

# R-Vorkurs


## Teil 1

Jens Klenke


28.03.2022



# Übersicht

1.  und *RStudio*
2. Grundlagen
3. Vektoren
4. Matrizen

# und *RStudio*

 ist eine freie Programmiersprache für statistische Berechnungen/Grafiken und wurde von Statistikern für Anwender mit statistischen Aufgaben entwickelt.

## Was ist *RStudio*? Brauche ich das?

*RStudio* ist eine integrierte Entwicklungsumgebung und grafische Benutzeroberfläche für .

*RStudio* bietet eine Menge an Komfortfunktionen, die das Arbeiten mit  vereinfachen.

⇒ **Wir arbeiten daher mit RStudio**

**Wichtig: *RStudio* ist nicht gleich ! Ohne eine -Installation bringt *RStudio* nichts.**

# Grundlagen

## Grundrechenarten

 kann als einfacher Taschenrechner benutzt werden.

```
5 + 7  
2 - 8  
12 * 12  
8 / 3
```

```
## [1] 12  
## [1] -6  
## [1] 144  
## [1] 2.666667
```

# Grundlagen

## ... und mehr

```
# "12 hoch 2"  
12^2  
  
# Logarithmus von 144 zur Basis 12  
log(144, base = 12)  
  
# "e hoch 3"  
exp(3)  
  
# "Logarithmus von e hoch 3"  
log(exp(3))  
  
# "Sinus von 2 Pi"  
sin(2 * pi)
```

```
## [1] 144  
## [1] 2  
## [1] 20.08554  
## [1] 3  
## [1] -2.449213e-16
```

# Grundlagen

## Wissenschaftliche Notation

**R** stellt besonders große bzw. besonders kleine Zahlen mit Hilfe der **wissenschaftlichen Notation** dar.

Die Zahl -2.449294e-16 aus dem **R**-Output ist

$$-2.449294 \times 10^{-16} = \frac{-2.449294}{10\,000\,000\,000\,000\,000} \approx 0.$$

- Auf den zweiten Blick ist die wissenschaftliche Notation also tatsächlich nutzerfreundlicher.

# Grundlagen

## Variablen

Variablen werden mit dem Zuweisungsoperator `<-` ("Kleinerzeichen" gefolgt von "Minuszeichen") erzeugt bzw. überschrieben.

```
x <- 13  
y <- 24
```

Mit den so erzeugten Variablen können wir natürlich auch rechnen.

```
x - y
```

```
## [1] -11
```

# Grundlagen

## Achtung – Häufiger Fehler:


Vergessen von `*`, wenn eine Variable mit einem Skalar multipliziert werden soll.

```
# Richtig:  
3 * x
```

```
## [1] 39
```

```
# Falsch:  
3x
```

### Problem:

 interpretiert `3x` als Variable, jedoch dürfen Variablennamen nicht mit Ziffern beginnen! Wir erhalten eine Fehlermeldung!



# Grundlagen

## Kommentare

Auf der vorherigen Folie steht im letzten Codesnippet eine kurze Erklärung angeführt von einem `#`. Dies kennzeichnet einen **Kommentar**:

*Kommentare werden beim Ausführen des Codes nicht berücksichtigt. Wir nutzen Kommentare, um unseren Code (für uns und andere) verständlicher zu machen.*

# Grundlagen

## Datentypen

Neben den Zahlen gibt es in  noch weitere Datentypen. Die für uns Wichtigsten (Zahlen eingeschlossen) sind:

```
a <- 5                # numeric (Zahlen)
class(a)
```

```
## [1] "numeric"
```

```
b <- "R ist toll!"    # character (Schriftzeichen)
class(b)
```

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

# Grundlagen

## Datentypen

```
d <- TRUE                # logical (TRUE / FALSE)
class(d)
```


```
## [1] "logical"
```

### Häufiger Fehler:

Vergessen von Anführungszeichen bei Schriftzeichen und Kleinschreiben von TRUE oder FALSE.

# Vektoren

## Erzeugen eines Vektors

Bisher haben wir nur Variablen betrachtet, die einen einzigen Wert enthalten. Die große Stärke von  sind vektor- und matrixbasierte Funktionen.

Ein Vektor wird gewöhnlich mit der Funktion `c()` (für **c**ombine) erzeugt und besteht aus einem oder mehreren Elementen **eines** Datentyps.

```
numeric_v  <- c(1, 4, 8)

character_v <- c("Pommes", "Falafel", "Ketchup")

logical_v   <- c(TRUE, FALSE, FALSE)

# oder kürzer `T` für `TRUE` und `F` für `FALSE`
logical_v   <- c(T, F, F)
```

**Achtung:** T und F sollten nicht als Variablen verwendet werden (warum?).

# Vektoren

## Erzeugen eines Vektors

Insbesondere für numerische Vektoren gibt es ein paar hilfreiche Shortcuts: Jede der folgenden Zeilen liefert denselben Output.

```
c(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
```

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

```
1:10
```

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

```
seq(from = 1, to = 10, by = 1)
```

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

# Vektoren

## *Missing Values*

Bei "echten" Daten kommt es häufig vor, dass es zu einigen Beobachtungen keine Daten für manche Variablen gibt.

### **Praxisbeispiel**

Angenommen in einer Umfrage werden die Variablen *Alter*, *Geschlecht* und *Körpergröße* erhoben.

Problem: Eine Person hat keine Angabe zur Körpergröße gemacht.

- In **R** kann der fehlende Wert speziell codiert werden: Anstelle von einer  $-99$  oder  $0$  (sieht man oft in echten Datensätzen) wird für den fehlenden Wert ein `NA` eingesetzt.
- Für viele Funktionen kann das Vorgehen bei *Missing Values* festgelegt werden (siehe auch *Übungsaufgabe 5*).

# Vektoren

## *Missing Values*

Beim Programmieren werden i.d.R. englische Namen verwendet. Meist sind diese kürzer und man vermeidet Probleme aufgrund von Umlauten.

```
age      <- c(25, 28, 29)
sex      <- c("m", "f", "m")
height  <- c(184, 165, NA)
```

# Vektoren

## Subsetting von Vektoren

Um eine Teilmenge der Elemente eines Vektors anzusprechen (*Subsetting* genannt), nutzen wir in eckigen Klammern eingeschlossene *Indizes*.

```
numeric_v[2]           # Gibt das 2. Element von numeric_v aus
```

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

```
character_v[c(1, 3)] # Gibt das 1. und 3. Element von character_v aus
```

```
## [1] "Pommes" "Ketchup"
```

```
logical_v[1:2]         # Gibt das 1. und 2. Element von logical_v aus
```

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



# Vektoren

## Subsetting von Vektoren

**Häufiger Fehler:** Vergessen von `c()` im 2. Fall. Mehrere Indizes müssen als *Vektor* übergeben werden.

Wir können auch logische Vektoren für das Subsetting nutzen. Diese müssen dieselbe Länge haben wie der Vektor der "gesubsetted" werden soll. Es werden die Elemente ausgegeben, an deren Stellen der logische Vektor den Wert `TRUE` hat.

```
x <- c(7, 2, 5, 3, 9)

# Elemente 2 und 5 sollen angesprochen werden
hv <- c(F, T, F, F, T)

x[hv]
```

```
## [1] 2 9
```

Das ist hilfreich, da viele Funktionen einen logischen Vektor zurückgeben (z. B. `is.na()` und logische Operatoren, später mehr dazu).

# Vektoren

## Ändern von Elementen

Manchmal müssen Elemente eines Vektors *geändert* werden. Dazu nutzen wir Subsetting zusammen mit dem Zuweisungsoperator.

**Beispiel:** Ändere das dritte Element des Vektors `x <- c(7, 2, 5, 3, 9)`

```
x <- c(7, 2, 5, 3, 9)
x[3] <- 49
```

Wenn wir mehrere Elemente ändern wollen:

```
x[c(1, 5)] <- c(25, 14)
x[c(2, 4)] <- -1      # Setzt das 2. und das 4. Element gleich -1
```

Wie sieht der erzeugte Vektor aus?

# Vektoren

## Vektorbasierte Funktionen

Wir betrachten nun den Vektor `age`. Dieser enthält das Alter von 5 Studierenden in diesem Kurs.

```
age <- c(24, 25, 29, 23, 26)
```

Wie können wir das Durchschnittsalter berechnen?

# Vektoren

## Vektorbasierte Funktionen

### Antwort:

```
age_sum <- age[1] + age[2] + age[3] + age[4] + age[5]  
age_sum / 5
```

```
## [1] 25.4
```

Dieses Vorgehen ist für kleine Vektoren machbar, aber umständlich. Für große Vektoren ist es sehr nervig und nicht mit zeitlichen Restriktionen vereinbar!

# Vektoren

## Vektorbasierte Funktionen

Bessere Alternative: *Vordefinierte Funktionen nutzen:*

```
# Mittelwert berechnen  
age_sum <- sum(age)  
age_sum / length(age)
```

```
## [1] 25.4
```


Oder noch einfacher:

```
mean(age)
```

```
## [1] 25.4
```

# Vektoren

## I Need Help

Damit haben wir schon einige -Funktionen kennengelernt. Oft ist unklar, wie diese Funktionen funktionieren.

⇒ -Hilfe nutzen:

Angenommen wir sind nicht sicher, was `mean()` genau berechnet oder wie die Funktion aufgerufen werden muss.

```
?mean
```

Die -Hilfe ist insb. für Anfänger manchmal schwer zu verstehen...

Nicht verzweifeln, sondern einfach ein bisschen rumprobieren oder eine Suchmaschine bemühen!

**Zwei extrem wichtige Fähigkeiten beim Programmieren, die gleich schon mal geübt werden können!**

# Vektoren

## Vektorbasierte Funktionen – Eine kleine Übersicht

```
x <- c(5, 2, 1055, 101:200)

length(x)      # Länge von x
sum(x)         # Summe der Elemente von x
mean(x)        # Arithmetisches Mittel der Elemente von x
min(x)         # kleinstes Element
max(x)         # größtes Element
which.min(x)   # Index des kleinsten Elements
which.max(x)   # Index des größten Elements
prod(x)        # Produkt aller Elemente
seq_along(x)   # Vektor mit Indizes der Elemente von x
```

# Vektoren

## Übungsaufgaben zu Vektoren

1. Erzeugen Sie einen Vektor `numbers` mit den Elementen `( 4 6 -3 2.5 18 pi 85 )`.

*Hinweis:* Die Zahl  $\pi$  ist in `R` bereits als `pi` vordefiniert.

2. Berechnen Sie das arithmetische und das harmonische Mittel von `numbers`.

*Hinweis:* Für einen numerischen Vektor  $X$  der Länge  $n$  ist das arithmetische Mittel  $\bar{X} = \frac{1}{n} \sum_{i=1}^n X_i$  und das harmonische Mittel  $\bar{X}_{harm} = \frac{n}{\sum_{i=1}^n 1/X_i}$ .

3. Sie kommen zu dem Schluss, dass die höchste und die niedrigste Zahl die Schätzung verzerren und entscheiden darum, diese Werte zu ignorieren. Ersetzen Sie beide Werte durch `NA` und berechnen Sie die Mittelwerte aus Aufgabe 2 erneut.



# Vektoren

## Übungsaufgaben zu Vektoren

4. Nutzen Sie die Funktion `seq()` um die Folge  $(0, 0.5, 1, 1.5, \dots, 99, 99.5, 100)$  zu erzeugen. Wie viele Elemente besitzt dieser Vektor? Überprüfen Sie Ihre Vermutung mit `length()`  
characters mit den Elementen  $(a \ a \ a \ b \ b \ b \ b \ c \ c)$ . Finden Sie dazu heraus wie die Funktion `rep()` funktioniert und nutzen Sie diese.
5. Überschreiben Sie jetzt den Vektor `characters` mit  $(x \ y \ z \ x \ y \ z \ x \ y \ z)$ . Nutzen Sie wieder die Funktion `rep()`.
6. Ersetzen Sie nun alle Elemente mit dem Inhalt  $z$  durch  $v$ .

# Vektoren

## Übungsaufgaben zu Vektoren

8. Kopieren Sie folgenden Code in Ihr R-Skript

```
a <- c(2, 5, 7, 5, 12, 6)
b <- c(1, 2, 3, 4, 5, 6)
x <- c(1:2)
y <- 3
z <- c(1, 2, 3, 4)
```

Berechnen Sie nun  $(a + b)$ ,  $(a + x)$ ,  $(a + y)$  und  $(a + z)$ . Finden Sie heraus, wie **R** jeweils vorgeht und schreiben Sie einen kurzen Kommentar.

9. Erzeugen Sie einen Vektor mit den Elementen  $(1 \ 2 \ 3 \ a \ b)$  (Also eine Mischung aus numeric und character). Was passiert? Schreiben Sie einen Kommentar.

# Matrizen

## Matrizen erzeugen

Wir können mehrere Vektoren des gleichen Datentyps und gleicher Länge zu einer Matrix zusammenfassen.

```
x <- 1:3
y <- 4:6

xy <- cbind(x, y)
xy
```

```
##      x y
## [1,] 1 4
## [2,] 2 5
## [3,] 3 6
```

# Matrizen

## Matrizen erzeugen

Außerdem können wir auch einen einzelnen Vektor in Matrixform bringen.

```
matrix(1:6, ncol = 2)
```

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

# Matrizen

## Matrixsubsetting

Um ein einzelnes Element einer Matrix anzusprechen, müssen wir **zwei** Indizes verwenden: einen für die *Zeile* und einen für die *Spalte* des gewünschten Elements.

Das folgende Beispiel zeigt, wie man das Element in der 3. Zeile und der 2. Spalte erhält.

```
X <- matrix(1:6, ncol = 2)
X[3, 2]
```

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

# Matrizen

## Matrixsubsetting

Um dieses Element zu ändern, nutzen wir wieder den Zuweisungsoperator `<-`.

```
X[3, 2] <- 13  
X
```

```
##      [,1] [,2]  
## [1,]    1    4  
## [2,]    2    5  
## [3,]    3   13
```

# Matrizen

## Matrixsubsetting

Um ganze Spalten einer Matrix zu erhalten, wird der Index für die *Zeile* freigelassen.

```
X[, 1] # gibt die 1. Spalte zurück
```

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

Um ganze Zeilen zu erhalten, wird der Index für die *Spalte* freigelassen.

```
X[2:3, ] # gibt die 2. und 3. Zeile zurück
```

```
##      [,1] [,2]  
## [1,]    2    5  
## [2,]    3   13
```

# Matrizen

## Matrixalgebra – Übersicht

```
A <- cbind(c(13, 4, 8), c(2, 8.2, 1))  
B <- cbind(c(9, 2.3, -1), c(12, 53, -3))  
  
A * B           # Elementweise Multiplikation  
  
C <- t(A) %*% B  # Matrixmultiplikation  
  
t(A)            # Transponieren  
  
diag(C)         # Elemente der Hauptdiagonale  
  
solve(C)        # Invertieren  
  
eigen(C)        # Eigenwerte und Eigenvektoren
```



# Matrizen

## Matrixbasierte Funktionen

Wie bei Vektoren gibt es auch eine Reihe von Funktionen für Matrizen.

```
X <- matrix(1:40, ncol = 4)
colSums(X)    # Spaltensummen
rowSums(X)    # Zeilenensummen
colMeans(X)   # Spaltenmittelwerte
rowMeans(X)   # Zeilenmittelwerte
```

# Matrizen

## Übungsaufgaben zu Matrizen

1. Erzeugen Sie mit dem Inputvektor `1:12` und `matrix()` folgende Matrix  $X$ .

$$X = \begin{pmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \\ 7 & 8 \\ 9 & 10 \\ 11 & 12 \end{pmatrix}$$

2. Nehmen Sie die Matrix aus 2.1 und vertauschen Sie die Spalten. Das Ergebnis soll an die Variable ``Y`` übergeben werden.
3. Berechnen Sie  $XY^T$ .

# Matrizen

## Übungsaufgaben zu Matrizen

4. Erzeugen Sie eine  $2 \times 2$  Matrix aus der 2. und 5. Zeile der Matrix `X`.
5. Erzeugen Sie die Matrix `X` mit `X <- matrix(8:-7, nrow = 4)`.
  - Ersetzen Sie die Elemente auf der Hauptdiagonalen durch `NA`s.
  - Ersetzen Sie jetzt alle `NA`s in der Matrix durch `1`. Nutzen Sie dazu die Funktion `is.na()`.