rescipy-lectures

Dec 16, 2022

Contents

Tutorials on Python for scientific research.

reScipy lectures is a collection of tutorials aiming to provide researchers with documentation to easily start performing scientific data analysis with the Python programming language.

These tutorials are heavily based on the Scientific Python Lecture Notes, while they attempt to provide additional information on the Python ecosystem, specifically for scientific research.

This website is part of the reScipy project.

CHAPTER 1

1. Python language

1.1 1.1 Introduction

Python is a multi-platform, free and open-source programming language, first released in 1991. The current version, Python 3, is one of the most widely used programming languages to date. Python is an *interpreted* language (as opposed to *compiled* as C or Fortran), which means one does not need to compile the code before executing it. This allows you to run Python code *interactively* (you can modify it and immediately run it) through the provided Python **interpreter**, a command-line tool which can run on-the-fly the code you type in. Python is defined as a high-level, general purpose language and supports object-oriented programming.

An overview of the Python scientific ecosystem is given in section 1.1 of the SciPy Lecture Notes.

1.2 1.2 Installation

Python comes in various flavors and can be installed in different ways. However, the easiest way to have Python in your system is to install a scientific distribution such as the Anaconda distribution, which provides a manager for a full set of libraries and software to perform data analysis with Python.

1.3 1.3 Usage

After the installation of Anaconda Python 3 version, you can run the Ananconda Navigator which provides you with the tools to use and setup your Python installation.

Between these tools, JupyterLab is the core application you will use to actually perform your data analysis. It consists of a browser-based user-interface for managing Jupyter Notebooks. A Jupyter Notebook is an interactive-programming browser-interface with a Python interpreter running under the hood.

After opening JupyterLab, in the Launcher main window you can open a new Notebook (or do *File* > *New* > *Notebook*) and start already to type some Python code. In the empty cell that appears, you can try to type:

```
print("Hello World!")
```

or simply

```
7 * 6
```

and press the Run button (the small triangle just above the cell) or just hit *Shift+Enter*. Congratulations, you just run your first Python code! See the next paragraph for more information on Python programming.

To have an idea on what one can actually do into a Jupyter Notebook, have a look at the Bokeh and Holoviews galleries, which provides some interactive examples.

You can go through the JupyterLab User Guide and the Jupyter Notebook documentation to to get familiar with the user interface.

1.4 1.4 Actual programming

Printing `Hello World!` has been pretty exciting but with Python you can do quite more!

Actually, one does not need at all to be a programmer in order to use Python for data analysis, still, knowing the basis of the language will help you immensely in doing it properly.

The section 1.2 of the SciPy Lecture Notes dedicated to Python provides a very good introduction to the language itself, and initially one should at least be familiar with the sections 1.2.1, 1.2.2 and 1.2.3. It would still be useful to go through the rest, also if you are interested in write some code or automatize some operations. In the latter case, the Spyder development environment, included in Anaconda, could come in handy.

Depending on your inclination, these same basic aspects are covered in more detail in chapters 1 to 5 of the excellent official Python tutorial, while if you are planning to develop code, consider going through the chapters 6 to 10.

1.5 1.5 Modules and packages

In the following are reported a few key concepts strictly related to programming, but which have to be clear in order to use Python for data analysis.

When launching a new Notebook (*i.e.* a new Python interpreter) you are provided by default with just a very basic set of functions, such as *e.g.* `print()` seen above or `abs()`:

```
abs(-7)
```

which returns the absolute value of a number.

In order to perform something more, you have to load (or more precisely to **import**) additional *packages* into the active session. Many of these packages are already present by default, others can be installed. For example, supposing you want to calculate the cosine of π, you would do:

```
import math
math.cos(math.pi)
```

This piece of code already illustrates two important aspects in Python:

- **Importing.** Here, `math` is a *module*, which is included by default in the Python distribution, but needs to be 'activated' with the `import math` statement.

- **Object-oriented programming.** The `math` module contains several objects, like functions (which are called *methods*, such as `cos()`) and variables (which are called *attributes*, such as `pi`). These objects are accessed through the dot `.` notation. So `math.cos()` or `math.sin()` give the cosine and sine functions, respectively, while `math.pi` returns the π constant.

To better illustrate this let's try a variant of the importing:

```
from math import cos, pi
```

This line of code is pretty self-explaining. In this way, `cos()` and `pi` have been made directly available and one can just write:

```
cos(pi)
```

with the same result as before.

You can inspect the type of `cos` and `pi` objects with the `type()` function. For example:

```
type(pi)
```

will return `float`, indicating `pi` is a floating point number.

To summarize, here `math` is a *module*, which contains several *methods* (*i.e.* functions, as it is `cos()`) and *attributes* (*i.e.* variables, as it is `pi`).

Similarly, other types of objects in Python can have their own methods and attributes. As an example, the object `mydata`, which we assume has been properly constructed, can posses, let's say, the `mydata.temperature` attribute (which would probably be a float number representing the temperature at which data has been acquired) or the `mydata.normalize()` method (which, for example, could rescale `mydata` values, so that the integral under the curve is equal to one).

A collection of modules is called a *package*. So to give another example, let's take the `convolve` *method* contained in the `signal` *module* of the `scipy` *package*. To access this function any of this will work:

```
import scipy
scipy.signal.convolve()
```

```
from scipy import signal
signal.convolve()
```

```
from scipy.signal import convolve
convolve()
```

```
from scipy.signal import convolve as conv
conv()
```

In the last example, `convolve` has been imported with the *shorthand* `conv`. This is an useful and extensively used practice, especially when you need to use the same object several times.

The same concept of importing applies similarly to Python *scripts*: simple text files, you may have written by yourself, typically with '.py' extension, and containing custom definitions of functions or other objects you want to reuse.

To have an insight into scripts and modules, check the section 1.2.5 of SciPy lectures and chapter 6 of the Python tutorial.

CHAPTER 2

2. Data handling

2.1 2.1 Introduction

Standard Python containers, such as lists and dictionaries, are not designed for numerical computation. In order to efficiently perform scientific calculations, the **NumPy** package provides powerful multi-dimensional array objects, with related functions and tools for numerical calculations.

In the following sections *NumPy arrays* and higher-level data containers built on top of them (namely **pandas** *DataFrames* and **xarray** *Datasets*) will be introduced. The last section will describe how to load scientific data and conveniently store it in the *NeXus/HDF5* file format.

Important note: before going through this lecture you need to add the `conda-forge` channel to your environment and install the `nxarray` package through pip. Check the section on *Packages installation* to learn how to do it.

2.2 2.2 NumPy arrays

Numpy arrays, provided by the NumPy package, are the core objects for numerical calculation in Python. Such arrays are multi-dimensional data containers, efficiently mapped into hardware memory. An array, for example, could contain measurement of an experiment, a recorded signal, pixel intensities of an image or point values in multi-dimensional space.

To have a first overview of the NumPy package and *NumPy arrays* follow the related chapter 1.4 of the SciPy lectures, the official quick-start guide and the basics for beginners. Few of these examples will also introduce already some basics of data plotting. For a more technical insight, you can have also a look at chapter 2.2 of the SciPy lectures.

2.3 2.3 Extending NumPy

NumPy provides a set of very powerful functions for data analysis, and *NumPy arrays* are extremely robust and efficient for numerical computation. On the other hand, in practical data analysis, it could be difficult to handle, explore and relate between them these arrays.

To this concern, several Python packages exist, providing more flexible and expressive data structure, extending *NumPy arrays* (which are always under the hood) and making them easier and more intuitive to use.

The most popular of these packages is pandas, which is designed to simplify the handling of 'labeled' and tabular data, by providing the *Series* (1D) and *DataFrame* (2D) data structures. The 10 minutes introduction and the basic functionalities on the official website are a good starting point to have an idea of this package.

Pandas is a very powerful package that makes tabular data handling much easier. Nevertheless, it does not well support higher-dimensional data and it is missing an integrated management of metadata (the attributes related to your data), thus making it not always suited for scientific research data.

To this concern, the xarray package, built on top of NumPy and pandas, fills in excellently these lacks, being often the recommended choice in managing scientific research data. **xarray** provides the *DataArray* structure, a labeled N-dimensional array with its coordinates and attributes, and the *Dataset* structure, a container of *DataArrays* sharing the same coordinates. To have a better idea of the implementation of these two data structures, check their design description and to have an insight on their basic usage have a look at the quick overview on the official website.

Here, it is worth noting that another important feature provided by xarray is the possibility to easily extend it with domain-specific functionalities, by adding custom 'accessors' on the xarray objects. This aspect will be covered in more detail in chapter 4. Analysis.

2.4 2.4 Loading and saving

The examples and tutorials in the previous sections already showed some basics of data loading and saving, for NumPy, pandas and xarray respectively.

Despite all these packages support import/export of HDF5 (a file format designed to efficiently store and organize large amount of data), none of them provide an integrated interface to the NeXus file format, the standard *de facto* for scientific data storage, based on HDF5 and increasingly adopted in laboratories and large-scale facilities all over the world.

With this respect, the nxarray package comes into play, bridging xarray with the NeXus format. This package actually extends xarray, providing convenient loading and saving methods for NeXus files, directly to *DataArrays* and *Datasets*. The architecture of a NeXus file closely resembles the structure of an xarray *Dataset*, and indeed, even if they have been developed independently, both of them are actually specifically designed for handling scientific data with its relevant metadata.

After installation, you can already start to use nxarray, by importing it at any moment with:

```
import nxarray as nxr
```

Now the `nxr.save()` method will be available to xarray objects. For example, the `ds` *Dataset* of the previous examples can be saved to a NeXus file to disk simply with:

```
ds.nxr.save('ds.nxs')
```

You can load it back, let's say to another *Dataset* `my_ds` with:

```
my_ds = nxr.load('ds.nxs')
```

and you can check that the whole structure of your Dataset is the same.

A *DataArray* can also be saved to a NeXus file. In this case, a *Dataset*, with your *DataArray* inside, will be created and saved to file. For example the `data` *DataArray* of the previous examples can be equally saved with:

```
data.nxr.save('data.nxs')
```

This time, when you will load it, a *Dataset* will be returned, with your original *DataArray* inside it:

```
ds2 = nxr.load('data.nxs')
my_data = ds2['data']
```

This section concludes with a consideration. **NeXus**, as reported by its website introduction, 'is an effort by an international group of scientists motivated to define a common data exchange format'. Indeed, NeXus/HDF5 files are the best choice to save scientific data, and scientist are (and should) adopting it extensively. Data values stored in .nxs file are in binary format, which is the most efficient way to handle numbers, in term of disk space and computational speed. At the same time, .nxs files can be easily loaded and their content visualized quickly, together with all the relevant metadata associated. If you still are inclined to save your data as plain text because 'I can see what's inside', most probably you are simply using the wrong tools to access your data. As the wise man said:

"You are a scientist, not a novelist. Save your data as binary, not as text."

CHAPTER 3

3. Plotting

3.1 3.1 Introduction

Now that you learn how to properly handle your data, it is time to visualise it!

Plotting in Jupyter Notebooks, and in general in Python, relies mainly on two alternative but complementary packages: matplotlib and bokeh.

Matplotlib is the standard and widely used Python plotting backend that provides static, publication-quality plots, which can be tuned and controlled down to the smallest detail.

Bokeh, on the other hand, provides interactive and dynamic plotting inside Jupyter Notebooks and it is best suited to easily explore your data.

To make plotting (but not only) easier and more descriptive, the holoviz project provides several packages extending matplotlib and bokeh (such as hvplot and holoviews). Make sure to check them out to be proficient in plotting with Python.

Note: to go through this lecture you need to install the `hvplot` package from the `pyviz` channel. Check the section on *Packages installation* to learn how to do it.

Before even introducing how these plotting packages work, let's try them out quickly. Indeed, in case your data is available as an xarray (or pandas) object, you can already plot it very easily with the available methods and without any prior knowledge of the plotting packages. Supposing you have a `ds` *Dataset* (such as the one from the xarray examples), it can be simply plotted just with:

```
ds.plot()
```

A *matplotlib* plot will appear, representing the first available DataArray in your Dataset. Obviously the same `.plot()` method is directly available also for single DataArrays (as well as for pandas Series and DataFrames). As you can see, matplotlib plots are static images (see Section 3.2 for more information).

To try out *bokeh* instead, the simplest way is to import the `hvplot` package, which specifically provides additional methods to xarray (and pandas) objects for bokeh plotting:

```
import hvplot.xarray
```

Now you can simply type:

```
ds.hvplot()
```

And in this case a *bokeh* plot will appear, representing the same data as the plot before. As before the `.hvplot()` method is available for DataArrays (as well as for pandas Series and DataFrames). As you can see, bokeh plots can be zoomed and panned and many other tools are provided (see Section 3.3 for more details).

3.2 3.2 Publication-quality figures: matplotlib

Matplotlib is the most used Python package for 2D plotting. The interface to the plotting library is provided by its *pyplot* module. Supposing we have few simple numpy arrays:

```
import numpy as np

x = np.linspace(-np.pi, np.pi, 256)
c = np.cos(x)
```

we can import *pyplot*:

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

and create a plot simply with:

```
plt.plot(x, c)
```

The plot properties can be set through the dedicated *plt* methods, *e.g.*:

```
plt.title("My Simple Plot")
plt.plot(x, c, label='myData')
plt.legend()
```

This workflow is the so-called *pyplot-style* approach, in which everything is passed to the *plt* module, which automatically manages the plots. An alternative way to use the matplotlib interface is the so-called *Object-Oriented (OO)-style* in which plot objects are assigned explicitly to variables on which methods are called:

```
fig, ax = plt.subplots()
ax.set_title("My Simple Plot")
ax.plot(x, c, label='myData')
ax.legend()
```

To plot an image (*i.e.* a 2D array, here `yy`) you can use `imshow()`:

```
plt.imshow(yy)
```

To have a basic introduction on matplotlib functioning and terminology go through the Usage Guide, while to learn using pyplot check the tutorial and the dedicated SciPy lecture.

Useful tip: to understand when a plot should show up (or why it is not) be sure to read the interactive mode section of the Usage Guide.

If you are looking for a simpler and quicker interface to matplotlib, have a look at seaborn, an high-level interface built on top of matplotlib and integrating closely with pandas data structures. Check out the gallery to see how to obtain publication quality figures with few lines of codes.

3.3 3.3 Interactive plotting: bokeh

Matplotlib is very powerful when it comes to control each aspect of your figure. On the other hand it is not always the quickest way to explore your data. In this case, Bokeh comes into play, providing several tools to explore or stream your data, but also to combine plots and widgets within applications or dashboards.

The standard import for Bokeh in a Jupyter Notebook is:

```
from bokeh.plotting import figure, output_notebook, show
output_notebook()
```

Here `output_notebook()` tells Bokeh to show the plot inline within the notebook.

To create a plot just type:

```
p = figure()
p.line(x)
show(p)
```

Also in this case we can explicitly use the *OO-style* and change the plot properties:

```
p = figure(title="simple line example", x_axis_label='x', y_axis_label='y')
l = p.line(x, legend_label="Temp.", li)
l.glyph.line_width = 2
show(p)
```

To learn more about Bokeh usage check the User Guide and the Tutorials.

CHAPTER 4

Appendices

4.1 Packages installation

4.1.1 via Anaconda Navigator

To install a package from Anaconda Navigator, select the `Environment` section on the left pane. The default environment should be `base (root)` (in case select a different environment). Check that the filter is set to `All` and not only to `Installed`. Search for the package you want to install, select it and hit `Apply` on the lower right corner. The package will be installed with all the related dependencies.

If you know the package is available only in a certain channel, press the `Channels` button, select `Add` and insert the name of the channel you want to add. Indeed, many packages and dependencies are available through the `conda-forge` channel. It is recommended that you add it to your environment.

If you cannot find the package in any channel, then the package is not available through conda and you need to install it with pip. See the next section.

4.1.2 via Anaconda Prompt

You can install a package launching your terminal (or Anaconda Prompt as Administrator if you are on Windows) and typing:

```
$ conda install [packagename]
```

Or in case you need to install it from a particular channel:

```
$ conda install -c [channelname] [packagename]
```

To add a channel to your environment, so packages are installed directly from it, just type:

```
$ conda config --add channels [channelname]
```

In case the package is not available through conda, you can install it with pip:

```
$ pip install [packagename]
```

4.2 Python packages for scientific research

This page lists some useful packages for scientific research with Python. If you are new to Python, start with the introductory tutorials of the *reScipy lectures* to learn how to use this programming language to perform data analysis.

Packages are divided into different categories, depending on the application domain, with a link to the relevant section of the lectures. Many of these packages are already included in the most common Python distributions, others can be installed as explained in their documentation or in the *Packages installation* appendix.

4.2.1 Environments

See *Python language* for a brief introduction to these packages.

- jupyterlab | Extensible user interface for interactive and reproducible computing.
- spyder | Integrated development environment for scientific data analysis.

4.2.2 Data structures

See *Data handling* for a brief introduction to these packages.

- numpy | Multi-dimensional arrays and matrices, with a collection of high-level mathematical functions to operate on them.
- pandas | Data structures and operations for manipulating columnar and tabular data, built on NumPy.
- xarray | Labeled multi-dimensional NumPy-based arrays with dimensions, coordinates and attributes.

4.2.3 NeXus/HDF5

See *Data handling* for a brief introduction to these packages.

- nexusformat | Python API to open, create, and manipulate NeXus data written in the HDF5 format.
- nxarray | xarray extension for high-level NeXus file input and output.

4.2.4 Core plotting libraries

See *Plotting* for a brief introduction to these packages.

- matplotlib | Core plotting library for Python and NumPy arrays.
- bokeh | Plotting library for interactive visualization in web browsers.

4.2.5 High-level plotting

See *Plotting* for a brief introduction to these packages.

- seaborn | High-level interface to make plotting with matplotlib quick and effective.

- hvplot | High-level plotting library to quickly produce bokeh plots from pandas and xarray objects, based on holoviews.
- holoviews | Descriptive data plotting built on top of matplotlib and bokeh.

4.2.6 Data analysis

See the related websites and the documentation links provided for an exhaustive description of these packages.

- mantid | General-purpose graphical user interface and Python library to support the processing of materials-science data.
- scipy | Performant algorithms for scientific computing such as optimization, integration, interpolation, algebraic equations, differential equations (see also Scipy : high-level scientific computing and Image manipulation and processing using Numpy and Scipy).
- scikit-image | Collection of algorithms for image processing (see also Scikit-image: image processing).
- lmfit | Advanced high-level interface for non-linear optimization and curve fitting.
- nexpy | Graphical user interface to easily access and analyse NeXus data (see in particular NeXpy - Python Graphical User Interface).

4.2.7 Visualization and dashboarding

If you are specifically interested in an overview of the packages for data visualization and dashboarding then pyviz.org is what you are looking for. In the following are listed some of these packages.

Multi-dimensional visualization

- ipyvolume | Library to visualize 3d volumes and 3d scatter plots in jupyterlab.
- mayavi | Visualization of scalar, vector and tensor data in 2d and 3d (as Python module or through graphical user interface).

Dashboarding

- streamlit | Framework to quickly build shareable web apps from scripts.
- panel | Library to create custom interactive web apps and dashboards with user-defined widgets.
- voila | Conversion of Jupyter notebooks into shareable interactive dashboards.
- ipywidgets | Widget library to build interactive user interfaces within jupyterlab.

CHAPTER 5

Feedback

Please report any feedback or error by opening an issue on the issue tracker of the code repository.