Programmierkurs Julia

FSS 2023

Lehrstuhl Prof. Martin Schlather*

12. Februar 2024

Universität Mannheim

*Das Skript wurde von Moritz Fromm und Johannes Nägele erstellt.

Inhaltsverzeichnis

Kurs 1 **3**

1.1 Vorwort . . . 3
1.2 Arithmetik und Operatoren . . . 3
1.3 Variablen . . . 4
1.4 Strings . . . 5
1.5 Arrays . . . 5
1.5.1 Lineare Algebra . . . 7
1.5.2 Indexierung . . . 8
1.5.3 Elemente hinzufügen oder verändern . . . 9
1.5.4 Werkzeuge für mehrdimensionale Arrays . . . 10
1.6 Typen . . . 12
1.7 Funktionen . . . 13
1.7.1 Einführung und Motivation . . . 13
1.7.2 Verschiedene Methoden . . . 14
1.7.3 Keyword arguments . . . 15
1.7.4 Broadcasting . . . 15
1.7.5 Anonyme Funktionen . . . 17
1.7.6 Inputtypen . . . 17
1.8 Style Guide . . . 18
1.8.1 Variablenbenennung . . . 18
1.8.2 Zeichenkodierung . . . 18

Kurs 2 **20**

2.1 Logische Typen und Operatoren . . . 20
2.1.1 Short-circuit evaluation . . . 21
2.2 Operator-Hierarchie . . . 22
2.3 Kontrollstrukturen . . . 22
2.3.1 Conditional Evaluation . . . 22
2.3.2 Compound Expressions . . . 24

Kurs 3 **25**

3.1 Exkurs: Map . . . 25
3.2 Rekursion . . . 27
3.3 `for`-Loops . . . 27
3.4 `while`-Loops . . . 29

Kurs 4 **31**
4.1 Multiple dispatch . . . 31
4.1.1 Beispiel . . . 31
4.2 Pakete . . . 33
4.3 Pipes . . . 33
4.4 DataFrames . . . 34
4.4.1 Datentypen und kategorische Variablen . . . 38
4.4.2 Datensätze zusammenführen . . . 40

Kurs 5 **42**
5.0.1 Variablen transformieren und anpassen . . . 42
5.0.2 Fehlende Werte . . . 44
5.1 Plots . . . 47
5.1.1 Einführung . . . 47
5.1.2 Scatterplots . . . 52

Kurs 6 **58**
6.1 I/O . . . 58
6.1.1 CSV . . . 58
6.1.2 Excel . . . 59
6.1.3 JSON . . . 59
6.1.4 JLD2 . . . 60
6.1.5 FileIO . . . 61
6.2 Lineare Regression . . . 62
6.3 Typen . . . 63
6.3.1 Werkzeuge für Types . . . 64
6.3.2 Type declarations . . . 64
6.3.3 Eigene Typen . . . 66
6.3.4 Unions . . . 70

Kurs 7 **71**
7.1 Metaprogramming . . . 71
7.1.1 Expressions . . . 71
7.1.2 Mickey Mouse Example . . . 73
7.1.3 Eigene Makros . . . 74
7.1.4 Ausblick: Loop unrolling (SIMD) . . . 76
7.1.5 Ausblick: Eigene Syntax bzw. Modelldefinitionen . . . 76

Kurs 8 **79**
8.1 Ein fancy Beispiel . . . 79

Kurs 9 **84**
9.1 Zufallszahlen . . . 84
9.1.1 Distributions.jl . . . 86

9.2 Performance 86
9.2.1 Type instabilities 87
9.2.2 Row vs. column major 89
9.2.3 Ausblick 91
9.2.4 Kontextabhängige Optimierung 94

Kurs 10 (Bonusvorlesung) **96**
10.1 Eine Meta-Diskussion über Julia 96
10.1.1 Was sollte eine Programmiersprache können? 96
10.1.2 Klassische Objektorientierung: Basics 97
10.1.3 Typ-Hierarchie 98
10.1.4 Warum wollen wir keine Vererbung? 99
10.1.5 Mögliche Probleme und Verbesserungsmöglichkeiten 106
10.1.6 Weiterführende Resourcen 107

Kurs 1

1.1 Vorwort

In diesem Kurs wollen wir in die Programmiersprache Julia einsteigen. Warum gerade Julia? Die kurze Antwort wäre wohl: Julia ist genauso praktisch wie Python, aber gleichzeitig so schnell wie C. Eine etwas längere Antwort könnt ihr euch vielleicht nach diesem Kurs selbst geben (Spoiler: Julia ist auch in vielerlei anderer Hinsicht einfach geil).

Grundsätzlich sollte es definitiv möglich sein, mit wenig oder sogar gar keiner Programmiervorkenntnis zu starten. Wir versuchen aber, euch trotzdem nicht zu langweilen und zumindest einen kleinen Vorgeschmack auf die große weite (und schöne) Welt der Programmierung zu geben. Jedenfalls gilt: Wenn ihr irgendwo nicht mehr weiterkommt oder Erklärungen unklar sind, dann sagt Bescheid – Rückmeldung ist sowieso immer willkommen!

1.2 Arithmetik und Operatoren

Als allererstes wollen wir uns einfache Beispiele für Arithmetik anschauen:

[1]:
```
# Addition
4 + 4
```

[1]: 8

[2]:
```
# Subtraktion
1.5 - 0.5
```

[2]: 1.0

[3]:
```
# Potenz
4^3
```

[3]: 64

[4]:
```
# Modulo
4 % 3
```

[4]: 1

Wie man vielleicht bereits gesehen hat, wird Code, der hinter einem `#` steht, nicht ausgeführt. Dies nennt man einen Kommentar:

[5]:
```
# Mit Kommentaren können wir Code erklären, dokumentieren und lesbarer machen
```

1.3 Variablen

Im Folgenden wird einer Variablen `x` der Wert `4` zugewiesen.

[6]:
```
x = 4
```

[6]: 4

Wir können Variablen wie (zum Beispiel) Zahlen verwenden, Operationen damit verhalten sich genauso:

[7]:
```
y = 6
(x + y)^2
```

[7]: 100

In Julia gibt es viele Kurzschreibweisen, die relativ praktisch sind. Ein Beispiel wäre `a += b` was zu `a = a + b` äquivalent ist.

[8]:
```
z = 1
z += 1
```

[8]: 2

Elementar ist auch die Funktion `println`. Diese printet (druckt) uns direkt den Wert eines eingegebenen Arguments in die Konsole:

[9]:
```
txt = "Hallo"
println(1)
println(x)
println(z)
println(txt)
```

```
1
4
```

```
2
Hallo
```

1.4 Strings

Das `"Hallo"` im obigen Codeblock nennt man einen String. Wenn wir also Anführungszeichen setzen, können wir quasi normalen Text als Wert bzw. Variablenbelegung verwenden. Mit der Funktion `string` können wir außerdem Werte in Strings umwandeln (conversion) bzw. Strings zusammenfügen (string concatenation).

[10]:
```
string(1) # conversion
```

[10]:
```
"1"
```

[11]:
```
apples = 3
head = "Ich habe übrigens "
string(head, apples, " Äpfel") # concatenation
```

[11]:
```
"Ich habe übrigens 3 Äpfel"
```

Ein bisschen hübscher geht es allerdings mit sogenannter *string interpolation*. Das bedeutet: Wir können mittels `$` Werte direkt in einen String einfügen:

[12]:
```
apples = 3
head = "Ich habe übrigens $(apples)" # string interpolation
head * " Äpfel" # Kurzschreibweise: concatenation mit *
```

[12]:
```
"Ich habe übrigens 3 Äpfel"
```

1.5 Arrays

Ein Array ist eine Sammlung von Objekten, die in einem mehrdimensionalen Gitter gehalten werden. In Julia sind Arrays extrem mächtig, deshalb werden wir hier ein bisschen Zeit investieren. Unser erstes Beispiel ist aber ganz simpel ein `Vector`, also ein eindimensionaler Array, mit den Elementen 1, 2, 3, 4:

[13]:
```
# (Spalten-)Vektor
x = [1, 2, 3, 4]
```

[13]:
```
4-element Vector{Int64}:
 1
 2
 3
```

```
4
```

Arrays können nicht nur Zahlen, sondern auch alle möglichen anderen Datentypen (und somit auch andere Arrays) enthalten:

[14]:
```
[] # leerer Array
```

[14]:
```
Any[]
```

[15]:
```
[1, 2, "Hello", 3.0, 4.0, "World"]
```

[15]:
```
6-element Vector{Any}:
 1
 2
  "Hello"
 3.0
 4.0
  "World"
```

[16]:
```
a = [[1, 2], [3, 4]]
```

[16]:
```
2-element Vector{Vector{Int64}}:
 [1, 2]
 [3, 4]
```

Wenn die Argumente in den eckigen Klammern durch einzelne Semikolons (`;`) oder neue Zeilen abgetrennt sind, dann werden diese nicht mehr als Elemente betrachtet, sondern deren Inhalte *vertikal* aneinandergehängt.

[17]:
```
# Vergleiche mit vorhergehendem Beispiel!
b = [[1, 2]; [3, 4]]
```

[17]:
```
4-element Vector{Int64}:
 1
 2
 3
 4
```

Ähnlich funktioniert die Trennung der Argumente durch Tabs oder Leerzeichen. Dann werden deren Inhalte nämlich *horizontal* zusammengefügt:

[18]:
```
# Vergleiche mit vorhergehendem Beispiel!
c = [[1, 2] [3, 4]] # Achtung: Länge der Argumente muss gleich sein!
```

[18]:
```
2×2 Matrix{Int64}:
 1  3
```

```
 2  4
```

[19]:
```
# Zeilenvektor 1x4
y = [1 2 3 4]
```

[19]:
```
1×4 Matrix{Int64}:
 1  2  3  4
```

Wenn wir diese Konzepte kombinieren, dann können wir uns weiter an höherdimensionale Arrays herantasten: Hier kommt der einfachste Weg, um händisch eine Matrix zu definieren. Durch die Leerzeichen bekommt man Zeilenvektoren, durch das Semikolon werden diese untereinander in eine Matrix gepackt:

[20]:
```
mat = [1 2; 3 4]
```

[20]:
```
2×2 Matrix{Int64}:
 1  2
 3  4
```

Für die, die es genauer wissen wollen: Auch hier verwenden wir eigentlich wieder Kurzschreibweisen; die umständliche Variante sieht so aus:

[21]:
```
# vcat: vertical concatenation, hcat: horizontal concatenation
vcat(hcat(1, 2), hcat(3, 4))
```

[21]:
```
2×2 Matrix{Int64}:
 1  2
 3  4
```

Dabei sind `vcat` und `hcat` Spezialfälle der Funktion `cat`, die wir später noch kennenlernen werden.

1.5.1 Lineare Algebra

Für die Erstellung einer Einheitsmatrix laden wir noch ein zusätzliches Modul aus Julias Standardbibliothek:

[22]:
```
using LinearAlgebra # für Operator I
3I(4)
```

[22]:
```
4×4 Diagonal{Int64, Vector{Int64}}:
 3  ·  ·  ·
 ·  3  ·  ·
 ·  ·  3  ·
 ·  ·  ·  3
```

Mit Vektoren und Matrizen können wir dann wie gewohnt Addition und Multiplikation rechnen:

[23]:
```
z = [0, 2]
println(x + x)
println(mat + mat)
println(mat * z)
```

```
[2, 4, 6, 8]
```

```
[2 4; 6 8]
[4, 8]
```

Wichtige Operationen sind auch:

[24]:
```
# Matrixinversion
mat^(-1) * mat # wegen numerischer Ungenauigkeit nicht genau I(2)
```

[24]:
```
2×2 Matrix{Float64}:
 1.0          0.0
 2.22045e-16  1.0
```

[25]:
```
# Löse LGS: mat * ? = z
mat\z
```

[25]:
```
2-element Vector{Float64}:
  2.0
 -0.9999999999999999
```

1.5.2 Indexierung

Man kann über Indizes gezielt auf ein Element in einem Array zugreifen:

[26]:
```
mat_index = [1 2 "Hello"; 3 4 "World"]
mat_index[1, 3]
```

[26]:
```
"Hello"
```

Für mehrere Elemente gibt zahlreiche weitere Möglichkeiten der Indizierung:

[27]:
```
# Zeile 1 und Spalte 1-2:
println(mat_index[1, 1:2])
# Elemente über einen Vektor von Indizes spezifizieren
println(mat_index[1, [1,3]])
# die komplette (erste) Spalte ausgeben
println(mat_index[:, 1])
```

```
# den letzten Wert einer Zeile ausgeben
println(mat_index[2, end])
```

```
Any[1, 2]
Any[1, "

Hello"]
Any[1, 3]
World
```

1.5.3 Elemente hinzufügen oder verändern

Betrachten wir wieder einen einfachen eindimensionalen Array:

[28]:
```
x = [1, [2, 3], "hello"]
```

[28]:
```
3-element Vector{Any}:
 1
  [2, 3]
  "hello"
```

[29]:
```
# gezielt ein Element verändern (slicing)
x[1] = 2
println(x)
```

```
Any[2, [2, 3], "hello"]
```

[30]:
```
# Arrays aneinanderhängen
y = [x, x]
```

[30]:
```
2-element Vector{Vector{Any}}:
 [2, [2, 3], "hello"]
 [2, [2, 3], "hello"]
```

[31]:
```
# ein Element am Schluss hinzufügen
push!(x, 4)
```

[31]:
```
4-element Vector{Any}:
 2
  [2, 3]
  "hello"
 4
```

[32]:
```
# ein Element am Anfang hinzufügen
pushfirst!(x, "Z")
```

```
[32]: 5-element Vector{Any}:
       "Z"
      2
       [2, 3]
       "hello"
      4
```

```
[33]: # entferne und ersetze Elemente an einer genauen Position
      splice!(x, 3:4, [1, 5, 10])
      println(x)
```

```
Any["Z", 2, 1, 5, 10, 4]
```

Analog gibt es die Funktionen `pop!`, `popfirst!` und `deletat!` um Elemente zu entfernen.

1.5.4 Werkzeuge für mehrdimensionale Arrays

Im Grunde genommen funktioniert für mehrdimensionale Arrays alles analog:

```
[34]: twodim1 = [1 2 "Hello"; 3 4 "Geeks"]
      twodim2 = [5 6 7; 8 9 10]
```

```
[34]: 2×3 Matrix{Int64}:
       5  6   7
       8  9  10
```

```
[35]: # gezielt Elemente verändern (slicing)
      twodim1[1, 1] = 7
      println(twodim1)
```

```
Any[7 2 "Hello"; 3 4 "Geeks"]
```

```
[36]: # Arrays vertikal aneinanderhängen (aufpassen mit Dimensionen!)
      [twodim1; twodim2]
```

```
[36]: 4×3 Matrix{Any}:
       7  2    "Hello"
       3  4    "Geeks"
       5  6   7
       8  9  10
```

```
[37]: # Arrays horizontal aneinanderhängen (aufpassen mit Dimensionen!)
      [twodim1 twodim2]
```

```
[37]: 2×6 Matrix{Any}:
       7  2  "Hello"  5  6   7
       3  4  "Geeks"  8  9  10
```

Dementsprechend werden im Folgenden aber *nicht* die Elemente der einzelnen Arrays aneinandergehängt, sondern die einzelnen Arrays quasi wieder als Elemente aufgefasst:

```
[38]: # erzeugt zweidimensionalen Array mit den jeweiligen Arrays als Einträgen
      [twodim1, twodim2]
```

```
[38]: 2-element Vector{Matrix{Any}}:
       [7 2 "Hello"; 3 4 "Geeks"]
       [5 6 7; 8 9 10]
```

Mithilfe der Funktion `cat` können wir zudem höherdimensionale Arrays bauen:

```
[39]: # Erstellung eines 3D-Arrays
      multidim = cat(
          [1 2 3; 3 4 5; 5 6 7],
          ["a" "b" "c"; "c" "d" "e"; "e" "f" "g"],
          dims = 3
      )
```

```
[39]: 3×3×2 Array{Any, 3}:
      [:, :, 1] =
       1  2  3
       3  4  5
       5  6  7

      [:, :, 2] =
       "a"  "b"  "c"
       "c"  "d"  "e"
       "e"  "f"  "g"
```

```
[40]: # Indexierung genau wie in den anderen Fällen
      multidim[1, 3, 1]
```

```
[40]: 3
```

Weitere nützliche Funktionen sind:

```
[41]: # Umformung der Matrix zu einem Vektor
      vec(twodim1)
```

```
[41]: 6-element Vector{Any}:
       7
```

```
 3
 2
 4
  "Hello"
  "Geeks"
```

[42]:
```
# Umformung zu einer Matrix mit anderen Dimensionen
reshape(twodim1, (3, 2))
```

[42]:
```
3×2 Matrix{Any}:
 7  4
 3   "Hello"
 2   "Geeks"
```

[43]:
```
println(length(x)) # Länge
println(size(multidim)) # Dimensionen
println(zeros(3, 3)) # Matrix mit Nullen
println(ones(3, 3)) # Matrix mit Einsen
```

```
6
(3, 3, 2)
[0.0 0.0 0.0; 0.0 0.0 0.0; 0.0 0.0 0.0]
[1.0 1.0 1.0; 1.0 1.0 1.0; 1.0 1.0 1.0]
```

1.6 Typen

Grundsätzlich hat in Julia jeder Wert (value) einen bestimmten Typ, einige davon haben wir bereits kennen gelernt:

[44]:
```
# Integer
typeof(2)
```

[44]:
```
Int64
```

[45]:
```
# Float
typeof(2.0)
```

[45]:
```
Float64
```

[46]:
```
# String
typeof("2")
```

[46]:
```
String
```

[47]:
```
# Array
typeof([1, 2, 3])
```

[47]:
```
Vector{Int64} (alias for Array{Int64, 1})
```

Variablen sind lediglich *bindings* – also quasi Platzhalter für Werte. Somit hat hat auch unser Array `x` einen Typ, dieser kann sich aber durch neue Belegung ändern:

[48]:
```
typeof(x)
```

[48]:
```
Vector{Any} (alias for Array{Any, 1})
```

[49]:
```
x = 1
typeof(x)
```

[49]:
```
Int64
```

1.7 Funktionen

1.7.1 Einführung und Motivation

Wir haben bisher schon einige Funktionen wie `println`, `typeof`, oder `push!` kennengelernt, die Julia uns ab Werk bereitstellt. Der Clou an der ganzen Sache ist nun, dass wir uns auch eigene Funktionen schreiben können. Folgende Funktion macht beispielsweise nichts anderes, als den Minus eines Inputs zurückzugeben:

[50]:
```
function minus(a)
    return -a
end

minus(10)
```

[50]:
```
-10
```

Dabei nennt man `a` ein (Funktions-)Argument und den Teil nach `return` einen Rückgabewert.

[51]:
```
# fyi
typeof(minus)
```

[51]:
```
typeof(minus) (singleton type of function minus, subtype of Function)
```

Warum wollen wir Funktionen schreiben? Erstens möchten wir Code nicht jedesmal neu schreiben, sondern wiederverwerten. Zweitens ist es auch guter Stil, weil es Code sehr viel übersichtlicher und leichter zu debuggen (fehlerbeheben) macht. Grund dafür ist unter anderem, dass Variablen, die wir innerhalb der Funktion neu anlegen, nur dort

sichtbar sind (local scope). Und drittens sorgt das im Falle von Julia oftmals auch für schnelleren Code.

Ein Beispiel für den ersten Punkt wäre beispielsweise die Mitternachtsformel. Da hätten wir keine Lust, für jedes Polynom jedesmal die gleiche Formel abzutippen.

[52]:
```
# löst ax² + bx + c = 0
function mitternacht(a, b, c)
    x₁ = -b + sqrt(b^2 - 4a*c)
    x₂ = -b - sqrt(b^2 - 4a*c)
    # Teilen durch 2a
    x₁ /= 2a
    x₂ /= 2a
    return x₁, x₂
end

# löst x² + x + 0 = 0
mitternacht(1, 1, 0)
```

[52]:
```
(0.0, -1.0)
```

Für Funktionen, die gut in einer Zeile geschrieben werden können, empfiehlt sich folgende Kurzschreibweise:

[53]:
```
immernoch_minus(a) = -a
immernoch_minus(-10)
```

[53]:
```
10
```

1.7.2 Verschiedene Methoden

Funktionen können verschiedene Methoden[1] (also unterschiedliche Funktionalität für verschiedene Inputs) haben.[2]

[54]:
```
# Unterschiedliches Verhalten je nach Anzahl der Inputs
f(a) = a^2
f(a, b) = a^2 + b

println(f(2))
println(f(2, 2))
```

[1] Nicht zu verwechseln mit dem Begriff Methode (Funktion innerhalb einer Klasse) aus der objektorientierten Programmierung.

[2] Für die Leute, die schon ein bisschen Erfahrung mit etwa C++ haben: Im Prinzip ist das wie das Überladen von Funktionen – mit dem subtilen Unterschied, dass bei C++ der Inputtyp schon zur Kompilierzeit bekannt sein muss. Dieses Thema (multiple dispatch) behandeln wir später noch.

```
4
6
```

[55]:
```
test(a) = 2 # allgemeines Verhalten
test(a::Integer) = 4 # Verhalten für Inputtyp Integer (type declaration)

println(test(1.0)); println(test(2))
```

```
2
4
```

Konvention: Funktionen, die nicht (nur) zurückgeben, sondern modifizieren, haben ein `!` am Ende.

1.7.3 Keyword arguments

[56]:
```
# b ist ein keyword argument
g(a; b = 1, c = "hallo") = a^2 + b
g(2)
```

[56]:
```
5
```

[57]:
```
multiply(a, b) = a * b
multiply(a::Complex, b) = println("kein Bock auf komplexe Zahlen :/")
multiply(1+2im, 1)
multiply("test", "test")
```

```
kein Bock auf komplexe Zahlen :/
```

[57]:
```
"testtest"
```

1.7.4 Broadcasting

Funktionen können punktweise angewendet werden, indem hinter den Funktionsnamen noch ein `.` ergänzt wird.

[58]:
```
exp.([0, 1, 2])
```

[58]:
```
3-element Vector{Float64}:
 1.0
 2.718281828459045
 7.38905609893065
```

Eigentlich ist dieser einzelne `.` nur eine Kurzschreibweise für ein allgemeineres Konzept, das man *broadcasting* nennt. Das hier wäre derselbe Befehl ohne Kurzschreibweise:

[59]:
```
broadcast(exp, [0, 1, 2])
```

[59]:
```
3-element Vector{Float64}:
 1.0
 2.718281828459045
 7.38905609893065
```

Der eigentliche Sinn erschließt sich uns aber erst bei Inputs ungleicher Länge.

[60]:
```
# 3x1 + 1x1
[0, 1, 2] .+ 1
```

[60]:
```
3-element Vector{Int64}:
 1
 2
 3
```

[61]:
```
broadcast(+, 1, [0, 1, 2])
```

[61]:
```
3-element Vector{Int64}:
 1
 2
 3
```

Man kann sich `broadcast` ungefähr so vorstellen: für die verschiedenen Inputs wird gecheckt, ob man sie auf gemeinsame Dimensionen bringen kann, indem man Koordinaten mit Länge 1 verlängert. Derart bekommt man

[62]:
```
# 4x1 + 1x4
[1, 2, 3, 4] .+ [10 20 30 40]
```

[62]:
```
4×4 Matrix{Int64}:
 11  21  31  41
 12  22  32  42
 13  23  33  43
 14  24  34  44
```

[63]:
```
# 4x1 + 1x4
I(4) .+ [10 20 30 40]
```

[63]:
```
4×4 Matrix{Int64}:
 11  20  30  40
 10  21  30  40
 10  20  31  40
 10  20  30  41
```

1.7.5 Anonyme Funktionen

[64]:
```
f(x) = x^2 - 1
f.([1, 2, 3])
```

[64]:
```
3-element Vector{Int64}:
 0
 3
 8
```

[65]:
```
broadcast(x -> x^2 - 1, [1, 2, 3])
```

[65]:
```
3-element Vector{Int64}:
 0
 3
 8
```

1.7.6 Inputtypen

Wir müssen uns immer überlegen, ob Funktionsargumente einen passenden Typ haben. Beispielsweise macht es wenig Sinn, den Minus eines Textes zurückzugeben:

[66]:
```
# genauso wenig funktioniert minus("1")
minus("Nonsens-Text")
```

```
MethodError: no method matching -(::String)

Closest candidates are:
  -(::Pkg.Resolve.VersionWeight)
   @ Pkg ~/.julia/juliaup/julia-1.10.0+0.aarch64.apple.darwin14/share/julia/stdlib
  ↪v1.10/Pkg/src/Resolve/versionweights.jl:25
  -(::Pkg.Resolve.VersionWeight, ::Pkg.Resolve.VersionWeight)
   @ Pkg ~/.julia/juliaup/julia-1.10.0+0.aarch64.apple.darwin14/share/julia/stdlib
  ↪v1.10/Pkg/src/Resolve/versionweights.jl:19
  -(::Bool, ::Complex{Bool})
   @ Base complex.jl:307
  …


Stacktrace:
 [1] minus(a::String)
   @ Main ./In[50]:2
 [2] top-level scope
```

```
@ In[66]:2
```

1.8 Style Guide

Generell gilt beim Programmieren: Nur dokumentierter Code ist guter Code. Man hat nämlich herzlich wenig davon, wenn man nach einem halben Jahr seinen eigenen Code nicht mehr versteht – ganz zu schweigen von irgendwelchen Dritten, die sich dann damit rumschlagen müssen. Das Thema Dokumentation sprengt in seiner Fülle zwar unseren Rahmen, aber Kommentare im Code sind dabei so oder so obligatorisch! Daneben gibt es für jede Programmiersprache eigene good practises, wie man Code aufschreiben sollte.

1.8.1 Variablenbenennung

Wir folgen hier dem Julia Style Guide und empfehlen, sich auch für die Programmieraufgaben daran zu orientieren.

- Kleinschreibung (lower case) für Variablen; falls schwer lesbar: Trennung mit `_`
- Typen und Module werden mit Großbuchstaben begonnen und getrennt (upper camel case): `BeispielTyp`
- Funktionen und Makros in lower case
- Mutating functions (Funktionen, die den Input bearbeiten können) mit `!` am Ende

Ausführlichere Vorgaben und Beispiele liefert das Blue Style Guide. Ganz hilfreich sind hieraus noch die Empfehlungen

- Vermeide unnötige spaces in Klammern. Schreibe also `string(1, 2)` anstelle von `string( 1 , 2 )`
- Wann immer angemessen, umgebe Binäroperatoren mit space, wie etwa `1 == 2` oder `y = x + 1`.

Dann muss man sich nämlich nicht mit Hässlichkeiten wie `print( x +y )` herumschlagen.

1.8.2 Zeichenkodierung

Last but not least: In fast allen Programmiersprachen ist es ein absolutes No-Go in Zeichen zu codieren, die nicht ASCII-Characters sind (also etwa Umlaute), Julia ist hier allerdings eine schöne Ausnahme. In der Dokumentation steht dazu:

Variable names must begin with a letter (A-Z or a-z), underscore, or a subset of Unicode code points greater than 00A0; in particular, Unicode character categories Lu/Ll/Lt/Lm/Lo/Nl (letters), Sc/So (currency and other symbols), and a few other letter-like characters (e.g. a subset of the Sm math symbols) are allowed. Subsequent characters may also include ! and digits (0-9 and other characters in categories Nd/No), as well as other Unicode code points: diacritics and other modifying marks (categories

Mn/Mc/Me/Sk), some punctuation connectors (category Pc), primes, and a few other characters.

Wie immer gilt: Nicht alles was geht, ist auch per se eine gute Idee. Beispielsweise ist vermutlich folgende Variablenbenennung nicht unbedingt sinnvoll:

[67]:
```
😇 = 1
```

[67]:
```
1
```

Gerade im Kontext von Mathematik-naher Programmierung sind aber beispielsweise griechische Symbole extrem praktisch. Derartige Symbole können durch die Eingabe in LaTeX-Schreibweise (z. B. `\lambda`) und anschließendes Drücken von Tab eingefügt werden (ausführliche Liste hier).

[68]:
```
# so macht man es in den meisten Programmiersprachen
lambda = 2
# vs. in Julia
λ = 2
```

[68]:
```
2
```

Kurs 2

2.1 Logische Typen und Operatoren

Neben den Datentypen, die wir in der letzten Kurs kennen gelernt haben, sind auch sogenannte Booleans essentiell. Diese haben nur zwei mögliche Werte: `true` oder `false`.

[1]:
```
# foo, bar, foobar, baz usw. sind das englische Äquivalent zu bla, blub, dings usw.
foo = true
typeof(foo)
```

[1]:
```
Bool
```

[2]:
```
# Negation
!false
```

[2]:
```
true
```

Wenn wir Werte mit Vergleichsoperatoren vergleichen, dann ist der Rückgabetyp gerade `Bool`. Weil `=` ja schon für Variablenzuweisung belegt ist, testen wir mit `==` auf Gleichheit:

[3]:
```
# kleiner
println(3 < 5)
# größer
println(3 > 5)
# gleich
println(3 == 3)
# ungleich
println(3 != 5)
# kleiner gleich
println(3 <= 3)
```

```
true
false
true
true
true
```

Wir können Booleans aka logische Aussagen mittels logischem Und bzw. Oder verknüpfen:

[4]:
```
# logisches Oder
true_value = (3 < 5) | (4 > 5)
# logisches Und
false_value = (3 < 5) & (4 > 5)
println(true_value)
println(false_value)
```

```
true
false
```

Grundsätzlich muss man aber ein bisschen mit der Klammersetzung aufpassen, denn sowas wie `5 | 4` ist ebenfalls wohldefiniert in Form eines bitweisen Oder-Vergleichs!

[5]:
```
# in Binärdarstellung haben wir 101 | 100 = 101 (bitweise!) also gerade wieder 5
5 | 4
```

[5]: 5

Dementsprechend läuft dann das Beispiel von oben ohne Klammern ziemlich schief:

[6]:
```
# das gleiche wie 3 < 5 > 5
not_so_true_value = 3 < 5 | 4 > 5
```

[6]: false

2.1.1 Short-circuit evaluation

Bei einem Vergleich wie `a & b` werden zunächst `a` und `b` ausgewertet, bevor der Vergleich stattfindet. Alternativ gibt es daher noch den Operator bzw. Vergleich `a && b` (short-circuit evaluation). Hier wird von links nach rechts ausgewertet, das heißt: Wenn `a` schon falsch ist, dann wird `b` nicht mehr berechnet. Das wollen wir manchmal, denn - wir können so Rechenzeit sparen - wenn `b` einen Fehler ausgibt, falls `a` schon nicht erfüllt ist, dann können wir das damit abfangen.

Letzteres sieht man im folgenden Beispiel:

[7]:
```
x = 1//2 # rationale Zahl 1/2
println(fieldnames(typeof(x))) # wir haben folgende Felder in x
denomis2(x) = (typeof(x) <: Rational) && (x.den == 2)
denomis2(x)
```

```
(:num, :den)
```

[7]: true

```
[8]: denomis2(0.5)
```

```
[8]: false
```

Analog gibt es zu `|` den `||`-Operator.

2.2 Operator-Hierarchie

Julia wendet die Operatoren in der folgenden Reihenfolge an (von der frühesten Auswertung zur spätesten):

Kategorien	Operatoren	Assoziativität
Syntax	`.` followed by `::`	Left
Exponentiation	`^`	Right
Unary	`+ - √`	Right
Bitshifts	`<< >> >>>`	Left
Fractions	`//`	Left
Multiplication	`* / % & \ ÷`	Left
Addition	`+ - | ⊻`	Left
Syntax	`: ..`	Left
Syntax	`|>`	Left
Syntax	`<|`	Right
Comparisons	`> < >= <= == === != !== <:`	Non-associative
Control flow	`&&` followed by `||` followed by `?`	Right
Pair	`=>`	Right
Assignments	`= += -= *= /= //= \= ^= ÷= %= |= &= ⊻= <<= >>= >>>=`	Right

Eine ausführlichere Darstellung findet sich hier.

2.3 Kontrollstrukturen

2.3.1 Conditional Evaluation

Kommen wir nun zu einer zentralen Anwendung logischer Operatoren: Mit diesen können wir Bedingungen im Code formulieren, denen zufolge bestimmte Codeabschnitte dann ausgewertet oder eben nicht ausgewertet werden.

Beispiel: Definition einer Betragsfunktion $|x|$:

[9]:
```
x = randn() # standardnormalverteilte Zufallsvariable

function betrag(x)
    if x < 0 # falls
        println(-x)
    else # falls nicht
        println(x)
    end
end

println("x = ", x)
println("betrag(x) = ", betrag(x))
```

```
x = 2.267408903577545
2.267408903577545
betrag(x) = nothing
```

Beispiel: Vergleich zweier Zahlen x und y:

[10]:
```
function order(x, y)
    if x < y
        println(x ," ist kleiner als ", y)
    elseif x > y # nicht wundern, das heißt in jeder Sprache anders (ifelse, if␣
 ↪else, else if, elif...)
        println(x ," ist größer als ", y)
    else
        println(x ," ist gleich groß wie ", y)
    end
end

order(1, 1)
```

```
1 ist gleich groß wie 1
```

Ternärer Operator

Hier haben wir einfach nur wieder eine verbereitete Kurzschreibweise – diesmal für `if`/`else` durch den Operator `?:`.

[11]:
```
betrag_neu(x) = x >= 0 ? x : -x
betrag_neu(-2)
```

[11]: 2

Weil der Operator `?:` drei Inputs hat, heißt er auch ternärer Operator.

2.3.2 Compound Expressions

Wenn man mehrere Zeilen Code zusammenfassen möchte, geht das mit `begin`-Blöcken oder Semikolons (`;`).

[12]:
```
z = begin
    x = 1
    y = 2
    x + y
end
```

[12]: 3

[13]:
```
z = (x = 1; y = 2; x + y)
```

[13]: 3

Auf diese Weise können wir zum Beispiel eine Funktion mit mehreren Anweisungen in eine Zeile quetschen:

[14]:
```
trick() = (a = 1; b = 1; return a + b)
trick()
```

[14]: 2

Wichtige weitere Kontrollstrukturen gibt es unter anderem beim *exception handling*, also Umgang mit Fehlern (etwa via `try`/`catch`). Das aber nur am Rande; als nächstes lernen wir zuletzt noch Kontrollstrukturen namens Loops kennen.

Kurs 3

3.1 Exkurs: Map

Die Funktion `map` funktioniert ähnlich wie broadcasting.

[1]:
```
# Wie gewohnt
map(exp, [1, 2, 3])
```

[1]:
```
3-element Vector{Float64}:
  2.718281828459045
  7.38905609893065
 20.085536923187668
```

[2]:
```
exp.([1, 2, 3])
```

[2]:
```
3-element Vector{Float64}:
  2.718281828459045
  7.38905609893065
 20.085536923187668
```

Es gibt allerdings ein paar kleine Unterschiede:

[3]:
```
# Vergleiche mit broadcast
map(+, [1, 2, 3, 4], [1 2 3 4])
```

[3]:
```
4-element Vector{Int64}:
 2
 4
 6
 8
```

Das liegt daran, dass `map` die beiden Inputs nach Reißverschlussprinzip miteinander kombiniert, also etwa so:

[4]:
```
[1, 2, 3, 4] .+ [1 2 3 4]
```

```
[4]: 4×4 Matrix{Int64}:
      2  3  4  5
      3  4  5  6
      4  5  6  7
      5  6  7  8
```

```
[5]: zip([1, 2, 3, 4], [1 2 3 4])
```

```
[5]: zip([1, 2, 3, 4], [1 2 3 4])
```

Dementsprechend schert sich `map` in unserem Beispiel nicht um Dimensionen (Zeilen vs. Spalten) und hat auch kein Problem damit, wenn die Länge nicht übereinstimmt:

```
[6]: map(+, [1, 2, 3], [1 2 3 4])
```

```
[6]: 3-element Vector{Int64}:
      2
      4
      6
```

Tatsächlich ist map ein super allgemeines Konzept, das zum Beispiel auch beim highperformance computing auftaucht. Dort hat man nämlich üblicherweise sogenannte map-reduce patterns. Die Intuition hierbei ist: Man hat in der Praxis immer hochparallele Systeme (viele Rechenkerne oder GPU(s), TPU(s)) und muss daher eine Aufgabe in möglichst gleiche Teile zerlegen, die individuell ausgerechnet (map) und nachher wieder zusammengeführt (reduce) werden[1]. Ein Beispiel wäre etwa das maximale Element einer großen Matrix `A` zu finden. Dabei hätten wir vielleicht verschiedene Maschinen, die dann jeweils das Maximum einer Spalte ausrechnen und an unseren Hauptrechner (host) zurückgeben. Dieser braucht dann nur noch die Ergebnisse zu reducen, dann sind wir fertig:

```
[7]: mat = [1 2; 3 4] # Beispielmatrix
     results = map(x -> maximum(x), eachcol(mat)) # könnte beispielsweise auf␣
      ↪verschiedenen Rechenkernen stattfinden
     println(results)
     reduce(max, results) # wir brauchen hier max, weil das ein binärer Operator ist
```

```
[3, 4]
```

```
[7]: 4
```

Ein letzter Tipp: Wenn die „gemapte“ Funktion keinen Output hat, verwendet man `foreach`.

[1] Aus mathematischer Sicht braucht man für map-reduce Assoziativität (klar für das Maximum) und Existenz eines neutralen Elements (das wäre dafür gerade `-Inf`). Mehr Infos dazu hier.

3.2 Rekursion

Mit Rekursion meinen wir den Prozess, dass Funktionen sich *selbst* wieder aufrufen können:

[8]:
```
function pow(a, n) # a^n
    if n == 0 # wir wollen irgendwann in diesem case landen
        return 1
    elseif n < 0
        return(pow(a, n + 1) / a) # a^n = a^(n+1) / a
    else
        return(pow(a, n - 1) * a) # a^n = a^(n-1) * a
    end
end

pow(2, 4)
```

[8]: 16

3.3 `for`-Loops

Untenstehend ist ein ziemlich unpraktischer Weg, um sich die Zahlen 1-10 auszugeben.

[9]:
```
println(1)
println(2)
println(3)
println(4)
println(5)
println(6)
println(7)
println(8)
println(9)
```

```
1
2
3
4
5
6
7
8
9
```

Wir wollen nämlich repetitive Aufgaben durch sogenannte Loops (Schleifen) lösen.

[10]:
```
function print_numbers(first_number, last_number)
    for i in first_number:last_number # vllt ein satz zu diesem range Objekt
        println(i)
    end
end

print_numbers(1, 9)
```

```
1
2
3
4
5
6
7
8
9
```

Es kann nicht nur über ranges von Zahlen (wie `first_number:last_number`) iteriert werden sondern auch über beliebige Elemente in einem Array:

[11]:
```
array1 = ["H", "A", "L", "L", "O", 1, 2, 3]

for element in array1
    print(element) # Ausgabe ohne neue Zeile am Ende
end
```

```
HALLO123
```

Auch bei der Erstellung von Arrays tauchen Loops auf:

[12]:
```
array2 = [2 * i^2 for i in 1:10]
```

[12]:
```
10-element Vector{Int64}:
   2
   8
  18
  32
  50
  72
  98
 128
 162
 200
```

Wenn wir dagegen umgekehrt über die Indizes eines Arrays loopen wollen, dann geht

das zwar durchaus mit `1:length(array2)`, man sollte tendenziell aber eher den Befehl `eachindex` nutzen:

[13]:
```
for i in eachindex(array2)
    println(i)
end
```

```
1
2
3
4
5
6
7
8
9
10
```

Ähnlich funktioniert `axes` für höherdimensionale Objekte:

[14]:
```
for j in axes(mat, 2) # loope über Spalten
    println(mat[1, j]) # fixiere außerdem erste Zeile
end
```

```
1
2
```

3.4 `while`-Loops

Hier können wir nun solange (`while`) einen Befehl ausführen, wie eine bestimmte Bedingung erfüllt ist:

[15]:
```
function countdown_from_n(n)
    while n > 0
        println(n)
        n -= 1
    end
end

countdown_from_n(5)
```

```
5
4
3
```

```
2
1
```

Anders als in vielen anderen Sprachen (insbesondere Interpretersprachen wie Python oder R) müssen wir uns in Julia meist nicht davor fürchten, dass selbstgeschriebene Loops inperformant werden. Denn an irgendeiner Stelle muss ja der Loop implementiert sein (außer man schreibt in Assembler), in Python etwa meistens in C/C++. Das bedeutet: Schneller, spezialisierter Code aus C/C++ (etwa NumPy) wird für, sagen wir, Matrizenrechnung eingebunden und ersetzt Python-Loops. In Julia entfällt dieser Schritt, weil sowieso das Meiste direkt (nativ) in Julia geschrieben wird bzw. weil Julia schon sehr performant ist. Eine Ausnahme wären etwa hochoptimierte Bibliotheken für Lineare Algebra (aber die werden sowieso in jeder Sprache extern eingebunden).

Zusammengefasst bietet uns sogenannte Vektorisierung wie in R oder Matlab per se keine Vorteile (das wäre gerade sowas wie broadcasting oder map in Julia). Nichtsdestotrotz steigt der Rechenaufwand verschachtelter Loops natürlich exponentiell und es gibt einige Tricks, die man zum Beschleunigen nutzen kann – dazu später mehr.

Kurs 4

4.1 Multiple dispatch

Wie vermutlich schon so ein bisschen angeklungen ist, unterscheidet sich Julia sowohl von statisch-typisierten Programmiersprachen wie C++, als auch von klassischen dynamisch-typisierten Programmiersprachen wie Python.

Bei C++ muss zur Kompilierzeit bekannt sein, was für ein Typ in eine bestimme Funktion gesteckt wird. Dieser Code wird dann in Maschinencode übersetzt. Bei Python muss der Typ einer Variablen wiederum erst zur Laufzeit bekannt sein, allerdings findet auch keine Übersetzung in Maschinencode statt. Julia ist quasi eine Hybrid-Lösung: Sobald eine Zeile Code ausgeführt wird, wird für den *konkret hineingesteckten Typ* ein Kompiliervorgang gestartet und der kompilierte Code ausgeführt. Diesen Vorgang nennt man *multiple dispatch*, wozu Julia einen *just-in-time compiler* (JIT) nutzt. Dadurch ist Julia faktisch dynamisch-typisiert, kann aber bei richtiger Verwendung quasi genauso schnell wie C/C++ sein.

4.1.1 Beispiel

Wir vergleichen, was nach dem Kompilierprozess rauskommt, wenn man einmal `Integer` und einmal `String` in dieselbe Funktion wirft:

[1]:
```
# Beispielfunktion für unterschiedliche Input-Typen
f(a, b) = a * b
# Wir nutzen ein Makro (lernen wir später kennen)
@code_llvm f(1, 2)
```

```
;  @ In[1]:2 within `f`
define i64 @julia_f_652(i64
signext %0, i64 signext %1)
#0 {
top:
; ┌ @ int.jl:88 within `*`
   %2 = mul i64 %1, %0
; └
  ret i64 %2
```

```
}
```

[2]:
```
@code_llvm f("1", "2")
```

```
;  @ In[1]:2 within `f`
define nonnull {}*
@julia_f_710({}* noundef
nonnull %0, {}* noundef
nonnull %1) #0 {
top:
  %2 = alloca [2 x
{}*], align 8
  %.sub = getelementptr inbounds
[2 x {}*],
[2 x {}*]*
%2, i64 0, i64 0
; ┌ @ strings/basic.jl:260 within `*`
; │┌ @ strings/substring.jl:225 within `string`
    store {}* %0,
{}** %.sub, align 8
    %3 = getelementptr inbounds
[2 x {}*],
[2 x {}*]*
%2, i64 0, i64 1
    store {}* %1,
{}** %3, align 8
    %4 = call nonnull
{}*
@j1__string_712({}*
inttoptr (i64 4766089072 to
{}*), {}**
nonnull %.sub, i32 2)
; └└
  ret {}* %4
}
```

Dass hier zwei verschiedene Outputs herauskommen, ist im Wesentlichen der Grund dafür, dass Julia performant (schnell) ist. Allerdings kann genau dieses Verhalten manchmal auch für langsamen Code führen – nämlich genau dann, wenn der Output Kompilierprozesses nicht spezialisiert genug ist (mehr dazu später).

4.2 Pakete

Für die kommenden Anwendungen werden wir sogenannte Pakete (packages) nutzen, die uns zusätzliche Funktionalität liefern. Wenn wir ein Paket namens `ExamplePackage` installieren wollen, dann geht das mittels `using Pkg` und `Pkg.add(ËxamplePackage")`.[1]

In der Julia-Konsole ist das durch den Paketmanager, welchen man durch die Eingabe von `]` erreicht, sogar noch einfacher (`add ExamplePackage`). Ein installiertes Paket lässt sich dann mit `using ExamplePackage` in Julia nutzen.

4.3 Pipes

Durch Pipes bekommen wir eine hübschere Syntax:

[3]:
```
# Beispielrechnung mit verschachtelten Funktionen
foo = 0; add6(x) = x + 6; mul2(x) = 2*x
println(mul2(add6(foo)))
```

```
12
```

[4]:
```
# Base Julia hat schon Pipes
foo |> add6 |> mul2 |> println
```

```
12
```

Das Paket Pipe.jl bietet uns noch ein paar zusätzliche Möglichkeiten:

[5]:
```
# Die Ausgabe dieses Befehls ist beim ersten Ausführen länger
using Pkg; Pkg.add("Pipe")
```

```
  Resolving package versions…
 No Changes to `~/.julia/environments/v1.10/Project.toml`
 No Changes to `~/.julia/environments/v1.10/Manifest.toml`
```

[6]:
```
# lade Paket
using Pipe

# Beispielfunktion mit zwei Argumenten
polynom(x, y) = x^2 + y + 5

# benutze den @pipe decorator
@pipe foo |> polynom(_, 2)
```

[1] Tatsächlich laden wir hier mit `using` ein Modul bzw. namespace (wir wissen noch nicht was das ist), generell sind Pakete hauptsächlich Modulsammlungen.

[6]:
```
7
```

4.4 DataFrames

Für unser nächstes Thema brauchen wir das Paket DataFrames.jl.

[7]:
```
# Die Ausgabe dieses Befehls ist beim ersten Ausführen länger
Pkg.add("DataFrames")
```

```
  Resolving package versions…
 No Changes to `~/.julia/environments/v1.10/Project.toml`
 No Changes to `~/.julia/environments/v1.10/Manifest.toml`
```

[8]:
```
using DataFrames
```

DataFrames sind im Grunde genommen Tabellen, wie zum Beispiel in Excel. Wer schon `dplyr` oder `pandas` kennt, findet hier einen guten Vergleich zu `DataFrames.jl`.

In unserem ersten Beispiel schauen wir uns einen Datensatz mit vergebenen Noten an:

[9]:
```
function grades_2020()
    name = ["Sally", "Bob", "Alice", "Hank"]
    grade_2020 = [1, 5, 8.5, 4]
    DataFrame(; name, grade_2020)
end
df = grades_2020()
```

[9]:

	name	grade_2020
	String	Float64
1	Sally	1.0
2	Bob	5.0
3	Alice	8.5
4	Hank	4.0

Auf Spalten kann direkt über ihren Namen zugegriffen werden:

[10]:
```
df.name
```

```
4-element Vector{String}:
 "Sally"
 "Bob"
 "Alice"
 "Hank"
```

Alternativ würde auch die Syntax `df[!, :name]` funktionieren. Hierbei ist wichtig zu verstehen, dass dies keine Kopie der Spalte erzeugt, sondern Änderungen sich direkt in

den DataFrame übertragen.

[11]:
```
df.name[1] = "Sharon"
df
```

[11]:

	name	grade_2020
	String	Float64
1	Sharon	1.0
2	Bob	5.0
3	Alice	8.5
4	Hank	4.0

`df[:, :name_der_spalte]` oder `df[:, nummer_der_spalte]` hingegen erzeugen Kopien der ausgewählten Spalten des DataFrames (das ist also anders als bei Arrays):

[12]:
```
names = df[!, :name]
names[1] = "Alice" # oder df[:,1]
```

[12]:
```
"Alice"
```

[13]:
```
names[1] = "Alice"
df
```

[13]:

	name	grade_2020
	String	Float64
1	Alice	1.0
2	Bob	5.0
3	Alice	8.5
4	Hank	4.0

filter

Die Funktion `filter` hilft Zeilen eines DataFrames nach beliebigen Kriterien auszuwählen, als Beispiel hierfür definieren wir uns zunächst folgende Funktion:

[14]:
```
equals_alice(name::String) = name == "Alice"
equals_alice("Bob")
```

[14]:
```
false
```

Mithilfe von `filter` können wir jetzt alle Einträge der Spalte `name` in dem Dataframe durchgehen und die Reihen mit dem Namen `Älice"` filtern.

[15]:
```
filter(:name => equals_alice, df)
```

[15]:

	name	grade_2020
	String	Float64
1	Alice	1.0
2	Alice	8.5

Alternativ hätten wir uns auch die Hilfsfunktion `equals_alice` sparen und stattdessen eine anonyme Funktion nutzen können:

[16]:
```
filter(:name => x -> x == "Bob", df)
```

[16]:

	name	grade_2020
	String	Float64
1	Bob	5.0

Noch kürzer geht es mithilfe der generischen Funktion `==("Bob")`:

[17]:
```
filter(:name => ==("Bob"), df)
```

[17]:

	name	grade_2020
	String	Float64
1	Bob	5.0

`subset` und `select`

Statt `filter` können (und wollen wir meistens) auch die Funktion `subset` nutzen. Aus zwei Gründen sollte man aber `filter` trotzdem schon mal gesehen haben:

- `filter` funktioniert auch für andere Typen als DataFrames (zum Beispiel Dicts)
- zum Teil ist `filter` performanter

Umgekehrt ist `subset` aber zumeist ein bisschen angenehmer:

- besseres handling von fehlenden Werten
- Syntax ist konsistent mit den anderen Befehlen für DataFrames.

Zentral ist jedenfalls, dass `subset` nicht Werte einzelner Reihen vergleicht, sondern stets die komplette Spalte vergleicht:

[18]:
```
# vergleiche mit filter, unsere anonyme Funktion operiert nun auf einen Vektor!
subset(df, :name => x -> x .==("Bob"))
```

[18]:

	name	grade_2020
	String	Float64
1	Bob	5.0

Wenn wir unser Problem wie gehabt nicht in einen Vergleich der ganzen Spalte umformulieren wollen, dann geht das mithilfe der Funktion `ByRow`:

[19]:
```
subset(df, :name => ByRow(==("Bob")))
```

[19]:

	name	grade_2020
	String	Float64
1	Bob	5.0

Die Funktion `select` filtert im Gegensatz zu `filter`/`subset` nicht Reihen, sondern Spalten aus.

[20]:
```
function responses()
    id = [1, 2]
    q1 = [28, 61]
    q2 = [:us, :fr]
    q3 = ["F", "B"]
    q4 = ["B", "C"]
    q5 = ["A", "E"]
    DataFrame(; id, q1, q2, q3, q4, q5)
end
df2 = responses()
```

[20]:

	id	q1	q2	q3	q4	q5
	Int64	Int64	Symbol	String	String	String
1	1	28	us	F	B	A
2	2	61	fr	B	C	E

Beispielsweise können wir so die Spalten `id` und `q1` auswählen:

[21]:
```
select(df2, :id, :q1) # alternativ würde auch select(df2, "id", "q1") funktionieren
```

[21]:

	id	q1
	Int64	Int64
1	1	28
2	2	61

Mithilfe der Funktion `Not` kann man eine oder mehrere Spalten aussortieren:

[22]:
```
select(df2, Not([:q2, :q5]))
```

[22]:

	id	q1	q3	q4
	Int64	Int64	String	String
1	1	28	F	B
2	2	61	B	C

Die Position von Spalten kann derart verändert werden:

[23]:
```
select(df2, :q3, :) # erst q3, dann der Rest
```

[23]:

	q3	id	q1	q2	q4	q5
	String	Int64	Int64	Symbol	String	String
1	F	1	28	us	B	A
2	B	2	61	fr	C	E

[24]:
```
select(df2, 2, :q3, :) # erst die zweite Spalte (:q1), dann :q3, dann der Rest
```

[24]:

	q1	q3	id	q2	q4	q5
	Int64	String	Int64	Symbol	String	String
1	28	F	1	us	B	A
2	61	B	2	fr	C	E

Und zuletzt können mithilfe von `select` auch die Namen der Spalten geändert werden:

[25]:
```
select(df2, 1 => "participant", :q1 => "age", :q2 => "nationality")
df2
```

[25]:

	id	q1	q2	q3	q4	q5
	Int64	Int64	Symbol	String	String	String
1	1	28	us	F	B	A
2	2	61	fr	B	C	E

Wie für Julia typisch, gibt es alle oben genanten Funktionen auch mit einem `!` (wie etwa `select!`). Der Unterschied besteht darin, dass in diesem Fall keine Kopie des Dataframes erstellt, sondern der orginale DataFrame verändert wird.

4.4.1 Datentypen und kategorische Variablen

Wie man in den oberen Bespielen erkennen kann, versucht Julia jeder Spalte einen Datentyp zuzordnen, was allerdings nicht immer ganz so gut funktioniert.

[26]:
```
function wrong_types()
    id = 1:4
    date = ["28-01-2018", "03-04-2019", "01-08-2018", "22-11-2020"]
    age = ["adolescent", "adult", "infant", "adult"]
    DataFrame(; id, date, age)
end
df = wrong_types()
```

[26]:

	id	date	age
	Int64	String	String
1	1	28-01-2018	adolescent
2	2	03-04-2019	adult
3	3	01-08-2018	infant
4	4	22-11-2020	adult

Falsche Datentypen können das Sortieren in DataFrames erschweren. In dem oberen Fall hat zum Beispiel die Spalte `date` das Format `String`, obwohl es in Julia dafür einen speziellen Datentyp (nämlich `Date`) gibt, mithilfe dessen man Daten im Datumsformat einfach vergleichen kann. Mit dem Paket `Dates` lässt sich dies aber leicht beheben.

[27]:
```
Pkg.add("Dates")
```

```
  Resolving package versions…
 No Changes to `~/.julia/environments/v1.10/Project.toml`
 No Changes to `~/.julia/environments/v1.10/Manifest.toml`
```

[28]:
```
using Dates

function fix_date_format(df)
    dates = Date.(df.date, dateformat"dd-mm-yyyy") # specify date format
    df.date = dates # reassign dates
end

fix_date_format(df)
df
```

[28]:

	id	date	age
	Int64	Date	String
1	1	2018-01-28	adolescent
2	2	2019-04-03	adult
3	3	2018-08-01	infant
4	4	2020-11-22	adult

Zur Überprüfung vergleichen wir das Geburtsdatum der Personen 1 und 2 und erhalten:

[29]:
```
df[1, :date] < df[2, :date]
```

[29]:
```
true
```

In der Spalte `age` trifft man auf ein ähnliches Problem, da man den Variablen gerne eine hierarchische Struktur geben würde: `adult > adolescent > infant`. Dies lässt sich mithilfe des Paketes `CategoricalArrays` beheben:

[30]:
```
Pkg.add("CategoricalArrays")
```

```
  Resolving package versions…
 No Changes to `~/.julia/environments/v1.10/Project.toml`
 No Changes to `~/.julia/environments/v1.10/Manifest.toml`
```

[31]:
```
using CategoricalArrays

function fix_age(df)
    levels = ["infant", "adolescent", "adult"]
    ages = categorical(df.age; levels, ordered = true)
    df.age = ages
end
```

```
fix_age(df)
df
```

[31]:

	id	date	age
	Int64	Date	Cat…
1	1	2018-01-28	adolescent
2	2	2019-04-03	adult
3	3	2018-08-01	infant
4	4	2020-11-22	adult

Auch hier kann man noch einmal überprüfen ob sich die Personen anhand ihres Alter vergleichen lassen können :

[32]:
```
df[1, :age] < df[2, :age]
```

[32]: true

4.4.2 Datensätze zusammenführen

Im folgenden Abschnitt wird behandelt, wie man verschiedene DataFrames zusammenführen bzw. kombinieren kann. Hierfür nimmt man meistens verschiedene Versionen der Funktion `join`; die jeweilige Funktionalität ist im Cheatsheet von Tom Kwong schön visualisiert.

Betrachten wir zwei DataFrames:

[33]:
```
function grades_2021() # grades_2020() haben wir schon
    name = ["Kevin", "Sally", "Hank"]
    grade_2021 = [8, 7, 5.5]
    DataFrame(; name, grade_2021)
end
grades_2021()
```

[33]:

	name	grade_2021
	String	Float64
1	Kevin	8.0
2	Sally	7.0
3	Hank	5.5

[34]:
```
grades_2020()
```

[34]:

	name	grade_2020
	String	Float64
1	Sally	1.0
2	Bob	5.0
3	Alice	8.5
4	Hank	4.0

Bei `innerjoin` wird ein Argument/Spaltenname mitgegeben, über welchen die beiden DataFrames zusammen geführt werden sollen. So werden beispielsweise für `:name` alle Elemente aus der Spalte `:name` des einen DataFrames mit den Elementen des anderen DataFrames verglichen. Falls diese Einträge übereinstimmen, werden die restliche Information (Einträge der Reihe) aus beiden DataFrames in Spalten zusammengeführt.

[35]:
```
innerjoin(grades_2020(), grades_2021(), on=:name)
```

[35]:

	name	grade_2020	grade_2021
	String	Float64	Float64
1	Sally	1.0	7.0
2	Hank	4.0	5.5

`outerjoin` nimmt hingegen alle Elemente, die in den jeweiligen DataFrames zumindest einmal vorkommen und führt diese zusammen. Hierbei werden nicht vorhandene Information durch ein "missing" ersetzt.

[36]:
```
outerjoin(grades_2020(), grades_2021(); on=:name)
```

[36]:

	name	grade_2020	grade_2021
	String	Float64?	Float64?
1	Sally	1.0	7.0
2	Hank	4.0	5.5
3	Bob	5.0	missing
4	Alice	8.5	missing
5	Kevin	missing	8.0

Zuletzt übergibt `leftjoin/rightjoin` alle Werte aus dem linken/rechten Dataframe und führt sie mit den Einträgen aus dem rechten/linken DataFrame zusammen, für welche der rechte/linke Dataframe in der entsprechenden Kategorie (zum Beispiel `:name`) übereinstimmt:

[37]:
```
leftjoin(grades_2020(), grades_2021(); on=:name)
```

[37]:

	name	grade_2020	grade_2021
	String	Float64	Float64?
1	Sally	1.0	7.0
2	Hank	4.0	5.5
3	Bob	5.0	missing
4	Alice	8.5	missing

[ ]:
```

```

Kurs 5

Wir setzen mit den bereits bekannten DataFrames `grades_2020` und `grades_2021` fort:

[1]:
```
using DataFrames, Pipe

function grades_2020()
    name = ["Sally", "Bob", "Alice", "Hank"]
    grade_2020 = [1, 5, 8.5, 4]
    DataFrame(; name, grade_2020)
end

function grades_2021() # grades_2020() haben wir schon
    name = ["Kevin", "Sally", "Hank"]
    grade_2021 = [8, 7, 5.5]
    DataFrame(; name, grade_2021)
end
```

[1]:
```
grades_2021 (generic function with 1 method)
```

5.0.1 Variablen transformieren und anpassen

In diesem Beispiel wird der Datensatz `grades_2020` via `plus_1` transformiert:

[2]:
```
plus_1(grades) = grades .+ 1
transform(grades_2020(), :grade_2020 => plus_1)
```

[2]:

	name	grade_2020	grade_2020_plus_1
	String	Float64	Float64
1	Sally	1.0	2.0
2	Bob	5.0	6.0
3	Alice	8.5	9.5
4	Hank	4.0	5.0

Wird keine target location übergeben, wird automatisch eine neue Spalte erstellt. Falls die alte Spalte überschrieben werden soll, kann man diese als target location auswählen:

[3]:
```
transform(grades_2020(), :grade_2020 => plus_1 => :grade_2020)
```

[3]:

	name	grade_2020
	String	Float64
1	Sally	2.0
2	Bob	6.0
3	Alice	9.5
4	Hank	5.0

Alternativ kann man das Argument `renamecols = false` setzen:

[4]:
```
transform(grades_2020(), :grade_2020 =>  plus_1; renamecols = false)
```

[4]:

	name	grade_2020
	String	Float64
1	Sally	2.0
2	Bob	6.0
3	Alice	9.5
4	Hank	5.0

Groupby and Combine

Mit Hilfe dieser beider Befehle können Dataframes aufgeteilt und im Anschluss auch wieder zusammengeführt werden.

[5]:
```
function all_grades()
    df1 = grades_2020()
    df1 = select(df1, :name, :grade_2020 => :grade)
    df2 = grades_2021()
    df2 = select(df2, :name, :grade_2021 => :grade)
    return vcat(df1, df2)
end
all_grades()
```

[5]:

	name	grade
	String	Float64
1	Sally	1.0
2	Bob	5.0
3	Alice	8.5
4	Hank	4.0
5	Kevin	8.0
6	Sally	7.0
7	Hank	5.5

Die Idee ist nun den obigen DataFrame aufzuteilen und für jeden Schüler den Mittelwert seiner Note zu berechen und anschließend einen DataFrame mit den Durchschnittsnoten auszugeben.

[6]:
```
grouped_grades = groupby(all_grades(), :name)
```

[6]: GroupedDataFrame with 5 groups based on key: name

First Group (2 rows): name = SSally”

	name	grade
	String	Float64
1	Sally	1.0
2	Sally	7.0

…

Last Group (1 row): name = ”Kevin”

	name	grade
	String	Float64
1	Kevin	8.0

[7]:
```
using Statistics
combine(grouped_grades, :grade => mean)
```

[7]:

	name	grade_mean
	String	Float64
1	Sally	4.0
2	Bob	5.0
3	Alice	8.5
4	Hank	4.75
5	Kevin	8.0

5.0.2 Fehlende Werte

In Julia gibt es einen Datentyp `missing` für fehlende Werte, der sich bei Rechenoperationen oder Vergleichsoperationen von anderen Datentypen unterscheidet:

[8]:
```
missing * 2
```

[8]: missing

[9]:
```
missing == true
```

[9]: missing

Bei einer realen Datenanalyse kommt es aber sehr häufig vor, dass Datenpunkte fehlen, daher müssen wir uns überlegen, wie wir damit sinnvoll umgehen. Betrachte zur Demonstration folgenden DataFrame:

[10]:
```
df_missing = DataFrame(
    name = [missing, "Sally", "Alice", "Hank"],
    age = [17, missing, 20, 19],
    grade_2020 = [5.0, 1.0, missing, 4.0],
)
```

[10]:

	name	age	grade_2020
	String?	Int64?	Float64?
1	missing	17	5.0
2	Sally	missing	1.0
3	Alice	20	missing
4	Hank	19	4.0

Wichtig ist zum Beispiel die Funktion `dropmissing`. Diese entfernt jede Reihe, in der ein `missing` value enthalten ist:

[11]:
```
dropmissing(df_missing)
```

[11]:

	name	age	grade_2020
	String	Int64	Float64
1	Hank	19	4.0

Es können auch nur in bestimmenten Spalten alle `missing` werte entfernt werden.

[12]:
```
dropmissing(df_missing, [:name, :age])
```

[12]:

	name	age	grade_2020
	String	Int64	Float64?
1	Alice	20	missing
2	Hank	19	4.0

`ismissing` ist eine Funktion, die überprüft, ob der zugrundeliegende Datentyp dem Typ `Missing` entspricht und gibt dann dementsprechen ein `true` oder `false` zurück. Damit können wir zum Beispiel anstelle der Funktion `dropmissing` wieder unsere altbekannte Funktion `filter` nutzen:

[13]:
```
filter(:name => !ismissing, df_missing)
```

[13]:

	name	age	grade_2020
	String?	Int64?	Float64?
1	Sally	missing	1.0
2	Alice	20	missing
3	Hank	19	4.0

[14]:
```
dropmissing(df_missing, [:name])
```

[14]:

	name	age	grade_2020
	String	Int64?	Float64?
1	Sally	missing	1.0
2	Alice	20	missing
3	Hank	19	4.0

Wenn wir hier aber wieder nach zwei Spalten filtern wollen, dann wird es relativ hässlich (aber bitte trotzdem einmal durchdenken!):

[15]:
```
filter([:name, :age] => (x, y) -> all((!).(ismissing.([x, y]))), df_missing)
```

[15]:

	name	age	grade_2020
	String?	Int64?	Float64?
1	Alice	20	missing
2	Hank	19	4.0

Um fehlende Werte zu überspringen um zum Beispiel die Funktion `mean` auf die Spalte `age` anwenden zu können, gibt es die Funktion `skipmissing`.

[16]:
```
mean_age = mean(skipmissing(df_missing.age))
```

[16]:
```
18.666666666666668
```

Oft will man für fehlende Werte durch andere Werte ersetzen, mit denen man besser arbeiten kann. Dies kann etwa mithilfe der Funktion `coalesce` erreicht werden:

[17]:
```
coalesce.([missing, "some value", missing], "zero")
```

[17]:
```
3-element Vector{String}:
 "zero"
 "some value"
 "zero"
```

Angenommen, wir wollen nun die Werte in der Spalte `age` einfach durch das Durchschnittsalter aller anderer Schüler approximieren:

[18]:
```
df_missing
```

[18]:

	name	age	grade_2020
	String?	Int64?	Float64?
1	missing	17	5.0
2	Sally	missing	1.0
3	Alice	20	missing
4	Hank	19	4.0

[19]:
```
transform(df_missing, :age => ByRow(x -> coalesce(x, mean_age)); renamecols = false)
```

[19]:

	name	age	grade_2020
	String?	Real	Float64?
1	missing	17	5.0
2	Sally	18.6667	1.0
3	Alice	20	missing
4	Hank	19	4.0

5.1 Plots

5.1.1 Einführung

Es gibt viele verschiedene plotting libraries in Julia; wir geben eine Einführung in die Standardbibliothek Plots.jl. Alternativ empfehlen wir für fortgeschrittene Anwender Makie.jl und die Einführung hier (hat ein etwas umfasserendes Ökosystem) oder Gadfly (ist sehr ähnlich zu ggplot2, sprich ist eine an The Grammar of Graphics orientierte Bibliothek, und sieht per default ziemlich gut aus).

Ein Cheatsheet findet man hier.

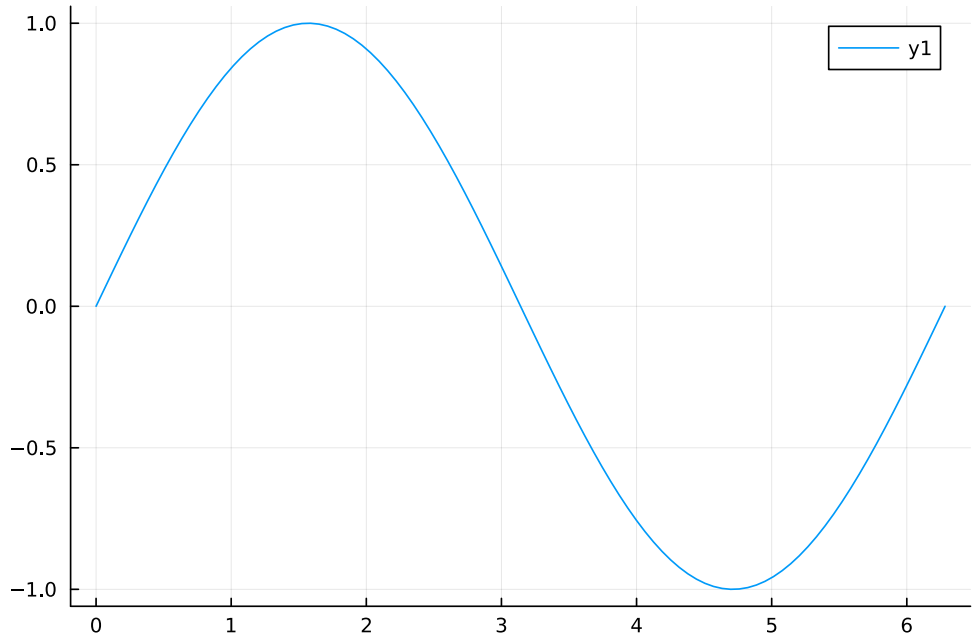
[20]:
```
# Die Ausgabe dieses Befehls ist beim ersten Ausführen länger
using Pkg; Pkg.add("Plots")
```

```
  Resolving package versions…
 No Changes to `~/.julia/environments/v1.10/Project.toml`
 No Changes to `~/.julia/environments/v1.10/Manifest.toml`
```

Wir starten mit dem einfachen Sinus-plot und erweitern diesen iterativ um verschiedene manuell konfigurierbare Attribute.

[21]:
```
using Plots
x = range(0, 2*pi, length=100)
y = sin.(x)
plot(x, y)
```

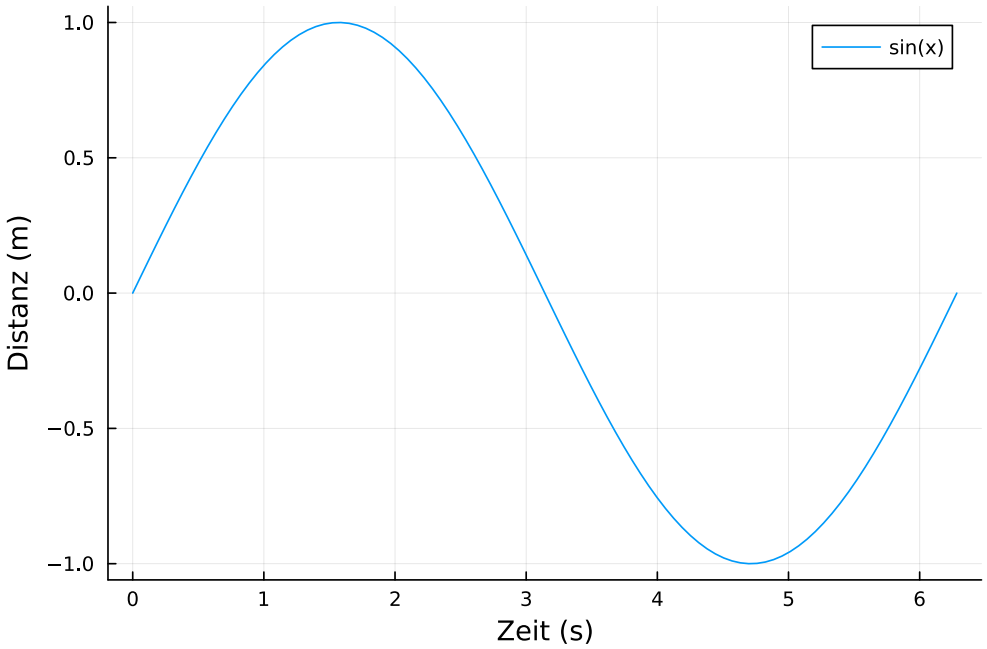
[21]:



Hier haben wir jetzt einen (völlig unspektakulären) Standardplot und man kann nun anfangen, diesem Attribute hinzuzufügen, wie zum Bespiel die Beschriftung der Achsen sowie der Kurve selbst:

[22]:
```
x = range(0, 2*pi, length=100)
y = sin.(x)
plot(x, y,
    xlabel = "Zeit (s)",
    ylabel = "Distanz (m)",
    label = "sin(x)"
)
```

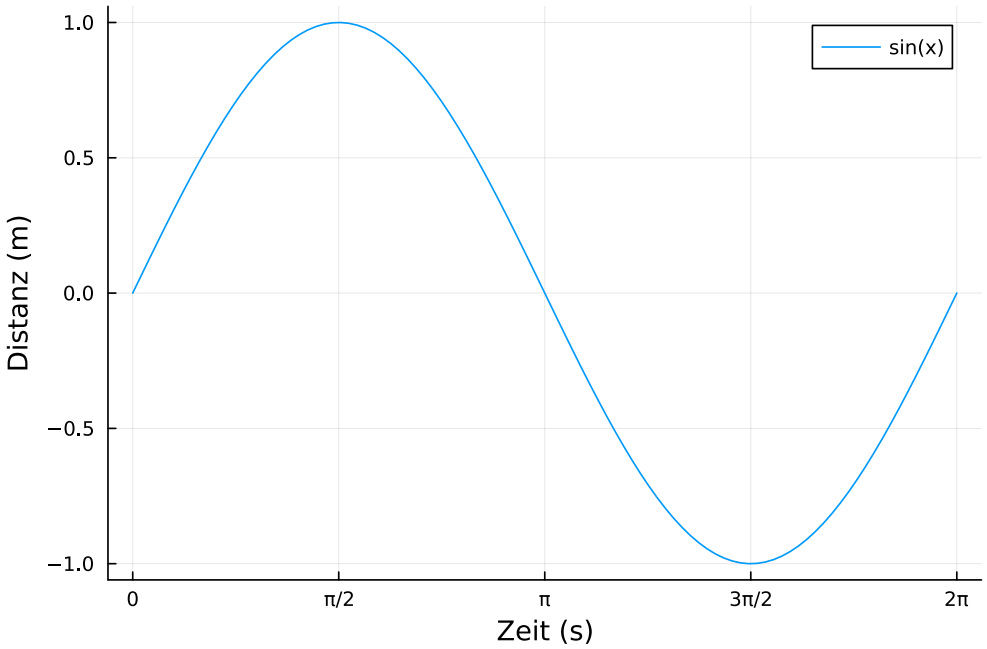
[22]:



Mit Hilfe von `xticks` bzw. `yticks` kann ein Tupel übergeben werden, welches im ersten Argument einen Array für die Achsenabschnitte und im zweiten Argument einen Array für deren Beschriftung enthält:

[23]:
```
plot(x, y,
    xlabel = "Zeit (s)",
    ylabel = "Distanz (m)",
    label = "sin(x)",
    xticks = ([0, π/2, π, 3π/2, 2π], ["0", "π/2", "π", "3π/2", "2π"])
)
```

[23]:



Um einen zweiten Plot zu dem bereits bestehenden hinzuzufügen, gibt es verschiedene Möglichkeiten: Falls der Plot im gleichen Fenster sein soll, kann man diesen durch `plot!` ganz einfach in dieselbe Grafik einzeichnen:

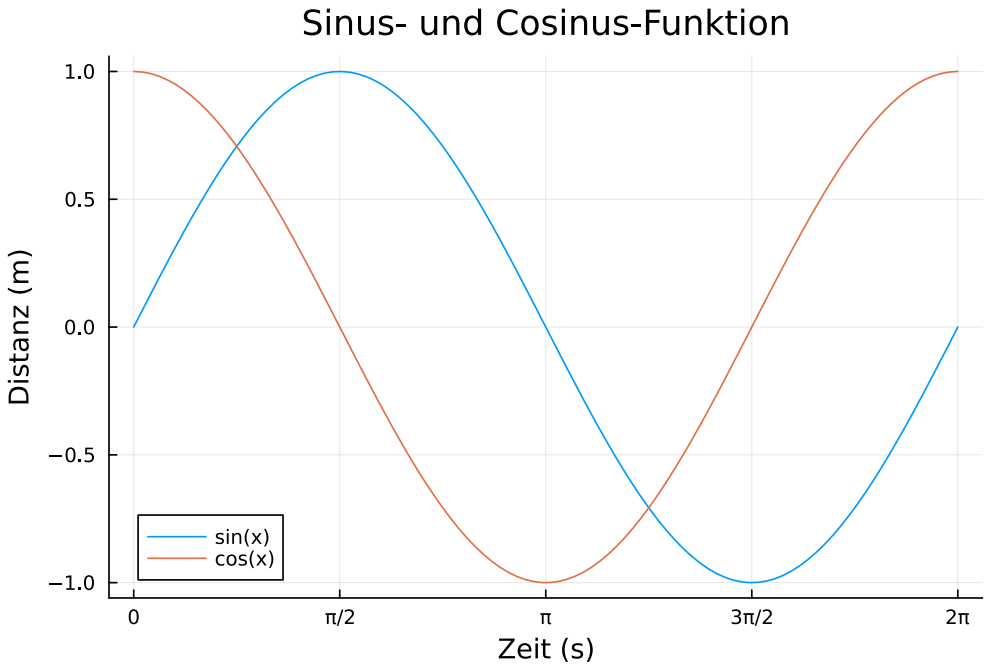
[24]:
```
plot(x, y1,
    xlabel = "Zeit (s)",
    ylabel = "Distanz (m)",
    label = "sin(x)",
    xticks = ([0, π/2, π, 3π/2, 2π], ["0", "π/2", "π", "3π/2", "2π"])
)
```

```
UndefVarError: `y1` not defined

Stacktrace:
 [1] top-level scope
   @ In[24]:1
```

[25]:
```
y1 = sin.(x)
y2 = cos.(x)
plot(x, y1,
    xlabel = "Zeit (s)",
```

```
    ylabel = "Distanz (m)",
    label = "sin(x)",
    xticks = ([0, π/2, π, 3π/2, 2π], ["0", "π/2", "π", "3π/2", "2π"])
)
plot!(x, y2,
    label = "cos(x)",
    title = "Sinus- und Cosinus-Funktion"
)
```

[25]:



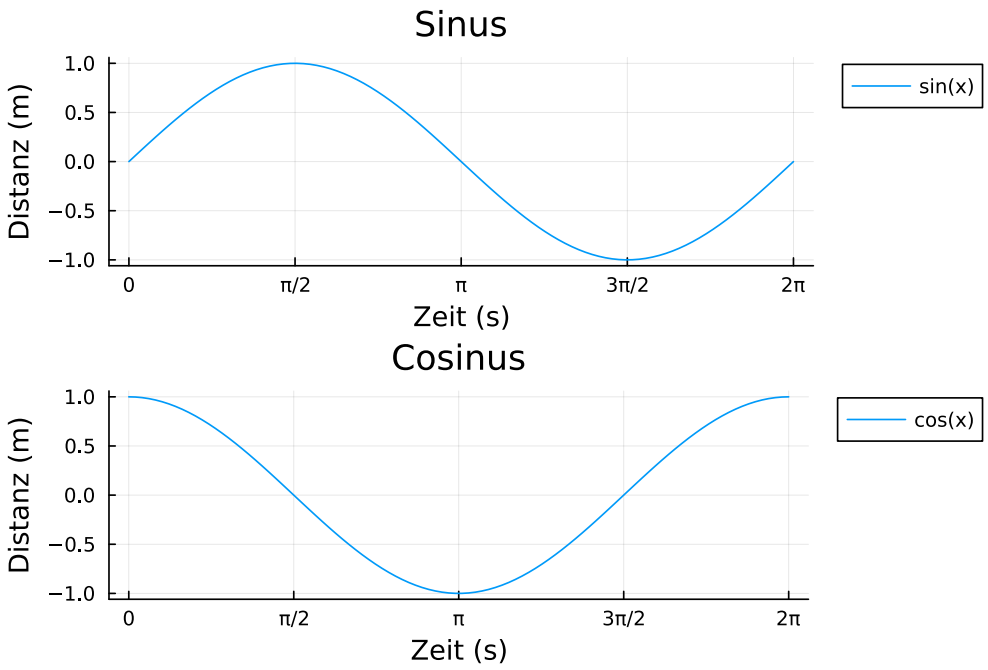
Falls man zwei verschiedene Fenster für die jeweiligen Plots haben will, kann man zwei seperate plots erstellen und diese mit einem dritten Plotbefehl sowie einer Positionsangabe neben- oder untereinander plotten:

[26]:

```
y1 = sin.(x)
y2 = cos.(x)

p1 = plot(x, y1,
    xlabel = "Zeit (s)",
    ylabel = "Distanz (m)",
    label = "sin(x)",
    xticks = ([0, π/2, π, 3π/2, 2π], ["0", "π/2", "π", "3π/2", "2π"]),
    title = "Sinus"
```

```
)
p2 = plot(x, y2,
    xlabel = "Zeit (s)",
    ylabel = "Distanz (m)",
    label = "cos(x)",
    xticks = ([0, π/2, π, 3π/2, 2π], ["0", "π/2", "π", "3π/2", "2π"]),
    title = "Cosinus"
)

plot(p1, p2, layout=(2, 1),legend=:outertopright)
```

[26]:



5.1.2 Scatterplots

Ein weiteres Bespiel aus Plots.jl sind scatter plots. Hierfür laden wir uns zunächst einen klassischen R-Datensatz in Julia und stellen diesen dann Stück für Stück grafisch dar.

[27]:

```
# Die Ausgabe dieses Befehls ist beim ersten Ausführen länger
Pkg.add("RDatasets");
```

```
  Resolving package versions…
No Changes to `~/.julia/environments/v1.10/Project.toml`
No Changes to `~/.julia/environments/v1.10/Manifest.toml`
```

[28]:
```
using DataFrames
using RDatasets

# ein Klassiker
iris = dataset("datasets", "iris")
first(iris,10)
```

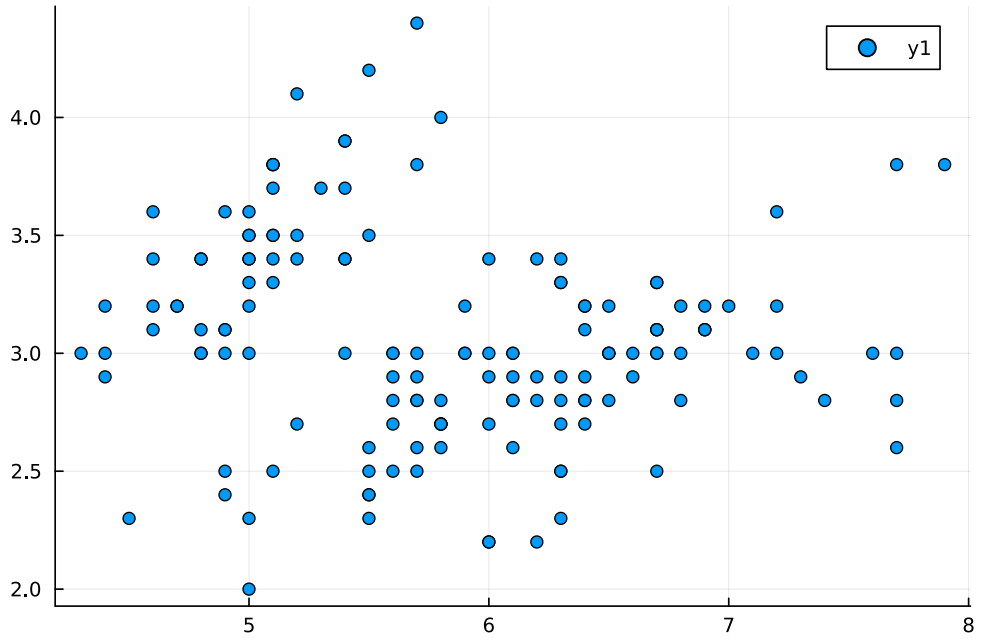
[28]:

	SepalLength	SepalWidth	PetalLength	PetalWidth	Species
	Float64	Float64	Float64	Float64	Cat…
1	5.1	3.5	1.4	0.2	setosa
2	4.9	3.0	1.4	0.2	setosa
3	4.7	3.2	1.3	0.2	setosa
4	4.6	3.1	1.5	0.2	setosa
5	5.0	3.6	1.4	0.2	setosa
6	5.4	3.9	1.7	0.4	setosa
7	4.6	3.4	1.4	0.3	setosa
8	5.0	3.4	1.5	0.2	setosa
9	4.4	2.9	1.4	0.2	setosa
10	4.9	3.1	1.5	0.1	setosa

Wir erstellen einen ersten Scatterplot über die beiden Spalten `PetalLength` und `SepalWidth`:

[29]:
```
scatter(iris.SepalLength, iris.SepalWidth)
```

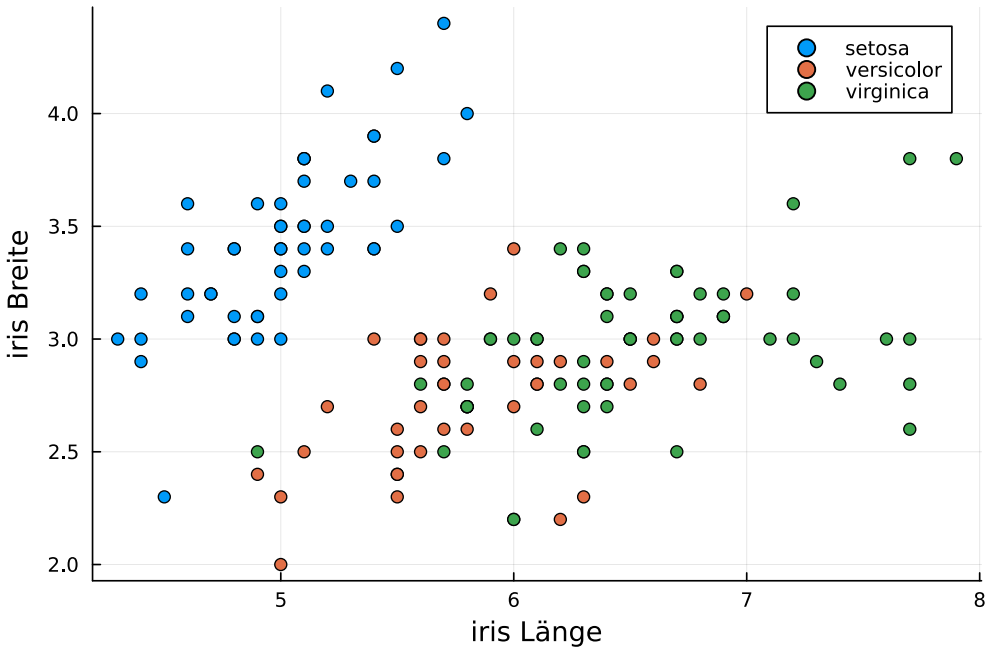
[29]:



Alternativ könnten wir auch den normalen `plot`-Befehl in Verbindung mit `seriestype = :scatter` nutzen. Auch hier könnten analog zum obigen Bespiel erstmal die Achsen beschriftet werden. Stattdessen sollen hier aber die Punkte anhand eines gewissen Attributs gruppiert werden. Hierfür nehmen wir uns die Spalte `Species`.

[30]:
```
scatter(iris.SepalLength, iris.SepalWidth,
    xlabel = "iris Länge",
    ylabel = "iris Breite",
    group = iris.Species
)
```

[30]:

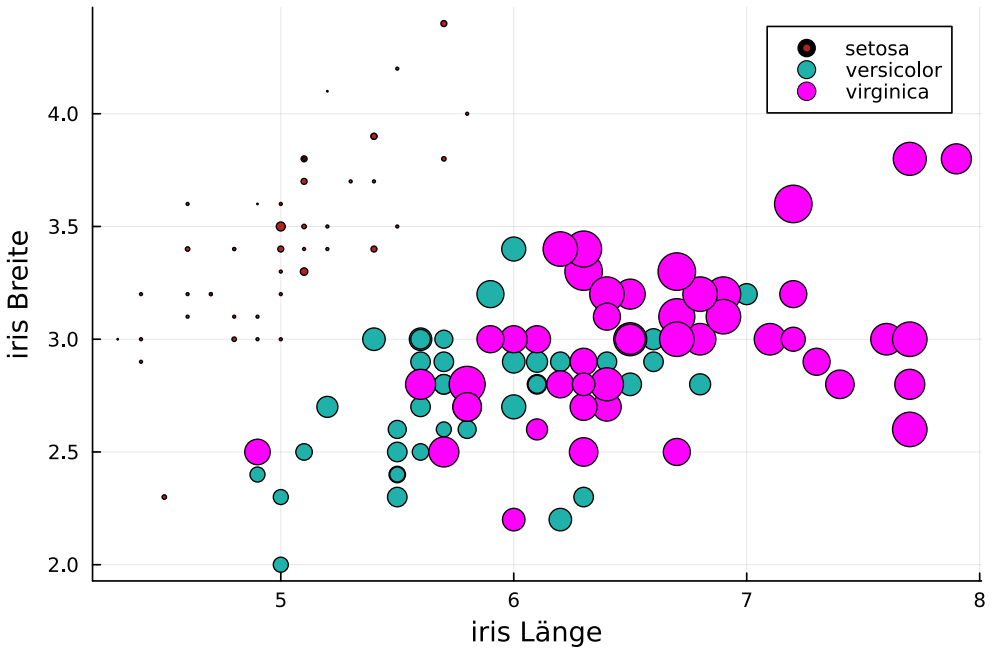


Die Farbe und Größe der Datenpunkte, orientiert anhand einer bestimmten Kategorie des Datensatzes, können ebenfalls leicht verändert/hinzugefügt werden, um weitere Attribute des Datensatzes in den Plot zu integrieren.

[31]:
```
scatter(iris.SepalLength, iris.SepalWidth,
    xlabel = "iris Länge",
    ylabel = "iris Breite",
    group = iris.Species,
    markercolor = [:firebrick :lightseagreen :magenta],
    markersize = iris.PetalWidth*5
```

```
)
```

[31]:

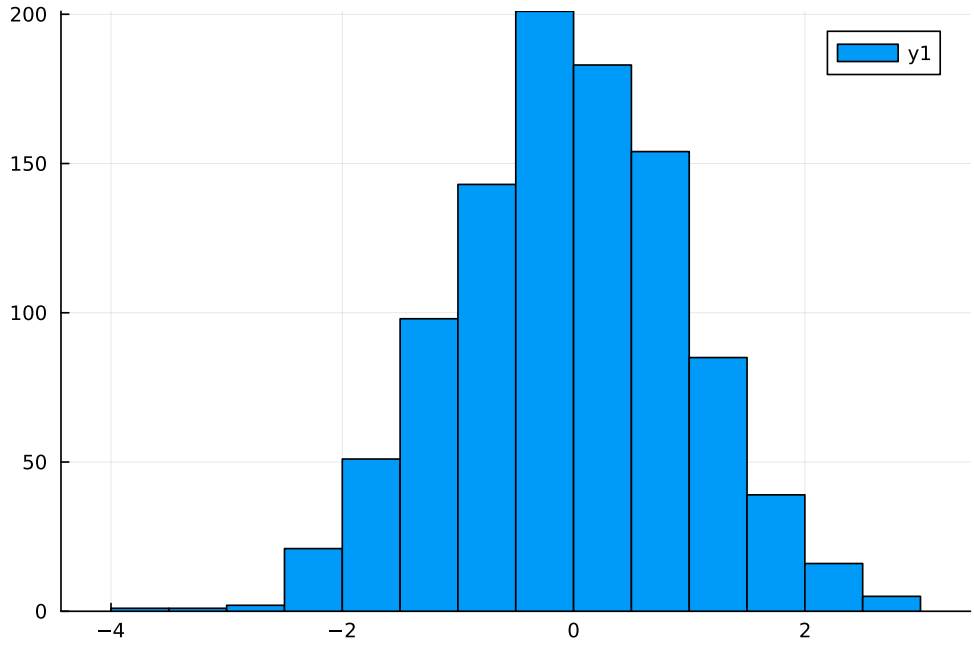


Auch hier können noch viele weitere Attribute ergänzt werden, diese findet man in diesem und den folgenden Abschnitten der Dokumentation.

Als letztes Beispiel aus Plots.jl ist hier noch ein Histogramm bei dem einige andere Attribute des Histogramms, wie Farbe der Plots oder *bins*, angepasst sind. Zudem werden hier auch zwei unterschiedlich plot-Typen kombiniert:

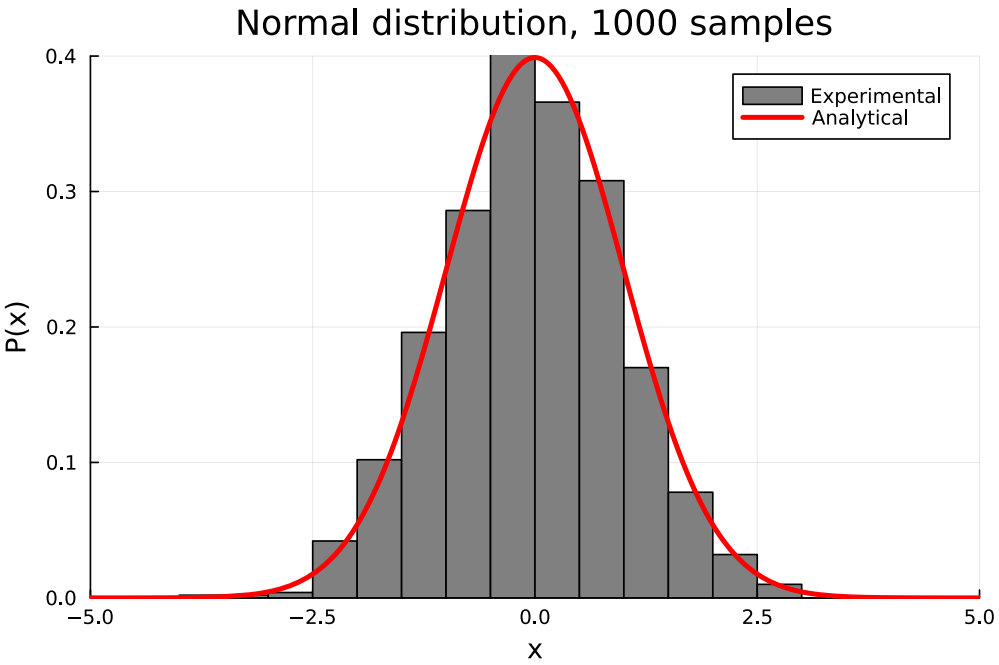
[32]:
```
x = randn(10^3)
histogram(x)
```

[32]:



[33]:
```
p(x) = 1/sqrt(2pi) * exp(-x^2/2)
b_range = range(-5, 5, length=21)

histogram(x, label="Experimental", bins=b_range, normalize=:pdf, color=:gray)
plot!(p, label="Analytical", lw=3, color=:red)
xlims!(-5, 5)
ylims!(0, 0.4)
title!("Normal distribution, 1000 samples")
xlabel!("x")
ylabel!("P(x)")
```

[33]:


Normal distribution, 1000 samples
Experimental
Analytical
P(x)
x
0.0
0.1
0.2
0.3
0.4
−5.0
−2.5
0.0
2.5
5.0

Kurs 6

6.1 I/O

In diesem Abschnitt wollen wir uns damit beschäftigen, wie wir in Julia Daten einlesen bzw. abspeichern können (input/output). Nehmen wir ab jetzt an, dass wir diesen DataFrame mit Daten speichern wollen:

[1]:
```
using DataFrames
df = DataFrame(
    "param1" => [1, 2],
    "param2" => [3, 4],
    "dicts" => [
        Dict(
            "d1" => 1.0,
            "d2" => 1.0
        ),
        Dict("d3" => 2.0)
    ]
)
```

[1]:

	param1	param2	dicts
	Int64	Int64	Dict…
1	1	3	Dict("d1->1.0, "d2->1.0)
2	2	4	Dict("d3->2.0)

6.1.1 CSV

Das wohl einfachste Dateiformat ist CSV (comma-seperated values format). Hier haben wir letzen Endes nur eine Textdatei, in welcher wir einzelne Werte durch Kommata abtrennen.

[2]:
```
using Pkg; Pkg.add("CSV")
using CSV
```

```
  Resolving package versions…
```

```
  No Changes to `~/.julia/environments/v1.10/Project.toml`
  No Changes to `~/.julia/environments/v1.10/Manifest.toml`
```

[3]:
```
CSV.write("data.csv", df)
```

[3]:
```
"data.csv"
```

Oft wird anstelle des Kommas auch ein anderes Trennzeichen verwendet (zum Beispiel ein Semikolon). Wenn man nämlich wie im Deutschen Floats als `1,0`, `2,0`, etc. kodiert, dann braucht man logischerweise ein anderes Trennzeichen. Das geht wie folgt:

[4]:
```
CSV.write("data.csv", df, delim=";")
```

[4]:
```
"data.csv"
```

[5]:
```
read_csv = CSV.read("data.csv", DataFrame; delim=";")
```

[5]:

	param1	param2	dicts
	Int64	Int64	String
1	1	3	Dict("d1-> 1.0, "d2-> 1.0)
2	2	4	Dict("d3-> 2.0)

Wie man aber hier schon sieht: Für geschachtelte Datenstrukturen ist CSV nicht besonders gut geeignet; unsere Dicts werden hier nur als String eingelesen:

[6]:
```
read_csv.dicts[1]
```

[6]:
```
"Dict(\"d1\" => 1.0, \"d2\" => 1.0)"
```

6.1.2 Excel

Häufig trifft man natürlich auch auf Excel-Tabellen, dafür empfehlen wir das Paket XLSX.jl. Mehr dazu im Praxiskurs Julia.

6.1.3 JSON

JSON steht für JavaScript Object Notation und ist ebenfalls leicht lesbar. Wir schreiben unsere Daten dabei wieder in ein Textfile; die Struktur ist allerdings nicht wie bei einer Tabelle. Diese ähnelt eher dem eines Dicts (also hat attribute-value Paare).

[7]:
```
Pkg.add("JSON"); using JSON
```

```
  Resolving package versions…
  No Changes to `~/.julia/environments/v1.10/Project.toml`
  No Changes to `~/.julia/environments/v1.10/Manifest.toml`
```

[8]:
```
stringdata = JSON.json(df)
println(stringdata)
```

```
{"param1":[1,2],"param2":[3,4],"dicts":[{"d1":1.0,"d2":1.0},{"d3":2.0}]}
```

[9]:
```
open("data.json", "w") do f
    write(f, stringdata)
end
```

[9]:
```
72
```

[10]:
```
# create variable to write the information
read_json = Dict()
open("data.json", "r") do f
    global read_json
    dicttxt = read(f,String) # file information to string
    read_json = JSON.parse(dicttxt)  # parse and transform d ta
end
read_json
```

[10]:
```
Dict{String, Any} with 3 entries:
  "param2" => Any[3, 4]
  "param1" => Any[1, 2]
  "dicts"  => Any[Dict{String, Any}("d1"=>1.0, "d2"=>1.0), Dict{String, Any}("d…
```

6.1.4 JLD2

Dieses Package erlaubt uns Daten im HDF5-Format zu speichern, welches besser als etwa JSON für große Datenmengen geeignet ist. Bei JSON hat man da einige Defizite: Bei JSON wird jeder Wert durch Characters repräsentiert. Das bedeutet, der Float `3.141592653589793` wird mit 17 Zeichen gespeichert, wovon jedes in der Regel 8-Bit braucht (siehe hier). Also speichern wir 136 anstelle der eigentlich benötigten 64 Bit! Zudem gibt es auch keine Kompression und der Zugriff auf Subdatensätze ist nicht performant. Bei HDF5 speichert man dagegen deutlich intelligenter; HDF5 ist nämlich ein Binärformat (binary format). Nähere Informationen findet man hier.

An dieser Stelle sei noch genannt, dass es alternative Pakete wie JLD.jl und HDF5.jl gibt, die je nach Anwendung besser geeignet sind.

[11]:
```
Pkg.add("JLD2"); using JLD2
```

```
  Resolving package versions…
 No Changes to `~/.julia/environments/v1.10/Project.toml`
 No Changes to `~/.julia/environments/v1.10/Manifest.toml`
```

[12]:
```
jldsave("df.jld2"; df)
```

[13]:
```
read_hdf5 = load("df.jld2")
```

[13]:
```
Dict{String, Any} with 1 entry:
  "df" => 2×3 DataFrame…
```

6.1.5 FileIO

Für viele Anwendungen müssen wir uns aber gar nicht mit diesen einzelnen Paketen rumschlagen, sondern brauchen nur das Paket `FileIO.jl` zu laden. Dieses stellt ein vereinheitlichtes Interface via `load` und `save` für zahlreiche Dateiformate bereit.

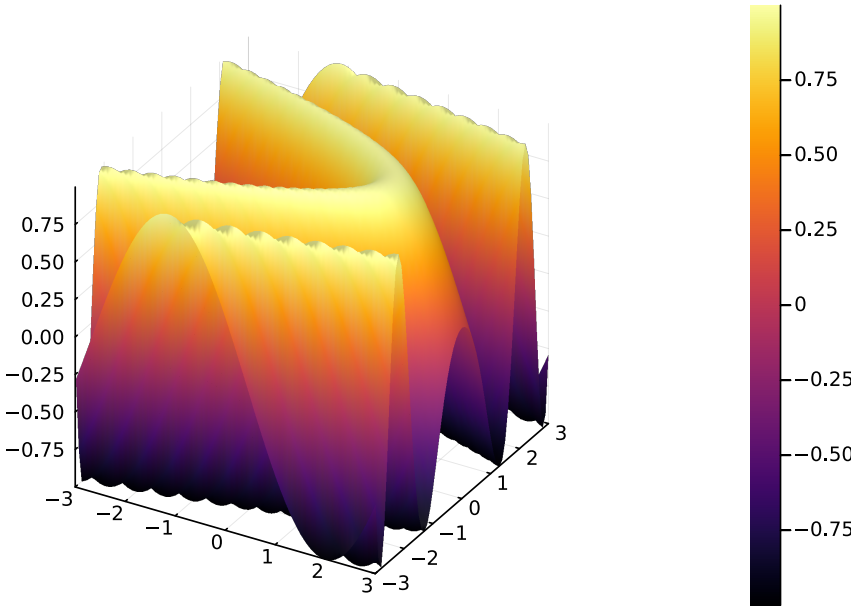
[14]:
```
Pkg.add("FileIO"); using FileIO
```

```
  Resolving package versions…

 No Changes to `~/.julia/environments/v1.10/Project.toml`
 No Changes to `~/.julia/environments/v1.10/Manifest.toml`
```

[15]:
```
x = collect(-3:0.1:3)
y = collect(-3:0.1:3)

xx = reshape([xi for xi in x for yj in y], length(y), length(x))
yy = reshape([yj for xi in x for yj in y], length(y), length(x))

z = sin.(xx .+ yy.^2)

data_dict = Dict("x" => x, "y" => y, "z" => z)

save("data_dict.jld2", data_dict)
```

[16]:
```
read_hdf5 = load("data_dict.jld2")

x2 = read_hdf5["x"]
y2 = read_hdf5["y"]
z2 = read_hdf5["z"]

using Plots; plot(x2, y2, z2, st = :surface)
```

[16]:



6.2 Lineare Regression

Wir wollen in diesem Kurs nicht näher auf dieses Thema eingehen; der Vollständigkeit halber sei hier ein Minimalbeispiel für Lineare Regression.

[17]:
```
using Pkg; Pkg.add("GLM")
```

```
  Resolving package versions…
 No Changes to `~/.julia/environments/v1.10/Project.toml`
 No Changes to `~/.julia/environments/v1.10/Manifest.toml`
```

[18]:
```
using DataFrames, GLM
data = DataFrame(X=[1,2,3], Y=[2,4,7])
```

[18]:

	X	Y
	Int64	Int64
1	1	2
2	2	4
3	3	7

[19]:
```
fm = @formula(Y ~ X)
linear_regressor = lm(fm, data)
```

[19]:
```
StatsModels.TableRegressionModel{LinearModel{GLM.LmResp{Vector{Float64}},
GLM.DensePredChol{Float64, LinearAlgebra.CholeskyPivoted{Float64,
Matrix{Float64}, Vector{Int64}}}}, Matrix{Float64}}

Y ~ 1 + X

Coefficients:
```

	Coef.	Std. Error	t	Pr(>|t|)	Lower 95%	Upper 95%
(Intercept)	-0.666667	0.62361	-1.07	0.4788	-8.59038	7.25704
X	2.5	0.288675	8.66	0.0732	-1.16797	6.16797

Wie man auf weitere Ergebnisse und Informationen zugreift, ist hier erklärt.

6.3 Typen

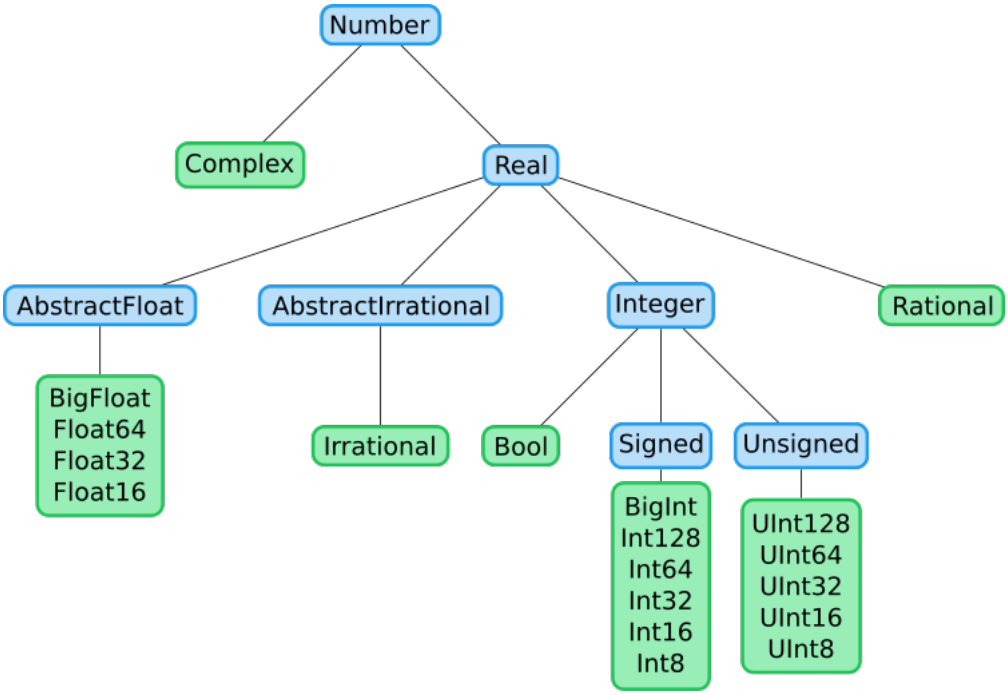
Wir haben schon ganz am Anfang festgestellt, dass quasi alles, mit dem wir in Julia herumhantieren, einen Typ hat. Das ist aber nicht nur ein Nebenkriegsschauplatz, sondern faktisch super wichtig, weil wir uns auch eigene Typen (aka Datenstrukturen) und das dazugehörige Verhalten definieren können.[1]

Grundsätzlich gibt es nur zwei Arten von Typen:

- abstract types: Können nicht instanziiert werden und haben keine Attribute (sind quasi Äste an Baum)
- concrete types: Können instanziiert werden, aber haben keine Subtypen (sind quasi Blätter an Baum).

Das wird an folgender Grafik ziemlich klar.

[1] Für die Leser mit etwas Vorwissen: Typen sind ähnlich zu structs in C, Klassen gibt es nicht. Im Gegensatz zu C definiert ein Typ aber tatsächlich einen neuen (benannten) Datentyp.



Alle blauen Felder sind *abstract types*, alles grünen Felder sind *concrete types*. Beispielsweise kann ein `Int64` initialisiert werden (einfach indem man etwa `a = 1` eingibt), ein `Real` kann das nicht. Dieser abstrakte Typ ist sozusagen lediglich der Kleber zwischen den verschiedenen reellwertigen Zahlen(sub-)typen. Dabei nennt man `Int64` einen *Subtyp* von `Real`; umgekehrt ist `Real` ein *Supertyp* von `Int64`. Weil ja wie gesagt in Julia alles aus Typen besteht, gilt es für diese nun einiges an Handwerkszeug zu erlernen.

6.3.1 Werkzeuge für Types

6.3.2 Type declarations

Wichtig ist zunächst der `::`-Operator. Wir benutzen ihn vor allem für sanity checks (wir bekommen compile time anstelle von runtime errors) Spezialisierungen von Funktionen. Manchmal hilft er aber auch dem Compiler schnelleren Code zu produzieren (dazu später mehr).

[20]:
```
(1+2)::Float64
```

```
TypeError: in typeassert, expected Float64, got a value of type Int64

Stacktrace:
 [1] top-level scope
```

```
   @ In[20]:1
```

[21]:
```
(1+2)::Int
```

[21]: 3

[22]:
```
foo::Int = 100.0
foo
```

[22]: 100

Die Funktion `typeof` haben wir bereits gesehen und gibt uns den concrete type eines instanziierten Typs zurück:

[23]:
```
typeof("abc")
```

[23]: String

[24]:
```
isa("abc", AbstractString) # String ist ein Subtyp von AbstractString!
```

[24]: true

[25]:
```
isa(1, Float64) # ein Integer ist kein Float!
```

[25]: false

[26]:
```
isa(1.0, Float64)
```

[26]: true

[27]:
```
1.0 isa Number # alternative Syntax
```

[27]: true

[28]:
```
supertype(Int64) # der direkte (erste) Supertyp von Int64
```

[28]: Signed

[29]:
```
subtypes(Real) # direkte (erste) Subtypen des abstract types Real
```

[29]:
```
7-element Vector{Any}:
 AbstractFloat
 AbstractIrrational
 FixedPointNumbers.FixedPoint
 Integer
 Rational
```

```
StatsBase.PValue
StatsBase.TestStat
```

[30]:
```
Int <: Real # <: checkt ob Typ ein Subtyp des rechten Typs ist
```

[30]:
```
true
```

[31]:
```
Any # alle Objekte sind davon ein Subtyp; das ist quasi die Wurzel unseres␣
↪Typenbaums
```

[31]:
```
Any
```

Ein kleine Sache von der man sich nicht verwirren lassen sollte: Wenn wir den Namen eines Typs eingeben, so wird dieser im Format `DataType` gehalten bzw. hat wiederum den Typ `DataType`:

[32]:
```
typeof(Int)
```

[32]:
```
DataType
```

[33]:
```
typeof(DataType) # hier diesselbe Logik
```

[33]:
```
DataType
```

[34]:
```
# wir schreiben uns gerade nochmal die typeof-Funktion
whichtype(::T) where T = T
whichtype("foo")
```

[34]:
```
String
```

6.3.3 Eigene Typen

Wir haben nun gesehen, wie wir mit bestehenden Typen umgehen können. Hier kommt nun der Teil, wo wir selbst kreativ werden können. Bevor wir zu dem sich eigentlich selbst erklärenden Beispiel kommen, sei noch gesagt: Es gibt verschiedene Arten von concrete types, ein `Int64` wäre beispielsweise ein *primitive type*. Für uns sind diese aber erstmal egal, viel wichtiger sind dagegen sogenannte *composite types*, die mit dem Keyword `struct` erstellt werden.

[35]:
```
abstract type Person end # abstract type

function fullname(p::Person) # type declaration in der Funktionssignatur; wir␣
↪definieren Verhalten hier für einen abstrakten Typ!
    return "$(p.name) $(p.lastname)" # Zugriff auf Datenfelder via .
end
```

```
struct Student <: Person # composite type
    name::String # Feld
    lastname::String # Feld
    age::Int # Feld
    major::String # Feld
end

s = Student("Jane", "Doe", 22, "Computer Science")
fullname(s)
```

[35]: "Jane Doe"

[36]:
```
s.name
```

[36]: "Jane"

[37]:
```
fullname(1) # geht nicht, weil für diesen Inputtypen unsere Funktion nicht␣
↪definiert ist!
```

```
MethodError: no method matching fullname(::Int64)

Closest candidates are:
  fullname(::Person)
   @ Main In[35]:3


Stacktrace:
 [1] top-level scope
   @ In[37]:1
```

Wie wir sehen, können wir uns einen concrete type Student instanziieren, indem wir einfach die Funktion `Student` aufrufen und als Argumente die benötigten Datenfelder übergeben. Weil sozusagen aus dem Rezept `struct ... end` ein konkret lebender Wert in `s` erzeugt wird, nennt man die Funktion `Student` einen Konstruktor. Genauer gesagt, haben es wir hier mit dem *default constructor* zu tun, den Julia uns automatisch bereitstellt.

Oftmals wollen wir aber den Konstruktionsprozess einer Instanz modifizieren, weil wir zum Beispiel manche Felder mit default-Werten befüllen wollen. Betrachte dazu

[38]:
```
abstract type Equity end # Eigenkapital

struct Stock <: Equity # Aktie
    symbol::String
```

```
    name::String
end

struct StockQuantity # Anzahl einer Aktie, zum Beispiel im TradeRepublic-Konto
    stock::Stock
    quantity
end

my_stock = Stock("ADS", "Adidas")
StockQuantity(my_stock, 2)
```

[38]:
```
StockQuantity(Stock("ADS", "Adidas"), 2)
```

Nun wollen wir sagen, dass wir per default eine Anzahl von 0 haben:

[39]:
```
StockQuantity(stock) = StockQuantity(stock, 0)
StockQuantity(my_stock)
```

[39]:
```
StockQuantity(Stock("ADS", "Adidas"), 0)
```

Dieses Vorgehen nennt man einen äußeren Konstruktor, weil eben außerhalb der Typdefinition ein neues „Rezept“ auftaucht. Manchmal wollen wir aber auch den default constructor überschreiben, weil wir zum Beispiel den Benutzer vor einer sinnfreien Initialisierung schützen wollen. Dafür können wir einen *inner constructor* verwenden:

[40]:
```
const DAX_companies = ["SAP", "BASF", "Merck"]

struct SafeStock <: Equity # Aktie kann nur mit sinnvollen Werten initialisiert␣
 ↪werden
    symbol::String
    name::String
    function SafeStock(symbol, name)
        if !(symbol in DAX_companies)
            println("$(symbol) ist keine bekannte AG!")
        else
            new(symbol, name)
        end
    end
end

SafeStock("DF", "d-fine")
```

```
DF ist keine bekannte AG!
```

[41]:
```
SafeStock("SAP", "SAP")
```

```
[41]: SafeStock("SAP", "SAP")
```

Parametrisierung

Bei unserem vorherigen Typ `StockQuantity` gibt es ein kleines Problem: Vielleicht wollen wir nun ein Feature in unserer Trading-App, die es erlaubt, auch nur Prozente einer Aktie zu halten. Dann müssten wir entweder einen neuen Typ schreiben, der dann ein Feld `quantity::Float64` hat (unpraktisch), oder wir lassen die type declaration weg und schreiben nur `quantity` (was äquivalent zu `quantity::Any` wäre). In letzterem Fall ist dann aber wiederum ungünstig, dass – wenn wir eine Variable vom Typ `StockQuantity` gegeben haben‘ – zur Kompilierzeit unklar ist, ob wir Prozente oder ganzzahlige Werte halten.

Deshalb gibt es für Typen noch eine nette Mechanik namens *Parametrisierung*:

```
[42]: struct StockHolding{T<:Number}
          stock::Stock
          quantity::T
      end

      StockHolding(my_stock, 0.5)
```

```
[42]: StockHolding{Float64}(Stock("ADS", "Adidas"), 0.5)
```

Mutability

Instanzen eines structs sind *immutable*, das bedeutet: Datenfelder können nach der Instanziierung nicht mehr verändert werden!

```
[43]: s.name = "Euler"
```

```
setfield!: immutable struct of type Student cannot be changed

Stacktrace:
 [1] setproperty!(x::Student, f::Symbol, v::String)
   @ Base ./Base.jl:41
 [2] top-level scope
   @ In[43]:1
```

Das ist per se gut, denn wenn der Compiler weiß, dass sich nichts ändern kann, dann muss genau dafür während der Laufzeit nicht mehr gecheckt werden, sprich Code kann stärker optimiert werden. Je nach Anwendung ist das aber schon eine Funktionalität, die wir gerne hätten. Deshalb benutzt man dafür *mutable composite types*:

[44]:
```
mutable struct InsecureStudent <: Person # könnte Studiengang wechseln
    name::String
    major::String
end

s = InsecureStudent("John Doe", "WiMa")
s.major = "taxidriver"
s
```

[44]:
```
InsecureStudent("John Doe", "taxidriver")
```

6.3.4 Unions

Der Union-Typ ist sehr nützlich, wenn wir verschiedene Datentypen kombinieren wollen, die aus verschiedenen Typhierarchien stammen.

[45]:
```
1 isa Union{Int, String}
```

[45]:
```
true
```

[46]:
```
"1" isa Union{Int, String}
```

[46]:
```
true
```

[47]:
```
abstract type Art end
struct Painting <: Art
    artist::String
    title::String
end
```

[48]:
```
struct BasketOfThings
    things::Vector{Union{Painting,Stock}}
    reason::String
end
```

[49]:
```
mona_lisa = Painting("Leonardo da Vinci", "Mona Lisa")
BasketOfThings([my_stock, mona_lisa], "Lehrpreis für das 3. OG")
```

[49]:
```
BasketOfThings(Union{Painting, Stock}[Stock("ADS", "Adidas"), Painting("Leonardo
da Vinci", "Mona Lisa")], "Lehrpreis für das 3. OG")
```

[50]:
```
"1" isa Union{Int, String}
```

[50]:
```
true
```

Kurs 7

7.1 Metaprogramming

Der Begriff Metaprogramming heißt soviel wie: Code, der Code generiert. Eine der schönen Eigenschaften von Julia ist, dass Metaprogramming hier sehr elegant funktioniert. In Sprachen wie C++ ist das immer absolut nervig, weil man sozusagen einen extra Überbau über die Sprache benötigt, der den Code manipuliert, *bevor* er kompiliert wird. Insbesondere also *bevor* der kompilierte Code ausgeführt werden kann. Das sind dann sogenannte preprocessor directives.

In Julia verwendet man stattdessen sogenannte Makros. Der große Vorteil gegenüber zu etwa C++ ist, dass man hier durch den JIT beispielsweise einen Variablennahmen mit regex *zur Laufzeit* erzeugen und dann im nächsten Schritt ein Makro zur Generierung von neuem Code mit dem Variablennahmen verwenden kann. Das heißt im Klartext: Diese strikte Trennung von Codeerzeugung und Codeausführung gibt es nicht mehr.

Offtopic erklärt das ganz gut den (mehr oder weniger lustigen) Witz vom Julia-Mitentwickler Jeff Bezanson, der Name Julia könne auch für "Jeff's uncommon lisp is automated" stehen. Lisp ist nämlich eine Programmiersprache deren Stärke insbesondere das Metaprogramming ist. Julias Funktionalität in diesem Bereich kommt also nicht von ungefähr.

7.1.1 Expressions

Um diese kleine Vorbemerkung ein bisschen plastischer zu machen, begeben wir uns nun auf eine Mini-Reise durch die Ausführungschritte von Julia-Code (ausführlichere Infos hier). Wenn wir in Julia eine Zeile Code eingeben, dann haben wir zunächst ja einfach nur Text, also einen String. Dieser wird dann in eine `Expr` (expression) umgewandelt. Das nennt man `parsen`:

[1]:
```
input_expr = Meta.parse("1 + 2")
```

[1]:
```
:(1 + 2)
```

[2]:
```
input_expr |> typeof
```

```
[2]: Expr
```

Unsere expression wird schlussendlich in ausführbaren Maschinencode übersetzt und ausgeführt:

```
[3]: eval(input_expr) # Nebenbemerkung: eval läuft IMMER in der global scope!
```

```
[3]: 3
```

Wenn wir jetzt aber nochmal einen Schritt zurück machen, dann sehen wir, dass in der `Expr` unser Eingabetext in eine hierarchische Baumstruktur gebracht wurde:

```
[4]: input_expr |> dump
```

```
Expr
  head: Symbol call
  args: Array{Any}((3,))
    1: Symbol +
    2: Int64 1
    3: Int64 2
```

```
[5]: input_expr.head
```

```
[5]: :call
```

```
[6]: input_expr.args
```

```
[6]: 3-element Vector{Any}:
      :+
     1
     2
```

Dabei gibt es immer einen *head* und einen *body*, also sozusagen Stamm und Äste. In unserem Fall ist der head `:call`, weil wir den Operator `+` auf die Inputs `1` und `2` anwenden. Betrachten wir ein etwas schwierigeres Beispiel:

```
[7]: advanced_expr = :(f(1) + 2) # wir erstellen uns direkt eine Expr
```

```
[7]: :(f(1) + 2)
```

Hier stellen wir nun fest, dass anstelle der `1` eine weitere (geschachtelte) Expression getreten ist:

```
[8]: advanced_expr |> dump
```

```
Expr
  head: Symbol call
  args: Array{Any}((3,))
```

```
1: Symbol +
2: Expr
  head: Symbol call
  args: Array{Any}((2,))
    1: Symbol f
    2: Int64 1
3: Int64 2
```

Ganz analog ist in dieser der head `:call`, weil wir wir einen Funktionsaufruf (call) `f(x)` haben.

[9]:
```
advanced_expr.args[2].head
```

[9]:
```
:call
```

Außerdem können wir (ähnlich wie bei Strings) *interpolation* nutzen:

[10]:
```
x = 2
:(sqrt($x))
```

[10]:
```
:(sqrt(2))
```

7.1.2 Mickey Mouse Example

Wofür brauchen wir nun Makros? Nehmen wir einmal an, wir würden gerne die Variablen `a` bis `z` so belegen, dass `a = 1, ..., z = 26`. Dann wollen wir das natürlich nicht händisch ausschreiben, sondern möglichst arbeitssparend tun. Dafür können wir das Makro `@eval` nutzen:

[11]:
```
# Anfangsbuchstabe (character) ist a
ch = 'a'

for i in 1:26
    # wir nutzen nun das Makro, weil wir so nicht extra eine Expr erzeugen müssen
    @eval $(Symbol(ch)) = $i # Interpolation mit $
    ch += 1
end

y
```

[11]:
```
25
```

Jetzt kann man sich fragen: Warum genau sollte man das tun wollen? Variablen derart zu benennen ist wahrscheinlich Quatsch, stattdessen würde man wohl eher mit dem Array `collect(1:26)` arbeiten. Eine sinnvolle Anwendung findet man aber zum Beispiel beim Makro `@assert`:

```
[12]: @macroexpand @assert (a == 1) "a ist nicht 1!!"
```

```
[12]: :(if a == 1
          nothing
      else
          Base.throw(Base.AssertionError("a ist nicht 1!!"))
      end)
```

7.1.3 Eigene Makros

Grundsätzlich gibt es viele Zwecke für Metaprogramming, aber in der Regel fällt die Nutzung von Makros in eine von zwei Kategorien:

- Wir wollen neue "language features" implementieren
- Wir wollen syntactic sugar (hübschere Schreibweisen)

Betrachten wir ein einfaches Makro, das quadriert:

```
[13]: macro squared(ex)
          return :($(ex) * $(ex))
      end

      @squared(2)
```

```
[13]: 4
```

```
[14]: @macroexpand @squared(2)
```

```
[14]: :(2 * 2)
```

Auf den ersten Blick sieht alles gut aus, allerdings gibt es ein Problem:

```
[15]: function foo()
          x = 2
          return @squared x
      end
```

```
[15]: foo (generic function with 1 method)
```

```
[16]: foo()
```

```
[16]: 576
```

Unser Makro interpretiert `x` als eine globale Variable und nicht als lokale Variable in der Funktion! Der Versuch, dort den entsprechenden Wert reinzukopieren, schlägt also fehl.

[17]:
```
@code_lowered foo()
```

[17]:
```
CodeInfo(
1 ─      x = 2
│   %2 = Main.x * Main.x
└──      return %2
)
```

Dieses Problem lässt sich durch *escaping* beheben. Wir teilen unserem Makro mit, dass `ex` vom Compiler in Ruhe gelassen werden sollte.

[18]:
```
macro squared(ex)
    return :($(esc(ex)) * $(esc(ex)))
end

function foo()
    x = 2
    return @squared x
end
```

[18]:
```
foo (generic function with 1 method)
```

[19]:
```
foo()
```

[19]:
```
4
```

Hier ist ein weiteres Beispiel, wo escaping notwendig ist; der Compiler ersetzt Variablen- und Funktionsnamen stets durch eigene. Das wollen wir hier nicht.

[20]:
```
macro trick(expr)
    trick_msg = :(println("Wir benutzen eine trickreiche Funktion!"))
    return Expr(:block, trick_msg, expr)
end
```

[20]:
```
@trick (macro with 1 method)
```

[21]:
```
@trick bar(x) = 1
```

```
Wir benutzen eine trickreiche Funktion!
```

[21]:
```
#30#bar (generic function with 1 method)
```

[22]:
```
macro trick(expr)
    trick_msg = :(println("Wir benutzen eine trickreiche Funktion!"))
    return esc(Expr(:block, trick_msg, expr))
end
```

```
[22]: @trick (macro with 1 method)
```

```
[23]: @trick foobar(x) = 1
```

```
Wir benutzen eine trickreiche Funktion!
```

```
[23]: foobar (generic function with 1 method)
```

Wichtige Makros sind außerdem - `@time` - `@macroexpand` - `@which` - `@edit`

7.1.4 Ausblick: Loop unrolling (SIMD)

Ein anwendungsnäheres Beispiel als oben wäre etwa das sogenannte loop unrolling. Sagen wir, wir haben einen Vektor, der als Länge ein Vielfaches von 4 hat. Nun muss man wissen: Viele Hardwarearchitekturen haben beschleunigte Operationen implementiert, wenn man viele punktweise (gleiche) Operationen durchführt. Das nennt man auch SIMD (single instruction, multiple data). Weil Loops erst zur Laufzeit ausgeführt werden, weiß der Kompiler aber nicht immer, dass wir hier so etwas vorliegen haben. Wir können uns also ein Makro schreiben, das aus

```
for i=1:4n
    c[i] = a[i]*b[i]
end
```

den Code

```
for i=1:n
    c[i*4]     = a[i*4]    *b[i*4]
    c[i*4 + 1] = a[i*4 + 1]*b[i*4 + 1]
    c[i*4 + 2] = a[i*4 + 2]*b[i*4 + 2]
    c[i*4 + 3] = a[i*4 + 3]*b[i*4 + 3]
end
```

generiert.

Auf diese Weise weiß der Compiler, dass hier sozusagen 4 mal die gleiche Operation durchgeführt wird und kann diesen Prozess optimieren.

7.1.5 Ausblick: Eigene Syntax bzw. Modelldefinitionen

Ebenfalls praktisch sind Makros beim Erstellen von eigener Modellsyntax. Ein Beispiel aus eigener Hand wäre etwa die Definition eines makroökonomischen Modells. Um jetzt nicht zu sehr abzuschweifen, hier die Intuition nur ganz kurz: Man bekommt als Problem ein (nichtlineares) Gleichungssystem und will da einen Solver draufwerfen. Dazu kann man das Gleichungssystem in die Form $f(x) = 0$ bringen, wobei $x = (x_1, ..., x_n)$ ein Array in Julia ist.

Realiter hat man aber viele verschiedene Modelle mit jeweils 10-100 Gleichungen und will daher das Modell nicht von Hand umstellen. Das geht aber mit Metaprogramming automatisch! In dem untenstehenden Screenshot sieht man das eigene Makro `@sfc_model` mit verschiedenen Submakros, womit man das Modell definieren kann. Durch ein `Dict` werden Variablennamen automatisch auf Arrayindizes gemapt.

```
1  @sfc_model begin
2      @endogenous Y T YD C H_s H_h H
3      @exogenous G
4      @parameters θ α_1 α_2
5      @model begin
6          Y = C + G
7          T = θ*Y
8          YD = Y-T
9          C = α_1*YD + α_2*H[-1]
10         H_s + H_s[-1] = G - T
11         H_h + H_h[-1] = YD - C
12         H = H_s + H_s[-1] + H[-1]
13     end
14 end | Endogenous Variables: [:Y, :T, :YD, :C, :H_s, :H_h, :H]
15
16 values = Dict(
17     :θ => 0.2,
18     :α_1 => 0.6,
19     :α_2 => 0.4,
20 )
```

Hier sieht man den Julia-Ouput; es wird (magic!) automatisch eine Funktion `f!` erzeugt, die (wenn die exogenen Variablen bzw. Parameter bereitgestellt werden) man dem Solver übergeben kann.

```
function f!(diff, endos, lags, exos, params)
    diff[1] = endos[1] - (endos[4] + exos[1, 1])
    diff[2] = endos[2] - params[1] * endos[1]
    diff[3] = endos[3] - (endos[1] - endos[2])
    diff[4] = endos[4] - (params[2] * endos[3] + params[3] * lags[7, 1])
    diff[5] = (endos[5] + lags[5, 1]) - (exos[1, 1] - endos[2])
    diff[6] = (endos[6] + lags[6, 1]) - (endos[3] - endos[4])
    diff[7] = endos[7] - (endos[5] + lags[5, 1] + lags[7, 1])
end

Endogenous Variables: [:Y, :T, :YD, :C, :H_s, :H_h, :H]
Exogenous Variables: [:G]
Parameters: [:θ, :α_1, :α_2]
Equations:

   (1)  Y = C + G
   (2)  T = θ * Y
   (3)  YD = Y - T
   (4)  C = α_1 * YD + α_2 * H[-1]
   (5)  H_s + H_s[-1] = G - T
   (6)  H_h + H_h[-1] = YD - C
   (7)  H = H_s + H_s[-1] + H[-1]


julia>
```

Die Lösung könnte mithilfe von NLsolve.jl, wobei `F` Ergebnis und `x` Input der Funktion ist, dann so aussehen:

```
nlsolve((F, x) -> f!(F, x, lags, exos, params), initial_values, autodiff = :forward)
```

Kurs 8

Nachdem wir uns in den vergangenen Kursen mit vielen Programmierkonzepten näher beschäftigt haben (allen voran Typen und Makros), wollen wir nun ein praxisnahes Beispiel betrachten, welches ein bisschen die verschiedenen Fäden zusammenführt. Das Setting ist: Definiere Arithmetik für verschiedene Temparatureinheiten (von hier geklaut).

8.1 Ein fancy Beispiel

[1]:
```
# Wir wollen zu diesen Funktionen (die bereits ab Werk in Julia sind, deshalb im␣
 ↪Base module) neue Funktionalität hinzufügen.
import Base: +, -, *, promote, promote_rule, convert, show

# definiere neuen Typ
abstract type Temperature end
types = [:Celsius, :Kelvin, :Fahrenheit]

# lege für jede Temperatur interne Daten an und definiere Addition/Subtraktion␣
 ↪gleicher Temperaturen
for T in types
    # @eval begin ... end analog zu eval(...)
    @eval begin
        # $T kopiert die entsprechende Temperatur sozusagen als Teil des Codes an␣
 ↪die Stelle nach struct,
        # damit wir diesen Block nicht mehrfach schreiben müssen
        # Das ist metaprogramming!
        struct $T <: Temperature
            value::Float64
        end

        +(x::$T, y::$T) = $T(x.value + y.value)
        -(x::$T, y::$T) = $T(x.value - y.value)
    end
end
```

[2]:
```
Celsius(1.0)
```

[2]:
```
Celsius(1.0)
```

[3]:
```
+(Celsius(1.0), Celsius(2.0))
```

[3]:
```
Celsius(3.0)
```

[4]:
```
Celsius(1.0) + Celsius(2.0)
```

[4]:
```
Celsius(3.0)
```

[5]:
```
# Lege fest, wie t in erstes Argument umgwandelt wird.
convert(::Type{Kelvin}, t::Celsius) = Kelvin(t.value + 273.15)
convert(::Type{Kelvin}, t::Fahrenheit) = Kelvin(Celsius(t))
convert(::Type{Celsius}, t::Kelvin) = Celsius(t.value - 273.15)
convert(::Type{Celsius}, t::Fahrenheit) = Celsius((t.value - 32) * 5 / 9)
```

```
# Wenn wir wollen, können wir auch conversion in Fahrenheit definieren
# convert(::Type{Fahrenheit}, t::Celsius) = Fahrenheit(t.value*9/5 + 32)
# convert(::Type{Fahrenheit}, t::Kelvin) = Fahrenheit(Ceslius(t))
```

[5]: convert (generic function with 197 methods)

```
[6]: convert(Kelvin, Celsius(1.0))
```

[6]: Kelvin(274.15)

```
[7]: # Baue die obigen convert-Funktionen in die Konstruktoren ein.
for T in types, S in types
    if S != T
        @eval $T(temp::$S) = convert($T, temp)
    end
end
```

```
[8]: # Wir bekommen dieses nette Verhalten (Initialisierung mit anderen Temperaturen
 ↪ist möglich)
Kelvin(Celsius(1.0))
```

[8]: Kelvin(274.15)

Mit *promotion* bezeichnet man die *conversion* von gemischten Typen zu einem gemeinsamen Typ:

```
[9]: # Gegeben zwei Einheiten: Welche hätten wir lieber?
promote_rule(::Type{Kelvin}, ::Type{Celsius}) = Kelvin
promote_rule(::Type{Fahrenheit}, ::Type{Kelvin}) = Kelvin
promote_rule(::Type{Fahrenheit}, ::Type{Celsius}) = Celsius
```

[9]: promote_rule (generic function with 135 methods)

```
[10]: promote_type(Kelvin, Celsius)
```

[10]: Kelvin

```
[11]: promote(Kelvin(1.0), Celsius(1.0))
```

[11]: (Kelvin(1.0), Kelvin(274.15))

```
[12]: # definiere Arithmetik für unterschiedliche Temperatur-structs
+(x::Temperature, y::Temperature) = +(promote(x, y)...);
-(x::Temperature, y::Temperature) = -(promote(x, y)...);
```

[13]:
```
Fahrenheit(4) + Celsius(5)
```

[13]:
```
Celsius(-10.555555555555555)
```

Was passiert hier? Zunächst wird durch promotion gefragt, in welchem Typ wir gerne rechnen würden. Entsprechend werden beide Inputs dorthin konvertiert. Für jeden einzelnen Typ ist Arithmetik aber bereits wohldefiniert, also bekommen wir etwas Sinnvolles raus.

[14]:
```
# Wir wollen am Ende schreiben können: 2°K anstelle Kelvin(2)
abstract type TemperatureSymbol end
# Lege für jede Temperatur einen zusätzlichen struct an, damit dieser als␣
 ↪repräsentatives Symbol benutzt werden kann
# ist nicht SI-konform, aber nvm
symbols = Symbol.(["°C", "°K", "°F"])
for i in 1:length(symbols)
    @eval begin
        struct $(symbols[i]) <: TemperatureSymbol
        end
        *(x::Real, y::Type{$(symbols[i])}) = $(types[i])(x)
    end
end
```

[15]:
```
Kelvin(2)
```

[15]:
```
Kelvin(2.0)
```

Folgendes soll fehlschlagen; würde man sich den struct `TemperatureSymbol` sparen und direkt die Multiplikation auf Temperaturen definieren, dann hätte man `const °C = Celsius(1)` und man hätte diesen Sicherheitsmechanismus nicht (Operation würde ungewünscht funktionieren).

[16]:
```
2°C + °C
```

```
MethodError: no method matching +(::Celsius, ::Type{°C})

Closest candidates are:
  +(::Any, ::Any, ::Any, ::Any…)
   @ Base operators.jl:587
  +(::Celsius, ::Celsius)
   @ Main In[1]:19
  +(::Temperature, ::Temperature)
   @ Main In[12]:2
```

```
Stacktrace:
 [1] top-level scope
   @ In[16]:1
```

[17]:
```
0°C + 0°K
```

[17]:
```
Kelvin(273.15)
```

Nun implementieren wir noch eine hübschere Ausgabe:

[18]:
```
# custom pretty-printing (nutze Base.show)
for T in types
    # Die Benennung der Ausgabe ist hier auch ein wieder ein bisschen arbiträr, nicht SI-konform
    @eval show(io::IO, x::$T) = println("$(x.value) °" * $(String(T)))
end
Kelvin(1)
```

[18]:
```
1.0 °Kelvin
```

Kurs 9

9.1 Zufallszahlen

Für viele Anwendungen braucht man Zufallszahlen. Zwar haben wir in Julia nur Pseudozufallszahlen (die Zahlen werden nach einer deterministischen Methode berechnet), die sich zudem irgendwann wiederholen (das nennt man Periode). Allerdings verhalten sich die Zahlen im besten Fall sehr ähnlich zu tatsächlich zufällig (aus einer Verteilung) gezogenen Zahlen und die Periode ist oft so groß, dass sie in der Praxis keine Rolle spielt (für den Mersenne-Twister: `4.3*10^6001`).

[1]:
```
# Die Ausgabe dieses Befehls ist beim ersten Ausführen länger
using Pkg
Pkg.add("Random")
```

```
  Resolving package versions…

 No Changes to `~/.julia/environments/v1.10/Project.toml`
 No Changes to `~/.julia/environments/v1.10/Manifest.toml`
```

[2]:
```
using Random
```

Zentral sind die ab Werk verfügbaren Sampler der uniformen Verteilung und Normalverteilung:

[3]:
```
rand() # uniform verteilter Wert
```

[3]:
```
0.4117587007419041
```

[4]:
```
randn() # normalverteilter Wert
```

[4]:
```
-0.8897390477233826
```

Um zu zeigen, dass die Funktionen funktionieren wie gedacht, ist hier ein Histogramm mit Kerndichteschätzer (wenn einem das nichts sagt – einfach ignorieren).

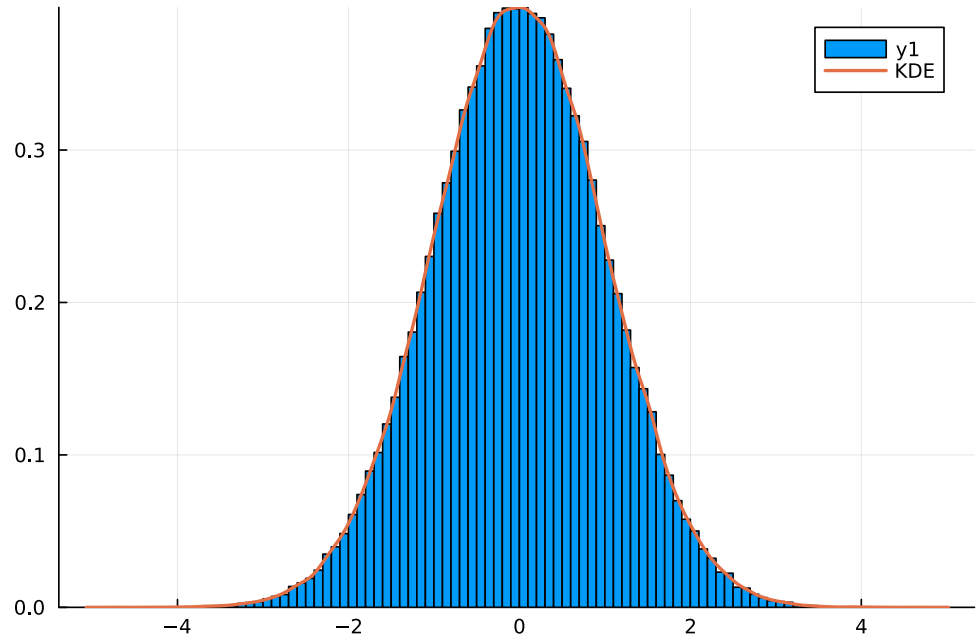
[5]:
```
Pkg.add("KernelDensity")
```

```
  Resolving package versions…
```

```
  No Changes to `~/.julia/environments/v1.10/Project.toml`
  No Changes to `~/.julia/environments/v1.10/Manifest.toml`
```

[6]:
```
using Plots
using KernelDensity

data = randn(100000)
density = kde(data)
histogram(
    data,
    normalize = true
)
plot!(density.x, density.density, linewidth=2, label="KDE")
```

[6]:



Um reproduzierbare Zufallszahlen zu generieren, setzen wir einen sogenannten *seed*. Das ist extrem nützlich, wenn wir zum Beispiel Simulationen in einer Studie überprüfbar machen wollen.

[7]:
```
println(rand(2))
println(rand(2))

Random.seed!(1) # setze den seed auf 1
```

```
println(rand(2))
println(rand(2))

Random.seed!(1) # setze den seed auf 1
println(rand(2))
```

```
[0.6754513108273708, 0.9211369853177723]
[0.10438708209419478, 0.16297616901254075]
[0.07336635446929285, 0.34924148955718615]
[0.6988266836914685, 0.6282647403425017]
[0.07336635446929285, 0.34924148955718615]
```

9.1.1 Distributions.jl

Das Paket Distributions.jl stellt uns viele weitere Verteilungen bereit.

[8]:
```
Pkg.add("Distributions")
using Distributions
```

```
  Resolving package versions…
 No Changes to `~/.julia/environments/v1.10/Project.toml`
 No Changes to `~/.julia/environments/v1.10/Manifest.toml`
```

[9]:
```
rand(Cauchy())
```

[9]:
```
0.7209257129375767
```

9.2 Performance

Julia ist per se ziemlich schnell – das ist ja gerade einer der Gründe, warum wir diese Sprache nutzen. Allerdings gibt es für ein bestimmtes Problem oft viele Wege nach Rom, wobei manche effizienter als andere sind. In manchen Fällen ist es sogar so, dass die Suche nach Optimierungen ein Fass ohne Boden ist.

Gerade deshalb sollte man sich aber nicht immer den Kopf über jedes Detail zerbrechen, das möglicherweise performancerelevant ist. Donald E. Knuth (legendärer Programmierer und unter anderem Erfinder von TeX) sagte dazu mal: „Premature optimization is the root of all evil“. In anderen Worten: Meistens sollte man eher versuchen schönen generischen Code zu schreiben und sich hinterher um Details kümmern, als sich umgekehrt in diesen zu verrennen – was nicht nur Zeit kostet, sondern im schlimmsten Fall zu Spaghetticode führt (siehe auch hier).

Sinnvoll ist dagegen zum Beispiel ein sogenannter Profiler (etwa `@profview`), der uns Fingerzeige liefert, wo möglicherweise Speed flöten geht.

9.2.1 Type instabilities

Julia ist deshalb schnell, weil es für unterschiedliche Typen als Inputs auch tatsächlich verschiedene Funktionen (oder Codeabschnitte) in Assembler kompiliert. Man muss also nicht innerhalb der Funktion sozusagen wieder abchecken: "Was wäre, wenn hier jetzt diese Operation mit Typ X wäre?". Stattdessen wird die Funktion von vornherein spezialisiert. Umgekehrt müssen wir diesen Prozess nicht händisch wie in C/C++ machen, stattdessen wird das für uns von Julia übernommen.

Es gibt allerdings Fälle, in denen ist Julia sich nicht sicher, ob sich der Typ innerhalb eines Codeabschnitts ändern kann und geht deshalb auf Nummer sicher. Soll heißen: Dieser Codeabschnitt ist dann weniger stark spezialisert und operiert üblicherweise auf Typen der Art `Union{Typ_1, Typ_2}`. Das ist auch einer der Gründe, warum man *globale Variablen* vermeiden sollte (wenngleich sich hier nicht nur der Typ, sondern zusätzlich auch der Wert unvorhergesehen ändern kann). Für die, die es genauer wissen wollen, hier ein kleiner Thread zur Frage, warum es überhaupt type instabilities gibt.

Realitätsnahes Beispiel 1

[10]:
```
Pkg.add("BenchmarkTools")
```

```
  Resolving package versions…
 No Changes to `~/.julia/environments/v1.10/Project.toml`
 No Changes to `~/.julia/environments/v1.10/Manifest.toml`
```

[11]:
```
using BenchmarkTools
```

`@time`, printed die benötigte Zeit und gibt das Ergebnis zurück. Aber Achtung: Beim ersten Ausführen muss der Code erst kompiliert, sodass wir beim ersten Mal immer langsamer sind!. Deshalb benutzen wir das Makro `@btime`, das mehrere Durchläufe macht und die Zeit zum Kompilieren ignoriert.

[12]:
```
aa = Any[1:1000;];
ai = [1:1000;];

@btime sum($aa)
@btime sum($ai)
```

```
  12.666 µs (969 allocations: 15.14 KiB)
  117.387 ns (0 allocations: 0 bytes)
```

[12]:
```
500500
```

[13]:
```
@code_warntype sum(aa)
```

```
MethodInstance for sum(::Vector{Any})
  from sum(a::AbstractArray; dims, kw…) @
Base reducedim.jl:1010
Arguments
  #self#::Core.Const(sum)
  a::Vector{Any}
Body::Any
1 —
      nothing
│   %2 = Base.:(:)::Core.Const(Colon())
│   %3 = Core
.NamedTuple()::Core.Const(NamedTuple())
│   %4 = Base.pairs(%3)::Core.Const(Base.Pairs{Symbol, Union{},
Tuple{}, @NamedTuple{}}())
│   %5 = Base.:(var"#sum#828")(%2, %4, #self#,
a)::Any
└──      return %5
```

[14]:
```
@code_warntype sum(ai)
```

```
MethodInstance for sum(::Vector{Int64})
  from sum(a::AbstractArray; dims, kw…) @
Base reducedim.jl:1010
Arguments
  #self#::Core.Const(sum)
  a::Vector{Int64}
Body::Int64
1 —      nothing
│   %2 = Base.:(:)::Core.Const(Colon())
│   %3 = Core.NamedTuple()::Core.Const(NamedTuple())
│   %4 = Base.pairs(%3)::Core.Const(Base.Pairs{Symbol, Union{},
Tuple{}, @NamedTuple{}}())
│   %5 = Base.:(var"#sum#828")(%2, %4, #self#, a)::Int64
└──      return %5
```

Realitätsnahes Beispiel 2

[15]:
```
struct RealPoint
    x::Real
    y::Real
end
```

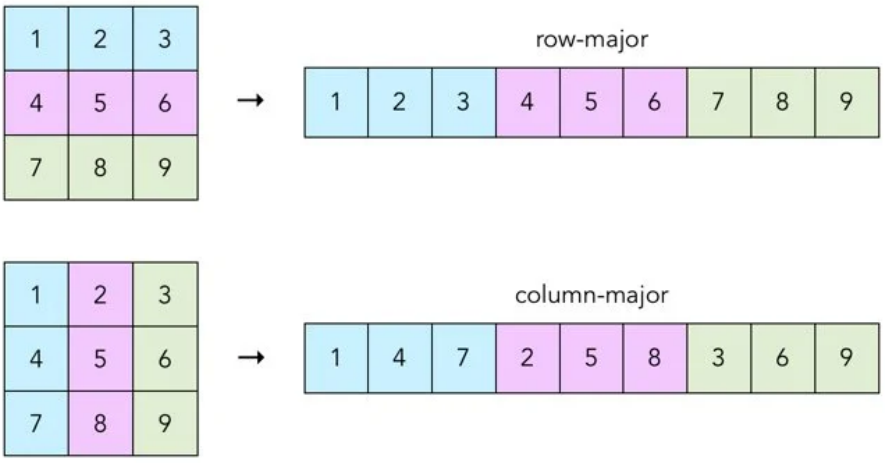
Hier haben wir den abstrakten Typ `Real` als Typ unserer Felder verwendet. Das funktioniert zwar, aber ist nicht besonders toll. Denn `Real` ist kein konkreter Typ: Der Julia-Compiler kann keine Annahmen über das Datenlayout von RealPoint treffen, und die Methodenauswahl muss zur Laufzeit und nicht zur Kompilierzeit erfolgen. Der richtige Weg, dies zu tun, wäre wie folgt:

[16]:
```
struct Point{T<:Real}
    x::T
    y::T
end
```

9.2.2 Row vs. column major

Konzept

In den meisten Programmiersprachen sind Matrizen bzw. Arrays technisch nichts anderes als eindimensionale Arrays, also quasi Vektoren (Achtung: Wir reden hier nicht von echten Typen der Art `Vector`). Das bedeutet: Elemente werden zeilen- oder spaltenweise (row-/column major) sukzessive in einer langen Liste gespeichert.



Es kommt ein bisschen auf Hardware bzw. konkrete Architektur an, aber generell gilt: Man möchte tendenziell auf (physisch) nahe beieinanderliegende Elemente zugreifen, weil der Zugriff dann schneller erfolgt. In anderen Worten: Im Algorithmus sollten häufige Sprünge möglichst vermieden werden!

Ein Beispiel dafür wäre die Implementierung einer Matrixmultiplikation.

Beispiel: Matrixmultiplikation

[17]:
```
# Die Ausgabe dieses Befehls ist beim ersten Ausführen länger
Pkg.add("BenchmarkTools")
```

```
  Resolving package versions…
 No Changes to `~/.julia/environments/v1.10/Project.toml`
 No Changes to `~/.julia/environments/v1.10/Manifest.toml`
```

[18]:
```
using BenchmarkTools

m = 1000; n = 1000; k = 1000
X = rand(m, k)
Y = rand(k, n)
Z = zeros(m, n)

function naive_matmul!(A, B, C)
    C .= 0
    for i in 1:size(A)[1]
        for j in 1:size(B)[2]
            for k in 1:size(A)[2]
                @inbounds C[i, j] += A[i, k] * B[k, j]
            end
        end
    end
end

@btime naive_matmul!(X, Y, Z)
isapprox(Z, X * Y, atol = 1e-10)
```

```
  860.442 ms (0 allocations: 0 bytes)
```

[18]:
```
false
```

[19]:
```
# lediglich Reihenfolge j, k, i ist anders
function smart_matmul!(A, B, C)
    C .= 0
```

```
    # @simd bringt hier scheinbar nichts
    @simd for j in 1:size(B)[2]
        for k in 1:size(A)[2]
            for i in 1:size(A)[1]
                @inbounds C[i, j] += A[i, k] * B[k, j]
            end
        end
    end
end

@btime smart_matmul!(X, Y, Z)
isapprox(Z, X * Y, atol = 1e-10)
```

```
  146.564 ms (0 allocations: 0 bytes)
```

[19]:
```
false
```

Hieran kann man auch gut erkennen, dass man in der Regel keine neuen Objekte erzeugen, sondern auf bereits existierende operieren möchte. Konkret: Die Matrix C wird nicht neu definiert und mit `return` zurückgegeben. Denn wäre dies der Fall, so müsste bei jedem Aufruf der Matrixmultiplikation neuer Speicher angelegt werden. Und wenn wir das öfters tun, kostet es Zeit.

9.2.3 Ausblick

Viele Probleme, die aus mathematischer Sicht trivial sind, sind in der Implementierung alles andere als das. Wie man hier etwa sehen kann, sind wir immer noch mindestens Faktor 10 von einer wirklich performanten Version entfernt:

[20]:
```
@btime Z = X * Y
```

```
  12.146 ms (2 allocations: 7.63 MiB)
```

[20]:
```
1000×1000 Matrix{Float64}:
 248.763  244.419  255.15   247.548  …  252.651  251.893  249.701  241.943
 242.098  240.038  238.714  243.128     245.266  244.324  244.971  235.97
 260.221  248.341  251.112  252.18      257.177  252.312  256.524  250.629
 251.871  239.562  246.121  247.213     252.698  246.607  251.3    244.089
 250.492  244.648  251.3    253.718     251.79   254.043  254.998  247.755
 251.3    245.925  248.781  253.141  …  260.304  252.432  255.33   254.409
 247.171  236.187  243.783  246.074     251.934  242.487  249.611  242.461
 257.26   250.23   255.792  260.869     260.035  256.13   252.936  256.314
 256.703  256.49   260.458  258.509     258.233  263.285  263.036  254.804
 252.319  241.425  248.421  246.819     248.246  252.53   256.604  248.1
 262.511  252.759  253.938  258.193  …  257.686  254.082  257.841  250.738
```

```
253.734  250.619  247.925  249.73      251.59   252.495  251.937  241.399
250.237  240.907  248.345  250.376     251.86   249.028  251.196  245.941
  ⋮                                  ⋱
251.865  242.414  250.404  251.106     253.066  255.31   246.305  243.361
251.111  236.434  245.811  248.242     248.217  244.752  250.946  243.306
258.549  256.446  258.103  260.924  …  264.117  258.981  261.268  254.391
256.509  251.609  258.368  254.877     254.718  252.008  253.88   248.642
241.859  232.635  235.538  237.647     244.688  236.064  237.706  237.525
247.832  240.951  243.747  248.087     250.267  246.553  246.668  245.329
254.292  244.25   248.282  251.942     257.054  253.575  254.469  248.239
248.788  240.074  253.666  248.974  …  255.079  252.774  252.532  245.472
252.233  241.474  247.52   252.184     254.788  250.691  250.82   242.858
251.34   242.793  252.963  249.043     250.915  253.709  256.167  245.089
239.116  233.265  242.228  239.185     246.716  241.815  246.695  239.462
250.75   245.041  249.633  252.386     252.384  251.775  251.874  246.731
```

Ein großer Teil dieser Optimierung AVX.

[21]:
```
# Die Ausgabe dieses Befehls ist beim ersten Ausführen länger
Pkg.add("LoopVectorization")
```

```
  Resolving package versions…
 No Changes to `~/.julia/environments/v1.10/Project.toml`
 No Changes to `~/.julia/environments/v1.10/Manifest.toml`
```

[22]:
```
# exportiert @turbo für AVX
using LoopVectorization

@inline function avx_matmul!(A, B, C)
    C .= 0
    @turbo for j in 1:size(B)[2]
        for k in 1:size(A)[2]
            for i in 1:size(A)[1]
                @inbounds C[i, j] += A[i, k] * B[k, j]
            end
        end
    end
end

@btime avx_matmul!(X, Y, Z)
```

```
  41.967 ms (0 allocations: 0 bytes)
```

Der Rest ist hauptsächlich - memory modelling - Cache/blocking inlining - packing, padding und stark von der Hardware abhängig.

Natürlich ist das extrem abhängig von Matrixgröße und LoopVectorization besser, wenn caching keine große Rolle spielt. Siehe auch hier und Docs.

Hier beispielsweise ein interessanter Thread, wie im Paket Octavian.jl Matrixmultiplikation optimiert wird.

Kleinere Dimensionen

[23]:
```
using LinearAlgebra

BLAS.get_config()
```

[23]:
```
LinearAlgebra.BLAS.LBTConfig
Libraries:
└ [ILP64] libopenblas64_.dylib
```

[24]:
```
function smarter_avx_matmul!(A, B, C)
    # die Umordnung des Loops wird automatisch gemacht!
    @turbo for i ∈ 1:size(A,1), j ∈ 1:size(B,2)
        # so ist es sogar noch ein bisschen schneller, vermutlich weil dieses
 ↪Statement besser parallelisiert werden kann
        C[i,j] = 0.0
        for k ∈ 1:size(A,2)
            C[i,j] += A[i,k] * B[k,j]
        end
    end
end
```

[24]:
```
smarter_avx_matmul! (generic function with 1 method)
```

[25]:
```
m = 100; n = 100; k = 100
X = rand(m, k)
Y = rand(k, n)
Z = zeros(m, n)

@btime smarter_avx_matmul!(X, Y, Z)
```

```
  41.291 µs (0 allocations: 0 bytes)
```

[26]:
```
@btime Z = X * Y
```

```
  38.000 µs (2 allocations: 78.17 KiB)
```

[26]:
```
100×100 Matrix{Float64}:
 23.3285  22.9181  27.5102  22.052   …  27.5316  26.2614  28.3475  26.5794
```

```
21.5445  20.6621  21.9968  20.4913     23.8148  23.2619  22.8579  23.1033
25.4996  23.9935  25.2988  22.9159     27.0579  25.1413  25.7109  25.4037
27.1252  25.6127  27.9762  25.0869     29.523   29.237   30.0486  27.9906
26.7188  24.4995  27.061   21.9922     28.2783  28.097   27.843   28.7584
22.2857  22.0596  23.7147  23.5421  …  26.2706  23.3847  24.8675  23.4296
23.2242  22.1748  24.944   21.2667     26.4045  23.1604  25.2124  24.3827
23.4167  21.903   23.7289  22.3581     25.7641  23.8335  24.747   23.8166
24.8857  23.2759  25.0829  21.9281     27.3472  25.5462  27.4597  25.0575
24.4988  21.7311  26.4867  23.0033     27.4105  26.018   27.0624  26.307
23.6456  24.1284  24.6378  22.3722  …  26.1557  24.9388  26.082   26.3098
26.2925  23.3098  26.0149  24.6596     28.9955  28.3965  29.1296  27.9723
21.861   19.0739  23.5684  21.0009     24.9294  22.084   24.2481  23.3009
  ⋮                                  ⋱
23.5194  23.6039  25.6492  21.824      24.6497  26.0128  26.9081  24.7958
23.7     23.2718  24.9494  23.7495     26.5935  26.0364  26.6067  26.517
23.4372  19.7556  24.516   19.7108  …  22.1659  23.3877  21.6459  24.9734
27.4031  24.1954  28.101   25.1837     28.9254  27.4451  28.5568  28.2805
21.6251  21.8262  23.5298  20.9646     24.7957  23.7163  24.5383  23.7978
24.6987  23.4974  23.7239  22.8325     26.4692  25.1596  25.7651  25.7688
24.5221  24.6001  24.9501  22.2902     26.6135  25.4364  26.5226  26.2589
22.8736  23.1459  23.6516  21.4354  …  27.9184  25.5021  27.2321  25.1332
24.6395  22.4307  24.9295  23.6915     26.843   25.4137  25.6201  25.6009
27.5125  24.7291  27.2677  23.984      27.677   27.7725  28.0995  27.0845
23.872   23.4015  25.7657  22.0744     25.8626  26.0151  26.4485  24.6041
24.2483  23.565   23.8459  22.2664     27.6957  25.0028  26.2527  24.3264
```

Weiter Tipps findet man in der Dokumentation.

9.2.4 Kontextabhängige Optimierung

Eine Sache, die man nicht vergessen darf, ist: Code ist oftmals nur für eine konkrete, spezielle Anwendung optimal. Kommen wir beispielsweise zu unseren Zufallszahlen von vorhin zurück. Da ist die Effizienz zum Teil abhängig davon, ob wir *oft* aus derselben Verteilung samplen, weil die Initialisierung des Samplers aufwendig sein kann.

[27]:
```
# sample from categorical distribution
function categorical(probabilities)
    u = rand()
    sum = 0.0
    for i in eachindex(probabilities)
        sum += probabilities[i]
        if u <= sum
            return i
```

```
        end
    end
end
@btime categorical([0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.
↪9])
```

```
  22.024 ns (1 allocation: 144 bytes)
```

[27]: 11

Unser naiver Code ist langsamer als der Sampler aus Distributions.jl:

[28]:
```
dist_often = Categorical([0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.
↪01, 0.9])
@btime rand(dist_often)
```

```
  20.353 ns (0 allocations: 0 bytes)
```

[28]: 11

Dieser hat aber eine ziemlich schlechte Performance, wenn wir ihn in jeder Iteration neu definieren müssen:

[29]:
```
@btime begin
    dist_often = Categorical([0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.
↪01, 0.01, 0.9])
    rand(dist_often)
end
```

```
  59.554 ns (2 allocations: 288 bytes)
```

[29]: 11

Kurs 10 (Bonusvorlesung)

10.1 Eine Meta-Diskussion über Julia

Nachdem wir mittlerweile einen Überblick darüber gewonnen haben, *wie* Julia ungefähr aufgebaut ist, soll nun noch etwas Intuition dazu geben *warum* Julia eigentlich so funktioniert. Vor allem möchten wir klären, inwiefern und warum sich Julia von herkömmlichen objektorientierten Programmiersprachen unterscheidet.

Generell kann man sagen: In der Theorie funktionieren zwar vielerlei Konzepte, die intuitiv und nahe an der Realität dessen, was man modellieren bzw. wovon man abstrahieren will, sind. Aber: In der Praxis geht es fast immer darum, wie man logische Relationen zwischen Objekten mit Dingen wie Performance, Übersichtlichkeit und Codeorganisation in Einklang bringt.

10.1.1 Was sollte eine Programmiersprache können?

Eine einigermaßen benutzerfreundliche Programmiersprache sollte ein paar Dinge können. Dazu gehört beispielsweise so etwas wie *Polymorphismus*, das bedeutet, dass man für verschiedene Entitäten unterschiedlichen Typs *das gleiche Interface* (Schnittstelle) bereitsstellt. Das multiple dispatch von Julia ist beispielsweise eine Form von Polymorphismus; mit Schnittstelle meint hier einfach eine Funktion, die Entitäten sind verschiedene Variablen. Warum will man das? Naja, überlegen wir mal was wäre, wenn wir anstelle des Operators `+` nur spezialisierte Operatoren `+_Float32`, `+_Float64`, `+_Int32` hätten – das wäre schon ziemlich unpraktisch.

Des Weiteren möchte man in der Lage sein, die Eigenschaften und das Verhalten bereits definierter Strukturen an neue Strukturen weiterzugeben (code reuse). Zuletzt (ohne Anspruch auf Vollständigkeit) möchte man sogenannte *encapsulation*, das heißt Datenbündelung bzw. die Einschränkung von Zugriff auf interne Komponenten mancher Strukturen. Schauen wir uns nun an, wie diese abstrakten Konzepte in anderen (objektorientierten) Programmiersprachen umgesetzt sind.

10.1.2 Klassische Objektorientierung: Basics

Klassen

Das Pendant zu Julias Typen sind dort sogenannte Klassen. Eine Instanz einer Klasse (also eine konkrete Realisierung) nennt man Objekt – daher der Begriff objektorientierte Programmierung (OOP). Im OOP-Ansatz beinhalten Objekte neben normalen Datenfeldern üblicherweise Funktionen, die auf sich selbst operieren. Diese nennt man Methoden. In Pseudocode sieht dann also sowas wie

[1]:
```
struct Foo
    a
end
example_function(x) = print(x.a)
```

[1]:
```
example_function (generic function with 1 method)
```

mit Aufruf

[2]:
```
foo = Foo(1)
example_function(foo)
```

```
1
```

stattdessen so aus:

```
class Bar:
    function __init__(self, a) # Konstruktor
        self.a = a
    end
    example_function(self) = print(self.a) # Methode
```

mit Aufruf

```
bar = Bar(1)
bar.example_function() # Methode lebt innerhalb des Objekts
```

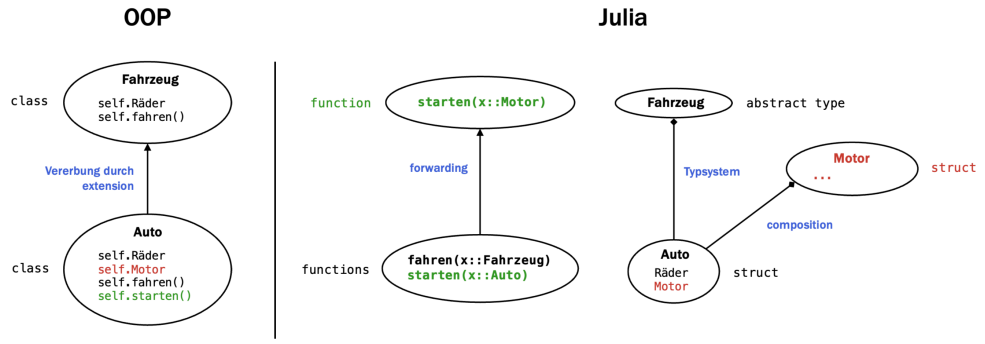
Außerdem kann man meistens festlegen, ob Attribute und Methoden *private* oder *public* sind. Auf Privates kann dann außerhalb der Klassendefinition nicht mehr zugegriffen werden (das ist gerade *encapsulation*.)

Vererbung

Der Clou an der ganzen Sache ist nun, dass eine Basisklasse ihre Eigenschaften durch Vererbung an neue Klassen weitergeben kann. So kann zum Beispiel die Klasse `Fahrzeug` an die Klasse `Auto` vererben (siehe links in der folgenden Grafik[1]). Eigentlich gibt es verschiedene Arten der Vererbung, wir betrachten hier oBdA Vererbung durch extension.

[1] (Achtung: Die Pfeile sind gemäß dem UML-Standard und gehen Richtung Generalisierung.)

Das soll heißen: Eine Klasse wird um neue Fähigkeiten erweitert; im Beispiel überträgt sich die Funktion `self.fahren` (das ist Polymorphismus). Vererbung beschreibt daher eine ist-ein-Beziehung (ein Auto ist ein Fahrzeug).



10.1.3 Typ-Hierarchie

In Julias Typ-Hierarchie sind Typen nach ihrem *Verhalten* sortiert (behavioral inheritance; wiederum Polymorphismus). Das bedeutet insbesondere, dass wir von konkreten Details der Typenimplementierung abstrahieren und uns nur interessiert, welche Operationen wir mit einem bestimmten Typ anstellen können.

Im nachfolgenden Beispiel sei eine Person dadurch definiert, dass sie einen Namen hat. Außerdem können Lehrer eine Note vergeben und StudentInnen können eine Note bekommen. Eine Besonderheit sind hier MusikstudentInnen: Weil diese (wie ja jeder weiß) gerne Mozart wären, nennen sie als (Künstler-)Namen stets ebendiesen. Die Funktionen `give_grade` und `get_grade` sind für uns nicht weiter spannend, sondern dienen nur dazu, die Systematik einer verhaltensbasierten Hierarchie zu zeigen.

[3]:
```
abstract type AbstractPerson end
abstract type AbstractStudent <: AbstractPerson end
abstract type AbstractTeacher <: AbstractPerson end

struct Person <: AbstractPerson
  name::String
end

struct Student <: AbstractStudent
  name::String
  grade::Int
  hobby::String
end

struct MusicStudent <: AbstractStudent
```

```
  grade::Int
end

struct Teacher <: AbstractTeacher
  name::String
end
```

[4]:
```
say_name(x::AbstractPerson) = x.name
say_name(x::MusicStudent) = "I am Wolfgang Amadeus Mozart!!"

give_grade(x::Teacher) = 3.0
get_grade(x::Student) = 3.0
```

[4]:
```
get_grade (generic function with 1 method)
```

10.1.4 Warum wollen wir keine Vererbung?

In objektorientierten Programmiersprachen sind Objekte nach ihrer konkreten Implementierungsstruktur sortiert (structural inheritance). Dass dabei die Attribute, also die Datenfelder in einem Objekt, vererbt werden, ist aber im Kontext von performantem Code nicht unbedingt ein guter Ansatz. Denn oftmals sind manche Daten für manche Funktionalitäten redundant, wodurch man sich overhead (unnötigen Aufwand) einhandelt.

Im obigen Beispiel haben wir konkret schon gesehen, dass etwa der Musikstudent das Datenfeld `name` gar nicht braucht. Bei einer Vererbung würde er jenes aber von `Student` bekommen. Weil dieses Beispiel aber natürlich ein bisschen artifiziell ist, folgt nun noch eine realitätsnahe Situation.

Bespiel: AbstractArray

Betrachten wir eine range `a`, dann ist diese ein Array.

[5]:
```
a = 1:10
a isa AbstractArray
```

[5]:
```
true
```

Allerdings hat `a` nur die Felder `start` und `stop`. Ergo: Dort liegen nicht wie üblich 10 Zahlen rum, stattdessen wird davon abstrahiert.

[6]:
```
fieldnames(typeof(a))
```

[6]:
```
(:start, :stop)
```

Dies ist sinnvoll, denn das Allozieren langer Arrays kostet Zeit (und zwar $\mathcal{O}(n)$ vs. $\mathcal{O}(1)$) sowie Speicherplatz und ist für unsere Zwecke unnötig.

[7]:
```
using BenchmarkTools
@btime 1:100000;
@btime collect(1:100000);
```

```
  0.833 ns (0 allocations: 0 bytes)

  15.292 μs (2 allocations: 781.30 KiB)
```

Hätten wir stattdessen etwa einen Array `a = [1.3, 4.7, 0.6]` mit irgendwelchen Datenpunkten, dann könnten wir natürlich nicht nur Anfang und Ende abspeichern. Am Ende des Tages wollen wir also Typen, die gleiches Verhalten haben (nämlich indexing, also `A[i]`, ist wohldefiniert). Das nennt man auch *duck typing* (if it quacks like a duck, it might as well be a duck). Umgekehrt wollen wir per se aber eben nicht gleiche Datenfelder, weil das die Performance stören kann.

Ehrlicherweise muss man sagen, dass Vererbung noch aus vielerlei anderen Gründen problematisch sein kann, aber das offensichtliche Problem von Performance ist für unsere Zwecke schon hinreichend blöd.

Composition over inheritance

Nicht nur, aber insbesondere aus den obigen Gründen wird auch im OOP-Paradigma in der Regel gerade nicht vererbt, sondern bevorzugt werden Objekte aus kleineren Objekten zusammengesetzt. Demnach erbt das Objekt Auto beispielsweise nicht von Fahrzeug, sondern wird einfach aus Rädern, Motor, usw. zusammengesteckt (composition). Im Gegensatz zur Vererbung hat man hier also keine ist-ein-, sondern eine hat-ein-Beziehung (ein Auto hat einen Motor). Dieses Prinzip bzw. Vorgehen nennt man *composition over inheritance* (siehe rechte Seite in der oberen Grafik).

Erwähnenswert ist an dieser Stelle noch das sogenannte *function forwarding*. Das ist eine ziemliche simple Sache: Wenn wir ein Auto aus Einzelteilen zusammensetzen, dann wäre es schön, wenn sich die Funktionen des Hupe auf das Auto übertragen würden – mit dem Auto zu hupen ist ja gerade dasselbe wie mit der (Auto-)Hupe zu hupen:

[8]:
```
struct Horn
    sound::String
end

toot_twice(h::Horn) = h.sound^2
toot_loud(h::Horn) = uppercase(string(h.sound, "!"))
```

[8]:
```
toot_loud (generic function with 1 method)
```

[9]:
```
# brauchen wir später noch
abstract type AbstractCar end

# wir setzen das Auto zusammen – composition!
struct Car <: AbstractCar
    räder
    horn::Horn
    # inner constructor
    Car(sound) = new(4, Horn(sound))
end
```

[10]:
```
# forwarding
for method in (:toot_twice, :toot_loud)
    @eval $method(c::Car) = $method(c.horn)
end

# Das macht das gleiche wie:
# double(wif::Car) = double(wif.interesting)
# shout(wif::Car) = shout(wif.interesting)

benzer = Car("tröt")
```

[10]:
```
Car(4, Horn("tröt"))
```

[11]:
```
toot_twice(benzer)
```

[11]:
```
"trötttröt"
```

[12]:
```
toot_loud(benzer)
```

[12]:
```
"TRÖT!"
```

Wie immer gibt es auch hier Pakete, die uns die Arbeit erleichtern und uns noch etwas mächtigere Makros an die Hand geben (zum Beispiel Lazy.jl).

Warum nicht Objekte anstelle von Typen?

Die Frage danach, warum man also Vererbung eigentlich sowieso nicht haben möchte, ist nun geklärt. Trotzdem bleibt die Frage danach, weshalb genau man Funktionen *außerhalb* eines Typs und nicht trotzdem wie bei der OOP *innerhalb* eines Objekts hat.

Darauf habe ich keine klare Antwort, aber zunächst gilt, dass Funktionen ja oft *vor* Typen leben. Beispielsweise kann ich einen neuen Typ und darauf den Operator `+` definieren; die Funktion `x -> x + x` wird dann darauf auch operieren können, obwohl sie nie manuell angepasst wurde. Es macht also nicht wirklich Sinn, dieses deutlich allgemeinere

Konzept zumindest teilweise wieder in eine Kiste namens Typ zu packen – die Sicherheit, dass die Funktion wirklich nur für den richtigen Typ ausgeführt wird bekommt man sowieso durch type declarations. Und darüber hinaus erreicht man encapsulation für Funktionen sehr einfach durch Module.

Quasi-Vererbung

Wenn aber trotzdem Datenfelder vererben möchte (im Sinne von: man spart sich Arbeit, weil man nichts doppelt schreiben muss), weil man vielleicht in einem bestimmten Kontext keine composition haben möchte, dann geht das mithilfe von Makros auch (siehe auch aktuelle Paketentwicklungen).

Traits

Vorneweg eine kleine Nebenbemerkung: In vielen Sprachen gibt es sogenannte *design patterns*. Damit bezeichnet man im Grunde genommen häufig vorkommende Tricks bzw. Muster, die man für bestimmte Implementierungszwecke nutzt. Sogenannte *traits* sind zum Beispiel in der Sprache Rust ein *language feature*. Das bedeutet, sie sind eine ab Werk verfügbare Funktionalität. In Julia sind traits dagegen ein design pattern, welches auch als THTT (Tim Holy Trait Trick) bekannt und nach dem Entdecker Tim Holy benannt ist.

Das Konzept eines traits lässt sich auf verschiedene Weisen betrachten:

- Wir wollen das gleiche Verhalten für verschiedene Typen, welche ansonsten eigentlich nicht verwandt sind bzw. etwas abstraker formuliert
- Wir wollen compile-time information über Typen, diese soll aber nicht über die Typenhierarchie festgelegt werden.
- Traits sind Mehrfachvererbung für Typsysteme.
- Wir geben dem Compiler ein (unverbindliches) Versprechen, dass ein bestimmter Typ die richtigen interfaces implementiert und daher als Input für jegliche andere Funktion verwendet werden kann, die dieses Interface benötigt (design contract).

Das klingt wahrscheinlich alles erstmal ziemlich kryptisch und wird vielleicht an folgendem Beispiel klarer:

[13]:
```
struct MusicTeacher <: AbstractTeacher
    name::String
end
```

Nun haben wir `MusicStudent` und `MusicTeacher`, die beide Musik mögen, aber ansonsten nicht viel gemeinsam haben. Naheliegenderweise soll hier der Wert des Audioequipments der Musikliebhaber höher sein als der einer gewöhnlichen Person:

[14]:
```
const Audiophile = Union{MusicStudent, MusicTeacher}

function audio_equipment_value(x::AbstractPerson) # zum Beispiel in €
    if x isa Audiophile
        return 200
    else
        return 100
    end
end
```

[14]:
```
audio_equipment_value (generic function with 1 method)
```

Um die Musikliebhaber zusammenzufassen, haben wir uns eine `Union` definiert und bekommen dadurch das gewünschte Verhalten.

[15]:
```
ada = Student("Ada Lovelace", 1.0, "Programming")
julia = MusicStudent(2.0)
```

[15]:
```
MusicStudent(2)
```

[16]:
```
audio_equipment_value(ada)
```

[16]:
```
100
```

[17]:
```
audio_equipment_value(julia)
```

[17]:
```
200
```

Diese Lösung funktioniert zwar, allerdings ist sie sehr unschön. Denn `Union{MusicStudent, MusicTeacher}` muss jedes Mal geupdated werden, wenn wir einen neuen audiophilen Typ wie etwa `SongWriter` definieren. Zudem sind Unions unter Umständen inperformant. Deshalb gehen wir die Sache lieber via traits an:

[18]:
```
abstract type MusicStyle end
struct MusicLover end # empty struct (singleton type)
struct NonMusicLover end # empty struct (singleton type)

# Aus der Doku: When a type is applied like a function it is called a constructor
# Wir könnten theoretisch auch eine neue Funktion anstelle von MusicStyle
# definieren, aber so ist es relativ elegant.
MusicStyle(::Any) = NonMusicLover() # default behaviour
MusicStyle(::MusicTeacher) = MusicLover() # MusicTeacher hat trait
MusicStyle(::MusicStudent) = MusicLover() # MusicStudent hat trait
```

[18]:
```
MusicStyle
```

Analog zu `audio_equipment_value` betrachten wir nun die Funktion `sing`, die sich nach dem trait (also der Eigenschaft) `MusicStyle` unterscheidet.

[19]:
```
sing(x::AbstractPerson) = sing(MusicStyle(x), x) # Trick!!!
sing(::NonMusicLover, x) = error("$(say_name(x)) does not want to sing.")
sing(::MusicLover, x) = println("La La La ♫")
```

[19]:
```
sing (generic function with 3 methods)
```

[20]:
```
sing(ada)
```

```
Ada Lovelace does not want to sing.

Stacktrace:
 [1] error(s::String)
   @ Base ./error.jl:35
 [2] sing(::NonMusicLover, x::Student)
   @ Main ./In[19]:2
 [3] sing(x::Student)
   @ Main ./In[19]:1
 [4] top-level scope
   @ In[20]:1
```

[21]:
```
sing(julia)
```

```
La La La ♫
```

In dieser einfachen Variante ist ein trait also lediglich der Datentyp `MusicStyle`. Weil unser trait ja sogar binär ist (mag/mag nicht), könnte man sich diese zwei singleton types auch sparen und direkt eine Funktion definieren, die entsprechend `true`/`false` zurückgibt und dann für `if`/`else` genutzt wird:

[22]:
```
likes_music(::AbstractPerson) = false
likes_music(::MusicStudent) = true
likes_music(::MusicTeacher) = true
```

[22]:
```
likes_music (generic function with 3 methods)
```

[23]:
```
function listening_minutes(x::AbstractPerson)
    if !likes_music(x)
      return 0
    end
    num_records = rand(1:10)
    if typeof(x) <: AbstractStudent
```

```
    return 2num_records
  else
    return num_records
  end
end
```

[23]:
```
listening_minutes (generic function with 1 method)
```

Wichtig ist: Auch hier haben wir keine erhöhte Laufzeit, denn die verschiedenen Branches (Abzweigungen) werden einfach rauskompiliert!

[24]:
```
x = Teacher("Gilbert Strang")
listening_minutes(x)
```

[24]:
```
0
```

[25]:
```
@code_typed listening_minutes(x) # nur noch return 0
```

[25]:
```
CodeInfo(
1 —     return 0
) => Int64
```

[26]:
```
x = MusicStudent(2.0)
@code_typed listening_minutes(x) # nur noch return 2num_records
```

[26]:
```
CodeInfo(
1 —      nothing::Nothing
│   %2 = invoke
Random.rand($(QuoteNode(Random.TaskLocalRNG()))::Random.TaskLocalRNG,
$(QuoteNode(Random.SamplerRangeNDL{UInt64, Int64}(1,
0x000000000000000a)))::Random.SamplerRangeNDL{UInt64, Int64})::Int64
│   %3 = Base.mul_int(2, %2)::Int64
└──      return %3
) => Int64
```

[27]:
```
x = MusicTeacher("Clara Schumann")
@code_typed listening_minutes(x) # nur noch return num_records
```

[27]:
```
CodeInfo(
1 —      nothing::Nothing
│   %2 = invoke
Random.rand($(QuoteNode(Random.TaskLocalRNG()))::Random.TaskLocalRNG,
$(QuoteNode(Random.SamplerRangeNDL{UInt64, Int64}(1,
0x000000000000000a)))::Random.SamplerRangeNDL{UInt64, Int64})::Int64
└──      return %2
```

```
) => Int64
```

Pakete wie SimpleTraits.jl oder WhereTraits.jl vereinfachen die Implementierung von Traits. Tatsächlich gibt es wohl aber keinen technischen Grund, warum dispatch auf Traits nicht auch ab Werk und mit vereinfachter Syntax in Base Julia eingebaut werden könnten (dispatch: „The choice of which method to execute when a function is applied is called dispatch.“).

In welchem Verhältnis stehen aber traits zur Vererbung? Tatsächlich kann man traits als bessere Alternative zur Mehrfachvererbung sehen (zwei oder mehrere *parent*-Klassen haben eine *child*-Klasse). Diese ist nämlich konzeptuell problematisch, da relativ schnell Ambiguitätsprobleme auftreten (siehe etwa diamond problem).

10.1.5 Mögliche Probleme und Verbesserungsmöglichkeiten

Keine privaten Felder

Aktuell kann man in Julia Datenfelder nicht schützen, das heißt als privat markieren. Funktional ändert das zwar gar nichts, aber manchmal wäre es halt schon praktisch, inkompente User vor dem Herumpfuschen in Datenstrukturen zu schützen. Nehmen wir mal an, wir hätten den Typ `Motor`, dann sollte der User in aller Regel die Funktion `starten` nutzen und nicht manuell am Datenfeld `Motor.Einspritzdüse` rumschrauben. Man kann das zwar verhinden, indem man die `getproperty`-Funktion eines Typs überlädt, aber so richtig toll ist das auch nicht.

Kein contract enforcing

Sowohl bei Vererbung, als auch bei traits, geht es darum, was ein Ding *kann* aka welche Interfaces es implementiert. Zum Beispiel wäre es vielleicht sinnvoll vorauszusetzen, dass ein `AbstractCar` das Interface `move!` implementiert (ein Auto muss fahren können).

[28]:
```
# Beispielmethode, die von move! abhängt
park!(c::AbstractCar) = move!(c, :nearest_parking_lot)
```

[28]:
```
park! (generic function with 1 method)
```

Wenn nun aber ein User vergisst, die `move!`-Funktion zu implementieren, dann bekommen wir *erst zur Laufzeit* einen Fehler!

[29]:
```
park!(benzer)
```

```
UndefVarError: `move!` not defined

Stacktrace:
 [1] park!(c::Car)
```

```
  @ Main ./In[28]:2
[2] top-level scope
  @ In[29]:1
```

Ein anderes Beispiel wäre die Implementierung eines Arrays (siehe Übungsblatt 7, Zusatzaufgabe), wo man erst beim Aufrufen des Konstruktors einen Fehler erhält, falls die Funktion bzw. das Interface `Base.size` nicht implementiert wird (hier ein Überblick über die wichtigsten Standardinterfaces von Julia).

Diese Tatsache ist problematisch, da so unter anderem

- lange Laufzeit den Entwicklungsprozess erschwert
- Fehler in der Regel nicht durch Linter (type checks) gefunden werden.

In der Praxis hat man also einfach nur docstrings (also Text-Dokumentation), die angeben, welche Methoden auf welche Weise implementiert werden sollten. Man kann sich zwar Gedanken machen, wie man damit möglichst schlau umgeht (zum Beispiel gewisse Tests), aber leider hat man trotzdem nicht dieselbe Zuverlässigkeit wie bei statisch typisierten Programmiersprachen. Pakete wie Interfaces.jl versuchen dafür Lösungen zu finden.

[30]:
```
# Folgendes ist ein docstring (das ist das, was wir sehen, wenn wir ?move! eingeben)
"""
    move!(
        c::AbstractCar,
        location
    ) -> nothing
"""
function move! end
```

[30]:
```
move!
```

10.1.6 Weiterführende Resourcen

Hier sind einige Links, die zum Erstellen dieses Kapitels genutzt wurden. Dort finden sich auch noch weitere Beispiele und Erklärungen (falls jemand wahnsinnig motiviert sein sollte…).

- Dokumentation zu traits, man wird daraus aber nicht wirklich schlau, finde ich
- Das Buch Hands-on design patterns and best practices with Julia: proven solutions to common problems in software design for Julia 1.x von Tom Kwong
- Christopher Rackauckas' Blogpost Type-Dispatch Design: Post Object-Oriented Programming for Julia
- GitHub repositories Object Orientation and Polymorphism in Julia und Dispatching Design Patterns von Aaron Christianson

- Erklärungen in den Paketen SimpleTraits.jl und Traits.jl (deprecated) von Mauro Werder
- Der oft genannte THTT im Originalissue
- Pakete WhereTraits.jl, CanonicalTraits.jl, BinaryTraits.jl
- Dokumentation der Design Patterns des Pakets ADCME.jl von Kailai Xu und Eric Darve
- Das repository patterns: Object-oriented design pattern examples in Julia von Yueh-Hua Tu
- Überlegungen von Harrison Grodin zur Weiterentwicklung von Julia, vor allem der Abschnitt zu Traits
- Ganz neu diese Vergleichsgrafik der verschiedenen Trait-Pakete