

# R Kurs Unterlagen

Anna-Lena Schubert, Jan Goettmann, Jose Carlos Garcia Alanis, Meike Steinhilber, Cordula Hunzelmann

2021-10-13



# Inhaltsverzeichnis



# Kapitel 1

## Über dieses Buch

TEXT



## Kapitel 2

# Einführung

(Anna-Lena)





# Kapitel 3

## Datenstruktur

Thema	Inhalte
RMarkdown	<i>Titel, Chunks, knitten</i>
Hilfe	<i>help-Fenster, ?, #was passiert hier</i>
Werte, Vektoren & Listen	<i>chr, num, log, c(), list(), typeof(), coercion, Abruf von Elementen, list(list())</i>
Workspace	<i>rm(), Besen</i>
Berechnungen	<i>mit Values, Vektoren, Funktionen, z-Standardisierung</i>
Matrizen	<i>matrix(), Indizierung</i>
tidy Daten	<i>Zeilen: Beobachtungen, Spalten: Variablen</i>
tidyverse	<i>Installation und library (package)</i>
data.frame & tibble	<i>Unterschiede, as.data.frame(), as_tibble(), \$, [], Zugriff auf Elemente, Reihennamen, Faktoren</i>
Daten laden & speichern	<i>Import per klick, read./_, sep=, dec=, .xlsx, .svs, write_csv()</i>
Daten anschauen	<i>View(), head(), str(), count()</i>

### 3.1 RMarkdown

Das R Markdown Skript ist ein besonderes Dateiformat für R Skripte. Es enthält Fließtext und eingebetteten R Code:

```

1  ---
2  title: "R Markdown Titel"
3  author: "Name"
4  date: "8 10 2021"
5  output: html_document
6  ---
7
8  ```{r setup=chunk, include=FALSE}
9  knitr::opts_chunk$set(echo = TRUE)
10 library(tidyverse)
11 #ggf. Daten laden
12 ```
13
14 # Überschrift: Leerer R Markdown Code
15 Beschreibender Fließtext
16
17 ```{r}
18 #Code zum Kennzeichnen eines Chunks
19 ```
20 Text der Codeergebnis interpretiert
21
22 ## Unterüberschrift
23 Fließtext geht weiter
24 ```{r, echo=FALSE}
25 #dieser Code wird nicht gedruckt, nur die Ausgabe
26 ```
27

```

Knittet man dies Skript mit dem Wollknäul Button (5.) in der oberen Leiste, integriert es den ausgeführten Code mit dem Fließtext und druckt ein übersichtliches Dokument (html, pdf, txt oder doc). Das ist praktisch um z.B. Auswertungsergebnisse zu präsentieren.

1. Im Header werden Titel und Dokumententyp für das Ausgabe-Dokument festgelegt
2. Die Code Blöcke (**Chunks**) sind mit je drei rückwärts gestellten Hochkommata (**Backticks**) am Anfang und Ende des Chunks eingerahmt. Werden sie vom R Markdown Skript als solche erkannt, wird auch die Hintergrundfarbe automatisch abgeändert. Im ersten chunk sollten **globale Chunk Optionen** festgelegt, alle notwendigen **Packages** geladen und die **Daten** eingelesen werden.
3. Den Fließtext kann man mit Überschriften (#) und Unterüberschriften (##) strukturieren, im Code kennzeichnet # Kommentare
4. Zu Beginn eines Chunks muss man innerhalb einer geschwungenen Klammer spezifizieren("“{...}“):
  - Es ist möglich Code von anderen Programmiersprachen (z.B. Python oder TeX) einzubetten, standard ist **r**
  - (optional) Nach einem Leerzeichen: Einzigartiger Chunk-Name
  - (optional) Nach einem Komma: Befehle, um die Ausgabe des Chunks in das neue Dokument zu steuern:
    - **include = FALSE** Weder Code noch Ergebnis erscheinen
    - **echo = FALSE** Nur das Code-Ergebnis erscheint
    - **message = FALSE** Nachrichten zum Code erscheinen nicht
    - **warning = FALSE** Warnungen zum Code erscheinen nicht
    - **fig.cap = "..."** Hiermit lassen sich Grafiken beschriften

## 3.2 Hilfe

Sie merken, dass die Befehle und Funktionen zum Teil sehr spezifisch und Sie sich kaum alles behalten können. Am wichtigsten ist die Reihenfolge und Vollständigkeit der Zeichen: vergessen Sie ein Komma, ein Backtick oder eine Klammer zu, dann kann R den Code schon nicht interpretieren. Zum Glück erkennt R Studio das oft und weist einen darauf während des Codens mit einem roten **x** neben der Zeilennummer hin. Andernfalls dürfen Sie versuchen, die Fehlermeldung beim Ausführen zu verstehen.

Wenn Sie den Namen einer Funktion oder eines Packages nicht direkt erinnern, können Sie den Anfang des Namens im **Chunk** oder in der **Console** eingeben, RStudio bietet einem nach einem kurzen Moment eine Liste möglicher Optionen an, aus der Sie wählen können. Haben Sie eine Funktion gewählt, können sie die **Tab**-Taste drücken und es werden die verschiedenen Funktionsargumente angezeigt, um die Funktion zu spezifizieren, was oft sehr hilfreich ist. Möchten Sie wissen, was eine Funktion macht oder in welcher Reihenfolge die Funktionsargumente eingegeben werden, können Sie `?FUN` in die **Console** eintippen, wobei `FUN` Platzhalter für den Funktionsnamen ist. Alternativ können Sie im **Help**-tab unten rechts suchen. Die Dokumentation ist oft sehr ausführlich. Die Möglichkeit einschlägige Suchmaschinen im Internet zu verwenden ist fast zu trivial, um sie hier aufzuführen, oft werden Sie dabei auf **StackOverflow** weitergeleitet. Auf Englisch gestellte Fragen oder Probleme führen zu besseren Treffern. Noch trivialer ist es, im Skript des Kurses oder im eigenen Code nachzuschauen. Falls Sie gründlich nachlesen möchten, gibt es auch ganze Bücher, die einem eine Einführung in R geben: z.B. **R Cookbook** oder **R for Data Science**.

## 3.3 Werte & Vektoren

Datenformate in R sind von einfach zu komplex: **Value**, Vektor, **matrix**, (**array**), **data.frame**, **tibble** und **list**. Die kleinste Objekteinheit in R ist ein **Value**. Es gibt unterschiedliche Typen von **Values**:

1. Text, bzw. Charakter (**chr**), manchmal auch String genannt,
2. (komplexe Zahlen, **cmplx**)
3. Nummer (**num**), bzw. **double**
4. (ganze Zahlen, integer **int** genannt)
5. logische Werte (**logi**), manchmal auch Boolean genannt
6. fehlende Werte (**NA**), Not Available

Sie weisen einem Objektnamen einen Wert per `<-` zu (Shortkey:ALT&-), der Datentyp des **Values** wird automatisch Rkannt.

```
var1 <- "kreativ"   # typ chr
var2 <- 3.5          # typ num
var3 <- TRUE         # typ logi
```

Mit der Funktion `typeof()` können sie sich den Datentypen anzeigen lassen. Vektoren reihen Werten desselben Datentyps auf `c(Wert1, Wert2, ...)`:

```
vec1 <- c(3, 6, 3.4)    # c() kombiniert die Werte zu einem Vektor, der dem Namen zugeordnet ist
```

Fassen Sie Werte von verschiedenen Typen zu einem Vektor zusammen, werden beide Werte zum Typen mit der kleineren Typenzahl umgewandelt (*coercion*).

```
c("kreativ", 3.5)    # ich versuche ein `chr` und eine `num` zu einem Vektor zu kombinieren
```

```
## [1] "kreativ" "3.5"
```

3.5 wird in `ausgegeben`, die Nummer wurde zu Text.

### 3.3.1 Coercion (Umwandlung von Typen)

Sie können den Datentypen auch per Funktion ändern, z.B. `as.character()`, `as.double()`:

```
as.character(c(1, TRUE, "abc", 4.1627)) # Verändert eine Reihe von Werten zum Typ character
```

```
## [1] "1"      "TRUE"    "abc"     "4.1627"
```

```
as.double(c(2, TRUE, "abc", 4.1627))    # Verändert die Werte zum Typ double, geht es nicht
```

```
## Warning: NAs durch Umwandlung erzeugt
```

```
## [1] 2.0000    NA      NA 4.1627
```

Coercion gibt es auch in Matrizen, Arrays (Mehrdimensionale Matrizen) und in Spaltenvektoren von Datensätzen (`data.frames` und `tibbles`). Nur Listen können verschiedene Datentypen und Elemente enthalten `list(Element1, Element2, ...)`. Das geht soweit, dass Listen selbst Listen enthalten können.

### 3.3.2 Aufruf einzelner Elemente per Index:

Um auf Elemente zuzugreifen, können Sie deren Indexnummer verwenden:

```
vec_4 <- c(1, 3, 3, 7)    #Definition des Vektors
vec_4[2]                  #Abruf des zweiten Elements des Vektors
```

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

Das geht sogar in verschachtelten Listen:

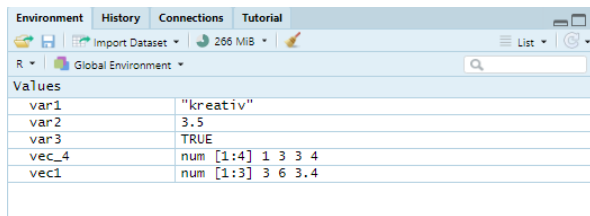
```
mylist <- list(list(1, "a"), c(2, 3)) # Definiert eine Liste aus Liste & Vector, die je a
mylist[[1]][2]                       # Ruft Element 1 der äußeren Liste: (1, "a"), und da
```

```
## [[1]]
## [1] "a"
```

Ich habe jetzt mehrere Variablen (Values, Vektoren, Listen) definiert, sie sind in meinem RStudio im **Environment**-tab oben rechts aufgetaucht.

## 3.4 Der Workspace

Rechts oben im Fenster ist das **Environment**-tab. Hier sieht man alle im **globalen Workspace** definierten Objekte (Datenstrukturen: Werte, Vektoren, Matrizen, Arrays, Listen, `data.frames`, `tibbles`; und Funktionen) aufgelistet:



Values	
var1	"kreativ"
var2	3.5
var3	TRUE
vec_4	num [1:4] 1 3 3 4
vec1	num [1:3] 3 6 3.4

Per Doppelklick können Sie die Objekte jeweils einzeln oben links im extra Fenster (**Datenansicht**-tab) anschauen. `rm(Objektname)` ist die Funktion zum Entfernen einzelner Objekte aus dem **globalen Workspace**. Das Besen-symbol im **Environment**-tab oben rechts fegt den **globalen Workspace** leer. Es ist zu beachten, dass R Markdown beim **knitten** nicht auf den **globalen Workspace** zugreift, sondern einen eigenen Workspace aus dem Code in den **Chunks** erstellt. Beim Ausführen einzelner **Chunks** per Markieren und **STRG/CTRL+Enter** oder **grüner Pfeil rechts** wird jedoch auf den **globalen Workspace** zugegriffen. Beim Schließen von RStudio werden Sie gefragt, ob Sie den **globalen Workspace** in die `.RData` als `img` speichern lassen, dann stehen die Objekte in der nächsten Sitzung wieder zur Verfügung, solange Sie dieselbe Projektdatei öffnen. Offene Skipte und offene **Datenansicht**-tabs werden beim Schließen ebenfalls mit der Projektdatei assoziiert. Geladene Packages gehen leider verloren, diese müssen Sie jedes Mal beim Starten von RStudio neu laden: `library(Packagename)`. Deshalb ist es Konvention am Anfang jedes Skriptes erstmal die Packages zu laden. Haben Sie Objekte im Workspace gespeichert, können Sie deren Namen verwenden, um sich auf diese zu beziehen und z.B. weitere Berechnungen vorzunehmen.

## 3.5 Einfache Berechnungen

### 3.5.1 Rechnen mit Values

```
x <- 5      # definiert den Wert der Variable x
y <- 5      # definiert den Wert der Variable x
x + y      # Summe von x und y
x*y        # Produkt von x und y
sqrt(x)    # Wurzel aus x
x**(1/2)   # x hoch 0.5
```

Möchten Sie das Ergebnis speichern, müssen Sie dem berechneten Wert einen Namen zuweisen:

```
z <- x + y  # weist dem Namen z das Ergebnis dieser Gleichung zu, "z" erscheint im Word
```

### 3.5.2 Rechnen mit Vektoren

Operationen können häufig vektorisiert, d.h. auf alle Objekte eines Vektors angewendet werden:

```
e <- vec_4 * 5 # multipliziert alle Elemente des Vektors mit 5 und speichert das Ergebnis
```

Es gibt eine ganze Reihe vorgefertigter Funktionen, um mit Vektoren zu rechnen:

### 3.5.3 Übersicht Berechnungsfunktionen

Folgende Funktionen können Sie auf **num**-Vektoren und Matrizen anwenden, je nach Funktion auch auf **chr** Vektoren oder Datensätze, wobei diese sich dann meist nur auf die Einträge in der oberen Ebene, z.B. auf die Anzahl der Spalten und nicht auf die Spalteneinträge beziehen.

Funktion	Bedeutung	Funktion	Bedeutung
<b>min(x)</b>	Minimum	<b>mean(x)</b>	Mittelwert
<b>max(x)</b>	Maximum	<b>median(x)</b>	Median
<b>range(x)</b>	Range	<b>var(x)</b>	Varianz
<b>sort(x)</b>	sortiert x	<b>sd(x)</b>	Standardabweichung
<b>sum(x)</b>	Summe aller Elemente	<b>quantile(x)</b>	Quantile von x
<b>cor(x,y)</b>	Korrelation von x und y	<b>length()</b>	Länge von x

### 3.5.4 Beispiel einer z-Standardisierung eines Vektors mit 3 Einträgen

```

geschwister <- c(8,4,12)           # def. der Variable
mw_geschw <- mean(geschwister)    # MW
mw_geschw

## [1] 8

sd_geschw <- sd(geschwister)       # SD
sd_geschw

## [1] 4

z_geschw <- (geschwister-mw_geschw)/sd_geschw # z-Standardisierung des Vektors
z_geschw

## [1] 0 -1 1

```

## 3.6 Matrizen

Matrizen sind 2D Datenstrukturen, sie bestehen aus Vektoren gleicher Länge und enthalten einen Datentyp. Mit dem Befehl `matrix()` können sie erstellt werden:

```

mat <- matrix(NaN, nrow=4, ncol=4) # Eine Matrix mat mit 4 Reihen, 4 Spalten und leeren Einträgen
                                   # NaN (Not a Number) ist zwar ein double, aber rechnen kann man nicht
mat

##      [,1] [,2] [,3] [,4]
## [1,]  NaN  NaN  NaN  NaN
## [2,]  NaN  NaN  NaN  NaN
## [3,]  NaN  NaN  NaN  NaN
## [4,]  NaN  NaN  NaN  NaN

```

Ich habe eine 4x4 Matrix erstellt, die mit NaNs gefüllt ist. Hätte ich diverse Datentypen zugeordnet, wären diese zum niedrigeren `coerced` worden. Matrizen können mit `matrixname[Zeile,Spalte]` manipuliert werden. Beispiel:

```

mat[,1] <- vec_4 # Weil Spalte 1. von mat und vec_4 dieselbe Länge haben, kann ich Spalte 1 neu zugeordnen
mat             # Dadurch, dass der Eintrag für die Zeilennummer leer ist, beziehe ich mich auf vec_4

##      [,1] [,2] [,3] [,4]
## [1,]    1  NaN  NaN  NaN
## [2,]    3  NaN  NaN  NaN
## [3,]    3  NaN  NaN  NaN
## [4,]    7  NaN  NaN  NaN

```

```
mat[,2] <- 8 # Wird einem Bereich ein einzelner Wert zugeordnet, wird dieser vervielfacht
mat
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    1    8  NaN  NaN
## [2,]    3    8  NaN  NaN
## [3,]    3    8  NaN  NaN
## [4,]    7    8  NaN  NaN
```

```
mat[,3] <- c(FALSE, TRUE) # Wird eine ganzzahlige Teilmenge (2 von 4) zugewiesen, wird
mat                               # Anm.: das nennt sich recyceln,
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    1    8    0  NaN
## [2,]    3    8    1  NaN
## [3,]    3    8    0  NaN
## [4,]    7    8    1  NaN
```

Coercion: TRUE wurde zu 1 und FALSE wurde zu 0. Wenn man nun eine bestimmte Spalte oder Zeile sehen möchte, kann man dies über die Indizierung tun, hierbei kann man sich beliebig austoben.

```
mat[,1] # Ich möchte nur die erste Spalte über alle Zeilen ausgeben
```

```
## [1] 1 3 3 7
```

```
mat[1,1] # Nur den ersten Wert der ersten Spalte
```

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

Hier wird es turbulent:

```
mat[c(1,3),] # Nur Zeile 1 und 3 von allen Spalten
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    1    8    0  NaN
## [2,]    3    8    0  NaN
```

```
mat[-1,2:4] # Nicht Zeile 1 und Spalten 2-4
```

```
##      [,1] [,2] [,3]
## [1,]    8    1  NaN
## [2,]    8    0  NaN
## [3,]    8    1  NaN
```

Da ich jetzt Bereiche der Matrix auswählen kann, lohnt es sich Berechnungen vorzunehmen

```
(mat[,2]+mat[,1])*mat[,3] # Spalte 2 minus Spalte 1 und dann mal Spalte 3
```

```
## [1]  0 11  0 15
```



Es sind immer noch nicht angegebene Nummernwerte in der Matrix, solange ich mich beim Berechnen auf Bereiche der Matrix beschränke, die vergebene numerische Werte haben, gibt es kein Problem, ansonsten schon:

```
mat[1,] # Zeile 1
```

```
## [1] 1 8 0 NaN
```

```
sum(mat[1,]) # Summe über Zeile 1 mit NaN
```

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

Die Summe kann nicht berechnet werden. Zum Auslassen der NaNs wird das Funktionsargument `na.rm=TRUE` verwendet:

```
sum(mat[1,], na.rm=TRUE) # Summe über Zeile 1 ohne NaN
```

```
## [1] 9
```

```
mean(mat, na.rm=TRUE) # MW der Matrix ohne NaN
```

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

Nun, da wir mit dem Rechnen in Matrizen vertraut sind möchte ich die letzte Spalte mit Einträgen füllen:

```
typeof(mat) # gebe den Typ der Einträge der Matrix aus
```

```
## [1] "double"
```

```
mat_sav <- mat # in weiser Voraussicht speichere ich die bisherige Matrix unter anderem  
mat[,4] <- c("coercion", "kann", "nervig", "sein") # weise Spalte 4 einen Vektor mit chr Einträgen  
mat
```

```
##      [,1] [,2] [,3] [,4]  
## [1,] "1"  "8"  "0"  "coercion"  
## [2,] "3"  "8"  "1"  "kann"  
## [3,] "3"  "8"  "0"  "nervig"  
## [4,] "7"  "8"  "1"  "sein"
```

```
typeof(mat)
```

```
## [1] "character"
```

Könnte ich eben noch den Mittelwert einer Spalte bilden, so geht das jetzt nicht mehr, da alle Einträge der Matrix zu `chr` coerced wurden. In einem typischen Datensatz sind aber Variablen verschiedener Typen (`num` und `chr`) enthalten. Dieses Problem ließe sich mit Listen lösen, welche aber unübersichtlich sind. Datensätze bestehen manchmal aus unüberschaubar vielen Einträgen und deshalb müssen sie übersichtlich strukturiert sein.

### 3.7 tidy Daten

Es gibt eine Konvention dafür, wie man Datensätze, die mehreren Beobachtungseinheiten (Fällen) verschiedene Parameter (Variablen) zuordnet. Wichtig für die eigene strukturierte Arbeit ist in erster Linie Konsistenz, z.B. dass Sie bei Variablennamen aus mehreren Wörtern immer den Unterstrich als Trennzeichen verwenden. Es hat sich als überlegen für die Auswertung von Daten herausgestellt, Fälle in Zeilen und Variablen in Spalten einzuordnen, dieses Prinzip dürfte Einigen schon von SPSS bekannt sein.

	Variable1	Variable2	Was ist 'tidy' data?
Fall1	Wert11	Wert12	<u>Eine Zeile pro Beobachtung</u>
Fall2	Wert21	Wert22	<u>Eine Spalte pro Variable</u>
Fall3	Wert31	Wert32	Eine Tabelle pro Untersuchung
Fall4	Wert41	Wert42	eindeutige Namen
Fall5	Wert51	Wert52	Konsistenz
Fall6	Wert61	Wert62	...

Es gibt noch weitere Regeln für konsistentes und ordentliches Arbeiten in R und mit Datensätzen im Allgemeinen, z.B. dass man keine Farbcodierungen verwenden sollte. Vorerst genügt es, wenn Sie sich an die Basics hier halten. Diese Art Daten zu strukturieren lässt sich im `data.frame` und noch besser im `tibble` umsetzen: Beides sind Tabellen mit Spaltenvektoren, die jeweils verschiedene Datentypen enthalten können. Deswegen stellen beide das bevorzugte und für unsere Zwecke wichtigste Datenformat dar.

#### 3.7.1 tidyverse

Bevor wir uns dem schönsten Datenformat, den `tibbles` widmen, müssen wir das entsprechende Package einmalig in der `Console` installieren. Ich habe den Code auskommentiert, weil das Package bei mir bereits installiert ist:

```
#install.packages("tidyverse") # R kennt den Namen noch nicht, deswegen ""
```

Das Package `tidyverse` enthält eine Reihe nützlicher Packages, die eine saubere Datenverarbeitung zum Ziel haben. Packages müssen bei jeder Sitzung neu aktiviert bzw. angehängt werden. Für uns relevante Packages im `tidyverse` sind `tibble`, `readr`, `stringr`, `dplyr`, `purrr` und `ggplot2`.

```
library(tidyverse) # Bitte an den Anfang eines Skriptes, um beim Starten der R Sessi
```

### 3.8 data.frames (df) und tibbles (tib)

Es gibt mehr Gemeinsamkeiten als Unterschiede zwischen beiden Datenformaten. Beides sind Tabellen mit Spaltenvektoren (Variablen), die je verschiedene Datentypen enthalten können. Hier zunächst die Übersicht über die Funktionen zum Managen des Datensatzes:

	Funktion zum	<code>data.frame()</code>	<code>tibble()</code>
Datenformat konvertieren		<code>as.data.frame()</code>	<code>as_tibble()</code>
Definieren		<code>data.frame(var1,...)</code>	<code>tibble(var1,...)</code>
Aufrufen des Datensatzes		<code>df</code>	<code>tib</code>
Auswählen einer Variable		<code>df\$var</code>	<code>tib\$var</code>
Auswählen eines Bereiches		<code>df[rowIdx,colIdx]</code>	<code>tib[rowIdx,colIdx]</code>
Definieren neuer Variablen		<code>df\$var_neu &lt;- c(...)</code>	<code>tib\$var_neu &lt;- c(...)</code>
Reihennamen vergeben		<code>row.names(df) &lt;- c("name1",...)</code>	<code>relocate(tib,var)</code>

Sie können die beiden Datensatz-Formate einfach in das jeweils andere konvertieren. Die Definition geht per Formel `data.frame()` und die Aneinanderreihung der Spaltenvektoren. Es bietet sich an, dabei direkt Namen für die Spaltenvektoren zu vergeben. Für tibbles geht das analog mit der Formel `tibble()`

```
test_df <- data.frame("text"=mat[,4] ,           # Komma zwischen Spaltenvektoren
                     "ist_Verb"=mat_sav[,3])    # verwende die Spaltenvektoren aus den vorherigen
test_df
```

```
##      text ist_Verb
## 1 coercion      0
## 2   kann       1
## 3 nervig       0
## 4   sein       1
```

In Bezug auf weitere Funktionen des Packages `tidyverse` sind tibbles ein wenig praktischer. Große tibbles werden übersichtlicher angezeigt (nur die ersten 10 Zeilen) wenn man sie aufruft.

```
test_tib <- as_tibble(test_df)
test_tib
```

```
## # A tibble: 4 x 2
##   text      ist_Verb
##   <chr>      <dbl>
## 1 coercion      0
## 2 kann         1
## 3 nervig       0
## 4 sein         1
```

Einzelne Spalten können ganz einfach aufgerufen werden, in dem man den `$`-Operator benutzt. Schreibt man diesen direkt hinter den Namen des Datensatzes, klappt automatisch eine Liste mit allen Spalten auf:

```
test_tib$text      # Mit dem $-Operator können einzelnen Spalten direkt per Name adressiert werden

## [1] "coercion" "kann"      "nervig"   "sein"
```

Es ist auch möglich, mehrere Zeilen und/oder Spalten auszugeben, dies funktioniert wie bei Matrizen per Indexnummer:

```
test_tib[2:4,1] # Die Zeilen 2 bis 4 werden aus Spalte 1 ausgegebenen

## # A tibble: 3 x 1
##   text
##   <chr>
## 1 kann
## 2 nervig
## 3 sein
```

Die Adressierung einzelner Spalten und Zeilen ermöglicht dann zum Beispiel die Berechnung von Kennwerten nur für einzelnen Spalten. Z.B. kann man die Kosten für Konzertkarten im Jahr 2022 aufsummieren lassen:

```
tickets_2022 <- tibble(Artist=c("Ed Sheeran", "Billy Ellish", "The Weeknd", "Dua Lipa"),
                       Kosten=c(79.32, 282, 116, 136, 68.71 ))
sum(tickets_2022$Kosten)

## [1] 682.03
```

Der `$`-Operator wird für fast alle höheren Datentypen verwendet, um auf diese zuzugreifen. Dies gilt zum Beispiel auch für die meisten Outputs von Funktionen (t-Test, Anova, SEMs) und Listen, es müssen aber wie im tibble Namen für die Listeneinträge vergeben sein:

```
liist_of_things <- list(tibbi = test_tib,      # erstellt eine Liste aus diversen Objekten
                       ticki = tickets_2022,  # den Namen der Listeneinträge werden
                       geschwi = geschwister, # die Objekte aus dem Workspace zugeordnet
                       vari = var1)

liist_of_things$geschwi                        # per $-Operator und Name in der Liste

## [1] 8 4 12

liist_of_things$ticki # Und so die Variable ticki (hier das tickets_2022 tibble)
```

```
## # A tibble: 5 x 2
##   Artist      Kosten
##   <chr>      <dbl>
## 1 Ed Sheeran    79.3
## 2 Billy Ellish  282
## 3 The Weeknd   116
```

```
## 4 Dua Lipa          136
## 5 Imagine Dragons   68.7
```

Theoretisch könnte ich auch noch `\$Artist` hinzufügen, dann würde mir nur die erste Spalte des tibbles ausgegeben. Mir fällt auf, ich habe den Namen eine Künstlerin in `tickets_2022` falsch geschrieben:

```
tickets_2022$Artist[2] <- "Billy Eilish" # $-Operator und Indexing lassen sich kombinieren
```

Sie können also nicht nur Elemente aus Datensätzen abrufen, sondern diese mit dem `<-` neu zuweisen. Man kann das `$` auch verwenden um neue Spalten in die Datensätze einzufügen:

```
tickets_2022$Priorität <- c(2, 4, 3, 5, 1) # definiert eine neue Spalte im Datensatz
tickets_2022$Prioritaet <- tickets_2022$Priorität # besser als statt ä in Variablenamen
tickets_2022                                     # nun gibt es eine Spalte zu viel
```

```
## # A tibble: 5 x 4
##   Artist      Kosten Priorität Prioritaet
##   <chr>      <dbl>    <dbl>    <dbl>
## 1 Ed Sheeran    79.3         2         2
## 2 Billy Eilish  282          4         4
## 3 The Weeknd   116          3         3
## 4 Dua Lipa     136          5         5
## 5 Imagine Dragons 68.7         1         1
```

```
tickets_2022$Priorität <- NULL # entfernt eine Spalte, vorsichtig hierbei(!)
```

Ein weiterer Unterschied zwischen tibbles und `data.frames` ist, dass tibbles keine Reihennamen kennen. Das vereinfacht das Format. Möchten Sie trotzdem gerne Reihennamen vergeben, dann müssen Sie sich mit einer neuen Variable (z.B. Reihename) behelfen, die Sie mit `relocate(tib, var)` an den Anfang des Datensatzes stellen.

### 3.8.1 Faktoren

Vektoren, die kategoriale Einträge enthalten können Sie mit `factor()` als Faktor zuweisen:

```
Gegut_vec <- c("m", "f", "d", "f", "f", "m", "f", "f", "f", "m", "m", "f", "f", "m", "f", "m", "f")
Gegut_fac <- factor(Gegut_vec) # macht Faktor aus kategorialem Vector und speichert
Gegut_fac # ruft den Faktor auf:
```

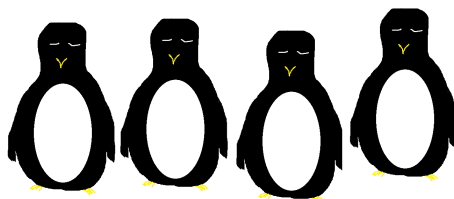
```
## [1] m f d f f m f f f m m f m f f m f f m
## Levels: d f m
```

Es werden die einzelnen Ausprägungen und die möglichen Ausprägungen als Levels ausgegeben. `levels()` gibt nur die möglichen Ausprägungen eines Faktors aus. Faktoren eignen sich oft besser als Vektoren zum Plotten und Rechnen,

deswegen ist es nützlich kategoriale Spaltenvektoren im Datensatz zu Faktoren zu machen. Jetzt, wo Sie mit dem Management von Datensätzen vertraut sind wollen wir vorhandene Datensätze einlesen:

### 3.9 Einlesen und Speichern von Daten

Daten können in R Studio auf unterschiedliche Weise eingelesen werden. Ich habe dieses Bild zur Abwechslung für Ihre Augen erstellt.



Es gibt frei verfügbare Datensätze in Packages, z.B. einen Datensatz zu Pinguinen: `palmerpenguins`.

Horst AM, Hill AP, Gorman KB (2020). `palmerpenguins`: Palmer Archipelago (Antarctica) penguin data. R package version 0.1.0. <https://allisonhorst.github.io/palmerpenguins/>

Nach einmaliger Installation des Packages (`install.packages("palmerpenguins")`) muss es geladen werden:

```
library(palmerpenguins) # jedes Mal beim Durchlaufen des Skripts. Ohne ""
pengu <- penguins      # penguins ist zwar schon ein tibble, aber Namenszuweisung zum
```

In der Regel werden Sie aber einen selbst erhobenen oder einen aus dem Internet heruntergeladenen Datensatz einlesen wollen. Mein Tipp ist, den Datensatz in das Working Directory zu speichern, dann finden Sie ihn schneller und er ist in der Nähe Ihrer Auswertung, noch eleganter ist es einen Unterordner namens `data` in den Ordner des Working Directories anzulegen, in den Sie alle Datensätze zu ihrem Projekt speichern können. Im **File**-tab unten rechts navigieren Sie zu der Datei mit dem Datensatz und dann klicken Sie diese zum Importieren des Datensatzes an (alternativ können Sie im **Environment**-tab über den Button **Import Dataset** einen Datensatz zum Importieren auf ihrem Computer suchen). In RStudio erscheint ein Fenster zum Importieren, unten rechts wird der automatisch der dem Dateiformat und unten links angegebenen Optionen entspricht, ggf. werden sogar benötigte Packages geladen. Um einen Datensatz per Code zu importieren sind Dateiformat, die Trennzeichen (`sep`) und die Dezimalzeichen (`dec`) besonders relevant. Das Standard-Dateiformat ist `.csv`, hier sind Kommata Trennzeichen (`sep=","`) und Punkte kennzeichnen Dezimalstellen (`dec="."`). Sie können die Funktionen `read_csv()` oder `read_delim()` für dieses Dateiformat verwenden, letztere sollte Trenn- und Dezimalzeichen auto-

matisch erkennen. Hier ist eine Übersicht zu den Einlesefunktionen in base R (also ohne zusätzlich geladene Packages) und im tidyverse Package, der Unterschied ist, dass base R Funktionen die Daten in einen `data.frame` laden, tidyverse Funktionen in ein tibble:

Funktion zum	sep	dec	in base R	im tidyverse
<u>autolesen</u>	auto	.	<code>read.delim()</code>	<code>read_delim()</code>
autolesen	auto	,	<code>read.delim2()</code>	<code>read_delim2()</code>
lesen von	,	.	<code>read.csv()</code>	<code>read_csv()</code>
lesen von	leer	.	<code>read.table()</code>	<code>read_table()</code>
<u>schreiben</u>	,	.	<code>write.csv()</code>	<code>write_csv()</code>

Wichtigstes und oft einziges Funktionsargument ist der vollständige Dateiname, er wird in " angegeben. Falls Sie die Datei in einem Unterordner vom Working Directory gespeichert haben, wird den Name des Unterordners mit einem / dem Dateinamen vorangestellt (z.B. "data/Datensatz1.csv"). Das Einlesen von Daten funktioniert nur, wenn der einzulesende Datensatz per `<-` einem Namen zugewiesen wird. Beispiel zum Laden eines .csv Datensatzes:

```
socken <- read_delim("data/socken.csv") # liest meinen socken.csv Datensatz aus dem Unterordner
socken                                # Daten in ein tibble namens socken
```

```
## # A tibble: 2 x 3
##   Stoff Gewicht Bewertung
##   <chr>   <dbl>     <dbl>
## 1 Seide    0.03         10
## 2 Wolle    0.08          9
```

Excel Dateien werden mit Funktionen `read_excel()`, `read_xls()` oder `read_xlsx()` aus dem Package `readxl`, SPSS Dateien mit der Funktion `read_svs()` aus dem Package `haven` eingelesen. Auch zum Einlesen von SAS, Stata oder anderen Dateiformaten gibt es entsprechende Funktionen. Die Standardfunktion zum Abspeichern von Datensätzen in eine Datei ist `write_csv()`, bzw. in base R `write.csv()`, da dieses Dateiformat die beste Kompatibilität mit anderer Software aufweist. Beim Speichern muss man neben dem Dateinamen und ggf. Dateipfad noch den Namen des Datensatzes als erstes Funktionsargument angeben:

Es gibt noch ein weiteres erwähnenswertes Dateiformat, das von R selbst: `.RDS`. Die Funktionen `saveRDS()` und `readRDS()` bieten die beste Funktionalität in R.

### 3.10 Datensätze (dat) anschauen

Um sich die geladenen Daten ganz anzuschauen kann man sie im `Workspace` anklicken, oder deren Namen an die Funktion `view(dat)` übergeben. `head(dat)` zeigt einem den Kopf des Datensatzes an, genau genommen die ersten 6 Zeilen:

```
long_tib <- tibble(Person_Id=c(1:20),
                  Gender=Gegut_fac,
                  Eigenschaft=var1) # definiert ein 20 x 3 tibble,
head(long_tib)                   # zeigt die ersten 6 Zeilen jeder Variable an
```

```
## # A tibble: 6 x 3
##   Person_Id Gender Eigenschaft
##   <int> <fct>   <chr>
## 1         1 m      kreativ
## 2         2 f      kreativ
## 3         3 d      kreativ
## 4         4 f      kreativ
## 5         5 f      kreativ
## 6         6 m      kreativ
```

Einen Überblick über die Datenstruktur, inklusive Factor-levels erhält man mit der Funktion `str(dat)`:

```
str(long_tib) # zeigt die Struktur der Daten
```

```
## tibble [20 x 3] (S3: tbl_df/tbl/data.frame)
## $ Person_Id : int [1:20] 1 2 3 4 5 6 7 8 9 10 ...
## $ Gender    : Factor w/ 3 levels "d","f","m": 3 2 1 2 2 3 2 2 3 ...
## $ Eigenschaft: chr [1:20] "kreativ" "kreativ" "kreativ" "kreativ" ...
```

Zeile 1 gibt Auskunft über Größe und die Klasse des Objektes, tibbles sind eine Art `data.frame`. In den weiteren Zeilen werden die Datentypen ggf. Faktorlevel und die ersten Werte angezeigt. Mit der Funktion `count(dat,var)` lassen sich die Häufigkeiten der Levels eines Faktors ausgeben:

```
count(long_tib, Gender) # zählt im long_tib die Levels des Faktors Gender
```

```
## # A tibble: 3 x 2
##   Gender      n
##   <fct> <int>
## 1 d         1
## 2 f        12
## 3 m         7
```



## 3.11 Einführung in Dplyr und tidyverse

Dplyr ist Teil des tidyverse Packages und ermöglicht es, Daten sehr einfach zu manipulieren und in eine Form zu bringen, um diese dann zu analysieren. Um das zu tun greifen wir auf den Star Wars Datensatz zurück, den das dplyr Package mitliefert:

```
# Lest die Daten bitte ein, der Datensatz heisst "starwars.RDS" und befindet sich in eurem Projekt
starwars <- readRDS("starwars.RDS")
```

Der Datensatz enthält Informationen über unsere Star Wars Helden, ähnlich dem Datensatz, den wir uns in der letzten Sitzung ausgedacht haben:

```
head(starwars,5) # Wir lassen uns erstmal die ersten 5 Zeilen des Datensatzes ausgeben
```

```
## # A tibble: 5 x 11
##   name          height mass hair_color skin_color eye_color Age sex gender
##   <chr>         <int> <dbl> <fct>    <fct>    <fct>    <dbl> <fct> <fct>
## 1 Luke Skywalker   172    77 blond     fair      blue      19  male masculi~
## 2 C-3PO            167    75 <NA>      gold      yellow    112 none masculi~
## 3 R2-D2            96    32 <NA>      white, blue red       33 none masculi~
## 4 Darth Vader      202   136 none      white     yellow    41.9 male masculi~
## 5 Leia Organa      150    49 brown     light     brown     19  fema~ femini~
## # ... with 2 more variables: homeworld <chr>, species <chr>
```

Bevor wir einsteigen, schaut euch an, wie die einzelnen Variablen im Datensatz verteilt sind. Benutzt dazu den `summary()` Befehl, was fällt euch auf ?

```
summary(starwars)
```

```
##      name          height          mass          hair_color  skin_color
## Length:87      Min.    : 66.0    Min.    : 15.00    none    :37    fair    :17
## Class :character 1st Qu.:167.0    1st Qu.: 55.60    brown   :18    light   :11
## Mode  :character Median :180.0    Median : 79.00    black   :13    dark    : 6
##              Mean   :174.4    Mean   : 97.31    white   : 4    green   : 6
##              3rd Qu.:191.0    3rd Qu.: 84.50    blond   : 3    grey    : 6
##              Max.   :264.0    Max.   :1358.00    (Other): 7    pale    : 5
##              NA's    : 6      NA's    :28      NA's    : 5    (Other):36
## eye_color      Age          sex          gender
## brown  :21    Min.    : 8.00    female    :16    feminine :17
## blue   :19    1st Qu.: 35.00    hermaphroditic: 1    masculine:66
## yellow :11    Median : 52.00    male      :60    NA's      : 4
## black  :10    Mean   : 87.57    none      : 6
## orange : 8    3rd Qu.: 72.00    NA's      : 4
## red    : 5    Max.   :896.00
## (Other):13    NA's    :44
## homeworld      species
```

```
## Length:87          Length:87
## Class :character   Class :character
## Mode :character    Mode :character
##
##
##
##
```

### 3.12 Dplyr: Die wichtigsten Befehle

- Filtern von Beobachtungen nach Wert (`filter()`).
- Reihen neu Sortieren (`arrange()`).
- Auswahl von Variablen nach Name (`select()`).
- Erstellen von neuen Variablen aus bereits existierenden (`mutate()`).
- Viele Werte zu einem einzelnen Wert zusammenfassen (`summarise()`).

Der vielleicht wichtigste Befehl ist der `group_by()` Befehl, mit dem Ihr die oben genannten Befehle auf einzelne Gruppen innerhalb eines Datensatzes anwenden könnt.

Diese 6 sogenannten “Verben” bilden die Grundlage für tidyverse. Damit ist es möglich mehrere einfache Schritte miteinander zu verketteten, um ein komplexes Ergebnis zu erzielen. Alle Befehle funktionieren auf die gleiche Art und Weise:

1. Das erste Argument ist ein Dataframe.
2. Die nachfolgenden Argumente beschreiben, was mit dem Dataframe geschehen soll, wobei die Variablennamen (ohne Anführungszeichen) verwendet werden.
3. Das Ergebnis ist ein neuer Dataframe

Hier ein Beispiel, zum `filter()` Befehl, dazu müsst ihr auch wissen, wie Ihr die gewünschten Beobachtungen mit Hilfe der Vergleichsoperatoren auswählen können. R bietet euch hier die Standardoperatoren:

1. `>` (größer)
2. `>=` (größer gleich)
3. `<` (kleiner)
4. `<=` (kleiner gleich)
5. `!=` (nicht gleich)
6. `==` (gleich)