UNIVERSITÄT
BERN

3. Working with numerical data
Part II

Athina Tzovara

University of Bern

Athina.Tzovara@unibe.ch

Last week: introduction to working with numerical data

1. Handling numerical data in python: Numpy

2. Working with arrays

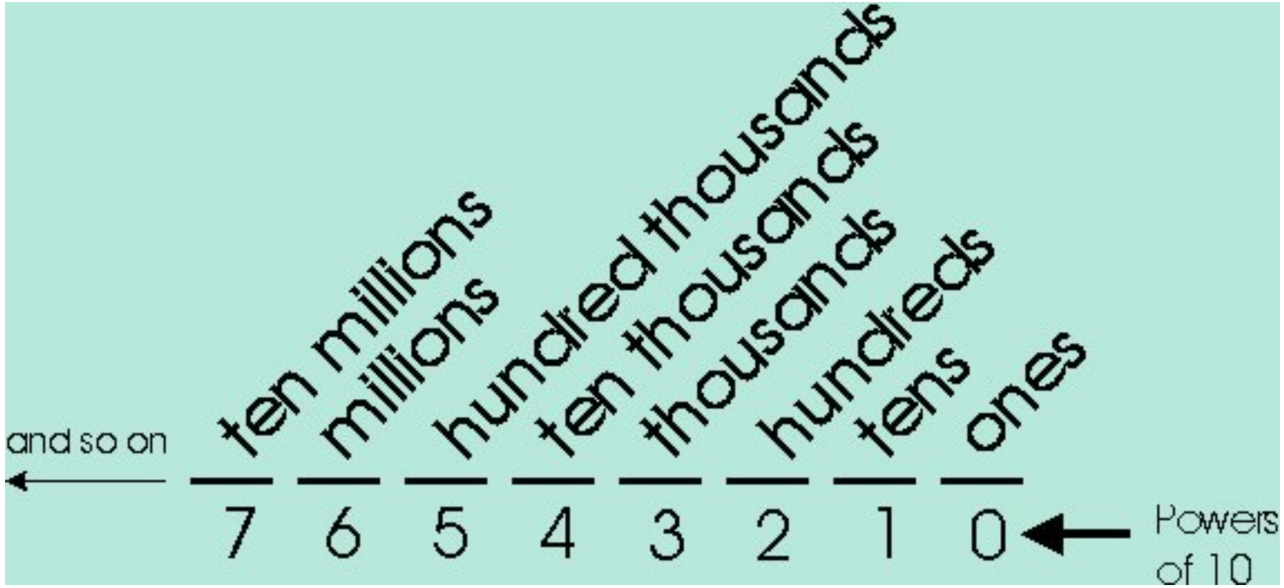
3. Universal Functions

4. Aggregations: min; max;

Today

1. Numerical data types
2. Comparisons; masks and Boolean logic
3. Broadcasting
4. Examples

The decimal numerical system



Decimal:

Used in daily life

Number 10: base

The binary system

Binary:

Used in electronics & computing

Number 2: base {0,1}

0: electric signal off
1: electric signal on

Binary code to compose data in computer-based machines

Reading the binary system (base 2)

Positional system:

Every digit is raised to the powers of 2

Start with the rightmost digit: 2^0

Binary	**1**	**0**	**1**	**0**	**1**	**0**
	x	x	x	x	x	x
	2^5	2^4	2^3	2^2	2^1	2^0

$$32 + 0 + 8 + 0 + 2 + 0 = \boxed{42}$$

Decimal

Reading a binary system (base 2)

Positional system:

Every digit is raised to the powers of 2

Start with the rightmost digit: 2^0

Decimal	Binary
1	00000001
2	00000010
3	00000011
4	00000100
5	00000101
6	00000110
7	00000111
8	00001000

Reading the decimal system (base 10)

Positional system:

Every digit is raised to the powers of 10

Start with the rightmost digit: 10^0

Unique Digits = 0, 1, 2, 3, 4, 5, 6, 7, 8, 9

2	4	8	7	1	6
x	x	x	x	x	x
10^5	10^4	10^3	10^2	10^1	10^0

248’716

Other numeric systems

Octal System - Base 8 -

Unique Digits = 0, 1, 2, 3, 4, 5, 6, 7

DECIMAL	0	1	2	3	4	5	6	7	8	9	10
OCTAL	0	1	2	3	4	5	6	7	010	011	012

Hexadecimal System - Base 16 -

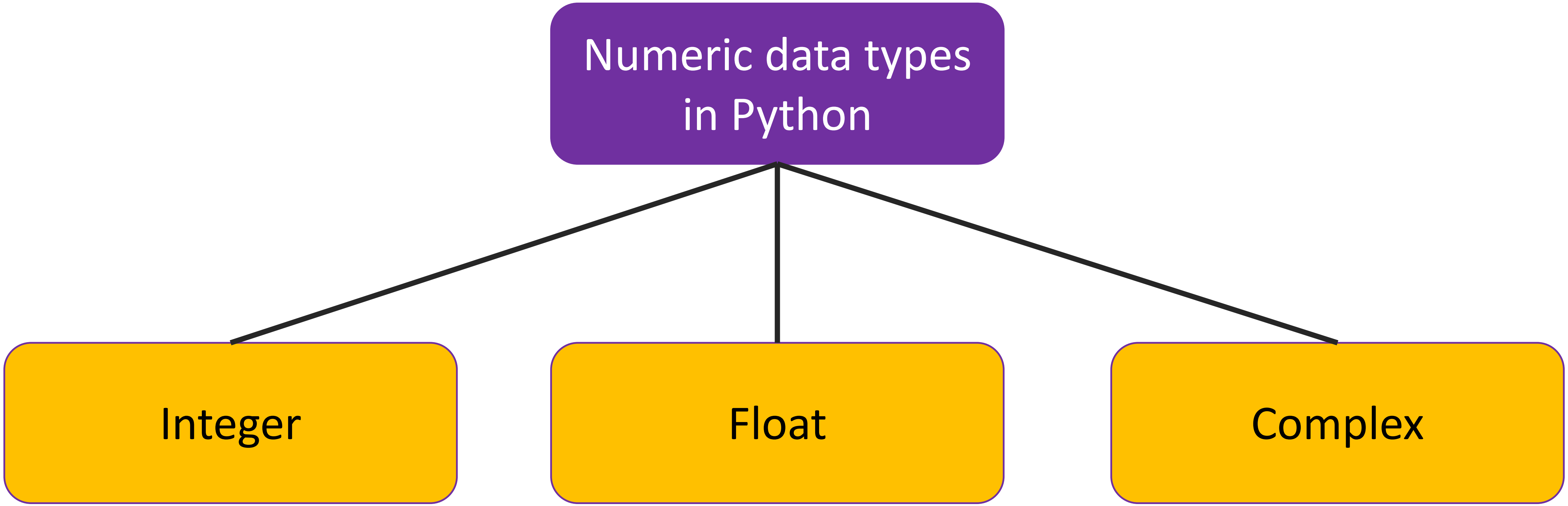
Unique Digits = 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F

DECIMAL	0	1	2	3	4	5	6	7	8	9	10
HEXADECIMAL	0	1	2	3	4	5	6	7	8	9	A

Numeric systems - overview

DECIMAL	BINARY	OCTAL	HEXADECIMAL
0	0000	0	0
1	0001	1	1
2	0010	2	2
3	0011	3	3
4	0100	4	4
5	0101	5	5
...	...	...	...
9	1001	011	9
10	1010	012	A
11	1011	013	B
12	1100	014	C
13	1101	015	D
14	1110	016	E
15	1111	017	F
...	...	...	...
99	1100011	143	63
100	1100100	144	64

Numeric data types in python



Integers

- **Integers**

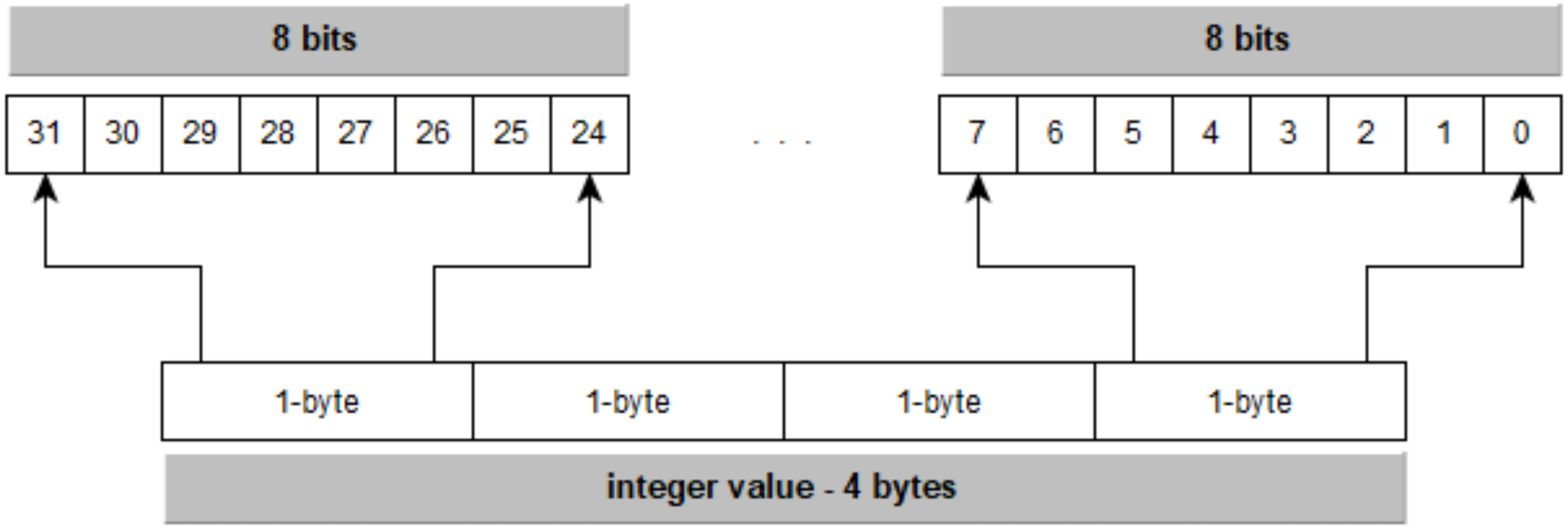
```
10 100 0 -1000
```

In many programming languages: integers are fixed size

Most frequently: 4 bytes --> 2^{32} different values

4 byte integer

Most frequently: 4 bytes --> 2^{32} different values



Salvador-Meneses et al. 2019

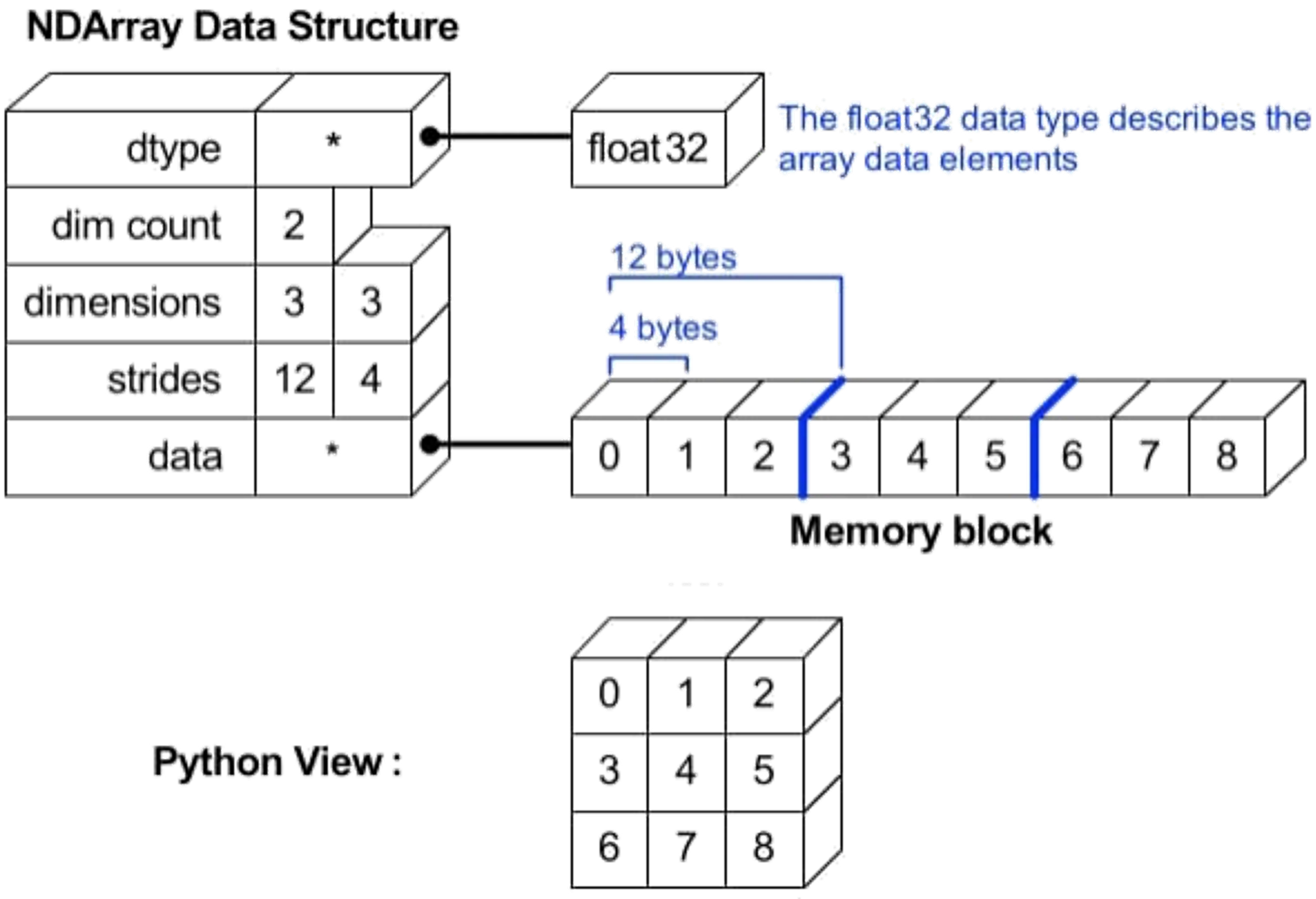
Integers in Python 3

Integers in python 3: unlimited (flexible-sized)
Python will automatically assign more memory as needed

We can calculate very large numbers without additional steps

However: **in numpy: integers have a fixed size**!

Reminder: Data structure of an array



Strides:
how many bytes we have to skip in memory to move to the next position along a certain axis

Python Integer

A ‘Universal’ data type, irrespective of the base that we use

```python
print(92819)
type(92819)
```

```
92819

int
```

```python
print(0o10)
type(0o10)
```

```
8

int
```

```python
print(0x10)
type(0x10)
```

```
16

int
```

Integers in any base system will print as base 10 integers

For python all data type integers are of int data type

Integers in NumPy

We have different numerical types for an integer in NumPy

`numpy.short`	`short`	Platform-defined
`numpy.ushort`	`unsigned short`	Platform-defined
`numpy.intc`	`int`	Platform-defined
`numpy.uintc`	`unsigned int`	Platform-defined
`numpy.int_`	`long`	Platform-defined
`numpy.uint`	`unsigned long`	Platform-defined
`numpy.longlong`	`long long`	Platform-defined
`numpy.ulonglong`	`unsigned long long`	Platform-defined

Source: https://numpy.org/doc/stable/user/basics.types.html

Integers in NumPy

We have different numerical types for an integer in NumPy

`numpy.short`	`short`	Platform-defined
`numpy.ushort`	`unsigned short`	Platform-defined
`numpy.intc`	`int`	Platform-defined
`numpy.uintc`	`unsigned int`	Platform-defined
`numpy.int_`	`long`	Platform-defined
`numpy.uint`	`unsigned long`	Platform-defined
`numpy.longlong`	`long long`	Platform-defined
`numpy.ulonglong`	`unsigned long long`	Platform-defined

Signed integer, compatible with Python int and C long

Source: https://numpy.org/doc/stable/user/basics.types.html

Float and complex numbers in NumPy

Float

`numpy.half` / `numpy.float16`		Half precision float: sign bit, 5 bits exponent, 10 bits mantissa
`numpy.single`	`float`	Platform-defined single precision float: typically sign bit, 8 bits exponent, 23 bits mantissa
`numpy.double`	`double`	Platform-defined double precision float: typically sign bit, 11 bits exponent, 52 bits mantissa.
`numpy.longdouble`	`long double`	Platform-defined extended-precision float

Complex

`numpy.csingle`	`float complex`	Complex number, represented by two single-precision floats (real and imaginary components)
`numpy.cdouble`	`double complex`	Complex number, represented by two double-precision floats (real and imaginary components).
`numpy.clongdouble`	`long double complex`	Complex number, represented by two extended-precision floats (real and imaginary components).

Source: https://numpy.org/doc/stable/user/basics.types.html

Data types in numpy

Many of the NumPy data types have **platform - dependent definitions** (e.g. 32-bit or 64-bit machines)

Fixed - sized aliases are provided:

```
np.int_
```

```
numpy.int64
```

And also:

```
np.longlong
```

```
numpy.int64
```

Data types in numpy

Many of the NumPy data types have **platform - dependent definitions** (e.g. 32-bit or 64-bit machines)

Fixed - sized aliases are provided:

```
np.int_
```

```
numpy.int64
```

```
np.intc
```

```
numpy.int32
```

And also:

```
np.longlong
```

```
numpy.int64
```

```
np.short
```

```
numpy.int16
```

```
np.ushort
```

```
numpy.uint16
```

Overflow Errors

Because of **fixed size numeric types** in NumPy

→

overflow errors when a value requires more memory than available

```
np.power(100, 8, dtype=np.int64)
```

```
10000000000000000
```

Correct calculation for int64

```
np.power(100, 8, dtype=np.int32)
```

```
1874919424
```

Wrong result for int32!

Overflow Errors for NumPy integers

Because of **fixed size numeric types** in NumPy

→

overflow errors when a value requires more memory than available

```
np.power(100, 8, dtype=np.int64)
```

```
10000000000000000
```

Correct calculation for int64

```
np.power(100, 8, dtype=np.int32)
```

```
1874919424
```

Wrong result for int32!

Overflow errors: only for NumPy integers
Python integers: flexible size; they can expand to accommodate any integer; they will not overflow

Overflow Errors for NumPy integers

Minimum and maximum values of NumPy integer and Python integers:

```
np.iinfo(int)
```

```
iinfo(min=-9223372036854775808, max=9223372036854775807, dtype=int64)
```

```
np.iinfo(np.int)
```

```
iinfo(min=-9223372036854775808, max=9223372036854775807, dtype=int64)
```

Minimum and maximum values of NumPy int8 and uint8:

```
np.iinfo(np.int8)
```

```
iinfo(min=-128, max=127, dtype=int8)
```

```
np.iinfo(np.uint8)
```

```
iinfo(min=0, max=255, dtype=uint8)
```

Numerical data types, Python and NumPy

Python 3: Integers are unlimited (no need to worry about data types)

NumPy: Fixed-size integers (we may need to select the appropriate data type if we plan to interfere with machine-level programming)

With the wrong data type for integers in numpy: overflow error

Today

1. Numerical data types
2. **Comparisons; masks and Boolean logic**
3. Broadcasting
4. Examples

Why are we interested in comparisons?

Essential for working with numerical data & calculations

E.g.

- **Retrieve all customers that have made more than 100 purchases last month**

- **All tree species that are taller than 2 meters**

- **Food items with more than 200 calories and less than 6 ingredients**

How do we find which items of an array fulfil a condition?

Our numpy array:

```
z = np.array([2,4,1,5,9,0])
```

```
z
```

```
array([2, 4, 1, 5, 9, 0])
```

Find items that are larger than 2:

```
z>2
```

```
array([False,  True, False,  True,  True, False])
```

The result is a Boolean array

```
z[z>2]
```

```
array([4, 5, 9])
```

Return values of z array larger than 2

Comparison Operators as ufuncs

Operator	Equivalent ufunc	Description
`+`	`np.add`	Addition (e.g., `1 + 1 = 2`)
`-`	`np.subtract`	Subtraction (e.g., `3 - 2 = 1`)
`-`	`np.negative`	Unary negation (e.g., `-2`)
`*`	`np.multiply`	Multiplication (e.g., `2 * 3 = 6`)
`/`	`np.divide`	Division (e.g., `3 / 2 = 1.5`)
`//`	`np.floor_divide`	Floor division (e.g., `3 // 2 = 1`)
`**`	`np.power`	Exponentiation (e.g., `2 ** 3 = 8`)
`%`	`np.mod`	Modulus/remainder (e.g., `9 % 4 = 1`)

Last week: ufuncs as arithmetic operators

+ - * / : for element-wise operations in an array

We can also use similar operators for comparison, as element-wise ufuncs:
< >

More examples

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

```
x < 3  # less than
```

```
array([ True,  True, False, False, False], dtype=bool)
```

```
x > 3  # greater than
```

```
array([False, False, False,  True,  True], dtype=bool)
```

```
x <= 3  # less than or equal
```

```
array([ True,  True,  True, False, False], dtype=bool)
```

```
x >= 3  # greater than or equal
```

```
array([False, False,  True,  True,  True], dtype=bool)
```

```
x != 3  # not equal
```

```
array([ True,  True, False,  True,  True], dtype=bool)
```

```
x == 3  # equal
```

```
array([False, False,  True, False, False], dtype=bool)
```

2D arrays

```
rng = np.random.RandomState(0)
x = rng.randint(10, size=(3, 4))
x
```

```
array([[5, 0, 3, 3],
       [7, 9, 3, 5],
       [2, 4, 7, 6]])
```

```
x < 6
```

```
array([[ True,  True,  True,  True],
       [False, False,  True,  True],
       [ True,  True, False, False]], dtype=bool)
```

The result is a Boolean array!

More examples: dataset

Context

This data set dates from 1988 and consists of four databases: Cleveland, Hungary, Switzerland, and Long Beach V. It contains 76 attributes, including the predicted attribute, but all published experiments refer to using a subset of 14 of them. The "target" field refers to the presence of heart disease in the patient. It is integer valued 0 = no disease and 1 = disease.

Content

Attribute Information:

1. age
2. sex
3. chest pain type (4 values)
4. resting blood pressure
5. serum cholestoral in mg/dl
6. fasting blood sugar > 120 mg/dl
7. resting electrocardiographic results (values 0,1,2)
8. maximum heart rate achieved
9. exercise induced angina
10. oldpeak = ST depression induced by exercise relative to rest
11. the slope of the peak exercise ST segment
12. number of major vessels (0-3) colored by flourosopy
13. thal: 0 = normal; 1 = fixed defect; 2 = reversable defect
    The names and social security numbers of the patients were recently removed from the database, replaced with dummy values.

https://www.kaggle.com/johnsmith88/heart-disease-dataset

Why are Boolean arrays useful?

```
[Sex, RestBP, Chol, Age] = np.load('Heart.npy')
```

```
Chol
```

```
array([233, 286, 229, 250, 204, 236, 268, 354, 254, 203, 192, 294, 256,
       263, 199, 168, 229, 239, 275, 266, 211, 283, 284, 224, 206, 219,
       340, 226, 247, 167, 239, 230, 335, 234, 233, 226, 177, 276, 353,
       243, 225, 199, 302, 212, 330, 230, 175, 243, 417, 197, 198, 177,
       290, 219, 253, 266, 233, 172, 273, 213, 305, 177, 216, 304, 188,
       282, 185, 232, 326, 231, 269, 254, 267, 248, 197, 360, 258, 308,
       245, 270, 208, 264, 321, 274, 325, 235, 257, 216, 234, 256, 302,
       164, 231, 141, 252, 255, 239, 258, 201, 222, 260, 182, 303, 265,
       188, 309, 177, 229, 260, 219, 307, 249, 186, 341, 263, 203, 211,
       183, 330, 254, 256, 407, 222, 217, 282, 234, 288, 239, 220, 209,
       258, 227, 204, 261, 213, 250, 174, 281, 198, 245, 221, 288, 205,
       309, 240, 243, 289, 250, 308, 318, 298, 265, 564, 289, 246, 322,
       299, 300, 293, 277, 197, 304, 214, 248, 255, 207, 223, 288, 282,
       160, 269, 226, 249, 394, 212, 274, 233, 184, 315, 246, 274, 409,
       244, 270, 305, 195, 240, 246, 283, 254, 196, 298, 247, 294, 211,
       299, 234, 236, 244, 273, 254, 325, 126, 313, 211, 309, 259, 200,
       262, 244, 215, 231, 214, 228, 230, 193, 204, 243, 303, 271, 268,
       267, 199, 282, 269, 210, 204, 277, 206, 212, 196, 327, 149, 269,
       201, 286, 283, 249, 271, 295, 235, 306, 269, 234, 178, 237, 234,
       275, 212, 208, 201, 218, 263, 295, 303, 209, 223, 197, 245, 261,
       242, 319, 240, 226, 166, 315, 204, 218, 223, 180, 207, 228, 311,
       149, 204, 227, 278, 220, 232, 197, 335, 253, 205, 192, 203, 318,
       225, 220, 221, 240, 212, 342, 169, 187, 197, 157, 176, 241, 264,
       193, 131, 236, 175])
```

Values of
serum cholesterol in mg/dl

Boolean arrays as masks

Array with Cholesterol values

```
Chol
```

```
array([233, 286, 229, 250, 204, 236, 268, 354, 254, 203, 192, 294, 256,
       263, 199, 168, 229, 239, 275, 266, 211, 283, 284, 224, 206, 219,
       340, 226, 247, 167, 239, 230, 335, 234, 233, 226, 177, 276, 353,
       243, 225, 199, 302, 212, 330, 230, 175, 243, 417, 197, 198, 177,
       290, 219, 253, 266, 233, 172, 273, 213, 305, 177, 216, 304, 188,
       282, 185, 232, 326, 231, 269, 254, 267, 248, 197, 360, 258, 308,
       245, 270, 208, 264, 321, 274, 325, 235, 257, 216, 234, 256, 302,
       164, 231, 141, 252, 255, 239, 258, 201, 222, 260, 182, 303, 265,
       188, 309, 177, 229, 260, 219, 307, 249, 186, 341, 263, 203, 211,
       183, 330, 254, 256, 407, 222, 217, 282, 234, 288, 239, 220, 209,
       258, 227, 204, 261, 213, 250, 174, 281, 198, 245, 221, 288, 205,
       309, 240, 243, 289, 250, 308, 318, 298, 265, 564, 289, 246, 322,
       299, 300, 293, 277, 197, 304, 214, 248, 255, 207, 223, 288, 282,
       160, 269, 226, 249, 394, 212, 274, 233, 184, 315, 246, 274, 409,
       244, 270, 305, 195, 240, 246, 283, 254, 196, 298, 247, 294, 211,
       299, 234, 236, 244, 273, 254, 325, 126, 313, 211, 309, 259, 200,
       262, 244, 215, 231, 214, 228, 230, 193, 204, 243, 303, 271, 268,
       267, 199, 282, 269, 210, 204, 277, 206, 212, 196, 327, 149, 269,
       201, 286, 283, 249, 271, 295, 235, 306, 269, 234, 178, 237, 234,
       275, 212, 208, 201, 218, 263, 295, 303, 209, 223, 197, 245, 261,
       242, 319, 240, 226, 166, 315, 204, 218, 223, 180, 207, 228, 311,
       149, 204, 227, 278, 220, 232, 197, 335, 253, 205, 192, 203, 318,
       225, 220, 221, 240, 212, 342, 169, 187, 197, 157, 176, 241, 264,
       193, 131, 236, 175])
```

Boolean array as a mask

```
Chol < 180
```

```
array([False, False, False, False, False, False, False, False, False,
       False, False, False, False, False, False,  True, False, False,
       False, False, False, False, False, False, False, False, False,
       False, False,  True, False, False, False, False, False, False,
        True, False, False, False, False, False, False, False, False,
       False,  True, False, False, False, False,  True, False, False,
       False, False, False,  True, False, False, False,  True, False,
       False, False, False, False, False, False, False, False, False,
       False, False, False, False, False, False, False, False, False,
       False, False, False, False, False, False, False, False, False,
       False,  True, False,  True, False, False, False, False, False,
       False, False, False, False, False, False, False,  True, False,
       False, False, False, False, False, False, False, False, False,
       False, False, False, False, False, False, False, False, False,
       False, False, False, False, False, False, False, False, False,
       False,  True, False, False, False, False, False, False, False,
       False, False, False, False, False, False, False, False, False,
       False, False, False, False, False, False, False, False, False,
       False, False, False, False, False, False, False,  True, False,
       False, False, False, False, False, False, False, False, False,
       False, False, False, False, False, False, False, False, False,
       False, False, False, False, False, False, False, False, False,
       False, False, False, False,  True, False, False, False, False,
       False, False, False, False, False, False, False, False, False,
       False, False, False, False, False, False, False, False, False,
       False, False, False, False, False, False, False,  True, False,
       False, False, False, False, False, False, False, False, False,
       False,  True, False, False, False, False, False, False, False,
       False, False, False, False, False, False, False, False, False,
       False, False, False,  True, False, False, False, False, False,
       False, False, False,  True, False, False, False, False, False,
       False, False, False, False, False, False, False, False, False,
       False, False, False, False,  True, False, False,  True,  True,
       False, False, False,  True, False,  True])
```

Boolean arrays as masks

We can also select values from the array; the ones where Cholesterol is lower than 180:

```
Chol[Chol<180]
```

```
array([168, 167, 177, 175, 177, 172, 177, 164, 141, 177, 174, 160, 126,
       149, 178, 166, 149, 169, 157, 176, 131, 175])
```

Boolean arrays for counting entries

How many patients with a heart attack have less cholesterol than 200 mg/dl?

```
np.count_nonzero(Chol < 200)
```

```
49
```

This will create a Boolean array

np.count_nonzero() will count the number of True entries in our Boolean array

```
np.sum(Chol<200)
```

```
49
```

The same can be achieved with **np.sum**
for a Boolean array **False: 0 True: 1**

Boolean arrays for counting entries

Are there *any* patients with less cholesterol than 180 mg/dl?

```
np.any(Chol < 180)
```

```
True
```

```
np.any(Chol < 100)
```

any patients with less cholesterol than 100 mg/dl?

```
False
```

```
np.all(Chol > 100)
```

Do *all* patients have more cholesterol than 100 mg/dl?

```
True
```

```
np.all(Chol < 180)
```

Do *all* patients have more cholesterol than *180* mg/dl?

```
False
```

Boolean operators

Reminder: bitwise logic operators in Python **& | ^ ~**

<table>
<tr><th>OPERATOR</th><th>DESCRIPTION</th><th>SYNTAX</th></tr>
<tr><td>&</td><td>Bitwise AND</td><td>x & y</td></tr>
<tr><td>|</td><td>Bitwise OR</td><td>x | y</td></tr>
<tr><td>~</td><td>Bitwise NOT</td><td>~x</td></tr>
<tr><td>^</td><td>Bitwise XOR</td><td>x ^ y</td></tr>
</table>

Boolean operators

Example: we want extreme values of cholesterol:

How many patients with cholesterol higher than 300 OR lower than 180?

```
np.sum((Chol > 300) | (Chol < 180))
```

```
66
```

OR

Boolean operators

Example: we want a range of cholesterol values:

How many patients with cholesterol higher than 150 AND lower than 180?

```
np.sum((Chol > 150) & (Chol < 180))
```

```
17
```

AND

Boolean operators and, or vs. &, |

When do we use each of them?

and or : False or True for an entire object

& | : False or True for bits inside an object

When we use *and or*, **we ask python to treat the object as a single Boolean entity:**

```
bool(42), bool(0)
```

```
(True, False)
```

```
bool(42 and 0)
```

```
False
```

```
bool(42 or 0)
```

```
True
```

Boolean operators and, or vs. &, |

When we use & and | on integers: the expression operates on the bits of the element, applying the *and* or the *or* to the individual bits making up the number:

```
bin(42)
```

```
'0b101010'
```

```
bin(59)
```

```
'0b111011'
```

```
bin(42 & 59)
```

```
'0b101010'
```

```
bin(42 | 59)
```

```
'0b111011'
```

We compare:

the corresponding bits of the binary representation to yield the result.

Boolean operators and, or vs. &, |

When we use & and | on integers: the expression operates on the bits of the element, applying the *and* or the *or* to the individual bits making up the number:

```
A = np.array([1, 0, 1, 0, 1, 0], dtype=bool)
B = np.array([1, 1, 1, 0, 1, 1], dtype=bool)
A | B
```

```
array([ True,  True,  True, False,  True,  True], dtype=bool)
```

Using `or` on these arrays will try to evaluate the truth or falsehood of the entire array object, which is not a well-defined value:

```
A or B
```

```
---------------------------------------------------------------------
ValueError                                Traceback (most recent cal
<ipython-input-38-5d8e4f2e21c0> in <module>()
----> 1 A or B

ValueError: The truth value of an array with more than one element i
```

Today

1. Numerical data types

2. Comparisons; masks and Boolean logic

3. **Broadcasting**

4. Examples

Broadcasting

```
import numpy as np
```

```
a = np.array([0, 1, 2])
b = np.array([5, 5, 5])
a + b
```

```
array([5, 6, 7])
```

For arrays of the same size: binary operations are performed on an element-by-element basis

```
a + 5
```

```
array([5, 6, 7])
```

scalar (zero dimensional array)

Broadcasting: binary operations can be performed on arrays of different sizes

Broadcasting

Intuition behind broadcasting:

To add a scalar to an array: “stretch” or duplicate the value “5” into the array [5, 5, 5]

```
a + 5
```

```
array([5, 6, 7])
```

scalar (zero dimensional array)

Broadcasting for one array

Extending broadcasting to arrays of higher dimensions

```
M = np.ones((3, 3))
M
```

```
array([[ 1.,  1.,  1.],
       [ 1.,  1.,  1.],
       [ 1.,  1.,  1.]])
```

```
M + a
```

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

The one-dimensional array a is **broadcasted** across the second dimension to match the shape of M.

Broadcasting for two arrays

There are cases where dimensions from both arrays can change

The geometry of **both** arrays needs to be stretched in order to perform the desired operation

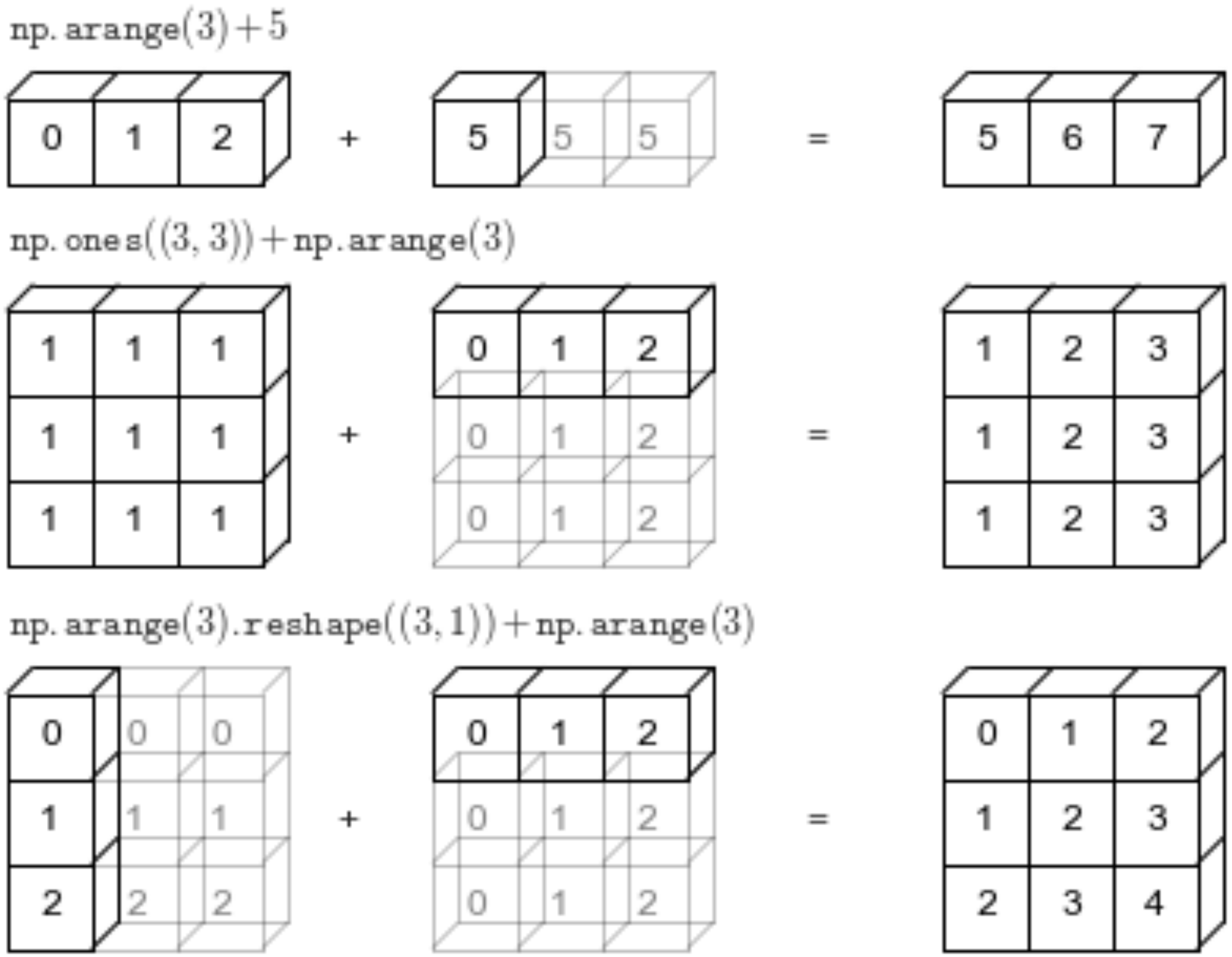
```
a = np.arange(3)
b = np.arange(3)[:, np.newaxis]

print(a)
print(b)
```

```
[0 1 2]
[[0]
 [1]
 [2]]
```

```
a + b
```

```
array([[0, 1, 2],
       [1, 2, 3],
       [2, 3, 4]])
```

Broadcasting: overview



Broadcasting: overview

Three rules of broadcasting:

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

2. If the shape of the two arrays **does not match** in any dimension, the array with shape equal to 1 is **stretched** to match the other shape

3. If in any dimension the sizes disagree and neither is equal to 1, an error is raised

Example of broadcasting I

```
M = np.ones((2, 3))
a = np.arange(3)
```

- `M.shape = (2, 3)`
- `a.shape = (3,)`

How will broadcasting work?

Example of broadcasting I

```
M = np.ones((2, 3))
a = np.arange(3)
```

- `M.shape = (2, 3)`
- `a.shape = (3,)`

How will broadcasting work?

1. Array a has fewer dimensions → it will be padded to the left with ones (**Rule 1**)
   - `M.shape -> (2, 3)`
   - `a.shape -> (1, 3)`

Example of broadcasting I

```
M = np.ones((2, 3))
a = np.arange(3)
```

- `M.shape = (2, 3)`
- `a.shape = (3,)`

How will broadcasting work?

1. Array a has fewer dimensions → it will be padded to the left with ones (**Rule 1**)

- `M.shape -> (2, 3)`
- `a.shape -> (1, 3)`

2. The first dimension of array with shape 1 disagrees → it will be stretched to match (**Rule 2**)

- `M.shape -> (2, 3)`
- `a.shape -> (2, 3)`

Example of broadcasting I

```
M = np.ones((2, 3))
a = np.arange(3)
```

- `M.shape = (2, 3)`
- `a.shape = (3,)`

How will broadcasting work?

1. Array a has fewer dimensions → it will be padded to the left with ones (**Rule 1**)
   - `M.shape -> (2, 3)`
   - `a.shape -> (1, 3)`

2. The first dimension of array with shape 1 disagrees → it will be stretched to match (**Rule 2**)
   - `M.shape -> (2, 3)`
   - `a.shape -> (2, 3)`

Result

```
M + a
```

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

Example of broadcasting II

```
a = np.arange(3).reshape((3, 1))
b = np.arange(3)
```

- `a.shape = (3, 1)`
- `b.shape = (3,)`

How will broadcasting work?

Example of broadcasting II

```
a = np.arange(3).reshape((3, 1))
b = np.arange(3)
```

- `a.shape = (3, 1)`
- `b.shape = (3,)`

How will broadcasting work?

1. Array b has fewer dimensions → it will be padded with ones (**Rule 1**)

- `a.shape -> (3, 1)`
- `b.shape -> (1, 3)`

Example of broadcasting II

```
a = np.arange(3).reshape((3, 1))
b = np.arange(3)
```

- `a.shape = (3, 1)`
- `b.shape = (3,)`

How will broadcasting work?

1. Array b has fewer dimensions → it will be padded with ones (**Rule 1**)

    - `a.shape -> (3, 1)`
    - `b.shape -> (1, 3)`

2. Both dimensions of arrays with shape 1 disagree → we need to upgrade each of them to match (**Rule 2**)

    - `a.shape -> (3, 3)`
    - `b.shape -> (3, 3)`

Example of broadcasting II

```
a = np.arange(3).reshape((3, 1))
b = np.arange(3)
```

- `a.shape = (3, 1)`
- `b.shape = (3,)`

How will broadcasting work?

1. Array b has fewer dimensions → it will be padded with ones (**Rule 1**)

   - `a.shape -> (3, 1)`
   - `b.shape -> (1, 3)`

2. Both dimensions of arrays with shape 1 disagree → we need to upgrade each of them to match (**Rule 2**)

   - `a.shape -> (3, 3)`
   - `b.shape -> (3, 3)`

Result

```
a + b
```

```
array([[0, 1, 2],
       [1, 2, 3],
       [2, 3, 4]])
```

Example of broadcasting III

```
M = np.ones((3, 2))
a = np.arange(3)
```

- `M.shape = (3, 2)`
- `a.shape = (3,)`

How will broadcasting work?

Example of broadcasting III

```
M = np.ones((3, 2))
a = np.arange(3)
```

- `M.shape = (3, 2)`
- `a.shape = (3,)`

How will broadcasting work?

1. Array a has fewer dimensions → it will be padded with ones (**Rule 1**)

- `M.shape -> (3, 2)`
- `a.shape -> (1, 3)`

Example of broadcasting III

```
M = np.ones((3, 2))
a = np.arange(3)
```

- `M.shape = (3, 2)`
- `a.shape = (3,)`

How will broadcasting work?

1. Array a has fewer dimensions → it will be padded with ones (**Rule 1**)

- `M.shape -> (3, 2)`
- `a.shape -> (1, 3)`

2. The first dimension of array with shape 1 disagrees → it will be stretched (**Rule 2**)

- `M.shape -> (3, 2)`
- `a.shape -> (3, 3)`

Example of broadcasting III

```
M = np.ones((3, 2))
a = np.arange(3)
```

- `M.shape = (3, 2)`
- `a.shape = (3,)`

How will broadcasting work?

1. Array a has fewer dimensions → it will be padded with ones (**Rule 1**)

- `M.shape -> (3, 2)`
- `a.shape -> (1, 3)`

2. The first dimension of array with shape 1 disagrees → it will be stretched (**Rule 2**)

- `M.shape -> (3, 2)`
- `a.shape -> (3, 3)`

Result

```
M + a
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent cal
<ipython-input-13-9e16e9f98da6> in <module>()
----> 1 M + a

ValueError: operands could not be broadcast together with shapes (3,
```

Violation of Rule 3: the final shapes do not match → incompatible arrays

Example of broadcasting III - alternative

```
M = np.ones((3, 2))
a = np.arange(3)
```

- `M.shape = (3, 2)`
- `a.shape = (3,)`

Padding to the right side by reshaping the array:

```
a[:, np.newaxis].shape
```

```
(3, 1)
```

```
M + a[:, np.newaxis]
```

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

Centering an array

Reminder: **ufuncs** of numpy: remove the need for explicit slow loops.
Broadcasting extends this ability.

Example. Centering an array, 10 observations, 3 values each:

```
X = np.random.random((10, 3))
```

We can compute the mean of each dimension:

```
Xmean = X.mean(0)
Xmean
```

```
array([ 0.53514715,  0.66567217,  0.44385899])
```

```
X_centered = X - Xmean
```

```
X_centered.mean(0)
```

```
array([  2.22044605e-17,  -7.77156117e-17,  -1.66533454e-17])
```

Broadcasting: we center the array by subtracting the mean

Fancy indexing

How can we access multiple elements of an array at once?

```
import numpy as np
rand = np.random.RandomState(42)

x = rand.randint(100, size=10)
print(x)
```

```
[51 92 14 71 60 20 82 86 74 74]
```

Different options:

```
[x[3], x[7], x[2]]
```

```
[71, 86, 14]
```

```
ind = [3, 7, 4]
x[ind]
```

```
array([71, 86, 60])
```

Fancy indexing for 1D

How can we access multiple elements of an array at once?

```
import numpy as np
rand = np.random.RandomState(42)

x = rand.randint(100, size=10)
print(x)
```

```
[51 92 14 71 60 20 82 86 74 74]
```

Fancy indexing: the shape of result reflects the shape of the index arrays; not of the array we index

```
ind = np.array([[3, 7],
                [4, 5]])
x[ind]
```

```
array([[71, 86],
       [60, 20]])
```

Fancy indexing for multiple dimensions

```
X = np.arange(12).reshape((3, 4))
X
```

```
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
```

Standard indexing: first index row; second index column

```
X[row[:, np.newaxis], col]
```

```
array([[ 2,  1,  3],
       [ 6,  5,  7],
       [10,  9, 11]])
```

Fancy indexing: Broadcasting rules apply

```
row[:, np.newaxis] * col
```

```
array([[0, 0, 0],
       [2, 1, 3],
       [4, 2, 6]])
```

Combined indexing

```
print(X)
```

```
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]]
```

What do you expect here?

```
X[2, [2, 0, 1]]
```

Combined indexing

```
print(X)
```

```
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]]
```

What do you expect here?

```
X[2, [2, 0, 1]]
```

```
array([10,  8,  9])
```

Combined indexing

```
print(X)
```

```
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]]
```

```
X[2, [2, 0, 1]]
```

```
array([10,  8,  9])
```

What do you expect here?

```
X[1:, [2, 0, 1]]
```

Combined indexing

```
print(X)
```

```
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]]
```

```
X[2, [2, 0, 1]]
```

```
array([10,  8,  9])
```

What do you expect here?

```
X[1:, [2, 0, 1]]
```

```
array([[ 6,  4,  5],
       [10,  8,  9]])
```

Sorting an array

np.sort() <-- Sorting function in numpy

```
>>> a = np.array([[1,4],[3,1]])
>>> np.sort(a)                # sort along the last axis
array([[1, 4],
       [1, 3]])
>>> np.sort(a, axis=None)     # sort the flattened array
array([1, 1, 3, 4])
>>> np.sort(a, axis=0)        # sort along the first axis
array([[1, 1],
       [3, 4]])
```

Sorting an array

np.argsort() <-- Indexes that would sort an array

```
>>> x = np.array([3, 1, 2])
>>> np.argsort(x)
array([1, 2, 0])
```

1D array

Sorting an array

np.argsort() <-- Indexes that would sort an array

```
>>> x = np.array([[0, 3], [2, 2]])
>>> x
array([[0, 3],
       [2, 2]])
```

```
>>> ind = np.argsort(x, axis=0)  # sorts along first axis (down)
>>> ind
array([[0, 1],
       [1, 0]])
```

```
>>> ind = np.argsort(x, axis=1)  # sorts along last axis (across)
>>> ind
array([[0, 1],
       [0, 1]])
```

2D array

Today

1. Numerical data types
2. Comparisons; masks and Boolean logic
3. Broadcasting
4. Examples