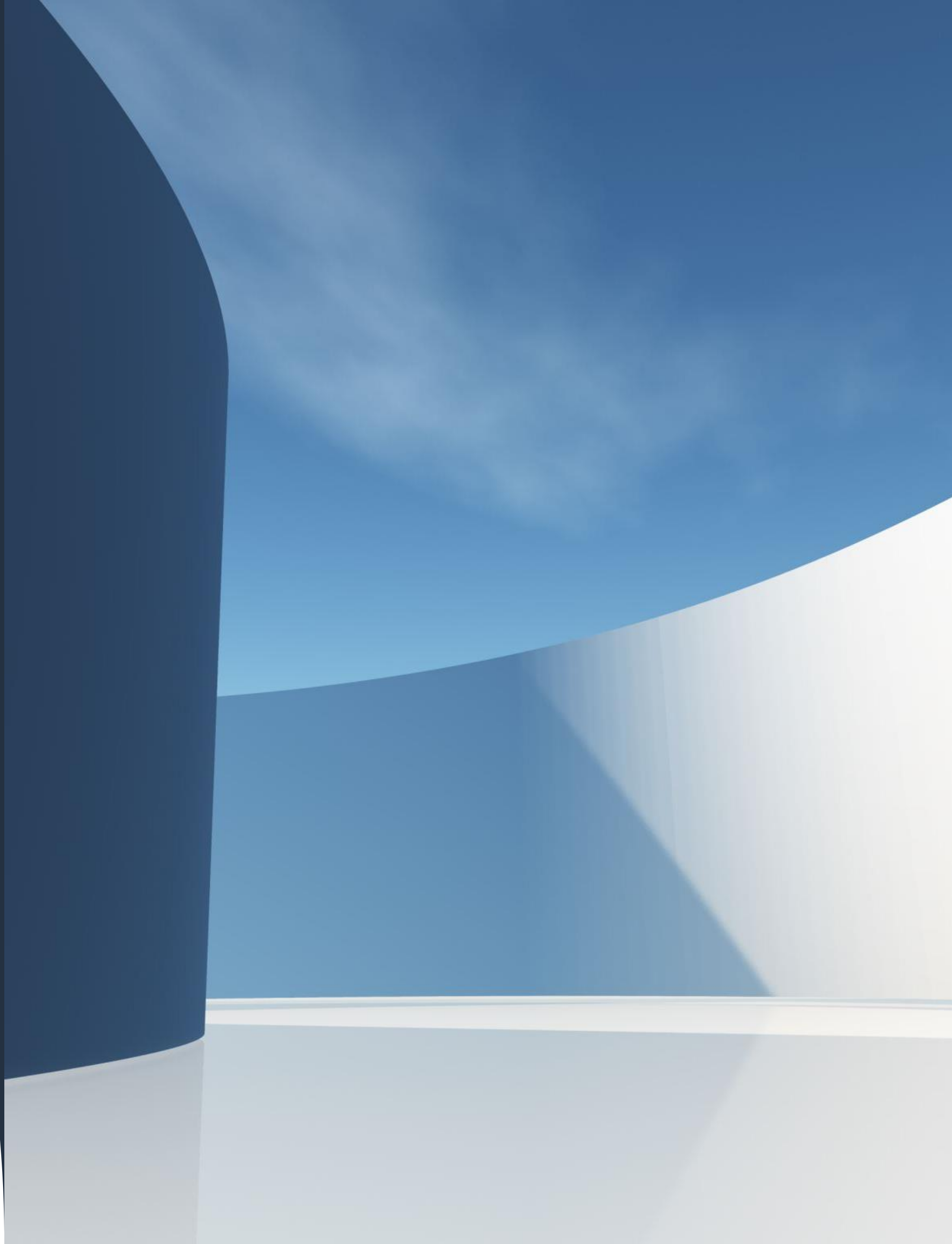
Artificial Intelligence with Python

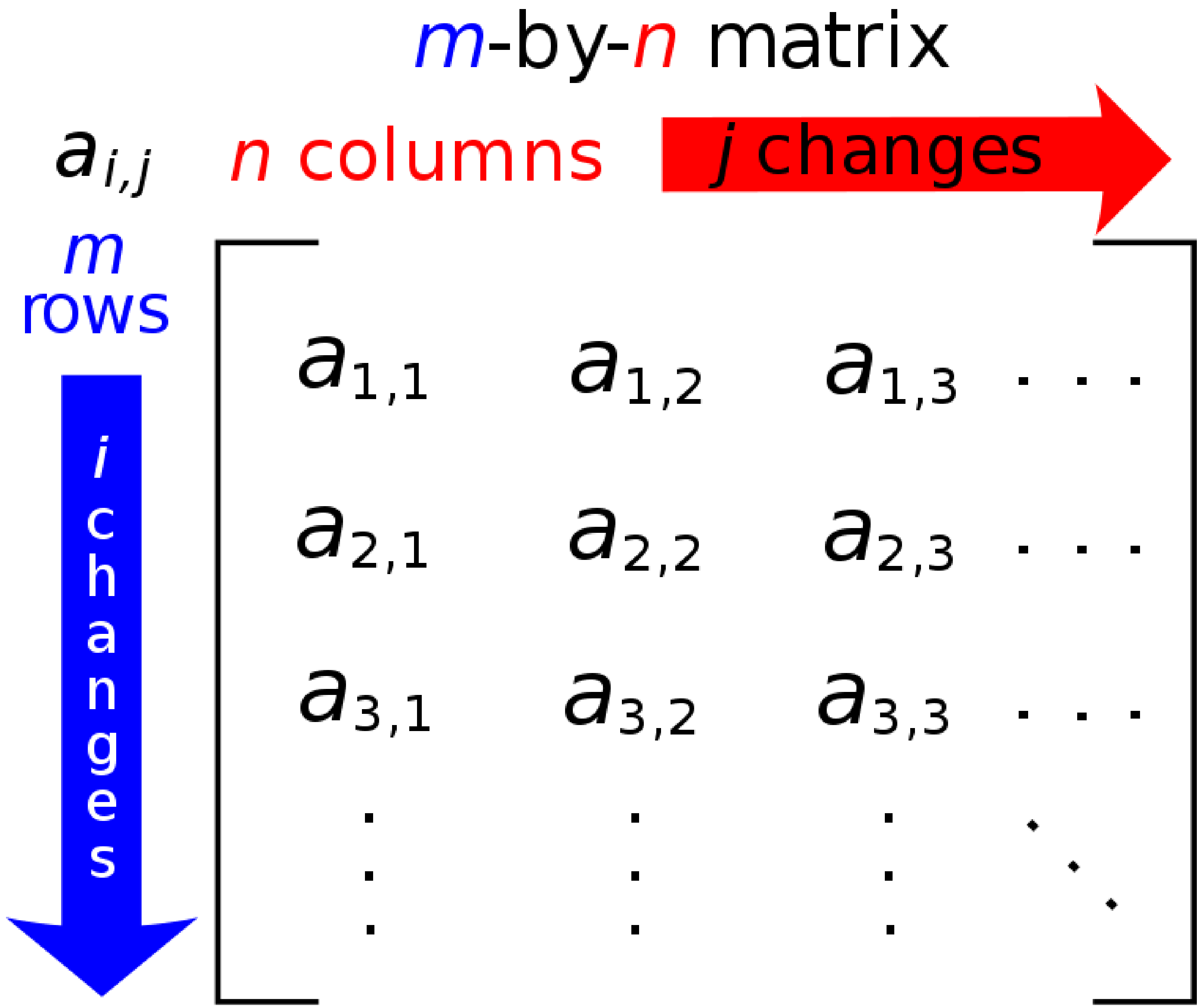
Module 2



Arrays, Vectors, Tables, Matrices

In data analysis, artificial intelligence applications, and other numerical computation, it is common to gather numerical information into vectors and matrices in addition to lists.

- Vectors and matrices are actually mathematical terms related to linear algebra.
- A vector can be thought of as a one-dimensional collection of numbers like a list. Strictly speaking, vectors can be either horizontal vectors or vertical vectors (in mathematics), but in this course it is better to think of them as only one-dimensional lists.
- Programmers usually talk about arrays. Arrays and matrices are usually two-dimensional, meaning that they contain information in horizontal rows and vertical columns.


m-by-n matrix
$a_{i,j}$
n columns
j changes
m rows
i changes
$a_{1,1}$ $a_{1,2}$ $a_{1,3}$ $\cdots$
$a_{2,1}$ $a_{2,2}$ $a_{2,3}$ $\cdots$
$a_{3,1}$ $a_{3,2}$ $a_{3,3}$ $\cdots$
$\vdots$ $\vdots$ $\vdots$ $\ddots$

Numpy and Arrays

- The basic data structure in Numpy is an array.
- It can be a one-dimensional or multi-dimensional collection of items.
- In Numpy, dimensions are referred to by the term axis.
- The elements of a one-dimensional table run along the 0 axis.
- The horizontal rows of a two-dimensional table correspond to axis 0 and the vertical columns to axis 1.
- The same idea is refined into three-dimensional and multi-dimensional tables.

Array is created with the array () command, which passes items as a list (one-dimensional table) or as a list of lists (two-dimensional and multi-dimensional tables). E.g.

```
import numpy as np
x = np.array([1,3,4,5])
A = np.array([[1,3],[4,5]])
```

Arrays in Numpy has a shape, which indicates the number of elements in the table along each axis as a plural (tuple). E.g.

```
np.shape(x)
np.shape(A)
```

It is not appropriate to create large arrays with square brackets. Instead, arrays can be easily created to the desired size with the functions zeros (), ones (), and full (), to which information about the size of the array is conveyed as a plural. E.g.

```
Z = np.zeros(5)
print(Z)
np.shape(Z)
Z2 = np.zeros((4,5)) # notice doulbe pracets
print(Z2)
np.shape(Z2)
Y = np.ones((2,3))
print(Y)

F=np.full((7,8),11)
```

Evenly spaced numbers can be easily generated with the linspace () and arange () functions. linspace () works by passing an initial value, an end value, and the number of elements to it. E.g.

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

print(x)
```

The result is thus 10 elements between the start and end values, including the endpoints. arange () in turn takes the step length as the third parameter. E.g.

```
x2 = np.arange(0,5,0.2)

print(x2)
```

Note that an array created with arange () does not reach a final value (so it works like range ()).

- Random integers between [a, b] can be generated with the command randint (), i.e

```
a = 1
b = 6
amount = 50

nopat = np.random.randint(a,b+1,amount)
```

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

Note how the endpoint is not included here, so the second parameter must be b + 1. Normally distributed random numbers are obtained with the command randn (n), where n indicates the amount of numbers to be generated.

```
x = np.random.randn(100)
```

```
[ 0.75948121 -1.52614652  0.36108226 -0.122655   -0.42378865 -0.01338113
 -0.86468659 -0.38296704  0.33904075  0.39693655  0.12580988  0.42805291
 -0.61821592 -1.21607352 -0.77620632 -0.88161925 -0.19942423  0.72449656
  0.34832248  0.55305657 -2.18011439  0.81765234 -0.78908818  1.04298337
  0.50500826 -0.07476364  0.81507621 -1.46679423  0.75536542 -0.58915783
  0.96053591  0.44425606 -0.61107295  0.06019584  1.41008019 -1.13399064
 -0.01682138  0.44117592 -0.26530648 -0.95422878 -0.88450354  0.78258495
 -1.40766456  0.40597932  1.40104541  0.32753598 -0.12414173 -0.47956185
  0.40251508  1.27251447 -0.42750484 -1.29580338  1.30635906 -1.2830698
 -0.90860039  1.3933708   1.44960961 -0.32023353  1.43133604  0.52130642
  0.47430428  0.16917653  0.21148149 -1.85573734 -1.5604202   0.52706713
  0.97407354  0.68315589 -1.20535152  1.86570328  0.97047759 -0.32282353
 -1.92214446 -0.48321172 -0.99713635 -0.53261058  0.95324937 -0.79555131
 -1.12878909 -1.71960682 -0.89614207  1.24103384  0.52265145 -0.44354155
  0.76444339  0.50327258  0.03262767  0.94766783  0.75505687 -0.4860571
 -1.2489595  -0.25964536  0.19906089 -0.70021928 -1.09243943 -0.97166677
 -1.76913334  0.23251245 -1.77293583 -1.45628282]
```

The random () command produces random numbers evenly distributed over a semi-open interval [0.0, 1.0). So these are decimal numbers. E.g.

```
x = np.random.random(10)
```

```
[0.37958998 0.86159841 0.40059515 0.33304987 0.23739153 0.92304853
 0.68160485 0.37723005 0.77229061 0.95467812]
```

The **randint** and **randn** functions are both part of the NumPy library in Python and are used to generate random numbers, but they serve different purposes:

1. **numpy.random.randint**: This function generates random integers from a specified low (inclusive) to a specified high (exclusive). It is used to generate random integers within a specified range. For example, **np.random.randint(1, 10)** would generate a random integer between 1 and 9.
2. **numpy.random.randn**: This function generates an array of random numbers drawn from a standard normal distribution (mean = 0, standard deviation = 1). It is used to generate random numbers from a normal distribution with a mean of 0 and a standard deviation of 1. The shape of the output array is determined by the arguments passed to the function. For example, **np.random.randn(3, 3)** would generate a 3x3 array of random numbers drawn from a standard normal distribution.

The numbers of dimensions and elements in the table are determined by the attributes ndim and size. E.g.

```
x.size
x.ndim
A.size

A.ndim
```

You can change the format of the table with the reshape (n, m) command, where n and m represent the new format of the table. However, the deformation must not change the number of embryos. E.g.

```
A = np.array([1,2,3,4,5,6,7,8,9,10,11,12])
np.shape(A)
A.reshape(3,4)
A.reshape(2,3,2)

A.reshape(6,2)
```

As the shape changes, it is important to keep in mind the order that the numpy uses in the transformation. The conversion is done on a row major order basis.
You can repeat a row, column, or any table multiple times with the repeat () command. E.g.

```
A = np.repeat([[1,2,3]],4,axis=0)

B = np.repeat([[1],[2],[3]],3,axis=1)
```

Care should be taken when copying tables as shown in the following example

```
A = np.array([1,2])
B = A
B[0] = 99
print(B)
```

The first item in Table A has also changed! To prevent this, follow the steps below to make a copy

```
B = A.copy()
```

In this case, A and B are independent tables whose changes are not reflected in each other.

Cutting Matrices (indexing)

Cutting a one-dimensional array is done in the usual way with square brackets by indexing.
Remember that places are indexed from zero!
The two-dimensional array is indexed (cut) by two indexes within one square bracket:

```
A = np.array([[1,2,3],[4,5,6]])
print(A[0,0])
print(A[0,1])
```

Cutting Matrices (indexing)

The three-dimensional array is cut with three indexes, respectively.
A negative index works in the same way as with lists, i.e. it counts from the end.
A colon is used to cut an entire row or column. E.g.
A[:,0] # fisrt column, ":"reads all rowsA[0,:] # first row, ":" reads all columns
The colon can also be used to create index spaces with the start: end: step syntax,
so that the intersection no longer targets the end.
 The index spacing can be used to index both rows and columns.
E.g.

```
A = np.array([1,2,3,4,5,6,7,8,9])
A[0:6:1]

A[0:6:2]
```

Items can be updated with a combination of an signing statement and a cut. E.g.

```
A = np.array([[1,2,3],[4,5,6]])
A[0,0] = 17

A[1,:] = [11,12,13]
```

Tables can be stacked with the vstack () command. E.g.

```
new = np.vstack((A,A))
```

This method can therefore be used to add horizontal rows to the beginning or end of a table. However, stackable tables must be compatible in shape. Horizontal stacking connects tables side by side. E.g.

```
new2 = np.hstack((A,A))
```

Deleting a row and column is done with the delete command, to which information about the index of the row / column to be deleted and the dimension of the deletion operation must be passed (0 = for rows, 1 = for columns).

```
B = np.delete(A,[0],0)
C = np.delete(A,[1],1)
```

A single item cannot be removed as it would make holes in the matrix.
The table can be traversed (iterated) in many different ways. For example, one line at a time

```
A = np.array([1,2,3,4,5,6])
A = A.reshape(2,3)
n,m = np.shape(A)
for i in range(n):

print("Row",i,"is",A[i,:])
```

Column by column

```
for j in range(m):

print("Column",j,"is",A[:,j])
```

Traversing elements of a matrix

```
for i in range(n):
for j in range(m):
print("Element",i,j,"is",A[i,j])
```

Another way to iterate the table item by item is to use a single iteration structure.

This is especially useful for high-dimension arrays

```
for a in np.nditer(A):
print(a)
```

Notice how here again the order goes to the lines above (dimension / axis 0).

In Python, **np.nditer()** is a function provided by the NumPy library that allows you to iterate over elements of a NumPy array in a flexible and efficient manner, regardless of its shape or dimensionality.

Calculations with Matrices

As with regular numbers, calculations can also be performed on matrices. Addition and subtraction work on the tables item by item, ie the counterparts in the same place in the two matrices are added together (or subtracted from each other). Note that the list structure in Python does not support these kinds of vectorized calculations, Numpy library is needed. E.g.

```
A = np.array([[1,2],[3,4]])
B = np.array([[3,9],[4,-1]])
A+B

A-B
```

Multiplication and division also work with the same idea.

```
A*B
A/B
```

This way, you can create a large matrix that contains the desired number for each item (or use the full () command). E.g.

```
T = 5*np.ones((10,10))
```

These calculations, of course, require that the matrices A and B are the same size (shape the same) or that the other is a mere number. E.g.

```
A-1
B+2
```

succeed without the use of ones (). The increase in power by item is done with the command

```
A**2
```

There is also a completely unique multiplication (matrix product, matrix multiplication) for matrices, which does not work element by element but corresponds to the concept of multiplication in linear algebra. The matrix product AB is defined if the number of columns in A is the same as the number of rows in B (the exact definition can be found in Wikipedia, for example).
Numpy's matmul () command or operator @ performs this multiplication.
E.g.

```
np.matmul(A,B) # or A @ B
```

For one-dimensional tables (i.e. vectors), dot () produces the point product (scalar product) of the vectors. In that case, the compatibility of the forms of income factors is not taken into account.

```
x.dot(x)
```

If **x** is a one-dimensional array (vector), **x.dot(x)** calculates the dot product of **x** with itself. The dot product of two vectors is the sum of the products of their corresponding elements. For example, if **x** is **[1, 2, 3]**, then **x.dot(x)** would be 1×1+2×2+3×3=141×1+2×2+3×3=14.

It is also possible to calculate the matrix product between the matrix and the vertical vector, in which case an elementary product would not even be possible. E.g.

```
b = np.array([[5],[7]])
np.matmul(A,b)
```

So note that matmul () / dot () does not compute an elementary product but something completely different (related to a linear algebra).

In addition to the basic calculations, the application of elementary functions to the matrix is done item by item. E.g.

```
np.sqrt(A) # the square root of each item
np.sin(A)
np.cos(A)
np.tan(A)
np.log(A)
np.exp(A)
np.log10(A)

np.log2(A)
```

It is important to use the numpy library and not the math library, which also has elementary functions, but they do not work with matrices.

Comparison operators also target matrices by item. E.g.

```
A > 1

A <= 0
```

The result is a matrix formed from the truth values True and False, to which the functions any () / all () can be assigned,
which tells whether one of the truth values is True and, respectively,
whether all the values are True. E.g.

```
np.all(A>1)
np.any(A>1)
```

Operators can be used at the same time as the cut to select only items from a table that fulfill
any of the conditions. E.g.

```
A = np.array([1,2,3,4,5,6,7])

B = A[A>3]
```

This is based on the fact that the cut can be made using the list as indexes.

Let's go back briefly to the matrix product. Consider a pair of equations formed by two equations

$$\begin{cases} 2x + y = 11 \\ -4x + 3y = 3 \end{cases}$$

This can be represented by (vertical) vectors in the form

$$\begin{pmatrix} 2x+y \\ -4x+3y \end{pmatrix} = \begin{pmatrix} 11 \\ 3 \end{pmatrix}$$

when the equality (comparison) of the vectors is handled item by item. However, it now appears that the left side of the previous formula can be written as a multiplication of the matrix and the vector, in which case the equation is

$$\begin{pmatrix} 2 & 1 \\ -4 & 3 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} 11 \\ 3 \end{pmatrix}$$

In practice, the solution of the group of equations is not done through the inverse matrix because it is quite a heavy operation for large matrices. Instead, other methods are used, we won’t go into details here.

However, it is noted that numpy can solve a group of equations with the command solve ().

For example, in the case of the previous situation

```
A = np.array([[2,1],[-4,3]])
b = np.array([11,3])

X = np.linalg.solve(A,b)
```

This code will output the solution for *x*, which is the array **[3. 5.]**, indicating that x=3 and y =5.

Statistical Functions

Many statistical indicators can be calculated from the numerical values in the table. The sum is calculated with the command sum (), which can optionally be given as the second argument the number of the axis over which the sum is calculated. E.g.

```
A = np.array([[1,2,3],[4,5,6]])
np.sum(A)
np.sum(A,0) # column sums, summed along rows

np.sum(A,1) # row sums, summed along the columns
```

The product is calculated similarly with the prod () command.

```
np.prod(A)
np.prod(A,0)

np.prod(A,1)
```

There are also cumulative variants cumsum () and cumprod () for sum and result.

There are also cumulative variants cumsum () and cumprod () for sum and result.
The largest and smallest elements are found with the min () / max () commands. E.g.

- np.min(A)
- np.min(A,0)
- np.min(A,1)
- np.max(A)
- np.max(A,0)
- np.max(A,1)

The average of the elements is calculated with the function mean (), which can also be assigned to a specific axis.

```
np.mean(A)

np.mean(A,0)

np.mean(A,1)
```

The median, ie the "middle" value of the numerical values, is determined by the function median ()

- np.median(A)
- np.median(A,0)
- np.median(A,1)

The variance and its standard root deviation measure how far the numbers are from their mean. For these, there are functions var () and std ()

- np.std(A)
- np.std(A,0)
- np.std(A,1)
- np.var(A)
- np.var(A,0)
- np.var(A,1)

Drawing Graphs

- Let’s consider drawing (mathematical) graphs with Python. To do this, install the matplotlib library by issuing a command below at the command prompt
- pip install matplotlib
- Numeric libraries / programs (including python) basically draw individual data points on the screen and connect them with (different) lines. The result is a so-called broken line. These points must be created before you can draw. E.g.

```
import numpy as np

import matplotlib.pyplot as plt

x = [1,2,3,4]

y = [1,4,9,16]

plt.plot(x,y)

plt.show()
```

Note that only calling the show () function will display the graph.

The above about drawing also applies to drawing graphs of mathematical functions. That is, first there must be a (suitably dense) lattice point and the corresponding values of the function formed. The graph is still a dashed line, but the density of the score gives "roundness" to the graph. E.g.

```
x = np.linspace(0,7,100)

y = np.sin(x)

plt.plot(x,y)

plt.show()
```

You can add titles and captions to your graph with the name telling their purpose. E.g.

```
plt.title('Title')

plt.xlabel('x')

plt.ylabel('y')

plt.show()
```

These commands can be given additional attributes to control the appearance of the text; eg fontsize = 12, color = "red". A comprehensive list can be found online

https://matplotlib.org/3.7.1/api/text_api.html

The color of the line can be controlled with the optional arguments of the plot () command. E.g.

```
plt.plot(x,y,"g")
```

Here, "g" means green; other ready-made options are k = black, b = blue, r = red, c = cyan, m = magenta, y = yellow, w = white. More specifically, the color can be set with RGB color codes, e.g.

```
plt.plot(x,y,color="#4b0082")
```

where the color code follows the way HTML defines colors as hexadecimal numbers.

In the same way, you can also control the plot marker, the most important of which are o = circle, x, +, s = square, *, D = diamond. E.g.

```
plt.plot(x,y,"o")
```

The line style can be adjusted with the following linestyle parameter options: "-" solid line, "-" dashed line, ":" dotted line, "-." dash point

The line thickness can be set with the linewidth parameter. E.g.

```
plt.plot(x,y,linewidth=2,linestyle="--")
```

Another way to display multiple graphs in the same figure window is to divide the window into subplots.

```
plt.subplot(1,2,1) # 1x2 grid 1. subpicture

plt.plot(x,y)

plt.title("first")

plt.subplot(1,2,2) # 1x2 grid 2. subpicture

plt.plot(x,2*y)

plt.title("second")

plt.suptitle("Common Title")

plt.show()
```

You can draw bar charts with the bar () command. E.g.

```
plt.bar(['2018','2019','2020'],[120000,125000,130000],color="blue")
plt.title("Title")
plt.xlabel("years")
plt.ylabel("Sales")
plt.show()
```

You can make a horizontal column with the barh () command.

In data sciences and machine learning, statistical distributions of data are often studied. These can be quickly illustrated with histograms. E.g.

```
x = np.random.randn(2000)

plt.hist(x,10)

plt.ylabel('frequencies')

plt.show()
```

Scattering pattern refers only to the representation of individual data points with dots without lines. E.g.

```
plt.scatter(x,y)

plt.show()
```

Again, additional quantities can be used to adjust the colors of the scattering pattern, etc.

```
plt.scatter(x,y,color="r",marker="o",label="Points")
```

THANKS