Practical session: Introduction to Python

universidad SANJORGE GRUPO SANVALERO

1. Installation

We will use Python bundled with a few key scientific libraries (Numpy, Scipy, and Matplotlib) in an installation called "Anaconda" that you can download from https://www.anaconda.com/products/individual

After installing Anaconda, you can write your first Python program by following the steps at https://docs.anaconda.com/anaconda/user-guide/getting-started/

2. Python and Jupyter

Python is a *programming language*, and an extremely popular one at that. Python is well known for its readability, is consistently ranked as a top programming language, and is used by both top software companies and scientists doing computation.

Python is an example of an *interpreted* programming language, which means that it is converted on-the-fly to machine code. This can make development simpler, but generally executes slower than *compiled* languages, in which the source code must be run through a compiler before it is executed, like Java, C, or C++.

Jupyter is a *programming environment*, often used for writing and sharing Python code. The Jupyter **Notebook** is a type of file that combines documentation and computer code. The notebook itself is comprised of "cells" that either accept Python code, or text written in **Markdown**, a highly simplified syntax for writing HTML code. With the notebook, you can include descriptions of your workflow along with the actual code you used to do your data analysis.

3. Python basics

You already saw how to print text using Python using the `print` command. Now, you are going to practice with a few of Python's basic data structures.

In the following examples, you should execute the commands in a Notebook using code cells, and you should write your comments using markdown cells.

At its most basic level, Python can function like a calculator. For example, you can run simple commands:

```
In []: 2 + 2
```

In addition, Python can work with *strings*, which are textual representations of data. Strings are enclosed in single quotes `' '` or double quotes `" "`. Like numbers, strings can be "added"

together; however, instead of performing a numeric computation, Python will *concatenate* the strings, combining them:

```
In []: x = "a" + "b"
       x
```

Python is an example of a *dynamically typed* language, which means that the types of objects are mutable (can change) and do not need to be declared. Instead, Python detects the type of the object based on the assignment I made. Let's check it out with the `type` command:

```
In []: type(x)
```

Arrays

Python can work with arrays of numbers, such as columns of data or tables of data (rows and columns). However, by default it is set up to handle lists of any kind of data -- perhaps names or addresses, not just numbers -- so we have to use the "array" function from Numpy (numerical Python) to tell Python that a given set of numbers should be treated as a numerical array.

Run the following command to import the Numpy library and to give it the nickname np:

```
In []: import numpy as np
```

We wish to create a numerical array, as opposed to a list of numbers. To see how these differ, first run

```
In []: x = np.array([1,2,3,4])
       y = np.array([4,0,3,2])
       z = x + y
       print(z)
```

and compare the results with:

```
In []: x = [1,2,3,4]
       y = [4,0,3,2]
       z = x + y
       print(z)
```

what is the difference between the two cases? Write your comments on a Markdown cell.

For our purposes, we are *not* interested in the "list" behavior of the second set of commands, but only the "array" behavior of the first set. It's also worth noting that Python overwrites x, y, and z with no error message, even when it means changing their variable types -- this behavior is different from that of programming languages that declare variables.

Special Arrays

Numpy's "`arange`" function can be used to generate a series of numbers, either in +1 increments (the default) or in increments you specify. To understand how functions work, you can type `help(np.arange)`

Compare the output of:

```
In []: x1 = np.arange(1,5)
```

```
In []: x2 = np.arange(1,5,2).
```

What is the increment, the starting point and the end point of the series in each case?

Simple Math

Although python can do advanced math, we won't need that, so you should just remember a few simple operators and functions:

```
+ addition
- subtraction
* multiplication
/ division
** to-the-power-of
```

Note that the operators listed above do math "element-wise", meaning if you, e.g., multiply two single column arrays, the two first elements will multiply, the two second elements will multiply, the two third elements will multiply, etc.

Also, Numpy contains functions to compute the sine, cosine, etc. as well as the value of constants such as π.

Investigate on your own and write the code to obtain the mathematical function: $y = \sin(2\pi f_0 t)$, where t is the array [0 0.1 0.2 … 10] and f_0 equals 0.5.

Now, plot your function. First, you need to import the library Pyplot:

```
In []: import matplotlib.pyplot as plt
```

And then you can plot the function and add labels for the axes and a title:

```
In []: plt.figure()
       plt.plot(t,y)
       plt.title('signal y')
       plt.xlabel('t')
       plt.ylabel('y = sin(2*pi*f*t)')
```

```
plt.show()
```

Defining a Python Function

A good way of creating conveniently reusable chunks of functionality is to define a function. Look at the commands below. The first line starts with the keyword **`def`**, indicating a function definition. The function has the name *convert_distance* and takes a single argument. When the function is called, the argument value provided is assigned to the variable **`miles`**. Note how the *body* of the definition is *indented* relative to the first line (which is not indented). This indentation is crucial because it signals that the indented lines *belong* to the body of the definition.

```
In []:  def convert_distance(miles):
            kilometers = (miles * 8.0) / 5.0;
            print("Converting distance in miles to kilometers:")
            print("Distance in miles: ", miles)
            print("Distance in kilometers:", kilometers)
```

After defining the function, you can use it anywhere in the code:

```
In []: convert_distance(44)
```

In this example we are going to calculate the Fibonacci series:

```
In []:  def fib(n):    # write Fibonacci series up to n
            a, b = 0, 1
            while a < n:
                  print(a, end=' ')
                  a, b = b, a+b
```

```
In []:  fib(10)
```

This example introduces several new features:

- The first line contains a multiple assignment: the variables a and b simultaneously get the new values 0 and 1. On the last line this is used again, demonstrating that the expressions on the right-hand side are all evaluated first before any of the assignments take place. The right-hand side expressions are evaluated from the left to the right.

- The while loop executes as long as the condition (here: a < n) remains true. In Python, like in C, any non-zero integer value is true; zero is false. The condition may also be a string or list value, in fact any sequence; anything with a non-zero length is true, empty sequences are false. The test used in the example is a simple comparison. The standard comparison operators are written the same as in C.

As a final exercise, write a Python function to plot the equation $y = \cos(2\pi f_0 t)$. The function should receive f_0 and the number of periods to be plotted as arguments.

Test your function with different values for f_0 and tmax.

References

This tutorial has been partially adapted from the following online resources:

The Python Tutorial at https://docs.python.org/
Programming I - Intro to Python by Harrison Smith
Introduction to Python and Jupyter by The Carpentries (CC-BY 4.0)