Programmieren im Bauingenieurwesen

Einführung und Grundlagen

Autoren
Sebastian Pech, Raphael Suda
Datum
7. November 2019
Erstellt mit Julia
1.2.0
Version
7a9b66e

Inhaltsverzeichnis

1 Einführung und Vorwort

Diese Lehrunterlagen dienen der Unterstützung bei der Lehrveranstaltung (LVA) Programmieren im Bauingenieurwesen. Alle Inhalte, die in der LVA besprochen werden, sind chronologisch geordnet in diesem Dokument gelistet und durch nützliche Informationen und Beispiele ergänzt. Einige Abschnitte schließen mit einer Aufgabensammlung, welche zum Erlangen einer positiven Benotung der LVA gelöst und in den TUWEL-Kurs hochgeladen werden muss.

Ziel der LVA ist es, einen ersten, aber umfassenden Einblick in die Welt der Programmiersprachen zu geben. Inhalte, die über die allgemeinen Programmiergrundlagen hinausgehen, sind dabei thematisch auf das Bauingenieurwesen Studium zugeschnitten.

1.1 Ziele

Wir haben selbst erlebt, dass in einigen Master LVAs und besonders bei der Diplomarbeit Programmierkenntnisse gefordert werden. Nach dem Besuch der LVA sollten daher Aufgaben wie Versuchsauswertungen, Adaption von bestehenden Programmen oder Erstellen eigener Programme unter Anleitung gelöst werden können.

Die nötigen Inhalte, um selbständig Programme zu konzeptionieren und programmieren, werden in der LVA nicht abgedeckt. Dieses Thema ist komplex und erfordert bereits etwas Erfahrung. Für Interessierte sind daher in einigen Abschnitten Verweise auf weiterführende Informationen angegeben.

1.2 Programmiersprache

Die Welt der Programmiersprachen ist vielfältig und umfangreich. Es gibt eine Vielzahl von Anwendungsgebieten, für die spezifische Sprachen entwickelt wurden. Da viele Aufgaben von Bauingenieuren naturwissenschaftlich und mathematisch fundiert sind, liegt es nahe, in dieser LVA eine der sogenannten wissenschaftlichen Sprachen zu verwenden.

Der Begriff bezeichnet Programmiersprachen, die standardmäßig mathematische Objekte wie Vektoren und Matrizen unterstützen oder z.B. über Statistik-Bibliotheken verfügen. Einige Vertreter dieser Klasse sind MATLAB, R, Python mit SciPy und Julia.

Jede dieser Sprachen hat ihre Vor- und Nachteile. MATLAB wird besonders häufig im universitären Bereich genutzt und bietet ein benutzerfreundliches Interface. R ist besonders auf statistische Auswertungen zugeschnitten. Python ist von dieser Gruppe die allgemeinste Sprache und ist eine der beliebtesten Programmiersprachen überhaupt. Julia ist eine der jüngeren Programmiersprachen, zeichnet sich aber durch ein sehr gutes Sprachdesign und hohe Performance aus.

In den Anfängen dieser LVA wurde MATLAB gelehrt, was hauptsächlich der starken Verbreitung an Universitäten geschuldet war. Ein Nachteil von MATLAB ist: Eine nicht-universitäre Lizenz ist sehr teuer. Unser Ziel ist es, dass alle Studierenden, auch nach ihrem Studium, Zugang zu der Software haben und ihren Arbeitsalltag mit Programmierkenntnissen vereinfachen können.

Python und R sind frei verfügbar und weit verbreitet. Ein Nachteil im Zusammenhang mit der LVA ist aber, dass sich diese Programmiersprachen relativ stark vom MATLAB unterscheiden. Da MATLAB in anderen LVAs verwendet wird und wir nicht wollen, dass Programmieranfänger im Laufe des Studiums mit zwei sehr unterschiedlichen Sprachen konfrontiert werden, sind Python und R keine Optionen für uns.

Julia ist ebenfalls frei verfügbar und hat eine zu MATLAB sehr ähnliche Syntax. Dies und das stringente Sprachdesign sind für uns Grund genug, Julia in dieser LVA zu verwenden.

Die syntaktischen Unterschiede von MATLAB, Python und Julia sind in diesem Cheatsheet[1] grob zusammengefasst. Sollte es notwendig sein, ein Programm nicht in Julia zu schreiben, empfehlen wir, das Cheatsheet als erste Hilfe zur Hand zu nehmen.

1.3 Lehrunterlagen

Die Unterlagen wurden an einigen Stellen um nützliche Verweise auf weiterführende Informationen oder Quellen erweitert. Grundsätzlich werden zwei Arten von Links unterschieden: Auf Einträge in der Julia-Dokumentation

[1] https://cheatsheets.quantecon.org

unter https://docs.julialang.org wird im Text immer durch Angabe der Abschnittsnamen verwiesen. Links, die nicht auf die Dokumentation verweisen, werden immer mit einer zusätzlichen Fußnote angeben. Beispielsweise sind weitere Informationen zu Strings in der Dokumentation unter Manual/Strings oder auf der Website Think Julia[2] zu finden

Damit können die Links sowohl in der pdf-Version als auch in der Printversion geöffnet werden.

1.4 Kontrollprogramm

Zusätzlich zu den Unterlagen gibt es ein Kontrollprogramm, mit dem die Aufgaben zu den einzelnen Abschnitten kontrolliert werden können. Das Programm wird in Form eines eigenständigen Julia-Pakets verwendet und muss mit dem Paketmanager durch Eingabe von

```
add https://github.com/sebastianpech/JuliaSkriptumKontrolle.jl#JuliaSkriptum
```

installiert werden (Näheres dazu im Abschnitt Interaktion). Das Laden des Paketes erfolgt nach der Installation mit

```
using JuliaSkriptumKontrolle
```

Bei den Aufgaben geht es meistens darum, kurze, in sich abgeschlossene Codes oder Funktionen zu schreiben. Eine einzelne Funktion kann z.B. wie folgt kontrolliert werden:

```
@Aufgabe "1.2.3" function meine_funktion(x,y)
    # Hier den Code einfügen
end
```

Dabei startet die Angabe `@Aufgabe "1.2.3"` die Kontrollmechanismen für die Aufgabe mit der Nummer 1.2.3. Die Nummer entspricht immer der Überschriftennummerierung der entsprechenden Aufgabe.

Wird bei einer Aufgabe gefordert, mehrere Funktionen oder zusätzliche externe Pakete zu verwenden, kann die allgemeine Version des vorherigen Codes verwendet werden:

```
@Aufgabe "2.1.3" begin
    using EinExtraPaket # Ein Paket laden

    function meine_funktion(x)
        # Hier den Code einfügen
    end

    # Hier den anderen Code einfügen
end
```

Sind spezielle Angaben notwendig, ist das bei der jeweiligen Aufgabe vermerkt.

Ist eine Aufgabe nicht vollständig korrekt gelöst, wird eine Fehlermeldung ausgegeben. Bei der folgenden Aufgabe soll eine Funktion definiert werden, die den Winkel zwischen zwei Vektoren beliebiger Dimension berechnet.

```
@Aufgabe "10.3.4" function winkel(v1,v2)
    v1-v2 # Offensichtlich falsch
end
```

```
Error: AssertionError: Die Funktion sollte einen Winkel zurück geben.
```

Der aktuelle Status aller Aufgaben (Noch nicht behandelt, Falsch, Richtig) kann mit der folgenden Funktion angezeigt werden:

```
JuliaSkriptumKontrolle.status()
```

```
---------------------------------
|  Aufgabe | Status |     Punkte |
|----------+--------+------------|
|    3.5.1 |      ? |        1.0 |
|    7.4.1 |      ? |        1.0 |
|    7.4.2 |      ? |        1.0 |
|    7.4.3 |      ? |        1.0 |
|    7.4.4 |      ? |        1.0 |
```

[2] https://benlauwens.github.io/ThinkJulia.jl/latest/book.html#chap08

```
|    7.4.5 |      ? |        1.0 |
|    8.5.1 |      ? |        1.0 |
|    8.5.2 |      ? |        1.0 |
|    8.5.3 |      ? |        1.0 |
|    8.5.4 |      ? |        1.0 |
|    8.5.5 |      ? |        1.0 |
|    8.5.6 |      ? |        1.0 |
|   10.3.1 |      ? |        2.0 |
|   10.3.2 |      ? |        1.0 |
|   10.3.3 |      ? |        1.0 |
|   10.3.4 |      × |        1.0 |
|   11.3.1 |      ? |        1.0 |
|   11.3.2 |      ? |        3.0 |
|   11.3.3 |      ? |        3.0 |
|   11.3.4 |      ? |        3.0 |
| 11.3.5.1 |      ? |        1.5 |
| 11.3.5.2 |      ? |        1.5 |
|   11.3.6 |      ? |        4.0 |
|   11.3.7 |      ? |        2.0 |
|   11.3.8 |      ? |        1.0 |
|   12.3.1 |      ? |        1.0 |
|   13.3.1 |      ? |        2.0 |
|   14.3.1 |      ? |        5.0 |
|   14.3.2 |      ? |        3.0 |
|   14.3.3 |      ? |        3.0 |
|----------+--------+------------|
|          |      ∑ | 0.0 / 51.0 |
-----------------------------------
```

Die Ausgabe enthält auch die Anzahl an erreichten und möglichen Punkten. Da Aufgabe `10.3.4` in einem der vorherigen Beispiele falsch gelöst wurde, steht anstelle von ”?”, ”×”.

Die Abgabe der Aufgaben erfolgt in TUWEL. Es sollen alle Aufgaben gesammelt in einer Julia-Datei (Endung `.jl`) abgegeben werden. Dabei müssen die kontrollierten Aufgaben nacheinander in einer Datei (z.B. `loesung_aufgaben.jl`) angegeben werden:

```julia
@Aufgabe ”1.2.3” function meine_funktion(x,y)
    # Hier den Code einfügen
end

@Aufgabe ”2.1.3” begin
    using EinExtraPaket # Ein Paket laden

    function meine_funktion(x)
        # Hier den Code einfügen
    end

    # Hier den anderen Code einfügen
end

@Aufgabe ”10.3.4” function winkel(v1,v2)
    v1-v2 # Offensichtlich falsch
end

JuliaSkriptumKontrolle.status()
```

Wird die obige Datei mit `julia meine_loesungen.jl` ausgeführt (Näheres dazu im Abschnitt Interaktion), wird, wenn alle Aufgaben richtig gelöst sind, kein Fehler ausgegeben.

2 Installation

In diesem Abschnitt werden einige Begriffe, wie z.B. *Terminal* oder *REPL*, verwendet, die Erklärung dieser folgt im Abschnitt Interaktion.

2.1 Julia

Julia ist für Windows, macOS und Linux kostenlos auf der Webseite julialang.org verfügbar. Unter Downloads[3] befinden sich die für die Installation notwendigen Dateien.

2.1.1 Windows

Julia ist für Windows 7 und alle spätere Versionen in 32 bit und 64 bit verfügbar. Folgende Schritte sind für die Installation erforderlich:

1. Die richtige Version (32 bit oder 64 bit) der Installationsdatei `julia.exe` herunterladen.
2. Die heruntergeladene Datei ausführen, um Julia zu entpacken.

Im entpackten Ordner befindet sich die Datei `julia`. Diese kann entweder durch Doppelklick oder über die Commandline durch Eingabe des Befehls `julia` ausgeführt werden. Sollte bei Aufruf in der Commandline ein Fehler „`Der Befehl "julia" ist entweder falsch geschrieben oder konnte nicht gefunden werden`“ auftreten, wurde der Installationspfad wahrscheinlich bei der Installation nicht zur Systemvariable `PATH` hinzugefügt. Windows kann die Datei `julia` dann nicht finden. Unter diesem Link[4] befindet sich eine kurze Erklärung zur Einrichtung der Systemvariable `PATH`. Der Punkt „Geben Sie ... den Wert der Umgebungvariable PATH an.“ in dieser Erklärung ist dabei folgendermaßen zu verstehen: Unter Path sind alle Pfade gespeichert, die bei der Eingabe eines Befehls in der Commandline nach übereinstimmenden Befehlen durchsucht werden. Die Pfade sind dabei durch Semikola `;` voneinander getrennt. Unter diesem Punkt der Anleitung ist daher der Pfad der Julia-Installation am Ende der Variable hinzuzufügen:

```
# "Alte" Systemvariable PATH
Pfad01;Pfad02;Pfad03

# "Neue" Systemvariable PATH
Pfad01;Pfad02;Pfad03;Julia-Pfad
```

2.1.2 Linux

Zur Installation von Julia auf einem Linux-Betriebssystem müssen zuerst die „Generic Binaries“ für die richtige Prozessor-Architektur heruntergeladen werden. Die Datei im `.tar.gz`-Format muss danach in einen beliebigen Ordner auf dem Computer entpackt werden. In diesem Ordner befindet sich unter `bin/` die Datei `julia`. Um Julia im Terminal ausführen zu können, muss gewährleistet sein, dass das System diese Datei findet. Am einfachsten ist es, einen „Symbolic Link“ in einem der Ordner unter der Systemvariable `PATH` zu erzeugen. Welche Ordner in `PATH` hinterlegt sind, kann im Terminal mittels `echo $PATH` abgefragt werden. Meist ist `/usr/local/bin` einer davon. Folgender Befehl erstellt den Symbolic Link in diesem Ordner:

```
sudo ln -s <Pfad zum entpackten Ordner>/bin/julia /usr/local/bin/julia
```

Danach kann Julia durch Eingabe von `julia` im Terminal gestartet werden.

2.1.3 macOS

Julia ist für macOS 10.8 und spätere Versionen verfügbar. Für Mac-User befindet sich auf julialang.org unter Downloads[5] eine `.dmg`-Datei zum Download. Dieses Disk Image muss geöffnet und die darin enthaltene Datei `Julia-<version>.app` in den gewünschten Dateipfad (vozugsweise Programme) kopiert werden. Julia kann dann direkt über diese Datei ausgeführt werden. Um Julia im Terminal auszuführen, muss die Ausführbare Datei vom System gefunden werden können. Dafür kann analog zu der Anleitung für Linux vorgegangen werden. Der Befehl zum Erstellen des Symbolic Link muss auf die Binary in `Julia-<verison>.app` verweisen:

```
sudo ln -s /Applications/Julia-<version>.app/Contents/Resources/julia/bin/julia /usr/local/bin/julia
```

[3] https://julialang.org/downloads/
[4] https://www.java.com/de/download/help/path.xml
[5] https://julialang.org/downloads/

2.2 Juno

Generell reicht es aus, Julia zu installieren und über die Command Line zu benutzen. Entwicklungsumgebungen, wie z.B. *Juno*, bieten allerdings viele Features, die (nicht nur) den Einstieg ins Programmieren erleichtern. Dazu muss zuerst der erweiterbare Texteditor *Atom* installiert werden. Die Installationsdateien für Windows, macOS und Linux sind kostenlos auf der Website atom.io[6] erhältlich.

Nach der Installation von Atom muss *Juno* in Atom installiert werden. Dazu ist es erforderlich, Atom zu öffnen und in die Einstellungen zu wechseln (`Strg+,` oder `Cmd+,` bzw. über die Menüleiste) und dort das Installations-Panel auszuwählen. Durch Installation des Pakets `uber-juno` (am einfachsten über die Suchleiste) kann Juno in Atom verwendet werden. Falls für Julia nicht der Standard-Installationspfad ausgewählt wurde, muss unter Packages > Julia > Settings im Feld „Julia Path“ der tatsächliche Dateipfad angegeben werden.

Wenn die Installation funktioniert hat, kann unter Packages > Juno > Open Console die REPL geöffnet und durch Drücken der Taste `Enter` eine Julia-Session gestartet werden.

3 Interaktion

Um Julia auszuführen, gibt es verschieden Möglichkeiten:

- Interaktiv in Form der REPL (Read-eval-print loop) über das Programm-Icon oder Eingabe von `julia` in der Commandline.
- Als Skript durch Eingabe von `julia script.jl [arg1, arg2 ...]` in der Commandline (Die eckigen Klammern kennzeichnen optionale Argumente),
- In einem Editor, der Julia unterstützt (z.B. Atom mit Juno),
- Als sogenanntes Notebook mit Jupyter.

Jupyter-Notebooks eignen sich, um Berechnungen zu dokumentieren und z.B. um Grafiken und Formeln zu erweitern; dies sprengt aber den Rahmen der LVA. Für Interessierte ist hier auf das Paket `IJulia.jl`[7] verwiesen, welches Julia für Jupyter-Notebooks aktiviert.

3.1 Die REPL

Die REPL bietet die rudimentärste Methode des Zugriffs auf Julia. Die Funktionalität, wie es auch im Namen REPL steckt, besteht darin:

1. **R**ead: Einen Befehl einzulesen
2. **E**val: Den eingelesenen Befehl auszuwerten
3. **P**rint: Das Ergebnis auszugeben
4. **L**oop: Zurück zu 1. und auf den nächsten Befehl zu warten

Die REPL kann durch Starten von Julia über das Programm-Icon, den Befehl `julia` oder direkt in Juno geöffnet werden (Ausführen der Funktion `exit()` oder Drücken von `Strg-D` schließt die REPL wieder).

Je nach Betriebssystem öffnet sich dann ein Fenster das dem in der folgenden Abbildung ähnelt.

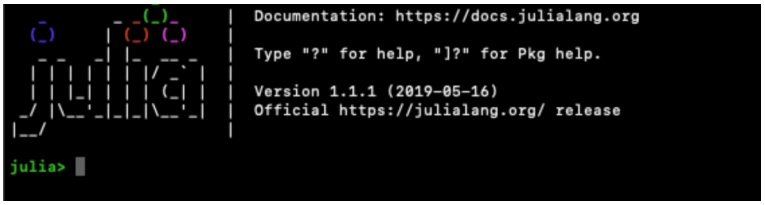


Abbildung 1: REPL-Fenster direkt nach dem Start

[6] https://atom.io
[7] https://github.com/JuliaLang/IJulia.jl

Nach dem Start steht der Cursor nach dem Text `julia>`, welcher Prompt genannt wird. Die REPL befindet sich jetzt im „Julian mode“. Das bedeutet, jetzt können Julia-Befehle eingegeben und ausgewertet werden.

Die REPL kennt noch drei weitere Modi, die durch Eingabe eines speziellen Zeichens direkt nach der Prompt gestartet werden können:

1. „Help mode“ (`help?>`) durch `?`: Erlaubt die Anzeige der Hilfe für bestimmte Ausdrücke.
2. Paketmanager (`(v1.2) pkg>`) durch `]`: Startet den Paketmanager, mit dem zusätzliche Pakete installiert werden können.
3. „Shell mode“ (`shell>`) durch `;`: Ermöglicht das Ausführen von Befehlen im Betriebssystem.

Der „Julian mode“ ist, wenn der Cursor direkt hinter der Prompt steht, immer durch Dücken der Backspace-Taste erreichbar.

Die REPL biete eine sogenannte Autovervollständigung für Julia-Befehle. Wird nur ein Teil eines Befehls eingegeben, kann dieser durch Drücken von `TAB` vervollständigt werden. Das ist sehr praktisch, da man sich als BenutzerIn nicht den exakten Wortlaut jedes Befehls merken muss.

3.2 Ausführung als Skript

Eine Julia-Datei oder ein Julia-Skript (Endung `.jl`) kann über die Commandline ausgeführt werden. Je nach Betriebsystem ist diese über einen anderen Weg erreichbar:

- Windows: Die Commandline verbirgt sich hinter dem Programm cmd.exe. Dieses ist am einfachsten durch die Tastenkombination `windows+r` (Run) und Eingabe von „cmd“, im erschienenen Fenster, gefolgt von `Enter` erreichbar.
- macOS: In macOS wird die Commandline Terminal genannt. Über die Spotlight-Suche (`cmd+space`) und Eingabe von „Terminal“ gelangt man zu der richtigen App.
- Linux: Linux-User sollten wissen, wie sie zur Commandline gelangen.

Die Commandline ist wie die Julia-REPL aufgebaut und funktioniert nach dem selben Prinzip. Die wichtigsten Befehle für die Navigation sind:

- `pwd`: Gibt das aktuelle Verzeichnis aus (auch „Working Directory“ genannt).
- `cd pfad`: Wechselt in das Verzeichnis `pfad`.
- `ls`: Listet den Inhalt des aktuellen Verzeichnisses auf.

Diese Befehle genügen zum Lösen der Aufgaben, je nach Betriebssystem sind aber noch sehr viele weitere verfügbar.

Nach der Installation von Julia ist der Befehl `julia` verfügbar. Soll ein Skript `mein-script.jl` ausgeführt werden, ist das durch Eingabe von `julia mein-script.jl` in der Commandline möglich. Im Aufruf von Julia gibt `mein-script.jl` dabei der Pfad zu der auszuführenden Datei an. Bei dieser Angabe handelt es sich um einen sogenannten relativen Pfad, relativ bezogen auf das aktuelle Verzeichnis. Befindet sich die Datei in einem Unterordner `scripts` würde der Befehl

- unter Windows: `julia scripts\mein-script.jl` und
- unter macOS und Linux: `julia scripts/mein-script.jl`

lauten. Das Gegenstück zu relativen Pfaden sind absolute Pfade. Ein absoluter Pfad bezieht sich immer auf das Laufwerk z.B. `C:\` oder `/`.

Generell empfehlen wir, mit relativen Pfaden zu arbeiten, da das Skript dann auf jedem Gerät ohne Anpassung der Pfade ausgeführt werden kann. Im folgenden Beispiel wird eine Datei `meinedatei.txt` mit einem absoluten und mit einem relativen Pfad geöffnet.

```
file = open("/Users/someuser/meinedatei.txt") # Absoluter Pfad unter Linux
file = open("C:\\Benuter\\meinedatei.txt") # Absoluter Pfad unter Windows
file = open("meinedatei.txt") # Relativer Pfad
```

Die absoluten Pfade funktioniert nur dann, wenn auf jedem Gerät, auf dem der Befehl ausgeführt wird, auch der Ordner mit dem exakt gleichen Pfad vorhanden ist. Liegt `meinedatei.txt` allerdings immer im Verzeichnis,

in dem das Skript gestartet wird, kann die Datei beim relativen Pfad unabhängig vom Gerät gefunden werden. Im Windows-Pfad wird als Trennzeichen `\\` verwendet, obwohl der Windows-Pfad eigentlich nur einen einzelnen `\` enthält. Das ist erforderlich, da `\` einen sogenannten „Special Character“ einleitet. Nähere Informationen zu diesem Thema werden im Abschnitt Strings gegeben.

3.3 Juno Spezifika

Juno enthält eine integrierte REPL, die über Packages > Juno > Show Console geöffnet werden kann. Diese REPL unterstützt den vollen Funktionsumfang der Julia REPL.

Neben der REPL sind Befehle auch direkt aus der Datei, die gerade editiert wird, heraus ausführbar. Durch Drücken von `Strg+Enter` unter Windows und Linux bzw. `Shift+Enter` unter macOS kann die aktuelle Zeile an die REPL geschickt werden. Damit können schnell auch größere Codeblöcke ausgeführt werden, da die Funktion automatisch syntaktisch sinnvolle Zeilen aus der Umgebung gruppiert.

```
function meine_funktion()
    return 10
end

x = 10
y = 12
```

Wird z.B. im oberen Skript `Strg+Enter` bzw. `Shift+Enter` in der Zeile `return 10` gedrückt, werden auch die Zeile davor und danach ausgewertet. Im Gegensatz dazu wird beim Senden der Zeile `x = 10` nur diese Zeile ausgeführt.

Auf der Website von Juno[8] sind weitere Funktionen in einer interaktiven Grafik erklärt.

3.4 Spezielle REPL-Modes

3.4.1 Hilfe

Julia-Quellcode ist selbst-dokumentiert. Das bedeutet, dass die Beschreibung und Hilfe für Funktionen direkt in den Code geschrieben wird. Beim Programmieren muss dabei eine Richtlinie eingehalten werden, damit die Hilfe für eine Funktion vom Benutzer auch aufgerufen werden kann (siehe Docstrings im Abschnitt Funktionen).

Wie schon im Abschnitt zur Interaktion beschrieben, gibt es einen eigenen REPL-Mode, der die Hilfe für Funktionen anzeigt. Dieser Modus kann durch Eingabe von `?` direkt nach `julia>` aktiviert werden. Die Prompt sollte sich dann in `help?>` ändern. Jetzt kann jede beliebige Funktionen (z.B. `abs` für den Absolutwert einer Zahl) eingegeben werden. Durch Bestätigen mit Enter erscheint die folgende Ausgabe:

```
help?> abs
search: abs abs2 abspath AbstractSet abstract type AbstractChar AbstractDict AbstractFloat AbstractArray AbstractRange
AbstractString AbstractVector

  abs(x)

  The absolute value of x.

  When abs is applied to signed integers, overflow may occur, resulting in the return of a negative value. This overflow
occurs only when abs is applied to
  the minimum representable value of a signed integer. That is, when x == typemin(typeof(x)), abs(x) == x < 0, not -x as
might be expected.

  Examples
  ==========

  julia> abs(-3)
  3

  julia> abs(1 + im)
  1.4142135623730951

  julia> abs(typemin(Int64))
  -9223372036854775808
```

[8] https://junolab.org

Die Hilfe zu den Funktionen ist auch unter docs.julialang.org erreichbar. Dort wurde die Dokumentation gesammelt und um nützliche Tipps und Erklärungen erweitert.

Falls Probleme oder Unklarheiten bei der Verwendung von Julia auftreten, gibt es verschiedene Kanäle, über die Fragen gestellt werden können:

- Stack Overflow[9]
- Discourse[10]
- Github[11]

Github ist die zentrale Plattform, auf der Julia-Pakete (und auch Julia selbst) verwaltet werden. Findet man einen Fehler, kann man dort eine Nachricht schreiben und direkt mit EntwicklerInnen Kontakt aufnehmen.

3.4.2 Paketmanager

Julia zeichnet sich durch eine sehr aktive Community aus, die Basiskomponenten der Sprache laufend um neue Funktionen erweitert. Solche Erweiterungen werden durch Pakete zu Julia hinzugefügt.

Der Paketmanager kann aus der REPL durch Eingabe von `]` direkt nach `julia>` aktiviert werden. Die Prompt ändert sich dann in `(v1.2) pkg>`. Der Paketmanager ist ein sehr mächtiges Tool, dessen Funktionsumfang unter Standard Library/Pkg dokumentiert ist. Im Zuge der LVA beschränken wir uns auf das Installieren, Aktualisieren und Entfernen von Paketen.

Ein Paket, z.B. `Example.jl`, kann durch `add` installiert werden:

```
(v1.2) pkg> add Example
  Updating registry at `~/.julia/registries/General`
  Updating git-repo `https://github.com/JuliaRegistries/General.git`
 Resolving package versions...
 Installed Example - v0.5.3
  Updating `~/.julia/environments/v1.2/Project.toml`
  [7876af07] + Example v0.5.3
  Updating `~/.julia/environments/v1.2/Manifest.toml`
  [7876af07] + Example v0.5.3
```

Gibt es neue Versionen von Paketen, können diese gesammelt mit `update` aktualisiert werden:

```
(v1.2) pkg> update
  Updating registry at `~/.julia/registries/General`
  Updating git-repo `https://github.com/JuliaRegistries/General.git`
  Updating git-repo `https://github.com/JunoLab/Atom.jl.git`
 Resolving package versions...
  Updating `~/.julia/environments/v1.2/Project.toml`
 [no changes]
  Updating `~/.julia/environments/v1.2/Manifest.toml`
 [no changes]
```

Zum Entfernen eines Pakets kann der Befehl `rm` verwendet werden:

```
(v1.2) pkg> rm Example
  Updating `~/.julia/environments/v1.2/Project.toml`
  [7876af07] - Example v0.5.3
  Updating `~/.julia/environments/v1.2/Manifest.toml`
  [7876af07] - Example v0.5.3
```

3.5 Aufgaben

In der LVA werden ein paar ausgewählte Pakete zur Erweiterung von Julia verwendet. Die erste Aufgabe ist die Installation dieser Pakete auf dem eigenen Computer.

3.5.1 Installation der für die LVA benötigten Pakete

Die folgenden Pakete sollen mit dem Paketmanager installiert werden.

[9] https://stackoverflow.com/questions/tagged/julia
[10] https://discourse.julialang.org
[11] https://github.com

- `CSV.jl`
- `DataFrames.jl`
- `Plots.jl`
- `Unitful.jl`
- `Measurements.jl`
- `Debugger.jl`
- `RDatasets.jl`
- `Distributions.jl`

Die Aufgabe kann durch Kontrolle eines leeren Codeblocks überprüft werden:

```
@Aufgabe "3.5.1" begin
end
```

4 Kommentare

In umfangreichen Programmen sammeln sich schnell hunderte Zeilen an Code an. Je länger ein Programm ist, desto schwieriger ist es, dessen Aufgabe aus den Codezeilen herauszulesen. Deshalb ist es sinnvoll, ein Programm mit Notizen zu ergänzen, die den Sinn und Zweck hinter Teilen des Codes erklären. Diese Notizen heißen *Kommentare*, beginnen mit einem #-Symbol und enden im Normalfall mit dem Zeilenende:

```
# Berechne die Fläche eines Kreises aus dessen Durchmesser
A = ( d^2 * π ) / 4
```

Da der Kommentar erst mit dem #-Symbol beginnt, könnte er auch am Ende der Zeile stehen:

```
A = ( d^2 * π ) / 4  # Fläche eines Kreises
```

Alle Zeichen ab dem # bis zum Zeilenende werden ignoriert und haben keine Auswirkung auf die Programmausführung.

Am sinnvollsten sind Kommentare, die nicht offensichtliche Eigenschaften des Programms erklären. Dieser Kommentar ist beispielsweise redundant und nicht sinnvoll:

```
d = 10  # weise d den Wert 10 zu
```

Der nachfolgende Kommentar enthält dagegen nicht offensichtliche Informationen und ist deswegen umso hilfreicher:

```
d = 10  # Durchmesser in Millimeter
```

Gut gewählte Variablennamen bieten ebenfalls die Möglichkeit, den Code verständlicher zu gestalten.

Für besonders lange Notizen gibt es in Julia mehrzeilige Kommentare. Mehrzeilige Kommentare können zwischen #= und =# geschrieben werden:

```
#= Weil dieser Kommentar besonders lang ist,
reicht eine Zeile nicht aus. Daher muss er
als mehrzeiliger Kommentar markiert werden. =#
```

5 Variablen

Variablen ermöglichen es, einer Zeichenfolge einen Wert zuzuweisen und diesen Wert über diese Zeichenfolge wieder abzurufen bzw. manchmal auch zu verändern. Die Zuweisung (*assignment*) erfolgt durch =. Dabei wird eine neue Variable erstellt und der eingegebenen Wert zugewiesen:

```
julia> message = "Das ist ein String"
"Das ist ein String"

julia> n = 19
19
```

```
julia> α = 17.32
17.32
```

Die erste Zeile weist der Variable `message` ein `String` zu, die zweite Zeile übergibt ein `Integer` an die Variable `n` und die dritte Zeile weist `α` eine Zahl vom Typ `Float` zu. Einige relevante Datentypen werden später im Abschnitt Datentypen genauer beschrieben.

Nach der Zuweisung von Variablen können diese im Code verwendet werden:

```
julia> a = 1; b = 2;

julia> summe = a + b
3
```

Im ersten Ausdruck dieses Beispiels werden den Variablen `a` und `b` ganzzahlige Werte zugewiesen. Die beiden Semikola (`;`) haben dabei unterschiedliche Funktionen. Das erste Semikolon trennt die beiden Zuweisungen voneinander. Auf diese Weise können beide Zuweisungen in einer Zeile erfolgen. Das abschließende Semikolon unterbindet die Ausgabe des letzten Wertes in der REPL. Ohne dieses Semikolon würde nach Eingabe des ersten Ausdruckes der Wert `2` in der REPL ausgegeben werden. Um die Lehrunterlagen etwas kompakter zu halten, wird in vielen Beispiel diese Schreibweise verwendet.

Generell sollten aussagekräftige Namen für Variablen gewählt werden. Meist ist dadurch bereits ohne Kommentare ersichtlich, wofür die Variable verwendet wird.

Variablennamen können beliebig lange sein und nahezu alle Unicode-Zeichen enthalten, dürfen allerdings nicht mit Zahlen beginnen. Großbuchstaben können in Variablennamen verwendet werden, meist kommen aber nur Kleinbuchstaben zum Einsatz. Unicode-Zeichen, wie z.B. `α`, können in der Julia REPL durch Eingeben von Abkürzungen ähnlich zu LaTeX und Drücken der `TAB`-Taste erstellt werden (z.B. `\alpha TAB`). Bei aus mehreren Wörten zusammengesetzten Variablennamen bietet es sich an, die Wörter durch einen Unterstrich, `_`, zu trennen (z.B. `sum_of_squares`).

Wird ein ungültiger Name für eine Variable verwendet, tritt ein *Syntax Error* auf:

```
julia> 42number = "meaning of life"
Error: syntax: "42" is not a valid function argument name
```

Zusätzlich tritt ein *Syntax Error* auf, wenn festgelegte *keywords* (z.B. `for` oder `end`) als Name festgelegt werden. Die Julia-Dokumentation enthält unter Manual/Variables noch nähere Informationen zu Variablen und deren Namen.

6 Datentypen

Ein grundlegender Baustein von Programmen sind Werte (*values*), wie z.B. Zahlen oder Buchstaben. Jeder Wert gehört zu einem bestimmten Datentypen. Dieser Datentyp kann mit der Funktion `typeof()` abgefragt werden:

```
julia> typeof(2)
Int64

julia> typeof(42.0)
Float64

julia> typeof("hello world")
String

julia> typeof('c')
Char

julia> typeof(true)
Bool
```

6.1 Characters und Strings

6.1.1 Characters

Ein Wert vom Typ `Char` repräsentiert ein einzelnes Zeichen und ist von einfachen Anführungszeichen umgeben. Dabei kann jedes verfügbare *Unicode*-Zeichen als `Char` verwendet werden:

```
julia> 'x'
'x': ASCII/Unicode U+0078 (category Ll: Letter, lowercase)

julia> 'α'
'α': Unicode U+03b1 (category Ll: Letter, lowercase)

julia> typeof('x')
Char
```

6.1.2 Strings

Strings werden von doppelten Anführungszeichen umgeben.

```
julia> "Das ist ein String"

"Das ist ein String"

julia> typeof("19.0")
String
```

Ein `String` ist eine Aneinanderreihung von `Char`-Werten. Über eckige Klammern kann auf jedes dieser Zeichen zugegriffen werden:

```
julia> fruit = "banana"
"banana"

julia> third = fruit[3]
'n': ASCII/Unicode U+006e (category Ll: Letter, lowercase)

julia> last = fruit[end]
'a': ASCII/Unicode U+0061 (category Ll: Letter, lowercase)

julia> secondlast = fruit[end-1]
'n': ASCII/Unicode U+006e (category Ll: Letter, lowercase)
```

Nachdem in der ersten Zeile der Variable `fruit` ein `String` zugewiesen wird, greifen die übrigen Zeilen auf diesen `String` zu und weisen jeweils ein Zeichen einer neuen Variablen zu. Der Ausdruck in den eckigen Klammern heißt dabei *Index* und muss ein Integer sein. Die Indexierung in Julia startet bei 1, die Variable `third` beinhaltet also das dritte Zeichen des Strings. Für den Zugriff auf das letzte Zeichen kann der Index `end` verwendet werden.

Neben einzelnen Zeichen besteht auch die Möglichkeit, auf einen Teil des Strings zuzugreifen. Dies erfolgt durch Eingabe des ersten und letzten Index des Teils, getrennt durch einen Doppelpunkt, `:`.

```
julia> fruit[3:end]
"nana"
```

Die Anzahl der Zeichen eines Strings kann mit der Funktion `length()` abgefragt werden:

```
julia> length("julia")
5
```

Strings sind nicht veränderbar. Austauschen eines Zeichens durch Zugriff auf einen bestimmten Index ist daher nicht möglich. Julia gibt dann einen *Method-Error* aus.

```
julia> s = "banana"
"banana"

julia> s[3:end] = "nane"
Error: MethodError: no method matching setindex!(::String, ::String, ::UnitRange{Int64})
```

Mittels `replace()` kann allerdings eine Zeichenfolge im String durch eine neue Zeichenfolge ersetzt und in einer neuen Variable abgespeichert werden:

```
julia> s_1 = replace(s,"nana"=>"nane")  # Ersetze im String s "nana" durch "nane"
"banane"

julia> s_2 = replace(s,'a'=>'u')  # Ersetze im String s 'a' durch 'u'
"bununu"

julia> s  # Der String s bleibt unverändert
"banana"
```

Das erste Argument der Funktion ist der zu verändernde String. Das zweite Argument setzt sich zusammen aus dem String oder Character, der ersetzt werden soll, einem Pfeil bestehend aus `=` und `>` sowie dem String oder Character der, eingefügt werden soll. Die ursprüngliche Variable `s` bleibt dabei unverändert.

Es gibt mehrere Möglichkeiten, Strings zusammenzusetzen. Eine davon ist die Verwendung des *-Operators.

```
julia> name = "Raphael"
"Raphael"

julia> "Der Name"*name*" hat "*string(length(name))*" Buchstaben."
"Der NameRaphael hat 7 Buchstaben."
```

Der *-Operator ist für `String` und `Int` nicht definiert, daher muss `length(name)` erst durch die Funktion `string()` in einen `String` konvertiert werden. In dieser Form wird der Code schnell unübersichtlich, weshalb Julia die sogenannte „String-Interpolation“ bietet. Jeder Ausdruck in runden Klammern oder jeder Variablenname nach dem Symbol `$` wird ausgewertet und das Ergebnis in den String eingegliedert. Diese Ausdrücke können nicht nur Werte und Variablen, sondern auch Funktionen und Operatoren enthalten.

```
julia> "Der Name $name hat $(length(name)) Buchstaben."
"Der Name Raphael hat 7 Buchstaben."
```

Der ^-Operator wiederholt einen String beliebig oft:

```
julia> "stop"^5
"stopstopstopstopstop"
```

Die Funktionen `uppercase` und `lowercase` wandeln alle Buchstaben eines Strings in Groß- oder Kleinbuchstaben um. Beim Vergleichen von Strings unabhängig von Groß- und Kleinschreibung können diese Funktionen hilfreich sein.

```
julia> lowercase("In diesem Satz stehen auch Grossbuchstaben.")
"in diesem satz stehen auch grossbuchstaben."

julia> uppercase("gross- und kleinschreibung wird hier klein geschrieben.")
"GROSS- UND KLEINSCHREIBUNG WIRD HIER KLEIN GESCHRIEBEN."

julia> "Julia" == "julia"
false

julia> lowercase("Julia") == lowercase("julia")
true

julia> uppercase("Julia") == uppercase("julia")
true
```

6.1.3 Special Characters und Escaping

In den meisten Programmiersprachen gibt es spezielle Zeichen für Tabulatorabstände oder Zeilenumbrüche. Diese Zeichen werden mit einem Backslash (\) eingeleitet.

```
julia> '\t'  # Tabulatorabstand
'\t': ASCII/Unicode U+0009 (category Cc: Other, control)

julia> '\n'  # Zeilenumbruch
'\n': ASCII/Unicode U+000a (category Cc: Other, control)

julia> print("tabs\tabs")
tabs    abs
julia> print("Erste Zeile\nZweite Zeile")
Erste Zeile
Zweite Zeile
```

Dementsprechend ist auch der Backslash ein speziell zu verwendendes Zeichen in Julia, das mit einem vorangehenden Backslash maskiert werden muss. Dieses Maskieren nennt man auch *Escaping*. Neben dem Backslash gibt es noch weitere spezielle Zeichen, die maskiert werden müssen. Beispielsweise würde Julia hinter einem `$`-Zeichen einen auszuwertenden Ausdruck und bei jedem Anführungszeichen den Anfang oder das Ende eines Strings erwarten.

```
julia> print("tabs\\tabs")
tabs\tabs
julia> print("Julia ist jeden \$ Wert.")
Julia ist jeden $ Wert.
julia> print("Dieser Satz enthält \"Anführungszeichen\".")
Dieser Satz enthält "Anführungszeichen".
julia> print("""Dieser Satz enthält "Anführungszeichen".""")
Dieser Satz enthält "Anführungszeichen".
```

Strings können auch mit drei Anführungszeichen geschrieben werden. In diesem Fall müssen Anführungszeichen innerhalb des Strings nicht maskiert werden.

Weitere interessante Funktionen sind im Online-Buch Think Julia[12] und in der Dokumentation unter Manual/Strings erläutert.

6.2 Zahlen

In Julia sind verschiedene Datentypen für Zahlen definiert. Im Beispiel oben ist der ganzzahlige Wert `2` vom Typ `Int64` und `42.0` vom Typ `Float64`. Diese beiden Datentypen werden für Zahlen am häufigsten verwendet. Daneben bietet Julia allerdings noch die Typen `Complex` und `Rational` für komplexe und rationale Zahlen.

```
julia> typeof(1.3+4im)
Complex{Float64}

julia> typeof(23//4)
Rational{Int64}
```

Für Zahlen vom Typ `Int` und `Float` gibt es verschiedene Typen, die sich nach ihrer Anzahl an Bits im Speicher unterscheiden. Die Zahl am Ende des Typs gibt genau diesen Wert an. Neben den Typen `Int64` und `Float64` gibt es daher noch `Int8`, `Int16`, `Int32` und `Int128` sowie `Float16` und `Float32`. Diese Zahlentypen benötigen zwar weniger Speicher (ausgenommen `Int128`), es ist aber zu beachten, dass diese Integer-Typen nur für Zahlen bis zu einer gewissen Größe funktionieren und die beiden Float-Typen weniger Kommastellen bereitstellen als `Float64`. Eine Übersicht dazu bieten die Tabellen unter Manual/Integers and Floating-Point Numbers in der Julia-Dokumentation.

Mithilfe der Operatoren `+`, `-`, `*` und `/` können Zahlen addiert, subtrahiert, multipliziert und dividiert werden:

```
julia> 17 + 2
19

julia> 23.0 - 4
19.0

julia> 1.0 * 19.0
19.0

julia> 38 / 2
19.0
```

Während der Datentyp des Ergebnisses einer Addition, Subtraktion und Multiplikation abhängig vom Datentyp der Eingangswerte ist, ergibt die Division zweier ganzzahliger Werte immer eine Float-Zahl. Potenzen werden mit `^` eingegeben:

```
julia> 2^3
8
```

Die Reihenfolge der Operationen folgt dabei den üblichen mathematischen Regeln:

1. Klammern vor
2. Potenzen vor

[12] https://benlauwens.github.io/ThinkJulia.jl/latest/book.html#chap08

3. Multiplikation und Division vor
4. Addition und Subtraktion.

Gleichrangige Operationen führt Julia immer der Reihe nach von links nach rechts aus. Klammern bieten die Möglichkeit, den Code zu strukturieren und besser lesbar zu machen, auch wenn diese nicht immer notwendig sind:

```
julia> d = 5;

julia> A = d^2*π/4#OhneAbständeliestsichCodeschwer
19.634954084936208

julia> A = (d^2 * π) / 4  # Abstände und Klammern verbessern die Lesbarkeit
19.634954084936208
```

Manchmal ist es notwendig, zu einer Variable eine Zahl zu addieren und das Ergebnis in der gleichen Variable zu speichern. Für solche Fälle gibt es die Operatoren `+=`, `-=`, `*=`, `/=` und `^=`.

```
julia> a = 1.0
1.0

julia> a = a + 1
2.0

julia> a += 1  # wie a = a + 1
3.0

julia> a -= 1  # wie a = a - 1
2.0

julia> a *= 2  # wie a = a * 2
4.0

julia> a /= 2  # wie a = a / 2
2.0

julia> a ^= 2  # wie a = a^2
4.0
```

Die Funktionen `string()` und `parse()` konvertieren Zahlen in Strings und umgekehrt:

```
julia> string(19.0)
"19.0"

julia> parse(Int64,"17")
17

julia> parse(Float64,"1.2345")
1.2345
```

Wird `parse()` mit einem ungültigen String aufgerufen (z.B. `parse(Int,"foo")`) verursacht das einen Fehler. Es gibt die Möglichkeit, anstelle von `parse()`, `tryparse()` zu verwenden. Die Funktion verursacht keinen Fehler, sondern gibt für einen nicht umwandelbaren String `nothing` zurück.

`float()` und `Int()` wandeln beliebige Zahlen in Float-Zahlen bzw. in ganzzahlige Werte um. Bei Verwendung von `Int()` muss gewährleistet sein, dass die übergebene Zahl tatsächlich ganzzahlig ist, sonst meldet Julia einen `InexactError`. `trunc` wandelt eine Float-Zahl direkt in ein Integer um und ignoriert dabei die Nachkommastellen.

```
julia> float(3)
3.0

julia> float(4//3)
1.3333333333333333

julia> Int(2.3)
Error: InexactError: Int64(2.3)

julia> Int(2.0)
2

julia> Int(4//4)
1
```

```
julia> trunc(3.99999)
3.0

julia> trunc(-2.3)  # trunc rundet nicht ab!
-2.0
```

Zum Runden von Zahlen eignen sich die Funktionen `round()`, `floor()` und `ceil()`.

```
julia> round(1.47)
1.0

julia> floor(1.47)
1.0

julia> ceil(1.47)
2.0
```

Während erstere Funktion zur nächstliegenden ganzen Zahl rundet, ermöglichen die anderen beiden Funktionen Ab- und Aufrunden von Zahlen. Die Anzahl der zu rundenden Stellen kann bei jeder dieser drei Funktionen über die zusätzlichen Argumente `digits` und `sigdigits` festgelegt werden.

```
julia> round(345.123; digits=2)
345.12

julia> round(345.123; sigdigits=2)
350.0

julia> round(0.00345123; digits=4)
0.0035

julia> round(0.00345123; sigdigits=4)
0.003451
```

Das Keyword `digits` definiert dabei die Anzahl der Nachkommastellen, `sigdigits` legt die Anzahl der signifikanten Stellen fest. Einfacher gesagt: `digits` beginnt ab dem Komma zu zählen, während `sigdigits` ab der ersten Ziffer ungleich 0 zählt.

Zwei weitere interessante Operatoren sind die ganzzahlige Division ÷[13] (*floor division*) und der Modulo-Operator, `%`.

```
julia> 7÷3
2

julia> 7%3
1
```

÷ führt eine Division durch und rundet das Ergebnis auf die nächste ganze Zahl ab. Der Modulus-Operator gibt den Rest einer Division aus. Damit kann z.B. leicht überprüft werden, ob eine Zahl durch eine andere Zahl teilbar ist.

Julia hat standardmäßig mathematische Funktionen wie `sin`, `cos`, `log`, `abs` und viele mehr inkludiert.

```
julia> sin(π/6)  # Die Argumente von sin, cos und tan müssen in rad angegeben werden
0.49999999999999994

julia> cos(π/6)
0.8660254037844387

julia> tan(π/6)
0.5773502691896257

julia> log(100)  # log ist standardmäßig der natürliche Logarithmus
4.605170185988092

julia> log(10,100)  # Mit zwei Argumenten ist das erste die Basis
2.0

julia> log10(100)  # log10 ist der Logarithmus mit Basis 10
2.0

julia> abs(-19.0)
19.0
```

[13] ÷ wird durch Eingabe von `\div` und Drücken der TAB-Taste erstellt.

`sin(π/6)` sollte exakt `0.5` ergeben. Da der Datentyp von `π/6`, `Float64` ist, ist keine exakte Darstellung der eigentlich irrationalen Zahl möglich. Bei Float-Zahlen wird nur eine begrenze Anzahl von Dezimalstellen gespeichert, wodurch hier ein vernachlässigbarer Fehler entsteht.

Eine weitere interessante Funktion ist `zero`. Sie nimmt eine Zahl als Argument und gibt `0` im gleichen Datentyp zurück. Besonders im Zusammenhang mit Typenstabilität (siehe Abschnitt Kontrollstruktur) kann die Funktion hilfreich sein.

```
julia> zero(19)
0

julia> zero(19.0)
0.0

julia> zero(38//2)
0//1

julia> zero(19.0+3im)
0.0 + 0.0im
```

6.3 Booleans

Eine Wert vom Typ `Bool` kann entweder `true` oder `false` annehmen. Diesen Datentypen kann man sich wie eine Zahl vorstellen, die entweder `0` (im Falle von `false`) oder `1` (im Falle von `true`) ist. Verschiedene Operationen in Julia geben eine Zahl vom Typ `Bool` zurück, wie z.B. das Vergleichen zweier Zahlen:

```
julia> x = 13; y = 17.0;

julia> x == y  # Ist x gleich y?
false

julia> x != y  # Ist x ungleich y?
true

julia> x ≠ y  # \ne TAB (wie !=)
true

julia> x > y  # Ist x größer als y?
false

julia> x < y  # Ist x kleiner als y?
true

julia> x >= y  # Ist x größer oder gleich y?
false

julia> x ≥ y  # \ge TAB (wie >=)
false

julia> x <= y  # Ist x kleiner oder gleich y?
true

julia> x ≤ y  # \le TAB (wie <=)
true
```

Mit den Operatoren `>` und `<` können auch zwei Wörter auf ihre alphabetische Reihenfolge überprüft werden:

```
julia> "apple" < "banana"
true

julia> "apple" > "banana"
false

julia> "apple" > "Banana"
true
```

Dabei ist zu beachten, dass Großbuchstaben immer vor Kleinbuchstaben eingeordnet werden.

Alle Ausdrücke, die eine Zahl vom Typ `Bool` zurückgeben, können mit den logischen Operatoren `&&` (*and*), `||` (*or*) und `!` (*not*) kombiniert werden.

```
julia> s = "melon";

julia> s > "banana" && length(s) == 6  # Beide Bedingungen müssen true ergeben
false

julia> s > "banana" || length(s) == 6  # Mindestens eine Bedingung muss true ergeben
true

julia> !(s > "banana" && length(s) == 6)  # Beide Bedingungen müssen false ergeben
true
```

Bei `&&` müssen beide Ausdrücke vor und nach dem Operator `true` ergeben, damit der gesamte Ausdruck `true` ist. Damit der Ausdruck mit `||` erfüllt ist und `true` ergibt, muss lediglich einer der beiden Ausdrücke erfüllt sein. `!` kehrt den darauffolgenden logischen Wert um.

Wie bei den arithmetischen Operationen wird auch bei logischen Operatoren eine definierte Reihenfolge eingehalten. Zuerst werden alle vergleichenden Operatoren ausgewertet (`==`, `!=`, `≠`, `>`, `<`, `>=`, `<=`, `≥`, `≤`), danach die Operatoren `&&` und `||`. Vergleichende Operatoren können aneinandergekettet werden. Das Ergebnis der Aneinanderkettung von Vergleichen ist gleich dem Ergebnis von mit `&&` ausgewerteten Vergleichen:

```
julia> a = 1; b = 1; c = 2;

julia> a == b < c > a
true

julia> a == b && b < c && c > a
true
```

7 Funktionen

In den vorhergehenden Kapiteln wurden schon einige Funktionen verwendet (z.B. `abs`, `typeof` oder `length`). Eine Funktion wird durch ihren Namen, gefolgt von runden Klammern aufgerufen. Innerhalb dieser runden Klammern stehen die *Argumente* der Funktion. Vereinfacht lässt sich eine Funktion durch eine Box erklären, die mit Werten (den Argumenten) befüllt wird. Innerhalb der Box wird dann Code ausgeführt und am Ende ein Ergebnis (*Return Value*) zurückgegeben. Der Return Value kann einfach einer Variable zugewiesen werden:

```
julia> l = length("spam")
4
```

Während manche Funktionen keine Argumente benötigen, erfordern andere Funktionen mehrere Eingangswerte.

```
julia> pwd()
"/private/var/folders/0z/xyn859fx3vj4762qh24f8dq80000gn/T/jl_mbP3kE"

julia> length("banana")
6

julia> round(3.1415,digits=3)
3.142
```

`pwd` steht für *print working directory* und zeigt den aktuellen Arbeitspfad an.

Analog gibt es sowohl Funktionen, die keinen *Return Value* zurückgeben, als auch Funktionen, die mehrere Ergebnisse ausgeben.

```
julia> p = println("Diese Zeile wird angezeigt, println weist p aber keinen Wert zu")
Diese Zeile wird angezeigt, println weist p aber keinen Wert zu

julia> p

julia> a = [10,35,4,65,9];

julia> maximum(a)
65

julia> findmax(a)
(65, 4)
```

Der erste Aufruf von `println` schreibt zwar das übergebene Argument in der REPL, die Ausgabe von `p` in der REPL zeigt allerdings keinen Wert an. Das liegt daran, dass `println` keinen Return Value an die Variable `p` übergibt (genau genommen gibt die Funktion den Wert `nothing` zurück). In der nächsten Zeile wird das Array `a` erstellt. Auf Arrays wird später noch genauer eingegangen. Während `maximum` nur den Maximalwert von `a` zurückgibt, ermittelt `findmax` zusätzlich die Stelle des Maximalwertes.

Die Argumente einer Funktion können verschiedenste Ausdrücke sein. Sowohl Werte und Variablen, als auch Funktionen können als Argument übergeben werden. Julia wertet diese Ausdrücke dann aus und übergibt das Ergebnis an die Funktion.

```
julia> s1 = "foo"; s2 = "bar";

julia> string(s1, s2)
"foobar"

julia> round(20/9)
2.0

julia> length(pwd())
66
```

Manchen Funktionen können sogenannte *Keyword Arguments* übergeben werden. `round` ist beispielsweise eine solche Funktion. Mit dem *Keyword* `digits` gefolgt von `=` und einer `Int`-Zahl, kann die Anzahl der Nachkommastellen festgelegt werden.

```
julia> round(2.3345323,digits=2)
2.33
```

Gerade bei Funktionen mit vielen (optionalen) Argumenten sind Keyword Arguments sehr hilfreich, da sie in beliebiger Reihenfolge an die Funktion übergeben werden können.

7.1 Neue Funktionen erstellen

Vor allem für Operationen, die oft wiederholt werden, ist es sinnvoll eigene Funktionen zu definieren; und zwar aus folgenden Gründen:

- Das Debugging kann schrittweise in den einzelnen Funktionen erfolgen.
- Änderungen im Code müssen nur in den einzelnen Funktionen umgesetzt werden.
- Durch das Zusammenfassen von Operationen in Funktionen wird der Code besonders performant.
- Sinnvoll konzipierte Funktionen für kleine Aufgaben sind vielseitig einsetzbar und können meist in mehreren Programmen verwendet werden.

Die Definition einer eigenen Funktion beginnt mit dem Keyword `function` am Zeilenanfang, gefolgt vom Funktionsnamen und den Argumenten in runden Klammern. Danach folgen beliebig viele Zeilen Code und ein abschließendes `end`. Funktionsnamen können wie Variablennamen gebildet werden. Groß- und Kleinbuchstaben sind zulässig, es dürfen nahezu alle Unicode-Zeichen verwendet werden. Der Name darf allerdings nicht mit einer Zahl beginnen oder einem von Julia verwendeten Keyword entsprechen. Zusätzlich sollte ein Name nicht gleichzeitig für eine Funktion und eine Variable verwendet werden. Bei jedem Aufruf der Funktion werden die Zeilen zwischen `function` und `end` ausgeführt und das Ergebnis der letzten Zeile als Return Value zurückgegeben. Der Return Value einer Funktion kann aber auch mit dem Keyword `return` festgelegt werden.

```
function my_square(x)
    println("Quadrat von $(x) wird berechnet...")  # Diese Zeile wird angezeigt
    return x^2  # Diese Zeile wird ausgeführt und als Return Value ausgegeben
end
```

```
julia> my_square(3)
Quadrat von 3 wird berechnet...
9

julia> my_square(8)
Quadrat von 8 wird berechnet...
64
```

Die Funktion `my_square` schreibt zuerst einen String in die REPL und gibt dann das Quadrat des übergebenen Arguments zurück. Generell ist es in Julia nicht notwendig, den Block innerhalb der Funktionsdefinition einzurücken, dadurch wird der Code allerdings wesentlich übersichtlicher.

Funktionen ohne Argumente können durch leere Klammern erstellt werden:

```
function hello_world()
    println("Hello World")
end
```

Für die Definition besonders kurzer Funktionen eignet sich folgende Abkürzung:

```
my_square(x) = x^2
```

Eine neue Funktion kann im Code erst verwendet werden, nachdem sie definiert wurde. Da Julia ein Skript vom Beginn an Zeile für Zeile bis zum Ende ausführt, ist es daher sinnvoll, die Funktionsdefinitionen an den Anfang des Skripts zu schreiben.

Im Abschnitt Hilfe wird beschrieben, dass Julia selbstdokumentiert ist. Diese Dokumentation erfolgt durch sogenannte Docstrings direkt vor der Funktionsdefinition. Ein Docstring wird durch drei Anführungszeichen begrenzt und kann folgendermaßen aussehen:

```
"""
    my_square(x)

Schreibe "Quadrat von x wird berechnet..." in die REPL und gib das Quadrat von x zurück.
"""
function my_square(x)
    println("Quadrat von $(x) wird berechnet...")
    return x^2
end
```

Nach Ausführung dieses Code-Blocks wird in der Hilfe zu `my_square` das Docstring angezeigt.

Mehr Informationen zu Funktionen gibt es in der Dokumentation unter Manual/Functions.

7.2 Scope

Der *Scope* einer Variable ist ein Bereich, in dem diese Variable sichtbar ist. In Julia gibt es den *Global Scope* und den *Local Scope*. Die Definition einer Funktion erzeugt einen solchen Local Scope. Auf eine globale Variable kann auch in einem Local Scope zugegriffen werden. Umgekehrt ist es nicht möglich, im Global Scope auf eine lokale Variable zuzugreifen:

```
function f(x)
    a = x^2
end
```

```
julia> a = 1;  # Globale Variable a erzeugen

julia> f(3)  # Im Local Scope von f wird die lokale Variable a erzeugt
9

julia> a  # Der Aufruf von f verändert die globale Variable a nicht
1
```

Selbst wenn in der Funktion `a` ein Wert zugewiesen wird, bleibt der Wert von a unverändert, da die Zuweisung in der ersten Zeile (`a = 1`) im Global Scope geschieht und die zweite Zuweisung (`a = x^2`) im Local Scope der Funktion ausgeführt wird. Die beiden Variablen heißen zwar gleich, beeinflussen sich aber gegenseitig nicht. Nachdem die Funktion ausgeführt wurde, werden alle Variablen des Local Scope dieser Funktion gelöscht und können nicht mehr aufgerufen werden. Nur noch der Return Value bleibt im Speicher (sofern dieser einer Variable zugewiesen wird).

```
function g(x)
    v = "fuer immer verloren"  # Lokale Variable v erzeugen
    x*2  # Return Value
end
```

```
julia> a = g(3)
6
```

```
julia> v
Error: UndefVarError: v not defined
```

Mehr Informationen zu Scopes gibt es in der Dokumentation unter Manual/Scope of Variables.

7.3 Methoden

Ein Wesentliches Feature von Funktionen in Julia ist *Multiple Dispatch*. Eine Funktion kann unterschiedliche Operationen ausführen, abhängig davon, welchen Datentyp die Argumente haben und wieviele Argumente übergeben werden.

```
julia> my_mult(x::Number) = x*2  # Erste Methode von my_mult
my_mult (generic function with 1 method)

julia> my_mult(x::String) = x^2  # Zweite Methode von my_mult
my_mult (generic function with 2 methods)

julia> my_mult(x::String,y::Int64) = x^y  # Dritte Methode von my_mult
my_mult (generic function with 3 methods)

julia> my_mult(1.45)
2.9

julia> my_mult("spam")
"spamspam"

julia> my_mult("foo",4)
"foofoofoofoo"
```

Die Funktion `my_mult` hat nun 3 *Methoden*. Die erste Methode ist für ein Argument vom Typ `Number` definiert. Über `::` gefolgt vom Datentypen wird dieser dem Argument zugewiesen. In Julia sind Datentypen in einer Hierarchie strukturiert. Jeder Typ kann einen übergeordneten *Supertype* und/oder mehrere untergeordnete *Subtypes* haben. Eine Methode für einen bestimmten Datentypen kann auch auf dessen Subtypen angewendet werden. Eine für `Number` definierte Funktion kann daher auch auf alle möglichen Integer- und Float-Typen, aber auch komplexe und rationale Zahlen angewendet werden, da es sich dabei um Subtypen von `Number` (oder um Subtypen von Subtypen von Subtypen ... von `Number`) handelt. Die zweite Methode wird angewendet, wenn ein `String` an die Funktion übergeben wird und die dritte Funktion wird nur dann angewendet, wenn zusätzlich zu einem `String` eine `Int64`-Zahl Argument ist.

Abhängig von den eingegebenen Argumenten sucht Julia die passende Methode aus, wenn keine Methode für die Art und Anzahl von Argumenten definiert ist, gibt Julia einen *MethodError* zurück.

Mehr Informationen zu Methoden und Multiple Dispatch gibt es in der Dokumentation unter Manual/Methods.

7.4 Aufgaben

7.4.1 Dreimal printen

Schreibe eine Funktion `print_three`, die ein beliebiges Argument dreimal in einer Zeile printed und dabei einen Abstand zwischen den Argumenten einfügt.

Beispiele:

```
julia> print_three("spam")
spam spam spam

julia> print_three(19)
19 19 19
```

7.4.2 Zwei Funktionen verketten

Schreibe eine Funktion `verkette(f,g,x)`, die die beiden Funktionen `f` und `g` verkettet und für `x` auswertet (`f(g(x))`). Die ausgewertete Funktion soll als Return Value zurückgegeben werden.

Beispiele:

```
julia> f(x) = x+2
f (generic function with 1 method)

julia> g(x) = x^2
g (generic function with 1 method)

julia> verkette(f,g,3)
11

julia> verkette(length,g,"spam")
8
```

7.4.3 Kreisfläche berechnen

Schreibe eine Funktion `kreis_flaeche(r)`, die den Radius eines Kreises als Argument benötigt und dann die Fläche dieses Kreises als Return Value zurückgibt.

Beispiele:

```
julia> kreis_flaeche(3)
28.274333882308138

julia> kreis_flaeche(2.57)
20.74990531769522
```

7.4.4 Kreisumfang berechnen

Schreibe eine Funktion `kreis_umfang(r)`, die den Radius eines Kreises als Argument benötigt und dann den Umfang dieses Kreises als Return Value zurückgibt.

Beispiele:

```
julia> kreis_umfang(3)
18.84955592153876

julia> kreis_umfang(2.57)
16.147786239451534

julia> 2 * kreis_flaeche(5)/kreis_umfang(5)
5.0
```

7.4.5 Kreiseigenschaften printen

Schreibe eine Funktion `kreis(r)`, die den Radius eines Kreises als Argument benötigt und Radius, Fläche und Umfange dieses Kreises auf zwei Nachkommastellen gerundet printed. Es können gleich die Funktionen der vorherigen beiden Aufgaben verwendet werden. Dafür muss die Funktionsdefinition allerdings ebenfalls in den Block des Kontrollprogrammes kopiert werden:

```
@Aufgabe "7.4.5" begin
    function kreis_flaeche(r)
        # Berechnung Kreisflaeche
    end
    function kreis_umfang(r)
        # Berechnung Kreisumfang
    end
    function kreis(r)
        # Code zum Lösen der Aufgabe
    end
end
```

Die Ausgabe muss genau gleich wie im Beispiel unten formatiert sein, damit das Kontrollprogramm das Beispiel als korrekt gelöst auswertet.

Beispiele:

```
julia> kreis(3)
r = 3.0
A = 28.27
U = 18.85
```

8 Arrays, Range, Tuple, Dict

Dieses Kapitel beinhaltet einige wichtige Typen in Julia, die mehrere Werte aufnehmen können. Derartige Datentypen werden *Sammlungen* (englisch *collections*) genannt.

8.1 Arrays

Wie ein *String* ist ein *Array* eine geordnete Abfolge von Werten. Während ein String aus Werten vom Typ *Char* besteht, können Arrays Werte aller möglichen Typen enthalten. Die Werte innerhalb eines Arrays heißen *Elemente*.

Arrays können auf verschiedenste Weisen erzeugt werden. Alle Elemente getrennt durch ein Komma innerhalb von eckigen Klammer zu schreiben ist die einfachste Variante:

```
primes = [2, 3, 5, 7, 11, 13]
baustoffe = ["Holz", "Beton", "Stahl"]
```

Die erste Zeile erzeugt ein Array aus sechs Int-Zahlen, die zweite Zeile erzeugt ein Array aus drei Strings. Die Elemente eines Arrays müssen aber nicht vom gleichen Typ sein; sogar Arrays innerhalb von Arrays sind möglich:

```
stahl = ["S235", 235.0, [10, 20]]
```

Mit `typeof()` kann der Typ eines Arrays abgefragt werden:

```
julia> typeof(primes)
Array{Int64,1}

julia> typeof(baustoffe)
Array{String,1}

julia> typeof(stahl)
Array{Any,1}
```

Innerhalb der geschwungenen Klammer steht dabei zuerst, von welchem Typ die Werte des Arrays sind, und an zweiter Stelle, welche Ordnung das Array hat. Ein *leeres Array* hat keine Elemente und kann z.B. mit leeren, eckigen Klammern (`[]`) erzeugt werden. Das so erzeugte leere Array ist dabei vom Typ `Array{Any,1}`. Ein leeres Array mit Werten eines bestimmten Typs, wie z.B. `Float64`, kann mittels `Array{Float64,1}()` oder `Float64[]` erzeugt werden; mit der Schreibweise `Float64[]` können allerdings nur Arrays erster Ordnung erzeugt werden.

```
julia> []
0-element Array{Any,1}

julia> Array{Float64,1}()
0-element Array{Float64,1}

julia> Float64[]
0-element Array{Float64,1}
```

8.1.1 Arrays sind veränderbar

Die Syntax für den Zugriff auf ein bestimmtes Element eines Arrays ist dieselbe wie beim Zugriff auf einen Character innerhalb eines Strings: das Array, direkt gefolgt von einer Int-Zahl als Index innerhalb eckiger Klammern.

```
julia> baustoffe[2]
"Beton"
```

Wie bei Strings kann auch auf Teile eines Arrays zugegriffen werden:

```
julia> primes[2:4]
3-element Array{Int64,1}:
 3
 5
 7

julia> primes[3:end]
```

```
4-element Array{Int64,1}:
  5
  7
 11
 13
```

Anders als Strings sind Arrays allerdings veränderbar. Das heißt, einzelne Elemente oder Teile eines Arrays können durch den =-Operator neu zugewiesen werden. Der zugewiesenen Datentyp muss mit dem Typen des Arrays übereinstimmen.

```
julia> baustoffe[3] = "Ziegel"
"Ziegel"

julia> baustoffe[2] = 2
Error: MethodError: Cannot `convert` an object of type Int64 to an object of type String
Closest candidates are:
  convert(::Type{T<:AbstractString}, !Matched::T<:AbstractString) where T<:AbstractString at strings/basic.jl:208
  convert(::Type{T<:AbstractString}, !Matched::AbstractString) where T<:AbstractString at strings/basic.jl:209
  convert(::Type{T}, !Matched::T) where T at essentials.jl:167

julia> print(baustoffe)
["Holz", "Beton", "Ziegel"]
```

Folgendes Verhalten kann dabei möglicherweise zu Verwirrung führen:

```
julia> a = [1, 4, 3];  # Array a erzeugen

julia> b = a;  # Array a zu b zuweisen

julia> b[2] = 2;  # Zweites Element von b ersetzen

julia> print("a = $(a)")  # Auch a wurde verändert
a = [1, 2, 3]
julia> print("b = $(b)")
b = [1, 2, 3]
```

Verändern eines Wertes im Array `b` verändert also auch den Array `a`. Das liegt daran, dass der Ausdruck `b = a` den Array `a` **nicht** kopiert, sondern auf ihn verweist. Zum Kopieren eines Arrays eignet sich die Funktion `copy()`.

```
julia> a = [1, 4, 3];  # Array a erzeugen

julia> b = copy(a);  # Kopie von a in b speichern

julia> b[2] = 2;  # Zweiten Eintrag von b ersetzen

julia> print("a = $(a)")  # a bleibt unverändert
a = [1, 4, 3]
julia> print("b = $(b)")
b = [1, 2, 3]
```

8.1.2 Funktionen der Array Library

`length` gibt die Anzahl der Elemente eines Arrays zurück. Ein Array innerhalb eines Arrays ist dabei ein Element.

```
julia> print(primes)
[2, 3, 5, 7, 11, 13]
julia> length(primes)
6

julia> print(stahl)
Any["S235", 235.0, [10, 20]]
julia> length(stahl)
3

julia> length(stahl[3])
2
```

`push!` fügt ein neues Element am Ende des Arrays ein:

```
julia> fib = [1, 1, 2, 3, 5];
```

```
julia> push!(fib, 8);

julia> print(fib)
[1, 1, 2, 3, 5, 8]
```

`append!` hängt ein Array am Ende des Arrays an:

```
julia> append!(fib, [13, 21, 34]);

julia> print(fib)
[1, 1, 2, 3, 5, 8, 13, 21, 34]
```

`insert!` fügt ein Element bei einem bestimmten Index in das Array ein:

```
julia> countdown = [5, 4, 3, 1, 0];

julia> insert!(countdown, 4, 2);  # Die Zahl 2 wird an vierter Stelle eingefügt

julia> print(countdown)
[5, 4, 3, 2, 1, 0]
```

`sort!` sortiert die Elemente eines Arrays vom niedrigsten zum höchsten Wert:

```
julia> c = ['d', 'g', 'a', 'W', 'f'];

julia> sort!(c);

julia> print(c)
['W', 'a', 'd', 'f', 'g']
```

`sort` hingegen erzeugt eine **sortierte Kopie** des Arrays:

```
julia> c1 = ['d', 'g', 'a', 'W', 'f'];

julia> c2 = sort(c1);

julia> print("c1 = $(c1)")
c1 = ['d', 'g', 'a', 'W', 'f']
julia> print("c2 = $(c2)")
c2 = ['W', 'a', 'd', 'f', 'g']
```

Generell hat sich in Julia etabliert, dass Funktionen, die direkt ihr Argument verändern, mit `!` enden.

8.1.3 Elemente löschen

Es gibt verschiedene Möglichkeiten, einzelne Elemente aus einem Array zu löschen. `pop!` entfernt das letzte Element aus dem Array und gibt dieses Element zurück:

```
julia> chars = ['a', 'b', 'c'];

julia> pop!(chars)
'c': ASCII/Unicode U+0063 (category Ll: Letter, lowercase)

julia> print(chars)
['a', 'b']
```

Das erste Element eines Arrays kann mittels `popfirst!` entfernt werden:

```
julia> chars = ['a', 'b', 'c'];

julia> popfirst!(chars)
'a': ASCII/Unicode U+0061 (category Ll: Letter, lowercase)

julia> print(chars)
['b', 'c']
```

Ist der Index des zu entfernenden Elements bekannt, kann dieses mit `splice!` aus dem Array gelöscht werden:

```
julia> chars = ['a', 'b', 'c'];

julia> splice!(chars, 2)
'b': ASCII/Unicode U+0062 (category Ll: Letter, lowercase)
```

```
julia> print(chars)
['a', 'c']
```

Ähnlich funktioniert auch `deleteat!`. Der Unterschied ist dabei, dass nicht das entfernte Element, sondern das Array zurückgegeben wird:

```
julia> chars = ['a', 'b', 'c'];

julia> deleteat!(chars,2)
2-element Array{Char,1}:
 'a'
 'c'

julia> print(chars)
['a', 'c']
```

8.1.4 Strings und Arrays

Strings und Arrays sind in manchen Punkten sehr ähnlich. Ein String ist eine Abfolge von Characters und da ein Array mit beliebigen Datentypen befüllt werden kann, ist ein Array aus Characters auch eine Abfolge von Characters. Trotzdem sind beides verschiedene Typen mit unterschiedlichen Eigenschaften. `collect` konvertiert einen String in ein Array von Characters:

```
julia> a = collect("spam");

julia> print(a)
['s', 'p', 'a', 'm']
```

Mittels `split` kann ein String entlang bestimmter Zeichen in mehrere Teile getrennt werden. So können beispielsweise Sätze in einzelne Worte aufgeteilt werden:

```
julia> s1 = split("Julia ist großartig")  # Standardmäßig wird bei jedem Leerzeichen getrennt
3-element Array{SubString{String},1}:
 "Julia"
 "ist"
 "großartig"

julia> s2 = split("Stahl;Beton;Ziegel;Holz",';')  # Als zweites Argument kann das Trennzeichen festgelegt werden
4-element Array{SubString{String},1}:
 "Stahl"
 "Beton"
 "Ziegel"
 "Holz"
```

`join` verbindet ein Array aus Strings zu einem einzelnen String:

```
julia> s1 = split("julia ist großartig")
3-element Array{SubString{String},1}:
 "julia"
 "ist"
 "großartig"

julia> sort!(s1)
3-element Array{SubString{String},1}:
 "großartig"
 "ist"
 "julia"

julia> s2 = join(s1,' ')
"großartig ist julia"
```

8.1.5 Arrays höherer Ordnung

In vielen numerischen Aufgaben sind Arrays zweiter oder höherer Ordnung hilfreich. In Julia wird ein Array zweiter Ordnung auch als *Matrix* bezeichnet. Die Eingabe erfolgt wie bei Arrays erster Ordnung in eckigen Klammern, allerdings werden die Elemente zeilenweise mit Leerzeichen zwischen den Elementen einer Zeile und Semikola zwischen den Zeilen eingegeben. Alternativ kann auch ein Zeilenumbruch anstatt der Semikola eingefügt werden.

```
a = [1 2; 3 4]
b = [4 2
     3 1]
```

Eine `m×n`-Matrix hat `m` Zeilen und `n` Spalten. Mit der Syntax `matrix[m,n]` kann auf ein Element der Matrix zugegriffen werden. Ebenso kann auf Teile der Matrix zugegriffen werden. Ein Doppelpunkt `:` ohne Indizes bezeichnet dabei die gesamte Zeile oder Spalte.

```
julia> m = rand(3,6)
3×6 Array{Float64,2}:
 0.960264  0.388763  0.940487  0.716114  0.405838  0.0563248
 0.134675  0.709269  0.863312  0.51876   0.148659  0.908457
 0.426839  0.2905    0.263677  0.894448  0.279718  0.933875

julia> m[2,3]  # Zweite Zeile, dritte Spalte
0.8633116011978317

julia> m[1:2,6]  # Erste und zweite Zeile, sechste Spalte
2-element Array{Float64,1}:
 0.05632478777006744
 0.9084568696124204

julia> m[3,:]  # Dritte Zeile, alle Spalten
6-element Array{Float64,1}:
 0.42683890910160804
 0.29049973854302946
 0.26367679965109203
 0.8944479761685689
 0.27971803379010973
 0.9338747647773251
```

Die Funktion `rand` erzeugt Zufallszahlen in unterschiedlicher Art. Ohne Argumente wird eine einzelne Zufahlszahl zwischen 0 und 1 zurückgegeben. Es können allerdings auch mehrere Int-Zahlen als Argumente übergeben werden, um ein Array mit Zufallszahlen zu erzeugen. Dabei gibt die erste Zahl die Anzahl der Elemente in erster Ordnung, die zweite Zahl die Anzahl der Elemente in zweiter Ordnung, usw. an. `rand(3, 2, 4)` erzeugt also ein `3×2×4`-Array dritter Ordnung, voller Zufallszahlen zwischen 0 und 1.

```
julia> rand()
0.44350398271657276

julia> rand(2)
2-element Array{Float64,1}:
 0.8219751947695386
 0.7991383464677408

julia> rand(2,2)
2×2 Array{Float64,2}:
 0.597576   0.436177
 0.0422274  0.345135

julia> rand(3,2,4)
3×2×4 Array{Float64,3}:
[:, :, 1] =
 0.745088  0.623403
 0.321313  0.125938
 0.105571  0.6688

[:, :, 2] =
 0.908163  0.268319
 0.796243  0.0584103
 0.141615  0.725353

[:, :, 3] =
 0.438505   0.700797
 0.663393   0.324569
 0.0389957  0.667119

[:, :, 4] =
 0.349661  0.133909
 0.863501  0.469648
 0.41352   0.822012
```

Mit `rand` können aber nicht nur Zufallszahlen zwischen `0` und `1` erzeugt werden. Die zufällige Auswahl eines

Elements aus einem `Array` ist ebenso möglich. Nachdem an erster Stelle der Argumente das Array stehen muss, können wieder beliebig viele `Int`-Zahlen als Anzahl der Elemente je Ordnung übergeben werden.

```
julia> a = ["Holz","Stahl","Ziegel","Beton"];

julia> rand(a)
"Ziegel"

julia> rand(a,3)
3-element Array{String,1}:
 "Ziegel"
 "Holz"
 "Holz"

julia> rand(a,2,3)
2×3 Array{String,2}:
 "Holz"  "Beton"  "Holz"
 "Holz"  "Beton"  "Holz"
```

Die Funktion `zero` kann nicht nur auf Zahlen, sondern auch auf Arrays mit Zahlen angewendet werden. In diesem Zusammenhang sind auch die Funktionen `one`, `ones` und `zeros` interessant. Achtung: Bei `one` muss das Argument eine quadratische Matrix sein.

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

julia> zero(a)  # Erzeugt ein Array voller Nullen, gleich groß wie das Argument
2×2 Array{Int64,2}:
 0  0
 0  0

julia> one(a)  # Erzeugt eine Einheitsmatrix gleich groß wie das Argument
2×2 Array{Int64,2}:
 1  0
 0  1

julia> zeros(2,3)  # Erzeugt ein 2x3-Array voller Nullen
2×3 Array{Float64,2}:
 0.0  0.0  0.0
 0.0  0.0  0.0

julia> ones(3,2)  # Erzeugt ein 3x2-Array voller Einsen
3×2 Array{Float64,2}:
 1.0  1.0
 1.0  1.0
 1.0  1.0
```

`similar` erzeugt ein Array vom gleichen Datentyp wie das übergebene Argument.

```
julia> a = [1 2; 3 4];

julia> b = ["a" "b"; "c" "d"];

julia> similar(a)
2×2 Array{Int64,2}:
 4603841408  4666090400
 4666090336  4683597808

julia> similar(b)
2×2 Array{String,2}:
 #undef  #undef
 #undef  #undef
```

8.1.6 Broadcasting

Viele Operationen in Julia können elementweise auf Arrays angewendet werden. Diese Vorgehensweise wird auch *Broadcasting* genannt. Dazu wird bei Operatoren wie `+` und `^` ein `.` vorangestellt oder bei Funktionen ein `.` hinter dem Funktionsnamen angehängt:

```
julia> a = [1 2 3; 3 4 5]
```

```
2×3 Array{Int64,2}:
 1  2  3
 3  4  5

julia> a .^2
2×3 Array{Int64,2}:
 1   4   9
 9  16  25

julia> a .+ 4
2×3 Array{Int64,2}:
 5  6  7
 7  8  9

julia> b = ["Tom" "Jerry"; "Michael" "John"];

julia> length.(b)
2×2 Array{Int64,2}:
 3  5
 7  4
```

Durch Anwenden von Vergleichsoperatoren oder Funktionen die einen Wert vom Typ `Bool` zurückggeben, mittels *Broadcasting* auf alle Element eines Arrays, wird ein Bit-Array (Besteht nur aus `0` und `1`, für `false` und `true`) erzeugt. Mittels eines solchen Arrays ist der Zugriff auf Teile eines anderen Arrays möglich. Dabei werden nur die Teile des Arrays ausgewählt, bei deren Position im Bit-Array der Wert `1` steht.

```
julia> primes .>= 10
6-element BitArray{1}:
 0
 0
 0
 0
 1
 1

julia> primes[primes .>= 10]
2-element Array{Int64,1}:
 11
 13

julia> occursin.("o",b) # String beinhaltet ein o
2×2 BitArray{2}:
 1  0
 0  1

julia> b[occursin.("o",b)]
2-element Array{String,1}:
 "Tom"
 "John"
```

8.2 Range

Eine `Range` ist eine Abfolge von `Int`-Zahlen. Durch Angabe der ersten und letzten Zahl der Sequenz, getrennt durch einen Doppelpunkt, kann eine `Range` erstellt werden. Dabei ist die Schrittweite standardmäßig mit `1` definiert. Durch Angabe einer dritten `Int`-Zahl kann auch die Schrittweite festgelegt werden. Die Schrittweite kann für eine absteigende `Range` auch negativ sein, dann muss allerdings der größere Wert an erster Stelle stehen.

```
julia> 1:10
1:10

julia> 0:2:10
0:2:10

julia> 10:-2:0  # Negative Schrittweiten erzeugen absteigende Ranges
10:-2:0
```

`rand` kann auch auf eine `Range` angewendet werden.

```
julia> rand(1:10)
1
```

```
julia> print(sort(rand(1:45,6)))  # Die Lotto-Zahlen nächster Woche?
[5, 9, 15, 24, 33, 37]
```

Eine `Range` kann vor allem für Schleifen hilfreich sein. Mithilfe der Funktion `collect` können damit aber auch schnell Arrays erzeugt werden.

```
julia> print(collect(1:5))
[1, 2, 3, 4, 5]
julia> print(collect(6:-2:0))
[6, 4, 2, 0]
```

Julia hat verschiedene Arten von Sammlungen implementiert. Viele dieser Sammlungen können mithilfe von `collect` in Arrays umgewandelt werden.

`Range`s können auch dazu verwendet werden um auf Teile eines Arrays zuzugreifen.

```
julia> primes[1:2:end]
3-element Array{Int64,1}:
  2
  5
 11

julia> primes[end:-1:1]
6-element Array{Int64,1}:
 13
 11
  7
  5
  3
  2
```

8.3 Tuple

Der Datentyp `Tuple` ist sehr ähnlich zum Array erster Ordnung. Der wichtigste Unterschied ist, dass `Tuple` nicht veränderbar sind. Am einfachsten wird ein Tuple durch die Eingabe von mehreren durch Kommas getrennte Werte erzeugt. Übersichtlicher ist es aber, die Werte mit runden Klammern zu umgeben:

```
julia> t1 = 1, 2, 3, 4
(1, 2, 3, 4)

julia> t2 = ('a', 'b', 'c')
('a', 'b', 'c')
```

Wie beim Array können auch unterschiedliche Datentypen in ein Tuple geschrieben werden. Die Datentypen werden dabei im Datentypen des Tuples geschrieben:

```
julia> t = (1, 3.1415, ”foo”, ('c',))
(1, 3.1415, ”foo”, ('c',))

julia> typeof(t)
Tuple{Int64,Float64,String,Tuple{Char}}
```

Tuple mit nur einem Element müssen mit einem Komma abgeschlossen werden. Ein leeres Tuple kann mittels `tuple()` erzeugt werden.

```
julia> t1 = ('!',)  # Tuple
('!',)

julia> c = ('!')  # Character
'!': ASCII/Unicode U+0021 (category Po: Punctuation, other)

julia> t2 = tuple()
()
```

Wie bei Arrays kann mittels eckiger Klammern auf einzelne oder mehrere Elemente eines Tuples zugegriffen werden. Da ein Tuple allerdings nicht veränderbar ist, gibt Julia beim Versuch der Neudefinition eines Wertes einen *MethodError* zurück.

```
julia> t = (2, 3, 5, 7, 11, 13, 17, 19)
(2, 3, 5, 7, 11, 13, 17, 19)
```

```
julia> print(t[3])
5
julia> print(t[4:6])
(7, 11, 13)
julia> print(t[5:end])
(11, 13, 17, 19)
julia> t[4] = 23
Error: MethodError: no method matching setindex!(::NTuple{8,Int64}, ::Int64, ::Int64)
```

Manchmal ist es notwendig, ein `Tuple` in ein `Array` umzuwandeln, z.B. um Werte ändern zu können. Mittels `collect` kann ein `Tuple` einfach in ein `Array` umgewandelt werden:

```
julia> print(collect(("spam", "foo", "bar")))
["spam", "foo", "bar"]
```

8.4 Dictionaries

Ein `Dict` (Abkürzung für *Dictionary*) ist ebenfalls sehr ähnlich zum Array erster Ordnung. Während sich beim Array die Indizes auf Int-Zahlen beschränken, können die Indizes eines *Dictionaries* nahezu jeder Datentyp sein. Die Indizes eines Dictionaries heißen *Keys* und die Werte werden *Values* genannt. Jeder Wert ist dabei einem Key zugewiesen. Mit einem Dictionary kann z.B. ein Wörterbuch erstellt werden, in dem jeder Key ein deutsches Wort ist, dem das passende spanische Wort zugeordnet ist. Mit `Dict()` wird ein Dictionary erstellt:

```
julia> de2sp = Dict("eins" => "uno", "zwei" => "dos", "drei" => "tres")
Dict{String,String} with 3 entries:
  "eins" => "uno"
  "drei" => "tres"
  "zwei" => "dos"
```

Ein weiterer wichtiger Unterschied zu Arrays ist, dass Dictionaries nicht geordnet sind. Es kann daher sein, dass die Elemente nicht in der eingegebenen Reihenfolge angezeigt werden.

Die Syntax für die Eingabe von Key-Value-Paaren ist dabei folgende:

- ein Key-Value-Paar besteht aus dem Key gefolgt von `=>` und dem zugehörigen Wert;
- die einzelnen Einträge werden durch ein Komma getrennt.

Durch eckige Klammern und Eingabe eines Index kann auf die Werte zugegriffen werden. Falls der eingegebene Index nicht im Dictionary vorhanden ist, gibt Julia einen *KeyError* zurück.

```
julia> de2sp["zwei"]
"dos"

julia> de2sp["vier"]
Error: KeyError: key "vier" not found
```

Das sollte geändert werden. Neue Indizes können auf folgende Weise hinzugefügt werden:

```
julia> de2sp["vier"] = "cuatro"
"cuatro"

julia> print(de2sp)
Dict("eins" => "uno","drei" => "tres","zwei" => "dos","vier" => "cuatro")
```

Auf diese Weise können vorhandene Werte neu definiert werden. Die Funktionen `keys` und `values` geben Sammlungen mit den Keys oder Values eines Dictionaries zurück.

```
julia> print(keys(de2sp))
["eins", "drei", "zwei", "vier"]
julia> print(values(de2sp))
["uno", "tres", "dos", "cuatro"]
```

Diese Sammlungen haben die Typen `Base.KeySet` und `Base.ValueIterator`, auf die wir hier nicht genauer eingehen. Es sei jedoch festgehalten, dass diese Sammlungen andere Eigenschaften als Arrays haben und daher manche Operationen damit nicht möglich sind. `collect` bietet auch hier die Möglichkeit, diese Sammlungen in `Arrays` umzuwandeln:

```
julia> print(collect(keys(de2sp)))
["eins", "drei", "zwei", "vier"]
```

8.5 Aufgaben

8.5.1 Drei Arrays zusammenhängen

Schreibe eine Funktion `connect_three`, die drei Arrays als Argumente nimmt und diese Arrays zu einem Array zusammenfügt. Die Funktion muss nur für Arrays mit gleichem Datentypen funktionieren (z.B. dreimal Array{String,1} oder dreimal Array{Any,1}).

Beispiele:

```
julia> a = [1, 1, 2];

julia> b = [3, 5];

julia> c = [8, 13];

julia> print(connect_three(a,b,c))
[1, 1, 2, 3, 5, 8, 13]
```

8.5.2 Innere Werte eines Arrays

Schreibe eine Funktion `interior`, die ein Array als Argument nimmt und ein neues Array zurückgibt, das alle Werte bis auf den ersten und den letzten enthält.

Beispiele:

```
julia> nums = [0,1,2,3,4];

julia> res1 = interior(nums)
3-element Array{Int64,1}:
 1
 2
 3

julia> res2 = interior(res1)
1-element Array{Int64,1}:
 2
```

8.5.3 Ist das Array sortiert?

Schreibe eine Funktion `is_sort`, die ein Array als Argument nimmt, überprüft, ob dieses Array sortiert ist, und dementsprechend `true` oder `false` zurückgibt.

Beispiele:

```
julia> unsorted_nums = [3,2,5,1];

julia> sorted_nums = sort(unsorted_nums);

julia> is_sort(unsorted_nums)
false

julia> is_sort(sorted_nums)
true
```

8.5.4 Anagram-Test

Zwei Wörter sind Anagramme, wenn die Buchstaben eines Wortes durch Umordnen das zweite Wort ergeben. Schreibe eine Funktion `is_anagram`, die zwei Strings als Argumente nimmt, überprüft, ob diese Strings Anagramme sind, und dementsprechend `true` oder `false` zurückgibt. Groß- und Kleinschreibung soll dabei nicht berücksichtigt werden (`Stein` ist gleichwertig zu `stein`).

Beispiele:

8.5.5 N-tes Element eines Dict ausgeben

Da Dictionaries nicht geordnet sind, bietet Julia keine Möglichkeit auf das `n`-te Element eines Dictionaries zuzugreifen. Schreibe eine Funktion `dict_index(d,n)`, die ein Dict `d` und eine Int-Zahl `n` als Argumente nimmt, das Dictionary nach den Keys sortiert und den `n`-ten Value in dieser Reihenfolge zurückgibt.

Beispiele:

```
julia> d = Dict("a"=>1,"b"=>2, "c"=>3, "d"=>4)
Dict{String,Int64} with 4 entries:
  "c" => 3
  "b" => 2
  "a" => 1
  "d" => 4

julia> dict_index(d, 2)
2
```

8.5.6 Tuples zusammenfügen

Da Tuple nicht veränderlich sind, funktioniert das Zusammenfügen zweier Tuple standardmäßig nicht. Schreibe eine Funktion `tuple_append`, die zwei Tuples als Argumente nimmt, das zweite an das erste anhängt und das zusammengefügte Tuple als Return Value zurückgibt.

Beispiele:

```
julia> t1 = ('a','b')
('a', 'b')

julia> t2 = ('c','d','e')
('c', 'd', 'e')

julia> tuple_append(t1,t2)
('a', 'b', 'c', 'd', 'e')
```

9 Exceptions

Ein Programm zu schreiben, ist eine komplexe Aufgabe, bei der es häufig dazu kommt, dass das Programm nicht das macht, was es geplanterweise tun soll und mit einem Fehler abbricht.

Grob können diese Fehler in *Syntax Errors*, *Logical Errors* und *Runtime Errors* eingeteilt werden.

9.1 Syntax Errors

Ein *Syntax Error* tritt in den meisten Fällen bereits vor der eigentlichen Ausführung des Programms auf und wird durch einen falsch geschriebenen Befehl verursacht. Dies betrifft hauptsächlich Keywords der Programmiersprache (`if`, `else`, `function`, ...) und ungültige Variablennamen.

```
julia> 42number = "meaning of life"
Error: syntax: "42" is not a valid function argument name
```

Nach Auftreten des Fehlers wird das Programm unterbrochen. Die Fehlermeldung wird mit `Error:` eingeleitet und enthält den Hinweis `syntax`. Üblicherweise gibt Julia Hinweise darauf, was das Problem verursacht hat, es lohnt sich also immer, Fehlermeldungen zu lesen.

9.2 Logical Errors

Bei *Logical Errors* handelt es sich um Fehler in der Programmlogik (z.B. eine falsche Formel oder falsche Annahme).

In der folgenden Funktion ist ein logischer Fehler. Eigentlich sollte die Funktion x^n berechnen, gibt aber x^{n+1} zurück.

```
function pow(x::Number,n::Int)
    x_pow = x
    for i in 1:n
        x_pow *= x
    end
    return x_pow
end
```

```
julia> pow(3,2)  # Sollte 9 ergeben
27
```

Das Problem mit logischen Fehler ist, dass das Programm ohne Fehlermeldung läuft, dass Ergebnis aber falsch ist.

9.3 Runtime Errors

Runtime Errors sind Fehler, die während der Laufzeit des Programms auftreten. In Julia gibt es eine Vielzahl solcher Fehlern, die immer dann ausgelöst werden, wenn etwas Unvorhergesehenes passiert. In weiterer Folge werden nur die häufigsten behandelt, eine vollständige Liste befindet sich in der Dokumentation unter Manual/Control Flow/Exception Handling.

`BoundsError`

Zugriff auf einen nicht besetzten Index in einem Array.

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

```
Error: BoundsError: attempt to access 3-element Array{Int64,1} at index [4]
```

`DomainError`

Das Argument ist nicht innerhalb des (mathematisch) definierten Bereichs der Funktion.

```
sqrt(-1)
```

```
Error: DomainError with -1.0:
sqrt will only return a complex result if called with a complex argument. T
ry sqrt(Complex(x)).
```

`InexactError`

Eine Operation kann nicht exakt ausgeführt werden.

```
convert(Int,12.1)
```

```
Error: InexactError: Int64(12.1)
```

Um in diesem Fall von der `Float64`-Zahl `12.1` zur `Int`-Zahl `12` zu gelangen, muss vorher gerundet werden.

`MethodError`

Es existiert keine Methode, die den angegebenen Argumenten entspricht

```
abs(”Ein String”)
```

```
Error: MethodError: no method matching abs(::String)
Closest candidates are:
  abs(!Matched::Pkg.Resolve.MaxSum.FieldValues.FieldValue) at /Users/sabae/
buildbot/worker/package_macos64/build/usr/share/julia/stdlib/v1.2/Pkg/src/r
esolve/FieldValues.jl:67
  abs(!Matched::Pkg.Resolve.VersionWeights.VersionWeight) at /Users/sabae/b
uildbot/worker/package_macos64/build/usr/share/julia/stdlib/v1.2/Pkg/src/re
solve/VersionWeights.jl:40
  abs(!Matched::Missing) at missing.jl:79
  ...
```

`UndefVarError`

Die Variable ist nicht definiert.

```
nicht_definiert
```

```
Error: UndefVarError: nicht_definiert not defined
```

9.3.1 Stacktrace

Der sogenannte *Stacktrace* gibt Auskunft darüber, welche Funktionsaufrufe zu dem Fehler geführt haben. Er wird immer am Ende der Fehlermeldung ausgegeben.

Im folgenden Beispiel wird nach mehreren verschachtelten Aufrufen auf einen Index außerhalb eines Arrays zugegriffen.

```
function level03(x)
    # Zugriff ist ok
    x[1]
    # Zugriff ist nicht ok
    x[length(x)+1]
end
level02(x) = level03(x)
level01(x) = level02(x)

level01([1,2,3])
```

Der *Stacktrace* zu diesem Aufruf sieht wie folgt aus:

```
Stacktrace:
 [1] getindex at array.jl:728 [inlined]
 [2] level03(::Array{Int64,1}) at none:5
 [3] level02(::Array{Int64,1}) at none:1
 [4] level01(::Array{Int64,1}) at none:3
```

Bei `[1]` wird der auslösende Funktionsaufruf angezeigt, der Zugriff auf einen Arrayindex. Jede weitere Zeile entspricht dem nächsthöheren Aufruf. Wird der *Stacktrace* von `[4]` nach `[2]` gelesen, ist erkennbar, dass `level01` `level02` aufruft und `level02` `level03`.

Der *Stacktrace* ist sehr hilfreich, wenn es darum geht zu reproduzieren, warum und wo ein Fehler aufgetreten ist.

10 Lineare Algebra

Neben der Unterstützung von multi-dimensionalen Arrays (Vektoren, Matrizen, Tensoren) bietet Julia auch eine Implementierung von nützlichen Funktionen der linearen Algebra. Das Lineare-Algebra-Paket ist standardmäßig in Julia enthalten und kann wie alle anderen Pakete mit `using` geladen werden.

```
using LinearAlgebra
```

Details zu dem Paket sind unter Standard Library/Linear Algebra in der Dokumentation zu finden.

10.1 Lineare Gleichungssysteme

Eine übliche Problemstellung der linearen Algebra ist das Lösen von Gleichungssystemen wie dem folgenden:

$$x - y = 22$$

$$2x + 3y = 11$$

Um das System zu lösen, müssen zuerst die Koeffizientenmatrix und der Vektor auf der rechten Seite definiert werden.

```
A = [1 -1
     2  3]
c = [22, 11]
```

Um Informationen über die Lösbarkeit des System zu erhalten, wird die Determinante mit `det` ermittelt.

```
det(A)
```

```
5.0
```

Offensichtlich hat das Gleichungssystem eine eindeutige Lösung. Im nächsten Schritt muss **A** invertiert werden.

```
A_inv = inv(A)
```

```
2×2 Array{Float64,2}:
  0.6  0.2
 -0.4  0.2
```

Der Lösungsvektor **b** ergibt sich dann zu:

```
b = A_inv*c
```

```
2-element Array{Float64,1}:
 15.400000000000002
 -6.6000000000000005
```

Die Kontrolle kann einfach durch Ausmultiplizieren erfolgen.

```
A*b
```

```
2-element Array{Float64,1}:
 22.000000000000004
 11.000000000000004
```

Da dieser Ablauf sehr häufig vorkommt und sich abhängig von **A** gut optimieren lässt, bietet Julia mit dem \-Operator eine Kurzschreibweise, bei der das zusätzliche Invertieren entfällt.

```
A\c
```

```
2-element Array{Float64,1}:
 15.4
 -6.6
```

10.2 Vektorfunktionen

Das Lineare-Algebra-Paket beinhaltet auch Funktionen zur Berechnung des Skalarprodukts `dot` oder `·` (`\cdot`+TAB-Taste), des Kreuzprodukts `cross` oder `×` (`\times`+TAB-Taste) und der Länge `norm` von Vektoren.

```
julia> [1,2,3] · [4,5,6]
32

julia> [1,2,3] × [4,5,6]
3-element Array{Int64,1}:
 -3
  6
 -3

julia> norm([1,2,3])
3.7416573867739413
```

10.3 Aufgaben

10.3.1 Lösbarkeit von linearen Gleichungssystemen

Es soll eine Funktion definiert werden, die `true` zurückgibt, falls ein lineares Gleichungssystem lösbar ist (nicht zwingend eindeutig), und `false` zurückgibt, falls das System nicht lösbar ist.

Beispiele:

```
julia> lösbar([1 -1
       2  3],[22, 11])
true

julia> lösbar([1 -1
       2  3
       9 -1],[22, 11, 3])
false

julia> lösbar([1 3
       2  8
```

```
        3 11],[9, -3, 6])
true

julia> lösbar([1 1
        2 2],[2, 3])
false
```

10.3.2 Löse das Gleichungssystem

Es soll das folgende Gleichungssystem in Matrixform gebracht und gelöst werden.

$$x - y + z = 22$$

$$2x + 3y = 11$$

$$10x - y/2 + 5z = 0$$

Für die Kontrolle muss die allgemeine Form der Kontrollfunktion verwendet werden.

```
@Aufgabe "10.3.2" begin
    # Code einfügen.
    # Hier, in der letzten Zeile, muss der Lösungsvektor berechnet werden
end
```

10.3.3 Eigenwerte und Eigenvektoren

Mit den Funktion `eigvals` und `eigvecs` können Eigenwerte bzw. Eigenvektoren berechnet werden. Es soll eine Funktion geschrieben werden, die aus Eigenwerten und Eigenvektoren, in dem Format wie sie von den beiden Funktionen für eine Matrix ausgegeben werden, wieder die Matrix rekonstruieren.

Beispiele:

```
julia> A = [1 -1 1; 2  3 0; 10 -0.5 5]; λ = eigvals(A); ϕ = eigvecs(A);

julia> rekonstruiere(λ,ϕ)
3×3 Array{Float64,2}:
  1.0  -1.0   1.0
  2.0   3.0  -1.72085e-15
 10.0  -0.5   5.0
```

10.3.4 Winkel zwischen den Vektoren

Es soll eine Funktion definiert werden, die den Winkel in Radiant zwischen zwei Vektoren beliebiger Dimension berechnet.

Beispiele:

```
julia> winkel([1,5],[3,7])
0.20749622643520305

julia> winkel([2,3,-1],[8,-1,2])
1.2090381087121735

julia> winkel([1,0,0],[1,0,0])
0.0
```

11 Kontrollstrukturen

Der Code in den bisher behandelten Beispiele setzte sich aus einer sich nicht-verzweigenden Verkettung von Befehlen zusammen. Für sehr rudimentäre Programme mag das ausreichen, in den meisten Fällen werden aber Verzweigungen in Form von `if`-Bedingungen und Schleifen benötigt.

11.1 Verzweigungen

Dieses Konstrukt erlaubt es, abhängig von einer *Bedingung* (die ausgewertet vom Typ `Bool` sein muss) einen Codeteil auszuführen oder zu überspringen. Die einfachste Form ist das `if`-Statement:

```
x = -1
if x < 0
    println("x ist negativ.")
end
```

```
x ist negativ.
```

In diesem Beispiel ist die *Bedingung* `x < 0`. Wie bereits im Abschnitt zu den Datentypen gezeigt, ergibt die Auswertung dieses Ausdrucks einen Wert vom Typ `Bool`:

```
julia> x < 0
true

julia> typeof(x < 0)
Bool
```

Die zweite Form ist die Kombination aus `if` und `else`. `else` bezeichnet den Zweig, der ausgeführt wird, falls die Bedingung nicht erfüllt ist.

```
x = 1
if x < 0
    println("x ist negativ.")
else
    println("x ist positiv.")
end
```

```
x ist positiv.
```

Da `x < 0` ausgewertet `false` ergibt, wird der `if`-Zweig übersprungen und der `else`-Zweig ausgeführt.

Die dritte Form ist im Grunde eine vereinfachte Schreibweise für Kombinationen von Form eins und Form zwei. Der folgende Code

```
x = 9
if x < 0
    println("x ist negativ.")
else
    if x <= 10
        println("x ist eine Zahl von 0 bis 10.")
    else
        println("x ist größer als 10.")
    end
end
```

```
x ist eine Zahl von 0 bis 10.
```

kann einfacher und übersichtlicher durch `elseif` geschrieben werden.

```
x = 9
if x < 0
    println("x ist negativ.")
elseif x <= 10
    println("x ist eine Zahl von 0 bis 10.")
else
    println("x ist größer als 10.")
end
```

```
x ist eine Zahl von 0 bis 10.
```

11.2 Schleifen

Computer sind besonders gut geeignet, um repetitive Aufgaben zu lösen. Solche Aufgaben werden durch Schleifen definiert. Julia bietet zwei Arten von Schleifen: `for`- und `while`-Schleife.

Die `while`-Schleife hat eine gewisse Ähnlichkeit zur `if`-Bedingung. Eine Aufgabe in einer solchen Schleife wird immer wieder ausgeführt, solange eine *Bedingung* erfüllt ist.

Im folgenden Beispiel wird jede Zahl von 0 bis 10 ausgegeben.

```
function zähle_bis(n)
    i = 1
    while i <= n
        println("Bin bei Zahl $i")
        i += 1
    end
end
zähle_bis(10)
```

```
Bin bei Zahl 1
Bin bei Zahl 2
Bin bei Zahl 3
Bin bei Zahl 4
Bin bei Zahl 5
Bin bei Zahl 6
Bin bei Zahl 7
Bin bei Zahl 8
Bin bei Zahl 9
Bin bei Zahl 10
```

Der Code wird folgendermaßen durchschritten:

1. Initialisiere `i` mit `1`
2. Falls die Bedingung `i <= 10` erfüllt ist, gehe zu (3), ansonsten gehe zu (6)
3. Ausgabe von `"Bin bei Zahl $i"`
4. Erhöhe `i` um `1`
5. Gehe zurück zu (2)
6. Ende

Dieser Ablauf wird Schleife genannt, weil (5) wieder zurück zu (2) springt. Bei diesem Code ist Vorsicht geboten: Vergisst man die Zeile `i += 1`, ergibt die Bedingung `i <= 10` nie `false`. Das bedeutet, die Schleife läuft endlos (Endlosschleife). In der REPL oder in Juno kann eine Endlosschleife durch die Tastenkombination `Ctrl+C` abgebrochen werden.

Beim obigen Beispiel handelt es sich um eine Zählschleife. Dieses Konstrukt ist so häufig, dass es einen eigenen Schleifentyp dafür gibt: Die `for`-Schleife. Der Code in der `for`-Schleife wird für jedes Element in einer ihr übergebenen Menge ausgeführt. Das vorherige Beispiel vereinfacht sich daher zu:

```
function zähle_bis(n)
    for i in 1:n
        println("Bin bei Zahl $i")
    end
end
zähle_bis(10)
```

```
Bin bei Zahl 1
Bin bei Zahl 2
Bin bei Zahl 3
Bin bei Zahl 4
Bin bei Zahl 5
Bin bei Zahl 6
Bin bei Zahl 7
Bin bei Zahl 8
Bin bei Zahl 9
Bin bei Zahl 10
```

Anstelle von `1:n` kann jede Art von Liste angegeben werden.

```
for buchstabe in ["A", "B", "C", "D"]
    println("Bin bei Buchstabe $buchstabe")
end
```

```
Bin bei Buchstabe A
Bin bei Buchstabe B
Bin bei Buchstabe C
Bin bei Buchstabe D
```

11.2.1 break und continue

Eine Schleifeniteration kann jederzeit durch den Aufruf von `break` abgebrochen und den Aufruf von `continue` übersprungen werden.

Im folgenden Beispiel werden von einem `String` nur die Vokale gesammelt. Falls ein `!` gefunden wird, werden die gesammelten Vokale ausgegeben und die Schleife beendet.

```
function sammle_vokale(text)
    vokale_sammlung = ""
    for buchstabe in text
        if buchstabe == '!'
            println(vokale_sammlung)
            break  # Beende die Schleife
        end
        if !(buchstabe in ('a','e','i','o','u'))
            continue  # Zurück zum Beginn der Schleife
        end
        vokale_sammlung *= buchstabe
    end
end
sammle_vokale("Finde die Vokale in diesem Satz. Gehe aber nicht weiter als bis zum !. Das sollst du nicht mehr ansehen")
```

```
ieieoaeiieeaeeaeieieaiu
```

`break` wird oft auch in Kombination mit `while`-Schleifen verwendet, die endlos laufen. Wenn beispielsweise in der Schleife auf eine Dateneingabe gewartet wird oder die Abbruchbedingung sehr komplex ist. In diesem Fall kann eine Endlosschleife absichtlich durch Verwenden der Bedingung `true` konstruiert werden.

11.2.2 Performance Optimierung Julia

Bei den folgenden Aufgaben ist es wichtig, möglichst effizienten Code zu schreiben. Optimierung ist ein sehr komplexes Thema, es gibt in Julia aber ein paar grundlegende Punkte, die eine Effizienzsteigerung bringen.

Um eine Auskunft über die Laufzeit eines Codes zu erhalten (Benchmark-Test), biete Julia das `@time`-Macro.

Die folgende Funktion berechnet `n`-mal die Inverse einer Matrix `A` und ist nur als Beispielfunktion für die Verwendung von `@time` gedacht.

```
function time_dummy(n::Int)
    A = [3 1 0
         0 1 0
         3 3 3]
    for i in 1:n
        inv(A)
    end
end
```

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

```
julia> @time time_dummy(100000)
  0.114778 seconds (674.81 k allocations: 213.099 MiB, 16.50% gc time)

julia> @time time_dummy(100000)
  0.078157 seconds (600.00 k allocations: 209.046 MiB, 18.76% gc time)

julia> @time time_dummy(100000)
  0.077027 seconds (600.00 k allocations: 209.046 MiB, 15.00% gc time)
```

Das `@time`-Macro gibt die erforderliche Zeit und Angaben über das Speichermanagement aus. Der erste Aufruf der neu geschriebenen Funktion `time_dummy` braucht wesentlich länger als die nächsten beiden. Das liegt daran, dass Julia beim ersten Aufruf die Funktion für die angegebenen Argumente kompiliert.

Dieser Vorgang ist sehr wichtig, um eine performante Ausführung zu ermöglichen, und wird stark durch die sogenannte *Typenstabilität* beeinflusst.

Typenstabilität Stabilität bedeutet hier, dass sich der Datentyp einer Variable mit der Laufzeit nicht ändert. Eine mögliche Fehlerquelle dafür ist im folgendem Beispiel zu sehen[14].

[14] http://www.johnmyleswhite.com/notebook/2013/12/06/writing-type-stable-code-in-julia/

```
function summe_sin_A(n::Int)
    ∑ = 0
    for i in 1:n
        ∑ += sin(n)
    end
    return ∑
end

function summe_sin_B(n::Int)
    ∑ = 0.0
    for i in 1:n
        ∑ += sin(n)
    end
    return ∑
end
```

Der einzige Unterschied der Funktionen A und B befindet sich in der ersten Zeile bei der Definition von `Σ` (`\sum`+`TAB`-Taste). Im Fall von A wird mit `Int` im Fall von B mit `Float64` initialisiert.

Um die Funktionen zu benchmarken, werden beide in Schleifen ausgeführt.

```
function benchmark_summe_A()
    for _ in 1:100
        summe_sin_A(100000)
    end
end

function benchmark_summe_B()
    for _ in 1:100
        summe_sin_B(100000)
    end
end
```

Die Laufzeit der beiden Fälle ergibt sich zu:

```
julia> benchmark_summe_A(); benchmark_summe_B(); # Erster Aufruf zum Kompilieren

julia> @time benchmark_summe_A()
  0.010950 seconds (4 allocations: 160 bytes)

julia> @time benchmark_summe_B()
  0.000002 seconds (4 allocations: 160 bytes)
```

Der große Unterschied zwischen den beiden Funktionen resultiert aus dem Rückgabewert der Funktion `sin`:

```
julia> typeof(sin(3))
Float64
```

`sin` gibt `Float64` zurück. Das bedeutet bei Fall A muss `Σ` von `Int` in `Float64` konvertiert werden, was immens viel Zeit benötigt.

Ein weiterer häufig gemachter Fehler tritt bei Funktionen auf, die für gleiche Datentypen der Argumente verschiedene Datentypen der Ergebnisse zurückgeben. Beispielsweise gibt die folgende Definition der Rampen-Funktion inkonsistente Datentypen zurück:

```
function ramp(x)
    if x < 0
        return 0
    else
        return x
    end
end
```

```
julia> ramp(3.0)
3.0

julia> ramp(-3.0)
0

julia> ramp(3)
3

julia> ramp(-3)
0
```

Eine Möglichkeit wäre, die Funktion für die Typen `Float64` und `Int` zu spezialisieren oder besser die `zero`-Funktion zu verwenden:

```
function ramp(x)
    if x < 0
        return zero(x)
    else
        return x
    end
end
```

```
julia> ramp(3.0)
3.0

julia> ramp(-3.0)
0.0

julia> ramp(3)
3

julia> ramp(-3)
0
```

Vermeiden von globalen Variablen Eine globale Variable ist beispielsweise `x` im folgenden Code:

```
x = "Ich bin global!"

function test_global()
    println(x)
end
test_global()
```

```
Ich bin global!
```

`x` ist innerhalb der Funktion `test_global` verwendbar, obwohl die Variable außerhalb definiert wurde. Der Nachteil von globalen Variablen, aus Sicht der Optimierung, ist, dass sie von mehreren Stellen aus verändert werden können. Das führt dazu, dass Julia die Funktionen, die die Variablen verwenden, wieder nicht exakt für den Typ der globalen Variable spezialisieren kann, da sich dieser Ändern kann:

```
x = 100
test_global()
```

```
100
```

`test_global` gibt jetzt den neuen Wert von `x` aus. In diesem Fall hat sich nicht nur der Wert, sondern auch der Typ von `String` zu `Int` geändert.

In manchen Fällen ist es sinnvoll, globale Variablen zu verwenden, dann sollten diese mit dem Keyword `const` als Konstanten definiert werden (Konstanten werden üblicherweise in Blockbuchstaben geschrieben).

```
const GRAV_BESCHL = 9.81 # m/s²
function Kraft(masse) # masse in kg
    masse*GRAV_BESCHL
end
Kraft(8.0)
```

```
78.48
```

Eine Konstante darf sich nicht ändern; beim Versuch eine Konstante zu ändern, gibt Julia einen Error zurück. Damit ist der Code wieder typenstabil.

```
GRAV_BESCHL = "Neuer Wert"
```

```
Error: invalid redefinition of constant GRAV_BESCHL
```

11.3 Aufgaben

11.3.1 Gerade oder Ungerade

Schreibe ein Funktion, die „Gerade“ oder „Ungerade“ für eine Zahl mit `println` ausgibt.

Beispiele:

```
julia> gerade_ungerade(11)
Ungerade

julia> gerade_ungerade(218)
Gerade
```

11.3.2 Skript Flächeninhalt

Es soll ein **Skript** zur Berechnung von Flächeninhalt und Umfang eines Kreises erstellt werden, das BenutzerInnen erlaubt, mehrmals Eingaben zu machen. Die Ausgabewerte sollen auf zwei Stellen gerundet werden. Das Skript soll erst abgebrochen werde„ wenn eine leere Eingabe gemacht wird. Wird eine falsche Eingabe gemacht, muss „Falsche Eingabe“ ausgegeben werden.

Um eine Eingabe von der Tastatur einzulesen, kann die folgende Funktion verwendet werden:

```
"""
    wert_einlesen(text)

Lies eine Texteingabe vom Benutzer ein, gib als Aufforderung `text` aus und gib den eingegebenen Wert
als String zurück.
"""
function wert_einlesen(text::AbstractString)
    # Aufforderung ausgeben
    print(text)
    print(": ")
    # Eine Zeile von der Benutzereingabe einlesen
    return readline()
end
```

Die Ausgabe nach Ausführung des Skripts mit `julia kreis.jl` muss exakt mit dem folgenden Text übereinstimmen.

```
Radius eingeben: 3
A = 28.27
U = 18.85

Radius eingeben: abc
Falsche Eingabe

Radius eingeben: 1.2
A = 4.52
U = 7.54

Radius eingeben:
```

Für die Kontrolle mit dem Kontrollprogramm muss die allgemeine Form der Eingabe verwendet werden. Zusätzlich muss die Schleife für die wiederholte Ausführung über eine Funktion gestartet werden. Die Funktion muss wie im folgenden Beispiel am Ende des Kontrollblocks stehen. Wichtig für die Kontrolle ist, dass alle Ausgaben exakt mit dem Beispiel oben übereinstimmen.

```
@Aufgabe "11.3.2" begin
    # ...
    # Platz für Zusatzfunktionen die eventuell verwendet werden
    # ...
    function run()
        # ...
        # Hauptfunktion die die Schleife beinhaltet
        # ...
        while
            # ...
        end
```

```
        # ...
    end
end
```

11.3.3 Berechne π

Die Zahl π kann über eine so genannte Monte-Carlo-Simulation angenähert werden. Dabei werden zufällig Punkte in einem Quadrat mit bekannter Seitenlänge platziert. Durch Einschreiben eines Kreise in das Quadrat und Ermitteln der Anzahl an Punkten, die im Kreis gelandet sind, kann die Fläche des Kreises relativ zur Fläche des Quadrates ermittelt werden. π wird dann einfach über die Kreisflächenformel berechnet.

Schreibe eine Funktion, die π mittels Monte-Carlo-Simulation approximiert. Das einziges Argument der Funktion ist die Anzahl an zufällig ermittelten Punkte. Um den Rechenaufwand zu reduzieren, soll nur ein Vierteilkreis in einem Quadrat berücksichtigt werden. Die ermittelte relative Fläche muss dann natürlich noch mit dem Faktor 4 vergrößert werden.

Beispiele (mit steigender Anzahl von Punkten steigt auch die Genauigkeit):

```
julia> π_mc(10)
4.0

julia> π_mc(100)
3.16

julia> π_mc(1_000_000)
3.144028

julia> π_mc(1_000_000_000)
3.141652184
```

In Julia kann zur besseren Lesbarkeit ein _ in Zahlen eingefügt werden. Im vorherigen Beispiel dient _ als Tausendertrennzeichen, hat aber keinen Einfluss auf die Zahl selbst (`100 == 1_00`).

11.3.4 Numerisch Integrieren

Schreibe eine Funktion, die das bestimmte Integral einer beliebigen 1D-Funktion, definiert durch x- und y-Werte, berechnet. Verwende dazu die Trapezregel[15].

Beispiele:

```
julia> x = collect(0:0.0001:1);

julia> ∫(x,sin.(x.^2))
0.3102683026238847

julia> ∫(x,cos.(x.^2))
0.9045242364978205

julia> ∫(x,x)
0.4999999999999999
```

Je feiner die Auflösung, desto genauer wird das Ergebnis.

```
julia> x = collect(0:0.1:2π);

julia> ∫(x,sin.(x))
0.003455020910590271

julia> x = collect(0:0.01:2π);

julia> ∫(x,sin.(x))
5.0730443500457005e-6

julia> x = collect(0:0.0001:2π);

julia> ∫(x,sin.(x))
3.6386392119709672e-9
```

[15] https://de.wikipedia.org/wiki/Trapezregel

11.3.5 Wie hoch ist die Wahrscheinlichkeit

Die Monte-Carlo-Simulation kann auch dazu verwendet werden, um Wahrscheinlichkeiten zu approximieren. Schreibe zwei Funktionen, welche die Wahrscheinlichkeit abschätzen, dass

1. mit einem idealen Würfel (1–6) 3× hintereinander 4 gewürfelt wird und
2. mit einem idealen Würfel (1–6) bei 3 Würfen mindestens ein 4er gewürfelt wird.

Die Idee hinter der Simulation ist, die Würfe mit einem Zufallszahlengenerator `rand(1:6)` zu simulieren und anschließend die positiven Ergebnisse in Relation zu den gesamten Würfen zu setzten.

Wie schon bei der Berechnung von π soll das einzige Argument bei den Funktionen die Anzahl an zufälligen Würfen sein. Die Nummern für die Kontrolle sind `11.3.5.1` und `11.3.5.2`.

Beispiele:

```
julia> P_est_1(10)
0.0

julia> P_est_1(100)
0.0

julia> P_est_1(1_000_000)
0.004568

julia> P_est_1(1_000_000_000)
0.004628781

julia> P_est_2(10)
0.5

julia> P_est_2(100)
0.43

julia> P_est_2(1_000_000)
0.42093

julia> P_est_2(1_000_000_000)
0.421269301
```

Die exakten Ergebnisse sind:

```
julia> 1/6^3 # Fall 1
0.004629629629629629

julia> 1/6+5/6*1/6+5/6*5/6*1/6 # Fall 2
0.42129629629629634
```

11.3.6 Split

Im Abschnitt zu Strings wurde die Funktion `split` vorgestellt, mit der Strings durch Trennen bei einem Zeichen, in ein String-Array umgewandelt werden können. Bei dieser Aufgabe soll die Funktion nachprogrammiert werden, natürlich ohne die Verwendung von `split`. Das Trennzeichen muss vom Typ `char` sein und im Gegensatz zur `split` Funktion darf das resultierende Array keine leeren Strings `""` enthalten. Achtung, auch die Sonderfälle

- kein Trennzeichen
- Trennzeichen am Anfang
- Trennzeichen am Ende und
- mehrere Trennzeichen nacheinander

müssen behandelt werden.

Beispiele:

```
julia> my_split("Das ist ein String",' ')
4-element Array{String,1}:
 "Das"
 "ist"
```

```
 "ein"
 "String"

julia> my_split(".Das.ist.ein..String.",'.')
4-element Array{String,1}:
 "Das"
 "ist"
 "ein"
 "String"
```

11.3.7 Wurzelziehen

Nicht immer war Wurzelziehen so komfortabel wie heute. Mit der Unterstützung eines Computers oder Taschenrechners gelingt das ganz einfach mit der Funktion `sqrt`. In der Literatur findet man allerdings einige Verfahren, mit welchen „händisches“ Wurzelziehen möglich ist. Eines dieser Verfahren wird babylonisches Wurzelziehen[16] genannt.

Zur Berechnung der Quadratwurzel x von a wird zuerst eine Zahl x_0 als Kandidat gewählt (z.B. $x_0 = \frac{a}{2}$). Anstelle von $x = \sqrt{a}$ wird dann die Gleichung mit dem Kandidaten $x_0 \cdot x_0 \stackrel{!}{=} a$ gelöst. Wurde zufällig $x_0 = x$ gewählt, ist der Algorithmus beendet. Wahrscheinlicher ist jedoch, dass $x_0 \cdot x_0 \neq a$. Je nach der Wahl von x_0 ist x_0^2 jetzt größer oder kleiner als a, was auch bedeutet, dass x_0 jetzt entsprechend größer oder kleiner als x ist. Ändern wir die Gleichung für a zu $x_0 \cdot b = a$ können wir durch Umformen eine zweite Zahl b ermitteln, für die gilt

$$\begin{cases} b < x, & \text{wenn } x_0 > x \\ b > x, & \text{wenn } x_0 < x \end{cases}$$

Damit wurde eine obere und eine untere Schranke für x bestimmt, wobei jeder Wert innerhalb dieser Schranke eine bessere Annäherung and die Quadratwurzel ist. Der nächste Wert x_1 wird dann als Mittelwert von x_0 und b ermittelt. Allgemein ergibt sich daraus der Ausdruck

$$x_{n+1} = \frac{x_n + \frac{a}{x_n}}{2}.$$

Für eine ausreichend hohe Zahl von n kann damit die Quadratwurzel von a ermittelt werden. Es soll eine Funktion definiert werden, die mit dem beschriebenen Verfahren die Quadratwurzel einer beliebigen Zahl mit einem absoluten Fehler von 10^{-5} berechnet. Ein solches Abbruchkriterium wird auch Konvergenzkriterium genannt.

Beispiele:

```
julia> bsqrt(9)
3.0000000000393214

julia> bsqrt(103.12)
10.154801819992677
```

11.3.8 Dictionaries zusammenfügen

Schreibe eine Funktion `dict_merge`, die zwei Dicts vom gleichen Typ als Argumente nimmt, diese beiden Dictionaries zusammenführt (merged) und als Return Value zurückgibt. Sind Keys in beiden `Dicts` vorhanden, sollen die Werte aus dem ersten mit den Werten aus dem zweiten `Dict` überschrieben werden.

```
julia> d1 = Dict("Knopparp"=>71.99,"Söderhamn"=>314.1)
Dict{String,Float64} with 2 entries:
  "Söderhamn" => 314.1
  "Knopparp"  => 71.99

julia> d2 = Dict("Landskrona"=>494.1,"Ektorp"=>359.1)
Dict{String,Float64} with 2 entries:
  "Ektorp"     => 359.1
  "Landskrona" => 494.1
```

[16] https://de.wikipedia.org/wiki/Heron-Verfahren

```
julia> dict_merge(d1, d2)
Dict{String,Float64} with 4 entries:
  "Ektorp"     => 359.1
  "Söderhamn"  => 314.1
  "Landskrona" => 494.1
  "Knopparp"   => 71.99
```

12 Debugging

12.1 Grundlagen und einfache Fälle

Debugging bezeichnet die Suche und das Entfernen von Fehlern (sogenannten *bugs*) aus einem Programm. Besonders bei sehr komplexer Software mit vielen verschachtelten Funktionen und unterschiedlichen Eingabewerten kann es schwierig sein, sich die Programmlogik für jeden Fall vorzustellen. Daher gibt es in Julia Tools, die bei der Suche nach Fehlern helfen können.

Die einfachste Form des Debuggings funktioniert durch Hinzufügen von print-Befehlen. Das Macro `@show` ist dafür besonders geeignet, da es neben dem Wert auch den Namen einer Variable ausgibt.

```
a = 100.1
@show a
```

In der folgenden Funktion hat sich ein Fehler eingeschlichen.

```
"Berechne x^n"
function bug_pow(x::Number,n::Int)
    x_pow = x
    for i in 1:n
        x_pow *= x
    end
    return x_pow
end
```

Eigentlich sollte die Funktion x^n berechnen, die Ergebnisse stimmen allerdings nicht:

```
bug_pow(3.5,6), 3.5^6
```

```
(6433.9296875, 1838.265625)
```

Um den Fehler zu finden, kann ein print-Statement in der `for`-Schleife eingefügt werden, um `i` und `x_pow` auszugeben.

```
"Berechne x^n"
function bug_show_pow(x::Number,n::Int)
    x_pow = x
    for i in 1:n
        @show i,x_pow
        x_pow *= x
    end
    return x_pow
end
```

Ein erneuter Aufruf der Funktion liefert folgende Ausgabe:

```
julia> bug_show_pow(3.5,6)
(i, x_pow) = (1, 3.5)
(i, x_pow) = (2, 12.25)
(i, x_pow) = (3, 42.875)
(i, x_pow) = (4, 150.0625)
(i, x_pow) = (5, 525.21875)
(i, x_pow) = (6, 1838.265625)
6433.9296875
```

Offensichtlich wird in der Schleife `x`, 6× multipliziert, allerdings startet `x_pow` nicht mit 1, sondern mit `x`, weshalb gesamt x^{n+1} ergibt.

Diese Methode des Debuggings funktioniert bei einfachen Fällen, hat aber den Nachteil, dass der Code verändert werden muss. Julia bietet in Kombination mit `Debugger.jl`[17] und Juno einen Debugger im klassischen Sinn, der eine wesentlich präzisere und effizientere Fehlersuche erlaubt.

[17] https://github.com/JuliaDebug/Debugger.jl

12.2 Das Debugger-Paket

Mit einem klassischen Debugger ist es möglich, die Ausführung des Quellcodes zu unterbrechen und schrittweise zu analysieren. Dabei gibt es die Möglichkeit, sogenannte *Breakpoints* zu setzen, an welchen der Code unterbrochen werden soll.

Um die Funktion `bug_pow` mit dem Debugger zu analysieren, muss der Debugger-Modus über einen Klick auf den Debugknopf gestartet werden.

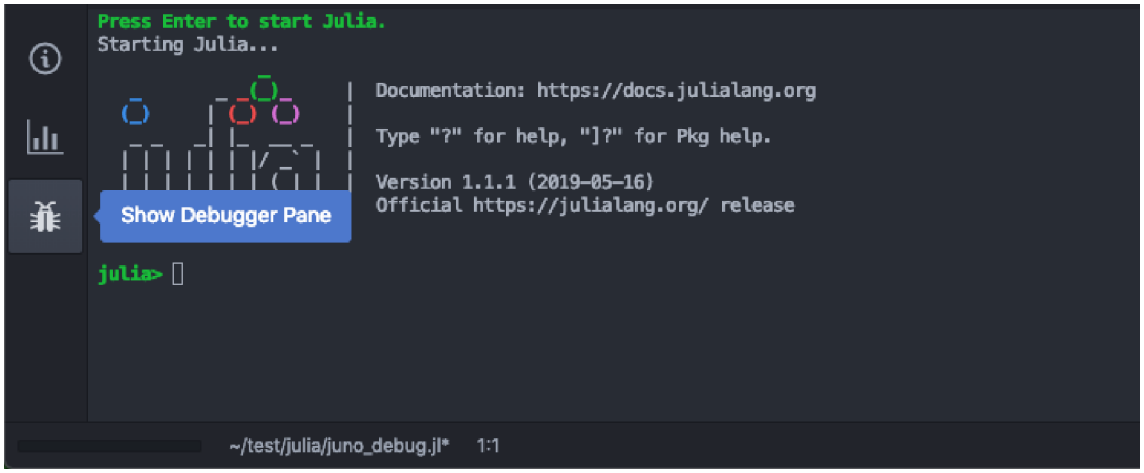


Abbildung 2: Debugknopf in Juno

Über die Seitenleiste mit den Zeilennummern kann nun durch einen Klick ein Breakpoint gesetzt werden. Dadurch wird die Ausführung in der 4. Zeile angehalten.

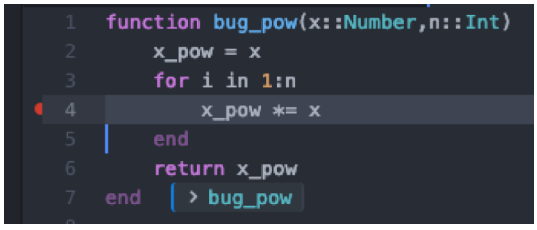


Abbildung 3: Erster Breakpoint

Mit dem Befehl `Juno.@run` startet der Debugger die Ausführung und stoppt beim ersten Breakpoint. Um direkt in der ersten Zeile der Funktion zu stoppen, kann der Befehl `Juno.@enter` verwendet werden.

```
Juno.@run bug_pow(3.5,6)
```

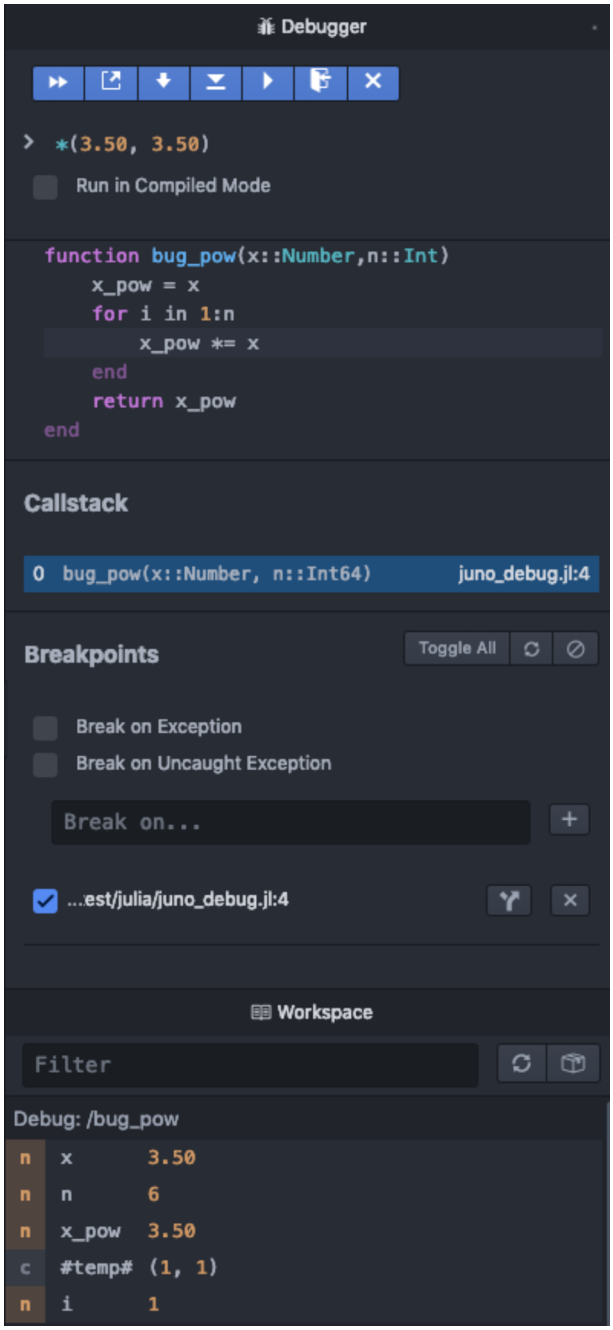


Abbildung 4: Erster Halt

Mit einem Blick in den Workspace kann man erkennen, dass im ersten Schleifendurchgang `i == 1`, die Variable `x_pow` bereits einen Wert ungleich 1 hat. Von dem Breakpoint aus gibt es die folgenden Möglichkeiten zum Fortsetzten der Ausführung:

- *Continue*: Funktioniert wie der Befehl `Juno.@run` und lässt den Code bis zum nächsten Breakpoint laufen.
- *Finish Function*: Läuft bis zum Ende der aktuellen Funktion oder bis zum nächsten Breakpoint.
- *Next line*: Springt zur nächsten Zeile.
- *Step to Selected Line*: Springt zur im Editor ausgewählten Zeile oder bis zum nächsten Breakpoint.
- *Next Expression*: Springt zum nächsten Ausdruck (z.B. `2*3+1` stoppt bei `2*3`, der nächste Ausdruck ist dann `6+1`)
- *Step into Function*: Funktioniert wie der Befehl `Juno.@enter` und springt in die erste Zeile der aktuellen Funktion.

- *Stop Debugging*: Beendet das Debugging.

Nähere Informationen zum Debugger gibt es in diesem Blogeintrag[18].

12.3 Aufgaben

12.3.1 Finde den Fehler

Die folgende Funktion berechnet den größten gemeinsamer Teiler zweier Zahlen `a` und `b`.

```
function ggt(a::Int,b::Int)
    for teiler in min(a,b):-1:0
        if a%teiler == 0 && b%teiler == 0
            return teiler
        end
    end
    return 1
end
```

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

Julia hat mit der Funktion `gcd` bereits eine Implementierung dieser Aufgabe. Damit lässt sich an 1000 zufälligen Zahlenkombinationen zwischen 1 und 1000 einfach zeigen, dass die Funktion `ggt` für viele Fälle das korrekte Ergebnis liefert:

```
julia> a = rand(1:1000,1000); b = rand(1:1000,1000);

julia> ggt.(a,b) == gcd.(a,b)
true
```

Für die folgenden Fälle stimmt das Ergebnis allerdings nicht.

```
julia> ggt(10,0) == gcd(10,0)
Error: DivideError: integer division error

julia> ggt(0,10) == gcd(0,10)
Error: DivideError: integer division error

julia> ggt(-2304,288) == gcd(-2304,288)
false
```

Finde die Fehler mit dem Debugger und korrigiere die Funktion.

13 Spezielle Datentypen

13.1 Datum und Uhrzeiten

Das korrekte Behandeln von Datumsangabe und Uhrzeiten ist grundsätzlich nicht die größte Herausforderung, allerdings gibt es einige Parameter zu beachten, wenn mit Daten und Zeitangaben gerechnet wird. Julia bietet für diesen Zweck das `Dates`-Paket, das in der *Standard Library* enthalten ist und mit

```
using Dates
```

geladen werden kann.

Die beiden Haupttypen des `Dates`-Pakets sind `Date` (minimale Angabe in Tagen z.B. `1.2.2019`) und `DateTime` (minimale Angabe in ms z.B. `1.2.2019 12:30:59.001`). Die Eingabe eines Datums bzw. eines Datums mit Uhrzeit erfolgt immer in der Reihenfolge Jahr, Monat, Tag, Stunde, Minute, Sekunde, Millisekunde.

```
julia> Date(2019,2,1)
2019-02-01

julia> DateTime(2019,2,1)
2019-02-01T00:00:00

julia> DateTime(2019,2,1,12,30,59,1)
2019-02-01T12:30:59.001
```

[18] https://julialang.org/blog/2019/03/debuggers

Wird eine Stelle bei der Definition vernachlässigt, verwendet Julia stattdessen den kleinst-möglichen Wert:

```
Date(2019)
```

```
2019-01-01
```

Zwischen zwei Datums- oder Zeitangaben (die auf einen exakten Zeitpunkt hinweisen) sind die folgenden Regeln definiert:

```
julia> date1 = Date(2019,2,1)
2019-02-01

julia> date2 = Date(2019,6,12)
2019-06-12

julia> date1 > date2
false

julia> date1 == date2
false

julia> date1 != date2
true

julia> date2 - date1
131 days

julia> datetime1 = DateTime(2019,2,1)
2019-02-01T00:00:00

julia> datetime2 = DateTime(2019,6,12,13,1)
2019-06-12T13:01:00

julia> datetime2-datetime1
11365260000 milliseconds
```

Die beiden gezeigten Subtraktionen ergeben die Zeitspanne zwischen den beiden Zeitpunkten, jeweils in der kleinst-möglichen Auflösung. Solche Zeitspannen (auch in anderer Auflösung z.B. Jahr oder Monat) sind auch im `Dates`-Paket enthalten.

```
julia> Dates.Year(2)
2 years

julia> Dates.Month(3)
3 months

julia> Dates.Day(1)
1 day

julia> Dates.Hour(4)
4 hours

julia> Dates.Minute(8)
8 minutes

julia> Dates.Second(12)
12 seconds

julia> Dates.Millisecond(80)
80 milliseconds
```

In Kombination mit einer Zeitspanne zeigt sich, wie praktisch das Paket sein kann. Neben der Subtraktion ist für Zeitspanne und Zeitpunkt auch die Addition definiert:

```
julia> Date(2019,1,1) + Dates.Day(3)
2019-01-04

julia> Date(2019,1,1) - Dates.Day(1)
2018-12-31

julia> Date(2019,1,31) + Dates.Month(1)
2019-02-28
```

Zusätzlich funktionieren auch die Angabe von Datumsbereichen wie gewohnt:

```
julia> print(collect(Date(2019,1,1):Dates.Day(2):Date(2019,1,10)))
Dates.Date[2019-01-01, 2019-01-03, 2019-01-05, 2019-01-07, 2019-01-09]
```

Jede Zeitspanne und jedes Datum werden durch eine zugeordnete Einheit (Tag, Monat, Jahr) und eine `Int`-Zahl festgelegt. In manchen Fällen ist es notwendig, diese `Int`-Zahl zu extrahieren. Für Zeitspannen gibt es dafür die Funktion `Dates.value`:

```
julia> d1 = Dates.Day(10)
10 days

julia> d2 = Dates.Month(10)
10 months

julia> d1 == d2
false

julia> Dates.value(d1) == Dates.value(d2)
true
```

Für Datums und Zeitangaben gibt es die Funktionen:

```
julia> dt = DateTime(2019,6,12,13,1)
2019-06-12T13:01:00

julia> Dates.year(dt)
2019

julia> Dates.month(dt)
6

julia> Dates.day(dt)
12

julia> Dates.hour(dt)
13

julia> Dates.minute(dt)
1

julia> Dates.second(dt)
0

julia> Dates.millisecond(dt)
0
```

13.2 Spezielle Zahlentypen

Aufgrund einiger Basisregeln, nach denen in Julia programmiert wird, eignet sich die Sprache besonders dafür, erweitert zu werden. In der Julia-Paketdatenbank befindet sich eine Vielzahl von praktischen Paketen, welche die Standardfunktionalität erweitern. Zwei davon sind:

- `Unitful.jl`[19], das physikalische Einheiten definiert und
- `Measurements.jl`[20], das Berechnungen mit Fehlertoleranzen erlaubt.

Beide Paket definieren eine neue Art, Zahlen anzugeben, welche aber weiterhin auf den Rechenregeln die in Julia definiert sind basiert. Als Beispiel wird eine Funktion definiert die den Flächeninhalt eines Rechtecks berechnet:

```
A_rechteck(h,b) = h*b
```

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

Wie gewohnt kann die Funktion mit unterschiedlichen Zahlen, z.B. `Float64` oder `Int`, aufgerufen werden:

```
julia> A_rechteck(2.1,3.1)
6.510000000000001

julia> A_rechteck(2,3)
6
```

[19] http://painterqubits.github.io/Unitful.jl/stable/
[20] https://juliaphysics.github.io/Measurements.jl/stable/

Mit `Unitful.jl` ist es möglich, Zahlen mit einer Einheit zu versehen:

```
julia> using Unitful

julia> 1u"m"
1 m

julia> 1u"cm"
1 cm

julia> force = 12u"kg"*9.81u"m/s^2"
117.72 kg m s^-2

julia> force == 117.72u"N"
true
```

Der neue Zahlentyp lässt sich jetzt auch mit der Funktion `A_rechteck` verwenden:

```
julia> fläche = A_rechteck(2u"m",11.1u"cm")
22.2 cm m

julia> uconvert(u"mm^2",fläche)
222000.0 mm^2
```

Mit `Measurements.jl` können Fehler bzw. Abweichungen einer Zahl einfach durch das Symbol `±` (`\pm`+`TAB`-Taste) berücksichtigt werden:

```
julia> using Measurements

julia> a = 1.1±0.1
1.1 ± 0.1

julia> b = 2.1±0.05
2.1 ± 0.05

julia> a+b
3.2 ± 0.11

julia> sin(a)
0.891 ± 0.045
```

Das Paket geht dabei von normalverteilten Variablen aus; der Wert nach `±` ist die Standardabweichung. Die Variable `a` im obigen Beispiel bezeichnet daher eine normalverteilte Variable mit Mittelwert `1.1` und Standardabweichung `0.1`

Bei der Verwendung von `A_rechteck` kann mit `Measurements.jl` der Fehler bei der Berechnung der Fläche, unter Berücksichtigung des Fehlers der Seiten, ermittelt werden:

```
julia> A_rechteck(2±0.1,3±0.5)
6.0 ± 1.0
```

13.3 Aufgaben

13.3.1 Ein Zeiträtsel

Jemand verwechselt Tages- und Monatszahl und erscheint daher sechs Wochen zu früh zu einem Termin. An welchem Tag ist der Termin eigentlich?[21] Schreibe eine Funktion, die den entsprechenden Tag zurückgibt. Verwende für die Lösung die arithmetischen Funktionen für Daten.

14 Arbeiten mit Dateien und dem Dateisystem

Beim Automatisieren von Prozessen kommt es oft vor, dass große Dateimengen (z.B. Messwerte) eingelesen und verarbeitet werden müssen. Dafür ist es notwendig, zum einen die Dateien bzw. Dateinamen zu finden und zum anderen benötigt man eine Möglichkeit, Dateien einzulesen.

[21] https://www.spektrum.de/raetsel/melusines-geburtstag/1660200

14.1 Das Dateisystem

Julia bietet ein einheitliches Interface, um auf das Dateisystem des Computers zuzugreifen. Wird ein Juliaprozess ausgeführt, läuft er in dem Verzeichnis, in dem er gestartet wurde, dem sogenannten „working directory“. Anzeigen lässt sich dieses mit der Funktion `pwd` für „print working direcory“.

```
julia> pwd()
"/private/var/folders/0z/xyn859fx3vj4762qh24f8dq80000gn/T/jl_Oeiykp"
```

Die Funktion `readdir` gibt alle Dateien und Ordner eines Verzeichnisses aus.

```
julia> print(readdir()) # Standardmäßig wird das working directory verwendet
["Datei01.txt", "Datei02.txt", "Datei05.csv", "Ordner01"]
julia> print(readdir("Ordner01")) # Optional kann ein Pfad angegeben werden
["Datei03.txt"]
```

Die Ausgabe von `readdir` besteht allerdings nur aus den Datei- oder Ordnername, als `String`. Es ist also nicht möglich zu erkennen, ob es sich um einen Ordner oder eine Datei handelt. Dafür gibt es die Funktionen `isdir` und `isfile`.

```
julia> isdir.(readdir()), isfile.(readdir())
(Bool[0, 0, 0, 1], Bool[1, 1, 1, 0])
```

Weitere praktische Funktionen, z.B. zum Verschieben und Löschen von Dateien, sind in der Dokumentation unter Base/Filesystem angeführt.

14.2 Arbeiten mit Dateien

Beim Einlesen von Dateien kann grob in zwei Typen unterschieden werden:

- einfache Textdateien und
- Binärdateien.

Textdateien sind solche, die mit einem simplen Editor geöffnet werden können und für die es nicht notwendig ist, die Daten vor der Verarbeitung zu Dekodieren. Klassische Formate sind csv, txt, tex, xml.

Eine solche Datei kann mit der Funktion `open` geöffnet werden. `open` erfordert neben der Angabe des Dateipfades, auch einen Modus. Beim Umgang mit Dateien gibt es die folgenden Modi:

- `r` read: Die Datei wird nur gelesen.
- `w` write: Die Datei wird erzeugt (bzw. überschrieben, falls sie vorhanden ist) und mit Daten gefüllt.
- `a` append: Die Datei wird um Daten erweitert.

```
julia> f = open("Datei04.txt","w")
IOStream(<file Datei04.txt>)

julia> print(readdir())
["Datei01.txt", "Datei02.txt", "Datei04.txt", "Datei05.csv", "Ordner01"]
```

Die Ausgabe von `readdir` zeigt, dass bereits durch den Aufruf von `open`, eine neue Datei `Datei04.txt` erzeugt wurde. Der Befehl `open` öffnet einen sogenannten Stream zu der Datei. Um nun Text in die neue Datei zu schreiben, muss dieser Text zuerst in den Stream geschrieben werden. Das ist mit dem `write`-Befehl möglich.

```
julia> write(f,"Erste Zeile\n")
12

julia> write(f,"Zweite Zeile")
12
```

Dadurch landen die Texte zuerst im Stream. Erst wenn der Stream mit `close` geschlossen wird, werden die Daten geschrieben.

```
julia> close(f)
```

Das Pendant zu `write` ist die Funktion `read`, die eine Datei vollständig mit einem angegeben Dateitypen einliest. Da es häufig vorkommt, dass eine Datei aufgeteilt in die einzelnen Zeilen benötigt wird, gibt es noch die Funktion `readlines`, die ein `String-Array` zurückgibt.

```
julia> f = open("Datei04.txt","r")
IOStream(<file Datei04.txt>)

julia> read(f,String)
"Erste Zeile\nZweite Zeile"

julia> close(f)

julia> f = open("Datei04.txt","r")
IOStream(<file Datei04.txt>)

julia> readlines(f)
2-element Array{String,1}:
 "Erste Zeile"
 "Zweite Zeile"

julia> close(f)
```

Generell sollten Streams immer mit `close` geschlossen werden, wenn sie nicht mehr in Verwendung sind.

14.2.1 Strukturierte Textdateien

Besonders häufig ist es notwendig, Textdateien mit einer gewissen Struktur einzulesen. Für sehr einfache Fälle mit homogenen Daten, wie z.B. einer Zahlenmatrix, kann das in Julia enthaltene `DelimitedFiles`-Paket verwendet werden.

```
julia> using DelimitedFiles

julia> A = rand(2,2)
2×2 Array{Float64,2}:
 0.576308   0.961758
 0.0265431  0.731027

julia> writedlm("MeineMatrix.txt",A)

julia> isfile("MeineMatrix.txt")
true

julia> f = open("MeineMatrix.txt","r")
IOStream(<file MeineMatrix.txt>)

julia> readlines(f)
2-element Array{String,1}:
 "0.5763076728261607\t0.961758292751214"
 "0.026543141270017445\t0.7310273066667246"

julia> close(f)

julia> B = readdlm("MeineMatrix.txt")
2×2 Array{Float64,2}:
 0.576308   0.961758
 0.0265431  0.731027

julia> A == B
true
```

Bei der folgenden Datei kann es sich um eine sehr übliche, bei einer Messung aufgezeichnete, Datei handeln.

```
julia> f = open("Datei05.csv","r")
IOStream(<file Datei05.csv>)

julia> readlines(f)
5-element Array{String,1}:
 "time;temp;Status"
 "1;12,0;A"
 "2;13,1;A"
 "5; 9;B"
 "10;14;A"

julia> close(f)
```

Es gibt eine Kopfzeile, verwendete Datentypen sind `Int`, `Float` und `String` und die `Float`-Zahlen sind anstelle eines Punktes mit einem Komma getrennt.

`readdlm` liest die Datei zwar ein, das Resultat ist aber noch nicht sinnvoll verwendbar:

```
julia> readdlm("Datei05.csv")
5×2 Array{Any,2}:
 "time;temp;Status"  ""
 "1;12,0;A"          ""
 "2;13,1;A"          ""
 "5;"                "9;B"
 "10;14;A"           ""
```

Mit einem Blick in die Hilfe von `readdlm` können wir das Ergebnis durch Angeben des Trennzeichens mit `delim=';'` und Überspringen der Kopfzeile mit `skipstart=1` etwas verbessern.

```
julia> readdlm("Datei05.csv",';',skipstart=1)
4×3 Array{Any,2}:
  1    "12,0"  "A"
  2    "13,1"  "A"
  5   9        "B"
 10  14        "A"
```

Weitere Optionen sind in der Dokumentation unter Standard Library/Delimited Files zu finden.

Das `DelimitedFiles`-Paket stößt bei inhomogenen Daten oder auch bei fehlenden Daten schnell an seine Grenzen. Um die Daten von oben nutzbar zu machen, müssten noch die Kommas durch Punkte und die daraus resultierenden Strings in Zahlen umgewandelt werden. `CSV.jl` in Kombination mit `DataFrames.jl` bietet für diesen Fall schon alle nötigen Optionen zum Einlesen:

```
julia> using CSV, DataFrames

julia> messwerte = DataFrame(CSV.File("Datei05.csv",delim=';',decimal=',',types=[Int,Float64,String]))
4×3 DataFrames.DataFrame
| Row | time  | temp    | Status |
|     | Int64 | Float64 | String |
|-----+-------+---------+--------|
| 1   | 1     | 12.0    | A      |
| 2   | 2     | 13.1    | A      |
| 3   | 5     | 9.0     | B      |
| 4   | 10    | 14.0    | A      |
```

Durch das Konvertieren in einen `DataFrame` ist jetzt auch der Zugriff auf einzelne Werte sehr einfach möglich:

```
julia> messwerte.temp, messwerte.Status
([12.0, 13.1, 9.0, 14.0], ["A", "A", "B", "A"])
```

Ein `DataFrame` ist ein Datentyp, der besonders für die Verarbeitung von tabellarischen Daten geeignet ist. Nähere Details zu DataFrames[22] und `CSV.jl`[23] können der jeweiligen Dokumentation entnommen werden.

14.3 Aufgaben

Die Aufgaben für dieses Beispiel basieren alle auf dem selben Datensatz. In den Beispielen geht es darum, als ManagerIn eines Prepper-Camps[24], das aus mehreren Lagern besteht, den Überblick über den Lagerbestand zu behalten. Für jedes Lager wird eine Liste mit Ein- und Ausgängen geführt:

```
Artikel,Bestand
Batterie,5
Reis,10
Fernglas,5
Taschenmesser,38
Konserven,28
Konserven,-23
Batterie,29
Fernglas,-2
Fernglas,-1
```

[22] https://juliadata.github.io/DataFrames.jl/stable/
[23] https://juliadata.github.io/CSV.jl/stable/
[24] https://de.wikipedia.org/wiki/Prepper

Im Kontrollprogramm ist für das Erstellen von Testdatensätzen die Funktion

```
julia> JuliaSkriptumKontrolle.setup("14.3")
```

definiert. Nach dem Aufruf wird im aktuellen Verzeichnis ein neuer Ordner `14-3` erstellt.

```
julia> print(readdir("14-3"))
["Lager1.csv", "Lager10.csv", "Lager2.csv", "Lager3.csv", "Lager4.csv", "Lager5.csv", "Lager6.csv", "Lager7.csv", "Lager8.csv
", "Lager9.csv"]
```

In dem Ordner befinden sich die Bestandslisten für alle Lager. Die Lager sind immer nach dem Schema `Lager[id].csv` benannt. Der Code soll in allen Prepper-Camps verwendet werden können, d.h. die Anzahl an Dateien kann sich immer ändern und da alle Camps verschiedene Betriebssysteme verwenden, muss immer `joinpath` beim Verketten von Pfaden verwendet werden:

```
joinpath("14-3","Lager1.csv")
```

```
"14-3/Lager1.csv"
```

Wie im Abschnitt Interaktion erwähnt, sind die Trennzeichen bei Pfaden je nach Betriebssystem verschieden. Da das zu einigen Problemen führen kann, gibt es die Funktion `joinpath`, die das richtige Trennzeichen wählt.

14.3.1 Bestandsermittlung Lager

Es wird eine Funktion benötigt, die den Bestand jedes Gegenstands pro Lager ermittelt (Summe der Ein- und Ausgänge). Die Funktion soll ein `Dict` mit der Zuordnung `Lager id => Bestand` zurückgeben, wobei `Bestand` wieder ein `Dict` mit der Zuordnung `Artikel => Anzahl` ist. Ein `Dict` muss nicht notwendigerweise von Anfang an mit Werten befüllt sein:

```
a = Dict{String,Int}()
```

```
Dict{String,Int64} with 0 entries
```

Dieser Aufruf erzeugt ein leeres `Dict`, bei dem die Keys vom Typ `String` und die Values vom Typ `Int` sein müssen. Werte können dann wie bei jedem anderen `Dict` hinzugefügt werden:

```
julia> a["Taschenlampe"] = 10
10

julia> a["Taschenlampe"] += -5
5

julia> a["Kein Int"] = 1.2
Error: InexactError: Int64(1.2)

julia> a[11] = 3
Error: MethodError: Cannot `convert` an object of type Int64 to an object of type String
Closest candidates are:
  convert(::Type{String}, !Matched::Union{CategoricalArrays.CategoricalString{R}, CategoricalArrays.CategoricalValue{T,R}
where T} where R) at /Users/sebastianpech/.julia/packages/CategoricalArrays/qcwgl/src/value.jl:82
  convert(::Type{String}, !Matched::WeakRefStrings.WeakRefString) at /Users/sebastianpech/.julia/packages/WeakRefStrings/
POE8H/src/WeakRefStrings.jl:77
  convert(::Type{T<:AbstractString}, !Matched::T<:AbstractString) where T<:AbstractString at strings/basic.jl:208
  ...

julia> a
Dict{String,Int64} with 1 entry:
  "Taschenlampe" => 5
```

Beispiele:

```
julia> bestand = read_lagers();

julia> typeof(bestand)
Dict{Int64,Dict{String,Int64}}

julia> print(bestand[1])
Dict("Gummistiefel" => 37,"Wasserflasche" => 46,"Taschenlampe" => 3,"Konserven" => 3,"Medikamente" => 17,"Schlafsack" => 3,
"Taschenmesser" => 0,"Reis" => 63,"Batterie" => 45,"Dosenöffner" => 0)
julia> print(bestand[3])
Dict("Taschenlampe" => 32,"Wasserflasche" => 85,"Gummistiefel" => 31,"Batterie" => 51,"Medikamente" => 66,"Kocher" => 1,
"Taschenmesser" => 32,"Fernglas" => 7,"Konserven" => 23,"Schlafsack" => 6,"Reis" => 18,"Dosenöffner" => 35)
```

14.3.2 Bestand suchen

Es soll eine Funktion geschrieben werden, die in allen Lagern nach einem Gegenstand `X` sucht, von dem noch mindestens `n` Stück vorhanden sind. Ausgabe der Funktion ist ein `Int`-Array mit den ids der Lager. Wird kein Lager gefunden, soll ein leeres Array ausgegeben werden. Als Basis soll die Funktion aus der ersten Aufgabe `read_lagers` herangezogen werden. Für die Kontrolle muss die Funktion dann in den Kontrollblock kopiert werden:

```
@Aufgabe "14.3.2" begin
    # using ...
    function read_lagers()
        # code ...
    end
    function lager_mit(artikel,stückzahl)
        # code ...
    end
end
```

Beispiele:

```
julia> print(lager_mit("Taschenlampe",4))
[4, 10, 2, 3, 5, 8, 6]
julia> print(lager_mit("Reis",30))
[7, 1]
```

14.3.3 Gesamtbestand

Es soll eine Funktion geschrieben werden, die den Bestand des gesamten Camps ermittelt. Die Ausgabe soll wie bei der ersten Aufgabe als `Dict` erfolgen. Als Basis soll wieder die Funktion aus der ersten Aufgabe verwendet werden.

Beispiele:

```
julia> print(gesamt_bestand())
Dict("Taschenlampe" => 200,"Salz" => 84,"Wasserflasche" => 260,"Gummistiefel" => 357,"Batterie" => 222,"Medikamente" => 330,
"Kocher" => 124,"Taschenmesser" => 150,"Fernglas" => 275,"Konserven" => 185,"Schlafsack" => 32,"Bier" => 177,"Reis" => 153,
"Dosenöffner" => 187)
```

15 Plotting

Die grafische Darstellung von Daten ist in vielen Wissenschaften ein immens wichtiger Punkt. Julia bietet eine Reihe von Paketen zum „Plotten“ von Daten. `Plots.jl` vereint diese in einer Sprache zum Beschreiben von Grafiken. Geladen wird das Paket mittels

```
using Plots
```

`Plots.jl` bieten einen sehr großen Funktionsumfang, wodurch es bei jedem Neustart von Julia durchaus länger dauern kann, bis der erste Plot sichtbar wird.

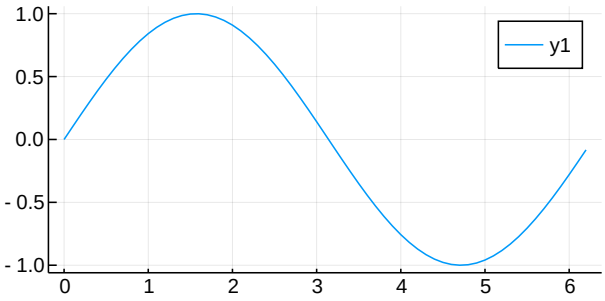
In `Plots.jl` werden meistens Datenvektoren dargestellt. Das Paket bietet Implementierungen für einige Elementstypen wie

- alle Zahlentypen in Julia,
- `Date` und `DateTime` sowie
- Zahlen aus `Measurements.jl`

und ist sehr leicht erweiterbar.

Die einfachste Form von Plot kann folgendermaßen erzeugt werden:

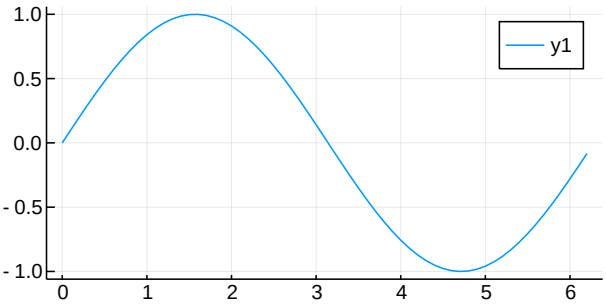
```
x = 0:0.1:2π
y = sin.(x)
plot(x,y)
```



Dabei handelt es sich um einen Linienplot. Das erste Argument der Funktion `plot` sind die x-Werte, das zweite die y-Werte, die Länge der beiden Vektoren muss dabei übereinstimmen.

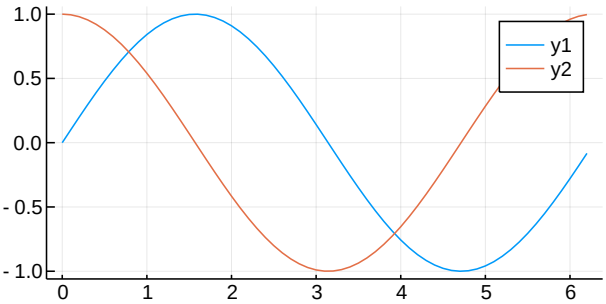
Jeder Aufruf von `plot` erzeugt ein *Plotobjekt*, das dann weiterverwendet werden kann.

```
plt = plot(x,y)
plt
```

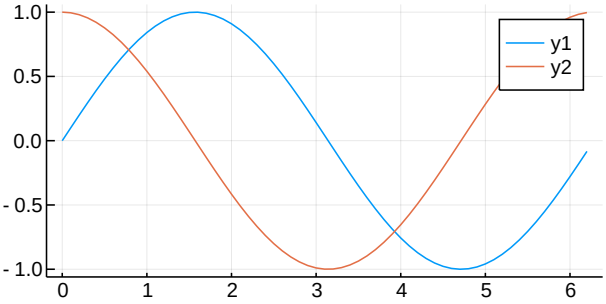


Um mehrere Daten übereinander in einem Plot darzustellen, gibt es die Möglichkeit, dies direkt mit einem Aufruf von `plot` oder durch nachträgliches Hinzufügen mit `plot!` zu machen

```
y2 = cos.(x)
plot(x,[y y2]) # plot([x x2],[y y2]) für verschiedene x-Werte
```




```
plot!(plt,x,y2)
```



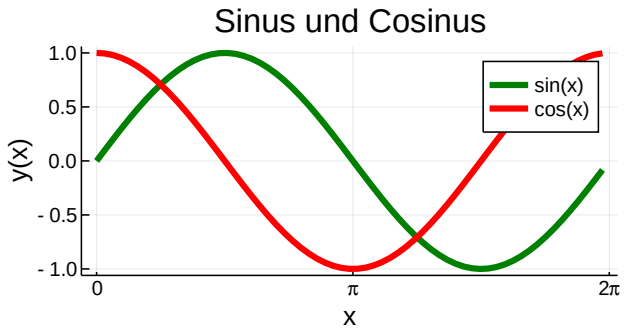
`Plots.jl` bietet sogenannte *Magic Arguments*. Damit ist es möglich, die Darstellung der Daten zu verändern, wobei von der Bedeutung des Konfigurationswerts auf die zugehörige Einstellung geschlossen wird.

Beispielsweise:

- `:red` deutet auf eine Farbwahl hin
- `(0,10)` deutet bei Achskonfigurationen auf den Minimal- und den Maximalwert der Achse hin
- `:log` deutet bei Achskonfigurationen auf die Wahl einer logarithmischen Skalierung hin

Der Plot mit Sinus und Cosinus kann damit optisch vervollständigt werden:

```
plot(x,[y,y2],
     line=([:green :red], :solid, 4),
     label=["sin(x)" "cos(x)"],
     xaxis=("x",([0,π,2π],["0","\\pi","2\\pi"])),
     yaxis=("y(x)"),
     title="Sinus und Cosinus")
```



Dabei haben die Argumente den folgenden Effekt:

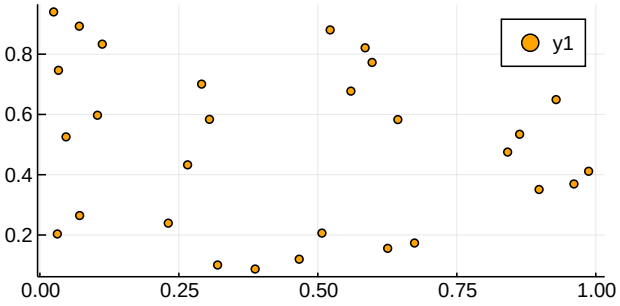
- `line=([:green :red], :solid, 4)`: Stelle `y` grün, `y2` rot, beide als durchgezogene Linie mit der Stärke 4 dar.
- `label=["sin(x)" "cos(x)"]`: Bezeichne `y` in der Legende mit sin(x) und `y2` mit cos(x).
- `xaxis=("x",([0,π,2π],["0","\\pi","2\\pi"]))`: Setze den Text unter der x-Achse auf x und schreibe an die Stellen von `0`, `π` und `2π` anstelle der Werte die entsprechenden Texte.
- `yaxis=("y(x)")`: Setze den Text neben der y-Achse auf y(x)
- `title="Sinus und Cosinus"`: Setze den Plottitel auf „Sinus und Cosinus“

Für eine gute Datenvisualisierung reichen Linienplots nicht aus. `Plots.jl` hat eine große Anzahl an verschiedenen Visualisierungsarten, in weiterer Folge werden Scatterplots, Balkendiagramme und Histogramme besprochen. Generell können alle gezeigten Optionen bei den anderen Diagrammarten ebenfalls verwendet und unterschiedliche Plots kombiniert werden.

15.1 Scatterplot

Scatterplots ermöglichen die Ausgabe von einzelnen Punkten, bei denen es nicht sinnvoll, ist sie mit einer Linie zu verbinden. Um Scatterplots zu formatieren, muss `line` durch `marker` ersetzt werden.

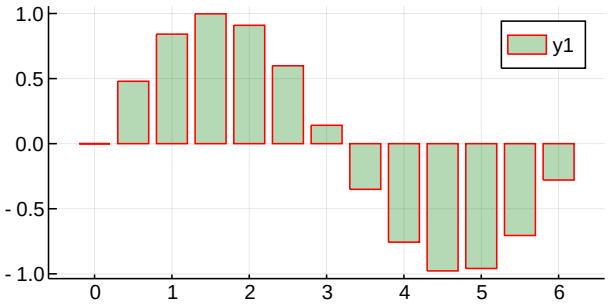
```
x = rand(30)
y = rand(30)
scatter(x,y,marker=(:circle,3,:orange))
```



15.2 Barchart

Balkendiagramme werden meist verwendet, um Häufigkeiten darzustellen. `Plots.jl` bietet dafür die `bar`-Funktion. Um Balkendiagramme zu formatieren, kann `line` und `fill` verwendet werden.

```
x = 0:0.5:2π
y = sin.(x)
bar(x,y,line=(:red),fill=(:green,.3))
```



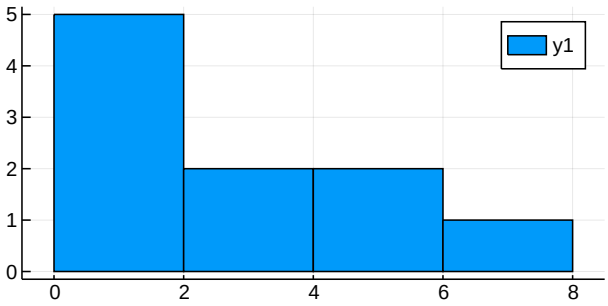
Histogramme sind im Grunde Balkendiagramme, allerdings werden sie ausschließlich zum Anzeigen von Häufigkeitsverteilungen verwendet. Ein großer Vorteil der `histogram`-Funktion ist, dass die Verteilung nicht ermittelt werden muss.

Für die folgenden Werte soll ein Histogramm erstellt werden:

```
label = [1,1,2,3,1,4,4,1,1,7]
```

Soll die `bar`-Funktion verwendet werden, müsste zuerst die Häufigkeit jeder Zahl in `label` gezählt werden. Mit `histogram` ist das nicht notwendig:

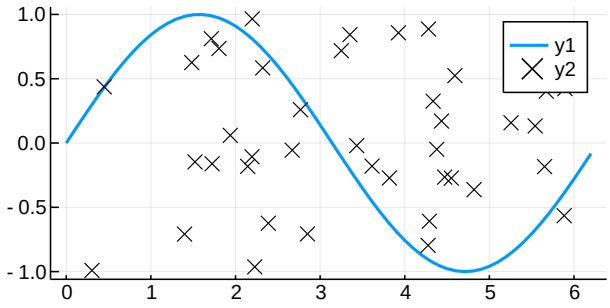
```
histogram(label)
```



15.3 Kombination verschiedener Visualisierungsarten

Die einfachste Methode verschiedene Plots zu kombinieren ist durch Verwendung der mit `!` gekennzeichneten Pendants der Plotfunktion, `plot!`, `scatter!`, `bar!`, `histogram!`.

```
x = 0:0.1:2π
y = sin.(x)
plt = plot(x,y,line=(2))
scatter!(plt,rand(40)*6,1 .- rand(40)*2,marker=(:x))
```



15.4 Speichern von Plots

Bisher konnten die Plots nur in einem eigenen Fenster angezeigt werden. In den meisten Fällen benötigt man diese in einem Grafikformat für ein anderes Dokument. `Plots.jl` bietet dafür die Funktion `savefig` in zwei Varianten:

```
savefig("mein_plot_1.png")
```

speichert den Inhalt des aktuellen Plotfensters und

```
savefig(plt,"mein_plot_2.png")
```

speichert `plt`. Das Dateiformat wird dabei automatisch aus dem Dateinamen extrahiert. Die obigen Aufrufe erzeugen `png`-Dateien. Alternativ wird häufig auch das Format `pdf` verwendet. `pdf` hat den Vorteil, dass die Grafik als Vektorgrafik erzeugt wird und damit ohne Qualitätsverlust auf eine beliebige Größe skaliert werden kann.

16 Statistik

In Julia sind einige grundlegende Statistikfunktionen im `Statistics`-Paket enthalten. Das Paket kann wie alle anderen Pakete mit `using` geladen werden.

```
using Statistics
```

Um die folgenden Beispiele anschaulich zu machen, werden Testdaten benötigt. Diese werden aus der Iris-Datenbank[25] entnommen. Die Datenbank enthält Abmessungen und Spezies von 150 Iris-Blumen und ist in `RDatasets.jl` enthalten.

```
using RDatasets
iris = dataset("datasets", "iris")
```

Die Variable `iris` ist ein `DataFrame` mit fünf Spalten:

```
names(iris)
```

```
5-element Array{Symbol,1}:
 :SepalLength
 :SepalWidth
 :PetalLength
 :PetalWidth
 :Species
```

Die ersten vier Spalten enthalten `Float`-Zahlen mit Längenangaben und die letzte Spalte enthält einen `String` mit der Spezies der Pflanze.

Häufig benötigte Funktionen für statistische Berechnungen sind

[25] https://en.wikipedia.org/wiki/Iris_flower_data_set

- Mittelwert `mean()`,
- Standardabweichung `std()`,
- Median `median()` und
- Varianz `var()`.

Für die Spalte `SepalLength` können mit diesen Funktionen die folgenden Werte ermittelt werden:

```
julia> mean(iris.SepalLength)
5.843333333333334

julia> std(iris.SepalLength)
0.828066127977863

julia> median(iris.SepalLength)
5.8

julia> var(iris.SepalLength)
0.6856935123042507
```

Neben diesen Funktionen gibt es noch weitere zur Berechnung der Kovarianz (`cov()`), des Pearson-Korrelationskoeffizient (`cor()`) und der Quantile (`quantile()`). Nähere Details dazu sind in der Dokumentation unter Standard Library/Statistics zu finden.

Oft ist es erforderlich, Annahmen zu Wahrscheinlichkeitsverteilungen gewisser Daten zu treffen. `Distributions.jl` definiert dafür die notwendigen Funktionen für eine Reihe von verschiedenen Verteilungen. Zusätzlich enthält das Paket Funktionen zum „Fitting“ und „Sampling“ von Daten und kann wie folgt geladen werden:

```
using Distributions
```

Um die Daten in den folgenden Beispielen zu visualisieren, wird ebenfalls `Plots.jl` geladen:

```
using Plots
```

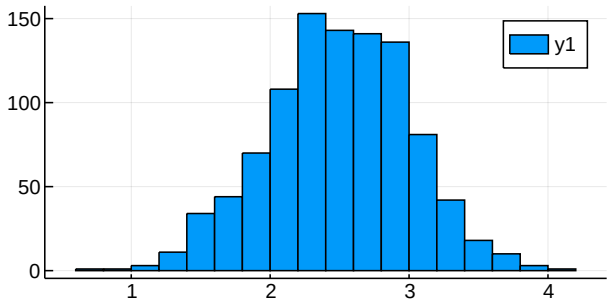
Beim „Sampling“ geht es darum, Zufallszahlen aus einer gewissen Verteilung zu ziehen. Ein Zufallszahlengenerator, mit einer zugrundeliegenden Normalverteilung (Mittelwert 2.5 und Standardabweichung 0.5), kann folgendermaßen erzeugt werden:

```
d = Normal(2.5,0.5)
```

```
Distributions.Normal{Float64}μ(=2.5, σ=0.5)
```

Mit `rand` generiert man nun z.B. 1000 Zufallszahlen, die dieser Verteilung entsprechen.

```
data = rand(d,1000)
histogram(data)
```



Weitere Verteilungen sind in der Dokumentation[26] gelistet.

Neben dem „Sampling“ von Daten ist auch das „Fitting“, also Ermitteln der Parameter für Verteilungsfunktionen, möglich. Dafür wird die Funktion `fit` verwendet. In einem ersten Schritt werden für die Spalte `SepalLength` aus den Daten in `iris` die Parameter für eine Normalverteilung ermittelt:

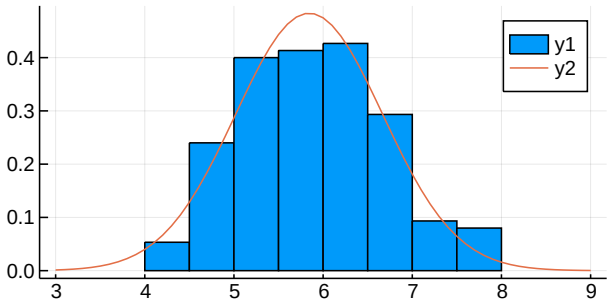
```
d = fit(Normal, iris.SepalLength)
```

[26] https://juliastats.github.io/Distributions.jl/stable/

```
Distributions.Normal{Float64}μ(=5.843333333333335, σ=0.8253012917851409)
```

Der Vergleich mit den zuvor ermittelten Werten zeigt eine ungefähre Übereinstimmung. Um die Qualität des „Fittings“ zu beurteilen, werden ein normalisiertes Histogramm der Daten und die Dichtefunktion der ermittelten Verteilung dargestellt.

```
plt = histogram(iris.SepalLength,normed=true)
x = 3:0.1:9
# Berechnen der Werte der Dichtefunktion für d an den Stellen in x
y = pdf.(d,x)
plot!(plt,x,y)
```



Offensichtlich passt die Dichtefunktion gut zu den Messwerten. Die angepasste Normalverteilung kann jetzt wie zuvor für das Generieren von Zufallswerten verwendet werden.