Syntax basics

Variables, control flow, operators, printing, the shape of the language

The parts you'd otherwise pick up by osmosis, compressed. If a construct behaves exactly like its equivalent in Python or Go, it gets one line.

Variables and assignment

```
x = 5                       # no declaration keyword, no type needed
x::Int = 5                  # optional type assertion on the binding (1.8+)
const MAX = 100             # binding cannot be reassigned (value may still mutate)
a = b = 0                   # chained
a, b = 1, 2                 # tuple destructuring
a, b = b, a                 # swap
x += 1; x *= 2              # compound assignment (no ++ or --)
_ = f()                     # conventional throwaway
```

Naming convention: `lower_snake` for variables and functions (often just `lowercase` with no separator for short names), `UpperCamel` for types and modules, `SCREAMING` for constants only when it aids readability, trailing `!` for functions that mutate their arguments.

Unicode identifiers are legal and idiomatic in numerical code: `α`, `Δt`, `∇f`, `σ²`. See section 20 for how to type them.

Comments

```
# line comment

#=
block comment,
which nests properly
=#

"""
    myfunc(x, y)

Docstrings go *before* the definition and are Markdown. They're
retrievable with `?myfunc` and become the package docs.
"""
myfunc(x, y) = x + y
```

Numbers and literals

```
1                   # Int64 on 64-bit
1.0                 # Float64
1f0                 # Float32
0x1f                # UInt8 — hex literals are unsigned, width from digit count
0b1010              # UInt8 binary
1_000_000           # underscores as digit separators
1//3                # Rational{Int}
1 + 2im             # Complex{Int}
big"123456789012345678901234567890"   # BigInt
3.0e-5
Inf, -Inf, NaN
2x                  # numeric literal juxtaposition: means 2*x. So is 3(a+b).
```

Type conversion and parsing:

```
Int(3.0)            # 3 — errors if not exactly representable
round(Int, 3.7)     # 4;  floor/ceil/trunc take a type too
float(3)            # 3.0
parse(Int, "42")
parse(Float64, "1.5")
```

```
tryparse(Int, "x") # nothing instead of throwing
string(42)
```

Operators

```
+  -  *  /  \  ^  %        # \ is left division: A \ b solves Ax = b
÷                          # integer division, also div(a,b)
==  !=  <  <=  >  >=       # comparison; chainable: 1 < x <= 10
===  !==                   # identity / egal
&&  ||  !                  # short-circuiting boolean
&  |  ⊻  ~  >>  <<         # bitwise (xor is ⊻ or xor(a,b))
∈  ∉  ⊆                    # in, not-in, subset (ASCII: in, issubset)
```

^ is exponentiation, never xor. && and || short-circuit and are commonly used as one-line guards:

```
x < 0 && throw(ArgumentError("negative"))
isempty(v) || process(v)
```

Ternary and the "return this or that" forms:

```
y = x > 0 ? "pos" : "nonpos"
y = ifelse(x > 0, a, b)     # evaluates both branches, no branch misprediction
```

Control flow

```
if x > 0
    ...
elseif x < 0
    ...
else
    ...
end

for i in 1:10 ... end
for (i, v) in enumerate(v) ... end
for i in 1:3, j in 1:3 ... end        # nested, one `end`, j varies fastest
while cond ... end
break; continue
```

There is no `switch`/`case`. Options: an if-chain, a `Dict` of functions, or dispatch on a type or a `Val`. `Match.jl` provides pattern matching if you miss Elixir's `case`.

No `do...while`. Use `while true ... cond || break; end`.

Blocks and scope

```
begin
    a = 1
    b = 2
end            # groups expressions, does NOT create a scope; value is last expr

let x = 1
    x + 1      # DOES create a scope; useful for capturing loop variables
end

(a = 1; a + 1)  # semicolon form of begin/end, value is last expr
```

Scope rules that matter:

- `function`, `for`, `while`, `let`, `struct`, `module`, comprehensions: **new scope**.
- `if`, `begin`, &&: **no new scope**. Variables assigned inside leak out.
- Inside a function, assigning to a name that exists globally creates a *local* unless you write `global x = ...`.
- **trap:** in a script (not the REPL), a `for` loop at top level cannot assign to a global without `global`. This is the single most common "why is my counter zero" question.

```
total = 0
for i in 1:10
    global total += i       # required at top level in a script
end
```

Everything is an expression

Assignments, `if`, `for` (returns `nothing`), and blocks all have values.

```
x = if cond; 1 else 2 end
f(x) = (y = 2x; y + 1)
```

Functions return the value of their last expression; `return` is optional but clearer in anything longer than a line.

Printing and output

```
print("no newline")
println("with newline")
println("interpolated: $x and $(x + 1)")
printf_style = @sprintf("%.3f", π)     # using Printf
@printf("%5.2f %s\n", 1.5, "ok")
display(x)                             # rich display (arrays get pretty-printed)
show(x)                                # developer-facing repr
@show x y                              # debug: prints "x = 1" and "y = 2", returns values
@info "message" key = value            # structured log to stderr
@warn "careful"; @error "bad"
```

`println` writes to stdout; `@info`/`@warn`/`@error` write to stderr with source location, which is what you want in libraries.

Functions, the short version

```
function add(x, y)
    x + y
end

add(x, y) = x + y            # assignment form, for one-liners
add(x, y::Int = 0) = x + y   # default argument
add(x; scale = 1) = scale*x  # keyword argument (after the semicolon)
(x -> x^2)(3)                # anonymous
```

Arguments are **passed by sharing**: you get a reference to the same object, so mutating an array argument is visible to the caller, but rebinding the name isn't. Same as Python or JS.

Full detail in section 07.

Collections, the short version

```
v = [1, 2, 3]                  # Vector{Int64}
m = [1 2; 3 4]                 # 2x2 Matrix — space separates columns, ; rows
t = (1, "two", 3.0)            # Tuple, heterogeneous, immutable
nt = (a = 1, b = 2)            # NamedTuple
d = Dict("a" => 1, "b" => 2)   # Dict{String, Int64}
s = Set([1, 2, 2])             # Set{Int64} with 2 elements
r = 1:10                       # UnitRange, lazy
r2 = 0:0.5:2                   # StepRange
```

Detail in section 08.

Type annotations, the short version

```
f(x::Int) = ...                # method signature: this is DISPATCH, not a check
x::Int = 5                     # binding assertion: converts and enforces
v = Int[]                      # typed empty vector
Vector{Float64}(undef, 10)     # uninitialised, 10 elements
```

Annotating function arguments does not make code faster. Annotating struct fields does. Section 04.

Running Julia

```
julia                              # REPL
julia script.jl arg1 arg2          # ARGS == ["arg1","arg2"]
julia --project                    # activate the Project.toml in this dir tree
julia --project -t auto            # ... with all threads
julia -e 'println(1+1)'            # one-liner
julia -i script.jl                 # run then drop into REPL
```

A script has no `main` and no entry point ceremony; top-level code runs top to bottom. The `if abspath(PROGRAM_FILE) == @__FILE__` idiom is the `if __name__ == "__main__"` equivalent, if you want a file that's both script and library.