Matlab to Python handout

For

Python for Machine Learning and Data Science

Mechatronic Engineering Program, AGH, University of Krakow,

Mateusz Heesch, Adam Machynia, Ziemowit Dworakowski

Spis treści

Introduction

Welcome to this introductory handout that aims to shed light on the key differences between MATLAB and Python – two widely used programming languages in various scientific and engineering fields. While this handout focuses on syntax differences, allowing you to easily move from matlab to python, the concept and general differences between these two languages are more complex and can be briefly characterized as follows:

- **Purpose and Domain**: MATLAB was designed primarily for numerical computing and is renowned for its powerful built-in functions and toolboxes tailored to engineering, mathematics, and science. Python, on the other hand, is a versatile, general-purpose language with extensive libraries and frameworks that make it suitable for a wide range of applications, from web development to data analysis and scientific computing.

- **Syntax**: MATLAB's syntax is highly specialized for mathematical and matrix operations, often resembling mathematical notation. Python employs a more intuitive and general-purpose syntax, making it easier to learn and read for beginners.

- **Cost and Licensing**: MATLAB is a commercial software package with licensing fees, which can be a barrier for students and researchers on tight budgets. Python, being open source, is free to use, making it accessible to a broader audience.

- **Community and Ecosystem**: Python has a vast and active user community, resulting in a rich ecosystem of packages and libraries. MATLAB also has a substantial user base, but its ecosystem is more tailored to specific domains.

- **Integration**: Python seamlessly integrates with other programming languages, databases, and tools, making it an excellent choice for data science and web development. MATLAB offers integration but may not be as versatile in this regard.

- **Learning Curve**: For someone just starting Python is more user-friendly. MATLAB has a steeper learning curve due to its specialized nature, but its extensive resources can aid the learning process. In your case, you already know basics of MATLAB thus python may feel strange at first. Still, the general concept of a language is very similar and easy to transition to.

- **Performance**: While MATLAB is optimized for numerical computations and can be faster for certain tasks, Python offers various libraries, like NumPy and SciPy, to bridge the performance gap, especially when combined with optimized numerical libraries.

Note that this copy is prepared for download, printing and general use. There is also another editable version of this document. If you would like to suggest changes, make comments or ask for clarifications, please use the following link:

https://docs.google.com/document/d/10Bo_PsFke1VY2l06-67ThHrQQYm_L-Kr/edit?usp=sharing&ouid=108134038542576340173&rtpof=true&sd=true

Python documentation is available here: https://docs.python.org/3/

Differences between matlab and python / basic syntax

Indexing from 0

As opposed to matlab, indexing anything starts at 0, not 1. So to access the first element of a vector named *abc* we would call abc[0], for second abc[1], etc.

Importing libraries

As opposed to matlab where all installed packages are included in scripts we write, in Python only the basic functions will be automatically included. If we want to utilize some specific libraries, we'll have to import them BEFORE USING THEM, which looks as follows:

```
import os
```

having done that, we can now utilize functions from *os* library, e.g. *os.listdir(dir_path)* which lists contents of directory

We can also only import specific functions from a library, going off above example:

```
from os import listdir
```

in which case we'll only import the *listdir* function, and can access it directly - as

```
listdir(dir_path)
```

Lastly, we can change the names of modules we import, especially useful for ones with long names, or those that we'll be calling multiple times. E.g. Let's say we want to create a plot of values in array1.
We could do it this way:

```
import matplotlib.pyplot
matplotlib.pyplot.plot(array1)
```

Or:

```
import matplotlib.pyplot as plt
plt.plot(array1)
```

Important note - there are a few “common” import naming conventions for the libraries we'll be using (and you'll see them in most tutorials and open source code using these libraries too):

```
numpy as np
pandas as pd
matplotlib.pyplot as plt
```

Meaning of various brackets

Collections

Various brackets are used to define basic python collections (data structures):
[x, y, z] creates a list with x y z variables, lists are effectively 1d arrays (though for calculations on them, we'll refer to numpy)
(x, y, z) creates a tuple with x y z variables. Tuples function similarly to lists, with the exception of being immutable (and in return more memory efficient)
{'key1': value1, 'key2': value2} creates a dictionary with two key: value pairs. Dictionaries function similarly to matlab structs, although they do use different indexing
further reading: https://docs.python.org/3/tutorial/datastructures.html
for more complex collections, refer to https://docs.python.org/3/library/collections.html

Functions/indexing

indexing is ALWAYS done using square brackets, be it tuple, string, list, dictionary, or ndarray. To access n-th element of list array1, we'll call *array1[n]* , to access key *"abc"* of dictionary *dict1* we'll call *dict1["abc"]*
similarly to matlab, functions are ALWAYS called using round brackets, e.g. *function1(x, y, z)*
accessing parameters of objects is done via . - e.g. to access shape of numpy ndarray called *array1*, we'll call *array1.shape* - similarly to class methods, e.g. *array1.reshape(new_shape)*

Importance of indentations (tab/4x space)

In python, quadruple space (usually done by pressing "tab") is used to denote code "hierarchy", as opposed to "begin" and "end" in matlab, or curly brackets in C / C++. Example illustrating this would be:

```
for i in range(10):
    do_something() # in loop
    do_something_else() # in loop
do_something_else_2() # not in loop
```

"For loop" syntax

The **for** statement is used to iterate over the elements of a sequence (such as a string, tuple or list) or other iterable object.

```
for i in iterable:          #colon
    # do sth                #indented
# do sth outside the loop       #the loop is finished
```

e.g.

```
for i in range(10):
    print(i)
```

Note: colon and indent.
https://docs.python.org/3/tutorial/controlflow.html#for-statements

The built-in function range() is often used with the for statement. It returns an immutable sequence of numbers.

```
range(stop)
range(start, stop[, step])
```

start The value of the start parameter (or 0 if the parameter was not supplied)
stop The value of the stop parameter (this value is the first not included in resulting set)
step The value of the step parameter (or 1 if the parameter was not supplied)
e.g. `list(range(5)) [0,1,2,3,4]`
For a positive step, the contents of a range r are determined by the formula r[i] = start + step*i where i >= 0 and r[i] < stop.
For a negative step, the contents of the range are still determined by the formula r[i] = start + step*i, but the constraints are i >= 0 and r[i] > stop.

Function syntax

Function syntax starts with “def”, followed by function name, function arguments in brackets, and then lastly a colon. The function *usually* ends with the “return” line, which decides what value is returned when this function is called in code, however not all functions *need* to return something.

```
def sum_values(a, b=0):
    c = a + b
    return c
```

You can assign a default value to an argument, allowing further uses of the function to omit it if changing it is not necessary, such as the “b” in example above.

Class syntax

Creating a class

```
from math import sqrt

class TwoDCar:
    def __init__(self, color='red'):
        self.position = [0, 0]
        self.distance_traveled = 0
        self.color = color
    def move2D(self, move_vector):
        self.position[0] = self.position[0] + move_vector[0]
        self.position[1] = self.position[1] + move_vector[1]
        dist = sqrt(move_vector[0]**2 + move_vector[1]**2)
        self.distance_traveled = self.distance_traveled + dist
```

Classes are “template” definitions which contain some data (parameters) and are able to do some things (methods). Class should always have the “__init__(self)” function which is the constructor - the function called when the class is instantiated (see next section). Besides the constructor, a class can have any number of methods depending on the needs. In general all methods should have “self” as the first argument (not entirely true, but for simplicity’s sake let’s stick to that), which denotes the object itself, and is used to access its parameters and methods.

Creating a class instance (object)

```
car_default = TwoDCar()
car_blue = TwoDCar(color='blue') # overwrites default value of 'color' with
'blue'
```

Interacting with an object

```
car_blue.move2D([10, 0]) # will call the move2D method of the object
car_blue.move2D([5, 5])

print(car_blue.position) # will access the position parameter, with expected
value of [15, 5]
print(car_blue.distance_traveled) # will access distance_traveled parameter,
with expected value of ~17.07
car_blue.distance_traveled = 0 # will reset the value of distance_traveled
parameter
```

Python libraries relevant for data science / machine learning

numpy

NumPy array

Base class of numpy library is **numpy.ndarray** - which stands for n-dimensional array. It allows us to handle arrays with arbitrary number of dimensions, and perform various mathematical operations on them. Contrary to the well-known “python is slow” fact, numpy library operates on C bindings, making it very fast if used properly.

Broadly speaking numpy arrays are very similar to matlab arrays, and will likely be the most commonly used data format during this course (next to pandas dataframes)

Array indexing

“The elements [of an array] are all of the same type, referred to as the array dtype. An array can be indexed by a tuple of nonnegative integers, by booleans, by another array, or by integers. The rank of the array is the number of dimensions. The shape of the array is a tuple of integers giving the size of the array along each dimension.” (from *intro*)

1-D or 2-D arrays, called vector and matrix, are also represented by ndarray.

Creation

From list

```
a = np.array([1, 2, 3]) # 1-D array
b = np.array([[1, 2, 3], [4, 5, 6]]) # 2-D array
```

Create array of zeros

```
z = np.zeros((2, 3)) # first argument is shape - int or tuple
```

Create array of ones

```
o = np.ones((2, 3)) # first argument is shape - int or tuple
```

Create an empty array

```
e = np.empty((2, 3))
```

Its initial content is random and depends on the state of the memory

See also: np.arange(), np.linspace().

Operations on arrays’ elements

Indexing

Ndarray indexing is similar to python lists indexing.

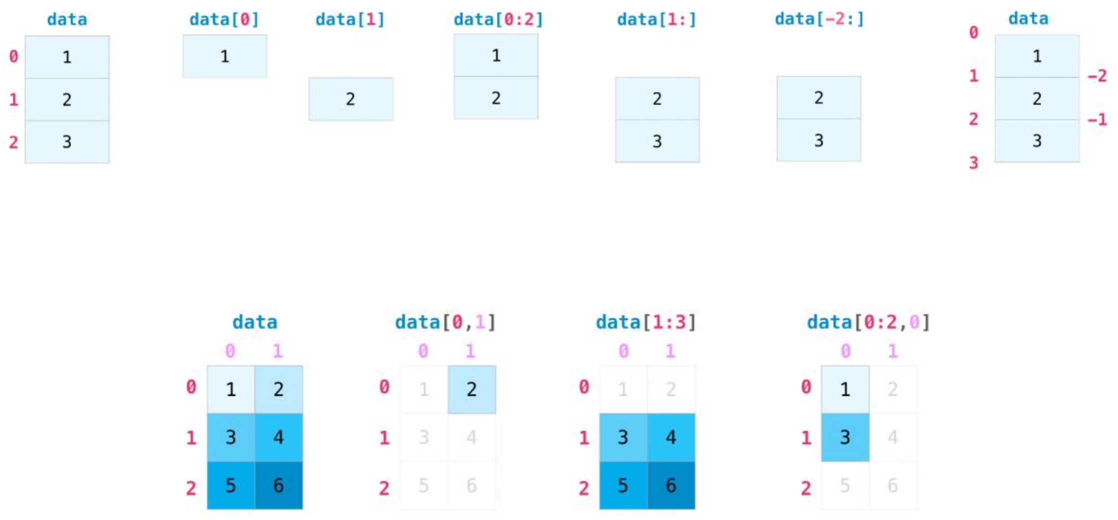


Fig. Indexing example (https://numpy.org/doc/stable/user/absolute_beginners.html)

NumPy arrays might also be indexed using logical conditions and boolean arrays.

```
b = np.array([[1, 2, 3], [4, 5, 6]])
print(b[b>4])
[5 6]
```

Operations on arrays

Stack arrays vertically: np.vstack()

Stack arrays horizontally: np.hstack()

```
a = np.array([1, 2])
b = np.array([3, 4])
hs = np.hstack((a,b))
vs = np.vstack((a,b))
print(hs)
[1 2 3 4]
print(vs)
[[1 2]
 [3 4]]
```

Arithmetic operators like +, -,* , / are applied **elementwise.**

For matrix product use @ operator or dot method.

```
a = np.array([[1, 2], [3,4]])
b = np.array([[1, 1], [2, 2]])
print(a*b) # elementwise multiplication
[[1 2]
 [6 8]]
print(a@b) # matrix multiplication
```

```
[[ 5  5]
 [11 11]]
print(a.dot(b)) # matrix multiplication
[[ 5  5]
 [11 11]]
```

Getting minimum, maximum, sum, mean or standard deviation of array elements.

```
a = np.array([[1, 2], [3,4]])
a.min() # or: np.min(a)
1
a.max()
4
a.sum()
10
a.mean()
2.5
a.std()
1.118033988749895
```

Copy/view of an array

Note the difference between copying an array and referring by creating its view. In the latter case the same memory buffer is read/modified. Some methods work on copies, some on views, but it can be forced by adequate methods: *ndarray.copy()* and *ndarray.view()*.
Example of a view – both arrays are modified

```
a = np.array([[1, 2], [3,4]])
v=a
v[1,1]=0
print(v)
[[1 2]
 [3 0]]
print(a)
[[1 2]
 [3 0]]
```

Example of a copy – only the copy is modified

```
a = np.array([[1, 2], [3,4]])
c=a.copy()
c[1,1]=0
print(c)
[[1 2]
 [3 0]]
print(a)
```

```
[[1 2]
 [3 4]]
```

Saving/loading ndarray
np.save – save ndarray as .npy file
np.savez – save more than one ndarray object in a single file, as a .npz file
np.load – load .npy or .npz file

```
np.save('name', a)
b = np.load('name.npy')
```

The most important NumPy vs Matlab differences

	NumPy	**Matlab**
Indexing	0	1
Arithmetic operators	Elementwise	Matrix operations
Copy an array	y = x.copy()	y=x
Get slice of an array	y = x[1, :].copy()	y=x(2,:)
Short-circuiting logical operators	and or	&& ||
Logical operators	logical_and/logical_or	& and |
Bitwise operators	& and |	bitand/bitor

Intro: https://numpy.org/doc/stable/user/absolute_beginners.html
Quickstart: https://numpy.org/doc/stable/user/quickstart.html
Documentation: https://numpy.org/doc/stable/user/index.html
https://numpy.org/doc/stable/reference/index.html
NumPy vs Matlab: https://numpy.org/doc/stable/user/numpy-for-matlab-users.html

matplotlib

Matplotlib is a library for creating visualizations, especially plots, in Python. Generally, it works similarly to Matlab.
Figures and subplots
Use *plt.figure()* to create a new figure. As in Matlab, figures can be uniquely numbered: *plt.figure(3).*
To create figure and axes following functions can be used:

- *plt.subplots()* – figure with a single axes,
- *plt.subplots(2,2)* – figure with a grid of axes,
- *plt.subplot_mosaic(args)* – figure with a mosaic of axes.

```
import matplotlib.pyplot as plt # import pyplot
fig = plt.figure()  # an empty figure with no Axes
fig, ax = plt.subplots()  # a figure with a single Axes
```

```
fig, axs = plt.subplots(2, 2)  # a figure with a 2x2 grid of Axes
# a figure with one axes on the left, and two on the right:
fig, axs = plt.subplot_mosaic([['left', 'right_top'],
                               ['left', 'right_bottom']])
```

In Matplotlib there are two APIs:

- an explicit "Axes" ("object-oriented") interface: after creating an axes or a figure functions are called on them e.g. ax.plot(),
- an implicit "pyplot" interface: uses pyplot functions for plotting, keeping track of last figure or axes e.g. plt.plot().

See: Matplotlib Application Interfaces (APIs)

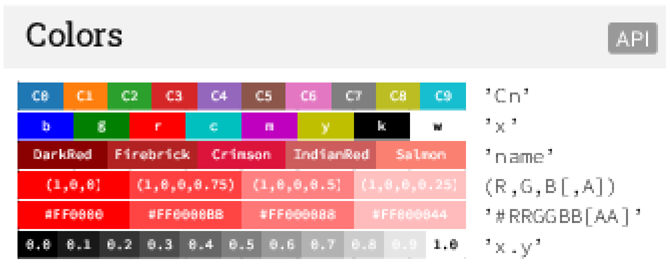
Data for plotting should be in the form of a *numpy.array.*

Basic plot types

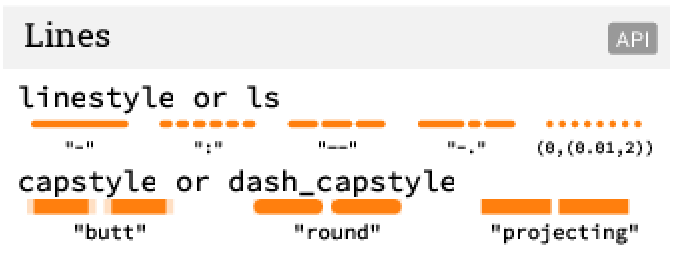
plot(x, y) – plot y versus x as lines and/or markers
scatter(x, y) – a scatter plot
imshow(X) – display data as an image
All plot types: https://matplotlib.org/stable/plot_types/index.html

Plot style can be customized by optional arguments (keyword properties) in plot, scatter and so on methods. See an example below.
Some properties for 2D lines are:

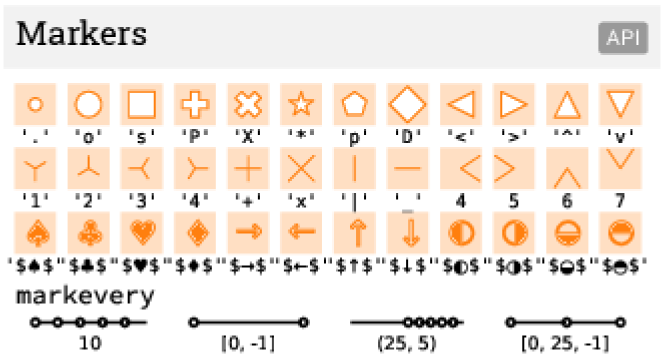
- color or c



- linestyle or ls



- linewidth or lw: float number,
- marker



- markersize or ms: float number.

Unlike in notebooks, while creating standalone script calling *plt.show()* function is necessary to make figures visible.

To save a figure use: *fig.savefig(‘name.png').*

Simple example

```
import matplotlib.pyplot as plt
import numpy as np

x = np.linspace(0, 2 * np.pi, 200)
y = np.sin(x)
y2 = np.cos(x)

# object oriented style
fig, ax = plt.subplots()
ax.plot(x, y, color='orange', linestyle='--', label='sine')
ax.plot(x, y2, color=[0, 0.5, 1], marker='*', label='cosine')
ax.legend()
ax.set_xlabel('x')
ax.set_ylabel('y')
ax.set_title('an example of a plot')

# pyplot style
plt.figure(2)
plt.scatter(x,y, marker='s', s=5)
plt.xlabel('x')
plt.ylabel('y')
plt.title('scatter plot')

plt.show()
```

Matplotlib cheatsheets: https://matplotlib.org/cheatsheets/

https://matplotlib.org

pandas

Pandas is a data manipulation and analysis library for Python and will be introduced further during the course.
https://pandas.pydata.org/

sklearn

Scikit-learn is a machine learning library for Python and will be introduced further during the course.
https://scikit-learn.org/stable/

seaborn

Seaborn is a data visualization library based on matplotlib. It grants more functionalities. Seaborn is focused on statistical graphics and operates on pandas data structures (dataframes).
https://seaborn.pydata.org

Setting up python locally (very optional)

Though during the classes we'll be working on google colab with everything mostly set up, you can also set up python locally on your machine (bearing in mind it's in no way necessary for the class, it may however be more convenient to some people). Recommended set-up for ease of use would be to use conda:

https://conda.io/projects/conda/en/latest/user-guide/install/index.html
with spyder:
https://anaconda.org/anaconda/spyder
https://docs.spyder-ide.org/current/installation.html
or regular jupyter notebooks:
https://anaconda.org/anaconda/jupyter
https://docs.anaconda.com/ae-notebooks/user-guide/basic-tasks/apps/jupyter/index.html