NumPy Python array computing

START HERE

`import numpy as np`
The usual import

`a[a > 0]`
Filter with a mask

`np.array([1, 2, 3])`
Array from a list

`a.sum(axis=0)`
Reduce along an axis

`np.arange(<n>)`
0 .. n-1

`a @ b`
Matrix product

`a.reshape(<r>, <c>)`
Change the shape

`np.save("<file>.npy", a)`
Save an array

Create array, arange, zeros → `shape` → **Reshape** reshape, T, stack → `compute` → **Compute** ufuncs, broadcasting → `reduce` → **Reduce** sum, mean, argmax → `store` → **Save** save, savez, savetxt

CONCEPTS

An ndarray holds one dtype in a fixed-shape block. Whole-array expressions run in C, so replace Python loops with them.

Views vs copies
Basic slices are views: writing to a[1:3] changes a. Fancy (list or mask) indexing returns a copy. Use a.copy() to detach.

CREATE

`np.array([[1, 2], [3, 4]])`
Array from nested lists

`np.zeros((<r>, <c>))`
Array of zeros
np.ones and np.full((r, c), v) work alike.

`np.arange(<start>, <stop>, <step>)`
Evenly spaced by step
Stop is excluded. Float steps can surprise: prefer linspace.

`np.linspace(<a>, <b>, <n>)`
n points, both ends included

`np.zeros_like(<a>)`
Zeros with the shape and dtype of a

`np.array(<seq>, dtype=np.float32)`
Choose the dtype

INSPECT

`a.shape`
Tuple of axis lengths

`a.ndim / a.size`
Number of axes / of elements

`a.dtype`
Element type

`a.astype(np.float64)`
Convert type (makes a copy)

Common dtypes

dtype	Meaning
`int64`	default integer (Linux, macOS, Windows NumPy 2)
`float64`	default float
`bool`	True / False
`uint8`	0..255, e.g. image pixels
`datetime64[D]`	calendar dates

INDEX AND SLICE

`a[<i>, <j>]`
One element

INDEX AND SLICE (cont.)

`a[<a>:<b>:<step>]`
Slice (end excluded)

`a[:, <j>]`
One column

`a[<i>, :]`
One row

`a[a > <x>]`
Elements matching a mask
Returns a flat 1-D copy.

`a[[<i>, <j>, <k>]]`
Pick rows by index list
Result is a copy, in the order given.

`a[:, np.newaxis]`
Add an axis of length 1

`np.where(<cond>, <x>, <y>)`
x where true, else y
With only a condition it returns the indices.

`np.clip(<a>, <lo>, <hi>)`
Limit values to a range

SHAPE AND JOIN

`a.reshape(<r>, -1)`
Reshape, -1 infers one length
Needs the same element count; usually returns a view.

`a.T`
Transpose

`a.ravel() / a.flatten()`
Flatten (view if possible / copy)

`np.concatenate([<a>, <b>], axis=0)`
Join along an existing axis

`np.vstack([<a>, <b>]) / np.hstack([<a>, <b>])`
Stack vertically / horizontally

MATH AND BROADCASTING

`a + b, a * 2, a ** 2`
Elementwise arithmetic

`np.sqrt(a), np.exp(a), np.log(a)`
Elementwise functions

`a > <x>, a == b`
Elementwise comparison gives a bool array

`np.isclose(<a>, <b>)`
Float comparison within tolerance
Never test floats with ==.

`np.isnan(a)`
Mask of NaN values

MATH AND BROADCASTING (cont.)

Broadcasting rules

Shapes	Result
`(3, 4) + (4,)`	(3, 4)
`(3, 1) + (1, 4)`	(3, 4)
`(3, 4) + (3,)`	error

Broadcasting
Shapes line up from the right. Each pair of lengths must be equal or one of them 1, which is stretched. Use a[:, None] to fix a mismatch.

REDUCTIONS

`a.sum() / a.mean()`
Total / average of everything

`a.sum(axis=0)`
Collapse rows: one value per column
axis=1 gives one value per row.

`a.min() / a.max()`
Smallest / largest

`a.argmax() / a.argmin()`
Index of largest / smallest (flat)
Add axis=0 for per-column positions.

`a.std() / a.var()`
Standard deviation / variance
ddof=1 gives the sample estimate.

`np.median(a)`
Median

`np.any(<mask>) / np.all(<mask>)`
Any / all True

`np.nansum(a) / np.nanmean(a)`
Reductions that skip NaN

SORT AND SEARCH

`np.sort(a)`
Sorted copy
axis=-1 by default; a.sort() sorts in place.

`np.argsort(a)`
Indices that would sort a

`np.unique(a, return_counts=True)`
Distinct values and their counts

LINEAR ALGEBRA

`a @ b`
Matrix product

`np.linalg.inv(<a>)`
Matrix inverse

LINEAR ALGEBRA (cont.)

`np.linalg.solve(<a>, <b>)`
Solve a x = b
Prefer it over inv(a) @ b.

`np.linalg.eig(<a>)`
Eigenvalues and eigenvectors

`np.linalg.norm(<v>)`
Vector length (2-norm)

RANDOM

`rng = np.random.default_rng(<seed>)`
Create a seeded generator
Use this instead of the legacy np.random.seed.

`rng.random((<r>, <c>))`
Uniform floats in [0, 1)

`rng.integers(<lo>, <hi>, size=<n>)`
Random ints, hi excluded

`rng.normal(<mu>, <sigma>, size=<n>)`
Normal samples

`rng.choice(<a>, size=<n>, replace=False)`
Sample from an array

`rng.shuffle(<a>)`
Shuffle in place

SAVE AND LOAD

`np.save("<file>.npy", <a>)`
Write one array (binary)

`np.load("<file>.npy")`
Read it back

`np.savez("<file>.npz", <name>=<a>)`
Several arrays in one file

`np.savetxt("<file>.csv", <a>, delimiter=",")`
Write plain text

`np.loadtxt("<file>.csv", delimiter=",")`
Read plain text numbers
skiprows=1 skips a header; use pandas for messy files.

RECIPES

Normalise columns

```
X = rng.normal(size=(100, 3))
Z = (X - X.mean(axis=0)) /
X.std(axis=0)
print(Z.mean(axis=0).round(6))
```

Pairwise distances by broadcasting

```
d = np.linalg.norm(
  p[:, None, :] - p[None, :, :],
  axis=-1)    # (n, n)
```