Gotchas

Where Julia contradicts your instincts. 1-based, column-major, silent overflow, copies vs views

Places where the correct Julia answer is not the one your fingers will type.

Indexing and memory layout

```
a = [1, 2, 3]
a[1]              # 1-based
a[end]; a[end-1]; a[2:end]
a[0]              # BoundsError

A = rand(1000, 1000)
A[:, 1]           # a column — CONTIGUOUS in memory, fast
A[1, :]           # a row — strided, slow
```

Julia is **column-major**, like Fortran, MATLAB and R; not row-major like C, NumPy or Go. Consequences:

- The innermost loop should vary the *first* index.
- `A[:, j]` is the cheap slice.
- Reshaping and flattening go down columns: `vec([1 2; 3 4]) == [1, 3, 2, 4]`.

```
for j in axes(A, 2), i in axes(A, 1)    # correct order
    A[i, j] = f(i, j)
end
```

Copies, references and views

```
a = [1, 2, 3]
b = a              # SAME object
b[1] = 99          # a is now [99, 2, 3]
c = copy(a)        # shallow copy
d = deepcopy(a)    # recursive

s = a[2:3]         # slicing ALLOCATES A COPY (unlike NumPy)
v = @view a[2:3]   # a view, no allocation, writes through to a
@views begin ... end   # every slice in this block becomes a view
```

trap: coming from NumPy you expect slices to be views; coming from Go you expect subslices to share. Julia copies. In hot code this is the number one source of surprise allocation.

Integer arithmetic

```
7 / 2         # 3.5 — always Float64, even Int/Int
7 ÷ 2         # 3   — integer division, div(7,2)
7 % 2         # 1   — rem, sign follows the dividend
mod(-7, 2)    # 1   — sign follows the divisor; rem(-7, 2) == -1
2^10          # 1024 — ^ is power
typemax(Int) + 1   # typemin(Int). SILENT WRAPAROUND, no exception.
```

If overflow matters, use `BigInt`, `Int128`, `SafeInt` from SaferIntegers.jl, or `Base.Checked.checked_add`.

`^` on integers with a negative exponent throws — `2^-1` is a `DomainError`. Write `2.0^-1` or `1//2`.

Equality

Operator	Semantics	`NaN`	Use for
`==`	value equality, may be `missing`	`NaN == NaN` is false	general comparison
`===`	identity / bit-identical	true for the same object	checking "same object"
`isequal`	like == but total; hash-consistent	`isequal(NaN, NaN)` is true	Dict keys, sorting
≈ (`isapprox`)	approximate float equality	—	tests, numerics

```
0.1 + 0.2 == 0.3      # false
0.1 + 0.2 ≈ 0.3       # true (\approx<tab>)
isapprox(a, b; atol = 1e-8, rtol = 1e-5)
```

Strings are UTF-8 and indexed by byte

```
s = "héllo"
length(s)        # 5 characters
ncodeunits(s)    # 6 bytes
s[1]             # 'h'
s[2]             # 'é'  (2 bytes)
s[3]             # StringIndexError — index 3 is mid-character
s[nextind(s, 2)] # 'l' — the correct way to step
collect(s)       # Vector{Char} if you need O(1) random access
for c in s ... end   # iteration is always character-wise and correct
```

Also: `"a" * "b"` concatenates; `+` on strings is an error. Repetition is `"ab"^3`.

Scope in loops at top level

```
# in a script:
count = 0
for i in 1:3
    count += 1        # UndefVarError — `count` here is a new local
end
```

Fix with `global count += 1`, or wrap in a function (better). The REPL has a softer rule since 1.5, so this bites only in scripts — which makes it worse, because it works when you test it interactively.

Mutation convention

A trailing `!` means "mutates the first argument". This is convention, not enforcement, but the entire ecosystem obeys it.

```
sort(v)      # returns a sorted copy
sort!(v)     # sorts in place, returns v
push!, append!, empty!, filter!, map!, replace!, mul!, copyto!
```

Argument passing

Pass-by-sharing. Rebinding inside a function does nothing to the caller; mutating does.

```
function f(v)
    v[1] = 99      # visible to caller
    v = [0, 0]     # NOT visible — rebinds the local name
end
```

Immutable types (`Int`, `Float64`, `Tuple`, non-`mutable struct`) cannot be mutated at all, so passing them is always safe.

Type annotations don't do what you think

```
f(x::Float64) = x^2     # a dispatch rule, not a performance hint
f(x) = x^2              # exactly as fast when called with a Float64
```

Julia compiles a specialised version per concrete argument type either way. Annotate for correctness, dispatch and documentation. The place annotations *do* matter for speed is struct fields (section 06) and, occasionally, `Ref`/container element types.

Vector{Int} is not a Vector{Real}

```
Vector{Int} <: Vector{Real}     # false — parametric types are INVARIANT
Vector{Int} <: Vector{<:Real}   # true
```

Write `f(v::AbstractVector{<:Real})`, not `f(v::Vector{Real})`. The latter accepts almost nothing. See section 04.

Ranges are lazy

```
r = 1:1_000_000     # 8 bytes, not 8 MB
collect(r)          # now it's 8 MB
sum(1:1_000_000)    # O(1), it uses the closed form
```

`1:n` where `n < 1` is empty, not an error, and not reversed. Descending is `n:-1:1` or `reverse(1:n)`.

Comprehension and generator parens

```
[f(x) for x in v]      # Vector, materialised
(f(x) for x in v)      # lazy generator, allocates nothing
sum(f(x) for x in v)   # no intermediate array — prefer this in hot code
```

Precompilation and first-call latency

The first call to any function compiles it. A fresh session plotting something can take ten seconds; the second plot is instant. This is normal, not a hang. The workflow implication is significant: **keep one REPL alive for hours and use Revise**, rather than re-running a script. Restarting is the Julia equivalent of a clean rebuild.

Similarly, `] add SomeBigPackage` triggers precompilation of the whole dependency graph, which can take minutes once and then never again.

Threads default to one

```
Threads.nthreads()    # 1, unless you started with -t auto
```

Nothing warns you. Everything "multithreaded" silently runs serially. Set `JULIA_NUM_THREADS=auto` in your shell profile.

Project environments are not the default

Plain `julia` puts you in the shared `@v1.12` environment, where installs are global. Use `julia --project` or set `JULIA_PROJECT=@.` so you're always in the nearest project. Otherwise you'll build a giant global environment whose dependency resolution eventually deadlocks.

Small things

- `1:3 == [1,2,3]` is **false** (different types); `1:3 == 1:3` is true; `collect(1:3) == [1,2,3]` is true.
- `nothing` prints as nothing at the REPL — a function returning `nothing` looks like it returned silently.
- `Dict` iteration order is unspecified and changes between runs; use `OrderedDict` (DataStructures.jl) or sort the keys.
- `x = [1,2,3]; x .+= 1` mutates in place; `x = x .+ 1` allocates a new array.
- Adjacent numeric literal and identifier means multiplication: `2π`, `3x`. So you cannot name a variable `2x`, and `x2` is fine but `2x` is `2*x`.
- The empty array literal `[]` is `Vector{Any}`. Write `Int[]` or `Float64[]`.
- `end` inside indexing means "last index of this dimension", and is also the block terminator. Context disambiguates.