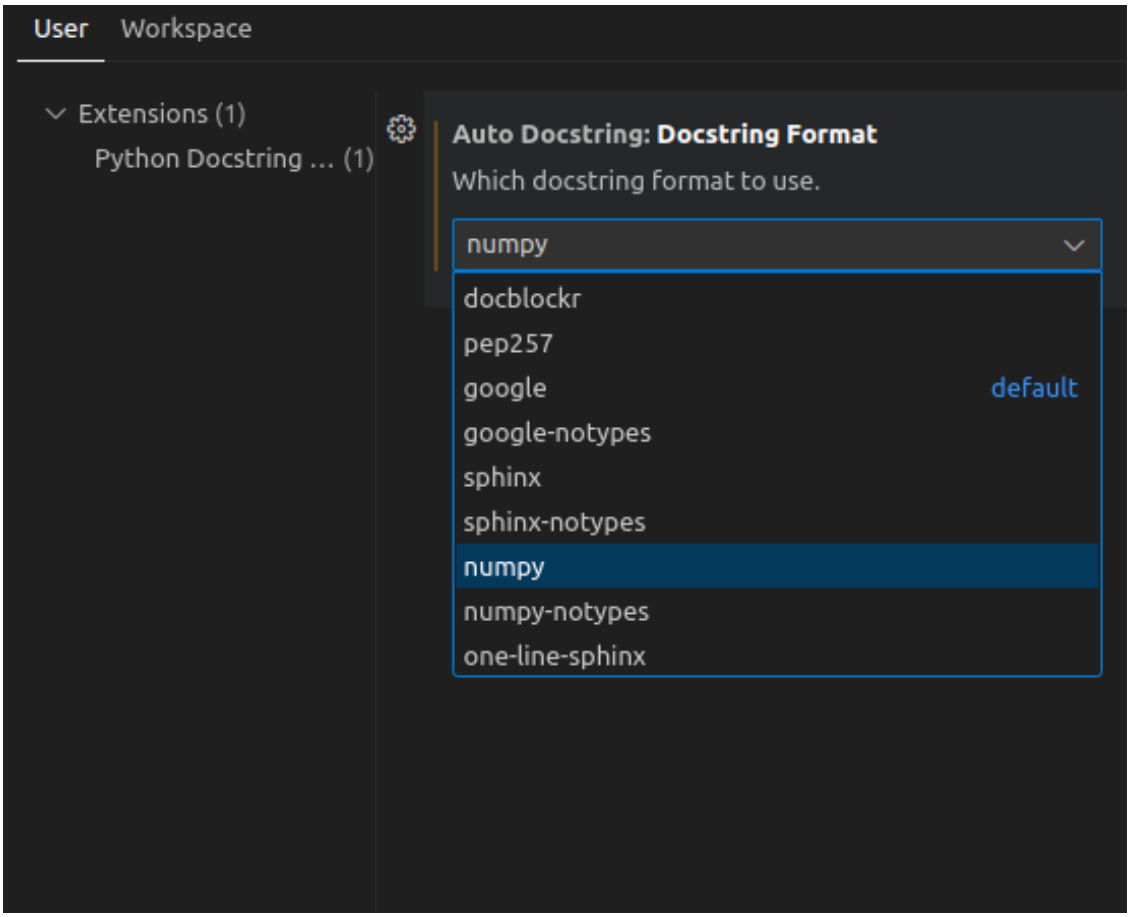
Documenting a code consists in writing ***docstrings***. *Docstrings* are special strings attached to a code section that a have special meaning for the interpreter. They are what is displayed using the `help` function.

In this part, the `src` package introduced previsouly is documented using VSCode.

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:

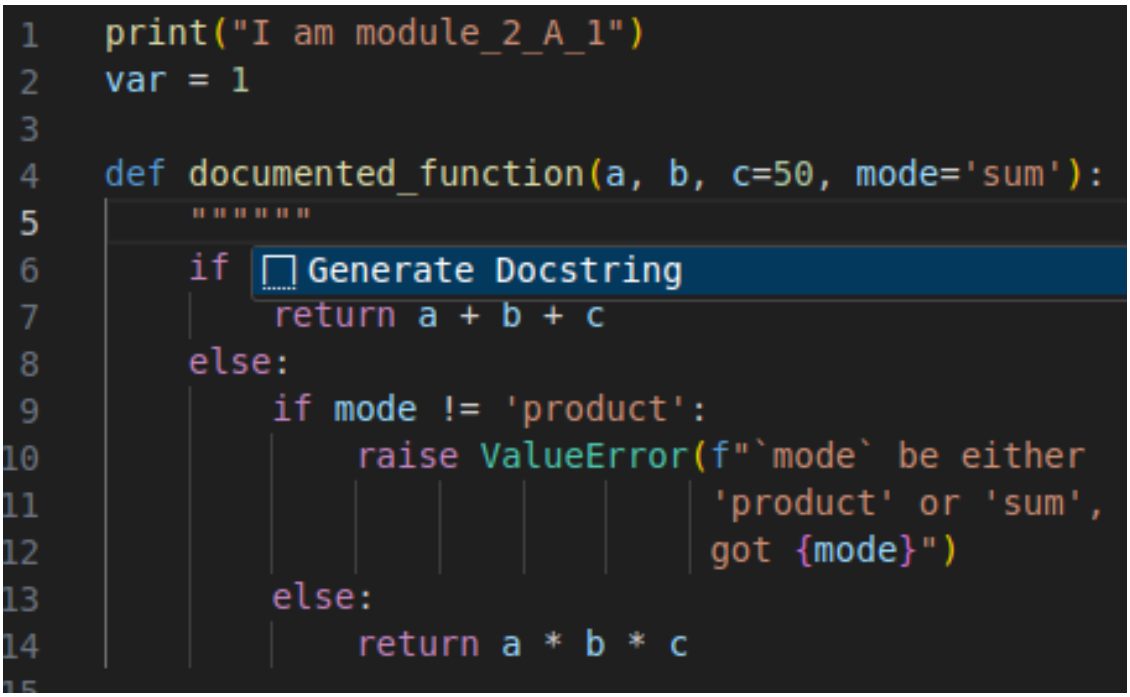
[1]:
```
def abc(a: int, c = [1,2]):
    """_summary_

```

```
    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
print("I am module_2_A_1")
var = 1

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* ‘.

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

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:

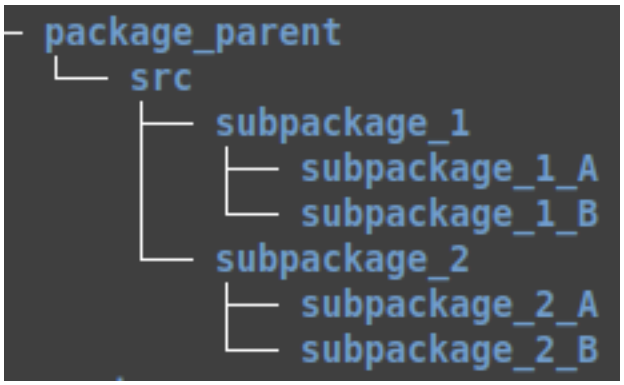
   - html
   - pdf

9.2.3 Details

Setting up the tools

All the doc geeneration proces 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`

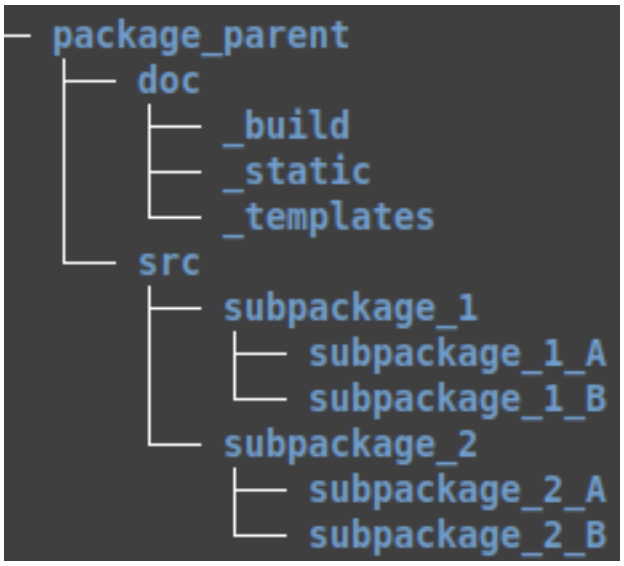
Some files are created in 'doc':

- `conf.py`: contains `sphinx` configuration for the project (step 1 et 3 décrites described in part 'Overview').

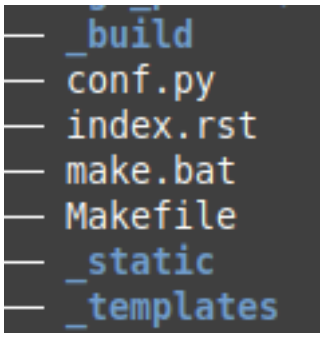
- `index.rst`: structure of the final documentation (step 2).

  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`:

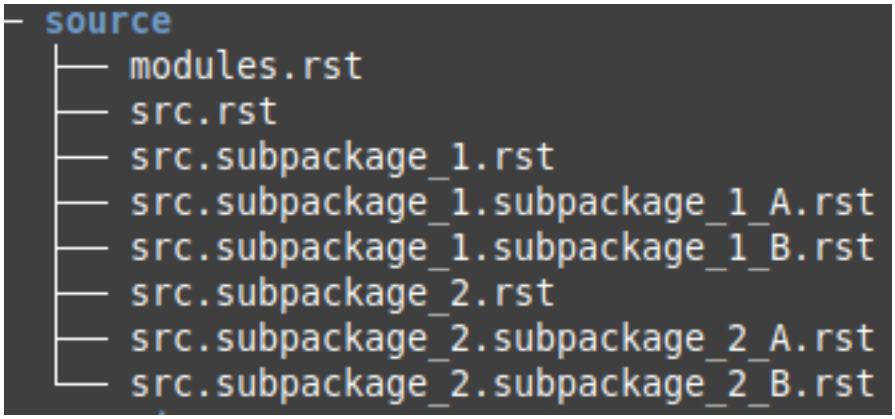
[1]:
```
extensions = ['sphinx.ext.napoleon']
```

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 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:

```
3    You can adapt this file completely to your liking, but it should at least
4    contain the root `toctree` directive.
5
6  Welcome to my_package's documentation!
7  ======================================
8
9  .. toctree::
10    :maxdepth: 2
11    :caption: Contents:
12     Here is a short description for our doc page.
13
14    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:

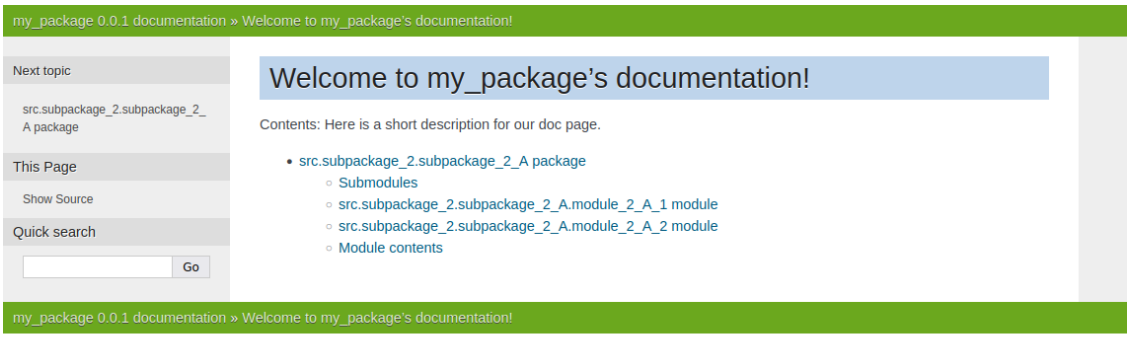
```
from sys import path
path.insert(0,  r'/absolute/path/to/dir/package_parent')
```

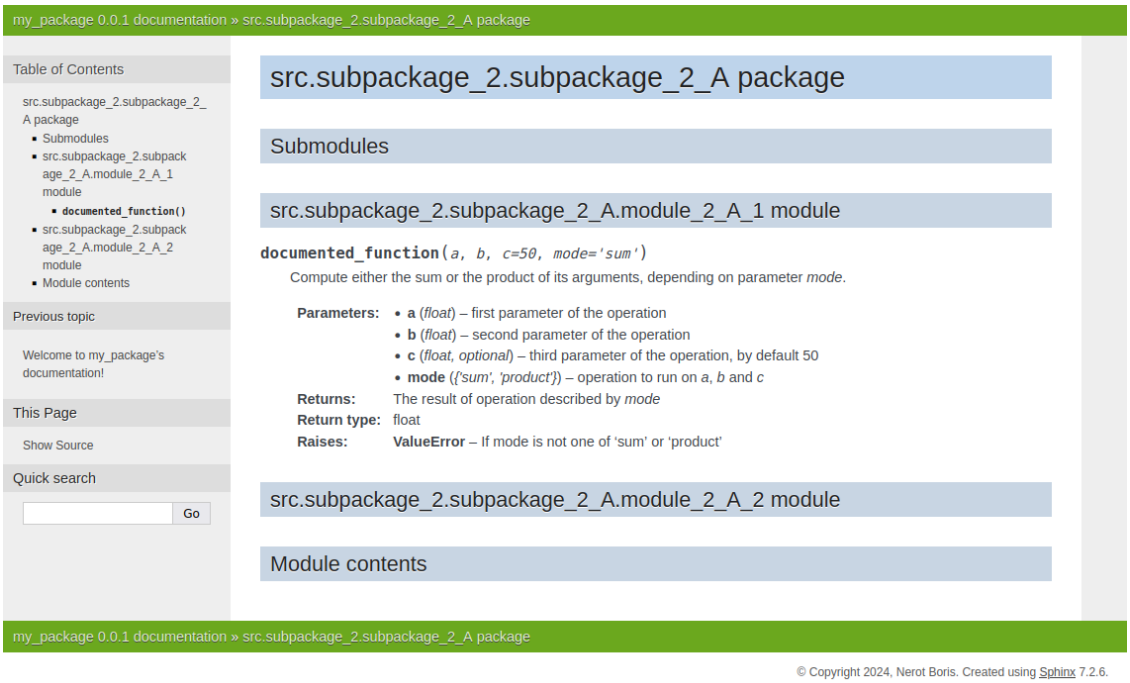
Then, back to commond prompt in the `doc` directory:

```
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.