032CM - 2026

PROGRAMMING FOR COMPUTATIONAL CHEMISTRY

Working with NumPy arrays

Gianluca Levi

gianluca.levi@units.it, giale@hi.is

Office: Building C11, 3rd floor, Room 329

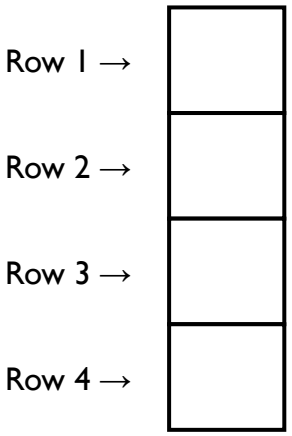
Fall 2026

What is a NumPy array?

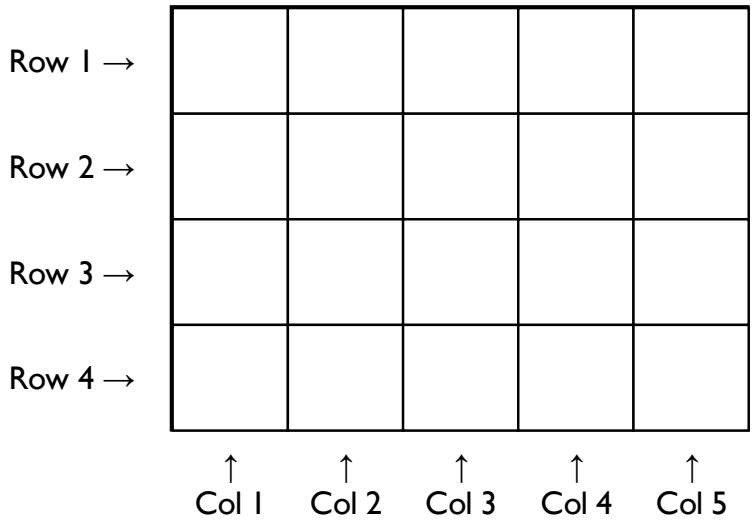
Multidimensional collection of values of the same type

- Naturally represents **scalars** (0-D), **vectors** (1-D), **matrices** (2-D), and higher-order "grids" of values.
- **Core attributes**
  - dtype — single data type of all elements (e.g., int, float).
  - shape — dimensions (e.g., (n,), (m, n), (m, n, l))
  - ndim — number of dimensions
  - size — total number of elements

1-D array (vector)



2-D array (matrix)



How to create NumPy arrays

np.array(seq) — Directly from sequence

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

np.zeros(shape, dtype=float) — all zeros

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

np.ones(shape, dtype=float) — all ones

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

np.full(shape, val, dtype=float) — constant fill

```
np.full((2,2), 99)
```

np.eye(n, dtype=float) — identity matrix

```
np.eye(3)
```

How to create NumPy arrays continued

np.arange([start], stop, [step]) — evenly spaced values (step)

```
np.arange(0, 10, 1)
```

np.linspace(start, stop, num) — evenly spaced values (number of points)

```
np.linspace(0, 1, 10)
```

np.random.random(size) — random floats in [0, 1)

```
np.random.random((2, 3))
```

np.empty(shape, dtype=float) — uninitialized (arbitrary values)

```
np.empty((4, 2))
```

np.zeros_like(a) / np.ones_like(a) — match shape/dtype of array

```
np.zeros_like(A)
np.ones_like(A)
```

Why using Numpy arrays instead of lists?

Lists

- Convenient for small, heterogeneous data
- Each element handled separately within loops (**slow** math)
- **Not well-suited to larger-scale numerical calculations**

Arrays

- Operations apply to **every element simultaneously**
- **No Python loops** needed

} **Vectorization**

Why using Numpy arrays instead of lists?

Lists

- Convenient for small, heterogeneous data
- Each element handled separately within loops (**slow** math)
- **Not well-suited to larger-scale numerical calculations**

Arrays

- Operations apply to **every element simultaneously**
- **No Python loops** needed

} **Vectorization**

Try it!

Calculate the values of the function $y = \sin(x)$ for **10000** evenly spaced values of x between 0 and 2π using (1) Python lists and loops, and (2) NumPy arrays with vectorized operations. Measure the time of execution in each case using %%timeit.

Operations with arrays

Mathematical operations are **vectorized**

- Executed element by element **without writing loops**
- Use fast C/Fortran code under the hood

Addition (elementwise)

```
c = a + b
d = np.add(a, b)     # equivalent
```

Multiplication vs. Matrix Product

```
a * b             # elementwise multiplication
np.multiply(a, b)   # equivalent

a @ b             # matrix multiplication
np.dot(a, b)        # equivalent
```

Try it!

In a Jupyter notebook, write a Python code to compute the **matrix product** of two arbitrary 2D NumPy arrays **using only for loops** and compare the result to NumPy built-in np.dot(). Before the calculation, check that the inner dimensions of the two arrays match.

Operations with arrays

Aggregation functions

- Summarize array values to provide quick **statistical summaries**
- **Fast and vectorized**

np.sum(arr) → sum of all elements

np.mean(arr) → average value

np.var(arr) → variance

np.std(arr) → standard deviation

np.min(arr) → minimum value

np.max(arr) → maximum value

np.argmin(arr) → index of min

np.argmax(arr) → index of max

Indexing and slicing arrays

Select a **single element**, **row**, or **column**

Extract **ranges or subarrays** using slice notation

Use **boolean conditions** to select elements that meet a criterion

```
a[1]          → second row
a[::2]        → every other row
a[:, 1]       → second column
a[2:, 1]      → all elements after the third in second column
idx = (a > 5) → boolean mask
a[idx]        → elements greater than 5
a[a > 5]      → equivalent
```

Assignment 3

Problem 4

Use the original Cartesian coordinates in bohr from the table in Problem 3 to compute:

(a) The distance (in bohr) between the oxygen atom and each hydrogen atom.

(b) The H–O–H bond angle (in degrees).

Hints:

1. The distance between two points in Cartesian coordinates, $\mathbf{r}_1 = (x_1, y_1, z_1)$ and $\mathbf{r}_2 = (x_2, y_2, z_2)$, is given by:
$$d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2 + (z_2 - z_1)^2}$$

2. To compute the bond angle, define two vectors $\mathbf{a}$ and $\mathbf{b}$ along the O–H bonds, both pointing away from the oxygen atom:
$$\mathbf{a} = \mathbf{r}_{\mathrm{H1}} - \mathbf{r}_{\mathrm{O}}, \qquad \mathbf{b} = \mathbf{r}_{\mathrm{H2}} - \mathbf{r}_{\mathrm{O}}$$
The bond angle θ can then be obtained from the dot product:
$$\cos(\theta) = \frac{\mathbf{a} \cdot \mathbf{b}}{|\mathbf{a}|\,|\mathbf{b}|}$$
where $|\mathbf{a}| = \sqrt{a_x^2 + a_y^2 + a_z^2}$ is the norm (or magnitude) of vector $\mathbf{a}$.

3. You can represent each atomic position as a NumPy array using `np.array([x, y, z])`. In the calculations, you can use `np.dot`, `np.linalg.norm`, and `np.arccos`.

4. The result of `np.arccos` is in radians. Convert it to degrees using `np.degrees`, or by multiplying by `180 / np.pi`.