

Python for economists

Ewen Gallic

September 2019

Contents

List of Tables	9
List of Figures	11
Opening remarks	13
0.1 Objectives	13
0.2 Who are these notes for?	13
1 Introduction	15
1.1 Background information	15
1.2 Versions	17
1.3 Working space	17
1.3.1 Python in a terminal	17
1.3.2 IPython	19
1.3.3 Spyder	21
1.3.4 Jupyter Notebook	22
1.3.4.1 Evaluation of an instruction	24
1.3.4.2 Text cells	25
1.3.4.3 Deleting a cell	27
1.4 Variables	27
1.4.1 Assignment and deletion	27
1.4.2 Naming Conventions	29
1.5 Comments	30
1.6 Modules and packages	30
1.7 The Help System	32
2 Types of Data	35
2.1 Strings	35

2.1.1	Concatenation of Strings	37
2.1.2	Indexing and Extraction	39
2.1.3	Available Methods with Strings	41
2.1.3.1	Conversion to upper or lower case	41
2.1.3.2	Search Pattern for Strings	42
2.1.3.3	Splitting Strings	43
2.1.3.4	Cleaning, completion	44
2.1.3.5	Replacements	45
2.1.4	Conversion to character strings	46
2.1.5	Exercise	47
2.2	Numerical values	47
2.2.1	Integers	47
2.2.2	Floating Point Numbers	48
2.2.3	Complex numbers	49
2.2.4	Conversions	51
2.2.4.1	Conversion to Integer	51
2.2.4.2	Conversion to Floating Point Number	51
2.2.4.3	Conversion to Complex	52
2.3	Booleans	53
2.4	Empty Object	53
2.5	Dates and Times	54
2.5.1	Module Datetime	54
2.5.1.1	Date	55
2.5.1.2	Time	61
2.5.1.3	Datetime	62
2.5.1.4	Timedelta	65
2.5.2	pytz Module	67
2.5.3	Exercices	67
3	Structures	69
3.1	Lists	69
3.1.1	Extraction of the Elements	70
3.1.2	Modification	72
3.1.2.1	Replacement	72
3.1.2.2	Adding Elements	72
3.1.2.3	Deleting Elements	73
3.1.2.4	Multiple assignments	73
3.1.3	Verifying if a Value is Present	74

3.1.4	Copy of List	74
3.1.5	Sorting	76
3.2	Tuples	77
3.2.1	Extraction of the Elements	77
3.2.2	Modification	77
3.3	Sets	78
3.3.1	Modifications	79
3.3.1.1	Adding Elements	79
3.3.1.2	Deletion	80
3.3.2	Belonging test	81
3.3.3	Copying a Set	81
3.3.4	Conversion to a List	82
3.4	Dictionaries	82
3.4.1	Extraction of the Elements	83
3.4.2	Keys and values	85
3.4.3	Search for Belonging	86
3.4.4	Modification	86
3.4.4.1	Replacement	86
3.4.4.2	Adding Elements	87
3.4.4.3	Deleting elements	88
3.4.5	Copy of a Dictionary	89
3.4.6	Exercise	90
4	Operators	91
4.1	Arithmetic Operators	91
4.1.1	Addition	91
4.1.2	Subtraction	92
4.1.3	Multiplication	92
4.1.4	Division	92
4.1.5	Modulo	93
4.1.6	Power	93
4.1.7	Order	93
4.1.8	Mathematical Operators on Strings	94
4.1.9	Mathematical Operators on Lists or tuples	94
4.2	Comparison Operators	95
4.2.1	Equality, Inequality	96
4.2.2	Inferiority and Superiority, Strict or Broad	96
4.2.3	Inclusion and exclusion	99

4.3	Logical operators	100
4.3.1	And logical	100
4.3.2	Logical OR	101
4.3.3	Logical Not	102
4.4	Some Functions	103
4.5	Some Constants	104
4.6	Exercise	105
5	Loading and Saving Data	107
5.1	Load Data	108
5.1.1	Fichiers textes	108
5.1.1.1	Import from the Internet	111
5.1.2	CSV Files	112
5.1.2.1	Import From the Internet	113
5.1.3	JSON Files	114
5.1.3.1	Import from the Internet	114
5.1.4	Excel Files	115
5.2	Exporting data	115
5.2.1	Text Files	115
5.2.2	CSV Files	117
5.2.3	JSON Files	118
5.2.4	Exercise	120
6	Conditions	121
6.1	Conditional <code>if</code> Instructions	121
6.2	<code>if-else</code> Conditional Instructions	123
6.3	<code>if-elif</code> Conditional Instructions	124
6.4	Exercise	125
7	Loops	127
7.1	Loops with <code>while()</code>	127
7.2	Loops with <code>for()</code>	128
7.3	Exercise	132
8	Functions	135
8.1	Definition	135
8.1.1	Adding a Description	136
8.1.2	Parameters of a Function	136
8.1.2.1	Call Without Parameter Names	137

8.1.2.2	Positional Arguments, Arguments by Keywords . . .	137
8.1.2.3	Function as an Argument to Another Function . . .	140
8.2	Scope of a Function	140
8.3	Lambda Functions	143
8.4	Returning Several Values	144
8.5	Exercise	145
9	Introduction to Numpy	147
9.1	Arrays	147
9.1.1	Creation	148
9.1.1.1	Some Functions Generating <code>array</code> Objects	150
9.1.2	Dimensions	153
9.1.3	Extracting Elements from an Array	155
9.1.3.1	Extraction Using Boolean	158
9.1.4	Modification	159
9.1.4.1	Insterting Elements	160
9.1.4.2	Deleting / Removing Elements	161
9.1.5	Copyi of an Array	164
9.1.6	Sorting	166
9.1.7	Transposition	166
9.1.8	Operations on Arrays	168
9.1.8.1	+ and - Operators	168
9.1.8.2	* and / Operators	169
9.1.8.3	Power	170
9.1.8.4	Operations on Matrices	171
9.1.9	Logical Operators	172
9.1.10	Some Constants	173
9.1.11	Universal functions	174
9.1.12	Mathematical and Statistical Methods and Functions	176
9.2	Generation of Pseudo-random Numbers	179
9.3	Exercise	183
10	References	185

List of Tables

2.3	Formatting codes	59
4.1	Comparison operators	95
4.2	Some numerical functions	103
4.3	Some constants integrated in Python	105
5.1	Main Values for How to Open Files.	109
5.2	Parameters of the <code>reader()</code> Function	112
9.1	Logical Functions	172
9.2	Formatting Codes	173
9.3	Unary Universal Function	174
9.4	Binary Universal Functions	175
9.5	Mathematical and Statistical Methods	176
9.6	Statistical Functions	178
9.7	Some Functions for Pseudo-random Number Generation	180

List of Figures

- 1.1 Programming, Scripting, and Markup Languages. 16
- 1.2 Python in a terminal. 18
- 1.3 Anaconda’s home window. 19
- 1.4 IPython console. 20
- 1.5 Spyder. 22
- 1.6 Jupyter Notebook. 23
- 1.7 An Empty Notebook. 24
- 1.8 Evaluated Cell. 24
- 1.9 Text cell not evaluated. 26

Opening remarks

These notes were produced as part of an introductory course on Python for students in the Econometrics and Big Data course of [Aix-Marseille School of Economics / École d'Économie d'Aix-Marseille \(AMSE\)](#)

0.1 Objectives

The purpose of this book is to introduce you to the Python programming language, to be able to use it efficiently and independently. The reader can and is strongly encouraged to execute all the examples provided. Some chapters are closed with exercises to better assimilate the concepts covered as they are read.

Obviously, Python being a very vast language, these notes cannot and are not intended to be exhaustive of the use of this computer language.

0.2 Who are these notes for?

Initially, this book is intended for beginners who wish to learn the basics of Python. It is intended for AMSE students but may be of interest to individuals with an approach to data through the economic discipline wishing to discover Python.

I would like to thank Adrien Pacifico for his informative comments.

Chapter 1

Introduction

This document is mainly constructed using different references, including :

- books : Briggs (2013), Grus (2015), VanderPlas (2016), McKinney (2017) ;
- (excellents) notebooks : Navaro (2018).

1.1 Background information

Python is a multiplatform programming language, written in C, under a free license. It is an interpreted language, *i.e.*, it requires an interpreter to execute commands, and has no compilation phase. Its first public version dates from 1991. The main programmer, [Guido van Rossum](#), had started working on this programming language in the late 1980s. The name given to the Python language comes from the interest of its main creator in a British television series broadcast on the BBC called “*Monty Python’s Flying Circus*”.

The popularity of Python has grown strongly in recent years, as confirmed by the survey results provided since 2011 by [Stack Overflow](#). Stack Overflow offers its users the opportunity to complete a survey in which they are asked many questions to describe their experience as a developer. [The results of the 2019 survey](#) show a new breakthrough in the use of Python by developers. As shown in [Figure 1.1](#) 41.1% of respondents indicate that they develop in Python, *i.e.*, 2.3 percentage points higher than a year earlier.

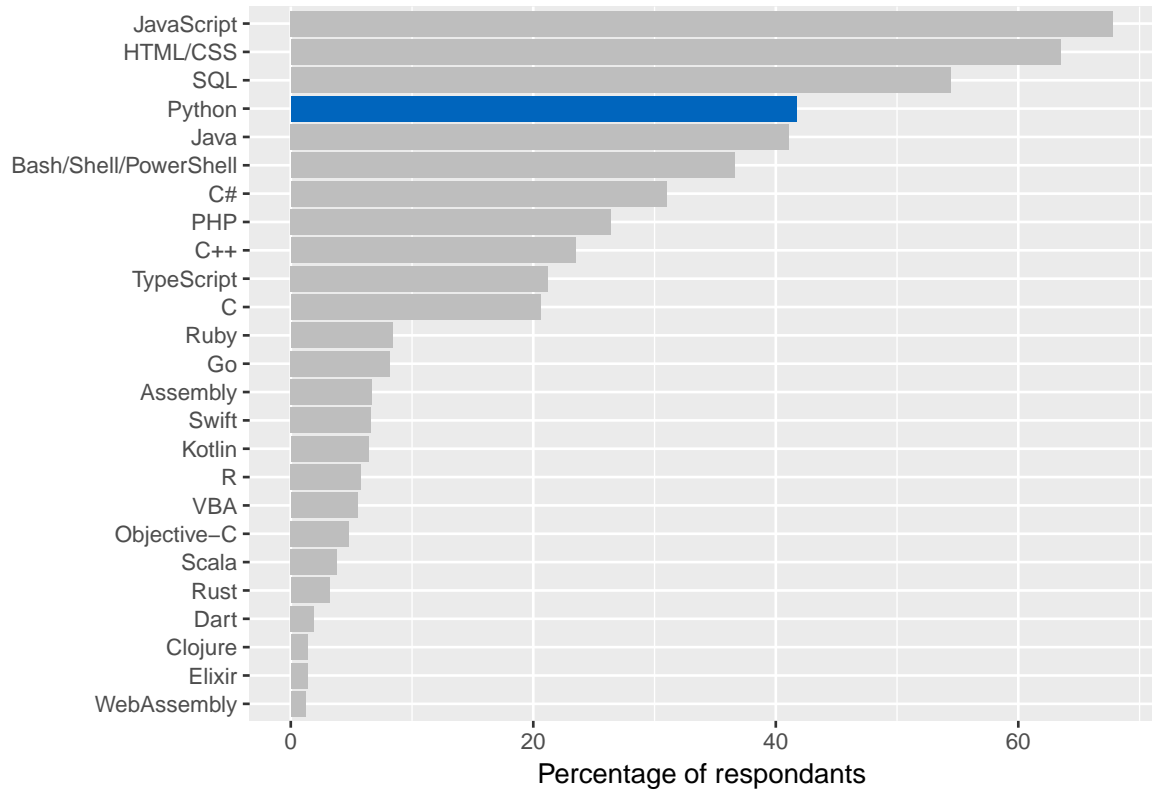


Figure 1.1: Programming, Scripting, and Markup Languages.

1.2 Versions

These course notes are intended to provide an introduction to Python, version 3.x. In this sense, the examples provided will correspond to this version, not to the previous ones.

Compared to version 2.7, version 3.0 has made significant changes. It should be noted that Python 2.7 will take “its retirement” on January 1, 2020. After this date, support will no longer be provided.

1.3 Working space

There are many environments in which to program in Python. We will briefly present some of them.

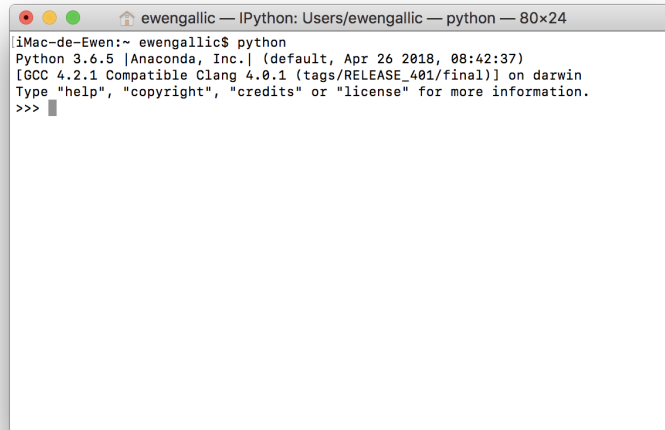
It is assumed here that you have installed [Anaconda] (<https://www.anaconda.com/>) on your computer. Anaconda is a free and open source distribution of the Python and R programming languages for *data science* and machine learning applications. In addition, when the terminal is mentioned in the notes, it is assumed that the operating system of your machine is either Linux or Mac OS.

1.3.1 Python in a terminal

It is possible to call Python from a terminal, by executing the following command (under Windows: in the start menu, launch the “Python 3.6” software):

```
python
```

What can be seen on screen is reproduced in Figure 1.2 :

A screenshot of a terminal window on a Mac. The window title is "ewengallic — IPython: Users/ewengallic — python — 80x24". The terminal content shows the command "python" being executed, followed by the Python 3.6.5 startup banner: "Python 3.6.5 [Anaconda, Inc.] (default, Apr 26 2018, 08:42:37) [GCC 4.2.1 Compatible Clang 4.0.1 (tags/RELEASE_401/final)] on darwin. Type "help", "copyright", "credits" or "license" for more information." The prompt ">>>" is visible at the end of the output.

```
iMac-de-Ewen:~ ewengallic$ python
Python 3.6.5 [Anaconda, Inc.] (default, Apr 26 2018, 08:42:37)
[GCC 4.2.1 Compatible Clang 4.0.1 (tags/RELEASE_401/final)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Figure 1.2: Python in a terminal.

We note the presence of the characters `>>>>` (*prompt*), which invite the user to enter a command. Expressions are evaluated once they are submitted (using the ‘ENTER’ key) and the result is given, when there is no error in the code.

The presence of the characters `>>>` (*prompt*), which invite the user to enter a command can be noticed. Expressions are evaluated once they are submitted (using the ‘ENTER’ key) and the result is given, when there is no error in the code.

For example, when evaluating `2+1`:

```
>>> 2+1
3
>>>
```

The *prompt* at the end can be noted: this tells the user that Python is ready to receive new instructions.

1.3.2 IPython

There is a slightly more friendly environment than Python in the terminal: IPython. It is also an interactive terminal, but with many more features, including syntax highlighting or auto-completion (using the tab key).

IPython can be opened using a terminal, using the following instruction:

```
ipython
```

IPython can also be launched from Anaconda's home window, by clicking on the Launch button of the `qtconsole` application, visible in the Figure 1.3.

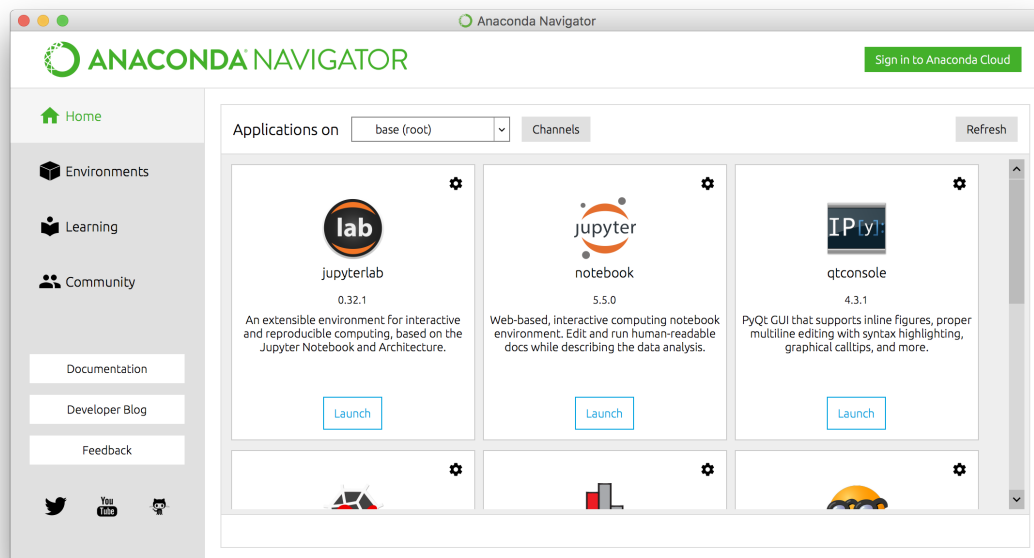


Figure 1.3: Anaconda's home window.

The IPython console, once launched, looks as follows:

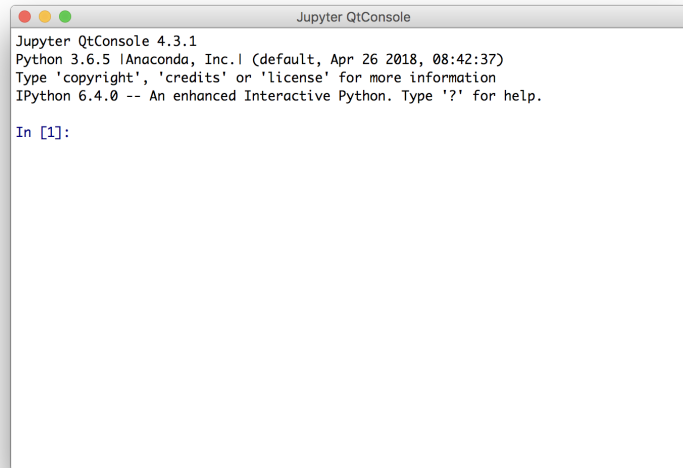


Figure 1.4: IPython console.

Let's submit a simple instruction:

```
print("Hello World")
```

The results shows:

```
In [1]: print("Hello World")  
Hello World
```

```
In [2]:
```

Several things should be noted. First, we note that at the end of the execution of the instruction, IPython indicates that it is ready to receive new instructions, by the presence of the *prompt* `In[2]:`. The number in brackets refers to the instruction number. We note that it went from 1 to 2 after the execution. We also note that the result of the call to the `print()` function, with the string of characters (delimited by quotation marks), displays on the screen what was contained between the parentheses.

1.3.3 Spyder

While when using Python in a terminal, it is recommended to have a text editor open next to it (to be able to save instructions), such as, for example, [Sublime Text](#) for Linux or Mac OS users, or [notepad+++](#) for Windows.

Another alternative is to use a single integrated development environment (IDE) that includes both an editor and a console. This is what [Spyder](#) offers, with many additional features, such as project management, file explorer, command log, debugger, etc.

To launch Spyder, one can open a terminal and simply evaluate `Spyder` (it is also possible to launch the software using the Start Menu for Windows users). Spyder can also be launched via [Anaconda](#).

The development environment, as shown in [Figure 1.5](#), is divided into several windows:

- on the left: the script editor;
- at the top right: a window to display Python help, the system tree or the variables created;
- bottom right: one or more consoles.

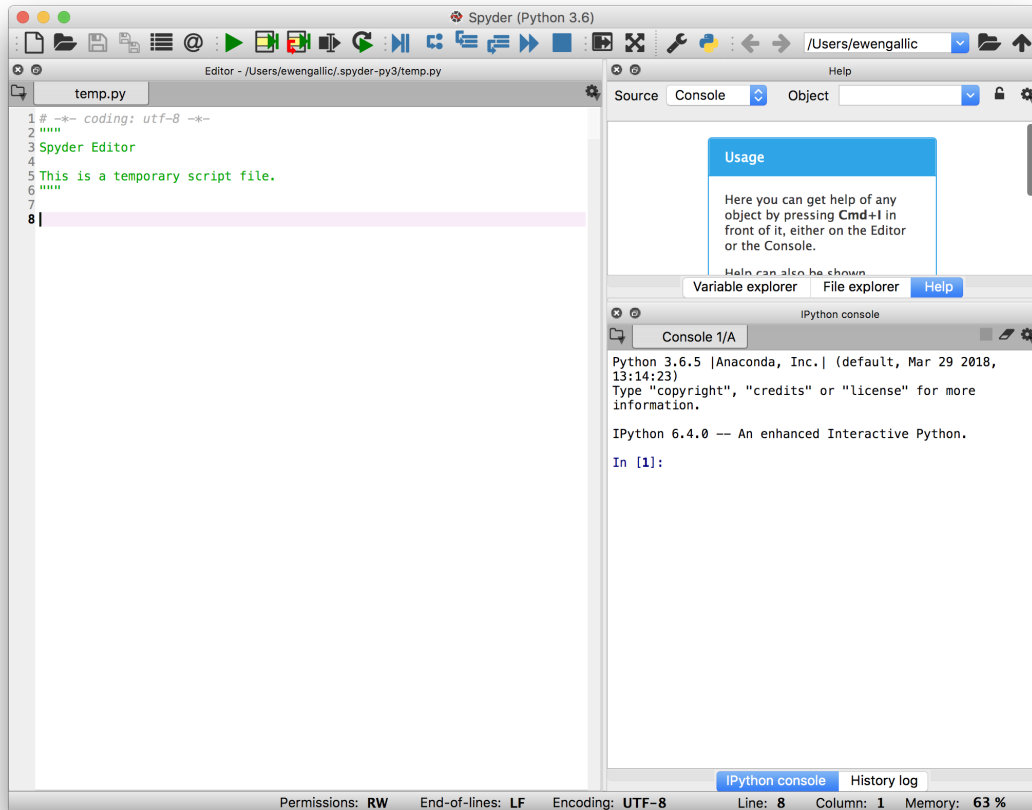


Figure 1.5: Spyder.

1.3.4 Jupyter Notebook

A graphical user interface in a web browser for IPython has gained a strong popularity in the recent years: [Jupyter Notebook](#). It is an open-source application for creating and sharing documents that contain code, equations, graphical representations and text. It is possible to include and execute different language codes in Jupyter notebooks.

Jupyter Notebook can be launched through Anaconda. After clicking on the Launch button of Jupyter Notebook in Anaconda, the default web browser launches and offers a tree structure, as depicted in Figure 1.6. Without realizing it, a local web

server was launched as well as a Python process (a *kernel*).

If the browser does not launch automatically, the page that should have been displayed can be accessed at the following address: <http://localhost:8890/tree?>.

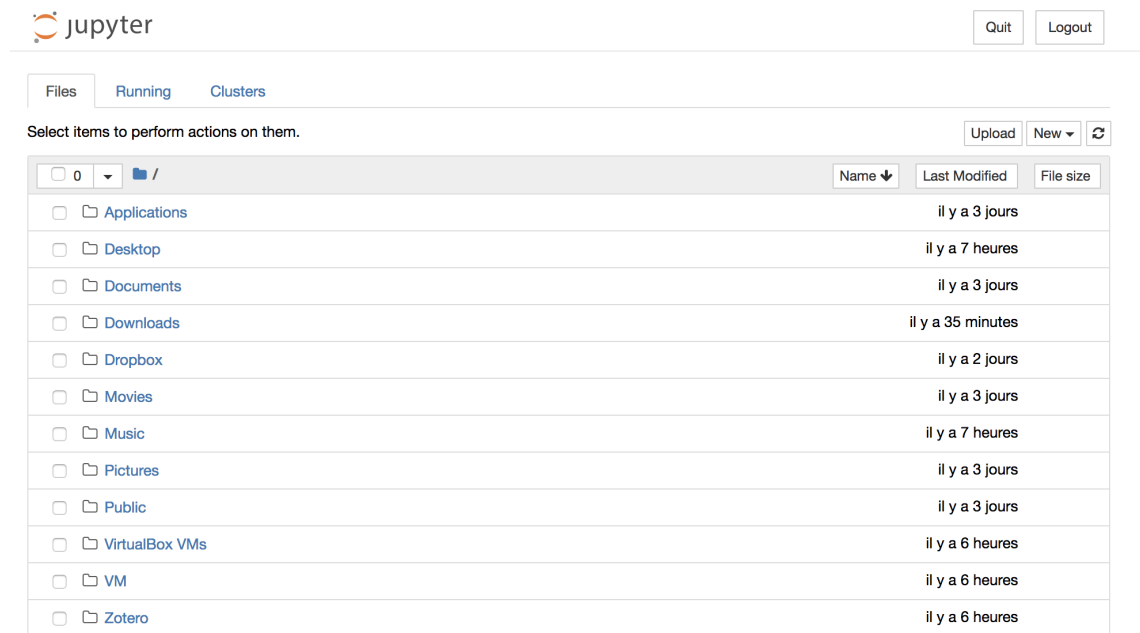


Figure 1.6: Jupyter Notebook.

To address the main functions of Jupyter, create a `jupyter` folder in a directory of our choice. Once this folder has been created, navigate through the Jupyter tree structure in the web browser.

Once in the folder, create a new `Python 3 Notebook` (by clicking on the `New` button at the top left of the window, then on `Python 3`).

A notebook named `Untitled` has just been created, the page displays an empty document, as shown in Figure 1.7.

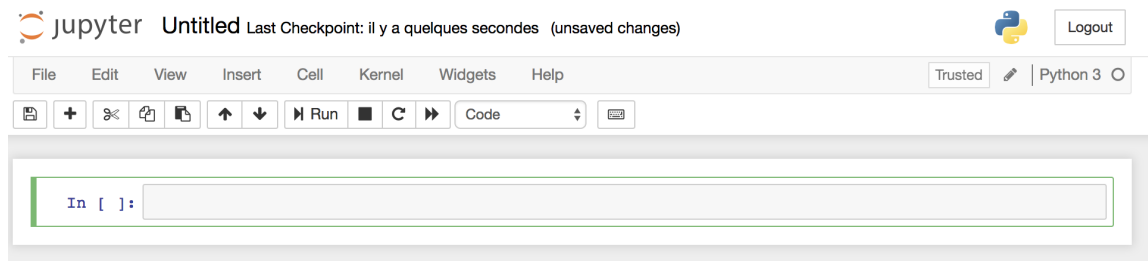


Figure 1.7: An Empty Notebook.

If we look in our file explorer, in the newly created `jupyter` folder, a new file has appeared: `Untitled.ipynb`.

1.3.4.1 Evaluation of an instruction

Let us go back to the web browser, to the page displaying your *notebook*.

Below the menu bar, we notice the presence of a framed area, **a cell**, that starts with `IN []:`, like what we saw in the console on IPython. On the right, the grey area invites us to submit instructions in Python.

Let us write the following instruction:

```
2+1
```

To submit the instruction for evaluation, there are several ways (make sure you have clicked inside the cell):

- in the menu bar: `Cell > Run Cells`;
- in the shortcut bar: button `Run` ;
- with the keyboard: hold down the `CTRL` key and press `Enter`.

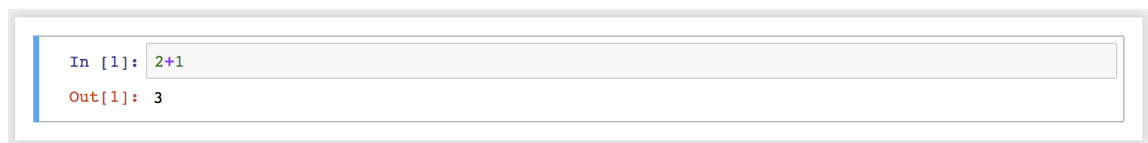


Figure 1.8: Evaluated Cell.

1.3.4.2 Text cells

Among the advantages of *notebooks* over traditional scripts is the possibility to add text boxes to accompany the codes and the corresponding output after evaluation.

Let's add a cell below the first one. To do this, one can proceed either:

- using the menu bar: **Insert > Insert Cell Below** (to insert a cell below; if you want an insertion above, just choose **Insert Cell Above**);
- by clicking in the frame of the cell from which you want to add (anywhere except in the grayed out code area, so that you can switch to **command' mode**), then pressing the **B** key on the keyboard (**A'** for insertion above).

The new cell calls for a Python instruction to be entered. To indicate that the content should be interpreted as text, it is necessary to specify it. Again, there are several ways to do this:

- using the menu bar: **Cell > Cell Type > Markdown**;
- using the shortcut bar: in the drop-down menu where **Code** is written, by selecting **Markdown**;
- in command mode (after clicking inside the cell frame, but not in the code area), by pressing the **M** key on the keyboard.

The cell is then ready to receive text, written in markdown. For more information on writing in Markdown, you can refer to this [cheat sheet] (<https://github.com/adam-p/markdown-here/wiki/Markdown-Cheatsheet>).

Let's enter a few lines of text to see very briefly how the cells written in Markdown work.

```
# A Level 1 Title
```

```
I will write *text in italics* and also **in bold**.
```

```
## A Level 2 Title
```

```
I can write lists:
```

```
- with an item  
- a second one  
- and a third nesting a new list:  
  - with a subitem  
  - and a second one
```

- a fourth one including a numbered nested list:
 1. with a subitem
 1. and another.

```
## Another Level 2 Title
```

I can even put equations in \LaTeX .
Like $X \sim \mathcal{N}(0.1)$.

To learn more about \LaTeX , we can refer to this :
[Wikipedia page](https://en.wikibooks.org/wiki/LaTeX/Mathematics).

Which gives, in Jupyter:

```
In [1]: 1+1
Out[1]: 2
```

```

# A Level 1 Title

I will write text in italics and also in bold.

## A Level 2 Title

I can write lists:

- with an item
- a second one
- and a third nesting a new list:
  - with a subitem
  - and a second one
- a fourth one including a numbered nested list:
  1. with a subitem
  1. and another.

## Another Level 2 Title

I can even put equations in  $\LaTeX$ .
Like  $X \sim \mathcal{N}(0.1)$ .

To learn more about  $\LaTeX$ , we can refer to this :
\[Wikipedia page\]\(https://en.wikibooks.org/wiki/LaTeX/Mathematics\).

```

Figure 1.9: Text cell not evaluated.

Then, the cell still has to be evaluated, as if it were a cell containing a Python instruction, to switch to a Markdown display (CTRL and ENTER).

To **edit the text** once we have switched to markdown, a simple double-click in the cell text box does the trick.

To **change the cell type so that it becomes code**:

- using the menu bar: `Cell > Cell Type > Code` ;
- using the shortcut bar: in the drop-down menu where `Code` is written, by selecting `Code` ;
- in command mode, press the key on the Y keyboard.

1.3.4.3 Deleting a cell

To delete a cell:

- using the menu bar: `Edit > Delete Cells` ;
- using the shortcut bar: scissor icon;
- in command mode, press the D keyboard key twice.

1.4 Variables

1.4.1 Assignment and deletion

When we evaluated the `2+1` instructions earlier, the result was displayed in the console, but it was not saved. In many cases, it is useful to keep the content of the result in an object, so that it can be reused later. To do this, *variables* are used. To create a variable, we use the equality sign (`=`), followed by what we want to save (text, a number, several numbers, etc.) and preceded by the name we will use to designate this variable.

For example, if we want to store the result of the calculation `2+1` in a variable that we will name `x`, we write:

```
x = 2+1
```

To display the value of our variable `x`, we can use the function `print()`:

```
print(x)
```

```
## 3
```

To change the value of the variable, a new assignment can be made:

```
x = 4
print(x)
```

```
## 4
```

It is also possible to give more than one name to the same content (a copy of `x` is made):

```
x = 4;
y = x;
print(y)
```

```
## 4
```

If the copy is modified, the original will not be affected:

```
y = 0
print(y)
```

```
## 0
```

```
print(x)
```

```
## 4
```

A variable can be **deletet** with the instruction `del`:

```
del y
```

The display of the content of `'y'` returns an error:

```
print(y)
```

```
## Error in py_call_impl(callable, dots$args, dots$keywords):
##   NameError: name 'y' is not defined
##
## Detailed traceback:
```

```
## File "<string>", line 1, in <module>
```

But we note that the variable `x` has not been deleted:

```
print(x)
```

```
## 4
```

1.4.2 Naming Conventions

The name of a variable can be composed of alphanumeric characters as well as the underscore (`_`) (there is no limit on the length of the name). It is forbidden to start the name of the variable with a number. It is also prohibited to include a space in the name of a variable.

To increase the readability of the variable names, several methods exist. We will adopt the following:

- all letters in lowercase;
- the separation of terms by an underscore (`_`).

For example, for a variable containing the value of a user's identifier: `id_user`.

It should be noted that the variable names are **case sensitive**:

```
x = "toto"
print(x)
```

```
## toto
```

```
print(X)
```

```
## Error in py_call_impl(callable, dots$args, dots$keywords):
##   NameError: name 'X' is not defined
```

```
##
```

```
## Detailed traceback:
```

```
## File "<string>", line 1, in <module>
```

1.5 Comments

There are several ways to add comments in python.

One way is to use the number sign (#) to make a **comment on a single line**. Everything that follows the number sign to the end of the line will not be evaluated by Python. On the other hand, what comes before the number sign will be.

```
# Un commentaire print("Bonjour")  
print("Hello") # Un autre commentaire
```

```
## Hello
```

The introduction of a **block of comments** (comments on several lines) is done by surrounding what is to be commented with a delimiter: three single or double quotation marks:

```
"""  
A comment that starts on a line  
and continues on to another  
and stops at the third"""
```

1.6 Modules and packages

Some basic functions in Python are loaded by default. Others require a **module** to be loaded. These modules are files that contain **definitions** as well as **instructions**.

Package are defined as a combination of modules that offer a set of functions.

Among the *packages* that will be used in these notes are:

- [NumPy](#), a fundamental *package* for scientific calculations
- [pandas](#), a *package* allowing easy data manipulation and analysis
- [Matplotlib](#), a *package* allowing us to create graphics.

To load a module (or a *package*), we use the command `import`. For example, to load the *package* `pandas`:

```
import pandas
```

This allows us to use functions contained in the module or package. For example, here we can use the function `Series()`, contained in the *package* `pandas`, to create an array of data indexed to a dimension :

```
x = pandas.Series([1, 5, 4])
print(x)
```

```
## 0    1
## 1    5
## 2    4
## dtype: int64
```

It is possible to give an alias to the module or package that is imported, by specifying it using the following syntax:

```
import module as alias
```

This is common practice to shorten the names of modules that will be used a lot. For example, for `pandas`, the name is usually shortened to `pd`:

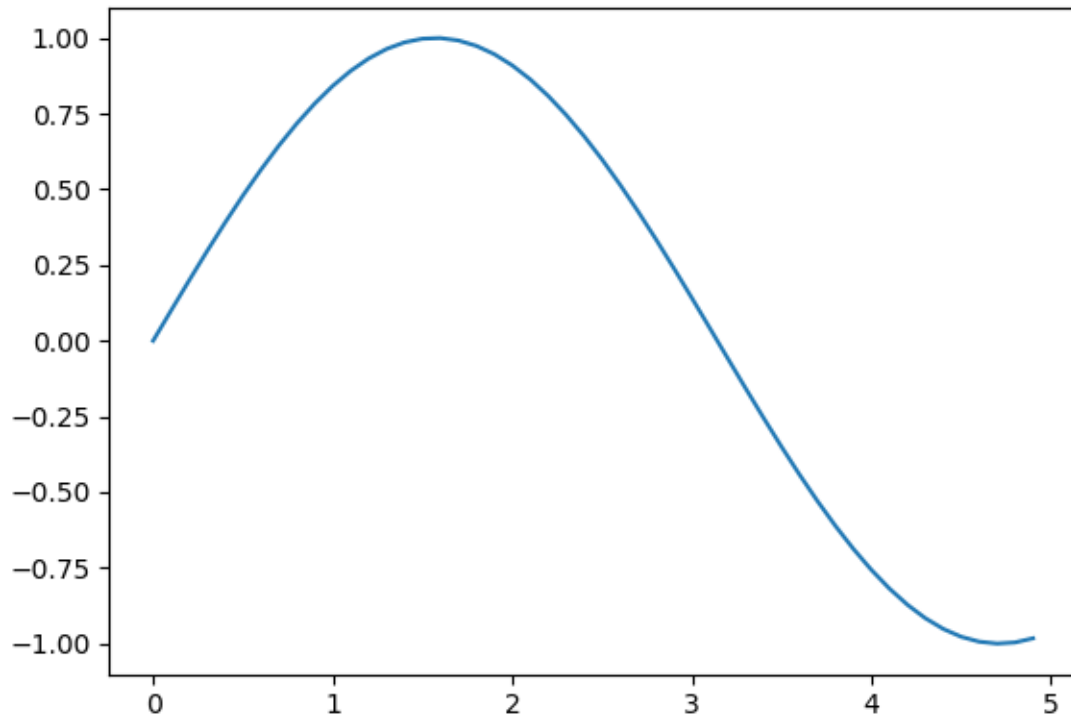
```
import pandas as pd
x = pd.Series([1, 5, 4])
print(x)
```

```
## 0    1
## 1    5
## 2    4
## dtype: int64
```

A single function can also be imported from a module, and an alias can be assigned to it (optionally). For example, with the `pyplot()` function of the package `matplotlib`, we usually do the following:

```
import matplotlib
import matplotlib.pyplot as plt
import numpy as np
x = np.arange(0, 5, 0.1);
```

```
y = np.sin(x)
plt.plot(x, y)
```



1.7 The Help System

To conclude this introduction, it seems important to mention the presence of **help** and **documentation** in Python.

For information on functions, it is possible to refer to the [online documentation] (<https://docs.python.org/3/>). It is also possible to get help inside the environment we are using, using the question mark (?).

For example, when using IPython (which, let's remember, is the case when working

with Jupyter Notebook), the help can be accessed using different syntaxes:

- `?` : provides an introduction and an overview of the features offered in Python (you leave it with the **ESC** key for example)
- `object?` : provides details about `object` (for example `x?` or `plt.plot?`)
- `object??` : more details about `object`
- `%quickref` : short reference on Python syntaxes
- `help()` : access to the Python help system.

Note: the **tabulation** key on the keyboard allows not only **autocompletion**, but also an **exploration of the content** of an object or module.

In addition, when it comes to finding help on a more complex problem, the right thing to do is not hesitate to search on a search engine, in mailing lists and of course on the many questions on [Stack Overflow](#).

Chapter 2

Types of Data

Many types of data are integrated into Python. In this section we will discuss strings, numerical values, booleans (`TRUE/FALSE`), the `null` value, dates and times.

2.1 Strings

A **string** is a collection of characters such as letters, numbers, spaces, punctuation marks, etc.

Strings are marked with single, double, or triple quotation marks.

Here is an example:

```
x = "Hello World"
```

To display the content of our variable `x` containing the string in the console, the function `print()` can be used:

```
print(x)
```

```
## Hello World
```

As indicated just before, single quotation marks can be used to create a string:

```
y = 'How are you?'  
print(y)
```

```
## How are you?
```

To include apostrophes in a character string created using single quotation marks, one must use an escape character: a backslash (\):

```
z = 'I\'m fine'  
print(z)
```

```
## I'm fine
```

Note that if the string is created using double quotation marks, it is not necessary to use the escape character:

```
z = "I'm \"fine\""  
print(z)
```

```
## I'm "fine"
```

To specify a line break, we use the following string: `\n`.

```
x = "Hello, \nWorld"  
print(x)
```

```
## Hello,  
## World
```

In the case of character strings on **multiple lines**, using single or double quotation marks will return an error (*EOL while scanning literal*, *i.e.*, detection of a syntax error, Python was expecting something else at the end of the line). To write a string on several lines, Python suggests using quotation marks (single or double) at the beginning and end of the string three times:

```
x = """Hello,
World"""
print(x)

## Hello ,
## World
```

Remarque 2.1.1

The character `\` (backslash) is the escape character. It allows to display certain characters, such as quotation marks in a string defined by quotation marks, or control characters, such as tabulation, line breaks, etc. Here are some common examples:

Code	Description	Code	Description
<code>\n</code>	New line	<code>\r</code>	Line break
<code>\t</code>	Tabulation	<code>\b</code>	Backspace
<code>\\</code>	Backslash	<code>\'</code>	Quotation mark
<code>\"</code>	Double quotation mark	<code>\`</code>	Grave accent

To obtain the **length of a string**, Python offers the function `len()`:

```
x = "Hello World !"
print(len(x))

## 13
```

```
print(x, len(x))

## Hello World ! 13
```

2.1.1 Concatenation of Strings

To concatenate strings, *i.e.*, to put them end to end, Python offers to use the operator `+`:

```
print("Hello" + " World")
```

```
## Hello World
```

The `*` operator allows us to repeat a string several times:

```
print( 3 * "Go Habs Go! " + "Woo Hoo!")
```

```
## Go Habs Go! Go Habs Go! Go Habs Go! Woo Hoo!
```

When two literals of strings are side by side, Python concatenates them:

```
x = ('You shall ' 'not ' "pass!")  
print(x)
```

```
## You shall not pass!
```

It is also possible to **add the content of a variable** to a string, using brackets (`{}`) and the method `format()`:

```
x = "I like to code in {}"  
langage_1 = "R"  
langage_2 = "Python"  
preference_1 = x.format(langage_1)  
print(preference_1)
```

```
## I like to code in R
```

```
preference_2 = x.format(langage_2)  
print(preference_2)
```

```
## I like to code in Python
```

It is possible to add **more than one variable content** in a string, always with brackets and the method `format()`:

```
x = "I like to code in {} and in {}"  
preference_3 = x.format(langage_1, langage_2)  
print(preference_3)  
  
## I like to code in R and in Python
```

2.1.2 Indexing and Extraction

Strings can be indexed. Be careful, **the index of the first character starts at 0**.

To obtain the *i*th character of a string, brackets can be used. The syntax is as follows:

```
x[i-1]
```

For example, to display the first character, then the fifth of the `Hello` string:

```
x = "Hello"  
print(x[0])
```

```
## H
```

```
print(x[4])
```

```
## o
```

The extraction can be done starting at the end of the chain, by preceding the value of the index with the minus sign (-).

For example, to display the penultimate character of our string `x`:

```
print(x[-2])
```

```
## l
```

The extraction of a substring by specifying its start and end position (implicitly or not) is also done with the brackets. We just need to specify the two index values: `[start:end]` as in the following example:

```
x = "You shall not pass!"

# From the fourth character (not included) to the ninth (included)
print(x[4:9])

## shall
```

When the first value is not specified, the beginning of the string is taken by default; when the second value is not specified, the end of the string is taken by default.

```
# From the 4th character (non included) to the end of the string
print(x[4:])
# From the beginning of the string to the penultimate (included)
print(x[:-1])
# From the 3rd character before the end (included) to the end
print(x[-5:])
```

```
## shall not pass!
```

```
## You shall not pass
```

```
## pass!
```

It is possible to add a third clue in the brackets: **the step**.

```
# From the 4th character (not included),
# to the end of the string, in steps of 3
print(x[4::3])
```

```
## sln s
```

To obtain the chain in the opposite direction:


```
print(x[::-1])

## !ssap ton llahs uoY
```

2.1.3 Available Methods with Strings

Many methods are available for strings. By adding a dot (.) after the name of an object designating a string and then pressing the tab key, the available methods are displayed in a drop-down menu.

For example, the `count()` method allows us to count the number of occurrences of a pattern in the string. To count the number of occurrences of `in` in the following string:

```
x = "le train de tes injures roule sur le rail de mon indifférence"
print(x.count("in"))

## 3
```

Remarque 2.1.2

Once the method call has been written, by placing the cursor at the end of the line and pressing the `Shift` and `Tabulation` keys, explanations can be displayed.

2.1.3.1 Conversion to upper or lower case

The `lower()` and `upper()` methods allow us to pass a string in lowercase and uppercase characters, respectively.

```
x = "le train de tes injures roule sur le rail de mon indifférence"
print(x.lower())
print(x.upper())
```

```
## le train de tes injures roule sur le rail de mon indiffé  
rence
```

```
## LE TRAIN DE TES INJURES ROULE SUR LE RAIL DE MON INDIFFÉ  
RENCE
```

2.1.3.2 Search Pattern for Strings

When we wish to **find a pattern** in a string, we can use the method `find()`. A pattern to be searched is provided in parameters. The `find()` method returns the smallest index in the string where the pattern is found. If the pattern is not found, the returned value is `-1`.

```
print(x.find("in"))  
print(x.find("hello"))
```

```
## 6
```

```
## -1
```

It is possible to add as an option an indication allowing to **restrict the search on a substring**, by specifying the start and end index :

```
print(x.find("in", 7, 20))
```

```
## 16
```

Note: the end index can be omitted; in this case, the end of the string is used:

```
print(x.find("in", 20))
```

```
## 49
```

Remarque 2.1.3

If one does not want to know the position of the sub-chain, but only its presence or absence, one can use the operator `in`: `print("train" in x)`

To perform a search **without regard to case**, the method `capitalize()` can be used:

```
x = "Mademoiselle Deray, il est interdit de manger de la choucroute ici."
print(x.find("deray"))
```

```
## -1
```

```
print(x.capitalize().find("deray"))
```

```
## 13
```

2.1.3.3 Splitting Strings

To **split a string into substrings**, based on a pattern used to delimit the substrings (*e.g.*, a comma or a space), the method `split()` can be used:

```
print(x.split(" "))
```

```
## ['Mademoiselle', 'Deray,', 'il', 'est', 'interdit', 'de', 'manger', 'de', 'la', 'choucroute', 'ici.']
```

By indicating a numerical value as parameters, it is possible to limit the number of substrings returned:

```
# Le nombre de sous-chaînes maximum sera de 3
print(x.split(" ", 3))
```

```
## ['Mademoiselle', 'Deray,', 'il', 'est interdit de manger de la choucroute ici.']
```

The `splitlines()` method also allows us to separate a string of characters according to a pattern, this pattern being an end of line character, such as a line break or a carriage return for example.

```
x = '''No, I am your Father!
- No... No. It's not true! That's impossible!
- Search your feelings. You know it to be true.
- Nooooooooo! Nooooo!'''
print(x.splitlines())

## ["No, I am your Father!", "- No... No. It's not true! That
   's impossible!", '- Search your feelings. You know it to be
   true.', '- Nooooooooo! Nooooo!"]
```

2.1.3.4 Cleaning, completion

To remove blank characters (*e.g.*, spaces, line breaks, quadratins, etc.) at the beginning and end of a string, we can use the `strip()` method, which is sometimes very useful for cleaning strings.

```
x = "\n\n  Pardon, du sucre ?  \n \n"
print(x.strip())

## Pardon, du sucre ?
```

It is possible to specify in parameter which characters to remove at the beginning and end of the string:

```
x = "www.egallic.fr"
print(x.strip("wrf."))

## egallic
```

Sometimes we have to make sure to obtain a **string of a given length** (when we have to provide a file with fixed widths for each column for example). The `rjust()` method is then a great help. By entering a string length and a fill character, it returns

the string with a possible completion (if the length of the returned string is not long enough with respect to the requested value), repeating the fill character as many times as necessary.

For example, to have a longitude coordinate stored in a string of characters of length 7, adding spaces may be necessary:

```
longitude = "48.11"
print(x.rjust(7, " "))
```

```
## www.egallic.fr
```

2.1.3.5 Replacements

The `replace()` method allows to perform **replacement of patterns** in a character string:

```
x = "Criquette ! Vous, ici ? Dans votre propre salle de bain ? Quelle surprise !"
print(x.replace("Criquette", "Ridge"))
```

```
## Ridge ! Vous, ici ? Dans votre propre salle de bain ?
   Quelle surprise !
```

This method is very convenient for **removing spaces** for example:

```
print(x.replace(" ", ""))
```

```
## Criquette!Vous,ici?Dansvotrepropresalledebain?
   Quellesurprise!
```

Here is a table listing some of the available methods ([exhaustive list in the documentation] (<https://docs.python.org/3/library/stdtypes.html#string-methods>)):

Méthode	Description
<code>capitalize()</code>	Capitalization of the first character and lowercase of the rest
<code>casefold()</code>	Removes case distinctions (useful for comparing strings without regard to case)

Méthode	Description
<code>count()</code>	Counts the number of occurrences (without overlap) of a pattern
<code>encode()</code>	Encodes a string of characters in a specific encoding
<code>find()</code>	Returns the smallest clue where a substring is found
<code>lower()</code>	Returns the string having passed each alphabetical character in lower case
<code>replace()</code>	Replaces one pattern with another
<code>split()</code>	Separates the chain into substring according to a pattern
<code>title()</code>	Returns the string after passing each first letter of a word through a capital letter
<code>upper()</code>	Returns the string having passed each alphabetical character in upper case

2.1.4 Conversion to character strings

When we want to concatenate a string with a number, Python returns an error.

```
nb_followers = 0
message = "He has " + nb_followers + "followers."

## Error in py_call_impl(callable, dots$args, dots$keywords):
##   TypeError: must be str, not int
##
## Detailed traceback:
##   File "<string>", line 1, in <module>

print(message)

## Error in py_call_impl(callable, dots$args, dots$keywords):
##   NameError: name 'message' is not defined
##
## Detailed traceback:
##   File "<string>", line 1, in <module>
```

We should then convert the object that is not a string into a string beforehand. To do this, Python offers the function `str()`:

```
message = "He has " + str(nb_followers) + " followers."
print(message)

## He has 0 followers.
```

2.1.5 Exercise

1. Create two variables named `a` and `b` so that they contain the following strings respectively: `23 to 0` and `C'est la piquette, Jack!`.
2. Display the number of characters from `a`, then `b`.
3. Concatenate `a` and `b` in a single string, adding a comma as a separating character.
4. Same question by choosing a separation that allows a line break between the two sentences.
5. Using the appropriate method, capitalize `a` and `b`.
6. Using the appropriate method, lowercase `a` and `b`.
7. Extract the word `la` and `Jack` from the string `b`, using indexes.
8. Look for the sub-chain `piqu` in `b`, then do the same with the sub-chain `mauvais`.
9. Return the position (index) of the first character `a` found in the string `b`, then try with the character `w`.
10. Replace the occurrences of the pattern `a` by the pattern `Z` in the substring `b`.
11. Separate the string `b` using the comma as a sub-chain separator.
12. (Bonus) Remove all punctuation characters from string `b`, then use an appropriate method to remove white characters at the beginning and end of the string. (Use the 'regex' library).

2.2 Numerical values

There are four categories of numbers in Python: integers, floating point numbers and complexes.

2.2.1 Integers

Integers (`ints`), in Python, are signed integers.

Remarque 2.2.1

The type of an object is accessed using the `type()` function in Python.

```
x = 2
y = -2
print(type(x))

## <class 'int'>
```

```
print(type(y))

## <class 'int'>
```

2.2.2 Floating Point Numbers

Floats are real numbers. They are written using a dot to distinguish the integer part from the decimal part of the number.

```
x = 2.0
y = 48.15162342
print(type(x))

## <class 'float'>
```

```
print(type(y))

## <class 'float'>
```

Scientific notations can also be used, using `E` or `e` to indicate a power of 10. For example, to write 3.2^{12} :

```
x = 3.2E12
y = 3.2e12
print(x)
```



```
## 3200000000000.0
```

```
print(y)
```

```
## 3200000000000.0
```

In addition, when the number is equal to a fraction of 1, it is possible to avoid writing the zero:

```
print(0.35)
```

```
## 0.35
```

```
print(.35)
```

```
## 0.35
```

2.2.3 Complex numbers

Python allows us to natively manipulate complex numbers, of the form $z = a + ib$, where a and b are floating point numbers, and such that $i^2 = (-i)^2 = -1$. The real part of the number, $\Re(z)$, is a while its imaginary part, $\Im(z)$, is b .

In python, the imaginary unit i is denoted by the letter `j`.

```
z = 1+3j  
print(z)
```

```
## (1+3j)
```

```
print(type(z))
```

```
## <class 'complex'>
```

It is also possible to use the `complex()` function, which requires two parameters (the real part and the imaginary part):

```
z = complex(1, 3)
print(z)
```

```
## (1+3j)
```

```
print(type(z))
```

```
## <class 'complex'>
```

Several methods are available with complex numbers. For example, to access the conjugate, Python provides the method `conjugate()`:

```
print(z.conjugate())
```

```
## (1-3j)
```

Access to the real part of a complex or its imaginary part is done using the `real()` and `imag()` methods, respectively.

```
z = complex(1, 3)
print(z.real())
```

```
## Error in py_call_impl(callable, dots$args, dots$keywords):
##   TypeError: 'float' object is not callable
##
## Detailed traceback:
##   File "<string>", line 1, in <module>
```

```
print(z.imag())
```

```
## Error in py_call_impl(callable, dots$args, dots$keywords):
##   TypeError: 'float' object is not callable
##
## Detailed traceback:
```

```
## File "<string>", line 1, in <module>
```

2.2.4 Conversions

To convert a number to another digital format, Python has a few functions.

2.2.4.1 Conversion to Integer

The **conversion of an integer or string** is done using the function `int()`:

```
x = "3"
x_int = int(x)
print(type(x_int))

## <class 'int'>
```

```
print(type(x))
```

```
## <class 'str'>
```

Note that the conversion of a floating point number truncates the number to keep only the integer part:

```
x = 3.6
x_int = int(x)
print(x_int)

## 3
```

2.2.4.2 Conversion to Floating Point Number

To **convert a number or string to a floating point number or string** (if possible), Python suggests using the function `float()`.

```
x = "3.6"
x_float = float(x)
print(type(x_float))

## <class 'float'>
```

With an integer:

```
x = 3
x_float = float(x)
print(x_float)

## 3.0
```

2.2.4.3 Conversion to Complex

The conversion of a number or a string of characters into a complex number is done with the function `complex()`:

```
x = "2"
x_complex = complex(x)
print(x_complex)

## (2+0j)
```

With a *float* :

```
x = 2.4
x_complex = complex(x)
print(x_complex)

## (2.4+0j)
```

2.3 Booleans

Logical data can have two values: `True` or `False`. They correspond to a logical condition. Care must be taken to ensure that the case is well respected.

```
x = True
y = False
print(x, y)
```

```
## True False
```

`True` can be automatically converted to 1; `False` to 0. This can be very convenient, for example, when counting true or false values in the columns of a data table.

```
res = True + True + False + True*True
print(res)
```

```
## 3
```

2.4 Empty Object

The empty object, commonly called `null`, has an equivalent in Python: `None`. To assign it to a variable, one should be careful with case:

```
x = None
print(x)
```

```
## None
```

```
print(type(x))
```

```
## <class 'NoneType'>
```

The `None` object is a neutral variable, with “null” behavior.

To test if an object is the `None` object, we proceed as follows (the result is a Boolean):

```
x = 1
y = None
print(x is None)
```

```
## False
```

```
print(y is None)
```

```
## True
```

2.5 Dates and Times

There are several modules to manage dates and time in Python. We will explore part of the `datetime` module.

2.5.1 Module `Datetime`

Python has a module called `datetime` which offers the possibility to manipulate dates and durations (*dates* and *times*).

There are several types of objects designating dates:

- `date`: a date according to the Gregorian calendar, indicating the year, month and day
- `time`: a given time, without taking into account a particular day, indicating the hour, minute, second (possibly the microsecond and time zone as well)
- `datetime`: a date combining `date` and `time`;
- `timedelta`: a time between two objects of the type `dates`, `time` or `datetime`;
- `tzinfo`: an abstract basic type, providing information about time zones;
- `timezone`: a type using the `tzinfo` type as a fixed offset from UTC.

2.5.1.1 Date

Objects of type `date` refer to dates in the Gregorian calendar, for which the following characteristics are mentioned: year, month and day.

To create a `date` object, the syntax is as follows:

```
date(year, month, day)
```

For example, to create the date of April 23, 2013:

```
from datetime import date
debut = date(year = 2013, month = 4, day = 23)
print(debut)
```

```
## 2013-04-23
```

```
print(type(debut))
```

```
## <class 'datetime.date'>
```

Remarque 2.5.1

It is not mandatory to specify the name of the parameters in the call to the `date` function. However, the order of priority should be as follows: year, month, day.

The attributes of the created date can then be accessed (they are integers):

```
print(debut.year) # Extract the year
```

```
## 2013
```

```
print(debut.month) # Extract the month
```

```
## 4
```

```
print(debut.day) # Extract the day
```

```
## 23
```

Some methods are available for objects of the type `date`. We will review some of them.

2.5.1.1.1 `ctime()`

The `ctime()` method returns the date as a string.

```
print(debut.ctime())
```

```
## Tue Apr 23 00:00:00 2013
```

2.5.1.1.2 `weekday()`

The `weekday()` method returns the position of the day of the week (Monday being 0, Sunday 6)

```
print(debut.weekday())
```

```
## 1
```

Remarque 2.5.2

This method can be very handy when analyzing data to explore aspects of weekly seasonality.

2.5.1.1.3 `isoweekday()`

In the same vein as `weekday()`, the `isoweekday()` method returns the position of the day of the week, this time assigning the value 1 to Monday and 7 to Sunday.


```
print(debut.isoweekday())
```

```
## 2
```

2.5.1.1.4 toordinal()

The `toordinal()` method returns the day number, taking as a reference the value 1 for the first day of year 1.

```
print(debut.toordinal())
```

```
## 734981
```

2.5.1.1.5 isoformat()

The `isoformat()` method returns the date in [ISO numbering](#), as a string.

```
print(debut.isoformat())
```

```
## 2013-04-23
```

2.5.1.1.6 isocalendar()

The `isocalendar()` method returns a nuplet (c.f. Section ??) with three elements: year, week number and day of week (all three in ISO numbering).

```
print(debut.isocalendar())
```

```
## (2013, 17, 2)
```

2.5.1.1.7 `replace()`

The `replace()` method returns the date after making a modification.

```
x = debut.replace(year=2014)
y = debut.replace(month=5)
z = debut.replace(day=24)
print(x, y, z)

## 2014-04-23 2013-05-23 2013-04-24
```

This has no impact on the original object:

```
print(debut)

## 2013-04-23
```

It is possible to modify several elements at the same time:

```
x = debut.replace(day=24, month=5)
print(x)

## 2013-05-24
```

2.5.1.1.8 `strftime()`

The `strftime()` method returns, as a string, a representation of the date, depending on a mask used.

For example, to have the date represented as DD-MM-YYYY (two-digit day, two-digit month and four-digit year):

```
print(debut.strftime("%d-%m-%Y"))

## 23-04-2013
```

In the previous example, two things are noteworthy: the presence of formatting instructions (which begin with the percentage symbol) and the presence of other characters (here, hyphens). It can be noted that characters can be replaced by others, this is a choice to represent the date by separating its elements with dashes. It is possible to adopt another type of writing, for example with slashes, or even other character strings:

```
print(debut.strftime("%d/%m/%Y"))
```

```
## 23/04/2013
```

```
print(debut.strftime("Jour : %d, Mois : %m, Annee : %Y"))
```

```
## Jour : 23, Mois : 04, Annee : 2013
```

As for the formatting guidelines, they correspond to the codes required by the C standard (c.f. the [Python documentation](#)). Here are some of them:

Table 2.3: Formatting codes

Code	Description	Example
%a	Abbreviation of the day of the week (depends on the location)	Tue
%A	Full weekday (depends on location)	Tuesday
%b	Abbreviation of the month (depends on the location)	Apr
%B	Name of the full month (depends on location)	April
%c	Date and time (depends on location) in format %a %e %b %H:%M:%S:%Y	Tue Apr 23 00:00:00 2013
%C	Century (00-99) (integer part of the year's division by 100)	20
%d	Day of the month (01-31)	23
%D	Date in format %m/%d/%y	04/23/13
%e	Day of the month in decimal number (1-31)	23
%F	Date in format %Y-%m-%d	2013-04-23
%h	Same as %b	Apr
%H	Hour (00-24)	00

Code	Description	Example
%I	Hour (01–12)	12
%j	Day of the year (001–366)	113
%m	Month (01–12)	04
%M	Minute (00–59)	00
%n	Line break in output, white character in input	\n
%p	AM/PM	AM
%r	Hour in format 12 AM/PM	12:00:00 AM
%R	Same as %H:%M	00:00
%S	Second (00–61)	00
%t	Tabulation in output, white character in input	\t
%T	Same as %H:%M:%S	00:00:00
%u	Day of the week (1–7), starts on Monday	2
%U	Week of the year (00–53), Sunday as the beginning of the week, and the first Sunday of the year defines the week	16
%V	Week of the year (00–53). If the week (which begins on a Monday) that contains January 1 has four or more days in the New Year, then it is considered Week 1. Otherwise, it is considered as the last of the previous year, and the following week is considered as week 1 (ISO 8601 standard)	17
%w	Day of the week (0–6), Sunday being equal to 0	2
%W	Week of the year (00–53), Monday being the first day of the week, and typically, the first Monday of the year defines week 1 (U.K. convention)	16
%x	Date (depends on location)	04/23/13
%X	Hour (depends on location)	00:00:00'
%y	Year without the “century” (00–99)	13
%Y	Year (in input, only from 0 to 9999)	2013
%z	Offset in hours and minutes with respect to UTC time	
%Z	Abbreviation of the time zone (output only) CEST	

2.5.1.2 Time

Time objects refer to specific times without taking into account a particular day. They provide information on the hour, minute, second (possibly the microsecond and time zone as well).

To create a `time` object, the syntax is as follows:

```
time(hour, minute, second)
```

For example, to create the moment 23:04:59 (twenty-three hours, four minutes and fifty-nine seconds):

```
from datetime import time
moment = time(hour = 23, minute = 4, second = 59)
print(moment)
```

```
## 23:04:59
```

```
print(type(moment))
```

```
## <class 'datetime.time'>
```

We can add information about the microsecond. Its value must be between zero and one million.

```
moment = time(hour = 23, minute = 4, second = 59, microsecond = 230)
print(moment)
```

```
## 23:04:59.000230
```

```
print(type(moment))
```

```
## <class 'datetime.time'>
```

The attributes of the created date (they are integers) can then be accessed, including the following:

```
print(moment.hour) # Extract the hour
```

```
## 23
```

```
print(moment.minute) # Extract the minute
```

```
## 4
```

```
print(moment.second) # Extract the second
```

```
## 59
```

```
print(moment.microsecond) # Extract the microsecond
```

```
## 230
```

Some methods for time objects are available. Their use is similar to objects of the `date` class (refer to Section [2.5.1.1](#)).

2.5.1.3 Datetime

The `datetime` objects combine the elements of the `date` and `time` objects. They provide the day in the Gregorian calendar as well as the hour, minute, second (possibly the microsecond and time zone).

To create a `datetime` object, the syntax is as follows:

```
datetime(year, month, day, hour, minute, second, microsecond)
```

For example, to create the date 23-04-2013 at 17:10:00:

```
from datetime import datetime
x = datetime(year = 2013, month = 4, day = 23,
             hour = 23, minute = 4, second = 59)
print(x)
```

```
## 2013-04-23 23:04:59
```

```
print(type(x))
```

```
## <class 'datetime.datetime'>
```

The `datetime` objects have the attributes of the `date` objects (c.f. Section [2.5.1.1](#)) and `time` type (c.f. Section [2.5.1.2](#)).

As for methods, relatively more are available. We will comment on some of them.

2.5.1.3.1 `today()` et `now()`

The `today()` and `now()` methods return the current `datetime`, the one at the time the instruction is evaluated:

```
print(x.today())
```

```
## 2019-09-20 19:14:30.914757
```

```
print(datetime.today())
```

```
## 2019-09-20 19:14:30.918129
```

The distinction between the two lies in the time zone. With `today()`, the attribute `tzinfo` is set to `None`, while with `now()`, the attribute `tzinfo`, if specified, is taken into account.

2.5.1.3.2 `timestamp()`

The `timestamp()` method returns, as a floating point number, the *timestamp* POSIX corresponding to the `datetime` object. The *timestamp* POSIX corresponds to the Posix time, equivalent to the number of seconds elapsed since January 1, 1970, at 00:00:00 UTC.

```
print(x.timestamp())
```

```
## 1366751099.0
```

2.5.1.3.3 date()

The `date()` method returns a `date` type object whose year, month and day attributes are identical to those of the object :

```
x_date = x.date()
print(x_date)
```

```
## 2013-04-23
```

```
print(type(x_date))
```

```
## <class 'datetime.date'>
```

2.5.1.3.4 time()

The `time()` method returns an object of type `time` whose hour, minute, second, microsecond attributes are identical to those of the object :

```
x_time = x.time()
print(x_time)
```

```
## 23:04:59
```

```
print(type(x_time))
```

```
## <class 'datetime.time'>
```


2.5.1.4 Timedelta

The objects of type `timedelta` represent times between two dates or times.

To create an object of type `timedelta`, the syntax is as follows:

```
timedelta(days, hours, minutes, seconds, microseconds)
```

It is not mandatory to provide a value for each parameter. When a parameter does not receive a value, its default value is 0.

For example, to create an object indicating a duration of 1 day and 30 seconds:

```
from datetime import timedelta
duree = timedelta(days = 1, seconds = 30)
duree
```

```
## datetime.timedelta(1, 30)
```

```
datetime.timedelta(1, 30)
```

The attributes (having been defined) can then be accessed. For example, to access the number of days represented by the duration:

```
duree.days
```

```
## 1
```

```
1
```

The `total_seconds()` method is used to obtain the duration expressed in seconds:

```
duree = timedelta(days = 1, seconds = 30, hours = 20)
duree.total_seconds()
158430.0
```

2.5.1.4.1 Time Between Two Objects `date` or `datetime`.

When subtracting two objects of type `date`, the number of days between these two dates is obtained, in the form of an object of type `timedelta`:

```

from datetime import timedelta
beginning = date(2018, 1, 1)
end = date(2018, 1, 2)
nb_days = end - beginning
print(type(nb_days))

## <class 'datetime.timedelta'>

```

```

print(nb_days)

## 1 day, 0:00:00

```

When subtracting two objects of type `datetime`, we obtain the number of days, seconds (and microseconds, if entered) separating these two dates, in the form of an object of type `timedelta`:

```

beginning = datetime(2018, 1, 1, 12, 26, 30, 230)
end = datetime(2018, 1, 2, 11, 14, 31)
duration = end - beginning
print(type(duration))

## <class 'datetime.timedelta'>

```

```

print(duration)

## 22:48:00.999770

```

It can be noted that the durations given take into account leap years. Let us first look at the number of days between February 28 and March 1 for a non-leap year:

```

beginning = date(2021, 2, 28)
end = date(2021, 3, 1)
duration = end - beginning
duration

```

```
datetime.timedelta(1)
```

Now let's look at the same thing, but in the case of a leap year:

```
beginning_leap = date(2020, 2, 28)
end_leap = date(2020, 3, 1)
beginning_leap = end_leap - beginning_leap
beginning_leap
```

```
datetime.timedelta(2)
```

It is also possible to **add durations to a date**:

```
debut = datetime(2018, 12, 31, 23, 59, 59)
print(debut + timedelta(seconds = 1))
```

```
## 2019-01-01 00:00:00
```

2.5.2 pytz Module

If date management is of particular importance, a library proposes to go a little further, especially with regard to time zone management. This library is called `pytz`. Many examples are available on [the project web page] (<https://pypi.org/project/pytz/>).

2.5.3 Exercices

1. Using the appropriate function, store the date of August 29, 2019 in an object called `d` then display the type of the object.
2. Using the appropriate function, display the current date.
3. Store the next date in an object named `d2` : "2019-08-29 20:30:56". Then, display in the console with the `print()` function the year, minute and second attributes of `d2`.
4. Add 2 days, 3 hours and 4 minutes to `d2`, and store the result in an object called `d3`.
5. Display the difference in seconds between `d3` and `d2`.

6. From the object `d2`, display the date of `d2` as a string so that it follows the following syntax: “Month Day, Year”, with “Month” the name of the month (August), “Day” the two-digit day number (29) and “Year” the year of the date (2019).

Chapter 3

Structures

Python features several different basic integrated structures. In this section we will discuss some of them: lists, tuples, sets and dictionaries.

3.1 Lists

One of the most flexible structures in Python is the list. It is a grouping of values. The creation of a list is done by writing the values by separating them with a comma and surrounding them by square brackets ([and]).

```
x = ["Pascaline", "Gauthier", "Xuan", "Jimmy"]
print(x)

## ['Pascaline', 'Gauthier', 'Xuan', 'Jimmy']
```

The content of a list is not necessarily text:

```
y = [1, 2, 3, 4, 5]
print(y)

## [1, 2, 3, 4, 5]
```

It is even possible to include elements of different types in a list:

```
z = ["Piketty", "Thomas", 1971]
print(z)

## ['Piketty', 'Thomas', 1971]
```

A list can contain another list:

```
tweets = ["aaa", "bbb"]
followers = ["Anne", "Bob", "Irma", "John"]
compte = [tweets, followers]
print(compte)

## [['aaa', 'bbb'], ['Anne', 'Bob', 'Irma', 'John']]
```

3.1.1 Extraction of the Elements

Access to the elements is made thanks to its indexation (be careful, the index of the first element is 0):

```
print(x[0]) # The first element of x

## Pascaline
```

```
print(x[1]) # The second element of x

## Gauthier
```

Access to an element can also be done by starting from the end, by putting the minus sign (-) in front of the index:

```
print(x[-1]) # The last element of x

## Jimmy
```

```
print(x[-2]) # The penultimate element of x
```

```
## Xuan
```

Splitting a list so as to obtain a subset of the list is done with the colon (:):

```
print(x[1:2]) # The first and second elements of x
```

```
## ['Gauthier']
```

```
print(x[2:]) # From the second element (not included) to the end of x
```

```
## ['Xuan', 'Jimmy']
```

```
print(x[:-2]) # From the first to the penultimate (not included)
```

```
## ['Pascaline', 'Gauthier']
```

Remarque 3.1.1

The extraction from a list using the brackets returns a list.

When extracting items from the list using the brackets, it is possible to add a third parameter, the step :

```
print(x[::2]) # Every other element
```

```
## ['Pascaline', 'Xuan']
```

Access to nested lists is done by using the brackets several times:

```
tweets = ["aaa", "bbb"]
followers = ["Anne", "Bob", "Irma", "John"]
conuts = [tweets, followers]
res = conuts[1][3] # The 4th item of the 2nd item on the list counts
```

The **number of elements in a list** is obtained with the function `len()` :

```
print(len(conuts))
```

```
## 2
```

```
print(len(conuts[1]))
```

```
## 4
```

3.1.2 Modification

Lists are mutable, *i.e.*, their content can be modified once the object has been created.

3.1.2.1 Replacement

To **modify** an element in a list, the indexes can be used:

```
x = [1, 3, 5, 6, 9]
x[3] = 7 # Replacing the 4th element
print(x)
```

```
## [1, 3, 5, 7, 9]
```

3.1.2.2 Adding Elements

To **add items to a list**, the method `append()` can be used:

```
x.append(11) # Add value 11 at the end of the list
print(x)
```

```
## [1, 3, 5, 7, 9, 11]
```


It is also possible to use the `extend()` method, to concatenate lists:

```
y = [13, 15]
x.extend(y)
print(x)

## [1, 3, 5, 7, 9, 11, 13, 15]
```

3.1.2.3 Deleting Elements

To removing an item from a list, the method `remove()` can be used:

```
x.remove(3) # Remove the fourth element
print(x)

## [1, 5, 7, 9, 11, 13, 15]
```

The `del` command can also be used:

```
x = [1, 3, 5, 6, 9]
del x[3] # Remove the fourth element
print(x)

## [1, 3, 5, 9]
```

3.1.2.4 Multiple assignments

Several values can be modified at the same time:

```
x = [1, 3, 5, 6, 10]
x[3:5] = [7, 9] # Replaces 4th and 5th values
print(x)

## [1, 3, 5, 7, 9]
```

The modification can increase the size of the list:

```
x = [1, 2, 3, 4, 5]
x[2:3] = ['a', 'b', 'c', 'd'] # Replaces the 3rd value
print(x)

## [1, 2, 'a', 'b', 'c', 'd', 4, 5]
```

Several values can be deleted at the same time:

```
x = [1, 2, 3, 4, 5]
x[3:5] = [] # Removes the 4th and 5th values
print(x)

## [1, 2, 3]
```

3.1.3 Verifying if a Value is Present

By using the operator `in`, it is possible to test the belonging of an object to a list:

```
x = [1, 2, 3, 4, 5]
print(1 in x)

## True
```

3.1.4 Copy of List

Be careful, copying a list is not trivial in Python. Let's take an example.

```
x = [1, 2, 3]
y = x
```

Let's modify the first element of `y`, and look at the content of `y` and `x`:

```
y[0] = 0
print(y)

## [0, 2, 3]
```

```
print(x)

## [0, 2, 3]
```

As can be seen, using the equal sign simply created a reference and not a copy.

To copy a list, there are several ways to do so. Among them, the use of the `list()` function:

```
x = [1, 2, 3]
y = list(x)
y[0] = 0
print("x : ", x)

## x : [1, 2, 3]
```

```
print("y : ", y)

## y : [0, 2, 3]
```

It can be noted that when a splitting is done, a new object is created, not a reference:

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

## x : [1, 2, 3, 4]
```

```
print("y : ", y)
## y : [0, 2]
```

3.1.5 Sorting

To sort the objects in the list (without creating a new one), Python offers the method `sort()` :

```
x = [2, 1, 4, 3]
x.sort()
print(x)
## [1, 2, 3, 4]
```

It also works with text values, sorting in alphabetical order:

```
x = ["c", "b", "a", "a"]
x.sort()
print(x)
## ['a', 'a', 'b', 'c']
```

It is possible to provide the `sort()` method with parameters. Among these parameters, there is one, `key`, which provides a function for sorting. This function must return a value for each object in the list, on which the sorting will be performed. For example, with the `len()` function, which, when applied to text, returns the number of characters:

```
x = ["aa", "a", "aaaa", "aa"]
x.sort(key=len)
print(x)
## ['a', 'aa', 'aa', 'aaaa']
```

3.2 Tuples

The *tuples* are sequences of Python objects.

To create a tuple, one lists the values, separated by commas:

```
x = 1, 4, 9, 16, 25
print(x)
```

```
## (1, 4, 9, 16, 25)
```

It should be noted that tuples are identified by a series of values, surrounded in two brackets.

3.2.1 Extraction of the Elements

The elements of a tuple are extracted in the same way as those in the lists (see Section 3.1.1).

```
print(x[0])
```

```
## 1
```

3.2.2 Modification

Unlike lists, tuples are **inalterable** (i.e. cannot be modified after they have been created):

```
x[0] = 1
```

```
## Error in py_call_impl(callable, dots$args, dots$keywords):
##   TypeError: 'tuple' object does not support item assignment
##
```

```
## Detailed traceback:
```

```
##   File "<string>", line 1, in <module>
```

It is possible to **nest tuples** inside another tuple. To do this, parentheses are used:

```
x = ((1, 4, 9, 16), (1, 8, 26, 64))
print(x)

## ((1, 4, 9, 16), (1, 8, 26, 64))
```

3.3 Sets

Sets are unordered collections of unique elements. The sets are unalterable, not indexed.

To create a set, Python provides the `set()` function. One or more elements constituting the set are provided, separated by commas and surrounded by braces (`{}`):

```
new_set = set({"Marseille", "Aix-en-Provence",
              "Nice", "Rennes"})
print(new_set)

## {'Nice', 'Rennes', 'Aix-en-Provence', 'Marseille'}
```

Equivalently, rather than using the `set()` function, the set can only be defined using the brackets:

```
new_set = {"Marseille", "Aix-en-Provence", "Nice", "Rennes"}
print(new_set)

## {'Nice', 'Rennes', 'Aix-en-Provence', 'Marseille'}
```

On the other hand, if the set is empty, Python returns an error if the `set()` function is not used: il est nécessaire d'utiliser la fonction `set` :

```
empty_set = {}
type(empty_set)
```

```
## <class 'dict'>
```

The type of the object we have just created is not `set` but `dict` (i.e. Section 3.4). Also, to create the empty set, we use `set()`:

```
empty_set = set()
print(type(empty_set))
```

```
## <class 'set'>
```

During the creation of a set, if there are duplicates in the values provided, these will be deleted to keep only one value:

```
new_set = set({"Marseille", "Aix-en-Provence", "Nice", "Marseille", "Rennes"})
print(new_set)
```

```
## {'Nice', 'Rennes', 'Aix-en-Provence', 'Marseille'}
```

The length of a set is obtained using the `len()` function:

```
print(len(new_set))
```

```
## 4
```

3.3.1 Modifications

3.3.1.1 Adding Elements

To add an element to a set, Python offers the `add()` method:

```
new_setv.add("Toulon")
```

```
## Error in py_call_impl(callable, dots$args, dots$keywords):
##   NameError: name 'new_setv' is not defined
##
```

```
## Detailed traceback:  
##   File "<string>", line 1, in <module>
```

```
print(new_set)
```

```
## {'Nice', 'Rennes', 'Aix-en-Provence', 'Marseille'}
```

If the element is already present, it will not be added:

```
new_set.add("Toulon")  
print(new_set)
```

```
## {'Nice', 'Aix-en-Provence', 'Toulon', 'Rennes', 'Marseille'  
   '}'
```

3.3.1.2 Deletion

To remove a value from a set, Python offers the method `remove()`:

```
new_set.remove("Toulon")  
print(new_set)
```

```
## {'Nice', 'Aix-en-Provence', 'Rennes', 'Marseille'}
```

If the value is not present in the set, Python returns an error message:

```
new_set.remove("Toulon")
```

```
## Error in py_call_impl(callable, dots$args, dots$keywords):  
##   KeyError: 'Toulon'  
##  
## Detailed traceback:  
##   File "<string>", line 1, in <module>
```



```
print(new_set)

## {'Nice', 'Aix-en-Provence', 'Rennes', 'Marseille'}
```

3.3.2 Belonging test

One of the advantages of sets is the quick search for presence or absence of values (faster than in a list). As with the lists, the belonging tests are performed using the operator `in`:

```
print("Marseille" in new_set)
```

```
## True
```

```
print("Paris" in new_set)
```

```
## False
```

3.3.3 Copying a Set

To copy a set, as for lists (c.f. Section ??), the equality sign should not be used. Copying a set is done using the `copy()` method:

```
new_set = set({"Marseille", "Aix-en-Provence", "Nice"})
y = new_set.copy()
y.add("Toulon")
print("y : ", y)
```

```
## y : {'Toulon', 'Nice', 'Aix-en-Provence', 'Marseille'}
```

```
print("set : ", new_set)
```

```
## set : {'Nice', 'Aix-en-Provence', 'Marseille'}
```

3.3.4 Conversion to a List

One of the interests of sets is that they contain unique elements. Also, when you want to obtain the distinct elements of a list, it is possible to convert it into a set (with the `set()` function), then to convert the set into a list (with the `list()` function):

```
my_list = ["Marseille", "Aix-en-Provence", "Marseille", "Marseille"]
print(my_list)
```

```
## ['Marseille', 'Aix-en-Provence', 'Marseille', 'Marseille']
```

```
my_set = set(my_list)
print(my_set)
```

```
## {'Aix-en-Provence', 'Marseille'}
```

```
my_new_list = list(my_set)
print(my_new_list)
```

```
## ['Aix-en-Provence', 'Marseille']
```

3.4 Dictionaries

Python dictionaries are an implementation of key-value objects, the keys being indexed.

Keys are often text, values can be of different types and structures.

To create a dictionary, you can proceed by using braces (`{}`). As encountered in the Section 3.3, if we evaluate the following code, we get a dictionary :

```
empty_dict = {}  
print(type(empty_dict))
```

```
## <class 'dict'>
```

To create a dictionary with entries, the braces can be used. Each entry is separated by commas, and the key is distinguished from the associated value by two points (:):

```
my_dict = { "nom": "Kyrie",  
            "prenom": "John",  
            "naissance": 1992,  
            "equipés": ["Cleveland", "Boston"]}  
print(my_dict)
```

```
## {'nom': 'Kyrie', 'prenom': 'John', 'naissance': 1992, '  
    equipés': ['Cleveland', 'Boston']}
```

It is also possible to create a dictionary using the `dict()` function, by providing a sequence of key-values:

```
x = dict([("Julien-Yacine", "Data-scientist"),  
         ("Sonia", "Director")])  
print(x)
```

```
## {'Julien-Yacine': 'Data-scientist', 'Sonia': 'Director'}
```

3.4.1 Extraction of the Elements

Extraction from dictionaries is based on the same principle as for lists and tuples (see Section [@ref\(#structure-liste-extraction\)](#)). However, the extraction of an element from a dictionary is not based on its position in the dictionary, but by its key:

```
print(my_dict["prenom"])
```

```
## John
```

```
print(my_dict["equipes"])  
## ['Cleveland', 'Boston']
```

If the extraction is done by a key not present in the dictionary, an error will be returned:

```
print(my_dict["age"])  
## Error in py_call_impl(callable, dots$args, dots$keywords):  
##   KeyError: 'age'  
##  
## Detailed traceback:  
##   File "<string>", line 1, in <module>
```

You can test the presence of a key with the operator `in`:

```
print("prenom" in my_dict)  
## True
```

```
print("age" in my_dict)  
## False
```

The extraction of values can also be done using the `get()` method, which returns a `None` value if the key is not present:

```
print(my_dict.get("prenom"))  
## John
```

```
print(my_dict.get("age"))  
## None
```

3.4.2 Keys and values

Using the `key()` method, the keys of the dictionary can be accessed:

```
the_keys = my_dict.keys()
print(the_keys)

## dict_keys(['nom', 'prenom', 'naissance', 'equipes'])
```

```
print(type(the_keys))

## <class 'dict_keys'>
```

It is then possible to transform this key enumeration into a list:

```
the_keys_list = list(the_keys)
print(the_keys_list)

## ['nom', 'prenom', 'naissance', 'equipes']
```

The `values()` method provides the dictionary values:

```
the_values = my_dict.values()
print(the_values)

## dict_values(['Kyrie', 'John', 1992, ['Cleveland', 'Boston
  ']])
```

```
print(type(the_values))

## <class 'dict_values'>
```

The `items()` method provides keys and values in the form of tuples:

```
the_items = my_dict.items()
print(the_items)

## dict_items([('nom', 'Kyrie'), ('prenom', 'John'), ('
    naissance', 1992), ('equipes', ['Cleveland', 'Boston'])])

print(type(the_items))

## <class 'dict_items'>
```

3.4.3 Search for Belonging

Thanks to the methods `keys()`, `values()` and `items()`, it is easy to search for the presence of objects in a dictionary.

```
print("age" in the_keys)

## False

print("nom" in the_keys)

## True

print(['Cleveland', 'Boston'] in the_values)

## True
```

3.4.4 Modification

3.4.4.1 Replacement

To replace the value associated with a key, the brackets (`[]`) and the equality sign (`=`) can be used.

For example, to replace the values associated with the `team` key:

```
my_dict["equipes"] = ["Montclair Kimberley Academy",
                    "Cleveland Cavaliers", "Boston Celtics"]
print(my_dict)

## {'nom': 'Kyrie', 'prenom': 'John', 'naissance': 1992, '
    equipes': ['Montclair Kimberley Academy', 'Cleveland
    Cavaliers', 'Boston Celtics']}
```

3.4.4.2 Adding Elements

Adding an element to a dictionary can be done with brackets (`[]`) and the equality sign (`=`):

```
my_dict["taille_cm"] = 191
print(my_dict)

## {'nom': 'Kyrie', 'prenom': 'John', 'naissance': 1992, '
    equipes': ['Montclair Kimberley Academy', 'Cleveland
    Cavaliers', 'Boston Celtics'], 'taille_cm': 191}
```

To add the content of another dictionary to a dictionary, Python offers the `update()` method.

Let's create a second dictionary first:

```
second_dict = {"masse_kg" : 88, "debut_nba" : 2011}
print(second_dict)

## {'masse_kg': 88, 'debut_nba': 2011}
```

Let's add the content of this second dictionary to the first:

```
my_dict.update(second_dict)
print(my_dict)
```

```
## {'nom': 'Kyrie', 'prenom': 'John', 'naissance': 1992, '
    equipes': ['Montclair Kimberley Academy', 'Cleveland
    Cavaliers', 'Boston Celtics'], 'taille_cm': 191, 'masse_kg
    ': 88, 'debut_nba': 2011}
```

If the second dictionary is subsequently modified, it will not affect the first:

```
second_dict["poste"] = "PG"
print(second_dict)
```

```
## {'masse_kg': 88, 'debut_nba': 2011, 'poste': 'PG'}
```

```
print(my_dict)
```

```
## {'nom': 'Kyrie', 'prenom': 'John', 'naissance': 1992, '
    equipes': ['Montclair Kimberley Academy', 'Cleveland
    Cavaliers', 'Boston Celtics'], 'taille_cm': 191, 'masse_kg
    ': 88, 'debut_nba': 2011}
```

3.4.4.3 Deleting elements

There are several ways to delete an element in a dictionary. For example, with the operator `del` :

```
del my_dict["debut_nba"]
print(my_dict)
```

```
## {'nom': 'Kyrie', 'prenom': 'John', 'naissance': 1992, '
    equipes': ['Montclair Kimberley Academy', 'Cleveland
    Cavaliers', 'Boston Celtics'], 'taille_cm': 191, 'masse_kg
    ': 88}
```

It is also possible to use the `pop()` method:


```
res = my_dict.pop("masse_kg")
print(my_dict)

## {'nom': 'Kyrie', 'prenom': 'John', 'naissance': 1992, '
    equipes': ['Montclair Kimberley Academy', 'Cleveland
    Cavaliers', 'Boston Celtics'], 'taille_cm': 191}
```

In the previous instruction, we added an assignment of the result of applying the `pop()` method to a variable named `res`. As can be seen, the `pop()` method, in addition to deleting the key, returned the associated value:

```
print(res)
```

```
## 88
```

3.4.5 Copy of a Dictionary

To copy a dictionary, and not create a reference (which is the case if you use the equality sign), Python provides, as for sets, a `copy()` method:

```
d = {"Marseille": 13, "Rennes" : 35}
d2 = d.copy()
d2["Paris"] = 75
print("d: ", d)
```

```
## d:  {'Marseille': 13, 'Rennes': 35}
```

```
print("d2: ", d2)
```

```
## d2:  {'Marseille': 13, 'Rennes': 35, 'Paris': 75}
```

3.4.6 Exercise

1. Create a dictionary named `photo`, including the following key-value pairs:
2. key: `id`, value: `1`,
3. key: `description`, value: `A photo of the Old Port of Marseille`,
4. key: `loc`, value: a list in which the following coordinates are given `5.3772133`, `43.302424`.
5. add the following key-value pair to the `photo` dictionary: key : `user`, value : `bob`.
6. Look for an entry with a key that is worth `description` in the `photo` dictionary. If this is the case, display the corresponding entry (key and value).
7. Delete the entry in `photo` whose key is `user`.
8. Modify the value of the entry `loc` in the `photo` dictionary, to propose a new list, whose coordinates are as follows: `5.3692712` and `43.2949627`.

Chapter 4

Operators

Python includes different operators, allowing operations to be performed between operands, *i.e.*, between variables, literals or expressions.

4.1 Arithmetic Operators

The basic arithmetic operators are integrated in Python.

We have already used some of them in the previous chapters to perform operations on integers or floating point numbers (addition, subtraction, etc.). Let's take a quick look at the most common arithmetic operators used to perform operations on numbers.

4.1.1 Addition

An addition between two numbers is made using the + symbol:

```
print(1+1) # Addition
```

```
## 2
```

4.1.2 Subtraction

A subtraction between two numbers is performed using the `-` symbol:

```
print(1-1) # Subtraction
```

```
## 0
```

4.1.3 Multiplication

A multiplication between two numbers is performed using the `*` symbol:

```
print(2*2) # Multiplication
```

```
## 4
```

4.1.4 Division

A (real) division between two numbers is made using the symbol `/`:

```
print(3/2) # Division
```

```
## 1.5
```

To perform a Euclidean division (or division with remainder), slash is doubled:

```
print(3//2) # Euclidean division
```

```
## 1
```

4.1.5 Modulo

The modulo (remainder of the Euclidean division) is obtained using the symbol %:

```
print(12%10) # Modulo
```

```
## 2
```

4.1.6 Power

To raise a number to a given power, we use two stars (**):

```
print(2**3) # 2^3
```

```
## 8
```

4.1.7 Order

The order of operations follows the PEMDAS rule (*Parentheses, Exponents, Multiplication and Division, Addition and Subtraction*).

For example, the following instruction first performs the calculation 2×2 , then adds 1:

```
print(2*2+1)
```

```
## 5
```

The following instruction, using brackets, first calculates $2 + 1$, then multiplies the result with 2:

```
print(2*(2+1))
```

```
## 6
```

4.1.8 Mathematical Operators on Strings

Some mathematical operators presented in Section 4.1 can be applied to strings.

When using the symbol `+` between two strings, Python concatenates these two strings (see Section 2.1.1) :

```
a = "euro"
b = "dollar"
print(a+b)

## eurodollar
```

When a string is “multiplied” by a scalar n , Python repeats this string n times:

```
2*a

## 'euroeuro'
```

4.1.9 Mathematical Operators on Lists or tuples

Some mathematical operators can also be applied to lists.

When using the symbol `+` between two lists, Python concatenates them into one:

```
l_1 = [1, "apple", 5, 7]
l_2 = [9, 11]
print(l_1 + l_2)

## [1, 'apple', 5, 7, 9, 11]
```

Same with tuples:

```
t_1 = (1, "apple", 5, 7)
t_2 = (9, 11)
print(t_1 + t_2)
```

```
## (1, 'apple', 5, 7, 9, 11)
```

By “multiplying” a list by a scalar n , Python repeats this list n times:

```
print(3*l_1)
```

```
## [1, 'apple', 5, 7, 1, 'apple', 5, 7, 1, 'apple', 5, 7]
```

Same with tuples:

```
print(3*t_1)
```

```
## (1, 'apple', 5, 7, 1, 'apple', 5, 7, 1, 'apple', 5, 7)
```

4.2 Comparison Operators

Comparison operators allow objects of all basic types to be compared with each other. The result of a comparison test produces Boolean values.

Table 4.1: Comparison operators

Operator	Python Operator	Description
=	==	Equal to
≠	!= (or <>)	Different from
>	>	Greater than
≥	>=	& Greater than or equal to
<	<	Lower than
≤	<=	Less than or equal to
∈	in	In
∉	not in	Not it

4.2.1 Equality, Inequality

To test the content equality between two objects:

```
a = "Hello"
b = "World"
c = "World"

print(a == c)
```

```
## False
```

```
print(b == c)
```

```
## True
```

The inequality between two objects:

```
x = [1,2,3]
y = [1,2,3]
z = [1,3,4]

print(x != y)
```

```
## False
```

```
print(x != z)
```

```
## True
```

4.2.2 Inferiority and Superiority, Strict or Broad

To know if an object is inferior (strictly or not) or superior (strictly or not) to another:


```
x = 1
y = 1
z = 2

print(x < y)
```

```
## False
```

```
print(x <= y)
```

```
## True
```

```
print(x > z)
```

```
## False
```

```
print(x >= z)
```

```
## False
```

It is also possible to compare two strings. The comparison is carried out according to the lexicographical order:

```
m_1 = "eat"
m_2 = "eating"
m_3 = "drinking"
print(m_1 < m_2) # eat before eating?
```

```
## True
```

```
print(m_3 > m_1) # drinking after eat?
```

```
## False
```

When comparing two lists together, Python works step by step. Let's look through an example to see how this comparison is done.

Let's create two lists:

```
x = [1, 3, 5, 7]
y = [9, 11]
```

Python will start by comparing the first elements of each list (here, it is possible, the two elements are comparable; otherwise, an error would be returned):

```
print(x < y)

## True
```

As $1 < 9$, Python returns `True`.

Let's change `x` so that the first element is greater than the first element of `y`.

```
x = [10, 3, 5, 7]
y = [9, 11]
print(x < y)
```

```
## False
```

This time, as $10 > 9$, Python returns `False`.

Now let's change the first element of `x` so that it is equal to `y`:

```
x = [10, 3, 5, 7]
y = [10, 11]
print(x < y)
```

```
## True
```

This time, Python compares the first element of `x` with that of `y`. As the two are identical, the second elements are compared. This can be demonstrated by evaluating the following code:

```
x = [10, 12, 5, 7]
y = [10, 11]
print(x < y)
```

```
## False
```

4.2.3 Inclusion and exclusion

As encountered several times in Chapter 3, the inclusion tests are performed using the operator `in`.

```
print(3 in [1,2, 3])
```

```
## True
```

To test if an item is excluded from a list, tuple, dictionary, etc., we use `not in`:

```
print(4 not in [1,2, 3])
```

```
## True
```

```
print(4 not in [1,2, 3, 4])
```

```
## False
```

With a dictionary:

```
dictionnaire = {"nom": "Rockwell", "prenom": "Criquette"}
"age" not in dictionnaire.keys()
```

```
## True
```

4.3 Logical operators

Logical operators operate on one or more logical objects (Boolean).

4.3.1 And logical

The `and` operator allows logical “AND” comparisons to be made. We compare two objects, `x` and `y` (these objects can result from a previous comparison, for this both only need to be Boolean).

If one of the two objects `x` and `y` is true, the logical “AND” comparison returns true:

```
x = True
y = True
print(x and y)

## True
```

If at least one of them is false, the logical “AND” comparison returns false:

```
x = True
y = False

print(x and y)

## False
```

```
print(y and y)

## False
```

If one of the two compared objects is equal to the empty value (`None`), then the logical “AND” comparison returns :

- the value `None` if the other object is worth `True` or `None`
- the value `False` if the other object is worth `False`.

```
x = True
y = False
z = None
print(x and z)
```

```
## None
```

```
print(y and z)
```

```
## False
```

```
print(z and z)
```

```
## None
```

4.3.2 Logical OR

The operator `or` allows logical “OR” comparisons to be made. Again, we compare two Booleans, `x` and `y`.

If at least one of the two objects `x` and `y` is true, the logical “OR” comparison returns true:

```
x = True
y = False
print(x or y)
```

```
## True
```

If both are false, the logical “OR” comparison returns false:

```
x = False
y = False
print(x or y)
```

```
## False
```

If one of the two objects is `None`, the logical “OR” comparison returns :

- True if the other object is worth True
- None if the other object is worth False or None.

```
x = True
y = False
z = None
print(x or z)
```

```
## True
```

```
print(y or z)
```

```
## None
```

```
print(z or z)
```

```
## None
```

4.3.3 Logical Not

The operator `not`, when applied to a Boolean, evaluates the latter at its opposite value:

```
x = True
y = False
print(not x)
```

```
## False
```

```
print(not y)
```

```
## True
```

When using the operator `not` on an empty value (`None`), Python returns `True`:

```
x = None
not x
```

```
## True
```

4.4 Some Functions

Python has many useful functions for manipulating structures and data. Table 4.2 lists some of them. Some require the loading of the `math` library, others require the `statistics` library. We will see other functions specific to the `NumPy` library in Chapter 9.

Table 4.2: Some numerical functions

Function	Description
<code>math.ceil(x)</code>	Smallest integer greater than or equal to <code>x</code>
<code>math.copysign(x, y)</code>	Absolute value of <code>x</code> but with the sign of <code>y</code>
<code>math.floor(x)</code>	Smallest integer less than or equal to <code>x</code>
<code>math.round(x, ndigits)</code>	Rounded from <code>x</code> to <code>ndigits</code> decimal places
<code>math.fabs(x)</code>	Absolute value of <code>x</code>
<code>math.exp(x)</code>	Exponential of <code>x</code>
<code>math.log(x)</code>	Natural logarithm of <code>x</code> (based on <code>e</code>)
<code>math.log(x, b)</code>	Logarithm based on <code>b</code> of <code>x</code>
<code>math.log10(x)</code>	Logarithm in base 10 of <code>x</code>
<code>math.pow(x, y)</code>	<code>x</code> high to the power <code>y</code>
<code>math.sqrt(x)</code>	Square root of <code>x</code>
<code>math.fsum()</code>	Sum of the values of <code>x</code>

Function	Description
<code>math.sin(x)</code>	Sine of <code>x</code>
<code>math.cos(x)</code>	Cosine of <code>x</code>
<code>math.tan(x)</code>	Tangent of <code>x</code>
<code>math.asin(x)</code>	Arc-sine of <code>x</code>
<code>math.acos(x)</code>	Arc-cosine of <code>x</code>
<code>math.atan(x)</code>	Arc-tangent of <code>x</code>
<code>math.sinh(x)</code>	Hyperbolic sine of <code>x</code>
<code>math.cosh(x)</code>	Hyperbolic cosine of <code>x</code>
<code>math.tanh(x)</code>	Hyperbolic tangent of <code>x</code>
<code>math.asinh(x)</code>	Hyperbolic arc-sine of <code>x</code>
<code>math.acosh(x)</code>	Hyperbolic arc-cosine of <code>x</code>
<code>math.atanh(x)</code>	Hyperbolic arc-tangent of <code>x</code>
<code>math.degree(x)</code>	Conversion of radians <code>x</code> to degrees
<code>math.radians(x)</code>	Conversion of <code>x</code> from degrees to radians
<code>math.factorial(x)</code>	Factorial of <code>x</code>
<code>math.gcd(x, y)</code>	Largest common divisor of <code>x</code> and <code>y</code>
<code>math.isclose(x, y, rel_tol=1e-09, abs_tol=0.0)</code>	Compare <code>x</code> and <code>y</code> and returns if they are close to the tolerance level <code>rel_tol</code> (<code>abs_tol</code> is the absolute minimum tolerance)
<code>math.isfinite(x)</code>	Returns <code>True</code> if <code>x</code> is either finite, or <code>NaN</code>
<code>math.isinf(x)</code>	Returns <code>True</code> if <code>x</code> is infinite, <code>False</code> otherwise
<code>math.isnan(x)</code>	Returns <code>True</code> if <code>x</code> is <code>NaN</code> , <code>False</code> if not
<code>statistics.mean(x)</code>	Average of <code>x</code>
<code>statistics.median(x)</code>	Median of <code>x</code>
<code>statistics.mode(x)</code>	Mode of <code>x</code>
<code>statistics.stdev(x)</code>	Standard deviation of <code>x</code>
<code>statistics.variance(x)</code>	Variance of <code>x</code>

4.5 Some Constants

The `math` library offers some constants, as shown in Table 4.3.

Table 4.3: Some constants integrated in Python

Function	Description
<code>math.pi</code>	The number Pi (π)
<code>math.e</code>	The constant e
<code>math.tau</code>	The constant τ , equal to 2π
<code>math.inf</code>	The infinite (∞)
<code>-math.inf</code>	Minus infinity ($-\infty$)
<code>math.nan</code>	Floating point number <i>not to number</i>

4.6 Exercise

1. Calculate the rest of the Euclidean division of 10 by 3.
2. Display the largest common divisor between 6209 and 4435.
3. Let us consider two objects: `a = 18` and `b = -4`. Test it if:
 - `a` is strictly less than `b`,
 - `a` is greater than or equal to `b`,
 - `a` is different from `b`.
4. Let `x` be the list such as `x = [1, 1, 1, 2, 3, 5, 8]`. Check whether:
 - 1 is in `x`;
 - 0 is in `x`;
 - 1 and 0 are in `x`;
 - 1 or 0 are in `x`;
 - 1 or 0 is not present in `x`.

Chapter 5

Loading and Saving Data

To explore data and/or perform statistical or econometric analyses, it is important to know how to import and export data.

First of all, it is important to mention the notion of a working directory. In computer science, the current directory of a process refers to a directory of the file system associated with that process.

When we launch Jupyter Notebook, a tree structure is displayed, and we navigate inside it to create or open a *notebook*. The directory containing the *notebook* is the current directory. When Python is told to import data (or export objects), the origin (or destination) will be indicated **relatively** in the current directory, unless absolute paths (*i.e.*, a path from the root /) are used.

If a Python program is started from a terminal, the current directory is the directory in which the terminal is located at the time the program is started.

To display the current directory in Python, the following code can be used:

```
import os
cwd = os.getcwd()
print(cwd)

## /Users/ewengallic/Dropbox/Universite_Aix_Marseille/
##   Magistere_2_Programming_for_big_data/Cours/chapters/python/
##   Python_for_economists
```

Remarque 5.0.1

The `listdir()` function of the `os` library is very useful: it allows to list all the documents and directories contained in the current directory, or in any directory if the parameter `path` informs the path (absolute or relative). After importing the function (`from os import getcwd`), it can be called: `os.listdir()`.

5.1 Load Data

Depending on the data format, data import techniques differ.

Remarque 5.1.1

Chapter ?? provides other ways to import data, with the `pandas` library.

5.1.1 Fichiers textes

When the data is present in a text file (ASCII), Python offers the `open()` function.

The (simplified) syntax of the `open()` function is as follows:

```
open(file, mode='r', buffering=-1,
      encoding=None, errors=None, newline=None)
```

Here is what the parameters correspond to (there are others):

- **file**: a string indicating the path and name of the file to be opened;
- **mode**: specifies the way the file is opened (see the lines below for possible values);
- **buffering**: specifies using an integer the behavior to be adopted for buffering (1 to buffering per line; an integer > 1 to indicate the size in bytes of the chunks to be buffered);
- **encoding**: specifies the encoding of the file;
- **errors**: specifies how to handle encoding and decoding errors (*e.g.*, **strict** returns an exception error, **ignore** ignores errors, **replace** replaces them, **backslashreplace** replaces malformed data with escape sequences);
- **newline** : controls the end of the lines (`\n`, `\r`, etc.).

Table 5.1: Main Values for How to Open Files.

Value	Description
r	Opening to read (default)
w	Opening to write
x	Opening to create a document, fails if the file already exists
a	Opening to write, adding at the end of the file if it already exists
+	Opening for update (read and write)
b	To be added to an opening mode for binary files (rb or wb)
t	Text mode (automatic decoding of bytes in Unicode). Default if not specified (adds to the mode, like b)

It is important to remember to **close the file** once we have finished using it. To do this, we use the `close()` method.

In the `fichiers_exemples` folder is a file called `text_file.txt` which contains three lines of text. Let's open this file, and use the `.read()` method to display its content:

```
path = "./fichiers_exemples/fichier_texte.txt"
# Opening in read-only mode (default)
my_file = open(path, mode = "r")
print(my_file.read())
```

```
## Bonjour, je suis un fichier au format txt.
## Je contiens plusieurs lignes, l'idée étant de montrer
    comment fonctionne l'importation d'un tel fichier dans
    Python.
## Trois lignes devraient suffir.
```

```
my_file.close()
```

A common practice in Python is to open a file in a `with` block. The reason for this choice is that a file opened in such a block is automatically closed at the end of the block.

The syntax is as follows:

```
# Opening in read-only mode (default)
with open(path, "r") as mon_fichier:
    data = function_to_get_data_from_my_file()
```

For example, to retrieve each line as an element of a list, a loop running through each line of the file can be used. At each iteration, the line is retrieved:

```
# Opening in read-only mode (default)
with open(path, "r") as my_file:
    data = [x for x in my_file]
print(data)
```

```
## ['Bonjour, je suis un fichier au format txt.\n', "Je
    contiens plusieurs lignes, l'idée étant de montrer comment
    fonctionne l'importation d'un tel fichier dans Python.\n",
    'Trois lignes devraient suffir.']
```

Note: at each iteration, the `strip()` method can be applied. It returns the character string of the line, by removing any white characters at the beginning of the string :

```
# Opening in read-only mode (default)
with open(path, "r") as my_file:
    data = [x.strip() for x in my_file]
print(data)
```

```
## ['Bonjour, je suis un fichier au format txt.', "Je contiens
    plusieurs lignes, l'idée étant de montrer comment
    fonctionne l'importation d'un tel fichier dans Python.", '
    Trois lignes devraient suffir.']
```

The `readlines()` method can also be used to import lines into a list:

```
with open(path, "r") as my_file:
    data = my_file.readlines()
print(data)
```

```
## ['Bonjour, je suis un fichier au format txt.\n', "Je
    contiens plusieurs lignes, l'idée étant de montrer comment
```

```
fonctionne l'importation d'un tel fichier dans Python.\n",  
'Trois lignes devraient suffir.']
```

Character encoding may be a problem during import. In this case, it may be a good idea to change the value of the `encoding` parameter of the `open()` function. The available encodings depend on the locale. The available values are obtained using the following method (code not executed in these notes):

```
import locale  
locale.locale_alias
```

5.1.1.1 Import from the Internet

To import a text file from the Internet, methods from the `urllib` library can be used:

```
import urllib  
from urllib.request import urlopen  
url = "http://egallic.fr/Enseignement/Python/fichiers_exemples/fichier_texte.txt"  
with urllib.request.urlopen(url) as my_file:  
    data = my_file.read()  
print(data)  
  
## b"Bonjour, je suis un fichier au format txt.\nJe contiens  
plusieurs lignes, l'id\xc3\xa9e \xc3\xa9tant de montrer  
comment fonctionne l'importation d'un tel fichier dans  
Python.\nTrois lignes devraient suffir."
```

As can be seen, the encoding of characters is a concern here. We can apply the method `decode()`:

```
print(data.decode())  
  
## Bonjour, je suis un fichier au format txt.  
## Je contiens plusieurs lignes, l'idée étant de montrer  
comment fonctionne l'importation d'un tel fichier dans  
Python.  
## Trois lignes devraient suffir.
```

5.1.2 CSV Files

CSV files (*comma separated value*) are very common. Many databases export their data to CSV (e.g., World Bank, FAO, Eurostat, etc.). To import them into Python, you can use the `csv` module.

Again, we use the `open()` function, with the parameters described in Section 5.1.1. Then, we use the `reader()` method of the `csv` module:

```
import csv
with open('./fichiers_exemples/fichier_csv.csv') as my_file:
    my_file_reader = csv.reader(my_file, delimiter=',', quotechar='"')
    data = [x for x in my_file_reader]

print(data)

## [['nom', 'prénom', 'équipe'], ['Irving', ' "Kyrie"', ' "
    Celtics"'], ['James', ' "Lebron"', ' "Lakers"', ''], ['
    Curry', ' "Stephen"', ' "Golden State Warriors"']]
```

The `reader()` method can take several arguments, described in Table 5.2.

Table 5.2: Parameters of the `reader()` Function

Argument	Description
<code>csvfile</code>	The object opened with <code>open()</code>
<code>dialect</code>	Parameter specifying the “dialect” of the CSV file (e.g., <code>excel</code> , <code>excel-tab</code> , <code>unix</code>)
<code>delimiter</code>	The character delimiting the fields (<i>i.e.</i> , the values of the variables)
<code>quotechar</code>	Character used to surround fields containing special characters
<code>escapechar</code>	Escape character
<code>doublequote</code>	Controls how the <code>quotechar</code> appear inside a field: when <code>True</code> , the character is doubled, when <code>False</code> , the escape character is used as a prefix to the <code>quotechar</code>
<code>lineterminator</code>	String of characters used to end a line
<code>skipinitialspace</code>	When <code>True</code> , the white character located just after the field separation character is ignored

Argument	Description
<code>strict</code>	When True, returns an exception error if there is a bad input of CSV

A CSV file can also be imported as a dictionary, using the `csv.DictReader()` method of the CSV module :

```
import csv
path = "./fichiers_exemples/fichier_csv.csv"
with open(path) as my_file:
    my_file_csv = csv.DictReader(my_file)
    data = [ligne for ligne in my_file_csv]
print(data)

## [OrderedDict([('nom', 'Irving'), ('prénom', ' "Kyrie"'), ('équipe', ' "Celtics"')]), OrderedDict([('nom', 'James'), ('prénom', ' "Lebron"'), ('équipe', ' "Lakers"'), (None, [''])]), OrderedDict([('nom', 'Curry'), ('prénom', ' "Stephen"'), ('équipe', ' "Golden State Warriors"')])]
```

5.1.2.1 Import From the Internet

As with txt files, a CSV file hosted on the Internet can be loaded:

```
import csv
import urllib.request
import codecs

url = "http://egallic.fr/Enseignement/Python/fichiers_exemples/fichier_csv.csv"
with urllib.request.urlopen(url) as my_file:
    my_file_csv = csv.reader(codecs.iterdecode(my_file, 'utf-8'))
    data = [ligne for ligne in my_file_csv]
print(data)

## [['nom', 'prénom', 'équipe'], ['Irving', ' "Kyrie"', ' "Celtics"'], ['James', ' "Lebron"', ' "Lakers"', ''], ['
```

```
Curry', ' "Stephen"', ' "Golden State Warriors"']]
```

5.1.3 JSON Files

To import files in JSON format (*JavaScript Object Notation*), which are widely used when communicating with an API, you can use the `json` library, and its `load()` method:

```
import json
url = './fichiers_exemples/tweets.json'

with open(url) as my_file_json:
    data = json.load(my_file_json)
```

Then, you can display the imported content using the `pprint()` function:

```
from pprint import pprint
pprint(data)

## {'created_at': 'Wed Sep 26 07:38:05 +0000 2018',
##  'id': 11,
##  'loc': [{'long': 5.3698}, {'lat': 43.2965}],
##  'text': 'Un tweet !',
##  'user_mentions': [{'id': 111, 'screen_name': 'nom_twittos1'},
##                    {'id': 112, 'screen_name': 'nom_twittos2'}]}}
```

5.1.3.1 Import from the Internet

Once again, it is possible to import JSON files from the Internet:

```
import urllib
from urllib.request import urlopen
url = "http://egallic.fr/Enseignement/Python/fichiers_exemples/tweets.json"
with urllib.request.urlopen(url) as my_file:
```

```
data = json.load(my_file)
pprint(data)

## {'created_at': 'Wed Sep 26 07:38:05 +0000 2018',
##  'id': 11,
##  'loc': [{'long': 5.3698}, {'lat': 43.2965}],
##  'text': 'Un tweet !',
##  'user_mentions': [{'id': 111, 'screen_name': 'nom_twittos1
##                       },
##                       {'id': 112, 'screen_name': 'nom_twittos2
##                       }]]}
```

5.1.4 Excel Files

Excel files (`xls` or `xlsx`) are also widely used in economics. The reader is referred to Section ?? for a method of importing Excel data with the `pandas` library.

5.2 Exporting data

It is not uncommon to have to export data, for instance to share it. Again, the function `open()` is used, by playing with the value of the parameter `mode` (see Table 5.1).

5.2.1 Text Files

Let's say we need to export lines of text to a file. Before giving an example with the `open()` function, let's look at two important functions to convert the contents of some objects to text.

The first, `str()`, returns a string version of an object. We have already applied it to numbers that we wanted to concatenate in Section 2.1.4.

```
x = ["pomme", 1, 3]
str(x)
```

```
## '['pomme', 1, 3]"
```

The result of this instruction returns the list as a string: `"['pomme', 1, 3]"`.

The second function that seems important to address is `repr()`. This function returns a string containing a printable representation on an object screen. In addition, this channel can be read by the interpreter.

```
y = "Fromage, tu veux du fromage ?\n"
repr(y)
```

```
## "'Fromage, tu veux du fromage ?\n'"
```

The result writes: `"'Fromage, tu veux du fromage ?\n'"`.

Let's say we want to export two lines:

- the first, a text that indicates a title (“Kyrie Irving Characteristics”);
- the second, a dictionary containing information about Kyrie Irving (see below).

Let's define this dictionary:

```
z = { "name": "Kyrie",
      "surname": "Irving",
      "date_of_birth": 1992,
      "teams": ["Cleveland", "Boston", "Nets"]}
```

One of the syntaxes for exporting data in txt format is:

```
# Ouverture en mode lecture (par défaut)
path = "path/to/file.txt"
with open(path, "w") as my_file:
    function_to_export()
```

We create a variable indicating the path to the file. Then we open the file in writing mode by specifying the parameter `mode = "w"`. Then, we still have to write our lines in the file.

```
path = "./fichiers_exemples/Irving.txt"
with open(path, mode = "w") as my_file:
```

```
my_file.write("Characteristics of Kyrie Irving\n")
my_file.writelines(repr(z))
```

```
## 32
```

If the file is already existing, having used `mode="w"`, the old file will be overwritten by the new one. If we want to add lines to the existing file, we will use `mode="a"` for example:

```
with open(path, mode = "a") as my_file:
    my_file.writelines("\nAnother line\n")
```

If we want to be warned if the file already exists, and to make the writing fail if this is the case, we can use `mode="x"`:

```
with open(path, mode = "x") as my_file:
    my_file.writelines("A new line that will not be added\n")
```

```
## Error in py_call_impl(callable, dots$args, dots$keywords):
##   FileExistsError: [Errno 17] File exists: './fichiers_exemples/Irving.txt'
##
## Detailed traceback:
##   File "<string>", line 1, in <module>
```

5.2.2 CSV Files

As economists, we are more likely to have to export data in CSV format rather than text, due to the rectangular structure of the data we are handling. As for the import of CSV (c.f. Section 5.1.2), on utilise le module `csv`. we use the module `csv`. To write to the file, we use the `writer()` method. The formatting parameters of this function are the same as those of the `reader()` function (see Table 5.2).

Example of creating a CSV file:

```
import csv
path = "./fichiers_exemples/fichier_export.csv"
```

```

with open(path, mode='w') as my_file:
    my_file_write = csv.writer(my_file, delimiter=',',
                               quotechar='"',
                               quoting=csv.QUOTE_MINIMAL)

    my_file_write.writerow(['Country', 'Year', 'Quarter', 'GR_PIB'])
    my_file_write.writerow(['France', '2017', 'Q4', 0.7])
    my_file_write.writerow(['France', '2018', 'Q1', 0.2])

## 29
## 20
## 20

```

Of course, most of the time, we do not write each entry by hand. We export the data contained in a structure. Section ?? provides examples of this type of export, when the data are contained in two-dimensional tables created with the `pandas` library.

5.2.3 JSON Files

It may be necessary to save structured data in JSON format, for example when an API (*e.g.*, the Twitter API) has been used that returns objects in JSON format.

To do this, we will use the `json` library, and its `dump()` method. This method allows to serialize an object (for example a list, like what you get with the Twitter API queried with the `twitter-python` library) in JSON.

```

import json
x = [1, "apple", ["seed", "red"]]
y = { "name": "Kyrie",
      "surname": "John",
      "year_of_birth": 1992,
      "teams": ["Cleveland", "Boston", "Nets"]}
x_json = json.dumps(x)
y_json = json.dumps(y)

print("x_json: ", x_json)

```

```
## x_json:  [1, "apple", ["seed", "red"]]
```

```
print("y_json: ", y_json)
```

```
## y_json:  {"name": "Kyrie", "surname": "John", "
  year_of_birth": 1992, "teams": ["Cleveland", "Boston", "
  Nets"]}
```

As can be seen, there are some minor problems with accentuated character rendering. We can specify, using the parameter `ensure_ascii` evaluated at `False` that we do not want to make sure that non-ascii characters are escaped by sequences of type `\uXXXX`.

```
x_json = json.dumps(x, ensure_ascii=False)
y_json = json.dumps(y, ensure_ascii=False)
```

```
print("x_json: ", x_json)
```

```
## x_json:  [1, "apple", ["seed", "red"]]
```

```
print("y_json: ", y_json)
```

```
## y_json:  {"name": "Kyrie", "surname": "John", "
  year_of_birth": 1992, "teams": ["Cleveland", "Boston", "
  Nets"]}
```

```
path = "./fichiers_exemples/export_json.json"
```

```
with open(path, 'w') as f:
    json.dump(json.dumps(x, ensure_ascii=False), f)
    f.write('\n')
    json.dump(json.dumps(y, ensure_ascii=False), f)
```

```
## 1
```

If we want to re-import in Python the content of the file `export_json.json`:

```
path = "./fichiers_exemples/export_json.json"
with open(path, "r") as f:
    data = []
    for line in f:
        data.append(json.loads(line, encoding="utf-8"))

print(data)

## [['[1, "apple", ["seed", "red"]]', '{"name": "Kyrie", "
  surname": "John", "year_of_birth": 1992, "teams": ["
  Cleveland", "Boston", "Nets"]}']]
```

5.2.4 Exercise

1. Create a list named `a` containing information on the unemployment rate in France in the second quarter of 2018. This list must contain three elements:
 - the year;
 - the quarter;
 - the value of the unemployment rate (9.1%).
2. Export the contents of the list `a` in CSV format, preceded by a line specifying the names of the fields. Use the semicolon (;) as a field separator.
3. Import the file created in the previous question into Python.

Chapter 6

Conditions

Often, depending on the evaluation of an expression, one wants to perform one operation rather than another. For example, when a new variable is created in a statistical analysis, and this variable takes its values according to another, it may be necessary to use **conditional instructions** : “if the value is less than x , then... otherwise, ...”.

In this short chapter, we look at how to write conditional instructions.

6.1 Conditional if Instructions

The simplest conditional instruction that can be found is `if`. If and only if an expression is evaluated at `True`, then an instruction will be evaluated.

The syntax is as follows:

```
if expression:  
    instruction
```

The lines after the colon (`:`) must be placed in a block, using a tab stop.

Remarque 6.1.1

A code block is a grouping of statements. Nested codes indented at the same position are part of the same block:

```
block 1 line
block 1 line
    block2 line
    block2 line
block line1
```

In the code below, we define a variable, `x`, that contains the integer 2. The following instruction evaluates the expression `x == 2` (see Section [@ref\(#opérateurs-comparaison\)](#) for reminders on comparison operators). If the result of this expression is `True`, then the content of the block is evaluated.

```
x = 2
if x == 2:
    print("Hello")
```

```
## Hello
```

If we change the value of `x` so that the expression `x == 2` returns `False`:

```
x = 3
if x == 2:
    print("Hello")
```

Inside the block, several instructions can be written that will be evaluated if the expression is `True`:

```
x = 2
if x == 2:
    y = "Hello"
    print(y + ", x vaut : " + str(x))
```

```
## Hello, x vaut : 2
```

Remarque 6.1.2

When writing a code, it may be practical to use ‘if’ conditional instructions to evaluate or not certain parts of the code. For example, when we write a script, there are times when we have to re-evaluate the beginning, but some parts don’t need to be re-evaluated every time, like graphical outputs (which takes time). Of course, it is possible to comment on these parts of codes that do not require a new evaluation. But we can also put the instructions in a conditional block:

- at the beginning of the script, we create a variable `graph = False`;
- before creating a graph, it is placed in a block `if graph:`

When executing the script, it is then possible to choose to create and export the graphs of the `if graph:` blocks by modifying the `graph` variable as desired.

6.2 if-else Conditional Instructions

If the condition is not verified, other instructions can be evaluated using the ‘if-else’ instructions.

The syntax is as follows:

```
if expression:
    instructions
else:
    other_instructions
```

For example, suppose we want to create a variable related to temperature, taking the value `warm` if the value of the variable `temperature` exceeds 28 degrees C, otherwise `cold`. Let’s say the temperature is 26 degrees C:

```
temperature = 26
heat = ""

if temperature > 28:
    heat = "hot"
else:
    heat = "cold"

print("It is " + heat + " out there")
```

```
## It is cold out there
```

If the temperature is now 32 degrees C:

```
temperature = 32
heat = ""

if temperature > 28:
    heat = "hot"
else:
    heat = "cold"

print("It is " + heat + " out there")

## It is hot out there
```

6.3 if-elif Conditional Instructions

If the condition is not verified, another one can be tested and then other instructions evaluated if the second one is verified. Otherwise, another one can be tested, and so on. Instructions may also be evaluated if none of the conditions have been assessed at `True`. To do this, conditional ‘if-elif’ instructions can be used.

The syntax is as follows:

```
if expression:
    instructions
elif expression_2:
    instructions_2
elif expression_3:
    instructions_3
else:
    other_instruction
```

The previous example lacks some common sense. Can we say that the fact that it is 28 degrees C or less it is cold? Let’s add a few nuances:

```
temperature = -4
heat = ""

if temperature > 28:
    heat = "hot"
elif temperature <= 28 and temperature > 15:
    heat = "not too hot"
elif temperature <= 15 and temperature > 0:
    heat = "cold"
else:
    heat = "very cold"

print("It is " + heat + " out there")

## It is very cold out there
```

Remarque 6.3.1

The advantage of using `if-elif` conditional instructions over writing several `if` conditional instructions in succession is that with the first way of doing things, comparisons stop as soon as one is completed, which is more efficient.

6.4 Exercise

Let us consider a list named `europa` containing the following values, as strings: “Germany”, “France” and “Spain”.

Let us consider a second list, named `asia`, containing in the form of strings: “Vietnam”, “China” and “India”.

The objective will be to create a `continent` variable that will indicate either Europe, Asia or other at the end of the code execution.

Using conditional instructions of the `if-elif` type, write a code that checks the value of a variable named `country`. Another variable, named `continent` will take values depending on the content of the former one, such that:

- if the country value is present in the `europa` list, the variable `continent` should be set to `Europe`
- if the country value is present in the `asia` list, the variable `continent` should be set to `Asia`
- if the country value is not present in `europa` or `asia`, the variable `continent` will be set to `Other`.

To do this:

1. Create the two lists `europa` and `asia` as well as the variable `country` (setting the value to `Spain`) and the variable `continent` (initiated with an empty character string).
2. Write the code to achieve the explained objective, and display the content of the `continent` variable at the end of the execution.
3. Change the initial value of `country` to `China` then `Brazil` and in each case, execute the code written in the previous question.

Chapter 7

Loops

When the same operation has to be repeated several times, for a given number of times or as long as a condition is verified (or as long as it is not verified), loops can be used, which is much less painful than evaluating by hand or by copying and pasting the same instruction.

We will discuss two types of loops in this chapter:

- those for which we do not know *a priori* the number of iterations (the number of repetitions) to be performed: `while()` loops
- those for which we know *a priori* how many iterations are necessary: `for()` loops

Remarque 7.0.1

It is possible to stop a `for()` loop before a predefined number of iterations; in the same way, it is possible to use a `while()` loop by knowing in advance how many iterations to perform.

7.1 Loops with `while()`

The principle of a `while()` loop is that instructions inside the loop will be repeated as long as a condition is met. The idea is to make this condition depend on one or more objects that will be modified during the iterations (otherwise, the loop would

turn infinitely).

The syntax is as follows:

```
while condition:
    instructions
```

As for conditional instructions (see Section 6), the instructions are placed inside a block.

Let's look at an example of a `while()` loop:

```
x = 100
while x/3 > 1:
    print(x/3)
    x = x/3

## 33.333333333333336
## 11.111111111111112
## 3.703703703703704
## 1.234567901234568
```

```
print(x/3>1)
```

```
## False
```

```
print(x/3)
```

```
## 0.41152263374485604
```

In this loop, at each iteration, the value of `x` divided by 3 is displayed, then the value of `x` is replaced by a third of its current value. This operation is repeated as long as the expression `x/3 > 1` returns `True`.

7.2 Loops with `for()`

When we know the number of iterations in advance, we can use a `for()` loop. The syntax is as follows:


```
for object in possible_values:  
    instructions
```

with `object` the name of a local variable at the function `for()`, `possible_values` an object comprising n elements defining the values that `object` will take for each of the n turns, and `instructions` the instructions that will be executed at each iteration.

In the following example, we will calculate the square of the first n integers. The values that our `object` variable (which we will call `i`) will take will be integers from 1 to n . To obtain a sequence of integers in Python, we can use the `range()` function, which takes the following parameters:

- `start` : (optional, default, 0) start value for the sequence (included) ;
- `stop` : end value of the sequence (not included) ;
- `step` : (optional, default 1) the step.

Before calculating the sequence of the n first squares, let's look at an example of how the `range()` function works:

```
print(list(range(0, 4))) # Les entiers de 0 à 3
```

```
## [0, 1, 2, 3]
```

```
print(list(range(4))) # Les entiers de 0 à 3
```

```
## [0, 1, 2, 3]
```

```
print(list(range(2, 10))) # Les entiers de 2 à 9
```

```
## [2, 3, 4, 5, 6, 7, 8, 9]
```

```
print(list(range(2, 10, 3))) # Les entiers de 2 à 9 par pas de 3
```

```
## [2, 5, 8]
```

To display the sequence of the first 10 first squares, we can write:

```
message = "The squared value of {} is {}"
n=10
for i in range(0, n+1):
    print(message.format(i,i**2))

## The squared value of 0 is 0
## The squared value of 1 is 1
## The squared value of 2 is 4
## The squared value of 3 is 9
## The squared value of 4 is 16
## The squared value of 5 is 25
## The squared value of 6 is 36
## The squared value of 7 is 49
## The squared value of 8 is 64
## The squared value of 9 is 81
## The squared value of 10 is 100
```

During the first iteration, *i* is 0. In the second case, *i* is 1. In the third, *i* is 2, etc.

If we want to store the result in a list:

```
n=10
n_squares = []
for i in range(0, n+1):
    n_squares.append(i**2)

print(n_squares)

## [0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

It is not mandatory to use the `range()` function in a `for()` loop, you can define the values “by hand”:

```
message = "The squared value of {} is {}"
for i in [0, 1, 2, 8, 9, 10]:
    print(message.format(i,i**2))

## The squared value of 0 is 0
## The squared value of 1 is 1
```

```
## The squared value of 2 is 4
## The squared value of 8 is 64
## The squared value of 9 is 81
## The squared value of 10 is 100
```

In the same spirit, it is not mandatory to iterate on numerical values:

```
message = "There is(are) {} letter(s) in the name: {}"
for first_name in ["Pascaline", "Gauthier", "Xuan", "Jimmy"]:
    print(message.format(len(first_name), first_name))
```

```
## There is(are) 9 letter(s) in the name: Pascaline
## There is(are) 8 letter(s) in the name: Gauthier
## There is(are) 4 letter(s) in the name: Xuan
## There is(are) 5 letter(s) in the name: Jimmy
```

Nothing prevents loops from being made inside loops:

```
message = "i equals {} and j equals {}"
for i in range(0,3):
    for j in range(0,3):
        print(message.format(i, j))
```

```
## i equals 0 and j equals 0
## i equals 0 and j equals 1
## i equals 0 and j equals 2
## i equals 1 and j equals 0
## i equals 1 and j equals 1
## i equals 1 and j equals 2
## i equals 2 and j equals 0
## i equals 2 and j equals 1
## i equals 2 and j equals 2
```

As can be seen, iteration is done for each value of *i*, and for each of these values, a second iteration is performed on the values of *j*.

Remarque 7.2.1

The letters `i` and `j` are often used to designate a counter in a `for()` loop, but this is obviously not a requirement.

In a loop, if we want to increment a counter, we can use the symbol `+=` rather than writing ‘`counter = counter + ...`’ :

```
message = "New value for j: {}"  
j = 10  
for i in range(0, 4):  
    j += 5  
    print(message.format(j))
```

```
## New value for j: 15  
## New value for j: 20  
## New value for j: 25  
## New value for j: 30
```

```
print(j)
```

```
## 30
```

7.3 Exercise

1. Write a very naive program to determine if a number is prime or not. To do this:
 1. define a `number` variable containing a natural integer of your choice (not too large),
 2. using a loop, check if each integer up to the square root of your number, is a divisor of your number (stop if ever it is the case)
 3. at the loop output, write a conditional instruction indicating whether or not the number is a prime one.

2. Choose a ‘mystery’ number between 1 and 100, and store it in an object called `mystere_number`. Then, create a loop that at each iteration performs a random draw of an integer between 1 and 100. As long as the number drawn is different from the mystery number, the loop must continue. At the output of the loop, a variable called `nb_drawings` will contain the number of prints made to obtain the mystery number.

Note: to draw a random number between 1 and 100, the method `randint()` of the module `random` may help).

3. Use a loop to scan integers from 1 to 20 using a for loop, displaying in the console at each iteration if the current number is even.
4. Use a `for()` loop to repeat the Fibonacci sequence until its tenth term (the F_n sequence is defined by the following recurrence relationship: $F_n = F_{n-1} + F_{n-2}$; the initial values are $F_0 = 0$ and $F_1 = 1$).

Chapter 8

Functions

Most of the time, we use the basic functions or those contained in modules. However, when retrieving data online or formatting data imported from various sources, it may be necessary to create our own functions. The advantage of creating one's functions is revealed when one has to carry out a series of instructions repeatedly, with some slight differences (we can then apply the functions within a loop, as we discussed in Chapter 7).

8.1 Definition

A function is declared using the keyword `def`. What it returns is returned using the keyword `return`.

La syntaxe est la suivante :

```
def name_function(arguments):  
    body of the function
```

Once the function is defined, it is called by referring to its name:

```
name_function()
```

So, all we need to do is add parentheses to the name of the function to call it. Indeed, `function_name` refers to the object that contains the function that is called using the

expression `function_name()`. For example, if we want to define the function that calculates the square of a number, here is what we can write:

```
def square(x):  
    return x**2
```

It can then be called:

```
print(square(2))
```

```
## 4
```

```
print(square(-3))
```

```
## 9
```

8.1.1 Adding a Description

It is possible (and strongly recommended) to add a description of what the function does, by adopting some conventions (see <https://www.python.org/dev/peps/pep-0257/>) =

```
def square(x):  
    """returns the squared value of x"""  
    return x**2
```

When the next instruction is then evaluated, the description of the function is displayed:

```
~?~(square)
```

In Jupyter Notebook, after writing the name of the function, the description can also be displayed by pressing the **Shift** and **Tabulation** keys on the keyboard.

8.1.2 Parameters of a Function

In the example of the `square()` function we created, we filled in only one argument, called `x`. If the function we wish to create requires several argument, they must be

separated by a comma.

Let us consider, for example, the following problem. We have a production function $Y(L, K, M)$, which depends on the number of workers L and the amount of capital K , and the equipment M , such that $Y(L, K, M) = L^{0.3}K^{0.5}M^2$. This function can be written in Python as follows:

```
def production(l, k, m):
    """
    Returns the value of the production according to
    labour, capital and materials

    Keyword arguments:
    l -- labour (float)
    k -- capital (float)
    m -- materials (float)
    """
    return l**0.3 * k**0.5 * m**(0.2)
```

8.1.2.1 Call Without Parameter Names

Using the previous example, if we are given $L = 60$ and $K = 42$ and $M = 40$, we can deduce the production:

```
prod_val = production(60, 42, 40)
print(prod_val)
```

```
## 46.289449781254994
```

It should be noted that the name of the parameters has not been mentioned here. When the function was called, the value of the first parameter was assigned to the first parameter (`l`), the second to the second parameter (`k`) and finally the third to the third parameter (`m`).

8.1.2.2 Positional Arguments, Arguments by Keywords

There are two types of arguments that can be given to a function in Python:

- the positional arguments;
- arguments by keywords.

Unlike positional arguments, keyword arguments have a default value assigned by default. We speak of a formal argument to designate the arguments of the function (the variables used in the body of the function) and an effective argument to designate the value that we wish to give to the formal argument. To define the value to be given to a formal argument, we use the equality symbol. When calling the function, if the user does not explicitly define a value, the default value will be assigned. Thus, it is not necessarily necessary to specify the arguments by keywords when calling the function.

It is important to note that positional arguments (those that do not have a default value) must appear first in the argument list.

Let's take an example with two positional arguments (`l` and `m`) and one argument per keyword (`k`):

```
def production_2(l, m, k=42):
    """
    Returns the value of the production according to
    labour, capital and materials

    Keyword arguments:
    l -- labour (float)
    m -- materials (float)
    k -- capital (float) (default 42)
    """
    return l**0.3 * k**0.5 * m**(0.2)
```

The `production_2()` function can be called, to give the same result, in the following three ways:

```
# By naming all argument, by omitting k
prod_val_1 = production_2(l = 42, m = 40)
# By naming all argument and specifying k
prod_val_2 = production_2(l = 42, m = 40, k = 42)
# By naming only the argument k
prod_val_3 = production_2(42, 40, k = 42)
# Without naming any argument
prod_val_4 = production_2(42, 40, 42)
```

```
res = [prod_val_1, prod_val_2, prod_val_3, prod_val_4]
print(res)
```

```
## [41.59215573604822, 41.59215573604822, 41.59215573604822,
    41.59215573604822]
```

Remarque 8.1.1

If the function contains several positional arguments; when evaluating:

- or all positional arguments are named by their name;
- or none;
- there are no in-between.

As long as all the positional arguments are named during the evaluation, they can be listed in different orders:

```
def production_3(a, l, m = 40, k=42):
    """
    Returns the value of the production according to
    labour, capital and materials

    Keyword arguments:
    a -- total factor productivity (float)
    l -- labour (float)
    m -- materials (float) (default 40)
    k -- capital (float) (default 42)
    """
    return a * l**0.3 * k**0.5 * m**(0.2)

prod_val_1 = production_3(1, 42, m = 38)
prod_val_2 = production_3(a = 1, l = 42)
prod_val_3 = production_3(l = 42, a = 1)
prod_val_4 = production_3(m = 40, l = 42, a = 1)

res = [prod_val_1, prod_val_2, prod_val_3, prod_val_4]
print(res)
```

```
## [41.16765711449734, 41.59215573604822, 41.59215573604822,
```

```
41.59215573604822]
```

8.1.2.3 Function as an Argument to Another Function

A function can be provided as an argument to another function.

```
def square(x):  
    """Returns the squared value of x"""  
    return x**2  
  
def apply_fun_to_4(fun):  
    """Applies the function `fun` to 4"""  
    return fun(4)  
  
print(apply_fun_to_4(square))  
  
## 16
```

8.2 Scope of a Function

When a function is called, the body of that function is interpreted. Variables that have been defined in the body of the function are assigned to a local *namespace*. In other words, they live only within this local space, which is created at the moment of the call of the function and destroyed at the end of it. This is referred to as the scope of the variables. Thus, a variable with a local scope (assigned in the local space) can have the same name as a global variable (defined in the global workspace), without designating the same object, or overwrite this object.

Let's look at this through an example.

```
# Definition of a global variable:  
value = 1  
  
# Definition of a local variable in function f
```

```
def f(x):
    value = 2
    new_value = 3
    print("value equals: ", value)
    print("new_value equals: ", new_value)
    return x + value
```

Let's call the `f()` function, then look at the `value` and `new_value` values after executing the function.

```
res = f(3)
```

```
## value equals: 2
## new_value equals: 3
```

```
print("value equals: ", value)
```

```
## value equals: 1
```

```
print("new_value equals: ", new_value)
```

```
## Error in py_call_impl(callable, dots$args, dots$keywords):
##   NameError: name 'new_value' is not defined
##
## Detailed traceback:
##   File "<string>", line 1, in <module>
```

As can be seen, during the evaluation, the local variable of the name `value` was 2, which did not refer to the variable of the same name defined in the global environment. After executing the `f()` function, this local `value` variable is deleted, and the same applies to the local `new_value` variable, which does not exist in the global environment (hence the error returned).

Without going into too much detail, it seems important to know some principles about the scope of variables. Variables are defined in environments, which are embedded in each other. If a variable is not defined in the body of a function, Python will search in a parent environment.

```
value = 1
def f(x):
    return x + value

print(f(2))
```

```
## 3
```

If we define a function within another function, and call a variable not defined in the body of that function, Python will search in the directly superior environment. If it does not find, it will search in the even higher environment, and so on until it reaches the global environment.

```
# The value variable is not defined in
# the local environment of g().
# Python will then search in f().
value = 1
def f():
    value = 2
    def g(x):
        return x + value

    return g(2)

print(f())
```

```
## 4
```

```
# The value variable is not defined in g() or f()
# but in the higher environment (here, global)
value = 1
def f():
    def g(x):
        return x + value

    return g(2)
```

```
print(f())
```

```
## 3
```

If a variable is defined in the body of a function and we want it to be accessible in the global environment, we can use the keyword `global`:

```
def f(x):  
    global y  
    y = x+1
```

```
f(3)  
print(y)
```

```
## 4
```

Remarque 8.2.1

The variable that we want to define globally from a local space of the function must not have the same name of one of the parameters.

8.3 Lambda Functions

Python offers what are called lambdas functions, or anonymous functions. A lambda function has only one instruction whose result is that of the function.

They are defined using the keyword `lambda`. The syntax is as follows:

```
name_function = lambda arguments : result
```

The arguments are to be separated by commas.

Let's take the function `square()` created previously:

```
def square(x):  
    return x**2
```

The equivalent lambda function is written:

```
square_2 = lambda x: x**2
print(square_2(4))
```

```
## 16
```

With several parameters, let's look at the lambda function equivalent to the `production()` function:

```
def production(l, k, m):
    """
    Returns the value of the production according to
    labour, capital and materials.

    Keyword arguments:
    l -- labour (float)
    k -- capital (float)
    m -- materials (float)
    """
    return l**0.3 * k**0.5 * m**(0.2)
```

```
production_2 = lambda l,k,m : l**0.3 * k**0.5 * m**(0.2)
print(production(42, 40, 42))
```

```
## 40.987803063838406
```

```
print(production_2(42, 40, 42))
```

```
## 40.987803063838406
```

8.4 Returning Several Values

It can sometimes be convenient to return several elements in return for a function. Although the list is a candidate for this feature, it may be better to use a dictionary, to be able to access the values with their key!


```

import statistics
def desc_stats(x):
    """Returns the mean and standard deviation of `x`"""
    return {"mean": statistics.mean(x),
            "std_dev": statistics.stdev(x)}

x = [1,3,2,6,4,1,8,9,3,2]
res = stat_des(x)

## Error in py_call_impl(callable, dots$args, dots$keywords):
##   NameError: name 'stat_des' is not defined
##
## Detailed traceback:
##   File "<string>", line 1, in <module>

print(res)

## 5

message = "The average value equals {} and the standard deviation is {}"
print(message.format(res["mean"], res["std_dev"]))

## Error in py_call_impl(callable, dots$args, dots$keywords):
##   TypeError: 'int' object is not subscriptable
##
## Detailed traceback:
##   File "<string>", line 1, in <module>

```

8.5 Exercise

1. Create a function named `sum_n_entiers` which returns the sum of the first integer n . Its only parameter will be n .
2. Using a loop, display the sum of the first 2 integers, then 3 first integers, then 4 first integers, etc. up to 10.

3. Create a function that from two points represented by pairs of coordinates (x_1, y_1) and (x_2, y_2) returns the Euclidean distance between these two points. Propose a second solution using a lambda function.

Chapter 9

Introduction to Numpy

This chapter is devoted to an important library for numerical calculations: NumPy (abbreviation of *Numerical Python*).

It is common practice to import NumPy by assigning it the alias `np`:

```
import numpy as np
```

9.1 Arrays

NumPy offers a popular data structure, arrays, on which calculations can be performed efficiently. Arrays are a useful structure for performing basic statistical operations as well as pseudo-random number generation.

The structure of the tables is similar to that of the lists, but the latter are slower to process and use more memory. The gain in processing speed of the ‘NumPy’ arrays comes from the fact that the data is stored in contiguous memory blocks, thus facilitating read access.

To be convinced, we can use the example of Pierre Navaro [given in his *notebook on NumPy*](#). Let’s create two lists of 1000 length each, with numbers drawn randomly using the `random()` function of the `random` module. Let’s divide each element in the first list by the element at the same position in the second line, then calculate the sum of these 1000 divisions. Then let’s look at the execution time using the magic function `%timeit`:

```

from random import random
from operator import truediv
l1 = [random() for i in range(1000)]
l2 = [random() for i in range(1000)]
# %timeit s = sum(map(truediv, l1, l2))

```

(uncomment the last line and test on a Jupyter Notebook)

Now, let's transform the two lists into NumPy tables with the `array()` method, and do the same calculation with a NumPy method:

```

a1 = np.array(l1)
a2 = np.array(l2)
# %timeit s = np.sum(a1/a2)

```

As can be seen by executing these codes in an IPython environment, the execution time is much faster with the NumPy methods for this calculation.

9.1.1 Creation

The creation of an array can be done with the `array()` method, from a list, as we just did:

```

list = [1,2,4]
table = np.array(liste)

```

```

## Error in py_call_impl(callable, dots$args, dots$keywords):
##   NameError: name 'liste' is not defined
##
## Detailed traceback:
##   File "<string>", line 1, in <module>

```

```

print(table)

```

```

## Error in py_call_impl(callable, dots$args, dots$keywords):
##   NameError: name 'table' is not defined
##
## Detailed traceback:

```

```
## File "<string>", line 1, in <module>
```

```
print(type(table))
```

```
## Error in py_call_impl(callable, dots$args, dots$keywords):  
  NameError: name 'table' is not defined  
##  
## Detailed traceback:  
## File "<string>", line 1, in <module>
```

If `array()` is provided with a list of nested lists of the same length, a multidimensional array will be created:

```
list_2 = [ [1,2,3], [4,5,6] ]  
table_2 = np.array(list_2)  
print(table_2)
```

```
## [[1 2 3]  
##  [4 5 6]]
```

```
print(type(table_2))
```

```
## <class 'numpy.ndarray'>
```

Tables can also be created from tuples:

```
tup = (1, 2, 3)  
table = np.array(tup)  
print(table)
```

```
## [1 2 3]
```

```
print(type(table))
```

```
## <class 'numpy.ndarray'>
```

An 1-dimension array can be casted to a 2-dimension array (if possible), by changing its `shape` attribute:

```
table = np.array([3, 2, 5, 1, 6, 5])
table.shape = (3,2)
print(table)

## [[3 2]
##   [5 1]
##   [6 5]]
```

9.1.1.1 Some Functions Generating array Objects

Some of the functions in NumPy produce pre-filled arrays. This is the case of the `zeros()` function. When given an integer value n , the `zeros()` function creates a one-dimensional array, with n 0 :

```
print( np.zeros(4) )

## [0. 0. 0. 0.]
```

The type of zeros (e. g. `int`, `int32`, `int64`, `float`, `float32`, `float64`, etc.) can be specified using the `dtype` argument:

```
print( np.zeros(4, dtype = "int") )

## [0 0 0 0]
```

More explanations on the types of data with NumPy are available on [the online documentation](#).

The type of the elements of an array is indicated via the argument `dtype`:

```
x = np.zeros(4, dtype = "int")
print(x, x.dtype)

## [0 0 0 0] int64
```

It is also possible to convert the type of elements into another type, using the `astype()` method:

```
y = x.astype("float")
print(x, x.dtype)
```

```
## [0 0 0 0] int64
```

```
print(y, y.dtype)
```

```
## [0. 0. 0. 0.] float64
```

When provided with a tuple longer than 1, `zeros()` creates a multidimensional array:

```
print( np.zeros((2, 3)) )
```

```
## [[0. 0. 0.]
##   [0. 0. 0.]]
```

```
print( np.zeros((2, 3, 4)) )
```

```
## [[[0. 0. 0. 0.]
##    [0. 0. 0. 0.]
##    [0. 0. 0. 0.]]
##
##   [[0. 0. 0. 0.]
##    [0. 0. 0. 0.]
##    [0. 0. 0. 0.]]]
```

The `empty()` function of Numpy also returns an array on the same principle as `zeros()`, but without initializing the values inside.

```
print( np.empty((2, 3), dtype = "int") )
```

```
## [[0 0 0]
##   [0 0 0]]
```

The `ones()` function of Numpy returns the same kind of arrays, with 1s in initialized values:

```
print( np.ones((2, 3), dtype = "float" )  
  
## [[1. 1. 1.]  
##  [1. 1. 1.]
```

To choose a specific value for initialization, you can use the `full()` function of Numpy:

```
print( np.full((2, 3), 10, dtype = "float" )  
  
## [[10. 10. 10.]  
##  [10. 10. 10.]
```

```
print( np.full((2, 3), np.inf) )  
  
## [[inf inf inf]  
##  [inf inf inf]]
```

The `eye()` function of Numpy creates a two-dimensional array in which all elements are initialized to zero, except those of the diagonal initialized to 1 :

```
print( np.eye(2, dtype="int64" )  
  
## [[1 0]  
##  [0 1]]
```

By modifying the keyword argument `k`, the diagonal can be shifted:

```
print( np.eye(3, k=-1) )  
  
## [[0. 0. 0.]  
##  [1. 0. 0.]  
##  [0. 1. 0.]
```


The `identity()` function of Numpy creates an identity matrix in the form of an array:

```
print( np.identity(3, dtype = "int") )
```

```
## [[1 0 0]
##  [0 1 0]
##  [0 0 1]]
```

The `arange()` function of Numpy allows to generate a sequence of numbers separated by a fixed interval, all stored in an array. The syntax is as follows:

```
np.arange( start, stop, step, dtype )
```

with `start` the start value, `stop` the finish value, `step` the step, *i.e.*, the spacing between the numbers in the sequence and `type` the type of numbers :

```
print( np.arange(5) )
```

```
## [0 1 2 3 4]
```

```
print( np.arange(2, 5) )
```

```
## [2 3 4]
```

```
print( np.arange(2, 10, 2) )
```

```
## [2 4 6 8]
```

9.1.2 Dimensions

To know the size of an array, the value of the attribute `ndim` can be displayed:

```
print("ndim tableau : ", table.ndim)
```

```
## ndim tableau : 2
```

```
print("ndim tableau_2 : ", table_2.ndim)
```

```
## ndim tableau_2 : 2
```

The number of elements in the array can be obtained by the `size` attribute or by the `size()` function of Numpy:

```
print("size tableau : ", table.size)
```

```
## size tableau : 6
```

```
print("size tableau_2 : ", table_2.size)
```

```
## size tableau_2 : 6
```

```
print("np.size(tableau) :", np.size(table))
```

```
## np.size(tableau) : 6
```

The `shape` attribute returns a tuple indicating the length for each dimension of the array:

```
print("size tableau : ", table.shape)
```

```
## size tableau : (3, 2)
```

```
print("size tableau_2 : ", table_2.shape)
```

```
## size tableau_2 : (2, 3)
```

9.1.3 Extracting Elements from an Array

Access to the elements of an array is done in the same way as for lists (see Section 3.1.1), using indexes. The syntax is as follows:

```
array[lower:upper:step]
```

with `lower` the lower boundary of the index range, `upper` the upper range, and `step` the spacing between the values.

- When `lower` is not specified, the first element (indexed 0) is considered as the value assigned to `lower`.
- When `upper` is not specified, the last element is considered as the value assigned to `upper`.
- When `step` is not specified, a step of 1 is assigned by default.

Let's take a quick look at some examples, using two objects: an array of dimension 1, and a second of dimension 2.

```
table_1 = np.arange(1,13)
table_2 = [ [1, 2, 3], [4, 5, 6], [7, 8, 9], [10, 11, 12]]
table_2 = np.array(table_2)
```

Access to the first element:

```
message = "table_{0}[0] : {1} (type : {2})"
print(message.format(0, table_1[0], type(table_1[0])))
```

```
## table_0[0] : 1 (type : <class 'numpy.int64'>)
```

```
print(message.format(1, table_2[0], type(table_2[0])))
```

```
## table_1[0] : [1 2 3] (type : <class 'numpy.ndarray'>)
```

Access to the elements can be done from the end:

```
print("tableau_1[-1] : ", tableau_1[-1]) # last element
```

```
## Error in py_call_impl(callable, dots$args, dots$keywords):
  NameError: name 'tableau_1' is not defined
```

```
##
## Detailed traceback:
##   File "<string>", line 1, in <module>
```

```
print("tableau_2[-1] :", tableau_2[-1]) # last element
```

```
## Error in py_call_impl(callable, dots$args, dots$keywords):
##   NameError: name 'tableau_2' is not defined
##
## Detailed traceback:
##   File "<string>", line 1, in <module>
```

Slicing is possible:

```
# the elements from the 2nd (not included) to the 4th
print("Slice Table 1 : \n", table_1[2:4])
```

```
## Slice Table 1 :
##   [3 4]
```

```
print("Sclie Table 2 : \n", table_2[2:4])
```

```
## Sclie Table 2 :
##   [[ 7  8  9]
##    [10 11 12]]
```

For two-dimensional arrays, the elements can be accessed in the following ways:

```
# Within the 3rd element, access the 1st element
print(table_2[2][0])
```

```
## 7
```

```
print(table_2[2,0])
```

```
## 7
```

To extract columns from an array with two entries:

```
print("Second column: \n", table_2[:, [1]])
```

```
## Second column:  
## [[ 2]  
## [ 5]  
## [ 8]  
## [11]]
```

```
print("Second and third columns: \n", table_2[:, [1,2]])
```

```
## Second and third columns:  
## [[ 2  3]  
## [ 5  6]  
## [ 8  9]  
## [11 12]]
```

For this last instruction, we specify with the first argument not filled in (before the two points) that we want all the elements of the first dimension, then, with the comma, we indicate that we look inside each element of the first dimension, and that we want the values at positions 1 and 2 (therefore the elements of columns 2 and 3).

To extract only some elements from a 1-dimensional array, we can specify the indices of the elements to be recovered:

```
print("2nd and 4th elements: \n", table_2[[1,3]])
```

```
## 2nd and 4th elements:  
## [[ 4  5  6]  
## [10 11 12]]
```

9.1.3.1 Extraction Using Boolean

To extract or not elements from a table, you can use Boolean tables as masks. The idea is to provide a boolean array (a mask) of the same size as the one for which you want to extract elements under certain conditions. When the value of the Boolean in the mask is set to `True`, the corresponding element of the array is returned; otherwise, it is not.

```
table = np.array([0, 3, 2, 5, 1, 4])
res = table[[True, False, True, False, True, True]]
print(res)
```

```
## [0 2 1 4]
```

Only the elements in positions 1, 3, 5 and 6 were returned.

In practice, the mask is only very rarely created by the user, but rather comes from a logical instruction applied to the interest table. For example, in our table, we can first create a mask to identify even elements:

```
mask = table % 2 == 0
print(mask)
```

```
## [ True False  True False False  True]
```

```
print(type(mask))
```

```
## <class 'numpy.ndarray'>
```

Once this mask is created, it can be applied to the array to extract only those elements for which the corresponding value in the mask is `True`:

```
print(table[mask])
```

```
## [0 2 4]
```

9.1.4 Modification

To replace the values in an array, equal sign (=) can be used:

```
table = np.array([ [1, 2, 3], [4, 5, 6], [7, 8, 9], [10, 11, 12]])
table[0] = [11, 22, 33]
print(table)
```

```
## [[11 22 33]
##  [ 4  5  6]
##  [ 7  8  9]
##  [10 11 12]]
```

If a scalar is provided during replacement, the value will be repeated for all elements of the dimension :

```
table[0] = 100
print(table)
```

```
## [[100 100 100]
##  [  4   5   6]
##  [  7   8   9]
##  [ 10  11  12]]
```

Same idea with a slicing:

```
table[0:2] = 100
print(table)
```

```
## [[100 100 100]
##  [100 100 100]
##  [  7   8   9]
##  [ 10  11  12]]
```

In fact, a breakdown with just the two points without specifying the start and end parameters of the breakdown followed by an equal sign and a number replaces all the values in the table with this number:

```
table[:] = 0
print(table)

## [[0 0 0]
##  [0 0 0]
##  [0 0 0]
##  [0 0 0]]
```

9.1.4.1 Inserting Elements

To add elements, we use the `append()` function of NumPy. Note that calling this function does not change the object to which the values are added. If we want the changes to be made to this object, we must overwrite it:

```
t_1 = np.array([1,3,5])
print("t_1 : ", t_1)
```

```
## t_1 :  [1 3 5]
```

```
t_1 = np.append(t_1, 1)
print("t_1 after the insertion: ", t_1)
```

```
## t_1 after the insertion:  [1 3 5 1]
```

To add a column to a two-dimensional table:

```
t_2 = np.array([[1,2,3], [5,6,7]])
print("t_2 : \n", t_2)
```

```
## t_2 :
##  [[1 2 3]
##  [5 6 7]]
```



```
add_col_t_2 = np.array([[4], [8]])
t_2 = np.append(t_2, add_col_t_2, axis = 1)
print("t_2 after the insertion: \n", t_2)
```

```
## t_2 after the insertion:
## [[1 2 3 4]
## [5 6 7 8]]
```

To add a line, we use the `vstack()` function of Numpy:

```
ajout_ligne_t_2 = np.array([10, 11, 12, 13])
t_2 = np.vstack([t_2, ajout_ligne_t_2])
print("t_2 après ajout ligne : \n", t_2)
```

```
## t_2 après ajout ligne :
## [[ 1  2  3  4]
## [ 5  6  7  8]
## [10 11 12 13]]
```

9.1.4.2 Deleting / Removing Elements

To delete elements, we can use the `delete()` function of NumPy:

```
print("t_1 : ", t_1)
# Remove the last element
```

```
## t_1 : [1 3 5 1]
```

```
np.delete(t_1, (-1))
```

```
## array([1, 3, 5])
```

Note: for the deletion to be effective, the result of `np.delete()` is assigned to the object.

To delete multiple items:

```
print("t_1 : ", t_1)
# Remove the first and second elements:
```

```
## t_1 : [1 3 5 1]
```

```
t_1 = np.delete(t_1, ([0, 2]))
print(t_1)
```

```
## [3 1]
```

To delete a column from a two-dimensional table:

```
print("t_2 : ", t_2)
# Remove the last column:
```

```
## t_2 : [[ 1  2  3  4]
##       [ 5  6  7  8]
##       [10 11 12 13]]
```

```
np.delete(t_2, (0), axis=1)
```

```
## array([[ 2,  3,  4],
##        [ 6,  7,  8],
##        [11, 12, 13]])
```

Delete multiple columns:

```
print("t_2 : ", t_2)
# Remove the first and third columns:
```

```
## t_2 : [[ 1  2  3  4]
##       [ 5  6  7  8]
##       [10 11 12 13]]
```

```
np.delete(t_2, ([0,2]), axis=1)
```

```
## array([[ 2,  4],  
##        [ 6,  8],  
##        [11, 13]])
```

And to delete a row:

```
print("t_2 : ", t_2)  
# Remove the first line:
```

```
## t_2 :  [[ 1  2  3  4]  
##        [ 5  6  7  8]  
##        [10 11 12 13]]
```

```
np.delete(t_2, (0), axis=0)
```

```
## array([[ 5,  6,  7,  8],  
##        [10, 11, 12, 13]])
```

Delete multiple lines:

```
print("t_2 : ", t_2)  
# Remove the first and third lines:
```

```
## t_2 :  [[ 1  2  3  4]  
##        [ 5  6  7  8]  
##        [10 11 12 13]]
```

```
np.delete(t_2, ([0,2]), axis=0)
```

```
## array([[5, 6, 7, 8]])
```

9.1.5 Copyi of an Array

Copying an array, as with lists (c.f. Section 3.1.4), should not be done with the equal symbol (=). Let's see why.

```
table_1 = np.array([1, 2, 3])
table_2 = table_1
```

Let's modify the first element of `table_2`, and observe the content of `table_2` and `table_1`:

```
table_2[0] = 0
```

```
## Error in py_call_impl(callable, dots$args, dots$keywords):
##   NameError: name 'table_2' is not defined
##
## Detailed traceback:
##   File "<string>", line 1, in <module>
```

```
print("Table 1: \n", table_1)
```

```
## Table 1:
##  [1 2 3]
```

```
print("Table 2: \n", table_2)
```

```
## Table 2:
##  [1 2 3]
```

As can be seen, using the equal sign simply created a reference and not a copy.

There are several ways to copy an array. Among them, the use of the `np.array()` function:

```
table_1 = np.array([1, 2, 3])
table_2 = np.array(table_1)
table_2[0] = 0
print("table_1 : ", table_1)
```

```
## table_1 : [1 2 3]
```

```
print("table_2 : ", table_2)
```

```
## table_2 : [0 2 3]
```

The `copy()` method can also be used:

```
table_1 = np.array([1, 2, 3])
```

```
table_2 = table_1.copy()
```

```
table_2[0] = 0
```

```
print("table_1 : ", table_1)
```

```
## table_1 : [1 2 3]
```

```
print("table_2 : ", table_2)
```

```
## table_2 : [0 2 3]
```

It can be noted that when a slicing is made, a new object is created, not a reference:

```
table_1 = np.array([1, 2, 3, 4])
```

```
table_2 = table_1[:2]
```

```
table_2[0] = 0
```

```
print("table_1 : ", table_1)
```

```
## table_1 : [0 2 3 4]
```

```
print("table_2 : ", table_2)
```

```
## table_2 : [0 2]
```

9.1.6 Sorting

The NumPy library provides a function to sort the tables, `sort()`:

```
table = np.array([3, 2, 5, 1, 6, 5])
print("Sorted Table: ", np.sort(table))

## Sorted Table:  [1 2 3 5 5 6]
```

```
print("Table: ", table)

## Table:  [3 2 5 1 6 5]
```

As we can see, the `sort()` function of NumPy offers a view: the table is not modified, which is not the case if we use the `sort()` method:

```
table = np.array([3, 2, 5, 1, 6, 5])
table.sort()
print("The array was modified: ", table)

## The array was modified:  [1 2 3 5 5 6]
```

9.1.7 Transposition

To obtain the transposition of an array, the attribute `T` can be used. It should be noted that you get a view of the object: the object is not changed.

```
table = np.array([3, 2, 5, 1, 6, 5])
table.shape = (3,2)
print("Array: \n", table)

## Array:
##  [[3 2]
##   [5 1]
##   [6 5]]
```

```
print("Transposed Array: \n", table.T)
```

```
## Transposed Array:  
##  [[3 5 6]  
##  [2 1 5]]
```

The `transpose()` function of NumPy can also be used:

```
print(np.transpose(table))
```

```
##  [[3 5 6]  
##  [2 1 5]]
```

Be careful, if a name is assigned to the transpose, either by using the attribute `T` or the method `np.transpose()`, it creates a reference, not a copy of an element...

```
table_transpose = np.transpose(table)  
table_transpose[0,0] = 99  
print("Array: \n", table)
```

```
## Array:  
##  [[99  2]  
##  [ 5  1]  
##  [ 6  5]]
```

```
print("Transpose of the Array: \n", table_transpose)
```

```
## Transpose of the Array:  
##  [[99  5  6]  
##  [ 2  1  5]]
```

To know if an array is a view or not, we can display the `base` attribute, which returns `None` if it is not the case:

```
print("table: ", table.base)
```

```
## table: None

print("table_transpose : ", table_transpose.base)

## table_transpose :  [[99  2]
## [ 5  1]
## [ 6  5]]
```

9.1.8 Operations on Arrays

It is possible to use operators on the tables. Their effect requires some explanation.

9.1.8.1 + and - Operators

When the operator + (-) is used between two tables of the same size, an addition (subtraction) is performed:

```
t_1 = np.array([1, 2, 3, 4])
t_2 = np.array([5, 6, 7, 8])
t_3 = np.array([[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12]])
t_4 = np.array([[13, 14, 15, 16], [17, 18, 19, 20], [21, 22, 23, 24]])
t_1 + t_2

## array([ 6,  8, 10, 12])
```

```
t_3 + t_4

## array([[14, 16, 18, 20],
##        [22, 24, 26, 28],
##        [30, 32, 34, 36]])
```

```
t_1 - t_2
```



```
## array([-4, -4, -4, -4])
```

When the operator + (-) is used between a scalar and an array, the scalar is added (subtracted) to all elements of the array:

```
print("t_1 + 3 : \n", t_1 + 3)
```

```
## t_1 + 3 :  
## [4 5 6 7]
```

```
print("t_1 + 3. : \n", t_1 + 3.)
```

```
## t_1 + 3. :  
## [4. 5. 6. 7.]
```

```
print("t_3 + 3 : \n", t_3 + 3)
```

```
## t_3 + 3 :  
## [[ 4  5  6  7]  
## [ 8  9 10 11]  
## [12 13 14 15]]
```

```
print("t_3 - 3 : \n", t_3 - 3)
```

```
## t_3 - 3 :  
## [[-2 -1  0  1]  
## [ 2  3  4  5]  
## [ 6  7  8  9]]
```

9.1.8.2 * and / Operators

When the operator * (/) is used between two tables of the same size, a multiplication (division) forward term is performed:

```
t_1 * t_2
```

```
## array([ 5, 12, 21, 32])
```

```
t_3 * t_4
```

```
## array([[ 13,  28,  45,  64],
##        [ 85, 108, 133, 160],
##        [189, 220, 253, 288]])
```

```
t_3 / t_4
```

```
## array([[0.07692308, 0.14285714, 0.2          , 0.25          ],
##        [0.29411765, 0.33333333, 0.36842105, 0.4          ],
##        [0.42857143, 0.45454545, 0.47826087, 0.5          ]])
```

When the operator `*` (`/`) is used between a scalar and an array, all the elements of the array are multiplied (divided) by this scalar :

```
print("t_1 * 3 : \n", t_1 * 3)
```

```
## t_1 * 3 :
## [ 3  6  9 12]
```

```
print("t_1 / 3 : \n", t_1 / 3)
```

```
## t_1 / 3 :
## [0.33333333 0.66666667 1.          1.33333333]
```

9.1.8.3 Power

It is also possible to raise each number in a table to a given power:

```
print("t_1 ** 3 : \n", t_1 ** 3)

## t_1 ** 3 :
## [ 1  8 27 64]
```

9.1.8.4 Operations on Matrices

In addition to the term-by-term operations/subtraction/multiplication/division (or on a scalar), it is possible to perform some calculations on two-dimensional tables (matrices).

We've already seen the tranposition of a matrix in Section 9.1.7.

To perform a matrix product, NumPy provides the function `dot()`:

```
np.dot(t_3, t_4.T)

## array([[150, 190, 230],
##        [382, 486, 590],
##        [614, 782, 950]])
```

We have to make sure that the matrices are compatible, otherwise, an error will be returned:

```
np.dot(t_3, t_4)

## Error in py_call_impl(callable, dots$args, dots$keywords):
##   ValueError: shapes (3,4) and (3,4) not aligned: 4 (dim 1)
##     != 3 (dim 0)
##
## Detailed traceback:
##   File "<string>", line 1, in <module>
```

The matrix product can also be obtained using the operator `@`:

```
t_3 @ t_4.T
## array([[150, 190, 230],
##        [382, 486, 590],
##        [614, 782, 950]])
```

The product of a vector with a matrix is also possible:

```
np.dot(t_1, t_3.T)
## array([ 30,  70, 110])
```

9.1.9 Logical Operators

To perform logical tests on the elements of a table, NumPy offers functions, listed in Table 9.1. The result returned by applying these functions is a Boolean array.

Table 9.1: Logical Functions

Code	Description
<code>greater()</code>	Greater than
<code>greater_equal()</code>	Greater than or equal to
<code>less()</code>	Lower than
<code>less_equal()</code>	Lower than or equal to
<code>equal()</code>	Equal to
<code>not_equal()</code>	Different from
<code>logical_and()</code>	Logical And
<code>logical_or()</code>	Logical Or
<code>logical_xor()</code>	Logical XOR

For example, to obtain the elements of `t` between 10 and 20 (included):

```
t = np.array([[1, 10, 3, 24], [9, 12, 40, 2], [0, 7, 2, 14]])
mask = np.logical_and(t <= 20, t >= 10)
print("mask: \n", mask)
```

```
## mask:
## [[False True False False]
## [False True False False]
## [False False False True]]

print("the elements of t between 10 and 20: \n",
      t[mask])

## the elements of t between 10 and 20:
## [10 12 14]
```

9.1.10 Some Constants

NumPy provides some constants, some of which are shown in Table 9.2.

Table 9.2: Formatting Codes

Code	Description
<code>np.inf</code>	Infinity (we get $-\infty$ by writing <code>-np.inf</code> or <code>np.NINF</code>)
<code>np.nan</code>	Representation as a floating point number of Not a Number
<code>np.e</code>	Euler constant (e)
<code>np.euler_gamma</code>	Euler-Mascheroni constant (γ)
<code>np.pi</code>	Pi (π)

We can note the presence of the value `NaN`, which is a special value among the floating point numbers. The behavior of this constant is special.

When we add, subtract, multiply or divide a number by this `NaN` value, we obtain `NaN`:

```
print("Addition : ", np.nan + 1)

## Addition : nan
```

```
print("Substraction : ", np.nan - 1)
```

```
## Substraction : nan
```

```
print("Multiplication : ", np.nan + 1)
```

```
## Multiplication : nan
```

```
print("Division : ", np.nan / 1)
```

```
## Division : nan
```

9.1.11 Universal functions

Universal functions (*ufunc* for *universal functions*) are functions that can be applied term-by-term to the elements of an array. There are two types of universal functions: unannic functions, which perform an operation on a single operand, and binary functions, which perform an operation on two operands.

Among the *ufuncs* are arithmetic operations (addition, multiplication, power, absolute value, etc.) and common mathematical functions (trigonometric, exponential, logarithmic functions, etc.). Table 9.3 lists some universal functions, while Table 9.4 lists some universal binary functions.

Table 9.3: Unary Universal Function

Code	Description
<code>negative(x)</code>	Opposite elements of elements of <code>x</code>
<code>absolute(x)</code>	Absolute values of the elements of <code>x</code>
<code>sign(x)</code>	Signs of the elements of <code>x</code> (0, 1 or -1)
<code>rint(x)</code>	Ronded value of <code>x</code> to the nearest integer
<code>floor(x)</code>	Truncated value of <code>x</code> to the next smaller integer
<code>ceil(x)</code>	Truncated value of <code>x</code> to the next larger integer
<code>sqrt(x)</code>	Square root of <code>x</code>

Code	Description
<code>square(x)</code>	Squared value of <code>x</code>
<code>sin(x)</code> , <code>cos(x)</code> , <code>tan(x)</code>	Sine (cosine, and tangent) of the elements of <code>x</code>
<code>sinh(x)</code> , <code>cosh(x)</code> , <code>tanh(x)</code>	Hyperbolic sine (cosine, and tangent) of the elements of <code>x</code>
<code>arcsin(x)</code> , <code>arccos(x)</code> , <code>arctan(x)</code>	Arc-sine (arc-cosine, and arc-tangent) of <code>x</code>
<code>arcsinh(x)</code> , <code>arccosh(x)</code> , <code>arctanh(x)</code>	Hyperbolic arc-sine (arc-cosine, and arc-tangent) of the elements of <code>x</code>
<code>hypoth(x,y)</code>	Hypotenuse $\sqrt{x^2 + y^2}$
<code>degrees(x)</code>	Conversion of the angles values of <code>x</code> from radians to degrees
<code>radians(x)</code>	Conversion of the angles values of <code>x</code> from degrees to radians
<code>exp(x)</code>	Exponential of the values of <code>x</code>
<code>expm1(x)</code>	$e^x - 1$
<code>log(x)</code>	Natural logarithm of the elements of <code>x</code>
<code>log10(x)</code>	Logarithm of the elements of <code>x</code> in base 10
<code>log2(x)</code>	Logarithm of the elements of <code>x</code> in base 2
<code>log1p(x)</code>	$\ln(1 + x)$
<code>exp2(x)</code>	2^x
<code>isnan(x)</code>	Boolean table indicating <code>True</code> for the elements <code>NaN</code>
<code>isfinite(x)</code>	Boolean table indicating <code>True</code> for non-infinite and non- <code>NaN</code> elements
<code>isinf(x)</code>	Boolean array indicating <code>True</code> for infinite elements

Table 9.4: Binary Universal Functions

Code	Description
<code>add(x,y)</code>	Term by term addition of the elements of <code>x</code> and <code>y</code>
<code>subtract(x,y)</code>	Term by term subtraction of the elements of <code>x</code> and <code>y</code>
<code>multiply(x,y)</code>	Term by term multiplication of the elements of <code>x</code> and <code>y</code>
<code>divide(x,y)</code>	Term by term division of the elements of <code>x</code> and <code>y</code>
<code>floor_divide(x,y)</code>	Largest integer smaller or equal to the division of the elements of <code>x</code> and <code>y</code>

Code	Description
<code>power(x,y)</code>	Elements of <code>x</code> to the power of the elements of <code>y</code>
<code>mod(x,y)</code>	Remainder of Euclidean term by term divisions of the elements of <code>x</code> by the elements of <code>y</code>
<code>round(x,n)</code>	Rounded value of the elements of <code>x</code> up to <code>n</code> digits
<code>arctan2(x,y)</code>	Polar angles of <code>x</code> and <code>y</code>

To use these functions, proceed as in the following example:

```
t_1 = np.array([[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12]])
t_2 = np.array([[13, 14, 15, 16], [17, 18, 19, 20], [21, 22, 23, 24]])
np.log(t_1) # Natural Logarithm
```

```
## array([[0.          , 0.69314718, 1.09861229, 1.38629436],
##        [1.60943791, 1.79175947, 1.94591015, 2.07944154],
##        [2.19722458, 2.30258509, 2.39789527, 2.48490665]])
```

```
np.subtract(t_1, t_2) # Substraction of the elements of `t_1` by those of `t_2`
```

```
## array([[ -12,  -12,  -12,  -12],
##        [ -12,  -12,  -12,  -12],
##        [ -12,  -12,  -12,  -12]])
```

9.1.12 Mathematical and Statistical Methods and Functions

NumPy provides many methods to calculate statistics on all array values, or on one of the array axes (for example on the equivalent of rows or columns in two-dimensional arrays). Some of them are reported in Table 9.5.

Table 9.5: Mathematical and Statistical Methods

Code	Description
<code>sum()</code>	Returns the sum of the elements
<code>prod()</code>	Returns the product of the elements

Code	Description
<code>cumsum()</code>	Returns the cumulative sum of the elements
<code>cumprod()</code>	Returns the cumulative product of the elements
<code>mean()</code>	Returns the average
<code>var()</code>	Returns the variance
<code>std()</code>	Returns the standard error
<code>min()</code>	Returns the minimum value
<code>max()</code>	Returns the maximum value
<code>argmin()</code>	Returns the index of the first element with the lowest value
<code>argmax()</code>	Returns the index of the first element with the largest value

Let's give an example of the use of these methods:

```
t_1 = np.array([[1, 2, 3, 4], [-1, 6, 7, 8], [9, -1, 11, 12]])
print("t_1 : \n", t_1)
```

```
## t_1 :
## [[ 1  2  3  4]
## [-1  6  7  8]
## [ 9 -1 11 12]]
```

```
print("Sum of the elements: ", t_1.sum())
```

```
## Sum of the elements: 61
```

```
print("Covariance of the elements: ", t_1.var())
```

```
## Covariance of the elements: 18.076388888888889
```

To apply these functions to a given axis, we modify the value of the parameter `axis`:

```
print("Sum per column: ", t_1.sum(axis=0))
```

```
## Sum per column: [ 9  7 21 24]
```

```
print("Sum per row: ", t_1.sum(axis=1))
```

```
## Sum per row:  [10 20 31]
```

NumPy also offers some statistically specific functions, some of which are listed in Table 9.6.

Table 9.6: Statistical Functions

Code	Description
<code>sum(x)</code> , <code>nansum(x)</code>	Sum of the elements of <code>x</code> (<code>nansum(x)</code> does not take into account NaN values)
<code>mean(x)</code> , <code>nanmean()</code>	Average of <code>x</code>
<code>median(x)</code> , <code>nanmedian()</code>	Median of <code>x</code>
<code>average(x)</code>	Average of <code>x</code> (possibility to use weights using the <code>weight</code> parameter)
<code>min(x)</code> , <code>nanmin()</code>	Minimum of <code>x</code>
<code>max(x)</code> , <code>nanmax()</code>	Maximum of <code>x</code>
<code>percentile(x,p)</code> , <code>nanpercentile(n,p)</code>	P-th percentile of <code>x</code>
<code>var(x)</code> , <code>nanvar(x)</code>	Variance of <code>x</code>
<code>std(x)</code> , <code>nanstd()</code>	Standard-deviation of <code>x</code>
<code>cov(x)</code>	Covariance of <code>x</code>
<code>corrcoef(x)</code>	Correlation coefficient

To use the statistical functions:

```
t_1 = np.array([[1, 2, 3, 4], [-1, 6, 7, 8], [9, -1, 11, 12]])
print("t_1 : \n", t_1)
```

```
## t_1 :
##  [[ 1  2  3  4]
##   [-1  6  7  8]
```

```
## [ 9 -1 11 12]]
```

```
print("Variance: ", np.var(t_1))
```

```
## Variance: 18.076388888888889
```

If the array has NaN values, for example, to calculate the sum, if `sum()` is used, the result will be NaN. To ignore the values NaN, we use a specific function (here, `nansum()`):

```
t_1 = np.array([[1, 2, np.NaN, 4], [-1, 6, 7, 8], [9, -1, 11, 12]])
print("Sum: ", np.sum(t_1))
```

```
## Sum: nan
```

```
print("Sum ignoring NaN values: ", np.nansum(t_1))
```

```
## Sum ignoring NaN values: 58.0
```

To calculate a weighted average (let's consider a vector):

```
v_1 = np.array([1, 1, 4, 2])
w = np.array([1, 1, .5, 1])
print("Weighted average: ", np.average(v_1, weights=w))
```

```
## Weighted average: 1.7142857142857142
```

9.2 Generation of Pseudo-random Numbers

The generation of pseudo-random numbers is allowed by the `random` module of Numpy. The reader interested in the more statistical aspects will be able to find more concepts covered in the `stats` sub-module of SciPy.

```
from numpy import random
```

Table 9.7 lists some functions that allow to draw numbers in a pseudo-random way with the `random` module of `Numpy` (by evaluating `random`, we get an exhaustive list).

Table 9.7: Some Functions for Pseudo-random Number Generation

Code	Description
<code>rand(size)</code>	Drawing <code>size</code> obs. from a Uniform distribution $[0, 1]$
<code>uniform(a,b,size)</code>	Drawing <code>size</code> obs. from a Uniform distribution $[a; b]$
<code>randint(a,b,size)</code>	Drawing <code>size</code> obs. from a Uniform distribution $[a; b[$
<code>randn(size)</code>	Drawing <code>size</code> obs. from a Normal distribution $\mathcal{N}(0, 1)$
<code>normal(mu, std, size)</code>	Drawing <code>size</code> obs. from a Normal distribution with <code>mu</code> mean and standard error <code>std</code>
<code>binomial(size, n, p)</code>	Drawing <code>size</code> obs. from a Binomial distribution $\mathcal{B}in(n, p)$
<code>beta(alpha, beta, size)</code>	Drawing <code>size</code> obs. from a Beta distribution $Beta(\alpha, \beta)$
<code>poisson(lambda, size)</code>	Drawing <code>size</code> obs. from a Poisson distribution $\mathcal{P}(\lambda)$
<code>standard_t(df, size)</code>	Drawing <code>size</code> obs. from a Student distribution $\mathcal{S}t(df)$

Here is an example of generating pseudo random numbers according to a Gaussian distribution:

```
x = np.random.normal(size=10)
print(x)
```

```
## [ 1.2045678  0.62947989 -1.11421506  1.48626408
     -1.15608001 -1.07159849
##    0.77041608 -0.21934874 -0.5421896  0.68204223]
```

A multidimensional array can be generated. For example, a two-dimensional array, in which the first dimension contains 10 elements, each containing 4 random draws according to a $\mathcal{N}(0,1)$:

```
x = np.random.randn(10, 4)
print(x)

## [[ 0.60530527 -0.86279092  0.31726444 -1.25625927]
## [ 0.63417263  1.84491222  2.0506628  -0.722184  ]
## [-0.98217855  0.48028984  1.78485585 -0.37294828]
## [ 0.05824012 -0.4965277  -0.09092745  0.89620369]
## [-0.85446036 -1.06870108  0.85759164 -1.28842648]
## [ 0.28288546 -0.43186816  0.93785772  0.54925557]
## [ 1.52594688  0.69341129 -0.06140043  1.15137295]
## [-0.11861859 -1.19806045 -0.77662476  1.42544774]
## [ 0.67120661  0.48387841 -0.45160398 -0.25340085]
## [-2.01401797  1.71397631  0.52017544  1.72642461]]
```

The generation of numbers is based on a *seed*, i.e. a number that initiates the generator of pseudo random numbers. It is possible to fix this seed, so that reproducible results can be obtained, for example. To do this, we can use the `seed()` method, to which we indicate a value as a parameter :

```
np.random.seed(1234)
x = np.random.normal(size=10)
print(x)

## [ 0.47143516 -1.19097569  1.43270697 -0.3126519
##    -0.72058873  0.88716294
##    0.85958841 -0.6365235  0.01569637 -2.24268495]
```

By fixing the seed again, one will obtain exactly the same draft:

```
np.random.seed(1234)
x = np.random.normal(size=10)
print(x)

## [ 0.47143516 -1.19097569  1.43270697 -0.3126519
##    -0.72058873  0.88716294
##    0.85958841 -0.6365235  0.01569637 -2.24268495]
```

To avoid affecting the global environment by the random seed, the `RandomState` method of the `random` sub-module of NumPy can be used:

```
from numpy.random import RandomState
rs = RandomState(123)
x = rs.normal(10)
print(x)
```

```
## 8.914369396699438
```

In addition, the `switching()` function of the `random` sub-module allows a random switch:

```
x = np.arange(10)
y = np.random.permutation(x)
print("x : ", x)
```

```
## x : [0 1 2 3 4 5 6 7 8 9]
```

```
print("y : ", y)
```

```
## y : [9 7 4 3 8 2 6 1 0 5]
```

The `shuffle()` function of the `random` submodule allows to perform a random permutation of the elements :

```
x = np.arange(10)
print("x avant permutation : ", x)
```

```
## x avant permutation : [0 1 2 3 4 5 6 7 8 9]
```

```
np.random.permutation(x)
```

```
## array([7, 5, 4, 1, 0, 8, 3, 9, 6, 2])
```

```
print("x après permutation : ", x)

## x après permutation : [0 1 2 3 4 5 6 7 8 9]
```

9.3 Exercise

First exercise

Consider the following vector: $x = [1 \ 2 \ 3 \ 4 \ 5]$

1. Create this vector using an array called **x**.
2. Display the type of **x** and its length.
3. Extract the first element, then do the same with the last one.
4. Extract the first three elements and store them in a vector called **a**.
5. Extract the 1st, 2nd and 5th elements of the vector (be careful with the positions); store them in a vector called **b**.
6. Add the number 10 to the vector **x**, then multiply the result by 2.
7. Add **a** and **b**, comment on the result.
8. Make the following addition: **x+a**; comment on the result, then look at the result of **a+x**.
9. Multiply the vector by the scalar 'c' which will be set to 2.
10. Multiply **a** and **b**; comment on the result.
11. Perform the following multiplication: **x*a**; comment on the results.
12. Retrieve the positions of the multiples of 2 and store them in a vector called **ind**, then store only the multiples of 2 of **x** in a vector called **mult_2**.
13. Display the elements of **x** that are multiples of 3 *and* multiples of 2.
14. Display the elements of **x** that are multiples of 3 *or* multiples of 2.
15. Calculate the sum of the elements of **x**.
16. Replace the first element of **x** with a 4.
17. Replace the first element of **x** with the value NaN, then calculate the sum of the elements of **x**.
18. Delete the vector **x**.

Second exercise

1. Create the following matrix: $A = \begin{bmatrix} -3 & 5 & 6 \\ -1 & 2 & 2 \\ 1 & -1 & -1 \end{bmatrix}$.

2. Display the size of **A**, its number of columns, its number of rows and its length.
3. Extract the second column from **A**, then the first row.
4. Extract the element in the third position in the first line.
5. Extract the submatrix of dimension 2×2 from the lower corner of **A**, *i. e.*,
 $\begin{bmatrix} 2 & 2 & -1 & -1 \end{bmatrix}$.
6. Calculate the sum of the columns and then the rows of **A**.
7. Display the diagonal of **A**.
8. Add the vector $\begin{bmatrix} 1 & 2 & 3 \end{bmatrix}^\top$ to the right of the matrix **A** and store the result in an object called **B**.
9. Remove the fourth vector from **B**.
10. Remove the first and third lines from **B**.
11. Add scalar 10 to **A**.
12. Add the vector $\begin{bmatrix} 1 & 2 & 3 \end{bmatrix}^\top$ to **A**.
13. Add the identity matrix I_3 to **A**.
14. Divide all the elements of the matrix **A** by 2.
15. Multiply the matrix **A** by the line vector $\begin{bmatrix} 1 & 2 & 3 \end{bmatrix}^\top$.
16. Display the transposition of **A**.
17. Perform the product with transposition $A^\top A$.

->

Chapter 10

References

Briggs, Jason R. 2013. *Python for Kids: A Playful Introduction to Programming*. no starch press.

Grus, Joel. 2015. *Data Science from Scratch: First Principles with Python*. " O'Reilly Media, Inc."

McKinney, Wes. 2017. *Python for Data Analysis: Data Wrangling with Pandas, Numpy, and Ipython (2nd Edition)*. " O'Reilly Media, Inc."

Navaro, Pierre. 2018. "Python Notebooks." <https://github.com/pnavaro/python-notebooks>.

VanderPlas, Jake. 2016. *Python Data Science Handbook: Essential Tools for Working with Data*. " O'Reilly Media, Inc."