Julia für Numerik

Einführung in die wissenschaftliche Programmierung

Meik Hellmund

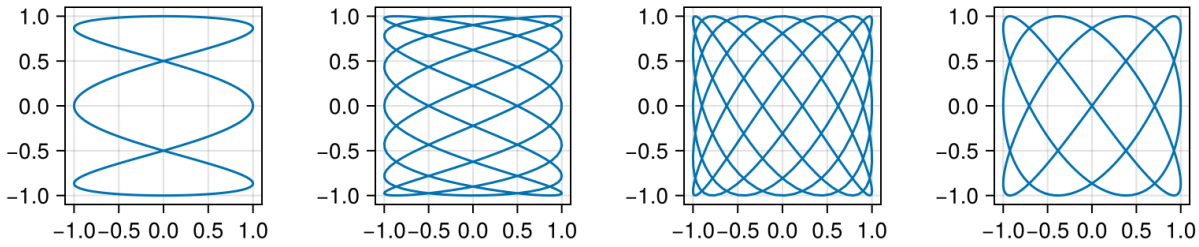
2024-04-22

Was ist Julia?

Julia ist eine noch recht junge für *scientific computing* konzipierte moderne Programmiersprache.

Ein kleines Codebeispiel:

```
using CairoMakie
a = [3, 7, 5, 3]
b = [1, 3, 7, 4]
δ = π/2
t = LinRange(-π, π, 300)
f = Figure(size=(800, 180))
for i in 1:4
    x = sin.( a[i] .* t .+ δ )
    y = sin.( b[i] .* t )
    lines(f[1, i], x, y, axis=(; aspect = 1))
end
f
```



Geschichte

- 2009 Beginn der Entwicklung am *Computer Science and Artificial Intelligence Laboratory* des MIT
- 2012 erste release v0.1
- 2018 Version v1.0
- aktuell: v1.10.2 vom 1. März 2024

Zum ersten release 2012 haben die Schöpfer von Julia ihre Ziele und Motivation in dem Blogbeitrag Why we created Julia interessant zusammengefasst.

Für ein Bild von *Stefan Karpinski, Viral Shah, Jeff Bezanson* und *Alan Edelman* bitte hier klicken: https://news.mit.edu/2018/julia-language-co-creators-win-james-wilkinson-prize-numerical-software-1226.

Warum Julia?

aus The fast track to Julia

"Julia is an open-source, multi-platform, high-level, high-performance programming language for technical computing. Julia has an LLVM-based JIT compiler that allows it to match the performance of languages such as C and FORTRAN without the hassle of low-level code. Because the code is compiled on the fly you can run (bits of) code in a shell or REPL , which is part of the recommended workflow .

> Julia is dynamically typed, provides multiple dispatch, and is designed for parallelism and distributed computation. Julia has a built-in package manager."

open source

- offene Entwicklung auf GitHub
- Implementierungen für alle gängigen Betriebssysteme

high-performance programming language for technical computing

- viele Funktionen für *scientific computing* eingebaut,
- (bewusste) Ähnlichkeit zu Python, R und Matlab,
- komplexe Berechnungen in wenigen Zeilen
- einfaches Interface zu anderen Sprachen wie C oder Python

a JIT compiler

- interaktives Arbeiten möglich: `read-eval-print loop (REPL)` mit
- just-in-time (JIT) Compilation
- dadurch Laufzeiten vergleichbar mit statischen Sprachen wie C/C++, Fortran oder Rust

a built-in package manager

- riesiges *ecosystem* an einfach installierbaren Paketen, z.B.
  - Mathematische Optimierung
  - Machine Learning
  - Data Visualization
  - Differentialgleichungen
  - Mathematische Modellierung

Eine kleine Auswahl an Online-Material zu Julia

- Dokumentation - die offizielle Dokumentation
- Cheat Sheet - "a quick & dirty overview"
- Introducing Julia– ein WikiBook
- The Julia Express - Kurzfassung, Julia auf 16 Seiten
- Think Julia - Einführung in die Programmierung mit Julia als Sprache
- Das Julia Forum
- Was fürs Auge: Beispiele zum Julia-Grafikpaket `Makie`

1 Entwicklungsumgebungen

Für diesen Kurs werden wir die webbasierte Entwicklungsumgebung Jupyterhub verwenden. Sie steht allen Teilnehmerïnnen für die Dauer des Kurses zur Verfügung.

Für langfristiges Arbeiten empfiehlt sich eine eigene Installation.

1.1 Installation auf eigenem Rechner (Linux/MacOS/MS Windows)

1. Julia mit dem Installations- und Update-Manager **juliaup** installieren: https://github.com/JuliaLang/juliaup/
2. als Editor/IDE **Visual Studio Code** installieren: https://code.visualstudio.com/
3. im VS Code Editor die **Julia language extension** installieren: https://www.julia-vscode.org/docs/stable/gettingstarted/

Einstieg:

- In VS Code eine neue Datei mit der Endung `.jl` anlegen
- Julia-Code schreiben
- `Shift-Enter` oder `Ctrl-Enter` am Ende einer Anweisung oder eines Blocks startet eine Julia-REPL, Code wird in die REPL kopiert und ausgeführt
- Tastenbelegungen für VS Code und für Julia in VS Code

1.1.1 Die Julia-REPL

Wenn man Julia direkt startet, wird die Julia-REPL (*read-eval-print* Schleife) gestartet, in der man interaktiv mit Julia arbeiten kann.

```
$ julia
               _
   _       _ _(_)_     |  Documentation: https://docs.julialang.org
  (_)     | (_) (_)    |
   _ _   _| |_  __ _   |  Type "?" for help, "]?" for Pkg help.
  | | | | | | |/ _` |  |
  | | |_| | | | (_| |  |  Version 1.10.2 (2024-03-01)
 _/ |\__'_|_|_|\__'_|  |  Official https://julialang.org/ release
|__/                   |

julia>
```

1.2 Arbeiten auf dem Jupyterhub-Webserver

1.2.1 Jupyterhub & Jupyter

- Jupyterhub ist ein Multi-User-Server für Jupyter.
- Jupyter ist eine web-basierte interaktive Programmierumgebung, die
- ursprünglich in und für Python geschrieben, inzwischen eine Vielzahl von Programmiersprachen nutzen kann.
- In Jupyter bearbeitet man sogenannte *notebooks*. Das sind strukturiere Textdateien (JSON), erkennbar an der Dateiendung `*.ipynb`.

Unser Jupyterhub-Server: https://misun103.mathematik.uni-leipzig.de/

Nach dem Einloggen in Jupyterhub erscheint ein Dateimanager:

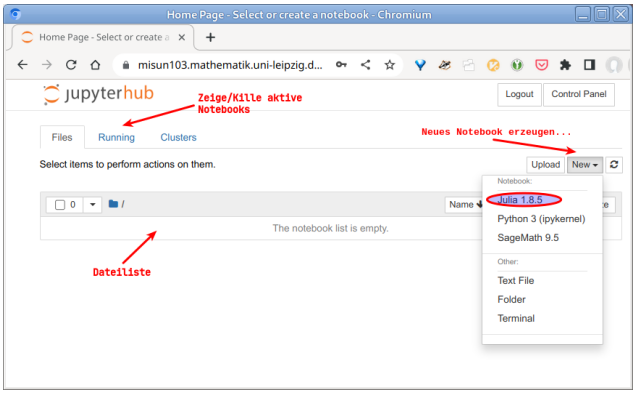


Abbildung 1.1: Jupyterhub Dateimanager

Mit diesem kann man:

- vorhandene *notebooks* öffnen,
- neue *notebooks* anlegen,
- Dateien, z.B. *notebooks*, vom lokalen Rechner hochladen,
- die Sitzung mit `Logout` beenden (bitte nicht vergessen!).

1.2.2 Jupyter notebooks

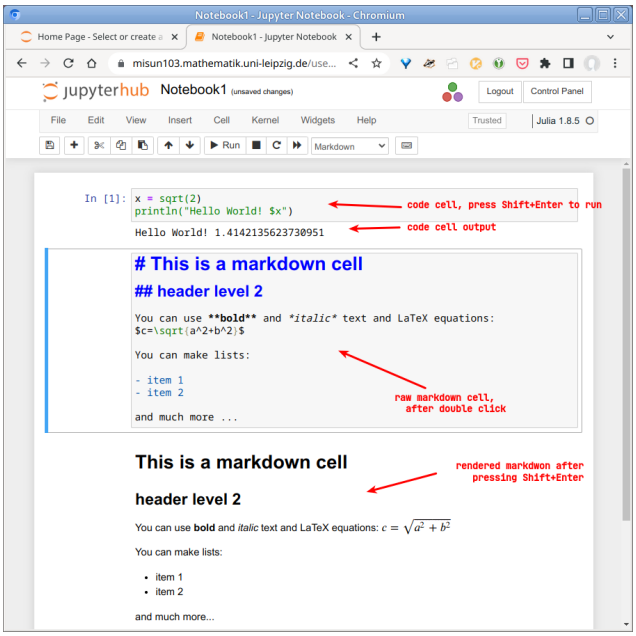


Abbildung 1.2: Jupyter Notebook

Notebooks bestehen aus Zellen. Zellen können

- Code oder
- Text/Dokumentation (Markdown)

enthalten. In Textzellen kann die Auszeichnungssprache Markdown zur Formatierung und LaTeX für mathematische Gleichungen verwendet werden.

> **Tipp**
>
> Bitte die Punkte `User Interface Tour` und `Keyboard Shortcuts` im `Help`-Menü anschauen!

Die Zelle, die man gerade bearbeitet, kann im `command mode` oder `edit mode` sein.

	Command mode	*Edit mode*
Mode aktivieren	`ESC`	Doppelklick oder `Enter` in Zelle
neue Zelle	`b`	`Alt-Enter`
Zelle löschen	`dd`	
Notebook speichern	`s`	`Ctrl-s`
Notebook umbenennen	`Menu -> File -> Rename`	`Menu -> File -> Rename`
Notebook schließen	`Menu -> File -> Close & Halt`	`Menu -> File -> Close & Halt`
run cell	`Ctrl-Enter`	`Ctrl-Enter`
run cell, move to next cell	`Shift-Enter`	`Shift-Enter`
run cell, insert new cell below	`Alt-Enter`	`Alt-Enter`

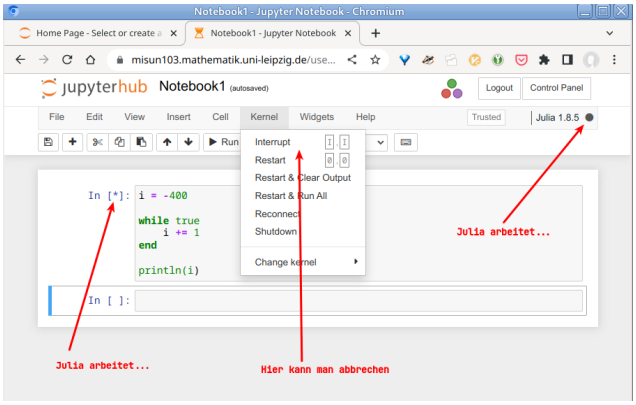


Abbildung 1.3: Julia arbeitet...

Wenn eine Zelle "arbeitet", wird ihre Zellen-Nummer zu einem `*` und die `kernel busy`-Anzeige (Voller schwarzer Punkt oben rechts neben der Julia-Version) erscheint. Falls das zu lange dauert (ein *infinite loop* ist schnell programmiert), dann

- `Menu -> Kernel -> Interrupt` anklicken; falls das wirkungslos ist,
- `Menu -> Kernel -> Restart` anklicken.

> **Wichtig**
>
> Nach einem `kernel restart` müssen alle Zellen, die weiterhin benötigte Definitionen, `using`-Anweisungen etc enthalten, wieder ausgeführt werden.
> **Am Ende der Arbeit bitte immer**:
>
> - `Menu -> File -> Save & Checkpoint`
> - `Menu -> File -> Close & Halt`
> - `Logout`

2 First Contact

Dieses Kapitel soll beim 'Loslegen' helfen. Es läßt viele Details weg und die Codebeispiele sind oft eher suboptimal.

2.1 Julia als Taschenrechner

Berechne $12^{1/3} + \frac{3\sqrt{2}}{\sin(0.5) - \cos(\frac{\pi}{4})\log(3)} + e^5$

```
12^(1/3) + 3sqrt(2) / (sin(.5) - cos(pi/4)*log(3)) + exp(5)
```

```
136.43732662344087
```

Man beachte:

- Potenzen schreibt man `a^b`.
- Die Konstante `pi` ist vordefiniert.
- `log()` ist der natürliche Logarithmus.
- Das Multiplikationszeichen `a*b` kann nach einer Zahl weggelassen werden, wenn eine Variable, Funktion oder Klammer folgt.

2.2 Die wichtigsten Tasten: `Tab` und `?`

Man drücke beim Programmieren immer wieder die Tabulatortaste, sobald 2...3 Buchstaben eines Wortes getippt sind. Es werden dann mögliche Ergänzungen angezeigt bzw. ergänzt, wenn die Ergänzung eindeutig ist. Das spart Zeit und bildet ungemein:

```
lo ▷ Tab
```

```
log
log2
lock
local
log1p
log10
lowercase
lowercasefirst
```

```
pri ▷ Tab
```

```
print
println
printstyled
primitive type
```

Die eingebaute Julia-Hilfe `?name` zu allen Funktionen und Konstrukten ist sehr umfassend. Hier ein eher kurzes Beispiel:

```
?for
```

```
search: for foreach fourthroot foldr floor mapfoldr factorial EOFError
```

```
for
```

`for` loops repeatedly evaluate a block of statements while iterating over a sequence of values.

The iteration variable is always a new variable, even if a variable of the same name exists in the enclosing scope. Use `outer` to reuse an existing local variable for iteration.

Examples

```
julia> for i in [1, 4, 0]
           println(i)
       end
1
4
0
```

2.3 Variablen und Zuweisungen

Variablen entstehen durch Zuweisung *(assignment)* mit dem Zuweisungsoperator `=` . Danach können sie in weiteren Anweisungen verwendet werden.

```
x = 1 + sqrt(5)
y = x / 2
```

```
1.618033988749895
```

Im interaktiven Betrieb zeigt Julia das Ergebnis der letzten Operation an.

> **Hinweis**
>
> Zuweisungen sind keine mathematischen Gleichungen. Die Semantik des Zuweisungsoperators (Gleichheitszeichens) ist:
>
> - berechne die rechte Seite und
> - weise das Ergebnis der linken Seite zu.
>
> Ausdrücke wie `x + y = sin(2)` sind daher unzulässig. Links darf nur ein Variablenname stehen.

2.4 Datentypen

Julia ist eine stark typisierte Sprache. Alle Objekte haben einen Typ. So gibt es unter anderem die Basistypen

- Ganze Zahlen *(integers)*,
- Gleitkommazahlen *(floating point numbers)*,
- Zeichenketten *(strings)* und
- Wahrheitswerte *(booleans)*.

Den Typ einer Variablen kann man mit der Funktion `typeof()` ermitteln.

```
for x ∈ (42, 12.0, 3.3e4, "Hallo!", true)
    println("x = ", x, " ..... Typ: ", typeof(x))
end
```

```
x = 42 ..... Typ: Int64
x = 12.0 ..... Typ: Float64
x = 33000.0 ..... Typ: Float64
x = Hallo! ..... Typ: String
x = true ..... Typ: Bool
```

Die Standard-Gleitkommazahl hat eine Länge von 64 Bit, entspricht also einer `double` in C/C++/Java.

Julia ist eine dynamisch typisierte Sprache. Variablen haben keinen Typ. Sie sind typlose Referenzen (Zeiger) auf Objekte. Wenn man vom „Typ einer Variablen“ spricht, meint man den Typ des Objektes, das der Variablen gerade zugewiesen ist.

```
x = sqrt(2)

println( typeof(x), " - Wert von x = $x" )

x = "Jetzt bin ich keine Gleitkommazahl mehr!"

println( typeof(x), "  - Wert von x = $x" )
```

```
Float64 - Wert von x = 1.4142135623730951
String  - Wert von x = Jetzt bin ich keine Gleitkommazahl mehr!
```

2.5 Druckanweisungen

Die Funktion `println()` unterscheidet sich von `print()` dadurch, dass sie am Ende einen Zeilenvorschub ausgibt.

```
print(y)
print("...die Zeile geht weiter...")
print("immernoch...")
println(y)
println("Neue Zeile")
println("Neue Zeile")
```

```
1.618033988749895...die Zeile geht weiter...immernoch...1.618033988749895
Neue Zeile
Neue Zeile
```

Beide Funkionen können als Argument eine Liste von *strings* und Variablen bekommen. Man kann Variablen auch in *strings* einbetten, indem man dem Variablennamen ein Dollarzeichen voranstellt *(string interpolation)*.

```
x = 23
y = 3x + 5
zz = "Fertig!"
println("x= ", x, " ...und y= ", y, "...", zz)   # 1. Variante
println("x= $x ...und y= $y...$zz")              # Variante mit string interpolation
```

```
x= 23 ...und y= 74...Fertig!
x= 23 ...und y= 74...Fertig!
```

Wenn man ein Dollarzeichen drucken will...

muss man einen *backslash* voranstellen. Wenn man einen *backslash* drucken will, muss man ihn verdoppeln.

```
println("Ein Dollar: 1\$ und drei backslashs: \\\\\\ ")
```

```
Ein Dollar: 1$ und drei backslashs: \\\
```

2.6 Funktionen

Funktionsdefinitionen beginnen mit dem Schlüsselwort `function` und enden mit dem Schlüsselwort `end`. In der Regel haben sie eine oder mehrere Argumente und geben beim Aufruf ein berechnetes Objekt mit der `return`-Anweisung zurück.

```
function hypotenuse(a, b)    # heute besonders umständlich
    c2 = a^2 + b^2
    c  = sqrt(c2)
    return c
end
```

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

Nach ihrer Definition kann die Funktion benutzt (aufgerufen) werden. Die in der Definition verwendeten Variablen `a,b,c,c2` sind lokale Variablen und stehen außerhalb der Funktionsdefinition nicht zur Verfügung.

```
x = 3
z = hypotenuse(x, 4)
println("z = $z")
println("c = $c")
```

```
z = 5.0
LoadError: UndefVarError: `c` not defined

Stacktrace:
 [1] top-level scope
   @ In[14]:4
```

Sehr einfache Funktionen können auch als Einzeiler definiert werden.

```
hypotenuse(a, b) = sqrt(a^2+b^2)
```

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

2.7 Tests

Tests liefern einen Wahrheitswert zurück.

```
x = 3^2
x < 2^3
```

```
false
```

Neben den üblichen arithmetischen Vergleichen `==, !=, <, <= ,> ,>=` gibt es noch viele andere Tests. Natürlich kann man das Ergebnis eines Tests auch einer Variablen zuweisen, welche dann vom Typ `Bool` ist. Die logischen Operatoren `&&`, `||` und Negation `!` können in Tests verwendet werden.

```
test1 = "Auto" in ["Fahrrad", "Auto", "Bahn"]
test2 = x == 100 ||  !(x <= 30 && x > 8)
test3 = startswith("Lampenschirm", "Lamp")
println("$test1 $test2 $test3")
```

```
true false true
```

2.8 Verzweigungen

Verzweigungen (bedingte Anweisungen) haben die Form

```
if <Test>
    <Anweisung1>
    <Anweisung2>
    ...
end
```

Ein `else`-Zweig und `elseif`-Zweige sind möglich.

```
x = sqrt(100)

if x > 20
    println("Seltsam!")
else
    println("OK")
    y = x + 3
end
```

```
OK
13.0
```

Einrückungen verbessern die Lesbarkeit, sind aber fakultativ. Zeilenumbrüche trennen Anweisungen. Das ist auch durch Semikolon möglich. Obiger Codeblock ist für Julia identisch zu folgender Zeile:

```
# Bitte nicht so programmieren! Sie werden es bereuen!
x=sqrt(100); if x > 20 println("Seltsam!") else println("OK"); y = x + 3 end
```

```
OK
13.0
```

Es wird dringend empfohlen, von Anfang an den eigenen Code übersichtlich mit sauberen Einrückungen zu formatieren!

2.9 Einfache `for`-Schleifen

zum wiederholten Abarbeiten von Anweisungen haben die Form

```
for  <Zähler> = Start:Ende
    <Anweisung1>
    <Anweisung2>
    ...
end
```

Beispiel:

```
sum = 0
for i = 1:100
    sum = sum + i
end
sum
```

```
5050
```

2.10 Arrays

1-dimensionale Arrays (Vektoren) sind eine einfache Form von Containern. Man kann sie mit echigen Klammern anlegen und auf die Elemente per Index zugreifen. Die Indizierung beginnt mit 1.

```
v = [12, 33.2,  17, 19, 22]
```

```
5-element Vector{Float64}:
 12.0
 33.2
 17.0
 19.0
 22.0
```

```
typeof(v)
```

```
VectorFloat64 (alias for ArrayFloat64, 1)
```

```
v[1] = v[4] + 10
v
```

```
5-element Vector{Float64}:
 29.0
 33.2
 17.0
 19.0
 22.0
```

Man kann leere Vektoren anlegen und sie verlängern.

```
v = []     # leerer Vektor
push!(v, 42)
push!(v, 13)
v
```

```
2-element Vector{Any}:
 42
 13
```

Nachtrag: wie die Wirkung der Tab-Taste auf dieser Seite simuliert wurde...

```
using REPL

function Tab(s)
    l = filter(x->startswith(x,s), REPL.doc_completions(s))
    println.(l[2:end])
    return   # return nothing, since broadcast println produces empty vector
end

▷ = |>     # https://docs.julialang.org/en/v1/manual/functions/#Function-composition-and-piping

pri = "pri";
```

```
pri ▷ Tab
```

```
print
println
```

```
printstyled
primitive type
```

3 Erste Miniprogramme

3.1 Was sollte man zum Programmieren können?

- **Denken in Algorithmen**:
  Welche Schritte sind zur Lösung des Problems nötig? Welche Daten müssen gespeichert, welche Datenstrukturen angelegt werden? Welche Fälle können auftreten und müssen erkannt und behandelt werden?

- **Umsetzung des Algorithmus in ein Programm**:
  Welche Datenstrukturen, Konstrukte, Funktionen… stellt die Programmiersprache bereit?
- **Formale Syntax**:
  Menschen verstehen 'broken English'; Computer verstehen kein 'broken C++' oder 'broken Julia'. Die Syntax muss beherrscht werden.
- **„Ökosystem" der Sprache**:
  Wie führe ich meinen Code aus? Wie funktionieren die Entwicklungsumgebungen? Welche Optionen versteht der Compiler? Wie installiere ich Zusatzbibliotheken? Wie lese ich Fehlermeldungen? Wo kann ich mich informieren?
- **die Kunst des Debugging**:
  Programmieranfänger sind oft glücklich, wenn sie alle Syntaxfehler eliminiert haben und das Programm endlich 'durchläuft'. Bei komplexeren Programmen fängt die Arbeit jetzt erst an, denn nun muss getestet und die Fehler im Algorithmus gefunden und behoben werden.
- **Gefühl für die Effizienz und Komplexität von Algorithmen**
- **Besonderheiten der Computerarithmetik**, insbesondere der Gleitkommazahlen

Das erschließt sich nicht alles auf einmal. Haben Sie Geduld mit sich und 'spielen' Sie mit der Sprache.

Das Folgende soll eine kleine Anregung dazu sein.

3.2 Project Euler

Eine hübsche Quelle für Programmieraufgaben mit mathematischem Charakter und sehr unterschiedlichen Schwierigkeitsgraden ist Project Euler. Die Aufgaben sind so gestaltet, dass keinerlei Eingaben notwendig und das gesuchte Ergebnis eine einzige Zahl ist. So kann man sich voll auf das Programmieren des Algorithmus konzentrieren.

3.2.1 Beispiel 1

> Project Euler Problem No 1
>
> If we list all the natural numbers below 10 that are multiples of 3 or 5, we get 3, 5, 6 and 9. The sum of these multiples is 23. Find the sum of all the multiples of 3 or 5 below 1000.

1. Algorithmus:
   - Teste alle natürlichen Zahlen <1000 auf Teilbarkeit durch 3 oder durch 5 und
   - summiere die teilbaren Zahlen auf.

2. Umsetzung:

   Wie liefert Julia den Rest der Ganzzahldivision? Solche Funktionen heißen typischerweise `rem()` (für *remainder*) oder `mod()`. Nachlesen in der Doku oder ausprobieren von `?rem` und `?mod` in der Julia-REPL zeigt, dass es beides gibt. Der Unterschied liegt in der Behandlung negativer ganzer Zahlen. Für natürliche Zahlen `n,m` liefern `mod(n,m)` und `rem(n,m)` dasselbe und letzteres hat auch noch die Infix-Form `n % m`.

   Wie testet man eine Reihe von Werten durch und summiert auf? Da gibt es ein Standardmuster: `for`-Schleife und Akkumulatorvariable:

Eine Lösungsvariante:

```
s = 0                               # Akkumulatorvariable initialisieren
for i in 1:999                      # Schleife
    if i % 3 == 0 || i % 5 == 0     # Test
        s += i                      # Zu Akkumulator addieren
    end                             # Ende Test
end                                 # Ende Schleife
println(" Die Antwort ist: $s")                        # Ausgabe des Ergebnisses
```

```
 Die Antwort ist: 233168
```

Natürlich geht das auch etwas kürzer:

Noch eine Lösungsvariante:

```
sum([i for i in 1:999 if i%3==0||i%5==0])
```

```
233168
```

3.2.2 Beispiel 2

Project Euler Problem No 92

A number chain is created by continuously adding the square of the digits in a number to form a new number until it has been seen before.
For example,
44 → 32 → 13 → 10 → **1** → **1**
85 → **89** → 145 → 42 → 20 → 4 → 16 → 37 → 58 → **89**
Therefore any chain that arrives at 1 or 89 will become stuck in an endless loop. What is most amazing is that EVERY starting number will eventually arrive at 1 or 89.
How many starting numbers below ten million will arrive at 89?

Programme kann man *top-down* und *bottom-up* entwickeln. *Top-down* würde hier wohl bedeuten: „Wir fangen mit einer Schleife `for i = 1:9999999` an." Der andere Weg führt über einzeln testbare Komponenten und ist oft zielführender. (Und ab einer gewissen Projektgröße nähert man sich sowieso von 'top' und 'bottom' gleichzeitig dem Ziel.)

Funktion Nr. 1

Es soll untersucht werden, wie sich die Zahlen unter dem wiederholten Berechnen der 'Quadratquersumme' (Summe der Quadrate der Ziffern) verhalten. Also sollte man eine Funktion schreiben und testen, die diese 'Quadratquersumme' berechnet.

`q2sum(n)` berechnet die Summe der Quadrate der Ziffern von n im Dezimalsystem:

```
function q2sum(n)
    s = 0                       # Akkumulator für die Summe
    while n > 9                 # solange noch mehrstellig...
       q, r = divrem(n, 10)     # berechne Ganzzahlquotient und Rest bei Division durch 10
       s += r^2                 # addiere quadrierte Ziffer zum Akkumulator
       n = q                    # mache weiter mit der durch 10 geteilten Zahl
    end
    s += n^2                    # addiere das Quadrat der letzten Ziffer
    return s
end
```

... und das testen wir jetzt natürlich:

```
for i in [1, 7, 33, 111, 321, 1000, 73201]
   j = q2sum(i)
   println("Quadratquersumme von $i = $j")
end
```

```
Quadratquersumme von 1 = 1
Quadratquersumme von 7 = 49
Quadratquersumme von 33 = 18
Quadratquersumme von 111 = 3
Quadratquersumme von 321 = 14
Quadratquersumme von 1000 = 1
Quadratquersumme von 73201 = 63
```

Im Sinne der Aufgabe wenden wir die Funktion wiederholt an:

```
n = 129956799
for i in 1:14
    n = q2sum(n)
    println(n)
end
```

```
439
106
37
58
89
145
42
20
4
16
37
58
89
145
```

... und haben hier einen der '89er Zyklen' getroffen.

Funktion Nr. 2

Wir müssen testen, ob die wiederholte Anwendung der Funktion `q2sum()` schließlich zu **1** führt oder in diesem **89er**-Zyklus endet.

`q2test(n)` ermittelt, ob wiederholte Quadratquersummenbildung in den 89er-Zyklus führt:

```
function q2test(n)
    while n !=1 && n != 89       # solange wir nicht bei 1 oder 89 angekommen sind....
        n = q2sum(n)             # wiederholen
    end
    if n==1 return false end     # keine 89er-Zahl
    return true                  # 89er-Zahl gefunden
end
```

... und damit können wir die Aufgabe lösen:

```
c = 0                        # mal wieder ein Akkumulator
for i = 1 : 10_000_000 - 1   # so kann man in Julia Tausenderblöcke zur besseren Lesbarkeit abtrennen
    if q2test(i)             # q2test() gibt einen Boolean zurück, kein weiterer Test nötig
        c += 1               #    'if x == true' ist redundant, 'if x' reicht völlig
    end
end
println("$c numbers below ten million arrive at 89.")
```

```
8581146 numbers below ten million arrive at 89.
```

Zahlen, bei denen die iterierte Quadratquersummenbildung bei 1 endet, heißen übrigens fröhliche Zahlen und wir haben gerade die Anzahl der traurigen Zahlen kleiner als 10.000.000 berechnet.

Hier nochmal das vollständige Programm:

unsere Lösung von Project Euler No 92:

```
function q2sum(n)
    s = 0
    while n > 9
       q, r = divrem(n, 10)
       s += r^2
       n = q
    end
    s += n^2
    return s
end

function q2test(n)
    while n !=1 && n != 89
        n = q2sum(n)
    end
    if n==1 return false end
    return true
end

c = 0
for i = 1 : 10_000_000 - 1
    if q2test(i)
        c += 1
    end
end
println("$c numbers below ten million arrive at 89.")
```

```
8581146 numbers below ten million arrive at 89.
```

4 Grundlagen der Syntax

4.1 Namen von Variablen, Funktionen, Typen etc.

- Namen können Buchstaben, Ziffern, den Unterstrich _ und das Ausrufezeichen `!` enthalten.
- Das erste Zeichen muss ein Buchstabe oder ein Unterstrich sein.
- Groß- und Kleinbuchstaben werden unterschieden: `Nmax` und `NMAX` sind verschiedene Variablen.
- Als Zeichensatz wird Unicode verwendet. Damit stehen über 150 Schriften und zahlreiche Symbole zur Verfügung.
- Es gibt eine kurze Liste reservierter Schlüsselwörter: `if, then, function, true, false,...`

Beispiel

zulässig: `i, x, Ω, x2, DieUnbekannteZahl, neuer_Wert, 🎷, Zähler, лічильник, einself!!!!,...`
unzulässig: `Uwe's_Funktion, 3achsen, A#B, $this_is_not_Perl, true,...`

Anmerkung

Neben den *reserved keywords* der Kernsprache sind zahlreiche weitere Funktionen und Objekte vordefiniert, wie z.B. die mathematischen Funktionen `sqrt(), log(), sin()`. Diese Definitionen finden sich in dem Modul `Base`, welches Julia beim Start automatisch lädt. Namen aus `Base` können umdefiniert werden, solange sie noch nicht verwendet wurden:

```
log = 3
1 + log
```

```
4
```

Jetzt ist natürlich der Logarithmus kaputt:

```
x = log(10)
```

```
LoadError: MethodError: objects of type Int64 are not callable
Maybe you forgot to use an operator such as *, ^, %, / etc. ?

Stacktrace:
 [1] top-level scope
   @ In[3]:1
```

(siehe auch https://stackoverflow.com/questions/65902105/how-to-reset-any-function-in-julia-to-its-original-state)

4.2 Anweisungen

- Im Normalfall enthält eine Zeile eine Anweisung.
- Wenn eine Anweisung am Zeilenende als unvollständig erkennbar ist durch
  - offene Klammern
  - Operationszeichen,

dann wird die nächste Zeile als Fortsetzung aufgefasst.
- Mehrere Anweisungen pro Zeile können durch Semikolon getrennt werden.
- Im interaktiven Betrieb (REPL oder Notebook) unterdrückt ein Semikolon nach der letzten Anweisung die Ausgabe des Ergebnisses dieser Anweisung.

Beispiel

Im interaktiven Betrieb wird der Wert der letzten Anweisung auch ohne explizites `print()` ausgegeben:

```
println("Hallo 🌍!")
x = sum([i^2 for  i=1:10])
```

```
Hallo 🌍!
385
```

Das Semikolon unterdrückt das:

```
println("Hallo 🌍!")
x = sum([i^2 for  i=1:10]);
```

```
Hallo 🌍!
```

Warnung

Bei mehrzeiligen Anweisungen muss die fortzusetzende Zeile mit einer offenen Operation oder Klammer enden:

```
x = sin(π/2) +
   3 * cos(0)
```

```
4.0
```

Also geht das Folgende schief, aber leider **ohne eine Fehlermeldung**!

```
x = sin(π/2)
  + 3 * cos(0)
println(x)
```

```
1.0
```

Hier wird das `+` in der zweiten Zeile als Präfix-Operator (Vorzeichen) interpretiert. Damit sind 1. und 2. Zeile jeweils für sich vollständige, korrekte Ausdrücke (auch wenn die 2. Zeile natürlich völlig nutzlos ist) und werden auch so abgearbeitet. Moral: Wenn man längere Ausdrücke auf mehrere Zeilen aufteilen will, sollte man immer eine Klammer aufmachen. Dann ist egal, wo der Zeilenumbruch ist:

```
x = ( sin(π/2)
   + 3 * cos(0) )
println(x)
```

```
4.0
```

4.3 Kommentare

Julia kennt 2 Arten von Kommentaren im Programmtext:

```
# Einzeilige Kommentare beginnen mit einem Doppelkreuz.

x = 2    # alles vom '#' bis zum Zeilenende ist ein Kommentar und wird ignoriert. x = 3
```

```
2
```

```
#=
   Ein- und mehrzeilige Kommentare können zwischen #= ...  =# eingeschlossen werden.
   Dabei sind verschachtelte Kommentare möglich.
   #=
      d.h., anders als in C/C++/Java endet der Kommentar nicht mit dem ersten
      Kommentar-Endezeichen, sondern die #=...=# - Paare wirken wie Klammern.
   =#
   Der automatische 'syntax highlighter' weiss das leider noch nicht, wie die wechselnde
   Graufärbung dieses Kommentars zeigt.
=#


x #= das ist ein seltener Variablenname! =#  = 3
```

```
3
```

4.4 Datentypen Teil I

- Julia ist eine stark typisierte Sprache. Alle Objekte haben einen Typ. Funktionen/Operationen erwarten Argumente mit dem richtigen Typ.
- Julia ist eine dynamisch typisierte Sprache. Variablen haben keinen Typ. Sie sind Namen, die durch Zuweisung `x = ...` an Objekte gebunden werden können.
- Wenn man vom „Typ einer Variablen“ spricht, meint man den Typ des Objektes, das der Variablen gerade zugewiesen ist.
- Funktionen/Operatoren können verschiedene *methods* für verschiedene Argumenttypen implementieren.
- Abhängig von den konkreten Argumenttypen wird dann bei Verwendung einer Funktion entschieden, welche Methode benutzt wird (*dynamic dispatch*).

Einfache Basistypen sind z.B.:

```
Int64, Float64, String, Char, Bool
```

```
x = 2
x, typeof(x), sizeof(x)
```

```
(2, Int64, 8)
```

```
x = 0.2
x, typeof(x), sizeof(x)
```

```
(0.2, Float64, 8)
```

```
x = "Hallo!"
x, typeof(x), sizeof(x)
```

```
("Hallo!", String, 6)
```

```
x = 'Ω'
x, typeof(x), sizeof(x)
```

```
('Ω', Char, 4)
```

```
x = 3 > π
x, typeof(x), sizeof(x)
```

```
(false, Bool, 1)
```

- `sizeof()` liefert die Größe eines Objekts oder Typs in Bytes (1 Byte = 8 Bit)
- 64bit Ganzzahlen und 64bit Gleitkommazahlen entsprechen dem Befehlssatz moderner Prozessoren und sind daher die numerischen Standardtypen.
- Zeichen/*chars* `'A'` und Zeichenketten/*strings* `"A"` der Länge 1 sind verschiedene Objekte.

4.5 Ablaufsteuerung

4.5.1 `if`-Blöcke

- Ein `if`-Block *kann* **beliebig viele** `elseif`-Zweige und als letztes maximal **einen** `else`-Zweig enthalten.
- Der Block hat einen Wert, den Wert der letzten ausgeführten Anweisung.

```
x = 33
y = 44
z = 34

if x < y && z != x                    # elseif- und else-Blöcke sind optional
    println("yes")
    x += 10
elseif x < z                          # beliebig viele elseif-Blöcke
    println(" x is smaller than z")
elseif x == z+1
    println(" x is successor of z")
else                                  # maximal ein else-Block
    println("Alles falsch")
end                                   # Wert des gesamten Blocks ist der Wert der
                                      # letzten ausgeführten Auswertung
```

```
yes
43
```

Kurze Blöcke kann man in eine Zeile schreiben:

```
if x > 10 println("x is larger than 10") end
```

```
x is larger than 10
```

Der Wert eines `if`-Blocks kann natürlich zugewiesen werden:

```
y = 33
z = if y > 10
      println("y is larger than 10")
      y += 1
    end
z
```

```
y is larger than 10
34
```

4.5.2 Auswahloperator (ternary operator) `test ? exp1 : exp2`

```
x = 20
y = 15
z = x < y ? x+1 : y+1
```

```
16
```

ist äquivalent zu

```
z = if x < y
        x+1
    else
        y+1
    end
```

```
16
```

4.6 Vergleiche, Tests, Logische Operationen

4.6.1 Arithmetische Vergleiche

- `==`
- `!=`, `≠`
- `>`
- `>=`, `≥`
- `<`
- `<=`, `≤`

Wie üblich, ist der Test auf Gleichheit `==` vom Zuweisungsoperator `=` zu unterscheiden. Vergleichen lässt sich so gut wie alles:

```
"Aachen" < "Leipzig", 10 ≤ 10.01, [3,4,5] < [3,6,2]
```

```
(true, true, true)
```

Nun ja, fast alles:

```
3 < "vier"
```

```
LoadError: MethodError: no method matching isless(::Int64, ::String)

Closest candidates are:
  isless(::Missing, ::Any)
   @ Base missing.jl:87
  isless(::Any, ::Missing)
   @ Base missing.jl:88
  isless(::Real, ::AbstractFloat)
   @ Base operators.jl:178
  ...


Stacktrace:
 [1] <(x::Int64, y::String)
   @ Base ./operators.jl:352
 [2] top-level scope
   @ In[22]:1
```

Die Fehlermeldung zeigt ein paar Grundprinzipien von Julia:

- Operatoren sind auch nur Funktionen: `x < y` wird zum Funktionsaufruf `isless(x, y)`.
- Funktionen (und damit Operatoren) können verschiedene *methods* für verschiedene Argumenttypen implementieren.
- Abhängig von den konkreten Argumenttypen wird beim Aufruf der Funktion entschieden, welche Methode benutzt wird (*dynamic dispatch*).

Man kann sich alle Methoden zu einer Funktion anzeigen lassen. Das gibt einen Einblick in das komplexe Typssystem von Julia:

```
methods(<)
```

```
# 75 methods for generic function "<" from Base:
  [1] <(x::BigFloat, y::BigFloat)
     @ Base.MPFR mpfr.jl:848
  [2] <(x::BigFloat, y::AbstractIrrational)
     @ irrationals.jl:102
  [3] <(x::BigFloat, y::UnionFloat16, Float32, Float64)
     @ Base.MPFR mpfr.jl:880
  [4] <(x::BigFloat, y::Integer)
     @ Base.MPFR mpfr.jl:878
  [5] <(x::Bool, y::Bool)
     @ bool.jl:158
  [6] <(x::Float16, y::AbstractIrrational)
     @ irrationals.jl:98
  [7] <(x::Float16, y::UnionInt16, UInt16)
     @ float.jl:604
  [8] <(x::Float16, y::UnionInt128, Int64, UInt128, UInt64)
     @ float.jl:598
  [9] <(x::Int128, y::Float64)
     @ float.jl:576
 [10] <(x::Int128, y::Float32)
     @ float.jl:576
 [11] <(a::Base.BinaryPlatforms.CPUID.ISA, b::Base.BinaryPlatforms.CPUID.ISA)
     @ Base.BinaryPlatforms.CPUID cpuid.jl:21
 [12] <(x::Float64, y::UInt128)
     @ float.jl:585
 [13] <(x::Float64, y::Int128)
     @ float.jl:585
 [14] <(x::Float64, y::UInt64)
     @ float.jl:585
 [15] <(x::Float64, y::Int64)
     @ float.jl:585
 [16] <(x::Float64, y::AbstractIrrational)
     @ irrationals.jl:94
 [17] <(::Missing, ::Missing)
     @ missing.jl:83
 [18] <(::Missing, ::Any)
     @ missing.jl:84
 [19] <(::Any, ::Missing)
     @ missing.jl:85
 [20] <(x::RationalBigInt, y::AbstractIrrational)
     @ irrationals.jl:137
 [21] <(x::Base.JuliaSyntax.Kind, y::Base.JuliaSyntax.Kind)
     @ Base.JuliaSyntax /cache/build/default-maughin-0/julialang/julia-release-1-dot-10/base/JuliaSyntax/src/k
 [22] <(x::UInt64, y::Float64)
     @ float.jl:576
```

```
[23] <(x::UInt64, y::Float32)
    @ float.jl:576
[24] <(x::BigInt, y::BigInt)
    @ Base.GMP gmp.jl:735
[25] <(x::BigInt, f::UnionFloat16, Float32, Float64)
    @ Base.GMP gmp.jl:738
[26] <(x::BigInt, i::Integer)
    @ Base.GMP gmp.jl:736
[27] <(i::Integer, x::BigInt)
    @ Base.GMP gmp.jl:737
[28] <(x::Float32, y::UInt128)
    @ float.jl:585
[29] <(x::Float32, y::Int128)
    @ float.jl:585
[30] <(x::Float32, y::UInt64)
    @ float.jl:585
[31] <(x::Float32, y::Int64)
    @ float.jl:585
[32] <(x::Float32, y::AbstractIrrational)
    @ irrationals.jl:96
[33] <(::Tuple, ::Tuple)
    @ tuple.jl:527
[34] <(::Tuple, ::Tuple)
    @ tuple.jl:528
[35] <(::Tuple, ::Tuple)
    @ tuple.jl:529
[36] <(x::Int64, y::Float64)
    @ float.jl:576
[37] <(x::Int64, y::Float32)
    @ float.jl:576
[38] <(x::UInt128, y::Float64)
    @ float.jl:576
[39] <(x::UInt128, y::Float32)
    @ float.jl:576
[40] <(x::Integer, y::BigFloat)
    @ Base.MPFR mpfr.jl:879
[41] <(x::Integer, y::Rational)
    @ rational.jl:392
[42] <(x::Ptr, y::Ptr)
    @ pointer.jl:279
[43] <(::Irrationals, ::Irrationals) where s
    @ irrationals.jl:79
[44] <(x::AbstractIrrational, y::AbstractIrrational)
    @ irrationals.jl:80
[45] <(x::Rational, y::Rational)
    @ rational.jl:383
[46] <(x::T, y::T) where T<:UnionFloat16, Float32, Float64
    @ float.jl:536
[47] <(x::T, y::T) where T<:UnionUInt128, UInt16, UInt32, UInt64, UInt8
    @ int.jl:513
[48] <(x::T, y::T) where T<:UnionInt128, Int16, Int32, Int64, Int8
    @ int.jl:83
[49] <(x::T, y::T) where T<:Real
    @ promotion.jl:522
[50] <(x::AbstractIrrational, y::RationalBigInt)
    @ irrationals.jl:136
[51] <(x::AbstractIrrational, y::RationalT) where T
    @ irrationals.jl:118
```

```
[52] <(x::AbstractIrrational, y::BigFloat)
    @ irrationals.jl:99
[53] <(x::AbstractIrrational, y::Float16)
    @ irrationals.jl:97
[54] <(x::AbstractIrrational, y::Float32)
    @ irrationals.jl:95
[55] <(x::AbstractIrrational, y::Float64)
    @ irrationals.jl:93
[56] <(x::AbstractFloat, y::Rational)
    @ rational.jl:408
[57] <(x::RationalT, y::AbstractIrrational) where T
    @ irrationals.jl:127
[58] <(x::Rational, y::AbstractFloat)
    @ rational.jl:408
[59] <(x::Rational, y::Integer)
    @ rational.jl:391
[60] <(x::UnionFloat16, Float32, Float64, y::BigFloat)
    @ Base.MPFR mpfr.jl:881
[61] <(f::UnionFloat16, Float32, Float64, x::BigInt)
    @ Base.GMP gmp.jl:739
[62] <(x::UnionInt16, UInt16, y::Float16)
    @ float.jl:605
[63] <(x::UnionInt32, UInt32, y::UnionFloat16, Float32)
    @ float.jl:602
[64] <(x::UnionFloat16, Float32, y::UnionInt32, UInt32)
    @ float.jl:601
[65] <(x::UnionInt128, Int64, UInt128, UInt64, y::Float16)
    @ float.jl:599
[66] <(x::UnionUInt128, UInt16, UInt32, UInt64, UInt8, y::UnionInt128, Int16, Int32, Int64, Int8)
    @ int.jl:520
[67] <(x::UnionInt128, Int16, Int32, Int64, Int8, y::UnionUInt128, UInt16, UInt32, UInt64, UInt8)
    @ int.jl:519
[68] <(x::Real, y::Real)
    @ promotion.jl:462
[69] <(a::NamedTuplen, b::NamedTuplen) where n
    @ namedtuple.jl:258
[70] <(x::Base.TwicePrecisionT, y::Base.TwicePrecisionT) where T
    @ twiceprecision.jl:786
[71] <(a::AbstractSet, b::AbstractSet)
    @ abstractset.jl:513
[72] <(t1::TupleAny, Any, Any, Any, Any, Any, Any, Any, Any, Any, Any, Any, Any, Any, Any, Any, Any, Any, Any
    @ tuple.jl:540
[73] <(t1::Tuple, t2::Tuple)
    @ tuple.jl:530
[74] <(x)
    @ operators.jl:1222
[75] <(x, y)
    @ operators.jl:352
```

Zuletzt noch: Vergleiche können gekettet werden.

```
10 < x ≤ 100   # das ist äquivalent zu
               #   10 < x && x ≤ 100
```

```
true
```

4.6.2 Tests

Einge Funktionen vom Typ `f(c::Char) -> Bool`

```
isnumeric('a'), isnumeric('7'), isletter('a')
```

```
(false, true, true)
```

und vom Typ `f(s1::String, s2::String) -> Bool`

```
contains("Lampenschirm", "pensch"), startswith("Lampenschirm", "Lamb"), endswith("Lampenschirm", "rm")
```

```
(true, false, true)
```

- Die Funktion `in(item, collection) -> Bool` testet, ob `item` in `collection` ist.
- Sie hat auch das Alias `∈(item, collection)` und
- sowohl `in` als auch `∈` können auch als Infix-Operatoren geschrieben werden.

```
x = 3
x in [1, 2, 3, 4, 5]
```

```
true
```

```
x ∈ [1, 2, 33, 4, 5]
```

```
false
```

4.6.3 Logische Operationen: `&&`, `||`, `!`

```
3 < 4 && !(2 > 8) && !contains("aaa", "b")
```

```
true
```

Bedingte Auswertung (*short circuit evaluation*)

- in `a && b` wird `b` nur ausgewertet, wenn `a == true`
- in `a || b` wird `b` nur ausgewertet, wenn `a == false`

(i) Damit kann `if test statement end` auch als `test && statement` geschrieben werden.

(ii) Damit kann `if !test statement end` als `test || statement` geschrieben werden.

Als Beispiel[1] hier eine Implementierung der Fakultätsfunktion *(factorial)*:

```
function fact(n::Int)
    n >= 0 || error("n must be non-negative")
    n == 0 && return 1
    n * fact(n-1)
end

fact(5)
```

```
120
```

Natürlich kann man alle diese Tests auch Variablen vom Typ `Bool` zuordnen und diese Variablen können als Tests in `if`- und `while`-Blöcken verwendet werden:

[1] aus der Julia-Dokumentation

```
x = 3 < 4
y = 5 ∈ [1, 2, 5, 7]
z = x && y
if z                        # äquivalent zu:  if 3 < 4  &&  5 in [1,2,5,7]
    println("Stimmt alles!")
end
```

```
Stimmt alles!
```

- In Julia müssen alle Tests in einem logischen Ausdruck vom Typ `Bool` sein.
- Es gibt keine implizite Konvertierung à la *"0 is false and 1 (or anything != 0) is true"*
- Wenn `x` ein numerischer Typ ist, dann muss daher das C-Idiom `if(x)` als `if x != 0` geschrieben werden.
- Es gibt eine Ausnahme zur Unterstützung der *short circuit evaluation*:
  - bei den Konstrukten `a && b && c...` bzw `a || b || c...` muss der letzte Teilausdruck nicht vom Typ `Bool` sein, wenn diese Konstrukte nicht als Tests in `if` oder `while` verwendet werden:

```
z = 3 < 4 && 10 < 5 &&  sqrt(3^3)
z, typeof(z)
```

```
(false, Bool)
```

```
z = 3 < 4 && 10 < 50 &&  sqrt(3^3)
z, typeof(z)
```

```
(5.196152422706632, Float64)
```

4.7 Schleifen *(loops)*

4.7.1 Die `while` ("solange")-Schleife

Syntax:

```
while *condition*
   *loop body*
end
```

Eine Reihe von Anweisungen (der Schleifenkörper) wird immer wieder abgearbeitet, solange eine Bedingung erfüllt ist.

```
i = 1          # typischerweise braucht der Test der
               # while-Schleife eine Vorbereitung ...
while i < 10
   println(i)
   i += 2      # ... und ein update
end
```

```
1
3
5
7
9
```

Der Körper einer `while`- und `for`-Schleife kann die Anweisungen `break` und `continue` enthalten. `break` stoppt die Schleife, `continue` überspringt den Rest des Schleifenkörpers und beginnt sofort mit dem nächsten Schleifendurchlauf.

```
i = 0

while i<10
   i += 1

   if i == 3
      continue     # beginne sofort nächsten Durchlauf,
   end             #  überspringe Rest des Schleifenkörpers

   println("i = $i")

   if i ≥ 5
      break        # breche Schleife ab
   end
end

println("Fertig!")
```

```
i = 1
i = 2
i = 4
i = 5
Fertig!
```

Mit `break` kann man auch Endlosschleifen verlassen:

```
i = 1

while true
    println(2^i)
    i += 1
    if i > 8 break end
end
```

```
2
4
8
16
32
64
128
256
```

4.7.2 `for`-Schleifen

Syntax:

```
for *var* in *iterable container*
   *loop body*
end
```

Der Schleifenkörper wird für alle Items aus einem Container durchlaufen.

Statt `in` kann immer auch ∈ verwendet werden. Im Kopf einer `for`-Schleife kann auch `=` verwendet werden.

```
for i ∈ ["Mutter", "Vater", "Tochter"]
    println(i)
end
```

```
Mutter
Vater
Tochter
```

Oft benötigt man einen numerischen Schleifenzähler. Dafür gibt es das *range*-Konstrukt. Die einfachsten Formen sind `Start:Ende` und `Start:Schrittweite:Ende`.

```
endwert = 5

for i ∈ 1:endwert
    println(i^2)
end
```

```
1
4
9
16
25
```

```
for i = 1:5.5  print(" $i") end
```

```
 1.0 2.0 3.0 4.0 5.0
```

```
for i = 1:2:14 print("  $i") end
```

```
  1  3  5  7  9  11  13
```

```
for k = 14 : -2.5 : 1 print("   $k") end
```

```
    14.0   11.5   9.0   6.5   4.0   1.5
```

Geschachtelte Schleifen *(nested loops)*

Ein `break` beendet die innerste Schleife.

```
for i = 1:3
    for j = 1:3
        println( (i,j) )
        if j == 2
            break
        end
    end
end
```

```
(1, 1)
(1, 2)
(2, 1)
(2, 2)
(3, 1)
(3, 2)
```

Man kann *nested loops* auch in einer `for`-Anweisung zusammenfassen. Dann beendet ein `break` die Gesamtschleife.

```
for i = 1:3, j=1:3     # im Prinzip dasselbe wie oben, aber:
    println( (i,j) )
    if j == 2
        break          # break bricht hier die Gesamtschleife ab
    end
```

```
end
```

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

! **Wichtig**: Die Semantik ist völlig anders, als bei C-artigen `for`-Schleifen!

Bei jedem Schleifendurchlauf wird die Laufvariable neu mit dem nächsten Element aus dem Container initialisiert.

```
for i = 1:5
    print(i," ... ")
    i += 2
    println(i)
end
```

```
1 ... 3
2 ... 4
3 ... 5
4 ... 6
5 ... 7
```

Die C-Semantik von `for(i=1; i<5; i++)` entspricht der `while`-Schleife:

```
i = 1
while i<5
   *loop body*   # hier kann auch wirksam an i rumgepfuscht werden
   i += 1
end
```

4.8 Unicode

Julia verwendet Unicode als Zeichensatz. Damit können für Variablen, Funktionen etc auch Bezeichner aus nicht-lateinischen Schriften (zB Kyrillisch, Koreanisch, Sanskrit, Runen, Emoji,...) verwendet werden. Die Frage, wie man solche Zeichen in seinem Editor eingeben kann und ob der verwendete Bildschirm-Font sie darstellen kann, ist nicht Julias Problem.

- Einige Unicode-Zeichen, z.B. `≤, ≠, ≥, π, ∈, √`, können anstelle von `<=, !=, >=, pi, in, sqrt` verwendet werden.
- über 3000 Unicode-Zeichen können in Julia in einer LaTeX-ähnlichen Weise mit der Tab-Vervollständigung eingegeben werden.
  - `\alpha<TAB>` wird zu α,
  - `\euler<TAB>` wird zu e (Eulersche Zahl `exp(1)`, spezielles Schreibschrift-e, `U+0212F`)
  - `\le<TAB>` wird zu ≤,
  - `\in<TAB>` wird zu ∈,
  - `\:rainbow:<TAB>` wird zu 🌈
    Hier geht es zur Liste.

4.9 Eigenheiten und Stolperfallen der Syntax

- Man kann den Multiplikationsoperator `*` nach einer numerischen Konstanten weglassen, wenn eine Variable, Funktion oder öffnende Klammer folgt.

```
z = 3.4x + 2(x+y) + xy
```

ist daher korrektes Julia. Beachte allerdings, dass der Term `xy` als eine Variable mit dem Namen xy interpretiert wird **und nicht** als Produkt von x und y!

- Diese Regel hat ein paar Tücken:

Das funktioniert wie erwartet:

```
e = 7
3e
```

```
21
```

Hier wird die Eingabe als Gleitkommazahl interpretiert – und `3E+2` oder `3f+2` (Float32) ebenso.

```
3e+2
```

```
300.0
```

Ein Leerzeichen schafft Eindeutigkeit:

```
3e + 2
```

```
23
```

Das funktioniert:

```
x = 4
3x + 3
```

```
15
```

...und das nicht. `0x`, `0o`, `0b` wird als Anfang einer Hexadezimal-, Oktal- bzw. Binärkonstanten interpretiert.

```
3y + 0x
```

```
LoadError: ParseError:
# Error @ file:///home/hellmund/Julia/23/Book/chapters/In[49]#1:6In[49]:1:6
3y + 0x
#    └─ ── invalid numeric constant

Stacktrace:
 [1] top-level scope
   @ In[49]:1
```

- Es gibt noch ein paar andere Fälle, bei denen die sehr kulante Syntax zu Überraschungen führt.

```
Wichtig  = 21
Wichtig! = 42       # Bezeichner können auch ein ! enthalten
(Wichtig, Wichtig!)
```

```
(21, 42)
```

```
Wichtig!=88
```

```
true
```

Julia interpretiert das als Vergleich `Wichtig != 88`.

Leerzeichen helfen:

```
Wichtig! = 88
Wichtig!
```

```
88
```

- Operatoren der Form `.*`, `.+`,... haben in Julia eine spezielle Bedeutung (*broadcasting*, d.h., vektorisierte Operationen).

```
1.+2.
```

```
LoadError: ParseError:
# Error @ file:///home/hellmund/Julia/23/Book/chapters/In[53]#1:1In[53]:1:1
1.+2.
└ ── ambiguous `.` syntax; add whitespace to clarify (eg `1.+2` might be `1.0+2` or `1 .+ 2`)

Stacktrace:
 [1] top-level scope
   @ In[53]:1
```

Wieder gilt: Leerzeichen schaffen Klarheit!

```
1. + 2.
```

```
3.0
```

5 Arbeit mit Julia: REPL, Pakete, Introspection

5.1 Dokumentation

Die offizielle Julia-Dokumentation https://docs.julialang.org/ enthält zahlreiche Übersichten, darunter:

- https://docs.julialang.org/en/v1/base/punctuation/ Verzeichnis der Symbole
- https://docs.julialang.org/en/v1/manual/unicode-input/ Verzeichnis spezieller Unicode-Symbole und deren Eingabe in Julia via Tab-Vervollständigung
- https://docs.julialang.org/en/v1/manual/mathematical-operations/#Rounding-functions Liste mathematischer Funktionen

5.2 Julia REPL (Read - Eval - Print - Loop)

Nach dem Start von Julia in einem Terminal kann man neben Julia-Code auch verschiedene Kommandos eingeben

Kommando	Wirkung
`exit()` oder `Ctrl-d`	exit Julia
`Ctrl-c`	interrupt
`Ctrl-l`	clear screen
Kommando mit `;` beenden	Ausgabe unterdrückt
`include("filename.jl")`	Datei mit Julia-Code einlesen und ausführen

Der REPL hat verschiedene Modi:

Modus	Prompt	Modus starten	Modus verlassen
default	`julia>`		`Ctrl-d`
Package manager	`pkg>`	`]`	`backspace`
Help	`help?>`	`?`	`backspace`
Shell	`shell>`	`;`	`backspace`

5.3 Jupyter-Notebooks (IJulia)

In einem Jupyter-Notebook sind die Modi sind als Einzeiler in einer eigenen Input-Zelle nutzbar:

(i) ein Kommando des Paket-Managers:

```
] status
```

(ii) eine Help-Abfrage:

```
?sin
```

(iii) Ein Shell-Kommando:

```
;ls
```

5.4 Der Paketmanager

Wichtiger Teil des *Julia Ecosystems* sind die zahlreichen Pakete, die Julia erweitern.

- Einige Pakete sind Teil jeder Julia-Installation und müssen nur mit einer `using Paketname`-Anweisung aktiviert werden.
  - Sie bilden die sogenannte *Standard Library* und dazu gehören
  - `LinearAlgebra`, `Statistics`, `SparseArrays`, `Printf`, `Pkg` und andere.
- Über 9000 Pakete sind offiziell registriert, siehe https://julialang.org/packages/.
  - Diese können mit wenigen Tastendrücken heruntergeladen und installiert werden.
  - Dazu dient der *package manager* `Pkg`.
  - Man kann ihn auf zwei Arten verwenden:
    - als normale Julia-Anweisungen, die auch in einer `.jl`-Programmdatei stehen können:
      ```
      using Pkg
      Pkg.add("PaketXY")
      ```
    - im speziellen pkg-Modus des Julia-REPLs:
      ```
      ] add PaketXY
      ```
  - Anschließend kann das Paket mit `using PaketXY` verwendet werden.
- Man kann auch Pakete aus anderen Quellen und selbstgeschriebene Pakete installieren.

5.4.1 Einige Funktionen des Paketmanagers

Funktion	`pkg` - Mode	Erklärung
`Pkg.add("MyPack")`	`pkg> add MyPack`	add `MyPack.jl` to current environment
`Pkg.rm("MyPack")`	`pkg> remove MyPack`	remove `MyPack.jl` from current environment
`Pkg.update()`	`pkg> update`	update packages in current environment
`Pkg.activate("mydir")`	`pkg> activate mydir`	activate directory as current environment
`Pkg.status()`	`pkg> status`	list packages
`Pkg.instantiate()`	`pg> instantiate`	install all packages according to `Project.toml`

5.4.2 Installierte Pakete und Environments

- Julia und der Paketmanager verwalten
  1. eine Liste der mit dem Kommando `Pkg.add()` bzw. `]add` explizit installierten Pakete mit genauer Versionsbezeichnung in einer Datei `Project.toml` und
  2. eine Liste aller dabei auch als implizite Abhängigkeiten installierten Pakete in der Datei `Manifest.toml`.
- Das Verzeichnis, in dem diese Dateien stehen, ist das `environment` und wird mit `Pkg.status()` bzw. `]status` angezeigt.
- Im Normalfall sieht das so aus:

```
(@v1.8) pkg> status
     Status `~/.julia/environments/v1.8/Project.toml`
  [1dea7af3] OrdinaryDiffEq v6.7.1
  [91a5bcdd] Plots v1.27.1
  [438e738f] PyCall v1.93.1
```

- Man kann für verschiedene Projekte eigene `environments` benutzen. Dazu kann man entweder Julia mit

```
julia --project=path/to/myproject
```

starten oder in Julia das environment mit `Pkg.activate("path/to/myproject")` aktivieren. Dann werden `Project.toml, Manifest.toml` dort angelegt und verwaltet. (Die Installation der Paketdateien erfolgt weiterhin irgendwo unter `$HOME/.julia`)

5.4.3 Zum Installieren von Paketen auf unserem Jupyter-Server `misun103`:

- Es gibt ein zentrales Repository, in dem alle in diesem Kurs erwähnten Pakete bereits installiert sind.
- Dort haben Sie keine Schreibrechte.
- Sie können aber zusätzliche Pakete in Ihrem `HOME` installieren. Dazu ist als erster Befehl nötig, das aktuelle Verzeichnis zu aktivieren:

```
] activate .
```

(Man beachte den Punkt!)

Danach können Sie mit `add` im Pkg-Modus auch Pakete installieren:

```
] add PaketXY
```

Achtung! Das kann dauern! Viele Pakete haben komplexe Abhängigkeiten und lösen die Installation von weiteren Paketen aus. Viele Pakete werden beim Installieren vorkompiliert. Im REPL sieht man den Installationsfortschritt, im Jupyter-Notebook leider nicht.

5.5 Der Julia JIT *(just in time)* Compiler: Introspection

Julia baut auf die Werkzeuge des *LLVM Compiler Infrastructure Projects* auf.

Stages of Compilation

stage & result	introspection command
Parse $\Longrightarrow$ Abstract Syntax Tree (AST)	`Meta.parse()`
Lowering: transform AST $\Longrightarrow$ Static Single Assignment (SSA) form	`@code_lowered`
Type Inference	`@code_warntype`, `@code_typed`
Generate LLVM intermediate representation	`@code_llvm`
Generate native machine code	`@code_native`

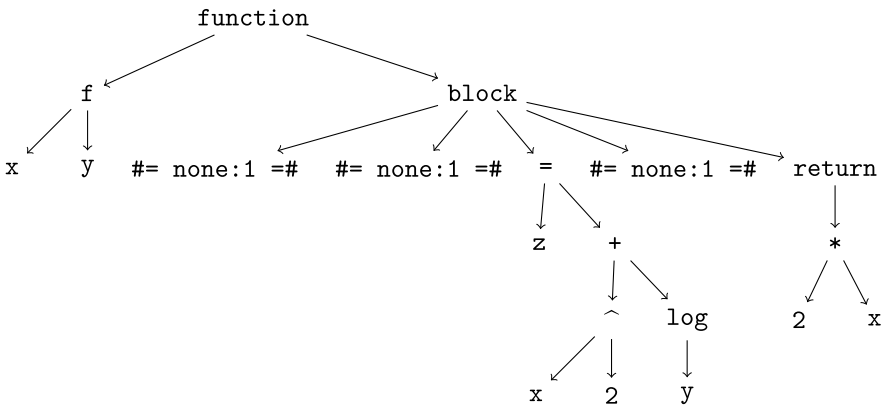
```
function f(x,y)
    z = x^2 + log(y)
    return 2z
end
```

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

```
p = Meta.parse( "function f(x,y); z=x^2+log(y); return 2x; end  ")
```

```
:(function f(x, y)
      #= none:1 =#
      #= none:1 =#
      z = x ^ 2 + log(y)
      #= none:1 =#
      return 2x
  end)
```

```
using TreeView

walk_tree(p)
```




```
@code_lowered f(2,4)
```

```
CodeInfo(
1 — %1 = Main.:^
│   %2 = Core.apply_type(Base.Val, 2)
│   %3 = (%2)()
│   %4 = Base.literal_pow(%1, x, %3)
│   %5 = Main.log(y)
│        z = %4 + %5
│   %7 = 2 * z
└──      return %7
)
```

```
@code_warntype f(2,4)
```

```
MethodInstance for f(::Int64, ::Int64)
  from f(x, y) @ Main In[2]:1
Arguments
  #self#::Core.Const(f)
  x::Int64
  y::Int64
Locals
  z::Float64
Body::Float64
1 — %1 = Main.:^::Core.Const(^)
│   %2 = Core.apply_type(Base.Val, 2)::Core.Const(Val2)
│   %3 = (%2)()::Core.Const(Val2())
│   %4 = Base.literal_pow(%1, x, %3)::Int64
│   %5 = Main.log(y)::Float64
│        (z = %4 + %5)
│   %7 = (2 * z)::Float64
└──      return %7
```

```
@code_typed f(2,4)
```

```
CodeInfo(
1 — %1 = Base.mul_int(x, x)::Int64
```

```
│   %2 = Base.sitofp(Float64, y)::Float64
│   %3 = invoke Base.Math._log(%2::Float64, $(QuoteNode(Val:e())))::Val:e, :log::Symbol)::Float64
│   %4 = Base.sitofp(Float64, %1)::Float64
│   %5 = Base.add_float(%4, %3)::Float64
│   %6 = Base.mul_float(2.0, %5)::Float64
└──      return %6
) => Float64
```

```
@code_llvm f(2,4)
```

```
;  @ In[2]:1 within `f`
define double @julia_f_1836(i64 signext %0, i64 signext %1) #0 top:;  @ In[2]:2 within `f`; ┌ @ intfuncs.jl:33
```

```
@code_native f(2,4)
```

```
    .text
    .file   "f"
    .section    .rodata.cst8,"aM",@progbits,8
    .p2align    3                               # -- Begin function julia_f_1874
.LCPI0_0:
    .quad   0x4000000000000000              # double 2
    .text
    .globl  julia_f_1874
    .p2align    4, 0x90
    .type   julia_f_1874,@function
julia_f_1874:                           # @julia_f_1874
; ┌ @ In[2]:1 within `f`
# %bb.0:                                # %top
    push rbp
    mov  rbp, rsp
    sub  rsp, 16
; │ @ In[2]:2 within `f`
; │┌ @ intfuncs.jl:332 within `literal_pow`
; ││┌ @ int.jl:88 within `*`
    imul rdi, rdi
    mov  qword ptr [rbp - 8], rdi        # 8-byte Spill
; │└└
; │┌ @ math.jl:1576 within `log`
; ││┌ @ float.jl:294 within `float`
; │││┌ @ float.jl:268 within `AbstractFloat`
; ││││┌ @ float.jl:159 within `Float64`
                                        # implicit-def: $xmm0
    vcvtsi2sd    xmm0, xmm0, rsi
; ││└└└
; ││ @ math.jl:1578 within `log` @ special/log.jl:267
    movabs   rax, offset j__log_1876
    movabs   rdi, 140181127206600
    call rax
    mov  rdi, qword ptr [rbp - 8]        ▯▯▯90m# 8-byte Reload
    vmovaps  xmm1, xmm0
; │└
; │┌ @ promotion.jl:422 within `+`
; ││┌ @ promotion.jl:393 within `promote`
; │││┌ @ promotion.jl:370 within `_promote`
; ││││┌ @ number.jl:7 within `convert`
; │││││┌ @ float.jl:159 within `Float64`
                                        # implicit-def: $xmm0
```

```
    vcvtsi2sd    xmm0, xmm0, rdi
; ||└┘└┘
; ||  @ promotion.jl:422 within `+` @ float.jl:409
    vaddsd   xmm1, xmm0, xmm1
; │└
; │  @ In[2]:3 within `f`
; │┌ @ promotion.jl:423 within `*` @ float.jl:411
    movabs   rax, offset .LCPI0_0
    vmovsd   xmm0, qword ptr [rax]           # xmm0 = mem[0],zero
    vmulsd   xmm0, xmm0, xmm1
; │└
    add  rsp, 16
    pop  rbp
    ret
.Lfunc_end0:
    .size   julia_f_1874, .Lfunc_end0-julia_f_1874
; └
                                        # -- End function
    .section    ".note.GNU-stack","",@progbits
```

6 Maschinenzahlen

```
for x ∈ ( 3, 3.3e4,  Int16(20), Float32(3.3e4), UInt16(9) )
    @show x sizeof(x) typeof(x)
    println()
end
```

```
x = 3
sizeof(x) = 8
typeof(x) = Int64

x = 33000.0
sizeof(x) = 8
typeof(x) = Float64

x = 20
sizeof(x) = 2
typeof(x) = Int16

x = 33000.0f0
sizeof(x) = 4
typeof(x) = Float32

x = 0x0009
sizeof(x) = 2
typeof(x) = UInt16
```

6.1 Ganze Zahlen *(integers)*

Ganze Zahlen werden grundsätzlich als Bitmuster fester Länge gespeichert. Damit ist der Wertebereich endlich. **Innerhalb dieses Wertebereichs** sind Addition, Subtraktion, Multiplikation und die Ganzzahldivision mit Rest exakte Operationen ohne Rundungsfehler.

Ganze Zahlen gibt es in den zwei Sorten `Signed` (mit Vorzeichen) und `Unsigned`, die man als Maschinenmodelle für $\mathbb{Z}$ bzw. $\mathbb{N}$ auffassen kann.

6.1.1 *Unsigned integers*

```
subtypes(Unsigned)
```

```
5-element Vector{Any}:
 UInt128
 UInt16
 UInt32
 UInt64
 UInt8
```

UInts sind Binärzahlen mit n=8, 16, 32, 64 oder 128 Bits Länge und einem entsprechenden Wertebereich von

$$0 \leq x < 2^n$$

Sie werden im *scientific computing* eher selten verwendet. Bei hardwarenaher Programmierung dienen sie z.B. dem Umgang mit Binärdaten und Speicheradressen. Deshalb werden sie von Julia standardmäßig auch als Hexadezimalzahlen (mit Präfix `0x` und Ziffern `0-9a-f`) dargestellt.

```
x = 0x0033efef
@show x typeof(x) Int(x)

z = UInt(32)
@show z typeof(z);
```

```
x = 0x0033efef
typeof(x) = UInt32
Int(x) = 3403759
z = 0x0000000000000020
typeof(z) = UInt64
```

6.1.2 *Signed Integers*

```
subtypes(Signed)
```

```
6-element Vector{Any}:
 BigInt
 Int128
 Int16
 Int32
 Int64
 Int8
```

Integers haben den Wertebereich

$$-2^{n-1} \leq x < 2^{n-1}$$

In Julia sind ganze Zahlen standardmäßig 64 Bit groß:

```
x = 42
typeof(x)
```

```
Int64
```

Sie haben daher den Wertebereich:

$$-9.223.372.036.854.775.808 \leq x \leq 9.223.372.036.854.775.807$$

32-Bit-Integers haben den Wertebereich

$$-2.147.483.648 \leq x \leq 2.147.483.647$$

Der Maximalwert $2^{31} - 1$ is zufällig gerade eine Mersenne-Primzahl:

```
using Primes
isprime(2^31-1)
```

```
true
```

Negative Zahlen werden im Zweierkomplement dargestellt:

$x \Rightarrow -x$ entspricht: *flip all bits, then add 1*

Das sieht so aus:

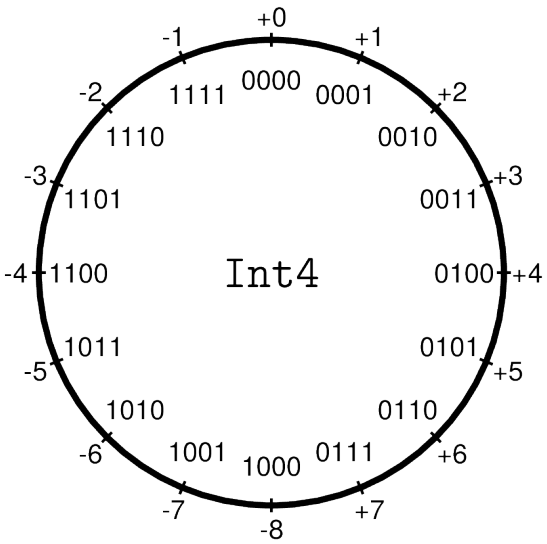


Abbildung 6.1: Eine Darstellung des fiktiven Datentyps `Int4`

Das höchstwertige Bit zeigt das Vorzeichen an. Sein „Umkippen“ entspricht allerdings nicht der Operation `x → -x`.

> **!** Wichtig
>
> Die gesamte Ganzzahlarithmetik ist eine **Arithmetik modulo** 2^n
>
> ```
> x = 2^62 - 10 + 2^62
> ```
>
> ```
> 9223372036854775798
> ```
>
> ```
> x + 20
> ```
>
> ```
> -9223372036854775798
> ```
>
> Keine Fehlermeldung, keine Warnung! Ganzzahlen fester Länge liegen nicht auf einer Geraden, sondern auf einem Kreis!

Schauen wir uns ein paar *Integers* an:

```
using Printf

for i in (1, 2, 7, -7, 2^50, -1, Int16(7), Int8(7))
    s = bitstring(i)
    t = typeof(i)
    @printf "%20i %-5s ↠ %s\n" i t s
end
```

```
                   1 Int64 ↠ 0000000000000000000000000000000000000000000000000000000000000001
                   2 Int64 ↠ 0000000000000000000000000000000000000000000000000000000000000010
                   7 Int64 ↠ 0000000000000000000000000000000000000000000000000000000000000111
                  -7 Int64 ↠ 1111111111111111111111111111111111111111111111111111111111111001
     1125899906842624 Int64 ↠ 0000000000000100000000000000000000000000000000000000000000000000
                  -1 Int64 ↠ 1111111111111111111111111111111111111111111111111111111111111111
```

```
7 Int16 ⇒  0000000000000111
7 Int8  ⇒  00000111
```

Julia hat Funktionen, die über die Datentypen informieren (*introspection*):

```
typemin(Int64), typemax(Int64)
```

```
(-9223372036854775808, 9223372036854775807)
```

```
typemin(UInt64), typemax(UInt64), BigInt(typemax(UInt64))
```

```
(0x0000000000000000, 0xffffffffffffffff, 18446744073709551615)
```

```
typemin(Int8), typemax(Int8)
```

```
(-128, 127)
```

6.2 Arithmetik ganzer Zahlen

Addition, Multiplikation

Die Operationen `+`,`-`,`*` haben die übliche exakte Arithmetik **modulo** 2^{64}.

Potenzen `a^b`

- Potenzen `a^n` werden für natürliche Exponenten `n` ebenfalls modulo 2^{64} exakt berechnet.
- Für negative Exponenten ist das Ergebnis eine Gleitkommazahl.
- `0^0` ist selbstverständlich gleich 1.

```
(-2)^63, 2^64, 3^(-3), 0^0
```

```
(-9223372036854775808, 0, 0.037037037037037035, 1)
```

- Für natürliche Exponenten wird *exponentiation by squaring* verwendet, so dass z.B. `x^23` nur 7 Multiplikationen benötigt:

$$x^{23} = \Big(\big((x^2)^2 \cdot x\big)^2 \cdot x\Big)^2 \cdot x$$

Division

- Division `/` erzeugt eine Gleitkommazahl.

```
x = 40/5
```

```
8.0
```

Ganzzahldivision und Rest

- Die Funktionen `div(a,b)`, `rem(a,b)` und `divrem(a,b)` berechnen den Quotient der Ganzzahldivision, den dazugehörigen Rest *(remainder)* bzw. beides als Tupel.
- Für `div(a,b)` gibt es die Operatorform `a ÷ b` (Eingabe: `\div<TAB>`) und für `rem(a,b)` die Operatorform `a % b`.
- Standardmäßig wird bei der Division „zur Null hin gerundet“, wodurch der dazugehörige Rest dasselbe Vorzeichen wie der Dividend `a` trägt:

```
@show divrem( 27, 4)
@show ( 27 ÷  4,  27 %  4)
@show (-27 ÷  4, -27 %  4 )
@show ( 27 ÷ -4,  27 % -4);
```

```
divrem(27, 4) = (6, 3)
(27 ÷ 4, 27 % 4) = (6, 3)
(-27 ÷ 4, -27 % 4) = (-6, -3)
(27 ÷ -4, 27 % -4) = (-6, 3)
```

- Eine von `RoundToZero` abweichende Rundungsregel kann bei den Funktionen als optionales 3. Argument angegeben werden.
- `?RoundingMode` zeigt die möglichen Rundungsregeln an.
- Für die Rundungsregel `RoundDown` („in Richtung minus unendlich”), wodurch der dazugehörige Rest dasselbe Vorzeichen wie der Divisor `b` bekommt, gibt es auch die Funktionen `fld(a,b)` *(floored division)* und `mod(a,b)`:

```
@show divrem(-27, 4, RoundDown)
@show (fld(-27,  4), mod(-27,  4))
@show (fld( 27, -4), mod( 27, -4));
```

```
divrem(-27, 4, RoundDown) = (-7, 1)
(fld(-27, 4), mod(-27, 4)) = (-7, 1)
(fld(27, -4), mod(27, -4)) = (-7, -1)
```

Für alle Rundungsregeln gilt:

```
div(a, b, RoundingMode) * b + rem(a, b, RoundingMode) = a
```

Der Datentyp `BigInt`

Der Datentyp `BigInt` ermöglicht Ganzzahlen beliebiger Länge. Der benötigte Speicher wird dynamisch allokiert.

Numerische Konstanten haben automatisch einen ausreichend großen Typ:

```
z = 10
@show typeof(z)
z = 10_000_000_000_000_000    # 10 Billiarden
@show typeof(z)
z = 10_000_000_000_000_000_000  # 10 Trillionen
@show typeof(z)
z = 10_000_000_000_000_000_000_000_000_000_000_000_000_000   # 10 Sextilliarden
@show typeof(z);
```

```
typeof(z) = Int64
typeof(z) = Int64
typeof(z) = Int128
typeof(z) = BigInt
```

Meist wird man allerdings den Datentyp `BigInt` explizit anfordern müssen, damit nicht modulo 2^{64} gerechnet wird:

```
@show 3^300       BigInt(3)^300;
```

```
3 ^ 300 = 4157753088978724465
BigInt(3) ^ 300 = 136891479058588375991326027382088315966463695625337436471480190078368997177499076593800206150
```

Arbitrary precision arithmetic kostet einiges an Speicherplatz und Rechenzeit.

Wir vergleichen den Zeit- und Speicherbedarf bei der Aufsummation von 10 Millionen Ganzzahlen als `Int64` und als `BigInt`.

```
# 10^7 Zufallszahlen, gleichverteilt zwischen -10^7 und 10^7
vec_int = rand(-10^7:10^7, 10^7)

# Dieselben Zahlen als BigInts
vec_bigint = BigInt.(vec_int)
```

```
10000000-element Vector{BigInt}:
  1499425
  -283237
  3359545
  -577098
 -4488579
  7320968
 -5500047
 -9533158
  8263174
  2684875
 -5754970
  8260662
  7384818
        ⋮
  7232579
 -8801554
  7502030
 -1289227
  -322341
 -5648907
  7543304
  6316694
 -3868997
   624286
 -1689231
 -2661950
```

Einen ersten Eindruck vom Zeit- und Speicherbedarf gibt das `@time`-Macro:

```
@time x = sum(vec_int)
@show x typeof(x)
```

```
  0.012154 seconds (254 allocations: 11.719 KiB, 67.69% compilation time)
x = -17812326975
typeof(x) = Int64
Int64
```

```
@time x = sum(vec_bigint)
@show x typeof(x);
```

```
  0.166128 seconds (17.82 k allocations: 1.226 MiB, 7.64% compilation time)
x = -17812326975
typeof(x) = BigInt
```

Durch die Just-in-Time-Compilation von Julia ist die einmalige Ausführung einer Funktion wenig aussagekräftig. Das Paket `BenchmarkTools` stellt u.a. das Macro `@benchmark` bereit, das eine Funktion mehrfach aufruft und die Ausführungszeiten als Histogramm darstellt.

```
using BenchmarkTools

@benchmark sum($vec_int)
```

```
BenchmarkTools.Trial: 1103 samples with 1 evaluation.
 Range (min … max):  3.805 ms …  5.334 ms  ┊ GC (min … max): 0.00% … 0.00%
 Time  (median):     4.511 ms              ┊ GC (median):    0.00%
 Time  (mean ± σ):   4.524 ms ± 87.964 μs  ┊ GC (mean ± σ):  0.00% ± 0.00%

  3.8 ms          Histogram: frequency by time         4.77 ms <

 Memory estimate: 0 bytes, allocs estimate: 0.
```

```
@benchmark sum($vec_bigint)
```

```
BenchmarkTools.Trial: 33 samples with 1 evaluation.
 Range (min … max):  153.327 ms … 159.505 ms  ┊ GC (min … max): 0.00% … 0.00%
 Time  (median):     153.417 ms               ┊ GC (median):    0.00%
 Time  (mean ± σ):   153.700 ms ±   1.068 ms  ┊ GC (mean ± σ):  0.00% ± 0.00%

  153 ms           Histogram: frequency by time          160 ms <

 Memory estimate: 48 bytes, allocs estimate: 3.
```

Die `BigInt`-Addition ist mehr als 30 mal langsamer.

6.3 Gleitkommazahlen

Aus *floating point numbers* kann man im Deutschen **[Gleit|Fließ]–[Komma|Punkt]–Zahlen** machen und tatsächlich kommen alle 4 Varianten in der Literatur vor.

In der Numerischen Mathematik spricht man auch gerne von **Maschinenzahlen**.

6.3.1 Grundidee

- Eine „feste Anzahl von Vor- und Nachkommastellen“ ist für viele Probleme ungeeignet.
- Eine Trennung zwischen „gültigen Ziffern“ und Größenordnung (Mantisse und Exponent) wie in der wissenschaftlichen Notation ist wesentlich flexibler.

$$345.2467 \times 10^3 \qquad 34.52467 \times 10^4 \qquad \underline{3.452467 \times 10^5} \qquad 0.3452467 \times 10^6 \qquad 0.03452467 \times 10^7$$

- Zur Eindeutigkeit muss man eine dieser Formen auswählen. In der mathematischen Analyse von Maschinenzahlen wählt man oft die Form, bei der die erste Nachkommastelle ungleich Null ist. Damit gilt für die Mantisse m:

$$1 > m \geq (0.1)_b = b^{-1},$$

wobei b die gewählte Basis des Zahlensystems bezeichnet.

- Wir wählen im Folgenden die Form, die der tatsächlichen Implementation auf dem Computer entspricht und legen fest: Die Darstellung mit genau einer Ziffer ungleich Null vor dem Komma ist die **Normalisierte Darstellung**. Damit gilt
$$(10)_b = b > m \geq 1.$$
- Bei Binärzahlen `1.01101`: ist diese Ziffer immer gleich Eins und man kann auf das Abspeichern dieser Ziffer verzichten. Diese tatsächlich abgespeicherte (gekürzte) Mantisse bezeichnen wir mit M, so dass
$$m = 1 + M$$
gilt.

> **Maschinenzahlen**
>
> Die Menge der Maschinenzahlen $\mathbb{M}(b, p, e_{min}, e_{max})$ ist charakterisiert durch die verwendete Basis b, die Mantissenlänge p und den Wertebereich des Exponenten $\{e_{min}, ..., e_{max}\}$.
> In unserer Konvention hat die Mantisse einer normalisierten Maschinenzahl eine Ziffer (der Basis b) ungleich Null vor dem Komma und $p - 1$ Nachkommastellen.
> Wenn $b = 2$ ist, braucht man daher nur $p - 1$ Bits zur Speicherung der Mantisse normalisierter Gleitkommazahlen.
> Der Standard IEEE 754, der von der Mahrzahl der modernen Prozessoren und Programmiersprachen implementiert wird, definiert
>
> - `Float32` als $\mathbb{M}(2, 24, -126, 127)$ und
> - `Float64` als $\mathbb{M}(2, 53, -1022, 1023)$.

6.3.2 Aufbau von `Float64` nach Standard IEEE 754

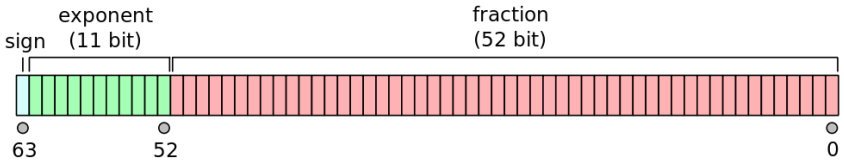


Abbildung 6.2: Aufbau einer `Float64` (Quelle: Codekaizen, CC BY-SA 4.0, via Wikimedia Commons)

- 1 Vorzeichenbit S
- 11 Bits für den Exponenten, also $0 \leq E \leq 2047$
- die Werte $E = 0$ und $E = (11111111111)_2 = 2047$ sind reserviert für die Codierung von Spezialwerten wie $\pm 0, \pm\infty$, NaN *(Not a Number)* und denormalisierte Zahlen.
- 52 Bits für die (gekürzte) Mantisse M, $\quad 0 \leq M < 1$, das entspricht etwa 16 Dezimalstellen
- Damit wird folgende Zahl dargestellt:
$$x = (-1)^S \cdot (1 + M) \cdot 2^{E-1023}$$

Ein Beispiel:

```
x = 27.56640625
bitstring(x)
```

```
"0100000000111011100100010000000000000000000000000000000000000000"
```

Das geht auch schöner:

```
function printbitsf64(x::Float64)
    s = bitstring(x)
    printstyled(s[1], color = :blue, reverse=true)
```

```
    printstyled(s[2:12], color = :green, reverse=true)
    printstyled(s[13:end], color=:red, bold=true, reverse=true)
    print("\n")
end

printbitsf64(x)
```

```
0100000000111011100100010000000000000000000000000000000000000000
```

und wir können S (in blau),E (grün) und M (rot) ablesen:

$$
\begin{aligned}
S &= 0 \\
E &= (10000000011)_2 = 2^{10} + 2^1 + 2^0 = 1027 \\
M &= (.1011100100001)_2 = \frac{1}{2} + \frac{1}{2^3} + \frac{1}{2^4} + \frac{1}{2^5} + \frac{1}{2^8} + \frac{1}{2^{12}} \\
x &= (-1)^S \cdot (1 + M) \cdot 2^{E-1023}
\end{aligned}
$$

... und damit die Zahl rekonstruieren:

```
x = (1 + 1/2 + 1/8 + 1/16 + 1/32 + 1/256 + 1/4096) * 2^4
```

```
27.56640625
```

- Die Maschinenzahlen $\mathbb{M}$ bilden eine endliche, diskrete Untermenge von $\mathbb{R}$. Es gibt eine kleinste und eine größte Maschinenzahl und abgesehen davon haben alle x∈$\mathbb{M}$ einen Vorgänger und Nachfolger in $\mathbb{M}$.
- Was ist der Nachfolger von x in $\mathbb{M}$? Dazu setzen wir das kleinste Mantissenbit von 0 auf 1.
- Die Umwandlung eines Strings aus Nullen und Einsen in die entsprechende Maschinenzahl ist z.B. so möglich:

```
ux = parse(UInt64, "0100000000111011100100010000000000000000000000000000000000000001", base=2)
reinterpret(Float64, ux)
```

```
27.566406250000004
```

Das kann man in Julia allerdings auch einfacher haben:

```
y = nextfloat(x)
```

```
27.566406250000004
```

Der Vorgänger von x ist:

```
z = prevfloat(x)
println(z)
printbitsf64(z)
```

```
27.566406249999996
0100000000111011100100001111111111111111111111111111111111111111
```

6.4 Maschinenepsilon

- Den Abstand zwischen `1` und dem Nachfolger `nextfloat(1)` nennt man **Maschinenepsilon**.
- Für `Float64` mit einer Mantissenlänge von 52 Bit ist $\epsilon = 2^{-52}$.

```
@show nextfloat(1.) - 1   2^-52   eps(Float64);
```

```
nextfloat(1.0) - 1 = 2.220446049250313e-16
2 ^ -52 = 2.220446049250313e-16
eps(Float64) = 2.220446049250313e-16
```

- Das Maschinenepsilon ist ein Maß für den relativen Abstand zwischen den Maschinenzahlen und quantifiziert die Aussage: „64-Bit-Gleitkommazahlen haben eine Genauigkeit von etwa 16 Dezimalstellen."
- Das Maschinenepsilon ist etwas völlig anderes als die kleinste positive Gleitkommazahl:

```
floatmin(Float64)
```

```
2.2250738585072014e-308
```

- Ein Teil der Literatur verwendet eine andere Definition des Maschinenepsilons, die halb so groß ist.

$$\epsilon' = \frac{\epsilon}{2} \approx 1.1 \times 10^{-16}$$

  ist der maximale relative Fehler, der beim Runden einer reellen Zahl auf die nächste Maschinenzahl entstehen kann.
- Da Zahlen aus dem Intervall $(1 - \epsilon', 1 + \epsilon']$ auf die Maschinenzahl 1 gerundet werden, kann man ϵ' auch definieren als: *die größte Zahl, für die in der Maschinenzahlarithmetik noch gilt:* $1 + \epsilon' = 1$.

Auf diese Weise kann man das Maschinenepsilon auch berechnen:

```
Eps = 1
while(1 != 1 + Eps)
    Eps /= 2
    println(1+Eps)
end
Eps
```

```
1.5
1.25
1.125
1.0625
1.03125
1.015625
1.0078125
1.00390625
1.001953125
1.0009765625
1.00048828125
1.000244140625
1.0001220703125
1.00006103515625
1.000030517578125
1.0000152587890625
1.0000076293945312
1.0000038146972656
1.0000019073486328
1.0000009536743164
1.0000004768371582
1.000000238418579
1.0000001192092896
1.0000000596046448
1.0000000298023224
1.0000000149011612
1.0000000074505806
1.0000000037252903
1.0000000018626451
```

```
1.0000000009313226
1.0000000004656613
1.0000000002328306
1.0000000001164153
1.0000000000582077
1.0000000000291038
1.000000000014552
1.000000000007276
1.000000000003638
1.000000000001819
1.0000000000009095
1.0000000000004547
1.0000000000002274
1.0000000000001137
1.0000000000000568
1.0000000000000284
1.0000000000000142
1.000000000000007
1.0000000000000036
1.0000000000000018
1.0000000000000009
1.0000000000000004
1.0000000000000002
1.0
1.1102230246251565e-16
```

oder als Bitmuster:

```
Eps=1
while 1 != 1 + Eps
    Eps/= 2
    printbitsf64(1+Eps)
end
Eps
```

```
0011111111111000000000000000000000000000000000000000000000000000
0011111111111010000000000000000000000000000000000000000000000000
0011111111111001000000000000000000000000000000000000000000000000
0011111111111000100000000000000000000000000000000000000000000000
0011111111111000010000000000000000000000000000000000000000000000
0011111111111000001000000000000000000000000000000000000000000000
0011111111111000000100000000000000000000000000000000000000000000
0011111111111000000010000000000000000000000000000000000000000000
0011111111111000000001000000000000000000000000000000000000000000
0011111111111000000000100000000000000000000000000000000000000000
0011111111111000000000010000000000000000000000000000000000000000
0011111111111000000000001000000000000000000000000000000000000000
0011111111111000000000000100000000000000000000000000000000000000
0011111111111000000000000010000000000000000000000000000000000000
0011111111111000000000000001000000000000000000000000000000000000
0011111111111000000000000000100000000000000000000000000000000000
0011111111111000000000000000010000000000000000000000000000000000
0011111111111000000000000000001000000000000000000000000000000000
0011111111111000000000000000000100000000000000000000000000000000
0011111111111000000000000000000010000000000000000000000000000000
0011111111111000000000000000000001000000000000000000000000000000
0011111111111000000000000000000000100000000000000000000000000000
0011111111111000000000000000000000010000000000000000000000000000
0011111111111000000000000000000000001000000000000000000000000000
0011111111111000000000000000000000000100000000000000000000000000
```

```
0011111111110000000000000000000000000100000000000000000000000000000
0011111111110000000000000000000000000010000000000000000000000000000
0011111111110000000000000000000000000001000000000000000000000000000
0011111111110000000000000000000000000000100000000000000000000000000
0011111111110000000000000000000000000000010000000000000000000000000
0011111111110000000000000000000000000000001000000000000000000000000
0011111111110000000000000000000000000000000100000000000000000000000
0011111111110000000000000000000000000000000010000000000000000000000
0011111111110000000000000000000000000000000001000000000000000000000
0011111111110000000000000000000000000000000000100000000000000000000
0011111111110000000000000000000000000000000000010000000000000000000
0011111111110000000000000000000000000000000000001000000000000000000
0011111111110000000000000000000000000000000000000100000000000000000
0011111111110000000000000000000000000000000000000010000000000000000
0011111111110000000000000000000000000000000000000001000000000000000
0011111111110000000000000000000000000000000000000000100000000000000
0011111111110000000000000000000000000000000000000000010000000000000
0011111111110000000000000000000000000000000000000000001000000000000
0011111111110000000000000000000000000000000000000000000100000000000
0011111111110000000000000000000000000000000000000000000010000000000
0011111111110000000000000000000000000000000000000000000001000000000
0011111111110000000000000000000000000000000000000000000000100000000
0011111111110000000000000000000000000000000000000000000000010000000
0011111111110000000000000000000000000000000000000000000000001000000
0011111111110000000000000000000000000000000000000000000000000100000
0011111111110000000000000000000000000000000000000000000000000010000
0011111111110000000000000000000000000000000000000000000000000001000
0011111111110000000000000000000000000000000000000000000000000000100
0011111111110000000000000000000000000000000000000000000000000000010
0011111111110000000000000000000000000000000000000000000000000000001
0011111111110000000000000000000000000000000000000000000000000000000
1.1102230246251565e-16
```

Die Menge der (normalisierten) Maschinenzahlen

- Im Intervall $[1, 2)$ liegen 2^{52} äquidistante Maschinenzahlen.
- Danach erhöht sich der Exponent um 1 und die Mantisse M wird auf 0 zurückgesetzt. Damit enthält das Intervall $[2, 4)$ wiederum 2^{52} äquidistante Maschinenzahlen, ebenso das Intervall $[4, 8)$ bis hin zu $[2^{1023}, 2^{1024})$.
- Ebenso liegen in den Intervallen $[\frac{1}{2}, 1)$, $[\frac{1}{4}, \frac{1}{2}), \ldots$ je 2^{52} äquidistante Maschinenzahlen, bis hinunter zu $[2^{-1022}, 2^{-1021})$.
- Dies bildet die Menge $\mathbb{M}_+$ der positiven Maschinenzahlen und es ist

$$\mathbb{M} = -\mathbb{M}_+ \cup \{0\} \cup \mathbb{M}_+$$

Die größte und die kleinste positive normalisiert darstellbare Gleitkommazahl eines Gleitkommatyps kann man abfragen:

```
@show floatmax(Float64)
printbitsf64(floatmax(Float64))

@show floatmin(Float64)
printbitsf64(floatmin(Float64))
```

```
floatmax(Float64) = 1.7976931348623157e308
0111111111101111111111111111111111111111111111111111111111111111
floatmin(Float64) = 2.2250738585072014e-308
0000000000010000000000000000000000000000000000000000000000000000
```

6.5 Runden auf Maschinenzahlen

- Die Abbildung rd: $\mathbb{R} \to \mathbb{M}$ soll zur nächstgelegenen darstellbaren Zahl runden.
- Standardrundungsregel: *round to nearest, ties to even*
  Wenn man genau die Mitte zwischen zwei Maschinenzahlen trifft *(tie)*, wählt man die, deren letztes Mantissenbit 0 ist.
- Begründung: damit wird statistisch in 50% der Fälle auf- und in 50% der Fälle abgerundet und so ein „statistischer Drift“ bei längeren Rechnungen vermieden.

- Es gilt:

$$\frac{|x - \mathrm{rd}(x)|}{|x|} \leq \frac{1}{2}\epsilon$$

6.6 Maschinenzahlarithmetik

Die Maschinenzahlen sind als Untermenge von $\mathbb{R}$ nicht algebraisch abgeschlossen. Schon die Summe zweier Maschinenzahlen wird in der Regel keine Maschinenzahl sein.

> **!** Wichtig
>
> Der Standard IEEE 754 fordert, dass die Maschinenzahlarithmetik das *gerundete exakte Ergebnis* liefert:
> Das Resultat muss gleich demjenigen sein, das bei einer exakten Ausführung der entsprechenden Operation mit anschließender Rundung entsteht.
>
> $$a \oplus b = \mathrm{rd}(a + b)$$
>
> Analoges muss für die Implemetierung der Standardfunktionen wie wie `sqrt()`, `log()`, `sin()` ... gelten: Sie liefern ebenfalls die Maschinenzahl, die dem exakten Ergebnis am nächsten kommt.

Die Arithmetik ist *nicht assoziativ*:

```
1 +  10^-16 + 10^-16
```

```
1.0
```

```
1 + (10^-16 + 10^-16)
```

```
1.0000000000000002
```

Im ersten Fall (ohne Klammern) wird von links nach rechts ausgewertet:

$$\begin{aligned} 1 \oplus 10^{-16} \oplus 10^{-16} &= (1 \oplus 10^{-16}) \oplus 10^{-16} \\ &= \mathrm{rd}\left(\mathrm{rd}(1 + 10^{-16}) + 10^{-16}\right) \\ &= \mathrm{rd}(1 + 10^{-16}) \\ &= 1 \end{aligned}$$

Im zweiten Fall erhält man:

$$\begin{aligned} 1 \oplus (10^{-16} \oplus 10^{-16}) &= \mathrm{rd}\left(1 + \mathrm{rd}(10^{-16} + 10^{-16})\right) \\ &= \mathrm{rd}(1 + 2 * 10^{-16}) \\ &= 1.0000000000000002 \end{aligned}$$

Es sei auch daran erinnert, dass sich selbst „einfache“ Dezimalbrüche häufig nicht exakt als Maschinenzahlen darstellen lassen:

$$(0.1)_{10} = (0.0001100110011001100110011001100...)_2 = (0.000\overline{1100})_2$$
$$(0.3)_{10} = (0.0100110011001100110011001100..)_2 = (0.0\overline{1001})_2$$

```
printbitsf64(0.1)
printbitsf64(0.3)
```

```
0011111110111001100110011001100110011001100110011001100110011010
0011111111010011001100110011001100110011001100110011001100110011
```

Folge:

```
0.1 + 0.1 == 0.2
```

```
true
```

```
0.2 + 0.1 == 0.3
```

```
false
```

```
0.2 + 0.1
```

```
0.30000000000000004
```

Bei der Ausgabe einer Maschinenzahl muss der Binärbruch in einen Dezimalbruch entwickelt werden. Man kann sich auch mehr Stellen dieser Dezimalbruchentwicklung anzeigen lassen:

```
using Printf
@printf("%.30f", 0.1)
```

```
0.100000000000000005551115123126
```

```
@printf("%.30f", 0.3)
```

```
0.299999999999999988897769753748
```

Die Binärbruch-Mantisse einer Maschinenzahl kann eine lange oder sogar unendlich-periodische Dezimalbruchentwicklung haben. Dadurch sollte man sich nicht eine „höheren Genauigkeit“ suggerieren lassen!

> **!** Wichtig
>
> Moral: wenn man `Floats` auf Gleichheit testen will, sollte man fast immer eine dem Problem angemessene realistische Genauigkeit `epsilon` festlegen und darauf testen:
>
> ```
> epsilon = 1.e-10
>
> if abs(x-y) < epsilon
>    # ...
> end
> ```

6.7 Normalisierte und Denormalisierte Maschinenzahlen

Die Lücke zwischen Null und der kleinsten normalisierten Maschinenzahl $2^{-1022} \approx 2.22 \times 10^{-308}$ ist mit denormalisierten Maschinenzahlen besiedelt.

Zum Verständnis nehmen wir ein einfaches Modell:

- Sei $\mathbb{M}(10,4,\pm5)$ die Menge der Maschinenzahlen zur Basis 10 mit 4 Mantissenstellen (eine vor dem Komma, 3 Nachkommastellen) und dem Exponentenbereich $-5 \leq E \leq 5$.
- Dann ist die normalisierte Darstellung (Vorkommastelle ungleich 0) von z.B. 1234.0 gleich 1.234e3 und von 0.00789 gleich 7.890e-3.
- Es ist wichtig, dass die Maschinenzahlen bei jedem Rechenschritt normalisiert gehalten werden. Nur so wird die Mantissenlänge voll ausgenutzt und die Genauigkeit ist maximal.
- Die kleinste positive normalisierte Zahl in unserem Modell ist `x = 1.000e-5`. Schon `x/2` müsste auf 0 gerundet werden.
- Hier erweist es sich für viele Anwendungen als günstiger, auch denormalisierte *(subnormal)* Zahlen zuzulassen und `x/2` als `0.500e-5` oder `x/20` als `0.050e-5` darzustellen.
- Dieser *gradual underflow* ist natürlich mit einem Verlust an gültigen Stellen und damit Genauigkeit verbunden.

Im `Float`-Datentyp werden solche *subnormal values* dargestellt durch ein Exponentenfeld, in dem alle Bits gleich Null sind:

```
using Printf

x = 2 * floatmin(Float64)   # 2*kleinste normalisierte Gleitkommazahl > 0

while x != 0
    x /= 2
    @printf "%14.6g  " x
    printbitsf64(x)
end
```

```
  2.22507e-308  0000000000010000000000000000000000000000000000000000000000000000
  1.11254e-308  0000000000001000000000000000000000000000000000000000000000000000
  5.56268e-309  0000000000000100000000000000000000000000000000000000000000000000
  2.78134e-309  0000000000000010000000000000000000000000000000000000000000000000
  1.39067e-309  0000000000000001000000000000000000000000000000000000000000000000
  6.95336e-310  0000000000000000100000000000000000000000000000000000000000000000
  3.47668e-310  0000000000000000010000000000000000000000000000000000000000000000
  1.73834e-310  0000000000000000001000000000000000000000000000000000000000000000
  8.69169e-311  0000000000000000000100000000000000000000000000000000000000000000
  4.34585e-311  0000000000000000000010000000000000000000000000000000000000000000
  2.17292e-311  0000000000000000000001000000000000000000000000000000000000000000
  1.08646e-311  0000000000000000000000100000000000000000000000000000000000000000
  5.43231e-312  0000000000000000000000010000000000000000000000000000000000000000
  2.71615e-312  0000000000000000000000001000000000000000000000000000000000000000
  1.35808e-312  0000000000000000000000000100000000000000000000000000000000000000
  6.79039e-313  0000000000000000000000000010000000000000000000000000000000000000
  3.39519e-313  0000000000000000000000000001000000000000000000000000000000000000
   1.6976e-313  0000000000000000000000000000100000000000000000000000000000000000
  8.48798e-314  0000000000000000000000000000010000000000000000000000000000000000
  4.24399e-314  0000000000000000000000000000001000000000000000000000000000000000
    2.122e-314  0000000000000000000000000000000100000000000000000000000000000000
    1.061e-314  0000000000000000000000000000000010000000000000000000000000000000
  5.30499e-315  0000000000000000000000000000000001000000000000000000000000000000
  2.65249e-315  0000000000000000000000000000000000100000000000000000000000000000
  1.32625e-315  0000000000000000000000000000000000010000000000000000000000000000
  6.63124e-316  0000000000000000000000000000000000001000000000000000000000000000
  3.31562e-316  0000000000000000000000000000000000000100000000000000000000000000
  1.65781e-316  0000000000000000000000000000000000000010000000000000000000000000
  8.28905e-317  0000000000000000000000000000000000000001000000000000000000000000
  4.14452e-317  0000000000000000000000000000000000000000100000000000000000000000
  2.07226e-317  0000000000000000000000000000000000000000010000000000000000000000
  1.03613e-317  0000000000000000000000000000000000000000001000000000000000000000
  5.18065e-318  0000000000000000000000000000000000000000000100000000000000000000
  2.59033e-318  0000000000000000000000000000000000000000000010000000000000000000
```

```
 1.29516e-318  0000000000001000000000000000000000000000000000000000000000000000
 6.47582e-319  0000000000000100000000000000000000000000000000000000000000000000
 3.23791e-319  0000000000000010000000000000000000000000000000000000000000000000
 1.61895e-319  0000000000000001000000000000000000000000000000000000000000000000
 8.09477e-320  0000000000000000100000000000000000000000000000000000000000000000
 4.04739e-320  0000000000000000010000000000000000000000000000000000000000000000
 2.02369e-320  0000000000000000001000000000000000000000000000000000000000000000
 1.01185e-320  0000000000000000000100000000000000000000000000000000000000000000
 5.05923e-321  0000000000000000000010000000000000000000000000000000000000000000
 2.52962e-321  0000000000000000000001000000000000000000000000000000000000000000
 1.26481e-321  0000000000000000000000100000000000000000000000000000000000000000
 6.32404e-322  0000000000000000000000010000000000000000000000000000000000000000
 3.16202e-322  0000000000000000000000001000000000000000000000000000000000000000
 1.58101e-322  0000000000000000000000000100000000000000000000000000000000000000
 7.90505e-323  0000000000000000000000000010000000000000000000000000000000000000
 3.95253e-323  0000000000000000000000000001000000000000000000000000000000000000
 1.97626e-323  0000000000000000000000000000100000000000000000000000000000000000
 9.88131e-324  0000000000000000000000000000010000000000000000000000000000000000
 4.94066e-324  0000000000000000000000000000001000000000000000000000000000000000
            0  0000000000000000000000000000000000000000000000000000000000000000
```

6.8 Spezielle Werte

Die Gleitkommaarithmetik kennt einige spezielle Werte, z.B.

```
nextfloat(floatmax(Float64))
```

```
Inf
```

```
for x ∈ (NaN, Inf, -Inf, -0.0)
    @printf "%6g  " x
    printbitsf64(x)
end
```

```
   NaN  0111111111111000000000000000000000000000000000000000000000000000
   Inf  0111111111110000000000000000000000000000000000000000000000000000
  -Inf  1111111111110000000000000000000000000000000000000000000000000000
    -0  1000000000000000000000000000000000000000000000000000000000000000
```

- Ein Exponentenüberlauf *(overflow)* führt zum Ergebnis `Inf` oder `-Inf`.

```
2/0, -3/0, floatmax(Float64) * 1.01, exp(1300)
```

```
(Inf, -Inf, Inf, Inf)
```

- Damit kann weitergerechnet werden:

```
-Inf + 20, Inf/30, 23/-Inf, sqrt(Inf),  Inf * 0, Inf - Inf
```

```
(-Inf, Inf, -0.0, Inf, NaN, NaN)
```

- `NaN` *(Not a Number)* steht für das Resultat einer Operation, das undefiniert ist. Alle weiteren Operationen mit `NaN` ergeben ebenfalls `NaN`.

```
0/0, Inf - Inf, 2.3NaN, sqrt(NaN)
```

```
(NaN, NaN, NaN, NaN)
```

- Da `NaN` einen undefinierten Wert repräsentiert, ist es zu nichts gleich, nichtmal zu sich selbst. Das ist sinnvoll, denn wenn zwei Variablen `x` und `y` als `NaN` berechnet wurden, sollte man nicht schlußfolgern, dass sie gleich sind.
- Zum Testen auf `NaN` gibt es daher die boolsche Funktion `isnan()`.

```
x = 0/0
y = Inf - Inf
@show x==y   NaN==NaN  isfinite(NaN) isinf(NaN) isnan(x) isnan(y);
```

```
x == y = false
NaN == NaN = false
isfinite(NaN) = false
isinf(NaN) = false
isnan(x) = true
isnan(y) = true
```

- Es gibt eine „minus Null“. Sie signalisiert einen Exponentenunterlauf *(underflow)* einer betragsmäßig zu klein gewordenen *negativen* Größe.

```
@show 23/-Inf   -2/exp(1200)    -0.0==0.0;
```

```
23 / -Inf = -0.0
-2 / exp(1200) = -0.0
-0.0 == 0.0 = true
```

6.9 Mathematische Funktionen

Julia verfügt über die üblichen mathematischen Funktionen
`sqrt, exp, log, log2, log10, sin, cos,..., asin, acos,..., sinh,..., gcd, lcm, factorial,...,abs, max, min,...,`

darunter z.B. die Rundungsfunktionen

- `floor(T,x)` = $\lfloor x \rfloor$
- `ceil(T,x)` = $\lceil x \rceil$

```
floor(3.4),   floor(Int64, 3.5),   floor(Int64, -3.5)
```

```
(3.0, 3, -4)
```

```
ceil(3.4),    ceil(Int64, 3.5),    ceil(Int64, -3.5)
```

```
(4.0, 4, -3)
```

Es sei noch hingewiesen auf `atan(y, x)`, den Arkustangens mit 2 Argumenten, Er ist in anderen Programmiersprachen oft als Funktion mit eigenem Namen *atan2* implementiert. Dieser löst das Problem der Umrechnung von kartesischen in Polarkoordinaten ohne lästige Fallunterscheidung.

- `atan(y,x)` ist Winkel der Polarkoordinaten von (x,y) im Intervall $(-\pi, \pi]$. Im 1. und 4. Quadranten ist er also gleich `atan(y/x)`

```
atan(3, -2),    atan(-3, 2),    atan(-3/2)
```

```
(2.1587989303424644, -0.982793723247329, -0.982793723247329)
```

6.10 Umwandlung Strings $\Longleftrightarrow$ Zahlen

Die Umwandlung ist mit den Funktionen `parse()` und `string()` möglich.

```
parse(Int64, "1101", base=2)
```

```
13
```

```
string(13, base=2)
```

```
"1101"
```

```
string(1/7)
```

```
"0.14285714285714285"
```

```
string(77, base=16)
```

```
"4d"
```

Zur Umwandlung der numerischen Typen ineinander kann man die Typnamen verwenden. Typenamen sind auch Konstruktoren:

```
x = Int8(67)
@show x   typeof(x);
```

```
x = 67
typeof(x) = Int8
```

```
z = UInt64(3459)
```

```
0x0000000000000d83
```

```
y = Float64(z)
```

```
3459.0
```

6.11 Literatur

- D. Goldberg, What Every Computer Scientist Should Know About Floating-Point Arithmetic
- C. Vuik, Some Disasters caused by numerical errors

7 Ein Beispiel zur Stabilität von Gleitkommaarithmetik

7.1 Berechnung von π nach Archimedes

Eine untere Schranke für 2π, den Umfang des Einheitskreises, erhält man durch die Summe der Seitenlängen eines dem Einheitskreis eingeschriebenen regelmäßigen n-Ecks. Die Abbildung links zeigt, wie man beginnend mit einem Viereck der Seitenlänge $s_4 = \sqrt{2}$ die Eckenzahl iterativ verdoppelt.

Abb. 1

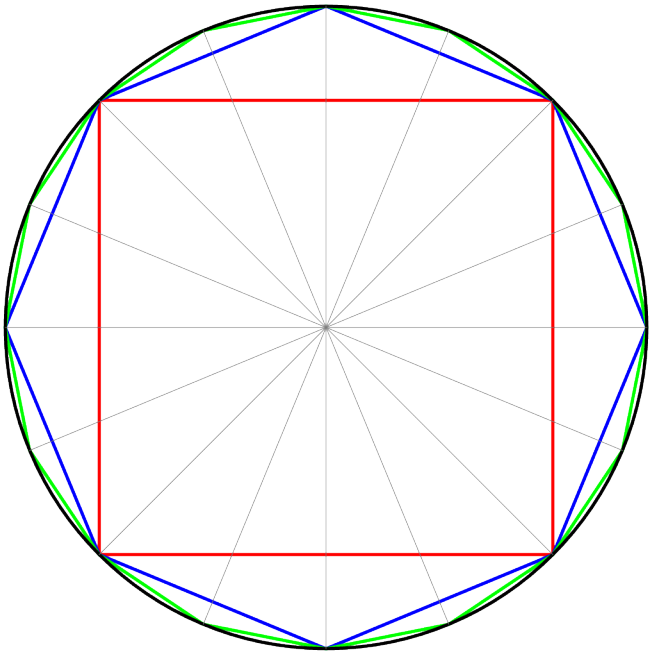
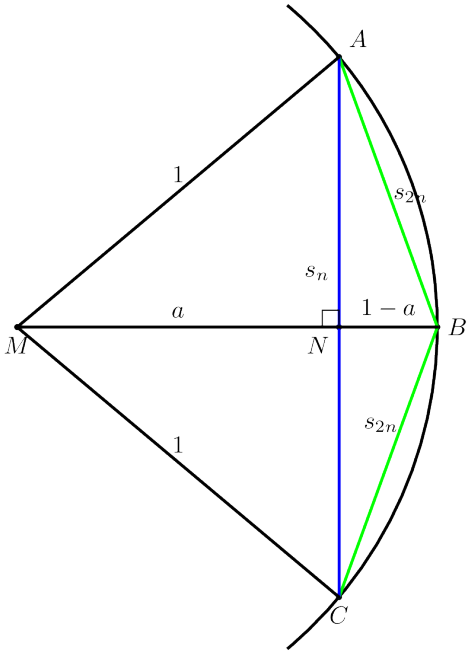


Abb.2



Die zweite Abbildung zeigt die Geometrie der Eckenverdoppelung.

Mit $|\overline{AC}| = s_n$, $|\overline{AB}| = |\overline{BC}| = s_{2n}$, $|\overline{MN}| = a$, $|\overline{NB}| = 1 - a$, liefert Pythagoras für die Dreiecke MNA und NBA jeweils

$$a^2 + \left(\frac{s_n}{2}\right)^2 = 1 \qquad \text{bzw.} \qquad (1-a)^2 + \left(\frac{s_n}{2}\right)^2 = s_{2n}^2$$

Elimination von a liefert die Rekursion

$$s_{2n} = \sqrt{2 - \sqrt{4 - s_n^2}} \qquad \text{mit Startwert} \qquad s_4 = \sqrt{2}$$

für die Länge s_n **einer** Seite des eingeschriebenen regelmäßigen n-Ecks.

Die Folge $(n \cdot s_n)$ konvergiert monoton von unten gegen den Grenzwert 2π:

$$n\, s_n \to 2\pi$$

Der relative Fehler der Approximation von 2π durch ein n-Eck ist:

$$\epsilon_n = \left| \frac{n\, s_n - 2\pi}{2\pi} \right|$$

7.2 Zwei Iterationsvorschriften[1]

Die Gleichung

$$s_{2n} = \sqrt{2 - \sqrt{4 - s_n^2}} \qquad \text{(Iteration A)}$$

ist mathematisch äquivalent zu

$$s_{2n} = \frac{s_n}{\sqrt{2 + \sqrt{4 - s_n^2}}} \qquad \text{(Iteration B)}$$

(Bitte nachrechnen!)

Allerdings ist Iteration A schlecht konditioniert und numerisch instabil, wie der folgende Code zeigt. Ausgegeben wird die jeweils berechnete Näherung für π.

```
using Printf

iterationA(s) =  sqrt(2 - sqrt(4 - s^2))
iterationB(s) = s / sqrt(2 + sqrt(4 - s^2))

s_B = s_A = sqrt(2)    # Startwert

ϵ(x) = abs(x - 2π)/2π  # rel. Fehler

ϵ_A = Float64[]   # Vektoren für den Plot
ϵ_B = Float64[]
is  = Float64[]

@printf """                          approx. Wert von π
         n-Eck       Iteration A            Iteration B
"""

for i in 3:35
    push!(is, i)
    s_A = iterationA(s_A)
    s_B = iterationB(s_B)
    doublePi_A = 2^i * s_A
    doublePi_B = 2^i * s_B
    push!(ϵ_A, ϵ(doublePi_A))
    push!(ϵ_B, ϵ(doublePi_B))
    @printf "%14i %20.15f  %20.15f\n" 2^i doublePi_A/2 doublePi_B/2
end
```

```
                    approx. Wert von π
         n-Eck       Iteration A            Iteration B
             8    3.061467458920719     3.061467458920718
            16    3.121445152258053     3.121445152258052
            32    3.136548490545941     3.136548490545939
            64    3.140331156954739     3.140331156954753
           128    3.141277250932757     3.141277250932773
           256    3.141513801144145     3.141513801144301
           512    3.141572940367883     3.141572940367092
```

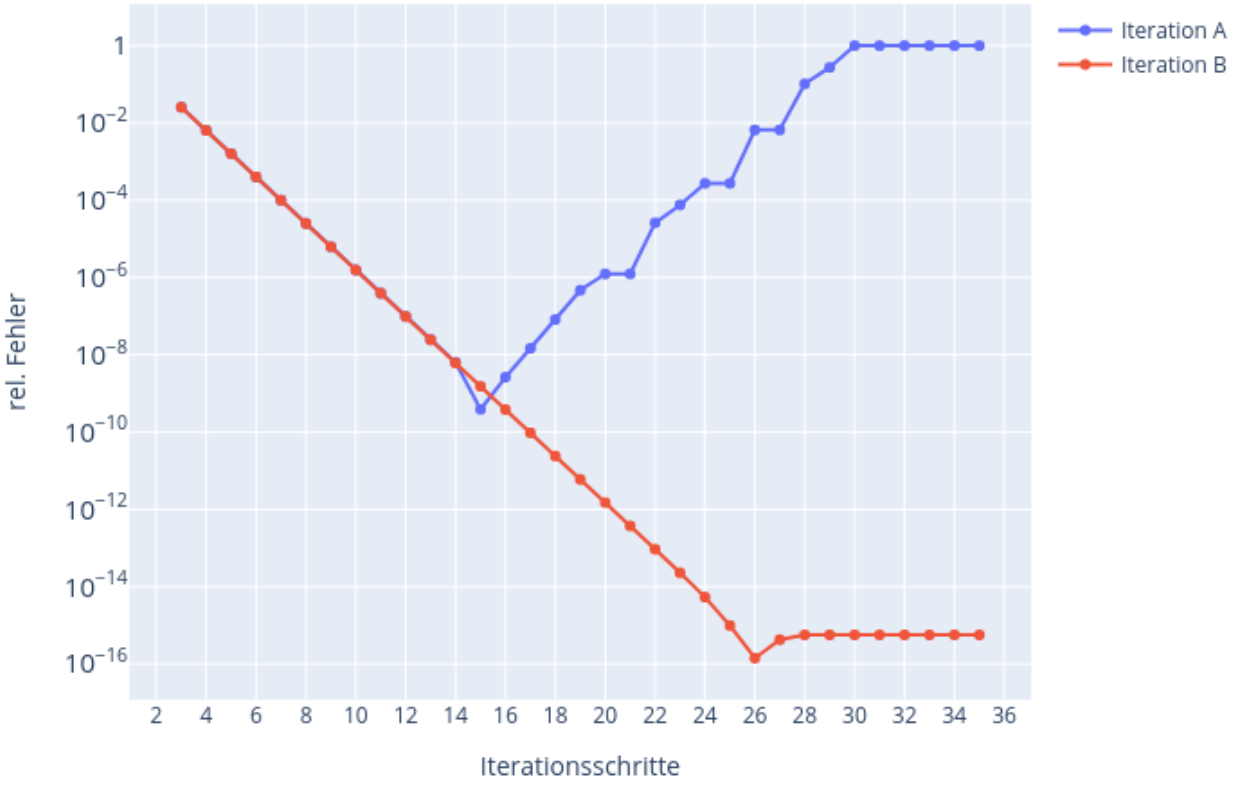
[1] nach: Christoph Überhuber, „Computer-Numerik“ Bd. 1, Springer 1995, Kap. 2.3

1024	3.141587725279961	3.141587725277160
2048	3.141591421504635	3.141591421511200
4096	3.141592345611077	3.141592345570118
8192	3.141592576545004	3.141592576584873
16384	3.141592633463248	3.141592634338564
32768	3.141592654807589	3.141592648776986
65536	3.141592645321215	3.141592652386592
131072	3.141592607375720	3.141592653288993
262144	3.141592910939673	3.141592653514594
524288	3.141594125195191	3.141592653570994
1048576	3.141596553704820	3.141592653585094
2097152	3.141596553704820	3.141592653588619
4194304	3.141674265021758	3.141592653589501
8388608	3.141829681889202	3.141592653589721
16777216	3.142451272494134	3.141592653589776
33554432	3.142451272494134	3.141592653589790
67108864	3.162277660168380	3.141592653589794
134217728	3.162277660168380	3.141592653589794
268435456	3.464101615137754	3.141592653589795
536870912	4.000000000000000	3.141592653589795
1073741824	0.000000000000000	3.141592653589795
2147483648	0.000000000000000	3.141592653589795
4294967296	0.000000000000000	3.141592653589795
8589934592	0.000000000000000	3.141592653589795
17179869184	0.000000000000000	3.141592653589795
34359738368	0.000000000000000	3.141592653589795

Während Iteration B sich stabilisiert bei einem innerhalb der Maschinengenauigkeit korrekten Wert für π, wird Iteration A schnell instabil. Ein Plot der relativen Fehler ϵ_i bestätigt das:

```
using PlotlyJS

layout = Layout(xaxis_title="Iterationsschritte", yaxis_title="rel. Fehler",
            yaxis_type="log", yaxis_exponentformat="power",
            xaxis_tick0=2, xaxis_dtick=2)

plot([scatter(x=is, y=ϵ_A, mode="markers+lines", name="Iteration A", yscale=:log10),
      scatter(x=is, y=ϵ_B, mode="markers+lines", name="Iteration B", yscale=:log10)],
      layout)
```

```
┌ Warning: attempting to remove probably stale pidfile
│   path = "/home/hellmund/.jlassetregistry.lock"
└ @ Pidfile ~/.julia/packages/Pidfile/DDu3M/src/Pidfile.jl:260
```



7.3 Stabilität und Auslöschung

Bei $i = 26$ erreicht Iteration B einen relativen Fehler in der Größe des Maschinenepsilons:

```
ϵ_B[22:28]
```

```
7-element Vector{Float64}:
 5.3716034620272725e-15
 9.895059008997609e-16
 1.4135798584282297e-16
 4.240739575284689e-16
 5.654319433712919e-16
 5.654319433712919e-16
 5.654319433712919e-16
```

Weitere Iterationen verbessern das Ergebnis nicht mehr. Sie stabilisieren sich bei einem relativen Fehler von etwa 2.5 Maschinenepsilon:

```
ϵ_B[end]/eps(Float64)
```

```
2.5464790894703255
```

Die Form Iteration A ist instabil. Bereits bei $i = 16$ beginnt der relative Fehler wieder zu wachsen.

Ursache ist eine typische Auslöschung. Die Seitenlängen s_n werden sehr schnell klein. Damit ist $a_n = \sqrt{4 - s_n^2}$ nur noch wenig kleiner als 2 und bei der Berechnung von $s_{2n} = \sqrt{2 - a_n}$ tritt ein typischer Auslöschungseffekt auf.

```
setprecision(80) # precision für BigFloat

s = sqrt(BigFloat(2))

@printf "     a = √(4-s^2) als BigFloat       und als Float64\n\n"

for i = 3:44
    s = iterationA(s)
    x = sqrt(4-s^2)
    if i > 20
        @printf "%i %30.26f %20.16f \n" i  x  Float64(x)
    end
end
```

```
      a = √(4-s^2) als BigFloat       und als Float64

21   1.99999999999775591177215422   1.9999999999977560
22   1.99999999999943897794303856   1.9999999999994389
23   1.99999999999985974448576005   1.9999999999998597
24   1.99999999999996493612143919   1.9999999999999649
25   1.99999999999999123403035980   1.9999999999999913
26   1.99999999999999780850758995   1.9999999999999978
27   1.99999999999999945212689707   1.9999999999999996
28   1.99999999999999986303172344   1.9999999999999998
29   1.99999999999999996575793045   2.0000000000000000
30   1.99999999999999999143948303   2.0000000000000000
31   1.99999999999999999785987034   2.0000000000000000
32   1.99999999999999999946496800   2.0000000000000000
33   1.99999999999999999986624159   2.0000000000000000
34   1.99999999999999999996656040   2.0000000000000000
35   1.99999999999999999999163886   2.0000000000000000
36   1.99999999999999999999790889   2.0000000000000000
37   1.99999999999999999999947722   2.0000000000000000
38   1.99999999999999999999986931   2.0000000000000000
39   1.99999999999999999999996691   2.0000000000000000
40   1.99999999999999999999999173   2.0000000000000000
41   1.99999999999999999999999835   2.0000000000000000
42   1.99999999999999999999999835   2.0000000000000000
43   1.99999999999999999999999835   2.0000000000000000
44   1.99999999999999999999999835   2.0000000000000000
```

Man sieht die Abnahme der Zahl der signifikanten Ziffern. Man sieht auch, dass eine Verwendung von `BigFloat` mit einer Mantissenlänge von hier 80 Bit das Einsetzen des Auslöschungseffekts nur etwas hinaussschieben kann.

Gegen instabile Algorithmen helfen in der Regel nur bessere, stabile Algorithmen und nicht genauere Maschinenzahlen!

8 Das Typsystem von Julia

Man kann umfangreiche Programme in Julia schreiben, ohne auch nur eine einzige Typdeklaration verwenden zu müssen. Das ist natürlich Absicht und soll die Arbeit der Anwender vereinfachen.

Wir blicken jetzt trotzdem mal unter die Motorhaube.

8.1 Die Typhierarchie am Beispiel der numerischen Typen

Das Typsystem hat die Struktur eines Baums, dessen Wurzel der Typ `Any` ist. Mit den Funktionen `subtypes()` und `supertype()` kann man den Baum erforschen. Sie zeigen alle Kinder bzw. die Mutter eines Knotens an.

```
subtypes(Int64)
```

```
Type[]
```

Das Ergebnis ist eine leere Liste von Typen. `Int64` ist ein sogenannter **konkreter Typ** und hat keine Untertypen.

Wir klettern jetzt mal die Typhierarchie auf diesem Ast nach oben bis zur Wurzel (Informatiker-Bäume stehen bekanntlich immer auf dem Kopf).

```
supertype(Int64)
```

```
Signed
```

```
supertype(Signed)
```

```
Integer
```

```
supertype(Integer)
```

```
Real
```

```
supertype(Real)
```

```
Number
```

```
supertype(Number)
```

```
Any
```

Das wäre übrigens auch schneller gegangen: Die Funktion `supertypes()` (mit Plural-s) zeigt alle Vorfahren an.

```
supertypes(Int64)
```

```
(Int64, Signed, Integer, Real, Number, Any)
```

Nun kann man sich die Knoten angucken:

```
subtypes(Real)
```

```
4-element Vector{Any}:
 AbstractFloat
 AbstractIrrational
 Integer
 Rational
```

Mit einer kleinen rekursiven Funktion kann man schnell einen ganzen (Unter-)Baum ausdrucken:

```
function show_subtype_tree(T, i=0)
    println("       "^i, T)
    for Ts ∈ subtypes(T)
        show_subtype_tree(Ts, i+1)
    end
end

show_subtype_tree(Number)
```

```
Number
       Complex
       Real
              AbstractFloat
                     BigFloat
                     Float16
                     Float32
                     Float64
              AbstractIrrational
                     Irrational
              Integer
                     Bool
                     Signed
                            BigInt
                            Int128
                            Int16
                            Int32
                            Int64
                            Int8
                     Unsigned
                            UInt128
                            UInt16
                            UInt32
                            UInt64
                            UInt8
              Rational
```

Hier das Ganze nochmal als Bild (gemacht mit LaTeX/TikZ)

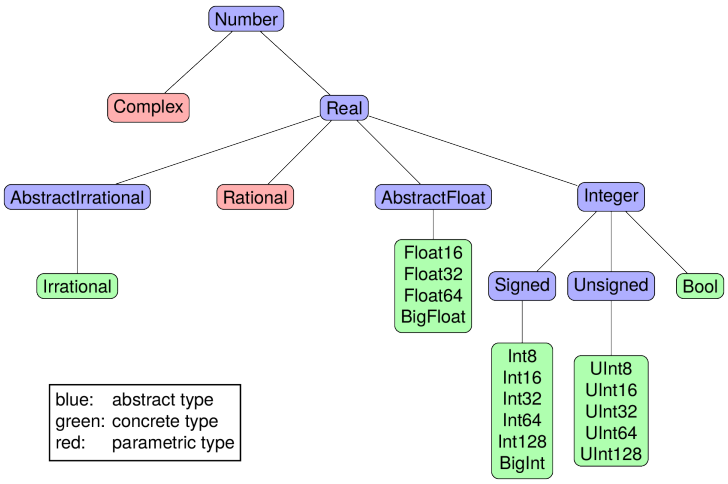


Abbildung 8.1: Die Hierarchie der numerischen Typen

Natürlich hat Julia nicht nur numerische Typen. Die Anzahl der direkten Abkömmlinge (Kinder) von `Any` ist

```
length(subtypes(Any))
```

```
650
```

und mit (fast) jedem Paket, das man mit `using ...` lädt, werden es mehr.

8.2 Abstrakte und Konkrete Typen

- Ein Objekt hat immer einen **konkreten** Typ.
- Konkrete Typen haben keine Untertypen mehr, sie sind immer „Blätter“ des Baumes.
- Konkrete Typen spezifizieren eine konkrete Datenstruktur.

- Abstrakte Typen können nicht instanziiert werden, d.h., es gibt keine Objekte mit diesem Typ.
- Sie definieren eine Menge von konkreten Typen und gemeinsame Methoden für diese Typen.
- Sie können daher in der Definition von Funktionstypen, Argumenttypen, Elementtypen von zusammengesetzten Typen u.ä. verwendet werden.

Zum **Deklarieren** *und* **Testen** der “Abstammung” innerhalb der Typhierarchie gibt es einen eigenen Operator:

```
Int64 <: Number
```

```
true
```

Zum Testen, ob ein Objekt einen bestimmten Typ (oder einen abstrakten Supertyp davon) hat, dient `isa(object, typ)`. Es wird meist in der Infix-Form verwendet und sollte als Frage `x is a T?` gelesen werden.

```
x = 17.2

42 isa Int64,  42 isa Real,  x isa Real,   x isa Float64,   x isa Integer
```

```
(true, true, true, true, false)
```

Da abstrakte Typen keine Datenstrukturen definieren, ist ihre Definition recht schlicht. Entweder sie stammen direkt von `Any` ab:

```
abstract type MySuperType end

supertype(MySuperType)
```

```
Any
```

oder von einem anderen abstrakten Typ:

```
abstract type MySpecialNumber <: Integer end

supertypes(MySpecialNumber)
```

```
(MySpecialNumber, Integer, Real, Number, Any)
```

8.3 Die numerischen Typen `Bool` und `Irrational`

Da sie im Baum der numerischen Typen zu sehen sind, seien sie kurz erklärt:

`Bool` ist numerisch im Sinne von `true=1, false=0`:

```
true + true + true, false - true, sqrt(true), true/4
```

```
(3, -1, 1.0, 0.25)
```

`Irrational` ist der Typ einiger vordefinierter Konstanten wie π und *e*. Laut Dokumentation ist `Irrational` ein *"Number type representing an exact irrational value, which is automatically rounded to the correct precision in arithmetic operations with other numeric quantities".*

8.4 Union-Typen

Falls die Baum-Hierarchie nicht ausreicht, kann man auch abstrakte Typen als Vereinigung beliebiger (abstrakter und konkreter) Typen definieren.

```
IntOrString = Union{Int64,String}
```

```
Union{Int64, String}
```

Beispiel

Das Kommando `methods(<)` zeigt, dass unter den über 70 Methoden, die für den Vergleichsoperator definiert sind, einige auch *union types* verwenden, z.B. ist

```
<(x::Union{Float16, Float32, Float64}, y::BigFloat)
```

eine Methode für den Vergleich einer Maschinenzahl fester Länge mit einer Maschinenzahl beliebiger Länge.

8.5 Zusammengesetzte (*composite*) Typen: `struct`

Eine `struct` ist eine Zusammenstellung von mehreren benannten Feldern und definiert einen konkreten Typ.

```
abstract type Point end

mutable struct Point2D <: Point
```

```
    x :: Float64
    y :: Float64
end

mutable struct Point3D <: Point
    x :: Float64
    y :: Float64
    z :: Float64
end
```

Wie wir schon bei Ausdrücken der Form `x = Int8(33)` gesehen haben, kann man Typnamen direkt als Konstruktoren einsetzen:

```
p1 = Point2D(1.4, 3.5)
```

```
Point2D(1.4, 3.5)
```

```
p1 isa Point3D,  p1 isa Point2D,   p1 isa Point
```

```
(false, true, true)
```

Die Felder einer `struct` können über ihren Namen mit dem `.`-Operator adressiert werden.

```
p1.y
```

```
3.5
```

Da wir unsere `struct` als `mutable` deklariert haben, können wir das Objekt `p1` modifizieren, indem wir den Feldern neue Werte zuweisen.

```
p1.x = 3333.4
p1
```

```
Point2D(3333.4, 3.5)
```

Informationen über den Aufbau eines Typs oder eines Objekts von diesem Typ liefert `dump()`.

```
dump(Point3D)
```

```
Point3D <: Point
  x::Float64
  y::Float64
  z::Float64
```

```
 dump(p1)
```

```
Point2D
  x: Float64 3333.4
  y: Float64 3.5
```

8.6 Funktionen und *Multiple dispatch*

> **Objekte, Funktionen, Methoden**
>
> In klassischen objektorientierten Sprachen wie C++/Java haben Objekte üblicherweise mit ihnen assoziierte Funktionen, die Methoden des Objekts.
> In Julia gehören Methoden zu einer Funktion und nicht zu einem Objekt. (Eine Ausnahme sind die Konstruktoren, also Funktionen, die genauso heißen wie ein Typ und ein Objekt dieses Typs erzeugen.)
> Sobald man einen neuen Typ definiert hat, kann man sowohl neue als auch bestehende Funktionen um neue Methoden für diesen Typ ergänzen.
>
> - Eine Funktion kann mehrfach für verschiedene Argumentlisten (Typ und Anzahl) definiert werden.
> - Die Funktion hat dann mehrere Methoden.
> - Beim Aufruf wird an Hand der konkreten Argumente entschieden, welche Methode genutzt wird *(multiple dispatch)*.
> - Es ist typisch für Julia, dass für Standardfunktionen viele Methoden definiert sind. Diese können problemlos um weitere Methoden für eigene Typen erweitert werden.

Den Abstand zwischen zwei Punkten implementieren wir als Funktion mit zwei Methoden:

```
function distance(p1::Point2D, p2::Point2D)
    sqrt((p1.x-p2.x)^2 + (p1.y-p2.y)^2)
end

function distance(p1::Point3D, p2::Point3D)
    sqrt((p1.x-p2.x)^2 + (p1.y-p2.y)^2 + (p1.z-p2.z)^2)
end
```

```
distance (generic function with 2 methods)
```

```
distance(p1, Point2D(2200, -300))
```

```
1173.3319266090054
```

Wie schon erwähnt, zeigt `methods()` die Methodentabelle einer Funktion an:

```
methods(distance)
```

```
# 2 methods for generic function "distance" from Main:
 [1] distance(p1::Point3D, p2::Point3D)
     @ In[23]:5
 [2] distance(p1::Point2D, p2::Point2D)
     @ In[23]:1
```

Das Macro `@which`, angewendet auf einen vollen Funktionsaufruf mmit konkreter Argumentliste, zeigt an, welche Methode zu diesen konkreten Argumenten ausgewählt wird:

```
@which sqrt(3.3)
```

```
sqrt(x::UnionFloat32, Float64)
     @ Base.Math math.jl:685
```

```
z = "Hallo" * '!'
println(z)

@which "Hallo" * '!'
```

```
Hallo!
*(s1::UnionAbstractChar, AbstractString, ss::UnionAbstractChar, AbstractString...)
     @ Base strings/basic.jl:260
```

Methoden können auch abstrakte Typen als Argument haben:

```
"""
  Berechnet den Winkel ϕ (in Grad) der Polarkoordinaten (2D) bzw.
    Kugelkoordinaten (3D) eines Punktes
"""
function phi_winkel(p::Point)
    atand(p.y, p.x)
end

phi_winkel(p1)
```

```
0.0601593431937626
```

> **Tipp**
>
> Ein in *triple quotes* eingeschlossene Text unmittelbat vor der Funktionsdefinition wird automatisch in die Hilfe-Datenbank von Julia integriert:
>
> ```
> ?phi_winkel
> ```
>
> ```
> search: phi_winkel
> ```
>
> Berechnet den Winkel ϕ (in Grad) der Polarkoordinaten (2D) bzw. Kugelkoordinaten (3D) eines Punktes

Beim *multiple dispatch* wird die Methode angewendet, die unter allen passenden die spezifischste ist. Hier eine Funktion mit mehreren Methoden (alle bis auf die letzte in der kurzen *assignment form* geschrieben):

```
f(x::String, y::Number)  = "Args: String + Zahl"
f(x::String, y::Int64)   = "Args: String + Int64"
f(x::Number, y::Int64)   = "Args: Zahl   + Int64"
f(x::Int64,  y:: Number) = "Args: Int64  + Zahl"
f(x::Number)             = "Arg: eine Zahl"

function f(x::Number, y::Number, z::String)
    return "Arg: 2 x Zahl + String"
end
```

```
f (generic function with 6 methods)
```

Hier passen die ersten beiden Methoden. Gewählt wird die zweite, da sie spezifischer ist, `Int64 <: Number`.

```
f("Hallo", 42)
```

```
"Args: String + Int64"
```

Es kann sein, dass diese Vorschrift zu keinem eindeutigen Ergebnis führt, wenn man seine Methoden schlecht gewählt hat.

```
f(42, 42)
```

```
LoadError: MethodError: f(::Int64, ::Int64) is ambiguous.

Candidates:
  f(x::Int64, y::Number)
    @ Main In[30]:4
  f(x::Number, y::Int64)
    @ Main In[30]:3
```

```
Possible fix, define
  f(::Int64, ::Int64)


Stacktrace:
 [1] top-level scope
   @ In[32]:1
```

8.7 Parametrisierte numerische Typen: `Rational` und `Complex`

- Für rationale Zahlen (Brüche) verwendet Julia `//` als Infix-Konstruktor:

```
@show Rational(23, 17)    4//16 + 1//3;
```

```
Rational(23, 17) = 23//17
4 // 16 + 1 // 3 = 7//12
```

- Die imaginäre Einheit $\sqrt{-1}$ heißt `im`

```
@show Complex(0.4)     23 + 0.5im/(1-2im);
```

```
Complex(0.4) = 0.4 + 0.0im
23 + (0.5im) / (1 - 2im) = 22.8 + 0.1im
```

`Rational` und `Complex` bestehen, ähnlich wie unser `Point2D`, aus 2 Feldern: Zähler und Nenner bzw. Real- und Imaginärteil.

Der Typ dieser Felder ist allerdings nicht vollständig festgelegt. `Rational` und `Complex` sind *parametrisierte* Typen.

```
x = 2//7
@show typeof(x);
```

```
typeof(x) = Rational{Int64}
```

```
y = BigInt(2)//7
@show typeof(y)    y^48;
```

```
typeof(y) = Rational{BigInt}
y ^ 48 = 281474976710656//367033682172941254412302110320336601888801
```

```
x = 1 + 2im
typeof(x)
```

```
Complex{Int64}
```

```
y = 1.0 + 2.0im
typeof(y)
```

```
ComplexF64 (alias for ComplexFloat64)
```

Die konkreten Typen `Rational{Int64}`, `Rational{BigInt}`,..., `Complex{Int64}`, `Complex{Float64}`, ... sind Subtypen von `Rational` bzw. `Complex`.

```
Rational{BigInt} <: Rational
```

```
true
```

Die Definitionen sehen etwa so aus:

```
struct MyComplex{T<:Real} <: Number
    re::T
    im::T
end

struct MyRational{T<:Integer} <: Real
    num::T
    den::T
end
```

Die erste Definition besagt:

- `MyComplex` hat zwei Felder `re` und `im`, beide vom gleichen Typ `T`.
- Dieser Typ `T` muss ein Untertyp von `Real` sein.
- `MyComplex` und alle seine Varianten wie `MyComplex{Float64}` sind Untertypen von `Number`.

und die zweite besagt analog:

- `MyRational` hat zwei Felder `num` und `den`, beide vom gleichen Typ `T`.
- Dieser Typ `T` muss ein Untertyp von `Integer` sein.
- `MyRational` und seine Varianten sind Untertypen von `Real`.

Nun ist $\mathbb{Q} \subset \mathbb{R}$, oder auf julianisch `Rational <: Real`. Also können die Komponenten einer komplexen Zahl auch rational sein:

```
z = 3//4 + 5im
dump(z)
```

```
Complex{Rational{Int64}}
  re: Rational{Int64}
    num: Int64 3
    den: Int64 4
  im: Rational{Int64}
    num: Int64 5
    den: Int64 1
```

Diese Strukturen sind ohne das `mutable`-Attribut definiert, also *immutable*:

```
x = 2.2 + 3.3im
println("Der Realteil ist: $(x.re)")

x.re = 4.4
```

```
Der Realteil ist: 2.2
LoadError: setfield!: immutable struct of type Complex cannot be changed

Stacktrace:
 [1] setproperty!(x::ComplexF64, f::Symbol, v::Float64)
   @ Base ./Base.jl:41
 [2] top-level scope
   @ In[42]:4
```

Das ist so üblich. Wir betrachten das Objekt `9` vom Typ `Int64` ja auch als unveränderlich. Das Folgende geht natürlich trotzdem:

```
x += 2.2
```

```
4.4 + 3.3im
```

Hier wird ein neues Objekt vom Typ `Complex{Float64}` erzeugt und `x` zur Referenz auf dieses neue Objekt gemacht.

Die Möglichkeiten des Typsystems verleiten leicht zum Spielen. Hier definieren wir eine `struct`, die wahlweise eine Maschinenzahl oder ein Paar von Ganzzahlen enthalten kann:

```
struct MyParms{T <: Union{Float64, Tuple{Int64, Int64}}}
    param::T
end

p1 = MyParms(33.3)
p2 = MyParms( (2, 4) )

@show p1.param  p2.param;
```

```
p1.param = 33.3
p2.param = (2, 4)
```

8.8 Typen als Objekte

(1) Typen sind ebenfalls Objekte. Sie sind Objekte einer der drei "Meta-Typen"

- `Union` (Union-Typen)
- `UnionAll` (parametrisierte Typen)
- `DataType` (alle konkreten und sonstige abstrakte Typen)

```
@show    23779 isa Int64       Int64 isa DataType;
```

```
23779 isa Int64 = true
Int64 isa DataType = true
```

```
@show  2im isa Complex         Complex isa UnionAll;
```

```
2im isa Complex = true
Complex isa UnionAll = true
```

```
@show  2im isa Complex{Int64}     Complex{Int64} isa DataType;
```

```
2im isa Complex{Int64} = true
Complex{Int64} isa DataType = true
```

Diese 3 konkreten "Meta-Typen" sind übrigens Subtypen des abstrakten "Meta-Typen" `Type`.

```
subtypes(Type)
```

```
4-element Vector{Any}:
 Core.TypeofBottom
 DataType
 Union
 UnionAll
```

---

(2) Damit können Typen auch einfach Variablen zugewiesen werden:

```
x3 = Float64
@show   x3(4)      x3 <: Real   x3==Float64  ;
```

```
x3(4) = 4.0
x3 <: Real = true
x3 == Float64 = true
```

> **Hinweis**
>
> Dies zeigt auch, dass die Style-Vorgaben in Julia wie „Typen und Typvariablen starten mit Großbuchstaben, sonstige Variablen und Funktionen werden klein geschrieben." nur Konventionen sind und von der Sprache nicht erzwungen werden. Man sollte sie trotzdem einhalten, um den Code lesbar zu halten.

Wenn man solche Zuweisungen mit `const` für dauerhaft erklärt, entsteht ein *type alias*.

```
const MyCmplxF64 = MyComplex{Float64}

z = MyComplex(1.1, 2.2)
typeof(z)
```

```
MyCmplxF64 (alias for MyComplexFloat64)
```

---

(3) Typen können Argumente von Funktionen sein.

```
function myf(x, S, T)
    if S <: T
        println("$S is subtype of $T")
    end
    return S(x)
end

z = myf(43, UInt16, Real)

@show z  typeof(z);
```

```
UInt16 is subtype of Real
z = 0x002b
typeof(z) = UInt16
```

Wenn man diese Funktion mit Typsignaturen definieren möchte, kann man natürlich

```
function myf(x, S::Type, T::Type) ... end
```

schreiben. Üblicher ist hier die (dazu äquivalente) spezielle Syntax

```
function myf(x, ::Type{S}, ::Type{T}) where {S,T} ... end
```

bei der man in der `where`-Klausel auch noch Einschränkungen an die zulässigen Werte der Typvariablen `S` und `T` stellen kann.

Wie definiere ich eine spezielle Methode von `myf`, die nur aufgerufen werden soll, wenn `S` und `T` gleich `Int64` sind? Das ist folgendermaßen möglich:

```
function myf(x, ::Type{Int64}, ::Type{Int64}) ... end
```

`Type{Int64}` wirkt wie ein "Meta-Typ", dessen einzige Instanz der Typ `Int64` ist.

---

(4) Es gibt zahlreiche Operationen mit Typen als Argumenten. Wir haben schon `<:(T1, T2)`, `supertype(T)`, `supertypes(T)`, `subtypes(T)` gesehen. Erwähnt seien noch `typejoin(T1,T2)` (nächster gemeinsamer Vorfahre im Typbaum) und Tests wie `isconcretetype(T)`, `isabstracttype(T)`, `isstructtype(T)`.

8.9 Invarianz parametrisierter Typen

Kann man in parametrisierten Typen auch nicht-konkrete Typen einsetzen? Gibt es `Complex{AbstractFloat}` oder `Complex{Union{Float32, Int16}}`?

Ja, die gibt es; und es sind konkrete Typen, man kann also Objekte von diesem Typ erzeugen.

```
z5 = Complex{Integer}(2, 0x33)
dump(z5)
```

```
Complex{Integer}
  re: Int64 2
  im: UInt8 0x33
```

Das ist eine heterogene Struktur. Jede Komponente hat einen individuellen Typ `T`, für den `T<:Integer` gilt.

Nun gilt zwar

```
Int64 <: Integer
```

```
true
```

aber es gilt nicht, dass

```
Complex{Int64} <: Complex{Integer}
```

```
false
```

Diese Typen sind beide konkret. Damit können sie in der Typhierarchie von Julia nicht in einer Sub/Supertype-Relation zueinander stehen. Julias parametrisierte Typen sind in der Sprache der theoretischen Informatik **invariant**. (Wenn aus `S<:T` folgen würde, dass auch `ParamType{S} <: ParamType{T}` gilt, würde man von **Kovarianz** sprechen.)

8.10 Generische Funktionen

Der übliche (und in vielen Fällen empfohlene!) Programmierstil in Julia ist das Schreiben generischer Funktionen:

```
function fsinnfrei1(x, y)
    return x * x * y
end
```

```
fsinnfrei1 (generic function with 1 method)
```

Diese Funktion funktioniert sofort mit allen Typen, für die die verwendeten Operationen definiert sind.

```
fsinnfrei1( Complex(2,3), 10),  fsinnfrei1("Hallo", '!')
```

```
(-50 + 120im, "HalloHallo!")
```

Man kann natürlich Typ-Annotationen benutzen, um die Verwendbarkeit einzuschränken oder um unterschiedliche Methoden für unterschiedliche Typen zu implementieren:

```
function fsinnfrei2(x::Number, y::AbstractFloat)
    return x * x * y
end

function fsinnfrei2(x::String, y::String)
    println("Sorry, I don't take strings!")
end


@show fsinnfrei2(18, 2.0)   fsinnfrei2(18, 2);
```

```
fsinnfrei2(18, 2.0) = 648.0
LoadError: MethodError: no method matching fsinnfrei2(::Int64, ::Int64)

Closest candidates are:
  fsinnfrei2(::Number, ::AbstractFloat)
   @ Main In[57]:1


Stacktrace:
 [1] macro expansion
   @ show.jl:1181 [inlined]
 [2] top-level scope
   @ In[57]:10
```

! Wichtig

Explizite Typannotationen sind fast immer irrelevent für die Geschwindigkeit des Codes!
Dies ist einer der wichtigsten *selling points* von Julia.
Sobald eine Funktion zum ersten Mal mit bestimmten Typen aufgerufen wird, wird eine auf diese Argumenttypen spezialisierte Form der Funktion generiert und compiliert. Damit sind generische Funktionen in der Regel genauso schnell, wie die spezialisierten Funktionen, die man in anderen Sprachen schreibt.
Generische Funktionen erlauben die Zusammenarbeit unterschiedlichster Pakete und eine hohe Abstraktion.
Ein einfaches Beispiel: Das Paket `Measurements.jl` definiert einen neuen Datentyp `Measurement`, einen Wert mit Fehler, und die Arithmetik dieses Typs. Damit funktionieren generische Funktionen automatisch:

```
using Measurements

x = 33.56±0.3
y = 2.3±0.02

fsinnfrei1(x, y)
```

```
2590.0 ± 52.0
```

8.11 Typ-Parameter in Funktionsdefinitionen: die `where`-Klausel

Wir wollen eine Funktion schreiben, die für **alle komplexen Integer** (und nur diese) funktioniert, z.B. eine in $\mathbb{Z}[i]$ mögliche Primfaktorzerlegung. Die Definition

```
function isprime(x::Complex{Integer}) ... end
```

liefert nun nicht das Gewünschte, wie wir in Kapitel 8.9 gesehen haben. Die Funktion würde für ein Argument vom Typ `Complex{Int64}` nicht funktionieren, da letzteres kein Subtyp von `Complex{Integer}` ist.

Wir müssen eine Typ-Variable einführen. Dazu dient die `where`-Klausel.

```
function isprime(x::Complex{T}) where {T<:Integer}
    ...
end
```

Das ist zu lesen als:

> „Das Argument x soll von einem der Typen `Complex{T}` sein, wobei die Typvariable `T` irgendein Untertyp von `Integer` sein kann."

Noch ein Beispiel:

```
function kgV(x::Complex{T}, y::Complex{S}) where {T<:Integer, S<:Integer}
    ...
end
```

> Die Argumente x und y können verschiedene Typen haben und beide müssen Subtypen von `Integer` sein.

Wenn es nur eine `where`-Klausel wie im vorletzten Beispiel gibt, kann man die geschweiften Klammern weglassen und

```
function isprime(x::Complex{T}) where T<:Integer
    ...
end
```

schreiben. Das lässt sich noch weiter kürzen zu

```
function isprime(x::Complex{<:Integer})
    ...
end
```

Diese verschiedenen Varianten können verwirrend sein, aber das ist nur Syntax.

```
C1 = Complex{T} where {T<:Integer}
C2 = Complex{T} where T<:Integer
C3 = Complex{<:Integer}

C1 == C2 == C3
```

```
true
```

Kurze Syntax für einfache Fälle, ausführliche Syntax für komplexe Varianten.

Als letztes sein bemerkt, dass `where T` die Kurzform von `where T<:Any` ist, also eine völlig unbeschränkte Typvariable einführt. Damit ist sowas möglich:

```
function fgl(x::T, y::T) where T
    println("Glückwunsch! x und y sind vom gleichen Typ!")
end
```

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

Diese Methode erfordert, dass die Argumente genau den gleichen, aber ansonsten beliebigen Typ haben.

```
fgl(33, 44)
```

```
Glückwunsch! x und y sind vom gleichen Typ!
```

```
fgl(33, 44.0)
```

```
LoadError: MethodError: no method matching fgl(::Int64, ::Float64)

Closest candidates are:
  fgl(::T, ::T) where T
   @ Main In[61]:1


Stacktrace:
 [1] top-level scope
   @ In[63]:1
```

9 Ein Fallbeispiel: Der parametrisierte Datentyp PComplex

Wir wollen als neuen numerischen Typen **komplexe Zahlen in Polardarstellung** $z = re^{i\phi} = (r, \phi)$ einführen.

- Der Typ soll sich in die Typhierarchie einfügen als Subtyp von 'Number'.
- r und ϕ sollen Gleitkommazahlen sein. (Im Unterschied zu komplexen Zahlen in 'kartesischen' Koordinaten hat eine Einschränkung auf ganzzahlige Werte von r oder ϕ mathematisch wenig Sinn.)

9.1 Die Definition von `PComplex`

Ein erster Versuch könnte so aussehen:

```
struct PComplex1{T <: AbstractFloat} <: Number
    r :: T
    ϕ :: T
end

z1 = PComplex1(-32.0, 33.0)
z2 = PComplex1{Float32}(12, 13)
@show z1 z2;
```

```
z1 = PComplex1{Float64}(-32.0, 33.0)
z2 = PComplex1{Float32}(12.0f0, 13.0f0)
```

Julia stellt automatisch *default constructors* zur Verfügung:

- den Konstruktor `PComplex1`, bei dem der Typ `T` von den übergebenen Argumenten abgeleitet wird und
- Konstruktoren `PComplex{Float64},...` mit expliziter Typangabe. Hier wird versucht, die Argumente in den angeforderten Typ zu konvertieren.

Wir wollen nun, dass der Konstruktor noch mehr tut. In der Polardarstellung soll $0 \leq r$ und $0 \leq \phi < 2\pi$ gelten.

Wenn die übergebenen Argumente das nicht erfüllen, sollten sie entsprechend umgerechnet werden.

Dazu definieren wir einen *inner constructor*, der den *default constructor* ersetzt.

- Ein *inner constructor* ist eine Funktion innerhalb der `struct`-Definition.
- In einem *inner constructor* kann man die spezielle Funktion `new` verwenden, die wie der *default constructor* wirkt.

```
struct PComplex{T <: AbstractFloat} <: Number
    r :: T
    ϕ :: T

    function PComplex{T}(r::T, ϕ::T) where T<:AbstractFloat
        if r<0            # flip the sign of r and correct phi
            r = -r
            ϕ += π
        end
```

```
        if r==0 ϕ=0 end  # normalize r=0 case to phi=0
        ϕ = mod(ϕ, 2π)   # map phi into interval [0,2pi)
        new(r, ϕ)        # new() ist special function,
    end                  #   available only inside inner constructors

end
```

```
z1 = PComplex{Float64}(-3.3, 7π+1)
```

```
PComplex{Float64}(3.3, 1.0)
```

Für die explizite Angabe eines *inner constructors* müssen wir allerdings einen Preis zahlen: Die sonst von Julia bereitgestellten *default constructors* fehlen.

Den Konstruktor, der ohne explizite Typangabe in geschweiften Klammern auskommt und den Typ der Argumente übernimmt, wollen wir gerne auch haben:

```
PComplex(r::T, ϕ::T) where {T<:AbstractFloat} = PComplex{T}(r,ϕ)

z2 = PComplex(2.0, 0.3)
```

```
PComplex{Float64}(2.0, 0.3)
```

9.2 Eine neue Schreibweise

Julia verwendet `//` als Infix-Konstruktor für den Typ `Rational`. Sowas Schickes wollen wir auch.

In der Elektronik/Elektrotechnik werden Wechselstromgrößen durch komplexe Zahlen beschrieben.. Dabei ist eine Darstellung komplexer Zahlen durch "Betrag" und "Phase" üblich und sie wird gerne in der sogenannten Versor-Form (engl. *phasor*) dargestellt:

$$z = r\angle\phi = 3.4\ \angle 45^\circ$$

wobei man in der Regel den Winkel in Grad notiert.

> **Mögliche Infix-Operatoren in Julia**
>
> In Julia ist eine große Anzahl von Unicode-Zeichen reserviert für die Verwendung als Operatoren. Die definitive Liste ist im Quellcode des Parsers.
> Auf Details werden wir in einem späteren Kapitel noch eingehen.
> Und ja, der Julia-Parser ist in einem Lisp(genauer: Scheme)-Dialekt geschrieben. In Julia ist ein kleiner Scheme-Interpreter namens femtolisp integriert. Geschrieben hat ihn einer der "Väter" von Julia bevor er mit der Arbeit an Julia begann.

Das Winkel-Zeichen ∠ steht leider nicht als Operatorsymbol zur Verfügung. Wir weichen aus auf ⩽. Das kann in Julia als als `\lessdot<tab>` eingegeben werden.

```
⩽(r::Real, ϕ::Real) = PComplex(r, π*ϕ/180)

z3 = 2. ⩽ 90.
```

```
PComplex{Float64}(2.0, 1.5707963267948966)
```

(Die Typ-Annotation – `Real` statt `AbstractFloat` – ist ein Vorgriff auf kommende weitere Konstruktoren. Im Moment funktioniert der Operator ⩽ erstmal nur mit `Floats`.)

Natürlich wollen wir auch die Ausgabe so schön haben. Details dazu findet man in der Dokumentation.

```
using Printf
```

```
function Base.show(io::IO, z::PComplex)
    # wir drucken die Phase in Grad, auf Zehntelgrad gerundet,
    p = z.ϕ * 180/π
    sp = @sprintf "%.1f" p
    print(io, z.r, "∢", sp, '°')
end

@show z3;
```

```
z3 = 2.0∢90.0°
```

9.3 Methoden für `PComplex`

Damit unser Typ ein anständiges Mitglied der von `Number` abstammenden Typfamilie wird, brauchen wir allerdings noch eine ganze Menge mehr. Es müssen Arithmetik, Vergleichsoperatoren, Konvertierungen usw. definiert werden.

Wir beschränken uns auf Multiplikation und Quadratwurzeln.

Module

- Um die `methods` der existierenden Funktionen und Operationen zu ergänzen, muss man diese mit ihrem 'vollen Namen' ansprechen.
- Alle Objekte gehören zu einem Namensraum oder `module`.
- Die meisten Basisfunktionen gehören zum Modul `Base`, welches standardmäßig immer ohne explizites `using ...` geladen wird.
- Solange man keine eigenen Module definiert, sind die eigenen Definitionen im Modul `Main`.
- Das Macro `@which`, angewendet auf einen Namen, zeigt an, in welchem Modul der Name definiert wurde.

```
f(x) = 3x^3
@which f
```

```
Main
```

```
wp = @which +
ws = @which(sqrt)
println("Modul für Addition: $wp, Modul für sqrt: $ws")
```

```
Modul für Addition: Base, Modul für sqrt: Base
```

```
qwurzel(z::PComplex) = PComplex(sqrt(z.r), z.ϕ / 2)
```

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

Die Funktion `sqrt()` hat schon einige Methoden:

```
length(methods(sqrt))
```

```
19
```

Jetzt wird es eine Methode mehr:

```
Base.sqrt(z::PComplex) = qwurzel(z)

length(methods(sqrt))
```

```
20
```

```
sqrt(z2)
```

```
1.4142135623730951∠8.6°
```

und nun zur Multiplikation:

```
Base.:*(x::PComplex, y::PComplex) = PComplex(x.r * y.r, x.ϕ + y.ϕ)

@show z1 * z2;
```

```
z1 * z2 = 6.6∠74.5°
```

(Da das Operatorsymbol kein normaler Name ist, muss der Doppelpunkt bei der Zusammensetzung mit `Base.` sein.)

Wir können allerdings noch nicht mit anderen numerischen Typen multiplizieren. Dazu könnte man nun eine Vielzahl von entsprechenden Methoden definieren. Julia stellt *für numerische Typen* noch einen weiteren Mechanismus zur Verfügung, der das etwas vereinfacht.

9.4 Typ-Promotion und Konversion

In Julia kann man bekanntlich die verschiedensten numerischen Typen nebeneinander verwenden.

```
1//3 + 5 + 5.2 + 0xff
```

```
265.53333333333336
```

Wenn man in die zahlreichen Methoden schaut, die z.B. für `+` und `*` definiert sind, findet man u.a. eine Art 'catch-all-Definition'

```
+(x::Number, y::Number) = +(promote(x,y)...)
*(x::Number, y::Number) = *(promote(x,y)...)
```

(Die 3 Punkte sind der splat-Operator, der das von promote() zurückgegebene Tupel wieder in seine Bestandteile zerlegt.)

Da die Methode mit den Typen `(Number, Number)` sehr allgemein ist, wird sie erst verwendet, wenn spezifischere Methoden nicht greifen.

Was passiert hier?

9.4.1 Die Funktion `promote(x,y,...)`

Diese Funktion versucht, alle Argumente in einen gemeinsamen Typen umzuwandeln, der alle Werte (möglichst) exakt darstellen kann.

```
promote(12, 34.555, 77/99, 0xff)
```

```
(12.0, 34.555, 0.7777777777777778, 255.0)
```

```
z = promote(BigInt(33), 27)
@show z typeof(z);
```

```
z = (33, 27)
typeof(z) = Tuple{BigInt, BigInt}
```

Die Funktion `promote()` verwendet dazu zwei Helfer, die Funktionen `promote_type(T1, T2)` und `convert(T, x)`

Wie üblich in Julia, kann man diesen Mechanismus durch eigene *promotion rules* und `convert(T,x)`-Methoden erweitern.

9.4.2 Die Funktion `promote_type(T1, T2,...)`

Sie ermittelt, zu welchem Typ umgewandelt werden soll. Argumente sind Typen, nicht Werte.

```
@show promote_type(Rational{Int64}, ComplexF64, Float32);
```

```
promote_type(Rational{Int64}, ComplexF64, Float32) = ComplexF64
```

9.4.3 Die Funktion `convert(T,x)`

Die Methoden von `convert(T, x)` wandeln `x` in ein Objekt vom Typ `T` um. Dabei sollte eine solche Umwandlung verlustfrei möglich sein.

```
z = convert(Float64, 3)
```

```
3.0
```

```
z = convert(Int64, 23.00)
```

```
23
```

```
z = convert(Int64, 2.3)
```

```
LoadError: InexactError: Int64(2.3)

Stacktrace:
 [1] Int64
   @ ./float.jl:912 [inlined]
 [2] convert(::Type{Int64}, x::Float64)
   @ Base ./number.jl:7
 [3] top-level scope
   @ In[23]:1
```

Die spezielle Rolle von `convert()` liegt darin, dass es an verschiedenen Stellen *implizit* und automatisch eingesetzt wird:

> The following language constructs call convert:
>
> - Assigning to an array converts to the array's element type.
> - Assigning to a field of an object converts to the declared type of the field.
> - Constructing an object with new converts to the object's declared field types.
> - Assigning to a variable with a declared type (e.g. local x::T) converts to that type.
> - A function with a declared return type converts its return value to that type.

– und natürlich in `promote()`

Für selbstdefinierte Datentypen kann man convert() um weitere Methoden ergänzen.

Für Datentypen innerhalb der Number-Hierarchie gibt es wieder eine 'catch-all-Definition'

```
convert(::Type{T}, x::Number) where {T<:Number} = T(x)
```

Also: Wenn für einen Typen `T` aus der Hierarchie `T<:Number` ein Konstruktor `T(x)` mit einem numerischen Argument `x` existiert, dann wird dieser Konstruktor `T(x)` automatisch für Konvertierungen benutzt. (Natürlich können auch speziellere Methoden für `convert()` definiert werden, die dann Vorrang haben.)

9.4.4 Weitere Konstruktoren für `PComplex`

```
## (a) r, ϕ beliebige Reals, z.B.  Integers, Rationals

PComplex{T}(r::T1, ϕ::T2) where {T<:AbstractFloat, T1<:Real, T2<: Real} =
    PComplex{T}(convert(T, r), convert(T, ϕ))

PComplex(r::T1, ϕ::T2) where {T1<:Real, T2<: Real} =
    PComplex{promote_type(Float64, T1, T2)}(r, ϕ)

## (b) Zur Umwandlung von Reals: Konstruktor mit
##      nur einem Argument r

PComplex{T}(r::S) where {T<:AbstractFloat, S<:Real} =
    PComplex{T}(convert(T, r), convert(T, 0))

PComplex(r::S)  where {S<:Real} =
    PComplex{promote_type(Float64, S)}(r, 0.0)

## (c) Umwandlung Complex -> PComplex

PComplex{T}(z::Complex{S}) where {T<:AbstractFloat, S<:Real} =
    PComplex{T}(abs(z), angle(z))

PComplex(z::Complex{S}) where {S<:Real} =
    PComplex{promote_type(Float64, S)}(abs(z), angle(z))
```

```
PComplex
```

Ein Test der neuen Konstruktoren:

```
3//5 ∢ 45,  PComplex(Complex(1,1)),   PComplex(-13)
```

```
(0.6∢45.0°, 1.4142135623730951∢45.0°, 13.0∢180.0°)
```

Wir brauchen nun noch *promotion rules*, die festlegen, welcher Typ bei `promote(x::T1, y::T2)` herauskommen soll. Damit wird `promote_type()` intern um die nötigen weiteren Methoden erweitert.

9.4.5 *Promotion rules* für `PComplex`

```
Base.promote_rule(::Type{PComplex{T}}, ::Type{S}) where {T<:AbstractFloat,S<:Real} =
    PComplex{promote_type(T,S)}

Base.promote_rule(::Type{PComplex{T}}, ::Type{Complex{S}}) where
    {T<:AbstractFloat,S<:Real} = PComplex{promote_type(T,S)}
```

1. **Regel:** Wenn ein `PComplex{T}` und ein `S<:Real` zusammentreffen, dann sollen beide zu `PComplex{U}` umgewandelt werden, wobei `U` der Typ ist, zu dem `S` und `T` beide umgewandelt (*promoted*) werden können.
2. **Regel** Wenn ein `PComplex{T}` und ein `Complex{S}` zusammentreffen, dann sollen beide zu `PComplex{U}` umgewandelt werden, wobei `U` der Typ ist, zu dem `S` und `T` beide umgewandelt werden können.

Damit klappt nun die Multiplikation mit beliebigen numerischen Typen.

```
z3, 3z3
```

```
(2.0∠90.0°, 6.0∠90.0°)
```

```
(3.0+2im) * (12∠30.3),  12sqrt(z2)
```

```
(43.26661530556787∠64.0°, 16.970562748477143∠8.6°)
```

Zusammenfassung: unser Typ `PComplex`

```
struct PComplex{T <: AbstractFloat} <: Number
    r :: T
    ϕ :: T

    function PComplex{T}(r::T, ϕ::T) where T<:AbstractFloat
        if r<0            # flip the sign of r and correct phi
            r = -r
            ϕ += π
        end
        if r==0 ϕ=0 end  # normalize r=0 case to phi=0
        ϕ = mod(ϕ, 2π)   # map phi into interval [0,2pi)
        new(r, ϕ)        # new() ist special function,
    end                  #   available only inside inner constructors

end

# additional constructors
PComplex(r::T, ϕ::T) where {T<:AbstractFloat} = PComplex{T}(r,ϕ)


PComplex{T}(r::T1, ϕ::T2) where {T<:AbstractFloat, T1<:Real, T2<: Real} =
    PComplex{T}(convert(T, r), convert(T, ϕ))

PComplex(r::T1, ϕ::T2) where {T1<:Real, T2<: Real} =
    PComplex{promote_type(Float64, T1, T2)}(r, ϕ)


PComplex{T}(r::S) where {T<:AbstractFloat, S<:Real} =
    PComplex{T}(convert(T, r), convert(T, 0))

PComplex(r::S)  where {S<:Real} =
    PComplex{promote_type(Float64, S)}(r, 0.0)


PComplex{T}(z::Complex{S}) where {T<:AbstractFloat, S<:Real} =
    PComplex{T}(abs(z), angle(z))

PComplex(z::Complex{S}) where {S<:Real} =
    PComplex{promote_type(Float64, S)}(abs(z), angle(z))

# nice input
∢(r::Real, ϕ::Real) = PComplex(r, π*ϕ/180)

# nice output
using Printf

function Base.show(io::IO, z::PComplex)
    # wir drucken die Phase in Grad, auf Zehntelgrad gerundet,
    p = z.ϕ * 180/π
    sp = @sprintf "%.1f" p
    print(io, z.r, "∢", sp, '°')
end

# arithmetic
Base.sqrt(z::PComplex) = PComplex(sqrt(z.r), z.ϕ / 2)

Base.:*(x::PComplex, y::PComplex) = PComplex(x.r * y.r, x.ϕ + y.ϕ)

# promotion rules
Base.promote_rule(::Type{PComplex{T}}, ::Type{S}) where
    {T<:AbstractFloat,S<:Real} = PComplex{promote_type(T,S)}

Base.promote_rule(::Type{PComplex{T}}, ::Type{Complex{S}}) where
    {T<:AbstractFloat,S<:Real} = PComplex{promote_type(T,S)}
```

10 Funktionen und Operatoren

Funktionen verarbeiten ihre Argumente zu einem Ergebnis, das sie beim Aufruf zurückliefern.

10.1 Formen

Funktionen können in verschiedenen Formen definiert werden:

I. Als `function ... end`-Block

```
function hyp(x,y)
    sqrt(x^2+y^2)
end
```

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

II. Als "Einzeiler"

```
hyp(x, y) = sqrt(x^2 + y^2)
```

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

III. Als anonyme Funktionen

```
(x, y) -> sqrt(x^2 + y^2)
```

```
#4 (generic function with 1 method)
```

10.1.1 Block-Form und `return`

- Mit `return` wird die Abarbeitung der Funktion beendet und zum aufrufenden Kontext zurückgekehrt.
- Ohne `return` wird der Wert des letzten Ausdrucks als Funktionswert zurückgegeben.

Die beiden Definitionen

```
function xsinrecipx(x)
    if x == 0
        return 0.0
    end
    return x * sin(1/x)
end
```

und ohne das zweite explizite `return` in der letzten Zeile:

```
function xsinrecipx(x)
    if x == 0
        return 0.0
    end
    x * sin(1/x)
end
```

sind also äquivalent.

- Eine Funktion, die "nichts" zurückgibt (*void functions* in der C-Welt), gibt den Wert `nothing` vom Typ `Nothing` zurück. (So wie ein Objekt vom Typ `Bool` die beiden Werte `true` und `false` haben kann, so kann ein Objekt vom Typ `Nothing` nur einen einzigen Wert, eben `nothing`, annehmen.)
- Eine leere `return`-Anweisung ist äquivalent zu `return nothing`.

```
function fn(x)
    println(x)
    return
end

a = fn(2)
```

```
2
```

```
a
```

```
@show a typeof(a);
```

```
a = nothing
typeof(a) = Nothing
```

10.1.2 Einzeiler-Form

Die Einzeilerform ist eine ganz normale Zuweisung, bei der links eine Funktion steht.

```
hyp(x, y) = sqrt(x^2 + y^2)
```

Julia kennt zwei Möglichkeiten, mehrere Anweisungen zu einem Block zusammenzufassen, der an Stelle einer Einzelanweisung stehen kann:

- `begin ... end`-Block
- Eingeklammerte durch Semikolon getrennte Anweisungen.

In beiden Fällen ist der Wert des Blockes der Wert der letzten Anweisung.

Damit funktioniert auch

```
hyp(x, y) = (z = x^2; z += y^2; sqrt(z))
```

und

```
hyp(x, y) = begin
                z = x^2
                z += y^2
                sqrt(z)
            end
```

10.1.3 Anonyme Funktionen

Anonyme FUnktionen kann man der Anonymität entreisen, indem man ihnen einen Namen zuweist.

```
hyp = (x,y) -> sqrt(x^2 + y^2)
```

Ihre eigentliche Anwendung ist aber im Aufruf einer *(higher order)* Funktion, die eine Funktion als Argument erwartet.

Typische Anwendungen sind `map(f, collection)`, welches eine Funktion auf alle Elemente einer Kollektion anwendet. In Julia funktioniert auch `map(f(x,y), collection1, collection2)`:

```
map( (x,y) -> sqrt(x^2 + y^2), [3, 5, 8], [4, 12, 15])
```

```
3-element Vector{Float64}:
  5.0
 13.0
 17.0
```

```
map( x->3x^3, 1:8 )
```

```
8-element Vector{Int64}:
    3
   24
   81
  192
  375
  648
 1029
 1536
```

Ein weiteres Beispiel ist `filter(test, collection)`, wobei ein Test eine Funktion ist, die ein `Bool` zurückgibt.

```
filter(x -> ( x%3 == 0 && x%5 == 0), 1:100  )
```

```
6-element Vector{Int64}:
 15
 30
 45
 60
 75
 90
```

10.2 Argumentübergabe

- Beim Funktionsaufruf werden von den als Funktionsargumente zu übergebenden Objekten keine Kopien gemacht. Die Variablen in der Funktion verweisen auf die Originalobjekte. Julia nennt dieses Konzept *pass_by_sharing*.

- Funktionen können also ihre Argumente wirksam modifizieren, falls es sich um *mutable* Objekte, wie z.B. `Vector`, `Array` handelt.
- Es ist eine Konvention in Julia, dass die Namen von solchen argumentmodifizierenden Funktionen mit einem Ausrufungszeichen enden. Weiterhin steht dann üblicherweise das Argument, das modifiziert wird, an erster Stelle und es ist auch der Rückgabewert der Funktion.

```
V = [1, 2, 3]

W = fill!(V, 17)
                        # '===' ist Test auf Identität
@show  V  W  V===W;     # V und W benennen dasselbe Objekt
```

```
V = [17, 17, 17]
W = [17, 17, 17]
V === W = true
```

```
function fill_first!(V, x)
        V[1] = x
        return V
end

U = fill_first!(V, 42)

@show V   U   V===U;
```

```
V = [42, 17, 17]
U = [42, 17, 17]
V === U = true
```

10.3 Varianten von Funktionsargumenten

- Es gibt Positionsargumente (1. Argument, 2. Argument, ...) und *keyword*-Argumente, die beim Aufruf durch ihren Namen angesprochen werden müssen.
- Sowohl Positions- als auch *keyword*-Argumente können *default*-Werte haben. Beim Aufruf können diese Argumente weggelassen werden.
- Die Reihenfolge der Deklaration muss sein:
  1. Positionsargumente ohne Defaultwert,
  2. Positionsargumente mit Defaultwert,
  3. — Semikolon —,
  4. kommagetrennte Liste der Keywordargumente (mit oder ohne Defaultwert)
- Beim Aufruf können *keyword*-Argumente an beliebigen Stellen in beliebiger Reihenfolge stehen. Man kann sie wieder durch ein Semikolon von den Positionsargumenten abtrennen, muss aber nicht.

```
fa(x, y=42; a) = println("x=$x, y=$y, a=$a")

fa(6, a=4, 7)
fa(6, 7; a=4)
fa(a=-2, 6)
```

```
x=6, y=7, a=4
x=6, y=7, a=4
x=6, y=42, a=-2
```

Eine Funktion nur mit *keyword*-Argumenten wird so deklariert:

```
fkw(; x=10, y) = println("x=$x, y=$y")

fkw(y=2)
```

```
x=10, y=2
```

10.4 Funktionen sind ganz normale Objekte

- Sie können zugewiesen werden

```
f2 = sqrt
f2(2)
```

```
1.4142135623730951
```

- Sie können als Argumente an Funktionen übergeben werden.

```
# sehr naive numerische Integration

function Riemann_integrate(f, a, b; NInter=1000)
    delta = (b-a)/NInter
    s = 0
    for i in 0:NInter-1
        s += delta * f(a + delta/2 + i * delta)
    end
    return s
end


Riemann_integrate(sin, 0, π)
```

```
2.0000008224672694
```

- Sie können von Funktionen erzeugt und als Ergebnis `returnt` werden.

```
function generate_add_func(x)
    function addx(y)
        return x+y
    end
    return addx
end
```

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

```
h =  generate_add_func(4)
```

```
(::var"#addx#15"{Int64}) (generic function with 1 method)
```

```
h(1)
```

```
5
```

```
h(2), h(10)
```

```
(6, 14)
```

Die obige Funktion `generate_add_func()` lässt sich auch kürzer definieren. Der innere Funktionsname `addx()` ist sowieso lokal und außerhalb nicht verfügbar. Also kann man eine anonyme Funktion verwenden.

```
generate_add_func(x) = y -> x + y
```

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

10.5 Zusammensetzung von Funktionen: die Operatoren ∘ und |>

- Die Zusammensetzung *(composition)* von Funktionen kann auch mit dem Operator ∘ (`\circ + Tab`) geschrieben werden

$$(f \circ g)(x) = f(g(x))$$

```
(sqrt ∘ + )(9, 16)
```

```
5.0
```

```
f = cos ∘ sin ∘ (x->2x)
f(.2)
```

```
0.9251300429004277
```

```
@show map(uppercase ∘ first, ["ein", "paar", "grüne", "Blätter"]);
```

```
map(uppercase ∘ first, ["ein", "paar", "grüne", "Blätter"]) = ['E', 'P', 'G', 'B']
```

- Es gibt auch einen Operator, mit dem Funktionen "von rechts" wirken und zusammengesetzt werden können *(piping)*

```
25 |> sqrt
```

```
5.0
```

```
1:10 |> sum |> sqrt
```

```
7.416198487095663
```

- Natürlich kann man auch diese Operatoren 'broadcasten' (s. Kapitel 12.7). Hier wirkt ein Vektor von Funktionen elementweise auf einen Vektor von Argumenten:

```
["a", "list", "of", "strings"] .|> [length, uppercase, reverse, titlecase]
```

```
4-element Vector{Any}:
 1
  "LIST"
  "fo"
  "Strings"
```

10.6 Die do-Notation

Eine syntaktische Besonderheit zur Definition anonymer Funktionen als Argumente anderer Funktionen ist die do-Notation.

Sei `higherfunc(f,a,...)` eine Funktion, deren 1. Argument eine Funktion ist.

Dann kann man `higherfunc()` auch ohne erstes Argument aufrufen und statt dessen die Funktion in einem unmittelbar folgenden do-Block definieren:

```
higherfunc(a, b) do x, y
   Körper von f(x,y)
end
```

Am Beispiel von `Riemann_integrate()` sieht das so aus:

```
# das ist dasselbe wie Riemann_integrate(x->x^2, 0, 2)

Riemann_integrate(0, 2) do x x^2 end
```

```
2.6666659999999993
```

Der Sinn besteht natürlich in der Anwendung mit komplexeren Funktionen, wie diesem aus zwei Teilstücken zusammengesetzten Integranden:

```
r = Riemann_integrate(0, π) do x
        z1 = sin(x)
        z2 = log(1+x)
        if x > 1
            return z1^2
        else
            return 1/z2^2
        end
    end
```

```
1578.9022037353475
```

10.7 Funktionsartige Objekte

Durch Definition einer geeigneten Methode für einen Typ kann man beliebige Objekte *callable* machen, d.h., sie anschließend wie Funktionen aufrufen.

```
# struct speichert die Koeffiziente eines Polynoms 2. Grades
struct Poly2Grad
    a0::Float64
    a1::Float64
    a2::Float64
end

p1 = Poly2Grad(2,5,1)
p2 = Poly2Grad(3,1,-0.4)
```

```
Poly2Grad(3.0, 1.0, -0.4)
```

Die folgende Methode macht diese Struktur `callable`.

```
function (p::Poly2Grad)(x)
    p.a2 * x^2 + p.a1 * x + p.a0
end
```

Jetzt kann man die Objekte, wenn gewünscht, auch wie Funktionen verwenden.

```
@show p2(5)  p1(-0.7)  p1;
```

```
p2(5) = -2.0
p1(-0.7) = -1.0100000000000002
p1 = Poly2Grad(2.0, 5.0, 1.0)
```

10.8 Operatoren und spezielle Formen

- Infix-Operatoren wie `+,*,>,∈,...` sind Funktionen.

```
+(3, 7)
```

```
10
```

```
f = +
```

```
+ (generic function with 189 methods)
```

```
f(3, 7)
```

```
10
```

- Auch Konstruktionen wie `x[i]`, `a.x`, `[x; y]` werden vom Parser zu Funktionsaufrufen umgewandelt.

x[i]	getindex(x, i)
x[i] = z	setindex!(x, z, i)
a.x	getproperty(a, :x)
a.x = z	setproperty!(a, :x, z)
[x; y;...]	vcat(x, y, ...)

Tabelle 10.1: Spezielle Formen (Auswahl)

(Der Doppelpunkt vor einer Variablen macht diese zu einem Symbol.)

> **Hinweis**
>
> Für diese Funktionen kann man eigene Methoden implementieren. Zum Beispiel könnten bei einem eigenen Typ das Setzen eines Feldes (`setproperty!()`) die Gültigkeit des Wertes prüfen oder weitere Aktionen veranlassen.
> Prinzipiell können `get/setproperty` auch Dinge tun, die gar nichts mit einem tatsächlich vorhandenen Feld der Struktur zu tun haben.

10.9 Update-Form

Alle arithmetischen Infix-Operatoren haben eine update-Form: Der Ausdruck

```
x = x ⊙ y
```

kann auch geschrieben werden als

```
x ⊙= y
```

Beide Formen sind semantisch äquivalent. Insbesondere wird in beiden Formen der Variablen `x` ein auf der rechten Seite geschaffenes neues Objekt zugewiesen.

Ein Speicherplatz- und Zeit-sparendes **in-place-update** eines Arrays/Vektors/Matrix ist möglich entweder durch explizite Indizierung

```
for i in eachindex(x)
    x[i] += y[i]
end
```

oder durch die dazu semantisch äquivalente *broadcast*-Form (s. Kapitel 12.7):

```
x .= x .+ y
```

10.10 Vorrang und Assoziativität von Operatoren

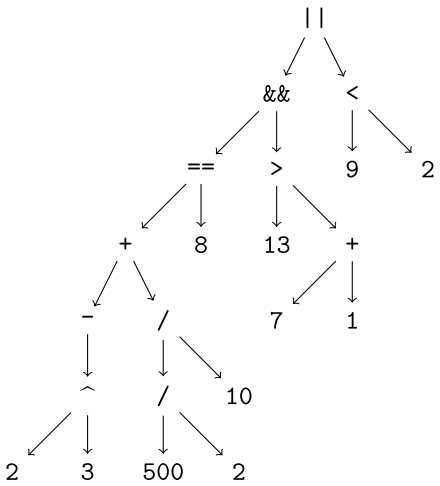
Zu berechnende Ausdrücke

```
-2^3+500/2/10==8 && 13 > 7 + 1 || 9 < 2
```

```
false
```

werden vom Parser in eine Baumstruktur überführt.

```
using TreeView

walk_tree(Meta.parse("-2^3+500/2/10==8 && 13 > 7 + 1 || 9 < 2"))
```



- Die Auswertung solcher Ausdrücke wird durch
  - Vorrang *(precedence)* und
  - Assoziativität geregelt.
- 'Vorrang' definiert, welche Operatoren stärker binden im Sinne von "Punktrechnung geht vor Strichrechnung".
- 'Assoziativität' bestimmt die Auswertungsreihenfolge bei gleichen oder gleichrangigen Operatoren.
- Vollständige Dokumentation

10.10.1 Assoziativität

Sowohl Addition und Subtraktion als auch Multiplikation und Divison sind jeweils gleichrangig und linksassoziativ, d.h. es wird von links ausgewertet.

```
200/5/2      # wird von links ausgewertet als (200/5)/2
```

```
20.0
```

```
200/2*5      # wird von links ausgewertet als (200/2)*5
```

```
500.0
```

Zuweisungen wie `=`, `+=`, `*=`,... sind gleichrangig und rechtsassoziativ.

```
x = 1
y = 10

# wird von rechts ausgewertet:  x += (y += (z = (a = 20)))

x += y +=  z = a = 20

@show x y z a;
```

```
x = 31
y = 30
```

```
z = 20
a = 20
```

Natürlich kann man die Assoziativität in Julia auch abfragen. Die entsprechenden Funktionen werden nicht explizit aus dem `Base`-Modul exportiert, deshalb muss man den Modulnamen beim Aufruf angeben.

```
for i in (:/, :+=, :(=), :^)
    a = Base.operator_associativity(i)
    println("Operation $i is  $(a)-assoziative")
end
```

```
Operation / is  left-assoziative
Operation += is  right-assoziative
Operation = is  right-assoziative
Operation ^ is  right-assoziative
```

Also ist der Potenzoperator rechtsassoziativ.

```
2^3^2    # rechtsassoziativ,  = 2^(3^2)
```

```
512
```

10.10.2 Vorrang

- Julia ordnet den Operatoren Vorrangstufen von 1 bis 17 zu:

```
for i in (:+, :-, :*, :/, :^, :(=))
    p = Base.operator_precedence(i)
    println("Vorrang von  $i = $p")
end
```

```
Vorrang von  + = 11
Vorrang von  - = 11
Vorrang von  * = 12
Vorrang von  / = 12
Vorrang von  ^ = 15
Vorrang von  = = 1
```

- 11 ist kleiner als 12, also geht 'Punktrechnung vor Strichrechnung'
- Der Potenz-Operator `^` hat eine höhere *precedence*.
- Zuweisungen haben die kleinste *precedence*

```
#   Zuweisung hat kleinsten Vorrang, daher Auswertung als  x = (3 < 4)

x = 3 < 4
x
```

```
true
```

```
(y = 3) < 4   # Klammern schlagen natürlich jeden Vorrang
y
```

```
3
```

Nochmal zum Beispiel vom Anfang von Kapitel 10.10:

```
-2^3+500/2/10==8 && 13 > 7 + 1 || 9 < 2
```

```
false
```

```
for i ∈ (:^, :+, :/, :(==), :&&, :>, :|| )
    print(i, " ")
    println(Base.operator_precedence(i))
end
```

```
^ 15
+ 11
/ 12
== 7
&& 6
> 7
|| 5
```

Nach diesen Vorrangregeln wird der Beispielausdruck also wie folgt ausgewertet:

```
((-(2^3)+((500/2)/10)==8) && (13 > (7 + 1))) || (9 < 2)
```

```
false
```

(Das entspricht natürlich dem oben gezeigten *parse-tree*)

Es gilt also für den Vorrang:

> Potenz > Multiplikation/Division > Addition/Subtraktion > Vergleiche > logisches && > logisches || > Zuweisung

Damit wird ein Ausdruck wie

```
a = x <= y + z && x > z/2
```

sinnvoll ausgewertet als `a = ((x <= (y+z)) && (x < (z/2)))`

- Eine Besonderheit sind noch
  - unäre Operatoren, also insbesondere + und - als Vorzeichen
  - *juxtaposition*, also Zahlen direkt vor Variablen oder Klammern ohne ∗-Symbol

  Beide haben Vorrang noch vor Multiplikation und Division.

> **!** Wichtig
>
> Damit ändert sich die Bedeutung von Ausdrücken, wenn man *juxtaposition* anwendet:
>
> ```
> 1/2*π,  1/2π
> ```
>
> ```
> (1.5707963267948966, 0.15915494309189535)
> ```

- Im Vergleich zum Potenzoperator ^ gilt (s. https://discourse.julialang.org/t/confused-about-operator-precedence-for-2-3x/8214/7):

  > Unary operators, including juxtaposition, bind tighter than ^ on the right but looser on the left.

Beispiele:

```
-2^2    # -(2^2)
```

```
-4
```

```
x = 5
2x^2    # 2(x^2)
```

```
50
```

```
2^-2    # 2^(-2)
```

```
0.25
```

```
2^2x  # 2^(2x)
```

```
1024
```

- Funktionsanwendung `f(...)` hat Vorrang vor allen Operatoren

```
sin(x)^2 === (sin(x))^2   # nicht sin(x^2)
```

```
true
```

10.10.3 Zusätzliche Operatoren

Der Julia-Parser definiert für zahlreiche Unicode-Zeichen einen Vorrang auf Vorrat, so dass diese Zeichen von Paketen und selbstgeschriebenem Code als Operatoren benutzt werden können.

So haben z.B.

```
↑ ↓ ⇵ ⟰ ⟱ ⤈ ⤉ ⤊ ⤋ ⤒ ⤓ ⥉ ⥌ ⥍ ⥏ ⥑ ⥔ ⥕ ⥘ ⥙ ⥜ ⥝ ⥠ ⥡ ⥣ ⥥ ⥮ ⥯ ￪ ￬
```

den Vorrang 12 wie Multiplikation/Division (und sind wie diese linksassoziativ) und z.B.

```
⊕ ⊖ ⊞ ⊟ |++| ∪ ∨ ⊔ ± ∓ ∔ ∸ ≏ ⊎ ⊻ ⋎ ⋓ ⩱ ⩲ ⩳ ⩴ ⩵ ⩶ ⩷ ⩸ ⩹ ⩺ ⩻ ⩼ ⩽ ⩾ ⩿ ⪀ ⪁ ⪂ ⪃ ⪄ ⪅ ⪆ ⪇ ⪈ ⪉ ⪊ ⪋ ⪌ ⪍ ⪎ ⪏
```

haben den Vorrang 11 wie Addition/Subtraktion.

11 Container

Julia bietet eine große Auswahl von Containertypen mit weitgehend ähnlichem Interface an. Wir stellen hier `Tuple`, `Range` und `Dict` vor, im nächsten Kapitel dann `Array`, `Vector` und `Matrix`.

Diese Container sind:

- **iterierbar**: Man kann über die Elemente des Containers iterieren:

```
for x ∈ container ... end
```

- **indizierbar**: Man kann auf Elemente über ihren Index zugreifen:

```
x = container[i]
```

und einige sind auch

- **mutierbar**: Man kann Elemente hinzufügen, entfernen und ändern.

Weiterhin gibt es eine Reihe gemeinsamer Funktionen, z.B.

- `length(container)` — Anzahl der Elemente
- `eltype(container)` — Typ der Elemente
- `isempty(container)` — Test, ob Container leer ist
- `empty!(container)` — leert Container (nur wenn mutierbar)

11.1 Tupeln

Ein Tupel ist ein nicht mutierbarer Container von Elementen. Es ist also nicht möglich, neue Elemente dazuzufügen oder den Wert eines Elements zu ändern.

```
t = (33, 4.5, "Hello")

@show   t[2]              # indizierbar

for i ∈ t println(i) end   # iterierbar
```

```
t[2] = 4.5
33
4.5
Hello
```

Ein Tupel ist ein **inhomogener** Typ. Jedes Element hat seinen eigenen Typ und das zeigt sich auch im Typ des Tupels:

```
typeof(t)
```

```
Tuple{Int64, Float64, String}
```

Man verwendet Tupel gerne als Rückgabewerte von Funktionen, um mehr als ein Objekt zurückzulieferen.

```
# Ganzzahldivision und Rest:
# Quotient und Rest werden den Variablen q und r zugewiesen
```

```
q, r = divrem(71, 6)
@show  q  r;
```

```
q = 11
r = 5
```

Wie man hier sieht, kann man in bestimmten Konstrukten die Klammern auch weglassen. Dieses *implict tuple packing/unpacking* verwendet man auch gerne in Mehrfachzuweisungen:

```
x, y, z = 12, 17, 203
```

```
(12, 17, 203)
```

```
y
```

```
17
```

Manche Funktionen bestehen auf Tupeln als Argument oder geben immer Tupeln zurück. Dann braucht man manchmal ein Tupel aus einem Element.

Das notiert man so:

```
x = (13,)         # ein 1-Element-Tupel
```

```
(13,)
```

Das Komma - und nicht die Klammern – macht das Tupel.

```
x= (13)         # kein Tupel
```

```
13
```

11.2 Ranges

Wir haben *range*-Objekte schon in numerischen `for`-Schleifen verwendet.

```
r = 1:1000
typeof(r)
```

```
UnitRange{Int64}
```

Es gibt verschiedene *range*-Typen. Wie man sieht, sind es über den Zahlentyp parametrisierte Typen und `UnitRange` ist z.B. ein *range* mit der Schrittweite 1. Ihre Konstruktoren heißen in der Regel `range()`.

Der Doppelpunkt ist eine spezielle Syntax.

- `a:b` wird vom Parser umgesetzt zu `range(a, b)`
- `a:b:c` wird umgesetzt zu `range(a, c, step=b)`

Ranges sind offensichtlich iterierbar, nicht mutierbar aber indizierbar.

```
(3:100)[20]   # das zwanzigste Element
```

```
22
```

Wir erinnern an die Semantik der `for`-Schleife: `for i in 1:1000` heißt **nicht**:

- 'Die Schleifenvariable `i` wird bei jedem Durchlauf um eins erhöht' **sondern**

- ‘Der Schleifenvariable werden nacheinander die Werte 1,2,3,...,1000 aus dem Container zugewiesen’.

Allerdings wäre es sehr ineffektiv, diesen Container tatsächlich explizit anzulegen.

- *Ranges* sind “lazy” Vektoren, die nie wirklich irgendwo als konkrete Liste abgespeichert werden. Das macht sie als Iteratoren in `for`-Schleifen so nützlich: speichersparend und schnell.
- Sie sind ‘Rezepte’ oder Generatoren, die auf die Abfrage ‘Gib mir dein nächstes Element!’ antworten.
- Tatsächlich ist der Muttertyp `AbstractRange` ein Subtyp von `AbstractVector`.

Das Macro `@allocated` gibt aus, wieviel Bytes an Speicher bei der Auswertung eines Ausdrucks alloziert wurden.

```
@allocated [1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20]
```

```
0
```

```
@allocated 1:20
```

```
0
```

Zum Umwandeln in einen ‘richtigen’ Vektor dient die Funktion `collect()`.

```
collect(20:-3:1)
```

```
7-element Vector{Int64}:
 20
 17
 14
 11
  8
  5
  2
```

Recht nützlich, z.B. beim Vorbereiten von Daten zum Plotten, ist der *range*-Typ `LinRange`.

```
LinRange(2, 50, 300)
```

```
300-element LinRange{Float64, Int64}:
 2.0, 2.16054, 2.32107, 2.48161, 2.64214, …, 49.5184, 49.6789, 49.8395, 50.0
```

`LinRange(start, stop, n)` erzeugt eine äquidistante Liste von `n` Werten von denen der erste und der letzte die vorgegebenen Grenzen sind. Mit `collect()` kann man bei Bedarf auch daraus den entsprechenden Vektor gewinnen.

11.3 Dictionaries

- *Dictionaries* (deutsch: “assoziative Liste” oder “Zuordnungstabelle” oder ...) sind spezielle Container.
- Einträge in einem Vektor `v` sind durch einen Index 1,2,3.... addressierbar: `v[i]`
- Einträge in einem *dictionary* sind durch allgemeinere *keys* addressierbar.
- Ein *dictionary* ist eine Ansammlung von *key-value*-Paaren.
- Damit haben *dictionaries* in Julia den parametrisierten Typ `Dict{S,T}`, wobei `S` der Typ der *keys* und `T` der Typ der *values* ist

Man kann sie explizit anlegen:

```
# Einwohner 2020 in Millionen, Quelle: wikipedia

EW = Dict("Berlin" => 3.66,  "Hamburg" => 1.85,
          "München" => 1.49, "Köln" => 1.08)
```

```
Dict{String, Float64} with 4 entries:
  "München" => 1.49
  "Köln"    => 1.08
  "Berlin"  => 3.66
  "Hamburg" => 1.85
```

```
typeof(EW)
```

```
Dict{String, Float64}
```

und mit den *keys* indizieren:

```
EW["Berlin"]
```

```
3.66
```

Das Abfragen eines nicht existierenden *keys* ist natürlich ein Fehler.

```
EW["Leipzig"]
```

```
LoadError: KeyError: key "Leipzig" not found

Stacktrace:
 [1] getindex(h::Dict{String, Float64}, key::String)
   @ Base ./dict.jl:498
 [2] top-level scope
   @ In[18]:1
```

Man kann ja auch vorher mal anfragen...

```
haskey(EW, "Leipzig")
```

```
false
```

... oder die Funktion `get(dict, key, default)` benutzen, die bei nicht existierendem Key keinen Fehler wirft sondern das 3. Argument zurückgibt.

```
@show get(EW, "Leipzig", -1)   get(EW, "Berlin", -1);
```

```
get(EW, "Leipzig", -1) = -1
get(EW, "Berlin", -1) = 3.66
```

Man kann sich auch alle `keys` und `values` als spezielle Container geben lassen.

```
keys(EW)
```

```
KeySet for a Dict{String, Float64} with 4 entries. Keys:
  "München"
  "Köln"
  "Berlin"
  "Hamburg"
```

```
values(EW)
```

```
ValueIterator for a Dict{String, Float64} with 4 entries. Values:
  1.49
  1.08
  3.66
  1.85
```

Man kann über die `keys` iterieren...

```
for i in keys(EW)
    n = EW[i]
    println("Die Stadt $i hat $n Millionen Einwohner.")
end
```

```
Die Stadt München hat 1.49 Millionen Einwohner.
Die Stadt Köln hat 1.08 Millionen Einwohner.
Die Stadt Berlin hat 3.66 Millionen Einwohner.
Die Stadt Hamburg hat 1.85 Millionen Einwohner.
```

odere gleich über `key-value`-Paare.

```
for (stadt, ew) ∈ EW
    println("$stadt : $ew  Mill.")
end
```

```
München : 1.49  Mill.
Köln : 1.08  Mill.
Berlin : 3.66  Mill.
Hamburg : 1.85  Mill.
```

11.3.1 Erweitern und Modifizieren

Man kann in ein `Dict` zusätzliche `key-value`-Paare eintragen...

```
EW["Leipzig"] = 0.52
EW["Dresden"] = 0.52
EW
```

```
Dict{String, Float64} with 6 entries:
  "Dresden" => 0.52
  "München" => 1.49
  "Köln"    => 1.08
  "Berlin"  => 3.66
  "Leipzig" => 0.52
  "Hamburg" => 1.85
```

und einen `value` ändern.

```
# Oh, das war bei Leipzig die Zahl von 2010, nicht 2020

EW["Leipzig"] = 0.597
EW
```

```
Dict{String, Float64} with 6 entries:
  "Dresden" => 0.52
  "München" => 1.49
  "Köln"    => 1.08
  "Berlin"  => 3.66
  "Leipzig" => 0.597
  "Hamburg" => 1.85
```

Ein Paar kann über seinen `key` auch gelöscht werden.

```
delete!(EW, "Dresden")
```

```
Dict{String, Float64} with 5 entries:
  "München" => 1.49
  "Köln"    => 1.08
  "Berlin"  => 3.66
  "Leipzig" => 0.597
  "Hamburg" => 1.85
```

Zahlreiche Funktionen können mit `Dicts` wie mit anderen Containern arbeiten.

```
maximum(values(EW))
```

```
3.66
```

11.3.2 Anlegen eines leeren Dictionaries

Ohne Typspezifikation ...

```
d1 = Dict()
```

```
Dict{Any, Any}()
```

und mit Typspezifikation:

```
d2 = Dict{String, Int}()
```

```
Dict{String, Int64}()
```

11.3.3 Umwandlung in Vektoren: `collect()`

- `keys(dict)` und `values(dict)` sind spezielle Datentypen.
- Die Funktion `collect()` macht daraus eine Liste vom Typ `Vector`.
- `collect(dict)` liefert eine Liste vom Typ `Vector{Pair{S,T}}`

```
collect(EW)
```

```
5-element Vector{Pair{String, Float64}}:
 "München" => 1.49
    "Köln" => 1.08
  "Berlin" => 3.66
 "Leipzig" => 0.597
 "Hamburg" => 1.85
```

```
collect(keys(EW)), collect(values(EW))
```

```
(["München", "Köln", "Berlin", "Leipzig", "Hamburg"], [1.49, 1.08, 3.66, 0.597, 1.85])
```

11.3.4 Geordnetes Iterieren über ein Dictionary

Wir sortieren die Keys. Als Strings werden sie alphabetisch sortiert. Mit dem `rev`-Parameter wird rückwärts sortiert.

```
for k in sort(collect(keys(EW)), rev = true)
    n = EW[k]
    println("$k hat $n Millionen Einw.  ")
end
```

```
München hat 1.49 Millionen Einw.
Leipzig hat 0.597 Millionen Einw.
Köln hat 1.08 Millionen Einw.
Hamburg hat 1.85 Millionen Einw.
Berlin hat 3.66 Millionen Einw.
```

Wir sortieren `collect(dict)`. Das ist ein Vektor von Paaren. Mit `by` definieren wir, wonach zu sortieren ist: nach dem 2. Element des Paares.

```
for (k,v) in sort(collect(EW), by = pair -> last(pair), rev=false)
    println("$k hat $v Mill. EW")
end
```

```
Leipzig hat 0.597 Mill. EW
Köln hat 1.08 Mill. EW
München hat 1.49 Mill. EW
Hamburg hat 1.85 Mill. EW
Berlin hat 3.66 Mill. EW
```

11.3.5 Eine Anwendung von Dictionaries: Zählen von Häufigkeiten

Wir machen 'experimentelle Stochastik' mit 2 Würfeln:

Gegeben sei `l`, eine Liste mit den Ergebnissen von 100 000 Pasch-Würfen, also 100 000 Zahlen zwischen 2 und 12.

Wie häufig sind die Zahlen 2 bis 12?

Wir (lassen) würfeln:

```
l = rand(1:6, 100_000) .+ rand(1:6, 100_000)
```

```
100000-element Vector{Int64}:
  4
  7
  8
  3
  7
  8
  7
  9
  5
  7
 10
  8
  9
  ⋮
  6
  5
  8
  9
  9
 12
  8
  5
  7
  8
  7
 11
```

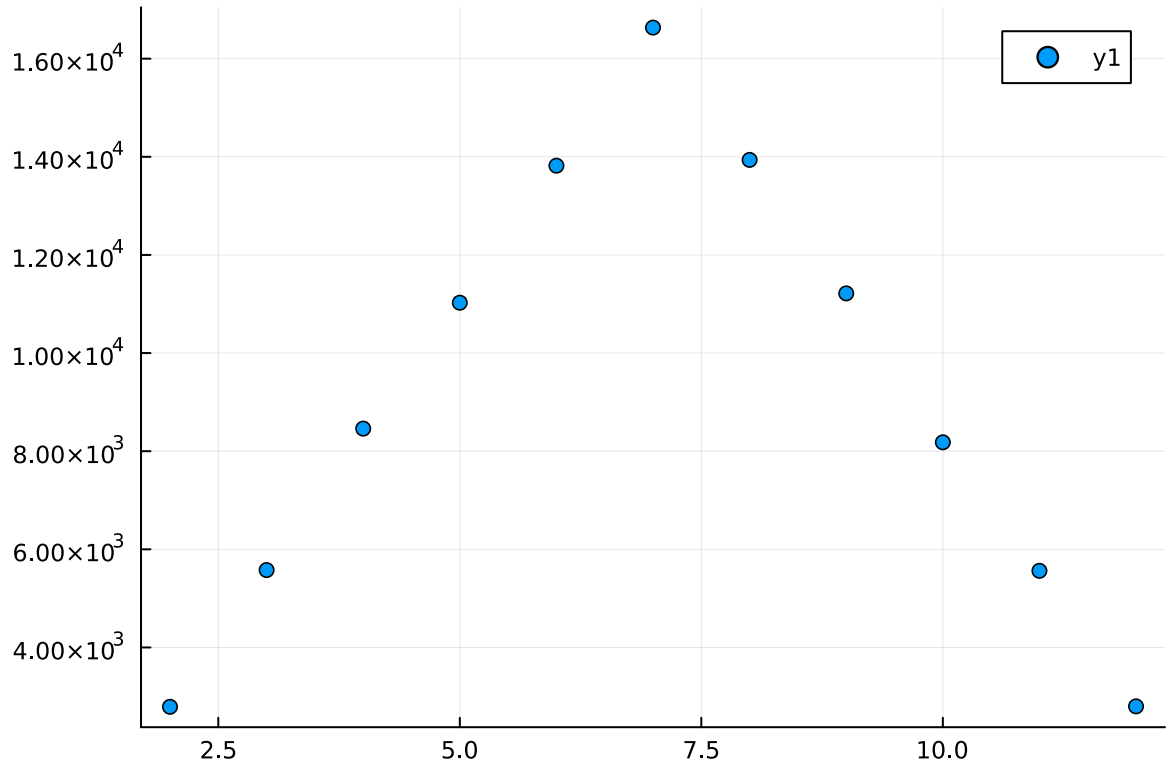
Wir zählen mit Hilfe eines Dictionaries die Häufigkeiten der Ereignisse. Dazu nehmen wir das Ereignis als `key` und seine Häufigkeit als `value`.

```
# In diesem Fall könnte man das auch mit einem einfachen Vektor
# lösen. Eine bessere Illustration wäre z.B. Worthäufigkeit in
# einem Text. Dann ist i keine ganze Zahl sondern ein Wort=String

d = Dict{Int,Int}()     # das Dict zum 'reinzählen'

for i in l                    # für jedes i wird d[i] erhöht.
    d[i] = get(d, i, 0) + 1
end
d
```

```
Dict{Int64, Int64} with 11 entries:
  5  => 11026
  12 => 2799
  8  => 13937
  6  => 13819
  11 => 5562
  9  => 11216
  3  => 5577
  7  => 16633
  4  => 8460
  2  => 2790
  10 => 8181
```

Das Ergebnis:

```
using Plots

plot(collect(keys(d)), collect(values(d)), seriestype=:scatter)
```



Das Erklär-Bild dazu:

https://math.stackexchange.com/questions/1204396/why-is-the-sum-of-the-rolls-of-two-dices-a-binomial-distribution-what-is-define

12 Vektoren, Matrizen, Arrays

12.1 Allgemeines

Kommen wir nun zu den wohl wichtigsten Containern für numerische Mathematik:

- Vektoren `Vector{T}`
- Matrizen `Matrix{T}` mit zwei Indizes

- N-dimensionale Arrays mit N Indizes `Array{T,N}`

Tatsächlich ist `Vector{T}` ein Alias für `Array{T,1}` und `Matrix{T}` ein Alias für `Array{T,2}`.

```
Vector{Float64} === Array{Float64,1} && Matrix{Float64} ===  Array{Float64,2}
```

```
true
```

Beim Anlegen durch eine explizite Elementliste wird der 'kleinste gemeinsame Typ' für den Typparameter `T` ermittelt.

```
v = [33, "33", 1.2]
```

```
3-element Vector{Any}:
 33
   "33"
  1.2
```

Falls `T` ein numerischer Typ ist, werden die Elemente in diesen Typ umgewandelt.

```
v = [3//7, 4, 2im]
```

```
3-element Vector{Complex{Rational{Int64}}}:
 3//7 + 0//1*im
 4//1 + 0//1*im
 0//1 + 2//1*im
```

Informationen über einen Array liefern die Funktionen:

- `length(A)` — Anzahl der Elemente
- `eltype(A)` — Typ der Elemente
- `ndims(A)` — Anzahl der Dimensionen (Indizes)
- `size(A)` — Tupel mit den Dimensionen des Arrays

```
v1 = [12, 13, 15]

m1 = [ 1    2.5
       6    -3 ]

for f ∈ (length, eltype, ndims, size)
    println("$(f)(v) = $(f(v)),          $(f)(m1) = $(f(m1))")
end
```

```
length(v) = 3,          length(m1) = 4
eltype(v) = Complex{Rational{Int64}},          eltype(m1) = Float64
ndims(v) = 1,          ndims(m1) = 2
size(v) = (3,),          size(m1) = (2, 2)
```

- Die Stärke des 'klassischen' Arrays für das wissenschaftliche Rechnen besteht darin, dass es einfach nur ein zusammenhängendes Speichersegment ist, in dem die Komponenten gleicher Länge (z.B. 64 Bit) geordnet hintereinander abgespeichert sind. Damit ist der Speicherbedarf minimal und die Zugriffsgeschwindigkeit auf eine Komponente, sowohl beim Lesen als auch beim Modifizieren, maximal. Der Platz der Komponente `v[i]` ist sofort aus `i` berechenbar.
- Julias `Array{T,N}` (und damit Vektoren und Matrizen) ist für die üblichen numerischen Typen `T` in dieser Weise implementiert. Die Elemente werden *unboxed* gespeichert. Im Gegensatz dazu ist z.B. ein `Vector{Any}` implementiert als Liste von Adressen von Objekten *(boxed)* und nicht als Liste der Objekte selbst.
- Julias `Array{T,N}` speichert seine Elemente direkt *(unboxed)*, wenn `isbitstype(T) == true`.

```
isbitstype(Float64),
isbitstype(Complex{Rational{Int64}}),
isbitstype(String)
```

```
(true, true, false)
```

12.2 Vektoren

12.2.1 Listen-artige Funktionen

- `push!(vector, items...)` — fügt Elemente am Ende des Vektors an
- `pushfirst!(vector, items...)` — fügt Elemente am Anfang des Vektors an
- `pop!(vector)` — entfernt letztes Element und liefert es als Ergebnis zurück,
- `popfirst!(vector)` — entfernt erstes Element und liefert es zurück

```
v = Float64[]           # leerer Vector{Float64}

push!(v, 3, 7)
pushfirst!(v, 1)

a = pop!(v)
println("a= $a")

push!(v, 17)
```

```
a= 7.0
3-element Vector{Float64}:
  1.0
  3.0
 17.0
```

Ein `push!()` kann sehr aufwändig sein, da eventuell neuer Speicher alloziert und dann der ganze bestehende Vektor umkopiert werden muss. Julia optimiert das Speichermanagement. Es wird in einem solchen Fall Speicher auf Vorrat alloziert, so dass weitere `push!`s sehr schnell sind und man 'fast O(1)-Geschwindigkeit' erreicht.

Trotzdem sollte man bei zeitkritischem Code und sehr großen Feldern Operationen wie `push!()` oder `resize()` vermeiden.

12.2.2 Weitere Konstruktoren

Man kann Vektoren mit vorgegebener Länge und Typ uninitialisiert anlegen. Das geht am Schnellsten, die Elemente sind zufällige Bitmuster.

```
# fixe Länge 1000, uninitialisiert

v = Vector{Float64}(undef, 1000)
v[345]
```

```
2.0052860224647e-311
```

- `zeros(n)` legt einen `Vector{Float64}` der Länge `n` an und initialisiert mit Null.

```
v = zeros(7)
```

```
7-element Vector{Float64}:
 0.0
 0.0
 0.0
 0.0
 0.0
 0.0
 0.0
```

- `zeros(T,n)` legt einen Nullvektor vom Typ `T` an.

```
v=zeros(Int, 4)
```

```
4-element Vector{Int64}:
 0
 0
 0
 0
```

- `fill(x, n)` legt `Vector{typeof(x)}` der Länge `n` an und füllt mit `x`.

```
v = fill(sqrt(2), 5)
```

```
5-element Vector{Float64}:
 1.4142135623730951
 1.4142135623730951
 1.4142135623730951
 1.4142135623730951
 1.4142135623730951
```

- `similar(v)` legt einen uninitialisierten Vektor von gleichem Typ und Größe wie `v` an.

```
w = similar(v)
```

```
5-element Vector{Float64}:
 4.0e-323
 0.0
 6.93719042537717e-310
 0.0
 0.0
```

12.2.3 Konstruktion durch implizite Schleife *(list comprehension)*

Implizite `for`-Schleifen sind eine weitere Methode, Vektoren zu erzeugen.

```
v4 = [i for i in 1.0:8]
```

```
8-element Vector{Float64}:
 1.0
 2.0
 3.0
 4.0
 5.0
 6.0
 7.0
 8.0
```

```
v5 = [log(i^2) for i in 1:4 ]
```

```
4-element Vector{Float64}:
 0.0
 1.3862943611198906
 2.1972245773362196
 2.772588722239781
```

Man kann sogar noch ein `if` unterbringen.

```
v6 = [i^2 for i in 1:8 if i%3 != 2]
```

```
5-element Vector{Int64}:
  1
  9
 16
 36
 49
```

12.2.4 Bitvektoren

Neben `Vector{Bool}` gibt es noch den speziellen Datentyp `BitVector` (und allgemeiner auch `BitArray`) zur Speicherung von Feldern mit Wahrheitswerten.

Während für die Speicherung eines `Bool`s ein Byte verwendet wird, erfolgt die Speicherung in einem BitVector bitweise.

Der Konstruktor wandelt einen `Vector{Bool}` in einen `BitVector` um.

```
vb = BitVector([true, false, true, true])
```

```
4-element BitVector:
 1
 0
 1
 1
```

Für die Gegenrichtung gibt es `collect()`.

```
collect(vb)
```

```
4-element Vector{Bool}:
 1
 0
 1
 1
```

BitVectoren entstehen z.B. als Ergebnis von elementweisen Vergleichen (s. Kapitel 12.7).

```
v4 .> 3.5
```

```
8-element BitVector:
 0
 0
 0
 1
 1
 1
 1
 1
```

12.2.5 Indizierung

Indizes sind Ordinalzahlen. Also **startet die Indexzählung mit 1**.

Als Index kann man verwenden:

- Integer
- Integer-wertigen Range (gleiche Länge oder kürzer)
- Integer-Vektor (gleiche Länge oder kürzer)
- Bool-Vektor oder BitVector (gleiche Länge)

Mit Indizes kann man Arrayelemente/teile lesen und schreiben.

```
v = [ 3i + 5.2 for i in 1:8]
```

```
8-element Vector{Float64}:
  8.2
 11.2
 14.2
 17.2
 20.2
 23.2
 26.2
 29.2
```

```
v[5]
```

```
20.2
```

Bei Zuweisungen wird die rechte Seite wenn nötig mit `convert(T,x)` in den Vektorelementetyp umgewandelt.

```
v[6] = 9999
v
```

```
8-element Vector{Float64}:
    8.2
   11.2
   14.2
```

```
  17.2
  20.2
9999.0
  26.2
  29.2
```

Überschreiten der Indexgrenzen führt zu einem `BoundsError`.

```
v[77]
```

```
LoadError: BoundsError: attempt to access 8-element Vector{Float64} at index [77]

Stacktrace:
 [1] getindex(A::Vector{Float64}, i1::Int64)
   @ Base ./essentials.jl:13
 [2] top-level scope
   @ In[22]:1
```

Mit einem `range`-Objekt kann man einen Teilvektor adressieren.

```
vp = v[3:5]
vp
```

```
3-element Vector{Float64}:
 14.2
 17.2
 20.2
```

```
vp = v[1:2:7]   # range mit Schrittweite
vp
```

```
4-element Vector{Float64}:
  8.2
 14.2
 20.2
 26.2
```

- Bei der Verwendung als Index kann in einem Range der Spezialwert `end` verwendet werden.
- Bei der Verwendung als Index kann der "leere" Range `:` als Abkürzung von `1:end` verwendet werden. Das ist nützlich bei Matrizen: `A[:, 3]` adressiert die gesamte 3. Spalte von `A`.

```
v[6:end] = [7, 7, 7]
v
```

```
8-element Vector{Float64}:
  8.2
 11.2
 14.2
 17.2
 20.2
  7.0
  7.0
  7.0
```

Indirekte Indizierung

Die indirekte Indizierung mit einem *Vector of Integers/Indices* erfolgt nach der Formel

$v[[i_1,\ i_2,\ i_3, ...]] = [\ v[i_1],\ v[i_2],\ v[i_3], ...]$

```
v[ [1, 3, 4] ]
```

```
3-element Vector{Float64}:
  8.2
 14.2
 17.2
```

ist also gleich

```
[ v[1], v[3], v[4] ]
```

```
3-element Vector{Float64}:
  8.2
 14.2
 17.2
```

Indizierung mit einem Vektor von Wahrheitswerten

Als Index kann man auch einen `Vector{Bool}` oder `BitVector` (s. Kapitel 12.2.4) **derselben Länge** verwenden.

```
v[ [true, true, false, false, true, false, true, true] ]
```

```
5-element Vector{Float64}:
  8.2
 11.2
 20.2
  7.0
  7.0
```

Das ist nützlich, da man z.B.

- Tests broadcasten kann (s. Kapitel 12.7),
- diese Tests dann einen BitVector liefern und
- bei Bedarf solche Bitvektoren durch die Bit-weisen Operatoren `&` und `|` verknüpft werden können.

```
v[  (v .> 13) .& (v.<20) ]
```

```
2-element Vector{Float64}:
 14.2
 17.2
```

12.3 Matrizen und Arrays

Die bisher vorgestellten Methoden für Vektoren übertragen sich auch auf höherdimensionale Arrays.

Man kann sie uninitialisiert anlegen:

```
A = Array{Float64,3}(undef, 6,9,3)
```

```
6×9×3 Array{Float64, 3}:
[:, :, 1] =
 6.93719e-310  2.5e-323      6.93719e-310  …  1.73e-322     2.69493e-312
 6.93719e-310  2.5e-323      6.93719e-310     1.73e-322     2.80103e-312
 2.0e-323      6.93719e-310  3.5e-323         6.93719e-310  2.71615e-312
 2.0e-323      6.93719e-310  3.5e-323         6.93719e-310  2.86469e-312
 6.93719e-310  3.0e-323      6.93719e-310     2.65249e-312  2.77981e-312
 6.93719e-310  3.0e-323      6.93719e-310  …  0.0           2.56761e-312

[:, :, 2] =
 1.4854e-313   2.94957e-312  3.20421e-312  …  3.92569e-312  3.67105e-312
 2.94957e-312  2.99201e-312  3.22543e-312     3.92569e-312  4.32887e-312
 2.94957e-312  6.97e-322     3.11933e-312     3.96813e-312  4.32887e-312
 2.94957e-312  3.03445e-312  2.90713e-312     9.24e-322     4.32887e-312
 2.94957e-312  3.03445e-312  1.9098e-313      3.86203e-312  4.32887e-312
 2.94957e-312  3.18299e-312  4.4e-323      …  3.69227e-312  4.32887e-312

[:, :, 3] =
 4.32887e-312  4.77449e-312  1.15e-321     …  6.02647e-312  6.93719e-310
 4.37131e-312  4.45619e-312  1.15e-321        6.04769e-312  6.93719e-310
 1.02e-321     4.49863e-312  1.15e-321        3.39519e-313  6.93719e-310
 4.51985e-312  4.20155e-312  1.15e-321        6.93719e-310  6.93719e-310
 4.56229e-312  4.03179e-312  1.15e-321        6.93719e-310  6.93719e-310
 4.68961e-312  1.15e-321     4.98669e-312  …  6.93719e-310  6.93719e-310
```

In den meisten Funktionen kann man die Dimensionen auch als Tupel übergeben. Die obige Anweisung lässt sich auch so schreiben:

```
A = Array{Float64, 3}(undef, (6,9,3))
```

Funktionen wie `zeros()` usw. funktionieren natürlich auch.

```
m2 = zeros(3, 4, 2)  # oder zeros((3,4,2))
```

```
3×4×2 Array{Float64, 3}:
[:, :, 1] =
 0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0

[:, :, 2] =
 0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0
```

```
M = fill(5 , (3, 3))   # oder fill(5, 3, 3)
```

```
3×3 Matrix{Int64}:
 5  5  5
 5  5  5
 5  5  5
```

Die Funktion `similar()`, die einen Array gleicher Größe uninitialisiert erzeugt, kann auch einen Typ als weiteres Argument bekommen.

```
M2 = similar(M, Float64)
```

```
3×3 Matrix{Float64}:
 6.93719e-310  0.0           6.93719e-310
 0.0           6.93719e-310  0.0
 6.93718e-310  0.0           6.93719e-310
```

12.3.1 Konstruktion durch explizite Elementliste

Während man Vektoren kommagetrennt in eckigen Klammern notiert, ist die Notation für höherdimensionale Objekte etwas anders.

- Eine Matrix:

```
M2 = [2  3  -1
      4  5  -2]
```

```
2×3 Matrix{Int64}:
 2  3  -1
 4  5  -2
```

- dieselbe Matrix:

```
M2 = [2 3 -1; 4 5 -2]
```

```
2×3 Matrix{Int64}:
 2  3  -1
 4  5  -2
```

- Ein Array mit 3 Indizes:

```
M3 = [2   3  -1
      4   5   6 ;;;
      7   8   9
     11  12  13]
```

```
2×3×2 Array{Int64, 3}:
[:, :, 1] =
 2  3  -1
 4  5   6

[:, :, 2] =
  7   8   9
 11  12  13
```

- und nochmal die Matrix `M2`:

```
M2 = [2;4;; 3;5;; -1;-2]
```

```
2×3 Matrix{Int64}:
 2  3  -1
 4  5  -2
```

Im letzten Beispiel kommen diese Regeln zur Anwendung:

- Trenner ist das Semikolon.
- Ein Semikolon `;` erhöht den 1. Index.
- Zwei Semikolons `;;` erhöhen den 2. Index.
- Drei Semikolons `;;;` erhöhen den 3. Index usw.

In den Beispielen davor wurde folgende Syntaktische Verschönerung (*syntactic sugar*) angewendet:

- Leerzeichen trennen wie 2 Semikolons – erhöht also den 2. Index: a_{12} a_{13} a_{14} ...
- Zeilenumbruch trennt wie ein Semikolon – erhöht also den 1. Index.

! Wichtig

- Vektorschreibweise mit Komma als Trenner geht nur bei Vektoren, nicht mit "Semikolon, Leerzeichen, Newline" mischen!
- Vektoren, $1 \times n$-Matrizen und $n \times 1$-Matrizen sind drei verschiedene Dinge!

```
v1 = [2,3,4]
```

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

```
v2 = [2;3;4]
```

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

```
v3 = [2 3 4]
```

```
1×3 Matrix{Int64}:
 2  3  4
```

```
v3 = [2;3;4;;;]
```

```
3×1 Matrix{Int64}:
 2
 3
 4
```

Einen "vector of vectors" a la C/C++ kann man natürlich auch konstruieren.

```
v = [[2,3,4], [5,6,7,8]]
```

```
2-element Vector{Vector{Int64}}:
 [2, 3, 4]
 [5, 6, 7, 8]
```

```
v[2][3]
```

```
7
```

Das sollte man nur in Spezialfällen tun. Die Array-Sprache von Julia ist in der Regel bequemer und schneller.

12.3.2 Indizes, Teilfelder, Slices

```
# 6x6 Matrix mit Zufallszahlen gleichverteilt aus [0,1) ∈ Float64
A = rand(6,6)
```

```
6×6 Matrix{Float64}:
 0.811067  0.96878    0.754231  0.0384932  0.418153  0.335135
 0.308752  0.764529   0.237113  0.750797   0.372172  0.0430564
 0.40899   0.842218   0.951231  0.174413   0.481761  0.892013
```

```
 0.620949  0.475068   0.248491  0.913321   0.578826  0.503095
 0.210374  0.0712402  0.84757   0.865595   0.810713  0.468191
 0.501687  0.556233   0.23264   0.971521   0.945795  0.697768
```

Die übliche Indexnotation:

```
A[2, 3] = 77.77777
A
```

```
6×6 Matrix{Float64}:
 0.811067  0.96878     0.754231  0.0384932  0.418153  0.335135
 0.308752  0.764529   77.7778    0.750797   0.372172  0.0430564
 0.40899   0.842218    0.951231  0.174413   0.481761  0.892013
 0.620949  0.475068    0.248491  0.913321   0.578826  0.503095
 0.210374  0.0712402   0.84757   0.865595   0.810713  0.468191
 0.501687  0.556233    0.23264   0.971521   0.945795  0.697768
```

Man kann mit Ranges Teilfelder adressieren:

```
B = A[1:2, 1:3]
```

```
2×3 Matrix{Float64}:
 0.811067  0.96878    0.754231
 0.308752  0.764529  77.7778
```

Das Adressieren von Teilen mit geringerer Dimension wird auch *slicing* genannt.

```
# die 3. Spalte als Vektor (slicing)

C = A[:, 3]
```

```
6-element Vector{Float64}:
  0.7542305303279327
 77.77777
  0.9512308709560736
  0.24849130999951186
  0.8475695032233193
  0.2326401648683074
```

```
# die 3. Zeile als Vektor (slicing)

E = A[3, :]
```

```
6-element Vector{Float64}:
 0.4089901544453163
 0.8422175146475817
 0.9512308709560736
 0.17441346014383508
 0.4817612970215346
 0.8920132401510948
```

Natürlich sind damit auch Zuweisungen möglich:

```
# Man kann slices und Teilfeldern auch etwas zuweisen

A[2, :] = [1,2,3,4,5,6]
A
```

```
6×6 Matrix{Float64}:
 0.811067  0.96878    0.754231  0.0384932  0.418153  0.335135
 1.0       2.0        3.0       4.0        5.0       6.0
 0.40899   0.842218   0.951231  0.174413   0.481761  0.892013
 0.620949  0.475068   0.248491  0.913321   0.578826  0.503095
 0.210374  0.0712402  0.84757   0.865595   0.810713  0.468191
 0.501687  0.556233   0.23264   0.971521   0.945795  0.697768
```

12.4 Verhalten bei Zuweisungen, `copy()` und `deepcopy()`, Views

12.4.1 Zuweisungen und Kopien

- Variablen sind Referenzen auf Objekte.
- *Eine Zuweisung zu einer Variablen erzeugt kein neues Objekt.*

```
A = [1, 2, 3]
B = A
```

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

`A` und `B` sind jetzt Namen desselben Objekts.

```
A[1] = 77
@show B;
```

```
B = [77, 2, 3]
```

```
B[3] = 300
@show A;
```

```
A = [77, 2, 300]
```

Dieses Verhalten spart viel Zeit und Speicher, ist aber nicht immer gewünscht. Die Funktion `copy()` erzeugt eine 'echte' Kopie des Objekts.

```
A = [1, 2, 3]
B = copy(A)
A[1] = 100
@show A B;
```

```
A = [100, 2, 3]
B = [1, 2, 3]
```

Die Funktion `deepcopy(A)` kopiert rekursiv. Auch von den Elementen, aus denen `A` besteht, werden (wieder rekursive) Kopien erstellt.

Solange ein Array nur primitive Objekte (Zahlen) enthält, sind `copy()` und `deepcopy()` äquivalent.

Das folgende Beispiel zeigt den Unterschied zwischen `copy()` und `deepcopy()`.

```
mutable struct Person
    name :: String
    age  :: Int
end
```

```
A = [Person("Meier", 20), Person("Müller", 21), Person("Schmidt", 23)]
B = A
C = copy(A)
D = deepcopy(A)
```

```
3-element Vector{Person}:
 Person("Meier", 20)
 Person("Müller", 21)
 Person("Schmidt", 23)
```

```
A[1] = Person("Mustermann", 83)
A[3].age = 199

@show B C D;
```

```
B = Person[Person("Mustermann", 83), Person("Müller", 21), Person("Schmidt", 199)]
C = Person[Person("Meier", 20), Person("Müller", 21), Person("Schmidt", 199)]
D = Person[Person("Meier", 20), Person("Müller", 21), Person("Schmidt", 23)]
```

12.4.2 Views

Wenn man mittels *indices/ranges/slices* einer Variablen ein Teilstück eines Arrays zuweist, wird von Julia grundsätzlich **ein neues Objekt** konstruiert.

```
A = [1  2  3
     3  4  5]

v = A[:, 2]
@show  v

A[1, 2] = 77
@show  A  v;
```

```
v = [2, 4]
A = [1 77 3; 3 4 5]
v = [2, 4]
```

Manchmal möchte man aber gerade hier eine Referenz-Semantik haben im Sinne von: "Vektor `v` soll der 2. Spaltenvektor von `A` sein und auch bleiben (d.h., sich mitändern, wenn sich `A` ändert)."

Dies bezeichnet man in Julia als *views*: Wir wollen, dass die Variable `v` nur einen 'alternativen Blick' auf die Matrix `A` darstellt.

Das kann man erreichen durch das `@view`-Macro:

```
A = [1 2 3
     3 4 5]

v = @view A[:,2]
@show  v

A[1, 2] = 77
@show  v;
```

```
v = [2, 4]
v = [77, 4]
```

Diese Technik wird von Julia aus Effizienzgründen auch bei einigen Funktionen der linearen Algebra verwendet. Ein Beispiel ist der Operator `'`, der zu einer Matrix `A` die adjungierte Matrix `A'` liefert.

- Die adjungierte *(adjoint)* Matrix `A'` ist die transponierte und elementweise komplex-konjugierte Matrix zu `A`.
- Der Parser macht daraus den Funktionsaufruf `adjoint(A)`.

- Für reelle Matrizen ist die Adjungierte gleich der transponierten Matrix.
- Julia implementiert `adjoint()` als *lazy function*, d.h.,
- es wird aus Effizienzgründen kein neues Objekt konstruiert, sondern nur ein alternativer 'View' auf die Matrix ("mit vertauschten Indizes") und ein alternativer 'View' auf die Einträge (mit Vorzeichenwechsel im Imaginärteil).
- Aus Vektoren macht `adjoint()` eine $1 \times n$-Matrix (einen Zeilenvektor).

```
A = [ 1.  2.
      3.  4.]
B = A'
```

```
2×2 adjoint(::Matrix{Float64}) with eltype Float64:
 1.0  3.0
 2.0  4.0
```

Die Matrix `B` ist nur ein modifizierter 'View' auf `A`:

```
A[1, 2] =10
B
```

```
2×2 adjoint(::Matrix{Float64}) with eltype Float64:
  1.0  3.0
 10.0  4.0
```

Aus Vektoren macht `adjoint()` eine $1 \times n$-Matrix (einen Zeilenvektor).

```
v = [1, 2, 3]
v'
```

```
1×3 adjoint(::Vector{Int64}) with eltype Int64:
 1  2  3
```

Eine weitere solche Funktion, die einen alternativen 'View', eine andere Indizierung, derselben Daten liefert, ist `reshape()`.

Hier wird ein Vektor mit 12 Einträgen in eine 3x4-Matrix verwandelt.

```
A = [1,2,3,4,5,6,7,8,9,10,11,12]

B = reshape(A, 3, 4)
```

```
3×4 Matrix{Int64}:
 1  4  7  10
 2  5  8  11
 3  6  9  12
```

12.5 Speicherung eines Arrays

- Speicher wird linear adressiert. Eine Matrix kann zeilenweise *(row major)* oder spaltenweise *(column major)* im Speicher angeordnet sein.
- C/C++/Python(NumPy) verwenden eine zeilenweise Speicherung: Die 4 Elemente einer 2x2-Matrix sind abgespeichert in der Reihenfolge $a_{11}, a_{12}, a_{21}, a_{22}$.
- Julia, Fortran, Matlab speichern spaltenweise: $a_{11}, a_{21}, a_{12}, a_{22}$.

Diese Information ist wichtig, um effizient über Matrizen zu iterieren:

```
function column_major_add(A, B)
    (n,m) = size(A)
    for j = 1:m
        for i = 1:n     # innere Schleife durchläuft eine Spalte
            A[i,j] += B[i,j]
        end
    end
end

function row_major_add(A, B)
    (n,m) = size(A)
    for i = 1:n
        for j = 1:m     # inere Schleife durchläuft eine Zeile
            A[i,j] += B[i,j]
        end
    end
end
```

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

```
A = rand(10000, 10000);
B = rand(10000, 10000);
```

```
using BenchmarkTools

@benchmark row_major_add($A, $B)
```

```
BenchmarkTools.Trial: 2 samples with 1 evaluation.
 Range (min … max):  3.572 s …   3.602 s  ┊ GC (min … max): 0.00% … 0.00%
 Time  (median):     3.587 s              ┊ GC (median):    0.00%
 Time  (mean ± σ):   3.587 s ± 21.158 ms  ┊ GC (mean ± σ):  0.00% ± 0.00%

  █                                                        █
  █▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁█ ▁
  3.57 s         Histogram: frequency by time         3.6 s <

 Memory estimate: 0 bytes, allocs estimate: 0.
```

```
@benchmark column_major_add($A, $B)
```

```
BenchmarkTools.Trial: 5 samples with 1 evaluation.
 Range (min … max):  1.187 s …   1.262 s  ┊ GC (min … max): 0.00% … 0.00%
 Time  (median):     1.193 s              ┊ GC (median):    0.00%
 Time  (mean ± σ):   1.212 s ± 33.109 ms  ┊ GC (mean ± σ):  0.00% ± 0.00%

  ██  █                             █                      █
  ██▁▁█▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁█▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁█ ▁
  1.19 s         Histogram: frequency by time        1.26 s <

 Memory estimate: 0 bytes, allocs estimate: 0.
```

12.5.1 Lokalität von Speicherzugriffen und *Caching*

Wir haben gesehen, dass die Reihenfolge von innerem und äußerem Loop einen erheblichen Geschwindigkeitsunterschied macht:

Es ist effizienter, wenn die innerste Schleife über den linken Index läuft, also eine Spalte und nicht eine Zeile durchläuft. Die Ursache dafür liegt in der Architektur moderner Prozessoren.

- Speicherzugriffe erfolgt über mehrere Cache-Ebenen.
- Ein *cache miss*, der ein Nachladen aus langsameren Caches auslöst, bremst aus.
- Es werden immer gleich größere Speicherblöcke nachgeladen, um die Häufigkeit von *cache misses* zu minimieren.
- Daher ist es wichtig, Speicherzugriffe möglichst lokal zu organisieren.

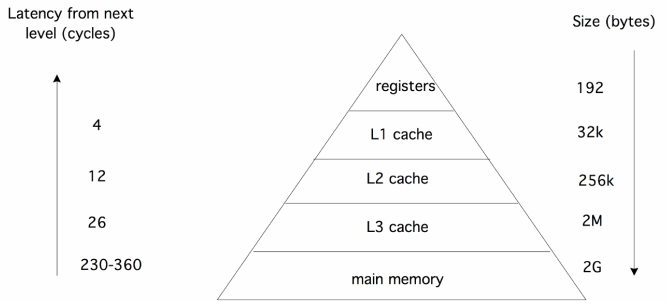


Figure 1.5: Memory hierarchy of an Intel Sandy Bridge, characterized by speed and size.

Abbildung 12.1: Speicherhierarchie von Intel-Prozessoren, aus: Victor Eijkhout,*Introduction to High-Performance Scientific Computing*, https://theartofhpc.com/

12.6 Mathematische Operationen mit Arrays

Arrays der gleichen Dimension (z.B. alle 7×3-Matrizen) bilden einen linearen Raum.

- Sie können mit Skalaren multipliziert werden und
- sie können addiert und subtrahiert werden.

```
0.5 * [2, 3, 4, 5]
```

```
4-element Vector{Float64}:
 1.0
 1.5
 2.0
 2.5
```

```
0.5 * [ 1  3
        2  7] - [ 2  3; 1 2]
```

```
2×2 Matrix{Float64}:
 -1.5  -1.5
  0.0   1.5
```

12.6.1 Matrixprodukt

Das Matrixprodukt ist definiert für

1. Faktor	2. Faktor	Produkt
$(n\times m)$-Matrix	$(m\times k)$-Matrix	$(n\times k)$-Matrix
$(n\times m)$-Matrix	m-Vektor	n-Vektor
$(1\times m)$-Zeilenvektor	$(m\times n)$-Matrix	n-Vektor
$(1\times m)$-Zeilenvektor	m-Vektor	Skalarprodukt
m-Vektor	$(1\times n)$-Zeilenvektor	$(m\times n)$-Matrix

Beispiele:

```
A = [1 2 3
     4 5 6]
v = [2, 3]
w = [1, 3, 4];
```

- (2,3)-Matrix ∗ (3,2)-Matrix

```
A * A'
```

```
2×2 Matrix{Int64}:
 14  32
 32  77
```

- (3,2)-Matrix ∗ (2,3)-Matrix

```
A' * A
```

```
3×3 Matrix{Int64}:
 17  22  27
 22  29  36
 27  36  45
```

- (2,3)-Matrix ∗ 3-Vektor

```
A * w
```

```
2-element Vector{Int64}:
 19
 43
```

- (1,2)-Vektore ∗ (2,3)-Matrix

```
v' * A
```

```
1×3 adjoint(::Vector{Int64}) with eltype Int64:
 14  19  24
```

- (3,2)-Matrix ∗ 2-Vektor

```
A' * v
```

```
3-element Vector{Int64}:
 14
 19
 24
```

- (1,2)-Vektor ∗ 2-Vektor (Skalarprodukt)

```
v' * v
```

```
13
```

2-Vektor ∗ (1,3)-Vektor (äußeres Produkt)

```
v * w'
```

```
2×3 Matrix{Int64}:
 2  6   8
 3  9  12
```

12.7 Broadcasting

- Beim *broadcasting* werden Operationen oder Funktionen **elementweise** auf Arrays angewendet.
- Die Syntax dafür ist ein Punkt *vor* einem Operationszeichen oder *nach* einem Funktionsnamen.
- Der Parser setzt `f.(x,y)` um zu `broadcast(f, x, y)` und analog für Operatoren `x .⊙ y` zu `broadcast(⊙, z, y)`.
- Dabei werden Operanden, denen eine oder mehrere Dimensionen fehlen, in diesen Dimensionen (virtuell) vervielfältigt.
- Das *broadcasting* von Zuweisungen `.=`, `.+=`,... verändert die Semantik. Es wird kein neues Objekt erzeugt, sondern die Werte werden in das links stehende Objekt (welches die richtige Dimension haben muss) eingetragen.

Einige Beispiele:

- Elementweise Anwendung einer Funktion

```
sin.([1, 2, 3])
```

```
3-element Vector{Float64}:
 0.8414709848078965
 0.9092974268256817
 0.1411200080598672
```

- Das Folgende liefert nicht die algebraische Wurzel aus einer Matrix, sondern die elementweise Wurzel aus jedem Eintrag.

```
A = [8 2
     3 4]

sqrt.(A)
```

```
2×2 Matrix{Float64}:
 2.82843  1.41421
 1.73205  2.0
```

- Das Folgende liefert nicht A^2, sondern die Einträge werden quadriert.

```
A.^2
```

```
2×2 Matrix{Int64}:
 64   4
  9  16
```

- Zum Vergleich das Ergebnis der algebraischen Operationen:

```
@show A^2 A^(1/2);
```

```
A ^ 2 = [70 24; 36 22]
A ^ (1 / 2) = [2.780234855920959 0.42449510866609885; 0.6367426629991483 1.9312446385887614]
```

- Broadcasting geht auch mit Funktionen mehrerer Variablen.

```
hyp(a,b) = sqrt(a^2+b^2)

B = [3 4
     5 7]

hyp.(A, B)
```

```
2×2 Matrix{Float64}:
 8.544    4.47214
 5.83095  8.06226
```

Bei Operanden verschiedener Dimension wird der Operand mit fehlenden Dimensionen in diesen durch Vervielfältigung virtuell 'aufgeblasen'.

Wir addieren einen Skalar zu einer Matrix:

```
A = [ 1 2 3
      4 5 6]
```

```
2×3 Matrix{Int64}:
 1  2  3
 4  5  6
```

```
A .+ 300
```

```
2×3 Matrix{Int64}:
 301  302  303
 304  305  306
```

Der Skalar wurde durch Replikation auf dieselbe Dimension wie die Matrix gebracht. Wir lassen uns von `broadcast()` die Form des 2. Operanden nach dem *broadcasting* anzeigen:

```
broadcast( (x,y) -> y, A, 300)
```

```
2×3 Matrix{Int64}:
 300  300  300
 300  300  300
```

(Natürlich findet diese Replikation nur virtuell statt. Dieses Objekt wird bei anderen Operationen nicht wirklich erzeugt.)

Als weiteres Beispiel: Matrix und (Spalten-)Vektor

```
A .+ [10, 20]
```

```
2×3 Matrix{Int64}:
 11  12  13
 24  25  26
```

Der Vektor wird durch Wiederholung der Spalten aufgeblasen:

```
broadcast((x,y)->y, A, [10,20])
```

```
2×3 Matrix{Int64}:
 10  10  10
 20  20  20
```

Matrix und Zeilenvektor: Der Zeilenvektor wird zeilenweise vervielfältigt:

```
A .* [1,2,3]'      # Adjungierter Vektor
```

```
2×3 Matrix{Int64}:
 1   4   9
 4  10  18
```

Der 2. Operand wird von `broadcast()` durch Vervielfältigung der Zeilen 'aufgeblasen'.

```
broadcast((x,y)->y, A, [1,2,3]')
```

```
2×3 Matrix{Int64}:
 1  2  3
 1  2  3
```

Broadcasting bei Zuweisungen

Zuweisungen `=`, `+=`, `/=`,..., bei denen links ein Name steht, laufen in Julia so ab, dass aus der rechten Seite ein Objekt konstruiert und diesem Objekt der neue Name zugewiesen wird.

Beim Arbeiten mit Arrays will man allerdings sehr oft aus Effizienzgründen einen bestehenden Array weiterverwenden. Die rechts berechneten Einträge sollen in das bereits existierende Objekt auf der linken Seite eingetragen werden.

Das erreicht man mit den Broadcast-Varianten `.=`, `.+=`,... der Zuweisungsoperatoren.

```
A .= 3
```

```
2×3 Matrix{Int64}:
 3  3  3
 3  3  3
```

```
A .+= [1, 4]
```

```
2×3 Matrix{Int64}:
 4  4  4
 7  7  7
```

12.8 Weitere Array-Funktionen - eine Auswahl

Julia stellt eine große Anzahl von Funktionen bereit, die mit Arrays arbeiten.

```
A = [22 -17 8 ; 4 6 9]
```

```
2×3 Matrix{Int64}:
 22  -17  8
  4    6  9
```

- Finde das Maximum

```
maximum(A)
```

```
22
```

- Finde das Maximum jeder Spalte

```
maximum(A, dims=1)
```

```
1×3 Matrix{Int64}:
 22  6  9
```

- Finde das Maximum jeder Zeile

```
maximum(A, dims=2)
```

```
2×1 Matrix{Int64}:
 22
  9
```

- Finde das Minimum und seine Position

```
amin, i = findmin(A)
```

```
(-17, CartesianIndex(1, 2))
```

- Was ist ein `CartesianIndex`?

```
dump(i)
```

```
CartesianIndex{2}
  I: Tuple{Int64, Int64}
    1: Int64 1
    2: Int64 2
```

- Extrahiere die Indizes des Minimum als Tupel

```
i.I
```

```
(1, 2)
```

- Summe und Produkt aller Einträge

```
sum(A), prod(A)
```

```
(32, -646272)
```

- Spaltensumme (1. Index wird reduziert)

```
sum(A, dims=1)
```

```
1×3 Matrix{Int64}:
 26  -11  17
```

- Zeilensummen (2. Index wird reduziert)

```
sum(A, dims=2)
```

```
2×1 Matrix{Int64}:
 13
 19
```

- Summiere nach elementweiser Anwendung einer Funktion

```
sum(x->sqrt(abs(x)), A)   #   sum_ij sqrt(|a_ij|)
```

```
19.09143825297046
```

- Reduziere (falte) den Array mit einer Funktion

```
reduce(+, A)    #  equivalent to sum(A)
```

```
32
```

- `mapreduce(f, op, array)`: Wende `f` auf alle Einträge an, dann reduziere mit `op`

```
mapreduce(x -> x^2, +, A )     # Summe der Quadrate aller Einträge
```

```
970
```

- Gibt es Elemente in A, die > 5 sind?

```
any(x -> x>5, A)
```

```
true
```

- Wieviele Elemente in A sind > 5?

```
count(x-> x>5, A)
```

```
4
```

- sind alle Einträge positiv?

```
all(x-> x>0, A)
```

```
false
```

13 Lineare Algebra in Julia

```
using LinearAlgebra
```

Das `LinearAlgebra`-Paket liefert unter anderem:

- zusätzliche Subtypen von `AbstractMatrix`: genauso verwendbar, wie andere Matrizen, z.B.
  - `Tridiagonal`
  - `SymTridiagonal`
  - `Symmetric`
  - `UpperTriangular`
- zusätzliche/erweiterte Funktionen: `norm`, `opnorm`, `cond`, `inv`, `det`, `exp`, `tr`, `dot`, `cross`, ...
- einen universellen Solver für lineare Gleichungssysteme: `\`
  - `x = A \ b` löst $A\mathbf{x} = \mathbf{b}$ durch geeignete Matrixfaktorisierung und Vorwärts/Rückwärtssubstition
- Matrixfaktorisierungen
  - `LU`
  - `QR`
  - `Cholesky`
  - `SVD`
  - ...
- Berechnung von Eigenwerte/-vektoren
  - `eigen`, `eigvals`, `eigvecs`
- Zugriff auf BLAS/LAPACK-Funktionen

13.1 Matrixtypen

```
A = SymTridiagonal(fill(1.0, 4), fill(-0.3, 3))
```

```
4×4 SymTridiagonal{Float64, Vector{Float64}}:
  1.0  -0.3    ⋅     ⋅
 -0.3   1.0  -0.3    ⋅
   ⋅   -0.3   1.0  -0.3
   ⋅     ⋅   -0.3   1.0
```

```
B = UpperTriangular(A)
```

```
4×4 UpperTriangular{Float64, SymTridiagonal{Float64, Vector{Float64}}}:
 1.0  -0.3   0.0   0.0
  ⋅    1.0  -0.3   0.0
  ⋅     ⋅    1.0  -0.3
  ⋅     ⋅     ⋅    1.0
```

Diese Typen werden platzsparend gespeichert. Die üblichen Rechenoperationen sind implementiert:

```
A + B
```

```
4×4 Matrix{Float64}:
  2.0  -0.6   0.0   0.0
 -0.3   2.0  -0.6   0.0
  0.0  -0.3   2.0  -0.6
  0.0   0.0  -0.3   2.0
```

Lesende Indexzugriffe sind möglich,

```
A[1,4]
```

```
0.0
```

schreibende nicht unbedingt:

```
A[1,3] = 17
```

```
LoadError: ArgumentError: cannot set off-diagonal entry (1, 3)

Stacktrace:
 [1] setindex!(A::SymTridiagonal{Float64, Vector{Float64}}, x::Int64, i::Int64, j::Int64)
   @ LinearAlgebra ~/.julia/juliaup/julia-1.10.2+0.x64.linux.gnu/share/julia/stdlib/v1.10/LinearAlgebra/src/tr
 [2] top-level scope
   @ In[7]:1
```

Die Umwandlung in eine 'normale' Matrix ist z.B. mit `collect()` möglich:

```
A2 = collect(A)
```

```
4×4 Matrix{Float64}:
  1.0  -0.3   0.0   0.0
 -0.3   1.0  -0.3   0.0
  0.0  -0.3   1.0  -0.3
  0.0   0.0  -0.3   1.0
```

13.1.1 Die Einheitsmatrix `I`

`I` bezeichnet eine Einheitsmatrix (quadratisch, Diagonalelemente = 1, alle anderen = 0) in der jeweils erforderlichen Größe

```
A + 4I
```

```
4×4 SymTridiagonal{Float64, Vector{Float64}}:
  5.0  -0.3    ⋅     ⋅
 -0.3   5.0  -0.3    ⋅
   ⋅   -0.3   5.0  -0.3
   ⋅     ⋅   -0.3   5.0
```

13.2 Normen

Um Fragen wie Kondition oder Konvergenz eines Algorithmus studieren zu können, brauchen wir eine Metrik. Für lineare Räume ist es zweckmäßig, die Metrik über eine Norm zu definieren:

$$d(x, y) := ||x - y||$$

13.2.1 p-Normen

Eine einfache Klasse von Normen im $\mathbb{R}^n$ sind die p-Normen

$$||\mathbf{x}||_p = \left(\sum |x_i|^p\right)^{\frac{1}{p}},$$

die die die euklidische Norm $p = 2$ verallgemeinern.

> **Die Max-Norm $p = \infty$**
>
> Sei $x_{\max}$ die *betragsmäßig* größte Komponente von $\mathbf{x} \in \mathbb{R}^n$. Dann gilt stets
>
> $$|x_{\max}| \leq ||\mathbf{x}||_p \leq n^{\frac{1}{p}} |x_{\max}|$$
>
> (Man betrachte einen Vektor, dessen Komponenten alle gleich $x_{\max}$ sind bzw. einen Vektor, dessen Komponenten außer $x_{\max}$ alle gleich Null sind.)
> Damit folgt
>
> $$\lim_{p\to\infty} ||\mathbf{x}||_p = |x_{\max}| =: ||\mathbf{x}||_\infty.$$

In Julia definiert das `LinearAlgebra`-Paket eine Funktion `norm(v, p)`.

```
v = [3, 4]
w = [-1, 2, 33.2]

@show norm(v)  norm(v, 2)  norm(v, 1) norm(v, 4) norm(w, Inf);
```

```
norm(v) = 5.0
norm(v, 2) = 5.0
norm(v, 1) = 7.0
norm(v, 4) = 4.284572294953817
norm(w, Inf) = 33.2
```

- Wenn das 2. Argument `p` fehlt, wird `p=2` gesetzt.
- Das 2. Argument kann auch `Inf` (also $+\infty$) sein.
- Das 1. Argument kann ein beliebiger Container voller Zahlen sein. Die Summe $\sum |x_i|^p$ erstreckt sich über *alle* Elemente des Containers.
- Damit ist für eine Matrix `norm(A)` gleich der *Frobenius-Norm* der Matrix `A`.

```
A = [1 2 3
     4 5 6
     7 8 9]
norm(A)      # Frobenius norm
```

```
16.881943016134134
```

Da Normen homogen unter Multiplikation mit Skalaren sind, $||\lambda\mathbf{x}|| = |\lambda| \cdot ||\mathbf{x}||$, sind sie durch die Angabe der Einheitskugel vollständig bestimmt. Subadditivität der Norm (Dreiecksungleichung) ist äquivalent zur Konvexität der Einheitskugel (Code durch anklicken sichtbar).

```
using Plots

colors=[:purple, :green, :red, :blue,:aqua, :black]
x=LinRange(-1, 1, 1000)
y=LinRange(-1, 1, 1000)

fig1=plot()
for p ∈ (0.8, 1, 1.5, 2, 3.001, 1000)
    contour!(x,y, (x,y) -> p * norm([x, y], p), levels=[p], aspect_ratio=1,
        cbar=false, color=[pop!(colors)], contour_labels=true, ylim=[-1.1, 1.1])
```

```
end
fig1
```

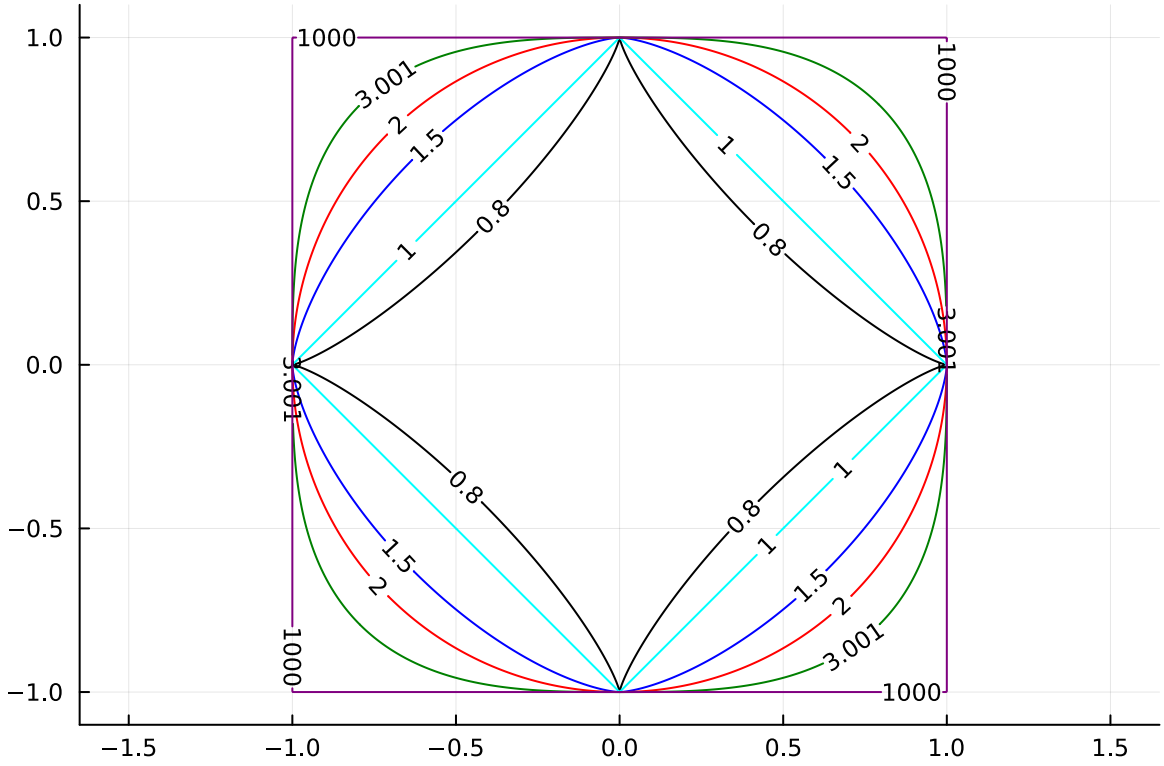


Abbildung 13.1: Einheitskugeln im $\mathbb{R}^2$ für verschiedene p-Normen: p=0.8; 1; 1.5; 2; 3.001 und 1000

Wie man sieht, muß $p \geq 1$ sein, damit die Einheitskugel konvex und $||.||_p$ eine Norm ist.

Die Julia-Funktion `norm(v, p)` liefert allerdings für beliebige Parameter `p` ein Ergebnis.

13.2.2 Induzierte Normen (Operatornormen)

Matrizen A repräsentieren lineare Abbildungen $\mathbf{v} \mapsto A\mathbf{v}$. Die von einer Vektornorm Induzierte Matrixnorm beantwortet die Frage:

> *„Um welchen Faktor kann ein Vektor durch die Transformation A maximal gestreckt werden?“*

Auf Grund der Homogenität der Norm unter Multiplikation mit Skalaren reicht es aus, das Bild der Einheitskugel unter der Transformation A zu betrachten.

Definition

Sei V ein Vektorraum mit einer Dimension $0 < n < \infty$ und A eine $n \times n$-Matrix. Dann ist

$$||A||_p = \max_{||\mathbf{v}||_p=1} ||A\mathbf{v}||_p$$

Induzierte Normen lassen sich für allgemeines p nur schwer berechnen. Ausnahmen sind die Fälle

- $p = 1$: Spaltensummennorm
- $p = 2$: Spektralnorm und
- $p = \infty$: Zeilensummennorm

Diese 3 Fälle sind in Julia in der Funktion `opnorm(A, p)` aus dem `LinearAlgebra`-Paket implementiert, wobei wieder `opnorm(A) = opnorm(A, 2)` gilt.

```
A = [ 0    1
      1.2  1.5 ]

@show opnorm(A, 1) opnorm(A, Inf) opnorm(A, 2) opnorm(A);
```

```
opnorm(A, 1) = 2.5
opnorm(A, Inf) = 2.7
opnorm(A, 2) = 2.0879899930905124
opnorm(A) = 2.0879899930905124
```

Das folgende Bild zeigt die Wirkung von A auf Einheitsvektoren. Vektoren gleicher Farbe werden aufeinander abgebildet. (Code durch anklicken sichtbar):

```
using CairoMakie

# Makie bug  https://github.com/MakieOrg/Makie.jl/issues/3255
# Würgaround https://github.com/MakieOrg/Makie.jl/issues/2607#issuecomment-1385816645
tri = BezierPath([
    MoveTo(Point2f(-0.5, -1)), LineTo(0, 0), LineTo(0.5, -1), ClosePath()
])

A = [ 0    1
      1.2  1.5 ]

t = LinRange(0, 1, 100)
xs = sin.(2π*t)
ys = cos.(2π*t)
Axys = A * [xs, ys]

u = [sin(n*π/6) for n=0:11]
v = [cos(n*π/6) for n=0:11]
x = y = zeros(12)

Auv = A * [u,v]

fig2 = Figure(size=(800, 400))
lines(fig2[1, 1], xs, ys, color=t, linewidth=5, colormap=:hsv, axis=(; aspect = 1, limits=(-2,2, -2,2),
      title=L"$\mathbf{v}$", titlesize=30))
arrows!(fig2[1,1], x, y, u, v, arrowsize=10, arrowhead=tri, colormap=:hsv, linecolor=range(0,11), linewidth=3)

Legend(fig2[1,2], MarkerElement[], String[], L"⇒", width=40, height=30, titlesize=30, framevisible=false)

lines(fig2[1,3], Axys[1], Axys[2], color=t, linewidth=5, colormap=:hsv, axis=(; aspect=1, limits=(-2,2, -2,2),
      title=L"$A\mathbf{v}$", titlesize=30))
arrows!(fig2[1,3], x, y, Auv[1], Auv[2], arrowsize=10, arrowhead=tri, colormap=:hsv, linecolor=range(0,11),
      linewidth=3)

fig2
```

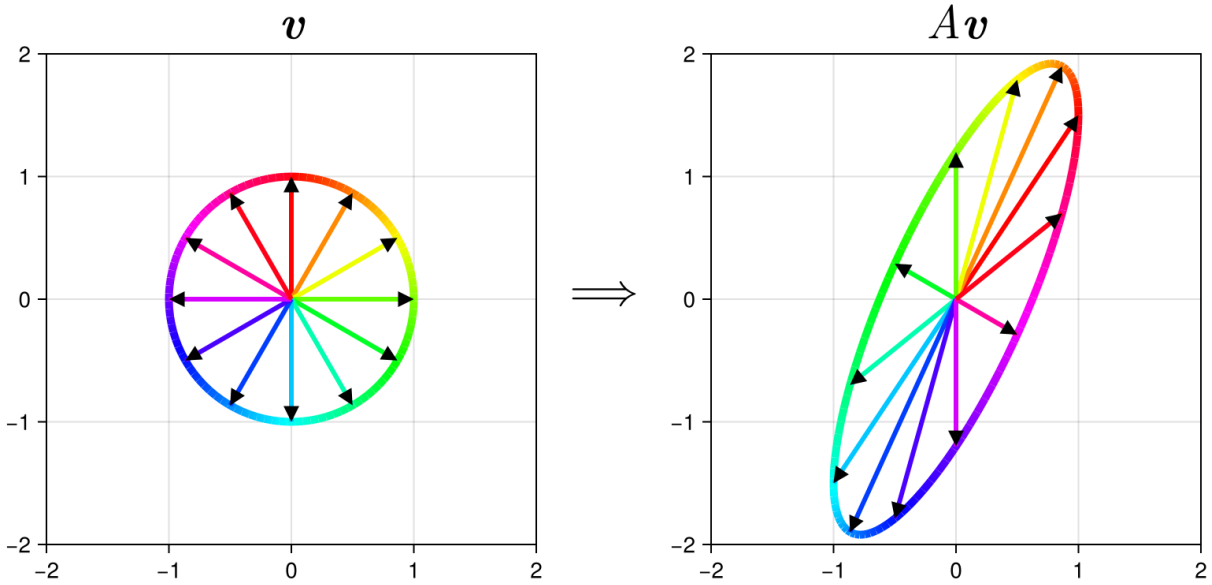


Abbildung 13.2: Bild der Einheitskugel unter $v \mapsto Av$ mit $||A|| \approx 2.088$

13.2.3 Konditionszahl

Für p = 1, p = 2 (default) oder p = Inf liefert `cond(A,p)` die Konditionszahl in der p-Norm

$$\text{cond}_p(A) = ||A||_p \cdot ||A^{-1}||_p$$

```
@show cond(A, 1)  cond(A, 2) cond(A)  cond(A, Inf);
```

```
cond(A, 1) = 5.625
cond(A, 2) = 3.633085176038432
cond(A) = 3.633085176038432
cond(A, Inf) = 5.625000000000001
```

13.3 Matrixfaktorisierungen

Basisaufgaben der numerischen linearen Algebra:

- Löse ein lineares Gleichungssystem $A\mathbf{x} = \mathbf{b}$.
- Falls keine Lösung existiert, finde die beste Annäherung, d.h., den Vektor $\mathbf{x}$, der $||A\mathbf{x} - \mathbf{b}||$ minimiert.
- Finde Eigenwerte und Eigenvektoren $A\mathbf{x} = \lambda\mathbf{x}$ von A.

Diese Aufgaben sind mit Matrixfaktorisierungen lösbar. Einige grundlegende Matrixfaktorisierungen:

- **LU-Zerlegung** $A = L \cdot U$
  - faktorisiert eine Matrix als Produkt einer *lower* und einer *upper* Dreiecksmatrix
  - im Deutschen auch LR-Zerlegung (aber die Julia-Funktion heisst `lu()`)
  - geht (eventuell nach Zeilenvertauschung - Pivoting) immer
- **Cholesky-Zerlegung** $A = L \cdot L^*$
  - die obere Dreiecksmatrix ist die konjugierte der unteren,
  - halber Aufwand im Vergleich zu LU
  - geht nur, wenn A hermitesch und positiv definit ist

- **QR-Zerlegung** $A = Q \cdot R$
  - zerlegt A als Produkt einer orthogonalen (bzw. unitären im komplexen Fall) Matrix und einer oberen Dreiecksmatrix
  - Q ist längenerhaltend (Drehungen und/oder Spiegelungen); die Stauchungen/Streckungen werden durch R beschrieben
  - geht immer
- **SVD** *(Singular value decomposition)*: $A = U \cdot D \cdot V^*$
  - U und V sind orthogonal (bzw. unitär), D ist eine Diagonalmatrix mit Einträgen in der Diagonale $\sigma_i \geq 0$, den sogenannten *Singulärwerten* von A.
  - Jede lineare Transformation $\mathbf{v} \mapsto A\mathbf{v}$ läßt sich somit darstellen als eine Drehung (und/oder Spiegelung) V^*, gefolgt von einer reinen Skalierung $v_i \mapsto \sigma_i v_i$ und einer weitere Drehung U.

13.3.1 LU-Faktorisierung

LU-Faktorisierung ist Gauß-Elimination. Das Resultat der Gauß-Elimination ist die obere Dreiecksmatrix U. Die untere Dreiecksmatrix L enthält Einsen auf der Diagonale und die nichtdiagonalen Einträge l_{ij} sind gleich minus den Koeffizienten, mit denen im Gauß-Algorithmus Zeile Z_j multipliziert und zu Zeile Z_i addiert wird. Ein Beispiel:

$$A = \begin{bmatrix} 1 & 2 & 2 \\ 3 & -4 & 4 \\ -2 & 1 & 5 \end{bmatrix} \begin{matrix} \\ Z_2 \mapsto Z_2 - 3Z_1 \\ Z_3 \mapsto Z_3 + 2Z_1 \end{matrix} \implies \begin{bmatrix} 1 & 2 & 2 \\ & -10 & -2 \\ & 5 & 9 \end{bmatrix} \begin{matrix} \\ \\ Z_3 \mapsto Z_3 + \frac{1}{2}Z_2 \end{matrix} \implies \begin{bmatrix} 1 & 2 & 2 \\ & -10 & -2 \\ & & 8 \end{bmatrix} = U$$

$$A = \begin{bmatrix} 1 & & \\ +3 & 1 & \\ -2 & -\frac{1}{2} & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 2 & 2 \\ & -10 & -2 \\ & & 8 \end{bmatrix}$$

- Häufig in der Praxis: $A\mathbf{x} = \mathbf{b}$ muss für ein A und viele rechte Seiten $\mathbf{b}$ gelöst werden.
- Die Faktorisierung, deren Aufwand kubisch $\sim n^3$ mit der Matrixgröße n wächst, muss nur einmal gemacht werden.
- Der anschliessende Aufwand der Vorwärts/Rückwärtssubstition für jedes $\mathbf{b}$ ist nur noch quadratisch $\sim n^2$.

Das `LinearAlgebra`-Paket von Julia enthält zur Berechnung einer LU-Zerlegung die Funktion `lu(A, options)`:

```
A = [ 1   2  2
      3  -4  4
     -2   1  5]

L, U, _ = lu(A, NoPivot())
display(L)
display(U)
```

```
3×3 Matrix{Float64}:
  1.0   0.0  0.0
  3.0   1.0  0.0
 -2.0  -0.5  1.0
3×3 Matrix{Float64}:
 1.0    2.0   2.0
 0.0  -10.0  -2.0
 0.0    0.0   8.0
```

Pivoting

Sehen wir uns einen Schritt der Gauß-Elimination an:

$$\begin{bmatrix} * & \cdots & * & * & \cdots & * \\ & \ddots & \vdots & \vdots & & \vdots \\ & & * & * & \cdots & * \\ & & & a_{ij} & \cdots & a_{in} \\ & & & a_{i+1,j} & \cdots & a_{i+1,n} \\ & & & \vdots & & \vdots \\ & & & a_{mj} & \cdots & a_{mn} \end{bmatrix}$$

Ziel ist es, als nächstes den Eintrag $a_{i+1,j}$ zum Verschwinden zu bringen, indem zur Zeile Z_{i+1} ein geeignetes Vielfaches von Zeile Z_i addiert wird. Das geht nur, wenn das *Pivotelement* a_{ij} nicht Null ist. Falls $a_{ij} = 0$, müssen wir Zeilen vertauschen um dies zu beheben.

Darüber hinaus ist die Kondition des Algorithmus am besten, wenn man bei jedem Schritt die Matrix so anordnet, dass das Pivotelement das betragsmäßig größte in der entsprechenden Spalte der noch zu bearbeitenden Umtermatrix ist. Beim (Zeilen-)Pivoting wird bei jedem Schritt durch Zeilenvertauschung sichergestellt, dass gilt:

$$|a_{ij}| = \max_{k=i,\ldots,m} |a_{kj}|$$

LU in Julia

- Die Faktorisierungen in Julia geben ein spezielles Objekt zurück, das die Matrixfaktoren und weitere Informationen enthält.
- Die Julia-Funktion `lu(A)` führt eine LU-Faktorisierung mit Pivoting durch.

```
F = lu(A)
typeof(F)
```

```
LU{Float64, Matrix{Float64}, Vector{Int64}}
```

Elemente des Objekts:

```
@show F.L F.U F.p;
```

```
F.L = [1.0 0.0 0.0; 0.3333333333333333 1.0 0.0; -0.6666666666666666 -0.5 1.0]
F.U = [3.0 -4.0 4.0; 0.0 3.333333333333333 0.6666666666666667; 0.0 0.0 8.0]
F.p = [2, 1, 3]
```

Man kann auch gleich auf der linken Seite ein entsprechendes Tupel verwenden:

```
L, U, p = lu(A);
p
```

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

Der Permutationsvektor zeigt an, wie die Zeilen der Matrix permutiert wurden. Es gilt:

$$L \cdot U = PA$$

. Die Syntax der indirekten Indizierung erlaubt es, die Zeilenpermutation durch die Schreibweise `A[p,:]` anzuwenden:

```
display(A)
display(A[p,:])
display(L*U)
```

```
3×3 Matrix{Int64}:
  1   2  2
  3  -4  4
 -2   1  5
3×3 Matrix{Int64}:
  3  -4  4
  1   2  2
 -2   1  5
3×3 Matrix{Float64}:
  3.0  -4.0  4.0
  1.0   2.0  2.0
 -2.0   1.0  5.0
```

Die Vorwärts/Rückwärtssubstition mit einem `LU`- erledigt der Operator `\`:

```
b = [1, 2, 3]
x = F \ b
```

```
3-element Vector{Float64}:
 -0.10000000000000002
 -0.012500000000000006
  0.5625
```

Probe:

```
A * x - b
```

```
3-element Vector{Float64}:
 0.0
 0.0
 0.0
```

In Julia verbirgt sich hinter dem `\`-Operator ein ziemlich universeller "matrix solver" und man kann ihn auch direkt anwenden:

```
A \ b
```

```
3-element Vector{Float64}:
 -0.10000000000000002
 -0.012500000000000006
  0.5625
```

Dabei wird implizit eine geeignete Faktorisierung durchgeführt, deren Ergebnis allerdings nicht abgespeichert.

13.3.2 QR-Zerlegung

Die Funktion `qr()` liefert ein epezielles QR-Objekt zurück, das die Komponenten Q und R enthält. Die orthogonale (bzw. unitäre) Matrix Q ist in einer optimierten Form abgespeichert. Umwandlung in eine "normale" Matrix ist bei Bedarf wie immer mit `collect()` möglich.

```
F = qr(A)
@show typeof(F) typeof(F.Q)
display(collect(F.Q))
```

```
display(F.R)
```

```
typeof(F) = LinearAlgebra.QRCompactWY{Float64, Matrix{Float64}, Matrix{Float64}}
typeof(F.Q) = LinearAlgebra.QRCompactWYQ{Float64, Matrix{Float64}, Matrix{Float64}}
3×3 Matrix{Float64}:
 -0.267261   0.872872  0.408248
 -0.801784  -0.436436  0.408248
  0.534522  -0.218218  0.816497
3×3 Matrix{Float64}:
 -3.74166  3.20713  -1.06904
  0.0      3.27327  -1.09109
  0.0      0.0       6.53197
```

13.3.3 Passende Faktorisierung

Die Funktion `factorize()` liefert eine dem Matrixtyp angepasste Form der Faktorisierung, siehe Dokumentation für Details. Wenn man Lösungen zu mehreren rechten Seiten $\mathbf{b}_1, \mathbf{b}_2, \ldots$ benötigt, sollte man die Faktorisierung nur einmal durchführen:

```
Af = factorize(A)
```

```
LU{Float64, Matrix{Float64}, Vector{Int64}}
L factor:
3×3 Matrix{Float64}:
  1.0        0.0  0.0
  0.333333   1.0  0.0
 -0.666667  -0.5  1.0
U factor:
3×3 Matrix{Float64}:
 3.0  -4.0      4.0
 0.0   3.33333  0.666667
 0.0   0.0      8.0
```

```
Af \ [1, 2, 3]
```

```
3-element Vector{Float64}:
 -0.10000000000000002
 -0.012500000000000006
  0.5625
```

```
Af \ [5, 7, 9]
```

```
3-element Vector{Float64}:
 0.4000000000000001
 0.42500000000000004
 1.875
```

14 Zeichen, Strings und Unicode

14.1 Zeichencodierungen (Frühgeschichte)

Es gab - abhängig von Hersteller, Land, Programmiersprache, Betriebsssystem,... - eine große Vielzahl von Codierungen.

Bis heute relevant sind:

14.1.1 ASCII

Der *American Standard Code for Information Interchange* wurde 1963 in den USA als Standard veröffentlicht.

- Er definiert $2^7 = 128$ Zeichen, und zwar:
  - 33 Steuerzeichen, wie `newline`, `escape`, `end of transmission/file`, `delete`
  - 95 graphisch darstellbare Zeichen:
    - 52 lateinische Buchstaben `a-z, A-Z`
    - 10 Ziffern `0-9`
    - 7 Satzzeichen `.,:;?!"`
    - 1 Leerzeichen
    - 6 Klammern `[{()}]`
    - 7 mathematische Operationen `+-*/<>=`
    - 12 Sonderzeichen `#$%&'\^_|~`@`
- ASCII ist heute noch der "kleinste gemeinsame Nenner" im Codierungs-Chaos.
- Die ersten 128 Unicode-Zeichen sind identisch mit ASCII.

14.1.2 ISO 8859-Zeichensätze

- ASCII nutzt nur 7 Bits.
- In einem Byte kann man durch Setzen des 8. Bits weitere 128 Zeichen unterbringen.
- 1987/88 wurden im ISO 8859-Standard verschiedene 1-Byte-Codierungen festgelegt, die alle ASCII-kompatibel sind, darunter:

Codierung	Region	Sprachen
ISO 8859-1 (Latin-1)	Westeuropa	Deutsch, Französisch,...,Isländisch
ISO 8859-2 (Latin-2)	Osteuropa	slawische Sprachen mit lateinischer Schrift
ISO 8859-3 (Latin-3)	Südeuropa	Türkisch, Maltesisch,...
ISO 8859-4 (Latin-4)	Nordeuropa	Estnisch, Lettisch, Litauisch, Grönländisch, Sami
ISO 8859-5 (Latin/Cyrillic)	Osteuropa	slawische Sprachen mit kyrillischer Schrift
ISO 8859-6 (Latin/Arabic)		
ISO 8859-7 (Latin/Greek)		
...		
ISO 8859-15 (Latin-9)		1999: Revision von Latin-1: jetzt mit Euro-Zeichen!

14.2 Unicode

Das Ziel des Unicode-Consortiums ist eine einheitliche Codierung für alle Schriften der Welt.

- Unicode Version 1 erschien 1991
- Unicode Version 15 erschien 2021 mit 149 186 Zeichen (das sind 4489 mehr als Unicode 14), darunter:
  - 161 Schriften
  - mathematische und technische Symbole
  - Emojis und andere Symbole, Steuer- und Formatierungszeichen
- davon entfallen über 90 000 Zeichen auf die CJK-Schriften (Chinesisch/Japanisch/Koreanisch)

14.2.1 Technische Details

- Jedem Zeichen wird ein `codepoint` zugeordnet. Das ist einfach eine fortlaufende Nummer.
- Diese Nummer wird hexadezimal notiert
  - entweder 4-stellig als `U+XXXX` (0-te Ebene)
  - oder 5…6-stellig als `U+XXXXXX` (weitere Ebenen)
- Jede Ebene geht von `U+XY0000` bis `U+XYFFFF`, kann also $2^{16} = 65\,534$ Zeichen enthalten.
- Vorgesehen sind bisher 17 Ebenen `XY=00` bis `XY=10`, also der Wertebereich von `U+0000` bis `U+10FFFF`.
- Damit sind maximal 21 Bits pro Zeichen nötig.
- Die Gesamtzahl der damit möglichen Codepoints ist etwas kleiner als 0x10FFFF, da aus technischen Gründen gewisse Bereiche nicht verwendet werden. Sie beträgt etwa 1.1 Millionen, es ist also noch viel Platz.
- Bisher wurden nur Codepoints aus den Ebenen
  - Ebene 0 = BMP *Basic Multilingual Plane* `U+0000 - U+FFFF`,
  - Ebene 1 = SMP *Supplementary Multilingual Plane* `U+010000 - U+01FFFF`,
  - Ebene 2 = SIP *Supplementary Ideographic Plane* `U+020000 - U+02FFFF`,
  - Ebene 3 = TIP *Tertiary Ideographic Plane* `U+030000 - U+03FFFF` und
  - Ebene 14 = SSP *Supplementary Special-purpose Plane* `U+0E0000 - U+0EFFFF` vergeben.
- `U+0000` bis `U+007F` ist identisch mit ASCII
- `U+0000` bis `U+00FF` ist identisch mit ISO 8859-1 (Latin-1)

14.2.2 Eigenschaften von Unicode-Zeichen

Im Standard wird jedes Zeichen beschrieben duch

- seinen Codepoint (Nummer)
- einen Namen (welcher nur aus ASCII-Großbuchstaben, Ziffern und Minuszeichen besteht) und
- verschiedene Attributen wie
  - Laufrichtung der Schrift
  - Kategorie: Großbuchstabe, Kleinbuchstabe, modifizierender Buchstabe, Ziffer, Satzzeichen, Symbol, Seperator,….

Im Unicode-Standard sieht das dann so aus (zur Vereinfachung nur Codepoint und Name):

```
...
U+0041 LATIN CAPITAL LETTER A
U+0042 LATIN CAPITAL LETTER B
U+0043 LATIN CAPITAL LETTER C
U+0044 LATIN CAPITAL LETTER D
...
U+00E9 LATIN SMALL LETTER E WITH ACUTE
U+00EA LATIN SMALL LETTER E WITH CIRCUMFLEX
...
```

```
U+0641 ARABIC LETTER FEH
U+0642 ARABIC LETTER QAF
...
U+21B4 RIGHTWARDS ARROW WITH CORNER DOWNWARDS
...
```

Wie sieht ‘RIGHTWARDS ARROW WITH CORNER DOWNWARDS’ aus?

```
'\U21b4'
```

```
'↴': Unicode U+21B4 (category So: Symbol, other)
```

14.2.3 Eine Auswahl an Schriften

Eine kleine Hilfsfunktion:

```
"""
printuc(c, n):
print n characters from unicode table, starting with character c
"""
function printuc(c, n)
    for i in 0:n-1
        print(c + i)
    end
end
```

```
printuc
```

Kyrillisch

```
printuc('\U0400', 100)
```

```
ЀЁЂЃЄЅІЇЈЉЊЋЌЍЎЏАБВГДЕЖЗИЙКЛМНОПРСТУФХЦЧШЩЪЫЬЭЮЯабвгдежзийклмнопрстуфхцчшщъыьэюяѐёђѓєѕіїјљњћќѝўџѠѡѢ
```

Tamilisch

```
printuc('\U0be7',20)
```

```
௧௨௩௪௫௬௭௮௯௰௱௲௳௴௵௶௷௸
```

Schach

```
printuc('\U2654', 12)
```

```
♔♕♖♗♘♙♚♛♜♝♞♟
```

Mathematische Operatoren

```
printuc('\U2200', 255)
```

```
∀∁∂∃∄∅∆∇∈∉∊∋∌∍■∏∐∑−∓∔∕∖∗∘∙√∛∜∝∞∟∠∡∢∣∤∥∦∧∨∩∪∫∬∭∮∯∰∱∲∳∴∵∶∷∸∹∺∻∼∽∾∿≀≁≂≃≄≅≆≇≈≉≊≋≌≍≎≏≐≑≒≓≔≕≖≗≘≙≚≛≜≝≞≟≠≡≢≣≤≥≦≧≨≩≪≫≬≭
```

Runen

```
printuc('\U16a0', 40)
```

```
ᚠᚡᚢᚣᚤᚥᚦᚧᚨᚩᚪᚫᚬᚭᚮᚯᚰᚱᚲᚳᚴᚵᚶᚷᚸᚹᚺᚻᚼᚽᚾᚿᛀᛁᛂᛃᛄᛅᛆᛇ
```

Scheibe (Diskus) von Phaistos

- Diese Schrift ist nicht entziffert.
- Es ist unklar, welche Sprache dargestellt wird.
- Es gibt nur ein einziges Dokument in dieser Schrift: die Tonscheibe von Phaistos aus der Bronzezeit

```
printuc('\U101D0', 46 )
```

14.2.4 Unicode transformation formats: UTF-8, UTF-16, UTF-32

Unicode transformation formats legen fest, wie eine Folge von Codepoints als eine Folge von Bytes dargestellt wird.

Da die Codepoints unterschiedlich lang sind, kann man sie nicht einfach hintereinander schreiben. Wo hört einer auf und fängt der nächste an?

- **UTF-32**: Das einfachste, aber auch speicheraufwändigste, ist, sie alle auf gleiche Länge zu bringen. Jeder Codepoint wird in 4 Bytes = 32 Bit kodiert.

- Bei **UTF-16** wird ein Codepoint entweder mit 2 Bytes oder mit 4 Bytes dargestellt.
- Bei **UTF-8** wird ein Codepoint mit 1,2,3 oder 4 Bytes dargestellt.
- **UTF-8** ist das Format mit der höchsten Verbreitung. Es wird auch von Julia verwendet.

14.2.5 UTF-8

- Für jeden Codepoint werden 1, 2, 3 oder 4 volle Bytes verwendet.
- Bei einer Codierung mit variabler Länge muss man erkennen können, welche Bytefolgen zusammengehören:
  - Ein Byte der Form 0xxxxxxx steht für einen ASCII-Codepoint der Länge 1.
  - Ein Byte der Form 110xxxxx startet einen 2-Byte-Code.
  - Ein Byte der Form 1110xxxx startet einen 3-Byte-Code.
  - Ein Byte der Form 11110xxx startet einen 4-Byte-Code.
  - Alle weiteren Bytes eines 2-,3- oder 4-Byte-Codes haben die Form 10xxxxxx.
- Damit ist der Platz, der für den Codepoint zur Verfügung steht (Anzahl der x):
  - Ein-Byte-Code: 7 Bits
  - Zwei-Byte-Code: 5 + 6 = 11 Bits
  - Drei-Byte-Code: 4 + 6 + 6 = 16 Bits
  - Vier-Byte-Code: 3 + 6 + 6 + 6 = 21 Bits
- Damit ist jeder ASCII-Text automatisch auch ein korrekt codierter UTF-8-Text.
- Sollten die bisher für Unicode festgelegten 17 Ebenen = 21 Bit = 1.1 Mill. mögliche Zeichen mal erweitert werden, dann wird UTF-8 auf 5- und 6-Byte-Codes erweitert.

14.3 Zeichen und Zeichenketten in Julia

14.3.1 Zeichen: `Char`

Der Datentyp `Char` kodiert ein einzelnes Unicode-Zeichen.

- Julia verwendet dafür einfache Anführungszeichen: `'a'`.

- Ein `Char` belegt 4 Bytes Speicher und
- repräsentiert einen Unicode-Codepoint.
- `Chars` können von/zu `UInts` umgewandelt werden und
- der Integer-Wert ist gleich dem Unicode-codepoint.

14.3.2 Zeichenketten: `String`

- Für Strings verwendet Julia doppelte Anführungszeichen: `"a"`.
- Sie sind UTF-8-codiert, d.h., ein Zeichen kann zwischen 1 und 4 Bytes lang sein.

```
@show typeof('a') sizeof('a') typeof("a") sizeof("a");
```

```
typeof('a') = Char
sizeof('a') = 4
typeof("a") = String
sizeof("a") = 1
```

- `Char`s können von/zu `UInt`s umgewandelt werden.

```
UInt('a')
```

```
0x0000000000000061
```

```
b = Char(0x2656)
```

```
'♖': Unicode U+2656 (category So: Symbol, other)
```

Bei einem Nicht-ASCII-String unterscheiden sich Anzahl der Bytes und Anzahl der Zeichen:

```
asciistr = "Hello World!"
@show length(asciistr) ncodeunits(asciistr);
```

```
length(asciistr) = 12
ncodeunits(asciistr) = 12
```

(Das Leerzeichen zählt natürlich auch.)

```
str = "😄 Hellö 🎶"
@show length(str) ncodeunits(str);
```

```
length(str) = 9
ncodeunits(str) = 16
```

Iteration über einen String iteriert über die Zeichen:

```
for i in str
    println(i, "  ", typeof(i))
end
```

```
😄  Char
   Char
H  Char
e  Char
l  Char
l  Char
ö  Char
   Char
🎶  Char
```

14.3.3 Verkettung von Strings

“Strings mit Verkettung bilden ein nichtkommutatives Monoid.”

Deshalb wird in Julia die Verkettung multiplikativ geschrieben.

```
 str * asciistr * str
```

```
"😄 Hellö 🎶Hello World!😄 Hellö 🎶"
```

Damit sind auch Potenzen mit natürlichem Exponenten definiert.

```
str^3,  str^0
```

```
("😄 Hellö 🎶😄 Hellö 🎶😄 Hellö 🎶", "")
```

14.3.4 Stringinterpolation

Das Dollarzeichen hat in Strings eine Sonderfunktion, die wir schon oft in `print()`-Anweisungen genutzt haben. MAn kann damit eine Variable oder einen Ausdruck interpolieren:

```
a = 33.4
b = "x"

s = "Das Ergebnis für $b ist gleich $a und die verdoppelte Wurzel daraus ist $(2sqrt(a))\n"
```

```
"Das Ergebnis für x ist gleich 33.4 und die verdoppelte Wurzel daraus ist 11.55854662143991\n"
```

14.3.5 Backslash escape sequences

Der *backslash* \ hat in Stringkonstanten ebenfalls eine Sonderfunktion. Julia benutzt die von C und anderen Sprachen bekannten *backslash*-Codierungen für Sonderzeichen und für Dollarzeichen und Backslash selbst:

```
s = "So bekommt man \'Anführungszeichen\" und ein \$-Zeichen und einen\nZeilenumbruch und ein \\ usw... "
print(s)
```

```
So bekommt man 'Anführungszeichen" und ein $-Zeichen und einen
Zeilenumbruch und ein \ usw...
```

14.3.6 Triple-Quotes

Man kann Strings auch mit Triple-Quotes begrenzen. In dieser Form bleiben Zeilenumbrüche und Anführungszeichen erhalten:

```
s = """
 Das soll
ein "längerer"
  'Text' sein.
"""

print(s)
```

```
 Das soll
ein "längerer"
  'Text' sein.
```

14.3.7 Raw strings

In einem `raw string` sind alle backslash-Codierungen außer `\"` abgeschaltet:

```
s = raw"Ein $ und ein \ und zwei \\ und ein 'bla'..."
print(s)
```

```
Ein $ und ein \ und zwei \\ und ein 'bla'...
```

14.4 Weitere Funktionen für Zeichen und Strings (Auswahl)

14.4.1 Tests für Zeichen

```
@show isdigit('0') isletter('Ψ') isascii('\U2655') islowercase('α')
@show isnumeric('½') iscntrl('\n') ispunct(';');
```

```
isdigit('0') = true
isletter('Ψ') = true
isascii('♕') = false
islowercase('α') = true
isnumeric('½') = true
iscntrl('\n') = true
ispunct(';') = true
```

14.4.2 Anwendung auf Strings

Diese Tests lassen sich z.B. mit `all()`, `any()` oder `count()` auf Strings anwenden:

```
all(ispunct, ";.:")
```

```
true
```

```
any(isdigit, "Es ist 3 Uhr! 🕒" )
```

```
true
```

```
count(islowercase, "Hello, du!!")
```

```
6
```

14.4.3 Weitere String-Funktionen

```
@show startswith("Lampenschirm", "Lamp")  occursin("pensch", "Lampenschirm")
@show endswith("Lampenschirm", "irm");
```

```
startswith("Lampenschirm", "Lamp") = true
occursin("pensch", "Lampenschirm") = true
endswith("Lampenschirm", "irm") = true
```

```
@show uppercase("Eis") lowercase("Eis")  titlecase("eiSen");
```

```
uppercase("Eis") = "EIS"
lowercase("Eis") = "eis"
titlecase("eiSen") = "Eisen"
```

```
# remove newline from end of string

@show chomp("Eis\n")  chomp("Eis");
```

```
chomp("Eis\n") = "Eis"
chomp("Eis") = "Eis"
```

```
split("π ist irrational.")
```

```
3-element Vector{SubString{String}}:
 "π"
 "ist"
 "irrational."
```

```
replace("π ist irrational.", "ist" => "ist angeblich")
```

```
"π ist angeblich irrational."
```

14.5 Indizierung von Strings

Strings sind nicht mutierbar aber indizierbar. Dabei gibt es ein paar Besonderheiten.

- Der Index nummeriert die Bytes des Strings.
- Bei einem nicht-ASCII-String sind nicht alle Indizes gültig, denn
- ein gültiger Index adressiert immer ein Unicode-Zeichen.

Unser Beispielstring:

```
str
```

```
"😄 Hellö 🎶"
```

Das erste Zeichen

```
str[1]
```

```
'😄': Unicode U+1F604 (category So: Symbol, other)
```

Dieses Zeichen ist in UTF8-Kodierung 4 Bytes lang. Damit sind 2,3 und 4 ungültige Indizes.

```
str[2]
```

```
LoadError: StringIndexError: invalid index [2], valid nearby indices [1]=>'😄', [5]=>' '

Stacktrace:
 [1] string_index_err(s::String, i::Int64)
   @ Base ./strings/string.jl:12
 [2] getindex_continued(s::String, i::Int64, u::UInt32)
   @ Base ./strings/string.jl:440
 [3] getindex(s::String, i::Int64)
   @ Base ./strings/string.jl:433
 [4] top-level scope
   @ In[35]:1
```

Erst das 5. Byte ist ein neues Zeichen:

```
str[5]
```

```
' ': ASCII/Unicode U+0020 (category Zs: Separator, space)
```

Auch bei der Adressierung von Substrings müssen Anfang und Ende jeweils gültige Indizes sein, d.h., der Endindex muss ebenfalls das erste Byte eines Zeichens indizieren und dieses Zeichen ist das letzte des Teilstrings.

```
str[1:7]
```

```
"😄 He"
```

Die Funktion `eachindex()` liefert einen Iterator über die gültigen Indizes:

```
for i in eachindex(str)
    c = str[i]
    println("$i: $c")
end
```

```
1: 😄
5:
6: H
7: e
8: l
9: l
10: ö
12:
13: 🎶
```

Wie üblich macht collect() aus einem Iterator einen Vektor.

```
collect(eachindex(str))
```

```
9-element Vector{Int64}:
  1
  5
  6
  7
  8
  9
 10
 12
 13
```

Die Funktion `nextind()` liefert den nächsten gültigen Index.

```
@show nextind(str, 1) nextind(str, 2);
```

```
nextind(str, 1) = 5
nextind(str, 2) = 5
```

Warum verwendet Julia einen Byte-Index und keinen Zeichenindex? Der Hauptgrund dürfte die Effizienz der Indizierung sein.

- In einem langen String, z.B. einem Buchtext, ist die Stelle `s[123455]` mit einem Byte-Index schnell zu finden.
- Ein Zeichen-Index müsste in der UTF-8-Codierung den ganzen String durchlaufen, um das n-te Zeichen zu finden, da die Zeichen 1,2,3 oder 4 Bytes lang sein können.

Einige Funktionen liefern Indizes oder Ranges als Resultat. Sie liefern immer gültige Indizes:

```
findfirst('l', str)
```

```
8
```

```
findfirst("Hel", str)
```

```
6:8
```

```
str2 = "αβγδε"^3
```

```
"αβγδεαβγδεαβγδε"
```

```
n = findfirst('γ', str2)
```

```
5
```

So kann man ab dem nächsten nach `n=5` gültigen Index weitersuchen:

```
findnext('γ', str2, nextind(str2, n))
```

```
15
```