QLS-612 2023 Module 4

Jérôme Dockès 2023-05-02

1. Numerical computation in Python

2. The Numpy ndarray

3. Numpy in practice: manipulating brain images

What does the code below compute?

(**values** is a sequence of 1,000 floating-point numbers)

C version:

```c
double squared = 0.0;
for (int i = 0; i != size; ++i) {
  squared += values[i] * values[i];
}
double res = sqrt(squared);
```

Python version:

```python
squared = 0.0
for val in values:
    squared += val**2
res = math.sqrt(squared)
```

Computing the L$_2$ norm of a 1,000-dimensional vector

C version:

```
double squared_norm = 0.0;
for (int i = 0; i != size; ++i) {
  squared_norm += values[i] * values[i];
}
double norm = sqrt(squared_norm);
```
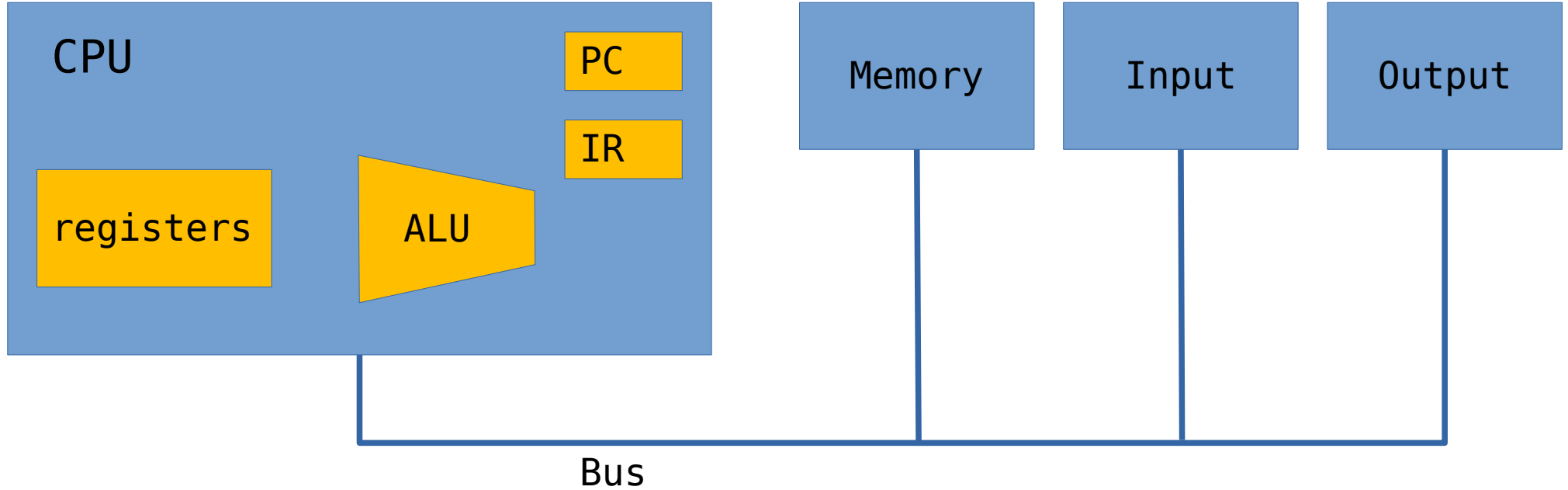
Duration: ~ **1.3 μs**

Python version:

```
squared_norm = 0.0
for val in values:
    squared_norm += val**2
norm = math.sqrt(squared_norm)
```
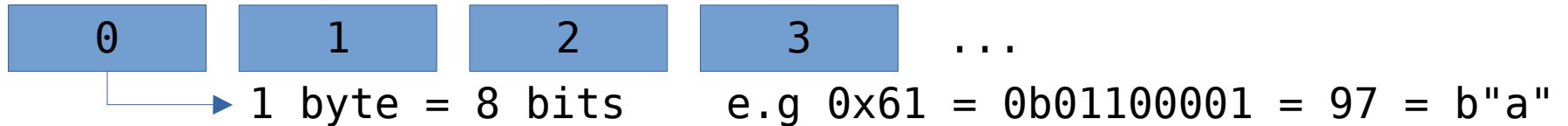
Duration: ~ **220 μs**

Why is the C program so much faster?
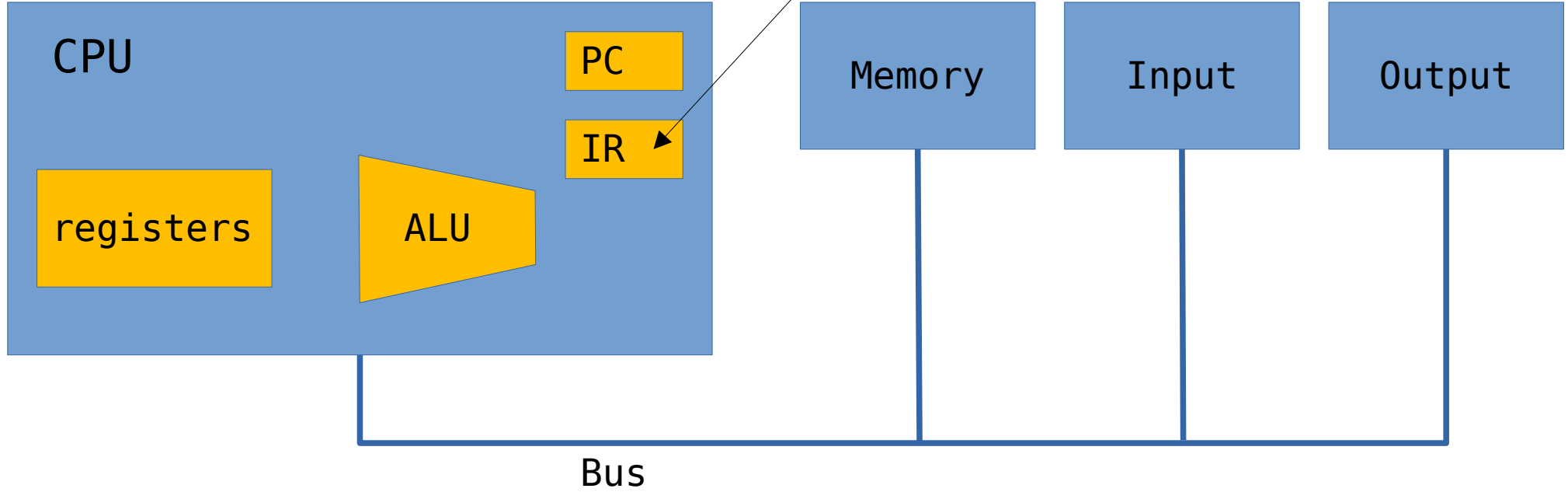
fetch, decode, execute, write-back

CPU

PC

IR

registers

ALU

Memory

Input

Output

Bus

Memory (RAM): long sequence of bytes, each with an address

| 0 | 1 | 2 | 3 | ... |

1 byte = 8 bits   e.g 0x61 = 0b01100001 = 97 = b"a"

fetch, decode, execute, write-back

f20f58c1: "ADD xmm0 and xmm1, store result in xmm0"

CPU

PC

IR

registers

ALU

Memory

Input

Output

Bus

Memory (RAM): long sequence of bytes, each with an address

| 0 | 1 | 2 | 3 | ... |

1 byte = 8 bits   e.g 0x61 = 0b01100001 = 97 = b"a"
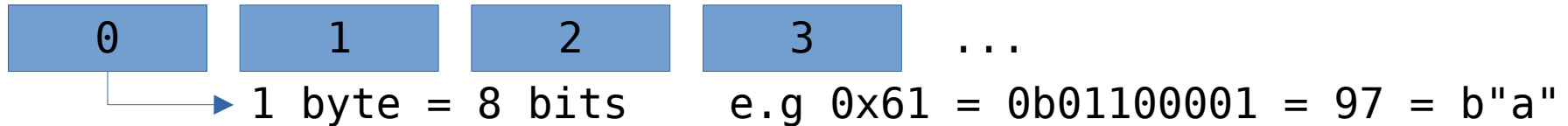
Machine instructions: low-level operations
directly implemented by the hardware

dot += val * val;

```
f20f1045f8  movsd  -0x8(%rbp),%xmm0     copy content of &val to xmm0

f20f59c0    mulsd  %xmm0,%xmm0          multiply content of xmm0 by
                                          itself and store result in xmm0

f20f104df0  movsd  -0x10(%rbp),%xmm1    copy content of &dot to xmm1

f20f58c1    addsd  %xmm1,%xmm0          add contents of xmm1 and xmm0
                                          and store result in xmm0

f20f1145f0  movsd  %xmm0,-0x10(%rbp)    copy content of xmm0 into &dot
```

The compiler can also perform optimizations

C pipeline

```c
double squared_norm = 0.0;
for (int i = 0; i != size; ++i) {
  squared_norm += values[i] * values[i];
}
double norm = sqrt(squared_norm);
```

compile

```
f20f59c1    mulsd  %xmm1,%xmm0
f20f104df0  movsd  -0x10(%rbp),%xmm1
f20f58c1    addsd  %xmm1,%xmm0
f20f1145f0  movsd  %xmm0,-0x10(%rbp)
...
```

run

compute_norm

CPU

Python Bytecode: high-level, abstract instructions
for an idealized machine

```
dot += val * val

LOAD_FAST          0  load dot
LOAD_FAST          1  load val
LOAD_FAST          1  load val
BINARY_MULTIPLY       *
INPLACE_ADD           +=
STORE_FAST         0  store dot
```

These would not mean anything to a CPU

→ They must be translated by the Python VM at runtime

C pipeline

```c
double squared_norm = 0.0;
for (int i = 0; i != size; ++i) {
  squared_norm += values[i] * values[i];
}
double norm = sqrt(squared_norm);
```

Compile ahead of time

```
f20f59c1    mulsd   %xmm1,%xmm0
f20f104df0  movsd   -0x10(%rbp),%xmm1
f20f58c1    addsd   %xmm1,%xmm0
f20f1145f0  movsd   %xmm0,-0x10(%rbp)
...
```

Run on hardware

compute_norm

CPU

Python pipeline

```python
squared_norm = 0.0
for val in values:
    squared_norm += val**2
norm = math.sqrt(squared_norm)
```

Compile at runtime

```
LOAD_FAST
LOAD_CONST
BINARY_POWER
INPLACE_ADD
STORE_FAST
...
```

Run on CPython VM

compute_norm.py

python

CPU

Python is **dynamically** typed

```python
squared_norm = 0.0
for val in values:
    squared_norm += val * val
norm = math.sqrt(squared_norm)
```

could be anything!

```
...
LOAD_FAST
LOAD_FAST
LOAD_FAST
BINARY_MULTIPLY
INPLACE_ADD
STORE_FAST
...
```

val

PyFloatObject

ob_refcnt = 1

ob_type

ob_fval = 1.234

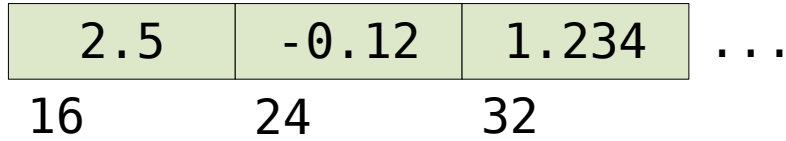**PyTypeObject**

...

tp_as_number
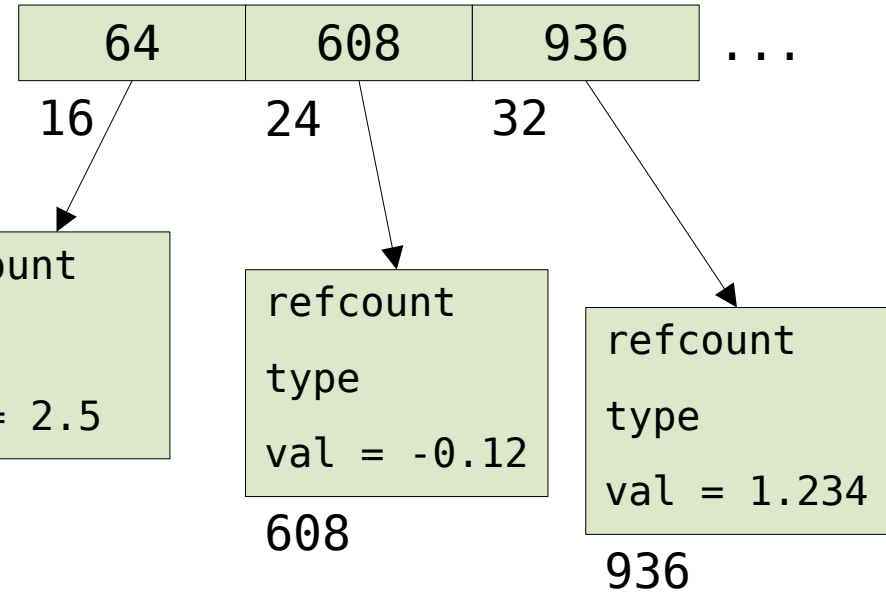
...

PyNumberMethods

...

nb_multiply

...

float_mul(PyObject *v, PyObject *w);

Spatial locality

C array

| 2.5 | -0.12 | 1.234 | ...

16   24   32

Python list

| 64 | 608 | 936 | ...

16   24   32

```
refcount

type

val = 2.5
```
64

```
refcount

type

val = -0.12
```
608

```
refcount

type

val = 1.234
```
936

CPU

Registers
hundreds of Bytes
< 1 ns

Cache (L2)
~ 1 MB
~ 5 ns

Memory
~ 16 GB
~ 100 ns

Bus

The C program spends most of its time performing numerical operations.

The Python program spends most of its time running the interpreter's machinery, inspecting types and looking up functions, allocating and freeing memory, and waiting for memory read operations.

For a fast computation we need

- **static typing** & compiling to optimized **machine code**
- data locality — an **efficient data structure** like a C array

The CPython interpreter is written in C.
It can call C functions and manipulate C data structures.
It provides an API to link our own C code into the Python runtime

numpy provides:

- many routines implemented in C and exposed in Python modules
- a powerful and optimized data structure: the numpy array

C: ~ 1.3 μs

```c
double squared_norm = 0.0;
for (int i = 0; i != size; ++i) {
  squared_norm += values[i] * values[i];
}
double norm = sqrt(squared_norm);
```

Python: ~ 220 μs

```python
squared_norm = 0.0
for val in values:
    squared_norm += val**2
norm = math.sqrt(squared_norm)
```

Python with numpy: ~ 3 μs

```python
norm = np.linalg.norm(values)
```

Getting started with numpy

```
pip install -U numpy scipy
```

Keeping your friends:
```
export MKL_NUM_THREADS=4
export NUMEXPR_NUM_THREADS=4
export OMP_NUM_THREADS=4
```
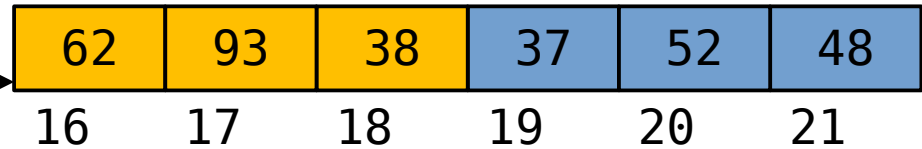
```python
import numpy as np

my_array = np.array([1, 2, 3])
```
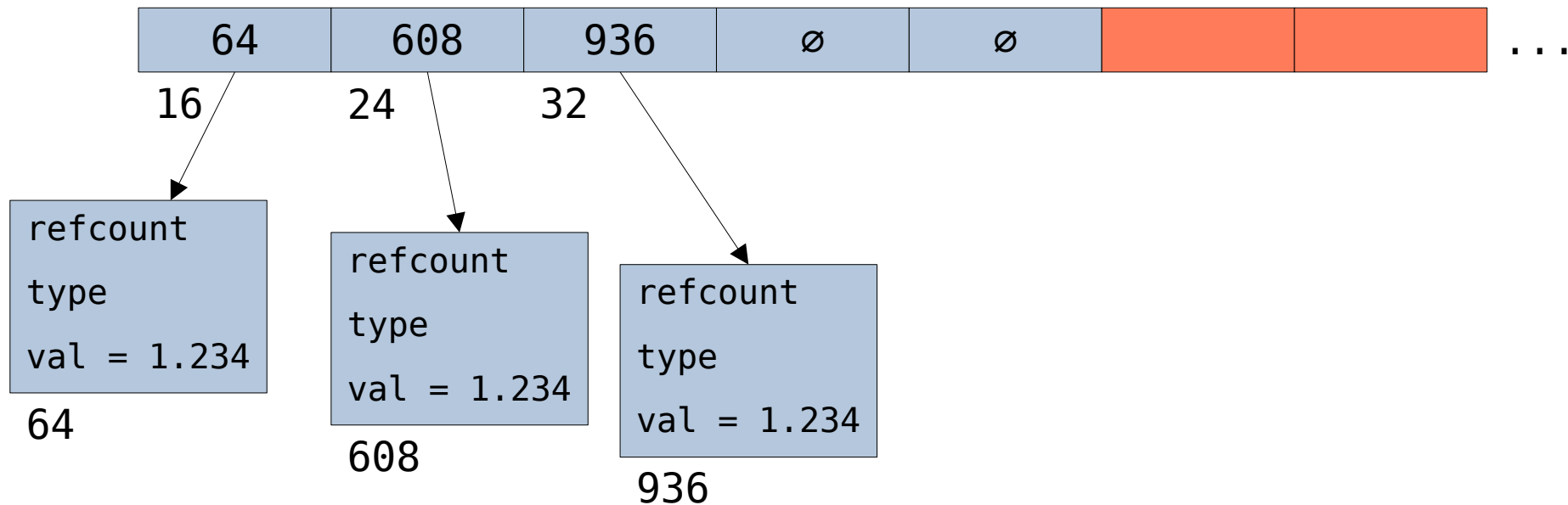
The numpy ndarray

```
a = np.array([[ 62 , 93 , 38 ],
              [ 37 , 52 , 48 ]])
```

dtype: "int8"
shape: (2, 3)
strides: (3, 1)
data: 16

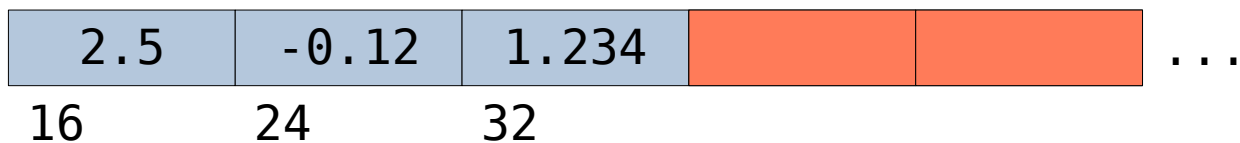| 62	93	38	37	52	48
16	17	18	19	20	21

- 1-D contiguous array of **fixed size** and **homogeneous type**

- metadata & access methods provide the interface of an N-D tensor
- dimensions are called "**axes**"
- very powerful **indexing** and **broadcasting**

Python list

| 64 | 608 | 936 | ∅ | ∅ | | | ... |

16   24   32

refcount

type

val = 1.234

64

refcount

type

val = 1.234

608

refcount

type

val = 1.234

936

Numpy array

| 2.5 | -0.12 | 1.234 | | | ... |

16   24   32

numpy flashcard

- **numpy array:**
  - memory buffer (C array)
  - metadata indicating how to traverse it
  - fixed size and (fixed) homogeneous type

Array creation

From a Python sequence: np.array([1, 2, 3])
From a range: np.arange(10)
With a fill value: np.zeros((5, 3)), np.ones(2, dtype="int")
Diagonal matrices: np.eye(3), np.diag([1, 2, 3])
With random values: np.random.random(3)

Mathematical operations

Element-wise operators: +, -, *, /, **, %, ==, <, ...
Matrix multiplication: @, np.ndarray.dot(), np.dot()
Transpose: .T, .transpose()
Aggregates: .sum(), .mean(), .max(), .var(), ...
"universal functions": np.exp(), np.sqrt(),
                           np.logical_or(), np.maximum(), ...

- create an array with 2 rows and 3 columns filled with 0.0

- create an array with 2 rows and 3 columns filled with 1

- create an array containing numbers from 0 to 999

- compute the difference between elements of 2 arrays of your choice

- what happens if the 2 arrays don't have the same shape?

- compute the inner product of [0, 1, 2] and [0, 1, 10]

- what are the default dtypes produced by np.zeros, np.arange, np.array?

- what happens if we try to store a number too large for the dtype?

Basic indexing

- Indexing 1D arrays works similar to Python sequences:
  a[10], a[2:5], a[::-1]

- For multi-dimensional arrays we can provide one index per
  axis:
  a[2, 3], a[:, 0]

- Create a 2D array and select the first row, then the
  second column

- What happens when we assign a value to a slice of the
  array? What about Python lists?

several views of the same buffer

my_array.T

dtype: "int8"

shape: (3, 2)

strides: (1, 3)

data: 16

62	37
93	52
38	48

my_array[::-1, :2]

dtype: "int8"

shape: (2, 2)

strides: (-3, 1)

data: 19

37	52
62	93

my_array

dtype: "int8"

shape: (2, 3)

strides: (3, 1)

data: 16

62	93	38
37	52	48

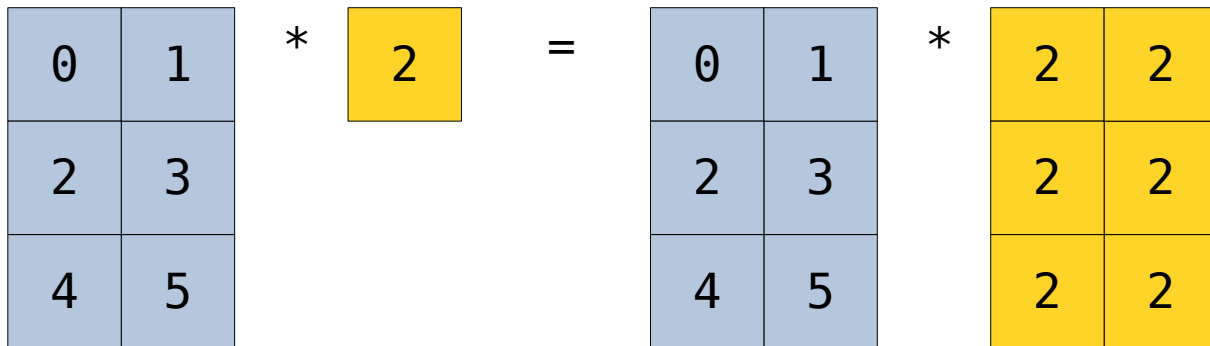| 62	93	38	37	52	48
16	17	18	19	20	21
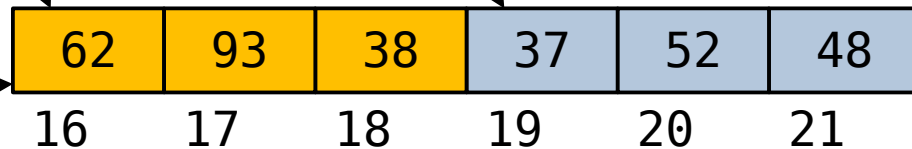
indexing, .reshape, .view, .copy, np.newaxis, np.expand_dims

numpy flashcard

- **numpy array:**
  - memory buffer (C array)
  - metadata indicating how to traverse it
  - fixed size and (fixed) homogeneous type
- **indexing:** with a tuple, one item per dimension (axis). items can be:
  - **basic indexing:**
    - integers: select one element
    - slices: select a range
    - '...' means ("insert as many ':' as necessary")
    - np.newaxis, None mean "insert a new axis of length 1"
- **reshaping:** '-1' means "the value that will make shapes match"

broadcasting

- What is the result of a * 2
  - when a is a Python list?
  - when a is a numpy array?

- Axes in which the length of the array is 1 can be expanded
- Dimensions can be added at the start of the array's shape

```
a: (3, 2)  ➡  a:      (3, 2)  ➡  a:              (3, 2)
2: (,)        [[2]]: (1, 1)       [[2, 2], ...]: (3, 2)
```

broadcasting

- Axes in which the length of the array is 1 can be expanded
- Dimensions can be added at the start of the array's shape

```
a: (3, 2)        a: (3, 2)           a: (3, 2)
b: (2,)     ➔    b: (1, 2)     ➔     b: (3, 2)
```

broadcasting

$$\begin{array}{|c|c|} 0 & 1 \\ 2 & 3 \\ 4 & 5 \end{array}$$ * $\begin{array}{|c|c|c|} 2 & 10 & 0 \end{array}$ = ?

- Axes in which the length of the array is 1 can be expanded
- Dimensions can be added at the start of the array's shape

a: (3, 2) → a: (3, 2)
b: (3,)   b: (1, 3)

🚫 Mismatch: 2 != 3

numpy flashcard

- **numpy array:**
  - memory buffer (C array)
  - metadata indicating how to traverse it
  - fixed size and (fixed) homogeneous type
- **indexing:** with a tuple, one item per dimension (axis). items can be:
  - **basic indexing:**
    - integers: select one element
    - slices: select a range
    - '...' means ("insert as many ':' as necessary")
    - np.newaxis, None mean "insert a new axis of length 1"
- **reshaping:** '-1' means "the value that will make shapes match"
- **broadcasting:**
  - prepend axes of length 1 to the shortest shape until ndims match
  - expand axes of length 1 until shapes match
  - perform elementwise operation

Experimenting with broadcasting

What will be the shape of the result if we apply an element-wise operator to **a** and **b** a if they have the following shapes:

- a: (3, 2), b: (2,)
- a: (3, 2), b: (3,)
- a: (3, 2), b: (3, 1)
- a: (3,),   b: (3, 1)
- a: (7, 5, 1), b: (1, 1, 4)

Adding higher-order features to a design matrix

We have a 5 x 2 design matrix: $((X_1),(X_2))$

We want to add higher-order features so it becomes:

$((X_1),(X_1)^2,(X_1)^3,(X_2),(X_2)^2,(X_2)^3)$

```python
x = np.arange(10).reshape(5, -1)
p = np.arange(3)

new_x = "???"
```

Advanced indexing

Basic indexing happens when the indexing object is a tuple of integers, slices or ellipsis.

Advanced indexing happens when the index along one axis is a sequence.

There are 2 types of advanced indexing:

- integers: specify a list of indices in each dimension
- Boolean arrays: extract the items where the index array is True

numpy flashcard

- **numpy array:**
  - memory buffer (C array)
  - metadata indicating how to traverse it
  - fixed size and (fixed) homogeneous type
- **indexing:** with a tuple, one item per dimension (axis). items can be:
  - **basic indexing:**
    - integers: select one element
    - slices: select a range
    - '...' means ("insert as many ':' as necessary")
    - np.newaxis, None mean "insert a new axis of length 1"
  - **advanced indexing:**
    - sequences of integers: select specific arbitrary elements
    - arrays of Booleans: select elements where the index object is True
- **reshaping:** '-1' means "the value that will make shapes match"
- **broadcasting:**
  - prepend axes of length 1 to the shortest shape until ndims match
  - expand axes of length 1 until shapes match
  - perform elementwise operation

Applying advanced integer indexing

- compute the trace of a matrix
- extract the corners of an array as is done by print(np.array(...))

Applying boolean indexing & broadcasting

- normalize the rows of a 2D array
- normalize the columns of a 2D array

```python
x = np.zeros((5, 3))
x[1:, 1:] = np.random.random(size=(4, 2))
```

Manipulating brain images with numpy

Basic steps of preparing images for analysis

- spatial smoothing
- resampling
- masking: extracting features

```python
import numpy as np
from matplotlib import pyplot as plt

images = np.load("images.npz")
plt.imshow(images["mni_template"], cmap="gray")
plt.show()
```

Images are stored in the "matrix" convention.
How to display it oriented in the "image" convention?
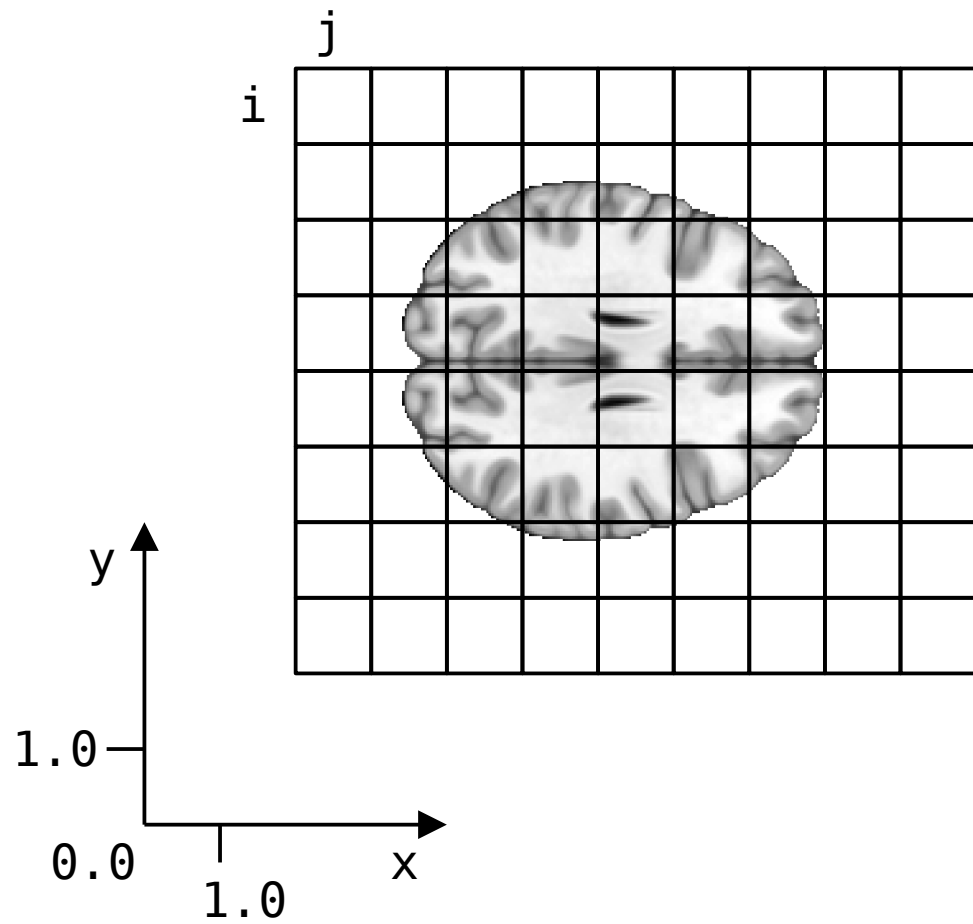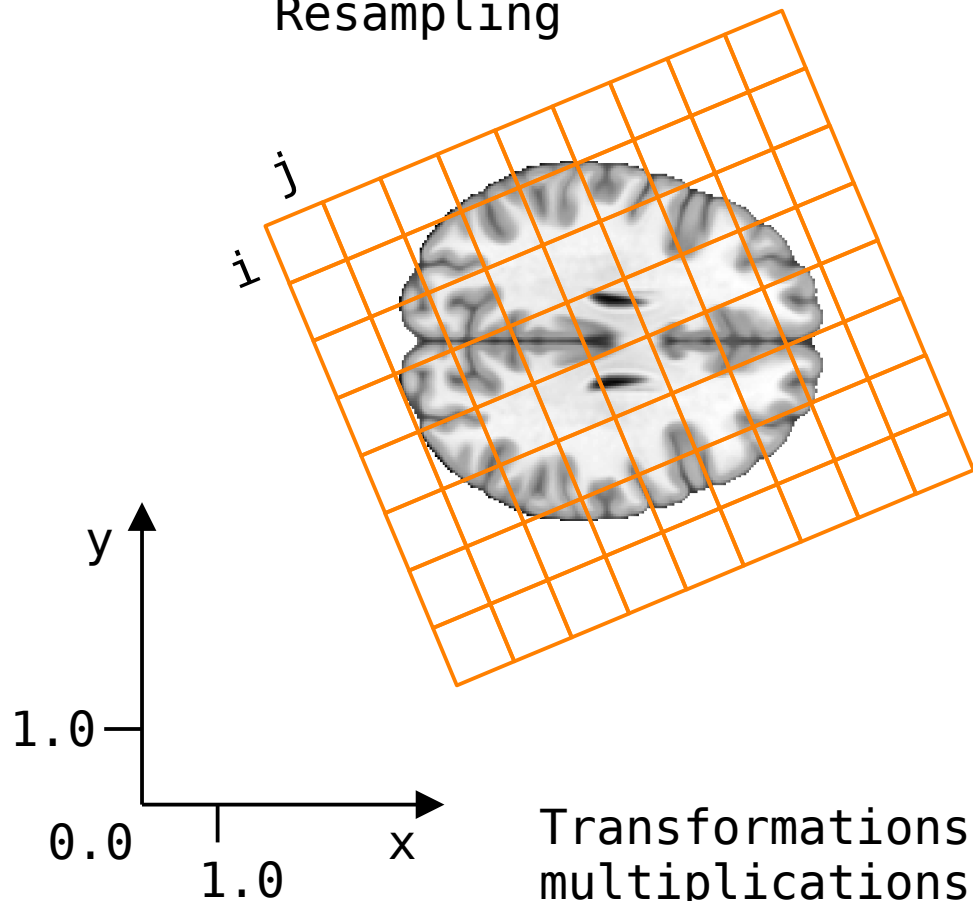
Spatial smoothing

exercises/questions/smoothing.py

Masking

exercises/questions/mask_transformations.py

Resampling

Resampling

Conversion between (i, j) indices
and (x, y) coordinates is an
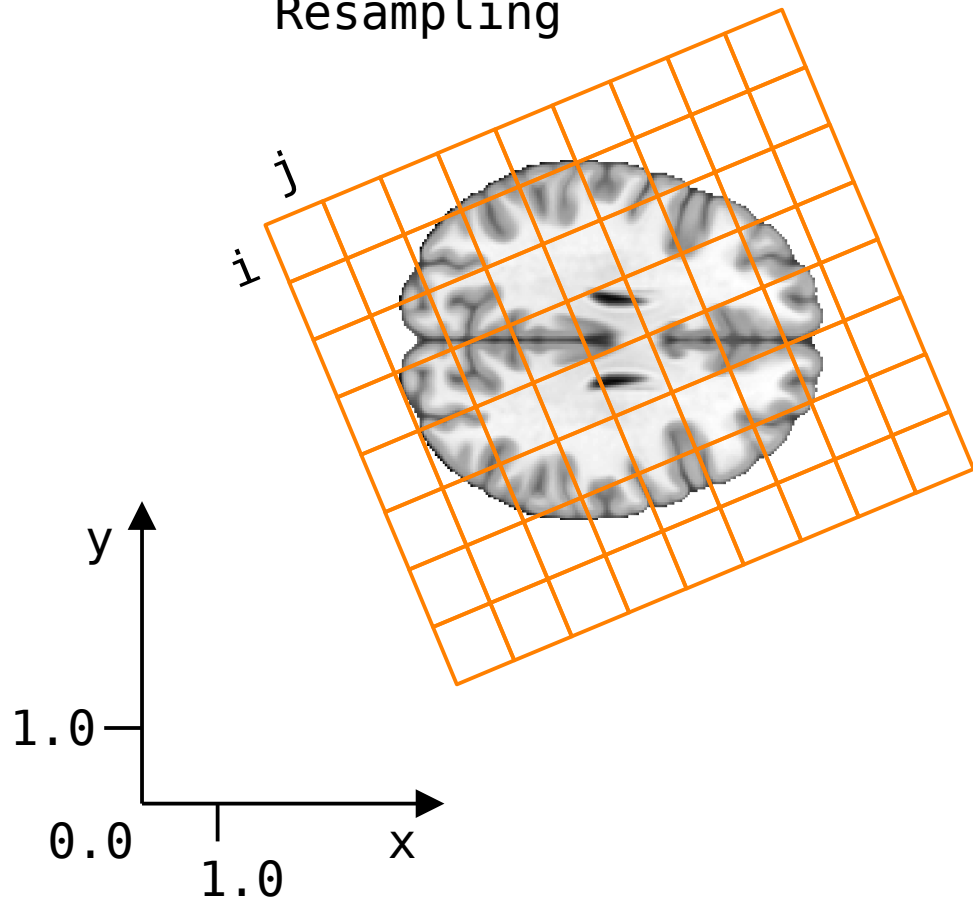affine transformation:

$(x, y) = A(i, j) + b$

Can be conveniently represented
with an augmented matrix

$$
\begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} A & b \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} i \\ j \\ 1 \end{bmatrix}
$$

Transformations can be composed by chaining matrix
multiplications

Bijections can be reversed by inverting the
transformation matrix

Resampling

Images are provided with an "affine" matrix that performs the conversion from pixels or voxels to a real-world coordinate system such as the MNI space.

Example: 2mm pixels, aligned with the unit vectors, and pixel (0, 0) at position (-21, -29):

$$A = \begin{matrix} 2 & 0 & -21 \\ 0 & 2 & -19 \\ 0 & 0 & 1 \end{matrix}$$

We can do the opposite conversion by multiplying by the inverse of A

This also allows aligning images that are sampled on different grids.

Resampling

exercises/questions/resampling.py

Putting together the whole processing pipeline

Now we have all the ingredients, choose either full_pipeline_mask.py or full_pipeline_atlas.py and apply all the necessary operations to our images:

- resample the mask or atlas to the functional images' grid
- smooth the functional images
- transform them to signals (timeseries) using the mask or atlas

You can them apply the inverse transformation to see the effect of this processing

Averaging timeseries after grouping by atlas region

exercises/questions/atlas_transformations.py