# Data Science and Machine Learning for Engineering Applications

Lecture Notes 2: Numpy

March 15, 2023 - Politecnico di Torino

## 1 Numpy Introduction

*Numpy* [1] stands for *Numerical Python* and is a library that allows you to work with multidimensional arrays. It is designed to **store** and **operate** with **dense** numerical data **efficiently**. The arrays are **dense** because they are not represented with a sparse representation. All the dimensions of your multidimensional arrays are filled with important data. It is optimized to work with dense matrices and not with sparse matrices). For example, there are no zeros inside dense arrays. The library provides many built-in options to **access the data** arrays efficiently and to **perform math and logic operations**. Most of the Machine Learning libraries are based internally on Numpy. You can find a good guide here.

## 2 Numpy arrays

The main object the Numpy library provides is the **array**. Arrays represent the concept of **tensors**. A tensor is a generic vector with **n dimensions**. Tensors' elements have all the **same type**. This is the main difference with respect to lists. Numpy arrays can represent multidimensional arrays such as vectors (1-dim arrays), matrices (2-dim arrays), or tensors (n-dim arrays).

### 2.1 Numpy arrays vs lists

You can define multidimensional arrays with nested Python lists. For example, you can define a list of lists to create a 2-dimensional array (matrix). Or a list of lists of lists to create a 3-dimensional array (tensor). Python and Numpy are **row-based**. If you define a one-dimensional vector, it is a row vector.

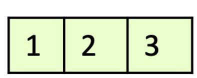
The following code creates a vector (1-dim array) with Python lists:

```python
my_matrix_from_list = [1, 2, 3]
print(my_matrix_from_list)
```

`Output:`

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

This list represents a row vector:



The following code creates a matrix (2-dim array) with Python lists:

```python
my_matrix_from_list = [[1, 2, 3], [4, 5, 6]]
print(my_matrix_from_list)
```

Output:

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

| | | |
|---|---|---|
| 1 | 2 | 3 |
| 4 | 5 | 6 |

However, since lists can contain **heterogeneous data types**, they keep **overhead** information. For each item, it should keep the reference to them. Moreover, each item is an object with some metadata (i.e., the header), such as the **type** and the **identifier**. Instead, Numpy contains only **fixed-type** data. Therefore, it doesn't need the overhead to specify each item type (the type information is stored only once and is the same for all the items). It also stores data in **contiguous** memory addresses, allowing **faster indexing**. In conclusion, Numpy provides you with two main advantages:

- Higher **flexibility** of indexing methods and operands.
- Higher **efficiency** of operations

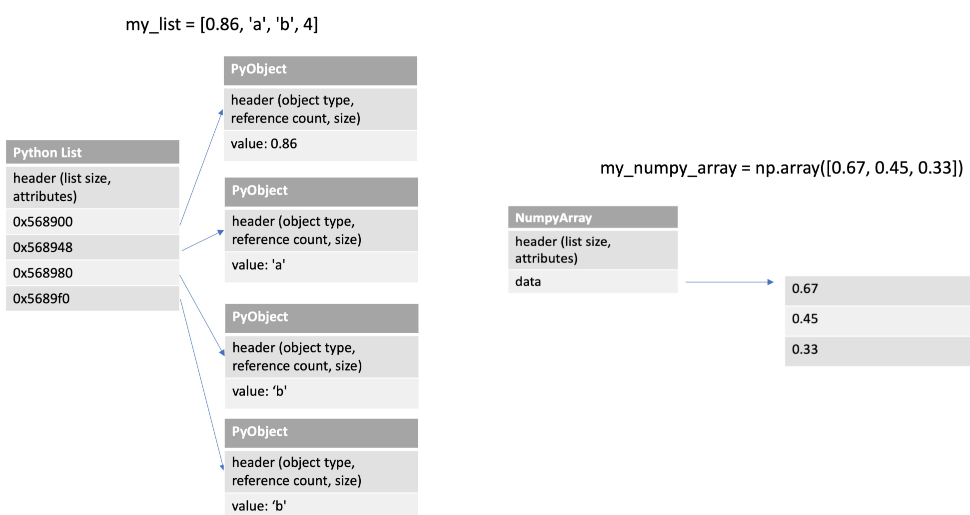
You can create Numpy arrays directly from lists. To create the same matrix with Numpy:

```python
import numpy as np

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

Output:

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

The following figure shows how data is stored in lists vs Numpy arrays:




## 2.2 Numpy data types

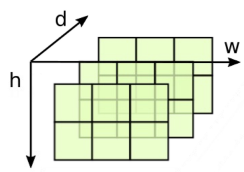
Numpy defines its own data types:

- Numerical types: int8, int16, int32, int64, uint8, uint16, uint32, uint64, float16, float32, float64
- Boolean values: bool

The intX types are integers with different memory sizes (e.g., int64 occupies more memory space than int8 but can contain larger integer numbers). The uintX types are unsigned integers (i.e., without positive and negative signs).

## 2.3 Multidimensional arrays

A multidimensional array is a collection of elements organized along an arbitrary **number of dimensions**. The following figure shows an example of a 3-dimensional array.



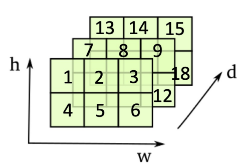

If you want to create a 3-dimensional array directly from Python lists:

```
import numpy as np

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

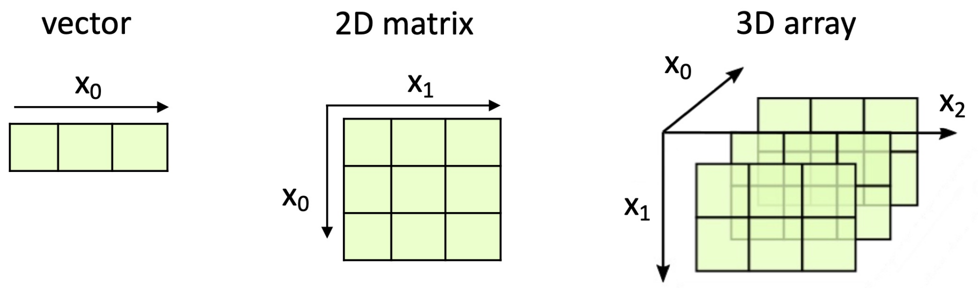
Output:

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

[[ 7 8 9]
[10 11 12]]

[[13 14 15]
[16 17 18]]]
```

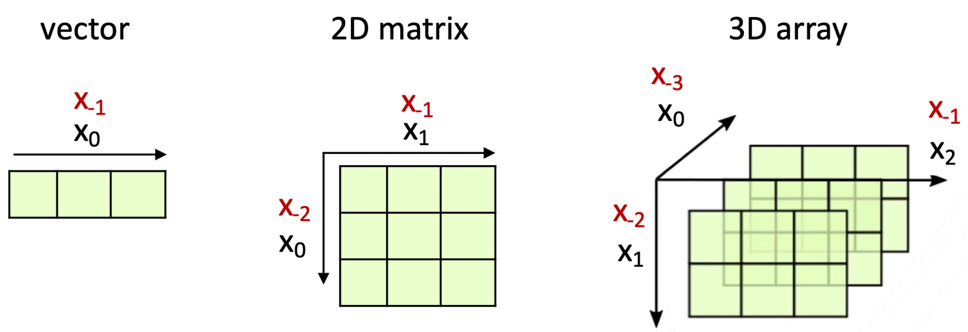
The array will represent the following 3-dimensional tensor:



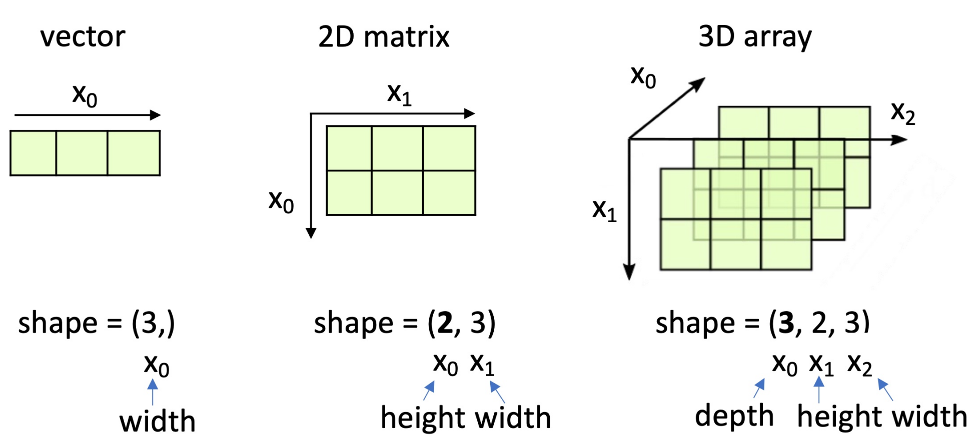

**Numpy arrays** are characterized by a set of **axes** and a **shape**. The **axes** define the number of dimensions of an array. Given an n-dimensional array, axes range from 0 to $n-1$. For example, for a row vector, the axis is only 0 ($x_0$); for a matrix, the axes are 0 and 1 ($x_0$, $x_1$); for a 3-dim tensor, the axes are 0, 1, 2 ($x_0$, $x_1$, $x_2$).




For the row vector, $x_o$ is the only dimension, and it represents the horizontal axis. However, if we define a matrix, the horizontal axis is no more $x_o$, but it became $x_1$. **This is because every time a dimension is added, the newly added dimension takes the name $x_0$, and all the other previous dimensions are shifted (or incremented) by one**. For example, the horizontal axis is $x_0$ for a 1-dim array, $x_1$ for a 2-dim array, and $x_2$ for a 3-dim array. You can also number the axis with a negative notation. This can be useful because axis -1 always refers to the **row axis** (i.e., axis -1 refers to the axis with the highest positive index of the array that is always the row dimension/horizontal axis).




The **shape** is a **tuple** that specifies the **number of elements along each axis** of a Numpy array (i.e., how many elements have each axis). When you add a dimension, it is added on the **left** of the shape tuple.




## 2.4 Column vectors vs Row vectors

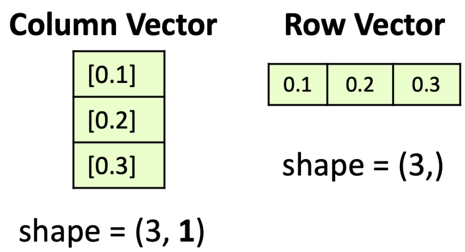
Arrays with 1-dimension are always a **row vector**. If you want to create a **column vector**, you should define a 2D matrix with 3 rows and 1 column.

```
import numpy as np
```

```python

row_array = np.array([0.1, 0.2, 0.3]) # Define a row vector
col_array = np.array([[0.1], [0.2], [0.3]]) # Define a column vector
print(f"Row vector: shape {row_array.shape}")
print(row_array)
print(f"\nColumn vector: shape {col_array.shape}")
print(col_array)
```

Output:

```
Row vector:  shape (3,)
[0.1 0.2 0.3]

Column vector:  shape (3, 1)
[[0.1]
[0.2]
[0.3]]
```




## 2.5 Create Numpy arrays

### 2.5.1 Creation from list

As shown before, you can directly create an array from a list with `np.array(my_list, dtype=np.float16)`. You can also specify the data type in the construct with the `dtype` parameter. If not specified, the data type will be automatically inferred.

```python
import numpy as np
my_arr = np.array([[1, 1], [2, 2]], dtype=np.float32)
print(my_arr)
```

Output:

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

## 2.6 Creation from scratch

You can also create an array with a given **shape** filled with all 0 `np.zeros(shape)`, all 1 `np.ones(shape)`, or with a specific value `np.full(shape, value)`. The **shape** is a **tuple**.

```python
import numpy as np
my_arr_0 = np.zeros((3, 2)) # 3 rows and 2 columns filled with 0
my_arr_1 = np.ones((3, 2)) # 3 rows and 2 columns filled with 1
my_arr_full = np.full((3, 2), 0.5) # 3 rows and 2 columns filled with 0.5
print("Array with zeros:")
print(my_arr_0)
print("\nArray with ones:")
print(my_arr_1)
print("\nArray with full:")
print(my_arr_full)
```

Output:

```
Array with zeros:
[[0.  0.]
[0.  0.]
[0.  0.]]

Array with ones:
[[1.  1.]
[1.  1.]
[1.  1.]]

Array with full:
[[0.5 0.5]
[0.5 0.5]
[0.5 0.5]]
```

You can also create arrays with more complex functions:

- `np.linespace(start, stop, n)`: generates `n` samples from `start` to `stop` (both included). The generated samples are 1-dimensional (i.e., a row vector).
- `np.arange(start, stop, step)`: generates numbers from `start` (included) to `stop` (excluded) with a `step` (optional). The generated samples are 1-dimensional (i.e., a row vector). It is similar to the `range()` function.
- `np.random.normal(mean, std, shape)`: generates **random data** with a **normal distribution** with a given `mean`, `standard deviation`, and `shape`. The **dimensions** of the array depend on the shape.
- `np.random.random(shape)`: generates **random data** with a **uniform distribution in [0,1]** with a given `shape`. The **dimensions** of the array depend on the shape.

```
import numpy as np
arr1 = np.linspace(0, 1, 11) = np.linespace(0, 1, 11)
arr2 = np.arange(1, 11, 2)
arr3 = np.random.normal(0, 1, (3,2))
arr4 = np.random.random((3,2))
print("Linspace array:")
print(arr1)
print("\nArange array:")
print(arr2)
print("\nRandom Normal array:")
print(arr3)
print("\nRandom Uniform array:")
print(arr4)
```

Output:

```
Linespace array:
[0.  0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1.  ]

Arange array:
[1 3 5 7 9]

Random Normal array:
[[-1.08916723 0.77737725]
[-1.22526899 0.2342717 ]
[-0.50455924 1.1322722 ]]

Random Uniform array:
[[0.84256324 0.20861337]
[0.3547634 0.93538505]
[0.28628767 0.26374818]]
```

### 2.7 Attributes of Numpy arrays

There are also some attributes to inspect the properties of the arrays:

- `arr.ndim`: returns the number of dimensions of the array.
- `arr.shape`: returns the shape of the array as a tuple.
- `arr.size`: returns the size of the array (i.e., the total number of elements computed as the product of the shape values).

```python
import numpy as np
arr = np.array([[1, 2, 3], [4, 5, 6]])
print("number of dimensions:", arr.ndim)
print("shape:", arr.shape)
print("number of elements (size):", arr.size)
```

`Output:`

```
number of dimensions:  2
shape:  (2, 3)
number of elements (size):  6
```

## 3 Operations with Numpy arrays

You can perform many operations to manipulate arrays.

### 3.1 Universal functions

#### 3.1.1 Binary operations

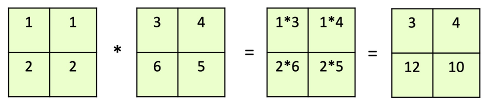
Universal functions **binary operations** are **element-wise** operations (performed element by element) with arrays (2 or more) of the **same shape**. You can perform **element-wise** addition `+`, difference `-`, multiplication `*`, etc., between each element of two arrays. **The resulting array will have the same shape as the two starting arrays**. All the arrays involved in these operations must have the **same shape**.

```python
import numpy as np

x = np.array([[1, 1], [2, 2]])
y = np.array([[3, 4], [6, 5]])
array_sum = x + y # element by element addition
array_mul = x * y # element by element multiplication
print("sum:\n", array_sum)
print("mul:\n", array_mul)
```

Output:

```
sum:
[[4 5]
[8 7]]
mul:
[[3 4]
[12 10]]
```

The following figure graphically shows how the **element-wise multiplication** is performed:




Notice that this kind of multiplication is different from matrices multiplication (i.e., rows times columns).

### 3.1.2 Unary operations

Universal functions **unary operations** are **element-wise** operations applied to each element of one array. You can perform **element-wise** absolute value `np.abs(arr)`, exponentiation `np.exp(arr)`, logarithm `np.log(arr)`, etc., **to each element of one array**. The operation is applied **separately** to each element of the array. The resulting array will have the **same shape** as the starting array. A new array will be created with the same shape (i.e., the original array will remain unchanged).

Compute the **absolute values** of the array elements:

```python
import numpy as np
x = np.array([[1, -1], [2, -2]])
array_abs = np.abs(x) # element-wise absolute value of the array
print("array_abs:\n", array_abs)
```

Output:

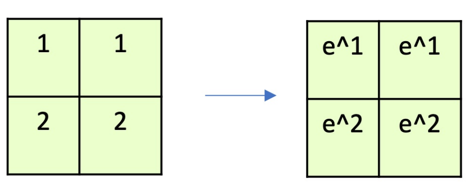
```
array_abs:
[[1 1]
[2 2]]
```

Compute the **exponentiation** of array elements:

```python
import numpy as np
x = np.array([[1, 1], [2, 2]])
array_exp = np.exp(x) # element-wise exponentiation of the array
print("array_exp:")
print(array_exp)
```

Output:

```
array_exp:
[[2.718 2.718]
[7.389 7.389]]
```



## 3.2 Aggregate functions

Aggregate functions are operations that return a **single value** from an array. You can compute the minimum `np.min(arr)` or `arr.min()`, the maximum `np.max(arr)` or `arr.max()`, the mean value `np.mean(arr)` or `mean.min()`, the standard deviation `np.std(arr)` or `arr.std()`, the sum `np.sum(arr)` or `arr.sum()`, the index of the element with the minimum value `np.argmin(arr)` or `arr.argmin()`, the index of the element with maximum value `np.argmax(arr)` or `arr.argmax()`, etc.

```python
import numpy as np
x = np.array([[1, 1], [2, 2]])
array_sum = np.sum(x) # sum of all the elements in the array
print("array_sum:", array_sum)
```

Output:

```
array_sum:  6
```

### 3.2.1 Aggregate functions along axis

You can specify the **axis** along with performing the operation. In this way, you apply the aggregate function **along a specified dimension** of your array. For example, if you have a 2-d array (i.e., a matrix), and you want to compute the sum **separately for each column**, you can specify `axis=0` in the `np.sum(arr, axis=0)` function or the `arr.sum(axis=0)` method. This will return a row vector (i.e., 1-dim array) with the sums along the columns. Notice that a **row** vector is returned either if you reduce over the columns or the rows (when you are computing aggregate functions for a 2-dimensional array, i.e., a matrix). You can perform an aggregate function along each dimension even with n-dimensional arrays (with $n > 2$). This will return an $n - 1$ dimensional array with the aggregated values.

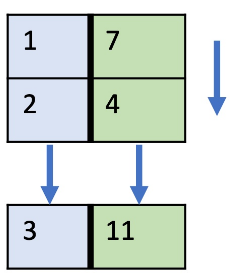
For example, if you want to perform a sum of all the elements in the array along **columns**:

```python
import numpy as np
x = np.array([[1, 7], [2, 4]])
column_sums = np.sum(x, axis=0) # sum of all the elements in the array along columns
print("column_sums:", column_sums)
```

Output:

```
column_sums:  [ 3, 11]
```

We can see that it returns a **row** vector.



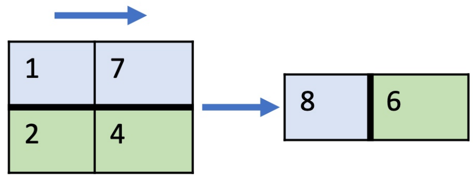

If, instead, you want to perform a sum of all the elements in the array along **rows**:

```python
import numpy as np

x = np.array([[1, 7], [2, 4]])
row_sums = np.sum(x, axis=1) # sum of all the elements in the array along rows
print("row_sums:", row_sums)
```

Output:

```
row_sums:  [ 8, 6]
```

We can see that it still returns a **row** vector.




## 3.3 Sorting functions

You can sort arrays with the `np.sort(arr)` function or the `arr.sort()` method. The `np.sort(arr)` function creates a **sorted copy** of the array `arr` (i.e., the array `arr` is not modified). In contrast, the `arr.sort()` method sorts the array `arr` **inplace** (i.e., `arr` is modified). If you don't specify an axis, by default, the array is sorted along the last axis (-1) corresponding to the row axis (i.e., the horizontal axis).

```
import numpy as np

x = np.array([[1, 9, 8], [10, 4, 2]])
sorted_x_along_rows = np.sort(x) # sort along rows
print("sorted_x_along_rows: \n", sorted_x_along_rows)
```

Output:

```
sorted_x_along_rows:
[[ 1 8 9]
[ 2 4 10]]
```

You can also specify the axis being sorted. For example, if you want to sort along columns:

```
import numpy as np
x = np.array([[1, 9, 8], [10, 4, 2]])
sorted_x_along_cols = np.sort(x, axis=0) # sort along columns (vertical axis = 0)
print("sorted_x_along_cols: \n", sorted_x_along_cols)
```

Output:

```
sorted_x_along_cols:
[[ 1 4 2]
[ 10 9 8]]
```

## 3.4 Algebraic operations

### 3.4.1 Inner product

You can compute the **inner** product of two vectors using `np.dot()`. Remember that the dot product between two vectors $\overrightarrow{x}$ and $\overrightarrow{y}$, with $n$ elements, is computed with the following formula:

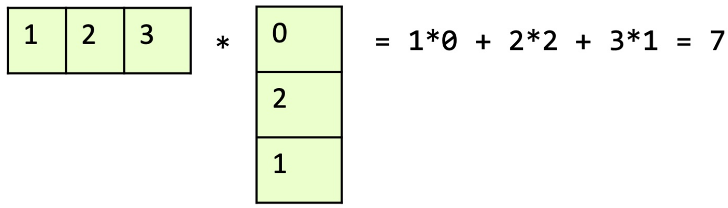
$$\overrightarrow{x} \cdot \overrightarrow{y} = \sum_{i=1}^{n} x_i * y_i = x_1 * y_1 + x_2 * y_2 + ... + x_n * y_n \tag{1}$$

Notice that `np.dot()` works even if the second vector is not a column vector (i.e., it is a row vector).

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

Output:

```
7
```

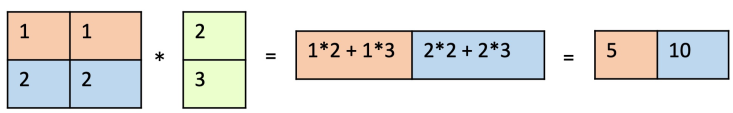



### 3.4.2 Matrices multiplication

You can also perform matrix multiplication (i.e., rows times columns) with the same function `np.dot()`.

**Matrix times vector**

When computing `np.dot(x, y)`, this time, `x` is a matrix, and `y` is a vector (it also works with a row vector). A matrix times a vector produces a new row vector with the matrix multiplication result.




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

Output:

```
[5, 10]
```

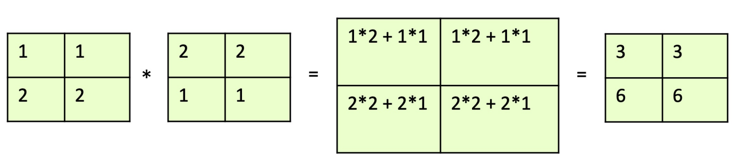
**Matrix times matrix**

When computing `np.dot(x, y)`, this time, either `x` and `y` are matrices. A matrix times a matrix will produce another matrix with the matrix multiplication result.

```python
import numpy as np
x = np.array([[1, 1], [2, 2]])
y = np.array([[2, 2], [1, 1]])
print(np.dot(x,y))
```

Output:

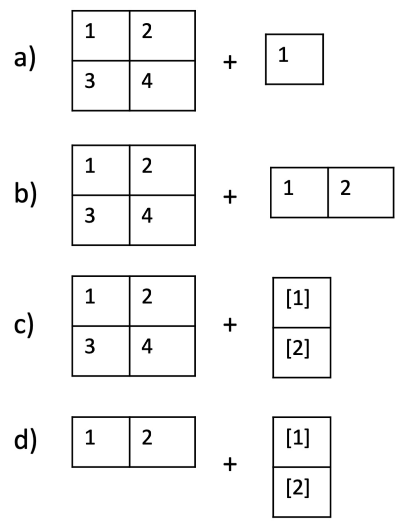
```
[[3, 3], [6, 6]]
```



# 4 Broadcasting

Broadcasting allows you to perform some operations between arrays with **different shape**. For example:

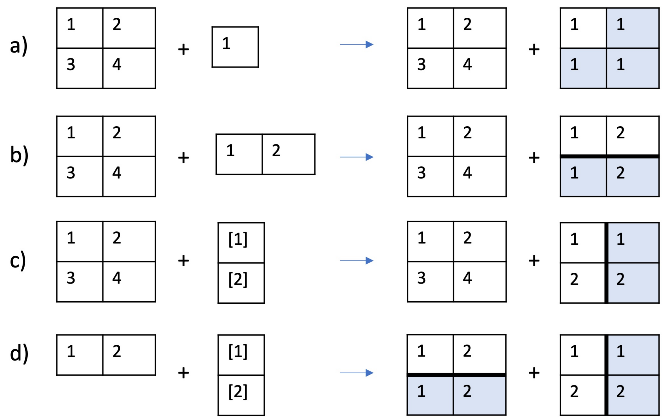
- a) Matrix `+`, `-`, `*`, `/` a scalar
- b) Matrix `+`, `-`, `*`, `/` a row vector
- c) Matrix `+`, `-`, `*`, `/` a column vector
- d) row vector `+`, `-`, `*`, `/` a column vector




If you remember, universal functions allow you to sum arrays with the same shape. However, Python **broadcasting** allows you to perform this operation even with some arrays with a different shape. The basic idea is to **replicate the shape of the smaller array to match the shape of the other array**. You can image **broadcasting** like making a copy of the smaller array's elements for matching the other array's size. However, internally, Numpy can operate without producing a copy (for efficiency reasons).

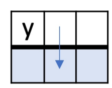
This is how broadcasting works for the examples in the previous Figure:

- a) The matrix remains the same, and the scalar is replicated to match the size of the matrix.
- b) The matrix remains the same, and the row vector is replicated (vertically) to match the size of the matrix.
- c) The matrix remains the same, and the column vector is replicated (horizontally) to match the size of the matrix.
- d) The row vector is replicated vertically, while the column vector is replicated horizontally.



Broadcasting is based on three rules:

- The shape of the array with **fewer dimensions** is **padded** with leading ones.
  E.g., `x.shape=(2, 3), y.shape=(3) -> y.shape=(1, 3)`
- 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** (copied).
  E.g., `x.shape=(2, 3), y.shape=(1, 3) -> stretch:y.shape=(2, 3)`




- If there is a dimension where both arrays have $shape > 1$, then **broadcasting cannot be performed**.
  Example where **broadcasting does not works**:
  0. `x.shape=(3, 2), y.shape=(3)`
  1. *Rule 1*: `y.shape=(3) -> y.shape=(1, 3)`
  2. *Rule 3*: shapes of `x.shape=(3, 2)` and `y.shape=(1, 3)` are **incompatibles** (i.e., both arrays have $shape > 1$ in the $x1$ dimension). This will raise an **exception** (i.e., an error).

# 5 Accessing Numpy arrays

You can access Numpy arrays in many ways:

- ***Simple indexing***: access single elements of the array (Section 5.1).
- ***Slicing***: access a slice of the array (5.2).
- ***Masking***: access portions of the array based on a boolean mask (5.3).
- ***Fancy indexing*** (Not covered)
- ***Combined indexing*** (Not covered)

The main difference is that **slicing** provides **views** of the considered array. **Views** allow to **read** and **write** data on the **original array**. If you modify some data in your **view**, the modification will also affect the original array. In contrast, **masking** and fancy indexing provide **copies** of the array. If you modify some data in your **copy**, the modification will **not** affect the original array.

## 5.1 Simple indexing

**Simple indexing** allows you to access **one single element** of the array. To do so, you should write in **square brackets** `[]` the indices along each axis of the element that you want to access, separated with commas. This example shows how to access the third element (i.e., column) in the second row (remember that indices start from 0).

```
import numpy as np
x = np.array([[1, 2, 3], [4, 5, 6]]) # define a matrix
el = x[1, 2] # x[second row, third column] -> index starts from 0
print(el)
```

Output:

```
6
```

If you want to modify the value of the second row and third column to 0, you can access that element and assign a new value:

```
import numpy as np
x = np.array([[1, 2, 3], [4, 5, 6]]) # define a matrix
x[1, 2] = 0 # modify the cell in the second row and third column
print(x[1, 2])
print(x)
```

Output:

```
0
[[1, 2, 3]
[4, 5, 0]]
```

You can also use **negative indices** (as for python lists):

```
import numpy as np
x = np.array([[1, 2, 3], [4, 5, 6]]) # define a matrix
el = x[1, -1] # x[second row, last element of the horizontal axis] -> index -1 starts
    counting from the end
print(el)
```

Output:

```
6
```

## 5.2 Slicing

**Slicing** allows access to **contiguous elements** of an array. It provides **views** on the array. **Views** allow to **read** and **write** data on the original array. In other words, if you modify the view, the modification will **affect** the original array. The syntax is similar to list slicing. **For each dimension**, you should specify, between **square brackets** `[]`, the `start` and `stop` indices, and the optional `step` as follows: `[start:stop:step, start:stop:step, ...]`. You can also omit the `start`, `stop`, and/or `step` values.

For example, if you want to get the first three rows and two columns of an array:

```
import numpy as np
x = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9], [10, 11, 12]]) # define a matrix
print("full array:")
print(x)
print("\n First three rows and two columns of the array:")
print(x[:3, :2])
```

Output:

```
full array:
[[ 1 2 3]
[ 4 5 6]
[ 7 8 9]
[10 11 12]]

First three rows and two columns of the array:
[[1 2]
[4 5]
[7 8]]
```

If you want to modify a slice of the array, you should assign a value (or an array) to the accessed slice:

```python
import numpy as np
x = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9], [10, 11, 12]]) # define a matrix
print("Original array:")
print(x)
x[:3, :2] = -1
print("\n Modified array")
print(x)
```

Output:

```
Original array:
[[ 1 2 3]
[ 4 5 6]
[ 7 8 9]
[10 11 12]]

Modified array:
[[ -1 -1 3]
[ -1 -1 6]
[ -1 -1 9]
[10 11 12]]
```

This example shows you that it is only a **view** of the original array. Therefore, if you modify the view, it also changes the original array:

```python
import numpy as np
x = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9], [10, 11, 12]]) # define a matrix
print("Original array:")
print(x)
x_view = x[:3, :2] # slice of the original array (view)
print("\n View of the array:")
print(x_view)
x_view[:,:] = -1 # assing -1 to all the values of the view
print("\n View of the array after modification:")
print(x_view)
print("\n Original array after view modification")
print(x)
```

Output:

```
Original array:
[[ 1 2 3]
[ 4 5 6]
[ 7 8 9]
[10 11 12]]

View of the array:
[[ 1 2]
[ 4 5]
[ 7 8]

View of the array after modification:
[[ -1 -1]
[ -1 -1]
[ -1 -1]

Original array after view modification:
[[ -1 -1 3]
[ -1 -1 6]
[ -1 -1 9]
[10 11 12]]
```

You can also modify only an element or a slice of the view. For example, now, you want to assign $-1$ only to the first element of the view:

```python
import numpy as np
x = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9], [10, 11, 12]]) # define a matrix
print("Original array:")
print(x)
x_view = x[:3, :2] # slice of the original array (view)
print("\n View of the array:")
print(x_view)
x_view[0,0] = -1 # assing -1 to the first element of the view
print("\n View of the array after modification:")
print(x_view)
print("\n Original array after view modification")
print(x)
```

Output:

```
Original array:
[[ 1 2 3]
[ 4 5 6]
[ 7 8 9]
[10 11 12]]

View of the array:
[[ 1 2]
[ 4 5]
[ 7 8]

View of the array after modification:
[[ -1 2]
[ 4 5]
[ 7 8]

Original array after view modification:
[[-1 2 3]
[ 4 5 6]
[ 7 8 9]
[10 11 12]]
```

If you **don't want that modification on the view also affect the original array**, you should do a **hard copy** while selecting the slice (with the `copy()` method). After the **copy** method, the modification on the slice will not affect the original array.

```python
import numpy as np
x = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9], [10, 11, 12]]) # define a matrix
print("Original array:")
print(x)
x_view = x[:3, :2].copy() # hard copy of the slice of the original array
print("\n View of the array:")
print(x_view)
x_view[:,:] = -1 # assing -1 to all the values of the view
print("\n View of the array after modification:")
print(x_view)
print("\n Original array after view modification")
print(x)
```

Output:

```
Original array:
[[ 1 2 3]
[ 4 5 6]
[ 7 8 9]
[10 11 12]]

View of the array:
[[ 1 2]
[ 4 5]
[ 7 8]

View of the array after modification:
[[ -1 -1]
[ -1 -1]
[ -1 -1]

Original array after view modification:
[[ 1 2 3]
[ 4 5 6]
[ 7 8 9]
[10 11 12]]
```

### 5.3 Masking

**Masking** allows you to use masks to select the elements of the array. The syntax is similar: you have to put between **square brackets** `[]` the mask. A **mask** is a Numpy array made of **boolean values** that should have the **same shape** as the original array. The result will **not** be of the same shape as the original vectors. But it will be a **one-dimensional vector** (a row vector) with a **copy** of all the selected elements of the original array (elements where the mask is True or 1). Unlike slicing, **masking** provides **copies** of the accessed data of the array. If you modify the accessed data, it will **not affect** the original array.

```python
import numpy as np
x = np.array([[1, -2, 3], [-4, 5, -6], [7, -8, 9], [-10, 11, -12]]) # define a matrix
print("Original array:")
print(x)
x_mask = x >= 0 # mask with True for the positive elements, False otherwise
print("\n Mask with the positive elements of the array:")
print(x_mask)
print("\n Masked elements of the array (row vector):")
print(x[x_mask])
```

`Output:`

```
Original array:
[[ 1 -2 3]
[ -4 5 -6]
[ 7 -8 9]
[-10 11 -12]]

Mask with the positive elements of the array:
[[ True False True]
[False True False]
[ True False True]
[False True False]]

Masked elements of the array (row vector):
[ 1 3 5 7 9 11]
```

This can be useful to compute some statistics based on a condition. For example, if you want to compute the mean of the positive elements:

```python
import numpy as np
x = np.array([[1, -2, 3], [-4, 5, -6], [7, -8, 9], [-10, 11, -12]]) # define a matrix
print("Original array:")
print(x)
x_mask = x >= 0 # mask with the boolean True for the positive elements, False otherwise
print("\n Mask with the positive elements of the array:")
print(x_mask)
print("\n Masked elements of the array (row vector):")
print(x[x_mask])
print("\n Mean of the positive elements:", x[x_mask].mean()) # compute the mean of positive
    elements
```

Output:

```
Original array:
[[ 1 -2 3]
[ -4 5 -6]
[ 7 -8 9]
[-10 11 -12]]

Mask with the positive elements of the array:
[[ True False True]
[False True False]
[ True False True]
[False True False]]

Masked elements of the array (row vector):
[ 1 3 5 7 9 11]

Mean of the positive elements:  6.0
```

Even if changes in the mask will not affect the original array, **you can exploit the mask to access the values of the original array you want to modify**. Therefore, you can use the mask to modify elements of the original vector based on a condition. For example, if you want to replace each negative number with the value 0:

```python
import numpy as np
x = np.array([[1, -2, 3], [-4, 5, -6], [7, -8, 9], [-10, 11, -12]]) # define a matrix
print("Original array:")
print(x)
x_mask = x < 0 # mask with True for the negative elements, False otherwise
print("\n Mask with the negative elements of the array:")
print(x_mask)
x[x_mask] = 0
print("\n Modified array):")
print(x)
```

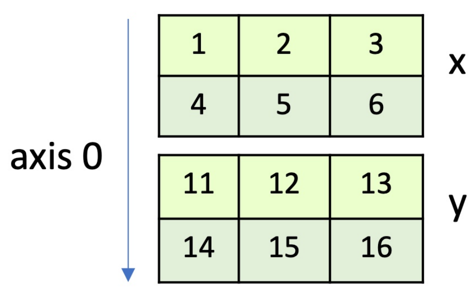
Output:

```
Original array:
[[ 1 -2 3]
[ -4 5 -6]
[ 7 -8 9]
[-10 11 -12]]

Mask with the negative elements of the array:
[[ True False True]
[False True False]
[ True False True]
[False True False]]

Modified array:
[[ 1 0 3]
[ 0 5 0]
[ 7 0 9]
[0 11 0]]
```

# 6 Working with arrays

## 6.1 Array concatenation

You can concatenate arrays along an **existing axis**. The resulting array will have the **same number of dimensions** of the input arrays. To this purpose, you can use the `np.concatenate()` function. You have to specify the arrays and the axis where you perform the concatenation. If not specified, the default axis is 0.



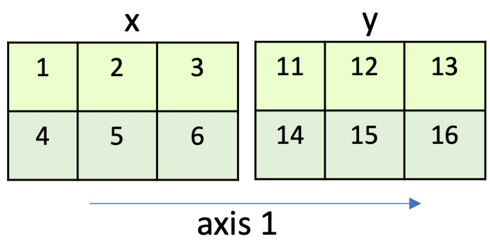

This example shows you have to concatenate two arrays on the **vertical** axis (along columns):

```python
import numpy as np
x = np.array([[1, 2, 3], [4, 5, 6]])
y = np.array([[11, 12, 13], [14, 15, 16]])
my_arr = np.concatenate((x,y))
print(my_arr)
```

`Output:`

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

If you want to concatenate two arrays on the **horizontal** axis (along rows) you have to specify $axis = 1$:

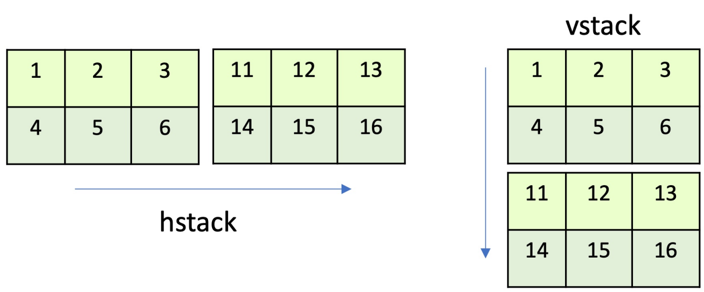



```python
import numpy as np
x = np.array([[1, 2, 3], [4, 5, 6]])
y = np.array([[11, 12, 13], [14, 15, 16]])
my_arr = np.concatenate((x,y), axis=1)
print(my_arr)
```

Output:

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

There are also two other equivalent functions for **horizontal** and **vertical** concatenations, called `np.hstack()` and `np.vstack()`, respectively. With these functions, you don't have to specify the axis.

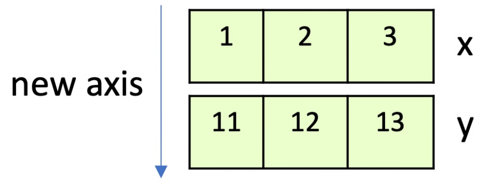



```python
import numpy as np
x = np.array([[1, 2, 3], [4, 5, 6]])
y = np.array([[11, 12, 13], [14, 15, 16]])
h = np.hstack((x, y))
v = np.vstack((x, y))
print("Horizontal concatenation:")
print(h)
print("\nVertical concatenation:")
print(v)
```

Output:

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

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

The functions `np.hstack()` and `np.vstack()` allows concatenating also **1-dimensional** vectors along **new axis**. This is not possible with `np.concatenate()`. For example, you can't vertically concatenate (along the vertical dimension) two row vectors with the `np.concatenate()` function because you don't have the vertical dimension in the row vectors.



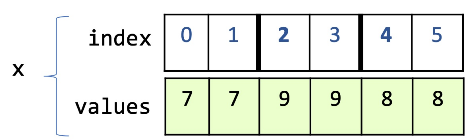

```python
import numpy as np
x = np.array([[1, 2, 3]])
y = np.array([[11, 12, 13]])
v = np.vstack((x, y))  # vertical concatenation
print("\nVertical concatenation:")
print(v)
```

Output:

```
Vertical concatenation:
[[ 1 2 3]
[11 12 13]]
```

## 6.2 Array splitting

You can split an array into a list of arrays. To this purpose, you should use the `np.split()` function. This function outputs a **list** of Numpy arrays. You have to pass the array and the list of split points as parameters. Each element of the list is a point where performing a split (e.g., [2, 4] means you want to split before element 2 and element 4).

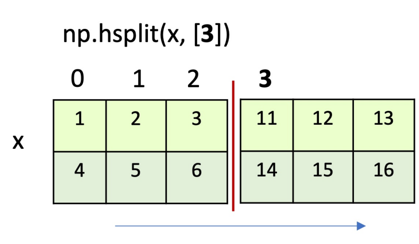


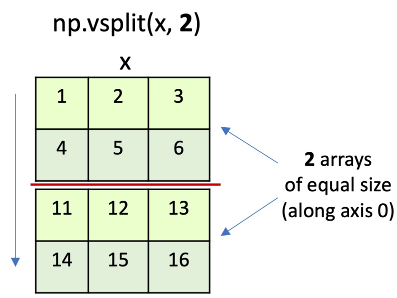

```python
import numpy as np
x = np.array([7, 7, 8, 8, 9, 9])
splitted_arrays = np.split(x, [2, 4]) # split before element 2 and 4
print(splitted_arrays)
```

Output:

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

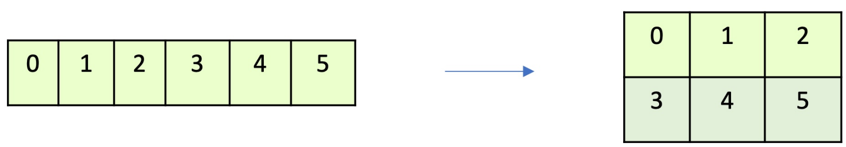
You can also perform a **horizontal** or a **vertical** split before some indices with `np.hsplit()` and `np.vsplit`, respectively.




## 6.3 Array reshaping

You can **change the shape** of a tensor with the `arr.reshape()` method. You will **keep the same elements while changing the shape** of the array. You have to specify the new shape as a tuple. For example, if you want to reshape a row vector with six elements into a matrix with two rows and three columns:




```python
import numpy as np
x = np.arange(6)
y = x.reshape((2, 3))
print(y)
```

Output:

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

The new array 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]

## 6.4 Adding new dimensions

You can also add a new dimension to an array. You can use the `np.newaxis` to add a new dimension with **shape=1** at the specified dimension position. For example, if you want to transform a row vector into a column vector (i.e., a matrix with 1 in the columns' dimension), you can do:

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

Output:

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

However, you can obtain the same result with `arr.reshape(-1,1)`

```python
import numpy as np
arr = np.array([1,2,3])
res = arr.reshape(-1,1)
print(res)
```

Output:

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

# 7 Computational efficiency

Numpy array operations are extremely **faster** and more **efficient** than operations performed with lists and explicit `for loops`. Most libraries that implement Machine Learning algorithms, such as *Scikit-Learn* (which we will see later), are based internally on Numpy to be executed efficiently and quickly. Next, we will learn another library based on Numpy for manipulating large tabular data, namely *Pandas*. When working with big data, you should use the *Numpy* or *Pandas* libraries for the data manipulation and avoid working with lists and explicit `for loops`.

The following example shows the difference in execution time to perform a `dot product` between two vectors with i) lists and explicit for loops implementation; and ii) the Numpy implementation. The Numpy implementation is two orders of magnitude faster. The efficiency gain grows if using larger arrays and more complex operations.

```python
import time
import numpy as np

l1 = [1]* 1000000 # Create a list with 1M ones
l2 = [2]* 1000009 # Create a list with 1M twos
arr1 = np.ones((1000000,)) # Create a Numpy array with 1M ones
arr2 = np.full((1000000,), 2) # Create a Numpy array with 1M twos

## Dot product with lists and explicit for loops
# get the start time
st = time.time()
dot_product = 0
for x, y in zip(l1, l2):
    dot_product += x*y
# get the end time
et = time.time()

# get the execution time
elapsed_time = et - st
print('Lists and explicit for loops:')
print(f'Dot product: {dot_product}')
print('Execution time: {:.4f} seconds'.format(elapsed_time))

## Dot product with Numpy
# get the start time
st = time.time()
dot_product = np.dot(arr1, arr2)
et = time.time()

# get the execution time
elapsed_time = et - st
print('\nNumpy implementation:')
print(f'Dot product: {dot_product}')
print('Execution time: {:.4f} seconds'.format(elapsed_time))
```

Output:

```
Lists and explicit for loops:
Dot product:  2000000
Execution time:  0.1100 seconds

Numpy implementation:
Dot product:  2000000.0
Execution time:  0.0031 seconds
```

## References

[1] Charles R. Harris et al. “Array programming with NumPy”. In: *Nature* 585.7825 (Sept. 2020), pp. 357–362. DOI: 10.1038/s41586-020-2649-2. URL: https://doi.org/10.1038/s41586-020-2649-2.