Introduction to Jupyter with Python

Anne Kværnø (and modified by André Massing)

Mar 10, 2020

1 Introduction

In the numerics part of TMA4125/30/35 Mathematics 4N/D all material will be made available in the form of Jupyter notebooks. This is a web-based system, making it possible to combine written text and executable code in one document. And this is quite convenient for a course in Numerical methods, as implementation is a crucial element of the topic. But implementing code is as you probably have experienced exhaustingly time-consuming. In the notebooks of this course the algorithms are already implemented, and they can be used immediatly. But you are supposed to be able to do your own tests, and also to make minor changes and improvements to the already implemented algorithms.

However, please note that you will not learn programming in this course. In fact, you are expected to be able to read, understand and modify simple codes. Some knowledge of Python or Matlab is a prerequisite. This note gives you a very short introduction to the most important constructions for the use of Python for numerical simulations. For those of you who are only familiar with MATLAB, take a look at [NumPy for Matlab users](https://docs.scipy.org/doc/numpy/user/numpy-for-matlab-users.html).

All the course notes can be downloaded from the course wiki-pages and then used on your own computer. See How to install and use Anaconda if you would like to try this option.

For more advanced programmers, there will also be pdf-versions of the documents (not this one), with the python code available in a separate file. These can be found on the wiki-page. But as all figures and tables are generated using the python codes, these are most useful if you run the code in parallel with your reading.

2 How to use a Jupyter notebook

A Jupyter notebook is composed of cells. These can contain either text (like this one) or code (as demonstrated below). As soon as you have copied the Jupyter note file to your own domain, you are free to use it and change it as you like. You are encouraged to experiment with the codes. You can not make any harm, if your note is completely messed up, just make a new copy.

If you have installed the Anaconda Python distribution as suggested above, you can use Anaconda Navigator to start the Jupyter notebook application. If you prefer the command line in a terminal, if can just type `jupyter notebook`.

There is plenty of documentation regarding the Jupyter Notebook which is part of the Jupyter project, see for instance

- Jupyter notebook documentation
- A gallery of interesting Jupyter Notebooks

Also, the JupyterLab is a modernized alternative to the Jupyter Notebook. It can be launched from the Anaconda Navigotar as well.

As soon as your first notebook is open, we recommend that you to make yourself more familiar with the Jupyter environment. Take a look at the tool menu, and see what is available. There are also many keyboard shortcuts (see Help -> Keyboard Shortcuts), learning some of those will make your work faster

Let us start: Below, you will find a short example of code. You can run this either by pressing either one of the following short cuts:

- **shift+enter**: executes cell and takes you to the next cell (if there is one)
- **ctrl+enter**: executes cell but stays in the same cell
- **alt+enter**: executes cell, insert a new cell below and takes you there

Alternatively, you can use `Run` from the menu.

```
a = 2
b = 6
c = a+b
print(c)
```

When a cell has been executed, the data are stored and can be used later.

```
print(a, b)
```

NB! New cells can be inserted from the menu (`Insert`) or by pressing `Esc b`. This is useful if you just want to try something without making changes in the existing code.

Executing the following cell loads a non-default css style for the notebook. Make sure that you download the corresponding css file `tma4125.css` from the course web page.

```
from IPython.core.display import HTML
def css_styling():
    styles = open("tma4125.css", "r").read()
    return HTML(styles)

# Comment out next line and execute this cell to restore the default notebook style
css_styling()
```

The code cells are executed in the sequence you choose. There is a number to the left of each cell helping you to keep track of this. So if you go back and re-execute one of the first cells, data from previous executed cells are still available. This may sometimes cause strange errors. So once in a while it may be useful to restart completely, which can be done from `Kernel` in the menu.

3 Numerical computations in Python

The implementation of numerical methods is not necessarily very complicated, most of the time we can rely on the following constructions:

- Functions
- Loops and control statements
- Vectors and matrices
- Plots and visualization

The use of those will be demonstrated later in this introductory note.

Our computations will heavily rely on two modules:

- Numpy: Arrays (vectors and matrices), and all the standard mathematical functions.
- Matplotlib: Graphs and figures.
- SciPy Library: Fundamental library for Scientific computing, including numerical routines for e.g. numerical integration, interpolation, optimization, linear algebra, and statistics.

Occassionally, we might also use

- Sympy: Python library for symbolic mathematics.
- pandas: Sophisticated Python data analysis libray.

A good entry point to all these libraries is the SciPy web-page.

In this course, the first cell will always be something similar to the next one below. All required modules and functions are imported, and Jupyter is made ready for showing plots inside the the document. *This cell should always be executed first!*

```
# Import required modules.
import numpy as np
from numpy import pi              # Import the number pi
from numpy.linalg import solve        # To solve Ax=b
import matplotlib.pyplot as plt

# For plotting.
newparams = {'figure.figsize': (8.0, 4.0), 'axes.grid': True,
             'lines.markersize': 8, 'lines.linewidth': 2,
             'font.size': 16}
plt.rcParams.update(newparams)
```

You may change the parameters in `newparams` to your own preferences. Then make your first plot:

```
x = np.linspace(0,1,101)
plt.plot(x, np.cos(2*pi*x))
```

The function `linspace` deserves closer attention, since we will be using it extensively throughout the course. `x = np.linspace(a,b,n+1)` generates an array of $n+1$ equidistributed points, that is

$$x = [x_0, x_1, \dots, x_n], \qquad x_i = a + ih, \quad h = (b-a)/n, \quad i = 0, 1, \dots, n.$$

So `x = np.linspace(-1,3,5)` generates the array $[-1, 0, 1, 2, 3]$. Try it yourself:

```
# Insert your code here
```

Functions defined on some intervals are only computed on discrete points x_i, and plots are made as straight lines between those. If the intervals between the points are small, it will look like a continuous function. In the preceeding notes, we always will represent the x-values of a certain interval (a, b) by `x = linspace(a,b,n+1)`, with `n` large (typically 100). The function will also be used whenever a uniform distribution of points is requested.

3.1 Numpy:

Numpy is the Python module handling vectors and matrices and all different kind of linear algebra. Vectors and matrices are considered as 1- and 2-dimensional arrays. Contrary to MATLAB, there is no distinction between row- and column vectors.

The following demonstrates some use of this module. Let A and b be given by

$$A = \begin{pmatrix} 1.4 & 2.2 & -1.0 \\ 1.6 & -2.7 & 1.2 \\ -3.2 & 1.2 & -1.8 \end{pmatrix}, \qquad \mathbf{x} = \begin{pmatrix} 1.0 \\ -2.0 \\ 3.0 \end{pmatrix}$$

In Python, these are represented by

```
A = np.array([[1.4, 2.2, -1.0],        # A: A 2-dimensional array
           [ 1.6, -2.7, 1.2],
           [ -3.2, 1.2, -1.8]])
x = np.array([1.0, -2.0, 3.0])         # x: A 1-dimensional array
print('A = \n', A)
print('\nx = ', x)
```

Notice that the index starts with 0 in Python, but usually with 1 in mathematics. Thus x_3 in mathematics corresponds to `x[2]` in Python, and a_{21} to `A[1,0]`.

```
print(A[1,0])
print(x[2])
```

If `a` and `b` are two arrays of the same dimension, standard operations as `+`, `-`, `*` and `/` are always element by element operations. The same is the case for `a**p`, returning each element of `a` in the power of `p`.

```
a = np.array([1, 2, 3])
b = np.array([3, 4, 5])
print('a+b = ', a+b)
print('a-b = ', a-b)
print('a/b = ', a/b)
print('a*b = ', a*b)
print('a**b = ', a**b)
print('a**2 = ', a**2)
```

Similar things happen if you perform this operations between an array `a` and scalar `p`, which just means that you perform the corresponding operation elment-wise. Try it out:

```
# Insert code here
```

The usual matrix-vector product is done by the command `A@x` or `dot(A,x)`. You can also use the same operators to compute the inner product between two vectors `x` and `y`.

```
y = A@x
print(y)
ip = y@x
print(ip)
```

And here is a list of quite useful linear algebra functions:

- `solve(A,b)` : Solves a linear system $A\mathbf{x} = \mathbf{b}$. Needs to be imported from `numpy.linalg`.
- `ones(n)` : returns $[1, 1, \ldots, 1] \in \mathbb{R}^n$
- `zeros(n)` : returns $[0, 0, \ldots, 0] \in \mathbb{R}^n$
- `len(x)` : The length of an array
- `shape(A)` : The size of an array. Returns (n, m) if $A \in \mathbb{R}^{n\times m}$
- `eye(n)` : The $n \times n$ identity matrix I_n.

Test them out, and make yourself familiar with them.

```
# Insert your code here
```

Matplotlib. Matplotlib is the module taking care of plots.

Example: Make a plot of the function $f(x) = cos(2\pi x)$ on $[0, 1]$.

```

x = np.linspace(0,1,100)
y = np.cos(2*pi*x)
plt.plot(x, y)
```

This is a very minimalistic example. The next example shows how to add things like labels and legends, and to choose different colors or line styles.

Example: Make a plot of the function

$$f(x) = x^2 + 2x, \qquad g(x) = 2x\sin(2\pi x) \text{ and } h(x) = x^3$$

on the interval $-1 \leq x \leq 1$.

```
x = np.linspace(-1, 1, 101)      # The x-values of the interval (-1,1)
f = x**2 + 2*x                # Notice the element by element operations.
g = 2*x*np.sin(2*pi*x)
h = x**3
plt.plot(x, f, 'r--')           # Plot f, red stapled line
plt.plot(x, g, 'b-.')            # Plot g, blue line
plt.plot(x, h, 'g-')            # Plot g, blue line
plt.xlabel('x')
plt.ylabel('y')
plt.title('A nice plot of the functions f,g and h')
plt.legend(['f(x)', 'g(x)','h(x)'])
```

You may notice that there is some output before the plot. If you find this annoying, it can be removed by adding a semicolon ; after the last command in the cell (try it).

3.2 Loops and control statements

Contrary to most other programming languages, MATLAB included, there are no `end` type expression in Python. Blocks to be executed within a loop or similar have to be indented, and the length of the indentation must be consistent (in these notes we use 4spaces). A block ends when the indentation ends.

In the following, some loop and control structures are demonstrated.

For loop: The aim of this program is to make a nice formatted output of the element of an array.

```
x = np.array([1.3, 4.6, 2.1, -5.8, 2.3, -3.2])             # The array
n = len(x)                                              # The length of the array
for i in range(n):                                      # Start of the loop
    print('i = {:2d}, x = {:6.2f}'.format(i, x[i]))     # Formatted output
print('\nAnd this concludes the loop')                  # End of the loop
```

Notice that `for i in range(n)` corresponds to $i = 0, 1, \ldots, n-1$.

Now write every second term, starting with the second one:

```
for i in range(1,n,2):                                  # Loops over i=1,3,5
    print('i = {:2d}, x = {:6.2f}'.format(i, x[i]))     # Formatted output
```

While loop: Alternatively, you could use a while loop:

```
i = 0
while (i<n):                                           # Start of the loop
    print('i = {:2d}, x = {:6.2f}'.format(i, x[i]))    # Formatted output
    i += 1                                             # Increment i by 1
```

If-statements and break: The loop could be terminated after the second execution:

```
for i in range(n):                  # Start of the loop
    print('i = {:2d}, x = {:6.2f}'.format(i, x[i]))     # Formatted output
    if i >= 1:                      # Break the loop if i >= 1.
        break
```

Functions. Let us present an example from physics. Throw a ball straight up in the air. Ignore the air resistance. Let the initial velocity be v_0. The position (height) of the ball and its velocity as a function of time t is given by

$$y(t) = v_0 t - \frac{1}{2} g t^2 \qquad \text{position}$$
$$v(t) = v_0 - gt \qquad \text{velocity}$$

where $g = 9.81\, m/s^2$ is the gravitational constant. The function below takes time, initial velocity and gravitational constant as inputs, and returns the velocity and the position. Notice that the initial velocity and the gravitational constant are given default values.

The function can be used to answer some questions, e.g.:

- What is the velocity and the position after 2 seconds if the initial velocity is 0 m/s?

```
t = 2.0
y1, v1 = ball(t)
print('v0=',0.0,',  t=', t, ',  y=', y1, ',   v=', v1)
```

- What is the velocity and the position after 2 seconds if the initial velocity is 1 m/s?

```
v_start = 1.0
y2, v2 = ball(t, v0=v_start)
print('v0=', v_start,',  t=', t, ',  y=', y2, ',   v=', v2)
```

- What is the velocity and the position after 2 seconds if the initial velocity is 1 m/s and the ball is thrown on the moon ($g = 1.625$
  $, m/s^2$)?

```
y3, v3 = ball(t, v0=v_start, g=1.625)
print('v0=', 1.0,',  t=', t, ',  y=', y3, ',   v=', v3, '  on the moon.')
```

Whenever possible, write your functions such that if the input is an array, so is the output. This is the case here, and it makes it easy to generate a graph of the function, e.g., the position as a function of time when the initial velocity is 10 m/s.

```
t = np.linspace(0,2,101)
v_start = 10
y, v = ball(t, v0=v_start)
plt.plot(t,y)
plt.xlabel('t')
plt.ylabel('y')
plt.title('Position of the ball when v0 = {} m/s'.format(v_start));
```

3.3 Help!

This note provides only a very short introduction to some useful functions and constructions in Python. More will be used throughout the course. We hope that most of it will be understandable or explained along the course, but if there are functions or structures you do not understand, please look them up yourself, for instance by

- Write `?function`.

```
?np.linspace
```

- Or look it up in Google or something similar.

3.4 Further reading

For a nice introduction for scientific computing in Pyhton, we can recommend

- Hans Petter Langtangen: A Primer on Scientific Programming with Python, Springer Verlag, 2016.