

Numpy

Aayush Bajaj

July 10, 2026

Contents

1	the ndarray model	1
2	views vs copies	2
3	broadcasting	3
4	vectorisation: why the speed	4
5	fancy vs boolean indexing	4
6	einsum, linalg, random	5
7	memory layout and performance	6
8	see also	6

numpy is one idea executed ruthlessly: a typed, contiguous block of memory plus enough metadata to *reinterpret* it without copying. everything else — broadcasting, views, vectorised speed — falls out of that design. learn the memory model and the api stops being a list of functions and becomes a handful of consequences. all timings and outputs on this page are real (python 3, numpy 2.4, apple m-series).

official docs:
<https://numpy.org/doc/stable/>
— the ‘numpy fundamentals’
section is genuinely good,
unusual for reference docs

1 the ndarray model

an **ndarray** is four things: a **buffer** (one flat run of bytes), a **dtype** (how many bytes per element and what they mean), a **shape** (how to pretend the flat buffer is n-dimensional), and **strides** (how many *bytes* to jump to advance one step along each axis). element (i, j) of a 2-d array lives at

$$\text{offset}(i, j) = i \cdot s_0 + j \cdot s_1 \quad \text{bytes,} \quad (1)$$

concretely, for a 3×4 array of **int32** in C order: one row is $4 \times 4 = 16$ bytes, so strides are $(16, 4)$:

```
1 import numpy as np
2
3 a = np.arange(12, dtype=np.int32).reshape(3, 4)
4 print(a)
5 print("dtype:", a.dtype, " shape:", a.shape, " strides:", a.strides)
6 print("itemsize:", a.itemsize, " total bytes:", a.nbytes)
7 # address arithmetic: element [i, j] lives at offset i*16 + j*4
```

```

8 i, j = 1, 2
9 offset = i * a.strides[0] + j * a.strides[1]
10 print(f"a[{i},{j}] = {a[i,j]}, byte offset = {offset}")
11 # transpose costs nothing: same buffer, strides swapped
12 print("a.T strides:", a.T.strides, " (no data moved)")

```

```

[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]]
dtype: int32  shape: (3, 4)  strides: (16, 4)
itemsize: 4  total bytes: 48
a[1,2] = 6, byte offset = 24
a.T strides: (4, 16)  (no data moved)

```

transpose, slicing, **reshape** (usually), **ravel** (usually) are all *stride tricks* — new metadata over the same 48 bytes. that “usually” is the subject of the next section.

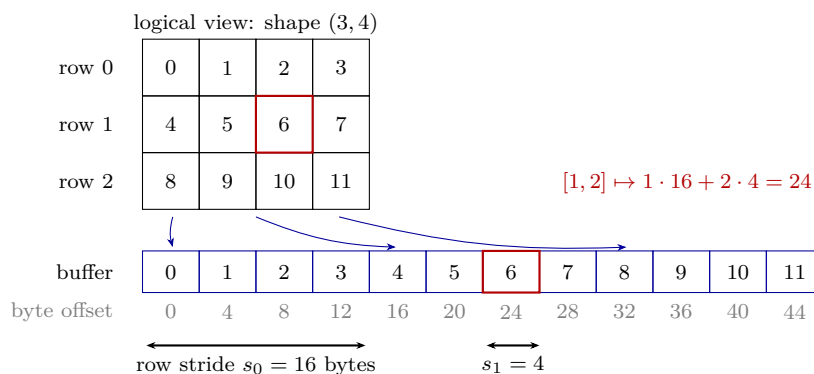


Figure 1: the strides picture: a 3×4 int32 array is really 48 flat bytes. shape says how to fold them; strides (16, 4) say a row step is 16 bytes and a column step is 4. element $[1, 2]$ sits at byte $1 \cdot 16 + 2 \cdot 4 = 24$.

2 views vs copies

the rules, then the gotchas:

- **basic slicing** (`a[2:7]`, `a[:, 0]`, `a[:, :2]`) returns a **view** — new strides, same buffer. writing to it writes to the original.
- **fancy and boolean indexing** return a **copy** — the selected elements need not be evenly strided, so a fresh buffer is unavoidable.
- **reshape/ravel** return a view when the requested layout is reachable by strides alone, and silently *copy* when it is not (classic case: reshaping a transpose).
- `v.base` tells the truth: non-None means view.

```

1 import numpy as np
2
3 a = np.arange(10)
4 v = a[2:7]           # basic slice: a VIEW
5 v[0] = 99           # writes through to a
6 print("after v[0]=99, a =", a)
7 print("v.base is a:", v.base is a)
8
9 f = a[[2, 3, 4]]     # fancy indexing: a COPY
10 f[0] = -1

```

```

11 print("after f[0]=-1, a =", a, " (unchanged)")
12
13 b = a.reshape(2, 5)      # reshape of contiguous data: view
14 print("b.base is a:", b.base is a)
15
16 m = np.arange(6).reshape(2, 3)
17 t = m.T.reshape(6)       # transpose is non-contiguous: this line copies
18 t[0] = 42
19 print("m unchanged by t[0]=42:", m.ravel())

```

```

after v[0]=99, a = [ 0  1 99  3  4  5  6  7  8  9]
v.base is a: True
after f[0]=-1, a = [ 0  1 99  3  4  5  6  7  8  9] (unchanged)
b.base is a: True
m unchanged by t[0]=42: [0 1 2 3 4 5]

```

the two directions of the gotcha: a view you thought was a copy *corrupts your source data*; a copy you thought was a view *silently drops your writes* (the `t[0] = 42` line above — no error, no effect on `m`). when it matters, say what you mean: `.copy()` explicitly, or `np.shares_memory(a, v)` to check.

3 broadcasting

the rules, stated precisely. to combine arrays elementwise, numpy compares shapes *right-aligned*, axis by axis:

- **rule 1** — pad the shorter shape with 1s on the *left*.
- **rule 2** — two axis lengths are compatible iff they are equal, or one of them is 1.
- **rule 3** — an axis of length 1 is *stretched* (stride 0, no data copied) to match the other.

so (3,1) with (4,): pad to (1,4), compare (3,1) vs (1,4) — both axes reconcile, result (3,4). an outer product without a loop:

```

1 import numpy as np
2
3 # broadcasting: right-align shapes, each axis must match or be 1
4 col = np.array([[0], [10], [20]])      # shape (3, 1)
5 row = np.array([1, 2, 3, 4])          # shape (4,) -> (1, 4)
6 print("col (3,1) + row (4,) ->", (col + row).shape)
7 print(col + row)
8
9 # standardising a data matrix: (5,3) - (3,) works, (5,3) - (5,) does not
10 X = np.arange(15.0).reshape(5, 3)
11 mu = X.mean(axis=0)                    # shape (3,)
12 print("\nX - mu shape:", (X - mu).shape)
13 try:
14     X - X.mean(axis=1)                  # shape (5,) -- misaligned!
15 except ValueError as e:
16     print("X - X.mean(axis=1):", e)
17 print("fix with keepdims -> shape", (X - X.mean(axis=1, keepdims=True)).shape)

```

```

col (3,1) + row (4,) -> (3, 4)
[[ 1  2  3  4]
 [11 12 13 14]
 [21 22 23 24]]

```

X - mu shape: (5, 3)

X - X.mean(axis=1): operands could not be broadcast together with shapes (5,3) (5,) fix with keepdims -> shape (5, 3)

the failed case is the canonical trap: a per-row mean has shape (5,), which right-aligns against the *columns* axis. `keepdims=True` preserves the (5,1) shape so the subtraction broadcasts down the correct axis.

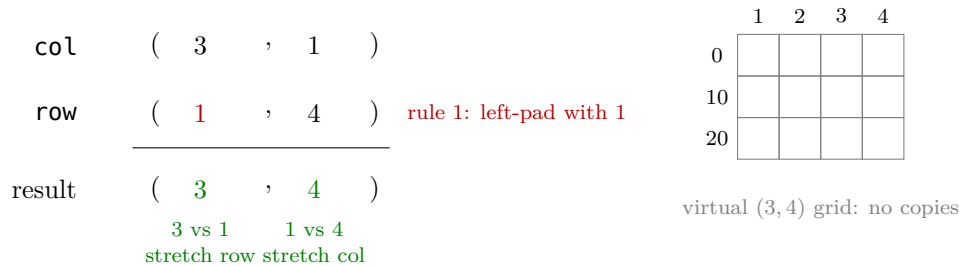


Figure 2: broadcasting shape alignment for (3,1) + (4,): shapes are compared right-aligned; missing axes are padded with 1; every 1 stretches (stride 0) to match. result (3,4).

4 vectorisation: why the speed

a python loop pays interpreter overhead — type dispatch, boxing, reference counting — *per element*. a numpy ufunc pays it *once*, then runs a compiled C loop over the buffer (with simd and cache-friendly access). the difference is not subtle:

```
1 import numpy as np, timeit
2
3 n = 1_000_000
4 xs = list(range(n))
5 arr = np.arange(n, dtype=np.int64)
6
7 t_loop = timeit.timeit(lambda: sum(x * x for x in xs), number=10) / 10
8 t_np = timeit.timeit(lambda: np.dot(arr, arr), number=10) / 10
9 print(f"python loop: {t_loop*1e3:8.2f} ms")
10 print(f"numpy dot: {t_np*1e3:8.2f} ms")
11 print(f"speedup: {t_loop/t_np:8.0f}x")
```

python loop: 30.52 ms
numpy dot: 0.47 ms
speedup: 65x

the working rule: if you wrote `for` over array elements, you probably meant a ufunc, a reduction, or an einsum. loops over *axes* (a handful of iterations) are fine; loops over *elements* (millions) are the bug.

5 fancy vs boolean indexing

two different machines behind similar syntax:

- **fancy** (integer-array) indexing selects by *position*: `a[[0,2,3]]` gathers rows. with one index array per axis, arrays are paired elementwise — `a[[0,1],[2,3]]` is `[a0,2, a1,3]`, *not* a 2×2 submatrix (that wants `np.ix_`).

- **boolean** indexing selects by *predicate*: `a[a > 5]` flattens the kept elements. as an assignment target it is the idiomatic in-place filter.

```

1 import numpy as np
2 rng = np.random.default_rng(0)
3
4 a = rng.integers(0, 10, (4, 5))
5 print("fancy rows [0,2,3]: shape", a[[0, 2, 3]].shape)
6 print("fancy pairs a[[0,1],[2,3]] =", a[[0, 1], [2, 3]], " (elementwise
  ↳ pairing!)")
7 mask = a > 5
8 print("boolean mask keeps", mask.sum(), "elements ->", a[mask][:6], "...")
9 a[a > 8] = 8 # in-place clamp via boolean assignment
10 print("after clamp, max =", a.max())

```

```

fancy rows [0,2,3]: shape (3, 5)
fancy pairs a[[0,1],[2,3]] = [5 1] (elementwise pairing!)
boolean mask keeps 10 elements -> [8 6 8 6 9 6] ...
after clamp, max = 8

```

6 einsum, linalg, random

`einsum` is index notation as an api: name the axes, repeat a label to multiply along it, omit a label from the output to sum it away. three canonical spellings, each verified against the library primitive:

once einsum clicks, half of your reshapes, transposes and newaxis insertions evaporate

```

1 import numpy as np
2 rng = np.random.default_rng(42)
3
4 A, B = rng.standard_normal((3, 4)), rng.standard_normal((4, 5))
5 M = rng.standard_normal((4, 4))
6 batch = rng.standard_normal((10, 3, 4))
7
8 # 1. matmul: contract the shared index j
9 print("matmul ok:", np.allclose(np.einsum("ij,jk->ik", A, B), A @ B))
10 # 2. trace: repeated index on one operand sums the diagonal
11 print("trace ok:", np.isclose(np.einsum("ii->", M), np.trace(M)))
12 # 3. batch matmul: batch index b comes along for the ride
13 out = np.einsum("bij,jk->bik", batch, B)
14 print("batch ok:", out.shape == (10, 3, 5) and np.allclose(out, batch @ B))
15
16 # linalg essentials
17 K = np.array([[3.0, 1.0], [1.0, 2.0]])
18 b = np.array([9.0, 8.0])
19 x = np.linalg.solve(K, b) # never invert; solve
20 w, V = np.linalg.eigh(K) # symmetric -> eigh
21 print("solve:", x, " residual:", np.linalg.norm(K @ x - b))
22 print("eigenvalues:", np.round(w, 4))
23
24 # random Generator API: seeded, local, no global state
25 g1, g2 = np.random.default_rng(7), np.random.default_rng(7)
26 print("reproducible:", np.array_equal(g1.normal(size=3), g2.normal(size=3)))

```

```

matmul ok: True
trace ok: True
batch ok: True
solve: [2. 3.] residual: 0.0
eigenvalues: [1.382 3.618]
reproducible: True

```

linalg house rules: `solve` over `inv` (faster, numerically saner), `eigh` over `eig` for symmetric matrices, `lstsq` for rectangular systems, and remember `@` is `matmul`. for randomness, the modern api is `np.random.default_rng(seed)` — a local, explicitly-seeded `Generator` — not the legacy global-state api (`np.random.seed` plus module-level draws), whose hidden global state makes reproducibility a whole-program property instead of a local one.

7 memory layout and performance

C order (default) stores rows contiguously; fortran order stores columns. the layout is invisible to semantics and very visible to the cache:

```

1 import numpy as np, timeit
2 t = lambda f, n: timeit.timeit(f, number=n) / n
3
4 a = np.ones((4000, 4000)) # C order: each row is 32 KB of contiguous memory
5 print("copy one row a[0,:]: %6.1f us" % (t(lambda: a[0, :].copy(), 100) *
6     ↳ 1e6))
7 print("copy one col a[:,0]: %6.1f us" % (t(lambda: a[:, 0].copy(), 100) *
8     ↳ 1e6))
9 print("ravel order='C' (view): %8.4f ms" % (t(lambda: a.ravel(order="C"), 100) *
10    ↳ 1e3))
11 print("ravel order='F' (copy): %8.2f ms" % (t(lambda: a.ravel(order="F"), 10) *
12    ↳ 1e3))
13
14 b = np.ones((2000, 2000))
15 bt = b.T.copy()
16 print("b + b.T (strided walk): %6.2f ms" % (t(lambda: b + b.T, 20) * 1e3))
17 print("b + bt (contiguous) : %6.2f ms" % (t(lambda: b + bt, 20) * 1e3))

```

```

copy one row a[0,:]:      0.6 us
copy one col a[:,0]:     37.8 us
ravel order='C' (view):   0.0001 ms
ravel order='F' (copy):   74.64 ms
b + b.T (strided walk):   6.48 ms
b + bt (contiguous) :    2.75 ms

```

reading a column of a C-order array touches one useful element per cache line — sixty times slower than the row it crosses. `ravel` in the array's own order is free (a view); against the grain it is a 74 ms full copy. mixing a transposed view into arithmetic (`b + b.T`) more than doubles the cost of the add. the moral: match your *iteration order* to your *storage order*, and when an algorithm insists on column access, pay for one explicit `np.asfortranarray` up front instead of strided reads forever.

this is also why matrix libraries care about blas layouts: gemm kernels are layout-specialised down to the simd register

8 see also

- `pandas` — labelled axes bolted onto these same buffers
- `matplotlib` — where the arrays go to be seen
- `mastering numpy` — 100 exercises to make the above reflexive
- `quadratic programming` — `np.linalg.solve` earning its keep on KKT systems