Rotman

NUMPY

February 15, 2022 Prepared by Niti / Zissis
TDMDAL & FinHUB

Rotman School of Management
UNIVERSITY OF TORONTO

Why NumPy?

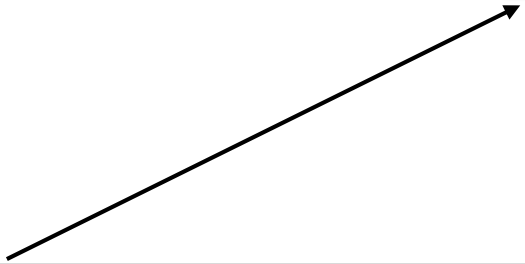
- It supplies an efficient N-dimensional array structure (ndarray)
- An ndarray is a grid of values of the same data type.
- Simplified data storage process and faster numerical operations.
- NumPy arrays form the core of nearly the entire ecosystem of data science tools in Python, such as
  - Pandas,
  - Scikit Learn
  - and more …

Rotman

ndarray: import

Before using ndarrays we need to import the NumPy module

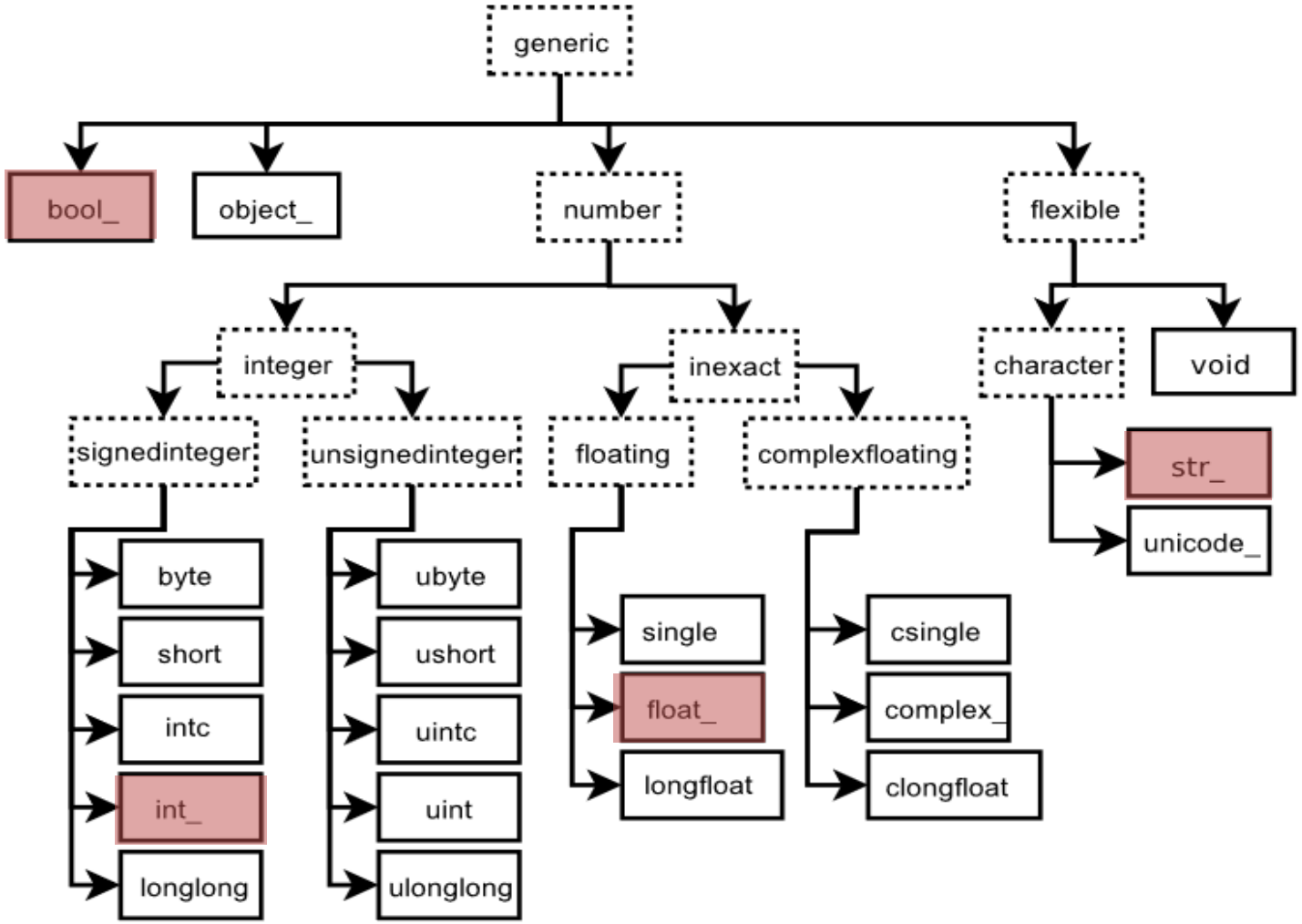
```
import numpy as np
```



import directives are used for importing a module and its methods to the working environment

an alias such as np can be given to avoid having to type the name of the module every time we want to call a method available in it

ndarray: dtype



Broad range of data types supported by NumPy

Most common:
int, float, bool, str

Datetime objects also supported

Rotman

ndarray: creation

One way to create an ndarray is through conversion from a list.
Several other ways are possible as we will later see (combined with initialization).

```
np.array([1, 2, 3])

Out: array([1, 2, 3])
```

```
np.array(['AAPL', 'GOOG', 'AMZN'])

Out: array(['AAPL', 'GOOG', 'AMZN'])
```

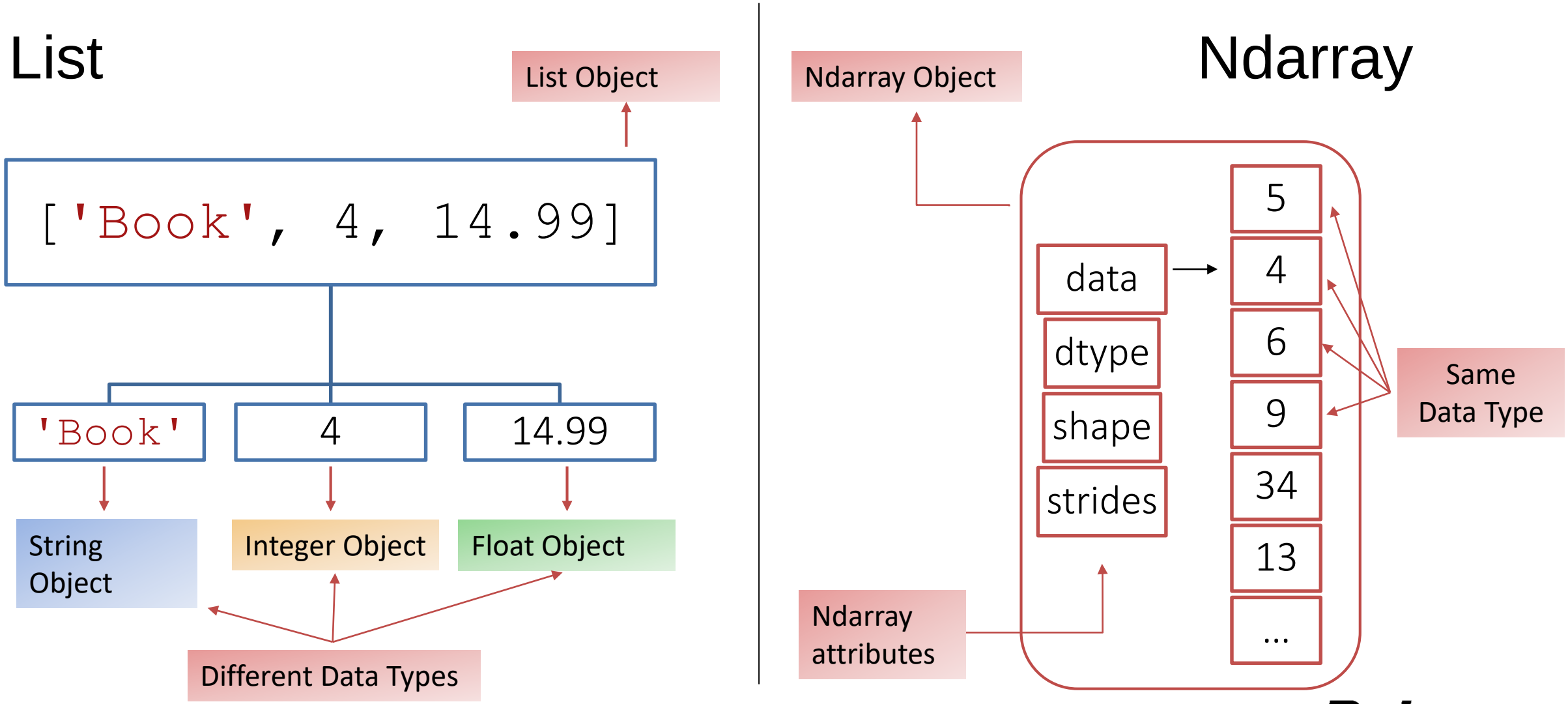
List vs ndarray

List vs Ndarray

List	Ndarray
Heterogeneous: elements can be of different data type	Homogeneous: elements are of the same data type
Memory inefficient	Memory efficient
Slow numerical manipulations	Fast numerical manipulations
Easy to expand or shrink	Cumbersome to expand or shrink
No need to declare a list before using it	Need to declare an ndarray before using it

- With ndarrays you trade flexibility for memory and operation efficiency.
- Ndarrays are at the core of scientific libraries.
- Ndarrays are tailored for matrix-matrix and matrix-vector multiplications.

List vs Ndarray Visual Comparison



Rotman

List vs Ndarray Code Example

An example of adding 2 to each element of:

a list

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

```
new_A = []
for i in A:
    new_A.append(i+2)
```

an ndarray

```
A = np.array([1, 2, 3])
```

```
doubleA = np.add(A,2)
```

List vs Ndarray Code Example

An example of adding 2 to each element of:

a list

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

```
new_A = []
for i in A:
    new_A.append(i+2)
```

an ndarray

```
A = np.array([1, 2, 3])
```

```
doubleA = np.add(A,2)
```

List vs Ndarray Code Example

An example of adding 2 to each element of:

a list

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

```
new_A = []
for i in A:
    new_A.append(i+2)
```

an ndarray

```
A = np.array([1, 2, 3])
```

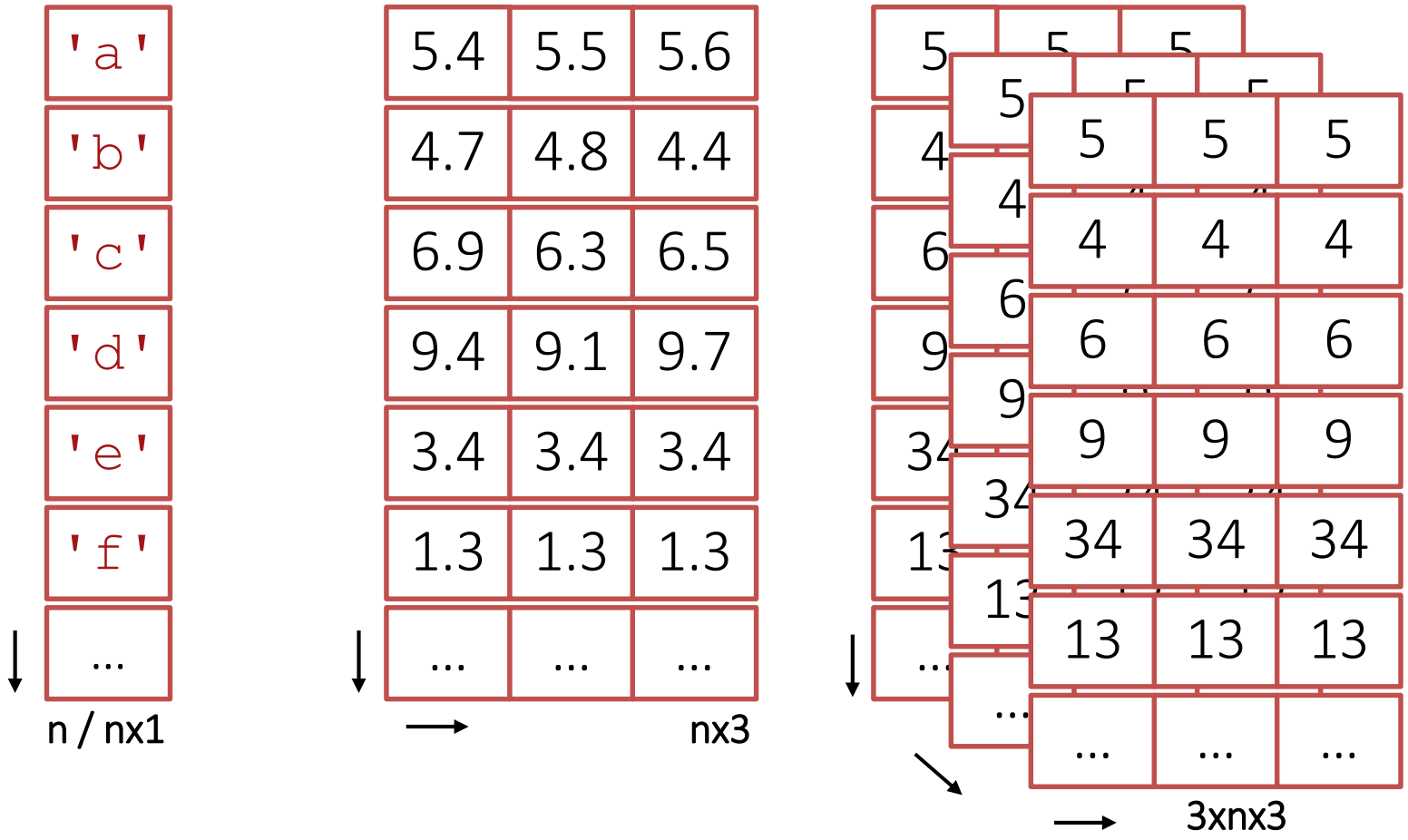
```
doubleA = np.add(A,2)
```

Numerical operations are made easier with NumPy with in-built methods. Often in a single line of code.

Rotman

Shape of ndarray

ndarray: shape

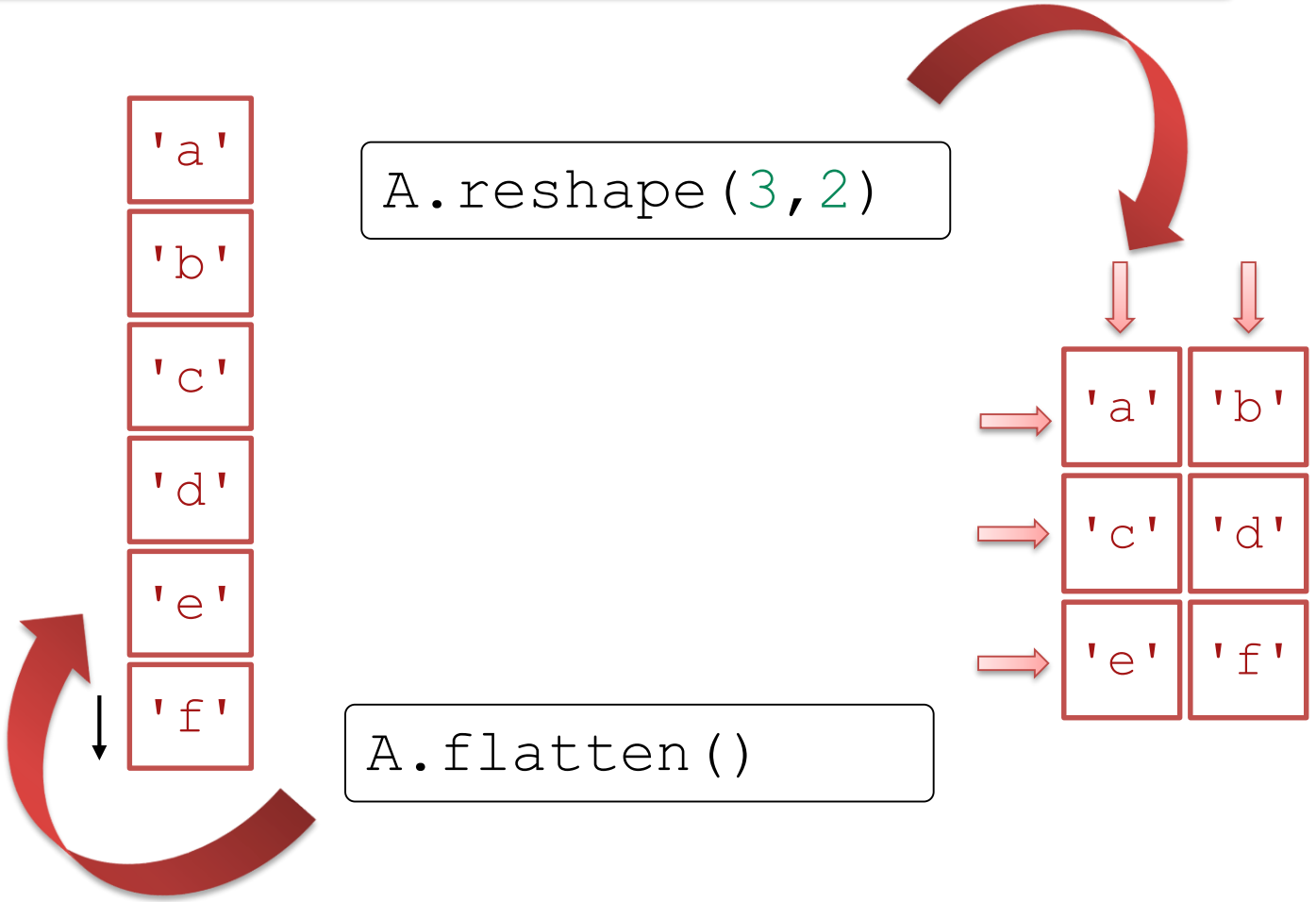


- Can be even more dimensions!
- No specific limit in how many dimensions. The only limit is the memory that you can allocate.

Rotman

ndarray: reshape

```
A = np.array(['a','b','c','d','e','f'])
```



other reshape methods:

- concatenate()
- hstack()
- vstack()
- transpose()

Rotman

Numerical Operations on ndarray

ndarray: numerical operations

x	y
0	5
1	4
2	3
3	2
4	1
5	0

Arithmetic

x/3	x+y
0	5
0.33	5
0.67	5
1.0	5
1.33	5
1.67	5

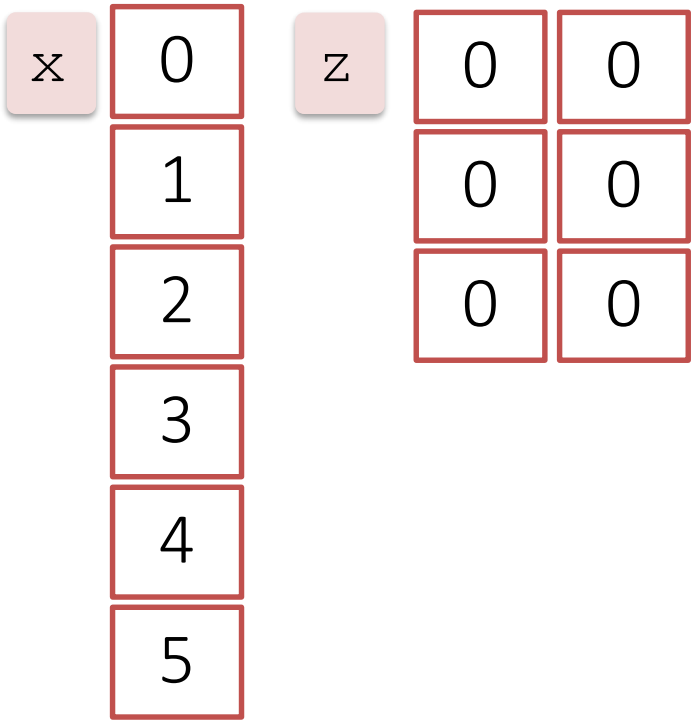
Conditional

x>=3	x<y
False	True
False	True
False	True
True	False
True	False
True	False

Rotman

ndarray: numerical operations

What if you want to perform an arithmetic operation on arrays with different shapes?



x+z

not possible!

operations are done element-wise

dimensions must match!

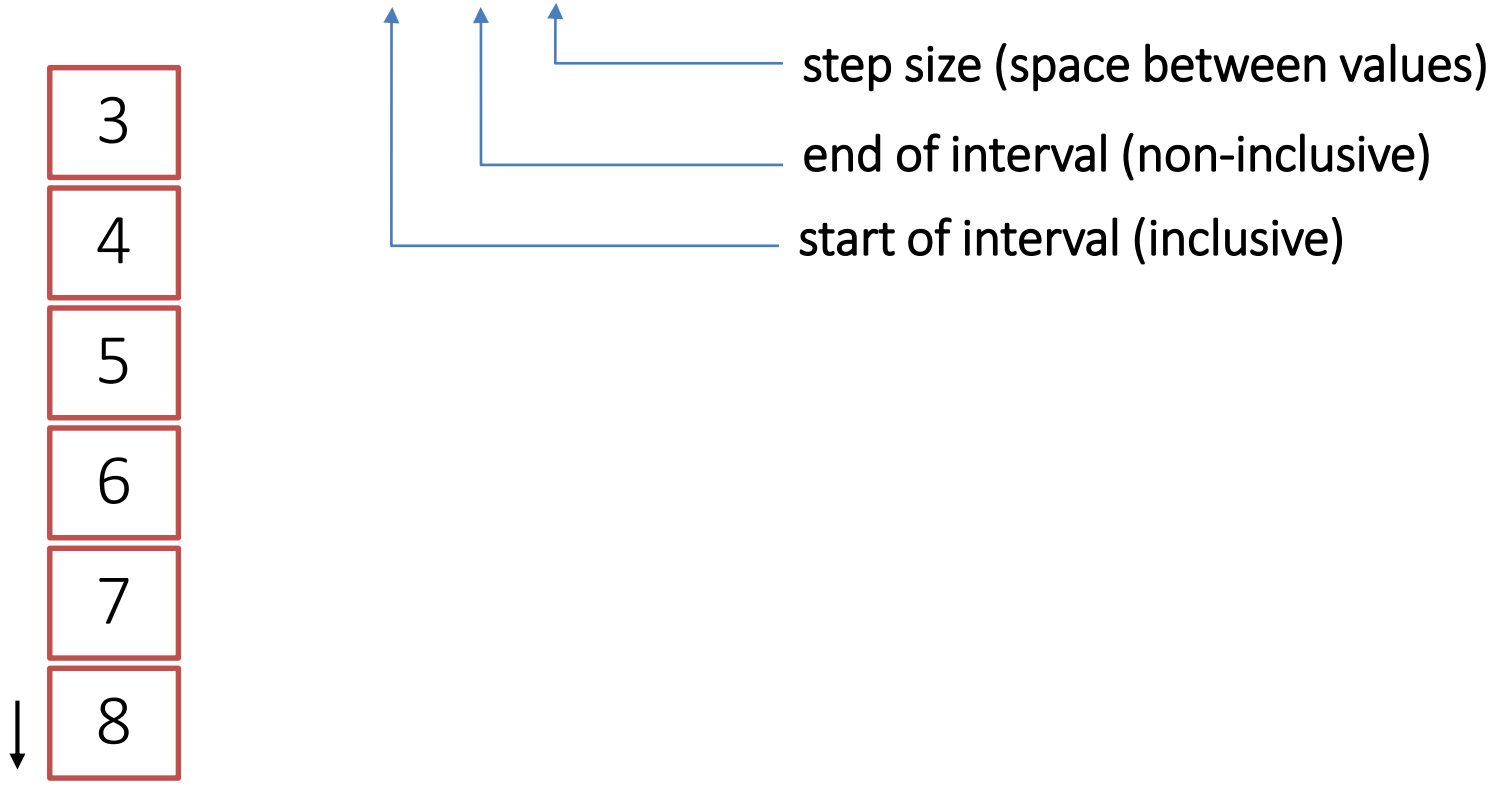
we must reshape the array to perform this operation

Rotman

Initializing ndarray

ndarray: initialization

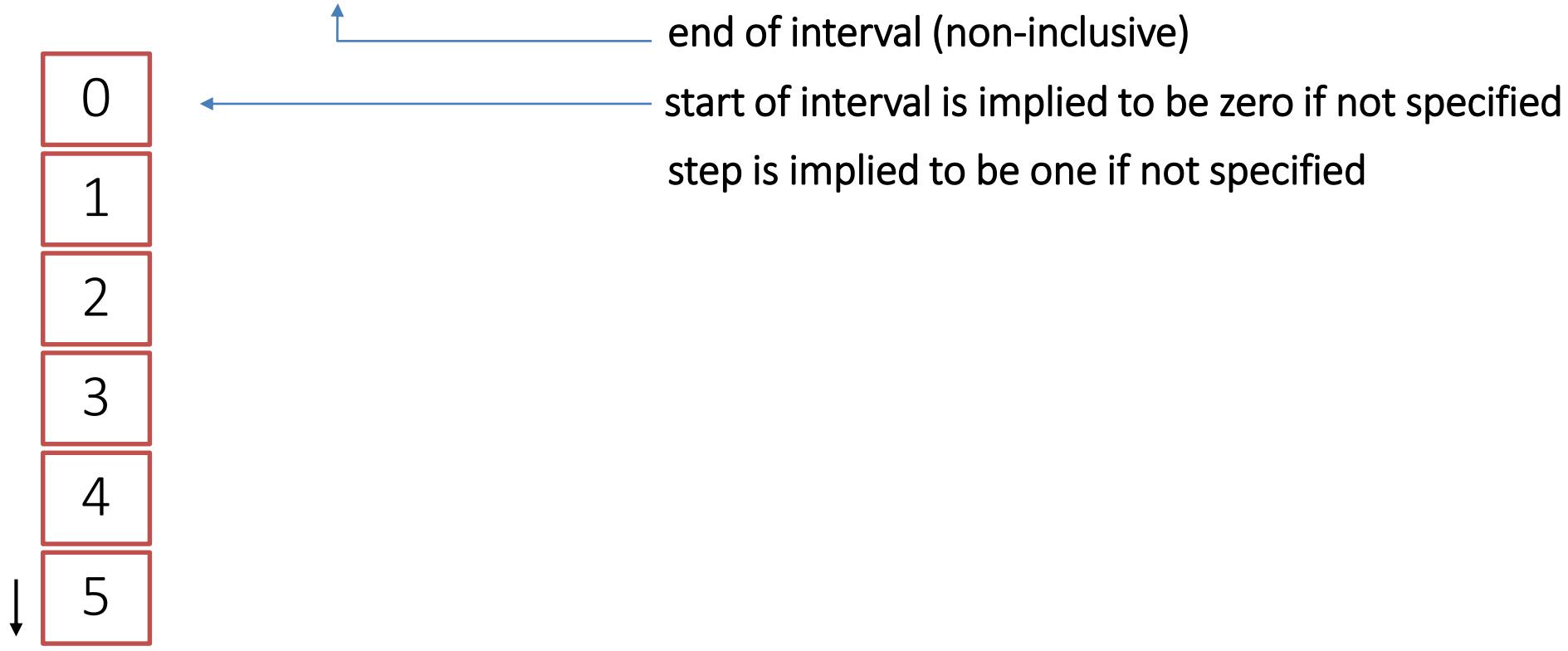
```
x = np.arange(3,9,1)
```



Rotman

ndarray: initialization

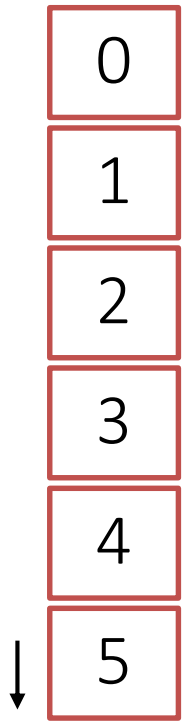
```
x = np.arange(6)
```



Rotman

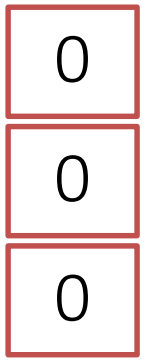
ndarray: initialization

```
x = np.arange(6)
```

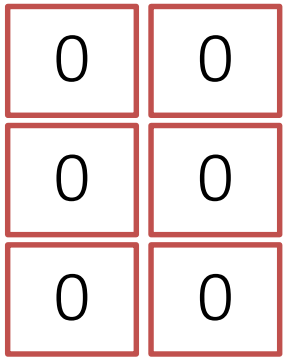


can also initialize with zero values and specify different shapes

```
x = np.zeros(3)
```




```
x = np.zeros((3,2))
```



Rotman

ndarray: initialization

Several other methods exist:

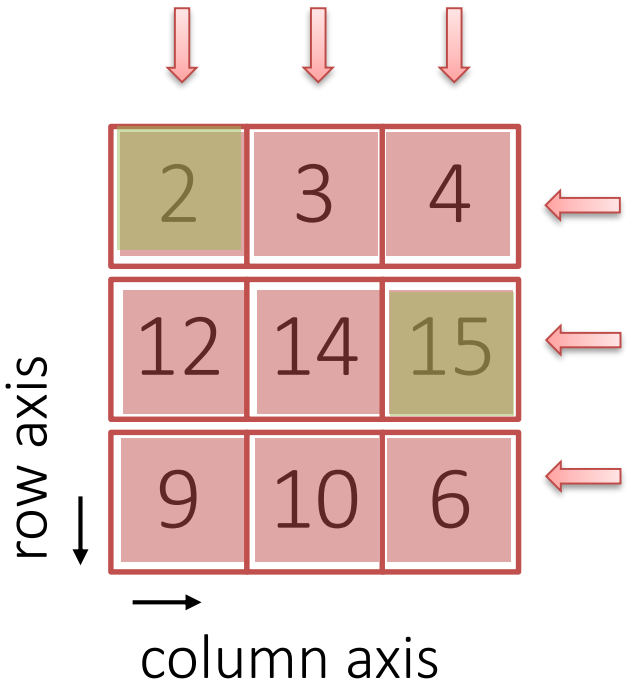
- np.full ➡ returns an array with a specified value for each element
- np.random.rand ➡ returns an array with random values
- np.ones ➡ returns an array with all values equal to one
- np.eye ➡ returns a 2-D diagonal array (ones on diagonal, zeros elsewhere)

Like before, the shape of the returned array can be specified when calling the method

Common Methods

ndarray: methods

```
B = np.array([(2,3,4),
              (12,14,15),
              (9,10,6)])
```




```
print(B.max())
→ 15
```

```
print(B.min())
→ 2
```

```
print(B.mean())
→ 8.33
```

Statistical methods are made easy with NumPy

Can be used over entire array or per dimension

Full guide:
https://numpy.org/doc/stable/reference/routines.statistics.html
https://numpy.org/doc/stable/reference/routines.math.html

Rotman

ndarray: methods

```
B = np.array([(2,3,4),
              (12,14,15),
              (9,10,6)])
```

Compute per row

```
B.sum(axis=1)
```

Compute per column

```
B.sum(axis=0)
```

2	3	4	9
12	14	15	41
9	10	6	25
23	27	25	

ndarray: methods (more advanced)

```
C = np.where(B>10,-3,B**2)
```

return an array C with elements from: -3 when B>10
B**2 when B≤10

2	3	4
12	14	15
9	10	6

Array B

4	9	16
-3	-3	-3
81	100	36

Array C

ndarray: methods (more advanced)

```
B = np.array([5,0,10,-2,7,8])

B.sort()

print(B)
➢ array([-2,0,5,7,8,10])
```

in-place sorting:
B is sorted and remains sorted from that point onwards

ndarray: methods (more advanced)

```
B = np.array([5,0,10,-2,7,8])

C = np.sort(B)

print(C)
➢ array([-2,0,5,7,8,10])

print(B)
➢ array([5,0,10,-2,7,8]))
```

Not in-place sorting:

- C is created as a sorted version of B
- B remains unchanged

Sort() does not take a direction argument. It always sorts in non-decreasing order.

But you can get the reverse order by just reversing the output. We cover this in the next section (Indexing)

Some useful packages

numpy.linalg for basic linear algebra:

```
from numpy import linalg
```

numpy.polynomial.polynomial for basic curve fitting and arithmetic with polynomials:

```
from numpy.polynomial.polynomial import polyfit
from numpy.polynomial.polynomial import polyval
```

→ Least squares fit of polynomial to data

→ Evaluate polynomial at different points

Links

https://numpy.org/doc/stable/reference/routines.linalg.html

https://numpy.org/doc/stable/reference/routines.polynomials.polynomial.html

Indexing ndarray

ndarray: indexing, slicing, stepping

NumPy follows the same convention to indexing, slicing and stepping as Python lists.

```
A = np.array(['a','b','c','d','e','f'])
```

Indexing

```
A[0]
```

```
'a'
```

Slicing

```
A[1:4])
```

```
array(['b','c','d'])
```

Stepping

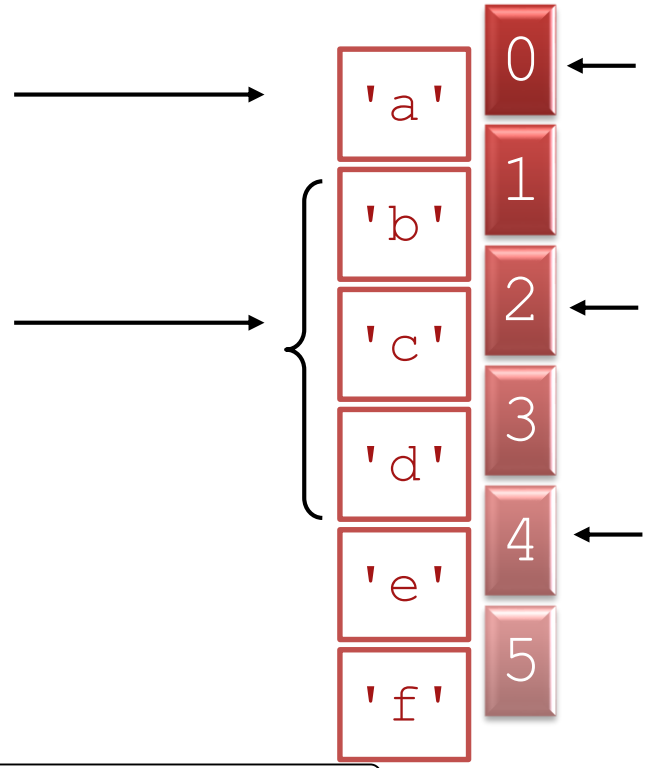
```
A[::2])
```

```
array(['a','c','e'])
```

Negative Indexing

```
A[::-1])
```

```
array(['f','e','d','c','b','a'])
```



Rotman

ndarray: indexing, slicing, stepping

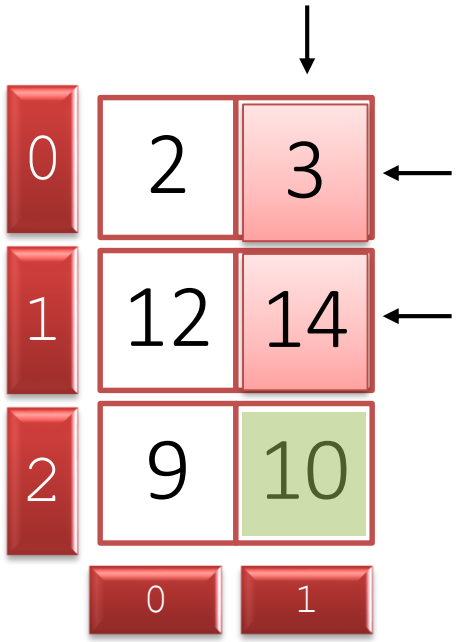
For arrays with 2 or more dimensions:

- comma separates each dimension
- colon (:) selects everything in that dimension

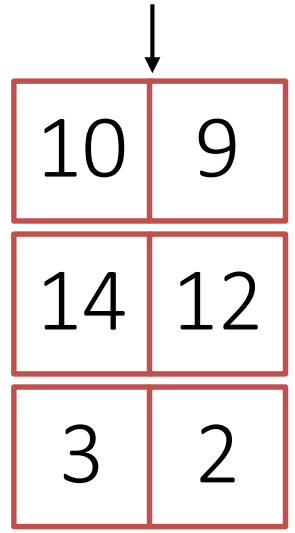
```
B = np.array([(2,3),
              (12,14),
              (9,10)])
```

```
print(B[2,1])
```

```
print(B[:2,1])
```




```
print(B[::-1,::-1])
```



Rotman

ndarray: indexing, slicing, stepping

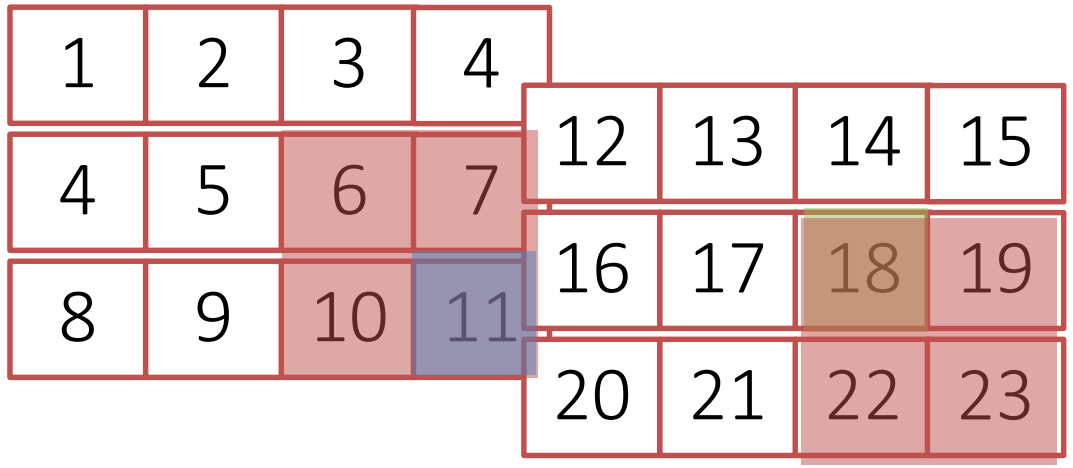
```
C = np.array([
      [[1,2,3,4],
       [4,5,6,7],
       [8,9,10,11]],

      [[12,13,14,15],
       [16,17,18,19],
       [20,21,22,23]]
    ])
```

```
print(C[0,2,-1])
```

```
print(C[1,1,2])
```

```
print(C[:,1:3,2:])
```



shape (2,3,4)

Rotman

Broadcasting

Broadcasting

- It is possible to do operations on arrays of **different shapes and dimensions** if NumPy can transform these arrays so that they all have the **same size.**
- The mechanism behind this is called **broadcasting.**
- There are 3 cases depending on the shape of the arrays.

Broadcasting: Case 1

CASE 1: If the two arrays differ in their number of dimensions, the shape of the one with fewer dimensions is *padded* with ones on its leading (left) side.

Existing Inventory

	Erasers	Books	Pens
Basket 1	0	0	0
Basket 2	10	10	10
Basket 3	20	20	20
Basket 4	30	30	30

(4,3)

+

You get a delivery:
no more erasers,
1 book for each basket and
2 pens for each basket

0	1	2
0	1	2
0	1	2
0	1	2

(3,) → (1,3) → (4,3)
Case 1 Case 2

=

Updated Inventory

Erasers	Books	Pens
0	1	2
10	11	12
20	21	22
30	31	32

(4,3)

Rotman

Broadcasting: Case 2

CASE 2: If the shape of the two arrays does not match in some dimension, the array with shape equal to 1 in that dimension is stretched to match the other shape.

Initial Inventory
for erasers, books, pens
stored in single column

Basket 1	0	0	0
Basket 2	10	10	10
Basket 3	20	20	20
Basket 4	30	30	30

(4,1) → (4,3) Case 2

+

You get a delivery:
no more erasers,
1 book for each basket and
2 pens for each basket

0	1	2
0	1	2
0	1	2
0	1	2

(3,) → (1,3) → (4,3)
Case 1 Case 2

=

Updated Inventory

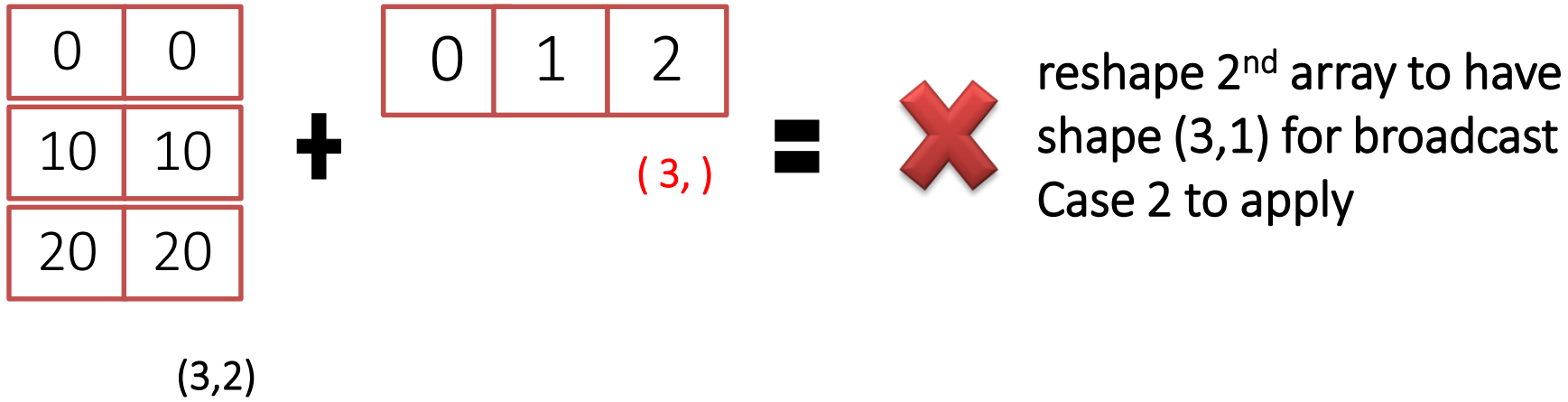
Erasers	Books	Pens
0	1	2
10	11	12
20	21	22
30	31	32

(4,3)

Rotman

Broadcasting: Case 3

CASE 3: If in every dimension the sizes disagree and no dimension is equal to 1, then an error is raised.



Rotman

Broadcasting

- Many examples in previous sections are using the broadcasting mechanism.
- Being able to write short code is not the only reason for broadcasting in NumPy.
- Broadcasting also makes array operations faster.
- If broadcasting is not possible, then a for loop is the answer.

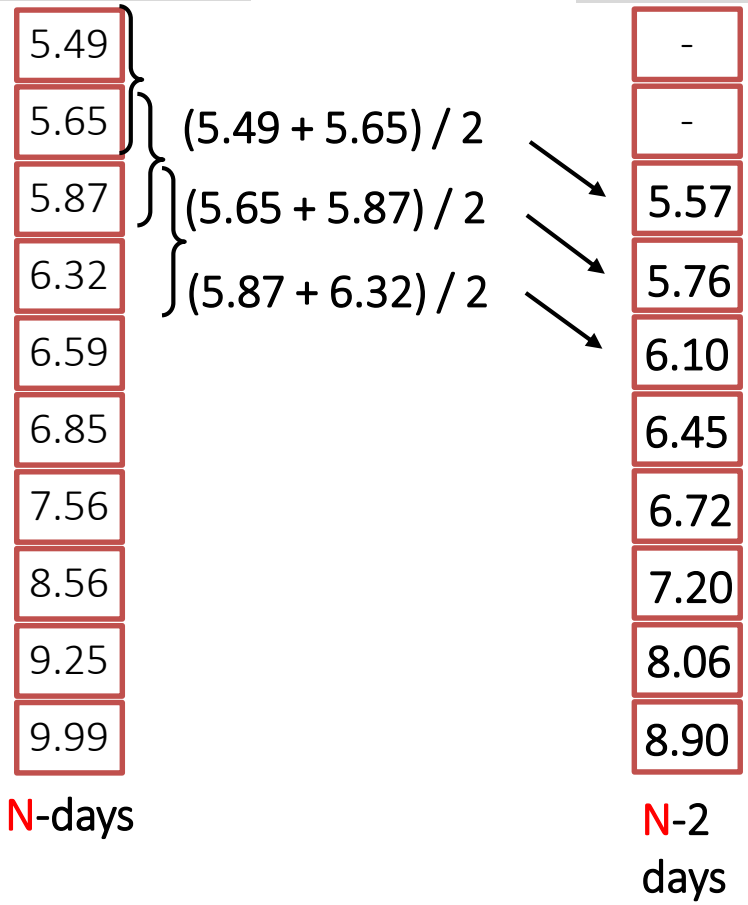
https://numpy.org/doc/stable/user/basics.broadcasting.html

Iterations

Iteration

daily price of a stock

Simple model that predicts today's price as the average of the price of the last 2 days



How do we do this with NumPy?

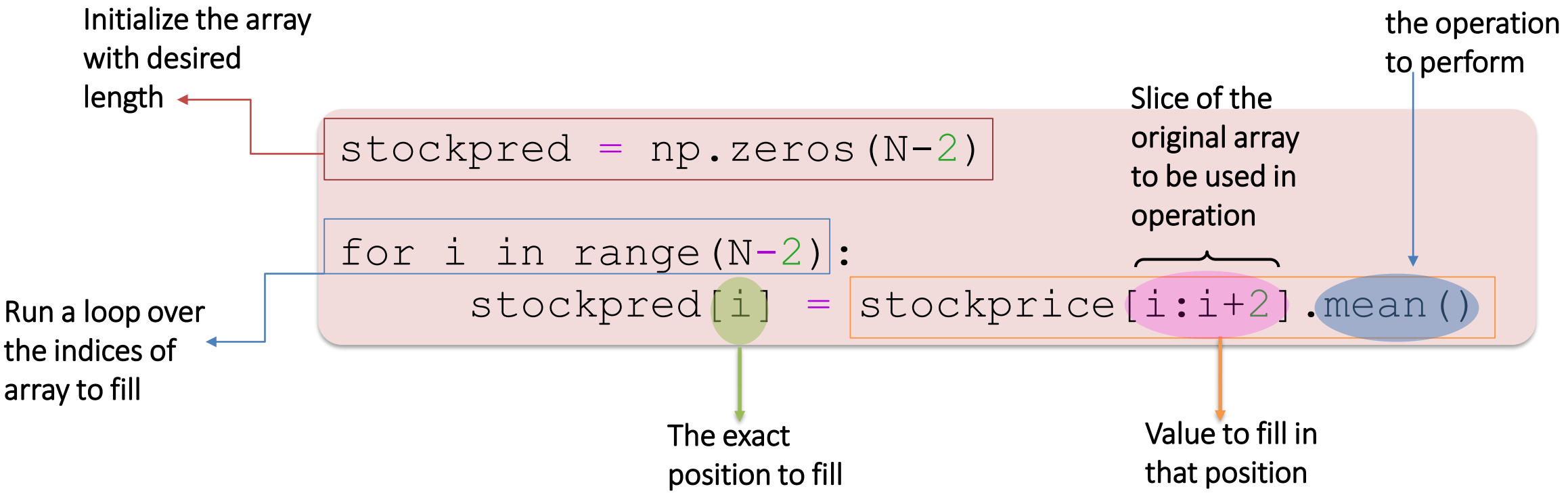
```
stockpred[0] = stockprice[0:2].mean()
stockpred[1] = stockprice[1:3].mean()
stockpred[2] = stockprice[2:4].mean()
stockpred[3] = stockprice[3:5].mean()
```

```
stockpred[i] = stockprice[i:i+2].mean()
```


Rotman

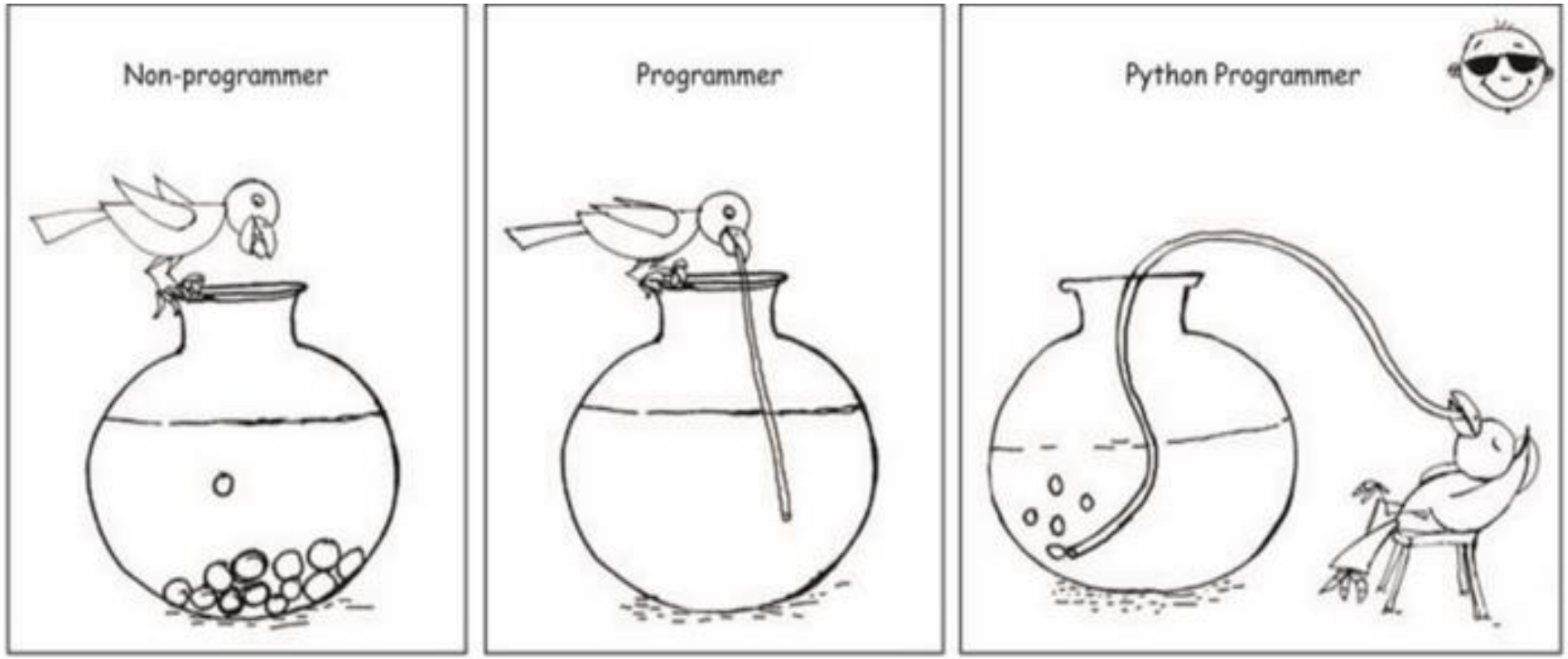
Iteration

Simple model that predicts today's price as the average of the price of the last 2 days



Rotman

Questions?



Who wants to become a Python Programmer?

Thank you