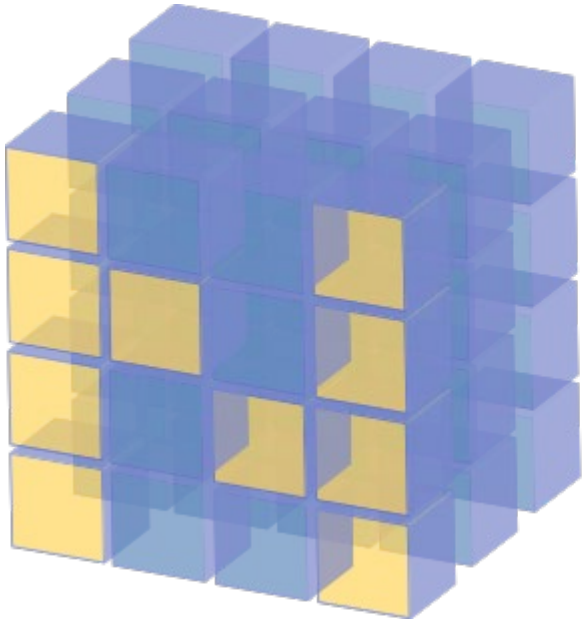
Information, Computation, Communication

Learning Python



NumPy

Agenda

- What is NumPy?
- Array creation
- Array indexing and slicing
- Iterating over arrays
- Array-manipulation routines
- Array methods
- Broadcasting

What is NumPy?

NumPy

- NumPy (numerical Python) is a Python library providing support for large, multi-dimensional arrays and matrices, together with a rich collection of mathematical functions to **efficiently** work with these arrays and matrices

- NumPy is the core Python library for scientific computing

- NumPy allows applying linear algebra operations to n-dimensional arrays without using for loops

NumPy

- NumPy gives Python the functionality comparable to MATLAB
- Internally, NumPy and MATLAB both rely on
  - Basic linear algebra subprograms (**BLAS**) - efficient routines for linear algebra operations
    - vector addition, scalar multiplication,
    - dot product, linear combinations,
    - matrix multiplication, etc.
  - Linear algebra package (**LAPACK**) - efficient routines for solving
    - systems of linear equations, eigenvalue problems,
    - singular value decomposition, matrix factorizations, etc.

Importing NumPy

- NumPy shares the names of its functions with functions in other modules (e.g., math module in Python standard library). Therefore, it is not recommended to use imports like the following

```
from numpy import *
```

- Instead, it is recommended to follow the convention of using `np` as alias:

```
import numpy as np
```

ndarray

- Core functionality of NumPy is its *N-dimensional array* (**ndarray**)
- In contrast to Python's built-in lists, ndarrays are homogeneously typed:

All elements of the ndarray must be of the same type

Array Creation

Array creation

- To convert lists to NumPy arrays:
  - use **array()** function and
  - pass a list (or a nested list) to it

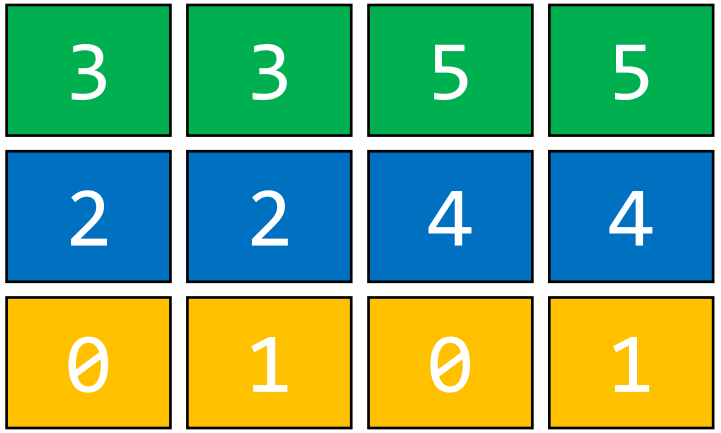
- Example of creating a one-dimensional array:

```
arr1d = np.array([2,2,4,4])
print(arr1d)
# [1 2 3 4]
```

Array creation

- Example of creating a two-dimensional array
- Default order is **row-major** (C-language style)

```
arr2d = np.array([[3,3,5,5],
                  [2,2,4,4],
                  [0,1,0,1]])

print(arr2d)
# [[3 3 5 5]
#  [2 2 4 4]
#  [0 1 0 1]]
For mismatched list sizes, deprecated warning will appear
```

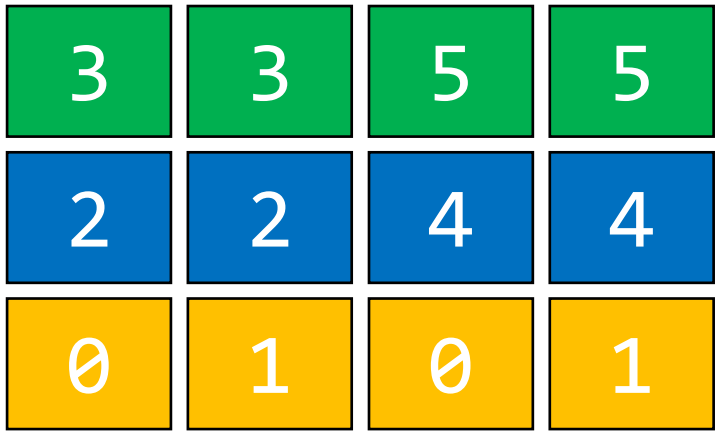


Array Attributes

- ***size***: number of elements in the array
- ***ndim***: number of array dimensions ***N***
- ***shape***: a tuple of ***N*** positive integers specifying the sizes of each dimension

Example:

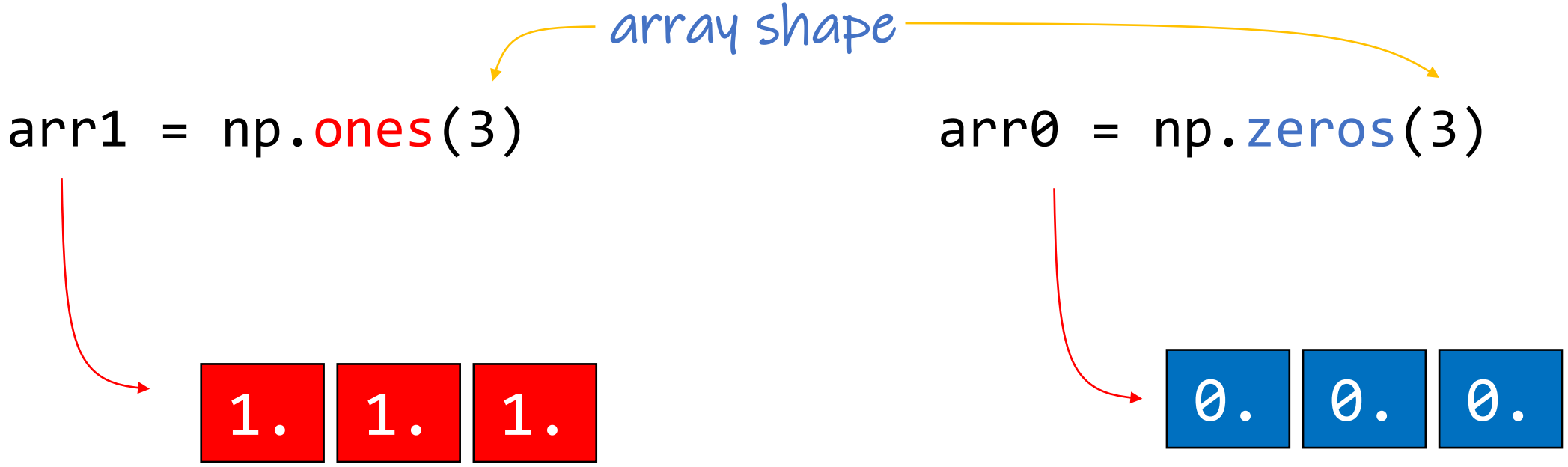
arr2d




```
print(arr2d.size)  # 12 elements
print(arr2d.ndim)  # 2 dimensions
print(arr2d.shape) # (3,4)
```

Intrinsic Array Creation: zeros() and ones()

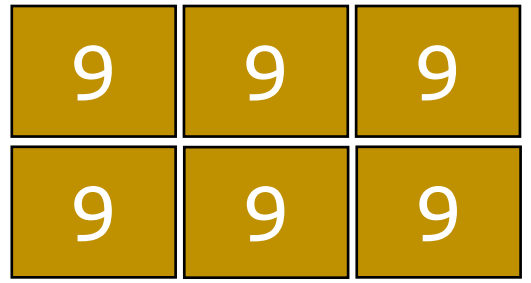
- NumPy provides methods for quick creation of arrays
- Default data type: float (64 bits)
- Examples:



Intrinsic Array Creation: full()

- NumPy provides methods for quick creation of arrays
- Can be used to create integer arrays; the type of the array data depends on the type of the fill value
- Example:

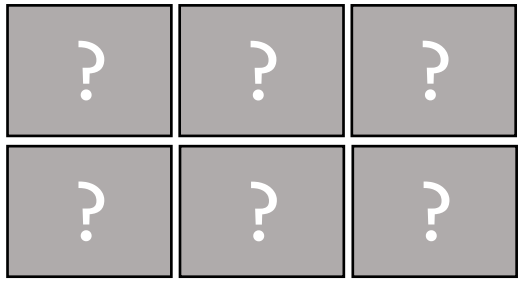
```
arrf = np.full(shape=(2,3), fill_value=9)
# or, in short
# arr3 = np.full((2,3), 9)

# Note: the type of the elements
# will be int here, because
# the fill_value here is an int
```



Intrinsic Array Creation: empty()

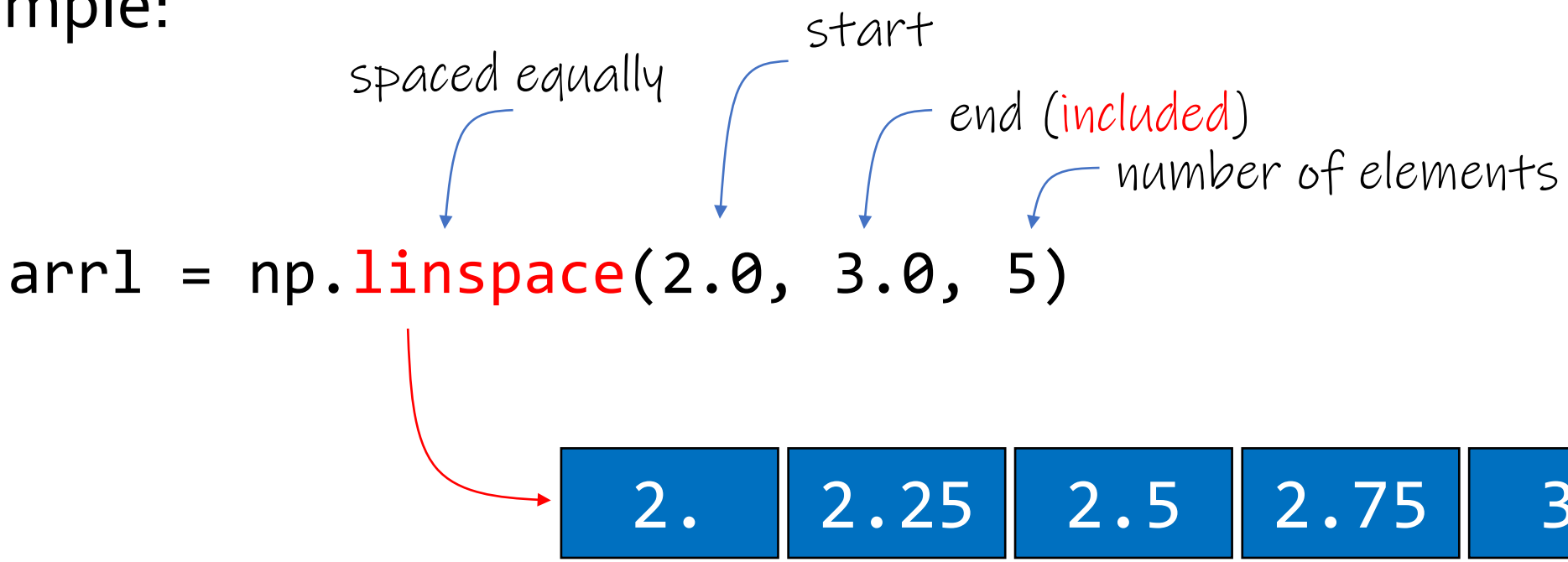
- Creating an array with uninitialized elements
- Example:

```
arrempty = np.empty((2,3))
```



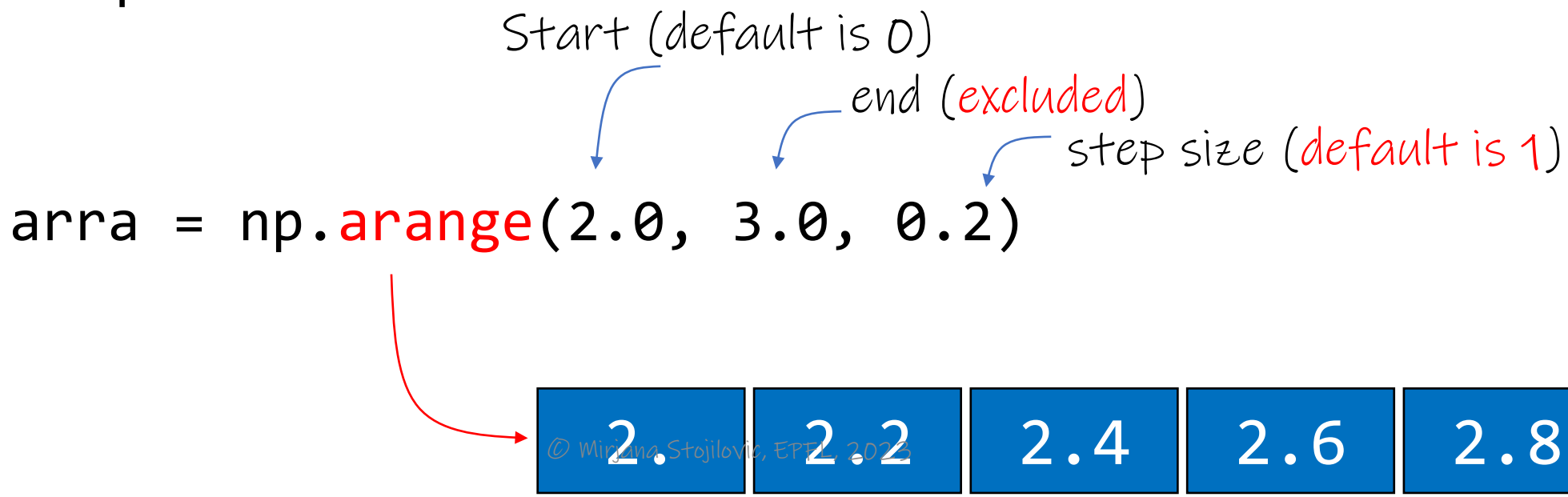
Intrinsic Array Creation: linspace()

- NumPy provides methods for quick creation of arrays
- Default data type: float (even if it may not seem needed)
- Example:



Intrinsic Array Creation: arange()

- arange() takes a starting value as the initial array element; it then adds the step size to it until (before) the end value is reached
- Unlike linspace, arange() can create an array of integers
- Example:



Intrinsic Array Creation: Random Module

- The **random module** in NumPy provides functions to
  - create random arrays
  - create random permutations of arrays
  - generate arrays with specific probability distributions
- Once NumPy is imported, random module is imported automatically too

Example: **rand function**

Intrinsic Array Creation: rand()

- **rand function:**
  - takes any number of integer arguments and
  - returns a random array such that
    - its dimensionality is equal to the number of integer arguments passed to the function, and
    - the respective dimensions have lengths equal to the values of the integer arguments
    - array elements are random samples from a uniform distribution over [0, 1).

Intrinsic Array Creation: rand()

```
arrand = np.random.rand(3)
```

one argument = one dimension
dimension size = 3
Note 8 decimal digits

0.60244244	0.08078644	0.81006613

```
arrand = np.random.rand(4,2)
```

two arguments = two dimensions
1st dimension size: 4
2nd dimension size: 2

0.74078305	0.60771826
0.98517758	0.65697074
0.72547252	0.71728332
0.28993407	0.98191922

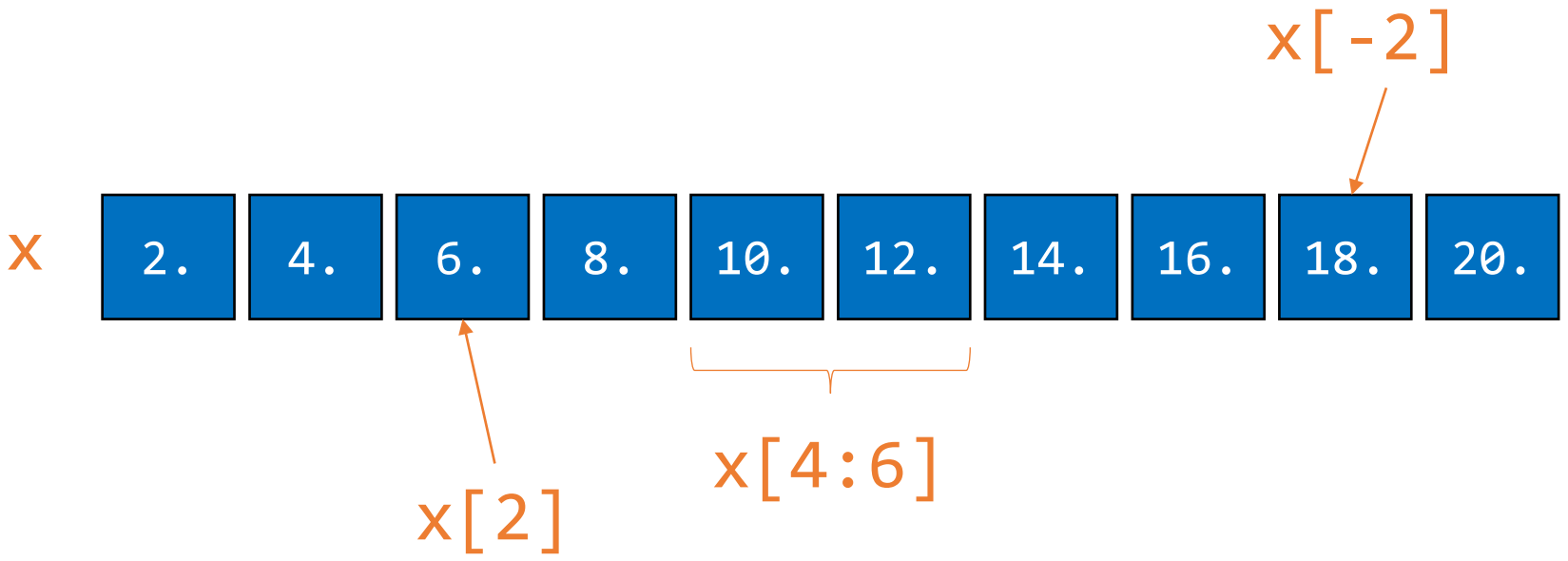
Array Indexing and Slicing

Extra reading: link

Indexing and Slicing

- Array indexing refers to any use of the square brackets [ ] to index array values
- Indexing
  - is used to reference data in an array or assign data to an array
  - zero-based (first index is zero)
  - accepts negative indices for indexing from the end of the array
- One can slice and stride arrays to extract parts of it

Indexing and Slicing

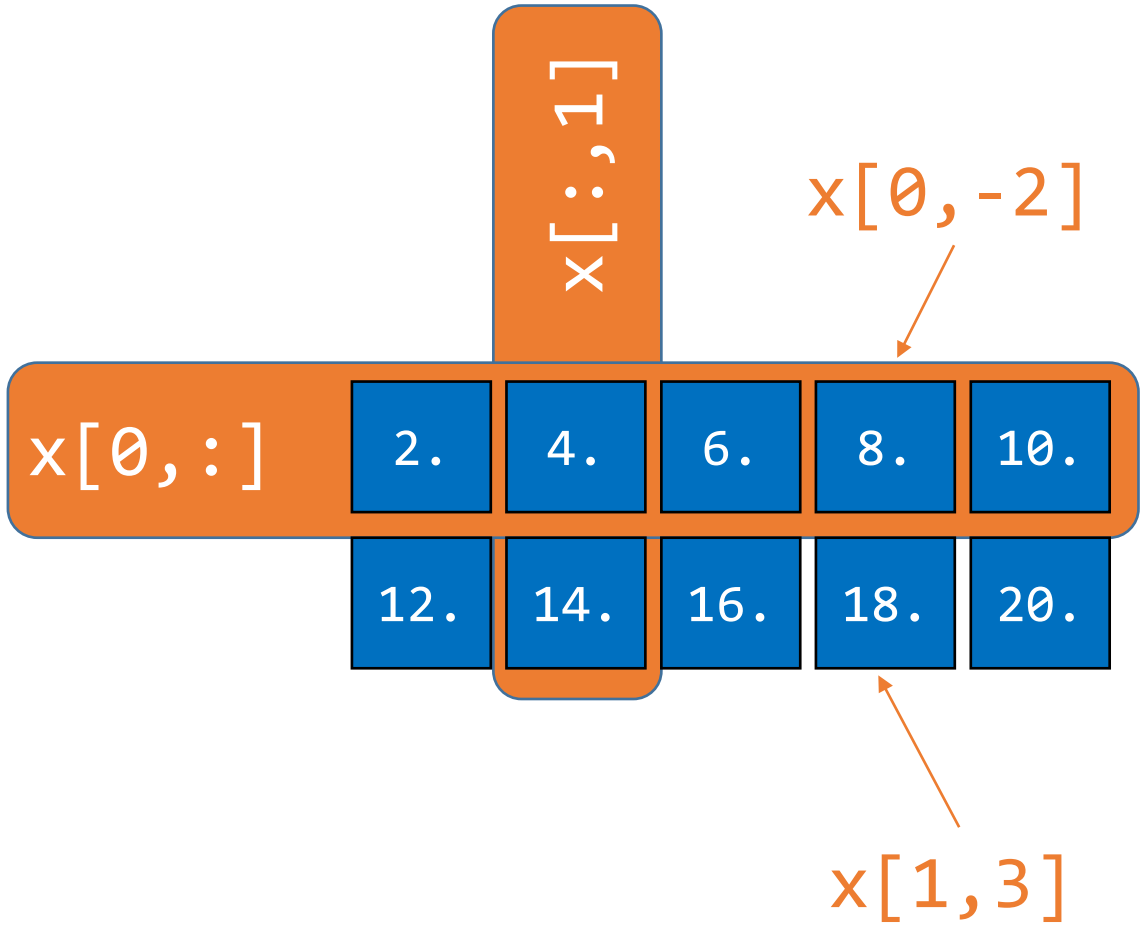


Note that slices of arrays do not copy the array data but only produce new views of the original data! This is different from list slicing.

```
x = np.arange(2.,22,2)
temp = x[4:6] # [10. 12.]
temp[1] = 7   # [10. 7.]
id(temp[1]) == id(x[5])) # True !!!
print(x)      # [2. 4. 6. 8. 10. 7. 14. 16. 18. 20.]
```

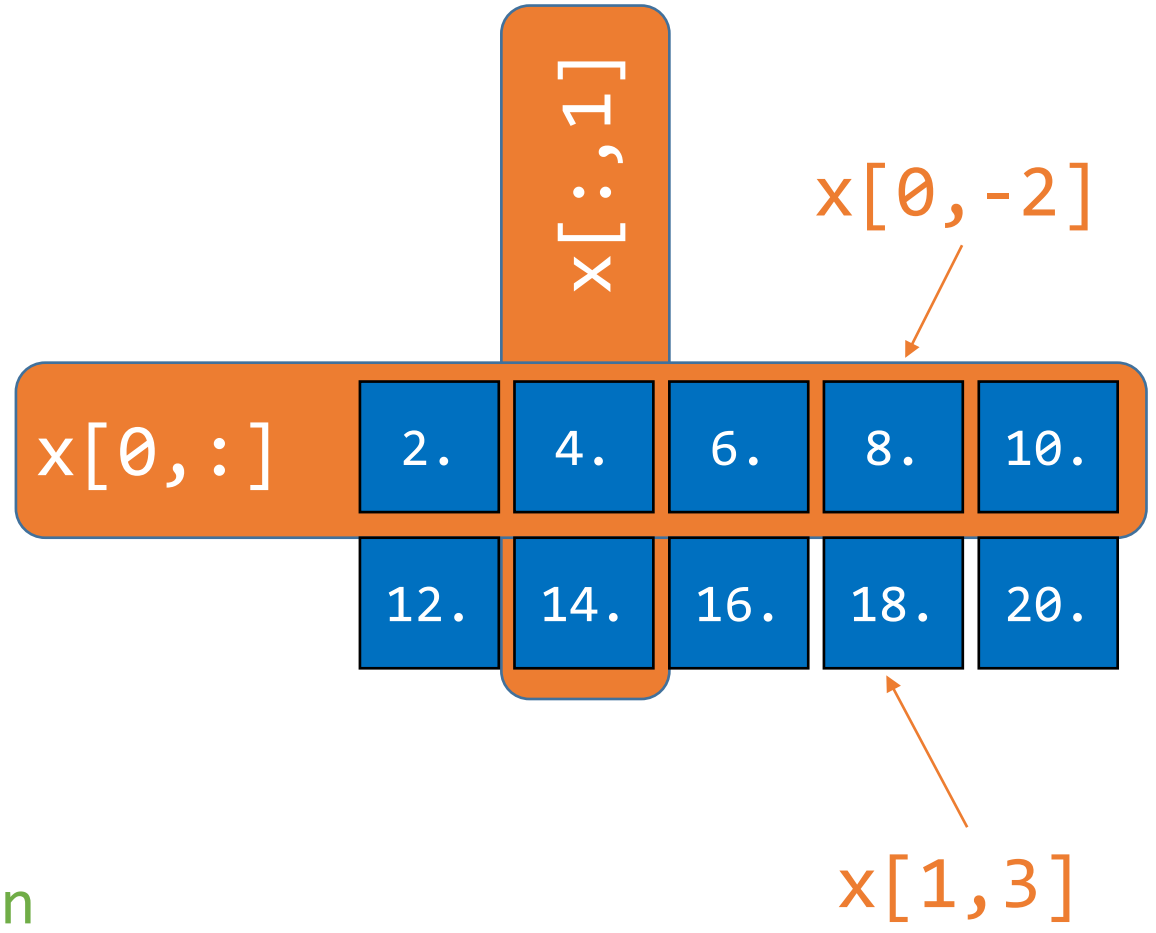
Reshaping Arrays

```
x.shape = (2,5)
# two-dim array
# 2 rows
# 5 columns
y = x
id(y) == id(x) # True

# Note:
x.shape = (3,5)
ValueError: cannot
reshape array of size
10 elements into
shape (3,5)
```



Indexing & Slicing Multi-Dimensional Arrays

```
y = x[1,3]
# row index: 1, # column index: 3
# returns 18.0
# indexing => copy is returned
id(y) == id(x[1,3]) # False

x[0,:]
# row: 0, column: ALL
# returns the entire first row
# slicing => a view is returned

x[:,1]
# row: ALL, column: 1
# returns the entire second column
# slicing => a view is returned
```

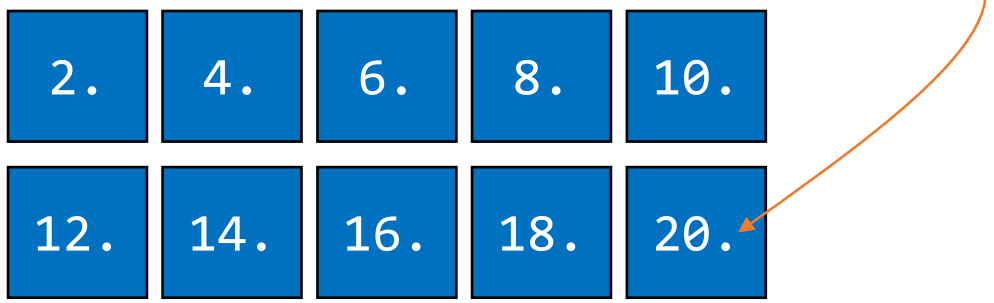


Advanced, Integer Array Indexing

- NumPy arrays may be indexed with other arrays
  - Index arrays must be of integer type
- A copy of the original data is returned, not a view

Advanced Indexing

```
x = np.arange(2.,22,2)
x.shape = (2,5)   # 2D array, 2 rows, 5 columns

x[[0,1],[2,4]]
# [0,1]: row indices
# [2,4]: column indices in the corresponding rows
# row 0, col 2
# row 1, col 4
# result: [6. 20.]

x[[0,1],[3]]
# row 0, col 3
# row 1, col 3
# result: [8. 18.]
```



View or a Copy?

- How to tell if the array is a view or a copy?
- An array has a base attribute
  - Base attribute of a view returns the original array
  - Base attribute of a copy returns None

```
x = np.arange(2.,22,2)
x.shape = (2,5) # 2D array, 2 rows, 5 columns

y = x[[0,1],[2,4]]
y.base == None  # True => y is a copy

z = x[1,0:3]    # [12. 14. 16.]
print(z.base)   # prints all elements of x => z is a view
```

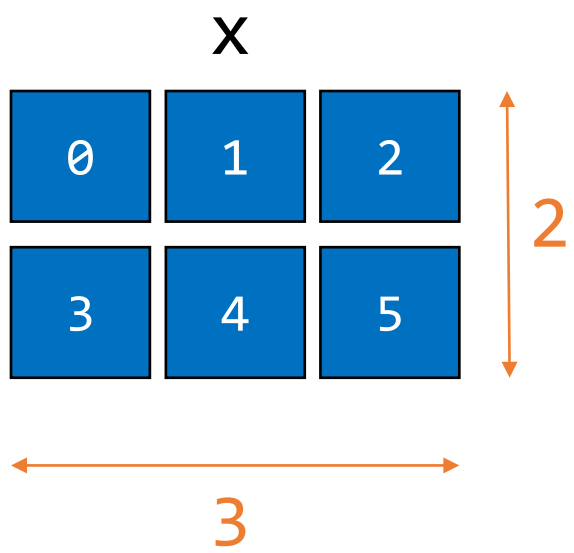
Iterating Over Arrays

Extra reading: link

Iterating Over Arrays: Reading

- Visiting every array element is simple using **nditer** object

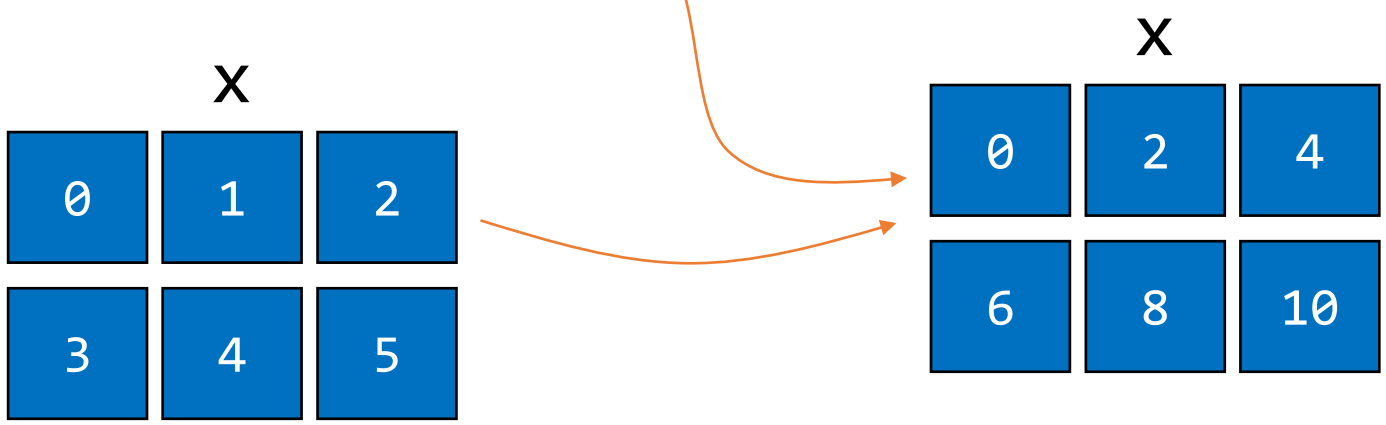
```
x = np.arange(6).reshape(2,3)
for i in np.nditer(x):
    print(i, end = ' ')
# will print all the array elements
# 0 1 2 3 4 5
```



Iterating Over Arrays: Writing

- By default, nditer treats the input operand (the array) as a read-only object. To modify array elements, you must specify read-write or write-only mode:

```
for i in np.nditer(x, op_flags = ['readwrite']):
    i[...] = i * 2
```

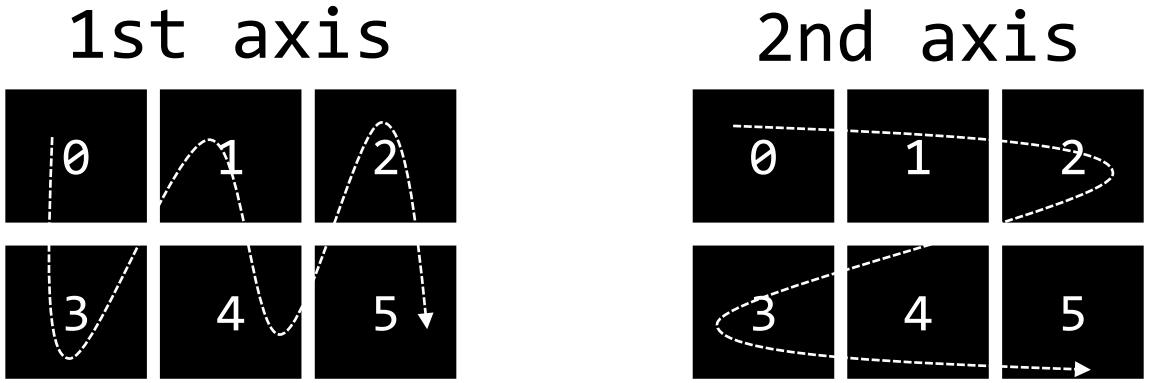


Array-Manipulation Routines

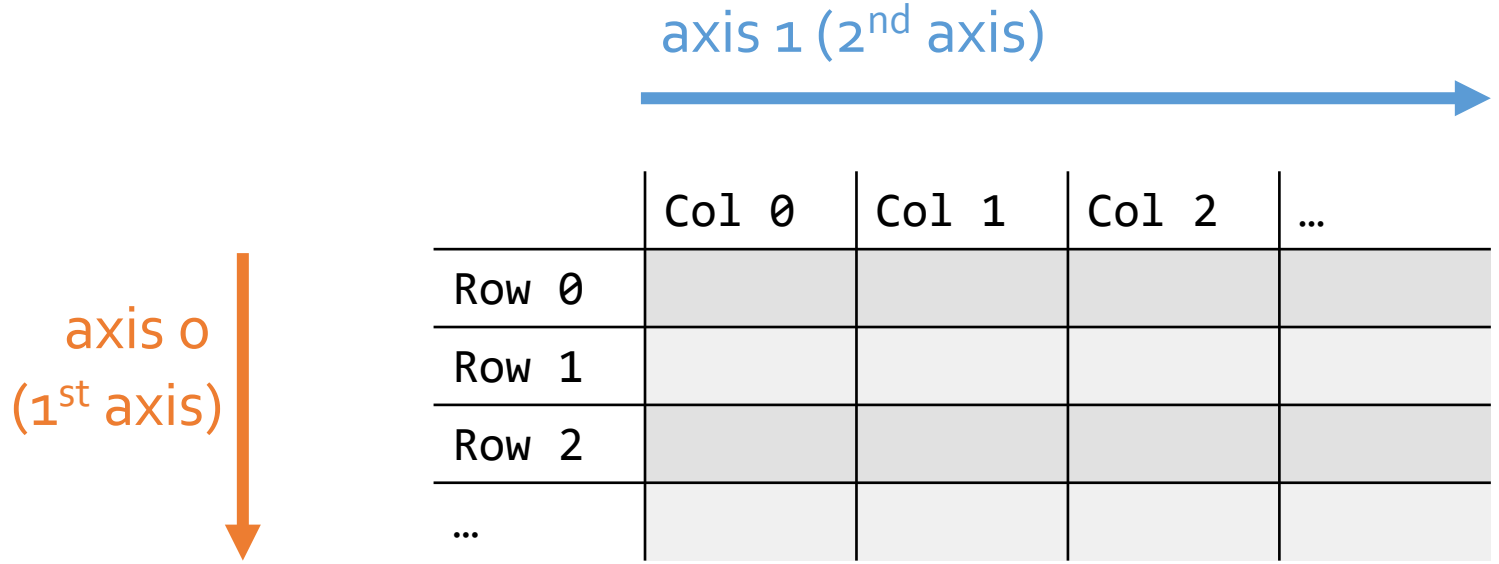
Extra reading: link

Transpose-Like Operations

- swapaxes(array,a1,a2): exchange axes a1 and a2
  - if *array* is an ndarray, then a view of *array* is returned
- Axes are defined for arrays with more than one dimension
- A two-dimensional array has two axes
  - axis zero (first axis): runs vertically downwards across rows
  - axis one (second axis): runs horizontally across columns



Another View of Axes

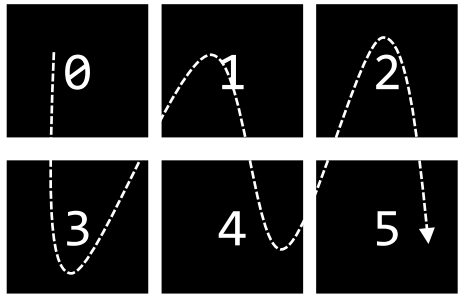


Transpose-Like Operations

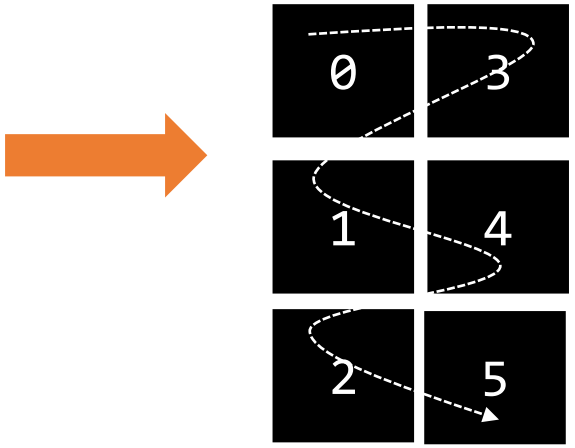
- swapaxes(array,a1,a2): exchange axes a1 and a2

```
x = np.arange(6).reshape(2,3)
np.swapaxes(x, 0, 1)
```

x



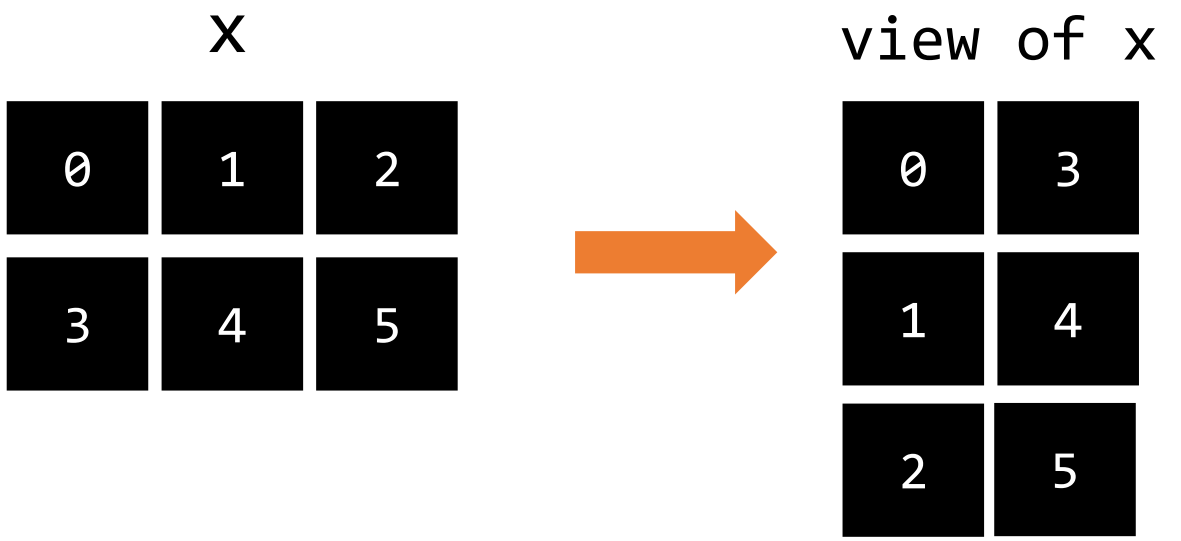
another view of x
(x is not modified!)



Transpose-Like Operations

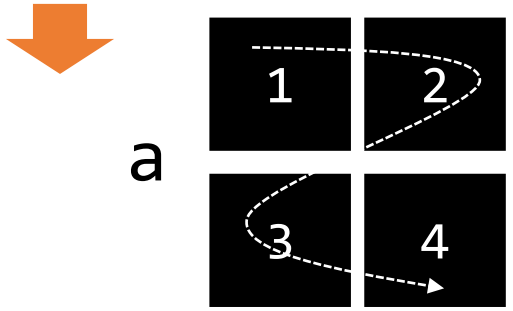
- transpose(array): permute the dimensions of an array
- another way to achieve the same: array.T

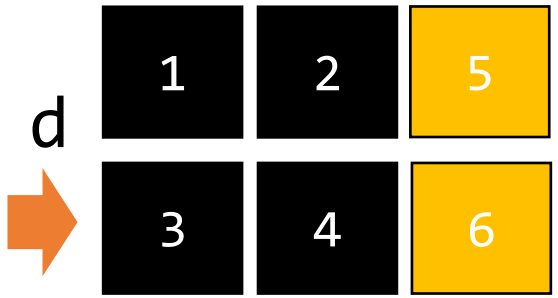
```
x = np.arange(6).reshape(2,3)
np.transpose(x)
```

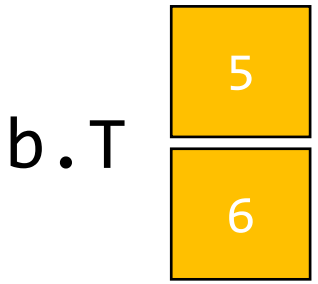


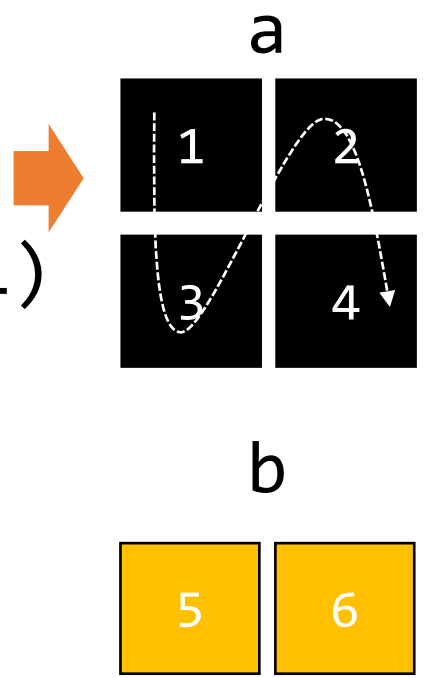
Joining Arrays

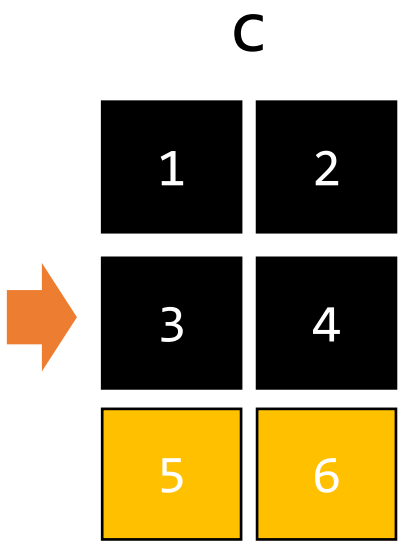
```
# join a sequence of arrays along an existing axis
a = np.array([[1,2], [3,4]])
b = np.array([[5,6]])
c = np.concatenate((a,b), axis=0)
d = np.concatenate((a,b.T), axis=1)
```





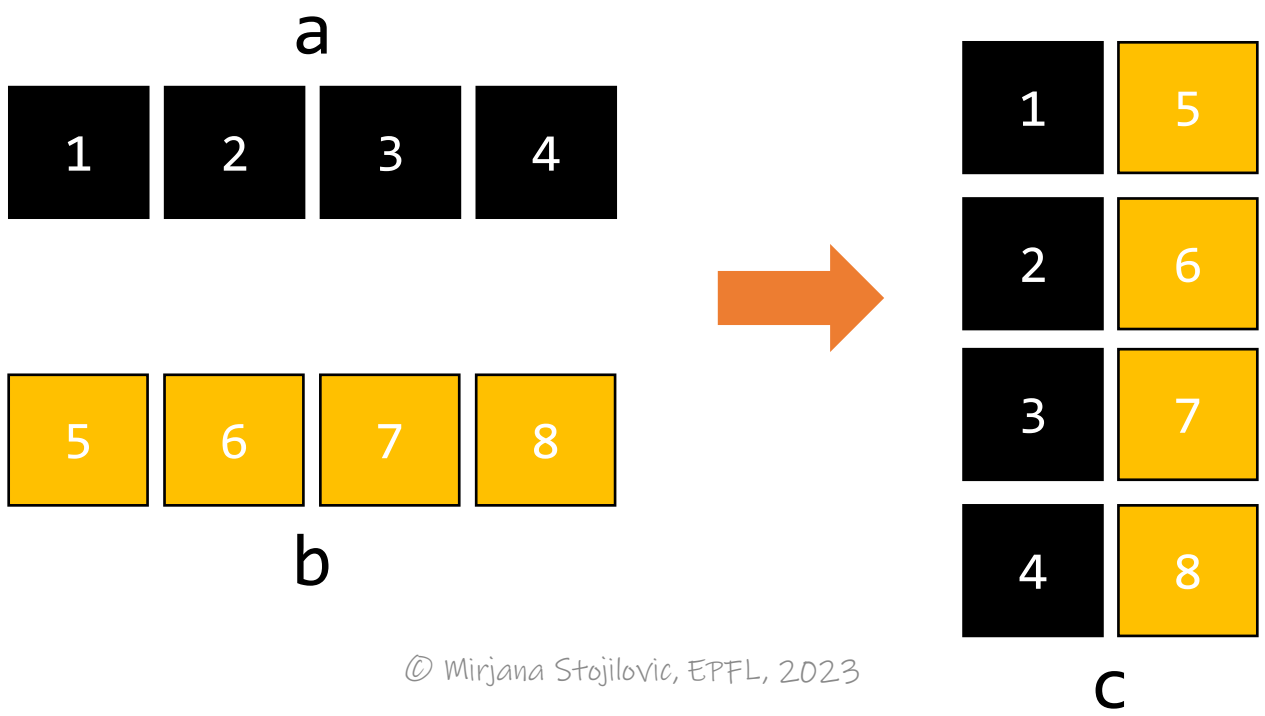






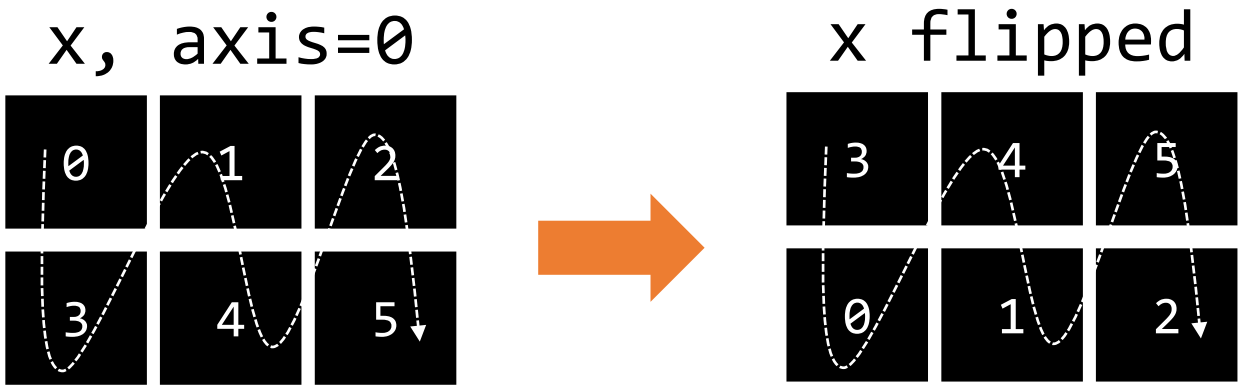
Joining Arrays

```
# stack 1-D arrays as columns into a 2-D array
a = np.array([1,2,3,4])
b = np.array([5,6,7,8])
c = np.column_stack((a,b))
```



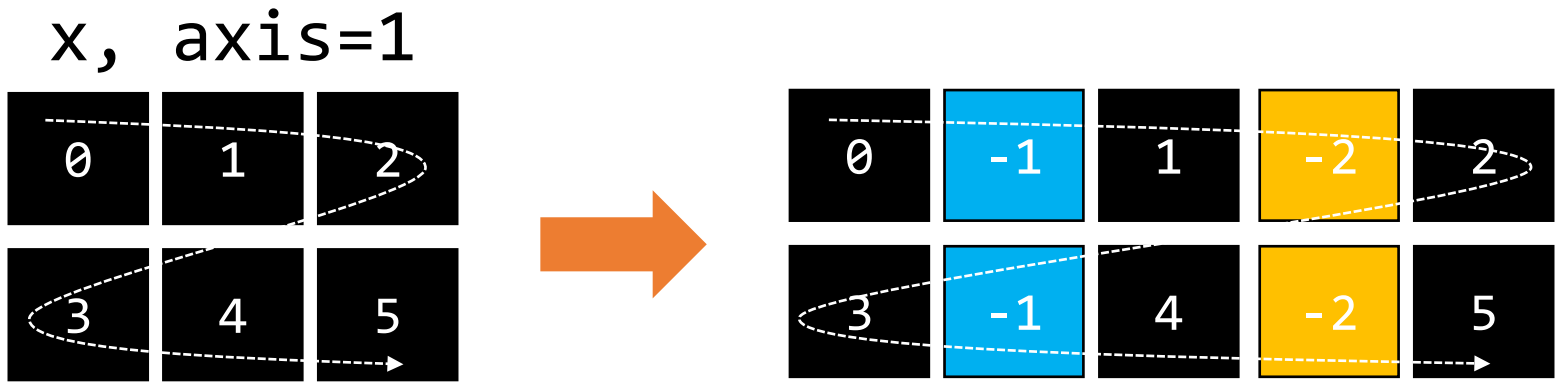
Rearranging elements: flip()

```
# reverse the order of elements in an array
# along the given axis
x = np.arange(6).reshape(2,3)
print(np.flip(x, axis = 0))
# flip w.r.t. the row axis
# does not change the array, only
# returns another view of it
```



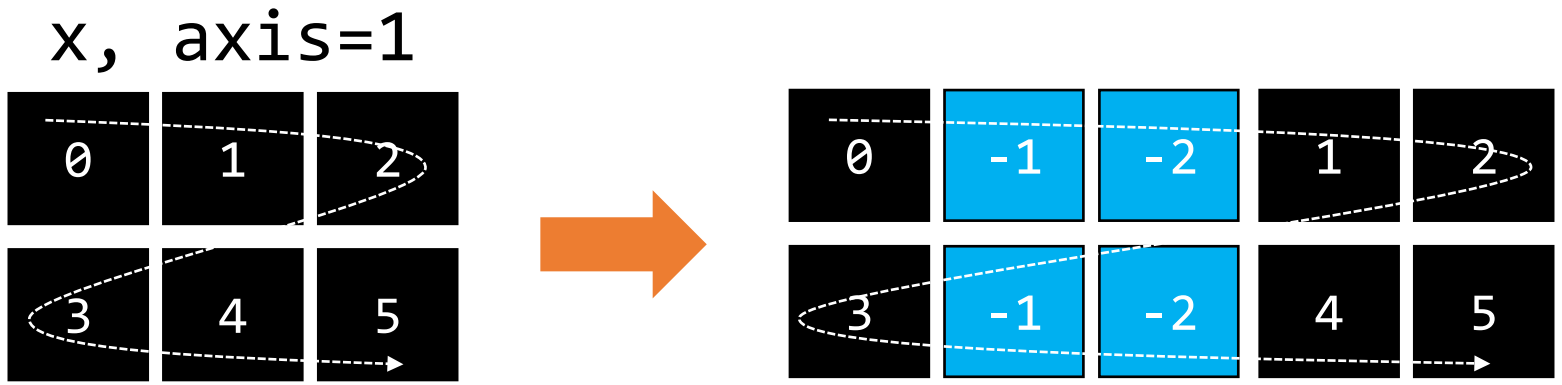
Adding Elements: insert()

```
# insert values along the given axis
# before the given indices
x = np.arange(6).reshape(2,3)
np.insert(x, [1, 2], [-1, -2], axis = 1)
# insert w.r.t. the column axis
```



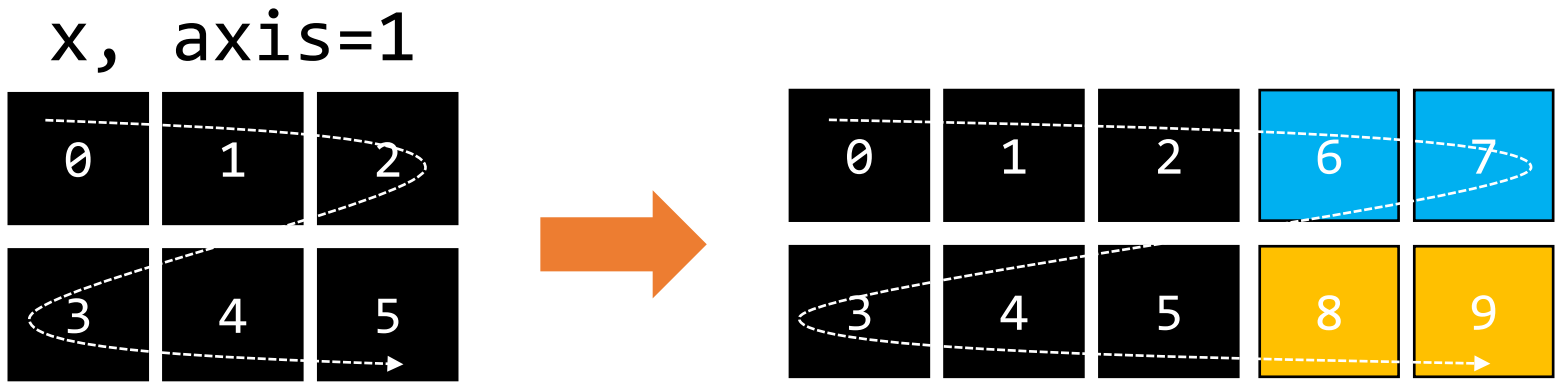
Adding Elements: insert()

```
# insert values along the given axis
# before the given indices
x = np.arange(6).reshape(2,3)
np.insert(x, [1], [-1, -2], axis = 1)
```



Adding Elements: append()

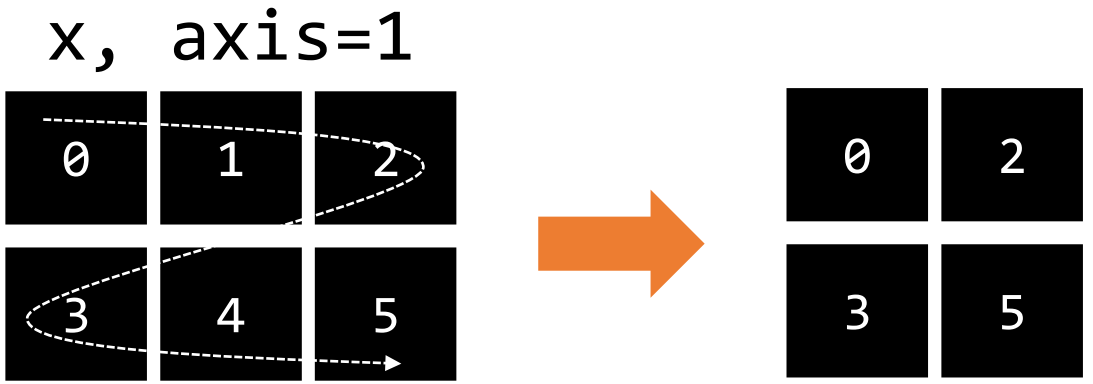
```
# append values to the end of an array
x = np.arange(6).reshape(2,3)
np.append(x, [[6, 7], [8, 9]], axis=1)
```



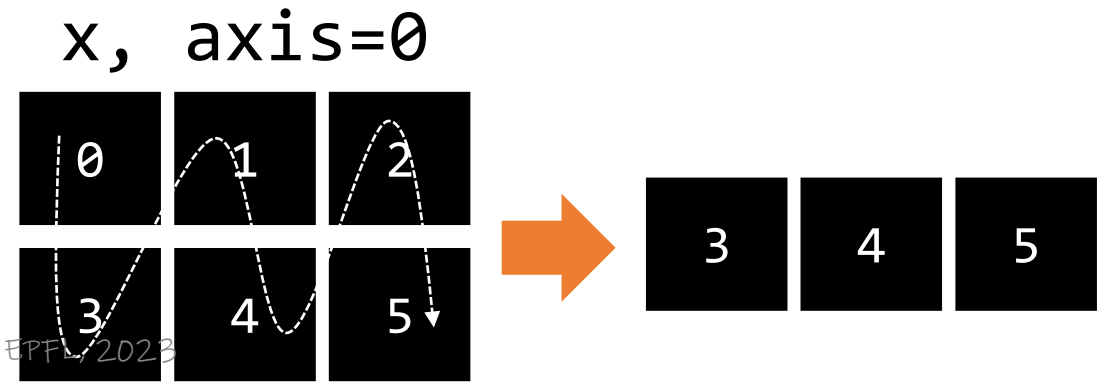
Adding Elements: delete()

```
# return a new array with sub-arrays
# along an axis deleted
x = np.arange(6).reshape(2,3)
```

```
np.delete(x, [1], axis=1)
```




```
np.delete(x, [0], axis=0)
```

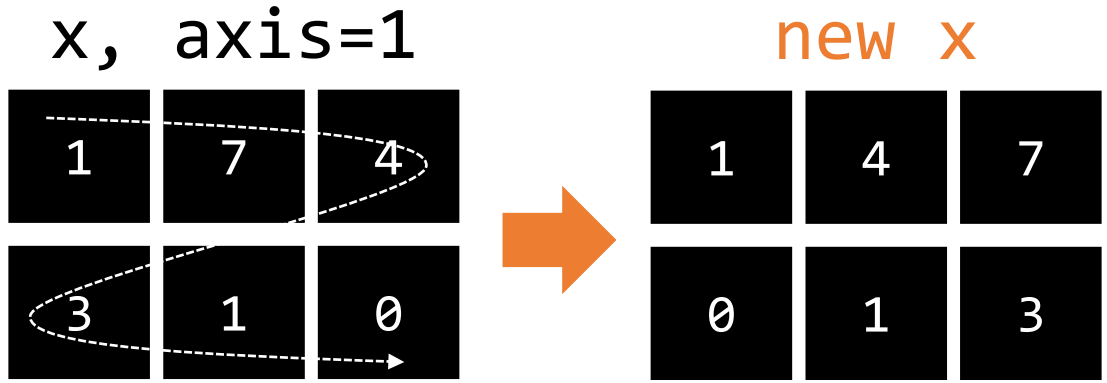


Array Methods

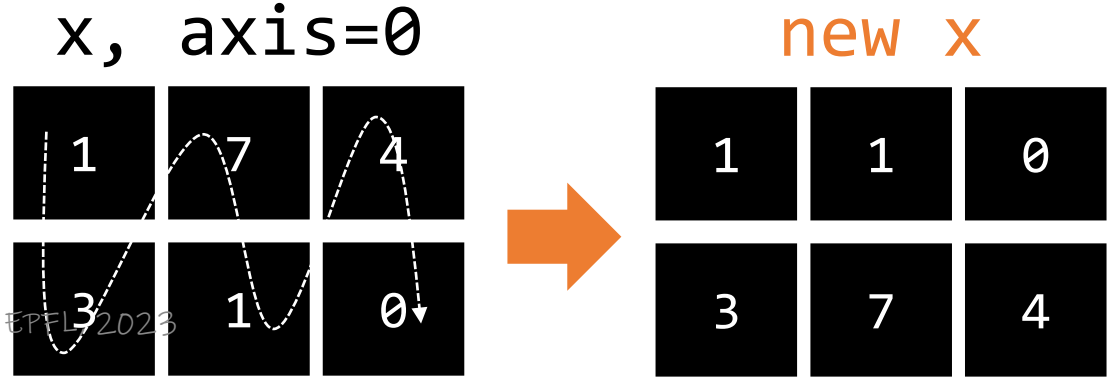
Extra reading: link

Sorting

```
# sort an array in place
# options: quicksort (default), mergesort, heapsort
x = np.array([[1,7,4],[3,1,0]])

x.sort(axis=1)
```



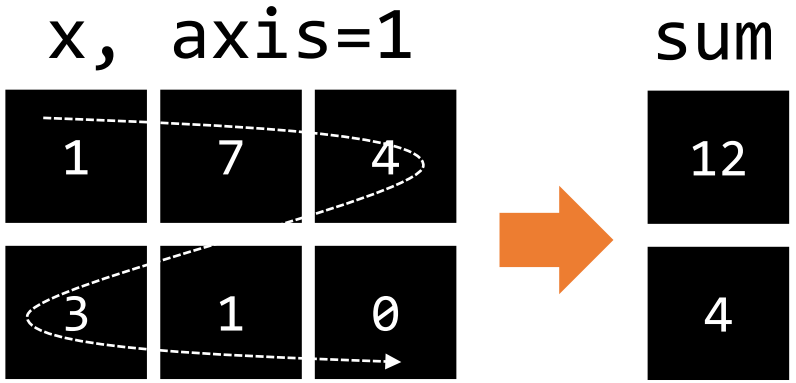

```
x.sort(axis=0, kind='quicksort')
```



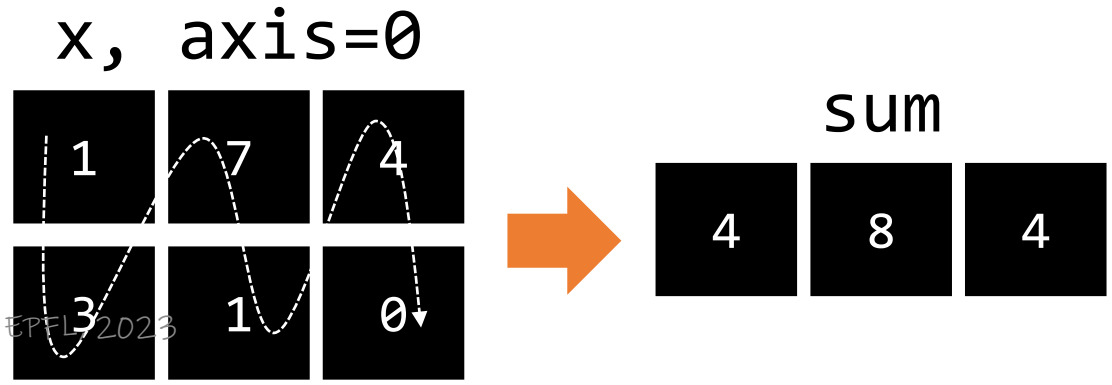
Sum

```
# return the sum of the elements over the given axis
x = np.array([[1,7,4],[3,1,0]])
```

```
np.sum(x, axis=1)
```

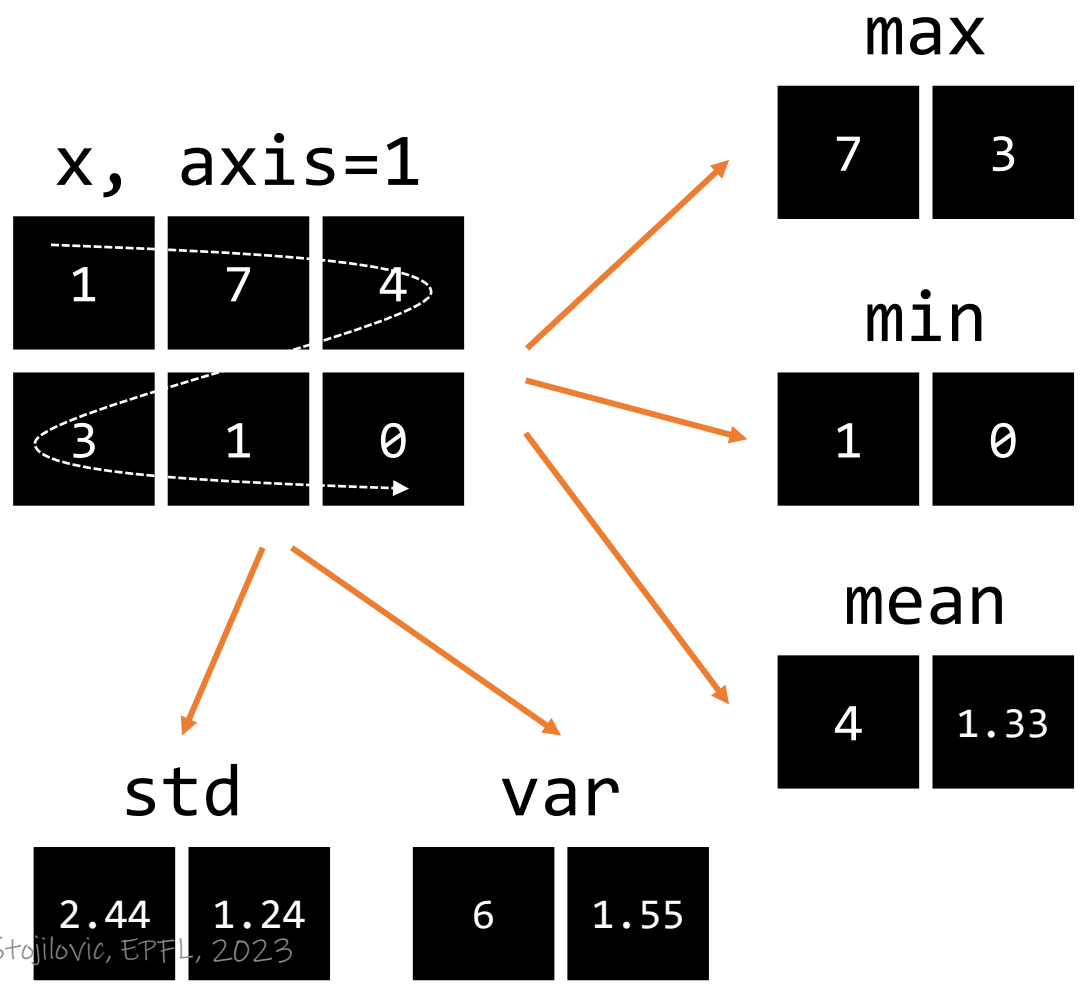



```
np.sum(x, axis=0)
```



Max, Min, Mean, Standard Deviation, Variance

```
# analyze the array along a given axis
x = np.array([[1,7,4],[3,1,0]])
np.max(x,axis=1)
np.min(x,axis=1)
np.mean(x,axis=1)
np.std(x,axis=1)
np.var(x,axis=1)
```



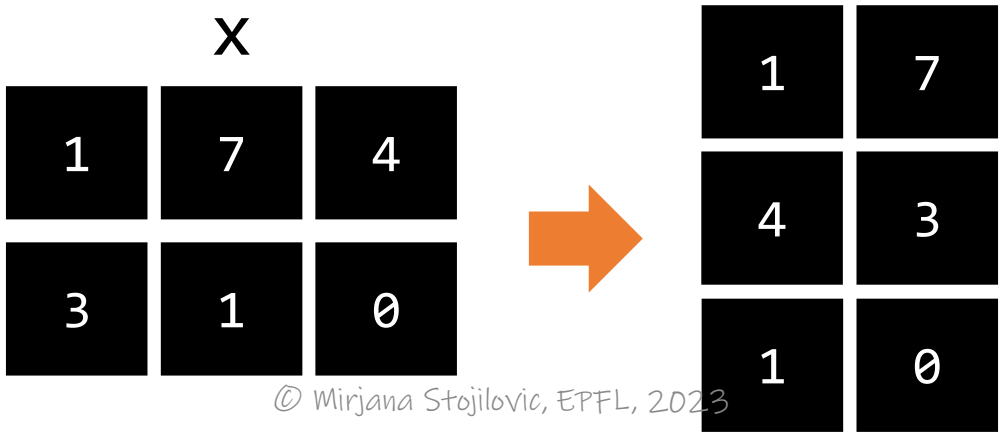
Array Reshape

```
# gives a new shape to an array
# without changing its data
x = np.array([[1,7,4],[3,1,0]])
np.reshape(x, 6)
# 1-D array of length 6
```

Array Reshape

```
# gives a new shape to an array
# without changing its data
x = np.array([[1,7,4],[3,1,0]])
np.reshape(x, (3,-1))
# 2-D array with three rows
```

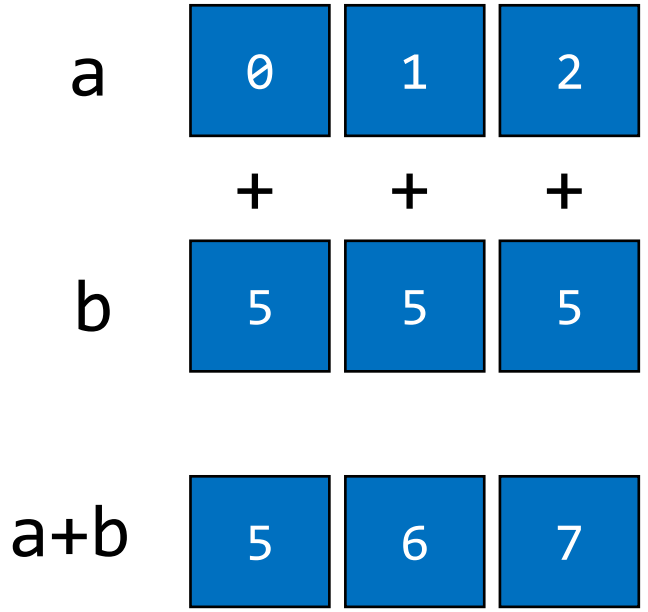
if dimension is -1, it's real value is inferred from the array length and remaining dimensions



Computation on Arrays: Broadcasting

```
# when adding two arrays of the same size
# addition is done on an element-by-element basis
a = np.array([0,1,2])
b = np.array([5,5,5])
a + b
```

What happens if the arrays are not of the same size and dimension?



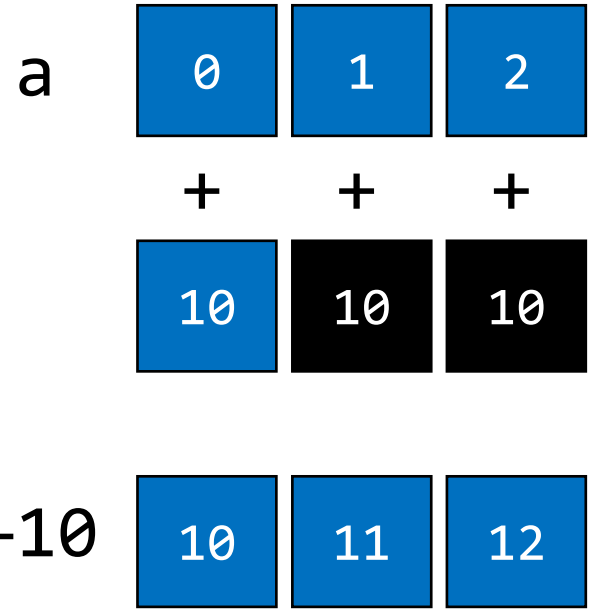
Broadcasting

- Broadcasting is a set of rules for applying binary functions (addition, subtraction, multiplication, etc.) on arrays of different sizes

- We can think of it as an operation that stretches or duplicates elements until arrays are of the same size and dimension

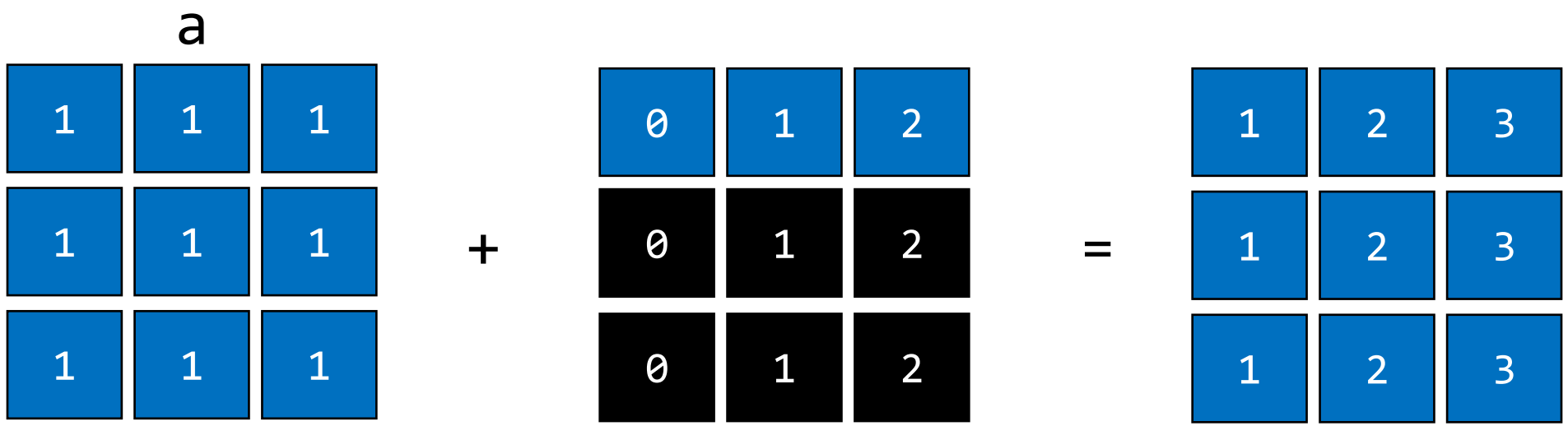
Broadcasting: Example

```
# when arrays are not of the same dimension
# broadcasting “fills in” the missing elements
a = np.array([0,1,2])
a + 10
```



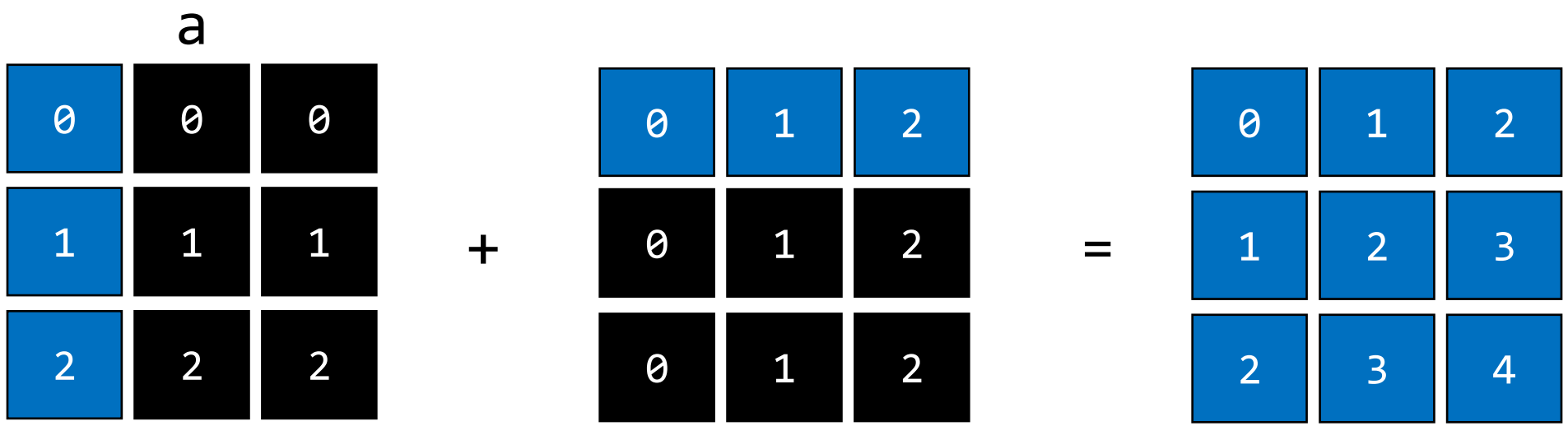
Broadcasting: Example

```
# when arrays are not of the same dimension
# broadcasting “fills in” the missing elements
a = np.ones((3,3))
a + np.arange(3)
```



Broadcasting: Example

```
# when arrays are not of the same dimension
# broadcasting “fills in” the missing elements
a = np.arange(3).reshape((3,1))
a + np.arange(3)
```



For more information on NumPy, refer to online resources: link

NumPy cheatsheet: link