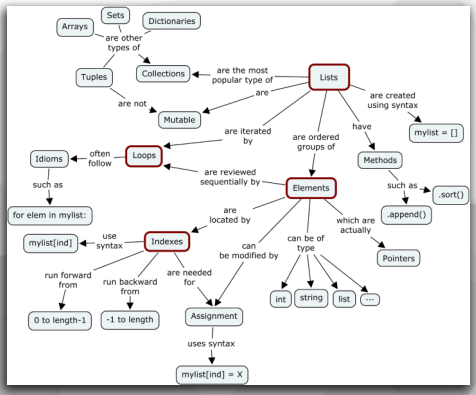
Modelling and Simulation in Physics - Introduction to Python

Hervé Wozniak - Université de Montpellier

Academic year 2024-2025 Release: 2 September 2024



Hervé Wozniak - University of Montpellier

Table of contents

Objectives

Course (UE HAP708P) objectives: numerical methods useful for physics

The whole course aims at providing the basics of numerical methods useful in the specific context of modelling and simulation in Physics. It is therefore not a computer science course.

The course is divided into two parts. The first part consists of learning or revising a programming language. The second part is devoted to the study of classical algorithms in physics.

Objectives of the Part I: allowing a quick grasp of a programming language (Python)

This part of the course focuses on the tool used to program these numerical methods, i.e. a particular language.

Python and a widely used graphical environment are chosen as the programming language and visualization tool.

The objective being to be quickly efficient in numerical analysis lab sessions, only the immediately useful notions will be seen in the course.

- basic manipulations (syntax, variables, loops, tests, functions)
- algebra and analysis (NumPy)
- visualization (Matplotlib)

Complements:

- direct applications (FFT, geometric data structures, etc.) [in progress]
- utilities for the physicist (SciPy)
- files
- program testing
- object-oriented programming [not yet translated]

Introduction

Contact information:

- Building 21, 4th floor, LUPM
- herve.wozniak@umontpellier.fr

Disclaimer:

The translation of the course is in progress. Some comments in the example programs could remain in French.

French resources:

- courspython.com[1] (the course for years prior to 2017, updated in 2022)
- python.developpez.com/cours/apprendre-python3[2] (G.Swinnen "Apprendre à programmer avec Python", free book in PDF)
- python.developpez.com/tutoriels/cours-python-uni-paris7/[3] (P. Fuchs, P. Poulain, "Cours de Python")

English resources:

- https://moodle.umontpellier.fr/course/view.php?id=16822
  - self-registration for those who are not yet registered
  - course + lab sessions only appear when the student is assigned to a group (A, B or C)
- https://github.com/Asabeneh/30-Days-Of-Python (tep-by-step guide to learn the Python programming language in 30 days)
- web.neurotiko.com/Universidad/ACM/SIGPython/Learning%20Python%205th%20Ed%202013.pdf[4] ("Learning Python", Mark Lutz, 5th edition 2013, O'Reilly, in English)
- docs.python.org/tutorial[5]
- scipy-lectures.org/[6] (NumPy and Matplotlib lecture notes)
- hal.inria.fr/hal-03427242[7] (Matplotlib)

Agenda for 2024-2025:

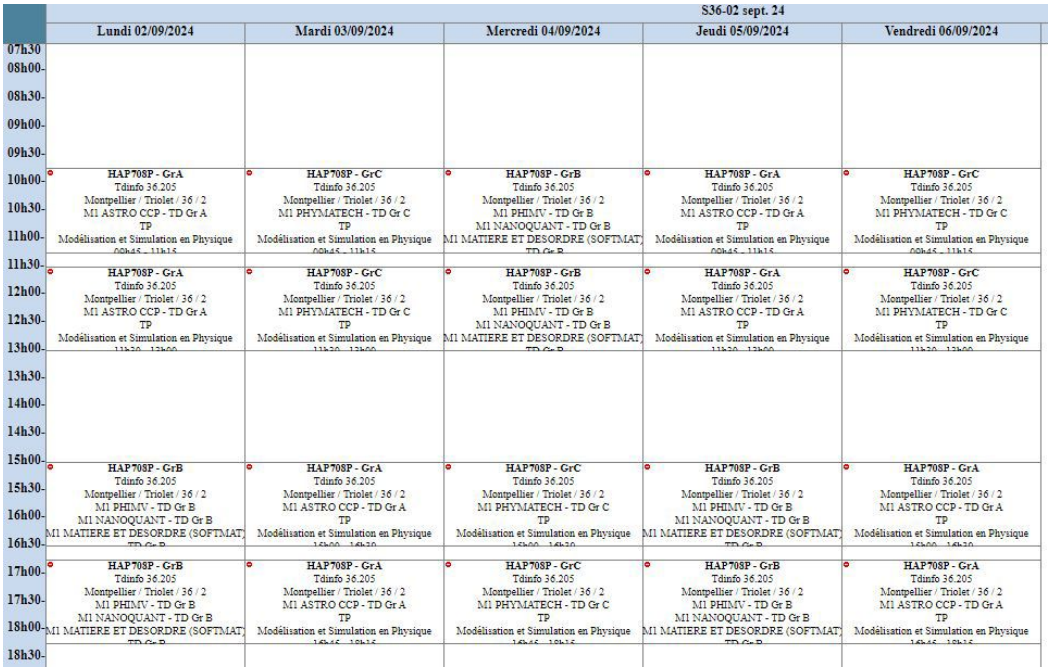
- two intensive weeks of lab work (main way of working, min. 6 sessions per group)

  + lectures (as less as possible as they supplement but do not substitute for the reading)

  + individual or small group projects (optionnal)
- refresher and finalization session (7th) of the lab work September 17th (groups A & B) and 24th (group C)
- CC1: October 1st 9h45 - 12h00 (python)
- CC2: December 9th 9h45-12h00 (numerical analysis)

---

1. http://courspython.com
2. http://python.developpez.com/cours/apprendre-python3
3. http://python.developpez.com/tutoriels/cours-python-uni-paris7/
4. https://web.neurotiko.com/Universidad/ACM/SIGPython/Learning%20Python%205th%20Ed%202013.pdf
5. http://docs.python.org/tutorial
6. http://scipy-lectures.org/
7. https://hal.inria.fr/hal-03427242



S36-02 sept. 24	Lundi 02/09/2024	Mardi 03/09/2024	Mercredi 04/09/2024	Jeudi 05/09/2024	Vendredi 06/09/2024
07h30					
08h00					
08h30					
09h00					
09h30					
10h00	HAP708P - GrA Tdinfo 36.205 Montpellier / Triolet / 36 / 2 M1 ASTRO CCP - TD Gr A TP Modélisation et Simulation en Physique	HAP708P - GrC Tdinfo 36.205 Montpellier / Triolet / 36 / 2 M1 PHYMATECH - TD Gr C TP Modélisation et Simulation en Physique	HAP708P - GrB Tdinfo 36.205 Montpellier / Triolet / 36 / 2 M1 PHIMV - TD Gr B M1 NANOQUANT - TD Gr B M1 MATIERE ET DESORDRE (SOFTMAT)	HAP708P - GrA Tdinfo 36.205 Montpellier / Triolet / 36 / 2 M1 ASTRO CCP - TD Gr A TP Modélisation et Simulation en Physique	HAP708P - GrC Tdinfo 36.205 Montpellier / Triolet / 36 / 2 M1 PHYMATECH - TD Gr C TP Modélisation et Simulation en Physique
10h30					
11h00					
11h30					
12h00	HAP708P - GrA Tdinfo 36.205 Montpellier / Triolet / 36 / 2 M1 ASTRO CCP - TD Gr A TP Modélisation et Simulation en Physique	HAP708P - GrC Tdinfo 36.205 Montpellier / Triolet / 36 / 2 M1 PHYMATECH - TD Gr C TP Modélisation et Simulation en Physique	HAP708P - GrB Tdinfo 36.205 Montpellier / Triolet / 36 / 2 M1 PHIMV - TD Gr B M1 NANOQUANT - TD Gr B M1 MATIERE ET DESORDRE (SOFTMAT)	HAP708P - GrA Tdinfo 36.205 Montpellier / Triolet / 36 / 2 M1 ASTRO CCP - TD Gr A TP Modélisation et Simulation en Physique	HAP708P - GrC Tdinfo 36.205 Montpellier / Triolet / 36 / 2 M1 PHYMATECH - TD Gr C TP Modélisation et Simulation en Physique
12h30					
13h00					
13h30					
14h00					
14h30					
15h00					
15h30	HAP708P - GrB Tdinfo 36.205 Montpellier / Triolet / 36 / 2 M1 PHIMV - TD Gr B M1 NANOQUANT - TD Gr B M1 MATIERE ET DESORDRE (SOFTMAT)	HAP708P - GrA Tdinfo 36.205 Montpellier / Triolet / 36 / 2 M1 ASTRO CCP - TD Gr A TP Modélisation et Simulation en Physique	HAP708P - GrC Tdinfo 36.205 Montpellier / Triolet / 36 / 2 M1 PHYMATECH - TD Gr C TP Modélisation et Simulation en Physique	HAP708P - GrB Tdinfo 36.205 Montpellier / Triolet / 36 / 2 M1 PHIMV - TD Gr B M1 NANOQUANT - TD Gr B M1 MATIERE ET DESORDRE (SOFTMAT)	HAP708P - GrA Tdinfo 36.205 Montpellier / Triolet / 36 / 2 M1 ASTRO CCP - TD Gr A TP Modélisation et Simulation en Physique
16h00					
16h30					
17h00	HAP708P - GrB Tdinfo 36.205 Montpellier / Triolet / 36 / 2 M1 PHIMV - TD Gr B M1 NANOQUANT - TD Gr B M1 MATIERE ET DESORDRE (SOFTMAT)	HAP708P - GrA Tdinfo 36.205 Montpellier / Triolet / 36 / 2 M1 ASTRO CCP - TD Gr A TP Modélisation et Simulation en Physique	HAP708P - GrC Tdinfo 36.205 Montpellier / Triolet / 36 / 2 M1 PHYMATECH - TD Gr C TP Modélisation et Simulation en Physique	HAP708P - GrB Tdinfo 36.205 Montpellier / Triolet / 36 / 2 M1 PHIMV - TD Gr B M1 NANOQUANT - TD Gr B M1 MATIERE ET DESORDRE (SOFTMAT)	HAP708P - GrA Tdinfo 36.205 Montpellier / Triolet / 36 / 2 M1 ASTRO CCP - TD Gr A TP Modélisation et Simulation en Physique
17h30					
18h00					
18h30					

S37-09 sept. 24	Lundi 09/09/2024	Mardi 10/09/2024	Mercredi 11/09/2024	Jeudi 12/09/2024	Vendredi 13/09/2024
07h30					
08h00					
08h30					
09h00					
09h30					
10h00	HAP708P - GrB Tdinfo 36.207 Montpellier / Triolet / 36 / 2 M1 PHIMV - TD Gr B M1 NANOQUANT - TD Gr B M1 MATIERE ET DESORDRE (SOFTMAT)	HAP708P - GrA TD info 5.127 Montpellier / Triolet / 05 / RdC M1 ASTRO CCP - TD Gr A TP Modélisation et Simulation en Physique	HAP708P - GrC Tdinfo 36.201 Montpellier / Triolet / 36 / 2 M1 PHYMATECH - TD Gr C TP Modélisation et Simulation en Physique	HAP708P - GrB Tdinfo 36.207 Montpellier / Triolet / 36 / 2 M1 PHIMV - TD Gr B M1 NANOQUANT - TD Gr B M1 MATIERE ET DESORDRE (SOFTMAT)	
10h30					
11h00					
11h30					
12h00	HAP708P - GrB Tdinfo 36.207 Montpellier / Triolet / 36 / 2 M1 PHIMV - TD Gr B M1 NANOQUANT - TD Gr B M1 MATIERE ET DESORDRE (SOFTMAT)	HAP708P - GrA TD info 5.127 Montpellier / Triolet / 05 / RdC M1 ASTRO CCP - TD Gr A TP Modélisation et Simulation en Physique	HAP708P - GrC Tdinfo 36.201 Montpellier / Triolet / 36 / 2 M1 PHYMATECH - TD Gr C TP Modélisation et Simulation en Physique	HAP708P - GrB Tdinfo 36.207 Montpellier / Triolet / 36 / 2 M1 PHIMV - TD Gr B M1 NANOQUANT - TD Gr B M1 MATIERE ET DESORDRE (SOFTMAT)	
12h30					
13h00					
13h30					
14h00					
14h30					
15h00	HAP708P - GrC Tdinfo 36.205 Montpellier / Triolet / 36 / 2 M1 PHYMATECH - TD Gr C TP Modélisation et Simulation en Physique	HAP708P - GrB Tdinfo 36.207 Montpellier / Triolet / 36 / 2 M1 PHIMV - TD Gr B M1 NANOQUANT - TD Gr B M1 MATIERE ET DESORDRE (SOFTMAT)	HAP708P - GrA TD info 5.127 Montpellier / Triolet / 05 / RdC M1 ASTRO CCP - TD Gr A TP Modélisation et Simulation en Physique	HAP708P - GrC Tdinfo 36.205 Montpellier / Triolet / 36 / 2 M1 PHYMATECH - TD Gr C TP Modélisation et Simulation en Physique	
15h30					
16h00					
16h30					
17h00	HAP708P - GrC Tdinfo 36.205 Montpellier / Triolet / 36 / 2 M1 PHYMATECH - TD Gr C TP Modélisation et Simulation en Physique	HAP708P - GrB Tdinfo 36.207 Montpellier / Triolet / 36 / 2 M1 PHIMV - TD Gr B M1 NANOQUANT - TD Gr B M1 MATIERE ET DESORDRE (SOFTMAT)	HAP708P - GrA TD info 5.127 Montpellier / Triolet / 05 / RdC M1 ASTRO CCP - TD Gr A TP Modélisation et Simulation en Physique	HAP708P - GrC Tdinfo 36.205 Montpellier / Triolet / 36 / 2 M1 PHYMATECH - TD Gr C TP Modélisation et Simulation en Physique	
17h30					
18h00					
18h30					

S38-16 sept. 24	Lundi 16/09/2024	Mardi 17/09/2024	Mercredi 18/09/2024	Jeudi 19/09/2024	Vendredi 20/09/2024
07h30					
08h00					
08h30					
09h00					
09h30					
10h00		HAP708P - GrC TP Info 12.1 Montpellier / Triolet / 12 / RdC M1 PHYMATECH - TD Gr C TP Modélisation et Simulation en Physique			
10h30					
11h00					
11h30		HAP708P - GrC TP Info 12.1 Montpellier / Triolet / 12 / RdC M1 PHYMATECH - TD Gr C TP Modélisation et Simulation en Physique			
12h00					
12h30					
13h00					
13h30					
14h00					
14h30					
15h00		HAP708P - GrB TD info 5.127 Montpellier / Triolet / 05 / RdC M1 PHIMV - TD Gr B M1 NANOQUANT - TD Gr B M1 MATIERE ET DESORDRE (SOFTMAT)			
15h30					
16h00					
16h30					
17h00		HAP708P - GrB TD info 5.127 Montpellier / Triolet / 05 / RdC M1 PHIMV - TD Gr B M1 NANOQUANT - TD Gr B M1 MATIERE ET DESORDRE (SOFTMAT)			
17h30					
18h00					
18h30					

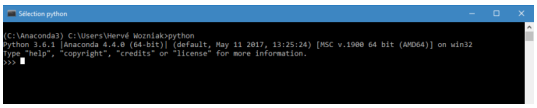
	S39-23 sept. 24				
	Lundi 23/09/2024	Mardi 24/09/2024	Mercredi 25/09/2024	Jeudi 26/09/2024	Vendredi 27/09/2024
07h30					
08h00					
08h30					
09h00					
09h30					
10h00					
10h30					
11h00					
11h30					
12h00					
12h30					
13h00					
13h30					
14h00					
14h30					
15h00		HAP708P - Gr.A TD info 5.127 Montpellier / Triolet / 05 / RdC M1 ASTRO CCP - TD Gr A TP Modélisation et Simulation en Physique			
15h30					
16h00					
16h30					
17h00		HAP708P - Gr.A TD info 5.127 Montpellier / Triolet / 05 / RdC M1 ASTRO CCP - TD Gr A TP Modélisation et Simulation en Physique			
17h30					
18h00					
18h30					

I Python basics

1. The 'Python' environment

In the case of *compiled* languages (C, C++, Fortran...), the execution of a *compiler* on a "source" file containing the program instructions produces an *executable* file. The launch of this executable produces the expected results.

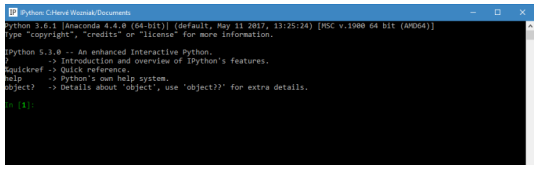
In the case of Python, as for Matlab, Scilab, IDL, Gnuplot, Octave, R, Mathematica, Mapple etc., the language is *interpreted*. It is therefore necessary to set up an environment that will make execution possible as commands are issued. The tip of the iceberg is therefore the interpreter that is simply launched by typing *python* in a terminal (under Linux) or in a command prompt window (under Windows), assuming that everything has been installed correctly.



Launching python in a command window under Windows. The three > are characteristic of the basic Python interpreter. Type 'python3' under Linux to make sure you do not launch python2.

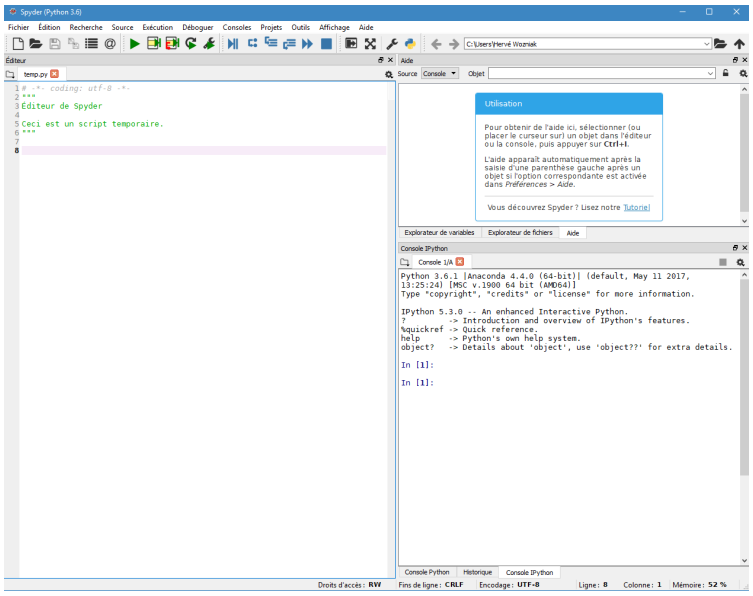
The interpreter is a software layer in itself. Other interpreters than the default one are possible, like IPython (Interactive Python).

The IPython interpreter (also called *IPython console*) has more features than the default one. Be careful not to confuse the version number of Python (here 3.6.1) with that of IPython (here 5.3.0).



Finally, if you want to keep track of the commands, or write a complete program to be reused later, you need a text editor (emacs, vi, edit, notepad, etc.), but not a word-processor like MS-Word, LibreOffice, or OpenOffice!

Alternatively, you can install an additional software layer (*"Integrated Development Environment "* aka *IDE*), such as **Spyder** or *"IDLE "*, which integrates in a unique window a text editor, the python console, the IPython interpreter and much more (help, debugger, file manager, etc.).



The installation does not stop there. Indeed, Python is a modular environment. In other words, some high-level mathematical or graphical libraries are not integrated into the core of Python. This is the case of *NumPy* and *Matplotlib*. These libraries build objects on top of the Python core, which are often easier to use. Arrays, which do not exist in Python unlike in C, C++ or Fortran, are defined in the `numpy` module. We will come back to this later.

Manual installation of these libraries (called *packages*) can be tedious. Therefore, **distributions**, which include many essential libraries as well as, most often, IPython, have been packaged according to the intended use of Python. For scientific programming, **SciPy** and **Anaconda** are examples of such distributions. Pyzo is often used in high schools and CPGE.

We will use **Anaconda** in the following. Although developed by a private company, it is free to distribute and use. It also includes Spyder and Jupyter Notebook. The results are obviously independent of the distribution used, as well as the operating system.

To be installed on your computer (already installed for lab sessions): https://www.anaconda.com/products/individual

2. Syntax elements

Like any language, Python is defined by a syntax, a grammar, rules...

2.1. Variables

A common concept in many programming languages is that of a *variable*. From the programmer's point of view, a variable contains the value that has been assigned to it. In fact, it is a reference to a memory address at which the value is stored in binary form (and if you want to be even more precise, what is stored is the offset from the start address of the data section).

a) Declaration, initialization, assignment

In many compiled languages, a variable must be **declared**, i.e. assigned a **type**. This is called *static typing*. The most common types in compiled languages are :

- Boolean (`bool` or `logical`): formally, the possible values are 0 (`False`) or 1 (`True`) but in practice the 1 is often a 'non-zero' because the memory space reserved to store the boolean is larger than a bit;
- integer (`int` or `integer`): conforms to the mathematical meaning of the term. Languages sometimes introduce the distinction between 'signed' and 'unsigned' (i.e. positive), which is interesting depending on how the integer is to be used (physical value, rank or counter for example). The bit reserved for the sign can then be used to double the range of integers;
- real (`float` or `real`): moves away from the strict mathematical meaning because it confuses decimals and irrationals. Often referred to as 'floating' because of the floating-point representation of real values. There are often several types of precision (simple, double, quadruple...). The representation of reals in memory, which also depends on the hardware architecture, is the main source of error in numerical modelling;
- complex (`complex`): stored as two reals in memory, languages take this type into account by extending the current operations (`+`, `−`, etc.) and adding specific operations (modulus, real part, imaginary part, etc.);
- character or string (`str` or `string`, `char`, `character`, etc.): unloved of the programming languages for a long time because a computer can only handle two numbers: 0 and 1. Hence the need to encode and decode the characters via a conversion table, called the ASCII table (which in turn depends on the encoding system).

In many languages, it is imperative to assign an **initialization** value when declaring or before any use.

For interpreted languages, the prior declaration of variables is not required; the type is then determined (initialization) or modified when a value is assigned. It is therefore a *dynamic typing*.

In Python typing is *dynamic*.

The declaration is implicit and simultaneous with the initialization through an **assignment** (=).

`type()` built-in function returns the type (see Section 2.4).

```
Python 3.6.1 |Anaconda 4.4.0 (64-bit)| (default, May 11 2017, 13:25:24) [MSC v.1900 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> a=42
>>> type(a)
<class 'int'>
>>> a=42.0
>>> type(a)
<class 'float'>
>>> a=10.+2j   # notez l'absence de signe entre 2 et j
>>> type(a)
<class 'complex'>
>>> print(a)
(10+2j)

```

⚠ *Warning* :

From version 3.9 of Python it is possible to declare the type of variables. For the moment we do not consider this (revolutionary!) possibility because lab computers are not yet updated with this version.

A variable name is a sequence of letters (a → z , A → Z) and numbers (0 → 9). It must always begin with a letter.

Only ordinary letters are allowed. Accented letters, cedilla's, spaces, special characters such as $, #, @, etc. are not allowed, with the exception of the character _ (underscore).

Variables that start with _ (underscore) or __ (double underscore) have a special meaning (see chapter "Object-oriented Python").

Case is significant (upper and lower case characters are distinguished).

33 "reserved words" (they are used by the language itself) cannot be used as variable names:

```
and as assert break class continue def del elif else except False finally
for from global if import in is lambda None nonlocal not or pass raise
return True try while with yield
```

Range of numerical types in Python

`bool` only take the values `True` (equivalent to 1 if used as an `int`) and `False` (equivalent to 0).

The range of possible values for numeric types (`int`, `float`, `complex`) generally depends on the number of bytes assigned to the variable. A signed `int` stored in 2 bytes can take values between -32767 and 32768 (0 to 65535 if unsigned).

In Python, the representation of integers is only limited by the total memory space. However, as soon as the storage of integers requires more than 32 or 64 bits (depending on the processor model, 4 or 8 bytes), the computing performance decreases.

Real numbers (`float`) are represented as 8 bytes floating point, which is *double precision* for most other languages. Being a recent language, Python natively integrates the transition to 64-bit processors. If you want to save memory (and because having 15 significant decimal places is often unnecessary), you should use a module that offers single precision floats. NumPy offers single precision floats.

⚠ *Warning* : decimal point

Typing `a=42` or `a=42.0` is not strictly equivalent in dynamic typing because the type is determined by the presence of the decimal point. If the variable is the result of a calculation which is to be used as a positional index in an array or a list, it is imperative to ensure that an integer is obtained, which can only be guaranteed by using an intrinsic conversion function (see below).

Extra : Decimals and fractions

The *fixed-point* representation of real numbers is most often reserved for special hardware devices as DSP (digital signal processors). It needs a specific module (`decimal`) in Python.

It is also the case for fractions (module `fractions`).

We will not deal with these specific forms of real numbers in the rest of the course as they are often useless in physical numerical simulation.

Complex numbers

Python complex literals are written as `real_part+imaginary_part`, where the `imaginary_part` is terminated with a `j` (e.g `0+1j`). Complex numbers may also be created with the `complex(real_part, imaginary_part)` built-in call.

b) Additional information on assignments

The = sign **is not** equivalent to the mathematical sign.

```
>>> a=42    # affecte la valeur 42 à la variable a
>>> a       # équivalent de print(a) en mode interactif
42
>>> a=b     # affecte la valeur de la variable b à la variable a
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'b' is not defined
>>> b=a     # affecte la valeur de a à la variable b qui est créée à ce moment précis
>>> b       # print(b)
42
>>> b=0     # affecte 0 à la variable b
>>> a=b     # affecte la valeur de la variable b à la variable a
>>> a
0
```

A rich syntax around =

```
>>> a,b=42,64
>>> a,b
(42, 64)
>>> a
42
>>> b,a=a,b
>>> a,b
(64, 42)           # permutation 'sur place'
>>>
```

```
>>> a=b=c=78.
>>> a
78.0
>>> b
78.0
>>> c
78.0
>>>
```

Particularities of strings

Variables of type `str` are composite → one can access each of the characters individually.

But `str` are **immutable**, i.e. you cannot change the value of only one of these items (character):

```
>>> a="j'aime python"
>>> print(a[1])   # la numérotation des éléments commence en 0
'
>>> a[1]=' '      # on veut remplacer ' par un blanc
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
```

***Extra* : Mutability**

We will return to the important notion of mutability and immutability in Section 4.3.

2.2. Operators

The first elements of syntax in programming languages related to numerical calculation are the arithmetic operators: +, −, *, /

Not surprisingly, they are present in Python.

Added to this are :

- the modulo operator `%` to calculate the remainder of an integer (Euclidean or floor) division;
- `//` calculates the quotient of integer division (floor division);
- `**` to calculate a power (exponentiation) (also known as `pow()`);

```
>>> 3 // 2
1
>>> 3 % 2
1
>>> 3 / 2
1.5
```

Mixed operation/assignment

A very rich syntax around =: +=, −=, *=, /=

```
>>> a=42.
>>> a+=1  # replaces a=a+1
>>> a
43.0
>>> a*=2  # replaces a=a*2
>>> a
86.0
>>> a-=10 # replaces a=a-10
>>> a
76.0
>>> a/=10 # replaces a=a/10
>>> a
7.6
```

⚠ *Warning : #*

`#` (hash) is not an operator in Python, but introduces a comment, usually at the end of a line following a command, or on a single line.

If you want to write on more than one line, the text is enclosed by a triple `"` (double quote) or a triple `'` (single quote) :

```
"""
Created on Mon Aug 27 14:26:24 2017

@author: Hervé Wozniak
"""
```

Operations on strings

For variables of type `str` the `+` operator performs a *concatenation*, `*` repeats a sequence:

```
>>> a="j'aime"   # simple, double ou triple quotes ou guillemets...
>>> b='python'
>>> c=a+b
>>> c
"j'aimepython"   # notez l'absence de blanc...
>>> print(a,b)   # ... que print() insére toujours
j'aime python
>>>d=4*c
>>> d
"j'aimepythonj'aimepythonj'aimepythonj'aimepython"
```

Operations on Booleans

Variables of type `bool` have their own operations. These variables are either initialized or the result of comparison operations.

`x` **`or`** `y`	true if x or y are true (or both)
`x` **`and`** `y`	true if x and y are both true
`not` `x`	true if x is false and vice versa
`x` **`<`** `y`	true if x is strictly less than y
`x` **`<=`** `y`	true if x is less than or equal to y
`x` **`>`** `y`	true if x is strictly greater than y
`x` **`>=`** `y`	true if x is greater than or equal to y
`x` **`==`** `y`	true if x is equal to y (**not to be confused with =**)
`x` **`!=`** `y`	true if x is different from y

```
>>> x=True
>>> y=False
>>> x or y
True
>>> x and y
False

>>> 2>=3
False
```

```
>>> z=2>=3   # assignment of the result of the comparison to a variable which becomes boolean
>>> print(z)
False

>>> z=2==3
>>> not z
True

>>> x=2
>>> y=3
>>> not ((x>=y) and (x>1))
True
```

⚠ *Warning* :

`bool` variables can be used as `int` variables equal to 0 or 1, and vice versa. This can be useful in some cases, but it can also be a source of confusion and even error, as in the example below.

```
>>> a=True
>>> 3*a
3
>>> b=False
>>> 3*b
0
>>> a+b
1
>>> a and b
False
>>> c=1
>>> a and c
1
```

2.3. Sequences

Python has variable types that are less common in scientific programming languages. These are called composite types.

`str` is already such a composite type; it exists in almost every language.

Several other Python types are for positionally ordered (`list`, `tuple` and `dict`) or unordered (`set`) sequences.

a) Lists

Collection of items separated by commas, all enclosed in square brackets:

```
>>> entree=['Liszt', 'Franz', 1811, 10, 22, 1886, 7, 31]
>>> print(entree)
['Liszt', 'Franz', 1811, 10, 22, 1886, 7, 31]
>>> type(entree)
<class 'list'>
>>> entree[0]
'Liszt'
>>> entree[3]
10
>>> entree[0]='Bach'
>>> entree
['Bach', 'Franz', 1811, 10, 22, 1886, 7, 31]
```

⚠ *Warning* :

"Lists " are objects (as are `str`'s for that matter...) whose manipulation we will be reported later. The items are modifiable (we say **mutable**, unlike strings) but some modifications (lengthening the list for example) require either *methods* or the use of *slicing*. The items of a `list` are ordered, and therefore individually addressable by their position in the list.

An empty list can be created:

```
>>> ma_liste=list()  # or ma_liste=[]
>>> type(ma_liste)
<class 'list'>
>>> ma_liste
[]

```

b) Tuples

Simply, `tuple` are immutable `list` (like `str`). The `( )` replaces the `[ ]`.

It is sometimes interesting/important to avoid the modification of a list. It is then converted into a tuple.

Many functions/commands return tuples.

An empty tuple is created with `tuple()`.

```
>>> un_tuple=(0,'bonjour',ma_liste)
>>> un_tuple
(0, 'bonjour', [])
>>> tuple_vide=tuple()
>>> type(tuple_vide)
<class 'tuple'>
>>> tuple_vide
()

```

c) Dictionnaries

The `dict` are sequences whose items are addressable with a *key* rather than a sequence number.

```
>>> tel = {'herve': 3902 , 'felix': 3559}
>>> tel
{'herve': 3902, 'felix': 3559}
>>> tel['herve']
3902
>>>
```

d) Sequence-specific operations

The `in` operator is used to find out if a particular value is present in a sequence.

For dictionaries, only the keys can be tested.

```
>>> l=[0,1,2,3,4]
>>> x=2
>>> x in l
True
>>> 'felix' in tel
True
>>> 3559 in tel
False
```

The + operator performs a concatenation of two lists or two tuples.

(You can think about this before using the `append` method, which some people already know!)

```
>>> l=(0,1,2,3,4)
>>> g=(10,11,12)
>>> l+g
(0, 1, 2, 3, 4, 10, 11, 12)
```

e) Slicing

It is possible to address only a part of a sequence with the syntax: `[start_index : end_index : step]`

The end index is excluded. This works on all ordered sequence types (`str`, `tuple`, `list`, etc).

An index can be negative: it then represents an offset from the end of the list.

If `step` is negative, the order is reversed.

```
>>> b=[0,1,2,3,4]
>>> print(b[0])    # renvoie la valeur d'indice 0
0
>>> print(b[0:1])  # renvoie une liste de 1 élément: 0
[0]
>>> print(b[0:2])  # renvoie une liste de 2 éléments: 0 et 1
[0, 1]
>>> print(b[1:2])  # idem mais à partir de 1 à 2 exclu, donc 1 seul élément
[1]
>>> print(b[1:4])
[1, 2, 3]
>>> print(b[1:4:2])
[1, 3]
>>> print(b[0:len(b):2])  # extrait un élément sur deux de la liste
[0, 2, 4]
#
# indices négatifs :
#
>>> print(b[-1])  # renvoie le dernier élément
4
>>> print(b[0:-1]) # renvoie une liste de 0 au dernier élément (exclu)
[0, 1, 2, 3]
>>> print(b[0:-2])  # renvoie une liste de 0 à l'avant-dernier élément (exclu)
[0, 1, 2]
>>> print(b[::-1])  # équivalent à la methode reverse()
[4, 3, 2, 1, 0]
```

Method : Replacing methods with slicing

It is tempting to use methods to modify a list (`insert()`, `pop()`, `append`(), ...). This mixes procedural programming with object-oriented programming, which can make the program less readable and even less efficient. But in most cases, methods can be replaced by slicing:

```
>>> b=[0,1,2,3,4]
>>> b[2:2]=['a']   # insert le caractère 'a' à l'indice 2. les [] autour de 'a' sont ici facultatifs car la chaîne ne contient qu'un seul caractère
>>> print(b)
[0, 1, 'a', 2, 3, 4]
>>> b[4:4]=[]    # tente de remplacer l'élément 4 par du vide, donc suppression
>>> print(b)
Out[14]: [0, 1, 'a', 2, 3, 4]  # ne marche pas car la borne sup est exclue
>>> b[4:5]=[]
>>> print(b)
[0, 1, 'a', 2, 4]   # l'élément 4 a été supprimé, pas le 5
>>> b[1:4]='abc'   # insertion de 3 éléments à partir de l'indice 1 donc 1:4 et non 1:3
>>> print(b)
```

```
[0, 'a', 'b', 'c', 4]
>>> b[1:1]=['abc']  # insertion d'une chaine à l'indice 1
```

methods (oriented object programming)	procedural programming	action	result
`b=[0,1,2,3,4]`	`b=[0,1,2,3,4]`		`print(b)`
`a=b.copy()`	`a=b[:]`	true copy (deep copy) new object created	`[0, 1, 2, 3, 4]`
`b.append(5)`	`b=b+[5]` or `b[len(b):len(b)]=[5]`	adds an item to the end of the list `b`	`[0, 1, 2, 3, 4, 5]`
`b.insert(2,10)`	`b[2:2]=[10]`	insert an item (here `10`) at a position in the list (here position `2`)	`[0, 1, 10, 2, 3, 4, 5]`
`b.pop(4)` or `del b[4]`	`b[4:5]=[]`	deletes the item in position `4`	`[0, 1, 10, 2, 4, 5]`
`b.reverse()`	`b=b[::-1]`	reverses the order of the items	`[5, 4, 2, 10, 1, 0]`
`b.remove(10)`	no simple equivalent (search 10 and delete)	removes the first item of `b` which is `10`	`[5, 4, 2, 1, 0]`

2.4. Intrinsic or built-in functions

These functions apply directly to variables.

We have already seen some of them, e.g. `type()` which displays the type of the variable.

a) on-screen printing and echoing

`print()` displays the content (value) of the variable(s) that are in round brackets (parentheses). Text can be added.

This function is mandatory inside a program (non-interactive mode).

In interactive mode (IPython or native console), just type the name of a variable to display (echo) its value.

```
>>> a=42.0
>>> a
42.0
>>> print("a=",a)
a= 42.0
```

b) Deleting a variable

`del()` is used to delete the reference to a variable which can no longer be used in the rest of the program.

```
>>> a,b=42,"Untel"
>>> a,b
(42, 'Untel')
>>> a
42
>>> b
'Untel'
>>> del(b)
>>> a,b
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'b' is not defined
>>>
```

c) Change of type

Force the type of a variable to another type:

- `int()`: changes to an integer;
- `float()`: transforms into a float;
- `str()` or `repr()`: allows most variables of another type to be transformed into strings.

```
>>> a=42.42
>>> b=str(a)
>>> b
'42.42'
>>> print(b)
42.42
>>> type(b)
<class 'str'>
>>> c=int(a)
>>> type(c)
<class 'int'>
>>> c
42
```

d) input()

`input()` allows interaction with the user (value input).

It returns a string (`str`) that must be converted to the desired value.

```
>>> clavier=input("Entrez quelque chose:")
Entrez quelque chose:42.0
>>> type(clavier)
<class 'str'>
>>> valeur=float(clavier)
>>> type(valeur)
<class 'float'>
>>> clavier=input("Entrez quelque chose:")
Entrez quelque chose:42
>>> valeur=int(clavier)
>>> type(valeur)
<class 'int'>
```

Extra :

Obviously, one should know beforehand the numeric type (`int`, `float`) into which one wants to convert the string entered from the keyboard. If this is not known, we must foresee all cases.

```
"""
Created on Wed Jul  7 11:18:20 2021

@author: hwozniak
"""
clavier=input("Entrez quelque chose:")
if clavier.isdigit():
    valeur=int(clavier)
else:
    try:
        valeur=float(clavier)
    except ValueError:
        print("je ne sais pas convertir")
        valeur=clavier
print("valeur=",valeur)
```

e) len()

`len()` returns the number of items in a sequence.

If the sequence is a `str`, the number of letters, including blanks, is returned.

If the sequence is a `list`, it is the number of items in the list.

```
>>> a='''j'aime python'''  # ou a="j'aime python"
>>> type(a)
<class 'str'>
>>> a
"j'aime python"
>>> print(a)      # notez la différence de présentation du valeur retournée
j'aime python
>>> len(a)
13
>>> a='j\'aime python'  # une des syntaxes alternatives
>>> a
"j'aime python"
>>> len(a)
13
>>> a='''j'\'aime python'''  # mais ne faut pas tout mélanger !
>>> a
"j''aime python"
>>> len(a)
14
>>>
```

```
>>> len(tuple_vide)  #tuple_vide=tuple()
0
>>> len(tel)     # {'herve': 3902 , 'felix': 3559}
2
>>> len(entree)  # ['Bach', 'Franz', 1811, 10, 22, 1886, 7, 31]
8
```

f) range()

In graphical visualization, it is often necessary to create a list of regularly spaced values.

- `range(number)` creates a sequence from 0 to `number` **excluded** (interval [0,`number`[).
- `range(start, end)` creates an arithmetic sequence of integers between `start` and `end` **excluded**.
- `range(start, end, step)` creates a sequence between `start` and `end` (excluded), by steps of `step`.

```
>>> r=range(10)
>>> r
range(0, 10)
>>> type(r)
<class 'range'>
>>>
```

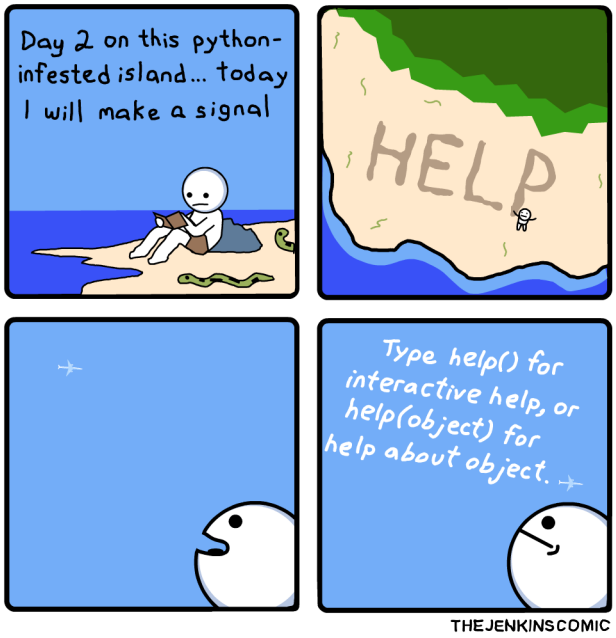
`range()` therefore creates an *object*. To make explicit what this object contains, it can be transformed into a `list`:

```
>>> r=list(range(10))
>>> r
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> r=list(range(5,10,2))
>>> r
[5, 7, 9]
>>> r=list(range(5,11,2))  # la borne supérieure est exclue
>>> r
[5, 7, 9]
>>> len(r)
3
>>> r[1]
7
```

g) help()

Do we really need to explain what `help()` does?

`help(int)` sends help about the `int()` function.



```
>>> help(help)
Help on _Helper in module _sitebuiltins object:

class _Helper(builtins.object)
 |  Define the builtin 'help'.
 |
 |  This is a wrapper around pydoc.help that provides a helpful message
 |  when 'help' is typed at the Python interactive prompt.
 |
 |  Calling help() at the Python prompt starts an interactive help session.
 |  Calling help(thing) prints help for the python object 'thing'.
 |
 |  Methods defined here:
 |
 |  __call__(self, *args, **kwds)
 |      Call self as a function.
 |
 |  __repr__(self)
 |      Return repr(self).
 |
 |  ----------------------------------------------------------------------
 |  Data descriptors defined here:
 |
 |  __dict__
 |      dictionary for instance variables (if defined)
 |
 |  __weakref__
 |      list of weak references to the object (if defined)
```

3. Control-flow operations

It is rare that a program is just a linear sequence of instructions. All high-level programming languages (since their origin) offer the possibility to jump around in the code using two different processes (*control-flow statements*):

- execute or not certain passages (possibly under conditions). It is *branching*;
- repeat certain parts of a program for a finite number of iterations or until a condition is met. It is *looping*.

3.1. Conditional branching

Conditional branching is where a program decides whether to do something or not according to the result of some condition.

Therefore, blocks of instructions must be defined that are executed or not depending on the test execution.

As a general rule, conditional branching is not used in interactive mode.

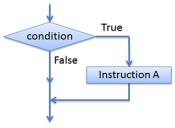
a) if statement

The simplest test uses the `if` statement which has the effect of stopping the execution flow to check a `condition`:

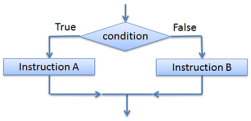
```
if boolean_condition:
    block of instructions if boolean_condition is True
else:
    block of instructions if boolean_condition is False
```

`if` is a *compound statement* (i.e. a statement that embed other statements).

Nested blocks of statements are indented with 4 blank characters or a tab.



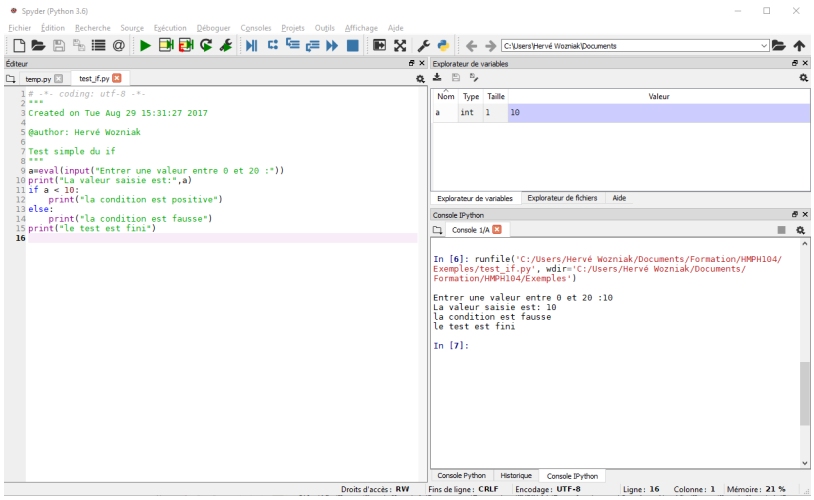
Graphic 1



Graphic 2

Example of first script

```
# -*- coding: utf-8 -*-
"""
Created on Tue Aug 29 15:31:27 2017

@author: Hervé Wozniak

Test simple du if
"""
a=float(input("Entrer une valeur entre 0 et 20 :"))
print("La valeur saisie est:",a)
if a < 10:
    print("la condition est positive")      # bloc si condition vraie
else:                                       # facultatif
    print("la condition est fausse")        # bloc si condition fausse
print("le test est fini")
```



The `else :` block is optional.

If it is absent, execution resumes after the `if ... :` block:

You can insert `elif ... :` (contraction of *else if*) which will test another condition after the `if ... :` block:

```
a=float(input("Entrer une valeur :"))
print("La valeur saisie est:",a)
if a < 0:
    print("a est strictement négatif")
elif a > 0:
    print("a est strictement positif")
else:
    print("a est nul")
print("le test est fini")

```

b) match statement

From version 3.10 of Python, the `match` statement will allow to execute a block of statements for values of a variable, like `switch` (C) or `case` (Fortran). It's too early to talk about it because the version of Python in the lab is 3.8.

3.2. Looping constructs

What a computer is much better at doing is repeating the same operation.

To infinity: we don't really see the point :-)

So for a finite number of iterations or until a condition is met.

a) for loop

The syntax of the `for` loop statement is similar to that of the `if` statement, with a block of statements indented with 4 blank characters (or a tab)

```
for variable(s) in liste_de_valeurs :
    block of repeated instructions

```

Unlike many other languages, we do not give a `min`, `max` and `step`, but the set of values to be iterated, stored in an *iterable* object.

The behaviour of other languages (numerical iteration) can be simulated with the intrinsic function `range()`.

```
>>> for i in range(3):  #pas nécessaire d'expliciter avec list(range())
...     print(i)
...
0
1
2
>>>
```

But the `for` statement is much more general:

```
>>> c = ["Marc", "est", "dans", "le", "jardin"]
>>> for i in range(len(c)):
...     print("i vaut",i, "et c[",i,"] vaut",c[i])
...
i vaut 0 et c[ 0 ] vaut Marc
i vaut 1 et c[ 1 ] vaut est
i vaut 2 et c[ 2 ] vaut dans
i vaut 3 et c[ 3 ] vaut le
i vaut 4 et c[ 4 ] vaut jardin
>>>
```

Easier, if you just need to display the list because a `list` is an iterable:

```
>>> for i in c:
...     print("i vaut", i)
...
i vaut Marc
i vaut est
i vaut dans
i vaut le
i vaut jardin
```

Other solutions exist if you need both the value and its positional index in the list:

```
>>> for i,v in enumerate(c):
...     print("i vaut", i, "et c[",i,"] vaut", v)
...
i vaut 0 et c[ 0 ] vaut Marc
i vaut 1 et c[ 1 ] vaut est
i vaut 2 et c[ 2 ] vaut dans
i vaut 3 et c[ 3 ] vaut le
i vaut 4 et c[ 4 ] vaut jardin
>>> list(enumerate(c))          # enumerate() crée un object (position, valeur)
[(0, 'Marc'), (1, 'est'), (2, 'dans'), (3, 'le'), (4, 'jardin')]
>>>
```

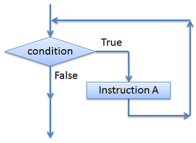
Extra : Iterators

We will generalize the notion of *iterators* and associated functions (*generators*) in the chapter on functions (Section 4.2)

b) while loop

One may want to iterate until a Boolean condition is met:

```
while boolean_condition:
        block of repeated instructions if boolean_condition is True

```



Warning :

With the `while` statement, the test is performed before entering the loop.

If the test expression is always `false`, the instructions inside the loop are never executed.

If the test expression is always `true`, the loop is infinite... the program never stops!

Extra : Advanced exercise

Some languages have a `repeat ... until` statement. This means that the test is executed on exit after the first iteration. How do you simulate this behaviour in Python?

```
repeat
   bloc_d'instructions
until condition_booleenne
```

c) Prematurely exiting the loops

If you need to exit a `for` loop or not wait for a `while` condition to be `False`, there is a `break` statement.

It is especially useful in nested loops (loop within a loop) and is often associated with other statements (`continue`) or clauses (`else`).

Find the prime numbers less than 10

```
>>> for n in range(2, 10):      # boucle EXTERNE avec n de 2 à 10
...     for x in range(2, n):   # boucle INTERNE de 2 à n
...         if n % x == 0:      # reste division euclienne n/x
...             print(n, 'egale', x, '*', n//x)  # ==0 donc diviseur trouvé
...             break           # pas besoin d'aller plus loin ; on itère sur n (externe)
...     else:                   # relative à for INTERNE et non pas au if !!
...         # on a terminé la boucle externe sans trouver de diviseur donc :
...         print(n, 'est un nombre premier')
```

```
...     # fin de la boucle interne
... # fin de la bouche externe
2 est un nombre premier
3 est un nombre premier
4 egale 2 * 2
5 est un nombre premier
6 egale 2 * 3
7 est un nombre premier
8 egale 2 * 4
9 egale 3 * 3
```

Note :

The `else` clause is only executed when all iterations have been performed. It is therefore the "N+1th" iteration.

In case of a `break`, the `else` clause is not executed because it is considered as part of the loop even if it is beyond the maximum number of iterations.

The `continue` statement allows the program pointer to skip the rest and jump immediately to the next iteration (whereas `break` exits the loop).

```
>>> for num in range(2, 10):
...     if num % 2 == 0:
...         print("Nombre pair", num)
...         continue   # on retourne au compteur de boucle
...     print("Nombre impair", num)
...
Nombre pair 2
Nombre impair 3
Nombre pair 4
Nombre impair 5
Nombre pair 6
Nombre impair 7
Nombre pair 8
Nombre impair 9
>>>
```

An equivalent form with the `else` of `if`:

```
>>> for num in range(2, 10):
...     if num % 2 == 0:
...         print("Nombre pair", num)
...     else:
...         print("Nombre impair", num)
...
Nombre pair 2
Nombre impair 3
Nombre pair 4
Nombre impair 5
Nombre pair 6
Nombre impair 7
Nombre pair 8
Nombre impair 9
>>>
```

d) List comprehension

List comprehension is a specific feature of Python. It provides a fast way to create lists based on a simple arithmetic algorithm. The syntax is close to mathematical syntax. These lists are often used in combination with the creation of `NumPy` arrays.

Example: we want to make a list of integers whose square is less than or equal to 25. So `a=[0, 1, 2, 3, 4, 5]`.

```
# première solution : usage d'une boucle explicite
a=[]
n=25
for i in range(n):
    if i**2 <=n:
        a=a+[i]
```

```
# seconde version : liste en compréhension
n=25
a=[i for i in range(n) if i**2 <=n]
```

In mathematical (set) notation, this recurrence is written: $\{i | i \in \mathbb{N},\ i^2 \leq 25\}$.

e) Loop advice

The `for` loop is generally simpler to code and often quicker to run than a `while`, so it's the first tool you should reach for whenever you need to step through a sequence or other iterable. In fact, as a general rule, you should resist the temptation to count things in Python. Its iteration tools automate much of the work you do to loop over collections in lower-level languages like C. `NumPy` will provide an additional level of abstraction to get rid of loops (implicit loops).

4. Internal and external organization of a program

A program cannot always consist of a sequence of instructions, loops and tests.

More often than not, it must be structured into functional units (functions, subprograms, etc.).

These units can become autonomous (libraries, modules, packages, etc.) and reused.

4.1. Function basics

A number of *built-in* functions have already been seen.

It is now a question of making your own *functions*.

The interest of a function is to return one or more values which depend on one or more input parameters (*arguments*).

If a function does not return values, it is formally called a *procedure*, but the definition in Python is identical (as in many languages).

A function/procedure isolates a block of instructions that form a well-defined task, possibly repeated several times with different arguments.

Warning :

We do not introduce `lambda` or anonymous functions.

Note :

Like tests and loops, functions and procedures change the order of instructions (control-flow).

The execution flow is momentarily interrupted for branching to a function.

When the function has finished its run the program resumes its normal execution.

def statement

`def` is a compound statement. It is thus followed by an indented nested block of statements, as for `for`, `if` etc.

The name of a function/procedure follows the same rules as for variables.

```
def nom_fonction(comma-separated list of arguments):
      block of instructions
```

Procedure without arguments

```
>>> def compteur3(): # début de la définition de la fonction
...     i = 0
...     while i < 3:
...         print(i)
...         i = i + 1
...
>>> compteur3()
0
1
2
```

Note :

Once in memory, the `counter3()` procedure can be called several times, possibly within other functions or procedures.

But as soon as you exit the Python or IPython console, the function is lost.

Method :

There are two ways to save functions and procedures:

- systematically place them at the beginning of the file containing the program (they are not executed before being called, so they are just stored in memory while waiting);
- store them in a separate file that is imported at the beginning of the program (see Section "modules and packages").

Procedure with arguments

It is rare to create functions or procedures without passing arguments.

```
>>> def compteur(stop):    # mise en mémoire de la fonction compteur()
...     i = 0
...     while i < stop:
...         print(i)
...         i = i + 1
...
>>> compteur(3)  # appel à compteur() avec l'argument 3
0
1
2
>>> compteur(1)
0
```

Scope of variable: local vs global

In the previous example, the `stop` variable is said to be *local*.

It is only visible inside the `counter()` procedure.

Any attempt to access the value of `stop` in the main program will fail. Spyder's code analyzer can see the problem before any execution!

⚠ *Warning* : Duplicate variable names

The name `stop` can be reused in the main program, with another type, without changing its value when calling the `counter()` function:

```
def compteur(stop):
    print('compteur:: stop,id(stop)=',stop,id(stop))
    i = 0
    while i < stop:    # on s'attend à une valeur numérique
        print(i)
        i = i + 1

stop="why not"  # initialisation à une 'str'
compteur(3)
print('main::stop,id(stop)=',stop,id(stop))
compteur(1)
print('main::stop,id(stop)=',stop,id(stop))

```

```
runfile('C:/Users/Hervé Wozniak/Documents/Formation/HMPH104/Exemples/compteur3.py',
wdir='C:/Users/Hervé Wozniak/Documents/Formation/HMPH104/Exemples')
compteur:: stop,id(stop)= 3 1836378640
0
1
2
main::stop,id(stop)= why not 2293388139576
compteur:: stop,id(stop)= 1 1836378576
0
main::stop,id(stop)= why not 2293388139576
```

⊕ *Extra* : id() built-in function

The `id()` built-in function returns a unique identifier (similar to the address in memory) that can be used to tell you whether two variables (*objects* to be exact) with the same name are identical.

💬 *Advice* : Advice

When functions are defined at the beginning of a program, it is strongly advised not to use the same variable names as in the main program. It is not forbidden but using different names significantly improves the reading and understanding of the code!

Scope of variable: local vs global

If you absolutely need to have access, in the function, to the value of a variable defined in the main program (program calling the function), then the variable must be defined as **global**.

```
def compteur():  # pas d'argument
    global stop  # declaration de portée, pas d'initialisation
    print('compteur:: stop,id(stop)=',stop,id(stop))
    i = 0
    while i < stop:
        print(i)
        i = i + 1

stop=3  # initialisation par le programme principal
compteur()
print('main::stop,id(stop)=',stop,id(stop))
stop=1
compteur()
print('main::stop,id(stop)=',stop,id(stop))
```

```
compteur:: stop,id(stop)= 3 140716676884320
0
1
2
main::stop,id(stop)= 3 140716676884320
```

```
compteur:: stop,id(stop)= 1 140716676884256
0
main::stop,id(stop)= 1 140716676884256
```

If a variable is declared `global` then its value can be changed in a function.

When the python interpreter encounters a variable declared `global`, it looks for a variable with the same reference (name) in the main program. This variable must therefore have been initialized beforehand, otherwise an error message is displayed.

Genuine function (with return statement)

Remainder : a function that executes lines of code without returning any value to the main program is called a procedure.

In the following example, the value of the ratio cannot be retrieved and stored in a variable to be used later.

```
>>> def ratio(x,y):
...     print(x/y)
...
>>> ratio(1,2)
0.5
>>> a,b=2,5
>>> ratio(a,b)
0.4
>>>
```

We must therefore modify the definition by specifying what is returned to the calling program with the `return` statement:

```
>>> def ratio(x,y):
...     return x/y
...
>>> c=ratio(a,b)   # stocke le résultat dans c
>>> print(c)       # affiche la valeur de c
0.4

```

Order of arguments

In the above example of the `ratio()` function, the numerator and denominator must not be swapped when passing arguments. But there is no indication whether the first argument is the denominator or the numerator.

```
>>> a,b=2,5
>>> ratio(a,b)
0.4
>>> ratio(b,a)
2.5
```

Arguments can be labelled as long as they have been assigned a default value:

```
>>> # on affecte des valeurs par défaut
>>> def ratio(numerateur=0., denominateur=1.):
...     return numerateur/denominateur
...
>>> ratio()
0.0            # retour par défaut
>>> ratio(1,2) #
0.5
>>> ratio(denominateur=2,numerateur=1)
0.5
```

Place of a function/procedure in a program

When writing a program (*"volume.py "*) that contains the definition of functions, these definitions must precede the body of the main program (caller).

In the following example we use Spyder:

```
1 def cube(n):
2     return n**3
3
4 def volume_sphere(r):
5     return 4 / 3 * 3.1415926 * cube(r)
6
7 r = float(input("Entrez la valeur du rayon : "))
8 print("Le volume de cette sphere vaut", volume_sphere(r))
```

Result:

```
1 Entrez la valeur du rayon : 1.0
2 Le volume de cette sphere vaut 4.188790133333333
```

Method **:**

Since Python is an interpreted language, functions must be defined before being used.

The functions are then stored in memory.

Therefore, the main program (calling functions and procedures) is always at the end.

This is true in all interpreted languages.

Extra **:**

When the execution pointer encounters a call to a function, it jumps to the memory address where the function starts.

The execution flow is therefore modified.

Obviously, the function must have been defined or imported beforehand (built-in functions are always available in memory).

4.2. Complement on iterators and generators

We deal with these concepts now because they involve a special case of functions.

Definition **:**

An *iterator* is an object that moves through a sequence, called *iterable*, usually created by a *generator*.

Example **:**

For example, the variable `i` in a the statement `for i in range(10)` is an *iterator*, while `range(10)` is a *generator* that creates an *iterable* object of type `range`.

Iterable objects are also lists (`list`), strings (`str`), dictionaries (`dict`), etc.

Generator functions and expressions

You may need to create your own generator without using the built-in tools.

Python provides tools that produce results only when needed, instead of all at once.

A *generator function* returns the next item of a sequence each time the iterator requests it.

These items do not need to be pre-computed and stored (e.g. in a list); **they are computed on request**.

The element that distinguishes a generator function from a normal function is the `yield` statement, which specifies what is to be returned to the iterator.

It is at the same place as the `return` of a basic function. Therefore, any generating function must end with the `yield` statement.

Memory aid : a generator function **yields** a value rather than **returning** one.

```
def generateur_pair(i):
    for j in range(i):
        if j%2==0:
            yield j

pair=generateur_pair(100)
print('Generateur=',list(pair))

def fonction_pair(i):
    for j in range(i):
        if j%2==0:
            return j

print('Fonction equivalente=',fonction_pair(100))
```

Generateur= [0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32, 34, 36, 38, 40, 42, 44, 46, 48, 50, 52, 54, 56, 58, 60, 62, 64, 66, 68, 70, 72, 74, 76, 78, 80, 82, 84, 86, 88, 90, 92, 94, 96, 98]

Fonction equivalente= 0

Warning **:**

As with `range()`, the generator function returns an object, not a list of the object's values. You must therefore use `list()` to make the contents of the generator explicit.

4.3. Mutability and immutability

This notion is tricky because it is linked to object-oriented languages (*OO*) of which Python is a part.

But it should be introduced now because it will help to understand the behaviour of a function's arguments. This is particularly important if there is an apparent bug when calling a function.

Before OO languages :

- variables: could be changed at will;
- constants: specifically declared and initialized so that they could not be changed along the execution.

This is the emblematic case of (old) Fortran 77 ("parameters").

In OO languages (Python, C++, Java, Ruby etc.) objects are manipulated via references (even `int`, `float`, etc. seen at the beginning of this course).

`b=a` does not create a full copy of `a`, but only a copy of the reference to the object (known with the `id()` built-in function).

Copying only a reference and not the whole object is fast and memory-saving.

If the object `a` is *immutable*, it cannot be modified (neither can `b`).

If `a` is *mutable*, changing `a` will also change `b`!

If you need to change `a` without touching `b`, you have to make a *deep copy*.

```
# exemple avec des list (mutables)
#
>>> a=['a','b','c']
>>> b=a
>>> print('id(a),id(b)=',id(a),id(b))
id(a),id(b)= 1595795076552 1595795076552

>>> b[0]='modif de b'       # change la valeur du premier élément de 'b'
>>> print('id(a),id(b)=',id(a),id(b))  # les références restent identiques
```

```
id(a),id(b)= 1595795076552 1595795076552

>>> print('a=',a)  # la liste 'a' a également été modifiée
a= ['modif de b', 'b', 'c']

>>> c=list(a)   # ou c=a[:], crée une copie de 'a'
>>>
>>> c[0]='nouvelle valeur de c'
>>> print('c=',c)
c= ['nouvelle valeur de c', 'b', 'c']

>>> print('id(a),id(b),id(c)=',id(a),id(b),id(c))
id(a),id(b),id(c)= 1595795076552 1595795076552 1595795171016

>>> print(a)
['modif de b', 'b', 'c']
```

⚠ *Warning* :

`list()` makes a deep copy, so creates a second *object* in memory.

As do the **slicing**:

```
>>> a=3*[0]
>>> print(id(a))
2209941188808
>>> b=a[:]      # opération de slicing crée aussi une copie
>>> print(id(b))
2209941190088
```

More subtle with immutable objects

```
>>> a=42.
>>> id(a)     # identifie l'objet de manière unique
2949452993088
>>> b=a
>>> id(b)
2949452993088 # c'est bien le même objet, avec un second nom
```

```
>>> b+=1  #on incrémente b
>>> a
42.0
>>> b
43.0
>>> id(a)
2949452993088
>>> id(b)
2949452992920  # ce n'est plus le même objet
>>> a*=2
>>> id(a)
2949452992824  # a n'est plus le même objet que 2949452993088
```

```
>>> c=(0,1,2,3)
>>> d=c
>>> id(c),id(d)
Out[20]: (2254575392064, 2254575392064)
>>> c+=(4,) #concatenate a fifth element
>>> c
Out[28]: (0, 1, 2, 3, 4)
>>> d
Out[29]: (0, 1, 2, 3) #now different from c
>>> id(c),id(d)
Out[35]: (2254575727056, 2254575392064)# not the same object anymore
```

Fundamental **:**

The basic types (`int`, `float`, `str`...) are immutable.

Changing their value therefore creates a new object.

Warning **:**

Not being able to change an *object* does not mean that you cannot change its *value*!

This is sometimes done at the cost of a new reference with the same name.

But the value of an immutable object (e.g. `str`) is the whole (e.g. string).

Definition **:**

Mutable types: `list`, `dict` (, `set`)

Immutable types: `bool`, `int`, `float`, `str`, `complex`, `tuple` (, `frozenset`)

`id()` allows to know if we have the same object (tests on the `id` of the objects and variables)

Fundamental **: Passing arguments by reference or by value**

Programming languages offer two modes for passing variables as arguments of a function:

passing by value: the values of the calling parameters are copied into local variables of the called function. Thus the called function works with a copy of the parameters. A major advantage of this mode is the absence of side effects: the called function can modify the local variables without affecting the value of the parameters in the calling program. A disadvantage, on the other hand, may be the time it takes to copy the parameters, as well as the amount of memory used.

passing by reference: the called function will work with the references of the calling arguments. Therefore, unless the syntax forbids it (as in C++), the function can modify the variables of the calling program that serve as arguments. An advantage of this mode of transmission is that it saves memory and computing time.

Passing by value is the only way to pass parameters in C and C++, but the possibility to have *pointer* arguments replaces a classical passing by reference.

In Python, argument-passing is done by reference.

But it is not necessarily possible to modify the arguments. It depends on whether they are mutable or not.

Example **: Modification of a list passed as an argument**

Since a `list` is mutable, its values can be modified in a procedure if it is passed as an argument, because it is the same reference in and out of the function (without even having declared it as `global`):

```
>>> def modif_list(l):
...     l[0]='modifié'
...
>>> ma_liste=3*[0.]
>>> print(ma_liste,id(ma_liste))
[0.0, 0.0, 0.0] 1595795075272
>>>
>>> modif_list(ma_liste)
>>> print(ma_liste,id(ma_liste))
['modifié', 0.0, 0.0] 1595795075272
```

⚠ *Warning* : Modification of a float passed as an argument

Changing the value of an immutable argument can be done at the cost of a new reference (with `return`) because the initial object is not modified:

```
>>> def modif_valeur(r):  # r est local
...     r+=1              # incrémente de 1
...     return r
...
>>> mon_float=1.0
>>> print('Initialisation=',mon_float,id(mon_float))
Initialisation= 1.0 1595791249816
>>>
>>> mon_float=modif_valeur(mon_float)
>>> print('Après appel=',mon_float,id(mon_float))
Après appel= 2.0 1595791249720
```

⊕ *Extra* : Another way to look at it

- Immutable arguments are **effectively** passed by value. Integers and strings are passed by reference instead of copying, but because you cannot change immutable objects in place anyhow, the effect is much like making a copy.
- Mutable arguments are **effectively** passed by reference. Mutable objects (like lists and dictionaries) can be changed in place in the function.

💬 *Advice* : Side effect on a function

If a `list` is passed as an argument to a function, but you need to make sure that the function cannot modify it (you want to avoid a **side effect**), then it is better to transform the list into a `tuple` before the call: `tuple(list)` and `list(tuple)` change the types.

```
>>> modif_list(tuple(ma_liste))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 2, in modif_list
TypeError: 'tuple' object does not support item assignment
```

⊕ *Extra* :

If you need to pass a `list` as an argument but want the function returns the same `list` modified without touching the first one, the function must return a modified copy (thus a new reference). Example with a deep copy:

```
>>> def modif_list(l):
...     lmodif=l[:]  # copie intégrale (ou profonde)
...     lmodif[0]='element modifié'
...     return lmodif
...
>>> a=3*[0]
>>> print(a,id(a))
[0, 0, 0] 2484717122824
>>> b=modif_list(a)
>>> print(b,id(b))
['element modifié', 0, 0] 2484717121736
```

4.4. Modules and packages

After defining a function, you may want to use it in several programs, written in several independent *".py "* files.

You may want to avoid having to copy and paste your function into several programs.

Possibly, the function may have been written by another programmer.

Most languages therefore propose to isolate functions/procedures in files different from the main program.

One or more functions/procedures can be merged into a file (called a *module* in Python).

Several modules can be batched in a function library (called a *package* in Python)

Creating a module

Functions/procedures must be located in a file with a *".py "* extension (for example *"puissance.py "*)

In the simplest case, this module file must be in the same directory as the file containing the main program.

```
def carre(valeur):
    return valeur**2

def cube(valeur):
    return valeur**3
```

Importing a module

In the main program, the desired function is imported with the `from... import` statement at the beginning of the program, or the entire module with `from...import *`.

```
from puissance import carre      # uniquement la fonction carre
from puissance import *          # importe tout le module
from puissance import carre,cube # les deux fonctions


```

Namespace

It is not recommended to import everything because:

- this unnecessarily increases the amount of memory used without knowing the size of the module;
- this can create function name conflicts, especially if several modules are imported.

To solve this last case, the whole module can be imported while keeping the namespace, using the form `import...:`

```
>>> import math  # module python présent dans toutes les distributions
>>> math.pi
3.141592653589793
>>> math.sqrt(2)
1.4142135623730951
>>> sqrt(2)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'sqrt' is not defined
```

Namespace: alias

As programming can become tedious with long module names, the module name can be simplified with an alias:

```
>>> import math as m
>>> m.pi
3.141592653589793
>>> m.sqrt(2)
1.4142135623730951
>>> sqrt(2)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'sqrt' is not defined
```

Namespace: deletion

Importing with the form `from...import` allows the programmer to get rid of the namespace:

```
>>> from math import pi,sqrt
>>> pi
3.141592653589793
>>> sqrt(2)
1.4142135623730951
```

Making your own module

If you collect several files containing functions in a directory called *repertoire_fonctions* which is grafted to your working directory (or present in the python *path*) then :

```
import repertoire_fonctions.puissance as p
print(p.carre(2))
```

To import all its functions, present in all the files of the *repertoire_fonctions* directory:

```
import repertoire_fonctions.puissance          # espace de nommage conservé
print(repertoire_fonctions.puissance.carre(2))
```

Reminder : Import statement

The syntax around `import` is rich:

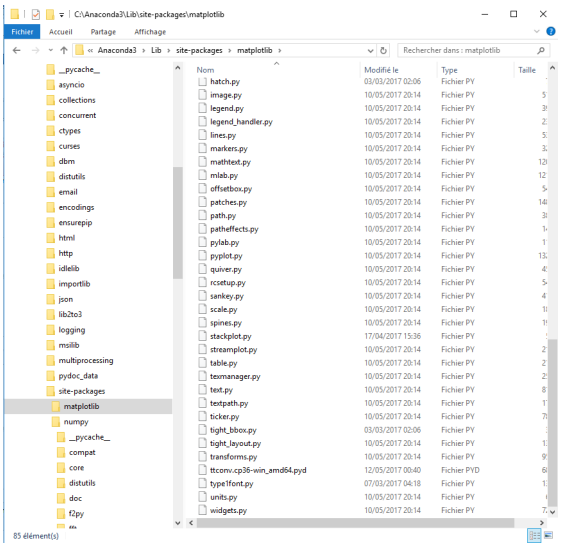
- `import mes_fonctions`: import all functions from *"mes_fonctions.py "*; they will be prefixed with `mes_fonctions`.
- `import mes_fonctions as mf`: import all functions that will be accessible preceded by `mf`.
- `from mes_fonctions import cube`: import only the `cube()` function of *"mes_fonctions.py "*, which takes the name of `cube()`
- `from mes_fonctions import cube as cc`: idem but the function is now called `cc()`
- `from mes_fonctions import *`: imports all functions, without prefix, with the risk of colliding with other functions of the same name.

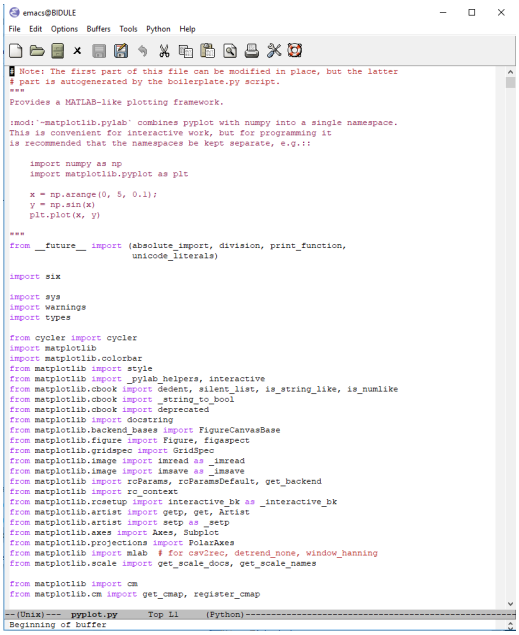
Package

Several modules gathered in a directory become a *package* which takes the name of the directory.

All or part of the *package* can be imported. For example, the `pyplot` module of the `matplotlib` package that we will see later:

```
import matplotlib.pyplot
```





Useful packages and built-in modules

`math`: `sqrt()`, `cos()`, `sin()` etc. functions https://docs.python.org/3/library/math.html;

`cmath`: to manipulate complex numbers;

`random`: random generators https://docs.python.org/3/library/random.html;

`os`: to communicate with the operating system, change the working directory (`os.chdir`), check the existence of a file (`os.path.exists`) etc;

`sys`: to change the console prompt ;-) but also the path to the modules and the command line arguments to use python as shell commands. https://docs.python.org/3/library/sys.html

etc.

```
1 def volume_sphere(r):
2     import math
3     return 4 / 3 * math.pi * cube(r)
```

⊕ *Extra* : PYTHONPATH

If you want to respect some basic rules of work organization, you should separate the programs and the place where they run.

So you have to tell Python :

1. the directory in which it should run (where the data files, calculation results, etc. may be located);
2. the directory(ies) in which the programs and modules are stored. This is the place where development takes place.

To execute, you just have to launch the python console, or IPython from the working directory. With Spyder, you have to navigate to the desired directory with the menu generally located on the top right by default. Spyder allows you to change your working directory without having to quit and restart it.

To tell Python where the programs are located, the directory(ies) must be specified in an environment variable at the *shell* command layer (Linux console level). **Before** launching python3:

```
export PYTHONPATH=$PYTHONPATH:/home/hwozniak/test
```

where what is after the *":"* can be the name of any directory that contains the programs and modules (note that the exact syntax may depend on the shell used but by default it is often *bash*). In Spyder, there is a dedicated menu.

On Windows there are also environment variables but as you rarely work in console mode (which is provided by the Anaconda installation), it is less useful. Nevertheless, for fans of console mode 'à la linux': 'System' menu, then 'Advanced system settings' (at the bottom), 'Environment variables...' (new window), etc. Good luck!

5. Documenting and making your program readable

5.1. Comments

We have already seen that any text starting with the `#` symbol will not be interpreted. It is therefore used to add a comment at the end of a line of code, or for a whole line by starting the line with `#` followed by a space.

You can also introduce a comment on several lines by opening and closing the area with a triple `'` (single quote) or a triple `"` (double quote).

The comments frequency depends on the complexity of the program. But, as a general rule, when you reread a program that has been written for more than 6 months, you will find that there should be more comments. Lines of code that seem obvious when written are often no longer obvious a few months later, or were never obvious when reread by a third party.

A comment should therefore be written as if it were addressed to a stranger who should understand how the program works, and not for yourself. Even if you end up being that stranger some time later!

5.2. Well written program

The aesthetics of a program are usually of no use in terms of execution speed or accuracy.

However, there are rules for writing, like writing a novel, to make it easier for someone else to read the program.

In a professional environment, it is rare for a program to be used only for oneself. It should therefore always be possible for someone else to take over, modify and improve it.

These writing rules mainly deal with the syntax, the organization of the code, and the division into functional blocks.

The PEP (*Python Enhancement Proposal*) that serves as the most common (or most famous) writing standard for Python is PEP 8 (https://www.python.org/dev/peps/pep-0008/). We will not describe it here, but point out that it is according to these writing rules that :

- indentations are 4 blank characters long (whereas tabs or more blank characters could be used);
- the imports are all grouped together at the beginning of the program (whereas one could import just before needing them);
- the syntax `from ... import *` must be avoided;
- operators +, −, =, etc. are surrounded by a white space before and after;
- the length of a line does not exceed 79 characters;
- etc.

Fundamental :

Don't waste your time making a program "beautiful" (you are not rated on aesthetics).

Make sure it works and gives good results first.

5.3. Docstrings

It is strongly recommended to write a comment text (called *docstring*) just after the declaration of a function, as in the following example (https://python.sdv.univ-paris-diderot.fr/15_bonnes_pratiques/):

```
def multiplie_nombres(nombre1, nombre2):
    """Multiplication de deux nombres entiers.

    Cette fonction ne sert pas à grand chose.

    Parameters
    ----------
    nombre1 : int
        Le premier nombre entier.
    nombre2 : int
        Le second nombre entier.

        Avec une description plus longue.
        Sur plusieurs lignes.

    Returns
    -------
    int
        Le produit des deux nombres.
    """
    return nombre1 * nombre2
```

PEP 257 (https://www.python.org/dev/peps/pep-0257/) recommends two ways of writing a docstring: Numpy style (https://sphinxcontrib-napoleon.readthedocs.io/en/latest/example_numpy.html) or Google style (https://sphinxcontrib-napoleon.readthedocs.io/en/latest/example_google.html).

The important thing is that this field contains enough information for the `help(multiple_numbers)` command to be useful. Indeed, `help()` returns the docstring, including for functions in modules.

It is also recommended that docstrings be defined as triple `"` rather than triple `'`.

II NumPy

The main interest of NumPy is :

- the creation and manipulation of **arrays**, as in C, C++, Fortran etc., with indices allowing the addressing of individual items, in a more user-friendly way than with `list`. This provides the equivalent of mathematical notions such as scalars (noted 0-D), vectors (1-D), matrices (2-D), cubes (3-D) and more generally tensors (n-D), as well as the algebraic operations that accompany them (scalar and vector products, diagonalization, etc.);
- benefit from vectorization, i.e. operations that apply to all elements of an array, without having to write explicit `for` loops;
- easy access to constants (π,...), functions ($\sqrt{}, \sin, \cos$, etc.) and basic numerical analysis routines (integration, derivation, solving differential equations, etc.).

It should be noted that it is only by using NumPy and vectorization that Python can compete in runtime with compiled languages, as some of NumPy's functions are pre-compiled in C and Fortran.

https://docs.scipy.org/doc/numpy/user/quickstart.html

https://numpy.org/devdocs/user/absolute_beginners.html

https://numpy.org/devdocs/user/quickstart.html

All recent distributions of Python contain a `numpy` module :

```
1 >>> import numpy
```

1. Array object

The manipulation of arrays is simpler and more efficient (placement in memory for example) than `list`.

The object created by NumPy is an `ndarray`. **nd** reminds us that the object can be n-D (of dimension *n*).

⚠ *Warning* : Vocabulary warning

There is extreme confusion about the terminology concerning NumPy arrays, partly because of differences between French, English, mathematical language and computer usage! Some of this confusion also stems from the mix of usage: "array" in computing whatever the form of the array, "vector/matrix/tensor" in mathematics and numerical physics which confuses matrix, tensor with 2-D array.

If an array element A is expressed as A_{ijkl} then it represents a tensor of *order* 4. The order is often called *rank* in documentations or online forums. This is a legacy of older versions of NumPy (the `rank()` function is replaced by the `ndim` attribute (or the `ndim()` function), and will soon be removed). But these arrays are often noted as 4-D, which could be interpreted as being of *dimension* 4. Now, the dimension of an array is the number of elements on each axis (row, column, depth, etc.), so the number of values that the indices i, j, k and l can take. NumPy uses the `shape` attribute (or the `shape()` function) to inform about the dimensions of the array.

We would thus distinguish between the *"dimensions of an array "* and the *"number of dimensions of an array "*...

In the specific case of a matrix (representing a second-order tensor, i.e. a 2-D array in NumPy), order and dimension are often confused: it is commonly said that a fourth-order matrix is a 4x4 matrix, i.e. with 4 rows and 4 columns. This is still a 2-D object (in the sense of arrays, and of order 2 in the sense of tensors).

Finally, the notion of `axis` is important in NumPy. It identifies in which direction of the array the requested operation is applied. In general, `axis=0` is the vertical (rows), `axis=1` is the horizontal (columns), `axis=2` is the depth. `axis=-1` is the last axis of the n-D array.

Definition :

A NumPy array (`ndarray`) has :

- a number of dimensions `ndim` (or returned by `numpy.ndim()`);
- axes (`axis=`) whose number is equal to `ndim`;
- with a number of elements per axis (number of rows/columns/etc.) given by `shape` or returned by `numpy.shape()`;
- so a total number of `size` elements (or returned by `numpy.size()`).

1.1. Manual construction of arrays

1-D numpy.array()

```
>>> import numpy
>>> a=numpy.array([0,1,2])
>>> print(a)
[0 1 2]
>>> type(a)
<class 'numpy.ndarray'>
>>> a.ndim  # nombre de dimensions du tableau
1           # donc vecteur
>>> a.shape # dimensions : nb elements sur chacun des axes
(3,)        #
```

The numbering of the indices starts at 0.

2-D numpy.array()

```
>>> b=numpy.array([[0,1,2],[3,4,5]])
>>> print(b)
[[0 1 2]
 [3 4 5]]
>>> b.ndim  # nb de dimensions
2
>>> b.shape  # nb de ligne et de colonnes
(2, 3)
>>> b.size   # nb total d'éléments
6
>>> print(b[0,0])  # origine en haut à gauche
0
>>> print(b[0,1])  # indices : ligne, colonne
1
```

Note the `list` of `list` as an argument to `numpy.array()` to set the values. It could also have been a `tuple` `list` or some other combination. But still a sequence.

`numpy.size()` returns the total number of elements in the array, which is just the product of the dimensions returned by `numpy.shape()`.

The first index identifies the row number, the second the column number, and so on for higher dimensions.

Warning : **0-D numpy.array()**

`a=numpy.array(42.)` creates a 0-D array, so mathematically a scalar. It is not a `float` like `a=42.` but an `ndarray` object.

1.2. Quick construction of regular arrays

There is little opportunity to explicitly fill in an array value by value.

Either the values are read from a file (e.g. experimental measurements) or the array is constructed from formulas or items from other arrays or functions.

With list comprehensions

In the case of large arrays with regular values, the nesting property of Python statements can be used:

```
>>> e=[[0 for i in range(3)] for j in range(4)]
>>> e
[[0, 0, 0], [0, 0, 0], [0, 0, 0], [0, 0, 0]]
>>> import numpy
>>> en=numpy.array(e)
>>> en
array([[0, 0, 0],
       [0, 0, 0],
       [0, 0, 0],
       [0, 0, 0]])
```

With list comprehensions

More compact expression:

```
>>> en2=numpy.array([[0 for i in range(3)] for j in range(4)])
>>> en2
array([[0, 0, 0],
       [0, 0, 0],
       [0, 0, 0],
       [0, 0, 0]])
```

Extra **:**

In the above example, instead of 0, we could have used an expression that depends on the indices i and j, or called a generating function.

```
>>> e=numpy.array([[i*j for i in range(3)] for j in range(4)])
```

```
array([[0, 0, 0],
       [0, 1, 2],
       [0, 2, 4],
       [0, 3, 6]])
```

Using methods/functions

`numpy.arange(start, end, step)` creates an evenly spaced 1-D array with values from `start` to `end` (excluded), by steps `step`, as would do the `range()` function seen before. But the result is an *ndarray*.

`numpy.reshape(nb_rows, nb_columns)` changes the number of dimensions (`ndim`) by setting the number of rows and columns.

```
>>> d=numpy.arange(15).reshape(3,5)
>>> d
array([[ 0,  1,  2,  3,  4],
       [ 5,  6,  7,  8,  9],
       [10, 11, 12, 13, 14]])
```

Data type

If we want to create an array with `float`, there are two cases:

- we want to represent integers in `float` form, so we specify the `dtype`:

```
>>> t=numpy.arange(1,3,dtype=float) # par défaut int
>>> print(t)
[ 1.  2.]
```

- if we want a sequence of `float` spaced by a fractional step, we use `numpy.linspace(start, end, nb_values)`, **but `end` is included** (contrary to the general rule of Python):

```
>>> t2=numpy.linspace(0,2,9)
>>> print(t2)
[ 0.    0.25  0.5   0.75  1.    1.25  1.5   1.75  2.  ]
```

Special cases

In algebra, one often has to create arrays with 0s and 1s.

`numpy.zeros(n)`, `numpy.ones(n)`: create an array of dimension `n` where `n` can be a `tuple` of dimensions

`numpy.identity(n)`: creates the `n x n` identity matrix (0 everywhere except the diagonal at 1).

```
>>> numpy.ones((3,5))
array([[ 1.,  1.,  1.,  1.,  1.],
       [ 1.,  1.,  1.,  1.,  1.],
       [ 1.,  1.,  1.,  1.,  1.]])
>>> numpy.zeros((3,5))
array([[ 0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.]])
>>> numpy.identity(3)
array([[1., 0., 0.],
       [0., 1., 0.],
       [0., 0., 1.]])
```

⊕ *Extra* : numpy.eye()

This function is more powerful than `numpy.identity()` because it allows to shift the diagonal and thus, for example, to make tri-diagonal matrices:

```
>>> numpy.eye(3,k=-1) + numpy.eye(3)*2 + numpy.eye(3,k=1)*3
array([[ 2.,  3.,  0.],
       [ 1.,  2.,  3.],
       [ 0.,  1.,  2.]])

```

⊕ *Extra* : numpy.diag()

This function also creates diagonal matrices, but from a list of values on the diagonal:

```
>>> b=numpy.diag([1,2,3,4])
>>> b
array([[1, 0, 0, 0],
       [0, 2, 0, 0],
       [0, 0, 3, 0],
       [0, 0, 0, 4]])
```

⊕ *Extra* : numpy.empty() and fill() method

`fill()` is a method (not a function, see chapter on object-oriented programming) and therefore applies to the variable containing the `ndarray`. This array has usually been created empty using `numpy.empty(shape)`:

```
>>> a=numpy.empty((2,2))
>>> a                                          # not really empty BE CAREFUL
array([[1.11387892e-311, 4.61769168e-311],
       [2.12199579e-314, 3.55583557e-104]])
>>> a.fill(1)
>>> a
array([[1., 1.],
       [1., 1.]])
```

In the style of (like)

It is common to want to create an array whose properties (`ndim`, `shape`, `dtype`) are identical to another.

Some functions seen above take the suffix "_like", the first argument being the array whose properties you want to copy.

This is the case of:

- `numpy.zeros_like()`
- `numpy.ones_like()`
- `numpy.empty_like()`
- `numpy.full_like()`

```
>>> x = numpy.arange(6,dtype=float).reshape((2, 3))
>>> x
array([[0., 1., 2.],
       [3., 4., 5.]])
>>> numpy.zeros_like(x)
array([[0., 0., 0.],
       [0., 0., 0.]])
```

2. Basic operations on arrays

2.1. Printing

`numpy` only displays the corners if the arrays are too large.

```
>>> print(numpy.linspace(0,1,1000000).reshape(100,100,100))  # matrice 3D, donc cube!
[[[  0.00000000e+00   1.00000100e-06   2.00000200e-06 ...,   9.70000970e-05
     9.80000980e-05   9.90000990e-05]
  [  1.00000100e-04   1.01000101e-04   1.02000102e-04 ...,   1.97000197e-04
     1.98000198e-04   1.99000199e-04]
  [  2.00000200e-04   2.01000201e-04   2.02000202e-04 ...,   2.97000297e-04
     2.98000298e-04   2.99000299e-04]
  ...,
  [  9.70000970e-03   9.70100970e-03   9.70200970e-03 ...,   9.79700980e-03
     9.79800980e-03   9.79900980e-03]
  [  9.80000980e-03   9.80100980e-03   9.80200980e-03 ...,   9.89700990e-03
     9.89800990e-03   9.89900990e-03]
  [  9.90000990e-03   9.90100990e-03   9.90200990e-03 ...,   9.99701000e-03
     9.99801000e-03   9.99901000e-03]]

 [[  1.00000100e-02   1.00010100e-02   1.00020100e-02 ...,   1.00970101e-02
     1.00980101e-02   1.00990101e-02]
  [  1.01000101e-02   1.01010101e-02   1.01020101e-02 ...,   1.01970102e-02
     1.01980102e-02   1.01990102e-02]
```

2.2. Size and dimension changes

Slicing

As with `list`, it is possible to specify a subset of the array, either for printing or for partial extraction (view).

The syntax is exactly the same as for lists: `[start_index: end_index: step]`. As is the operation of negative indices:

```
>>> b
array([[1, 0, 0, 0],
       [0, 2, 0, 0],
       [0, 0, 3, 0],
       [0, 0, 0, 4]])
>>> print(b[0:2,:])
```

```
[[1 0 0 0]
 [0 2 0 0]]
>>> print(b[0:2,::-1])
[[0 0 0 1]
 [0 0 2 0]]
```

⚠ *Warning* : Unlike lists

- it is impossible to insert or delete items of arrays with the slicing method as this alters the dimensions ;
- `a=b[:,:]` does not perform a deep copy (`a=numpy.array(b)` does). A slicing operation creates a view on the original array, which is just a way of accessing array data.

Concatenation (numpy.concatenate())

It is common to have to extend a vector (1-D array) or add a row or column to a 2-D or 3-D array.

Simple case of two vectors (`ndim=1`). This involves extending the vector `arr1` with `arr2`:

```
>>> import numpy as np
>>> arr1 = np.array([1, 2, 3])
>>> arr2 = np.array([4, 5, 6])
>>> arr = np.concatenate((arr1, arr2)) # tuple as argument !!!
>>> print(arr)
[1 2 3 4 5 6]
```

In this case, the same result can be obtained with the function `numpy.hstack((arr1,arr2))`. See below.

ndim > 1

In the case of arrays with a dimension greater than 1, it is necessary to specify the axis along which the concatenation should take place.

`axis=0` designates the vertical axis; therefore rows will be added. The second array is added below the first.

`axis=1` designates the horizontal axis; columns will be added. The second array is added to the right of the first.

`axis=None` renders the arrays 1-D (*flattening*) before concatenation, which allows you to do `numpy.reshape()` afterwards.

```
>>> import numpy as np
>>> arr1 = np.array([[1, 2], [3, 4]])
>>> arr2 = np.array([[5, 6], [7, 8]])
>>> arr = np.concatenate((arr1, arr2), axis=1)
>>> print(arr)
[[1 2 5 6]
 [3 4 7 8]]
```

A more complex case: we want to concatenate a vector (1-D array) to a matrix (2-D), i.e. add a row or a column.

But `numpy.concatenate()` only works with arrays of the same number of dimensions.

```
>>> a = np.array([[1, 2], [3, 4]])
>>> print("a=",a)
a=
[[1 2]
 [3 4]]

>>> b = np.array([[5, 6]])  # notez les doubles [[ ]], donc ndim=2
>>> print("b=",b)
b=
[[5 6]]

>>> c = np.array([5, 6]) # simple [ ] donc ndim=1
```

```
>>> print("c=",c)
c=
[5 6]

>> print('ndim a,b,c =',a.ndim,b.ndim,c.ndim)
ndim a,b,c = 2 2 1

>>> print('concatenation axis=0:',np.concatenate((a, b), axis=0))
concatenation axis=0:
[[1 2]
 [3 4]
 [5 6]]

>>> print("transposition(b)=",b.T) #transformation tableau à 1 ligne en tableau à 1 colonne
transposition(b)= [[5] [6]]

>>> print('concatenation axis=1:',np.concatenate((a, b.T), axis=1))
concatenation axis=1:
[[1 2 5]
 [3 4 6]]

>>> print('autre syntaxe')
>>> print(np.concatenate([a, b], axis=None))
autre syntaxe
[1 2 3 4 5 6]

>>> print('cas faux : concatenation entre tableaux de ndim différents')
>>> print(np.concatenate((a, c), axis=0))
cas faux : concatenation entre tableaux de ndim différents
Traceback (most recent call last):

...

  File "C:/Users/Hervé Wozniak/Documents/Mandibule/numpy.concatenate.py", line 28, in <module>
    print(np.concatenate((a, c), axis=0))

ValueError: all the input arrays must have same number of dimensions
```

Therefore, a mathematical vector can be represented by a 2-D array in NumPy, only for practical reasons.

numpy.newaxis

To concatenate a vector and a matrix it is therefore imperative to convert the vector into a 1 row (or 1 column) matrix.

The `numpy.newaxis` command adds a dimension to an array.

As `c` is a 1-D array in the previous example, `c[numpy.newaxis,:]` will return a 1 row 2-D array while `c[:,numpy.newaxis]` will return a 1 column 2-D array.

Stacking (numpy.stack, numpy.hstack, numpy.vstack)

Stacking (`numpy.stack`) is similar to concatenation but along a new axis. Thus, two 1-D arrays with `shape=(2,)` stacked together will form a 2x2 2-D array (`shape=(2,2)`).

If the direction of stacking is known in advance, simple functions `numpy.hstack()` (horizontal), `numpy.vstack()` (vertical), `numpy.dstack()` (depth for 3-D arrays) can be used, but the function `numpy.stack()` is more general.

One can also stack N vectors of dimension M to form a matrix (N,M) with `numpy.column_stack()`.

In some special cases, the result is identical to `numpy.concatenate()`.

Warning :

The first argument of `numpy.concatenate()` and `numpy.stack()` is a sequence (`list` or `tuple`) that contains the list of arrays to be concatenated.

A common mistake is to forget the round brackets (for a `tuple`) or the square brackets (for a `list`)

Extra :

There are functions for opposite operations (`numpy.split()`, etc.) which will not be discussed here.

https://numpy.org/doc/stable/reference/routines.array-manipulation.html

Repetition numpy.tile()

If one wants to repeat a row(s) or column(s) of an array to increase its size, `numpy.tile(array,reps)` can be used by specifying the number of instances (original + copies) along each axis.

`reps` is therefore a `tuple` as soon as `ndim` > 1. If `reps` has more items than `ndim`, then a new axis is created. If `reps` has fewer items than `ndim`, only the last axes are replicated (see example below).

`numpy.tile(array,1)` returns the original array (neutral operation) if array is 1-D (`reps=(1,1)` if 2-D, etc.).

```
# Cas 1-D

>>> a = np.array([0, 1, 2])
>>> np.tile(a, 2)
array([0, 1, 2, 0, 1, 2])

>>> np.tile(a,(1,2)) # un nouvel axe est créé (le résultat est 2-D avec [[ ]])
array([[0, 1, 2, 0, 1, 2]])

>>> np.tile(a, (2, 2)) # un nouvel axe est créé avec deux copies de a verticalement et horizontalement
array([[0, 1, 2, 0, 1, 2],
       [0, 1, 2, 0, 1, 2]])

>>> np.tile(a,(4,1)) # servira plus loin (broadcasting)
array([[0, 1, 2],
       [0, 1, 2],
       [0, 1, 2],
       [0, 1, 2]])

>>> np.tile(a, (2, 1, 2)) # deux nouveaux axes sont créés ndim=3, shape=(2,1,6)
>array([[[0, 1, 2, 0, 1, 2]],

       [[0, 1, 2, 0, 1, 2]]])

# Cas 2-D

>>> b = np.array([[1, 2], [3, 4]])
>>> np.tile(b, 2)  # equivalent à (1,2) donc pas de réplication verticale et une seule horizontale
array([[1, 2, 1, 2],
       [3, 4, 3, 4]])

>>> np.tile(b, (2, 1)) # une réplication verticale mais pas horizontale
array([[1, 2],
       [3, 4],
       [1, 2],
       [3, 4]])
```

3. Using arrays

3.1. Basic operations

Implicit loops

In classical non-vector languages, it is necessary to nest loops on the indices of arrays to apply operations to them.

This is also how algorithms in numerical analysis are described.

If you write the same thing rigorously in Python, you get programs that are much slower than their counterparts compiled in C or Fortran because Python is interpreted.

It is therefore imperative to train oneself to **avoid the use of loops** and to do this, one must understand the effects of functions and operations on NumPy arrays in order to take full advantage of the acceleration due to vectorization.

a) Element-wise arithmetic and logical operations

The operators +, −, *, **, / have an action "element by element" (element-wise arithmetics) on arrays **of the same size**, so you do not need to use Python loops:

```
>>> a = numpy.array( [20,30,40,50] )
>>> b = numpy.arange( 4 )
>>> b
array([0, 1, 2, 3])
>>> c = a-b
>>> c
array([20, 29, 38, 47])
>>> b**2
array([0, 1, 4, 9])
>>> 10*numpy.sin(a)
array([ 9.12945251, -9.88031624,  7.4511316 , -2.62374854])
```

We'll see later on the general case of functions that apply to each item of an array.

Comparison operations (==, <= etc.) are also element-wise:

```
>>> a<35
array([ True, True, False, False], dtype=bool)
>>> b
array([[1, 1],
       [2, 2],
       [3, 3]])
>>> c
array([[1, 1],
       [2, 2],
       [3, 3]])
>>> b==c
array([[ True,  True],
       [ True,  True],
       [ True,  True]])
>>> numpy.all(b==c) # True si tous les élements sont vrais
True
```

Comparing two boolean NumPy arrays cannot be bone with basic Python statement (and, or, not) because they are not element-wise operators.

Use `&`, `|` and `~` instead :

```
>>> a=np.array([True,False,False])
>>> b=np.array([True,True,False])
>>> a & b
array([ True, False, False])
>>> a | b
array([ True,  True, False])
>>> ~ a
array([False,  True,  True])
```

Extra : Particularity of operator ~

The operator ~ inverts all the bits that make up the value of a variable:

- When applied to an integer variable, `~x` returns the value `-x-1` ;
- It cannot be applied to a `float` or to composite types (error message);
- In the particular case of a boolean variable, the operation is equivalent to taking the negation. If `x=True`, then `~x` is `False`, as with the `not` operator.

But `not` **does not apply** to an array, while ~ returns an array where all elements are opposite.

```
>>> l=np.array((True,False))
>>> print(l)
[ True False]
>>> print(~l)
[False  True]
>>> print(not l)
Traceback (most recent call last):

  File "<ipython-input-52-aaec0ba91c8d>", line 1, in <module>
    print(not l)

ValueError: The truth value of an array with more than one element is ambiguous. Use
a.any() or a.all()
```

b) Basic reductions

- `numpy.sum()` : calculates the sum of all elements of the array passed as argument;
- `numpy.sum(axis=)` : same as above but sum in column (`axis=0`) or in row (`axis=1`);
- `numpy.min()`, `numpy.max()` : search for the smallest/largest item;
- `numpy.min(axis=)`, `numpy.max(axis=)` : returns the min or max value but only on one axis;
- `numpy.argmin()`, `numpy.argmax()` : returns the position (index) of the min or max;
- `numpy.cumsum(axis=)` : cumulative sum;
- `numpy.mean(axis=)` : average of the array elements, optionally over the specified axis;
- `numpy.trace()` : trace of a matrix (see example).

```
>>> a=numpy.tile([1,2,3],(3,1))
>>> a
array([[1, 2, 3],
       [1, 2, 3],
       [1, 2, 3]])

>>> numpy.trace(a)
6
```

Boolean operations

Can be used for array comparisons

```
>>> numpy.all([True, True, False])
False
>>> numpy.any([True, True, False])
True
```

```
>>> a = numpy.zeros((100, 100))
>>> numpy.any(a != 0)
False
>>> numpy.all(a == a)
True
>>> a = numpy.array([1, 2, 3, 2])
>>> b = numpy.array([2, 2, 3, 2])
>>> c = numpy.array([6, 4, 4, 5])
>>> ((a <= b) & (b <= c)).all()
True
```

c) Algebraic operations on arrays

In addition to the element-wise operations, it is possible to apply some algebraic operations on arrays:

- transpose: `numpy.transpose(array)` or `array.T` (in this syntax `array` is not modified)
- scalar product: `numpy.dot(array1,array2)`
- vector product : `numpy.cross(array1,array2)`
- tensor product: `numpy.tensordot(array1,array2)`
- "outer" (dyadic) product: `numpy.outer()` (takes in two vectors and returns a second order tensor)
- Hadamard product: `array1*array2`
- `numpy.einsum()` does most of the above in Einstein notation;

etc.

A more complete list can be found here: https://docs.scipy.org/doc/numpy/user/quickstart.html#functions-and-methods-overview.

Warning **:**

The result of these operations often depends on the `ndim` of the arrays passed as arguments. For example, in the case of `numpy.dot()`:

- if `array1` and `array2` are 1-D, it is the normal *scalar product* (for the scalar product with the complex conjugate of array1, use `numpy.vdot()`);
- if `array1` and `array2` are 2-D, this is *matrix multiplication*, but `numpy.matmul()` or `array1 @ array2` are preferable. `numpy.tensordot()` can also do this;
- if `array1` or `array2` is a scalar, it is equivalent to a simple multiplication `array1 * array2` or `numpy.multiply(array1, array2)`;

etc. (see the online documentation for n-D cases).

The `numpy.dot()` function is therefore both powerful (because it adapts to the number of dimensions of the arguments) but also a source of error if one is not sure of the dimensionality of the arguments.

This is also the case for `numpy.tensordot()` which does much more than just tensor calculation according to the number of dimensions of the arrays and arguments of the function.

Before using these functions, one should always understand the mathematics and test their use on analytical cases.

numpy.linalg

The sub-module `numpy.linalg` implements basic linear algebra, such as solving linear systems, singular value decomposition, etc.

However, it is not guaranteed to be compiled using efficient routines, and thus the use of scipy.linalg is recommended if computing time is an issue.

numpy.linalg.inv(array) : returns the inverse matrix of array (which is at least ndim=2 and square)

```
>>> from numpy.linalg import inv
>>> a = numpy.array([[1, 3, 3],
                     [1, 4, 3],
                     [1, 3, 4]])
>>> inv(a)
array([[ 7., -3., -3.],
       [-1.,  1.,  0.],
       [-1.,  0.,  1.]])
```

numpy.linalg.det(array) : returns the determinant of array

```
>>> from numpy.linalg import det
>>> a = numpy.array([[1, 2],
                    [3, 4]])
>>> det(a)
-2.0
```

numpy.linalg.solve() : solve linear equations

For solving the system of linear equations $3x_0 + x_1 = 9$ and $x_0 + 2x_1 = 8$:

```
>>> a = numpy.array([[3,1], [1,2]])
>>> b = numpy.array([9,8])
>>> x = numpy.linalg.solve(a, b)
>>> x
array([ 2.,  3.])
```

To check that the solution is correct (`numpy.allclose()`):

```
>>> numpy.allclose(numpy.dot(a, x), b)
True
```

numpy.linalg.eig() : eigenvalues and eigenvectors

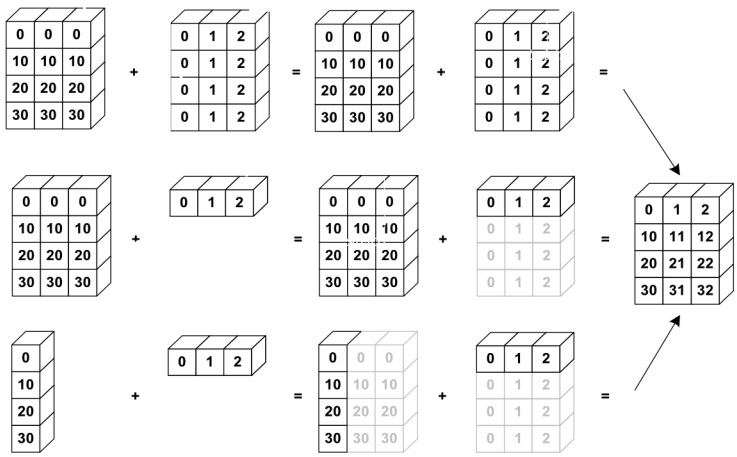
```
>>> from numpy.linalg import eig
>>> A = numpy.array([[ 1, 1, -2 ], [-1, 2, 1], [0, 1, -1]])
>>> A
array([[ 1,  1, -2],
       [-1,  2,  1],
       [ 0,  1, -1]])
>>> D, V = eig(A)
>>> D
array([ 2.,  1., -1.])
>>> V
array([[  3.01511345e-01,  -8.01783726e-01,   7.07106781e-01],
       [  9.04534034e-01,  -5.34522484e-01,  -3.52543159e-16],
       [  3.01511345e-01,  -2.67261242e-01,   7.07106781e-01]])
```

The columns of V are the eigenvectors of A associated with the eigenvalues that appear in D.

3.2. Broadcasting

Basic operations on numpy arrays (addition, etc.) are element-wise. This works on arrays of the same size.

Nevertheless, It's also possible to do operations on arrays of different sizes (`shape`) if NumPy can transform these arrays so that they all have the same `shape`: this conversion is called *broadcasting*.



For example, we want to add a constant vector to each row of a matrix. A first (bad) solution would be:

```python
import numpy as np

x = np.array([[1,2,3], [4,5,6], [7,8,9], [10, 11, 12]])
v = np.array([1, 0, 1]) # vecteur à ajouter à chaque ligne de x
y = np.empty_like(x)   # tableau vide de même shape que x pour recueillir le résultat

# additionne le vecteur v à chaque ligne de x avec une boucle explicite :
for i in range(4):
    y[i, :] = x[i, :] + v

print(y)

# [[ 2  2  4]
#  [ 5  5  7]
#  [ 8  8 10]
#  [11 11 13]]
```

As soon as the matrix `x` becomes very large (several million rows), the explicit loop is too slow (a compiled language such as C or Fortran would be preferable here).

However, we note that the operation of multiplying `v` by each row of `x` is equivalent to forming a matrix `vv` by replicating `v` vertically and then summing `x` and `vv` item by item (element-wise addition). This would give this program:

```python
import numpy as np

x = np.array([[1,2,3], [4,5,6], [7,8,9], [10, 11, 12]])
v = np.array([1, 0, 1])
vv = np.tile(v, (4, 1))   # Stack 4 copies de v verticalement
print(vv)
# "[[1 0 1]
#   [1 0 1]
#   [1 0 1]
#   [1 0 1]]"
y = x + vv  # additionne x et vv terme à terme
print(y)
```

```
# [[ 2  2  4]
#  [ 5  5  7]
#  [ 8  8 10]
#  [11 11 13]]

```

The *broadcasting* allows this operation to be carried out without even having to duplicate `v` by stacking. Indeed:

```
import numpy as np

x = np.array([[1,2,3], [4,5,6], [7,8,9], [10, 11, 12]])
v = np.array([1, 0, 1])
y = x + v  # Addition de v à chaque ligne de x par broadcasting
print(y)
# [[ 2  2  4]
#  [ 5  5  7]
#  [ 8  8 10]
#  [11 11 13]]
```

The line `y = x + v` does not produce an error when `x` is of `shape (4,3)` and `v` of `shape (3,)`. So `v` has the right dimension to be added to each of the lines because it itself represents a line.

The functions that enable broadcasting are called universal functions (*ufunc*). All these functions work term by term. They include all the classic arithmetic operations seen above.

In all, there are about sixty of them!

https://numpy.org/doc/stable/reference/ufuncs.html#available-ufuncs

Example: calculation of the barycentre of a large number of massive points

Let `nb_part` points, each of random mass between 0 and 1, of position `x, y, z` also random between 0 and 1. The distributions being uniform, we expect a total mass of 0.5*`nb_part` and a barycentre placed in (0.5,0.5,0.5)

```
# -*- coding: utf-8 -*-
"""
Created on Thu Jul  8 11:55:51 2021

@author: hwozniak
"""
import numpy as np
import time

''' exemple de vectorisation efficace
0. génération d'un grand volume de données
-----------------------------------------
'''
nb_part=1000000
ndim=3
# pour éventuellement ne sélectionner que quelques points
partidx=np.arange(nb_part) # mais ici on prend tout

mass=np.random.random(nb_part)
pos=np.random.random(ndim*nb_part).reshape(ndim,nb_part)

'''
1.a calcul centre de masse façon code non vectorisé (indices)
-------------------------------------------------------------
'''

t0=time.process_time()

cmp=np.zeros(ndim)
```

```
tmtot=0.
for i in partidx: tmtot  += mass[i]
for i in partidx:
    for j in range(ndim):
        cmp[j] += mass[i]*pos[j,i]
''' il faut intégrer le calcul de tmtot à la boucle sur partidx mais alors
    on ne verrait plus le gain sur l'ordre de la double boucle '''
#    tmtot  += mass[i]
for j in range(ndim):
    cmp[j]=cmp[j]/tmtot

t1=time.process_time()
print(' centre de masse=', cmp,'\n masse totale=', tmtot)
print(' temps CPU=', t1-t0)

'''
1.b boucles dans le bon ordre (second indice contigu en mémoire)
-----------------------------------------------------------------
'''

t0=time.process_time()

cmp=np.zeros(ndim)
tmtot=0.
for i in partidx: tmtot  += mass[i]
for j in range(ndim):
    for i in partidx:
        cmp[j] += mass[i]*pos[j,i]
for j in range(ndim):
    cmp[j]=cmp[j]/tmtot

t1=time.process_time()
print(' centre de masse=', cmp,'\n masse totale=', tmtot)
print(' temps CPU=', t1-t0)



'''
2. calcul en vectoriel
----------------------
'''

t0=time.process_time()

cmp=np.sum(mass[partidx]*pos[:,partidx],axis=1)
tmtot=np.sum(mass[partidx])
cmp=cmp/tmtot

t1=time.process_time()
print(' centre de masse=', cmp,'\n masse totale=', tmtot)
print(' temps CPU=', t1-t0)
```

Result

centre de masse= [0.49930523 0.49981319 0.50018287]

masse totale= 499791.39698681707

temps CPU= 2.6875

centre de masse= [0.49930523 0.49981319 0.50018287]

masse totale= 499791.39698681707

temps CPU= 2.640625

centre de masse= [0.49930523 0.49981319 0.50018287]

masse totale= 499791.39698682405

temps CPU= 0.046875

The following conclusions can be drawn:

1. the order of the loops is important because it affects the organization of the arrays in memory. The last index is the one that varies the fastest, i.e. it allows contiguous access in memory. The gain is a factor of 1.018 in computation time for the given example;
2. the fast calculation is given by the use of NumPy functions. A gain of at least a factor of 50!

***Extra* : Further information on broadcasting**

https://numpy.org/devdocs/user/theory.broadcasting.html

3.3. Working with indexes

There are many situations in numerical simulations where it is essential to know how to manipulate indices. In general, it is a question of knowing how to select elements of an array according to a Boolean condition and, possibly, to apply an operation on these selected elements. One may also have to find intersections between two lists of elements or to join these two lists.

Combined with broadcasting, some methods and functions complete the language to avoid having to write loops and nested loops.

a) Masking

Working with a selection of array indices

We have already seen the slicing mechanism. This mechanism allows you to select a single contiguous slice.

If you want to address a discontinuous selection, you use a Boolean array of the same dimension as the array you are working on.

```
import numpy as np

nb_part=100 # number of particles in arrays
ndim=3      # R^3 space

'''
random draw in a [0, 1[ box
'''
position=np.random.random(ndim*nb_part).reshape(ndim,nb_part)

print('max(position)=',np.max(position)) # less than 1 a priori

'''
selection of particles in a [0, 0.5[ box.
use of & operator
formally : (X < 0.5) AND (Y < 0.5) AND (Z < 0.5)
'''

in_mesh=(position[0,:]<0.5) & (position[1,:]<0.5) & (position[2,:]<0.5)

'''
in_mesh is of size nb_part
contains True for particles inside the box
'''
print('size(in_mesh)=',in_mesh.size)


```

```
'''
number of points inside the mesh; a priori nb_part/8 particles for a perfect uniform filling.
'''
print('points inside mesh=',in_mesh[in_mesh].size)

'''
check the result
'''
print('max(position[selection])=',np.max(position[:,in_mesh]))

```

Fundamental : Masking

In the previous example, `in_mesh` is called a *mask*.

When masking mecanism is used, the operations only apply to the array elements for which the corresponding mask element is True.

```
>>> a=np.array([0,1,2])     # must be a NumPy array
>>> mask=[True,False,False] # can be an NumPy array or a Python list
>>> a[mask]+=1
>>> print(a)
array([1, 1, 2])
```

Warning : Side effect

Let us complete the example of the particles with the computation of the angular momentum, only for the particles inside the box.

```
'''
computes angular momentum only on particles inside the box
'''
mass=np.random.random(nb_part)
vitess=np.random.random(ndim*nb_part).reshape(ndim,nb_part)

J=np.cross(mass[in_mesh]*position[:,in_mesh],vitess[:,in_mesh],axisa=0,axisb=0).T

print('shape of J =', J.shape)

```

The result can be : *"shape of J = (3, 9) "*, i.e. the second dimension is equal to the number of particles inside the box. If we want the array `J` to be the size `nb_part`, we must first create it and apply the mask.

```
J=np.zeros_like(position)

J[:,in_mesh]=np.cross(mass[in_mesh]*position[:,in_mesh],vitess[:,in_mesh],axisa=0,axisb=0).T

print('shape of J =', J.shape)
```

b) Working with sets of indices

Instead of working with a mask, we may have to work with the indices of the array elements where certain operations will be applied. So, first we need to get the indices of the mask elements with value True. Then, using the indices, we can perform any necessary operation.

We continue with the previous example.

Retrieving the index of the elements of the mask `in_mesh` that are `True` is done with the function `numpy.flatnonzero()`.

This function returns indices that are non-zero in the flattened version of the array in argument.

```
>>> idx_inmesh=np.flatnonzero(in_mesh)
>>> print(idx_inmesh)
array([ 3,  7,  9, 17, 19, 22, 41, 55, 61, 68, 74, 80, 83, 87, 88, 91, 95,
       97, 99], dtype=int64)
>>> in_mesh[idx_inmesh]
array([ True,  True,  True,  True,  True,  True,  True,  True,  True,
        True,  True,  True,  True,  True,  True,  True,  True,  True,
        True])
```

`idx_inmesh` is now an array containing a list of indices which can be used to adress any array of size `nb_part` (`position`, `vitess`, `in_mesh`, etc.).

In addition, one may want the indices of the particles outside the box. It is then sufficient to apply the Boolean rules.

```
>>> idx_outmesh=np.flatnonzero(~in_mesh) # not operator for arrays
>>> idx_outmesh.size
81
```

Having two sets of indices, those corresponding to the points outside the box (`idx_outmesh`) and those of the points inside (`idx_inmesh`), we can use the functions working on *sets* to unite again these two lists. We must then find the full list of indices.

```
>>> total_idx=np.union1d(idx,idx_out_mesh)
>>> total_idx.size
100
```

Note that `total_idx` est ordered by the `numpy.union1d()` function.

One can imagine that the *intersection* between `idx_inmesh` and `idx_outmesh` is an empty set. We can check that with the `numpy.intersect1d()` function:

```
>>> empty_idx=np.intersect1d(idx_inmesh,idx_outmesh)
>>> empty_idx
array([], dtype=int64)
```

Extra : Set operations

Functions performing operations on sets (`union1d`, `intersect1d`, etc.) are very useful for working on discrete meshes. For example, to select grid elements within a volume or, in general, a region of interest (ROI).

See https://numpy.org/doc/stable/reference/routines.set.html[8]

3.4. Other functions

The other powerful feature of NumPy is that it provides already implemented, sometimes complicated, functions:

- trigonometric (`sin()`, `cos()`, `arctan2()`, ...) and conversions (radians, degrees, ...)
- hyperbolic (`sinh()`, `cosh()`, ...)
- exponentials and logarithms (`exp()`, `log()`, `log10()`, ...)
- special (`i0()`: modified Bessel function of the first kind, order 0; `sinc(x)`: the famous $\frac{sin(x)}{x}$; etc.)

NumPy applies all the elementary functions `exp`, `log`, `sin`, etc. on each individual item of the array (element-wise operation).

These functions are *universal* (*ufunc*): they allow broadcasting since they apply to each item of the array.

In this sense, they are different from the functions of the same name in the `math` module which only apply to the native Python type (`int`, `float` etc.).

8. List of set routines

And:

- `sign(a)`: returns the sign of the values in `a` ;
- `floor(x)` : gives as output the greatest integer less than or equal to `x` ;
- `ceil(x)`: the least integer greater than or equal to `x`;
- `gradient(a)`: returns the gradient of an array;
- `trapz(y)`: integrates along the axis with the trapezoid method;
- `interp(x,xp,fp)`: interpolation;
- `convolve(a,v)`: convolution product;
- etc.

For arrays of complex numbers: `real(), imag(), abs(), angle(),conj(),vdot()`, etc.

Extra **:**

https://numpy.org/doc/stable/reference/routines.math.html

Randomness, probability and statistics

There is a large subset of functions for this area. See https://docs.scipy.org/doc/numpy/reference/routines.random.html for more details. For more advanced statistics, it is better to use the SciPy module, which we will not discuss in detail.

A function that can replace the one in the `random` package has the same name:

```
>>> numpy.random.random()
0.5540884899329033
>>> numpy.random.random(3)
array([ 0.86431861,  0.88519197,  0.30663316])
>>> numpy.random.random((2,3))
array([[ 0.66265691,  0.39385577,  0.09319192],
       [ 0.43483474,  0.42859904,  0.79189574]])
```

III Matplotlib

Matplotlib is a Python package that allows you to make figures and print them in different formats.

Since the emergence of *Big Data* (massive data science), many graphical modules are integrated into data analysis packages (Pandas, Plotly, Altair, Seaborn, etc. see http://pythonplot.com). These modules are often based on Matplotlib. For the physicist, and common use, Matplotlib is sufficient.

We will mainly look at the use of Matplotlib in its procedural form. There is a more powerful use in object-oriented form.

Most figures drawn with Matplotlib have a rudimentary appearance. By default, Matplotlib produces simple figures, which contain information without exceptionally nice formatting. However, given a little time, it is quite possible to obtain figures of a professional quality close to those created by commercial software. Below is an example published by Swagat Saurav Mishra on Twitter :

https://twitter.com/Swagat_arhsiM/status/1563675392451526656?t=8V_yE2bqrbWXIiEFyIvLjA&s=09

1. Basics

It starts with:

```
1 import matplotlib
```

But the most interesting module is `pyplot`, very often imported under the alias `plt`:

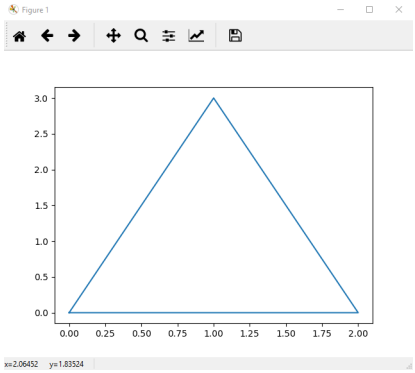
```
1 import matplotlib.pyplot as plt
```

This module allows to find commands very close to those of another language: MATLAB.

Main functions: figure(), plot(), show() et close()

- The `figure()` function is used to initialize the graph. Without this statement, Python will initialize a new figure with default characteristics. So using the `figure()` function allows you to start a clean environment, possibly tailored to your needs. It is possible to pass a window name (type `str`) or a number as an argument to `figure()`. In this case, if the same program is run several times without closing the window, the graph is plotted as many times with different colours;
- `plot()` is probably the most used function in scientific visualization. It allows you to plot two lists, or two NumPy arrays, which contain the abscissa and ordinate;
- finally, `show()` requests the display of the figure. It is used once all the operations for constructing the figure have been completed. Thus, many plotting actions can be accumulated before displaying.
- `show()` does not close the figure. It is still possible to add elements to it later until the `close()` instruction;
- `close()` is never used in interactive mode as it destroys the window. In programming, it is a good habit to place a `close()` before the call to `figure()` in order to recreate a new figure and not to take the risk of rewriting in an already existing figure of the same name.

```
1 >>> import matplotlib.pyplot as plt
2 >>> x=[0,2,1,0]
3 >>> y=[0,0,3,0]
4 >>> plt.figure()
5 <matplotlib.figure.Figure object at 0x00000280108B9710>
6 >>> plt.plot(x,y)
7 [<matplotlib.lines.Line2D object at 0x0000028014BC1550>]
8 >>> plt.show()
```



⚠ *Warning* :

In some built-in environments, such as *Pyzo* and *Spyder*, it is not necessary to place the `show()` statement. If you have to use your program outside such environments, it is better not to forget the `show()` statement.

So, get into the habit...

⊕ *Extra* :

Other graphical backends than the default one ('tkagg') do not provide for the use of the `show()` function (e.g. 'agg') but this feature is outside the scope of this introductory course to Matplotlib, focused on the module pyplot.

The figure appears in a dedicated window, equipped with several menus. It is then possible to save it or to modify it interactively (zoom for example).

If the previous example is restarted, the same figure is redrawn in a new graphics window. It is therefore necessary to close the window (`close()`) at the beginning of the program.

2. Drawing curves y=f(x)

Combination with NumPy

The example below illustrates the combined use of NumPy and Matplotlib:

```
1 import matplotlib.pyplot as plt
2 import numpy as np
3 # 50 points entre 0 et 2*Pi
4 x=np.linspace(0,2*np.pi,50)
5 y=np.cos(x)
6 plt.figure("Tracé d'un cosinus")
7 plt.plot(x,y)
8 plt.show()
```

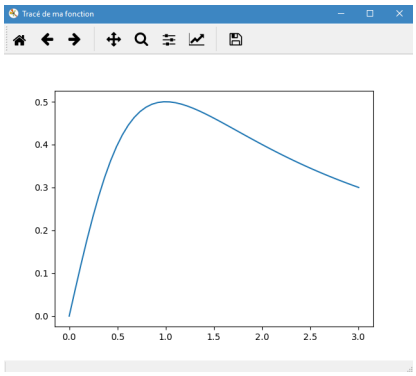
The 50 points give the illusion of a continuous curve. The number of points must obviously be adapted to the resolution of the graph. There is no need to oversample, i.e. to draw several points in the same pixel of the image or screen!

From now on we will systematically use the combination NumPy + Matplotlib, which is the most common usage among physicists using Python.

Drawing your own function

```
1 def ma_fonction(x):
2     y=x/(1+x**2)
3     return y
4
5 import matplotlib.pyplot as plt
6 import numpy as np
7 x=np.linspace(0,3,50)
8 y=ma_fonction(x)
9 plt.figure("Tracé de ma fonction")
```

```
plt.plot(x,y)
plt.show()
```



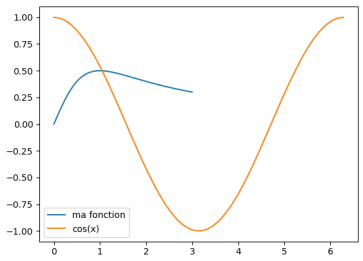
Note :

The function `my_function()` acts as a generator despite the `return` statement instead of `yield`. This behaviour is possible since Python3. It is an indirect application of NumPy broadcasting (applying a function to a vector returns a vector, but here the function does not belong to the NumPy module).

Overplotting curves

If the graph remains 'open', several curves can be drawn in the same window. The interval of the axes is calculated by Matplotlib with the instruction `show()`. The second curve is drawn in a different colour automatically.

```
def ma_fonction(x):
    y=x/(1+x**2)
    return y

import matplotlib.pyplot as plt
import numpy as np
# 50 points entre 0 et 2*Pi
x=np.linspace(0,3,50)
y=ma_fonction(x)
plt.figure("Tracé de mes fonctions")
plt.plot(x,y,label="ma fonction")
x=np.linspace(0,2*np.pi,50)
plt.plot(x,np.cos(x),label="cos(x)")
plt.legend()
plt.show()
```



Extra :

`plot(x,y,label="texte")` allows you to prepare a legend that will appear in a title block automatically created and placed by the `legend()` function.

Managing colours and plot type

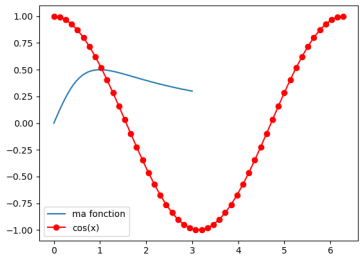
The 3rd argument of the `plot()` function allows you to manage the colour and the style (connected or not, big dots...).

https://matplotlib.org/stable/api/_as_gen/matplotlib.pyplot.plot.html#matplotlib.pyplot.plot

Colour	String (str)	Marker	String (str)
blue	'b'	**point**	'.'
red	'r'	**circle**	'o'
black	'k'	**triangle up / down**	'^' / 'v'
green	'g'	**Pixel**	','
cyan	'c'	**square**	's'
magenta	'm'	**solid line style**	'-'
yellow	'y'	**dashed line style**	'--'

The options can be combined: "ro-" means that the curve should be drawn in red, with large points connected by segments. The colour should be first.

`plt.plot(x,y,'ro-',label="cos(x)")`



Advanced plot()

You can also :

- specify the width of the lines: `plot(x,y,linewidth=);`
- define precise colours in the RGB space normalized to 1: `plot(x,y,color=(0.5,0.5,0.5))`.

With three identical RGB values, we obtain a grey level.

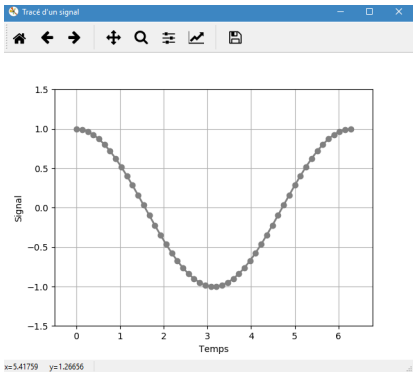
Redefining the axes / grid / labels

If you wish to control the extent of the axes of the graph, place a grid to facilitate reading and/or add labels on the abscissa and ordinate, the instructions are :

- `axis([xmin,xmax,ymin,ymax])`: defines the extent of the two axes;
- `grid()` without arguments, draws a grid;
- `xlabel('this is my x-coordinate'), ylabel('and my y-coordinate')`: draw the names of the axes;
- `title('this is the title')`: writes a title above the graph.

```
1 import matplotlib.pyplot as plt
2 import numpy as np
3 # 50 points entre 0 et 2*Pi
4 x=np.linspace(0,2*np.pi,50)
5 y=np.cos(x)
6 plt.figure("Tracé d'un signal")
7 plt.plot(x,y,'ro-',color=(0.5,0.5,0.5),linewidth=2)
8 plt.axis([-0.5, 2*np.pi+0.5, -1.5, +1.5])
9 plt.grid()
10 plt.xlabel('Temps')
```

```
plt.ylabel('Signal')
plt.show()
```



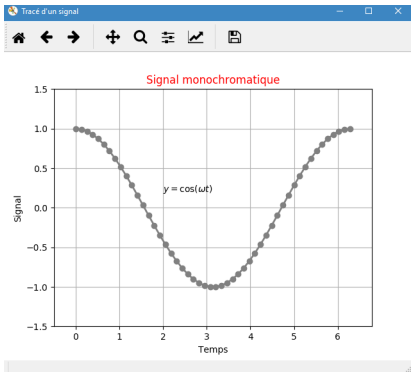
Extra : LaTeX

The text to be displayed may contain $\LaTeX$ statements, placed between two $ in a `str`, which includes mathematical formulas.

In practice, to avoid the misinterpretation of the \ symbol by Python, the string must be preceded by `r` (*raw string*)

In the following example, `r'$y=\cos(\omega t)$'` is placed with `text()` function.

```
import matplotlib.pyplot as plt
import numpy as np
# 50 points entre 0 et 2*Pi
x=np.linspace(0,2*np.pi,50)
y=np.cos(x)
plt.figure("Tracé d'un signal")
plt.plot(x,y,'ro-',color=(0.5,0.5,0.5),linewidth=2)
plt.axis([-0.5, 2*np.pi+0.5, -1.5, +1.5])
plt.grid()
plt.xlabel('Temps')
plt.ylabel('Signal')
plt.title('Signal monochromatique',color='r')
plt.text(2,0.2,r'$y=\cos(\omega t)$')
plt.show()
```



Multiple plots in one window

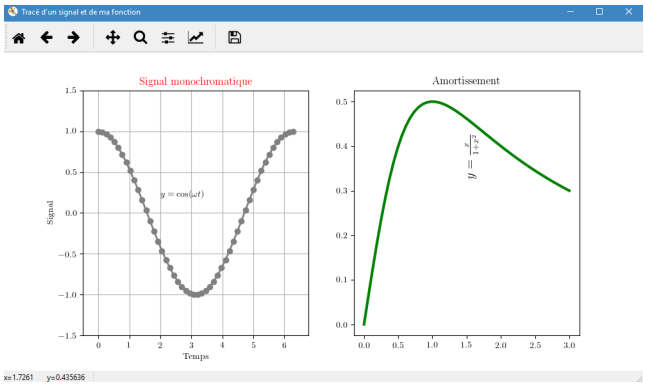
The `subplot()` function allows several graphs to be displayed in the same window.

But many configurations are possible and each one meets a specific layout need: two graphs aligned in a row, or in a column, sharing the same x-axis or y-axis, etc.

Example : moderately complex...

This example also introduces some options to the basic instructions...

```
def ma_fonction(x):
    y=x/(1+x**2)
    return y

import matplotlib.pyplot as plt
import numpy as np
# 50 points entre 0 et 2*Pi
x=np.linspace(0,2*np.pi,50)
y=np.cos(x)
# Figure de dimension 10 x 5 pouces donc rectangulaire
plt.figure("Tracé d'un signal et de ma fonction", figsize=(10,5))
# 1 ligne et 2 colonnes
# débute le tracé n°1
plt.subplot(1,2,1)
#
plt.plot(x,y,'ro-',color=(0.5,0.5,0.5),linewidth=2)
plt.axis([-0.5, 2*np.pi+0.5, -1.5, +1.5])
plt.grid()
plt.xlabel('Temps')
plt.ylabel('Signal')
plt.title('Signal monochromatique',color='r')
plt.text(2,0.2,r'$y=\cos(\omega t)$',style='italic')
#
# débute le tracé n°2
#
plt.subplot(1,2,2)
plt.plot(np.linspace(0,3,50),ma_fonction(x),'g-',linewidth=3)
plt.title('Amortissement')
plt.text(1.5,0.4,r'$y=\frac{x}{1+x^2}$',fontsize='x-large',rotation='vertical')
plt.show()
```



3. Other drawings

We do not always draw curves of analytical/mathematical expressions.

- y can represent a series of values y_i measured at a series of discrete points x_i, or at sampled times t_i (see FFT chapter). A distribution of measurement points is usually represented without connecting the points (*scatter plot*);
- y can represent the frequency of observations/measurements for a given $x_{i+1} - x_i$ sampling step. The *histogram* is then the best representation because it allows to properly visualize the width of the sampling in x.

3.1. Scatter plot

`scatter()` plots only points but, unlike `plot()`, allows you to change the size and colour of the points, and even to change the symbol for each (x_i, y_i) pair. This allows you to plot points from different experiments using different symbols or colours.

Definition :

The simplified syntax of scatter() is: `scatter(x,y,s,c,marker,cmap)`

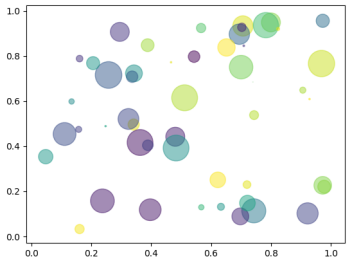
with :

- `x,y`: arrays containing the position of the points to be plotted;
- `s` (optional): `float` or array of the same length as `x` and `y`, containing the area of the symbols. In practice, it is advisable to make some tests to become familiar with the values (see the example);
- `c` (optional): array or value, containing the colours. Several possibilities are open. One or more numerical values refer to the `cmap` colour palette (colour map);
- `marker`: default 'o' but the list of possibilities is long;
- `cmap`: `str` containing the name of the colour palette. The default is `'viridis'`.

Other options exist... the list is long.

Example :

Here, each symbol has a random colour and size. The optional argument `alpha=0.5` sets the transparency to 50%, so that the different symbols can be seen to overlap.




```
import numpy as np
import matplotlib.pyplot as plt

# on fixe la graine du générateur aléatoire pour reproduire toujours le même graphique
np.random.seed(19680801)

N = 50
x = np.random.rand(N)
y = np.random.rand(N)
colors = np.random.rand(N)
area = (30 * np.random.rand(N))**2  # 0 to 15 point radii

plt.scatter(x, y, s=area, c=colors, alpha=0.5)
plt.show()

```

Extra :

List of all possible symbols:

marker	symbol	description
"."	•	point
","	·	pixel
"o"	●	circle
"v"	▼	triangle_down
"^"	▲	triangle_up
"<"	◀	triangle_left
">"	▶	triangle_right
"1"	Y	tri_down
"2"	⅄	tri_up
"3"	≺	tri_left
"4"	≻	tri_right
"8"	●	octagon
"s"	■	square
"p"	⬟	pentagon
"P"	✚	plus (filled)
"*"	★	star
"h"	⬢	hexagon1
"H"	⬢	hexagon2
"+"	+	plus
"x"	×	x
"X"	✖	x (filled)
"D"	◆	diamond
"d"	⧫	thin_diamond
"|"	|	vline
"_"	–	hline
0 (TICKLEFT)	–	tickleft
1 (TICKRIGHT)	–	tickright
2 (TICKUP)	'	tickup
3 (TICKDOWN)	,	tickdown
4 (CARETLEFT)	◀	caretleft
5 (CARETRIGHT)	▶	caretright
6 (CARETUP)	▲	caretup
7 (CARETDOWN)	▼	caretdown
8 (CARETLEFTBASE)	◀	caretleft (centered at base)
9 (CARETRIGHTBASE)	▶	caretright (centered at base)
10 (CARETUPBASE)	▲	caretup (centered at base)
11 (CARETDOWNBASE)	▼	caretdown (centered at base)
"None", " " or ""		nothing
'$...$'	*f*	Render the string using mathtext. E.g "f" for marker showing the letter f.
verts		A list of (x, y) pairs used for Path vertices. The center of the marker is located at (0, 0) and the size is normalized, such that the created path is encapsulated inside the unit cell.
path		A Path instance.
(numsides, 0, angle)		A regular polygon with numsides sides, rotated by angle.
(numsides, 1, angle)		A star-like symbol with numsides sides, rotated by angle.
(numsides, 2, angle)		An asterisk with numsides sides, rotated by angle.

Note :

To get the full list of available colour palettes, simply create an error on cmap and the list will be in the error message.

For example, `cmap='toto'` will cause an error...

or read: https://matplotlib.org/stable/tutorials/colors/colormaps.html

3.2. Histogram

The histogram is the basic representation for any statistical analysis.

The `hist()` function also **computes the histogram** before plotting it.

Definition :

```
hist(x, bins=None, range=None, density=False, weights=None,
cumulative=False, bottom=None, histtype='bar', align='mid',
orientation='vertical', rwidth=None, log=False, color=None, label=None,
stacked=False, *, data=None, **kwargs)
```

with:

- `x`: array or sequence of arrays that are not necessarily of the same length;
- `bins` (10 by default): if `bins` is an `int`, it is the number of intervals, all of the same width. If `bins` is a sequence, it defines the position of the left boundaries of each interval and ends with the position of the right boundary of the last interval. In this case, the intervals are defined one by one and are not necessarily the same width. Finally, if `bins` is a `str`, it indicates a particular automatic sampling strategy ('auto', 'fd', 'doane', 'scott', 'stone', 'rice', 'sturges', or 'sqrt').
- `range`: `tuple` containing the min and max limits of x to calculate frequencies. The default is `min(x)` and `max(x)`. If `bins` is a sequence, range has no effect.

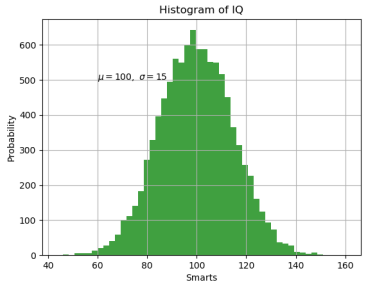
- `density`: if `True`, `hist()` calculates the probability density so that the integral of the histogram is 1.
- `weights`: arrays the weight of each of the `x` values. By default, each point has a weight of 1 in the count. For example, these weights allow to take into account measurement errors on `x` (weights inversely proportional to the errors).
- `cumulative`: `True`, `False` (default) or `-1`. So the last case, the cumulative is made towards the small values of x.
- `histtype`: 'bar' (default), 'barstacked', 'step', 'stepfilled'
- etc...

This function returns 3 items (in a tuple) that are often interesting to retrieve:

- frequency table (or sequence of tables corresponding to each table entered by `x`) ;
- left limits of the intervals + right limit of the last interval on the right;
- the 3rd value is only relevant for OO programming.

Example :

inspired by the matplotlib documentation:




```
import numpy as np
import matplotlib.pyplot as plt

# graine fixe pour reproductibilité
np.random.seed(19680801)

# moyenne et écart-type de la distribution normale
mu, sigma = 100, 15
x = mu + sigma * np.random.randn(10000)

plt.close()
plt.figure('Histogramme')

n, bins, patches = plt.hist(x, 50, density=False, facecolor='g', alpha=0.75)

plt.xlabel('Smarts')
plt.ylabel('Probability')
plt.title('Histogram of IQ')
plt.text(60., 500., r'$\mu=100,\ \sigma=15$')

# cas density=True
#plt.text(60, .025, r'$\mu=100,\ \sigma=15$')

plt.grid(True)
plt.show()

```

4. Plotting surfaces z=f(x,y)

We will plot the function $z = f(x, y)$.

In other words, as is often the case in physics, we have a function whose value depends on two space coordinates. For example the properties of a fluid (pressure, density, viscosity, etc.) on a plane, an electric potential etc.

The most conventional technique is to draw the curves of equal values, i.e. the set of points (x,y) for a particular value of z. This is known as equipotential, isobaric, isochoric, contour lines, etc., depending on the physical quantity to be plotted.

Alternatively, a colour representation for each intensity level (false colour coding) or a greyscale representation can be used.

Method : Statement of the problem

Let us take the example of a potential $\Phi(x, y)$. To plot the values of Φ as a function of coordinates (x, y) on a graph, we must first determine its values at particular points.

To do this, we define a *mesh* which covers the space $[x_{\min} : x_{\max}; y_{\min} : y_{\max}]$ to be visualized. As far as possible, we will choose a mesh with regular spacing that we note h_x and h_y for each of the two directions.

The coordinates of the *nodes* of the mesh are then :

$x_i = x_{\min} + i \times h_x$
$y_i = y_{\min} + j \times h_y$

The values of Φ are calculated only at the nodes of coordinates (x_i, y_j).

Numerically, this is equivalent to calculating a matrix/2-D array $Phi[i, j] \equiv \Phi_{ij} \equiv \Phi(x_i, y_j)$. It is an appropriate function of `matplotlib.pyplot` (`contour()`, `contourf()`, `imshow()`, etc.) which will then interpolate between these values to draw a curve which connects the points of same value.

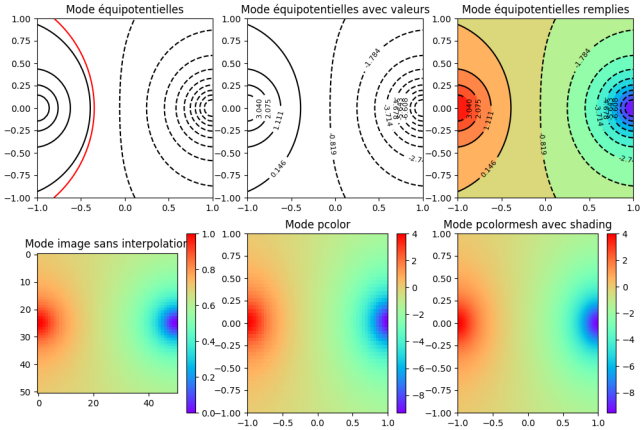
The first operation is to prepare two 2-D arrays which contain separately the `x` and `y` coordinates of each node for a given pair (i, j).

For a given (i, j) we want to know the value of `x`: so `x=XX[i,j]`. The same applies to `y=YY[i,j]`. `XX` and `YY` are 2D arrays filled with **redundant** information (only in case of a regular mesh).

The `numpy.meshgrid()` function performs this calculation.

The second operation is to calculate the values of `Phi` at each of the points in the `XX` and `YY` arrays. Then the 2-D array `Phi(i,j)` can be plotted.

Example : Equipotentials of a Coulombian or Newtonian potential



```
# -*- coding: utf-8 -*-
"""
Created on Tue Oct 10 14:26:24 2017

@author: Hervé Wozniak
"""
#
# Graphique 2D
import numpy as np
import matplotlib.pyplot as plt

# astuce pour faire afficher (ou pas) certaines informations
verbose = False #True

# ------------------------------
# Maillage de l'espace graphique
# ------------------------------

# bornes de l'espace physique (choisies symétriques)
xmin,xmax,ymin,ymax=-1.0,+1.0,-1.0,+1.0

# nombre de noeuds du maillage
# Nombre impair pour s'assurer d'un noeud au centre
N_x, N_y = 51,51

# abscisses et ordonnées
xp=np.linspace(xmin,xmax,N_x)
yp=np.linspace(ymin,ymax,N_y)

# calcule la résolution dans les deux directions
h_x=xp[1]-xp[0]  # N_x/(xmax-xmin)
h_y=yp[1]-yp[0]  # N_y/(ymax-ymin)
if verbose: print('Résolutions spatiales=',h_x,h_y)

# préparation du maillage
Xxp, Yyp = np.meshgrid(xp,yp)

if verbose: print('Coordonnées X des noeuds',Xxp)

# -----------------
# problème physique
# -----------------
#
# particule chargée n°1 (positive)
C_1=1.
x_1, y_1=-1.0,0.

# particule chargée n°2 (négative)
C_2=-2.
x_2, y_2=+1.0,0.

# constante de couplage de l 'interaction
K=1.0

# calcule les distances particules-noeuds
# ATTN : divisions par 0 lorsque particule sur un noeud
# solution par lissage
D_1=np.sqrt((Xxp-x_1)**2+(Yyp-y_1)**2+np.min((h_x,h_y)))
#                                     terme lissage
D_2=np.sqrt((Xxp-x_2)**2+(Yyp-y_2)**2+np.min((h_x,h_y)))

# calcule potentiel physique
Phi = K*C_1/D_1 + K*C_2/D_2
```

```
# IMPORTANT
# Phi a les mêmes dimensions que Xxp et Yyp
if verbose: print('Potentiel Phi=',Phi)
# ------------------------
# fin du problème physique
# ------------------------

# --------------------
# Début des graphiques
# --------------------

# debut de la figure en 'Courbes de niveaux'
ixplot,jyplot=2,3  #nb lignes, colonnes
iplot=1            # compteur de figure

# fermeture du tracé précédent
plt.close()

# ouverture nouvelle fenêtre
plt.figure('Equipotentielles',figsize=(jyplot*4,ixplot*4))
plt.subplot(ixplot,jyplot,iplot)

# trace les courbes d'équipotentielles pour N_contours
N_contours=15

# les isocontours négatifs sont en tracés en tirets (par défaut)
#              ,    ,    ,   vecteur avec les niveaux à tracer        ,  noir
plt.contour(Xxp, Yyp, Phi, np.linspace(Phi.min(),Phi.max(),N_contours), colors='k')
# N.B. : 2 des contours correspondent aux min et max de Phi : ils ne sont pas visibles
# équipotentielle nulle en rouge :
plt.contour(Xxp,Yyp,Phi,[0.],colors='r')

# axes de mêmes longueurs
plt.axis([xmin,xmax,ymin,ymax],'equal')
plt.title('Mode équipotentielles')

# La même chose avec des valeurs sur les contours
iplot+=1
plt.subplot(ixplot,jyplot,iplot)
CS=plt.contour(Xxp,Yyp,Phi,np.linspace(Phi.min(),Phi.max(),N_contours),colors='k')
plt.clabel(CS, inline=1, fontsize=8)
# axes de mêmes longueurs
plt.axis([xmin,xmax,ymin,ymax],'equal')
plt.title('Mode équipotentielles avec valeurs')

# Autre représentation avec coloriage des inter-contours
iplot+=1
plt.subplot(ixplot,jyplot,iplot)
plt.contourf(Xxp,Yyp,Phi,np.linspace(Phi.min(),Phi.max(),N_contours),cmap='rainbow')
CS=plt.contour(Xxp,Yyp,Phi,np.linspace(Phi.min(),Phi.max(),N_contours),colors='k')
plt.clabel(CS, inline=1, fontsize=8)
plt.axis([xmin,xmax,ymin,ymax],'equal')
plt.title('Mode équipotentielles remplies')

# debut de la figure en 'mode image'
iplot+=1
plt.subplot(ixplot,jyplot,iplot)
plt.title('Mode image sans interpolation')
# les pixels de l'image doivent être compris entre 0 et 1
plt.imshow((Phi-Phi.min())/(Phi.max()-
Phi.min()),origin="upper",cmap="rainbow",interpolation='None')
plt.colorbar()

```

```
126 # utilisation de pcolor()
127 iplot+=1
128 plt.subplot(ixplot,jyplot,iplot)
129 plt.title('Mode pcolor')
130 plt.pcolor(Xxp,Yyp,Phi,cmap='rainbow')
131 plt.colorbar()
132
133 # utilisation de pcolormesh()
134 iplot+=1
135 plt.subplot(ixplot,jyplot,iplot)
136 plt.title('Mode pcolormesh avec shading')
137 plt.pcolormesh(Xxp,Yyp,Phi,cmap='rainbow',shading="gouraud")
138 plt.colorbar()
139
140 # on montre la figure
141 plt.show()
142
143 # sauvegarde de la figure en format PNG 1200 x 1200 (1000 x inches)
144 plt.savefig('equipotentielles.png')
```

Extra **:**

There are mesh functions that give more compact information (`numpy.mgrid()`, `numpy.ogrid()`, etc.). To be checked in the NumPy documentation depending on the use case.

IV Applications: Fourier transforms (spectral analysis), graph theory, etc.

1. Fast (Discrete) Fourier Transform

The Fourier transform is an operation that no longer needs to be introduced at Master level. It generalizes Fourier series for non-periodic functions.

It is omnipresent in physics (optics, quantum mechanics, etc.) and in engineering sciences (acoustics, signal processing, filtering, etc.).

However, whatever the language and the algorithms used, its implementation always requires attention.

1.1. Basics

A transform is generally a method of expressing a function as a sum, discrete or continuous, of basic functions. Among the most commonly used transforms, we can mention :

- Fourier: continuous ($\hat{f}(\sigma) = \int_{-\infty}^{+\infty} dx f(x)\, e^{-i\sigma x}$) or discrete;
- Laplace: $F(p) = \int_0^{+\infty} dt f(t)\, e^{-pt}$.

But there are many others (Legendre, Hankel, etc.).

When the function $f(x)$ is periodic or the sum of periodic functions, the Fourier transform makes it possible to find the frequency or frequencies (periods) of the various components.

If x has the dimension of time, the conjugate variable in Fourier space is a frequency or pulsation (depending on whether or not the writing convention expresses the 2π factor in the exponential, which modifies its normalization); hence the term *spectral analysis*.

If x is a distance, σ is a wave number (inverse of a wavelength).

Let $f(t)$ be a time-dependent signal (real or complex) and $\hat{f}(\nu)$ its Fourier transform:

$\hat{f}(\nu) = \int_{-\infty}^{+\infty} dt f(t)\, e^{-i2\pi\nu t}$.

Due to the measurement of the signal, the value of $f(t)$ is only known at a finite number N of points regularly spaced by δ_t (sampling timestep). Thus, $f_k \equiv f(t_k), \qquad t_k \equiv k\delta_t, \qquad k = 0, \ldots, N-1.$

In discrete form, $\hat{f}$ is only known in a number N of frequencies ν_n. The frequency sampling step is $\frac{1}{N\delta_t}$, and is therefore linked to the total length of the measurement. The ν_n can then be expressed as $\nu_n \equiv \frac{n}{N\delta_t}, \qquad n \in [-\frac{N}{2}, \frac{N}{2}]$.

When $\nu_{\pm N/2} = \pm\frac{1}{2\delta_t}$, $\nu_{\pm N/2}$ is called "*Nyquist frequency*".

$\hat{f}$ is therefore expressed as :

$\hat{f}(\nu_n) = \int_{-\infty}^{+\infty} dt f(t)\, e^{-i2\pi\nu_n t} \sum_{k=0}^{N-1} \delta_t f_k\, e^{-i2\pi\nu_n t_k}$.

By injecting the values of ν_n and t_k into the above expression, we obtain :

$\hat{f}(\nu_n) \approx \delta_t \sum_{k=0}^{N-1} f_k e^{-i2\pi\, kn\,/N} \equiv \delta_t \hat{f}_n$

which is the definition of the *discrete Fourier transform (DFT)* $\hat{f}_n$.

The publication of a *fast Fourier transform (FFT)* algorithm by Cooley and Tukey (1965) facilitated the diffusion of the method in numerical codes.

If $\hat{f}_n \equiv \sum_{k=0}^{N-1} f_k e^{-i2\pi\, kn\,/N}$, then the inverse DFT is :

$$f_k = \frac{1}{N} \sum_{k=0}^{N-1} \hat{f}_n e^{+i2\pi\, kn\,/N} \qquad k = 0, \ldots, N-1.$$

Reminder : Nyquist frequency

The DFT is only defined between negative and positive Nyquist frequencies.

Conversely, the Nyquist frequency (or aliasing frequency) is the maximum frequency that a continuous signal must contain for its discrete representation (sampling at regular intervals) to be faithful.

It is half the sampling frequency: $|\nu_c| = \frac{1}{2}\frac{1}{\delta_t}$.

The Nyquist frequency is related to the **sampling** of the signal, not the signal itself.

It is totally determined by the characteristics of the measurement, i.e. the discretisation of the continuous (analog) signal.

Note :

If the frequency of the continuous signal is known (analogue → digital conversion, digitization, etc.), the Nyquist frequency can be used to determine the sampling properties (e.g. audio CD).

Thus, the sampling frequency of a signal must be equal to or greater than twice the maximum frequency contained in that signal, in order to convert that signal from a continuous to a discrete (time-discontinuous) form. This is the sampling (or Shannon) theorem.

Warning : Aliasing

If a signal represented by a sample contains frequencies above the Nyquist frequency, *ghosts* will appear in the DFT (*aliasing* phenomenon).

Frequencies above the Nyquist frequency will pollute the spectrum obtained by DFT at lower frequencies.

The inverse DFT no longer allows the original discrete signal to be recovered.

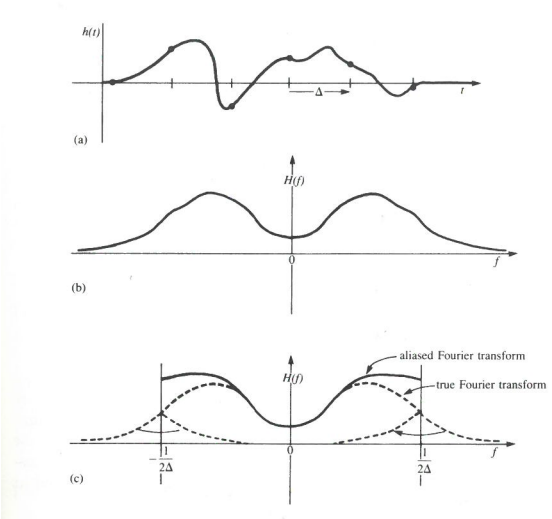


Figure 12.1.1. The continuous function shown in (a) is nonzero only for a finite interval of time T. It follows that its Fourier transform, shown schematically in (b), is not bandwidth limited but has finite amplitude for all frequencies. If the original function is sampled with a sampling interval Δ, as in (a), then the Fourier transform (c) is defined only between plus and minus the Nyquist critical frequency. Power outside that range is folded over or "aliased" into the range. The effect can be eliminated only by low-pass filtering the original function *before sampling*.

1.2. NumPy implementation

In NumPy, a choice of sign and normalization has been made, resulting in the definition of the DFT coefficients used earlier.

For other libraries, other conventions can be used:

- Direct DFT with $e^{+i\cdots}$ instead of $e^{-i\cdots}$;
- pulsation instead of frequency, hence a factor of $\frac{1}{2\pi}$ in the inverse DFT;
- symmetrization of the direct and inverse DFTs, with a factor of $\frac{1}{\sqrt{N}}$ in both definitions (NumPy optionally allows this).

Definition :

To calculate the DFT of a function `f` sampled at `N` points, the NumPy commands are:

- `F=numpy.fft.fft(f,n)` for a 1-D possibly complex signal, and `ifft()` for the inverse DFT.

  If $n < N$ the array `f` is truncated. If $n > N$, the array `f` is automatically enlarged with 0s (*padding*);
- `F=numpy.fft.fft2(f,shape,axes,norm)` in 2-D. `shape` is a `tuple` of two integers representing `n` on each of the axes. `axes` specifies on which axis the DFT should run (by default, both). `norm` changes the normalization convention ("ortho" uses the factor $\frac{1}{\sqrt{N}}$) ;
- `numpy.fft.fftn()` in n-D.

Result ordering

The values are in the so-called "standard order" because it is common to many libraries, in many languages.

If `F = fft(f, n)` then:

- `F[0]` contains the zero frequency term;
- `F[1:n/2]` contains the positive frequency terms while `F[n/2+1:n]` contains the negative frequency terms, in ascending frequency order.

`numpy.fft.fftshift(F)` shifts the transform and its frequencies to place the 0 in the centre of the array. `numpy.fft.ifftshift(F)` performs the opposite operation.

Extra :

The function `numpy.fft.fftfreq(n,d=sampling_interval)` returns an array with the frequencies corresponding to the elements of `F[:]`. If the second argument is omitted, the frequencies are given in units of frequency sampling steps, thus in the interval $\left[-\frac{1}{2}, +\frac{1}{2}\right[$.

Remember that ν_n can then be expressed as $\nu_n \equiv \frac{n}{N\delta_t}, \qquad n \in [-\frac{N}{2}, \frac{N}{2}[$.

Extent in the frequency domain

For an **even** number of points, `F[n/2]` contains the term at the Nyquist frequency (positive or negative).

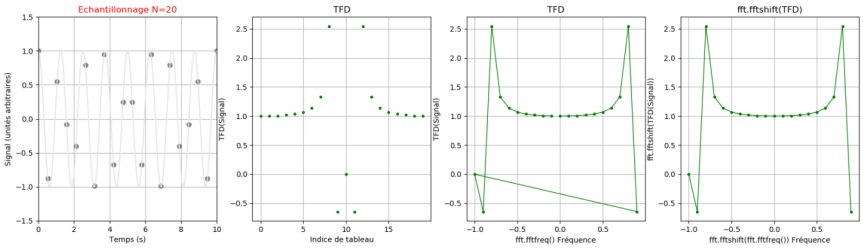
For an **odd** number of points, `F[(n-1)/2]` contains the term of greatest positive frequency, while `F[(n+1)/2]` contains the term of greatest negative frequency.

Example :

```
# -*- coding: utf-8 -*-
"""
Created on Mon Aug 27 14:26:24 2017

@author: Hervé Wozniak
"""
# caractéristique (secrète) du signal périodique analogique
```

```
freq_signal=0.8 # Hz

import matplotlib.pyplot as plt
import numpy as np

# caractéristiques de la mesure
npts_sample=20
length_sample = 10. # s

freq_sample = npts_sample / length_sample # Hz
print("Fréquence d'échantillonnage=",freq_sample, "Hz")

# on génère les points de mesure
x=np.linspace(0,length_sample,npts_sample)
# on génère le signal (ici en forme de cosinus)
y=np.cos(2*np.pi*x*freq_signal)

# transformée de Fourier discrète directe
tfy=np.fft.fft(y,npts_sample)
# fréquences associées aux éléments de 'tfy'
tfx=np.fft.fftfreq(npts_sample,d=1./freq_sample)

# Recherche fréquence de plus grande amplitude
print("max de TFD, indice et fréquence=",max(tfy),np.argmax(tfy),tfx[np.argmax(tfy)])

print("\n   La fréquence du signal est=",tfx[np.argmax(tfy)],' Hz \n')

print("Fréquence nulle,valeur=",tfx[0],tfy[0])
print("Fréquence de Nyquist, valeur=",tfx[int(0.5*npts_sample)],tfy[int(npts_sample/2)])
```

```
Fréquence d'échantillonnage= 2.0 Hz
max de TFD, indice et fréquence= (2.547289728769612+7.839751662649832j) 8 0.8
La fréquence du signal est= 0.8 Hz
Fréquence nulle,valeur= 0.0 (1.0000000000000002+0j)
Fréquence de Nyquist, valeur= -1.0 (1.0325074129013956e-14+1.7763568394002505e-15j)
```



1.3. Complexity

In the general case, since the signal is complex, the DFT is also complex.

`numpy.real(F)`, `numpy.imag(F)` are used to extract the real and imaginary parts of the spectrum respectively.

The representation in the complex plane is sometimes preferred:

- `numpy.abs(F)` (or `absolute(F)`) computes the amplitude of the spectrum (modulus of the complex number);
- `numpy.abs(F)**2` then represents the power of the spectrum;
- `numpy.angle(F)` is used to find the phase of the spectrum.

Definition **: Real signal**

When the signal is real, the negative frequency terms are the complex conjugates of the corresponding positive frequency terms. Calculating the negative frequency terms is therefore redundant and half the memory space allocated to the DFT can be saved.

We add '`r`' before '`fft`' in the name of the NumPy functions called: `numpy.fft.rfft(f,n)` for instance.

See https://docs.scipy.org/doc/numpy/reference/routines.fft.html for more information.

1.4. Fourier transform approximation

We have seen the relation between Fourier transform $\hat{f}(\nu_n)$ and DFT $\hat{f}_n$:

$\hat{f}(\nu_n) \approx \delta_t \sum_{k=0}^{N-1} f_k e^{-i2\pi kn/N} \equiv \delta_t \hat{f}_n$.

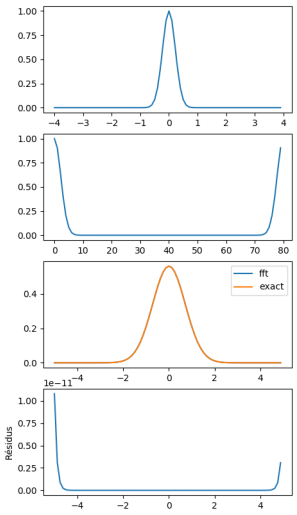
To calculate the Fourier transform of a continuous signal, one must :

- discretize the signal (make sure you have sampled enough) ;
- truncate it in time (make sure you have included all the periods, if the signal is periodic);
- discretise its spectrum: this is how we distinguish between DFT and DTFT (Discrete Time Fourier Transform) for which the frequency variable is continuous.

Example **: Gaussian example**

```
import numpy as np
import matplotlib.pyplot as plt

# définition de l'échantillonnage
nc = 40
dt = 0.1 # intervalle d'échantillonnage

# bornes dans le temps pour une gaussienne centrée en 0
tmax = (nc-1) * dt
tmin = -nc * dt

# definition d'un signal gaussien centré en t=0
t = np.linspace(tmin, tmax, 2*nc)
alpha = 10.0
f = np.exp(-alpha * t**2)

plt.close()
plt.figure("Tracé d'un signal et sa transformée de Fourier", figsize=(5,16))
plt.subplot(4,1,1)
plt.plot(t,f)

# on effectue un ifftshift pour positionner le temps zero comme premier element
a = np.fft.ifftshift(f)

plt.subplot(4,1,2)
plt.plot(a)

# on calcule la TFD
A = np.fft.fft(a)
# on effectue un fftshift pour positionner la frequence zero au centre
# on multiplie par le pas d'échantillonnage pour passer de la TFD à la TF
X = dt*np.fft.fftshift(A)

# calcul des frequences avec fftfreq, pour un échantillonnage de pas dt
n = t.size
freq = np.fft.fftfreq(n, d=dt)
f = np.fft.fftshift(freq)

```

```
# comparaison avec la solution exacte
exact=np.sqrt(np.pi/alpha) * np.exp( -(np.pi*f)**2 / alpha)
plt.subplot(4,1,3)
plt.plot(f, np.real(X), label="fft")
plt.plot(f, exact, label="exact")
plt.legend()

plt.subplot(4,1,4)
plt.plot(f, np.abs(np.real(X)-exact))
plt.ylabel("Résidus")

plt.show()
```



2. Convolution product

In signal processing and physics, many operations are related to the convolution product: filtering, smoothing (one of the functions is called '*kernel*'), auto- or inter-correlation, transfer/propagation, etc.

The convolution product is defined by $(g * h)(t) \equiv \int_{-\infty}^{+\infty} d\tau\, g(\tau)\, h(t-\tau)$ and can be calculated via the inverse Fourier transform of the simple product of the Fourier transforms of g and h: $\hat{g}(\nu) \times \hat{h}(\nu)$.

The inter-correlation function of g and h, $\mathcal{F}(g,h)$, which is defined by $\mathcal{F}(t) \equiv \int_{-\infty}^{+\infty} d\tau\, g(\tau+t)\, h(\tau)$, can be calculated from the product $\hat{g}(\nu) \times \hat{h}^*(\nu)$, where $\hat{h}^*$ is the complex conjugate of $\hat{h}$.

⚠ *Warning* :

One usually pays attention to the notion of periods or cycles when dealing with discrete Fourier transforms.

But when using the convolution product, one is most often dealing with non-periodic signals or kernels. The calculation of convolution products by DFTs is then often a source of errors, especially when the **padding** operation is forgotten.

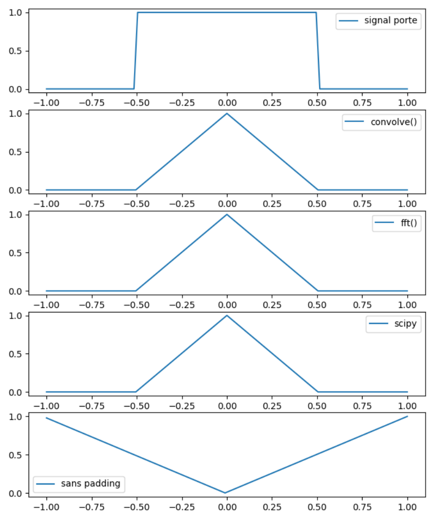
🔎 *Example* : Convolution product of the gate function by itself

```
import numpy as np
import matplotlib.pyplot as plt

# fonction porte de largeur 1 centrée en 0
def porte(t):
      res=np.zeros(t.shape)
      res[ (-0.5<=t)*(t<=0.5) ] = 1
```

```
      return res

convmod='full'  # 'full' = mode par défaut
npts=100
t=np.linspace(-1,1,npts)
dt=2./npts
print("Valeur du pas=",dt)
p=porte(t)
print("Longueur du signal=",len(p))
#
# produit de convolution discret avec convolve()
c=np.convolve(p,p,convmod) * dt
# le produit de convolution de vecteurs de dimensions N et M est N + M - 1
print("Dimensions du produit de convolution NxM-1=",np.shape(c))

# produit de convolution par FFT
# il faut IMPERATIVEMENT compléter avec des 0 jusqu'à N+M-1
# sinon le signal est considéré comme périodique
pa = np.pad(p, (0, npts-1), mode='constant')
pb = np.pad(p, (0, npts-1), mode='constant')
print("Longueur du signal après ajout de 0 à droite",len(pa))
print(pa)

tfp1=np.fft.fft(pa,len(pa)) * dt # produit par dt pour vraie TF
tfp2=np.fft.fft(pb,len(pb)) * dt

cfft=np.fft.ifft(tfp1*tfp2,len(tfp1)) / dt

# sans padding, c'est une convolution circulaire
tfp=np.fft.fft(p,len(p)) * dt # produit par dt pour vraie TF
cfft_false=np.fft.ifft(tfp*tfp,len(tfp)) / dt

# Utilisation de package SciPy
from scipy import signal
cfft2=signal.fftconvolve(p,p,convmod) * dt

plt.close()
plt.figure("Produit de convolution",figsize=(8.,10.))
nfig=5
plt.subplot(nfig,1,1)
plt.plot(t,porte(t),label="signal porte")
plt.legend()
plt.subplot(nfig,1,2)
plt.plot(np.linspace(-1,1,len(c)),c,label="convolve()")
plt.legend()
plt.subplot(nfig,1,3)
plt.plot(np.linspace(-1,1,len(cfft)),np.real(cfft),label="fft()")
plt.legend()
plt.subplot(nfig,1,4)
plt.plot(np.linspace(-1,1,len(cfft2)),np.real(cfft2),label="scipy")
plt.legend()
plt.subplot(nfig,1,5)
plt.plot(np.linspace(-1,1,len(cfft_false)),np.real(cfft_false),label="sans padding")
plt.legend()
plt.show()
```

```
Valeur du pas= 0.02
Longueur du signal= 100
Dimensions du produit de convolution NxM-1= (199,)
Longueur du signal après ajout de 0 à droite 199
[0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
0. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1.
1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1.
```

```
1. 1. 1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
0. 0. 0. 0. 0. 0. 0.]

```



Extra :

Padding is the operation that consists of doubling the array to be transformed in all its directions with 0s.

3. Geometric data structures (chapter in progress)

This chapter covers a number of data structures designed for storing multi-dimensional geometric data. Geometric data structures are fundamental to the efficient processing of data sets arising from many applications, including spatial databases, automated cartography (maps) and navigation, computer graphics, robotics and motion planning, solid modeling and industrial engineering, particle and fluid dynamics (both Eulerian and Lagrangian hydrodynamics), molecular dynamics and drug design in computational biology, machine learning, image processing and pattern recognition, computer vision, etc.

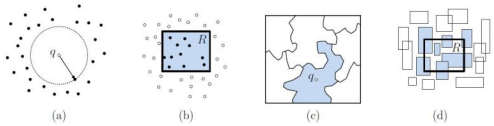
Fundamentally, the goal is to store a large datasets consisting of geometric objects (e.g.,

points/particles, lines and line segments, simple shapes (such as balls, rectangles, triangles), and complex shapes such as surface meshes) in order to answer queries on these data sets efficiently.

While some of our explorations will involve delving into geometry and linear algebra, fortunately most of what we will cover assumes no deep knowledge of geometric objects or their representations. Given a collection of geometric objects, there are numerous types of queries that we may wish to answer.

- Nearest-Neighbor Searching: Store a set of points so that qiven a query point q, it is possible to find the closest point of the set (or generally the closest k objects) to the query point (see Fig. 1(a)).
- Range Searching: Store a set of points so that given a query region R (e.g., a rectangle or circle), it is possible to report (or count) all the points of the set that lie inside this region (see Fig. 1(b)).
- Point location: Store the subdivision of space into disjoint regions (e.g., the subdivision of the globe into countries) so that given a query point q, it is possible determine the region of the subdivision containing this point efficiently (see Fig. 1(c)).
- Intersection Searching: Store a collection of geometric objects (e.g., rectangles), so that given a query consisting of an object R of this same type, it is possible to report (or count) all of the objects of the set that intersect the query object (see Fig. 1(d)).

- Ray Shooting: Store a collection of object so that given any query ray, it is possible to determine whether the ray hits any object of the set, and if so which object does it hit first.



Common geometric queries: (a) nearest-neighbor searching, (b) range searching, (c) point location, (d) intersection searching.

In all cases, you should imagine the size n of the set is huge, consisting for example of millions of objects, and the objective is to answer the query in time that is significantly smaller than n, ideally $O(\log n)$. It is not always possible to achieve efficient query times with storage that grows linearly with n. In such cases, we would like to achieve, for example, $O(n \log n)$. It will also be desirable to provide dynamical updates, allowing for the insertion and deletion of objects.

3.1. Point representation in 2D (quadtree, octree, etc.)

Suppose that we wish to store a set of n points in 2-dimensional space. In binary trees, each point naturally splits the real line in two. In two dimensions if we run a vertical and horizontal line through the point, it naturally subdivides the plane into four quadrants about this point.

The resulting data structure is called a *point quadtree*.

Each node has four (possibly null) children, corresponding to the four quadrants defined by

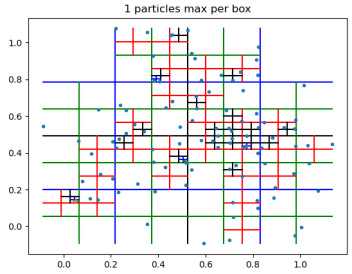
the 4-way subdivision. We label these according to the compass directions, as NW, NE, SW, and SE. In terms of implementation, you can think of assigning these the values 0, 1, 2, 3, and use them as indices to a 4-element array of children pointers.

Note **:**

In 3D, the corresponding structure is naturally called an *octree*. As the dimension grows, the term *quadtree* is often used in arbitrary dimensions, even though the outdegree of each node is 2^d, not four.

Similarly, the 8 directions can be encoded with a digit ranging from 0 to 7, which can easily be expressed with an *octal* (integer in base 8). For example `0o1` is an octal in python.

Example **: Space partitioning in N-body simulations**



Graphic 3

TBW

3.2. Point k-d tree

Point quadtrees can be generalized to higher dimensions, the number of children grows exponentially in the dimension, as 2^d. For example, in 20-dimensional space, every node has 2^{20}, or roughly a million children. The simple quadtree idea is not scalable to very high dimensions.

As in the case of a quadtree, the cell associated with each node is an axis-aligned rectangle (assuming the planar case) or a hyper-rectangle in d-dimensional space. When a new point is inserted into some node (equivalently into some cell), we will split the cell by a horizontal or vertical splitting line, which passes through this point. In higher dimensions, we split the cell by a $(d-1)$ dimensional hyperplane that is

orthogonal to one of the coordinate axes. In any dimension, such a split can be specified by giving the cutting axes (which can be represented as an integer from 0 to $d-1$), and also called the cutting value. Following the approach used in point quadtrees, the cutting value will be taken from the coordinates of the point being stored in this node. Thus, along with its left and right child pointers, we can think of every node as storing two items, an integer cutting dimension and a point.

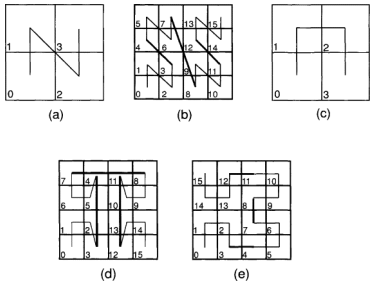
3.3. Space filling curves

We propose to map multi-dimensional data points to a one-dimensional space using a Z-order curve (*courbe de Lebesgue* in French, or Morton curve in computer science) or a fixed space-filling curve (SFC).

These techniques are used in a wide range of applications, including N-body dynamical and hydrodynamical simulations, real-time collision detection, 3D games, virtual reality applications, fast matrix multiplication, etc. They are directly used by tree-based method in molecular dynamics, Lagrangian hydrodynamics, mesh-free simulations etc. Indeed, once the data are sorted into Z-ordering, any one-dimensional data structure can be used, such as simple one dimensional arrays, binary search trees, B-trees, skip lists or hash tables.

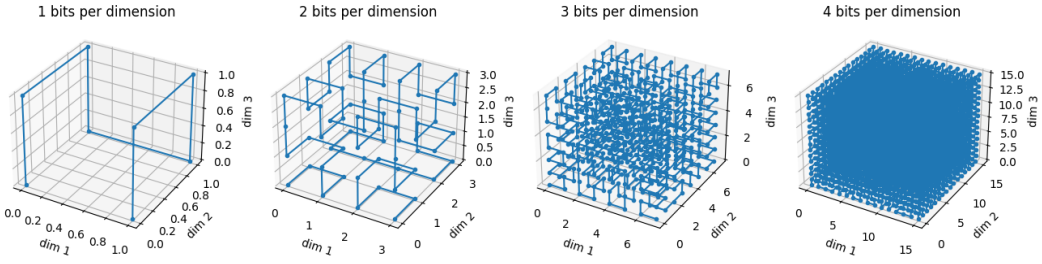
The SFC is a kind of generalization of the Z-order curve, as for the Hilbert curve. The Hilbert curve (aka the Hilbert space-filling curve) is a continuous fractal space-filling curve. Both the true Hilbert curve and its discrete approximations give a mapping between 1D and 2D space that preserves locality fairly well. This means that two data points which are close to each other in one-dimensional space are also close to each other after folding. This is not always the case with the Z-order curve. This method is particularly useful in N-body simulations for identifying the neighbouring particles of a point in space.

A number of SFC have been proposed, with the goal of maintaining proximity in space also in the one-dimensional embedding the curve defines. Since the data structure based on a SFC must adapt the partition pattern dynamically, SFC usually have recursive definitions. Figure above shows two building blocks ((a) and (c)) and the three best-known space-filling curves based on them — bit interleaving (b), the Gray code (d), and Hilbert's curve (e).



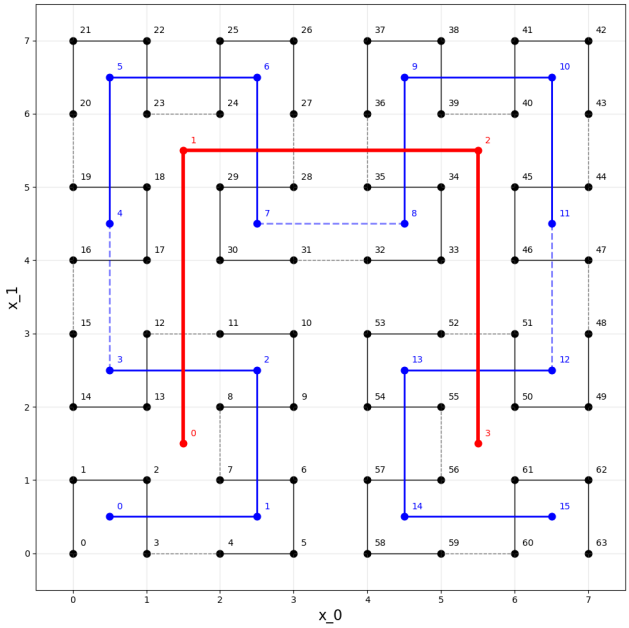
Graphic 4

SFC is widely used to index partitions of space using nested cubes, obtained by dividing each cube into 8 sub-cubes. Refinement level 0 is made up of a single cube. At level 1, the level 0 cube is subdivided into 8 sub-cubes. At level 2, each level 1 cube is again subdivided into 8, and so on.



Graphic 5

Two intelligent numbering schemes can be adopted as required: 1) a binary encoding that preserves both the level of refinement and the spatial direction 2) an encoding in which two cubes with successive numbers are necessarily adjacent in physical space. The figure above gives a 2D example.



Graphic 6

For example, a cube with coordinate 53 on the Hilbert curve has coordinates (4,3) in space. Its neighbors 52 and 54 are necessarily neighbors (and have coordinates (5,3) and (4,2)).

Example : Implementation for mesh-free N-body simulation

SFCs are continuous functions that map the unit interval into an n-dimensional hypercube. In conjunction with three-dimensional particle simulations, we are interested in discretized versions of SFCs that map the key space $\left[0, 2^{3L}\right[$ onto a grid of 3D integer coordinates $\left[0, 2^{L}\right[\times \left[0, 2^{L}\right[\times \left[0, 2^{L}\right[$, where L is the number of refinement (tree depth). In 3D, the number of bits required to store an SFC key or a point on the grid equals $3L$. Since current computer architectures have instructions for either 32- or 64-bit integers, reasonable choices for L are 10 or 21, or – depending on accuracy requirements – multiples thereof.

The utility of certain SFCs for numerical simulations stems from their relation to octrees. If we express a key k of the Morton Z-curve with $3L$bits as a sequence of L octal digits, $k = k_1 k_2 \cdots k_L$, Warren and Salmon found [23] that if the first octal digit l_1 of another key l matches k_1, then k and l decode into 3D coordinates that lie in the same octant of the root octree node. And by induction: if the first i octal digits of keys k and l match, then there exists an octree node at the i^{th} division level that contains the decoded 3D coordinates of both keys. Equivalently, an octree node at the i^{th} division level contains the 3D coordinates that encode into the

key range $\left[k, k + 8^{L-i}\right[$ for some unique k with $k \mod 8L - i = 0$. Consequently, the number of octal digits L in the SFC key is equal to the octree depth that the key is able to resolve. More generally, this correspondence between octal digits of the key and nodes of an octree applies to any type of SFC that traverses a cube octant by octant, with the Hilbert curve as a further example. The encoding and decoding of 3D grid points into Hilbert keys is computationally expensive compared to the simpler Morton Z-curve, but in contrast to the latter, any continuous segment of the curve is mapped to a compact 3D volume, while an interval of Morton keys may correspond to disconnected 3D volumes. In distributed simulations, the smaller surfaces of subdomains defined as segments of the Hilbert curve require less communication, outweighing the higher computational cost of key encoding compared to the Morton Z-curve.

V Useful routines for the physicist

Other modules are often useful for the physicist. We cannot make an exhaustive list because it depends on the field of application. But pointing out their existence makes it possible to go and read the online documentation and to use them if necessary. We have already mentioned **SciPy** but we will only mention a few sub-modules. See http://scipy-lectures.org/ for a better understanding of the SciPy ecosystem.

1. Major warning

⚠ *Warning* : On the proper use of a toolbox

If you were given a toolbox containing a hammer, three trowels, two squares, a plumb bob and a level, would you know how to build a cinder block wall?

With few exceptions, you have never built a wall and have never been trained as a mason.

So this toolbox is at best useless, at worst dangerous, if you don't know what the tools are for and are not trained in their safe use.

Python toolboxes

High-level modules, such as SciPy, Pytorch, etc., should be seen as masonry toolkits.

If you know how each of these tools works (numerical analysis algorithms for physics), then you will be able to choose the most suitable function to solve your problem without having to recode the algorithm you were taught. But this choice, if you want it to be an informed one, cannot be made just by reading an online documentation. You need to have practised algorithmics beforehand (cf. Félix Brümmer course).

2. Constants (https://docs.scipy.org/doc/scipy/reference/constants.html)

`scipy.constants` module contains:

- mathematical constants (π of course, but also e.g. the Gold number);
- usual physical constants under various names (`c` or `speed_of_light`, `h` or `Planck`, `hbar`, `k`, `G` or `gravitational_constant`, etc.);
- a database of values (mainly atomic and nuclear) from the CODATA2018 recommendation: Recommended Values of the Fundamental Physical Constants 2018 (https://physics.nist.gov/cuu/Constants/). For example, the ratio of the masses of the W/Z bosons is given. For each constant, the value, the unit and the uncertainty are returned.

```
>>> import scipy.constants
>>> scipy.constants.pi
3.141592653589793
>>> scipy.constants.Avogadro
6.02214076e+23
>>> scipy.constants.k
1.380649e-23
>>> scipy.constants.physical_constants['Boltzmann constant']
(1.380649e-23, 'J K^-1', 0.0)
>>> scipy.constants.physical_constants['W to Z mass ratio']
(0.88153, '', 0.00017)
```

3. Other SciPy modules

Note **:**

Once the choice of algorithm is made, it is advisable to use the corresponding function (if it exists) of a SciPy or Numpy type module. These modules implement the algorithms in compiled language (in C or Fortran). They are often optimised and therefore much faster and more accurate than a pure Python version. However, let's not forget the previous warning: the choice must always be informed by knowledge of the algorithm.

https://docs.scipy.org/doc/scipy/scipy-ref-1.7.0.pdf (3300 pages)

- `scipy.special`: special functions (as in NumPy). Some versions are compiled to be faster;
- `scipy.integrate`: quadrature;
- `scipy.optimize`: root finding, fitting by least square method, linear or quadratic programming, etc.;
- `scipy.interpolate`: polynomial interpolation, splines, etc.;
- `scipy.fft scipy.fftpack`: Fourier transform, and also Hankel. Includes `numpy.fft` but works better in some cases;
- `scipy.signal`: B-splines, filtering, spectral analysis;
- `scipy.linalg`: includes `numpy.linalg` and add a few routines based on BLAS and LAPACK;
- `scipy.sparse`: sparse matrix management and linear analysis algorithm taking advantage of the low density of matrices;
- `scipy.spatial`: mesh generation/sampling techniques (Delaunay triangulation, Voronoi);
- `scipy.stats`: more advanced probabilities/statistics than with `numpy`;
- `scipy.ndimage`: multi-dimensional image analysis ;
- `scipy.io`: reading (and sometimes writing) of many well-known file formats (wav, jpg, etc).

VI Files

In addition to modules, packages and programs that are in files, data can also be stored outside a program.

The most typical case is when you want to plot the result of a numerical calculation that requires several hours (or days) of CPU time. In this case, the program that solves the physical problem numerically is separated from the program that plots the results. The program in charge of the graphing reads back the data written by the program that solves the equations. The two programs may be written in different languages and run on different computers.

Another case is when data is produced with a physical measuring instrument (spectrograph, imager, sensor, etc.). The instrument produces a data file that can be viewed or manipulated later.

Formally we have already seen input-output functions: `print()` and `input()` (because the keyboard and the screen can be considered as files!).

Finally, it is good to know that under Linux, everything (hard disk, DVD, mouse, etc.) is a file, unlike Windows...

1. With NumPy and other modules

When it comes to reading files, there is always someone to remind you of the existence of `numpy.loadtxt()`. It's magic, it allows you to stop wondering!

But the central object of NumPy is the notion of array (ndarray). NumPy's I/O routines are therefore dedicated to reading/writing numerical arrays. Therefore, `numpy.loadtxt()` will only read a text file if its content is already formatted to have N rows and M columns of the same kind, made of integers or reals, but especially not mixed with text because NumPy does not make arrays with strings. Ideally, `numpy.savetxt()` will have been used to write the file.

Similarly for the binary format with extension .npz, specific to NumPy, one can consider using `numpy.load()` which returns a dictionary. If the name of the arrays was passed as an argument when writing with `numpy.savez()` or `numpy.savez_compressed()`, then the arrays can be found with the key of the dictionary bearing their name.

In both cases, the write/read mechanics only work well if the files have been written (and therefore re-read) by NumPy. NumPy's input/output routines are not universal.

If the files come from an external source (matlab, IDL, C/Fortran, etc.), or are written in a standard format (hdf5, wav, jpg, tiff, etc.) one can look for a readout module in `scipy.io`.

In all other cases (and there are many in numerical modelling with public or proprietary codes), it will be necessary to consider a 'proprietary' reading, most often inspired by written documentation.

2. Text files

Opening

For all file types, the first instruction to know is:

```
1 open(filename, mode)
```

`filename` is a string containing the file name.

`mode` specifies whether to :

- write ("w"). The contents are overwritten if the file already exists ;
- readonly ("r"). It is then impossible to write to it;
- read and write ("r+") ;
- write after the existing content ("a" for "append").

By default, files contain only `str`. They are designed to write text.

If you want to read/write binary (non-printable) data, you must add 'b' (for binary) to the opening mode.

For example, mode is "wb" to open a binary file in write mode.

Writing and closing text files (readable and editable)

Writing and closing are done using methods that apply to the opened file object. Only `str` strings can be written.

```
>>> mon_fichier=open('test.dat','w')
>>> print(mon_fichier)
<_io.TextIOWrapper name='test.dat' mode='w' encoding='cp1252'>
>>> mon_fichier.write(4*'ceci est mon test')
68
>>> mon_fichier.write(2*'ceci est mon test \n')  #\n est l'équivalent du <return> du clavier
38
>>> mon_fichier.write('fin de mon test')
15
>>> mon_fichier.close()  # ne jamais oublier de fermer un fichier sinon il est perdu
>>> mon_fichier.closed   # permet de vérifier
True
>>> exit()  # sortie de python
```

```
$ more test.dat
ceci est mon testceci est mon testceci est mon testceci est mon testceci est mon test
ceci est mon test
fin de mon test
```

```
(C:\Anaconda3) C:\Users\Hervé Wozniak\Documents>type test.dat
ceci est mon testceci est mon testceci est mon testceci est mon testceci est mon test
ceci est mon test
fin de mon test
```

```
>>> mon_fichier_text=open('test_texte.dat','w')
>>> mon_fichier_text.write(2)  # écrit l'entier int 2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: write() argument must be str, not int
>>> mon_fichier_text.write(str(2))
1
```

Reading from a text file

Several methods (see definition in the Object Oriented Python chapter) apply to the object 'file' when it has been opened beforehand:

- `read()` reads the entire file; `read(n)` reads n bytes;
- `readline()` reads only one line at a time (defined as the space ending with an `\n`).

```
>>> f=open('test.dat','r')
>>> a=f.read()    # lit l'intégralité du fichier
>>> print(a)
ceci est mon testceci est mon testceci est mon testceci est mon testceci est mon test
ceci est mon test
fin de mon test
```

```
>>> f=open('test.dat','r')
>>> l=f.readline()  # lit une ligne
>>> print(l)
ceci est mon testceci est mon testceci est mon testceci est mon testceci est mon test

>>> l=f.readline()
>>> print(l)
ceci est mon test

>>> l=f.readline()
>>> print(l)
fin de mon test
```

```
>>> f=open('test.dat','r')
>>> for line in f:     # lit toutes les lignes une à une
...     print(line)
...
ceci est mon testceci est mon testceci est mon testceci est mon testceci est mon test

ceci est mon test

fin de mon test
```

```
>>> f=open('test.dat','r')
>>> ll=f.readlines()   # lit toutes les lignes et stocke dans une list
>>> print(ll)
['ceci est mon testceci est mon testceci est mon testceci est mon testceci est mon test
 \n', 'ceci est mon test \n', 'fin de mon test']
>>> type(ll)
<class 'list'>
```

Advice : Best practice

The use of the `with... as...:` block ensures that the file is closed even in case of a read/write error.

```
>>> with open('workfile') as f:
...     read_data = f.read()
>>> f.closed
True
```

3. Binary files

Writing and closing binary files (unreadable and uneditable)

If you want to read/write binary (non-printable) data, you have to append '`b`' to the opening mode.

```
>>> mon_fichier=open('test.dat','wb')
>>> print(mon_fichier)
<_io.BufferedWriter name='test.dat'>
>>>
```

Writing, like reading, is much less simple than in text format.

```
>>> mon_fichier=open('test_binaire.dat','wb')
>>> mon_fichier.write(4*'ceci est mon test')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: a bytes-like object is required, not 'str'
```

Encoding and decoding

The transition from the binary representation on the medium (hard disk) to the representation in memory (RAM) requires decoding (when reading) or encoding (when writing) of the data.

It is a conversion between 1 (or more) bytes and a letter or number that the byte (or group of bytes) represents.

When opening in text mode, the encoding/decoding is imposed: a written/read byte is an octal conversion ⇌ character string (`str`). Several versions of this conversion are possible (*utf-8*, *iso-latin-1*, etc.). By default, Python uses `utf-8`, which translates accented characters.

⊕ *Extra* : ASCII conversion table

This is often referred to as an ASCII conversion table. The `chr()` and `ord()` functions allow you to switch from one representation to another (in base 10).

On parle souvent de table de conversion ASCII. Les fonctions `chr()` et `ord()` permettent de passer d'une représentation à une autre (en base 10).

```
>>> print(chr(65))  # code ASCII decimal vers caractère
A

>>> print(ord('A'))  # caractère vers code ASCII decimal ('101' en octal)
65
```

Fundamental :

In binary mode, the user must provide the encoding/decoding type.

There is nothing in the file to indicate whether you have integers or floats or whatever...

Example :

We use the binary file *test_data.dat* available on moodle.

In a terminal (console) under Linux (not Python!), we use the command `od -t` **xxx** `-N 8 test_data.dat` where **xxx** will take several values.

A group of 8 bytes can be interpreted as follows:

U U U U { G b @ in extended ASCII character mode (-t a);

U U U U { 307 342 @ in strict ASCII character mode (-t c);

125 125 125 125 173 307 342 100 in octal byte by byte (-t o1), so 8 bytes;

040342616755252525252 5 in -t o8;

`38459.854166666664` in double precision float (-t f8)

1.4660155e+13 7.0868506 in single precision float (-t f4), so two 4-byte reals;

1431655765 1088604027 in signed decimal (-t d4 equivalent to normal integers)

85 85 85 85 123 -57 -30 64 in -t d1 (1 byte integers).

55555555 40e2c77b in hexadecimal (-t x)

etc.

⚠ *Warning* :

If you don't know the data structure of a binary file, you are unlikely to be able to read it!

⊕ *Extra* :

The order of the bytes in the read group must also be considered:

little endian: low byte first. This is the case for x86_64, x86 (Intel 8086 and all subsequent ones), which are the most common systems on PCs and Apple today;

big endian: high byte first. Motorola 68000 processors, some IBM processors (Power), SPARC (Sun microsystem), some versions of ARM (for smartphones), etc. Thus, most of the time, high performance computing systems or specialised devices, such as network elements.

In particular, the packets that pass through the Internet are often in big endian, but not always...

struct package

The `struct` package (i.e. `import struct`) must be used to transform the bytes read into data that can be manipulated directly in a variable. Two methods are essential: `struct.pack()` and `struct.unpack()`

`struct.pack(fmt, v1, v2, ...)`: returns a str containing the values `v1`, `v2`, ... encoded according to the `fmt` format (which is a `str`).

`struct.unpack(fmt, buffer)`: decodes `buffer` according to the given `fmt` format. The result is a tuple.

Note also that `struct.calcsize(fmt)` calculates the amount of data that `fmt` represents. Therefore, in pack/unpack operations, `len(string)` should be equal to `struct.calcsize(fmt)`.

fmt

`fmt` is a `str` string that contains:

- a first character which specifies the byte ordering:

Character	byte order
@	native (depending on the host system)
=	native (see https://docs.python.org/3/library/struct.html for subtle things)
>	big endian
<	little endian
!	network = big endian

- as many characters as there are values to be encoded, specifying the `type()` of each variable:

Character	C type	Python type	Size (bytes)	Fortran type
x	pad byte			
c	char	bytes	1	character(len=1)
b	signed char	integer	1	
B	unsigne char	integer	1	
?	_Bool	bool	1	
h	short	integer	2	
H	unsigned short	integer	2	
i	**int**	**integer**	**4**	**integer(kind=2)**
I (i maj)	unsigned int	integer	4	integer(kind=2)
l (L min)	**long**	**integer**	**4**	**integer(kind=4)**
L	unsigned long	integer	4	integer(kind=4)

Character	C type	Python type	Size (bytes)	Fortran type
q	long long	integer	8	integer(kind=8)
Q	unsigned long long	integer	8	integer(kind=8)
n (native mode)	ssize_t	integer		
N (native mode)	size_t	integer		
e	(pas supporté)	float	2	real(kind=2)
f	**float**	**float**	**4**	**real(kind=4)**
d	**double**	**float**	**8**	**real(kind=8)**
s	char[]	bytes		
p	char[]	bytes		
P (native mode)	void *	integer		

Example : Reading test_data.dat on moodle

This file contains data from a WS-3610 weather station. The data format (HeavyWeatherPro v1.1) is explained there : http://www.niftythings.org/HeavyWeather%20History%20Format.txt

We read the first record of the file.

```
# -*- coding: utf-8 -*-
'''
Created on Thu Sep 20 14:54:05 2018

@author: Hervé Wozniak

HeavyWeatherPro V1.1
La Crosse WS-3610 weather station

Each row of data is stored in 56 byte chunks starting from the beginning of
the file (no header).

ROW
OFFSET  Type         Name               Unit
------  ---------    ----------------   -----
00      Double [8]   Timestamp          days from 12/30/1899 00:00:00 (GMT)
08      Float  [4]   Abs Pressure       hectopascals (millibars)
12      Float  [4]   Relative Pressure  hectopascals (millibars)
16      Float  [4]   Wind Speed         meters/second
20      ULong  [4]   Wind Direction     see below
24      Float  [4]   Wind Gust          meters/second
28      Float  [4]   Total Rainfall     millimeters
32      Float  [4]   New Rainfall       millimeters
36      Float  [4]   Indoor Temp        celsius
40      Float  [4]   Outdoor Temp       celsius
44      Float  [4]   Indoor Humidity    %
48      Float  [4]   Outdoor Humidity   %
```

```
52  ULong  [4]  unknown            - (Value is always 0)

Since the timestamp is a double, the fractional part represents fractions of
a day.  This is probably the same type as the Delphi TdateTime type.  More
information about this type can be found here:
http://www.aimtec.com.au/articles/ItsAboutTime/Default.htm
'''

import struct

nr=56
Wdict={'0':'N','1':'NNE','2':'NE','3':'ENE','4':'E','5':'ESE','6':'SE','7':'SSE',
'8':'S','9':'SSW','10':'SW','11':'WSW','12':'W','13':'WNW','14':'NW','15':'NNW'}

f = open("test_data.dat",'rb')
# data structure according to documentation
fmt="<d3fl7fl"
buffer=f.read(nr)
res=list(struct.unpack(fmt,buffer))

print("First record=",res)

'''
create lists and store data as first element
'''
time=[res[0]]
APres=[res[1]]
RPres=[res[2]]
Wspeed=[res[3]]
Wdirection=[Wdict[str(res[4])]]
Wgust=[res[5]]
TRain=[res[6]]
NRain=[res[7]]
ITemp=[res[8]]
OTemp=[res[9]]
IHumidity=[res[10]]
OHumidity=[res[11]]
'''
etc...
'''
```

Extra **: The whole information on formats**

https://docs.python.org/3/library/struct.html#format-characters

Warning **:**

Practical tests have shown that the '@' character gives integers of different sizes (in l or L) between Windows and Linux, but also from one machine to another, and even from one implementation to another. Also, on PCs, it is recommended to use only '<' (little endian) or '>' (big endian) depending on the source of the file.

4. Browsing files (especially binary ones)

tell() method

If `mon_fichier` is the variable pointing to the open file then `mon_fichier.tell()` returns the number of bytes counted since the beginning.

This instruction allows you to know the pointer position while reading a file.

However, it is only reliable for binary files as some types of text file encoding introduce extra bytes.

seek() method

To navigate quickly through a file, it is better to skip a part of a file rather than read it. This is sensitive when navigating through multi-GB files (e.g. video files).

The seek() method can be applied to the open file. If `mon_fichier` is the variable representing the open file, then we can apply the method `mon_fichier.seek(offset, from_what)`.

- `offset` represents the number of bytes you want to move through the file;
- `from_what` indicates the reference point from which offset is applied, using the following convention:
    - 0: from the beginning of the file ;
    - 1: from the current position ;
    - 2: from the end of the file.

This instruction therefore allows you to move forward, jump or move backward quickly in the file, provided you know how to calculate the range of the displacement.

⚠ *Warning* : tell() and seek() with files opened in text mode

The operation of `seek()` is tricky with files opened in text mode.

`from_what` can only be set to 0, so any displacement must be calculated from the beginning of the file.

The offset calculation must be done systematically with `tell()` in order to take into account the bytes 'hidden' by the encoding process. Obviously, this is not always possible. In practice, the combination of `seek()` and `tell()` is only useful for re-reading parts that have already been read, by memorising intermediate positions, which is a fairly infrequent use case.

Last special case: `seek(0,2)` is however accepted and places the pointer at the end of the file.

VII Testing a code: practice and limitations

1. Test levels

« Program testing can be used to show the presence of bugs, but never to show their absence (Edsger W. Dijkstra) »

In software engineering, four main levels of testing are Unit Testing, Integration Testing, System Testing and Acceptance Testing.

Unit testing: individual tests of program components, including functions;

Integration testing: these are carried out to validate the integration of the different parts of the code with each other, in their final operating environment. They enable problems with interfaces between different programs or functions to be highlighted;

In a contractual framework (company, relationship with the client, etc.), the following are added:

- **System testing** (or homologation): verification of the conformity of the application developed with the client's specifications. They are therefore based on the functional and technical specifications.
- **Acceptance testing**: it is the client who, on receipt of the code, carries out tests in its environment.

1.1. Unit testing

At the level of this course, they are mandatory. When developing a function, one must check:

- that the output result corresponds well to the operation that the function is supposed to carry out according to the input values (functional testing);
- that the result does not take an inordinate amount of time compared to the number of operations or calls to the function (performance testing).

a) Functional testing

Method **:**

To carry out a test, one gives oneself input conditions, and calculates the expected result(s) **by hand**. The function under test acts here as a transfer function in signal theory: for a signal form in input, a form is expected in output.

For example, to test a function that calculates the volume of a sphere, we can make the following assumptions:

- if the input is a radius of 1, the output should be 4π/3, or about 4.188790205;
- if the input is 0.62035049, the output must be about 1.0.

The "about" is important as the accuracy of the test will be limited by rounding errors related to the representation of real numbers, including the number π.

```
>>> import repertoire_fonctions.volume as v
>>> v.volume_sphere(1.)
4.1887902047863905
>>> v.volume_sphere(0.62035049)
0.999999995650523
>>>
```

Fundamental :

Is the fact of not obtaining 1.0 but 0.999999995650523 the symptom of a code error or an approximation on the floating representation of real numbers?

It is therefore necessary to define the desired precision beforehand (see next section).

```
>>> import repertoire_fonctions.volume
>>> print(repertoire_fonctions.volume.volume_sphere(1.0))  # math.pi
4.1887902047863905
>>> print(repertoire_fonctions.volume.volume_sphere_sans_math(1.0))  # 3.14159
4.188786666666666
```

The case of inverse functions

If developing a function and its inverse, it is recommended to check that `function(function_inverse()) = 1`

```
>>> import repertoire_fonctions.volume as v
>>>
>>> vol=v.volume_sphere_sans_math(1.)  # pi=3.14159
>>> print(v.rayon_sphere(vol))         # pi=math.pi
0.999999718445299
>>>
>>> vol=v.volume_sphere(1.)  # pi=math.pi
>>> print(v.rayon_sphere(vol))
1.0
```

b) Performance testing (at unit level)

Performance tests can sometimes be run at the unit test level to ensure that the writing is the most optimal in terms of elapsed computing time.

This is not always the best way to proceed as time is most measurable for functions with a large number of instructions or loops with a large number of iterations. It is also possible to simulate a large number of calls.

The many notions of time...

There are several types of time in computing. We will see two of them:

- *output time*: this is human time, the time between the launch of a program and the printing (on the screen or in a file) of the expected result;
- *CPU time*: this is the actual time taken to run the programme. This time is only accessible if the computer and the operating system do nothing else.

Warning :

The operating system is **always** running several programs at the same time to ensure the smooth running of the computer (management of the keyboard, screen, hard disks, RAM, interrupts, etc.). Not to mention the background tasks (email, network, anti-virus, etc.).

Although the CPU time may seem theoretical, it is the time used to compare two versions of an algorithm.

Indeed, the rendering time depends on the state of the computer environment at the time of execution; this state is constantly changing and is not reproducible.

```
import time
def test_time():
    import math
    a=2.0
    for i in range(1000000):
        a=math.sqrt(a)**2

t01=time.time()
test_time()
time.sleep(1)
t11=time.time()
print('time.time=',t11-t01)

t0=time.clock()
test_time()
time.sleep(1)
t1=time.clock()
print('time.clock=',t1-t0)

t000=time.perf_counter()
test_time()
time.sleep(1)
t111=time.perf_counter()
print('time.perf_counter=',t111-t000)

t0000=time.process_time()
test_time()
time.sleep(1)
t1111=time.process_time()
print('time.process_time=',t1111-t0000)
```

Only the `time.process_time()` function now gives a correct CPU time (under any OS) since python version 3.3. For older versions of python, choose `time.clock()`.

```
# exemple Windows
time.time= 1.3233082294464111
time.clock= 1.3159370104004893
time.perf_counter= 1.2995026777576868
time.process_time= 0.296875
```

```
# exemple Linux
time.time= 8.860139846801758  # 8 utilisateurs !
time.clock= 0.18000000000000002
time.perf_counter= 1.1734026479534805
time.process_time= 0.17370565699999996

```

⚠ *Warning* :

Beware of preconceived ideas. For example, *"numpy is always faster than math "*:

```
# numpy remplace math pour sqrt()
time.time= 2.1188647747039795
time.clock= 2.1311964747123966
time.perf_counter= 2.1106610255694704
time.process_time= 1.109375  # 0.29 pour math.sqrt()
```

By comparing several versions of the same function, we can make a decision.

In the previous example we add an `if` condition to check that the variable `a` is positive.

```
    for i in range(1000000):
        if (a>0):
            a=math.sqrt(a)**2
```

```
# sous Windows avec if
time.time= 1.350257396697998
time.clock= 1.3499593813679667
time.perf_counter= 1.3451674921236645
time.process_time= 0.359375  # au lieu de 0.296875
```

Input/Output (I/O) has a big impact on performance. It is preferable to read/write to memory during calculations, and then write to file at the end of the program (or on a long time basis, e.g. every hour for calculations lasting several hours).

```
    f=open('test.dat','w')
    for i in range(1000000):
        a=math.sqrt(a)**2
        f.write(str(a))
    f.close()
```

```
# sous Windows avec I/O
time.time= 3.395256757736206
time.clock= 3.39028105362695
time.perf_counter= 3.390345480238466
time.process_time= 2.40625   # au lieu de 0.296875
```

Beware of the position of for loops. Comparison between "loop inside function" or "loop on function call".

```
def test_time_unitaire():
    import numpy
    a=2.0
    a=numpy.sqrt(a)**2

t0=time.process_time()
for i in range(1000000):
        test_time_unitaire()
time.sleep(1)
t1=time.process_time()
print('time.process_time=',t1-t0)

time.process_time= 1.484375

t01=time.time()
test_time()
time.sleep(1)
t11=time.time()
print('time.time=',t11-t01)

time.time= 2.096216917037964
```

Strictly speaking, the same test should be repeated several times to take into account clock sampling fluctuations and some subtle CPU effects.

Three identical executions:

```
runfile('C:/Users/Hervé Wozniak/Documents/Mandibule/timing.py', wdir='C:/Users/Hervé
Wozniak/Documents/Mandibule')
time.time= 2.1188647747039795
time.clock= 2.1311964747123966
time.perf_counter= 2.1106610255694704
time.process_time= 1.109375

```

```
runfile('C:/Users/Hervé Wozniak/Documents/Mandibule/timing.py', wdir='C:/Users/Hervé Wozniak/Documents/Mandibule')
time.time= 2.157003164291382
time.clock= 2.131606074362935
time.perf_counter= 2.119807897763849
time.process_time= 1.125

runfile('C:/Users/Hervé Wozniak/Documents/Mandibule/timing.py', wdir='C:/Users/Hervé Wozniak/Documents/Mandibule')
time.time= 2.079700231552124
time.clock= 2.0911192822450175
time.perf_counter= 2.100728660711866
time.process_time= 1.078125
```

1.2. Integration testing

It is usually when you integrate the functions developed one by one that you obtain a programme that intends to solve a physical problem.

In the case of scientific programmes, a good integration test can be a physical situation whose analytical solution is known.

Physical tests are usually simple situations because analytical solutions for complicated situations are rarely known.

For example, to test a program that solves ODEs with a Runge-Kutta method, one can calculate the orbit of a body subjected to a central force with the initial conditions of the circular orbit.

2. Limitation of testing with reals: numerical accuracy

We wondered if the fact of not getting 1.0 but 0.999999995650523 when testing `volume_sphere()` was a symptom of a code error or an approximation on the `float` representation of real numbers.

Warning :

One should never test the equality of two `float` but only test if their difference is less than a given precision (set by the user or, better, the machine architecture).

Simple test:

```
>>> 3.3==(1.1+2.2)
False
>>> 1.1+2.2
3.3000000000000003
```

Extra :

In order to completely master the precision of arithmetic with floating-point numbers, as well as the errors that result from this type of arithmetic (infinities, divisions by zero, indeterminations, etc.), there is a module: `decimal`

It is then essential to understand how real numbers are represented in memory.

Precision is mastered but at the cost of a heavy syntax...

https://docs.python.org/3/library/decimal.html

```
>>> from decimal import *
>>> getcontext()
Context(prec=28, rounding=ROUND_HALF_EVEN, Emin=-999999, Emax=999999,
        capitals=1, clamp=0, flags=[], traps=[Overflow, DivisionByZero,
        InvalidOperation])
```

After importing `decimal`, the default precision is 28 decimal digits.

round()

The function `round(variable,nb_significant_figures)` allows comparisons to be made in simple cases:

```
>>> round(3.3,2)==round(1.1+2.2,2)
True
>>> round(1.1+2.2,2)
3.3
>>> round(1.1+2.2,16)
3.3000000000000003
```

Reminder **:**

The number of significant digits for a Python `float` is 15 (equivalent to double precision in other languages). It is therefore illusory to try to test decimals beyond the 15th!