Standard tasks

Files, processes, dates, data wrangling, HTTP, JSON

Files and paths

```
read("f.txt", String)
readlines("f.txt")
write("f.txt", "content")
readchomp(`cmd`)

open("out.csv", "w") do io
    println(io, "a,b")
    write(io, data)
end

for line in eachline("big.log")      # streaming, constant memory
    process(line)
end

readdir("."); readdir("."; join = true)
walkdir(".")                         # recursive, yields (root, dirs, files)
isfile, isdir, ispath, islink
mkdir, mkpath, rm(p; recursive = true, force = true)
cp(src, dst; force = true); mv
joinpath("a", "b", "c.txt")
splitpath, splitdir, splitext, basename, dirname
abspath, normpath, realpath, expanduser("~/x")
homedir(), tempdir(), tempname()
mktemp() do path, io ... end
mktempdir() do dir ... end
filesize(p), mtime(p), stat(p)
touch(p)
@__DIR__                             # directory of the current source file
pkgdir(MyPkg)                        # a package root — the way to find its assets
```

Binary IO:

```
data = read("f.bin")                 # Vector{UInt8}
open("f.bin") do io
    x = read(io, Int64)
    v = read(io, 100)
    seek(io, 0); position(io); eof(io)
end
```

Processes

Backticks build a `Cmd`. **There is no shell**, so no quoting bugs and no injection.

```
run(`ls -la`)                        # throws on non-zero exit
read(`git rev-parse HEAD`, String)
readchomp(`hostname`)
success(`which julia`)               # Bool instead of throwing

file = "name with spaces.txt"
run(`cat $file`)                     # passed as ONE argument, correctly

run(pipeline(`cat f.txt`, `grep x`))
run(pipeline(`cmd`, stdout = "out.txt", stderr = "err.txt"))
open(`sort`, "w", stdout) do io
    println(io, "b"); println(io, "a")
```

```
end

p = run(`long-thing`; wait = false)
kill(p); process_running(p); p.exitcode

withenv("VAR" => "value") do
    run(`printenv VAR`)
end
addenv(`cmd`, "K" => "v")
Cmd(`cmd`; dir = "/tmp")
```

If you genuinely need shell features, invoke a shell explicitly:

```
run(`sh -c "a | b > c"`)
```

Environment and arguments

```
ENV["HOME"]
get(ENV, "PORT", "8080")
haskey(ENV, "CI")
ARGS                                # Vector{String} of command-line args
PROGRAM_FILE
Sys.iswindows(), Sys.islinux(), Sys.isapple()
Sys.CPU_THREADS, Sys.total_memory()
VERSION                             # v"1.12.6"
@static if Sys.islinux() ... end    # compile-time branch
```

Argument parsing: `ArgParse.jl` (full-featured, argparse-shaped) or `Comonicon.jl` (turns a function signature into a CLI).

Dates and time

```
using Dates
now(); today(); now(UTC)
Date(2026, 8, 17); DateTime(2026, 8, 17, 14, 30)
Date("2026-08-17"); Date("17/08/2026", dateformat"dd/mm/yyyy")
Dates.format(now(), "yyyy-mm-dd HH:MM:SS")

now() + Day(3); now() - Hour(2)
Year, Month, Week, Day, Hour, Minute, Second, Millisecond
year(d), month(d), day(d), dayofweek(d), dayofyear(d), isleapyear(d)
d1 - d2                             # a Period
Dates.value(d1 - d2)                # as a number
firstdayofmonth(d), lastdayofmonth(d)
Date(2026,1,1):Day(1):Date(2026,12,31)   # ranges of dates work

time()                              # Unix seconds, Float64
time_ns()                           # monotonic nanoseconds — use for timing
unix2datetime(t), datetime2unix(d)
```

Timezones are not in the stdlib: `TimeZones.jl` (`ZonedDateTime`, `TimeZone("Europe/London")`).

Randomness

```
using Random
rand()                  # Float64 in [0,1)
rand(1:6); rand(v); rand(3, 3); rand(Bool)
randn(100)              # standard normal
randexp, randstring(8)
shuffle(v); shuffle!(v)
randperm(10); randsubseq(v, 0.1)
sample(v, 5; replace = false)       # StatsBase.jl
Random.seed!(42)
```

```
rng = MersenneTwister(42); rand(rng, 10)
rng = Xoshiro(42)                    # the current default algorithm
```

For reproducibility across Julia versions use `StableRNGs.jl` — the default RNG is explicitly allowed to change.

Data wrangling in Base

```
sort(v; by = x -> x.age, rev = true)
sort(v; lt = (a, b) -> a.n < b.n)
sortperm(v); partialsort(v, 1:10); partialsortperm
searchsorted(sorted_v, x); searchsortedfirst
filter(f, v); count(f, v); any(f, v); all(f, v)
findfirst, findlast, findall, findnext
findmax(v), argmax(v), argmax(f, v)
unique(v), unique(f, v), allunique(v)
union, intersect, setdiff, symdiff, issubset
reduce, foldl, mapreduce, accumulate, cumsum, cumprod
zip, enumerate, pairs
Iterators.partition(v, 3), flatten, product, take, drop, cycle, repeated, countfrom
group = Dict(); for x in v; push!(get!(group, key(x), []), x); end   # groupby by hand
```

`SplitApplyCombine.jl` gives you `group`, `groupreduce`, `innerjoin` for plain collections without a DataFrame.

Statistics and linear algebra (stdlib)

```
using Statistics
mean(v), median(v), std(v), var(v), quantile(v, [0.25, 0.75]), cor(x, y), cov(x, y)
mean(A; dims = 1)

using StatsBase          # not stdlib, but the standard extension
countmap(v), sample, weights, fit(Histogram, v), zscore, describe
```

Linear algebra has its own section (18).

Serialisation

```
using JSON3
obj = JSON3.read(json_string)              # lazy, indexable
obj = JSON3.read(json_string, MyStruct)    # typed, with StructTypes.jl
JSON3.write(obj)
JSON3.pretty(obj)

using StructTypes
StructTypes.StructType(::Type{MyStruct}) = StructTypes.Struct()
```

```
using CSV, DataFrames
df = CSV.read("data.csv", DataFrame)
CSV.write("out.csv", df)
CSV.File("data.csv") |> collect            # without DataFrames

using Serialization         # Julia-native binary; version-fragile, don't archive with it
serialize("f.jls", obj); deserialize("f.jls")

using JLD2                  # HDF5-based, good for arrays and long-term storage
jldsave("f.jld2"; a = 1, b = v)
load("f.jld2", "a")

using TOML                                 # stdlib
TOML.parsefile("Project.toml")
TOML.print(dict)
```

HTTP

```
using HTTP
```

```
r = HTTP.get("https://example.com")
r.status; String(r.body); Dict(r.headers)

HTTP.post(url; body = JSON3.write(payload),
              headers = ["Content-Type" => "application/json"])
HTTP.get(url; query = Dict("q" => "x"), retry = true, readtimeout = 30)

# server
router = HTTP.Router()
HTTP.register!(router, "GET", "/health", req -> HTTP.Response(200, "ok"))
HTTP.serve(router, "0.0.0.0", 8080)
```

`Oxygen.jl` wraps this with routing macros and OpenAPI generation if you want less ceremony. `Downloads.download(url, path)` (stdlib) is enough for fetching a file.

DataFrames, minimal

```
using DataFrames
df = DataFrame(a = 1:3, b = ["x","y","z"])

df.a                    # column, no copy
df[!, :a]               # column, no copy
df[:, :a]               # column, COPY
df[1, :]; df[1:2, [:a, :b]]
names(df); nrow(df); ncol(df); describe(df)

subset(df, :a => x -> x .> 1)
filter(:a => >(1), df)
transform(df, :a => (x -> 2x) => :doubled)
select(df, :a, :b => :renamed)
combine(groupby(df, :b), :a => sum => :total)
sort(df, :a; rev = true)
innerjoin(df1, df2; on = :id); leftjoin; outerjoin
stack(df); unstack(df)
```

The `source => function => destination` mini-language is the thing to learn; everything in DataFrames is expressed in it. `DataFramesMeta.jl` provides `@select`/`@subset`/`@by` macros if you prefer dplyr-shaped syntax.

Plotting

```
using Plots
plot(x, y; label = "series", xlabel = "t", lw = 2)
plot!(x, y2)             # add to the current plot (note the !)
scatter(x, y); histogram(v); heatmap(A); surface(x, y, z)
savefig("out.png")

# headless: avoid GR trying to open a window
ENV["GKSwstype"] = "100"
```

`Makie.jl` (CairoMakie for publication figures, GLMakie for interactive/GPU) is the more capable and more modern option, at the cost of heavier load time. `UnicodePlots.jl` draws in the terminal, which is genuinely useful over SSH.