L10: Numerical and Scientific Packages

Numpy

Modules

Module → Python file where objects (functions, classes, constants, etc.) can be accessed from another file.

It is a way of organizing long codes.

<u>Example</u>: Consider a file *arithmetic.py* containing some functions.

```
""" arithmetic.py """
def add(a, b):
    return a + b

def sub(a, b):
    return a - b

def mult(a, b):
    return a * b

def div(a, b):
    return a / b
```

We can access it from another Python file by **importing the module**:

```
""" MyProgram.py """
import arithmetic

print(add(7, 5))
```

Modules

An alternate syntax for importing objects from a module is as follows.

```
""" MyProgram.py"""
from arithmetic import add

print(add(7, 5))
```

We can import multiple objects by separating them by commas.

```
""" MyProgram.py """
from arithmetic import add,sub,mult,div

print(add(7, 5),sub(7, 5),mult(7, 5),div(7, 5))
```

To import all objects into a module:

```
""" MyProgram.py """
from arithmetic import *

print(add(7, 5), sub(7, 5),mult(7, 5),div(7, 5))
```

A module in the Lib folder (e.g., C: \Python\Lib) will be visible to any file.

Packages

Package → Group of related modules stored in a folder.

e.g., A mathematics package with the following structure:

```
mathematics/
    |-- __init__.py
    |-- arithmetic.py
    |-- geometry.py
```

It must contain a __init__.py

Python identifies the folder as a package

We can use the package in different ways:

```
import mathematics.arithmetic

print(mathematics.arithmetic.add(7, 5))
```

```
from mathematics import arithmetic

print(arithmetic.add(7, 5))
```

```
from mathematics.arithmetic import add

print(add(7, 5))
```

Introduction to `Numpy`

`Numpy` is a package for **scientific computing** with Python.

It contains:

- Data Type
- Classes
- Functions
- Modules

Interesting properties:

- Perfect **integration with Python** core
  - It can interact with data types from Python
- Wide **set of functionalities** for scientific computation
  - Similar to commercial packages such as Mathematica or Matlab
- **Fast execution** (most of the functions are implemented in C)
- **Multiplatform** package
- **Free**

Tensors in **Numpy**

Tensor → A container for data, (usually numerical data).
Can be seen as a generalization of **matrices** to n dimensions.
In the context of tensors, dimensions are usually called axes.

Numpy includes a data structure for tensors (multidimensional arrays).

What is an array?

- A type of data composed of simple data types.
- Items are sorted according to a defined sequence.

values →	19	20	19	18	20	18	18	19	19	20
indices →	0	1	2	3	4	5	6	7	8	9

Like in a list?

✓ Yes, as a sorted sequence.

✗ Not in terms of content: All the data in the array must be of the same type.

Creating a Tensor

- We can create a tensor from a Python list or tuple
- Using **`NUMPY`** functions
- Reading the data from a file
- Making a copy from another Tensor

Tensors in **Numpy**: creation using lists

```python
import numpy as np
```

We import NUMPY and assign a short name to avoid writing too much

```python
a = np.array([[1,2,3],[4,5,6]], dtype='int64')
```

Data type

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

We can create a tensor from a Python list

a is an instance of the `ndarray` class that has some key attributes:

a.ndim

Number of axes in the tensor.

```
In [12]: a.ndim
Out[12]: 2
```

a.shape

Tuple with the dimension of each axis of the tensor.

```
In [13]: a.shape
Out[13]: (2, 3)
```

a.size

Number of elements in the tensor.

```
In [14]: a.size
Out[14]: 6
```

a.dtype

Type of the data contained in the tensor.

```
In [15]: a.dtype
Out[15]: dtype('int64')
```

a.itemsize

Length in bytes of the tensor elements.

```
In [16]: a.itemsize
Out[16]: 8
```

Data Types in **`Numpy`**

`Numpy` provides a higher range of numeric data types than Python:

```
bool_       Boolean (True or False), stored as 1 byte
int_        Integer (int32 or int64, depending on the platform)
int8        Byte (-128 to 127)
int16       Integer (-32768 to 32767)
int32       Integer (-2.147.483.648 to 2.147.483.647)
int64       Integer (-9.223.372.036.854.775.808 to 9.223.372.036.854.775.807)
uint8       Unsigned integer (0 to 255)
uint16      Unsigned integer (0 to 65535)
uint32      Unsigned integer (0 to 4.294.967.295)
uint64      Unsigned integer (0 to 18.446.744.073.709.551.615)
float_      Shortcut for float64
float32     Simple precision float
float64     Double precision float
complex_    Shortcut to complex128
complex64   Complex number, real and imaginary parts with 32 bits each
complex128  Complex number, real and imaginary parts with 64 bits each
```

Tensors in **Numpy**

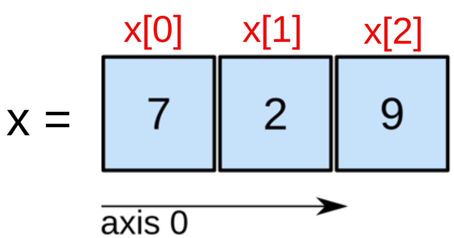
Scalars (0D tensors) $\rightarrow$ Tensor containing only one number.
A scalar tensor has 0 axes (ndim == 0).

```
In [2]: x = np.array(12)

In [3]: x
Out[3]: array(12)

In [4]: x.ndim
Out[4]: 0
```

The `ndim` attribute returns the **number of axes** of the tensor

Vectors (1D tensors) $\rightarrow$ Tensor containing an array of numbers.
A 1D tensor has 1 axis.

```
In [5]: x = np.array([7,2,9])

In [6]: x
Out[6]: array([7, 2, 9])

In [7]: x.ndim
Out[7]: 1
```

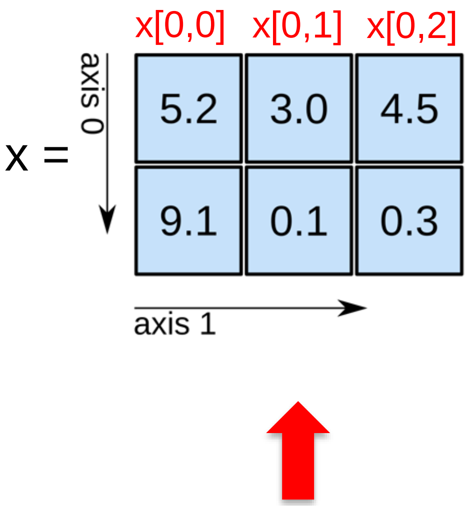


Note:
dimensions of an axis = number of elements in the axis.
x is a 1D tensor with a 3-dimensional vector

Tensors in **Numpy**

Matrices (2D tensors) $\rightarrow$ Tensor containing an array of vectors.
A 2D tensor has 2 axes (rows and columns).

```
In [9]: x
Out[9]:
array([[5.2, 3. , 4.5],
       [9.1, 0.1, 0.3]])

In [10]: x.ndim
Out[10]: 2
```



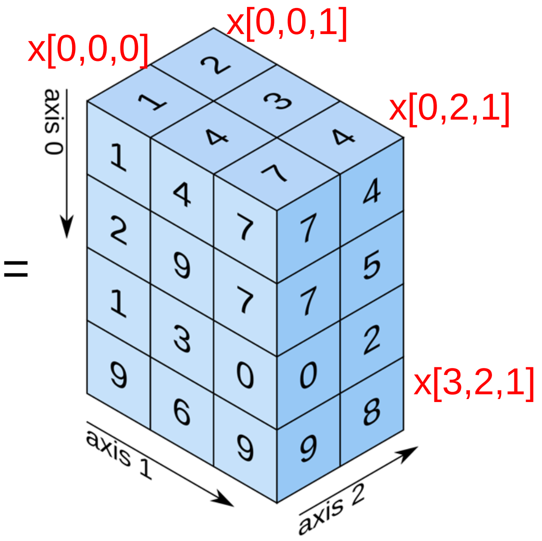
axis 0 $\rightarrow$ rows
axis 1 $\rightarrow$ columns

Tensors in **Numpy**

3D tensors → Tensor containing an array of matrices.
A 3D tensor has 3 axes and can be visualized as a cube.

```
In [67]: x = np.array([[[1,2],[4,3],[7,4]],[[2,7],[9,6],[7,5]],
[[1,2],[3,3],[0,2]],[[9,2],[6,3],[9,8]]])

In [68]: x
Out[68]:
array([[[1, 2],
        [4, 3],
        [7, 4]],

       [[2, 7],
        [9, 6],
        [7, 5]],

       [[1, 2],
        [3, 3],
        [0, 2]],

       [[9, 2],
        [6, 3],
        [9, 8]]])

In [69]: x.ndim
Out[69]: 3
```

x =



Higher-dimensional tensors

By packing 3D tensors in an array → We can create a 4D tensor, and so on.

Creating tensors with **Numpy** functions

`np.arange([start], stop[, step], dtype=None)`

Returns evenly spaced values in a given range

The "step" can be a real number (e.g decimal), contrary to Python (`range`)

```
In [6]: np.arange(5, 6, 0.1)
Out[6]: array([5. , 5.1, 5.2, 5.3, 5.4, 5.5, 5.6, 5.7, 5.8, 5.9])
```

`np.zeros(shape, dtype=float)`

Returns an array of zeros of the given dimensions (shape)

```
In [7]: np.zeros((2,3))
Out[7]:
array([[0., 0., 0.],
       [0., 0., 0.]])
```

`np.ones(shape, dtype=float)`

Returns an array of ones of the given dimensions (shape)

```
In [3]: np.ones((2,3,))
Out[3]:
array([[1., 1., 1.],
       [1., 1., 1.]])
```

Creating tensors from a file

- The reading and the paremeters depend on the format of the file
- It is common to read CSV files:
  - Data is separated using ;
  - The first line usually contains the name of each column
  - It is easy to export this file format from spreadsheet editors (e.g. Ms.Excel)

```
line 1 -> objID;Weight;Temp;Pressure
line 2 -> 12376;138.692294;0.002;46.253899
...
```

```
np.loadtxt(fname, dtype=numpy.float, comments="#",
           delimiter=None, converters=None, skiprows=0,
           usecols=None, unpack=False, ndmin=0)
```

```
a = np.loadtxt(‘data.csv’,delimiter=’;’, skiprows=1)
```

Creating tensors by copying another Tensor

Arrays in NUMPY are mutable

```
In [8]: a = np.array([1,2,3])

In [9]: a
Out[9]: array([1, 2, 3])

In [10]: b=a

In [11]: b[0]=4

In [12]: a
Out[12]: array([4, 2, 3])
```

`np.copy(array)`

Returns a copy of the array passed as a parameter

```
In [13]: a = np.array([1,2,3])

In [14]: b = np.copy(a)

In [15]: b[0]=4

In [16]: a
Out[16]: array([1, 2, 3])

In [17]: b
Out[17]: array([4, 2, 3])
```

Creating tensors with **`Numpy`** functions

`np.random.rand(d0,d1,...,dn)`

Returns an array of random numbers in [0,1] of the given shape

```
In [22]: a=np.random.rand(2,4)

In [23]: a
Out[23]:
array([[0.21230443, 0.48718693, 0.5484724 , 0.05722388],
       [0.22947542, 0.36771084, 0.20369547, 0.57490007]])
```

Indexing

The operator [ ] allows accessing the elements of the array.

We use as many indexes as dimensions separated by commas:

```
In [30]: a=np.array([[1,2,3],[4,5,6]])

In [31]: print(a[1,0])
4
```

The first index on each dimension is 0.

If the index is out of the limits of the array, Python generates an exception of type `IndexError`.

```
In [32]: a=np.array([[1,2,3],[4,5,6]])

In [33]: print(a[4,1])
Traceback (most recent call last):

  File "<ipython-input-33-ff605849a7fa>", line 1, in <module>
    print(a[4,1])

IndexError: index 4 is out of bounds for axis 0 with size 2
```

Indexing and slicing 1D

We can obtain a cut (a slice) of the array using the operator [ ] and indicating the start, the end, and the step of the cut for each dimension:

```
[start:end:step]
```

```
In [34]: one_d=np.arange(10)
```

End not included

```
In [35]: one_d
Out[35]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])

In [36]: one_d[2:6]
Out[36]: array([2, 3, 4, 5])

In [37]: one_d[:6]
Out[37]: array([0, 1, 2, 3, 4, 5])
```

Without start, 0 is assumed

```
In [38]: one_d[2:]
Out[38]: array([2, 3, 4, 5, 6, 7, 8, 9])
```

Without end, the end of the array is assumed

```
In [39]: one_d[2:8:2]
Out[39]: array([2, 4, 6])
```

Indexing and slicing 2D

```
In [57]: two_d
Out[57]:
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11],
       [12, 13, 14, 15],
       [16, 17, 18, 19]])

In [58]: two_d[0,0]
Out[58]: 0

In [59]: two_d[0]
Out[59]: array([0, 1, 2, 3])

In [60]: two_d[1:3]
Out[60]:
array([[ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])

In [61]: two_d[::2]
Out[61]:
array([[ 0,  1,  2,  3],
       [ 8,  9, 10, 11],
       [16, 17, 18, 19]])
```

In 2D arrays we can cut by rows, by columns, or both

We only take one dimension (row 0)

With operator [start:end:step] we will choose only rows

Indexing and slicing 2D

To cut by rows and columns we use the operator

`[start:end:step, start:end:step]`

row column

```
In [62]: two_d
Out[62]:
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11],
       [12, 13, 14, 15],
       [16, 17, 18, 19]])
```

```
In [63]: two_d[1:3,1]
    ...:
Out[63]: array([5, 9])
```

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

```
In [64]: two_d[1:3,1:3]
Out[64]:
array([[ 5,  6],
       [ 9, 10]])
```

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

```
In [65]: two_d[0:3:2,0:4:2]
Out[65]:
array([[ 0,  2],
       [ 8, 10]])
```

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

Indexing and slicing 2D

Slicing always **returns a view of the array**, referring to the same input data:

```
In [66]: two_d
Out[66]:
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11],
       [12, 13, 14, 15],
       [16, 17, 18, 19]])

In [67]: two_d[0:2,0:2] += 10

In [68]: two_d
Out[68]:
array([[10, 11,  2,  3],
       [14, 15,  6,  7],
       [ 8,  9, 10, 11],
       [12, 13, 14, 15],
       [16, 17, 18, 19]])
```

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

Advanced Indexing

NUMPY arrays can be accessed by using another array as index:

```
In [90]: a = np.arange(10)*3

In [91]: a
Out[91]: array([ 0,  3,  6,  9, 12, 15, 18, 21, 24, 27])

In [92]: i= np.array([1,4,6,6,8])

In [93]: a[i]
Out[93]: array([ 3, 12, 18, 18, 24])
```

In 2D arrays we can select rows with an array as index:

```
In [104]: two_d
Out[104]:
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11],
       [12, 13, 14, 15],
       [16, 17, 18, 19]])
```

```
In [105]: i = np.array([0,2,4])

In [106]: two_d[i]
Out[106]:
array([[ 0,  1,  2,  3],
       [ 8,  9, 10, 11],
       [16, 17, 18, 19]])
```

Advanced Indexing

We can use an array of Booleans to access array values:

```
In [107]: a = np.arange(10) * 2

In [108]: a
Out[108]: array([ 0,  2,  4,  6,  8, 10, 12, 14, 16, 18])

In [109]: b = a > 10

In [110]: b
Out[110]:
array([False, False, False, False, False, False,  True,  True,  True,
        True])

In [111]: a[b]
Out[111]: array([12, 14, 16, 18])

In [112]: a[a>10]
Out[112]: array([12, 14, 16, 18])
```

Only the values that correspond to a `True` value in the array index are taken

Operations with arrays

- NUMPY has implemented more than a hundred basic functions in order to operate with arrays, such as:
  - mathematical operations: add, substract, multiply, log...
  - trigonometric operations: cos, sin tanh, arctan...
  - boolean operations: bitwise_and, bitwise_or, left_shift...
  - comparisons: greater, lesser, logical_and ...

Element-wise operations

Element-wise operations are applied independently to each element in the tensor:

- For arrays with identical dimensions: The operations are performed between each pair of elements that occupy the same position

Numpy has many well-optimized element-wise operations for arrays.

```
In [20]: a = np.array([3,6,8,2,4,9,1,0])

In [21]: b = np.array([1,2,3,4,4,3,2,1])

In [22]: c=a+b

In [23]: c
Out[23]: array([ 4,  8, 11,  6,  8, 12,  3,  1])

In [24]: d=a*b

In [25]: d
Out[25]: array([ 3, 12, 24,  8, 16, 27,  2,  0])

In [26]: max_ab = np.maximum(a,b)

In [27]: max_ab
Out[27]: array([3, 6, 8, 4, 4, 9, 2, 1])
```

```
In [167]: a=np.array([[1,2],[3,4]])

In [168]: b=np.ones((2,2),dtype="int")

In [169]: a
Out[169]:
array([[1, 2],
       [3, 4]])

In [170]: b
Out[170]:
array([[1, 1],
       [1, 1]])

In [171]: print(a*b)
[[1 2]
 [3 4]]
```

Tensor reshaping

From 1D to 2D tensors: `reshape`

```
In [128]: a = np.arange(12)

In [129]: a
Out[129]: array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11])

In [130]: a = a.reshape(3,4)

In [131]: a
Out[131]:
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
```

From 2D to 3D

```
In [132]: a = a.reshape(2,2,3)

In [133]: a
Out[133]:
array([[[ 0,  1,  2],
        [ 3,  4,  5]],

       [[ 6,  7,  8],
        [ 9, 10, 11]]])
```

From ND to 1D: `flatten`

```
In [135]: a.flatten()
Out[135]: array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11])
```

Reshape an array

- `np.insert(array, **obj**, values, axis=None)`

```
In [140]: a = np.arange(3)

In [141]: a
Out[141]: array([0, 1, 2])

In [142]: np.insert(a, (0, -1), -1)
Out[142]: array([-1,  0,  1, -1,  2])
```

obj: defines the index or indices before which *values* are inserted

Insert -1 at the first and last position

- `np.append(array, values, axis=None)`

```
In [146]: a
Out[146]: array([0, 1, 2])

In [147]: np.append(a, -3)
Out[147]: array([ 0,  1,  2, -3])
```

Adds at the end

- `np.delete(array, obj, axis=None)`

```
In [148]: a
Out[148]: array([0, 1, 2])

In [149]: np.delete(a, 0)
Out[149]: array([1, 2])
```

obj: indicates the indices to remove along the specified axis

Reshape an array

`np.concatenate((a1, a2, ...), axis=0)`

- Joins a sequence of arrays "(a1, a2, ...)" along an existing axis.
- Its dimensions (shape) must match, except for the axis dimension.
- "Axis" is the dimension where arrays must be joined.

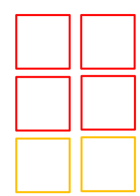
```
In [150]: a = np.array([[1, 2], [3, 4]])

In [151]: a
Out[151]:
array([[1, 2],
       [3, 4]])

In [152]: print(a.shape)
(2, 2)
```

Matrix 2x2 (2 rows, 2 cols)

```
In [153]: b = np.array([[5, 6]])

In [154]: print(b.shape)
(1, 2)
```

Matrix 1x2 (1 row, 2 cols)

```
In [155]: np.concatenate((a, b), axis=0)
Out[155]:
array([[1, 2],
       [3, 4],
       [5, 6]])
```

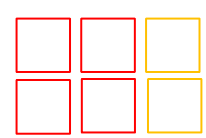


`concatenate`

```
In [156]: print(b.T.shape)
(2, 1)
```

`array.T` Transpose

```
In [157]: np.concatenate((a, b.T), axis=1)
Out[157]:
array([[1, 2, 5],
       [3, 4, 6]])
```



`concatenate`

Broadcasting

In the case of tensors that have different shapes Numpy has a mechanism to allow some element-wise operations $\rightarrow$ **broadcasting**

When possible, the smaller tensor will be broadcasted to match the shape of the larger tensor.

It is also possible when one of the final dimensions is 1 (axis x,y or z)

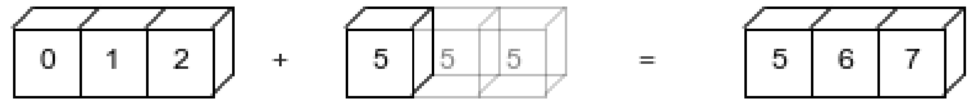
Two steps:

1. **Axes** are **added to the smaller tensor** to match the `ndim` of the larger tensor
2. The **smaller tensor is repeated** alongside these new axes to match the shape of the larger tensor.

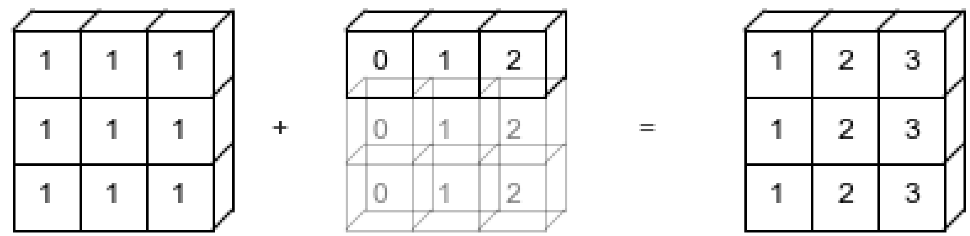
If it is not possible to apply broadcasting to do an operation between tensors of different shape, a `ValueError` exception is generated

Broadcasting: Arithmetic operations

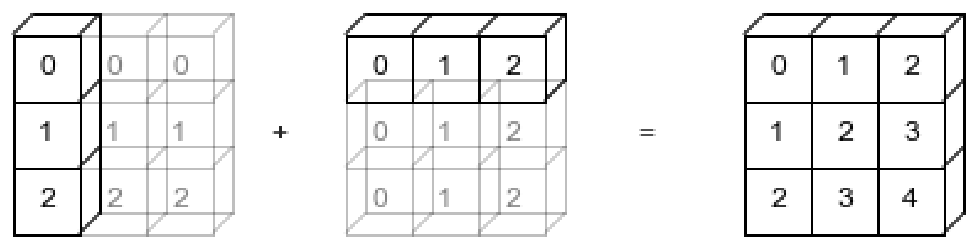
```
In [174]: a = np.arange(3)

In [175]: a
Out[175]: array([0, 1, 2])

In [176]: a+5
Out[176]: array([5, 6, 7])
```



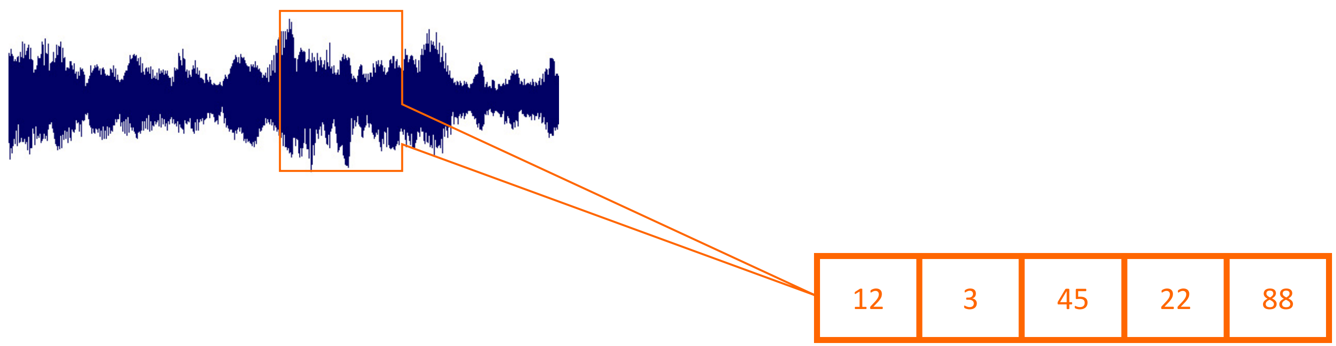

```
In [181]: a = np.ones((3,3),dtype="int")

In [182]: b = np.arange(3)

In [183]: print(a+b)
[[1 2 3]
 [1 2 3]
 [1 2 3]]
```



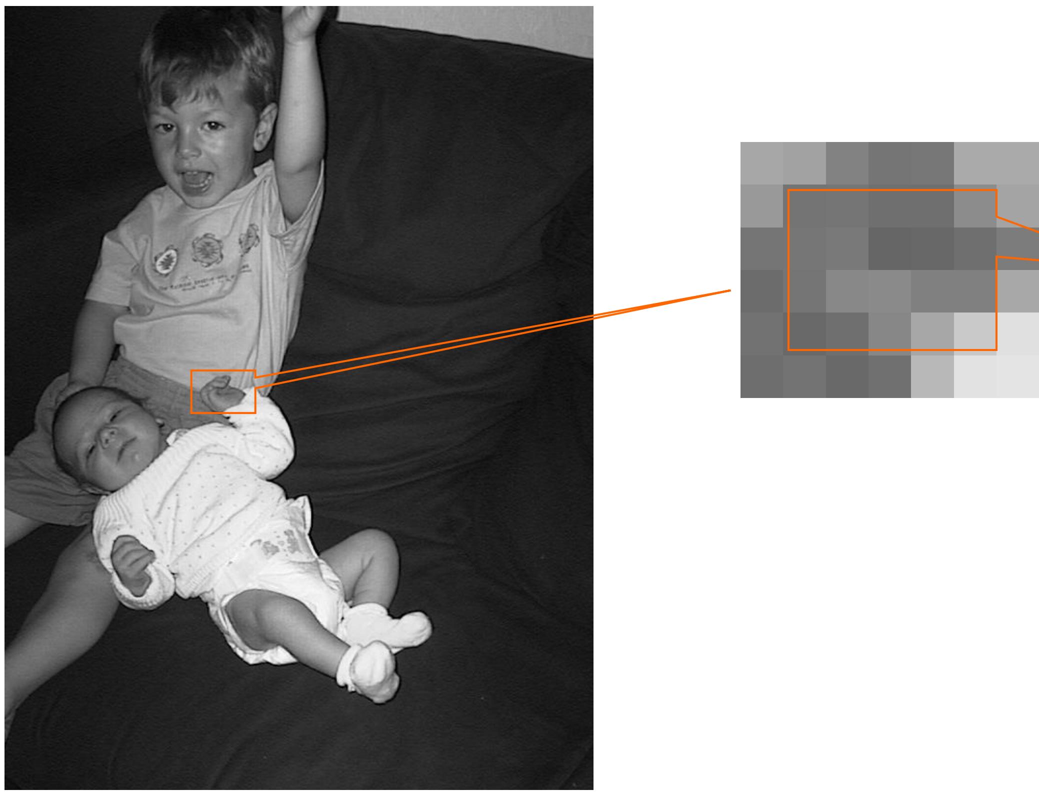

```
In [187]: a = np.arange(3).reshape((3,1))

In [188]: a
Out[188]:
array([[0],
       [1],
       [2]])

In [189]: b = np.arange(3)

In [190]: print (a+b)
[[0 1 2]
 [1 2 3]
 [2 3 4]]
```



How we represent an audio signal?



How we represent an image?



12	3	45	22	88
83	38	100	19	66
22	7	209	89	88
119	18	19	20	99

Data visualization

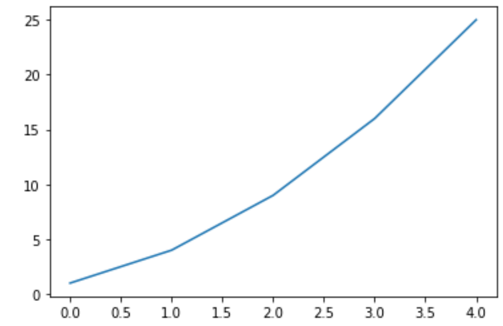
- Data visualization involves exploring data through visual representations. It is closely associated with **Data mining**
- Making good data representations is more than a beautiful image. When we have a simple and visually appealing representation of a data set, its meaning becomes clear to analysts.
- We could be able to see patterns and draw conclusions about data that we did not know that they existed.
- One of the most popular tools is matplotlib, a mathematical drawing library. http://matplotlib.org
- To use *matplotlib*:

```
import matplotlib.pyplot as plt
```

Data visualization

- A simple example

```
import matplotlib.pyplot as plt

quadrats = [1, 4, 9, 16, 25]
plt.plot(quadrats)
plt.show()
```



- To read an image from a file

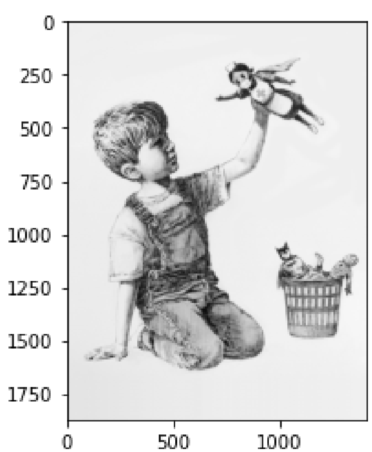
```
import matplotlib.image as mpimg

image = mpimg.imread("name_file")
plt.imshow(image)
```

- `Image` is a `NUMPY` array

Data visualization example

```python
import matplotlib.image as mpimg
import matplotlib.pyplot as plt

image = mpimg.imread("c:\\test.png")
print("type image:",type(image))
print("dimensions:",image.ndim)
print("shape:",image.shape)
plt.imshow(image)
```

```
type image: <class 'numpy.ndarray'>
dimensions: 3
shape: (1872, 1404, 4)
type image2: <class 'numpy.ndarray'>
dimensions2: 2
shape2: (500, 600)
```




```python
image2 = image[100:600,650:1250,0]
print("type image2:",type(image2))
print("dimensions2:",image2.ndim)
print("shape2:",image2.shape)
plt.imshow(image2)
```

```
type image: <class 'numpy.ndarray'>
dimensions: 3
shape: (1872, 1404, 4)
type image2: <class 'numpy.ndarray'>
dimensions2: 2
shape2: (500, 600)
```