Strings and symbols

Interpolation, byte indexing, regex, `Symbol`

Literals and interpolation

```
"hello"
'a'                      # Char — SINGLE quotes are a character, not a string
"a" * "b"                # concatenation is *, not +
"ab"^3                   # repetition
"$name is $(age + 1)"    # interpolation, any expression in $()
"\$literal dollar"

"""
Triple-quoted. Leading whitespace common to all lines is stripped,
so you can indent this inside a function without polluting the string.
"""

raw"C:\no\escapes\here"  # no escape processing, no interpolation
b"bytes"                 # a byte array literal (CodeUnits)
```

The byte-index trap

Strings are UTF-8, immutable, and indexed by **byte offset**.

```
s = "héllo"
length(s)        # 5   — characters
ncodeunits(s)    # 6   — bytes
s[1]             # 'h'
s[2]             # 'é'
s[3]             # StringIndexError: index 3 is mid-character
s[nextind(s, 2)] # 'l'
s[end]           # 'o'  — `end` is a valid index by construction

firstindex(s), lastindex(s)
eachindex(s)     # the valid byte indices
prevind(s, i), nextind(s, i), thisind(s, i)
```

Safe patterns: iterate with `for c in s`, index with results from `findfirst`/`eachindex`, or `collect(s)` into a `Vector{Char}` if you truly need positional access. `SubString` is the free-slicing type — `s[2:4]` returns a `String` copy, `@view` doesn't work on strings, but `SubString(s, 2, 4)` and the outputs of `split` are views.

The everyday operations

```
length, isempty, first, last
occursin("ell", s); occursin(r"\d+", s)
startswith(s, "he"); endswith(s, "o")
findfirst("ll", s)                    # a range of byte indices, or nothing
replace(s, "l" => "L"); replace(s, "a" => "b", "c" => "d")
split("a,b,,c", ","); split(s)     # bare split(): on whitespace, drops empties
split("a,b", ","; keepempty = false)
join(["a","b","c"], ", ", " and ")
strip(s); lstrip; rstrip; strip(s, ['"'])
uppercase, lowercase, titlecase, uppercasefirst
lpad("7", 3, "0"); rpad
repeat("ab", 3)
chop(s); chomp(s)                     # remove last char / trailing newline
reverse(s)
string(1, "x", 2.0)                   # generic stringify-and-concat
```

Building strings efficiently

Repeated `s = s * x` is quadratic. Use an `IOBuffer`:

```
io = IOBuffer()
for x in items
    print(io, x, ",")
end
result = String(take!(io))

# or
sprint(io -> foreach(x -> print(io, x), items))
```

Formatting

```
using Printf
@printf("%.3f %5d %s\n", π, 42, "ok")
s = @sprintf("%08.3f", π)

# runtime format strings (1.6+)
fmt = Printf.Format("%.2f")
Printf.format(fmt, 1.234)

round(π; digits = 3)            # often all you need
string(1234; base = 16)         # "4d2"
bitstring(1)
```

For anything table-shaped, `PrettyTables.jl` beats hand-rolled padding.

Regex

```
r"(\d+)-(\d+)"                  # Regex literal
r"case"i                        # flags: i, m, s, x

m = match(r"(\d+)-(\d+)", "ab 12-34 cd")
m === nothing                   # no match
m.match                         # "12-34"
m.captures                      # ["12", "34"]  (Union{String,Nothing} elements)
m[1]                            # "12"
m.offset                        # byte index of the match

occursin(r"^\d+$", s)
replace(s, r"\s+" => " ")
replace(s, r"(\w+)@(\w+)" => s"\2 at \1")     # s"" for substitution templates
for m in eachmatch(r"\d+", s); m.match; end
split(s, r",\s*")

# named captures
m = match(r"(?<year>\d{4})-(?<month>\d{2})", "2026-08")
m[:year]
```

Parsing and conversion

```
parse(Int, "42"); parse(Float64, "1.5"); parse(Int, "ff"; base = 16)
tryparse(Int, "x")          # nothing rather than throwing
string(42); string(3.14)
Char(65); Int('A')
codeunits(s)                # byte view
transcode(UInt16, s)
```

Symbols

A `Symbol` is an interned, immutable identifier. Same role as an Elixir atom, but used more narrowly: for property names, keyword names, option flags, and as the leaves of expression trees in metaprogramming.

```
:foo
Symbol("with spaces")
Symbol("prefix_", 42)           # Symbol("prefix_42")
```

```
String(:foo); Symbol("foo")     # round trip

nt = (host = "x", port = 80)
nt[:host]
getproperty(obj, :field)        # obj.field, computed
setproperty!(obj, :field, v)
propertynames(obj)
fieldnames(MyStruct)            # Tuple of Symbols
```

Symbol comparison is pointer comparison, so `:a == :b` is O(1) regardless of length — that's why they're used as dict keys for fixed vocabularies and as option flags:

```
sort(v; alg = :quicksort)
plot(x, y; linestyle = :dash)
```

They are *not* a general-purpose string type. Don't build them from user input in a loop — the intern table is never garbage collected.

Chars

```
c = 'a'
c + 1                       # 'b'
'a' <= c <= 'z'
isletter(c), isdigit(c), isspace(c), isuppercase(c), ispunct(c)
uppercase('a')
codepoint(c)
```