# numpy

- Cheatsheet 1
- Cheatsheet 2

Numpy , ( wiki ) is a python library for handling n-dimensional arrays. You may be wondering why we need another library to do something we can do with classic python structures (i.e. lists of lists). The answer is in the following extract from wikipedia which states:

> both MATLAB and NumPy rely on BLAS and LAPACK for efficient linear algebra computations.

### But what are these BLAS and LAPACK?

These are libraries or collections of functions for manipulating numerical data, performing complex calculations and applyinh linear algebra methids.

### What is so special about these functions?

The creation of these functions began in the 1970s. Then computers were very different from today. Each byte of memory cost a lot and the processors were much slower. The functions that did arithmetic operations had to be very cleverly designed. If we consider how important these calculations are (from calculating satellite orbits, to simple banking applications), we can imagine the magnitude of the effort made to implement them. It is indicative that BLAS began to develop at NASA . These functions "run" today in any system that does arithmetic. From robots on Mars, cell phones, to ... washing machines and cars!

As a result, these functions are probably the most widely tested and error-free code ever written.

Both their quality and the rise in processing power of personal computers make these libraries ideal for research. So numpy is nothing more than an attempt to "bring" the power of these functions to python. When loading numpy in python, your computer is essentially "calling" these functions.

We start by importing numpy:

In [3]:

```python
import numpy as np
```

Let's create a two-dimensional 2X3 (2 rows 3 columns) table:

In [557…

```python
a = np.array([[1,2,3], [4,5,6]])
a
```

Out[557…

```
array([[1, 2, 3],
       [4, 5, 6]])
```

We can access an element of the table as follows:

In [501…

```
a[1,2]
```

Out[501…

```
6
```

Similarly we can change an element of the table:

In [502…

```
a[1,2] = 10
a
```

Out[502…

```
array([[ 1,  2,  3],
       [ 4,  5, 10]])
```

The size of the table for each dimension:

In [4]:

```
a.shape
```

Out[4]:

```
(2, 3)
```

The number of dimensions:

In [7]:

```
a.ndim
```

Out[7]:

```
2
```

The number of items in the table:

In [9]:

```
a.size
```

Out[9]:

```
6
```

Table only with values of 1

In [15]:

```
np.ones([3,5])
```

Out[15]:

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

Table only with values 0

In [18]:

```
np.zeros([3,5])
```

Out[18]:

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

An empty table containing "junk" (ie various values from previous memory usage):

In [571…

```
np.empty([3,5])
```

Out[571…

```
array([[ 0.69209415,  0.74405598,  0.58994422,  0.51903866,  0.93660742],
       [ 0.69209415,  0.74405598,  0.58994422,  0.51903866,  0.93660742],
       [ 0.42445219,  0.54983743,  0.16428273,  0.71051264,  0.13425249]])
```

`arange` is like range with the difference that all values can be decimal:

In [21]:

```
np.arange(1,10,0.3)
```

Out[21]:

```
array([ 1. ,  1.3,  1.6,  1.9,  2.2,  2.5,  2.8,  3.1,  3.4,  3.7,  4. ,
        4.3,  4.6,  4.9,  5.2,  5.5,  5.8,  6.1,  6.4,  6.7,  7. ,  7.3,
        7.6,  7.9,  8.2,  8.5,  8.8,  9.1,  9.4,  9.7])
```

We can of course also iterate (i.e. apply `for` ) to a table. But every time we do this, a little bunny gets very upset:

In [23]:

```
for x in np.arange(1,10,.3):
    print(x)
```

```
1.0
1.3
1.6
1.9
2.2
2.5
2.8
3.1
3.4
3.7
4.0
4.3
4.6
4.9
5.2
5.5
5.8
6.1
```

```
6.4
6.7
7.0
7.3
7.6
7.9
8.2
8.5
8.8
9.1
9.4
9.7
```

The reason for this is that numpy contains hundreds of functions for very fast table management and transformation. So before you are tempted to do a `for`, at least do a google search if there is a function that does this for you..

In addition to arange there is also linspace which also creates a numerical progression from one number to another. The difference is that in linspace the third parameter indicates how many numbers we want the progress to have (while in arange it indicates the progress step)

```
In [26]: np.linspace(1,10,20)
```

```
Out[26]: array([  1.        ,   1.47368421,   1.94736842,   2.42105263,
                  2.89473684,   3.36842105,   3.84210526,   4.31578947,
                  4.78947368,   5.26315789,   5.73684211,   6.21052632,
                  6.68421053,   7.15789474,   7.63157895,   8.10526316,
                  8.57894737,   9.05263158,   9.52631579,  10.        ])
```

Random numbers:

```
In [30]: np.random.random([4,5])
```

```
Out[30]: array([[ 0.73041476,  0.85627737,  0.1132018 ,  0.74100275,  0.82242056],
                [ 0.66131635,  0.16408397,  0.00490488,  0.40217054,  0.2286295 ],
                [ 0.38351069,  0.3711942 ,  0.88431264,  0.55911999,  0.06282348],
                [ 0.53250803,  0.77532521,  0.23247395,  0.28375013,  0.36304702]])
```

In numpy we can apply functions across the whole table:

```
In [31]: a
```

```
Out[31]: array([[1, 2, 3],
                [4, 5, 6]])
```

```
In [32]: np.sin(a)
```

```
Out[32]: array([[ 0.84147098,  0.90929743,  0.14112001],
                [-0.7568025 , -0.95892427, -0.2794155 ]])
```

```
In [33]: np.exp(a)
```

```
Out[33]: array([[   2.71828183,    7.3890561 ,   20.08553692],
                [  54.59815003,  148.4131591 ,  403.42879349]])
```

**Caution!** np.log applies the natural logarithm (ln)

```
In [34]: np.log(a) # ln e
```

```
Out[34]: array([[ 0.        ,  0.69314718,  1.09861229],
                [ 1.38629436,  1.60943791,  1.79175947]])
```

Whereas the decimal number is:

```
In [36]: np.log10(a)
```

```
Out[36]: array([[ 0.        ,  0.30103   ,  0.47712125],
               [ 0.60205999,  0.69897   ,  0.77815125]])
```

Constants:

```
In [40]: np.pi
```

```
Out[40]: 3.141592653589793
```

```
In [42]: np.e
```

```
Out[42]: 2.718281828459045
```

Element to element (or elementwise) operations between tables:

```
In [36]: a=np.array([[1,2,3], [4,5,6]])
         b=np.array([[4,1,0], [5,0,1]])
```

```
In [45]: a+b
```

```
Out[45]: array([[5, 3, 3],
               [9, 5, 7]])
```

```
In [46]: a*b
```

```
Out[46]: array([[ 4,  2,  0],
               [20,  0,  6]])
```

Multiplication of tables:

```
In [40]: np.matmul(a, b.T)
```

```
Out[40]: array([[ 6,  8],
               [21, 26]])
```

In python 3.5 and later there is the "@" operator for this:

```
In [41]: a @ b.T
```

```
Out[41]: array([[ 6,  8],
               [21, 26]])
```

Multiplication of tables but with a different result for non-two-dimensional tables is also done by dot:

```
In [37]: np.dot(a,b.T)
```

```
Out[37]: array([[ 6,  8],
               [21, 26]])
```

`a.T` returns the transposed array. ( **ATTENTION!** This is different than the inversed!)

```
In [558… a
```

```
Out[558… array([[1, 2, 3],
               [4, 5, 6]])
```

```
In [559… a.T
```

```
Out[559… array([[1, 4],
                [2, 5],
                [3, 6]])
```

Synonymous with `a.T` is `transpose`:

```
In [560… a.transpose()
```

```
Out[560… array([[1, 4],
                [2, 5],
                [3, 6]])
```

Multiplication of tables is done with dot:

```
In [53]: a.dot(b.T)
```

```
Out[53]: array([[ 6,  8],
                [21, 26]])
```

or else:

```
In [562… np.dot(a,b.T)
```

```
Out[562… array([[ 6,  8],
                [21, 26]])
```

With `reshape` we change the dimensions of a table:

```
In [54]: a
```

```
Out[54]: array([[1, 2, 3],
                [4, 5, 6]])
```

```
In [56]: a.reshape(3,2)
```

```
Out[56]: array([[1, 2],
                [3, 4],
                [5, 6]])
```

```
In [57]: a.reshape(6,1)
```

```
Out[57]: array([[1],
                [2],
                [3],
                [4],
                [5],
                [6]])
```

```
In [422… a.reshape(1,6)
```

```
Out[422… array([[1, 2, 3, 4, 5, 6]])
```

If any parameter of the reshape is -1 then it automatically calculates its value (if it can):

```
In [423… np.random.random((6,2)).reshape(3,-1) # From 6 lines to 3 lines
```

```
Out[423… array([[ 0.04903716,  0.9785132 ,  0.84331155,  0.94860724],
                [ 0.85665382,  0.20847732,  0.24546295,  0.6111018 ],
                [ 0.07518856,  0.42283486,  0.47115603,  0.14732587]])
```

```
In [424… np.linspace(1,10,20).reshape(-1,10)# From 1 line to 2 lines
```

```
Out[424… array([[  1.        ,   1.47368421,   1.94736842,   2.42105263,
                  2.89473684,   3.36842105,   3.84210526,   4.31578947,
               4.78947368,    5.26315789],
            [  5.73684211,    6.21052632,    6.68421053,    7.15789474,
               7.63157895,    8.10526316,    8.57894737,    9.05263158,
               9.52631579,   10.        ]])
```

**Caution!** The table does not change with `reshape` . If we want to change the table then we use resize:

In [533…
```
a = np.random.random((2,3))
a
```

Out[533…
```
array([[ 0.86501027,  0.87671615,  0.86737332],
       [ 0.84734983,  0.14904097,  0.82635533]])
```

In [534…
```
a.reshape(3,2)
```

Out[534…
```
array([[ 0.86501027,  0.87671615],
       [ 0.86737332,  0.84734983],
       [ 0.14904097,  0.82635533]])
```

In [535…
```
a # Did not change
```

Out[535…
```
array([[ 0.86501027,  0.87671615,  0.86737332],
       [ 0.84734983,  0.14904097,  0.82635533]])
```

In [536…
```
a.resize(3,2)
```

In [537…
```
a # Changed!
```

Out[537…
```
array([[ 0.86501027,  0.87671615],
       [ 0.86737332,  0.84734983],
       [ 0.14904097,  0.82635533]])
```

`min` , `max` , `sum` are applied to the whole table:

In [538…
```
a
```

Out[538…
```
array([[ 0.86501027,  0.87671615],
       [ 0.86737332,  0.84734983],
       [ 0.14904097,  0.82635533]])
```

In [539…
```
a.min()
```

Out[539…
```
0.14904097385101789
```

If we provide the axis parameter, then it finds all the values separately for this dimension:

In [540…
```
a.min(axis=0)
```

Out[540…
```
array([ 0.14904097,  0.82635533])
```

In [541…
```
a.min(axis=1)
```

Out[541…
```
array([ 0.86501027,  0.84734983,  0.14904097])
```

In [542…
```
a
```

Out[542…
```
array([[ 0.86501027,  0.87671615],
       [ 0.86737332,  0.84734983],
       [ 0.14904097,  0.82635533]])
```

```
In [543…  a.sum()
Out[543…  4.4318458782793408

In [544…  a.sum(axis=0)
Out[544…  array([ 1.88142456,  2.55042132])

In [545…  a.sum(axis=1)
Out[545…  array([ 1.74172642,  1.71472315,  0.97539631])
```

`argmin` and `argmax` return the index of the lowest (or largest):

```
In [546…  a.argmin()
Out[546…  4

In [547…  a.argmin(axis=0)
Out[547…  array([2, 2])

In [548…  a.argmin(axis=1)
Out[548…  array([0, 1, 0])
```

As with lists, so with numpy tables we can use indexing to get ranges of values.

```
In [82]:  b = [1,2,3,4,5,6,7,8,9,10]

In [81]:  type(a)
Out[81]:  numpy.ndarray

In [84]:  type(b)
Out[84]:  list

In [86]:  b[5:]
Out[86]:  [6, 7, 8, 9, 10]

In [87]:  a
Out[87]:  array([[1, 2, 3],
                 [4, 5, 6]])

In [427…  a = np.random.random([10,5])

In [428…  a
Out[428…  array([[ 0.69209415,  0.74405598,  0.58994422,  0.51903866,  0.93660742],
                 [ 0.42445219,  0.54983743,  0.16428273,  0.71051264,  0.13425249],
                 [ 0.42144851,  0.69445979,  0.40491544,  0.46205062,  0.76795599],
                 [ 0.37588999,  0.63994809,  0.28475934,  0.21667052,  0.07422019],
                 [ 0.01741184,  0.28740475,  0.56879267,  0.63534581,  0.0612609 ],
                 [ 0.9298351 ,  0.35012857,  0.51996718,  0.44845842,  0.92092282],
                 [ 0.12143151,  0.94833192,  0.2439955 ,  0.44217524,  0.45783427],
                 [ 0.83292434,  0.90528182,  0.26152684,  0.46834753,  0.30246709],
                 [ 0.03192285,  0.70764385,  0.94167941,  0.02025402,  0.69930778],
                 [ 0.59585316,  0.7778326 ,  0.09498829,  0.45896575,  0.16663657]])
```

But we can put different intervals in each dimension. Eg: Get rows 2,3,4 and all columns:

In [429…

```
a[1:4,:]
```

Out[429…

```
array([[ 0.42445219,  0.54983743,  0.16428273,  0.71051264,  0.13425249],
       [ 0.42144851,  0.69445979,  0.40491544,  0.46205062,  0.76795599],
       [ 0.37588999,  0.63994809,  0.28475934,  0.21667052,  0.07422019]])
```

Rows 2,3,4 and columns 1,2:

In [430…

```
a[1:4, 0:2]
```

Out[430…

```
array([[ 0.42445219,  0.54983743],
       [ 0.42144851,  0.69445979],
       [ 0.37588999,  0.63994809]])
```

Rows 2 through 5 (without 5) with step 2 and columns 1,2

In [431…

```
a[1:4:2, 0:2]
```

Out[431…

```
array([[ 0.42445219,  0.54983743],
       [ 0.37588999,  0.63994809]])
```

Rows 5,4,3 and columns 1,2

In [432…

```
a[4:1:-1, 0:2]
```

Out[432…

```
array([[ 0.01741184,  0.28740475],
       [ 0.37588999,  0.63994809],
       [ 0.42144851,  0.69445979]])
```

We can also declare specific rows (or columns), instead of spaces:

In [433…

```
a
```

Out[433…

```
array([[ 0.69209415,  0.74405598,  0.58994422,  0.51903866,  0.93660742],
       [ 0.42445219,  0.54983743,  0.16428273,  0.71051264,  0.13425249],
       [ 0.42144851,  0.69445979,  0.40491544,  0.46205062,  0.76795599],
       [ 0.37588999,  0.63994809,  0.28475934,  0.21667052,  0.07422019],
       [ 0.01741184,  0.28740475,  0.56879267,  0.63534581,  0.0612609 ],
       [ 0.9298351 ,  0.35012857,  0.51996718,  0.44845842,  0.92092282],
       [ 0.12143151,  0.94833192,  0.2439955 ,  0.44217524,  0.45783427],
       [ 0.83292434,  0.90528182,  0.26152684,  0.46834753,  0.30246709],
       [ 0.03192285,  0.70764385,  0.94167941,  0.02025402,  0.69930778],
       [ 0.59585316,  0.7778326 ,  0.09498829,  0.45896575,  0.16663657]])
```

In [434…

```
a[[1, 5, 9],:]
```

Out[434…

```
array([[ 0.42445219,  0.54983743,  0.16428273,  0.71051264,  0.13425249],
       [ 0.9298351 ,  0.35012857,  0.51996718,  0.44845842,  0.92092282],
       [ 0.59585316,  0.7778326 ,  0.09498829,  0.45896575,  0.16663657]])
```

This is equivalent to:

In [435…

```
a[[1, 5, 9],]
```

Out[435…

```
array([[ 0.42445219,  0.54983743,  0.16428273,  0.71051264,  0.13425249],
       [ 0.9298351 ,  0.35012857,  0.51996718,  0.44845842,  0.92092282],
       [ 0.59585316,  0.7778326 ,  0.09498829,  0.45896575,  0.16663657]])
```

In [436…

```
a[[1, 5, 9],4:1:-1]
```

```
Out[436… array([[ 0.13425249,  0.71051264,  0.16428273],
                [ 0.92092282,  0.44845842,  0.51996718],
                [ 0.16663657,  0.45896575,  0.09498829]])
```

```
In [437… a[[1, 5, 9],[2,3,4]]
```

```
Out[437… array([ 0.16428273,  0.44845842,  0.16663657])
```

This is equivalent to:

```
In [439… b = [[1, 5, 9],[2,3,4]]
```

```
In [440… a[b]
```

```
Out[440… array([ 0.16428273,  0.44845842,  0.16663657])
```

The values in the indexes can be repeated:

```
In [442… a[[0, 0, 1],:]
```

```
Out[442… array([[ 0.69209415,  0.74405598,  0.58994422,  0.51903866,  0.93660742],
                [ 0.69209415,  0.74405598,  0.58994422,  0.51903866,  0.93660742],
                [ 0.42445219,  0.54983743,  0.16428273,  0.71051264,  0.13425249]])
```

Indexes can also be numpy arrays:

```
In [443… a = np.random.random([10, 3])
         a
```

```
Out[443… array([[ 0.72032351,  0.1700159 ,  0.86081639],
                [ 0.61504558,  0.09700244,  0.58100856],
                [ 0.69110959,  0.06757325,  0.04092057],
                [ 0.51988531,  0.53892816,  0.2990308 ],
                [ 0.15582942,  0.46055668,  0.83132364],
                [ 0.79502634,  0.29743753,  0.76092162],
                [ 0.93368062,  0.7454287 ,  0.76971832],
                [ 0.26996306,  0.05723047,  0.26819277],
                [ 0.72463148,  0.70074029,  0.03486837],
                [ 0.46151354,  0.38307966,  0.37576748]])
```

```
In [444… a[np.array([0,0])] # The firs line, twice
```

```
Out[444… array([[ 0.72032351,  0.1700159 ,  0.86081639],
                [ 0.72032351,  0.1700159 ,  0.86081639]])
```

```
In [445… a[np.array([0,0]),:] # Same as above
```

```
Out[445… array([[ 0.72032351,  0.1700159 ,  0.86081639],
                [ 0.72032351,  0.1700159 ,  0.86081639]])
```

```
In [446… a[:,np.array([0,0])] # The first column, twice
```

```
Out[446… array([[ 0.72032351,  0.72032351],
                [ 0.61504558,  0.61504558],
                [ 0.69110959,  0.69110959],
                [ 0.51988531,  0.51988531],
                [ 0.15582942,  0.15582942],
                [ 0.79502634,  0.79502634],
                [ 0.93368062,  0.93368062],
                [ 0.26996306,  0.26996306],
                [ 0.72463148,  0.72463148],
                [ 0.46151354,  0.46151354]])
```

Let's look at an example with 3 dimensions:

```
In [9]: b = np.random.random((4,3,2))
        b
```

```
Out[9]: array([[[0.89160374, 0.91028464],
                [0.09924166, 0.54140992],
                [0.73218525, 0.64126897]],

               [[0.76625221, 0.66317366],
                [0.72339298, 0.84799596],
                [0.37241228, 0.75853395]],

               [[0.9483431 , 0.93867438],
                [0.13213638, 0.5126784 ],
                [0.05343845, 0.91251162]],

               [[0.73087434, 0.50490406],
                [0.89551794, 0.69999949],
                [0.69710292, 0.89773047]]])
```

```
In [10]: b.sum(axis=0) # 0.89160374 + 0.76625221 + 0.9483431 + 0.73087434 = 3.33707
```

```
Out[10]: array([[3.33707339, 3.01703675],
                [1.85028895, 2.60208376],
                [1.85513889, 3.21004501]])
```

```
In [13]: b.sum(axis=1) # 0.89160374 + 0.09924166 + 0.73218525 = 1.72303065
```

```
Out[13]: array([[1.72303065, 2.09296352],
                [1.86205746, 2.26970357],
                [1.13391793, 2.3638644 ],
                [2.32349519, 2.10263403]])
```

```
In [16]: b.sum(axis=2) # 0.89160374 + 0.91028464 = 1.80188838
```

```
Out[16]: array([[1.80188838, 0.64065158, 1.37345422],
                [1.42942588, 1.57138893, 1.13094623],
                [1.88701748, 0.64481478, 0.96595008],
                [1.2357784 , 1.59551742, 1.59483339]])
```

```
In [21]: b.sum(axis=(0,1)) # 3.33707339 + 1.85028895 + 1.85513889
```

```
Out[21]: array([7.04250124, 8.82916552])
```

```
In [24]: b.sum(axis=(0,2)) # 3.33707339+ 3.01703675 = 6.35411014
```

```
Out[24]: array([6.35411014, 4.45237271, 5.06518391])
```

```
In [30]: b.sum(axis=(1,2)) # 1.72303065 + 2.09296352 = 3.8159941
```

```
Out[30]: array([3.81599417, 4.13176104, 3.49778233, 4.42612922])
```

```
In [33]: b.sum(axis=(0,1,2))
```

```
Out[33]: 15.87166676193048
```

For the first two tables ... take the last row ... and the first column

```
In [35]: b[ :2 , -1 , :1 ]
```

```
Out[35]: array([[0.73218525],
                [0.37241228]])
```

```
In [448…  a
```

```
Out[448…  array([[ 0.72032351,  0.1700159 ,  0.86081639],
                 [ 0.61504558,  0.09700244,  0.58100856],
                 [ 0.69110959,  0.06757325,  0.04092057],
                 [ 0.51988531,  0.53892816,  0.2990308 ],
                 [ 0.15582942,  0.46055668,  0.83132364],
                 [ 0.79502634,  0.29743753,  0.76092162],
                 [ 0.93368062,  0.7454287 ,  0.76971832],
                 [ 0.26996306,  0.05723047,  0.26819277],
                 [ 0.72463148,  0.70074029,  0.03486837],
                 [ 0.46151354,  0.38307966,  0.37576748]])
```

## Convert from n-d to 1D (flattening)

The first way is with a typical list comprehension. **CAUTION!** this is also the most "wrong" way. Firstly because it creates depressed bunnies and secondly because it only works with two-dimensional arrays (but with some changes it can also work with n-dimensional):

```
In [450…  [y for x in a for y in x]
```

```
Out[450…  [0.72032351104388803,
           0.17001589545430396,
           0.86081639054798709,
           0.6150455802614303,
           0.09700244086120513,
           0.58100855972950105,
           0.69110959169793174,
           0.067573249596323492,
           0.040920571671677952,
           0.51988530822513479,
           0.53892816090621609,
           0.2990307992517659,
           0.15582942357273566,
           0.4605566802126575,
           0.8313236400044498,
           0.7950263426004629,
           0.29743752966900272,
           0.76092162436963118,
           0.93368061958375614,
           0.74542870354155555,
           0.76971832134271212,
           0.26996305881053206,
           0.057230468408892787,
           0.2681927734696512,
           0.72463147887657697,
           0.70074028551106737,
           0.034868370907724544,
           0.46151353728526512,
           0.38307966172904284,
           0.37576747710142233]
```

The 2nd way is with the `flat` function. This function creates a generator:

```
In [144…  a
```

```
Out[144…  array([[ 0.72736581,  0.54898777,  0.30900569],
       [ 0.94525329,  0.39233765,  0.81590939],
       [ 0.20146162,  0.99969513,  0.92789164],
       [ 0.53228237,  0.93805259,  0.80061147],
       [ 0.26791742,  0.5269165 ,  0.5012809 ],
       [ 0.25878137,  0.36084797,  0.95754485],
       [ 0.18318426,  0.92218919,  0.86068247],
       [ 0.84290356,  0.77998675,  0.6906613 ],
       [ 0.23294411,  0.96024721,  0.59429307],
```

In [453…

```
a.flat
```

Out[453…

```
<numpy.flatiter at 0x7f954cbcfc00>
```

The third way is with ravel:

In [148…

```
a.shape
```

Out[148…

```
(10, 3)
```

In [149…

```
a.ravel().shape
```

Out[149…

```
(30,)
```

We can join two (or more) tables:

In [457…

```
a = np.random.random([2,3])
b = np.random.random([2,3])
```

In [458…

```
a
```

Out[458…

```
array([[ 0.29052439,  0.24849151,  0.36284575],
       [ 0.92366061,  0.43703868,  0.06389083]])
```

In [459…

```
b
```

Out[459…

```
array([[ 0.8748667 ,  0.73579282,  0.20178447],
       [ 0.21344032,  0.98158518,  0.73810592]])
```

The `vstack` function joins the tables vertically, ie one below the other:

In [460…

```
np.vstack([a,b])
```

Out[460…

```
array([[ 0.29052439,  0.24849151,  0.36284575],
       [ 0.92366061,  0.43703868,  0.06389083],
       [ 0.8748667 ,  0.73579282,  0.20178447],
       [ 0.21344032,  0.98158518,  0.73810592]])
```

In [461…

```
np.vstack([a,b,2*a])
```

Out[461…

```
array([[ 0.29052439,  0.24849151,  0.36284575],
       [ 0.92366061,  0.43703868,  0.06389083],
       [ 0.8748667 ,  0.73579282,  0.20178447],
       [ 0.21344032,  0.98158518,  0.73810592],
       [ 0.58104878,  0.49698301,  0.72569149],
       [ 1.84732122,  0.87407735,  0.12778165]])
```

The `hstack` function joins the tables horizontally, ie next to each other:

In [462…

```
a
```

Out[462…

```
array([[ 0.29052439,  0.24849151,  0.36284575],
       [ 0.92366061,  0.43703868,  0.06389083]])
```

```
In [463…  b
```

```
Out[463…  array([[ 0.8748667 ,  0.73579282,  0.20178447],
                 [ 0.21344032,  0.98158518,  0.73810592]])
```

```
In [464…  np.hstack([a,b])
```

```
Out[464…  array([[ 0.29052439,  0.24849151,  0.36284575,  0.8748667 ,  0.73579282,
                   0.20178447],
                 [ 0.92366061,  0.43703868,  0.06389083,  0.21344032,  0.98158518,
                   0.73810592]])
```

Attention! in `vstack` the number of columns must be the same. In `hstack` the number of lines must be the same:

```
In [465…  np.hstack([a,b[:,:-1]])
```

```
Out[465…  array([[ 0.29052439,  0.24849151,  0.36284575,  0.8748667 ,  0.73579282],
                 [ 0.92366061,  0.43703868,  0.06389083,  0.21344032,  0.98158518]])
```

```
In [466…  a
```

```
Out[466…  array([[ 0.29052439,  0.24849151,  0.36284575],
                 [ 0.92366061,  0.43703868,  0.06389083]])
```

```
In [467…  b[:,:-1]
```

```
Out[467…  array([[ 0.8748667 ,  0.73579282],
                 [ 0.21344032,  0.98158518]])
```

```
In [468…  np.vstack([a,b[:,:-1]])
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-468-4935f9921709> in <module>()
----> 1 np.vstack([a,b[:,:-1]])

~/anaconda3/envs/arkalos/lib/python3.6/site-packages/numpy/core/shape_base.
py in vstack(tup)
    235
    236     """
--> 237     return _nx.concatenate([atleast_2d(_m) for _m in tup], 0)
    238
    239 def hstack(tup):

ValueError: all the input array dimensions except for the concatenation axi
s must match exactly
```

Instead of `hstack` and `vstack` you can use `block` . Just make lists of the tables you want to join: [a, b] --> same as hstack. [[a], [b]] --> same as vstack.

```
In [469…  a
```

```
Out[469…  array([[ 0.29052439,  0.24849151,  0.36284575],
                 [ 0.92366061,  0.43703868,  0.06389083]])
```

```
In [470…  np.block([a,b])
```

```
Out[470…  array([[ 0.29052439,  0.24849151,  0.36284575,  0.8748667 ,  0.73579282,
                   0.20178447],
                 [ 0.92366061,  0.43703868,  0.06389083,  0.21344032,  0.98158518,
                   0.73810592]])
```

In [471…

```
np.block([[a], [b]])
```

Out[471…

```
array([[ 0.29052439,  0.24849151,  0.36284575],
       [ 0.92366061,  0.43703868,  0.06389083],
       [ 0.8748667 ,  0.73579282,  0.20178447],
       [ 0.21344032,  0.98158518,  0.73810592]])
```

`vsplit` and `hsplit` do the opposite of `vstack` and `hstack` :

In [504…

```
a = np.random.random((10,4))
a
```

Out[504…

```
array([[ 0.81246515,  0.64579695,  0.31261692,  0.72833299],
       [ 0.50548399,  0.45691983,  0.94293484,  0.10713851],
       [ 0.25177997,  0.2096684 ,  0.50523253,  0.37108323],
       [ 0.16177285,  0.31801499,  0.10796055,  0.45283983],
       [ 0.92833983,  0.40167612,  0.42314142,  0.55412818],
       [ 0.72985404,  0.64141386,  0.68094954,  0.41604735],
       [ 0.8929054 ,  0.88354153,  0.86002467,  0.54289843],
       [ 0.60979488,  0.36884681,  0.61865976,  0.74078811],
       [ 0.39804021,  0.08909003,  0.05669355,  0.16086856],
       [ 0.96258289,  0.47762343,  0.69156939,  0.96706104]])
```

In [505…

```
a.shape
```

Out[505…

```
(10, 4)
```

In [506…

```
np.hsplit(a, 2) # Creates a list with two arrays. Every array is 10X2
```

Out[506…

```
[array([[ 0.81246515,  0.64579695],
        [ 0.50548399,  0.45691983],
        [ 0.25177997,  0.2096684 ],
        [ 0.16177285,  0.31801499],
        [ 0.92833983,  0.40167612],
        [ 0.72985404,  0.64141386],
        [ 0.8929054 ,  0.88354153],
        [ 0.60979488,  0.36884681],
        [ 0.39804021,  0.08909003],
        [ 0.96258289,  0.47762343]]), array([[ 0.31261692,  0.72833299],
        [ 0.94293484,  0.10713851],
        [ 0.50523253,  0.37108323],
        [ 0.10796055,  0.45283983],
        [ 0.42314142,  0.55412818],
        [ 0.68094954,  0.41604735],
        [ 0.86002467,  0.54289843],
        [ 0.61865976,  0.74078811],
        [ 0.05669355,  0.16086856],
        [ 0.69156939,  0.96706104]])]
```

In an array we can apply logical operations:

In [507…

```
a
```

Out[507…

```
array([[ 0.81246515,  0.64579695,  0.31261692,  0.72833299],
       [ 0.50548399,  0.45691983,  0.94293484,  0.10713851],
       [ 0.25177997,  0.2096684 ,  0.50523253,  0.37108323],
       [ 0.16177285,  0.31801499,  0.10796055,  0.45283983],
       [ 0.92833983,  0.40167612,  0.42314142,  0.55412818],
       [ 0.72985404,  0.64141386,  0.68094954,  0.41604735],
       [ 0.8929054 ,  0.88354153,  0.86002467,  0.54289843],
       [ 0.60979488,  0.36884681,  0.61865976,  0.74078811],
       [ 0.39804021,  0.08909003,  0.05669355,  0.16086856],
       [ 0.96258289,  0.47762343,  0.69156939,  0.96706104]])
```

In [508…

```
a>0.5
```

Out[508…

```
array([[ True,  True, False,  True],
       [ True, False,  True, False],
       [False, False,  True, False],
       [False, False, False, False],
       [ True, False, False,  True],
       [ True,  True,  True, False],
       [ True,  True,  True,  True],
       [ True, False,  True,  True],
       [False, False, False, False],
       [ True, False,  True,  True]], dtype=bool)
```

Even more interesting is that we can put a table of boolean values (True, False) in the index of another table! The result is a new table that contains only the elements whose index was True:

In [490…

```
b = np.array([5,3,1])
b
```

Out[490…

```
array([5, 3, 1])
```

In [491…

```
b[[True, False,True]]
```

Out[491…

```
array([5, 1])
```

In [492…

```
b>2
```

Out[492…

```
array([ True,  True, False], dtype=bool)
```

Therefore, I can use boolean table operations as an index in the table itself!

In [493…

```
b[b>2]
```

Out[493…

```
array([5, 3])
```

E.g. get all the elements of `a` which are `> 0.5` :

In [509…

```
a[a>0.5]
```

Out[509…

```
array([ 0.81246515,  0.64579695,  0.72833299,  0.50548399,  0.94293484,
        0.50523253,  0.92833983,  0.55412818,  0.72985404,  0.64141386,
        0.68094954,  0.8929054 ,  0.88354153,  0.86002467,  0.54289843,
        0.60979488,  0.61865976,  0.74078811,  0.96258289,  0.69156939,
        0.96706104])
```

In [510…

```
a
```

Out[510…
```
array([[ 0.81246515,  0.64579695,  0.31261692,  0.72833299],
       [ 0.50548399,  0.45691983,  0.94293484,  0.10713851],
       [ 0.25177997,  0.2096684 ,  0.50523253,  0.37108323],
       [ 0.16177285,  0.31801499,  0.10796055,  0.45283983],
       [ 0.92833983,  0.40167612,  0.42314142,  0.55412818],
       [ 0.72985404,  0.64141386,  0.68094954,  0.41604735],
       [ 0.8929054 ,  0.88354153,  0.86002467,  0.54289843],
       [ 0.60979488,  0.36884681,  0.61865976,  0.74078811],
       [ 0.39804021,  0.08909003,  0.05669355,  0.16086856],
       [ 0.96258289,  0.47762343,  0.69156939,  0.96706104]])
```

It is important to emphasize that this "algebra" is also supported by R, Matlab, Octave. We can also assign into these items. E.g. take all the elements of `a` such that `> 0.5` and replace them with the value `10` :

In [511…
```
a[a>0.5] = 10
a
```

Out[511…
```
array([[ 10.        ,  10.        ,   0.31261692,  10.        ],
       [ 10.        ,   0.45691983,  10.        ,   0.10713851],
       [  0.25177997,   0.2096684 ,  10.        ,   0.37108323],
       [  0.16177285,   0.31801499,   0.10796055,   0.45283983],
       [ 10.        ,   0.40167612,   0.42314142,  10.        ],
       [ 10.        ,  10.        ,  10.        ,   0.41604735],
       [ 10.        ,  10.        ,  10.        ,  10.        ],
       [ 10.        ,   0.36884681,  10.        ,  10.        ],
       [  0.39804021,   0.08909003,   0.05669355,   0.16086856],
       [ 10.        ,   0.47762343,  10.        ,  10.        ]])
```

We can assign an entire table to a subset of another table. However, the number of dimensions between the two tables has to be the same:

In [530…
```
b = np.arange(1,11)
b
```

Out[530…
```
array([ 1,  2,  3,  4,  5,  6,  7,  8,  9, 10])
```

In [531…
```
b[b<4] = np.array([20,21,22])
b
```

Out[531…
```
array([20, 21, 22,  4,  5,  6,  7,  8,  9, 10])
```

The `~` operator in an index means the opposite. E.g:

In [514…
```
a = np.random.random((3,4))
a
```

Out[514…
```
array([[ 0.15776221,  0.43247282,  0.49855915,  0.57465768],
       [ 0.14101421,  0.19224792,  0.10055011,  0.85338245],
       [ 0.36397825,  0.10033724,  0.38265009,  0.85358012]])
```

All items greater than 0.8:

In [516…
```
a[a>0.8]
```

Out[516…
```
array([ 0.85338245,  0.85358012])
```

All items that are NOT greater than 0.8:

In [518…
```
a[~(a>8)]
```

```
Out[518… array([ 0.15776221,  0.43247282,  0.49855915,  0.57465768,  0.14101421,
                 0.19224792,  0.10055011,  0.85338245,  0.36397825,  0.10033724,
                 0.38265009,  0.85358012])
```

We can also use `and` , `or` , ...

```
In [520… a[(a<0.3) | (a>0.8)] # All items that are less than 0.3 or greater than 0.8
```

```
Out[520… array([ 0.15776221,  0.14101421,  0.19224792,  0.10055011,  0.85338245,
                 0.10033724,  0.85358012])
```

Numpy also supports some special prices:

```
In [521… np.inf # infinite
```

```
Out[521… inf
```

```
In [522… np.inf > 100000000
```

```
Out[522… True
```

E.g:

```
In [525… np.array([1])/np.array([0])
```

```
/Users/alexandroskanterakis/anaconda3/envs/arkalos/lib/python3.6/site-packa
ges/ipykernel_launcher.py:1: RuntimeWarning: divide by zero encountered in
true_divide
  """Entry point for launching an IPython kernel.
```

```
Out[525… array([ inf])
```

```
In [524… np.array([-1])/np.array([0])
```

```
/Users/alexandroskanterakis/anaconda3/envs/arkalos/lib/python3.6/site-packa
ges/ipykernel_launcher.py:1: RuntimeWarning: divide by zero encountered in
true_divide
  """Entry point for launching an IPython kernel.
```

```
Out[524… array([-inf])
```

There is also the special value `nan` (Not a Number)

```
In [529… np.nan
```

```
Out[529… nan
```

The `np.isnan` and `np.isinf` functions return `True` / `False` respectively and can be used to "remove" these values from a table, or to replace them with another value:

```
In [3]: a=np.array([1,2,3,np.nan,4, np.nan,5])
        a
```

```
Out[3]: array([ 1.,  2.,  3., nan,  4., nan,  5.])
```

```
In [4]: a[~np.isnan(a)]
```

```
Out[4]: array([1., 2., 3., 4., 5.])
```

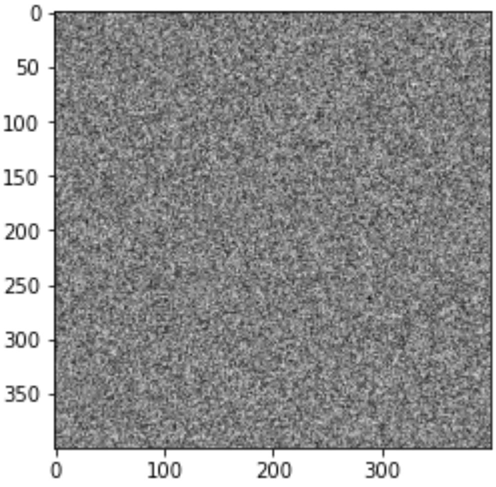
```
In [5]: a[np.isnan(a)] = -1
        a
```

```
Out[5]: array([ 1.,  2.,  3., -1.,  4., -1.,  5.])
```

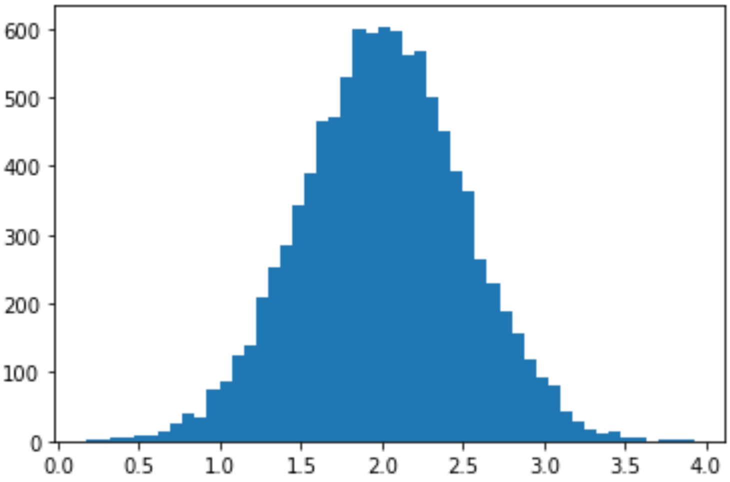
numpy is supported by matplotlib and all python science libraries:

```
In [6]: import matplotlib.pyplot as plt
```

```
In [7]: # Create an image from a 2D array (or else from a bitmap)
        plt.imshow(np.random.random((400,400)), cmap='gray')
        plt.show()
```



```
In [8]: # Histogram:
        mu, sigma = 2, 0.5
        v = np.random.normal(mu,sigma,10000)
        plt.hist(v, bins=50)
        plt.show()
```



## Linear algebra

The inverse of a table:

```
In [563…  a = np.random.random((3,3))
          a
```

```
Out[563…  array([[ 0.38449346,  0.29866245,  0.50693455],
                 [ 0.27030156,  0.94599582,  0.03163946],
                 [ 0.04847714,  0.49405979,  0.43031744]])
```

```
In [564…  np.linalg.inv(a)
```

```
Out[564… array([[ 2.43620688,  0.75888255, -2.92576581],
                [-0.71435503,  0.87677738,  0.77707859],
                [ 0.54572215, -1.09214466,  1.7612799 ]])
```

```
In [565… np.dot(a,np.linalg.inv(a))
```

```
Out[565… array([[  1.00000000e+00,   5.52820098e-17,  -9.52376299e-17],
                [ -2.08616481e-16,   1.00000000e+00,   9.61564695e-17],
                [ -4.59407761e-17,  -6.90990097e-17,   1.00000000e+00]])
```

```
In [566… a = np.array([[3,5], [0,4]])
         np.dot(a,np.linalg.inv(a))
```

```
Out[566… array([[ 1.,  0.],
                [ 0.,  1.]])
```

The unit table (I)

```
In [567… np.eye(4)
```

```
Out[567… array([[ 1.,  0.,  0.,  0.],
                [ 0.,  1.,  0.,  0.],
                [ 0.,  0.,  1.,  0.],
                [ 0.,  0.,  0.,  1.]])
```

```
In [574… arr = np.array([[1, 2], [3, 4]])
```

The determinant of a table:

```
In [576… np.linalg.det(arr)
```

```
Out[576… -2.0000000000000004
```

For an array whose determinant is zero it will result in a LinAlgError Error:

```
In [8]: arr = np.array([[3, 2], [6, 4]])
        np.linalg.inv(arr)
```

```
---------------------------------------------------------------------------
LinAlgError                               Traceback (most recent call last)
<ipython-input-8-8e7f61226c1e> in <module>
      1 arr = np.array([[3, 2], [6, 4]])
----> 2 np.linalg.inv(arr)

<__array_function__ internals> in inv(*args, **kwargs)

~/anaconda3/lib/python3.8/site-packages/numpy/linalg/linalg.py in inv(a)
    544     signature = 'D->D' if isComplexType(t) else 'd->d'
    545     extobj = get_linalg_error_extobj(_raise_linalgerror_singular)
--> 546     ainv = _umath_linalg.inv(a, signature=signature, extobj=extobj)
    547     return wrap(ainv.astype(result_t, copy=False))
    548

~/anaconda3/lib/python3.8/site-packages/numpy/linalg/linalg.py in _raise_li
nalgerror_singular(err, flag)
     86
     87 def _raise_linalgerror_singular(err, flag):
---> 88     raise LinAlgError("Singular matrix")
     89
     90 def _raise_linalgerror_nonposdef(err, flag):

LinAlgError: Singular matrix
```

## Data Saving anf Loading

Numpy has its own format for data storage:

In [42]:

```
A = np.random.random((2,3))
A
```

Out[42]:

```
array([[0.25755066, 0.45975957, 0.10701219],
       [0.03233526, 0.39741997, 0.29875754]])
```

In [43]:

```
np.save('my_data.npy', A)
```

In [44]:

```
! ls -l my_data.npy
```

```
-rw-r--r--  1 admin  staff  176 Apr 16 00:02 my_data.npy
```

In [ ]:

```
```

In [590…

```
B = np.load('my_data.npy')
B
```

Out[590…

```
array([[ 0.00116433,  0.0435687 ,  0.87706621],
       [ 0.90810222,  0.15021207,  0.42381173]])
```

# Scipy: high-level scientific computing

Important scipy routines

- File input / output: **scipy.io**
- Special functions: **scipy.special**
- Linear algebra operations: **scipy.linalg**
- Fast Fourier transforms: **scipy.fftpack**
- Optimization and fit: **scipy.optimize**
- Statistics and random numbers: **scipy.stats**
- Interpolation: **scipy.interpolate**
- Numerical integration: **scipy.integrate**
- Signal processing: **scipy.signal**
- Image processing: **scipy.ndimage**

Routines | Description

------------- | -------------

**scipy.cluster** | Vector quantization / Kmeans

**scipy.constants** | Physical and mathematical constants

**scipy.fftpack** | Fourier transform

**scipy.integrate** | Integration routines

**scipy.interpolate** | Interpolation

**scipy.io** | Data input and output

**scipy.linalg** | Linear algebra routines

**scipy.ndimage** | n-dimensional image package

**scipy.odr** | Orthogonal distance regression

**scipy.optimize** | Optimization

**scipy.signal** | Signal processing

**scipy.sparse** | Sparse matrices

**scipy.spatial** | Spatial data structures and algorithms

**scipy.special** | Any special mathematical functions

**scipy.stats** | Statistics

## Example: Linear algebra with scipy:

In [593…

```
from scipy import linalg
```

In [594…

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

**LU** factorization

In [597…

```
P, L, U = linalg.lu(arr)
```

In [598…

```
# Validation
from scipy import allclose, diag, dot
allclose(arr, P.dot(L.dot(U)))
```

Out[598…

```
True
```

**QR** factorization

In [599…

```
Q, R = linalg.qr(arr)
```

In [600…

```
# Validation
allclose(arr, Q.dot(R))
```

Out[600…

```
True
```

**SVD** factorization

In [601…

```
S, V, D = linalg.svd(arr)
```

In [602…

```
# Validation
allclose(arr, S.dot(diag(V)).dot(D))
```

Out[602…

```
True
```

Calculation of eigenvalues and eigenvectors

In [604…

```
eigvals, eigvecs = linalg.eig(arr)
```

In [606…

```
eigvals
```

Out[606…

```
array([-0.37228132+0.j,  5.37228132+0.j])
```

In [607…

```
eigvecs
```

Out[607…

```
array([[-0.82456484, -0.41597356],
       [ 0.56576746, -0.90937671]])
```