Introduction to Data Analysis with Python

Cecilia Graiff
ALMAnaCH, Inria Paris

Acknowledgements

This handbook is a work in progress created for the lectures in the course "Advanced Data Analysis with Python", given at SciencesPo in fall 2025. It is not a final or distributed version and it is susceptible to changes.

To write this handbook, I relied on several sources, which I will list below. This list is also being updated. Please note that the version you are using might not contain all references. In addition to this list, I leveraged the **official online documentation** of the Python packages that are handled.

- LxMLS lab guide
- Python Data Science Handbook
- An Introduction to Statistical Learning
- Introduction to Applied Linear Algebra
- Linear Algebra with Python

Contents

1 Getting started with Python

1.1 Installation

To begin with the courses, it is necessary that you install the latest version of Python. You can follow the instructions on the Python official documentation page.

1.2 Setup

1.2.1 Virtual Environments

To use Python for specific purposes, you will need to install specific versions of specific packages. For this reason, it is more practical to use dedicated **virtual environments**. A virtual environment is a directory tree that contains a specific version of Python, and specific versions of the installed packages.

You can create a virtual environment by using the `venv` package:

```
python -m venv myenv
```

This will create the directory. Now you need to activate your venv:

On Linux and MacOS:

```
source myenv/bin/activate
```

On Windows:

```
myenv/Scripts/activate
```

To deactivate the package, type:

```
deactivate
```

1.2.2 Installing packages

You can now proceed to install the packages you need. To install packages, you use `pip`, a program that comes with the Python installation and installs packages from the Python Package Index (PiPI) by default. To install a package with `pip`, run

```
python -m pip install your-package
```

If you want to install a specific version of that package (for example, in case you need an older one), type:

```
python -m pip install your-package = = version-number
```

In case you want the latest version of the package, you just need to run:

```
python -m pip install --upgrade your-package
```

In case you need to delete one of your package, you can just run:

```
python -m pip uninstall your-package
```

Please note that if you just want to change version, you do not need to uninstall and then reinstall the package: that's what the `upgrade` command is for! And if you need to know which version is already installed, then type:

```
python -m pip show your-package
```

These are the basic `pip` commands that you will need for this course. Do not hesitate to consult the official guide if you need more commands.

1.3 Running Python

We will start by creating and running a very basic Python program:

- Create a new file
- Rename it "dummy_python.py"
- Open it and write `print("Hello World!")`

You have several ways of running this program:

In an IDE

You can install a **Python interpreter** to use Python more easily. I recommend installing Visual Studio Code, which supports several programming languages, including Python.

In the CLI

To run your Python script in the CLI, open the terminal and run

```
python3 ./my_script.py
```

Beware to report the **correct path** to your script!

Jupyter Notebook

Jupyter Notebook is an interactive computational environment for data science and scientific computing, where you can combine code execution, rich text, mathematics, plots and rich media. It supports several other languages other than Python,

and it can be ran in the browser or in VSC. Here are the instruction on how to install it: Jupyter Documentation.

Opening in the Browser

After installing Jupyter Notebook (via Anaconda or pip):

1. Open a terminal (Linux/Mac) or Anaconda Prompt / Command Prompt (Windows).
2. Start Jupyter Notebook with the command:

```
jupyter notebook
```

3. Your default web browser will open automatically and display the Jupyter dashboard. You can create new notebooks or open existing files directly from the browser.

Opening in Visual Studio Code (VS Code)

Visual Studio Code can run Jupyter Notebooks directly inside the editor.

1. Make sure that **VSC is installed**.
2. Open VS Code and go to the **Extensions Marketplace** (left sidebar).
3. Install the extension called **Jupyter** (by Microsoft).
4. Open or create a notebook file (`.ipynb`).
5. You will see options to run each cell inside VS Code without opening the browser.
6. Make sure you have a Python interpreter installed and selected in the bottom-right corner.

I generally recommend opening notebooks in VSC rather than the browser. Another useful option is **Colab Notebooks** (see next section).

Verifying the Installation

To check if Jupyter Notebook was installed correctly, run the following command in your terminal or command prompt:

```
jupyter --version
```

If the version number is displayed, your installation is successful.

Colab Notebook

Google Colaboratory (Colab) is a free, cloud-based Jupyter Notebook environment provided by Google. It allows you to write and execute Python code directly in your browser, **without any installation**.

Colab gives you free access to powerful hardware such as **GPUs** (Graphics Processing Units). A GPU can make certain types of programs — including those involving data analysis and machine learning — run much faster than on a normal computer.

Opening Google Colab

1. Open your web browser and go to: https://colab.research.google.com.
2. You will need to sign in with your Google account.
3. The Colab homepage will show you recent notebooks, examples, and an option to create or upload new files.

There are a few things to keep in mind about Colab:

- You can create an empty Jupyter notebook directly on the webpage, but you can also open it from **your laptop**, **GitHub** (via link), or **Google Drive**.
- Notebooks are automatically saved to your Google Drive, and you can download them as `.ipynb` or `.py`.
- Notebooks by default run on CPU, but you have access to GPU thanks to Colab. To enable it, look for "Change runtime type" in the settings.

> Here, you can find an overview of Colab: Colab documentation.

1.4 Basic Python program

Let's familiarize a bit with Python. To begin with, Python enables you to perform **basic mathematical operations** and to **print text**, also with **user input**. Below are some examples:

Print some text:

```
print("Hello, world!")
```

Print text with user input:

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

Initialize variables and perform an operation on them:

```
x = 5
y = 3
print(x + y)  # Output: 8
```

Perform basic maths, included exponential operations:

```
print(2 + 5) # Output: 7
print(2 - 5) # Output: -3
print(2 * 5) # Output: 10
print(2 / 5) # Output: 0.4
print(2 ** 5) # Output: 32
```

We will now delve a bit deeper into Python's programming paradigms.

2 Git basics

Basic terminology:

Git	A distributed version control software.
GitHub	An online service to use Git in the browser.
Repository	A folder where you store the files of your coding project.
Clone	Create a local copy of that folder.
Branch	A pointer to a specific commit. The default branch is **master**.
Pull	Update the local branch with the latest commit.
Push	Update the remote branch with the changes from your local branch.

Git is a distributed, free and open-source version control software. It allows to track the changes and creating a version history. It also enables and facilitates collective code editing. It is distributed because every contributor has a full copy of the repository, included its history. For more information on Git, feel free to consult the Git Guides and the Git official documentation.

To be able to use Git, you need to install it on your laptop:

Windows

1. Download the Git installer from: Git Official Page
2. Run the installer and follow the setup instructions (default options are usually fine).
3. After installation, open `Git Bash` from the Start menu.

Linux (Debian/Ubuntu-based)

1. Open a terminal.
2. Run the following command:

```
sudo apt update
sudo apt install git
```

macOS

1. Open the Terminal app.
2. Run the following command:

```
xcode-select --install
```

3. Follow the prompts to install Git via the Xcode Command Line Tools.

Many available services exist to facilitate the use of Git. I recommend using **GitHub**, which is an online service. You will need to create an account here: GitHub.

Once you have installed Git and created your GitHub account, you can proceed to create your first **Git repository**. This is going to be a folder where you will

store the code of your project. In the top-right corner, click the "+" icon and select "New repository". You will be able to name your repository, to select its visibility (public or private), and please remember to check the box "add a README file"! Once the repository is created, you can **clone** it. This means that you will create a local copy of the repository on your laptop, and it is an action that you will need to do also when you want to collaborate to someone else's code. On the repository page, you will see a URL under the "Code" section. Copy the HTTPS URL and type the following command:

```
git clone your_https_address
```

Please note that other methods exist to create your repository and to clone it (e.g. SSH). If you are interested in other methods, feel free to consult the Git documentation.

Cloning the repository allows you to modify the code locally and pushing the modification to the remote when you are done. To perform this action, you can follow these steps:

1. Check the status of your repository:

   ```
   git status
   ```

   This command allows you to check whether there are any recent changes on the remote branch. Please beware that pushing changes without checking before if your local branch is up-to-date can lead to complications, so never hesitate to use this command.

2. **Optional**: If there are unstored changes, you should **pull** them:

   ```
   git pull
   ```

3. Add the files you want to commit:

```
git add filename
# or to add all changes
git add .
```

4. Commit the changes with a message:

```
git commit -m "Your commit message"
```

5. Push your changes to the remote:

```
      git push
```

3 Python basics

In this chapter, we will introduce some of the fundamental concepts in Python: **data types** and **structures**, **loops**, and **functions**.

3.1 Loops and conditonal statements

3.1.1 Loops

A loop allows a section of code to be repeated a certain number of times, until a stop condition is reached.

Listing 3.1: While loop example in Python

```
# This loop prints i until i reaches 6
i = 1
while i < 6:
    print("i =", i)
    i += 1
```

Explanation:

- We start with `i = 1`.
- The `while` loop continues as long as `i < 6`.
- Each iteration prints the current value of `i` and then increments it by 1.
- When `i` reaches 6, the loop condition becomes false and the loop stops.

The same logic can be expressed using `for`:

Listing 3.2: For loop example in Python

```
# This loop prints i until i reaches 6
for i in range(1, 6):
    print("i =", i)
```

Important Note

When coding in Python, you must pay attention to **indentation**. Indentation in Python is the whitespace (usually 4 spaces or a tab) at the beginning of a line that indicates a block of code. This is used to differentiate blocks of code **inside** and **outside** of a statement, (e.g., `while` or `for`). To avoid the (very common!) indentation errors, you can adopt the following precautions:

- Do not mix indentation with tabs and with spaces. I recommend always using **tab**.
- Use a **code editor** that will highlight indentation errors (recommended: **Visual Studio Code**).

3.1.2 Conditional statements

In Python, conditional statements control the conditions that regulate the execution of the program. You can set conditions using `if`, `elif` (short for *else if*), and `else`.

```
x = 10


if x > 0:
    print("Positive")
elif x == 0:
    print("Zero")
else:
```

```
    print("Negative")
```

Explanation:

- `if` checks the first condition.
- `elif` (else if) checks additional conditions if the first is False.
- `else` runs if none of the above conditions are True.

You might stumble upon the expression **control flow** the regulation of the order in which individual statements, instructions, or function calls are executed or evaluated. The control flow of a Python program is regulated by **conditional statements**, **loops**, and **function calls**.

3.2 Functions

In Python, a **function** is a reusable block of code that performs a specific task. Functions help make your code modular, organized, and easier to maintain.

You **define** a function using the `def` keyword, followed by the function name and parentheses `()`.

Listing 3.3: A simple function in Python

```
def greet():
    print("Hello, world!")
```

To run the function, you simply **call** it by its name:

Listing 3.4: Calling the function

```
greet()
```

Functions can accept input **parameters**:

Listing 3.5: Function with parameters

```
def greet(name):
    print(f"Hello, {name}!")
```

Calling it:

```
greet("Alice")
```

Functions can also **return** values:

Listing 3.6: Function that returns a value

```
def square(x):
    return x * x
```

Calling it:

```
result = square(5)
print(result)  # Output: 25
```

3.3 Basic data types

Data types define the kind of value a variable can hold, such as numbers, text, or more complex structures. Python supports several built-in data types, including `integers (int)`, `floating-point numbers (float)`, `strings (str)`, and `booleans (bool)`, as well as more advanced types like lists, tuples, and dictionaries. We will start by talking about **fundamental data types**.

Python supports several fundamental data types. The **integer** type `int` represents whole numbers, e.g., `x = 10`. The **floating point** type `float` is used for decimal

numbers, e.g., `pi = 3.14`. The **boolean** type `bool` has only two values: `True` or `False`, commonly used in logical operations, e.g., `is_valid = True`. The **string** type `str` stores sequences of characters, such as `name = "Alice"`.

```
# Examples of basic data types
x = 10              # int
pi = 3.14           # float
is_valid = True     # bool
name = "Alice"      # str
```

These types form the foundation of most Python programs and are automatically inferred during variable assignment. Here you can find some basic operations performed with data types:

```
# Integer and Float
a = 5
b = 2.0
print(a + b)        # 7.0
print(a * 3)        # 15
print(b / a)        # 0.4

# Boolean
x = True
y = False
print(x and y)      # False
print(x or y)       # True
print(not x)        # False

# String
s = "Hello"
```

```
t = "World"
print(s + " " + t)     # Hello World
print(s * 3)           # HelloHelloHello
```

3.4 Data structures

3.4.1 List

A **list** is a mutable, ordered collection in Python, defined using square brackets `[]`. Lists can contain elements of any data type and support operations like indexing, appending, and slicing.

```
# Creating a list
fruits = ["apple", "banana", "cherry"]

# Accessing elements
print(fruits[1])    # Output: banana

# Modifying the list
fruits.append("orange")
print(fruits)
# Output: ['apple', 'banana', 'cherry', 'orange']
```

Lists are versatile and commonly used for storing and manipulating sequences of items.

Creating a List

```
numbers = [1, 2, 3, 4, 5]
```

Indexing

Indexing refers to the process of accessing a specific element in a sequence, in this case a list, using its position (called **index number**). **Indexing in Python starts at 0**.

> It is very important to remember that **indexing in Python starts at 0**, meaning that the first element in a sequence has an index of 0, the second element has an index of 1, and so on. Thus, if you want to access the `n`th element of your list, you will find it at position `n-1`.

Below is an example of list indexing.

```
my_list = [10, 20, 30, 40, 50]


print(my_list[0])    # 10 (first element)
print(my_list[-1])   # 50 (last element)
```

Slicing

Slicing is the operation of extracting some elements from a list based on their indexes. It allows selecting a range of elements from a list by creating a shallow copy of a list containing only those elements.

Extract a portion of the list:

```
print(my_list[1:4])    # [20, 30, 40]
print(my_list[:3])     # [10, 20, 30]
print(my_list[::2])    # [10, 30, 50] (every 2nd element)
```

Appending

Appending is the operation of adding an element to a list. It is performed with the `append` command.

```
my_list = [1, 2, 3]
my_list.append(4)


print(my_list)  # [1, 2, 3, 4]
```

Removing Elements

```
numbers.remove(3)      # Remove first occurrence of 3
popped = numbers.pop() # Remove last item
```

Combining Lists

```
a = [1, 2]
b = [3, 4]
combined = a + b        # [1, 2, 3, 4]
```

Looping Through a List

```
for num in numbers:
    print(num)
```

3.4.2 Tuple

A **tuple** in Python is an immutable, ordered collection of elements. Tuples can hold elements of different data types and are defined using parentheses `()`. Since tuples

are immutable, their contents cannot be changed after creation. This makes them useful for representing fixed collections of data.

```
# Creating a tuple
person = ("Alice", 30, True)

# Accessing elements
print(person[0])  # Output: Alice

# Tuple unpacking
name, age, is_active = person
print(age)        # Output: 30
```

Tuples are commonly used when you want to group related data and ensure it remains unchanged throughout the program.

Creating a tuple

```
my_tuple = (1, 2, 3, 4)
```

Indexing and Slicing

```
print(my_tuple[0])      # 1
print(my_tuple[-1])     # 4
print(my_tuple[1:3])    # (2, 3)
```

Length of a tuple

```
print(len(my_tuple))    # 4
```

Looping through a tuple

```
for item in my_tuple:
    print(item)
```

Nested tuple and unpacking

```
nested = (1, (2, 3))
a, b = (10, 20)
print(nested[1][0])     # 2
print(a, b)             # 10 20
```

3.4.3 Dictionary

A **dictionary** (`dict`) is an unordered collection of key-value pairs in Python. Defined using curly braces `{}`, each key is associated to a specific value. Keys must be unique and immutable. Dictionaries are ideal for structured data and fast lookup based on custom keys.

Creating a Dictionary

```
person = {
    "name": "Alice",
    "age": 25,
    "city": "New York"
}
```

Accessing Values

```
print(person["name"])     # Alice
print(person.get("age")) # 25
```

Adding/Updating Values

```
person["email"] = "alice@example.com"  # Add
person["age"] = 26                     # Update
```

Removing Items

```
person.pop("city")        # Removes 'city' key
del person["email"]       # Deletes 'email' key
```

Looping Through a Dictionary

```
for key, value in person.items():
    print(key, value)
```

3.4.4 Set

A **set** is an unordered collection with no duplicate elements. **Curly braces** or the `set()` function can be used to create sets.

> Attention! When creating an **empty set**, **you have to use `set()`, not curly braces**; otherwise, **you will create an empty dictionary!**

Creating a Set

```
fruits = {"apple", "banana", "cherry"}
my_set = set() # Creates and empty set called my_set
```

Adding Elements

```
fruits.add("orange")
```

Removing Elements

```
fruits.remove("banana")    # Error if not found
fruits.discard("grape")    # No error if not found
```

Set Operations

```
a = {1, 2, 3}
b = {3, 4, 5}

print(a | b)    # Union: {1, 2, 3, 4, 5}
print(a & b)    # Intersection: {3}
print(a - b)    # Difference: {1, 2}
print(a ^ b)    # Symmetric difference: {1, 2, 4, 5}
```

Looping Through a Set

```
for fruit in fruits:
    print(fruit)
```