Data Analysis

The 21st century is the age of information. We live in the age of information, which means that almost every aspect of our daily lives generates data.

This data accumulates day by day due to the fact that it is constantly generated by activities:

- business,
- government,
- scientific,
- engineering,
- health,
- social,
- climatic,
- environment.

Data analysis is the activity of inspecting, pre-processing, mining, describing and visualising a given set of data.

The main objective of the data analysis process is to discover the information needed to make decisions. Data analysis offers many approaches, tools and techniques, all of which can be applied in different fields such as business, social science or basic science.

Machine Learning

Machine learning, **machine self-learning** or **machine learning systems** - an area of artificial intelligence devoted to algorithms that improve automatically through experience, that is, exposure to data. Machine learning algorithms build a mathematical model from sample data, called a learning set, to make predictions or decisions without being programmed explicitly by a human to do so. Machine learning algorithms are used in a wide variety of applications, such as spam protection (filtering Internet messages for unwanted correspondence), and image recognition, where it is difficult or impractical to develop conventional algorithms to perform the needed tasks.

General applications of machine learning include:

- **the analysis and use of huge databases**, where size, complexity and the requirement for continuous updating prevent non-automated analysis (e.g. in fields such as economics, medicine, chemistry)

- **system adaptation to the environment** through dynamic modification, allowing correct operation under changing conditions (robotics, control systems, manufacturing, data analysis)
- **search and analysis of relationships in large databases** for synthetic representation of information according to given criteria (expert systems, search engines) ***analysis, research and development of very complex problems**, difficult to describe and often without sufficient theoretical models, which are expensive, time-consuming or unreliable to obtain (physics, mathematics)

Basic libraries and tools for data analysis and machine learning in Python

---

Python language libraries for data analysis

Among the most fundamental data analysis libraries in the Python ecosystem are:

- **NumPy** (*Numerical Python*): is a scientific library for handling multidimensional arrays, matrices and methods for efficient processing of mathematical calculations,
- **SciPy** (*Scientific Python*): is a powerful library (using NumPy) for scientific computing that provides modules for solving tasks in linear algebra, integration, ordinary differential equations and signal processing, among others,
- **Pandas**: is a library for data mining and manipulation that offers tabular data structures such as DataFrames and various methods for analysing and modifying data,
- **Scikit-learn** (*Scientific Toolkit for Machine learning*): is a machine learning library that offers many supervised and unsupervised algorithms such as regression, classification, dimension reduction, cluster analysis and anomaly detection,
- **Matplotlib**: is the core data visualisation library and is the base for other visualisation libraries in Python. It offers 2D and 3D charts, graphs and diagrams used for data mining. It works with NumPy and SciPy,
- **Seaborn**: is a Matplotlib-based library that offers easy-to-draw, high-level, interactive and more structured graphs,
- **Plotly**: is a library for data visualisation. It offers high quality interactive graphs such as scatter plots, line plots, bar charts, histograms, boxplots, heat maps and more.

Python language tools to support data analysis and machine learning

In addition, there are a number of tools to support the data analysis process in Python, among which it is worth mentioning:

- **Anaconda**: is a distribution of the Python and R programming languages for scientific computing. Provides a package manager to manage libraries and software supporting the use of Python for extensive data analysis,
- **Jupyter Notebook**: is a browser-based, interactive computing environment for creating notebook-like documents. It allows you to combine text data in markdown format with code in Python and presents the results of its scripts,
- **Spyder**: is a free and open-source environment for the Python language supporting advanced features of a comprehensive development tool and scientific package with data mining, interactive execution and visualization capabilities,
- **Google Colab (Colaboratory)**: an online remote environment based on Jupiter Notebook, enabling GPU/TPU data analysis via a web browser. The basic version is free and only requires a Google account.

NumPy library

A fundamental library for numerical calculations in Python. It is the core for other libraries used in data analysis and machine learning.

Library website: https://numpy.org/

Documentation: https://numpy.org/doc/stable/

Installation: `pip install numpy`

NumPy Library Basics

Import NumPy library

In [89]:
```
import numpy as np
np.__version__
```

Out[89]:
```
'1.26.4'
```

One-dimensional arrays

The basic object of the NumPy library is an array, which is a series of homogeneous elements (of the same type).

The arrays created are of class ndarray.

The main advantages of arrays from the NumPy library include:

- take up less memory than Python lists,
- thanks to the homogeneity of the elements and the indication of their type, they are more efficient in processing

In [90]:
```
x = np.array([1, 3, 4, 5, 7])
x
```

Out[90]:
```
array([1, 3, 4, 5, 7])
```

In [92]:
```
print(x)
type(x)
```

```
[1 3 4 5 7]
```

Out[92]:
```
numpy.ndarray
```

Basic attributes of the ndarray class:

ndim - number of array dimensions

shape - array shape

size - number of array elements

dtype - data type

In [93]:
```
print(x.ndim)
print(x.shape)
print(x.size)
print(x.dtype)
```

```
1
(5,)
5
int64
```

In [94]:
```
xf = np.array([1.2, 1.5, 2.4])
print(xf)
print(xf.dtype)
```

```
[1.2 1.5 2.4]
float64
```

Two-dimensional arrays

Two-dimensional arrays are among the more common in data analysis.

In [95]:
```
array2d = np.array([[1, 4], [2, 6]])
print(array2d)
print(array2d.ndim)
print(array2d.shape)
print(array2d.size)
```

```
[[1 4]
 [2 6]]
2
(2, 2)
4
```

In [96]:
```
array2d_2 = np.array([[2, 5, 3],[4, 1, -5]])
print(array2d_2)
print(array2d_2.ndim)
print(array2d_2.shape)
print(array2d_2.size)
```

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

Three-dimensional arrays

Three-dimensional arrays are often used for image analysis and processing.

In [99]:
```
array3d = np.array([
    [
        [1, 3, 4],
        [-2, 5, -1]
      ],
    [
        [5, 2, 1],
        [4, 5, 8]
      ]
    ])
```

In [100…

```
print(array3d)
print(array3d.ndim)
print(array3d.shape)
print(array3d.size)
```

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

 [[ 5  2  1]
  [ 4  5  8]]]
3
(2, 2, 3)
12
```

Data types

The NumPy library supports many more numeric types than Python.

Table of available numerical types in NumPy

Data type	Details
bool	This is a Boolean type that stores 1 bit and accepts a value of True or False.
inti	Depending on the platform, integers can be either int32 or int64.
int8	It stores 1 byte whose values range from -128 to 127.
int16	Stores integers in the range -32768 to 32767.
int32	Stores integers in the range -2^{31} to $2^{31}-1$.
int64	Stores integers in the range -2^{63} to $2^{63}-1$.
uint8	Stores unsigned integers in the range 0 to 255.
uint16	Stores unsigned integers in the range 0 to 65535.
uint32	Stores unsigned integers in the range 0 to $2^{31}-1$.
uint64	Stores unsigned integers in the range 0 to $2^{64}-1$.
float16	Stores real numbers of half precision, sign bit with 5 bits of exponent and 10 bits of mantissa.
float32	Stores single-precision real numbers, sign bit with 8 bits of exponent and 23 bits of mantissa.
float64 or float	Stores double-precision real numbers, sign bit with 11 bits of exponent and 52 bits of mantissa.
complex64	A complex number stores two 32-bit values: real and imaginary number.
complex128 or complex	A complex number stores two 64-bit values: real and imaginary number.

In [101…
```
T = np.array([1,2,3,4])
T.dtype
```

Out[101…
```
dtype('int64')
```

In [102…
```
T = np.array([1.0, 2, 3])
T.dtype
```

Out[102…
```
dtype('float64')
```

In [103…
```
T = np.array([1.2, 2.3, 3.7], dtype='int')
print(T)
T.dtype
```

```
[1 2 3]
```

Out[103…
```
dtype('int64')
```

In [104…
```
T = np.array([1, 2, 3], dtype='float16')
print(T)
T.dtype
```

```
[1. 2. 3.]
```

Out[104… dtype('float16')

In [105…
```
T = np.array([1, 2, 3], dtype='complex64')
print(T)
T.dtype
```

```
[1.+0.j 2.+0.j 3.+0.j]
```

Out[105… dtype('complex64')

In [107…
```
T = np.array([1, 2, True, 0], dtype=np.bool_)
print(T)
T.dtype
```

```
[ True  True  True False]
```

Out[107… dtype('bool')

Changing the data type of an array is possible using the `astype` function.

In [108…
```
T = T.astype(np.float_)
print(T)
T = T.astype(np.bool_)
print(T)
```

```
[1. 1. 1. 0.]
[ True  True  True False]
```

Array creation functions

The NumPy library provides a number of functions for creating arrays, these include:

- `zeros` - function creates an array of specified dimensions filled with zeros,
- `ones` - function creates an array of specified dimensions filled with ones,
- `full` - function creates an array of specified dimensions filled with a certain value,
- `arange` - function creates an array from the specified range with a set step,
- `linespace` - function creates an array of numbers evenly spaced over a specified interval,
- `eye` - function produces a unit matrix,

- `np.random` - functions creating randomised arrays.

In [111…
```
z = np.zeros(shape=(3,4))
print(z)
z = np.zeros(shape=(5,4), dtype=np.int_)
print(z)
```

```
[[0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]]
[[False False False False]
 [False False False False]
 [False False False False]
 [False False False False]
 [False False False False]]
```

In [110…
```
o = np.ones(shape=(3,4))
print(o)
o = np.ones(shape=(5,4), dtype=np.bool_)
print(o)
```

```
[[1. 1. 1. 1.]
 [1. 1. 1. 1.]
 [1. 1. 1. 1.]]
[[ True  True  True  True]
 [ True  True  True  True]
 [ True  True  True  True]
 [ True  True  True  True]
 [ True  True  True  True]]
```

In [112…
```
T = np.full(shape=(4,2,3), fill_value=10)
print(T)
```

```
[[[10 10 10]
  [10 10 10]]

 [[10 10 10]
  [10 10 10]]

 [[10 10 10]
  [10 10 10]]

 [[10 10 10]
  [10 10 10]]]
```

In [115…
```
T = np.arange(10)
print(T)
```

```
T = np.arange(1, 11, dtype=np.float_)
print(T)
T = np.arange(0, 101, 10)
print(T)
T = np.arange(0, 1, 0.01)
print(T)
```

```
[0 1 2 3 4 5 6 7 8 9]
[ 1.  2.  3.  4.  5.  6.  7.  8.  9. 10.]
[  0  10  20  30  40  50  60  70  80  90 100]
[0.   0.01 0.02 0.03 0.04 0.05 0.06 0.07 0.08 0.09 0.1  0.11 0.12 0.13
 0.14 0.15 0.16 0.17 0.18 0.19 0.2  0.21 0.22 0.23 0.24 0.25 0.26 0.27
 0.28 0.29 0.3  0.31 0.32 0.33 0.34 0.35 0.36 0.37 0.38 0.39 0.4  0.41
 0.42 0.43 0.44 0.45 0.46 0.47 0.48 0.49 0.5  0.51 0.52 0.53 0.54 0.55
 0.56 0.57 0.58 0.59 0.6  0.61 0.62 0.63 0.64 0.65 0.66 0.67 0.68 0.69
 0.7  0.71 0.72 0.73 0.74 0.75 0.76 0.77 0.78 0.79 0.8  0.81 0.82 0.83
 0.84 0.85 0.86 0.87 0.88 0.89 0.9  0.91 0.92 0.93 0.94 0.95 0.96 0.97
 0.98 0.99]
```

In [21]:
```
T = np.linspace(0, 10, num=5)
print(T)
T = np.linspace(0, 10, num=101)
print(T)
```

```
[ 0.   2.5  5.   7.5 10. ]
[ 0.   0.1  0.2  0.3  0.4  0.5  0.6  0.7  0.8  0.9  1.   1.1  1.2  1.3
  1.4  1.5  1.6  1.7  1.8  1.9  2.   2.1  2.2  2.3  2.4  2.5  2.6  2.7
  2.8  2.9  3.   3.1  3.2  3.3  3.4  3.5  3.6  3.7  3.8  3.9  4.   4.1
  4.2  4.3  4.4  4.5  4.6  4.7  4.8  4.9  5.   5.1  5.2  5.3  5.4  5.5
  5.6  5.7  5.8  5.9  6.   6.1  6.2  6.3  6.4  6.5  6.6  6.7  6.8  6.9
  7.   7.1  7.2  7.3  7.4  7.5  7.6  7.7  7.8  7.9  8.   8.1  8.2  8.3
  8.4  8.5  8.6  8.7  8.8  8.9  9.   9.1  9.2  9.3  9.4  9.5  9.6  9.7
  9.8  9.9 10. ]
```

In [22]:
```
T = np.eye(3)
print(T)
T = np.eye(5, k = 1)
print(T)
T = np.eye(5, k = -2, dtype='int')
print(T)
```

```
[[1. 0. 0.]
 [0. 1. 0.]
 [0. 0. 1.]]
[[0. 1. 0. 0. 0.]
 [0. 0. 1. 0. 0.]
 [0. 0. 0. 1. 0.]
 [0. 0. 0. 0. 1.]
 [0. 0. 0. 0. 0.]]
[[0 0 0 0 0]
 [0 0 0 0 0]
 [1 0 0 0 0]
 [0 1 0 0 0]
 [0 0 1 0 0]]
```

Selecting values from an array

Using indexing

The Numpy library allows an array value to be accessed using indexing, which, as in Python, starts from the value 0.

One-dimensional arrays

`A[idx]` - retrieve the value from the array A with index idx

`A[[id1, id2, ..., idx]]` - retrieve values from the array A with indices id1, id2, ..., idx

`A[start:stop:step]` - retrieve values from array A with indices in the range <start;stop) with a given step (negative values indicate indices from the end of the array)

In [116…
```
A = np.arange(30)
print(A)
print(A[4])
print(A[[1,3,5,7,8]])
```

```
[ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23
 24 25 26 27 28 29]
4
[1 3 5 7 8]
```

In [117…
```
print(A[3:5])
print(A[10:20:2])
```

```
print(A[10::2])
```

```
[3 4]
[10 12 14 16 18]
[10 12 14 16 18 20 22 24 26 28]
```

In [118…
```
print(A[::2])
print(A[-2::-2])
print(A[-2:9:-2])
```

```
[ 0  2  4  6  8 10 12 14 16 18 20 22 24 26 28]
[28 26 24 22 20 18 16 14 12 10  8  6  4  2  0]
[28 26 24 22 20 18 16 14 12 10]
```

Multidimensional arrays

`A[d1ids, d2ids, ..., dnids]` - retrieve the values from the array A, where the identifiers d1ids, d2ids, ..., dnids, are the index ranges for the following dimensions (as for one-dimensional arrays)

`:` - (colon) replaces all values for a given dimension

In [119…
```
A = np.arange(25)
print(A)
A = A.reshape((5,5))
print(A)
```

```
[ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23
 24]
[[ 0  1  2  3  4]
 [ 5  6  7  8  9]
 [10 11 12 13 14]
 [15 16 17 18 19]
 [20 21 22 23 24]]
```

In [120…
```
print(A[1])
print(A[[1,3]])
print(A[[1,3], -2:])
```

```
[5 6 7 8 9]
[[ 5  6  7  8  9]
 [15 16 17 18 19]]
[[ 8  9]
 [18 19]]
```

In [121…
```
print(A[::2, :])
print(A[::2, [0,-1]])
```

```
[[ 0  1  2  3  4]
 [10 11 12 13 14]
 [20 21 22 23 24]]
[[ 0  4]
 [10 14]
 [20 24]]
```

Using masking

The NumPy library enables data selection by means of logical conditions, which return a so-called mask indicating which values from the array satisfy the given condition.

The use of `or`, `and`, `xor`, `not` operations requires the use of functions:

- `bitwise_or`
- `bitwise_and`
- `bitwise_xor`
- `bitwise_not`

In [122…
```
A = np.arange(-20,21)
print(A)
```

```
[-20 -19 -18 -17 -16 -15 -14 -13 -12 -11 -10  -9  -8  -7  -6  -5  -4  -3
  -2  -1   0   1   2   3   4   5   6   7   8   9  10  11  12  13  14  15
  16  17  18  19  20]
```

In [30]:
```
print(A > 0)
print(A[A > 0])
```

```
[False False False False False False False False False False False False
 False False False False False False False False False  True  True  True
  True  True  True  True  True  True  True  True  True  True  True  True
  True  True  True  True  True]
[ 1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20]
```

In [123…
```
print(np.bitwise_and(A>0,A<=10))
print(A[np.bitwise_and(A>0,A<=10)])
```

```
[False False False False False False False False False False False False
 False False False False False False False False False  True  True  True
  True  True  True  True  True  True False False False False False
 False False False False False]
[ 1  2  3  4  5  6  7  8  9 10]
```

In [124…
```
print(np.bitwise_or(np.bitwise_and(A>=-10,A<=-5),np.bitwise_and(A>=5,A<=10)))
print(A[np.bitwise_or(np.bitwise_and(A>=-10,A<=-5),np.bitwise_and(A>=5,A<=10))])
```

```
[False False False False False False False False False False  True  True
  True  True  True  True False False False False False False False False
 False  True  True  True  True  True  True False False False False False
 False False False False False]
[-10  -9  -8  -7  -6  -5   5   6   7   8   9  10]
```

Array operations

The NumPy library provides a number of operations on arrays, among which we can point out:

- addition - operator `+` or function `np.add
- subtraction - operator `-` or function `np.substract`
- multiplication - operator `*` or function `np.multiply`
- division - operator `/` or function `np.divide`
- matrix multiplication - operator `@` lub funkcje `np.dot`
- dimensional change - functions `reshape` / `ravel`, .T - matrix transposition
- iterating through the elements of the array - loop `for`

In [125…
```
mA = np.array([[1,2,3],[4,5,6]])
print(mA)
mB = np.array([[3,-2,5],[1,-4,5]])
print(mB)
```

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

Addition of arrays

In [126…
```
print(mA + mB)
print(4 + mB)
print(np.add(mA,2))
```

```
[[ 4  0  8]
 [ 5  1 11]]
[[7 2 9]
 [5 0 9]]
[[3 4 5]
 [6 7 8]]
```

Substraction of arrays

In [127…

```
print(mA - mB)
print(4 - mB)
print(np.subtract(mA,2))
```

```
[[-2  4 -2]
 [ 3  9  1]]
[[ 1  6 -1]
 [ 3  8 -1]]
[[-1  0  1]
 [ 2  3  4]]
```

Multiplication of arrays

In [128…

```
print(mA * mB)
print(2 * mB)
print(np.multiply(mA,5))
```

```
[[  3  -4  15]
 [  4 -20  30]]
[[ 6 -4 10]
 [ 2 -8 10]]
[[ 5 10 15]
 [20 25 30]]
```

Division of arrays

In [129…

```
print(mA / mB)
print(2 / mB)
print(np.divide(mA,5))
```

```
[[ 0.3333 -1.      0.6   ]
 [ 4.     -1.25    1.2   ]]
[[ 0.6667 -1.      0.4   ]
 [ 2.     -0.5     0.4   ]]
[[0.2 0.4 0.6]
 [0.8 1.  1.2]]
```

Changing the dimensions of an array

In [130…
```
A = np.arange(20)
print(A)
print(A.shape)
```

```
[ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19]
(20,)
```

In [132…
```
A = A.reshape(5,4)
print(A)
print(A.shape)
```

```
[[ 0  1  2  3  4  5  6  7  8  9]
 [10 11 12 13 14 15 16 17 18 19]]
(2, 10)
```

In [136…
```
A = A.reshape(2,10)
print(A)
print(A.shape)
```

```
[[ 0  1  2  3  4  5  6  7  8  9]
 [10 11 12 13 14 15 16 17 18 19]]
(2, 10)
```

In [135…
```
A = A.ravel()
print(A)
```

```
[ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19]
```

Iterating over array elements

In [138…
```
A = np.arange(1,17).reshape(4,4)
print(A)
```

```
[[ 1  2  3  4]
 [ 5  6  7  8]
 [ 9 10 11 12]
 [13 14 15 16]]
```

In [139…
```
for row in A:
  print(row)
```

```
[1 2 3 4]
[5 6 7 8]
[ 9 10 11 12]
[13 14 15 16]
```

In [140…
```
for row in A:
  print(row[:2])
```

```
[1 2]
[5 6]
[ 9 10]
[13 14]
```

In [141…
```
for element in A.flat:
  print(element)
```

```
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
```

Core functions in the *NumPy* library

---

mathematical

- `np.power` - exponentiation
- `np.sqrt` - square root
- `np.sin` , `np.sinh` , `np.arcsin` , `np.arcsinh` , `np.cos` , ... - trigonometric

- `np.log` , `np.log10` , `np.log2` - logarithmic
- `np.exp` - exponent

In [142…
```
T = np.array([1,3,6,9])
print(T)
print(np.power(T, 3))
```

```
[1 3 6 9]
[  1  27 216 729]
```

In [143…
```
print(np.sqrt(T))
print(np.sin(np.pi/2))
```

```
[1.     1.7321 2.4495 3.    ]
1.0
```

In [144…
```
print(np.log(T))
print(np.log(np.e))
```

```
[0.     1.0986 1.7918 2.1972]
1.0
```

logical

- `np.greater` , `np.greater_equal` , `np.less` , `np.less_equal` , `np.equal` , `np.not_equal` - greater, greater equal, lesser equal, lesser equal, different
- `np.all` - whether all elements are true?
- `np.any` - whether at least one element is true?

In [146…
```
print(T)
print(np.greater(T, 3))
print(T[np.greater(T, 3)])
print(np.greater_equal(T, 3))
print(T[np.greater_equal(T, 3)])
```

```
[1 3 6 9]
[False False  True  True]
[6 9]
[False  True  True  True]
[3 6 9]
```

In [147…
```
print(np.all(T))
print(np.any(np.zeros((2,2))))
```

```
True
False
```

statistical

- `np.max` , `np.min` - maximum/minimum value
- `np.mean` - average
- `np.median` - median
- `np.std` - standard deviation

In [148…
```
print(T)
print(np.max(T))
print(np.min(T))
```

```
[1 3 6 9]
9
1
```

In [149…
```
print(np.mean(T))
print(np.median(T))
print(np.std(T))
```

```
4.75
4.5
3.031088913245535
```

searching, sorting, counting

- `np.argmax` , `np.argmin` - returns index of maximum/minimum value
- `np.sort` - returns a sorted array
- `np.argsort` - returns the indexes of the sorted array
- `np.count_nonzero` - counts non-zero values

In [152…
```
T = np.array([1,0,-1, 4,-10, 8, 20, 0, 5])
print(T)
print(np.argmax(T))
print(np.argmin(T))
```

```
[  1   0  -1   4 -10   8  20   0   5]
6
4
```

In [155…
```
print(T)
print(np.sort(T))
print(np.argsort(T))
print(-np.sort(-T))
```

```
[  1   0  -1   4 -10   8  20   0   5]
[-10  -1   0   0   1   4   5   8  20]
[4 2 1 7 0 3 8 5 6]
[ 20   8   5   4   1   0   0  -1 -10]
```

In [156…
```
print(np.count_nonzero(T))
```

```
7
```

pseudo-random values

- `np.random.seed` - function to set up a pseudo-random number generator to obtain repeatable results
- `np.random.randn` - function returns the random value(s) for a normal distribution
- `np.random.rand` - function returns a random value between <0;1)
- `np.random.randint` - function returns a random integer(s) from the specified interval
- `np.random.choice` - function returns a random value from the specified set
- `np.random.shuffle` - function randomly distributes the elements of the given set

In [157…
```
np.random.seed(10)
```

In [158…
```
T = np.random.randn(5)
print(T)
T = np.random.randn(3, 4)
print(T)
```

```
[ 1.3316  0.7153 -1.5454 -0.0084  0.6213]
[[-0.7201  0.2655  0.1085  0.0043]
 [-0.1746  0.433   1.203  -0.9651]
 [ 1.0283  0.2286  0.4451 -1.1366]]
```

In [159…
```
T = np.random.rand(5)
print(T)
T = np.random.rand(3, 4)
print(T)
```

```
[0.3733 0.6741 0.4418 0.434  0.6178]
[[0.5131 0.6504 0.601  0.8052]
 [0.5216 0.9086 0.3192 0.0905]
 [0.3007 0.114  0.8287 0.0469]]
```

In [160…
```
T = np.random.randint(1,11, size=(3,4))
print(T)
```

```
[[2 5 1 9]
 [6 5 8 9]
 [9 3 7 3]]
```

In [163…
```
number = np.random.choice([1, 2, 3, 4])
print(number)
character = np.random.choice(["A", "B", "C", "D"])
print(character)
```

```
4
B
```

In [166…
```
data = np.arange(0,10)
print(data)
np.random.shuffle(data)
print(data)
```

```
[0 1 2 3 4 5 6 7 8 9]
[3 8 6 4 7 9 0 5 2 1]
```

Examples of NumPy library applications

Among the many possible applications of the NumPy library are the following:

- Euclidean norm in space R^n
- Calculating distance between points in space R^n
- Matrix multiplication
- Matrix determinant, matrix trace, inverse matrix
- Solving systems of equations

Euclidean norm (vector length) in space R^n

$$\|\mathbf{v}\| = \sqrt{v_1^2 + v_2^2 + \cdots + v_n^2}$$

In [167…
```
v = np.array([0,5])
print(np.linalg.norm(v))
v = np.array([0,5,4])
print(np.linalg.norm(v))
```

```
5.0
6.4031242374328485
```

Calculating distances between points in space R^n

$$d(\mathbf{p}, \mathbf{q}) = d(\mathbf{q}, \mathbf{p}) = \sqrt{(q_1 - p_1)^2 + (q_2 - p_2)^2 + \cdots + (q_n - p_n)^2}$$

$$= \sqrt{\sum_{i=1}^{n} (q_i - p_i)^2}$$

In [168…
```
p1 = np.array([1,2])
p2 = np.array([3,4])
print(np.linalg.norm(p1 - p2))
```

```
2.8284271247461903
```

In [169…
```
p1 = np.array([1,2,5])
p2 = np.array([3,4,8])
print(np.linalg.norm(p1 - p2))
```

```
4.123105625617661
```

Matrix multiplication

$$A = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix}$$

A - matrix dimension $m\ x\ n$

$$B = \begin{bmatrix} b_{11} & b_{12} & \cdots & b_{1p} \\ b_{21} & b_{22} & \cdots & b_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ b_{n1} & b_{n2} & \cdots & b_{np} \end{bmatrix}$$

B - matrix dimension $n\ x\ p$

$$C = AB$$

C - matrix dimension $m \; x \; p$

To multiply two matrices, the number of columns in the left matrix must agree with the number of rows in the right matrix.

Matrix multiplication is not commutative!

$$AB \neq BA$$

In [170…
```
mA = np.array([[1,2],[4,5]])
print(mA)
mB = np.array([[3],[1]])
print(mB)
```

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

In [171…
```
print(mA @ mB)
print(mA.dot(mB))
```

```
[[ 5]
 [17]]
[[ 5]
 [17]]
```

Determinant of a matrix

The determinant of an n-dimensional matrix is calculated as follows:

$n = 1$: $\det\left[\,a\,\right] = a$

$n = 2$: $\det\begin{bmatrix} a & b \\ c & d \end{bmatrix} = ad - bc$

$n > 2$: $\det A = \sum_{i=1}^{n} (-1)^{i+n} a_{in} |A_{in}|$, $\left(|A| = \sum_{i=1}^{n} (-1)^{i+n} a_{in} |A_{in}|\right)$

In [172…
```
A = np.array([[1, 5], [1, 3]])
round(np.linalg.det(A))
```

Out[172… -2

Matrix trace

The trace of matrix A is called the sum of diagonal elements.

$$\mathrm{tr}(A) = \sum_{i=1}^{n} a_{ii} = a_{11} + a_{22} + \cdots + a_{nn}$$

In [68]:
```
A = np.array([[1, 5], [1, 3]])
round(np.trace(A))
```

Out[68]: 4

Inverse matrix

A square matrix A has an inverse matrix when there exists a matrix B such that:

$$AB = BA = I$$

In [173…
```
A = [[1,4],[5,3]]
B = np.linalg.inv(A)
print(B)
```

```
[[-0.1765  0.2353]
 [ 0.2941 -0.0588]]
```

In [174…
```
np.set_printoptions(precision=4, suppress=True)
print(np.dot(A,B))
```

```
[[ 1.  0.]
 [-0.  1.]]
```

In [175…
```
print(np.dot(B,A))
```

```
[[ 1.  0.]
 [-0.  1.]]
```

Solving systems of equations

A system of equations written using the matrix method:

$$\begin{bmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \dots & a_{mn} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_m \end{bmatrix}$$

And in short:

$$\mathbf{AX} = \mathbf{B}$$

Solving the equation:

$$\mathbf{X} = \mathbf{A}^{-1}\mathbf{B}$$

Example of an equation:

$$\begin{cases} 3x + 5y = 10 \\ 6x - 2y = 3 \end{cases}$$

In [177…
```
A = np.array([[3, 5], [6, -2]])
A_inverse = np.linalg.inv(A)
B = np.array([[10],[3]])
X = A_inverse @ B
print(X)
```

```
[[0.9722]
 [1.4167]]
```

In [88]:
```
print(3*X[0] + 5*X[1])
print(6*X[0] - 2*X[1])
```

```
[10.]
[3.]
```