

Einführung in die Programmiersprache julia

Überblick

- julia ist eine höhere Programmiersprache, die seit 2009 entwickelt wird
- Hauptaugenmerk liegt auf dem numerischen und wissenschaftlichen Rechnen
- Einfache Syntax
- Hohe Ausführungsgeschwindigkeit
- Einbindung von Python und R ist möglich
- open source (MIT-Lizenz)
- Download und Dokumentation: <https://julialang.org/>



Installation von julia

- Unter <https://julialang.org/downloads/> die passende Datei herunterladen und installieren
- Nach der Installation kann julia über die REPL (read-evaluate-print-loop) verwendet werden
- Die REPL ist für das Einüben und Ausprobieren gut geeignet
- Für größere Projekte sollte jedoch eine IDE (wie etwa Juno) oder das Jupyter Notebook verwendet werden
- Über die REPL die folgenden Befehle ausführen um Jupyter Notebook zu installieren⁸

```
julia> using Pkg  
julia> Pkg.add("IJulia")
```

⁸Siehe <https://github.com/JuliaLang/IJulia.jl>

Problemklassen und Einzelprobleme

Problemklasse

Eine *Problemklasse* ist eine Sammlung gleichartiger Probleme, wobei diese nicht konkret formuliert sind, sondern in ihrer gemeinsamen, allgemeinen Form dargestellt sind. In der Regel kann eine Klasse über eine Funktion $f : I \rightarrow O$ repräsentiert werden, mit I als Menge der Eingabe- und O als Menge der Ausgabewerte.

Beispiele:

- Finden eines kürzesten Weges zwischen zwei Orten
- Ermittlung möglichst ähnlicher Kundengruppen eines Unternehmens
- Erkennung von Objekten in digitalen Bilddateien
- Zuordnung von Aufgaben zu Bearbeitern
- Ermittlung der optimalen Auslagerstrategie für ein Lager

Einzelproblem

Ein *Einzelproblem* ist eine konkrete Instanz einer Problemklasse, d.h. es wurden alle freie Parameter der Problemklasse gewählt.

Beispiele:

- Finden des kürzesten Weges zwischen der Mensa und dem HG I (bezüglich der euklidischen Norm)
- Ermittlung der ähnlichen Käufergruppen des Onlineshops www.irgendeinshop.de (bezüglich eines konkreten Ähnlichkeitsmaß)
- Unterscheidung von korrekt und nicht-korrekt verpackten Paketen auf einer Förderstrecke mittels einer festinstallierten Kamera
- Zuordnung der Mitarbeiter Meier, Müller und Schmidt auf die drei anstehenden Liefertouren X,Y und Z
- Ermittlung der Auslagerstrategie für das Lager der XYZ GmbH in Dortmund

Algorithmus

Algorithmus

Ein *Algorithmus* ist eine detaillierte und explizite Vorschrift zur schrittweisen Lösung eines Einzelproblems (einer definierten Problemklasse) durch eine Abfolge bekannter Befehle/Operationen.

Beispiele:

- Dijkstra-Algorithmus (Ermittlung eines kürzesten Pfades zwischen einem Start- und Endknoten in einem Graphen)
- Euklidischer Algorithmus (Berechnung des ggT zweier natürlicher Zahlen)
- Quicksort (Algorithmus zur Sortierung von Werten einer Liste)

Wichtige Funktionen

- Über den Befehl `?` wird der Hilfemodus aufgerufen, über `]?` die Package-Hilfe
- In den normalen Modus wechseln (etwa aus dem Hilfemodus heraus): `[Backspace]` oder eine leere Linie
- Alle Stellen sehen wo `func` definiert ist: `apropos("func")`
- Ausführung abbrechen: `[STRG] + [C]`
- Bildschirm leeren: `[STRG]+[L]`
- Julia Programm ausführen: `include("filename.jl")`
- Beenden: `exit()` oder `[STRG]+[D]`

- Einzeilige Kommentare werden über # begonnen
- Mehrzeilige Kommentare werden mit #= und mit =# beendet
- Beispiel:

```
julia> # Ein Kommentar  
julia> #= Dieser Kommentar  
julia> geht über drei Zeilen. Durch "?" gelangt man in den  
julia> Hilfemodus wo etwa "pi" gesucht werden kann =#  
help?> pi
```

- Die Ausgabe kann über ; unterdrückt werden
- Der letzte Wert wird in der Variable ans gespeichert

```
julia> 2+2*2  
6  
julia> ans  
6
```


Variablen

- Eine Variable in julia ist ein Name mit einem assoziierten Wert
- In vielen Programmiersprachen, wie etwa Java, muss der Datentyp einer Variable bei der Deklaration explizit angegeben werden, z.B.:

```
java> int x = 5;
```

- In julia wird eine Typenangabe nicht benötigt:

```
julia> x = 5;                                # Die Zuweisung erfolgt über =
```

- Insbesondere kann einer Variable auch ein Wert eines anderen Datentypes zugewiesen werden:

```
julia> y = 5                                # Hier ist y ein Integer
```

```
julia> y = "Ich bin ein String"             # Hier ist y ein String
```

julia erkennt den Datentypen selbstständig.

- Der Typ einer Variablen `a` kann mit dem Befehl `typeof(a)` ermittelt werden.

Datentypen I

- *Integer*: Ganze und natürliche Zahlen mit verschiedenen Wertebereichen sind möglich, etwa `uint64` für nicht-negative Zahlen und `int64` für ganze Zahlen in einer 64bit Darstellung

Syntax:

```
julia> n = 5
```

- *Float*: Gleitkommazahlen mit verschiedenen Wertebereichen bzw. Genauigkeiten, etwa `Float64`

Syntax:

```
julia> x = 5.0           # 5.0
julia> y = .7            # 0.7
julia> z = 2e-3          # 0.002 = 2*10(-3)
```

- Komplexe und rationale Zahlen sind möglich, werden in dieser Veranstaltung aber nicht weiter benötigt

Syntax:

```
julia> q = 1//4          # 1/4 als Bruch
julia> j = 1+4im         # 1+4i
```

Datentypen II

- **Bool:** Wahrheitswerte *wahr* und *falsch*

Syntax:

```
julia> a = true           # wahr   (1)
julia> b = false         # falsch (0)
```

- **Char:** einzelne Zeichen in einer 32bit Darstellung

Zeichen werden durch einfache Anführungszeichen gekennzeichnet

Syntax:

```
julia> b1 = 'a'           # a (klein)
julia> b2 = 'X'           # X (groß)
julia> b3 = '$'           # das Dollarsymbol $
```

- **String:** Zeichenketten beliebiger Länge

Strings werden durch doppelte Anführungszeichen gekennzeichnet; Sollen Anführungszeichen enthalten sein, so wird dies durch 3 doppelte Anführungszeichen gekennzeichnet

Syntax:

```
julia> s1 = "Hi"           # "Hi"
julia> s2 = """"Oben steht "Hi"!"""" # "Oben steht \"Hi\\\"!"
```

Einfache Funktionen

	Befehl	Bedeutung
mathematische Operationen	+	Addition
	-	Subtraktion
	/	Division
	*	Multiplikation
	%	Modulo
	^	Potenz
logische Operationen	&&	logisches UND
		logisches ODER
	!	logisches NICHT
Vergleiche	==	Gleich
	< (<=)	Kleiner (Kleiner oder Gleich)
	> (>=)	Größer (Größer oder Gleich)
	!=	Ungleich

Datentypen III

- **Array:** Ein Array ist eine geordnete Sammlung von Objekten in einem ein- oder mehrdimensionalen Feld, wobei die Datentypen der einzelnen Objekte verschieden sein können

Syntax:

```
julia> x = [1,2,3,4,5]           # Ein Spaltenvektor
julia> y = Array{Float64}(undef,1,3) # Eine 1x3 Matrix
                                     # undef bedeutet, dass keine Werte übergeben werden
julia> A = [[1 , 0] [0 , 1]]    # Eine 2-dimensionale Einheitsmatrix
```

Arrays werden insbesondere für Vektoren, Matrizen, Listen und Tabellen verwendet

- Der Zugriff auf einzelne Elemente eines Array erfolgt über Angabe des Index in eckigen Klammern:

```
julia> x[2]           # Ruft das zweite Element von x auf
julia> x[2:4]         # Ruft das 2-te bis 4-te Element von x auf
```

- julia legt automatisch anhand der Deklaration den zulässigen Datentypen der Elemente fest
Werden nur Integer Werte gespeichert, so können nur noch diese in dem Array gespeichert werden.
- Elemente eines Array können verändert werden

Datentypen IV

Der Doppelpunkt : besitzt verschiedene Funktionen. Eine wichtige ist das Erzeugen von Range-Objekten.

- Im einfachsten Fall werden Start- und Endpunkt gesetzt. Die Schrittweite wird implizit als Eins angenommen:

```
julia> r = 1:10          # 1,2,3,4,5,6,7,8,9,10
```

- Soll die Schrittweite angepasst werden, so wird dies in der Form Start:Schrittweite:Ende erzeugt:

```
julia> r = 1:2:10        # 1,3,5,7,9
```

```
julia> r = 6.5:-1:1.5    # 6.5, 5.5, 4.5, 3.4, 2.5, 1.5
```

- Eine Anwendungsmöglichkeit für Range-Objekte ist die Erzeugung von Arrays mit Hilfe der collect() Funktion:

```
julia> collect(0:2:1000); # Array der geraden Zahlen von 0 bis 1000
```

- Eine weitere wichtige Anwendung sind Schleifen und die sog. *comprehension*, etwa:

```
julia> x = [i*j for i=1:10, j =-5:1:5]
```

Datentypen V

- Arrays können mittels des `cat` Befehls verkettet werden

Befehl	Beschreibung	Spezieller Syntax
<code>cat(A...;dims=k)</code>	Verketten der Arrays entlang der Dimension(en) <code>k</code>	
<code>vcat(A...)</code>	wie <code>cat</code> mit <code>k=1</code> („Verkettung untereinander“)	<code>[A; B; C]</code>
<code>hcat(A...)</code>	wie <code>cat</code> mit <code>k=2</code> („Verkettung nebeneinander“)	<code>[A B C]</code>

- Beispiel:

```
julia> A = [1,2];
julia> B = [3,4];
julia> [A B]
2×2 Array{Int64,2}:
 1  3
 2  4
```

```
julia> [A;B]
4-element Array{Int64,1}:
 1
 2
 3
 4
```

Datentypen VI

- *Tuple*: Tuple sind sehr ähnlich zu Arrays, mit dem Unterschied, dass Tuple nicht verändert werden können

Syntax:

```
julia> t = (1, 1.0, 2) # Ein Tuple mit drei nicht manipulierbaren Einträgen
julia> typeof(t)
Tuple{Int64,Float64,Int64}
```

- *Dictionary*: In einem Dictionary werden für eindeutige Schlüssel dazugehörige Werte gespeichert

Syntax:

```
julia> dict1 = Dict{"a" => 1, "b" => 2, "c" => 3}
julia> dict2 = Dict{String,Integer}{"a"=>1, "b" => 2}
julia> dict1["a"]
1
```

Dictionaries sind ungeordnet (wie eine Menge) und werden etwa dazu verwendet Ergebnisse abzuspeichern und abzufragen

Zusammengesetzte Ausdrücke

- Mittels `begin...end` oder `(...;...;...)` können Ketten von Befehlen durchgeführt werden⁹
- Bei einer Zuweisung wird der letzte berechnete Ausdruck der Variablen zugewiesen
- Syntax:

```
a = begin
    b = 1
    c = 3
    (b*c)/(b+c)           # Hier wird a = (b*c)/(b+c) gesetzt
end
```

```
d = begin e = 4; f = 2; e/f end # d = e/f
oder
a = ( b = 1; c = 3; (b*c)/(b+c) )
```

⁹Im Folgenden wird auf die Angabe von „julia>“ im Code verzichtet.

If-else-elseif

Bedingungen werden in julia über folgenden Syntax implementiert:

```
if <Bedingung>
    <Befehle>
elseif <Bedingung> #es kann keine, eine oder mehrere elseif Bedingung verwendet werden
    <Befehle>
else
    <Befehle>
end
```

Beispiel:

```
if x < y
    println("x ist kleiner als y")
elseif x > y
    println("x ist größer y")
else
    println("x ist gleich y")
end
```

while- und for-Schleifen

- Befehle innerhalb einer while-Schleife werden so lange ausgeführt, bis die im Schleifenkopf formulierte Bedingung nicht mehr erfüllt ist

```
while <Bedingung>  
    <Befehle>  
end
```

- Die for-Schleife wird zum Iterieren über einzelne Elemente einer Sammlung verwendet:

```
for <element> in <Sammlung>  
    <Befehle>  
end
```

- Beispiele

```
n = 10  
while n > 0  
    println(n)  
    n = n - 1;  
end
```

```
for i in 10:-1:1  
    println(i)  
end
```

Funktionen I - Syntax

Eigene Funktionen können über den folgenden Syntax definiert werden:

```
function <Funktionsname>(<Eingabewerte>)  
    <Befehle>  
end
```

- Wird eine `return` Anweisung nicht verwendet, wird standardmäßig das letzte berechnete Wert zurückgegeben
- `return` Anweisungen sind also insbesondere dann notwendig, wenn etwa über eine `if` Abfrage der Rückgabewert bestimmt wird
- Die Funktion `func` berechnet den Mittelwert zweier Werte:

```
function func(arg1,arg2)  
    (arg1+arg2)/2;  
end
```

- Julia bietet auch eine sehr simple Notation für Funktionen an:

```
func(arg1,arg2) = (arg1+arg2)/2
```

Funktionen II - Angabe von Parametertypen

- Datentypen der Eingabewerte können mittels `::<Datentyp>` nach dem jeweiligen Eingabeparameter spezifiziert werden:

```
func(arg1::Float64,arg2::Float64) = (arg1+arg2)/2
```

- Datentyp des Rückgabewertes können mittels `::<Datentyp>` hinter der schließenden Klammer der Eingabewerte spezifiziert werden:

```
func2(x,y)::Int64 = x*y
```

- Operatoren wie die Addition `+` sind Funktionen mit einem speziellen Syntax
- julia bietet auch die Möglichkeit Funktionen ohne Namen, sog. anonyme Funktionen, zu definieren:

```
(x,y) -> x*y
```

Anonyme Funktionen werden insbesondere dann verwendet, wenn die Funktion nur einmal benötigt wird

Funktionen III - Methoden

Betrachten wir einmal die Funktion `f` und die dazugehörige Ausgabe:

```
f(x,y) = (x*y)/(x+y)
```

```
f (generic function with 1 method)
```

- Die Funktion `f` ist ein Objekt vom Typ `generic function` mit einer Methode
- Eine Methode einer Funktion ist die konkrete Ausgestaltung einer Funktion
- Die Addition besitzt 163 Methoden, wobei jede Methode für bestimmte Datentypen bestimmt ist
- `generic function` lässt sich also als das theoretische Konzept der Funktion verstehen, wobei jede Methode eine konkrete Ausprägung ist
- Bei der Addition ist eine Methode z.B. dafür da zwei Gleitkommazahlen zu addieren, eine andere hingegen für zwei Arrays
- Der Befehl `methods(<funktionsname>)` listet alle Methoden der Funktion auf und gibt die Stelle an, wo die Methode implementiert ist

Funktionen IV - Optionale Argumente

- Funktionen können auch optionale Argumente enthalten
- Optionale Argumente werden mit einem Standardwert angelegt, der von der Funktion verwendet wird, wenn für das optionale Argument kein Wert übergeben wird

- Beispiel:

```
function func(x,y,z=0)
    return x+y+z
end
```

Hier ist z ein optionales Argument

- Werden nur Werte für x und y übergeben, so wird für z der Wert 0 genommen

```
func(1,1)
2
```

- Werden drei Werte übergeben, so wird für z der dritte Wert genommen

```
func(1,1,1)
3
```

Funktionen V - Schlüsselwort Argumente

Bei den bisherigen Definitionen von Funktionen kam es auf die Reihenfolge der Argumente an.

- Zunächst ein Beispiel:

```
function func(x ; y = 1)
    return y/x
end
```

- y ist ein optionales Argument mit Schlüsselwort, d.h. wenn ein Wert für y übergeben werden soll, dann muss das unter Angabe der Schlüsselwortes (hier y) erfolgen

```
func(1,y=2)
2.0
```

- Schlüsselwort Argumente können an einer beliebiger Stelle innerhalb der Argumentenliste übergeben werden

```
func(y = 3, 4)
0.75
```

- Schlüsselwort Argumente schaffen also insbesondere Übersicht bei Funktionen mit vielen Argumenten
- Schlüsselwort Argumente werden bei der Definition nach dem Semikolon angegeben, dadurch sind sie von den normalen optionalen Argumenten unterscheidbar

Rekursion

Oftmals lassen sich Funktionen wesentlich einfacher *rekursiv* realisieren.

Rekursion

Eine Funktion f heißt *rekursiv*, wenn

- der Funktionsrumpf einen Aufruf der Funktion f selbst (direkt) oder eine andere Funktion g , die wiederum f aufruft (indirekt), enthält
- f eine Terminierungsbedingung (auch Rekursionsanker genannt) enthält
- jede Eingabe nach endlich vielen Schritten terminiert

Einige Datenstrukturen, wie etwa Bäume, werden ebenfalls Rekursiv realisiert.

Beispiel: Die Fakultät für eine natürliche Zahl (d.h. $n \geq 0$) ist definiert als

$$n! = \begin{cases} 1 & \text{falls } n = 0 \\ n \cdot (n-1)! & \text{sonst} \end{cases}$$

Diese Definition lässt direkt rekursiv umsetzen:

```
function fakultaet(n::Int)
    if(n < 0)
        error("n muss nicht-negativ sein!")
    elseif (n == 0)
        return 1
    else
        return n * fakultaet(n-1)
    end
end
```

Zusammengesetzte Datentypen

In Julia können zusammengesetzte Datentypen mit dem Schlüsselwort **struct** definiert werden:

```
struct <Name>  
    <Variablen mit optionalen Datentypen>  
end
```

Beispiel:

```
struct MeinDatentyp          # struct <Name>  
    a                        # a kann jeden Datentyp aufnehmen  
    b::Int                   # b muss ein Integer sein  
    c::String                 # c muss ein String sein  
end
```

Standard Konstruktoren

- Mit Konstruktoren lassen sich neue Objekte eines zusammengesetzten Datentyps erzeugen
- Standardmäßig werden bei der Definition eines zusammengesetzten Datentyps zwei Konstruktoren automatisch mit erzeugt
- Der eine Konstruktor nimmt alle Argumente und versucht diese passend umzuwandeln, sodass ein neues Objekt erzeugt werden kann
- Der andere Konstruktor nimmt nur Argumente mit den exakt gleichen Typen und erzeugt ein neues Objekt
- Syntax:
`<Variable> = <Name des zusammengesetzten Datentyps>(<Argumente>)`
- Beispiel: Neue Objekte vom Typ `MeinDatentyp` können mittels des Konstruktors erzeugt werden (Hier ist die Reihenfolge wie üblich wichtig):
`x = MeinDatentyp('x', 5, "Fünf")`

Eigene Konstruktoren

Eigene Konstruktoren können wie Funktionen erzeugt werden, wobei der Funktionsname gleich dem Namen des zusammengesetzten Datentyps sein muss.

Beispiel:

```
struct MyNumber  
    myint::Int  
end
```

```
MyNumber() = MyNumber(0)  
# floor() rundet x ab, Int() wandelt das Ergebnis in einen Integer um  
MyNumber(x::Float) = Int(floor(x))
```

Konstruktoren werden außerhalb der **struct** Umgebung definiert!

Zugriff auf die Attribute eines Struct

- Mittels der Funktion `fieldnames()` können die Namen der Attribute aufgelistet werden

- Beispiel:

```
l = fieldnames(MyNumber)
```

- Mittels des Punktoperators können die Werte der Attribute ausgelesen werden:

```
x = MyNumber(3)
```

```
y = x.myint          # y hat nun den Wert 3
```

- Achtung: Nach dem Erzeugen sind Objekte von zusammengesetzten Datentypen nicht veränderbar! (engl. immutable)
- Soll ein Objekt veränderbar sein, so muss dies bereits bei der Definition des zusammengesetzten Datentyps mit dem Schlüsselwert `mutable` angegeben werden:

```
mutable struct MyNumber2  
    myint::Int  
end
```

Hilfreiche Funktionen

- Konvertieren eines Wertes x in einen anderen Datentypen: `convert(Datentyp, x)`
- Runden eines Wertes x : `round(x)` (Achtung: Das Ergebnis ist ein Float!)
- Runden eines Wertes x als Integer: `round(Int, x)`
- Abrunden eines Wertes x : `floor(x)`
- Aufrunden eines Wertes x : `ceil(x)`
- Vektor mit n Einträgen gleich 0: `zeros(n)`
- Vektor mit n Einträgen gleich 1: `ones(n)`
- Matrix mit $n \times m$ Einträgen gleich 0: `zeros(n, m)`
- Matrix mit $n \times m$ Einträgen gleich 1: `ones(n, m)`
- Prüfen ob der Wert x in dem Vektor/ der Matrix A vorkommt:

`in(x, A)`

`x in A`

Der Packagemanager

- Packagemanager nutzen: `]` im REPL oder `using Pkg`
- Liste alle installierten Pakete: `Pkg.status()`
- Update für alle Pakete: `Pkg.update()`
- Paket installieren: `Pkg.add("PackageName")`
- Paket einbinden (vorher muss es installiert werden): `using PackageName`
- Paket entfernen: `Pkg.rm("PackageName")`

(Pseudo-) Zufallszahlen

- Paket einbinden: `using Random`
- Zufallssaat setzen: `seed!(seednumber)`
- Gleichverteilte Zufallszahl aus dem Intervall $[0, 1)$: `x = rand()`
- Gleichverteilte Zufallszahl aus den Werten 1, 2, 3, 4, 5: `x = rand(1:5)`
- Gleichverteilter Zufallswert aus der Menge A : `x = rand(A)`
- Matrix der Dimension $n \times m$ von Zufallszahlen: `vec = rand(n,m)`
- Matrix der Dimension $n \times m$ von Zufallswerten aus der Menge A : `vec = rand(A,n,m)`
- Standardnormalverteilte Zufallszahl: `x = randn()`
- Zufällige Permutation der Elemente von A : `shuffle(A)`
- Andere Verteilungen einbinden:
`using Distributions`
`verteilung = Bernoulli(0.2) # Beispiel`
`x = rand(verteilung)`

Dataframe

Dataframe

Ein Dataframe ist ein Datenstruktur, das Informationen geordnet nach Zeilen und Spalten beinhaltet. Er ist vergleichbar mit einer Tabelle. Eine Zeile stellt ein Objekt/ einen Datensatz dar. Eine Spalte stellt ein Merkmal der Objekte dar. In der ersten Spalte steht üblicherweise eine ID um die einzelnen Objekte zu unterscheiden.

Beispiel eines Dataframes in Julia:

Row	Länge	Breite	Höhe
	Int64	Int64	Int64
-----	-----	-----	-----
1	1	1	3
2	3	1	2
3	2	1	2

Julia Syntax für Dataframes

- Package einbinden: `Pkg.add("DataFrames")`
- Leeren Dataframe erzeugen: `df = Dataframe()`
- Dataframe mit Spaltennamen und Spaltenwerten erzeugen:
`df = DataFrame(A = 1:5, B = ["H", "A", "L", "L", "O"])`
- Dataframe aus einer Matrix *A* erzeugen: `df = DataFrame(A)`
(standard Spaltennamen werden automatisch ergänzt)
- Abrufen der Spalte mit dem Namen *A*: `df[:A]` oder `df.A`
- Spalten können auch über den Index aufgerufen werden: `df[1]` (Hier wird die 1. Spalte aufgerufen)
- Eine neue Spalte (hier C) kann direkt mit Inhalten dem Dataframe *df* hinzugefügt werden:
`df[:C] = [1,1,2,2,1]`

Eckdaten eines Dataframes

- Die Größe eines Dataframes, d.h. Zeilenanzahl \times Spaltenanzahl, eines Dataframes `df` erhält man über `size(df)`
- Die ersten n Zeilen eines Dataframes `df` können mittels `first(df,n)` ausgegeben werden
- Die letzten n Zeilen mittels `last(df,n)`
- Die Spaltennamen können mittels `names(df)` ausgegeben werden
- Informationen wie Minimum, Maximum, Anzahl der fehlenden Werte, Mittelwert, Median und Datentyp einer Spalte kann mit `describe(df)` kompakt ausgegeben werden

- Auswahl konkreter Bereiche durch `df[<Zeilenauswahl>,<Spaltenauswahl>]`, z.B.

```
df[3:8,:]      # 3. bis 8. Zeile mit allen Spalten
df[[1,5,6],:]  # Zeilen 1,5 und 6 mit allen Spalten
df[:,2:5]      # 2. bis 5. Spalte mit allen Zeilen
df[:,[:A,:B]]  # Alle Zeilen mit den Spalten :A und :B
```

- Insbesondere können logische Aussagen zur Auswahl genutzt werden:

```
df[df[:A].>0.5,:] #Alle Zeilen bei denen der Wert in Spalte A größer als 0.5 ist
```

Manipulation von Dataframes I

- Spalten umbenennen:

```
rename!(df, :AlterName => :NeuerName)  
rename!(df, f => t for (f, t) = zip([:A1, :A2], [:N1, :N2]))
```

- Zeilen hinzufügen:

```
push!(df, [wert1,wert2,wert3])
```

- Zeilen löschen:

```
deleterows!(df, 3:5)      # Löschen Zeilen 3,4,5 in df
```

- Spalten löschen:

```
delete!(df, :X)          # Lösche Spalte :X in df
```

- Der Befehl `groupby()` fasst unterteilt einen Dataframe in verschiedene Unter-Dataframes anhand der Einträge einer Spalte. Alle Zeilen mit dem gleichen Wert in dieser Spalte werden zusammengefasst:

```
groupby(df, :Spaltenname)
```

Manipulation von Dataframes II

- Alle Zeilen mit einem konkreten Wert in einer Spalte:

```
df[df[:Spaltenname] .== Wert, :]
```

- Mittels der `by()` Funktion wird der Dataframe wie bei `groupby()` gruppiert, anschließend kann eine Funktion auf die Spalten der Unter-Dataframes übergeben werden:

```
by(df, :A, df -> func(df[:B]))
```

- Einfaches sortieren anhand der ersten Spalte:

```
sort!(df)  
sort!(df, ref=true)
```

- Anhand ausgewählter Spalten sortieren:

```
sort!(df, (:SpalteA, :SpalteB))
```

- Sortieren mit verschiedenen Ordnungen/Sortierungen:

```
sort!(df, (order(:SpalteA, by = uppercase),  
           order(:SpalteB, rev = true)));
```

Spalten- und Zeilenweise Schleifen

■ Zeilenweise:

```
for zeile in eachrow(df)
    # mache etwas, etwa:
    println(zeile)
    println(typeof(zeile))
end
```

■ Spaltenweise:

```
for spalte in eachcol(df)
    # mache etwas, etwa
    println(spalte)
    println(typeof(spalte))
end
```

Konventionen und Empfehlungen für Namen

- Variablen:
 - Namen sollen in Kleinbuchstaben geschrieben werden
 - Trennung durch Unterstriche `_` soll nur verwendet werden, wenn der Name sonst zu schwer zu lesen ist
- Funktionen:
 - Namen in Kleinbuchstaben, ohne Unterstriche
 - Wird ein Argument verändert, so wird dies durch ein Ausrufezeichen `!` im Namen gekennzeichnet
 - Es sollte immer das erste Argument sein, das verändert werden soll
 - Beispiel: `sort()` gibt das sortierte Feld zurück und `sort!()` sortiert direkt im übergebenen Feld
- Module und Typen:
 - Namen beginnen mit einem Großbuchstaben
 - Jedes Wort im Namen soll mit einem Großbuchstaben beginnen und es sollen keine Unterstriche verwendet werden
 - Beispiel: `module MeinModul` oder `struct MeinDatentyp`
- Weitere (fortgeschrittene) Style Empfehlungen:
<https://docs.julialang.org/en/v1/manual/style-guide/#Style-Guide-1>