# Numpy

## Numpy

- NumPy is a Python library for handling multi-dimensional arrays.
- It contains both the data structures needed for the storing and accessing arrays, and operations and functions for computation using these arrays.
- Must have **the same data type** for all its elements.
- Use for
  - store matrices, solve systems of linear equations, find eigenvalues/vectors, find matrix decompositions, and solve other problems familiar from linear algebra
  - a gray-scale image ( a two dimensional array)
  - a color image ( a three dimensional array)
  - ...

In [1]:

```
import numpy as np
```

## Creation of arrays

### Give a (nested) list as a parameter

In [2]:

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

Out[2]:

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

In [3]:

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

Out[3]:

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

In [4]:

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

Out[4]:

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

       [[5, 6],
        [7, 8]]])
```

## Create common types of arrays

In [5]:

```
np.zeros((3,4))  # matrix with all elements being zero
```

Out[5]:

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

In [6]:

```
np.zeros((3,4), dtype=int) # specify that elements are ints instead of floats
```

Out[6]:

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

In [7]:

```
np.ones((2,3))  # initializes all elements to one
```

Out[7]:

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

In [8]:

```
np.full((2,3), fill_value=7) #  initializes all elements to a specified value (here 7)
```

Out[8]:

```
array([[7, 7, 7],
       [7, 7, 7]])
```

In [9]:

```
np.empty((2,4)) # leaves the elements uninitialized
```

Out[9]:

```
array([[ 8.10267659e-322,  0.00000000e+000,  0.00000000e+000,
         0.00000000e+000],
       [ 0.00000000e+000,  0.00000000e+000, -1.08044151e-315,
         8.39911598e-322]])
```

In [10]:

```
np.eye(5, dtype=int) # creates the identity matrix
```

Out[10]:

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

In [11]:

```
np.arange(0,10,2)  # like the range function, but produces an array instead of a list.
```

Out[11]:

```
array([0, 2, 4, 6, 8])
```

In [12]:

```
np.linspace(0, np.pi, 5)  # Evenly spaced range with 5 elements
```

Out[12]:

```
array([0.        , 0.78539816, 1.57079633, 2.35619449, 3.14159265])
```

## Arrays with random elements

In [13]:

```
np.random.uniform(0, 1, size=(3,4))   # Elements are uniformly distributed from half-open interv
al [0.0,1.0)
```

Out[13]:

```
array([[0.43876527, 0.74939316, 0.44161347, 0.96887671],
       [0.3842937 , 0.42998051, 0.62717956, 0.81493231],
       [0.63497256, 0.60568394, 0.6932477 , 0.49416679]])
```

In [14]:

```
np.random.normal(0, 1, (3,4))  # Elements are normally distributed with mean 0 and standard devi
ation 1
```

Out[14]:

```
array([[-0.25658794,  1.48754072, -0.02285597,  1.10793473],
       [ 0.8581241 ,  0.83039041,  0.19642398,  1.21215304],
       [ 0.49903826, -0.06692422,  0.39983218,  1.06187442]])
```

In [15]:

```
np.random.randint(-2, 10, (3,4))  # Elements are uniformly distributed integers from the half-open interval [-2,10)
```

Out[15]:

```
array([[2, 5, 6, 9],
       [4, 3, 3, 4],
       [3, 2, 8, 0]])
```

In [16]:

```
## Global
np.random.seed(0)
print(np.random.randint(0, 100, 10))
print(np.random.normal(0, 1, 10))
```

```
[44 47 64 67 67  9 83 21 36 87]
[ 1.26611853 -0.50587654  2.54520078  1.08081191  0.48431215  0.57914048
 -0.18158257  1.41020463 -0.37447169  0.27519832]
```

In [17]:

```
## Local
new_generator = np.random.RandomState(seed=123)  # RandomState is a class, so we give the seed to its constructor
new_generator.randint(0, 100, 10)
```

Out[17]:

```
array([66, 92, 98, 17, 83, 57, 86, 97, 96, 47])
```

Functions in `numpy.random`

- seed
- permutation
- rand
- randint
- randn
- binomial
- normal
- beta
- chisquare
- gamma
- uniform

# Array type and attributes

- `ndim` tells the number of dimensions,
- `shape` tells the size in each dimension,
- `size` tells the number of elements,
- `dtype` tells the element type.

In [18]:

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

Out[18]:

2

In [19]:

```
b.shape
```

Out[19]:

(2, 3)

In [20]:

```
b.size
```

Out[20]:

6

In [21]:

```
b.dtype
```

Out[21]:

dtype('int32')

# Indexing and Slicing

In [22]:

```
b=np.array([[1,2,3], [4,5,6]])
print(b)
print(b[1,2])    # row index 1, column index 2
print(b[0,-1])   # row index 0, column index -1
```

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

In [23]:

```
print(b[0])    # First row
print(b[1])    # Second row
```

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

In [24]:

```
# As with lists, modification through indexing is possible
b[0,0] = 10
print(b)
```

```
[[10  2  3]
 [ 4  5  6]]
```

In [25]:

```
a=np.array([1,4,2,7,9,5])
print(a)
print(a[1:3])
print(a[::-1])    # Reverses the array
```

```
[1 4 2 7 9 5]
[4 2]
[5 9 7 2 4 1]
```

In [26]:

```
print(b)
print(b[:,0])  # First column
print(b[0,:])  # First row
print(b[:,1:]) # Except the first column
print(b[:, [1,3]])
```

```
[[10  2  3]
 [ 4  5  6]]
[10  4]
[10  2  3]
[[2 3]
 [5 6]]
```

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
<ipython-input-26-a0d229978f35> in <module>()
      3 print(b[0,:])  # First row
      4 print(b[:,1:]) # Except the first column
----> 5 print(b[:, [1,3]])

IndexError: index 3 is out of bounds for axis 1 with size 3
```

# Reshaping

In [27]:

```
d=np.arange(4)             # 1d array
dr=d.reshape(1,4)          # row vector
dc=d.reshape(4,1)          # column vector
```

In [28]:

```
print(d)
```

```
[0 1 2 3]
```

In [29]:

```
print(dr)
print(d[np.newaxis, :]) # alternative way
```

```
[[0 1 2 3]]
[[0 1 2 3]]
```

In [30]:

```
print(dc)
print(d[:, np.newaxis]) # alternative way
```

```
[[0]
 [1]
 [2]
 [3]]
[[0]
 [1]
 [2]
 [3]]
```

In [31]:

```
a=np.random.randint(0, 10, (4,4))
print(a)
```

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

In [32]:

```
print(a.T)
```

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

# Array contatenation, splitting and stacking

The are two ways of combining several arrays into one bigger array:

- `Concatenate` takes n-dimensional arrays and returns an n-dimensional array
- `Stack` takes n-dimensional arrays and returns n+1-dimensional array.

Inverse operation of `concatenate` is `split`.

In [33]:

```
a=np.arange(2)
b=np.arange(2,5)
print(f"a has shape {a.shape}: {a}")
print(f"b has shape {b.shape}: {b}")
np.concatenate((a,b))  # concatenating 1d arrays
```

```
a has shape (2,): [0 1]
b has shape (3,): [2 3 4]
```

Out[33]:

```
array([0, 1, 2, 3, 4])
```

In [34]:

```
c=np.arange(1,5).reshape(2,2)
print(f"c has shape {c.shape}:", c, sep="\n")
np.concatenate((c,c))   # concatenating 2d arrays
```

```
c has shape (2, 2):
[[1 2]
 [3 4]]
```

Out[34]:

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

In [35]:

```
np.concatenate((c,c), axis=1)
```

Out[35]:

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

In [36]:

```
np.stack((b,b))
```

Out[36]:

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

In [37]:

```
np.stack((b,b), axis=1)
```

Out[37]:

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

If you want to catenate arrays with different dimensions, for example to add a new column to a 2d array, you must first reshape the arrays to have same number of dimensions:

In [38]:

```
print(a)

print("New row:")
print(np.concatenate((c,a.reshape(1,2))))
```

```
[0 1]
New row:
[[1 2]
 [3 4]
 [0 1]]
```

In [39]:

```
## split
d=np.arange(12).reshape(6,2)
print("d:")
print(d)
d1,d2 = np.split(d, 2)  # specify the number of equal parts the array is divided into,
print("d1:")
print(d1)
print("d2:")
print(d2)
```

```
d:
[[ 0  1]
 [ 2  3]
 [ 4  5]
 [ 6  7]
 [ 8  9]
 [10 11]]
d1:
[[0 1]
 [2 3]
 [4 5]]
d2:
[[ 6  7]
 [ 8  9]
 [10 11]]
```

In [40]:

```
d=np.arange(12).reshape(2,6)
print("d:")
print(d)
parts=np.split(d, (2,3,5), axis=1) # specify explicitly the break points
for i, p in enumerate(parts):
    print("part %i:" % i)
    print(p)
```

```
d:
[[ 0  1  2  3  4  5]
 [ 6  7  8  9 10 11]]
part 0:
[[0 1]
 [6 7]]
part 1:
[[2]
 [8]]
part 2:
[[ 3  4]
 [ 9 10]]
part 3:
[[ 5]
 [11]]
```

# Using Numpy for data analysis

- max , min , sum , mean , std ,...

In [41]:

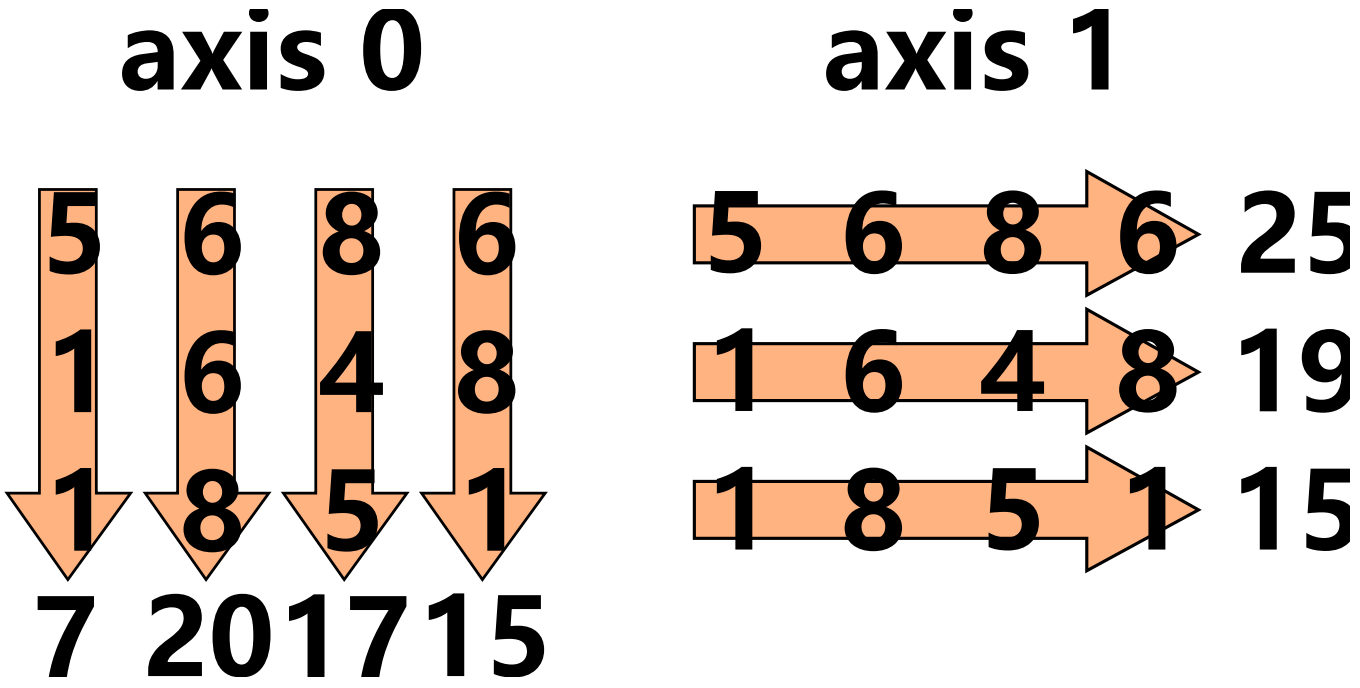
```
np.random.seed(0)
a=np.random.randint(-100, 100, (4,5))
print(a)
print(f"Minimum: {a.min()}, maximum: {a.max()}")
print(f"Sum: {a.sum()}")
print(f"Mean: {a.mean()}, standard deviation: {a.std()}")
```

```
[[ 72 -53  17  92 -33]
 [ 95   3 -91 -79 -64]
 [-13 -30 -12  40 -42]
 [ 93 -61 -13  74 -12]]
Minimum: -91, maximum: 95
Sum: -17
Mean: -0.85, standard deviation: 58.39886557117355
```

In [42]:

```
np.random.seed(9)
b=np.random.randint(0, 10, (3,4))
print(b)
print("Column sums:", b.sum(axis=0))
print("Row sums:", b.sum(axis=1))
```

```
[[5 6 8 6]
 [1 6 4 8]
 [1 8 5 1]]
Column sums: [ 7 20 17 15]
Row sums: [25 19 15]
```




## Comparisons and masking

In [43]:

```
a=np.array([1,3,4])
b=np.array([2,2,7])
c = a < b
print(c)
```

```
[ True False  True]
```

In [44]:

```
print(c.all())   # were all True
print(c.any())   # was some comparison True
```

```
False
True
```

In [45]:

```
print(np.sum(c))
```

2

## Example: the daily average temperatures

In [46]:

```
import pandas as pd
a=pd.read_csv("kumpula-weather-2017.csv")['Air temperature (degC)'].values
```

In [47]:

```
a[:6]
```

Out[47]:

```
array([  0.6,  -3.9,  -6.5, -12.8, -17.8, -17.8])
```

In [48]:

```
print("Number of days with the temperature below zero", np.sum(a < 0))
```

Number of days with the temperature below zero 49

In [49]:

```
np.sum((0 < a) & (a < 10))
```

Out[49]:

185

In [50]:

```
c = a > 0
print(c[:10])    # print only the first ten elements
print(a[c])      # Select only the positive temperatures
```

```
[ True False False False False False False False  True  True]
[ 0.6  0.5  1.7  1.1  0.8  1.1  1.6  1.   0.1  1.9  1.6  0.8  0.6  1.
  0.2  0.4  1.5  4.4  0.5  1.5  1.9  2.2  0.4  2.1  1.5  1.5  0.9  0.2
  1.2  1.4  2.2  0.6  0.6  3.8  3.4  2.   1.6  0.6  0.2  1.6  2.3  2.5
  0.5  1.9  4.8  6.6  2.4  0.4  0.2  1.5  4.3  3.   3.8  3.3  4.7  4.1
  3.3  3.7  6.   4.3  1.8  1.2  0.6  0.3  1.2  2.5  5.7  3.1  2.9  3.6
  2.8  3.2  4.4  4.1  2.3  2.2  7.6  9.4  9.2  6.7 10.1 11.1  6.1  4.4
  1.5  1.5  2.9  5.1  7.5  9.8  9.   7.3  9.4 11.2 16.1 16.9 14.7 14.6
 13.3 12.4 14.1 14.1  9.6 13.2 14.1  9.6 10.8  6.5  6.2 10.   9.9 10.4
 14.5 16.4 12.3 14.  16.3 16.8 12.6 12.6 14.8 15.1 18.4 19.3 16.9 17.8
 12.7 12.4 12.2 10.8 12.9 14.  13.5 13.7 15.3 16.3 17.2 14.1 16.4 14.5
 14.1 14.  11.5 14.7 16.1 17.4 18.  16.8 16.8 14.7 15.3 16.7 18.  15.4
 15.  13.7 14.6 15.8 15.6 16.9 15.7 16.1 17.2 19.  19.1 17.8 18.3 17.8
 18.9 17.5 18.2 16.6 17.1 16.  15.2 17.2 18.3 18.7 18.1 19.6 17.4 16.1
 16.4 17.6 19.  18.  18.2 17.4 16.2 14.9 13.5 12.9 11.2 10.4 10.7 12.
 14.4 16.8 17.2 15.1 12.9 12.9 11.9 11.4 10.4 10.3 10.6 13.1 14.6 14.6
 14.9 14.4 12.9 12.2 12.1  9.9  8.7  9.   9.2 12.4 11.5 12.8 12.5 12.6
 12.  12.1 10.1  8.9  8.8  9.1  9.2  8.3 11.2  8.8  7.7  8.1  9.3  8.6
  8.1  6.9  6.   7.5  7.2  8.3 10.7  8.5  8.3  4.6  2.   0.2  0.1  1.3
  0.8  2.1  0.3  3.3  2.1  1.2  0.1  0.5  3.2  8.   8.4  7.5  3.9  5.9
  6.7  7.2  5.9  3.1  2.4  1.8  3.1  3.5  5.8  5.9  4.   1.2  0.9  1.
  1.5  6.1  4.2  2.3  4.6  3.9  2.8  3.1  0.9  1.4  5.   1.3  5.2  4.2
  2.   1.4  1.6  1.6  1.6  1.7  2.4  0.1  2.   1.   2.6  2.5  1.2  0.3
  1.9  3.8  2.8  3.8  2.5  1.6]
```

In [51]:

```
a[~c] = 0
print(a[:6])
```

```
[0.6 0.  0.  0.  0.  0. ]
```

# Sorting arrays

In [52]:

```
a=np.array([2,1,4,3,5])
print(np.sort(a))          # Does not modify the argument
print(a)
```

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

In [53]:

```
a.sort()             # Modifies the argument
print(a)
```

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

In [54]:

```
a=np.array([23,12,47,35,59])
print("Array a:", a)
idx = np.argsort(a)
print("Indices:", idx)
```

```
Array a: [23 12 47 35 59]
Indices: [1 0 3 2 4]
```

# Matrix operations

In [55]:

```
np.random.seed(0)
a=np.random.randint(0,10, (3,2))
b=np.random.randint(0,10, (2,4))
print(a)
print(b)
```

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

In [56]:

```
print(np.matmul(a,b))
```

```
[[ 15  25  10  20]
 [ 30  33  30  36]
 [ 84  89  86 100]]
```

In [57]:

```
print(a@b)
```

```
[[ 15  25  10  20]
 [ 30  33  30  36]
 [ 84  89  86 100]]
```

# Invertible matrix

In [58]:

```
## Invertible matrix
s=np.array([[1, 6, 7],
 [7, 8, 1],
 [5, 9, 8]])
print("Original matrix:", s, sep="\n")
s_inv = np.linalg.inv(s)
print("Inverted matrix:", s_inv, sep="\n")
print("Matrix product:", s @ s_inv, sep="\n")                    # This should be pretty
close to the identity matrix np.eye(3)
print("Matrix product the other way:", s_inv @ s, sep="\n")
```

```
Original matrix:
[[1 6 7]
 [7 8 1]
 [5 9 8]]
Inverted matrix:
[[-0.61111111 -0.16666667  0.55555556]
 [ 0.56666667  0.3        -0.53333333]
 [-0.25555556 -0.23333333  0.37777778]]
Matrix product:
[[ 1.00000000e+00 -8.32667268e-17  4.44089210e-16]
 [-5.55111512e-17  1.00000000e+00  4.44089210e-16]
 [-4.44089210e-16  2.22044605e-16  1.00000000e+00]]
Matrix product the other way:
[[ 1.00000000e+00  9.99200722e-16 -8.88178420e-16]
 [ 0.00000000e+00  1.00000000e+00  8.88178420e-16]
 [ 0.00000000e+00 -4.44089210e-16  1.00000000e+00]]
```

# Solving system of linear equations

- np.linalg.solve

In [59]:

```
a = np.array([[3,1], [1,2]])
b = np.array([9,8])
print(np.linalg.solve(a, b))
```

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