

CHAPTER 2

Introduction to the Python World

The Python language, and the world around it, is made by interpreters, tools, editors, libraries, notebooks, etc. This Python world has expanded greatly in recent years, enriching and taking forms that developers who approach it for the first time can sometimes find complicated and somewhat misleading. Thus if you are approaching Python for the first time, you might feel lost among so many choices, especially on where to start.

This chapter gives you an overview of the entire Python world. First you will read a description of the Python language and its unique characteristics. You'll see where to start, what an interpreter is, and how to begin writing 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

The Python programming language was created by Guido Von Rossum in 1991 and started with a previous language called ABC. This language can be characterized by a series of adjectives:

- Interpreted
- Portable
- Object-oriented
- Interactive
- Interfaced
- Open source
- Easy to understand and use

Python is an *interpreted* programming language, that is, it's pseudo-compiled. Once you write the code, you need an *interpreter* to run it. The interpreter is a program that is installed on each machine that has the task of interpreting the source code and running it. Unlike with languages such as C, C++, and Java, there is no compile time with Python.

Python is a highly *portable* programming language. The decision to use an interpreter as an interface for reading and running code has a key advantage: portability. In fact, you can install an interpreter on any platform (Linux, Windows, and Mac) and the Python code will not change. Because of this, Python is often used as the programming language for many small-form devices, such as the Raspberry Pi and other microcontrollers.

Python is an *object-oriented* programming language. In fact, it allows you to specify classes of objects and implement their inheritance. But unlike C++ and Java, there are no constructors or destructors. Python also allows you to implement specific constructs in your code to manage exceptions. However, the structure of the language is so flexible that it allows you to program with alternative approaches with respect to the object-oriented one. For example, you can use functional or vectorial approaches.

Python is an *interactive* programming language. Thanks to the fact that Python uses an interpreter to be executed, this language can take on very different aspects depending on the context in which it is used. In fact, you can write code made of a lot of lines, similar to what you might do in languages like C++ or Java, and then launch the program, or you can enter the command line at once and execute it, immediately getting the results of the command. Then, depending on the results, you can decide what command to run next. This highly interactive way to execute code makes the Python computing environment similar to MATLAB. This feature of Python is one reason it's popular with the scientific community.

Python is a programming language that can be *interfaced*. In fact, this programming language can be interfaced with code written in other programming languages such as C/C++ and FORTRAN. Even this was a winning choice. In fact, thanks to this aspect, Python can compensate for what is perhaps its only weak point, the speed of execution. The nature of Python, as a highly dynamic programming language, can sometimes lead to execution of programs up to 100 times slower than the corresponding static programs compiled with other languages. Thus the solution to this kind of performance problem is to interface Python to the compiled code of other languages by using it as if it were its own.

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 includes 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 given itself the task of promoting, protecting, and advancing the Python programming language.

Finally, Python is a simple language to use and learn. This aspect is perhaps the most important, because it is the most direct aspect that a developer, even a novice, faces. The high intuitiveness and ease of reading of Python code often leads to “sympathy” for this programming language, and consequently it is the choice of most newcomers to programming. However, its simplicity does not mean narrowness, since Python is a language that is spreading in every field of computing. Furthermore, Python is doing all of this so simply, in comparison to existing programming languages such as C++, Java, and FORTRAN, which by their nature are very complex.

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. You have seen that the interpreter can accept either a single command at a time or entire files of Python code. However the approach by which it performs this is always the same.

Each time you press the Enter key, the interpreter begins to scan the code (either a row or a full file of code) token by token (called *tokenization*). These tokens are fragments of text that the interpreter arranges 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 that will be executed by a *Python virtual machine (PVM)*. See Figure 2-1.

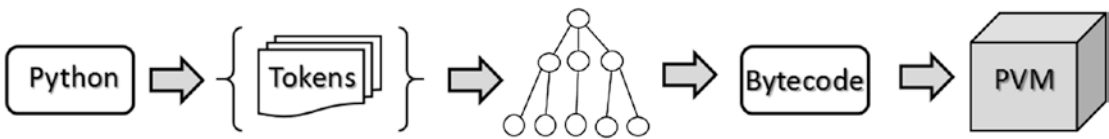


Figure 2-1. The steps performed by the Python interpreter

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

The standard Python interpreter is reported as Cython, since it was 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# (only for Windows); and PyPy, developed entirely in Python.

Cython

The Cython project is based on creating a compiler that translates Python code into C. This code is then executed within a Cython environment at runtime. This type of compilation system has made it possible to introduce C semantics into the Python code to make it even more efficient. This system has led to the merging of two worlds of programming language with the birth of *Cython*, which can be considered a new programming language. You can find a lot of documentation about it online; I advise you to visit <http://docs.cython.org>.

Jython

In parallel to Cython, there is a version totally built and compiled in Java, named *Jython*. It was created by Jim Hugunin in 1997 (<http://www.jython.org>). Jython is an implementation of the Python programming language in Java; it is further characterized by using Java classes instead of Python modules to implement extensions and packages of Python.

PyPy

The PyPy interpreter is a JIT (just-in-time) compiler, and it 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 smaller subset of Python commands, defined as *RPython*. For more information on this, consult the official website at <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, you will currently find two releases of Python that are used in parallel (version 2.7 and version 3.6). This kind of ambiguity can create 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 Python) decided to bring significant changes to the Python language, he soon found that these changes would make the new version 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 incompatibility and avoid creating huge amounts of unusable code, 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 3.6.5 (2018).

In the book we refer to the Python 3.x version; however, with a few exceptions, there should be no problem with the Python 2.7.x version (the last version is 2.7.14 and was released in September 2017).

Installing Python

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

On Debian-Ubuntu Linux systems, run this command

```
apt-get install python
```

On Red Hat Fedora Linux systems working with rpm packages, run this command

```
yum install python
```

If you are running Windows or MacOS X, you can go to 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 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.

Python Distributions

Due to the success of the Python programming language, many Python tools have been developed to meet various functionalities over the years. There are so many that it's virtually impossible to manage all of them manually.

In this regard, many Python distributions efficiently manage hundreds of Python packages. In fact, instead of individually downloading the interpreter, which includes only the standard libraries, and then needing to individually install all the additional libraries, it is much easier to install a Python distribution.

At the heart of these distributions are the *package managers*, which are nothing more than applications that automatically manage, install, upgrade, configure, and remove Python packages that are part of the distribution.

Their functionality is very useful, since the user simply makes a request on a particular package (which could be an installation for example), and the package manager, usually via the Internet, performs the operation by analyzing the necessary version, alongside all dependencies with any other packages, and downloading them if they not present.

Anaconda

Anaconda is a free distribution of Python packages distributed by Continuum Analytics (<https://www.anaconda.com>). This distribution supports Linux, Windows, and MacOS X operating systems. Anaconda, in addition to providing the latest packages released in the Python world, comes bundled with most of the tools you need to set up a Python development environment.

Indeed, when you install the Anaconda distribution on your system, you can use many tools and applications described in this chapter, without worrying about having to install and manage each separately. The basic distribution includes Spyder as the 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 and it 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 3.6.

```
conda create -n py36 python=3.6 anaconda
```

This will generate a new Anaconda environment with all the packages related to the Python 3.6 version. This installation will not affect in any way the environment built with Python 2.7. Once it's installed, you can activate the new environment by entering the following command.

```
source activate py36
```

On Windows, use this instead:

```
activate py36
```

```
C:\Users\Fabio>activate py36
(py36) C:\Users\Fabio>
```

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 `conda create` command. When you want to return to work with the original Python version, you have to use the following command:

```
source deactivate
```

On Windows, use this:

```
(py36) C:\Users\Fabio>deactivate
Deactivating environment "py36"...
C:\Users\Fabio>
```

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 known for the SciPy project (<https://www.enthought.com/products/canopy/>). This distribution supports Linux, Windows, and MacOS X 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 graphical.

Unfortunately, only the basic version of this distribution, called *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 works only on Windows and is downloadable from <http://code.google.com/p/pythonxy/>. This distribution uses Spyder as the IDE.

Using Python

Python is rich but simple and very flexible. It allows expansion of your development activities in many areas of work (data analysis, scientific, graphic interfaces, etc.). Precisely for this reason, Python can be used in many different contexts, often according to the taste and ability of the developer. This section presents the various approaches to using Python in the course of the book. According to the various topics discussed in different chapters, these different approaches will be used specifically, as they will be more suited to the task at hand.

Python Shell

The easiest way to approach the Python world is to open a session in the Python shell, which is a terminal running a command line. In fact, you can enter one command at a time and test its operation immediately. This mode makes clear the nature of the interpreter that underlies Python. In fact, the interpreter can read one 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.

This approach is helpful when approaching Python the first time. You can test commands one at a time without having to write, edit, and run an entire program, which could be composed of many lines of code.

This mode is also good for testing and debugging Python code one line at a time, or simply to make calculations. To start a session on the terminal, simply type this on the command line:

```
>>> python
```

```
Python 3.6.3 (default, Oct 15 2017, 03:27:45) [MSC v.1900 64 bit (AMD64)]
on win32
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

The best way to become familiar with Python is to write an entire program and then run it from the terminal. First write a program using a simple text editor. For example, you can use 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("Hi " + myname + ", I'm glad to say: Hello world!")
```

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? Fabio Nelli
Hi Fabio Nelli, I'm glad to say: Hello world!
```

Implement the Code Using an IDE

A more comprehensive approach than the previous ones is the use of an IDE (an *Integrated Development Environment*). These editors provide a work environment on which to develop your Python code. They are rich in tools that make developers' lives easier, 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, this approach provides you the opportunity to interact directly with the Python code.

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

Writing Python Code

In the previous section you saw how to write a simple program in which the string "Hello World" was printed. Now in this section you will get a brief overview of the basics of the Python language.

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 in this book.

If you already know the Python language, you can safely skip this introductory section. Instead if you are not familiar with programming and you find it difficult to understand the topics, I highly recommend that you visit 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 can calculate many types of data including complex numbers and conditions with Boolean values. As you can see from these calculations, the Python interpreter directly returns the result of the calculations without the need to use the `print()` function. The same thing applies to values contained in 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. To import a module in its entirety, you have to use the `import` command.

```
>>> import math
```

In this way all the functions contained in 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 can now calculate the sine of the value contained in 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 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 lead to very large errors, especially if the imported libraries are 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. Python provides a number of extremely useful data structures. These data structures are able to contain lots of data simultaneously and sometimes even data 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 a *key*. The data collected in a dictionary have no internal order but are 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
age 25
city London
```

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 the *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 its 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]
```

If you are using negative indices instead, 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:
...     print(item + 1)
...
2
3
4
5
6
```

Functional Programming

The `for-in` loop shown in the previous example is very similar to loops found in other programming languages. But actually, if you want to be a “Python” developer, you have to avoid using explicit loops. Python offers alternative approaches, specifying 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 somehow 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, it first defines the function that performs the operation on every single element, and then it passes it as the first argument to `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 into a single line of code.

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

Two other functions working 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 by using the `for` loop. They replace these cycles and their functionality, which can be alternatively expressed with simple functions. 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 mathematicians describe datasets. The values in 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 indentation assumes an integral role in the implementation of the code, by dividing it into logical blocks. In fact, while in Java, C, and C++, each line of code is separated from the next by a semicolon (;), 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 code line, the interpreter determines whether it belongs to a logical block or not.


```

>>> a = 4
>>> if a > 3:
...     if a < 5:
...         print("I'm four")
...     else:
...         print("I'm a little number")
...
I'm four

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

```

In this example you can see that depending on how the else command is indented, the conditions assume two different meanings (specified by me in the strings themselves).

IPython

IPython is a further development of Python that includes a number of tools:

- The IPython shell, which is a powerful interactive shell resulting in a greatly enhanced Python terminal.
- A QtConsole, which is a hybrid between a shell and a GUI, allowing you to display graphics inside the console instead of in separate windows.
- 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` on the command line.

```
> ipython
Python 3.6.3 (default, Oct 15 2017, 3:27:45) [MSC v.1900 64bit (AMD64)]
Type "copyright", "credits", or "license" for more information.
```

```
IPython 6.1.0 -- An enhanced Interactive Python. Type '?' for help
```

```
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]: 1.5
```

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

```
In [4]:
```

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

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

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

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

For output, you can apply the same concept.

```
In [6]: Out
Out[6]:
{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'}
```

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>)—see Figure 2-2.



Figure 2-2. *The Jupyter project logo*

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 move to form the new Jupyter project.

Jupyter QtConsole

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

```
ipython qtconsole
```

or

```
jupyter qtconsole
```

The application consists of a GUI that has all the functionality present in the IPython shell. See Figure 2-3.

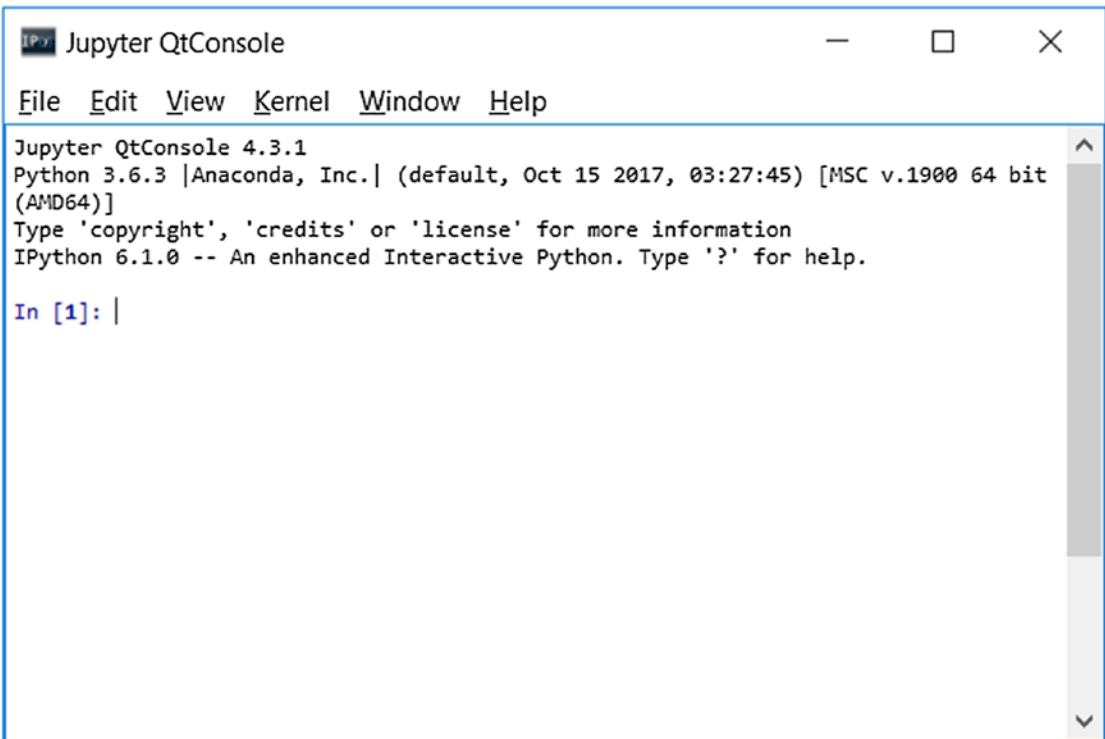


Figure 2-3. The IPython QtConsole

Jupyter Notebook

Jupyter Notebook is the latest evolution of this interactive environment (see Figure 2-4). In fact, with Jupyter Notebook, you can merge executable code, text, formulas, images, and animations into a single Web document. This is useful for many purposes such as presentations, tutorials, debug, and so forth.

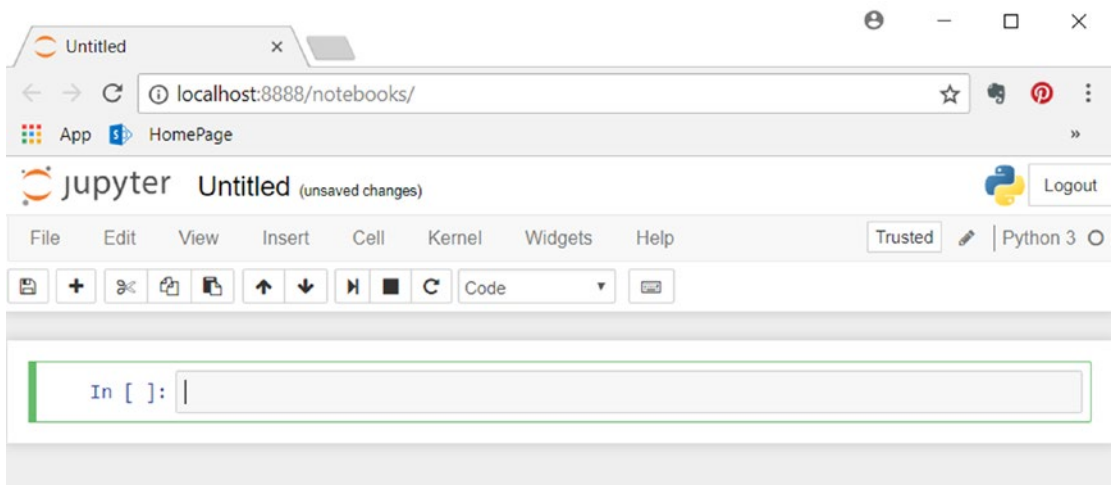


Figure 2-4. The web page showing the Jupyter Notebook

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 in the repository, go to the official page of PyPI at <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.

By launching it from the command line, you can manage all the packages and individually decide if a package should be installed, upgraded, or removed. Pip will check if the package is already installed, or if it needs to be updated, to control dependencies, and to assess whether other packages are necessary. Furthermore, it manages the downloading and installation processes.

```
$ 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, use this command:

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

On Linux Fedora, use this command:

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

On Windows, visit <https://pip.pypa.io/en/latest/installing/> and download `get-pip.py` onto your PC. Once the file is downloaded, run this command:

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

This way, you will install the package manager. Remember to add `C:\Python3.X\Scripts` to the PATH environment variable.

The IDEs for Python

Although most of the Python developers are used to implementing 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 that are 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.

Spyder

Spyder (Scientific Python Development Environment) is an IDE that has similar features to the IDE of MATLAB (see Figure 2-5). The text editor is enriched with syntax highlighting and code analysis tools. Also, you can integrate ready-to-use widgets in your graphic applications.

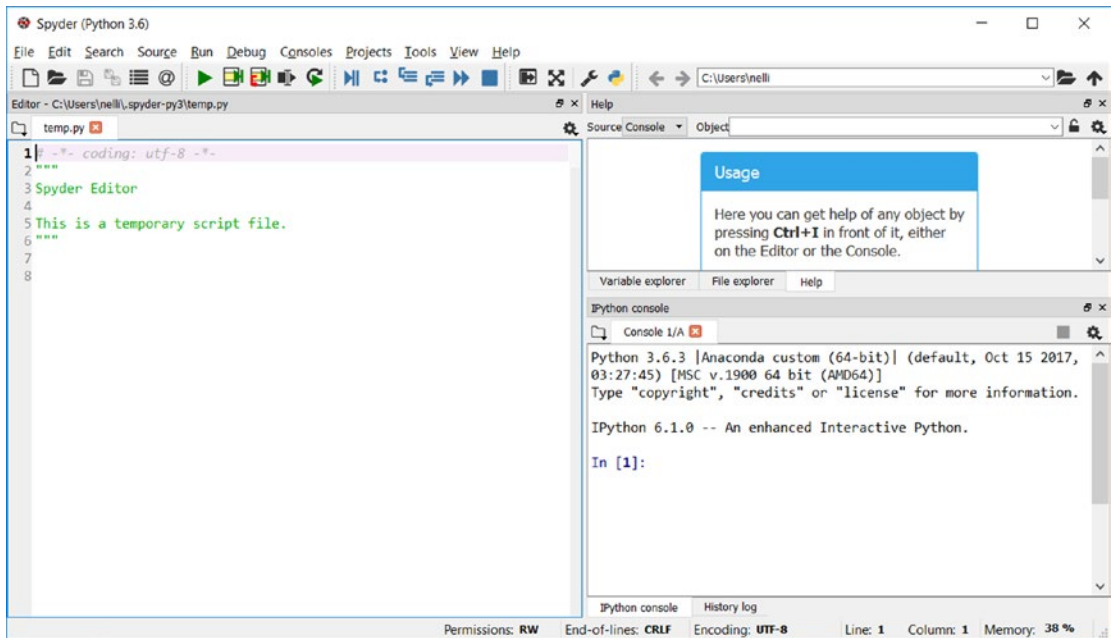


Figure 2-5. The Spyder IDE

Eclipse (pyDev)

Those of you who have 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 2-6). There is also an Eclipse version for developing in Python, thanks to the installation of an additional plugin called *pyDev*.

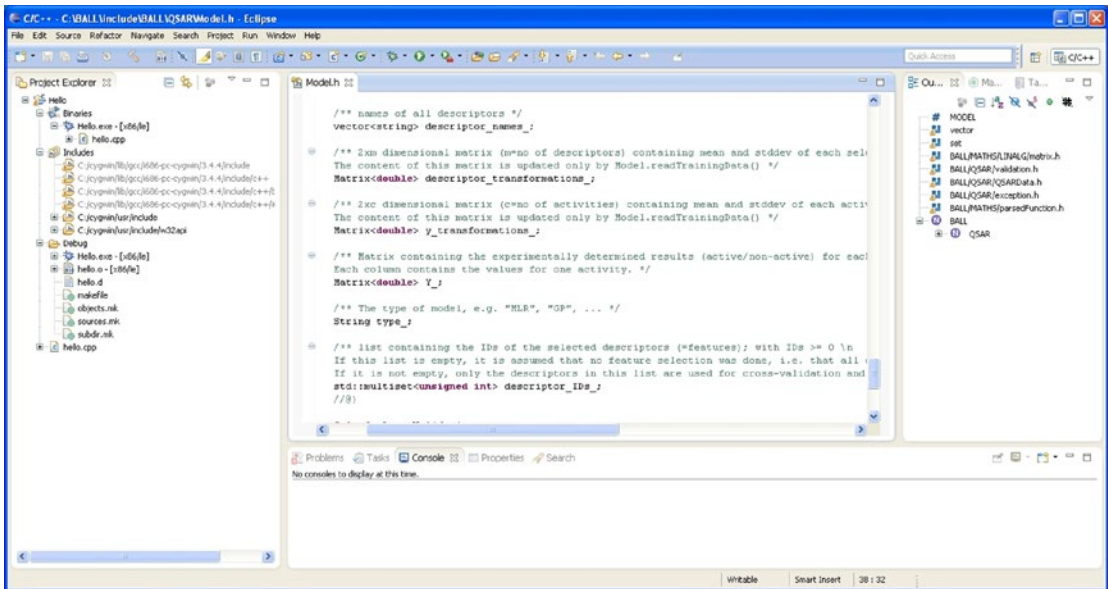


Figure 2-6. The Eclipse IDE

Sublime

This text editor is one of the preferred environments for Python programmers (see Figure 2-7). In fact, there are several plugins available for this application that make Python implementation easy and enjoyable.

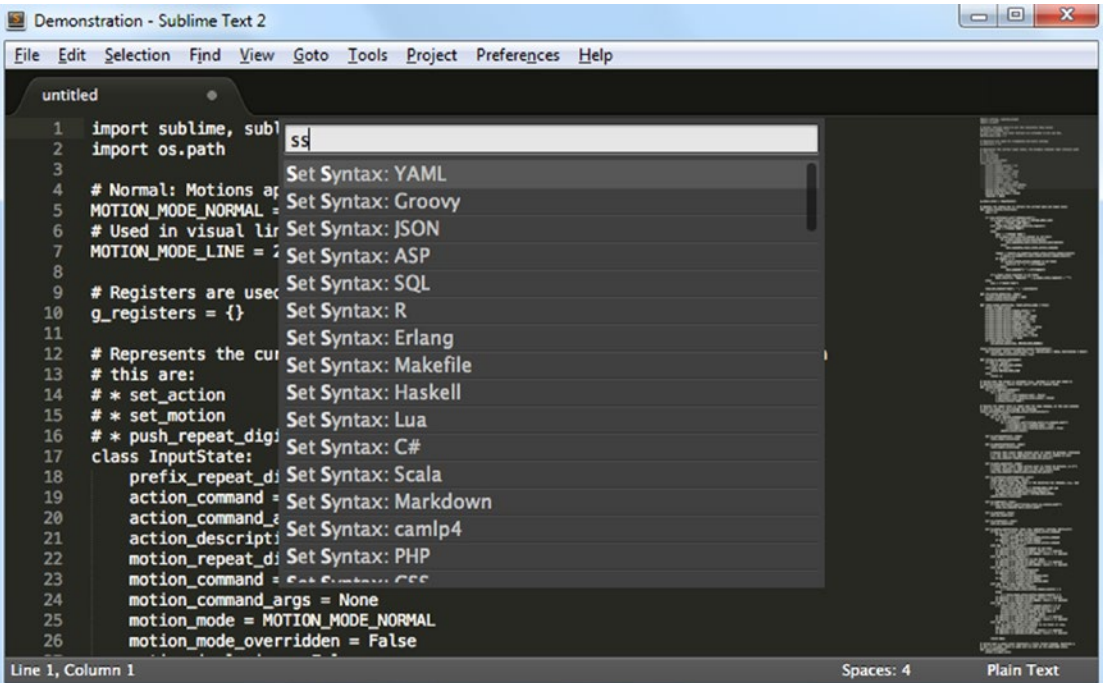


Figure 2-7. The Sublime IDE

Liclipse

Liclipse, similarly to Spyder, is a development environment specifically designed for the Python language (see Figure 2-8). It is very similar to the Eclipse IDE but it is fully adapted for a specific use in Python, without needing to install plugins like PyDev. So its installation and settings are much simpler than Eclipse.

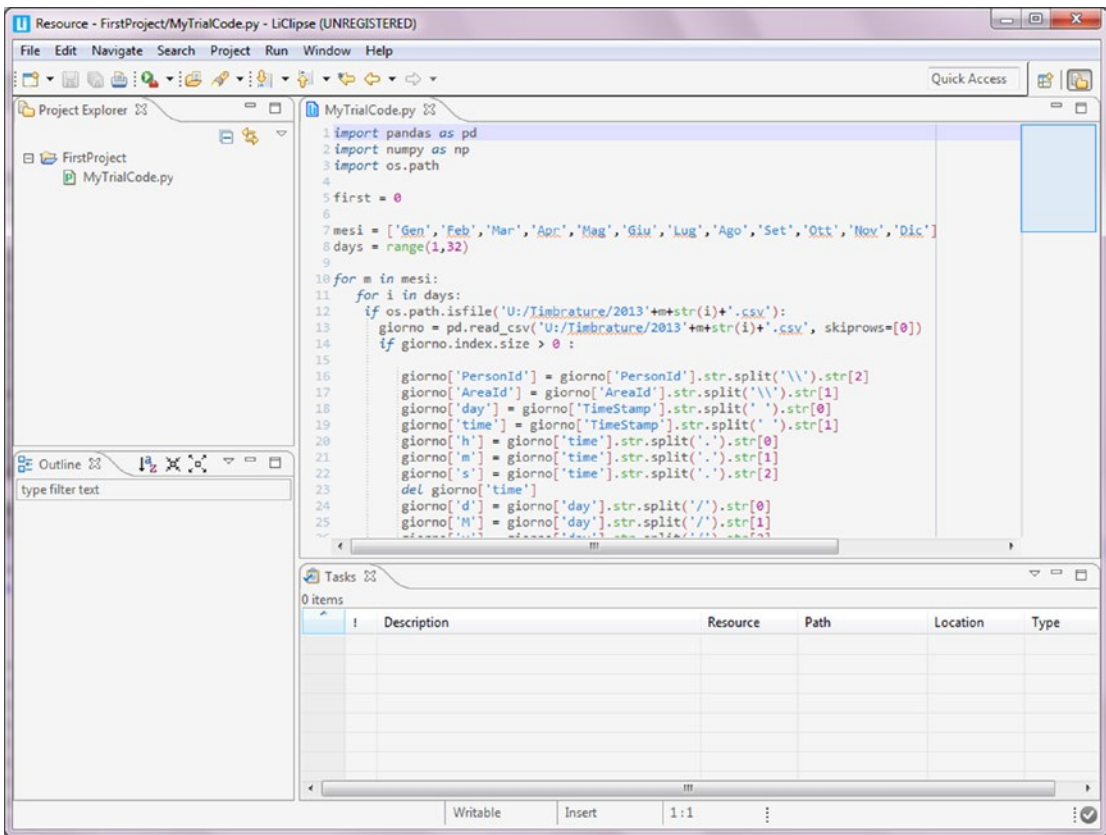


Figure 2-8. The Liclipse IDE

NinjaIDE

NinjaIDE (NinjaIDE is “Not Just Another IDE”), which characterized by a name that is a recursive acronym, is a specialized IDE for the Python language (see Figure 2-9). It’s a very recent application on which the efforts of many developers are focused. Already very promising, it is likely that in the coming years, this IDE will be a source of many surprises.

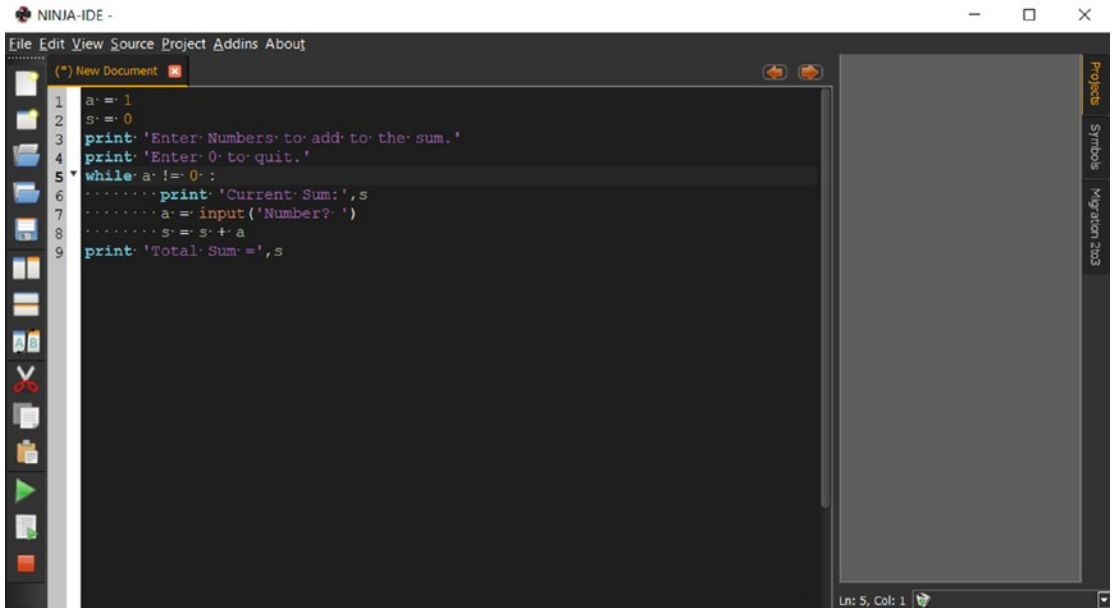


Figure 2-9. *The Ninja IDE*

Komodo IDE

Komodo is a very powerful IDE full of tools that make it a complete and professional development environment (see Figure 2-10). Paid software and written in C++, the Komodo development environment is adaptable to many programming languages, including Python.

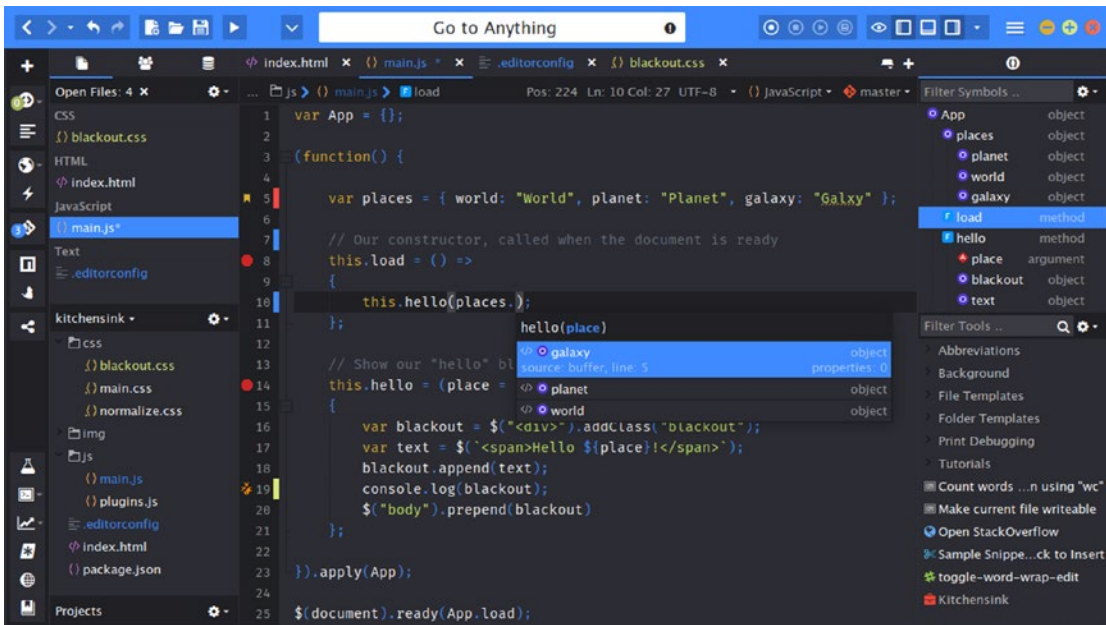


Figure 2-10. The Komodo IDE

SciPy

SciPy (pronounced “sigh pie”) is a set of open-source Python libraries specialized for scientific computing. Many of these libraries are the protagonists of many chapters of the book, given that their knowledge is critical to data analysis. Together they constitute a set of tools for calculating and displaying data. It has little to envy from 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 three in particular that are discussed in the following chapters:

- 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 the Python cannot provide. In fact, as you will see later in the book, 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 your computations. Throughout the book, this library is almost omnipresent because of its unique characteristics, so an entire chapter is devoted to it (Chapter 3).

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 datasets*: A set of tools for reading and writing data stored in the hard 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 of data analysis in Python. Therefore, the study and application of this package is the main goal on which you will work throughout the book (especially in Chapters 4, 5, and 6). Knowledge of its every detail, especially when it is applied to data analysis, is a fundamental objective of this book.

The fundamental concept of this package is the *DataFrame*, a two-dimensional tabular data structure with row and column labels.

Pandas applies the high-performance properties of the NumPy library to the manipulation of data in spreadsheets or in relational databases (SQL databases). In fact, by using sophisticated indexing, it will be easy to carry out many operations on this kind of data structure, 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. In Chapter 7, you learn about this rich library in detail so you will know how to represent the results of your analysis in the best way.

Conclusions

During the course of this chapter, all the fundamental aspects characterizing the Python world have been illustrated. The basic concepts of the Python programming language were introduced, with brief examples explaining its innovative aspects and how it stands out compared to other programming languages. In addition, different ways of using Python at various levels were presented. First you saw how to use a simple command-line interpreter, then a set of simple graphical user interfaces were shown until you got to complex development environments, known as IDEs, such as Spyder, Liclipse, and NinjaIDE.

Even the highly innovative project Jupyter (IPython) was presented, showing you how you can develop Python code interactively, in particular with the Jupyter Notebook.

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

In the next chapter, you deal with the first library that is the basis of all numerical calculations in Python: NumPy. You learn about the ndarray, a data structure which is the basis of the more complex data structures used in data analysis in the following chapters.