LECTURE NOTES | 0200x

Programming in Python

Tue Herlau

`tuhe@dtu.dk`

November 30, 2023

Version 2023.0

Foreword, August 2023

This compendium discuss a few, extra topics that are not discussed in our main textbook, *"Think Python"* [Dow16], but which are nevertheless of importance for engineers.

The notes do not replace *"Think Python"* and should not be read in isolation. We recommend you first complete your weekly reading in *"Think Python"*, and then read the relevant chapters here.

All feedback on the compendium is welcome, and you can send it by email (`tuhe@dtu.dk`) or post it on our Discord forum which you can find on the main website: `https://cp.compute.dtu.dk`.

Throughout the notes we use the following symbol(s):

> ✡ A section which is slightly technical, and where only the general gist of the result is exam relevant.
>
> ✡✡ A section with extra material included for completeness but not relevant for the exam.

Contents

Chapter 1

Plotting

Matplotlib allows you to create graphics. These can be very simple, i.e. 2d plots of black dots, to complex 2 and 3 dimensional plots that involve different colors and text. Although out of scope for this course, you can even make animations in matplotlib. Figure 1.1 shows two examples of complicated matplotlib plots suitable for a report or a scientific paper.

LIBRARY
EXTERNAL PACKAGE

Matplotlib is an example of a **library** or **external package** in python. That means that matplotlib is not available when you just download and install python from `https://python.org`, and the people who develop matplotlib are not the same group of people who develop e.g. the `math`-package in python. However, assuming you follow the installation guide for the course, one of the steps installed matplotlib automatically.

The diverse uses of matplotlib also means it can look quite complex,

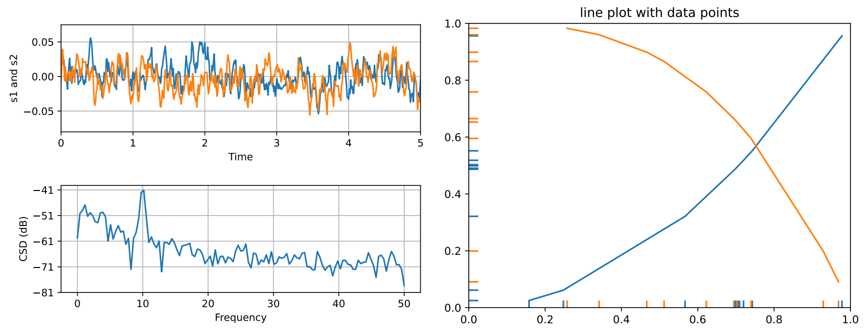
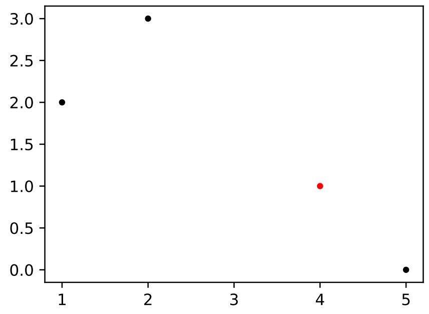


Figure 1.1: Two examples of matplotlib graphs that display some of the capabilities for 2d plots in matplotlib.



```python
# chapter_matplotlib/matplotlib_plots.py
import matplotlib.pyplot as plt
plt.plot(1, 2, 'k.')  # plot a black point at (1, 2)
plt.plot(2, 3, 'k.')  # plot a black point at (2, 3)
plt.plot(4, 1, 'r.')  # plot a red point at (4, 1)
plt.plot(5, 0, 'k.')  # plot a black point at (5, -1)
plt.show()
```

Figure 1.2: Example of a matplotlib plot of 5 points. Notice the arguments `'k.'` and `'r.'` are responsible for the coloring

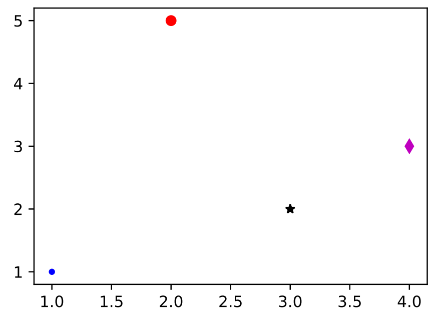
especially if you look at the official documentation. Don't let that discourage you. Most users (including myself) only learn a few things by heart, and then ask google or ChatGPT for help with the rest. My estimate is 90% of all complicated matplotlib graphs contains copy-pasted code.

What we will focus on here are the few bits of matplotlib that you will probably benefit from actually learning by heart. A slight problem is that matplotlib makes use of things in the python language you only learn later, such as lists or numpy (another python library). For that reason the chapter is divided into three parts that will only reference things you know at the time when you need to read them.

1.1 Plotting points

When it comes to displaying data simple is very often better, and the simplest kind of graphics is the humble plot of (x, y) points in a standard 2-axis x/y coordinate system. The code in fig. 1.2 display 5 points, 4 of which are black and one of which are red. To explain the code:

- `import matplotlib.pyplot as plt`: This line imports matplotlib and store the plotting library in a variable called `plt` we can subsequently use. The name can be anything, but `plt` is what everyone uses.
- `plt.plot(1, 2, 'k.')`: This will plot a point at coordinates $(x, y) = (1, 2)$, i.e., the left-most point in the figure. The third argument, `'k.'` is a `str` which determine the style of the point: `'k'` makes the point black and `'.'` makes it a dot.



```python
# chapter_matplotlib/matplotlib_plots.py
import matplotlib.pyplot as plt
plt.plot(1, 1, 'b.')  # blue point
plt.plot(2, 5, 'ro')  # red circle
plt.plot(3, 2, 'k*')  # black star
plt.plot(4, 3, 'md')  # magenta diamond
plt.show()
```

Figure 1.3: Examples of different marker styles and colors.

- `plt.plot(4, 1, 'r.')`: This plot the red point at coordinates $(x, y) = (4, 1)$. The color red is determined by `'r'` in the third argument.
- `plt.show()`: This line is responsible for actually showing the plot in a new window. A common mistake is to omit this line in which case nothing happens.

1.1.1 Point styles and labels

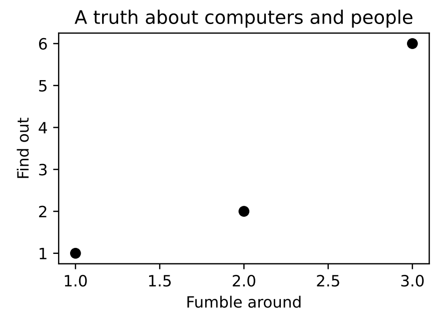
Lets expand a bit on the example by considering some alternative plot styles and axis labels. You can combine different colors and styles by changing the third input argument as the example in fig. 1.3 shows [1], but now we are moving into territory where you should use google whenever you want to do something specific rather than try to memorize tedious details!

1.1.2 Labels, legends and files

The last example shows how you can set axis labels, a plot title, and save your result to a PDF file. We recommend adding such information since it makes it easier for a reader, and you, to figure out what the plot is supposed to show. Try to run this code in fig. 1.4 and see what happens:

The file generated by this code, `my_plot.pdf`, will be saved in the directory from which you called python.

[1]see `https://matplotlib.org/2.1.1/api/_as_gen/matplotlib.pyplot.plot.html` for a fuller list of options to matplotlib

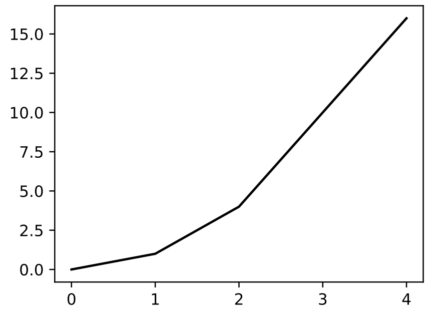


```python
# chapter_matplotlib/matplotlib_plots.py
import matplotlib.pyplot as plt
plt.plot(1, 1, 'ko')
plt.plot(2, 2, 'ko')
plt.plot(3, 6, 'ko')
plt.xlabel('Fumble around')
plt.ylabel('Find out')
plt.title('A truth about computers and people')
plt.savefig("my_plot.pdf") # Save to this file.
plt.show()
```

Figure 1.4: Setting axis labels, adding a title and saving to a file.

1.2 Plotting lines

Plotting lines is a straight forward generalization of plotting points. We still use the format `plt.plot(x, y)`, but in the case where `x` and `y` are lists of x and y coordinates the result will be plotted as a line. Carefully inspect the plot and code in fig. 1.5, and note that the first point is at $(0, 0)$, and the last point is at $(4, 16)$.

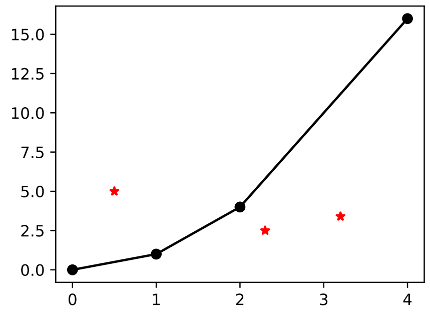


```python
# chapter_matplotlib/matplotlib_plots.py
import matplotlib.pyplot as plt
plt.plot([0, 1, 2, 4], [0, 1, 4, 16], 'k-')
plt.show()
```

Figure 1.5: Setting axis labels, adding a title and saving to a file.

You will also notice that when we plot lines, we get a few more options for styling the plot. The syntax `"k-"` tells matplotlib to color the line black (`"k"`) and plot it as an unbroken line (`"-"`) [2]. There are a wealth of alternative options, however, simple lines and points, as illustrated in fig. 1.6, will by far cover most of your needs.

[2]See `https://matplotlib.org/stable/api/_as_gen/matplotlib.lines.Line2D.html` for other choices, although google is typically much more helpful than the matplotlib documentation

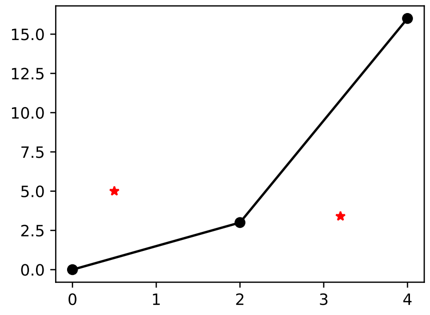


```python
# chapter_matplotlib/matplotlib_plots.py
import matplotlib.pyplot as plt
plt.plot([0, 1, 2, 4], [0, 1, 4, 16], 'ko-')
plt.plot([0.5, 2.3, 3.2], [5, 2.5, 3.4], 'r*')
plt.show()
```

Figure 1.6: Alternative line and point styles. Notice how we can control whether the data series is plotted as a line or a point cloud using a single option `"r*"`.

1.2.1 Legendary legends

Judging by the time spend talking about legends during exams, one would think plot legends are the cornerstone of academic writing. Anyway, they are very easy to add to your plot: You specify the string `label="Legendary Legend"`, and insert the line `plt.legend()` just before you call `plt.show()`, see fig. 1.7.



```python
# chapter_matplotlib/matplotlib_plots.py
import matplotlib.pyplot as plt
plt.plot([0, 2, 4], [0, 3, 16], 'ko-', label="Some stuff")
plt.plot([0.5, 3.2], [5, 3.4], 'r*', label="More stuff")
plt.xlabel("One thing")
plt.ylabel("Another thing")
plt.legend()
plt.show()
```

Figure 1.7: Two data series with inserted legends. The plot has also accrued typical plot crustaceans in the form of axis labels.

1.2.2 Advice I give to students about plotting ✡✡

Your reader will have limited time, including in your bachelor thesis. The plots are the one thing they are most likely to pay attention to, and therefore gives you the best opportunity to present the gist of your results. I try to emphasize the following:

- Decide what point you want to make with your plot and use that to decide on graphical elements: Typically what you want to say is simply that a graph goes up, or one graph is above another. Keep it very simple and use labels, etc. to the best effect.
- Make it clear what is being plotted: Simple is better. Avoid transforming data where possible.
- Simple line and point plots are often the best.
- Use the plot caption as mini-conclusions. For instance, suppose you want the reader to take away that your experiment is better than another because y increase as a function of x. Then simply tell that to the reader: *"We see that y increases as expected,* ***thereby confirming our hypothesis that we frobulated foobar****"*. A lazy reader who skip the rest of the text (i.e., most readers) will now be primed to just accept your conclusion.
- Have a few quality plots rather than 12 junk plots you don't discuss.
- Make your plots early. They will often help you debug and understand whatever you are doing, and serve as useful ways to communicate intermediate results to e.g. your team members, colleagues, or advisor.

Chapter 2

Numerical python

NumPy is considered the standard for handling numerical data in Python and is widely integrated into the scientific Python and PyData ecosystems.

NumPy is used across a range of disciplines for cutting-edge scientific and industrial research. Its API is heavily utilized in popular scientific Python packages like Pandas, SciPy, Matplotlib, scikit-learn, and PyTorch.

At its core, NumPy is used to represent and manipulate vectors and matrices. NumPy also provides a comprehensive set of mathematical functions such as matrix multiplication. Finally, NumPy is extremely fast.

When to use NumPy

Although NumPy can be an extremely useful tool, it does *not* replace similar data data-structures such as `list` or `tuple`. These data-structures remain the better choice for small lists, or when the items are not all numbers, and all other circumstances where you don't *need* NumPy-specific functionality.

You can think about NumPy as an electrical food processor and the regular `list` as a knife and cutting board: There are situations where the food processor excels, but the cutting board is more versatile and faster to use for simple tasks.

- *Use NumPy when:* You are dealing with vectors and matrices and you need matrix-operations such as multiplication.
- *Otherwise:* use a `list`, `tuple` or `dict`. If you feel something that just involves a list of things is simpler in NumPy, ask ChatGPT how it can be done using regular lists.

2.1 Arrays

Let's just jump into it. Since NumPy is a library, we first need to import it to use it as so: `import numpy as np`. This is how we can use NumPy to define an array from a regular python `list`:

```
>>> import numpy as np
>>> a = np.array([0, 1, 2, 3])
>>> a  # This variable now represents the array.
array([0, 1, 2, 3])
```

This array corresponds to the mathematical vector $\mathbf{x}^\top = \begin{bmatrix}0 & 1 & 2 & 3\end{bmatrix}$.

Obviously this is not very efficient, so NumPy contains functions to create constant arrays containing either all zero or all one entries as follows:

```
>>> import numpy as np
>>> np.zeros(4)
array([0., 0., 0., 0.])
>>> np.ones(4)
array([1., 1., 1., 1.])
```

Finally the following two functions are often used. the first create an array of integers and can be though of an numpy analogue to `range(5)`:

```
>>> np.arange(5)
array([0, 1, 2, 3, 4])
```

While this is useful for creating an array of n equidistant element. You call it in the format: `np.linspace(start, end, n)`:

```
>>> np.linspace(0, 10, 5)
array([ 0. ,  2.5,  5. ,  7.5, 10. ])
```

2.1.1 Creating matrices

What most people associate with numpy is matrices, and again numpy offers a range of ways to create matrices or, as it is called in numpy, 2d arrays.

Perhaps the most direct way is to convert a list-of-lists to a matrix:

```
>>> np.array([[0, 1, 2], [5, 6, 7]])
array([[0, 1, 2],
       [5, 6, 7]])
```

But the trick with `np.ones` and `no.zeros` work as well, except now you need to pass in a tuple `(2,3)`:

```
>>> np.zeros((2, 3))
array([[0., 0., 0.],
       [0., 0., 0.]])
```

Footguns Numpy is a bit famous for its uninformative errors. As an example, suppose we accidentally omit the double parenthesis:

```
>>> np.zeros(2, 3)
Traceback (most recent call last):
  File "<console>", line 1, in <module>
TypeError: Cannot interpret '3' as a data type
```

The conclusion is that you can expect some pain every now in a while when you use NumPy, so be careful to incrementally run (and test) your code.

2.1.2 Size and shape

In NumPy, the `size` of an array represents the physical number of elements whereas the `shape` will give you the height and width as a tuple. As an example:

```
>>> a = np.zeros((3, 5))
>>> a.size
15
>>> a.shape
(3, 5)
```

Obviously, the size is just the height multiplied by the width. For 1d arrays, the size behaves as expected, but the shape will return a 1-dimensional tuple:

```
>>> a = np.zeros(4)
>>> a.size
4
>>> a.shape
(4,)
```

2.2 Indexing

Most operations you can do with a python `list` can also be done with an array. For instance, suppose we start out with this array:

```
>>> x = np.array([[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12]])
>>> print(x)
[[ 1  2  3  4]
 [ 5  6  7  8]
 [ 9 10 11 12]]
>>> x[0, 1] # in row, column format
2
>>> x[2, 3] # The bottom-right element.
12
```

We can then access a single element by using square brackets:

```
>>> a = np.zeros((2,3))
>>> a[1,2] = 42
>>> print(a)
[[ 0.  0.  0.]
 [ 0.  0. 42.]]
```

Obviously this also works for regular arrays:

```
>>> a = np.array([2, 4, 6, 7])
>>> a[2]
6
>>> a[1] = 42
>>> print(a)
[ 2 42  6  7]
```

2.2.1 Slicing

Slicing refers to creating arrays from arrays using the `:`-symbol. The most common way to use slicing is to select row or column vectors from 2d arrays as follows:

```
>>> x = np.array([[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12]])
>>> print(x)
[[ 1  2  3  4]
 [ 5  6  7  8]
 [ 9 10 11 12]]
>>> x[0,:]
array([1, 2, 3, 4])
>>> x[:,1]
array([ 2,  6, 10])
```

We can also use this to set an entire row or column as follows:

```
>>> x[:,3] = [-1, -2, -3]
>>> print(x)
[[ 1  2  3 -1]
 [ 5  6  7 -2]
 [ 9 10 11 -3]]
```

2.3 Operations

NumPy of course allows you to add, subtract arrays, and multiply them with constants. Here are some examples that all use 2d arrays:

```
>>> x = np.array([[1, 2], [3, 4]])
>>> y = np.array([[0, -2], [1, -2]])
>>> print(x)
[[1 2]
 [3 4]]
>>> print(y)
[[ 0 -2]
 [ 1 -2]]
>>> x + y
array([[1, 0],
       [4, 2]])
>>> x - y
array([[1, 4],
       [2, 6]])
>>> 2 * x
array([[2, 4],
       [6, 8]])
```

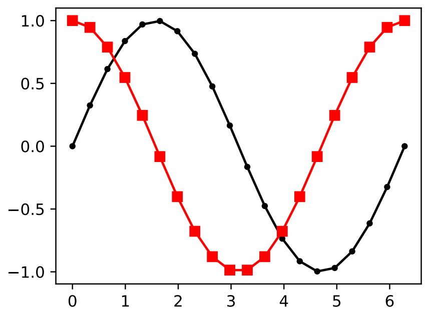
What about matrix-multiplication? The immediate idea, `x * y`, does *not* work:

```
>>> x * y
array([[ 0, -4],
       [ 3, -8]])
```

In fact, it will compute

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \begin{bmatrix} 0 & -2 \\ 1 & -2 \end{bmatrix} = \begin{bmatrix} 1 \times 0 & 2 \times (-2) \\ 3 \times 1 & 4 \times (-2) \end{bmatrix}. \tag{2.1}$$

To multiply two matrices, you need to use the (new!) `@`-symbol:



```python
# chapter_numerical/numerical_plots.py
import matplotlib.pyplot as plt
x = np.linspace(0, 2*np.pi , 20)
y = np.sin(x)
plt.plot(x, y, 'k.-')           # Use plt.plot as usual.
plt.plot(x, np.cos(x), 'rs-')  # Shorter.
plt.show()
```

Figure 2.1: We use Matplotlib to plot two arrays created by NumPy . Note that the Matplotlib syntax is similar to what you have already seen in chapter 1. Use the line style `"k-"` together with a much higher number n of points for smooth curves.

```
>>> x @ y
array([[  2,  -6],
       [  4, -14]])
```

This also works for matrix-vector multiplication. For instance,

```
>>> x @ np.array([1, 2])
array([ 5, 11])
>>> np.array([1, 2]) @ x
array([ 7, 10])
```

will compute

$$\begin{bmatrix}1 & 2\\3 & 4\end{bmatrix}\begin{bmatrix}1\\2\end{bmatrix} = \begin{bmatrix}1+4\\3+8\end{bmatrix} = \begin{bmatrix}5\\11\end{bmatrix}, \quad \begin{bmatrix}1 & 2\end{bmatrix}\begin{bmatrix}1 & 2\\3 & 4\end{bmatrix} = \begin{bmatrix}7 & 10\end{bmatrix}. \tag{2.2}$$

2.4 Operations and plotting

NumPy provides it's own version of the functions in the `math`-module, but which accepts NumPy arrays as inputs. For instance, this is how we can compute the cosine of a small array, which also showcase that NumPy has its own copy of the constant π:

```
>>> x = np.asarray([0, 1, np.pi/2])
>>> print(x)
[0.         1.         1.57079633]
```

```
>>> np.sin(x)
array([0.        , 0.84147098, 1.        ])
```

Similarly, nearly all libraries knows how to interact with numpy. For instance, fig. 2.1 illustrates how you can quickly plot a full period of the sine and cosine function in 20 equidistant locations (note how we use `np.linspace` to obtain the points):

Bibliography

[Dow16] Allen Downey. *Think Python*. O'Reilly Media, Sebastopol, CA, 2nd edition, updated for python 3 edition, 2016. Available at `https://greenteapress.com/wp/think-python-2e/`.