Data Science and AI for industrial systems

Hands-on session 2
Data cleaning and Clustering

Prof. Daniele Apiletti, Simone Monaco

24/02/2023

Politecnico di Torino

Introduction to Numpy

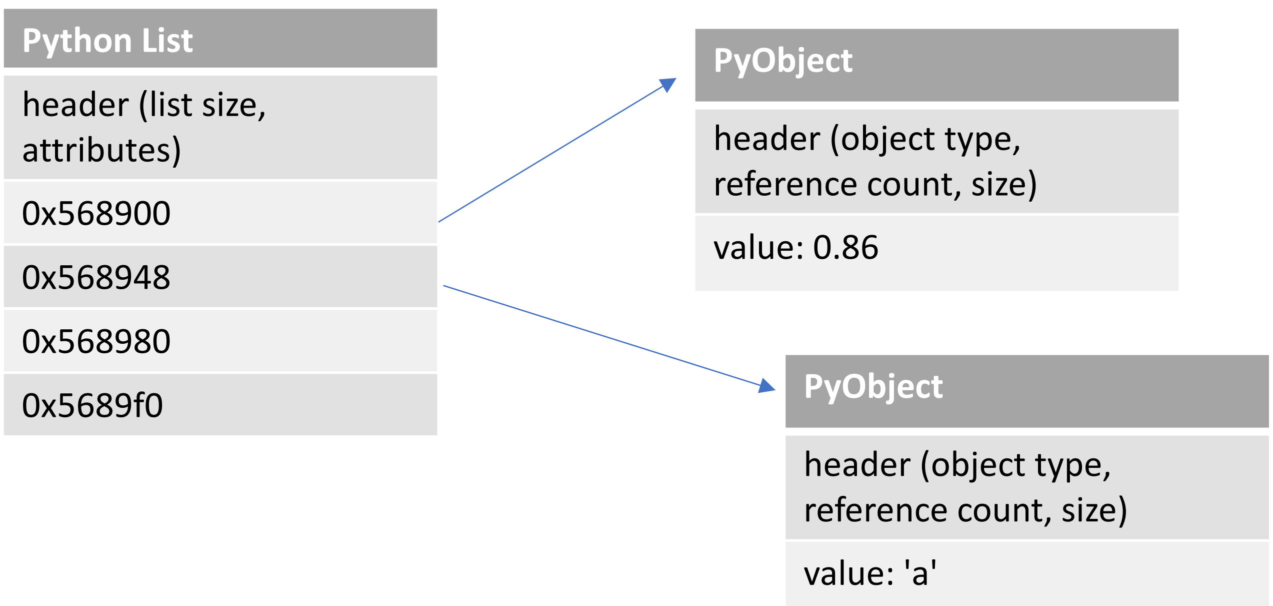
- Numpy (*Numerical Python*)
  - Store and operate on **dense** data buffers
  - **Efficient** storage and operations
- Features
  - Multidimensional arrays
  - Slicing/indexing
  - Math and logic operations
- Applications
  - Computation with vectors and matrices
  - Provides fundamental Python objects for data science algorithms
    - Internally used by scikit-learn and SciPy

- **Summary**
  - Numpy and computation **efficiency**
  - Numpy **arrays**
  - **Computation** with Numpy arrays
    - Broadcasting
  - **Accessing** Numpy arrays
  - Working with arrays, other functionalities

- **array** is the main object provided by Numpy
- Characteristics
  - Fixed Type
    - All its elements have the **same type**
  - Multidimensional
    - Allows representing vectors, matrices and n-dimensional arrays

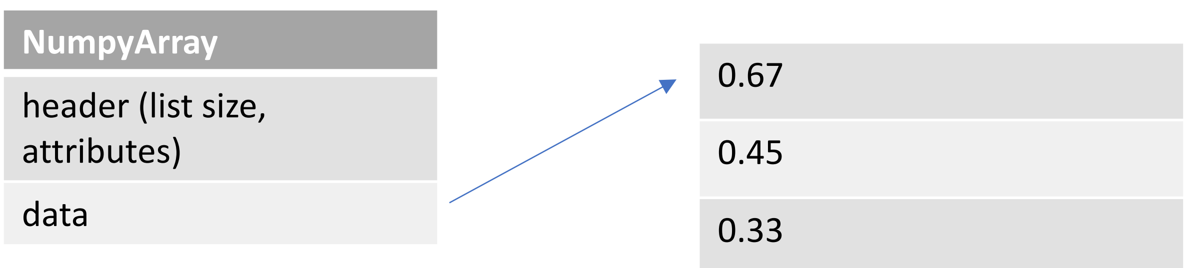
- Numpy arrays vs Python lists:
  - Also Python lists allow defining multidimensional arrays
    - E.g. my_2d_list = [[3.2, 4.0], [2.4, 6.2]]
- Numpy advantages:
  - Higher **flexibility** of indexing methods and operations
  - Higher **efficiency** of operations

- Since lists can contain heterogeneous data types, they keep **overhead** information
  - E.g. my_heterog_list = [0.86, 'a', 'b', 4]



Introduction to Numpy

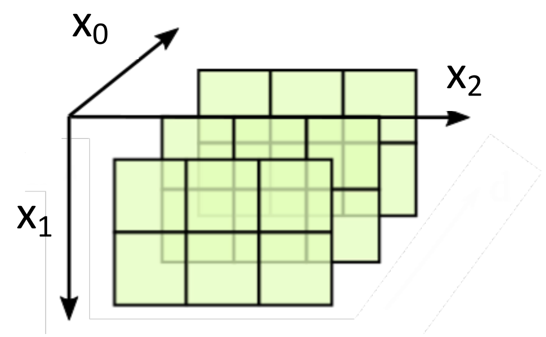
- Characteristics of numpy arrays
  - **Fixed-type** (no overhead)
  - **Contiguous** memory addresses (faster indexing)
  - E.g. my_numpy_array = np.array([0.67, 0.45, 0.33])



- Numpy data types
  - Numpy defines its own data types
  - Numerical types
    - int8, int16, int32, int64
    - uint8, … , uint64
    - float16, float32, float64
  - Boolean values
    - bool

Multidimensional arrays

- Collections of elements organized along an arbitrary number of dimensions
- Multidimensional arrays can be represented with
  - Python lists
  - Numpy arrays



Multidimensional arrays

- Multidimensional arrays with **Python lists**
  - Examples:

vector

1	2	3

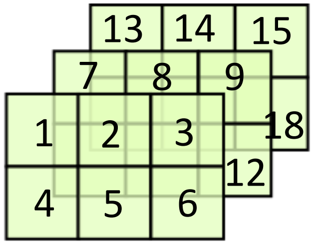
```
list1 = [1, 2, 3]
```

2D matrix

1	2	3
4	5	6

```
list2 = [[1,2,3], [4,5,6]]
```

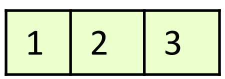
3D array

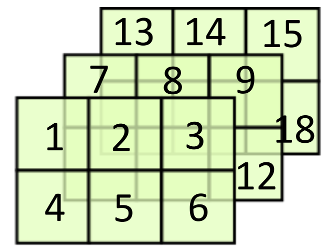



```
list3 = [[[1,2,3], [4,5,6]],
         [[7,8,9], [10,11,12]],
         [[13,14,15], [16,17,18]]]
```

Multidimensional arrays

- Multidimensional arrays with **Numpy**
  - Can be directly created from Python lists
  - Examples:



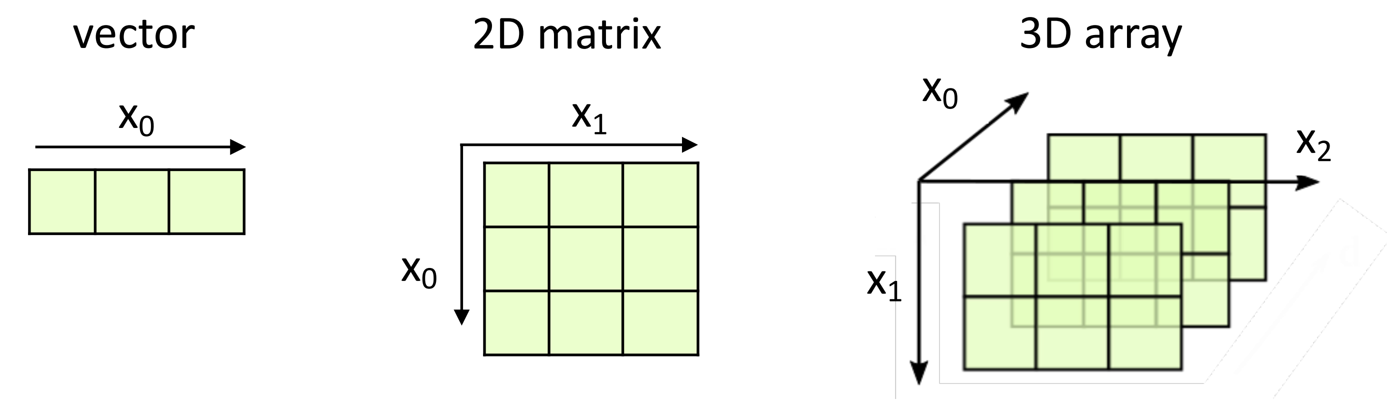



```python
import numpy as np
arr1 = np.array([1, 2, 3])
```

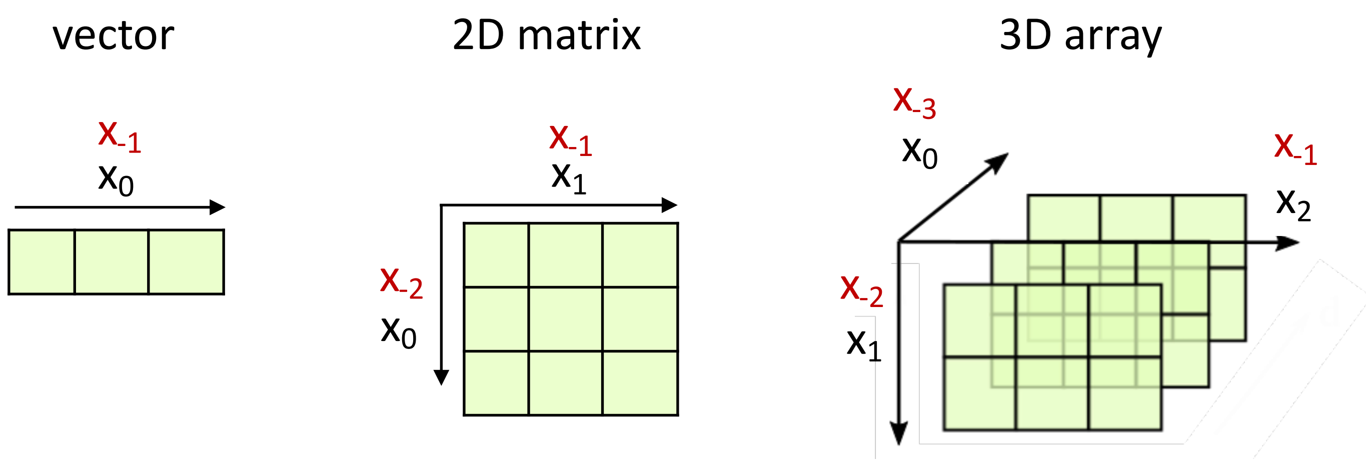
```python
import numpy as np
arr2 = np.array([[[1,2,3], [4,5,6]],
                 [[7,8,9], [10,11,12]],
                 [[13,14,15], [16,17,18]]])
```

Multidimensional arrays

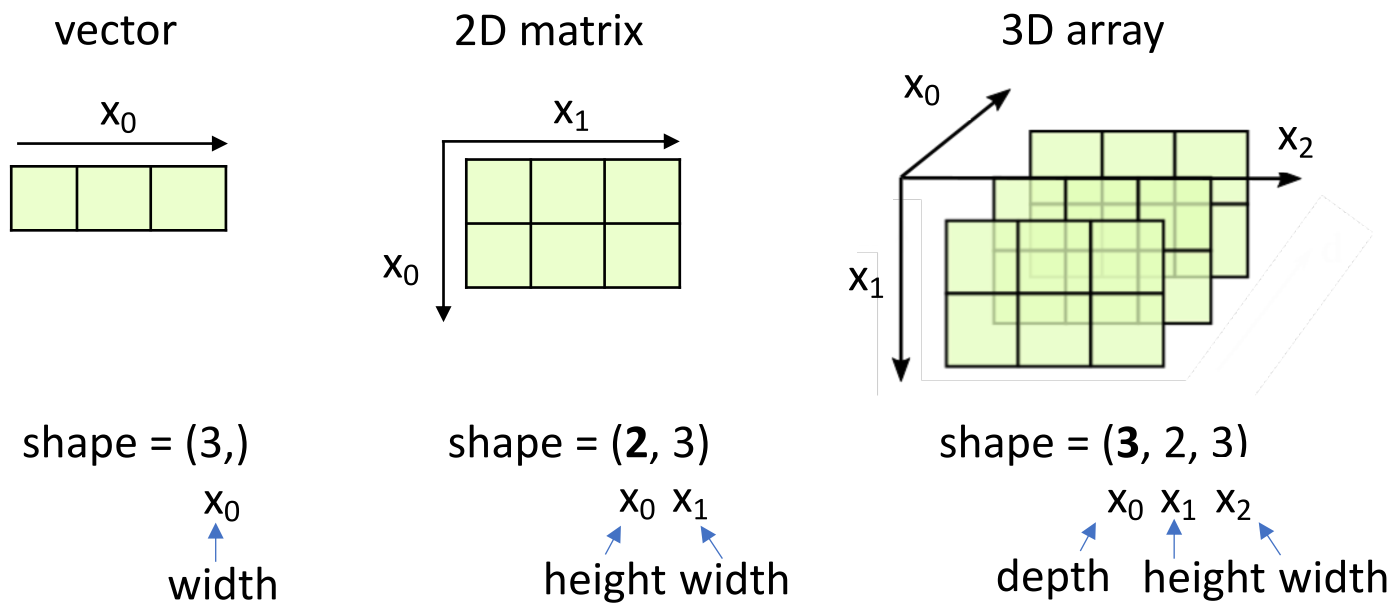
- Multidimensional arrays with **Numpy**
  - Characterized by a set of **axes** and a **shape**
  - The **axes** of an array define its dimensions
    - a (row) vector has 1 axis (1 dimension)
    - a 2D matrix has 2 axes (2 dimensions)
    - a ND array has N axes



- Multidimensional arrays with **Numpy**
  - Axes can be numbered with negative values
  - Axis −1 is always along the **row**



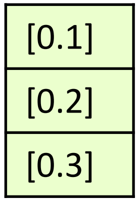
- Multidimensional arrays with **Numpy**
  - The **shape** of a Numpy array is a tuple that specifies the number of elements along each axis
    - Examples:



Multidimensional arrays

- Column vector vs row vector

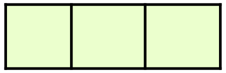
e.g. np.array([[0.1], [0.2], [0.3]])



shape = (3, **1**)

Column vector is a 2D matrix!

e.g. np.array([0.1, 0.2, 0.3])



shape = (3,)

Numpy arrays

- Creation from list:
  - np.array(my_list, dtype=np.float16)
    - Data type inferred if not specified
- Creation from scratch:
  - np.zeros(shape)
    - Array with all 0 of the given shape
  - np.ones(shape)
    - Array with all 1 of the given shape
  - np.full(shape, value)
    - Array with all elements to the specified value, with the specified shape

- Creation from scratch: examples

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

Out[1]:
```
[[1, 1, 1],
 [1, 1, 1]]
```

In [2]:
```
np.full((2,1)), 1.1)
```

Out[2]:
```
[[1.1],
 [1.1]]
```

Numpy arrays

- Creation from scratch:
  - np.linspace(start, stop, num)
    - Generates *num* samples from *start* to *stop* (included)
    - np.linspace(0,1,11) → [0.0, 0.1, … , 1.0]
  - np.arange(start, stop, step)
    - Generates numbers from *start* to *stop* (excluded), with step *step*
    - np.arange(1, 7, 2) → [1, 3, 5]
  - np.random.normal(mean, std, shape)
    - Generates random data with normal distribution
  - np.random.random(shape)
    - Random data uniformly distributed in [0, 1]

Numpy arrays

- Main attributes of a Numpy array
  - Consider the array
    - x = np.array([[2, 3, 4],[5,6,7]])
  - **x.ndim:** number of dimensions of the array
    - Out: 2
  - **x.shape**: tuple with the array shape
    - Out: (2,3)
  - **x.size**: array size (product of the shape values)
    - Out: 2*3=6

Summary:

- **Universal functions** (Ufuncs):
  - **Binary** operations (+,−,*,…)
  - **Unary** operations (exp(),abs(),…)
- **Aggregate** functions
- **Sorting**
- **Algebraic** operations (dot product, inner product)

- **Universal functions** (Ufuncs): element-wise operations
  - **Binary** operations with arrays of the **same shape**
    - +, -, *, /, % (modulus), // (floor division), ** (exponentiation)

Computation on Numpy

- Example:

In [1]:

```
x=np.array([[1,1],[2,2]])
y=np.array([[3, 4],[6, 5]])
x*y
```

Out[1]:

```
[[3, 4], [12, 10]]
```

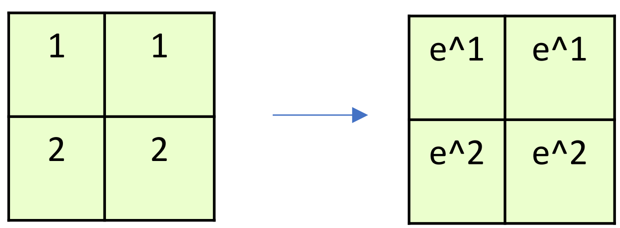
$$\begin{bmatrix} 1 & 1 \\ 2 & 2 \end{bmatrix} * \begin{bmatrix} 3 & 4 \\ 6 & 5 \end{bmatrix} = \begin{bmatrix} 1*3 & 1*4 \\ 2*6 & 2*5 \end{bmatrix} = \begin{bmatrix} 3 & 4 \\ 12 & 10 \end{bmatrix}$$

- **Universal functions** (Ufuncs):
  - **Unary** operations
    - np.abs(x)
    - np.exp(x), np.log(x), np.log2(x), np.log10(x)
    - np.sin(x), cos(x), tan(x), arctan(x), …
  - They apply the operation separately to each element of the array

- Example:

In [1]:
```
x=np.array([[1,1],[2,2]])
np.exp(x)
```

Out[1]:
```
[[2.718, 2.718],[7.389, 7.389]]
```



- **Note: original array (x) is not modified**

- **Aggregate functions**
  - **Return** a single value from an array
    - np.min(x), np.max(x), np.mean(x), np.std(x), np.sum(x)
    - np.argmin(x), np.argmax(x)
  - Or equivalently:
    - x.min(), x.max() x.mean(), x.std(), x.sum()
    - x.argmin(), x.argmax()
  - Example

In [1]:
```
x=np.array([[1,1],[2,2]])
x.sum()
```

Out[1]:
```
6
```

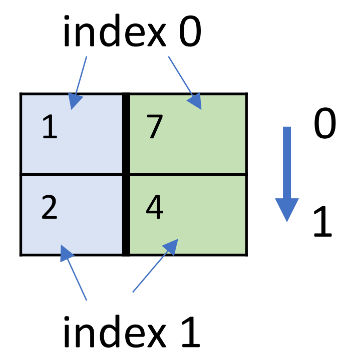
Aggregate functions along axis

- Allow specifying the **axis** along with performing the operation
- Examples

In [1]:
```
x=np.array([[1,7],[2,4]])
x.argmax(axis=0)
```

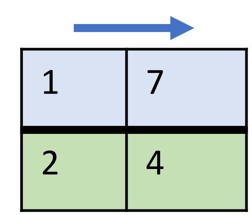
Out[1]:
```
[1, 0]
```

(index of maximum element within each column)



In [2]:
```
x.sum(axis=1)     # or axis=-1
```

Out[2]:
```
[8, 6]
```
(sum the elements of each row)

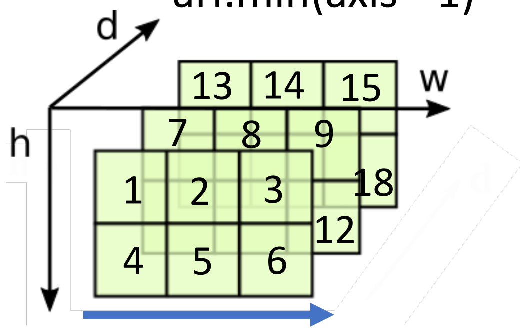


Computation on Numpy

Aggregate functions along axis

- The aggregation dimension is **removed** from the output

arr.min(axis=-1)

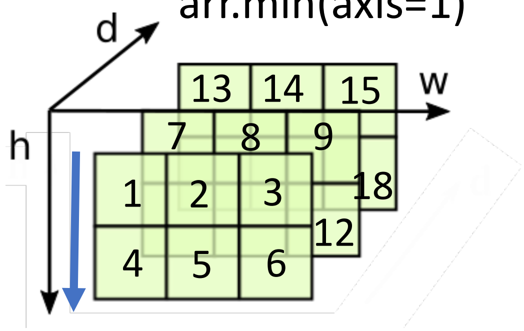


shape = (3, 2, 1)
[[[1], [4]],
[[7], [10]],
[[13], [16]]]

Final output

shape = (3, 2)
[[1, 4],
 [7, 10],
 [13, 16]]

arr.min(axis=1)



shape = (3, **1**, 3)
[[[1,2,3]],
 [[7,8,9]],
 [[13,14,15]]]

shape = (3, 3)
[[1, 2, 3],
 [7, 8, 9],
 [13, 14, 15]]

- **Sorting**
  - **np.sort(x):** creates a sorted copy of x
    - x is not modified
  - **x.sort():** sorts x inplace (x is modified)

Computation on Numpy

Sorting

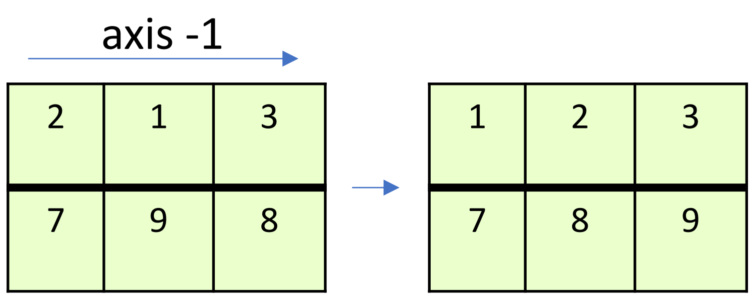
- Array is sorted along the last axis (–1) by default

In [1]:

```
x = np.array([[2,1,3],[7,9,8]])
np.sort(x)          # Sort along rows (axis -1)
```

Out[1]:

```
[[1,2,3],[7,8,9]]
```

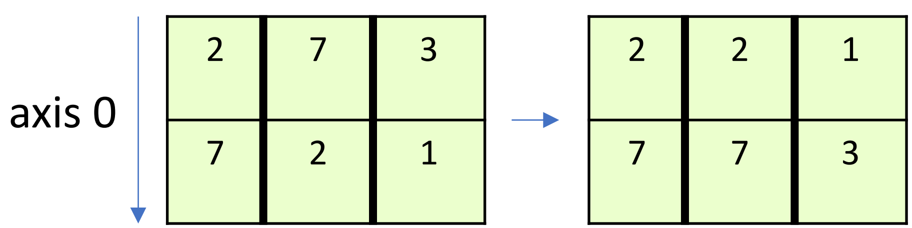


Sorting

- Allows specifying the axis being sorted

In [1]:
```
x = np.array([[2,7,3],[7,2,1]])
np.sort(x, axis=0)    # Sort along columns
```

Out[1]:
```
[[2,2,1],
[7,7,3]]
```

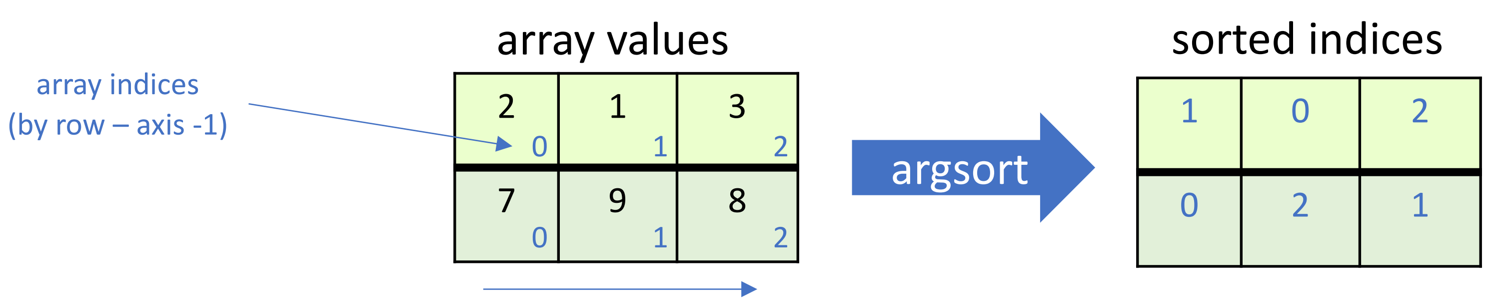


Sorting

- **np.argsort(x):** return the position of the indices of the sorted array (sorts by default on axis −1)

In [1]:
```
x = np.array([[2,1,3],[7,9,8]])
np.argsort(x)       # Sort along rows (axis -1)
```

Out[1]:
```
[[1,0,2],[0,2,1]]
```



Computation on Numpy

- **Algebraic operations**
  - np.dot(x, y)
    - inner product if x and y are two 1-D arrays

$$\begin{bmatrix} 1 & 2 & 3 \end{bmatrix} * \begin{bmatrix} 0 \\ 2 \\ 1 \end{bmatrix} = 7$$

In [1]:

```
x=np.array([1, 2, 3])
y=np.array([0, 2, 1])  # works even if y is a row vector
np.dot(x, y)
```

Out[1]:

```
7
```

Algebraic operations

- np.dot(x, y)
  - matrix multiplied by vector

$$\begin{bmatrix} 1 & 1 \\ 2 & 2 \end{bmatrix} * \begin{bmatrix} 2 \\ 3 \end{bmatrix} = \begin{bmatrix} 5 \\ 10 \end{bmatrix}$$

In [1]:

```
x=np.array([[1,1],[2,2]])
y=np.array([2, 3]) # works even if y is a row vector
np.dot(x, y)
```

Out[1]:

```
[5, 10]    # result is a row vector
```

Algebraic operations

- np.dot(x, y)
  - matrix multiplied by matrix

$$\begin{bmatrix} 1 & 1 \\ 2 & 2 \end{bmatrix} * \begin{bmatrix} 2 & 2 \\ 1 & 1 \end{bmatrix} = \begin{bmatrix} 3 & 3 \\ 6 & 6 \end{bmatrix}$$

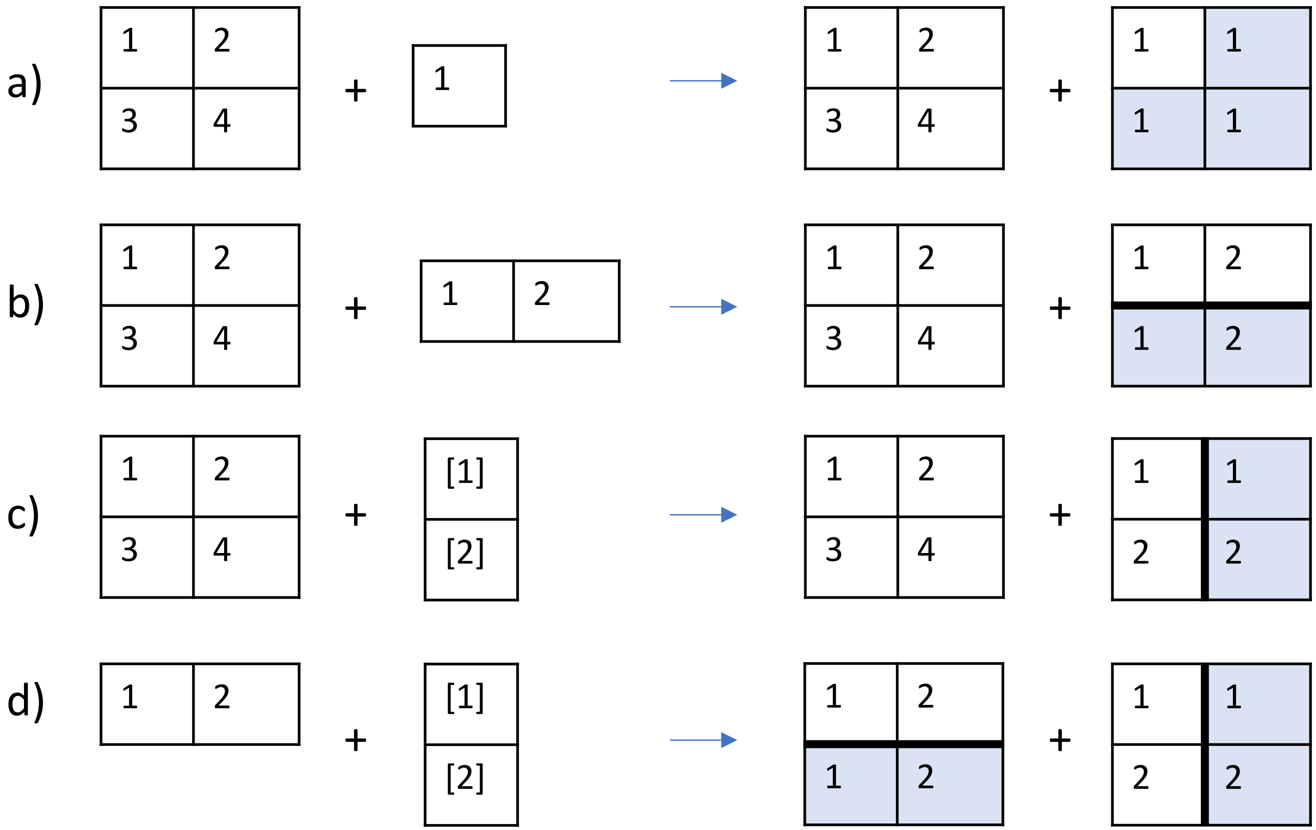
In [1]:
```
x=np.array([[1,1],[2,2]])
y=np.array([[2,2],[1,1]])
np.dot(x, y)
```

Out[1]:
```
[[3,3],[6,6]]
```

Notebook Examples

- **2-Numpy Examples.ipynb**
  - **1) Computation with arrays**

- Pattern designed to perform operations between arrays with **different shape**

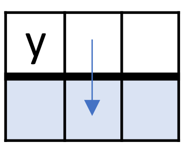


Broadcasting

- Rules of broadcasting
  1. The shape of the array with **fewer dimensions** is **padded** with <u>leading</u> ones

     x.shape = (2, 3), y.shape = (3) → y.shape = (**1**, 3)

  2. If the shape along a dimension is 1 for one of the arrays and >1 for the other, the array with shape = 1 in that dimension is **stretched to match the other array**



     x.shape = (2, 3), y.shape = (**1**, 3) → stretch: y.shape = (**2**, 3)

  3. If there is a dimension where both arrays have shape >1 and those shapes differ, then broadcasting **cannot be performed**

- Example: compute x + y
  - x = np.array([1, 2, 3])
  - y = np.array([[11], [12], [13]])
  - z = x + y

x.shape = (3,) y.shape = (3,1)

$$\begin{bmatrix}1 & 2 & 3\end{bmatrix} + \begin{bmatrix}[11]\\ [12]\\ [13]\end{bmatrix}$$

- Apply Rule 1
  - x.shape becomes (**1**, 3): x=[[1,2,3]]
- Apply Rule 2:
  - extend x on the vertical axis, y on the horizontal one

x.shape = (1,3) y.shape = (3,1)

$$\boxed{\begin{bmatrix}1 & 2 & 3\end{bmatrix}} + \begin{bmatrix}[11]\\ [12]\\ [13]\end{bmatrix}$$

$$\begin{bmatrix}1 & 2 & 3\\ 1 & 2 & 3\\ 1 & 2 & 3\end{bmatrix} + \begin{bmatrix}11 & 11 & 11\\ 12 & 12 & 12\\ 13 & 13 & 13\end{bmatrix} = \begin{bmatrix}12 & 13 & 14\\ 13 & 14 & 15\\ 14 & 15 & 16\end{bmatrix}$$

- Example: compute x + y
  - x = np.array([[1, 2],[3,4],[5,6]])
  - y = np.array([11, 12, 13])
  - z = x + y

x.shape = (3, 2)

y.shape = (3,)

  - Apply Rule 1
    - y.shape becomes (**1**, 3): y=[[11,12,13]]
  - Apply Rule 3
    - shapes (3, **2**) and (1, **3**) are incompatibles
    - Numpy will raise an **exception**

11	12	13

1	2
3	4
5	6

Notebook Examples

- **2-Numpy Examples.ipynb**
  - **2) Broadcasting: dataset normalization**

Accessing Numpy Arrays

- Numpy arrays can be accessed in many ways
  - Simple indexing
  - Slicing
  - Masking
  - Fancy indexing
  - Combined indexing
- Slicing provides **views** on the considered array
  - Views allow **reading** and **writing** data on the **original** array
- Masking and fancy indexing provide **copies** of the array

Accessing Numpy Arrays

- **Simple indexing**: read/write access to element
  - x[i, j, k, ...]

In [1]:
```
x = np.array([[2, 3, 4],[5,6,7]])
el = x[1, 2]          # read value (indexing)
print("el =", el)
```

Out[1]:
```
el = 7
```

In [2]:
```
x[1, 2] = 1           # assign value
print(x)
```

Out[2]:
```
[[2, 3, 4], [5, 6, 1]]
```

Accessing Numpy Arrays

- **Simple indexing:** returning elements **from the end**
- Consider the array
  - x = np.array([[2, 3, 4],[5,6,7]])
- x[0, **-1**]
  - Get last element of the first row: 4
- x[0, **-2**]
  - Get second element from the end of the first row: 3

Accessing Numpy Arrays

- **Slicing**: access contiguous elements
  - x[start:stop:step, …]
    - Creates a *view* of the elements from *start* (included) to *stop* (excluded), taken with fixed step
    - **Updates on the view yield updates on the original array**
    - Useful shortcuts**:**
      - **omit start** if you want to start from the beginning of the array
      - **omit stop** if you want to slice until the end
      - **omit step** if you don't want to skip elements

Accessing Numpy Arrays

- **Slicing**: access contiguous elements
  - Select **all rows** and the **last 2 columns**:

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

Out[1]:
```
[[2,3], [5,6], [8,9]]
```

1	2	3
4	5	6
7	8	9

  - Select the **first two rows** and the **first** and **third columns**

In [2]:
```
x[:2, ::2]      # or x[0:2, 0:3:2]
```

Out[2]:
```
[[1, 3], [4, 6]]
```

1	2	3
4	5	6
7	8	9

Accessing Numpy Arrays

- **Update a sliced array**

In [1]:

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

Out[1]:

```
[[1,0,0], [4,0,0], [7,0,0]]
```

Accessing Numpy Arrays

Update a view

In [1]:

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

Out[1]:

```
[[1,0,0], [4,0,0], [7,0,0]]
```

- To avoid updating the original array use .**copy**()
  - x1=x[:,1:].copy()

Accessing Numpy Arrays

- **Masking**: use boolean masks to select elements
  - x[mask]
    - mask
      - **boolean** numpy array that specifies which elements should be selected (select if True)
      - **same shape** as the original array
  - The result is a **one-dimensional vector** that is a **copy** of the original array elements selected by the mask

Accessing Numpy Arrays

- **Mask creation**
  - x *op* value (e.g *x==4*)
  - where *op* can be >, >=, <, <=, ==, !=
- Examples

In [1]:
```
x = np.array([1.2, 4.1, 1.5, 4.5])
x > 4
```

Out[1]:
```
[False, True, False, True]
```

In [2]:
```
x2 = np.array([[1.2, 4.1], [1.5, 4.5]])
x2 >= 4
```

Out[2]:
```
[[False, True], [False, True]]
```

- **Operations with masks (boolean arrays)**
  - Numpy allows boolean operations between masks with the same shape (bitwise operators)
    - & (and), | (or), ^ (xor), ~ (negation)
  - Example
    - mask = ~((x < 1) | (x > 5)) ⇔ ((x >= 1) & (x <= 5))
    - elements that are between 1 and 5 (included)

Accessing Numpy Arrays

- **Masking** examples

In [1]:
```
x = np.array([1.2, 4.1, 1.5, 4.5])
x[x > 4]
```

Out[1]:
```
[4.1, 4.5]
```

In [2]:
```
x2 = np.array([[1.2, 4.1], [1.5, 4.5]])
x2[x2 >= 4]
```

Out[2]:
```
[4.1, 4.5]
```

- Even if the shape of x2 is (2, 2), the result is **a one-dimensional** array containing the elements that satisfy the condition

Accessing Numpy Arrays

Update a masked array

In [1]:

```
x = np.array([1.2, 4.1, 1.5, 4.5])
x[x > 4] = 0        # Assignment is allowed
x
```

Out[1]:

```
[1.2, 0, 1.5, 0]
```

Accessing Numpy Arrays

- **Masking does not create views, but copies**

In [2]:

```
x = np.array([1.2, 4.1, 1.5, 4.5])
masked = x[x > 4]  # Masked is a copy of x
masked[:] = 0      # Assignment does not affect x
x
```

Out[2]:

```
[1.2, 4.1, 1.5, 4.5]
```

- **Fancy indexing**: specify the **index** of elements to be selected
  - Example: select elements from 1-dimensional array

x[1] x[3]

In [1]:

```
x = np.array([7.0, 9.0, 6.0, 5.0])
x[[1, 3]]
```

Out[1]:

```
[9.0, 5.0]
```

Accessing Numpy Arrays

- **Fancy indexing:** selection of **rows** from a 2-dimensional array

	0.0	1.0	2.0
x[1,:]	3.0	4.0	5.0
x[2,:]	6.0	7.0	8.0

In [1]:

```
x = np.array([[0.0, 1.0, 2.0], [3.0, 4.0, 5.0],
              [6.0, 7.0, 8.0]])
x[[1, 2]]
```

Out[1]:

```
[[3.0, 4.0, 5.0], [6.0, 7.0, 8.0]]
```

- **Fancy indexing**: selection of elements with coordinates
  - Result contains a 1-dimensional array with selected elements

0,0	0,1	0,2
1,0	1,1	1,2
2,0	2,1	2,2

In [1]:

```
x = np.array([[0.0, 1.0, 2.0], [3.0, 4.0, 5.0],
              [6.0, 7.0, 8.0]])
x[[1, 2], [0, 2]]
```

→ | 1, 0 | 2, 2 | (indices being selected)

Out[1]:

```
[3.0, 8.0]
```

Accessing Numpy Arrays

PoliTo DBMG

- Similarly to masking, fancy indexing provides **copies** (not views) of the original array

In [1]:
```
x = np.array([1.2, 4.1, 1.5, 4.5])
x[[1, 3]] = 0      # Assignment is allowed
x
```

Out[1]:
```
[1.2, 0, 1.5, 0]
```

In [2]:
```
x = np.array([1.2, 4.1, 1.5, 4.5])
sel = x[[1, 3]]    # sel is a copy of x
sel[:] = 0         # Assignment does not affect x
x
```

Out[2]:
```
[1.2, 4.1, 1.5, 4.5]
```

- **Combined indexing:**
  - Allows mixing the indexing types described so far
  - Important rule:
    - The number of dimensions of selected data is:
      - **The same as the input** if you mix:
        - masking+slicing, fancy+slicing
      - **Reduced by one** for each axis where simple indexing is used
        - Because simple indexing takes only 1 **single** element from an axis

Accessing Numpy Arrays

- **Combined indexing:** masking+slicing, fancy+slicing
  - Output has the same numer of dimensions as input

```
x = np.array([[0.0, 1.0, 2.0], [3.0, 4.0, 5.0], [6.0, 7.0, 8.0]])
```

```
x[[True,False,True], 1:]
# Masking + Slicing: [[1.0,2.0],[7.0,8.0]]
```

0.0	1.0	2.0
3.0	4.0	5.0
6.0	7.0	8.0

```
x[[0,2], :2]
# Fancy + Slicing: [[0.0,1.0],[6.0,7.0]]
```

0.0	1.0	2.0
3.0	4.0	5.0
6.0	7.0	8.0

Accessing Numpy Arrays

- **Combined indexing:** simple+slicing, simple+masking
  - Simple indexing **reduces** the number of dimensions

```
x = np.array([[0.0, 1.0, 2.0], [3.0, 4.0, 5.0], [6.0, 7.0, 8.0]])
```

```
x[0, 1:]
# Simple + Slicing: [1.0, 2.0]
```

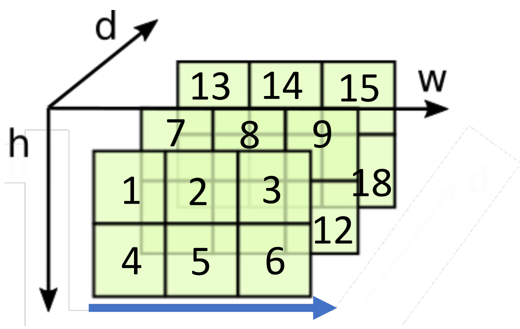
0.0	1.0	2.0
3.0	4.0	5.0
6.0	7.0	8.0

```
x[[True, False, True], 0]
# Simple + Masking: [0.0, 6.0]
```

0.0	1.0	2.0
3.0	4.0	5.0
6.0	7.0	8.0

- **Simple indexing + slicing**
  - The dimension selected with simple indexing is **removed** from the output

arr[:,:,0]

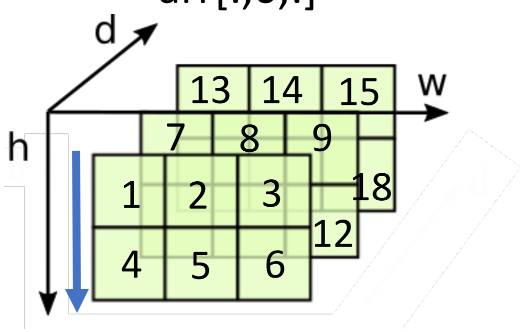


shape = (3, 2, 1)
[[[1], [4]],
[[7], [10]],
[[13], [16]]]

Final output

shape = (3, 2)
[[1, 4],
 [7, 10],
 [13, 16]]

arr[:,0,:]



shape = (3, **1**, 3)
[[[1,2,3]],
 [[7,8,9]],
 [[13,14,15]]]

shape = (3, 3)
[[1, 2, 3],
 [7, 8, 9],
 [13, 14, 15]]

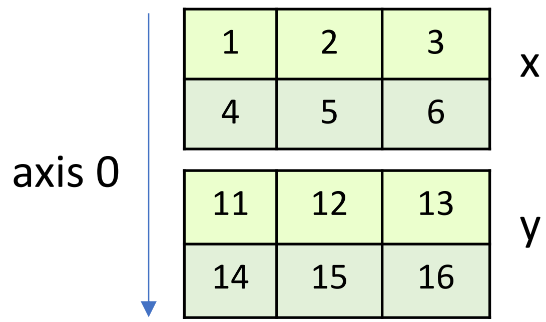
Notebook Examples

- **2-Numpy Examples.ipynb**
  - **3) Accessing Numpy Arrays**

Summary:

- Array concatenation
- Array splitting
- Array reshaping
- Adding new dimensions

- Array concatenation along **existing axis**
  - The result has the **same number of dimensions** of the input arrays

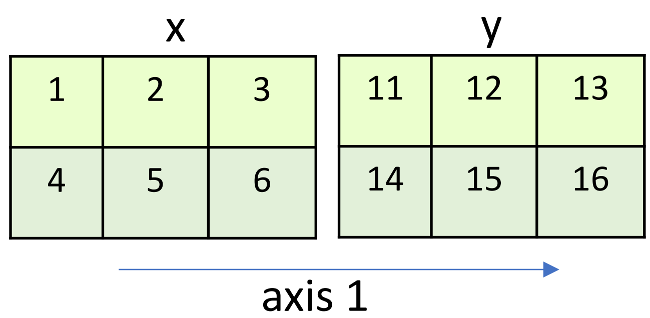


In [1]:
```
x = np.array([[1,2,3],[4,5,6]])
y = np.array([[11,12,13],[14,15,16]])
np.concatenate((x, y))      # Default axis: 0
```

Out[1]:
```
[[1,2,3],[4,5,6],[11,12,13],[14,15,16]]
```

Array concatenation along existing axis

- Concatenation along **rows (axis=1)**



In [1]:

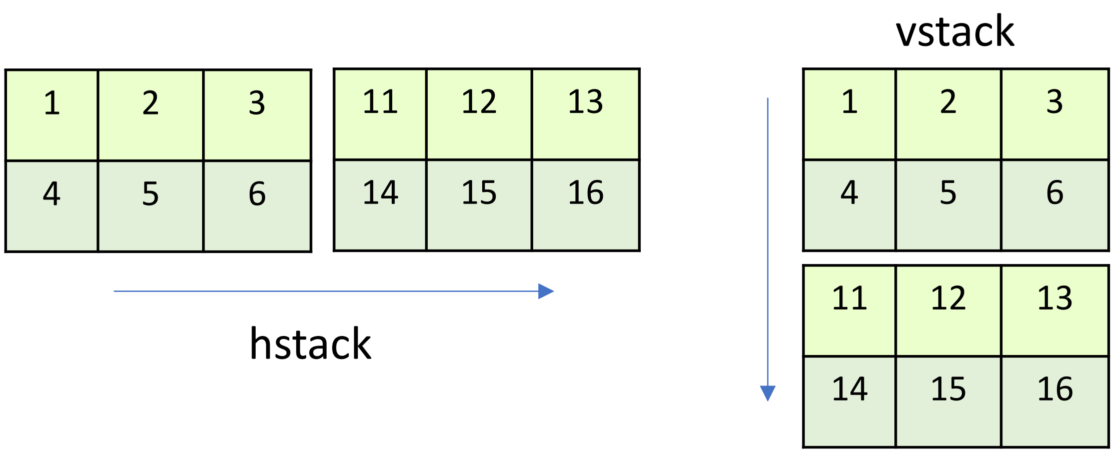
```
x = np.array([[1,2,3],[4,5,6]])
y = np.array([[11,12,13],[14,15,16]])
np.concatenate((x, y), axis=1)
```

Out[1]:

```
[[1,2,3,11,12,13],[4,5,6,14,15,16]]
```

Array concatenation: hstack, vstack

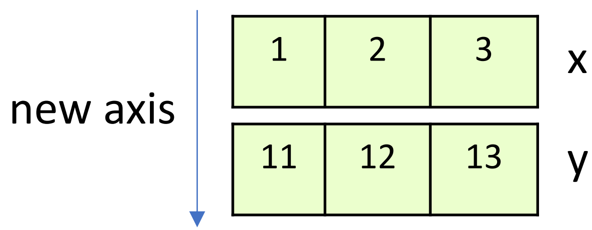
- Similar to np.concatenate()



In [1]:

```
x = np.array([[1,2,3],[4,5,6]])
y = np.array([[11,12,13],[14,15,16]])
h = np.hstack((x, y))       # along rows (horizontal)
v = np.vstack((x, y))       # along columns (vertical)
```

- **Array concatenation: hstack, vstack**
  - **vstack** allows concatenating 1-D vectors along **new axis** (not possible with np.concatenate)



In [1]:

```
x = np.array([1,2,3])
y = np.array([11,12,13])
v = np.vstack((x, y))       # vertically
```

Working with arrays

PoliTo DBMG

- **Splitting arrays (split, hsplit, vsplit)**
  - **np.split(arr, N, axis=0)**
    - outputs a **list** of Numpy arrays
    - If N is integer: divide *arr* into *N* equal arrays (along axis), if possible!
    - if N is a 1d array: specify the entries where the array is split (along *axis*)

x

index	0	1	**2**	3	**4**	5
values	7	7	9	9	8	8

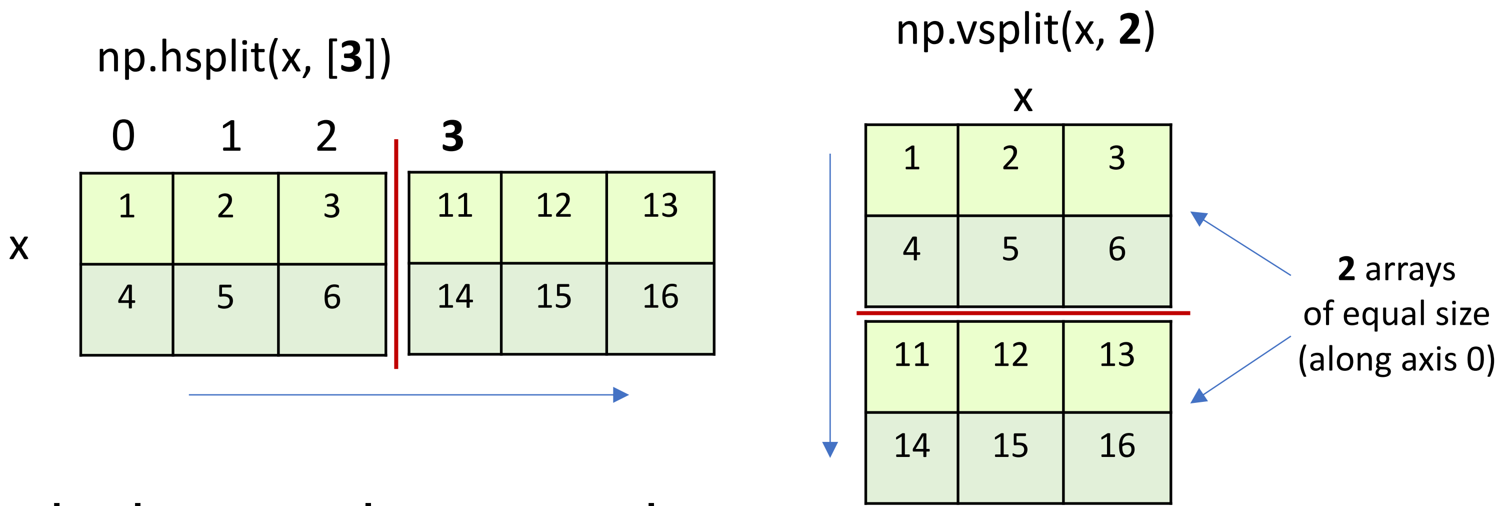
In [1]:

```
x = np.array([7, 7, 9, 9, 8, 8])
np.split(x,[2,4])         # split before element 2 and 4
                          # same as passing N = 3
```

Out[1]:

```
[array([7, 7]), array([9, 9]), array([8, 8])]
```

- **Splitting arrays (split, hsplit, vsplit)**
  - **hsplit, vsplit** with 2D arrays
    - return a **list** with the arrays after the split



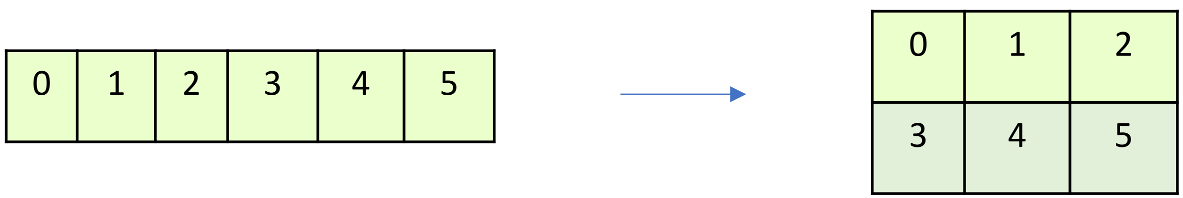
  - In both examples output is:

Out: [array([[1,2,3],[4,5,6]]), array([[11,12,13],[14,15,16]])]

Reshaping arrays

In [1]:

```
x = np.arange(6)
y = x.reshape((2,3))
```

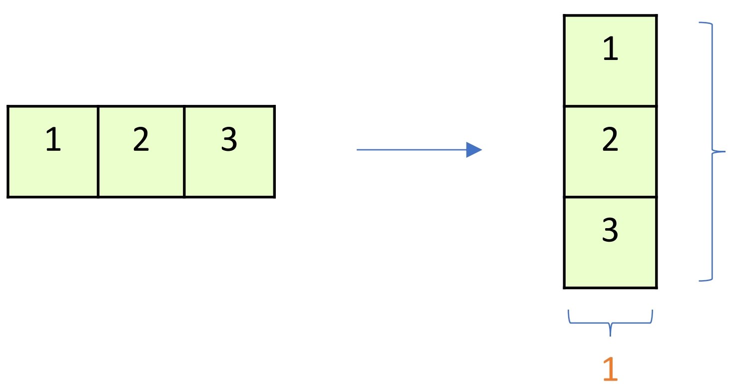


- y is filled following the index order:
  - y[0,0] = x[0], y[0,1] = x[1], y[0,2] = x[2]
  - y[1,0] = x[3], y[1,1] = x[4], y[1,2] = x[5]

- **Reshaping arrays**
  - At most one dimension can be −1 (“unknown”)
  - If present, the size is inferred from
    - The source array
    - The other dimensions

In [1]:

```
x = np.array([1,2,3])
y = x.reshape(-1,1)
```



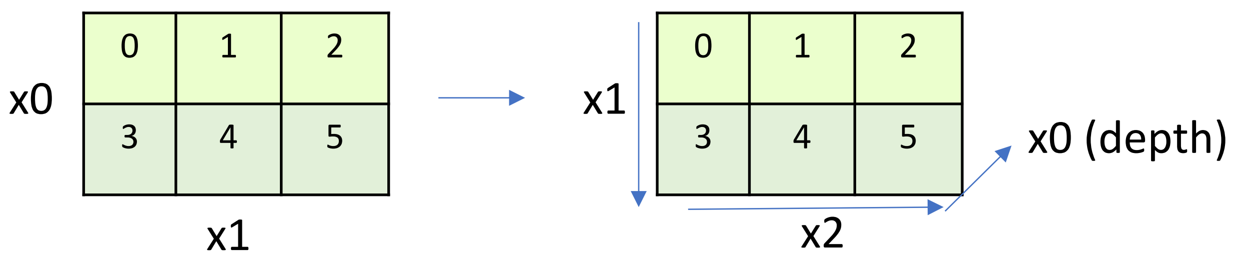
The first dimension (rows) is inferred to be 3, considering that the second dimension (columns) is 1 and x.size = 3

Adding new dimensions

- **np.newaxis** adds a new dimension with **shape=1** at the specified position

In [1]:
```
arr = np.array([[1,2,3],[4,5,6]])
res = arr[np.newaxis, :, :] # output shape = (1,2,3)
print(res)
```

Out[1]:
```
[[[1,2,3],[4,5,6]]]
```



- **Adding new dimensions**
  - **Application:** row vector to column vector
    - Alternative approach to .reshape(−1,1)

In [1]:

```
arr = np.array([1,2,3])
res = arr[:, np.newaxis]    # output shape = (3,1)
print(res)
```

Out[1]:

```
[[1],[2],[3]]
```

- Pandas
  - Provides useful data structures (Series and DataFrames) and data analysis tools
  - Based on **Numpy** arrays
  - Tools:
    - Managing **tables** and **series**
      - data selection
      - grouping, pivoting
    - Managing **missing data**
    - **Statistics** on data

Pandas Series

- **Series**: 1-Dimensional sequence of homogeneous elements
- Elements are associated to an explicit **index**
  - index elements can be either strings or integers
- Examples:

index	1	2	3
values	0.3	0.5	0.8

index	'3-July'	'4-July'	'5-July'
values	0.3	0.5	0.8

Pandas Series

- **Creation** from list
  - When not specified, index is set automatically with a progressive number

In [1]:

```python
import pandas as pd
s1 = pd.Series([2.0, 3.1, 4.5])
print(s1)
```

Out[1]:

```
0    2.0
1    3.1
2    4.5
```

Pandas Series

- **Creation** from list, specifying index

In [1]:
```
pd.Series([2.0, 3.1, 4.5], index=['mon', 'tue', 'wed'])
```

Out[1]:
```
'mon'    2.0
'tue'    3.1
'wed'    4.5
```

Pandas Series

- **Creation** from dictionary
  - keys define the index

In [1]:
```
pd.Series({'c':2.0, 'b':3.1, 'a':4.5})
```

Out[1]:
```
'c'    2.0
'b'    3.1
'a'    4.5
```

- Obtaining **values** and **index** from a Series

In [1]:
```
s1 = pd.Series([2.0, 3.1, 4.5], index=['mon', 'tue', 'wed'])
print(s1.values)   # Numpy array
print(s1.index)
```

Out[1]:
```
[2.0, 3.1, 4.5]
Index(['mon', 'tue', 'wed'], dtype='object')
```

- **Index** is a custom Python object defined in Pandas

PoliTo DBMG

- Accessing Series elements
- **Access by Index**
  - **Explicit:** the one specified while creating a Series
    - Use the Series.**loc** attribute
  - **Implicit:** number associated to the element order (similarly to Numpy arrays)
    - Use the Series.**iloc** attribute

- Accessing Series elements

In [1]:

```
s1 = pd.Series([2.0, 3.1, 4.5], index=['a', 'b', 'c'])
print(s1.loc['a'])          # With explicit index
print(s1.iloc[0])           # With implicit index
s1.loc['b'] = 10            # Allows editing values
print(f"Series:\n{s1}")
```

Out[1]:

```
2.0
2.0
Series:
'a'    2.0
'b'    10
'c'    4.5
```

- Accessing Series elements: **slicing**

In [1]:

```
s1 = pd.Series([2.0, 3.1, 4.5], index=['a', 'b', 'c'])
print(s1.loc['b':'c']) # explicit index (stop element included)
print(s1.iloc[1:3])    # implicit index (stop element excluded)
```

Out[1]:

```
b 3.1
c 4.5

b 3.1
c 4.5
```

PoliTo DBMG

- Accessing Series elements: **masking**

In [1]:
```
s1 = pd.Series([2.0, 3.1, 4.5], index=['a', 'b', 'c'])
print(s1[(s1>2) & (s1<10)])
```

Out[1]:
```
b 3.1
c 4.5
```

Pandas Series

- Accessing Series elements: **fancy indexing**

In [1]:

```
s1 = pd.Series([2.0, 3.1, 4.5], index=['a', 'b', 'c'])
print(s1.loc[['a', 'c']])
print(s1.iloc[[0, 2]])
```

Out[1]:

```
a 2.0
c 4.5


a 2.0
c 4.5
```

Pandas DataFrame

- **DataFrame**: 2-Dimensional array
  - Can be thought as a table where **columns are Series** objects that share the **same index**
    - Each column has a **name**

- Example:

Index	'Price'	'Quantity'	'Liters'
'Water'	1.0	5	1.5
'Beer'	1.4	10	0.3
'Wine'	5.0	8	1

Pandas DataFrame

- **Creation** from Series
  - Use a **dictionary** to set column names

In [1]:
```
price = pd.Series([1.0, 1.4, 5], index=['a', 'b', 'c'])
quantity = pd.Series([5, 10, 8], index=['a', 'b', 'c'])
liters = pd.Series([1.5, 0.3, 1], index=['a', 'b', 'c'])
df = pd.DataFrame({'Price':price, 'Quantity':quantity,
                   'Liters':liters})
print(df)
```

Out[1]:
```
   Price   Quantity   Liters
a    1.0          5      1.5
b    1.4         10      0.3
c    5.0          8      1.0
```

Pandas DataFrame

- **Creation** from dictionary of key-list pairs
  - **Each value (list)** is associated to a **column**
    - Column name given by the key
  - **Index** is automatically set to a progressive number
    - Unless explicitly passed as parameter (index=…)
- Example:

In [1]:

```
dct = { "c1": [0, 1, 2], "c2": [0, 2, 4] }
df = pd.DataFrame(dct)
print(df)
```

Out[1]:

```
     c1     c2
0    0      0
1    1      2
2    2      4
```

Pandas DataFrame

- **Creation** from list of dictionaries
  - **Each dictionary** is associated to a **row**
  - **Index** is automatically set to a progressive number
    - Unless explicitly passed as parameter (index=...)
- Example:

In [1]:

```
dic_list = [{'c1':i, 'c2':2*i} for i in range(3)]
df = pd.DataFrame(dic_list)
print(df)
```

Out[1]:

```
    c1   c2
0   0    0
1   1    2
2   2    4
```

Pandas DataFrame

- **Creation** from 2D Numpy array
- Example:

In [1]:

```
arr = np.arange(6).reshape((3,2))
df = pd.DataFrame(arr, columns=['c1', 'c2'],
                       index=['a', 'b', 'c'])
print(df)
```

Out[1]:

```
    c1    c2
a    0     1
b    2     3
c    4     5
```

- Load DataFrame from **csv** file
  - Allows specifying the column **delimiter (sep)**
  - Automatically read **header** from first line of the file (after **skipping** the specified number of rows)
  - Column data types are inferred

```
df = pd.read_csv('./mycsv.csv', sep=',', skiprows=1)
```

mycsv.csv

```
MyTitle
c1,c2,c3
0,1,2
3,4,5
6,7,8
```

	c1	c2	c3
0	0	1	2
1	3	4	5
2	6	7	8

- Load DataFrame from **csv** file
  - If it contains **null** values, you can specify how to recognize them
  - Empty columns are converted to “NaN” (Not a Number)
    - Using np.nan (NumPy’s representation of NaN)
  - The string 'NaN' is automatically recognized

```
df = pd.read_csv('./mycsv.csv', sep=',',
                 na_values=['no info', 'x'])
```

mycsv.csv

```
c1,c2,c3
0,no info,
3,4,5
6,x,NaN
```

	c1	c2	c3
0	0	NaN	NaN
1	3	4.0	5.0
2	6	NaN	NaN

type(np.nan) → float,
hence c2 and c3 are floats

- **Save** DataFrame to csv

```
df.to_csv('./savedcsv.csv', sep=',')
```

	c1	c2	c3
0	0	NaN	2
1	3	4	5
2	6	NaN	NaN

savedcsv.csv

```
 c1,c2,c3
0,0,,2
1,3,4,5
2,6,,
```

- Use **index=False** to avoid writing the index

```
df.to_csv('./savedcsv.csv', sep=',', index=False)
```

- Load DataFrame from **json** file

```
df = pd.read_json('./myjson.json')
```

myjson.json

```
{"c1":{"0":0, "1":3, "2":6},
"c2":{"0":null, "1":4, "2":null},
"c3":{"0":2, "1":5, "2":null}}
```

	c1	c2	c3
0	0	NaN	2
1	3	4	5
2	6	NaN	NaN

- Use **pd.to_json(path)** to save a DataFrame in json format

DataFrames and I/O

- Many other data types are supported
  - Excel, HTML, HDF5, SAS, …
- Check the pandas documentation
  - https://pandas.pydata.org/pandas-docs/stable/user_guide/io.html

- Obtaining **column names** and **index** from a DataFrame

Index	Price	Quantity	Liters
a	1.0	5	1.5
b	1.4	10	0.3
c	5.0	8	1

In [2]:

```
print(df.columns)  # Index object with column names
print(df.index)    # Index object
```

Out[2]:

```
Index(['Price', 'Quantity', 'Liters'], dtype='object')
Index(['a', 'b', 'c'], dtype='object')
```

- **Accessing** DataFrame data
  - Get a 2D Numpy array

Index	Price	Quantity	Liters
a	1.0	5	1.5
b	1.4	10	0.3
c	5.0	8	1

In [2]:
```
print(df.values)   # Numpy array with data
```

Out[2]:
```
array([[1.0, 5.0, 1.5],
       [1.4, 10.0, 0.3],
       [5.0, 8.0, 1.0]])
```

- **Accessing** DataFrames
  - Access a DataFrame column
  - Access rows and columns with indexing
    - **df.loc**
      - **Explicit** index
      - Slicing, masking, fancy indexing
    - **df.iloc**
      - **Implicit** index
- Whether a **copy** or **view** will be returned it depends on the context
  - Usually it is difficult to make assumptions
  - https://pandas-docs.github.io/pandas-docs-travis/user_guide/indexing.html

- **Accessing** DataFrame columns
  - Returns a **Series** with column data

Index	Price	Quantity	Liters
a	1.0	5	1.5
b	1.4	10	0.3
c	5.0	8	1

In [1]:
```
df['Quantity']
```

Out[1]:
```
a      5
b     10
c      8
```

Pandas DataFrame

- **Accessing** single DataFrame **row** by index
  - **loc** (explicit), **iloc** (implicit)
  - Return a **Series** with an element for each column

In [1]:

```
print(df.loc['a'])          # Get the first row (explicit)
print(df.iloc[0])           # Get the first row
```

Out[1]:

```
Price    1.0
Quantity 5.0
Liters   1.5


Price    1.0
Quantity 5.0
Liters   1.5
```

- **Accessing** DataFrames with **slicing**
  - Allows selecting rows and columns

In [1]:

```
print(df.loc['b':'c', 'Quantity':'Liters'])
```

Out[1]:

```
      Quantity   Liters
b           10      0.3
c            8        1
```

- **Accessing** DataFrames with **masking**
  - Select rows based on a condition

Index	Price	Quantity	Liters
a	1.0	**5**	**1.5**
b	1.4	10	0.3
c	5.0	8	1

In [1]:
```
mask = (df['Quantity']<10) & (df['Liters']>1)
df.loc[mask, 'Quantity':]   # Use masking and slicing
```

Out[1]:
```
     Quantity   Liters
a           5      1.5
```

Pandas DataFrame

PoliTo DBMG

- **Accessing** DataFrames with **fancy indexing**
  - To select **columns**…

Index	Price	Quantity	Liters
a	**1.0**	5	**1.5**
b	1.4	10	0.3
c	5.0	8	1

In [1]:

```
mask = (df['Quantity']<10) & (df['Liters']>1)
df.loc[mask, ['Price','Liters']]   # Use masking and fancy
```

Out[1]:

```
    Price      Liters
a     1.0         1.5
```

Pandas DataFrame

- **Accessing** DataFrames with **fancy indexing**
  - To select **rows** and **columns**…

Index	Price	Quantity	Liters
a	**1.0**	5	**1.5**
b	1.4	10	0.3
c	**5.0**	8	**1**

In [1]:
```
df.loc[['a', 'c'], ['Price','Liters']]
```

Out[1]:
```
     Price      Liters
a       1.0        1.5
c       5.0        1.0
```

- **Assign value** to selected items

In [1]:
```
df.loc[['a', 'c'], ['Price','Liters']] = 0
```

Index	Price	Quantity	Liters
a	**0.0**	5	**0.0**
b	1.4	10	0.3
c	**0.0**	8	**0.0**

- **Add new column** to DataFrame
  - DataFrame is modified **inplace**

Index	Price	Quantity	Liters
a	**0.0**	5	**0.0**
b	1.4	10	0.3
c	**0.0**	8	**0.0**

→

Index	Price	Quantity	Liters	Available
a	1.0	5	1.5	True
b	1.4	10	0.3	False
c	5.0	8	1	True

In [1]:

```
df['Available'] = pd.Series([True, False, True],
                            index=['a', 'b', 'c'])
```

- If the DataFrame already has a column with the specified name, then this is **replaced**

- **Add new column** to DataFrame
  - It is also possible to assign directly a **list**

Index	Price	Quantity	Liters
a	**0.0**	5	**0.0**
b	1.4	10	0.3
c	**0.0**	8	**0.0**

→

Index	Price	Quantity	Liters	Available
a	1.0	5	1.5	True
b	1.4	10	0.3	False
c	5.0	8	1	True

In [1]:
```
df['Available'] = [True, False, True]
```

Pandas DataFrame

- **Drop column(s)**
  - Returns a **copy** of the updated DataFrame
    - Unless `inplace=True`, in which case the original DataFrame is modified
      - This applies to many pandas methods -- always check the documentation!

Index	Price	Quantity	Liters	Available
a	1.0	~~5~~	~~1.5~~	True
b	1.4	~~10~~	~~0.3~~	False
c	5.0	~~8~~	~~1~~	True

In [1]:
```
df = df.drop(columns=['Quantity', 'Liters'])
```

Rename column(s)

- Use a **dictionary** which maps old names with new names
- Returns a **copy** of the updated DataFrame

Index	Price	Quantity	Liters	Available
a	1.0	5	1.5	True
b	1.4	10	0.3	False
c	5.0	8	1	True

→

Index	Price	**nItems**	**[L]**	Available
a	1.0	5	1.5	True
b	1.4	10	0.3	False
c	5.0	8	1	True

In [1]:

```
df = df.rename(columns={'Quantity': 'nItems',
                        'Liters': '[L]'})
```

Computation with Pandas

- Unary operations on Series and DataFrames
  - exponentiation, logarithms, …
- Operations between Series and DataFrames
  - Operations are performed **element-wise**, being aware of their **indices/columns**
- Aggregations (min, max, std, …)

- Unary operations on Series and DataFrames
  - They work with any **Numpy** ufunc
  - The operation is applied to each element of the Series/DataFrame
- Examples:
  - res = my_series/4 + 1
  - res = np.abs(my_series)
  - res = np.exp(my_dataframe)
  - res = np.sin(my_series/4)
  - ...

- Operations between Series (+,-,*,/)
  - Applied element-wise after **aligning indices**
    - Index elements which do not match are set to **NaN** (Not a Number)
  - Example:
    - res = my_series1 + my_series2

After index alignment index in the result is **sorted**

Index	
b	3
a	1
c	**10**

my_series1

Index	
a	1
b	3
d	**30**

my_series2

Index	
a	2
b	6
c	**NaN**
d	**NaN**

res

- Operations between DataFrames
  - Applied element-wise after **aligning indices** and **columns**
  - Example (align **index**):
    - res = my_dataframe1 + my_dataframe2

Index in the result is **sorted**

Index	Total	Quantity
b	3	4
a	1	2
c	**10**	**20**

my_dataframe1

Index	Total	Quantity
a	1	2
b	3	4
d	**30**	**40**

my_dataframe2

Index	Total	Quantity
a	2	4
b	6	8
c	**NaN**	**NaN**
d	**NaN**	**NaN**

res

- Operations between DataFrames
  - Example (align **columns**)
    - res = my_dataframe1 + my_dataframe2

Columns in the result are **sorted**

Index	Total	Quantity
a	1	**2**
b	3	**4**
c	5	**6**

my_dataframe1

Index	Total	Price
a	1	**2**
b	3	**4**
c	5	**6**

my_dataframe2

Index	Price	Quantity	Total
a	**NaN**	**NaN**	2
b	**NaN**	**NaN**	6
c	**NaN**	**NaN**	10

res

- Operations between DataFrames and Series
  - The operation is applied between the Series and each **row** of the DataFrame
    - Follows **broadcasting** rules
  - Example:
    - res = my_dataframe1 + my_series1

Index	Total	Quantity
a	1	2
b	3	4
c	5	6

my_dataframe1

Index	
Total	1
Quantity	2

my_series1

Index	Total	Quantity
a	2	4
b	4	6
c	6	8

res

- Pandas Series and DataFrames allow performing aggregations
  - mean, std, min, max, sum
- Examples

In [1]:
```
my_series.mean()   # Return the mean of Series elements
```

  - For DataFrames, aggregate functions are applied **column-wise** and return a Series

In [1]:
```
my_df.mean()       # Return a Series
```

- Example of **aggregations** with DataFrames: z-score normalization

In [1]:
```
mean_series = df.mean()
std_series = df.std()
df_norm = (df-mean_series)/std_series
```

Index	Total	Quantity
a	1	2
b	3	4
c	5	6

my_dataframe1

Index	
Total	3.0
Quantity	4.0

mean_series

Index	
Total	2.0
Quantity	2.0

std_series

Missing values

- Represented with **sentinel** value
  - **None**: Python null value
  - **np.nan**: Numpy Not A Number
- None is a Python **object**:
  - np.array([4, None, 5]) has dtype=Object
- np.NaN is a **Floating point** number
  - np.array([4, np.nan, 5]) has dtype=Float
- Using **nan** achieves better **performances** when performing numerical computations

Missing values

- Pandas supports both **None** and **NaN**, and automatically converts between them when appropriate
- Example:

In [1]:
```
pd.Series([4, None, 5, np.nan])
```

Out[1]:
```
0     4.0
1     NaN
2     5.0
3     NaN
dtype=float64
```

Missing values

- Operating on missing values (for Series and DataFrames)
  - isnull()
    - Return a boolean mask indicating null values
  - notnull()
    - Return a boolean mask indicating not null values
  - dropna()
    - Return filtered data containing null values
  - fillna()
    - Return new data with filled or input missing values

Missing values

- Operating on missing values: **isnull, notnull**
  - Return a new Series/DataFrame with the same shape as the input

In [1]:
```
s1 = pd.Series([4, None, 5, np.nan])
s1.isnull()
```

Out[1]:
```
0     False
1     True
2     False
3     True
dtype=bool
```

- Operating on missing values: **dropna**
  - For Series it removes null elements

In [1]:
```
s1 = pd.Series([4, None, 5, np.nan])
s1.dropna()
```

Out[1]:
```
0    4.0
2    5.0
dtype=float64
```

- Operating on missing values: **dropna**
  - For DataFrames it removes **rows** that contain at least a missing value (default behaviour)
    - Passing `how=all` removes rows if they contain all NaN's

Index	Total	Quantity
a	1	2
b	3	NaN
c	5	6

→

Index	Total	Quantity
a	1	2
c	5	6

  - Alternatively, it is possible to remove columns

```
dropped_df = df.dropna(axis='columns')
```

Missing values

- Operating on missing values: **fillna**
  - Fill null fields with a specified value (for both Series and DataFrames)

In [1]:
```
s1 = pd.Series([4, None, 5, np.nan])
s1.fillna(0)
```

Out[1]:
```
0    4.0
1    0.0
2    5.0
3    0.0
dtype=float64
```

Missing values

- Operating on missing values: **fillna**
  - The parameter **method** allows specifying different filling techniques
    - **ffill**: propagate last valid observation forward
    - **bfill**: use next valid observation to fill gap

In [1]:

```
s1 = pd.Series([4, None, 5, np.nan])
s1.fillna(method='ffill')
```

Out[1]:

```
0    4.0
1    4.0
2    5.0
3    5.0
```

Notebook Examples

- **3-Pandas Examples.ipynb**
  - **1. Accessing DataFrames and Series**

Combining Pandas objects

- Pandas provides 2 methods for combining Series and DataFrames
  - concat()
    - Concatenate a sequence of Series/DataFrames
  - append()
    - Append a Series/DataFrame to the specified object

- Concatenating 2 Series
  - Index is preserved, even if **duplicated**
    - There is nothing that prevents duplicate indices in pandas!

In [1]:
```
s1 = pd.Series(['a', 'b'], index=[1,2])
s2 = pd.Series(['c', 'd'], index=[1,2])
pd.concat((s1, s2))
```

Out[1]:
```
1    a
2    b
1    c
2    d
dtype=object
```

Combining Pandas objects

- Concatenating 2 Series
  - To avoid duplicates use **ignore_index**

In [1]:
```
s1 = pd.Series(['a', 'b'], index=[1,2])
s2 = pd.Series(['c', 'd'], index=[1,2])
pd.concat((s1, s2), ignore_index=True)
```

Out[1]:
```
0     a
1     b
2     c
3     d
dtype=object
```

- Concatenating 2 DataFrames
  - Concatenate **vertically** by default

In [1]:
```
pd.concat((df1, df2))
```

Index	Total	Quantity
a	1	2
b	3	4

Index	Total	Quantity
c	5	6
d	7	8

Index	Total	Quantity
a	1	2
b	3	4
c	5	6
d	7	8

- Concatenating 2 DataFrames
  - Missing columns are filled with NaN

In [1]:
```
pd.concat((df1, df2))
```

Index	Total	Quantity
a	1	2
b	3	4

Index	Total	Quantity	Liters
c	5	6	1
d	7	8	2

→

Index	Total	Quantity	Liters
a	1	2	**NaN**
b	3	4	**NaN**
c	5	6	1.0
d	7	8	2.0

- The **append**() method is a shortcut for concatenating DataFrames
  - Returns the result of the concatenation

In [1]:
```
df_concat = df1.append(df2)
```

is equivalent to:

In [1]:
```
df_concat = pd.concat((df1, df2))
```

- Joining DataFrames with relational algebra: **merge**()
  - Merge on:
    - The column(s) with same name in the two DFs, by default
    - Specific columns, by specifying `on=columns`
      - `left_on` and `right_on` may also be used
    - The indices, if `left_index/right_index` are True
      - This preserves the indices (discarded otherwise)
  - Depending on the DataFrames, a **one-to-one**, **many-to-one** or **many-to-many** join can be performed
    - `validate='1:1'|'1:m'|'m:1'|'m:m'` to enforce the specific merge

In [1]:
```
joined_df = pd.merge(df1, df2)
```

Examples (1)

`pd.merge(df1, df2)` ➔ merge on columns in common, ["k1"]

Index	**k1**	c2
i1	0	a
i2	1	b

Index	**k1**	c3
i1	1	b1
i2	0	a1

→

Index	**k1**	c2	c3
0	0	a	a1
1	1	b	b1

`pd.merge(df1, df2, right_index=True, left_index=True)` ➔ merge on index

Index	k1	c2
i1	0	a
i2	1	b
i3	0	c
i4	1	d

Index	k1	c3
i1	1	b1
i2	0	a1

→

Index	k1_x	c2	k1_y	c3
i1	0	a	1	b1
i2	1	b	0	a1

Examples (2)

`pd.merge(df1, df2)` ➔ performs a one-to-one merge

Index	k1	c2
i1	0	a
i2	1	b

Index	k1	c3
i1	1	b1
i2	0	a1

→

Index	k1	c2	c3
0	0	a	a1
1	1	b	b1

`pd.merge(df1, df2)` ➔ performs a many-to-one merge

Index	k1	c2
i1	0	a
i2	1	b
i3	0	c
i4	1	d

Index	k1	c3
i1	1	b1
i2	0	a1

→

Index	k1	c2	c3
0	0	a	a1
1	0	c	a1
2	1	b	b1
3	1	d	b1

Grouping data

- Pandas provides the equivalent of the SQL group by statement
- It allows the following operations:
  - **Iterating** on groups
  - **Aggregating** the values of each group (mean, min, max, …)
  - **Filtering** groups according to a condition

- **Applying** group by
  - Specify the column(s) where you want to group (**key**)
  - Obtain a DataFrameGroupBy object

```
df = pd.DataFrame({'k' : ['a','b','a','b'],
                   'c1': [2,10,3,15], 'c2' : [4,20,5,30]})
grouped_df = df.groupby('k')   # 2 groups: 'a' and 'b'
```

Index	**k**	c1	c2
0	a	2	4
1	b	10	20
2	a	3	5
3	b	15	30

Index	**k**	c1	c2
0	a	2	4
2	a	3	5
1	**b**	**10**	**20**
3	**b**	**15**	**30**

Grouping data

- **Iterating** on groups
  - Each group is a subset of the original DataFrame

In [1]:

```
for key, group_df in grouped_df:
    print(key)
    print(group_df)
```

Out[1]:

```
a
   k1  c1  c2
0  a    2   4
2  a    3   5
b
   k1  c1  c2
1  b   10  20
3  b   15  30
```

Index	k1	c1	c2
0	a	2	4
2	a	3	5

Index	k1	c1	c2
1	b	10	20
3	b	15	30

- **Aggregating** by group (min, max, mean, std)
  - The output is a DataFrame with the result of the aggregation for each group

In [1]:

```
grouped_df.mean()  # Mean, separately for each group
```

Out[1]:

```
k     c1    c2
a    2.5   4.5
b   12.5  25.0
```

Index	k1	c1	c2
0	a	2	4
2	a	3	5

Index	k1	c1	c2
1	b	10	20
3	b	15	30

The index of the result is the key of each group

- **Aggregating** a single column by group
  - The output is a Series with the result of the aggregation for each group

In [1]:
```
grouped_df['c1'].mean()
```

Out[1]:
```
k
a     2.5
b    12.5
Name: c1, dtype=float64
```

Index	k1	c1	c2
0	a	**2**	4
2	a	**3**	5

Index	k1	c1	c2
1	b	**10**	20
3	b	**15**	30

Grouping data

- **Filtering** data by group
  - The filter is expressed with a lambda function working with each group DataFrame (x)

In [1]:

```
# Keep groups for which column c1 has a mean > 5
grouped_df.filter(lambda x: x['c1'].mean()>5)
```

Out[1]:

	k	c1	c2
1	b	10	20
3	b	15	30

Index	k1	c1	c2
0	a	**2**	4
2	a	**3**	5

mean = 2.5
x: filtered out

Index	k1	c1	c2
1	b	**10**	20
3	b	**15**	30

mean = 12.5
x: kept in the result

Pivoting

- Pivoting allows inspecting relationships within a dataset
- Suppose to have the following dataset:

```
df = pd.DataFrame({'type':['a','b','b','a','b','a','b','a'],
                   'class':[3,2,3,3,2,1,1,2],
                   'fail':[1,1,1,0,1,0,0,0]})
```

Index	type	class	fail
0	a	3	1
1	b	2	1
2	b	3	1
3	a	3	0
4	b	2	1
5	a	1	0
6	b	1	0
7	a	2	0

- that shows **failures** for sensors of a given type and class during some test

In [1]:
```
df.pivot_table('fail', index='type',
               columns='class', aggfunc='sum')
```

- Shows the number of **failures** for all the combinations of **type** and **class**

Out[1]:
```
class   1  2  3
type
a       0  0  1
b       0  2  1
```

2 sensors of type b and class 2 had some failure

Index	type	class	fail
0	a	3	1
1	**b**	**2**	**1**
2	b	3	1
3	a	3	0
4	**b**	**2**	**1**
5	a	1	0
6	b	1	0
7	a	2	0

Pivoting

In [1]:
```
df.pivot_table('fail', index='type',
               columns='class', aggfunc='mean')
```

- Shows the percentage of **failures** for all the combinations of **type** and **class**

Out[1]:
```
class    1    2    3
type
a      0.0  0.0  0.5
b      0.0  1.0  1.0
```

50% of sensors of type a and class 3 had some failure

Index	type	class	fail
0	**a**	**3**	**1**
1	b	2	1
2	b	3	1
3	**a**	**3**	**0**
4	b	2	1
5	a	1	0
6	b	1	0
7	a	2	0

Multi-Index

- **Multi-Index** allows specifying an index hierarchy for
  - Series
  - DataFrames
- Example: index a Series by city and year

index	city	Rome	Rome	Turin	Turin
	year	2018	2019	2018	2019
	values	10	13	7	9

- Building a **multi-indexed Series**

In [1]:

```
ix = [['Rome', 'Rome', 'Turin', 'Turin'],
      ['2018', '2019', '2018', '2019']]
s1 = pd.Series([10,13,7,9], index=ix)
s1 = s1.sort_index()  # Multi-Index must be sorted
                      # if you want to use slicing
print(s1)
```

Out[1]:

```
Rome    2018    10
        2019    13
Turin   2018     7
        2019     9
```

Multi-Index

- **Naming** index levels

In [1]:
```
s1.index.names=['city', 'year']
print(s1)
```

Out[1]:
```
city   year
Rome   2018    10
       2019    13
Turin  2018     7
       2019     9
```

- **Accessing index levels**
  - **Slicing** and **simple indexing** are allowed
  - Slicing on index levels follows Numpy rules

In [1]:
```
print(s1.loc['Rome'])        # Outer index level
print(s1.loc[:,'2018'])     # All cities, only 2018
```

Out[1]:
```
year
2018     10
2019     13

city
Rome      10
Turin      7
```

Rome	Rome	Turin	Turin
2018	2019	2018	2019
10	13	7	9

Accessing index levels (Examples)

In [1]:

```
print(s1.loc['Turin', '2018':'2019'])
print(s1[s1>10])     # Masking
```

Out[1]:

```
city   year
Turin  2018    7
       2019    9

city  year
Rome  2019    13
```

Rome	Rome	Turin	Turin
2018	2019	2018	2019
10	13	7	9

- **Multi-indexed DataFrame**
  - Specify a multi-index for **rows**
  - **Columns** can be multi-indexed as well

		Humidity		Temperature	
		max	min	max	min
Turin	2018	33	48	6	33
	2019	35	45	5	35
Rome	2018	40	59	2	33
	2019	41	57	3	34

Multi-indexed DataFrame: creation

In [1]:

```
ix = [['Rome', 'Rome', 'Turin', 'Turin'],
      ['2018', '2019', '2018', '2019']]
cols = [['c1','c1','c2','c2'],['a','b','a','b']]
data = np.arange(16).reshape((4,4))
df = pd.DataFrame(data, index=ix, columns=cols)
print(df)
```

Out[1]:

```
            c1      c2
            a   b   a   b
Rome  2018  0   1   2   3
      2019  4   5   6   7
Turin 2018  8   9  10  11
      2019 12  13  14  15
```

Multi-Index

- **Multi-indexed DataFrame:** access with **outer** index level

In [1]:

```
print(df['c1'])               # Access by column
print(df.loc['Rome', 'c1']) # Access rows and cols
```

Out[1]:

```
              a   b
Rome  2018    0   1
      2019    4   5
Turin 2018    8   9
      2019   12  13

      a  b
2018  0  1
2019  4  5
```

		c1		c2	
		a	b	a	b
Rome	2018	0	1	2	3
	2019	4	5	6	7
Turin	2018	8	9	10	11
	2019	12	13	14	15

- **Multi-indexed DataFrame:** access with **outer** and **inner** index levels

In [1]:
```
df['c1', 'a']                # Access by column
```

Out[1]:
```
Rome   2018    0
       2019    4
Turin  2018    8
       2019   12
```

		c1		c2	
		a	b	a	b
Rome	2018	0	1	2	3
	2019	4	5	6	7
Turin	2018	8	9	10	11
	2019	12	13	14	15

- **Multi-indexed DataFrame:** access with **outer** and **inner** index levels

In [1]:
```
ix = pd.IndexSlice
df.loc[ix['Rome', '2018'], ix['c1':'c2', 'a']]
```

Out[1]:
```
c1  a    0
c2  a    2
```

		c1		c2	
		a	b	a	b
Rome	2018	0	1	2	3
	2019	4	5	6	7
Turin	2018	8	9	10	11
	2019	12	13	14	15

Multi-Index

- **Reset Index:** transform index to DataFrame columns and create new (single level) index

In [1]:
```
df.index.names = ['city', 'year']
df_reset = df.reset_index()
print(df_reset)
```

Out[1]:

	city	year	c1		c2	
			a	b	a	b
0	Rome	2018	0	1	2	3
1	Rome	2019	4	5	6	7
2	Turin	2018	8	9	10	11
3	Turin	2019	12	13	14	15

New index

Multi-Index

- **Set Index:** transform columns to Multi-Index
  - Inverse function of reset_index()

In [1]:
```
df_reset.set_index(['city', 'year'])
```

	city	year	c1		c2	
			a	b	a	b
0	Rome	2018	0	1	2	3
1	Rome	2019	4	5	6	7
2	Turin	2018	8	9	10	11
3	Turin	2019	12	13	14	15

→

city	**year**	c1		c2	
		a	b	a	b
Rome	**2018**	0	1	2	3
	2019	4	5	6	7
Turin	**2018**	8	9	10	11
	2019	12	13	14	15

New index

- **Unstack:** transform multi-indexed Series to a Dataframe

```
myseries.unstack()
```

city	year	
Rome	2018	0
	2019	4
Turin	2018	8
	2019	12

→

	2018	2019
Rome	0	4
Turin	8	12

- **Stack:** inverse function of unstack()
  - From DataFrame to multi-indexed Series

```
mydataframe.stack()
```

	2018	2019
Rome	0	4
Turin	8	12

→

Rome	2018	0
	2019	4
Turin	2018	8
	2019	12

PoliTo DBMG

Aggregates on multi-indices

- Allowed by passing the **level** parameter
- Level specifies the **row granularity** at which the result is computed

```
my_dataframe.max(level='city')
```

<table>
<tr><th rowspan="2">city</th><th rowspan="2">year</th><th colspan="2">c1</th><th colspan="2">c2</th></tr>
<tr><th>a</th><th>b</th><th>a</th><th>b</th></tr>
<tr><td rowspan="2">Rome</td><td>2018</td><td>0</td><td>1</td><td>2</td><td>3</td></tr>
<tr><td>2019</td><td>4</td><td>5</td><td>6</td><td>7</td></tr>
<tr><td rowspan="2">Turin</td><td>2018</td><td>8</td><td>9</td><td>10</td><td>11</td></tr>
<tr><td>2019</td><td>12</td><td>13</td><td>14</td><td>15</td></tr>
</table>

→

<table>
<tr><th rowspan="2">city</th><th colspan="2">c1</th><th colspan="2">c2</th></tr>
<tr><th>a</th><th>b</th><th>a</th><th>b</th></tr>
<tr><td>Rome</td><td>4</td><td>5</td><td>6</td><td>7</td></tr>
<tr><td>Turin</td><td>12</td><td>13</td><td>14</td><td>15</td></tr>
</table>

PoliTo DBMG

- **Aggregates on multi-indices**

```
my_dataframe.max(level='year')
```

city	year	c1		c2	
		a	b	a	b
Rome	2018	0	1	2	3
	2019	4	5	6	7
Turin	2018	8	9	10	11
	2019	12	13	14	15

→

year	c1		c2	
	a	b	a	b
2018	8	9	10	11
2019	12	13	14	15

Aggregates on multi-indices

- Can also aggregate columns
  - Specify axis=1

```
my_dataframe.max(axis=1, level=0)
```

<table>
<tr><th rowspan="2">city</th><th rowspan="2">year</th><th colspan="2">c1</th><th colspan="2">c2</th></tr>
<tr><th>a</th><th>b</th><th>a</th><th>b</th></tr>
<tr><td rowspan="2">Rome</td><td>2018</td><td>0</td><td>1</td><td>2</td><td>3</td></tr>
<tr><td>2019</td><td>4</td><td>5</td><td>6</td><td>7</td></tr>
<tr><td rowspan="2">Turin</td><td>2018</td><td>8</td><td>9</td><td>10</td><td>11</td></tr>
<tr><td>2019</td><td>12</td><td>13</td><td>14</td><td>15</td></tr>
</table>

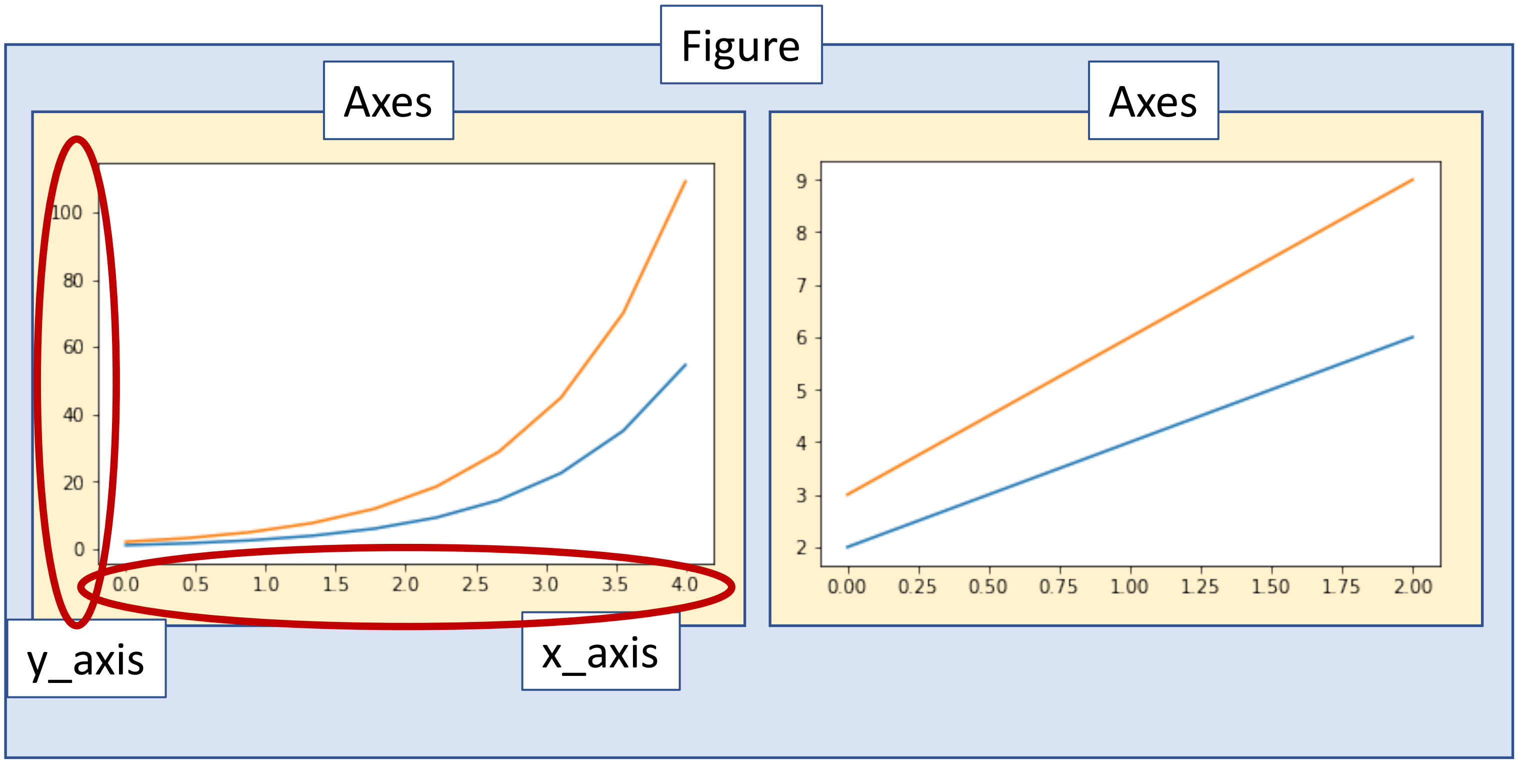
→

city	year	c1	c2
Rome	2018	1	3
Rome	2019	5	7
Turin	2018	9	11
Turin	2019	13	15

- **Two of the most commonly used graphical libraries are:**
  - **Matplotlib**
    - We present here only a very **short introduction** as the library is fairly large and visualization is not the focus of this course
  - **Seaborn** (data visualization library based on Matplotlib)
    - **Not covered by this course**

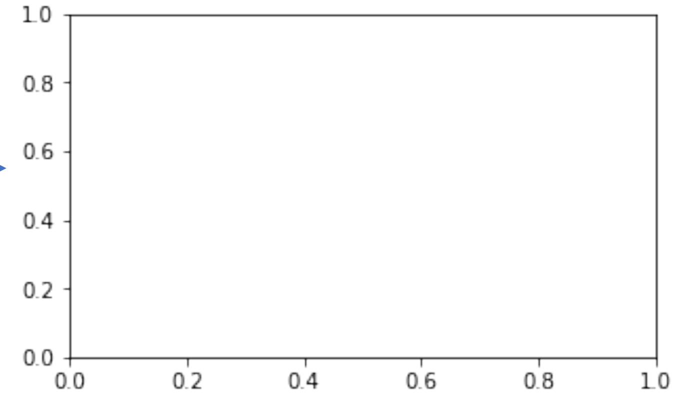
- **Matplotlib**
  - Set of methods that make matplotlib work like **matlab**
  - It has 2 **interfaces**:
    - **Matlab style plotting (Stateful)** ☹
      - Plotting methods are called from the **pyplot** package
      - They all work on the **current** Figure and Axes
    - **Object oriented (Stateless)** ☺
      - Plot functions are called as **methods** of a specific Figure and Axes
      - This allows modifying **many objects at a time** (the system does not keep a "current object" state)

- **Figures and Axes**



Creation of a new figure:

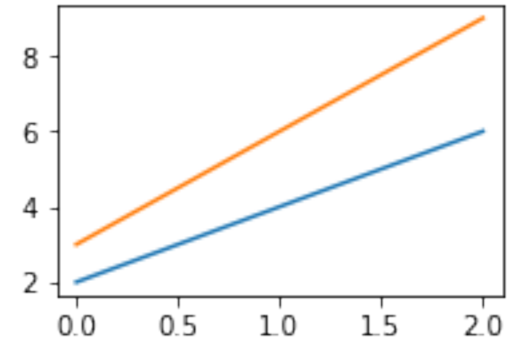
```python
import matplotlib.pyplot as plt
fig, ax = plt.subplots(figsize=(5, 3))
plt.show()
```



- Subplots returns a new **Figure** and its **Axes** object
- **figsize** specifies the figure size (width, height) in inches
- By default ax is a single Axes object (1 Figure with a single Axes)

Drawing a line plot (single Axes object)

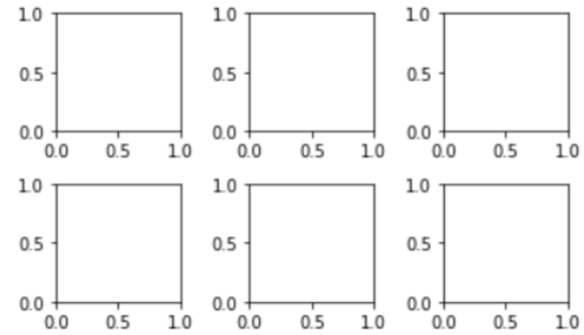
```
fig, ax = plt.subplots(figsize=(3, 2))
ax.plot([0,1,2],[2,4,6])
ax.plot([0,1,2],[3,6,9])
plt.show()
```



- The plot method of a specific Axes takes as input two lists (or NumPy arrays): **x, y** coordinates of the points
- The default style draws **segments** passing through the specified coordinates
- Subsequent calls of plot add new line to the same Axes

Creation of a new figure:

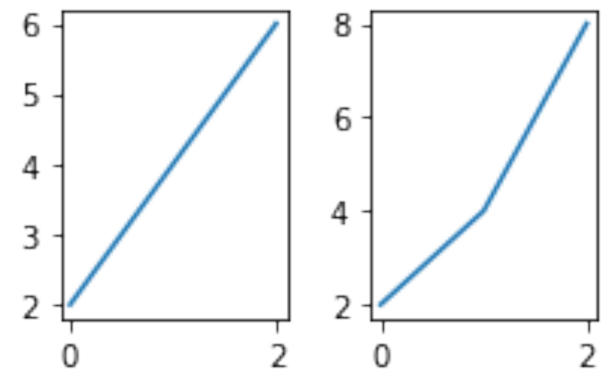
```
fig, ax = plt.subplots(2, 3, figsize=(5, 3))
plt.tight_layout()
plt.show()
```



- The first two parameters of subplots specify to create a figure with **2 rows, 3 columns** (6 Axes objects)
- **tight_layout()** is necessary at the end to let the subplots fit the frame size without blank spaces at the borders

Drawing a line plot (multiple Axes object)

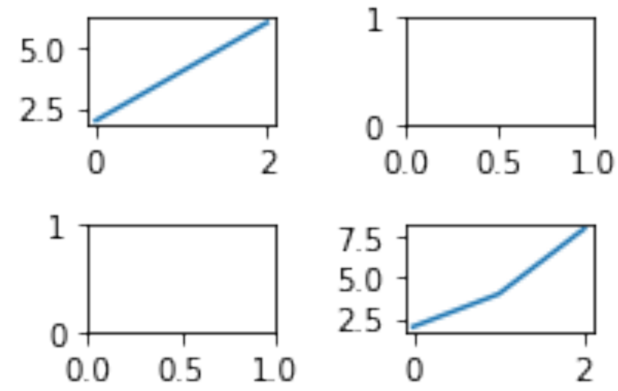
```
fig, ax = plt.subplots(1, 2,
                       figsize=(3, 2))
ax[0].plot([0,1,2],[2,4,6])
ax[1].plot([0,1,2],[3,6,9])
plt.tight_layout()
plt.show()
```



- The ax object is a **Numpy array** with the created Axes objects
- It has **shape = (n,)** if the figure has 1 row and n columns

Drawing a line plot (multiple Axes object)

```
fig, ax = plt.subplots(2, 2,
                       figsize=(3, 2))
ax[0, 0].plot([0,1,2],[2,4,6])
ax[1, 1].plot([0,1,2],[3,6,9])
plt.tight_layout()
plt.show()
```



- It has **shape = (m, n)** if the figure has m rows and n columns

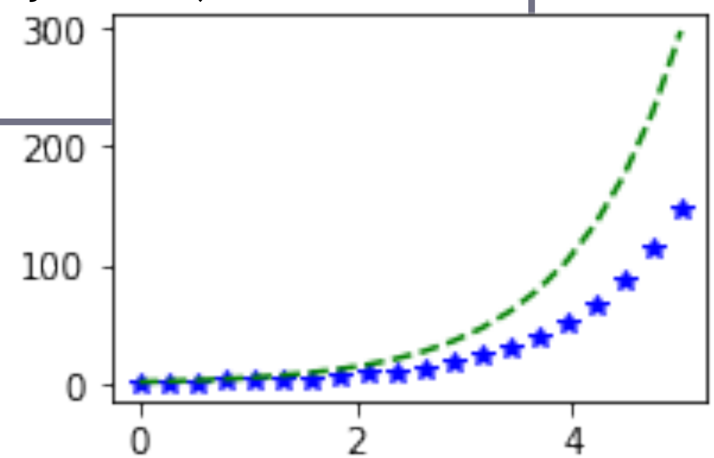
Plot types

- With Matplotlib you can design different plot types
- **The most common are:**
  - Line plot
  - Scatter plot
  - Bar chart

Line plot

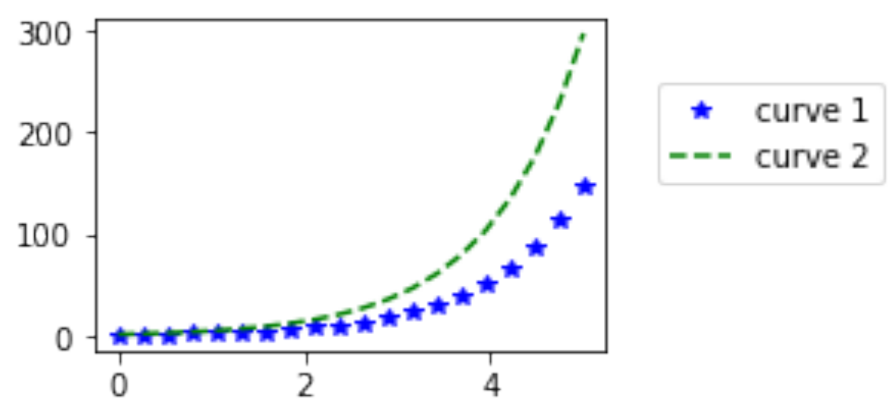
- Allows displaying a sequence of points/segments that **share the same properties**
  - E.g. same size, color, width, ...

```
x = np.linspace(0, 5, 20)
y = np.exp(x)
fig, ax = plt.subplots(figsize=(3, 2))
ax.plot(x, y, c='blue', linestyle='', marker='*')
ax.plot(x, 2*y, c='green', linestyle='--')
plt.show()
```



- Different plots can be associated to **labels** to be displayed in a **legend**

```
x = np.linspace(0, 5, 20)
y = np.exp(x)
fig, ax = plt.subplots(figsize=(3, 2))
ax.plot(x, y, c='blue', linestyle='', marker='*', label='curve 1')
ax.plot(x, 2*y, c='green', linestyle='--', label='curve 2')
ax.legend(loc=(1.1, 0.5))
plt.show()
```



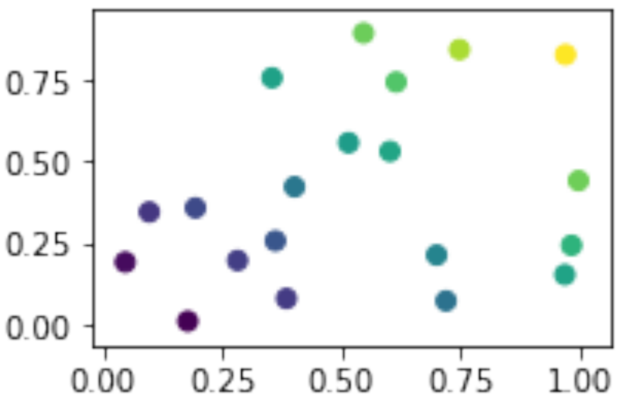
Line plot

- **linestyle** specifies the type of line
  - Examples: '-', '--' (or 'dashed'), ':' (or 'dotted')
- **marker** specifies the type of points to be drawn
  - Examples: 'o', '*', '+', '^'
- **c** specifies the color to be applied to markers and segments
  - Examples: 'red', 'orange', 'grey'
  - Examples: '#0F0F6B' (RGB)
  - Examples: (0.5, 1, 0.8, 0.8) (RGBA tuple)

Scatter plot

- Allows displaying a set of points and assign them custom properties
  - E.g. different color, size

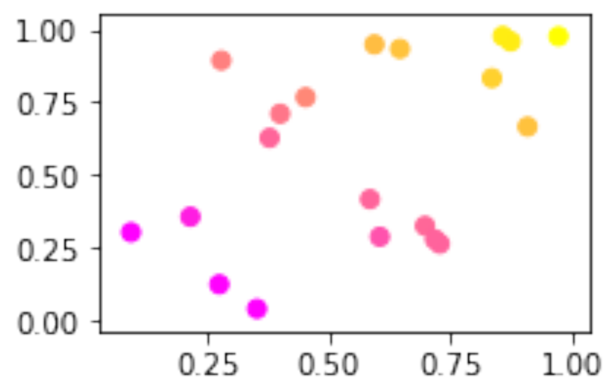
```
x = np.random.rand(20)
y = np.random.rand(20)
colors = x + y      # color as a function of x and y
fig, ax = plt.subplots(figsize=(3, 2))
ax.scatter(x, y, c=colors)
plt.show()
```



Scatter plot

- **c=colors** associate a number (float or integer) to each point
  - In the same sequence as they appear in x, y)
  - These numbers are used to select a color from a specific **colormap**
    - https://matplotlib.org/users/colormaps.html

```
colors = x + y      # color as a function of x and y
fig, ax = plt.subplots(figsize=(3, 2))
ax.scatter(x, y, c=colors, cmap='spring')
plt.show()
```

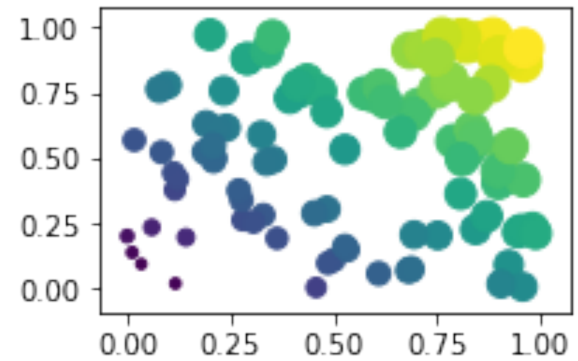


- **c=colors** associate a number (float or integer) to each point
  - Matplotlib considers the range of values of c to fit the whole range of colors of a colormap
  - c = [101, 120, 50, 60] -> range is 50-120

Scatter plot

- The size of each point can be set with the parameter **s**
- Size is the area in **dpi** (dots per inch)

```
x = np.random.rand(20)
y = np.random.rand(20)
colors = x + y     # color as a function of x and y
area = 100*(x+y)   # size as a function of x, y
fig, ax = plt.subplots(figsize=(3, 2))
ax.scatter(x, y, c=colors, s=area)
plt.show()
```

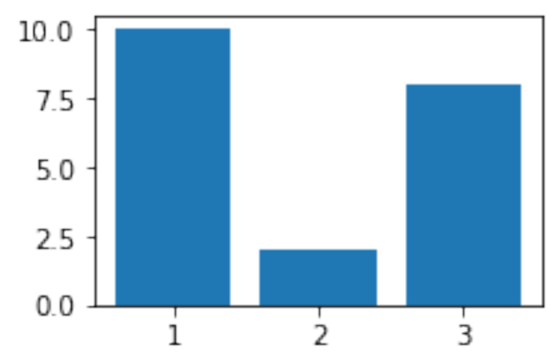


Bar chart

- Allows displaying a sequence of numbers as vertical or horizontal bars

```
height = [10, 2, 8]
x = [1, 2, 3]      # position of the bars, x axis

fig, ax = plt.subplots(figsize=(3, 2))
ax.bar(x, height)
plt.show()
```

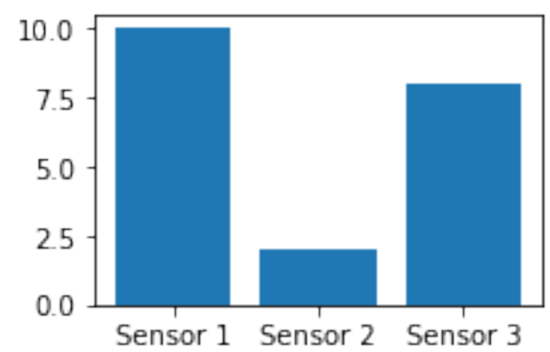


Bar chart

- Ticks on the horizontal axis can be **labeled** with some text

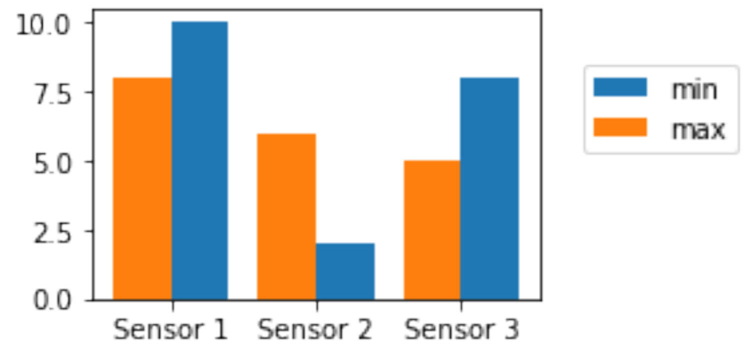
```
height = [10, 2, 8]
x = [1, 2, 3]      # position of the bars, x axis
labels = ['Sensor 1', 'Sensor 2', 'Sensor 3']

fig, ax = plt.subplots(figsize=(3, 2))
ax.bar(x, height, tick_label=labels)
plt.show()
```



Bar chart

- Bars can be grouped

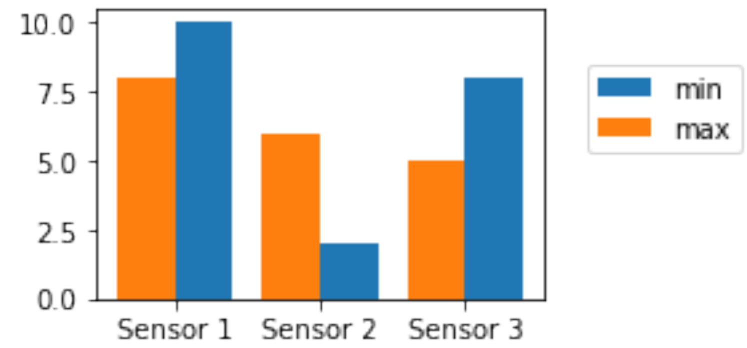



```
height_min = [10, 2, 8]
height_max = [8, 6, 5]
x = np.arange(3)
width = 0.4
labels = ['Sensor 1', 'Sensor 2', 'Sensor 3']

fig, ax = plt.subplots(figsize=(3, 2))
ax.bar(x+width/2, height_min, width=width, label='min')
ax.bar(x-width/2, height_max, width=width, label='max')
ax.set_xticks(x)              # setup positions of x ticks
ax.set_xticklabels(labels)    # set up labels of x ticks
ax.legend(loc=(1.1, 0.5))     # x, y position, in percentage
plt.show()
```

Bar chart

- Bars can be grouped




```
height_min = [10, 2, 8]
height_max = [8, 6, 5]
x = np.arange(3)
width = 0.4
labels = ['Sensor 1', 'Sensor 2', 'Sensor 3']

fig, ax = plt.subplots(figsize=(3, 2))
ax.bar(x+width/2, height_min, width=width, lab
ax.bar(x-width/2, height_max, width=width, lab
ax.set_xticks(x)              # setup positions
ax.set_xticklabels(labels)    # set up labels o
ax.legend(loc=(1.1, 0.5))     # x, y position,
plt.show()
```

However, other libraries might make our life easier!

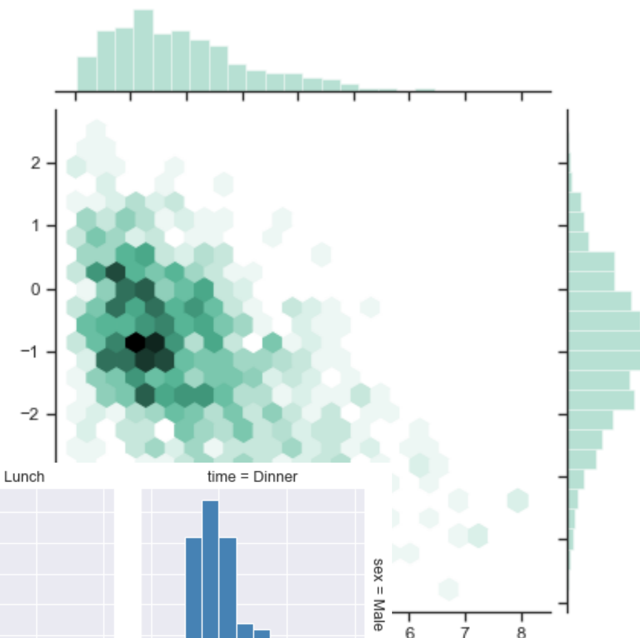
```
df = pd.DataFrame({
    "min": height_min,
    "max": height_max
  },
  index=labels
)
df.plot.bar()
```

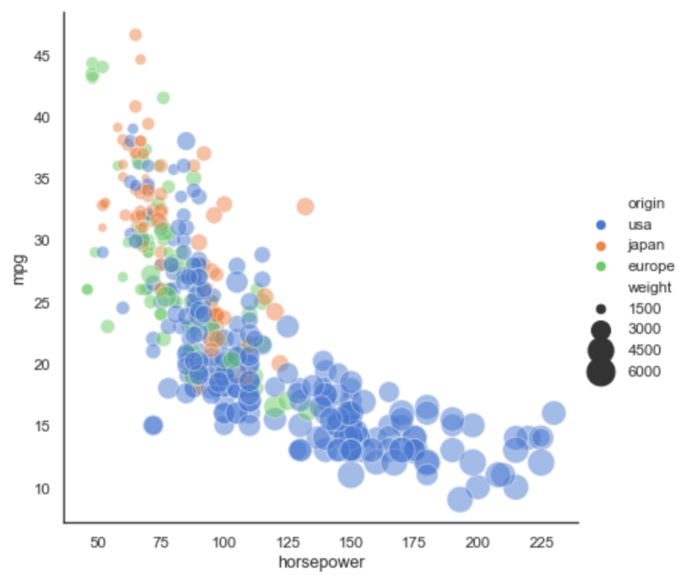
Writing images to file

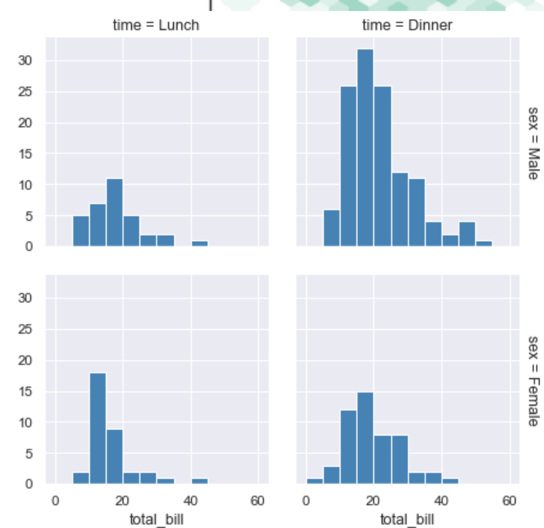
- Generated figures can be **saved** to file with different formats

```
fig, ax = plt.subplots(figsize=(3, 2))
ax.plot([0,1,2],[2,4,6])
ax.plot([0,1,2],[3,6,9])
fig.savefig(“./out/test.png") # or ‘.jpg’, ‘.eps’, ‘.pdf’
```

- Based on Matplotlib
  - High level **interface** for drawing complex chart with attractive visual impact







References

- **Matplotlib website:**
  - https://matplotlib.org/
- **Seaborn website:**
  - https://seaborn.pydata.org/