Python for economists

Ewen Gallic

September 2019

Contents

List of Tables

List of Figures

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'Economie 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.

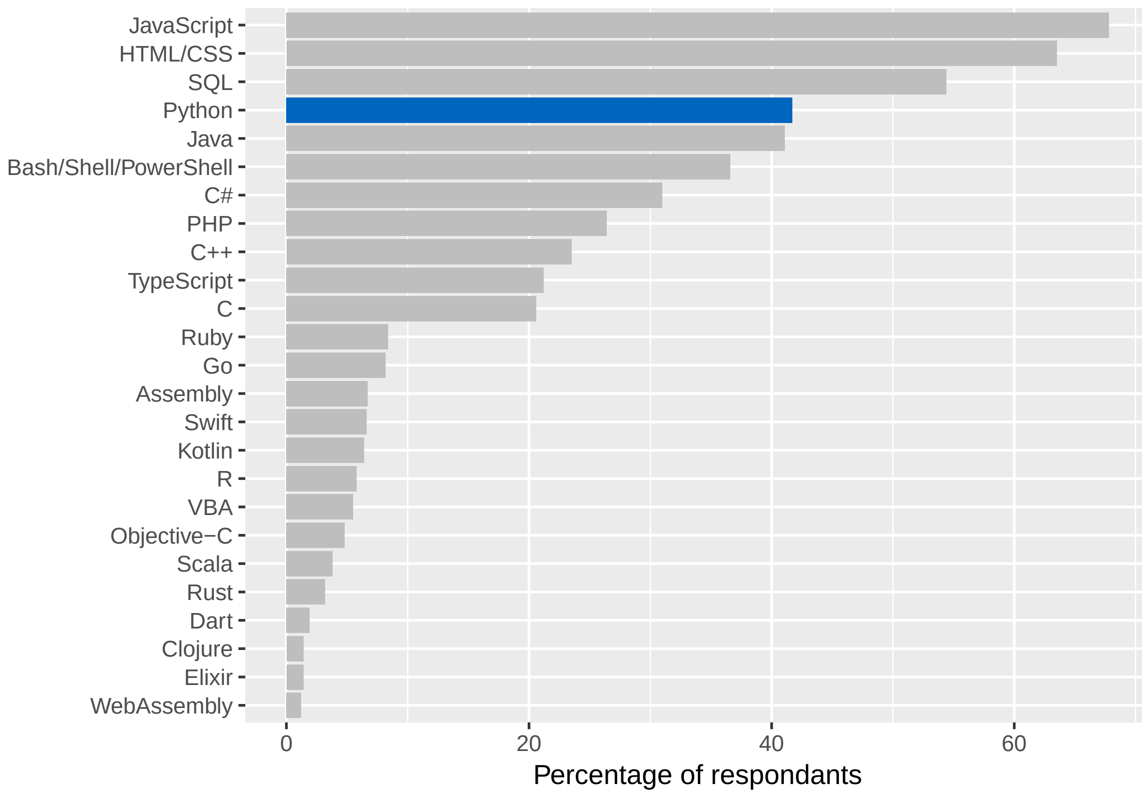


Figure 1.1: Programming, Scripting, and Markup Languages.

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

(Pycharm,...)

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 :

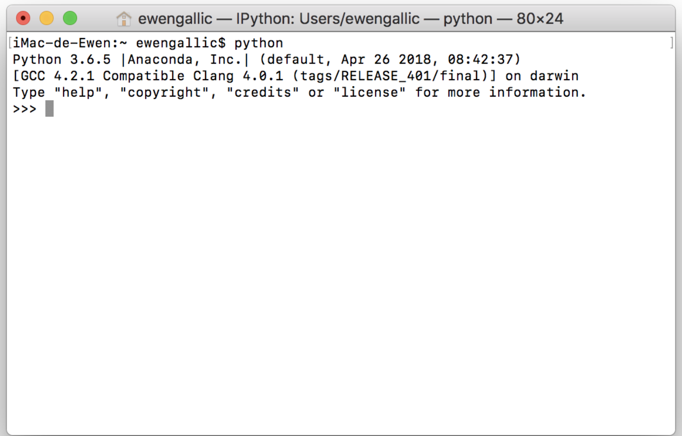


Figure 1.2: Python in a terminal.

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 'ENTEREE' 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.

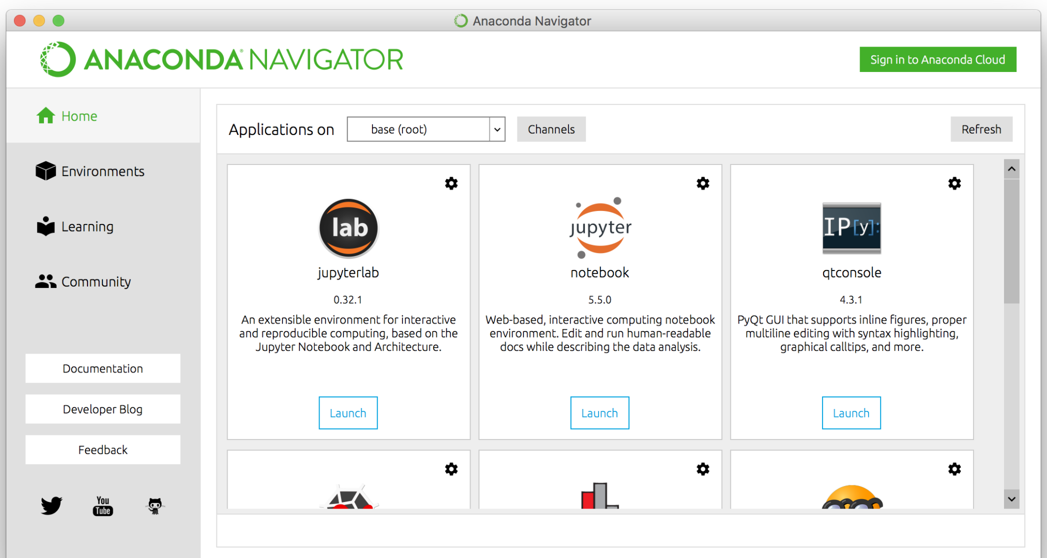


Figure 1.3: Anaconda's home window.

The IPython console, once launched, looks as follows:

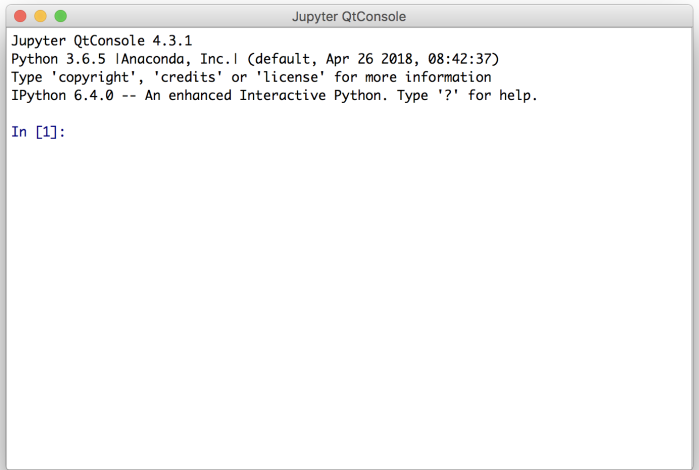


Figure 1.4: IPython console.

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.

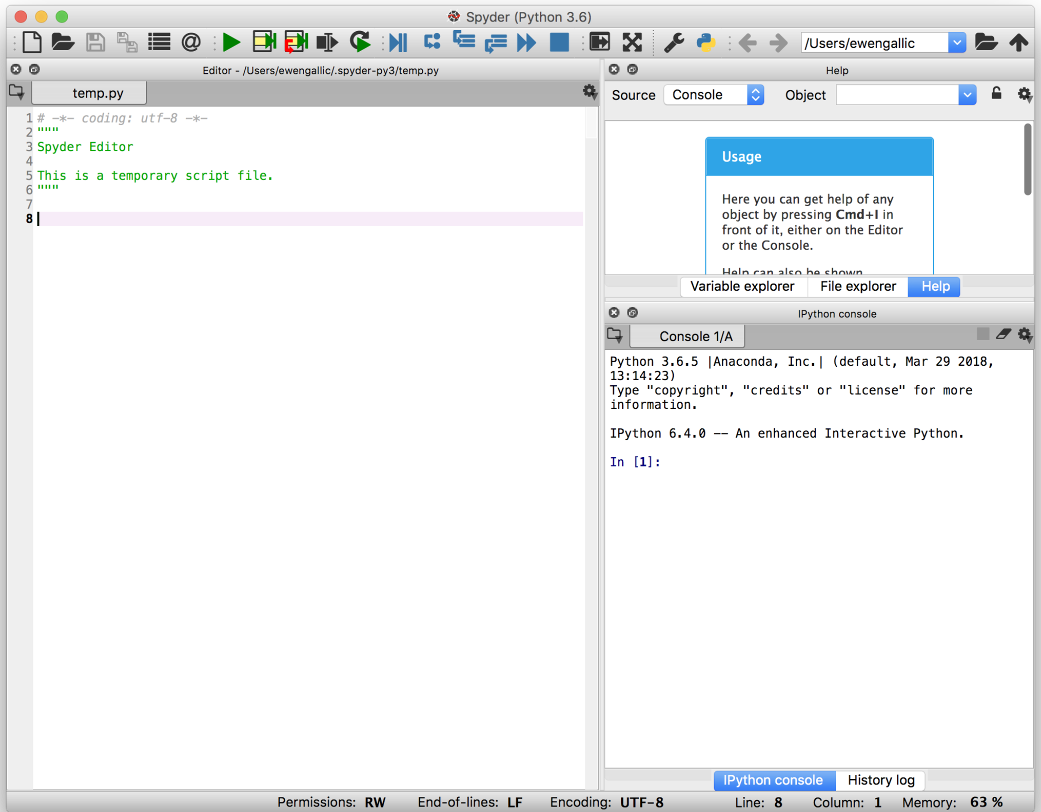


Figure 1.5: Spyder.

1.3.4 Jupyter Notebook

A graphical user interface in a web browser for IPython has gained 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?.

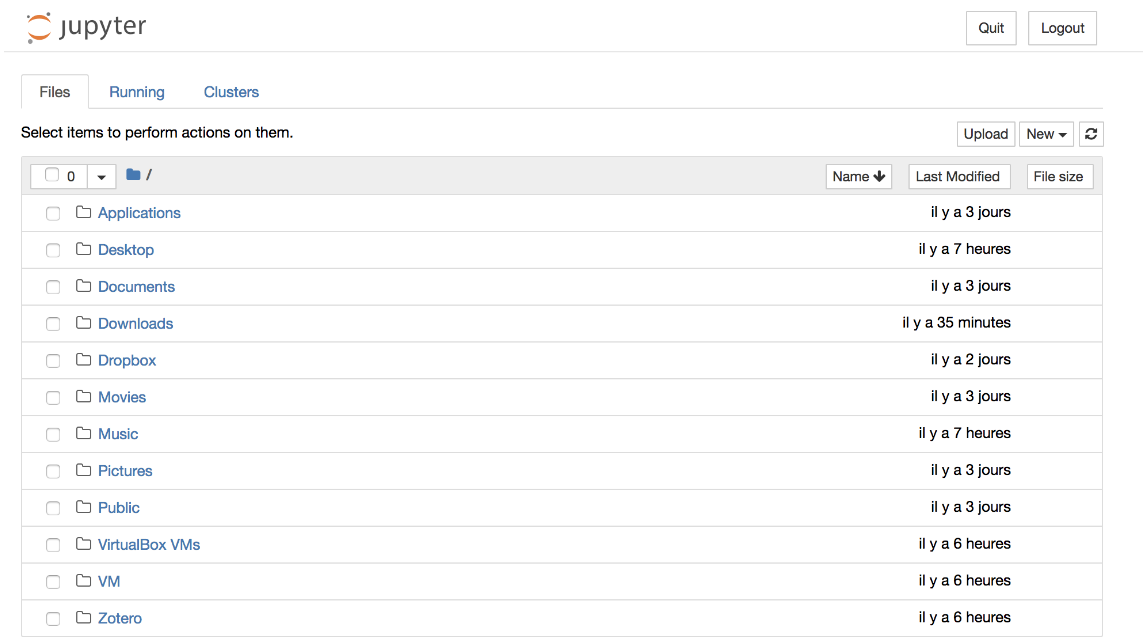


Figure 1.6: Jupyter Notebook.

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.

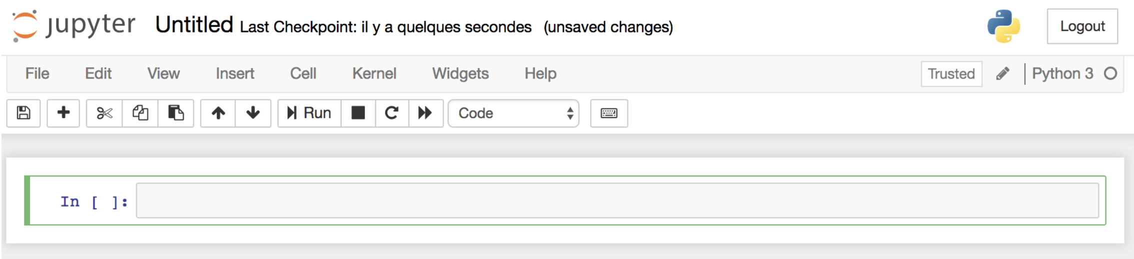


Figure 1.7: An Empty Notebook.

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 theBkey 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:

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 **deleted** 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`:

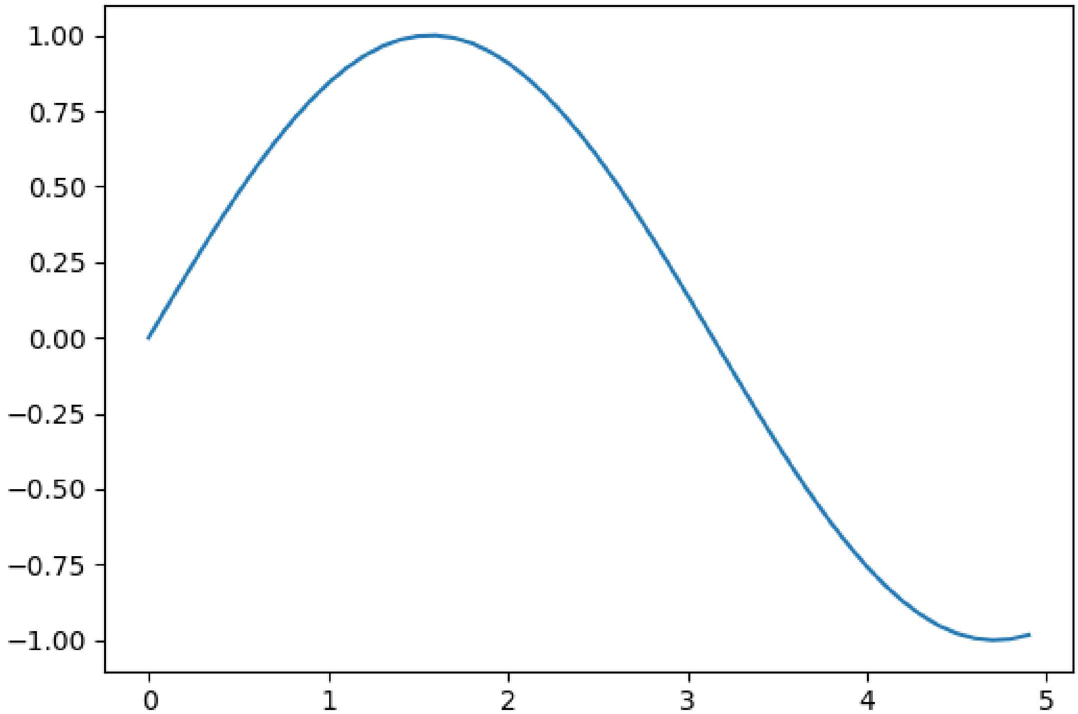
```
import pandas as pd
x = pd.Series([1, 5, 4])
print(x)
```

```
## 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:

- `?` : fournitprovides 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 trial 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
```

Remark 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
`\n`	New line	`\r`	Line break
`\t`	Tabulation	`\b`	Backspace
`\\`	Backslash	`\'`	Quotation mark
`\"`	Double quotation mark	``\` ``	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 ith 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 argument 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
```

Remark 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 Seach 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 arguments. 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
```

Remark 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 arguments, it is possible to limit the number of substrings returned:

```
# Will return the elements matched up to the index 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.
- Noooooooo! Noooo!"'''
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.', '- Noooooooo! Noooo!"']
```

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 arguments 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(longitude.rjust(7," "))
```

```
##   48.11
```

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)):

Method	Description
`capitalize()`	Capitalization of the first character and lowercase of the rest
`casefold()`	Removes case distinctions (useful for comparing strings without regard to case)

Method	Description
`count()`	Counts the number of occurrences (without overlap) of a pattern
`encode()`	Encodes a string of characters in a specific encoding
`find()`	Returns the smallest element where a substring is found
`lower()`	Returns the string having passed each alphabetical character in lower case
`replace()`	Replaces one pattern with another
`split()`	Separates the chain into substring according to a pattern
`title()`	Returns the string after passing each first letter of a word through a capital letter
`upper()`	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 the separation line as the separator character.
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.

Remark 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 arguments (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 calling the `real` and `imag` elements, respectively.

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

```
## 1.0
```

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

```
## 3.0
```

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 moduels 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'>
```

Remark 2.5.1

It is not mandatory to specify the name of the arguments 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
```

Remark 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`
`%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 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`

Code	Description	Example
`%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())
```

```
## 2019-10-08 17:53:08.259629
```

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

```
## 2019-10-08 17:53:08.263955
```

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 argument. When an argument 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, tuplets, 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
```

```python
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 (:):

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

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

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

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

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

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

Remark 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 argument, the step :

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

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

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

```python
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 arguments. Among these arguments, 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", "aaaaa", "aa"]
x.sort(key=len)
print(x)
```

```
## ['a', 'aa', 'aa', 'aaaaa']
```

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 tuplets 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, tuplets 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 tuplets** 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', 'Aix-en-Provence', 'Marseille', 'Rennes'}
```

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', 'Aix-en-Provence', 'Marseille', 'Rennes'}
```

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', 'Aix-en-Provence', 'Marseille', 'Rennes'}
```

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

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

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

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

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

```
## {'Rennes', 'Aix-en-Provence', 'Nice', '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)
```

```
## {'Rennes', 'Aix-en-Provence', 'Nice', '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 :  {'Nice', 'Toulon', '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,
  "equipes": ["Cleveland", "Boston"]}
print(my_dict)
```

```
## {'nom': 'Kyrie', 'prenom': 'John', 'naissance': 1992, '
   equipes': ['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:
    1. key: `id`, value: `1`,
    2. key: `description`, value: `A photo of the Old Port of Marseille`,
    3. key: `loc`, value: a list in which the following coordinates are given `5.3772133`, `43.302424`.

2. add the following key-value pair to the `photo` dictionary: key : `user`, value : `bob`.
3. 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).
4. Delete the entry in `photo` whose key is `user`.
5. 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*, *Adition 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
$\neq$	`!=` (or `<>`)	Different from
$>$	`>`	Greater than
$\geq$	`>=`	& Greater than or equal to
$<$	`<`	Lower than
$\leq$	`<=`	Less than or equal to
$\in$	`in`	In
$\notin$	`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 inferior (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
`math.ceil(x)`	Smallest integer greater than or equal to `x`
`math.copysign(x, y)`	Absolute value of `x` but with the sign of `y`
`math.floor(x)`	Smallest integer less than or equal to `x`
`math.round(x, ndigits)`	Rounded from `x` to `ndigits` decimal places
`math.fabs(x)`	Absolute value of `x`
`math.exp(x)`	Exponential of `x`
`math.log(x)`	Natural logarithm of `x` (based on e)
`math.log(x, b)`	Logarithm based on `b` of `x`
`math.log10(x)`	Logarithm in base 10 of `x`
`math.pow(x,y)`	`x` high to the power `y`
`math.sqrt(x)`	Square root of `x`
`math.fsum()`	Sum of the values of `x`

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

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
‘`math.pi`	The number Pi (π)
`math.e`	The constant e
`math.tau`	The constant τ, equal to 2π
`math.inf`	The infinite (∞)
`-math.inf`	Minus infinity ($-\infty$)
`math.nan`	Floating point number *not to number*

4.6 Exercise

1. Calculate the remainder 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
```

Remark 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 argument `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.

Remark 5.1.1

Chapter 10 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 arguments 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` argument 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 arguments 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: Arguments of the `reader()` Function

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

Argument	Description
`strict`	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 10.16.2 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 argument `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 argument `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 arguments 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/ffile_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 10.16.2 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 argument `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.

Remark 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(#operateurs-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
```

Remark 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
```

Remark 6.3.1

The advantage of using `if-elif' conditional instructions over writing severalif`' 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 `europe` 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 `europe` 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 `europe` or `asia`, the variable `continent` will be set to `Other`.

To do this:

1. Create the two lists `europe` 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

Remark 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 arguments:

- `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`.

Remark 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 `mystery_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 draws 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 Arguments 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 Argument 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 arguments has not been mentioned here. When the function was called, the value of the first argument was assigned to the first argument (`l`), the second to the second argument (`k`) and finally the third to the third argument (`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 ommitting 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]
```

Remark 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 ir 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
```

> **Remark 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 arguments.

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 arguments, 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 = desc_stats(x)
print(res)
```

```
## {'mean': 3.9, 'std_dev': 2.8460498941515415}
```

```
message = "The average value equals {} and the standard deviation is {}"
print(message.format(res["mean"], res["std_dev"]))
```

```
## The average value equals 3.9 and the standard deviation is
   2.8460498941515415
```

8.5 Exercise

1. Create a function named `sum_n_integers` which returns the sum of the first integer n. Its only argument 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(list)
print(table)
```

```
## [1 2 4]
```

```
print(type(table))
```

```
## <class 'numpy.ndarray'>
```

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`, `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 availableon 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 table_2 : ", table_2.ndim)
```

```
## ndim table_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 table : ", table.size)
```

```
## size table :  6
```

```
print("size table_2: ", table_2.size)
```

```
## size table_2:  6
```

```
print("np.size(table):", np.size(table))
```

```
## np.size(table): 6
```

The `shape` attribute returns a tuple indicating the length for each dimension of the array:

```
print("size table: ", table.shape)
```

```
## size table:  (3, 2)
```

```
print("size table_2: ", table_2.shape)
```

```
## size table_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 toupper'.
- 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] : {} (type : {})"
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("table_1[-1] : ", table_1[-1]) # last element
```

```
## table_1[-1] :  12
```

```
print("table_2[-1] : ", table_2[-1]) # last element
```

```
## table_2[-1] :  [10 11 12]
```

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 arguments 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 Insterting 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
print("Table 1: \n", table_1)
```

```
## Table 1:
##  [0 2 3]
```

```
print("Table 2: \n", table_2)
```

```
## Table 2:
##  [0 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
`greater()`	Greater than
`greater_equal()`	Greater than or equal to
`less()`	Lower than
`less_equal()`	Lower than or equal to
`equal()`	Equal to
`not_equal()`	Different from

Code	Description
`logical_and()`	Logical And
`logical_or()`	Logical Or
`logical_xor()`	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
`np.inf`	Infinity (we get $-\infty$ by writing `-np.inf` or `np.NINF`)
`np.nan`	Representation as a floating point number of Not a Number
`np.e`	Euler constant (e)
`np.euler_gamma`	Euler-Mascheroni constant (γ)
`np.pi`	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: uannic 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
`negative(x)`	Opposite elements of elements of `x`
`absolute(x)`	Absolute values of the elements of `x`
`sign(x)`	Signs of the elements of `x` (0, 1 or -1)
`rint(x)`	Ronded value of `x` to the nearest integer
`floor(x)`	Truncated value of `x` to the next smaller integer
`ceil(x)`	Truncated value of `x` to the next larger integer
`sqrt(x)`	Square root of `x`
`square(x)`	Squared value of `x`
`sin(x)`, `cos(x)`, `tan(x)`	Sine (cosine, and tangent) of the elements of `x`
`sinh(x)`, `cosh(x)`, `tanh(x)`	Hyperbolic sine (cosine, and tangent) of the elements of `x`
`arcsin(x)`, `arccos(x)`, `arctan(x)`	Arc-sine (arc-cosine, and arc-tangent) de `x`
`arcsinh(x)`, `arccosh(x)`, `arctanh(x)`	Hyperbolic arc-sinus (arc-cosine, and arc-tangent) of the elements of `x`
`hypoth(x,y)`	Hypotenuse $\sqrt{x^2+y^2}$
`degrees(x)`	Conversion of the angles values of `x` from radians to degrees
`radians(x)`	Conversion of the angles values of `x` from degrees to radians
`exp(x)`	Exponential of the values of `x`
`expm1(x)`	$e^x - 1$
`log(x)`	Natural logarithm of the elements of `x`
`log10(x)`	Logatithm of the elements of `x` in base 10
`log2(x)`	Logarithm of the elements of `x` in base 2
`log1p(x)`	$ln(1+x$
`exp2(x)`	2^x
`isnan(x)`	Boolean table indicating `True` for the elements `NaN`
`isfinite(x)`	Boolean table indicating `True` for non-infinite and non-NaN elements
`isinf(x)`	Boolean array indicating `True` for infinite elements

Table 9.4: Binary Universal Functions

Code	Description
`add(x,y)`	Term by term addition of the elements of `x` and `y`
`subtract(x,y)`	Term by term substraction of the elements of `x` and `y`
`multiply(x,y)`	Term by term multiplication of the elements of `x` and `y`
`divide(x,y)`	Term by term division of the elements of `x` and `y`
`floor_divide(x,y)`	Largest integer smaller or equal to the division of the elements of `x` and `y`
`power(x,y)`	Elements of `x` to the power of the elements of `y`
`mod(x,y)`	Remainder of Euclidean term by term divisions of the eleemnts of `x` by the elements of `y`
`round(x,n)`	Rounded value of the elements of `x` up to n digits
`arctan2(x,y)`	Polar angles of `x` and `y`

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
`sum()`	Returns the sum of the elements
`prod()`	Returns the product of the elements
`cumsum()`	Returns the cumulative sum of the elements
`cumprod()`	Returns the cumulative product of the elements
`mean()`	Returns the average
`var()`	Returns the variance
`std()`	Returns the standard error
`min()`	Returns the minnimum value
`max()`	Returns the maximum value
`argmin()`	Returns the index of the first element with the lowest value
`argmax()`	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:  18.07638888888889
```

To apply these functions to a given axis, we modify the value of the argument `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
`sum(x)`, `nansum(x)`	Sum of the elements of `x` (`nansum(x)` does not take into account `NaN` values)
`mean(x)`, `nanmean()`	Average of `x`
`median(x)`, `nanmedian()`	Median of `x`
`average(x)`	Average of `x` (possibility to use weights using the `weight` argument)
`min(x)`, `nanmin()`	Mininum of `x`
`max(x)`, `nanmax()`	Maximum of `x`
`percentile(x,p)`, `nanpercentile(n,p)`	P-th percentile of `x`
`var(x)`, `nanvar(x)`	Variance of `x`
`std(x)`, `nanstd()`	Standard-deviation of `x`
`cov(x)`	Covariance of `x`
`corrcoef(x)`	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:  18.07638888888889
```

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
`rand(size)`	Drawing `size` obs. from a Uniform distribution $[0, 1]$
`uniform(a,b,size)`	TDrawing `size` obs. from a Uniform distribution $[a; b]$
`randint(a,b,size)`	Drawing `size` obs. from a Uniform distribution $[a; b[$
`randn(size)`	Drawing `size` obs. from a Normal distribution $\mathcal{N}(0, 1)$
`normal(mu, std, size)`	Drawing `size` obs. from a Normal distribution with `mu` mean and standard error `std`
`binomial(size, n, p)`	Drawing `size` obs. from a Binomial distribution $\mathcal{B}in(n, p)$
`beta(alpha, beta, size)`	Drawing `size` obs. from a Beta distribution $Beta(\alpha, \beta)$
`poisson(lambda, size)`	Drawing `size` obs. from a Poisson distribution $\mathcal{P}(\lambda)$
`standard_t(df, size)`	Drawing `size` obs. from a Student distribution $\mathcal{S}t(\mathrm{df})$

Here is an example of generating pseudo random numbers according to a Gaussian distribution:

```
x = np.random.normal(size=10)
print(x)
```

```
## [-0.46929103 -0.40617772 -0.45754895  0.39494142
   0.94284541  2.46180313
##  -0.01887105  0.11080495  0.50283461  0.20855191]
```

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)
```

```
## [[ 0.30802506  1.11662112 -2.08762678 -0.52793389]
##  [ 0.996928   -0.65864341  0.77624146 -1.10173406]
##  [-0.07997368  0.52742341  1.03289108  1.13246365]
##  [ 0.15286372  1.26929707 -1.33740934 -1.18743712]
##  [-0.13972581 -1.79977333 -2.13167746 -0.58245566]
##  [-0.60448707  1.18512919  0.94176895 -0.6211865 ]
##  [-0.04806523  0.64177182  1.39752478 -0.94884771]
##  [ 1.49388632 -0.1832472   0.50866272  0.77952575]
##  [-2.34890576  0.54460348  0.32012378  1.2872278 ]
##  [ 0.25163866 -0.19595951 -0.14274807  0.71915014]]
```

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 an argument:

```
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: $x = \begin{bmatrix}1 & 2 & 3 & 4 & 5\end{bmatrix}$

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

Data manipulation with pandas

pandas is an open-source library based on NumPy providing easy-to-use data structures and data analysis tools. The reader familiar with the basic functions of the R language will find many similar features with pandas.

To access the features of pandas, it is common practice to load the library by assigning it the alias pd:

```
import pandas as pd
```

We will also use numpy functions (c.f. Section 9). Let's make sure to load this library, if it hasn't already been done:

```
import numpy as np
```

10.1 Structures

We will look at two types of structures, series (series') and dataframes (DataFrame').

10.1.1 Series

Series are one-dimensional tables of indexed data.

10.1.1.1 Creating Series from a List

To create one, we can define a list, then apply the function `Series` of `pandas`:

```
s = pd.Series([1, 4, -1, np.nan, .5, 1])
print(s)
```

```
## 0    1.0
## 1    4.0
## 2   -1.0
## 3    NaN
## 4    0.5
## 5    1.0
## dtype: float64
```

The previous example shows that the `s` series created contains both the data and an associated index. The `values` attribute is used to display the values that are stored in a `numpy` array:

```
print("values of s: ", s.values)
```

```
## values of s:  [ 1.   4.  -1.   nan  0.5  1. ]
```

```
print("type of values of s: ", type(s.values))
```

```
## type of values of s:  <class 'numpy.ndarray'>
```

The index is stored in a specific structure of `pandas`:

```
print("index of s: ", s.index)
```

```
## index of s:  RangeIndex(start=0, stop=6, step=1)
```

```
print("type of the index of s: ", type(s.index))
```

```
## type of the index of s:  <class 'pandas.core.indexes.range.
   RangeIndex'>
```

It is possible to assign a name to the series as well as to the index:

```
s.name = "my_serie"
s.name = "name_index"
print("name of the Serie: {} , name of the index: {}".format(s.name, s.index.name))
```

```
## name of the Serie: name_index , name of the index: None
```

```
print("Serie s: \n", s)
```

```
## Serie s:
##  0    1.0
## 1    4.0
## 2   -1.0
## 3    NaN
## 4    0.5
## 5    1.0
## Name: name_index, dtype: float64
```

10.1.1.2 Definition of the Index

The index can be defined by the user, at the time of creating the series:

```
s = pd.Series([1, 4, -1, np.nan],
             index = ["o", "d", "i", "l"])
print(s)
```

```
## o    1.0
## d    4.0
## i   -1.0
## l    NaN
## dtype: float64
```

The index can also be defined with numerical values, without being forced to follow a specific order:

```
s = pd.Series([1, 4, -1, np.nan],
              index = [4, 40, 2, 3])
print(s)
```

```
## 4      1.0
## 40     4.0
## 2     -1.0
## 3      NaN
## dtype: float64
```

The index can be modified later, by overwriting the attribute `index`:

```
s.index = ["o", "d", "i", "l"]
print("Série s : \n", s)
```

```
## Série s :
##  o     1.0
## d    4.0
## i   -1.0
## l    NaN
## dtype: float64
```

10.1.1.3 Creation of Special Series

A simple trick to creating series with a repeated value consists in providing a scalar to the `Series` function of `NumPy` and an index whose length will correspond to the number of times the scalar is repeated:

```
s = pd.Series(5, index = [np.arange(4)])
print(s)
```

```
## 0    5
## 1    5
## 2    5
## 3    5
## dtype: int64
```

A series can be created from a dictionary:

```
dictionary = {"King": "Arthur",
              "Knight_Round_Table": "Percival",
              "Druid": "Merlin"}
s = pd.Series(dictionary)
print(s)
```

```
## King                     Arthur
## Knight_Round_Table     Percival
## Druid                    Merlin
## dtype: object
```

As noted in the previous output, the dictionary keys were used for the index. When creating the series, specific values can be specified in the key argument: this will result in retrieving only the observations corresponding to these keys:

```
dictionary = {"King": "Arthur",
              "Knight_Round_Table": "Percival",
              "Druid": "Merlin"}
s = pd.Series(dictionary, index = ["King", "Druid"])
print(s)
```

```
## King      Arthur
## Druid     Merlin
## dtype: object
```

10.1.2 Dataframes

Dataframes correspond to the data format traditionally found in economics, two-dimensional tables, with column variables and observations in rows. The columns and rows of the dataframes are indexed.

10.1.2.1 Creating Dataframes from a Dictionary

To create a dataframe, the `DataFrame()` function of `pandas` can be provided with a dictionary that can be transformed into a `series`. This is the case of a dictionary where the values associated with the keys are all the same length:

```
dict = {"height" :
               [58, 59, 60, 61, 62,
                63, 64, 65, 66, 67,
                68, 69, 70, 71, 72],
        "weight":
               [115, 117, 120, 123, 126,
                129, 132, 135, 139, 142,
                146, 150, 154, 159, 164]
       }
df = pd.DataFrame(dict)
print(df)
```

```
##     height  weight
## 0       58     115
## 1       59     117
## 2       60     120
## 3       61     123
## 4       62     126
## 5       63     129
## 6       64     132
## 7       65     135
## 8       66     139
## 9       67     142
## 10      68     146
## 11      69     150
## 12      70     154
## 13      71     159
## 14      72     164
```

The position of the elements in the dataframe serves as an index. As for the series, the values are accessible in the `values` attribute and the index in the `index` attribute. The columns are also indexed:

```
print(df.columns)
```

```
## Index(['height', 'weight'], dtype='object')
```

The `head()` method displays the first few lines (the first 5, by default). You can modify its `n` arguments to indicate the number of lines to be returned:

```
df.head(2)
```

```
##    height  weight
## 0      58     115
## 1      59     117
```

When creating a dataframe from a dictionary, if the name of the columns to be imported is specified by a list of strings provided in the `columns` argument of the `DataFrame` function, it is possible to define not only the columns to be filled but also their order of appearance.

For example, to import only the `weight` column:

```
df = pd.DataFrame(dict, columns = ["weight"])
print(df.head(2))
```

```
##    weight
## 0     115
## 1     117
```

And to define the order in which the columns will appear:

```
df = pd.DataFrame(dict, columns = ["weight", "height"])
print(df.head(2))
```

```
##    weight  height
## 0     115      58
## 1     117      59
```

If a column name that is missing from the dictionary keys is specified, the resulting dataframe will contain a column with this name but filled with the values `NaN`:

```
df = pd.DataFrame(dict, columns = ["weight", "height", "age"])
print(df.head(2))
```

```
##    weight  height  age
## 0     115      58  NaN
## 1     117      59  NaN
```

10.1.2.2 Creating Dataframes from Series

A dataframe can be created from a series:

```
s = pd.Series([1, 4, -1, np.nan], index = ["o", "d", "i", "l"])
s.name = "name_variable"
df = pd.DataFrame(s, columns = ["name_variable"])
print(df)
```

```
##    name_variable
## o            1.0
## d            4.0
## i           -1.0
## l            NaN
```

If the series is not named, the `columns` argument of the `DataFrame` function must not be filled in. But in this case, the column will not have a name, just a numerical index.

```
s = pd.Series([1, 4, -1, np.nan], index = ["o", "d", "i", "l"])
df = pd.DataFrame(s)
print(df)
```

```
##      0
## o  1.0
## d  4.0
## i -1.0
```

```
## 1   NaN
```

```
print(df.columns.name)
```

```
## None
```

10.1.2.3 Creating Dataframes from a Dictionary List

A dataframe can be created from a list of dictionaries:

```
dico_1 = {
    "Name": "Pendragon",
    "Surname": "Arthur",
    "Role": "King of Britain"
}
dico_2 = {
    "Name": "",
    "Surname": "Perceval",
    "Role": "Knight of the Round Table"
}

df = pd.DataFrame([dico_1, dico_2])
print(df)
```

```
##         Name                       Role   Surname
## 0  Pendragon            King of Britain    Arthur
## 1             Knight of the Round Table  Perceval
```

If some keys are missing in one or more of the dictionaries in the list, the corresponding values in the dataframe will be `NaN`:

```
dico_3 = {
    "Surname": "Guinevere",
    "Role": "Queen of Britain"
}
```

```
df = pd.DataFrame([dico_1, dico_2, dico_3])
print(df)
```

```
##         Name                     Role    Surname
## 0  Pendragon          King of Britain     Arthur
## 1        NaN  Knight of the Round Table   Perceval
## 2        NaN         Queen of Britain  Guinevere
```

10.1.2.4 Création de dataframes à partir d'un dictionnaire de séries

A dataframe can also be created from a series dictionary. To illustrate the method, let's create two dictionaries:

```
# Annual 2017 GDP
# In millions of current dollars
dict_gdp_current = {
    "France": 2582501.31,
    "USA": 19390604.00,
    "UK": 2622433.96
}
# Annual consumer price index
dict_cpi = {
    "France": 0.2,
    "UK": 0.6,
    "USA": 1.3,
    "Germany": 0.5
}
```

From these two dictionaries, let's create two corresponding series:

```
s_gdp_current = pd.Series(dict_gdp_current)
s_cpi = pd.Series(dict_cpi)

print("s_gdp_current : \n", s_gdp_current)
```

```
## s_gdp_current :
##  France     2582501.31
## USA       19390604.00
```

```
## UK          2622433.96
## dtype: float64
```

```
print("\ns_cpi : \n", s_cpi)
```

```
##
## s_cpi :
##  France      0.2
## UK          0.6
## USA         1.3
## Germany     0.5
## dtype: float64
```

Then, let's create a dictionary of series:

```
dict_from_series = {
    "gdp": s_gdp_current,
    "cpi": s_cpi
}
print(dict_from_series)
```

```
## {'gdp': France      2582501.31
## USA        19390604.00
## UK          2622433.96
## dtype: float64, 'cpi': France      0.2
## UK          0.6
## USA         1.3
## Germany     0.5
## dtype: float64}
```

Finally, let's create our dataframe:

```
s = pd.DataFrame(dict_from_series)
print(s)
```

```
##                    gdp  cpi
## France      2582501.31  0.2
## Germany            NaN  0.5
```

```
## UK         2622433.96   0.6
## USA       19390604.00   1.3
```

Remark 10.1.1

The `dict_gdp_current` dictionary does not contain a `Germany` key, unlike the `dict_cpi` dictionary. When the dataframe was created, the GDP value for Germany was therefore assigned as `NaN`.

10.1.2.5 Creation of Dataframes from a Two-dimensional `NumPy` Array

A dataframe can also be created from a `Numpy` array. When creating, with the function `DataFrame()` of `NumPy`, it is possible to specify the name of the columns (otherwise, the indication of the columns will be numerical):

```
listing = [
    [1, 2, 3],
    [11, 22, 33],
    [111, 222, 333],
    [1111, 2222, 3333]
]
array_np = np.array(listing)
df = pd.DataFrame(array_np,
                  columns = ["a", "b", "c"])
print(df)
```

```
##        a     b     c
## 0      1     2     3
## 1     11    22    33
## 2    111   222   333
## 3   1111  2222  3333
```

10.1.2.6 Dimensions

The dimensions of a dataframe are accessed with the attribute `shape`.

```
print("shape : ", df.shape)
```

```
## shape :  (4, 3)
```

The number of lines can also be displayed as follows:

```
print("shape : ", len(df))
```

```
## shape :  4
```

And the number of columns:

```
print("shape : ", len(df.columns))
```

```
## shape :  3
```

10.1.2.7 Modification of the Index

As for the series, the index can be modified once the dataframe has been created, by overwriting the values of the attributes `index` and `columns`, for the index of rows and columns, respectively:

```
df.index = ["o", "d", "i", "l"]
df.columns = ["aa", "bb", "cc"]
print(df)
```

```
##      aa    bb    cc
## o     1     2     3
## d    11    22    33
## i   111   222   333
## l  1111  2222  3333
```

10.2 Selection

In this section, we look at different ways to select data in series and dataframes. There are two distinct ways to do this:

- a first one based on the use of brackets directly on the object for which we want to select certain parts;
- second based on indexers, accessible as attributes of `NumPy` objects (`loc`, `at`, `iat`, `iat`, etc.)

The second method avoids some confusion that may appear in the case of numerical indexes.

10.2.1 For Series

First, let's look at ways to extract values from series.

10.2.1.1 With brackets

The index can be used to extract the data:

```
s = pd.Series([1, 4, -1, np.nan, .5, 1])
s[0] # 1st element of s
```

```
## 1.0
```

```
s[1:3] # From the 2nd (included) to the 4th (not included)
```

```
## 1     4.0
## 2    -1.0
## dtype: float64
```

```
s[[0,4]] # First to 5th element (not included)
```

```
## 0     1.0
## 4     0.5
```

```
## dtype: float64
```

Note that unlike 'numpy' tables or lists, negative values for the index cannot be used to retrieve data by counting their position relative to the end:

```
s[-2]
```

```
## Error in py_call_impl(callable, dots$args, dots$keywords):
   KeyError: -2
##
## Detailed traceback:
##   File "<string>", line 1, in <module>
##   File "/anaconda3/lib/python3.6/site-packages/pandas/core/
   series.py", line 766, in __getitem__
##     result = self.index.get_value(self, key)
##   File "/anaconda3/lib/python3.6/site-packages/pandas/core/
   indexes/base.py", line 3103, in get_value
##     tz=getattr(series.dtype, 'tz', None))
##   File "pandas/_libs/index.pyx", line 106, in pandas._libs.
   index.IndexEngine.get_value
##   File "pandas/_libs/index.pyx", line 114, in pandas._libs.
   index.IndexEngine.get_value
##   File "pandas/_libs/index.pyx", line 162, in pandas._libs.
   index.IndexEngine.get_loc
##   File "pandas/_libs/hashtable_class_helper.pxi", line 958,
    in pandas._libs.hashtable.Int64HashTable.get_item
##   File "pandas/_libs/hashtable_class_helper.pxi", line 964,
    in pandas._libs.hashtable.Int64HashTable.get_item
```

In the case of an index composed of strings, it is then possible to refer either to the content of the index (to make it simple, its name) or to its position when extracting the data from the series:

```
s = pd.Series([1, 4, -1, np.nan],
              index = ["o", "d", "i", "l"])
print("Series s : \n", s)
```

```
## Series s :
##  o    1.0
```

```
## d     4.0
## i    -1.0
## l     NaN
## dtype: float64
```

```
print('s["d"] : \n', s["d"])
```

```
## s["d"] :
##   4.0
```

```
print('s[1] : \n', s[1])
```

```
## s[1] :
##   4.0
```

```
print("elements o and i : \n", s[["o", "i"]])
```

```
## elements o and i :
##   o     1.0
## i    -1.0
## dtype: float64
```

On the other hand, in the case where the index is defined with numerical values, to extract the values using the brackets, it will be by the value of the index and not by relying on the position:

```
s = pd.Series([1, 4, -1, np.nan],
              index = [4, 40, 2, 3])
print(s[40])
```

```
## 4.0
```

10.2.1.2 With Indexers

Pandas offers two types of multi-axis indication: `loc`, `iloc`. The first is mainly based on the use of axis labels, while the second is mainly based on positions using integers.

For the purposes of this section, let's create two series; one with a textual index, the other with a numerical index:

```
s_num = pd.Series([1, 4, -1, np.nan],
             index = [5, 0, 4, 1])
s_text = pd.Series([1, 4, -1, np.nan],
             index = ["c", "a", "b", "d"])
```

10.2.1.2.1 Extraction of a Single Element

To extract an object with `loc`, we use the name of the index :

```
print(s_num.loc[5], s_text.loc["c"])
```

```
## 1.0 1.0
```

To extract a single element with `iloc`, simply indicate its position:

```
(s_num.iloc[1], s_text.iloc[1])
```

```
## (4.0, 4.0)
```

10.2.1.2.2 Extraction of Several Elements

To extract several elements with `loc`, we use the names (labels) of the indices, which we provide in a list:

```
print("elements with labels 5 and 4:\n", s_num.loc[[5,4]])
```

```
## elements with labels 5 and 4:
##  5    1.0
## 4   -1.0
## dtype: float64
```

```
print("elements with labels c and b: \n", s_text.loc[["c", "b"]])
```

```
## elements with labels c and b:
##  c    1.0
## b   -1.0
## dtype: float64
```

To extract multiple elements with `iloc`:

```
print("elements at positions 0 and 2:\n", s_num.iloc[[0,2]])
```

```
## elements at positions 0 and 2:
##  5    1.0
## 4   -1.0
## dtype: float64
```

```
print("elements at positions 0 and 2: \n", s_text.iloc[[0,2]])
```

```
## elements at positions 0 and 2:
##  c    1.0
## b   -1.0
## dtype: float64
```

10.2.1.2.3 Slicing

It is possible to perform series slicing, to recover consecutive elements:

```
print("elements with label 5 to 4:\n", s_num.loc[5:4])
```

```
## elements with label 5 to 4:
##  5    1.0
## 0    4.0
## 4   -1.0
## dtype: float64
```

```
print("elements with label c to b: \n", s_text.loc["c":"b"])
```

```
## elements with label c to b:
##  c    1.0
## a    4.0
## b   -1.0
## dtype: float64
```

To extract multiple elements with `iloc`:

```
print("elements at positions 0 and 2:\n", s_num.iloc[0:2])
```

```
## elements at positions 0 and 2:
##  5    1.0
## 0    4.0
## dtype: float64
```

```
print("elements at positions 0 and 2: \n", s_text.iloc[0:2])
```

```
## elements at positions 0 and 2:
##  c    1.0
## a    4.0
## dtype: float64
```

As we have seen so far, the upper limit value is not included in the breakdown.

10.2.1.2.4 Mask

A mask can also be used to extract elements, either using `loc` or `iloc`:

```
print("\n",s_num.loc[[True, False, False, True]])
```

```
##
##  5    1.0
## 1    NaN
## dtype: float64
```

```
print("\n", s_text.loc[[True, False, False, True]])
```

```
##
##  c     1.0
## d     NaN
## dtype: float64
```

```
print("\n", s_num.iloc[[True, False, False, True]])
```

```
##
##  5     1.0
## 1     NaN
## dtype: float64
```

```
print("\n", s_text.iloc[[True, False, False, True]])
```

```
##
##  c     1.0
## d     NaN
## dtype: float64
```

10.2.1.2.5 What's the Point?

Why introduce such ways of extracting data and not just extract it using the brackets on the objects? Let's look at a simple example. Let's assume that we have the `s_num` series, with an index composed of integers that is not a sequence ranging from 0 to the number of elements. In this case, if we want to recover the 2nd element, because of the index composed of numerical values, we cannot obtain it by asking `s[1]`. To extract the 2nd of the series, we must know that its index is `0` and thus ask :

```
print("The element whose index is 0: ", s_num[0])
```

```
## The element whose index is 0:  4.0
```

To be able to perform the extraction according to the position, it is very useful to have this attribute `iloc`:

```
print("The element in 2nd position:", s_num.iloc[1])
```

```
## The element in 2nd position: 4.0
```

10.2.2 For dataframes

Now let's look at different ways to extract data from a dataframe. Let's create two dataframes as an example, one with a numerical index; another with a textual index :

```
dict = {"height" : [58, 59, 60, 61, 62],
        "weight": [115, 117, 120, 123, 126],
        "age": [28, 33, 31, 31, 29],
        "taille": [162, 156, 172, 160, 158],
       }
df_num = pd.DataFrame(dict)
df_text = pd.DataFrame(dict, index=["a", "e", "c", "b", "d"])
print("df_num : \n", df_num)
```

```
## df_num :
##     height  weight  age  taille
## 0       58     115   28     162
## 1       59     117   33     156
## 2       60     120   31     172
## 3       61     123   31     160
## 4       62     126   29     158
```

```
print("df_text : \n", df_text)
```

```
## df_text :
##     height  weight  age  taille
## a       58     115   28     162
## e       59     117   33     156
## c       60     120   31     172
## b       61     123   31     160
```

```
## d       62      126    29      158
```

To make it simple, when we want to perform an extraction with the `iloc` attributes, the syntax is as follows:

```
df.iloc[line_selection, column_selection]
```

with `line_selection`:

- a single value: `1` (second line) ;
- a list of values: `[2, 1, 3]` (3rd line, 2nd line and 4th line) ;
- a breakdown: `[2:4]` (from the 3rd line to the 4th line (not included)).

for `column_selection`:

- a single value: `1` (second column) ;
- a list of values: `[2, 1, 3]` (3rd column, 2nd column and 4th column) ;
- a breakdown: `[2:4]` (from the 3rd column to the 4th column (not included)).

With `loc`, the syntax is as follows:

```
df.loc[line_selection, column_selection]
```

with `line_selection` :

- a single value: ‘`a'' (line named`a‘);
- a list of names: `["a", "c", "b"]` (Rows named “a”, “c” and “b”) ;
- a mask: `df.['a']<10` (Rows for which the mask values are `True`).

with `column_selection` :

- a single value: ‘`a'' (column named`a‘);
- a list of values: `["a", "c", "b"]` (columns named “a”, “c” and “b”) ;
- a breakdown: ‘`["a": "c"]` (from the column named “a” to the column named “c”).

10.2.2.1 Extraction of a Rows

To extract a Rows from a dataframe, the name of the Rows can be used with `loc`:

```
print("Rows named 'e':\n", df_text.loc["e"])
```

```
## Rows named 'e':
##  height      59
## weight     117
## age         33
## taille     156
## Name: e, dtype: int64
```

```
print("\nRows named 'e':\n", df_num.loc[1])
```

```
##
## Rows named 'e':
##  height      59
## weight     117
## age         33
## taille     156
## Name: 1, dtype: int64
```

Or, its position with `iloc`:

```
print("Rows in position 0:\n", df_text.iloc[0])
```

```
## Rows in position 0:
##  height      58
## weight     115
## age         28
## taille     162
## Name: a, dtype: int64
```

```
print("\nRows in position 0:\n", df_num.iloc[0])
```

```
##
## Rows in position 0:
##  height      58
## weight     115
## age         28
```

```
## taille    162
## Name: 0, dtype: int64
```

10.2.2.2 Extraction of Several Rows

To extract multiple lines from a dataframe, their names can be used with `loc` (in an array):

```
print("Rows named a and c:\n", df_text.loc[["a", "c"]])
```

```
## Rows named a and c:
##     height  weight  age  taille
## a       58     115   28     162
## c       60     120   31     172
```

```
print("\nRows named 0 and 2:\n", df_num.loc[[0, 2]])
```

```
##
## Rows named 0 and 2:
##     height  weight  age  taille
## 0       58     115   28     162
## 2       60     120   31     172
```

Or, their position with `iloc`:

```
print("Rows at positions 0 and 3:\n", df_text.iloc[[0, 3]])
```

```
## Rows at positions 0 and 3:
##     height  weight  age  taille
## a       58     115   28     162
## b       61     123   31     160
```

```
print("\nRows at positions 0 and 3:\n", df_num.iloc[[0, 3]])
```

```
##
## Rows at positions 0 and 3:
##     height  weight  age  taille
## 0      58     115   28     162
## 3      61     123   31     160
```

10.2.2.3 Slicing of Several Rows

A line sequence can be retrieved by delimiting the first and last line to be retrieved according to their name and using `loc`:

```
print("Rows from label à to c:\n", df_text.loc["a":"c"])
```

```
## Rows from label à to c:
##     height  weight  age  taille
## a      58     115   28     162
## e      59     117   33     156
## c      60     120   31     172
```

```
print("\nRows from label 0 to 2:\n", df_num.loc[0:2])
```

```
##
## Rows from label 0 to 2:
##     height  weight  age  taille
## 0      58     115   28     162
## 1      59     117   33     156
## 2      60     120   31     172
```

With the attribute `iloc`, this is also possible (again, the upper bound is not included):

```
print("Rows at position 0 to 3 (not included):\n", df_text.iloc[0:3])
```

```
## Rows at position 0 to 3 (not included):
##     height  weight  age  taille
## a      58     115   28     162
## e      59     117   33     156
```

```
## c       60      120    31      172
```

```
print("\nRows at position 0 to 3 (not included):\n", df_num.iloc[0:3])
```

```
##
## Rows at position 0 to 3 (not included):
##      height  weight  age  taille
## 0       58      115   28      162
## 1       59      117   33      156
## 2       60      120   31      172
```

10.2.2.4 Mask

A mask can also be used to select certain rows For example, if we want to retrieve the rows for which the variable **height** has a value greater than 60, we use the following mask:

```
mask = df_text["height"]> 60
print(mask)
```

```
## a     False
## e     False
## c     False
## b      True
## d      True
## Name: height, dtype: bool
```

To filter:

```
print(df_text.loc[mask])
```

```
##     height  weight  age  taille
## b       61      123   31      160
## d       62      126   29      158
```

10.2.2.5 Extraction of a Single Column

To extract a column from a dataframe, we can use square brackets and refer to the name of the column (which is indexed by names):

```
print(df_text['weight'].head(2))
```

```
## a    115
## e    117
## Name: weight, dtype: int64
```

By selecting a single column, we obtain a series (the dataframe index is kept for the series):

```
print(type(df_text['weight']))
```

```
## <class 'pandas.core.series.Series'>
```

A column can also be extracted by referring to the attribute of the dataframe named after this column:

```
print(df_text.weight.head(2))
```

```
## a    115
## e    117
## Name: weight, dtype: int64
```

As for the series, we can rely on the attributes `loc` and `iloc`:

```
print("Column 2 (loc):\n", df_text.loc[:,"weight"])
```

```
## Column 2 (loc):
##  a    115
## e    117
## c    120
## b    123
## d    126
## Name: weight, dtype: int64
```

```
print("Column 2 (iloc):\n", df_text.iloc[:,1])
```

```
## Column 2 (iloc):
##  a    115
## e    117
## c    120
## b    123
## d    126
## Name: weight, dtype: int64
```

10.2.2.6 Extraction of Several Columns

To extract several columns, the names of the columns can be placed in a table:

```
print(df_text[["weight", "height"]])
```

```
##    weight  height
## a     115      58
## e     117      59
## c     120      60
## b     123      61
## d     126      62
```

The order in which these columns are called corresponds to the order in which they will be returned.

Again, we can use the `loc` attribute (we use the colon here to specify that we want all the lines):

```
print("Columns from weight to height:\n", df_text.loc[:,["weight", "height"]])
```

```
## Columns from weight to height:
##     weight  height
## a     115      58
## e     117      59
## c     120      60
## b     123      61
```

```
## d      126       62
```

And the `iloc` attribute:

```
print("Columns 2 and 1 :\n", df_num.iloc[:,[1,0]])
```

```
## Columns 2 and 1 :
##      weight  height
## 0      115      58
## 1      117      59
## 2      120      60
## 3      123      61
## 4      126      62
```

10.2.2.7 Slicing Several Columns

To perform a slice, the attributes `loc` and `iloc` can be used. We must be careful as the names of the columns used for the breakdown are not placed in a table here:

With `loc`:

```
print("Columns 2 and 2:\n", df_text.loc[:, "height":"age"])
```

```
## Columns 2 and 2:
##      height  weight  age
## a       58     115   28
## e       59     117   33
## c       60     120   31
## b       61     123   31
## d       62     126   29
```

And with the `iloc` attribute:

```
print("Columns from position 0 to 2 (not included) :\n",
      df_text.iloc[:, 0:2])
```

```
## Columns from position 0 to 2 (not included) :
##      height  weight
## a       58     115
## e       59     117
## c       60     120
## b       61     123
## d       62     126
```

10.2.2.8 Extraction of Rows and Columns

Now that we have reviewed multiple ways to select one or more rows or columns, we can also mention that it is possible to make selections of columns and rows in the same instruction.

For example, with `iloc`, let's select the rows from position 0 to position 2 (not included) and the columns from position 1 to 3 (not included):

```
print(df_text.iloc[0:2, 1:3])
```

```
##    weight  age
## a     115   28
## e     117   33
```

With `loc`, let's select the rows named `a` and `c` and the columns from the one named `weight` to `age`.

```
df_text.loc[["a", "c"], "weight":"age"]
```

```
##    weight  age
## a     115   28
## c     120   31
```

10.3 Renaming Columns in a Dataframe

To rename a column in a dataframe, `pandas` offers the method `rename()`. Let's take an example with our `df` dataframe:

```
dict = {"height" : [58, 59, 60, 61, 62],
        "weight": [115, 117, 120, 123, 126],
        "age": [28, 33, 31, 31, 29],
        "taille": [162, 156, 172, 160, 158],
       }
df = pd.DataFrame(dict)
print(df)
```

```
##    height  weight  age  taille
## 0      58     115   28     162
## 1      59     117   33     156
## 2      60     120   31     172
## 3      61     123   31     160
## 4      62     126   29     158
```

Let's rename the column `taille` to `height`, using a dicionnaire specified in the argument `columns`, with as key the current name of the column, and value the new name :

```
df.rename(index=str, columns={"taille": "height"}, inplace=True)
print(df)
```

```
##    height  weight  age  height
## 0      58     115   28     162
## 1      59     117   33     156
## 2      60     120   31     172
## 3      61     123   31     160
## 4      62     126   29     158
```

For the change to be effective, the argument `inplace` is set to `True`, otherwise the change is not made to the dataframe.

To rename several columns at the same time, we can provide several pairs of value keys in the dictionary:

```
df.rename(index=str,
          columns={"weight": "weight_pounds", "age" : "years"},
          inplace=True)
print(df)
```

```
##    height  weight_pounds  years  height
## 0      58            115     28     162
## 1      59            117     33     156
## 2      60            120     31     172
## 3      61            123     31     160
## 4      62            126     29     158
```

10.4 Filtering

To filter the data in a table, depending on the values encountered for some variables, masks are used, as indicated in Section 10.2.2.4.

Let's look at some examples here, by redefining our dataframe:

```
dict = {"height" : [58, 59, 60, 61, 62],
        "weight": [115, 117, 120, 123, 126],
        "age": [28, 33, 31, 31, 29],
        "height_cm": [162, 156, 172, 160, 158],
       }
df = pd.DataFrame(dict)
print(df)
```

```
##    height  weight  age  height_cm
## 0      58     115   28        162
## 1      59     117   33        156
## 2      60     120   31        172
## 3      61     123   31        160
## 4      62     126   29        158
```

The idea is to create a mask returning a series containing Boolean values, one per line. When the value of the mask line is set to `True`, the dataframe line on which the mask will be applied will be retained, while it will not be retained when the value of the mask line is set to `False`.

Let's look at a simple example, in which we want to keep observations only for which the value of the `age` variable is less than 30:

```
mask = df["age"] < 30
print(mask)
```

```
## 0     True
## 1    False
## 2    False
## 3    False
## 4     True
## Name: age, dtype: bool
```

Then, we simply have to apply this mask, with `loc`. We want all the columns, but only a few rows:

```
print(df.loc[mask])
```

```
##    height  weight  age  height_cm
## 0      58     115   28        162
## 4      62     126   29        158
```

Note: it also works without `loc`:

```
print(df[mask])
```

```
##    height  weight  age  height_cm
## 0      58     115   28        162
## 4      62     126   29        158
```

More simply, we can use the `query()` method of `pandas`. A Boolean expression to be evaluated is provided for this method to filter the data:

```
print(df.query("age<30"))
```

```
##    height  weight  age  height_cm
```

```
## 0       58      115    28          162
## 4       62      126    29          158
```

The query can be a little more complex, by combining comparison operators (see Section 4.2) and logical operators (see Section 4.3). For example, suppose that we want to filter the dataframe values to retain only those observations for which the size is less than or equal to 62 and the mass is strictly greater than 120. The request would then be:

```
print(df.query("weight > 120 and height < 62"))
```

```
##     height  weight  age  height_cm
## 3       61     123   31        160
```

It can be noted that the following instruction gives the same result:

```
print(df.query("weight > 120").query("height < 62"))
```

```
##     height  weight  age  height_cm
## 3       61     123   31        160
```

10.4.1 Checking whether a value belongs to dataframe

To create a mask indicating whether the values of a series or dataframe belong to a set, the `isin()` method can be used. For example, let's return a mask indicating if the values in the `height` column of `df` are in the range $[59, 60]$:

```
df.height.isin(np.arange(59,61))
```

```
## 0    False
## 1     True
## 2     True
## 3    False
## 4    False
## Name: height, dtype: bool
```

10.5 Missing Values

In economics, it is quite common to face incomplete data. The way missing data is managed by `pandas` is the use of two special values: `None` and `NaN`.

The value `None` can be used in the tables `NumPy` only when the type of the latter is `object`.

```
table_none = np.array([1, 4, -1, None])
print(table_none)
```

```
## [1 4 -1 None]
```

```
print(type(table_none))
```

```
## <class 'numpy.ndarray'>
```

With an array of type `object`, operations performed on the data will be less efficient than with a numerical array. (VanderPlas 2016, p 121).

The value `NaN` is a floating point number value (see Section 9.1.10). `NumPy` manages it differently from `NaN`, and does not assign an object pass type from the start in the presence of `NaN`:

```
table_none = np.array([1, 4, -1, np.nan])
print(table_none)
```

```
## [ 1.   4. -1. nan]
```

```
print(type(table_none))
```

```
## <class 'numpy.ndarray'>
```

With `pandas`, these two values, `None` and `NaN` can be present:

```
s = pd.Series([1, None, -1, np.nan])
print(s)
```

```
## 0    1.0
## 1    NaN
## 2   -1.0
## 3    NaN
## dtype: float64
```

```
print(type(s))
```

```
## <class 'pandas.core.series.Series'>
```

This also applies to dataframes:

```
dict = {"height" : [58, 59, 60, 61, np.nan],
        "weight": [115, 117, 120, 123, 126],
        "age": [28, 33, 31, np.nan, 29],
        "height_cm": [162, 156, 172, 160, 158],
       }
df = pd.DataFrame(dict)
print(df)
```

```
##    height  weight   age  height_cm
## 0    58.0     115  28.0        162
## 1    59.0     117  33.0        156
## 2    60.0     120  31.0        172
## 3    61.0     123   NaN        160
## 4     NaN     126  29.0        158
```

However, it should be noted that only the type of variables for which missing values exist are passed to `float64`:

```
print(df.dtypes)
```

```
## height       float64
## weight         int64
```

```
## age           float64
## height_cm       int64
## dtype: object
```

Remark 10.5.1

Note that the data is recorded on a `float64` type. When working on a table with no missing values, whose type is `int` or `bool`, if a missing value is entered, `pandas` will change the data type to `float64` and `object`, respectively.

`pandas` offers different ways to handle missing values.

10.5.1 Identify Missing Values

With the `isnull()` method, a boolean mask is returned, indicating `True` for observations with the value `NaN` or `None`:

```
print(s.isnull())
```

```
## 0    False
## 1     True
## 2    False
## 3     True
## dtype: bool
```

To know if a value is not zero, the `notnull()` method can be used:

```
print(s.notnull())
```

```
## 0     True
## 1    False
## 2     True
## 3    False
## dtype: bool
```

10.5.2 Remove Observations with Missing Values

The `dropna()` method allows to remove observations with null values:

```
print(df.dropna())
```

```
##    height  weight   age  height_cm
## 0    58.0     115  28.0        162
## 1    59.0     117  33.0        156
## 2    60.0     120  31.0        172
```

10.5.3 Removing Missing Values by Other Values

To replace missing values with other values, `pandas` proposes to use the method `fillna()`:

```
print(df.fillna(-9999))
```

```
##    height  weight     age  height_cm
## 0    58.0     115    28.0        162
## 1    59.0     117    33.0        156
## 2    60.0     120    31.0        172
## 3    61.0     123 -9999.0        160
## 4 -9999.0     126    29.0        158
```

10.6 Deletions

To delete a value on one of the axes of a series or dataframe, `NumPy` offers the method `drop()`.

10.6.1 Deleting Elements in a Series

To illustrate how the `drop()` method works, let's create a series with a numerical index, another with a textual index :

```
s_num = pd.Series([1, 4, -1, np.nan],
             index = [5, 0, 4, 1])
s_text = pd.Series([1, 4, -1, np.nan],
             index = ["c", "a", "b", "d"])
```

An element can be deleted from a series by using its name:

```
print("for s_num: \n", s_num.drop(5))
```

```
## for s_num:
##  0     4.0
## 4    -1.0
## 1     NaN
## dtype: float64
```

```
print("\nfor s_text: \n", s_text.drop("c"))
```

```
##
## for s_text:
##  a     4.0
## b    -1.0
## d     NaN
## dtype: float64
```

We can also retrieve the name according to the position, by going through a detour using the `index()` method:

```
print("s_num.index[0]: ", s_num.index[0])
```

```
## s_num.index[0]:  5
```

```
print("s_text.index[0]: ", s_text.index[0])
```

```
## s_text.index[0]:  c
```

```
print("for s_num: \n", s_num.drop(s_num.index[0]))
```

```
## for s_num:
##  0     4.0
## 4   -1.0
## 1     NaN
## dtype: float64
```

```
print("\nfor s_text: \n", s_text.drop(s_text.index[0]))
```

```
##
## for s_text:
##  a     4.0
## b   -1.0
## d     NaN
## dtype: float64
```

To delete several elements, we can provide several index names in a list using the `drop()` method:

```
print("for s_num: \n", s_num.drop([5, 4]))
```

```
## for s_num:
##  0     4.0
## 1     NaN
## dtype: float64
```

```
print("\nfor s_text: \n", s_text.drop(["c", "b"]))
```

```
##
## for s_text:
##  a     4.0
## d     NaN
## dtype: float64
```

Again, we can retrieve the name according to the position, by going through a detour using the `index()` method:

```
print("s_num.index[[0,2]]: ", s_num.index[[0,2]])
```

```
## s_num.index[[0,2]]:  Int64Index([5, 4], dtype='int64')
```

```
print("s_text.index[[0,2]]: ", s_text.index[[0,2]])
```

```
## s_text.index[[0,2]]:  Index(['c', 'b'], dtype='object')
```

```
print("for s_num: \n", s_num.drop(s_num.index[[0,2]]))
```

```
## for s_num:
##  0     4.0
## 1     NaN
## dtype: float64
```

```
print("\nfor s_text: \n", s_text.drop(s_text.index[[0,2]]))
```

```
##
## for s_text:
##  a     4.0
## d     NaN
## dtype: float64
```

It is also possible to use a slicing to obtain the series without the element(s) (See Section 10.2.1.2.3)

10.6.2 Deleting Elements in a Dataframe

To illustrate how the `drop()` method works on a dataframe, let's create one:

```
s_num = pd.Series([1, 4, -1, np.nan],
             index = [5, 0, 4, 1])
s_text = pd.Series([1, 4, -1, np.nan],
             index = ["c", "a", "b", "d"])
```

```
dict = {"height" : [58, 59, 60, 61, np.nan],
        "weight": [115, 117, 120, 123, 126],
        "age": [28, 33, 31, np.nan, 29],
        "height_cm": [162, 156, 172, 160, 158],
       }
df = pd.DataFrame(dict)
```

10.6.2.1 Deleting Rows

To delete a row from a dataframe, we can refer to its name (here, the names are numbers, but they are labels):

```
print("Delete the first row:  \n", df.drop(0))
```

```
## Delete the first row:
##      height  weight   age  height_cm
## 1     59.0     117  33.0        156
## 2     60.0     120  31.0        172
## 3     61.0     123   NaN        160
## 4      NaN     126  29.0        158
```

If the rows have text labels, they can first be retrieved using the `index()` method:

```
label_pos_0 = df.index[0]
print("Delete the first row:  \n", df.drop(label_pos_0))
```

```
## Delete the first row:
##      height  weight   age  height_cm
## 1     59.0     117  33.0        156
## 2     60.0     120  31.0        172
## 3     61.0     123   NaN        160
## 4      NaN     126  29.0        158
```

To delete several rows, the name of these rows in a list is given to the `drop()` method:

```
print("Delete the 1st and 4th rows:  \n", df.drop([0,3]))
```

```
## Delete the 1st and 4th rows:
##      height  weight   age  height_cm
## 1     59.0     117  33.0        156
## 2     60.0     120  31.0        172
## 4      NaN     126  29.0        158
```

Or, by indicating the positions of the lines:

```
label_pos = df.index[[0, 3]]
print("Delete the 1st and 4th rows:  \n", df.drop(label_pos))
```

```
## Delete the 1st and 4th rows:
##      height  weight   age  height_cm
## 1     59.0     117  33.0        156
## 2     60.0     120  31.0        172
## 4      NaN     126  29.0        158
```

It is also possible to use a slicing to obtain the series without the element(s) (See Sections 10.2.2.3 et 10.2.2.7)

10.6.2.2 Deleting Columns

To delete a column from a dataframe, we proceed in the same way as for rows, but by adding the parameter `axis=1` to the method `drop()` to specify that we are interested in the columns :

```
print("Delete the first column:  \n", df.drop("height", axis=1))
```

```
## Delete the first column:
##      weight   age  height_cm
## 0      115  28.0        162
## 1      117  33.0        156
## 2      120  31.0        172
## 3      123   NaN        160
## 4      126  29.0        158
```

We can first retrieve the labels of the columns according to their position using the method `columns()`:

```
label_pos = df.columns[0]
print("label_pos : ", label_pos)
```

```
## label_pos :  height
```

```
print("Delete the first column:  \n", df.drop(label_pos, axis=1))
```

```
## Delete the first column:
##     weight   age  height_cm
## 0      115  28.0        162
## 1      117  33.0        156
## 2      120  31.0        172
## 3      123   NaN        160
## 4      126  29.0        158
```

To delete several columns, the names of these columns are given in a list in the `drop()` method:

```
print("Delete the 1st and 4th columns:  \n",
 df.drop(["height", "height_cm"], axis = 1))
```

```
## Delete the 1st and 4th columns:
##     weight   age
## 0      115  28.0
## 1      117  33.0
## 2      120  31.0
## 3      123   NaN
## 4      126  29.0
```

Or, by indicating the positions of the columns:

```
label_pos = df.columns[[0, 3]]
print("Delete the 1st and 4th columns:  \n", df.drop(label_pos, axis=1))
```

```
## Delete the 1st and 4th columns:
```

```
##      weight   age
## 0      115   28.0
## 1      117   33.0
## 2      120   31.0
## 3      123    NaN
## 4      126   29.0
```

It is also possible to use a slicing to obtain the series without the element(s) (c.f. Sections 10.2.2.3 and 10.2.2.7)

10.7 Replacing Values

We will now look at how to modify one or more values, in the case of a series and then a dataframe.

10.7.1 For a Series

To modify a particular value in a series or dataframe, the equal symbol (`=`) can be used, having previously targeted the location of the value to be modified, using the extraction techniques explained in Section 10.2.

For example, let's consider the following series:

```
s_num = pd.Series([1, 4, -1, np.nan],
             index = [5, 0, 4, 1])
print("s_num: ", s_num)
```

```
## s_num:  5    1.0
## 0    4.0
## 4   -1.0
## 1    NaN
## dtype: float64
```

Let's modify the second element of `s_num`, to give it the value -3 :

```
s_num.iloc[1] = -3
print("s_num: ", s_num)
```

```
## s_num:  5     1.0
## 0    -3.0
## 4    -1.0
## 1     NaN
## dtype: float64
```

It is of course possible to modify several values at the same time.

Again, all we need to do is target the positions (there are many ways to do this) and provide an object of equivalent dimensions to replace the targeted values. For example, in our `s_num` series, we will replace the values in position 1 and 3 (2nd and 4th values) with -10 and -9 :

```
s_num.iloc[[1,3]] = [-10, -9]
print(s_num)
```

```
## 5      1.0
## 0    -10.0
## 4     -1.0
## 1     -9.0
## dtype: float64
```

10.7.2 For a Dataframe

Let's consider the following dataframe:

```
dict = {"city" : ["Marseille", "Aix",
                  "Marseille", "Aix", "Paris", "Paris"],
        "year": [2019, 2019, 2018, 2018,2019, 2019],
        "x": [1, 2, 2, 2, 0, 0],
        "y": [3, 3, 2, 1, 4, 4],
       }
df = pd.DataFrame(dict)
print("df: \n", df)
```

```
## df:
##          city  year  x  y
## 0   Marseille  2019  1  3
## 1         Aix  2019  2  3
## 2   Marseille  2018  2  2
## 3         Aix  2018  2  1
## 4       Paris  2019  0  4
## 5       Paris  2019  0  4
```

10.7.2.1 Changes of a Particular Value

Let's change the value of the first line of `df` for the column `year`, so that it is 2020. First, let's retrieve the position of the `year` column in the dataframe, using the `get_loc()` method applied to the `colnames` attribute of the dataframe:

```
pos_year = df.columns.get_loc("year")
print("pos_year: ", pos_year)
```

```
## pos_year:  1
```

Then, let's make the modification:

```
df.iloc[0,pos_year] = 2020
print("df: \n", df)
```

```
## df:
##          city  year  x  y
## 0   Marseille  2020  1  3
## 1         Aix  2019  2  3
## 2   Marseille  2018  2  2
## 3         Aix  2018  2  1
## 4       Paris  2019  0  4
## 5       Paris  2019  0  4
```

10.7.2.2 Modifications on One or More Columns

To modify all the values in a column to place a particular value, for example a 2 in the `x` column of `df`:

```
df.x = 2
print("df: \n", df)
```

```
## df:
##          city  year  x  y
## 0  Marseille  2020  2  3
## 1        Aix  2019  2  3
## 2  Marseille  2018  2  2
## 3        Aix  2018  2  1
## 4      Paris  2019  2  4
## 5      Paris  2019  2  4
```

It is also possible to modify the values in the column by providing a list of values:

```
df.x = [2, 3, 4, 2, 1, 0]
print("df: \n", df)
```

```
## df:
##          city  year  x  y
## 0  Marseille  2020  2  3
## 1        Aix  2019  3  3
## 2  Marseille  2018  4  2
## 3        Aix  2018  2  1
## 4      Paris  2019  1  4
## 5      Paris  2019  0  4
```

We can therefore imagine modifying the values of a column according to the values that we read in another column. For example, let's assume the following code: if the value of `y` is 2, then x is `a', if the value ofyis 1, if the value ofxis`b', otherwise it is `NaN`. First, let's build a list containing the values to insert (which we will name `nv_val`), using a loop. We will go through all the elements of the `y' column, and at each iteration add tonv_val`' the value obtained by making our comparisons:

```
new_val = []
for i in np.arange(len(df.index)):
        if df.y[i] == 2:
            new_val.append("a")
        elif df.y[i] == 1:
            new_val.append("b")
        else:
            new_val.append(np.nan)
print("new_val: ", new_val)
```

```
## new_val:  [nan, nan, 'a', 'b', nan, nan]
```

We are ready to modify the content of the `x` column of `df` to replace it with `new_val`:

```
df.x = new_val
print("df: \n", df)
```

```
## df:
##          city  year    x  y
## 0  Marseille  2020  NaN  3
## 1        Aix  2019  NaN  3
## 2  Marseille  2018    a  2
## 3        Aix  2018    b  1
## 4      Paris  2019  NaN  4
## 5      Paris  2019  NaN  4
```

To replace several columns at the same time:

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

```
## df:
##          city  year  x  y
## 0  Marseille  2020  2  1
## 1        Aix  2019  3  1
## 2  Marseille  2018  4  1
## 3        Aix  2018  2  1
## 4      Paris  2019  1  1
```

```
## 5       Paris  2019  0  1
```

In the previous instruction, we replaced the contents of the `x` and `y` columns with a vector of handwritten values for `x` and with the value 1 for all observations for `y`.

10.7.2.3 Modifications on One or More Rows

To replace a row with a constant value (of little interest in the current example):

```
df.iloc[1,:] = 1
print("df: \n", df)
```

```
## df:
##           city  year  x  y
## 0  Marseille  2020  2  1
## 1          1     1  1  1
## 2  Marseille  2018  4  1
## 3        Aix  2018  2  1
## 4      Paris  2019  1  1
## 5      Paris  2019  0  1
```

It may be more interesting to replace an observation as follows:

```
df.iloc[1,:] = ["Aix", 2018, 1, 2]
print("df: \n", df)
```

```
## df:
##           city  year  x  y
## 0  Marseille  2020  2  1
## 1        Aix  2018  1  2
## 2  Marseille  2018  4  1
## 3        Aix  2018  2  1
## 4      Paris  2019  1  1
## 5      Paris  2019  0  1
```

To replace several rows, the method is identical:

```
df.iloc[[1,3],:] = [
    ["Aix", 2018, 1, 2],
    ["Aix", 2018, -1, -1]
]
print("df: \n", df)
```

```
## df:
##          city  year  x  y
## 0  Marseille  2020  2  1
## 1        Aix  2018  1  2
## 2  Marseille  2018  4  1
## 3        Aix  2018 -1 -1
## 4      Paris  2019  1  1
## 5      Paris  2019  0  1
```

10.8 Adding Values

Now let's look at how to add values, first in a series, then in a dataframe.

10.8.1 For a Series

Let's consider the following series:

```
s_num = pd.Series([1, 4, -1, np.nan],
             index = [5, 0, 4, 1])
print("s_num: ", s_num)
```

```
## s_num:  5    1.0
## 0    4.0
## 4   -1.0
## 1    NaN
## dtype: float64
```

10.8.1.1 Adding a Single Value in a Series

To add a value, we use the `append()` method. Here, with `s_num`, as the index is manual, we are compelled to provide a series with a value for the index as well:

```
s_num_2 = pd.Series([1], index = [2])
print("s_num_2: \n", s_num_2)
```

```
## s_num_2:
##  2    1
## dtype: int64
```

```
s_num = s_num.append(s_num_2)
print("s_num: \n", s_num)
```

```
## s_num:
##  5    1.0
## 0    4.0
## 4   -1.0
## 1    NaN
## 2    1.0
## dtype: float64
```

Note that the `append()` method returns a view. To effectively add values, we must make a new assignment.

By having a series with an automatically generated numerical index, we can specify the value `True` for the `ignore_index` argument of the `append()` method to indicate that we do not take into account the value of the index of the object we add :

```
s = pd.Series([10, 2, 4])
s = s.append(pd.Series([2]), ignore_index=True)
print("s: \n", s)
```

```
## s:
##  0    10
## 1     2
## 2     4
## 3     2
```

```
## dtype: int64
```

10.8.1.2 Adding Several Values in a Series

To add several values, we use the `append()` method. Here, with `s_num`, as the index is manual, we are required to provide a series with a value for the index as well:

```
s_num_2 = pd.Series([1], index = [2])
s_num.append(s_num_2)
```

```
## 5     1.0
## 0     4.0
## 4    -1.0
## 1     NaN
## 2     1.0
## 2     1.0
## dtype: float64
```

```
print("s_num: ", s_num)
```

```
## s_num:  5     1.0
## 0     4.0
## 4    -1.0
## 1     NaN
## 2     1.0
## dtype: float64
```

By having a series with an automatically generated numerical index:

```
s = pd.Series([10, 2, 4])
s.append(pd.Series([2]), ignore_index=True)
```

```
## 0    10
## 1     2
## 2     4
## 3     2
```

```
## dtype: int64
```

10.8.2 For a dataframe

Let's go back to our dataframe:

```
dict = {"city" : ["Marseille", "Aix",
                  "Marseille", "Aix", "Paris", "Paris"],
        "year": [2019, 2019, 2018, 2018,2019, 2019],
        "x": [1, 2, 2, 2, 0, 0],
        "y": [3, 3, 2, 1, 4, 4],
       }
df = pd.DataFrame(dict)
print("df : \n", df)
```

```
## df :
##          city  year  x  y
## 0   Marseille  2019  1  3
## 1         Aix  2019  2  3
## 2   Marseille  2018  2  2
## 3         Aix  2018  2  1
## 4       Paris  2019  0  4
## 5       Paris  2019  0  4
```

10.8.2.1 Adding a Row in a Dataframe

As for a series, to add a row, we use the `append()` method. First, let's create a new dataframe with the line to add:

```
new_row = pd.DataFrame([["Marseille", "2021", 2, 4]],
                       columns = df.columns)
print("new_row: \n", new_row)
```

```
## new_row:
##          city  year  x  y
```

```
## 0  Marseille  2021  2  4
```

We made sure to have the same column name here, by indicating in the `columns` parameter of the `pd.DataFrame` method the name of the `df` columns, *i.e.*, `df.columns`.

Let's add the new row to `df`:

```
df = df.append(new_row, ignore_index=True)
```

Again, the `append()` method applied to a dataframe, returns a view and does not affect the object.

It can be noted that when adding a row, if the column names are not indicated in the same order as in the dataframe in which the addition is made, an indication must be added to the `out` parameter of the `append()` method:

- if `sort=True`, the order of the columns of the added row will be applied to the destination dataframe
- if `sort=False`, the column order of the destination dataframe will not be modified.

```
new_row = pd.DataFrame([["2021", "Marseille", 2, 4]],
                       columns = ["year", "coty", "x", "y"])
print("new_row: \n", new_row)
```

```
## new_row:
##      year       coty  x  y
## 0  2021  Marseille  2  4
```

```
print("avec sort=True : \n",
  df.append(new_row, ignore_index=True, sort = True))
```

```
## avec sort=True :
##          city       coty  x  y  year
## 0  Marseille       NaN  1  3  2019
## 1        Aix       NaN  2  3  2019
## 2  Marseille       NaN  2  2  2018
## 3        Aix       NaN  2  1  2018
## 4      Paris       NaN  0  4  2019
## 5      Paris       NaN  0  4  2019
```

```
## 6   Marseille         NaN   2   4   2021
## 7         NaN   Marseille   2   4   2021
```

10.8.2.2 Adding Multiple Rows in a Dataframe

To add several rows, it's exactly the same principle as with a single one, just add a dataframe of several rows, with the same names again.

The rows to be inserted:

```
new_rows = pd.DataFrame([
    ["Marseille", "2022", 2, 4],
    ["Aix", "2022", 3, 3]],
    columns = df.columns)
print("new_rows: \n", new_rows)
```

```
## new_rows:
##          city  year  x  y
## 0   Marseille  2022  2  4
## 1         Aix  2022  3  3
```

Then the insertion:

```
df = df.append(new_rows, ignore_index=True)
```

10.8.2.3 Adding a Column to a Dataframe

To add a column in a dataframe:

```
from numpy import random
df["z"] = random.rand(len(df.index))
print("df: \n", df)
```

```
## df:
##          city  year  x  y         z
## 0   Marseille  2019  1  3  0.117443
## 1         Aix  2019  2  3  0.393782
```

```
## 2  Marseille  2018  2  2  0.452730
## 3        Aix  2018  2  1  0.538148
## 4      Paris  2019  0  4  0.790622
## 5      Paris  2019  0  4  0.465836
## 6  Marseille  2021  2  4  0.435332
## 7  Marseille  2022  2  4  0.569479
## 8        Aix  2022  3  3  0.969259
```

10.8.2.4 Adding Multiple Columns to a Dataframe

To add several columns:

```
df["a"] = random.rand(len(df.index))
df["b"] = random.rand(len(df.index))
print("df: \n", df)
```

```
## df:
##         city  year  x  y         z         a         b
## 0  Marseille  2019  1  3  0.117443  0.040556  0.689236
## 1        Aix  2019  2  3  0.393782  0.548120  0.929546
## 2  Marseille  2018  2  2  0.452730  0.462577  0.918117
## 3        Aix  2018  2  1  0.538148  0.376472  0.975302
## 4      Paris  2019  0  4  0.790622  0.327912  0.397002
## 5      Paris  2019  0  4  0.465836  0.813529  0.262626
## 6  Marseille  2021  2  4  0.435332  0.646552  0.430151
## 7  Marseille  2022  2  4  0.569479  0.047426  0.764531
## 8        Aix  2022  3  3  0.969259  0.994958  0.599731
```

10.9 Removing Duplicate Values

To remove duplicate values from a dataframe, `NumPy` offers the `drop_duplicates()` method, which takes several optional arguments:

- `subset`: by indicating one or more column names, the search for duplicates is only done on these columns;

- `keep`: allows to indicate which observation to keep in case of identified duplicates:
  - if `keep='first'`, all duplicates are removed except the first occurrence,
  - if `keep='last'`, all duplicates are removed except the last occurrence,
  - if `keep='False'`, all duplicates are removed;
- `inplace`: boolean (default: `False`) to indicate if duplicates should be removed from the dataframe or if a copy should be returned (default).

Let's give some examples using this dataframe which makes up two duplicates when we consider its totality. If we focus only on years or cities, or both, other duplicates can be identified.

```
dict = {"city" : ["Marseille", "Aix",
                  "Marseille", "Aix", "Paris", "Paris"],
        "year": [2019, 2019, 2018, 2018,2019, 2019],
        "x": [1, 2, 2, 2, 0, 0],
        "y": [3, 3, 2, 1, 4, 4],
       }
df = pd.DataFrame(dict)
print(df)
```

```
##          city  year  x  y
## 0   Marseille  2019  1  3
## 1         Aix  2019  2  3
## 2   Marseille  2018  2  2
## 3         Aix  2018  2  1
## 4       Paris  2019  0  4
## 5       Paris  2019  0  4
```

To remove the duplicates:

```
print(df.drop_duplicates())
```

```
##          city  year  x  y
## 0   Marseille  2019  1  3
## 1         Aix  2019  2  3
## 2   Marseille  2018  2  2
## 3         Aix  2018  2  1
## 4       Paris  2019  0  4
```

Remove duplicates by keeping the last value of the identified duplicates:

```
df.drop_duplicates(keep='last')
```

```
##         city  year  x  y
## 0  Marseille  2019  1  3
## 1        Aix  2019  2  3
## 2  Marseille  2018  2  2
## 3        Aix  2018  2  1
## 5      Paris  2019  0  4
```

To remove identified duplicates when focusing on city names, and keeping only the first value :

```
print(df.drop_duplicates(subset = ["city"], keep = 'first'))
```

```
##         city  year  x  y
## 0  Marseille  2019  1  3
## 1        Aix  2019  2  3
## 4      Paris  2019  0  4
```

Same as above but with a focus on couples (city, year)

```
print(df.drop_duplicates(subset = ["city", "year"], keep = 'first'))
```

```
##         city  year  x  y
## 0  Marseille  2019  1  3
## 1        Aix  2019  2  3
## 2  Marseille  2018  2  2
## 3        Aix  2018  2  1
## 4      Paris  2019  0  4
```

Note that the original dataframe was not impacted, since we did not touch the `inplace` argument. If we now ask that the changes be made on the dataframe instead of getting a copy:

```
df.drop_duplicates(subset = ["city", "year"], keep = 'first', inplace = True)
print(df)
```

```
##          city  year  x  y
## 0   Marseille  2019  1  3
## 1         Aix  2019  2  3
## 2   Marseille  2018  2  2
## 3         Aix  2018  2  1
## 4       Paris  2019  0  4
```

To find out if a value is duplicated in a dataframe, `NumPy` offers the `duplicated()` method, which returns a mask indicating for each observation, whether it is duplicated or not. Its operation is similar to `df.drop_duplicates()`, except for the `inplace` parameter which is not present.

```
print(df.duplicated(subset = ["city"], keep = 'first'))
```

```
## 0    False
## 1    False
## 2     True
## 3     True
## 4    False
## dtype: bool
```

We can use the `any()` method to find out if there are duplicates:

```
print(df.duplicated(subset = ["city"], keep = 'first').any())
```

```
## True
```

10.10 Operations

It is often necessary to perform operations on the columns of a dataframe, especially when it comes to creating a new variable.

By using the principles of column modification (see Section @ref(#pandas-ajout-valeurs)), it is quite easy to imagine that it is possible to apply the functions and methods of `NumPy` (see Section 9.1) on the values of the columns.

For example, let's consider the following dataframe:

```
dict = {"height" :
               [58, 59, 60, 61, 62,
                63, 64, 65, 66, 67,
                68, 69, 70, 71, 72],
        "weight":
               [115, 117, 120, 123, 126,
                129, 132, 135, 139, 142,
                146, 150, 154, 159, 164]
       }
df = pd.DataFrame(dict)
print(df)
```

```
##     height  weight
## 0       58     115
## 1       59     117
## 2       60     120
## 3       61     123
## 4       62     126
## 5       63     129
## 6       64     132
## 7       65     135
## 8       66     139
## 9       67     142
## 10      68     146
## 11      69     150
## 12      70     154
## 13      71     159
## 14      72     164
```

Let's add the column `height_2`, increasing the values of the column `height` to square:

```
df["height_2"] = df.height**2
print(df.head(3))
```

```
##    height  weight  height_2
## 0      58     115      3364
## 1      59     117      3481
```

```
## 2       60      120       3600
```

Now, let's add the column `bmi`, providing the body mass indicator values for individuals in the dataframe (bmi $= \frac{\text{weight}}{\text{height}^2}$) :

```
df["bmi"] = df.weight / df.height_2
print(df.head(3))
```

```
##     height  weight  height_2       bmi
## 0       58     115      3364  0.034185
## 1       59     117      3481  0.033611
## 2       60     120      3600  0.033333
```

10.10.1 Statistics

`pandas` offers some methods for performing descriptive statistics for each column or row. To do this, the syntax is as follows (all arguments have a default value, the list is simplified here):

```
dataframe.stat_method(axis, skipna)
```

- `axis`: 0 for rows, 1 for columns
- `skipna`: if `True`, excludes missing values from the calculations

Among the available methods: - `mean()`: mean - `mode()`: mode - `median()`: median - `std()`: standard error - `min()`: minimum - `max()`: maximum - `mad()`: mean absolute deviation - `sum()`: sum - `prod()`: product - `count()`: counting the elements

For example, to calculate the average of the values for each column:

```
dico = {"height" : [58, 59, 60, 61, 62],
        "weight": [115, 117, 120, 123, 126],
        "age": [28, 33, 31, 31, 29],
        "height_cm": [162, 156, 172, 160, 158],
        "married": [True, True, False, False, True],
        "city": ["A", "B", "B", "B", "A"]
       }
```

```
df = pd.DataFrame(dico)
print(df.mean())
```

```
## height         60.0
## weight        120.2
## age            30.4
## height_cm     161.6
## married         0.6
## dtype: float64
```

If desired, we can average the column values (without any meaning here):

```
print(df.mean(axis=1))
```

```
## 0    72.8
## 1    73.2
## 2    76.6
## 3    75.0
## 4    75.2
## dtype: float64
```

These functions can be applied to a single column. For example, to display the minimum value:

```
print("min: ", df.height.min())
```

```
## min:  58
```

It is also useful to be able to obtain the position of the min and max values; this can be obtained with the methods `idxmin()` and `idxmax()`, respectively.

```
print("pos min: ", df.height.idxmin())
```

```
## pos min:  0
```

```
print("pos min: ", df.height.idxmax())
```

```
## pos min:  4
```

A very useful method is `describe()`, it allows to return descriptive statistics on all numerical columns :

```
print(df.describe())
```

```
##           height      weight        age   height_cm
## count   5.000000    5.000000   5.000000    5.000000
## mean   60.000000  120.200000  30.400000  161.600000
## std     1.581139    4.438468   1.949359    6.228965
## min    58.000000  115.000000  28.000000  156.000000
## 25%    59.000000  117.000000  29.000000  158.000000
## 50%    60.000000  120.000000  31.000000  160.000000
## 75%    61.000000  123.000000  31.000000  162.000000
## max    62.000000  126.000000  33.000000  172.000000
```

10.11 Sorting

It is easy to sort a dataframe in ascending or descending order by one or more of its columns. To do this, we use the method `sort_values()`. The syntax is as follows:

```
DataFrame.sort_values(by, axis=0, ascending=True,
                      inplace=False, kind="quicksort",
                      na_position="last")
```

- `by`: name or list of names of the column(s) used to sort
- `axis`: `0` for the index (default), `1` for the columns
- `ascending`: boolean or boolean list, when `True` the sorting is done by increasing values (default), when `False` it is done by decreasing values
- `inplace`: if `True`, the sorting affects the dataframe, otherwise it returns a view
- `kind`: choice of sorting algorithm (`quicksort` (default), `mergesort`, `heapsort`)
- `na_position`: if `first`, the missing values are placed at the beginning; if `last` (default), at the end.

Let us give some examples:

```
dict = {"height" : [58, 59, 60, 61, 62],
        "weight": [115, np.nan, 120, 123, 126],
        "age": [28, 33, 31, 31, 29],
        "height_cm": [162, 156, 172, 160, 158],
        "married": [True, True, np.nan, False, True],
        "city": ["A", "B", "B", "B", "A"]
       }
df = pd.DataFrame(dict)
```

If we sort the values in descending order of the values in the `height` column:

```
df.sort_values(by="height", ascending=False)
```

```
##    height  weight  age  height_cm married city
## 4      62   126.0   29        158    True    A
## 3      61   123.0   31        160   False    B
## 2      60   120.0   31        172     NaN    B
## 1      59     NaN   33        156    True    B
## 0      58   115.0   28        162    True    A
```

To sort in ascending order of the `married` values (recall that `True` is interpreted as 1 and `False` as 0), then decreasing by `weight`, placing the values `NaN` first:

```
df.sort_values(by=["married", "weight"],
               ascending=[True, False],
               na_position="first")
```

```
##    height  weight  age  height_cm married city
## 2      60   120.0   31        172     NaN    B
## 3      61   123.0   31        160   False    B
## 1      59     NaN   33        156    True    B
## 4      62   126.0   29        158    True    A
## 0      58   115.0   28        162    True    A
```

Note that the `NaN` values have increased for the subgroups composed according to the `married` values.

10.12 Concatenation

It is frequent to obtain data from multiple sources when conducting an analysis. It is then important to be able to combine the different sources into one. In this section, we will limit ourselves to concatenating different dataframes between them, in simple cases in which we know *a priori* that all we have to do is put two dataframes side by side or one below the other. The case of slightly more elaborate joints with matching according to one or more columns is discussed in Section 10.13.

First, let's create two dataframes with the same number of lines:

```
x_1 = pd.DataFrame(np.random.randn(5, 4),
                   columns=["a", "b", "c", "d"])
x_2 = pd.DataFrame(np.random.randn(5, 2),
                   columns = ["e", "f"])
print("x_1: \n", x_1)
```

```
## x_1:
##           a         b         c         d
## 0  0.231711 -0.474710 -0.309147 -2.032396
## 1 -0.174468 -0.642475 -0.625023  1.325887
## 2  0.531255  1.275284 -0.682826 -0.948186
## 3  0.777362  0.325113 -1.203486  1.209543
## 4  0.157622 -0.293555  0.111560  0.597679
```

```
print("\nx_2: \n", x_2)
```

```
##
## x_2:
##           e         f
## 0 -1.270093  0.120949
## 1 -0.193898  1.804172
## 2 -0.234694  0.939908
## 3 -0.171520 -0.153055
## 4 -0.363095 -0.067318
```

To "paste" the dataframe `x_2` to the right of `x_1`, we can use the `concat()` method of `pandas`. To indicate that concatenation is performed on the columns, the value `1` for the parameter `axix` is specified as follows:

```
print(pd.concat([x_1, x_2], axis = 1))
```

```
##           a         b         c         d         e
          f
## 0  0.231711 -0.474710 -0.309147 -2.032396 -1.270093
   0.120949
## 1 -0.174468 -0.642475 -0.625023  1.325887 -0.193898
   1.804172
## 2  0.531255  1.275284 -0.682826 -0.948186 -0.234694
   0.939908
## 3  0.777362  0.325113 -1.203486  1.209543 -0.171520
   -0.153055
## 4  0.157622 -0.293555  0.111560  0.597679 -0.363095
   -0.067318
```

To paste the dataframes below each other, the `append()` method can be used, as described in Section 10.8.2.1, or the `concat()` method can also be used.

```
x_3 = pd.DataFrame(np.random.randn(5, 2),
                   columns = ["e", "f"])
print("x_3: \n", x_3)
```

```
## x_3:
##            e         f
## 0  1.444721  0.325771
## 1 -0.855732 -0.697595
## 2 -0.276134 -1.258759
## 3  0.478094 -0.859764
## 4  0.571988 -0.173965
```

Let's add the observations of `x_3` below those of `x_2`:

```
print(pd.concat([x_2, x_3], axis = 0))
```

```
##           e         f
## 0 -1.270093  0.120949
## 1 -0.193898  1.804172
## 2 -0.234694  0.939908
## 3 -0.171520 -0.153055
```

```
## 4 -0.363095 -0.067318
## 0  1.444721  0.325771
## 1 -0.855732 -0.697595
## 2 -0.276134 -1.258759
## 3  0.478094 -0.859764
## 4  0.571988 -0.173965
```

As can be seen, the line index of `x_2` has not been modified. If we want it to be, we can specify it via the `ignore_index` argument:

```
print(pd.concat([x_2, x_3], axis = 0, ignore_index=True))
```

```
##           e         f
## 0 -1.270093  0.120949
## 1 -0.193898  1.804172
## 2 -0.234694  0.939908
## 3 -0.171520 -0.153055
## 4 -0.363095 -0.067318
## 5  1.444721  0.325771
## 6 -0.855732 -0.697595
## 7 -0.276134 -1.258759
## 8  0.478094 -0.859764
## 9  0.571988 -0.173965
```

If the column names are not identical, values `NaN` will be inserted:

```
x_4 = pd.DataFrame(np.random.randn(5, 2),
                   columns = ["e", "g"])
print("x_4: \n", x_4)
```

```
## x_4:
##            e         g
## 0  1.534900  0.872856
## 1  1.856835  0.025914
## 2  0.171984 -0.191163
## 3 -0.292936  1.655677
## 4 -0.207182 -0.686884
```

```
pd.concat([x_2, x_4], axis = 0, sort=False, ignore_index=True)
```

```
##           e         f         g
## 0 -1.270093  0.120949       NaN
## 1 -0.193898  1.804172       NaN
## 2 -0.234694  0.939908       NaN
## 3 -0.171520 -0.153055       NaN
## 4 -0.363095 -0.067318       NaN
## 5  1.534900       NaN  0.872856
## 6  1.856835       NaN  0.025914
## 7  0.171984       NaN -0.191163
## 8 -0.292936       NaN  1.655677
## 9 -0.207182       NaN -0.686884
```

10.13 Joins

It is more likely to use slightly more elaborate joins to bring together the different data sources into one. `pandas` offers a powerful way to gather data, the `merge()` function.

To illustrate the different joins in this section, let's create some dataframes:

```
exports_fr = pd.DataFrame(
    {"country" : "France",
     "year" : np.arange(2014, 2017),
     "exports" : [816.8192172, 851.6632573, 867.4014253]
    })

imports_fr = pd.DataFrame(
    {"country" : "France",
     "year" : np.arange(2015, 2018),
     "imports" : [898.5242962, 936.3691166, 973.8762149]
    })

exports_us = pd.DataFrame(
    {"country" : "USA",
```

```
     "year" : np.arange(2014, 2017),
     "exports" : [2208.678084, 2217.733347, 2210.442218]
    })

imports_us = pd.DataFrame(
    {"country" : "USA",
     "year" : np.arange(2015, 2018),
     "imports" : [2827.336251, 2863.264745, np.nan]
    })

imports_morocco = pd.DataFrame(
    {"pays" : "Morocco",
     "annee" : np.arange(2015, 2018),
     "imports" : [46.39884177, 53.52375588, 56.68165748]
    })
exports_morocco = pd.DataFrame(
    {"country" : "Morocco",
     "year" : np.arange(2014, 2017),
     "exports" : [35.50207915, 37.45996653, 39.38228396]
    })

exports = pd.concat([exports_fr, exports_us], ignore_index=True)
imports = pd.concat([imports_fr, imports_us], ignore_index=True)

print("exports: \n", exports)
```

```
## exports:
##   country  year      exports
## 0  France  2014   816.819217
## 1  France  2015   851.663257
## 2  France  2016   867.401425
## 3     USA  2014  2208.678084
## 4     USA  2015  2217.733347
## 5     USA  2016  2210.442218
```

```
print("\nimports: \n", imports)
```

```
##
```

```
## imports:
##     country  year      imports
## 0  France  2015   898.524296
## 1  France  2016   936.369117
## 2  France  2017   973.876215
## 3     USA  2015  2827.336251
## 4     USA  2016  2863.264745
## 5     USA  2017          NaN
```

The `merge()` function of `pandas` requires to specify the left table (which we will call here `x`) via the `left` argument on which the joining of the right table (which we will call here `y`) will be performed via the `right` argument.

By default, the `merge()` function performs an `inner` type join, *i.e.*, all `x` rows that match `y`, and all `x` and `y` columns will be in the result of the join :

```
print(pd.merge(left = imports, right = exports))
```

```
##   country  year      imports      exports
## 0  France  2015   898.524296   851.663257
## 1  France  2016   936.369117   867.401425
## 2     USA  2015  2827.336251  2217.733347
## 3     USA  2016  2863.264745  2210.442218
```

If we want to change the type of join, we can modify the value of the `how` parameter of the `merge()` function, to give it one of the following values:

- `left`: all `x` rows, and all `x` and `y` columns. Rows in `x` for which there is no match in `y` will have values `NaN` in the new columns. If there are several matches in the names between `x` and `y`, all combinations are returned
- `inner`: all `x` rows for which there are corresponding values in `y`, and all `x` and `y` columns. If there are several matches in the names between `x` and `y`, all possible combinations are returned
- `right`: all `y` rows, and all `y` and `x` columns. Rows in `y` for which there is no match in `x` will have values `NaN` in the new columns. If there are several matches in the names between `y` and `x`, all combinations are returned
- `outer`: all rows of `x` and `y`, and all columns of `x` and `y`. Lines of `x` for which there is no match in `y` and those of `y` for which there is no match in `x` will have values `NaN`.

```
print("left: \n", pd.merge(left = imports, right = exports, how="left"))
```

```
## left:
##    country  year      imports      exports
## 0  France  2015   898.524296   851.663257
## 1  France  2016   936.369117   867.401425
## 2  France  2017   973.876215          NaN
## 3     USA  2015  2827.336251  2217.733347
## 4     USA  2016  2863.264745  2210.442218
## 5     USA  2017          NaN          NaN
```

```
print("\nright: \n", pd.merge(left = imports, right = exports, how="right"))
```

```
##
## right:
##    country  year      imports      exports
## 0  France  2015   898.524296   851.663257
## 1  France  2016   936.369117   867.401425
## 2     USA  2015  2827.336251  2217.733347
## 3     USA  2016  2863.264745  2210.442218
## 4  France  2014          NaN   816.819217
## 5     USA  2014          NaN  2208.678084
```

```
print("\nouter: \n", pd.merge(left = imports, right = exports, how="outer"))
```

```
##
## outer:
##    country  year      imports      exports
## 0  France  2015   898.524296   851.663257
## 1  France  2016   936.369117   867.401425
## 2  France  2017   973.876215          NaN
## 3     USA  2015  2827.336251  2217.733347
## 4     USA  2016  2863.264745  2210.442218
## 5     USA  2017          NaN          NaN
## 6  France  2014          NaN   816.819217
## 7     USA  2014          NaN  2208.678084
```

The on argument, which expects a column name or list of names, is used to designate the columns used to make the join. The column names must be identical in both dataframes.

```
print(pd.merge(left = imports, right = exports, on = "country"))
```

```
##     country  year_x      imports  year_y      exports
## 0    France    2015   898.524296    2014   816.819217
## 1    France    2015   898.524296    2015   851.663257
## 2    France    2015   898.524296    2016   867.401425
## 3    France    2016   936.369117    2014   816.819217
## 4    France    2016   936.369117    2015   851.663257
## 5    France    2016   936.369117    2016   867.401425
## 6    France    2017   973.876215    2014   816.819217
## 7    France    2017   973.876215    2015   851.663257
## 8    France    2017   973.876215    2016   867.401425
## 9       USA    2015  2827.336251    2014  2208.678084
## 10      USA    2015  2827.336251    2015  2217.733347
## 11      USA    2015  2827.336251    2016  2210.442218
## 12      USA    2016  2863.264745    2014  2208.678084
## 13      USA    2016  2863.264745    2015  2217.733347
## 14      USA    2016  2863.264745    2016  2210.442218
## 15      USA    2017          NaN    2014  2208.678084
## 16      USA    2017          NaN    2015  2217.733347
## 17      USA    2017          NaN    2016  2210.442218
```

If the names of the columns used to make the join are different between the left and right dataframe, the argument `left_on` indicates the column name(s) of the left dataframe to be used for the join; and the argument `right_on` indicates the corresponding name(s) in the right dataframe:

```
pd.merge(left = imports_morocco, right = exports_morocco,
        left_on= ["pays", "annee"], right_on = ["country", "year"] )
```

```
##       pays  annee    imports  country  year    exports
## 0  Morocco   2015  46.398842  Morocco  2015  37.459967
## 1  Morocco   2016  53.523756  Morocco  2016  39.382284
```

With the argument `suffixes`, suffixes can be defined to be added to column names when there are columns in `x` and `y` with the same name but not used for joining. By default, the suffixes (`_x` and `_y`) are added.

```
print(pd.merge(left = imports, right = exports,
               on = "country",
               suffixes=("_left", "_right")).head(3))
```

```
##   country  year_left     imports  year_right     exports
## 0  France       2015  898.524296        2014  816.819217
## 1  France       2015  898.524296        2015  851.663257
## 2  France       2015  898.524296        2016  867.401425
```

10.14 Aggregation

Sometimes we want to aggregate the values of a variable, for example, from a quarterly to an annual dimension. With spatial observations, this can also be the case, for example, when data are available at the county level and the aggregate values at the state level.

To illustrate the different aggregation operations, let's create a dataframe with unemployment data for different French regions, departments and years:

```
unemployment = pd.DataFrame(
    {"region" : (["Bretagne"]*4 + ["Corse"]*2)*2,
     "departement" : ["Cotes-d'Armor", "Finistere",
                      "Ille-et-Vilaine", "Morbihan",
                      "Corse-du-Sud", "Haute-Corse"]*2,
     "year" : np.repeat([2011, 2010], 6),
     "workers" : [8738, 12701, 11390, 10228, 975, 1297,
                  8113, 12258, 10897, 9617, 936, 1220],
     "engineers" : [1420, 2530, 3986, 2025, 259, 254,
                    1334, 2401, 3776, 1979, 253, 241]
    })
print(unemployment)
```

```
##        region    departement  year  workers  engineers
## 0    Bretagne  Cotes-d'Armor  2011     8738       1420
## 1    Bretagne      Finistere  2011    12701       2530
```

```
## 2   Bretagne  Ille-et-Vilaine  2011    11390       3986
## 3   Bretagne         Morbihan  2011    10228       2025
## 4      Corse     Corse-du-Sud  2011      975        259
## 5      Corse      Haute-Corse  2011     1297        254
## 6   Bretagne    Cotes-d'Armor  2010     8113       1334
## 7   Bretagne        Finistere  2010    12258       2401
## 8   Bretagne  Ille-et-Vilaine  2010    10897       3776
## 9   Bretagne         Morbihan  2010     9617       1979
## 10     Corse     Corse-du-Sud  2010      936        253
## 11     Corse      Haute-Corse  2010     1220        241
```

As previously discussed (see Section 10.10.1), methods can be used to calculate simple statistics on the data set. For example, to display the average of each of the numerical columns:

```
print(unemployment.mean())
```

```
## year         2010.500000
## workers      7364.166667
## engineers    1704.833333
## dtype: float64
```

What we are interested in in this section is to perform calculations on subgroups of data. The principle is simple: first, the data are separated according to identified groups (*split*), then an operation is applied to each group (*apply*), and finally the results are collected (*combine*). To perform grouping, depending on factors before performing aggregation calculations, `pandas` offers the method `groupby()`. The argument provided is the column name(s) used to perform the groups.

10.14.1 Aggregation According to the Values of a Single Column

For example, let us assume that we want to obtain the total number of unemployed workers per year. First, we use the `groupby()` method on our dataframe, indicating that the groups must be created according to the values in the `year` column

```
print(unemployment.groupby("year"))
```

```
## <pandas.core.groupby.groupby.DataFrameGroupBy object at 0
   x129add978>
```

Then, we extract the variable `workers`:

```
print(unemployment.groupby("year").workers)
# Or
```

```
## <pandas.core.groupby.groupby.SeriesGroupBy object at 0
   x129a3cbe0>
```

```
print(unemployment.groupby("year")["workers"])
```

```
## <pandas.core.groupby.groupby.SeriesGroupBy object at 0
   x129add358>
```

And finally, we can perform the calculation on each subgroup and display the result:

```
print(unemployment.groupby("year")["workers"].sum())
```

```
## year
## 2010    43041
## 2011    45329
## Name: workers, dtype: int64
```

If we want to perform this calculation for several columns, for example `workers` and `engineers`, we just have to select *a priori* the grouping variance and the variables for which we want to perform the calculation :

```
unemployment.loc[:,["year", "workers", "engineers"]].groupby("year").sum()
```

```
##       workers  engineers
## year
## 2010    43041       9984
```

```
## 2011    45329      10474
```

10.14.2 Aggregation According to the Values of Several Columns

Now, let's assume that we want to aggregate by year and region. The only thing we need to do is to give a list containing the names of the columns used to create the different groups:

```
unemployment.loc[:,["year", "region",
             "workers", "engineers"]].groupby(["year",
                                               "region"]).sum()
```

```
##                 workers  engineers
## year region
## 2010 Bretagne     40885       9490
##      Corse         2156        494
## 2011 Bretagne     43057       9961
##      Corse         2272        513
```

10.15 Data Export and Import

pandas offers many functions for importing and exporting data in different formats.

10.15.1 Data Export

10.15.1.1 Exporting Tabular Data

10.15.1.1.1 To a CSV File {pandas-export_csv}

To export tabular data, such as those contained in a dataframe, NumPy offers the to_csv() method, which accepts many specifications. Let's look at some of them that seem to me to be the most common:

Table 10.1: Main arguments of the `to_csv` function

Argument	Description
`path_or_buf`	path to the file
`sep`	field separation character
`decimal`	character to be used for the decimal separator
`na_rep`	representation to be used for missing values
`header`	indicates whether the column names should be exported (`True` by default)
`index`	indicates whether the line names should be exported (`True` by default)
`mode`	python writing mode (see Table 5.1, by default `w`)
`encoding`	character encoding (`utf-8` by default)
`compression`	compression to be used for the destination file (`gzip`, `bz2`, `zip`, `xz`)
`line_terminator`	end of line character
`quotechar`	character used to put fields between *quotes*
`chunksize`	(integer) number of lines to be written at a time
`date_format`	date format for `datetime` objects

Let's assume that we want to export the contents of the 'unemployment' dataframe to a CSV file whose fields are separated by semicolons, and by not exporting the index :

```
unemployment = pd.DataFrame(
    {"region" : (["Bretagne"]*4 + ["Corse"]*2)*2,
     "departement" : ["Cotes-d'Armor", "Finistere",
                      "Ille-et-Vilaine", "Morbihan",
                      "Corse-du-Sud", "Haute-Corse"]*2,
     "year" : np.repeat([2011, 2010], 6),
     "workers" : [8738, 12701, 11390, 10228, 975, 1297,
                  8113, 12258, 10897, 9617, 936, 1220],
     "engineers" : [1420, 2530, 3986, 2025, 259, 254,
                    1334, 2401, 3776, 1979, 253, 241]
    })
print(unemployment)
```

```
##        region       departement   year   workers   engineers
```

```
## 0   Bretagne    Cotes-d'Armor  2011    8738        1420
## 1   Bretagne        Finistere  2011   12701        2530
## 2   Bretagne  Ille-et-Vilaine  2011   11390        3986
## 3   Bretagne         Morbihan  2011   10228        2025
## 4      Corse     Corse-du-Sud  2011     975         259
## 5      Corse      Haute-Corse  2011    1297         254
## 6   Bretagne    Cotes-d'Armor  2010    8113        1334
## 7   Bretagne        Finistere  2010   12258        2401
## 8   Bretagne  Ille-et-Vilaine  2010   10897        3776
## 9   Bretagne         Morbihan  2010    9617        1979
## 10     Corse     Corse-du-Sud  2010     936         253
## 11     Corse      Haute-Corse  2010    1220         241
```

For export:

```
path = "./fichiers_exemples/unemployment.csv"
unemployment.to_csv(path, decimal=";", index=False)
```

If you want the CSV file to be compressed into a `gzip` file, you name it with the extension `.csv.gz` and add the value `gzip` to the `compression` parameter:

```
path = "./fichiers_exemples/chomage.csv.gz"
unemployment.to_csv(path, decimal=";", index=False, compression="gzip")
```

10.15.1.1.2 To an HDF5 File

To save the data of a dataframe in an HDF5 file using HDFStore, `pandas` offers the method `to_hdf()` which works in the same way as the function `to_csv()` (see Section @ref(pandas-export_csv)).

The argument `path_or_buf` must be specified to indicate the path and the argument `key` to identify the object to be saved in the file.

The syntax is as follows:

```
path = "./fichiers_exemples/chomage.h5"
unemployment.to_hdf(path, "base_chomage", decimal=";", index=False)
```

10.16 Data Import

`pandas` offers many functions for importing data. In this version of the course notes, we will discuss 3: `read_csv()`, to read CSV files; `read_excel()`, to read Excel files; and `read_hdf()` to read HDF5 files.

10.16.1 CSV Files

To import data from a CSV file, `pandas` offers the function `read_csv()`:

```
path = "./fichiers_exemples/unemployment.csv"
unemployment = pd.read_csv(path, decimal=";")
```

It is possible to provide a URL pointing to a CSV file as a path, the `read_csv()` function.

Among the parameters that are frequently used:

- `sep`, `delimiter`: field separator
- `decimal`: decimal separator
- `header`: line number(s) to be used as data header
- `skiprows`: line number(s) to be skipped at the beginning
- `skipfooter`: line number(s) to be skipped at the end
- `nrows`: number of lines to read
- `na_values`: additional character strings to be considered as missing values (in addition to `#N/A`, `#N/A N/A`, `#NA`, `-1.#IND`, `-1.#QNAN`, `-NaN`, `-nan`, `1.#IND`, `1.#QNAN`, `N/A`, `NA`, `NULL`, `NaN`, `n/a`, `nan`, `null`)
- `quotechar`: quote character
- `encoding`: character encoding (`utf-8` by default).

10.16.2 Excel Files

To import Excel files, `pandas` offers the function `read_excel()`.

```
path = "./fichiers_exemples/chomage.xlsx"
unemployment = pd.read_excel(path, skiprows=2, header=1, sheet = 1)
print(unemployment)
```

```
##       Bretagne     Cotes-d'Armor  2011   8738  1420
## 0     Bretagne         Finistere  2011  12701  2530
## 1     Bretagne   Ille-et-Vilaine  2011  11390  3986
## 2     Bretagne          Morbihan  2011  10228  2025
## 3        Corse      Corse-du-Sud  2011    975   259
## 4        Corse       Haute-Corse  2011   1297   254
## 5     Bretagne     Cotes-d'Armor  2010   8113  1334
## 6     Bretagne         Finistere  2010  12258  2401
## 7     Bretagne   Ille-et-Vilaine  2010  10897  3776
## 8     Bretagne          Morbihan  2010   9617  1979
## 9        Corse      Corse-du-Sud  2010    936   253
## 10       Corse       Haute-Corse  2010   1220   241
```

Among the frequently used arguments:

- `header`: row number to be used as header
- `sheet`: name or sheet number
- `skiprows`: number of lines to be skipped at the beginning
- `thousands`: thousands separator.

10.16.3 HDF5 Files

```
path = "./fichiers_exemples/chomage.h5"
print(pd.read_hdf(path, "base_chomage"))
```

10.17 Exercise

Exercise 1: Import and export

1. Download the csv file by hand at the following address: http://egallic.fr/Enseignement/Python/Exercices/donnees/notes.csv and place it in the current directory. Import its content into Python.
2. Import the data back into Python, but this time by providing the url directly to the import function.

3. Now, import the contents of the file available at http://egallic.fr/Enseignement/Python/Exercices/donnees/notes_decim.csv. The field separator is a semicolon and the decimal separator is a comma.
4. Import the contents of the file http://egallic.fr/Enseignement/Python/Exercices/donnees/notes_h.csv. The column names are not present.
5. Importer le contenu du fichier http://egallic.fr/Enseignement/Python/Exercices/donnees/notes_h_s.csv. The first line is not to be imported.
6. Import the contents of the first sheet of the Excel file http://egallic.fr/Enseignement/Python/Exercices/donnees/notes.xlsx.
7. Import the content of the second sheet (`notes_h_s`) from the Excel file available here: http://egallic.fr/Enseignement/Python/Exercices/donnees/notes.xlsx. The first line is a comment not to be considered during the import.
8. Export the content of the object containing the notes of the previous question in csv format (comma as field separator, dot as decimal separator, do not keep the line numbers).

Exercise 2: Handling Dataframes

1. Using the `read_excel()` function of the `pandas` library, import the contents of the sheet entitled `notes_2012` from the Excel file available at the following address: http://egallic.fr/Enseignement/Python/Exercices/donnees/notes_etudiants.xlsx and store it in a variable called notes_2012.
2. Display the first 6 lines of the dataset, then the dimensions of the table.
3. Keep only the column `note_stat` of the data table `notes_2012` in an object called `tmp`.
4. Keep only the columns `number_student`, `note_stat` and `note_macro` in an object named `tmp`.
5. Replace the content of `tmp` with observations of `notes_2012` for which the individual obtained a stat score greater (strictly) than 10.
6. Replace the content of tmp with observations of `notes_2012` for which the individual obtained a stats score within the interval (10, 15).
7. Look for duplicates in the `notees_2012` data table; if so, remove them from the table.
8. Display the type of data in the column `num_etudiant`, then display the type of all note columns_2012.
9. Add the following columns to the `notes_2012` table:

(a) `note_stat_maj`: the stat score (`note_stat`) increased by one point,
(b) `note_macro_maj`: the macro note (`note_macro`) plus three points (do it in two

steps: first two more points, then one point).

10. Rename the column year to year.
11. From the file `notes_etudiants.xlsx` (see question 1), import the contents of the sheets `notes_2013`, `notes_2014` and `prenoms` and store them in the objects `grades_2013`, `grades_2014` and `first_names`, respectively.
12. Stack the contents of the data tables `grades_2012`, `grades_2013` and `grades_2014` in an object that will be called `grades`.
13. Merge the `grades` and `first_names` tables using a left join, so as to add the information contained in the first name table to the grade observations. The join must be done by the student number and the year, the final object will replace the content of the grades
14. Sort the `grades` table by increasing years and decreasing macro grades
15. Create a column `after_2012` which takes the value `True` if the observation concerns a score assigned after 2012.
16. By grouping on the `grades` dataframe compute:

(a) the annual mean and standard deviation of the scores for each of the two classes,
(b) the annual and gender average and standard deviation of the grades for each of the two classes.

Chapter 11

Data Visualization

In this chapter, we will explore the basics of data visualization with the `Matplotlib` library.

For the moment, these notes will not talk about the dynamic graphics that can be created in connection with JavaScript libraries, such as D3.js. In a future version, we might see how to make graphs with `seaborn`. (https://seaborn.pydata.org/).

To have quick access to a type of graphic that you want to create, you can refer to this excellent gallery: https://python-graph-gallery.com/.

Fans of the R software and programming language will be happy to find a graphical grammar like the one proposed by the[R package `ggplot2`] (https://ggplot2.tidyverse.org/) introduced by Hadley Wickham. Indeed, there is a Python library called `ggplot` : http://ggplot.yhathq.com/.

To graphically explore one' s data, it may be interesting to invest some time in the `Altair` library (https://altair-viz.github.io/). For a short video introduction, see: https://www.youtube.com/watch?v=aRxahWy-ul8.

11.1 Graphics with Matplotlib

To use the features offered by `matplotlib` (https://matplotlib.org/), some modules need to be loaded. The most common is the submodule `pyplot`, to which the alias `plt` is frequently assigned:

```
import matplotlib.pyplot as plt
```

To make a graph with the `pyplot` function, we first create a figure by defining its range, then add and/or modify its elements using the functions offered by `pyplot`.

To illustrate the features of `matplotlib`, we will need to generate values, using the `numpy` library, which we load:

```
import numpy as np
```

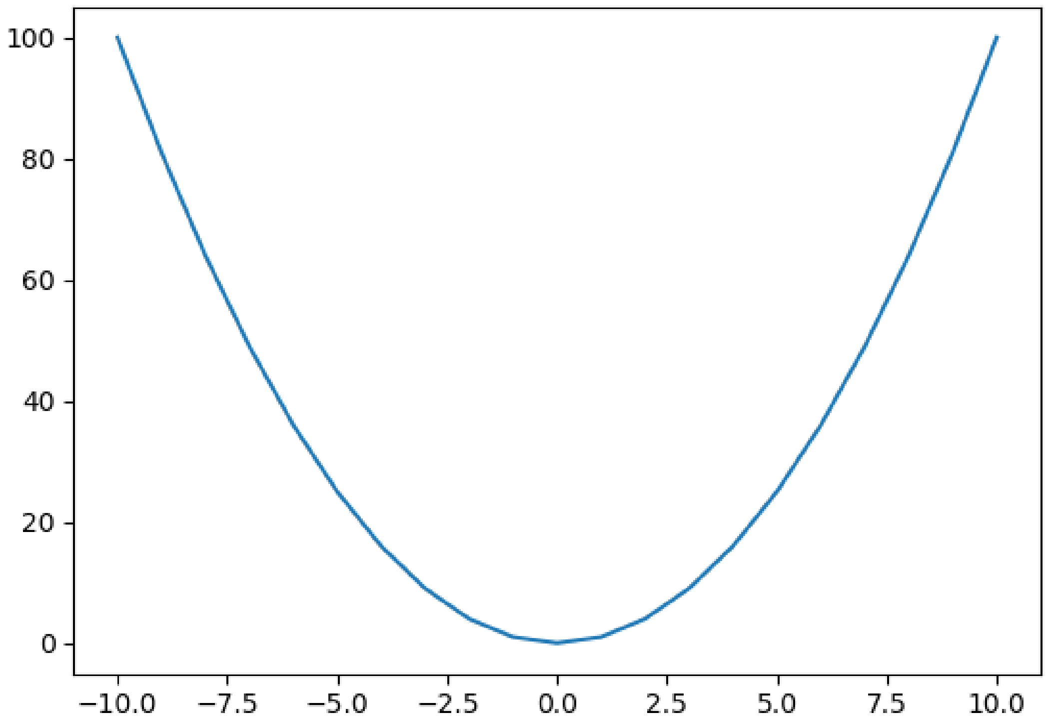
11.1.1 Geometries

11.1.1.1 Lines

To draw lines on a Cartesian coordinate system, we can use the `plot()` function, to which we provide the coordinates of the x-axis and y-axis as the first two arguments, respectively. The third argument defines the geometry.

By default, the geometry is a curve:

```
x = np.arange(-10, 11)
y = x**2
plt.plot(x, y)
```

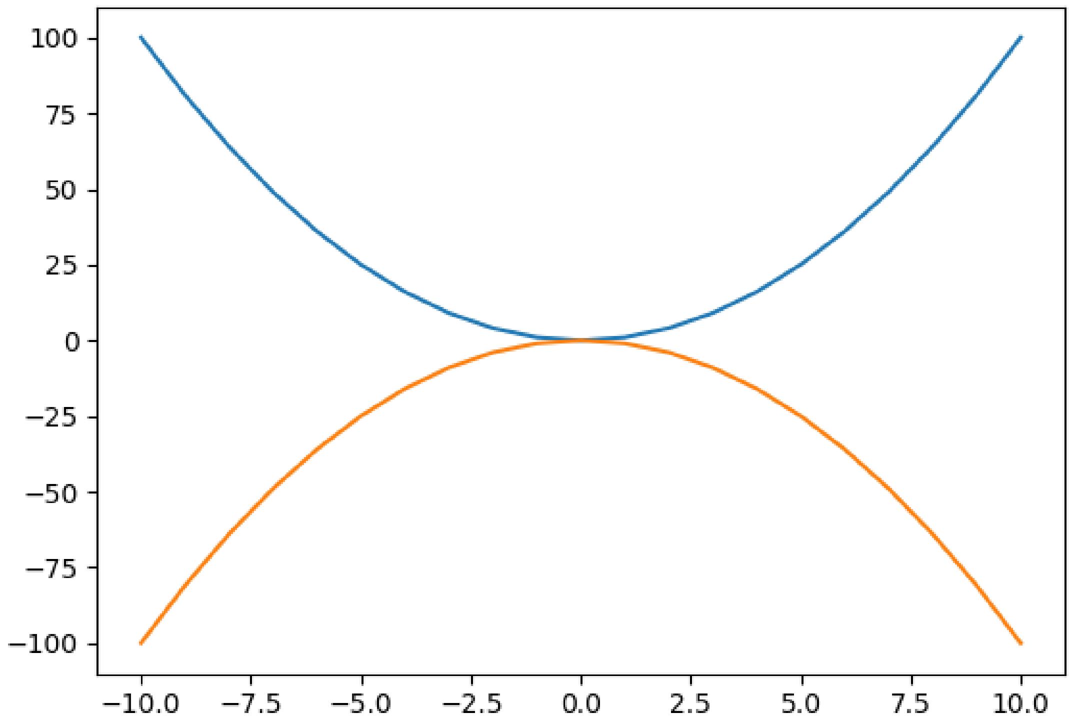


Once the graph is displayed, it can be closed with the `close()` function:

Similarly, the geometry can be specified as follows:

To add a curve to the graph, the `plot()` function is used several times:

```
y_2 = -x**2
plt.plot(x, y, "-")
plt.plot(x, y_2, "-")
```

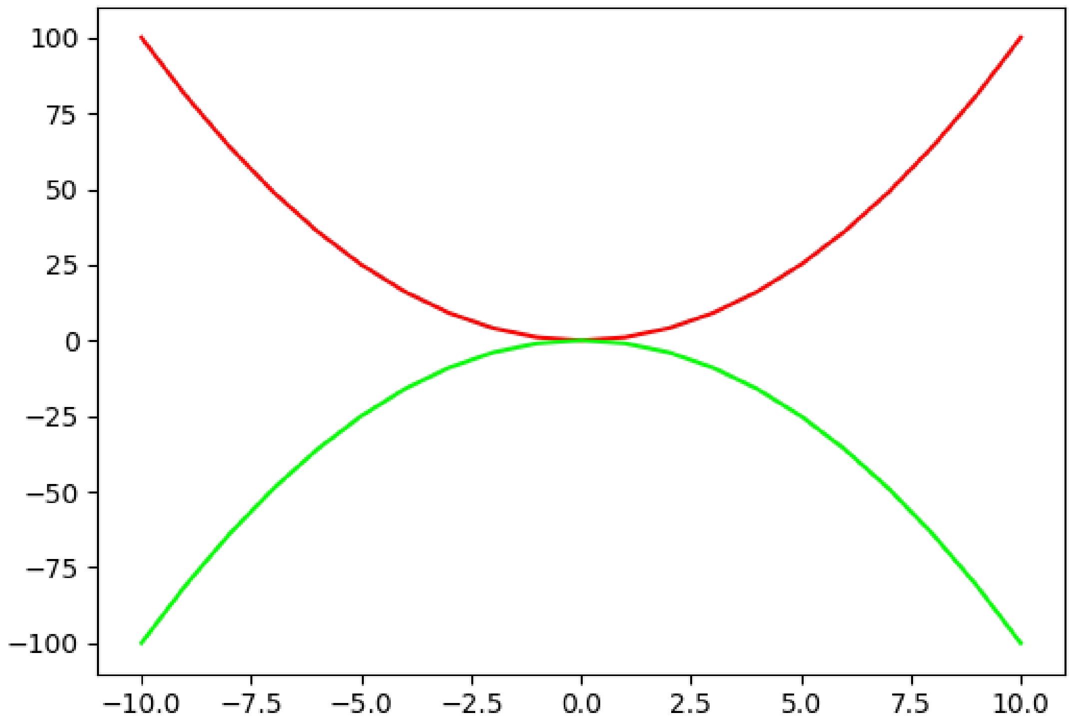


11.1.1.1.1 Aesthetic Arguments

11.1.1.1.1.1 Line Color

To change the color of a line, the argument `color` is used:

```
plt.plot(x, y, color="red")
plt.plot(x, y_2, color="#00FF00")
```



As can be seen, the reference to color can be done by using its name (the list of colors with a name is available on the matplotlib documentation). A hexadecimal code can also be used to refer to a color.

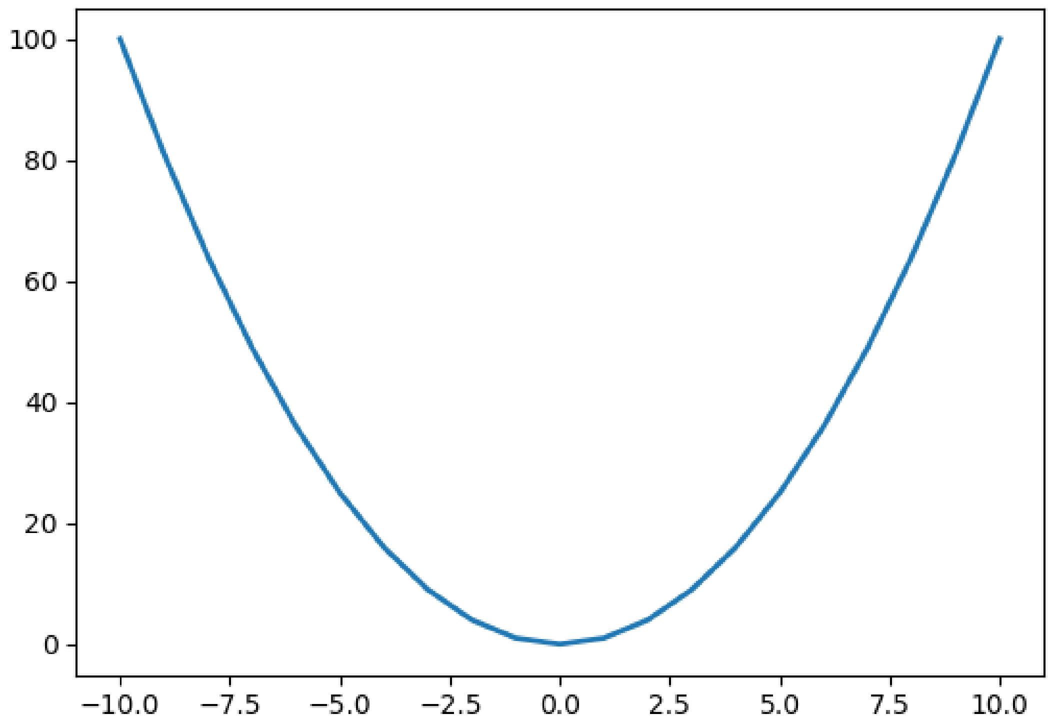
Remark 11.1.1

It may be interesting, when selecting colours, to think about the use afterwards: is it for the screen? For grayscale printing? It is also questionable whether the choice made will not hinder understanding for colourblind people (who represent about 4% of the French population). The Color Brewer website offers colour choices based on desired characteristics such as those mentioned.

11.1.1.1.1.2 Line Thickness

Line thickness can be modified using the `linewidth` argument, to which a numerical value is provided:

```
plt.plot(x, y, linewidth = 2)
```



11.1.1.1.1.3 Type of lines

To change the line type, the third argument of the function is modified. As we have seen on the previous example the default is a line. This corresponds to the value `-` being specified to the third argument. Table 11.1 indicates the different possible formats for the line.

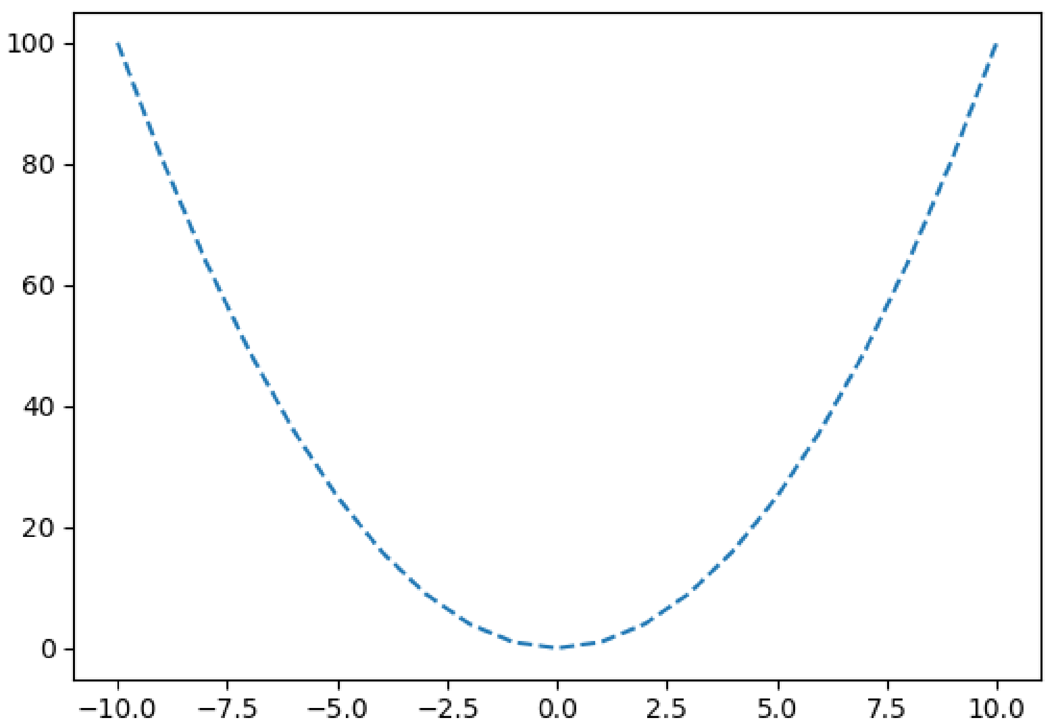
Table 11.1: Line formats

Value	Description
`-`	Solid line

Value	Description
`--`	Dashed line
`-.`	Dots and dashs
`:`	Dots

For example, to have a linear interpolation between our points with a graphical representation made using dashes :

```
plt.plot(x, y, "--")
```

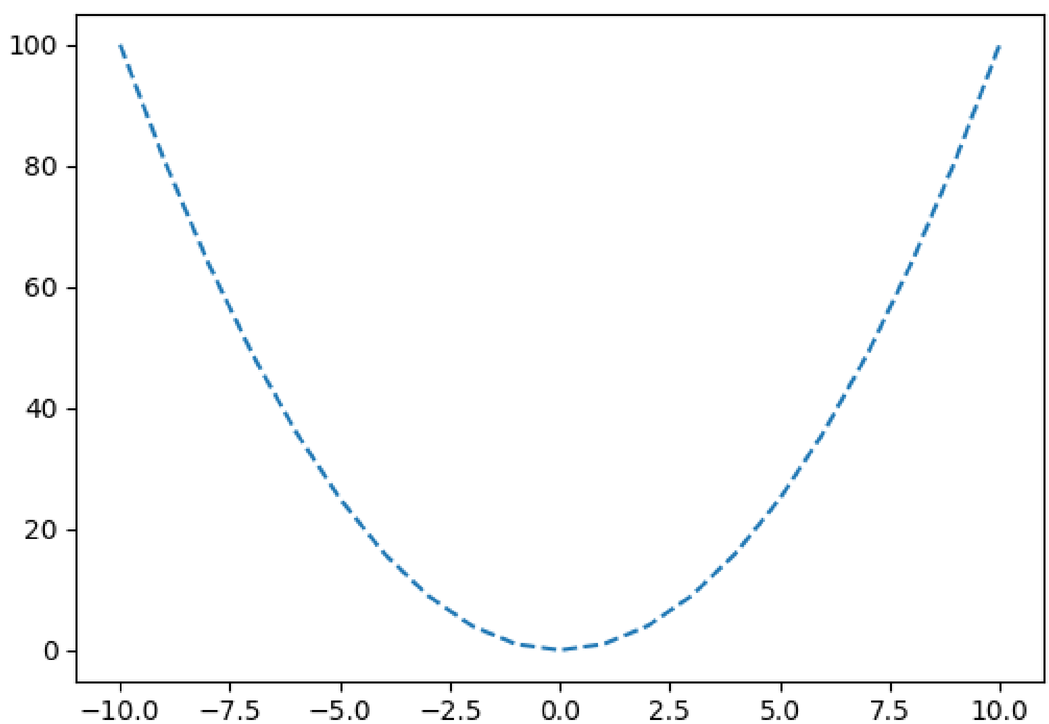


The line type can also be specified using the `linestyle` parameter, by indicating one of the values given in Table 11.2

Table 11.2: Line formats via the `linestyle` argument

Value	Description
`-` ou `solid`	Solid line
`--` ou `dashed`	Dashed
`-.` ou `dashdot`	Dashes and dots
`:` ou `dotted`	Dots
`None`	No line plotted

```
plt.plot(x, y, linestyle="dashed")
```



11.1.1.1.1.4 Markers

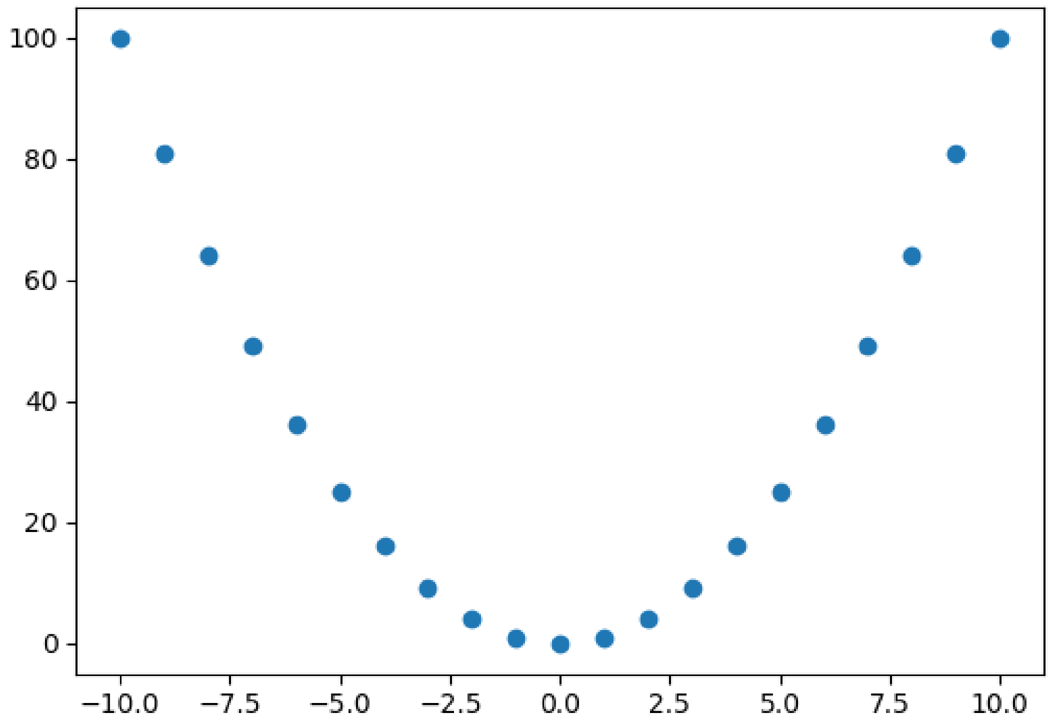
Table 11.3 shows formats that can be specified as markers at each point on the curve.

Table 11.3: Line formats

Value	Description
`.`	Dots
`,`	Pixels
`o`	Empty circles
`v`	Triangles pointing down
`^`	Triangles pointing up
`<`	Triangles pointing to the left
`>`	Triangles pointing to the right
`1`	'tri_down'
`2`	'tri_up'
`3`	'tri_left'
`4`	'tri_right'
`s`	Square
`p`	Pentagon
`*`	Asterisk
`h`	Hexagon 1
`H`	Hexagon 2
`+`	Plus symbol
`x`	Multiply symbol
`D`	Diamond
`d`	Thin diamond
`|`	Vertical line
`_`	Horizontal line

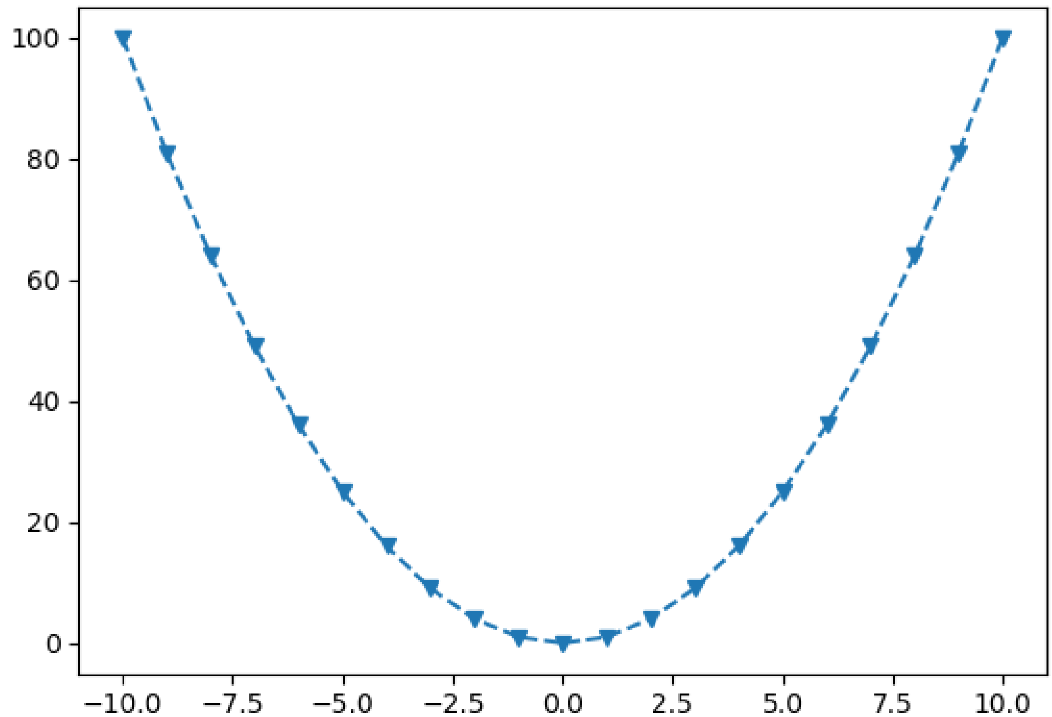
For example, with empty circles:

```
plt.plot(x, y, "o")
```



It can be noted that it is possible to combine the line types in 11.1 with marker types described in Table 11.3 :

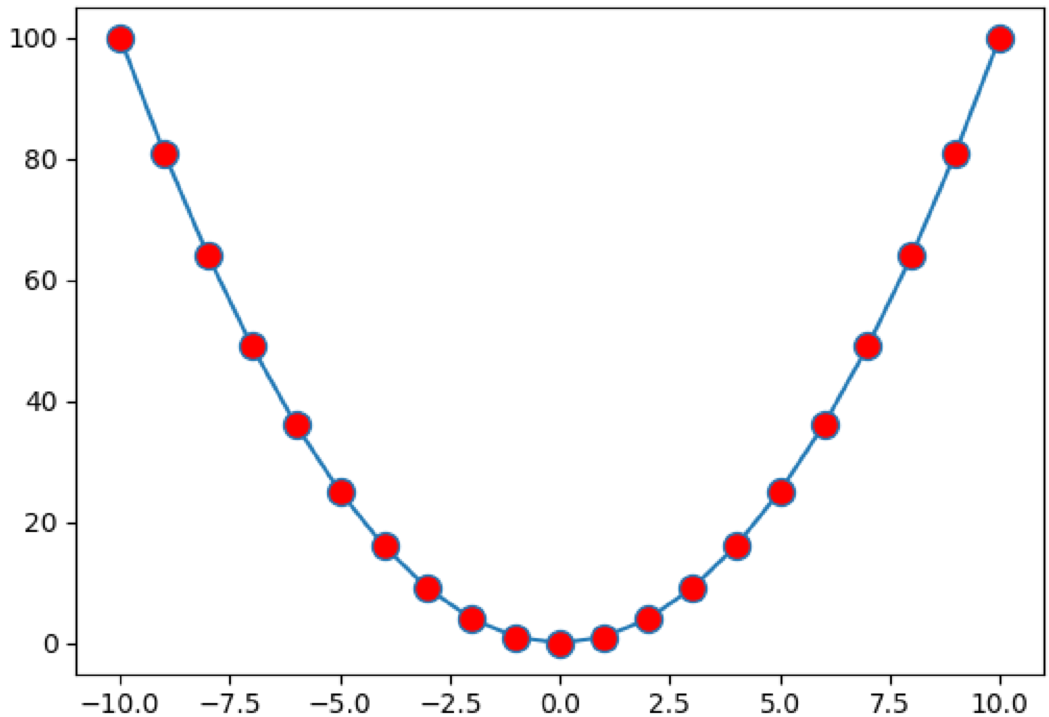
```
plt.plot(x, y, "--v")
```



To control the markers more precisely, the following arguments can be used:

- `marker`: indicates the type of marker (see Table 11.3)
- `markerfacecolor`: the desired color for the markers
- `markersize`: size of the markers.

```
plt.plot(x, y, marker="o", markerfacecolor = "red", markersize = 10)
```

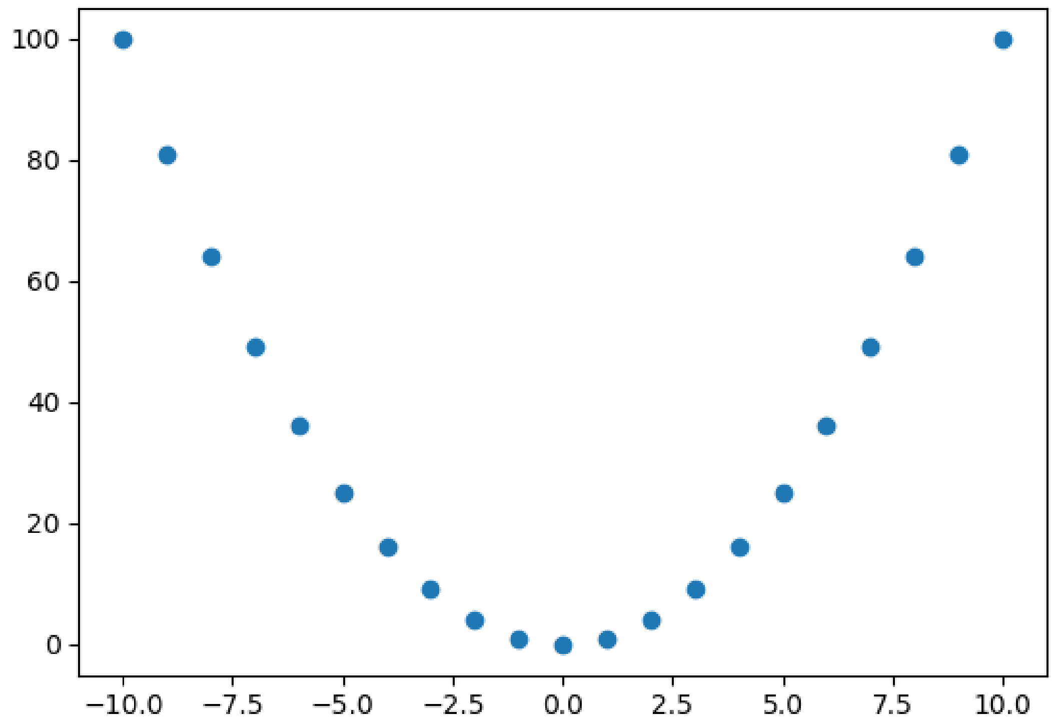


11.1.1.2 Scatter Plots

One of the graphs that we very frequently encounter is the scatterplot. To create one, we can use the `scatter()` function, which indicates the coordinates (x,y) of the points as well as some optional shape or aesthetic parameters.

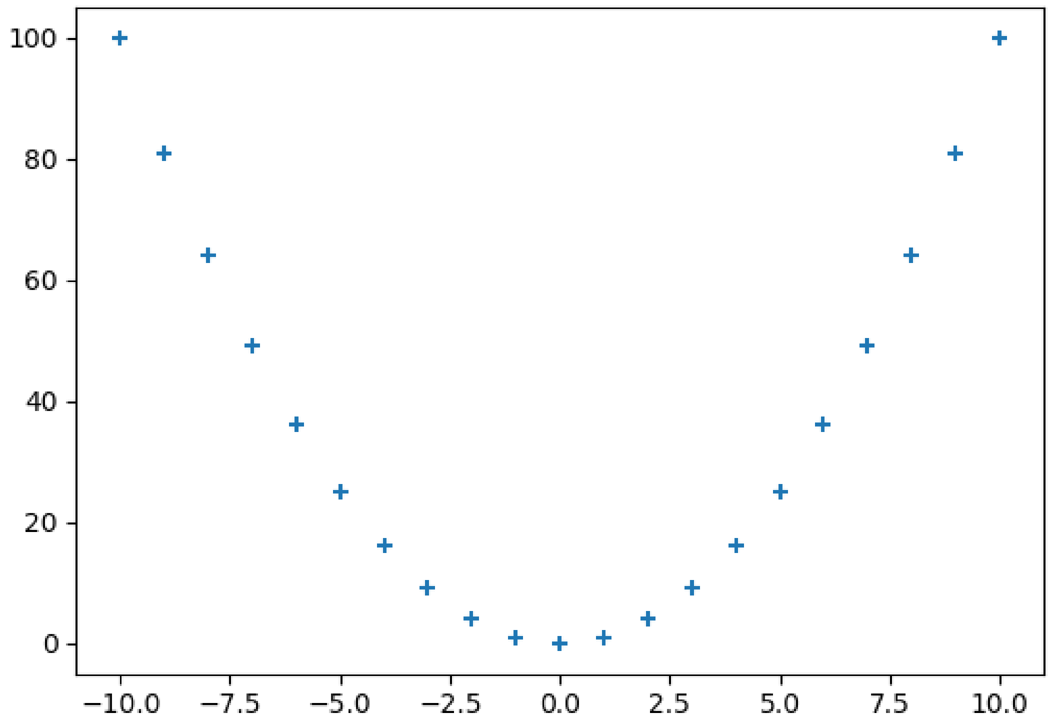
> **Remark 11.1.2**
>
> The online documentation of the `scatter()` function mentions that the `plot()` function (see Section 11.1.1.1) s faster to perform scatter plots in which the color or size of the points varies.

```
x = np.arange(-10, 11)
y = x**2
plt.scatter(x, y)
```



When we wish to change the shape of the markers, we specify it via the `marker` argument (see Table 11.3 for the possible values) :

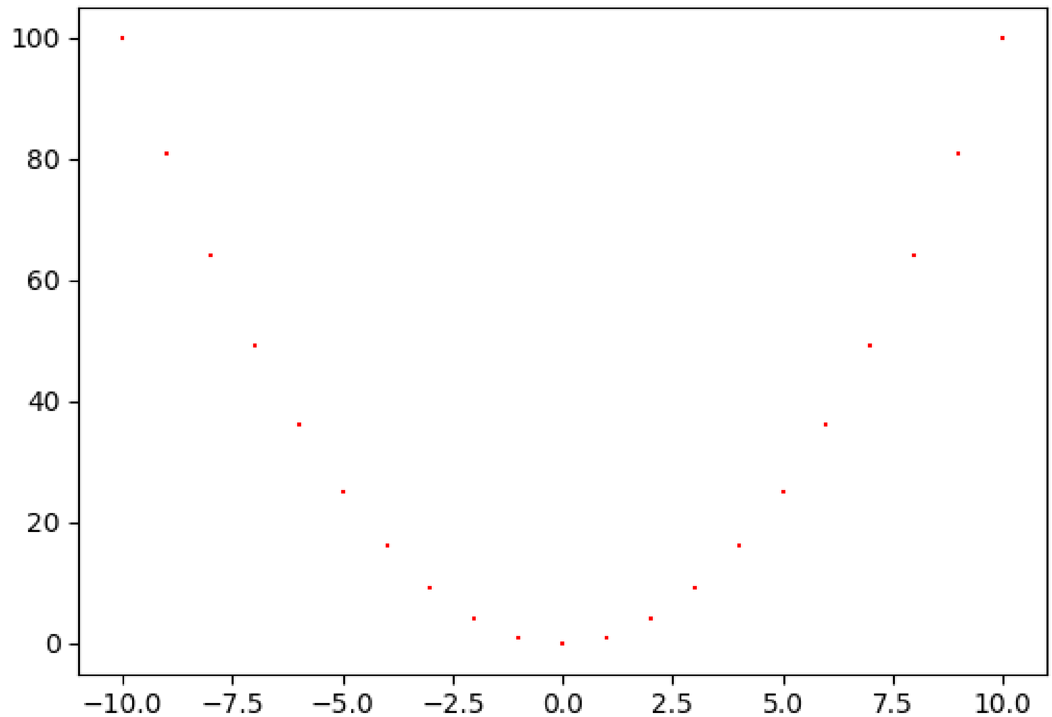
```
x = np.arange(-10, 11)
y = x**2
plt.scatter(x, y, marker="+")
```



11.1.1.3 Size and color

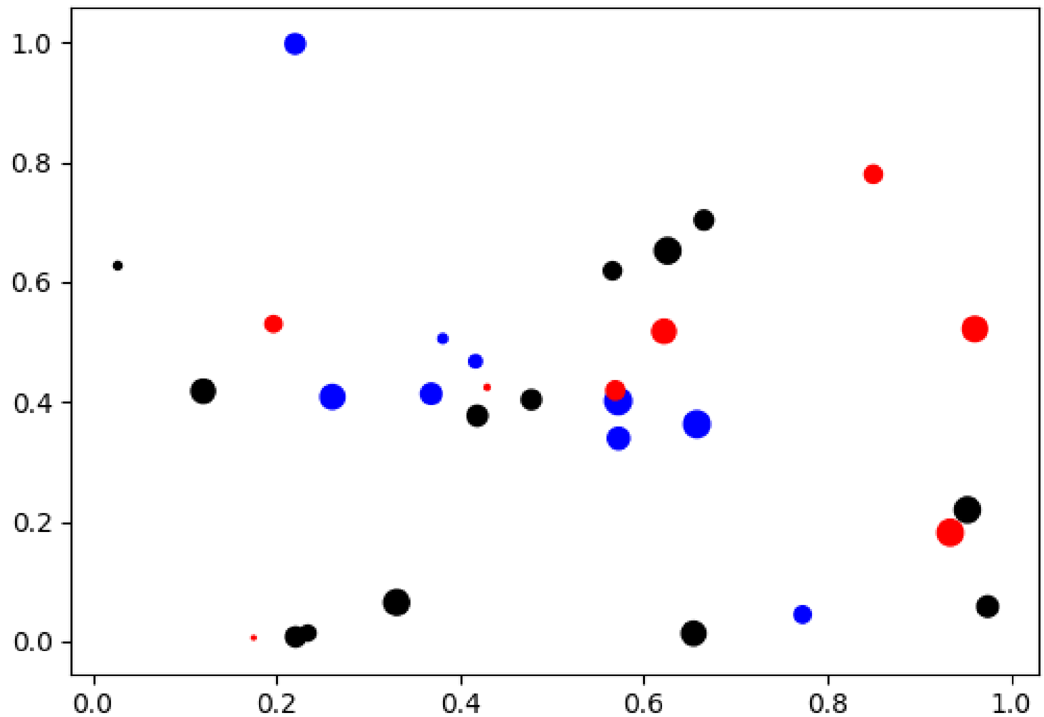
The size of the points is adjustable via the parameter `s`, while the color changes via the parameter `color` (or by its alias `c`):

```
x = np.arange(-10, 11)
y = x**2
plt.scatter(x, y, marker="+", color = "red", s = 2)
```



A specific colour and size can be associated with each point:

```
x = np.random.rand(30)
y = np.random.rand(30)
z = np.random.rand(30)
colours = np.random.choice(["blue", "black", "red"], 30)
plt.scatter(x, y, marker="o", color = colours, s = z*100)
```

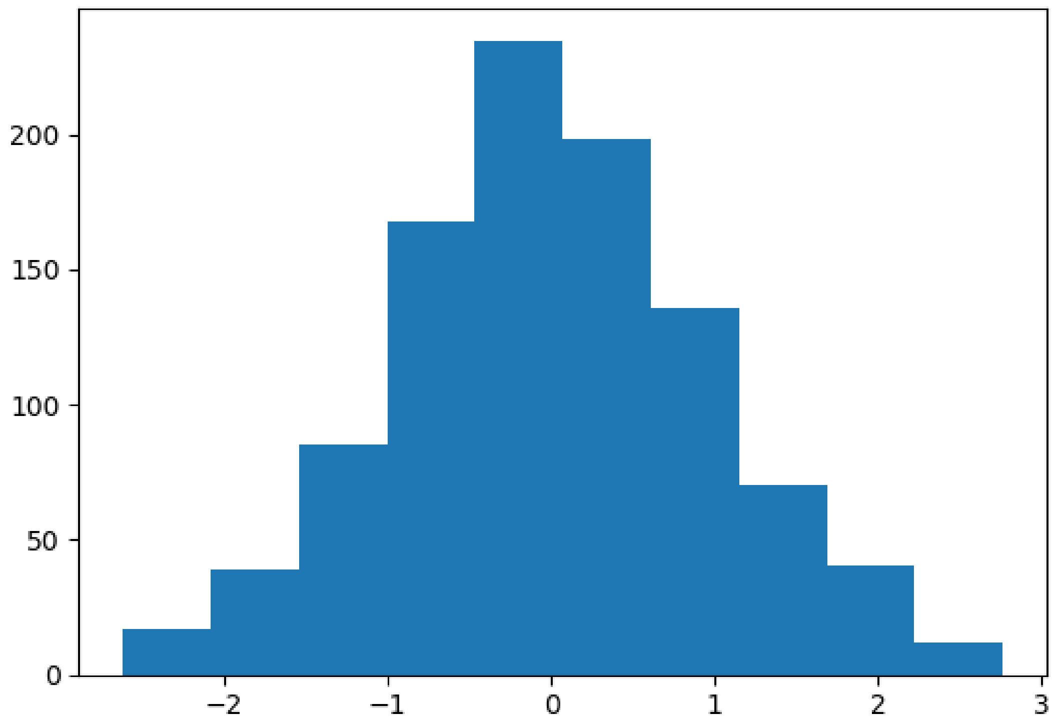


11.1.1.4 Histograms

To make a histogram with `pyplot`, the function `hist()` can be used:

```
x = np.random.randn(1000)
plt.hist(x)
```

```
## (array([ 17.,  39.,  85., 168., 235., 198., 136.,  70.,
   40.,  12.]), array([-2.62300428, -2.08431945, -1.54563461,
   -1.00694978, -0.46826494,
##         0.0704199 ,  0.60910473,  1.14778957,  1.68647441,
    2.22515924,
##         2.76384408]), <a list of 10 Patch objects>)
```

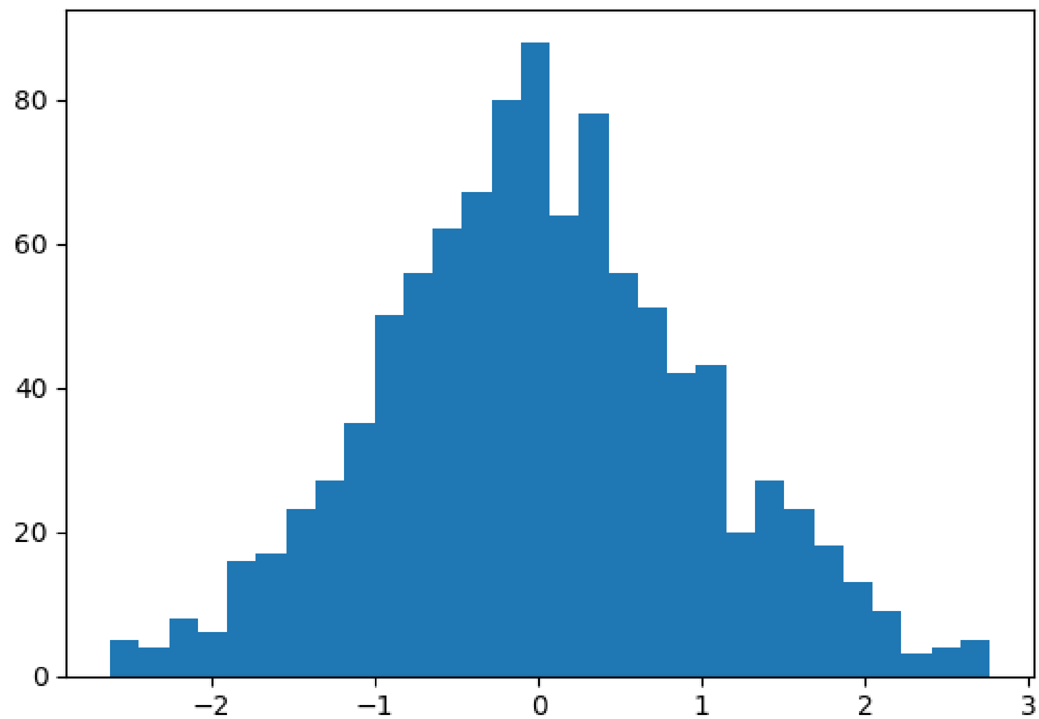


The `bins` argument is used to specify either the number of classes or their boundaries:

```
plt.hist(x, bins=30)
```

```
## (array([ 5.,  4.,  8.,  6., 16., 17., 23., 27., 35., 50.,
   56., 62., 67.,
##        80., 88., 64., 78., 56., 51., 42., 43., 20., 27.,
   23., 18., 13.,
##         9.,  3.,  4.,  5.]), array([-2.62300428,
   -2.44344267, -2.26388106, -2.08431945, -1.90475784,
##        -1.72519622, -1.54563461, -1.366073  , -1.18651139,
   -1.00694978,
##        -0.82738816, -0.64782655, -0.46826494, -0.28870333,
   -0.10914171,
##         0.0704199 ,  0.24998151,  0.42954312,  0.60910473,
    0.78866635,
```

```
##          0.96822796,  1.14778957,  1.32735118,  1.50691279,
    1.68647441,
##          1.86603602,  2.04559763,  2.22515924,  2.40472086,
    2.58428247,
##          2.76384408]), <a list of 30 Patch objects>)
```



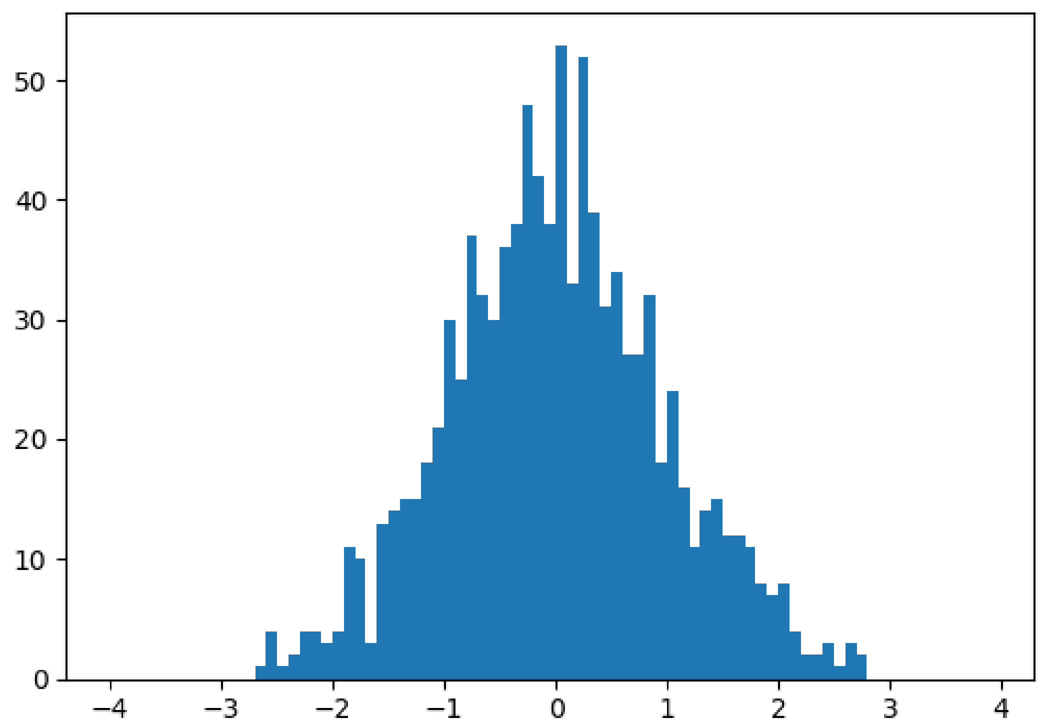
And with the boundaries:

```
bins = np.arange(-4, 4, .1)
plt.hist(x, bins=bins)
```

```
## (array([ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,
   0.,  0.,  0.,
##          1.,  4.,  1.,  2.,  4.,  4.,  3.,  4., 11., 10.,
```

```
    3., 13., 14.,
##          15., 15., 18., 21., 30., 25., 37., 32., 30., 36.,
    38., 48., 42.,
##          38., 53., 33., 52., 39., 31., 34., 27., 27., 32.,
    18., 24., 16.,
##          11., 14., 15., 12., 12., 11.,  8.,  7.,  8.,  4.,
    2.,  2.,  3.,
##           1.,  3.,  2.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,
    0.,  0.,  0.,
##           0.]), array([-4.00000000e+00, -3.90000000e+00,
    -3.80000000e+00, -3.70000000e+00,
##          -3.60000000e+00, -3.50000000e+00, -3.40000000e+00,
    -3.30000000e+00,
##          -3.20000000e+00, -3.10000000e+00, -3.00000000e+00,
    -2.90000000e+00,
##          -2.80000000e+00, -2.70000000e+00, -2.60000000e+00,
    -2.50000000e+00,
##          -2.40000000e+00, -2.30000000e+00, -2.20000000e+00,
    -2.10000000e+00,
##          -2.00000000e+00, -1.90000000e+00, -1.80000000e+00,
    -1.70000000e+00,
##          -1.60000000e+00, -1.50000000e+00, -1.40000000e+00,
    -1.30000000e+00,
##          -1.20000000e+00, -1.10000000e+00, -1.00000000e+00,
    -9.00000000e-01,
##          -8.00000000e-01, -7.00000000e-01, -6.00000000e-01,
    -5.00000000e-01,
##          -4.00000000e-01, -3.00000000e-01, -2.00000000e-01,
    -1.00000000e-01,
##           3.55271368e-15,  1.00000000e-01,  2.00000000e-01,
    3.00000000e-01,
##           4.00000000e-01,  5.00000000e-01,  6.00000000e-01,
    7.00000000e-01,
##           8.00000000e-01,  9.00000000e-01,  1.00000000e+00,
    1.10000000e+00,
##           1.20000000e+00,  1.30000000e+00,  1.40000000e+00,
    1.50000000e+00,
##           1.60000000e+00,  1.70000000e+00,  1.80000000e+00,
    1.90000000e+00,
##           2.00000000e+00,  2.10000000e+00,  2.20000000e+00,
    2.30000000e+00,
```

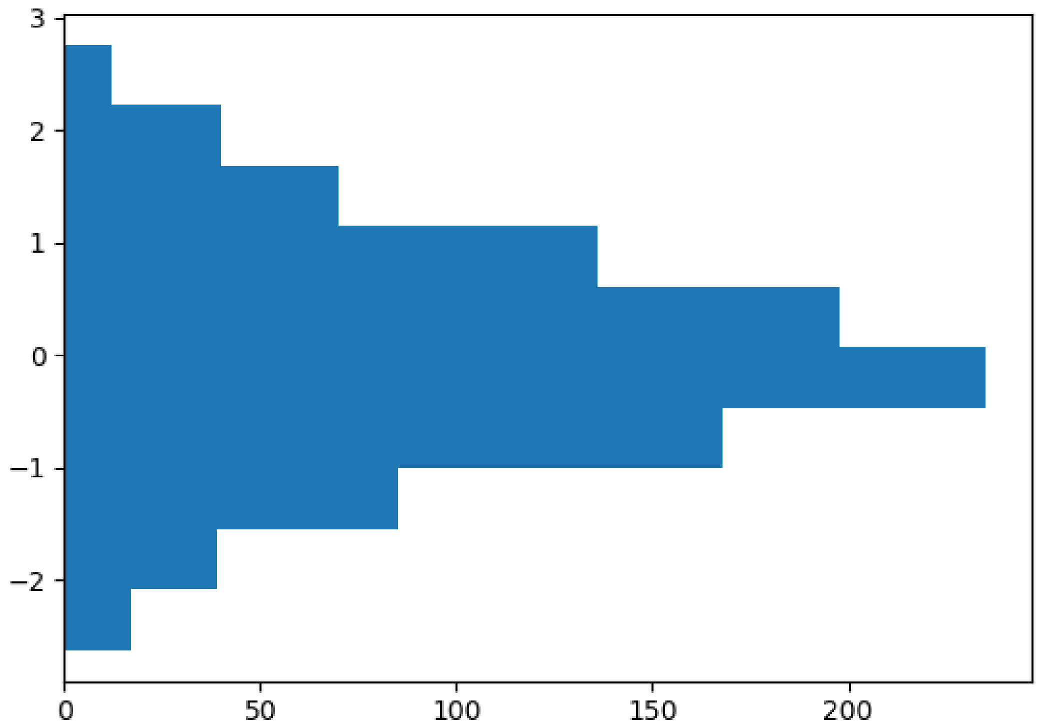
```
##          2.40000000e+00,  2.50000000e+00,  2.60000000e+00,
   2.70000000e+00,
##          2.80000000e+00,  2.90000000e+00,  3.00000000e+00,
   3.10000000e+00,
##          3.20000000e+00,  3.30000000e+00,  3.40000000e+00,
   3.50000000e+00,
##          3.60000000e+00,  3.70000000e+00,  3.80000000e+00,
   3.90000000e+00]), <a list of 79 Patch objects>)
```



The orientation is changed via the `orientation` argument, indicating either `vertical` (by default) or `horizontal`.

```
plt.hist(x, orientation='horizontal')
```

```
## (array([ 17.,  39.,  85., 168., 235., 198., 136.,  70.,
   40.,  12.]), array([-2.62300428, -2.08431945, -1.54563461,
   -1.00694978, -0.46826494,
##          0.0704199 ,  0.60910473,  1.14778957,  1.68647441,
    2.22515924,
##          2.76384408]), <a list of 10 Patch objects>)
```

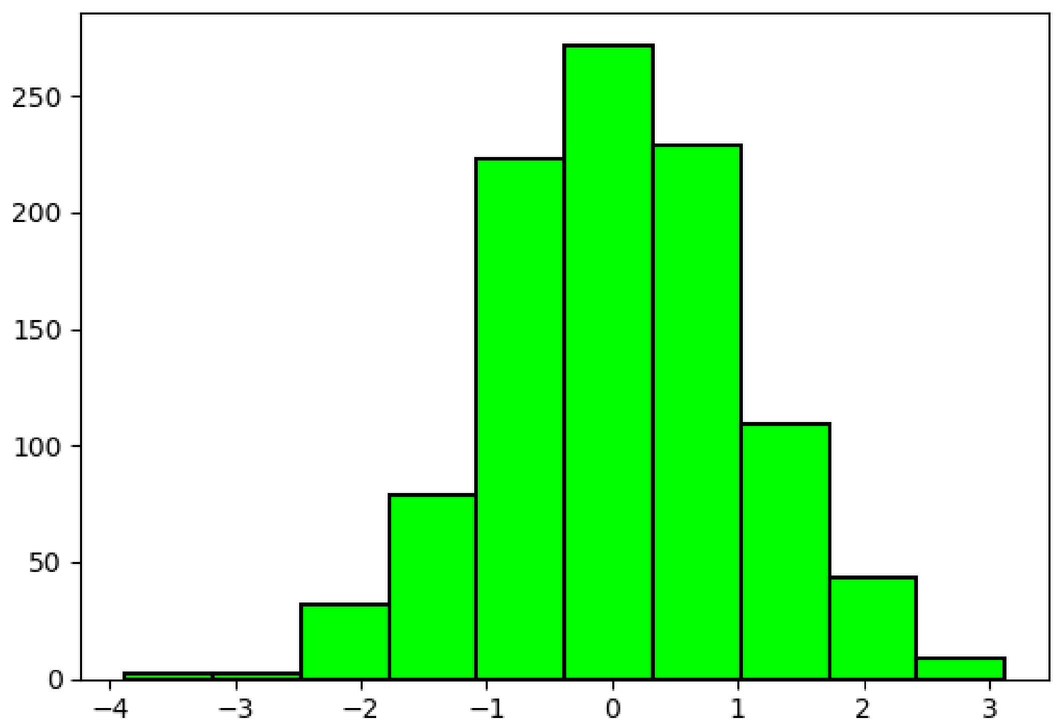


11.1.1.4.1 Aesthetic Arguments

To change the filling color, we use the `color` argument; to add a color delimiting the bars, we use the `edgecolor` argument; to define the contour thickness, we use the `linewidth` argument:

```
x = np.random.randn(1000)
plt.hist(x, color = "#00FF00", edgecolor='black', linewidth=1.5)
```

```
## (array([  2.,   2.,  32.,  79., 223., 272., 229., 109.,
   43.,   9.]), array([-3.88089841, -3.18024506, -2.4795917 ,
   -1.77893834, -1.07828499,
##        -0.37763163,  0.32302172,  1.02367508,  1.72432844,
    2.42498179,
##         3.12563515]), <a list of 10 Patch objects>)
```

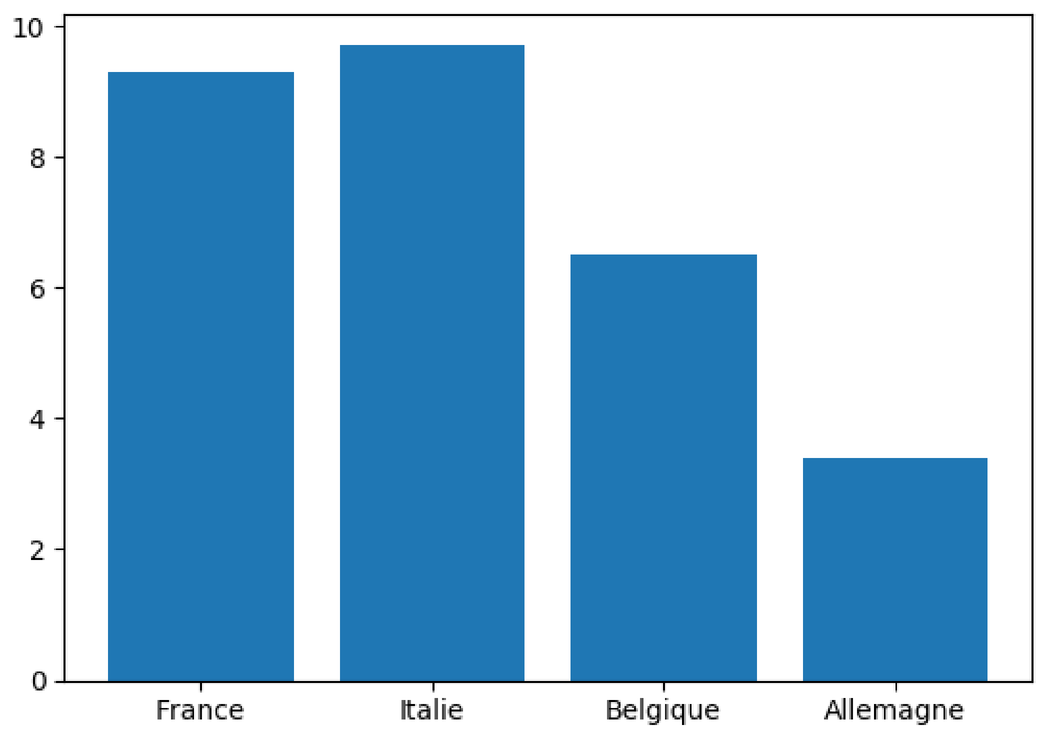


11.1.1.5 Bar Charts

To make bar graphs, `pyplot` offers the function `bar()`.

```
pays = ["France", "Italie", "Belgique", "Allemagne"]
unemployment = [9.3, 9.7, 6.5, 3.4]
plt.bar(pays, unemployment)
```

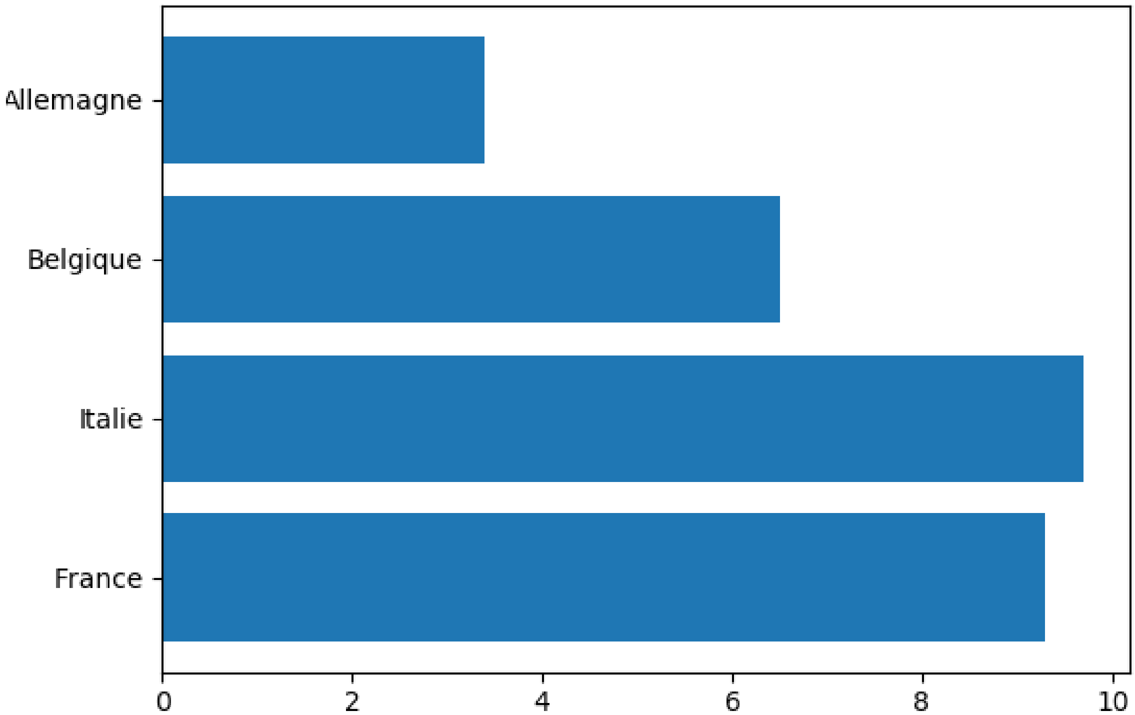
```
## <BarContainer object of 4 artists>
```



For a horizontal diagram, the `barh()` function is used in the same way:

```
plt.barh(pays, unemployment)
```

```
## <BarContainer object of 4 artists>
```



11.1.1.5.1 Several Series on a Bar Chart

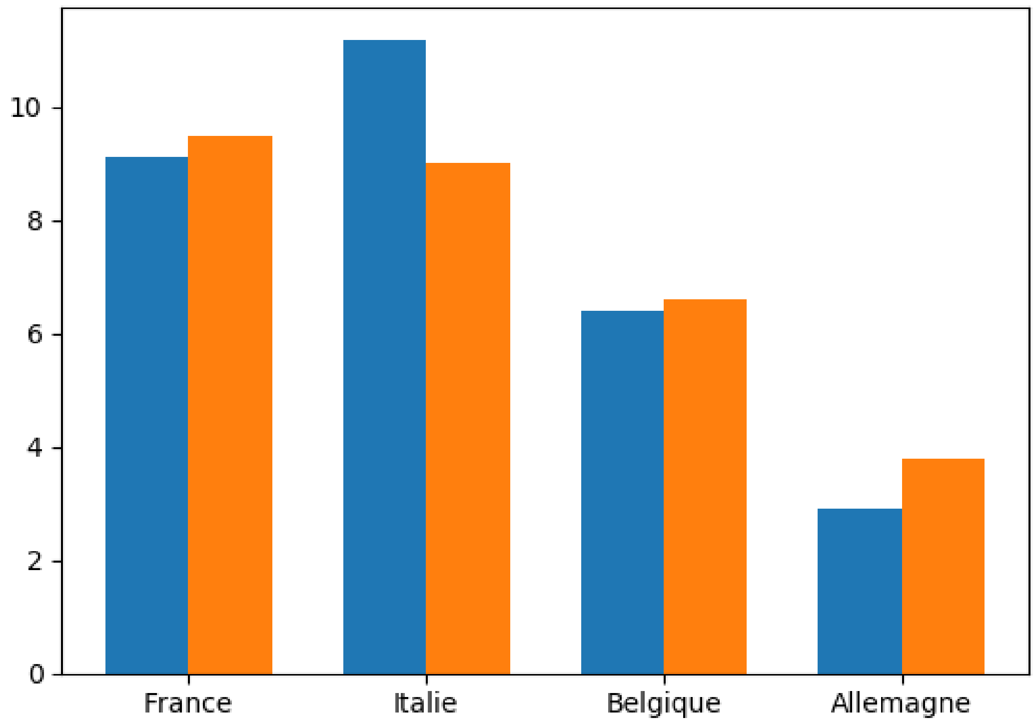
To compare several side-by-side series, it is necessary to borrow concepts that will only be introduced in Section 11.1.3 (the code is provided here rather as a quick reference to perform this kind of graphs).

```
countries = ["France", "Italie", "Belgique", "Allemagne"]
unemp_f = [9.1, 11.2, 6.4, 2.9]
unemp_h = [9.5, 9, 6.6, 3.8]
# Position on the x-axis for each label
position = np.arange(len(countries))
```

```
# Bar widths
width = .35

# Creating the figure and a set of subgraphics
fig, ax = plt.subplots()
r1 = ax.bar(position - width/2, unemp_f, width)
r2 = ax.bar(position + width/2, unemp_h, width)

# Modification of the marks on the x-axis and their labels
ax.set_xticks(position)
ax.set_xticklabels(countries)
```

```
## [<matplotlib.axis.XTick object at 0x134e50e10>, <matplotlib
   .axis.XTick object at 0x134e50748>, <matplotlib.axis.XTick
   object at 0x134e504a8>, <matplotlib.axis.XTick object at 0
   x134e85c88>]
```

```
## [Text(0,0,'France'), Text(0,0,'Italie'), Text(0,0,'Belgique
   '), Text(0,0,'Allemagne')]
```



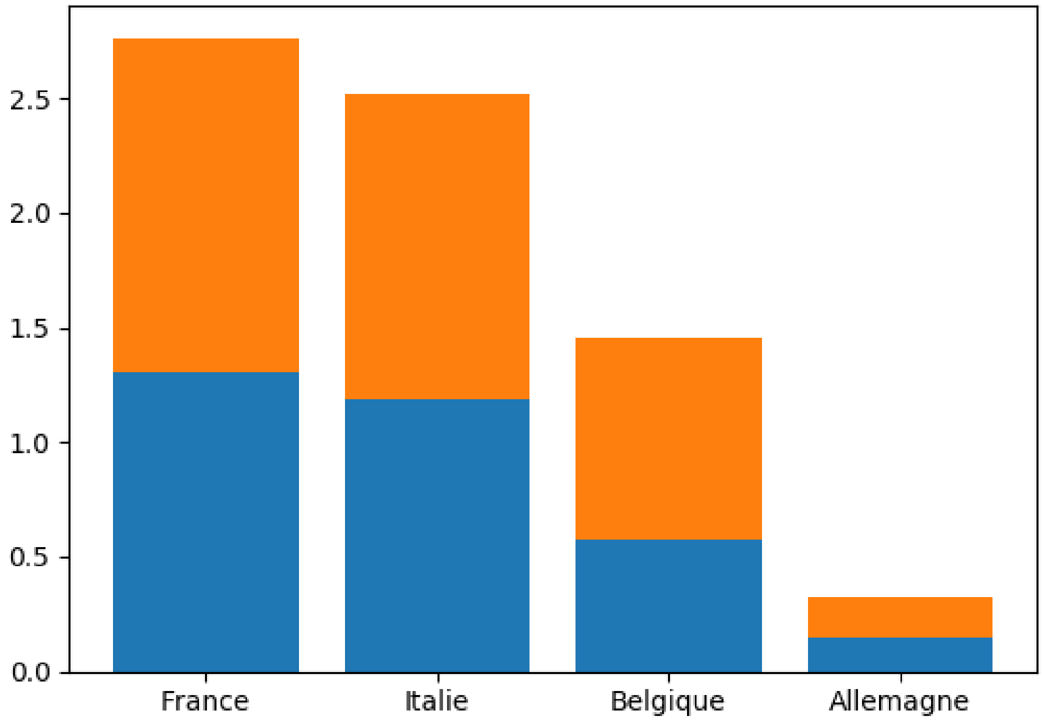
11.1.1.5.2 Stacked Bar Charts

To stack the values of the series, the starting value for the series is specified using the argument `bottom`:

```
countries = ["France", "Italie", "Belgique", "Allemagne"]
no_unemp_f = [1.307, 1.185, .577, .148]
no_unemp_h = [1.46, 1.338, .878, .179]

plt.bar(countries, no_unemp_f)
plt.bar(countries, no_unemp_h, bottom = no_unemp_f)
```

```
## <BarContainer object of 4 artists>
```

```
## <BarContainer object of 4 artists>
```



11.1.1.5.3 Aesthetic Arguments

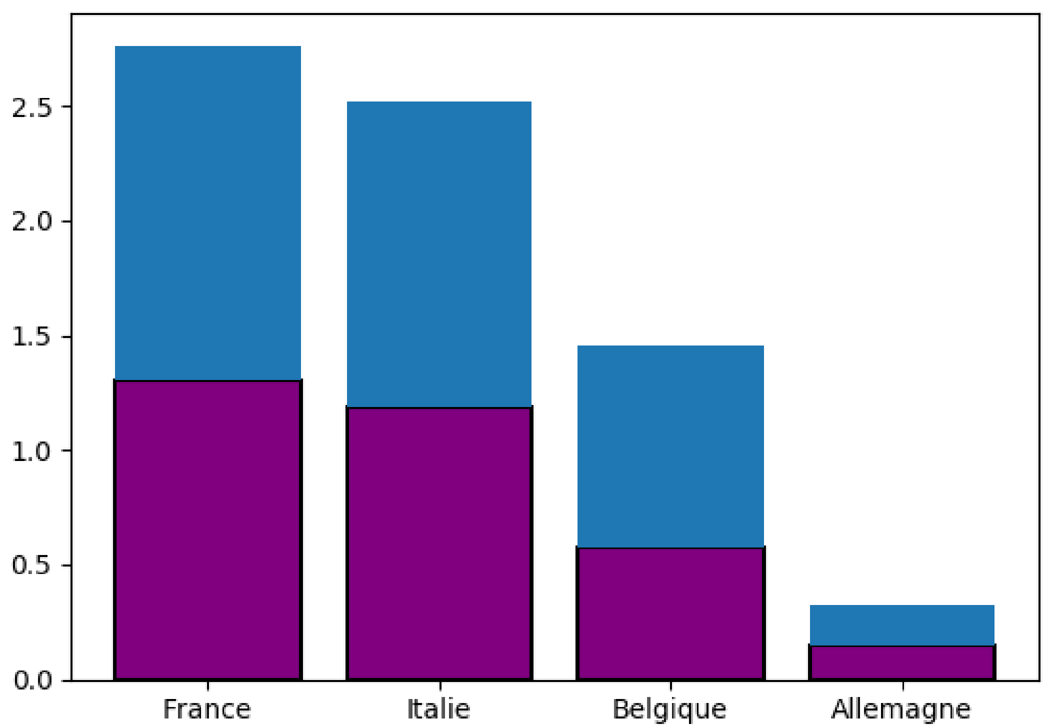
To change the filling color, we use the `color` argument; for the contour color, we enter the `edgecolor` argument; for the contour width, we rely on the `linewidth` argument:

```
countries = ["France", "Italie", "Belgique", "Allemagne"]
no_unemp_f = [1.307, 1.185, .577, .148]
no_unemp_h = [1.46, 1.338, .878, .179]

plt.bar(countries, no_unemp_f, color = "purple",
```

```
        edgecolor = "black", linewidth = 1.5)
plt.bar(countries, no_unemp_h, bottom = no_unemp_f)
```

```
## <BarContainer object of 4 artists>
```

```
## <BarContainer object of 4 artists>
```

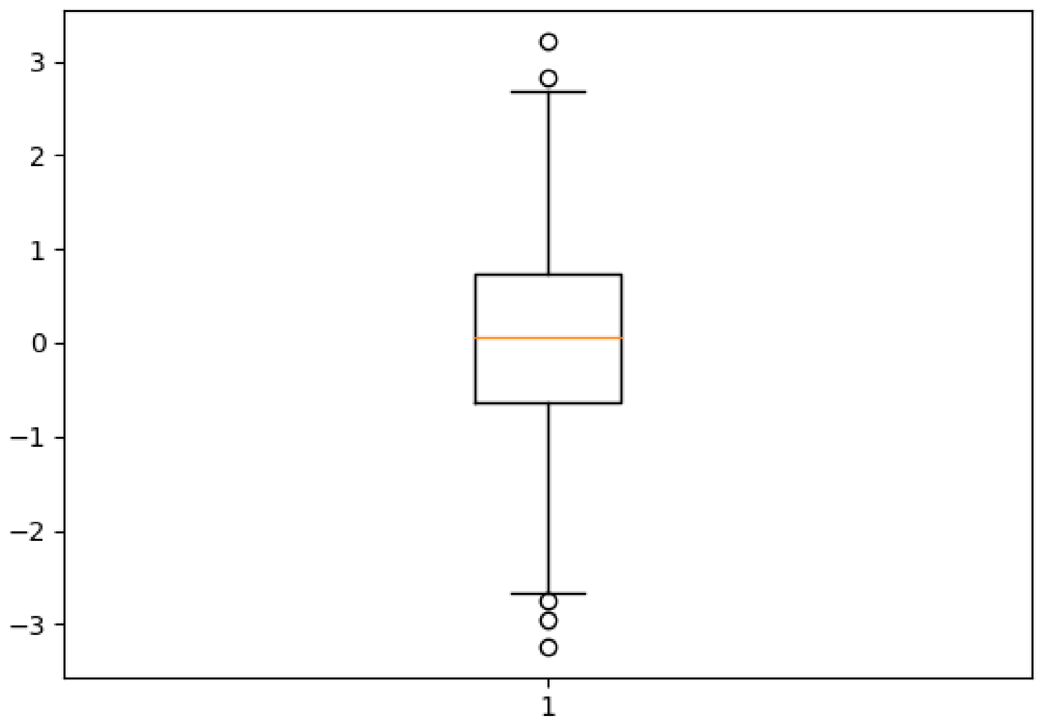


11.1.1.6 Boxplots

To make boxplot, `pyplot` offers the function `boxplot()` :

```
x = np.random.randn(1000)
plt.boxplot(x)
```

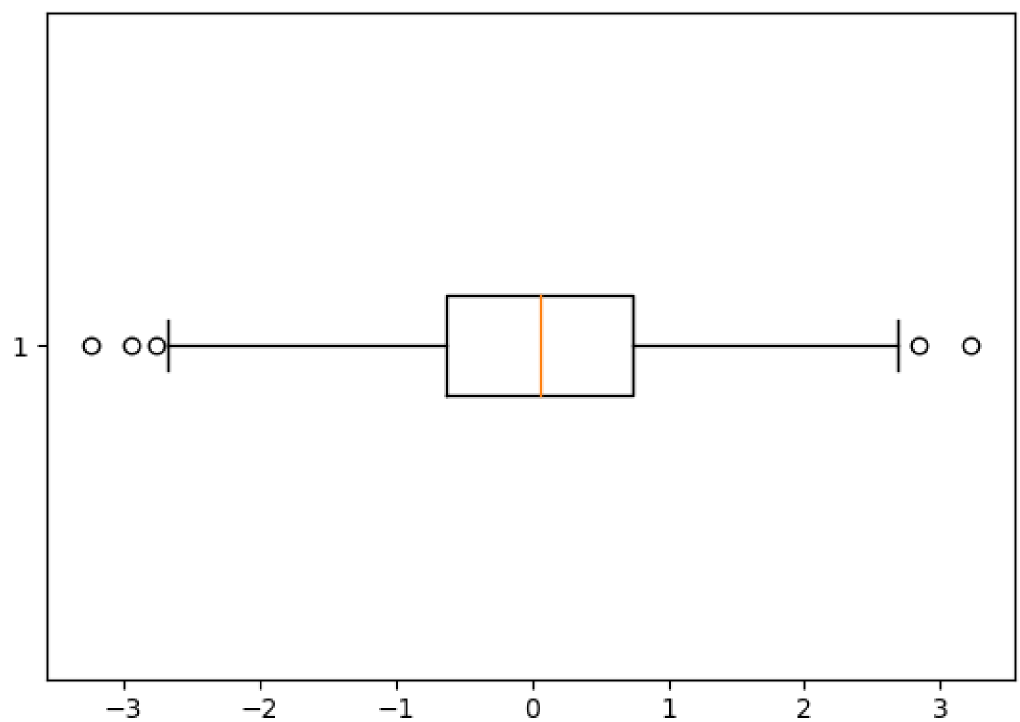
```
## {'whiskers': [<matplotlib.lines.Line2D object at 0
   x1352e8710>, <matplotlib.lines.Line2D object at 0x1352e8c50
   >], 'caps': [<matplotlib.lines.Line2D object at 0x1352f30f0
   >, <matplotlib.lines.Line2D object at 0x1352f3550>], 'boxes
   ': [<matplotlib.lines.Line2D object at 0x1352e85c0>], '
   medians': [<matplotlib.lines.Line2D object at 0x1352f39b0
   >], 'fliers': [<matplotlib.lines.Line2D object at 0
   x1352f3e10>], 'means': []}
```



By specifying `False` as the value of the argument `vert`, the boxplot is plotted horizontally:

```
plt.boxplot(x, vert = False)
```

```
## {'whiskers': [<matplotlib.lines.Line2D object at 0
   x135345550>, <matplotlib.lines.Line2D object at 0x135345a90
   >], 'caps': [<matplotlib.lines.Line2D object at 0x135345ef0
   >, <matplotlib.lines.Line2D object at 0x135352390>], 'boxes
   ': [<matplotlib.lines.Line2D object at 0x135345400>], '
   medians': [<matplotlib.lines.Line2D object at 0x1353527f0
   >], 'fliers': [<matplotlib.lines.Line2D object at 0
   x135352c50>], 'means': []}
```

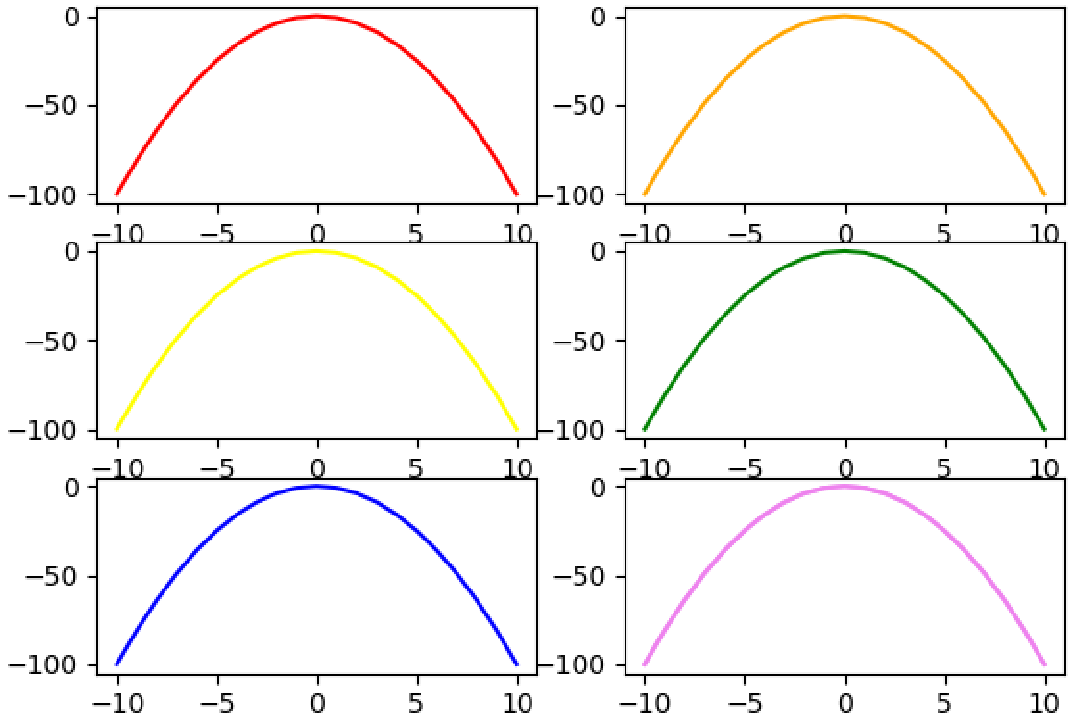


11.1.2 Several Graphs on a Figure

To place graphs next to each other, the function `subplot()` is used. The graphs will be placed as in a matrix, with a number of rows `no_rows` and a number of columns `no_col`. The dimensions of this matrix can be specified as arguments of the `subplot()` function, using the following syntax:

where `current` indicates the index of the active graph. Let's look at an example of how it works:

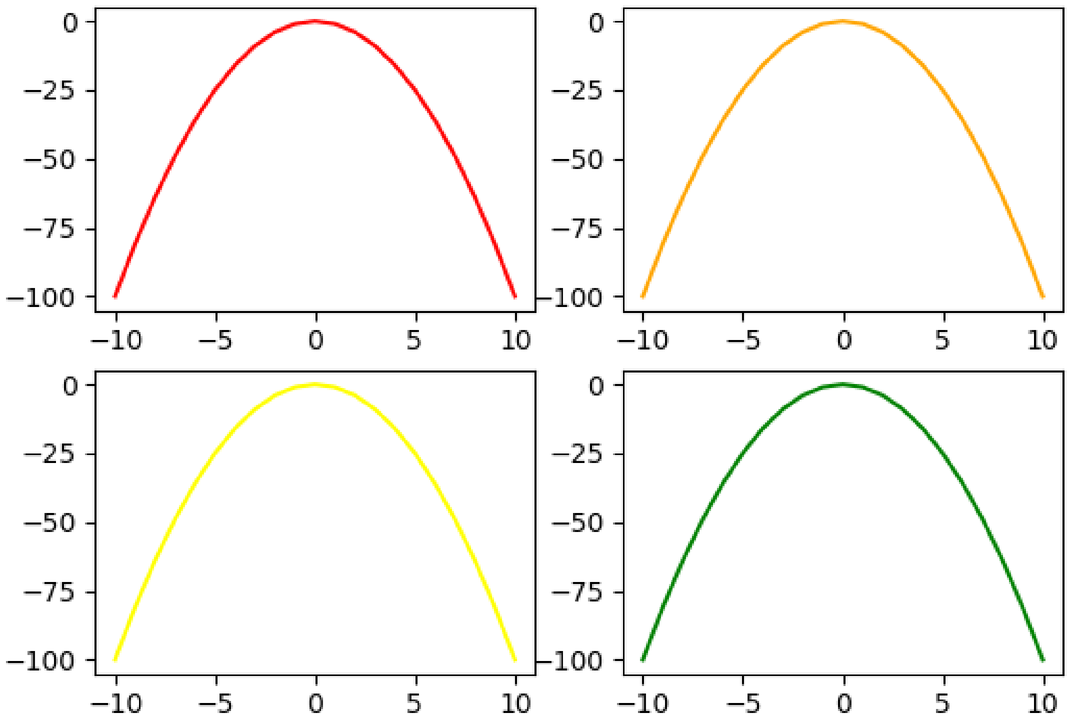
```
x = np.arange(-10, 11)
y = -x**2
# 3x2 dimension matrix of graphs

# Row 1, column 1
plt.subplot(3, 2, 1)
plt.plot(x, y, color = "red")

# Row 1, column 2
plt.subplot(3, 2, 2)
plt.plot(x, y, color = "orange")

# Row 2, column 1
plt.subplot(3, 2, 3)
plt.plot(x, y, color = "yellow")

# Row 2, column 2
plt.subplot(3, 2, 4)
plt.plot(x, y, color = "green")

# Row 3, column 1
plt.subplot(3, 2, 5)
plt.plot(x, y, color = "blue")

# Row 3, column 2
plt.subplot(3, 2, 6)
plt.plot(x, y, color = "violet")
```



Remember that the matrices are filled line by line in Python, which allows a good understanding of the value of the active graph number.

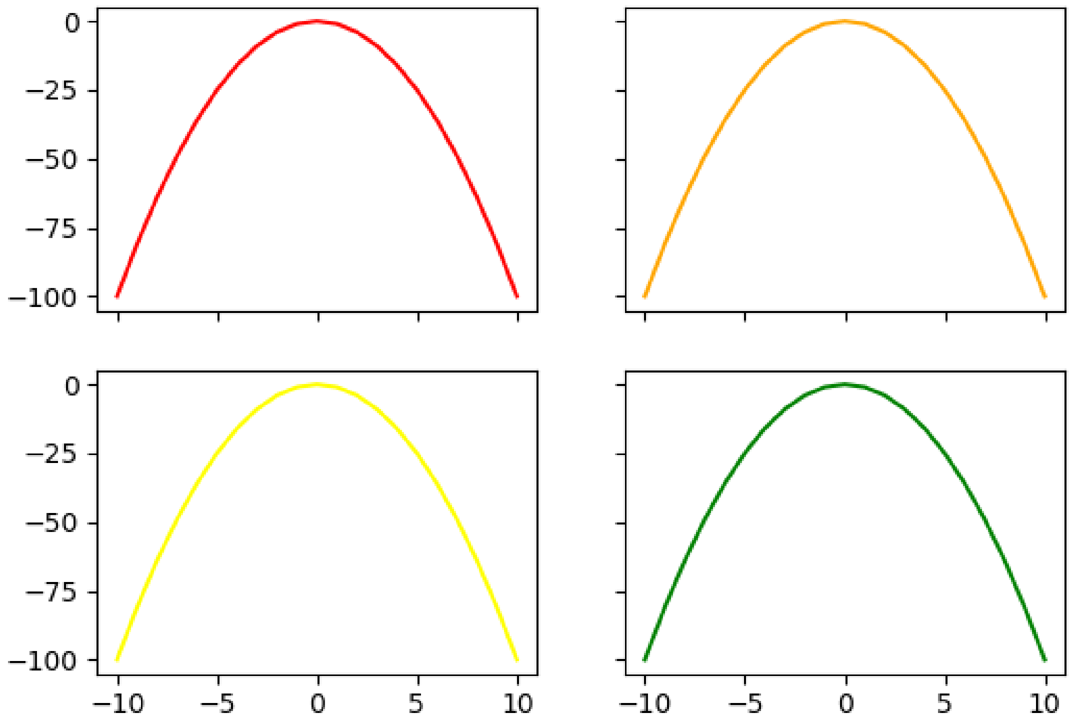
By using the function `subplots()` (beware of the final “s” of the name of the function that differentiates it from the previous one), it is possible to produce a matrix of graphs as well, by proceeding as follows:

```
f, ax_arr = plt.subplots(2, 2)
ax_arr[0, 0].plot(x, y, color = "red")
ax_arr[0, 1].plot(x, y, color = "orange")
ax_arr[1, 0].plot(x, y, color = "yellow")
ax_arr[1, 1].plot(x, y, color = "green")
```



This approach has the advantage of easily specifying the sharing of axes between the different subgraphs, via the `sharex` and `sharey` arguments:

```
f, ax_arr = plt.subplots(2, 2, sharey=True, sharex = True)
ax_arr[0, 0].plot(x, y, color = "red")
ax_arr[0, 1].plot(x, y, color = "orange")
ax_arr[1, 0].plot(x, y, color = "yellow")
ax_arr[1, 1].plot(x, y, color = "green")
```



11.1.3 Graphics Elements

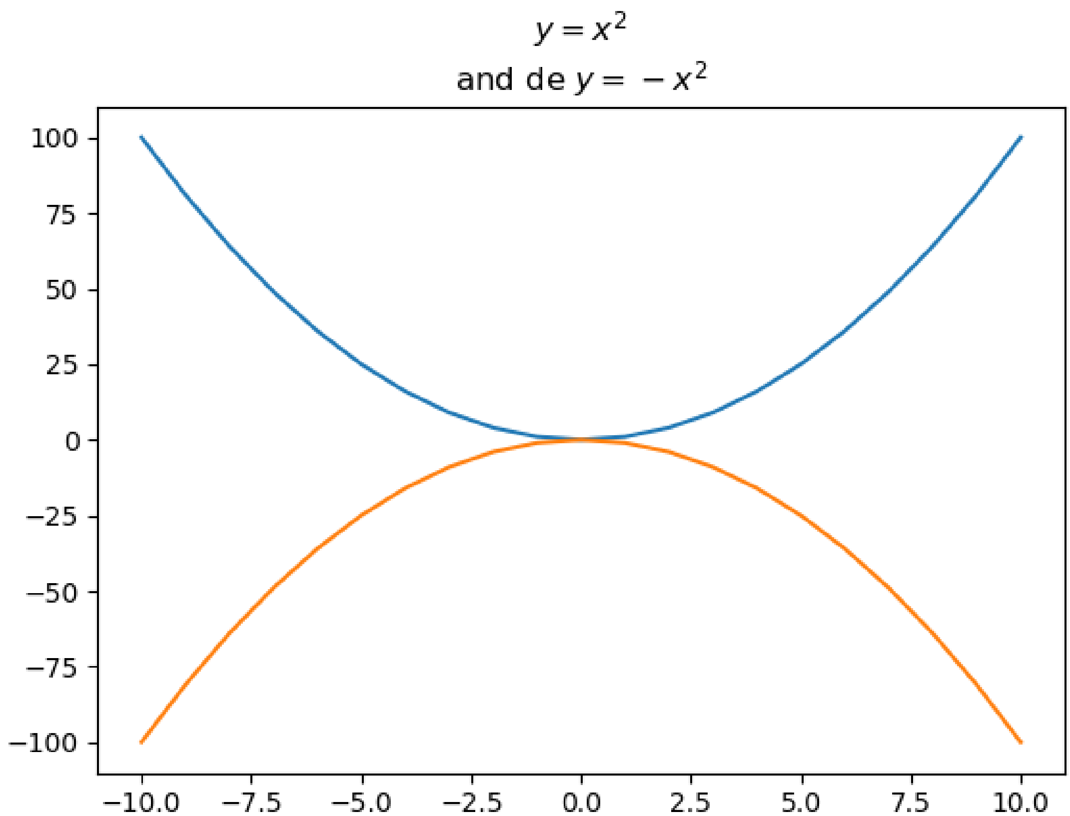
So far, we have looked at how to create different geometries, but we have not touched the axes, their values or labels (except in the unexplained example of side-by-side diagrams), or modified the legends or titles.

11.1.3.1 Title

To add a title to the graph, we can use the title() function:

```
x = np.arange(-10, 11)
y = x**2
y_2 = -y
```

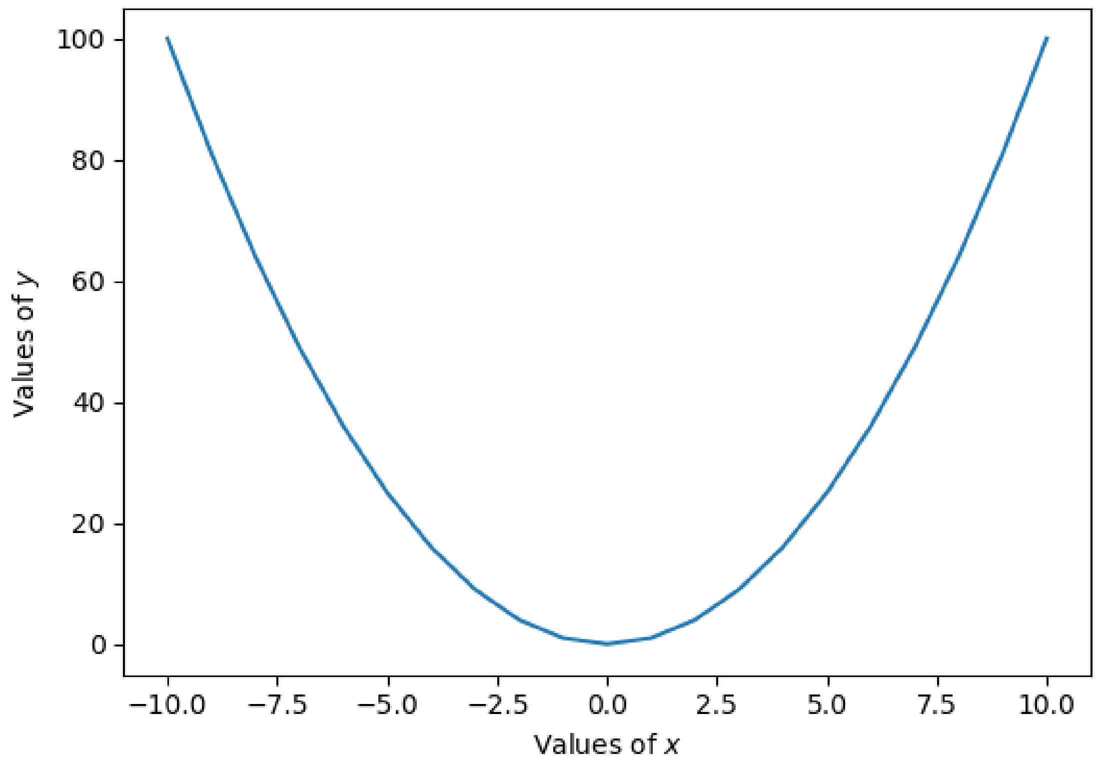
```
plt.plot(x, y)
plt.plot(x, y_2)
plt.title("$y = x^2$ \nand $y = -x^2$")
```



11.1.3.2 Axes

The `xlabel()` and `ylabel()` functions allow us to add labels to the axes:

```
x = np.arange(-10, 11)
y = x**2
plt.plot(x, y)
plt.xlabel("Values of $x$")
plt.ylabel("Values of $y$")
```

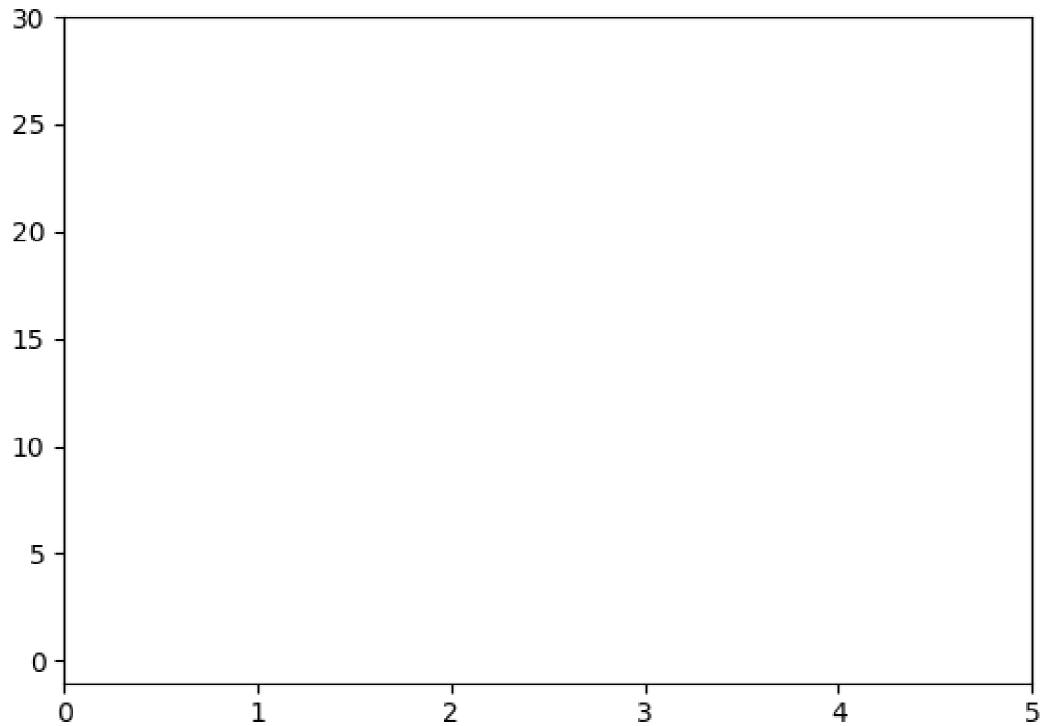


11.1.3.2.1 Limits

To control the axis limits, the `axis()` function is used, specifying the arguments `xmin`, `xmax`, `ymax`, `ymin` and `ymax`, denoting, respectively, the lower and upper bounds of the x-axis and the lower and upper bounds of the y-axis:

```
plt.axis(xmin = 0, xmax = 5, ymin = -1, ymax = 30)
```

```
## (0, 5, -1, 30)
```



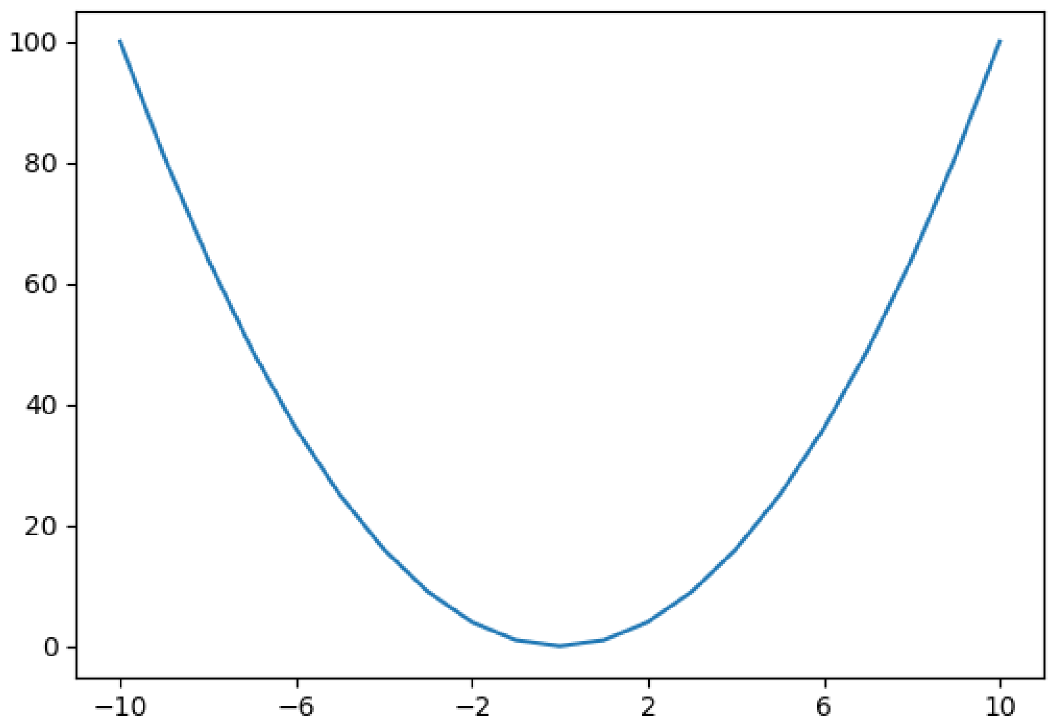
11.1.3.2.2 Marks and labels

The `xticks()` and `yticks()` functions are used to obtain or modify the marks of the x-axis and the y-axis, respectively.

```
x = np.arange(-10, 11)
y = x**2
plt.plot(x, y)
plt.xticks(np.arange(-10, 11, step = 4))
```

```
## ([<matplotlib.axis.XTick object at 0x135e61ac8>, <
   matplotlib.axis.XTick object at 0x135e61400>, <matplotlib.
   axis.XTick object at 0x135e612e8>, <matplotlib.axis.XTick
   object at 0x135a2c278>, <matplotlib.axis.XTick object at 0
   x135a2c780>, <matplotlib.axis.XTick object at 0x135a2cc88
```

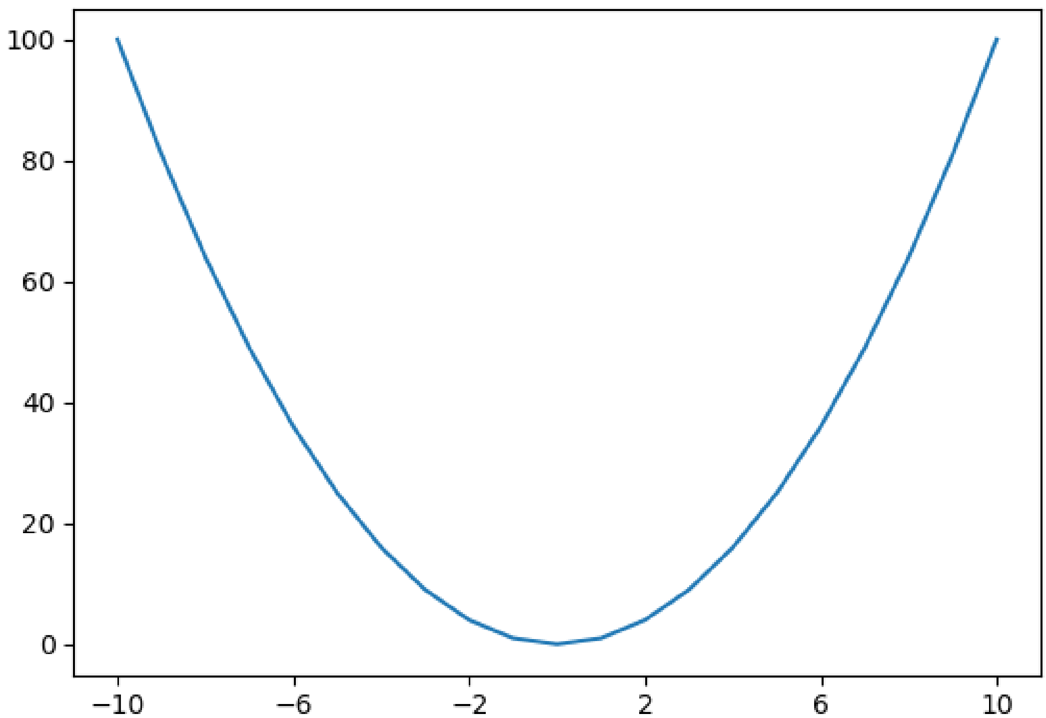
```
>], <a list of 6 Text xticklabel objects>)
```



It can be convenient to retrieve the positions and labels of a graph so that we can modify them, for example to define the spacing between each mark:

```
plt.plot(x, y)
locs_x, labels_x = plt.xticks()
locs_y, labels_y = plt.yticks()
loc_x_new = np.arange(locs_x[0], locs_x[-1], step = 5)
loc_y_new = np.arange(locs_y[0], locs_y[-1], step = 10)

plt.xticks(loc_x_new)
plt.yticks(loc_y_new)
```

```
## ([<matplotlib.axis.XTick object at 0x135eb1f60>, <
   matplotlib.axis.XTick object at 0x135eb1898>, <matplotlib.
   axis.XTick object at 0x135eb15f8>, <matplotlib.axis.XTick
   object at 0x135ed8710>, <matplotlib.axis.XTick object at 0
   x135ed8c18>, <matplotlib.axis.XTick object at 0x135edf198
   >], <a list of 6 Text xticklabel objects>)
```



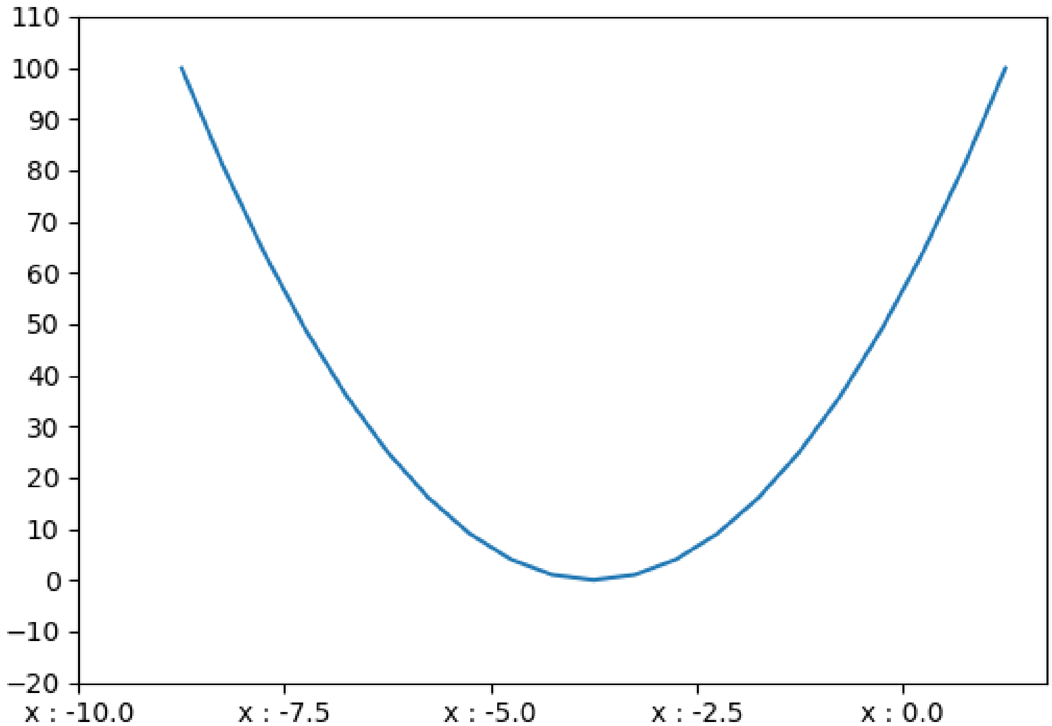
The labels on the marks can also be modified:

```
plt.plot(x, y)
locs_x, labels_x = plt.xticks()
locs_y, labels_y = plt.yticks()
loc_x_new = np.arange(locs_x[0], locs_x[-1], step = 5)
loc_y_new = np.arange(locs_y[0], locs_y[-1], step = 10)
```

```
labels_x_new = []
for i in np.arange(1, len(locs_x)):
        labels_x_new.append("x : " + str(locs_x[i]))

plt.xticks(loc_x_new, labels_x_new)
plt.yticks(loc_y_new)
```

```
## ([<matplotlib.axis.XTick object at 0x1360294a8>, <
   matplotlib.axis.XTick object at 0x136022da0>, <matplotlib.
   axis.XTick object at 0x136048898>, <matplotlib.axis.XTick
   object at 0x136048ba8>, <matplotlib.axis.XTick object at 0
   x13628c400>], <a list of 5 Text xticklabel objects>)
```

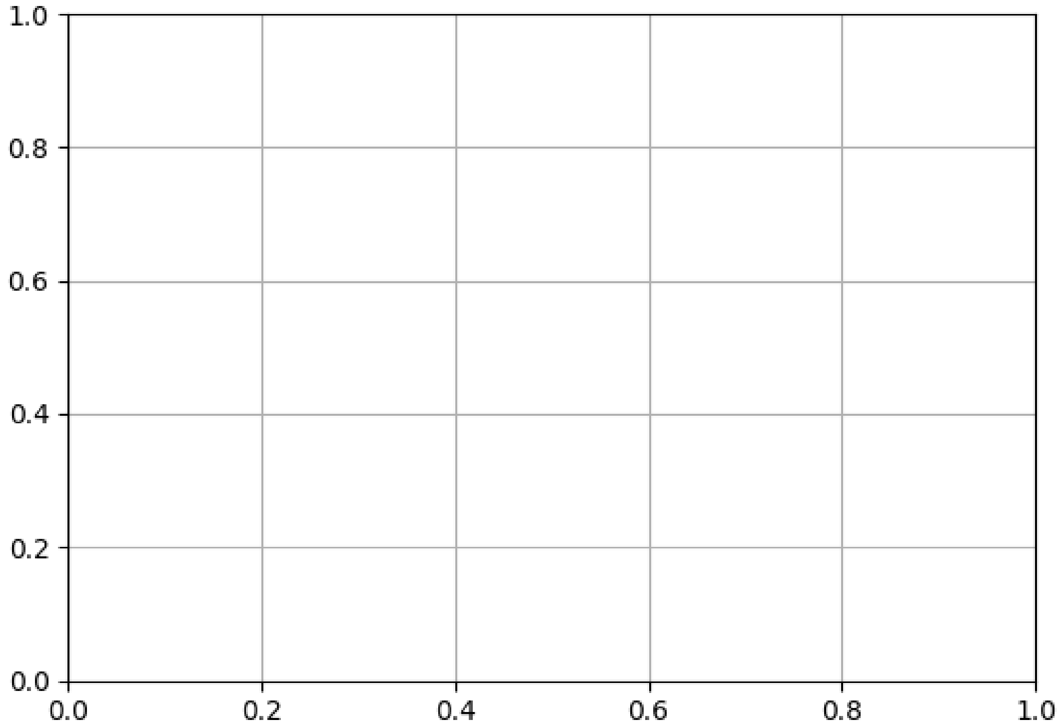
```
## ([<matplotlib.axis.YTick object at 0x13602f2e8>, <
   matplotlib.axis.YTick object at 0x136029b70>, <matplotlib.
   axis.YTick object at 0x136294be0>, <matplotlib.axis.YTick
   object at 0x13629c2e8>, <matplotlib.axis.YTick object at 0
   x13629ca90>, <matplotlib.axis.YTick object at 0x13629ce80>,
    <matplotlib.axis.YTick object at 0x1362a44e0>, <matplotlib
   .axis.YTick object at 0x1362a49e8>, <matplotlib.axis.YTick
   object at 0x136022b38>, <matplotlib.axis.YTick object at 0
   x1362a4ac8>, <matplotlib.axis.YTick object at 0x13629c128>,
    <matplotlib.axis.YTick object at 0x136048b70>, <matplotlib
   .axis.YTick object at 0x1362ad908>, <matplotlib.axis.YTick
   object at 0x1362ade10>], <a list of 14 Text yticklabel
   objects>)
```



11.1.3.2.3 Grid

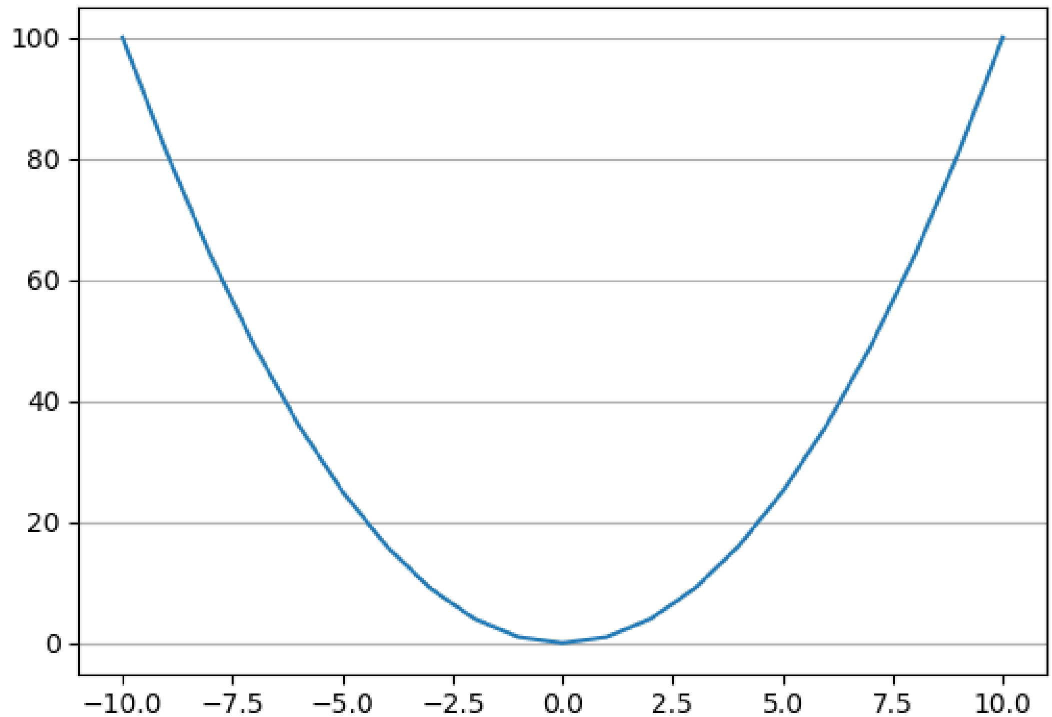
To add a grid, `grid()` function is used:

```
x = np.arange(-10, 11)
y = x**2
plt.plot(x, y)
plt.grid()
```



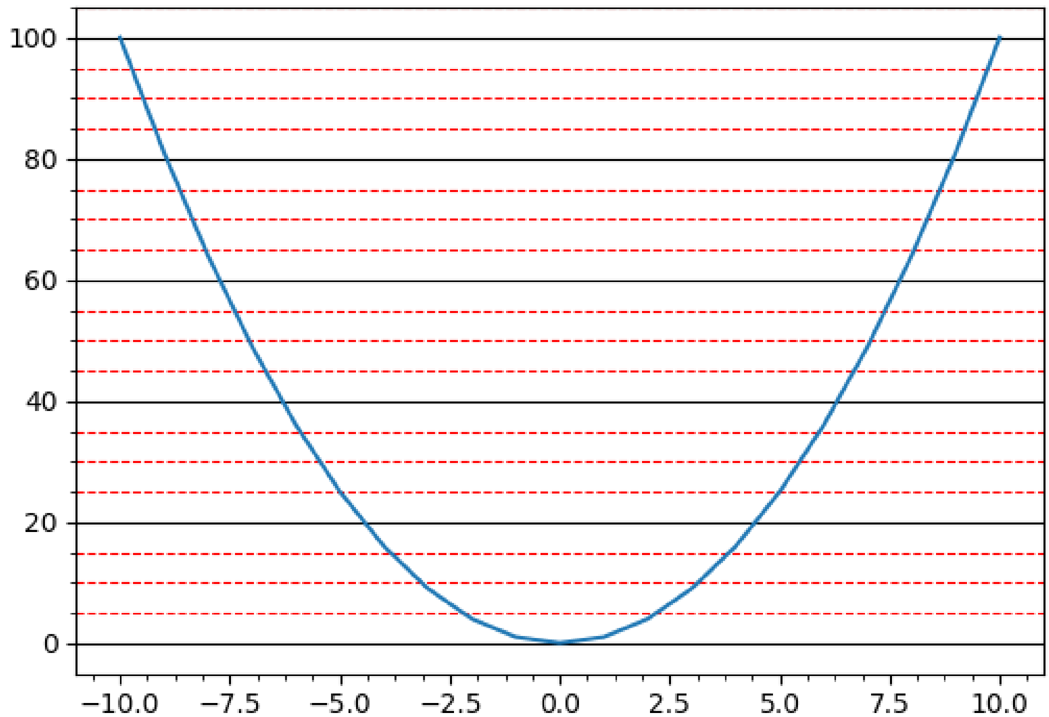
The `axis` argument allows to define if we want a grid for both axes (`both`, by default), only for the x-axis (`x`), or only for the y-axis (`y`):

```
plt.plot(x, y)
plt.grid(axis = "y")
```



It is possible to set the major or minor lines of the grid:

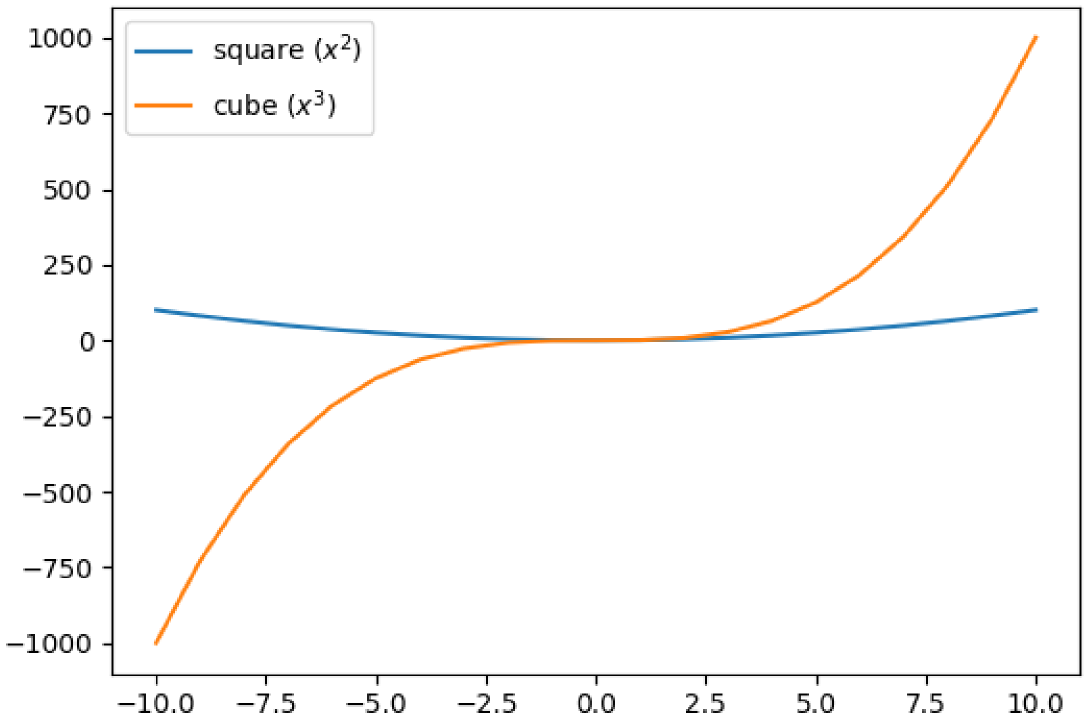
```
plt.plot(x, y)
plt.minorticks_on()
plt.grid(which = "major", axis = "y", color = "black")
plt.grid(which = "minor", axis = "y", color = "red", linestyle = "--")
```



11.1.3.3 Legends

When we wish to add a legend, we specify its label to the `label` argument in the call of the `plot` function, then we use the `legend()` function:

```
x = np.arange(-10, 11)
y = x**2
y_2 = x**3
plt.plot(x, y, label = "square ($x^2$)")
plt.plot(x, y_2, label = "cube ($x^3$)")
plt.legend()
plt.legend()
```



To specify the position of the legend, we can use the `loc` argument in the `legend()` function, indicating a value as reported in Table 11.1.3.3.

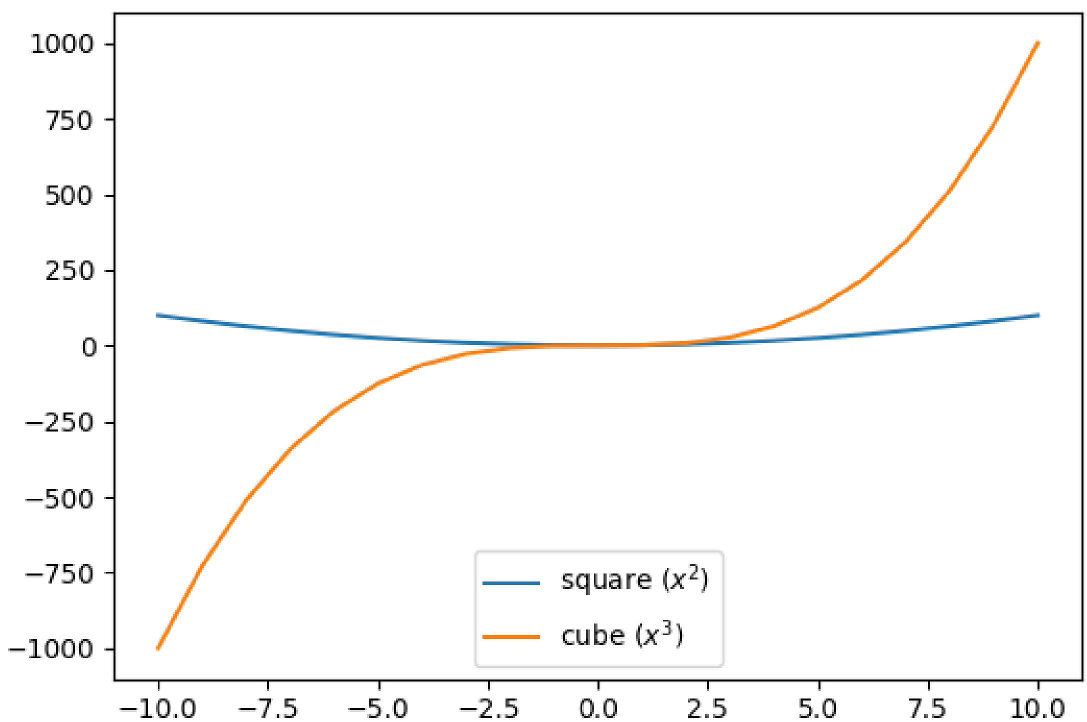
String	Code	Description
`best`	0	Let Python optimize the positroning
`upper right`	1	Upper right corner
`upper left`	2	Upper left corner
`lower left`	3	Lower left corner
`lower right`	4	Lower right corner
`right`	5	Right
`center left`	6	Centered in the middle on the left
`center right`	7	Centered in the middle on the right
`lower center`	8	Centered at the bottom
`upper center`	9	Centered at the top

`center` | 10 | Centered |

Table: Position of the legend

For example, to center the legend in the middle, at the bottom of the graph:

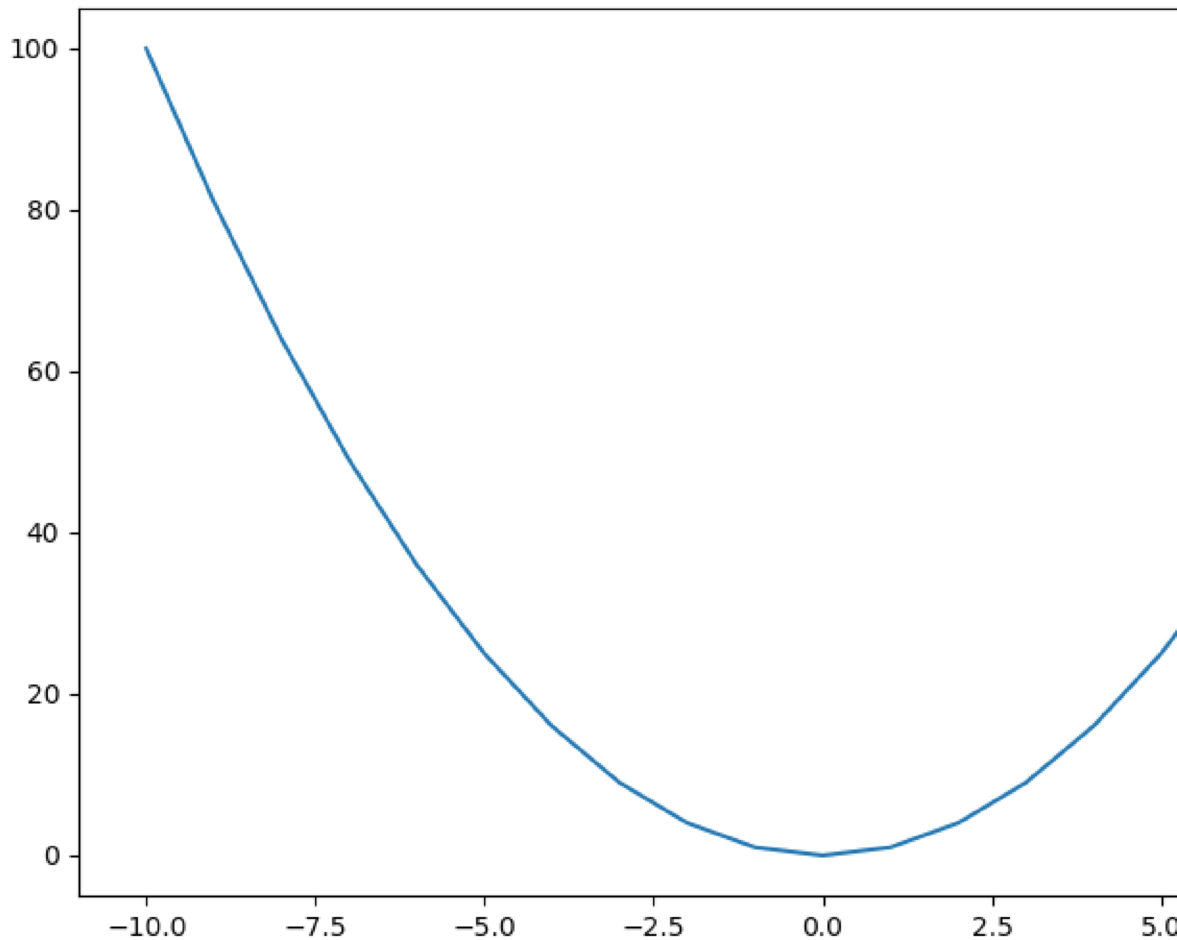
```
x = np.arange(-10, 11)
y = x**2
y_2 = x**3
plt.plot(x, y, label = "square ($x^2$)")
plt.plot(x, y_2, label = "cube ($x^3$)")
plt.legend(loc = "lower center")
```



11.1.4 Dimensions

To define the dimensions of a figure, we specify the `figsize` argument of the `figure()` function. It is provided with a tuple of integers whose first element corresponds to the length and the second to the height (the values are in inches):

```
x = np.arange(-10, 11)
y = x**2
plt.figure(figsize=(10,6))
plt.plot(x, y)
```



11.1.5 Exporting Graphs

To save a graph, the function `plt.savefig()` can be used. We specify the path to the file to be created, indicating the extension of the desired file (*e.g.*, `png` or `pdf`):

```
x = np.arange(-10, 11)
y = x**2
y_2 = x**3
plt.figure(figsize=(10,6))
plt.plot(x, y, label = "square ($x^2$)")
plt.plot(x, y_2, label = "cube ($x^3$)")
plt.legend(loc = "lower center")
plt.savefig("test.pdf")
```

The specified extension (in this example, `pdf`) determines the output file format. The extensions indicated in the keys of the dictionary returned by the following instruction (the values giving a description of the file type) can be used:

```
print(fig.canvas.get_supported_filetypes())
```

```
## {'ps': 'Postscript', 'eps': 'Encapsulated Postscript', 'pdf
   ': 'Portable Document Format', 'pgf': 'PGF code for LaTeX',
    'png': 'Portable Network Graphics', 'raw': 'Raw RGBA
   bitmap', 'rgba': 'Raw RGBA bitmap', 'svg': 'Scalable Vector
    Graphics', 'svgz': 'Scalable Vector Graphics', 'jpg': '
   Joint Photographic Experts Group', 'jpeg': 'Joint
   Photographic Experts Group', 'tif': 'Tagged Image File
   Format', 'tiff': 'Tagged Image File Format'}
```

11.2 Graphics with Seaborn

To be done.

https://seaborn.pydata.org/

Chapter 12

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.".