Introduction to Python

Francesco Marchetti

Lab of Fundamentals of Computational Mathematics

Summary

Welcome to the lab!

This lab is the "pratical" part of the course. After recalling some programming tools, we will experiment concretely with the topics of the course.
Some advice:

- Take advantage of these hours!
- Always try to understand the code that you are looking at.
- You will make mistakes, but the error messages will tell you what's wrong. Learn how to understand them (it is a matter of practicing).

What is Python

Python is a general-purpose programming language, open-source, which has been released in the 90s.

- It is relatively easy to be used.
- It is really widely-used, you can find many tutorial and helping documents in the web.

In this course, we will mainly work with arrays to implement our numerical methods. Keep in mind that, after all, we are just using a really powerful calculator.

Python with Spyder

In this lab, we use Spyder to write and launch Python codes. A stand-alone version is available at
–> `https://www.spyder-ide.org/`

This will be enough for our purposes, but a more complete and customable version of Spyder can be obtained by including it in the Anaconda environment.

Moreover, there are of course many other programs to work in Python, you are free to choose others.

Spyder's interface

When we launch Spyder, we see the following:

- Console: where we can insert and launch commands, starting from easy operations.
- Editor: where we can write more elaborated code, save and run it.
- Variable explorer: where the defined variables are displayed.
- Files: the file explorer.

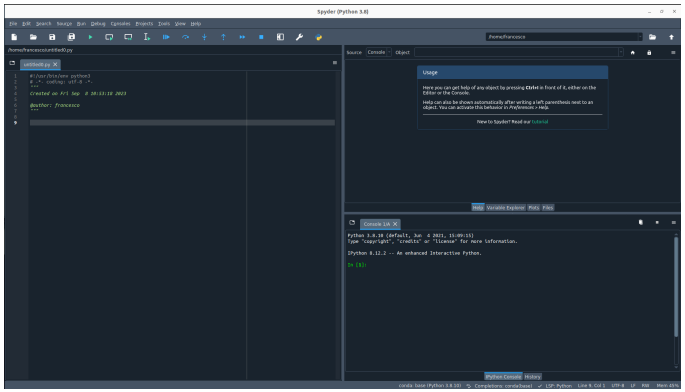

Figure: Spyder's interface

Files and working directory

You can create new folders to organize your scripts. Note that the directory that you see in the file explorer may not correspond to your current working directory, that is the folder that Spyder sees in that moment. To know the current working directory, you can launch the following commands: `import os` and then `os.getcwd()`. When we open a saved Python script from the explorer, we automatically move to its working directory.

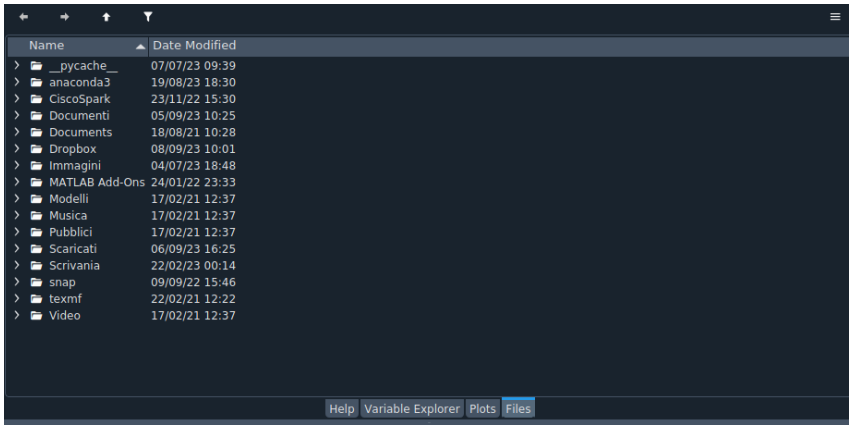

Figure: File explorer.

The numpy package

In Python, besides the most basic operations, many functions are implemented in a huge variety of different packages. Therefore, if we need a certain function, we can get it by importing the package it belongs to.

The numpy package contains many useful functions and we will use it continuously. We can import it in our script by writing at the beginning `import numpy as np`, where with `as np` we indicate that any function `fun` in numpy will be called as `np.fun`, without writing the full name. This nomenclature is widely used, so don't use others!

Some special numbers

Using numpy, we have some special numbers.

±np.inf	± *infinity*	np.pi	*Pi*
np.nan	*indefinite number*	np.finfo(float).eps	*machine precision*

Table: Some examples.

Some other useful functions

`np.sign`	*sign*	`np.abs`	*absolute value*
`np.sqrt`	*square root*	`np.sin`	*sine*
`np.cos`	*cosine*	`np.tan`	*tangent*
`np.log`	*natural logarithm*	`np.exp`	*exponential*
`np.log10`	*base-10 logarithm*	`np.round`	*rounding*
`np.floor`	*integer part (truncation)*	`np.ceil`	*integer part + 1*

Table: Some functions.

Don't learn them by heart! You can easily recover them if necessary.

Vectors from lists

In this course, we will widely use arrays and related operations to work with vectors and matrices.
Given a list `my_list` of n elements (numbers), we can generate a $1 \times n$ vector `my_vec` by writing

```
my_vec = np.array([[my_list]])
```

If we need a column-vector, we can transpose `my_vec` by writing

```
my_vec = my_vec.T
```

Equispaced vectors

We can define equispaced vectors of many components by two different options:

- The function `np.linspace(a,b,n)` defines an array of n components equally spacing from a to b. Note that we may use `np.linspace(a,b,n).reshape(1,-1)` to get a $1 \times n$ vector or `np.linspace(a,b,n).reshape(-1,1)` to get a $n \times 1$ vector.
- The function `np.arange(a,b,s)` defines an array equally spacing from a to b, with a step s. Again, we can use `reshape` to obtain vectors. Moreover, note that `np.arange(a,b,s)` does not include the endpoint! To include it, we can write `np.arange(a,b+s,s)`.

They are equivalent... but comfortable for different purposes!

Vettori concatenati

Vectors `v1,v2,...,vm` can be concatenated vertically or horizontally by using

`np.vstack((v1,v2,...,vm))` or `np.hstack((v1,v2,...,vm))`

Note that the result of such concatenation must be a valid array, that is a valid matrix or vector.

Matrices from lists

Similarly to vectors, matrices can be obtained manually from lists. Given m lists `l1`,`l2`,...,`lm` of n elements (numbers), we can generate a $m \times n$ matrix `my_mat` by writing

```
my_mat = np.array([[l1],[l2],...,[lm]])
```

Again,we can transpose `my_mat` by writing

```
my_mat = my_mat.T
```

Some useful matrices, size of arrays

Some useful matrices are generated as follows.

- `np.zeros((m,n))` generates a $m \times n$ matrix of zeros.
- `np.zeros((m,n))` generates a $m \times n$ matrix of ones.
- `np.eye(n)` generates the $n \times n$ identity matrix.

Note that we can get the size of any array (vector or matrix) `arr` by calling `arr.shape`. The result is of type `tuple`.

Lambda functions

Vary often it is useful to define functions via the `lambda` command. For example, suppose that we wish to define the function $f(x) = \sin(x) - \cos(x) - 3$, we can write

```
f = lambda x:  np.sin(x)-np.cos(x)-3
```

If a vector or matrix is passed as variable, the function works in a pointwise manner.

Operations on arrays

Let `v` and `w` be vectors (or matrices) of the same dimensions.

- `v+w` gives the sum of the two vectors;
- `v-w` gives the difference of the two vectors;
- `v*w` gives the pointwise product of the two vectors;
- `v/w` gives the pointwise ratio between the two vectors;
- `v**w` gives the pointwise raise to powers contained in `w`.

We remark that if c is a scalar then doing `v+c`, `v-c`, ... etc still result in a pointwise operation. If the dimensions are proper, `v@w` computes the matrix product between the vectors. The scalar product can be also computed in this way (how?).

The matplotlib package

To our purposes, it will be convenient to represent functions and results by means of plots.
We will make use of the `matplotlib` package, which provides all the necessary ingredients, including it as

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

Particularly, we will employ the function

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

which plots couples of corresponding points in column-vectors `x` and `y` in the usual Cartesian plane.
There are some related options and commands, but we will see them in time when applying them (it's easier).

Semilogarithmic plot

We focus on a particular type of plot that is important for our aims, we present it by means of the following exercise

Exercise

Let $f(x) = \sin(x)$ on $I = [0, 2\pi]$. Let us consider the related Taylor polynomial of degree 3 $p(x) = x - x^3/6$. It is well known that p approximates f in a neighborhood of. We want to describe the absolute error $|f(x) - p(x)|$ for $x \in I$. Let us plot in I:

1. The functions f e p in the same plot (consider a vector of equispaced points in I to evaluate the functions at).
2. The function $err(x) = |f(x) - p(x)|$ in a different plot.
3. Again $err(x)$... but using `plt.semilogy`!

Use `plt.figure()` before the plot to create a new window for it.

Comments

- By displaying f and p in the same plot, we obtain some qualitative information: we observe that they are similar near zero, and they behave differently elsewhere. This kind of information is useful to have some idea on what's going on (e.g. if a function is positive or negative, zeros etc).
- By showing the error with `plt.plot`, we obtain something more quantitative. However, actually it is not clear what happens near zero (what is the error? 1e-3 or 1e-9?).
- With `plt.semilogy`, we have far more precise information on the error.

Python functions and other structures

We recall that we can define functions in the form

```
def function_name(input1,...,inputn)

...

return output1,...,outputm
```

We recall that variables defined in the function are restricted inside that environment.

Of course, we will make use of **if-else** constructions, **for** and **while** cycles.

Logic operators

Python handles logic variables (type bool), by assigning 1 to True and 0 to False.

==	*equal*	!=	*not equal*
>	*greater*	<	*smaller*
>=	*greater equal*	<=	*smaller equal*
and	*and*	or	*or*
not	*not*		

Table: Some logical operations.

End of this introduction

We conclude this introduction with the following question: in Python, is it true or false that

```
3*0.2 == 0.6?
```