Content

1. NumPy Arrays: **Numeric** blocks with **dimensions**
2. **Creation and basic analysis** of arrays
3. Accessing **subsets** of arrays
4. Navigating through **dimensions**
5. Broadcasting and PyTorch tensors

Plan

For each chapter we have:

• Slides	|	concepts
• Demo	|	code
• Script with exercises	|	do-it-yourself (not for chapter 5)

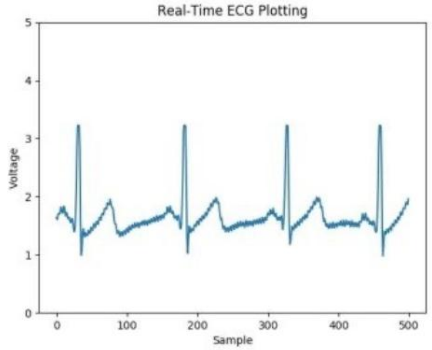
1. NumPy Arrays:
Numeric blocks with dimensions

NumPy is *the* Python package for **numerical data**

Audio waveform of a frog

Audio waveform of a closing door

Biological measurements



Yol et al., 2019

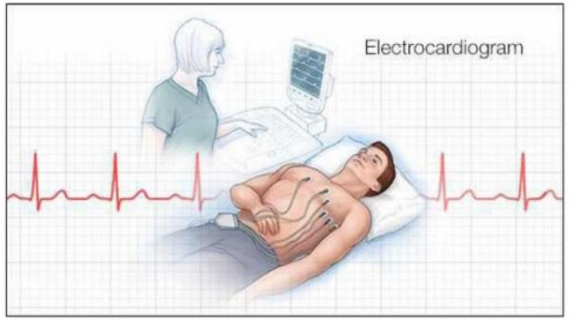
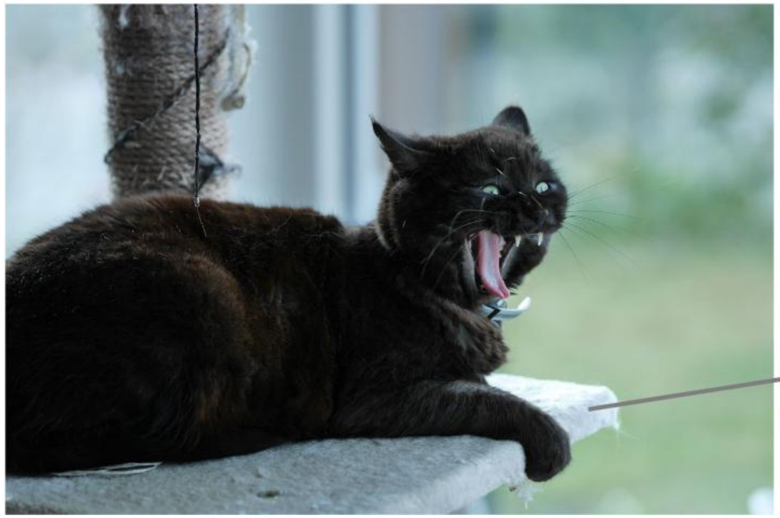


Image files



Representation of an image

$$\begin{bmatrix} 168 & 169 & \cdots \\ 207 & 196 & \\ 169 & 173 & \\ \vdots & & \end{bmatrix}$$

[Red Blue Green]

e.g. [168, 196, 207]

•••

Arrays are *the* data structure of NumPy

array

NumPy

Arrays are the extension of numbers into dimensions.

Arrays are **containers** of elements in a defined order

Similar to lists …

```python
my_list = [1, 2, 3, 4, 5]
my_array = np.array([1, 2, 3, 4, 5])

print("List:", my_list)
print("Numpy Array:", my_array)
```

✓ 0.0s

```
List: [1, 2, 3, 4, 5]
Numpy Array: [1 2 3 4 5]
```

… BUT …

1) An array is a **numeric "blocks"** of data

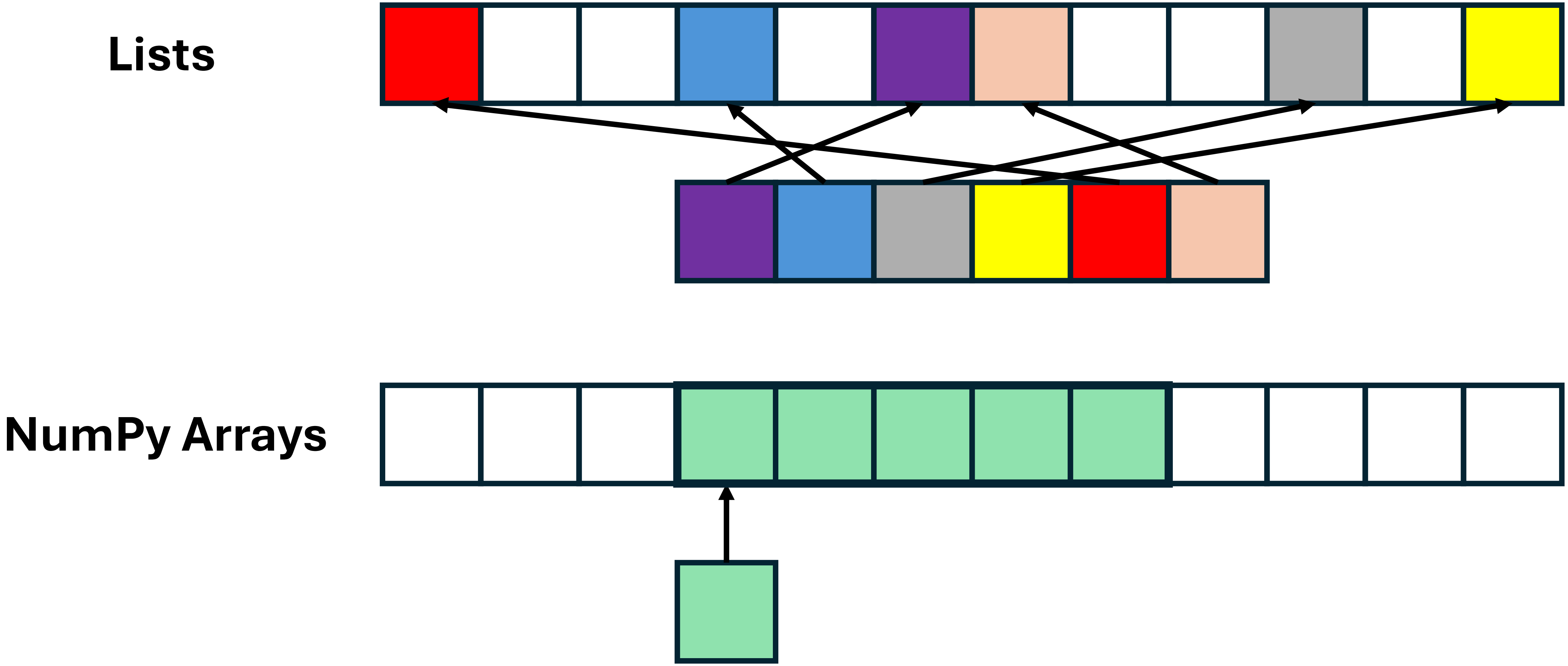
general / flexible

specific / fast

Consequence:

NumPy arrays have a **defined data type**

Sidenote: NumPy arrays = ***one*** pointer to one ***data block***



All data in a NumPy array is of the **defined data type**

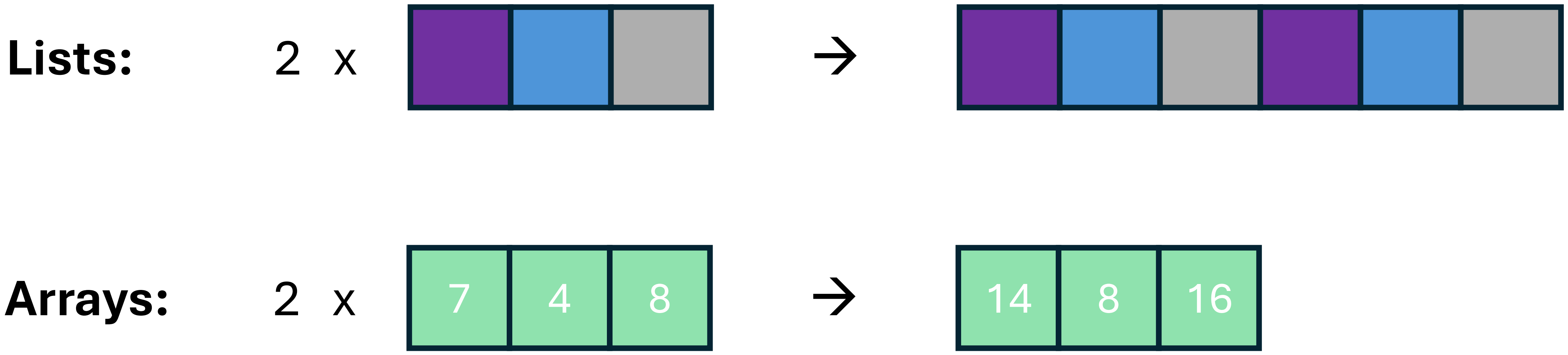
int:	7	4	8	6	8	(int8, int16, uint8 ...)

float:	2.4	0.6	7.01	1.4	0.2	1.21	(float16, float32 ...)

bool:	T	T	F	T

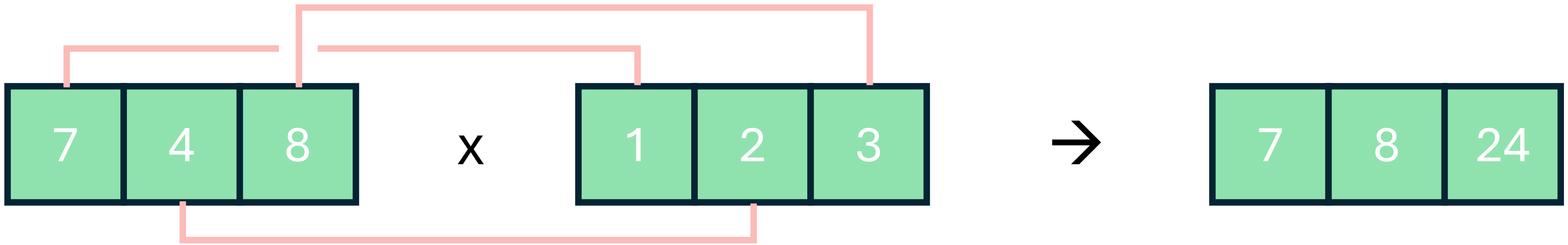
...

Math operations work on the array ***values*** (...not the structure)



... accordingly for addition etc.

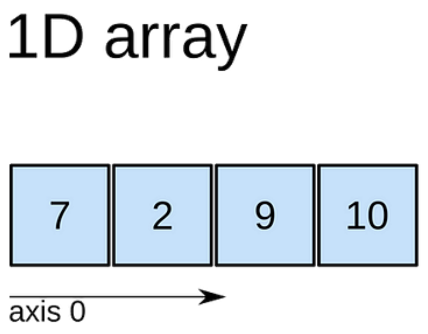
Calculations are **applied element-wise**

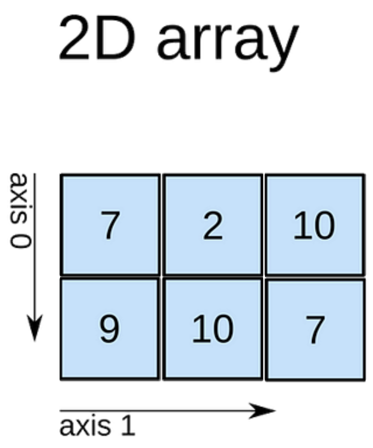


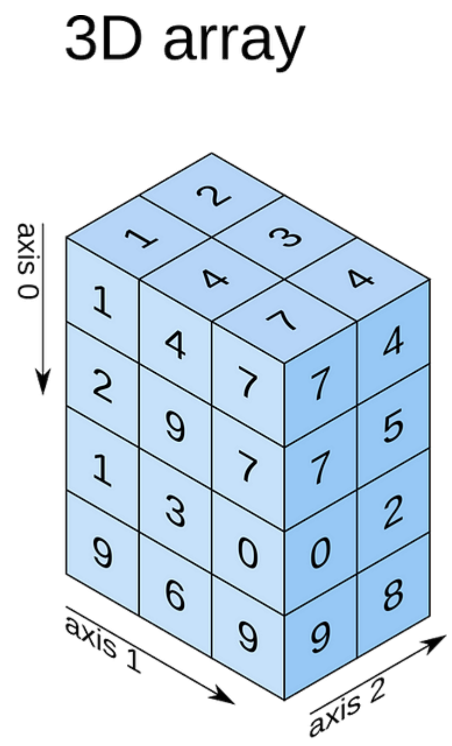
“Vectorization”

2) **Dimensionality** is an inherent property of arrays

Sidenote: An axis can have size 1 (≠ not existent)







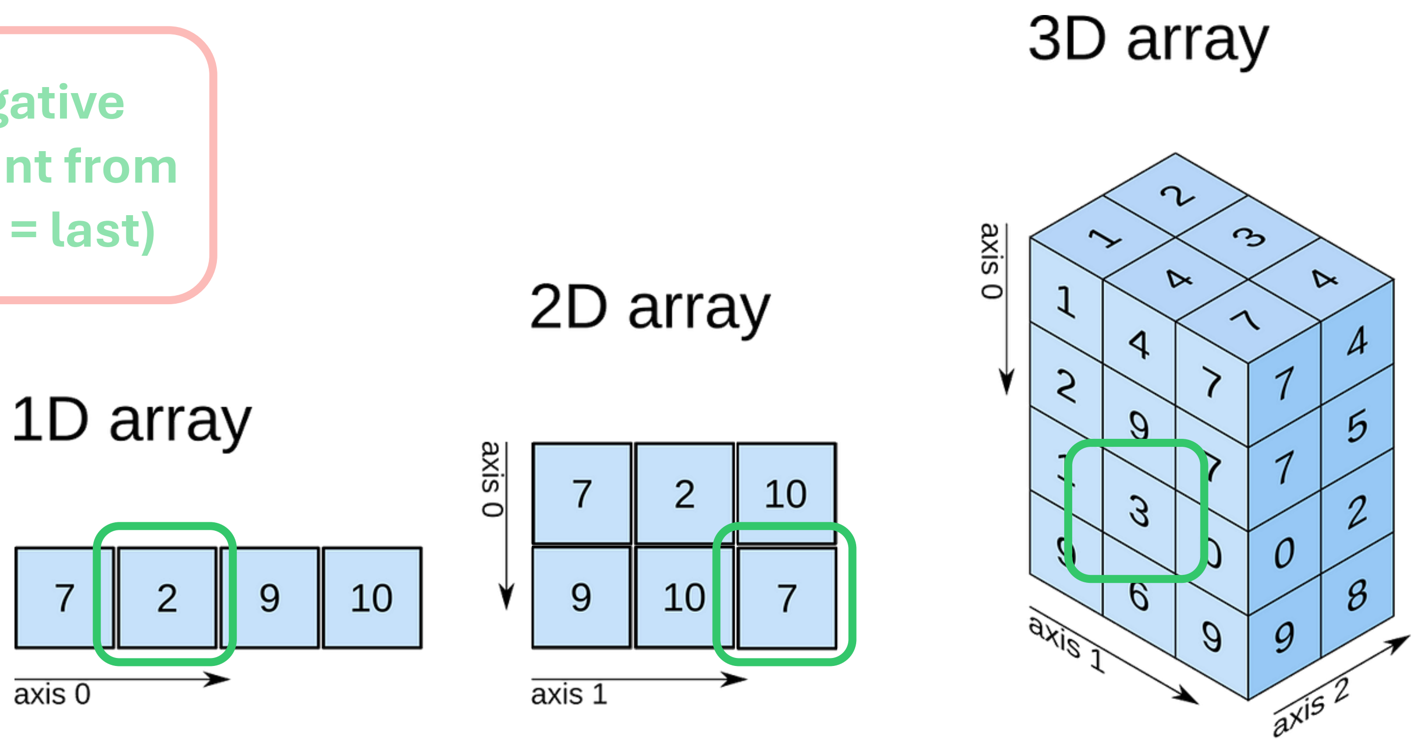
…

Shape:	(4,)	(2, 3)	(4, 3, 2)

Reminder: lists use only "length" (len)

Elements can be **accessed by index – dim by dim**

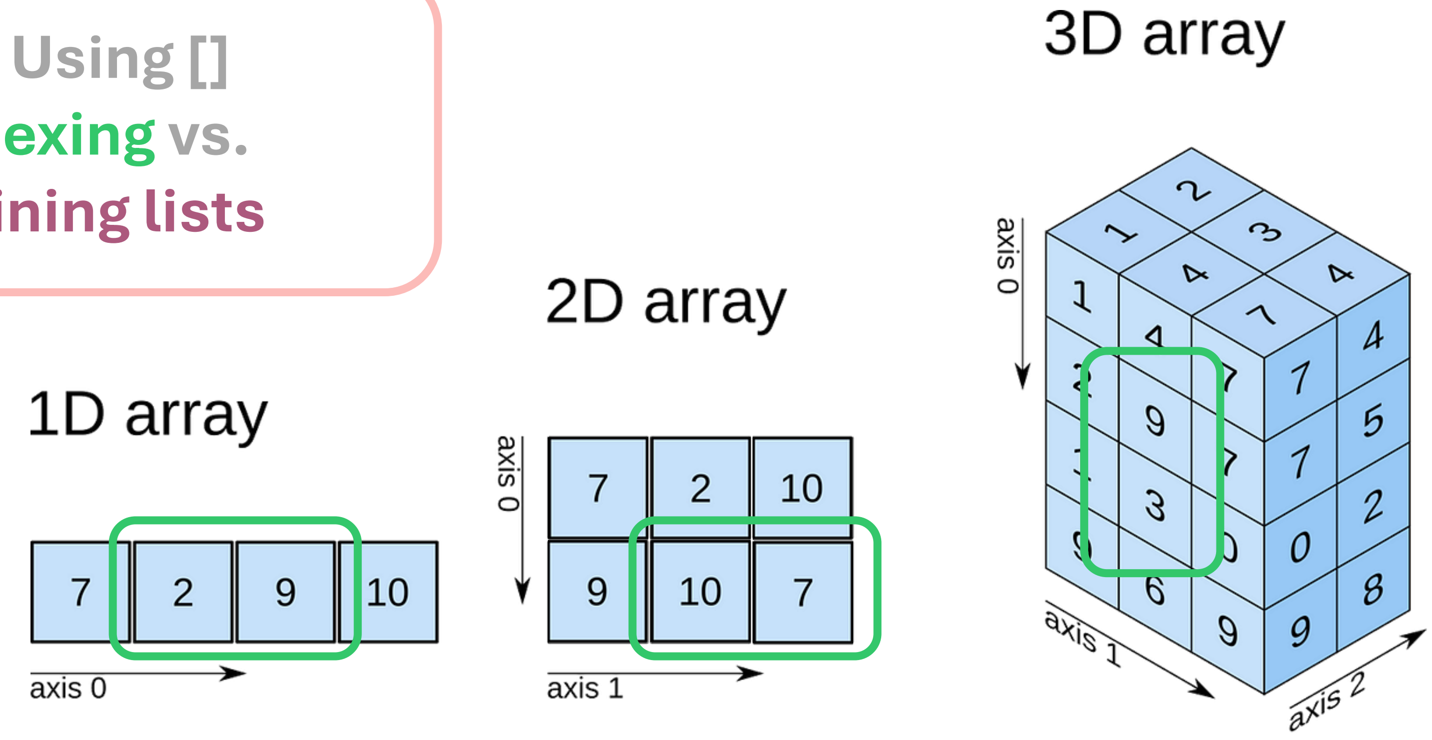
Note: negative indices count from the end (-1 = last)



Index:	[1]	[1, 2]	[2, -2, 0]
Value:	2	7	3

We can **use lists as indices**, resulting in new arrays

Note: Using []
for indexing vs.
for defining lists



Index:	[[1,2]]	[1, [1, 2]]	[[1, 2], 1, 0]
Values:	2, 9	10, 7	9, 3

Demo & Script/Exercises 1

- Read and click through **the first part at your own pace**

  → You may even skip parts if you feel comfortable with the topic
- Make sure to **solve the exercises** at the end

If you're done early:

- **Create** a challenging exercise for other students (about the current topic)

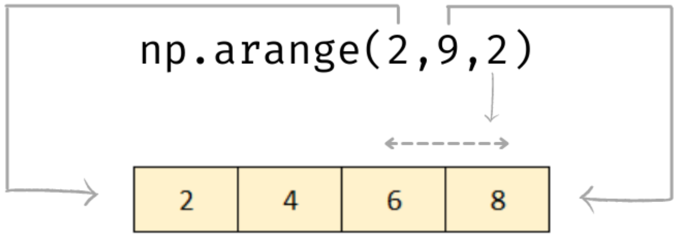
  → send it to roman.Schwob@unibe.ch → I will upload it to github
- Or find and **solve** a challenging exercise created by another student
- (Or take a longer break – this is also good coding practice...)

2. Creation and basic analysis of arrays

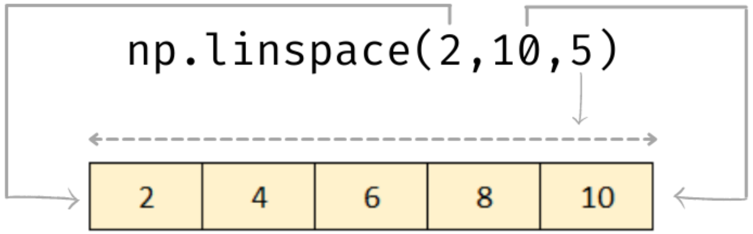
There are **many ways** to create new NumPy arrays

Equally spaced values:

specify the **step**



specify the **length**



Fixed values, **given shape**: np.zeros([2,3])

0	0	0
0	0	0

np.ones([2,3])

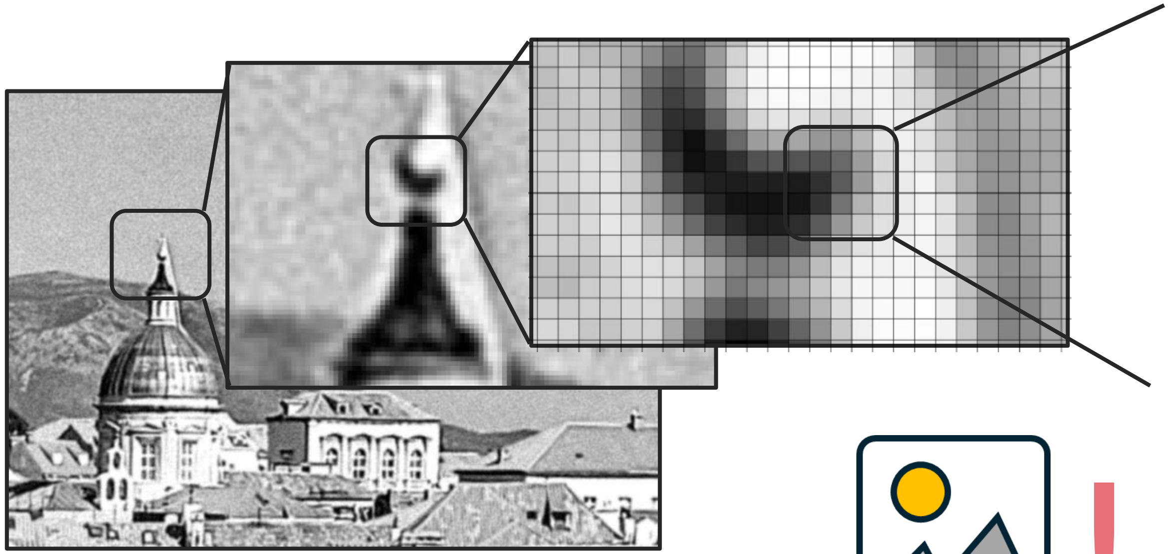
np.ones([2,3]) * 3 → ?

np.full([2,3], 3) → ?

Random values: np.random ...

.randint(), .random()

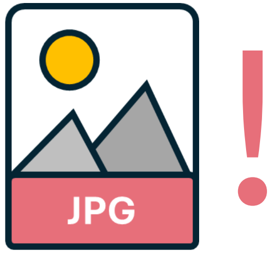
Arrays are often initialized by **loading data – ex. images**



121	113	148	178	201
110	109	125	152	199
78	101	115	152	199
23	109	125	163	199
101	112	150	200	199

NumPy

```
121, 113, 148, 178, 201,
110, 109, 125, 152, 199,
 78, 101, 115, 152, 199,
 23, 109, 125, 163, 199,
101, 112, 150, 200, 199,
```



Use provided packages to **load** images right into NumPy arrays

```python
import skimage

my_img = skimage.io.imread("my_img_file.jpg")
```

NumPy arrays can be displayed as images using **pyplot**

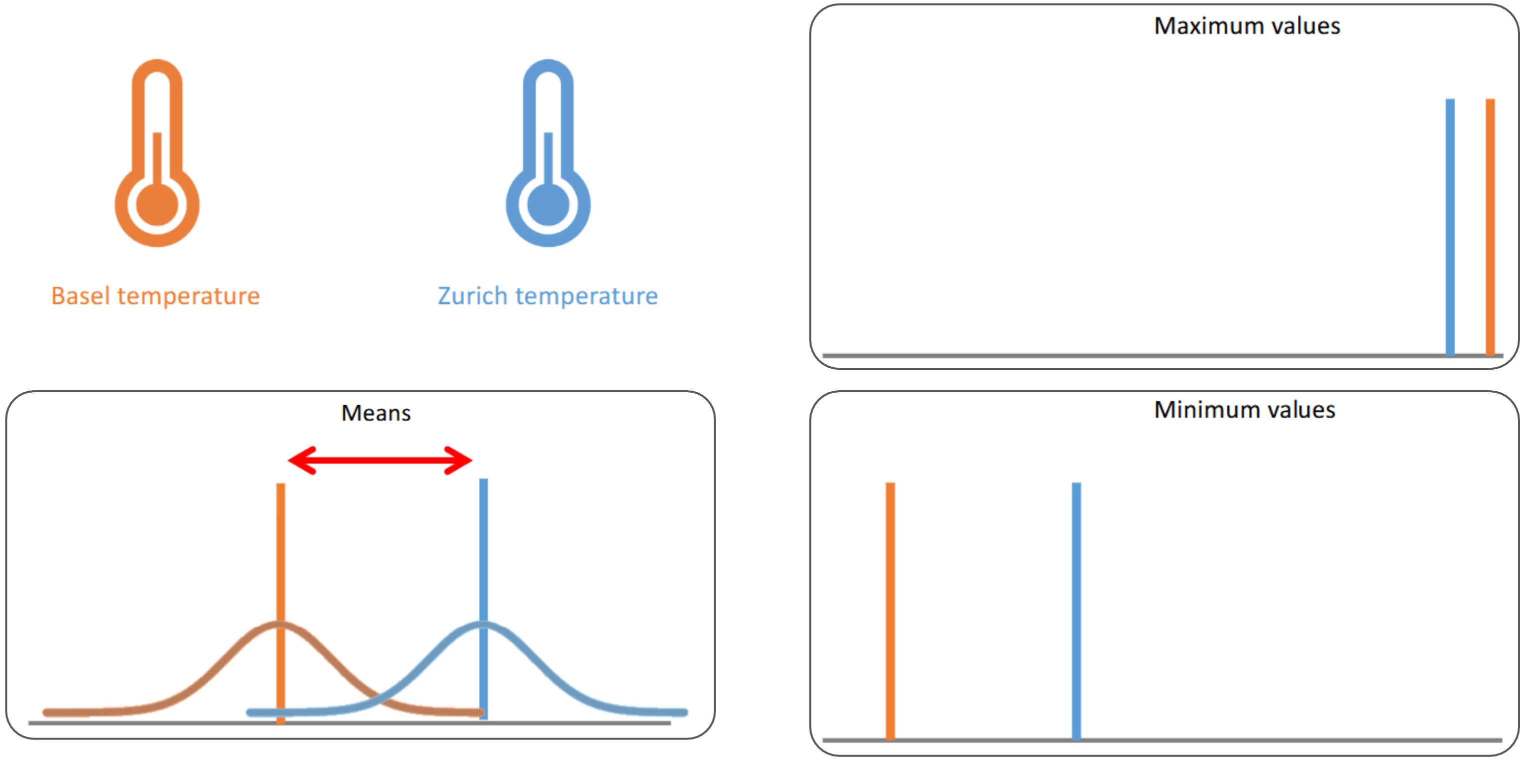
```
from matplotlib import pyplot as plt

plt.imshow(my_img_array, cmap='gray')
plt.show()
```

cmap = “gray”

cmap = “viridis”

Note: We use a colormap to define how different values are displayed.
(... except for RGB images, where it is pre-defined.)

We can use NumPy methods to **aggregate data**



Summary statistics:

- Mean `np.mean`
- Standard deviation `np.std`
- Sum `np.sum`
- Product `np.prod`
- Median `np.median`
- Minumum `np.min`
- Maximum `np.max`
- …

Code: `my_arr.mean()` is identical with `np.mean(my_array)` `(etc.)`

Note: use numpy methods, NOT python equivalents (much slower)

Demo & Script/Exercises 2

- Read and click through **the first part at your own pace**

  → You may even skip parts if you feel comfortable with the topic
- Make sure to **solve the exercises** at the end

If you're done early:

- **Create** a challenging exercise for other students (about the current topic)

  → send it to roman.Schwob@unibe.ch → I will upload it to github
- Or find and **solve** a challenging exercise created by another student
- (Or take a longer break – this is also good coding practice…)

3. Accessing subsets of arrays

Slicing allows to access **parts** of an array

```
my_array[start:stop:step]
```

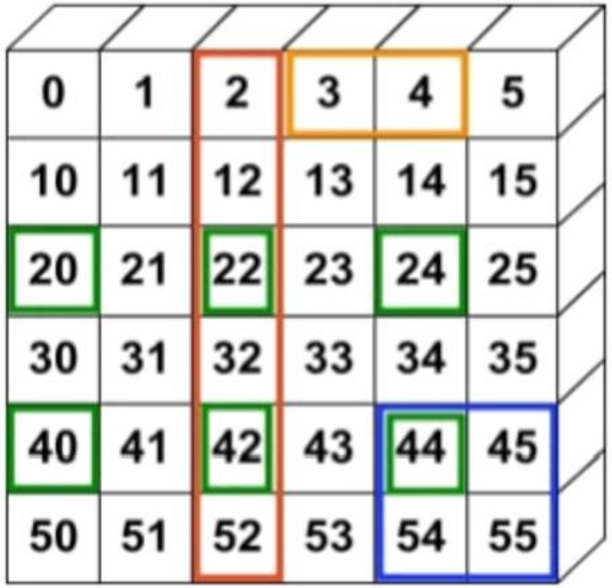
`start`:	*including* the number	(default = 0)
`stop`:	*excluding* the number	(default = size of dim)
`step`:	negative = reverse	(default = 1)

Remember: In Python, indices start at 0

Slicing works on **multidimensional** arrays

- Equivalent to indexing, can be combined
- Use ` : ` to select all elements of a dimension, ex. my_array[:, :3]

```
>> a[0,3:5]
array([3, 4])

>> a[4:,4:]
array([[44, 45],
       [54, 55]])

>> a[:,2]
array([ 2, 12, 22, 32, 42, 52])

>> a[2::2,::2]
array([[20, 22, 24],
       [40, 42, 44]])
```



Slicing **creates “views”** (not copies)

Create a copy if needed!

```
my_window = array1[:3, :]
array2 = my_window.copy()
```

```
array2[2,2] = 1
```

→

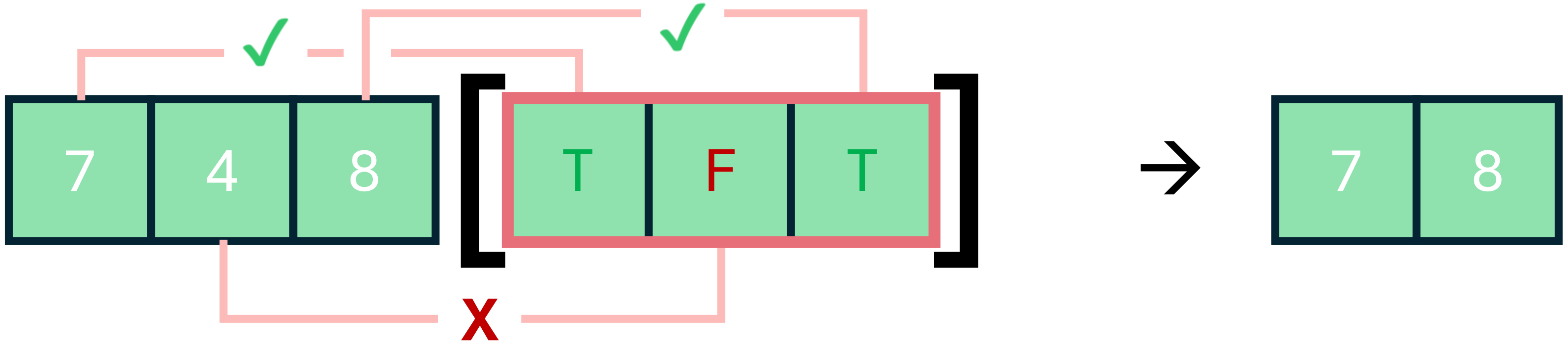
Different to python lists!

We can also use **comparisons** to get subsets of arrays

Recall: 2 x [7 | 4 | 8] → [14 | 8 | 16]

Comparison: [7 | 4 | 8] **> 5** → [T | F | T] **(“mask”)**

Using **masks as indices:** `arr[ arr>5 ]`



Demo & Script/Exercises 3

- Read and click through **the first part at your own pace**

  → You may even skip parts if you feel comfortable with the topic
- Make sure to **solve the exercises** at the end

If you're done early:

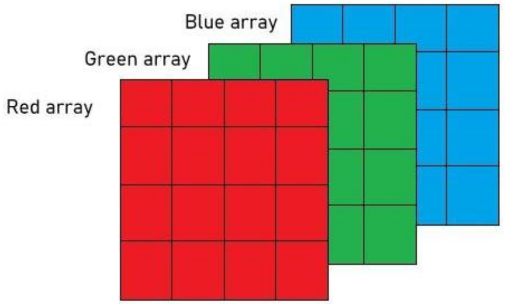
- **Create** a challenging exercise for other students (about the current topic)

  → send it to roman.Schwob@unibe.ch → I will upload it to github
- Or find and **solve** a challenging exercise created by another student
- (Or take a longer break – this is also good coding practice...)

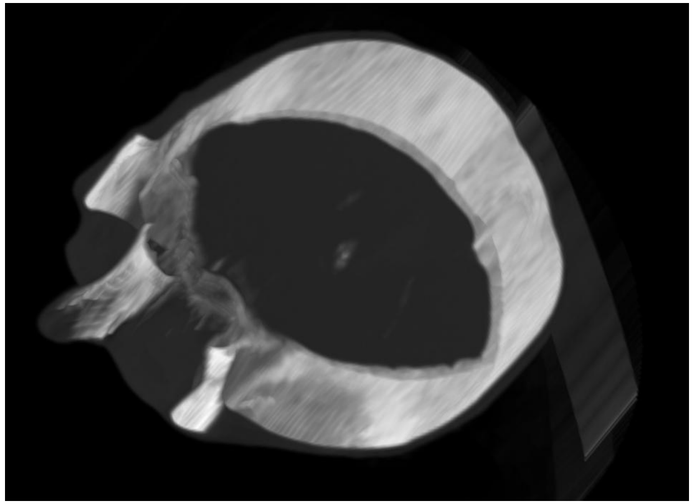
4. Navigating through dimensions

4D+ is hard to visualize but very abundant and important

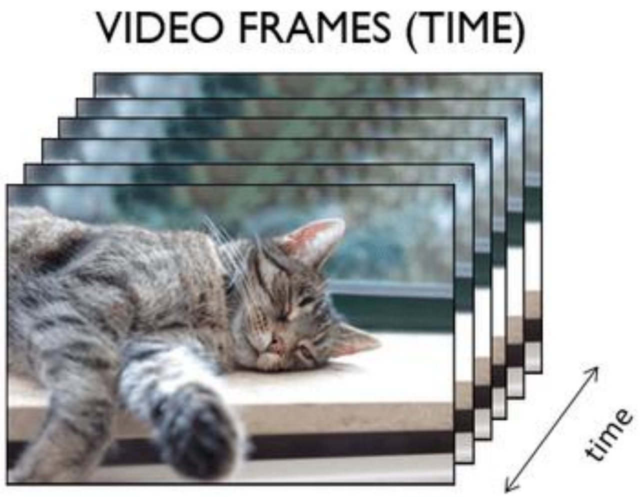
RGB (multichannel)



3D (spatial)



Video (time)

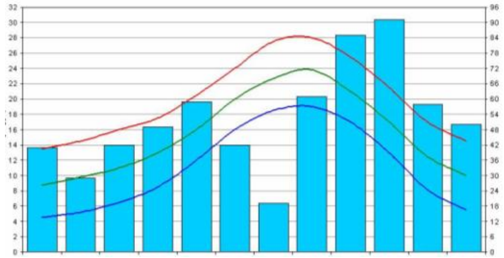


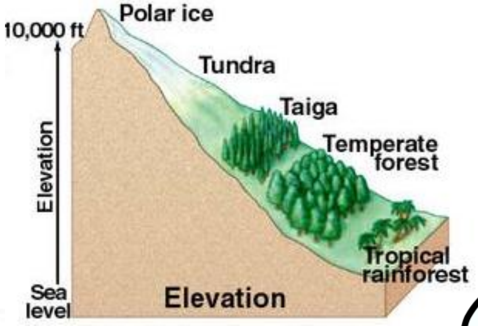
Now think about a 3D RGB video ...

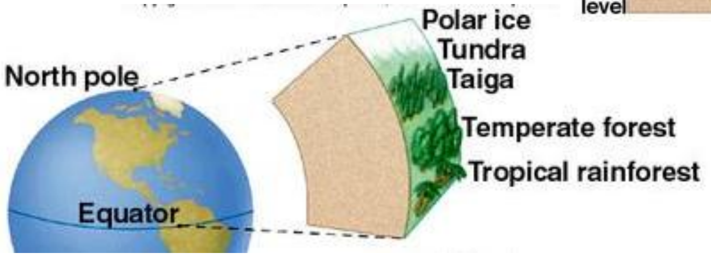
4D+ is hard to visualize but very abundant and important

Example climate factors for plant growth:

- Humidity
- Temperature
- Elevation
- Radiation
- Latitude
- ...







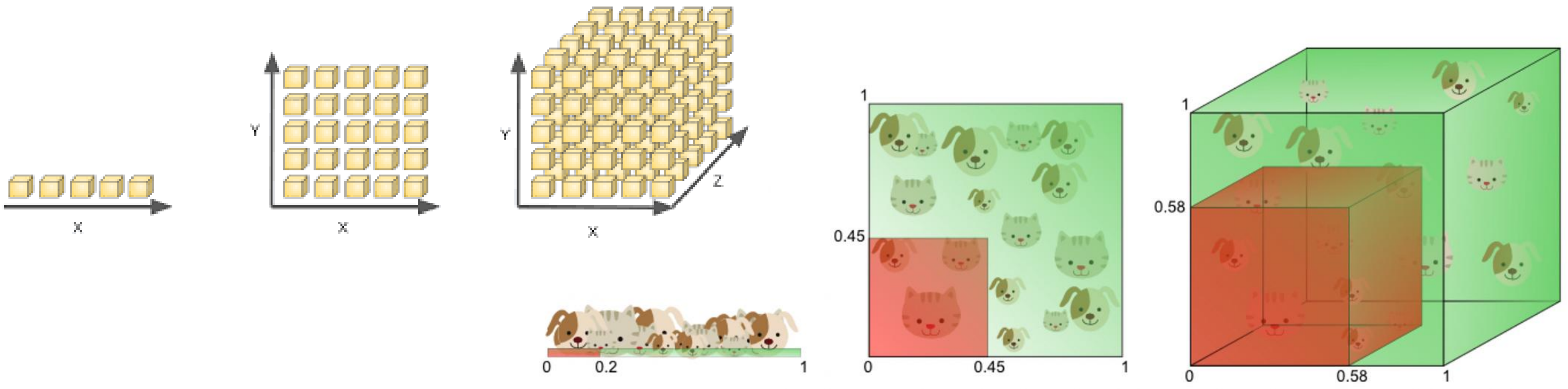
*Tipp: It can help to think **what happens on lower dimensions** and then apply it to the actual data. You do not have to visualize it to **trust its math**!*

Size of data increases **exponentially** with dimensions

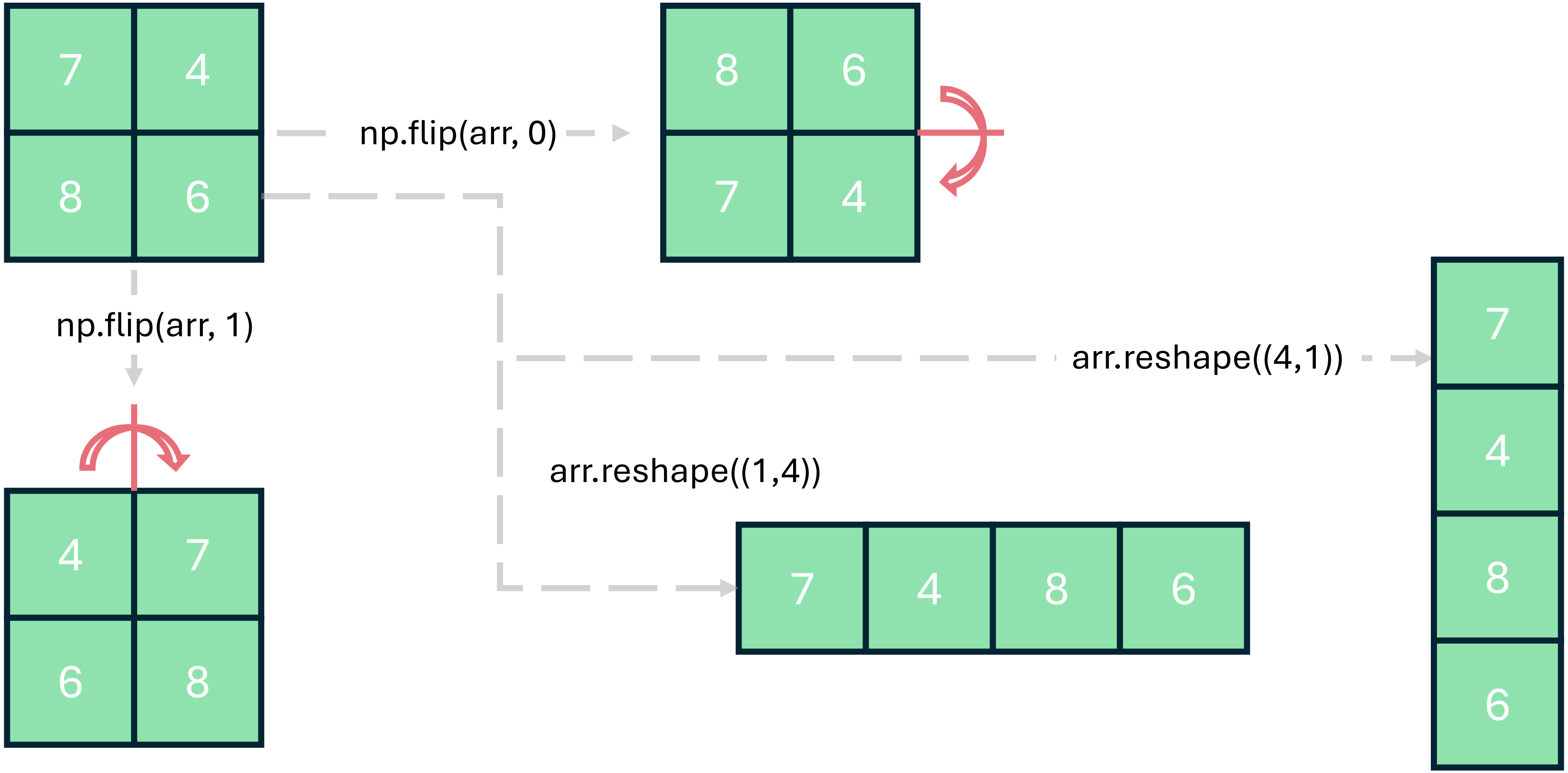
2D with 20 el. each vs. 5D with 4 el. each

400 1’024

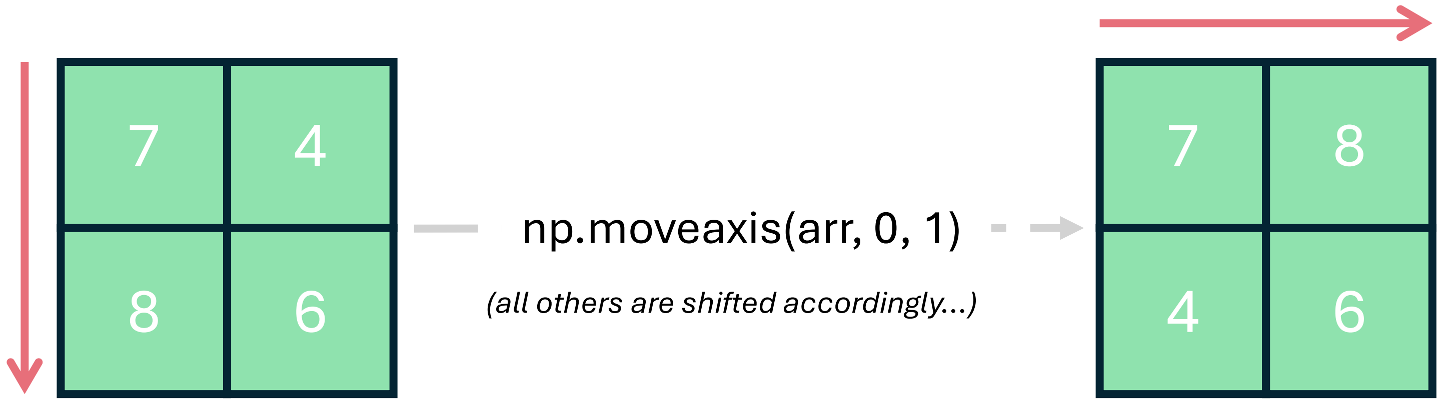
Beware the “**curse of dimensionality**”



Rearrange array dimensions using **flip, and reshape**



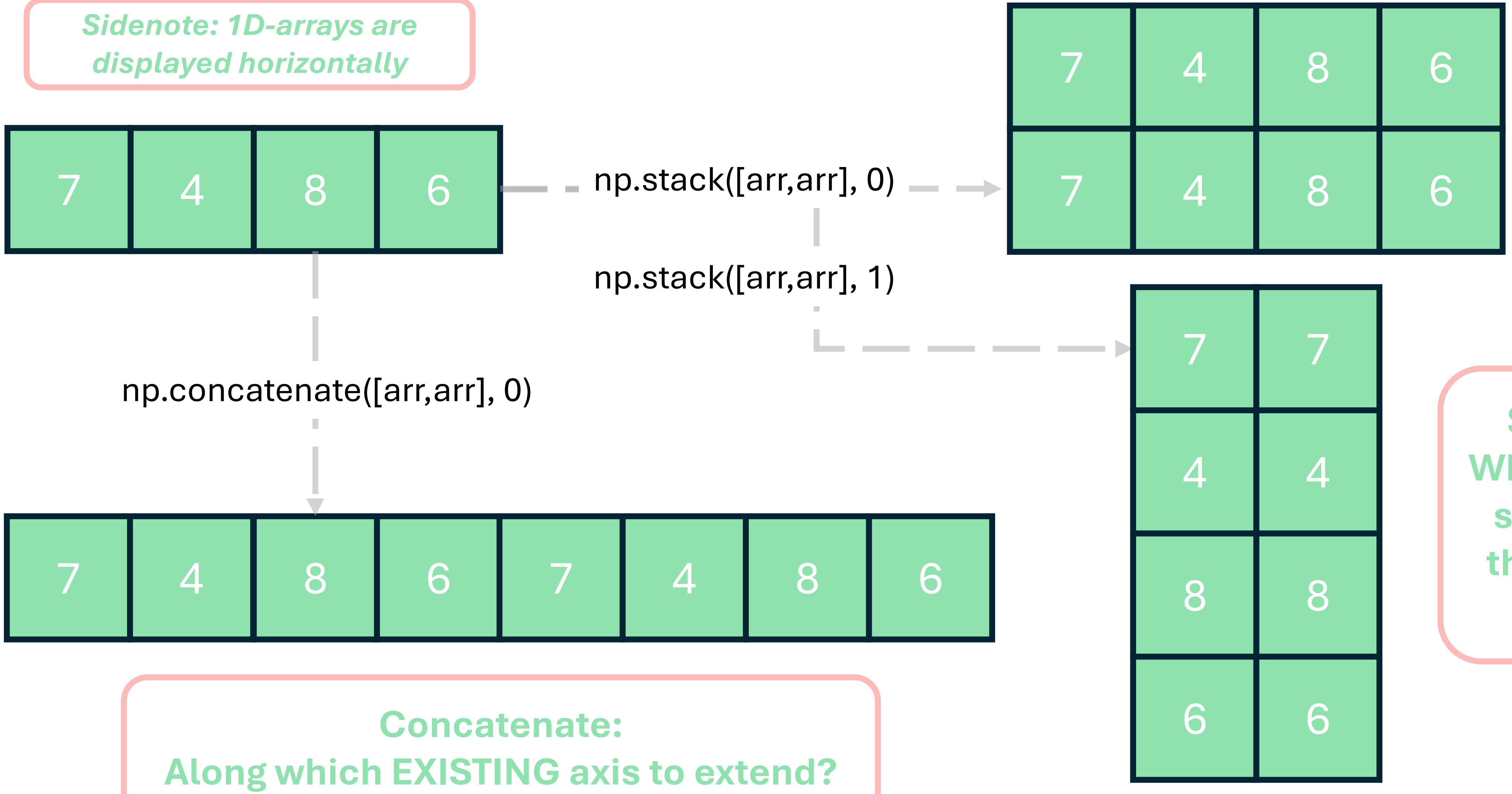
Rearrange array dimensions using **moveaxis**



Note: In this simple 2D case, we only have one option, which simply rotates the array
However, in higher dimensions we have many possibilities to move an axis !

Extend arrays using **stack and concatenate**

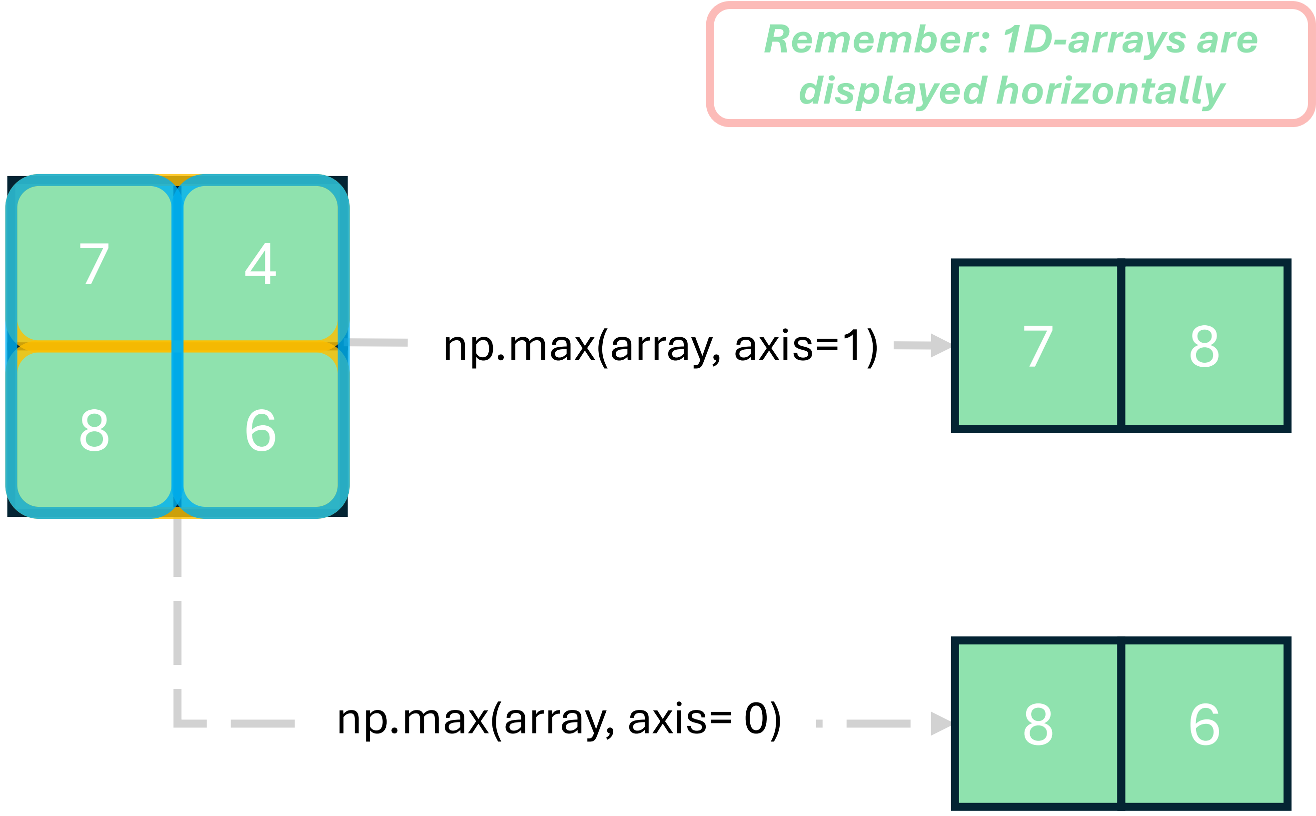
Sidenote: 1D-arrays are displayed horizontally



Concatenate:
Along which EXISTING axis to extend?

Stack:
Which one shall be the NEW axis?

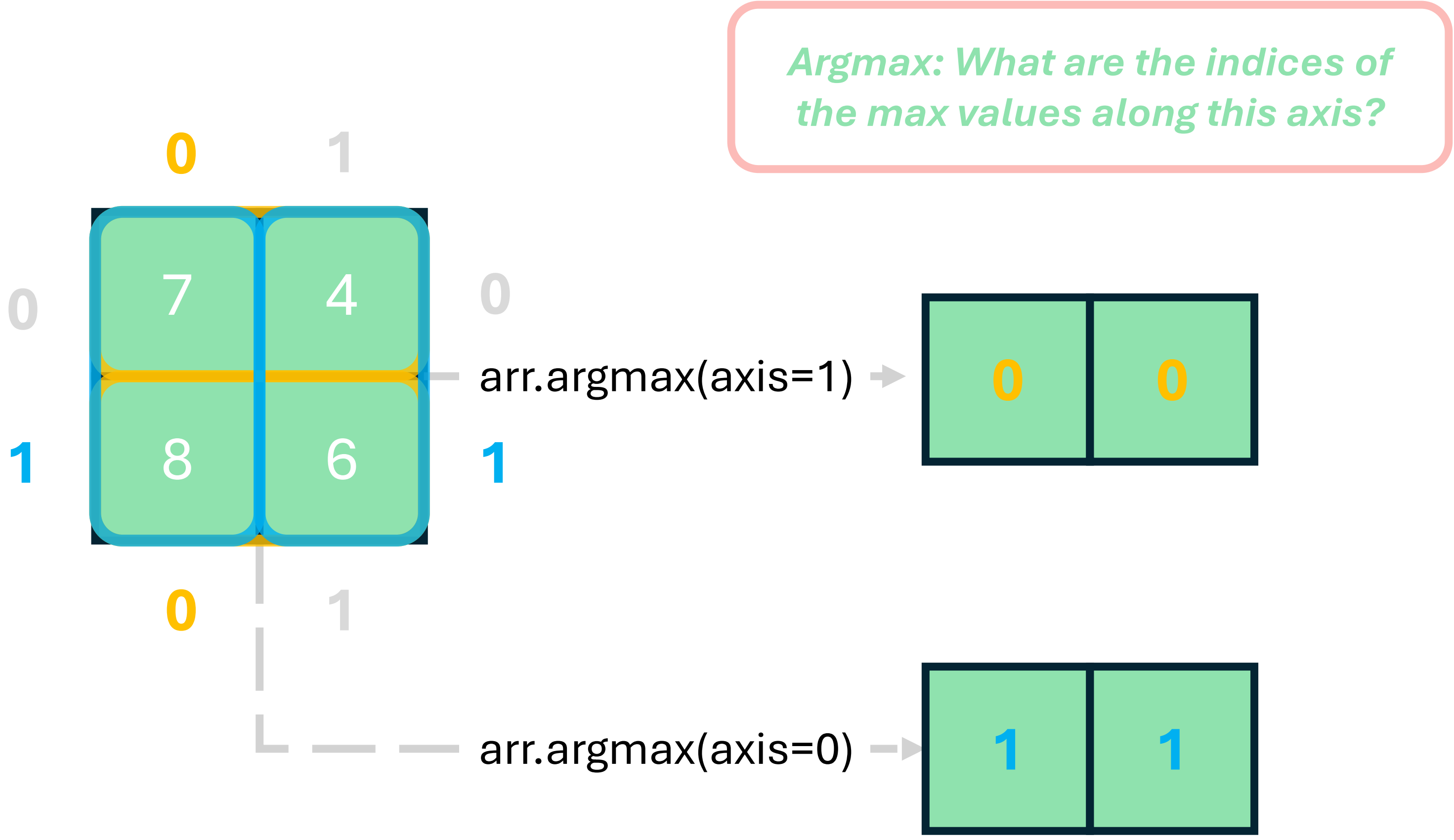
Aggregating functions can be applied **on axes**



Summary statistics:

- Mean np.mean
- Standard deviation np.std
- Sum np.sum
- Product np.prod
- Median np.median
- Minumum np.min
- Maximum np.max
- ...

We can also get the **location** of elements (argmax/argmin)



`np.argmin()`: same idea

We can also get the **location** of elements (argwhere)

numpy.argwhere

`numpy.argwhere(a)` [source]

Find the indices of array elements that are non-zero, grouped by element.

Parameters:

a : *array_like*

Input data.

Returns:

index_array : *(N, a.ndim) ndarray*

Indices of elements that are non-zero. Indices are grouped by element. This array will have shape `(N, a.ndim)` where `N` is the number of non-zero items.

Important: True = 1, False = 0

np.argwhere(arr == 7) → [0, 0]
np.argwhere((arr != 6) **&** (arr != 8)) → [[0, 0], [0, 1]]
np.argwhere((arr == 6) **|** (arr == 4)) → [[0, 1], [1, 1]]

7	4
8	6

np.argwhere(arr>=6)

array([[0, 0],
[1, 0],
[1, 1]], dtype=int64)

Sidenote: np.where() slightly different ...

T	F
T	T

arr>=6
(= mask)

Demo & Script/Exercises 4

- Read and click through **the first part at your own pace**

  → You may even skip parts if you feel comfortable with the topic
- Make sure to **solve the exercises** at the end

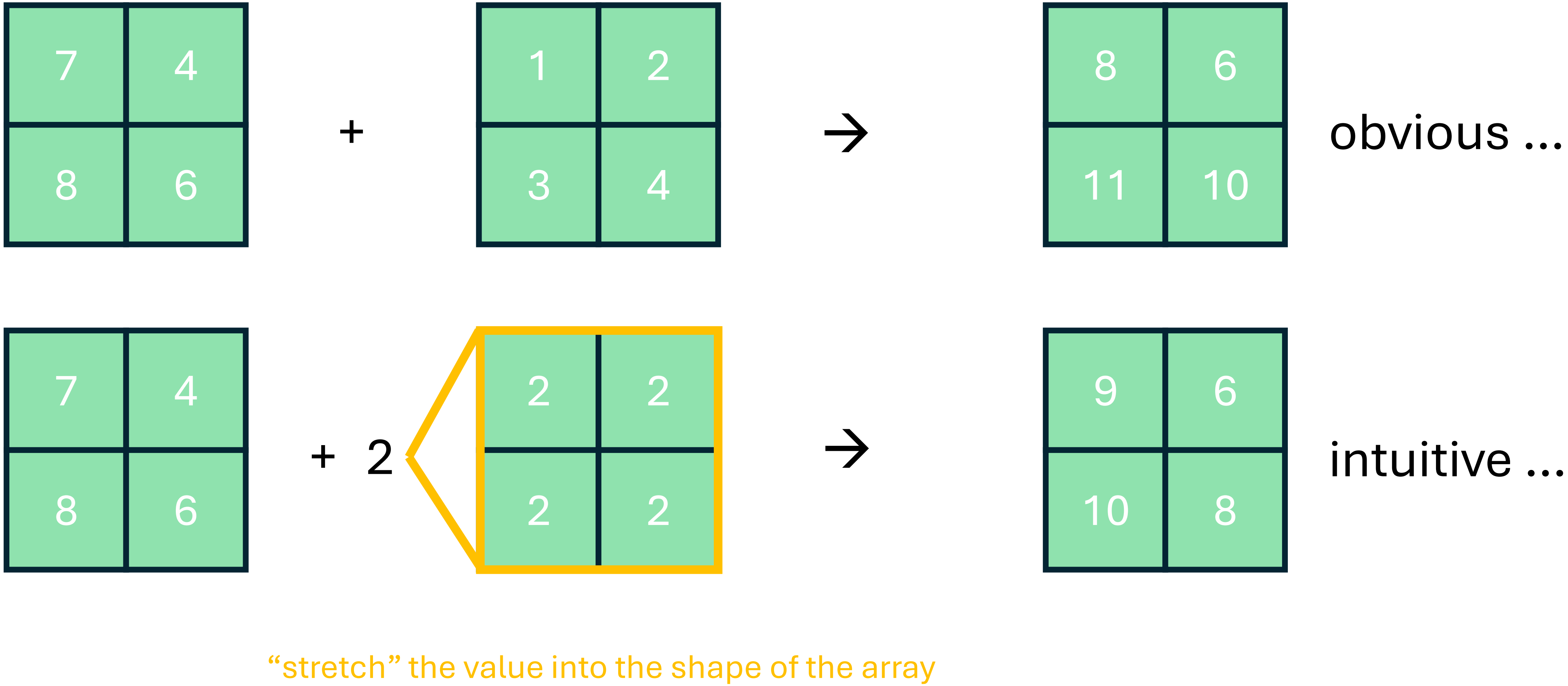
If you're done early:

- **Create** a challenging exercise for other students (about the current topic)

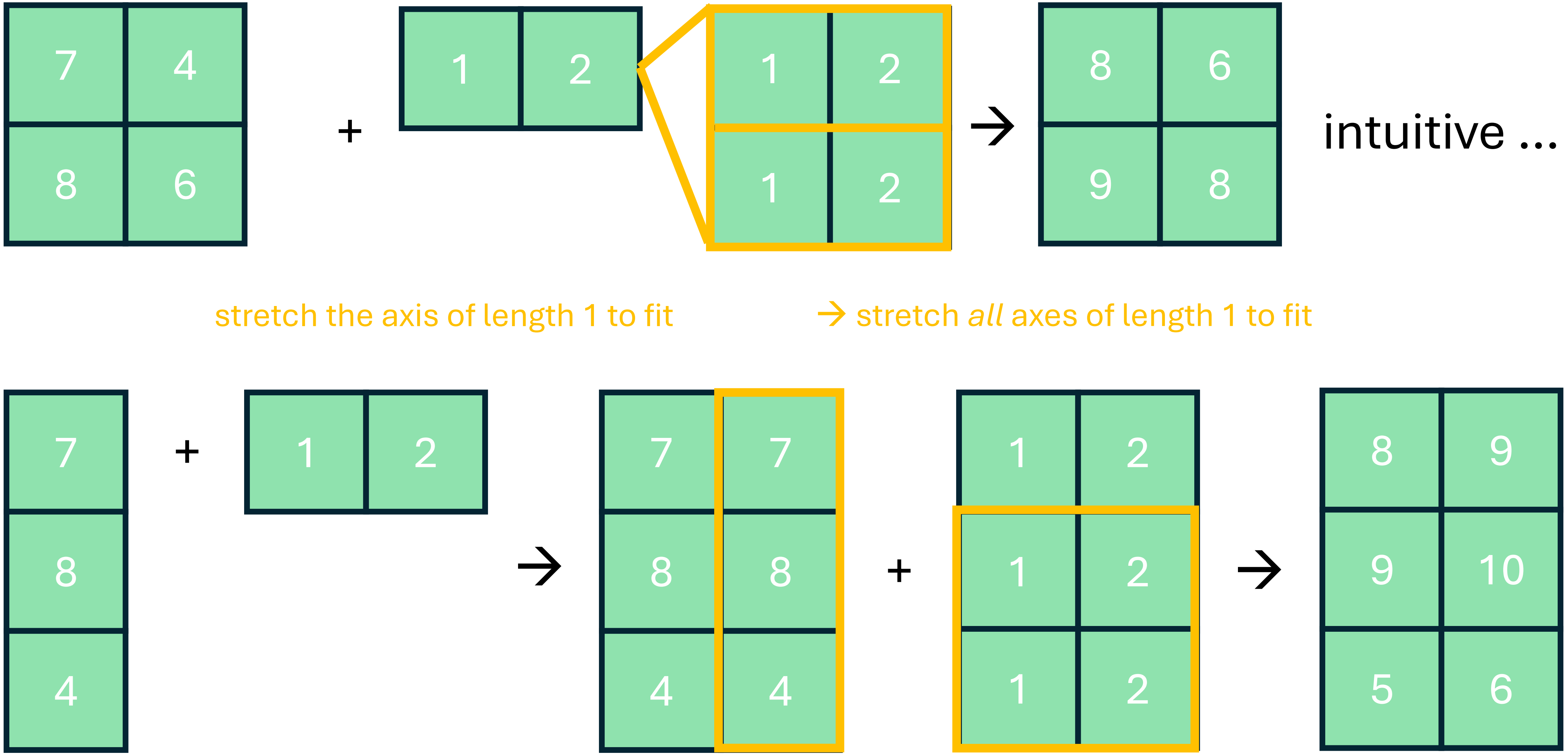
  → send it to roman.Schwob@unibe.ch → I will upload it to github
- Or find and **solve** a challenging exercise created by another student
- (Or take a longer break – this is also good coding practice…)

5. Broadcasting and PyTorch tensors

Operations are applied element-wise: **Broadcasting**



Operations are applied element-wise: **Broadcasting**



Operations are applied element-wise: **Broadcasting**

Three **rules of broadcasting** two arrays:

1. Dimensions differ: **pad shape** of the one with fewer dimensions **with ones** on the leading (left) side
   → create "fake higher-dimensional" (ex. [3,3] → [1,3,3])

2. In each dimension: mismatch → **stretch array with shape 1**

3. In each dimension: mismatch & no shape == 1 → error

Sorting arrays

Go to the **NumPy documentation** and find out how to sort arrays.

Alternatively: Ask ChatGPT or another **AI** friend ...

Try it on **examples** ...

Time: **~5 minutes**

(You were too fast? → find out what np.argsort() does!)

Torch tensors: deep learning version of NumPy arrays

Most things **similar** to NumPy: calculations, shape, slicing etc.

Also: **Easy conversion**... NumPy ↔ PyTorch

Some **new functionalities** (useful for DL, e.g. matrix computation):

- **GPU**: torch.tensor([1.0, 2.0]).**to('cuda')** # runs on GPU
- **Grad**: torch.tensor([2.0], **requires_grad=True**) # use derivatives

Also: some methods have (slightly) different names

Demonstrations 5