# Arrays and broadcasting

## Slicing, views, the dot, loop fusion, comprehensions

### Construction

```
[1, 2, 3]                       # Vector{Int64} (3-element, 1-D)
[1 2 3]                         # 1x3 Matrix — spaces make columns
[1 2; 3 4]                      # 2x2 Matrix — semicolons make rows
[1, 2, 3]'                      # adjoint: 1x3 row (lazy wrapper, not a copy)

zeros(3); zeros(Int, 2, 2); ones(2, 3); fill(7, 2, 2)
trues(3); falses(3)
Vector{Float64}(undef, 100)     # uninitialised — fastest, contains garbage
similar(A); similar(A, Float64); similar(A, 3, 3)
rand(3, 3); randn(100); rand(1:6, 10)
collect(1:5); range(0, 1; length = 11); range(0, 1; step = 0.1)
reshape(1:12, 3, 4)
repeat([1,2], 3); repeat([1 2], 2, 2)
Int[]                           # typed empty. NOT []
```

`[]` alone is `Vector{Any}` — a common accidental performance sink.

### Indexing

```
v[1]; v[end]; v[end-1]; v[2:end]; v[[1,3]]; v[v .> 2]     # boolean mask
A[i, j]; A[:, j]; A[i, :]; A[1:2, 2:3]; A[:]              # A[:] flattens (column-major)
A[CartesianIndex(1,2)]
first(v); last(v); v[begin]                                # `begin` works as an index too
```

Assignment mirrors it, with broadcast on the right-hand side:

```
A[:, 1] .= 0            # fill a column
A[:, 1] = zeros(3)      # same, but allocates the RHS
v[v .< 0] .= 0          # clamp negatives
```

**trap:** `A[:, 1] = 0` (scalar, no dot) is an error. Use `.=` when assigning a scalar into a slice.

### Views vs copies

```
B = A[:, 2:3]           # COPY
B = @view A[:, 2:3]     # view (SubArray), shares memory
B = view(A, :, 2:3)     # function form

@views function f(A)    # every slicing operation in this function becomes a view
    s = 0.0
    for j in axes(A, 2)
        s += sum(A[:, j])
    end
    s
end
```

`@views` on a function or block is usually the right move in numerical code. The exception: if you're going to keep the result around, a view holds the whole parent array alive.

### Iteration

```
for x in v end                      # elements
for i in eachindex(v) end           # correct indices for ANY array type
for (i, x) in pairs(v) end          # index => element
for (i, x) in enumerate(v) end      # 1,2,3... regardless of actual indices
for c in eachcol(A) end
for r in eachrow(A) end
for I in CartesianIndices(A) end    # A[I] works, I[1], I[2] are the components
```

```
for (a, b) in zip(v, w) end
axes(A, 1); size(A); size(A, 2); length(A); ndims(A)
```

Prefer `eachindex`/`axes` over `1:length(v)`. It's correct for views, offset arrays, and sparse arrays, and it enables bounds-check elision.

## Growing and mutating

```
push!(v, x); pop!(v)
pushfirst!(v, x); popfirst!(v)
append!(v, w); prepend!(v, w)
insert!(v, 2, x); deleteat!(v, 2); splice!(v, 2)
resize!(v, 10); empty!(v)
sizehint!(v, 10_000)          # preallocate capacity before a push! loop
```

Only `Vector` grows; matrices are fixed-size (use `vcat`/`hcat` or build a vector of rows and `reduce(hcat, ...)`).

## Combining

```
vcat(a, b); hcat(a, b); cat(a, b; dims = 3)
[a; b]        # vcat
[a b]         # hcat
[a b; c d]    # block matrix
reduce(vcat, list_of_vectors)     # much faster than repeated vcat in a loop
stack(list_of_vectors)            # 1.9+: list of vectors -> matrix
```

## Broadcasting: the dot

Any function becomes elementwise with a dot, and **adjacent dotted operations fuse into a single loop with no temporary arrays**.

```
x = [1.0, 2.0, 3.0]
sqrt.(x)
x .+ 1
x .* y
f.(x, y)

z = 2 .* x .+ 1 .- sqrt.(x)     # ONE pass, ONE allocation (the result)
@. z = 2x + 1 - sqrt(x)         # dots everything; writes into z, ZERO allocation
```

The fusion is the point. In NumPy, `2*x + 1 - np.sqrt(x)` builds three temporaries; in Julia it builds none.

Shape rules: dimensions of length 1 are stretched.

```
[1,2,3] .+ [10 20]        # 3x2 matrix — column plus row is an outer operation
A .- mean(A; dims = 1)    # centre each column
```

Protect an argument you *don't* want iterated with `Ref` or a 1-tuple:

```
in.([1,2,3], Ref([1,5]))          # elementwise membership in ONE set -> [true,false,false]
f.(x, Ref(some_big_config))
```

In-place broadcast:

```
x .= 0.0
y .= f.(x)          # writes into existing y
x .+= 1             # in place
x = x .+ 1          # allocates a new array — not the same thing
```

Custom types opt in via `Base.broadcastable`; scalars-by-default via `Base.broadcastable(x::MyType) = Ref(x)`.

## Comprehensions and generators

```
[x^2 for x in 1:10]
[x^2 for x in 1:10 if isodd(x)]
[i*j for i in 1:3, j in 1:4]              # 3x4 Matrix — multiple `for` make dimensions
[(i,j) for i in 1:3 for j in 1:i]         # nested (dependent), flattens to a Vector
Dict(k => f(k) for k in keys)
Set(x % 3 for x in 1:10)
```

```
Float64[x for x in 1:3]                  # force the element type

(x^2 for x in 1:10)                      # generator: lazy, allocates nothing
sum(x^2 for x in 1:10)                   # consumed without materialising
```

Use a generator whenever the result is immediately reduced.

## The functional toolkit

```
map(f, v); map(f, v, w); map!(f, dest, v)
filter(f, v); filter!(f, v)
reduce(+, v); reduce(vcat, vs); foldl; foldr
mapreduce(abs2, +, v)
sum, prod, maximum, minimum, extrema, count
sum(abs2, v)                         # function-first forms avoid a temporary
any(iseven, v); all(>(0), v)
findfirst(iseven, v)                 # index or nothing
findlast, findall, findmax, findmin
argmax(v); argmax(f, v)              # 1.7+: argmax over f's values, returns the input
sort(v; by, lt, rev, alg); sort!; sortperm(v); partialsort(v, 1:3)
unique(v); unique(f, v); allunique
union, intersect, setdiff, symdiff
zip, enumerate, pairs
Iterators.take(it, 5); drop; partition(v, 3); flatten; product; cycle; countfrom
Iterators.filter, Iterators.map          # lazy versions
```

## Dicts and sets

```
d = Dict("a" => 1, "b" => 2)
d = Dict{String,Int}()
d["c"] = 3
d["a"]                       # KeyError if missing
get(d, "z", 0)               # default
get!(d, "z", 0)              # default AND insert
haskey(d, "a"); delete!(d, "a"); pop!(d, "a", default)
keys(d); values(d); collect(pairs(d))
for (k, v) in d ... end      # order is unspecified and unstable between runs
merge(d1, d2); mergewith(+, d1, d2)

s = Set([1,2,3]); push!(s, 4); 3 in s; union(s1, s2)
```

For insertion-ordered dicts use `OrderedDict` from DataStructures.jl. `Base` also has `IdDict` (identity keys) and `WeakKeyDict`.

## Missing pieces worth knowing exist

- `SparseArrays` (stdlib): `sparse(I, J, V)`, `spzeros`, and the whole LinearAlgebra stack dispatches on them.
- `StaticArrays.jl`: `SVector{3,Float64}` — stack-allocated fixed-size arrays, 10x faster for small vectors. Use for 2-D/3-D geometry.
- `OffsetArrays.jl`: arbitrary index bases, which is why you write `eachindex` instead of `1:length`.
- `StructArrays.jl`: array-of-structs interface over a struct-of-arrays layout.