Badji Mokhtar University
Faculty of Earth Sciences
Department of planning

UEM1.1.2 Python Programming Workshop

S1 2026/2027

Section: M1 Smart Urbanism
Time: Sunday 14h30 - 17h00
Instructor: Dr. Zehir Hatem
Email: hatem.zehir@univ-annaba.dz
Office Hours: By appointment

Course Description

This course introduces students to Python programming for urban data analysis, with a focus on practical applications in smart urban planning. Designed for students with a basic understanding of programming concepts, it develops fundamental programming skills, including variables, data types, conditions, loops, functions, and data input/output. Students learn to work with urban datasets using Pandas and to process and analyze geospatial data using GeoPandas. The course also covers data cleaning, spatial joins, basic statistical analysis, and the calculation of urban indicators such as distances, areas, and mobility patterns. Finally, students learn to create thematic, interactive, and web-based maps using Matplotlib, Seaborn, Folium, and Plotly, enabling them to transform urban data into meaningful visual and analytical information.

Key Learning Outcomes

- ❏ Apply Python fundamentals to urban data analysis
- ❏ Clean and organize urban datasets using Pandas
- ❏ Analyze and process geospatial data with GeoPandas
- ❏ Create thematic and interactive urban maps
- ❏ Interpret urban indicators to support planning analysis

Course-Level Learning Outcome

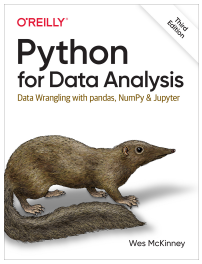
The following is a list of course-level learning outcomes.

- ✷ Develop basic Python programs for urban data analysis
- ✷ Clean, transform, and analyze urban datasets
- ✷ Process and analyze geospatial urban data
- ✷ Create informative thematic and interactive maps
- ✷ Interpret data to support urban planning decisions

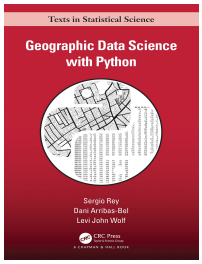
Textbooks and Course Materials

Online Resources: Lecture notes, documentation and python codes will be provided to students throughout the semester.

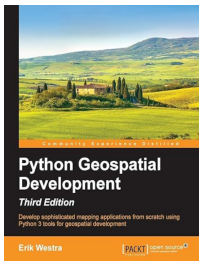
textbook: There is no required textbook for this course. However, the following is a list of textbooks that are strongly recommended for this course:



Title: Python for Data Analysis, 3nd Edition
Author: McKinney, W
Publisher: O'Reilly Media, publication year: 2022
Available at: wesmckinney.com/book/



Title: Geographic Data Science with Python
Author: Dani Arribas-Bel, Levi John Wolf, and Sergio Joseph Rey
Publisher: Chapman & Hall/CRC; publication year: 2023
ISBN-13: 978-0367263119



Title: Python Geospatial Development, 3rd Edition
Author: Westra, E
Publisher: Packt Publishing; publication year: 2016
ISBN-13: 978-1785288937

Course Policies

Class Policy and Rules of Conduct

Cell phones strictly prohibited during classes and labs. Electronic devices must be put on silent mode (🔕) and placed in your bags or pockets (not on the desk just in front of you).

Headphones should be removed all the time.

During lectures, students must turn off their monitors and take notes. Using the computer during lectures without authorization is strictly not allowed.

Video or audio recordings and taking photographs are NOT permitted without permission.

Persistent talking, whispering or any disruptive attitude will not be tolerated.

Contents

1 Fundamentals and Environment **11**

1.1 Warm-up Exercise: The Population Density of Annaba 11
1.2 Conceptual Foundations of Programming in Urban Planning 11
1.3 What is Python? 12
1.4 Tips for Learning Python 13
1.5 Python Programming Environment 13
1.6 Interactive Mode and Python as a Calculator 15
1.7 Initial Code Execution 16
1.8 Error Diagnostics and Debugging 17
1.9 Applied Introductory Practice 17
1.10 Comments 18
1.11 Variables and Data Types 18
1.12 Output and Input 22
1.13 Lists 24
1.14 Control Flow: Loops 26
1.15 Conditional Statements: if, elif, else 29
1.16 Functions 32
1.17 Data File Formats: CSV, JSON and Excel 36
1.18 Exercises (Set 2) 39
1.19 Solutions to the Exercises 41

2 Data Analysis with Pandas and GeoPandas **45**

2.1 Warm-up: the Density of Several Communes 45
2.2 What is a Python Library? 46
2.3 Pandas: Definition 46
2.4 Where is Pandas Used? 48
2.5 Why Pandas? 48
2.6 Tips for Learning Pandas 49
2.7 First Contact with Pandas 49
2.8 First Contact with GeoPandas 53

2.9 Shapefile and GeoJSON Formats 58
2.10 Reading Shapefile and GeoJSON Files 60
2.11 Real Data from OpenStreetMap 62
2.12 Spatial Joins, Areas and Distances 65
2.13 Exercises (Set 3) 71
2.14 Solutions to the Exercises (Set 3) 72
2.15 Quick Reference 75

List of Figures

1.1 This is the main Anaconda Navigator interface, which provides access to various data science tools and environments. 14

1.2 This is the main JupyterLab interface, which provides an interactive computing environment for data science tasks. 14

1.3 This is how to open the Anaconda Prompt from the Anaconda Navigator interface. The Anaconda Prompt is a command-line interface that allows users to manage conda environments and execute Python commands. . 15

List of Tables

1.1 A simplified urban data processing pipeline illustrating the flow from raw input data to derived output metrics. . . . 12
1.2 Main milestones in the history of Python. . . . 13
1.3 Arithmetic operators in Python. . . . 16
1.4 Basic data types in Python. . . . 19
1.5 Examples of valid and invalid variable names. . . . 20
1.6 Shortcut assignment operators. . . . 20
1.7 Indexes of the elements of the list `cities`. . . . 24
1.8 Common operations on lists. . . . 25
1.9 Comparison operators. . . . 28
1.10 Logical operators. . . . 31
1.11 Comparison of the CSV, JSON and Excel formats. . . . 40

2.1 The libraries used in this chapter. . . . 47
2.2 Main milestones in the history of Pandas. . . . 47
2.3 The two main objects of Pandas. . . . 48
2.4 Excel compared with Pandas. . . . 48
2.5 The two coordinate systems used in this chapter. . . . 57
2.6 The two ways of drawing a GeoDataFrame. . . . 58
2.7 The files that make a shapefile. . . . 58
2.8 Comparison of the shapefile and GeoJSON formats. . . . 60
2.9 Quick reference: Pandas. . . . 76
2.10 Quick reference: GeoPandas and OSMnx. . . . 77

List of Codes

1.1 Python Output Commands 16
1.2 Python Variable Assignment and Output 17
1.3 Python Error Example 17
1.4 Displaying City Name and Population 18
1.5 Population Density Calculation 18
1.6 Using Comments 18
1.7 Using Variables to Compute a Density 18
1.8 Updating a Variable 19
1.9 Checking the Type of a Variable 19
1.10 Scientific Notation 20
1.11 Shortcut Operators 20
1.12 Operators on Strings 21
1.13 Type Conversion 21
1.14 Printing Several Items 22
1.15 The sep and end Parameters 22
1.16 Special Characters 22
1.17 Formatted Strings 23
1.18 Reading Text from the User 23
1.19 Reading Numbers from the User 23
1.20 Creating a List of Cities 24
1.21 Accessing List Elements 24
1.22 Modifying and Slicing a List 24
1.23 Operations on Lists 25
1.24 Useful Functions on Lists 26
1.25 The range() and list() Functions 26
1.26 Repetitive Code Without a Loop 26
1.27 Structure of a Block 27
1.28 A for Loop over a List 27
1.29 A Loop with Several Instructions 27
1.30 A for Loop with range() 27

1.31 Comparison Inside a Loop . . . 28
1.32 A while Loop: Population Growth . . . 29
1.33 A Simple if Statement . . . 29
1.34 if with else . . . 29
1.35 Classifying a Density with if, elif and else . . . 30
1.36 A Wrong Order of Tests (to Avoid) . . . 30
1.37 Combining Conditions with and and not . . . 30
1.38 The in Operator and Chained Comparisons . . . 31
1.39 An Interactive Program with a Condition . . . 31
1.40 A Counter Inside a Loop . . . 32
1.41 The Same Calculation Repeated . . . 32
1.42 Structure of a Function . . . 33
1.43 Our First Function: density() . . . 33
1.44 Using the Returned Value . . . 33
1.45 Default Values and Named Arguments . . . 34
1.46 A Function Without return Gives None . . . 34
1.47 Combining Functions, Conditions and a Loop . . . 34
1.48 A Function Returning Two Values . . . 35
1.49 A Local Variable is Not Visible Outside its Function . . . 35
1.50 A Function with a Docstring . . . 36
1.51 The Data to Save: a List of Rows . . . 36
1.52 Writing a CSV File . . . 36
1.53 Reading a CSV File . . . 37
1.54 Writing a JSON File . . . 37
1.55 Reading a JSON File . . . 38
1.56 Writing and Reading an Excel File . . . 38
1.57 Size of the Three Files . . . 39
1.58 Solutions of Exercises 1 to 4 . . . 41
1.59 Solution of Exercise 5 . . . 42
1.60 Solutions of Exercises 6 to 8 . . . 42
1.61 Solutions of Exercises 9 and 10 . . . 42
1.62 Solutions of Exercises 11 and 12 . . . 43
1.63 Solution of Exercise 13 . . . 44
2.1 Density of Several Communes with Lists and a Loop . . . 45
2.2 Density of Several Communes with Pandas . . . 46
2.3 Three Ways to Import a Library . . . 46
2.4 Checking the Version of Pandas . . . 47
2.5 Creating a DataFrame . . . 49

2.6 Exploring a DataFrame . . . 50
2.7 Selecting a Column . . . 50
2.8 Creating a New Column . . . 51
2.9 Sum, Mean and Maximum of a Column . . . 51
2.10 Filtering Rows . . . 52
2.11 Sorting a Table . . . 52
2.12 Reading and Saving a CSV File . . . 53
2.13 Creating a GeoDataFrame of Points . . . 54
2.14 Drawing Points on a Real Map . . . 54
2.15 Changing the Style of the Points . . . 55
2.16 Other Background Maps . . . 55
2.17 Saving a Map as a Web Page . . . 56
2.18 Displaying the CRS of a GeoDataFrame . . . 56
2.19 A Simple Map without Background . . . 57
2.20 Files Created by a Shapefile . . . 58
2.21 Content of a GeoJSON File . . . 59
2.22 Reading a Shapefile and a GeoJSON File . . . 60
2.23 Exploring a GeoDataFrame Read from a File . . . 60
2.24 Saving a GeoDataFrame . . . 61
2.25 Filtering a GeoDataFrame . . . 61
2.26 Downloading Hospitals from OpenStreetMap . . . 62
2.27 Drawing the Hospitals on a Real Map . . . 63
2.28 Downloading Schools, Parks and Pharmacies . . . 63
2.29 Several Layers on the Same Map . . . 63
2.30 Drawing the Road Network . . . 64
2.31 Converting OSM Layers to Metres and Centroids . . . 64
2.32 Distance from a School to the Nearest Hospital . . . 64
2.33 Total Area of the Parks . . . 64
2.34 Saving OSM Data as a GeoJSON File . . . 65
2.35 Creating Two Polygons . . . 65
2.36 Drawing Zones and Points on a Static Map . . . 66
2.37 Drawing Zones and Points on a Real Map . . . 66
2.38 A Spatial Join . . . 67
2.39 Counting Places per Zone . . . 68
2.40 Area of the Zones in Square Kilometres . . . 69
2.41 Distance from Each Place to the Train Station . . . 69
2.42 Population Density from the Area of the Polygons . . . 70
2.43 Data of Exercise 14 . . . 71

2.44 Data of Exercise 15 71
2.45 Data of Exercise 16 71
2.46 Solution of Exercise 14 72
2.47 Solution of Exercise 15 73
2.48 Solution of Exercise 16 73
2.49 Solution of Exercise 17 75

Chapter 1

Fundamentals and Environment

Warm-up Exercise: The Population Density of Annaba

Before introducing any programming concept, let us solve an everyday problem of an urban planner.

- **Exercise.** The commune of Annaba has about 257 000 inhabitants and covers about 49 km^2 (approximate values, used for teaching purposes). What is its population density, in inhabitants per km^2?

Solved by hand (or with a calculator), the density is the ratio between the population and the area:

$$\text{density} = \frac{\text{population}}{\text{area}} = \frac{257{,}000}{49} \approx 5{,}245 \text{ inhabitants/km}^2$$

Now imagine repeating this calculation for every wilaya, every neighbourhood, and every census year. A calculator quickly becomes painful. Programming solves this: the instructions are written **once**, and the computer repeats them quickly and without mistakes. By the end of this course, you will be able to write a small program that computes urban indicators (density, area, growth, ...) for a whole list of cities.

Conceptual Foundations of Programming in Urban Planning

Urban planning relies on analyzing demography, spatial bounds, infrastructure capacities, and environmental metrics. Historically, these analyses were performed manually or through disconnected spreadsheet calculations and static geographic information systems (GIS). Modern computational urban planning replaces manual workflows with programmatic scripts, allowing analytical procedures to be fully automated, audited, and scaled across complex metropolitan datasets.

To establish programmatic workflows, urban planners must master core computational concepts. Programming is the act of composing an unambiguous sequence of instructions that a computer executes to process data and solve analytical problems. In an urban computing pipeline, raw population input data undergoes mathematical processing to generate a derived population density metric, while runtime diagnostic tools provide instant feedback if processing errors occur.

The fundamental components of an algorithmic data processing pipeline include:

- **Program**: A structured collection of explicit instructions executed sequentially to perform a specific computational task.

- **Instruction**: A single syntactical command evaluated by the Python interpreter to modify system state or compute a value.
- **Variable**: A named reference bound to a specific memory location holding dynamic values, such as population counts or land area figures.
- **Input**: External raw data ingested by a program, ranging from single numerical entries to large-scale tabular files.
- **Processing**: Mathematical operations, structural transformations, or logical evaluations applied to input data.
- **Output**: The final analytical result displayed on screen, rendered as a spatial visualization, or exported to tabular storage.
- **Error**: An exception generated when an instruction violates language syntax rules or attempts an invalid mathematical operation.

Pipeline Phase	**Urban Analytical Context**	**Computational Representation**
Input	Raw census population count	population = 257000
Input	Urban administrative area (km^2)	surface = 49
Processing	Ratio calculation of population density	densite = population / surface
Output	Calculated urban density metric (hab/km^2)	print(densite)

Table 1.1: A simplified urban data processing pipeline illustrating the flow from raw input data to derived output metrics.

What is Python?

Python is a high-level programming language (close to human language), interpreted (the code is executed line by line), free and open source. It was created by **Guido van Rossum** (Netherlands). Its name comes from the British comedy group *Monty Python*, not from the snake.

Python is particularly well suited to smart urbanism for several reasons:

- its syntax is simple, which makes it ideal for beginners;
- it has a huge ecosystem for data: `pandas` (tables), `matplotlib` (charts), `geopandas` (maps and GIS), `numpy` (numerical computing);
- it is widely used in spatial analysis, smart cities, IoT sensors and artificial intelligence.

Version History

In this course we use **Python 3**. If you find code online using `print "hello"` (without parentheses), it is written in Python 2 and will not work on your computer.

Version	Release Date	Remark
Python 0.9	February 1991	First public release
Python 1.0	January 1994	
Python 2.0	October 2000	
Python 3.0	December 2008	Break with version 2 (not backward compatible)
Python 2.7	July 2010	Last version of the 2.x branch
Python 2 (end)	January 1, 2020	End of official support
Python 3.x	Every year in October	Current versions: 3.13, 3.14, ...

Table 1.2: Main milestones in the history of Python.

Tips for Learning Python

1. **Practise every day**, even 20 minutes: programming is learned like a language, by using it.
2. **Type the code yourself**, do not copy and paste: your fingers and your brain memorize it.
3. **Do not be afraid of errors.** An error message is not a failure: it tells you what is wrong. Read it, especially the last line.
4. **Test step by step**: one instruction, one result, then the next one.
5. **Modify the examples**: change a value and observe what happens.
6. **Apply it to your own topics** (urban planning, maps, data from your commune): it is more motivating.
7. **Search online** (official documentation, forums): every programmer does it.
8. **Break a problem into small steps** before coding.

Python Programming Environment

Executing Python code for spatial and demographic analysis requires a robust software execution environment. The default standard for data science and urban analytics is the **Anaconda Distribution** (figure 1.1), a unified package management system that bundles the core Python interpreter with spatial, mathematical, and tabular data analysis libraries.

Within the Anaconda ecosystem, interactive development is primarily conducted using Jupyter Notebook or JupyterLab (figure 1.2). Unlike traditional script execution where code is processed entirely in a batch format, Jupyter provides an interactive computing environment structured into discrete execution cells.

Setting up an isolated computation environment prevents library version conflicts across projects. Urban planners create virtual environments using terminal commands managed by the conda framework. These commands should be executed in the anaconda prompt or terminal interface, you can refer to figure 1.3 to see where to find it in the Anaconda Navigator interface.

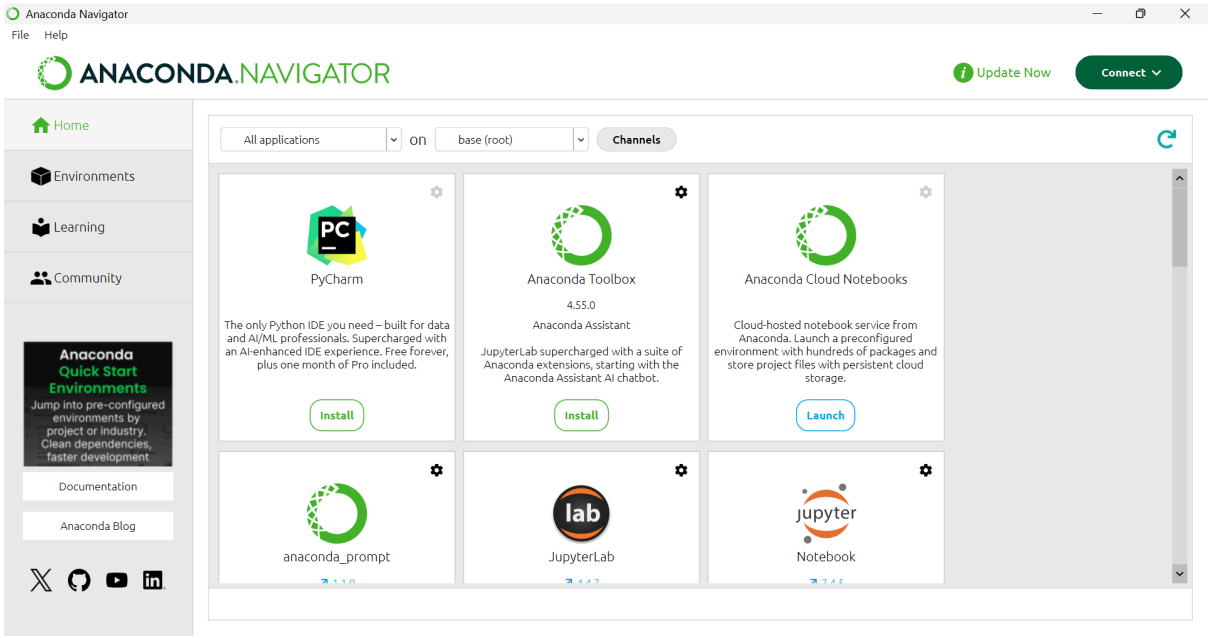


Figure 1.1: This is the main Anaconda Navigator interface, which provides access to various data science tools and environments.



Figure 1.2: This is the main JupyterLab interface, which provides an interactive computing environment for data science tasks.

```
conda create --name urban_env python=3.10
conda activate urban_env
```

Once the environment is active, launching JupyterLab initializes a local web server interface:

```
jupyter lab
```

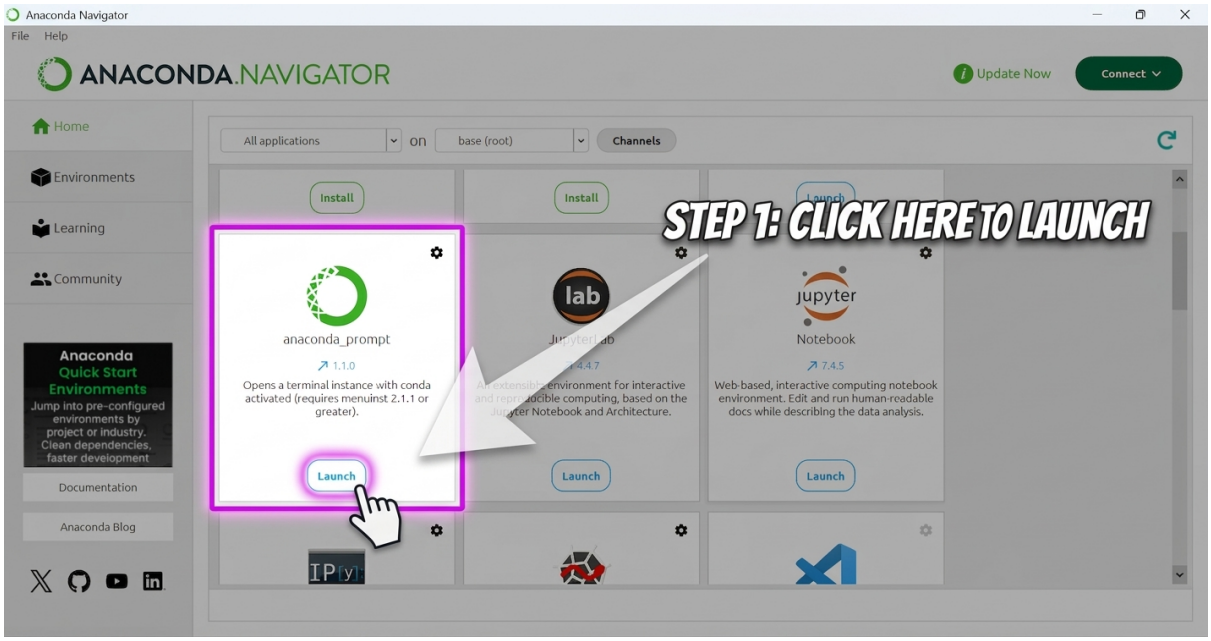


Figure 1.3: This is how to open the Anaconda Prompt from the Anaconda Navigator interface. The Anaconda Prompt is a command-line interface that allows users to manage conda environments and execute Python commands.

A Jupyter Notebook (.ipynb) (figure 1.2) consists of two main cell types:

- **Code Cells**: Executable environments containing valid Python instructions. When executed, the underlying computational engine (the Kernel) processes the instructions sequentially and renders the resulting output directly beneath the cell.
- **Markdown Cells**: Formatted text environments used to write documentation, formulate mathematical equations using LaTeX notation, and explain spatial planning workflows.

The **Kernel** operates as an active background process executing code and maintaining state variables in system memory. If an infinite loop or unhandled exception occurs, the kernel can be interrupted or restarted. Restarting the kernel clears all dynamic state memory, returning the session to a clean baseline.

Interactive Mode and Python as a Calculator

Open the **Anaconda Prompt** (figure 1.3) and type `python`. The `>>` symbol appears: Python is waiting for instructions. This is the **interactive mode**: you type an instruction, press *Enter*, and the result is displayed immediately.

```
>>> print("Hello Annaba")
Hello Annaba
>>> print('Smart Urbanism')
Smart Urbanism
```

Text is written between quotation marks (single ' or double "). Python can also be used as a calculator:

```
>>> 2 + 3
5
>>> 257000 / 49
5244.897959183673
>>> round(257000 / 49)
5245
```

Our warm-up exercise is now solved in a single line. The arithmetic operators are summarized in table 1.3; priorities are those of mathematics, and parentheses can be used when needed (`(2 + 3) * 4` gives 20).

Operator	**Meaning**	**Example**	**Result**	
+	Addition	`7 + 2`	9	
-	Subtraction	`7 - 2`	5	
*	Multiplication	`7 * 2`	14	
/	Division	`7 / 2`	3.5	
//	Integer division	`7 // 2`	3	
%	Remainder (modulo)	`7 % 2`	1	
**	Power	`7 ** 2`	49	

Table 1.3: Arithmetic operators in Python.

To leave the interactive mode, type:

```
>>> exit()
```

Initial Code Execution

Programmatic analysis begins with output commands. The primary output function in Python is print(), which transfers variable contents or literal values to the console output.

Code 1.1: Python Output Commands

```
1 print("Bonjour")
2 print("Analyse urbaine")
```

When evaluating these instructions, the interpreter outputs each string sequentially:

```
Bonjour Analyse urbaine
```

Urban data is retained in memory by binding scalar values to named variables. Assigning municipal population metrics to dynamic variables allows for downstream calculations:

Code 1.2: Python Variable Assignment and Output

```
1 population = 257000
2 print(population)
```

As shown in Listing 1.2, the variable `population` is assigned a numeric value of 257000. When printed, the interpreter outputs the stored value:

```
257000
```

Error Diagnostics and Debugging

Programming requires identifying and correcting operational exceptions. Intentionally introducing syntax or reference errors helps build diagnostic skills. Consider the deliberate typographic error below:

Code 1.3: Python Error Example

```
1 population = 257000
2 print(populaton)
```

Executing this block causes the Python interpreter to stop execution and produce a traceback message:

```
NameError: name 'populaton' is not defined
```

As shown, the interpreter identifies a `NameError` due to the misspelled variable name. Correcting the variable reference resolves the error and allows successful execution.

Understanding error classifications is essential for rapid debugging and continuous script development:

- **Syntax Errors**: Violations of Python's grammatical rules, such as missing colons or parentheses.
- **Name Errors**: References to undefined variables or functions.
- **Type Errors**: Operations performed on incompatible data types, such as adding a string to an integer.
- **Index Errors**: Accessing elements outside the bounds of a list or array.
- **Value Errors**: Passing arguments of the correct type but inappropriate value to a function.
- **Runtime Errors (Exception)**: Errors occurring during script execution, such as attempting to divide a municipal population value by a land surface area assigned a value of zero (ZeroDivisionError).

Applied Introductory Practice

New programmers should practice writing simple scripts that combine variable assignment, arithmetic operations, and output commands. For example, displaying the name of a city and its population:

Code 1.4: Displaying City Name and Population

```python
nom = "Annaba"
population = 257000
print(nom)
print(population)
```

The output of code 1.4 will display the city name followed by its population:

```
Annaba
257000
```

Another example, calculating the population density of a city given its population and land area:

Code 1.5: Population Density Calculation

```python
population = 257000  # Total population
surface = 49         # Land area in square kilometers
density = population / surface  # Calculate population density
print("Population Density:", density, "hab/km^2")  # Output the result
```

The output of code 1.5 will display the calculated population density:

```
Population Density: 5244.897959183673 hab/km^2
```

Comments

A **comment** is text ignored by Python, written for humans (you, your colleagues, your future self). It starts with the # symbol and extends to the end of the line. A good comment explains the *why*, not only the *what*.

Code 1.6: Using Comments

```python
# Density of the commune of Annaba
population = 257000    # number of inhabitants (approximate value)
surface = 49           # area in km2
print(population / surface)
```

Variables and Data Types

Definition of a Variable

A **variable** is a name that refers to a value stored in the computer's memory. It can be seen as a *labelled box* containing a piece of information. The = sign is the **assignment** operator: it puts the value on the right into the variable on the left.

Code 1.7: Using Variables to Compute a Density

```python
population = 257000
surface = 49
density = population / surface
print(density)
```

The main advantage is flexibility: if the population changes, only one line must be modified and the whole calculation follows. A variable can also be updated using its own value:

Code 1.8: Updating a Variable

```
population = 257000
population = population + 5000     # the population grows
print(population)
```

```
262000
```

Data Types

Every value has a **type**, which can be obtained with the `type()` function. The four basic types are listed in table 1.4.

Type	**Python Name**	**Description**	**Example**
Integer	`int`	Number without decimals	`257000`
Decimal	`float`	Number with decimals (written with a dot)	`5244.9`
Text	`str`	String of characters	`"Annaba"`
Boolean	`bool`	True or false	`True`, `False`

Table 1.4: Basic data types in Python.

Code 1.9: Checking the Type of a Variable

```
city = "Annaba"
population = 257000
density = 5244.9
is_coastal = True

print(type(city))
print(type(population))
print(type(density))
print(type(is_coastal))
```

```
<class 'str'>
<class 'int'>
<class 'float'>
<class 'bool'>
```

In Python, the decimal separator is the **dot**, not the comma: write `3.5`, not `3,5`. Booleans are written with a capital letter: `True` and `False`.

Variable Names

Python imposes the following rules on variable names:

- only letters, digits and the underscore _ are allowed;
- a name must **never** start with a digit;
- spaces and special characters (`-`, `$`, `@`, ...) are forbidden (accents are also best avoided);
- reserved words of Python (`if`, `for`, `while`, `True`, ...) cannot be used;

- names are case-sensitive: `City` and `city` are two different variables.

As a good practice, choose clear names in lowercase, using _ to separate words (table 1.5). A name such as `population_annaba` is far more readable than `p` or `x`.

Valid Name	**Invalid Name**
`population_annaba`	`2nd_city` (starts with a digit)
`area_km2`	`area km2` (contains a space)
`nb_dwellings`	`growth-rate` (contains a hyphen)
`density`	`for` (reserved word)

Table 1.5: Examples of valid and invalid variable names.

Scientific Notation

Very large or very small numbers are written with the letter `e`, meaning "times ten to the power of". A number written this way is always of type `float`.

Code 1.10: Scientific Notation

```
area_m2 = 4.9e7          # 4.9 x 10^7 = 49,000,000 m2
population = 2.57e5      # 2.57 x 10^5 = 257,000
thickness = 1.5e-3       # 1.5 x 10^-3 = 0.0015
print(area_m2)
```

```
49000000.0
```

Operators

The arithmetic operators of table 1.3 apply to variables. Python also provides shortcut operators that update a variable in place (table 1.6).

Shortcut	**Equivalent**	**Meaning**
`x += 5`	`x = x + 5`	Add 5
`x -= 5`	`x = x - 5`	Subtract 5
`x *= 2`	`x = x * 2`	Double
`x /= 2`	`x = x / 2`	Divide by 2

Table 1.6: Shortcut assignment operators.

Code 1.11: Shortcut Operators

```
population = 257000
population += 5000       # births and new arrivals
print(population)
```

```
262000
```

Operators on Strings

Only two operators make sense on text: `+` performs **concatenation** (joining two strings) and `*` performs **repetition** (string multiplied by an integer).

Code 1.12: Operators on Strings

```
city = "Annaba"
country = "Algeria"
print(city + country)
print(city + " - " + country)
print("-" * 20)
print("City " * 3)
```

```
AnnabaAlgeria
Annaba - Algeria
--------------------
City City City
```

Note that `"Annaba" + 257000` causes a `TypeError`: text and numbers cannot be added directly. This is solved by type conversion.

Type Conversion

A value can be converted from one type to another with the functions `int()`, `float()` and `str()`.

Code 1.13: Type Conversion

```
population = 257000
text = "Population: " + str(population)     # number -> text
print(text)

string = "49"
surface = int(string)                       # text -> integer
print(surface * 2)

print(int(3.99))       # the decimal part is cut off
print(float(5))
```

```
Population: 257000
98
3
5.0
```

Exercises (Set 1)

1. **A plot of land.** A rectangular building plot is 25 m wide and 40 m long.
   (a) Store these values in two variables.
   (b) Compute and display the area and the perimeter of the plot.
   (c) The land costs 45 000 DA/m^2. Compute the total price.

2. **Footprint ratio.** The footprint ratio is the ratio between the built area on the ground and the area of the plot. A plot of 600 m^2 carries a building whose ground footprint is 240 m^2. Compute the footprint ratio, then the remaining area available for a garden.
3. **Unit conversion.** A neighbourhood covers 3 500 000 m^2. Using scientific notation for the starting value, express its area in hectares (1 ha = 10 000 m^2) and in km^2.
4. **Text.** From the variables `city = "Annaba"` and `population = 257000`, build and display the sentence *Annaba has 257000 inhabitants.*

Output and Input

The print() Function and Its Forms

Displaying Several Items

Items separated by commas are displayed with a space between them, and may have different types.

Code 1.14: Printing Several Items

```
city = "Annaba"
population = 257000
print("City:", city)
print(city, "has", population, "inhabitants")
```

```
City: Annaba
Annaba has 257000 inhabitants
```

The sep and end Parameters

The `sep` parameter defines the separator between items (a space by default), and `end` defines what is added at the end of the output (a new line by default).

Code 1.15: The sep and end Parameters

```
print("Annaba", "Algiers", "Oran", sep=" - ")
print("2026", "10", "09", sep="/")

print("Annaba", end=" | ")
print("Algiers", end=" | ")
print("Oran")
```

```
Annaba - Algiers - Oran
2026/10/09
Annaba | Algiers | Oran
```

Special Characters

Within a string, `\n` inserts a new line and `\t` inserts a tabulation.

Code 1.16: Special Characters

```
print("Line 1\nLine 2")
print("City\tPopulation")
```

```
Line 1
Line 2
City     Population
```

Formatted Strings (f-strings)

The most convenient form places an `f` before the quotation marks and writes variables between curly braces. The specifier `:.2f` means "display with 2 digits after the decimal point".

Code 1.17: Formatted Strings

```
city = "Annaba"
population = 257000
surface = 49
density = population / surface

print(f"{city} has {population} inhabitants")
print(f"Density of {city}: {density} inh/km2")
print(f"Density of {city}: {density:.2f} inh/km2")
print(f"Density of {city}: {density:.0f} inh/km2")
print(f"Population: {population:,}")
```

```
Annaba has 257000 inhabitants
Density of Annaba: 5244.897959183673 inh/km2
Density of Annaba: 5244.90 inh/km2
Density of Annaba: 5245 inh/km2
Population: 257,000
```

Reading Data with input()

The `input()` function pauses the program, displays a message, and waits for the user to type an answer followed by *Enter*.

Code 1.18: Reading Text from the User

```
name = input("What is your name? ")
print("Hello", name)
```

Important: `input()` *always* returns text (`str`), even if the user types a number. To perform calculations, the answer must be converted with `int()` or `float()`.

Code 1.19: Reading Numbers from the User

```
population = int(input("Population of the city: "))
surface = float(input("Area in km2: "))

density = population / surface
print(f"Density: {density:.1f} inh/km2")
```

Without the conversion, an expression such as `"257000" / 49` would raise a `TypeError`.

Exercise 5: Density Calculator

Write a program that asks for the name of a city, its population and its area, then displays: *The density of [city] is [x] inh/km^2.* with one digit after the decimal point.

Lists

Definition

A **list** is a variable that contains several *ordered* values. It is written between square brackets, with elements separated by commas. Without lists, one variable per city would be needed (`city1`, `city2`, ...).

Code 1.20: Creating a List of Cities

```
cities = ["Annaba", "Algiers", "Oran", "Constantine"]
print(cities)
```

```
['Annaba', 'Algiers', 'Oran', 'Constantine']
```

A list can contain values of different types and can be empty (`empty = []`).

Usage: Indexes

Each element of a list has a position called an **index**. Counting starts at **0**, not at 1 (table 1.7). Negative indexes count from the end of the list.

Element	"Annaba"	"Algiers"	"Oran"	"Constantine"
Index	0	1	2	3
Negative index	-4	-3	-2	-1

Table 1.7: Indexes of the elements of the list `cities`.

Code 1.21: Accessing List Elements

```
cities = ["Annaba", "Algiers", "Oran", "Constantine"]
print(cities[0])
print(cities[2])
print(cities[-1])
print(cities[4])
```

```
Annaba
Oran
Constantine
IndexError: list index out of range
```

An element is modified by assigning a new value to its index. **Slicing** extracts a part of the list: `list[start:stop]` goes from `start` (included) to `stop` (excluded).

Code 1.22: Modifying and Slicing a List

```
cities = ["Annaba", "Algiers", "Oran", "Constantine"]
cities[1] = "Algiers-Centre"
print(cities)
print(cities[1:3])
print(cities[:2])
print(cities[2:])
```

```
['Annaba', 'Algiers-Centre', 'Oran', 'Constantine']
['Algiers-Centre', 'Oran']
['Annaba', 'Algiers-Centre']
['Oran', 'Constantine']
```

Lists of numbers are equally common, for instance a list of populations (approximate values) aligned with the list of cities: `populations = [257000, 3000000, 1500000, 450000]`.

Operations on Lists

Operation	Purpose	Example
`append(x)`	Adds `x` at the end	`cities.append("Setif")`
`insert(i, x)`	Inserts `x` at position `i`	`cities.insert(1, "Skikda")`
`remove(x)`	Removes the value `x`	`cities.remove("Oran")`
`pop(i)`	Removes and returns element `i`	`cities.pop(0)`
`sort()`	Sorts the list	`cities.sort()`
`reverse()`	Reverses the order	`cities.reverse()`
`+`	Joins two lists	`list1 + list2`
`*`	Repeats a list	`[0] * 5`
`in`	Tests membership	`"Annaba" in cities`

Table 1.8: Common operations on lists.

Code 1.23: Operations on Lists

```
1 cities = ["Annaba", "Algiers", "Oran"]
2 cities.append("Constantine")
3 print(cities)
4 cities.sort()
5 print(cities)
6 print("Oran" in cities)
7 print([0] * 5)
```

```
['Annaba', 'Algiers', 'Oran', 'Constantine']
['Algiers', 'Annaba', 'Constantine', 'Oran']
True
[0, 0, 0, 0, 0]
```

Useful Functions: len, range, list, min, max, sum

Code 1.24: Useful Functions on Lists

```
populations = [257000, 3000000, 1500000, 450000]

print(len(populations))      # number of elements
print(min(populations))      # smallest value
print(max(populations))      # largest value
print(sum(populations))      # sum of the values
print(sum(populations) / len(populations))    # average
```

```
4
257000
3000000
5207000
1301750.0
```

The `range()` function generates a sequence of integers: `range(5)` gives 0, 1, 2, 3, 4 (the stop value is **excluded**); `range(1, 5)` gives 1, 2, 3, 4; and `range(0, 10, 2)` gives 0, 2, 4, 6, 8 (the third number is the *step*). The `list()` function converts such a sequence, or a string, into a list.

Code 1.25: The range() and list() Functions

```
print(list(range(5)))
print(list(range(1, 11, 3)))
print(list("Annaba"))
```

```
[0, 1, 2, 3, 4]
[1, 4, 7, 10]
['A', 'n', 'n', 'a', 'b', 'a']
```

Control Flow: Loops

Why a Loop?

Suppose we want to display all the cities of a list, one by one:

Code 1.26: Repetitive Code Without a Loop

```
cities = ["Annaba", "Algiers", "Oran", "Constantine"]
print(cities[0])
print(cities[1])
print(cities[2])
print(cities[3])
```

This works, but it is impractical with the more than 1 500 communes of Algeria, and the code must be rewritten whenever the list changes size. The **loop** solves this problem.

Instructions, Blocks and Indentation

Before studying loops, three notions are required:

- An **instruction** is a line of code asking Python to do something (`print(...)`, `x = 3`, ...).
- A **block of instructions** is a group of instructions that go together and are executed within the same structure (a loop, a condition, ...).

- **Indentation** is the shift to the right (**4 spaces**, or the *Tab* key) at the beginning of a line. In Python, indentation *defines* the block. A line ending with a colon : announces that an indented block follows.

Code 1.27: Structure of a Block

```
instruction that announces a block:
    instruction 1 of the block
    instruction 2 of the block
instruction outside the block
```

Missing or inconsistent indentation causes an `IndentationError`: all lines of a block must have exactly the same shift.

The for Loop

Looping over a List

Code 1.28: A for Loop over a List

```
cities = ["Annaba", "Algiers", "Oran", "Constantine"]
for city in cities:
    print(city)
```

It reads: *for each* `city` *in* `cities`, *display* `city`. At each round (an *iteration*), the variable `city` takes the next value of the list. The result is identical to listing 1.26, but works for 4 or 4 000 cities. A block may contain several instructions; the lines that are not indented are outside the loop and run only once.

Code 1.29: A Loop with Several Instructions

```
cities = ["Annaba", "Algiers", "Oran", "Constantine"]
populations = [257000, 3000000, 1500000, 450000]

for i in range(len(cities)):
    print(f"{cities[i]}: {populations[i]} inhabitants")
print("End of the list")
```

```
Annaba: 257000 inhabitants
Algiers: 3000000 inhabitants
Oran: 1500000 inhabitants
Constantine: 450000 inhabitants
End of the list
```

Looping with range()

To repeat an action a known number of times, `for` is combined with `range()`. A variable used to accumulate a result is called an **accumulator**.

Code 1.30: A for Loop with range()

```
for i in range(1, 4):
    print("Station", i)

total = 0
for i in range(1, 6):
    total += i
print(total)
```

```
Station 1
Station 2
Station 3
15
```

Comparisons

A comparison produces a boolean (`True` or `False`), see table 1.9. Be careful not to confuse `=` (assignment) with `==` (comparison). Comparisons can be combined with `and`, `or` and `not`.

Operator	Meaning	Example	
`==`	Equal to	`5 == 5` gives `True`	
`!=`	Different from	`5 != 3` gives `True`	
`<`	Strictly less than	`3 < 5` gives `True`	
`>`	Strictly greater than	`3 > 5` gives `False`	
`<=`	Less than or equal to	`5 <= 5` gives `True`	
`>=`	Greater than or equal to	`4 >= 5` gives `False`	

Table 1.9: Comparison operators.

Comparisons are used in an `if` condition, which follows the same logic of an indented block:

Code 1.31: Comparison Inside a Loop

```
cities = ["Annaba", "Algiers", "Oran", "Constantine"]
populations = [257000, 3000000, 1500000, 450000]

for i in range(len(cities)):
    if populations[i] > 1000000:
        print(cities[i], ": large metropolis")
    else:
        print(cities[i], ": medium-sized city")
```

```
Annaba : medium-sized city
Algiers : large metropolis
Oran : large metropolis
Constantine : medium-sized city
```

The while Loop

A `while` loop repeats a block **as long as a condition is true**. It is used when the number of rounds is not known in advance. For example, in how many years will the population of Annaba reach 500 000 inhabitants with a growth of 2 % per year?

Code 1.32: A while Loop: Population Growth

```
population = 257000
years = 0
while population < 500000:
    population = population * 1.02
    years += 1
print(f"It takes {years} years")
```

```
It takes 34 years
```

If the condition never becomes false (for example if the update of the variable is forgotten), the loop never stops: this is an **infinite loop**. In that case, the execution must be interrupted manually (with `Ctrl+C` in the Anaconda Prompt, or by interrupting the kernel in Jupyter). In summary, use `for` when the number of repetitions is known or when going through a list, and `while` when repeating until a condition changes.

Conditional Statements: if, elif, else

So far, a program executes all its instructions, one after the other. But a program must often **make decisions**: *if* the density of a zone exceeds a threshold, classify it as urban; *if* a parcel is in a flood zone, refuse the building permit. A **conditional statement** executes a block of instructions *only when a condition is true*. The condition is an expression that gives `True` or `False`, such as the comparisons of table 1.9.

The if Statement

The `if` keyword is followed by a condition and a colon; the indented block below is executed only if the condition is true. The syntax is the same as for loops: the colon announces the block, and the indentation defines it.

Code 1.33: A Simple if Statement

```
population = 257000

if population > 200000:
    print("This is a large city")
print("End of the program")
```

```
This is a large city
End of the program
```

The last line is not indented: it is outside the block, so it is executed in all cases. If `population` were 150 000, only `End of the program` would be displayed.

The else Clause

`else` defines the block executed when the condition is false. Exactly one of the two blocks is executed.

Code 1.34: if with else

```
population = 150000

if population > 200000:
    print("Large city")
```

```
else:
    print("Medium-sized or small city")
```

```
Medium-sized or small city
```

Several Cases with elif

When there are more than two cases, `elif` (short for *else if*) adds further conditions. Python tests the conditions **from top to bottom** and executes the block of the **first true one**, ignoring all the others. The final `else` covers everything that remains. For example, a classification of the density of a zone (the thresholds are didactic, not official):

Code 1.35: Classifying a Density with if, elif and else

```
density = 5245          # inhabitants per km2

if density >= 3000:
    category = "very dense urban area"
elif density >= 1000:
    category = "urban area"
elif density >= 100:
    category = "peri-urban area"
else:
    category = "rural area"

print(category)
```

```
very dense urban area
```

The order of the tests matters. If the tests were written from the smallest threshold to the largest, a density of 5 245 would satisfy the very first one and be classified as peri-urban, the following tests never being examined:

Code 1.36: A Wrong Order of Tests (to Avoid)

```
density = 5245

if density >= 100:
    print("peri-urban area")        # true: the other tests are ignored
elif density >= 1000:
    print("urban area")
```

```
peri-urban area
```

Combining Conditions: and, or, not

Conditions can be combined with three logical operators (table 1.10).

For instance, a parcel is buildable if its area is at least 200 m^2 and it is not located in a flood zone:

Code 1.37: Combining Conditions with and and not

```
area = 320
flood_zone = False

```

Operator	Result	Example
`and`	True if *both* conditions are true	`area >= 200 and not flood`
`or`	True if *at least one* condition is true	`zone == "urban" or zone == "peri-urban"`
`not`	Reverses a condition	`not flood`

Table 1.10: Logical operators.

```
if area >= 200 and not flood_zone:
    print("Buildable parcel")
else:
    print("Non-buildable parcel")
```

```
Buildable parcel
```

Two more tools make conditions shorter. The operator `in` tests whether a value belongs to a list (or a text), and Python allows chained comparisons, as in mathematics:

Code 1.38: The in Operator and Chained Comparisons

```
zone = "peri-urban"
if zone in ["urban", "peri-urban"]:
    print("Urbanized zone")

population = 45000
if 10000 <= population < 50000:          # same as: >= 10000 and < 50000
    print("Small town")
```

```
Urbanized zone
Small town
```

Conditions and User Input

Combined with `input()` and type conversion, `if` makes interactive programs. Below, an example of a run is displayed, with the answer typed by the user after the colon.

Code 1.39: An Interactive Program with a Condition

```
population = int(input("Population: "))
surface = float(input("Area in km2: "))
density = population / surface

if density >= 1000:
    print(f"Density {density:.1f}: urban area")
else:
    print(f"Density {density:.1f}: not an urban area")
```

```
Population: 257000
Area in km2: 49
Density 5244.9: urban area
```

Conditions Inside Loops

A condition inside a loop acts on every element. Together with a **counter** (a variable that increases each time a condition is met), it answers questions such as "how many cities exceed a threshold?".

Code 1.40: A Counter Inside a Loop

```
cities = ["Algiers", "Oran", "Annaba", "Tamanrasset"]
populations = [2988145, 1584607, 609499, 115043]       # wilayas, 2008
    census

count = 0
for i in range(len(cities)):
    if populations[i] > 500000:
        count += 1
        print(cities[i], "exceeds 500,000 inhabitants")

print(count, "wilayas out of", len(cities))
```

```
Algiers exceeds 500,000 inhabitants
Oran exceeds 500,000 inhabitants
Annaba exceeds 500,000 inhabitants
3 wilayas out of 4
```

Common Mistakes

- **Confusing** `=` **and** `==`: `=` assigns a value, `==` compares. `if x = 5:` is a syntax error.
- **Forgetting the colon** after the condition, or the **indentation** of the block (`IndentationError`).
- **Comparing texts carelessly**: `"Urban" == "urban"` is `False`, since capitals matter. Texts coming from files should be cleaned before comparison (see Chapter 2).
- **Wrong order of the** `elif` **tests** (see above).
- **Comparing a text and a number** read with `input()`: convert first with `int()` or `float()`.

Functions

We have already used many **functions** supplied by Python: `print()`, `len()`, `sum()`, `range()`, `input()`, `int()`, ... Each has a name, receives values between parentheses, and does a job. We can also write **our own functions**.

Why Write Functions?

Imagine that the density must be computed for several zones:

Code 1.41: The Same Calculation Repeated

```
density_annaba = 609499 / 1439
density_oran = 1584607 / 2121
density_algiers = 2988145 / 1190
print(round(density_annaba, 1), round(density_oran, 1), round(
    density_algiers, 1))
```

```
423.6 747.1 2511.0
```

The same formula is copied three times. If it contains an error, or if it changes, it must be corrected everywhere. A function writes the calculation **once** and gives it a name, which makes the program shorter, easier to read and easier to correct.

Defining and Calling a Function

A function is defined with the keyword `def`, followed by its name, its **parameters** between parentheses, and a colon. The indented block is the **body** of the function. The keyword `return` gives back the result.

Code 1.42: Structure of a Function

```
def function_name(parameter1, parameter2):
    instructions
    return result
```

Code 1.43: Our First Function: density()

```
def density(population, surface):
    return round(population / surface, 1)

print(density(609499, 1439))
print(density(1584607, 2121))
print(density(2988145, 1190))
```

```
423.6
747.1
2511.0
```

Two distinct steps must be understood:

- the **definition** (`def ...`) only *describes* what the function does: nothing is computed and nothing is displayed;
- the **call** (`density(609499, 1439)`) executes the function with real values, called **arguments**. The arguments are matched with the parameters in order: `population` receives 609499 and `surface` receives 1439.

The value returned by `return` can be displayed, stored in a variable, or used in a calculation:

Code 1.44: Using the Returned Value

```
d = density(609499, 1439)
print(d * 2)
```

```
847.2
```

Default Values and Keyword Arguments

A parameter can have a **default value**, used when the argument is omitted. Arguments can also be given by **name** (*keyword arguments*), in any order, which makes calls more readable. The function below projects a population after a number of years of constant growth:

Code 1.45: Default Values and Named Arguments

```
def projection(population, rate=0.02, years=10):
    return round(population * (1 + rate) ** years)

print(projection(609499))                          # default rate and years
print(projection(609499, 0.03))                    # rate of 3 %
print(projection(609499, years=20))                # 20 years at the
    default rate
print(projection(years=5, population=609499))      # named arguments, any
    order
```

```
742976
819116
905683
672936
```

print() Versus return

A very common confusion: `print()` *displays* a value on screen, whereas `return` *gives it back* to the program that called the function. A function without `return` gives back the special value `None`:

Code 1.46: A Function Without return Gives None

```
def show_density(population, surface):
    print(round(population / surface, 1))      # displays, but returns
        nothing

result = show_density(609499, 1439)
print(result)
```

```
423.6
None
```

The density was displayed by the function, but the variable `result` is empty (`None`) and cannot be used in later calculations. **Rule of thumb:** a function that computes something should `return` it, and leave the display to the main program.

Functions and Conditions

The body of a function can contain any instruction seen before, in particular conditions. A function can have several `return`; as soon as one is executed, the function ends. This function classifies a density (the thresholds are didactic):

Code 1.47: Combining Functions, Conditions and a Loop

```
def classify(density):
    if density >= 1000:
        return "very dense"
    elif density >= 100:
        return "dense"
    elif density >= 10:
        return "sparse"
    else:
        return "very sparse"

names = ["Algiers", "Oran", "Annaba", "Tamanrasset"]
```

```python
populations = [2988145, 1584607, 609499, 115043]       # 2008 census
areas = [1190, 2121, 1439, 336839]                     # km2

for i in range(len(names)):
    d = density(populations[i], areas[i])
    print(f"{names[i]}: {d} inh/km2 -> {classify(d)}")
```

```
Algiers: 2511.0 inh/km2 -> very dense
Oran: 747.1 inh/km2 -> dense
Annaba: 423.6 inh/km2 -> dense
Tamanrasset: 0.3 inh/km2 -> very sparse
```

Notice how the loop reads almost like a sentence, because the details are hidden in two well-named functions. This is the main benefit of functions: **each one does one thing, and has a name that says it**.

Returning Several Values

A function can return several values separated by commas; the caller can receive them in several variables at once.

Code 1.48: A Function Returning Two Values

```python
def extremes(values):
    return min(values), max(values)

low, high = extremes(populations)
print("Smallest:", low)
print("Largest:", high)
```

```
Smallest: 115043
Largest: 2988145
```

Local Variables

The variables created *inside* a function (including its parameters) are **local**: they exist only during the call, and are invisible outside. This protects the rest of the program from accidental interference.

Code 1.49: A Local Variable is Not Visible Outside its Function

```python
def density(population, surface):
    result = population / surface
    return result

density(609499, 1439)
print(result)
```

```
NameError: name 'result' is not defined
```

Documenting a Function

A text placed between triple quotes on the first line of the body, a **docstring**, documents the function. It states what the function does, what its parameters are and what it returns. Combined with a clear name, it makes a function usable by a colleague (or by yourself, six months later).

Code 1.50: A Function with a Docstring

```
def hectares(area_m2):
    """Convert an area from square metres to hectares."""
    return area_m2 / 10000

print(hectares(3500000))
help(hectares)
```

```
350.0
Help on function hectares:

hectares(area_m2)
    Convert an area from square metres to hectares.
```

The built-in function `help()` displays the docstring of a function, including those of Python's own functions (try `help(len)`).

Summary

- `def name(parameters):` defines a function; the indented block is its body.
- A function does nothing until it is **called**; arguments are matched with parameters by position or by name.
- `return` gives a value back; `print()` only displays. Without `return`, the result is `None`.
- Parameters can have default values; a function can return several values.
- Variables created in a function are local to it.

Data File Formats: CSV, JSON and Excel

Urban data lives in files. Three formats cover most of what an urban planner will receive: **CSV** (exports of databases, open data portals), **JSON** (web services and APIs) and **Excel** (administrations, surveys, reports). They are not interchangeable: each one has strengths and traps. To compare them, we write the *same data* (three wilayas, 2008 census) in each format, using Python.

Code 1.51: The Data to Save: a List of Rows

```
rows = [
    ["wilaya", "population", "area_km2"],
    ["Annaba", 609499, 1439],
    ["Constantine", 938475, 2187],
    ["Oran", 1584607, 2121],
]
```

CSV: the Simplest Table

A **CSV** file (*Comma-Separated Values*) is a plain *text* file: one line per row, values separated by a **separator**. Despite its name, the separator is often a semicolon in French-language files, because the comma is used as the decimal mark. Python provides the `csv` module. In the code below, `with open(...) as f:` opens a file (`"w"` for writing) and closes it automatically at the end of the block.

Code 1.52: Writing a CSV File

```
import csv
```

```
2
3 with open("wilayas.csv", "w", newline="") as f:
4     writer = csv.writer(f, delimiter=";")
5     writer.writerows(rows)
6
7 print(open("wilayas.csv").read())
```

```
wilaya;population;area_km2
Annaba;609499;1439
Constantine;938475;2187
Oran;1584607;2121
```

The file is readable in any text editor. To read it, `csv.DictReader` gives each row as a dictionary whose keys are the column names:

Code 1.53: Reading a CSV File

```
1 with open("wilayas.csv") as f:
2     reader = csv.DictReader(f, delimiter=";")
3     for row in reader:
4         print(row)
5
6 print(type(row["population"]))
```

```
{'wilaya': 'Annaba', 'population': '609499', 'area_km2': '1439'}
{'wilaya': 'Constantine', 'population': '938475', 'area_km2': '2187'}
{'wilaya': 'Oran', 'population': '1584607', 'area_km2': '2121'}
<class 'str'>
```

Important limitation: a CSV file contains only text. Even though 609499 was written as a number, it is read back as the text `'609499'` (type `str`), and must be converted with `int()` before any calculation. A CSV file also has no standard way to indicate the separator, the decimal mark, or the *encoding* of accented characters (UTF-8 or the Windows encoding): these three details are the main source of errors when opening a CSV file received from someone else.

JSON: Structured Data

JSON (*JavaScript Object Notation*) is a text format whose syntax looks very much like Python lists and dictionaries. Unlike CSV, it **preserves types** (numbers, booleans, text) and allows **nested** structures: an object can contain another object or a list. It is the standard format of web services (APIs), and the base of the **GeoJSON** format used for geographic data (Chapter 2). The `json` module converts between Python objects and JSON.

Code 1.54: Writing a JSON File

```
1 import json
2
3 wilayas = [
4     {"wilaya": "Annaba", "population": 609499, "area_km2": 1439,
5      "coastal": True, "location": {"lon": 7.77, "lat": 36.90}},
6     {"wilaya": "Constantine", "population": 938475, "area_km2": 2187,
7      "coastal": False, "location": {"lon": 6.61, "lat": 36.37}},
8     {"wilaya": "Oran", "population": 1584607, "area_km2": 2121,
9      "coastal": True, "location": {"lon": -0.64, "lat": 35.69}},
10 ]
11
```

```python
with open("wilayas.json", "w") as f:
    json.dump(wilayas, f, indent=2)

print(open("wilayas.json").read()[:230])
```

```
[
  {
    "wilaya": "Annaba",
    "population": 609499,
    "area_km2": 1439,
    "coastal": true,
    "location": {
      "lon": 7.77,
      "lat": 36.9
    }
  },
  {
    "wilaya": "Constantine",
    "population": 938475,
    "a
```

Each wilaya contains an inner object `location`: this hierarchy cannot be expressed in a simple table without flattening it into extra columns. Note also that Python's `True` is written `true` in JSON. Reading restores the original Python objects and their types:

Code 1.55: Reading a JSON File

```python
with open("wilayas.json") as f:
    data = json.load(f)

print(data[0]["location"]["lat"])
print(type(data[0]["population"]))
print(type(data[0]["coastal"]))
```

```
36.9
<class 'int'>
<class 'bool'>
```

The functions `json.dumps()` and `json.loads()` (with an `s`, for *string*) do the same conversion with a text in memory instead of a file, which is what is needed when a web service returns JSON.

Excel: the Workbook

An **Excel** file (`.xlsx`) is a *workbook*: it contains several **sheets**, typed cells, formulas, colours, merged cells, charts. This is convenient for people but heavy for programs: the file is a compressed binary archive that cannot be read with a text editor, and Python needs an additional library to open it. The most convenient way is the `pandas` library, which is the subject of Chapter 2; here it is used only as a tool to write and read Excel files (the `openpyxl` library, which does the actual work, is included in Anaconda).

Code 1.56: Writing and Reading an Excel File

```python
import pandas as pd

table = pd.DataFrame(rows[1:], columns=rows[0])
table.to_excel("wilayas.xlsx", sheet_name="census", index=False)

```

```python
back = pd.read_excel("wilayas.xlsx", sheet_name="census")
print(back)
print(back["population"].dtype)
```

```
        wilaya  population  area_km2
0       Annaba      609499      1439
1  Constantine      938475      2187
2         Oran     1584607      2121
int64
```

Excel keeps the numeric type here (`int64`). The parameter `sheet_name` selects the sheet to read. Real administrative spreadsheets are however full of traps: titles above the table, merged cells, totals at the bottom, notes, and values that Excel has silently *reinterpreted*. A classic example: a code such as the wilaya number `05` (Batna) becomes the number 5, and the leading zero is lost.

Comparing the Three Formats

The three files contain the same information, but they are very different objects:

Code 1.57: Size of the Three Files

```python
import os

for name in ["wilayas.csv", "wilayas.json", "wilayas.xlsx"]:
    print(name, os.path.getsize(name), "bytes")
```

```
wilayas.csv 92 bytes
wilayas.json 499 bytes
wilayas.xlsx 5393 bytes
```

Which Format to Choose?

- To **exchange or archive a flat table**, prefer **CSV**: it is universal, light, and readable forever. Always state the separator and use the UTF-8 encoding.
- To **exchange hierarchical data** or to communicate with a **web service**, use **JSON**. Geographic data in GeoJSON is JSON.
- **Excel** is appropriate for people (a report, a form filled by hand, a table with formatting), and often unavoidable because administrations distribute their data this way. For a program, however, convert it to CSV as soon as possible, and keep the Excel file as the original source.

In all cases, the first reflex after loading a file is to **look at the data** before using it.

Exercises (Set 2)

Exercises 6 to 8 concern loops, exercises 9 and 10 conditions, exercises 11 and 12 functions, and exercise 13 file formats.

6. **The cities.** Given `cities = ["Annaba", "Algiers", "Oran", "Constantine"]` and `populations = [257000, 3000000, 1500000, 450000]`:
   (a) Using a `for` loop, display one sentence per city, for example *Annaba: 257000 inhabitants*.

	CSV	**JSON**	**Excel (.xlsx)**
Nature	Plain text	Plain text	Compressed binary file
Structure	One flat table	Nested objects and lists	Several sheets of cells
Types kept	No (all text)	Yes (numbers, booleans, text)	Yes, with risks of automatic conversion
Readable by a human	Yes, in any editor	Yes, in any editor	Only in Excel or similar software
Size	Very small	Larger (names repeated)	Larger (archive)
Typical use	Exports, open data, large tables	Web services, APIs, GeoJSON, settings	Administrative reports, surveys, manual editing
Main trap	Separator, decimal mark, encoding	Not a table: needs flattening	Merged cells, formatting, reinterpreted values
Python tools	`csv` module, Pandas	`json` module, Pandas	`openpyxl`, Pandas

Table 1.11: Comparison of the CSV, JSON and Excel formats.

(b) Display the total and the average population.
(c) Display the names of the cities whose population exceeds 400 000 inhabitants.

7. **A tramway line.** Using a `for` loop and `range`, display *Station 1*, *Station 2*, ..., *Station 8*. A trip between two consecutive stations costs 15 DA: at each station, also display the cumulative price since the first station.
8. **Urban growth.** A city of 257 000 inhabitants grows by 3 % per year. Using a `while` loop, determine after how many years its population will have doubled, and display the population year by year.
9. **Density classification.** Write a program that asks for a population and an area in km^2, computes the density, and displays its category: *very dense* (1 000 or more), *dense* (100 to less than 1 000), *sparse* (10 to less than 100), or *very sparse* (less than 10).
10. **Building permits.** Four parcels have the areas `[150, 320, 800, 210]` (in m^2), and the third one is in a flood zone (`flood = [False, False, True, False]`). A parcel is buildable if its area is at least 200 m^2 and it is not in a flood zone. Display the status of each parcel and the number of buildable parcels.
11. **Two functions.** Write a function `to_hectares(area_m2)` and a function `density(population, area, decimals=1)` that returns the density rounded to `decimals` digits. Test them with a neighbourhood of 3 500 000 m^2 and 120 000 inhabitants.
12. **A function and a loop.** Using a `classify(density)` function and the lists of listing 1.47, count how many of the four wilayas are classified as *dense*.
13. **Saving data.** Create a list of dictionaries for Algiers, Oran, Annaba and Tamanrasset (name, population, area, density computed with a function). Save it in a JSON file and in a CSV file (with the semicolon separator), then read the JSON file back and display the name of the densest wilaya.

Solutions to the Exercises

Set 1 (Variables and Data Types)

Code 1.58: Solutions of Exercises 1 to 4

```
# Exercise 1
width = 25
length = 40
area = width * length                  # 1000
perimeter = 2 * (width + length)       # 130
price = area * 45000                   # 45,000,000 DA
print(area, perimeter, price)

# Exercise 2
plot_area = 600
footprint = 240
ratio = footprint / plot_area          # 0.4
garden = plot_area - footprint         # 360
print(ratio, garden)

# Exercise 3
area_m2 = 3.5e6
print(area_m2 / 10000, "ha")           # 350.0 ha
print(area_m2 / 1e6, "km2")            # 3.5 km2

# Exercise 4
city = "Annaba"
population = 257000
```

```
print(city + " has " + str(population) + " inhabitants.")
```

Exercise 5 (Output and Input)

Code 1.59: Solution of Exercise 5

```
city = input("City name: ")
population = int(input("Population: "))
surface = float(input("Area (km2): "))
density = population / surface
print(f"The density of {city} is {density:.1f} inh/km2.")
```

Set 2 (Loops, Conditions, Functions and Files)

Code 1.60: Solutions of Exercises 6 to 8

```
# Exercise 6
cities = ["Annaba", "Algiers", "Oran", "Constantine"]
populations = [257000, 3000000, 1500000, 450000]

for i in range(len(cities)):
    print(f"{cities[i]}: {populations[i]} inhabitants")

print("Total:", sum(populations))
print("Average:", sum(populations) / len(populations))

for i in range(len(cities)):
    if populations[i] > 400000:
        print(cities[i])

# Exercise 7
for i in range(1, 9):
    price = (i - 1) * 15
    print(f"Station {i}: {price} DA")

# Exercise 8
population = 257000
target = 2 * population
year = 0
while population < target:
    population = population * 1.03
    year += 1
    print(f"Year {year}: {population:.0f} inhabitants")
print(f"The population doubled in {year} years")
```

Code 1.61: Solutions of Exercises 9 and 10

```
# Exercise 9
population = int(input("Population: "))
surface = float(input("Area in km2: "))
density = population / surface

if density >= 1000:
    category = "very dense"
elif density >= 100:
    category = "dense"
elif density >= 10:
```

```
    category = "sparse"
else:
    category = "very sparse"
print(f"Density: {density:.1f} inh/km2 ({category})")

# Exercise 10
areas = [150, 320, 800, 210]
flood = [False, False, True, False]
count = 0
for i in range(len(areas)):
    if areas[i] >= 200 and not flood[i]:
        print(f"Parcel {i + 1}: buildable")
        count += 1
    else:
        print(f"Parcel {i + 1}: not buildable")
print(count, "buildable parcels")
```

```
Population: 257000
Area in km2: 49
Density: 5244.9 inh/km2 (very dense)
Parcel 1: not buildable
Parcel 2: buildable
Parcel 3: not buildable
Parcel 4: buildable
2 buildable parcels
```

Code 1.62: Solutions of Exercises 11 and 12

```
# Exercise 11
def to_hectares(area_m2):
    return area_m2 / 10000

def density(population, area, decimals=1):
    return round(population / area, decimals)

print(to_hectares(3500000), "ha")
print(density(120000, 3500000 / 1000000), "inh/km2")
print(density(120000, 3500000 / 1000000, decimals=0), "inh/km2")

# Exercise 12
def classify(density):
    if density >= 1000:
        return "very dense"
    elif density >= 100:
        return "dense"
    elif density >= 10:
        return "sparse"
    else:
        return "very sparse"

names = ["Algiers", "Oran", "Annaba", "Tamanrasset"]
populations = [2988145, 1584607, 609499, 115043]
areas = [1190, 2121, 1439, 336839]

count_dense = 0
for i in range(len(names)):
    if classify(density(populations[i], areas[i])) == "dense":
        count_dense += 1
print(count_dense, "wilayas classified as dense")
```

```
350.0 ha
34285.7 inh/km2
34286.0 inh/km2
2 wilayas classified as dense
```

Code 1.63: Solution of Exercise 13

```python
# Exercise 13
import json
import csv

wilayas = []
for i in range(len(names)):
    wilayas.append({"wilaya": names[i], "population": populations[i],
                    "area_km2": areas[i], "density": density(
                        populations[i], areas[i])})

with open("solution.json", "w") as f:
    json.dump(wilayas, f, indent=2)

with open("solution.csv", "w", newline="") as f:
    writer = csv.DictWriter(f, fieldnames=["wilaya", "population", "
        area_km2", "density"],
                            delimiter=";")
    writer.writeheader()
    writer.writerows(wilayas)

with open("solution.json") as f:
    data = json.load(f)

densest = data[0]
for w in data:
    if w["density"] > densest["density"]:
        densest = w
print(densest["wilaya"])
```

```
Algiers
```

Chapter 2

Data Analysis with Pandas and GeoPandas

In Chapter 1 we learned the foundations of Python: variables, lists, loops, conditions, functions and files. These tools are enough to compute an indicator for a few cities, but they quickly become heavy when the data is a real table with thousands of rows, or a map. This chapter introduces two libraries that remove this difficulty: **Pandas**, for tables, and **GeoPandas**, for geographic data. They are the working tools of the smart urban planner.

How to read the code of this chapter. Every code block **runs on its own**: you can open a new empty notebook, type one block, and run it. When a block needs data created earlier (the table of communes, the points, the zones...), it starts with a part called `SETUP`, which rebuilds it. The lines under `NEW` are the only new ones: this is where you must concentrate. All the numbers used for Annaba are approximate teaching values, not official statistics.

Warm-up: the Density of Several Communes

In Chapter 1, we computed the density of one commune: 257 000 inhabitants on 49 km^2 give about 5 245 inhabitants per km^2. How can we do the same for several communes of the wilaya? With what we already know, three lists and a loop are enough:

Code 2.1: Density of Several Communes with Lists and a Loop

```
names = ["Annaba", "El Bouni", "Sidi Amar"]
populations = [257000, 130000, 90000]
areas = [49, 60, 70]

for i in range(len(names)):
    density = populations[i] / areas[i]
    print(f"{names[i]}: {density:.0f} inh/km2")
```

```
Annaba: 5245 inh/km2
El Bouni: 2167 inh/km2
Sidi Amar: 1286 inh/km2
```

It works, but the names, the populations and the areas are three separate lists, linked only by their position. Adding a column, sorting the communes or keeping only the dense ones would require more and more code. The same result with Pandas is shorter and keeps everything in one table:

Code 2.2: Density of Several Communes with Pandas

```
import pandas as pd

data = {
    "commune": ["Annaba", "El Bouni", "Sidi Amar"],
    "population": [257000, 130000, 90000],
    "area_km2": [49, 60, 70]
}

df = pd.DataFrame(data)
df["density"] = df["population"] / df["area_km2"]
print(df)
```

```
     commune  population  area_km2      density
0     Annaba      257000        49  5244.897959
1   El Bouni      130000        60  2166.666667
2  Sidi Amar       90000        70  1285.714286
```

There is no loop: one single line computes the density of *every* commune. This is exactly why we learn Pandas.

What is a Python Library?

Python itself knows how to do basic things: calculate, store lists, repeat instructions. It does not know, by itself, how to read an Excel file, analyse a table or draw a map. A **library** (or *package*) is a collection of ready-made code, written by other people, that adds such abilities. Instead of writing everything ourselves, we **import** the tools we need.

We have already used libraries in Chapter 1: `csv`, `json` and `os` are part of the **standard library**, installed with Python. Other libraries, such as `pandas`, are **third-party libraries**: they are written and maintained by the community, and they are already included in the Anaconda distribution.

Code 2.3: Three Ways to Import a Library

```
import math                  # the whole library
from math import sqrt        # only one tool of the library
import pandas as pd          # with a nickname

print(math.sqrt(16))
print(sqrt(16))
```

```
4.0
4.0
```

The nickname `pd` is a worldwide convention: everybody writes `import pandas as pd`, so everybody can read everybody's code. The libraries used in this chapter are listed in table 2.1.

Pandas: Definition

Pandas is an open-source Python library for **manipulating and analysing tabular data**: tables with rows and columns, like an Excel sheet or a CSV file.

- **Name:** it comes from *panel data*, a term used in econometrics for tables that follow individuals over time. It has nothing to do with the animal.

Library	Usual Nickname	Role in this Chapter
`pandas`	`pd`	Tables of data (DataFrames)
`geopandas`	`gpd`	Tables of data with geometries: points, lines, polygons
`shapely`		Geometric objects, used by GeoPandas (rectangles, points)
`matplotlib`	`plt`	Simple maps and charts
`folium`		Interactive web maps, used behind the scenes by `explore()`
`osmnx`	`ox`	Downloading OpenStreetMap data

Table 2.1: The libraries used in this chapter.

- **Author:** Wes McKinney, who started the project in 2008 while working in finance, and made it open source in 2009. He is also the author of the book *Python for Data Analysis*, recommended for this course.
- **Current version:** Pandas 3.0, released in early 2026 (table 2.2).

Version	Release Date	Remark
Start of the project	2008	Created by Wes McKinney
Open source	2009	Public release of the code
Pandas 1.0	January 2020	First major stable version
Pandas 2.0	April 2023	
Pandas 3.0	January 2026	Current major version (3.0.x); text columns use a dedicated text type

Table 2.2: Main milestones in the history of Pandas.

The version installed on your computer is obtained with:

Code 2.4: Checking the Version of Pandas

```
import pandas as pd
print(pd.__version__)
```

```
3.0.2
```

Important: many tutorials on the internet are old (Pandas 0.x or 1.x). Most of their code still works, but when something fails, check the version first, and consult the official website https://pandas.pydata.org.

The Two Key Objects

Pandas is built around two objects (table 2.3). In practice, we almost always work with a `DataFrame`, and select one of its columns when needed.

Object	What it Is	Excel Equivalent
`Series`	One column of values, with an index	A column
`DataFrame`	A whole table: several columns sharing the same rows	A sheet

Table 2.3: The two main objects of Pandas.

Where is Pandas Used?

- **Public administration:** census tables, budgets, statistics offices.
- **Finance and banking:** prices, transactions, risk analysis.
- **Science and research:** experimental results, climate data, epidemiology.
- **Companies:** sales analysis, customer data, industrial and telecommunications data, e-commerce.
- **Machine learning and artificial intelligence:** cleaning and preparing data before training a model.
- **Urban planning and smart cities:**
  - population and housing statistics by district;
  - traffic counts and public transport data;
  - energy and water consumption per building or neighbourhood;
  - sensor data (air quality, noise, temperature).

Why Pandas?

Urban planners already know spreadsheets. Table 2.4 shows why Pandas becomes necessary when data grows or when an analysis must be repeated.

	Excel	Pandas
Large files	Slow, limited to about one million rows	Handles millions of rows
Repeating the analysis	Redo the clicks by hand	Run the script again
History of operations	Hidden in the cells	Written in the code
Combining with maps	Very limited	GeoPandas
Cost	Paid licence	Free

Table 2.4: Excel compared with Pandas.

- **Reproducible:** a colleague can run your code and obtain exactly the same result.

- **Fast:** one line replaces hundreds of cell formulas.
- **Standard:** it is the basis of the whole Python data ecosystem (GeoPandas, Matplotlib, machine learning libraries).
- **Free and well documented.**

Tips for Learning Pandas

1. **Type every line yourself.** Reading is not learning.
2. **Always look at your data:** use `df.head()` after each important step.
3. **Start small:** test on a table of five rows before using a file of 50 000 rows.
4. **Read the error message,** starting from the last line. It often tells you exactly what is wrong, for example a typing mistake in a column name.
5. **Use** `help()`: `help(df.sort_values)` explains how a function works.
6. **Use your own data:** districts of Annaba, bus lines, schools. You learn faster with things you know.
7. **One idea at a time:** select, filter, create a column, group, sort.
8. **Do not memorise:** keep a cheat sheet (see the end of this chapter) and use the official documentation.
9. **Work with a neighbour:** explaining code to someone is the best test of understanding.

First Contact with Pandas

We use a small table of five communes of Annaba (approximate teaching values).

Creating a Table

A DataFrame is created from a dictionary: each key is a column name, and each value is the list of the values of that column (see the dictionaries of Chapter 1, listing 1.54).

Code 2.5: Creating a DataFrame

```
import pandas as pd

data = {
    "commune": ["Annaba", "El Bouni", "Sidi Amar", "El Hadjar", "
       Berrahal"],
    "population": [257000, 130000, 90000, 45000, 40000],
    "area_km2": [49, 60, 70, 55, 100]
}

df = pd.DataFrame(data)
print(df)
```

```
     commune  population  area_km2
0     Annaba      257000        49
1   El Bouni      130000        60
2  Sidi Amar       90000        70
3  El Hadjar       45000        55
4   Berrahal       40000       100
```

The number on the left (0 to 4) is the **index**: the label of each row.

Looking at the Table

Code 2.6: Exploring a DataFrame

```python
# ---- SETUP (repeated so that this block runs alone) ----
import pandas as pd

df = pd.DataFrame({
    "commune": ["Annaba", "El Bouni", "Sidi Amar", "El Hadjar", "
       Berrahal"],
    "population": [257000, 130000, 90000, 45000, 40000],
    "area_km2": [49, 60, 70, 55, 100]
})

# ---- NEW ----
print(df.head())        # first rows
print(df.shape)         # (rows, columns)
print(df.columns)       # column names
print(df.describe())    # quick statistics
```

```
     commune  population  area_km2
0     Annaba      257000        49
1   El Bouni      130000        60
2  Sidi Amar       90000        70
3  El Hadjar       45000        55
4   Berrahal       40000       100
(5, 3)
Index(['commune', 'population', 'area_km2'], dtype='str')
          population     area_km2
count        5.000000     5.000000
mean    112400.000000    66.800000
std      88748.521114    20.092287
min      40000.000000    49.000000
25%      45000.000000    55.000000
50%      90000.000000    60.000000
75%     130000.000000    70.000000
max     257000.000000   100.000000
```

- `df.head()` displays the first rows (five by default);
- `df.shape` gives the number of rows and columns;
- `df.columns` gives the column names;
- `df.describe()` gives quick statistics (mean, minimum, maximum…) for the numerical columns.

Selecting a Column

A column is selected by writing its name between square brackets and quotation marks.

Code 2.7: Selecting a Column

```python
# ---- SETUP (repeated so that this block runs alone) ----
import pandas as pd

df = pd.DataFrame({
    "commune": ["Annaba", "El Bouni", "Sidi Amar", "El Hadjar", "
       Berrahal"],
    "population": [257000, 130000, 90000, 45000, 40000],
    "area_km2": [49, 60, 70, 55, 100]
})
```

```
9
10 # ---- NEW ----
11 print(df["population"])
```

```
0    257000
1    130000
2     90000
3     45000
4     40000
Name: population, dtype: int64
```

Computing with Columns

Operations apply to the whole column at once, row by row. No loop is needed: this is the main strength of Pandas.

Code 2.8: Creating a New Column

```
1  # ---- SETUP (repeated so that this block runs alone) ----
2  import pandas as pd
3
4  df = pd.DataFrame({
5      "commune": ["Annaba", "El Bouni", "Sidi Amar", "El Hadjar", "
           Berrahal"],
6      "population": [257000, 130000, 90000, 45000, 40000],
7      "area_km2": [49, 60, 70, 55, 100]
8  })
9
10 # ---- NEW ----
11 df["density"] = df["population"] / df["area_km2"]
12 print(df)
```

```
     commune  population  area_km2      density
0     Annaba      257000        49  5244.897959
1   El Bouni      130000        60  2166.666667
2  Sidi Amar       90000        70  1285.714286
3  El Hadjar       45000        55   818.181818
4   Berrahal       40000       100   400.000000
```

Simple Statistics

Code 2.9: Sum, Mean and Maximum of a Column

```
1  # ---- SETUP (repeated so that this block runs alone) ----
2  import pandas as pd
3
4  df = pd.DataFrame({
5      "commune": ["Annaba", "El Bouni", "Sidi Amar", "El Hadjar", "
           Berrahal"],
6      "population": [257000, 130000, 90000, 45000, 40000],
7      "area_km2": [49, 60, 70, 55, 100]
8  })
9  df["density"] = df["population"] / df["area_km2"]
10
11 # ---- NEW ----
12 print(df["population"].sum())     # total population
13 print(df["population"].mean())    # average
```

```
print(df["density"].max())        # highest density
```

```
562000
112400.0
5244.897959183673
```

Filtering Rows

A condition applied to a column gives a series of `True` and `False` values (one per row), and the table keeps only the rows where the answer is `True`. It reads: *give me the rows of `df` where the density is greater than 1000.*

Code 2.10: Filtering Rows

```
# ---- SETUP (repeated so that this block runs alone) ----
import pandas as pd

df = pd.DataFrame({
    "commune": ["Annaba", "El Bouni", "Sidi Amar", "El Hadjar", "
       Berrahal"],
    "population": [257000, 130000, 90000, 45000, 40000],
    "area_km2": [49, 60, 70, 55, 100]
})
df["density"] = df["population"] / df["area_km2"]

# ---- NEW ----
dense = df[df["density"] > 1000]
print(dense)
```

```
     commune  population  area_km2      density
0     Annaba      257000        49  5244.897959
1   El Bouni      130000        60  2166.666667
2  Sidi Amar       90000        70  1285.714286
```

The comparison operators are those of table 1.9.

Sorting

Code 2.11: Sorting a Table

```
# ---- SETUP (repeated so that this block runs alone) ----
import pandas as pd

df = pd.DataFrame({
    "commune": ["Annaba", "El Bouni", "Sidi Amar", "El Hadjar", "
       Berrahal"],
    "population": [257000, 130000, 90000, 45000, 40000],
    "area_km2": [49, 60, 70, 55, 100]
})
df["density"] = df["population"] / df["area_km2"]

# ---- NEW ----
df = df.sort_values("density", ascending=False)
print(df)
```

```
     commune  population  area_km2      density
0     Annaba      257000        49  5244.897959
1   El Bouni      130000        60  2166.666667
2  Sidi Amar       90000        70  1285.714286
3  El Hadjar       45000        55   818.181818
4   Berrahal       40000       100   400.000000
```

Reading and Saving a CSV File

In Chapter 1, reading a CSV file required the `csv` module, a `with` block, a loop, and a conversion of every value from text to number (listing 1.53). With Pandas, **one line** reads the whole file, and the numbers are recognised automatically.

Code 2.12: Reading and Saving a CSV File

```
# ---- SETUP (repeated so that this block runs alone) ----
import pandas as pd

df = pd.DataFrame({
    "commune": ["Annaba", "El Bouni", "Sidi Amar", "El Hadjar", "
       Berrahal"],
    "population": [257000, 130000, 90000, 45000, 40000],
    "area_km2": [49, 60, 70, 55, 100]
})
df.to_csv("communes.csv", index=False)       # create the file first

# ---- NEW ----
df = pd.read_csv("communes.csv")
print(df)

df.to_csv("result.csv", index=False)
```

```
     commune  population  area_km2
0     Annaba      257000        49
1   El Bouni      130000        60
2  Sidi Amar       90000        70
3  El Hadjar       45000        55
4   Berrahal       40000       100
```

The first part of the block creates the file `communes.csv`, so that the example runs without any preparation. The same logic applies to the other formats of Chapter 1: `pd.read_excel()` and `df.to_excel()` for Excel files (listing 1.56), and `pd.read_json()` for JSON.

Important: the file must be in the same folder as the notebook, or you must give its full path. The most frequent error at the beginning is `FileNotFoundError`: it simply means that Python cannot find the file.

First Contact with GeoPandas

GeoPandas is Pandas with geography. A `GeoDataFrame` is a normal DataFrame **with one special column named** `geometry`, which stores shapes. Everything learned about DataFrames (selecting, filtering, sorting, grouping) still works. Three kinds of shapes exist:

- **Point:** a school, a bus stop, a hospital;

- **Line:** a road, a tramway line;
- **Polygon:** a commune, a district, a building, a park.

Creating Points from Coordinates

Coordinates are given as a **longitude** (x) and a **latitude** (y), in degrees. Annaba is located around $x = 7.76$ and $y = 36.90$. We build a table of five places and convert it into a GeoDataFrame.

Code 2.13: Creating a GeoDataFrame of Points

```
import pandas as pd
import geopandas as gpd

places = pd.DataFrame({
    "name": ["Train station", "Ibn Rochd hospital", "Saint Augustine
        basilica",
             "Badji Mokhtar university", "Rabah Bitat airport"],
    "x": [7.756, 7.740, 7.7536, 7.7137, 7.8092],
    "y": [36.899, 36.886, 36.8828, 36.8137, 36.8222]
})

points = gpd.GeoDataFrame(
    places,
    geometry=gpd.points_from_xy(places["x"], places["y"]),
    crs="EPSG:4326"
)

print(points)
```

```
                      name       x        y                    geometry
0             Train station  7.7560  36.8990     POINT (7.756 36.899)
1       Ibn Rochd hospital  7.7400  36.8860      POINT (7.74 36.886)
2  Saint Augustine basilica  7.7536  36.8828  POINT (7.7536 36.8828)
3  Badji Mokhtar university  7.7137  36.8137  POINT (7.7137 36.8137)
4       Rabah Bitat airport  7.8092  36.8222  POINT (7.8092 36.8222)
```

Careful: these coordinates are approximate, for teaching only. Always check them on a map before using them in a real study.

Seeing the Points on a Real Map

For the moment, our points are only numbers. With one word, `.explore()`, GeoPandas draws them on a **real interactive map**: you can zoom, move, and click on a point to see its information. In Jupyter, the map appears directly under the cell. The background map (the *tiles*) is downloaded from the internet, so a connection is needed.

Code 2.14: Drawing Points on a Real Map

```
# ---- SETUP (repeated so that this block runs alone) ----
import pandas as pd
import geopandas as gpd

places = pd.DataFrame({
    "name": ["Train station", "Ibn Rochd hospital", "Saint Augustine
        basilica",
             "Badji Mokhtar university", "Rabah Bitat airport"],
    "x": [7.756, 7.740, 7.7536, 7.7137, 7.8092],
```

```
    "y": [36.899, 36.886, 36.8828, 36.8137, 36.8222]
})

points = gpd.GeoDataFrame(
    places,
    geometry=gpd.points_from_xy(places["x"], places["y"]),
    crs="EPSG:4326"
)

# ---- NEW ----
bg = "Esri.WorldStreetMap"       # the background map we will always use
points.explore(tiles=bg)
```

Why do we choose the background ourselves? The default background of `explore()` comes from the free OpenStreetMap tile servers. These servers are run by volunteers and have a strict usage policy: heavy use is forbidden, and a whole classroom sharing the same internet connection can be blocked (the message *Access blocked – tile usage policy* appears). This is why we always write `tiles=bg`, with a background defined once at the top of the code.

The look of the map can be changed:

Code 2.15: Changing the Style of the Points

```
# ---- SETUP (repeated so that this block runs alone) ----
import pandas as pd
import geopandas as gpd

places = pd.DataFrame({
    "name": ["Train station", "Ibn Rochd hospital", "Saint Augustine basilica",
             "Badji Mokhtar university", "Rabah Bitat airport"],
    "x": [7.756, 7.740, 7.7536, 7.7137, 7.8092],
    "y": [36.899, 36.886, 36.8828, 36.8137, 36.8222]
})

points = gpd.GeoDataFrame(
    places,
    geometry=gpd.points_from_xy(places["x"], places["y"]),
    crs="EPSG:4326"
)
bg = "Esri.WorldStreetMap"

# ---- NEW ----
points.explore(tiles=bg, color="red", marker_kwds={"radius": 8})
```

Other backgrounds are available. The satellite view is very useful in urbanism:

Code 2.16: Other Background Maps

```
# ---- SETUP (repeated so that this block runs alone) ----
import pandas as pd
import geopandas as gpd

places = pd.DataFrame({
    "name": ["Train station", "Ibn Rochd hospital", "Saint Augustine basilica",
             "Badji Mokhtar university", "Rabah Bitat airport"],
    "x": [7.756, 7.740, 7.7536, 7.7137, 7.8092],
    "y": [36.899, 36.886, 36.8828, 36.8137, 36.8222]
})
```

```
11
12 points = gpd.GeoDataFrame(
13     places,
14     geometry=gpd.points_from_xy(places["x"], places["y"]),
15     crs="EPSG:4326"
16 )
17 bg = "Esri.WorldStreetMap"
18
19 # ---- NEW ----
20 points.explore(tiles="Esri.WorldStreetMap")    # streets (our choice)
21 points.explore(tiles="Esri.WorldImagery")      # satellite view
22 points.explore(tiles="Esri.WorldTopoMap")      # relief and topography
```

Try it. Check on the map whether the five places are really where they should be. If one of them is wrong, correct its x and y and draw the map again.

When working in a normal script instead of a notebook, the map can be saved as a web page, and then opened in a browser:

Code 2.17: Saving a Map as a Web Page

```
1  # ---- SETUP (repeated so that this block runs alone) ----
2  import pandas as pd
3  import geopandas as gpd
4
5  places = pd.DataFrame({
6      "name": ["Train station", "Ibn Rochd hospital", "Saint Augustine
           basilica",
7               "Badji Mokhtar university", "Rabah Bitat airport"],
8      "x": [7.756, 7.740, 7.7536, 7.7137, 7.8092],
9      "y": [36.899, 36.886, 36.8828, 36.8137, 36.8222]
10 })
11
12 points = gpd.GeoDataFrame(
13     places,
14     geometry=gpd.points_from_xy(places["x"], places["y"]),
15     crs="EPSG:4326"
16 )
17 bg = "Esri.WorldStreetMap"
18
19 # ---- NEW ----
20 m = points.explore(tiles=bg)
21 m.save("map.html")
```

What is a CRS?

A **CRS** (*Coordinate Reference System*) tells how the coordinates are related to the Earth. Two of them are essential (table 2.5).

Code 2.18: Displaying the CRS of a GeoDataFrame

```
1  # ---- SETUP (repeated so that this block runs alone) ----
2  import pandas as pd
3  import geopandas as gpd
4
5  places = pd.DataFrame({
6      "name": ["Train station", "Ibn Rochd hospital", "Saint Augustine
           basilica",
7               "Badji Mokhtar university", "Rabah Bitat airport"],
8      "x": [7.756, 7.740, 7.7536, 7.7137, 7.8092],
```

CRS	Unit	Use
EPSG:4326 (WGS 84)	Degrees (longitude, latitude)	What GPS and web maps use. Good to *locate*, bad to *measure*.
EPSG:32632 (UTM zone 32N)	Metres	Annaba (7.7°E) is in this zone. Good to *measure* areas and distances.

Table 2.5: The two coordinate systems used in this chapter.

```
    "y": [36.899, 36.886, 36.8828, 36.8137, 36.8222]
})

points = gpd.GeoDataFrame(
    places,
    geometry=gpd.points_from_xy(places["x"], places["y"]),
    crs="EPSG:4326"
)

# ---- NEW ----
print(points.crs)
```

```
EPSG:4326
```

A Quick Static Map

To obtain a simple drawing without any background, and without internet, use `.plot()`. In a script (outside Jupyter), `import matplotlib.pyplot as plt` and `plt.show()` are needed to display it.

Code 2.19: A Simple Map without Background

```
# ---- SETUP (repeated so that this block runs alone) ----
import pandas as pd
import geopandas as gpd

places = pd.DataFrame({
    "name": ["Train station", "Ibn Rochd hospital", "Saint Augustine basilica",
             "Badji Mokhtar university", "Rabah Bitat airport"],
    "x": [7.756, 7.740, 7.7536, 7.7137, 7.8092],
    "y": [36.899, 36.886, 36.8828, 36.8137, 36.8222]
})

points = gpd.GeoDataFrame(
    places,
    geometry=gpd.points_from_xy(places["x"], places["y"]),
    crs="EPSG:4326"
)

# ---- NEW ----
```

```
points.plot()
```

Method	Result
`gdf.plot()`	Simple drawing, no background, works offline
`gdf.explore(tiles=bg)`	Real interactive map, needs internet

Table 2.6: The two ways of drawing a GeoDataFrame.

Shapefile and GeoJSON Formats

Geographic data is distributed mostly in two formats, which complete the three formats seen in Chapter 1 (table 1.11).

Shapefile (.shp)

The shapefile was created by the company Esri (ArcGIS) in the 1990s and is still extremely common in administrations and planning agencies. It is **not one file, but a group of files** with the same name (table 2.7).

File	Content
`communes.shp`	The shapes (geometry)
`communes.dbf`	The attribute table (names, population...)
`communes.shx`	An index of the shapes
`communes.prj`	The coordinate system (CRS)
`communes.cpg`	The text encoding (often present)

Table 2.7: The files that make a shapefile.

The following block saves a small GeoDataFrame as a shapefile and lists the files created:

Code 2.20: Files Created by a Shapefile

```
# ---- SETUP (repeated so that this block runs alone) ----
import os
import geopandas as gpd
from shapely.geometry import box

zones = gpd.GeoDataFrame(
    {"zone": ["North", "South"]},
    geometry=[box(7.70, 36.86, 7.80, 36.93),
              box(7.65, 36.78, 7.85, 36.86)],
    crs="EPSG:4326"
)
os.makedirs("demo", exist_ok=True)
```

```
13
14 # ---- NEW ----
15 zones.to_file("demo/zones.shp")
16 print(sorted(os.listdir("demo")))
```

```
['zones.cpg', 'zones.dbf', 'zones.prj', 'zones.shp', 'zones.shx']
```

Never send only the `.shp` **file:** send all the files of the group, or a zip archive. A shapefile also has limits: column names are cut at 10 characters, and the size is limited to 2 GB.

GeoJSON (.geojson)

GeoJSON is a **single text file** based on JSON, the format studied in Chapter 1 (listing 1.54), with a structure adapted to geography. It is the standard of the web (web maps, open data portals, OpenStreetMap exports), and it is always in longitude and latitude (EPSG:4326). Since it is plain text, it can be opened in any text editor:

Code 2.21: Content of a GeoJSON File

```
1  # ---- SETUP (repeated so that this block runs alone) ----
2  import geopandas as gpd
3  from shapely.geometry import Point
4
5  station = gpd.GeoDataFrame(
6      {"name": ["Train station"]},
7      geometry=[Point(7.756, 36.899)],
8      crs="EPSG:4326"
9  )
10 station.to_file("station.geojson", driver="GeoJSON")
11
12 # ---- NEW ----
13 print(open("station.geojson").read())
```

```
{
"type": "FeatureCollection",
"name": "station",
"crs": { "type": "name", "properties": { "name": "urn:ogc:def:crs:OGC
   :1.3:CRS84" } },
"features": [
{ "type": "Feature", "properties": { "name": "Train station" }, "
   geometry": { "type": "Point", "coordinates": [ 7.756, 36.899 ] } }
]
}
```

We recognise the nested structure of JSON: a collection of *features*, each one having `properties` (the table attributes) and a `geometry` (type and coordinates).

Where can geographic data of Annaba be found?

- OpenStreetMap (roads, buildings, hospitals, schools): see the next sections, or the downloads of https://download.geofabrik.de;
- GADM (administrative boundaries): https://gadm.org;
- geoBoundaries: https://www.geoboundaries.org;
- Humanitarian Data Exchange: https://data.humdata.org;
- local sources: urban planning agencies, wilaya and APC services (do not hesitate to ask for the data).

	Shapefile	**GeoJSON**
Number of files	Several	One
Readable in a text editor	No	Yes
Coordinate system	Any, stored in the `.prj` file	Always longitude and latitude
Typical use	GIS software, administrations	Web, open data

Table 2.8: Comparison of the shapefile and GeoJSON formats.

Reading Shapefile and GeoJSON Files

A single function reads both formats: `gpd.read_file()`. The format is recognised automatically. In the block below, the `SETUP` part first creates two tiny test files (two rectangles named after communes, not real boundaries), so that the example runs without any download. In real life, this part is skipped, and a downloaded file is read.

Code 2.22: Reading a Shapefile and a GeoJSON File

```
# ---- SETUP (repeated so that this block runs alone) ----
import geopandas as gpd
from shapely.geometry import box

# two tiny test files (normally you would download them)
c = gpd.GeoDataFrame(
    {"commune": ["Annaba", "El Bouni"],
     "wilaya": ["Annaba", "Annaba"],
     "population": [257000, 130000]},
    geometry=[box(7.70, 36.86, 7.80, 36.93),
              box(7.60, 36.70, 7.70, 36.85)],
    crs="EPSG:4326"
)
c.to_file("communes.geojson", driver="GeoJSON")
c.to_file("communes.shp")

# ---- NEW ----
communes = gpd.read_file("communes.shp")          # shapefile
communes = gpd.read_file("communes.geojson")      # geojson
```

The result is explored exactly like a Pandas table:

Code 2.23: Exploring a GeoDataFrame Read from a File

```
# ---- SETUP (repeated so that this block runs alone) ----
import geopandas as gpd
from shapely.geometry import box

# two tiny test files (normally you would download them)
c = gpd.GeoDataFrame(
    {"commune": ["Annaba", "El Bouni"],
     "wilaya": ["Annaba", "Annaba"],
     "population": [257000, 130000]},
    geometry=[box(7.70, 36.86, 7.80, 36.93),
```

```
                box(7.60, 36.70, 7.70, 36.85)],
    crs="EPSG:4326"
)
c.to_file("communes.geojson", driver="GeoJSON")
c.to_file("communes.shp")

communes = gpd.read_file("communes.geojson")

# ---- NEW ----
print(communes.head())
print(communes.shape)
print(communes.columns)
print(communes.crs)
communes.plot()
```

```
    commune  wilaya  population  \
0    Annaba  Annaba      257000
1  El Bouni  Annaba      130000

                                            geometry
0  POLYGON ((7.8 36.86, 7.8 36.93, 7.7 36.93, 7.7...
1  POLYGON ((7.7 36.7, 7.7 36.85, 7.6 36.85, 7.6 ...
(2, 4)
Index(['commune', 'wilaya', 'population', 'geometry'], dtype='str')
EPSG:4326
```

A GeoDataFrame is saved with `to_file()`:

Code 2.24: Saving a GeoDataFrame

```
# ---- SETUP (repeated so that this block runs alone) ----
import geopandas as gpd
from shapely.geometry import box

# two tiny test files (normally you would download them)
c = gpd.GeoDataFrame(
    {"commune": ["Annaba", "El Bouni"],
     "wilaya": ["Annaba", "Annaba"],
     "population": [257000, 130000]},
    geometry=[box(7.70, 36.86, 7.80, 36.93),
              box(7.60, 36.70, 7.70, 36.85)],
    crs="EPSG:4326"
)
c.to_file("communes.geojson", driver="GeoJSON")
c.to_file("communes.shp")

communes = gpd.read_file("communes.geojson")

# ---- NEW ----
communes.to_file("result.geojson", driver="GeoJSON")
communes.to_file("result.shp")
```

It is filtered like a normal table. If the file has a column named `wilaya`:

Code 2.25: Filtering a GeoDataFrame

```
# ---- SETUP (repeated so that this block runs alone) ----
import geopandas as gpd
from shapely.geometry import box

# two tiny test files (normally you would download them)
```

```
c = gpd.GeoDataFrame(
    {"commune": ["Annaba", "El Bouni"],
     "wilaya": ["Annaba", "Annaba"],
     "population": [257000, 130000]},
    geometry=[box(7.70, 36.86, 7.80, 36.93),
              box(7.60, 36.70, 7.70, 36.85)],
    crs="EPSG:4326"
)
c.to_file("communes.geojson", driver="GeoJSON")
c.to_file("communes.shp")

communes = gpd.read_file("communes.geojson")

# ---- NEW ----
annaba = communes[communes["wilaya"] == "Annaba"]
annaba.plot()
```

Important: the real column names depend on the file. Always check them with `print(communes.columns)`.

Try it. Download the administrative boundaries of Algeria (level of the communes) from one of the sources above, read the file, and display the number of rows and the column names. How many communes does the file contain? Which column contains the name of the commune?

Real Data from OpenStreetMap

OpenStreetMap (OSM) is the free map of the world, built by volunteers, which can be seen as the Wikipedia of maps. It contains roads, buildings, hospitals, schools, bus stops and parks, including for Annaba. Every object has **tags** that describe it, for instance:

- `amenity=hospital`, `amenity=school`, `amenity=pharmacy`;
- `leisure=park`;
- `highway=primary` for a main road, and `building=yes` for a building.

To download OSM data directly into a GeoDataFrame, we use a library called **OSMnx**. An internet connection is needed when the code is run.

Downloading the Hospitals of Annaba

Code 2.26: Downloading Hospitals from OpenStreetMap

```
import osmnx as ox

hospitals = ox.features_from_place("Annaba, Algeria", tags={"amenity":
    "hospital"})

print(hospitals.shape)
print(hospitals[["name", "geometry"]].head())
hospitals.plot()
```

The result is a normal GeoDataFrame: everything learned before applies (`head`, `shape`, `plot`, `to_crs`, `sjoin`...). Three points of attention:

- some objects have no `name`: the value is then `NaN` (empty);

- OSM is made by volunteers, so it can be incomplete or outdated: always check the result;
- a hospital can be a **point** or a **polygon** (a building): both can be in the same table.

The hospitals are drawn on a real map. Only the columns `name` and `geometry` are kept, because OSM tables contain many messy columns that a map cannot always handle.

Code 2.27: Drawing the Hospitals on a Real Map

```
# ---- SETUP (repeated so that this block runs alone) ----
import osmnx as ox
bg = "Esri.WorldStreetMap"
hospitals = ox.features_from_place("Annaba, Algeria", tags={"amenity":
    "hospital"})

# ---- NEW ----
hospitals[["name", "geometry"]].explore(tiles=bg, color="red")
```

Other Objects: Only the Tag Changes

Code 2.28: Downloading Schools, Parks and Pharmacies

```
# ---- SETUP (repeated so that this block runs alone) ----
import osmnx as ox

# ---- NEW ----
schools = ox.features_from_place("Annaba, Algeria", tags={"amenity": "
    school"})
parks = ox.features_from_place("Annaba, Algeria", tags={"leisure": "
    park"})
pharmacies = ox.features_from_place("Annaba, Algeria", tags={"amenity":
     "pharmacy"})

print(len(schools), "schools")
print(len(parks), "parks")
print(len(pharmacies), "pharmacies")
```

Several layers can be drawn on the same map: the first layer creates the map, and the following ones are added with the parameter `m=m`.

Code 2.29: Several Layers on the Same Map

```
# ---- SETUP (repeated so that this block runs alone) ----
import osmnx as ox
bg = "Esri.WorldStreetMap"
hospitals = ox.features_from_place("Annaba, Algeria", tags={"amenity":
    "hospital"})
schools = ox.features_from_place("Annaba, Algeria", tags={"amenity": "
    school"})

# ---- NEW ----
m = hospitals[["name", "geometry"]].explore(tiles=bg, color="red")
schools[["name", "geometry"]].explore(m=m, color="blue")
```

The Road Network

Code 2.30: Drawing the Road Network

```
# ---- SETUP (repeated so that this block runs alone) ----
import osmnx as ox

# ---- NEW ----
roads = ox.graph_from_place("Annaba, Algeria", network_type="drive")
ox.plot_graph(roads)
```

Using OSM Data in an Analysis

A classic accessibility question is: *how far is a school from the nearest hospital?* Since objects can be points or polygons, we first convert both layers to metres, and replace each shape by its centre (its *centroid*), so that they can be compared.

Code 2.31: Converting OSM Layers to Metres and Centroids

```
# ---- SETUP (repeated so that this block runs alone) ----
import osmnx as ox
hospitals = ox.features_from_place("Annaba, Algeria", tags={"amenity":
    "hospital"})
schools = ox.features_from_place("Annaba, Algeria", tags={"amenity": "
    school"})

# ---- NEW ----
schools_m = schools.to_crs("EPSG:32632")
hospitals_m = hospitals.to_crs("EPSG:32632")

schools_m["geometry"] = schools_m.geometry.centroid
hospitals_m["geometry"] = hospitals_m.geometry.centroid
```

Then, the distance from the first school to every hospital, in kilometres, and the smallest of them:

Code 2.32: Distance from a School to the Nearest Hospital

```
# ---- SETUP (repeated so that this block runs alone) ----
import osmnx as ox
hospitals = ox.features_from_place("Annaba, Algeria", tags={"amenity":
    "hospital"})
schools = ox.features_from_place("Annaba, Algeria", tags={"amenity": "
    school"})

schools_m = schools.to_crs("EPSG:32632")
hospitals_m = hospitals.to_crs("EPSG:32632")
schools_m["geometry"] = schools_m.geometry.centroid
hospitals_m["geometry"] = hospitals_m.geometry.centroid

# ---- NEW ----
school = schools_m.geometry.iloc[0]
distances = hospitals_m.geometry.distance(school) / 1000     # km

print(distances.min())
```

The total green area of the parks, in km^2:

Code 2.33: Total Area of the Parks

```python
# ---- SETUP (repeated so that this block runs alone) ----
import osmnx as ox
parks = ox.features_from_place("Annaba, Algeria", tags={"leisure": "
    park"})

# ---- NEW ----
parks_m = parks.to_crs("EPSG:32632")
parks_m["area_km2"] = parks_m.geometry.area / 1000000

print(parks_m["area_km2"].sum())
```

Saving the Data

Code 2.34: Saving OSM Data as a GeoJSON File

```python
# ---- SETUP (repeated so that this block runs alone) ----
import osmnx as ox
hospitals = ox.features_from_place("Annaba, Algeria", tags={"amenity":
    "hospital"})

# ---- NEW ----
hospitals[["name", "geometry"]].to_file("hospitals_annaba.geojson",
    driver="GeoJSON")
```

This produces your own `.geojson` file, which can be reopened at any time with `gpd.read_file()`, without internet. Keep only simple columns (such as `name` and `geometry`) before saving: OSM tables have dozens of messy columns, and some formats, especially the shapefile, do not accept them.

Other Ways to Obtain OSM Data

OSM data of the whole of Algeria can also be downloaded as files from https://download.geofabrik.de (zip archives containing shapefiles: roads, buildings, places...) and read with `gpd.read_file()`, exactly as in the previous section. These files are large, so OSMnx is easier for a single city.

Credit: when a map made with OSM data is published, the mention "© OpenStreetMap contributors" is mandatory. Note that the *data* (what OSMnx downloads) and the *background tiles* (what `explore()` displays) are two different services, with two different usage rules.

Spatial Joins, Areas and Distances

For this section we build two simple zones by hand, so that everyone can test the code without any download. Later, they can be replaced by real files.

Creating Two Zones (Polygons)

The function `box()` of the `shapely` library draws a rectangle from its limits $(x_{min}, y_{min}, x_{max}, y_{max})$.

Code 2.35: Creating Two Polygons

```python
# ---- SETUP (repeated so that this block runs alone) ----
import geopandas as gpd
```

```python

# ---- NEW ----
from shapely.geometry import box

zones = gpd.GeoDataFrame(
    {"zone": ["North", "South"]},
    geometry=[box(7.70, 36.86, 7.80, 36.93),      # (xmin, ymin, xmax, ymax)
              box(7.65, 36.78, 7.85, 36.86)],
    crs="EPSG:4326"
)

print(zones)
```

```
    zone                                           geometry
0  North  POLYGON ((7.8 36.86, 7.8 36.93, 7.7 36.93, 7.7...
1  South  POLYGON ((7.85 36.78, 7.85 36.86, 7.65 36.86, ...
```

Showing Points and Zones Together

Code 2.36: Drawing Zones and Points on a Static Map

```python
# ---- SETUP (repeated so that this block runs alone) ----
import pandas as pd
import geopandas as gpd

places = pd.DataFrame({
    "name": ["Train station", "Ibn Rochd hospital", "Saint Augustine basilica",
             "Badji Mokhtar university", "Rabah Bitat airport"],
    "x": [7.756, 7.740, 7.7536, 7.7137, 7.8092],
    "y": [36.899, 36.886, 36.8828, 36.8137, 36.8222]
})

points = gpd.GeoDataFrame(
    places,
    geometry=gpd.points_from_xy(places["x"], places["y"]),
    crs="EPSG:4326"
)
from shapely.geometry import box

zones = gpd.GeoDataFrame(
    {"zone": ["North", "South"]},
    geometry=[box(7.70, 36.86, 7.80, 36.93),
              box(7.65, 36.78, 7.85, 36.86)],
    crs="EPSG:4326"
)
import matplotlib.pyplot as plt

# ---- NEW ----
ax = zones.plot(color="lightgreen", edgecolor="black")
points.plot(ax=ax, color="red")
plt.show()
```

The same on a real map, with several layers:

Code 2.37: Drawing Zones and Points on a Real Map

```python
# ---- SETUP (repeated so that this block runs alone) ----
```

```
import pandas as pd
import geopandas as gpd

places = pd.DataFrame({
    "name": ["Train station", "Ibn Rochd hospital", "Saint Augustine basilica",
             "Badji Mokhtar university", "Rabah Bitat airport"],
    "x": [7.756, 7.740, 7.7536, 7.7137, 7.8092],
    "y": [36.899, 36.886, 36.8828, 36.8137, 36.8222]
})

points = gpd.GeoDataFrame(
    places,
    geometry=gpd.points_from_xy(places["x"], places["y"]),
    crs="EPSG:4326"
)
from shapely.geometry import box

zones = gpd.GeoDataFrame(
    {"zone": ["North", "South"]},
    geometry=[box(7.70, 36.86, 7.80, 36.93),
              box(7.65, 36.78, 7.85, 36.86)],
    crs="EPSG:4326"
)
bg = "Esri.WorldStreetMap"

# ---- NEW ----
m = zones.explore(tiles=bg, color="green")
points.explore(m=m, color="red")
```

Spatial Join: Which Point is in Which Zone?

In Pandas, a join uses a common column. In GeoPandas, a **spatial join** uses **location**: it attaches to each point the information of the polygon in which it is located. Here, each point receives the name of its zone.

Code 2.38: A Spatial Join

```
# ---- SETUP (repeated so that this block runs alone) ----
import pandas as pd
import geopandas as gpd

places = pd.DataFrame({
    "name": ["Train station", "Ibn Rochd hospital", "Saint Augustine basilica",
             "Badji Mokhtar university", "Rabah Bitat airport"],
    "x": [7.756, 7.740, 7.7536, 7.7137, 7.8092],
    "y": [36.899, 36.886, 36.8828, 36.8137, 36.8222]
})

points = gpd.GeoDataFrame(
    places,
    geometry=gpd.points_from_xy(places["x"], places["y"]),
    crs="EPSG:4326"
)
from shapely.geometry import box

zones = gpd.GeoDataFrame(
    {"zone": ["North", "South"]},
```

```
    geometry=[box(7.70, 36.86, 7.80, 36.93),
              box(7.65, 36.78, 7.85, 36.86)],
    crs="EPSG:4326"
)

# ---- NEW ----
joined = gpd.sjoin(points, zones, predicate="within")
print(joined[["name", "zone"]])
```

```
                      name   zone
0             Train station  North
1       Ibn Rochd hospital  North
2  Saint Augustine basilica  North
3  Badji Mokhtar university  South
4       Rabah Bitat airport  South
```

The result is a normal table, so the Pandas tools apply. How many places in each zone?

Code 2.39: Counting Places per Zone

```
# ---- SETUP (repeated so that this block runs alone) ----
import pandas as pd
import geopandas as gpd

places = pd.DataFrame({
    "name": ["Train station", "Ibn Rochd hospital", "Saint Augustine
        basilica",
             "Badji Mokhtar university", "Rabah Bitat airport"],
    "x": [7.756, 7.740, 7.7536, 7.7137, 7.8092],
    "y": [36.899, 36.886, 36.8828, 36.8137, 36.8222]
})

points = gpd.GeoDataFrame(
    places,
    geometry=gpd.points_from_xy(places["x"], places["y"]),
    crs="EPSG:4326"
)
from shapely.geometry import box

zones = gpd.GeoDataFrame(
    {"zone": ["North", "South"]},
    geometry=[box(7.70, 36.86, 7.80, 36.93),
              box(7.65, 36.78, 7.85, 36.86)],
    crs="EPSG:4326"
)
joined = gpd.sjoin(points, zones, predicate="within")

# ---- NEW ----
print(joined.groupby("zone")["name"].count())
```

```
zone
North    3
South    2
Name: name, dtype: int64
```

Careful: both layers must have the **same CRS**. If they do not, convert one of them with `.to_crs()` before the join.

Computing Areas

Never compute an area in degrees. The layer is first converted to metres (EPSG:32632).

Code 2.40: Area of the Zones in Square Kilometres

```
# ---- SETUP (repeated so that this block runs alone) ----
import pandas as pd
import geopandas as gpd

places = pd.DataFrame({
    "name": ["Train station", "Ibn Rochd hospital", "Saint Augustine basilica",
             "Badji Mokhtar university", "Rabah Bitat airport"],
    "x": [7.756, 7.740, 7.7536, 7.7137, 7.8092],
    "y": [36.899, 36.886, 36.8828, 36.8137, 36.8222]
})

points = gpd.GeoDataFrame(
    places,
    geometry=gpd.points_from_xy(places["x"], places["y"]),
    crs="EPSG:4326"
)
from shapely.geometry import box

zones = gpd.GeoDataFrame(
    {"zone": ["North", "South"]},
    geometry=[box(7.70, 36.86, 7.80, 36.93),
              box(7.65, 36.78, 7.85, 36.86)],
    crs="EPSG:4326"
)

# ---- NEW ----
zones_m = zones.to_crs("EPSG:32632")

zones_m["area_m2"] = zones_m.geometry.area
zones_m["area_km2"] = zones_m["area_m2"] / 1000000

print(zones_m[["zone", "area_km2"]])
```

```
    zone     area_km2
0  North    69.207647
1  South   158.341699
```

Computing Distances

Distance from every place to the train station, which is the first row of the table:

Code 2.41: Distance from Each Place to the Train Station

```
# ---- SETUP (repeated so that this block runs alone) ----
import pandas as pd
import geopandas as gpd

places = pd.DataFrame({
    "name": ["Train station", "Ibn Rochd hospital", "Saint Augustine basilica",
             "Badji Mokhtar university", "Rabah Bitat airport"],
    "x": [7.756, 7.740, 7.7536, 7.7137, 7.8092],
```

```
    "y": [36.899, 36.886, 36.8828, 36.8137, 36.8222]
})

points = gpd.GeoDataFrame(
    places,
    geometry=gpd.points_from_xy(places["x"], places["y"]),
    crs="EPSG:4326"
)

# ---- NEW ----
points_m = points.to_crs("EPSG:32632")

station = points_m.geometry.iloc[0]         # first row = train station

points_m["dist_km"] = points_m.geometry.distance(station) / 1000
print(points_m[["name", "dist_km"]])
```

```
                       name     dist_km
0             Train station    0.000000
1        Ibn Rochd hospital    2.028132
2  Saint Augustine basilica    1.810039
3  Badji Mokhtar university   10.187625
4       Rabah Bitat airport    9.751771
```

This is a straight-line distance ("as the crow flies"), not a distance by road. The rule to remember is: **degrees to locate, metres to measure**. Always write `.to_crs("EPSG:32632")` before computing any area or distance.

A Real Density with GeoPandas

If a commune file contains a population column, the density is computed with the *real* area, taken from the map itself. We are back to the warm-up exercise of Chapter 1.

Code 2.42: Population Density from the Area of the Polygons

```
# ---- SETUP (repeated so that this block runs alone) ----
import geopandas as gpd
from shapely.geometry import box

# two tiny test files (normally you would download them)
c = gpd.GeoDataFrame(
    {"commune": ["Annaba", "El Bouni"],
     "wilaya": ["Annaba", "Annaba"],
     "population": [257000, 130000]},
    geometry=[box(7.70, 36.86, 7.80, 36.93),
              box(7.60, 36.70, 7.70, 36.85)],
    crs="EPSG:4326"
)
c.to_file("communes.geojson", driver="GeoJSON")
c.to_file("communes.shp")

communes = gpd.read_file("communes.geojson")

# ---- NEW ----
communes_m = communes.to_crs("EPSG:32632")
communes_m["area_km2"] = communes_m.geometry.area / 1000000
communes_m["density"] = communes_m["population"] / communes_m["area_km2
    "]
```

Exercises (Set 3)

Exercises 14 and 15 concern Pandas, exercises 16 and 17 concern GeoPandas.

14. **Density and ranking (Pandas).** Use the following table (approximate teaching values):

Code 2.43: Data of Exercise 14

```
import pandas as pd

df = pd.DataFrame({
    "commune": ["Annaba", "El Bouni", "Sidi Amar", "El Hadjar", "
       Berrahal"],
    "population": [257000, 130000, 90000, 45000, 40000],
    "area_km2": [49, 60, 70, 55, 100]
})
```

(a) Add a column `density`.
(b) What is the total population of the five communes?
(c) Keep only the communes with a density above 1 000 inhabitants per km^2.
(d) Sort the communes from the densest to the least dense. Which one is the densest?

15. **Green space per inhabitant (Pandas).** The World Health Organization is often quoted as recommending **at least 9 m^2 of green space per inhabitant**. The districts and values below are invented for the exercise.

Code 2.44: Data of Exercise 15

```
import pandas as pd

districts = pd.DataFrame({
    "district": ["A", "B", "C", "D", "E"],
    "population": [60000, 45000, 30000, 80000, 25000],
    "green_m2": [250000, 600000, 90000, 500000, 400000]
})
```

(a) Create a column `green_per_person` (in m^2 per inhabitant).
(b) Create a column `ok` that is `True` if the value is at least 9.
(c) Display only the districts **below** the recommendation.
(d) Bonus: how many m^2 of green space must be added in the worst district to reach 9 m^2 per inhabitant?

16. **Access to facilities (GeoPandas).** Start from the two layers of the chapter:

Code 2.45: Data of Exercise 16

```
import pandas as pd
import geopandas as gpd
from shapely.geometry import box

places = pd.DataFrame({
    "name": ["Train station", "Ibn Rochd hospital", "Saint
       Augustine basilica",
             "Badji Mokhtar university", "Rabah Bitat airport"],
    "x": [7.756, 7.740, 7.7536, 7.7137, 7.8092],
    "y": [36.899, 36.886, 36.8828, 36.8137, 36.8222]
})

```

```python
points = gpd.GeoDataFrame(
    places,
    geometry=gpd.points_from_xy(places["x"], places["y"]),
    crs="EPSG:4326"
)

zones = gpd.GeoDataFrame(
    {"zone": ["North", "South"]},
    geometry=[box(7.70, 36.86, 7.80, 36.93),
              box(7.65, 36.78, 7.85, 36.86)],
    crs="EPSG:4326"
)
```

(a) Convert both layers to EPSG:32632.
(b) Compute the area of each zone in km^2.
(c) Do a spatial join: in which zone is each place?
(d) Compute the distance, in kilometres, from each place to the **Saint Augustine basilica**.
(e) Which place is the farthest from the basilica?
(f) Draw the zones and the places on a real map with `.explore(tiles=bg)` (first write `bg = "Esri.WorldStreetMap"`).

17. **Services of Annaba with OpenStreetMap.**

(a) Download the **pharmacies** and the **parks** of Annaba with OSMnx.
(b) How many pharmacies and how many parks were obtained?
(c) Compute the total area of the parks in km^2.
(d) Convert the pharmacies to metres, replace their geometry by the centroid, and compute the distance, in kilometres, from each pharmacy to the nearest park.
(e) Save the pharmacies (columns `name` and `geometry`) in a `.geojson` file.
(f) Draw the pharmacies (in red) and the parks (in green) together on a real map.

Optional challenge: with the two zones of the chapter (or a real commune file), count with a spatial join how many hospitals or schools are located in each zone.

Solutions to the Exercises (Set 3)

Exercise 14

Code 2.46: Solution of Exercise 14

```python
# ---- SETUP (repeated so that this block runs alone) ----
import pandas as pd

df = pd.DataFrame({
    "commune": ["Annaba", "El Bouni", "Sidi Amar", "El Hadjar", "
       Berrahal"],
    "population": [257000, 130000, 90000, 45000, 40000],
    "area_km2": [49, 60, 70, 55, 100]
})

# ---- NEW ----
df["density"] = df["population"] / df["area_km2"]            # 1

print(df["population"].sum())                                # 2 ->
    562000
```

```python

print(df[df["density"] > 1000])                                    # 3

df = df.sort_values("density", ascending=False)                    # 4
print(df)     # Annaba is the densest (about 5245 inhab/km2)
```

```
562000
     commune  population  area_km2      density
0     Annaba      257000        49  5244.897959
1   El Bouni      130000        60  2166.666667
2  Sidi Amar       90000        70  1285.714286
     commune  population  area_km2      density
0     Annaba      257000        49  5244.897959
1   El Bouni      130000        60  2166.666667
2  Sidi Amar       90000        70  1285.714286
3  El Hadjar       45000        55   818.181818
4   Berrahal       40000       100   400.000000
```

Exercise 15

Code 2.47: Solution of Exercise 15

```python
# ---- SETUP (repeated so that this block runs alone) ----
import pandas as pd

districts = pd.DataFrame({
    "district": ["A", "B", "C", "D", "E"],
    "population": [60000, 45000, 30000, 80000, 25000],
    "green_m2": [250000, 600000, 90000, 500000, 400000]
})

# ---- NEW ----
districts["green_per_person"] = districts["green_m2"] / districts["
    population"]    # 1
districts["ok"] = districts["green_per_person"] >= 9
                                        # 2
print(districts[districts["ok"] == False])
                                                # 3

# Bonus: worst district
worst = districts.sort_values("green_per_person").iloc[0]
missing = 9 * worst["population"] - worst["green_m2"]
print(worst["district"], missing)
```

```
  district  population  green_m2  green_per_person     ok
0        A       60000    250000          4.166667  False
2        C       30000     90000          3.000000  False
3        D       80000    500000          6.250000  False
C 180000
```

Exercise 16

Code 2.48: Solution of Exercise 16

```python
# ---- SETUP (repeated so that this block runs alone) ----
import pandas as pd
```

```python
import geopandas as gpd

places = pd.DataFrame({
    "name": ["Train station", "Ibn Rochd hospital", "Saint Augustine basilica",
             "Badji Mokhtar university", "Rabah Bitat airport"],
    "x": [7.756, 7.740, 7.7536, 7.7137, 7.8092],
    "y": [36.899, 36.886, 36.8828, 36.8137, 36.8222]
})

points = gpd.GeoDataFrame(
    places,
    geometry=gpd.points_from_xy(places["x"], places["y"]),
    crs="EPSG:4326"
)
from shapely.geometry import box

zones = gpd.GeoDataFrame(
    {"zone": ["North", "South"]},
    geometry=[box(7.70, 36.86, 7.80, 36.93),
              box(7.65, 36.78, 7.85, 36.86)],
    crs="EPSG:4326"
)

# ---- NEW ----
points_m = points.to_crs("EPSG:32632")                          # 1
zones_m = zones.to_crs("EPSG:32632")

zones_m["area_km2"] = zones_m.geometry.area / 1000000           # 2
print(zones_m[["zone", "area_km2"]])

joined = gpd.sjoin(points_m, zones_m, predicate="within")       # 3
print(joined[["name", "zone"]])

basilica = points_m[points_m["name"] == "Saint Augustine basilica"].geometry.iloc[0]
points_m["dist_km"] = points_m.geometry.distance(basilica) / 1000   # 4
print(points_m[["name", "dist_km"]])

print(points_m.sort_values("dist_km", ascending=False).iloc[0]["name"])       # 5

bg = "Esri.WorldStreetMap"                                      # 6
m = zones.explore(tiles=bg, color="green")
points.explore(m=m, color="red")
```

```
    zone    area_km2
0  North   69.207647
1  South  158.341699
                    name   zone
0            Train station  North
1       Ibn Rochd hospital  North
2  Saint Augustine basilica  North
3  Badji Mokhtar university  South
4      Rabah Bitat airport  South
                    name   dist_km
0            Train station  1.810039
1       Ibn Rochd hospital  1.263017
2  Saint Augustine basilica  0.000000
3  Badji Mokhtar university  8.451788
```

```
4        Rabah Bitat airport  8.353368
Badji Mokhtar university
```

Exercise 17

Code 2.49: Solution of Exercise 17

```python
# ---- SETUP (repeated so that this block runs alone) ----
import osmnx as ox
bg = "Esri.WorldStreetMap"

# ---- NEW ----
pharmacies = ox.features_from_place("Annaba, Algeria", tags={"amenity": "pharmacy"})    # 1
parks = ox.features_from_place("Annaba, Algeria", tags={"leisure": "park"})

print(len(pharmacies), len(parks))                                                      # 2

parks_m = parks.to_crs("EPSG:32632")                                                    # 3
print((parks_m.geometry.area / 1000000).sum())

pharm_m = pharmacies.to_crs("EPSG:32632")                                               # 4
pharm_m["geometry"] = pharm_m.geometry.centroid
parks_m["geometry"] = parks_m.geometry.centroid

distances = []
for point in pharm_m.geometry:
    distances.append(parks_m.geometry.distance(point).min() / 1000)

pharm_m["dist_park_km"] = distances
print(pharm_m["dist_park_km"].describe())

pharmacies[["name", "geometry"]].to_file("pharmacies_annaba.geojson", driver="GeoJSON")    # 5

m = parks[["name", "geometry"]].explore(tiles=bg, color="green")                        # 6
pharmacies[["name", "geometry"]].explore(m=m, color="red")
```

Quick Reference

Pandas	**Code**
Import	`import pandas as pd`
Read a CSV file	`df = pd.read_csv("file.csv")`
First rows	`df.head()`
Size	`df.shape`
Column names	`df.columns`
Statistics	`df.describe()`
Select a column	`df["col"]`
New column	`df["new"] = df["a"] / df["b"]`
Filter rows	`df[df["col"] > 10]`
Sort	`df.sort_values("col")`
Sum, mean, maximum	`df["col"].sum()`
Group	`df.groupby("col")["x"].sum()`
Save a CSV file	`df.to_csv("out.csv", index=False)`

Table 2.9: Quick reference: Pandas.

GeoPandas and OSMnx	**Code**
Import	`import geopandas as gpd`
Read a `.shp` or `.geojson` file	`gpd.read_file("file")`
Points from x and y	`gpd.points_from_xy(x, y)`
Show the CRS	`gdf.crs`
Change the CRS	`gdf.to_crs("EPSG:32632")`
Area	`gdf.geometry.area`
Distance	`gdf.geometry.distance(other)`
Centre of shapes	`gdf.geometry.centroid`
Spatial join	`gpd.sjoin(a, b, predicate="within")`
Simple map	`gdf.plot()`
Real interactive map	`gdf.explore(tiles=bg)`
Add a layer to a map	`other.explore(m=m, color="red")`
Save a map	`m.save("map.html")`
Save a layer	`gdf.to_file("out.geojson", driver="GeoJSON")`
Download OSM objects	`ox.features_from_place("Annaba, Algeria", tags={"amenity": "school"})`

Table 2.10: Quick reference: GeoPandas and OSMnx.