Julia-Zusammenfassung (Programmierung + Plotting)

Cheat-Sheet für ein Einführungsmodul (Numerik/Stochastik-orientiert)

Contents

1 Arbeitsumgebung: REPL, Hilfe, Paketmanager

Julia wird oft interaktiv verwendet (REPL), was für Numerik/Stochastik sehr praktisch ist.

REPL-Modi (super wichtig)

In der REPL hast du verschiedene Modi:

- **Julia-Modus** (Standard): Rechne/Programmiere normal.
- **Hilfe** mit `?`: `?sin` zeigt Docstring.
- **Shell** mit `;`: Shell-Kommandos (z.B. `;ls`).
- **Pkg** mit `]`: Paketmanager.

Pkg-Grundbefehle (Projekt-Umgebungen)

Empfohlen: pro Kurs/Projekt eine eigene Umgebung.

```
] status
] add Plots
] add Random LinearAlgebra Statistics
] update
] activate .
] instantiate
```

Merke: Mit `activate .` nutzt du die Umgebung im aktuellen Ordner (erstellt `Project.toml`).

2 Grundlagen: Variablen, Typen, Zuweisung

2.1 Variablen und Literale

```
x = 3 # Int
y = 3.0 # Float64
z = 2 + 5im # Complex{Int64}
b = true # Bool
s = "Hallo" # String
```

2.2 Typen ansehen und konvertieren

```
typeof(x) # Int64 (meist)
Int(3.7) # 3
Float64(3) # 3.0
```

2.3 Wichtige "Sonderwerte"

```
nothing # "kein Wert" (z.B. Return in Prozeduren)
missing # "fehlender Wert" (für Daten/Statistik)
NaN # Not-a-Number (Float)
Inf, -Inf
```

2.4 Konstante

```
const g = 9.81
```

3 Operatoren und elementare Ausdrücke

3.1 Arithmetik

```
a + b, a - b, a * b, a / b # / liefert Float
a ÷b # ganzzahliger Quotient (div)
a % b # Rest (mod)
a^n # Potenz
```

3.2 Vergleich und Logik

```
x == y, x != y, x < y, x <= y
x && y, x || y # short-circuit (wichtig!)
!x
```

3.3 Broadcasting (Punkt-Operatoren) — extrem wichtig in Julia

Julia unterscheidet klar zwischen Skalar- und Elementoperationen:

```
v = [1,2,3]
v + 1 # ERROR (meist), weil + nicht elementweise definiert ist
v .+ 1 # [2,3,4] elementweise
sin.(v) # wendet sin auf jedes Element an
```

Merke: Der Punkt macht "apply to each element" und fused oft zu schnellem Code.

4 Anweisungen: Bedingungen

```
x = 4
if x < 0
    y = -1
elseif x == 0
    y = 0
else
    y = 1
end
```

Kurzformen:

```
y = (x >= 0) ? sqrt(x) : 0.0 # ternary
(x > 0) && println("positiv") # nur wenn Bedingung wahr
```

5 Schleifen (for/while), break/continue

5.1 for-Schleife

```
s = 0
for k in 1:10
    s += k
end
```

5.2 while-Schleife

```
x = 1.0
while x < 100
    x *= 1.5
end
```

5.3 break/continue

```
for k in 1:100
    if k % 7 == 0
        continue
    end
    if k > 30
        break
    end
end
```

5.4 Typische Iteratoren

```
for (i,val) in enumerate(["a","b","c"])
    println(i, " -> ", val)
end

for (a,b) in zip(1:3, 10:12)
    println(a, ", ", b)
end
```

6 Datenstrukturen: Tuple, Arrays, Dict, Set

6.1 Tuple und NamedTuple

```
t = (1, "x", 3.0)
name = (vorname="Mika", ects=6)

name.vorname
```

6.2 Arrays (Vektoren/Matrizen) und Indexing (1-basiert!)

```
v = [1,2,3] # Vector{Int}
A = [1 2; 3 4] # 2x2 Matrix
zeros(3) # [0.0,0.0,0.0]
ones(2,3)
collect(1:0.5:3) # Range -> Array

v[1] # erstes Element
A[2,1] # Zeile 2, Spalte 1
A[:,2] # ganze 2. Spalte
A[1,:] # ganze 1. Zeile
```

6.3 Views (schneller, ohne Kopie)

```
@views col2 = A[:,2] # View statt Kopie
```

6.4 Comprehensions (kurz und mächtig)

```
sq = [k^2 for k in 1:10]
M = [i+j for i in 1:3, j in 1:4]
```

6.5 Dict und Set

```
d = Dict("a"=>1, "b"=>2)
d["a"] # 1

S = Set([1,2,2,3]) # {1,2,3}
```

```
in(2, S) # true
```

7 Funktionen/Prozeduren: Definition, Rückgabe, Mutating-Konvention

7.1 Grundform

```
function f(x)
    return x^2 + 1
end

f(3)
```

7.2 Kurzform

```
g(x) = x^2 + 1
```

7.3 Mehrere Rückgabewerte (Tuple)

```
function minmax(v)
    return minimum(v), maximum(v)
end

mn, mx = minmax([3,1,9])
```

7.4 Default- und Keyword-Argumente

```
h(x, a=2) = a*x

function gauss(x; mu=0.0, sigma=1.0)
    return exp(-0.5*((x-mu)/sigma)^2) / (sigma*sqrt(2*pi))
end

gauss(0.2, mu=0.0, sigma=2.0)
```

7.5 Anonyme Funktionen und do-Block

```
map(x -> x^2, 1:5)

open("data.txt", "w") do io
    write(io, "Hallo\n")
end
```

7.6 Mutating-Funktionen: Konvention !

Wenn eine Funktion ihr Argument *in-place* verändert, endet sie oft auf `!`.

```
v = [3,1,2]
sort!(v) # verändert v selbst
```

8 Ein-/Ausgabe (I/O): Konsole und Dateien

8.1 Konsole

```
println("x = ", 3)
s = readline() # liest eine Zeile (String)
n = parse(Int, s) # String -> Int
```

8.2 Dateien lesen/schreiben

```
# schreiben
open("out.txt", "w") do io
    for k in 1:3
        println(io, k, ", ", k^2)
    end
end

# lesen (ganzer Inhalt)
txt = read("out.txt", String)

# zeilenweise lesen
open("out.txt", "r") do io
    for line in eachline(io)
        println("Zeile: ", line)
    end
end
```

9 Module: Code strukturieren, import/using

9.1 Eigenes Modul

```
module MyTools
export square, hello

square(x) = x^2
hello() = println("Hi!")

end
```

Nutzung:

```
include("MyTools.jl")
using .MyTools
square(5)
```

Merke:

- `include("file.jl")` lädt Code aus Datei.
- `using Modul` importiert exportierte Namen.
- `import Modul: f` ist nützlich, wenn du gezielt erweitern willst.

10 Fortgeschritten: Overloading + (Multiple) Dispatch (Julia-Kernidee)

In Julia ist **Multiple Dispatch** zentral: eine Funktion kann viele *Methoden* haben, ausgewählt nach den Typen *aller* Argumente.

10.1 Function Overloading (mehrere Methoden)

```
area(r::Real) = pi*r^2 # Kreis
area(a::Real, b::Real) = a*b # Rechteck

area(2)
area(3,4)
```

Methoden inspizieren:

```
methods(area)
```

10.2 Eigene Typen: `struct` und Methoden ("OOP-ähnlich")

Julia hat keine "Klassen" wie Python/Java, aber `struct` + Funktionen liefern denselben Effekt (nur idiomatischer).

```
struct Particle
    m::Float64
    x::Float64
    v::Float64
end

kinetic(p::Particle) = 0.5*p.m*p.v^2
```

10.3 Parametrische Typen

```
struct Box{T}
    value::T
end

b1 = Box(3)
b2 = Box("hi")
```

10.4 Multiple Dispatch mit mehreren Argumenten

```
abstract type Shape end
struct Circle <: Shape
    r::Float64
end
struct Rect <: Shape
    a::Float64
    b::Float64
end

area(s::Circle) = pi*s.r^2
area(s::Rect) = s.a*s.b
```

Warum wichtig? Du schreibst neue Typen und definierst "was Funktionen damit tun" ohne riesige `if`-Ketten.

10.5 Operatoren erweitern (Overloading von + etc.)

```
struct Vec2
    x::Float64
    y::Float64
end

import Base: +, -, show

+(a::Vec2, b::Vec2) = Vec2(a.x + b.x, a.y + b.y)
-(a::Vec2, b::Vec2) = Vec2(a.x - b.x, a.y - b.y)

function show(io::IO, v::Vec2)
    print(io, "Vec2(", v.x, ", ", v.y, ")")
end

Vec2(1,2) + Vec2(3,4)
```

10.6 Dispatch herausfinden

```
@which area(Circle(1.0))
```

11 Lineare Algebra & Numerik-Basics (typisch in Numerik/Stochastik)

(Sehr häufig in Übungen, auch wenn es nicht explizit im Modultext steht.)

11.1 LinearAlgebra: `dot`, `norm`, LGS

```
using LinearAlgebra

v = [1.0,2.0,3.0]
w = [3.0,0.0,1.0]

dot(v,w)
norm(v)

A = [3.0 1.0; 2.0 4.0]
b = [1.0, 2.0]
x = A \ b # löst A*x = b
```

11.2 Einfache Numerik-Aufgabe: Trapezregel

```
function trapz(f, a, b, n::Int)
    h = (b-a)/n
    s = 0.5*(f(a) + f(b))
    for k in 1:n-1
        s += f(a + k*h)
    end
```

```
    return h*s
end

trapz(sin, 0.0, pi, 10_000)
```

11.3 Random/Statistics: Simulation und Kenngrößen

```
using Random, Statistics

Random.seed!(1)
x = randn(1000) # Normalverteilung
mean(x), std(x)

p = mean(x .> 0) # Monte-Carlo-Schätzer: P(X>0)
```

12 Visualisierung: Plots erzeugen (2D/3D)

Für Einsteiger ist `Plots.jl` der Standard.

```
using Pkg
# ] add Plots
using Plots
```

12.1 2D-Funktionsplot

```
x = range(0, 2pi; length=400)
y = sin.(x)

plot(x, y, label="sin(x)", xlabel="x", ylabel="y", title="2D-Plot")
```

12.2 Scatter, Histogram, mehrere Kurven

```
scatter(x, y, label="Samples")

histogram(randn(10_000), bins=50, label="Histogramm")

plot(x, sin.(x), label="sin")
plot!(x, cos.(x), label="cos") # plot! fügt hinzu
```

12.3 Heatmap/Contour

```
xs = range(-2,2; length=200)
ys = range(-2,2; length=200)
Z = [exp(-(x^2+y^2)) for x in xs, y in ys]

heatmap(xs, ys, Z, title="Heatmap")
contour(xs, ys, Z, title="Contour")
```

12.4 3D Surface

```
surface(xs, ys, Z, title="3D-Surface")
```

12.5 Speichern

```
savefig("plot.pdf")
savefig("plot.png")
```

13 Typische Übungsaufgaben: Musterlösungen

13.1 1) "Lies Zahlen ein, berechne Mittelwert, plotte"

```
using Statistics, Plots

data = [1.0, 1.2, 0.9, 1.1, 1.05]
m = mean(data)

plot(data, marker=:circle, label="data")
hline!([m], label="mean")
```

13.2 2) Kleine Simulation (Bernoulli/Monte-Carlo)

```
using Random, Statistics, Plots

function estimate_pi(N::Int)
    inside = 0
    for k in 1:N
        x, y = rand(), rand()
        inside += (x^2 + y^2 <= 1.0)
    end
    return 4 * inside / N
end

Ns = [10^k for k in 1:6]
vals = [estimate_pi(N) for N in Ns]
plot(Ns, vals, xscale=:log10, marker=:circle, xlabel="N", ylabel="pi-Schätzer")
hline!([pi], label="pi")
```

14 Fehlerquellen & Best Practices (prüfungsrelevant, weil es viele Bugs verhindert)

14.1 1) Globals vermeiden (Performance/Verhalten)

In Julia sind globale Variablen oft langsam/fehleranfällig. Pack Logik in Funktionen.

```
function main()
    s = 0
    for k in 1:10^6
        s += k
    end
```

```
    return s
end
```

14.2 2) `missing` vs `nothing`

- `missing`: Daten fehlen (Statistik/Data).
- `nothing`: "kein Ergebnis"/"kein Objekt" (Programmlogik).

14.3 3) Debug/Inspection Tools

```
@show x
typeof(x)
methods(f)
@which f(args...)
```

15 Mini-Cheat-Sheet (zum schnellen Nachschlagen)

Thema	**Merksatz / Befehl**
Help/Docs	`?name` in REPL
Pkg	`]` `add`, `activate .`, `status`
Elementweise	Punkt: `sin.(x)`, `A .+ 1`
Indexing	1-basiert: `A[1,1]`
Schleifen	`for`, `while`, `break`, `continue`
Funktionen	`f(x)=...` oder `function f(x) ... end`
I/O	`readline()`, `open(...) do io ... end`
Module	`module ... end`, `export`, `using`, `import`
Dispatch	mehrere Methoden: `f(x::T)`; `methods(f)`
Plotting	`plot`, `scatter`, `histogram`, `savefig`

Empfehlung für die Klausur/Übung: Kannst du (i) einfache Aufgaben sauber in Funktionen schreiben, (ii) Arrays/Indexing/Broadcasting sicher nutzen, (iii) einfache Plots erzeugen, und (iv) ein Beispiel für Multiple Dispatch erklären/programmieren, dann bist du für dieses Modul in der Regel sehr gut aufgestellt.