
Python for scientists

Release 1

Pierre Cladé

Nov 06, 2019

Contents

1	Introduction to Python	2
1.1	The Python language	2
1.2	A taste of Python	2
1.3	How to execute Python code	3
1.4	Variable in Python	4
1.5	Functions	5
1.6	Data types in Python	5
1.7	Mutable objects / arguments in functions	12
1.8	Control structure	15
1.9	Accents and non Latin letters	19
1.10	Files and file like objects	21
1.11	Modules	23
1.12	Package	24
1.13	Error and Exception	26
1.14	Assert and test	28
1.15	The Python standard library	29
2	Numerical calculation	31
2.1	Numpy Array	31
2.2	Graphics in Python	41
2.3	Data fitting	43
2.4	Differential equation	46
2.5	Fourier analysis	47

3	Object oriented programming	48
3.1	Object oriented programming	48
3.2	Pauli matrices	55
3.3	Data analysis	55

1 Introduction to Python

1.1 The Python language

- Python is an interpreted language (by opposition to a compiled language like C or Fortran)
- Interpreted language are easier to use than compile language but slower. This is not a problem for scientific calculation because complex algorithms are programmed in C or Fortran.
- There are two versions of Python, the version 2 (currently 2.7) and the version 3 (currently 3.7). There are small differences in the syntax. We are using the 2.7 version. This is still the version installed by default on Linux.
- We strongly advise to install the Anaconda distribution <https://www.anaconda.com/download/>. This distribution was build for scientific calculation. It is available for different platforms (Linux, Mac or Windows).

1.2 A taste of Python

A first example in Python. The goal is to calculate the value of e^x . We will use the following Taylor sum :

$$e^x = \sum_{n=0}^{\infty} \frac{x^n}{n!}$$

We will calculate this sum up to a value n_{\max} of n such that $x^n/n! < \epsilon$. The variable ϵ will set the precision of the calculation (the smaller ϵ is, the better is the precision).

The Python code that can perform this calculation is

```
x = 3.14
epsilon = 1E-6
result = 0
n = 1
term = 1 # Initial value
while abs(term)>epsilon :
    result = result + term
    term = term * x/n
    n = n+1
print(result)
```

We see in this example some specificity of Python : variable are not declared and also the block of instruction in the while loop is determined by the indentation (spaces at the beginning of the line). Each instruction that control a block (while, for, def, if, ...) ends with a colon (':').

1.3 How to execute Python code

This small script can be executed in different ways.

- Copy-paste the example above in a file called `exponential.py`. You can run the script by calling

```
python exponential.py
```

This is the basic way of executing a Python program, in a non interacting mode.

- Run the python command shell simply with the `python` command. You can then write directly commands.
- Using IPython, a more sophisticated python command shell. Run the `ipython` command in terminal. In this shell, you have introspection, tab completion, history, ... You can use the `%run exponentielle.py` to execute the script. The advantage of using a command shell is that you have access to all the variables of your program after its execution. A more aesthetic version of IPython is available using the command `ipython qtconsole` (or in the menu).
- Using Spyder. Spyder combines an IPython qt console and a text editor. This environment was made to look like Matlab. It is very convenient for developing small scripts. To execute our script, simply open spyder, copy the code and run it with the F5 function key. You will see the result printed in the console.

- Using the Jupyter Notebook. This is used to create notebook similar to what is done in Mathematica. Run the command `jupyter notebook`. Your default web browser should open a new tab with the application. You can then open a new project. Write the code in a cell, execute it with the `shift + Enter` keys. Go to the next cell to continue.
- PyDev (Python IDE for Eclipse). Eclipse is an integrated development environment (IDE) used in computer programming. It has a lot of functionality and is very convenient for large projects.

1.4 Variable in Python

A variable is a symbol that represents a value (an object). This object can be used or modified during the execution of the program. In Python, the value of each variable as a specific type (number, string, or your own type).

The creation of a variable is done using the `=` sign. The interpreter evaluate the right hand side (rhs) and assign to the variable the result of the evaluation.

Note: In compiled language like C, usually the variable correspond to a place in memory. You should declare the type of the variable so that the compiler can allocate the memory. In those language, the instruction `a=1` means : copy the value of rhs (in this case 1) to the memory designed by the variable `a`. In C, the memory is created with the variable. In Python, the memory is created with the content. And then is assigned to the variable.

Note: The name of a variable is any sequence of letter or number or `_` which does not starts with a number. Variable are case sensitive

```
a = 1
A = 2
print(a)
_aA12bZz = 35 # OK
1J = 2 # Error
21A = 2 # Error but not the same as above. Why ?
```

1.5 Functions

Of course the exponential function should be defined as a function. This can be done using the following

```
def exp(x, epsilon=1E-6):  
    """ calculate e to the power x """  
    result = 0  
    n = 1  
    term = 1 # Initial value  
    while abs(term)>epsilon :  
        result = result + term  
        term = term * x/n  
        n = n+1  
    return result
```

- The string that follows the definition of the function is a documentation string. It is the public documentation and is available to the user using the `help(function_name)` command. It is not a comment (starting with a `#`). Comments are used to explain your code, public documentation are used to describe how to use your function.
- Of course the `exp` function exists already in the `math` module of python. We will see at the end of this section how to use it.
- The function has an optional argument `epsilon`. There are three ways to call the function

```
exp(1.5)  
exp(1.5, 1E-8)  
exp(1.5, epsilon=1E-8)
```

In the first way, the default value is used. In the second and third way, the optional value is used. The third way is more convenient, especially when many optional arguments are used.

1.6 Data types in Python

This is a brief introduction, look at the official documentation to have more details.

Numbers

There are three kinds of numbers:

- Integer (type `int`). For example `a = 5`, in binary `a = 0b1001`, in hexadecimal `a = 0x23`. Integer have an unlimited size (`print 2**1234`)
- Floating point number. For example `a = 1234.234` or `a = 6.2E-34`. Numbers are **double precision** (64 bits). The relative precision is 52 bits, about 10^{-16} .

```
a = 3.14
print(a == a + 1E-15) # False
print(a == a + 1E-16) # True
```

- Complex number. They are stored as two floating point numbers. For example `a = 1 + 3J`.

Operations between number are : sum `+`, product `*`, difference or opposite number `-`, division `/` integer division `//`, modulo `%` (for example `7%2`), power `(**)`.

Note: By default in Python 2, the division of two integer gives an integer. `print 1/2` gives 0. In order to avoid this, one should force Python to use floats : `a = 1/2.0`. In version 3 of Python, this behaviour has disappeared. In python2, it is possible to use the command `from __future__ import division` in order to use float division

```
>>> from __future__ import division
>>> print 1/2
0.5
```

For complex number, one can access to the real and imaginary parts

```
a = 1 + 3J
print(a.imag)
print(a.real)
print("Norm ", sqrt(a.real**2 + a.imag**2))
```

Boolean and comparison

Boolean have two values : `True` and `False`. Operation with boolean are (lowest to highest priority) : `or`, `and` and `not`. Comparison of number : `<`, `<=`, `==`, `>` et `>=`. In order to test if two values are different, one can use `<>` ou `!=`.

Warning: The `&`, `|` and `~` symbols are used for the `and`, `or` and `not` on integer bit by bit using binary representation (for example `6 & 5` give 4). They also work for booleans but they can be misleading as they have a very high priority

```
print(7==4 | 3==7) # 7==(4 | 3)==7 -> True
```

Note: `exp1 or exp2` is equivalent to `exp1` if `exp1` else `exp2` and `exp1 and exp2` is equivalent to `exp2` if `exp1` else `exp1`. If the result is known from the evaluation of `exp1` then `exp2` is not evaluated. The following example works even if `x<=0`

```
if x>0 and log(x)>4:  
    print("Hello")
```

Strings

Strings can be created using either `"` or `'`

```
s = "Peter"  
s = 'Peter'  
s = "Peter's dog"
```

To create a string with more than one line, use `"""`

```
s = """Hello  
What time is it ?"""
```

One can access to letters of the string in the same way as in a list (see below):

```
s = "Peter"  
print(s[3]) # Element number 3, i.e. the fourth one ; i.e "e"
```

String *concatenation* is obtain using the `+` sign

```
name = "Peter"  
message = "Hello"  
s = message + ", " + name
```

String formatting is used when a variable is inserted in a string. The recommended syntax is the following (it is not recommended to use the old syntax with the `%`)

```
hour = 15
minute = 30
s = "It's {0}:{1}".format(hour, minute)
print(s)
```

You can specify the argument of format using keys instead of position:

```
hour = 15
minute = 30
s = "It's {h}:{mn}".format(h=hour, mn=minute)
print(s)
```

It is possible to specify the way the number is displayed.

For example :

```
from math import pi
print('{0:.5f}'.format(pi)) # '3.14159'
c = 299792458. # Speed of light in m/s
print('c = {0:.3e} m/s'.format(c)) # '2.998e+08'
```

The full syntax (similar to C or Fortran) is described in <https://docs.python.org/2/library/string.html#formatstrings>.

Common methods on string are already implemented

- split
- strip
- join
- startswith, endswith
- lower(), upper()
- comparison (alphabetic order) : 'Peter' > 'John' is True.

Example

```
s = "  Where is Brian? Brian is in the kitchen.  \r\n"
s = s.strip() # string with leading and trailing whitespaces_
↳ characters removed
word_list = s.split() # list containing the words
word_set = set(word_list)
print("The sentence contains {0} different words".
↳ format(len(word_set)))
print("The words are {}".format(' and '.join(list(word_
↳ set))))
```

(continues on next page)


```

if "Brian" in s:
    print("The word Brian is in the sentence")

print(s.upper())

```

List in python

- List creation

```

l = [1, 2, 3, 4]
l = [] # Empty list
l.append(3) # now l==[3]
l.append(4) # now l==[3, 4]
l.insert(0, 3.24+1j) # now l==[3.24+1j, 3, 4]

```

- Modification of an element

```

l[3] = 23

```

- The command `range(n)` creates the list `[0, 1, 2, ..., n-2, n-1]`. This list starts at 0 et contains `n` numbers (it stops at `n-1`).
- A list can contain elements of any type.
- To create a list from an existing list, one can use a for loop

```

l = []
for i in range(n):
    l.append(i**2)

```

- There is a direct way called *list comprehension*

```

l = [i**2 for i in range(n)]

```

- List comprehension can be used to filter a list

```

[log(x) for x in my_list if x>0]

```

- There are two convenient ways to loop through a list :

```

l = [1, 3, 5, "Pierre"]
for elm in l:
    print(elm)

```

```
for i,elm in enumerate(l):
    print(elm, " is the item number ",i," of the list")
```

We have introduced the `for` loop. It iterates over all the element of the list. It works for any sequence (like the string). This syntax is quite different from other programming language.

Note: We have seen that is is possible to add an element in the list. We have used the syntax `l.append(elm)`. This is object oriented programming. The name of the object is followed by a dot and the name of the function used (this function is called a *method*) with its argument. Here, the `append` method is used to add `elm` to the list `l`.

Furthermore, note that the object `l` point to the same list. This list is modified. It's like adding a page to a book (or a ring binder). This behaviour is different from the one described in the previous section with the `format` method on string. In this case, a new string is created and returned by the `format` method. The original string is not modified. The method `append` applied to a list, modify the list and return nothing.

Tuple

Tuples are used to collect few objects together. They look like list, but are not mutable (they cannot be modified)

```
# tuple with first name, name, birthday, and age.
someone = ('Jean', 'Dupont', 'July 13th, 1973', 38)

print('Name :', someone[1])
```

Tuple are used when a function returns more that one value

```
def fonction(x):
    return x**2, x**3

a = fonction(4) # a is a tuple
a,b = fonction(4) # a->16 ans b->64
```

Dictionary

In a list or a tuple, elements are indexed by integers. In a dictionary, they are indexed by a key. Usually the key is a string or a number. The example of the previous section is more conveniently implemented using a dictionary

```
someone = {"first_name": "Jean", "name": "Dupont",  
           "birthday": "Jul 13th, 1973", "age": 38}  
print("Name :", someone['name'])
```

To loop through a dictionary, use the following

```
for key, val in dct.items():  
    print(key, val)
```

Set

Used to represent an unordered set of different elements. For example

```
a = set([1, 2, 3])  
b = set([3, 5, 6])  
  
c = a | b # union  
d = a & b # intersection
```

Example

```
pwd = input('Enter a password with at least one punctuation')  
punctuation = set("?,. ;: !")  
if (punctuation & set(pwd)) == set():  
    print("The password should contain at least one_  
↪punctuation")
```

Index in Python

We have seen the list, str and tuple data type. They can be indexed. One can get an item using square brackets. In Python, the first element is addressed by 0. Negative indexed are from the end (l[-i] is equivalent to l[len(l)-i]). Therefore, the last element is addressed by -1.

Slices are used to extract a sub part of a sequence. For example :

```
s = "Hello World"
print(s[0:5])
```

The slice determined by `start:stop` will return a new object of the same type containing elements from `start` to `stop` **excluded**. The length of the new object is `stop-start`. One can use negative numbers. For example `s[1:-1]` will return the string without the first and last letters. If `start` or `stop` is omitted, then it is replaced by 0 or the length of the object. For example `s[1:]` will return a string without the first letter.

Note that the `start:stop(:step)` notation is a shortcut to index with a `slice` object. The following are equivalent :

```
s[0:5] my_slice = slice(0, 5) s[my_slice]
```

None

There is a specific object in Python `None`. This is the object returned by function that ended without a return statement. The `None` object is often used as a “flag” to set a specific behaviour. The following `log_data` function (to create a log file) can either display the result or save it to a file depending on the value of the `file` argument.

```
def log_data(txt, file):
    if file is None:
        print(txt)
    else:
        file.write(txt)
```

1.7 Mutable objects / arguments in functions

In Python, a variable is simply an alias for an object that exists in the memory of the computer. An object may have more than one alias.

```
a = 3 # Python creates the object #1 containing 3.
b = a + 4 # Python creates the object #2 containing 7
c = a # The symbol c point to object #1
a = b # The symbol a point to object #2
c = 3.14 # The symbol c point to a third object. There is no
↪ way
      # to point to object #1. Python can delete it.
```

There are two kinds of objects : objects that can be modified (mutable) and those that cannot. List, dictionary or sets are mutable object but numbers or string are not. Let us look to the following example

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

- Line 1 : creation of list#1
- Line 2 : b point to the list#1
- Line 3 : python display item 1 of list#1 : it's 3
- Line 4 : list#1 is modified
- Line 5 : python display item 1 of list#1 : it's 4
- Line 6 : creation of list#2. Variable a point to list#2 and b to list#1
- Line 7 : python display item 1 of list#1 : it's 4

The instruction `a[1] = 4` and the instruction `a = [5,6,7,8]` are very different. In the first case, the object is modified in the second case a new object is created and linked to variable a

When a variable is the argument of a function, the mechanism is the same. Inside the function, the variable point to the same object. If this object is modified by the function, it is modified !

```
def exemple(arg):
    print(arg[1])
    arg[1] = 4
    print(arg[1])
    arg = [5,6,7,8]

a = [1,2,3,4]
exemple(a)
print(a[1])
```

Exercise : which number is displayed ?

```
a = [1,2,34,45]
b = a
```

(continues on next page)

```
c = a[1]
a[2] = 1
a[1] = 5
print(b[2]+c)
```

Local and global variable

Inside a function a variable is either local or global (in Python 3 a new concept of non local variable instead of global as been introduced, but for the following it is equivalent). By default, if there is an assignment inside the function (or if the variable is an argument), it is local. When there is no assignment and the variable is not an argument, the variable is global. By default, when the variable is global, one can read the object pointed out by the variable, one can modify it (but we do not assign a new object to the variable).

- Global variable

```
a = 1
def example():
    print(a)
```

- Local variable

```
a = 1
def example():
    a = 5
    print(a)
```

- Bug (python cannot read the local variable)

```
a = 1
def example():
    print(a)
    a = 5
    print(a)
```

There is a general rule : in a function, a global variable should be a well identified constant (like pi) or a well identified object (like a function). Everything else should be an argument of the function. If your function has too many arguments, consider using an object.

The global instruction in Python

Forget it !

1.8 Control structure

For loop

We have introduced the `for` loop with the list. The `for` loop works with object such as list, dict, set, string. With those object, the data are stored in the object and the `for` loop extract those data. The `for` loop work actually in the more general situation of iterable object (iterator). An iterator is an object that can produce the data for the `for` loop. The data can be calculated on the fly. For example, in python 2, `range` returns a list and `xrange` an iterator. In python 3, the `range` function returns an iterator.

If you want to iterate over two or more objects at the same time, use the `zip` command. For example :

```
X = [1,3,4,7]
Y = [3,5,1,2]
for x,y in zip(X,Y) :
    print(x,y)
```

If you want to the get the index, then use the `enumerate` instruction :

```
X = [1,3,4,7]
Y = [3,5,1,2]
s="Point number {i} at coordinate ({x},{y})"
for i, (x,y) in enumerate(zip(X,Y)) :
    print(s.format(x=x, y=y, i=i))
```

If you have a list of numbers and you want to iterate on intervals, use the following

```
for start, stop in zip(l[:-1], l[1:]):
    print('length = {}'.format(stop-start))
```

Usually you don't need the index, and we advise not to use the following :

```
for i in range(len(book)) :
    print(book[i])
```

but, the more human syntax,

```
for page in book:
    print (page)
```

One can exit from a for loop using the `break` instruction and terminate one iteration using the `continue` instruction. The `else` occurs when the loop ends “normally” (without `break`):

```
from math import ceil, sqrt
for p in xrange(int(ceil(sqrt(m)))):
    if p<=1:
        continue
    if m%p==0:
        is_prime = False
        break
else:
    is_prime = True
```

The following example will be much simpler using a separate function. Indeed, it is possible to leave a function any time using the `return` instruction.

```
def is_prime(m):
    for p in xrange(2,int(ceil(sqrt(m)))):
        if m%p==0:
            return False
    return True
```

Generators

It is possible to create your own iterator using a function like syntax: this is a generator. The idea is to replace the `return` by a `yield` statement. Each time the function encounters a `yield` it will return the value to the for loop.

Examples :

```
def simple_generator(): yield 1 yield 2
```

```
def concatenate(liste1, liste2):
```

```
    for elm in liste1: yield elm
```

```
    for elm in liste2: yield elm
```

```
def matrix_index_generator(N1, N2):
```

```
    for i in range(N1):
```



```
for j in range(N2): yield (i, j)
```

While loop

There is not much to say about while loop. Example

```
while condition:
    stat1
    stat2
```

if/else

Example :

```
if cond1:
    stat1
    stat2
elif cond2:
    stat3
    stat4
else:
    stat5
```

Function

Optional arguments

Look to the following function (from `scipy.integrate` module):

```
def quad(func, a, b, args=(), full_output=0, epsabs=1.49e-8,
        epsrel=1.49e-8, limit=50, points=None, weight=None,
        wvar=None, wopts=None, maxpl=50, limlst=50):
    """
    Compute a definite integral.

    ...
    """
```

Using this function, you can change in an explicit way the relative precision of the calculation :

```
quad(exp, 0, 10, epsrel=1E-3)
```

If all your optional arguments are in a dictionary, you can pass them directly using the following syntax :

```
option = {"epsrel":1E-3, 'limit':100}  
quad(exp, 0, 1, **option)
```

Lambda function

Use to define a function with an expression. It is very usefull when the function is an argument of a function. The syntax is `lambda arg1, arg2, ...: expr`.

Using the previous example :

```
quad(lambda x:exp(-x**2), 0, 1)
```

Variable length argument list

It is possible to define a function with a variable number or arguments. The general syntax is :

```
def my_function(a, b, *args, **kwd):  
    print(args)  
    print(kwd)  
  
my_function(1,2,3,4,e=3)  
# args = (3,4)  
# kwd = {'e':3}
```

Inside the function, `args` is a tuple containing unnamed arguments and `kwd` is a dictionary containing the named arguments.

It is also possible to call a function by splitting a tuple or a dictionary to make them arguments (or keyword arguments for a dictionary)

```
def f(a, b, c=2):  
    print(a, b, c)  
  
arg = (1, 2)  
f(*arg)
```

(continues on next page)

```
kwd = {'b':1, 'c':4}
f(2, **kwd)
```

A typical example of variable length argument list occurs when one wants to pass arguments from one function to another directly. For example, the `erf` function can be calculated by integrating e^{-x^2}

```
def erf(x, **kwd):
    return quad(lambda x:exp(-x**2), 0, x, **kwd)

print(erf(1, epsrel=1E-4))
```

1.9 Accents and non Latin letters

It can be useful to use letters with accent or non Latin letter in Python. The behaviour is very different between Python 2 and Python 3.

ASCII

The old way to use non English Latin letter was by extending the ASCII. This standard give a number between 0 and 127 to each English letter and usual symbol. The number for 128 to 255 were defined in local standards (for example the Windows-1252 for computer in western Europe - or ISO 8859-1 for linux/unix). Of course local standard are not convenient and from the beginning of the 1990s, the unicode standard was developed.

Unicode

“Unicode is a computing industry standard for the consistent encoding, representation and handling of text expressed in most of the world’s writing systems.” (wikipedia). Basically, unicode give a number to each written letter of all the known language (from the language of mathematics to Egyptian hieroglyphs). For example, the number associate to the “Planck constant over two pi” (\hbar) is 8463. A unicode string can therefore be represented by a list of integers. In order to display such a string, the computer need the correct font. A font, will usually cover only a specific part of the unicode standard.

The main problem is to store (encode) a unicode string in a file. Because unicode number are smaller than 2^{32} , an easy way will be to store each character in 32bits (4 bytes). This will be very convenient, however it will not be compatible with ASCII

standard in which one character corresponds to one byte. The UTF-8 encoding solves this problem : if the byte is smaller than 127, it indicates an ASCII character. If not, you have to look to more than one byte to get the unicode character.

Unicode in Python 2

In Python 2, string are ASCII strings. In order to use unicode string, you need to specify it with a leading `u`. Further more, a file is assumed to be ASCII. If your file is a unicode file, you need to specify the encoding at the beginning of the file. This is done in the first or second line of the file with pseudo comment string of the form `# -*- coding: encoding_name -*-`. The default encoding is `utf-8` for linux et probably `latin-1` for windows.

Examples

```
# -*- coding: utf-8 -*-

s = u"Pierre Clad  "
print s

hbar = 6.62E-34/(2*pi)
print u"\u210F = {0:5.2e}".format(hbar)
```

The function `unichr` allows to programatically convert a number to the corresponding unicode letter. The function `ord` to the opposite.

Unicode in Python 3

In Python 3, string are by default unicode string. Furthermore, not only strings are unicode, but also the python code can also use unicode. The following code will work using python 3

```
 $\alpha$  = 1/137.035990
print('The value of the fine structure constant is {0}').
    format( $\alpha$ )
```

We strongly advise to limit the use of unicode to string and not to the python code as in the example above. Indeed, two different unicode characters can be displayed the same way

```
A = 1
A = 2
print(A)
print(A)
```

In python 3, use `chr` to programatically convert a number to the corresponding unicode letter

In python 3, unicode string are used to represent text strings. In order to represent a list of bytes, a new type has been introduced, called byte strings. They can be created directly with the `b` prefix

```
b'This is a byte string'
```

Byte strings are used when dealing with communication at low level (TCP/IP socket or communication with an instrument). To convert a string to bytecode and vice versa , use the encode and decode methods :

```
"α = 1/137".encode('utf-8') b'xcexb1 = 1/137'.decode('utf-8')
```

1.10 Files and file like objects

Files

Example

```
f = open('/tmp/test', 'w', encoding='utf-8') #
f.write('Bonjour')
f.close()

f = open('/tmp/test') # default is reading
print f.readlines()
f.close()
```

The `open` function creates a file object. The second argument is used to say if we want do write ('w') or read ('r') the file. The file objects has several methods :

- `write(str)` : To write one string in the file
- `read` : To read all the file. `read(n)` to read a given number of characters.
- `readline` : read one line of the file
- `realines` : return a list with one item per line.

With statement

The with statement is a relatively new syntax in Python

```
with open("x.txt") as f:
    data = f.read()
    do_something
```

The with statement will take care of closing at the end of the instructions, even if an error occurs.

File like objects

Many interface are similar to files and works the same way.

For example, you can open a url using the urllib and read its content

```
import urllib

#Open Sherlock Holmes an count in how many lines there is the
↪word Holmes
url = "http://www.gutenberg.org/cache/epub/1661/pg1661.txt"
file = urllib.urlopen(url)
cpt = 0
for line in f.readlines():
    if 'Holmes' in line:
        cpt += 1

print 'The book contains {0} times the word Holmes'.
↪format(cpt)
```

Common text based interface to instrument (like RS232, VISA, GPIB, ...) works also like a file :

```
import visa

inst = visa.GPIBInstrument('GIPB::19')
inst.write('*IDN?\r\n')
print inst.readline()

import serial

verdi = serial.Serial(port='COM7', baudrate=19200)
```

(continues on next page)

```
verdi.write('?P\r\n')
power = verdi.readline().strip()
```

1.11 Modules

At the beginning of this chapter, we have created a function to calculate the exponential. How to use this `exp` from another file ? A solution could be to execute the file from the script. However, there is a much more convenient technique in python. The function can be imported using the command

```
from exponentielle import exp
```

We do not put the `.py` file extension. Using modules has many advantages :

- If the module was already imported, python does not execute the file a second time.
- One can choose what we want to import in the module.
- Python can automatically look for different paths to find the file.

Python is not a general purpose language. All the functions that are specific are defined in a module. Mathematical functions (and constants) are in the `math` module.

They can be imported in different ways :

- Import of the module

```
import math
print math.sin(math.pi/3)
```

- Import of specific objects in the module

```
from math import sin, pi
print sin(pi/3)
```

- Import of all the objects defined in the module

```
from math import *
print sin(pi/3)
```

The first method is more verbose because we have to specify the module name every time. However, we have to use this method in our example if we want to compare the `exp` function from two different modules.

In general, the second method should be preferred over the third one. Indeed, when all the functions are imported it is not easy then to know from which module the function was imported. Furthermore, this import can override existing function (for example, there is an `open` function in the module `os`. Therefore the command `from os import *` will override the standard `open` function).

The `import *` can be used for module with **well known functions** that are used **many times** in a program. This is the case for the `math` module.

1.12 Package

A package is a group of module (in other language it is called a library).

Many package are available on the Pypi web site. The `pip` shell command can be used to download and install the package on your computer. For example if you want to install the PyDAQmx package, simply run the command :

```
pip install PyDAQmx
```

If you don't have root permission, then use the `--user` option.

If the package is not available on Pypi, then the usual way consist in downloading the package and then install it by running the setup script :

```
wget https://pypi.python.org/packages/source/P/PyDAQmx/  
↳PyDAQmx-1.3.2.tar.gz  
tar -xvzf PyDAQmx-1.3.2.tar.gz  
cd PyDAQmx-1.3.2  
python setup.py install --user
```

A package is a collection of modules and sub-packages. In order to create a package, you simply have to put the files into the same directory (called for example `my_package`). Python will know that this directory is a package if there is a file called `__init__.py` in the directory. This file can be empty.

In order to import the modules, you can then do `import my_package.mod1` or `from my_package import mod1,...`

The `__init__.py` is a python module whose content is imported by `from my_package import ...`

Python will be able to import “my_package” :

- If your current working directory is the directory containing the “my_package” directory

- If this directory is in your search path. You can dynamically add a directory into the search path using the following :

```
import sys
sys.path.insert(0, 'directory/containing/my_package')
```

- If your directory is in the the “PYTHONPATH” environment variable.
- If you have installed your package (see instruction below).

Inside a package, it is common to import modules or sub-packages, for example in the `__init__.py` file. This is a local import. You should indicate to python that an import is local by prefixing the module name with a dot. For example, in the `__init__.py` of `my_package`

```
from .mod1 import my_function

from . import mod1
mod1.my_function()
```

If you are in a sub-package, then you can import from the parent package using two (or more) dots.

It is easy to create your package and distribute it. You have to write a `setup.py` file containing some informations about your package. Then Python will take care of everything. The `setup.py` file should be in the same directory as the “`my_package`” directory. A minimal example of a `setup.py` file is the following :

```
#!/usr/bin/env python
# -*- coding: utf_8 -*-

from distutils.core import setup
__version__ = "alpha"

long_description="""This is a very nice package

"""

setup(name='my_package',
      version=__version__,
      description='A very nice package',
      author=u'François Pignon',
      author_email='francois.pignon@trucmuch.fr',
      url='',
      packages=['my_package'],
    )
```

Then you can execute the following commands :

```
python setup.py install # Install the package
python setup.py sdist # create a tar.gz or zip of your package


pip install -e . --user
```

The last command is very useful for developers. It installs the project in editable mode, which means that modification of the project files will be taken into account (a simple install will copy the files to another place).

The `twine` library allows you to publish your packages on Pypi in an easy way :

```
twine upload dist/*
```

If you want to make test, then you can use the `test.pypi.org` instance :

```
twine upload --repository-url https://test.pypi.org/legacy/  dist/*
```

1.13 Error and Exception

Error

You should read and **understand** errors of Python. Python always indicate the line of the error, except when there is a syntax error (see examples bellow).

Examples of common error

```
from math import sin, sqrt

a = sin[1]

b = cos(2)

a,b,c = 2,8,4
Delta = b**1 - 4*a*c
root1 = (-b + sqrt(Delta)/(2*a)
root2 = (-b - sqrt(Delta)/(2*a)

mylist = [1,2,34]
print mylist(2)

if 1==1:
```

(continues on next page)

```

    print 'Hello'
    print 'World'

if 1==1:
    print 'Hello World'

```

Try ... except

In Python, it is possible to catch an error and have a specific behaviour. This is usually a very efficient way of dealing with specific cases. For example, if you want to take the sqrt of a negative number, python will throw a `ValueError` exception. You can then handle the specific case :

```

a = -1

try:
    b = sqrt(a)
except ValueError:
    b = 1j*sqrt(-a)

```

Of course this example can be solved with a `if else` statement. Let's look to the following :

```

filename = "/tmp/balbla"

if not os.path.exists(filename):
    print 'The file does not exists'
else:
    print open(filename).readlines()

```

Or the following :

```

filename = "/tmp/balbla"

try:
    print open(filename).readlines()
except IOError:
    print 'The file does not exists'

```

The second example is more safe : the execution of the program can be interrupted at any time by the OS to run another process. What happens if the file is deleted between the test and the open? The second example is also faster : the test is always

performed by the open function, there is no reasons to make it twice. There is no overhead in using a try.

Raise exception

It is possible to create you own exception.

Example

```
from math import sin, asin

def snell_descartes(theta1, n1, n2):
    sin_theta2 = n2*sin(theta2)/n1
    if abs(sin_theta2)>1:
        raise Exception('Total internal reflection not allowed
→ ')
    return asin(sin_theta2)
```

Or using try/except :

```
from math import sin, asin

def snell_descartes(theta1, n1, n2):
    sin_theta2 = n2*sin(theta2)/n1
    try:
        return asin(sin_theta2)
    except ValueError:
        raise Exception('Total internal reflection not allowed
→ ')
```

You can of course use the `snell_descartes` in a try except. It is possible to create your own error (like `IOError`, `ValueError`).

Exercise : Create a function `second_order_equation` that solves the equation $ax^2 + bx + c = 0$. This function will have an optional argument `allow_complex` that if true return complex roots else raise a `ValueError` exception.

1.14 Assert and test

A very good practice in programming is to always test your functions. Some programming guide says that you should write your test before writing your functions. We advise at least to write them at the same time. Specific modules exist on Python to deal with automatic tests (like the `unittest` module). This is beyond the scope of this introduction.

Let us look at an example. Imagine that you want to write your own `sqrt` function. You should test that it works for positive number and give an error for negative. The best way to test is to use the `assert` statement (see below).

```
from math import sqrt

def my_sqrt(x):
    return sqrt(x)

if __name__=="__main__":
    import random
    x = random.random()
    assert my_sqrt(x)**2==x, "The my_sqrt functions does not\
return the square root"

    try :
        x = sqrt(-1)
    except ValueError:
        pass
    else:
        raise Exception('The my_sqrt function does not raise\
an error for negative numbers')
```

Testing you code on simple examples helps you to build large projects. By writing tests, you can always check that there are no side effects due to modifications you are doing on the code.

Exercise : Create a full test for the `second_order_equation` equation written above.

1.15 The Python standard library

The Python standard library (<https://docs.python.org/2/library/>) contains tens of modules that allows to perform usual operations. Below is a list of usefull modules for every day programming. It does not include modules specific to scientific calculation because they are not part of the standard library.

- `string` — Common string operations
- `re` — Regular expression operations
- `datetime` — Basic date and time types
- `math` — Mathematical functions

- `shutil` — High-level file operations
- `os` — Miscellaneous operating system interfaces
- `logging` — Logging facility for Python
- `email` — An email and MIME handling package
- `sys` — System-specific parameters and functions
- `urllib` — Open arbitrary resources by URL
- `time` — Time access and conversions

Regular expressions

A regular expression is a string that describe a set of possible strings. For example on unix shell, the `*` mean any sequence of characters (`cp *.jpg` directory will copy all the jpeg files to the given directory).

In the `re` package of python, there is specific syntax that should be used to describe regular expression. This syntax is described in the documentation <https://docs.python.org/2/library/re.html>.

For example, I want to parse a date given by a string like “01-02-2013”. In English, this is described as two digits followed by a `-` then two digits another `-` and four digits. The `\d` should be used to describe a digit. The date will follow the following regular expression : `\d\d-\d\d-\d\d\d\d`. This can match exactly the previous string. But if you want to allow more or less digits (like “1-2-2015”) you can use the `+` : the regular expression `\d+-\d+\d+` means one or more digits followed by a dash and so on.

The `re` package allow you to test if a string matches a regular expression

```
import re

date = '11-02-1980'
date2 = "Feb. 11, 1980"

print re.match("\d\d-\d\d-\d\d\d\d", date) # return something
print re.match("\d\d-\d\d-\d\d\d\d", date2) # return nothing
```

Of course it is very useful once a string match the regular expression to extract the relevant information. This can be done by grouping the pattern in parenthesis. And then use the `groups` method of the result.

```
import re

date = '11-02-1980'

result = re.match("(\\d\\d)-(\\d\\d)-(\\d\\d\\d\\d)", date)
if result is not None:
    print result.group(1) # print the first group
```

Or, using a more explicit syntax:

```
import re

date = '11-02-1980'

result = re.match("(?P<day>\\d\\d)-(?P<month>\\d\\d)-(?P<year>
↪\\d\\d\\d\\d)",
                    date)
if result is not None:
    print result.group('day')
```

This is a very short introduction to the regular expression. Look to the documentation for more detailed.

Exercise : A directory contains a number of file likes IMAGE001.jpg, IMAGE002.png and so one. Create a function that check if a file name satisfy the following pattern and return the image number (the extension are case insensitive and given in a liste : .jpg, .png, .pdf, .jpeg). Create a function which from a given directory returns next available number (use `os.listdir`).

2 Numerical calculation

2.1 Numpy Array

Numpy array is not a native Python datatype. It is part of the `numpy` package. This is the datatype that should be used for large numeric data analysis or numerical calculation. They differ from list : the size of a numpy array cannot be modified and all the data inside an array are of the same type. Those constraints allow much faster computation. We will mainly use 1D and 2D array. They can however be of any dimension. Numpy arrays allow users to perform calculation in a way similar to Matlab or Scilab.

Some examples :

```

from numpy import *

a = array([4,23,3]) # 1D array
print(a[2])
a[2] = 13

b = array([[1,2,3],[2,3,4]])
print(b)

```

In order to use array, we should first import the numpy package. The array command can be used to create array from any iterable (for example a list). A list of number will create a 1D array. A list of lists having the same size will create a 2D array.

One can create an array containing the 0, 1, 2, 3, ... sequence using the arange command. This command has the same arguments than the range command (arange(end), arange(start, end), arange(start, end, step)). Items of an array can be changed using the same syntax as list. The size of an array can be obtained using the shape property. This property returns a tuple.

```

a = arange(10)
print(a[2])
a[3] = 2
print(a.shape)

```

Mathematical operation on array

One of the main advantage of using numpy array is that it is possible to perform usual operation, item by item, without using a loop. This holds for binary operation (sum, product, difference, ...) and also for usual mathematical operation (sin, cos, tan, exp, log, ...). In order to work with array, those function should be imported from the numpy module (and not the math module). Global functions on array, such as sum, prod, max, min, mean, std can be used either as method of the array or as function of the numpy module.

For example :

```

from numpy import *

a = arange(10)
print(a**2 - sin(a) )

even_numbers = arange(2,10000,2)
print(2*prod(even_numbers**2)/prod(even_numbers**2-1) )

```

(continues on next page)


```
x = linspace(0,5, 1000)
# using function
print(min(x+2/x))
#using methods
a = x + 2/x
a.min()
```

In this example, we have used the function `linspace(start, end, nb_steps)`. This function creates an array with `nb_steps` values between `start` and `end`.

Note also that the `sin` function is imported from the `numpy` library. This function takes an array and returns an array of the same size.

Note that the builtin functions `sum`, `max` and `min` are different than the one imported from `numpy`. If you use the builtin ones, you lose the advantage of using `numpy` array.

It is possible to create a function that works with an array from any function using the `vectorize` function of the `numpy` package :

```
def myfunction(x):
    ....
    return ....

myfunction = vectorize(myfunction)
print(myfunction(linspace(0,1)))
```

Data type

In order to get the data type of the array, one can use the `dtype` attribute. It is possible to set the data type of an array when it is created using the optional `dtype` parameter.

For example :

```
a = arange(10)
print(a.dtype)
a = arange(10, dtype='float64')
print(a.dtype)
```

`Numpy` array have more data type than `Python`. For example, floating point number can be saved as 32, 64 or 128 bits (`float32`, `float64`, `float128`), integer are usually limited to 64 bits (this depends on your system).

The data type of an array cannot be modified. If an array is defined as an integer array one cannot put inside a float number. This number will be converted to an integer (without warning).

```
a = arange(5)
a[1] = 3.141592
print(a[1]) # a[1]==3
```

Of course, you can create a new array from an existing one with the data type you want.

Array indexing

One can access to array items the same way than for list (item number starts at 0, one can extract an array using slices : `a[start:end:step]`, negative numbers start from the end). Numpy array can be modified. This can be done element by element or on a group of elements.

For example :

```
a = arange(10)
a[3] = 34
a[::2] = 0 #This is not possible for a list
```

Be aware that when an array is extracted using slices, it share the same memory space than the initial array. The following code will modify the array `a`.

```
a = arange(10)
b = a[::2]
b[:] = 0
print(a)
```

It is possible to extract an array from an array using a boolean array. For example :

```
a = array([23,25,10])
condition = array([False, True, False])
print(a[condition])
```

This is very useful because it is possible to create a boolean array using comparison and logical operators.

For example :

```

a = arange(10)
cond = (a>=5)
print(cond)

a[cond] = 5

b = arange(100)
cond = (b%2==0) & ~(b%5==2)
print(b[cond])

```

Logical operators on array are '&' for and, '|' for or and '~' for not. The operators `and`, `or` and `not` do not work with array (indeed they are not operator!).

2D array and linear algebra

Here are some examples of 2D array

```

from numpy import *

a = array([[1,2],[1,9]])

print(a[:,0]) # Column number 0
print(a[1,:]) # Line number 1

```

2D numpy array can be used as matrices. The matrix product **is not** the `*` operator (which makes the product element by element). In order to perform true product one can use the `dot` function of the numpy package. The `linalg` package contains also many function of linear algebra (see the documentation).

```

import numpy as np
from numpy import linalg

a = np.array([[1,2],[1,9]])
b = np.array([[3,1.],[3.14,6.02]])
c = np.dot(a,b)

print(linalg.eig(c)) # eigen vectors and eigen values

```

We briefly mention here the `matrix` package of numpy. Using the `matrix` type, one can create matrices or vectors. The product is done conveniently using the `*` operator.

Useful functions and methods

Array methods :

- `.sum()`, `.prod()`, `.mean()`, `.std()`. Those method have an `axis` option that perform the operation only on one axis.
- `.reshape(shape)` : change the shape. The shape argument is a tuple. The total number of items should be unchanged
- `.flatten()` : makes a 1D array
- `.transpose()`

Array creation :

- `arange` : similar to the range command.
- `linspace` : the `linspace(start, stop, num=50, endpoint=True)`. The `endpoint` option set weather or not stop is the last sample (or the next one).
- `zeros(tpl)`, where `tpl=(n1, n2, ...)` is a tuple containing the shape of the array.
- `ones(tpl)` : similar to zeros.
- `rand(nx, ny, ...)` : random number between 0 and 1. The shape will be `(nx, ny, ...)`.
- `X, Y = meshgrid(x, y)` : created 2D array X and Y such that `X[i, j] = x[j]` and `Y[i, j] = y[j]`

Saving numpy array

Ascii format

One can save and read 2D numpy array in a text file using the `savetxt(fname, tableau)` and `loadtxt` commands. This is a universal method (ascii file can be read by many applications). However it is slow and this method can lead to rounding error when numbers are converted to text.

The `loadtxt` can be used to read almost any file produced by another application. Look to the function documentation for the different options. For example one can set the separator in order to read a CSV (comma separated values) file.

```
loadtxt(filename, delimiter=',')
```

Binary format

The `load` and `save` function of the `numpy` module can be used to store directly numpy array in a file. This method is a direct copy of the memory. It is fast and uses less memory space. However it does not provide direct compatibility with other application.

It is possible to convert binary data of other application to array. If the data is written into a file, one can use the `fromfile` function. If the data comes directly from a function (for example, if it comes from a CCD camera or a digital scope), one can use the `frombuffer` function. They are universal functions that can be parametrized to read any format (signed/unsigned integer, big or little endian, 16 or 32 bits, ...).

Understanding numpy array

It's useful to understand how numpy arrays are stored in the memory. Because the size and the data type of the array are not mutable, the total amount of memory is fixed. The place where the data are stored is a contiguous block of memory, in a way similar to array in C or Fortran. One can access to this memory block using the `data` attribute of an array : this will return a buffer (basically a string of bytes). For example

```
from numpy import *
a = arange(5, dtype=int16)

print(len(a.data)) # 5*2 = 10 bytes
assert a.data[0]==chr(0) # The first byte is 0
```

The numpy array does not only contain a pointer to the starting address in the block of memory, but also informations on how to retrieve elements of the array. Each element of the array starts at a specific position in the block of memory. This position is a linear combination of the indexes in the array. For example, in a 10x20 2D array of `int32`, the position of element (i,j) will be $20 \times 4 \times i + 4 \times j$. The two coefficients (80, 4) are the only parameters you need in order to read the array. They are obtained from the `strides` property of the array.

It is important to know that you can create two arrays that share the same memory block, but with a different starting address and a different `strides`. This is the mechanism used when slicing an array or reshaping the array.

```

from numpy import *
a = arange(10, dtype=int16)
print(a.strides) # (2,)
b = a[::2]
print(b.strides) # (4,)

a = array([1,2],[3,4])
I,J = a.strides
a.strides = (J,I) # transposition

```

This mechanism is very fast and efficient because no copy of the data is performed. However, we should remember that modifying the content of one array will modify the content of all arrays that share the same memory block.

We have seen in the previous section that we can use operation on array (like $+$, $*$, $/$, ...). They will perform the operation element by element and return a new array. Those operation can also be used between an array and a scalar ($2*a$). Numpy has a mechanism called broadcasting that makes possible, under certain circumstance (see docs.scipy.org/doc/numpy/user/basics.broadcasting.html), to work with two arrays of different sizes. For example an array of size $m \times 1$ can operate with an array of size $1 \times n$ to form an array of size $m \times n$. The arrays are broadcast to have the correct dimension by duplicating the column or row. Indeed, there is no real duplication, simply numpy set the strides of the broadcast dimension to 0.

Beyond numpy

As we have seen, numpy array should be used when dealing with large data set. **You should not use** loops with numpy array. When there is a way to not use loop, execution can then be almost as fast as with a compiled language.

If you really have to use a loop and if this loop is the bottle neck of you program, then you can consider using alternative solution based on compilation. There are two ways : either by building your library directly in the language you choose, and then link it to python. In C, you can use the ctypes module and in Fortran, the F2PY module. The other, and more convenient, alternative is to use a tool to convert your function to a compiled language. This is how the Cython module work.

Cython

The Cython package is used to convert a Python function to a C function which is compiled.

Let us see a simple example. We want to calculate the value of π using the following formula :

$$\frac{\pi}{4} = \sum_{k=0}^{\infty} \frac{(-1)^k}{2k+1}$$

This formula is chosen because it is probably the slowest formula that can be used to calculate π . We will calculate the sum up to $k = 10^6$. The `%timeit` magic function of IPython was used.

```
# Using python
%timeit sum([( -1)**k/float(2*k+1) for k in range(10**6)])
# result : 493ms

%timeit sum([(1-2*(k%2))/float(2*k+1) for k in range(10**6)])
# result : 253 ms

def calc_pi(N):
    out = 0
    sgn = 1.
    for k in range(N):
        out += sgn/(2*k+1)
        sgn = -sgn
    return out

%timeit calc_pi(10**6)
# result 112ms

import numpy as np
k = np.arange(10**6, dtype=np.float64)
%timeit np.sum((1-(k%2))/(2*k+1))
# result : 17ms
```

Below is a simple example using cython. The online documentation (<http://docs.cython.org/>) will give you more informations on how to use cython and to compile the code below. We have created a `calc_pi.pyx` file from the pure python function. The `pyx` file contains regular Python code with declaration of the type of variable.

```
def calc_pi(int N):
    cdef double out = 0
    cdef int i
    cdef double sgn = 1
    for i in xrange(N):
        out += sgn/(2*i+1)
        sgn = -sgn
    return out
```

To use the code below, run the following script

```
import pyximport; pyximport.install()
import calc_pi

print(calc_pi(1E6))
```

Once compiled, the `calc_pi` function can be used as any python function. The duration is 4.9ms, four times faster than numpy. Even in the situation where numpy can be used, cython can be faster, the main reason is that in this example we don't need to store an array in memory.

ctypes

Ctypes is a library that allow to use C functions in Python. This library allow you to link any functions. You don't need the source code of the function but only a compiled shared library (.so for linux and .dll for windows). For this example, I will show you how to compile your shared library from a C file, but ctypes can be used even if you get a library from a third party (for example the driver of an instrument).

The C file to calculate π is the following (named `pi.c`):

```
#include <stdio.h>
#include <stdlib.h>

int calc_pi(int N, double * out){
    int i;
    double sgn = 1;
    *out = 0;
    for(i=0; i<N; i++){
        *out += sgn/(2*i+1);
        sgn = -sgn;
    }
}
```

Which is compiled (under linux) using :

```
gcc -shared -o libpi.so -fPIC pi.c -Wno-pointer-to-int-cast
```

The `libpi.so` is now linked using ctypes :

```
import ctypes
from ctypes import byref
lib = ctypes.cdll.LoadLibrary('./libpi.so')
```

(continues on next page)

(continued from previous page)

```
calc_pi = lib.calc_pi
out = ctypes.c_double(0) # creation of the output variable
calc_pi(10**6, byref(out))
print(out.value)
```

In this example, the memory for the output variable is created in the python script and then passed by reference to the C function. The execution time is similar to the Cython example.

Numba

Numba is a compiler for python. It is even simpler to use than cython, because no modification of the code is required. A simple decorator is used to accelerate a Python function.

```
import numba

@numba.jit(numba.float64(numba.int32), nogil=True)
def calc_pi_numba(N):
    out = 0
    sgn = 1.
    for k in range(N):
        out += sgn/(2*k+1)
        sgn = -sgn
    return out
```

2.2 Graphics in Python

There is a very complete library in Python that can be used to plot curves. This the matplotlib library. It relies on the numpy library because curves are always plotted using arrays.

The plot command

Let us start with an example :

```
from pylab import *
ion()
```

(continues on next page)

```
x = linspace(0,2*pi)
y = sin(x) + cos(y)**2
plot(x, y)
```

The `pylab` modules will import all the function that will make Python suitable for numerical and graphical computation. It will import the `matplotlib` and `numpy` modules. To use `pylab`, one can either import it using the `from pylab import *` command or using an interactive python shell with the command `ipython -pylab`.

The `plot` command takes two parameters : the X and Y axis coordinates (if only one parameter is given, the x axis will be equal to `arange(len(Y))`)

Optional parameters are used to modify the way the graph is plotted. The two main parameters are the color and the shape. For example, the command `plot(x, y, 'k^')`, will plot triangle (signe '^') in black ('k' is for black - the 'b' is for blue). Look to the documentation for more details.

Other commands

To clear a graph, we use the function `clf()` (clear figure)

On can add a title (`title("Le titre")`), labels for axis (`xlabel('x-axis')` et `ylabel`). The optional `label` parameter is used to make a legend on a graph with many plots

```
X = linspace(-2,2, 100)
Y = sin(X)**2*exp(-X**2)
Y_noise = Y + .1*(rand(len(X))-0.5)
plot(X,Y, label=u"Theory")
plot(X,Y_noise,'o', label=u"Experiment")
xlabel(u'Voltage [V]')
ylabel(u'Length [m]')
title("Nonsense graph")
legend()
grid(True)
```

Latex formula

In recent version of `Matplotlib`, it is possible to automatically insert Latex formula in graphs. They will be nicely converted

```
ylabel(u'Noise [$V/\sqrt{\mathrm{Hz}}$]')
```

Main graphical commands

- `plot(X, Y)`
- `loglog(X, Y)`, `semilogx(X, Y)`, `semilogy(X, Y)`
- `clf()` to clear the graph
- `xlabel('blabla')`, `ylabel('blabla')`, `title('blabla')`
- `xlim((x_inf, x_sup))`, `ylim((y_inf, y_sup))` to zoom onto a part of the graph
- `grid(True)` to plot the grid. The command `grid(True, which='both')` will plot a thin grid.
- `figure` to open a new figure (a new window). Figures are automatically numbered. You can switch to an existing figure using the `figure(n)` command.
- `subplot(nx, ny, m)` to make many plots on one figure.
- `imshow(image)` to plot a matrix using false color and `colorbar()` pour plot color scale.
- `text(x, y, s)` plot the text `s` at the `x`, `y` position. Optional parameters can be used to center correctly the text.
- `savefig(nom_fichier)`. Save the figure. The file format is determined by the file extension. We advise to use `pdf` for output that cannot be modified and `svg` if one want to modify it (using [Inkscape](#)- for example).

[Screenshots](#) are available on the `matplotlib` web site and can be used to quickly find the best way to plot a specific graph.

2.3 Data fitting

The `scipy.optimize` module contains function that can be used to perform data fitting.

The easiest function to use is the `curve_fit` function. Below is the documentation of the function.

curve_fit

Use non-linear least squares to fit a function, f , to data.

Assumes $ydata = f(xdata, *params) + \epsilon$

Parameters

f [callable] The model function, $f(x, \dots)$. It must take the independent variable as the first argument and the parameters to fit as separate remaining arguments.

xdata [An N-length sequence or an (k,N)-shaped array] for functions with k predictors. The independent variable where the data is measured.

ydata [N-length sequence] The dependent data — nominally $f(xdata, \dots)$

p0 [None, scalar, or M-length sequence] Initial guess for the parameters. If None, then the initial values will all be 1 (if the number of parameters for the function can be determined using introspection, otherwise a `ValueError` is raised).

sigma [None or N-length sequence] If not None, it represents the standard-deviation of ydata. This vector, if given, will be used as weights in the least-squares problem.

Returns

popt [array] Optimal values for the parameters so that the sum of the squared error of $f(xdata, *popt) - ydata$ is minimized

pcov [2d array] The estimated covariance of popt. The diagonals provide the variance of the parameter estimate.

Notes

The algorithm uses the Levenburg-Marquardt algorithm: `scipy.optimize.leastsq`. Additional keyword arguments are passed directly to that algorithm.

Examples

```
>>> import numpy as np
>>> from scipy.optimize import curve_fit
>>> def func(x, a, b, c):
...     return a*np.exp(-b*x) + c
```

```
>>> x = np.linspace(0,4,50)
>>> y = func(x, 2.5, 1.3, 0.5)
>>> yn = y + 0.2*np.random.normal(size=len(x))
```

```
>>> popt, pcov = curve_fit(func, x, yn)
```

Fitting image

The `curve_fit` function described above is used to minimize the following :

$$\sum_i (f(x_i, p_0, p_1, \dots) - y_i)^2$$

Indeed, it assumes that `f(x[i], ...)` = `f(x, ...)[i]` and uses the second method. Therefore, as long as the function `func` returns a 1D array, the `curve_fit` will return the optimal parameters that minimize the sum. The parameter `x` does not have to be a 1D array.

In order to fit pictures (2D array), one has to flatten the image array and creates the corresponding `x` and `y` coordinate. This can be done using the following code :

```
# image is a 2D array
ny, nx = image.shape
X,Y = meshgrid(range(nx), range(ny))
# two column matrices with X and Y
XY = array([X.flatten(), Y.flatten()]).transpose()
```

The `XY` array can be passed as the first parameter of the fit function. The fit function should then extract the corresponding `x` and `y` coordinates. For example :

```
def gauss(XY, amplitude, center_x, center_y, diameter):
    x = XY[:,0]
    y = XY[:,1]
    return amplitude*exp(-(x-center_x)**2 + (y-center_y))/
    ↪diameter**2)

popt, pcov = curve_fit(gauss, XY, image.flatten(), p0)
```

Note also that, because we have flattened the 2D array, you can select the points for your fit without keeping a matrix like structure. The following example will perform the fit only for points in the image with a positive intensity.

```
fimage = image.flatten()
condition = fimage>0
popt, pcov = curve_fit(gauss, XY[condition, :],
    ↪fimage[condition], p0)
```

2.4 Differential equation

The `scipy` library contains a large number of scientific modules. This is almost a standard library in the sense that it contains all standard numerical algorithms. The full documentation can be found at <http://docs.scipy.org/doc/scipy/reference/>.

The module that contains the functions for solving differential equation is the `scipy.integrate` module with the `odeint()` et `ode()` functions. We will present only the `ode()` function which is more complete. Those functions can solve ordinary differential equation :

$$\frac{dy}{dt} = f(t, y)$$

where `y` is a numpy array.

Let us look to the following example $y' = -y$ with $y(0) = 1$:

```
from scipy.integrate import ode

def f(t, y):
    return -y

r = ode(f).set_integrator('vode')
r.set_initial_value(y=1, t=0)

print r.integrate(1)
print r.t, r.y
```

Few remarks :

- Many solvers can be used (see the online documentation). Each solver is indeed a Fortran library that was adapted for `scipy`. Each solver has its own parameters. The most common parameters are the absolute (atol) and relative precision (rtol). Compared to well known imperative solvers (like the Euler or Runge-Kutta methods), the `scipy` solvers are able to adapt the size of the steps in order to obtain the required precision.

- For highest order differential equation, the trick consists in writing a first order equation by adding intermediate functions which are the derivative of the initial functions.

For example : solution of the linearized pendulum equation: $y'' = -y$, with initial conditions $y(0) = 1$ and $y'(0) = 0$.

```
from scipy.integrate import ode
from numpy import *

def f(t, Y):
    y, yprime=Y
    return array([yprime, -y])

r = ode(f).set_integrator('vode')
r.set_initial_value(array([1,0]),0)

print r.integrate(pi)
```

2.5 Fourier analysis

Formulas

Usefull formulae for the Fourier transform

- Continuous Fourier transform

$$\tilde{f}(\omega) = \int_{-\infty}^{\infty} f(t)e^{-i\omega t} dt$$

- Inverse Fourier transform

$$f(t) = \frac{1}{2\pi} \int_{-\infty}^{\infty} \tilde{f}(\omega)e^{+i\omega t} d\omega$$

- Discrete Fourier transform

$$\tilde{x}_k = \sum_{j=0}^N x_j e^{2i\pi \frac{kj}{N}}$$

This is the formula used by computers.

- Link between discrete Fourier transform and continuous Fourier transform :

$$\tilde{x}_k = \sum x_j e^{2i\pi(j\Delta t) \frac{k}{N\Delta t}} = \frac{1}{\Delta t} \sum f(t_j) e^{2i\pi t_j \frac{k}{T}} \Delta t = \frac{1}{\Delta t} \tilde{f}\left(\frac{k}{T}\right)$$

With this formula you can get the correct scale for the x and y axis.

FFT

The algorithm used to perform the discrete fourier transform is called the fast Fourier transform (FFT). This algorithm is efficient when the array can be recursively splitted in a small number of arrays of the same size. It is therefore very efficient when the size of the array is a power of two. In this case the complexity will be in $O(n \log n)$ where n is the size of the array. This algorithm is inefficient if the size of the array has large integer in his factorization. For example :

```
from numpy.fft import *
from numpy import rand

a = rand(2**17)
%timeit fft(a) # about 4.5 ms

a = rand(2**17-1) # 2**17-1 is a prime number
%timeit fft(a) # about 52 s
```

3 Object oriented programming

3.1 Object oriented programming

In python all variables (including functions) are objects. We have already seen some concepts of object oriented programming : attributes and methods. Let us look to complex numbers

```
z = 1 + 2J
print z.real # this is an attribute
zconj = z.conjugate() # this is a method
```

The type of the object referenced by the variable `z` is a complex number. It means that somewhere a `class` defined what a complex number is.

First example

Building a class

Let us look how to create our own `Complex` class.


```
class Complex(object):
    def __init__(self, x, y):
        self.real = x
        self.imag = y

z = Complex(1,2.)
```

In this example, we have created the class. The init method is called when the object is created and it sets the attributes `real` and `imag` of the complex number.

One can write the conjugate method and add other methods:

```
class Complex(object):
    ...
    def conjugate(self):
        return Complex(self.real, -self.imag)
    def modulus(self):
        return math.sqrt(self.real**2 + self.imag**2)

z = Complex(1,2.)
conjz = z.conjugate()
```

Special methods `__str__` and `__repr__`

The `__str__` and `__repr__` methods are used to create a string for simplified printing (`__str__`) or full representation (`__repr__`).

```
def __str__(self):
    if self.imag > 0:
        return "{0} + {1}j".format(self.real, self.imag)
    else:
        return "{0} - {1}j".format(self.real, -self.imag)
def __repr__(self):
    return "Complex({0}, {1})".format(repr(self.real),
    ↪repr(self.imag))
```

Special methods for binary operation

The `__add__`, `__mul__`, `__sub__` and `__div__` (and `__truediv__`) methods are used to implement addition, multiplication, subtraction and division. Let us implement the `__add__` method

```
def __add__(self, other):  
    return Complex(self.real+other.real, self.imag+other.imag)
```

This method will let us add two complex numbers using the '+' sign.

```
a = Complex(1,2)  
b = Complex(1,-1)  
  
print a+b
```

Instrument Driver

Driving instruments using objects

Example of a laser driver

```
class Verdi(object):  
    """ Verdi driver class  
  
    example :  
        laser = Verdi('COM7')  
        laser.set_power(15)  
    """  
    def __init__(self, port):  
        """ ... """  
        self.serial_connection = serial.Serial(port=port,   
↳ baudrate=19200)  
    def read_cmd(self, cmd):  
        self.serial_connection.write('{0}\r\n'.format(cmd))  
        return self.serial_connection.readline()  
    def write_cmd(self, cmd, val):  
        self.serial_connection.write('{0}:{1}\r\n'.format(cmd,   
↳ val))  
    def get_power(self):  
        return self.read_cmd('P')  
    def set_power(self, val):  
        self.write_cmd('P', '{0:6.4f}'.format(val))  
  
laser = Verdi('COM7')  
laser.set_power(15)
```

When are objects useful ?

- Classes provide high level information. In the previous example, all the information concerning the connection are inside the object.
- Implementation details are hidden (the user do not have to know how to connect to a serial port).
- The object version allows the code to pass many fewer variable (to set the power, we only give the power and not the port, baudrate or the brand of the instrument).
- The object version is better organized (all the function are grouped inside the class)
- The object version is more versatile : the user interface does not depend on the specific instrument. For example if you write a Millenia class, then you can use your programm for a Millenia by changing only one line (laser = Verdi(...)) is replaced by laser = Millenia(...)).

Tutorial on classes

The web page <https://docs.python.org/2/tutorial/classes.html> contains an interesting tutorial on classes.

Customizing object

We have already seen some special methods used to customize object. The data model of Python (<https://docs.python.org/2/reference/datamodel.html>) describes all the special methods. They are used to customize the way object are displayed or compared with another. They are also used to mimic different type like the numeric type, containers type (list, dictionary, ...), callable (function).

Two builtins functions are usefull when dealing with object : the function `isinstance(obj, cls)` tests if `obj` is an instance of `cls`. The `hasattr(obj, attr_name)` function test if `obj` has an attribute called `attr_name`.

When creating a numeric like object, consider also the implementation of reversed operator. When python see `a + b` then it call the method `__add__` on object `a`. In the example of Complex number that we have seen, we have implemented the addition of two Complex number. We can also modify the method `__add__` to be able to add a number to a Complex number (`a` is Complex but `b` is a number). We cannot modify the method `__add__` of float (for example) so that we can add

a Complex number to a float (a is a float). The way to do it is to implement the `__radd__` (reversed add) method. Below is the way to do implement the addition for Complex :

```
import numbers

class Complex(object):
    def __add__(self, other):
        if isinstance(other, numbers.Number):
            other = Complex(other, 0)
        if isinstance(other, Complex):
            return Complex(self.real+other.real,
                           self.imag+other.imag)
        raise NotImplemented
    def __radd__(self, other):
        return self + other # addition is commutative
```

If the operation is not implemented, you should raise a `NotImplemented` error. This error will be caught by the interpreter to try the reverse operation.

Properties

Classes implements different kinds of attributes : data attributes are used to store variables inside the object. Methods are used to call function that depends on the object and other arguments. Methods are use to perform an actions : modifying the object, performing calculations, ...

```
class MyClass(object):
    a = 1
    b = []
    def __init__(self):
        c = 2
        d = []
    def f(self):
        return 3

x = MyClass()
x.e = []
y = MyClass()
y.a = 2
y.b.append(4)
y.d.append(3)
print x.a, y.a
```

(continues on next page)

```
print x.b, y.b
print x.d, y.d
```

In this example, c, d, e are data attribute (they are linked to the object). But, a and b are class attribute. When one call `x.a`, because a is not a data attribute, then the interpreter look for the class, find a valide class attribute and return it. In the above example, `x.b` and `y.b` will be the same object, but not `x.d` and `y.d`. Assignment will always create a data attribute.

Basically, functions defined in the class are converted to methods. In the above example, `MyClass.f` is the original function and `x.f` is the method, i.e. a new function with `x` as first paramter.

Properties is another kind of attribute. When a method (without arguments) is used to get information from the object, the value return from this method is similar to an attribute. For example, complex number were described using real and imaginary part. To get the modulus of the complex number, we have used a method. But if the complex number were represented using a polar representation, then the norm would be an attribute. Using property allows to hide this implementation detail.

```
class Complex(object):
    def __init__(self, x=None, y=0, r=None, theta=None):
        if x is not None:
            self.real = x
            self.imag = y
        else:
            if r is None or theta is None:
                raise ValueError('Complex number should be
↳defined\                                     using either x and y or r and theta
↳')
            self.real = r*math.cos(theta)
            self.image = r*math.sin(theta)

    @property
    def r(self):
        return self.modulus()

    @property
    def theta(self):
        return math.atan2(self.imag, self.real)

    def __mul__(self, other):
        r = self.r*other.r
```

(continued from previous page)

```
theta = self.theta + other.theta
return Complex(r=r, theta=theta)
```

Properties are also usefull when you want to perform an action when an attribute is set. For example, using the Verdi class defined above, we can create the power property with a setter and a getter :

```
class Verdi(object):
    ...

    def get_power(self):
        ...
    def set_power(self, val):
        ...

    power = property(get_power, set_power, doc="blabla")
```

Now we can use this object like this :

```
laser = Verdi('COM7')

laser.power = 15 # Set the laser power to 15 W
print laser.power # Return the actual power of the laser
```

Inheritance

One of the main feature of object oriented programming is the possibility for classes to heritate from other classes. When a class B heritates from class A, then all attributes of class A are available for class B. Actually, we have already used heritage, as the every object heritates from the class object. For example, by default, every object have a method called `__repr__`.

In a typical situation, a class heritated from a single class. This situation occurs when one wants to specialize a class. For example, one can create a Imaginary number class. Such a number, will be a complex number, but the `__init__` and `__str__` method can be changed :

```
class Imaginary(Complex):
    def __init__(self, y=1):
        return Complex.__init__(self, x=0, y=y)
    def __str__(self):
        return "{0}j".format(self.imag)
```

3.2 Pauli matrices

The wikipedia http://en.wikipedia.org/wiki/Pauli_matrices page will remember you what the Pauli matrices are.

In quantum mechanics, Hamiltonian of a two level system can be described using a linear combination of Pauli matrices (vector part) and the Id matrice (scalar part).

1. Create a class that can be used to describe such a linear combination
2. Implements addition, subtraction and multiplication
3. Implements the exponential (we can create an `__exp__` method of the object and modify the function accordingly).

The answer of the first question is :

```
class PauliMatrices(object):
    def __init__(self, scalar, vector=(0,0,0)):
        self.scalar = scalar
        self.vector = np.array(vector)

    def __str__(self):
        out = ''
        if abs(self.scalar)!=0:
            out+='{0:~}Id '.format(self.scalar)
        for i,elm in enumerate(['x','y','z']):
            if self.vector[i]!=0:
                if isinstance(self.vector[i], numbers.
↪Complex)\
                                and out!='':
                    out += ' + '
                out+='{0:~}*σ{elm} '.format(self.vector[i],
↪elm=elm)
        return out
```

The `__str__` method could be improved.

3.3 Data analysis

The idea is to create a class used for data analysis. In this example we will use simple linea regression : the experimental data x and y will be fitted by the function $y = ax + b$.

In order to calculate the value of a and b we use the following :

```
det = ((x - x.mean())**2).mean()
a = ((x*y).mean() - x.mean()*y.mean()) / det
b = y.mean() - a*x.mean()
```

1. This class should be able to give the value of a and b but also to plot the data with the fit function. Furthermore, you should be possible to subclass the class to customize the default plot parameters such as the title, markers or label.
2. Rewrite your code in order to separate what is generic to any linear regression and to first order regression. Now write three classes : Regression, FirstOrderRegression and SecondOrderRegression
3. A linear regression is a fit by a function like :

$$f(x) = p_0 f_0(x) + p_1 f_1(x) + \dots + p_n f_n(x)$$

One can find on wikipedia the formula to obtain the p_i from x and y .

Write a GenericRegression class such that one can customize the function using, for example, the following code :

```
class MyRegression(GenericRegression):
    reression_order = 2
    @staticmethod
    def fit_function(x, p):
        return p[0] + p[1]*x + p[2]*exp(-x)
```

The `@staticmethod` decorator is used to specify that the `fit_function` is a data attribute and not a method (its first parameter should not be replaced by the instance).