Introduction to Data Science

FCUP

Pedro G. Ferreira

Introduction to Data Science Tools

Data Science tools have evolved quite a lot in the last years. There are several programming tools that have developed an ecosystem of modules and libraries for data science, visualization, data management, statistics, optimized numerical computing and machine learning tasks. These are often effortlessly integrated, which allow us now to build data analysis pipelines, from data pre-processing to results communication.

A data analysis pipeline is a structured sequence of processes through in which raw data is collected, cleaned, transformed, analyzed, and visualized to derive useful insights.

Among the various programming languages, R and Python have become the standard tools for developing data science projects. Both offer distinct advantages, and the choice between them often depends on personal preferences and project requirements.

Personally, I find R to be excellent for statistics, graphics, and bioinformatics and ecological applications, with a vast array of functionalities provided by thousands of packages. It has a powerful syntax for creating high-quality graphics (e.g. using ggplot2) and for building robust data analysis pipelines, particularly through the *tidyverse* suite of packages.

Python has seen exponential growth of its data science ecosystem, making it the go-to language for many data scientists. Python excels in areas like machine learning (e.g., Scikit-learn), deep learning (e.g., PyTorch, TensorFlow, JAX, Keras), and generative AI (e.g., Hugging Face Transformers). Its rich ecosystem bridges the gap between data science and production environments, making Python ideal for building end-to-end data pipelines.

In this course we will use the Python Data Science Ecosystem, although most of what will be explored here can also be easily adapted to R.

Python Modes

Let start by remembering that Python can be run in two different modes: interactive and scripting mode.

In the interactive mode, you open the Python shell environment and you type one or more instructions or commands and the program immediately performs the respective calculation and gives you the result. For this, you can open the Python shell in the command line just by typing python (assuming that the Python programming language has been previously installed). You can also use an environment like IPython[1] for a powerful interactive shell.

The second mode is the scripting mode. Here, you create a full program or script, using a text editor or even better an IDE, import all the necessary modules, read and write the necessary data and files from the operating system, perform the computation to obtain your expected results. This file is often saved with the ".py" extension. To run your script, in the command line execute a command as follows:

python script_name.py [additional parameters if necessary].

This will allow you to run from start to end all the instructions in the script file. You can also configure your IDE to run this command, by making a call to the operating system within the IDE.

Integrated Development Environments (IDE)

IDEs (Integrated Development Environments) are powerful coding tools that assist the programmer in various aspects of software development, such as writing, debugging (identifying code errors), and testing. They streamline the development process by providing features like code completion, syntax highlighting, and debugging utilities, all within a unified interface.

A **notebook** is an interactive programming environment that combines code development, documentation, and visualization. It operates on the concept of a digital lab notebook, allowing you to not only write and test code but also document your work and visualize the results in an organized manner. This structure promotes **reproducibility** enabling others (or yourself) to replicate your results by following the same sequence of steps.

While some tools are specifically designed for notebook development, many modern IDEs support both traditional script-based coding and interactive notebooks. This flexibility allows developers to switch between different workflows depending on the project requirements.

Fortunately, there are numerous powerful and free IDEs available today, offering a range of features to suit different preferences and needs. The choice of IDE often comes down to personal preference and specific project requirements, so it's worth experimenting to find what suits you best.

Here are some of the most popular **IDEs**:

- Visual Studio Code (by Microsoft): A lightweight, highly customizable IDE with extensive support for extensions and features, making it ideal for a wide range of programming languages and projects.
- Spyder: A scientific IDE specifically designed for Python, often used in data science and machine learning applications.
- PyCharm: A comprehensive Python IDE offering strong support for Django, web development, and data science.

[1] https://ipython.org/

For notebook platforms:

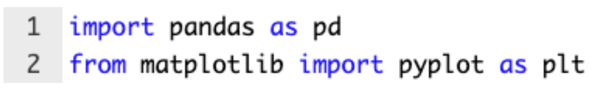
- JupyterLab & Jupyter Notebook: Widely used for data analysis, machine learning, and research. Jupyter provides an interactive environment for writing and executing code alongside documentation and visualizations.
- Google Colab: A cloud-based platform that allows you to run Jupyter notebooks without the need for local setup, providing access to GPUs for machine learning tasks.

If you haven't settled on an IDE or notebook platform yet, feel free to explore a few different options. Each offers unique features that might suit your programming style or computing environment. Try them out and choose the one that enhances your productivity and suits your project needs.

Modules

Modules, also called packages or libraries, are a way to organize code so that you can easily access functionalities that have been previously developed. They group related tools together and make easier to incorporate them into your program or script. Common example of data science modules are: *matplotlib, pandas, scikit-learn, scipy or nltk*.

We must import any module before using any associated functionality. Therefore, module imports are done at the very beginning of the programs. Let's see an example on how to import *pandas*, a tool to read from files and *matplotlib*, a module with tools to plot data.



```python
import pandas as pd
from matplotlib import pyplot as plt
```

Figure 1: Importing modules.

In Figure 1, we see that the module *pandas* is imported and the "alias" *pd* is used. This allows us to call the functions within this module using a much shorter word thus saving some time. From *matplotlib* we import the functions *pyplot* and call it *plt*.

Installing modules

Before we can effectively use modules in our program, we need to install them in our operating system, so that the Python programs knows where the corresponding files are located and is able to import them.

To install modules, you can use:

- Anaconda, a tool to install and manage python modules. It integrates with other tools (e.g. Spyder) to work in the Python Scientific Environment.
- Pip is a command line tool with several options to update and install modules.