Python introduction

H.G. Vivien[1]

[1] *Laboratoire de Physique et Chimie de l'Environnement et de l'Espace (LPC2E), UMR CNRS 7328 - Université d'Orléans, Orléans, France*

July 6, 2021

Introduction

The following document contains an introduction to the Python programming language, including some lesson about the basics, examples, and exercises. While it can form a useful basis I don't recommend solely relying on it to learn the language; using books and tutorial in parallel is the way to go to learn any language. I recommend particularly the website https://www.learnpython.org/, which proposes many interactive tutorials and example.

The following exercises will have a correction uploaded to https://github.com/At0micBee/Exercises, so that you can check your results and methods to mine. I will try to keep the answer as understandable to beginners as possible, rather than writing to be as efficient as can be.

This document is more centered towards scientific processes and data analysis, and thus might not be suited for other applications. The modules covered will mostly be used for science application, and the exercises require a solid basis in both math and physics.

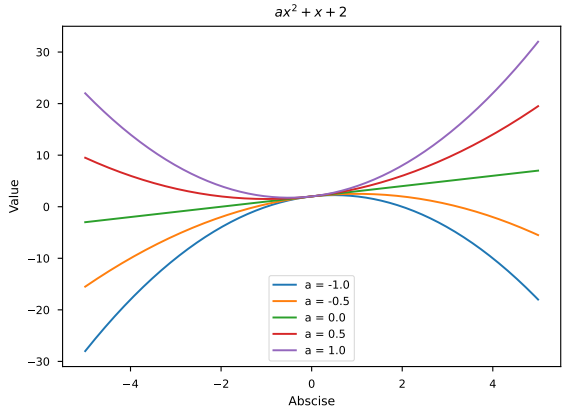

Figure 1: Example of solution to Ex. 1.

1 Polynomial coefficients

The definition of a second order polynomial is:

$$f(x) = ax^2 + bx + c \tag{1}$$

with a, b, c real numbers. We're going to look at the impact of changing these coefficients with x in the range $[-10,\ 10]$. Write a program that will modify each the parameters a, b or c independently. Each time we want to produce a figure that shows the resulting polynomial curve for each of the value taken by the variable parameter. Respectively, we want:

$$\begin{cases} a \in [-1,\ 1],\ b = 1,\ c = 2 \\ a = 1,\ b \in [-1,\ 1],\ c = 2 \\ a = 1,\ b = 2,\ c \in [-1,\ 1] \end{cases} \tag{2}$$

For example, the first case described here would produce Figure. 1

2 Polynomial roots

Create a program that computes the roots of a polynomial given its parameters a, b, c, knowing that:

$$\begin{cases} \Delta = b^2 - 4ac \\ r_\pm = \frac{-b \pm \sqrt{\Delta}}{2a} \end{cases} \tag{3}$$

1. Compute Δ, and figure out if the solution is real or complex
2. If it is real, solve it, otherwise print a message saying we won't compute it
3. As a bonus, add another if case to compute when $\Delta = 0$, and where $r = \frac{-b}{2a}$

3 Factorial function

Implement a function that takes in an `integer` and returns the result of the factorial of that number, knowing that the factorial function is:

$$n! = \prod_{i=1}^{n} i \tag{4}$$

Have it display the result for a few values. Once that is done, we want to look at the trend of this function. To achieve this, we want to compute the factorial of all integer up to 10, and plot the result.

4 Computing an average

We are given the following set of values:

```
1 v = [2.5, 3.2, 4.1, 0.8, 1.2, 2.4]
```

Copy this line to a program, and compute both the sum and the average value of this list, with the general form of the sum and average being:

$$s = \sum_{i=1}^{n} f(i) \tag{5}$$

$$a = \frac{1}{n} \sum_{i=1}^{n} f(i) \tag{6}$$

Where n is the number of values to take into account, i in the index of the values, and $f(i)$ is the value at index i.

5 Vector manipulation

Create two vectors of three elements $\vec{v_1}$ and $\vec{v_2}$ as follows:

$$\vec{v_1} = \begin{pmatrix} 0.2 \\ 1.3 \\ 7.4 \end{pmatrix},\ \vec{v_2} = \begin{pmatrix} -7.1 \\ 2.4 \\ -0.7 \end{pmatrix} \tag{7}$$

Compute the following:

1. The addition $\overrightarrow{v_{\text{add}}} = \vec{v_1} + \vec{v_2}$
2. Multiply them by a scalar: $2\vec{v_1}$ and $-3\vec{v_2}$
3. Combine them: $\overrightarrow{v_{\text{tot}}} = 2\vec{v_1} - 3\vec{v_2}$
4. Compute the scalar product: $S = \vec{v_1} \cdot \vec{v_2}$
5. Compute the rotational: $\overrightarrow{v_{\text{rot}}} = \vec{v_1} \wedge \vec{v_2}$

Reminder about the $\overrightarrow{curl}$ product between two vectors:

$$\vec{v_1} \wedge \vec{v_2} = \begin{pmatrix} v_{1y}v_{2z} - v_{2y}v_{1z} \\ v_{1z}v_{2x} - v_{2z}v_{1x} \\ v_{1x}v_{2y} - v_{2x}v_{1y} \end{pmatrix} \tag{8}$$

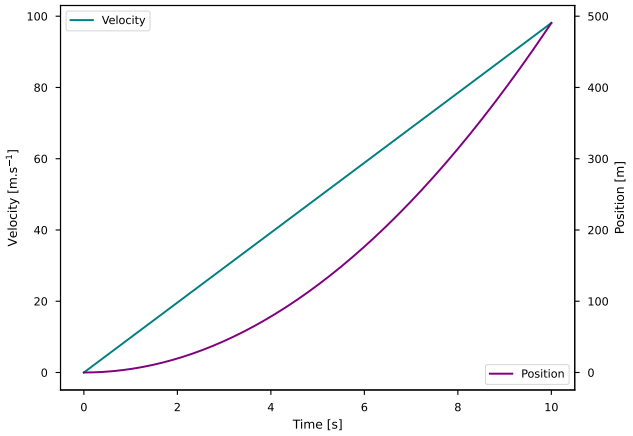

Figure 2: Plot showing the solution of Ex. 6. The velocity is read on the left and position on the right.

6 Gravity

Consider the force of gravity on a 1 dimensional axis, with $g = 9.81\text{m.s}^{-1}$. We want to compute the velocity and position of an object under this gravity field for 10 seconds. Assume the original position is the origin, and that it has no initial velocity.

Once that is done, we want to visualize the movement and speed on a plot to give us an idea of what is happening. Create a plot to represent both the velocity and position in function of time (see Figure. 2 for an example).

7 Surface areas

Create a function that returns the surface of a circle for a given radius r, and one that returns the surface of a square of side a. Compute the respective results for $a = r$ for a few values between 0 and 2.

Plot the results to visualize the difference of growth between the two.

8 Gravity, medium difficulty

Compute the force between the sun and an Earth mass planet for distances between 0.3 and 5AU. We have $M_\odot = 1.989 \cdot 10^{30}\text{kg}$, $M_\oplus = 5.972 \cdot 10^{24}\text{kg}$, $1\text{AU} = 150 \cdot 10^{9}\text{m}$, and with:

$$F_{\text{grav}} = G\frac{M_1 M_2}{r^2} \tag{9}$$

Where $G = 6.674 \cdot 10^{-11}\text{m}^3.\text{kg}^{-1}.\text{s}^{-2}$ is the gravitational constant, M_1 and M_2 are the masses of the objects and r is the distance separating them.

9 Studying infinite series

We are first going to look at a simple series, and then work our way towards something more complicated.

$$s = \sum_{n=1}^{\infty} \frac{1}{n} \tag{10}$$

We would like to do two things:

1. Estimate if it converges, and if it does to what value
2. Visualize the impact of each term as i grows

Obviously, we cannot cannot compute an infinity of terms, or it would literally take forever, but we can take advantage of the speed of the machine to compute a lot of them. We can start by only computing the first dozen terms, then we can push it to many more to study the series.

10 Gravity, but complicated this time

We're going to do a similar exercise to the Ex. 6, but more complicated this time. We're going to compute g at various altitude first, then use that value to compute the motion of a falling object. The general formula to compute gravity is:

$$g(r) = -\frac{GM(r)}{r^2} \tag{11}$$

Where r is the radius at which to estimate the gravity, $G = 6.674 \cdot 10^{-11}\text{m}^3.\text{kg}^{-1}.\text{s}^{-2}$ is the gravitational constant and M is the mass under the given radius. Considering that the Earth is a perfect sphere of 6371km, that the mass under that radius is of $5.972 \cdot 10^{24}\text{kg}$ and that the mass of the atmosphere is negligible, compute the gravity for the altitude z between 0 and 100km. **Be careful, the altitude starts at the radius value of** 6371km**!**

We can see that even for a change of only 100km, there is a non-negligible change in gravity. Assuming an object starts to fall from an altitude of 100km with no initial velocity, and assuming there is no drag forces in the atmosphere, compute the following:

1. The velocity at each point in time during the fall
2. The position though time
3. The time it will take to reach the surface

And end the simulation when it hits the ground. To compute each point in time, we're going to use a simple explicit Euler scheme, written simply as:

$$x_{i+1} = x_i + \frac{\partial x}{\partial t}\Delta t \tag{12}$$

Where x is the quantity we are trying to compute, i are the increment of the disretization, and $\frac{\partial x}{\partial t}\Delta t$ is the compute variation of parameter x during Δt.

A Terminology

In this section, we go over some of the terminology and notation related to Python and programming in general.

- *Data type*: Also sometimes referred to simply as *type*, it is the nature of some data in the code. For example 6 would be an `integer`, and `[4.2, 6.5, 2.8]` is a `list` of `float`.
- *Keyword*: In a programming language, a *keyword* is a word reserved for a specific use by the language. For example, `while` is a *keyword*.
- *Variable*: A *variable* is a container for a value within the program. It can store many different values of a *data type*.
- *Function*: As in mathematics, a *function* is a process that does something. Usually it acts on data within the code to either change it, or produce a result based on it.
- *Block of code*: Also referred to as *code block* or just *block* sometimes. It is used to described a bit of code that does a specific thing on its own.
- *Declaration*: When we create a *function* or *variable*, we call that the *declaration*. It is an easy way to refer to the original instantiation of a bit of code. It is also under certain circumstances called *implementation*.

B Modules

There are a lot of modules available to do a multitude of things in Python. We are mostly focusing on math, physics and data analysis, and for that there are a few key modules that are likely to be used very often:

- `numpy`: Math and array / matrices
- `pyplot`: Data visualization
- `pandas`: Dataframe, easy data handling
- `json`: JSON format reader and writer

If a module isn't installed, it can easily be installed using the Python `pip` tool. To install any Python module, open a terminal and use the following command:

```
pip install <module>
```

Where `<module>` is the name of the name of the module to be installed. Once a module is installed, it can be used by the python interpreter, but it doesn't mean the interpreter *knows* that is has to use it when running a given program file. To tell it that it is the case, it is necessary to `import` the required module in the file. To do that, we use the `import` command:

```
import numpy as np
```

In the previous example, we `import` the `numpy` module to be used in the program. The `as` *keyword* allows us to create an alias name for a module, to simplify the notation later on.

C Conditional statements

Effectively, these are tests that the program looks at, and is able to execute a certain *block of code* in function of the result of the test. They are also called `if/else` statements. The semantics of a conditional statement is the call to the right *keyword*, then the Boolean test, then the *block of code* to execute if the result is `True`.

In the following example, we look at an arbitrary variable `a` and check to see what values it possesses.

```
if a < 5:
    print("a is smaller than 5!")
elif a > 15:
    print("a is larger than 15!")
else:
    print("a is between 5 and 15!")
```

D Loops

Loops allow us to go over the same *block of code* and varying only a parameter at each iteration. The idea behind it is simple, we create some condition that will change iteratively, so that we don't have to write the same thing multiple times.

```
for i in range(0, 50, 1):
    print("The square of", i, "is", i**2)
```

In the example above, we compute the square of the first 50 integers. If we didn't use a `for` loop, we would have needed to write manually the operation for each number. A very important thing to understand, is that the `for` loop requires an object that is *iterable*. That means that the object used must offer multiple values, so that each iteration can take place in a meaningful way.

There are other ways to create loops, we can for example use the `while` loop, which uses a condition to establish whether or not the next iteration will happen or not.

E Simple examples

E.1 Declaring a variable

A *variable* stores some data for us, so that we don't have to worry about it ourselves. It can hole multiple *types*, and can be *declared* in a number of ways. Usually, we will simply give it a name, and tell it what to store directly.

```
import numpy as np

# Two examples where we give a value
    manually
```

```
some_number = 3.548
some_text = "This is a a text!"

# Giving a value through a function
an_array = np.linspace(0, 10, 2500)
```

E.2 Declaring a function

To *declare* a *function*, we need to use the `def` *keyword*. We then need to specify a name for our *function*, as well as explicitly tell whether or not it has some input values. The line is terminated with a column, and the lines belonging to the *function* process have to be indented once. At the end of the *function*, we have the option to `return` something out of this process.

For example, we can create a function that returns a polynomial based on input coefficients and point at which to estimate the function.

```
# Defining the polynomial function
def poly(a, b, c, x):
    return a * x**2 + b * x + c

# Storing the return value to a variable
res = poly(2, -3, 7, 2.5)
```

In this example, the *function* `poly` computes the value of the polynomial $f(x) = 2x^2 - 3x + 7$ at $x = 2.5$. Because the function has a `return` value, we can store it to a *variable* to use it later on (in a *variable* called `res` in the example).

E.3 Append to list

To append a value at the end of a list, there is a *function* called `.append()` that let's us do just that. It can any list `ls` of length n, and create the $n + 1$ index and assign it a value.

```
ls = [42, 56, 84]    # List of length 3

ls.append(-10)       # Appending -10

print(ls)            # List is now length 4
```

In the previous example, we have a list of length 3 (max index = 2), and we wanted to create index 3 with a value of -10. Using the `.append()` *function* allows us to do so very easily. Importantly, it also allows us to create a list of identical dimension to a reference when in a `for` loop. By creating an empty list, and appending a given value at each pass, we will obtain a lisst of identical length to our reference one.

```
# Creating a list of length 6, and an empty
    one
x = [0.0, 0.1, 0.2, 0.3, 0.4, 0.5]
y = []

# Going through the values of the first
# and appending to the second one
for value in x:
    y.append(value**2)

# List y is now length 6
print(y)
```

In the previous example, we compute the square of some value in a list that has a fixed length (here 6). Because this list (`x`) could have unknown length, we create and empty list (`y`) to store the result at each point, and need to fill it accordingly. To do so, we go through our reference list `x`, which contains the values we want to use, and append the result to our second list `y`. Because we appended as many times as there are values in the `x` list, the `y` now has the same length as `x`

E.4 Plotting some data

The following is a simple example showing how to use `pyplot` and `numpy` to visualize some data. We are going to plot the sin function for $x \in [-\pi, \pi]$.

```
# Calling the required modules
import numpy as np
import matplotlib.pyplot as pl

# Creating 1000 values for x in the range
# And taking their sinus values
x = np.linspace(-np.pi, np.pi, 1000)
y = np.sin(x)

# Creating a figure
pl.figure()

# Plotting our data
pl.plot(x, y)

# Useful functions for clarity of plot
pl.title("Sinus function")
pl.xlabel("Value of x")
pl.ylabel("Corresponding sinus value")

# Saving the figure
pl.savefig("sinus.pdf")

# Closing the figure
pl.close()
```