# Lecture#3

May 17, 2023

## 1 Python for Signal Processing

Danilo Greco, PhD - danilo.greco@uniparthenope.it - University of Naples Parthenope

#### 1.0.1 Lecture 3

This lecture will provide an overview on Numpy mathematical library:

1. installation,
2. documentation,
3. main functions and applications,
4. practical examples.

#### 1.0.2 What is Numpy?

Numpy is a fundamental library for scientific computing in Python.

It provides multidimensional array objects and an assortment of routines for fast operations on arrays, including

- mathematical,
- logical,
- shape manipulation,
- sorting,
- selecting,
- I/O,
- discrete Fourier transforms,
- basic linear algebra,
- basic statistical operations,
- random simulation

and much more.

Numpy is based on the ndarray object which encapsulates n-dimensional arrays of homogeneous data types.

**Differences from Python lists**

- ndarrays have fixed size, unlike lists which can grow dynamically
- ndarrays require elements of the same data type, unless for arrays of objects where each object can be different
- efficient execution of operations on large numbers of data

- lots of python packages rely on ndarrays rather than lists
- element-wise operations on ndarrays are faster than iterating over the list elements, thus avoiding useless loops

The implicit element-by-element behavior of operations in numpy is termed broadcasting.

## 2 Install

It can be installed by executing the following command:

> pip install numpy

or

> python -m pip install numpy

**NOTE: when installing with python version 3.x, replace pip or python with pip3 or python3 in the commands above.** Alternative installation can be done by installing many other scientific libraries that require numpy, such as Scipy, so that it get implicitly installed.

## 3 Documentation

The official numpy documentation is available on numpy website and provides an extensive guide for both users and developers, including setup and absolute beginners tutorials.

There is also an interesting book explaining how to use numpy for data analysis: >Wes McKinney, Python for Data Analysis: Data Wrangling with Pandas, NumPy, and IPython, O'Reilly Media

## 4 Main functions and applications

The first thing to do is inform the python interpreter that we are using the package. The following command tells python to import the library and use an alias for quicker references within our code:

[ ]:
```
import numpy as np
```

An array is a grid of values and it contains information about the raw data, how to locate an element, and how to interpret an element. All elements have the same data type named dtype.

An array can be created starting from lists or nested lists and its elements are located by a positional index starting at zero:

[ ]:
```
a = np.array([1,2,3,4,5]) # from a list
print(a[2])

m = np.array([[1,2,3],[4,5,6],[7,8,9]])  # from a nested list
print(m[0])
```

Other ways for creating arrays make use of the following functions:

```
[ ]: # create an array with all zeroes
     np.zeros(3)  # specifying the number of elements in the array
```

```
[ ]: # create an array with all ones
     np.ones((2,3))    # passing the desired shape, that is a tuple with rows and␣
      ↪columns for the array
```

```
[ ]: # create an empty array
     np.empty(3)   # values in the array may vary
```

```
[ ]: # create an array with incremental values
     a = np.arange(5)
     print(a)
     # specify start, stop and step size
     np.arange(0.2, 1.0, 0.2)
```

```
[15]: # create arrays of equally spaced values
      # requires start, stop and count
      np.linspace(0, 10, num=5)
```

```
[15]: array([ 0. ,  2.5,  5. ,  7.5, 10. ])
```

Sorting the values of an array is pretty simple by using:

```
[16]: a = np.array([3,4,2,1,2])
      print(np.sort(a))
```

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

and it's also possibile to obtain the sorted position of initial elements:

```
[17]: np.argsort(a)
```

```
[17]: array([3, 2, 4, 0, 1], dtype=int64)
```

To inquire the position of a searched element in an array, we can use:

```
[18]: np.argwhere(a==2)
```

```
[18]: array([[2],
             [4]], dtype=int64)
```

```
[19]: a = np.array([[1, 2],[3, 4]])
      b = np.array([[5, 6],[7, 8]])

      # concatenate arrays by rows
      np.concatenate((a, b),axis=0)
```

```
[19]: array([[1, 2],
             [3, 4],
        [5, 6],
        [7, 8]])
```

[20]:
```python
# concatenate arrays by columns
c = np.concatenate((a, b),axis=1)
print(c)
```

```
[[1 2 5 6]
 [3 4 7 8]]
```

To inquire information on arrays, we can use the following:

[21]:
```python
print("Dimensions: {}, Total size: {} elements, Shape: {}".format(c.ndim,c.
 ↪size,c.shape))
```

```
Dimensions: 2, Total size: 8 elements, Shape: (2, 4)
```

Arrays can also be reshaped in a very simple way:

[22]:
```python
c.reshape((4,2))
```

[22]:
```
array([[1, 2],
       [5, 6],
       [3, 4],
       [7, 8]])
```

**Note that these commands do not alter the original arrays: if you want to keep the transformation you must assign it to a variable**

[23]:
```python
d = c.reshape((4,2))
print(c)
print(d)
```

```
[[1 2 5 6]
 [3 4 7 8]]
[[1 2]
 [5 6]
 [3 4]
 [7 8]]
```

[24]:
```python
print(c[0]) # displays row 0 of array
print(d[:,1]) # displays all elements of column 1
```

```
[1 2 5 6]
[2 6 4 8]
```

It is possible to transform a 1D array into a 2D array:

[25]:
```python
a = np.array([3,4,1,2])
a
```

[25]:
```
array([3, 4, 1, 2])
```

[26]:
```
a_row = a[np.newaxis,:]
print(a_row)
print(a_row.shape)
```

```
[[3 4 1 2]]
(1, 4)
```

[27]:
```
a_col = a[:,np.newaxis]
print(a_col)
print(a_col.shape)
```

```
[[3]
 [4]
 [1]
 [2]]
(4, 1)
```

Indexing and slicing with numpy arrays is the same as python lists:

[28]:
```
print(a)
print("a[1]= {}".format(a[1]))
print("a[1:3]= {}".format(a[1:3]))
print("a[-2:]= {}".format(a[-2:]))
```

```
[3 4 1 2]
a[1]= 4
a[1:3]= [4 1]
a[-2:]= [1 2]
```

Selection of elements that fulfill a given condition can be done as follows:

[29]:
```
a = np.arange(8)
print(a)

print(a[a < 4])  # print all elements lower than 4
print(a<4)
print(a[a%2 == 0])  # print all elements divisible by two
print(a%2 == 0)
```

```
[0 1 2 3 4 5 6 7]
[0 1 2 3]
[ True  True  True  True False False False False]
[0 2 4 6]
[ True False  True False  True False  True False]
```

Multiple conditions can be specified with the logical operators & and |:

[30]:
```
print(a[(a < 4) & (a%2 == 0)])
print(a[(a < 4) | (a%2 == 0)])
```

```
[0 2]
[0 1 2 3 4 6]
```

The conditions specified within the square brackets generate a boolean mask that is used when indexing the array:

[ ]:
```
mask = (a < 4) & (a%2 == 0)
print(mask)
print(a[mask])
```

Element-by-element operations are the default behavior for ndarrays:

[ ]:
```
print(c)
print(c[0]+c[1])
print(c[0]-c[1])
print(c[0]*c[1])
print(c[0]/c[1])

# we can multiply a vector by a scalar
print(1.6*c[0])
```

We can sum the elements or find maximum or minimum values of an array by rows or by columns depending on the axis parameter:

[ ]:
```
c.sum(axis=0) # sum the rows in each column
```

[ ]:
```
c.sum(axis=1) # sum the columns in each row
```

[ ]:
```
c.min(axis=0) # find the minimum on each column
```

[36]:
```
c.max(axis=1) # find the maximum on each row
```

[36]:
```
array([6, 8])
```

whereas if no axis is specified the result is across the whole array:

[37]:
```
c.sum()
```

[37]:
```
36
```

[38]:
```
c.max(),c.min()
```

[38]:
```
(8, 1)
```

Operator * performs element-wise vector product, whereas matrix multiplication can be computer with the @ operator, as of:

[39]:
```
m1 = np.array([[3,1],
               [8,2]])
m2 = np.array([[6,1],
               [7,9]])
```

```
print('m1 =')
print(m1)
print('m2 =')
print(m2)
print(m1 * m2)
print(m1 @ m2)
```

```
m1 =
[[3 1]
 [8 2]]
m2 =
[[6 1]
 [7 9]]
[[18  1]
 [56 18]]
[[25 12]
 [62 26]]
```

[40]:
```
%%time
m1 @ m2
```

```
CPU times: total: 0 ns
Wall time: 0 ns
```

[40]:
```
array([[25, 12],
       [62, 26]])
```

Let's see how all elements of m1 @ m2 are calculated:

[41]:
```
print(np.dot(m1[0,:],m2[:,0]))
print(np.dot(m1[1,:],m2[:,0]))
print(np.dot(m1[0,:],m2[:,1]))
print(np.dot(m1[1,:],m2[:,1]))
```

```
25
62
12
26
```

[42]:
```
%%time
np.array([[np.dot(m1[0,:],m2[:,0]),np.dot(m1[0,:],m2[:,1])],[np.dot(m1[1,:],m2[:
 ↪,0]),np.dot(m1[1,:],m2[:,1])]])
```

```
CPU times: total: 0 ns
Wall time: 0 ns
```

[42]:
```
array([[25, 12],
       [62, 26]])
```

[43]:
```
mat_prod = np.array([[np.dot(m1[0,:],m2[:,0]),np.dot(m1[0,:],m2[:,1])],[np.
 ↪dot(m1[1,:],m2[:,0]),np.dot(m1[1,:],m2[:,1])]])
mat_prod
```

[43]:
```
array([[25, 12],
       [62, 26]])
```

[44]:
```
dim = 5
l = np.zeros(dim)
for i in range(dim):
    l[i] = i*i

l
```

[44]:
```
array([ 0.,  1.,  4.,  9., 16.])
```

[ ]:
```
loop = True
l = list()
while loop:
    value = float(input("Give me a number: "))
    if value == 0.0:
        loop = False
    else:
        l.append(value)

al = np.array(l)
```

[ ]:
```
l,al
```

[ ]:
```
type(l),type(al)
```

The inverse of a matrix $A$ is a matrix $A^{-1}$ such that $A \cdot A^{-1} = I$.

Computing the inverse is easily done in numpy with np.linalg.inv and it can be very useful in solving linear equations.

[ ]:
```
m1i = np.linalg.inv(m1)
print(m1)
print(m1i)
print(m1 @ m1i)
```

[ ]:
```
np.eye(4)
```

Numpy allows to generate random vectors with a desired shape and float values in the half-open interval [0.0, 1.0):

[ ]:
```
np.random.random((3,2,4))
```

```
[ ]: np.random.randint(4, size=(3,2)) # generates 3 by 2 random integer values lower␣
     ↪than 4
```

It is often necessary to obtain a list of elements with duplicates removed and some additional information about element count and positions. This can be achieved by the np.unique function:

```
[ ]: r = np.random.randint(4,size=10)
     r
```

```
[ ]: np.unique(r)
```

```
[ ]: values, indexes = np.unique(r,return_index=True)
```

```
[ ]: values,indexes
```

```
[ ]: def retmulti(a,b):
         return a,b

     ra, rb = retmulti(3,4)
     print(ra,rb)
     ra, _ = retmulti(3.8,6.7)
     print(ra,rb)
```

The given Python code demonstrates the use of the zip() function to iterate over multiple lists simultaneously. Let's break down what the code does step by step: Three lists, x, y, and z, are initialized with some values:

```
[ ]: x = [0, 1, 2]
     y = [3, 4, 5]
     z = [6, 7, 8]
```

The zip() function is then used to combine the three lists element-wise. It takes the corresponding elements from each list and returns an iterator of tuples, where each tuple contains the corresponding elements from all the lists. In this case, the resulting iterator will yield tuples (0, 3, 6), (1, 4, 7), and (2, 5, 8).

A for loop is used to iterate over the tuples obtained from zip(). The loop assigns the values from each tuple to the variables k, l, and m:

```
[ ]: for k, l, m in zip(x, y, z):
         print(k, l, m)
```

```
[ ]: x = [0,1,2]
     y = [3,4,5]
     z = [6,7,8]
     for k,l,m in zip(x,y,z):
         print(k,l,m)
```

```
[ ]: for v, i in zip(values,indexes):
         print('First occurence of {} is position {}'.format(v,i))
```

```
[ ]: values, counts = np.unique(r,return_counts=True)
     for v, i in zip(values,counts):
         print('Number of occurences of {} is {}'.format(v,i))
```

Array transposition can be done in two possible ways: * using the .T property of the class * using the .transpose() method which allows more complex transformations on the axes.

In the vast majority of cases, the property is the required option.

```
[ ]: print(c)  # the original matrix
     print(c.T)  # using the transpose property of the class
     print(c.transpose())  # using the transpose method of the class
```

## 5 An interesting example on mathematical formulas

If we need to calculate the MSE between our predicted data and the real ones, we should apply:

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(predVal_i - realVal_i)^2$$

While in most languages this formula would require an iterative loop to solve, it gets quite easy in numpy and can be done in at least two ways that do not imply explicit iterations:

```
[ ]: predVal = np.random.random(5)
     realVal = np.random.random(5)

     print('predVal {}'.format(predVal))
     print('realVal {}'.format(realVal))

     %timeit 1/predVal.shape[0]*np.sum(np.square(predVal-realVal)) # with the sum of␣
      ↪squares
     MSE = 1/predVal.shape[0]*np.sum(np.square(predVal-realVal)) # with the sum of␣
      ↪squares

     print('MSE SS= {:.3f}'.format(MSE))

     %timeit 1/predVal.shape[0]*np.dot(predVal-realVal,predVal-realVal) # using dot␣
      ↪product
     MSE = 1/predVal.shape[0]*np.dot(predVal-realVal,predVal-realVal) # using dot␣
      ↪product

     print('MSE DP= {:.3f}'.format(MSE))
```

Data can be loaded and saved with numpy specific routines but it isn't worth going into these functions because we will be learning a better way of dealing with data input with the Pandas library.

### 5.0.1 Basic plotting example

This example gives only a quick overview on how to plot data with numpy support, bearing in mind that a detailed description on data plotting will be given in lecture 4.

[ ]:
```
import numpy as np
import matplotlib.pyplot as plt

# Compute the x and y coordinates for points on a sine curve
x = np.arange(0, 4 * np.pi, 0.1)
y = np.sin(x)
x,y
```

[ ]:
```
# Plot the points using matplotlib
plt.plot(x, y)
plt.show()
```

[ ]:
```
```