



Politecnico
di Torino



Data Science Lab

Numpy: Numerical Python

Andrea Pasini
Flavio Giobergia
Elena Baralis

DataBase and Data Mining Group



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



Python lists vs NumPy



- “Build two randomly initialized $N \times N$ matrices A and B, then add them element-wise and place the output in C”

Python lists

```
from random import random

def build_random_matrix(n):
    mat = []
    for i in range(n):
        row = []
        for j in range(n):
            row.append(random())
        mat.append(row)
    return mat

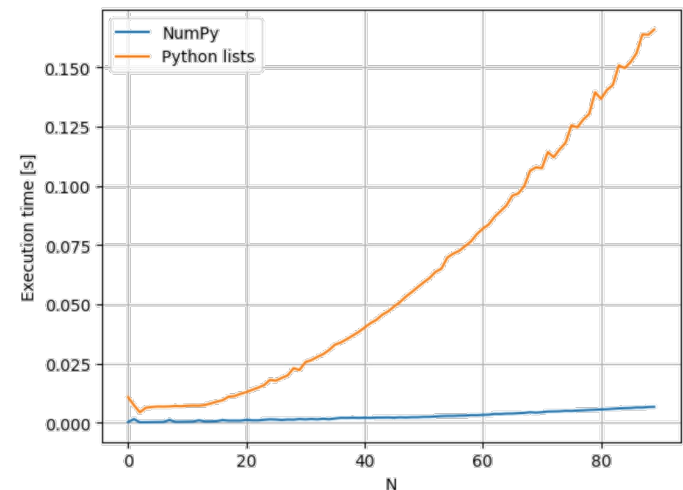
n = 100
A = build_random_matrix(n)
B = build_random_matrix(n)

C = []
for i in range(n):
    row = []
    for j in range(n):
        row.append(A[i][j] + B[i][j])
    C.append(row)
```

NumPy

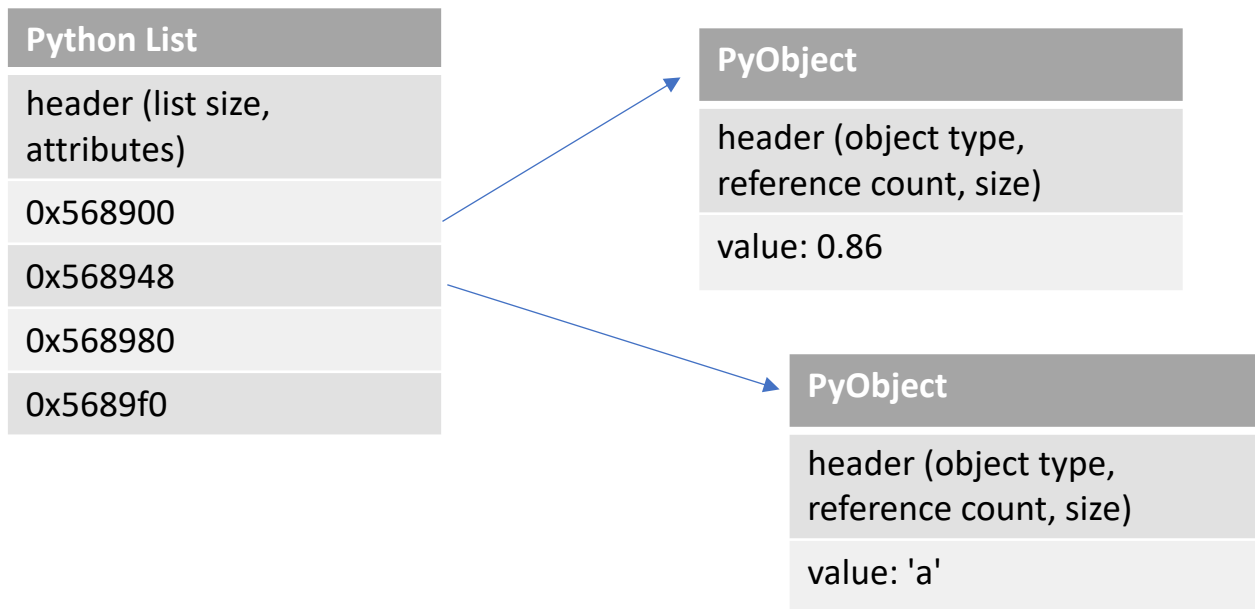
```
import numpy as np

n = 100
A = np.random.random((n, n))
B = np.random.random((n, n))
C = A + B
```



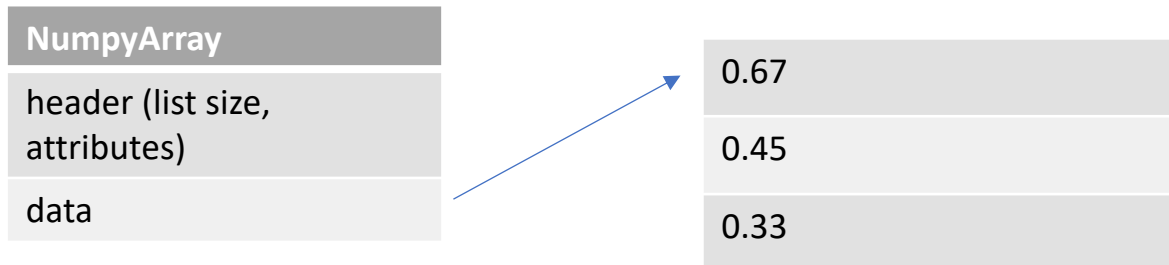


- Since lists can contain heterogeneous data types, they keep **overhead** information
 - E.g. `my_heterog_list = [0.86, 'a', 'b', 4]`





- 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])`



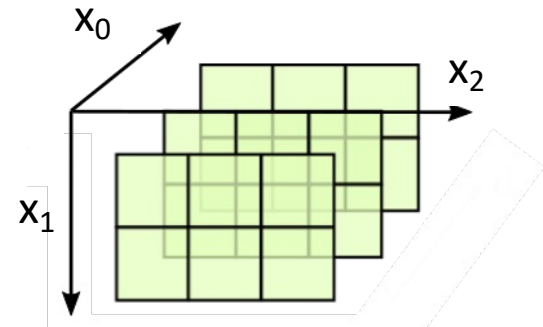


- Numpy data types
 - Numpy defines its own data types
 - Numerical types
 - int8, int16, int32, int64
 - uint8, ... , uint64
 - float16, float32, float64
 - Boolean values
 - bool



Multidimensional arrays

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





- Multidimensional arrays 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

		13	14	15	
		7	8	9	
1	2	3			18
4	5	6			12

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



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

1	2	3
---	---	---

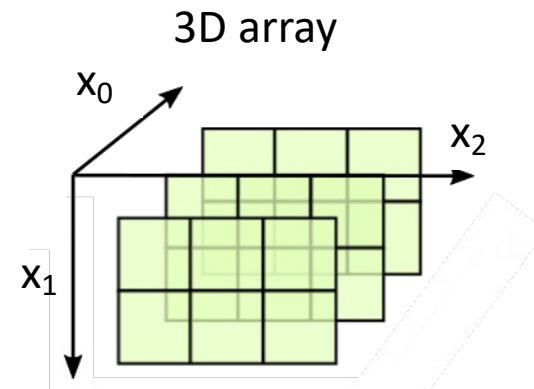
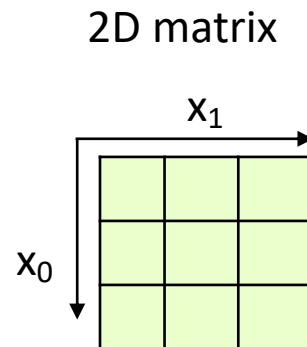
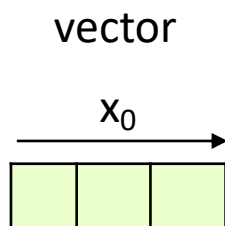
		13	14	15
		7	8	9
1	2	3		18
4	5	6		12

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

```
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]]])
```



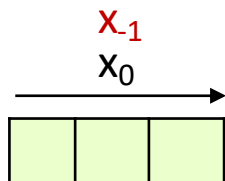
- Multidimensional arrays with **Numpy**
 - Characterized by a set of **axes** and a **shape**
 - The **axes** of an array define its dimensions
 - a (row) vector has 1 axis (1 dimension)
 - a 2D matrix has 2 axes (2 dimensions)
 - a ND array has N axes



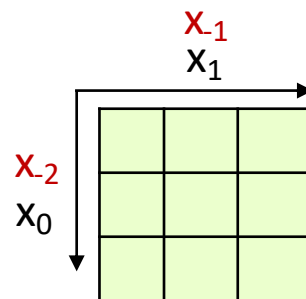


- Multidimensional arrays with **Numpy**
 - Axes can be numbered with negative values
 - Axis -1 is always along the **row**

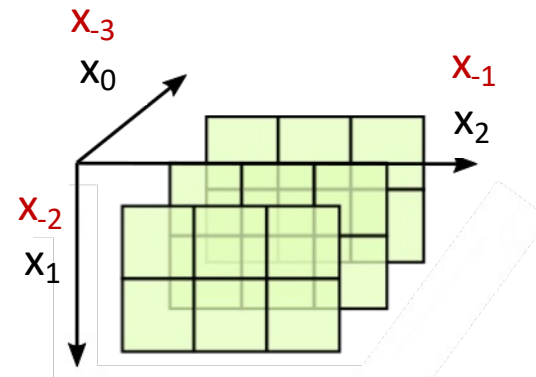
vector



2D matrix

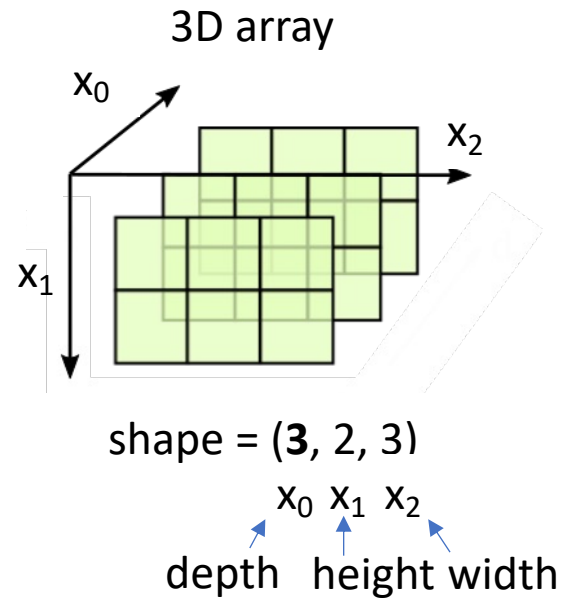
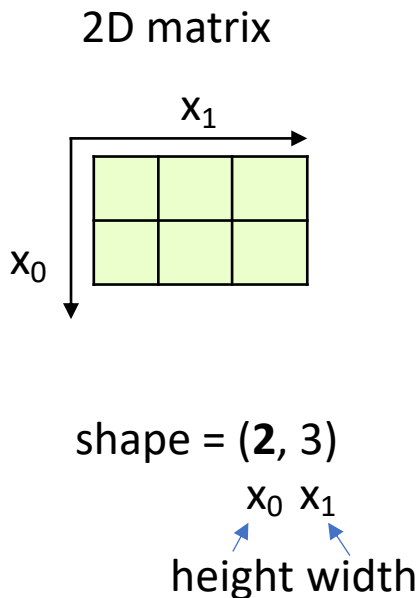
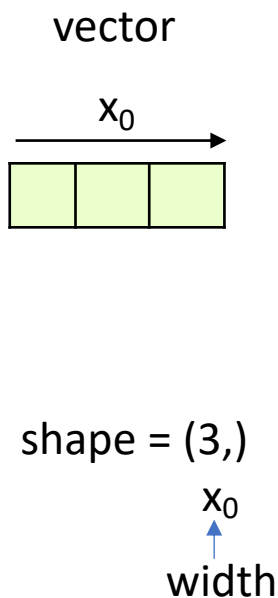


3D array





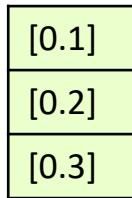
- Multidimensional arrays with **Numpy**
 - The **shape** of a Numpy array is a tuple that specifies the number of elements along each axis
 - Examples:





- Column vector vs row vector

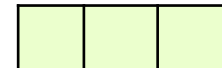
e.g. `np.array([[0.1], [0.2], [0.3]])`



shape = (3, 1)

Column vector is a 2D matrix!

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



shape = (3,)



- 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]]
```



- 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]`





- 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)



■ 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]]
```

1	1	*	3	4	=	1*3	1*4	=	3	4
2	2		6	5		2*6	2*5		12	10



- **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



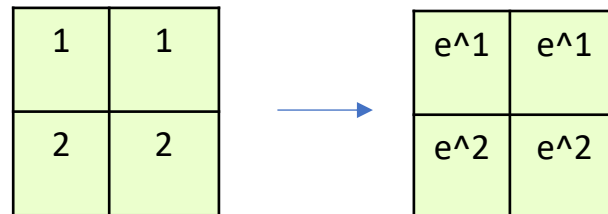
- Example:

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
```



np.argmin(), np.argmax()



- For 1-dimensional array $x \rightarrow$ position of the smallest/largest element of x

```
x = np.array([5, 3, 9, 0, 7])
x.argmax()
```

0 1 2 3 4

3

- For N-dimensional array $x \rightarrow$ position of the smallest/largest element of the *flattened* version of x
 - *Flattened* = collapsed into one dimension, `x.flatten()`

```
x = np.array([[8, 3, 9],
               [4, 2, 9]])
x.argmax()
```

x.flatten()

[8, 3, 9, 4, 2, 9]

0 1 2 3 4 5

4



Aggregate functions along axis

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

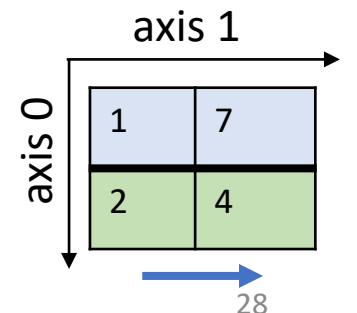
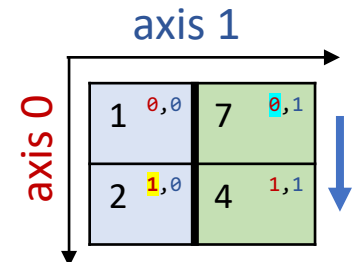
```
In [1]: x=np.array([[1,7],[2,4]])  
x.argmax(axis=0)
```

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

(index of maximum element within each column)

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

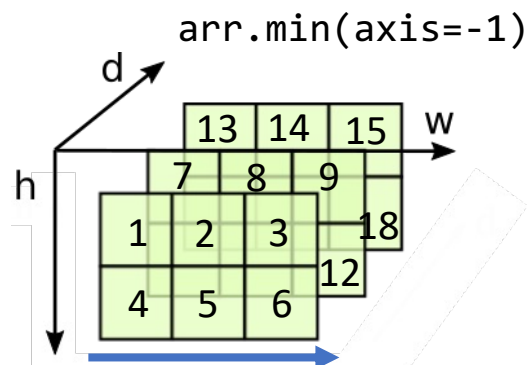
```
Out[2]: [8, 6] → (sum the elements of each row)
```





■ Aggregate functions along axis

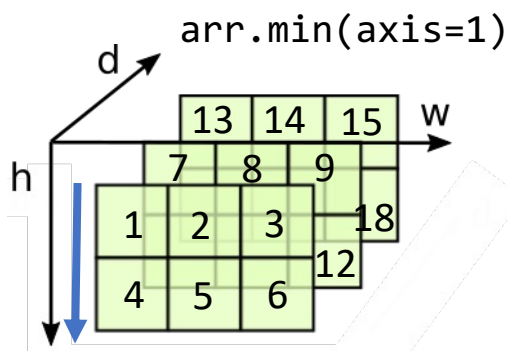
- The aggregation dimension is **removed** from the output



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

Final output

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



```
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)

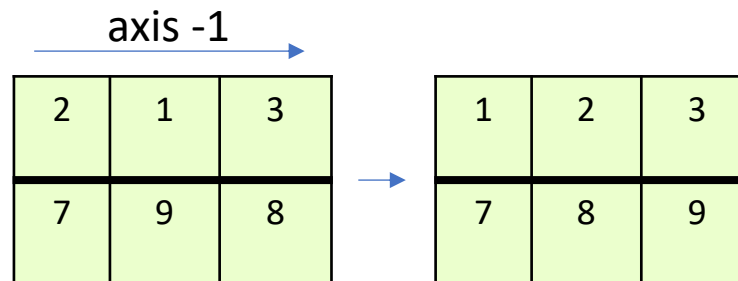


■ Sorting

- 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

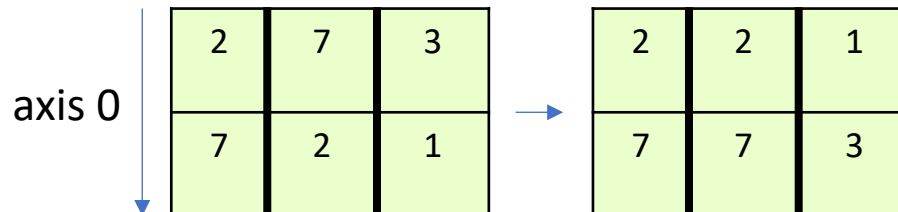
- Allows specifying the axis being sorted

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]]
```

array indices
(by row – axis -1)

array values

2	1	3
7	9	8

array indices (by row – axis -1) → 0 1 2



sorted indices

1	0	2
0	2	1



■ 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]]
```



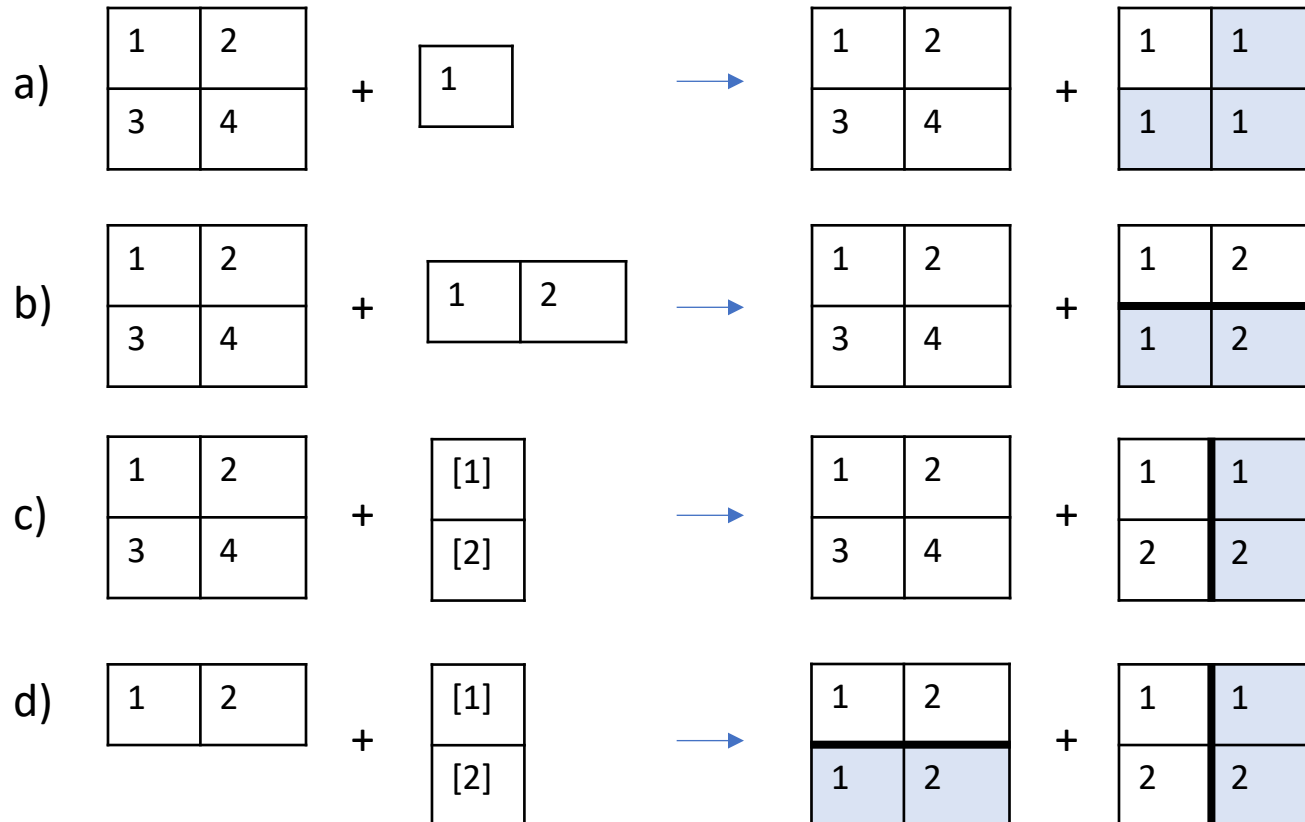
Notebook Examples

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





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



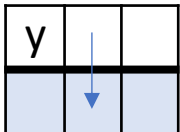


- Rules of broadcasting

- The shape of the array with **fewer dimensions** is **padded** with leading ones

$$x.shape = (2, 3), y.shape = (3) \longrightarrow y.shape = (1, 3)$$

- 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**



$$x.shape = (2, 3), y.shape = (1, 3) \longrightarrow \text{stretch: } y.shape = (2, 3)$$

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

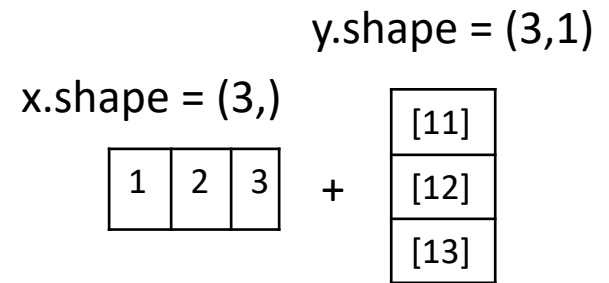


Broadcasting



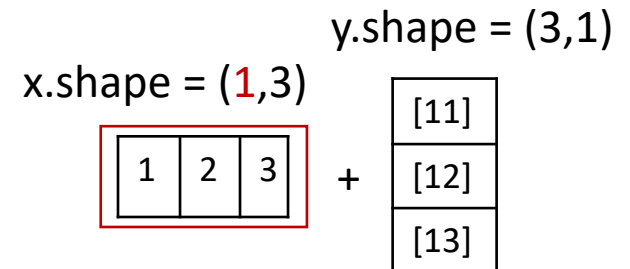
■ Example: compute $x + y$

- $x = \text{np.array}([1, 2, 3])$
- $y = \text{np.array}([[11], [12], [13]])$
- $z = x + y$



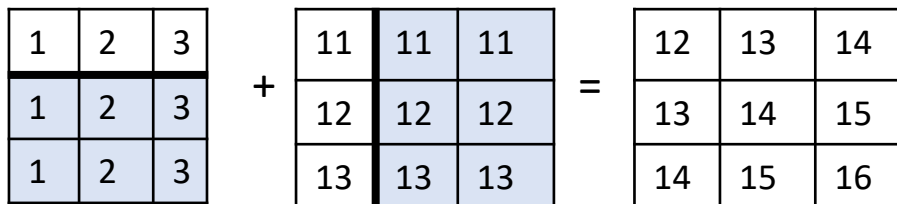
■ Apply Rule 1

- $x.\text{shape}$ becomes $(1, 3)$: $x = [[1,2,3]]$



■ Apply Rule 2:

- extend x on the vertical axis, y on the horizontal one





- Example: compute $x + y$

- $x = \text{np.array}(\text{[[1, 2],[3,4],[5,6]])}$

$x.\text{shape} = (3, 2)$

- $y = \text{np.array}([11, 12, 13])$

$y.\text{shape} = (3,)$

- $z = x + y$

- Apply Rule 1

- $y.\text{shape}$ becomes $(1, 3)$: $y = \text{[[11,12,13]]}$

11	12	13
----	----	----

- Apply Rule 3

- shapes $(3, 2)$ and $(1, 3)$ are incompatibles

- Numpy will raise an **exception**

1	2
3	4
5	6



Notebook Examples

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





- 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



- **Simple indexing:** read/write access to element

- $x[i, j, k, \dots]$

```
In [1]: x = np.array([[2, 3, 4],[5,6,7]])  
        e1 = x[1, 2]           # read value (indexing)  
        print("e1 =", e1)
```

```
Out[1]: e1 = 7
```

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

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

	0	1	2
0	2	3	4
1	5	6	7



- **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



- **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



- **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]]
```

	0	1	2
0	1	2	3
1	4	5	6
2	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]]
```

	0	1	2
0	1	2	3
1	4	5	6
2	7	8	9



■ 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]]
```




■ 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()`



- **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



■ 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), \wedge (xor), \sim (negation)
 - Example
 - $\text{mask} = \sim((x < 1) | (x > 5)) \Leftrightarrow ((x \geq 1) \& (x \leq 5))$
 - elements that are between 1 and 5 (included)



■ 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



■ 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]
```



- **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

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

x[1] x[3]
↓ ↓

```
Out[1]: [9.0, 5.0]
```




- **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]
```



- 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



- **Combined indexing:** masking+slicing, fancy+slicing
 - Output has the same number 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



- **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

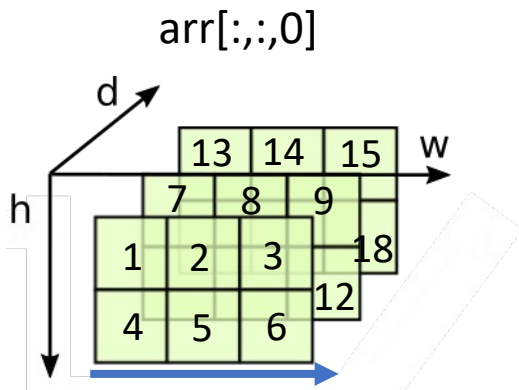
```
x[[True, False, True], 0]  
# Simple + Masking: [0.0, 6.0]
```

0.0	1.0	2.0
3.0	4.0	5.0
6.0	7.0	8.0



■ Simple indexing + slicing

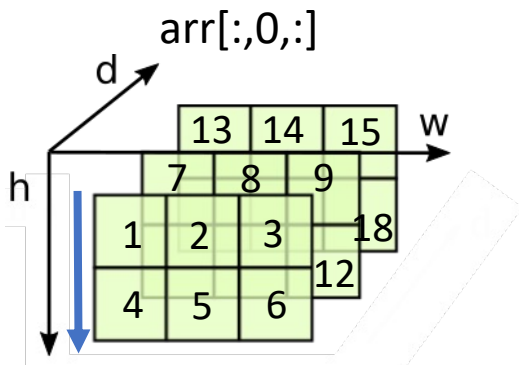
- The dimension selected with simple indexing is **removed** from the output



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

Final output

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



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**



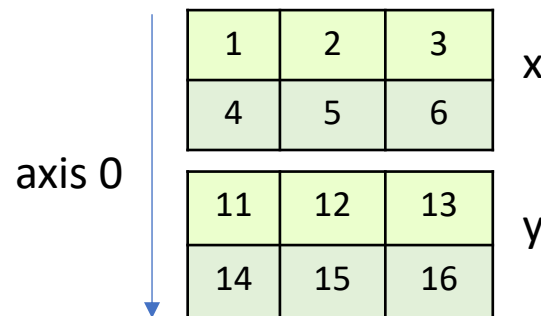


Summary:

- 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

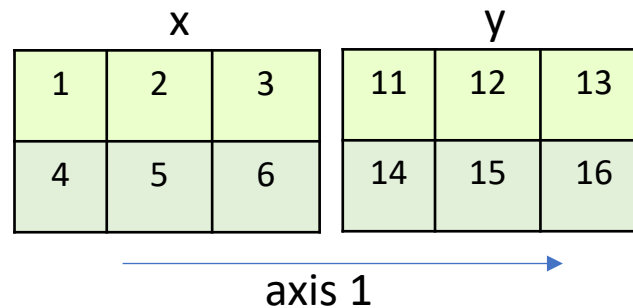


```
In [1]: x = np.array([[1,2,3],[4,5,6]])
        y = np.array([[11,12,13],[14,15,16]])
        np.concatenate((x, y))      # Default axis: 0
```

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



- **Array concatenation along existing axis**
 - Concatenation along **rows (axis=1)**



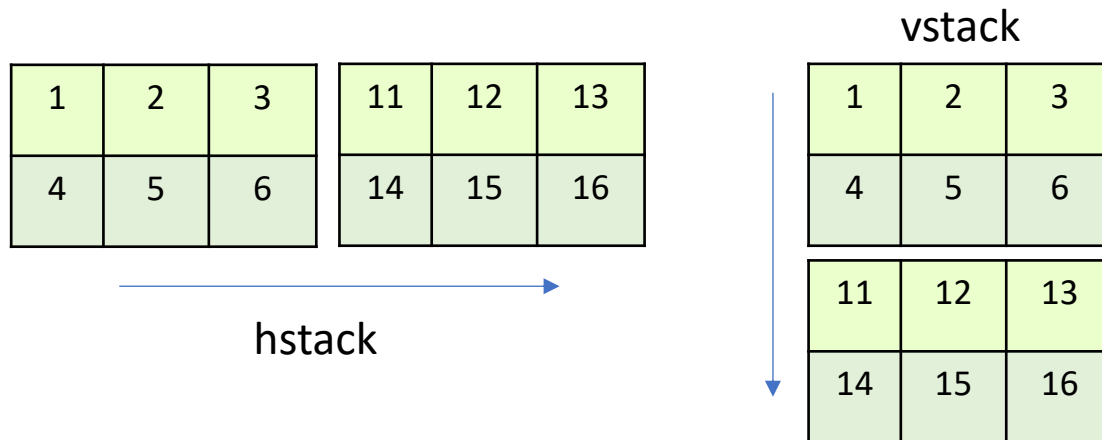
```
In [1]: x = np.array([[1,2,3],[4,5,6]])  
        y = np.array([[11,12,13],[14,15,16]])  
        np.concatenate((x, y), axis=1)
```

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



■ Array concatenation: `hstack`, `vstack`

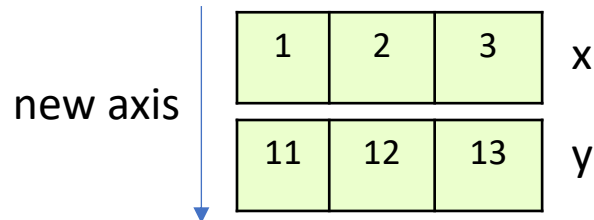
- Similar to `np.concatenate()`



```
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
```



■ 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
		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]:

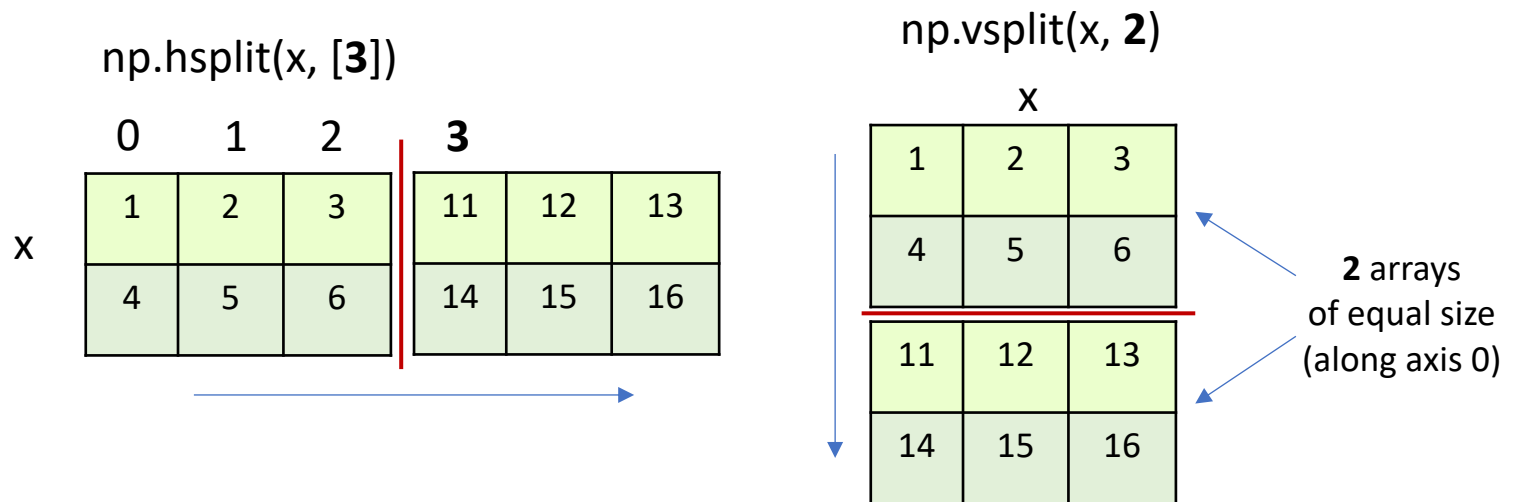
```
[array([7, 7]), array([9, 9]), array([8, 8])]
```



■ Splitting arrays (split, hsplit, vsplit)

■ hsplit, vsplit with 2D arrays

- return a **list** with the arrays after the split



- In both examples output is:

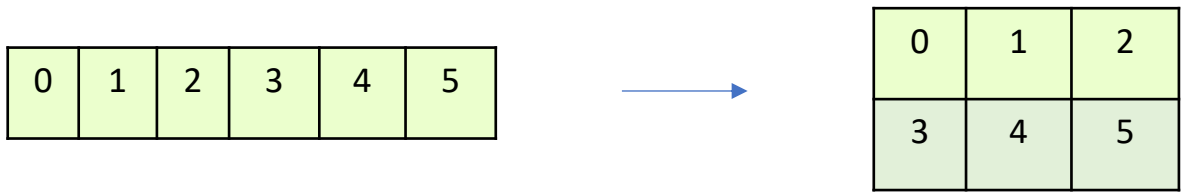
Out: [array([[1,2,3],[4,5,6]]), array([[11,12,13],[14,15,16]])]



■ Reshaping arrays

In [1]:

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



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

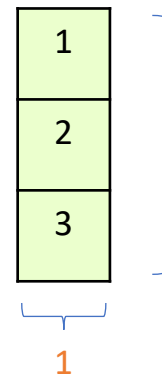
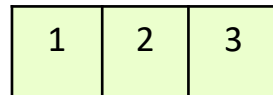


■ Reshaping arrays

- 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

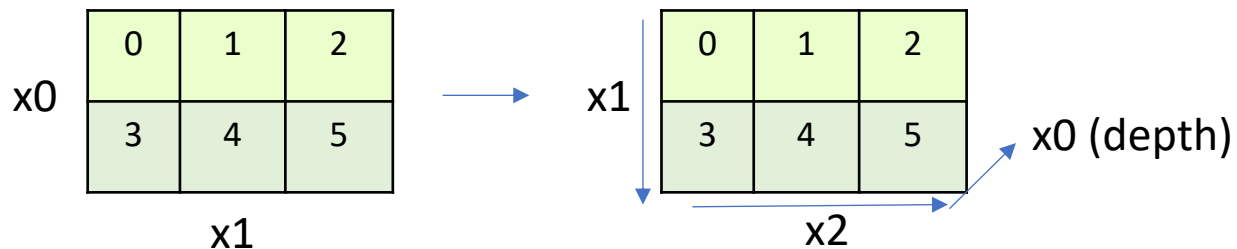
- **np.newaxis** adds a new dimension with **shape=1** at the specified position

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]]
```