Appendix 2: Mathematical operations

NumPy (https://numpy.org/) is a first-rate library for numerical programming. It is widely used in academia, finance and also in the industry.

The *Pandas* library introduced in Chapter 9 (09_tabular.pdf) is also built on top of *NumPy*, providing high-performance, easy-to-use data structures and data analysis tools, making data manipulation and visualization more convinient.

How to install numpy?

If you have Anaconda installed, then numpy was already installed together with it.

If you have a standalone Python3 and Jupyter Notebook installation, open a command prompt / terminal and type in:

```
pip3 install numpy
```

How to use numpy?

The numpy package is a module which you can simply import. It is usually aliased with the `np` abbreviation:

```
import numpy as np
```

NumPy Arrays

The most important structure that NumPy defines is an array data type formally called a `numpy.ndarray` (https://docs.scipy.org/doc/numpy/reference/arrays.ndarray.html) - for *N dimensional array*.

In [1]:

```
import numpy as np

a = np.zeros(3)
a
```

Out[1]:

```
array([0., 0., 0.])
```

In [2]:

```
type(a)
```

Out[2]:

```
numpy.ndarray
```

NumPy arrays are somewhat like native Python lists, except that:

- data must be homogeneous (all elements of the same type);
- these types must be one of the data types (dtypes) provided by NumPy.;

The most important of these dtypes are:

- `float64` : 64 bit floating-point number
- `int64` : 64 bit integer
- `bool` : 8 bit True or False There are also dtypes to represent complex numbers, unsigned integers, etc.

The default dtype for arrays is `float64` :

In [3]:

```
a = np.zeros(3)
type(a[0])
```

Out[3]:

```
numpy.float64
```

If we want to use integers we can specify it:

In [4]:

```
a = np.zeros(3, dtype=int)
type(a[0])
```

Out[4]:

```
numpy.int64
```

Shape and Dimension

Here `b` is a flat array with no dimension - neither row nor column vector.

The dimension is recorded in the `shape` attribute, which is a tuple.

In [5]:

```
b = np.zeros(10)
b.shape
```

Out[5]:

```
(10,)
```

To give it dimension, we can change the `shape` attribute:

In [6]:

```
b.shape = (10, 1)
b
```

Out[6]:

```
array([[0.],
       [0.],
       [0.],
       [0.],
       [0.],
       [0.],
       [0.],
       [0.],
       [0.],
       [0.]])
```

Make it a 2 by 2 array:

In [7]:

```
b = np.zeros(4)
b.shape = (2, 2)
b
```

Out[7]:

```
array([[0., 0.],
       [0., 0.]])
```

Dimension can also be specified initially when using the `np.zeros()` function.

In [8]:

```
b = np.zeros((2, 2))
b
```

Out[8]:

```
array([[0., 0.],
       [0., 0.]])
```

You can probably guess what `np.ones` creates.

In [9]:

```
b = np.ones(10)
b
```

Out[9]:

```
array([1., 1., 1., 1., 1., 1., 1., 1., 1., 1.])
```

Creating Arrays

We have already discussed `np.zeros()` and `np.ones()` .

Set up a grid of evenly spaced numbers.

In [10]:

```
b = np.linspace(2, 4, 5)
b
```

Out[10]:

```
array([2. , 2.5, 3. , 3.5, 4. ])
```

Create an identity matrix.

In [11]:

```
b = np.identity(3)
b
```

Out[11]:

```
array([[1., 0., 0.],
       [0., 1., 0.],
       [0., 0., 1.]])
```

NumPy arrays can be created from Python lists, tuples, etc.

In [12]:

```
b = np.array([10, 20])
b
```

Out[12]:

```
array([10, 20])
```

The data type can also be configured, here `float` is equivalent to `np.float64` :

In [13]:

```
b = np.array((10, 20), dtype=float)
b
```

Out[13]:

```
array([10., 20.])
```

Create a 2 dimensional, 2 by 2 array:

In [14]:

```
b = np.array([[1, 2], [3, 4]])
b
```

Out[14]:

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

Array indexing

For a flat array, indexing is the same as Python sequences.

In [15]:

```
c = np.linspace(1, 2, 5)
c
```

Out[15]:

```
array([1.  , 1.25, 1.5 , 1.75, 2.  ])
```

In [16]:

```
c[0]
```

Out[16]:

```
1.0
```

In [17]:

```
c[1:3]
```

Out[17]:

```
array([1.25, 1.5 ])
```

In [18]:

```
c[-1]
```

Out[18]:

```
2.0
```

For 2D arrays we use an index position for each dimension.

In [19]:

```
d = np.array([[1, 2], [3, 4]])
d
```

Out[19]:

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

In [20]:

```
d[0, 1]
```

Out[20]:

```
2
```

Note that indices are still zero-based, to maintain compatibility with Python sequences.

Columns and rows can be extracted as follows:

In [21]:

```
d[0, :]
```

Out[21]:

```
array([1, 2])
```

In [22]:

```
d[:, 1]
```

Out[22]:

```
array([2, 4])
```

NumPy arrays of integers can also be used to extract elements.

In [23]:

```
indices = np.array((0, 2, 3))
c[indices]
```

Out[23]:

```
array([1.  , 1.5 , 1.75])
```

A NumPy array of boolean values can be used to filter elements at the `True` locations.

In [24]:

```
e = np.array([0, 1, 1, 0, 0], dtype=bool)
e
```

Out[24]:

```
array([False,  True,  True, False, False])
```

In [25]:

```
c[e]
```

Out[25]:

```
array([1.25, 1.5 ])
```

Array Methods

Numpy arrays have useful methods, many of them should be familiar from previous lectures.

In [26]:

```
f = np.array((3, 2, 4, 1))
f
```

Out[26]:

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

In [27]:

```
f.sort() # Sorts a in place
f
```

Out[27]:

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

In [28]:

```
f.sum() # Sum
```

Out[28]:

```
10
```

In [29]:

```
f.mean() # Mean
```

Out[29]:

```
2.5
```

In [30]:

```
f.max() # Max
```

Out[30]:

```
4
```

In [31]:

```
f.argmax() # Returns the index of the maximal element
```

Out[31]:

```
3
```

In [32]:

```
f.cumsum() # Cumulative sum of the elements
```

Out[32]:

```
array([ 1,  3,  6, 10])
```

In [33]:

```
f.cumprod() # Cumulative product of the elements
```

Out[33]:

```
array([ 1,  2,  6, 24])
```

In [34]:

```
f.var() # Variance
```

Out[34]:

```
1.25
```

In [35]:

```
f.std() # Standard deviation
```

Out[35]:

```
1.118033988749895
```

In [36]:

```
f.shape = (2, 2)
f
```

Out[36]:

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

In [37]:

```
f.transpose() # or simpy f.T
```

Out[37]:

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

Many of the methods discussed above have equivalent functions in the NumPy namespace, e.g.:

In [38]:

```
print("Sum: {0}".format(np.sum(f)))
print("Mean: {0:.2f}".format(np.mean(f)))
```

```
Sum: 10
Mean: 2.50
```

Arithmetic Operations

The operators `+`, `-`, `*`, `/` and `**` all act **elementwise** on NumPy arrays.

In [39]:

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

Out[39]:

```
array([ 6,  8, 10, 12])
```

In [40]:

```
a * b
```

Out[40]:

```
array([ 5, 12, 21, 32])
```

In [41]:

```
a + 10
```

Out[41]:

```
array([11, 12, 13, 14])
```

In [42]:

```
a * 10
```

Out[42]:

```
array([10, 20, 30, 40])
```

Multi dimensional arrays follow the same general rules.

In [43]:

```
a.shape = (2, 2)
b.shape = (2, 2)
a + b
```

Out[43]:

```
array([[ 6,  8],
       [10, 12]])
```

In [44]:

```
a + 10
```

Out[44]:

```
array([[11, 12],
       [13, 14]])
```

In [45]:

```
a * b
```

Out[45]:

```
array([[ 5, 12],
       [21, 32]])
```

Calculate the *dot product* of two NumPy arrays.

In [46]:

```
np.dot(a, b)
```

Out[46]:

```
array([[19, 22],
       [43, 50]])
```

The `@` operator does the same thing.

In [47]:

```
a @ b
```

Out[47]:

```
array([[19, 22],
       [43, 50]])
```

Calculate the *cross product* of two NumPy arrays.

In [48]:

```
np.cross(a, b)
```

Out[48]:

```
array([-4, -4])
```

Random generation

Generate random numbers of the *standard normal* distribution:

In [49]:

```
g = np.random.randn(3)
g
```

Out[49]:

```
array([-0.04562699, -0.7312714 , -0.37247379])
```

Generate random integers between a lower (inclusive) and a higher (exclusive) bound:

In [50]:

```
g = np.random.randint(0, 100, 5)
g
```

Out[50]:

```
array([51, 57, 89, 13,  5])
```

Mutability and Copying Arrays

NumPy arrays are mutable data types, like Python lists. In other words, their contents can be altered (mutated) in memory after initialization.

To make an independent copy of a NumPy array, the `np.copy()` function can be used.

In [51]:

```
h = g
i = g.copy()
h[0] = 42

print(g)
print(h)
print(i)
```

```
[42 57 89 13  5]
[42 57 89 13  5]
[51 57 89 13  5]
```

Vectorized Functions

The `np.vectorize()` creates a *vectorized* function, which can be performed on a NumPy array in an elementwise manner.

In [52]:

```
# is_even() can be called on an integer number
def is_even(x): return x % 2 == 0

# is_even_vectorized() can be called on an array of integers
is_even_vectorized = np.vectorize(is_even)
is_even_vectorized(g)
```

Out[52]:

```
array([ True, False, False, False, False])
```

The NumPy function `np.where()` provides a vectorized alternative.

In [53]:

```
np.where(g % 2 == 0, 1, 0)
```

Out[53]:

```
array([1, 0, 0, 0, 0])
```

Comparisons

As a rule, comparisons on arrays are done elementwise.

In [54]:

```
z = np.array([2, 3])
y = np.array([2, 3])
z == y
```

Out[54]:

```
array([ True,  True])
```

In [55]:

```
y[0] = 5
z == y
```

Out[55]:

```
array([False,  True])
```

In [56]:

```
z != y
```

Out[56]:

```
array([ True, False])
```

The situation is similar for `>`, `<`, `>=` and `<=`.

We can also do comparisons against scalars:

In [57]:

```
x = np.linspace(0, 10, 5)
x
```

Out[57]:

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

In [58]:

```
x > 3
```

Out[58]:

```
array([False, False,  True,  True,  True])
```

This is particularly useful for *conditional extraction*:

In [59]:

```
cond = x > 3
x[cond]
```

Out[59]:

```
array([ 5. ,  7.5, 10. ])
```

Of course we can - and frequently do - perform this in one step:

In [60]:

```
x[x > 3]
```

Out[60]:

```
array([ 5. ,  7.5, 10. ])
```

Linear algebra

In [61]:

```
k = np.array([[1, 2], [3, 4]])
k
```

Out[61]:

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

Compute the determinant:

In [62]:

```
np.linalg.det(k)
```

Out[62]:

```
-2.0000000000000004
```

Compute the inverse:

In [63]:

```
np.linalg.inv(k)
```

Out[63]:

```
array([[-2. ,  1. ],
       [ 1.5, -0.5]])
```

Interpolation

Generate 20 evenly distributed number between 0 and 10 into `x`. Generate the sine function value into `y` for each elements in `x`.

In [64]:

```
x = np.linspace(0, 10, 20)
y = np.sin(x)
print(x)
print(y)
```

```
[ 0.          0.52631579  1.05263158  1.57894737  2.10526316  2.6315
7895
  3.15789474  3.68421053  4.21052632  4.73684211  5.26315789  5.7894
7368
  6.31578947  6.84210526  7.36842105  7.89473684  8.42105263  8.9473
6842
  9.47368421 10.        ]
[ 0.          0.50235115  0.86872962  0.99996678  0.86054034  0.4881
8921
 -0.01630136 -0.5163796  -0.87668803 -0.99970104 -0.85212237 -0.4738
9753
  0.03259839  0.53027082  0.88441346  0.99916962  0.84347795  0.4594
799
 -0.04888676 -0.54402111]
```

Generate 100 evenly distributed number between 0 and 10 into `xvals`. Calculate the interpolated values into `yinterp` for each elements in `xvals`, based on `x` and `y`.

In [65]:

```
xvals = np.linspace(0, 10, 100)
yinterp = np.interp(xvals, x, y)
print(xvals)
print(yinterp)
```

```
[ 0.          0.1010101   0.2020202   0.3030303   0.4040404   0.5050
5051
  0.60606061  0.70707071  0.80808081  0.90909091  1.01010101  1.1111
1111
  ...
  9.09090909  9.19191919  9.29292929  9.39393939  9.49494949  9.5959
596
  9.6969697   9.7979798   9.8989899  10.        ]
[ 0.          0.09641083  0.19282166  0.28923248  0.38564331  0.4820
5414
  0.55786304  0.6281781   0.69849316  0.76880822  0.83912328  0.8833
1152
  ...
  0.32083445  0.22326913  0.12570381  0.0281385  -0.06889218 -0.1639
1797
 -0.25894376 -0.35396954 -0.44899533 -0.54402111]
```

Visualize the results on a plot. *(For plotting, see Chapter 10 (10_plotting.pdf).)*

In [66]:

```
import matplotlib.pyplot as plt
%matplotlib inline

plt.plot(x, y, 'o')
plt.plot(xvals, yinterp, '-x')
plt.show()
```