Erste Schritte

Grundlagen Operations Research

Diese Seite fasst die Julia-Grundlagen zusammen, die in den Vorlesungen und Übungen dieses Kurses vorkommen, nicht mehr und nicht weniger. Alle Zellen sind ausgeführt; am besten tippen Sie die Beispiele parallel in Ihrer eigenen REPL nach (siehe Installation & Setup).

Variablen und Typen

Variablen entstehen durch einfache Zuweisung. Einen Typ müssen Sie nicht angeben, Julia leitet ihn selbst ab:

```
kosten = 25          # ganze Zahl (Int64)
gewicht = 2.5        # Kommazahl (Float64)
name = "Auftrag 1"   # Zeichenkette (String)
zulaessig = true     # Wahrheitswert (Bool)

typeof(kosten), typeof(gewicht)
```

```
(Int64, Float64)
```

In Zeichenketten können Sie Werte mit `$` einsetzen (*Interpolation*). So geben wir im Kurs Ergebnisse aus:

```
println("$name kostet $kosten EUR und wiegt $gewicht Tonnen.")
```

```
Auftrag 1 kostet 25 EUR und wiegt 2.5 Tonnen.
```

Vektoren und Matrizen

Vektoren schreibt man in eckigen Klammern. **Wichtig:** Julia zählt ab 1, nicht ab 0. Das erste Element ist `v[1]`:

```
d = [140, 110, 190, 120]   # z. B. Nachfragen von 4 Perioden

d[1], d[end], length(d), sum(d)
```

```
(140, 120, 4, 560)
```

Mit `push!` hängen Sie ein Element an, mit `d[2:3]` greifen Sie einen Ausschnitt heraus:

```
push!(d, 200)   # das ! signalisiert: d wird verändert
d[2:3]
```

```
2-element Vector{Int64}:
 110
 190
```

Matrizen trennen Spalten mit Leerzeichen und Zeilen mit Semikolon; der Zugriff erfolgt mit `[Zeile, Spalte]`:

```
c = [130 140 170;    # z. B. Transportkosten:
     210 150 130]    # 2 Anbieter x 3 Nachfrager

c[2, 3]
```

```
130
```

Ranges

Ein Bereich wie `1:5` steht für die Zahlen 1 bis 5. Das nutzen wir ständig für Indexmengen (z. B. „alle Perioden $t \in \{1, \ldots, 5\}$“):

```
T = 1:5
collect(T)   # collect zeigt die Werte als Vektor
```

```
5-element Vector{Int64}:
 1
 2
 3
 4
 5
```

Broadcasting: der Punkt-Operator

Ein Punkt vor einem Operator oder nach einem Funktionsnamen wendet die Operation **elementweise** auf einen Vektor an:

```
mengen = [10, 20, 15]
preise = [2.0, 1.5, 3.0]

erloes = mengen .* preise   # elementweise multiplizieren
```

```
3-element Vector{Float64}:
 20.0
```

```
 30.0
 45.0
```

```
sum(erloes), maximum(erloes)
```

```
(95.0, 45.0)
```

Das Gleiche funktioniert mit Funktionen, z. B. `value.(X)` in JuMP: „lies den Wert **jeder** Variablen im Container `X`".

Verzweigungen

Fallunterscheidungen folgen dem Muster `if` / `elseif` / `else` und enden (wie alle Blöcke in Julia) mit `end`:

```
bestand = 4

if bestand == 0
    println("Lager leer")
elseif bestand < 10
    println("Lager fast leer: $bestand Stück")
else
    println("Lager gefüllt")
end
```

```
Lager fast leer: 4 Stück
```

Vergleiche liefern Wahrheitswerte (`==`, `!=`, `<`, `<=`, `>`, `>=`), kombiniert wird mit `&&` (und), `||` (oder) und `!` (nicht).

Schleifen

Eine `for`-Schleife läuft über einen Range oder Vektor:

```
for t in 1:3
    println("Periode $t: Nachfrage = $(d[t])")
end
```

```
Periode 1: Nachfrage = 140
Periode 2: Nachfrage = 110
Periode 3: Nachfrage = 190
```

Sehr oft brauchen wir Summen mit Laufindex, genau wie das Summenzeichen $\sum$ in der Mathematik:

```
sum(d[t] for t in 1:4)   # entspricht der Summe von d1 bis d4
```

```
560
```

Diese Schreibweise werden Sie in jedem JuMP-Modell wiedersehen, z. B. in `@objective(modell, Min, sum(f * X[t] for t in T))`.

Funktionen

Eigene Funktionen definieren Sie mit `function` … `end`; der letzte Ausdruck ist automatisch der Rückgabewert:

```
function deckungsbeitrag(erloes, kosten)
    return erloes - kosten
end

deckungsbeitrag(500, 320)
```

```
180
```

Für Einzeiler gibt es die Kurzform:

```
quadrat(x) = x^2
quadrat(9)
```

```
81
```

Wie geht es weiter?

Damit kennen Sie alle Sprachbausteine, die in den Kurs-Modellen vorkommen. Weiter geht es mit Pakete & JuMP. Dort setzen wir daraus das erste Optimierungsmodell zusammen. Zum schnellen Nachschlagen dient das Cheatsheet Julia.

Bibliografie