Numeric PYTHON

Python Data Analysis with NumPy, Pandas, and Matplotlib

2
Numerical Programming

2.1 Definition of Numerical Programming

2.2 Overview

Figure 2.1 Analog data analysis in the office

The title of this book is “Numerical Python”, inspired by the term “numerical programming” – a phrase that, in everyday use, is often vague and sometimes even misleading. One might assume it refers to any kind of programming that deals with numbers – which would apply to almost all programs. Even applications that appear purely text-based, such as search engine algorithms, rely at their core on numerical methods. For example, the original PageRank algorithm developed by Google is based on the computation of an extremely large matrix with billions of rows and columns.[1)]

1 The enormous scale of the matrix used in the PageRank algorithm is illustrated in a lecture at Cornell University: “From the mathematical point of view, once we have M, computing the eigenvectors corresponding to the eigenvalue 1 is, at least in theory, a straightforward task. As in Lecture 1, just solve the system Ax = x! But when the matrix M has size 30 billion (as it does for the real Web graph), even mathematical software such as Matlab or Mathematica are clearly overwhelmed.” Source: *https://pi.math.cornell.edu/~mec/Winter2009/RalucaRemus/Lecture3/lecture3.html*

Numerical programming refers to the computer-assisted solution of mathematical problems using numerical methods – for example, solving linear and nonlinear systems of equations, numerical integration, matrix operations, statistical calculations, or optimization problems. These techniques are found in nearly all scientific and technical disciplines. Numerical programming is thus a subfield of scientific programming. The latter includes all types of software development in research and analysis contexts – including visualization, simulation, and data preparation.

The goal of this book is to introduce the key tools needed to implement numerical methods in the fields of data science, statistics, and machine learning with Python. The focus is on practical and efficient libraries such as NumPy, Pandas, and Matplotlib, which form the foundation for data-intensive applications.

2.3 The Relationship Between Python, NumPy, Matplotlib, SciPy, and Pandas

Python is a general-purpose programming language used in a wide variety of fields – such as system administration, web development, computational linguistics, and, as already mentioned, numerical programming, where speed and memory usage are critical. Pure Python – that is, without optimized libraries – is not competitive with specialized tools like MATLAB or R for demanding numerical tasks. The performance of the algorithms used is of utmost importance in numerical applications. For this reason, Python relies on its powerful modules: `NumPy`, `SciPy`, `Matplotlib`, and `Pandas`. As a result, Python is now one of the leading languages in numerical programming – and is often more efficient than MATLAB or R.

- **NumPy** provides the fundamental data structures, particularly multidimensional arrays (ndarrays) and matrices. It includes basic functions for creating, manipulating, and analyzing these structures and serves as the foundation for many other packages.
- **SciPy** builds on NumPy and extends it with a wide range of functionalities from scientific mathematics, such as numerical integration, interpolation, linear algebra, optimization, and Fourier transforms.
- **Matplotlib** enables graphical representation of data, as needed in many scientific contexts. It supports both simple plots and complex visualizations with multiple axes, subplots, or interactive elements.
- **Pandas** is specialized for working with tabular data (DataFrames). It offers powerful tools for time series analysis, grouping, pivot tables, handling missing data, and much more. Pandas also allows for reading and writing a wide range of common formats such as CSV, Excel, or JSON.

2.4 Python – An Alternative to MATLAB

Python is increasingly becoming the preferred programming language for data scientists, analysts, and scientific programmers. While tools like MATLAB and R used to dominate research, engineering, and statistics, the focus has now shifted more and more toward Python. This shift is due to a number of factors – including its high flexibility, a very active community, modular extensibility, and the fact that Python is free and open-source software.

MATLAB was originally developed for numerical computations in engineering and technical fields. It offers a specialized environment with numerous functions for linear algebra, signal processing, system simulation, and optimization. The language is self-contained, proprietary, and based on a paid licensing model. For many academic institutions, students, or startups, the licensing costs – especially for specialized toolboxes – can present a significant barrier.

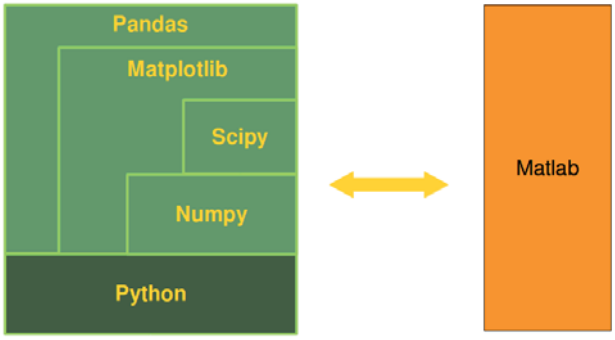

Figure 2.2 Relationship to MATLAB

Python, by contrast, was designed from the beginning as a general-purpose programming language. With libraries such as `NumPy`, `SciPy`, `Matplotlib`, `Pandas`, `SymPy`, and `scikit-learn`, Python is now capable of handling nearly all tasks that were once exclusive to MATLAB – and many more beyond, such as web development, automation, or machine learning.

One major advantage of Python is its seamless integration with modern development environments such as Jupyter Notebooks. This environment allows a combination of code, visualizations, and explanatory text – an ideal format for exploratory data analysis, scientific computing, and teaching.

Another strong point is the openness of its ecosystem: Python supports easy integration with C/C++ libraries, Fortran routines, or external tools. There are also many options for parallel and distributed computing (e.g., using `Dask` or `joblib`) as well as GPU-accelerated computing (e.g., `CuPy`, `PyTorch`, `TensorFlow`).

Last but not least, Python benefits from its enormous user base. Its open development culture leads to rapid improvements in tools, extensive documentation, tutorials, conferences, and a wealth of freely available resources.

3

Installation of NumPy, Matplotlib, Pandas, and JupyterLab

3.1 Introduction

This chapter describes how to install the libraries required for the subsequent chapters: NumPy, Matplotlib, Pandas, and JupyterLab.

JupyterLab is the modern, browser-based interface for interactive Python notebooks and is used throughout this book, as it is particularly well suited for experimenting with code, data analysis, and visualizations.

We present two installation methods:

- installation using `pip`, the standard package manager of Python,
- installation using `conda`, the package manager of Anaconda and Miniconda.

Figure 3.1 Speed Up!

Both methods have their advantages: while `pip` can be used directly with any Python installation, `conda` simplifies dependency management and provides optimized versions of many packages. Depending on the system and requirements, either method may be preferred.

Important note on Anaconda: Since 2020, Anaconda is no longer completely free for commercial use.[1)] Companies and organizations with 200 or more employees require a paid license. For private individuals, students, teachers, as well as smaller companies and non-commercial academic institutions, usage remains free – especially for purely private learning and study purposes.[2)] The underlying Python packages included in Anaconda (such as `NumPy`, `Pandas`, `Matplotlib`, and others) are open source and generally available free of charge. The license fees do not apply to the software itself, but to the commercial distribution, maintenance, and integration provided by Anaconda Inc., as well as to associated support services. Users who wish to avoid any potential licensing issues can instead rely on the freely available Miniconda or a custom-configured Python environment using `pip`.

Miniconda is a lightweight variant of Anaconda that also includes `conda` as a package manager but does not come with any preinstalled packages. It should be noted that the default channel ("defaults") is operated by Anaconda Inc. Organizations subject to the commercial licensing terms should therefore use alternative package sources such as conda-forge to avoid potential license costs.[3)]

3.2 Installation with conda and Miniconda

If the Anaconda or Miniconda distribution is used, the package manager `conda` is available. This package and environment manager allows for the easy installation and management of numerous scientific Python packages. When using Anaconda, most of the required packages are already installed and do not need to be installed manually as shown below. With Miniconda, the required packages can be installed using:

```
conda install numpy matplotlib pandas jupyterlab
# or alternatively, community-based and license-free:
conda install -c conda-forge numpy matplotlib pandas jupyterlab
```

One advantage of `conda` is that it automatically resolves dependencies and, in many cases, provides optimized, precompiled versions of packages – especially for numerically demanding applications.

1 See the license information at *https://www.anaconda.com/pricing*

2 Before using Anaconda in a professional or institutional context, the current license terms should always be carefully reviewed.

3 Please note that we cannot guarantee legal correctness here, especially since license terms may change at any time.

4
NumPy Introduction

4.1 Overview

4.1.1 What is NumPy?

NumPy is a module that provides basic data structures – multidimensional arrays and matrices – as well as important functionalities. These are used by other modules such as Matplotlib, SciPy, and Pandas.

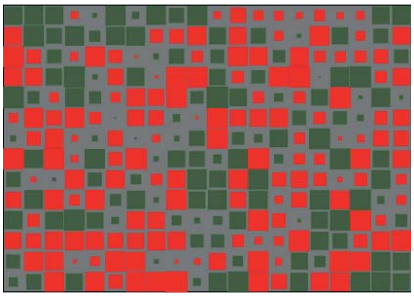

Figure 4.1 Visualization of a matrix as a Hinton diagram

The name NumPy is an acronym for "Numerical Python". From the outset, its design emphasized both memory efficiency and high computational performance. To this end, a significant portion of its core is implemented in C, enabling numerical and mathematical operations to be executed with exceptional speed.[1)]

NumPy extends Python with powerful data structures that allow efficient computations with large arrays and matrices – even for extremely large datasets ("Big Data"). In addition, the module provides a variety of high-quality mathematical functions specifically optimized for working with these structures. SciPy ("Scientific Python") is often mentioned alongside NumPy. It extends NumPy with additional functions such as minimization, regression, Fourier transformation, and many other tools.

1 See *https://stackoverflow.com/questions/1825857/how-much-of-numpy-and-scipy-is-in-c*

The diagram in Figure 4.1 was created with Python using NumPy and Matplotlib. It shows a so-called Hinton diagram for visualizing a 14×20 matrix: the size of the squares represents the magnitude of the matrix values, and the color indicates their sign – red for negative and green for positive values.

NumPy is based on two earlier Python modules that dealt with arrays. One of these is Numeric. Like NumPy, Numeric is a Python module for powerful numerical computations, but it is now outdated. Another predecessor of NumPy is Numarray, which was a complete rewrite of Numeric, but this module is also obsolete today. NumPy is the merger of these two, meaning it is built on the code of Numeric and the functionalities of Numarray.

4.1.2 A simple example

To work with NumPy, we first need to import it:

```
import numpy
# or, much more commonly, to save typing:
import numpy as np
```

In our first simple NumPy example, we define a one-dimensional NumPy array:

```
C = np.array([20.8, 21.9, 22.5, 22.7, 22.3, 21.0, 21.2, 20.9])
print(C)
```

This is the result of the code:

```
[20.8 21.9 22.5 22.7 22.3 21.  21.2 20.9]
```

Now we want to convert the above temperature values to degrees Fahrenheit. This can be done very easily with a NumPy array. The solution to our problem lies in simple scalar operations:

```
print(C * 9 / 5 + 32)
```

The code produces the following result:

```
[69.44 71.42 72.5  72.86 72.14 69.8  70.16 69.62]
```

Compared to this approach, the pure Python solution[2], which converts a list to a list of Fahrenheit temperatures using a list comprehension, turns out to be cumbersome:

```
cvalues = [20.8, 21.9, 22.5, 22.7, 22.3, 21.0, 21.2, 20.9]
fvalues = [x * 9 / 5 + 32 for x in cvalues]
print(fvalues)
```

The result appears as follows:

```
[69.44, 71.42, 72.5, 72.86, 72.14, 69.8, 70.16, 69.62]
```

So far, we have referred to C as an array. However, the internal type designation is `ndarray` or, more precisely, "C is an instance of the class `numpy.ndarray`":

```
print(type(C))
```

The result is:

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

In the following, we will usually use "array" and "ndarray" synonymously.

4.2 Comparison of NumPy Data Structures and Lists

4.2.1 Key Differences

The data structures of pure Python, i.e., without NumPy or others, offer significant advantages:

Advantages of Python data structures:

- Integers and floats are implemented as powerful classes. Thus, integer numbers can become almost "infinitely" large or small.[3]
- Lists offer efficient methods for inserting, appending, and deleting elements.
- Dictionaries offer fast lookups.

Advantages of NumPy data structures over Python:

- Array-based computations
- Efficiently implemented multidimensional arrays
- Designed for scientific computing

2 That is, Python without using the NumPy module

3 They are ultimately limited by memory size and still infinitely far from "infinite"!