Functions

Keyword args, splatting, `do` blocks, closures, destructuring

Definition forms

```
function add(x, y)
    return x + y          # `return` optional; last expression is the value
end

add(x, y) = x + y         # assignment form

const add2 = (x, y) -> x + y     # anonymous, bound to a name (slightly slower to dispatch)

function noargs()::Nothing       # return type annotation: CONVERTS the result
    nothing
end
```

A return type annotation is not a check, it's a `convert`. `f(x)::Int = x` will happily turn `2.0` into `2` and throw on `2.5`.

Arguments

```
f(a, b = 2; c, d = 4, kwargs...) = (a, b, c, d, kwargs)
#  ^  ^-- optional positional    ^-- required keyword (no default)
#         everything after ; is keyword-only
```

- Positional arguments participate in dispatch. **Keyword arguments do not.**
- Optional positional arguments generate extra methods behind the scenes (`f(a)` calls `f(a, 2)`).
- Keyword arguments are matched by name, order-independent, and can be splatted from a NamedTuple or Dict with `String` or `Symbol` keys.

```
opts = (c = 3, d = 5)
f(1; opts...)

# passing keywords through
wrapper(args...; kwargs...) = f(args...; kwargs...)
```

Splatting and slurping

```
f(args...) = length(args)      # slurp into a Tuple
t = (1, 2, 3)
f(t...)                        # splat
maximum([1,2,3]...)            # works, but prefer maximum([1,2,3])
```

Splatting a *long* runtime-length collection is slow (it forces a heap-allocated tuple and defeats inference). Splatting short, statically-known tuples is free.

Anonymous functions and closures

```
map(x -> x^2, 1:3)
filter(x -> x > 2, v)

function counter()
    n = 0
    () -> (n += 1)        # closure over n; n is boxed because it's reassigned
end
```

trap: closures that reassign captured variables box them, which is a performance cliff. If it matters, use a mutable struct or a `Ref` explicitly, or restructure so nothing is reassigned.

Multi-line anonymous:

```
f = function (x, y)
    x + y
end
```

`do` blocks

`do` is sugar that turns a trailing block into the *first* argument. Anywhere a function takes a function first, `do` reads better.

```
open("file.txt", "r") do io
    read(io, String)
end
# equivalent to: open(io -> read(io, String), "file.txt", "r")
```

This is Julia's resource-management idiom — the file closes even if the block throws. Same pattern for locks, temp dirs, connections:

```
lock(mylock) do
    shared_state += 1
end

mktempdir() do dir
    ...
end

Channel{Int}(10) do ch
    for i in 1:100; put!(ch, i); end
end
```

And for anything with a function argument:

```
map(1:3) do x
    y = expensive(x)
    y^2 + 1
end

sum(v) do x
    x^2
end
```

Composition and piping

```
(sqrt ∘ abs)(-4)          # 2.0;  ∘ is \circ<tab>
-4 |> abs |> sqrt         # 2.0
[-4, -9] .|> abs .|> sqrt # broadcast the pipe

filter(!isempty, lines)   # ! composes with a predicate to negate it
findfirst(==(3), v)       # ==(3) is a curried predicate; also <(5), in(set), isequal(x)
```

The curried-comparison forms (`==(x)`, `<(x)`, `in(c)`, `isequal(x)`) are used constantly with `findfirst`, `filter`, `count`, `any`, `all`.

Multiple return values and destructuring

```
minmax(3, 1)              # (1, 3) — just a tuple
lo, hi = minmax(3, 1)
a, b... = [1,2,3]         # a = 1, b = [2,3]
(; host, port) = config   # 1.7+: destructure by property name from a NamedTuple or struct

function stats(v)
    (mean = sum(v)/length(v), n = length(v))   # NamedTuple return: self-documenting
end
s = stats(v); s.mean
```

Returning a NamedTuple is the idiomatic alternative to out-parameters or a bespoke result struct, and it's type-stable and free.

Mutating functions

Convention: `!` suffix, mutated argument first.

```julia
function normalize!(v)
    v ./= sum(v)
    return v            # return the mutated argument, for chaining
end
```

Many `Base` functions have both forms, and the in-place one usually takes the destination as the *first* argument even when the non-mutating one takes it last:

```julia
map!(f, dest, src)
copyto!(dest, src)
mul!(C, A, B)          # C .= A * B without allocating
sort!(v); push!(v, x); append!(v, w); empty!(v)
```

Generic functions and `where`

```julia
f(x::T, y::T) where {T} = ...              # both arguments the same type
f(v::AbstractVector{T}) where {T <: Real} = zero(T)
f(::Type{T}) where {T} = T                 # dispatch on a type argument
```

Use `T` in the body when you need to construct something of the right type — `zero(T)`, `one(T)`, `similar(v)`, `T[]`. Hard-coding `0` or `Float64[]` in a generic function is the usual cause of "works for Float64, breaks for Complex or Measurement or Dual".

Recursion and iteration helpers

```julia
foldl(+, 1:5); foldr(-, 1:5)
reduce(+, v; init = 0)
mapreduce(abs2, +, v)          # fused map-then-reduce, no intermediate
accumulate(+, v); cumsum(v)
```

Julia has no tail-call optimisation. Deep recursion overflows the stack; convert to iteration or increase the stack with a `Task`:

```julia
t = Task(deep_recursive_thing)   # tasks get their own, growable stack
```

Documenting

````julia
"""
    solve(A, b) -> x

Solve `A x = b`.

# Arguments
- `A::AbstractMatrix`: the system matrix.
- `b::AbstractVector`: the right-hand side.

# Examples
```jldoctest
julia> solve([1 0; 0 1], [1, 2])
2-element Vector{Float64}:
 1.0
 2.0
```
"""
function solve(A, b) ... end
````

The `jldoctest` blocks are executed as tests by Documenter.jl, so your examples can't rot.

Function-like odds and ends

```julia
applicable(f, 1, 2)       # would this call find a method?
hasmethod(f, Tuple{Int})
Base.@nospecialize x      # tell the compiler NOT to specialise on this argument
@inline f(x) = ...        # hints; the compiler mostly knows better
@noinline
Base.@propagate_inbounds  # forward @inbounds through a wrapper
```