

Evaluation Randomisiert-Kontrollierter Studien und Experimente mit R

R Basics: Funktionen, Objekte, Operatoren und Fehlermeldungen

Prof. Dr. David Ebert & Mathias Harrer

Graduiertenseminar TUM-FGZ

Psychology & Digital Mental Health Care, Technische Universität München

Funktionen

Unabhängig von spezifischen Funktionen, können mithilfe von R klassische Rechenaufgaben gelöst werden:

Probieren wir es selbst:

```
(49*6)/7
```

```
1^3 + 5^3 + 3^3
```

Funktionen sind **Kernelemente von R**: Sie erlauben es, vordefinierte Operationen auszuführen. Es besteht eine Parallele zur **mathematischen Formulierung** einer Funktion $f(x)$; z.B. für die Quadratwurzel:

$$f(x) = \sqrt{x}$$

In R wird eine Funktion definiert, indem erst der **Name der Funktion** und dahinter in Klammern ihre **Inputs** (sog. **Argumente**) aufgeschrieben werden.

```
Funktionsname(Argument1 = Wert1, Argument2 = Wert2, ...)
```

In R wird so aus obiger Formel für die Quadratwurzel:

```
sqrt(x = 4)
```

Position Matching

Der Argumentname kann auch **weggelassen** werden, solange die **Reihenfolge** der Argumente eingehalten wird.

Beispiel: “`sqrt(x = 4)`” und “`sqrt(4)`” führen zum gleichen Ergebnis, da beides mal 4 als erstes Argument auftaucht.

1. Was ist die Quadratwurzel von 9? Dazu können die Funktionen `sqrt()` nutzen:

```
sqrt(9)
## [1] 3
```

2. Logarithmus `log()` aller Werte der Variable `age` im Datensatz:

```
log(data$age)
## [1] 3.850 3.737 3.951 3.135 3.828 ...
```

→ Statt eines konkreten Wertes wird die gesamte Variable in die Funktion eingespeist, indem die Variable `age` über das Dollarzeichen aus unserem Datensatz `data` ausgewählt wird.

3. Mittelwert mean() der Variable pss zum Post-Zeitunkt:

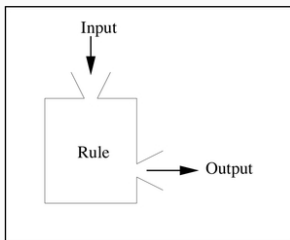
```
mean(data$pss.1)
## [1] NA
mean(data$pss.1, na.rm = TRUE)
[1] 20.423
```

In R kodiert NA, dass ein Wert fehlt.

Das Ergebnis der ersten Zeile Code ist **“not available”** (NA), da zum Post-Zeitpunkt **Beobachtungen fehlen** (d.h. NA sind) und der Mittelwert so nicht berechnet werden kann. Durch die Spezifikation des Arguments `na.rm` als `TRUE`, wird der Mittelwert nur über die **beobachteten Werte** gebildet und kann somit ausgegeben werden.

Funktionen als "Herzstück" von R

Auch deutlich komplexere Funktionen in R funktionieren nach dem gleichen Prinzip: Man gibt die Parameterinformationen ein, die eine Funktion benötigt, und die Funktion nutzt diese Information, um ihre Berechnungen durchzuführen und schließlich das Ergebnis anzuzeigen.



Die R Documentation

Viele Funktionen in R verlangen mehrere Argumente, und **niemand** kann die korrekte Nutzung aller Funktionen **auswendig lernen!**

- Die Lösung: Detaillierte Beschreibungen der Funktionen in der **R Documentation**.
- Die R Documentation kann entweder über **Help** im rechten unteren Fenster in Rstudio aufgerufen werden; oder direkt via Ausführen von `?funktionsname` in der Konsole; z.B. `?mean`.
- **Cave:** Die Dokumentation von Funktionen wird von den jeweiligen Package-Entwicklern selbst geschrieben. Sie ist daher nicht immer gleich informativ oder anfängerfreundlich.

Die R Documentation kann im Browser via rdocumentation.org oder rdrr.io eingesehen werden.

Default Arguments

- Unter "Default Arguments" werden Argumente einer Funktion verstanden, deren Wert **vordefiniert ist und automatisch genutzt wird.**
- Default Arguments müssen beim Schreiben der Funktion also nur hinzugefügt werden, wenn sie **explizit von den Voreinstellungen abweichen.**
- Default-Werte einer Funktion können im Abschnitt "Usage" in **R Documentation** eingesehen werden

Siehe z.B. den Documentation-Eintrag für `?mean`:

```
mean(x, trim = 0, na.rm = FALSE, ...)
```

Im Gegensatz zu “Object-Oriented Programming Languages” (z.B. Python, JS) konzentrieren sich **“Functional Programming Languages”** bei der Problemlösung auf **Funktionen**:

Hauptmerkmale von Functional Programming Languages

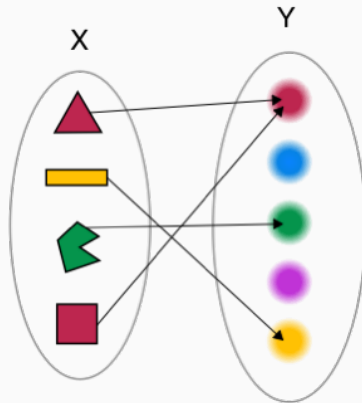
- **First-Class Functions:** Vielseitige Einsetzbarkeit von Funktionen.
- **Pure Functions:** Output der Funktion hängt vom Input ab (d.h. Output reproduzierbar) und keine Nebeneffekte der Funktion (wie z.B. Veränderung des Wertes einer globalen Variable).

Auch wenn R diesen Kriterien nicht vollkommen entspricht, kann **R im Kern als Functional Programming Language definiert** werden.

Wickham (2019)

Objekte

Objekte können als **Gegenspieler** von Funktionen verstanden werden: wir verwenden Funktionen, um Operationen an Objekten durchzuführen!



Um Objekte in R nutzen zu können, müssen wir diesen einen **Variablennamen** zuweisen. Dies ist möglich durch den **Zuweisungsoperator** `<-` (*assignment operator*).

```
geschlecht <- "Weiblich"
```

Eine Variablenname kann auch so zugewiesen werden:

```
"Weiblich" -> geschlecht  
geschlecht = "Weiblich"  
assign("geschlecht", "Weiblich")
```

Zur Inspektion des Objekts kann der Name des Objekts eingegeben werden:

```
geschlecht  
## [1] "Weiblich"
```

- Sobald Objekte einem Variablennamen zugewiesen worden sind, werden diese in RStudio im **Environment** rechts oben angezeigt.
- Dies bedeutet, dass das Objekt (temporär) in unserer Programmierungsumgebung **gespeichert** ist, und für weitere Operationen zur Verfügung steht.
- Existierende Objekte werden **überschrieben**, nicht vorhandene neu erzeugt.
- Mit der `rm` Funktion lassen sich Objekte aus dem Environment löschen, z.B. `rm(geschlecht)`.

Cave: Benennung von Objekten

- Objektnamen müssen mit einem Buchstaben beginnen und können nur Buchstaben, Zahlen, Unterstriche und Punkte beinhalten.
- Konsistenz ist immer von Vorteil: z.B. immer `namen.mit.punkten.trennen` oder `camelCase` verwenden.

(Wickham & Grolemund, 2016, Kap. 4.2)

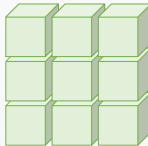
vector (numeric, character, factor, logical, ...)

$$\mathbf{a} = (a_1, a_2, a_3, \dots)$$

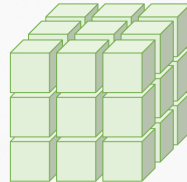


matrix (m rows, n columns)

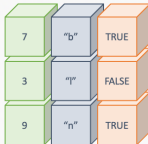
$$\mathbf{A} = \begin{bmatrix} a_{11} & \cdots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{m1} & \cdots & a_{mn} \end{bmatrix}$$



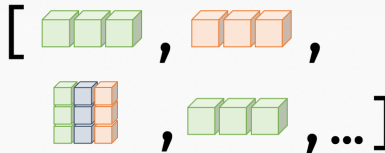
array (m columns, n layers, l rows)



data.frame (m rows, n columns)



list (n elements)



Vektoren

Vektoren (*vectors*) sind eine Sammlung von Werten (z.B. Zahlen, Wörter, Faktorstufen). Besteht ein Vektor nur aus einer Zahl, spricht man von einem **Skalar**.

Ein Vektor kann über die **concatenate**-Funktion `c` gebildet werden:

```
vector <- c(6, 9, 12, 18)
vector
```

```
#> [1]  6  9 12 18
```

Vektoren kommen in unterschiedlichen “Geschmacksrichtungen”:

- **numeric** oder **double**: in Zahlen gespeicherte Daten (z.B. Alter).
- **character**: in “Worten”/Buchstaben gespeicherte Daten.
- **logical**: binäre Variablen, die anzeigen, ob eine Bedingung TRUE oder FALSE ist.
- **factor**: in Zahlen gespeicherte Daten, wobei jede Zahl ein anderes Level einer Variable anzeigt (z.B. 1 = “wenig,” 2 = “mittel,” 3 = “hoch”).

Die Klasse eines Vektors kann mit der `class`-Funktion überprüft werden.

Cave: Vektorklassen und weiterführende Analysen

Die Klasse eines Vektors hat Implikationen auf weitere Analyseschritte. Für `characters` kann z.B. kein Mittelwert berechnet werden.

Alle Variablen des Datensatzes: Funktion `glimpse` aus dem package `{tidyverse}`.

```
library(tidyverse)
glimpse(data)
## Rows: 264
## Columns: 34
## $ id          <chr> "stress_gui_002", "stress_gui_140", ~
## $ group       <dbl> 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, ~
## $ sex         <dbl> 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, ~
## ...
```

Einzelne Variablen: Anwendung der `class`-Funktion.

```
class(data$id)
## [1] "character"
```

Dataframes

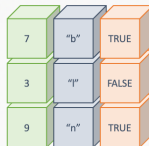
Dataframes (`data.frame`) sind die geläufigste **Struktur zur Sammlung von Daten** in R. Sie funktionieren wie einfache **Tabellen**: für jeden Zeileneintrag m gibt es Werte für n verschiedene Variablen.

Dataframes können aus **Vektoren zusammengestellt** werden. Im Gegensatz zur `matrix` können dabei unterschiedliche Vektorklassen (numeric, logical, character, ...) gebündelt werden.

```
name <- c("Lea", "Antonia", "Paula")
alter <- c(27, 22, NA)
weiblich <- c(TRUE, TRUE, TRUE)
data.frame(name, alter, weiblich)

##      name alter weiblich
## 1    Lea    27     TRUE
## 2 Antonia   22     TRUE
## 3  Paula   NA     TRUE
```

data.frame (m rows, n columns)



7	"b"	TRUE
3	"j"	FALSE
9	"n"	TRUE

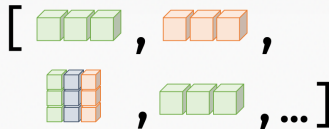
Listen

Listen (lists) sind die **flexibelste** Datenstruktur in R. Sie erlauben es, jegliche Art von Objekt in einem “übergeordneten” Objekt zu sammeln (z.B. Dataframes, Vektoren, Arrays, Matrizen, einfache Werte, ...).

```
df <- data.frame(name, alter, weiblich) # siehe Folie zu data.frames
df.beschreibung <- "Tabelle SHKs 2021"
universitäten <- c("TUM", "FAU")
list(df, df.beschreibung, universitäten)
```

```
## [[1]]
##      name alter weiblich
## 1     Lea   27     TRUE
## 2 Antonia  22     TRUE
## 3  Paula   NA     TRUE
## [[2]]
## [1] "Tabelle SHKs 2021"
## [[3]]
## [1] "TUM" "FAU"
```

list (n elements)



Operatoren

✓ Einige Operatoren haben wir bereits kennengelernt:

- **Grundrechenarten:** +, -, *, /
- **Potenz:** ^2, ^3, ^4, ...
- **Zuweisungsoperator:** <-, ->, =
- **“Pull”-Operator:** \$

→ Weitere Operatoren:

- **Vergleichsoperatoren:** >, >=, <, <=, != (nicht gleich), == (gleich)
- **Boole'sche Operatoren:** & (und), | (oder), ! (nicht)
- **Pipe-Operator:** %>%

(Wickham & Grolemund, 2016, Kap. 5.2)

Vergleichs- und Boole'sche Operatoren sind nützlich, um zu bestimmen, ob bestimmte Vektorelemente eine **Bedingung** erfüllen oder nicht.

```
"Variable" == "variable"  
# [1] FALSE
```

```
x <- 10  
y <- 20  
x > 5 & y != 10  
## [1] TRUE
```

```
data$cesd.0 > 16  
## [1] TRUE TRUE TRUE TRUE TRUE TRUE ...
```


Der **Pipe-Operator** `%>%` ist als einziger Operator nicht Teil von Base R¹. Er ist erst verfügbar, sobald das `{tidyverse}` Package geladen wurde. Pipes haben **zwei große Vorteile**:

- Funktionen können auf ein Objekt angewandt werden, ohne dass das Objekt in der Funktion jeweils nochmal benannt werden muss.
- Mit Pipes können mehrere Funktionen **aneinandergeschaltet** werden.

```
library(tidyverse)
data %>% pull(pss.0) %>% mean() %>% sqrt()
## [1] 5.051627
```

Die pull-Funktion

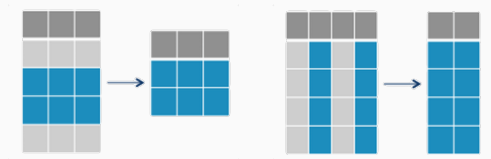
Die `pull`-Funktion ist das Äquivalent zum `$`-Operator innerhalb von Pipes. Die Funktion "zieht" eine Variable aus dem Datensatz und gibt sie weiter an die nächste Funktion.

¹Mit R Version 4.0.0 wurde nun auch ein Base-R Pipe-Operator eingeführt. Dieser benutzt jedoch `|>` als Symbol.

Indexing & Slicing

Es gibt mehrere Wege, um in R Daten aus einem Dataframe zu extrahieren:

1. Mithilfe des **\$-Operators** oder `pull` (*schon besprochen*).
2. Über **eckige Klammern** `[,]`.
3. Über die Funktion `filter` bzw. `select` aus dem *{tidyverse}*.



Slicing von Dataframes mit Eckigen Klammern

Subsetting von Dataframes mit eckigen Klammern ist etwas komplexer, erlaubt aber auch größere Flexibilität. Die generelle Form folgt der mathematischen Notation von Matrizen:

$$A[2,1] = \mathbf{A}_{2,1} = \begin{array}{c} \text{Variablen} \\ \overbrace{\begin{bmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ \textcolor{brown}{a}_{21} & a_{22} & \dots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \dots & a_{mn} \end{bmatrix}} \end{array}$$

Die allgemeine Form zum Slicing ist also `data.frame[row, column]`.

Slicing von Dataframes mit Eckigen Klammern

Um ein Subset der Daten auszuwählen, brauchen wir einen **Index**. Typischerweise ist dies eine **Zahl**, die die Zeilen- bzw. Spaltennummer(n) angibt.

```
data[3,15]  
## [1] 26
```

Es können auch mehrere Zeilen/Spalten ausgewählt werden:

```
data[1:3,c(15,17)]  
##      cesd.0 cesd.2  
## 1        18      16  
## 2        22      23  
## 3        26      27
```

Slicing von Dataframes mit Eckigen Klammern

Wird ein Slot frei gelassen, wird die gesamte Zeile/Spalte ausgewählt:

```
data[,2]
##      group
## 1         0
## 2         0
## 3         0
```

Eine Indizierung ist auch mit dem **Variablennamen** möglich:

```
data[1,"pss.0"]
## [1] 25
```

Slicing von Dataframes mit Eckigen Klammern

Besonders hilfreich ist der Einsatz von `logicals` durch Vergleichsoperatoren. So kann z.B. der PSS-Wert aller Personen gefilter werden, die älter als 40 sind:

```
data[data$age > 40, "pss.0"]  
##      pss.0  
## 1      25  
## 2      22  
## 3      25  
## [...]
```

Dies funktioniert, da der Boole'sche Ausdruck als Index fungiert:

```
data$age > 40  
## [1]  TRUE  TRUE  TRUE FALSE  TRUE ...
```

Slicing von Dataframes mit filter und select

Die `filter` und `select`-Funktionen sind Teil des *{tidyverse}*. Sie erleichtern das Filtern und Selektieren von Dataframes, und sind besonders “Pipe-freundlich.”

```
data %>%  
  filter(age > 40, sex == 0) %>%  
  select(pss.0, pss.1, pss.2) %>%  
  head(3)
```

```
#>   pss.0 pss.1 pss.2  
#> 1    25    15    21  
#> 2    22    18    24  
#> 3    25    NA    22
```

Die `head`-Funktion wird genutzt, um nur die ersten 3 Zeilen auszugeben.

Fehlermeldungen

Fehlerarten

- **Errors:** Eine Funktion kann nicht ausgeführt und muss gestoppt werden.
- **Warnings:** Es ist ein Fehler aufgetreten, aber die Funktion kann trotzdem (teilweise) ausgeführt werden.
- **Messages:** Information, dass eine Aktion für den Benutzer/ die Benutzerin ausgeführt wurde.

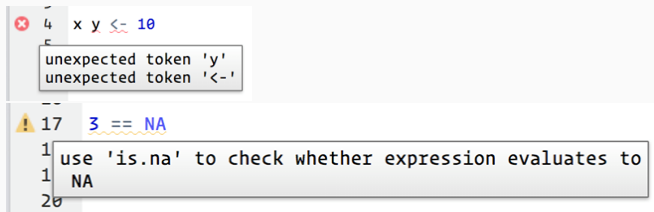
Kein Grund zur Panik

- Fehlermeldungen in sind am Anfang meist sehr verwirrend, aber "normal".
- Im Laufe der Zeit werden Fehlermeldungen immer informativer und leichter zu entziffern.
- Googeln der Fehlermeldung ist hilfreich. Dazu sollte die Ausagesprache aber zuvor auf Englisch gestellt werden: `sys.setenv(LANG = "en")`.

(Wickham & Grolemund, 2016, Kap. 6.2)


Im Skript in RStudio werden typische Fehler automatisch markiert

Syntaxfehler werden mit einem roten Kreuz und potentielle Probleme mit gelbem Ausrufezeichen am linken Rand markiert:



(Wickham & Grolemund, 2016, Kap. 6.2)

Praxis-Teil



```
128 }
129
130 }
131
132 .mail{
133     background: url(../img/mailico.png) no-repeat center;
134     display: inline-block;
135     width: 120px;
136     height: 140px;
137     float: left;
138     margin: 2px 7px 0 0;
139 }
140 .phone{
141     background: url(../img/phoneico.png) no-repeat center;
142     display: inline-block;
143     width: 20px;
144     height: 130px;
145     float: left;
146     margin: 2px 7px 0 0;
147 }
```


Fragen & Antworten: protectlab.org/workshop/rct-evaluation-in-r/r-entdecken/slicing/#uebung

1. Log-transformiere die Variable `age` in `data` und speichere das Ergebnis unter dem Namen `age.log`.
2. Quadriere die Werte in `pss.1` und speichere das Ergebnis unter dem Namen `pss.1.squared`.
3. Berechne den Mittelwert und die Standardabweichung (SD) der Variable `cesd.2`. Nutze bei Bedarf das Internet um herauszufinden, welche Funktion in R die Standardabweichung berechnet.
4. Packe den Mittelwert und die Standardabweichung von `cesd.2` in eine Liste.
5. Hat die Variable `mbi.0` die passende Objektklasse `numeric`? Überprüfe dies mit R Code.
6. Lege im Dataframe `data` zwei neue Variablen an: (1) `age.50plus`, eine logical-Variable die mit `TRUE` und `FALSE` angibt, ob das Alter `age` einer Person ≥ 50 ist; (2) `pss.diff`, eine Variable die den Unterschied zwischen `pss.0` und `pss.1` für jede Person angibt.
7. Ändere den Wert von `ft.helps` in der dritten und vierten Zeile zu `NA`.
8. Mit der `order` Funktion kann für Variablen ein Index gebildet werden. Dieser Index zeigt an, in welcher Reihenfolge die Elemente korrekt geordnet wären. Nutze die R Documentation (`?order`), um mehr über die Funktion zu erfahren. Nutze dann diese Funktion in einer eckigen Klammer, um `data` dem Alter `age` nach zu ordnen!

Fragen?

Anmerkungen?

Kommentare?



```
128 }
129
130 }
131
132 .mail{
133     background: url(../img/mailico.png) no-repeat center;
134     display: inline-block;
135     width: 120px;
136     height: 140px;
137     float: left;
138     margin: 2px 7px 0 0;
139 }
140
141 .phone{
142     background: url(../img/phoneico.png) no-repeat center;
143     display: inline-block;
144     width: 20px;
145     height: 18px;
146     float: left;
147     margin: 2px 7px 0 0;
148 }
```

Referenzen

Wickham, H. (2019). *Advanced r*. chapman; hall/CRC.

Wickham, H., & Grolemund, G. (2016). *R for data science: Import, tidy, transform, visualize, and model data*. " O'Reilly Media, Inc."