

Programmieren mit R für Einsteiger



Berry Boessenkool



frei verwenden, zitieren

2022-04-06 08:52

- 0. Intro
- 1. Grundlagen
- 2. Datentypen
- 3. Tabellen
- 4. Grafiken

- 0.1 Willkommen
- 0.2 Fünf-Minuten Showcase
- 0.3 Einrichtung: R & Rstudio
- 0.4 Übungsaufgaben

- ▶ **Berry**: Geoökologie @ Uni Potsdam, Dozent am HPI
- ▶ R Fan seit 2010
- ▶ R-Pakete  , Training & Beratung  , Community 

- ▶ **Pia**: Medizin @ University of St Andrews, Edinburgh Scotland
- ▶ Master Digital Health am HPI (Digital Health Center, DHC)
- ▶ RKI Intensivregister: Grafiken für Kristenstäbe / Bürger*innen

- ▶ **Bert**: Leiter Lehrstuhl Digital Health - Connected Healthcare (HPI)
- ▶ Studiengangsverantwortlicher Masterprogramm Digital Health
- ▶ R für Analyse von gesundheits-relevanten Daten aus dem Alltag

- ▶ Grundlagen und Anwendungen (möglichst konkret)
- ▶ Freude am Programmieren
- ▶ Gelerntes in eigenen Projekten verwenden
- ▶ effiziente und reproduzierbare Arbeit
- ▶ Inhaltlicher Ablauf auf openHPI
- ▶ Video pro Lektion
- ▶ Übungsaufgaben auf CodeOcean (im Browser) / in Rstudio
- ▶ Fragen? -> **Kursforum!** [Diskussionen] (unter den Videos erstellen)

- 0. Intro
- 1. Grundlagen
- 2. Datentypen
- 3. Tabellen
- 4. Grafiken

- 0.1 Willkommen
- 0.2 Fünf-Minuten Showcase
- 0.3 Einrichtung: R & Rstudio
- 0.4 Übungsaufgaben

- ▶ 5 Minuten Beispiel für die Nutzung von R und Rstudio
 - ▶ Als 'Live Coding' gezeigt, d.h. der Code wird im Skript entwickelt und vorgeführt
 - ▶ Die nachfolgenden Folie zeigt den Code zur Referenz
-
- ▶ Ziel ist nicht, alles im Detail zu verstehen,
 - ▶ sondern 'Appetit' auf R zu bekommen :)
 - ▶ Am Kursende kannst du diese Sachen auch
-
- ▶ sit back, relax, enjoy the show...

Showcase Code

```
# R als Taschenrechner: -> 'sinnvolles' Ergebnis
7 * 6

# Objekte erstellen um später damit zu arbeiten:
alter <- 33
alter + 1          # so alt bin ich nächstes Jahr
2022 - alter      # so finde ich raus, wann ich geboren wurde, falls ich das mal vergesse

# Daten einlesen:
wetter <- read.table(file="wetter.txt", header=TRUE)

# Aus einer Tabelle eine Spalte auswählen:
wetter$Regen

# Graphik - Scatterplot für Korrelation (Zusammenhang, kann aber dritte Gründe haben):
plot(wetter$Sonne, wetter$Temperatur, col="orange", pch=16, main="Viel Sonne an warmen Tagen")

# Graphik - Zeitreihe für Entwicklung / Saisonalität / Trend:
wetter$Datum <- as.Date(wetter$Datum, format="%Y-%m-%d")
plot(wetter$Datum, wetter$Luftdruck, type="l", col="salmon", lwd=2, xaxt="n")

# Pakete von anderen R Nutzern:
library(berryFunctions)
monthAxis()
```

- 0. Intro
- 1. Grundlagen
- 2. Datentypen
- 3. Tabellen
- 4. Grafiken

- 0.1 Willkommen
- 0.2 Fünf-Minuten Showcase
- 0.3 Einrichtung: R & Rstudio
- 0.4 Übungsaufgaben



- ▶ Programmiersprache für Datenanalyse / -visualisierung und Statistik in Forschung und Wirtschaft
- ▶ kostenlos, open source (nachvollziehbar, erweiterbar)
- ▶ große Nutzer-community (viele Methoden)
- ▶ macht deine Arbeit effizient und produktiv



R Studio

- ▶ Integrated Development Environment (IDE) für R:
Umgebung zur Entwicklung von Code, mit R integriert

Installation

- ▶ Anleitung
- ▶ Für den Kurs bitte eine aktuelle R Version (> 4.0) nutzen

Übersicht Rstudio

The screenshot shows the RStudio interface with several key components highlighted:

- Skripte (.R Dateien)**: A callout bubble points to the code editor pane where an R script named "analysis.R" is open, containing code related to a calculator and height calculations.
- Objekte**: A callout bubble points to the Environment pane, which displays the variable "height" with the value "183".
- Grafiken**: A callout bubble points to the Plots pane, which displays a scatter plot of data points.
- Die Konsole zum tatsächlichen R**: A callout bubble points to the Console pane, which shows the output of running the R code, including the calculation of $7 * 6$ resulting in [1] 42, and the assignment of height to 183.

Skript vs Console

The screenshot shows the RStudio interface. In the top-left, there's a script editor with two files open: 'analysis.R' and 'r_5min.R'. The code in 'r_5min.R' is:

```
1 # R in 5 minutes
2 # Berry Boessenkool
3
4 # R as calculator:
5 7 * 6
6
7 # create objects for later
8 height <- 183
9 height - 5
10
11
12
13
14
15
16
```

In the bottom-left, the 'Console' tab is active, showing the output of running the code:

```
C:/Dropbox/R/R.in.5.Min/ > 7 * 6
[1] 42
> height <- 183
> height - 5
[1] 178
>
```

A large orange callout bubble in the center says 'Skript: Code schreiben'.

In the bottom-right, the 'Plots' tab is active, displaying a scatter plot with colored data points. A second orange callout bubble says 'Code wird hier ausgeführt + Ergebnisse angezeigt'.

Code auch in Console möglich

The screenshot shows the RStudio interface. In the top-left, there are two tabs: 'analysis.R' and 'r_5min.R'. The 'analysis.R' tab contains the following R code:

```
1 # R in 5 minutes
2 # Berry Boessenkool
3
4 # R as calculator:
5 7 * 6
6
7 # create objects for later usage:
8 height <- 183
9 height - 5
10
11
12
13
14
15
16
```

In the bottom-left 'Console' tab, the user has run the command `> 7 * 6`, which returned the result `[1] 42`. They then assigned `height <- 183` and checked it again with `> height`, getting `[1] 178`.

A large blue callout bubble contains the text "länger behalten" (keep longer) above "kurz was prüfen" (check something quickly).

The top-right panel shows the 'Environment' pane with a variable named 'height' having the value 183.

The bottom-right panel displays a scatter plot with data points colored by value, showing a positive correlation between the x and y axes.

Zeilen ausführen

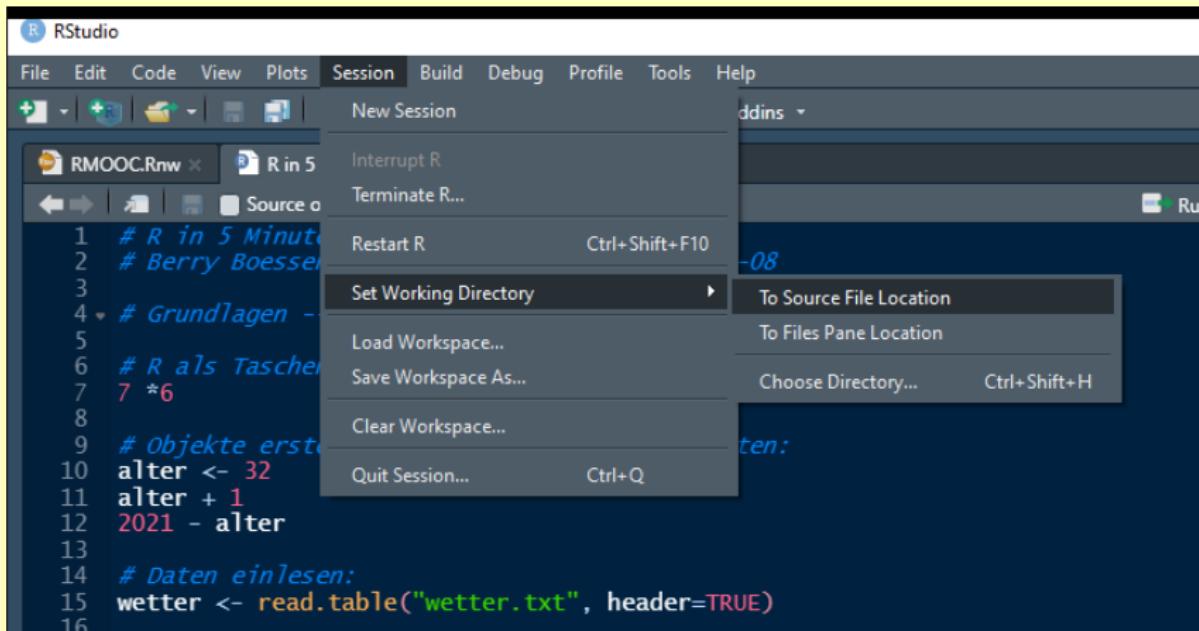
The screenshot shows the RStudio interface. In the top-left, there are two script files: 'analysis.R' and 'r_5min.R'. The 'Run' and 'Source' buttons in the toolbar above the editor area are highlighted with a red circle. A callout bubble contains the text 'Code an R schicken:' followed by two keyboard shortcuts: 'STRG + Enter: Auswahl / Zeile' and 'STRG + Shift + S: ganzes Skript'. The main workspace shows a scatter plot with colored data points. The bottom-left pane is the 'Console' tab, which displays the following R session:

```
> 7 * 6
[1] 42
> height <- 183
> height - 5
[1] 178
> |
```

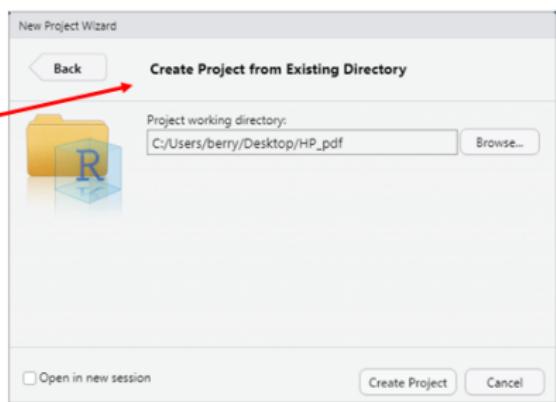
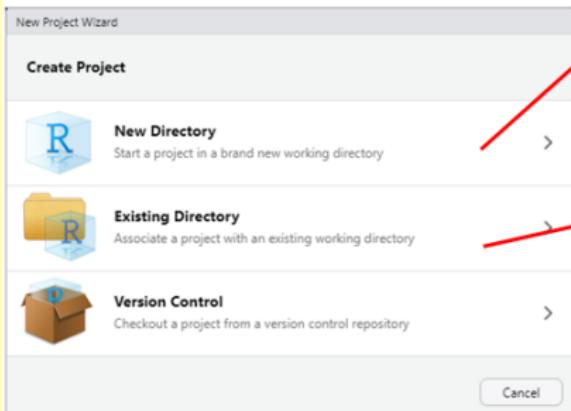
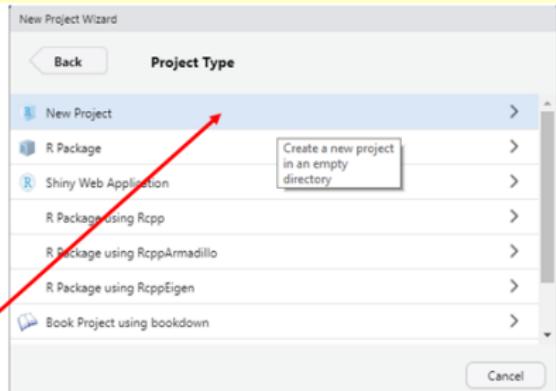
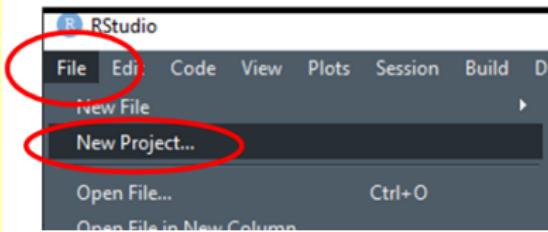
Einrichtung der R Arbeitsumgebung:

- ▶ R & Rstudio installieren und nutzen
- ▶ Skripte
- ▶ Zeilen ausführen

Für einfache Skripte:



R-projekte machen alles einfacher!



Soll deine Arbeit reproduzierbar sein, setze unter
Rstudio - Tools - Global Options - General

OFF: Restore .Rdata into workspace at startup
Save workspace to .RData on exit: **NEVER**

Weitere hilfreiche **Rstudio Einstellungen**

Im Folgenden einige praktische Tastaturkürzel aus
rviews.rstudio.com/categories/tips-and-tricks

Rstudio keyboard shortcuts (**ALT + SHIFT + K**)

STRG + **ENTER** im Skript: Zeile / Auswahl an R senden

UP / **DOWN** in der Console: letzte Befehle

STRG + **UP** Befehlsgeschichte

STRG + **SHIFT** + **P** vorherige Auswahl (mit Änderungen) senden

STRG + **SHIFT** + **S** / **ENTER** Gesamtes Skript ausführen

STRG + **SHIFT** + **N** Neue Skriptdatei anlegen

STRG + **O** / **S** Datei öffnen / Skript speichern

STRG (+ **SHIFT**) + **TAB** nächstes (vorheriges) Skript

ALT + Maus für Mehrzeilen-Cursor. oder: **STRG** + **ALT** + **UP** / **DOWN**

Windows Bildschirmrotation ausschalten: Desktop Rechtsclick - Grafikoptionen - Tastenkombinationen - deaktivieren

ALT + **UP** / **DOWN** Zeile im Skript verschieben

STRG + **1** Cursor auf Panel (1:source, 2:console, 3:help, 4:history, 5:files, 6:plot...)

STRG + **SHIFT** + **1** / **2** / **3** / ... Panel Vollbild

STRG + **SHIFT** + **C** Zeile auskommentieren

STRG + **SHIFT** + **O** Inhaltsübersicht (# abschnitt ----)

STRG + **SHIFT** + **M** pipe operator einfügen

- ▶ Working directory (Arbeitsverzeichnis)
- ▶ Projekte
- ▶ Effektiv arbeiten in Rstudio

- 0. Intro
- 1. Grundlagen
- 2. Datentypen
- 3. Tabellen
- 4. Grafiken

- 0.1 Willkommen
- 0.2 Fünf-Minuten Showcase
- 0.3 Einrichtung: R & Rstudio
- 0.4 Übungsaufgaben

Übungen auf CodeOcean - Zugang

Zu jeder Lektion in diesem Kurs gibt es interaktive Code-übungen ($n=25$), die aus 5 - 15 Aufgaben mit steigender Komplexität bestehen. Sie sind über die openHPI-Plattform zugänglich (nicht per URL direkt, da der Login über openHPI erfolgt).

Exercise 1

[Edit item](#) [Statistics](#)

Instructions:

Click the button below to launch the exercise.

 This is a graded exercise.  10.0 points

 [Launch exercise tool](#)

Sie können im Browser gelöst werden oder (besser): heruntergeladen und in Rstudio gelöst werden.

Für jede Woche gibt es eine unbewertete Spielwiese mit dem Code der Folien.

Übungen auf CodeOcean - ausführen

Sobald die Aufgabe geöffnet ist, kannst du im Skript schreiben und es ausführen durch Klicken auf 'Ausführen' (**ALT** + **R**)

The screenshot shows the CodeOcean exercise interface for an R exercise. At the top, there's a browser header with the URL <https://codeocean.openhpi.de/exercises/721/implement>. Below the header, the CodeOcean logo and navigation links for Administration, English, Help, and user profile are visible. The main content area has a breadcrumb trail: EXERCISES / R EXERCISE MASTER TEMPLATE / IMPLEMENT. A title 'R exercise master template' is followed by a progress bar at 0% completion. A keyboard shortcut 'ALT + r' is highlighted with a tooltip. On the left, a sidebar titled 'Collapse Action Sidebar' contains a 'Files' section with files: examples_1.R, examples_2.R, and t_dataset.txt. The main workspace shows R code for task 1 and task 2, and a command-line interface showing the execution of the script. The right side features a 'Collapse Output Sidebar' button.

```
1 # Structure of task files
2
3 # Task 1 -----
4 # Create an object named 'my_first_object' with the
5 # number 99.
6
7 # Make sure to save the script (CTRL + S) before running
8 # the following:
9 # You can conveniently save + source (= run full script
# including previous line)
10 # by clicking on "Source" (topright if script window) or
# pressing CTRL + SHIFT + S.
11
12 |
13
14 # Task 2 -----
15 # Create another object with the integer values from 5
# till 15.
16 # The desired objectnames is already included for your
# convenience.
17 # Replace the zero with the intended code for the
# solution.
18 my_second_object <- 5:15
19 my_second_object
20
21
22 # Now continue in examples_2.R
23
```

```
> codeoceanR::rt_score()
NULL
> my_second_object <- 5:15
> my_second_object
[1] 5 6 7 8 9 10 11 12 13 14 15
```

Übungen auf CodeOcean - bewerten

So oft du willst kannst du deine Lösungen überprüfen lassen durch Klicken auf 'Bewerten' (ALT + S)

ster template

The screenshot shows the CodeOcean interface. At the top right, there's a red 'HPI' logo. In the top right corner of the main area, there's a '100%' button and a '(Show)' link. Below the toolbar, a message says 'Keyboard shortcut: ALT + s'. The toolbar has three buttons: 'Run', 'Score' (which is highlighted in orange), and 'Request Comments'. To the right of the toolbar is a 'Collapse Output Sidebar' button. The main area contains a code editor with the following R script:

```

1 # Structure of task files
2
3 # Task 1 -----
4 # Create an object named 'my_first_object' with the
5 # number 99.
6 my_first_object <- 99
7
8 # Make sure to save the script (CTRL + S) before running
# the following:
9 codeoceanR::rt_score()
10 # You can conveniently save + source (= run full script
# including previous line)
11 # by clicking on "Source" (topright if script window) or
# pressing CTRL + SHIFT + S.
12
13
14 # Task 2 -----
15 # Create another object with the integer values from 5
# till 15.
16 # The desired objectnames is already included for your
# convenience.
17 # Replace the zero with the intended code for the
# solution.
18 my_second_object <- 5:15
19
20
21 # Now continue in examples_2.R
22

```

To the right of the code editor is a 'Results' section. It says '1 test files have been executed.' Below this is a green box titled 'Test File 1 (examples_tests.R)'. Inside the box, there are several status items:

- Passed Tests**: 8 out of 8
- Score**: 8 out of 8
- Feedback**: Well done. All tests have been passed.
- Error Messages**

At the bottom of the results section, it says 'Score: 100%' next to a green progress bar.

Bewerte oft, da die Meldungen immer spezifischer werden, je näher du der gewünschten Lösung kommst. Übertrage am Ende den Punktestand an openHPI durch Klicken auf "Code zur Bewertung abgeben".

Übungen in Rstudio - Einrichtung

Du kannst die Übungen in Rstudio lösen und dadurch in der normalen R-Arbeitsumgebung arbeiten, eine Zeile / Auswahl von Code ausführen, die Autovervollständigung nutzen, teilweise offline arbeiten (außer beim Bewerten), Tastaturkürzel verwenden, Debugging-Tools genießen, integrierte Grafikausgaben sowie Hilfe, Paketverwaltung, Versionskontrolle und mehr erhalten.

Hierfür musst du unser Paket installieren. Dazu Folgendes kopieren und ausführen:

```
install.packages("remotes")
remotes::install_github("openHPI/codeoceanR")
```

Dies ist nur einmal erforderlich und sollte weniger als eine Minute dauern.

Auf Linux zuerst `curl` und `openssl` installieren, siehe [Hinweise](#)
(ggf. Anleitung befolgen, z.B. `sudo apt install libcurl4-openssl-dev`):

```
install.packages("curl")
install.packages("openssl")
```

Für jede Übung:

- ▶ 1. via OpenHPI die CodeOcean Übung öffnen
- ▶ 2. downloaden, in geeignetem Ordner auf dem Rechner speichern (entpacken optional)

➤ R exercise master template

```
1 # Structure of task
2
3 # Task 1 -----
4 # Create an object i
number 99.
5
6
7 # Make sure to save
the following:
8 codeoceanR::irt_scor
9 # You can conveniently
including previous :
10 # by clicking on "Si
pressing CTRL + SH
11
```

Gelegentlich meldet CodeOcean den Fehler "Sorry, something went wrong".
-> Ignorieren, wenn der Download beginnt. Andernfalls Seite neu laden.

- ▶ 3. Die CodeOcean Registerkarte im Browser schließen (damit CO das dortige ungelöste Skript nicht automatisch speichert)

Übungen in Rstudio - Start

- 4. folgendes in R / Rstudio ausführen:

```
codeoceanR::rt_create()
```

- - bestätigen, den Browser Tab geschlossen zu haben
- - Datei auswählen (wenn entpackt, irgendeine Datei innerhalb des Ordners)

The screenshot shows the RStudio interface with the following details:

- Code Editor:** An R script named "Untitled1.R" is open. The code contains the following R commands:


```
1 install.packages("remotes")
2 remotes::install_github("openHPI/codeoceanR")
3 codeoceanR::rt_create()
4
5
```
- Console:** The output of the code execution is visible, including messages about package installation and the creation of a temporary directory.
- File Selection Dialog:** A "Select File" dialog box is displayed over the RStudio window, showing a list of files in the "Rexercises" folder. The file "R_exercise.zip" is selected.

Übungen in Rstudio - Aufgaben bearbeiten

`rt_create` sollte ein neues Rprojekt in einer separaten Rstudio-Instanz öffnen mit bereits geöffneten Übungsskriptdateien.
Kontaktiere Berry, wenn das nicht klappt.

```

1 Structure of task files
2
3 # Task 1 ----
4 # Create an object named 'my_first_object' with the number 99.
5
6
7 # Make sure to save the script (CTRL + S) before running the following:
8 codetools::rt_score()
9 # You can conveniently save + source (= run full script including previous line)
10 # by clicking on "Source" (topright of script window) or pressing CTRL + SHIFT + S.
11
12
13
14 # Task 2 ----
15 # Create another object with the integer values from 5 till 15.
16 # The desired objectname is already included for your convenience.
17 # Replace the zero with the intended code for the solution.
18 my_second_object <- 0
19
20
21 # Now continue in examples_2.R
22
1:1 (Top Level) : R Script ...

```

Console Terminal Jobs

C:\Users\berry\Desktop\Recessories\R_exercise\#
Platform: x86_64-w64-mingw32/x64 (64-bit)

R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.

R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

> |

Die Übung kann jederzeit mit der `zz_*.Rproj` Datei geöffnet werden.

Übungen in Rstudio - bewerten

`codeoceanR::rt_score()`

sendet deinen Code an CodeOcean (sichtbar bei erneutem Öffnen im Browser), führt das Testskript aus und zeigt die Rückmeldungen in Rstudio. Ganzes Skript ausführen (source): **CTRL + SHIFT + S**

```

codeoceanR::rt_score()

# A1 ----
# Plot a boxplot of the number of controls in the built-in cancer dataset
# 'esoph' without outlier distinction.
# The boxplot whiskers should cover the entire data range.
# t1_start
# t1_end
boxplot(esoph$controls, range=0)
# t1_end

# A2 ----
# Plot a histogram of the maximum daily temperatures at La Guardia Airport in the
# built-in dataset 'airquality'.
# Assign the (by default invisible) output to the object 'output_hist'.
# use 15 bins filled with the "coral" color.
# Make sure that the x-axis is between 50 and 100.
# t2_start
# t2_end
output_hist <- hist(airquality$Temp, breaks=15, xlim=c(50,100), col="coral")
# t2_end

# A3 ----
# use the probability density distribution instead of frequency (amount per bin).
# t3_start
# t3_end

```

Console output:

```

codeoceanR::rt_score()
> source('C:/Dropbox/R/RELEARNING/HPI-2021/R/exercises/prog11_grex6_Graphics2.R')
> T1: the code section t2 should contain the argument 'xlab'.
-> T6: The file 'grex6_out.pdf' does not exist.
-> T7: The correct answer for 'statements' is not given: it should have names.
-> T8: The code section t8 is empty.
8 examples, 4 passed, score: 50%
>

```

Plots section:

Histogram of airq

Bin Range (Temp)	Frequency
50-55	6
55-60	5
60-65	5
65-70	7
70-75	6
75-80	8
80-85	9
85-90	10
90-95	13
95-100	14

Zusammenfassung

interaktive Übungsaufgaben

- ▶ auf CodeOcean
- ▶ mit verstecktem Skript zum Testen deiner Lösungen
- ▶ in Rstudio: Übung runterladen, Registerkarte schließen,
`codeoceanR::rt_create()` ausführen
- ▶ mehrmals Punkte testen lassen (Score), nur einmal einreichen
(Submit)
- ▶ Scoring-Meldungen werden zunehmend spezifischer

- 0. Intro
- 1. Grundlagen
- 2. Datentypen
- 3. Tabellen
- 4. Grafiken

- 1.1 Syntax
- 1.2 Hilfe
- 1.3 Vektoren
- 1.4 Statistik

Operatoren I: Arithmetik

```
4*9  
## [1] 36
```

Code in diesen Folien ist mit grauen Kästen hinterlegt und farbig markiert (syntax-highlighting). R Ausgaben stehen hinter `##`.

`[1]` wird im Abschnitt 2.3 (Vektoren) erklärt.

Kommentare (nach einem Hashtag) werden von R ignoriert.
Sie machen Code für Menschen verständlich = leichter lesbar:

```
2 + 4.8 # Punkt als Dezimaltrennzeichen  
## [1] 6.8
```

```
2+4.8 # Leerzeichen egal für R, hilfreich für Lesbarkeit
```

```
3^2  
## [1] 9
```

```
sqrt(81) # Wurzel (square root)
## [1] 9
```

```
abs(-12) # Betrag (absolute value)
## [1] 12
```

```
log(100) # natürlicher Logarithmus (ln) mit Basis e (2.72)
## [1] 4.60517
```

```
log10(100) # Logarithmus mit Basis 10
## [1] 2
```

```
exp(3) # Exponentialfunktion e^3
## [1] 20.08554
```

Zuweisung (assignment): ein Objekt mit Daten im Workspace anlegen

```
alter <- 15.4
```

Rstudio: Tastaturtasten **ALT** + **-** drücken für **<-**

```
alter # ist jetzt im Workspace (quasi der R Speicher)
```

```
## [1] 15.4
```

```
alter + 5
```

```
## [1] 20.4
```

alter ist unverändert 15.4. Zum Ändern überschreiben:

```
alter <- 37.1
```

```
alter # immer die aktuelle Version, keine Geschichte dabei
```

```
## [1] 37.1
```

Groß-/Kleinschreibung (case) beachten:

```
Alter # ist kein existierendes Object
```

```
## Fehler: Objekt 'Alter' nicht gefunden
```

Objekte

```
ls() # Eigene Objekte im Workspace auflisten  
## [1] "alter"  
  
rm(alter) # ein Objekt löschen
```

```
pi # Eingebaute Konstante  
## [1] 3.141593
```

```
pi <- 3 # das kann überschrieben werden  
sin(pi/2) # aber dann kommt nicht mehr ganz 1 raus ...  
## [1] 0.997495
```

Empfehlung: bestehende Namen wie `pi`, `sin` nicht verwenden. Wenn ein eigenes Objekt `pi` existiert, wird nicht das eingebaute `pi` genutzt. Gute Objektnamen sind **kurz aber aussagekräftig**, zB `tempMaxBerlin`. Übliche Konventionen sind `lowerCamelStandard` und `unter_strich`. Den alten `punkt.standard` bitte nicht mehr verwenden. Der hat in anderen Programmiersprachen eine besondere Bedeutung.

Funktionen werden mit runden Klammern aufgerufen (ausgeführt):

```
log(7.4) # Funktionsaufruf
```

```
## [1] 2.00148
```

```
log(x=7.4) # explizite Benennung des Arguments
```

```
## [1] 2.00148
```

Argumente haben Namen. Diese können weggelassen werden, sofern sie in der richtigen Reihenfolge stehen.

`log` hat ein weiteres Argument `base`. Wenn das (wie bisher) nicht angegeben wird, wird 2.718 verwendet. Das ist der Standardwert für `base` (default). Für benutzerdefinierte Basis:

```
log(x=200, base=12)
```

```
## [1] 2.1322
```

Argumentnamen können abgekürzt werden, sofern sie einmalig sind:

```
log(200, b=12)
```

```
## [1] 2.1322
```

Zusammenfassung

Syntax, Objekte, Operatoren, Funktionen:

- ▶ `+ , - , * , / , ^`
- ▶ Leerzeichen und Kommentare (`#`) machen Code leichter lesbar
- ▶ `pi , sqrt , abs , log , log10 , exp`
- ▶ Objekte: `ALT + -` für Zuweisungspfeil (`<-`)
- ▶ Objektnamen Case sensitive
- ▶ `ob_jekt` oder `objekt`, nicht: `pi , daten , ...`
- ▶ `ls , rm`
- ▶ `funktion(1, argument=2, arg=3)`

Operatoren 2.0 ; Exponentialdarstellung

```
sin(15 * pi/180) # Grad zu dezimal
## [1] 0.247404

factorial(5) # Fakultät:  $n! = 1*2*3*4*\dots*n$ 
## [1] 120

exp(1) # eulersche Zahl e
## [1] 2.718282

e^3 # geht so nicht
## Fehler: Objekt 'e' nicht gefunden

3.91 * 10^-3 # wissenschaftliche Schreibweise: 3,91 E-3
## [1] 0.00391

3.91e-3 # scientific notation: keine Leerzeichen erlaubt
## [1] 0.00391

1e6 # schnell eine Million schreiben (ohne 6 Nullen)
## [1] 1e+06

options(scipen=9) # bis 1e9 wird ab jetzt ausgeschrieben
1e6 ; 1e14
## [1] 1000000
## [1] 1e+14
```

Technisch kann `=` statt `<-` für Zuweisungen verwendet werden.
Gemäß **style guide** sollte `=` nur für `Funktion(arg="wert")` verwendet werden.

`median(x <- 1:10)` erstellt auch `x` im `globalenv()` workspace,
`median(x=1:10)` nicht.

blog.revolutionanalytics.com, csgillespie.wordpress.com

- 0. Intro
- 1. Grundlagen
- 2. Datentypen
- 3. Tabellen
- 4. Grafiken
- 1.1 Syntax
- 1.2 Hilfe
- 1.3 Vektoren
- 1.4 Statistik

Dokumentation eingebauter Funktionen

`help("append") # Doku der Funktion 'append' öffnen`

`?append # Schnellvariante um weniger zu tippen`

Noch schneller: **F1** drücken, während der Cursor auf dem Befehl ist

Paket, in dem die Funktion enthalten ist

Titel & Beschreibung,
manchmal zusätzliche Info
im Abschnitt **Details** oder **Note**

Argumente, ggf. mit Defaults (Standardwerte)

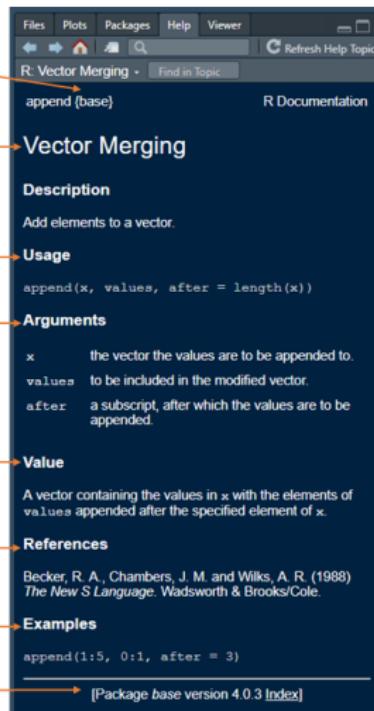
Beschreibung der Argumente

Ausgabe der Funktion (zurückgegebener Wert)

Verweise, oft auch unter **See Also**

Beispiele, oft sehr hilfreich!

Weitere Info zum Paket



(**Fn** bei Laptops)

Weitere Hilfe

```
help.search("append") # Alle verfügbaren Docs mit Bezug  
??append # auch hier wieder eine kürzere Variante  
  
help.start() # offline Handbücher und Material
```

- ▶ Kursforum für dieses MOOC
- ▶ StackOverflow für Programmierfragen <- Wichtigste Ressource
- ▶ (online) Buch: Grolemund & Wickham (2017) - R for Data Science
- ▶ Deutsches Buch: Uwe Ligges (2005) - Programmieren mit R
- ▶ Reference Card: Tom Short & Jonas Stein (2013)
- ▶ base und advanced Cheatsheets von Rstudio
- ▶ Mehr unter bookdown.org/brry/course/resources.html
- ▶ Online R Instanzen: rdrr.io/snippets, cocalc.com, colab.to/r

Antworten auf R Fragen finden:

- ▶ Dokumentationen von Funktionen aufrufen (`? / F1`)
- ▶ Stackoverflow
- ▶ Kursforum
- ▶ RefCard & Bücher

- 0. Intro
- 1. Grundlagen
- 2. Datentypen
- 3. Tabellen
- 4. Grafiken
- 1.1 Syntax
- 1.2 Hilfe
- 1.3 Vektoren
- 1.4 Statistik

Vektoren in R sind keine geometrischen Konstrukte, sondern eine geordnete Menge. (ordered set of values).

Vektoren werden erstellt mit `c` (Combine / Concatenate). Einträge werden mit einem Komma getrennt.

```
zahlen <- c(3, 7, -2.7654321, 11, 3.8, 9)
```

Objekt aufrufen (anzeigen):

```
zahlen
```

```
## [1] 3.000000 7.000000 -2.765432 11.000000 3.800000  
## [6] 9.000000
```

```
print(zahlen, digits=3) # Explizit anzeigen, mit Optionen
```

```
## [1] 3.00 7.00 -2.77 11.00 3.80 9.00
```

Vektoren II: Folgen

```
1:5 # Ganze Zahlen (integers) von : bis
## [1] 1 2 3 4 5

rep(1:4, times=3) # Zahlen mehrfach wiederholen
## [1] 1 2 3 4 1 2 3 4 1 2 3 4

rep(1:3, each=3, times=2)
## [1] 1 1 1 2 2 2 3 3 3 1 1 1 2 2 2 3 3 3

seq(from=3, to=-1, by=-0.5) # Sequenz
# Für absteigende Folgen muss 'by' negativ sein
## [1] 3.0 2.5 2.0 1.5 1.0 0.5 0.0 -0.5 -1.0

seq(1.32, 6.1, length.out=9) # 9 Elemente
## [1] 1.3200 1.9175 2.5150 3.1125 3.7100 4.3075 4.9050
## [8] 5.5025 6.1000

seq(1.32, 6.1, len=15) # Argumentnamen abkürzbar
```

Indexing: Submengen auswählen -> Eckige Klammern

```
vek <- c(3, 7, -2, 11, 4, 9)
```

```
vek[1] # Erstes Element zurückgeben  
## [1] 3
```

AltGr + 8 / 9,
Option + 5 / 6

```
vek[2:4] # Mehrere Elemente auswählen  
## [1] 7 -2 11
```

```
vek[ c(2,5,1,6,1) ] # Flexible Reihenfolge  
## [1] 7 4 3 9 3
```

```
vek[-2] # Alle Elemente außer das zweite  
## [1] 3 -2 11 4 9
```

```
vek[-(1:3)] # Alle Elemente außer den ersten drei  
## [1] 11 4 9
```

```
vek[-1:3] # geht nicht  
## Fehler in vek[-1:3]: only 0's may be mixed with negative subscripts  
-1:3      # weil -1 und 1 nicht beides erfüllt werden kann  
## [1] -1 0 1 2 3
```

head/tail, str, class, length

```
a <- seq(from=1, to=100, by=0.1)
head(a) # Die ersten 6 Elemente anzeigen
## [1] 1.0 1.1 1.2 1.3 1.4 1.5

tail(a, 8) # Die letzten 8 Elemente
## [1] 99.3 99.4 99.5 99.6 99.7 99.8 99.9 100.0

a[2] <- 87 # Einzelnes Element eines Objekts ändern
head(a) # das Objekt 'a' ist jetzt anders
## [1] 1.0 87.0 1.2 1.3 1.4 1.5

str(a) # Struktur: Datentyp, [Dimension], erste Werte
## num [1:991] 1 87 1.2 1.3 1.4 1.5 1.6 1.7 1.8 1.9 ...
class(a) # primär: numeric, logical, factor, character
## [1] "numeric"

length(a) # Länge (Anzahl Elemente) des Vektors
## [1] 991
```

```
2 * 7
```

```
## [1] 14
```

```
2:9 * 7          # 7 wird so oft wiederholt wie nötig
```

```
## [1] 14 21 28 35 42 49 56 63
```

```
2:9 * c(7,1)    # Dieses Konzept heißt "Recycling"
```

```
## [1] 14 3 28 5 42 7 56 9
```

```
2:9 * c(7,1,2) # Ergebnis mit Warnung, wenn's nicht passt
```

```
## Warning in 2:9 * c(7, 1, 2): longer object length is  
not a multiple of shorter object length
```

```
## [1] 14 3 8 35 6 14 56 9
```

Zusammenfassung

Vektoren erstellen und indizieren:

- ▶ `c` , `:` , `rep` , `seq`
- ▶ `v[n]` , `v[-n]` , `v[m:n]` , `v[-(m:n)]`
- ▶ `head` , `tail` , `str` , `class` , `length`
- ▶ Recycling

Vektoren mit Namen

```
Punktestand <- c(Christoph=19, Berry=17, "Anna Lena"=22)
```

Leerzeichen in Namen besser vermeiden

```
Punktestand[2] # Index: Position
```

```
## Berry
```

```
## 17
```

```
Punktestand["Berry"] # Index: Name
```

```
## Berry
```

```
## 17
```

```
names(Punktestand) # 'Punktestand' ist ein "named vector"
```

```
## [1] "Christoph" "Berry"      "Anna Lena"
```

```
names(Punktestand) <- LETTERS[1:3]
```

```
names(Punktestand)[2] <- "NeuerName"
```

```
Punktestand
```

```
##          A NeuerName          C
```

```
##          19           17           22
```

- 0. Intro
- 1. Grundlagen
- 2. Datentypen
- 3. Tabellen
- 4. Grafiken
- 1.1 Syntax
- 1.2 Hilfe
- 1.3 Vektoren
- 1.4 Statistik**

Statistische Maßzahlen I: Mittelwert, Streuung, Wertebereich

Vektor mit Körpergrößen:

```
groesse <- c(149.3, 173.6, 172.2, 172.9, 161.6, 179.2,  
           164.8, 162.8, 180.5, 165.1, 181.7, 171.4,  
           172.1, 148.1, 161.1, 171.9, 186.9) # cm
```

```
mean(groesse) # Mittelwert
```

```
## [1] 169.1294
```

```
var(groesse) # Varianz: cm^2
```

```
## [1] 112.591
```

```
sd(groesse) # Standardabweichung: cm
```

```
## [1] 10.61089
```

```
min(groesse) # Minimum
```

```
## [1] 148.1
```

```
max(groesse) # Maximum
```

```
## [1] 186.9
```

```
range(groesse) # Wertebereich
```

```
## [1] 148.1 186.9
```

Statistische Maßzahlen II: auch ohne Normalverteilung

```
median(groesse) # Ausreißer-unabhängig (anders als mean)
## [1] 171.9
```

```
mad(groesse)      # Median absolute deviation
## [1] 10.82298
```

```
quantile(groesse) # Anteil < bestimmte Werte
##    0%    25%    50%    75%   100%
## 148.1 162.8 171.9 173.6 186.9
```

```
quantile(groesse, probs=0.80) # 80% ist hier drunter
##    80%
## 178.08
```

```
summary(groesse)
##    Min.  1st Qu.  Median    Mean  3rd Qu.  Max.
## 148.1   162.8   171.9   169.1   173.6   186.9
```

Zeigt auch Anzahl NAs an (falls vorhanden), kann auch für Tabellen verwendet werden, siehe entsprechenden Abschnitt 3.1

Sortierungen

```
groesse <- round(groesse[1:8])      ;      groesse
## [1] 149 174 172 173 162 179 165 163
```

```
sort(groesse) # Aufsteigend sortieren
## [1] 149 162 163 165 172 173 174 179
```

```
sort(groesse, decreasing=TRUE)
## [1] 179 174 173 172 165 163 162 149
```

```
order(groesse)
## [1] 1 5 8 7 3 4 2 6
```

Das kleinste ist an Stelle 1, das zweitkleinste in `groesse[5]`, etc.

```
gewicht <- c(49, 77, 66, 91, 69, 72, 73, 74)
gewicht[order(groesse)] # Sortieren nach Reihenfolge Größe
## [1] 49 69 74 73 66 91 77 72
```

Zufallszahlen

```
sample(0:9, size=7)          # Zufällig Werte aus Vektor ziehen
## [1] 2 7 1 4 9 8 6

sample(0:9, size=7, replace=TRUE)      # Ziehen mit Zurücklegen
## [1] 5 6 9 0 7 6 5
```

Kontinuierliche Verteilungen:

```
rnorm(n=5, mean=20, sd=3.5)          # aus Normalverteilung
## [1] 21.9 19.0 20.4 19.9 11.2

rexp(n=5, rate=1/20)                 # Exponentialverteilung
## [1] 7.27 23.47 22.79 8.56 29.17

runif(n=5, min=15, max=25)          # Gleichverteilung (uniform)
## [1] 22.8 19.3 24.3 22.7 17.6

rbeta(n=5, shape1=3, shape2=9)       # Beta-verteilung
## [1] 0.1879 0.0687 0.2998 0.4172 0.1498
```

Diskrete Verteilungen:

```
rpois(n=5, lambda=20)              # Poisson-verteilung
## [1] 19 13 23 29 29

rbinom(n=5, size=100, prob=1/5)     # Binomial-verteilung
## [1] 27 16 21 27 22
```

Zusammenfassung

Statistische Maßzahlen, Sortierungen und Zufallszahlen:

- ▶ `mean` , `var` , `sd`
- ▶ `min` , `max` , `range` , `median` , `quantile` , `summary`
- ▶ `round` , `sort` , `order (decreasing)`
- ▶ `sample` , `rnorm` etc

unique und duplicated

```
val <- c(1, 7, 3, 3, 8, 5, 6, 6, 6, 7)
unique(val) # ursprüngliche Reihenfolge beibehalten
## [1] 1 7 3 8 5 6

duplicated(val)
## [1] FALSE FALSE FALSE  TRUE FALSE FALSE FALSE  TRUE
## [9]  TRUE  TRUE
duplicated(val, fromLast=TRUE)
## [1] FALSE  TRUE  TRUE FALSE FALSE FALSE  TRUE  TRUE
## [9] FALSE FALSE
```

```
werte <- c(149.3, 173.6, 172.2, 172.9, 161.6, 179.2, 164.8, 142.8)
```

```
round(werte, digits=-1) # Auf 10er runden
## [1] 150 170 170 170 160 180 160 140
round(werte, -2) # Auf 100er runden
## [1] 100 200 200 200 200 200 200 100
round(werte/5)*5 # Auf 5er runden
## [1] 150 175 170 175 160 180 165 145
```

Optionen zur Ausgabe

```
pi  
## [1] 3.141593
```

Das Verhalten von R kann in vielen Optionen angepasst werden, z.B. für den Umgang mit Warnmeldungen oder Ausgaben (printed output).

`digits` regelt, wieviele relevante Nachkommastellen angezeigt werden (bezogen auf die Größenordnung der Zahl):

```
oo <- options(digits=3) # ca 2 Nachkommastellen anzeigen  
oo # bisheriger Wert jetzt in oo (old options)  
## $digits  
## [1] 7  
  
pi  
## [1] 3.14  
  
options(oo) ; rm(oo) # Einstellungen zurücksetzen
```

```
sample(1:50, 3)
## [1] 18 45 5
sample(1:50, 3)
## [1] 37 29 22
```

Für die Folien immer wieder die gleichen "Zufallszahlen" erzeugen
-> Startpunkt für den RNG (Random Number Generator) setzen:

```
set.seed(12345)
```

```
sample(1:50, 3)
## [1] 14 16 26
```

```
set.seed(12345)
sample(1:50, 3)
## [1] 14 16 26
```

- 0. Intro
- 1. Grundlagen
- 2. Datentypen**
- 3. Tabellen
- 4. Grafiken

- 2.1 Funktionen**
- 2.2 Logik
- 2.3 Zeichenketten
- 2.4 Kategorien
- 2.5 Pakete

```
dividiere <- function(zahl, divisor=5)
{
  ausgabe <- zahl / divisor
  ausgabe <- round(ausgabe, digits=4)
  return(ausgabe)
}
```

AltGr + 7 / 0

Option + 8 / 9

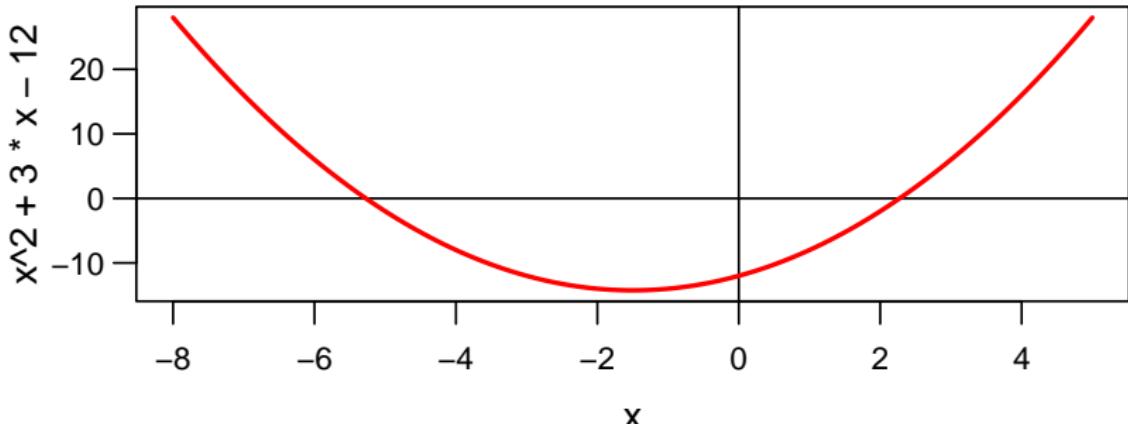
Eigene Funktion ausführen (wie andere Befehle auch):

```
dividiere(zahl=23, divisor=7) # jetzt aufrufbar
## [1] 3.2857
```

```
dividiere(23) # Standardwert (5) für 'divisor' verwendet
## [1] 4.6
```

Mit Funktionen kann die Prüfroutine in den Übungsaufgaben euren Code für verschiedene Inputs ausführen und prüfen.

Beispiel: PQ-Formel für Nullstellen einer quadratischen Funktion



```
function(p,q) # y = x^2 + px + q
{
  w <- sqrt( p^2 / 4 - q )
  c(-p/2-w, -p/2+w)
}
```

```
## [1] -5.274917 2.274917
```

```
dividiere <- function(zahl, divisor=5) # Bsp. von vorhin
{
  ausgabe <- zahl / divisor
  ausgabe <- round(ausgabe, digits=4)
  return(ausgabe)
}
```

Der Code innerhalb der Funktion (body) wird von R in einem separaten Environment ausgeführt.

Objekte innerhalb der Funktion (`zahl`, `divisor` & `ausgabe` im Beispiel) sind lokal bzw. temporär und werden nicht im normalen Workspace (global environment) angelegt.

Sie sind danach nicht im `globalenv()` verfügbar.

Return / geschweifte Klammern

`return()` beendet die Ausführung der Funktion. Code danach wird nicht aufgerufen.

`return()` kann weggelassen werden, dann wird das Ergebnis der letzten Anweisung (statement, "expression") zurück gegeben:

```
dividiere <- function(number, divisor=5){  
  output <- number / divisor  
  round(output, digits=4)  
}
```

Bei Funktionen, die nur eine einzige Anweisung (expression) enthalten, sind die geschweiften Klammern optional:

```
normalisiere <- function(x) (x-min(x)) / (max(x)-min(x))
```

von -7 bis 13 normieren auf 0 bis 1

```
normalisiere( c(8,-7,13,2,3) )
```

```
## [1] 0.75 0.00 1.00 0.45 0.50
```

Zusammenfassung

Funktionen um Code mehrfach zu verwenden:

- ▶ `meineFunktion <- function(x) {x+7}`
- ▶ Workspace: Objekte normalerweise im "global environment"
- ▶ Objekte innerhalb einer Funktion in temporärer Umgebung (environment)
- ▶ `return` am Ende von Funktionen ist optional
- ▶ `{}` sind optional, wenn die Funktion nur einen Befehl enthält

- 0. Intro
- 1. Grundlagen
- 2. Datentypen**
- 3. Tabellen
- 4. Grafiken
- 2.1 Funktionen
- 2.2 Logik
- 2.3 Zeichenketten
- 2.4 Kategorien
- 2.5 Pakete

Boolean = logical = Wahrheitswerte

```
7 > 4
## [1] TRUE      SHIFT + <
```

```
7 > 42
## [1] FALSE
```

T # = TRUE. T kann überschrieben werden, nicht nutzen

```
## [1] TRUE
```

```
! 7 > 4          # NICHT-Operator (Negierung, Gegenteil)
## [1] FALSE
```

```
TRUE & TRUE      SHIFT + 6          # UND-operator
## [1] TRUE
TRUE & FALSE
## [1] FALSE
```

```
TRUE | FALSE     AltGr + <, Option + 7  # ODER-operator
## [1] TRUE
```

Logische Operatoren I: Vektoren

```
x <- c(1, 2, 3, 4, 5)
y <- c(4, 5, 6, 7, 1)
```

Viele Operatoren sind vektorisiert, gehen also für einen ganzen Vektor:

```
x > 3
## [1] FALSE FALSE FALSE TRUE TRUE
y < 6
## [1] TRUE TRUE FALSE FALSE TRUE
x>3 & y<6      # für Vektor mit mehreren Wahrheitswerten
## [1] FALSE FALSE FALSE FALSE TRUE
x>3 | y<6
## [1] TRUE TRUE FALSE TRUE TRUE
```

`&&` und `||` evaluieren nur den ersten Wert:

```
x>3 && y<6      # für einen einzigen Wahrheitswert
## [1] FALSE
```

Logische Operatoren II: Größenvergleich

```
werte <- c(32, 28, 29, 30, 31, 32)

werte < 30
## [1] FALSE  TRUE  TRUE FALSE FALSE FALSE

werte <= 30
## [1] FALSE  TRUE  TRUE  TRUE FALSE FALSE

werte > 30
## [1]  TRUE FALSE FALSE FALSE  TRUE  TRUE

werte >= 30
## [1]  TRUE FALSE FALSE  TRUE  TRUE  TRUE

werte == 30
## [1] FALSE FALSE FALSE  TRUE FALSE FALSE

werte != 30
## [1]  TRUE  TRUE  TRUE FALSE  TRUE  TRUE
```

Logische Operatoren III: `which`, `any`, `all`

```
werte <- c(32, 28, 29, 30, 31, 32)
```

```
werte < 30
```

```
## [1] FALSE TRUE TRUE FALSE FALSE FALSE
```

```
which(werte < 30) # -> Index: Stellen mit TRUE im Vektor
```

```
## [1] 2 3
```

```
which(werte == max(werte))
```

```
## [1] 1 6
```

```
which.max(werte) # nur der erste Index!
```

```
## [1] 1
```

```
any(werte < 30) # ist mindestens eins der T/F Werte wahr?
```

```
## [1] TRUE
```

```
all(werte < 30) # sind alle TRUE?
```

```
## [1] FALSE
```

```
werte < 30
## [1] FALSE  TRUE  TRUE FALSE FALSE FALSE
as.numeric(werte < 30) # intern als Zahl: 0=FALSE, 1=TRUE
## [1] 0 1 1 0 0 0
sum(werte < 30) # Anzahl TRUEs im Vektor
## [1] 2
mean(werte < 30) # Anteil TRUE Werte
## [1] 0.3333333
```

Logische Selektion (= Subsetting, Indexing)

```
werte <- c( 32, 28, 29, 30, 31, 32)
namen <- c("a", "b", "c", "d", "e", "f")

namen[4]
## [1] "d"

werte < 30
## [1] FALSE TRUE TRUE FALSE FALSE FALSE

namen[werte < 30]
## [1] "b" "c"
```

Zum Auswählen mit logischen Werten ("Filtern") müssen beide Vektoren gleich lang sein.

Zusammenfassung

Logische Werte, Größenvergleich:

- ▶ `TRUE`, `FALSE` (nicht nutzen: `T`, `F`)
- ▶ `!`, `&`, `|`, `&&`, `||`
- ▶ `<`, `>`, `<=`, `>=`, `==`, `!=`
- ▶ `which`, `which.max`, `any`, `all`, `sum`, `mean`
- ▶ `vec[logical]`

Gleichheit nur für ganze oder gerundete Zahlen prüfen!

```
0.4 - 0.1 == 0.3          # nicht das erwartete Ergebnis!  
## [1] FALSE
```

```
print(0.4-0.1 , digits=22)  
## [1] 0.30000000000000004
```

```
round(0.4 - 0.1, digits=5) == round(0.3, digits=5)    # OK  
## [1] TRUE
```

```
all.equal(0.4 - 0.1, 0.3) # Hat eine Fehlertoleranz  
## [1] TRUE
```

Variante für Vektoren:

```
berryFunctions::almost.equal(c(6.34, 9.69, 3.77), 9.69)  
## [1] FALSE TRUE FALSE
```

Mehr solcher Tücken im **R inferno**.

```
T           # kann abgekürzt werden, allerdings:  
## [1] TRUE  
T <- 99      # kann T überschrieben werden  
T <- FALSE    # Streich: heimlich beim Kollegen eintippen ;)  
TRUE <- 77     # ist geschützt  
## Fehler in TRUE <- 77: invalid (do_set) left-hand side  
to assignment
```

```
xor(TRUE, FALSE) # EXKLUSIVES ODER (genau 1 von 2 wahr?)  
## [1] TRUE
```

```
isTRUE(T); isTRUE(F); isTRUE(NA) # T, F, F (für NAs)
```

A & B sowie A && B unterscheiden sich in einer weiteren Sache.

Wenn A falsch ist, wird in der zweiten Variante B gar nicht ausgewertet, weil die Ausgabe kein Vektor sein kann, wo noch ein TRUE auftreten könnte.

```
rechnung <- function(out){cat("Rechnung läuft\n") ; out}
```

Ausgabe beider 'rechnung'-Aufrufe:

```
rechnung(FALSE) & rechnung(TRUE)  
## Rechnung läuft  
## Rechnung läuft  
## [1] FALSE
```

Nur die linke Instanz wird ausgeführt:

```
rechnung(FALSE) && rechnung(TRUE)  
## Rechnung läuft  
## [1] FALSE
```

`&` hat Vorrang vor `|` (operator precedence, wie `*` vor `+`):

```
berryFunctions::TFtest(a|b&c, a|(b&c), (a|b)&c, na=FALSE)
##      a      b      c  __ a | b & c  __ a | (b & c)  __ (a | b) & c
## 1  TRUE  TRUE  TRUE      TRUE      TRUE      TRUE
## 2  TRUE  TRUE FALSE     TRUE      TRUE  FALSE
## 3  TRUE FALSE  TRUE     TRUE      TRUE  TRUE
## 4  TRUE FALSE FALSE     TRUE      TRUE  FALSE
## 5 FALSE  TRUE  TRUE     TRUE      TRUE  TRUE
## 6 FALSE  TRUE FALSE    FALSE     FALSE  FALSE
## 7 FALSE FALSE  TRUE    FALSE     FALSE  FALSE
## 8 FALSE FALSE FALSE   FALSE     FALSE  FALSE
```

Siehe auch [?Syntax](#)

- 0. Intro
- 1. Grundlagen
- 2. Datentypen
- 3. Tabellen
- 4. Grafiken

- 2.1 Funktionen
- 2.2 Logik
- 2.3 Zeichenketten
- 2.4 Kategorien
- 2.5 Pakete

Zeichenketten Grundlagen

```
"Moin Moin" # Anführungsstriche:  
'guten Tag' # beide Sorten möglich  
  
satz <- "Dies ist kein einheitlicher Satz"  
class(satz)  
## [1] "character"  
  
nchar(satz) # Anzahl Zeichen  
## [1] 32  
  
tolower(satz)  
## [1] "dies ist kein einheitlicher satz"  
toupper(satz)  
## [1] "DIES IST KEIN EINHEITLICHER SATZ"  
  
substr(satz, start=7, stop=16) # Leerzeichen zählen mit  
## [1] "st kein ei"  
  
cat(satz) # concatenate and type  
## Dies ist kein einheitlicher Satz
```

```
paste("Wort", 1:4) # vektorisiert: 'Wort' 4 mal recycelt  
## [1] "Wort 1" "Wort 2" "Wort 3" "Wort 4"
```

```
paste("Wort", 1:4, sep="_/") # Eigene Trennzeichenkette  
## [1] "Wort_1" "Wort_2" "Wort_3" "Wort_4"
```

```
paste0("Wort", 1:4) # Leerer charstring '' als separator  
## [1] "Wort1" "Wort2" "Wort3" "Wort4"
```

Mehrere Elemente zu einer einzigen Zeichenkette zusammenfügen:

```
paste0("Wort", 1:4, collapse="-")  
## [1] "Wort1-Wort2-Wort3-Wort4"
```

```
toString(c("Diese", 1:5, "Wörter")) # kommagetrennt  
## [1] "Diese, 1, 2, 3, 4, 5, Wörter"
```

```
satz
## [1] "Dies ist kein einheitlicher Satz"
worte <- strsplit(satz, split = " ")[[1]]
```

`strsplit` gibt eine list als Ausgabe. Um das erste Element auszuwählen (das einen Vektor mit 5 Einträgen hat), nutzen wir doppelte eckige Klammern.

```
worte
## [1] "Dies"           "ist"            "kein"
## [4] "einheitlicher" "Satz"
```

Zeichenketten suchen

```
worte
```

```
## [1] "Dies"      "ist"       "kein"      "einheitlicher" "Satz"
```

```
match("kein", worte) # Index des ersten gleichen Eintrages
```

```
## [1] 3
```

```
match("ei", worte)          # nur komplette Übereinstimmungen
```

```
## [1] NA
```

```
"kein" %in% worte      # Logischer Wert, ob Eintrag vorkommt
```

```
## [1] TRUE
```

```
grep("ei", worte)        # in welchen Elementen 'ei' vorkommt
```

```
## [1] 3 4
```

```
grep("ei", worte, value=TRUE) # Worte, die 'ei' enthalten
```

```
## [1] "kein"      "einheitlicher"
```

```
grepl("ei", worte)    # für jedes Wort: ist 'ei' enthalten?
```

```
## [1] FALSE FALSE  TRUE  TRUE FALSE
```

```
worte
## [1] "Dies"      "ist"       "kein"      "einheitlicher" "Satz"

# Ersetze jeweils den ersten Fund:
sub(pattern="ei", replacement="EI", x=worte)
## [1] "Dies"      "ist"       "kEIn"
## [4] "EInheitlicher" "Satz"

# Ersetze alle Vorkommnisse (Auftreten) von 'ei'
gsub(pattern="ei", replacement="EI", x=worte)
## [1] "Dies"      "ist"       "kEIn"
## [4] "EInhEItlicher" "Satz"
```

Zusammenfassung

Zeichenketten (Character strings):

- ▶ "zeichen", 'kette', nchar, tolower, toupper
- ▶ paste (sep,collapse), paste0, toString
- ▶ substr, strsplit
- ▶ match, %in%, grep, grepl
- ▶ sub, gsub

Zeichenketten: Sonderzeichen Backslash

Ein Backslash signalisiert, dass danach was besonderes kommt.

```
cat("Satz mit\nZeilenumbruch") # \newline
## Satz mit
## Zeilenumbruch
cat("1\t9","1234\t9","12345678\t9", sep="\n") # \tabstop
## 1      9
## 1234   9
## 12345678       9
```

AltGr + ß,

Option + Shift + 7

```
cat("Satz mit \" Symbol") # Anführungsstrich
## Satz mit " Symbol
cat('Satz mit " Symbol') # weniger Tipparbeit :)
## Satz mit " Symbol
```

```
cat("Satz mit \\ literal") # Backslash selbst
## Satz mit \ literal
```

```
cat("Zeichenkette mit \U{0B00} Grad Symbol") # \Unicode
## Zeichenkette mit ° Grad Symbol
```

Warum `strsplit` eine Liste ausgibt

```
v <- c("ab-cdefg-hij-k-lmn", "opqrstuvwxyz")
v
## [1] "ab-cdefg-hij-k-lmn" "opqrstuvwxyz"

w <- strsplit(v, split="-")
w # Ein Element für jedes Element im Ursprungsvektor
## [[1]]
## [1] "ab"      "cdefg"   "hij"     "k"       "lmn"
##
## [[2]]
## [1] "opqrstuvwxyz" "wxyz"

w[[1]]
## [1] "ab"      "cdefg"   "hij"     "k"       "lmn"
```

```
worte
## [1] "Dies"      "ist"       "kein"      "einheitlicher" "Satz"

# für jedes Wort: an welcher Stelle 'ei' anfängt
c(regex("ei", worte)) # -1 wenn nicht drin
## [1] -1 -1  2  1 -1
gregexpr("ei", worte) # ditto: alle Stellen -> list
## -1      -1      2      1, 5      -1
```

regulärer Ausdruck (regex): Großschreibung, Anfang / Ende

grep: global search for a regular expression, print out matched lines

```
x <- c("abz", "Abz", "yzab", "abyz", "nichts")
```

```
grep("ab", x, v=T) # value=TRUE abgekürzt für kurze Folien
## [1] "abz"   "yzab"  "abyz"
```

```
grep("ab", x, v=T, ignore.case=TRUE) # Großschreibung egal
## [1] "abz"   "Abz"   "yzab"  "abyz"
```

```
grep("^ab", x, v=T) # caret: Muss anfangen mit
## [1] "abz"   "abyz"
```

```
grep("yz", x, v=T)
## [1] "yzab"  "abyz"
```

```
grep("yz$", x, v=T) # dollar: Muss enden mit
## [1] "abyz"
```

Siehe auch: `startsWith` und `endsWith`

regulärer Ausdruck (regex): Wildcards

```
x <- c("cfu", "cfgu", "cfghu", "cnu", "cmu")
```

```
grep("c.u", x, v=T) # . : irgendein beliebiges Zeichen
## [1] "cfu" "cnu" "cmu"
```

```
grep("c.*u", x, v=T) # .*: egal wieviele beliebige Zeichen
## [1] "cfu"   "cfgu"  "cfghu" "cnu"   "cmu"
```

```
grep("c.{2}u", x, v=T) # .{2}: genau 2 beliebige Zeichen
## [1] "cfgu"
```

```
grep("c(f|n)u", x, v=T) # (x/y): x oder y
## [1] "cfu" "cnu"
```

```
grep("c[kmf]u", x, v=T) # [xyz]: irgendeins dieser Zeichen
## [1] "cfu" "cmu"
```

```
grep("c[^km]u", x, v=T) # [^xyz]: nicht diese Zeichen (normal ^ anfang)
## [1] "cfu" "cnu"
```

```
grep("c[k-o]u", x, v=T) # [a-zA-Z]: zwischen a und X
## [1] "cnu" "cmu"
```

regulärer Ausdruck (regex): Wiederholungen

repetition quantifiers für Häufigkeit des vorangehenden items:

```
x <- c("cd", "cxd", "cxxd", "cxxxd", "cxxxxd")
```

```
grep("cxd", x, v=T)
```

```
## [1] "cxd"
```

```
grep("cx?d" , x, v=T) # ?: 0 oder 1 mal
```

```
## [1] "cd"   "cxd"
```

```
grep("cx*d" , x, v=T) # *: 0 oder mehrfach
```

```
## [1] "cd"   "cxd"   "cxxd"   "cxxxd"   "cxxxxd"
```

```
grep("cx+d" , x, v=T) # +: einmal oder öfter
```

```
## [1] "cxd"   "cxxd"   "cxxxd"   "cxxxxd"
```

```
grep("cx{2}d" , x, v=T) # {n}: n mal
```

```
## [1] "cxxd"
```

```
grep("cx{2,}d" , x, v=T) # {n,}: n mal oder öfter
```

```
## [1] "cxxd"   "cxxxd"   "cxxxxd"
```

```
grep("cx{2,3}d" , x, v=T) # {n,m}: n bis m mal
```

```
## [1] "cxxd"   "cxxxd"
```

regulärer Ausdruck (regex): Regex Operatoren ignorieren

```
". \ | ( ) [ { ^ $ * + ?" # regex metacharacters
```

```
x <- c("ab.de", "abde", "a^bcde", "bcde")
```

```
grep("^bc", x, value=TRUE)  
## [1] "bcde"
```

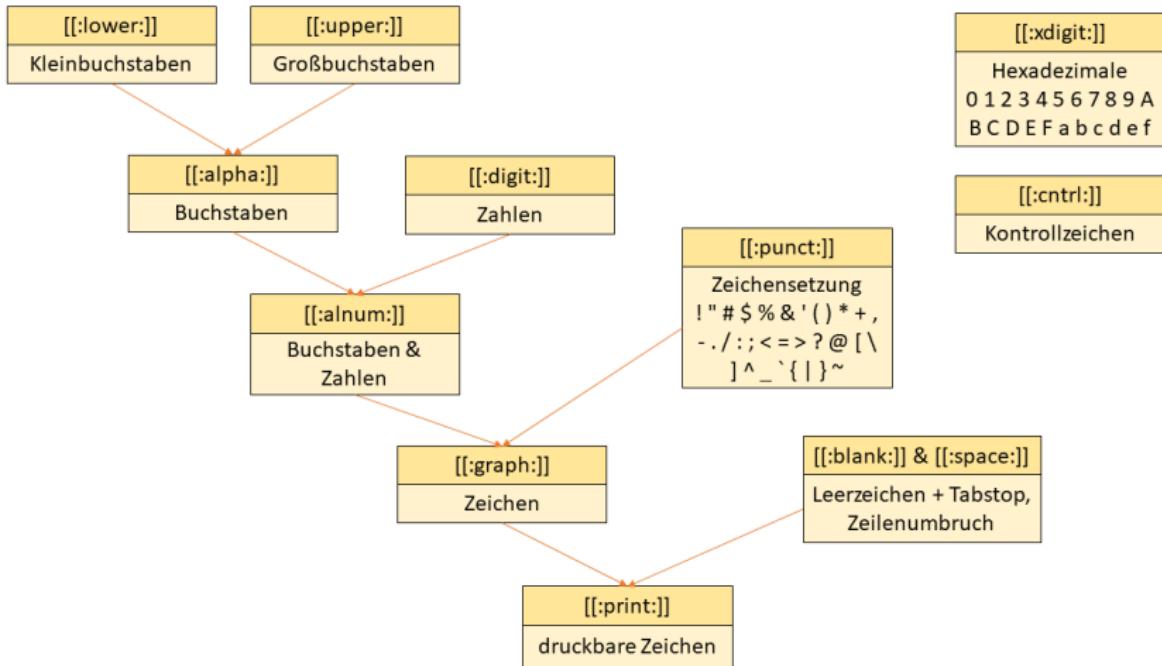
```
grep("^bc", x, value=TRUE, fixed=TRUE) # ohne regex  
## [1] "a^bcde"
```

```
grep(".de", x, value=TRUE)  
## [1] "ab.de" "abde" "a^bcde" "bcde"
```

```
grep("\\.de", x, value=TRUE) # echter Punkt (mit regex)  
## [1] "ab.de"
```

Fertige Sammlungen von Zeichenketten

```
grep(" [UV[:digit:]WX] ", c("ab3d", "abUd", "abcd"), v=T)  
## [1] "ab3d" "abUd"
```



- 0. Intro
- 1. Grundlagen
- 2. Datentypen**
- 3. Tabellen
- 4. Grafiken
- 2.1 Funktionen
- 2.2 Logik
- 2.3 Zeichenketten
- 2.4 Kategorien**
- 2.5 Pakete

Factors = kategoriale Variablen

```
factor(c("Boot", "Auto", "Zug", "Auto", "Boot"))
## [1] Boot Auto Zug  Auto Boot
## Levels: Auto Boot Zug
```

Standardmäßig alphabetisch sortiert. Für ordinalskalierte Kategorien:

```
factor(c("Boot", "Auto", "Zug", "Auto", "Boot"),
       levels=c("Boot", "Zug", "Auto"))
## [1] Boot Auto Zug  Auto Boot
## Levels: Boot Zug Auto
```

```
state.region[37] # eingebauter Datensatz mit factors
## [1] West
## Levels: Northeast South North Central West
```

```
class(state.region)
## [1] "factor"

levels(state.region)
## [1] "Northeast"      "South"          "North Central"
## [4] "West"
```

Häufigkeitstabellen

```
table(state.region) # Anzahl Vorkommen pro Wert
## state.region
##      Northeast          South       North      Central        West
##             9              16              12              13

noten <- c(3,5,2,3,1,2,2,3,5,2,2,2,1,3,4,4,2,4,3,6,3,1)
table(noten, dnn=NULL) # dnn: Dimensionsnamen weglassen
## 1 2 3 4 5 6
## 3 7 6 3 2 1

names(table(noten))
## [1] "1" "2" "3" "4" "5" "6"

geschlecht <- c("m", "m", "m", "w", "d", "w", "w", "m", "m", "d", "w",
               "d", "w", "m", "w", "w", "w", "w", "w", "w", "d", "m", "w")
table(geschlecht, noten) # Kreuztabelle (contingency table)
##            noten
## geschlecht 1 2 3 4 5 6
##             d 1 2 0 0 0 1
##             m 0 1 4 0 2 0
##             w 2 4 2 3 0 0
```

tagged (grouped) apply: eine Funktion gruppenweise anwenden

```
head(state.name) # Zeichenkette
## [1] "Alabama"      "Alaska"       "Arizona"      "Arkansas"
## [5] "California"   "Colorado"

head(state.region) # Kategorie (factor)
## [1] South West  West  South West  West
## Levels: Northeast South North Central West

nchar(state.name[1:6])
## [1] 7 6 7 8 10 8

tapply(X=state.name, INDEX=state.region, FUN=nchar)
## $Northeast
## [1] 11 5 13 13 10 8 12 12 7
## $South
## [1] 7 8 8 7 7 8 9 8 11 14 8 14 9 5 8 13
## $`North Central`
## [1] 8 7 4 6 8 9 8 8 12 4 12 9
## $West
## [1] 6 7 10 8 6 5 7 6 10 6 4 10 7
```

tagged (grouped) apply: aggregieren

```
mean_charlen <- function(x) mean(nchar(x))

mean_charlen(state.name)
## [1] 8.44

tapply(X=state.name, INDEX=state.region, FUN=mean_charlen)
##      Northeast          South North Central           West
##      10.111111      9.000000     7.916667     7.076923
```

Wenn die Funktion immer genau einen einzigen Wert ausgibt, simplifiziert `tapply` das Ergebnis.

Hier wird die Dimension auf einen Vektor* mit 4 Werten reduziert.
*: technisch gesehen ein Array

Für kleine, temporäre Sachen muss die Funktion nicht separat erstellt werden, sondern kann namenslos verwendet werden (anonymous function):

```
tapply(X=state.name, INDEX=state.region,
        FUN=function(x) mean(nchar(x)) )
```

Internals und Lesetipp

Intern sind Kategorien als Zahlen hinterlegt:

```
as.numeric(state.region)[c(37, 7, 27)]  
## [1] 4 1 3
```

Falls Zahlen als Factors eingelesen werden, würde `as.numeric(x)` die Levels geben, erst `as.numeric(as.character(x))` die eigentlichen Zahlen:

```
as.numeric(as.factor(19:17))  
## [1] 3 2 1  
  
# NICHT 19,18,17, sondern deren Levels
```

Mehr Details zu factors in [Advanced R](#).

Kategoriale Variablen in R:

- ▶ `factor`, `levels`
- ▶ `table` für Häufigkeitstabelle
- ▶ `tapply` (Werte, Kategorien, Funktion)
- ▶ Praktisch um Daten zu gruppieren, zB farblich in Grafiken
- ▶ intern als Integers hinterlegt, vorsicht mit `as.numeric`

- 0. Intro
 - 1. Grundlagen
 - 2. Datentypen**
 - 3. Tabellen
 - 4. Grafiken
- 2.1 Funktionen
 - 2.2 Logik
 - 2.3 Zeichenketten
 - 2.4 Kategorien
 - 2.5 Pakete**

- ▶ Viele R Nutzer schreiben Code für spezifische Aufgaben.
- ▶ Wenn das auch für andere nützlich sein könnte, wird das oft in einem R Paket verpackt. Das ist eine vorgegebene Form für Quellcode inkl. Dokumentation, Anleitung und Beispiele.
- ▶ Wenn umfangreiche Auflagen erfüllt sind, kann das auf CRAN veröffentlicht werden (Comprehensive R Archive Network).
- ▶ Dort sind >18'000 **Pakete** verfügbar, siehe **CRAN Task Views**

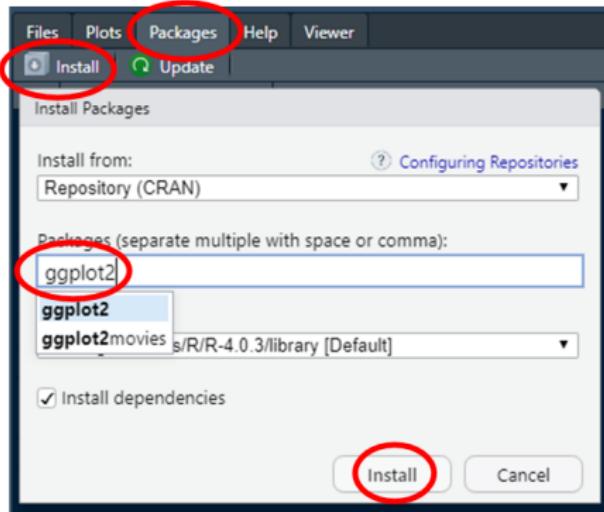
R Pakete installieren

Ein R Package downloaden und installieren:

```
install.packages("ggplot2")
```

Muss nur einmal ausgeführt werden, braucht keine admin Rechte.

Kann auch manuell in Rstudio gemacht werden:



Selten benötigt: Rückgängig machen mit

```
remove.packages("packagename")
```

R Pakete nutzen

Ein Paket aus der lokalen Bibliothek (library) laden:

```
library("ggplot2")
```

Ist in jeder neuen R Session benötigt.

Sollte als Code im Skript stehen, damit das Skript reproduzierbar ist.

Danach sind die Funktionen aus dem Paket direkt nutzbar.

Bei gleichnamigen Funktionen in mehreren Paketen wird das verwendet, welches zuletzt geladen wurde.

Damit klar ist, aus welchem Package eine Funktion verwendet wird, ist die `Paket::Funktion()` Struktur

```
rdwd::findID("Potsdam")
```

eindeutiger (und sicherer) als

```
library(rdwd)
```

```
findID("Potsdam")
```

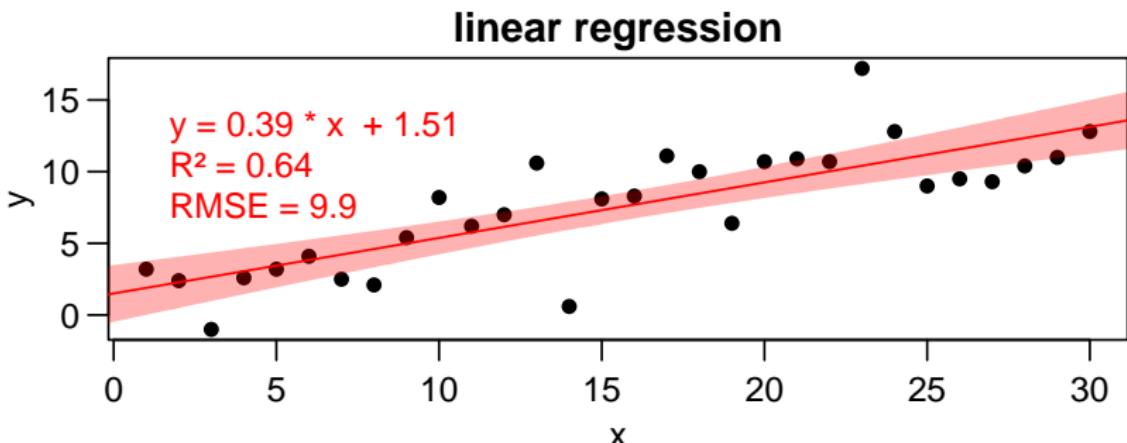
Ein guter Tipp: regelmäßig updaten - mit dem Rstudio button oder
`update.packages()`

R Pakete bei Bedarf automatisch installieren

```
# Wenn ein Paket nicht verfügbar ist, dann installiere es:  
if(!requireNamespace("berryFunctions", quietly=TRUE))  
  install.packages("berryFunctions")
```

Mehr unter bookdown.org/brry/course/packages

```
x <- 1:30  
y <- c(3.2, 2.4, -1, 2.6, 3.2, 4.1, 2.5, 2.1, 5.4, 8.2,  
      6.2, 7, 10.6, 0.6, 8.1, 8.3, 11.1, 10, 6.4, 10.7,  
      10.9, 10.7, 17.2, 12.8, 9, 9.5, 9.3, 10.4, 11, 12.8)  
berryFunctions::linReg(x, y, pos1="topleft")
```



Immer mitgelieferte Pakete (base R):

- ▶ geladen: `base`, `datasets`, `utils`, `grDevices`, `graphics`, `stats`, `methods`
- ▶ manuell zu laden: `compiler`, `grid`, `parallel`, `splines`, `tcltk`, `tools`

Definition (Source code) einer Funktion anschauen:

- ▶ ohne Klammern aufrufen: `append` statt `append()`
- ▶ auf github öffnen: `berryFunctions::funSource` (`F7` mit `rskey`)
 - ▶ base R: github.com/wch/r-source/src/library/base (stats, parallel, ...)
 - ▶ CRAN: github.com/cran
- ▶ `head` - `UseMethod` - `methods(head)` - zB `head.matrix` ([mehr](#))
- ▶ `abs` - `.Primitive` - `do_abs` in `src/main/names.c` und `/arithmetic.c`

Online Hilfe CRAN package Funktionen (ohne installieren):

www.rdocumentation.org

R Pakete mit weiterführendem Code (auf CRAN auch geprüft):

- ▶ `install.packages` (einmalig, manuell OK)
- ▶ `library` (in jedem Script)
- ▶ `Paket::Funktion()` macht Ursprung einer Funktion deutlich
- ▶ `requireNamespace` prüft, ob ein Paket geladen werden kann
- ▶ `berryFunctions::funSource` für Quellcode auf github

- 0. Intro
- 1. Grundlagen
- 2. Datentypen
- 3. Tabellen**
- 4. Grafiken

- 3.1 DataFrames
- 3.2 Matrizen
- 3.3 Dateien einlesen
- 3.4 Zusammenführen
- 3.5 Fehldaten
- 3.6 Datenquellen

data.frames erstellen

Tabellarische Daten werden in R fast immer als `data.frame` angelegt. Jede Spalte kann einen eigenen Datentyp haben (numeric, character, factor, logical, etc).

```
?data.frame
```

```
bdf <- data.frame(Zahlen=11:14, Buchstaben=letters[1:4],  
                    Booleans=(1:4)>2)  
bdf # bdf: BeispielDataFrame  
##   Zahlen Buchstaben Booleans  
## 1     11          a     FALSE  
## 2     12          b     FALSE  
## 3     13          c      TRUE  
## 4     14          d      TRUE
```

data.frames anschauen

Große Tabellen nicht komplett anzeigen (überfüllt die Console)

-> `str` und `summary` verwenden (und auch `head` / `tail`):

```
str(bdf)
```

```
## 'data.frame': 4 obs. of 3 variables:  
##   $ Zahlen : int 11 12 13 14  
##   $ Buchstaben: chr "a" "b" "c" "d"  
##   $ Booleans : logi FALSE FALSE TRUE TRUE
```

```
summary(bdf) # Hilfreiche Info pro Spalte
```

```
##      Zahlen      Buchstaben      Booleans  
## Min.   :11.00  Length:4          Mode :logical  
## 1st Qu.:11.75  Class :character  FALSE:2  
## Median :12.50  Mode   :character  TRUE :2  
## Mean   :12.50  
## 3rd Qu.:13.25  
## Max.   :14.00
```

Untermenge (Subset): data.frame-Auswahl nach Position (Index)

Zum Indexing eckige Klammern verwenden (wie bei Vektoren), allerdings zwei Dimensionen angeben (kommagetrennt):

```
bdf[ 3 , 1 ] # Wert in der dritten Zeile, erste Spalte  
## [1] 13
```

```
bdf[ , 2 ] # Alle Zeilen, zweite Spalte (-> Vektor)  
## [1] "a" "b" "c" "d"
```

```
bdf[2, ] # Alle Spalten = komplette Zeile (-> data.frame)  
##      Zahlen Buchstaben Booleans  
## 2      12          b     FALSE
```

```
?"[ " # für die Dokumentation über Subsetting
```

```
nrow(bdf)  
## [1] 4
```

```
ncol(bdf) # Siehe auch dim(bdf)  
## [1] 3
```

data.frame Auswahl mit Spaltennamen

```
bdf[, "Booleans"]  
## [1] FALSE FALSE TRUE TRUE
```

Mit Rstudio autocompletion (**TAB**-taste): **\$** (**Shift** + **4**)

```
bdf$Booleans  
## [1] FALSE FALSE TRUE TRUE
```

```
colnames(bdf)  
## [1] "Zahlen"      "Buchstaben"   "Booleans"
```

```
colnames(bdf)[2] <- "Zeichen"      # Ändert das Objekt bdf  
bdf  
##   Zahlen Zeichen Booleans  
## 1     11      a    FALSE  
## 2     12      b    FALSE  
## 3     13      c    TRUE  
## 4     14      d    TRUE
```

Auswahl mehrerer Zeilen / Spalten

```
bdf[2:3, ] # Zeilen 2 bis 3  
##   Zahlen Zeichen Booleans  
## 2      12      b     FALSE  
## 3      13      c      TRUE
```

```
bdf[c(4,1), ] # Vektoren zum Indizieren  
##   Zahlen Zeichen Booleans  
## 4      14      d      TRUE  
## 1      11      a     FALSE
```

```
bdf[bdf$Buchstaben=="c", ] # Logische Werte: 'Filtern'  
## [1] Zahlen Zeichen Booleans  
## <0 rows> (or 0-length row.names)
```

```
bdf[-2, 2:1] # Negativ-auswahl wie bei Vektoren  
##   Zeichen Zahlen  
## 1      a     11  
## 3      c     13  
## 4      d     14
```

Spalten ändern / hinzufügen / löschen

bestehende Spalte überschreiben:

```
bdf$Zahlen <- 45:48
```

neue Spalte am Ende hinzufügen:

```
bdf$Zeit3000m <- c(12.08, 10.27, 11.79, 13.50)
```

```
bdf
```

	Zahlen	Zeichen	Booleans	Zeit3000m
## 1	45	a	FALSE	12.08
## 2	46	b	FALSE	10.27
## 3	47	c	TRUE	11.79
## 4	48	d	TRUE	13.50

Spalte entfernen:

```
bdf$Zeichen <- NULL
```

```
bdf
```

	Zahlen	Booleans	Zeit3000m
## 1	45	FALSE	12.08
## 2	46	FALSE	10.27
## 3	47	TRUE	11.79
## 4	48	TRUE	13.50

data.frame Tipps und Tricks

Spaltenauswahl

- ▶ `df[, 2]` wählt immer die zweite Spalte aus
- ▶ `df[, "name"]` ist unabhängig der Spaltenreihenfolge und von der Lesbarkeit her besser verständlich (außer man weiß genau, was die zweite Spalte beinhaltet)
- ▶ `df$name` ist weniger Tipparbeit und kann mit Autocompletion ohne Tippfehler eingegeben werden

Generelle Tipps

- ▶ Wenn `nrow(df)` `NULL` zurückgibt, könnte `df` ein Vektor sein.
- ▶ `NROW(df)` zeigt `length(df)`, wenn `df` ein Vektor ist.
- ▶ Objekte konvertieren mit `as.data.frame(theMatrix)`
- ▶ Wenn Spaltennamen mit einer Zahl anfangen, wird der Prefix "X" oder "V" (Variable) vorgesetzt. Das kann besonders beim Einlesen von Daten passieren.

Zusammenfassung

Tabellen erstellen und indizieren:

- ▶ `data.frame` , `str` , `summary`
- ▶ `nrow` , `ncol` , `colnames`
- ▶ `df[r,c]` , `df[r,]` , `df[,c]` , `df[,"cname"]` ,
`df[,c("cn1","cn2")]`
- ▶ `$cname` , `$newCol <-`

Spalte als data.frame ausgeben

```
bdf[ , "Booleans"]          # -> Vektor
## [1] FALSE FALSE  TRUE  TRUE

bdf[ , "Booleans", drop=FALSE ] # -> data.frame
##   Booleans
## 1    FALSE
## 2    FALSE
## 3     TRUE
## 4     TRUE

bdf["Booleans"]      # Ohne Komma -> data.frame ACHTUNG
```

Nicht nutzen! `objekt[index]` ist für Vektoren!
R kann das, aber der Leser deines Codes nicht.

```
rownames(bdf) <- c("Alex", "Berry", "Christoph", "Daniel")
bdf
##           Zahlen Booleans Zeit3000m
## Alex          45     FALSE    12.08
## Berry         46     FALSE    10.27
## Christoph     47      TRUE    11.79
## Daniel         48      TRUE    13.50

bdf["Berry", ]
##           Zahlen Booleans Zeit3000m
## Berry         46     FALSE    10.27
```

- ▶ `tabelle[, spalte, drop=FALSE]` wenn ein `data.frame` statt eines Vektors gewollt ist (Syntax `tabelle[spalte]` nicht nutzen)
- ▶ `rownames`

- 0. Intro
- 1. Grundlagen
- 2. Datentypen
- 3. Tabellen
- 4. Grafiken

- 3.1 DataFrames
- 3.2 Matrizen
- 3.3 Dateien einlesen
- 3.4 Zusammenführen
- 3.5 Fehldaten
- 3.6 Datenquellen

Matrizen erstellen (Tabelle eines einzigen Datentyps)

```
matrix(data=1:6 , nrow=2, ncol=3)
##      [,1] [,2] [,3]
## [1,]    1    3    5
## [2,]    2    4    6

m <- matrix(1:6 , nrow=2, ncol=3, byrow=TRUE)
m
##      [,1] [,2] [,3]
## [1,]    1    2    3
## [2,]    4    5    6

dim(m) ; nrow(m) ; ncol(m) ; length(m)
## [1] 2 3
## [1] 2
## [1] 3
## [1] 6

class(m) # zwei Klassen: array ist Überklasse von matrix
## [1] "matrix" "array"
```

Elementweise Multiplikation zweier Matrizen

```
m
##      [,1] [,2] [,3]
## [1,]    1    2    3
## [2,]    4    5    6

m * 2    # Multiplikation per Element
##      [,1] [,2] [,3]
## [1,]    2    4    6
## [2,]    8   10   12

n <- matrix(rep(0:1,each=3), ncol=3) ; n
##      [,1] [,2] [,3]
## [1,]    0    0    1
## [2,]    0    1    1

m * n                      # pro Element, auch für +, -, etc
##      [,1] [,2] [,3]
## [1,]    0    0    3
## [2,]    0    5    6
```

```
colnames(m) <- c("A", "B", "C")    # wie bei data.frames  
rownames(m) <- c("row1", "row2")  
  
m[1,1] <- 989 # Matrix ändern (manipulation)  
  
m  
##          A B C  
## row1  989 2 3  
## row2      4 5 6
```

Matrizen haben einen einzigen Datentyp für das ganze Objekt

Wird ein Element zu einer Zeichenkette geändert, werden alle konvertiert:

```
m[1,1] <- "a"  
m  
##      A     B     C  
## row1 "a"  "2"  "3"  
## row2 "4"  "5"  "6"
```

Nicht nur Spalten werden als Vektor ausgegeben (wie bei data.frames), sondern auch Zeilen:

```
m["row1", ]  
##      A     B     C  
## "a"  "2"  "3"  
  
class(m["row1", ])                                # -> vector  
## [1] "character"  
is.vector(m["row1", ])                            # nur 1 Datentyp -> downcast  
## [1] TRUE
```

Funktion anwenden auf Zeilen / Spalten einer Matrix I

```
m <- matrix(1:12, ncol=4) ; m
##      [,1] [,2] [,3] [,4]
## [1,]    1    4    7   10
## [2,]    2    5    8   11
## [3,]    3    6    9   12
```

```
rowSums(m) # + colSums. Achtung: rowsum() ist was anderes
## [1] 22 26 30
```

```
colMeans(m) # siehe auch rowMeans
## [1] 2 5 8 11
```

```
# Funktion 'median' auf Spalten anwenden:
apply(m, MARGIN=2, median)
## [1] 2 5 8 11
```

Funktion anwenden auf Zeilen / Spalten einer Matrix II

```
m
##      [,1] [,2] [,3] [,4]
## [1,]     1     4     7    10
## [2,]     2     5     8    11
## [3,]     3     6     9    12
```

MARGIN=1: x Dimension behalten:

```
apply(m, MARGIN=1, FUN=median)
## [1] 5.5 6.5 7.5
```

Weitere Argumente, die an 'median' weitergegeben werden

```
apply(m, MARGIN=1, FUN=median, na.rm=TRUE)
## [1] 5.5 6.5 7.5
```

```
apply(m, 1, function(x) sum(x==2)) # anonyme Funktion
## [1] 0 1 0
```

```
apply(m, 1, FUN=function(x) cat(toString(x), " - "))
## 1, 4, 7, 10 - 2, 5, 8, 11 - 3, 6, 9, 12 -
## NULL
```

Zusammenfassung

`matrix`: Tabelle eines einzigen Datentypes

- ▶ `matrix` (`nrow`, `ncol`, `byrow`), `ncol`, `nrow`, `dim`, `length`
- ▶ Arithmetische Operationen erfolgen per Element
- ▶ `colnames`, `rownames`
- ▶ `rowMeans`, `colSums`, `apply` (`X=mat`, `MARGIN`, `FUN`)

über die Diagonale drehen / spiegeln (Zeilen und Spalten vertauschen)

```
m
##      [,1] [,2] [,3] [,4]
## [1,]     1     4     7    10
## [2,]     2     5     8    11
## [3,]     3     6     9    12
```

`t(m) # transponieren`

```
##      [,1] [,2] [,3]
## [1,]     1     2     3
## [2,]     4     5     6
## [3,]     7     8     9
## [4,]    10    11    12
```

Matrizenmultiplikation

```
m <- matrix(1:6 , nrow=2, ncol=3, byrow=TRUE)
```

```
m
```

```
##      [,1] [,2] [,3]
```

```
## [1,]    1    2    3
```

```
## [2,]    4    5    6
```

```
n <- matrix(rep(1:5,each=3), ncol=5)
```

```
n
```

```
##      [,1] [,2] [,3] [,4] [,5]
```

```
## [1,]    1    2    3    4    5
```

```
## [2,]    1    2    3    4    5
```

```
## [3,]    1    2    3    4    5
```

```
m %*% n
```

```
##      [,1] [,2] [,3] [,4] [,5]
```

```
## [1,]    6   12   18   24   30
```

```
## [2,]   15   30   45   60   75
```

?"%*%" # für die Dokumentation Anführungsstriche setzen

Data.frames können umgewandelt werden:

```
d <- data.frame(AA=5:8, BB=6:9)
class(d)
## [1] "data.frame"

dm <- as.matrix(d)      ;  dm
##      AA  BB
## [1,]  5  6
## [2,]  6  7
## [3,]  7  8
## [4,]  8  9

class(dm)
## [1] "matrix" "array"
```

- 0. Intro
- 1. Grundlagen
- 2. Datentypen
- 3. Tabellen
- 4. Grafiken

- 3.1 DataFrames
- 3.2 Matrizen
- 3.3 Dateien einlesen
- 3.4 Zusammenführen
- 3.5 Fehldaten
- 3.6 Datenquellen

Dateien einlesen (Daten importieren)

Generelle Syntax:

```
eine_tabelle <- read.table(file="dateiname.txt")
```

Eingelesene Daten immer prüfen!

(90% der Fehler Downstream passieren, weil das ignoriert wird):

```
daten_objekt_name <- read.table("Datei.txt")
str(daten_objekt_name)
```

Häufig benötigte Argumente:

```
read.table("Datei.txt", header=TRUE, dec=",", sep="\t")
```

Häufige Argumente für `read.table`

```
read.table(file="Dateiname.txt", # kann auch URL sein
header=TRUE,      # Erste Zeile als Spaltennamen lesen
dec=",",          # Komma als Dezimaltrennzeichen
sep="_",          # Unterstrich als Spaltentrenner ("\t" -> tabstop)
fill=TRUE,         # unvollständige Zeilen mit NAs am Ende füllen
skip=12,           # Erste 12 Zeilen ignorieren (zB mit Metadaten)
comment.char="%", # Zeilen ab % ignorieren (Standard: # wie in R Code)
na.strings=c(-999, "NN"), # Kennzeichnung Fehlwerte
stringsAsFactors=FALSE, # Zeichenketten nicht in factors umwandeln
text="1,2,3",        # kleiner Beispieldatensatz im Skript
...)
```

Der `stringsAsFactors` Default ist `FALSE` seit R Version 4.0.0 (2020-04-24).

```
read.table(header=TRUE, sep=",", text="
Beispiel, Spalte
Sinn, 42
Bond, 007")
## Beispiel Spalte
## 1     Sinn     42
## 2     Bond      7
```

Alternativen zu `read.table`

- ▶ `read.csv()` : comma separated values (`read.table` mit anderen Standardwerten)
- ▶ `read.fwf()` : feste Spaltenbreiten (fixed width formatted data)
- ▶ `readLines()` : Zeilen als Zeichenketten-Vektor einlesen
- ▶ `scan()` : selten benötigt (auch im Kern von `read.table` verwendet)

Komplexe Dateien:

- ▶ `readxl::read_excel()` : Exceldateien, siehe github.com/tidyverse/readxl
- ▶ `readBin()` : für binäre Dateien
- ▶ `raster::raster` / `rgdal::readGDAL` : Geodaten (grd, asc, tif)
- ▶ `ncdf4::nc_open()` + `ncdf4::ncvar_get` : NetCDF Dateien

Häufige Fehlermeldungen mit `read.table`

<p>Error in scan(file, what, nmax, sep, dec, quote, skip, nlines, na.strings, : line _ did not have _ elements</p>	<p><code>header</code>, <code>sep</code> oder <code>fill</code> richtig angeben.</p>
<p>Warning message: In <code>read.table</code>("_.txt", ...): incomplete final line found by <code>readTableHeader</code> on 'C:_.txt'</p>	<p>Manuell am Ende der Datei einen Zeilenumbruch einfügen (ENTER). line break = line feed (LF) = newline = carriage return (CR).</p>
<p><code>str(gelesenesDataframe)</code> zeigt "factor" in manchen numerischen Spalten</p>	<p>Datei prüfen! ggf. <code>dec</code> richtig angeben</p>

Schwere Fehlermeldungen:

error: Funktion bricht ab, Lösung notwendig

warning: Funktion macht ggf. was falsches. Nur ignorieren, wenn Grund bekannt aber nicht lösbar.

Daten mit Tausenderzeichen "3,590.18" einlesen:

```
df$spalte <- as.numeric(gsub(","," ", df$spalte))
```

Riskant: Kommas manuell ersetzen:

```
df$spalte <- as.numeric(gsub(",",".", df$spalte))
```

Ordner-spezifische Dateien nutzen

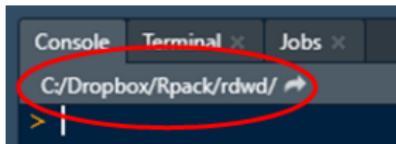
R liest und schreibt Dateien im Arbeitsverzeichnis (working directory).
Quick and dirty:

```
setwd("C:/Users/berry/Projekt_XY") # set working directory
```

Windows Nutzer: \ ist eine Unsitte von Microsoft. Beim Kopieren eines Pfades aus dem Datei-Browser (Explorer) alle ersetzen mit /.

Generell **mit Rstudio Projekten arbeiten**, siehe Folie im Abschnitt 1.2.

Aktuelles Arbeitsverzeichnis zeigen mit `getwd()`



Immer relative Dateinamen nutzen, also Dateien die im WD vorliegen:

```
tabelle <- read.table("Unterordner/Datei.txt")
```

```
dir() # Verfügbare Dateinamen im WD anzeigen
```

Ordner managen

Hilfreiche Funktionen:

```
getwd() # Aktuelles Working Directory (WD) zeigen  
setwd("../") # WD eine Ebene höher setzen  
  
dir() # Dateien / Ordner im WD auflisten  
dir(recursive=TRUE) # Auch Items in Unterordner anzeigen  
dir("../other_subfolder") # in anderem Ordner  
  
file.create() # Dateien erstellen  
file.rename() # Dateien umbenennen  
file.remove() # Dateien löschen, siehe auch unlink()  
file.copy() # Dateien kopieren  
file.exists() # Dateinamen auf Existenz prüfen  
  
dir.create() # Ordner erstellen  
dir.exists() # Ordnerpräsenz prüfen
```

```
write.table(x=mynewdata, file="output.txt",
            quote=FALSE, row.names=FALSE,
            fileEncoding="UTF-8")
```

Randnotiz: Daten niemals manuell ändern:

```
newtable <- edit(oldtable)
# oder auch kürzer:
fix(mytable) # behält nichtmal die alten Daten
```

Das wäre nicht reproduzierbar. Besser ein Skript schreiben:

```
mytable[265, "Temperatur"] <- 17.53 # original 175.3 vermutlich Typo
mytable[1:24, "Messwert"] <- NA # falsche Messmethode am ersten Tag
```

Zusammenfassung

Daten einlesen:

- ▶ `setwd`, `dir`, Rstudio projects
- ▶ `file.rename`, `file.exists`, `dir.create`, ...
- ▶ `read.table`(file, header, dec, sep, skip), `write.table`
- ▶ `read.csv`, `read.fwf`, `readxl::read_excel`, `scan`
- ▶ `str`

- 0. Intro
- 1. Grundlagen
- 2. Datentypen
- 3. Tabellen
- 4. Grafiken

- 3.1 DataFrames
- 3.2 Matrizen
- 3.3 Dateien einlesen
- 3.4 Zusammenführen**
- 3.5 Fehldaten
- 3.6 Datenquellen

cbind / rbind : Tabellen unverändert und ungeprüft zusammenfügen

`cbind`: Spalten nebeneinander zusammenlegen

`rbind`: Zeilen untereinander zusammenlegen

Die Dimension muss passen und die Spalten/Zeilennamen übereinstimmen

```
bdf <- data.frame(Zahl=11:13, Gruppe=letters[1:3])
```

```
cbind(data.frame(Poisson=rpois(3,80)), bdf) # column-bind
```

```
## Poisson Zahl Gruppe
```

```
## 1      92    11     a
```

```
## 2      74    12     b
```

```
## 3      80    13     c
```

```
zeile <- data.frame(Zahl=2, Gruppe="neu") ; zeile
```

```
## Zahl Gruppe
```

```
## 1    2    neu
```

```
rbind(bdf, zeile) # row-bind: Spaltennamen müssen gleich sein
```

```
## Zahl Gruppe
```

```
## 1    11     a
```

```
## 2    12     b
```

```
## 3    13     c
```

```
## 4    2     neu
```

Matrizen verbinden I: rbind

```
p <- matrix(11:16, ncol=3) # 2 x 3
q <- matrix(21:32, ncol=4) # 3 x 4
r <- matrix(31:39, ncol=3) # 3 x 3
```

p	q	r
11 13 15	21 24 27 30	31 34 37
12 14 16	22 25 28 31	32 35 38
23 26 29 32	33 36 39	

`rbind(p,r)` # *row-bind: Anzahl Spalten muss gleich sein*

```
##      [,1] [,2] [,3]
## [1,]    11   13   15
## [2,]    12   14   16
## [3,]    31   34   37
## [4,]    32   35   38
## [5,]    33   36   39
```

`rbind(p,q)` # *Error: 3 und 4 Spalten*

`## Fehler in rbind(p, q): number of columns of matrices must match (see arg 2)`

Matrizen verbinden II: cbind

```
p <- matrix(11:16, ncol=3) # 2 x 3
q <- matrix(21:32, ncol=4) # 3 x 4
r <- matrix(31:39, ncol=3) # 3 x 3
```

p	q	r
11 13 15	21 24 27 30	31 34 37
12 14 16	22 25 28 31	32 35 38
23 26 29 32	33 36 39	

`cbind(q,r) # column-bind: nrow muss gleich sein`

```
##      [,1] [,2] [,3] [,4] [,5] [,6] [,7]
## [1,]    21   24   27   30   31   34   37
## [2,]    22   25   28   31   32   35   38
## [3,]    23   26   29   32   33   36   39
```

`cbind(p,r) # Error: 2 und 3 Zeilen`

`## Fehler in cbind(p, r): number of rows of matrices must
match (see arg 2)`

Daten verknüpfen: merge |

```
Teilnehmer <- read.table(  
  header=TRUE, text="  
Name Alter  
Alexa 27  
Berry 32  
Chris 14  
David 45")
```

```
Probanden <- read.table(  
  header=TRUE, text="  
Person Groesse Gewicht  
Berry 1.83 82  
Chris 1.43 51  
David 1.75 72  
Erika 1.67 57")
```

Informationen aus beiden Datensätzen zusammenfügen (für Namen die in beiden Tabellen vorkommen):

```
merge(Teilnehmer, Probanden, by.x="Name", by.y="Person")  
##   Name Alter Groesse Gewicht  
## 1 Berry  32    1.83    82  
## 2 Chris  14    1.43    51  
## 3 David  45    1.75    72
```

Daten verknüpfen: `merge` II

Bei gleichen Namen erfolgt die Spaltenwahl automatisch:

```
colnames(Probanden) [1] <- "Name"
```

```
merge(Teilnehmer, Probanden)
##   Name Alter Groesse Gewicht
## 1 Berry    32     1.83     82
## 2 Chris    14     1.43     51
## 3 David    45     1.75     72
```

Alle Zeilen behalten, fehlende Angaben mit NAs füllen:

```
merge(Teilnehmer, Probanden, all=TRUE) # all.x / all.y
##   Name Alter Groesse Gewicht
## 1 Alexa   27      NA      NA
## 2 Berry    32     1.83     82
## 3 Chris    14     1.43     51
## 4 David    45     1.75     72
## 5 Erika    NA     1.67     57
```

Daten zusammenführen:

- ▶ `cbind` , `rbind`
- ▶ `merge` (`by`, `all`)

```
LIST_WITH_DFS <- list(  
  data.frame(date=1:4, AA=11:14),  
  data.frame(date=2:6, BB=22:26),  
  data.frame(date=3:7, CC=33:37)  
)  
LIST_WITH_DFS  
## [[1]]  
##   date AA  
## 1    1 11  
## 2    2 12  
## 3    3 13  
## 4    4 14  
##  
## [[2]]  
##   date BB  
## 1    2 22  
## 2    3 23  
## 3    4 24  
## 4    5 25  
## 5    6 26  
##  
## [[3]]  
##   date CC  
## 1    3 33  
## 2    4 34  
## 3    5 35  
## 4    6 36  
## 5    7 37
```

LIST_WITH_DFS

```
## [[1]]  
##   date AA  
## 1 1 11  
## 2 2 12  
## 3 3 13  
## 4 4 14
```

[[2]]
date BB
1 2 22
2 3 23
3 4 24
4 5 25
5 6 26

[[3]]
date CC
1 3 33
2 4 34
3 5 35
4 6 36
5 7 37

```
Reduce(merge, LIST_WITH_DFS)
```

```
##   date AA BB CC  
## 1 1 11 23 33  
## 2 2 12 22 NA
```

```
Reduce(function(...) merge(..., all=TRUE), LIST_WITH_DFS)
```

```
##   date AA BB CC  
## 1 1 11 NA NA  
## 2 2 12 22 NA  
## 3 3 13 23 33  
## 4 4 14 24 34  
## 5 5 NA 25 35  
## 6 6 NA 26 36  
## 7 7 NA NA 37
```

```
LIST_WITH_DFS <- lapply(LIST_WITH_DFS, function(x)
                           {colnames(x) <- c("date", "XX"); x})
```

```
LIST_WITH_DFS
```

```
## [[1]]
##   date XX
## 1   1 11
## 2   2 12
## 3   3 13
## 4   4 14
```

## [[1]]	## [2]	## [3]
## date XX	## date XX	## date XX
## 1 11	## 1 22	## 1 33
## 2 12	## 2 23	## 2 34
## 3 13	## 3 24	## 3 35
## 4 14	## 4 25	## 4 36
	## 5 26	## 5 37

```
## [[2]]
##   date XX
## 1   3 33
## 2   4 34
## 3   5 35
## 4   6 36
## 5   7 37
```

```
do.call(rbind, LIST_WITH_DFS)
```

```
##   date XX
## 1   1 11
## 2   2 12
## 3   3 13
## 4   4 14
## 5   2 22
## 6   3 23
## 7   4 24
## 8   5 25
## 9   6 26
## 10  3 33
## 11  4 34
## 12  5 35
## 13  6 36
## 14  7 37
```

- 0. Intro
- 1. Grundlagen
- 2. Datentypen
- 3. Tabellen
- 4. Grafiken

- 3.1 DataFrames
- 3.2 Matrizen
- 3.3 Dateien einlesen
- 3.4 Zusammenführen
- 3.5 Fehldaten
- 3.6 Datenquellen

Fehlende Werte (Missing values, NA=not available)

```
df <- data.frame(x=11:20, y=21:30)
df[3,2] <- NA
```

```
df
##      x  y
## 1    11 21
## 2    12 22
## 3    13 NA
## 4    14 24
## 5    15 25
## 6    16 26
## 7    17 27
## 8    18 28
## 9    19 29
## 10   20 30
```

```
is.na(df)
##            x     y
## [1,] FALSE FALSE
## [2,] FALSE FALSE
## [3,] FALSE  TRUE
## [4,] FALSE FALSE
## [5,] FALSE FALSE
## [6,] FALSE FALSE
## [7,] FALSE FALSE
## [8,] FALSE FALSE
## [9,] FALSE FALSE
## [10,] FALSE FALSE
```

```
na.omit(df)
##      x  y
## 1    11 21
## 2    12 22
## 4    14 24
## 5    15 25
## 6    16 26
## 7    17 27
## 8    18 28
## 9    19 29
## 10   20 30
```

na.rm-Argument

```
df
##      x  y
## 1  11 21
## 2  12 22
## 3  13 NA
## 4  14 24
## 5  15 25
## 6  16 26
## 7  17 27
## 8  18 28
## 9  19 29
## 10 20 30
```

```
mean(df$x)
## [1] 15.5
```

```
mean(df$y)
## [1] NA
```

Mittelwert der nicht-NA Einträge:

```
mean(df$y, na.rm=TRUE)
## [1] 25.77778
```

Nah dran am Original:

```
mean(21:30)
## [1] 25.5
```

Für Summe gefährlich (wächst mit Anzahl):

```
sum(df$y, na.rm=TRUE)
## [1] 232
```

sum(21:30) # na.rm unterschätzt Summe !!

```
## [1] 255
```

NA-Imputation: fehlende Werte mit Schätzungen füllen

Mit Mittelwert / Median / Min / Max / ... der anderen Werte füllen:

```
df$y[is.na(df$y)] <- mean(df$y, na.rm=TRUE)
```

```
df$y[is.na(df$y)] <- median(df$y, na.rm=TRUE)
```

Letzte Beobachtung fortsetzen (locf: last observation carried forwards):

```
df[3,2] <- NA # Zuweisung nicht vergessen
zoo::na.locf(df$y)
## [1] 21 22 22 24 25 26 27 28 29 30
```

Linear interpolieren:

```
approx(df$y, n=length(df$y))$y
## [1] 21 22 23 24 25 26 27 28 29 30
zoo::na.approx(df$y) # weniger Tippen :)
## [1] 21 22 23 24 25 26 27 28 29 30
```

Komplexe (multivariate) Modellierung -> Statistikkurse

Zusammenfassung

Fehldaten managen:

- ▶ `NA` , `is.na`
- ▶ `na.omit`
- ▶ `mean` / `median` / `sum` / ...(`na.rm=TRUE`)

NA-imputation:

- ▶ `x[is.na(x)] <- median(x, na.rm=TRUE)`
- ▶ `x <- zoo::na.locf(x)`
- ▶ `x <- zoo::na.approx(x)`

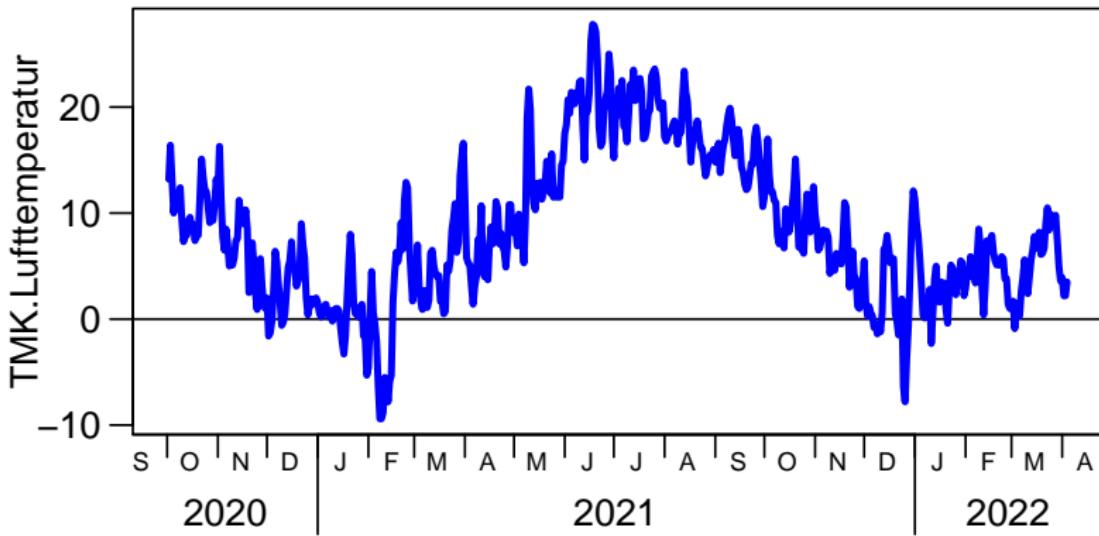
- 0. Intro
- 1. Grundlagen
- 2. Datentypen
- 3. Tabellen
- 4. Grafiken

- 3.1 DataFrames
- 3.2 Matrizen
- 3.3 Dateien einlesen
- 3.4 Zusammenführen
- 3.5 Fehldaten
- 3.6 Datenquellen

- ▶ NOAA Wetterdaten ncdc.noaa.gov/pub/data (USA)
- ▶ DWD Wetterdaten opendata.dwd.de (Deutschland), [rdwd](#)
- ▶ data.fivethirtyeight.com
- ▶ data.unicef.org (ggf .xls als Dateiendung hinzufügen)
- ▶ github.com/okulbilisim/awesome-datasience#data-sets
- ▶ github.com/caesar0301/awesome-public-datasets
- ▶ ropensci.org/packages/index.html
- ▶ dataviz.tools/category/data-sources/
- ▶ StorytellingWithData dataviz challenge 2018
- ▶ datasetsearch.research.google.com

DWD-Wetterdaten automatisiert einlesen mit rdwd

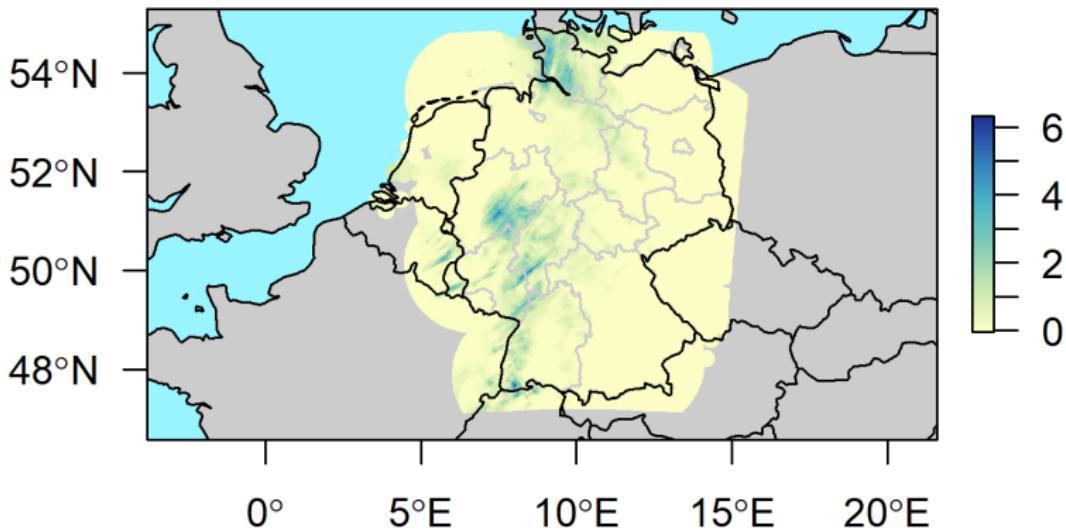
```
# install.packages("rdwd")
library(rdwd)
link <- selectDWD("Potsdam", res="daily", var="kl", per="r")
clim <- dataDWD(link, varnames=TRUE)
plotDWD(clim, "TMK.Lufttemperatur", line0=TRUE)
```



DWD-Radardaten automatisiert einlesen mit rdwd

```
link <- "hourly/radolan/historical/bin/2020/RW202004.tar.gz"  
rad <- dataDWD(link, base=gridbase, joinbf=TRUE, selection=702)  
plotRadar(rad$dat, main="2020-04-30 05:50 [mm] ")
```

2020-04-30 05:50 [mm]



Datenquellen

- ▶ Beispiele für frei verfügbare Ressourcen
- ▶ `rdwd` für Wetterdaten

Für diese Lektion gibt es keine Übungsaufgaben

- 0. Intro
- 1. Grundlagen
- 2. Datentypen
- 3. Tabellen
- 4. Grafiken

- 4.1 Punktdiagramme
- 4.2 Liniendiagramme
- 4.3 Balkendiagramme
- 4.4 Hinzufügen
- 4.5 Komposition
- 4.6 Verteilungsplots
- 4.7 Exportieren
- 4.8 Ausblick

iris: eingebauter Datensatz über Blütenblätter von Schwertlilien



Quellen: mirlab.org, desirableplants.com, 3.bp.blogspot.com

```
head(iris)
##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
## 1          5.1         3.5          1.4         0.2  setosa
## 2          4.9         3.0          1.4         0.2  setosa
## 3          4.7         3.2          1.3         0.2  setosa
## 4          4.6         3.1          1.5         0.2  setosa
## 5          5.0         3.6          1.4         0.2  setosa
## 6          5.4         3.9          1.7         0.4  setosa

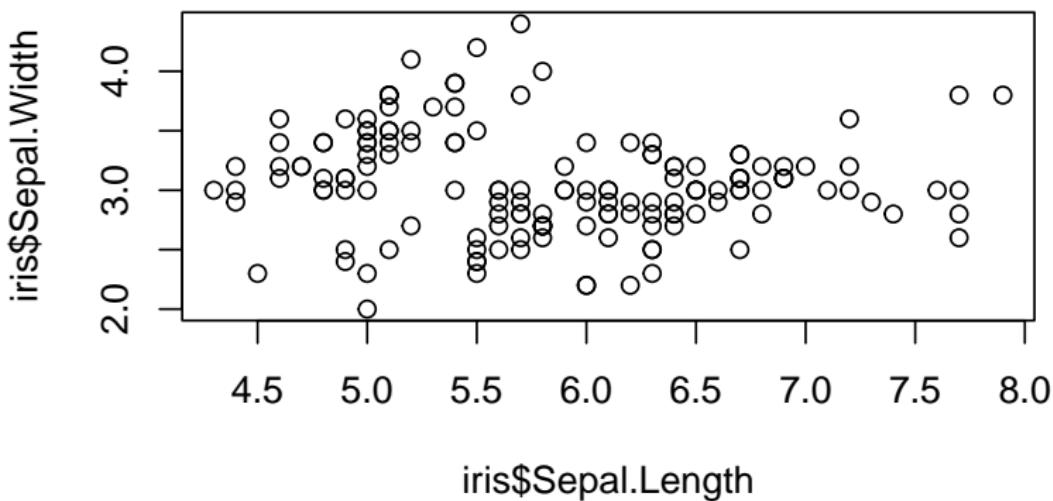
str(iris)
## 'data.frame': 150 obs. of  5 variables:
## $ Sepal.Length: num  5.1 4.9 4.7 4.6 5 ...
## $ Sepal.Width : num  3.5 3 3.2 3.1 3.6 ...
## $ Petal.Length: num  1.4 1.4 1.3 1.5 1.4 ...
## $ Petal.Width : num  0.2 0.2 0.2 0.2 0.2 ...
## $ Species: Factor w/ 3 levels "setosa","versicolor",...: 1 1 1 1 ...

?iris # Detaillierte Dokumentation
```

Streudiagramme (Scatterplots): `plot(x,y)`

```
plot(x=iris$Sepal.Length, y=iris$Sepal.Width)  
# Zeilenumbrüche mit eingerücktem Code (indentation)  
# verhindern nerviges horizontales Scrollen
```

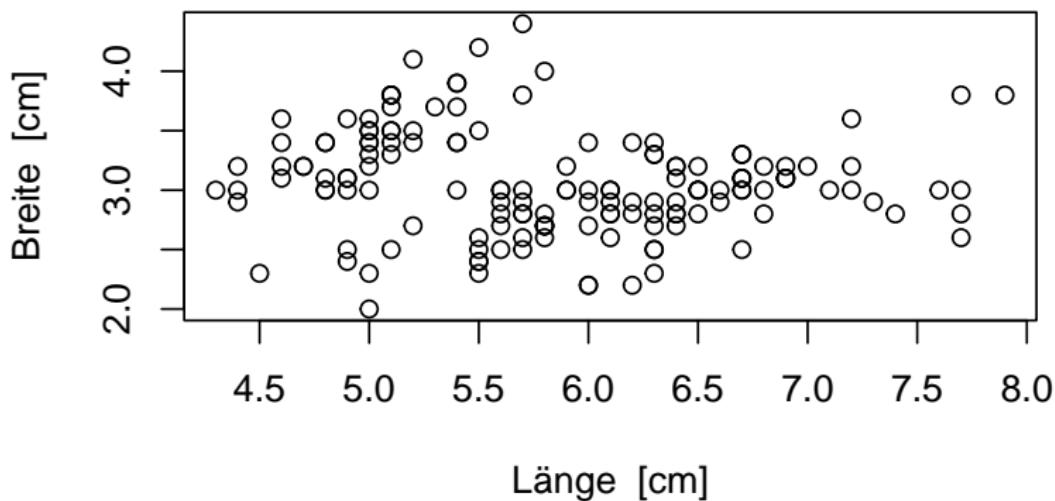
Beim Fehler "Figure margins too large",
das Rstudio Plots Panel größer ziehen.



Achsenbeschriftungen, Titel

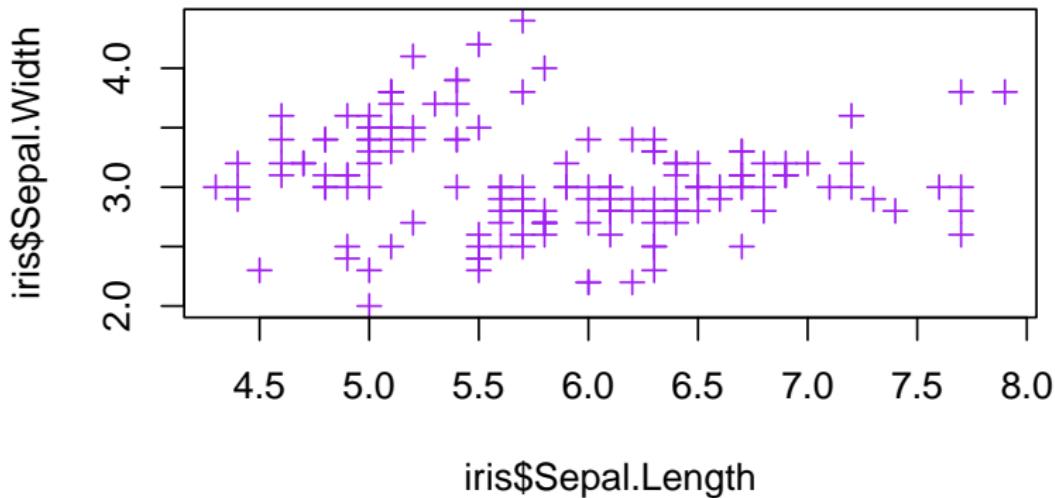
```
plot(iris$Sepal.Length, iris$Sepal.Width,  
      xlab="Länge [cm]", ylab="Breite [cm]",  
      main="Kelchblätter Schwertlilien") # Überschrift
```

Kelchblätter Schwertlilien



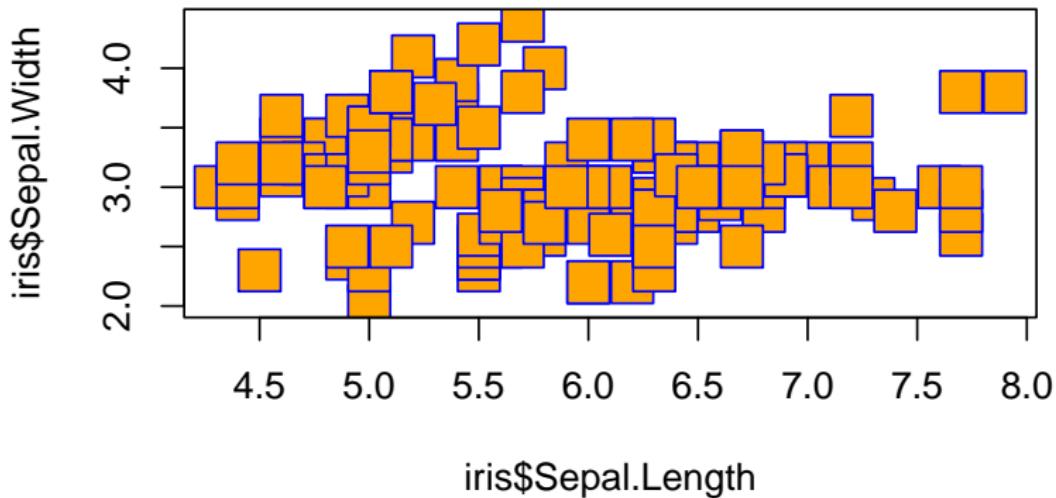
Farbe, Symbol I

```
plot(iris$Sepal.Length, iris$Sepal.Width,  
      col="purple", pch=3)  
# col: COLOR, pch: Point Character
```



Farbe, Symbol II

```
plot(iris$Sepal.Length, iris$Sepal.Width,  
      pch=22, col="blue", bg="orange", cex=2.8)  
# bg: BackGround, cex: Character Expansion
```



plot (x, y, pch = _)

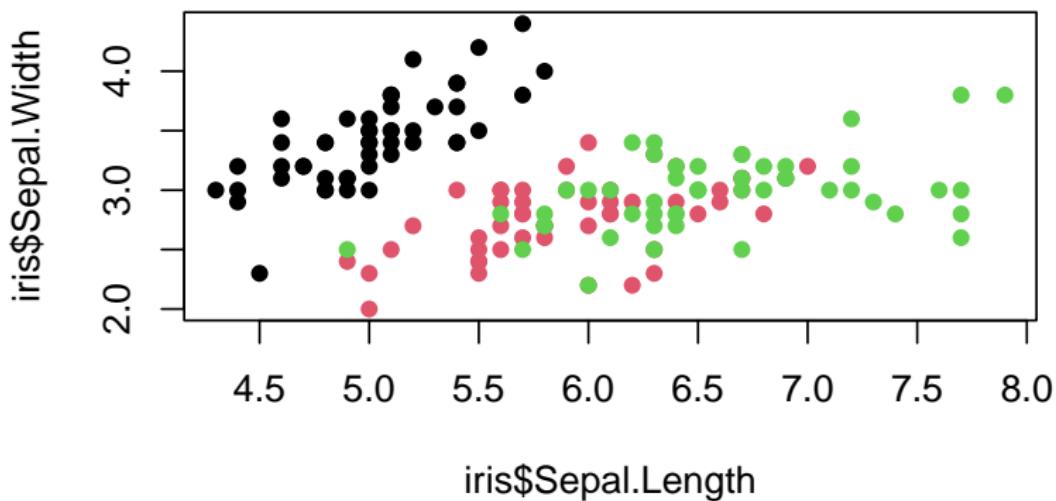
0	1	2	3	4	5	6	
□	○	△	+	×	◇	▽	
7	8	9	10	11	12	13	14
✉	*	◊	⊕	⊗	田	⊗	□
15	16	17	18	19	20		
■	●	▲	◆	●	●		
21	22	23	24	25	21:25 mit bg füllbar		
●	■	◆	▲	▼			

Farben Übersicht (mehr in bF Anhang)

1	peru	dimgrey	indianred	darksalmon	mediumpurple
2	pink	hotpink	lawngreen	darkviolet	midnightblue
3	plum	magenta	lightblue	dodgerblue	darkgoldenrod
4	snow	oldlace	lightcyan	ghostwhite	darkslateblue
5	white	skyblue	lightgray	lightcoral	darkslategray
6	azure	thistle	lightgrey	lightgreen	darkslategrey
7	beige	cornsilk	lightpink	mediumblue	darkturquoise
8	black	darkblue	limegreen	papayawhip	lavenderblush
#0399FF	brown	darkcyan	mintcream	powderblue	lightseagreen
grey	coral	darkgray	mistyrose	sandybrown	palegoldenrod
grey1	green	darkgrey	olivedrab	whitesmoke	paleturquoise
grey5	ivory	deeppink	orangered	darkmagenta	palevioletred
grey20	khaki	honeydew	palegreen	deepskyblue	blanchedalmond
grey40	linen	lavender	peachpuff	floralwhite	cornflowerblue
grey60	wheat	moccasin	rosybrown	forestgreen	darkolivegreen
grey80	bisque	navyblue	royalblue	greenyellow	lightgoldenrod
grey99	maroon	seagreen	slateblue	lightsalmon	lightslateblue
grey100	orange	seashell	slategray	lightyellow	lightslategray
red	orchid	aliceblue	slategrey	navajowhite	lightslategrey
tan	purple	burlywood	steelblue	saddlebrown	lightsteelblue
tan1	salmon	cadetblue	turquoise	springgreen	mediumseagreen
tan3	sienna	chocolate	violetred	yellowgreen	mediumslateblue
tan4	tomato	darkgreen	aquamarine	antiquewhite	mediumturquoise
blue	violet	darkkhaki	blueviolet	darkseagreen	mediumvioletred
cyan	yellow	firebrick	chartreuse	lemonchiffon	mediumaquamarine
gold	darkred	gainsboro	darkorange	lightskyblue	mediumspringgreen
navy	dimgray	goldenrod	darkorchid	mediumorchid	lightgoldenrodyellow

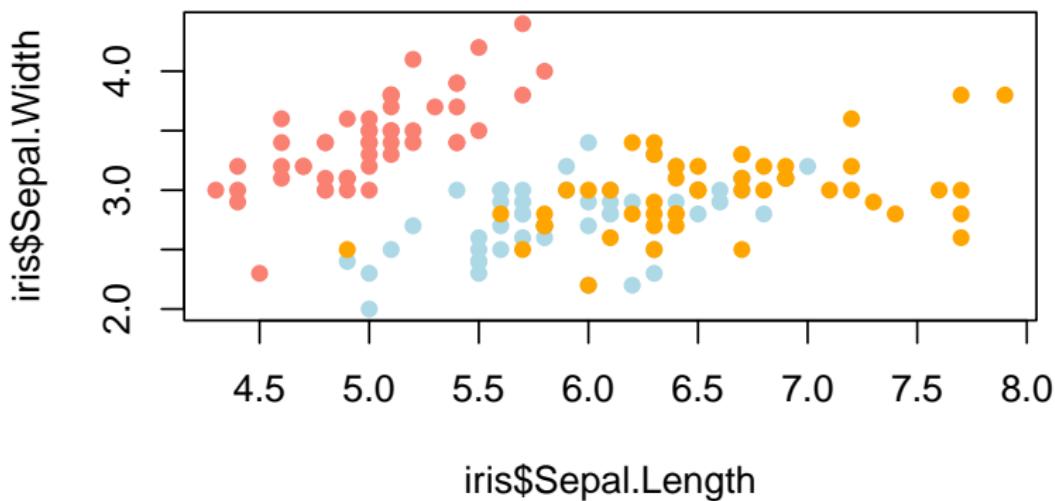
Vector mit Farben

```
plot(iris$Sepal.Length, iris$Sepal.Width, pch=16,  
      col=iris$Species) # Species ist ein factor  
# Schnellfarben 1:3 aus palette()
```



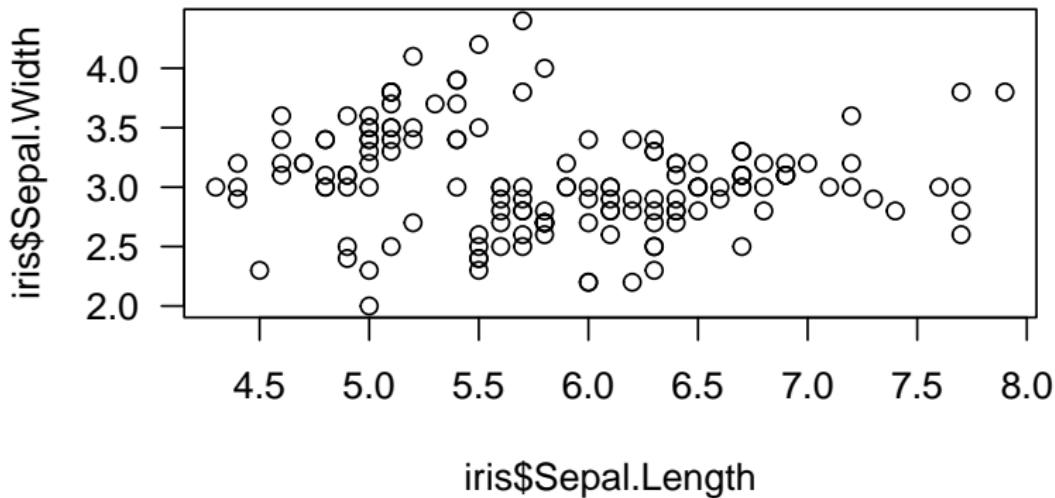
Vektor mit selbst definierten Farben

```
farben <- c("salmon", "lightblue", "orange")
plot(iris$Sepal.Length, iris$Sepal.Width, pch=16,
      col=farben[iris$Species])
```



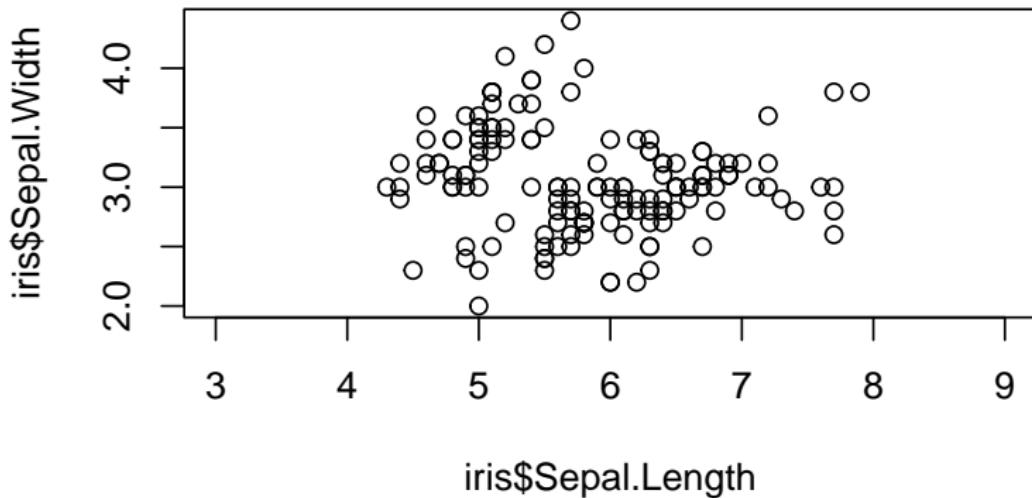
Orientierung Achsenbeschriftung

```
plot(iris$Sepal.Length, iris$Sepal.Width,  
      las=1)  
# las: LabelAxisStyle (Zahlen aufrecht)
```



Achsenbereich

```
plot(iris$Sepal.Length, iris$Sepal.Width,  
      xlim=c(3,9) )  
# xlim: X axis LIMits, analog für ylim
```



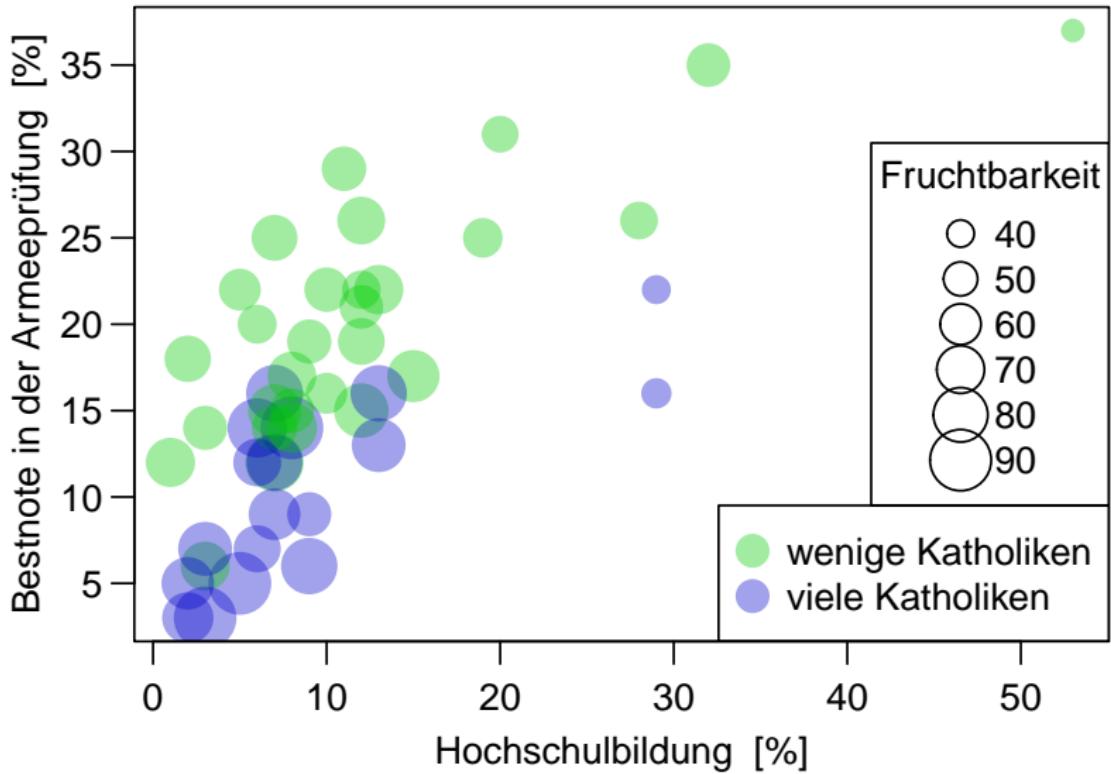
Streudiagramme - Häufige `plot` Argumente:

```
plot(  
  x, y,                                     # Punkt-koordinaten  
  xlab="Mein Label [km]", ylab="",          # Achsenbeschriftung  
  main="Grafiktitel",                      # Überschrift  
  xlim=c(20,80), ylim=1:0,                  # limits (können umgekehrt sein)  
  col="red", # Punktfarbe(n)  
  pch=0,    # point character (Symbol)  
  cex=1.8,  # character expansion (Symbolgröße)  
  las=1     # label axis style (Achsenzahlen aufrecht)  
)
```

Paul Murrell mnemonics

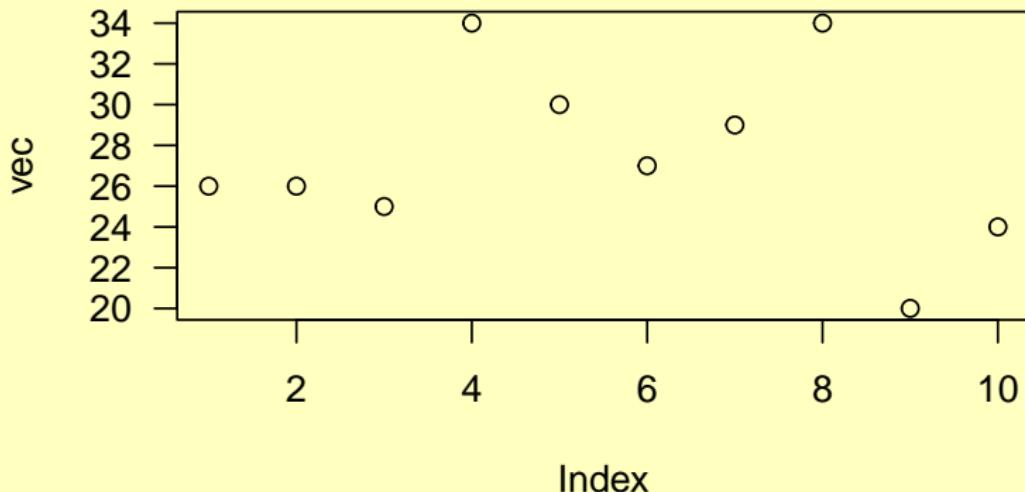
Lösung zur Übungsaufgabe

Schweiz, 1888



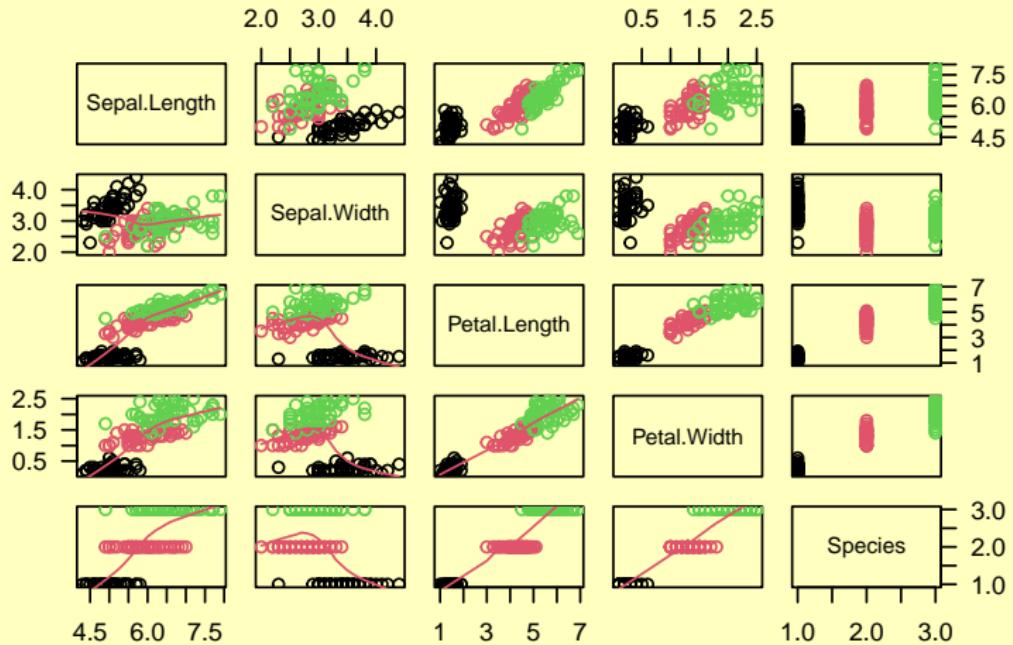
Vektor als Input

```
vec <- c(26,26,25,34,30,27,29,34,20,24)  
plot(vec, las=1) # Index 1:n auf der X-Achse
```



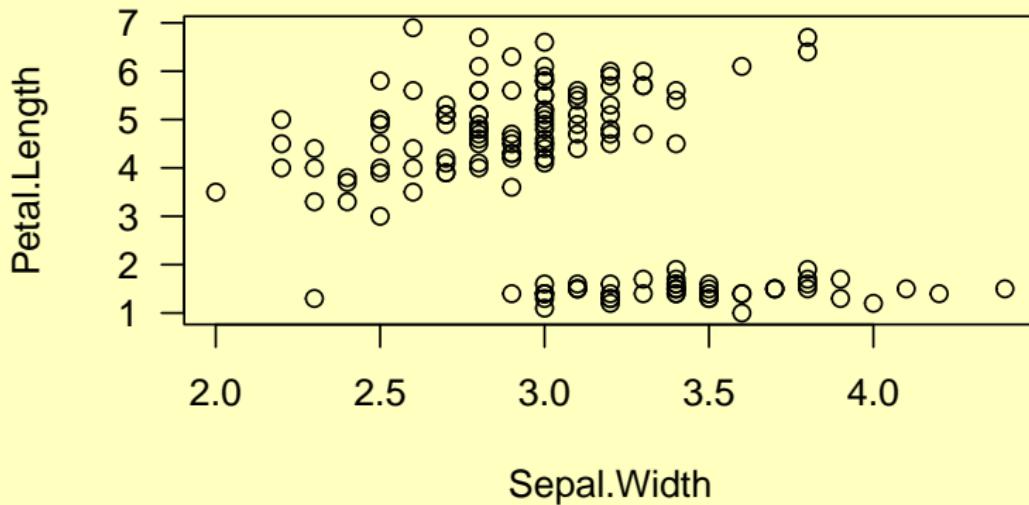
data.frame als Input -> pairs -plot

```
plot(iris, col=iris$Species, lower.panel=panel.smooth)
```



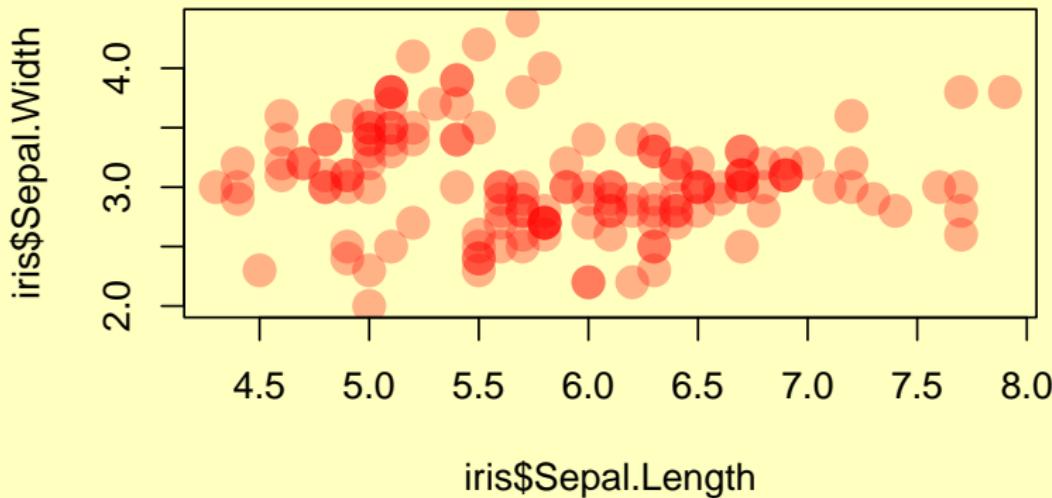
data.frame als Input

```
plot(iris[,2:3],  
     las=1)  
# Achsenbeschriftungen automatisch aus Spaltennamen
```



Transparente Farben

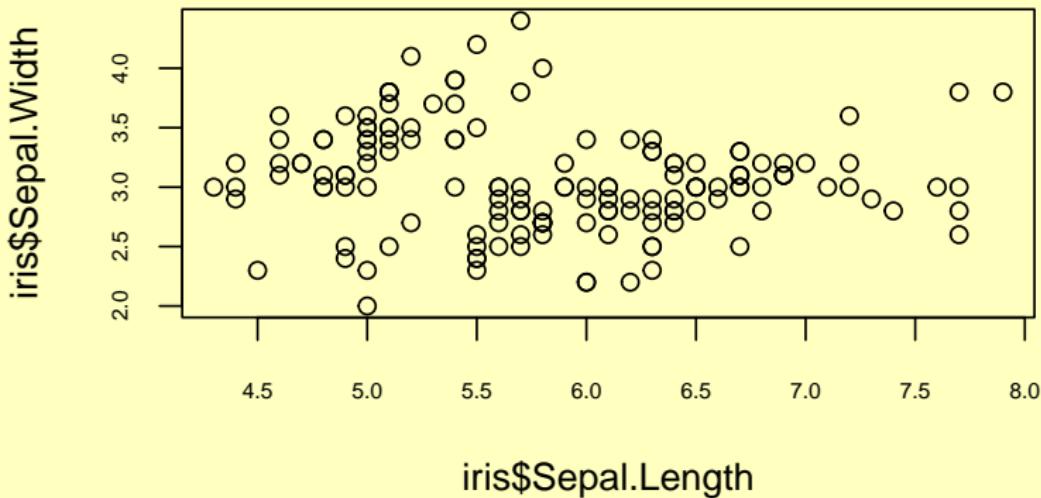
```
plot(iris$Sepal.Length, iris$Sepal.Width, cex=2,  
      col=berryFunctions::addAlpha("red"), pch=16)  
# addAlpha("red", alpha=0.3) erzeugt "#FF00004D"
```



Größe/Farbe/Schriftart verschiedener Elemente

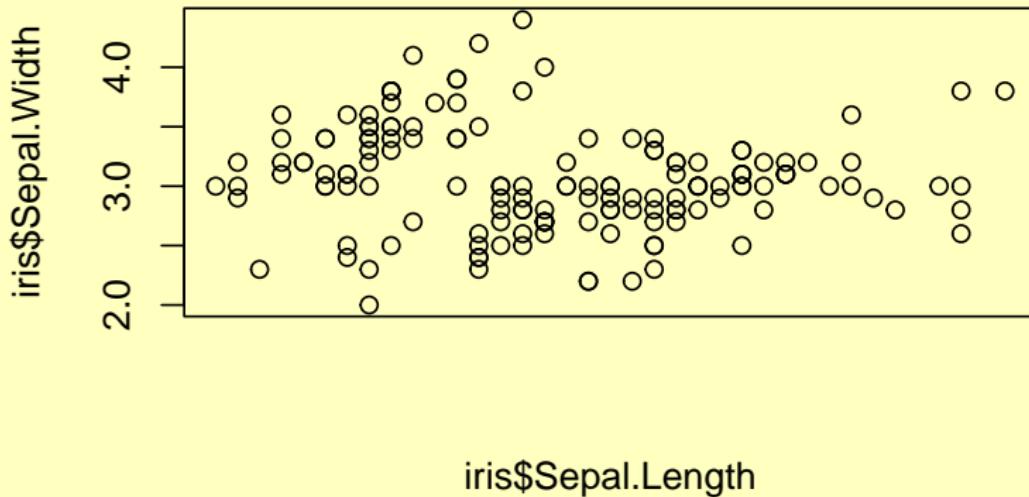
```
plot(iris$Sepal.Length, iris$Sepal.Width,  
      main="Zweizeilige\nÜberschrift", # font=3: kursiv  
      col.main="forestgreen", cex.axis=0.6, font.main=3)
```

Zweizeilige Überschrift



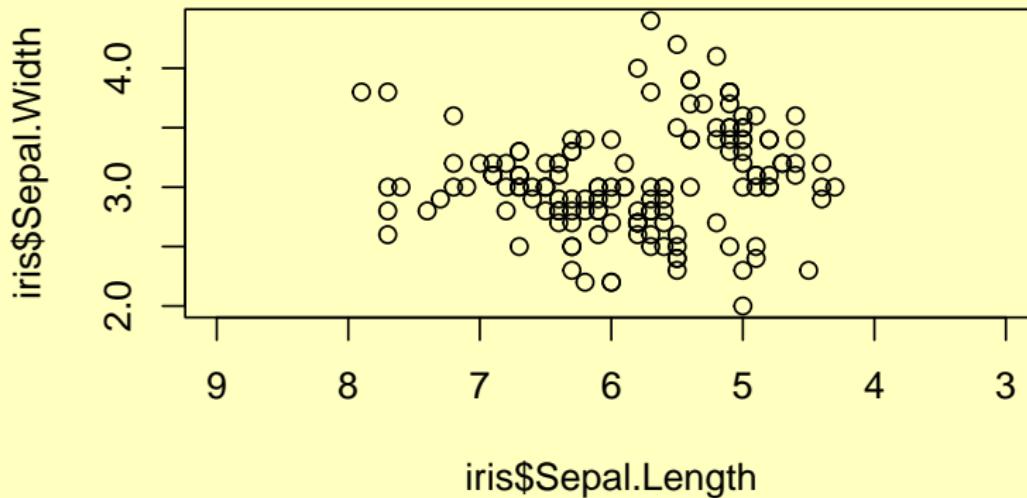
Achsenbeschriftung unterdrücken

```
plot(iris$Sepal.Length, iris$Sepal.Width,  
      xaxt="n")  
# xaxt: X Axis Type, n = none, nichts, nada, niente
```



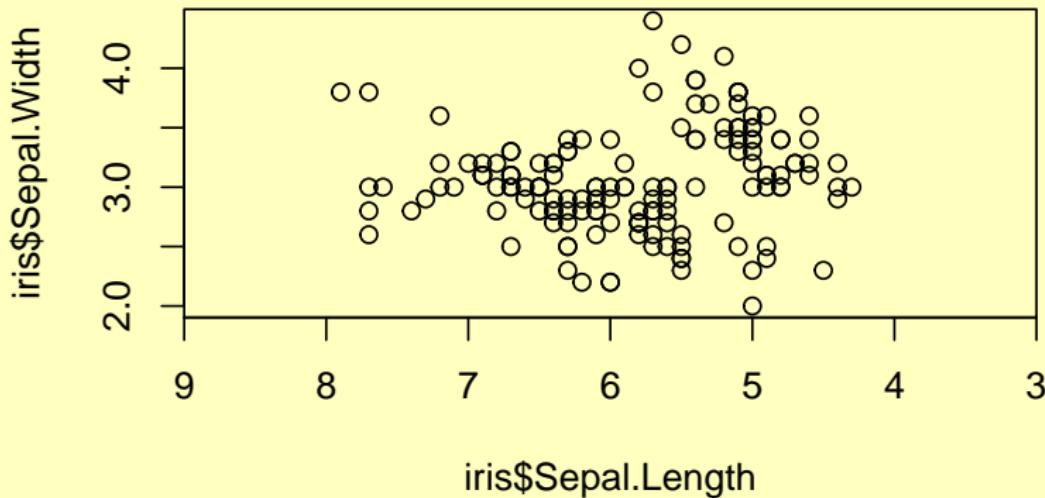
invertierte Achse

```
plot(iris$Sepal.Length, iris$Sepal.Width,  
      xlim=c(9,3) )  
# kann grafisch verwirrend sein
```



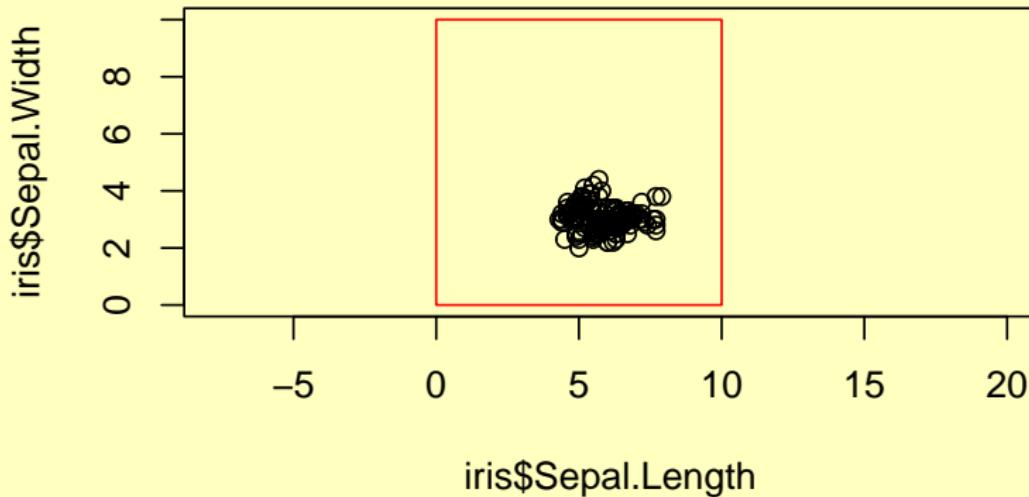
Achsenbereich genau

```
plot(iris$Sepal.Length, iris$Sepal.Width,  
      xlim=c(9,3), xaxs="i") # i=internal / r=regular  
# xaxis: X Axis Style. ("r" erweitert range je um 4%)
```



aspectratio: x zu y Verhältnis

```
plot(iris$Sepal.Length, iris$Sepal.Width,  
      ylim=c(0,10), asp=1)  
# xlim automatisch (Abstand pro Einheit wie für y)
```

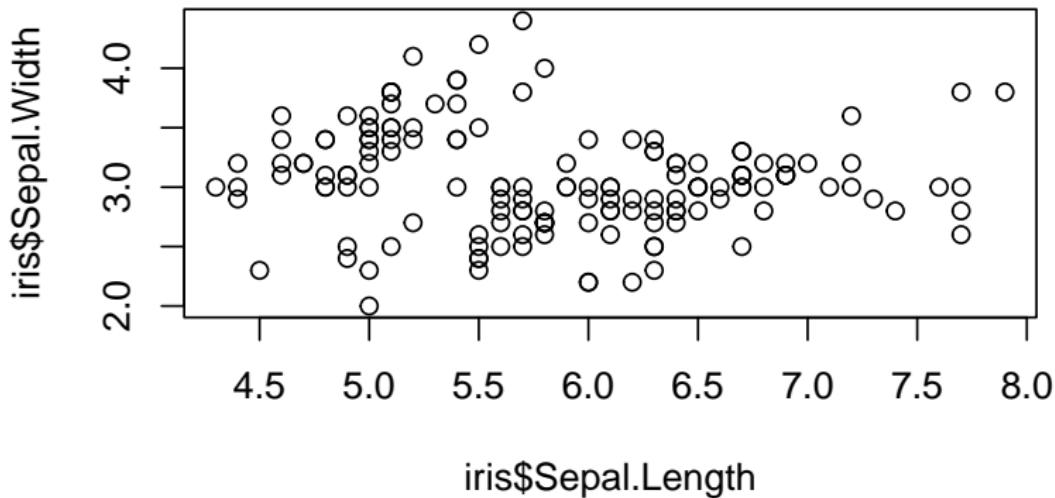


- 0. Intro
- 1. Grundlagen
- 2. Datentypen
- 3. Tabellen
- 4. Grafiken**

- 4.1 Punktdiagramme
- 4.2 Liniendiagramme
- 4.3 Balkendiagramme
- 4.4 Hinzufügen
- 4.5 Komposition
- 4.6 Verteilungsplots
- 4.7 Exportieren
- 4.8 Ausblick

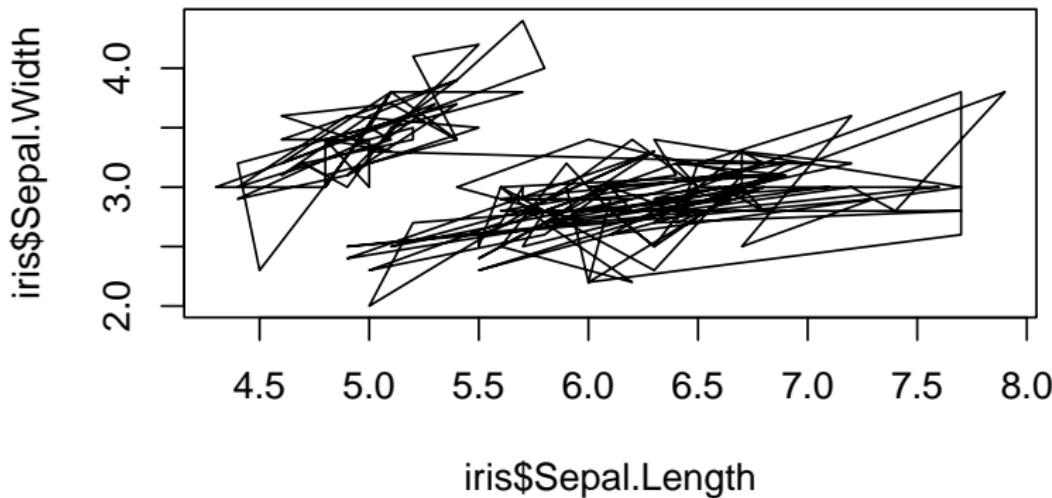
Bisher haben wir Punktdiagramme gezeichnet

```
plot(iris$Sepal.Length, iris$Sepal.Width,  
     type="p")  
# "p" (Punkte) ist der Standardwert (default)
```



Linien-diagramm, zB für Zeitreihen

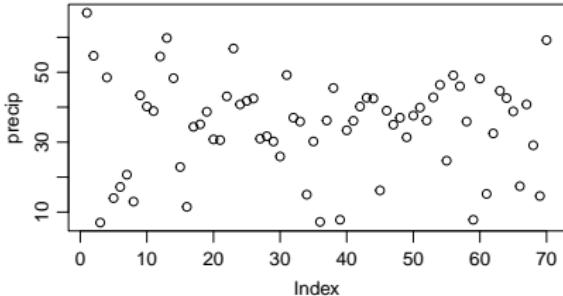
```
plot(iris$Sepal.Length, iris$Sepal.Width,  
      type="l")  
# type: l für Linien
```



Linien und Punkte

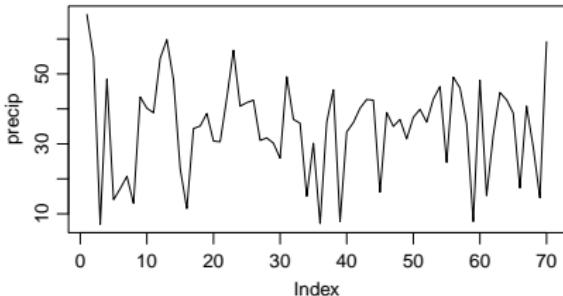
default: Points

```
plot(precip, type="p")
```



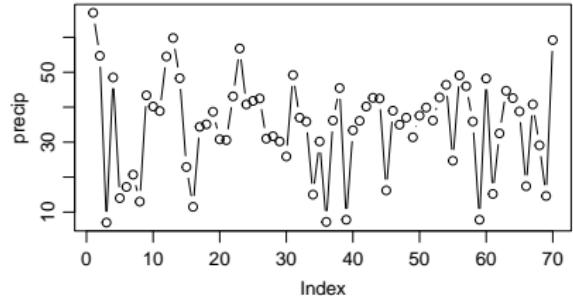
Lines

```
plot(precip, type="l")
```



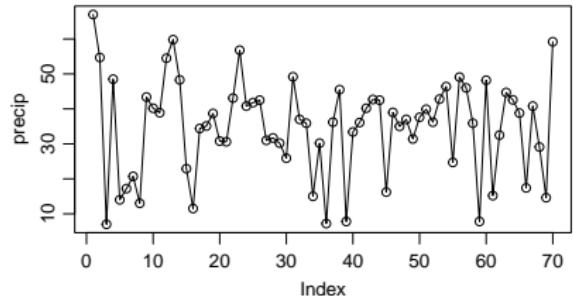
Both

```
plot(precip, type="b")
```



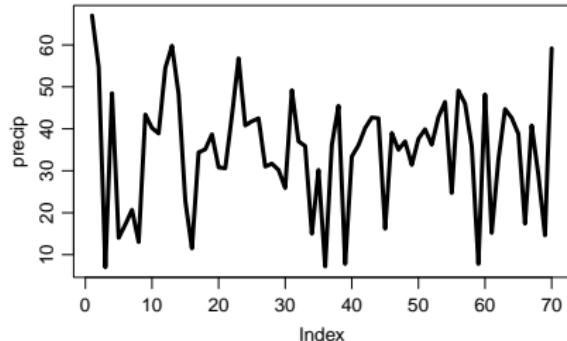
both Overplotted

```
plot(precip, type="o")
```

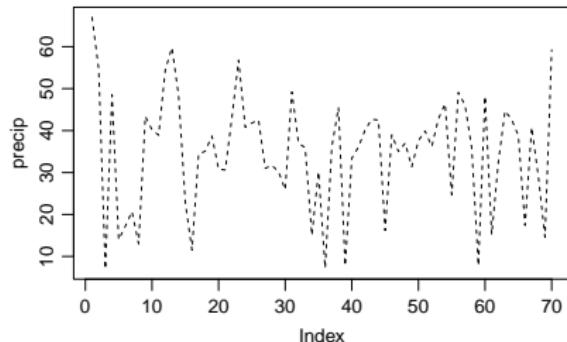


Linientypen

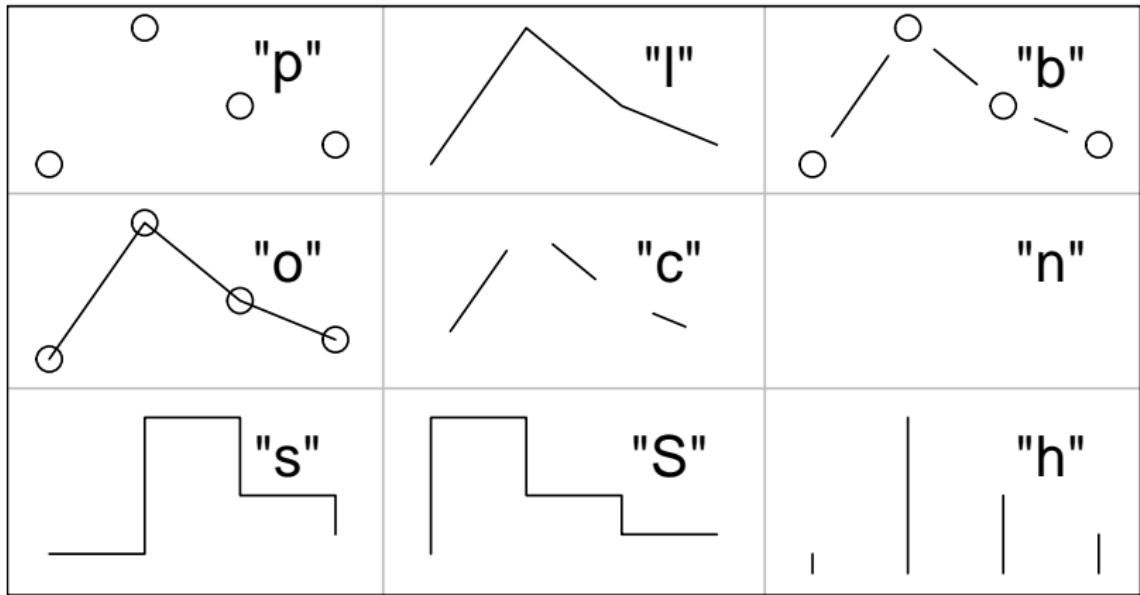
```
plot(precip, type="l", lwd=3.5) # Line Width
```



```
plot(precip, type="l", lty=2) # Line Type
```



plot (x, y, type = _)



plot (x, y, lty = __)

"blank" (no line) 0

"solid" (default) 1

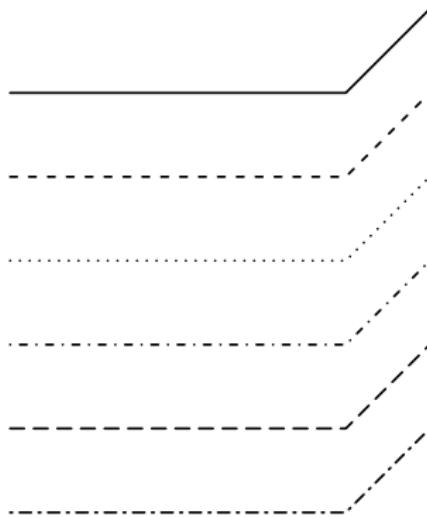
"dashed" 2

"dotted" 3

"dotdash" 4

"longdash" 5

"twodash" 6



Zusammenfassung

Liniengrafiken - Häufige `plot` Argumente:

```
plot(  
  x, y,      # Punkt-koordinaten  
  type="l",   # zeichne Linien anstatt Punkte  
  lwd=3,     # line width (Liniendicke)  
  lty=2      # line type (durchgehend/gestrichelt/gepunktet)  
)
```

- 0. Intro
- 1. Grundlagen
- 2. Datentypen
- 3. Tabellen
- 4. Grafiken**

- 4.1 Punktdiagramme
- 4.2 Liniendiagramme
- 4.3 Balkendiagramme**
- 4.4 Hinzufügen
- 4.5 Komposition
- 4.6 Verteilungsplots
- 4.7 Exportieren
- 4.8 Ausblick

Zwei eingebaute Datensätze

```
str(longley)
## 'data.frame': 16 obs. of 7 variables:
## $ GNP.deflator: num 83 88.5 88.2 89.5 96.2 98.1 99 100 ...
## $ GNP          : num 234 259 258 285 ...
## $ Unemployed   : num 236 232 368 335 ...
## $ Armed.Forces: num 159 146 162 165 ...
## $ Population   : num 108 109 110 111 ...
## $ Year         : int 1947 1948 1949 1950 1951 1952 1953 1954 ...
## $ Employed     : num 60.3 61.1 60.2 61.2 ...
```

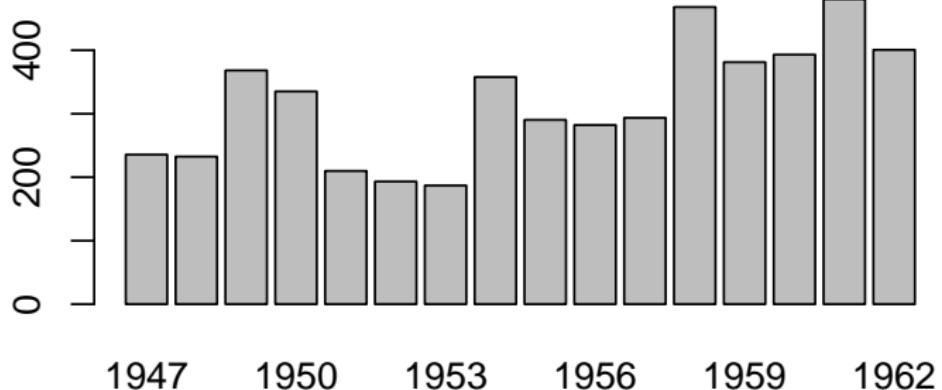
VADeaths

	Rural Male	Rural Female	Urban Male	Urban Female
## 50-54	11.7	8.7	15.4	8.4
## 55-59	18.1	11.7	24.3	13.6
## 60-64	26.9	20.3	37.0	19.3
## 65-69	41.0	30.9	54.6	35.1
## 70-74	66.0	54.3	71.1	50.0

```
data("longley") # lädt das ins globalenv()
# Rstudio str + View danach erklckbar
```

Säulendiagramme

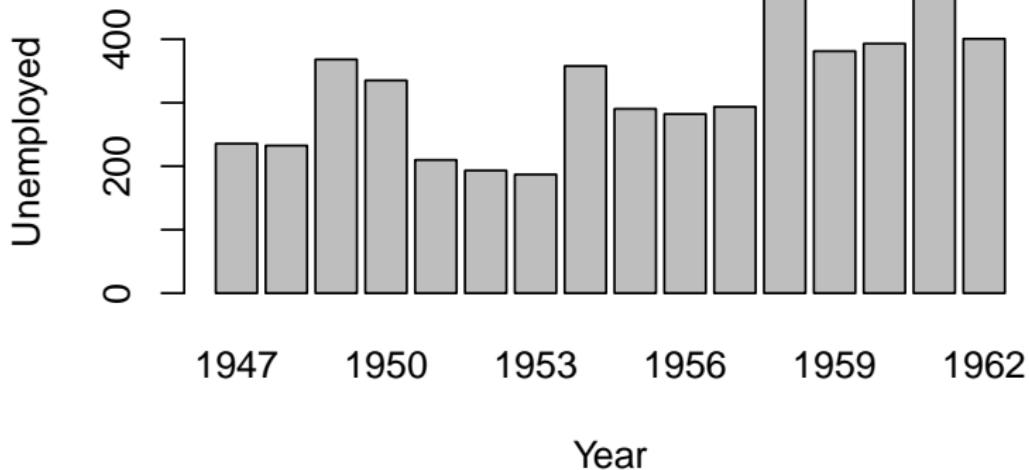
```
barplot(longley$Unemployed, names.arg=longley$Year)
```



Formula Interface: weniger tippen. \sim : lesen als 'abhängig von'

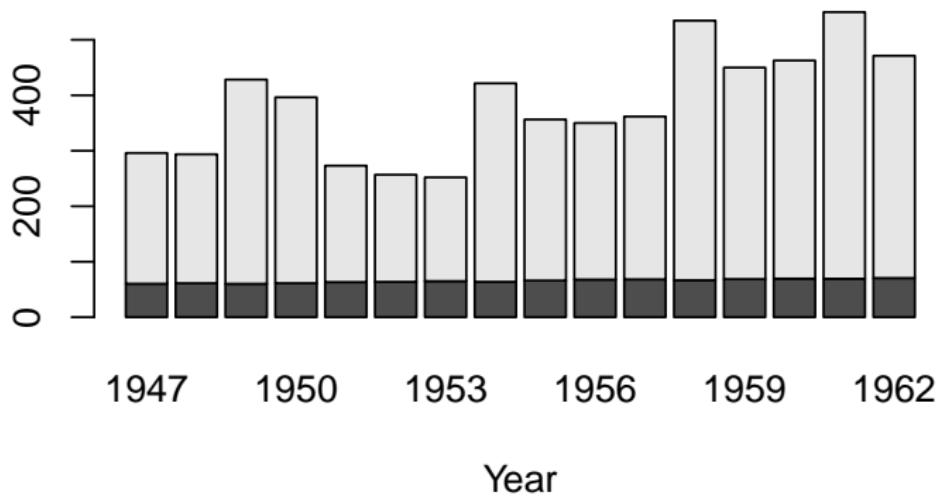
```
barplot(Unemployed ~ Year, data=longley)
```

AltGr + +, Option + N (+ Space)



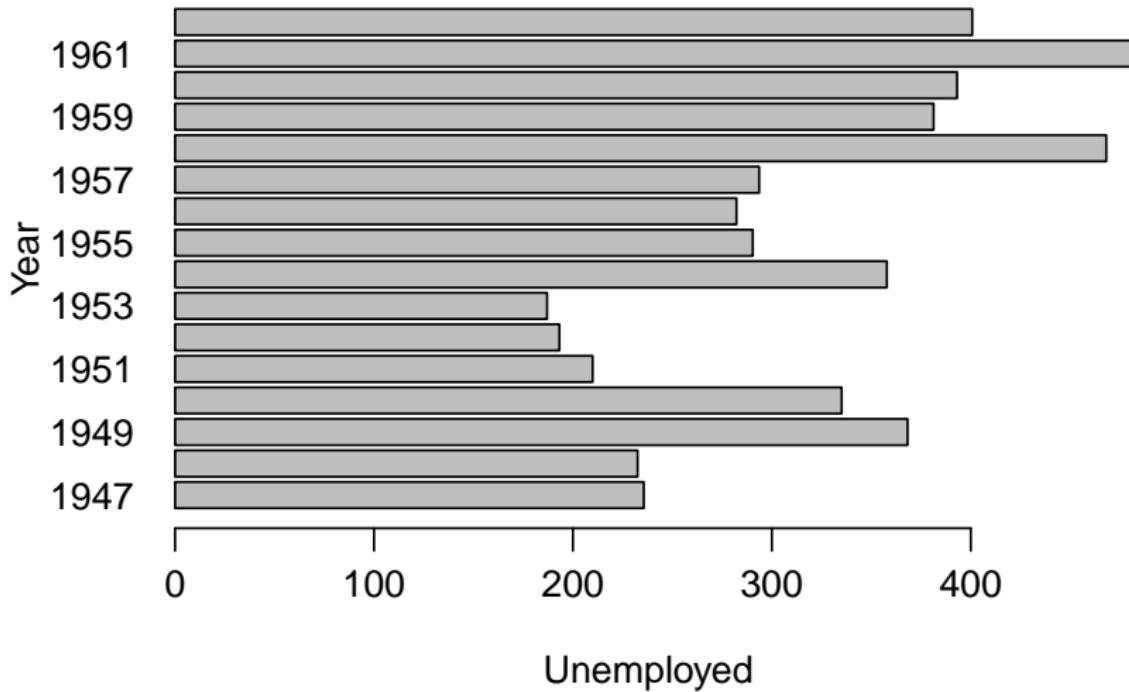
Formula Interface: flexibel erweiterbar

```
barplot(cbind(Employed, Unemployed) ~ Year, data=longley)
```



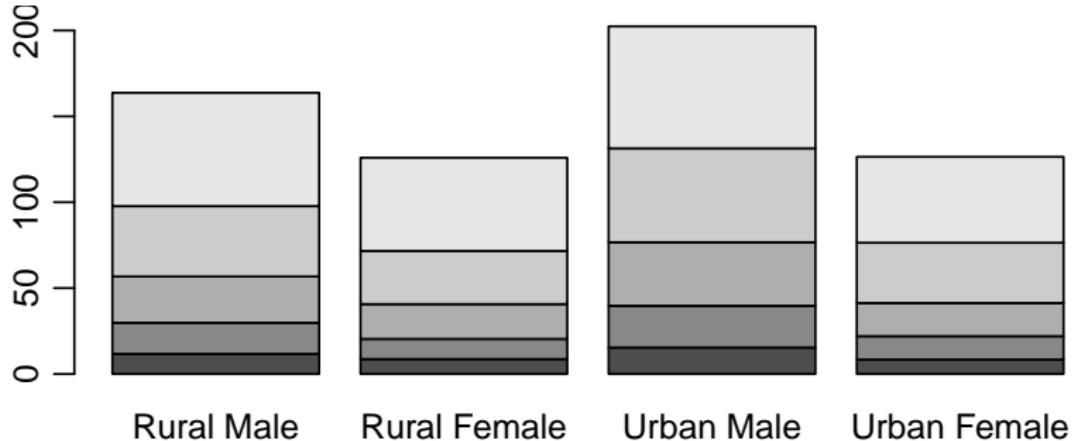
Balkendiagramm: horizontales Säulendiagramm

```
barplot(Unemployed~Year, data=longley, horiz=TRUE, las=1)
```



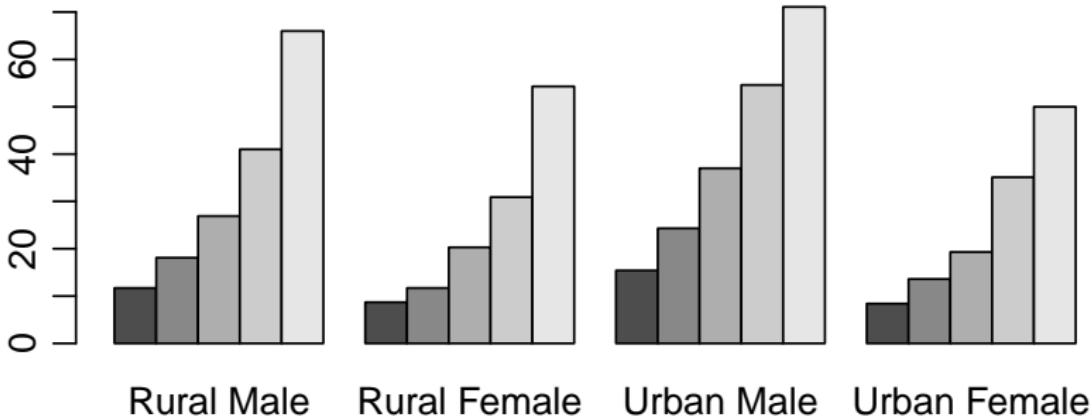
Balkendiagramm für eine Matrix

```
barplot(VADeaths)
```



Balkendiagramm: Gruppiert statt gestapelt

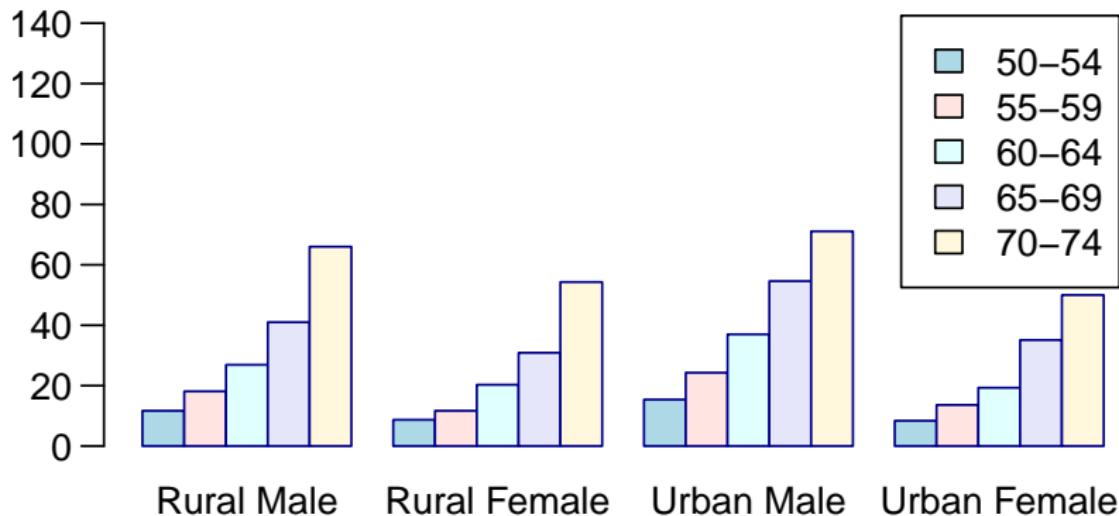
```
midpoints <- barplot(VADeaths, beside=TRUE)
```



```
midpoints
##      [,1] [,2] [,3] [,4]
## [1,]  1.5  7.5 13.5 19.5
## [2,]  2.5  8.5 14.5 20.5
## [3,]  3.5  9.5 15.5 21.5
## [4,]  4.5 10.5 16.5 22.5
## [5,]  5.5 11.5 17.5 23.5
```

Balkendiagramm: Legende

```
barplot(VADeaths, beside=TRUE, las=1,  
        col=c("lightblue","mistyrose","lightcyan",  
              "lavender","cornsilk"),  
        legend=TRUE, ylim=c(0, 150), border="darkblue")
```



Säulen- und Balkendiagramme:

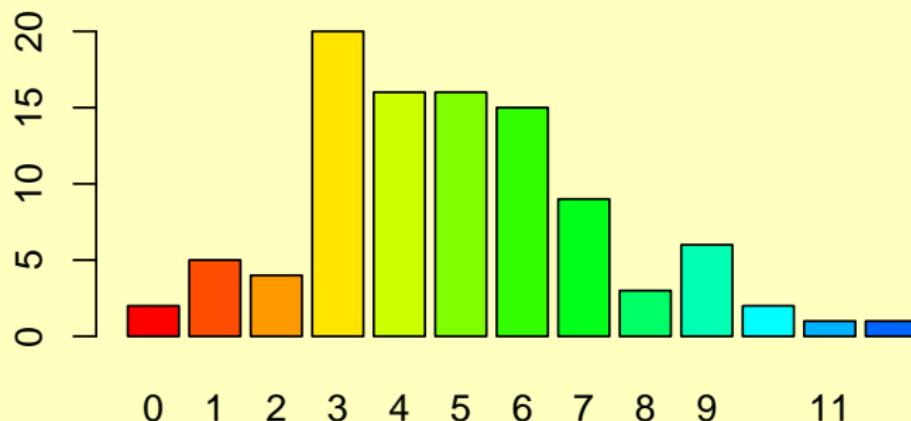
- ▶ `barplot`
- ▶ Formula interface: `y~x`

`barplot` Argumente:

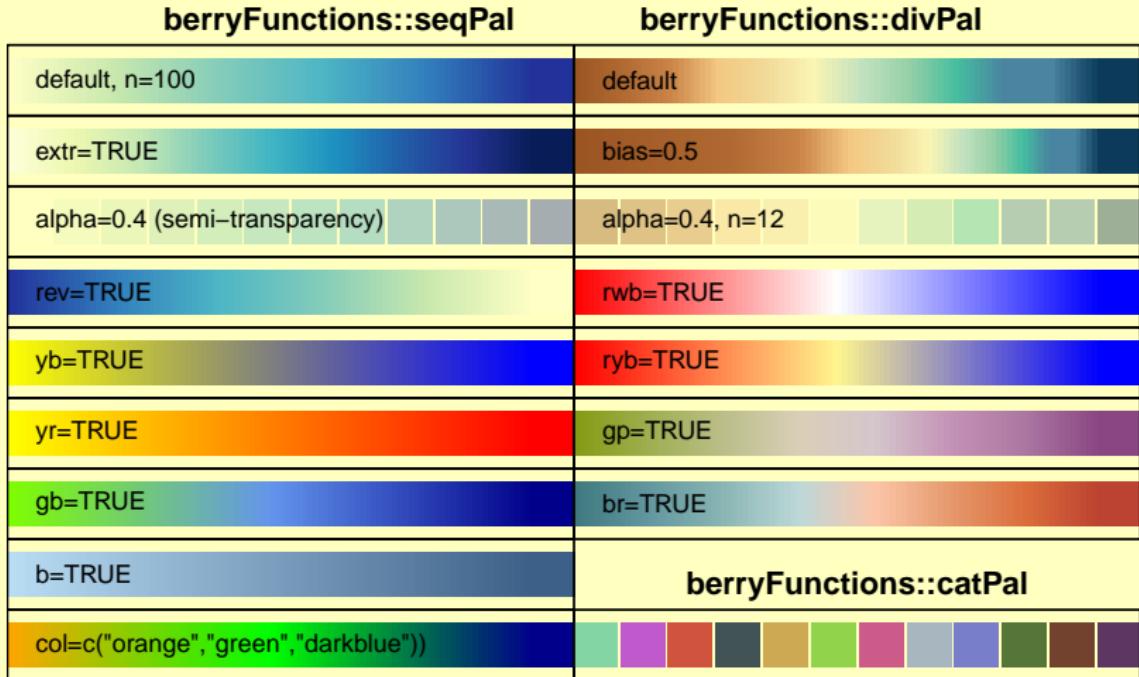
- ▶ `height`
- ▶ `horiz`
- ▶ `las, col, ylim`
- ▶ `beside`
- ▶ `legend`

Eingebaute Farbpaletten

```
zahlen <- table(stats::rpois(100, lambda=5))  
bp <- barplot(zahlen, col=rainbow(20))
```

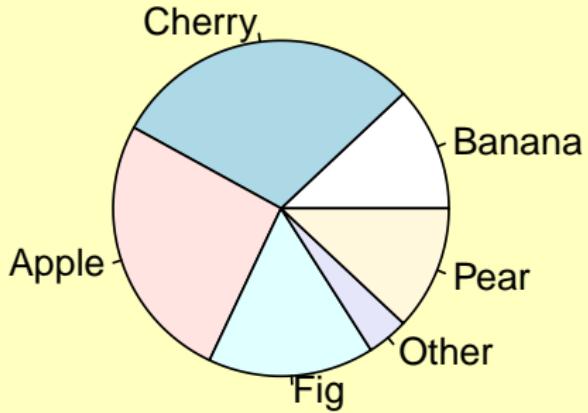


bessere Farbpaletten

`berryFunctions::showPal()`

Tortendiagramme nicht verwenden

```
vec <- c(12, 30, 26, 16, 4, 12)
names(vec) <- c("Banana", "Cherry", "Apple", "Fig", "Other", "Pear")
pie(vec)
```



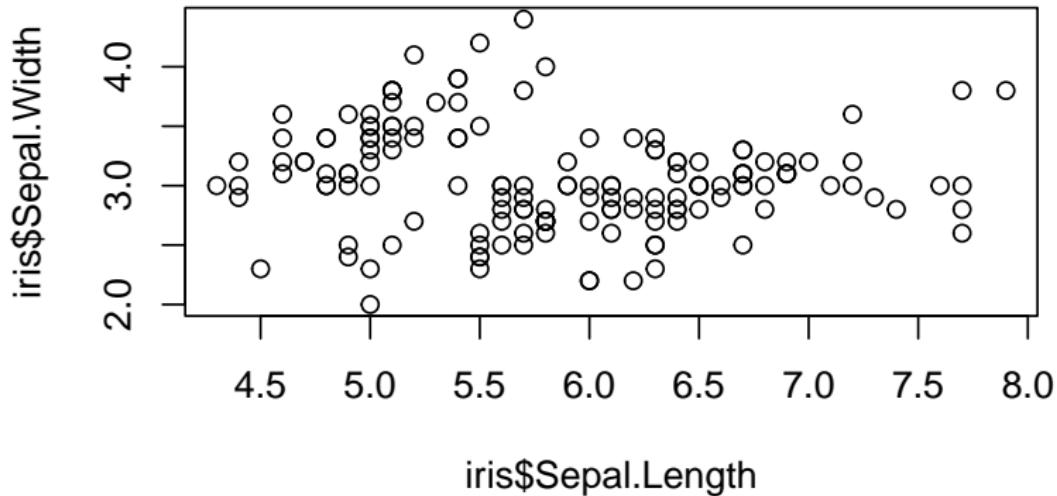
Wenn Diagrammbetrachter Proportionen vergleichen können sollen, wähle lieber Balkendiagramme, siehe z.B. [death to pie charts](#) oder [save the pies for dessert](#).

- 0. Intro
- 1. Grundlagen
- 2. Datentypen
- 3. Tabellen
- 4. Grafiken**

- 4.1 Punktdiagramme
- 4.2 Liniendiagramme
- 4.3 Balkendiagramme
- 4.4 Hinzufügen**
- 4.5 Komposition
- 4.6 Verteilungsplots
- 4.7 Exportieren
- 4.8 Ausblick

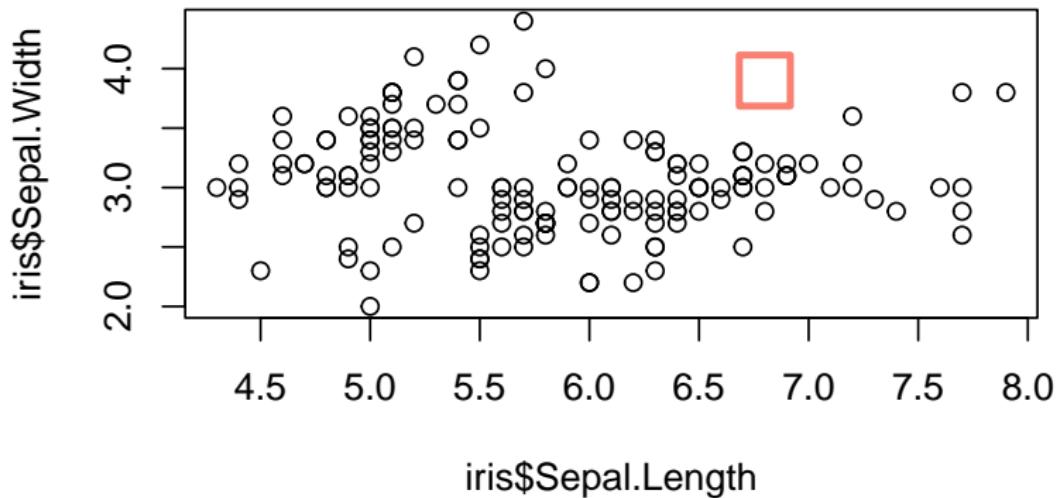
High-level-Befehle wie `plot` erzeugen eine komplette, neue Grafik

```
plot(x=iris$Sepal.Length, y=iris$Sepal.Width)  
# Datensatz siehe Abschnitt 5.1 Punktdiagramme
```



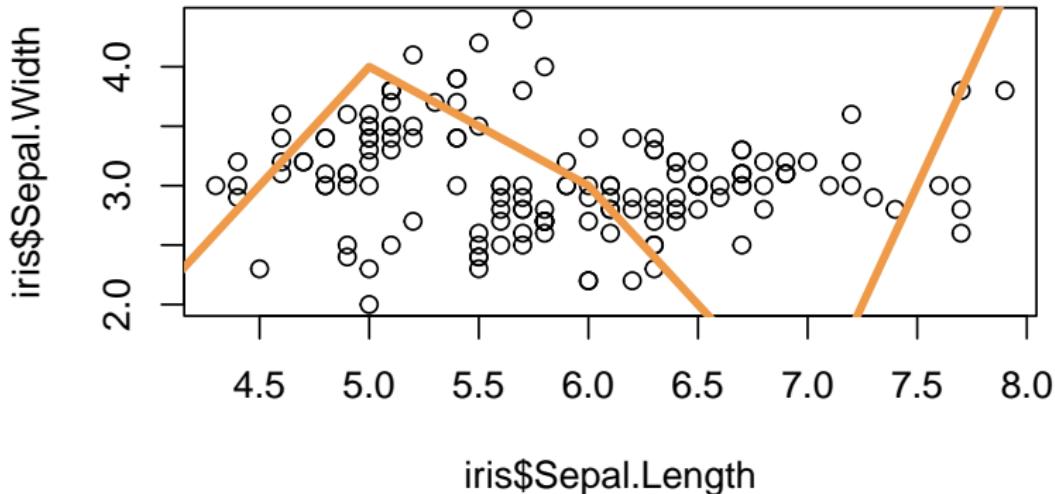
Low-level-Befehle fügen einer bestehenden Grafik etwas hinzu

```
plot(x=iris$Sepal.Length, y=iris$Sepal.Width)
points(x=6.8, y=3.9, pch=0, cex=3, col="salmon", lwd=3)
```



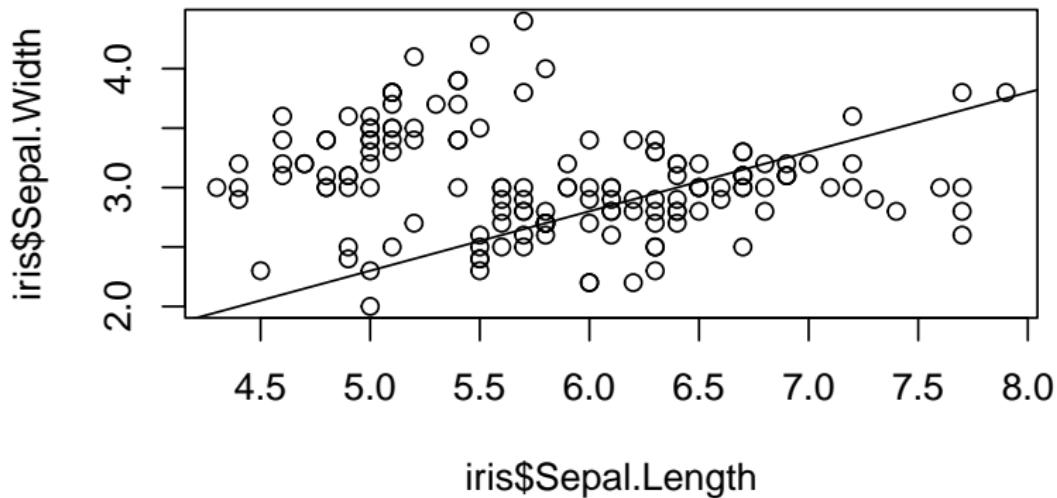
points und lines akzeptieren viele Argumente von plot.default

```
plot(x=iris$Sepal.Length, y=iris$Sepal.Width)
lines(x=4:8, y=c(2,4,3,1,5), lwd=3.5, col="tan2")
```



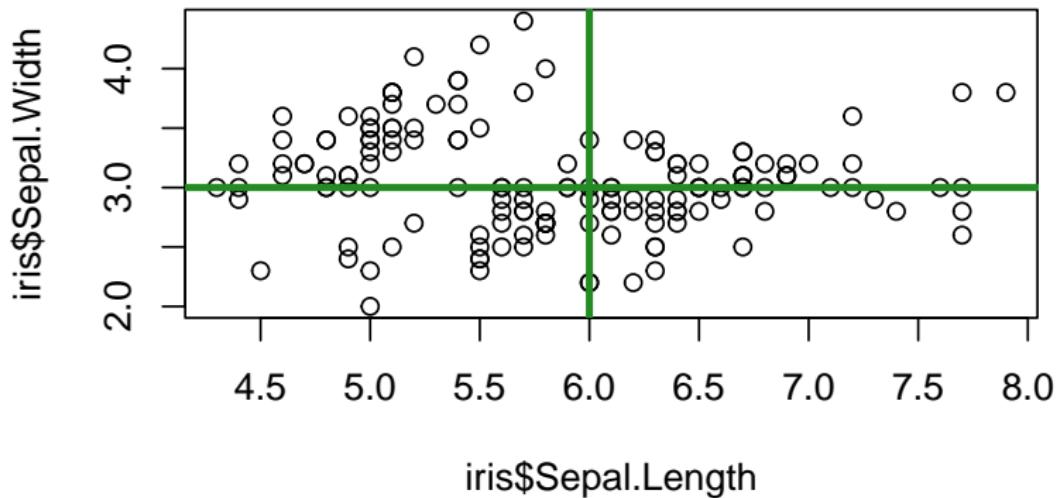
Geraden

```
plot(x=iris$Sepal.Length, y=iris$Sepal.Width)
abline(a=-0.2, b=0.5) # y-Achsen-Schnittpunkt, Steigung
```



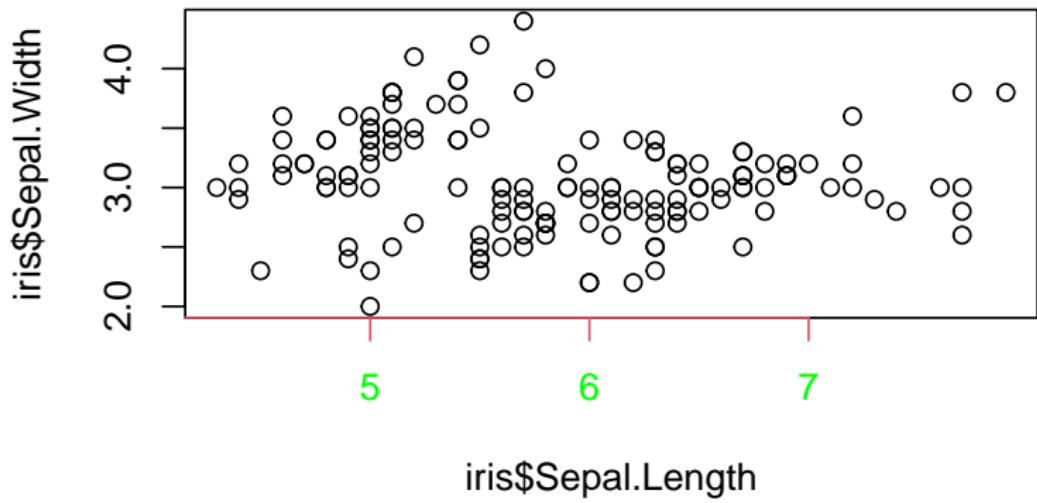
vertikale / horizontale Linien

```
plot(x=iris$Sepal.Length, y=iris$Sepal.Width)
abline(h=3, v=6, lwd=3, col="forestgreen")
```



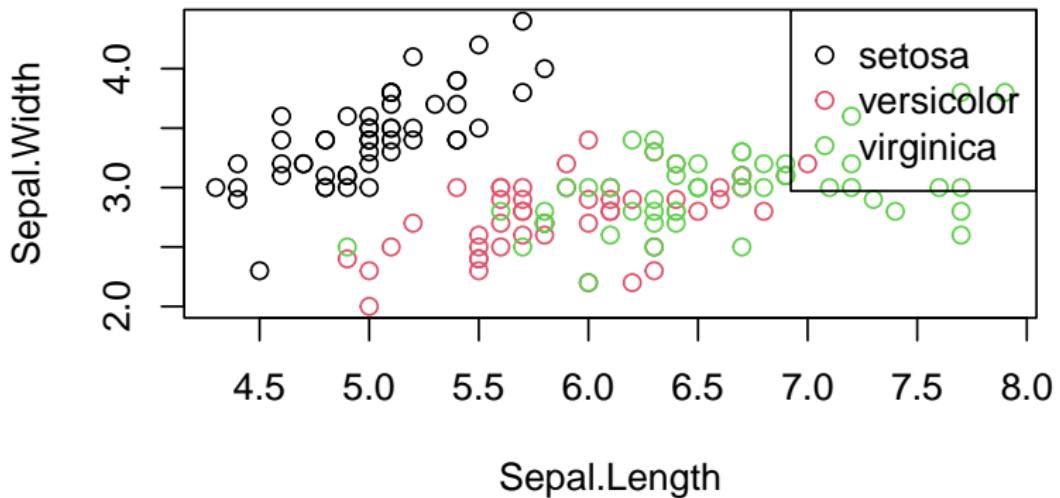
Achsen

```
plot(x=iris$Sepal.Length, y=iris$Sepal.Width, xaxt="n")
axis(side=1, at=4:7, col="#DF536B", col.axis="green")
```



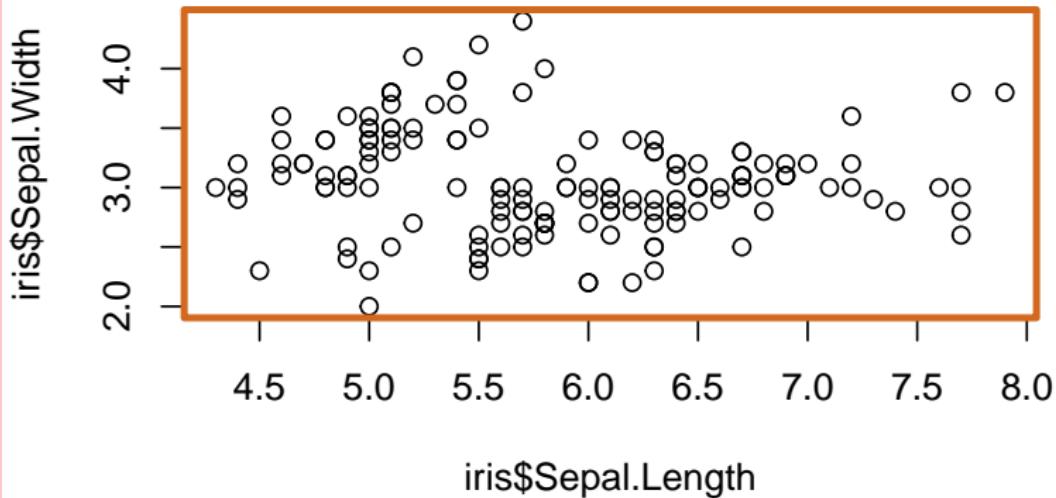
Legende

```
plot(Sepal.Width~Sepal.Length, data=iris, col=Species)
legend("topright", levels(iris$Species), col=1:3, pch=1)
```



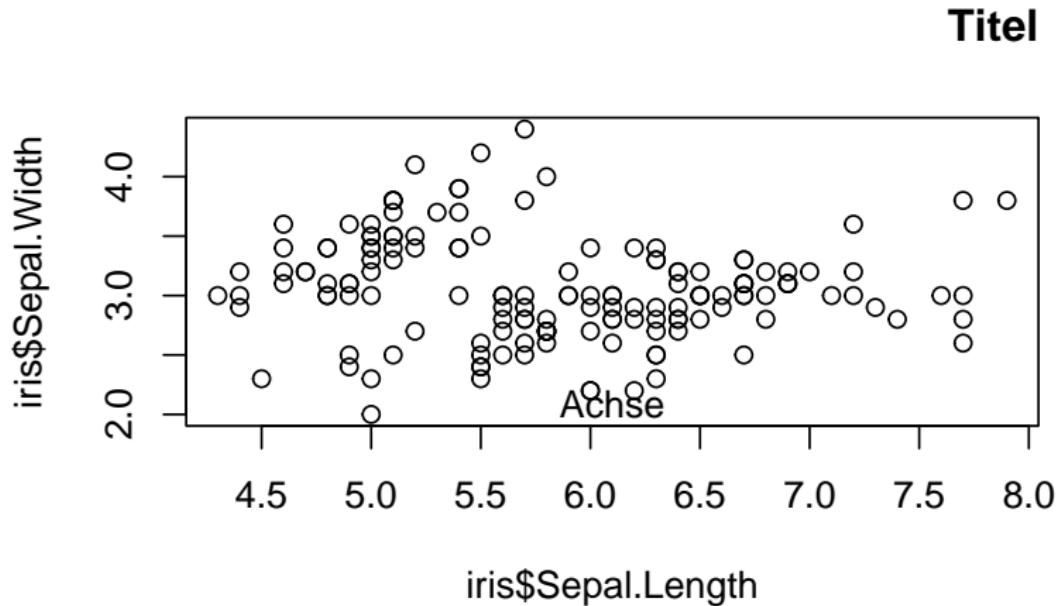
Umrandung

```
plot(x=iris$Sepal.Length, y=iris$Sepal.Width)
box(col="chocolate", lwd=3) ; box("outer", col="pink")
```



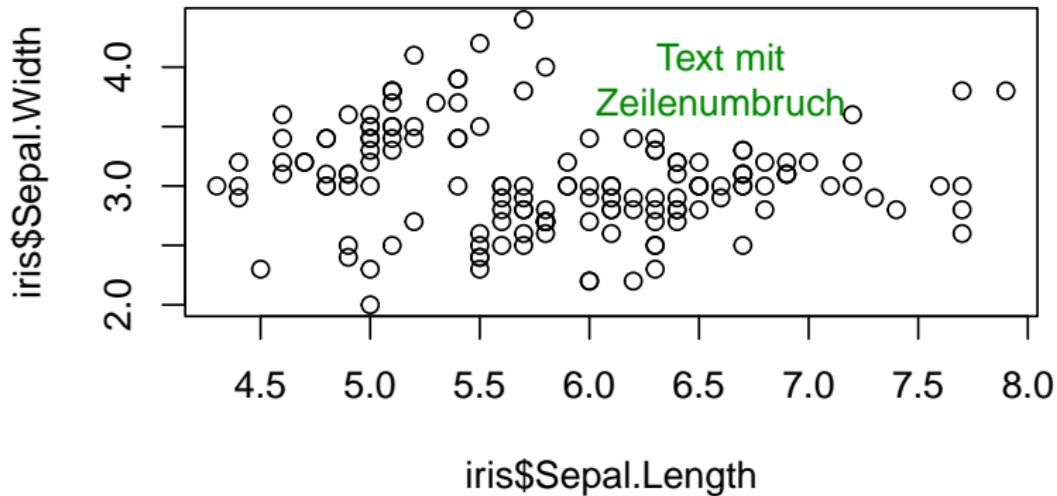
Titel-elemente nachträglich hinzufügen

```
plot(x=iris$Sepal.Length, y=iris$Sepal.Width)
title(main="Titel", adj=1) ; title(xlab="Achse", line=-1)
```



Text in der Grafik

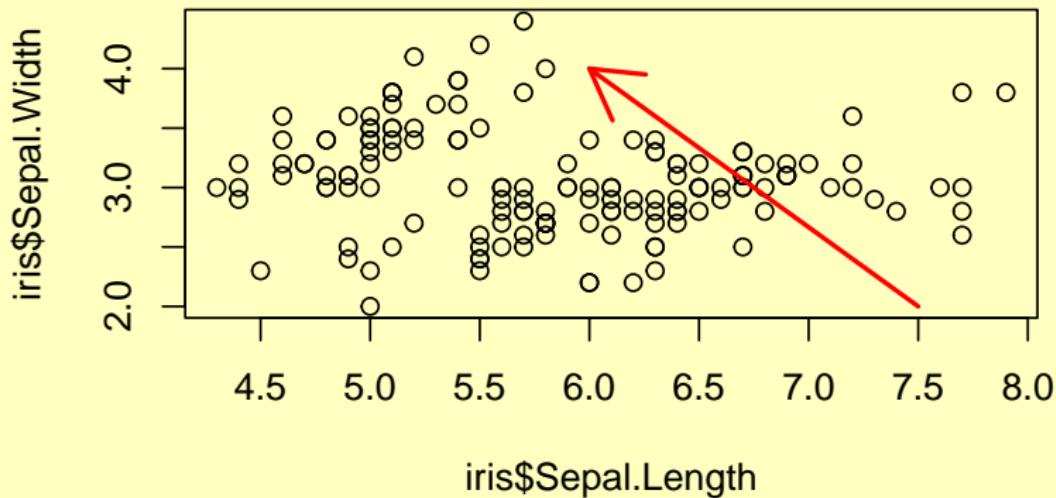
```
plot(x=iris$Sepal.Length, y=iris$Sepal.Width)
text(6.6, 3.9, "Text mit\nZeilenumbruch", col="green4")
```



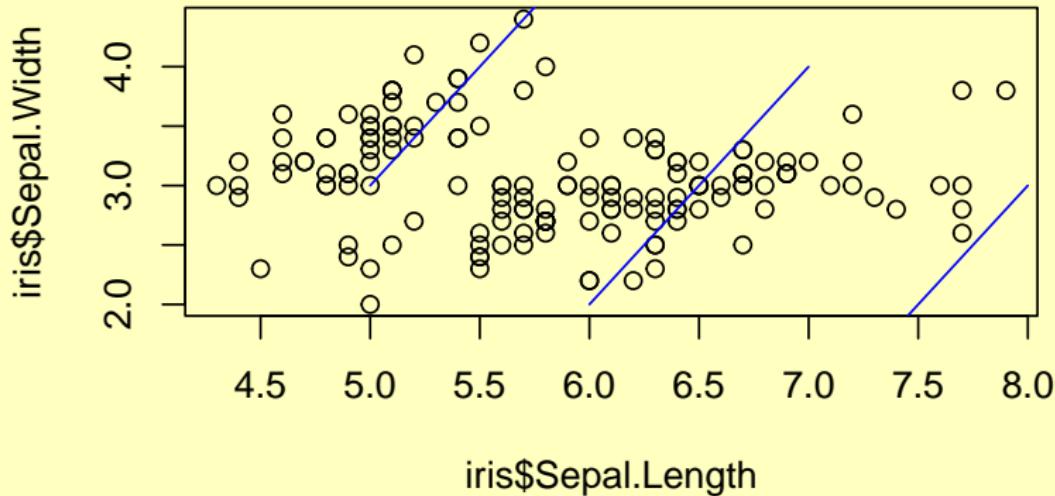
flexible Diagramme mit Low-level-Befehlen:

- ▶ `points`, `lines`, `abline`
- ▶ `axis`, `legend`, `box`
- ▶ `title`, `text`

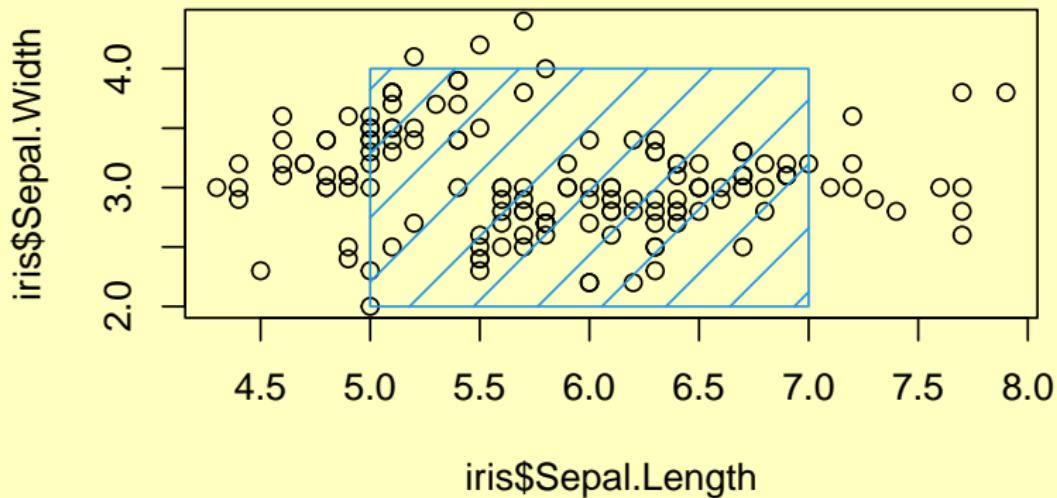
```
plot(x=iris$Sepal.Length, y=iris$Sepal.Width)
arrows(x0=7.5,y0=2, x1=6,y1=4, col="red", lwd=2)
```



```
plot(x=iris$Sepal.Length, y=iris$Sepal.Width)
segments(x0=5:7, y0=3:1, x1=6:8, y1=5:3, col="blue")
```

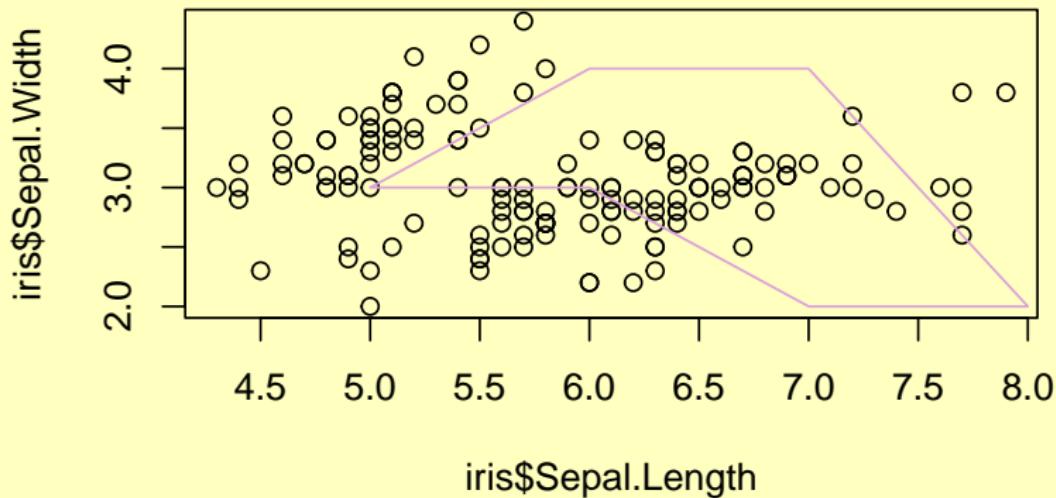


```
plot(x=iris$Sepal.Length, y=iris$Sepal.Width)
rect(xleft=5,ybottom=2, xright=7,ytop=4, col=4, density=5)
```



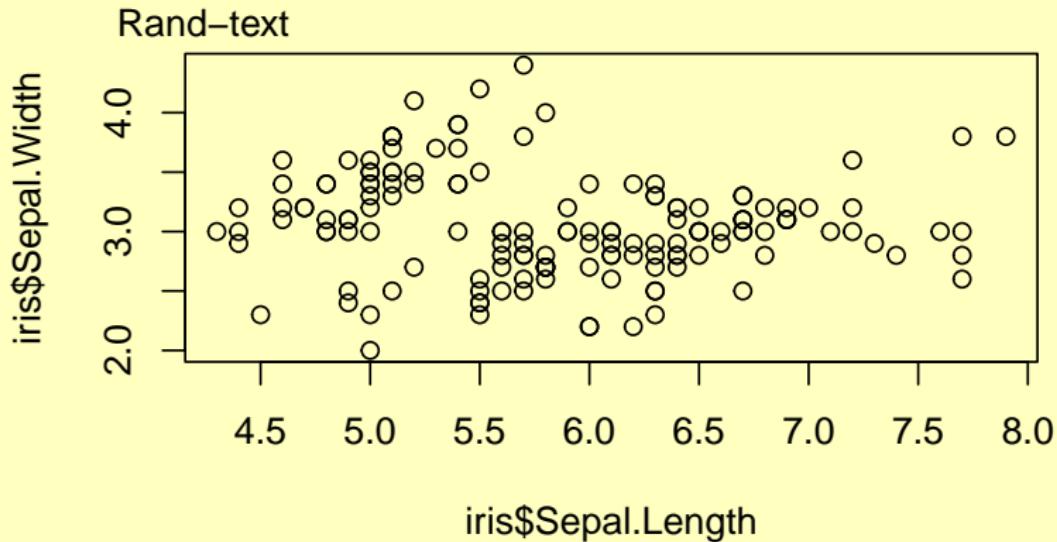
Polygone

```
plot(x=iris$Sepal.Length, y=iris$Sepal.Width)
polygon(x=c(5,6,7,8,7,6), y=c(3,4,4,2,2,3), border="plum")
```



Text im Rand

```
plot(x=iris$Sepal.Length, y=iris$Sepal.Width)
mtext(side=3, text="Rand-text", line=0.2, adj=-0.1)
```

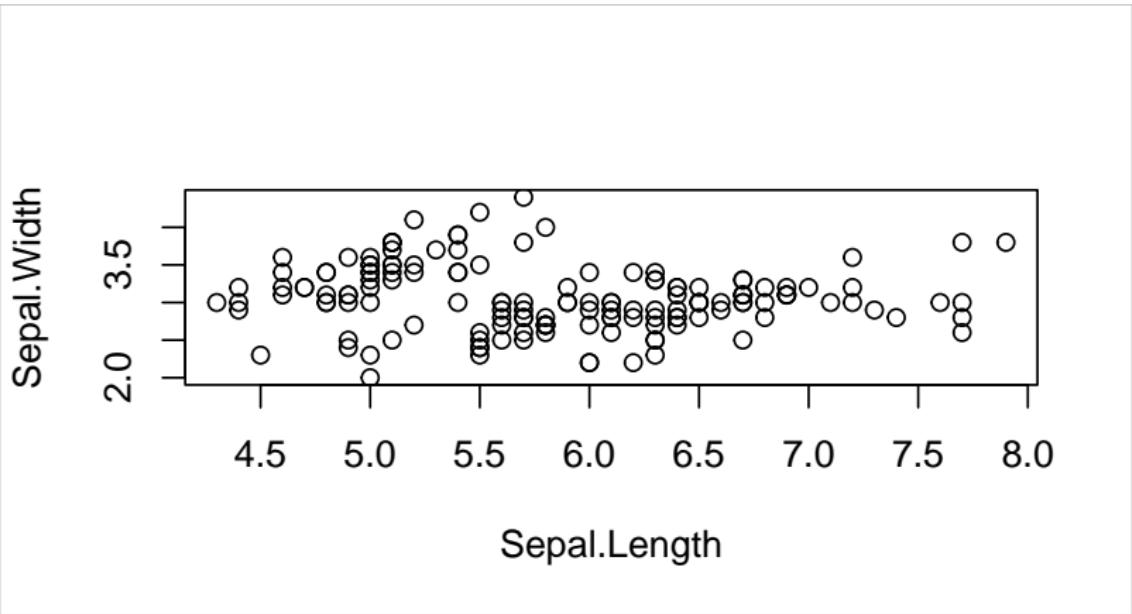


- 0. Intro
- 1. Grundlagen
- 2. Datentypen
- 3. Tabellen
- 4. Grafiken

- 4.1 Punktdiagramme
- 4.2 Liniendiagramme
- 4.3 Balkendiagramme
- 4.4 Hinzufügen
- 4.5 Komposition
- 4.6 Verteilungsplots
- 4.7 Exportieren
- 4.8 Ausblick

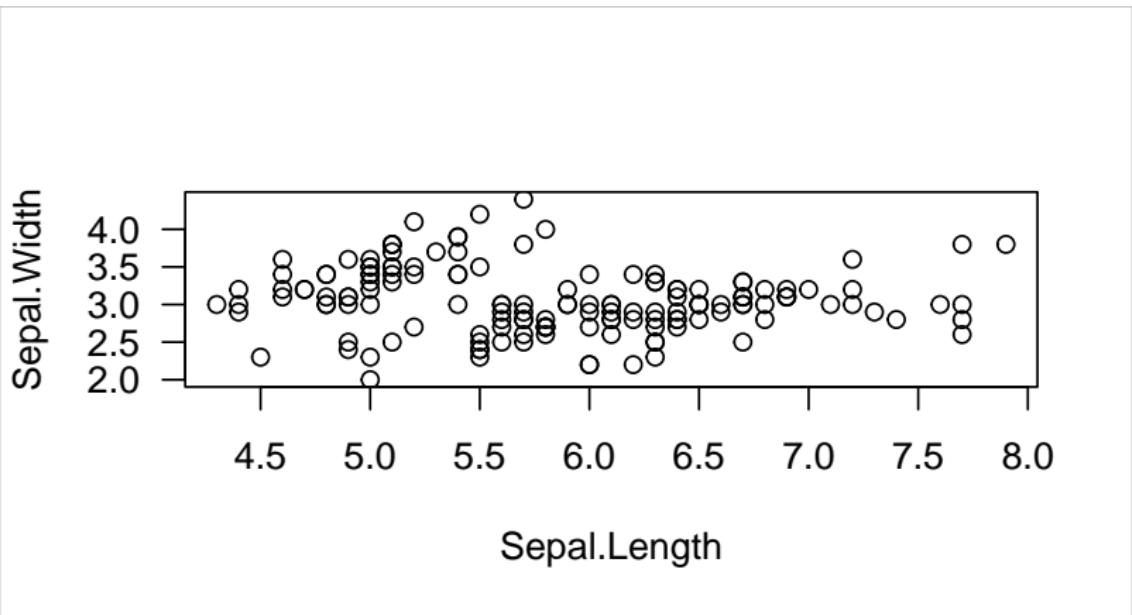
par : Default Parameter einer Grafik

```
# default (standard) Parameter:  
plot(Sepal.Width~Sepal.Length, data=iris)
```



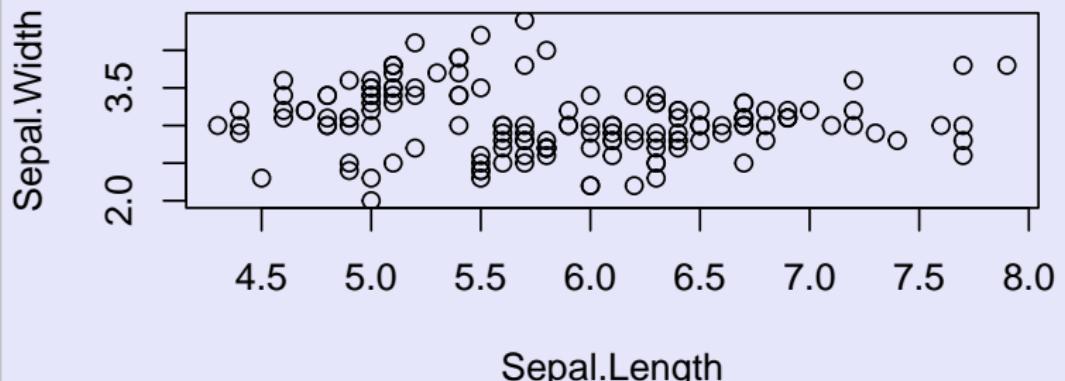
par : Parameter der folgenden Grafiken

```
par(las=1) # ab jetzt für alle Plots im gleichen Device:  
plot(Sepal.Width~Sepal.Length, data=iris)
```



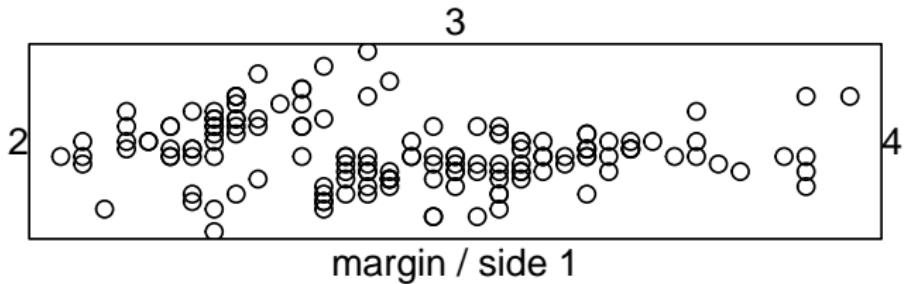
par : bg - background nicht für pch=21:25 sondern für Device

```
par(bg="lavender") # Device = zB pdf, RStudio Grafiken  
plot(Sepal.Width~Sepal.Length, data=iris)
```



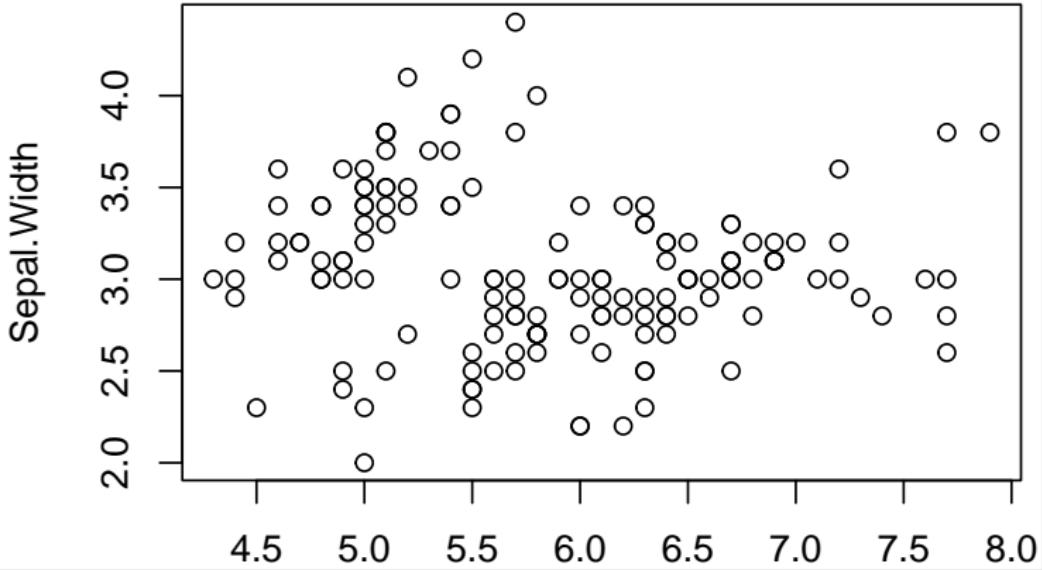
par : Reihenfolge der Rand-Argumente

```
plot(Sepal.Width~Sepal.Length, data=iris, axes=F, ann=F)  
mtext(c("margin / side 1",2:4), side=1:4, las=1) ; box()
```



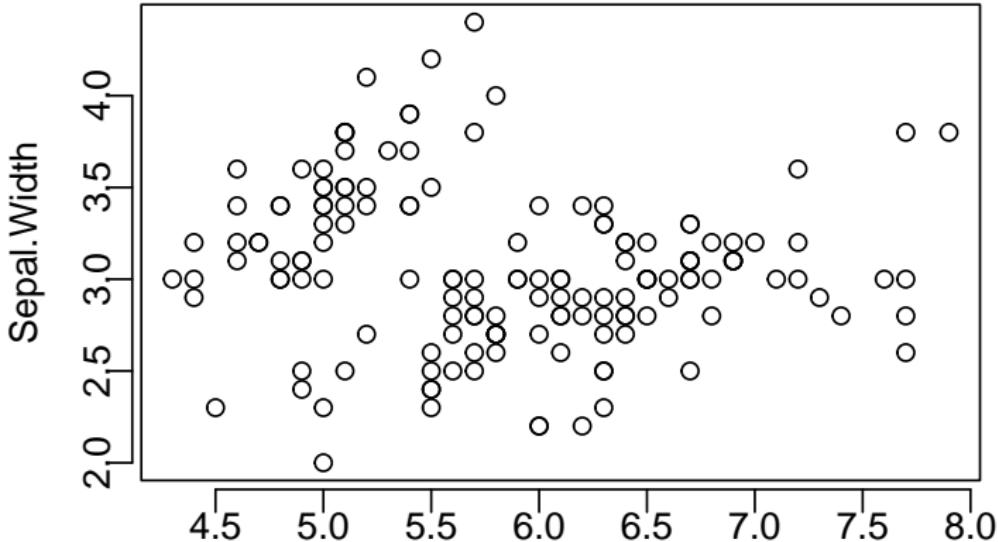
par : mar - margins = Ränder

```
par(mar=c(2,6,1,0.5)) # Einheit: Textzeilen  
plot(Sepal.Width~Sepal.Length, data=iris)
```



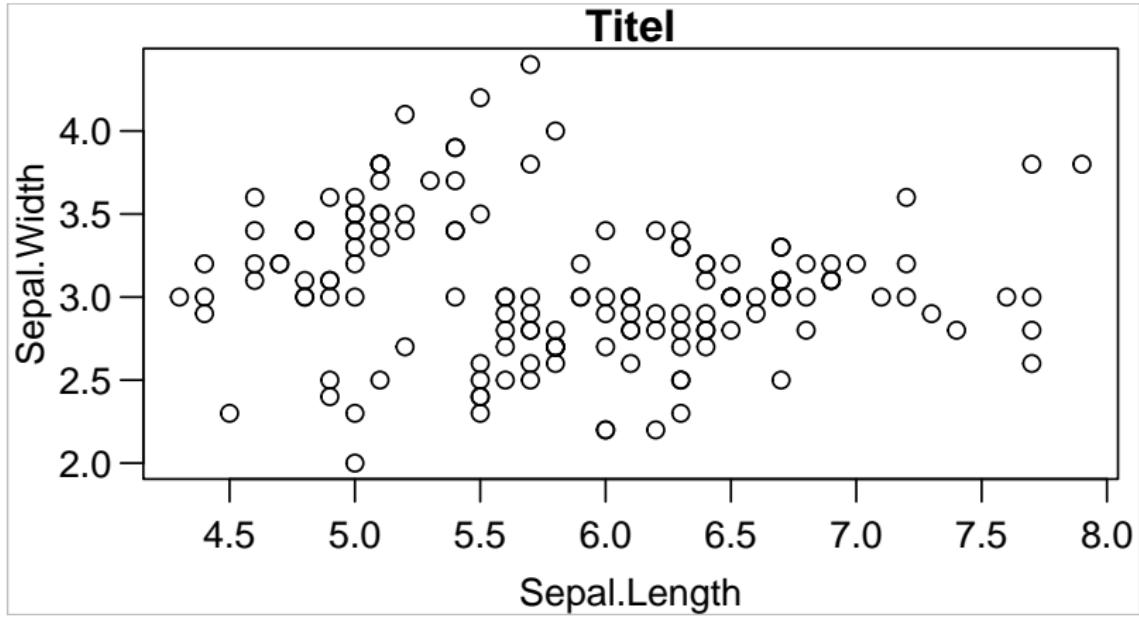
par : mgp - margin placements

```
par(mar=c(2,6,1,0.5), mgp=c(2.1, 0.5, 0.2)) # Abst. Titel,  
plot(Sepal.Width~Sepal.Length, data=iris) # Zahlen, Linie
```



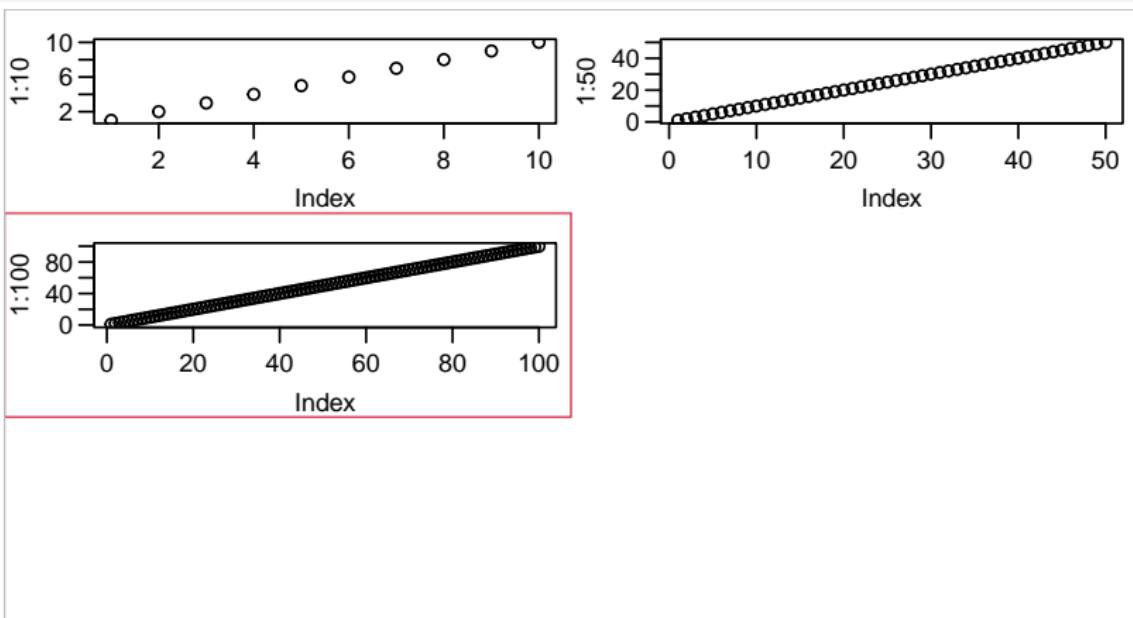
par : Häufig praktisch für minimalistische Ränder

```
par(mar=c(3,3,1,0.5), mgp=c(2,0.7,0), las=1)
plot(Sepal.Width~Sepal.Length, data=iris, main="Titel")
```



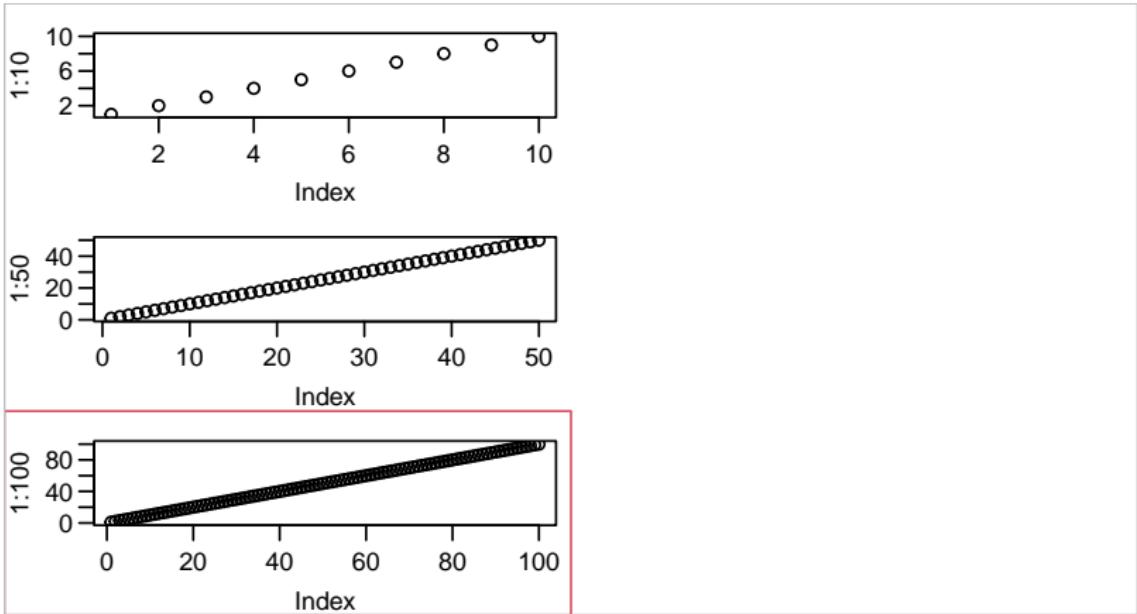
par : **mfrow** - **m**ultiple **f**igures, **r**owwise gefüllt

```
par(mfrow=c(3,2), mar=c(3,3,1,0.5), mgp=c(2,0.7,0), las=1)
plot(1:10); plot(1:50); plot(1:100); box("figure", col=2)
```



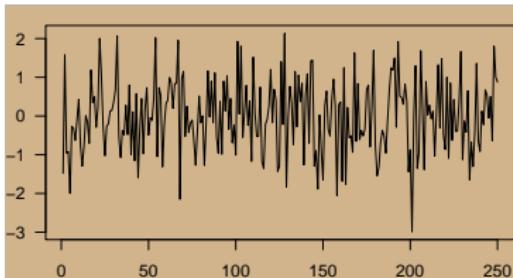
par : **mfcoll** - multiple figures, columnwise gefüllt

```
par(mfcoll=c(3,2), mar=c(3,3,1,0.5), mgp=c(2,0.7,0), las=1)
plot(1:10); plot(1:50); plot(1:100); box("figure", col=2)
```



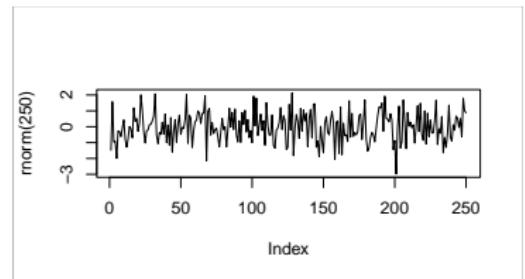
par Einstellungen wiederherstellen

```
op <- par(mar=c(2,2,1,0),
          las=1, bg="tan")
plot(rnorm(250), type="l")
box("figure", col="grey70")
```



```
op # ursprüngliche Parameter
## $mar
## [1] 5.1 4.1 4.1 2.1
## $las
## [1] 0
## $bg
## [1] "transparent"
```

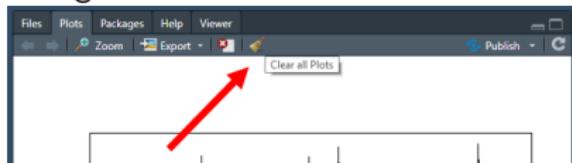
```
par(op)
plot(rnorm(250), type="l")
box("figure", col="grey70")
```



Alternative:

```
dev.off() # graphics.off()
```

Device schließen, beim nächsten Plot sind alle Standards wieder hergestellt.
Das geht auch in RStudio:



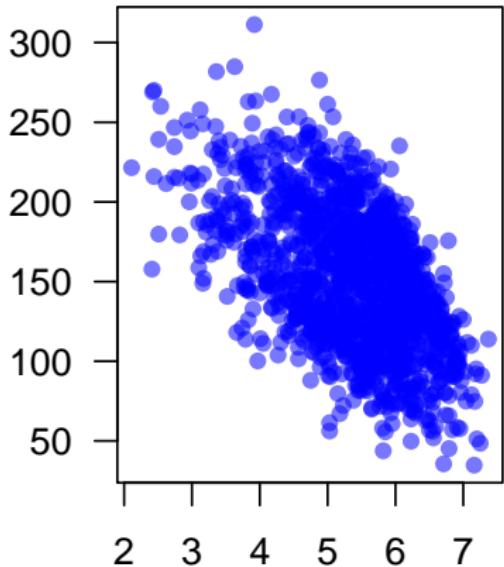
Multipanel Diagramme

`par`: Einstellungen für Grafiken (Parameter)

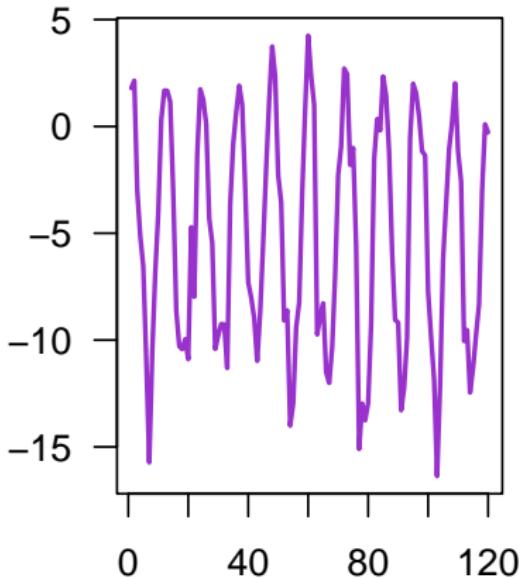
► `par (mfrow,mfcol, mar,mgp, las, bg)`

Zugspitze 1900:2020

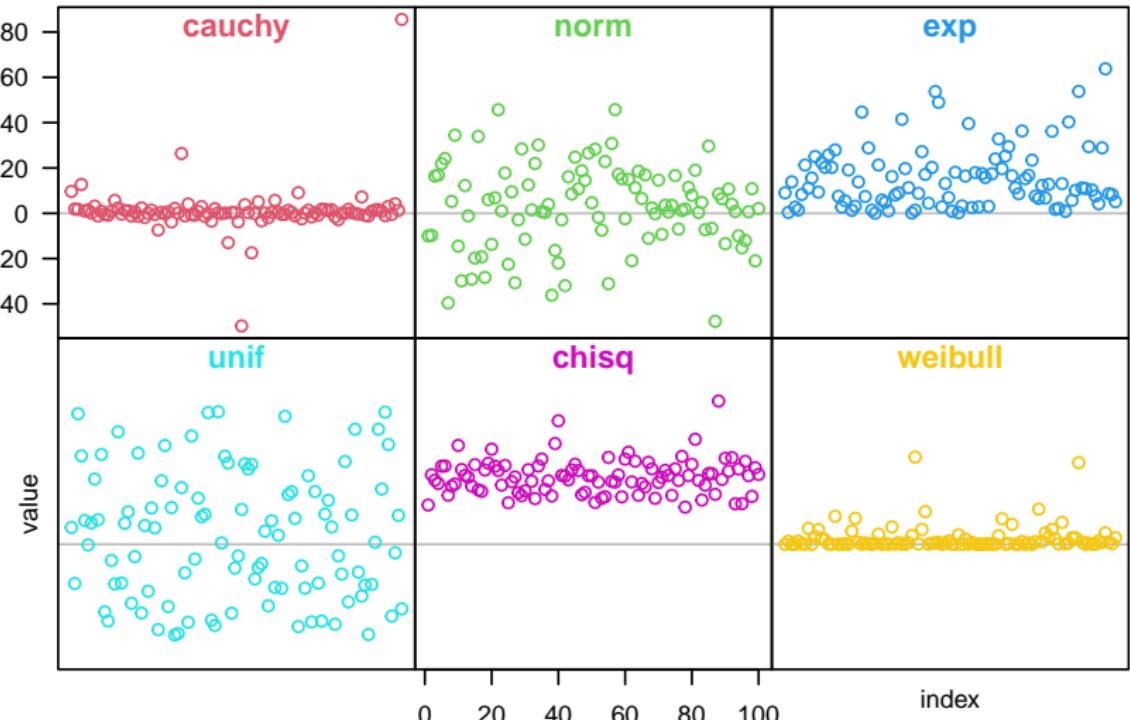
Sonnenscheindauer [Monatsstunden]
vs Wolkenbedeckung [Achtel]



Erste 120 monatliche Temperaturwerte



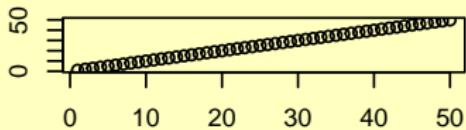
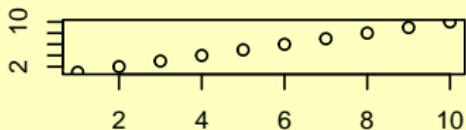
Lösung für Übung B2



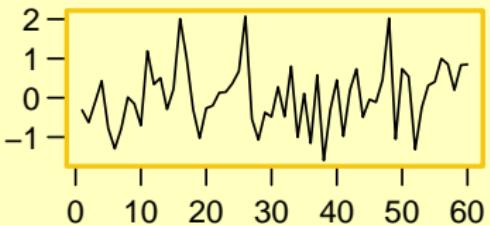
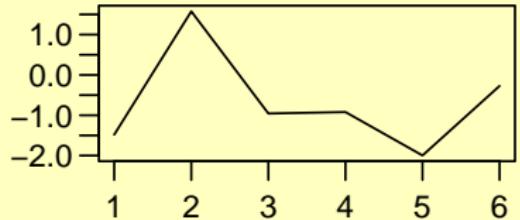
par: oma - outer margins

```
par(oma=c(0,4,3,0), mfcoll=c(3,2), mar=c(3,3,1,0.5) )  
plot(1:10); plot(1:50); title(main="header", outer=TRUE)
```

header



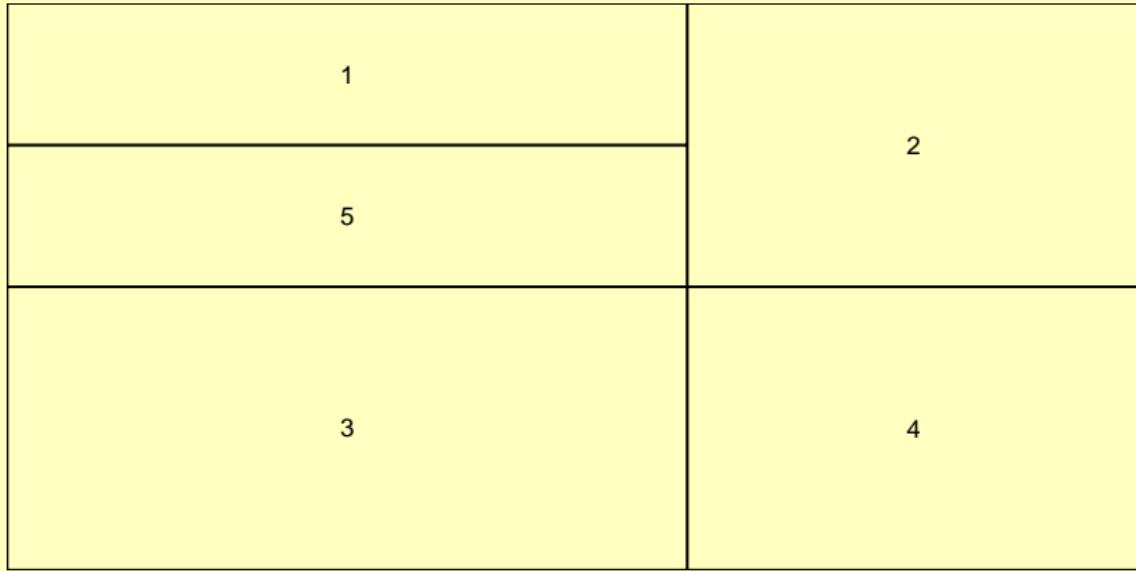
```
par(    mfrow=c(2,2)  , oma=c(0,1,3,0)      )
```



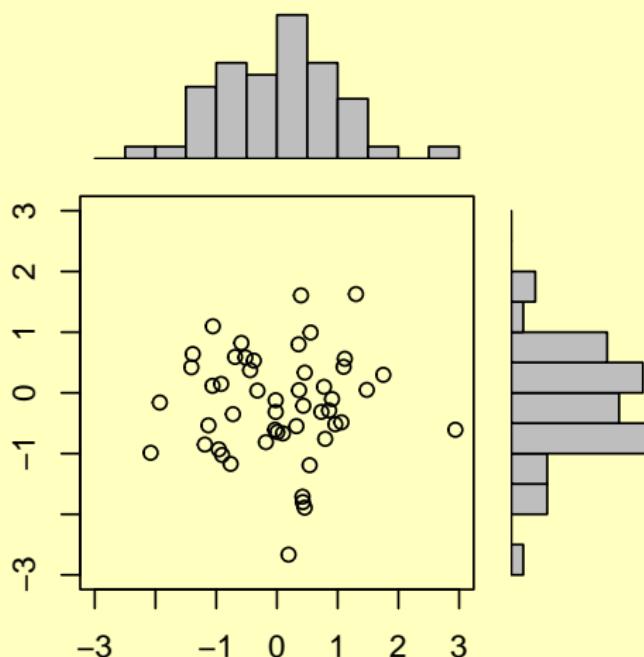
- box() = box('plot')
- box('figure'), after par(mfrow) or layout()
- box('inner')
- box('outer'), if outer margins were set by par(oma)



```
lay <- layout(matrix(c(1,1,2,2,  
                      5,5,2,2,  
                      3,3,4,4,  
                      3,3,4,4), ncol=4, byrow=TRUE),  
              widths=c(6,6,4,4))  
layout.show(lay)
```



Code im Examples-Abschnitt zu `?layout`:



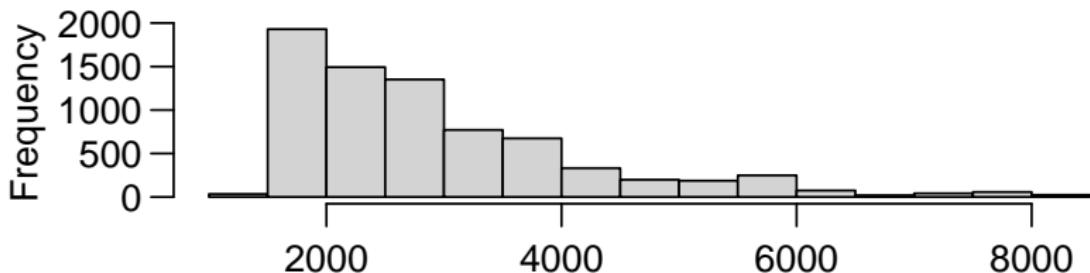
- 0. Intro
- 1. Grundlagen
- 2. Datentypen
- 3. Tabellen
- 4. Grafiken

- 4.1 Punktdiagramme
- 4.2 Liniendiagramme
- 4.3 Balkendiagramme
- 4.4 Hinzufügen
- 4.5 Komposition
- 4.6 Verteilungsplots
- 4.7 Exportieren
- 4.8 Ausblick

Histogramm: Anzahl Werte pro Bereich

```
h <- hist(EuStockMarkets, las=1, main="Aktienindex 90er")
```

Aktienindex 90er



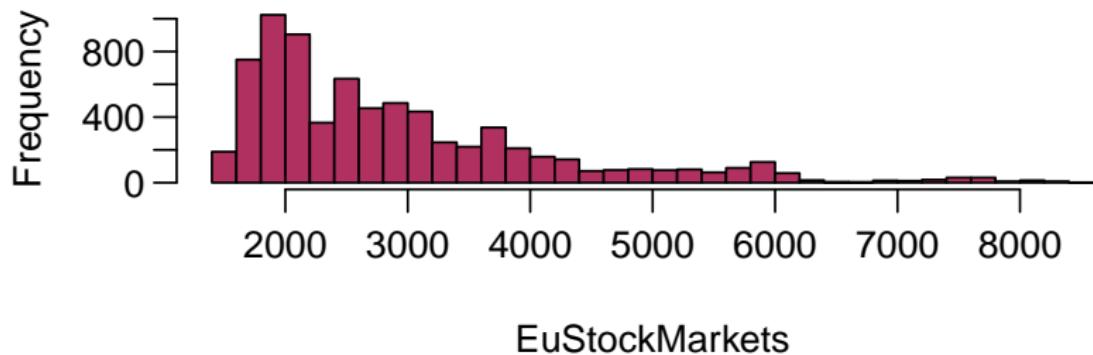
EuStockMarkets

```
str(h)
## List of 6
## $ breaks  : int [1:16] 1000 1500 2000 2500 3000 ...
## $ counts   : int [1:15] 34 1930 1493 1352 771 ...
## $ density  : num [1:15] 9.14e-06 5.19e-04 ...
## $ mids     : num [1:15] 1250 1750 2250 2750 3250 ...
## $ xname    : chr "EuStockMarkets"
## $ equidist: logi TRUE
```

hist : Anzahl bins (Klassen)

```
hist(EuStockMarkets, las=1, breaks=50, col="maroon")
```

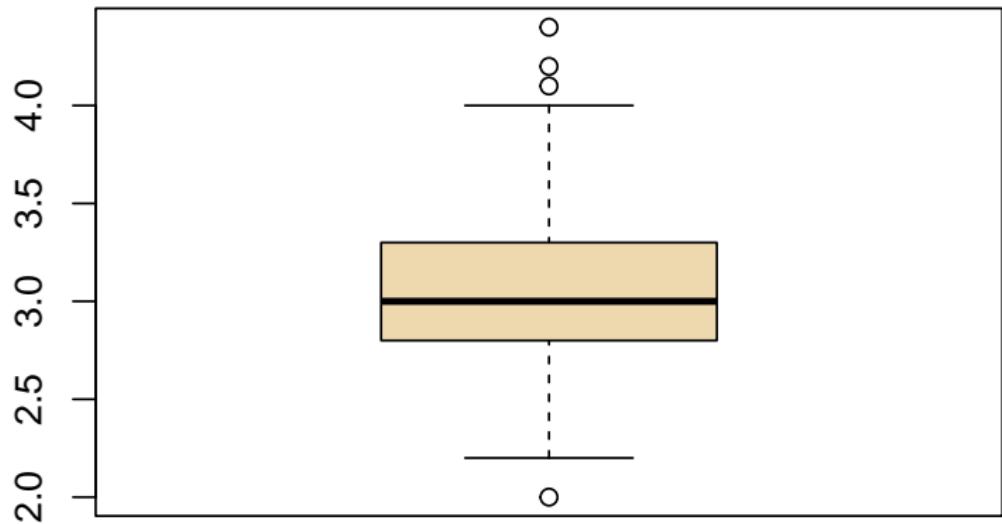
Histogram of EuStockMarkets



Boxplot: Quartile grafisch dargestellt

```
summary(iris$Sepal.Width)
##      Min. 1st Qu. Median      Mean 3rd Qu.      Max.
## 2.000   2.800   3.000   3.057   3.300   4.400

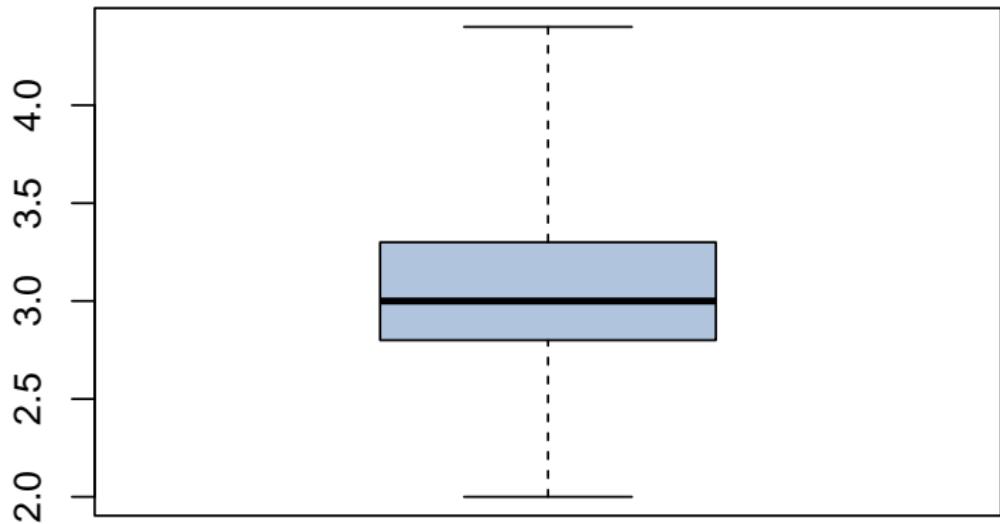
boxplot(iris$Sepal.Width, col="wheat2")
```



Punkte sind meist kein 'Outlier' sondern gehören dazu

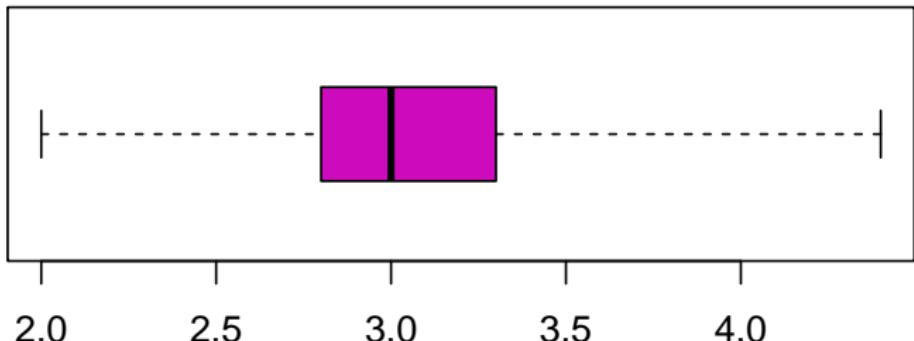
`boxplot()` zeichnet 'Ausreißer' standardmäßig als individuelle Punkte. Die Schwelle dafür, als solche eingestuft zu werden, ist eine willkürliche Distanz außerhalb der IQR Box. (IQR = InterQuartileRange = Spanne zwischen 25% und 75% Quantile der Daten)

```
boxplot(iris$Sepal.Width, range=0, col="lightsteelblue")
```

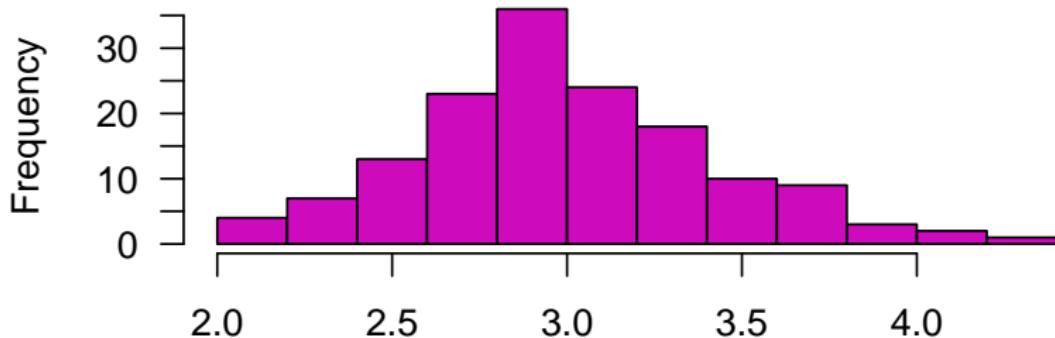


Horizontale Boxplots

```
boxplot(iris$Sepal.Width, horizontal=TRUE, range=0, col=6)
```

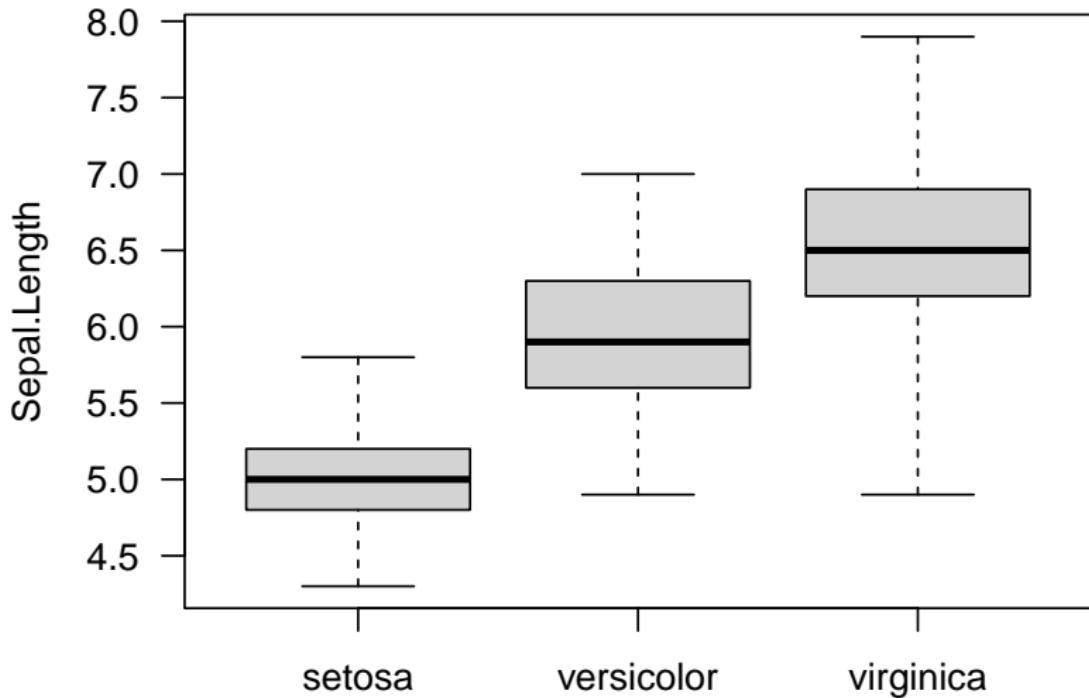


```
hist(iris$Sepal.Width, col=6, main="", las=1)
```



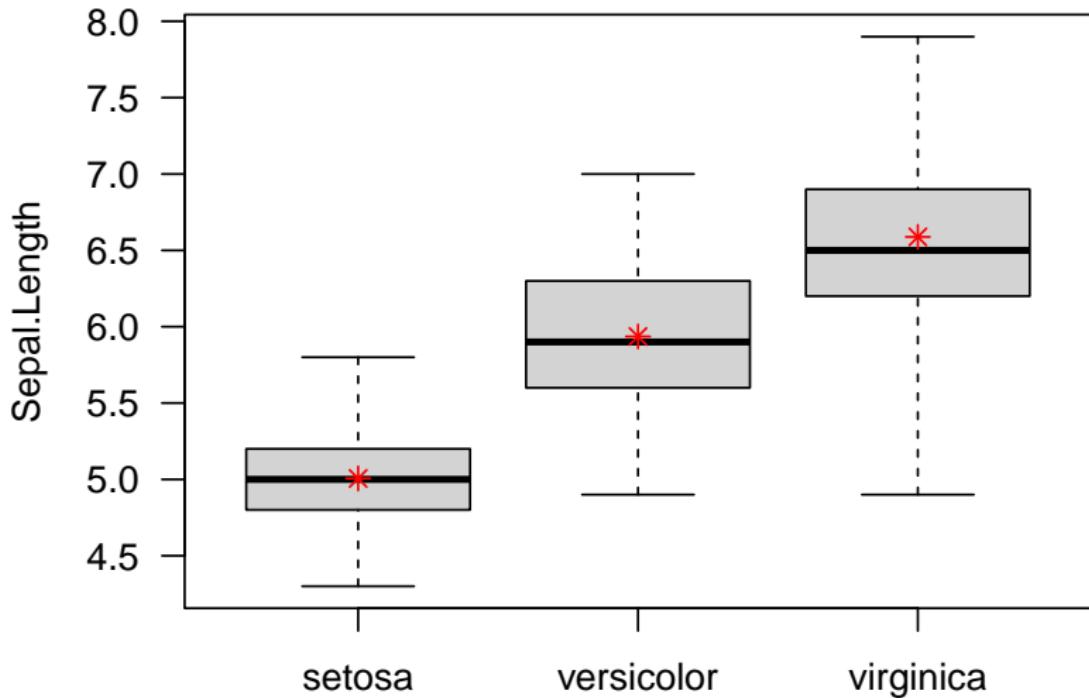
Boxplots mehrerer Spalten

```
boxplot(Sepal.Length ~ Species, data=iris, range=0, las=1)
# auch manchmal hilfreich (+ einfacher als bei barplots):
werte ~ gruppe1+gruppe2 # für doppelt gruppierte Daten
```



Boxplots mit arithmetischen Mittelwerten

```
boxplot(Sepal.Length ~ Species, data=iris, range=0, las=1)
averages <- tapply(iris$Sepal.Length, iris$Species, mean)
points(1:3, averages, pch=8, col="red")
```

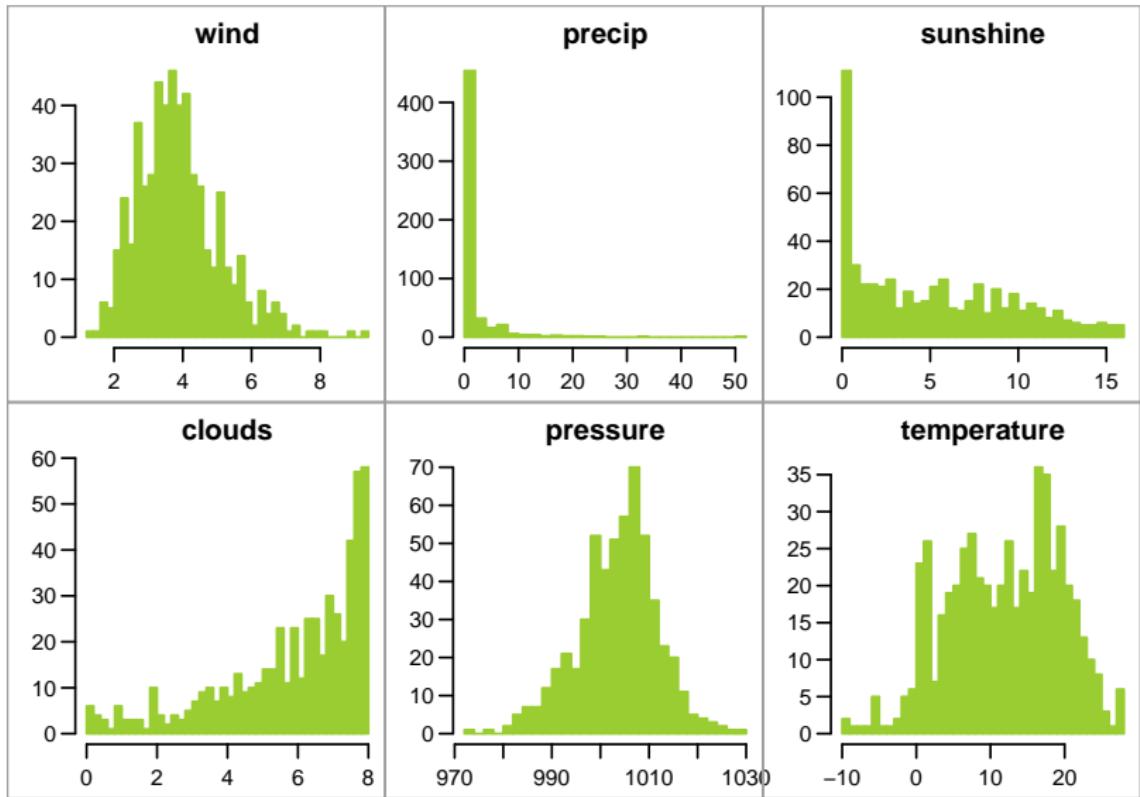


Zusammenfassung

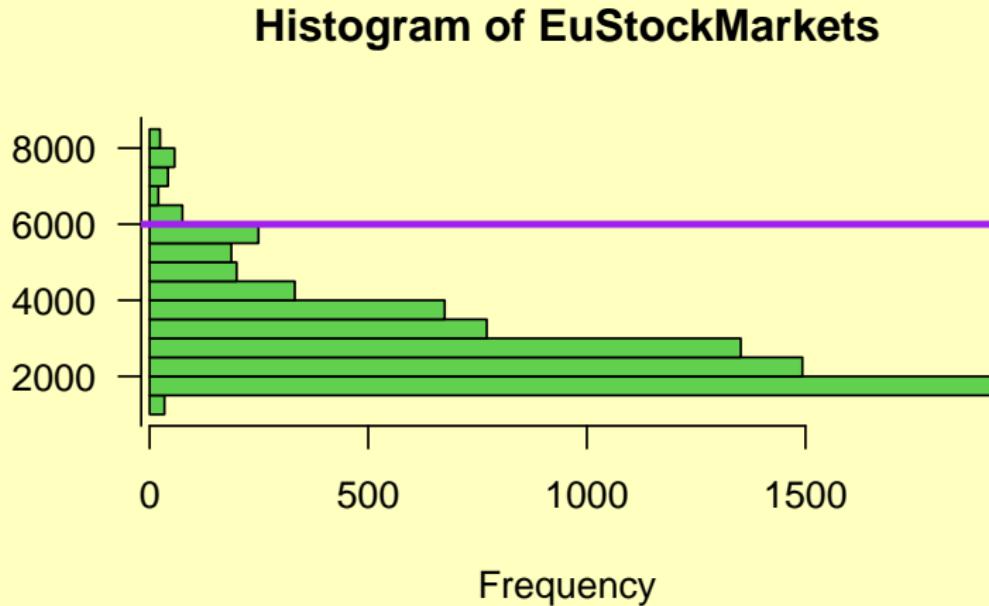
Histogramme und Boxplots: grafische Darstellung der Häufigkeitsverteilung

- ▶ `hist (breaks, las,col,main)`
- ▶ `boxplot (range=0, horizontal)`
- ▶ `boxplot (y ~ x)`
- ▶ Boxplots um viele Gruppen zu vergleichen, Histogramme um die tatsächliche Verteilung zu sehen

Mögliche Lösung zu A11

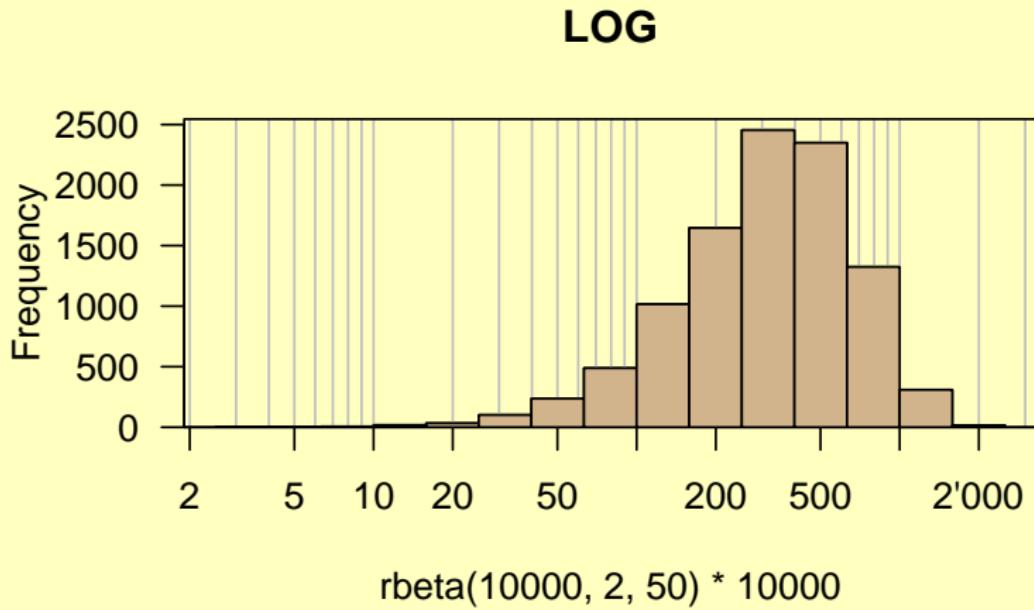


```
hpos <- berryFunctions::horizHist(EuStockMarkets, col=3)  
abline(h=hpos(6000), col="purple", lwd=3)
```



Logarithmisches Histogramm

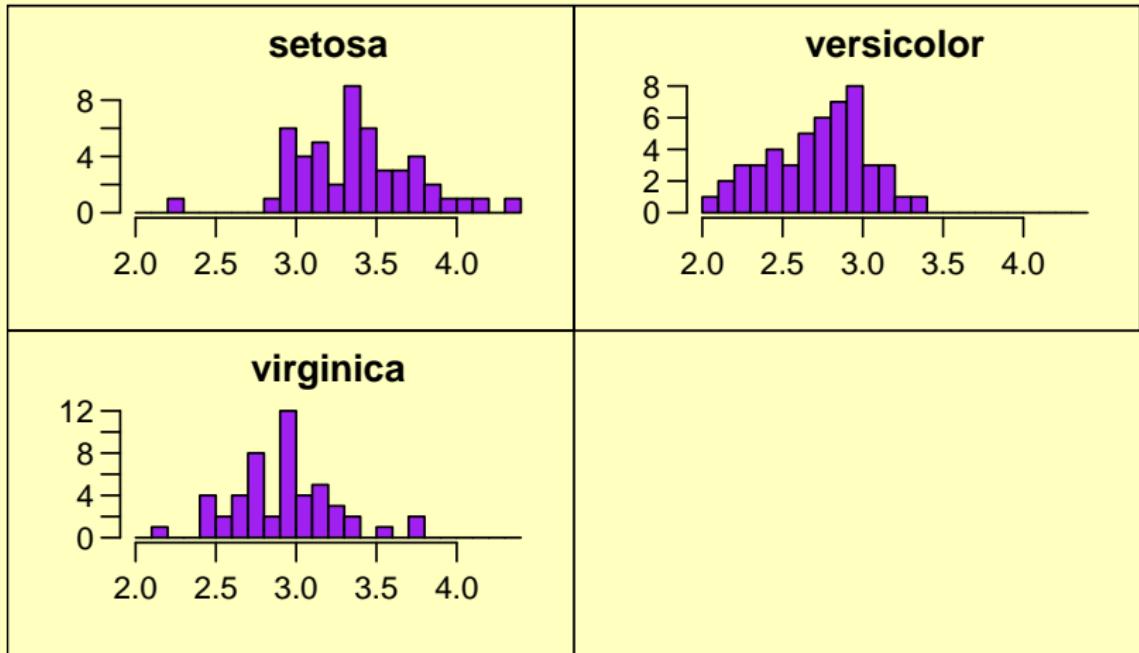
```
berryFunctions::logHist(rbeta(1e4, 2, 50)*1e4, main="LOG")
```



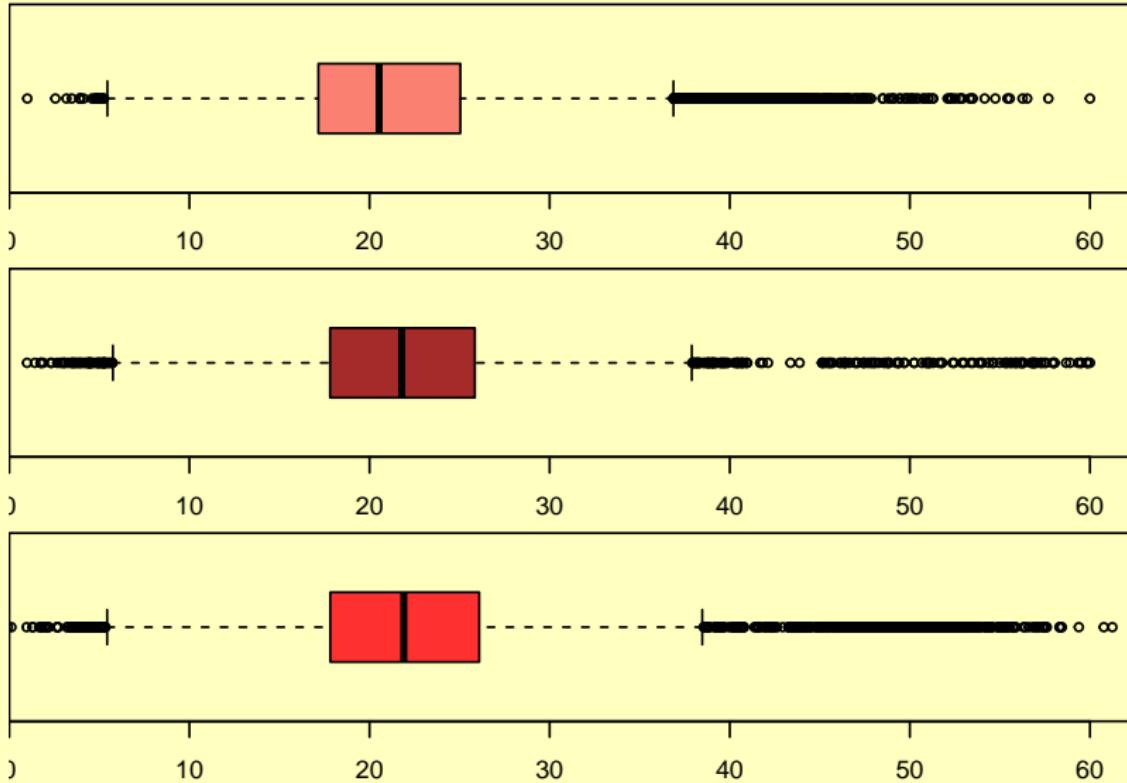
Gruppiertes Histogramm

```
berryFunctions::groupHist(iris, "Sepal.Width", "Species",  
                           unit="cm")
```

Histograms of Sepal.Width [cm] in iris, grouped by Species

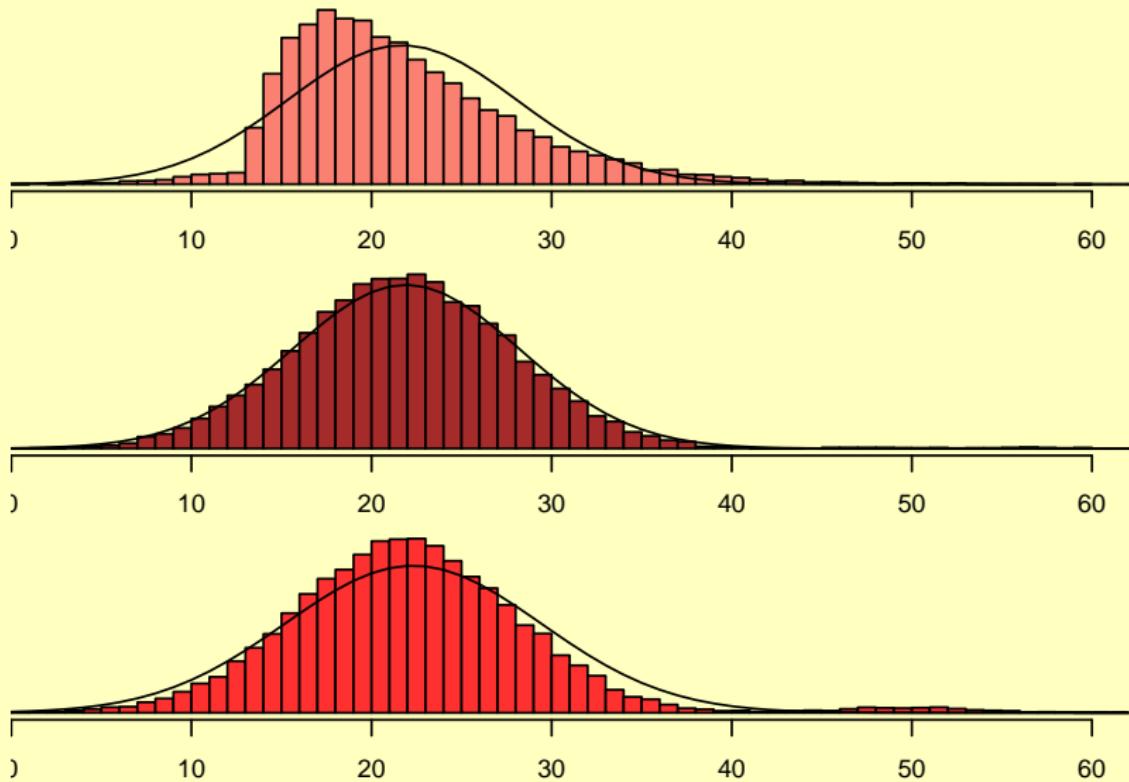


Boxplots sollten nur verglichen werden,



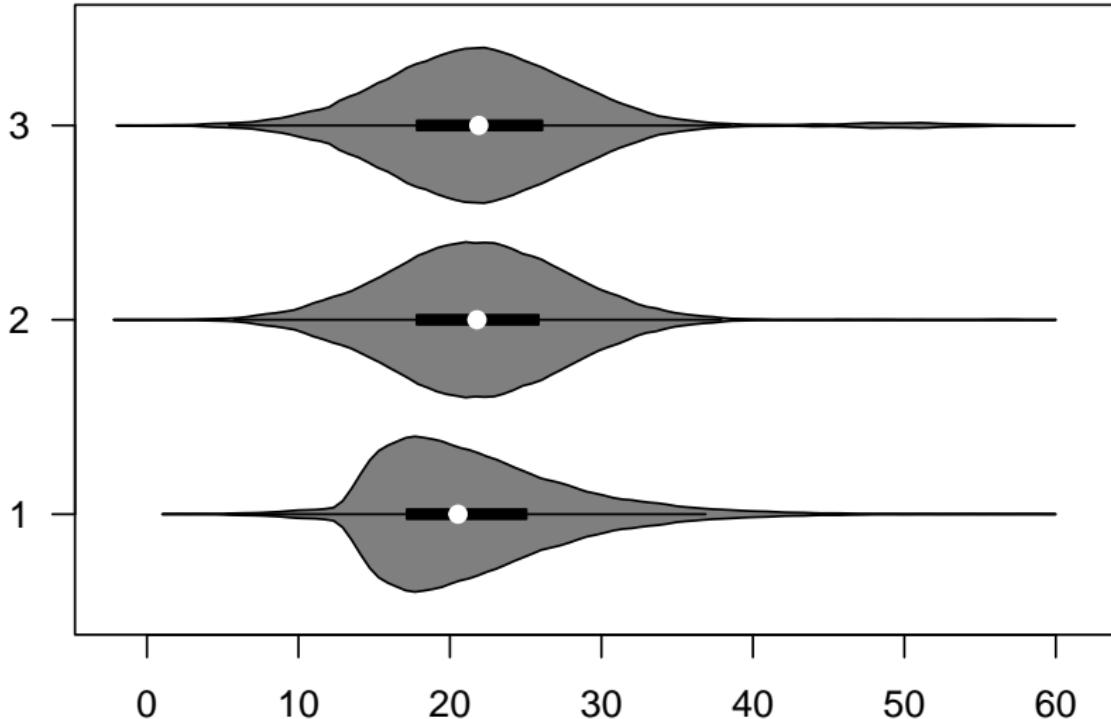
wenn die unterliegende Verteilungen gleich sind!

siehe z.B. why not to trust statistics und boxplot R



Violinplots: guter Mittelweg zwischen Boxplot und Histogramm

```
vioplot::vioplot(x,y,z, h=0.5, horizontal=TRUE)
```



- 0. Intro
- 1. Grundlagen
- 2. Datentypen
- 3. Tabellen
- 4. Grafiken

- 4.1 Punktdiagramme
- 4.2 Liniendiagramme
- 4.3 Balkendiagramme
- 4.4 Hinzufügen
- 4.5 Komposition
- 4.6 Verteilungsplots
- 4.7 Exportieren
- 4.8 Ausblick

Grafiken speichern als PDF

Dateiformate:

PDF : Vektor-grafik (zoombar), mehrere Seiten

PNG : Raster-Grafik, kleine Dateigröße

JPG : Anti-Aliasing, für Fotos (nicht für Grafiken)

```
pdf("MusterGrafik.pdf") # PDF-graphics-device öffnen  
par(mar=c(3,3,1,0), las=1)  
# Einstellungen für die exportierte Grafik  
plot(rnorm(700), col=colors(), col.axis="turquoise")  
# Grafik erscheint nicht im standard Rstudio Device  
dev.off() # externes Device schließen
```

pdf `width` und `height` sind per Default je 7 inches (17.8 cm).

Code einrücken macht Zusammengehörigkeit deutlich (für Menschen, nicht für R).

Mehrere Grafiken speichern

Jede Grafik hat eine eigene Seite im PDF:

```
pdf("MusterGrafik.pdf", height=5)
  plot(rnorm(700), col=colors(), col.axis="turquoise")
  boxplot(iris)
  hist(rnorm(700))
dev.off()
```

Zeilen gruppieren, geblockt an R senden

```
7777 # irgendwelcher vorhergehender Code
{
  pdf("MusterGrafik.pdf")
  boxplot(iris)
  hist(rnorm(700))
  dev.off()
}
888 # weitere andere Sachen
```

Tipps und Tricks zum Exportieren

Alles zwischen `pdf/postscript/png/jpeg/bmp/tiff/xfig/X11/svg` und `dev.off()` wird in die externe Grafik geschrieben.

Bei einem Fehler innerhalb geblockter Ausführung nicht vergessen, `dev.off()` manuell aufzurufen.

PDFs in Sumatra (nur auf Windows OS) öffnen, sodass sie editiert werden können, während sie geöffnet sind. Änderungen sind live sichtbar.

Sumatra Hinweise

Grafikdatei von R aus öffnen:

```
{ pdf("MusterGrafik.pdf")
  boxplot(iris)
  dev.off()
  file.show("MusterGrafik.pdf")
}
```

Grafiken speichern als PNG

```
png("DateiName.png", width=12, height=9, units="in",
    res=300, bg="transparent", pointsize=14)
par(mar=c(3,3,1,0.5), las=1)
plot(1:20)
title(main="Plot title")
points(5,9)
dev.off()
```

`units` in inches um ggf. auf `pdf` umzusteigen (kann nur inches, keine cm).

Fortlaufende Nummerierung bei mehreren Bildern:

```
png("Bild_%03d.png")
par(bg="thistle")
plot(2:8, type="h", lwd=7) ; plot(1:8)
dev.off() # ohne dev.off ist das letzte Bild noch leer
```

`_%03d.png` _001.png Gute Wahl :)

`_%3d.png` _ 1.png Leerzeichen in Dateinamen: generell keine gute Idee

`_%d.png` _1.png Sortiert nach Dateinamen kommt `20.png` vor `17.png`

Zusammenfassung

Grafiken speichern:

- ▶ `pdf("datei.pdf") ; plot(42) ; dev.off()`
- ▶ PDF-viewer verwenden, der geöffnete Dateien nicht gegen Bearbeitung sperrt
- ▶ `png (width, height, units, res, bg, pointsize)`
- ▶ `{}` um Code geblockt auszuführen

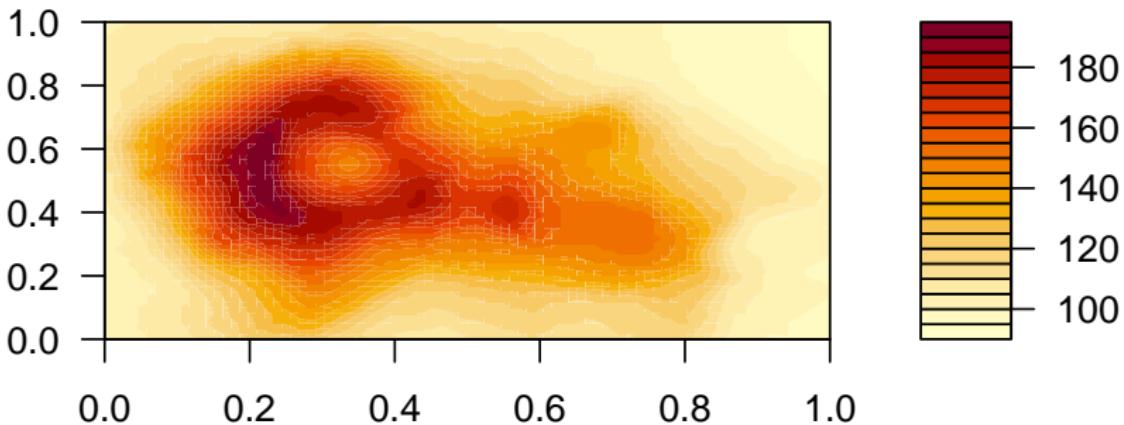
- 0. Intro
- 1. Grundlagen
- 2. Datentypen
- 3. Tabellen
- 4. Grafiken

- 4.1 Punktdiagramme
- 4.2 Liniendiagramme
- 4.3 Balkendiagramme
- 4.4 Hinzufügen
- 4.5 Komposition
- 4.6 Verteilungsplots
- 4.7 Exportieren
- 4.8 Ausblick

Animation

Mit ffmpeg:

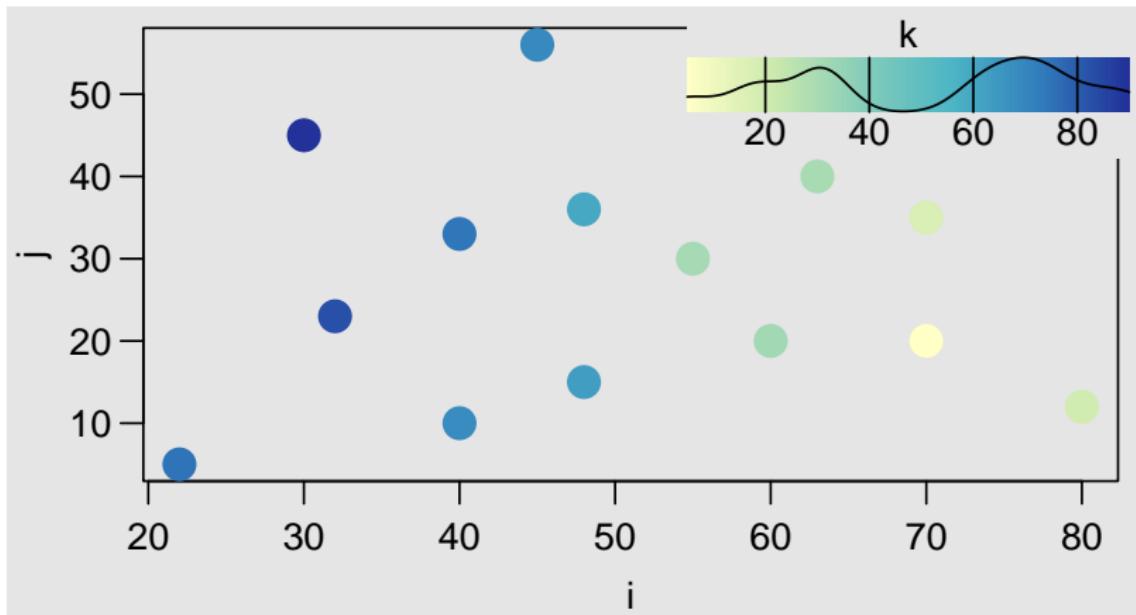
```
set.seed(12)
volc <- apply(volcano, 1:2, function(x) x+cumsum(rnorm(100)))
library(animation); library(pbapply)
saveVideo(pbsapply(1:100, function(i)
    filled.contour(volc[i,,], zlim=range(volc))),
    video.name="volc.mp4", interval=0.07,
    ffmpeg="C:/ff_folder/bin/ffmpeg.exe")
```



3D Punkte in Farbskala

```
i <- c(22, 40, 48, 60, 80, 70, 70, 63, 55, 48, 45, 40, 30, 32)  
j <- c( 5, 10, 15, 20, 12, 20, 35, 40, 30, 36, 56, 33, 45, 23)  
k <- c(75, 68, 63, 32, 20, 05, 17, 30, 31, 60, 69, 74, 90, 83)
```

```
berryFunctions::colPoints(i,j,k, add=FALSE, y1=0.75,  
                           density=list(bw=5), cex=2)
```



3D Linien in Farbskala

```
tfile <- system.file("extdata/rivers.txt", package="berryFunctions")
rivers <- read.table(tfile, header=TRUE, dec=",")
berryFunctions::colPoints(x,y,n, data=rivers, add=FALSE,
                           lines=TRUE, lwd=3, y1=0.8, density=FALSE)
```

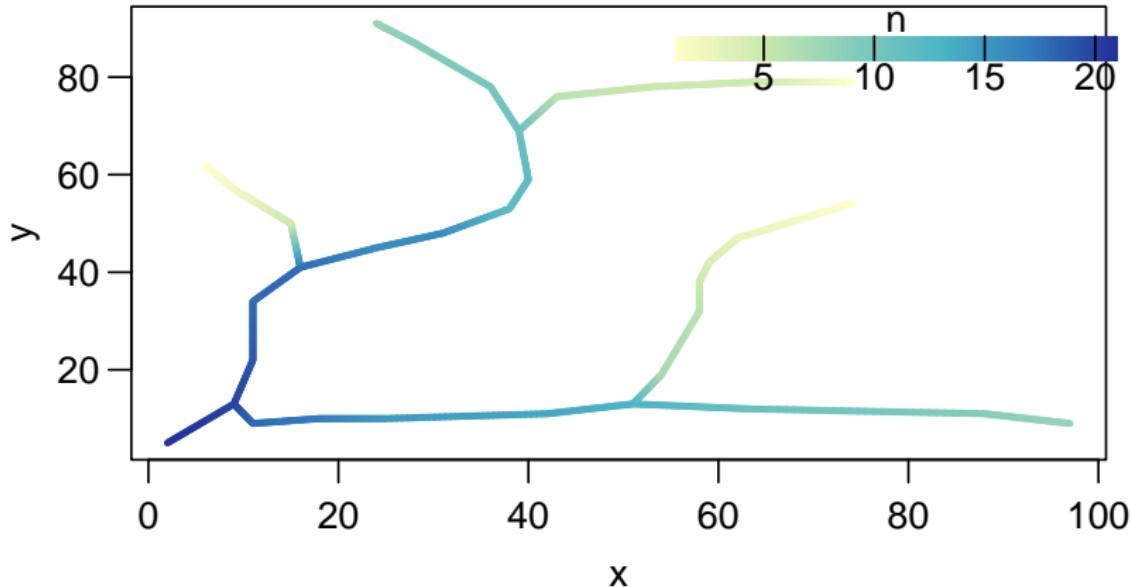


Tabelle in Farbskala

```
berryFunctions::tableColVal(t(longley[5:14,-6]), digits=1,
                             nameswidth=0.2, cex=0.6)
```

t(longley[5:14, -6])	1951	1952	1953	1954	1955	1956	1957	1958	1959	1960
GNP.deflator	96.2	98.1	99.0	100.0	101.2	104.6	108.4	110.8	112.6	114.2
GNP	329.0	347.0	365.4	363.1	397.5	419.2	442.8	444.5	482.7	502.6
Unemployed	209.9	193.2	187.0	357.8	290.4	282.2	293.6	468.1	381.3	393.1
Armed.Forces	309.9	359.4	354.7	335.0	304.8	285.7	279.8	263.7	255.2	251.4
Population	112.1	113.3	115.1	116.2	117.4	118.7	120.4	122.0	123.4	125.4
Employed	63.2	63.6	65.0	63.8	66.0	67.9	68.2	66.5	68.7	69.6

Dieses gesamte Kapitel erstellt Grafiken mit dem sogenannten base R.
Ein beliebter 'Dialekt' ist im Paket `ggplot2` verfügbar.
Beide Ansätze haben jeweils Vorteile und ihre Daseinsberechtigung.

Einige hilfreiche Links dazu:

<http://minimaxir.com/2017/08/ggplot2-web>

https://uc-r.github.io/ggplot_intro

<http://flowingdata.com/2016/03/22/comparing-ggplot2-and-r-base-graphics>

leaflet : interaktive Karten (Dieses Beispiel online)

```
library(rdwrd) ; data(geoIndex) ; library(leaflet)
leaflet(geoIndex) %>% addTiles() %>%
  addCircles(~lon, ~lat, radius=900, stroke=F, color=~col) %>%
  addCircleMarkers(~lon, ~lat, popup=~display, stroke=F, color=~col)
```



Zusammenfassung

Verschiedene anwendungsbezogene Grafiken:

- ▶ `berryFunctions::colPoints+tableColVal`
- ▶ `leaflet`
- ▶ `ggplot2` für einen anderen Dialekt

Für diese Lektion gibt es keine Übungsaufgaben

Programmieren mit R für Einsteiger

