Introduction to lab 1 Python programming and the Python environment

This lab introduces the basics of Python programming, reviews the alternative development environments available for Python development and outlines the libraries we will be covering in the next 4 weeks of the module.

I strongly recommend that rather than simply reading the code examples given below, that you run the interactive Python programming environment, and enter and play around with the code examples given. This way you will start to build the practical knowledge of Python necessary to undertake real data analytics tasks.

Python is already installed, on both Windows and Linux on the ITL machines, therefore, you can skip the section on "installing Python", but this has been left in this document as many of you will wish to set up Python on your own machines.

Introduction to the Python programming environment

This document will give you an overview of the entire Python programming environment. First, we give a description of the Python language and the characteristics that make it unique. You'll see where to start, what an interpreter is, and how to begin to write the first lines of code in Python. Then you are presented with some new, more advanced forms of interactive writing with respect to the shells such as IPython and IPython Notebook.

Python--The Programming Language

Python is a programming language created by Guido Von Rossum in 1991 starting with the previous language called ABC. Python can be characterized by a series of adjectives:

- interpreted
- portable
- object-oriented
- interactive
- interfaced
- open-source
- easy to understand and use

Python is a programming language **interpreted**, that is pseudo-compiled. Once you have written the code of a program, this in order to be run needs an **interpreter**. The interpreter is a program that is installed on each machine that has the task of interpreting the source code and run it. Therefore unlike language such as C, C ++, and Java, there is no compile time.

Python is a highly **portable** programming language. The decision to use an interpreter as an interface for reading and running the code has a key advantage: portability. In fact, you can install on any existing platform (Linux, Windows, Mac) an interpreter specifically adapted to it while the Python code to be interpreted will remain unchanged. Python also, for this aspect, was chosen as the programming language for many small-form devices, such as the Raspberry Pi and other microcontrollers.

Python is an **open-source** programming language. CPython, which is the reference implementation of the Python language is completely free and open-source. Additionally every module or library in the network is open-source and their code is available online. Every month, an extensive developer community brings some improvements to make this language and all its libraries even richer and more efficient. CPython is managed by the nonprofit **Python Software Foundation**, which was created in 2001 and has set itself the task of promoting, protecting, and advancing the Python programming language.

Python--The Interpreter

As described in the previous sections, each time you run the *python* command the Python interpreter starts, characterized by a

>>> prompt.

The Python interpreter is simply a program that reads and interprets the commands passed to the prompt. Each time you press the Enter key, the interpreter begins to scan the code written (either a row or a full file of code) token by token (**tokenization**). These tokens are fragments of text which the interpreter will arrange in a tree structure. The tree obtained is the logical structure of the program which is then converted to **bytecode** (.pyc or .pyo). The process chain ends with the bytecode which will be executed by a **Python virtual machine (PVM)**. See Figure 2-1.

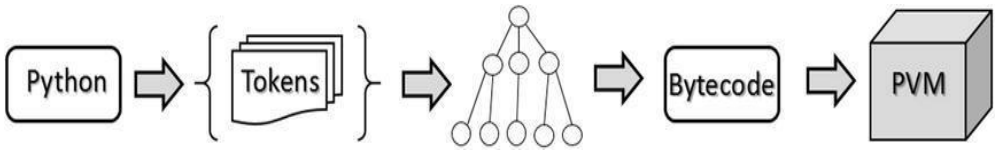

Figure 2-1. *The steps performed by the Python interpreter*

You can find very good documentation on this topic at the link https://www.ics.uci.edu/~pattis/ICS-31/lectures/tokens.pdf.

The standard interpreter of Python is reported as Cython, since it was totally written in C. There are other areas that have been developed using other programming languages such as **Jython**, developed in Java; **IronPython,** developed in C # (and then only for Windows); and **PyPy,** developed entirely in Python.

PyPy

The PyPy interpreter is a JIT (just-in-time) compiler, which converts the Python code directly in machine code at runtime. This choice was made to speed up the execution of Python. However, this choice has led to the use of a small subset of Python commands, defined as **RPython**. For more information on this please consult the official website: http://pypy.org.

Python 2 and Python 3

The Python community is still in transition from interpreters of the Series 2 to Series 3. In fact, currently you will find two releases of Python that are used in parallel (version 2.7 and version 3.4). This kind of ambiguity can create much confusion, especially in terms of choosing which version to use and the differences between these two versions. One question that you surely must be asking is why version 2.x is still being released if it is distributed around a much more enhanced version such as 3.x.

When Guido Van Rossum (the creator of the Python language) decided to bring significant changes to the Python language, he soon found that these changes would make the new Python incompatible with a lot of existing code. Thus he decided to start with a new version of Python called Python 3.0. To overcome the problem of compatibility and create huge amounts of unusable code spreading to the network, it was decided to maintain a compatible version, 2.7 to be precise.

Python 3.0 made its first appearance in 2008, while version 2.7 was released in 2010 with a promise that it would not be followed by big releases, and at the moment the current version is 3.4 (2014).

In this lab we will refer to the Python 3.x version; however, with some few exceptions, there should be no problems with the Python 2.x version.

Installing Python

In order to develop programs in Python you have to install it on your operating system. Differently from Windows, Linux distributions and Mac OS X should already have within them a preinstalled version of Python. If not, or if you would like to replace it with another version, you can easily install it. The installation of Python differs between operating systems; however, it is a rather simple operation.

On Debian-Ubuntu Linux systems

```
apt-get install python
```

On Red Hat, Fedora Linux systems working with rpm packages

```
yum install python
```

If your operating system is Windows or Mac OS X you can go on the official Python site (http://www.python.org) and download the version you prefer. The packages in this case are installed automatically.

However, today there are distributions that provide along with the Python interpreter a number of tools that make the management and installation of Python, all libraries, and associated applications easier. I strongly recommend you choose one of the distributions available online.

Anaconda

Anaconda is a free distribution of Python packages distributed by Continuum Analytics (https://store.continuum.io/cshop/anaconda/). This distribution supports Linux, Windows, and Mac OSX operating systems. Anaconda, provides the latest packages released for the Python programming environment. Indeed, when you install the Anaconda distribution on your system, you have the opportunity to use many tools and applications described in this document, without worrying about having to install and manage each of them separately. The basic distribution includes Spyder as IDE, IPython QtConsole, and Notebook.

The management of the entire Anaconda distribution is performed by an application called **conda**. This is the package manager and the environment manager of the Anaconda distribution that handles all of the packages and their versions.

```
conda install <package name>
```

One of the most interesting aspects of this distribution is the ability to manage multiple development environments, each with its own version of Python. Indeed, when you install Anaconda, the Python version 2.7 is installed by default. All installed packages then will refer to that version. This is not a problem, because Anaconda offers the possibility to work simultaneously and independently with other Python versions by creating a new environment. You can create, for instance, an environment based on Python 3.4.s

```
conda create -n py34 python=3.4 anaconda
```

This will generate a new Anaconda environment with all the packages related to the Python 3.4 version. This installation will not affect in any way the environment built with Python 2.7. Once installed, you can activate the new environment entering the following command.

```
source activate py34
```

on Windows instead: activate py34 C:\Users\”username”> activatepy34

You can create as many versions of Python as you want; you need only to change the parameter passed with the *python* option in the command **conda create**. When you want to return to work with the original Python version you have to use the following command:

source deactivate

on Windows

[py34] C:\Users\"user name">deactivate

Enthought Canopy

There is another distribution very similar to Anaconda and it is the Canopy distribution provided by Enthought, a company founded in 2001 and very famous especially for the SciPy project (https://www.enthought.com/products/canopy/). This distribution supports Linux, Windows, and Mac OX systems and it consists of a large amount of packages, tools, and applications managed by a package manager. The package manager of Canopy, as opposed to conda, is totally graphic.

Unfortunately, only the basic version of this distribution, defined **Canopy Express**, is free; in addition to the package normally distributed, it also includes IPython and an IDE of Canopy that has a special feature that is not present in other IDEs. It has embedded the IPython in order to use this environment as a window for testing and debugging code.

Python(x,y)

Python (x, y) is a free distribution that only works on Windows and is downloadable from http://code.google.com/p/pythonxy/. This distribution uses Spyder as its IDE.

Using Python

Python is a language rich but simple at the same time, very flexible; it allows expansion of your development activities in many areas of work (data analysis, Scientific's, graphic interfaces, etc.). Precisely for this reason, the possibility of using Python can take very many different contexts, often according to the taste and ability of the developer. This section presents the various approaches to using Python.

Python Shell

The easiest way to approach the Python world is to open a session on the Python shell, a terminal running command lines. In fact, you can enter a command line at a time and test its operation immediately. This mode makes clear the nature of the interpreter that underlies the operation of Python. In fact the interpreter is able to read a command at a time, keeping the status of the variables specified in the previous lines, a behavior similar to that of Matlab and other calculation software.

You have the ability to test command every time without having to write, edit, and run an entire program. To start a session on the terminal, simply write in the command line

```
>>> python

Python 3.6.1 |Anaconda custom (64-bit)| (default, May 11 2017, 13:09:58)
[GCC 4.4.7 20120313 (Red Hat 4.4.7-1)] on linux
Type "help", "copyright", "credits" or "license" for more information.

>>>
```

Now the Python shell is active and the interpreter is ready to receive commands in Python. Start by entering the simplest of commands but a classic for getting started with programming.

```
>>> print (“Hello World !”)

Hello World!
```

Run an Entire Program Code

The most familiar way for each programmer is to write an entire program code and then run it from the terminal. First write a program using a simple text editor; you can use as example the code shown in Listing 2-1 and save it as *MyFirstProgram.py*.

Listing 2-1. MyFirstProgram.py

```
myname= input (“What is your name?”)
print (“I am” + myname + “, I am glad to meet you!”)
```

Now you've written your first program in Python, and you can run it directly from the command line by calling the python command and then the name of the file containing the program code.

```
python myFirstProgram.py
What is your name? abcde
I am  abcde, I am glad to meet you !
```

Implement the Code Using an IDE

A more comprehensive approach than the previous ones is the use of an IDE (or **Integrated Development Environment**). These editors are complex software that provide a work environment on which to develop your Python code. They are rich in tools that make life easier for developers, especially when debugging. In the following sections you will see in detail what IDEs are currently available.

Interact with Python

The last approach, and in my opinion, perhaps the most innovative, is the interactive one. In fact, in addition to the three previous approaches, which are those that for better or worse are used by all developers of other programming languages, this approach provides the opportunity to interact directly with the Python code.

In this regard, the Python environment has been greatly enriched with the introduction of **IPython**. IPython is a very powerful tool, designed specifically to meet the needs of interaction between the Python interpreter and the developer, which under this approach takes the role of analyst, engineer, or researcher. In a later section IPython and its features will be explained in more detail.

Writing Python Code

In the previous section you saw how to write a simple program in which the string "Hello World" was printed. In this section you will get a brief overview of the basics of the Python language just to get familiar with the most important basic aspects.

This section is not intended to teach you to program in Python, or to illustrate syntax rules of the programming language, but just to give you a quick overview of some basic principles of Python necessary to continue with the topics covered on this module.

If you already know the Python language you can safely skip this introductory section. Instead if you are not familiar with the programming and you find it difficult to understand the topics, I recommend you see the online documentation, tutorials, and courses of various kinds.

Make Calculations

You have already seen that the **print ()** function is useful for printing almost anything. Python, in addition to being a printing tool, is also a great calculator. Start a session on the Python shell and begin to perform these mathematical operations:

```
>>>1+2
3
>>>(1.045*3)/4
0.78375
>>>4**2
```

```
16
>>>((4+5j)*(2+3j))
(-7+22j)
>>>4<(2*3)
True
```

Python is able to calculate many types of data including complex numbers and conditions with Boolean values. As you can see from the above calculations, the Python interpreter returns directly the result of the calculations without the need to use the **print ()** function. The same thing applies to values contained within variables. It's enough to call the variable to see its contents.

```
>>>a=12*3.4
>>>a
40.8
```

Import New Libraries and Functions

You saw that Python is characterized by the ability to extend its functionality by importing numerous packages and modules available. To import a module in its entirety you have to use the **import** command.

```
>>> import math
```

In this way all the functions contained within the *math* package are available in your Python session so you can call them directly. Thus you have extended the standard set of functions available when you start a Python session. These functions are called with the following expression.

```
library_name.function_name()
```

For example, you are now able to calculate the sine of the value contained within the variable *a*.

```
>>> math.sin(a)
```

As you can see the function is called along with the name of the library. Sometimes you might find the following expression for declaring an import.

```
>>> from math import *
```

Even if this works properly, it is to be avoided for a good practice. In fact writing an import in this way involves the importation of all functions without necessarily defining the library to which they belong.

```
>>>sin(a)
0.040693257349864856
```

This form of import can actually lead to very large errors, especially if the imported libraries are beginning to be numerous. In fact, it is not unlikely that different libraries have functions with the same name, and importing all of these would result in an override of all functions with the same name previously imported. Therefore the behavior of the program could generate numerous errors or worse, abnormal behavior.

Actually, this way to import is generally used for only a limited number of functions, that is, functions that are strictly necessary for the functioning of the program, thus avoiding the importation of an entire library when it is completely unnecessary.

```
>>> from math import sin
```

Data Structure

You saw in the previous examples how to use simple variables containing a single value. Actually Python provides a number of extremely useful data structures. These data structures are able to contain several data simultaneously, and sometimes even of different types. The various data structures provided are defined differently depending on how their data are structured internally.

- list
- set
- strings
- tuples
- dictionary
- deque
- heap

This is only a small part of all the data structures that can be made with Python. Among all these data structures, the most commonly used are **dictionaries** and **lists**.

The type **dictionary**, defined also as **dicts**, is a data structure in which each particular value is associated with a particular label called **key**. The data collected in a dictionary have no internal order but only definitions of key/value pairs.

```
>>> dict = {'name':'William', 'age':25, 'city':'London'}
```

If you want to access a specific value within the dictionary you have to indicate the name of the associated key.

```
>>>dict["name"]
'William'
```

If you want to iterate the pairs of values in a dictionary you have to use the **for-in** construct. This is possible through the use of the **items()** function.

```
>>>for key, value in dict.items():
...    print(key,value)
...
name William
city London
age 25
```

The type **list** is a data structure that contains a number of objects in a precise order to form a sequence to which elements can be added and removed. Each item is marked with a number corresponding to the order of the sequence, called **index**.

```
>>> list = [1,2,3,4]
>>> list
[1, 2, 3, 4]
```

If you want to access the individual elements it is sufficient to specify the index in square brackets (the first item in the list has 0 as index), while if you take out a portion of the list (or a sequence), it is sufficient to specify the range with the indices *i* and *j* corresponding to the extremes of the portion.

```
>>> list[2]
3
>>> list[1:3]
[2, 3]
```

Instead if you are using negative indices, this means you are considering the last item in the list and gradually moving to the first.

```
>>> list[-1]
4
```

In order to do a scan of the elements of a list you can use the **for-in** construct.

```
>>> items = [1,2,3,4,5]
>>> for item in items:
...       item + 1
...
2
3
4
```

5
6

Functional Programming (Python >= 3.4)

The **for-in** loop shown in the previous example is very similar to those found in other programming languages. But actually, if you want to be a "Python" developer you have to avoid using explicit loops. Python offers other alternative approaches, specifying these programming techniques such as **functional programming** (expression-oriented programming).

The tools that Python provides to develop functional programming comprise a series of functions:

- map(function, list)
- filter(function, list)
- reduce(function, list)
- lambda
- list comprehension

The *for* loop that you have just seen has a specific purpose, which is to apply an operation on each item and then gather the result. This can be done by the **map()** function.

```
>>> items = [1,2,3,4,5]
>>> def inc(x): return x+1
...
>>> list(map(inc,items))
[2, 3, 4, 5, 6]
```

In the previous example, first you have defined the function that performs the operation on every single element, and then you have passed it as the first argument to the *map()*. Python allows you to define the function directly within the first argument using **lambda** as a function. This greatly reduces the code, and compacts the previous construct, in a single line of code.

```
>>> list(map((lambda x: x+1),items))
[2, 3, 4, 5, 6]
```

Two other functions that work in a similar way are **filter()** and **reduce()**. The **filter()** function extracts the elements of the list for which the function returns True. The **reduce()** function instead considers all the elements of the list to produce a single result. To use **reduce()**, you must import the module **functools**.

```
>>> list(filter((lambda x: x < 4), items))
[1, 2, 3]
>>> from functools import reduce
```

```
>>> reduce((lambda x,y: x/y), items)
0.008333333333333333
```

Both of these functions implement other types of using the *for* loop. They are going to replace these cycles and their functionality, which can be alternatively expressed with simple functions calling. That is what constitutes **functional programming**.

The final concept of functional programming is **list comprehension**. This concept is used to build lists in a very natural and simple way, referring to them in a manner similar to how the mathematicians describe data sets. The values of the sequence are defined through a particular function or operation.

```
>>> S = [x**2 for x in range(5)]
>>> S
[0, 1, 4, 9, 16]
```

Indentation

A peculiarity for those coming from other programming languages is the role that **indentation** plays. Whereas you used to manage the indentation for purely aesthetic reasons, making the code somewhat more readable, in Python it assumes an integral role in the implementation of the code, dividing it into logical blocks. In fact, while in Java, C, and C ++ each command line of code is separated from the next by a ';', in Python you should not specify any symbol that separates them, included the braces to indicate a logical block. These roles in Python are handled through indentation; that is, depending on the starting point of the line of code, the interpreter considers that line whether it belongs to a logical block or not.

```
>>> a = 5
>>> if a > 3:
...       if a < 5:
...           print("I'm four")
...   else:
...      print("I'm a little number")
...
```
Nothing would be printed in this case !

```
>>> if a > 3:
...       if a < 5:
...           print("I'm four")
...       else:
...           print("I'm a big number")
...
I'm a big number
```

In this example you can see that depending on how the else command is indented, the conditions assume two different meanings.

IPython

IPython is a further development of Python that includes a number of tools: **IPython shell**, a powerful interactive shell resulting in a greatly enhanced Python terminal; a **QtConsole,** which is a hybrid between a shell and a GUI, allowing in this way to display graphics inside the console instead of in separate windows; and finally, the **IPython Notebook,** which is a web interface that allows you to mix text, executable code, graphics, and formulas in a single representation.

IPython Shell

This shell apparently resembles a Python session run from a command line, but actually, it provides many other features that make this shell much more powerful and versatile than the classic one. To launch this shell just type *ipython* in the command line.

```
> ipython
Python 2.7.8 (default, Jul  2 2014, 15:12:11) [M
Type "copyright", "credits", or "license" for more information.

IPython 2.4.1 -- An enhanced Interactive Python.
?         -> Introduction and overview of IPython's features.
%quickref -> Quick reference.
help      -> Python's own help system.
object?   -> Details about 'object', use 'object??' for extra details.

In [1]:
```

As you can see, a particular prompt appears with the value *In [1]*. This means that it is the first line of input. Indeed, IPython offers a system of numbered prompts (indexed) with input and output caching.

```
In [1]: print "Hello World!"
Hello World!

In [2]: 3/2
Out[2]: 1

In [3]: 5.0/2
Out[3]: 2.5

In [4]:
```

The same thing applies to values in output that are indicated with the value *Out[1]*, *Out [2]*, and so on. IPython saves all inputs that you enter storing them as variables. In fact, all the inputs entered were included as fields within a list called **In**.

```
In [4]: In
Out[4]: ['', u'print "Hello World!"', u'3/2', u'5.0/2', u'_i2', u'In']
```

The indices of the list elements are precisely the values that appear in each prompt. Thus, to access a single line of input you can simply specify precisely that value.

```
In [5]: In[3]
Out[5]: u'5.0/2'
```

Even for output you can apply the same.

```
{2: 1,
 3: 2.5,
 4: ['',
  u'print "Hello World!"',
  u'3/2',
  u'5.0/2',
  u'_i2',
  u'In',
  u'In[3]',
  u'Out'],
 5: u'5.0/2'}
```

IPython Qt-Console

In order to launch this application from the command line you must enter the following command:

```
ipython qtconsole
```

The application consists of a GUI in which you have all the functionality present in the IPython shell. See Figure 2-2.

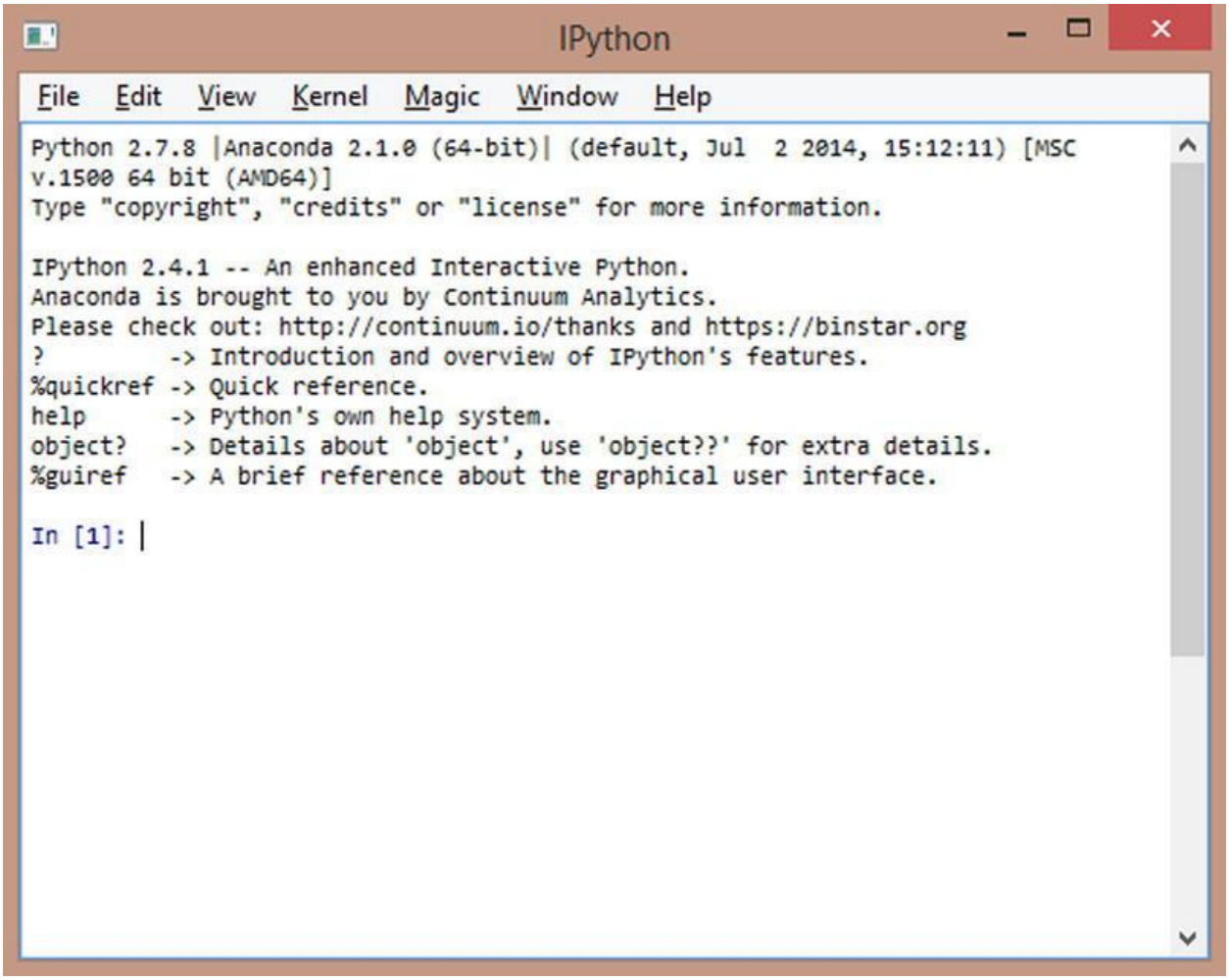

***Figure 2-2**. The IPython QtConsole*

IPython Notebook

IPython Notebook is the latest evolution of this interactive environment (see Figure 2-3). In fact, with IPython Notebook, you can merge executable code, text, formulas, images, and animations into a single Web document, useful for many purposes such as presentations, tutorials, debugging, and so forth.

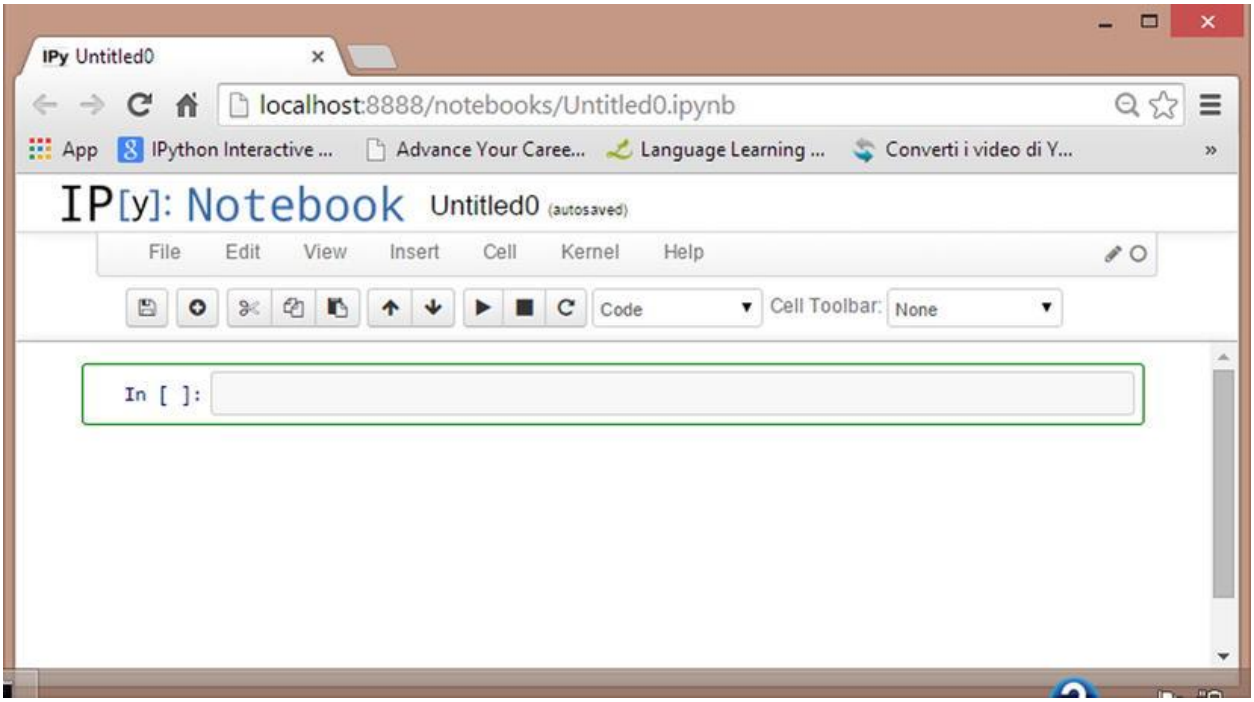

Figure 2-3*. The web page showing the IPython Notebook*

The Jupyter Project

IPython is a project that has grown enormously in recent times, and with the release of IPython 3.0, everything is moving toward a new project called **Jupyter** (https://jupyter.org).

IPython will continue to exist as a Python shell, and as a kernel of Jupyter, but the Notebook and the other language-agnostic components belonging to the IPython project will all move to form the new Jupyter project.

PyPI--The Python Package Index

The **Python Package Index (PyPI)** is a software repository that contains all the software needed for programming in Python, for example, all Python packages belonging to other Python libraries. The content repository is managed directly by the developers of individual packages that deal with updating the repository with the latest versions of their released libraries. For a list of the packages contained within the repository you should go to see the official page of PyPI with this link: https://pypi.python.org/pypi.

As far as the administration of these packages, you can use the **pip** application which is the package manager of PyPI.

Launching it from the command line, you can manage all the packages individually deciding if the package is to be installed, upgraded, or removed. Pip will check if the package is already installed, of if it needs to be updated, to control dependencies, that is, to assess whether other packages are necessary. Furthermore, it manages their downloading and installation.

```
$ pip install <<package_name>>
$ pip search <<package_name>>
$ pip show <<package_name>>
$ pip uninstall <<package_name>>
```

Regarding the installation, if you have Python 3.4+ (released March 2014) and Python 2.7.9+ (released December 2014) already installed on your system, the pip software is already included in these releases of Python. However, if you are still using an older version of Python you need to install pip on your system. The installation of pip on your system depends on the operating system on which you are working.

On Linux Debian-Ubuntu:

```
$ sudo apt-get install python-pip
```

On Linux Fedora

```
$ sudo yum install python-pip
```

On Windows:

Visit the site *www.pip-installer.org/en/latest/installing.html* and download **get-pip.py** on your PC. Once the file is downloaded, run the command

```
python get-pip.py
```

In this way, you will install the package manager. Remember to add *C:\Python2.X\Scripts* in the PATH environment variable.

The IDEs for Python

Although most of the Python developers are used to implement their code directly from the shell (Python or IPython), some **IDEs (Interactive Development Environments)** are also available. In fact, in addition to a text editor, these graphics editors also provide a series of tools very useful during the drafting of the code. For example, the auto-completion of code, viewing the documentation associated with the commands, debugging, and breakpoints are only some of the tools that this kind of application can provide.

IDLE (Integrated Development Environment)

IDLE is a very simple IDE created specifically for development in Python. It is the official IDE included in the standard Python release, so it is embedded within the standard distribution of Python (see Figure 2-5). IDLE is a piece of software that is fully implemented in Python.

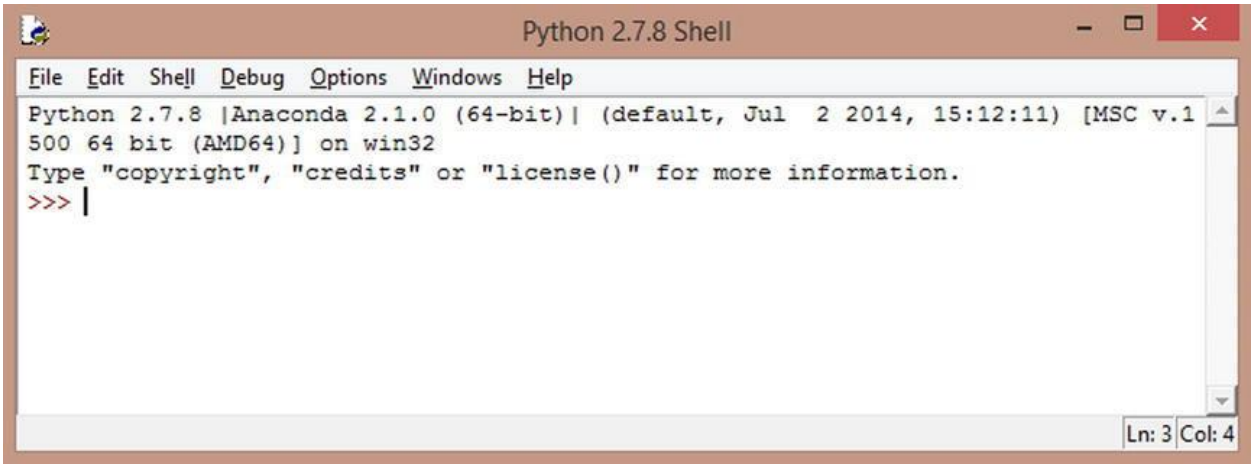

Figure 2-5. *The IDLE Python shell*

Spyder

Spyder (Scientific Python Development Environment) is an IDE that has similar features to the IDE of Matlab (see Figure 2-6). The text editor is enriched with syntax highlighting and code analysis tools. Also, using this IDE you have the option to integrate ready-to-use widgets in your graphic applications. Please note that when you install anaconda, the Spyder gets installed by default.

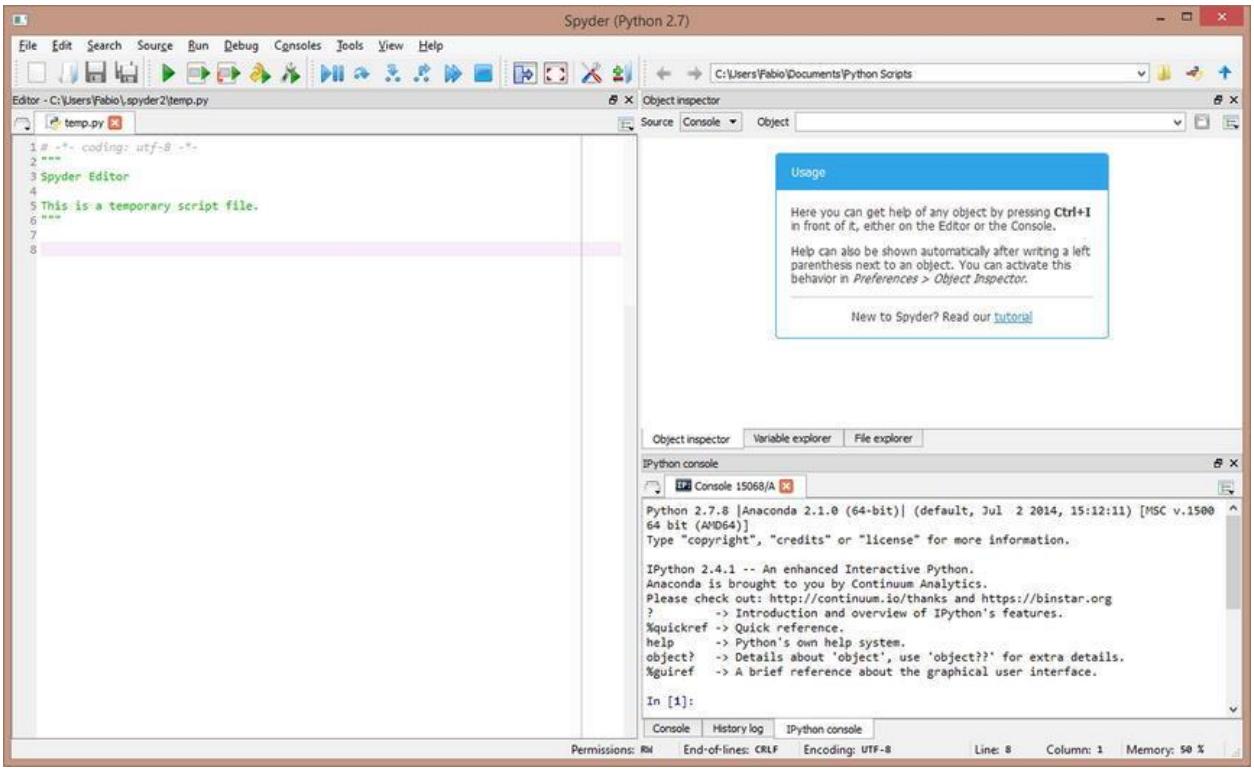

Figure 2-6. *The Spyder IDE*

Eclipse (pyDev)

Those who developed in other programming languages certainly know Eclipse, a universal IDE developed entirely in Java (therefore requiring Java installation on your PC) that provides a development environment for many programming languages (see Figure 2-7). So there is also an Eclipse version for developing in Python thanks to the installation of an additional plug-in called **pyDev**.

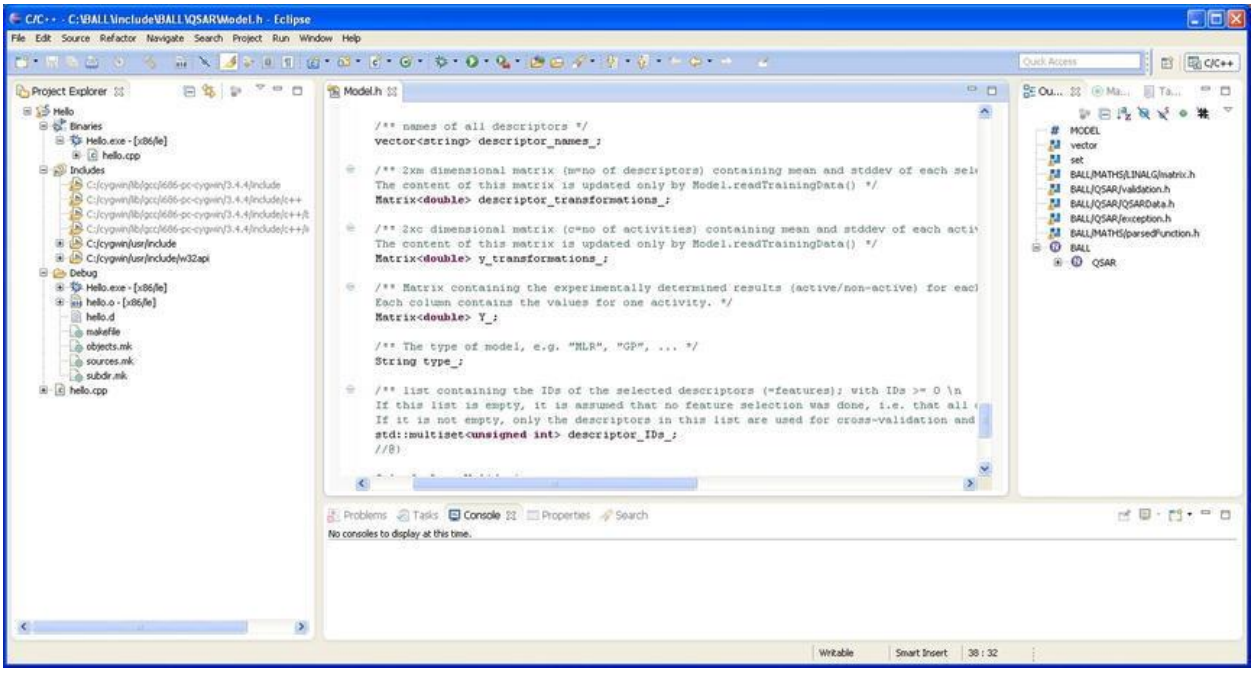

Figure 2-7. *The Eclipse IDE*

Sublime

This text editor is one of the preferred environment for Python programmers (see Figure 2-8). In fact, there are several plug-ins available for this application that make Python implementation easy and enjoyable.

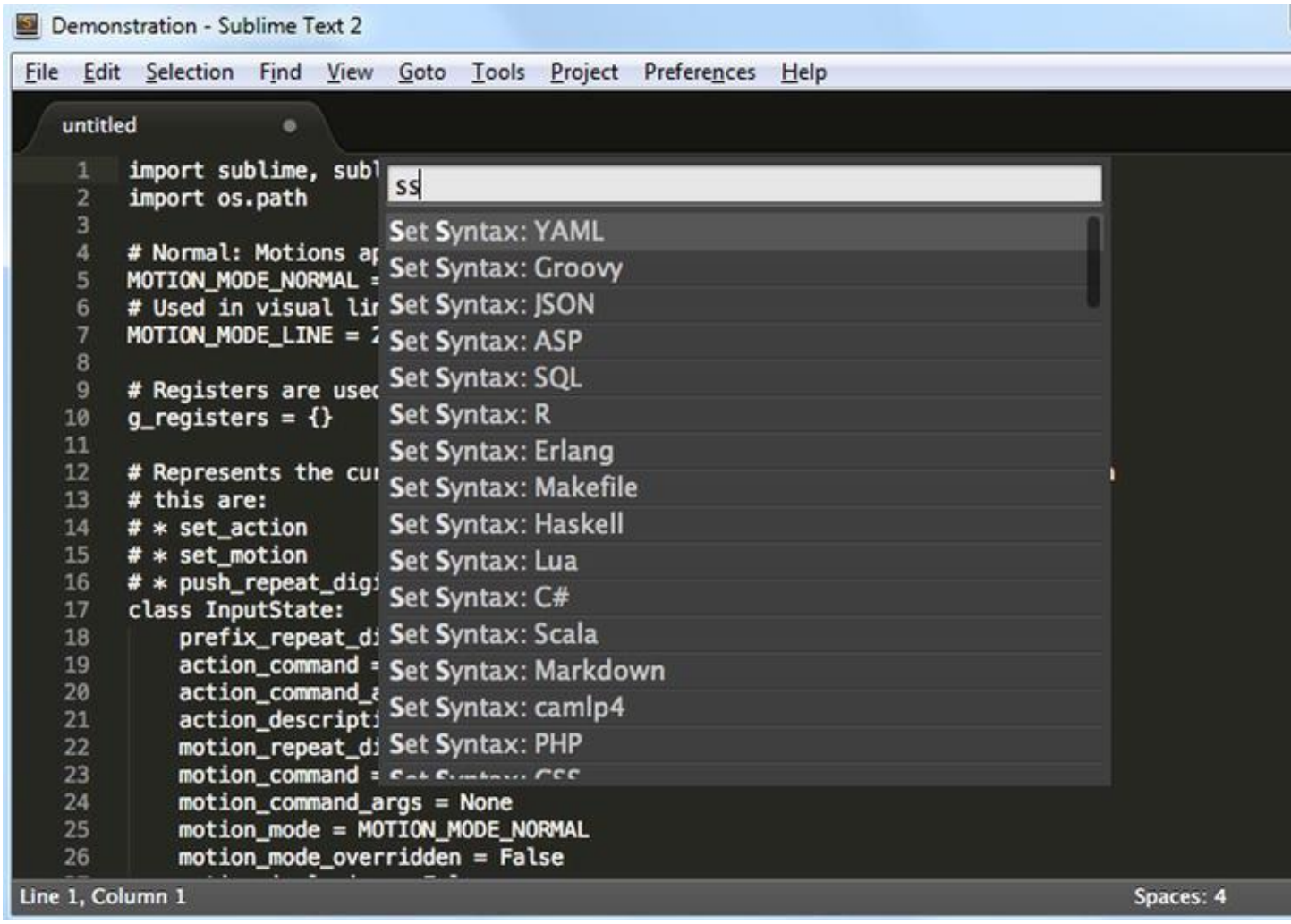

Figure 2-8. The Sublime IDE

Liclipse

Liclipse, similarly to Spyder, is a development environment specifically designed for the Python language (see Figure 2-9). Basically it is totally similar to the Eclipse IDE but it is fully adapted for a specific use of Python, without installing plug-ins like PyDev. So its installation and its setting are much simpler than Eclipse.

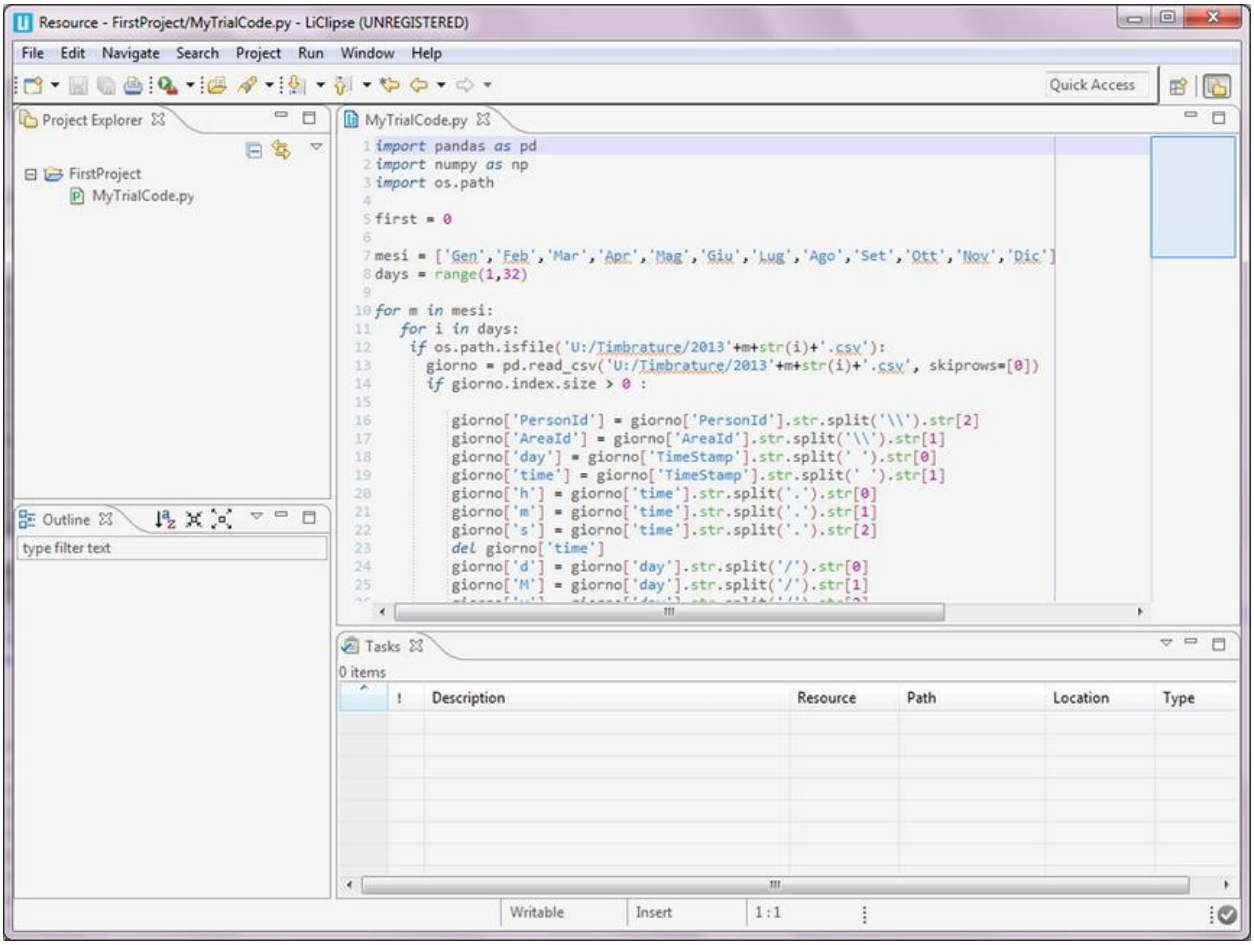

Figure 2-9. The Liclipse IDE

NinjaIDE

NinjaIDE (NinjaIDE is "Not Just Another **IDE**") characterized by a name that is a recursive acronym, is a specialized IDE for the Python language. It's a very recent application on which the efforts of many developers are focused. Being already very promising, it is likely that in the coming years, this IDE will be a source of many surprises.

Komodo IDE

Komodo is a very powerful IDE full of tools that make it a complete and professional development environment. Paid software, written in C ++, Komodo is an IDE that provides a development environment adaptable to many programming languages, including Python.

SciPy

SciPy (pronounced "Sigh Pie") is a set of open-source Python libraries specialized for scientific computing. Many of these libraries will be key to the content of this module. given that their knowledge is critical to data analysis. Together they constitute a set of tools for calculating and displaying data that compares well with other specialized environments for calculation and data analysis (such as R or Matlab). Among the libraries that are part of the SciPy group, there are some in particular that will be discussed in the module, including:

- NumPy
- Matplotlib
- Pandas

NumPy

This library, whose name means Numerical Python, constitutes the core of many other Python libraries that have originated from it. Indeed, NumPy is the foundation library for scientific computing in Python since it provides data structures and high-performing functions that the basic package of Python does not provide. In fact, as you will see later in the module, NumPy defines a specific data structure that is an *N*-dimensional array defined as **ndarray**.

Knowledge of this library is essential in terms of numerical calculations since its correct use can greatly influence the performance of a computation. Throughout the module, this library will be almost omnipresent because of its unique characteristics.

This package provides some features that will be added to the standard Python:

- ndarray: a multidimensional array much faster and more efficient than those provided by the basic package of Python.
- element-wise computation: a set of functions for performing this type of calculation with arrays and mathematical operations between arrays.
- reading-writing data sets: a set of tools for reading and writing data stored on disk.
- Integration with other languages such as C, C ++, and FORTRAN: a set of tools to integrate code developed with these programming languages

Pandas

This package provides complex data structures and functions specifically designed to make the work on them easy, fast, and effective. This package is the core for data analysis with Python. The fundamental concept of this package is the **DataFrame**, a two-dimensional tabular data structure with row and column labels.

Pandas combines the high performance properties of the NumPy library to apply them to the manipulation of data in spreadsheets or in relational databases (SQL database). In fact, using

sophisticated indexing it will be easy to carry out many operations on this kind of data structures, such as reshaping, slicing, aggregations, and the selection of subsets.

matplotlib

This package is the Python library that is currently most popular for producing plots and other data visualizations in 2D. Since data analysis requires visualization tools, this is the library that best suits this purpose.

Conclusions

In the course of this lab all the fundamental aspects characterizing the Python's environment have been illustrated. The Python programming language is introduced in its basic concepts with brief examples, explaining the innovative aspects that it introduces and especially how it stands out compared to other programming languages. In addition, different ways of using Python at various levels have been presented. First you have seen how to use a simple command-line interpreter, then a set of simple graphical user interfaces are shown until you get to such complex development environments, known as IDE, such as Spyder and NinjaIDE.

The highly innovative project IPython was presented, showing the possibility to develop the Python code interactively, in particular with the IPython Notebook.

Moreover, the modular nature of Python has been highlighted with the ability to expand the basic set of standard functions provided by Python with external libraries. In this regard, the PyPI online repository has been shown along with other Python distributions such as Anaconda and Enthought Canopy.sss