

# V01 – Crashkurs

12. April 2021

## Contents

<b>1 R im Terminal</b>	<b>2</b>
<b>2 Einfache Operationen</b>	<b>2</b>
2.1 Rechnen . . . . .	2
2.2 Logik . . . . .	3
2.3 Hilfe . . . . .	4
2.4 Operatorpriorität . . . . .	4
<b>3 Variablen</b>	<b>4</b>
<b>4 Skripte</b>	<b>5</b>
<b>5 RStudio</b>	<b>5</b>
<b>6 Atomare Vektoren</b>	<b>6</b>
6.1 Indizierung . . . . .	7
6.2 Skalare . . . . .	8
6.3 Copy on Modify . . . . .	8
6.4 Fehlende Werte . . . . .	8
6.5 Plots . . . . .	9
6.6 Simulation von Zufallsexperimenten . . . . .	10
6.7 Typen . . . . .	10
6.8 Text . . . . .	11
<b>7 Funktionen definieren</b>	<b>12</b>
<b>8 Listen</b>	<b>13</b>
<b>9 Kontrollstrukturen</b>	<b>16</b>
9.1 Bedingte Anweisungen . . . . .	16
9.2 while-Schleife . . . . .	17
9.3 for-Schleife . . . . .	17
<b>10 Zufall</b>	<b>18</b>
<b>11 Summary Statistics</b>	<b>20</b>
<b>12 Matrizen</b>	<b>21</b>
<b>13 Plots</b>	<b>23</b>
<b>14 Pakete</b>	<b>26</b>
<b>15 Tibbles</b>	<b>27</b>
15.1 Daten einlesen . . . . .	27

16 apply-Funktionen	28
17 Statistik	29
17.1 Lineare Regression . . . . .	30

## Ziele der Lektion

Dieser Crashkurs soll eine Einführung in den Umgang mit R darstellen und eine grobe Übersicht über die Fähigkeiten von R geben. Dabei werden die meiste Themen nur oberflächlich behandelt und erst in späteren Lektionen im Detail ausgeführt.

# 1 R im Terminal

R ist eine freie (*open source*) Programmiersprache für statistische Berechnungen und Grafiken.

Außerdem ist R das Programm, welches diese Programmiersprache ausführt. Die aktuelle Version von R kann unter <https://www.r-project.org> heruntergeladen werden.

Wir beginnen mit der Nutzung von R im Terminal und gehen später zu einer grafischen Benutzeroberfläche über.

Starten wir das Programm R, werden einige allgemeine Informationen angezeigt – unter anderem die Versionsnummer.

```
R version 3.6.3 (2020-02-29) -- "Holding the Windsock"
Copyright (C) 2020 The R Foundation for Statistical Computing
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.
```

>

Für diese Lektion gehen wir von einer Version  $\geq 3.6$  aus.

Die Ausgabe endet mit der **Eingabeaufforderung** (*command prompt*) >. Dort können wir nun R-Befehle eingeben und ausführen (**Enter-Taste**).

# 2 Einfache Operationen

## 2.1 Rechnen

Wir können in der Eingabeaufforderung > verschiedene mathematische Terme eingeben, die R für uns auswertet.

Im Folgenden sind Eingaben in die Eingabeaufforderung zu sehen. Darunter steht mit **##** markiert die Ausgabe.

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

```
3*4
## [1] 12
5^6
## [1] 15625
(-3.5 + 7) * 2 + 0.5 * 16^0.5
## [1] 9
cos(pi)
## [1] -1
```

Hier sind einige mathematische Funktionen die zum Grundwortschatz von R gehören:

- Grundrechenarten: `+`, `-`, `*`, `/`,
- Potenz `^`, Wurzel `sqrt()`
- Division mit Rest: `%%` (Rest, “modulo”), `%/%` (ohne Rest)
- trigonometrische Funktionen `sin()`, `cos()`, `tan()`; Arkusfunktionen `asin()`, `acos()`, `atan()`; analog `sinh()`, `asinh()`, ... für Hyperbelfunktionen
- `exp()`, `log()` (Basis  $e$ ), `log10()` (Basis 10), `log2()` (Basis 2)
- Betrag `abs()`, Vorzeichen `sign()`
- Aufrunden `ceiling()`, Abrunden `floor()`, Nachkommastellen abschneiden `trunc()`, kaufmännisches Runden `round()`

Außerdem ist die Konstante `pi` ( $\pi$ ) implementiert.

Reelle Zahlen können mit der sogenannten **E-Notation** eingegeben werden:  $7.6e-5$  ( $7,6 \cdot 10^{-5}$ ),  $-1.23e4$  ( $-1,23 \cdot 10^4$ ). Je nach Größe des Wertes entscheidet sich R auch bei der Ausgabe für dieses Format.

```
1e2
## [1] 100
1e20
## [1] 1e+20
```

Da ein Computer nur eine endliche Teilmenge der reellen Zahlen darstellen kann (**Maschinengenauigkeit**), kann es zu Rechenfehlern kommen.

```
sin(pi)
## [1] 1.224606e-16
```

Reelle Zahlen haben den Typ `double`. Der Typ eines Objektes wird mit der Funktion `typeof()` abgefragt.

```
typeof(1)
## [1] "double"
typeof(pi)
## [1] "double"
```

Zum Typ `double` gehören auch `Inf` (unendlich), `-Inf` und `NaN` (*not a number*).

```
1/0
## [1] Inf
-1/0
## [1] -Inf
0/0
## [1] NaN
```

## 2.2 Logik

R wertet Vergleiche und aussagenlogische Formeln aus.

```
1 > 2
## [1] FALSE
```

```
2 <= 2
## [1] TRUE
(! (4 > 5) & (2+1 == 1+2)) | (0 > 1)
## [1] TRUE
```

R kennt die Wahrheitswerte TRUE, FALSE und NA (*not available*, dazu später mehr). Wahrheitswerte sind vom Typ **logical**.

```
typeof(TRUE)
## [1] "logical"
```

Operatoren auf dem Typ **logical** sind: | (oder), & (und), ! (nicht). Die Operatoren <, >, <=, >=, != (ungleich) == (gleich) ergeben einen Wahrheitswert durch den Vergleich von Zahlen.

Wegen beschränkter Maschinengenauigkeit ist bei Gleichheitstests für **double**-Werte besondere Vorsicht geboten.

```
sin(pi) == 0 # ergibt nicht das, was man vielleicht erwartet
## [1] FALSE
abs(sin(pi)) < 1e-14 # besser
## [1] TRUE
```

## 2.3 Hilfe

Mit dem **Hilfsoperator** ? rufen wir die R-Dokumentation auf. Für mehr Infos zu mathematischen Funktionen siehe zB ?Arithmetic, ?sin, ?log, ?sinh, ?factorial, ?sqrt.

## 2.4 Operatorpriorität

Die **Operatorpriorität** (*operator precedence*) gibt an, in welcher Reihenfolge Operatoren ausgewertet werden. Zum Beispiel gilt “Punkt vor Strich” und “Arithmetik vor Logik”.

```
5 + 4 * 3^2
## [1] 41
1 + 3 < 2 * 4
## [1] TRUE
```

Siehe ?Syntax für eine Liste der Operatoren geordnet nach Prioritäten.

## 3 Variablen

Der Zuweisungsoperator <- verknüpft **Werte** mit **Namen** zu **Variablen**: name <- value. Mit den so definierten Variablen können wir rechnen.

```
variable_name <- 5
asdf <- 7
variable_name
## [1] 5
asdf
## [1] 7
variable_name * asdf
## [1] 35
asdf <- asdf + 1
asdf
## [1] 8
```

Um den Wert einer Variablen zu erhalten, geben wir ihren Namen nach der Eingabeaufforderung ein.

Variablen haben **Typen**. Diese werden automatisch zugewiesen (**implizite Typisierung**, *implicit typing*). Im Gegensatz dazu muss in einer Programmiersprache mit expliziter Typisierung (*explicit typing*) wie zB C der Typ beim Erstellen der Variable angeben werden (R: `x <- 1.23`, C: `double x = 1.23;`).

Den **Typ** einer Variable finden wir mit der Funktion `typeof()` heraus:

```
b <- TRUE
typeof(b)
## [1] "logical"
i <- 1
typeof(i)
## [1] "double"
x <- 3.14
typeof(x)
## [1] "double"
word <- "hallo"
typeof(word)
## [1] "character"
```

Gültige **Namen** bestehen aus Buchstaben, Zahlen, ., oder \_ und beginnen mit einem Buchstaben oder einem Punkt nicht gefolgt von einer Zahl. .8sam, 8sam sind keine gültigen Namen, . .8sam, .\_. schon. Außerdem dürfen **reservierte Wörter** nicht als Variablennamen verwendet werden. Die reservierten Wörter sind:

```
if else repeat while function for in next break
TRUE FALSE NULL Inf NaN NA NA_integer_ NA_real_ NA_complex_ NA_character_
...
```

sowie .1, .2, .3, usw.

## 4 Skripte

Um eine Folge von Befehlen zu speichern, können diese in eine Textdatei geschrieben werden. Die Textdatei bekommt die Endung `.R`, um anzugeben, dass es sich um R-Code handelt. Wir bezeichnen diese Dateien als (R-)Skripte. Das Konsolenprogramm `Rscript` führt Skripte aus. In R wird mittels `source("Pfad/zum/Skript.R")` ein Skript ausgeführt.

Alle Zeichen nach und inklusive `#` sind **Kommentare** und werden bei der Ausführung ignoriert.

Befehle werden durch Zeilenumbrüche getrennt. Um mehrere Befehle in eine Zeile zu schreiben, müssen sie durch ein Semikolon ; getrennt werden. Dadurch wird der Code jedoch unübersichtlich.

R wird (typischerweise) nicht *kompiliert* sondern *interpretiert*. Das heißt, wir erhalten keine ausführbare Datei (wie etwa bei C) sondern verwenden immer unseren Code (in Form einer Textdatei) und die Programme R oder `Rscript`, um den Code auszuführen.

## 5 RStudio

Für die meisten Anwender ist es komfortabler eine **integrierte Entwicklungsumgebung** (*integrated development environment*, IDE) zu verwenden als Terminal und Texteditor. Die verbreitetste IDE für R ist **RStudio** (<https://www.rstudio.com/>). Wie R selbst ist auch RStudio *open source*.

RStudio greift auf die lokale Instanz von R zu, um Code auszuführen. R muss also für die Nutzung von RStudio installiert sein.

Es ist hilfreich einige **Short-Cuts** von RStudio zu lernen.

- **Windows** (Mac falls abweichend): Beschreibung
- **Alt+- (Option+-)**: füge `<-` ein

- **Ctrl+Z (Cmd+Z), Ctrl+Shift+Z (Cmd+Shift+Z):** undo bzw redo
- **Ctrl+1, Ctrl+2:** fokussiere Texteditor / Konsole
- **Ctrl+S (Cmd+S):** Datei speichern
- **Ctrl+F (Cmd+F):** Suchen und Ersetzen
- **Ctrl+Shift+Enter (Cmd+Shift+Return):** `source()` aktuelle Datei
- **Ctrl+Enter (Cmd+Return):** führe aktuelle Zeile (oder markierten Code) aus
- **Alt+Shift+K (Option+Shift+K):** Übersicht über Short-Cuts anzeigen
- **Tab:** Auto-Vervollständigung
- **Up-Arrow:** (nur in der Konsole) zuletzt ausgeführte Befehle aufrufen

Unter `Tools → Global Options → Appearance` kann ein Stil für das Syntax-Highlighting ausgewählt werden. Dieser sorgt dafür, dass in R-Skripten zB Strings in einer anderen Farbe als Zahlen dargestellt werden. Für differenzierteres Syntax-Highlighting sollte `Tools → Global Options → Code → Display → Highlight R function calls` aktiviert sein.

## 6 Atomare Vektoren

Daten können in R in unterschiedlichen *Datenstrukturen* gespeichert werden. Die einfachste Datenstruktur ist der **atomare Vektor**. Mit der Funktion `c()` (*combine*) können wir mehrere Werte (oder atomare Vektoren) gleichen Typs zu einem neuen atomaren Vektor zusammensetzen.

```
x <- c(1, 3, 5)
x
## [1] 1 3 5
y <- c(0, x, -42, x)
y
## [1] 0 1 3 5 -42 1 3 5
```

Die Anzahl der Elemente eines atomaren Vektors erhalten wir mit der Funktion `length()`.

```
length(x)
## [1] 3
length(y)
## [1] 8
```

Der Operator `:` erzeugt einen atomaren Vektor von auf- oder absteigenden Zahlen mit Differenz 1.

```
1:5
## [1] 1 2 3 4 5
3:-3
## [1] 3 2 1 0 -1 -2 -3
0.5:3.9
## [1] 0.5 1.5 2.5 3.5
```

Etwas mehr Kontrolle über die Sequenz der Zahlen haben wir mit der Funktion `seq()`.

```
seq(0, 3, by=0.5)
## [1] 0.0 0.5 1.0 1.5 2.0 2.5 3.0
seq(0, 3, length.out=5)
## [1] 0.00 0.75 1.50 2.25 3.00
```

Die Funktion `rep()` wiederholt einen Vektor.

```
rep(c(0,2), times=3)
## [1] 0 2 0 2 0 2
rep(c(0,2), each=3)
## [1] 0 0 0 2 2 2
```

Siehe dazu auch die entsprechenden Seiten der Dokumentation mittels `?`:`, `?seq`, `?rep`.

Die meisten Funktionen und Operatoren in R sind *vektorisiert*, dh, sie geben sinnvolle Ausgaben, wenn die Argumente nicht einzelne Werte sondern atomare Vektoren sind.

```
1:3 + 4:6
## [1] 5 7 9
2 * 1:3
## [1] 2 4 6
c(0,2,4) * 1:3
## [1] 0 4 12
sin(seq(0, 2*pi, length.out=6))
## [1] 0.000000e+00 9.510565e-01 5.877853e-01 -5.877853e-01 -9.510565e-01
## [6] -2.449213e-16
sum(1:99) # Summe aller Elemente des atomaren Vektors 1:99
## [1] 4950
x <- 1:5 < 5:1
x
## [1] TRUE TRUE FALSE FALSE FALSE
any(x) # Enthält das Argument x mindestens einmal TRUE?
## [1] TRUE
all(x) # Sind alle Elemente von x TRUE?
## [1] FALSE
```

## 6.1 Indizierung

Mit `x[i]` greifen wir auf das `i`-te Element des atomaren Vektors `x` zu. Wir können mit `[` auch mehrere Elemente auswählen.

```
x <- 11:15
x[2]
## [1] 12
x[c(1,4)]
## [1] 11 14
```

**Achtung:** Der erste Eintrag hat den Index 1 (nicht 0 wie in C).

Eine weitere Möglichkeit, Einträge eines Vektors auszuwählen, ist ein atomarer Vektor von Wahrheitswerten.

```
x[c(T, F, T, F, T)]
## [1] 11 13 15
```

Hierbei sind `T` und `F` Abkürzungen für `TRUE` und `FALSE`.

Dies ist besonders hilfreich in Zusammenhang mit vektorisierten Funktionen und Operatoren, deren Rückgabewert vom Typ `logical` ist.

```
x
## [1] 11 12 13 14 15
x %% 2
## [1] 1 0 1 0 1
x %% 2 == 0
## [1] FALSE TRUE FALSE TRUE FALSE
x[x %% 2 == 0] # wähle alle geraden Einträge
## [1] 12 14
```

Wir können beim Indizieren auch direkt neue Werte zuweisen.

```
x[c(1,5)] <- c(-1, -2)
x
## [1] -1 12 13 14 -2
x[2:4] <- 0
x
## [1] -1 0 0 0 -2
```

## 6.2 Skalare

R kennt keine besonderen Typen oder Datenstrukturen für Skalare. Skalare sind einfach atomare Vektoren mit Länge 1.

```
length(0)
## [1] 1
y <- 5
y[1]
## [1] 5
```

## 6.3 Copy on Modify

R setzt eine sogenannte **Copy-on-Modify**-Semantik um, dh eine Änderung einer Variablen führt zu einer Kopie des Wertes. Insbesondere gibt es keine Referenzen oder Pointer wie in C/C++.

```
x <- 1:5
y <- x # Kopie
y[3] <- -10
y # wurde geändert
## [1] 1 2 -10 4 5
x # bleibt unverändert
## [1] 1 2 3 4 5
```

## 6.4 Fehlende Werte

Mit `NA` (*not available*) wird ein fehlender Wert markiert.

```
x <- 1:5
x[3:7]
## [1] 3 4 5 NA NA
```

Sehr ähnlich zu `NA` ist der Wert `NaN`, der bei undefinierten Berechnungen zurückgegeben wird.

```
0/0
## [1] NaN
log(-1)
## Warning in log(-1): NaNs produced
## [1] NaN
```

**Achtung:** Tendenziell liefern Funktionen in R eher Werte wie `NA` oder `NaN` anstatt eine explizite Fehlermeldung auszugeben. Dies kann es manchmal erschweren Fehler im Code zu finden.

Vorsicht ist geboten bei mathematischen und logischen Operationen mit `NA` und `NaN`!

```
0 + NaN
## [1] NaN
NA | FALSE
## [1] NA
1:3 < c(0, NaN, 5)
```

```
## [1] FALSE    NA   TRUE
NA != 1
## [1] NA
NA == NA
## [1] NA
```

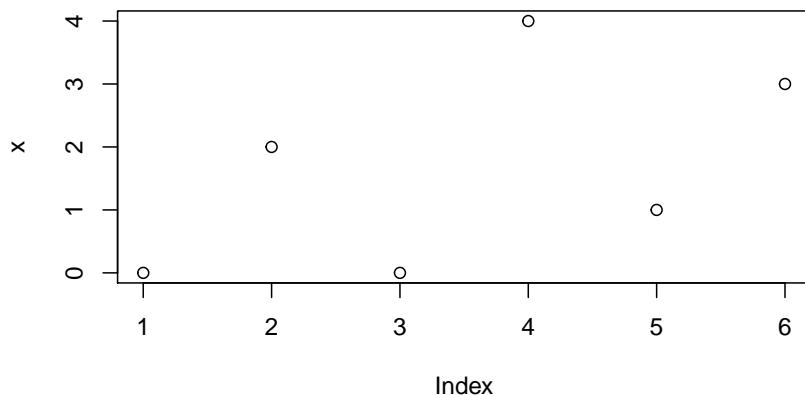
Um zu testen, ob eine Variable den Wert NA hat, kann == also nicht genutzt werden. Stattdessen verwenden wir die Funktion `is.na()`.

```
c(is.na(NA), is.na(NaN), is.na(0))
## [1] TRUE TRUE FALSE
# is.na() ist vektorisiert:
x <- c(1, NA, 2, NaN, 3)
x
## [1] 1 NA 2 NaN 3
is.na(x)
## [1] FALSE TRUE FALSE TRUE FALSE
x[!is.na(x)] # x ohne NA-Werte
## [1] 1 2 3
x[is.na(x)] <- 0
x
## [1] 1 0 2 0 3
```

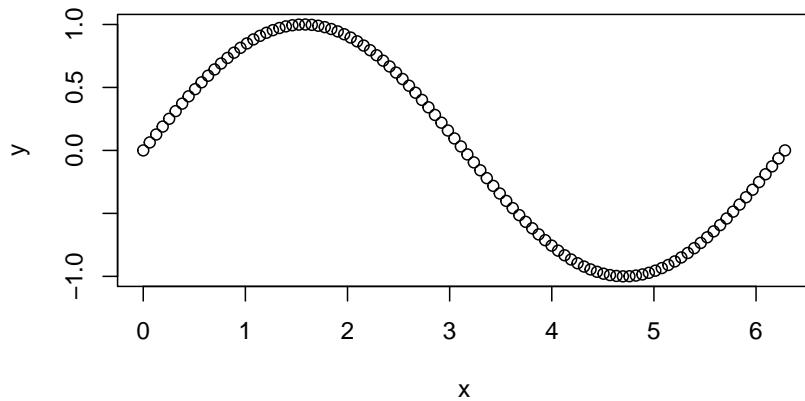
## 6.5 Plots

Mit der Funktion `plot()` werden die Werte eines numerischen atomaren Vektors grafisch dargestellt.

```
x <- c(0,2,0,4,1,3)
plot(x)
```



```
# Funktion plotten, Funktionsgraph zeichnen:
x <- seq(0, 2*pi, length.out = 100)
y <- sin(x)
plot(x, y)
```

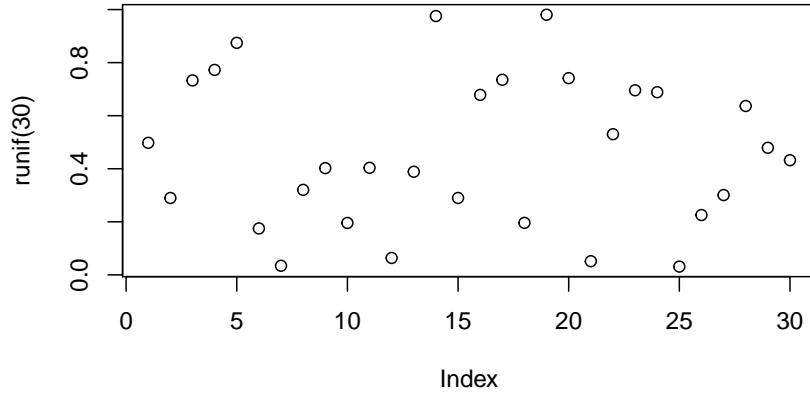


Die `plot()`-Funktion wird später noch ausführlicher vorgestellt.

## 6.6 Simulation von Zufallsexperimenten

R zeichnet sich durch einige nützliche Funktionen zum Simulieren von Zufallsexperimenten aus. Zum Beispiel erzeugen wir mit `runif()` auf dem Intervall  $[0, 1]$  gleichverteilte Zufallszahlen.

```
runif(3)
## [1] 0.08075014 0.83433304 0.60076089
runif(3)
## [1] 0.157208442 0.007399441 0.466393497
plot(runif(30))
```



Weitere Funktionen im Zusammenhang mit Zufallszahlen und Verteilungen werden später noch ausführlicher vorgestellt.

## 6.7 Typen

Wie bereits erwähnt, kann R mit atomaren Vektoren verschiedenen Typs umgehen. Der Typ kann mit der Funktion `typeof()` abgefragt werden. Wir haben bereits mit atomaren Vektoren vom Typ `double` und

logical gearbeitet. Der Typ für Text heißt `character`.

```
typeof(c(3, pi, exp(1)))
## [1] "double"
typeof(c(T, F, T))
## [1] "logical"
typeof(c("asdf", "Hallo Welt!"))
## [1] "character"
```

## 6.8 Text

Die Elemente eines atomaren Vektors vom Typ `character` sind Zeichenketten (und nicht einzelne Buchstaben), genannt **Strings**. Mit `length()` erhalten wir die Anzahl der Strings eines `character`-Vektors. Die Anzahl der Symbole in einem String wird von `nchar()` zurückgegeben.

```
length("ein String")
## [1] 1
nchar("ein String")
## [1] 10
x <- c("zwei", "Strings")
length(x)
## [1] 2
nchar(x)
## [1] 4 7
```

`cat()` und `print()` geben Werte auf der Konsole aus.

```
x <- 1:10
x
## [1] 1 2 3 4 5 6 7 8 9 10
print(x)
## [1] 1 2 3 4 5 6 7 8 9 10
cat(x)
## 1 2 3 4 5 6 7 8 9 10
```

Hier ist `print(x)` gleich der Eingabe von `x` in der Konsole. Arbeitet man mit Skripten und dem Befehl `source()` oder definiert man selbst Funktionen (nächster Abschnitt), wird die Ausgabe bei `x` unterdrückt und muss explizit mit `print(x)` gefordert werden.

Im Unterschied zu `print()` nimmt `cat()` eine beliebige Anzahl an Argumenten. Diese werden mit einem Leerzeichen getrennt zusammengefügt. Das Trennsymbol kann mit dem Argument `sep` bestimmt werden. `cat()` erzeugt automatisch keinen Zeilenumbruch. Dieser kann mit `\n` eingefügt werden.

```
cat(1);cat("A")
## 1A

cat(1, "\n");cat(2)
## 1
## 2
cat(1:3, "A\nB\n\nC\n", sep="_")
## 1_2_3_A
## B
##
## C
```

Für die Umwandlung von Zahlen in den Typ `character` ohne direkte Ausgabe auf der Konsole, siehe `?sprintf`, `?as.character`, `?paste`.

```

str1 <- as.character(pi)
str1
## [1] "3.14159265358979"
str2 <- sprintf("Pi ist ungefähr %.2f und e ist ca. %.2f", pi, exp(1))
str2
## [1] "Pi ist ungefähr 3.14 und e ist ca. 2.72"
str3 <- paste0("pi = ", pi, " und e = ", exp(1))
str3
## [1] "pi = 3.14159265358979 und e = 2.71828182845905"

```

## 7 Funktionen definieren

Mit dem reservierten Wort `function` können wir eigene Funktionen definieren. Die Syntax dafür folgt dem Schema `fun_name <- function(arg1, arg2, arg3, ...) expression`.

`expression` ist hierbei ein einzelner Befehl oder eine Folge von Befehlen (getrennt durch Zeilenumbruch oder `;`), die von `{ }` umschlossen ist.

```

myfun <- function(x, y) {
  thesum <- x+y
  2*thesum
}
myfun(1, 2)
## [1] 6

f <- function(x, y) x + 10*y
f(1,2)
## [1] 21

```

Der Rückgabewert der Funktion ist der letzte ausgewertete Ausdruck. Alternativ können wir den Rückgabewert explizit mit `return()` bestimmen. Befehle nach `return()` werden nicht ausgeführt.

```

myfun <- function(x, y) {
  thesum <- x+y
  x # führt nicht zu Ausgabe auf Konsole
  print(y) # Ausgabe auf Konsole
  return(2*thesum)
  print(123) # ignoriert, da nach return()
}
myfun(1, 2)
## [1] 2
## [1] 6

```

Die Argumente einer Funktion können **Default-Werte** haben. Schema: `function(arg1=default1, arg2=default2, ...)`. Argumente mit Default-Werten müssen beim Aufruf nicht angegeben werden. Sie sind daher **optionale Argumente**.

```

f <- function(x, y=0) {
  x + 10*y
}
f(1, 2)
## [1] 21
f(3)
## [1] 3

```

Beim Funktionsaufruf können wir die Namen der Argumente explizit benennen.

```

f <- function(x, y=0, z=0) {
  x + 10*y + 100*z
}
f(1, z=2)
## [1] 201
f(3, 4, 5)
## [1] 543
f(z=6, y=7, x=8)
## [1] 678

```

Beim Aufruf `f(x)` wird der Wert von `x` kopiert und der Funktion `f()` übergeben (und keine Referenz auf `x`). Dieses Verhalten wird mit **call by value** bezeichnet (im Gegensatz zu **call by reference**).

```

swapFirstTwo <- function(x) {
  tmp <- x[2]
  x[2] <- x[1]
  x[1] <- tmp
}
x <- 1:3
x
## [1] 1 2 3
swapFirstTwo(x) # kein Effekt
x
## [1] 1 2 3

# Stattdessen müssten wir wie folgt vorgehen:
swapFirstTwo <- function(x) {
  tmp <- x[2]
  x[2] <- x[1]
  x[1] <- tmp
  return(x)
}
x <- 1:3
x <- swapFirstTwo(x)
x
## [1] 2 1 3

```

*Bemerkung:* Da `swapFirstTwo()` mit einem Befehl realisierbar ist, lohnt es sich nicht eine eigene Funktion dafür zu definieren.

```

x <- 1:3
x[1:2] <- x[2:1] #swapFirstTwo
x
## [1] 2 1 3

```

Bei längeren Skripten ist es jedoch äußerst hilfreich, den Code dadurch zu **strukturieren**, dass zusammgehörende oder wiederkehrende Folgen von Befehlen zu Funktionen zusammengefasst werden.

## 8 Listen

Die wichtigste Datenstruktur in R neben atomaren Vektoren sind **Listen**. Dies sind geordnete Sammlungen von R-Objekten beliebigen Typs. Sie können mit `list()` erzeugt werden.

```

lst1 <- list(c(1,5), c("hallo", "wie geht's"))
lst1

```

```

## [[1]]
## [1] 1 5
##
## [[2]]
## [1] "hallo"      "wie geht's"
lst2 <- list(1, 2, list("a", 3))
lst2
## [[1]]
## [1] 1
##
## [[2]]
## [1] 2
##
## [[3]]
## [[3]][[1]]
## [1] "a"
##
## [[3]][[2]]
## [1] 3

```

**Vektor** ist der Überbegriff für **atomare Vektoren** und **Listen**.

`length()` gibt die Anzahl der Element einer Liste aus.

```

length(lst1)
## [1] 2
length(lst2)
## [1] 3

```

Listen können geschachtelt werden, atomare Vektoren nicht.

```

list(1, list(2, list(3)))
## [[1]]
## [1] 1
##
## [[2]]
## [[2]][[1]]
## [1] 2
##
## [[2]][[2]]
## [[2]][[2]][[1]]
## [1] 3
c(1, c(2, c(3)))
## [1] 1 2 3

```

Auf einzelne Elemente einer Liste greifen wir mit `[[` zu.

```

lst1[[1]]
## [1] 1 5
lst1[[2]]
## [1] "hallo"      "wie geht's"

```

Wenn wir mehrere Elemente auswählen wollen, benutzen wir `[`. Dann erhalten wir eine Liste der ausgewählten Elemente.

```

lst2[1:2]
## [[1]]

```

```
## [1] 1
##
## [[2]]
## [1] 2
lst2[1] # auch Liste!
## [[1]]
## [1] 1
```

Listen können Elemente beliebigen Typs enthalten, sogar Listen und Funktionen. Im Gegensatz dazu sind die Elemente atomarer Vektoren alle vom selben Typ und dieser Typ muss ein **Basistyp** sein (zB `logical`, `double` oder `character`). Es gibt keine atomaren Vektoren mit Elementen vom Typ “Liste” oder Elementen vom Typ “Funktion”.

```
m <- list(max, min) # Liste der Funktionen max() und min()
m[[1]](1:3)
## [1] 3
m[[2]](1:3)
## [1] 1
```

Um Listen etwas kompakter auf der Konsole auszugeben, verwenden wir die Funktion `str()`.

```
str(lst1)
## List of 2
## $ : num [1:2] 1 5
## $ : chr [1:2] "hallo" "wie geht's"
```

Zwei oder mehrere Listen können mit `c()` zusammengehängt werden.

```
str(c(lst1, lst2))
## List of 5
## $ : num [1:2] 1 5
## $ : chr [1:2] "hallo" "wie geht's"
## $ : num 1
## $ : num 2
## $ :List of 2
##   ..$ : chr "a"
##   ..$ : num 3
```

`unlist()` wandelt eine Liste in einen atomaren Vektor; `as.list()` wandelt einen atomaren Vektor in eine Liste.

```
x <- list(1, 2, c(3,4))
str(x)
## List of 3
## $ : num 1
## $ : num 2
## $ : num [1:2] 3 4
unlist(x)
## [1] 1 2 3 4
str(as.list(c(5,6,7)))
## List of 3
## $ : num 5
## $ : num 6
## $ : num 7
```

## 9 Kontrollstrukturen

Wie aus anderen Programmiersprachen bekannt, gibt es auch in R **Kontrollstrukturen** (*Control Flow*). Wir gehen hier davon aus, dass ihre Funktionsweise bereits bekannt ist und erklären nur die Syntax.

### 9.1 Bedingte Anweisungen

Die Syntax für bedingte Anweisungen entspricht dem Schema `if (condition) expression` bzw `if (condition) expression_TRUE else expression_FALSE`

`condition` ist ein Ausdruck, der evaluiert entweder TRUE oder FALSE ergibt.

```
x <- 1
y <- FALSE

if (x > 0) {
  cat("x größer 0\n")
}
## x größer 0

if (y) {
  cat("y ist wahr\n")
} else {
  cat("y ist falsch\n")
}
## y ist falsch
```

Falls der bedingt auszuführende Code nur aus einer Anweisung besteht, können die geschweiften Klammern weggelassen werden. Jedoch ist Vorsicht bei einem darauffolgenden `else` geboten.

```
if (TRUE) print("Das ist immer wahr!")
## [1] "Das ist immer wahr!"

if (FALSE) 1 # if-Ausdruck endet hier
else 2 # ERROR

if (FALSE) 1 else 2
## [1] 2
if (FALSE) 1 else
  2
## [1] 2
if (FALSE) {
  1
} else 2
## [1] 2
```

Iterative if `else`- Ausdrücke erlauben die Unterscheidung mehrerer Werte.

```
w <- "blue"
if (w == "red") {
  cat("roses are red\n")
} else if (w == "blue") {
  cat("violets are blue\n")
} else if (w == "green") {
  cat("gras is green\n")
} else {
  cat("color unknown\n")
```

```
}
```

## violets are blue

## 9.2 while-Schleife

while-Schleifen haben in R die Form `while(condition) expression`.

```
x <- 3
while (x > 0) {
  print(x)
  x <- x - 1
}
## [1] 3
## [1] 2
## [1] 1
x
## [1] 0
```

Schleifen können mit `break` abgebrochen werden. Mit `next` wird nur der aktuelle Schleifendurchlauf beendet und mit dem nächsten angefangen.

```
x <- 10
while (TRUE) {
  x <- x - 1
  if (x %% 2 != 0) next
  cat(x)
  if (x <= 0) break
}
## 86420
```

## 9.3 for-Schleife

Mit einer for-Schleife können wir über einen Vektor (atomarer oder Liste) iterieren: `for (iterator in vector) expression`. `next` und `break` können hier analog zu while-Schleifen benutzt werden.

```
x <- 0
for (i in 1:10) x <- x + i
x
## [1] 55
lst <- list(FALSE, 1, "two")
for (y in lst) {
  print(length(y))
  cat(y, "\n")
}
## [1] 1
## FALSE
## [1] 1
## 1
## [1] 1
## two
```

In diesem Zusammenhang ist die Funktion `seq_along()` nützlich. Sie erzeugt einen Index-Vektor ihres Argumentes.

```
seq_along(lst)
## [1] 1 2 3
```

```

for (i in seq_along(lst))
  cat(letters[i], ":", lst[[i]], "\n")
## a : FALSE
## b : 1
## c : two

```

Bemerkung: Der Vektor `letters` enthält die Buchstaben des Alphabets.

```

letters
## [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l" "m" "n" "o" "p" "q" "r" "s"
## [20] "t" "u" "v" "w" "x" "y" "z"

```

## 10 Zufall

Mit R lassen sich Zufallszahlen aus verschiedenen Verteilungen ziehen. Die Benennung der entsprechenden Funktion folgt dem Schema `rDISTRI()`, wobei `DISTRI` durch ein Kürzel einer Verteilung ersetzt werden muss.

```

runif(8) # uniform distribution
## [1] 0.7064338 0.9485766 0.1803388 0.2168999 0.6801629 0.4988456 0.6416793
## [8] 0.6602843
rbinom(8, size=5, prob=0.5) # binomial distribution
## [1] 1 3 3 5 5 2 2 2
rnorm(8, mean=0, sd=1) # normal distribution
## [1] -0.93584735 -0.01595031 -0.82678895 -1.51239965 0.93536319 0.17648861
## [7] 0.24368546 1.62354888

```

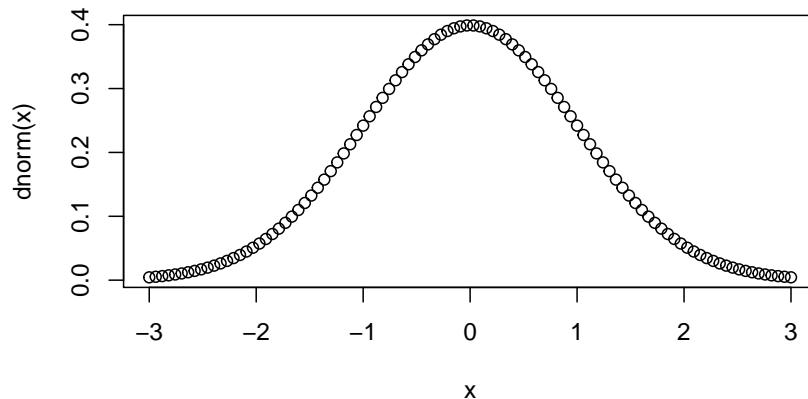
Siehe `?Distributions` für eine Übersicht und eine Beschreibung der Parameter.

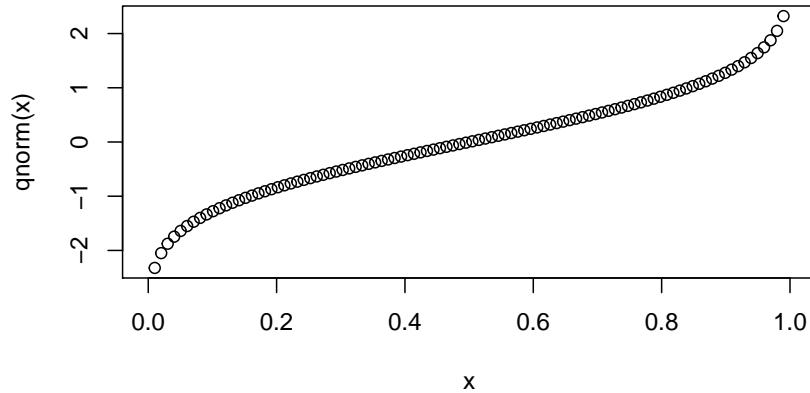
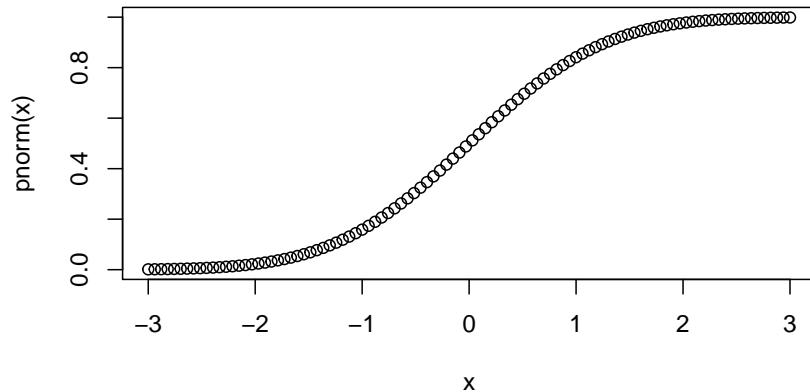
Dichte, Verteilungsfunktion und Quantilfunktion folgen dem Schema `dDISTRI()`, `pDISTRI()`, bzw `qDISTRI()`.

```

x <- seq(-3, 3, length.out=100)
plot(x, dnorm(x))
plot(x, pnorm(x))
x <- seq(0, 1, length.out=100)
plot(x, qnorm(x))

```





Für die `rDISTR()`-Funktionen benutzt R einen **Pseudo-Zufallszahlengenerator**. Da ein Computer eine deterministische Maschine ist, kann R keine “echten” Zufallswerte erzeugen. Ein Pseudo-Zufallszahlengenerator startet mit einem Initialwert und erzeugt darauf aufbauend eine Folge von Zahlen die “zufällig aussieht”, aber vollkommen deterministisch ist. Der Initialwert für den R-internen Pseudo-Zufallszahlengenerator kann mit der Funktion `set.seed()` gesetzt werden. Dies ist sehr nützlich um reproduzierbare “Zufalls-Simulationen” zu erstellen.

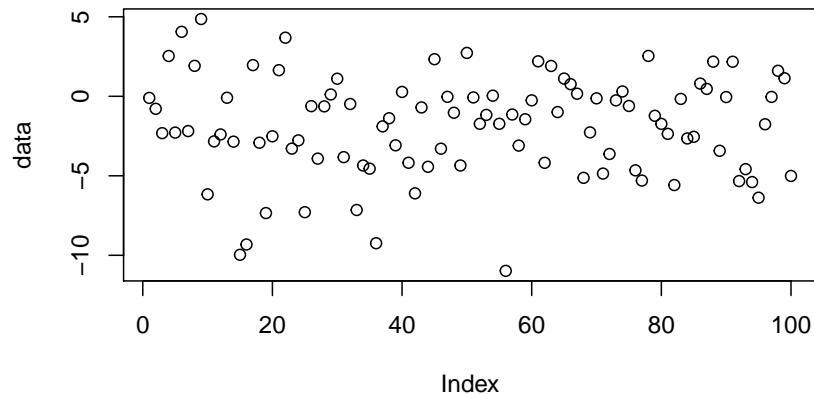
```
runif(3) # 1. Aufruf
## [1] 0.5446034 0.2785972 0.4467025
runif(3) # 2. Aufruf ergibt neue Werte
## [1] 0.37151118 0.02806097 0.46598719
set.seed(42)
runif(3) # 3.
## [1] 0.9148060 0.9370754 0.2861395
runif(3) # 4.
## [1] 0.8304476 0.6417455 0.5190959
set.seed(42)
runif(3) # gleiche Werte wie bei 3.
## [1] 0.9148060 0.9370754 0.2861395
runif(3) # gleiche Werte wie bei 4.
```

```
## [1] 0.8304476 0.6417455 0.5190959
```

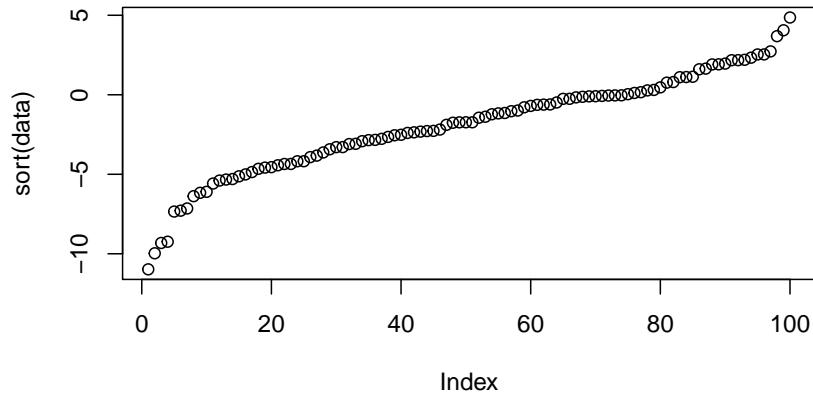
## 11 Summary Statistics

Wir listen ein paar Funktionen auf, die einfache Zusammenfassungen numerischer Daten erlauben.

```
data <- rnorm(100, mean=-2, sd=3) # erzeuge Daten
plot(data)
```



```
mean(data) # arithmetisches Mittel
## [1] -1.900262
median(data) # Median
## [1] -1.73061
var(data) # empirische Varianz
## [1] 9.86938
sd(data) # empirische Standardabweichung
## [1] 3.141557
min(data) # Minimum
## [1] -10.97927
max(data) # Maximum
## [1] 4.859936
which.min(data) # Argmin
## [1] 56
which.max(data) # Argmax
## [1] 9
summary(data) # mehrere Statistiken, inklusive Quartile
##      Min. 1st Qu.  Median      Mean 3rd Qu.      Max.
## -10.97927 -3.98502 -1.73061 -1.90026  0.05703  4.85994
plot(sort(data))
```



## 12 Matrizen

R kann mit Matrizen rechnen. Diese werden mit der Funktion `matrix()` erzeugt.

```
matrix(1:12, nrow=3)
##      [,1] [,2] [,3] [,4]
## [1,]    1    4    7   10
## [2,]    2    5    8   11
## [3,]    3    6    9   12
matrix(1:12, ncol=3)
##      [,1] [,2] [,3]
## [1,]    1    5    9
## [2,]    2    6   10
## [3,]    3    7   11
## [4,]    4    8   12
matrix(1:12, ncol=3, byrow=TRUE)
##      [,1] [,2] [,3]
## [1,]    1    2    3
## [2,]    4    5    6
## [3,]    7    8    9
## [4,]   10   11   12
```

Das erste Argument ist ein Vektor aller Einträge der Matrix. Mit der Angabe von `nrow` oder `ncol` wird die Zeilen- bzw Spaltenzahl festgelegt. In welcher Reihenfolge die Daten in die Matrix gefüllt werden, kann mit dem Argument `byrow` beeinflusst werden. Standardmäßig wird Spalte für Spalte von oben nach unten befüllt (*column-major order*).

Die Abfrage der Größe einer Matrix wird in folgenden Beispielen gezeigt.

```
m <- matrix(1:12, nrow=3)
nrow(m) # Zeilenzahl
## [1] 3
ncol(m) # Spaltenzahl
## [1] 4
length(m) # Anzahl Einträge gesamt
## [1] 12
dim(m) # Vektor der Zeilen- und Spaltenzahl
```

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

Auf Elemente von Matrizen kann mit der Angabe von zwei Indizes (für Zeile und Spalte) zugegriffen werden. Die Indizes werden durch ein Komma getrennt. Mehrfachauswahl ist analog zu atomaren Vektoren möglich.

```
m
##      [,1] [,2] [,3] [,4]
## [1,]    1    4    7   10
## [2,]    2    5    8   11
## [3,]    3    6    9   12
m[3,2] # Zeile zuerst (3), Spalte später (2)
## [1] 6
m[2:3,1]
## [1] 2 3
m[c(1,2),c(1,3)] # 2x2 Untermatrix der Elemente aus Zeile 1 und 2,
##      [,1] [,2]
## [1,]    1    7
## [2,]    2    8
# die gleichzeitig auch in Spalte 1 oder 3 stehen;
# entspricht Streichung von Zeile 3 und Spalten 2 und 4
```

Wird der Zeilen- oder Spaltenindex freigelassen, wird jede Zeile bzw Spalte ausgewählt, dh ganze Spalten bzw Zeilen.

```
m[2, ] # 2. Zeile
## [1] 2 5 8 11
m[ ,c(4,1)] # Spalte 4 und 1
##      [,1] [,2]
## [1,]    10    1
## [2,]    11    2
## [3,]    12    3
```

Wir können mit Matrizen rechnen:

```
m1 <- matrix(0:5, nrow=2)
m2 <- matrix(-5:0, nrow=2)
m1
##      [,1] [,2] [,3]
## [1,]    0    2    4
## [2,]    1    3    5
m2
##      [,1] [,2] [,3]
## [1,]   -5   -3   -1
## [2,]   -4   -2    0
m1 + m2 # elementweise Addition
##      [,1] [,2] [,3]
## [1,]   -5   -1    3
## [2,]   -3    1    5
2 * m1 # skalare Multiplikation
##      [,1] [,2] [,3]
## [1,]    0    4    8
## [2,]    2    6   10
m1 * m2 # elementweise Multiplikation
##      [,1] [,2] [,3]
## [1,]    0   -6   -4
## [2,]   -4   -6    0
```

```

t(m2) # transponierte Matrix
## [,1] [,2]
## [1,] -5 -4
## [2,] -3 -2
## [3,] -1  0
m1 %*% t(m2) # Matrixmultiplikation
## [,1] [,2]
## [1,] -10 -4
## [2,] -19 -10

```

Mit `cbind()` und `rbind()` werden Vektoren oder Matrizen an den Spalten bzw Zeilen zu neuen Matrizen verbunden.

```

cbind(m1, c(10,20))
## [,1] [,2] [,3] [,4]
## [1,] 0 2 4 10
## [2,] 1 3 5 20
rbind(m2, m1)
## [,1] [,2] [,3]
## [1,] -5 -3 -1
## [2,] -4 -2  0
## [3,] -1  0  2
## [4,]  1  3  5

```

Weitere Funktionen auf Matrizen:

```

m <- matrix(1:9, nrow=3)
m[2,2] <- 0
m[3,3] <- 0
m
## [,1] [,2] [,3]
## [1,] 1 4 7
## [2,] 2 0 8
## [3,] 3 6 0
det(m) # Determinante
## [1] 132
y <- solve(m, rep(1,3)) # löse Gleichungssystem
y
## [1] 0.19696970 0.06818182 0.07575758
m %*% y
## [,1]
## [1,] 1
## [2,] 1
## [3,] 1

```

## 13 Plots

Die Funktion `plot()` hat einige optionale Argumente, mit denen die Gestalt des Plots beeinflusst werden kann.

```

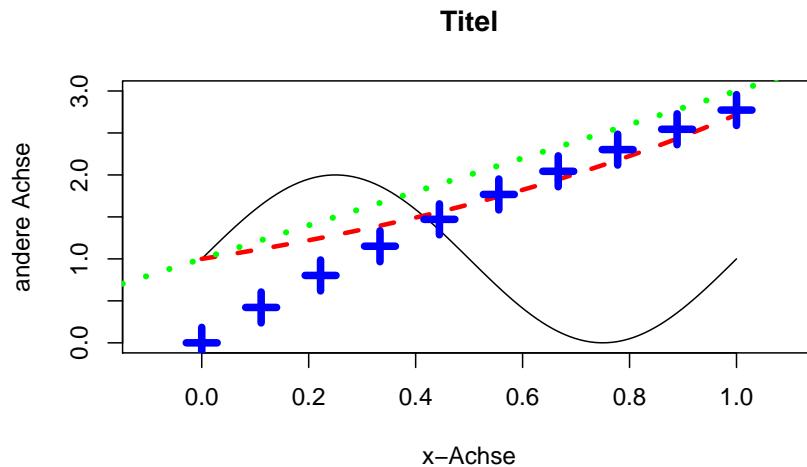
x <- seq(0, 1, length.out=100)
y1 <- sin(2*pi*x)+1
y2 <- exp(x)
x2 <- seq(0, 1, length.out=10)
y3 <- log(x2+1)*4

```

```

plot(x, y1, # Erstelle neuen Plot
      type="l", # Zeichne Plot vom Typ "l": Linie
      xlim=c(-0.1,1.1), ylim=c(0, 3), # c(min,max) der jeweiligen Achse
      main="Titel", # Überschrift
      xlab="x-Achse", ylab="andere Achse") # Achsenbeschriftung
lines(x, y2, # Füge zum bestehenden Plot eine neue Kurve vom Typ "l" (Linie) hinzu
      lty=2, # Linientyp 2: gestrichelt
      lwd=3,
      col="red") # Farbe rot
points(x2, y3, # Füge zum bestehenden Plot eine neue Kurve vom Typ "p" (Punkte) hinzu
       pch=3, # Symbol Nummer 3 (plus) für die Punkte verwenden
       col="#0000FF", # Farbe im Format "#RRGGBB" R: rot, G: grün, B: blau, im Hexadezimalsystem
       cex=2, # Größe des Symbols
       lwd=5) # Liniendicke für das Malen des Symbols
abline(1,2, # Füge zum bestehenden Plot eine neue Linie mit intercept 1 und Steigung 2 hinzu
       col="green",
       lwd=4,
       lty=3) # Linientyp 3: gepunktet

```

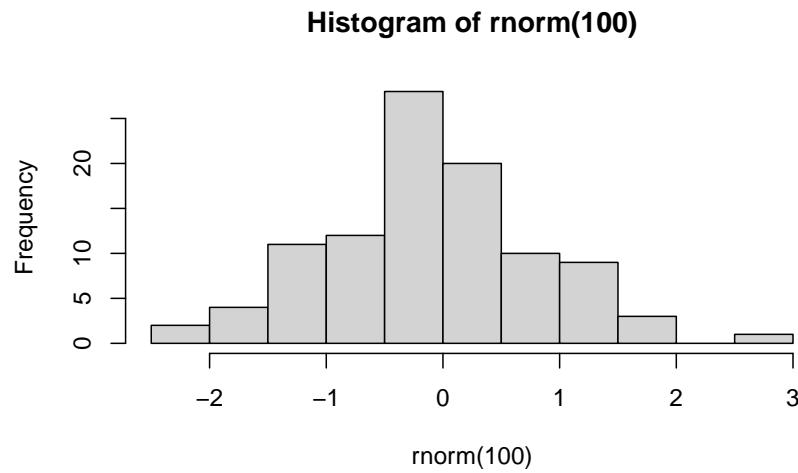


Für eine Dokumentation der Plotfunktion siehe `?plot.default`.

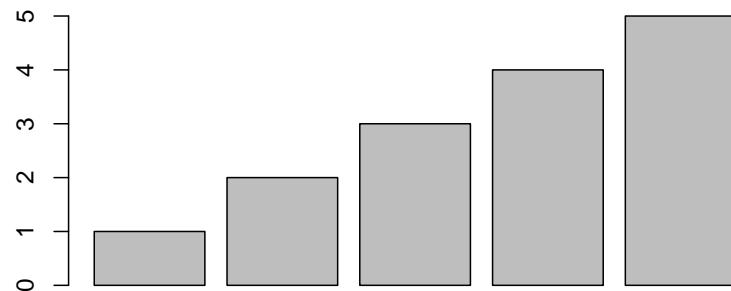
Die Funktion `plot()` erzeugt standardmäßig einen neuen Plot, wohingegen `points()`, `lines()` und `abline()` in den zuletzt erzeugten Plot zeichnen.

Wir zeigen noch beispielhaft weitere Möglichkeiten Plots zu erzeugen. Um Details herauszufinden, können wir jederzeit die Dokumentation mittels `? konsultieren.`

```
hist(rnorm(100))
```



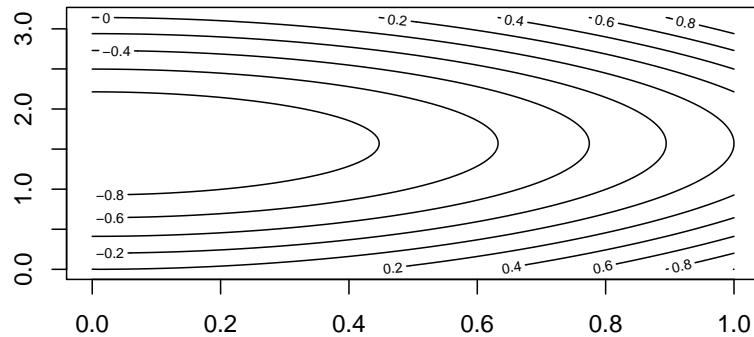
```
barplot(1:5)
```



```

x <- seq(0,1,length.out = 100)
y <- seq(0,pi,length.out = 100)
f <- function(x,y) {
  x^2-sin(y)
}
z <- matrix(nrow = length(x), ncol=length(y)) # erzeuge leere Matrix der angegebenen Dimensionen
for (ix in seq_along(x)) {
  for (iy in seq_along(y)) {
    z[ix, iy] <- f(x[ix], y[iy])
  }
}
contour(x, y, z)

```



## 14 Pakete

Pakete erweitern die Liste der in R verfügbaren Funktionen.

`library(package_name)` macht den Inhalt eines Paketes verfügbar.

Ist das Paket noch nicht lokal vorhanden, muss es zunächst heruntergeladen und installiert werden. Dies geschieht mittels `install.packages("package_name")`.

`install.packages()` muss nur einmal aufgerufen werden und sollte nicht Teil von R-Skripten sein. `library()` muss bei jeder neuen “R-Session” aufgerufen werden.

Ist ein Paket installiert aber noch nicht mit `library()` geladen, können Funktionen des Paketes auch mittels `package_name::function_name()` aufgerufen werden.

Das Paket `lubridate` stellt Funktionen zum Umgang mit Kalenderdaten zur Verfügung. ZB wandelt die Funktion `lubridate::dmy()` einen Text-String in ein Datums-Objekt. Mit `lubridate::day()`, `lubridate::month()`, `lubridate::year()` erhalten wir die entsprechenden Komponenten des Datums.

```
# install.packages("lubridate") bereits ausgeführt
x <- lubridate::dmy("20.4.20")
lubridate::day(x)
## [1] 20
lubridate::month(x)
## [1] 4

month(x) # ERROR: Funktion unbekannt

library(lubridate)
##
## Attaching package: 'lubridate'
## The following objects are masked from 'package:base':
##
##     date, intersect, setdiff, union
month(x)
## [1] 4
year(x)
## [1] 2020
```

## 15 Tibbles

Die gängigste Datenstruktur, mit der Datensätze in R verfügbar gemacht werden, sind **Tibbles**. Tibbles sind Tabellen. Sie haben eine Rechteckstruktur wie Matrizen. Jede Spalte steht für eine Beobachtungsvariable / ein Feature und jede Zeile für eine Beobachtung. Im Gegensatz zu Matrizen können verschiedene Spalten von unterschiedlichem Typ sein.

Tibbles werden vom Paket `tibble` bereitgestellt.

```
library(tibble)
tbl <- tibble( # Erzeugen eines Tibbles
  name = c("Alice", "Bob", "Claire", "Dave"), # Variable 1 / Spalte 1 hat die Bezeichnung 'name' und
  # als Wert den angegebenen atomaren Vektor vom Typ character
  age = c(21, 25, 37, 22), # Variable 2 / Spalte 2: Bezeichnung 'age', Typ: double
  left_handed = c(T, F, F, T) # Variable 3 / Spalte 3: Bezeichnung 'left_handed', Typ: logical
  # alle Spalten haben Länge 4
)
tbl
## # A tibble: 4 x 3
##   name     age left_handed
##   <chr>   <dbl> <lgl>
## 1 Alice     21 TRUE
## 2 Bob       25 FALSE
## 3 Claire    37 FALSE
## 4 Dave      22 TRUE
tbl[1,] # Indizierung analog zu Matrizen
## # A tibble: 1 x 3
##   name     age left_handed
##   <chr>   <dbl> <lgl>
## 1 Alice     21 TRUE
tbl[,c(1,3)]
## # A tibble: 4 x 2
##   name   left_handed
##   <chr>   <lgl>
## 1 Alice   TRUE
## 2 Bob    FALSE
## 3 Claire FALSE
## 4 Dave   TRUE
tbl$age # Außerdem können benannte Spalten mir $ ausgewählt werden
## [1] 21 25 37 22
```

### 15.1 Daten einlesen

Typischerweise möchte man externe Daten einlesen und nicht Daten von Hand im R-Code erstellen. Daten können in den unterschiedlichsten Formaten vorliegen. Eines der einfachsten Formate ist `.csv` (*comma seperated values*). Dateien dieses Formats sind einfache Textdateien, die eine Tabellenstruktur haben, wobei Beobachtungen durch Zeilenumbrüche und Spalten durch , oder ; getrennt werden. Um eine solche Datei als Tibble einzulesen, nutzen wir die Funktion `readr::read_csv()`:

```
library(readr)
cntrs <- read_csv("NV01_countries.csv")
## Parsed with column specification:
## cols(
##   .default = col_character(),
##   Population = col_double(),
##   `Area (sq. mi.)` = col_double(),
```

```

## `Infant mortality (per 1000 births)` = col_number(),
## `GDP ($ per capita)` = col_double(),
## `Literacy (%)` = col_number(),
## `Other (%)` = col_number(),
## Climate = col_number(),
## Birthrate = col_number(),
## Deathrate = col_number()
## )
## See spec(...) for full column specifications.

```

Die Datei "01\_countries.csv" listet Länder mit verschiedenen Daten wie Fläche, Bevölkerungszahl, etc. `read_csv()` gibt einige Informationen aus, wie die Datei eingelesen wurde. Diese ignorieren wir für den Moment.

Der Datensatz ist zu groß, um ihn vollständig in der Konsole auszugeben. Solch große Tibbles werden automatisch verkürzt ausgegeben.

```

nrow(cntrs) # Zeilenzahl
## [1] 227
ncol(cntrs) # Spaltenzahl
## [1] 20
cntrs
## # A tibble: 227 x 20
##   Country Region Population `Area (sq. mi.)` `Pop. Density (~ `Coastline (coa~
##   <chr>   <chr>        <dbl>           <dbl> <chr>           <chr>
## 1 Afghan~ ASIA~ 31056997 647500 48,0 0,00
## 2 Albania EASTE~ 3581655 28748 124,6 1,26
## 3 Algeria NORTH~ 32930091 2381740 13,8 0,04
## 4 Americ~ OCEAN~ 57794 199 290,4 58,29
## 5 Andorra WESTE~ 71201 468 152,1 0,00
## 6 Angola SUB-S~ 12127071 1246700 9,7 0,13
## 7 Anguil~ LATIN~ 13477 102 132,1 59,80
## 8 Antigu~ LATIN~ 69108 443 156,0 34,54
## 9 Argent~ LATIN~ 39921833 2766890 14,4 0,18
## 10 Armenia C.W.~ 2976372 29800 99,9 0,00
## # ... with 217 more rows, and 14 more variables: `Net migration` <chr>, `Infant
## # mortality (per 1000 births)` <dbl>, `GDP ($ per capita)` <dbl>, `Literacy
## # (%)` <dbl>, `Phones (per 1000)` <chr>, `Arable (%)` <chr>, `Crops
## # (%)` <chr>, `Other (%)` <dbl>, Climate <dbl>, Birthrate <dbl>,
## # Deathrate <dbl>, Agriculture <chr>, Industry <chr>, Service <chr>

```

Mit `View()` wird ein Datensatz im RStudio-Data-Viewer gezeigt. Alternativ kann dieser mit einem Links-klick auf die Variable im *Environment-Pane* in RStudio geöffnet werden.

## 16 apply-Funktionen

Um den Code übersichtlich und besser verständlich zu machen, werden in R oft Funktionen wie `apply()`, `lapply()` und `sapply()` anstatt `for`-Schleifen eingesetzt.

Mit `lapply(x, f)` wird die Funktion `f()` auf jedes Element des Vektors `x` angewendet.

```

lst <- list(T, 42, "asdf")
fun <- function(x) {
  paste0("value: ", x, ", type: ", typeof(x))
}

```

```
res <- lapply(lst, fun)
res
## [[1]]
## [1] "value: TRUE, type: logical"
##
## [[2]]
## [1] "value: 42, type: double"
##
## [[3]]
## [1] "value: asdf, type: character"
```

Die Ausgabe von `lapply()` ist immer eine Liste. Sind die Ausgabewerte der Funktion immer vom selben Typ, kann stattdessen mit `sapply()` ein atomarer Vektor (oder eine Matrix) ausgegeben werden.

```
sapply(lst, fun)
## [1] "value: TRUE, type: logical"    "value: 42, type: double"
## [3] "value: asdf, type: character"
```

Mit einer `for`-Schleife würden wir statt `res <- sapply(lst, fun)` folgenden, etwas unübersichtlicheren Code schreiben.

```
res <- rep("", length(lst))
for (i in seq_along(lst)) {
  res[i] <- fun(lst[[i]])
}
```

Der Befehl `apply(mat, 1, f)` für eine Matrix `mat` und eine Funktion `f()` wendet `f()` auf jede Zeile von `mat` an. Dabei muss `f(x)` für atomare Vektoren `x` ein sinnvolles Ergebnis liefern.

```
mat <- matrix(1:12, nrow = 3)
mat
##      [,1] [,2] [,3] [,4]
## [1,]    1    4    7   10
## [2,]    2    5    8   11
## [3,]    3    6    9   12
apply(mat, 1, mean) # Mittelwert für jede Zeile
## [1] 5.5 6.5 7.5
fun <- function(x) c(min(x), max(x))
apply(mat, 1, fun)
##      [,1] [,2] [,3]
## [1,]    1    2    3
## [2,]   10   11   12
```

Analog wendet man mit `apply(mat, 2, f)` die Funktion `f()` auf die Spalten der Matrix `mat` an.

## 17 Statistik

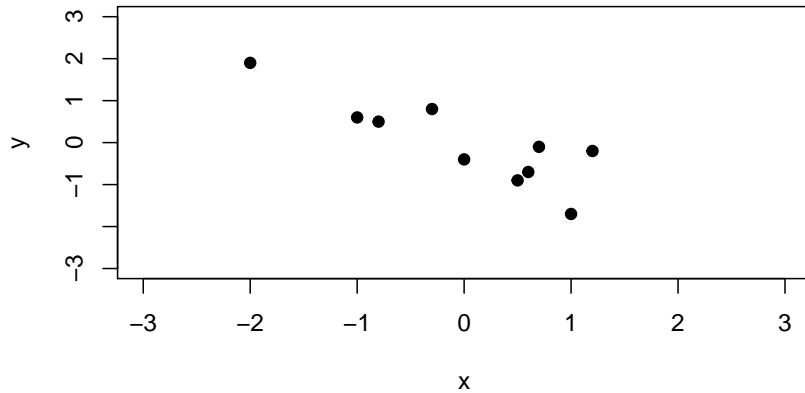
R ist in besonderem Maße dafür ausgelegt, statistische Auswertungen und Simulationen durchzuführen und Resultate ggf. grafisch darzustellen. Dies haben wir bereits in einigen Abschnitten bemerken können:

- erzeuge Zufallszahlen mit `rDISTRI()`
- Verteilungs-, Dichte- und Quantil-Funktionen: `pDISTRI()`, `dDISTRI()`, `qDISTRI()`
- Datensätze als Tibbles einlesen
- grafische Darstellung mit `plot()`, `hist()`, ...
- simple Statistiken der Daten berechnen: `mean()`, `var()`, ...

## 17.1 Lineare Regression

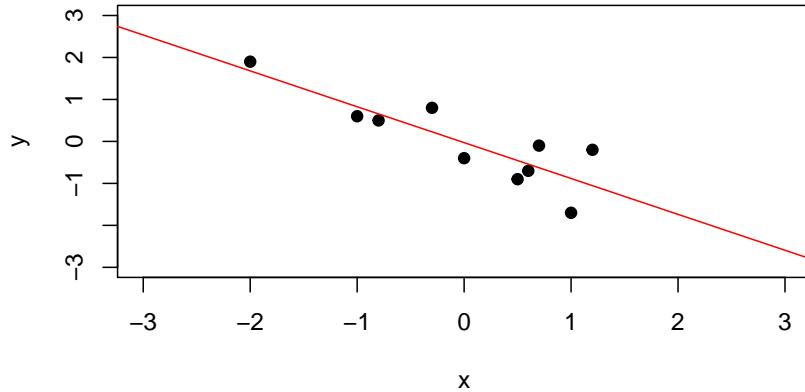
Gegeben seien folgende Daten:

```
x <- c(-2, -1, -0.8, -0.3, 0, 0.5, 0.6, 0.7, 1, 1.2)
y <- c(1.9, 0.6, 0.5, 0.8, -0.4, -0.9, -0.7, -0.1, -1.7, -0.2)
plot(x, y, xlim = c(-3, 3), ylim = c(-3, 3), pch = 19)
```



Wir suchen eine Gerade der Form  $g(x) = \text{intercept} + \text{slope} * x$ , wobei `intercept` und `slope` reelle Zahlen sind, sodass  $g(x)$  möglichst nahe an  $y$  ist (genauer: minimiere quadratischen Fehler  $\text{sum}((g(x)-y)^2)$ ). In R lässt sich dies mit Hilfe der Funktion `lm()` (*linear model*) schnell umsetzen.

```
fit <- lm(y ~ x)
fit
##
## Call:
## lm(formula = y ~ x)
##
## Coefficients:
## (Intercept)          x
## -0.02855      -0.85468
plot(x, y, xlim = c(-3, 3), ylim = c(-3, 3), pch = 19)
abline(fit, col = "red")
```



Der Rückgabewert der Funktion `lm()` ist eine Liste mit verschiedenen benannten Werten. Die Namen erhalten wir mit `names()`.

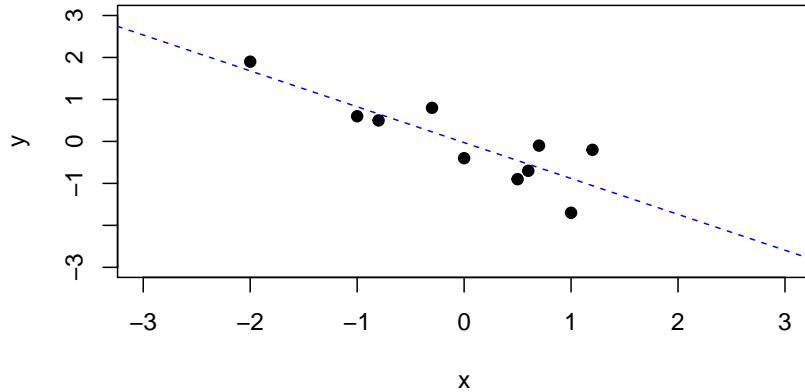
```
typeof(fit)
## [1] "list"
names(fit)
##  [1] "coefficients"   "residuals"      "effects"        "rank"
##  [5] "fitted.values"  "assign"        "qr"             "df.residual"
##  [9] "xlevels"        "call"          "terms"          "model"
```

Die gesuchten Koeffizienten `intercept` und `slope` finden wir im ersten Eintrag (als atomarer Vektor vom Typ `double` mit Länge zwei). Dieser hat den Namen `"coefficients"`.

```
fit[[1]]
## (Intercept)      x
## -0.02854677 -0.85467688
fit$coefficients # benannte Listen-Einträge können mit '$' abgerufen werden
## (Intercept)      x
## -0.02854677 -0.85467688
```

Im obigen Plot findet R automatisch diesen Eintrag in `fit`, um die Gerade mittels `abline()` zu zeichnen. Wir können dies auch “von Hand” ausführen.

```
coeff <- fit$coefficients
plot(x, y, xlim = c(-3, 3), ylim = c(-3, 3), pch = 19)
abline(coeff[1], coeff[2], col = "blue", lty=2)
```



Die Funktion `lm()` macht mehr, als nur die Koeffizienten zu bestimmen. Eine Zusammenfassung der Resultate erhalten wir mit

```
summary(fit)
##
## Call:
## lm(formula = y ~ x)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -0.8168 -0.3351 -0.1569  0.4499  0.8542
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept) -0.02855   0.17626 -0.162  0.87536
## x          -0.85468   0.18307 -4.669  0.00161 **
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.5574 on 8 degrees of freedom
## Multiple R-squared:  0.7315, Adjusted R-squared:  0.6979
## F-statistic: 21.79 on 1 and 8 DF,  p-value: 0.001606
```

Wir werden im weiteren Verlauf der Vorlesung noch genauer darauf eingehen, was diese Ausgaben bedeuten.

# 02 – Datenstrukturen

19. April 2021

## Contents

<b>1</b>	<b>Objekte</b>	<b>1</b>
<b>2</b>	<b>Vektoren</b>	<b>2</b>
2.1	Atomare Vektoren . . . . .	2
2.2	Combine . . . . .	7
2.3	Typenumwandlung . . . . .	8
2.4	Listen . . . . .	8
<b>3</b>	<b>Attribute</b>	<b>10</b>
3.1	Namen . . . . .	11
3.2	Weitere Bemerkungen . . . . .	12
<b>4</b>	<b>Matrizen und Arrays</b>	<b>14</b>
<b>5</b>	<b>S3 Objekte</b>	<b>19</b>
5.1	Faktoren . . . . .	20
5.2	Date . . . . .	21
5.3	POSIXct . . . . .	21
5.4	proc_time . . . . .	22
<b>6</b>	<b>Tibbles</b>	<b>22</b>
<b>7</b>	<b>Überblick Datenstrukturen</b>	<b>25</b>
<b>8</b>	<b>Numerische Verwirrung</b>	<b>25</b>
<b>9</b>	<b>Much Ado About Nothing</b>	<b>25</b>

## 1 Objekte

"To understand computations in R, two slogans are helpful:

- Everything that exists is an object.
- Everything that happens is a function call."

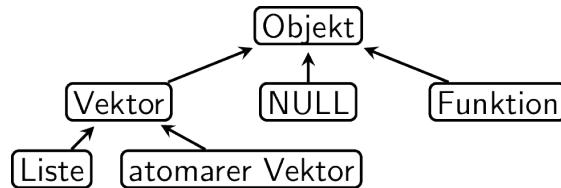
— *John M. Chambers*

Alles, was *existiert*, ist ein **Objekt**. Alles, was *passiert*, ist ein **Funktionsaufruf**.

Ein **R-Skript** ist eine Folge von Funktionsaufrufen getrennt durch ; oder Zeilenumbruch.

Beispiele für Objekte sind **Listen** und **atomare Vektoren**. Sie werden zusammen als **Vektoren** bezeichnet und sind Objekte zum Speichern von Daten. Funktionen sind ebenfalls Objekte. **NULL** repräsentiert das **Null-Objekt**, dazu später mehr.

```
NULL
## NULL
```



Objekte haben (unter anderem) folgende Eigenschaften:

- `typeof(x)` ist der **Typ** von x
- `class(x)` ist die **Klasse** von x
- `length(x)` ist die **Länge** von x

```
object_properties <- function(x) c(typeof(x), class(x), length(x))
object_properties(NULL)
## [1] "NULL" "NULL" "0"
object_properties(list(1, "asdf"))
## [1] "list" "list" "2"
object_properties(c(1, pi, exp(1)))
## [1] "double" "numeric" "3"
object_properties(mean)
## [1] "closure" "function" "1"
```

Wir werden später noch genauer auf den Typ und die Klasse eingehen.

## 2 Vektoren

### 2.1 Atomare Vektoren

Alle Elemente eines atomaren Vektors haben den gleichen Typ und dieser Typ muss ein Basistyp sein.

Die **6 Basistypen** sind: `logical`, `integer`, `double`, `character`, `complex`, `raw`.

```
misc_atomic <- list(T, 1:3, pi, "asdf", 2+3i, raw(4))
sapply(misc_atomic, typeof)
## [1] "logical"    "integer"    "double"     "character"   "complex"    "raw"
```

Die Typen `complex`, `raw` treten eher selten auf und werden in der Vorlesung nicht näher behandelt.

`typeof()` gibt den entsprechenden Basistyp aus. `class()` gibt für atomare Vektoren fast das Gleiche aus aber anstatt "double" den Wert "numeric".

```
sapply(misc_atomic, class)
## [1] "logical"    "integer"    "numeric"    "character"   "complex"    "raw"
```

Die Funktionen `is.logical()`, `is.integer()`, `is.double()`, `is.character()` testen den Typ eines atomaren Vektors. `is.numeric()` ist für den Test auf `integer` oder `double` geeignet. `is.atomic()`, `is.list()`, `is.vector()` testen entsprechend, ob ihr Argument atomar, eine Liste bzw. ein Vektor ist.

```
# wende is.-Funktionen auf verschiedene Objekte an
is_type <- function(x) c(is.logical(x), is.integer(x), is.double(x), is.numeric(x),
                        is.character(x), is.atomic(x), is.list(x), is.vector(x))
misc <- c(misc_atomic, list(list(), matrix(1:4, ncol=2), NULL))
res <- sapply(misc, is_type)
rownames(res) <- paste("is.", c("logi", "inte", "doub", "nume", "char", "atom",
                            "list", "vect"), sep="")
```

```

colnames(res) <- sapply(misc, function(x) class(x)[1])
print(res)
## logical integer numeric character complex raw list matrix NULL
## is.logi TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## is.inte FALSE TRUE FALSE FALSE FALSE FALSE FALSE FALSE TRUE FALSE
## is.doub FALSE FALSE TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## is.nume FALSE TRUE TRUE FALSE FALSE FALSE FALSE FALSE TRUE FALSE
## is.char FALSE FALSE FALSE TRUE FALSE FALSE FALSE FALSE FALSE FALSE
## is.atom TRUE TRUE TRUE TRUE TRUE TRUE FALSE TRUE TRUE TRUE
## is.list FALSE FALSE FALSE FALSE FALSE FALSE TRUE TRUE FALSE FALSE
## is.vect TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE FALSE FALSE

```

NULL ist atomar ist insofern, dass es keine Teilelemente haben kann, die nicht NULL sind. Allerdings ist NULL kein Vektor. Der Zusammenhang zwischen Matrizen und Vektoren wird später genauer erklärt.

Wir können atomare Vektoren der Länge n mittels Funktionen der Form `BASE_TYPE(n)` erzeugen. Diese Vektoren sind mit den entsprechenden Default-Werten gefüllt.

```

# Die Funktion str() gibt die "Struktur" ihres Argumentes aus. Diese
# Ausgabe ist oft informativer als print().
str(list(logical(3), integer(3), double(3), character(3)))
## List of 4
## $ : logi [1:3] FALSE FALSE FALSE
## $ : int [1:3] 0 0 0
## $ : num [1:3] 0 0 0
## $ : chr [1:3] "" "" ""
# Achtung: str() kürzt 'double' mit 'num' ab...

```

Skalare sind atomare Vektoren der Länge 1. In R gibt es keine eigenen Typen oder Datenstrukturen nur für Skalare (anders als zB in C)

```

is.atomic(0)
## [1] TRUE
length(0)
## [1] 1
0[1]
## [1] 0

```

Greifen wir auf Vektoren an einer Stelle zu, die keinen definierten Wert hat, so wird NA zurückgegeben. Wir können Zuweisungen an Indizes größer der Länge des Vektors durchführen.

```

x <- 1:3
length(x)
## [1] 3
x[2:6]
## [1] 2 3 NA NA NA
x[5] <- 42
x
## [1] 1 2 3 NA 42
length(x)
## [1] 5

```

### 2.1.1 logical

Mögliche Werte eines Skalars vom Typ `logical` sind `TRUE`, `FALSE`, `NA` (not available). `T`, `F` dienen als Abkürzungen für `TRUE` und `FALSE`, sind aber keine reservierten Wörter.

```

T
## [1] TRUE
T <- FALSE # evil
T
## [1] FALSE
T <- TRUE # undo
typeof(NA)
## [1] "logical"

```

### 2.1.2 integer

integer-Werte enthalten eine endliche Teilmenge der ganzen Zahlen und `NA_integer_`. Sie werden mit dem Suffix L notiert, zB 42L. Bei der Ausgabe auf der Konsole wird das Suffix nicht ausgegeben.

Eine der wenigen Operatoren, die standardmäßig den Typ `integer` ausgeben ist :

```

sapply(list(42, 42L, NA_integer_, 1:3, 2L*2L, 2L^2L), typeof)
## [1] "double" "integer" "integer" "integer" "integer" "double"
.Machine$integer.max # maximaler integer-Wert
## [1] 2147483647
as.integer(.Machine$integer.max+1)
## Warning: NAs introduced by coercion to integer range
## [1] NA

```

### 2.1.3 double

double-Werte enthalten eine endliche Teilmenge der reellen Zahlen und `NA_real_` (*not available*), `Inf` (unendlich), `-Inf`, `NaN` (*not a number*). Zahlen ohne Suffix L werden als `double` interpretiert (zB 1, 2.3). Double-Werte können mit der E-Notation (zB `12e-34`) eingegeben werden.

```

lst <- list(0, pi, 1e3, Inf, NaN, NA_real_)
sapply(lst, typeof)
## [1] "double" "double" "double" "double" "double" "double"
unlist(lst)
## [1] 0.000000 3.141593 1000.000000 Inf NaN NA

```

Beachte Maschinengenauigkeit beim Rechnen mit `double`-Werten.

```

sqrt(2)^2 == 2
## [1] FALSE
sqrt(2)^2 - 2
## [1] 4.440892e-16

```

double-Werte sind im IEEE 754 Standard gespeichert. Daraus ergeben sich die Genauigkeit sowie die maximalen und minimalen Werte. Diese sind in der Liste `.Machine` angegeben. Dabei bezeichnet `eps` den Abstand zwischen 1 und der nächst größeren darstellbaren Zahl.

```

str(.Machine[1:13])
## List of 13
## $ double.eps : num 2.22e-16
## $ double.neg.eps : num 1.11e-16
## $ double.xmin : num 2.23e-308
## $ double.xmax : num 1.8e+308
## $ double.base : int 2
## $ double.digits : int 53
## $ double.rounding : int 5
## $ double.guard : int 0

```

```

## $double.ulp.digits : int -52
## $double.neg.ulp.digits: int -53
## $double.exponent : int 11
## $double.min.exp : int -1022
## $double.max.exp : int 1024
1 == 1 + .Machine$double.eps
## [1] FALSE
1 == 1 + .Machine$double.eps/2
## [1] TRUE
.Machine$double.xmax
## [1] 1.797693e+308
.Machine$double.xmax*2
## [1] Inf

```

Statt double-Vektoren auf exakte Gleichheit zu testen, prüft man meist, ob der Betrag der Differenz sehr klein ist.

```

abs(c(sin(pi), sqrt(2)^2) - c(0, 2)) < 1e-14
## [1] TRUE TRUE
# Achtung: Die Größenordnung des Fehlers ändert sich durch manche Operationen
sqrt(sqrt(2)^2-2)
## [1] 2.107342e-08

```

Bei der Ausgabe auf der Konsole werden standardmäßig 7 geltende Ziffern ausgegeben. Es können aber 15 bis 16 Dezimalstellen gespeichert und berechnet werden.

```

98765432109876543210
## [1] 9.876543e+19
98765432109876543210 - 98765432100000000000 # 15 Ziffern korrekt
## [1] 9876537344
1 + 1e-20 # Ausgabe als 1
## [1] 1
1 + 1e-20 - 1 # Rechnen mit 1 statt 1e-20
## [1] 0
1 + 1e-10 # Ausgabe als 1 (7 geltende Ziffern)
## [1] 1
1 + 1e-10 - 1 # Rechnen mit 1 + 1e-10
## [1] 1e-10

```

Mit Inf, -Inf, NaN und NA\_real\_ kann gerechnet werden.

```

1 / 0
## [1] Inf
-1 / 0
## [1] -Inf
0/0
## [1] NaN
log(0)
## [1] -Inf
1 / Inf
## [1] 0
Inf - Inf
## [1] NaN
Inf + Inf
## [1] Inf
Inf / Inf

```

```

## [1] NaN
-Inf * Inf
## [1] -Inf
1 + NaN
## [1] NaN
NaN - NaN
## [1] NaN
1 + NA_real_
## [1] NA
NA_real_ - NA_real_
## [1] NA
NA_real_ - NaN
## [1] NA
NaN - NA_real_
## [1] NaN

```

## 2.1.4 character

character-Werte (Strings) sind Text, notiert in einfachen oder doppelten Anführungszeichen, zB "blub" oder 'bla'. Der Text kann **Escape-Sequenzen** enthalten zB neue Zeile: \n, backslash: \\, Anführungszeichen ", ', Unicode \U....

```

"einfache ' Anführungszeichen"
## [1] "einfache ' Anführungszeichen"
'doppelte " Anführungszeichen'
## [1] "doppelte \" Anführungszeichen"
str <- "1. Zeile\n\"2. Zeile\\", backslash \\
print(str)
## [1] "1. Zeile\n\"2. Zeile\\", backslash \\
cat(str)
## 1. Zeile
## "2. Zeile", backslash \
x <- c("ä", "Hallo", "", NA_character_, "\n")
typeof(x)
## [1] "character"
length(x)
## [1] 5
nchar(x)
## [1] 1 5 0 NA 1

```

Ab R Version 4 können **raw** Strings, dh ohne Interpretation von Escape-Sequenzen, mit **r"(...)"** angegeben werden, wobei ... eine Sequenz von Symbolen ist, die nicht die Sequenz )" enthält.

```

cat(r"(n\\\"\\\")")
## \n\\\"\\"

```

In R gibt es 4 **character**-Vektoren, die direkt verfügbar sind.

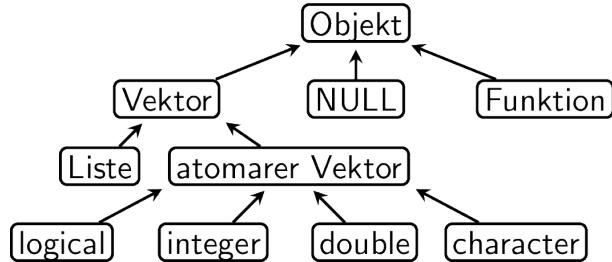
```

letters
## [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l" "m" "n" "o" "p" "q" "r" "s"
## [20] "t" "u" "v" "w" "x" "y" "z"
LETTERS
## [1] "A" "B" "C" "D" "E" "F" "G" "H" "I" "J" "K" "L" "M" "N" "O" "P" "Q" "R" "S"
## [20] "T" "U" "V" "W" "X" "Y" "Z"
month.name
## [1] "January"   "February"   "March"      "April"       "May"        "June"

```

```
## [7] "July"      "August"     "September" "October"    "November"   "December"
month.abb
## [1] "Jan" "Feb" "Mar" "Apr" "May" "Jun" "Jul" "Aug" "Sep" "Oct" "Nov" "Dec"
```

## 2.1.5 Übersicht



Typ	Klasse	not available	erzeugen	Default
logical	logical	NA	logical(n)	FALSE
integer	integer	NA_integer_	integer(n)	0L
double	numeric	NA_real_	double(n)	0
character	character	NA_character_	character(n)	""

Achtung: Alle NA-Werte werden in der Konsole als NA ausgegeben.

```
NA
## [1] NA
NA_integer_
## [1] NA
NA_real_
## [1] NA
NA_character_
## [1] NA
```

## 2.2 Combine

`c()` (`combine`) kombiniert Vektoren (gleichen Typs) *flach*: `c(x, c(y, z))` gleich `c(c(x, y), z)` gleich `c(x, y, z)` (Assoziativität).

```
c(1, c(2, 3:4))
## [1] 1 2 3 4
c(c(1, 2), 3:4)
## [1] 1 2 3 4
vec <- c(c(list(1), list(T)), list("asdf", list()))
str(vec)
## List of 4
## $ : num 1
## $ : logi TRUE
## $ : chr "asdf"
## $ : list()
vec <- c(list(1), list(T), list("asdf", list()))
str(vec)
## List of 4
## $ : num 1
## $ : logi TRUE
```

```
## $ : chr "asdf"
## $ : list()
```

NULL ist ein “neutrales Element” von `c()`.

```
c(1:3, NULL, 11:13)
## [1] 1 2 3 11 12 13
str(c(NULL, list(1), NULL, list(T), NULL))
## List of 2
## $ : num 1
## $ : logi TRUE
```

## 2.3 Typenumwandlung

Umwandlung eines Objektes von einem Typ in einen anderen wird in R mit **Coercion** (dt: Zwang) bezeichnet. Typenumwandlung kann explizit gefordert werden oder implizit geschehen. (Bei anderen Programmiersprachen ist das Wort Coercion zT nur für implizite Typenumwandlung reserviert).

Explizit kann Typenumwandlung durch `as.logical()`, `as.integer()`, `as.double()`, `as.character()` durchgeführt werden.

```
x <- as.double(c("1.2", "3.4", "1e-2"))
x
## [1] 1.20 3.40 0.01
sum(x)
## [1] 4.61
```

Bei der Nutzung von `c()` kann es zu impliziter Coercion kommen. Es werden ggf *speziellere* Typen in *allgemeinere* umgewandelt. Von speziell nach allgemein ist die Typen-Ordnung: `logical`, `integer`, `double`, `complex`, `character`, `raw`

```
c(TRUE, 15)
## [1] 1 15
c(42, FALSE)
## [1] 42 0
typeof(c(1L, 2))
## [1] "double"
c(TRUE, 1L, 1, "eins")
## [1] "TRUE" "1"    "1"    "eins"
c(c(TRUE, 1L), 1, "eins") # Assoziativität durch Coercion gebrochen
## [1] "1"    "1"    "1"    "eins"
```

Auch viele Funktionen wandeln ggf die Typen ihrer Argumente um, zB kann die Anzahl gerader Elemente im atomaren Vektor `x` durch `sum(x %% 2 == 0)` berechnet werden.

```
sum(c(T,F,F,T,T))
## [1] 3
x <- 1:100
sum(x %% 7 == 0) # Anzahl Vielfacher von 7 in x
## [1] 14
```

## 2.4 Listen

Mit einer Liste können mehrere Objekte zu einer Einheit zusammengefasst werden. Möchte man in einer Funktion mehr als ein Objekt zurückgeben, können diese zunächst in eine Liste geschrieben und dann die Liste zurückgegeben werden.

Die Elemente von Listen können verschiedene, beliebige Typen haben.

```
lst <- list(  
  1:3,  
  "a",  
  c(TRUE, FALSE, TRUE),  
  c(2.3, 5.9),  
  list(1,"b"),  
  sum # Funktion  
)  
str(lst)  
## List of 6  
## $ : int [1:3] 1 2 3  
## $ : chr "a"  
## $ : logi [1:3] TRUE FALSE TRUE  
## $ : num [1:2] 2.3 5.9  
## $ :List of 2  
## ..$ : num 1  
## ..$ : chr "b"  
## $ :function (..., na.rm = FALSE)
```

Listen selbst haben Typ und Klasse `list`. Dies kann mit `is.list()` getestet werden.

```
c(typeof(lst), class(lst), length(lst))  
## [1] "list" "list" "6"  
c(is.list(lst), is.list(1:3))  
## [1] TRUE FALSE
```

Listen sind Vektoren.

```
is.vector(lst)  
## [1] TRUE
```

Der Zugriff auf Elemente erfolgt mit `[[`. Mit `[` werden Unterlisten erzeugt.

```
str(lst[c(2,4)])  
## List of 2  
## $ : chr "a"  
## $ : num [1:2] 2.3 5.9  
str(lst[2]) # Liste!  
## List of 1  
## $ : chr "a"  
str(lst[[2]]) # Element der Liste  
## chr "a"
```

Zuweisung von `NULL` löscht Listenelemente.

```
lst[c(2,5,6)] <- NULL  
str(lst)  
## List of 3  
## $ : int [1:3] 1 2 3  
## $ : logi [1:3] TRUE FALSE TRUE  
## $ : num [1:2] 2.3 5.9  
lst[1] <- list(NULL) # NULL als Listenelement  
str(lst)  
## List of 3  
## $ : NULL  
## $ : logi [1:3] TRUE FALSE TRUE
```

```
## $ : num [1:2] 2.3 5.9
```

Die Funktion `list()` erstellt eine Liste ihrer Argumente, wobei `list()` die leere Liste erzeugt. Falls die Argumente von `c()` Listen sind, werden deren Elemente zu einer Liste zusammengefügt.

```
list()
## list()
str(c(list(), list(1), list(c(T, F), letters[1:3])))
## List of 3
## $ : num 1
## $ : logi [1:2] TRUE FALSE
## $ : chr [1:3] "a" "b" "c"
```

Umwandlungen zwischen Listen und atomaren Vektoren sind mit `unlist()` und `as.list()` möglich.

```
str(list(1:3))
## List of 1
## $ : int [1:3] 1 2 3
str(as.list(1:3))
## List of 3
## $ : int 1
## $ : int 2
## $ : int 3
lst <- list(1:3, as.list(4:6))
str(lst)
## List of 2
## $ : int [1:3] 1 2 3
## $ :List of 3
##   ..$ : int 4
##   ..$ : int 5
##   ..$ : int 6
str(unlist(lst))
##  int [1:6] 1 2 3 4 5 6
```

### 3 Attribute

Attribute sind Meta-Daten von Objekten.

Die Attribute eines Objektes sind eine ungeordnete Sammlung aus (Name, Wert)-Paaren, wobei Namen maximal einmal vorkommen dürfen. Der Wert kann ein beliebiges Objekt außer `NULL` sein.

Jedes Objekt außer `NULL` kann Attribute haben.

Attribute können mit `attr()`, `attributes()`, `structure()`, `str()` abgefragt bzw gesetzt werden (siehe `?`). `attr(obj, "Name") <- Wert` setzt für das Objekt `obj` das Attribut namens `Name` auf den Wert `Wert`.

```
x <- 1:5
str(x)
##  int [1:5] 1 2 3 4 5
attr(x, "asdf") <- c(1, 7)
str(x)
##  int [1:5] 1 2 3 4 5
##  - attr(*, "asdf")= num [1:2] 1 7
attr(x, "blub") <- list("Hallo", c(T,F,T))
str(x)
##  int [1:5] 1 2 3 4 5
```

```

## - attr(*, "asdf")= num [1:2] 1 7
## - attr(*, "blub")=List of 2
##   ..$ : chr "Hallo"
##   ..$ : logi [1:3] TRUE FALSE TRUE
typeof(attributes(x))
## [1] "list"
str(attributes(x))
## List of 2
## $ asdf: num [1:2] 1 7
## $ blub:List of 2
##   ..$ : chr "Hallo"
##   ..$ : logi [1:3] TRUE FALSE TRUE

y <- NULL
attr(y, "a") <- "b" # ERROR
## Error in attr(y, "a") <- "b": attempt to set an attribute on NULL

```

### 3.1 Namen

Das Attribut `names` wird genutzt, um Einträgen eines Vektors Namen zu geben.

Dies kann direkt beim Erstellen des Vektors mit `c()` oder `list()` geschehen oder im Nachhinein mittels `attr(, "names") <-`, oder `names() <-` durchgeführt werden. Bei `c()` und `list()` kann der Name in Anführungszeichen gesetzt werden, bei speziellen Namen muss dies sogar gemacht werden. Sind Namen gesetzt, werden sie bei der Ausgabe auf der Konsole angezeigt.

```

x <- 1:4
attr(x, "names") <- letters[1:4]
x
## a b c d
## 1 2 3 4
str(x)
## Named int [1:4] 1 2 3 4
## - attr(*, "names")= chr [1:4] "a" "b" "c" "d"
names(x) <- LETTERS[1:4]
x
## A B C D
## 1 2 3 4

y <- c(eins=1, zwei=2, drei=3)
y
## eins zwei drei
## 1 2 3

# z <- c(a=1, 42=3) # ERROR
z <- c("a"=1, "42"=3, "!"=24) # works
z
## a 42 "
## 1 3 24

lst <- list(wort="asdf", Zahlen=1:3)
lst
## $wort
## [1] "asdf"

```

```
##  
## $Zahlen  
## [1] 1 2 3  
str(lst)  
## List of 2  
## $ wort : chr "asdf"  
## $ Zahlen: int [1:3] 1 2 3
```

Mittels \$ kann auf benannte Elemente von Listen (nicht von atomaren Vektoren) ohne den Index zugegriffen werden oder neue Elemente erzeugt werden.

```
lst$wort  
## [1] "asdf"  
lst$Zahlen  
## [1] 1 2 3  
lst$logisch  
## NULL  
lst$logisch <- TRUE  
lst$"0" <- 123  
str(lst)  
## List of 4  
## $ wort : chr "asdf"  
## $ Zahlen : int [1:3] 1 2 3  
## $ logisch: logi TRUE  
## $ 0 : num 123
```

Mit attr(x, "names") oder names(x) werden alle Namen angezeigt.

```
attr(x, "names")  
## [1] "A" "B" "C" "D"  
names(y)  
## [1] "eins" "zwei" "drei"
```

Um Namen zu entfernen, nutzen wir unname(), names(x) <- NULL oder attr(x, "names") <- NULL.

```
unname(y)  
## [1] 1 2 3  
y # wurde nicht geändert, call by value!  
## eins zwei drei  
## 1 2 3  
y <- unname(y)  
y  
## [1] 1 2 3  
  
str(x)  
## Named int [1:4] 1 2 3 4  
## - attr(*, "names")= chr [1:4] "A" "B" "C" "D"  
names(x) <- NULL  
str(x)  
## int [1:4] 1 2 3 4
```

## 3.2 Weitere Bemerkungen

Viele Operationen erhalten die Attribute ihrer Inputs nicht. Die einzigen Attribute, die meist erhalten bleiben, sind dim und names.

```

z <- structure(1:5, names=letters[1:5], something="some thing")
str(z)
##  Named int [1:5] 1 2 3 4 5
##  - attr(*, "names")= chr [1:5] "a" "b" "c" "d" ...
##  - attr(*, "something")= chr "some thing"
attributes(z[1])
## $names
## [1] "a"
attributes(sum(z))
## NULL
attributes(z) <- NULL # remove all attributes
z
## [1] 1 2 3 4 5

```

Das Attribut `dim` erlaubt es, aus Vektoren Matrizen und Arrays zu machen. Mehr dazu gleich.

Weitere Attribute ermöglichen neue Datenstrukturen auf Vektoren aufzubauen (zB Tibbles, Faktoren, Datenstrukturen für Zeit und Datum, ...). Später mehr dazu.

```

attributes(tibble::tibble(a=c(7,8), b=c(T,F)))
## $names
## [1] "a" "b"
##
## $row.names
## [1] 1 2
##
## $class
## [1] "tbl_df"     "tbl"        "data.frame"

```

Names erleichtern den Zugriff auf Elemente einer Liste und machen Code leichter verständlich: `fit$coeff` vs `fit[[1]]`. Attribute sollten jedoch nicht zum Speichern von Daten/Beobachtungen dienen (nur für Meta-Daten).

```

age <- c(Alice=25, Bob=23) # schlechter Stil, da Namen Teil der Beobachtung sind
age
## Alice   Bob
## 25     23
tibble::tibble(name=c("Alice", "Bob"), age=c(25,23)) # besser
## # A tibble: 2 x 2
##   name    age
##   <chr> <dbl>
## 1 Alice    25
## 2 Bob     23

```

Die Funktion `identical()` überprüft, ob Objekte exakt gleich (identisch) sind. Dies schließt Gleichheit von Attributen mit ein.

```

x <- 1:3
attr(x, "asdf") <- 42
x == 1:3
## [1] TRUE TRUE TRUE
identical(x, 1:3)
## [1] FALSE
# identical kann auch zum Testen von NA-Werten genutzt werden:
identical(NA, NA)
## [1] TRUE

```

```
identical(NA, NaN)
## [1] FALSE
NULL == NULL
## logical(0)
identical(NULL, NULL)
## [1] TRUE
```

## 4 Matrizen und Arrays

Um aus einem Vektor eine **Matrix** zu machen, setzen wir das Attribut **dim** (**Dimension**) auf den atomaren Vektor aus Zeilen- und Spaltenzahl.

```
x <- 1:6
class(x)
## [1] "integer"
attr(x, "dim") <- c(2,3)
class(x)
## [1] "matrix" "array"
x
##      [,1] [,2] [,3]
## [1,]    1    3    5
## [2,]    2    4    6
identical(x, matrix(1:6, nrow=2)) # exakt gleich
## [1] TRUE
```

Matrizen haben 2 Dimensionen. Die Verallgemeinerung auf  $n$  Dimensionen wird in R als **Array** bezeichnet. Seit R Version 4 sind Matrizen auch Arrays.

```
x1 <- structure(1:24, dim=24) # 1D-Array
x1
##  [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24
class(x1)
## [1] "array"
x2 <- structure(1:24, dim=c(4,6)) # Matrix (2D-Array)
x2
##      [,1] [,2] [,3] [,4] [,5] [,6]
## [1,]    1    5    9   13   17   21
## [2,]    2    6   10   14   18   22
## [3,]    3    7   11   15   19   23
## [4,]    4    8   12   16   20   24
class(x2)
## [1] "matrix" "array"
x3 <- structure(1:24, dim=2:4) # 3D-Array
x3
## , , 1
##
##      [,1] [,2] [,3]
## [1,]    1    3    5
## [2,]    2    4    6
##
## , , 2
##
##      [,1] [,2] [,3]
## [1,]    7    9   11
```

```

## [2,]    8   10   12
##
## , , 3
##
##      [,1] [,2] [,3]
## [1,]    13   15   17
## [2,]    14   16   18
##
## , , 4
##
##      [,1] [,2] [,3]
## [1,]    19   21   23
## [2,]    20   22   24
class(x3)
## [1] "array"
x4 <- structure(1:24, dim=1:4) # 4D-Array
x4
## , , 1, 1
##
##      [,1] [,2]
## [1,]    1    2
##
## , , 2, 1
##
##      [,1] [,2]
## [1,]    3    4
##
## , , 3, 1
##
##      [,1] [,2]
## [1,]    5    6
##
## , , 1, 2
##
##      [,1] [,2]
## [1,]    7    8
##
## , , 2, 2
##
##      [,1] [,2]
## [1,]    9   10
##
## , , 3, 2
##
##      [,1] [,2]
## [1,]   11   12
##
## , , 1, 3
##
##      [,1] [,2]
## [1,]   13   14
##
## , , 2, 3

```

```

## [,1] [,2]
## [1,] 15 16
##
## , , 3, 3
##
## [,1] [,2]
## [1,] 17 18
##
## , , 1, 4
##
## [,1] [,2]
## [1,] 19 20
##
## , , 2, 4
##
## [,1] [,2]
## [1,] 21 22
##
## , , 3, 4
##
## [,1] [,2]
## [1,] 23 24
class(x4)
## [1] "array"

```

Wir können `dim` setzen mittels `attr(, "dim") <-`, `dim() <-` oder `array()`. Den Wert von `dim` erhalten wir mit `attr(, "dim")` oder `dim()`.

Ein Array `x` mit `length(dim(x))` gleich 2 ist eine Matrix. Matrizen können wir auch mit `matrix()` erzeugen.

```

x
## [,1] [,2] [,3]
## [1,] 1 3 5
## [2,] 2 4 6
dim(x)
## [1] 2 3
dim(x) <- c(3,2)
x
## [,1] [,2]
## [1,] 1 4
## [2,] 2 5
## [3,] 3 6

```

Der Wert von `dim` ist ein `integer`-Vektor der Länge  $\geq 1$ , sodass `prod(dim(x))` (Produkt aller Elemente) gleich `length(x)` ist.

Die Einträge des zugrunde liegenden Vektors werden in einer Rechteckstruktur (bei Matrizen) bzw "Hyperrechteckstruktur" (bei Arrays) angeordnet. Für Matrizen geschieht dies in der *column-major order* (außer man setzt das Argument `byrow` der Funktion `matrix` entsprechend).

```

A <- matrix(1:9, nrow=3)
A
## [,1] [,2] [,3]
## [1,] 1 4 7
## [2,] 2 5 8

```

```

## [3,] 3 6 9
dim(A) <- NULL # entferne Dimensions-Attribut
A # Der Matrix A lag der Vektor 1:9 zugrunde
## [1] 1 2 3 4 5 6 7 8 9
B <- matrix(1:9, nrow=3, byrow=T)
B
## [,1] [,2] [,3]
## [1,] 1 2 3
## [2,] 4 5 6
## [3,] 7 8 9
dim(B) <- NULL
B # Beim Erstellen der Matrix B wurde 1:9 umgeordnet
## [1] 1 4 7 2 5 8 3 6 9

```

Arrays können sowohl auf einem atomaren Vektor (beliebigen Basis-Typs) als auch auf einer Liste aufbauen (Die mit Abstand häufigste Anwendung sind jedoch Matrizen aufbauend auf atomaren Vektoren vom Typ `double`, `integer` oder `logical`.)

```

lst <- list("a", 1, T, list())
matrix(lst, ncol=2)
## [,1] [,2]
## [1,] "a"  TRUE
## [2,] 1    List,0

```

Die Ausgabe von `typeof()` für ein Array entspricht der Ausgabe für den zugrunde liegenden Vektor.

```

mat_types <- list(
  matrix(c(T, F, T, F), ncol=2),
  matrix(1:4, ncol=2),
  matrix(as.double(1:4), ncol=2),
  matrix(letters[1:4], ncol=2),
  matrix(as.list(1:4), ncol=2)
)
sapply(mat_types, typeof)
## [1] "logical"   "integer"    "double"     "character"  "list"

```

Nutze `is.array()`, `is.matrix()` oder `class()` um herauszufinden, ob ein Objekt eine Matrix oder ein Array ist.

```

classify_array <- function(x) c(is.atomic(x), is.list(x), is.vector(x), is.matrix(x), is.array(x))
misc <- list(1:4, matrix(c(T,T), nrow=2), matrix(lst, ncol=2), array(1:4, dim = c(2,2,1)))
res <- sapply(misc, classify_array)
rownames(res) <- paste("is.",c("atom", "list", "vect", "matr", "arry"), sep="")
colnames(res) <- sapply(misc, function(x) paste(class(x)[1], typeof(x)))
print(res)
##           integer integer matrix logical matrix list array integer
## is.atom      TRUE      TRUE     FALSE      TRUE
## is.list      FALSE     FALSE     TRUE      FALSE
## is.vect      TRUE     FALSE     FALSE      FALSE
## is.matr     FALSE      TRUE     TRUE      FALSE
## is.arry     FALSE      TRUE     TRUE      TRUE

```

Die Zeilen und Spalten (alle Dimensionen) von Matrizen (Arrays) können benannt werden. Dies geschieht mittels des Attributes `dimnames`. Der Wert von `dimnames` ist eine Liste (ggf mit benannten Werten) der selben Länge wie `dim`, deren Einträge `character`-Vektoren sind. Abkürzungen für `attr(, "dimnames")` sind `dimnames()` bzw `rownames()`, `colnames()` für die ersten beiden Einträge der Liste.

```

x <- matrix(1:4, ncol=2)
rownames(x) <- letters[1:2]
colnames(x) <- letters[11:12]
x
##   k l
## a 1 3
## b 2 4
str(attributes(x))
## List of 2
##  $ dim      : int [1:2] 2 2
##  $ dimnames:List of 2
##    ..$ : chr [1:2] "a" "b"
##    ..$ : chr [1:2] "k" "l"
dimnames(x) <- list("Zeilen" = c("z1", "z2"), "Spalten" = c("s1", "s2"))
x
##      Spalten
## Zeilen s1 s2
##      z1  1  3
##      z2  2  4

```

Weitere nützliche Funktionen im Zusammenhang mit Matrizen und Arrays sind (siehe ?):

- nrow(), ncol(), NROW(), NCOL()

```

y <- cbind(1:3, 7:9)
y
##      [,1] [,2]
## [1,]    1    7
## [2,]    2    8
## [3,]    3    9
dim(y)
## [1] 3 2
c(ncol(y), NCOL(y), nrow(y), NROW(y), length(y)) # hier sind nrow, NROW gleich
## [1] 2 2 3 3 6
z <- 1:3
nrow(z)
## NULL
ncol(z)
## NULL
c(NROW(z), NCOL(z)) # hier unterschiedlich
## [1] 3 1

```

- transponierte Matrix t(), „transponieren“ für Arrays aperm()
- Matrizen und Arrays verbinden: rbind(), cbind(), abind::abind()

```

x <- cbind(1:3)
x
##      [,1]
## [1,]    1
## [2,]    2
## [3,]    3
x <- cbind(x, 11:13)
x
##      [,1] [,2]
## [1,]    1    11

```

```

## [2,]    2   12
## [3,]    3   13
y <- t(x)
y
##      [,1] [,2] [,3]
## [1,]    1    2    3
## [2,]   11   12   13
abind::abind(y, y+500, along=3)
## , , 1
##
##      [,1] [,2] [,3]
## [1,]    1    2    3
## [2,]   11   12   13
##
## , , 2
##
##      [,1] [,2] [,3]
## [1,]  501  502  503
## [2,]  511  512  513

```

- Matrixoperationen: `%%*` (Matrixmultiplikation), `det()` (Determinante), `solve()` (Lineares Gleichungssystem lösen und Matrix invertieren), `qr()` (QR-Zerlegung), `chol()` (Cholesky-Zerlegung), `svd()` (Singulärwertzerlegung), `eigen()` (Eigenwerte und -vektoren)

Bemerkung: Einige Attribute werden beim Setzen auf Korrektheit (zB richtiger Typ) getestet (`class`, `comment`, `dim`, `dimnames`, `names`, `row.names`, `tsp`; siehe `?attr`). Die meisten Attribute werden jedoch nicht geprüft.

```

x <- 1:3
attr(x, "dim") <- 4 # ERROR
## Error in attr(x, "dim") <- 4: dims [product 4] do not match the length of object [3]
attr(x, "names") <- letters # ERROR
## Error in attr(x, "names") <- letters: 'names' attribute [26] must be the same length as the vector [3]

```

## 5 S3 Objekte

Durch das Setzen des `class`-Attributs machen wir Objekte zu sogenannten **S3 Objekten**. Diese werden im Kapitel über objektorientierte Programmierung ausführlicher behandelt. Hier gibt es einen Überblick.

Beispiele für S3 Klassen sind `tibble`, `factor`, `Date`, `POSIXct` (Zeit), (gleich ausführlicher).

Einige Funktionen (`print()`, `plot()`, `summary()`) behandeln Objekte je nach Klasse unterschiedlich.

Ist `class` gesetzt, wird der Wert dieses Attributes von `class()` zurückgegeben (**explizite Klasse**).

```

attr(x, "class") <- "Spitze"
class(x)
## [1] "Spitze"

```

Ist `class` nicht gesetzt, gibt `class()` die **implizite Klasse** aus: "matrix", "array", "function", "numeric" oder den Wert von `typeof(x)`.

```

type_and_class <- function(x) c(typeof(x), class(x)[1], {y <- attr(x, "class")[1]; if(is.null(y)) "NULL" else y})
misc <- c(misc_atomic, list(list(), matrix(1:4, ncol=2), array(1, dim=1), sum, mean,
NULL, structure(1:3, class="asdf"), factor(1:3)))
res <- sapply(misc, type_and_class)
rownames(res) <- c("typeof()", "class()", "attr class")

```

```

colnames(res) <- c("logi", "inte", "doub", "char", "comp", "raw", "list", "matr",
                  "arra", "sum", "mean", "NULL", "S3", "factor")
print(res)
##          logi      inte      doub      char      comp      raw      list
##  typeof() "logical" "integer" "double" "character" "complex" "raw"    "list"
##  class()  "logical" "integer" "numeric" "character" "complex" "raw"    "list"
##  attr class "NULL"   "NULL"   "NULL"   "NULL"   "NULL"   "NULL"   "NULL"
##          matr      arra      sum      mean      NULL     S3      factor
##  typeof() "integer" "double" "builtin" "closure" "NULL"   "integer" "integer"
##  class()  "matrix"  "array"  "function" "function" "NULL"   "asdf"   "factor"
##  attr class "NULL"   "NULL"   "NULL"   "NULL"   "NULL"   "asdf"   "factor"

```

Das Klassen-Attribut kann mit `unclass()` oder `attr(, "class") <- NULL` entfernt werden.

```

str(x)
## 'Spitze' int [1:3] 1 2 3
x <- unclass(x)
str(x)
## int [1:3] 1 2 3

```

## 5.1 Faktoren

Faktoren sind `integer`-Vektoren mit `attr(x, "class")` gleich "factor" und dem Attribut `levels`.

Sie speichern kategorische Daten. Die Menge der möglichen Werte gibt das Attribut `levels` an.

Die Funktion `factor()` erzeugt eine Faktor. `table()` listet wie oft welcher Wert auftritt.

```

x <- c("m", "f", "f", "f", "m", "f")
x_factor <- factor(x)
x_factor
## [1] m f f f m f
## Levels: f m
str(attributes(x_factor))
## List of 2
## $ levels: chr [1:2] "f" "m"
## $ class : chr "factor"
table(x_factor)
## x_factor
## f m
## 4 2
unclass(x_factor)
## [1] 2 1 1 1 2 1
## attr(,"levels")
## [1] "f" "m"

# create factor "by hand":
y <- rep(1:3,3)
y
## [1] 1 2 3 1 2 3 1 2 3
is.factor(y)
## [1] FALSE
attr(y, "class") <- "factor"
attr(y, "levels") <- letters[26:24]
is.factor(y)
## [1] TRUE

```

```

y
## [1] z y x z y x z y x
## Levels: z y x
table(y)
## y
## z y x
## 3 3 3

```

Siehe auch `?factor`.

## 5.2 Date

Objekte der Klasse `Date` speichern Kalenderdaten.

Sie sind `double`-Vektoren, deren `class`-Attribut den Wert "Date" hat.

`Sys.Date()` gibt ein `Date`-Objekt zurück, das das aktuelle Datum enthält.

```

Sys.Date()
## [1] "2021-04-17"
attributes(Sys.Date())
## $class
## [1] "Date"

```

Der zugrunde liegende `double`-Wert ist die Anzahl Tage seit 1.1.1970.

```

unclass(Sys.Date())
## [1] 18734
structure(c(0,1), class="Date")
## [1] "1970-01-01" "1970-01-02"

```

## 5.3 POSIXct

Möchte man Datum und Uhrzeit speichern, kann man auf die Klasse `POSIXct` (*Portable Operating System Interface*, `c()`, `time`) zurückgreifen.

Dies ist ein `double` Vektor, dessen `class`-Attribut den Wert `c("POSIXct", "POSIXt")` hat.

Dies bedeutet, dass `POSIXct` eine Unterklasse von `POSIXt` ist (später mehr).

`Sys.time()` gibt ein `POSIXct`-Objekt zurück, das das aktuelle Datum und die Uhrzeit enthält.

```

x <- Sys.time()
x
## [1] "2021-04-17 16:47:18 CEST"
attributes(x)
## $class
## [1] "POSIXct" "POSIXt"
unclass(x)
## [1] 1618670839

```

Der zugrunde liegende `double`-Wert ist die Anzahl Sekunden seit 1.1.1970

```

unclass(Sys.time())
## [1] 1618670839
structure(c(0,1), class=c("POSIXct", "POSIXt"))
## [1] "1970-01-01 01:00:00 CET" "1970-01-01 01:00:01 CET"

```

## 5.4 proc\_time

`proc.time()` gibt die Laufdauer des R-Prozesses als S3 Objekt der Klasse `proc_time` an.

```
x <- proc.time()
x
##    user  system elapsed
##    1.93    0.43   3.09
class(x)
## [1] "proc_time"
str(attributes(x))
## List of 2
##  $ names: chr [1:5] "user.self" "sys.self" "elapsed" "user.child" ...
##  $ class: chr "proc_time"
str(unclass(x))
##  Named num [1:5] 1.93 0.43 3.09 NA NA
##  - attr(*, "names")= chr [1:5] "user.self" "sys.self" "elapsed" "user.child" ...
```

Zugrunde liegt ein `double`-Vektor der Länge 5.

`user` gibt an, wie lange der R-Prozess die CPU beansprucht hat (in Sekunden); `system` ist die Dauer der Belastung der CPU durch das Betriebssystems im Auftrag des R-Prozesses (zB für Daten laden, speichern, Ausgabe); `elapsed` ist die gesamt vergangene Zeit.

Mit `proc_time`-Objekten kann man rechnen.

```
pt <- proc.time()
x <- 0
for (i in 1:1e7) x <- x + 1
proc.time() - pt
##    user  system elapsed
##    0.21    0.00    0.20
```

`system.time(expression)` misst die Rechenzeit zum Auswerten des Ausdrucks `expression` und gibt diese als `proc_time`-Objekt zurück.

```
tm <- system.time({
  data <- runif(1e6)
  stats <- c(mean(data), var(data))
})
class(tm)
## [1] "proc_time"
tm
##    user  system elapsed
##    0.04    0.00    0.03
```

## 6 Tibbles

**Tibbles** sind ebenfalls S3-Objekte (der Klasse `tbl_df`).

```
library(tibble)
tib1 <- tibble(x = 1:3, y = letters[3:1])
tib1
## # A tibble: 3 x 2
##       x     y
##   <int> <chr>
## 1     1     c
## 2     2     b
## 3     3     a
```

```

## 2      2 b
## 3      3 a
class(tib1)
## [1] "tbl_df"     "tbl"        "data.frame"
str(attributes(tib1))
## List of 3
##  $ names   : chr [1:2] "x" "y"
##  $ row.names: int [1:3] 1 2 3
##  $ class    : chr [1:3] "tbl_df" "tbl" "data.frame"

```

Tibbles basieren auf **Data-Frames**. Data-Frames sind in Base-R ohne das Laden von Paketen verfügbar.

```

df <- data.frame(x = 1:3, y = letters[3:1])
df
##   x y
## 1 1 c
## 2 2 b
## 3 3 a

```

Data-Frames (und damit Tibbles) sind S3-Objekte, welche auf Listen aufbauen. Jeder Eintrag der Liste muss die gleiche Länge haben.

```

typeof(tib1)
## [1] "list"
str(unclass(tib1))
## List of 2
##  $ x: int [1:3] 1 2 3
##  $ y: chr [1:3] "c" "b" "a"
##  - attr(*, "row.names")= int [1:3] 1 2 3
tib1$x # wie bei Liste
## [1] 1 2 3

```

Typischerweise werden Datensätze in Tibbles geladen und ausgehend von dieser Datenstruktur ausgewertet. Jede Zeile entspricht einer Beobachtung, jede Spalte einer beobachteten Variable / einem Feature.

Mit `nrow()` erhalten wir die Anzahl der Beobachtungen, mit `ncol()` Anzahl der Variablen. `length()` entspricht `ncol()`, da ein Tibble eine Liste seiner Spalten ist und `length()` die Länge der Liste ausgibt.

```

c(ncol(tib1), nrow(tib1), length(tib1))
## [1] 2 3 2

```

Ein Tibble kann Listen als Spalten haben.

```

lst <- list(T, 1:5, "asdf", 0, list(0))
tib2 <- tibble(asdf=LETTERS[1:5], Y=lst)
tib2 # Y ist Listen-Spalte
## # A tibble: 5 x 2
##   asdf   Y
##   <chr> <list>
## 1 A     <lg1 [1]>
## 2 B     <int [5]>
## 3 C     <chr [1]>
## 4 D     <dbl [1]>
## 5 E     <list [1]>

```

Ein Tibble kann Matrizen oder Tibbles als Spalten haben.

```

tib3 <- tibble(x=1:5)
tib3$y <- matrix(1:20, nrow=5)
tib3$z <- tib2
str(tib3)
## # tibble [5 x 3] (S3: tbl_df/tbl/data.frame)
## $ x: int [1:5] 1 2 3 4 5
## $ y: int [1:5, 1:4] 1 2 3 4 5 6 7 8 9 10 ...
## $ z: tibble [5 x 2] (S3: tbl_df/tbl/data.frame)
##   ..$ asdf: chr [1:5] "A" "B" "C" "D" ...
##   ..$ Y   :List of 5
##   ...$ : logi TRUE
##   ...$ : int [1:5] 1 2 3 4 5
##   ...$ : chr "asdf"
##   ...$ : num 0
##   ...$ :List of 1
##   ...$ : num 0
tib3
## # A tibble: 5 x 3
##       x     y[,1]   [,2]   [,3]   [,4] z$asdf $Y
##   <int> <int> <int> <int> <int> <chr> <list>
## 1     1      1      6     11     16 A     <lgsl [1]>
## 2     2      2      7     12     17 B     <int [5]>
## 3     3      3      8     13     18 C     <chr [1]>
## 4     4      4      9     14     19 D     <dbl [1]>
## 5     5      5     10     15     20 E     <list [1]>

```

Die Bedingung für Validität eines Tibbles ist, dass alle Einträge der zugrundeliegenden Liste auf einem Vektor-Typ basieren und die Ausgabe von `NROW()` übereinstimmt.

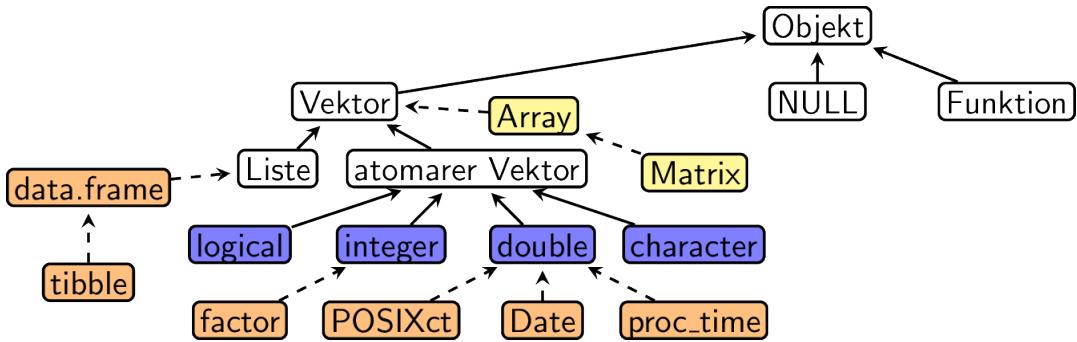
Um Konvertierungen zwischen Matrizen und Tibbles durchzuführen, kann `as_tibble()` bzw `as.matrix()` benutzt werden.

```

x <- matrix(1:12, ncol=3, dimnames=list(NULL, LETTERS[1:3]))
x
##      A B  C
## [1,] 1 5  9
## [2,] 2 6 10
## [3,] 3 7 11
## [4,] 4 8 12
tib <- as_tibble(x)
tib
## # A tibble: 4 x 3
##       A     B     C
##   <int> <int> <int>
## 1     1      5      9
## 2     2      6     10
## 3     3      7     11
## 4     4      8     12
as.matrix(tib)
##      A B  C
## [1,] 1 5  9
## [2,] 2 6 10
## [3,] 3 7 11
## [4,] 4 8 12

```

## 7 Überblick Datenstrukturen



## 8 Numerische Verwirrung

Wir listen ein paar Dinge, die im Umgang mit numerischen Typen Schwierigkeiten bereiten können.

Beachte die unterschiedlichen Typen von 0L vs 0, 1:1 vs 1, 2L\*2L vs 2L^2L.

```
sapply(list(0L, 0, 1:1, 1, 2L*2L, 2L^2L), typeof)
## [1] "integer" "double"  "integer" "double"  "integer" "double"
```

Der Begriff `numeric` wird inkonsistent verwendet.

In den Funktionen `class()`, `as.numeric()`, `numeric(n)`, `print()`, `str()` ist `numeric` synonym zu `double`.

```
c(class(1L), class(pi))
## [1] "integer" "numeric"
typeof(numeric(3))
## [1] "double"
double(0)
## numeric(0)
str(double(3))
## num [1:3] 0 0 0
c(typeof(numeric(3)), typeof(as.numeric(1L)))
## [1] "double" "double"
```

Die Funktion `is.numeric()` behandelt `numeric` als Überbegriff für `integer` und `double`.

```
c(is.numeric(1L), is.numeric(pi))
## [1] TRUE TRUE
```

Die *not available*-Wert vom Typ `double` heißt `NA_real_`

Welche der folgenden Ausdrücke ist wahr? `0 == 0L`, `identical(0, 0L)`, `sin(pi) == 0`

Auch `integer`-Werte leiden unter der Endlichkeit des Computers (*integer overflow*).

```
c(as.integer(2^30), as.integer(2^31), as.integer(2^30)*2L)
## Warning: NAs introduced by coercion to integer range
## Warning in as.integer(2^30) * 2L: NAs produced by integer overflow
## [1] 1073741824          NA          NA
```

## 9 Much Ado About Nothing

Es gibt verschiedene Objekte in R, die auf das Fehlen eines Wertes hinweisen.

NULL ist Rückgabewert von `c()` und hat Länge 0 und Typ NULL.

```
c()
## NULL
c(typeof(NULL), length(NULL))
## [1] "NULL" "0"
```

`BASE_TYPE(0)` ergibt einen atomaren Vektor der Länge 0 vom Typ `BASE_TYPE`.

```
c(typeof(integer(0)), length(integer(0)))
## [1] "integer" "0"
identical(NULL, integer(0))
## [1] FALSE
```

Die Werte `NA`, `NA_integer_`, `NA_real_`, `NA_character_` haben Länge 1 und den entsprechenden Basistyp.

```
c(length(NA), typeof(NA_integer_))
## [1] "1"      "integer"
```

`matrix(nrow=0, ncol=0)` und `array(dim=c(0))` haben Länge 0 und einen Basistyp (`logical`), aber im Unterschied zu `logical(0)` das Attribut `dim`.

```
m <- matrix(nrow=0, ncol=0)
c(typeof(m), length(m))
## [1] "logical" "0"
attributes(m)
## $dim
## [1] 0 0
identical(m, logical(0))
## [1] FALSE
```

Für einen atomaren Vektor sind `x[0]`, `x[NULL]` gleich `BASE_TYPE(0)`, wohingegen `x[]` dem ganzen Vektor `x` entspricht. `x[NA_integer_]`, `x[NA]` geben Vektoren mit NA-Werten zurück.

```
x <- 1:3
x[0]
## integer(0)
x[NULL]
## integer(0)
x[]
## [1] 1 2 3
identical(x, x[])
## [1] TRUE
x[NA_integer_]
## [1] NA
x[NA]
## [1] NA NA NA
```

`NaN` verhält sich größtenteils wie `NA_real_`. Die beiden Werte sind jedoch nicht identisch.

```
c(typeof(NaN), typeof(NA_real_))
## [1] "double" "double"
c(length(NaN), length(NA_real_))
## [1] 1 1
identical(NA_real_, NaN)
## [1] FALSE
```

Teste fehlende Werte mit `is.null()`, `is.nan()` oder `is.na()`.

```

classify_missing <- function(x) c(is.null(x), is.na(x), is.nan(x))
misc <- list(NA, NaN, 0, NA_character_, "")
res <- sapply(misc, classify_missing)
rownames(res) <- paste("is.", c("null", "na", "nan"), sep="")
colnames(res) <- c("NA", "NaN", "0", "NA_character_", "\\\"\\\"")
print(res)
##          NA    NaN      0 NA_character_      ""
## is.null FALSE FALSE FALSE      FALSE FALSE
## is.na    TRUE  TRUE FALSE      TRUE FALSE
## is.nan   FALSE TRUE FALSE      FALSE FALSE
is.null(NULL)
## [1] TRUE
is.nan(NULL)
## logical(0)
is.na(NULL)
## logical(0)
is.null(logical(0))
## [1] FALSE
is.nan(logical(0))
## logical(0)
is.na(logical(0))
## logical(0)
x <- c(1, NA, 3, NaN, 5)
is.null(x)
## [1] FALSE
is.nan(x)
## [1] FALSE FALSE FALSE  TRUE FALSE
is.na(x)
## [1] FALSE  TRUE FALSE  TRUE FALSE

```

Gleichheit: Welche Werte haben folgende Ausdrücke? 1:3 == NA, identical(NA, NA), NULL == NULL, identical(NULL, NULL)

```

1:3 == NA
## [1] NA NA NA
NA == NA
## [1] NA
identical(NA, NA)
## [1] TRUE
NULL == NULL
## logical(0)
identical(NULL, NULL)
## [1] TRUE

```

Die vier häufigen Basistypen haben Default-Werte (FALSE, 0L, 0, ""). Default-Werte sind nicht nichts. Insbesondere ist length("") gleich 1, aber nchar("") gleich 0 und nchar(NA\_character\_) ist NA\_integer\_.

```

c(length(""), nchar(""), length(NA_character_), nchar(NA_character_))
## [1] 1 0 1 NA
typeof(nchar(NA_character_))
## [1] "integer"

```

# V03 – Subsetting

26. April 2021

## Contents

<b>1</b>	<b>Intro</b>	<b>1</b>
<b>2</b>	<b>Auswahl mehrerer Elemente</b>	<b>1</b>
2.1	Vektoren . . . . .	1
2.2	Arrays . . . . .	4
2.3	Tibbles . . . . .	8
<b>3</b>	<b>Auswahl eines einzelnen Elementes</b>	<b>9</b>
3.1	[[ . . . . .	9
3.2	\$ . . . . .	10
<b>4</b>	<b>Fehlende Indizes und Indizes Out-of-Bounds</b>	<b>11</b>
<b>5</b>	<b>Subsetting und Zuweisung</b>	<b>12</b>
<b>6</b>	<b>Anwendungen</b>	<b>15</b>
6.1	Lookup-Tabellen . . . . .	15
6.2	Matching and Merging . . . . .	15
6.3	Random samples / bootstrap . . . . .	16
6.4	Sortieren . . . . .	17
6.5	Aggregierte Zeilen expandieren . . . . .	18
6.6	Spalten aus Tibbles entfernen . . . . .	18
6.7	Konditionales Subsetting . . . . .	19
6.8	Boolesche Algebra vs Mengen . . . . .	19

## 1 Intro

Mit **Subsetting** (Teilmenge bilden) wird das Auswählen von Elementen aus Vektoren (oder darauf aufbauenden Datenstrukturen) bezeichnet. Zum Teil werden dafür auch die Begriffe Indizierung (*indexing*) oder *array slicing* benutzt.

R besitzt verschiedene, sehr mächtige Subsetting-Systeme, um Teile von atomaren Vektoren, Listen, Matrizen, Arrays oder Tibbles auszuwählen und zu verändern.

## 2 Auswahl mehrerer Elemente

### 2.1 Vektoren

Es gibt 6 Möglichkeiten, einen Vektor zu subsetten.

### 2.1.1 Positive ganze Zahlen

Der Rückgabewert besteht aus den Elementen an den angegebenen Positionen (Indizes). (Die erste Position ist 1, nicht 0.)

```
x <- c(2.1, 4.2, 3.3, 5.4, 1.5) # Die Nachkommastelle gibt die Position an
x[c(3, 1)]
## [1] 3.3 2.1
lst <- as.list(x)
str(lst[c(5L, 1L, 3L, 2L, 4L)]) # sortiere Liste um
## List of 5
## $ : num 1.5
## $ : num 2.1
## $ : num 3.3
## $ : num 4.2
## $ : num 5.4
str(lst[2]) # auch Liste
## List of 1
## $ : num 4.2
```

Mehrfaches Auftreten eines Index führt zu mehrfacher Rückgabe.

```
x[c(1, 1, 2, 1)]
## [1] 2.1 2.1 4.2 2.1
```

Bei reellen Zahlen als Indizes werden die Nachkommastellen abgeschnitten, da `double` automatisch in `integer` konvertiert wird.

```
str(lst[c(2.1, 2.9)])
## List of 2
## $ : num 4.2
## $ : num 4.2
```

### 2.1.2 Negative ganze Zahlen

Die Elemente an den angegebenen Indizes werden weggelassen.

```
x[-c(3, 1)]
## [1] 4.2 5.4 1.5
```

Negative und positive ganze Zahlen dürfen nicht gemischt werden.

```
# lst[c(-1, 2)] # ERROR
```

### 2.1.3 0, NULL

Ein Vektor der Länge 0 wird zurückgegeben.

```
x[0]
## numeric(0)
x[NULL]
## numeric(0)
x[integer(0)]
## numeric(0)
lst[0]
## list()
```

0 kann mit negativen oder mit positiven ganzen Zahlen gemischt werden (dann wird 0 ignoriert).

```
x[0:2]
## [1] 2.1 4.2
x[0:-2]
## [1] 3.3 5.4 1.5
```

Beachte den Unterschied zwischen `1:length(x)` und `seq_along(x)`.

```
f <- function(x) {for (i in 1:length(x)) cat("o"); cat("\n")}
g <- function(x) {for (i in seq_along(x)) cat("o"); cat("\n")}

x
## [1] 2.1 4.2 3.3 5.4 1.5
f(x)
## ooooo
g(x)
## ooooo

y <- x[x>9]
f(y)
## oo
g(y)

y
## numeric(0)
1:length(y)
## [1] 1 0
seq_along(y)
## integer(0)
```

## 2.1.4 Nichts

Der ursprüngliche Vektor wird zurückgegeben. Dies ist nicht besonders nützlich für Vektoren aber für Arrays und Tibbles (wie wir bald sehen werden).

```
x[]
## [1] 2.1 4.2 3.3 5.4 1.5
str(lst[])
## List of 5
## $ : num 2.1
## $ : num 4.2
## $ : num 3.3
## $ : num 5.4
## $ : num 1.5
```

## 2.1.5 Wahrheitswerte

Jene Elemente werden ausgewählt, an deren Position der Index-Vektor `TRUE` ist.

```
str(lst[c(T, T, F, F, T)])
## List of 3
## $ : num 2.1
## $ : num 4.2
## $ : num 1.5
x[x>3]
## [1] 4.2 3.3 5.4
```

Ist der Index-Vektor vom Typ `logical` kürzer als der indizierte Vektor, so wird der Index-Vektor so lange wiederholt, bis er die selbe Länge hat (**Recycling**).

```
x[c(T,F)] # gleich x[c(T,F,T,F,T)]
## [1] 2.1 3.3 1.5
x[c(F,T,F)] # gleich x[c(F,T,F,F,T)]
## [1] 4.2 1.5
```

## 2.1.6 Strings

Sind die Elemente eines Vektors benannt, kann ein `character`-Vektor aus diesen Namen genutzt werden, um entsprechende Elemente auszuwählen.

```
y <- structure(x, names=letters[1:4])
y # letzter Eintrag hat keinen Namen
##      a      b      c      d <NA>
## 2.1 4.2 3.3 5.4 1.5
y[c("d", "c", "a", "x")] # kein Element mit Namen "x"
##      d      c      a <NA>
## 5.4 3.3 2.1  NA
y[c("", "NA", "<NA>", NA_character_)] # unbenannte Elemente können nicht mit einem Namen gefunden werden
## <NA> <NA> <NA> <NA>
##  NA  NA  NA  NA
y[c("a", "a", "a", "b", "b")] # Mehrfachauswahl
##      a      a      a      b      b
## 2.1 2.1 2.1 4.2 4.2
```

## 2.2 Arrays

Es gibt 3 Arten, wie höher-dimensionale Strukturen indiziert werden können: durch einen einzelnen atomaren Vektor, mit mehreren atomaren Vektoren, mit einer Matrix.

### 2.2.1 Ein atomarer Vektor

Arrays (und damit Matrizen) bauen auf Vektoren auf und können genau wie Vektoren indiziert werden.

```
# erzeuge character-Matrix mit Indizes als Text:
matr <- outer(1:5, 1:4, FUN = "paste", sep = ",")
matr
##      [,1] [,2] [,3] [,4]
## [1,] "1,1" "1,2" "1,3" "1,4"
## [2,] "2,1" "2,2" "2,3" "2,4"
## [3,] "3,1" "3,2" "3,3" "3,4"
## [4,] "4,1" "4,2" "4,3" "4,4"
## [5,] "5,1" "5,2" "5,3" "5,4"
vals <- matr
dim(vals) <- NULL # entferne Dimensionsattribut
vals
## [1] "1,1" "2,1" "3,1" "4,1" "5,1" "1,2" "2,2" "3,2" "4,2" "5,2" "1,3" "2,3"
## [13] "3,3" "4,3" "5,3" "1,4" "2,4" "3,4" "4,4" "5,4"
matr[c(4, 15)] # gleich vals[c(4, 15)]
## [1] "4,1" "5,3"
matr[c(T,F)]
## [1] "1,1" "3,1" "5,1" "2,2" "4,2" "1,3" "3,3" "5,3" "2,4" "4,4"
matr[c(-1:-10)]
## [1] "1,3" "2,3" "3,3" "4,3" "5,3" "1,4" "2,4" "3,4" "4,4" "5,4"
```

Die Elemente von Arrays werden in der sogenannten *column-major order* gespeichert. Für höher-dimensionale Arrays bedeutet dies, dass bei Iteration über die Einträge die Dimensionen von links nach rechts durchgegangen werden.

```
# Gib alle 27 Indextupel eines 3x3x3 Arrays aus.
a <- array(1:27, rep(3,3))
index_matrix <- sapply(seq_along(dim(a)), function (i) slice.index(a,i))
index_strings <- apply(index_matrix, 1, function(x) paste0("[", paste(x, collapse=", "), ", ]"))
index_tbl <- tibble::tibble("Position als Vektor"=seq_along(a), Indextupel=index_strings)
print(index_tbl, n=27)
## # A tibble: 27 x 2
##       `Position als Vektor` Indextupel
##       <int> <chr>
## 1 1 [1, 1, 1]
## 2 2 [2, 1, 1]
## 3 3 [3, 1, 1]
## 4 4 [1, 2, 1]
## 5 5 [2, 2, 1]
## 6 6 [3, 2, 1]
## 7 7 [1, 3, 1]
## 8 8 [2, 3, 1]
## 9 9 [3, 3, 1]
## 10 10 [1, 1, 2]
## 11 11 [2, 1, 2]
## 12 12 [3, 1, 2]
## 13 13 [1, 2, 2]
## 14 14 [2, 2, 2]
## 15 15 [3, 2, 2]
## 16 16 [1, 3, 2]
## 17 17 [2, 3, 2]
## 18 18 [3, 3, 2]
## 19 19 [1, 1, 3]
## 20 20 [2, 1, 3]
## 21 21 [3, 1, 3]
## 22 22 [1, 2, 3]
## 23 23 [2, 2, 3]
## 24 24 [3, 2, 3]
## 25 25 [1, 3, 3]
## 26 26 [2, 3, 3]
## 27 27 [3, 3, 3]
```

## 2.2.2 Mehrere atomare Vektoren

Um die Dimensions-Eigenschaft auszunutzen, können wir durch , getrennt für jede Dimension einzeln einen Index-Vektor angeben.

Alle 6 Arten zur Indizierung von Vektoren sind erlaubt. Für die Nutzung von character-Subsetting muss das Attribut `dimnames` gesetzt sein.

Verschiedene Dimensionen können verschiedene Indizierungsarten nutzen.

Subsetting mit *nichts* kann hier sinnvoll eingesetzt werden, um ganze Zeilen / Spalten / Dimensionen auszuwählen.

```
m <- matrix(1:9, nrow = 3)
colnames(m) <- c("A", "B", "C")
m
```

```

##      A B C
## [1,] 1 4 7
## [2,] 2 5 8
## [3,] 3 6 9
m[1:2, ]
##      A B C
## [1,] 1 4 7
## [2,] 2 5 8
m[c(TRUE, FALSE, TRUE), c("B", "A")]
##      B A
## [1,] 4 1
## [2,] 6 3
m[0, -2]
##      A C
dim(m[0, -2])
## [1] 0 2

```

Standardmäßig wird das Resultat von `[` auf die kleinst-mögliche Anzahl an Dimensionen vereinfacht (Dimensionen mit Wert 1 werden weggelassen / “gedropt”). Ggf wird das `dim`-Attribut vollständig entfernt. Dieses Verhalten wird mit dem optionalen Argument `drop=FALSE` verhindert.

```

m[1,]
## A B C
## 1 4 7
dim(m[1,]) # Vektor, keine Matrix
## NULL
m[1,,drop=FALSE] # 1x3 Matrix
##      A B C
## [1,] 1 4 7
dim(m[1,,drop=FALSE])
## [1] 1 3
a <- array(1:16, dim=rep(2,4))
dim(a[,1,1,]) # aus 4D-Array wird Matrix
## [1] 2 2
dim(a[,1,1,,drop=FALSE]) # bleibt 4D-Array
## [1] 2 1 1 2

```

**Bemerkung:** Faktoren haben auch ein `drop`-Argument. Jedoch ist die Bedeutung etwas anders: Es gibt an, ob Levels erhalten bleiben. Der Default ist `FALSE`.

```

z <- factor(c("a", "b", "a"))
z[c(1,3)] # zwei Levels (obwohl eines nicht vorkommt)
## [1] a a
## Levels: a b
z[c(1,3), drop = TRUE] # ein Level
## [1] a a
## Levels: a

```

### 2.2.3 Eine Matrix

Als dritte Möglichkeit können Arrays mit  $n$  Dimensionen durch `integer`- (oder ggf `character`)-Matrizen mit  $n$  Spalten indiziert werden. Jede Zeile der Matrix gibt ein Index-Tupel an; das entsprechende Element wird ausgewählt. Das Resultat ist ein Vektor (ohne Dimension).

```

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

```

```

## [1,] "1,1" "1,2" "1,3" "1,4"
## [2,] "2,1" "2,2" "2,3" "2,4"
## [3,] "3,1" "3,2" "3,3" "3,4"
## [4,] "4,1" "4,2" "4,3" "4,4"
## [5,] "5,1" "5,2" "5,3" "5,4"
select <- rbind(
  c(1, 1),
  c(3, 1),
  c(2, 4)
)
select
##      [,1] [,2]
## [1,]    1    1
## [2,]    3    1
## [3,]    2    4
matr[select]
## [1] "1,1" "3,1" "2,4"

```

Die Funktion `arrayInd()` wandelt einen Vektor aus Vektor-Indizes in eine Matrix aus Array-Indizes.

```

arrayInd(1:6, c(2,3))
##      [,1] [,2]
## [1,]    1    1
## [2,]    2    1
## [3,]    1    2
## [4,]    2    2
## [5,]    1    3
## [6,]    2    3
arrayInd(1:6, c(3,2))
##      [,1] [,2]
## [1,]    1    1
## [2,]    2    1
## [3,]    3    1
## [4,]    1    2
## [5,]    2    2
## [6,]    3    2

```

Für ein `logical`-Objekt gibt `which()` die Indizes der TRUE-Einträge zurück. Mit `arr.ind=TRUE` ist die Rückgabe bei Arrays die Matrix aus Array-Indizes, ansonsten werden Vektor-Indizes zurückgegeben.

```

x <- matrix(1:9, nrow=3)
x
##      [,1] [,2] [,3]
## [1,]    1    4    7
## [2,]    2    5    8
## [3,]    3    6    9
x %% 2 == 0
##      [,1] [,2] [,3]
## [1,] FALSE TRUE FALSE
## [2,] TRUE FALSE TRUE
## [3,] FALSE TRUE FALSE
which(x %% 2 == 0)
## [1] 2 4 6 8
which(x %% 2 == 0, arr.ind=T)
##      row col

```

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

## 2.3 Tibbles

Tibbles haben Eigenschaften von Listen und von Matrizen. Werden sie mit einem einzelnen atomaren Vektor indiziert, verhalten sie sich wie Listen. Beim Indizieren mit zwei Vektoren verhalten sie sich wie Matrizen.

```
library(tibble)
tb <- tibble(x = 1:3, y = 3:1, z = letters[1:3])
tb
## # A tibble: 3 x 3
##       x     y     z
##   <int> <int> <chr>
## 1     1     3 a
## 2     2     2 b
## 3     3     1 c
tb[tb$x == 2, ]
## # A tibble: 1 x 3
##       x     y     z
##   <int> <int> <chr>
## 1     2     2 b
tb[c(1, 3), ]
## # A tibble: 2 x 3
##       x     y     z
##   <int> <int> <chr>
## 1     1     3 a
## 2     3     1 c
tb[c("x", "z")]
## # A tibble: 3 x 2
##       x     z
##   <int> <chr>
## 1     1 a
## 2     2 b
## 3     3 c
tb[, c("x", "z")]
## # A tibble: 3 x 2
##       x     z
##   <int> <chr>
## 1     1 a
## 2     2 b
## 3     3 c
```

Werden Tibbles mit [ indiziert, erhält man immer ein Tibble. `drop = FALSE` ist bei Tibble der Default.

```
tb[["x"]]
## [1] 1 2 3
tb["x"]
## # A tibble: 3 x 1
##       x
##   <int>
## 1     1
## 2     2
```

```
## 3      3
tb[, "x"]
## # A tibble: 3 x 1
##      x
## <int>
## 1      1
## 2      2
## 3      3
tb[, "x", drop=TRUE]
## [1] 1 2 3
```

### 3 Auswahl eines einzelnen Elementes

Neben `[` gibt es noch 2 weitere Subsetting-Operatoren: `[[` und `$`. Diese dienen der Auswahl eines einzelnen Elementes. Dabei entspricht `x$y` dem Aufruf von `x[["y"]]`.

#### 3.1 `[[`

Der Subsetting-Operator `[[` hat für Listen größte Bedeutung. Mit `[` erhält man immer eine Liste, mit `[[` das Element.

```
x <- list(bla=1:3, blub="a", 4:6)
x[1]
## $bla
## [1] 1 2 3
x[[1]]
## [1] 1 2 3
```

Für das Indizieren mit `[[` werden eine einzelne natürliche Zahl oder ein String benutzt.

```
x[[2]]
## [1] "a"
x[["blub"]]
## [1] "a"
```

Wird ein atomarer Vektor der Länge  $> 1$  übergeben, wird rekursiv indiziert, was typischerweise nur bei besonderen Datenstrukturen zum Einsatz kommt.

```
b <- list(x=1, a=list(y=2, b=list(z=3, c=list(d=5, w=4))))
str(b)
## List of 2
## $ x: num 1
## $ a:List of 2
##   ..$ y: num 2
##   ..$ b:List of 2
##     ..$ z: num 3
##     ..$ c:List of 2
##       ..$ d: num 5
##       ..$ w: num 4
# folgende Ausdrücke haben den selben Effekt:
b[["a"]][["b"]][["c"]][["d"]]
## [1] 5
b[[2]][[2]][[2]][[1]]
## [1] 5
b[[c("a", "b", "c", "d")]]
```

```
## [1] 5
b[[c(2, 2, 2, 1)]]
## [1] 5
```

Auch einzelne Elemente atomarer Vektoren können durch `[[` ausgewählt werden. Dann unterscheidet sich das Resultat nicht von `[`.

```
x <- 11:15
x[[2]]
## [1] 12
x[[c(2,3)]]
## Error in x[[c(2, 3)]]: attempt to select more than one element in vectorIndex
```

`[[` wird für atomare Vektoren genutzt, um zu verdeutlichen, dass genau ein Element ausgewählt wird und um Fehler zu vermeiden.

```
# Anstatt:
if (x[i] > 0 & y[j] < 0) do_stuff()
# ist es besser Stil zu schreiben:
if (x[[i]] > 0 && y[[j]] < 0) do_stuff()
```

**Hinweis:** `&&` und `||` sind die nicht-vektorisierten Versionen von `&` und `l`. Da `if`-Ausdrücke genau einen Wahrheitswert verlangen, wird dort bevorzugt `&&`, `||` und `[[` benutzt anstatt `&`, `l`, `[`.

Bei den selten vorkommenden Listen-Matrizen (oder Listen-Arrays) ist auch Indizierung mit einem Index pro Dimension in `[[` erlaubt.

```
lst_mat <- matrix(lapply(1:6, sample), nrow=2)
lst_mat
##      [,1]     [,2]     [,3]
## [1,] 1      Integer,3 Integer,5
## [2,] Integer,2 Integer,4 Integer,6
lst_mat[3]
## [[1]]
## [1] 3 1 2
lst_mat[1,2]
## [[1]]
## [1] 3 1 2
lst_mat[[3]]
## [1] 3 1 2
lst_mat[[1,2]]
## [1] 3 1 2
```

## 3.2 `$`

`x$y` entspricht in etwa `x[["y"]]`. Damit können benannte Listeneinträge und Spalten von Tibbles ausgewählt werden.

Wenn der Name einer Spalte in einer Variable steht, kann `$` nicht genutzt werden.

```
tbl <- tibble(number=1:5, letter=letters[1:5])
var <- "number"
tbl$var
## Warning: Unknown or uninitialised column: `var`.
## NULL
tbl[[var]]
## [1] 1 2 3 4 5
```

Im Gegensatz zu `[[` wird bei `$ partial matching` durchgeführt wird. Dabei genügt es, den Anfang des Namens eines Eintrages zu schreiben, um diesen auszuwählen, solange dadurch das Element eindeutig identifiziert wird (**prefix condition**).

```
x <- list(abc = -5, anders = 7)
x$ab
## [1] -5
x$an
## [1] 7
x$a # nicht eindeutig
## NULL
x[["ab"]]
## NULL
x[["ander"]]
## NULL
```

Da dies sehr leicht zu Fehlern führen kann, ist empfohlen diese Funktionalität nicht auszunutzen. Um vor *partial matching* zu warnen, kann die globale Option `warnPartialMatchDollar` gesetzt werden.

```
options(warnPartialMatchDollar = TRUE)
x$an
## Warning in x$an: partial match of 'an' to 'anders'
## [1] 7
```

## 4 Fehlende Indizes und Indizes Out-of-Bounds

Indizes größer als die Länge des Vektors und Index-Strings, die nicht zu den Namen der Elemente des Vektors gehören, sind **out-of-bounds**.

Die Subsetting-Operatoren verhalten sich bei *out-of-bounds*-Werten unterschiedlich.

```
x <- 1:3
x[4]
## [1] NA
x[[4]]
## Error in x[[4]]: subscript out of bounds
lst <- as.list(x)
lst[4]
## [[1]]
## NULL
lst[[4]]
## Error in lst[[4]]: subscript out of bounds
names(x) <- letters[1:3]
names(lst) <- letters[1:3]
x["d"]
## <NA>
## NA
x[["d"]]
## Error in x[["d"]]: subscript out of bounds
lst["d"]
## $<NA>
## NULL
lst[["d"]]
## NULL
lst$d
## NULL
```

Vektor-Typ	[.integer	[[.integer	[.character	[[.character	\$
<b>atomar</b>	NA	<i>ERROR</i>	NA	<i>ERROR</i>	-
<b>list</b>	list(NULL)	<i>ERROR</i>	list(NULL)	NULL	NULL

Für benannte Vektoren wird der Name von fehlenden Elementen als "<NA>" angezeigt.

```
x <- c(a=1, 2, b=3)
x[1:4]
##   a         b <NA>
## 1 2 3 NA
```

Indizierung mit NA-Werten (NA\_integer\_, NA) ergibt NA-Werte.

```
x <- 1:3
x[NA_integer_]
## [1] NA
x[NA] # NA ist logical -> recycling zu x[c(NA, NA, NA)]
## [1] NA NA NA
```

## 5 Subsetting und Zuweisung

Alle Subsetting-Operatoren können mit dem Zuweisungsoperator <- kombiniert werden.

```
x <- 1:5
x[c(1, 2)] <- 11:12
x
## [1] 11 12 3 4 5

x[[3]] <- 23
x
## [1] 11 12 23 4 5

x[-1] <- 32:35
x
## [1] 11 32 33 34 35

x[c(1, 1, 1)] <- 41:43 # Effekt schlecht vorhersehbar -> Vermeiden!
x
## [1] 43 32 33 34 35

x[c(1, NA)] <- c(1, 2) # ERROR
## Error in x[c(1, NA)] <- c(1, 2): NAs are not allowed in subscripted assignments

x[c(T, F, NA)] <- 50 # NA wird bei Zuweisung wie FALSE behandelt
x
## [1] 50 32 33 50 35

# obiges Verhalten ist nützlich in Kombination mit bedingter Zuweisung
tb <- tibble(a = c(1, 10, NA))
tb$a < 5
## [1] TRUE FALSE    NA
tb$a[tb$a < 5] <- 0
tb$a
## [1] 0 10 NA
```

**Partial matching** bei `$` wird nicht bei der Zuweisung durchgeführt.

```
x <- list(abc = 1)
x$ab
## Warning in x$ab: partial match of 'ab' to 'abc'
## [1] 1
x$ab <- 2
str(x)
## List of 2
## $ abc: num 1
## $ ab : num 2
```

Bei der Zuweisung kommt es ggf zur Typenumwandlung (*Coercion*).

```
x <- 1:4
x[3:4] <- c(T,F)
x
## [1] 1 2 1 0
x[1] <- "0"
x
## [1] "0" "2" "1" "0"
```

Betrachte die Zuweisung `x[i] <- y` für Vektoren `x` und `y` gleichen Typs mit `length(x[i])` gleich  $n > 0$  und `length(y)` gleich  $m > 0$ . Sei  $k = \lceil n/m \rceil \in \mathbb{N}$  die kleinste natürliche Zahl sodass  $mk \geq n$ . Dann wird der Vektor `y`  $k$ -mal wiederholt und die ersten  $n$  Elemente `x[i]` zugewiesen. Dies entspricht `x[i] <- rep(y, k)[1:n]`. Ist  $mk = n$  vereinfacht sich dies zu `x[i] <- rep(y, k)`. Ist  $mk \neq n$  wird eine Warnung ausgegeben. Dieses Verhalten wird als **Recycling** bezeichnet.

```
x <- 1:9
x[1:6] <- 11:13 # n=6, m=3, k=2
x
## [1] 11 12 13 11 12 13 7 8 9
x[7:10] <- 22 # n=4, m=1, k=4
x
## [1] 11 12 13 11 12 13 22 22 22
x[1:6] <- 31:34 # n=6, m=4, k=2
## Warning in x[1:6] <- 31:34: number of items to replace is not a multiple of
## replacement length
x
## [1] 31 32 33 34 31 32 22 22 22
x[1:2] <- 41:43 # n=2, m=3, k=1
## Warning in x[1:2] <- 41:43: number of items to replace is not a multiple of
## replacement length
x
## [1] 41 42 33 34 31 32 22 22 22
lst <- as.list(1:8)
lst[2:7] <- list(T, "up") # n=6, m=2, k=3
str(lst)
## List of 8
## $ : int 1
## $ : logi TRUE
## $ : chr "up"
## $ : logi TRUE
## $ : chr "up"
## $ : logi TRUE
## $ : chr "up"
```

```
## $ : int 8
```

Es ist möglich *out-of-bounds* Zuweisungen durchzuführen, wodurch der Vektor automatisch vergrößert wird. Dies ist besonders für Listen praktisch, kann aber auch zu Performance-Problemen führen.

```
lst <- list()
lst$a <- 1
lst[["b"]] <- 2
lst[[3]] <- 3
str(lst)
## List of 3
## $ a: num 1
## $ b: num 2
## $ : num 3

N <- 1e7
system.time({
  x <- integer(0)
  for (i in 1:N) x[i] <- 42L # out of bounds assignment
})
##    user  system elapsed
##  2.04    0.14   2.19
system.time({
  x <- integer(N)
  for (i in 1:N) x[i] <- 42L # in bounds assignment
})
##    user  system elapsed
##  0.33    0.00   0.33
```

Subsetting mit *nichts* kann in Verbindung mit Zuweisung nützlich sein, da Attribute des ursprünglichen Objektes erhalten bleiben.

```
tb <- tibble(x=2^(1:4), y=c("1.2", "2.3", "3.4", "4.5"))
tb <- lapply(tb, as.integer)
class(tb) # explizite Klasse wurde nicht übertragen
## [1] "list"
tb <- tibble(x=2^(1:4), y=c("1.2", "2.3", "3.4", "4.5"))
tb[] <- lapply(tb, as.integer)
class(tb) # Tibble-Klasse bleibt erhalten
## [1] "tbl_df"     "tbl"        "data.frame"
```

Die Kombination aus Subsetting und Zuweisungsoperator notieren wir hier [`<-`, `[[<-`, bzw `$<-`.

Listenelemente können mittels `[[<-` und `NULL` aus der Liste entfernt werden. Um `NULL` einer Liste hinzuzufügen, kann `[<-` mit `list(NULL)` genutzt werden.

```
x <- list(a = 1, b = 2)
x[["b"]] <- NULL
str(x)
## List of 1
## $ a: num 1
y <- list(a = 1)
y[["b"]] <- list(NULL)
str(y)
## List of 2
## $ a: num 1
```

```
## $ b: NULL
```

## 6 Anwendungen

### 6.1 Lookup-Tabellen

Um für Abkürzungen den vollständigen Ausdruck nachzuschlagen, erstellen wir einen benannten `character`-Vektor. Die Namen sind die Abkürzungen, die Werte entsprechen den vollständigen Begriffen.

```
x <- c("m", "w", "u", "w", "m")
lookup <- c(m = "männlich", w = "weiblich", u = NA)
lookup[x]
##          m           w           u           w           m
## "männlich" "weiblich"      NA "weiblich" "männlich"
unname(lookup[x])
## [1] "männlich" "weiblich" NA           "weiblich" "männlich"
```

### 6.2 Matching and Merging

Angenommen wir haben eine Lookup-Tabelle mit mehreren Spalten. Mit der Funktion `match()` finden wir die richtigen Indizes in der Lookup-Tabelle und können damit unseren Datensatz erweitern.

```
students <- tibble(
  name = c("Alice", "Bob", "Carl", "Dave", "Eve"),
  grade = c(1, 2, 2, 3, 1)
)
students
## # A tibble: 5 x 2
##   name   grade
##   <chr> <dbl>
## 1 Alice     1
## 2 Bob       2
## 3 Carl      2
## 4 Dave      3
## 5 Eve       1
info <- tibble(
  grade = 3:1,
  desc = c("Poor", "Good", "Excellent"),
  fail = c(T, F, F)
)
info
## # A tibble: 3 x 3
##   grade desc      fail
##   <int> <chr>    <lgl>
## 1     3 Poor     TRUE
## 2     2 Good    FALSE
## 3     1 Excellent FALSE
id <- match(students$grade, info$grade)
id
## [1] 3 2 2 1 3
info[id, ]
## # A tibble: 5 x 3
##   grade desc      fail
##   <int> <chr>    <lgl>
```

```

## 1 1 Excellent FALSE
## 2 2 Good FALSE
## 3 2 Good FALSE
## 4 3 Poor TRUE
## 5 1 Excellent FALSE
cbind(students, info[id, -1])
##   name grade desc fail
## 1 Alice 1 Excellent FALSE
## 2 Bob   2 Good FALSE
## 3 Carl  2 Good FALSE
## 4 Dave  3 Poor TRUE
## 5 Eve   1 Excellent FALSE

```

Etwas kürzer ist dies mit der Funktion `merge()` zu bewerkstelligen.

```

merge(x=students, y=info, by.x="grade", by.y="grade")
##   grade name desc fail
## 1     1 Alice Excellent FALSE
## 2     1 Eve   Excellent FALSE
## 3     2 Bob   Good FALSE
## 4     2 Carl  Good FALSE
## 5     3 Dave  Poor TRUE
# Da grade die einzige Spalte ist, die in beiden Tibbles vorkommt, geht es noch kürzer:
stu <- merge(students, info)
stu
##   grade name desc fail
## 1     1 Alice Excellent FALSE
## 2     1 Eve   Excellent FALSE
## 3     2 Bob   Good FALSE
## 4     2 Carl  Good FALSE
## 5     3 Dave  Poor TRUE

```

### 6.3 Random samples / bootstrap

`sample(x, n)` wählt `n` Elemente des Vektors `x` zufällig ohne zurücklegen (bei `replace=TRUE` mit zurücklegen) aus. Siehe `?sample`. Um aus gegebenen Daten neue, ‘zufällige’ Daten zu generieren, wählen wir zufällige Zeilen-Indizes aus.

```

stu[sample(nrow(tb)), ] # Randomly reorder
##   grade name desc fail
## 3     2 Bob   Good FALSE
## 1     1 Alice Excellent FALSE
## 4     2 Carl  Good FALSE
## 2     1 Eve   Excellent FALSE
stu[sample(nrow(tb), 3), ] # Select 3 random rows
##   grade name desc fail
## 1     1 Alice Excellent FALSE
## 2     1 Eve   Excellent FALSE
## 3     2 Bob   Good FALSE
stu[sample(nrow(tb), 6, replace = TRUE), ] # Select 6 bootstrap replicates
##   grade name desc fail
## 3     2 Bob   Good FALSE
## 3.1   2 Bob   Good FALSE
## 4     2 Carl  Good FALSE
## 4.1   2 Carl  Good FALSE

```

```
## 2      1 Eve Excellent FALSE
## 3.2    2 Bob     Good FALSE
```

## 6.4 Sortieren

Sortiere nach Zeilen oder Spalten eines Tibbles mittels `order()`.

```
x <- c(20, 30, 10)
order(x)
## [1] 3 1 2
x[order(x)]
## [1] 10 20 30

stu2 <- stu[sample(nrow(stu)), ] # randomly perturb
stu2
##   grade name      desc fail
## 1     1 Alice Excellent FALSE
## 2     1 Eve Excellent FALSE
## 3     2 Bob      Good FALSE
## 5     3 Dave     Poor  TRUE
## 4     2 Carl     Good FALSE
stu2[order(stu2$grade), ] # sort rows by grade
##   grade name      desc fail
## 1     1 Alice Excellent FALSE
## 2     1 Eve Excellent FALSE
## 3     2 Bob      Good FALSE
## 4     2 Carl     Good FALSE
## 5     3 Dave     Poor  TRUE
stu2[, order(names(stu2))] # sort columns by name
##      desc fail grade name
## 1 Excellent FALSE     1 Alice
## 2 Excellent FALSE     1 Eve
## 3 Good FALSE      2 Bob
## 5 Poor  TRUE      3 Dave
## 4 Good FALSE      2 Carl
```

Sortieren basiert auf dem Vergleichsoperatoren. Diese sind auch für `character`-Werte definiert und folgen der **lexikographischen Ordnung**. (siehe auch `?<`)

```
"a" < "A"
## [1] TRUE
"A" < "ä"
## [1] TRUE
"ä" < "aa"
## [1] TRUE
"aa" < "ab"
## [1] TRUE
"ab" < "b"
## [1] TRUE
```

Beachte den Unterschied zwischen `sort()` (sortierter atomarer Vektor), `order()` (Indizes zur Sortierung), `rank()` (Rang: “1. Platz”, “2. Platz”, …)

```
x <- c(40, 20, 10, 30)
sort(x)
## [1] 10 20 30 40
```

```

rank(x)
## [1] 4 2 1 3
order(x)
## [1] 3 2 4 1
tb <- tibble(num = x, rank=rank(x))
tb
## # A tibble: 4 x 2
##       num   rank
##   <dbl> <dbl>
## 1     40     4
## 2     20     2
## 3     10     1
## 4     30     3
tb[order(x),]
## # A tibble: 4 x 2
##       num   rank
##   <dbl> <dbl>
## 1     10     1
## 2     20     2
## 3     30     3
## 4     40     4

```

## 6.5 Aggregierte Zeilen expandieren

Angenommen mehrfach vorkommende Beobachtungen werden nur einmal gelistet. Dafür wird in einer zusätzlichen Spalte (n) ihre Häufigkeit notiert. Um den expandierten Datensatz zu erhalten kann ein entsprechende Index-Vektor mit `rep()` erzeugt werden.

```

tb <- tibble(x = c(2, 4, 1), y = c(9, 11, 6), n = c(3, 5, 1))
rep(1:nrow(tb), tb$n)
## [1] 1 1 1 2 2 2 2 2 3
tb[rep(1:nrow(tb), tb$n), ]
## # A tibble: 9 x 3
##       x     y     n
##   <dbl> <dbl> <dbl>
## 1     2     9     3
## 2     2     9     3
## 3     2     9     3
## 4     4    11     5
## 5     4    11     5
## 6     4    11     5
## 7     4    11     5
## 8     4    11     5
## 9     1     6     1

```

## 6.6 Spalten aus Tibbles entfernen

Um Spalten zu entfernen, kann man sie auf `NULL` setzen.

```

tib <- tibble(x = 1:3, y = 3:1, z = letters[1:3])
tib$z <- NULL

```

Alternativ können nur alle übrigen Spalten ausgewählt werden.

```
tib$z <- letters[1:3]
tib[c("x", "y")]
## # A tibble: 3 x 2
##       x     y
##   <int> <int>
## 1     1     3
## 2     2     2
## 3     3     1
```

Falls dabei nur die zu entfernenden Spalten beschrieben werden sollen, ist `setdiff()` hilfreich.

```
names(tib)
## [1] "x" "y" "z"
setdiff(names(tib), "z")
## [1] "x" "y"
tb[setdiff(names(tib), "z")]
## # A tibble: 3 x 2
##       x     y
##   <dbl> <dbl>
## 1     2     9
## 2     4    11
## 3     1     6
```

## 6.7 Konditionales Subsetting

Eine der nützlichsten Subsetting-Funktionalitäten in R ist das Subsetting mit `logical`-Vektoren in Kombination mit logischen Operatoren, um nur Daten auszuwählen die bestimmte Bedingungen erfüllen.

```
head(mtcars) # betrachte die obersten Zeilen des Datensatzes mtcars
##          mpg cyl disp hp drat    wt  qsec vs am gear carb
## Mazda RX4   21.0   6 160 110 3.90 2.620 16.46  0  1    4    4
## Mazda RX4 Wag 21.0   6 160 110 3.90 2.875 17.02  0  1    4    4
## Datsun 710  22.8   4 108  93 3.85 2.320 18.61  1  1    4    1
## Hornet 4 Drive 21.4   6 258 110 3.08 3.215 19.44  1  0    3    1
## Hornet Sportabout 18.7   8 360 175 3.15 3.440 17.02  0  0    3    2
## Valiant     18.1   6 225 105 2.76 3.460 20.22  1  0    3    1
# siehe ?mtcars
mtcars[mtcars$gear == 5, ]
##          mpg cyl disp hp drat    wt  qsec vs am gear carb
## Porsche 914-2 26.0   4 120.3 91 4.43 2.140 16.7  0  1    5    2
## Lotus Europa  30.4   4  95.1 113 3.77 1.513 16.9  1  1    5    2
## Ford Pantera L 15.8   8 351.0 264 4.22 3.170 14.5  0  1    5    4
## Ferrari Dino  19.7   6 145.0 175 3.62 2.770 15.5  0  1    5    6
## Maserati Bora 15.0   8 301.0 335 3.54 3.570 14.6  0  1    5    8
mtcars[mtcars$gear == 5 & mtcars$cyl == 4, ]
##          mpg cyl disp hp drat    wt  qsec vs am gear carb
## Porsche 914-2 26.0   4 120.3 91 4.43 2.140 16.7  0  1    5    2
## Lotus Europa  30.4   4  95.1 113 3.77 1.513 16.9  1  1    5    2
```

Achtung: Nutze hierbei `&` und `|` (vektorisiert) und nicht `&&` und `||` (skalare Operatoren).

## 6.8 Boolesche Algebra vs Mengen

Es gibt eine natürliche Äquivalenz zwischen `integer`-Subsetting (Mengenoperationen) und `logical` Subsetting (Boolesche Algebra).

Mengenoperationen sind effektiver, falls das erste oder letzte TRUE gefunden werden soll oder falls die meisten Einträge des logical-Vektors FALSE sind.

`which()` konvertiert einen logical Index-Vektor in die entsprechende integer-Repräsentation.

```
x <- sample(7)
b <- x < 4
x
## [1] 5 6 1 2 3 4 7
b
## [1] FALSE FALSE  TRUE  TRUE  TRUE FALSE FALSE
which(b)
## [1] 3 4 5
x[which(b)[1]] # erster Wert < 4
## [1] 1
```

Die Umkehrfunktion ist in Base-R nicht implementiert. Wir können sie jedoch einfach selbst schreiben.

```
unwhich <- function(b, n) {
  out <- rep(FALSE, n)
  out[b] <- TRUE
  out
}
unwhich(which(b), length(b))
## [1] FALSE FALSE  TRUE  TRUE  TRUE FALSE FALSE
```

Achtung: `x[-which(b)]` ist nicht äquivalent zu `x[!b]`: Falls alle Einträge von `b` gleich FALSE sind, ist `which(b)` gleich `integer(0)` und `-integer(0)` ist immer noch `integer(0)`.

```
x <- 1:3
b <- c(F, T, F)
x[!b]
## [1] 1 3
x[-which(b)] # gleiches Resultat
## [1] 1 3
b <- c(F, F, F)
x[!b]
## [1] 1 2 3
x[-which(b)] # verschiedene Resultate
## integer(0)
```

Es gibt Äquivalenzen auch zwischen logischen und den Mengenoperatoren. Seien `bx` und `by` logical-Vektoren mit den zugehörigen Index-Vektoren `ix <- which(bx)` und `iy <- which(by)`.

- `bx & by` entspricht `intersect(ix, iy)`,
- `bx | by` entspricht `union(ix, iy)`,
- `bx & !by` entspricht `setdiff(ix, iy)`,
- `xor(bx, by)` entspricht `setdiff(union(ix, iy), intersect(ix, iy))`.

```
bx <- 1:10 %% 2 == 0
ix <- which(bx)
bx
## [1] FALSE  TRUE FALSE  TRUE FALSE  TRUE FALSE  TRUE
ix
## [1] 2 4 6 8 10
by <- 1:10 %% 5 == 0
iy <- which(by)
by
```

```

## [1] FALSE FALSE FALSE FALSE  TRUE FALSE FALSE FALSE  TRUE
iy
## [1] 5 10

bx & by
## [1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE  TRUE
intersect(ix, iy)
## [1] 10

bx | by
## [1] FALSE  TRUE FALSE  TRUE  TRUE  TRUE FALSE  TRUE FALSE  TRUE
union(ix, iy)
## [1] 2 4 6 8 10 5

bx & !by
## [1] FALSE  TRUE FALSE  TRUE FALSE  TRUE FALSE  TRUE FALSE FALSE
setdiff(ix, iy)
## [1] 2 4 6 8

xor(bx, by)
## [1] FALSE  TRUE FALSE  TRUE  TRUE FALSE  TRUE FALSE FALSE
setdiff(union(ix, iy), intersect(ix, iy))
## [1] 2 4 6 8 5

```

# V04 – Funktionen

26. April 2021

## Contents

<b>1</b>	<b>Intro</b>	<b>1</b>
<b>2</b>	<b>Formen des Funktionsaufrufs</b>	<b>3</b>
2.1	Präfixform . . . . .	4
2.2	Infix Funktionen . . . . .	4
2.3	Ersetzungsfunktionen . . . . .	5
2.4	Spezialformen . . . . .	7
2.5	Funktionsaufruf mit Argumentliste . . . . .	7
<b>3</b>	<b>Klassifizierung von Funktionen</b>	<b>7</b>
<b>4</b>	<b>Komponenten einer Funktion</b>	<b>8</b>
<b>5</b>	<b>Funktionsargumente</b>	<b>9</b>
5.1	Lazy Evaluation 1 . . . . .	9
5.2	Default arguments . . . . .	9
5.3	Missing arguments . . . . .	10
5.4	dot-dot-dot . . . . .	11
<b>6</b>	<b>Beenden einer Funktion</b>	<b>12</b>

## 1 Intro

"To understand computations in R, two slogans are helpful:

- Everything that exists is an object.
- Everything that happens is a function call."

— John M. Chambers

Alles, was etwas tut, ist ein *Funktionsaufruf*. Funktionen selbst sind *Objekte*.

Funktionen in R können

- anderen Funktionen als Argument dienen,
- Rückgabewert einer Funktion sein,
- in einer Datenstruktur (Liste) gespeichert werden

Damit sind sie sogenannte **First-Class-Funktionen**.

```
# Funktion als Argument
apply(matrix(1:4, nrow=2), 2, sum)  # Zeilensumme
## [1] 3 7

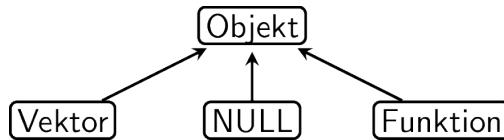
# Funktion als Rückgabewert
create_fun <- function (x) {
```

```

fun <- function() {
  print(x)
}
return(fun)
}
f <- create_fun("blub")
f()
## [1] "blub"

# Funktion in Liste
functs <- list(
  half = function(x) x / 2,
  twice = function(x) x * 2
)
functs$twice(10)
## [1] 20

```



Mit `function(arg_list)` expression können wir ein Funktions-Objekt erzeugen.

Damit wir die Funktion später aufrufen können, weisen wir ihr typischerweise einen Variablenamen zu:  
`function_name <- function(arg_list) expression`.

Wird kein Name zugewiesen sprechen wir von **anonymen Funktionen**.

```

head(mtcars)
##          mpg cyl disp  hp drat    wt  qsec vs am gear carb
## Mazda RX4    21.0   6 160 110 3.90 2.620 16.46  0  1    4    4
## Mazda RX4 Wag 21.0   6 160 110 3.90 2.875 17.02  0  1    4    4
## Datsun 710   22.8   4 108  93 3.85 2.320 18.61  1  1    4    1
## Hornet 4 Drive 21.4   6 258 110 3.08 3.215 19.44  1  0    3    1
## Hornet Sportabout 18.7   8 360 175 3.15 3.440 17.02  0  0    3    2
## Valiant      18.1   6 225 105 2.76 3.460 20.22  1  0    3    1
# siehe ?mtcars
sapply(mtcars, function(x) length(unique(x))) # anonyme Funktion
##   mpg cyl disp  hp drat    wt  qsec vs am gear carb
##  25   3   27   22   22   29   30   2   2   3   6

```

Für `function_name` gelten die selben Regeln für mögliche Namen wie bei anderen Objekten.

Erinnerung: Gültige Namen für Variablen bestehen aus Buchstaben, Ziffern, ., oder \_ und beginnen mit einem Buchstaben oder dem Punkt nicht gefolgt von einer Ziffern und sind keine **reservierte Wörter**.

Weitere gültige Namen sind beliebige Zeichenketten in Backticks (“rückwärts geneigtes Hochkomma”, `).

```

`a b` <- 1
# a b # ERROR
`a b`
## [1] 1
# $ <- 2 # ERROR
`$` <- 2
# $ # ERROR
`$` 

```

```
## [1] 2
`x` <- 42 # ist das Gleiche wie x
x
## [1] 42
```

Diese Regeln, inklusive Backticks, gelten auch für die Namen von Funktionsargumenten.

```
f <- function(`_`, `***`, `42`) {
  c(`_`, `***`, `42`)
}
f(1,2,3)
## [1] 1 2 3
```

Beim Funktionsaufruf können statt Backticks auch Anführungszeichen genutzt werden.

```
f(`***`=1, `42`=2, `_`=3)
## [1] 3 1 2
c(`***`=1, `42`=2, `_`=3)
## *** 42 -
## 1 2 3
```

## 2 Formen des Funktionsaufrufs

Schreiben wir keine Klammern hinter eine Funktion, greifen wir auf das Funktionsobjekt zu statt die Funktion aufzurufen. `print()` gibt für eine Funktion ihren R-Code aus.

```
f <- function(x,y) x+y
f # auf oberster Ebene gleich print(f)
## function(x,y) x+y
```

Bekannt ist der Funktionsaufruf mit `()`, notiert als `function_name(arg_list)`.

Da Operatoren wie `+` / `%%` `%%` `&` `|` `<` `:` `<-` oder Konstrukte wie `if`, `while`, `for`, `[[` etwas tun, sind sie nach obigem Zitat auch Funktionen. Ihr Aufruf hat jedoch eine besondere Form.

Funktionsaufrufe können verschiedene Formen haben:

- **Präfixform**: Funktionsname vor den Argumenten, `f(x,y,z)`
- **Infixform**: Funktionsname zwischen den Argumenten `x+y`
- **Ersetzungsfunktion** (*replacement function*): `names(z) <- c("a", "b", "c")`
- **Spezialformen** ohne konsistente Struktur, zB: `[[`, `if`, `for`

Um auf das Funktionsobjekt einer Funktion zuzugreifen, die nicht in Präfixform aufgerufen wird, verwenden wir Backticks ```. Also etwa ``+`` oder ``[``. Mit `?`+`` gelangen wir etwa zur Hilfe für arithmetische Operatoren.

```
sapply(c(T,F,T), `!`)
## [1] FALSE TRUE FALSE
```

Alle Funktionen können in der Präfixform aufgerufen werden.

```
x + y
`+`(x, y)

names(z) <- c("x", "y", "z")
`names<-`(z, c("x", "y", "z"))

for(i in 1:10) print(i)
`for`(i, 1:10, print(i))
```

```

`+`(1, 2) # 1 + 2
## [1] 3
`<-`(x, 11:15) # x <- 11:15
`[`(x, 2:4) # x[2:4]
## [1] 12 13 14

```

## 2.1 Präfixform

Die Zuordnung von übergebenen Werten zu Argumenten einer Funktion kann durch Position oder Argumentname bestimmt werden. Es ist möglich nur Teile des Namens anzugeben und R findet automatisch den vollständigen Namen (**partial matching**).

```

f <- function(abc, bcd1, bcd2) {
  list(a = abc, b1 = bcd1, b2 = bcd2)
}
str(f(1, 2, 3))
## List of 3
## $ a : num 1
## $ b1: num 2
## $ b2: num 3
str(f(2, 3, abc = 1))
## List of 3
## $ a : num 1
## $ b1: num 2
## $ b2: num 3
str(f(2, 3, a = 1))
## List of 3
## $ a : num 1
## $ b1: num 2
## $ b2: num 3
str(f(1, 3, b = 1))
## Error in f(1, 3, b = 1): argument 3 matches multiple formal arguments

seq(0, 1, len=4) # vollständiger Name "length.out"
## [1] 0.0000000 0.3333333 0.6666667 1.0000000

```

Guter Stil: Nutze nur für die ersten ein bis zwei Argumente die Position, sonst die Namen. Verzichte auf *partial matching*.

## 2.2 Infix Funktionen

Wird eine Funktion mit dem Namen `%op_name%` definiert, kann statt ``%op_name%`(x,y)` auch `x %op_name% y` geschrieben werden.

```

`%asdf%` <- function(x, y) 100*x+y
`%asdf%`(5,7)
## [1] 507
5 %asdf% 7
## [1] 507
`%"` <- function(x, y) paste(x,y)
`%"`(3, "Äpfel")
## [1] "3 Äpfel"
3 %" "Äpfel" %%" T
## [1] "3 Äpfel" TRUE"

```

Operatoren anderer Form können nicht definiert werden. Jedoch kann man vorhandene Operatoren mit neuen Werten belegen. Dies sollte jedoch vermieden werden, um keine allzu große Verwirrung zu stiften.

```
^+` <- function(x,y) x-y # evil
3+4
## [1] -1
rm(`+`) # undo
3+4
## [1] 7
```

Eine Liste von vordefinierten Infix-Funktionen (Operatoren) kann mit `?Syntax` abgerufen werden.

Bemerkung: Es gibt den Operator `:=`, der jedoch in Base-R nicht genutzt wird. Der Operator `**` ist synonym zu `^`.

In `?Syntax` wird auch die Operator-Priorität (Ausführungsreihenfolge) angezeigt.

Bei gleicher Priorität wird von links nach rechts ausgeführt mit wenigen Ausnahmen wie etwa `^`.

```
%-%` <- function(a, b) paste0("(", a, " %-% ", b, ")")
"a" %-% "b" %-% "c"
## [1] "((a %-% b) %-% c)"
2^2^3 == 2^(2^3)
## [1] TRUE
```

Achtung: Die Operatorpriorität des Sequenz-Operators `:` ist niedriger als `^` und Vorzeichen (unärem `+`, `-`) aber höher als `*`, `/` und (binären) `+`, `-`.

```
x <- 1:6
x[2:length(x)-1] # möglicherweise nicht das, was man will
## [1] 1 2 3 4 5
x[2:(length(x)-1)]
## [1] 2 3 4 5
```

Guter Stil: Leerzeichen um alle Operatoren mit strikt niedrigere Priorität als `:`, also etwa `x[1:length(x) - 1]`.

## 2.3 Ersetzungsfunktionen

Ersetzungsfunktionen sind eine der wenigen Funktionen, die ihre Argumente verändern.

Sie haben einen Namen der Form `function_name<-` und müssen Argumente mit den Namen `x` und `value` haben.

Der Aufruf einer solchen Funktion folgt dem Muster `function_name(x) <- value`. Dabei wird nicht nur die Funktion aufgerufen, sondern deren Resultat auch automatisch `x` zugewiesen.

```
second<-` <- function(x, value) {
  x[2] <- value
  x
}
x <- 1:5
second(x) <- 10L
x
## [1] 1 10 3 4 5
`second<-`(x, 100L)
## [1] 1 100 3 4 5
x # call by value
## [1] 1 10 3 4 5
```

```

x <- `second<-`(x, 100L) # dies entspricht second(x) <- 100L
x
## [1] 1 100 3 4 5

```

Würde `function_name(x) <- value` "normal" ausgeführt werden, ergäbe dieser Befehl keinen Sinn, da `function_name(x)` kein gültiger Variablenname ist.

```

f <- function(x) x+5
f(x) <- 2 # ERROR
## Error in f(x) <- 2: could not find function "f<-

```

Zusätzliche Argumente werden zwischen `x` und `value` gelistet.

```

`modify<-` <- function(x, position, value) {
  x[position] <- value
  x
}
modify(x, 1) <- -10
x
## [1] -10 100 3 4 5
x <- `modify<-`(x, 2, -10)
x
## [1] -10 -10 3 4 5

```

Ersetzungsfunktionen können geschachtelt werden. Dabei spielen sowohl ``function_name<-`()` als auch `function_name()` eine Rolle. Die Übersetzung in Präfix-Notation wird ggf komplexer.

```

x <- 1:3
names(x) <- LETTERS[1:3]
x
## A B C
## 1 2 3
x <- `names<-`(x, letters[1:3]) # in Präfixnotation:
x
## a b c
## 1 2 3

x[2] <- 12
x
## a b c
## 1 12 3
x <- `[<-`(x, 2, 22) # in Präfixnotation:
x
## a b c
## 1 22 3

names(x)[2] <- "two"
names(x)
## [1] "a"   "two" "c"
# in Präfixnotation:
`*tmp*` <- x # erstelle Variable *tmp*
x <- `names<-`(`*tmp*`, `[<-`(`names(`*tmp*`), 2, "two"))
rm(`*tmp*`) # entferne Variable *tmp*

# allgemein
outside(inside(x, args_in), args_out) <- value

```

```
# in Präfixnotation:
`*tmp*` <- x
x <- `inside`<-`(
  `*tmp*`,
  args_in,
  `outside`<-`(
    inside(`*tmp*`, args_in),
    args_out,
    value
  )
)
rm(`*tmp*`)
```

## 2.4 Spezialformen

Alle Spezialformen haben auch eine Präfixform.

```
`(`(x) # (x)
`{(`x) # {x}
`[(`x, i) # x[i]
`[(`x, i) # x[[i]]
`if`(cond, true) # if (cond) true
`if`(cond, true, false) # if (cond) true else false
`for`(var, seq, action) # for(var in seq) action
`while`(cond, action) # while(cond) action
`repeat`(expr) # repeat expr
`next`() # next
`break`() # break
`function`(alist(arg1, arg2), body, env) # function(arg1, arg2) body
```

Ist die Präfixform einer Funktion bekannt, kann damit die Dokumentation aufgerufen werden. `?(`` ergibt einen Fehler. `?(`` ruft die Dokumentation auf.

Alle Funktionen mit Spezialform sind sogenannte primitive Funktionen, dazu später mehr.

```
`for`  
## .Primitive("for")
```

## 2.5 Funktionsaufruf mit Argumentliste

Um Argumente in einer Liste an eine Funktion zu übergeben, nutze `do.call()`.

```
x <- c(1, 5, NA, 7)
args <- list(x, na.rm = TRUE)
do.call(mean, args)
## [1] 4.333333
# entspricht
mean(x, na.rm = TRUE)
## [1] 4.333333
```

## 3 Klassifizierung von Funktionen

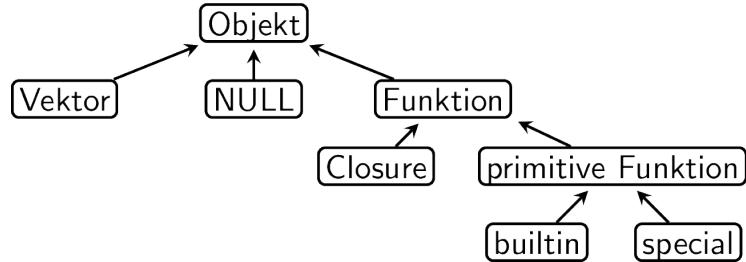
Die Klasse eines Funktions-Objektes ist `function`. Der Typ ist `closure`, `builtin` oder `special`.

```

c(class(mean), class(sum), class(`[`))
## [1] "function" "function" "function"
c(typeof(mean), typeof(sum), typeof(`[`))
## [1] "closure" "builtin" "special"

```

Funktionen vom Typ `builtin` oder `special` sind sogenannte **primitive Funktionen**. Sie existieren ausschließlich in Base-R. Es gibt nur rund 200 davon. Alle anderen Funktionen, insbesondere alle mit `function` erstellten Funktionen, sind Closures.



## 4 Komponenten einer Funktion

Closures bestehen aus drei Teilen: `formals()` (Argumente), `body()` und `environment()` (Umgebung). Der Funktionsname zählt nicht als Teil der Funktion.

```

f02 <- function(x) {
  # A comment
  x ^ 2
}
formals(f02)
## $x
body(f02)
## {
##   x^2
## }
environment(f02)
## <environment: R_GlobalEnv>

```

Die Funktionsumgebung (`environment()`) gibt an, wo die Funktion erstellt wurde. Dies ist wichtig dafür, wie Variablennamen ausgewertet werden und wird im Kapitel über Umgebungen ausführlich besprochen.

```

environment(mean)
## <environment: namespace:base>

```

Wie andere Objekte können auch Funktionen Attribute haben. Das Attribut `srcref` (*source reference*) gibt den Code an, mit dem die Funktion erstellt wurde. Dabei werden (im Gegensatz zu `body()`) auch Kommentare angegeben.

```

attr(f02, "srcref")
## function(x) {
##   # A comment
##   x ^ 2
## }

```

Primitive Funktionen haben keine `formals()`, `body()` oder `environment()`, da sie direkt kompilierten C-Code aufrufen.

```
typeof(sum) # primitive Funktion
## [1] "builtin"
formals(sum)
## NULL
body(sum)
## NULL
environment(sum)
## NULL
```

Um auf die Argument-Liste primitiver Funktionen zuzugreifen, nutze `args()`.

```
formals(args(sum))
## $...
##
##
## $na.rm
## [1] FALSE
```

## 5 Funktionsargumente

### 5.1 Lazy Evaluation 1

Argumente von Funktionen werden erst dann ausgewertet, wenn darauf zugegriffen wird. Dies wird als **lazy evaluation** bezeichnet.

```
stop("This is an error!") # erzeugt ERROR
## Error in eval(expr, envir, enclos): This is an error!
f <- function(x) {
  10
}
f(stop("This is an error!")) # kein Error, da x nicht ausgewertet
## [1] 10
```

Auch die logischen Operatoren (vektorisiert: `|`, `&`. skalar: `||`, `&&`) sind Funktionen. Ausnutzen lässt sich *lazy evaluation* nur für die skalaren Operatoren.

```
TRUE | stop("!")
## Error in eval(expr, envir, enclos): !
FALSE | stop("!")
## Error in eval(expr, envir, enclos): !
TRUE || stop("!")
## [1] TRUE
FALSE || stop("!")
## Error in eval(expr, envir, enclos): !
```

### 5.2 Default arguments

*Lazy evaluation* ermöglicht es Default-Argumente in Abhängigkeit von anderen Argumenten oder später definierten Variablen zu schreiben. Allerdings führt dies zu schwer verständlichem Code.

```
f <- function(x = 1, y = x * 2, z = a + b) {
  a <- 10
  b <- 100
  c(x, y, z)
}
```

```
f()
## [1] 1 2 110
```

Default-Argumente werden im Inneren der Funktion ausgewertet. Vom Benutzer übergebene Argumente werden in der äußeren Umgebung ausgewertet.

```
# ls() gibt die in der aktuellen Umgebung definierten Variablennamen aus
f <- function(x = ls()) {
  a <- 1
  x
}
f()
## [1] "a" "x"
f(ls())
## [1] "f"
```

### 5.3 Missing arguments

Die Funktion `missing()` gibt an, ob ein Argument beim Funktionsaufruf übergeben wurde. Insbesondere lässt sich unterscheiden, ob ein Argument vom Nutzer kommt oder der Default-Wert ist.

```
f <- function(x, y = 10) {
  list(missing_x=missing(x), missing_y=missing(y), y=y)
  # missing(z) # ERROR
}
str(f())
## List of 3
## $ missing_x: logi TRUE
## $ missing_y: logi TRUE
## $ y        : num 10
str(f(5))
## List of 3
## $ missing_x: logi FALSE
## $ missing_y: logi TRUE
## $ y        : num 10
str(f(y=10))
## List of 3
## $ missing_x: logi TRUE
## $ missing_y: logi FALSE
## $ y        : num 10
str(f(5,15))
## List of 3
## $ missing_x: logi FALSE
## $ missing_y: logi FALSE
## $ y        : num 15
```

Guter Stil: Ist ein Argument optional, sollte es einen Default-Wert haben. Dies ist leider selbst in Base-R nicht immer umgesetzt.

```
print(sample) # zeige R-Code der Funktion sample()
## function (x, size, replace = FALSE, prob = NULL)
## {
##   if (length(x) == 1L && is.numeric(x) && is.finite(x) && x >=
##     1) {
##     if (missing(size))
```

```

##           size <- x
##           sample.int(x, size, replace, prob)
##       }
##   else {
##       if (missing(size))
##           size <- length(x)
##       x[sample.int(length(x), size, replace, prob)]
##   }
## }
## <bytecode: 0x00000000120ed138>
## <environment: namespace:base>
args(sample) # x und size haben keine Default Werte
## function (x, size, replace = FALSE, prob = NULL)
## NULL
sample(1:5) # size wegzulassen ist jedoch möglich
## [1] 5 3 4 1 2
sample <- function(x, size = NULL, replace = FALSE, prob = NULL) { # besserer Stil
  if (is.null(size)) size <- length(x)
  x[sample.int(length(x), size, replace = replace, prob = prob)]
}

```

## 5.4 dot-dot-dot

Funktionen mit dem Sonderargument ... können beliebige zusätzliche Argumente übergeben werden.

Mittels ... wird auf das  $n$ -te Zusatzargument zugegriffen.

```

f <- function(...) {
  list(first = ..1, third = ..3)
}
str(f(1, 2, 3))
## List of 2
## $ first: num 1
## $ third: num 3

```

Im Inneren der Funktion können die Zusatzargumente mit ... weitergegeben werden.

```

f <- function(y, z) {
  list(y = y, z = z)
}
g <- function(x, ...) {
  f(...)
}
str(g(x = 1, y = 2, z = 3))
## List of 2
## $ y: num 2
## $ z: num 3

```

Mit list(...) werden die Zusatzargumente in einer Liste gespeichert und können dann (ggf mit Namen) aufgerufen werden.

```

f <- function(...) {
  list(...)
}
str(f(a = 1, b = 2))
## List of 2

```

```
## $ a: num 1
## $ b: num 2
```

Bei Funktionen der `apply()`-Familie kann wegen ... der übergebenen Funktion weitere Parameter mitgegeben werden.

```
x <- list(c(1, 3, NA), c(4, NA, 6))
sapply(x, mean, na.rm = TRUE)
## [1] 2 5
```

## 6 Beenden einer Funktion

Funktionsaufrufe können auf zwei Arten beendet werden: durch die Rückgabe eines Wertes oder durch ein Fehlerausgabe.

Rückgabe eines Wertes kann implizit geschehen (zuletzt ausgewerteter Ausdruck) oder explizit durch `return()`.

```
f <- function(x) {
  0
  if (x < 0) -1 else 1
}
f(-5)
## [1] -1
f(5)
## [1] 1
g <- function(x) {
  return(0)
  if (x < 0) return(-1) else return(1)
}
g(1)
## [1] 0
```

Normalerweise werden zurückgegebene Werte von Funktionsaufrufen auf der obersten Ebene (in der Konsole, nicht im Inneren von anderen Funktionen) ausgegeben. Dies kann jedoch mit `invisible()` verhindert werden.

```
f <- function() 1
f()
## [1] 1
g <- function() invisible(1)
g() # keine Ausgabe
print(g()) # Ausgabe erzwingen
## [1] 1
```

Die Funktion `~` entspricht der Identität `function(x) x`. Mit ihr wird `invisible()` aufgehoben.

```
(42)
## [1] 42
g()
(g())
## [1] 1
```

Alle Funktionen - auch die Sonderformen - haben einen Rückgabewert. ZT wird dieser unsichtbar zurückgegeben.

Falls spezielle Rückgabewerte nicht sinnvoll sind, wird meist `invisible(NULL)` zurückgegeben.

```
if (FALSE) 42
(if (FALSE) 42
## NULL
```

Die Funktion `<-` gibt ihr rechtes Argument unsichtbar zurück. Dies ermöglicht die Verkettung von Zuweisungen.

```
x <- 2
(x <- 2)
## [1] 2
x <- y <- z <- 3
c(x, y, z)
## [1] 3 3 3
```

Die Funktion `stop()` erzeugt eine Fehlerausgaben. Dabei wird die Funktion sofort beendet.

```
f <- function() {
  stop("I'm an error")
  print("HALLO!")
  return(10)
}
f()
## Error in f(): I'm an error
```

Nutze `on.exit()`, um Code bei jeglichem Verlassen der Funktion (mit Rückgabewert oder Error) auszuführen.

```
f <- function(x) {
  print("Hello")
  on.exit(print("Goodbye!"), add = TRUE)
  print("so...")
  if (x) return(10) else stop("Error")
  print("blub")
}
f(TRUE)
## [1] "Hello"
## [1] "so..."
## [1] "Goodbye!"
## [1] 10
f(FALSE)
## [1] "Hello"
## [1] "so..."
## Error in f(FALSE): Error
## [1] "Goodbye!"
```

# V05 – stringr

3. Mai 2021

## Contents

<b>1</b>	<b>Intro</b>	<b>1</b>
<b>2</b>	<b>Basics</b>	<b>1</b>
2.1	Weiteres . . . . .	2
<b>3</b>	<b>Formatieren</b>	<b>3</b>
<b>4</b>	<b>Reguläre Ausdrücke</b>	<b>4</b>
4.1	Funktionen zur Nutzung von RegEx . . . . .	5
4.2	RegEx Syntax . . . . .	7
<b>5</b>	<b>Weiteres</b>	<b>10</b>

## 1 Intro

Zur Manipulation von Strings dient das Paket **stringr**. Es ist Teil der Paketsammlung **tidyverse**.

```
library(stringr)
# oder
# library(tidyverse) # lädt stringr und ein paar weitere Pakete
```

Funktionen in **stringr** zur Manipulation von Strings beginnen mit **str\_**.

## 2 Basics

Einige Funktionen aus Base-R haben ein Äquivalent in **stringr**.

```
# Base-R: nchar()
str_length(c("a", "abs", NA, ""))
## [1] 1 3 NA 0

# Base-R: paste0()
str_c("Apfel", "saft", "schorle", 3, TRUE, pi)
## [1] "Apfelsaftschorle3TRUE3.14159265358979"
str_c("Apfel", "saft", "schorle", 3, TRUE, pi, sep = ", ")
## [1] "Apfel, saft, schorle, 3, TRUE, 3.14159265358979"
```

**str\_c()** ist vektorisiert, dh die Funktion wirkt auf Vektoren elementweise. Ggf findet Recycling statt.

```
str_c(1:3, LETTERS[1:3])
## [1] "1A" "2B" "3C"
str_c("prefix-", LETTERS[1:3], "-suffix")
## [1] "prefix-A-suffix" "prefix-B-suffix" "prefix-C-suffix"
str_c("prefix-", LETTERS[1:3], "-suffix", collapse = " * ")
```

```
## [1] "prefix-A-suffix * prefix-B-suffix * prefix-C-suffix"
str_c(1:3, LETTERS[1:3], T, collapse = " * ", sep=",")
## [1] "1,A,TRUE * 2,B,TRUE * 3,C,TRUE"
```

Zum Subsetting der Symbole eines Strings benötigt es eigene Funktionen, da `[[` und `[` schon eine andere Funktion haben (`character`-Vektoren subsetten). `stringr` bietet hierzu die Funktion `str_sub()`.

Die Symbole eines Strings `s` haben analog zu Vektoren die Indizes `1:str_length(s)`.

```
x <- c("Apple", "Banana", "Pear")
str_sub(x, start=1, end=3)
## [1] "App" "Ban" "Pea"
str_sub(x, 1, 3)
## [1] "App" "Ban" "Pea"
str_sub(x, 3) # only start set => subset until end of string
## [1] "ple" "nana" "ar"
str_sub(x, end=3) # only end set => subset from beginning of string
## [1] "App" "Ban" "Pea"
str_sub(x, -3, -1) # negativ numbers: count from end of string
## [1] "ple" "ana" "ear"
str_sub("ab", 1, 5) # values > str_length() act same as = str_length()
## [1] "ab"
```

`str_subs()` ist in jedem Argument vektorisiert. Ggf geschieht Recycling.

```
str_sub(x, 1:3, 3:5) # start and end are also vectorized
## [1] "App" "ana" "ar"
str_sub("123456", c(1,3), 6:3) # all arguments are repeated to match longest argument
## [1] "123456" "345"    "1234"   "3"
```

Analog zu `[<-` kann mit `str_sub<-` Subsetting mit Zuweisung durchgeführt werden.

```
str_sub(x, 1, 1) <- "_"
x
## [1] "_pple"  "_anana" "_ear"
str_sub(x, 1:3, 4) <- c("XX", "YY", "ZZ") # vectorized assignment with non-matching string lengths
x
## [1] "XXe"    "_YYna"  "_eZZ"
```

## 2.1 Weiteres

```
# Base-R: trimws()
str_trim(c(" a a", "b b ", " c c ")) # remove whitespace from beginning and end
## [1] "a a" "b b" "c c"
str_squish(c(" a a", "b b ", " c c ")) # also reduce repeated whitespace inside
## [1] "a a" "b b" "c c"

# Base-R: tolower(), toupper()
x <- c("Apple", "BANANA!", "pear")
str_to_upper(x)
## [1] "APPLE"  "BANANA!" "PEAR"
str_to_lower(x)
## [1] "apple"  "banana!" "pear"
str_to_title(x)
## [1] "Apple"  "Banana!" "Pear"
```

Was `rep(times=)` für Vektoren ist, ist `str_dup()` für Strings.

```
str_dup(c("a", "bB"), 2)
## [1] "aa"    "bBbB"
str_dup(c("a", "bB"), 2:3)
## [1] "aa"    "bBbBbB"
```

### 3 Formatieren

Um Werte in Strings zu schreiben, setzt `stringr` auf das Paket `glue`, <https://glue.tidyverse.org/>, welches auch Teil des Tidyverse ist, aber nicht extra geladen werden muss (wenn `stringr` geladen ist).

Teile eines Strings, die von geschweiften Klammern `{ }` umschlossen sind, werden mittels `glue` als Ausdrücke in R ausgewertet.

```
x <- sample.int(100, 2)
str_glue("3 * 4 = {3*4}")
## 3 * 4 = 12
str_glue("Die Summe von {x[1]} und {x[2]} ist {sum(x)} .")
## Die Summe von 45 und 23 ist 68.
str_glue("Der glue-Text '{'}'Wort'{'}' erzeugt: {'Wort'}")
## Der glue-Text '{'Wort'}' erzeugt: Wort
```

Der Rückgabewert von `str_glue()` ist ein `character`-Vektor der S3-Klasse `glue`. Dies sorgt für eine etwas andere Ausgabe auf der Konsole.

```
word <- "blub"
word_glue <- str_glue(word)
word == word_glue
## [1] TRUE
identical(word, word_glue)
## [1] FALSE
word
## [1] "blub"
word_glue
## blub
typeof(word_glue)
## [1] "character"
attributes(word_glue)
## $class
## [1] "glue"      "character"
unclass(word_glue)
## [1] "blub"
```

Um die Ausgabe einer Dezimalzahl zu formatieren, nutze die Base-R-Funktion `format()`, siehe `?format` zusammen mit `str_glue()` oder die Base-R-Funktion `sprintf()`.

```
sprintf("Pi ist %.f.", pi)
## [1] "Pi ist 3.141593."
sprintf("Pi ist %.2f.", pi)
## [1] "Pi ist 3.14."
str_glue("Pi ist {pi}.")
## Pi ist 3.14159265358979.
format(pi, digits=3)
## [1] "3.14"
str_glue("Pi ist {format(pi, digits=3)}.")
```

```
## Pi ist 3.14.

str_glue() ist vektorisiert.

descr <- c('gerade', 'ungerade')
num <- sample(5)
str_glue("Die Zahl {num} ist {descr[num %% 2 + 1]}.")
## Die Zahl 4 ist gerade.
## Die Zahl 3 ist ungerade.
## Die Zahl 1 ist ungerade.
## Die Zahl 2 ist gerade.
## Die Zahl 5 ist ungerade.
```

`str_glue_data()` wird an erster Position ein Tibble übergeben, dessen Spaltennamen als Variablen in den Ausdrücken des Strings genutzt werden können.

```
library(tibble)
tb <- tibble(x=1:3, y=letters[1:3])
str_glue_data(tb, "Variable x ist {x} und y ist {y}.")
## Variable x ist 1 und y ist a.
## Variable x ist 2 und y ist b.
## Variable x ist 3 und y ist c.
```

## 4 Reguläre Ausdrücke

Mit regulären Ausdrücken (*regular expressions*, **RegEx**) wird in Strings nach Mustern gesucht.

Mit `str_view()` wird die erste Übereinstimmung des Musters im String angezeigt, mit `str_view_all()` alle (nicht überlappend).

```
x <- c("apple", "banana", "mango")
str_view(x, "an")
## PhantomJS not found. You can install it with webshot::install_phantomjs(). If it is installed, please
```

In regulären Ausdrücken haben einige Symbole spezielle Bedeutung. Zu diesen sogenannten **Metazeichen** zählen `[ ]` `( )` `{ }` `|` `?` `+` `-` `*` `^` `$` `\ .` `:` `!` `<` `=`.

ZB steht `|` für *oder* und `.` für ein beliebiges Zeichen. Die genaue Funktionsweise dieser und anderer Metazeichen wird weiter unten erklärt.

```
x <- c("apple", "banana", "mango", "n|m ch|go |an")
str_view_all(x, "n|m")
```

Sollen die Metazeichen als normale Symbole interpretiert werden, müssen sie mit `\` markiert werden (*escaping*). Dabei ist `\` selbst ein Sonderzeichen für Strings in R. Um `\` in einem String zu schreiben, muss "`\\`" geschrieben werden.

```
cat("I\\I\\n")
## I\I
x <- "a.|.|.|b"
str_view(x, ".")
```

`\` ist sowohl für R-Strings als auch für RegEx ein Metazeichen und muss jeweils *escaped* werden, wenn nach dem Symbol `\` gesucht werden soll.

```
cat("I\\\\I\\n")
## I\I
cat("I\\\\\\I\\n")
```

```
## I\I
x <- "a\b"
cat(x, "\n")
## a\b
str_view(x, "\\")
## Error in stri_locate_first_regex(string, pattern, opts_regex = opts(pattern)): Unrecognized backslash
str_view(x, "\\\\")
```

R Version 4 bietet die Möglichkeit mit `r"(...)"` raw Strings zu schreiben, bei denen \ nicht als Metazeichen interpretiert wird.

```
x <- r"(a\b)"
cat(x, "\n")
## a\b
str_view(x, r"(\\"))
```

## 4.1 Funktionen zur Nutzung von RegEx

- `str_detect()`: Enthält der String das Muster?
- `str_subset()`: Gib alle Strings zurück, die das Muster enthalten.
- `str_which()`: Gib die Indizes der Strings zurück, die das Muster enthalten.
- `str_count()`: Wie oft passt das Muster (nicht-überlappenden) in den String?

```
x <- c("apple", "banana", "mango", "lime")
str_detect(x, ".n")
## [1] FALSE  TRUE  TRUE FALSE
str_subset(x, ".n")
## [1] "banana" "mango"
str_which(x, ".n")
## [1] 2 3
str_count(x, ".n")
## [1] 0 2 1 0
str_count(x, ".n.") # nicht überlappend
## [1] 0 1 1 0
```

- `str_extract()`: Gib die gefundenen Muster aus.
- `str_locate()`: Gib Start- und Endindex der gefundenen Muster aus.

Dazu gibt es die entsprechenden `_all`-Funktionen. *All* ist dabei immer nicht-überlappend gemeint.

```
x <- c("apple", "banana", "mango", "lime")
str_extract(x, "a.")
## [1] "ap" "an" "an" NA
str(str_extract_all(x, "a."))
## List of 4
## $ : chr "ap"
## $ : chr [1:2] "an" "an"
## $ : chr "an"
## $ : chr(0)
str_locate(x, "a.")
##      start end
## [1,]      1   2
## [2,]      2   3
## [3,]      2   3
## [4,]     NA  NA
str_locate_all(x, "a.")
```

```

## [[1]]
##      start end
## [1,]    1    2
##
## [[2]]
##      start end
## [1,]    2    3
## [2,]    4    5
##
## [[3]]
##      start end
## [1,]    2    3
##
## [[4]]
##      start end

```

`str_split(str, pattern)` spaltet einen String `str` an allen Stellen, wo `pattern` gefunden wird.

```

x <- c("Kaffee, Schokolade, Kekse", "Orange, Melone")
str_split(x, ", ")
## [[1]]
## [1] "Kaffee"      "Schokolade" "Kekse"
##
## [[2]]
## [1] "Orange" "Melone"
x <- "asdf_7,jklö qwert"
str_split(x, "_|,| ") # split at _ and , and space
## [[1]]
## [1] "asdf"    "7"      "jklö"   "qwert"

```

Möchte man die Interpretation des Musters als RegEx abschalten, kann der String mit `fixed()` markiert werden.

```

str_split("ab.cde.f", ".")
## [[1]]
## [1] " " " " " " " "
str_split("ab.cde.f", fixed("."))
## [[1]]
## [1] "ab"  "cde" "f"

```

`fixed()` kann jedem `pattern`-Argument einer `stringr` Funktion angewendet werden.

```

x <- fixed("asdf")
class(x)
## [1] "fixed"      "pattern"    "character"
attr(x, "options")
## $case_insensitive
## [1] FALSE
x <- fixed("asdf", ignore_case = TRUE)
attr(x, "options")
## $case_insensitive
## [1] TRUE

str_view_all("xa.bxA.bxa.BxA.Bxaobx", fixed("a.b", ignore_case=T))

```

## 4.2 RegEx Syntax

Die Funktionsweise regulärer Ausdrücke mit `stringr` wird gut zusammengefasst auf Seite 2 des offiziellen Cheatsheets: <https://github.com/rstudio/cheatsheets/raw/master/strings.pdf>.

### 4.2.1 Match Character

#### MATCH CHARACTERS

string (type regexp this)	matches (which matches this)	example
<code>a</code> (etc.)	a (etc.)	<code>see("a")</code>
<code>\.</code>	.	<code>see("\.")</code>
<code>\!</code>	!	<code>see("\!")</code>
<code>\?</code>	?	<code>see("\?")</code>
<code>\\\</code>	\	<code>see("\\\\")</code>
<code>\(</code>	(	<code>see("\(")</code>
<code>\)</code>	)	<code>see("\)")</code>
<code>\{</code>	{	<code>see("\{")</code>
<code>\}</code>	}	<code>see("\}")</code>
<code>\n</code>	new line (return)	<code>see("\n")</code>
<code>\t</code>	tab	<code>see("\t")</code>
<code>\s</code>	any whitespace ( <code>\\$</code> for non-whitespaces)	<code>see("\s")</code>
<code>\d</code>	any digit ( <code>\D</code> for non-digits)	<code>see("\d")</code>
<code>\w</code>	any word character ( <code>\W</code> for non-word chars)	<code>see("\w")</code>
<code>\b</code>	word boundaries	<code>see("\b")</code>
<code>[:digit:]</code> <sup>1</sup>	digits	<code>see("[[:digit:]]")</code>
<code>[:alpha:]</code> <sup>1</sup>	letters	<code>see("[[:alpha:]]")</code>
<code>[:lower:]</code> <sup>1</sup>	lowercase letters	<code>see("[[:lower:]]")</code>
<code>[:upper:]</code> <sup>1</sup>	uppercase letters	<code>see("[[:upper:]]")</code>
<code>[:alnum:]</code> <sup>1</sup>	letters and numbers	<code>see("[[:alnum:]]")</code>
<code>[:punct:]</code> <sup>1</sup>	punctuation	<code>see("[[:punct:]]")</code>
<code>[:graph:]</code> <sup>1</sup>	letters, numbers, and punctuation	<code>see("[[:graph:]]")</code>
<code>[:space:]</code> <sup>1</sup>	space characters (i.e. <code>\s</code> )	<code>see("[[:space:]]")</code>
<code>[:blank:]</code> <sup>1</sup>	space and tab (but not new line)	<code>see("[[:blank:]]")</code>
.	every character except a new line	<code>see(".")</code>

<sup>1</sup> Many base R functions require classes to be wrapped in a second set of `[ ]`, e.g. `[:digit:]`

```
str_view_all("abc123ABC", "\d\d")
str_view_all("aÁá? ! :)", "[alpha:]")
```

#### 4.2.2 Alternates

##### ALTERNATES

alt <- function(rx) str\_view\_all("abcde", rx)

regexp	matches	example
ab d	or	alt("ab d")
[abe]	one of	alt("[abe]")
[^abe]	anything but	alt("[^abe]")
[a-c]	range	alt("[a-c]")

```
str_view_all(" a a b b", "a a|b ")
```

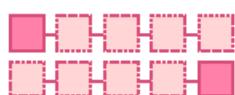
```
x <- c("apple", "banana", "mango", "lime")
str_view_all(x, "[aeiou] [^aeiou] [aeiou]") # Vokal, kein Vokal, Vokal
```

```
str_view_all("aÁá", "[a-zA-Z]")
```

#### 4.2.3 Anchors

##### ANCHORS

anchor <- function(rx) str\_view\_all("aaa", rx)



regexp	matches	example
^a	start of string	anchor("^a")
a\$	end of string	anchor("a\$")

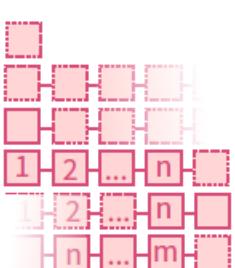
```
str_view_all(c("aba", "bab ab"), "^ab")
```

Beachte: ^ hat innerhalb von [, ] eine andere Bedeutung als außerhalb.

#### 4.2.4 Quantifiers

##### QUANTIFIERS

quant <- function(rx) str\_view\_all(".a.aa.aaa", rx)



regexp	matches	example
a?	zero or one	quant("a?")
a*	zero or more	quant("a*")
a+	one or more	quant("a+")
a{n}	exactly n	quant("a{2}")
a{,n}	n or more	quant("a{2,}")
a{n,m}	between n and m	quant("a{2,4}")

Quantoren geben an, wie oft der vorangegangene Ausdruck auftauchen muss.

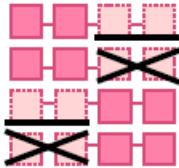
```
x <- c("apple", "banana", "mango", "lime")
str_view(x, "p+")
```

Quantoren sind *gierig*, dh es wird ein Teil-String maximaler Länge ausgewählt. Wird einem Quantor ein ? nachgestellt, ist der vorausgegangene Quantor *genügsam* (Teil-String minimaler Länge auswählen).

```
str_view(c("oABxxBxBo", "oAxBxBxBo"), "A.*B")
```

#### 4.2.5 Look Arounds

##### LOOK AROUND



look <- function(rx) str\_view\_all("bacad", rx)

regexp	matches	example
a(?=c)	followed by	look("a(?=c)") bacad
a(?!c)	not followed by	look("a(?!c)") bacad
(?<=b)a	preceded by	look("(?<=b)a") bacad
(?<!b)a	not preceded by	look("(?<!b)a") bacad

Für **Look Arounds** müssen vor oder nach dem gesuchten Muster bestimmte Symbole oder Muster vorhanden sein. Diese sind aber nicht Teil des gefundenen Muster.

```
str_view(x, "[aeiou] [^aeiou] .*$")
```

#### 4.2.6 Groups

##### GROUPS

ref <- function(rx) str\_view\_all("abbaab", rx)

Use parentheses to set precedent (order of evaluation) and create groups

regexp	matches	example
(ab d)e	sets precedence	alt("(ab d)e") abcde

Use an escaped number to refer to and duplicate parentheses groups that occur earlier in a pattern. Refer to each group by its order of appearance

string	regexp	matches	example
(type this)	(to mean this)	(which matches this)	(the result is the same as ref("abba"))
\1	\1 (etc.)	first () group, etc.	ref("(a)(b)\1\2\1") abbaab

Einzelne Bestandteile eines regulären Ausdrucks können mit ( ) zu einer Einheit **gruppiert** werden.

```
x <- c("apple", "banana", "mango", "lime")
str_view(x, "(an) +")
```

str\_match() und str\_match\_all() geben neben dem String-Teil, der dem gesamten Muster entspricht, auch diejenigen Teile aus, die den einzelnen Gruppen entsprechen.

```
x <- c("apple", "banana", "mango", "lime")
str_match(x, "(a)(.*)([aeiou])")
## [,1] [,2] [,3] [,4]
## [1,] "apple" "a"   "pp"  "e"
## [2,] "anana" "a"   "na"  "a"
## [3,] "ango"   "a"   "ng"  "o"
## [4,] NA      NA    NA   NA
```

Diese Mechanik wird beim Ersetzen mit RegEx genutzt. str\_replace() und str\_replace\_all() ersetzen gefundene Muster mit neuen Werten. Diese Ersetzungen können mittels \1, \2, \3, ... auf die Werte der gefundenen Gruppen verweisen.

```
str_replace("banana", "a", "o")
## [1] "bonana"
str_replace_all("banana", "a", "o")
## [1] "bonono"
str_replace_all("banana", "a(.)", "a\\1\\1")
## [1] "bannanna"
```

```
str_replace("1 * (2 + 3); (-:", "\\\(([^\\()]*))\\)", "[\\1]")
## [1] "1 * [2 + 3]; (-:"
```

Als `replacement` kann `str_replace()` auch eine Funktionen übergeben werden, die den gefundenen Teil-String in seine Ersetzung umwandelt.

```
colours <- str_c("\\b", colors(), "\\b", collapse="|")
col2hex <- function(col) {
  rgb <- col2rgb(col)
  rgb(rgb["red", ], rgb["green", ], rgb["blue", ], max = 255)
}

cat(str_sub(colours, end=50))
## \bwhite\b/\baliceblue\b/\bantiquewhite\b/\bantique
col2hex("orange")
## [1] "#FFA500"

x <- c(
  "Roses are red, violets are blue.",
  "My favorite color is green.")
str_replace_all(x, colours, col2hex)
## [1] "Roses are #FF0000, violets are #0000FF."
## [2] "My favorite color is #00FF00."
```

## 5 Weiteres

CheatSheet <https://github.com/rstudio/cheatsheets/raw/master/strings.pdf>

Auf der Seite <https://regexpr.com/> lassen sich RegEx ausprobieren. Leider werden nicht alle oben genannten Möglichkeiten unterstützt.

Viele Texteditoren und IDEs (zB RStudio) unterstützen RegEx für Such- und Ersetz-Funktion.

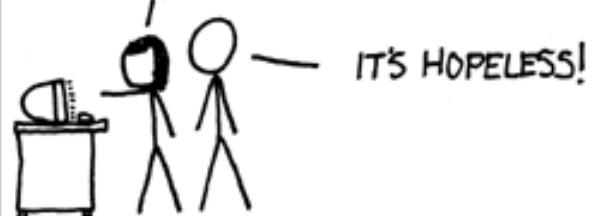
<https://xkcd.com/208/>

WHENEVER I LEARN A  
NEW SKILL I CONCOCT  
ELABORATE FANTASY  
SCENARIOS WHERE IT  
LETS ME SAVE THE DAY.

OH NO! THE KILLER  
MUST HAVE FOLLOWED  
HER ON VACATION!



BUT TO FIND THEM WE'D HAVE TO SEARCH  
THROUGH 200 MB OF EMAILS LOOKING FOR  
SOMETHING FORMATTED LIKE AN ADDRESS!



EVERYBODY STAND BACK.



I KNOW REGULAR  
EXPRESSIONS.



\ BACKSLASH  
\\ REAL BACKSLASH  
/// REAL REAL BACKSLASH  
\\\\ ACTUAL BACKSLASH, FOR REAL THIS TIME  
\\\\\\\\ ELDER BACKSLASH  
\\\\\\\\\\\\ BACKSLASH WHICH ESCAPES THE SCREEN AND ENTERS YOUR BRAIN  
\\\\\\\\\\\\\\\\ BACKSLASH SO REAL IT TRANSCENDS TIME AND SPACE  
\\\\\\\\\\\\\\\\\\\\ BACKSLASH TO END ALL OTHER TEXT  
\\\\\\\\\\\\\\\\\\\\\\\\... THE TRUE NAME OF BA'AL, THE SOUL-EATER

# V06 – tibble

3. Mai 2021

## Contents

<b>1</b>	<b>Tabellen</b>	<b>1</b>
1.1	nycflights13	5
<b>2</b>	<b>Relationale Datenbanken</b>	<b>6</b>
2.1	Definitionen der Grundbegriffe	6
2.2	Operationen	7

## 1 Tabellen

Ein **Data-Frame** ist eine tabellenförmige Datenstruktur.

In Base-R wird sie in S3-Objekten der Klasse `data.frame` gespeichert.

```
df <- data.frame(x=1:3, y=letters[1:3])
df
##   x y
## 1 1 a
## 2 2 b
## 3 3 c
class(df)
## [1] "data.frame"
```

Zeilen nennen wir auch **Beobachtungen**, Spalten nennen wir auch **Variablen**.

Einträge in verschiedenen Spalten können unterschiedlichen Typ haben.

Meist ist eine Spalte ein atomarer Vektor. Listen-, Matrix- und Data-Frame-Spalten sind prinzipiell möglich.

Im Zusammenhang mit Funktionen der Paketsammlung *Tidyverse* ist es üblich Data-Frames in Form von **Tibbles** zu nutzen.

Ein Tibble ist ein R-Objekt der Klasse `c("tbl_df", "tbl", "data.frame")`. Es baut also auf einem `data.frame`-Objekt auf.

### Bemerkung:

- Tabelle ist ein konzeptioneller Begriff, keine spezielle Datenstruktur in R.
- Jedes Data-Frame (und damit auch jedes Tibble) repräsentiert eine Tabelle.
- Es gibt weitere Datenstrukturen in R, die Tabellen repräsentieren, zB `data.table` (was nicht unbedingt zur besseren Unterscheidung der Begriffe beiträgt).

Um Tibbles zu nutzen, muss das Paket `tibble` geladen werden. Es gehört zum Tidyverse.

```
library(tibble) # oder library(tidyverse)
tb <- tibble(x=1:3, y=letters[1:3])
tb
## # A tibble: 3 x 2
```

```

##      x y
## <int> <chr>
## 1     1 a
## 2     2 b
## 3     3 c
class(tb)
## [1] "tbl_df"     "tbl"        "data.frame"

```

Bei der Ausgabe eines Tibbles auf der Konsole wird unter dem Spaltennamen der Typ der Spalte in Kurzform angezeigt.

```

tibble(logical=F, integer=0L, double=0, character="", list=list(1))
## # A tibble: 1 x 5
##   logical integer double character list
##   <lgl>     <int> <dbl> <chr>    <list>
## 1 FALSE        0     0   ""      <dbl [1]>

```

Mit `as_tibble()` werden passende Listen, Matrizen und Data-Frames in Tibbles konvertiert.

```

lst <- list(x=1:2, y=letters[1:2])
str(lst)
## List of 2
## $ x: int [1:2] 1 2
## $ y: chr [1:2] "a" "b"
as_tibble(lst)
## # A tibble: 2 x 2
##   x     y
##   <int> <chr>
## 1     1 a
## 2     2 b

mat <- matrix(1:4, ncol=2)
colnames(mat) <- c("A", "B")
mat
##      A B
## [1,] 1 3
## [2,] 2 4
as_tibble(mat)
## # A tibble: 2 x 2
##   A     B
##   <int> <int>
## 1     1     3
## 2     2     4

df
##   x y
## 1 1 a
## 2 2 b
## 3 3 c
as_tibble(df)
## # A tibble: 3 x 2
##   x     y
##   <int> <chr>
## 1     1 a
## 2     2 b

```

```
## 3      3 c
```

Mit `add_column()` und `add_row()` werden Spalten bzw Zeilen zu einem Tibble hinzugefügt.

```
tb <- tibble(x = 1:3, y = 3:1)

add_column(tb, z = -1:1, w = 0)
## # A tibble: 3 x 4
##       x     y     z     w
##   <int> <int> <int> <dbl>
## 1     1     3    -1     0
## 2     2     2     0     0
## 3     3     1     1     0
add_column(tb, z = -1:1, .before = "y")
## # A tibble: 3 x 3
##       x     z     y
##   <int> <int> <int>
## 1     1    -1     3
## 2     2     0     2
## 3     3     1     1

add_row(tb, x = 4:5, y = 0:-1)
## # A tibble: 5 x 2
##       x     y
##   <int> <int>
## 1     1     3
## 2     2     2
## 3     3     1
## 4     4     0
## 5     5    -1
add_row(tb, x = 4, y = 0, .before = 2)
## # A tibble: 4 x 2
##       x     y
##   <dbl> <dbl>
## 1     1     3
## 2     4     0
## 3     2     2
## 4     3     1
add_row(tb, x = 4) # set NA if needed
## # A tibble: 4 x 2
##       x     y
##   <dbl> <int>
## 1     1     3
## 2     2     2
## 3     3     1
## 4     4    NA
```

Die Subsetting-Operatoren (ggf mit Zuweisung) `[`, `[[`, `$`, `[<-`, `[[<-`, `$<-` können ebenso benutzt werden.

```
tb <- tibble(x = 1:3)
tb$y <- 3:1
tb[3, 2] <- 12
tb[c(1,3), ]
## # A tibble: 2 x 2
##       x     y
```

```

##  <int> <int>
## 1    1    3
## 2    3   12
tb[tb$y > 2, ]
## # A tibble: 2 x 2
##   x     y
##   <int> <int>
## 1    1    3
## 2    3   12
tb[[1]] <- letters[1:3]
tb[["x"]]
## [1] "a" "b" "c"

```

Spaltennamen sollten den Regeln für Variablennamen folgen, damit ein einfacher Zugriff möglich ist.

```

# Negativ-Beispiel: Spaltenname "0"
tb["0"] <- 1:3
# tb$0 # ERROR
tb$"0"
## [1] 1 2 3
tb$`0` # alternativ
## [1] 1 2 3

```

Die Funktionen `head()` und `tail()` sind alternative Notationen für Subsetting der ersten oder letzten Zeilen.

```

tb <- tibble(pos = 1:26, lower = letters, upper = LETTERS)
head(tb, 4)
## # A tibble: 4 x 3
##   pos lower upper
##   <int> <chr> <chr>
## 1    1     a     A
## 2    2     b     B
## 3    3     c     C
## 4    4     d     D
tail(tb, 4)
## # A tibble: 4 x 3
##   pos lower upper
##   <int> <chr> <chr>
## 1    23    w     W
## 2    24    x     X
## 3    25    y     Y
## 4    26    z     Z

```

Mit `bind_rows()`, `bind_cols()` fügen wir mehrere Tibbles gleicher Struktur zusammen. Im Gegensatz zu den vorigen sind diese Funktionen im Paket `dplyr` (auch Tidyverse).

```

t1 <- tibble(x=1:2, y=letters[1:2])
t2 <- tibble(x=11:12, y=letters[11:12])

dplyr::bind_rows(t1, t2)
## # A tibble: 4 x 2
##   x     y
##   <int> <chr>
## 1    1     a
## 2    2     b
## 3   11     k

```

```

## 4    12 1

t3 <- tibble(x=11:12, z=0:-1)
dplyr::bind_rows(t1, t3) # set NA if needed
## # A tibble: 4 x 3
##       x     y     z
##   <int> <chr> <int>
## 1     1     a     NA
## 2     2     b     NA
## 3    11 <NA>     0
## 4    12 <NA>    -1

t4 <- tibble(u = c(T,F), v = c(pi, exp(1)))
dplyr::bind_cols(t1, t4)
## # A tibble: 2 x 4
##       x     y     u     v
##   <int> <chr> <lgl> <dbl>
## 1     1     a     TRUE  3.14
## 2     2     b    FALSE  2.72

```

## 1.1 nycflights13

Neben Funktionen stellen manche Pakete auch Datasets bereit.

Das Paket `nycflights13` enthält mehrere Tibbles. Das Tibble `flights` enthält Informationen zu allen Flügen von New York City im Jahr 2013.

```

#install.packages("nycflights13")
library(nycflights13)
flights
## # A tibble: 336,776 x 19
##       year month   day dep_time sched_dep_time dep_delay arr_time sched_arr_time
##   <int> <int> <int>    <int>          <int>    <dbl>    <int>          <int>
## 1  2013     1     1      517          515      2     830          819
## 2  2013     1     1      533          529      4     850          830
## 3  2013     1     1      542          540      2     923          850
## 4  2013     1     1      544          545     -1    1004         1022
## 5  2013     1     1      554          600     -6     812          837
## 6  2013     1     1      554          558     -4     740          728
## 7  2013     1     1      555          600     -5     913          854
## 8  2013     1     1      557          600     -3     709          723
## 9  2013     1     1      557          600     -3     838          846
## 10 2013     1     1      558          600     -2     753          745
## # ... with 336,766 more rows, and 11 more variables: arr_delay <dbl>,
## #   carrier <chr>, flight <int>, tailnum <chr>, origin <chr>, dest <chr>,
## #   air_time <dbl>, distance <dbl>, hour <dbl>, minute <dbl>, time_hour <dttm>

```

Für nähere Informationen zu den Spalten (zB Einheiten) siehe `?flights`.

Bei großen Tibbles werden nur die ersten Zeilen und Spalten ausgegeben.

Wir setzen ein paar Optionen, um die Ausgabe hier kompakter zu gestalten.

```

options(
  tibble.print_min=6,
  tibble.print_max=6,

```

```

tibble.max_extra_cols=0)
flights
## # A tibble: 336,776 x 19
##   year month   day dep_time sched_dep_time dep_delay arr_time sched_arr_time
##   <int> <int> <int>   <int>           <int>     <dbl>   <int>           <int>
## 1 2013     1     1     517         515        2.00    830         819
## 2 2013     1     1     533         529        4.00    850         830
## 3 2013     1     1     542         540        2.00    923         850
## 4 2013     1     1     544         545       -1.00   1004        1022
## 5 2013     1     1     554         600       -6.00    812         837
## 6 2013     1     1     554         558       -4.00    740         728
## # ... with 336,770 more rows

```

Oft sind Daten eines Datasets über mehrere Tabellen (Tibbles) unterschiedlicher Struktur verteilt.

Das Paket `nycflights13` enthält neben `flights` noch die Tabellen `airlines`, `airports`, `planes`, `weather`.

Die Tabelle `flights` enthält eine Variable `carrier` – die 2-Buchstaben-Abkürzung der Airline. In der Tabelle `airlines` ist der Langname der Airline angegeben.

```

flights$carrier[1:10]
## [1] "UA" "UA" "AA" "B6" "DL" "UA" "B6" "EV" "B6" "AA"
airlines
## # A tibble: 16 x 2
##   carrier name
##   <chr>   <chr>
## 1 9E      Endeavor Air Inc.
## 2 AA      American Airlines Inc.
## 3 AS      Alaska Airlines Inc.
## 4 B6      JetBlue Airways
## 5 DL      Delta Air Lines Inc.
## 6 EV      ExpressJet Airlines Inc.
## # ... with 10 more rows

```

Dieses Dataset wird uns in den folgenden Kapiteln als Beispiel dienen.

## 2 Relationale Datenbanken

Die Theorie der **Relationalen Datenbanken** mit der **relationalen Algebra**, stellt Beziehungen von Tabellen in Datasets und Operationen auf Datasets auf formale Beine.

In diesem Abschnitt schneiden wir nur sehr knapp diese Themen an und vereinfachen an vielen Stellen. Mit dem vollen Umfang des Themenfelds können mehrere eigene Kurse gefüllt werden.

Die Nomenklatur in der Datenbank-Theorie unterscheidet sich etwas von der traditionell im Zusammenhang mit R verwendeten.

R / allgemein	Dataset	Tabelle	Spalte	Zeile
R (alternativ)		Tibble, Data-Frame	Variable	Beobachtung
DB-Theorie	Datenbank	Relation	Attribut	Tupel

### 2.1 Definitionen der Grundbegriffe

Eine endliche Folge von Mengen  $(W_1, \dots, W_m)$  heißt **Relationenschema**.

Eine endliche Teilmenge  $R \subseteq W_1 \times \dots \times W_m$  heißt **Relation** des Relationenschemas  $(W_1, \dots, W_m)$ . In diesem

Zusammenhang heißen die  $W_j$  auch **Wertebereiche**.

Ein Element  $t \in R$  einer Relation  $R$  heißt **Tupel**.

**Bemerkungen:**

1. Relationen entsprechen in etwa Tabellen und Tupel den Zeilen. Allerdings gibt es Unterschiede:
  - Tabellen haben eine Ordnung ihrer Zeilen, Relationen sind ungeordnete Mengen.
  - In einer Tabelle kann eine Zeile mehrere Male vorkommen, in einer Relation nicht (Menge).
2. In der Datenbank-Theorie gehört in ein Relationenschema noch jeweils ein Name (Spaltennamen) zu jedem Wertebereich  $D_j$ . Zur Vereinfachung lassen wir jedoch hier Namen weg.

Wir identifizieren **Spalten** (genannt Attribut, hat nichts mit Attributen von R-Objekten zu tun) durch ihren Spaltenindex  $j \in \{1, \dots, m\}$ .

Die "Zeilenzahl"  $n$  einer Relation  $R$  ist also die Anzahl der ihrer Elemente  $n = |R|$ .

Die "Spaltenzahl"  $m$  einer Relation  $R$  ist die Anzahl der Wertebereiche im zugehörigen Relationenschema.

**Beachte:** Diese formale Definition entspricht nicht der Implementation eines Tibbles in R. Dort ist `length(tb)` die Spaltenzahl.

## 2.2 Operationen

### 2.2.1 Mengenoperationen

Auch wenn Tibbles nicht Relationen direkt repräsentieren (siehe Bemerkung oben), können wir Operationen auf Relationen damit nachvollziehen.

Wir können Tibbles Relationen ähnlich machen, indem wir mehrfach auftretende Zeilen entfernen.

```
tb <- tibble(
  x = c(1, 1, 2, 2, 1),
  y = c("a", "b", "b", "b", "a"))
dplyr::distinct(tb) # dplyr ist auch Teil des Tidyverse
## # A tibble: 3 x 2
##       x     y
##   <dbl> <chr>
## 1     1     a
## 2     1     b
## 3     2     b
```

Für zwei Relationen des gleichen Relationenschemas  $R, S \subseteq W_1 \times \dots \times W_m$  können Standard-Mengenoperationen durchgeführt werden.

```
# Relationenschema (double, character)
# wobei double für die Menge der möglichen double-Werte in R steht
R <- tibble(
  x = c(1, 1),
  y = c("a", "b"))
S <- tibble(
  x = c(1, 2),
  y = c("a", "a"))
R
## # A tibble: 2 x 2
##       x     y
##   <dbl> <chr>
## 1     1     a
## 2     1     b
S
```

```
## # A tibble: 2 x 2
##       x     y
##   <dbl> <chr>
## 1     1     a
## 2     2     a
```

Im Folgenden steht  $\wedge$  für “und” und  $\vee$  für “oder”.

- Vereinigung  $R \cup S := \{t | t \in R \vee t \in S\}$

```
library(dplyr)
##
## Attaching package: 'dplyr'
## The following objects are masked from 'package:stats':
##
##     filter, lag
## The following objects are masked from 'package:base':
##
##     intersect, setdiff, setequal, union
union(R, S)
## # A tibble: 3 x 2
##       x     y
##   <dbl> <chr>
## 1     1     a
## 2     1     b
## 3     2     a
```

- Schnitt  $R \cap S := \{t | t \in R \wedge t \in S\}$

```
intersect(R, S)
## # A tibble: 1 x 2
##       x     y
##   <dbl> <chr>
## 1     1     a
```

- Differenz  $R \setminus S := \{t | t \in R \wedge t \notin S\}$

```
setdiff(R, S)
## # A tibble: 1 x 2
##       x     y
##   <dbl> <chr>
## 1     1     b
```

- Symmetrische Differenz  $R \Delta S := (R \cup S) \setminus (R \cap S)$

```
# Keine eigene Funktion vorhanden
setdiff(union(R, S), intersect(R, S))
## # A tibble: 2 x 2
##       x     y
##   <dbl> <chr>
## 1     1     b
## 2     2     a
```

## 2.2.2 Kartesisches Produkt

Seien  $R \subseteq W_1 \times \dots \times W_\ell$  und  $S \subseteq W_{\ell+1} \times \dots \times W_m$  Relationen.

- Kartesisches Produkt  $R \times S := \{(x_1, \dots, x_m) | (x_1, \dots, x_\ell) \in R \wedge (x_{\ell+1}, \dots, x_m) \in S\}$ .

```

# Schema (integer, character)
R <- tibble(
  x = 1:3,
  y = letters[1:3])
# Schema (character, logical)
S <- tibble(
  u = LETTERS[1:2],
  v = c(T, F))
R
## # A tibble: 3 x 2
##       x     y
##   <int> <chr>
## 1     1     a
## 2     2     b
## 3     3     c
S
## # A tibble: 2 x 2
##       u     v
##   <chr> <lgl>
## 1 A     TRUE
## 2 B     FALSE
tidyR::crossing(R, S) # tidyR ist auch Teil des Tidyverse
## # A tibble: 6 x 4
##       x     y     u     v
##   <int> <chr> <chr> <lgl>
## 1     1     a     A     TRUE
## 2     1     a     B     FALSE
## 3     2     b     A     TRUE
## 4     2     b     B     FALSE
## 5     3     c     A     TRUE
## 6     3     c     B     FALSE

```

# V07 – dplyr

3. Mai 2021

## Contents

<b>1</b>	<b>Verben für eine Tabelle</b>	<b>1</b>
1.1	filter()	2
1.2	arrange()	3
1.3	select()	5
1.4	mutate()	10
1.5	summarize()	11
1.6	Pipe	12
1.7	group by	13
1.8	Column-wise operations	18
1.9	Row-wise operations	19
1.10	Non-Standard Evaluation	21
<b>2</b>	<b>Verben für zwei Tabellen</b>	<b>23</b>
2.1	Mutating Joins	23
2.2	Filtering joins	27
<b>3</b>	<b>Relationale Datenbanken</b>	<b>28</b>
3.1	Erinnerung Grundbegriffe	28
3.2	Operationen	28

## 1 Verben für eine Tabelle

Das Paket `dplyr` gehört zum Tidyverse.

`dplyr` stellt Funktionen (genannt *Verben*) für die Manipulation von Daten in Tibbles bereit.

Zur Manipulation eines einzelnen Tibbles enthält `dplyr` folgende Verben:

- `filter()`: wähle bestimmte Zeilen aus
- `arrange()`: ordne Zeilen neu an
- `select()`: wähle bestimmte Spalten aus
- `mutate()`: erstelle neue Spalten
- `summarize()`: fasse Spalten zusammen, zB Mittelwert

Wir verwenden das Tibble `nycflights13::flights` in unseren Beispielen und setzen ein paar Optionen, um die Ausgabe kompakter zu gestalten.

```
library(tidyverse) # lädt auch dplyr
## -- Attaching packages --> tidyverse 1.3.1 --
## v ggplot2 3.3.3     v purrrr   0.3.4
## v tibble  3.1.0     v dplyr    1.0.5
## v tidyverse 1.1.3    v stringr  1.4.0
## v readr   1.4.0     v forcats 0.5.1
## -- Conflicts --> tidyverse_conflicts() --
## x dplyr::filter() masks stats::filter()
```

```

## x dplyr::lag()    masks stats::lag()
library(nycflights13)
options(
  tibble.print_min=4,
  tibble.print_max=4,
  tibble.max_extra_cols=0)
flights
## # A tibble: 336,776 x 19
##   year month   day dep_time sched_dep_time dep_delay arr_time sched_arr_time
##   <int> <int> <int>     <int>           <int>     <dbl>     <int>           <int>
## 1 2013     1     1      517            515       2     830            819
## 2 2013     1     1      533            529       4     850            830
## 3 2013     1     1      542            540       2     923            850
## 4 2013     1     1      544            545      -1    1004           1022
## # ... with 336,772 more rows

```

## 1.1 filter()

`filter(tb, fltr_1, ..., fltr_n)` gibt ein Tibble all derjenigen Zeilen von `tb` zurück, die **alle** Bedingungen (`fltr_1, ..., fltr_n`) erfüllen.

`fltr_1, ..., fltr_n` müssen zu einem `logical`-Vektor der Länge `nrow(tb)` (oder 1) evaluieren.

NA wirkt wie `FALSE`.

Spaltennamen von `tb` werden wie Variablen behandelt (Zugriff ohne `tb$`).

```

sel <- flights$dep_delay == 0
nrow(flights) == length(sel)
## [1] TRUE
typeof(sel)
## [1] "logical"
filter(flights, sel)
## # A tibble: 16,514 x 19
##   year month   day dep_time sched_dep_time dep_delay arr_time sched_arr_time
##   <int> <int> <int>     <int>           <int>     <dbl>     <int>           <int>
## 1 2013     1     1      559            559       0     702            706
## 2 2013     1     1      600            600       0     851            858
## 3 2013     1     1      600            600       0     837            825
## 4 2013     1     1      607            607       0     858            915
## # ... with 16,510 more rows

dep_delay # kein belegter Variablenname
## Error in eval(expr, envir, enclos): object 'dep_delay' not found
filter(flights, dep_delay < -30) # Spaltennamen fungieren als Variablen
## # A tibble: 3 x 19
##   year month   day dep_time sched_dep_time dep_delay arr_time sched_arr_time
##   <int> <int> <int>     <int>           <int>     <dbl>     <int>           <int>
## 1 2013     11    10     1408            1440      -32     1549            1559
## 2 2013     12     7      2040            2123      -43      40            2352
## 3 2013      2     3      2022            2055      -33     2240            2338

filter(flights, dep_time < 5, day == 1) # fltr1, fltr2 wie fltr1 & fltr2
## # A tibble: 4 x 19
##   year month   day dep_time sched_dep_time dep_delay arr_time sched_arr_time
##   <int> <int> <int>     <int>           <int>     <dbl>     <int>           <int>

```

```
##  <int> <int> <int> <int> <int> <dbl> <int> <int>
## 1 2013    3    1    4 2159    125    318    56
## 2 2013    6    1    2 2359     3    341    350
## 3 2013    7    1    1 2029    212    236 2359
## 4 2013    7    1    2 2359     3    344    344
```

Weitere Beispiel:

```
filter(flights, month %in% c(11, 12), origin == "JFK" | day == 27)
## # A tibble: 19,199 x 19
##   year month   day dep_time sched_dep_time dep_delay arr_time sched_arr_time
##   <int> <int> <int> <int> <int> <dbl> <int> <int>
## 1 2013    11    1     5 2359      6    352    345
## 2 2013    11    1    35 2250    105    123 2356
## 3 2013    11    1   542 545     -3    831    855
## 4 2013    11    1   549 600     -11   912    923
## # ... with 19,195 more rows
filter(flights, air_time / distance > 0.5)
## # A tibble: 45 x 19
##   year month   day dep_time sched_dep_time dep_delay arr_time sched_arr_time
##   <int> <int> <int> <int> <int> <dbl> <int> <int>
## 1 2013     1    28   1332 1300     32   1551   1406
## 2 2013     1    28   1917 1825     52   2118   1935
## 3 2013     1    30   1037 955      42   1221   1100
## 4 2013    10    17   1535 1540     -5   1724   1651
## # ... with 41 more rows
filter(flights, is.na(dep_time))
## # A tibble: 8,255 x 19
##   year month   day dep_time sched_dep_time dep_delay arr_time sched_arr_time
##   <int> <int> <int> <int> <int> <dbl> <int> <int>
## 1 2013     1    1     NA 1630      NA     NA    1815
## 2 2013     1    1     NA 1935      NA     NA    2240
## 3 2013     1    1     NA 1500      NA     NA    1825
## 4 2013     1    1     NA 600       NA     NA    901
## # ... with 8,251 more rows
```

**Bemerkung:** Siehe `?slice`, um Zeilen nach Zeilennummer auszuwählen.

## 1.2 arrange()

`arrange(tb, val_1, ..., val_n)` gibt das Tibble `tb` nach `val_1` aufsteigend sortiert aus.

Bei gleichen Werten in `val_i` wird durch `val_{i+1}` entschieden.

`val_1, ..., val_n` müssen zu atomaren Vektoren der Länge `nrow(tb)` evaluieren. Spaltennamen werden wie Variablen behandelt.

```
sort_by <- nrow(flights):1
arrange(flights, sort_by)
## # A tibble: 336,776 x 19
##   year month   day dep_time sched_dep_time dep_delay arr_time sched_arr_time
##   <int> <int> <int> <int> <int> <dbl> <int> <int>
## 1 2013     9    30     NA     840      NA     NA    1020
## 2 2013     9    30     NA    1159      NA     NA    1344
## 3 2013     9    30     NA    1210      NA     NA    1330
## 4 2013     9    30     NA    2200      NA     NA    2312
```

```

## # ... with 336,772 more rows
arrange(flights, dep_delay)
## # A tibble: 336,776 x 19
##   year month   day dep_time sched_dep_time dep_delay arr_time sched_arr_time
##   <int> <int> <int>     <int>           <int>     <dbl>     <int>           <int>
## 1 2013     12     7     2040         2123      -43      40         2352
## 2 2013      2     3     2022         2055      -33     2240         2338
## 3 2013     11    10     1408         1440      -32     1549         1559
## 4 2013      1    11     1900         1930      -30     2233         2243
## # ... with 336,772 more rows
arrange(flights, abs(dep_delay))
## # A tibble: 336,776 x 19
##   year month   day dep_time sched_dep_time dep_delay arr_time sched_arr_time
##   <int> <int> <int>     <int>           <int>     <dbl>     <int>           <int>
## 1 2013     1     1     559         559       0     702         706
## 2 2013     1     1     600         600       0     851         858
## 3 2013     1     1     600         600       0     837         825
## 4 2013     1     1     607         607       0     858         915
## # ... with 336,772 more rows
arrange(flights, year, month, day)
## # A tibble: 336,776 x 19
##   year month   day dep_time sched_dep_time dep_delay arr_time sched_arr_time
##   <int> <int> <int>     <int>           <int>     <dbl>     <int>           <int>
## 1 2013     1     1     517         515       2     830         819
## 2 2013     1     1     533         529       4     850         830
## 3 2013     1     1     542         540       2     923         850
## 4 2013     1     1     544         545      -1    1004        1022
## # ... with 336,772 more rows

```

`desc()` wandelt einen Vektor so um, dass die Sortierung (scheinbar) absteigend ist.

```

desc(1:10)
## [1] -1 -2 -3 -4 -5 -6 -7 -8 -9 -10
desc(letters)
## [1] -1 -2 -3 -4 -5 -6 -7 -8 -9 -10 -11 -12 -13 -14 -15 -16 -17 -18 -19
## [20] -20 -21 -22 -23 -24 -25 -26
arrange(flights, desc(dep_delay))
## # A tibble: 336,776 x 19
##   year month   day dep_time sched_dep_time dep_delay arr_time sched_arr_time
##   <int> <int> <int>     <int>           <int>     <dbl>     <int>           <int>
## 1 2013     1     9     641         900     1301     1242         1530
## 2 2013     6    15    1432        1935     1137     1607         2120
## 3 2013     1    10    1121        1635     1126     1239         1810
## 4 2013     9    20    1139        1845     1014     1457         2210
## # ... with 336,772 more rows
arrange(flights, year, desc(month), day)
## # A tibble: 336,776 x 19
##   year month   day dep_time sched_dep_time dep_delay arr_time sched_arr_time
##   <int> <int> <int>     <int>           <int>     <dbl>     <int>           <int>
## 1 2013     12     1     13     2359       14      446         445
## 2 2013     12     1     17     2359       18      443         437
## 3 2013     12     1     453      500      -7      636         651
## 4 2013     12     1     520      515       5      749         808
## # ... with 336,772 more rows

```

**Bemerkung:** Umsetzung in Base-R

```
arrange(flights, dep_delay) # in dplyr
flights[order(flights$dep_delay), ] # in Base-R
```

## 1.3 select()

`select(tb, vars_1, ..., vars_n)` gibt die in `vars_1, ..., vars_n` beschriebenen Spalten des Tibbles `tb` aus.

Zur Beschreibung der zu wählenden Spalten wird die **Tidy-Select**-Syntax genutzt, die in ihrer Auswertung nicht den Standard-Regeln von R genügt (**Non-Standard Evaluation**).

Die verschiedenen Möglichkeiten zur Spaltenwahl lassen sich in 2 Kategorien einteilen: *Position-Select* und *Condition-Select*.

### 1.3.1 Position-Select

Bei *Position-Select* können Spalten durch ihren Spaltenindex ausgewählt werden. Alternativ können die Spaltennamen angegeben werden (auch ohne Anführungszeichen, falls Spaltenname gültiger Variablenname ist).

```
select(flights, 3, 5) # Spaltenindex
## # A tibble: 336,776 x 2
##   day sched_dep_time
##   <int>          <int>
## 1 1             515
## 2 1             529
## 3 1             540
## 4 1             545
## # ... with 336,772 more rows
select(flights, flight, "origin", dest) # Spaltenname
## # A tibble: 336,776 x 3
##   flight origin dest
##   <int> <chr>   <chr>
## 1 1545  EWR     IAH
## 2 1714  LGA     IAH
## 3 1141  JFK     MIA
## 4 725   JFK     BQN
## # ... with 336,772 more rows
select(flights, flight, 9) # gemischt
## # A tibble: 336,776 x 2
##   flight arr_delay
##   <int>     <dbl>
## 1 1545      11
## 2 1714      20
## 3 1141      33
## 4 725       -18
## # ... with 336,772 more rows
```

Spaltennamen werden in Verbindung mit `:`, unäres `-`, und `c()` wie ihre Index-Nummern behandelt.

```
select(flights, year:day, arr_delay)
## # A tibble: 336,776 x 4
##   year month   day arr_delay
##   <int> <int> <int>     <dbl>
## 1 2013     1     1       11
```

```

## 2 2013 1 1 20
## 3 2013 1 1 33
## 4 2013 1 1 -18
## # ... with 336,772 more rows
select(flights, -(year:day))
## # A tibble: 336,776 x 16
##   dep_time sched_dep_time dep_delay arr_time sched_arr_time arr_delay carrier
##   <int>      <int>     <dbl>     <int>      <int>     <dbl> <chr>
## 1 517          515        2     830        819      11  UA
## 2 533          529        4     850        830      20  UA
## 3 542          540        2     923        850      33  AA
## 4 544          545       -1    1004       1022     -18  B6
## # ... with 336,772 more rows
select(flights, -c(2:dep_delay, sched_arr_time))
## # A tibble: 336,776 x 13
##   year arr_time arr_delay carrier flight tailnum origin dest air_time distance
##   <int>     <int>     <dbl> <chr>   <int> <chr>   <chr> <dbl>     <dbl>
## 1 2013      830       11  UA     1545 N14228  EWR   IAH     227    1400
## 2 2013      850       20  UA     1714 N24211  LGA   IAH     227    1416
## 3 2013      923       33  AA     1141 N619AA  JFK   MIA     160    1089
## 4 2013     1004      -18  B6     725  N804JB  JFK   BQN     183    1576
## # ... with 336,772 more rows

```

Stehen Spaltennamen oder -nummern in Variablen, sollten sie mit `all_of()` übergeben werden, um `-` nutzen zu können und Zweideutigkeiten zu vermeiden.

```

vars <- c("day", "dep_delay", "sched_arr_time")
select(flights, all_of(vars))
## # A tibble: 336,776 x 3
##   day dep_delay sched_arr_time
##   <int>     <dbl>      <int>
## 1 1          2        819
## 2 1          4        830
## 3 1          2        850
## 4 1         -1       1022
## # ... with 336,772 more rows
select(flights, -all_of(vars))
## # A tibble: 336,776 x 16
##   year month dep_time sched_dep_time arr_time arr_delay carrier flight tailnum
##   <int> <int>     <int>      <int>     <int>     <dbl> <chr>   <int> <chr>
## 1 2013    1      517        515     830        11  UA     1545 N14228
## 2 2013    1      533        529     850        20  UA     1714 N24211
## 3 2013    1      542        540     923        33  AA     1141 N619AA
## 4 2013    1      544        545    1004       -18  B6     725  N804JB
## # ... with 336,772 more rows
vars <- 1:3
select(flights, all_of(vars), dep_delay)
## # A tibble: 336,776 x 4
##   year month day dep_delay
##   <int> <int> <int>     <dbl>
## 1 2013    1    1        2
## 2 2013    1    1        4
## 3 2013    1    1        2
## 4 2013    1    1       -1

```

```

## # ... with 336,772 more rows
day <- 2
select(flights, day)
## # A tibble: 336,776 x 1
##   day
##   <int>
## 1 1
## 2 1
## 3 1
## 4 1
## # ... with 336,772 more rows
select(flights, all_of(day))
## # A tibble: 336,776 x 1
##   month
##   <int>
## 1 1
## 2 1
## 3 1
## 4 1
## # ... with 336,772 more rows

```

Mit `any_of()` statt `all_of()` entsteht kein Fehler, falls ein Wert des übergebenen Vektors nicht in der Tabelle als Spaltenname auftaucht.

```

vars <- c("day", "month", "Jahr")
# select(flights, all_of(vars)) # ERROR
select(flights, any_of(vars))
## # A tibble: 336,776 x 2
##   day month
##   <int> <int>
## 1 1     1
## 2 1     1
## 3 1     1
## 4 1     1
## # ... with 336,772 more rows

```

### 1.3.2 Condition-Select

Mit *Condition-Select* werden Spalten nach bestimmten Bedingungen ihrer Spaltennamen oder Werte ausgewählt.

**Prädikate** sind Funktionen, die angewendet auf eine ganze Spalte, ein einzelnes `TRUE` oder `FALSE` ergeben.

Spalten können mit Prädikaten ausgewählt werden (Bedingung an Werte der Spalte). Dabei muss das Prädikat innerhalb der Pseudo-Funktion `where()` stehen. `where()` ist keine Funktion mit Ein- und Ausgabe, sondern dient der Markierung ihres Arguments als Prädikat.

```

select(flights, where(is.character))
## # A tibble: 336,776 x 4
##   carrier tailnum origin dest
##   <chr>   <chr>   <chr>  <chr>
## 1 UA      N14228  EWR    IAH
## 2 UA      N24211  LGA    IAH
## 3 AA      N619AA  JFK    MIA
## 4 B6      N804JB  JFK    BQN
## # ... with 336,772 more rows

```

```

numeric_1000 <- function(x) is.numeric(x) && mean(x, na.rm=T) > 1000
select(flights, where(numeric_1000))
## # A tibble: 336,776 x 7
##   year dep_time sched_dep_time arr_time sched_arr_time flight distance
##   <int>     <int>          <int>     <int>          <int>     <int>     <dbl>
## 1 2013      517          515     830          819     1545     1400
## 2 2013      533          529     850          830     1714     1416
## 3 2013      542          540     923          850     1141     1089
## 4 2013      544          545    1004          1022     725     1576
## # ... with 336,772 more rows

```

Um eine Bedingung an den Namen einer Spalte zu stellen (zB enthält ein bestimmtes RegEx-Muster), können spezielle Funktionen, sogenannte **Select-Helpers** (siehe `?select_helpers`), genutzt werden.

```

select(flights, ends_with("_delay"))
## # A tibble: 336,776 x 2
##   dep_delay arr_delay
##   <dbl>     <dbl>
## 1 2          11
## 2 4          20
## 3 2          33
## 4 -1         -18
## # ... with 336,772 more rows
select(flights, starts_with("dep_"))
## # A tibble: 336,776 x 2
##   dep_time dep_delay
##   <int>     <dbl>
## 1 517      2
## 2 533      4
## 3 542      2
## 4 544     -1
## # ... with 336,772 more rows
select(flights, contains("dep"), day:year) # Condition- und Position-Select in einem Aufruf (aber versch)
## # A tibble: 336,776 x 6
##   dep_time sched_dep_time dep_delay day month year
##   <int>          <int>     <dbl> <int> <int> <int>
## 1 517            515      2     1     1  2013
## 2 533            529      4     1     1  2013
## 3 542            540      2     1     1  2013
## 4 544            545     -1     1     1  2013
## # ... with 336,772 more rows
select(flights, matches("[aeiou]{2}"))
## # A tibble: 336,776 x 6
##   year carrier tailnum air_time hour time_hour
##   <int> <chr>    <chr>    <dbl> <dbl> <dttm>
## 1 2013  UA      N14228     227     5 2013-01-01 05:00:00
## 2 2013  UA      N24211     227     5 2013-01-01 05:00:00
## 3 2013  AA      N619AA     160     5 2013-01-01 05:00:00
## 4 2013  B6      N804JB     183     5 2013-01-01 05:00:00
## # ... with 336,772 more rows

```

Zu den Select-Helpers zählt auch `everything()`, womit alle noch nicht selektierten Spalten gewählt werden. Dies kann zum Umordnen der Spalten dienen.

```
select(flights, time_hour, air_time, everything())
## # A tibble: 336,776 x 19
##   time_hour           air_time   year month   day dep_time sched_dep_time
##   <dttm>             <dbl> <int> <int> <int>    <int>        <int>
## 1 2013-01-01 05:00:00     227  2013     1     1      517        515
## 2 2013-01-01 05:00:00     227  2013     1     1      533        529
## 3 2013-01-01 05:00:00     160  2013     1     1      542        540
## 4 2013-01-01 05:00:00     183  2013     1     1      544        545
## # ... with 336,772 more rows
```

Verschiedene Select-Methoden können mit den logischen Operatoren `&`, `|`, `!` verbunden werden.

```
select(flights, !(contains("dep") | where(is.character)) & 1:arr_delay | hour)
## # A tibble: 336,776 x 7
##   year month   day arr_time sched_arr_time arr_delay hour
##   <int> <int> <int>    <int>        <int>    <dbl> <dbl>
## 1 2013     1     1      830          819        11     5
## 2 2013     1     1      850          830        20     5
## 3 2013     1     1      923          850        33     5
## 4 2013     1     1     1004         1022       -18     5
## # ... with 336,772 more rows
# Spalten, die weder 'dep' enthalten noch vom Typ character sind und
# aus vor einschließlich 'arr_delay' stehen oder die Spalte 'hour' sind.
```

### 1.3.3 Spalten umbenennen

Mit `select()` können Spalten umbenannt werden. Allerdings besteht die Ausgabe nur aus den genannten Spalten.

Mit `rename()` werden alle Spalten beibehalten.

```
select(flights, jahr = year, monat = month, tag = day)
## # A tibble: 336,776 x 3
##   jahr monat tag
##   <int> <int> <int>
## 1 2013     1     1
## 2 2013     1     1
## 3 2013     1     1
## 4 2013     1     1
## # ... with 336,772 more rows
rename(flights, jahr = year, monat = month, tag = day)
## # A tibble: 336,776 x 19
##   jahr monat tag dep_time sched_dep_time dep_delay arr_time sched_arr_time
##   <int> <int> <int>    <int>        <int>    <dbl> <int>        <int>
## 1 2013     1     1      517          515        2     830        819
## 2 2013     1     1      533          529        4     850        830
## 3 2013     1     1      542          540        2     923        850
## 4 2013     1     1      544          545       -1     1004       1022
## # ... with 336,772 more rows
```

`select()` ändert die Reihenfolge der Spalten, `rename()` nicht.

```
select(flights, monat = month, tag = day, jahr = year, everything())
## # A tibble: 336,776 x 19
##   monat tag jahr dep_time sched_dep_time dep_delay arr_time sched_arr_time
##   <int> <int> <int>    <int>        <int>    <dbl> <int>        <int>
```

```

## 1 1 1 2013 517 515 2 830 819
## 2 1 1 2013 533 529 4 850 830
## 3 1 1 2013 542 540 2 923 850
## 4 1 1 2013 544 545 -1 1004 1022
## # ... with 336,772 more rows
rename(flights, monat = month, tag = day, jahr = year)
## # A tibble: 336,776 x 19
##   jahr monat tag dep_time sched_dep_time dep_delay arr_time sched_arr_time
##   <int> <int> <int> <int> <int> <dbl> <int> <int>
## 1 2013 1 1 517 515 2 830 819
## 2 2013 1 1 533 529 4 850 830
## 3 2013 1 1 542 540 2 923 850
## 4 2013 1 1 544 545 -1 1004 1022
## # ... with 336,772 more rows

```

## 1.4 mutate()

`mutate(tb, name_1 = val_1, ..., name_n = val_n)` fügt dem Tibble `tb` die Spalten `name_1, ..., name_n` mit den Einträgen aus `val_1, ..., val_n` hinzu.

`val_1, ..., val_n` müssen zu Vektoren der Länge 1 oder `nrow(tb)` evaluieren. Spaltennamen können wie Variablen benutzt werden.

```

# create smaller tibble
flights_sml <- select(flights,
  year:day,
  ends_with("delay"),
  distance,
  air_time)

new_col <- nrow(flights_sml):1
mutate(flights_sml, num = new_col, one = 1)
## # A tibble: 336,776 x 9
##   year month day dep_delay arr_delay distance air_time num   one
##   <int> <int> <int> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
## 1 2013 1 1 2 11 1400 227 336776 1
## 2 2013 1 1 4 20 1416 227 336775 1
## 3 2013 1 1 2 33 1089 160 336774 1
## 4 2013 1 1 -1 -18 1576 183 336773 1
## # ... with 336,772 more rows

mutate(flights_sml,
  gain = dep_delay - arr_delay,
  speed = distance / air_time * 60)
## # A tibble: 336,776 x 9
##   year month day dep_delay arr_delay distance air_time gain speed
##   <int> <int> <int> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
## 1 2013 1 1 2 11 1400 227 -9 370.
## 2 2013 1 1 4 20 1416 227 -16 374.
## 3 2013 1 1 2 33 1089 160 -31 408.
## 4 2013 1 1 -1 -18 1576 183 17 517.
## # ... with 336,772 more rows

mutate(flights_sml, dep_delay_rank = rank(dep_delay))

```

```
## # A tibble: 336,776 x 8
##   year month   day dep_delay arr_delay distance air_time dep_delay_rank
##   <int> <int> <int>     <dbl>     <dbl>     <dbl>     <dbl>             <dbl>
## 1 2013     1     1       2       11     1400     227     211256
## 2 2013     1     1       4       20     1416     227     222226
## 3 2013     1     1       2       33     1089     160     211256
## 4 2013     1     1      -1      -18     1576     183     174169
## # ... with 336,772 more rows
```

Neue Spalten überschreiben alte mit dem selben Namen.

Siehe in `?mutate` die Argumente `.before`, `.after` zur Bestimmung der Einfügeposition neuer Spalten.

Mit `transmute()` an Stelle von `mutate()` werden nur die angegebenen Spalten zurückgegeben. Siehe auch in `?mutate` das Argument `.keep`.

```
transmute(flights_sml,
  dep_delay,
  gain = dep_delay - arr_delay,
  speed = distance / air_time * 60)
## # A tibble: 336,776 x 3
##   dep_delay gain speed
##   <dbl> <dbl> <dbl>
## 1 2       -9  370.
## 2 4      -16  374.
## 3 2      -31  408.
## 4 -1      17  517.
## # ... with 336,772 more rows
```

## 1.5 summarize()

`summarize(tb, name_1 = val_1, ..., name_n = val_n)` erzeugt aus dem Tibble `tb` ein neues Tibble mit Zeile(n) der Werte `val_1, ..., val_n` in den Spalten `name_1, ..., name_n`.

`val_1, ..., val_n` müssen kompatible Länge haben, dh die selbe Länge oder 1. Spaltennamen können wie Variablen benutzt werden. Länge 1 Elemente werden ggf recycelt.

```
x <- 1:2
summarize(flights,
  x_col = x, # Länge 2
  y_col = "Ypsilon", # Länge 1
  delay_mean = mean(dep_delay, na.rm = TRUE), # Länge 2
  delay_range = range(dep_delay, na.rm = TRUE)) # Länge 1
## # A tibble: 2 x 4
##   x_col y_col   delay_mean delay_range
##   <int> <chr>     <dbl>        <dbl>
## 1 1     Ypsilon     12.6        -43
## 2 2     Ypsilon     12.6       1301
```

Typische Summary-Funktionen sind zB:

- `mean()`, `sd()`, `var()`
- `median()`, `quantile()`, `min()`, `max()`, `range()`
- `n()` (Anzahl der Zeilen), `n_distinct()` (Anzahl verschiedener Elemente)

```
summarize(flights,
  count_flights = n(), # == nrow(flights)
```

```

count_origin = n_distinct(origin), # short for length(unique(origin))
mean_pos_dep_delay = mean(dep_delay[dep_delay > 0], na.rm = T),
missing_delays = sum(is.na(dep_delay) | is.na(arr_delay)))
## # A tibble: 1 x 4
##   count_flights count_origin mean_pos_dep_delay missing_delays
##       <int>          <int>             <dbl>          <int>
## 1       336776            3            39.4          9430

```

## 1.6 Pipe

Meist werden Verknüpfungen mehrerer Funktionen im Tidyverse mit dem Pipe-Operator `%>%` notiert (wird von `dplyr` automatisch aus dem Paket `magrittr` geladen).

`x %>% f()` ist äquivalent zu `f(x)` und `x %>% f(y)` entspricht `f(x, y)`.

Das linke Argument des Operators `%>%` wird zum ersten Argument des Funktionsaufrufs der rechten Seite.

```

x <- c(1:5, NA)
sqr <- function(x) x^2

# berechne Norm:
# 1. geschachtelt
sqrt(sum(sqr(x), na.rm=T))
## [1] 7.416198

# 2. Zwischenergebnisse
tmp <- sqr(x)
tmp <- sum(tmp, na.rm=T)
tmp <- sqrt(tmp)
tmp
## [1] 7.416198

# 3. Pipe
x %>%
  sqr() %>%
  sum(na.rm=T) %>%
  sqrt()
## [1] 7.416198

```

Um die Position des linken Wertes in der Argumentliste der rechten Funktion zu bestimmen, kann explizit der Variablenname `.` genutzt werden.

```

5 %>% # 5
`-`^(3, .) %>% # 3-5 = -2
`^`^(., 2) %>% # (-2)^2 = 4
`+`^(., 4) # 4 + 4 = 8
## [1] 8

```

Um die Lese- und Ausführungsreihenfolge auch bei Zuweisungen gleichzusetzen, wird der Zuweisungsoperator `->` genutzt.

```

x <- c(2, 3, 6, NA)
norm_x <-
  x %>%
  `^`^(1:3)%>%
  `^`^(2) %>%

```

```

  sum() %>%
  sqrt()
# oder:
x %>%
  ~[~(1:3)%>%
  ~~~(2) %>%
  sum() %>%
  sqrt() ->
  norm_x
norm_x
## [1] 7

```

Achtung: Geschachtelten Funktionsaufruf nicht mit %>% mischen!

```

x <- 1:4
x %>% mean(sqrt(.))
## Error in mean.default(., sqrt(.)): 'trim' must be numeric of length one
x %>% list(sqrt(.))
## [[1]]
## [1] 1 2 3 4
##
## [[2]]
## [1] 1.000000 1.414214 1.732051 2.000000
# . ist 1. Argument von list(), sqrt(.) ist 2. Argument

```

Nutze Ctrl+Shift+M (windows) Cmd+Shift+M (mac) um den Pipe-Operator schnell einzufügen.

Die Verben in dplyr sind auf Nutzung mit dem Pipe-Operator ausgelegt (Tibble immer an erster Stelle).

```

flights %>%
  select(ends_with("delay"), distance, air_time) %>%
  filter(!is.na(dep_delay), !is.na(arr_delay)) %>%
  mutate(
    gain = dep_delay - arr_delay,
    speed = distance / air_time * 60) %>%
  summarize(
    gain_mean = mean(gain),
    speed_mean = var(speed))
## # A tibble: 1 x 2
##   gain_mean speed_mean
##       <dbl>      <dbl>
## 1      5.66     3676.

```

## 1.7 group by

Mit `group_by()` lässt sich `summarize()` auf Gruppen von Zeilen anwenden.

```

flights %>%
  summarize(
    total_delay_mean = mean(dep_delay, na.rm = TRUE),
    total_delay_var = var(dep_delay, na.rm = TRUE))
## # A tibble: 1 x 2
##   total_delay_mean total_delay_var
##       <dbl>          <dbl>
## 1          12.6          1617.

```

```

flights %>%
  group_by(year, month, day) %>%
  summarise(
    daily_delay_mean = mean(dep_delay, na.rm = TRUE),
    daily_delay_var = var(dep_delay, na.rm = TRUE)) ->
  daily
## `summarise()` has grouped output by 'year', 'month'. You can override using the `groups` argument.
daily
## # A tibble: 365 x 5
## # Groups:   year, month [12]
##   year month day daily_delay_mean daily_delay_var
##   <int> <int> <int>          <dbl>          <dbl>
## 1 2013     1     1          11.5          2049.
## 2 2013     1     2          13.9          1384.
## 3 2013     1     3          11.0          990.
## 4 2013     1     4          8.95          769.
## # ... with 361 more rows

```

group\_by(tb, var\_1, ..., var\_n) gibt das Tibble tb als **gruppiertes Tibble** mit einem zusätzlichen Attribut **groups** zurück.

Für jede in tb vorhandene Kombination aus Werten der Spalten var\_1, ..., var\_n wird eine Gruppe angelegt. Das Attribut **groups** enthält für jede Gruppe den Index-Vektor aller Zeilen mit den entsprechenden Werten bei var\_1, ..., var\_n.

```

class(daily)
## [1] "grouped_df" "tbl_df"      "tbl"        "data.frame"

daily %>%
  group_by(month) %>% # nochmal gruppieren
  attr("groups") ->
  groups
groups # Wert des Attributs "groups"
## # A tibble: 12 x 2
##   month      .rows
##   <int> <list<int>>
## 1     1      [31]
## 2     2      [28]
## 3     3      [31]
## 4     4      [30]
## # ... with 8 more rows
groups[[1,2]] # Indizes der Gruppenelemente Januar
## <list_of<integer>[1]>
## [[1]]
## [1] 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25
## [26] 26 27 28 29 30 31
groups[[2,2]] # Indizes der Gruppenelemente Februar
## <list_of<integer>[1]>
## [[1]]
## [1] 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56
## [26] 57 58 59

```

Die Gruppierung wird mit **ungroup()** aufgehoben (Attribut entfernt).

```
by_day <- group_by(flights, year, month, day)
attr(ungroup(by_day), "groups")
## NULL
```

group\_cols() ist ein Selection-Helper um Gruppierungsvariablen mit select() auszuwählen.

```
by_day %>% select(group_cols())
## # A tibble: 336,776 x 3
## # Groups:   year, month, day [365]
##   year month   day
##   <int> <int> <int>
## 1 2013     1     1
## 2 2013     1     1
## 3 2013     1     1
## 4 2013     1     1
## # ... with 336,772 more rows
```

### 1.7.1 grouped summarize

Wird summarize(gtb, name\_1 = val\_1, ..., name\_n = val\_n) auf ein gruppiertes Tibble gtb angewendet, werden val\_1, ..., val\_n für jede Gruppe einzeln ausgewertet und angewendet. Das Rückgabe-Tibble enthält für jede Gruppe eine Zeile.

Siehe oben.

```
flights %>%
  group_by(year, month, day, origin) %>%
  summarize(n = n()) # count flights per day
## `summarise()` has grouped output by 'year', 'month', 'day'. You can override using the `groups` arg
## # A tibble: 1,095 x 5
## # Groups:   year, month, day [365]
##   year month   day origin     n
##   <int> <int> <int> <chr> <int>
## 1 2013     1     1 EWR     305
## 2 2013     1     1 JFK     297
## 3 2013     1     1 LGA     240
## 4 2013     1     2 EWR     350
## # ... with 1,091 more rows
# short notation:
flights %>%
  count(year, month, day, origin)
## # A tibble: 1,095 x 5
##   year month   day origin     n
##   <int> <int> <int> <chr> <int>
## 1 2013     1     1 EWR     305
## 2 2013     1     1 JFK     297
## 3 2013     1     1 LGA     240
## 4 2013     1     2 EWR     350
## # ... with 1,091 more rows
flights %>%
  count(year, month, day, origin, sort=TRUE) # count and sort descending
## # A tibble: 1,095 x 5
##   year month   day origin     n
##   <int> <int> <int> <chr> <int>
## 1 2013     4    15 EWR     377
```

```

## 2 2013 4 11 EWR 376
## 3 2013 4 18 EWR 376
## 4 2013 4 10 EWR 375
## # ... with 1,091 more rows

```

### 1.7.2 grouped filter

Wird `filter(gtb, fltr_1, ..., fltr_n)` auf ein gruppiertes Tibble `gtb` angewendet, werden `fltr_1, ..., fltr_n` für jede Gruppe einzeln ausgewertet und angewendet.

```

# flights with departure delay larger than average of all flights
flights %>%
  select(origin, dep_delay, everything()) %>% # reorder
  filter(dep_delay > mean(dep_delay, na.rm=TRUE))
## # A tibble: 77,584 x 19
##   origin dep_delay year month   day dep_time sched_dep_time arr_time
##   <chr>     <dbl> <int> <int> <int>     <int>          <int>     <int>
## 1 LGA         13  2013     1     1     623          610        920
## 2 EWR         24  2013     1     1     632          608        740
## 3 EWR         47  2013     1     1     732          645       1011
## 4 JFK         13  2013     1     1     743          730       1107
## # ... with 77,580 more rows

# flights with departure delay larger than average of flights from same origin
flights %>%
  select(origin, dep_delay, everything()) %>% # reorder
  group_by(origin) %>%
  filter(dep_delay > mean(dep_delay, na.rm=TRUE))
## # A tibble: 76,198 x 19
## # Groups:   origin [3]
##   origin dep_delay year month   day dep_time sched_dep_time arr_time
##   <chr>     <dbl> <int> <int> <int>     <int>          <int>     <int>
## 1 LGA         13  2013     1     1     623          610        920
## 2 EWR         24  2013     1     1     632          608        740
## 3 EWR         47  2013     1     1     732          645       1011
## 4 JFK         13  2013     1     1     743          730       1107
## # ... with 76,194 more rows

# top 3 most arrival delay in dataset
flights %>%
  select(year:day, arr_delay, everything()) %>% # reorder
  filter(rank(desc(arr_delay)) <= 3)
## # A tibble: 3 x 19
##   year month   day arr_delay dep_time sched_dep_time dep_delay arr_time
##   <int> <int> <int>     <dbl>     <int>          <int>     <dbl>     <int>
## 1 2013     1     9      1272      641          900     1301     1242
## 2 2013     1    10      1109     1121         1635     1126     1239
## 3 2013     6    15      1127     1432         1935     1137     1607

# top 3 most arrival delay per day
flights %>%
  select(year:day, arr_delay, everything()) %>% # reorder
  group_by(year, month, day) %>%
  filter(rank(desc(arr_delay)) <= 3)

```

```

## # A tibble: 1,085 x 19
## # Groups: year, month, day [365]
##   year month day arr_delay dep_time sched_dep_time dep_delay arr_time
##   <int> <int> <int>     <dbl>     <int>          <int>     <dbl>     <int>
## 1 2013     1     1      851      848        1835      853    1001
## 2 2013     1     1      338     1815        1325      290    2120
## 3 2013     1     1      456     2343        1724      379    314
## 4 2013     1     2      323     1412        838       334    1710
## # ... with 1,081 more rows

# if more than 5e7 miles in dataset, then all flights in dataset, else none
# (not meaningful...)
flights %>%
  select(carrier, distance, everything()) %>% # reorder
  filter(sum(distance) > 5e7)
## # A tibble: 336,776 x 19
##   carrier distance year month day dep_time sched_dep_time dep_delay arr_time
##   <chr>     <dbl> <int> <int> <int>     <int>          <int>     <dbl>     <int>
## 1 UA        1400  2013     1     1      517        515        2     830
## 2 UA        1416  2013     1     1      533        529        4     850
## 3 AA        1089  2013     1     1      542        540        2     923
## 4 B6        1576  2013     1     1      544        545       -1    1004
## # ... with 336,772 more rows

# all flights from carriers which fly more than 5e7 miles total
flights %>%
  select(carrier, distance, everything()) %>% # reorder
  group_by(carrier) %>%
  filter(sum(distance) > 5e7)
## # A tibble: 161,410 x 19
## # Groups: carrier [3]
##   carrier distance year month day dep_time sched_dep_time dep_delay arr_time
##   <chr>     <dbl> <int> <int> <int>     <int>          <int>     <dbl>     <int>
## 1 UA        1400  2013     1     1      517        515        2     830
## 2 UA        1416  2013     1     1      533        529        4     850
## 3 B6        1576  2013     1     1      544        545       -1    1004
## 4 DL        762   2013     1     1      554        600       -6    812
## # ... with 161,406 more rows

```

### 1.7.3 grouped mutate

Wird `mutate(gtb, name_1 = val_1, ..., name_n = val_n)` auf ein gruppiertes Tibble `gtb` angewendet, werden `val_1, ..., val_n` für jede Gruppe einzeln ausgewertet.

```

# flight's proportion of total distance
flights %>%
  mutate(prop_distance = distance / sum(distance)) %>%
  select(carrier, distance, prop_distance)
## # A tibble: 336,776 x 3
##   carrier distance prop_distance
##   <chr>     <dbl>        <dbl>
## 1 UA        1400    0.00000400
## 2 UA        1416    0.00000404
## 3 AA        1089    0.00000311

```

```

## 4 B6      1576  0.00000450
## # ... with 336,772 more rows

# flight's proportion of total distance for each carrier
flights %>%
  group_by(carrier) %>%
  mutate(prop_distance = distance / sum(distance)) %>%
  select(carrier, distance, prop_distance)
## # A tibble: 336,776 x 3
## # Groups:   carrier [16]
##   carrier distance prop_distance
##   <chr>     <dbl>      <dbl>
## 1 UA        1400      0.0000156
## 2 UA        1416      0.0000158
## 3 AA        1089      0.0000248
## 4 B6        1576      0.0000270
## # ... with 336,772 more rows

```

## 1.8 Column-wise operations

Mit `across()` wenden wir Operationen, die durch Verben wie `summarize()`, `mutate()`, `filter()`, ..., bereitgestellt werden, auf eine Menge von Spalten an.

Dabei wird diese Menge an Spalten mit der Tidy-Select-Syntax von `select()` ausgewählt.

```

tb <- tibble(let=letters[1:5], unif=runif(5), norm=rnorm(5), exp=rexp(5))
tb %>%
  summarize(across(where(is.numeric), mean))
## # A tibble: 1 x 3
##   unif   norm   exp
##   <dbl>  <dbl>  <dbl>
## 1 0.336 -0.259  1.36

rescale01 <- function(x) {
  rng <- range(x, na.rm = TRUE)
  (x - rng[1]) / (rng[2] - rng[1])
}
tb %>%
  mutate(across(unif:exp, rescale01))
## # A tibble: 5 x 4
##   let    unif   norm   exp
##   <chr> <dbl>  <dbl>  <dbl>
## 1 a     0.0887 0.506  0.923
## 2 b     1       0.226  0.583
## 3 c     0.718   1       1
## 4 d     0.181   0       0.552
## # ... with 1 more row

tb %>%
  filter(across(-let, function(x) x>0))
## # A tibble: 1 x 4
##   let    unif   norm   exp
##   <chr> <dbl>  <dbl>  <dbl>
## 1 c     0.601  0.747  2.10

```

Das erste Argument von `across(cols, fun)` beschreibt in der Tidy-Select-Syntax die Gruppe der Spalten. Das zweite Argument ist eine einzelne Funktion oder eine Liste von Funktionen.

Für `summarize()` müssen die Funktionen auf allen ausgewählten Spalten einen Vektor der selben Länge oder 1 ergeben; `mutate()` Länge `nrow(tb)` oder 1.

Für `filter()` müssen die Funktionen angewendet auf Spalten `logical`-Vektoren der Länge `nrow(tb)` oder 1 ergeben.

Der angewendeten Funktion können weitere konstante Argumente übergeben werden.

```
tb[1,2] <- NA
tb %>%
  summarize(across(where(is.numeric), mean, na.rm=TRUE))
## # A tibble: 1 x 3
##   unif   norm   exp
##   <dbl>  <dbl>  <dbl>
## 1 0.400 -0.259  1.36
```

Mit dem Argument `.names` werden für `mutate()` und `summarize()` die Namen der neuen Spalten bestimmt. Dabei wird `glue`-Syntax verwendet, wobei Variablen `fn` und `col` für Listennamen der Funktionen bzw ursprünglicher Spaltenname stehen.

```
tb %>%
  summarise(across(where(is.numeric), list(mini=min, maxi=max), na.rm=T, .names = "{fn}. {col}"))
## # A tibble: 1 x 6
##   mini.unif maxi.unif mini.norm maxi.norm mini.exp maxi.exp
##   <dbl>     <dbl>     <dbl>     <dbl>     <dbl>     <dbl>
## 1 0.00740   0.834    -0.935    0.747    0.201    2.10
```

## 1.9 Row-wise operations

Die meisten Funktionen in R sind auf irgendeine Weise vektorisiert. Möchten wir in jeder Zeile eines Tibbles die Summe von der Einträge von zwei Spalten berechnen, machen wir dies nicht mit einer Schleife, sondern addieren die beiden Spalten als Vektoren mit dem vektorisierten `+` Operator.

Solche Spalten-Operationen sind immer Zeilen-Operationen vorzuziehen.

Dennoch kann es in seltenen Fällen passieren, dass zeilenweises Ausführen von Funktionen auf einer Tabelle nötig ist.

Dann kann `rowwise(tb)` eingesetzt werden. Im Wesentlichen entspricht dies `group_by(tb, 1:nrow(tb))`. Dh jede Zeile ist in einer eigenen Gruppe.

```
tb <- tibble(x = 1:2, y = 3:4, z = 5:6)
tb %>% rowwise() -> rtb
rtb

## # A tibble: 2 x 3
## # Rowwise:
##   x     y     z
##   <int> <int> <int>
## 1 1     3     5
## 2 2     4     6
tb %>% mutate(m = mean(c(x, y, z)))
## # A tibble: 2 x 4
## # Rowwise:
##   x     y     z     m
##   <int> <int> <int> <dbl>
## 1 1     3     5     3.5
```

```
## 2      2      4      6    3.5
rtb %>% mutate(m = mean(c(x, y, z)))
## # A tibble: 2 x 4
## # Rowwise:
##   x     y     z     m
## <int> <int> <int> <dbl>
## 1     1     3     5     3
## 2     2     4     6     4
```

Wird `rowwise(tb, id_1, ..., id_n)` noch eine oder mehrere `id`-Spalten übergeben, zählen diese als Gruppierungsvariablen. Damit bleiben sie beim Aufruf mit `summarize()` erhalten.

```
tb <- tibble(id = letters[1:6], w = 10:15, x = 20:25, y = 30:35, z = 40:45)
tb
## # A tibble: 6 x 5
##   id      w     x     y     z
##   <chr> <int> <int> <int> <int>
## 1 a      10    20    30    40
## 2 b      11    21    31    41
## 3 c      12    22    32    42
## 4 d      13    23    33    43
## # ... with 2 more rows
rtb <- tb %>% rowwise(id)
rtb %>% mutate(total = sum(c(w, x, y, z)))
## # A tibble: 6 x 6
## # Rowwise:  id
##   id      w     x     y     z   total
##   <chr> <int> <int> <int> <int> <int>
## 1 a      10    20    30    40    100
## 2 b      11    21    31    41    104
## 3 c      12    22    32    42    108
## 4 d      13    23    33    43    112
## # ... with 2 more rows
rtb %>% summarize(total = sum(c(w, x, y, z)))
## `summarise()` has grouped output by 'id'. You can override using the `groups` argument.
## # A tibble: 6 x 2
## # Groups:   id [6]
##   id   total
##   <chr> <int>
## 1 a     100
## 2 b     104
## 3 c     108
## 4 d     112
## # ... with 2 more rows
```

Um Spalten mit Tidy-Select-Syntax auszuwählen, nutze `c_across()`.

```
rtb %>% mutate(total = sum(c_across(w:z)))
## # A tibble: 6 x 6
## # Rowwise:  id
##   id      w     x     y     z   total
##   <chr> <int> <int> <int> <int> <int>
## 1 a      10    20    30    40    100
## 2 b      11    21    31    41    104
## 3 c      12    22    32    42    108
```

```

## 4 d      13   23   33   43   112
## # ... with 2 more rows
rtb %>% summarise(total = sum(c_across(where(is.numeric()))))
## `summarise()` has grouped output by 'id'. You can override using the `groups` argument.
## # A tibble: 6 x 2
## # Groups:   id [6]
##   id     total
##   <chr> <int>
## 1 a      100
## 2 b      104
## 3 c      108
## 4 d      112
## # ... with 2 more rows

```

Wie bei `group_by()` wird die zeilenweise Gruppierung mit `ungroup()` aufgehoben.

```

# compute the proportion of total for each column:
rtb %>%
  mutate(total = sum(c_across(w:z))) %>%
  ungroup() %>%
  mutate(across(w:z, function(x) x/total))
## # A tibble: 6 x 6
##   id     w     x     y     z total
##   <chr> <dbl> <dbl> <dbl> <dbl> <int>
## 1 a     0.1   0.2   0.3   0.4   100
## 2 b     0.106 0.202 0.298 0.394  104
## 3 c     0.111 0.204 0.296 0.389  108
## 4 d     0.116 0.205 0.295 0.384  112
## # ... with 2 more rows

```

## 1.10 Non-Standard Evaluation

Viele `dplyr`-Funktionen nutzen **Non-Standard Evaluation** ihrer Argumente. D.h. sie folgen nicht den üblichen Regeln, wie `call-by-value`. Stattdessen wird der gesamte als Argument eingegebene Ausdruck von den Funktionen verwertet.

So ist es möglich bestimmte Anweisungen zu vereinfachen.

```

tb <- tibble(x=1:3, y=3:1)
filter(tb, x < 3, y > 1) # x, y nicht als Variablen vorhanden
## # A tibble: 2 x 2
##   x     y
##   <int> <int>
## 1 1     3
## 2 2     2
tb[tb$x < 3 & tb$y > 1, ] # ohne Non-Standard Evaluation
## # A tibble: 2 x 2
##   x     y
##   <int> <int>
## 1 1     3
## 2 2     2

```

Dies vereinfacht bestimmte Funktionsaufrufe, bringt aber auch Nachteile mit sich.

Abhängig davon, welche Spaltennamen in `tb` vorhanden sind, hat `filter(tb, x==y)` verschiedene Bedeutungen. Zuerst wird in `tb` nach entsprechenden Spaltennamen gesucht. Nur falls ein Name dort nicht gefunden

wird, wird außerhalb gesucht.

```
x <- 1
y <- 2
tb <- tibble(x=1:3, y=3:1)
filter(tb, x==y) # ist gleich:
## # A tibble: 1 x 2
##      x     y
## 1     2     2
tb[tb$x == tb$y, ]
## # A tibble: 1 x 2
##      x     y
## 1     2     2
## 1     2     2
```

```
x <- 1
y <- 2
tb <- tibble(u=1:3, y=3:1)
filter(tb, x==y) # ist gleich:
## # A tibble: 1 x 2
##      u     y
## 1     3     1
tb[x == tb$y, ]
## # A tibble: 1 x 2
##      u     y
## 1     3     1
## 1     3     1
```

```
x <- 1
y <- 2
tb <- tibble(x=1:3, v=3:1)
filter(tb, x==y) # ist gleich:
## # A tibble: 1 x 2
##      x     v
## 1     2     2
tb[tb$x == y, ]
## # A tibble: 1 x 2
##      x     v
## 1     2     2
## 1     2     2
```

```
x <- 1
y <- 2
tb <- tibble(u=1:3, v=3:1)
filter(tb, x==y) # ist gleich:
## # A tibble: 0 x 2
tb[x == y, ]
## # A tibble: 0 x 2
```

Außerdem sind solche Argumente nicht *referentially transparent*, dh Werte können nicht immer direkt durch eine äquivalente Variable ersetzt werden.

```
tb <- tibble(x=1:3, y=3:1)
filter(tb, x == 1)
```

```
## # A tibble: 1 x 2
##       x     y
##   <int> <int>
## 1     1     3
my_var <- "x"
filter(tb, my_var == 1)
## # A tibble: 0 x 2
```

Eine Lösung für letzteres Problem ist, explizit das Daten-Tibble anzugeben. Dies kann auch mit `.data` geschehen, was immer das erste Argument referenziert (nützlich für `%>%`).

```
filter(tb, tb[[my_var]] == 1)
## # A tibble: 1 x 2
##       x     y
##   <int> <int>
## 1     1     3
filter(tb, .data[[my_var]] == 1)
## # A tibble: 1 x 2
##       x     y
##   <int> <int>
## 1     1     3
srt <- "dep_delay"
fltr <- "dep_time"
flights %>%
  arrange(.data[[srt]]) %>%
  filter(.data[[fltr]] < 500)
## # A tibble: 1,484 x 19
##   year month   day dep_time sched_dep_time dep_delay arr_time sched_arr_time
##   <int> <int> <int>   <int>           <int>     <dbl>   <int>           <int>
## 1  2013     5     8     445           500      -15     620           640
## 2  2013     5     5     446           500      -14     636           640
## 3  2013     9     4     446           500      -14     618           648
## 4  2013    10     1     447           500      -13     614           648
## # ... with 1,480 more rows
```

Mehr zum Umgang mit non-standard Evaluation unter <https://cran.r-project.org/web/packages/dplyr/vignettes/programming.html> und in einem späteren Kapitel des Kurses (*Meta-Programmierung*).

## 2 Verben für zwei Tabellen

Wie im Kapitel Tibbles beschrieben, fügen `bind_rows()` und `bind_cols()` Tibbles mit kompatibler Struktur zusammen, unabhängig von konkreten Werten.

In diesem Kapitel zeigen wir, wie wir Tabellen abhängig vom Vorkommen von Werten einer Tabelle in der anderen zusammenfügen.

### 2.1 Mutating Joins

Das Dataset `nycflights13` enthält neben `flights` noch die Tabellen `airlines`, `airports`, `planes`, `weather`.

Die Tabelle `flights` enthält eine Variable `carrier` – die 2-Buchstaben-Abkürzung der Airline. In der Tabelle `airlines` ist der Langname der Airline angegeben.

```
library(tidyverse)
library(nycflights13)
```

```

flights2 <- flights %>%
  select(year:day, hour, tailnum, carrier)

flights2
## # A tibble: 336,776 x 6
##   year month   day hour tailnum carrier
##   <int> <int> <int> <dbl> <chr>   <chr>
## 1 2013     1     1     5 N14228  UA
## 2 2013     1     1     5 N24211  UA
## 3 2013     1     1     5 N619AA  AA
## 4 2013     1     1     5 N804JB  B6
## # ... with 336,772 more rows
airlines
## # A tibble: 16 x 2
##   carrier name
##   <chr>   <chr>
## 1 9E      Endeavor Air Inc.
## 2 AA      American Airlines Inc.
## 3 AS      Alaska Airlines Inc.
## 4 B6      JetBlue Airways
## # ... with 12 more rows

```

Soll nun zu einem Flug der Langname der Airline angezeigt werden, müssen beide Tabellen genutzt werden.

`left_join(tb1, tb2, by)` erzeugt ein Tibble mit den Spalten aus `tb1` und `tb2`. `by` benennt ein Paar aus Spalten, je eine aus `tb1` und `tb2`. Sie werden als **Key** bezeichnet. Zeilen der Ausgabe setzen sich aus Zeilen aus `tb1` und `tb2` zusammen, deren Key-Werte übereinstimmen.

```

flights2 %>%
  left_join(airlines, by = "carrier")
## # A tibble: 336,776 x 7
##   year month   day hour tailnum carrier name
##   <int> <int> <int> <dbl> <chr>   <chr>   <chr>
## 1 2013     1     1     5 N14228  UA      United Air Lines Inc.
## 2 2013     1     1     5 N24211  UA      United Air Lines Inc.
## 3 2013     1     1     5 N619AA  AA      American Airlines Inc.
## 4 2013     1     1     5 N804JB  B6      JetBlue Airways
## # ... with 336,772 more rows

```

Das Argument `by` gibt an, welche Variablen zum Vereinigen der Tabellen genutzt werden.

- `by = NULL` (default): nutze die Variable, die in beiden Tabellen gleich heißt
- `by = "name"`: nutze Variable `name`, die in beiden Tabellen vorkommt.
- `by = c("name1" = "name2")`: verbinde Variable `name1` aus `tb1` mit `name2` aus `tb2`.

```

flights2 <- flights %>%
  select(month:day, hour, origin, dest, tailnum)
airports2 <- airports %>% select(1:2)

flights2
## # A tibble: 336,776 x 6
##   month   day hour origin dest tailnum
##   <int> <int> <dbl> <chr>  <chr>  <chr>
## 1     1     1     5 EWR    IAH    N14228
## 2     1     1     5 LGA    IAH    N24211
## 3     1     1     5 JFK    MIA    N619AA

```

```

## 4      1      1      5 JFK      BQN      N804JB
## # ... with 336,772 more rows
airports2
## # A tibble: 1,458 x 2
##   faa      name
##   <chr> <chr>
## 1 04G  Lansdowne Airport
## 2 06A  Moton Field Municipal Airport
## 3 06C  Schaumburg Regional
## 4 06N  Randall Airport
## # ... with 1,454 more rows

flights2 %>%
  left_join(airports2, c("dest" = "faa"))
## # A tibble: 336,776 x 7
##   month   day   hour origin dest  tailnum name
##   <int> <int> <dbl> <chr> <chr> <chr> <chr>
## 1     1     1     5 EWR   IAH   N14228 George Bush Intercontinental
## 2     1     1     5 LGA   IAH   N24211 George Bush Intercontinental
## 3     1     1     5 JFK   MIA   N619AA Miami Intl
## 4     1     1     5 JFK   BQN   N804JB <NA>
## # ... with 336,772 more rows
flights2 %>%
  left_join(airports2, c("origin" = "faa"))
## # A tibble: 336,776 x 7
##   month   day   hour origin dest  tailnum name
##   <int> <int> <dbl> <chr> <chr> <chr> <chr>
## 1     1     1     5 EWR   IAH   N14228 Newark Liberty Intl
## 2     1     1     5 LGA   IAH   N24211 La Guardia
## 3     1     1     5 JFK   MIA   N619AA John F Kennedy Intl
## 4     1     1     5 JFK   BQN   N804JB John F Kennedy Intl
## # ... with 336,772 more rows
flights2 %>%
  left_join(airports2, c("origin" = "faa")) %>%
  left_join(airports2, c("dest" = "faa"))
## # A tibble: 336,776 x 8
##   month   day   hour origin dest  tailnum name.x      name.y
##   <int> <int> <dbl> <chr> <chr> <chr> <chr> <chr>
## 1     1     1     5 EWR   IAH   N14228 Newark Liberty ~ George Bush Intercont-
## 2     1     1     5 LGA   IAH   N24211 La Guardia     George Bush Intercont-
## 3     1     1     5 JFK   MIA   N619AA John F Kennedy ~ Miami Intl
## 4     1     1     5 JFK   BQN   N804JB John F Kennedy ~ <NA>
## # ... with 336,772 more rows
flights2 %>%
  left_join(airports2, c("origin" = "faa")) %>%
  left_join(
    airports2,
    c("dest" = "faa"),
    suffix=c("_origin", "_dest"))
## # A tibble: 336,776 x 8
##   month   day   hour origin dest  tailnum name_origin      name_dest
##   <int> <int> <dbl> <chr> <chr> <chr> <chr> <chr>
## 1     1     1     5 EWR   IAH   N14228 Newark Liberty I~ George Bush Intercon-

```

```

## 2      1      1      5 LGA      IAH      N24211  La Guardia      George Bush Intercon-
## 3      1      1      5 JFK      MIA      N619AA  John F Kennedy I~ Miami Intl
## 4      1      1      5 JFK      BQN      N804JB  John F Kennedy I~ <NA>
## # ... with 336,772 more rows

```

Existiert ein Key-Wert mehrere Male in einer der beiden Spalten, werden alle passenden Kombinationen an Zeilen hinzugefügt.

```

x <- tibble(
  key = c(1, 2, 2, 3, 3),
  val_x = c("x1", "x2a", "x2b", "x3a", "x3b"))
y <- tibble(
  key = c(1, 1, 2, 3, 3),
  val_y = c("y1a", "y1b", "y2", "y3a", "y3b"))
left_join(x, y)
## Joining, by = "key"
## # A tibble: 8 x 3
##       key val_x val_y
##   <dbl> <chr> <chr>
## 1     1 x1    y1a
## 2     1 x1    y1b
## 3     2 x2a   y2
## 4     2 x2b   y2
## # ... with 4 more rows

```

Werden mehrere Keys angegeben, müssen die Werte aller Keys übereinstimmen, um eine vollständige Ausgabezeile zu erhalten.

```

x <- tibble(
  key1 = c(1, 1, 2, 2, 3),
  key2 = c(1, 2, 1, 2, 3),
  val = c("x1", "x2", "x3", "x4", "x5"))
y <- tibble(
  key1 = c(1, 1, 2, 2, 3),
  key2 = c(1, 2, 1, 2, 4),
  val = c("y1", "y2", "y3", "y4", "y5"))
left_join(x, y, by = c("key1", "key2"))
## # A tibble: 5 x 4
##       key1  key2 val.x val.y
##   <dbl> <dbl> <chr> <chr>
## 1     1     1 x1    y1
## 2     1     2 x2    y2
## 3     2     1 x3    y3
## 4     2     2 x4    y4
## # ... with 1 more row

```

`left_join()`, `right_join()`, `full_join()`, `inner_join()` unterscheiden sich darin, was passiert, wenn ein Key-Wert nur in einer von beiden Tabellen vorkommt.

```

x <- tibble(
  key = 1:3,
  val_x = c("x1", "x2", "x3"))
y <- tibble(
  key = c(1, 2, 4),
  val_y = c("y1", "y2", "y3"))
x %>%

```

```

  left_join(y, by = "key")
## # A tibble: 3 x 3
##   key  val_x val_y
##   <dbl> <chr> <chr>
## 1     1 x1    y1
## 2     2 x2    y2
## 3     3 x3    <NA>
x %>%
  right_join(y, by = "key")
## # A tibble: 3 x 3
##   key  val_x val_y
##   <dbl> <chr> <chr>
## 1     1 x1    y1
## 2     2 x2    y2
## 3     4 <NA>  y3
x %>%
  full_join(y, by = "key")
## # A tibble: 4 x 3
##   key  val_x val_y
##   <dbl> <chr> <chr>
## 1     1 x1    y1
## 2     2 x2    y2
## 3     3 x3    <NA>
## 4     4 <NA>  y3
x %>%
  inner_join(y, by = "key")
## # A tibble: 2 x 3
##   key  val_x val_y
##   <dbl> <chr> <chr>
## 1     1 x1    y1
## 2     2 x2    y2

```

## 2.2 Filtering joins

**Filtering joins** wählen eine Teilmenge von Zeilen einer Tabelle aus, basierend auf dem Vorkommen von Key-Werten in einer zweiten Tabelle.

- `semi_join(tb1, tb2, by)`: wählt alle Beobachtungen in `tb1`, deren Key in `tb2` vorkommt.
- `anti_join(tb1, tb2, by)`: wählt alle Beobachtungen in `tb1`, deren Key nicht in `tb2` vorkommt.

```

# create tibble of top10 destinations
top_dest <- flights %>%
  count(dest, sort = TRUE) %>%
  head(10)
top_dest
## # A tibble: 10 x 2
##   dest      n
##   <chr> <int>
## 1 ORD    17283
## 2 ATL    17215
## 3 LAX    16174
## 4 BOS    15508
## # ... with 6 more rows

```

```

flights %>%
  semi_join(top_dest)
## Joining, by = "dest"
## # A tibble: 141,145 x 19
##   year month   day dep_time sched_dep_time dep_delay arr_time sched_arr_time
##   <int> <int> <int>    <int>          <int>    <dbl>    <int>          <int>
## 1 2013     1     1      542          540      2     923          850
## 2 2013     1     1      554          600     -6     812          837
## 3 2013     1     1      554          558     -4     740          728
## 4 2013     1     1      555          600     -5     913          854
## # ... with 141,141 more rows

# dies entspricht
flights %>%
  filter(dest %in% top_dest$dest)
## # A tibble: 141,145 x 19
##   year month   day dep_time sched_dep_time dep_delay arr_time sched_arr_time
##   <int> <int> <int>    <int>          <int>    <dbl>    <int>          <int>
## 1 2013     1     1      542          540      2     923          850
## 2 2013     1     1      554          600     -6     812          837
## 3 2013     1     1      554          558     -4     740          728
## 4 2013     1     1      555          600     -5     913          854
## # ... with 141,141 more rows

# finde flights mit tailnum, die nicht einem Flugzeug aus planes zugeordnet werden kann
flights %>%
  anti_join(planes, by = "tailnum") %>%
  count(tailnum, sort = TRUE)
## # A tibble: 722 x 2
##   tailnum     n
##   <chr>   <int>
## 1 <NA>     2512
## 2 N725MQ     575
## 3 N722MQ     513
## 4 N723MQ     507
## # ... with 718 more rows

```

## 3 Relationale Datenbanken

Wir wenden uns wieder kurz der Theorie **Relationalen Datenbanken** und der **relationalen Algebra** zu und zeigen, wie einige der oben vorgestellten Funktionen im Kontext von Relationen formalisiert werden können.

### 3.1 Erinnerung Grundbegriffe

Eine endliche Folge von Mengen  $(W_1, \dots, W_m)$  heißt **Relationenschema**.

Eine endliche Teilmenge  $R \subseteq W_1 \times \dots \times W_m$  heißt **Relation** des Relationenschemas  $(W_1, \dots, W_m)$ . In diesem Zusammenhang heißen die  $W_j$  auch **Wertebereiche**.

Ein Element  $t \in R$  einer Relation  $R$  heißt **Tupel**.

### 3.2 Operationen

Aus dem vorigen Kapitel sind bereits folgende Operationen auf Relationen bekannt:

- Vereinigung `union()`
- Schnitt `intersect()`
- Differenz: `setdiff()`
- Symmetrische Differenz
- Kartesisches Produkt: `tidy::crossing()`

### 3.2.1 Projektion

Seien  $R \subseteq W_1 \times \cdots \times W_m$  eine Relation und  $\beta = (j_1, \dots, j_\ell)$  mit  $j_1, \dots, j_\ell \in \{1, \dots, m\}$ .

Die Projektion von  $R$  auf  $\beta$  ist

- $\pi_\beta(R) = \{t_\beta \mid t \in R\}$ , wobei  $t_\beta = (t_{j_1}, \dots, t_{j_\ell})$  für  $t = (t_1, \dots, t_m)$ .

```
# Schema (integer, character, logical)
R <- tibble(
  x = c(1L, 1L, 1L, 2L),
  y = letters[1:4],
  z = c(T, F, F, T))
R
## # A tibble: 4 x 3
##       x     y     z
##   <int> <chr> <lgl>
## 1     1     a     TRUE
## 2     1     b    FALSE
## 3     1     c    FALSE
## 4     2     d     TRUE
beta <- c(1,3)
unique(R[beta])
## # A tibble: 3 x 2
##       x     z
##   <int> <lgl>
## 1     1  TRUE
## 2     1 FALSE
## 3     2  TRUE
# oder
R %>%
  select(all_of(beta)) %>%
  distinct()
## # A tibble: 3 x 2
##       x     z
##   <int> <lgl>
## 1     1  TRUE
## 2     1 FALSE
## 3     2  TRUE
```

### 3.2.2 Selektion

Seien  $R \subseteq W_1 \times \cdots \times W_m$  und  $p$  ein Prädikat, dh eine (nicht notwendigerweise endliche) Teilmenge  $p \subseteq W_1 \times \cdots \times W_m$ .

In der Regel wird  $p$  durch einen logischen Ausdruck  $A$  beschrieben, etwa  $t_2 > 0$ . Dies steht für die Menge  $\{t \in W_1 \times \cdots \times W_m \mid t_2 > 0\}$ .

- Selektion  $\sigma_p(R) := \{t \in R \mid t \in p\}$ ,  $\sigma_A(R) := \{t \in R \mid t \text{ erfüllt } A\}$ .

```
R <- tibble(
  x = c(1:4),
  y = letters[1:4])
R[R$x>2,]
## # A tibble: 2 x 2
##       x     y
##   <int> <chr>
## 1     3     c
## 2     4     d
# oder
R %>% filter(x > 2)
## # A tibble: 2 x 2
##       x     y
##   <int> <chr>
## 1     3     c
## 2     4     d
```

**Achtung:** Eine Selektion im Sinne relationaler Datenbanken entspricht der `dplyr`-Funktion `filter()`, eine Projektion entspricht `select()`.

### 3.2.3 Verbund (Join)

Seien  $R \subseteq W_1 \times \dots \times W_\ell$  und  $S \subseteq W_{\ell+1} \times \dots \times W_m$  Relationen und  $p \subseteq W_1 \times \dots \times W_m$  ein Prädikat.

- Verbund  $R \bowtie_p S := \sigma_p(R \times S)$ .

Ein natürlicher Verbund (natural join) ist ein Verbund mit Gleichheitsbedingung zusammen mit einer Projektion zur Entfernung der duplizierten Spalten.

Seien  $R \subseteq U_1 \times \dots \times U_\ell \times W_1 \times \dots \times W_m$  und  $S \subseteq W_1 \times \dots \times W_m \times V_1 \times \dots \times V_k$ .

- natürlicher Verbund  $R \bowtie S := \{(u_1, \dots, u_\ell, w_1, \dots, w_m, v_1, \dots, v_k) | (u_1, \dots, u_\ell, w_1, \dots, w_m) \in R \wedge (w_1, \dots, w_m, v_1, \dots, v_k) \in S\}$ .

```
R <- tibble(
  x = c("x1", "x2", "x3"),
  key = 1:3)
S <- tibble(
  key = c(1, 2, 4),
  y = c("y1", "y2", "y3"))
inner_join(R, S, by="key") #natürlicher Verbund
## # A tibble: 2 x 3
##       x     key     y
##   <chr> <dbl> <chr>
## 1 x1     1     y1
## 2 x2     2     y2
```

- Semi Join  $R \bowtie S := \{(u_1, \dots, u_\ell, w_1, \dots, w_m) \in R | \exists v_j \in V_j: (w_1, \dots, w_m, v_1, \dots, v_k) \in S\}$

```
R <- tibble(
  x = c("x1", "x2", "x3"),
  key = 1:3)
S <- tibble(
  key = c(1, 2, 4),
  y = c("y1", "y2", "y3"))
semi_join(R, S, by="key")
## # A tibble: 2 x 2
```

```
##   x      key
## <chr> <int>
## 1 x1      1
## 2 x2      2
```

### 3.2.4 Division

Seien  $R \subseteq W_1 \times \cdots \times W_\ell \times W_{\ell+1} \times \cdots \times W_m$  und  $S \subseteq W_{\ell+1} \times \cdots \times W_m$  Relationen. Definiere  $\beta = (1, \dots, \ell)$ .

- Division:  $R \div S := \pi_\beta(R) \setminus \pi_\beta((\pi_\beta(R) \times S) \setminus R)$ .

Die Division  $R \div S$  enthält alle Tupel  $u \in W_1 \times \cdots \times W_\ell$  mit  $\{u\} \times S \subseteq R$ .

### 3.2.5 Rechenregeln

Mit einer mathematischen Definition ist es einfacher Rechenregeln der Relations- oder Tabellen-Operationen zu finden.

Diese Rechenregeln können genutzt werden, um eine Operationsfolge durch eine effizientere Operationsfolge mit dem gleichen Ergebnis zu ersetzen.

Hier sind ein paar Beispiele für Rechenregeln:

Seien  $R, S, T$  Relationen und  $A, B, C$  Ausdrücke, die eine Bedingung für eine Selektion  $\sigma$  beschreiben.

- $(R \times S) \cup (R \times T) = R \times (S \cup T)$
- $(R \times S) \div S = R$  für kompatible  $R, S$  im Sinne der Definition von Division
- $\sigma_A(R) = \sigma_A(\sigma_A(R))$
- $\sigma_A(\sigma_B(R)) = \sigma_B(\sigma_A(R))$
- $\sigma_{A \vee B}(R) = \sigma_A(R) \cup \sigma_A(R)$
- $\sigma_{A \wedge B}(R) = \sigma_A(\sigma_B(R))$
- Teile  $D = A \wedge B \wedge C$  so auf, dass  $A$  nur Bedingungen ab  $R$  und  $B$  nur Bedingungen an  $S$  stellt. Dann gilt  $\sigma_{A \wedge B \wedge C}(R \times S) = \sigma_C(\sigma_A(R) \times \sigma_B(S))$ .
- $\sigma_A(R \setminus S) = \sigma_A(R) \setminus \sigma_A(S) = \sigma_A(R) \setminus S$
- $\sigma_A(R \cup S) = \sigma_A(R) \cup \sigma_A(S)$
- $\sigma_A(R \cap S) = \sigma_A(R) \cap \sigma_A(S) = \sigma_A(R) \cap S$
- $\pi_\beta(\sigma_A(R)) = \sigma_A(\pi_\beta(R))$  für eine Menge an Spalten  $\beta$ , sodass  $A$  nur Bedingungen an die Spalten in  $\beta$  stellt
- $\pi_\alpha(\pi_\beta(R)) = \pi_\alpha(R)$  für  $\alpha \subseteq \beta$
- $\pi_\beta(R \cup S) = \pi_\beta(R) \cup \pi_\beta(S)$

# V08 – tidyverse

10. Mai 2021

## Contents

<b>1 Tidy Data</b>	<b>1</b>
1.1 Pivoting . . . . .	3
1.2 separate and unite . . . . .	5
1.3 Fehlende Werte . . . . .	7
<b>2 Normalisierung Relationaler Datenbanken</b>	<b>9</b>

## 1 Tidy Data

Wir bezeichnen hier mit dem Begriff **Dataset** eine zusammengehörende Menge von Tabellen.

**Bemerkung:** Als Kandidat für eine Übersetzung des englischen Begriffs *data set* bietet sich *Datensatz* an. Dieser Begriff wird jedoch in der Informatik in Zusammenhang mit der Theorie von Datenbanken anders verwendet als in der Statistik, siehe [https://de.wikipedia.org/wiki/Datensatz#Abweichende\\_Bedeutung\\_des\\_Begriffs](https://de.wikipedia.org/wiki/Datensatz#Abweichende_Bedeutung_des_Begriffs).

Ein Dataset ist **tidy** falls folgende Bedingungen erfüllt sind:

- Jede experimentelle Variable / Messgröße bildet eine benannte Spalte.
- Jede Beobachtung / Messung bildet eine (unbenannte) Zeile.
- Jedes Experiment / jede Art Beobachtungstyp bildet eine Tabelle.

(aus Wickham, Hadley (2014). Tidy Data. *Journal of Statistical Software* **59**. <https://vita.had.co.nz/papers/tidy-data.pdf>)

Diese Beschreibung ist bewusst vage gehalten. Einen formalen Ansatz stellen wir kurz am Ende der Lektion vor.

Um dennoch diese Idee von **tidy data** zu verstehen, sehen wir uns einige Beispiele an.

Folgende Tabellen enthalten jeweils die Anzahl an Tuberkulose-Fällen in Afghanistan, Brasilien und China zwischen 1999 und 2000 (WHO).

```
library(tidyverse)
table1
## # A tibble: 6 x 4
##   country     year   cases population
##   <chr>     <int>   <int>      <int>
## 1 Afghanistan 1999     745 19987071
## 2 Afghanistan 2000    2666 20595360
## 3 Brazil      1999  37737 172006362
## 4 Brazil      2000  80488 174504898
## 5 China       1999 212258 1272915272
## 6 China       2000 213766 1280428583
table2
## # A tibble: 12 x 4
```

```

##   country      year type      count
##   <chr>      <int> <chr>      <int>
## 1 Afghanistan 1999 cases      745
## 2 Afghanistan 1999 population 19987071
## 3 Afghanistan 2000 cases     2666
## 4 Afghanistan 2000 population 20595360
## 5 Brazil      1999 cases     37737
## 6 Brazil      1999 population 172006362
## 7 Brazil      2000 cases     80488
## 8 Brazil      2000 population 174504898
## 9 China       1999 cases     212258
## 10 China      1999 population 1272915272
## 11 China      2000 cases     213766
## 12 China      2000 population 1280428583
table3
## # A tibble: 6 x 3
##   country      year rate
##   <chr>      <int> <chr>
## 1 Afghanistan 1999 745/19987071
## 2 Afghanistan 2000 2666/20595360
## 3 Brazil      1999 37737/172006362
## 4 Brazil      2000 80488/174504898
## 5 China       1999 212258/1272915272
## 6 China       2000 213766/1280428583
table4a # cases
## # A tibble: 3 x 3
##   country      `1999` `2000`
##   <chr>      <int>  <int>
## 1 Afghanistan    745    2666
## 2 Brazil        37737   80488
## 3 China         212258  213766
table4b # population
## # A tibble: 3 x 3
##   country      `1999`      `2000`
##   <chr>      <int>      <int>
## 1 Afghanistan  19987071  20595360
## 2 Brazil      172006362  174504898
## 3 China       1272915272 1280428583
table5
## # A tibble: 6 x 4
##   country      century year  rate
##   <chr>      <chr>   <chr> <chr>
## 1 Afghanistan 19      99    745/19987071
## 2 Afghanistan 20      00    2666/20595360
## 3 Brazil      19      99    37737/172006362
## 4 Brazil      20      00    80488/174504898
## 5 China       19      99    212258/1272915272
## 6 China       20      00    213766/1280428583

```

Am ehesten entspricht `table1` den Voraussetzungen für tidy data.

Ein Beispiel für ein größeres tidy Dataset ist das Paket `nycflights13`.

Es enthält 5 Tibbles, die die Flüge von New York City aus im Jahr 2013 beschreiben:

- **airlines**: Airline names
- **airports**: Airport metadata
- **flights**: Flights data
- **planes**: Plane metadata
- **weather**: Hourly weather data

## 1.1 Pivoting

Der erste Schritt zum Erstellen eines tidy Datasets ist das Feststellen der Variablen und Beobachtungen.

Im zweiten Schritt wird meist eines der folgenden zwei typischen Problemen gelöst:

- Eine Variable ist über mehrere Spalten verteilt.
- Eine Beobachtung ist über mehrere Zeilen verteilt.

`pivot_longer(tb, cols, names_to="name", values_to="value")` nimmt die in `cols` beschriebenen Spalten des Tibbles `tb` und schreibt deren Werte in eine Spalte namens `"value"` und die zugehörigen Spaltennamen in eine Spalte namens `"name"`.

Mit `pivot_longer()` erhöht sich die Zeilenzahl (oder lässt sie gleich), die Tabelle wird also *longer*.

```
table4a # cases
## # A tibble: 3 x 3
##   country     `1999` `2000`
## * <chr>       <int>   <int>
## 1 Afghanistan    745    2666
## 2 Brazil        37737   80488
## 3 China         212258  213766

table4b # population
## # A tibble: 3 x 3
##   country     `1999`     `2000`
## * <chr>       <int>       <int>
## 1 Afghanistan 19987071  20595360
## 2 Brazil      172006362 174504898
## 3 China       1272915272 1280428583

table4a %>%
  pivot_longer(c(`1999`, `2000`), names_to = "year", values_to = "cases") ->
  tidy4a

table4b %>%
  pivot_longer(c(`1999`, `2000`), names_to = "year", values_to = "population") ->
  tidy4b
```

```

## 3 Brazil      1999  172006362
## 4 Brazil      2000  174504898
## 5 China       1999  1272915272
## 6 China       2000  1280428583
left_join(tidy4a, tidy4b)
## Joining, by = c("country", "year")
## # A tibble: 6 x 4
##   country     year   cases population
##   <chr>      <chr> <int>      <int>
## 1 Afghanistan 1999    745  19987071
## 2 Afghanistan 2000   2666  20595360
## 3 Brazil      1999  37737  172006362
## 4 Brazil      2000  80488  174504898
## 5 China       1999  212258 1272915272
## 6 China       2000  213766 1280428583

```

Spalten können wie `dplyr::select()` angegeben werden, also auch mit den *Select-Helpers*.

```

table4a %>%
  pivot_longer(matches("[0-9]{4}"), names_to = "year", values_to = "cases")
## # A tibble: 6 x 3
##   country     year   cases
##   <chr>      <chr> <int>
## 1 Afghanistan 1999    745
## 2 Afghanistan 2000   2666
## 3 Brazil      1999  37737
## 4 Brazil      2000  80488
## 5 China       1999  212258
## 6 China       2000  213766

```

Das Gegenstück zu `pivot_longer()` ist `pivot_wider()`.

`pivot_wider(tb, names_from=name, values_from=value)` erzeugt für jeden in der Spalte `name` von `tb` auftretenden Wert eine Spalte mit diesem Wert als Namen und den zugehörigen Werten aus der Spalte `value`.

Mit `pivot_wider()` erhöht sich die Spaltenzahl (oder bleibt gleich), die Tabelle wird also *wider*.

```

table2
## # A tibble: 12 x 4
##   country     year type      count
##   <chr>      <int> <chr>     <int>
## 1 Afghanistan 1999 cases      745
## 2 Afghanistan 1999 population 19987071
## 3 Afghanistan 2000 cases     2666
## 4 Afghanistan 2000 population 20595360
## 5 Brazil      1999 cases     37737
## 6 Brazil      1999 population 172006362
## 7 Brazil      2000 cases     80488
## 8 Brazil      2000 population 174504898
## 9 China       1999 cases     212258
## 10 China      1999 population 1272915272
## 11 China      2000 cases     213766
## 12 China      2000 population 1280428583
table2 %>%
  pivot_wider(names_from = type, values_from = count)
## # A tibble: 6 x 4

```

```

##   country      year  cases population
##   <chr>      <int> <int>      <int>
## 1 Afghanistan 1999    745 19987071
## 2 Afghanistan 2000   2666 20595360
## 3 Brazil      1999 37737 172006362
## 4 Brazil      2000 80488 174504898
## 5 China       1999 212258 1272915272
## 6 China       2000 213766 1280428583

```

Beide Funktionen haben noch weitere Argumente, die ihr Verhalten in speziellen Situationen beeinflussen. Hierzu sei auf die Dokumentation `?pivot_wider` und `?pivot_longer` verwiesen.

## 1.2 separate and unite

`separate(tb, col, into, sep =",")` macht aus einer `character`-Spalte `col` mehrere (mit den Namen `into`), indem an bestimmten Symbolen (oder Positionen) getrennt wird.

Das Argument `sep` gibt an, an welchem Zeichen getrennt werden soll. Der Wert von `sep` wird als regulärer Ausdruck interpretiert.

```

table3
## # A tibble: 6 x 3
##   country      year rate
##   <chr>      <int> <chr>
## 1 Afghanistan 1999 745/19987071
## 2 Afghanistan 2000 2666/20595360
## 3 Brazil      1999 37737/172006362
## 4 Brazil      2000 80488/174504898
## 5 China       1999 212258/1272915272
## 6 China       2000 213766/1280428583
table3 %>%
  separate(rate, into = c("cases", "population"), sep = "/")
## # A tibble: 6 x 4
##   country      year cases population
##   <chr>      <int> <chr>      <chr>
## 1 Afghanistan 1999 745 19987071
## 2 Afghanistan 2000 2666 20595360
## 3 Brazil      1999 37737 172006362
## 4 Brazil      2000 80488 174504898
## 5 China       1999 212258 1272915272
## 6 China       2000 213766 1280428583

```

Der Default-Wert von `sep` ist ein regulärer Ausdruck für alle nicht-alphanumerischen Symbole ("`^[:alnum:]`+").

```

table3 %>%
  separate(rate, into = c("cases", "population"))
## # A tibble: 6 x 4
##   country      year cases population
##   <chr>      <int> <chr>      <chr>
## 1 Afghanistan 1999 745 19987071
## 2 Afghanistan 2000 2666 20595360
## 3 Brazil      1999 37737 172006362
## 4 Brazil      2000 80488 174504898
## 5 China       1999 212258 1272915272
## 6 China       2000 213766 1280428583

```

Die neuen Spalten sind — wie die alte — vom Typ `character`. Mit der Option `convert = TRUE` aktivieren wir automatische Typumwandlung.

```
table3 %>%
  separate(rate, into = c("cases", "population"), convert = TRUE)
## # A tibble: 6 x 4
##   country     year   cases population
##   <chr>     <int>   <int>      <int>
## 1 Afghanistan 1999     745 19987071
## 2 Afghanistan 2000    2666 20595360
## 3 Brazil      1999 37737 172006362
## 4 Brazil      2000  80488 174504898
## 5 China       1999 212258 1272915272
## 6 China       2000 213766 1280428583
```

Alternativ zu `separate()` kann `extract()` genutzt werden. Anstatt `sep` ist hier das vierte Argument `regex` — ein regulärer Ausdruck mit einer RegEx-Gruppe pro Element von `into`.

```
table3 %>%
  extract(rate, into = c("cases", "population"), regex="([0-9]*)/([0-9]*)", convert = TRUE)
## # A tibble: 6 x 4
##   country     year   cases population
##   <chr>     <int>   <int>      <int>
## 1 Afghanistan 1999     745 19987071
## 2 Afghanistan 2000    2666 20595360
## 3 Brazil      1999 37737 172006362
## 4 Brazil      2000  80488 174504898
## 5 China       1999 212258 1272915272
## 6 China       2000 213766 1280428583
```

Das Gegenstück zu `separate()` ist `unite()`.

`unite(tb, "new_col", col_1, ..., col_n, sep="")` Fügt die Spalten `col_1, ..., col_n` zu einer neuen `character`-Spalte `new_col` zusammen. Die Werte der einzelnen Spalten werden dabei durch `sep` getrennt.

```
table5
## # A tibble: 6 x 4
##   country     century year   rate
##   <chr>     <chr>    <chr> <chr>
## 1 Afghanistan 19      99  745/19987071
## 2 Afghanistan 20      00  2666/20595360
## 3 Brazil      19      99  37737/172006362
## 4 Brazil      20      00  80488/174504898
## 5 China       19      99  212258/1272915272
## 6 China       20      00  213766/1280428583
table5 %>%
  unite("year", century, year, sep=" ")
## # A tibble: 6 x 3
##   country     year   rate
##   <chr>     <chr> <chr>
## 1 Afghanistan 1999 745/19987071
## 2 Afghanistan 2000 2666/20595360
## 3 Brazil      1999 37737/172006362
## 4 Brazil      2000 80488/174504898
## 5 China       1999 212258/1272915272
```

```

## 6 China      2000 213766/1280428583
table5 %>% # convert year to integer
  unite("year", century, year, sep="") %>%
  mutate_at("year", as.integer)
## # A tibble: 6 x 3
##   country      year  rate
##   <chr>        <int> <chr>
## 1 Afghanistan  1999 745/19987071
## 2 Afghanistan  2000 2666/20595360
## 3 Brazil       1999 37737/172006362
## 4 Brazil       2000 80488/174504898
## 5 China        1999 212258/1272915272
## 6 China        2000 213766/1280428583

```

Auch hier verweisen wir wieder auf `?separate`, `?extract`, `?unite` für ausführlichere Informationen.

### 1.3 Fehlende Werte

```

stocks <- tibble(
  year    = c(2015, 2015, 2015, 2015, 2016, 2016, 2016),
  qtr     = c( 1,    2,    3,    4,    2,    3,    4),
  return  = c(1.88, 0.59, 0.35,  NA, 0.92, 0.17, 2.66)
)

```

Fehlende Werte können explizit sein (NA oder anderer Wert, der Fehlen markiert) oder implizit (taucht nicht in der Tabelle auf).

In `stocks` fehlt Quartal 2015.4 explizit. Quartal 2016.1 fehlt implizit.

```

# manche Operationen machen implizit fehlende Werte explizit
stocks %>%
  pivot_wider(names_from=year, values_from=return)
## # A tibble: 4 x 3
##   qtr `2015` `2016`
##   <dbl>   <dbl>   <dbl>
## 1 1     1.88    NA
## 2 2     0.59    0.92
## 3 3     0.35    0.17
## 4 4     NA      2.66

```

Die Funktion `complete()` vervollständigt die Beobachtungen, sodass es für alle Kombinationen der Werte der übergebenen Spalten eine Zeile gibt. Implizit fehlende Werte werden dadurch explizit.

```

stocks %>%
  complete(year, qtr) ->
  stocks_cmpl
stocks_cmpl
## # A tibble: 8 x 3
##   year  qtr return
##   <dbl> <dbl>   <dbl>
## 1 2015  1     1.88
## 2 2015  2     0.59
## 3 2015  3     0.35
## 4 2015  4     NA
## 5 2016  1     NA

```

```
## 6 2016      2  0.92
## 7 2016      3  0.17
## 8 2016      4  2.66
```

Um NA-Werte durch andere Werte zu ersetzen nutze `replace_na()`.

```
stocks_cmpl %>%
  replace_na(list(return = 99)) ->
  stocks_99
stocks_99
## # A tibble: 8 x 3
##   year   qtr  return
##   <dbl> <dbl>  <dbl>
## 1 2015     1    1.88
## 2 2015     2    0.59
## 3 2015     3    0.35
## 4 2015     4    99
## 5 2016     1    99
## 6 2016     2    0.92
## 7 2016     3    0.17
## 8 2016     4    2.66
```

Um Zeilen mit NA-Werten zu entfernen, nutze `drop_na()`.

```
stocks_cmpl %>%
  drop_na()
## # A tibble: 6 x 3
##   year   qtr  return
##   <dbl> <dbl>  <dbl>
## 1 2015     1    1.88
## 2 2015     2    0.59
## 3 2015     3    0.35
## 4 2016     2    0.92
## 5 2016     3    0.17
## 6 2016     4    2.66
```

Oft werden fehlende Werte im ursprünglichen Dataset nicht mit NA sondern einem anderen Wert markiert. `na_if()` ersetzt bestimmte Werte in einem Vektor mit NA. Damit wir `na_if()` auf Tabellen anwenden können, nutzen wir `mutate()`.

```
stocks_99 %>%
  mutate(return = na_if(return, 99))
## # A tibble: 8 x 3
##   year   qtr  return
##   <dbl> <dbl>  <dbl>
## 1 2015     1    1.88
## 2 2015     2    0.59
## 3 2015     3    0.35
## 4 2015     4    NA
## 5 2016     1    NA
## 6 2016     2    0.92
## 7 2016     3    0.17
## 8 2016     4    2.66
```

## 2 Normalisierung Relationaler Datenbanken

Wir wenden uns wieder kurz der Theorie **Relationalen Datenbanken** zu und deuten auf den Zusammenhang des Begriffes *tidy data* und der Normalformen von Datenbanken hin.

Ziel der Normalisierung eines Datasets ist es Redundanzen zu verringern.

Redundanzen können bei Änderungen von Werten Widersprüche (sogenannte *Anomalien*) erzeugen.

Die verschiedenen **Normalformen** von Datenbanken sind ein formaler Zugang zu einer Idee, die dem Konzept von *tidy data* ähnelt.

Wir geben hier die Normalformen in leicht vereinfachter Form wider. Zunächst benötigen wir jedoch noch weitere Definitionen.

Ein **Schlüsselkandidat** ist eine minimale Menge an Spalten, die die Zeilen einer Tabelle eindeutig identifiziert.

Eine Spalte  $S \in \{1, \dots, m\}$  ist **funktional abhängig** von einer Menge an Spalten  $\beta \subset \{1, \dots, m\}$ , wenn sich (die Werte in)  $S$  als Funktion (der Werte) der Spalten  $\beta$  schreiben lässt.

Ist  $S \notin \beta$ , dann heißt  $\beta$  **Determinante** für  $S$ .

Jede Nicht-Schlüssel-Spalte (Spalte die nicht Teil eines Schlüsselkandidaten ist) ist funktional abhängig von einem Schlüsselkandidaten.

```
tb <- tibble(
  Name = c("Alice", "Bob", "Claire", "Dave"),
  Ort = c("Ahausen", "C-Stadt", "C-Stadt", "C-Stadt"),
  PLZ = c(1111, 2222, 2223, 2223))
# Name ist Schlüsselkandidat
# PLZ ist Determinante, da Ort funktional abhängig ist von PLZ
tb
## # A tibble: 4 x 3
##   Name     Ort      PLZ
##   <chr>   <chr>   <dbl>
## 1 Alice   Ahausen  1111
## 2 Bob     C-Stadt  2222
## 3 Claire  C-Stadt  2223
## 4 Dave    C-Stadt  2223

tb <- tibble(
  weekday = c("Friday", "Monday", "Tuesday"),
  day = c(1, 1, 15),
  month = c(5, 6, 6),
  year = c(2020, 2020, 2021))
# Für diese konkrete Tabelle ist weekday Schlüsselkandidat
# Jedoch wäre dies nicht mehr der Fall, wenn noch einige weitere Einträge in der Tabelle hinzukommen
# Dann wäre evtl {day, month, year} ein nützlicherer Schlüsselkandidat
tb
## # A tibble: 3 x 4
##   weekday   day   month   year
##   <chr>     <dbl> <dbl>   <dbl>
## 1 Friday      1     5   2020
## 2 Monday      1     6   2020
## 3 Tuesday    15     6   2021
```

Die Normalformen werden meist mit 1NF, 2NF, ... für die  $k$ -te Normalform bezeichnet oder BCNF für die Boyce-Codd-Normalform.

**1NF:** Jede Spalte ist *atomisch*.

Ein Spalte ist *atomisch* wenn sie nicht sinnvoll in weitere Spalten aufgeteilt werden kann.

```
lectures <- tibble(
  lecture_id = c(12, 34, 56),
  lecture = c("Analysis (Gauß)", "Lineare Algebra (Euclid)", "Geometrie (Euclid)"),
  year = c(1815, -310, -309))
lectures %>%
  extract(lecture, into = c("lecture", "teacher"), regex="^([^\(\)]*) \\\(([^\\()]*))\\\\")  
## # A tibble: 3 x 4
##   lecture_id lecture      teacher  year
##       <dbl> <chr>       <chr>    <dbl>
## 1       12 Analysis    Gauß     1815
## 2       34 Lineare Algebra Euclid   -310
## 3       56 Geometrie   Euclid   -309
```

**2NF:** 1NF und keine Nicht-Schlüssel-Spalte ist funktional abhängig von einer echten Teilmenge eines Schlüsselkandidaten.

```
tb <- tibble(
  weekend = c(F, F, F, T, F),
  weekday = c("Friday", "Monday", "Monday", "Saturday", "Friday"),
  first_monday = c(F, T, F, F, F),
  month_name = month.name[c(5,6,6,5,5)],
  day = c(1,1,14,15,15),
  month = c(5,6,6,5,5),
  year = c(2020, 2020, 2021, 2021, 2020))
tb
## # A tibble: 5 x 7
##   weekend weekday first_monday month_name   day month  year
##   <lgl>   <chr>   <lgl>       <chr>     <dbl> <dbl> <dbl>
## 1 FALSE   Friday  FALSE       May          1     5  2020
## 2 FALSE   Monday  TRUE        June         1     6  2020
## 3 FALSE   Monday  FALSE       June         14    6  2021
## 4 TRUE    Saturday FALSE       May          15    5  2021
## 5 FALSE   Friday  FALSE       May          15    5  2020
# Schlüsselkandidat: (day, month, year)
# month_name funktional abhängig von month
tb %>% select(-month_name) -> tb2
tb %>% select(month_name, month) %>% distinct() -> months
tb2
## # A tibble: 5 x 6
##   weekend weekday first_monday   day month  year
##   <lgl>   <chr>   <lgl>     <dbl> <dbl> <dbl>
## 1 FALSE   Friday  FALSE       1     5  2020
## 2 FALSE   Monday  TRUE        1     6  2020
## 3 FALSE   Monday  FALSE       14    6  2021
## 4 TRUE    Saturday FALSE       15    5  2021
## 5 FALSE   Friday  FALSE       15    5  2020
months
## # A tibble: 2 x 2
##   month_name month
##   <chr>       <dbl>
## 1 May          5
## 2 June         6
```

**3NF:** 2NF und keine Nicht-Schlüssel-Spalte ist funktional abhängig von anderen Nicht-Schlüssel-Spalten.  
Äquivalent: 2NF und keine Menge an Nicht-Schlüssel-Spalten bilden eine Determinante.

```
tb2
## # A tibble: 5 x 6
##   weekend weekday first_monday   day month  year
##   <lgl>   <chr>   <lgl>     <dbl> <dbl> <dbl>
## 1 FALSE   Friday  FALSE        1     5  2020
## 2 FALSE   Monday  TRUE         1     6  2020
## 3 FALSE   Monday  FALSE        14    6  2021
## 4 TRUE    Saturday FALSE        15    5  2021
## 5 FALSE   Friday  FALSE        15    5  2020
# weekday ist Determinante für weekend
tb2 %>% select(-weekend) -> tb3
tb2 %>% select(weekend, weekday) %>% distinct() -> weekend
tb3
## # A tibble: 5 x 5
##   weekday first_monday   day month  year
##   <chr>   <lgl>     <dbl> <dbl> <dbl>
## 1 Friday  FALSE        1     5  2020
## 2 Monday  TRUE         1     6  2020
## 3 Monday  FALSE        14    6  2021
## 4 Saturday FALSE        15    5  2021
## 5 Friday  FALSE        15    5  2020
weekend
## # A tibble: 3 x 2
##   weekend weekday
##   <lgl>   <chr>
## 1 FALSE   Friday
## 2 FALSE   Monday
## 3 TRUE    Saturday
```

**BCNF:** 3NF und jede Determinante ist ein Schlüsselkandidat.

```
tb3 %>% select(day, weekday, first_monday) %>% distinct() -> first_monday
tb3 %>% select(-first_monday) -> tb4
tb4
## # A tibble: 5 x 4
##   weekday   day month  year
##   <chr>     <dbl> <dbl> <dbl>
## 1 Friday      1     5  2020
## 2 Monday      1     6  2020
## 3 Monday     14    6  2021
## 4 Saturday   15    5  2021
## 5 Friday     15    5  2020
first_monday
## # A tibble: 5 x 3
##   day weekday first_monday
##   <dbl> <chr>   <lgl>
## 1 1     Friday  FALSE
## 2 1     Monday  TRUE
## 3 14    Monday  FALSE
## 4 15    Saturday FALSE
## 5 15    Friday  FALSE
```

Für eine ausführlichere Beschreibung dieser und weiterer Normalformen, siehe [https://en.wikipedia.org/wiki/Database\\_normalization](https://en.wikipedia.org/wiki/Database_normalization)

Das moderne und schwammige, aber in der Anwendung nützliche Konzept von *tidy data* ist maßgeblich von den älteren (1970er) und formalen Definition der Normalformen beeinflusst.

Oft ist es sinnvoll, eine neues Dataset zunächst zu normalisieren und in einer normalisierten Form abzuspeichern. Während der Analyse wird das Dataset jedoch oft aus praktischen Gründen nicht in einer Normalform sein.

# V09 – Umgebungen

10. Mai 2021

## Contents

<b>1</b>	<b>Motivation</b>	<b>1</b>
<b>2</b>	<b>Grundbegriffe</b>	<b>2</b>
<b>3</b>	<b>Umgebungsobjekte</b>	<b>2</b>
<b>4</b>	<b>Umgebungsdiagramme</b>	<b>6</b>
<b>5</b>	<b>Umgebungen für Funktionen</b>	<b>8</b>
5.1	Funktionsumgebung . . . . .	8
5.2	Ausführungsumgebung . . . . .	10
5.3	Aufrufende Umgebung . . . . .	14
<b>6</b>	<b>Scoping-Prinzipien</b>	<b>14</b>
6.1	Name-Masking . . . . .	14
6.2	Dynamic Lookup . . . . .	15
6.3	Neuanfang . . . . .	16
<b>7</b>	<b>Zuweisungsoperatoren</b>	<b>17</b>
<b>8</b>	<b>Rekursion über Umgebungen</b>	<b>18</b>
<b>9</b>	<b>Umgebungen für Pakete</b>	<b>19</b>
9.1	Suchpfad . . . . .	19
9.2	Namespace . . . . .	20
9.3	Locked Environments . . . . .	21
<b>10</b>	<b>Übersicht Umgebungen</b>	<b>22</b>
<b>11</b>	<b>Lazy Evaluation 2</b>	<b>22</b>

## 1 Motivation

Welche Ausgaben erzeugen folgende Code-Beispiele?

```
# Beispiel 1
x <- 2
y <- 2
f1 <- function() {
  x <- 1
  c(x, y)
}
f1()
## Error, c(1, 1), c(1, 2), c(2, 2) ?
```

```

# Beispiel 2
f2 <- function() {
  if (!exists("a")) { # Existiert eine Variable mit dem Namen a?
    a <- 1
  } else {
    a <- a + 1
  }
  a
}
f2()
## 1
f2()
### Error, 1, 2, 3 ?

# Beispiel 3
x <- 10
f3 <- function() x + 1
f3()
## 11
x <- 100
f3()
### Error, 1, 11, 101 ?

```

## 2 Grundbegriffe

**Namensbindung** (*name binding*): Mit NAME  $\leftarrow$  WERT wird der Wert WERT an den Namen NAME gebunden. In diesem Zusammenhang werden Namen auch als *Symbol* bezeichnet.

Der Gültigkeitsbereich einer Namensbindung wird auch mit **Scope** bezeichnet.

```

x <- 5
f(x)
# hier endet der Scope der Namensbindung von x mit 5
# es beginnt der Scope von x gebunden mit 6
x <- 6
g(x)

```

Die **Auflösung** eines Namens (*name resolution*) bezeichnet das Ersetzen eines Namens mit dem gebundenen Wert.

Scoping-Regeln steuern die Auflösung von Namen. Die Scoping-Regeln von R basieren auf **Umgebungen** (*environments*), einer weiteren Datenstruktur.

Jeder Namensbindung ist genau eine Umgebung zugeordnet. Umgebungen enthalten eine beliebige Anzahl an Namensbindungen.

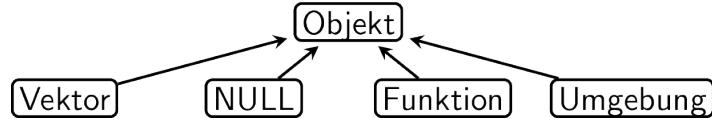
## 3 Umgebungsobjekte

In diesem Kapitel nutzen wir das Paket `rlang`, welches unter anderem einige Funktionen für den Umgang mit Umgebungen bereitstellt.

```

# install.packages("rlang")
library(rlang)

```



Umgebungen (*environments*) sind Objekte, die Listen mit benannten Elementen ähneln, jedoch 4 Unterschiede aufweisen:

- Jeder Name darf nur einmal auftreten.
- Elemente sind nicht geordnet (kein “*i*-ter Eintrag”).
- Jede Umgebung (außer der leeren Umgebung) hat eine Eltern-Umgebung (*parent environment*).
- Umgebungen werden nicht kopiert, wenn sie geändert werden (kein *copy-on-modify* sondern *Referenz-Semantik*).

Mit `rlang:::env()` wird eine neue Umgebung analog zu `list()` erzeugt.

```

l1 <- list(
  a = FALSE,
  b = "a",
  c = 2.3,
  d = 1:3
)
e1 <- env(
  a = FALSE,
  b = "a",
  c = 2.3,
  d = 1:3
)
  
```

Umgebungen haben Typ und Klasse `environment`. `length()` gibt die Anzahl der Elemente zurück.

```

c(typeof(e1), class(e1))
## [1] "environment" "environment"
length(e1)
## [1] 4
  
```

Für Umgebungen können die Subsetting-Operatoren `[[` und `$` genutzt werden, `[[` jedoch nur mit Strings nicht mit Zahlen, da die Elemente einer Umgebung keine Ordnung haben.

Da Umgebungen anders als etwa Listen einer *Referenz-Semantik* folgen, können zwei Namen die gleiche Umgebung als Wert haben.

```

e1$d
## [1] 1 2 3
e1other <- e1
# e1other ist keine Kopie von e1, sondern eine 'Referenz' auf das selbe Objekt
e1other$d <- 0 # Änderung von e1other bedeutet Änderung des referenzierten Objektes
e1$d # Daher ist auch das von e1 referenziert Objekt geändert
## [1] 0

# vergleiche
l1$d
## [1] 1 2 3
l1other <- l1
l1other$d <- 0
l1$d
## [1] 1 2 3
  
```

Gleichheit von Umgebungen wird mit `identical()` getestet, nicht mit `==`.

```
e1 == e1other
## Error in e1 == e1other: comparison (1) is possible only for atomic and list types
identical(e1,e1other)
## [1] TRUE
```

Wegen der Referenz-Semantik können Umgebungen sich selbst (bzw eine Referenz auf sich selbst) enthalten. Dies ist nicht möglich für Listen.

```
e1$d <- e1
identical(e1$d$d$d$d$d$d, e1)
## [1] TRUE
l1$d <- l1
l1$d$d
## [1] 1 2 3
```

Da `print()` für Umgebungen nicht besonders informativ ist, nutzen wir oft `rlang::env_print()`.

```
e1 # gleich print(e1)
## <environment: 0x0000000015353b08>
env_print(e1)
## <environment: 0000000015353B08>
## parent: <environment: global>
## bindings:
##  * a: <lglt>
##  * b: <chr>
##  * c: <dbl>
##  * d: <env>
```

`rlang::env_names()` gibt die Namen zurück, die eine Umgebung enthält.

```
env_names(e1)
## [1] "a" "b" "c" "d"
```

Die meisten Umgebungen werden nicht vom Nutzer erstellt, sondern von R selbst.

`rlang::empty_env()` gibt die **leere Umgebung** zurück.

```
env_names(empty_env())
## character(0)
```

`rlang::global_env()` gibt die **globale Umgebung**. Die globale Umgebung speichert die Namensbindungen, die in der Konsole erzeugt werden.

```
env_names(global_env())
## [1] "l1"           "e1"           "l1other"       ".Random.seed" "e1other"
```

`rlang::env_label()` gibt das Label einer Umgebung aus. Für spezielle Umgebungen ist das Label ein spezifischer Name, für alle anderen ein *address code*.

```
env_label(empty_env())
## [1] "empty"
env_label(global_env())
## [1] "global"
env_label(e1)
## [1] "0000000015353B08"
```

`rlang::current_env()` ist die **aktuelle Umgebung**, in der die aktuelle Code-Ausführung stattfindet.

Im Gegensatz zur leeren und globalen ist dies keine feste Umgebung. In der Konsole ist `rlang::global_env()` gleich `rlang::current_env()`, im Inneren von Funktionen nicht.

```
env_label(current_env())
## [1] "global"
f <- function() env_label(current_env())
f()
## [1] "000000001C625EB8"
```

Stößt R bei der Code-Ausführung auf einen Variablenamen, muss dieser *aufgelöst* werden, um den zugehörigen Wert zu finden. Dazu wird der Name zuerst in der aktuellen Umgebung gesucht. Falls er dort nicht gefunden wird, wird in der Eltern-Umgebung der aktuellen gesucht. Dies wird wiederholt, bis der Name (und damit der zugehörige Wert) in einer Umgebung gefunden wird (ansonsten ERROR).

Jede Umgebung außer der leeren hat genau eine Eltern-Umgebung. Die Eltern-Umgebung kann explizit mit `rlang::env()` bei der Erzeugung einer Umgebung gesetzt werden. Der Default-Wert ist `rlang::current_env()`.

```
e2a <- env(d = 4, e = 5)
e2b <- env(e2a, a = 1, b = 2, c = 3)
```

`rlang::env_parent()` erlaubt den Zugriff auf die Eltern-Umgebung.

```
env_label(e2a)
## [1] "000000001C8813B0"
env_label(env_parent(e2b))
## [1] "000000001C8813B0"
env_label(env_parent(e2a))
## [1] "global"
env_parent(empty_env())
## Error: The empty environment has no parent
```

`rlang::env_parents()` gibt eine Liste aller “Vorfahren” einer Umgebung bis zur globalen oder leeren Umgebung zurück.

```
e2c <- env(empty_env(), d = 4, e = 5)
e2d <- env(e2c, a = 1, b = 2, c = 3)
env_parents(e2b)
## [[1]] <env: 000000001C8813B0>
## [[2]] $ <env: global>
env_parents(e2d)
## [[1]] <env: 000000001D1ADFF8>
## [[2]] $ <env: empty>
```

Um Namensbindungen von Umgebungen zu verwalten, sind die Subsetting Operatoren `[[`, `[[<-`, `$`, `$<-` und die folgende Funktionen aus dem Paket `rlang` von Nutzen: `env_get()`, `env_get_list()`, `env_bind()`, `env_has()`, `env_unbind()`.

```
e <- empty_env()
e$x <- 1
## Error in e$x <- 1: cannot assign values in the empty environment
e <- env() # erstelle neue Umgebung mit der globalen Umgebung als parent
e$x <- 1
e[["y"]] <- "one"
env_bind(e, z=T)
e$y
## [1] "one"
e[["z"]]
## [1] TRUE
env_get(e, "x")
```

```

## [1] 1
str(env_get_list(e, c("x", "y", "z")))
## List of 3
## $ x: num 1
## $ y: chr "one"
## $ z: logi TRUE
env_has(e, c("w", "x", "y", "z"))
##      w      x      y      z
## FALSE  TRUE  TRUE  TRUE
env_unbind(e, c("x", "z"))
env_has(e, c("w", "x", "y", "z"))
##      w      x      y      z
## FALSE FALSE TRUE FALSE

```

## 4 Umgebungsdiagramme

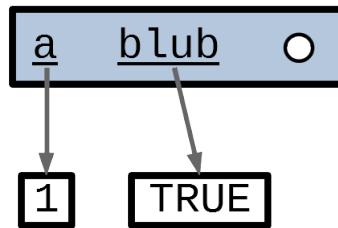
Wir visualisieren die Struktur der Umgebungen und ihrer Elemente mit Umgebungsdiagrammen.

Eine Umgebung wird als Rechteck mit einem Kreis im Inneren dargestellt. Der Kreis steht für die Elternumgebung.



Namen der Umgebung stehen im Inneren des Rechtecks und sind unterstrichen. Von ihnen gehen Pfeile auf Rechtecke außerhalb des Umgebungsrechtecks aus. In diesen äußeren Rechtecken steht der Wert der entsprechenden Namensbindung.

```
env(a = 1, blub = TRUE)
```



Funktionsobjekte werden dargestellt als Rechteck mit einem Kreis und (, ) im Inneren. Argumentnamen stehen ggf durch , getrennt zwischen den Klammern.

```

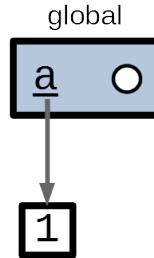
function() {
  # ...
}
function(x) {
  # ...
}
function(x, y) {
  # ...
}

```



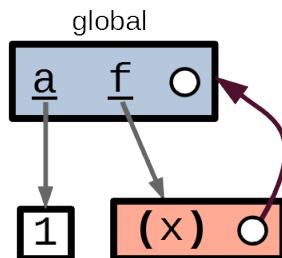
Die globale Umgebung ist mit dem Wort “global” überschrieben.

```
a <- 1 # Namensbindung in der globalen Umgebung
```



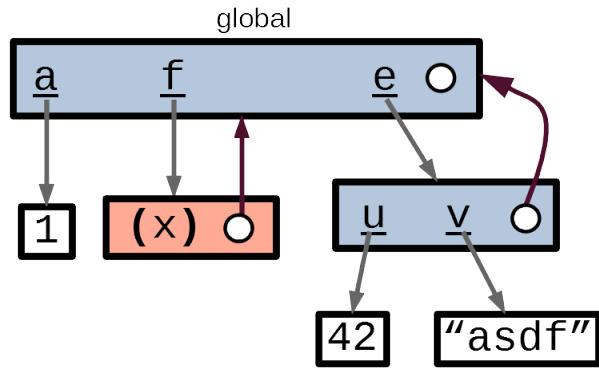
Vom Kreis einer Funktion geht ein Pfeil zu der Umgebung, in der die Funktion erstellt wurde (die sogenannte **Funktionsumgebung** dieser Funktion).

```
a <- 1
f <- function(x) {
  # ...
}
```



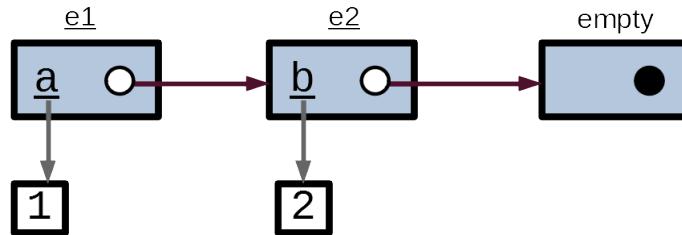
Vom Kreis einer Umgebung geht ein Pfeil zu ihrer Elternumgebung. Bei der globalen Umgebung lassen wir diesen Pfeil meist weg.

```
a <- 1
f <- function(x) {
  # ...
}
e <- env(u = 42, v = "asdf") # default parent is global env
```



Die leere Umgebung ist mit dem Wort “empty” überschrieben. Da sie keine Elternumgebung hat, ist ihr Kreis ausgefüllt. Wenn die globale Umgebung nicht wichtig ist, lassen wir sie weg und schreiben ggf über wichtige Umgebungen ihren unterstrichenen Variablennamen.

```
e1 <- env(empty_env(), b = 2)
e2 <- env(e1, a = 1)
```



## 5 Umgebungen für Funktionen

Bei der Erstellung und Ausführung von Funktionen sind drei Umgebungen wichtig:

- die **Funktionsumgebung** (*function environment*)
- die **Ausführungsumgebung** (*execution environment*)
- die **aufrufende Umgebung** (*caller environment*)

Sie steuern Scoping für Funktionen. Dh, diese Umgebungen legen fest, wie Namen von Variablen im Inneren einer Funktion aufgelöst werden.

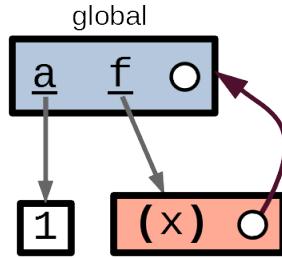
### 5.1 Funktionsumgebung

`rlang::fn_env()` gibt die **Funktionsumgebung** an, welche bei der Erstellung einer Funktion auf die zu diesem Zeitpunkt aktuelle Umgebung gesetzt wird. Sie ist Teil des Funktionsobjektes (außer bei primitiven Funktionen).

#### Beispiel 1:

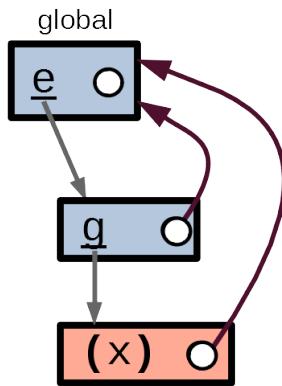
```
a <- 1
f <- function(x) x + a
env_label(current_env())
## [1] "global"
env_label(fn_env(f))
## [1] "global"
```

`f` ist ein Name in der globalen Umgebung. An diesen Name ist ein Wert gebunden. Der Wert ist ein Funktionsobjekt. Das Funktionsobjekt hat eine Funktionsumgebung, nämlich die globale Umgebung.



### Beispiel 2:

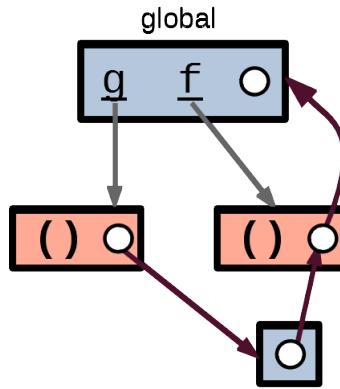
```
e <- env() # erzeuge Umgebung mit globaler Umgebung als Eltern-Umgebung
e$g <- function(x) x+2
```



### Beispiel 3:

Wird eine Funktion `g()` in einer anderen Funktion `f()` erzeugt, ist die Funktionsumgebung von `g()` gleich der sogenannten *Ausführungsumgebung* beim Aufruf von `f()`.

```
f <- function() {
  print(env_label(current_env()))
  function(){} # Rückgabewert: ein Funktionsobjekt mit leerem body()
}
g <- f() # print current env inside f
## [1] "0000000015614678"
env_label(fn_env(g)) # function env of g equals current env inside f
## [1] "0000000015614678"
```



## 5.2 Ausführungsumgebung

Wird eine Funktion aufgerufen, wird eine neue Umgebung erstellt: eine **Ausführungsumgebung** (*execution environment*). In ihr leben die Argumente der Funktion und die Namen, die im Inneren der Funktion gebunden werden.

Die Eltern-Umgebung einer Ausführungsumgebung ist die Funktionsumgebung.

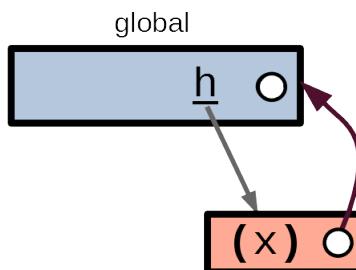
Bei jedem Funktionsaufruf wird eine neue Ausführungsumgebung erstellt.

Illustrieren wir den Vorgang des Funktionsaufrufs am Beispiel der Funktion `h()`.

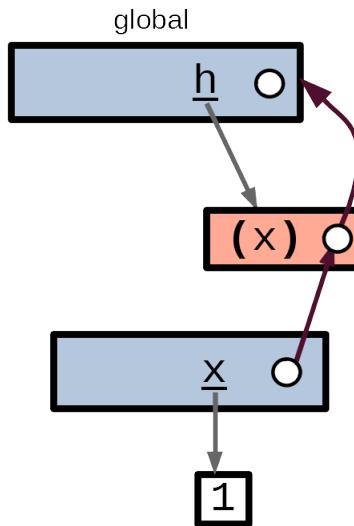
```

h <- function(x) {
  # 2.
  a <- 2
  # 3.
  x + a
}
# 1.
y <- h(1)
# 4.
  
```

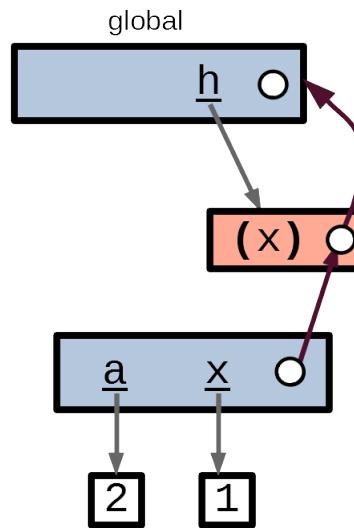
Umgebungsdiagramm bei # 1.:



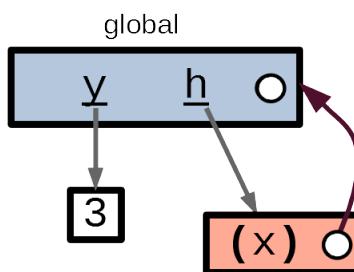
Umgebungsdiagramm bei # 2.: Die Ausführungsumgebung entsteht. In ihr ist der übergebene Wert 1 und den Namen des zugehörigen Argumentes `x` gebunden.



Umgebungsdiagramm bei # 3.:



Umgebungsdiagramm bei # 4.: Die Ausführungsumgebung verschwindet.



**Hinweis:** Wir machen hier noch eine kleine Vereinfachung bzgl wann genau die Namensbindung zwischen x und 1 entsteht, siehe letzter Abschnitt dieses Kapitels (*Lazy Evaluation 2*).

Normalerweise wird die Ausführungsumgebung gelöscht, sobald die Funktionsausführung beendet ist.

Sind jedoch andere, weiter existierende Objekte von einer Ausführungsumgebung abhängig, bleibt sie weiter bestehen.

Zum Beispiel kann die Ausführungsumgebung von der Funktion zurückgegeben und dann weiter verwendet werden.

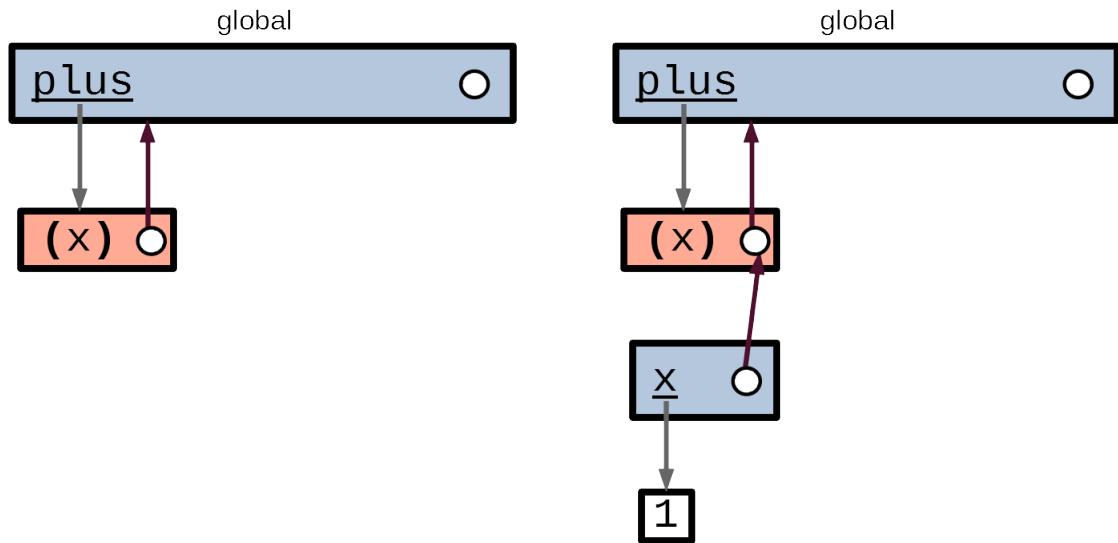
```
h <- function(x) {  
  a <- x * 2  
  current_env()  
}  
e <- h(10)  
env_print(e)  
## <environment: 000000001CA77CA0>  
## parent: <environment: global>  
## bindings:  
##   * a: <dbl>  
##   * x: <dbl>  
env_label(fn_env(h))  
## [1] "global"
```

Zu einer Funktion können daher zur selben Zeit mehrere Ausführungsumgebungen (von verschiedenen Funktionsaufrufen) existieren.

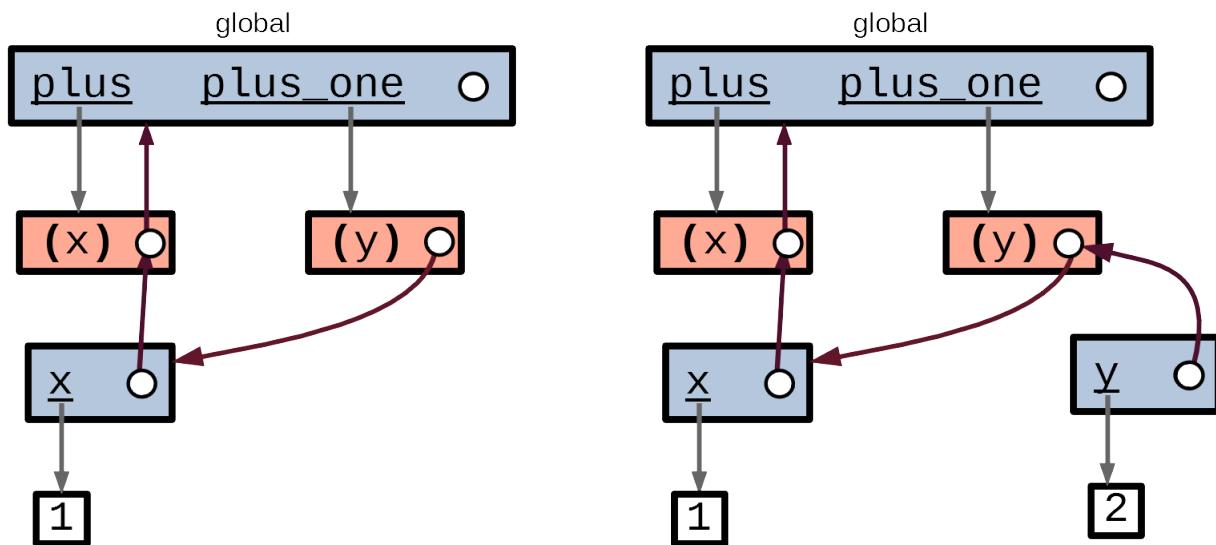
Wird eine Funktion im Innern einer anderen Funktion kreiert und zurückgegeben, wird die Ausführungsumgebung der äußeren Funktion ebenfalls nicht gelöscht, da diese als Funktionsumgebung der inneren Funktion dient.

```
plus <- function(x) {  
  # 2.  
  function(y) {  
    # 4.  
    x + y  
  }  
}  
# 1.  
plus_one <- plus(1)  
# 3.  
plus_one(2)  
## [1] 3
```

Umgebungsdiagramm bei # 1. und # 2.



Umgebungsdiagramm bei # 3. und # 4.



Bei der Berechnung von `x + y` wird `y` direkt in einer Ausführungsumgebung von `plus_one()` gefunden. `x` wird dort nicht gefunden. Also wird in der Elternumgebung gesucht. Dies ist eine Ausführungsumgebung von `plus()`. Sie enthält den Namen `x`.

**Hinweis:** `rlang::env_parent()` ist keine Ersetzungsfunktion. Das Base-R Äquivalent `parent.env()` schon.

```
library(rlang)
f <- function() x
e1 <- env()
fn_env(f) <- e1
e2 <- env(e1)
parent.env(e1) <- e2 # evil
f()
```

### 5.3 Aufrufende Umgebung

Die Umgebung, aus der heraus eine Funktion aufgerufen wird, heißt **aufrufende Umgebung** (*caller environment*). Sie wird von `caller_env()` zurückgegeben.

```
library(stringr)
f <- function() {
  cat(str_c("f: ",
    "current: ", env_label(current_env()), ", ",
    "caller: ", env_label(caller_env()), "\n"))
  g()
}
g <- function() {
  cat(str_c("g: ",
    "current: ", env_label(current_env()), ", ",
    "caller: ", env_label(caller_env()), "\n"))
  h()
}
h <- function() {
  cat(str_c("h: ",
    "current: ", env_label(current_env()), ", ",
    "caller: ", env_label(caller_env()), "\n"))
}
f()
## f: current: 00000001DB36CA0, caller: global
## g: current: 00000001DB3AED0, caller: 000000001DB36CA0
## h: current: 00000001DB3BC38, caller: 000000001DB3AED0
```

Bei einem Funktionsaufruf werden Argumente in der aufrufenden Umgebung ausgewertet (nicht in Ausführungs- oder Funktionsumgebung).

```
a <- 1 # a in aufrufender Umgebung
f <- function(x) {
  a <- 2 # a in Ausführungsumgebung
  x + 10
}
e <- env(a = 3) # a in Funktionsumgebung
fn_env(f) <- e
f(a)
## [1] 11
```

## 6 Scoping-Prinzipien

Die Auflösung von Namen in R kann durch drei Prinzipien beschrieben werden: *Name-Masking*, *Neuanfang*, *dynamic Lookup*.

### 6.1 Name-Masking

Zum Auflösen eines Namens wird dieser zuerst in der aktuellen Umgebung gesucht. Wird der Name dort nicht gefunden, werden nacheinander die Vorfahren (Elternumgebung, Elternumgebung der Elternumgebung, ...) der aktuellen Umgebung durchsucht.

Ein Name im Inneren einer Funktion wird zuerst in der Ausführungsumgebung, dann in der Funktionsumgebung (bei Erzeugung der Funktion aktuelle Umgebung) gesucht. Dadurch werden ggf Namen in den Vorfahren überdeckt (**Name-Masking**).

```
# Beispiel 1
x <- 10
y <- 20
f1 <- function() {
  x <- 1
  c(x, y)
}
f1()
## [1] 1 20
```

Funktionen sind normale Objekte. Dementsprechend gilt *Name-Masking* prinzipiell auch für Funktions-Variablen.

```
f <- function(x) x + 1
g <- function() {
  f <- function(x) x + 100
  f(10)
}
g()
## [1] 110
```

Bei einem Funktionsaufruf wird jedoch explizit nach einem Funktionsobjekt gesucht. Nicht-Funktions-Objekte überdecken keine Funktionsobjekte.

```
f <- function(x) x + 1
g <- function() {
  f <- 100
  f(10)
}
g()
## [1] 11

c <- 3
c(c, 1)
## [1] 3 1
```

## 6.2 Dynamic Lookup

Die Auflösung der Namen geschieht erst dann, wenn die Werte wirklich benötigt werden. D.h. die beim Funktionsaufruf aktuellen Werte (nicht bei der Funktionserzeugung) werden benutzt (**dynamic Lookup**).

```
# Beispiel 3
x <- 10
f3 <- function() x + 1
f3()
## [1] 11
x <- 100
f3()
## [1] 101
```

Mit `codetools::findGlobals()` können die externen Abhängigkeiten einer Funktion gefunden werden.

```
f <- function() x + 1
codetools::findGlobals(f)
## [1] "+" "x"
```

Um jegliche Abhängigkeit zu unterbinden, könnte man die Funktionsumgebung auf die leere Umgebung

setzen. Dies ist aber in der Regel nicht sinnvoll.

```
fn_env(f) <- empty_env()
f() # ERROR
## Error in x + 1: could not find function "+"
```

Dynamic Lookup ermöglicht Rekursion.

```
infinite_recursion <- function() infinite_recursion()
infinite_recursion() # evil
```

### 6.3 Neuanfang

Die Ausführungsumgebung wird bei jedem Aufruf neu erzeugt. Alte Werte werden nicht übertragen (**Neuanfang**).

```
# Beispiel 2
f2 <- function() {
  if (!env_has(current_env(), "a")) {
    a <- 1
  } else {
    a <- a + 1
  }
  a
}
f2()
## [1] 1
f2()
## [1] 1
```

Achtung: Namen werden nicht in der aufrufenden Umgebung gesucht (außer sie ist ein Vorfahre der Ausführungsumgebung). Jedoch entstammen die Werte der Argumente einer Funktion aus der aufrufenden Umgebung.

```
x <- 4
y <- 4
z <- 4
create_function <- function() {
  y <- 2 # Ausführungsumgebung von create_function() und Funktionsumgebung von
  function() {
    x <- 1 # Ausführungsumgebung von f()
    c(x=x, y=y, z=z)
  }
}
call_function <- function(fun) {
  x <- 3
  y <- 3
  z <- 3
  print(fun())
}
f <- create_function()
call_function(f)
## x y z
## 1 2 4
z <- 5
call_function(f)
```

```
## x y z
## 1 2 5
```

## 7 Zuweisungsoperatoren

Mittels `?`<--`` erfahren wir, dass es insgesamt 5 Zuweisungsoperatoren gibt: `<-`, `<<-`, `->`, `->>`, `=`.

Bei der Zuweisung `x <- value`, `value -> x`, `x = value` wird eine Namensbindung zwischen dem Namen `x` und dem Wert `value` in der aktuellen Umgebung erzeugt.

```
# äquivalent sind:
x <- 42
x = 42
42 -> x
env_bind(current_env(), x = 42)
```

Die Operatoren sind Funktionen mit dem Rückgabewert `value`. `<-` und `=` werden wie `^` und im Gegensatz zu den meisten anderen Operatoren von rechts nach links gruppiert (Ausführungsreihenfolge), weswegen `x <- y <- value` sowohl `x` als auch `y` den Wert `value` zuweist.

`=` ist größtenteils synonym zu `<-`. Allerdings hat `=` noch eine zusätzlich Bedeutung bei der Benennung von Funktionsargumenten. Beachte den Unterschied zwischen `f(y <- 1)` und `f(y = 1)`.

```
f <- function(x=0, y=0) print(c(x,y))
y <- 1
f(y = 2)
## [1] 0 2
y
## [1] 1
f(y <- 2)
## [1] 2 0
y
## [1] 2
```

Achtung: Was bedeutet `x<-2`? `x <- 2` oder `x < -2`? Guter Stil: Setze immer Leerzeichen um Zuweisungsoperatoren. In RStudio werden bei der Nutzung von `Alt + -` Leerzeichen automatisch gesetzt.

Der Zuweisungsoperator `<-` erzeugt (oder ändert) eine Namensbindung in der aktuellen Umgebung `current_env()`, wohingegen `x <<- 5` (oder `5 ->> x`) nach dem Namen `x` in der Eltern-Umgebung der aktuellen Umgebung sucht. Wird er dort nicht gefunden, werden die weiteren Vorfahren durchsucht.

Wird der Name `x` in einer der Vorfahren-Umgebungen gefunden, wird (in der zuerst gefunden Namensbindung) der Wert entsprechend geändert (hier: auf 5 gesetzt).

Wird der Name nicht gefunden, wird eine neue Namensbindung in der globalen Umgebung angelegt.

```
cnt1 <- 0
count1 <- function() {
  cnt1 <- cnt1 + 1
}
count1()
count1()
cnt1
## [1] 0
cnt2 <- 0
count2 <- function() {
  cnt2 <<- cnt2 + 1
}
```

```
count2()
count2()
cnt2
## [1] 2
```

## 8 Rekursion über Umgebungen

Wird ein Variablenname in einer Umgebung nicht gefunden, wird in deren Eltern-Umgebung gesucht, wobei jede Umgebung außer der leere Umgebung (`rlang::empty_env()`) genau eine Eltern-Umgebung hat. Ausgestattet mit diesem Wissen, können wir selbst Funktionen schreiben, die uns die Umgebung ausgeben, in der ein gegebener Name gebunden ist.

```
where <- function(name, env = caller_env()) {
  if (identical(env, empty_env())) {
    stop("Can't find ", name, call. = FALSE)
  } else if (env_has(env, name)) {
    env
  } else {
    where(name, env_parent(env))
  }
}
x <- 2
y <- 3
env_label(where("x"))
## [1] "global"
f <- function() {
  x <- 1
  cat("x:", env_label(where("x")), "\n")
  cat("y:", env_label(where("y")), "\n")
}
f()
## x: 000000001CA5D840
## y: global
env_label(where("sd"))
## [1] "package:stats"
```

Die Folge von Umgebungen, in denen ein Name gesucht werden würde, heißt **Suchpfad** (*search path*).

```
searchpath_labels <- function(env = caller_env()) {
  lbl <- env_label(env)
  if (identical(env, empty_env())) {
    return(lbl)
  } else {
    return(c(lbl, searchpath_labels(env_parent(env))))
  }
}
searchpath_labels()
## [1] "global"          "package:stringr"  "package:rlang"
## [4] "package:stats"    "package:graphics" "package:grDevices"
## [7] "package:utils"    "package:datasets" "package:methods"
## [10] "Autoloads"        "package:base"     "empty"
f <- function() searchpath_labels()
f()
## [1] "000000001D3E6F28"  "global"           "package:stringr"
```

```
## [4] "package:rlang"      "package:stats"      "package:graphics"
## [7] "package:grDevices"  "package:utils"      "package:datasets"
## [10] "package:methods"    "Autoloads"        "package:base"
## [13] "empty"
```

Der Suchpfad, ausgehend von der globalen Umgebung, enthält Umgebungen einiger Pakete und endet mit der leeren Umgebung.

Da es nur eine leere Umgebung gibt und diese die einzige ohne Eltern-Umgebung ist, gilt: Alle existierenden Umgebungen bilden mit der Verbindung durch die “Eltern-Relation” einen Baum mit Wurzel `empty_env()`.

## 9 Umgebungen für Pakete

### 9.1 Suchpfad

Wird ein Paket mit `library()` geladen, wird es zu einer Vorfahren-Umgebung der globalen Umgebung. Die direkte Eltern-Umgebung der globalen Umgebung ist das zuletzt geladene Paket. Dessen Eltern-Umgebung ist wiederum das zuvor geladene Paket.

```
library(magrittr) # to use pipe %>%
##
## Attaching package: 'magrittr'
## The following object is masked from 'package:rlang':
##
##     set_names
global_env() %>% env_label()
## [1] "global"
global_env() %>% env_parent() %>% env_label()
## [1] "package:magrittr"
global_env() %>% env_parent() %>% env_parent() %>% env_label()
## [1] "package:stringr"
```

Geht man weiter zurück, findet man alle Pakete in der Reihenfolge, in der sie geladen wurden. Dies entspricht dem Suchpfad. Alle Objekte, die von der Konsole aus aufgerufen werden können, sind in einem der Umgebungen des Suchpfades zu finden.

```
searchpath_labels()
## [1] "global"          "package:magrittr"  "package:stringr"
## [4] "package:rlang"    "package:stats"    "package:graphics"
## [7] "package:grDevices" "package:utils"    "package:datasets"
## [10] "package:methods"   "Autoloads"      "package:base"
## [13] "empty"
```

Die letzte Umgebung vor der leeren ist immer `package:base`, die auch mittels `rlang::base_env()` abrufbar ist. Sie enthält elementare Bausteine von R. Die Eltern-Umgebung von `package:base` ist die leere Umgebung.

```
head(env_names(base_env()), 20)
## [1] "!"          "$"          "€"
## [4] "("          "*"          "+"
## [7] "-"          "/"          ":"
## [10] "<"         "="          ">"
## [13] "as.matrix.data.frame" "@"          "F"
## [16] "I"          "registerS3methods" "as.POSIXct.Date"
## [19] "T"          "["
```

Die Sammlung von Pakete, die beim Start von R automatisch geladen werden, nennen wir **Base-R**: `stats`,

```
graphics, grDevices, utils, datasets, methods, base.
```

Autoloading taucht ebenfalls im Suchpfad auf, erlaubt On-Demand-Laden von Paketen und kann ignoriert werden.

## 9.2 Namespace

Falls Funktionen aus einem Paket Funktionen eines anderen Paketes aufrufen, scheint unsere Beschreibung des Suchpfades zu suggerieren, dass die richtige Auflösung der Funktionsnamen von der Ladereihenfolge der Pakete abhängt. Namespaces sorgen dafür, dass diese Funktionen immer richtig funktionieren.

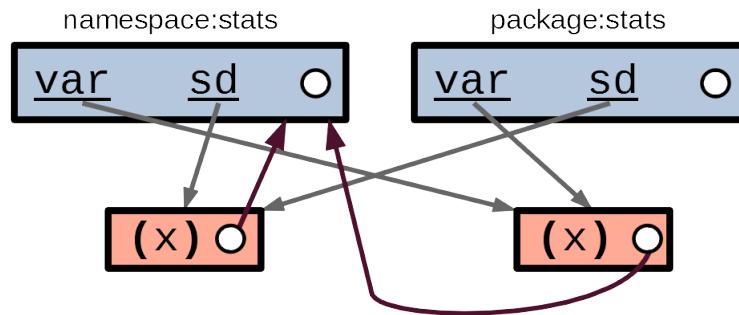
Die Funktion `sd()` (aus dem Paket `stats`) ist mittels `var()` definiert.

```
sd
## function (x, na.rm = FALSE)
## sqrt(var(if (is.vector(x) || is.factor(x)) x else as.double(x),
##       na.rm = na.rm))
## <bytecode: 0x000000015539478>
## <environment: namespace:stats>
```

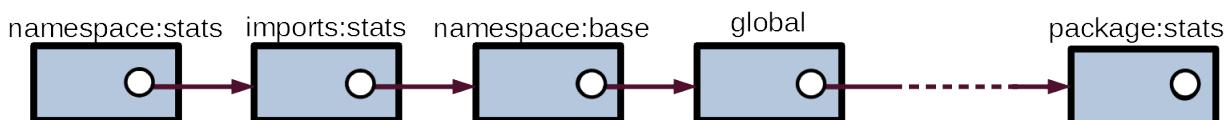
Angenommen wir selbst haben eine Funktion `var` definiert. Wie findet `sd()` die richtige `var()`-Funktion?

```
var <- function(x) 0
sd(1:5) # nutzt offensichtlich nicht unsere Definition von var()
## [1] 1.581139
```

Jede Funktion eines Paketes ist sowohl in einer Namespace- als auch in einer Paket-Umgebung gebunden. Die Funktionsumgebung der Funktion ist gleich der Namespace-Umgebung.

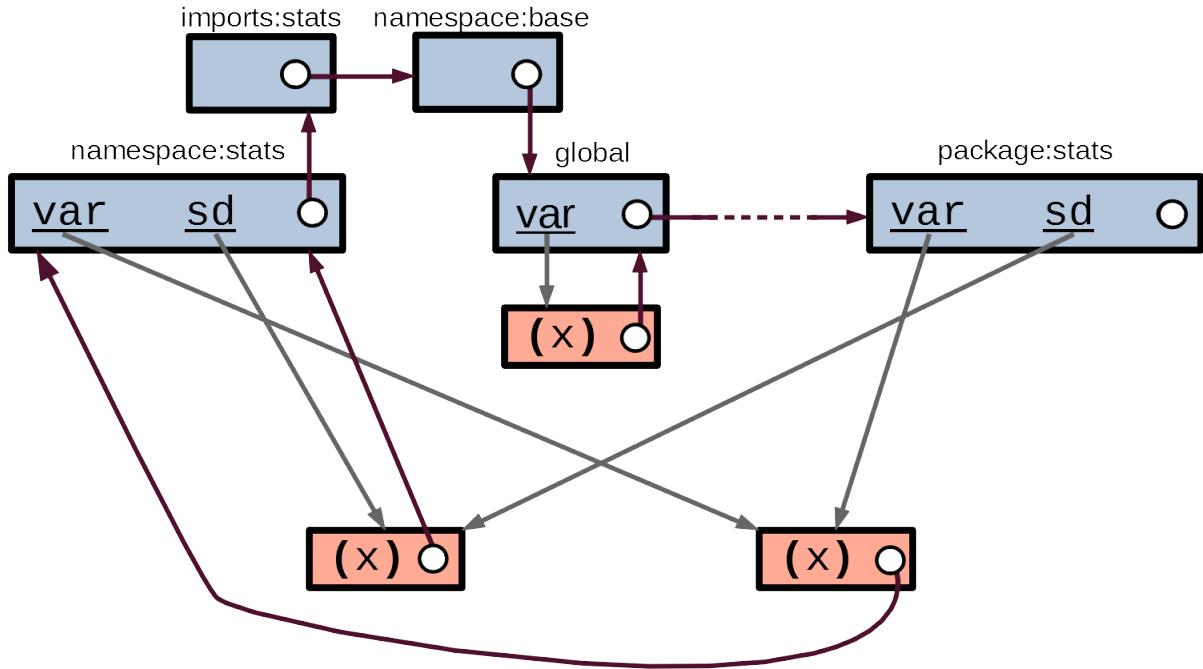


Die Namespace-Umgebung ist (über zwei andere Umgebungen) Nachkomme der globalen Umgebung. Die Paket-Umgebung ist Vorfahre der globalen Umgebung.



```
searchpath_labels(fn_env(sd))
## [1] "namespace:stats"    "imports:stats"      "namespace:base"
## [4] "global"              "package:magrittr"    "package:stringr"
## [7] "package:rlang"        "package:stats"       "package:graphics"
## [10] "package:grDevices"    "package:utils"       "package:datasets"
## [13] "package:methods"      "Autoloads"          "package:base"
## [16] "empty"
```

Zusammen erhalten wir folgendes Bild.



Die Namensbindung unserer selbst definierten Funktion `var()` besteht in der globalen Umgebung. In `package:stats` sind die Namen `sd` und `var`, die mit den entsprechenden Funktionen des Paketes `stats` verbunden sind. Wird `sd()` aufgerufen, wird der Name `sd` zuerst in der globalen Umgebung gesucht, nicht gefunden und dann in `package:stats` gesucht und gefunden. Die Funktion wird ausgeführt. Dabei muss der Name `var` aufgelöst werden. Dieser wird (nach einer Ausführungsumgebung) zuerst in der Funktionsumgebung `fn_env(sd)` gesucht. Dies ist `namespace:stats`, wo der Name `var` gefunden wird. Dieser ist mit der entsprechenden Funktion aus dem Paket `stats` verbunden.

### 9.3 Locked Environments

Namensbindungen in Paket-Umgebungen sind *locked*, dh die Namensbindungen in diesen Umgebungen können nicht ohne Weiteres von außen geändert werden.

```
env_bind(base_env(), "any"=all)
## Error in env_bind(base_env(), any = all): cannot change value of locked binding for 'any'
```

Siehe auch `?lockBinding`.

```
unlockBinding("any", base_env())
base_any <- any
env_bind(base_env(), "any"=all) # evil
any(T, T, F)
## [1] FALSE
env_bind(base_env(), "any"=base_any)
lockBinding("any", base_env())
any(T, T, F)
## [1] TRUE
```

## 10 Übersicht Umgebungen

Es gibt spezielle Umgebungsobjekte, die immer existieren: die globale Umgebung (`rlang::global_env()`), die leere Umgebung (`rlang::empty_env()`), die Umgebung von Base-R (`rlang::base_env()`).

Die aktuelle Umgebung `rlang::current_env()` hängt vom Zustand des Programms ab.

Jede Umgebung, außer der leeren Umgebung, hat eine Eltern-Umgebung (`rlang::parent_env()`). Die Eltern-Umgebung zusammen mit deren Eltern usw werden als Vorfahren bezeichnet. Andersherum können die Begriffe Kind- und Nachkommen-Umgebung genutzt werden.

Teil einer jeden Funktion vom Typ Closure ist die Funktionsumgebung (`rlang::fn_env()`). Eine Funktion wird aus der aufrufenden Umgebung `rlang::caller_env()` aufgerufen. Der Funktionscode wird in der Ausführungsumgebung aufgerufen. Die Eltern-Umgebung der Ausführungsumgebung ist die Funktionsumgebung.

Für ein Paket mit dem Namen `packnm` werden folgende Umgebungen angelegt: `namespace:pckgnm`, `imports:pckgnm`, `namespace:base` als Nachkommen der globalen Umgebung und `package:pckgnm` als Vorfahre der globalen Umgebung.

## 11 Lazy Evaluation 2

Argumente einer Funktion werden erst dann aufgelöst, wenn sie gebraucht werden. Zuvor sind sie nur **Versprechen** (dass der Name aufgelöst werden kann, engl *Promises*). Dies ermöglicht die sogenannte *lazy Evaluation*.

```
f1 <- function(x) {  
  10  
}  
f1(stop("This is an error!"))  
## [1] 10
```

Um die Auswertung von Argumenten zu erzwingen kann `force()` genutzt werden.

```
f2 <- function(x) {  
  force(x)  
  10  
}  
f2(stop("This is an error!"))  
## Error in force(x): This is an error!
```

Versprechen sind Verknüpfungen zwischen Namen der Argumente einer Funktion und Ausdrücken in der aufrufenden Umgebung. Sie sind also nicht direkt mit Werten verbunden.

Wird der Name eines Versprechens aufgelöst, wird das Versprechen eingelöst und eine Namensbindung zwischen dem Namen und dem Wert des Ausdrucks erstellt. Danach existiert das Versprechen nicht mehr, nur noch die "normale" Namensbindung zwischen Name und Wert.

Im Inneren von `f1` ist `x` ein Versprechen, das nie eingelöst wird, weswegen der Ausdruck `stop("This is an error!")` nie ausgewertet wird.

In `f2` wird durch `force(x)` die Einlösung des Versprechens erzwungen, wodurch der Ausdruck `stop("This is an error!")` ausgewertet wird und in dem Fehler resultiert.

Statt `force(x)` könnte man auch einfach `x` schreiben. `force()` verdeutlicht jedoch, dass man einen bestimmten Effekt erwartet.

```
force  
## function (x)  
## x
```

```
## <bytecode: 0x00000000146688e8>
## <environment: namespace:base>
```

Das folgende Beispiel der Funktion `capture()` zeigt, dass Werte zwischen dem Erstellen des Versprechens und dem Einlösen noch geändert werden können.

```
# gib eine Funktion zurück, die immer den Wert von x ausgibt
capture1 <- function(x) {
  function() print(x)
}
y <- 10
g1 <- capture1(y)
g2 <- capture1(y)
g1()
## [1] 10
y <- 20
g2()
## [1] 20
y <- 30
g1()
## [1] 10
g2()
## [1] 20
```

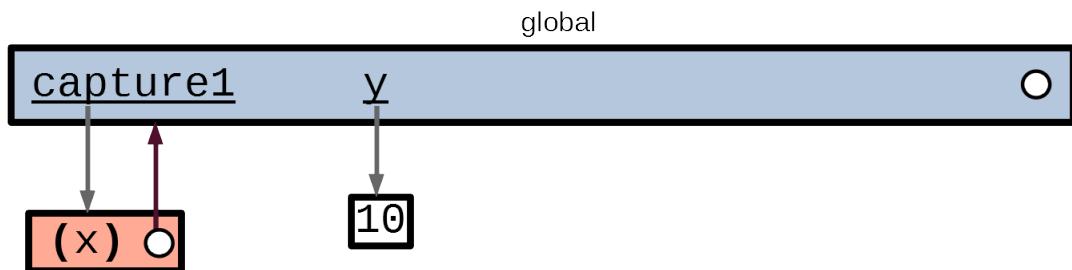
Wir gehen den Code Schritt für Schritt durch und zeichnen Umgebungsdiagramme.

Wir oben angemerkt, haben wir Umgebungsdiagramme bisher in soweit vereinfacht, dass wir Versprechen immer sofort aufgelöst haben. Jetzt machen wir sie sichtbar.

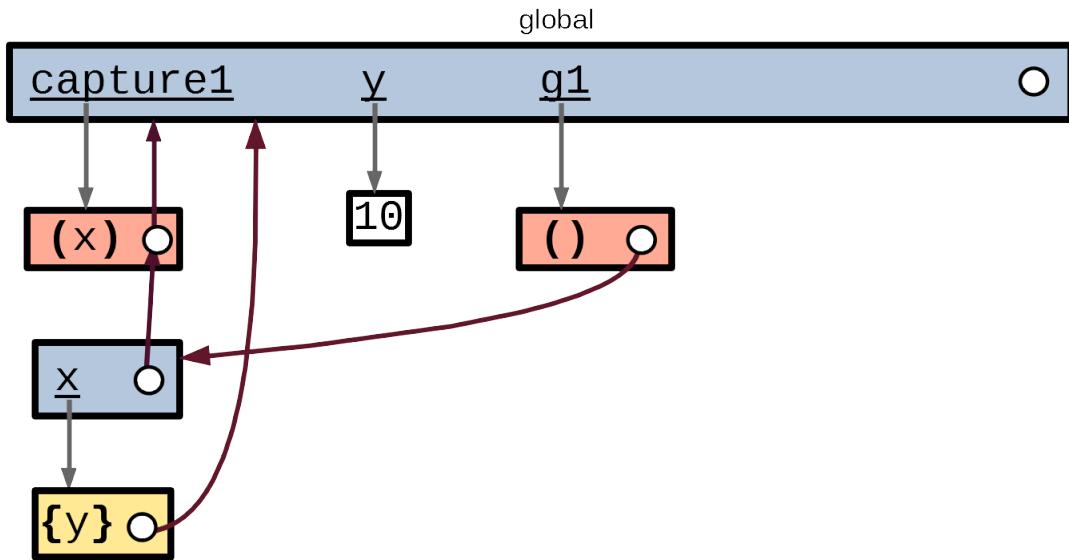
Ein Versprechen wird durch ein Rechteck mit einem Kreis und `{, }` im Inneren dargestellt. Der Ausdruck, der beim Auflösen evaluiert werden soll, steht zwischen `{` und `}`. Vom Kreis geht ein Pfeil zu der Umgebung, in der dieser Ausdruck ausgewertet wird (die aufrufende Umgebung).

```
capture1 <- function(x) {
  function() print(x)
}

y <- 10
# Namensbindung y --> 10 in der globalen Umgebung
# 1.
```

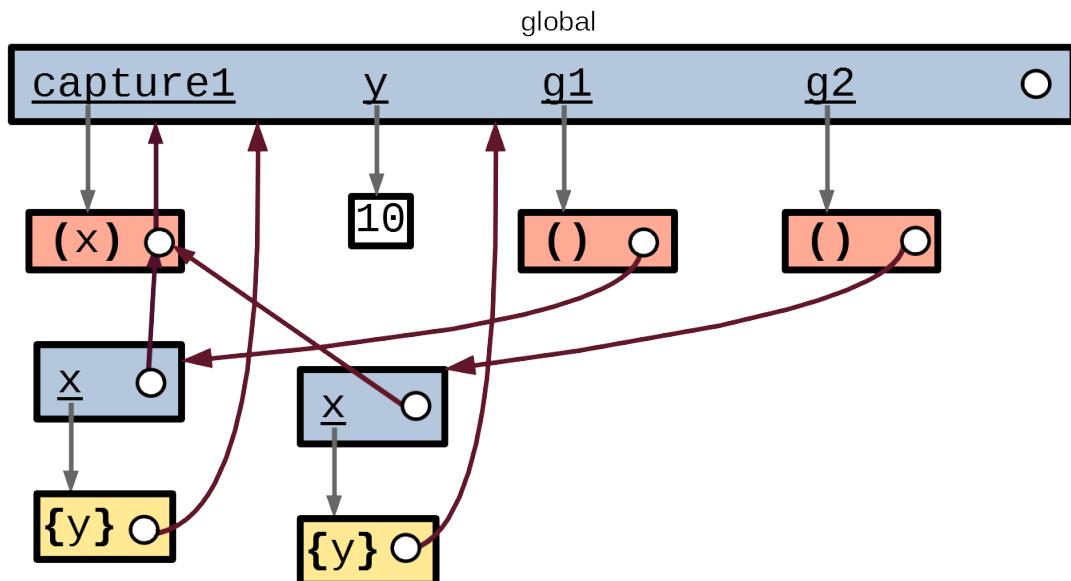


```
g1 <- capture1(y)
# In der Ausführungsumgebung [envA1] von capture wird ein Versprechen zwischen
# dem Namen x und dem Ausdruck y (mit Auswertung in der aufrufenden Umgebung
# [global]) erstellt.
# 2.
```



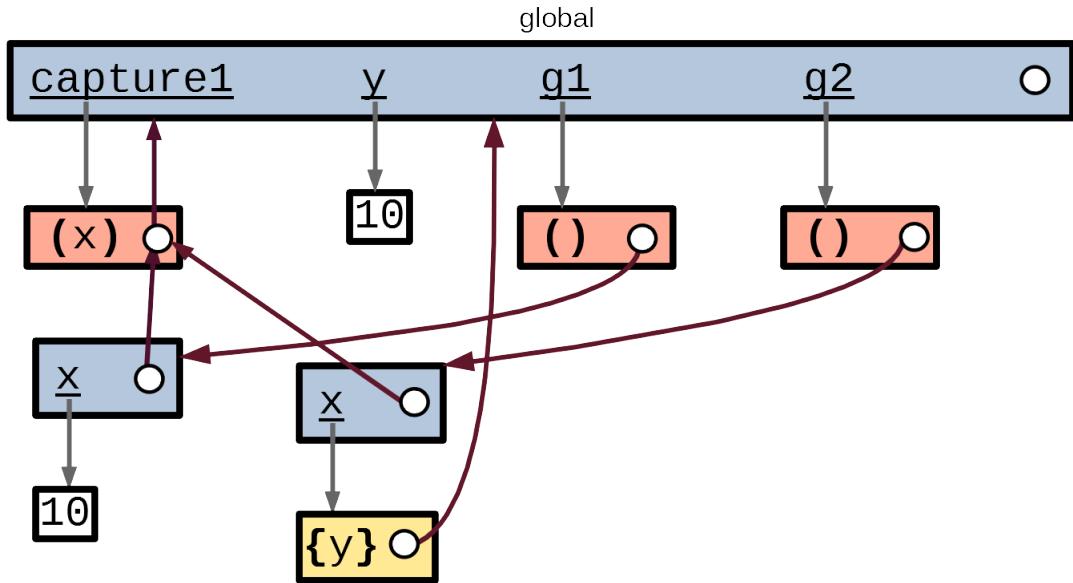
```

g2 <- capture1(y)
# Gleichtes wie oben, aber
# neue Ausführungsumgebung [envA2] (Neuanfang)
# 3.
  
```

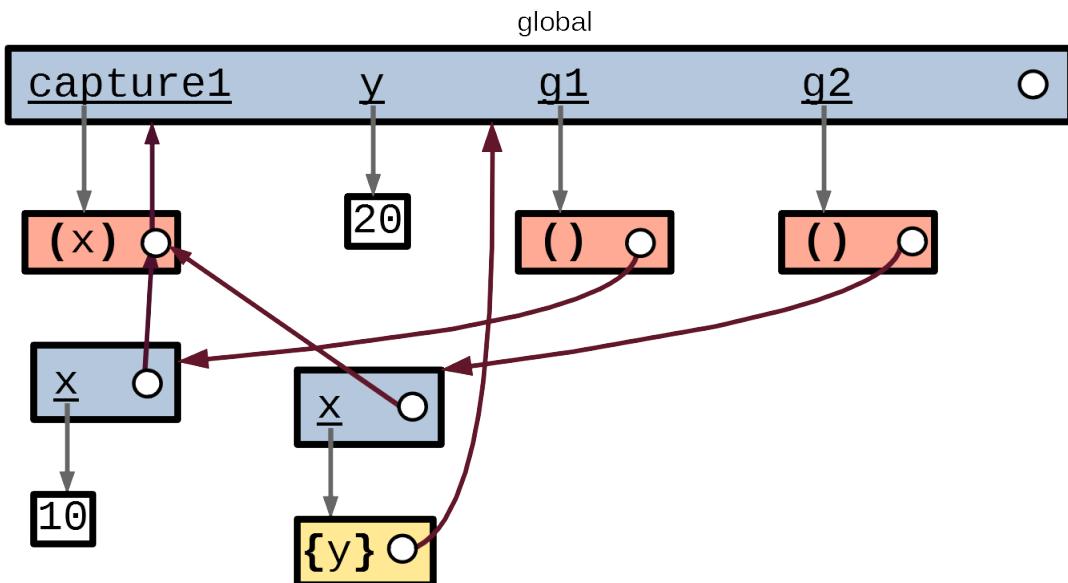


```

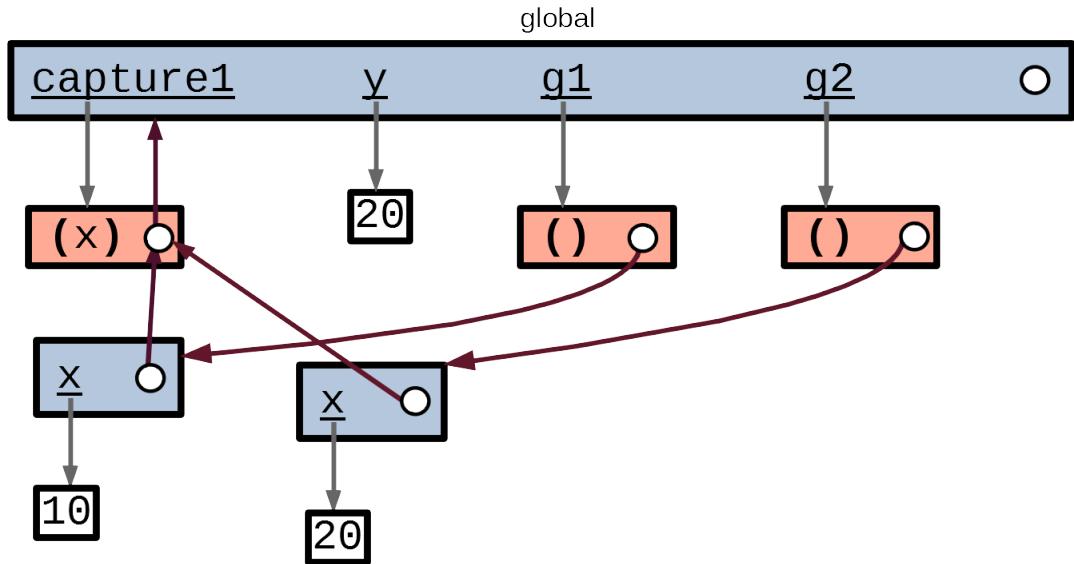
g1()
## [1] 10
# x muss Auflöst werden. Dazu wird
# das Versprechen in [envA1] eingelöst, y zu 10 aufgelöst, und
# in [envA1] die Namensbindung x --> 10 erstellt.
# 4.
  
```



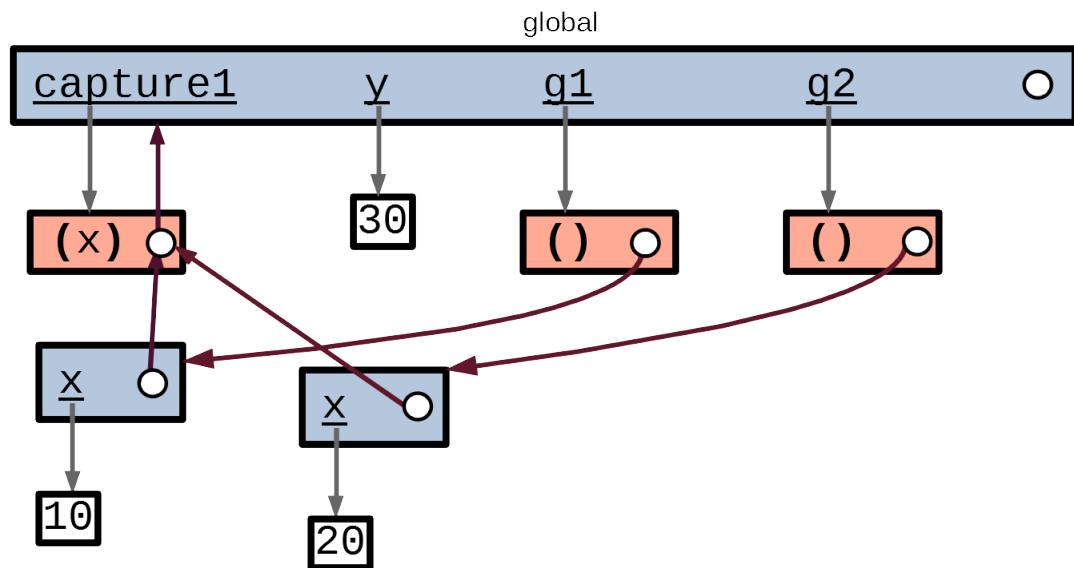
```
y <- 20
# Namensbindung y --> 20 in der globalen Umgebung
# 5.
```



```
g2()
## [1] 20
# x muss aufgelöst werden. Dazu wird
# das Versprechen in [envA2] eingelöst, y zu 20 aufgelöst, und
# in [envA2] die Namensbindung x --> 20 erstellt.
# 6.
```



```
y <- 30
# Namensbindung y --> 30 in der globalen Umgebung
# 7.
```



```
g1()
## [1] 10
# x muss aufgelöst werden.
# Es wird die Namensbindung x --> 10 in [envA1] gefunden.

g2()
## [1] 20
# x muss aufgelöst werden.
# Es wird die Namensbindung x --> 20 in [envA2] gefunden.
```

Wir möchten jedoch den Wert von y beim Aufruf der Funktion `capture()` aufnehmen (nicht beim Aufruf der

Funktion `g1()` bzw `g2()`). Lösung: Erzwinge Auflösung mittels `force()`.

```
capture2 <- function(x) {
  force(x)
  function() print(x)
}
y <- 10

g3 <- capture2(y)
# durch force() wird hier schon das Versprechen
# von x eingelöst und die Namensbindung x --> 10 erstellt.

y <- 20

g3()
## [1] 10
```

Mit der Funktion `rlang::env_bind_lazy()` können wir selbst Versprechen geben und somit *lazy evaluation* nicht nur für Funktionsargumente nutzen.

```
x <- 1
y <- 2
env_bind_lazy(current_env(), v = c(x, y)) # Versprechen des Namens v zu dem Ausdruck c(x, y)
x <- 5
v
## [1] 5 2
```

Bemerkung: Intern gibt es einen eigenständigen Datentyp für Versprechen namens `promise`. Jedoch kann die Funktion `typeof()` diesen nie ausgeben, da bei `typeof(x)` ein etwaiges Versprechen `x` eingelöst werden würde, bevor dessen Typ bestimmt werden kann.

```
env_bind_lazy(current_env(), x = 1)
typeof(x)
## [1] "double"
```

# V10 - Funktionale Programmierung

17. Mai 2021

## Contents

<b>1</b>	<b>Funktionale Programmierung</b>	<b>1</b>
1.1	Reine Funktionen . . . . .	3
1.2	Funktionen höherer Ordnung . . . . .	4
<b>2</b>	<b>Funktionale</b>	<b>5</b>
2.1	apply und Co . . . . .	5
2.2	Mathematische Funktionale . . . . .	13
<b>3</b>	<b>Funktionsfabrik</b>	<b>15</b>
3.1	Counter . . . . .	17
3.2	Maximum Likelihood . . . . .	18
3.3	Funktionsinterpolation . . . . .	21
<b>4</b>	<b>Funktionsoperatoren</b>	<b>22</b>
4.1	Verhaltensändernde Funktionsoperatoren . . . . .	22
4.2	Ausgabe-Funktionsoperatoren . . . . .	23
4.3	Eingabe-Funktionsoperatoren . . . . .	24

## 1 Funktionale Programmierung

Funktionale Programmierung ist eines von vielen **Programmierparadigmen**.

Abstrakte Lösungen von Problemen können auf vielfältige Art und Weise in einer Programmiersprache umgesetzt werden. Ein Programmierparadigma gibt einen fundamentalen Programmierstil an, der bestimmte Umsetzungen bevorzugt.

Die funktionale Programmierung ist durch folgende Eigenschaften gekennzeichnet, die größtenteils in R enthalten / umsetzbar sind:

### 1. Funktionen erster Klasse

Funktionen werden behandelt wie andere Objekte. Sie sind Funktionen erster Klasse oder *first class functions*. Insbesondere können Funktionen Argumente und Rückgabewerte von Funktionen sein.

```
sapply(mtcars, mean) # Funktion mean() als Argument
##      mpg      cyl      disp       hp      drat       wt      qsec
##  20.090625  6.187500 230.721875 146.687500  3.596563  3.217250 17.848750
##      vs       am      gear      carb
##  0.437500  0.406250  3.687500  2.812500

create_fun <- function() {
  f <- function() print("Ich bin eine Funktion")
  return(f) # Funktion als Rückgabewert
}
x <- create_fun()
```

```

x()
## [1] "Ich bin eine Funktion"

str(list(mean, sd, var)) # Funktionen als Elemente einer Liste
## List of 3
## $ :function (x, ...)
## $ :function (x, na.rm = FALSE)
## $ :function (x, y = NULL, na.rm = FALSE, use)

```

## 2. Anonyme Funktionen

Funktionen können ohne Namen existieren und aufgerufen werden. Solche Funktionen heißen anonym.

```

(function(x) x+2)(40)
## [1] 42

```

## 3. Funktionskomposition

Funktionen werden nicht als Abfolge von Anweisungen dargestellt, sondern als ineinander verschachtelte Funktionsaufrufe. Im Tidyverse wird mit dem Pipe-Operator ein Mittelweg bestritten.

```

library(tidyverse)
## -- Attaching packages ----- tidyverse 1.3.1 --
## v ggplot2 3.3.3     v purrrr   0.3.4
## v tibble  3.1.0     v dplyr    1.0.5
## v tidyverse 1.1.3    v stringr   1.4.0
## v readr    1.4.0     v forcats  0.5.1
## -- Conflicts ----- tidyverse_conflicts() --
## x dplyr::filter() masks stats::filter()
## x dplyr::lag()   masks stats::lag()
library(nycflights13)
flights %>%
  mutate(rank_arr_delay = rank(desc(arr_delay))) %>%
  filter(rank_arr_delay <= 3)
## # A tibble: 3 x 20
##   year month   day dep_time sched_dep_time dep_delay arr_time sched_arr_time
##   <int> <int> <int> <int>           <int> <dbl> <int>           <int>
## 1 2013     1     9     641          900 1301 1242          1530
## 2 2013     1    10    1121         1635 1126 1239          1810
## 3 2013     6    15    1432         1935 1137 1607          2120
## # ... with 12 more variables: arr_delay <dbl>, carrier <chr>, flight <int>,
## #   tailnum <chr>, origin <chr>, dest <chr>, air_time <dbl>, distance <dbl>,
## #   hour <dbl>, minute <dbl>, time_hour <dttm>, rank_arr_delay <dbl>

```

## 4. Funktionsumgebung

Funktionen können Bezug auf Variablen in der Umgebung ihrer Erstellung nehmen.

```

create_printer <- function(x) {
  function() print(x)
}
hi_printer <- create_printer("hi")
hi_printer()
## [1] "hi"

```

## 1.1 Reine Funktionen

Eine Funktion heißt **rein** (*pure*), wenn folgende Bedingungen erfüllt sind.

1. Bei gleicher Eingabe (Argumente) erhalten wir die gleiche Ausgabe.
2. Sie hat keine Nebenwirkungen (*side effects*).

Nebenwirkungen sind zB Ändern von globalen Variablen, Ausgabe auf Konsole, Datei erstellen.

Ein Merkmal funktionaler Programmierung ist, dass reine Funktionen gegenüber unreinen stark bevorzugt werden.

ZB ist `sin()` eine reine Funktion.

Der Begriff der reinen Funktion ist streng genommen für R wenig sinnvoll, da selbst `function() 1+2` vom `+`-Operator abhängig ist.

```
f <- function() 1+2
f()
## [1] 3
`+` <- `-
f()
## [1] -1
rm(`+`) # undo
```

Wir nennen eine Funktion **reinlich**, falls sie keine Nebenwirkungen außer Warnungen oder Fehlermeldungen) hat und bei gleichen übergebenen Argumenten die gleiche Ausgabe hat, sofern die Funktion nicht durch Name-Masking beeinflusst wird.

Hinweis: **reinlich** ist in der Literatur kein gängiger Begriff.

```
function(x) x+1 # reinlich

x <- 1
function() x+2 # nicht reinlich: abhängig von x in globaler Umgebung

function() x <- 3 # nicht reinlich: ändert x in globaler Umgebung

x <- 42 # nicht reinlich: ändert x in globaler Umgebung

h <- function(a) {
  x <- a+7
  x
}
h(8) # reinlich, obwohl im Inneren nicht reinliche Funktion benutzt wird

print(x) # nicht reinlich: Nebenwirkung: Ausgabe auf Konsole

plot(x) # nicht reinlich: Nebenwirkung: Modifikation des Graphics Devices

ggplot(tb, aes(x, y)) + geom_point() # reinlich: erzeugt Plot-Objekt, das zurückgegeben wird, print() v

log(-1) # reinlich, trotz Ausgabe von Warnung auf Konsole

f <- function() 0
call_f <- function() f()
# in der globalen Umgebung: nicht reinlich
# als Code für ein R-Paket: reinlich
```

Ein Vorteil reiner (oder reinlicher) Funktionen ist, dass sie leichter auf Korrektheit zu testen sind, da ihr Ergebnis nur von der Eingabe abhängt.

Außerdem ist die Funktionsweise beim Lesen ihres Quellcodes potentiell leichter zu verstehen, da alle relevanten Elemente direkt angegeben sind.

## 1.2 Funktionen höherer Ordnung

Funktionen, die Funktionen als Argumente oder als Rückgabewert haben, heißen **Funktionen höherer Ordnung**. Alle anderen Funktionen sind **Funktionen erster Ordnung**.

Funktionen höherer Ordnung lassen sich unterteilen in **Funktionale**, **Funktionsfabriken** und **Funktionsoperatoren**.

<i>In</i>	<i>Out</i>	Vector	Function
Vector		Regular function	Function factory
Function		Functional	Function operator

Funktionale haben mindestens eine Argument der Klasse `function` und geben einen Vektor-Typen (Liste, atomarer Vektor, Matrix, Array, Data-Frame, Tibble, ...) aus.

```
randomize <- function(f) f(runif(1e3)) # Funktional
randomize(mean)
## [1] 0.5060361
randomize(mean)
## [1] 0.5007658
randomize(sum)
## [1] 488.9722
```

Funktionsfabriken bilden Vektor-Typen auf Funktionen ab.

```
add_x <- function(x) { # Funktionsfabrik
  function(a) a+x
}
add_3 <- add_x(3)
add_7 <- add_x(7)
add_3(2)
## [1] 5
add_7(2)
## [1] 9
```

Funktionsoperatoren bilden Funktionen auf Funktionen ab.

```
chatty <- function(f) { # Funktionsoperator
  function(x, ...) {
    res <- f(x, ...)
    cat("Processing ", x, "\n", sep = "")
    res
  }
}
sqr <- function(x) x ^ 2
chatty_sqr <- chatty(sqr)
```

```

chatty_sqr(1)
## Processing 1
## [1] 1
s <- c(3, 2, 1)
sapply(s, sqr)
## [1] 9 4 1
sapply(s, chatty_sqr)
## Processing 3
## Processing 2
## Processing 1
## [1] 9 4 1

```

## 2 Funktionale

### 2.1 apply und Co

Die Funktionen der `apply`-Familie zählen zu den Funktionalen. Sie nehmen einen Vektor (oder davon abgeleiteten Datentyp) und eine Funktion als Argumente entgegen und geben einen Vektor (oder Array) aus.

Sie ersetzen häufige `for`-Schleifen-Muster durch eine kompaktere und aussagekräftigere Darstellung.

#### 2.1.1 lapply()

Das einfachste Funktional dieser Familie ist `lapply()`. `lapply(x, f)` wendet die Funktion `f` auf jedes Element des Vektors `x` an. Das Resultat wird als Liste ausgegeben. Mit `lapply(x, f, arg1, arg2)` können der Funktion `f` weitere Argumente übergeben werden.

`lapply()` ist mittels einer `for`-Schleife implementiert.

```

# eigene Implementierung von lapply()
lapply2 <- function(x, f, ...) {
  out <- rep(list(NULL), length(x))
  for (i in seq_along(x)) out[[i]] <- f(x[[i]], ...)
  out
}

```

Der Vorteil von `lapply(f, x)` gegenüber einer expliziten `for`-Schleife ist die kompakte Form und dadurch bessere Lesbarkeit des Codes.

Viele `for`-Schleifen-Muster können durch `lapply` ersetzt werden. Dies funktioniert immer dann, wenn der `i`-te Durchgang der `for`-Schleife nur vom `i`-ten Eintrag des Vektors und keinem der vorangegangenen Ergebnisse abhängt.

```

for (x in xs) {...}
lapply(xs, function(x) {...})

for (i in seq_along(xs)) {...} # besser als for (i in 1:length(xs)) {<...>}
lapply(seq_along(xs), function(i) {<...>})

for (nm in names(xs)) {...}
lapply(names(xs), function(nm) {...})

```

#### 2.1.2 sapply()

Die Funktion `sapply()` führt eine Vereinfachung des Rückgabewertes durch. Dazu wird intern die Funktion `simplify2array()` genutzt.

```

simplify2array(list(1:2, 11:12, 21:22))
##      [,1] [,2] [,3]
## [1,]    1    11   21
## [2,]    2    12   22
simplify2array(list(matrix(1:4, nrow=2), matrix(11:14, nrow=2)))
## , , 1
##
##      [,1] [,2]
## [1,]    1    3
## [2,]    2    4
##
## , , 2
##
##      [,1] [,2]
## [1,]   11   13
## [2,]   12   14
sapply(1:3, function(x, y) c(x, y), y=0)
##      [,1] [,2] [,3]
## [1,]    1    2    3
## [2,]    0    0    0

# eigene Implementierung von sapply()
sapply2 <- function(x, f, ...) {
  res <- lapply2(x, f, ...)
  simplify2array(res)
}

```

### 2.1.3 replicate()

`replicate(n, expr)` führt den Ausdruck `expr`  $n$ -mal aus und gibt das Ergebnis als Array zurück. `replicate(n, expr, simplify=FALSE)` gibt eine Liste aus. Typischerweise ist das Ergebnis von `expr` von Zufallszahlen abhängig (`rDISTRI()`-Funktionen).

```

str(replicate(5, runif(10)))
## num [1:10, 1:5] 0.2815 0.5305 0.4329 0.9172 0.0275 ...
str(replicate(5, runif(sample(1:10, 1))), simplify = FALSE)
## List of 5
## $ : num [1:10] 0.621 0.357 0.536 0.449 0.175 ...
## $ : num [1:5] 0.843 0.64 0.993 0.709 0.753
## $ : num [1:3] 0.446 0.147 0.439
## $ : num [1:10] 0.57 0.476 0.87 0.343 0.668 ...
## $ : num [1:7] 0.631 0.637 0.294 0.52 0.846 ...

```

### 2.1.4 mapply()

`mapply()` verallgemeinert `sapply()` für mehrere Inputs. `mapply(f, x1, x2)` führt `f(x1[[i]], x2[[i]])` für jedes  $i$  aus. Dabei müssen die Längen der Vektoren `x1` und `x2` übereinstimmen. Weitere (konstante) Argumente können `f` mit dem Argument `MoreArgs` in einer Liste übergeben werden. Wird `SIMPLIFY=FALSE` gesetzt, wird eine Liste statt einem atomaren Typ zurückgegeben.

```

mapply(function (x,y,z) c(x-y,y-z,z-x), 1:5, 11:15, 21:25)
##      [,1] [,2] [,3] [,4] [,5]
## [1,] -10  -10  -10  -10  -10
## [2,] -10  -10  -10  -10  -10
## [3,]  20   20   20   20   20

```

```

str(mapply(function (x,y,z) c(x,y,z), 1:3, 11:13, SIMPLIFY=FALSE, MoreArgs=list(z=100)))
## List of 3
## $ : num [1:3] 1 11 100
## $ : num [1:3] 2 12 100
## $ : num [1:3] 3 13 100

```

Achtung: Bei `mapply()` ist die Funktion das erste Argument, nicht das zweite wie bei `lapply()` und `sapply()`.

Achtung: Aus unerfindlichen Gründen heißt bei `mapply()` das Argument `SIMPLIFY` und bei `replicate()` und `sapply()` heißt es `simplify`

### 2.1.5 `rollapply()`

Existiert ein bestimmtes `for`-Schleifen-Muster nicht bereits in R, kann man es oft selbst implementieren.

Wir implementieren die Funktion `rollmean()` um den gleitenden Mittelwert zu berechnen. Für  $k \in \mathbb{N}$  ist der gleitende Mittelwert  $m \in \mathbb{R}^{n-k+1}$  von  $x \in \mathbb{R}^n$  definiert durch

$$m_j := \frac{1}{k} \sum_{i=-\lfloor k/2 \rfloor}^{\lceil k/2 \rceil} x_{j+i}$$

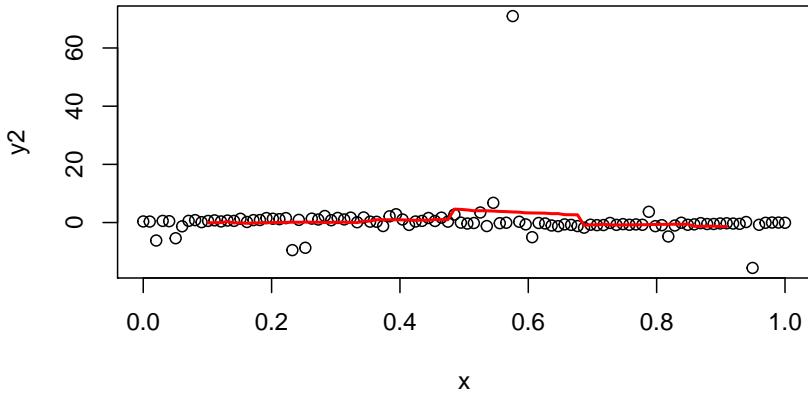
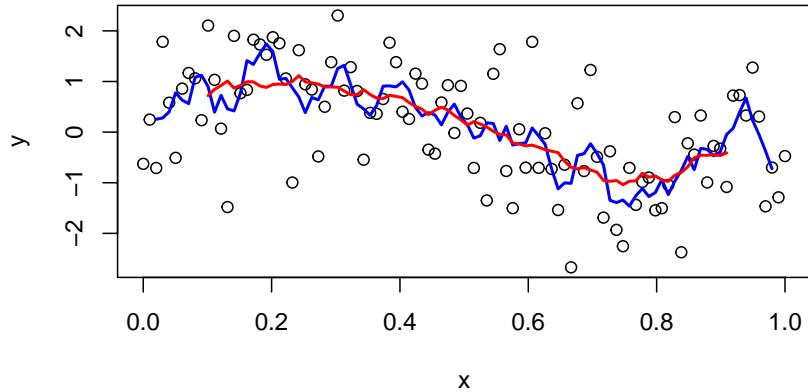
```

rollmean <- function(x, k) {
  out <- rep(NA_real_, length(x))
  neg_offset <- floor(k / 2)
  pos_offset <- ceiling(k / 2) - 1
  for (i in (1 + neg_offset):(length(x) - pos_offset)) {
    out[[i]] <- mean(x[(i - neg_offset):(i + pos_offset)])
  }
  out
}
x <- seq(0, 1, length = 1e2)
fx <- sin(2 * pi * x)

noise1 <- rnorm(1e2)
y <- fx + noise1
plot(x, y)
lines(x, rollmean(y, 5), col = "blue", lwd = 2)
lines(x, rollmean(y, 20), col = "red", lwd = 2)

noise2 <- rcauchy(1e2) / 3
y2 <- fx + noise2
plot(x, y2)
lines(x, rollmean(y2, 20), col = "red", lwd = 2)

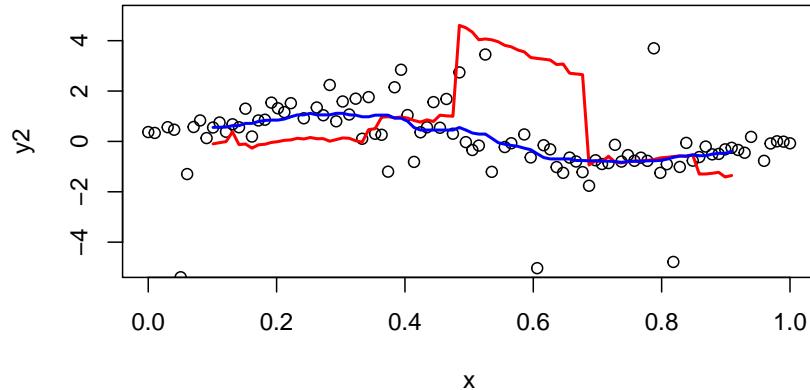
```



Für unseren zweiten Datensatz scheint `rollmean` unangebracht. Wir würden stattdessen gerne eine entsprechende Funktion `rollmedian()` anwenden. Da dafür das gleiche `for`-Schleifen-Muster benötigt wird, bietet es sich an, eine eigene `apply`-Funktion dafür zu implementieren.

```
rollapply <- function(x, k, f, ...) {
  out <- rep(NA_real_, length(x))
  neg_offset <- floor(k / 2)
  pos_offset <- ceiling(k / 2) - 1
  for (i in (1 + neg_offset):(length(x) - pos_offset)) {
    out[[i]] <- f(x[(i - neg_offset):(i + pos_offset)], ...)
  }
  out
}

plot(x, y2, ylim=c(-5,5))
lines(x, rollapply(y2, 20, mean), col = "red", lwd = 2)
lines(x, rollapply(y2, 20, median), col = "blue", lwd = 2)
```



### 2.1.6 Schleifen vs Funktionale

Benutzung von `apply`-Funktionalen führt zu übersichtlicheren Programmen und besser lesbarem Code.

Effizienz ist nicht der Grund, warum `lapply()` statt einer `for`-Schleife genutzt werden sollte. Im Hintergrund einer `apply`-Funktion wird eine Schleife durchlaufen.

```
N <- 1e5
x <- runif(N)
out <- double(N)

# vergleiche:
system.time(for (i in 1:N) out[[i]] <- x[[i]]^2)
##   user  system elapsed
## 0.02    0.00    0.02

# mit:
system.time(sapply(x, `^`, 2))
##   user  system elapsed
## 0.04    0.00    0.05

# eigentlich würde man natürlich so vorgehen:
system.time(x^2)
##   user  system elapsed
## 0      0      0
```

Explizite Schleifen lassen sich in bestimmten Situationen nur schwer vermeiden. Es ist nicht immer möglich, `while`-Schleifen in eine `apply`-Form zu bringen.

### 2.1.7 `apply()`

`apply(A, d, f, ...)` führt die Funktion `f` auf den Dimensionen `d` des Arrays `A` aus. `d` kann ein numerischer oder, falls Dimensionen benannt sind, ein `character` Vektor sein.

`d` gibt diejenigen Dimensionen an, über die iteriert wird. Das Komplement sind diejenigen Dimensionen, die an `f` übergeben werden.

Die Ausgabe von `f` wird als Vektor interpretiert (Dimensionen werden ignoriert) und entlang der ersten Dimension geschrieben.

```

# Matrix (2D)
m <- matrix(1:12, nrow=3)
dimnames(m) <- list(zeilen=character(0), spalten=character(0))
m
##          spalten
## zeilen [,1] [,2] [,3] [,4]
##  [1,]    1    4    7   10
##  [2,]    2    5    8   11
##  [3,]    3    6    9   12
apply(m, 1, range) # iteriere über alle Zeilen
##          zeilen
## spalten [,1] [,2] [,3]
##  [1,]    1    2    3
##  [2,]   10   11   12
# dimnames der Ausgabe von apply ist etwas verwirrend. Besser wäre
fix_nm <- function(arr, name) {
  names(dimnames(arr))[[1]] <- name
  return(arr)
}
fix_nm(apply(m, 1, range), "range") # iteriere über alle Zeilen
##          zeilen
## range [,1] [,2] [,3]
##  [1,]    1    2    3
##  [2,]   10   11   12
fix_nm(apply(m, "spalten", range), "range") # iteriere über alle Spalten
##          spalten
## range [,1] [,2] [,3] [,4]
##  [1,]    1    4    7   10
##  [2,]    3    6    9   12

```

Sei  $\text{Array}(n_1, \dots, n_N)$  die Menge der Arrays mit  $N$  Dimensionen der Länge  $n_1, \dots, n_N$ , dh  $\text{Array}(n_1, \dots, n_N) \subset \mathbb{R}^{n_1 \times \dots \times n_N}$  für **double**-Arrays.

Sei  $\text{Vector}(m)$  die Menge der Vektoren mit Länge  $m$ .

Allgemein gilt:

- Für ein Array  $A \in \text{Array}(n_1, \dots, n_N)$  mit  $N$  Dimensionen,
- einen Dimensions-Index-Vektor  $d = (d_1, \dots, d_D) \in \{1, \dots, N\}$  mit  $d_i \neq d_j$  für  $i \neq j$
- und eine Funktion  $f : \text{Array}(n_{\bar{d}_1}, \dots, n_{\bar{d}_{\bar{D}}}) \rightarrow \text{Vektor}(m)$ , wobei  $\bar{D} = N - D$  und  $\{\bar{d}_1, \dots, \bar{d}_{\bar{D}}\} = \{1, \dots, N\} \setminus \{d_1, \dots, d_D\}$ ,
- gilt  $Y := \text{apply}(A, d, f) \in \text{Array}(m, n_{d_1}, \dots, n_{d_D})$ .
- Dabei gilt  $Y[\cdot, i_1, \dots, i_D] = f(A[\dots, i_1, \dots, i_2, \dots, i_D, \dots])$ , wobei auf der rechten Seite  $i_j$  an der Stelle  $d_j$  steht, für  $i_j \in \{1, \dots, n_{d_j}\}$ .

Beispiel mit  $N = 5, D = 2, m = 20$ :

- $A \in \text{Array}(11, 12, 13, 14, 15)$
- $d = (3, 4)$
- $f : \text{Array}(11, 12, 15) \rightarrow \text{Vektor}(20)$
- $Y := \text{apply}(A, d, f) \in \text{Array}(20, 13, 14)$
- $Y[\cdot, i_1, i_2] = f(A[\cdot, \cdot, i_1, i_2, \cdot])$  für  $i_1 \in \{1, \dots, 13\}, i_2 \in \{1, \dots, 14\}$

```

# Beispiel mit N=3
a <- array(1:24, dim = 2:4, dimnames = list(x1=character(0), x2=character(0), x3=character(0)))
a
## , , 1

```

```

##          x2
## x1      [,1] [,2] [,3]
##  [1,]    1    3    5
##  [2,]    2    4    6
##
## , , 2
##
##          x2
## x1      [,1] [,2] [,3]
##  [1,]    7    9   11
##  [2,]    8   10   12
##
## , , 3
##
##          x2
## x1      [,1] [,2] [,3]
##  [1,]   13   15   17
##  [2,]   14   16   18
##
## , , 4
##
##          x2
## x1      [,1] [,2] [,3]
##  [1,]   19   21   23
##  [2,]   20   22   24
# D=1, m=1
fix_nm(apply(a, 1, colSums), "colSums")
##          x1
## colSums [,1] [,2]
##  [1,]    9   12
##  [2,]   27   30
##  [3,]   45   48
##  [4,]   63   66
fix_nm(apply(a, "x2", colSums), "colSums")
##          x2
## colSums [,1] [,2] [,3]
##  [1,]    3    7   11
##  [2,]   15   19   23
##  [3,]   27   31   35
##  [4,]   39   43   47
# D=2, m=1
apply(a, c(1,2), sum)
##          x2
## x1      [,1] [,2] [,3]
##  [1,]   40   48   56
##  [2,]   44   52   60
# D=2, m=2
apply(a, c("x3","x1"), range)
## , , 1
##
##          x3
## x2      [,1] [,2] [,3] [,4]

```

```

## [1,] 1 7 13 19
## [2,] 5 11 17 23
##
## , , 2
##
##      x3
## x2      [,1] [,2] [,3] [,4]
## [1,] 2 8 14 20
## [2,] 6 12 18 24

```

### 2.1.8 outer()

outer() berechnet das dyadische (äußere) Produkt zweier Arrays.

Für Vektoren  $a, b$  gilt  $Y = \text{outer}(a, b, f)$  mit  $Y[i, j] = f(a[i], b[j])$ .

$f$  muss dabei vektorisiert sein, da  $f$  nur einmal auf entsprechend *recycelten* Arrays  $A, B$  aufgerufen wird, sodass  $f(A, B) = Y = \text{outer}(a, b, f)$ .

```

outer(1:3, 1:5, `*`) # Multiplikationstabelle
##      [,1] [,2] [,3] [,4] [,5]
## [1,] 1 2 3 4 5
## [2,] 2 4 6 8 10
## [3,] 3 6 9 12 15
outer(1:3, 1:5, `+`) # Summationstabelle
##      [,1] [,2] [,3] [,4] [,5]
## [1,] 2 3 4 5 6
## [2,] 3 4 5 6 7
## [3,] 4 5 6 7 8
outer(1:3, 1:5, pmin)
##      [,1] [,2] [,3] [,4] [,5]
## [1,] 1 1 1 1 1
## [2,] 1 2 2 2 2
## [3,] 1 2 3 3 3

```

Allgemein gilt:

- Für  $a \in \text{Array}(n_1, \dots, n_N)$ ,  $b \in \text{Array}(m_1, \dots, m_M)$
- und  $f : \mathbb{R} \times \mathbb{R} \rightarrow \mathbb{R}$  vektorisiert
- ist  $\text{outer}(a, b, f) = f(A, B) \in \text{Array}(n_1, \dots, n_N, m_1, \dots, m_M)$
- mit  $A, B \in \text{Array}(n_1, \dots, n_N, m_1, \dots, m_M)$ , sodass  $A[i_1, \dots, i_N, j_1, \dots, j_M] = a[i_1, \dots, i_N]$  und  $B[i_1, \dots, i_N, j_1, \dots, j_M] = b[j_1, \dots, j_M]$  für alle  $i_1, \dots, i_N, j_1, \dots, j_M$

Angenommen wir haben eine Liste von Funktionen und eine Liste von Datensätzen und möchten jede Funktion auf jeden Datensatz anwenden.

outer() kann auch auf Listen angewendet werden, wenn die Funktion  $f$  für Listen vektorisiert ist.

```

functs <- list(mean=mean, median=median, mid=function(x)(max(x)-min(x))/2)
X <- replicate(5, runif(sample(20,1)), simplify=FALSE)
str(X)
## List of 5
## $ : num [1:14] 0.8744 0.0861 0.9984 0.7357 0.1859 ...
## $ : num [1:11] 0.8918 0.6811 0.6772 0.2367 0.0857 ...
## $ : num [1:14] 0.887 0.621 0.91 0.222 0.159 ...
## $ : num [1:19] 0.484 0.513 0.746 0.95 0.53 ...
## $ : num [1:14] 0.583 0.0304 0.4273 0.2546 0.9599 ...

```

```

call <- function(x_list, f_list) mapply(function(f,x) f(x), f_list, x_list)
outer(X, funs, call)
##      mean     median      mid
## [1,] 0.4722397 0.4776795 0.4561486
## [2,] 0.5943356 0.6656311 0.4411423
## [3,] 0.5489152 0.6404765 0.4185863
## [4,] 0.5186691 0.5095012 0.4229157
## [5,] 0.5359626 0.5687091 0.4647491

```

## 2.1.9 Weitere

- `vapply()`: wie `sapply()` mit Angabe von Ausgabe-Prototyp.
- `rapply()`: rekursiv über geschachtelte Listen

## 2.2 Mathematische Funktionale

Das Paket `stats`, das beim Start von R automatisch geladen wird, enthält unter anderem die Funktionale:

- `integrate()`: integriere eine Funktion
- `optimize()` (bzw `optim()` oder `constrOptim()`): minimiere eine Funktion

### 2.2.1 Integrieren

Für die vektorisierte Implementierung `f()` einer Funktion  $f : [a, b] \rightarrow \mathbb{R}$  nähert `integrate(f, lower=a, upper=b)` den Wert  $\int_a^b f(x)dx$  durch numerische Integration an.

`f()` muss vektorisiert sein, dh für einen `double`-Vektor `x` soll `f(x)` gleich `sapply(x,f)` sein.

```

int <- integrate(sin, 0, pi)
print(int)
## 2 with absolute error < 2.2e-14
str(int)
## List of 5
## $ value      : num 2
## $ abs.error   : num 2.22e-14
## $ subdivisions: int 1
## $ message     : chr "OK"
## $ call        : language integrate(f = sin, lower = 0, upper = pi)
## - attr(*, "class")= chr "integrate"
int$value
## [1] 2

```

`integrate()` wählt automatisch die Unterteilung in Unterintervalle. Die genaue Steuerung des benutzten Algorithmus kann durch weitere Argumente der Funktion übergeben werden. Siehe `?integrate`.

Wir implementieren unsere eigene Version von `integrate()` mittels einer Riemann-Summe.

```

integrate2 <- function(f, lower, upper, subintervals) {
  x <- seq(lower, upper, length.out = subintervals+1)
  sum(f(x[-1])) / subintervals * (upper-lower)
}
integrate2(sin, 0, pi, 10)
## [1] 1.983524
integrate2(sin, 0, pi, 1000)
## [1] 1.999998

```

**Achtung:** Numerische Integration ist nicht trivial. Der für `integrate()` implementierte Algorithmus ist zwar deutlich effizienter als die Riemann-Summe. Allerdings wird eine Funktion, die in weiten Teilen nahezu konstant ist, ggf als konstant interpretiert.

```
density <- function(q) dnorm(q, sd=0.01, mean=1) # integral is 1
integrate(density, lower=-10, upper=10)
## 0 with absolute error < 0
integrate2(density, lower=-10, upper=10, 1000)
## [1] 1.014384
```

## 2.2.2 Optimieren

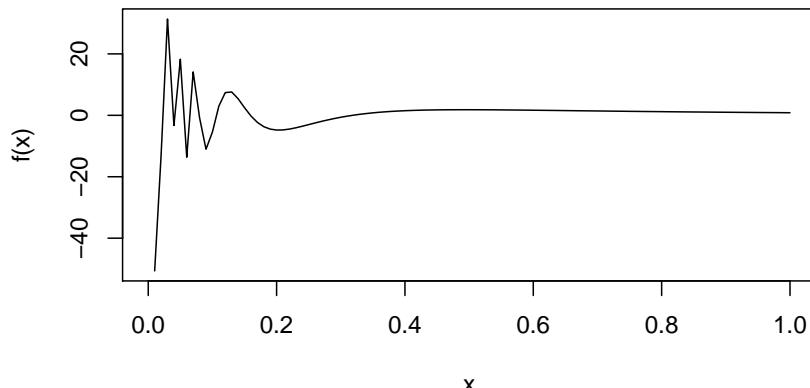
Die Funktion `optimize()` minimiert eine gegebene eindimensionale Funktion innerhalb eines gegebenen Intervalls.

```
opti <- optimize(sin, c(0, 2 * pi))
str(opti)
## List of 2
## $ minimum : num 4.71
## $ objective: num -1
opti$minimum / pi
## [1] 1.500001
```

`optim()` kann Funktionen mit höher-dimensionalen Inputs minimieren.

Beachte, dass numerische Minimierung allgemeiner Funktionen eine schwierige Aufgabe ist, die viel Rechenzeit beanspruchen kann und ggf nicht zufriedenstellende Ergebnisse liefert (zB lokale Minima).

```
f <- function(x) 1/x*sin(1/x)
str(optimize(f, c(0, 1))) # lokales Minimum
## List of 2
## $ minimum : num 0.204
## $ objective: num -4.81
curve(f)
## Warning in sin(1/x): NaNs produced
```



In vielen Fällen sind die Ergebnisse nicht zufriedenstellend. Es empfiehlt sich einen simplen (nicht notwendigerweise effizienten) Algorithmus zum Vergleich zu implementieren: Grid-Search!

```

grid_optim <- function(f, lower, upper, grid_cnt = 5)  {
  grid_vectors <- lapply(seq_along(lower), function(i) seq(lower[[i]], upper[[i]], length.out = grid_cnt))
  grid <- as.matrix(expand.grid(grid_vectors))
  v <- apply(grid, 1, f)
  list(par = grid[which.min(v), ], value = min(v))
}

compare_optim <- function(f, lower, upper, grid_cnt = 5) {
  res_grid <- grid_optim(f, lower, upper, grid_cnt)
  res_optim <- optim((lower+upper)/2, f, method="L-BFGS-B", lower=lower, upper=upper)
  cat("grid: val=",res_grid$value, " par: ",res_grid$par, "\n")
  cat("optim: val=",res_optim$value, " par: ",res_optim$par, "\n")
}

f <- function(x) sum(x^2)
compare_optim(f, c(-3,-1), c(10,20))
## grid: val= 1.0625 par: 0.25 -1
## optim: val= 7.108619e-27 par: 7.019422e-14 -4.670537e-14

g <- function(x) { # has areas of constant value
  z <- sum(x^2)
  if (z < 1) return(0)
  if (z < 2) return(1)
  return(-1)
}
compare_optim(g, c(-10,-10), c(10,10))
## grid: val= -1 par: -10 -10
## optim: val= 0 par: 0 0

```

### 3 Funktionsfabrik

Eine Funktionsfabrik erstellt eine Funktion aus einem Vektor-Typ.

```

power1 <- function(exp) { # eine einfache Funktionsfabrik
  force(exp) # Lazy-Evaluation
  function(x) {
    x ^ exp
  }
}

square <- power1(2)
cube <- power1(3)
square(3)
## [1] 9
cube(3)
## [1] 27

```

Wir erinnern uns an den Begriff der Funktionsumgebung.

```

library(rlang)

env_print(fn_env(square))
## <environment: 000000001C7B60A8>
## parent: <environment: global>

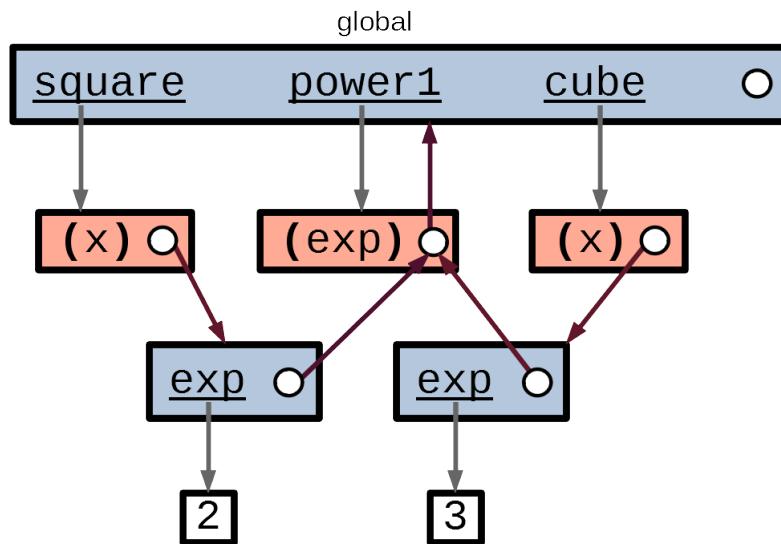
```

```
## bindings:
## * exp: <dbl>
env_print(fn_env(cube))
## <environment: 000000001C752760>
## parent: <environment: global>
## bindings:
## * exp: <dbl>
```

Mit `env_get()` können wir den Wert einer Namensbindung in einer Umgebung abfragen.

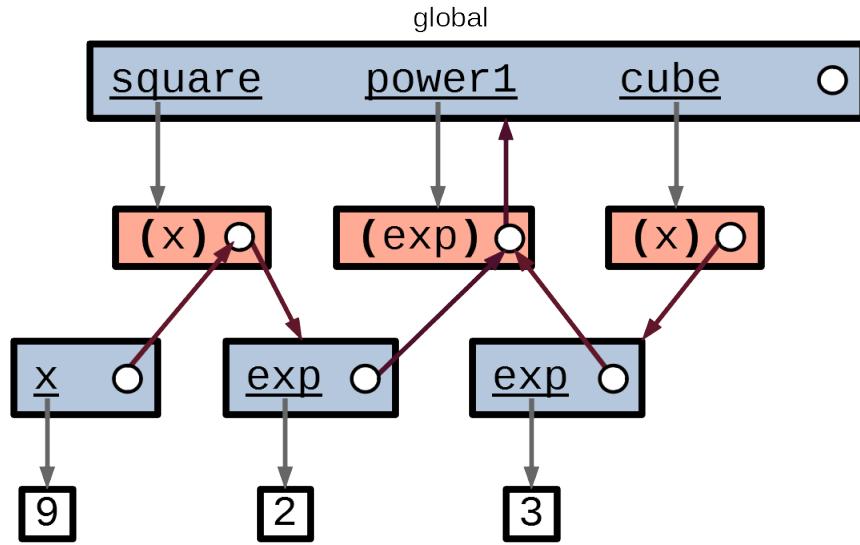
```
env_get(fn_env(square), "exp")
## [1] 2
env_get(fn_env(cube), "exp")
## [1] 3
```

Folgendes Diagramm veranschaulicht die beiden produzierten Funktionen.



Beim Aufruf von `square(9)` wird eine Ausführungsumgebung mit der Namensbindung `x --> 9` erzeugt.

```
square(9)
## [1] 81
```



### 3.1 Counter

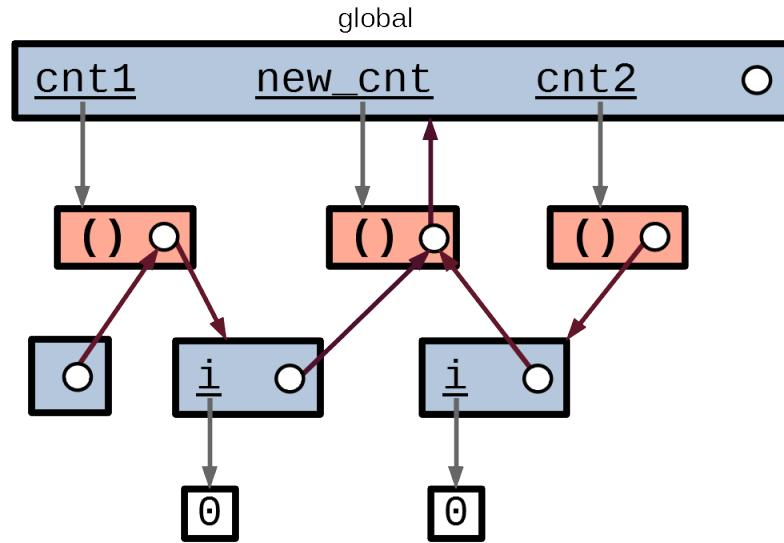
Unter Benutzung des Zuweisungsoperators `<-` bauen wir eine Funktionsfabrik, die Zählfunktionen erstellt.

```

new_cnt <- function() {
  i <- 0
  function() {
    i <- i + 1
    i
  }
}
cnt1 <- new_cnt()
cnt2 <- new_cnt()
  
```

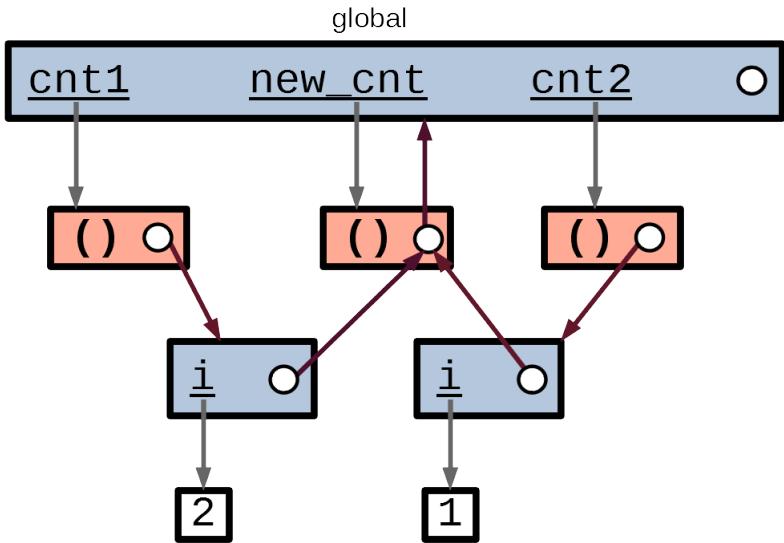
Erinnerung: Die Ausführungsumgebung von `new_cnt()` wird bei jedem Aufruf von `new_cnt()` neu erstellt und übernimmt dann Rolle der Funktionsumgebung der produzierten Funktion `cnt1()` bzw. `cnt2()`.

Beim Aufruf von `cnt1()` entsteht auch eine Ausführungsumgebung. Diese enthält allerdings keine Namensbindungen.



```

cnt1()
## [1] 1
cnt1()
## [1] 2
cnt2()
## [1] 1
  
```



### 3.2 Maximum Likelihood

Wir haben  $n \in \mathbb{N}$  Beobachtungen  $X \in \mathbb{N}_0^n$  bzw  $X$  aus den natürlichen Zahlen mit 0 und gehen davon aus, dass diese unabhängig aus einer Poisson-Verteilung gezogen wurden.

```

X <- c(41, 30, 31, 38, 29, 24, 30, 29, 31, 38)
  
```

Wir wissen jedoch nicht welchen Parameter  $\lambda$  die zugrunde liegende Verteilung hat und möchten diesen aus den Daten schätzen.

Eine Möglichkeit ist der Maximum-Likelihood-Schätzer: Wir betrachten die Zähldichte  $p_{n,\lambda}: \mathbb{N}_0^n \rightarrow [0, \infty)$  der multivariaten Poisson-Verteilung zum Parameter  $\lambda$ , setzen unsere Beobachtungen  $X$  ein und maximieren  $\lambda \mapsto p_{n,\lambda}(X)$  bzgl  $\lambda \in (0, \infty)$ .

Für  $k \in \mathbb{N}_0$  gilt

$$p_{1,\lambda}(k) = \exp(-\lambda) \frac{\lambda^k}{k!}$$

und für  $x \in \mathbb{N}_0^n$  wegen Unabhängigkeit

$$p_{n,\lambda}(x) = \prod_{i=1}^n p_{1,\lambda}(x_i) = \exp(-\lambda n) \prod_{i=1}^n \frac{\lambda^{x_i}}{x_i!}.$$

Häufig wird statt der Dichte selbst ihr Logarithmus maximiert, was das gleiche Ergebnis liefert, da  $x \mapsto \log(x)$  streng monoton wachsend ist. Es gilt

$$\log(p_{n,\lambda}(x)) = -\lambda n + \sum_{i=1}^n (x_i \log(\lambda) - \log(x_i!)) = -\lambda n + \log(\lambda) \sum_{i=1}^n x_i - \sum_{i=1}^n \log(x_i!) .$$

```
log_n_dpois <- function(lambda, x) {
  n <- length(x)
  -n*lambda + log(lambda)*sum(x) - sum(lfactorial(x))
}
```

Wir setzen verschiedene Werte für  $\lambda$  ein. Höhere Werte deuten auf ein  $\lambda$  hin, das besser zu den Daten passt.

```
log_n_dpois(10, x)
## [1] -183.6405
log_n_dpois(30, x)
## [1] -30.98598
log_n_dpois(50, x)
## [1] -67.01095
```

Da die Daten fix sind, können wir eine Funktion  $\lambda \mapsto \log(p_{n,\lambda}(x))$  implementieren, der wir das  $x$  nicht übergeben müssen. An dieser Stelle ist eine Funktionsfabrik sinnvoll.

```
make_log_n_dpois_x <- function(x) {
  n <- length(x)
  function(lambda) {
    -n*lambda + log(lambda)*sum(x) - sum(lfactorial(x))
  }
}
log_n_dpois_x <- make_log_n_dpois_x(X)
log_n_dpois_x(10)
## [1] -183.6405
log_n_dpois_x(30)
## [1] -30.98598
log_n_dpois_x(50)
## [1] -67.01095
```

Wir bemerken, dass einige Terme der log-Dichte nicht von  $\lambda$  sondern nur von  $x$  abhängen. Es genügt diese einmal zu berechnen.

```
make_log_n_dpois_x <- function(x) {
  n <- length(x)
  sum_x <- sum(x)
  sum_fac_x <- sum(lfactorial(x))
```

```

function(lambda) {
  -n*lambda + log(lambda)*sum_x - sum_fac_x
}
}
log_n_dpois_x <- make_log_n_dpois_x(X)
log_n_dpois_x(10)
## [1] -183.6405
log_n_dpois_x(30)
## [1] -30.98598
log_n_dpois_x(50)
## [1] -67.01095

```

Wir wollen natürlich nicht von Hand verschiedene Werte einsetzen, sondern automatisch das Maximum finden. Hierzu dient die Funktion `optimize()`.

```

str(optimize(log_n_dpois_x, c(0, 100), maximum = TRUE))
## List of 2
## $ maximum : num 32.1
## $ objective: num -30.3
# alternativ:
str(optimize(log_n_dpois, c(0, 100), x = X, maximum = TRUE))
## List of 2
## $ maximum : num 32.1
## $ objective: num -30.3

```

Um eine gute Näherung des Maximums zu finden, muss `optimize()` die ihr übergebene Funktion ggf sehr häufig aufrufen. Eine effiziente Implementierung dieser Funktion ist daher von Nutzen.

```

n <- 1e6
Y <- rpois(n, 42)
log_n_dpois_y <- make_log_n_dpois_x(Y)

t <- proc.time()
str(optimize(log_n_dpois_y, c(0, 100), maximum = TRUE))
## List of 2
## $ maximum : num 42
## $ objective: num -3285785
print(proc.time()-t)
##    user  system elapsed
##        0        0        0

t <- proc.time()
str(optimize(log_n_dpois, c(0, 100), x = Y, maximum = TRUE))
## List of 2
## $ maximum : num 42
## $ objective: num -3285785
print(proc.time()-t)
##    user  system elapsed
##  0.75  0.01  0.77

```

Funktionsfabriken haben hier zwei positive Effekte:

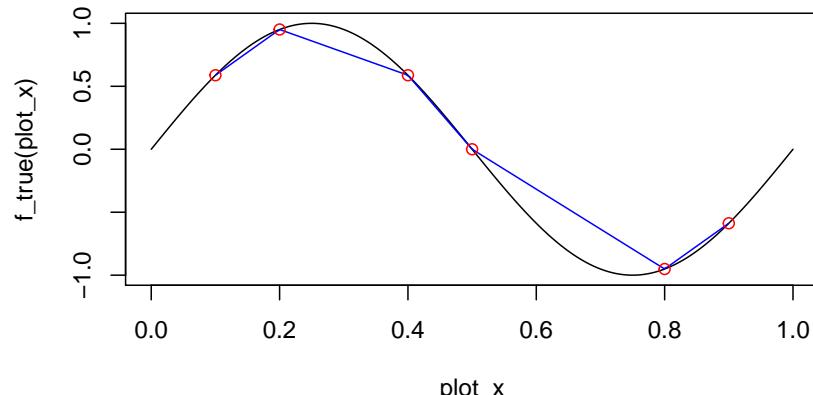
- Wir können einige Werte in der Funktionsfabrik selbst vorherberechnen und damit etwas CPU-Zeit sparen.
- Das zweistufige Design reflektiert die mathematische Struktur des zugrunde liegenden Problems.

Diese Vorteile werden größer bei komplexeren Maximum-Likelihood-Problemen mit mehreren Parametern und Datenvektoren.

### 3.3 Funktionsinterpolation

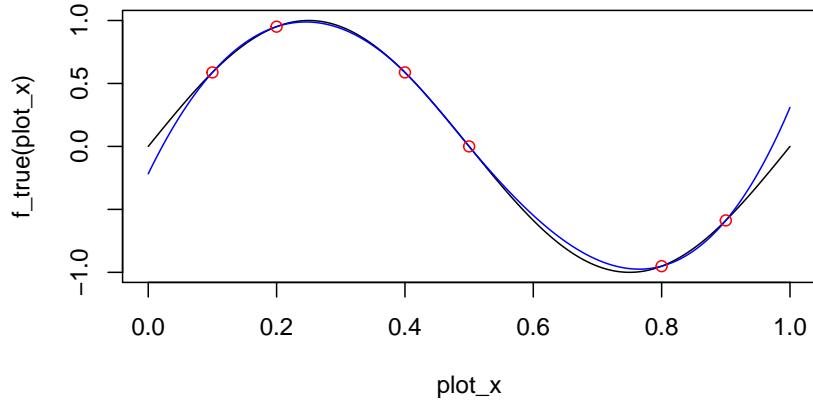
Die Funktionsfabrik `approxfun()` nimmt Werte  $(x_i, y_i)_{i=1, \dots, n}$  entgegen und gibt eine Funktion  $f$  aus, sodass  $f(x_i) = y_i$ . Für Zwischenwerte  $x \notin \{x_1, \dots, x_n\}$  wird ein linear interpolierter Wert zurückgegeben.

```
# erzeuge Daten
f_true <- function(x) sin(x*pi*2)
x <- c(0.1, 0.2, 0.4, 0.5, 0.8, 0.9)
y <- f_true(x)
# berechne (lineare) Interpolationsfunktion
f_lin <- approxfun(x, y)
# plotte Resultate
plot_x <- seq(0, 1, length.out = 200)
plot(plot_x, f_true(plot_x), type = "l")
lines(plot_x, f_lin(plot_x), col="blue")
points(x, y, col="red")
```



Analog zur linearen Interpolation durch `approxfun()` führt `splinefun()` eine Interpolation mit kubischen Splines (Polynome dritten Grades) durch.

```
# berechne (kubische) Interpolationsfunktion
f_cub <- splinefun(x, y)
# plotte Resultate
plot_x <- seq(0, 1, length.out = 200)
plot(plot_x, f_true(plot_x), type = "l")
lines(plot_x, f_cub(plot_x), col="blue")
points(x, y, col="red")
```



## 4 Funktionsoperatoren

Funktionsoperatoren bilden Funktionen auf Funktionen ab. Aus der Mathematik ist etwa der Ableitungsoperator  $D : C^1(\mathbb{R}) \rightarrow C^0(\mathbb{R})$  bekannt. In R werden Funktionsoperatoren meist nicht für rein mathematische Zwecke eingesetzt. Einige Beispiele behandeln wir im Folgenden.

### 4.1 Verhaltensändernde Funktionsoperatoren

Zähle Funktionsaufrufe und schreibe . auf die Konsole, um eine Fortschrittsanzeige zu erzeugen.

```
dot_every <- function(n, f) {
  i <- 1
  function(...) {
    if (i %% n == 0) cat(".")

    i <- i + 1
    f(...)
  }
}
x <- lapply(1:100, runif)
x <- lapply(1:100, dot_every(10, runif))
## .....
```

Um den Effekt sichtbar zu machen (in der live-Ausführung), wollen wir die Ausführung der Funktion `runif()` künstlich verzögern. Wir nutzen dazu `Sys.sleep()` und schreiben einen entsprechenden Funktionsoperator.

```
delay_by <- function(delay, f) {
  function(...) {
    Sys.sleep(delay)
    f(...)
  }
}

new_runif <- dot_every(10, delay_by(0.05, runif))
x <- lapply(1:100, new_runif)
## .....
```

#### 4.1.1 Memoisation

Um die Ausführung einer häufig genutzten Funktion zu beschleunigen, speichern wir alle berechneten Input-Output-Paare in eine Liste. Wird die Funktion mit einem bestimmten Input ein zweites Mal aufgerufen, wird ohne weitere Berechnung der passende Output-Wert nachgeschlagen und ausgegeben.

`memoise::memoise()` ist ein Funktionsoperator, der dieses Verhalten automatisch einer Funktion beibringt.

```
library(memoise)
```

```
fib <- function(n) {
  if (n < 2) return(1)
  fib(n - 2) + fib(n - 1)
}
system.time(fib(25))
##    user  system elapsed
##    0.10    0.00    0.09
system.time(fib(26))
##    user  system elapsed
##    0.16    0.00    0.16
fib <- memoise(fib)
system.time(fib(25))
##    user  system elapsed
##    0.03    0.00    0.03
system.time(fib(26))
##    user  system elapsed
##    0      0      0
```

## 4.2 Ausgabe-Funktionsoperatoren

Anstatt einer Ausgabe auf der Konsole möchten wir den Text als String-Objekt (`character`-Vektor) zurückgeben. Dies ermöglicht die Funktion `capture.output()`.

```
capture_it <- function(f) { # Funktionsoperator
  force(f)
  function(...) {
    capture.output(f(...))
  }
}
str_out <- capture_it(str)
str(1:10)
##  int [1:10] 1 2 3 4 5 6 7 8 9 10
str_out(1:10)
## [1] " int [1:10] 1 2 3 4 5 6 7 8 9 10"
```

Um die CPU-Zeit der Funktion auszugeben, benutze `system.time()`.

```
time_it <- function(f) { # Funktionsoperator
  force(f)
  function(...) {
    system.time(f(...))
  }
}
compute_sum <- list(
  vec = function(x) sum(x),
  loop = function(x) {
    s <- 0
```

```

    for (a in x) s <- s+a
    s
  }
)
x <- runif(1e7)
call_fun <- function(f, ...) f(...)
lapply(compute_sum, time_it(call_fun), x)
## $vec
##   user  system elapsed
##       0       0       0
##
## $loop
##   user  system elapsed
##   0.19   0.00   0.19

```

### 4.3 Eingabe-Funktionsoperatoren

`pryr::partial()` nimmt eine Funktion mit mehreren Parametern entgegen und gibt die gleiche Funktion zurück, sodass bestimmte Parameter auf einen konstanten Wert gesetzt sind. `f <- partial(g, b = 1)` entspricht `f <- function(a) g(a, b = 1)`.

```

library(pryr)
rowapply <- partial(apply, MARGIN=1)
A <- matrix(1:9, nrow=3)
rowapply(A, sum)
## [1] 12 15 18
rowSums(A)
## [1] 12 15 18

```

Eine nicht-vektorisierte Funktion kann mittels `Vectorize()` vektorisiert werden. Dies führt nicht zu einer verbesserten Performance, sondern dient der Vereinfachung des Codes.

```

# vektorisiere das Attribut "size" der Funktion sample(),
# sodass eine Liste zurückgegeben wird (SIMPLYFY = FALSE)
sample2 <- Vectorize(sample, "size", SIMPLIFY = FALSE)
str(sample2(1:5, c(1, 1, 3)))
## List of 3
## $ : int 2
## $ : int 5
## $ : int [1:3] 5 4 1
str(sample2(1:5, 5:3))
## List of 3
## $ : int [1:5] 1 2 4 5 3
## $ : int [1:4] 2 1 4 3
## $ : int [1:3] 2 5 1

```

Um einer Funktion ihre Argumente als Liste zu übergeben, nutze `do.call()`. Mit dem entsprechenden Funktionsoperator wandelt man Funktionen so um, dass sie nur noch ein Argument nehmen: Die Liste der ursprünglichen Argumente.

```

args_as_list <- function(f) {
  force(f)
  function(args) {
    do.call(f, args)
  }
}

```

```
x <- c(NA, runif(100), 1000)
args <- list(
  list(x),
  list(x, na.rm = TRUE),
  list(x, na.rm = TRUE, trim = 0.1)
)
lapply(args, args_as_list(mean))
## [[1]]
## [1] NA
##
## [[2]]
## [1] 10.37333
##
## [[3]]
## [1] 0.4783179
```

# V11 - Objektorientierte Programmierung

24. Mai 2021

## Contents

<b>1</b>	<b>Intro</b>	<b>1</b>
1.1	Beispiel print() . . . . .	1
1.2	Objektorientierte Programmierung . . . . .	2
1.3	OOP in R . . . . .	3
1.4	Sloop . . . . .	3
<b>2</b>	<b>S3</b>	<b>3</b>
2.1	Method-Dispatch . . . . .	5
2.2	Vererbung . . . . .	5
2.3	Common Methods . . . . .	7
2.4	Internal generics . . . . .	10
2.5	Group generics . . . . .	11
<b>3</b>	<b>S4</b>	<b>13</b>
3.1	Klassen . . . . .	13
3.2	Generische Funktionen und Methoden . . . . .	15
3.3	Method-Dispatch . . . . .	18
3.4	S4 and existing code . . . . .	18
<b>4</b>	<b>R6</b>	<b>18</b>
4.1	Klassen und Methoden . . . . .	18
4.2	Referenzsemantik . . . . .	19
4.3	Interna . . . . .	20

## 1 Intro

### 1.1 Beispiel print()

print() gibt ein R-Objekt auf der Konsole aus. Die Art der Ausgabe hängt von der Klasse des Objektes ab.

```
x <- 0
print(x)
## [1] 0
print(structure(x, class="Date"))
## [1] "1970-01-01"
print(structure(x, class=c("POSIXct", "POSIXt")))
## [1] "1970-01-01 01:00:00 CET"
```

Wie würden wir eine solche Methode implementieren?

```
print2 <- function(x) {
  cls <- class(x)[[1]]
  if (cls == "Date") {
    print2.Date(x)
```

```

} else if (cls == "POSIXct") {
  print2.POSIXct(x)
} else {
  print2.default(x)
}
}

# implementiere print2.Date, print2.POSIXct, print2.default

```

Angenommen wir möchten für eine neue Klasse an Objekten eine selbst-definierte `print()`-Funktion implementieren.

```

weight <- structure(x, class = "mass", unit = "kg")
print(weight)
## [1] 0
## attr(,"class")
## [1] "mass"
## attr(,"unit")
## [1] "kg"
print2.mass <- function(m) {
  print(paste(m, attr(m, "unit")))
}
print2.mass(weight)
## [1] "0 kg"

```

Wir müssten die `print()`-Funktion nochmal neu schreiben, um auch die neue Klasse behandeln zu können.

```

print2 <- function(x) {
  cls <- class(x)[[1]]
  if (cls == "Date") {
    print2.Date(x)
  } else if (cls == "POSIXct") {
    print2.POSIXct(x)
  } else if (cls == "mass") {
    print2.mass(x)
  } else {
    print2.default(x)
  }
}

print2(weight)
## [1] "0 kg"

```

Es geht auch einfacher und ohne das Verändern bestehender Funktionen.

```

print.mass <- function(m) {
  print(paste(m, attr(m, "unit")))
}
print(weight)
## [1] "0 kg"

```

Wie ist das nur möglich??? Antwort: Objektorientierte Programmierung und Polymorphismus!

## 1.2 Objektorientierte Programmierung

Objektorientierte Programmierung (kurz OOP) zeichnet sich (unter anderem) durch zwei Merkmale aus: **Polymorphismus** und **Datenkapselung** (*Encapsulation*).

Polymorphismus bedeutet, dass sich der selbe Funktionsname für verschiedene Funktionen nutzen lässt und abhängig von der Klasse der Argumente die richtige Funktion aufgerufen wird.

Bei Datenkapselung werden Daten und Funktionen, die auf diesen Daten operieren, zu einer Einheit zusammengefasst.

In der OOP wird die formale Beschreibung dieser Einheit als **Klasse** bezeichnet. Variablen, die dieser Beschreibung entsprechen, sind **Objekte** der Klasse. Die zur Klasse gehörenden Funktionen heißen **Methoden**, die Daten heißen **Felder** (*fields*).

Der Begriff **Objekt** wird mehrfach verwendet:

- In R: Alles, was existiert, ist ein Objekt (ein R-Objekt).
- In der OOP: Instanzen von Klassen heißen Objekte.

Zwei Unter-Paradigmen der OOP sind **encapsulated OOP** und **functional OOP**.

In der **encapsulated OOP** gehören Methoden direkt zu den Objekten oder Klassen und werden typischerweise als `object.method(arg1, arg2)` aufgerufen.

In der **functional OOP** werden **generische Funktionen** (*generics*) aufgerufen `generic(object, arg1, arg2)`, die in Abhängigkeit von der Klasse des Objekts die richtige Methode aufrufen.

### 1.3 OOP in R

In R gibt es gleich mehrere OOP-Systeme. In Base-R sind enthalten: S3, S4, *reference classes* (RC). Mit zusätzlichen Paketen können weitere OOP-Systeme geladen werden, zB: R6, R.oo, proto.

Wir besprechen die wichtigsten 3 OOP-Systeme: S3, S4, R6.

S3 ist die erste (1992) und simpelste Umsetzung von OOP in R. Es folgt dem Paradigma der functional OOP und erzwingt kaum Datenkapselung. Es gibt nur wenige Regeln, die S3 steuern. Dadurch ist es sehr flexibel, aber auch fehleranfällig.

S4 (1998) setzt ebenfalls auf functional OOP mit mehr Fokus auf Datenkapselung und deutlich strikteren Regeln als S3.

R6 (2017) folgt der encapsulated OOP. R6-Objekte folgen nicht der Copy-on-modify-Semantik.

### 1.4 Sloop

Um OOP in R besser zu verstehen, ist das Paket `sloop` nützlich. Wir werden es im Laufe dieses Kapitels des Öfteren einsetzen.

```
# install.packages("sloop")
library(sloop)
```

Mit `sloop::otype()` lässt sich feststellen, aus welchem OOP-System ein Objekt entstammt.

```
otype(1:10)
## [1] "base"
otype(tibble::tibble(x=1:3, y=3:1)) # Tibbles sind S3-Objekte
## [1] "S3"
otype(lubridate::hm("13:37")) # das Paket lubridate nutzt S4
## [1] "S4"
```

## 2 S3

S3 ist das am häufigsten verwendete OOP-System in R.

Es folgt dem Paradigma der funktionalen OOP.

S3 ist sehr simpel und dadurch leicht zu erlernen.

Wir kennen bereits einige S3-Klassen: `factor`, `data.frame`, `tibble`, `POSIXct`, `Date`, `proc_time`.

Ein S3-Objekt ist nichts anderes als ein Objekt mit Attribut `class`.

```
tyrion <- structure(list(name="Tyrion", age=38), class="Person")
milk <- structure(list(amount=0.75, animal="cow", time_good=0.05), class="Milk")
```

Methoden werden von generischen Funktionen aufgerufen. Dieser Vorgang wird mit **Method-Dispatch** bezeichnet.

Eine generische Funktion besteht aus dem Aufruf der Funktion `UseMethod()`.

```
elapse_time <- function(obj, time) {
  UseMethod("elapse_time")
}
```

Methoden sind Funktionen mit dem Namen `GENERIC.CLASS()`. Beim Aufruf der generischen Funktion wird die Klasse des ersten Argumentes abgefragt und versucht eine entsprechende Methode aufzurufen. Ist keine Methode vorhanden, wird `GENERIC.default()` aufgerufen, falls vorhanden.

```
elapse_time.Person <- function(obj, time) {
  obj$age <- obj$age + time
  return(obj)
}
elapse_time.Milk <- function(obj, time) {
  obj$time_good <- obj$time_good - time
  if (obj$time_good < 0) return("Milk is now spoiled!")
  return(obj)
}
elapse_time.default <- function(obj, time) {
  cat("cannot elapse time for object of class", class(obj), "\n")
}
str(elapse_time(tyrion, 1))
## List of 2
## $ name: chr "Tyrion"
## $ age : num 39
## - attr(*, "class")= chr "Person"
str(elapse_time(milk, 1))
## chr "Milk is now spoiled!"
str(elapse_time(structure(0, class="blub"), 1))
## cannot elapse time for object of class blub
## NULL
```

Stil: Da für S3 Funktionsnamen mit `.` eine besondere Bedeutung haben, sollten “normale” (Funktions-)Namen kein `.` enthalten. Leider brechen auch viele Base-R Funktion mit dieser Konvention (zB `data.frame()`).

Einige bekannte Generics sind zB `print()`, `summary()`, `str()`, `mean()`. Mittels `sloop::ftype()` können wir herausfinden, ob eine Funktion generisch ist.

```
ftype(mean)
## [1] "S3"      "generic"
ftype(elapse_time)
## [1] "S3"      "generic"
ftype(apply)
## [1] "function"
```

S3 ermöglicht es, neue Klassen und Methoden zu definieren, ohne die bisherigen Funktionen verändern zu

müssen.

```
print.Person <- function(x) {  
  print(sprintf("%s (%d)", x$name, x$age))  
}  
print(tyrion)  
## [1] "Tyrion (38)"  
tyrion # gleich print(tyrion)  
## [1] "Tyrion (38)"
```

## 2.1 Method-Dispatch

Generische Funktionen rufen `UseMethod()` auf. Sonst sollten sie keine weiteren Befehle enthalten.

`UseMethod()` hat zwei Argumente:

1. (nötig) das Präfix der aufzurufenden Methoden, typischerweise identisch mit dem Namen der generischen Funktion
2. (optional) das Argument, das zum Method-Dispatch genutzt wird

```
# Dispatches on x  
generic <- function(x, y, ...) {  
  UseMethod("generic")  
}  
  
# Dispatches on y  
generic2 <- function(x, y, ...) {  
  UseMethod("generic2", y)  
}
```

Beachte, dass keine (weiteren) Argumente der generischen Funktion an `UseMethod()` übergeben werden. Diese jedoch alle als Argumente der von `UseMethod()` aufgerufenen Methode zur Verfügung stehen. Um dies zu bewerkstelligen ist viel Dunkle Magie von Nöten. Einen kleinen Einblick in die Dunklen Künste wird uns das nächste Kapitel (Meta-Programmierung) geben.

Alle Methoden einer generischen Funktion werden von `sloop::s3_methods_generic()` angezeigt.

```
s3_methods_generic("elapse_time")  
## # A tibble: 3 x 4  
##   generic   class  visible source  
##   <chr>     <chr>  <lgcl>  <chr>  
## 1 elapse_time default TRUE   .GlobalEnv  
## 2 elapse_time Milk    TRUE   .GlobalEnv  
## 3 elapse_time Person  TRUE   .GlobalEnv
```

## 2.2 Vererbung

Das Attribut `class` ist ein `character`-Vektor. Einträge nach dem ersten werden als Vorfahren der Klasse an Position 1 bezeichnet. Ein Objekt einer Klasse soll auch als Objekt seiner Vorfahren-Klassen fungieren können. Dadurch werden Eigenschaften der Vorfahren vererbt.

```
tib <- tibble::tibble(x=1:3, y=3:1)  
class(tib)  
## [1] "tbl_df"     "tbl"        "data.frame"  
class(Sys.time())  
## [1] "POSIXct"   "POSIXt"
```

`inherits()` gibt aus, ob ein Objekt von einer bestimmten Klasse erbt.

```

inherits(tib, "data.frame")
## [1] TRUE
inherits(tib, "Person")
## [1] FALSE

```

Beim Method-Dispatch wird Vererbung beachtet. Beim Aufruf einer generischen Funktion `function(arg1, arg2, ...)` UseMethod(`prefix`) wird bei S3 Klassen ein Vektor `paste0(prefix, ".", c(class(arg1), "default"))` erstellt und der Reihe nach Methoden dieser Namen gesucht. Die zuerst gefundene Methode wird ausgeführt.

```

tyrion <- structure(
  list(name="Tyrion", age=38),
  house="Lannister",
  class=c("Noble", "Person")
)
varys <- structure(
  list(name="Varys", age=51),
  class=c("Person")
)
varys
## [1] "Varys (51)"
tyrion
## [1] "Tyrion (38)"
print.Noble <- function(x) {
  print(sprintf("%s of house %s (%d)", x$name, attr(x, "house"), x$age))
}
varys
## [1] "Varys (51)"
tyrion
## [1] "Tyrion of house Lannister (38)"

```

`sloop::s3_dispatch()` zeigt an, welche Methoden beim Aufruf einer generischen Funktion gesucht und gefunden werden.

```

s3_dispatch(print(tib))
##   print.tbl_df
## => print.tbl
## * print.data.frame
## * print.default
s3_dispatch(print(varys))
## => print.Person
## * print.default
s3_dispatch(print(tyrion))
## => print.Noble
## * print.Person
## * print.default

```

Bei impliziten Klassen (kein explizites `class`-Attribut, zB Matrizen) gibt `class()` leider keine vollständige Auskunft darüber, welche Methodennamen gesucht werden.

```

x <- matrix(1:10, nrow = 2)
s3_dispatch(print(x))
##   print.matrix
##   print.integer
##   print.numeric
## => print.default

```

Die Funktion `sloop::s3_class()` zeigt die korrekte Liste an.

```
s3_class(x)
## [1] "matrix"  "integer" "numeric"
```

Der Aufruf von `NextMethod()` in einer Methode führt zum Aufruf der nächsten Methode in der Vererbungshierarchie.

```
showoff <- function(x) UseMethod("showoff")
showoff.default <- function(x) print("showoff: default")
showoff.a <- function(x) {
  print("showoff: a")
  NextMethod()
}
showoff.b <- function(x) {
  print("showoff: b")
  NextMethod()
}
x <- structure(list(), class = c("z", "a", "b", "c"))
s3_dispatch(showoff(x))
##      showoff.z
## => showoff.a
## -> showoff.b
##      showoff.c
## -> showoff.default
showoff(x)
## [1] "showoff: a"
## [1] "showoff: b"
## [1] "showoff: default"
```

## 2.3 Common Methods

### 2.3.1 Basis-Konstruktoren

Der **Konstruktor** einer Klasse ist eine Funktion, die ein Objekt der Klasse erstellt.

Die Namenskonvention für Konstruktoren ist `new_CLASS()`

```
new_Date <- function(x = double()) {
  stopifnot(is.double(x))
  structure(x, class = "Date")
}

new_Date(c(-1, 0, 1))
## [1] "1969-12-31" "1970-01-01" "1970-01-02"

new_Person <- function(x) {
  stopifnot(is.list(x))
  structure(x, class="Person")
}

new_Person(list(name="Daenerys", age=22))
## [1] "Daenerys (22)"
```

`stopifnot(expr)` gibt eine Fehlermeldung aus, falls `expr` nicht zu ausschließlich TRUE evaluiert.

```

stopifnot(TRUE)
stopifnot(c(T, T))
stopifnot(1 == 0)
## Error: 1 == 0 is not TRUE
stopifnot("Haus")
## Error: "Haus" is not TRUE

```

Beim Erstellen einer Unterklasse sollten folgende Prinzipien beachtet werden:

1. Eine Unterklasse sollte auf dem selben Datentyp aufbauen wie die Elternklasse.
2. Das Klassenattribut der Unterklasse hat die Form `c(subclass, parent_class)`
3. Die Unterklasse sollte alle Felder und Attribute der Elternklasse haben.

Diese Prinzipien sollten durch den Konstruktor gesichert sein.

Um Unterklassen zu erlauben, sollte der Konstruktor der Elternklasse ... und `subclass = NULL` als zusätzliche Argumente haben.

```

new_my_class <- function(x, attr1, ..., subclass = NULL) {
  stopifnot(is.numeric(x))
  stopifnot(is.logical(attr1))
  structure(
    x,
    attr1name = attr1,
    ...,
    class = c(subclass, "my_class")
  )
}
new_subclass <- function(x, attr1, attr2, ..., subclass = NULL) {
  stopifnot(is.character(attr2))
  new_my_class(x, attr1, attr2name = attr2, ..., subclass = c(subclass, "subclass"))
}

new_Person <- function(x, ..., subclass = NULL) {
  stopifnot(is.list(x))
  structure(
    x,
    ...,
    class = c(subclass, "Person")
  )
}
new_Noble <- function(x, house, ..., subclass = NULL) {
  stopifnot(is.character(house))
  new_Person(x, house=house, ..., subclass = c(subclass, "Noble"))
}
tyrion <- new_Noble(list(name="Tyrion", age=38), "Lannister")
str(tyrion)
## List of 2
## $ name: chr "Tyrion"
## $ age : num 38
## - attr(*, "house")= chr "Lannister"
## - attr(*, "class")= chr [1:2] "Noble" "Person"

```

### 2.3.2 Validators

Da S3-Klassen keine Struktur aufgezwungen wird, können leicht unsinnige Objekte entstehen.

Um zu prüfen, ob ein Objekt einer Klasse der erwarteten Struktur entspricht, bietet es sich an eine Validator-Funktion `validate_CLASS()` zu schreiben.

```
validate_Person <- function(p) {
  if (!inherits(p, "Person")) stop('Object must inherit "Person".')
  x <- unclass(p)
  if (!is.list(x)) stop('Object must be a list.')
  if (is.null(x$name)) stop('List must contain the entry name.')
  if (is.null(x$age)) stop('List must contain the entry age.')
  if (!is.character(x$name)) stop('Name must be of type character.')
  if (!is.numeric(x$age)) stop('Age must be a numeric type.')
  if (length(x$name) != 1) stop('Name must be of length 1.')
  if (length(x$age) != 1) stop('Age must be of length 1.')
  invisible(p)
}
mr_cheese <- structure(list(name="Cheddar", weight=200), class="Person")
validate_Person(tyrion)
validate_Person(mr_cheese)
## Error in validate_Person(mr_cheese): List must contain the entry age.
```

### 2.3.3 Hilfs-Konstruktoren

Um das Erstellen von Objekten einer Klasse zu vereinfachen, definiert man entsprechende Hilfsfunktionen, mit dem Namen `CLASS()`.

```
Person <- function(name, age) {
  p <- new_Person(list(name=name, age=age))
  validate_Person(p)
  p
}
Person("Daenerys", 22)
## [1] "Daenerys (22)"
Noble <- function(name, age, house) {
  p <- new_Noble(list(name=name, age=age), house)
  validate_Person(p)
  p
}
dany <- Noble("Daenerys", 22, "Targaryen")
dany
## [1] "Daenerys of house Targaryen (22)"
```

### 2.3.4 Coercion

Um Coercion (Typenumwandlung) für eine selbst definierte S3-Klasse durchzuführen, empfiehlt es sich generische Funktionen vom Namen `as_CLASS()` zu definieren.

Achtung: In Base-R wird dieser Konvention nicht nachgekommen. Funktionen zur expliziten Typenumwandlung heißen dort meist `as.CLASS()`, zB `as.factor()`. Sie sind jedoch nicht Methoden der generischen Funktion `as()`. Sie sind auch selbst nicht generisch, wodurch etwa `as.factor()` nicht auf zusätzliche Klassen (als Eingabe) erweitert werden kann.

Die Methode `as_CLASS.CLASS()` sollte ihre Eingabe unverändert zurückgeben.

Die Methode `as_CLASS.default()` sollte ggf eine passende Fehlermeldung ausgeben.

```
as_Person <- function(x, ...) UseMethod("as_Person")
as_Person.Person <- function(x) x
```

```

as_Person.character <- function(x, age) {
  Person(x, age)
}
as_Person.list <- function(x, ...) {
  p <- new_Person(x)
  validate_Person(p)
  p
}
as_Person(varys)
## [1] "Varys (51)"
as_Person(list(name="Arya", age=14))
## [1] "Arya (14)"
as_Person("Cersei", age=33)
## [1] "Cersei (33)"

```

Unterklassen sollte man in ihre Elternklasse wandeln können. Dies sollte mit einer entsprechenden Methode `as_CLASS.SUB_CLASS()` oder `as_CLASS.CLASS()` der Coercion-Generic `as_CLASS()` geschehen.

```

as_Person.Noble <- function(x) {
  attr(x, "house") <- NULL
  class(x) <- "Person"
  x
}
tyrion
## [1] "Tyrion of house Lannister (38)"
as_Person(tyrion)
## [1] "Tyrion (38)"

```

## 2.4 Internal generics

Einige generische Funktionen rufen nicht `UseMethod()` auf, da sie in C implementiert sind und stattdessen entsprechende C-Funktionen aufrufen. Diese Funktionen heißen **internal Generics**. Siehe `?InternalMethods`.

```

x <- sample(10)
s3_dispatch(x[1])
## [.integer
## [.numeric
## [.default
## -> [.internal]
s3_dispatch(mtcars[1])
## -> [.data.frame
## [.default
## -> [.internal]
`[[.Noble` <- function(x, i) {
  if (i == 1 || i == "name")
    return(sprintf("%s of house %s", x$name, attr(x, "house")))
  unclass(x)[[i]]
}
tyrion[["name"]]
## [1] "Tyrion of house Lannister"
tyrion[[1]]
## [1] "Tyrion of house Lannister"
tyrion[[2]]
## [1] 38

```

Internal Generics gibt es nur in Base-R. Internal Generics können nicht selbst definiert werden.

Internal Generics führen Method Dispatch nur bei expliziten Klassen durch, nicht bei impliziten.

```
f <- function(...) UseMethod("f")
f.Person <- function(...) return("f: Person")
f.numeric <- function(...) return("f: numeric")
f(tyrion) # method-dispatch for explicit class
## [1] "f: Person"
f(1) # method-dispatch for implicit class
## [1] "f: numeric"
# c() is internal generic
c.Person <- function(...) return("c: Person")
c.numeric <- function(...) return("c: numeric")
c(tyrion) # method-dispatch for explicit class
## [1] "c: Person"
c(1) # no method-dispatch for implicit class
## [1] 1
```

## 2.5 Group generics

Es gibt 4 Group-Generics: Math, Ops, Summary und Complex. Zu jeder Group-Generic gehört eine Gruppe interner generischer Funktionen. Zu Summary gehören etwa die Funktionen `all()`, `any()`, `sum()`, `prod()`, `min()`, `max()`, `range()`. Siehe `?groupGeneric`.

Funktionen, die zu einer Group-Generic gehören, haben in ihrer Dispatch-Liste zusätzlich Methoden der Group-Generics (hinter den eigenen Methoden). Ist keine spezifische Methode definiert, kann eine Group-Generic-Methode aufgerufen werden.

Group-Generics existieren nur in Base-R.

```
s3_dispatch(min(Sys.time()))
##   min.POSIXct
##   min.POSIXt
##   min.default
## => Summary.POSIXct
##   Summary.POSIXt
##   Summary.default
## -> min (internal)
min(Sys.time(), as.POSIXct("2020-02-20 20:20:20"))
## [1] "2020-02-20 20:20:20 CET"

df <- data.frame(a = c(-1,1), b = c(T,F))
s3_dispatch(sin(df))
##   sin.data.frame
##   sin.default
## => Math.data.frame
##   Math.default
## * sin (internal)
sin(df)
##      a      b
## 1 -0.841471 0.841471
## 2  0.841471 0.000000
```

Als Beispiel implementieren wir die Group-Generic `Summary` für die Klasse `mass`.

```

weight <- structure(1:3, class="mass", unit="kg")
weight
## [1] "1 kg" "2 kg" "3 kg"
sum(weight)
## [1] 6
s3_dispatch(sum(weight))
##     sum.mass
##     sum.default
##     Summary.mass
##     Summary.default
## => sum (internal)
Summary.mass <- function(x, ...) {
  m <- NextMethod()
  structure(m, class = "mass", unit = attr(x, "unit"))
}
s3_dispatch(sum(weight))
##     sum.mass
##     sum.default
## => Summary.mass
##     Summary.default
## -> sum (internal)
sum(weight)
## [1] "6 kg"
max(weight)
## [1] "3 kg"

```

### 2.5.1 Double Dispatch

Bei Generics der Group-Generic `Ops` (enthält `+`, `-`, `*`, `...`) ist die ausgeführte Methode von der Klasse beider Argumente abhängig. Dies nennt man Multiple- oder Double-Dispatch.

Die Dispatch-Listen beider Argumente werden nach einer passenden Methode durchsucht. Falls die gefundenen Methoden gleich sind oder nur für ein Argument eine (externe) Methode gefunden wurde, wird diese angewendet. Ansonsten wird die interne Methode genutzt.

```

john <- Person("John", 21)
`+` <- function(x, y) {
  if ("Person" %in% class(x) && "Person" %in% class(y) &&
      all(sort(c(x$name, y$name)) == sort(c("John", "Daenerys"))))
    return("Happy End(?)")
  else
    return("...")}
}
1 + 1
## [1] 2
1 + john
## [1] "..."
dany + 1
## [1] "..."
dany + john
## [1] "Happy End(?)"

```

### 3 S4

In S3 gibt es kaum Regeln. Viele unsinnige Konstrukte können erstellt werden.

```
x <- 1:5
class(x) <- "data.frame" # Unsinn
x
## NULL
## <0 rows> (or 0-length row.names)
```

Dies kann leicht zu unvorhersehbarem Verhalten und Fehlern im Code führen, die schwer aufzuspüren sind.

S4 ist in der Funktionsweise ähnlich zu S3, besitzt jedoch striktere Regeln, die einige Unsinn von vorn herein unmöglich machen.

Wie S3 folgt auch S4 dem Paradigma der funktionalen OOP, ist dabei jedoch deutlich formaler und strikter als S3.

- S4-Klassen haben eine formale Definition via `setClass()`.
- S4-Klassen können mehrere (gleichberechtigte) Eltern haben (*multiple inheritance*).
- Felder von S4-Klassen heißen Slots. Auf sie wird mittels des Operators `@` zugegriffen.
- Methoden werden mittels `setMethod()` definiert.
- S4-Generic können Method-Dispatch in Abhängigkeit mehreren Argumenten durchführen (*multiple dispatch*).

S4 ist im Paket `methods` implementiert. Dies ist in RStudio automatisch geladen, jedoch nicht unbedingt im Batch-Mode (Kommandozeilenprogramm `RScript`). Deswegen ist es sinnvoll bei der Nutzung von S4 das Paket explizit zu laden.

```
library(methods)
```

#### 3.1 Klassen

S4-Klassen werden mit `setClass()` erzeugt. Dieser Funktion wird der Name der Klasse (typischerweise in UpperCamelCase), die Slots (Felder) und ggf die Eltern-Klasse (Argumentname `contains`) übergeben.

Slots werden als benannter `character`-Vektor übergeben. Die Namen werden zu den Namen der Slots. Die Werte sind die Klasse (auch implizite) der Slots.

```
.Person <- setClass("Person",
  slots = c(name = "character", age = "numeric")
)
.Employee <- setClass("Employee",
  contains = "Person",
  slots = c(boss = "Person")
)
```

Der Rückgabewert von `setClass()` ist der Basis-Konstruktors der Klasse. Er überprüft, ob die Klassen seiner Argumente den in `setClass()` geforderten Klassen entsprechen.

```
tyrion <- .Person(name = "Tyrion", age = 38)
.Person(name = "Night King", age = FALSE)
## Error in validObject(.Object): invalid class "Person" object: invalid object for slot "age" in class
```

Auf Slots kann mittels `@` oder `slot()` zugegriffen werden. Bei Zuweisungen wird Korrektheit von Typ/Klasse geprüft.

```
tyrion$name
## [1] "Tyrion"
slot(tyrion, "age")
```

```
## [1] 38
tyrion$age <- tyrion$age + 1
tyrion$name <- 42 # ERROR
## Error in (function (cl, name, valueClass) : assignment of an object of class "numeric" is not valid ...
```

slotNames() gibt die Namen aller Slots eines Objektes aus.

```
slotNames(tyrion)
## [1] "name" "age"
```

Um weitere Checks durchzuführen und mehr Kontrolle über die Erzeugung eines Objektes zu haben, werden weitere Konstruktoren definiert.

```
Person <- function(name, age = NA_real_, ...) {
  stopifnot(length(name) == 1 && length(age) == 1)
  .Person(name = name, age = age)
}
.Person() # Setzt Default-Werte.
## An object of class "Person"
## Slot "name":
## character(0)
##
## Slot "age":
## numeric(0)
Person()
## Error in stopifnot(length(name) == 1 && length(age) == 1): argument "name" is missing, with no default
Person("Night King")
## An object of class "Person"
## Slot "name":
## [1] "Night King"
##
## Slot "age":
## [1] NA
```

Ein unbenanntes Argument des Basis-Konstruktors wird als Objekt der Eltern-Klasse interpretiert. Dadurch vereinfachen sich Konstruktoren von Unterklassen.

```
Employee <- function(name, age, boss) {
  person <- Person(name = name, age = age)
  .Employee(person, boss = boss)
}
```

Um die Klasse eines Objektes festzustellen, nutze is().

```
is(tyrion)
## [1] "Person"
pod <- Employee("Podrick", 19, boss=tyrion)
is(pod)
## [1] "Employee" "Person"
is(pod, "Person")
## [1] TRUE
is(tyrion, "Imp")
## [1] FALSE
```

Der Typ eines S4 Objektes ist S4. Die Klasse ist die entsprechende S4-Klasse (ohne Eltern).

```
typeof(tyrion)
## [1] "S4"
class(tyrion)
## [1] "Person"
## attr(,"package")
## [1] ".GlobalEnv"
class(pod)
## [1] "Employee"
## attr(,"package")
## [1] ".GlobalEnv"
```

Das Paket lubridate stellt einige S4 Klassen bereit, zB Period.

```
library(lubridate)
##
## Attaching package: 'lubridate'
## The following objects are masked from 'package:base':
##
##     date, intersect, setdiff, union
time <- hms("1:23:45")
time
## [1] "1H 23M 45S"
class(time)
## [1] "Period"
## attr(,"package")
## [1] "lubridate"
typeof(time)
## [1] "double"
otype(time)
## [1] "S4"
slotNames(time)
## [1] ".Data"   "year"    "month"   "day"     "hour"    "minute"
time@minute
## [1] 23
time@.Data
## [1] 45
is(time, "Period")
## [1] TRUE
```

## 3.2 Generische Funktionen und Methoden

Eine neue generische S4 Funktion wird mit `setGeneric()` erzeugt. Dabei wird der Name der generischen Funktion übergeben und eine Funktion, die `standardGeneric()` aufruft.

```
setGeneric("myGeneric", function(x) standardGeneric("myGeneric"))
```

Methoden werden mit `setMethod(f, signature, definition)` hinzugefügt. `f` ist der Name der Generic als String, `signature` die Klasse dieser Methode, und `definition` die Funktion, die zu der entsprechenden Methode gemacht werden soll.

```
setMethod("myGeneric", "Person", function(x) {
  # method implementation
})
```

### 3.2.1 Show

S4-Objekte werden nicht durch `print()` ausgegeben sondern durch `show()`.

Um die Argumente von `show()` herauszufinden, finden wir diese Generic mittels `getGeneric()` und geben die Argumente mittels `formals()` aus.

```
names(formals(getGeneric("show")))
## [1] "object"

setMethod("show", "Person", function(object) {
  cat(is(object)[[1]], "\n",
      "  Name: ", object@name, "\n",
      "  Age:  ", object@age, "\n",
      sep = ""
  )
})
tyrion
## Person
##  Name: Tyrion
##  Age: 39
```

### 3.2.2 Zugriffsmethoden

Häufig wird ein Programmierstil verwendet, bei dem ein Nutzer einer Klasse nicht mit `@` oder `slot()` auf Solts zugreift, sondern mittels eigens erzeugter Zugriffsmethoden. Dadurch erhält der Erzeuger der Klasse mehr Kontrolle und kann die Validität der Objekte der Klasse sichern.

Zugriffsmethoden unterteilen sich in *Getter* (geben Wert zurück) und *Setter* (Zuweisung eines neuen Wertes).

Zugriffsmethoden können einfache Funktionen sein. Oft bietet es sich jedoch an sie als Methoden einer generischen Funktion zu erstellen.

```
setGeneric("name", function(x) standardGeneric("name"))
## [1] "name"
setMethod("name", "Person", function(x) x@name) # getter
name(tyrion)
## [1] "Tyrion"

setGeneric("name<-", function(x, value) standardGeneric("name<-"))
## [1] "name<-"
setMethod("name<-", "Person", function(x, value) { # setter
  stopifnot(length(value) == 1) # validity test
  x@name <- value
  x
})
name(tyrion) <- "Tyrion Lannister"
name(tyrion)
## [1] "Tyrion Lannister"
```

### 3.2.3 Coercion

Coercion wird mit `as()` durchgeführt.

Für verwandte Klassen gibt es eine natürliche Umwandlung.

```
as(pod, "Person")
## Person
```

```
##  Name: Podrick
##  Age: 19

tyrion_employee <- as(tyrion, "Employee")
tyrion_employee@boss
## Person
##  Name:
##  Age:
```

Um eigene Methoden zur Typenumwandlung der generischen Funktion `as()` hinzuzufügen, verwende `setAs()`.

```
setAs("Person", "Employee", function(from) {
  stop("Can not coerce an Person to an Employee", call. = FALSE)
})
as(tyrion, "Employee")
## Error: Can not coerce an Person to an Employee
```

### 3.2.4 Introspection

Zu einer Generic oder Klasse gehörenden Methoden können mit `sloop::s4_methods_generic()` bzw `sloop::s4_methods_class()` aufgelistet werden.

```
s4_methods_generic("initialize")
## # A tibble: 15 x 4
##   generic   class      visible source
##   <chr>     <chr>      <lgcl>  <chr>
## 1 initialize .environment    TRUE    ""
## 2 initialize ANY             TRUE    "methods"
## 3 initialize array           TRUE    ""
## 4 initialize environment    TRUE    ""
## 5 initialize envRefClass   TRUE    "methods"
## 6 initialize externalRefMethod TRUE    ""
## 7 initialize matrix          TRUE    ""
## 8 initialize MethodsList    TRUE    ""
## 9 initialize Module          TRUE    "Rcpp"
## 10 initialize mts            TRUE    ""
## 11 initialize oldClass       TRUE    ""
## 12 initialize Period         TRUE    "lubridate"
## 13 initialize signature      TRUE    ""
## 14 initialize traceable     TRUE    ""
## 15 initialize ts             TRUE    ""

s4_methods_class("Person")
## # A tibble: 6 x 4
##   generic   class  visible source
##   <chr>     <chr>  <lgcl>  <chr>
## 1 coerce    Person  TRUE   R_GlobalEnv
## 2 coerce<- Person  TRUE   R_GlobalEnv
## 3 coerce<-> Person TRUE   R_GlobalEnv
## 4 name     Person  TRUE   R_GlobalEnv
## 5 name<-> Person TRUE   R_GlobalEnv
## 6 show     Person  TRUE   R_GlobalEnv
```

### 3.3 Method-Dispatch

Der Method-Dispatch kann in S4 etwas komplizierter werden als in S3 wegen 2 zusätzlicher Features: *multiple Inheritance* (Klasse kann mehrere Eltern haben) und *multiple Dispatch* (Dispatch abhängig von Klassen mehrerer Argumente).

Für mehr Informationen zu Method-Dispatch in S4, siehe <https://adv-r.hadley.nz/s4.html#s4-dispatch>

### 3.4 S4 and existing code

Für die Argumente `slots` und `contains` der Funktion `setClass()` können S4-Klassen, S3-Klassen und implizite Klassen verwendet werden. Um eine S3 Klasse verwenden zu können, muss sie jedoch erst mittels `setOldClass()` registriert werden, zB `setOldClass("data.frame")`.

Die S3-Klassen aus Base-R werden in den Paketen von Base-R registriert. Wir müssen sie also nicht noch einmal selbst registrieren. Registrierung ist in erster Line für selbst erstellte S3-Klassen nötig.

Erbt eine S4-Klasse von einer S3-Klasse oder einer impliziten Klasse, ist das vererbte Objekt über den automatisch generierten Slot `.Data` abrufbar.

```
RangedNumeric <- setClass(  
  "RangedNumeric",  
  contains = "numeric",  
  slots = c(min = "numeric", max = "numeric"))  
)  
rn <- RangedNumeric(1:10, min = 1, max = 10)  
rn@min  
## [1] 1  
rn@Data  
## [1] 1 2 3 4 5 6 7 8 9 10
```

## 4 R6

**R6** ist ein weiteres OOP-System für R. Die wichtigsten Unterschiede zu S3 und S4 sind:

- R6 setzt encapsulated OO um: Es gib keine generischen Funktionen; Methoden gehören zu Objekten.
- Keine Copy-on-modify-Semantik sondern Referenzsemantik.
- R6 gehört nicht zu Base-R. Es muss durch das Paket "**R6**" geladen werden.

```
library(R6)
```

### 4.1 Klassen und Methoden

Um R6-Klassen zu erzeugen, wird die Funktion `R6::R6Class()` benutzt. Dies ist die einzige Funktion aus dem Paket **R6**, die wir benötigen.

Das erste Argument von `R6::R6Class()` ist der Name der Klasse, nach Konvention in UpperCamelCase.

Das zweite Argument von `R6::R6Class()` trägt den Namen `public`. Wir übergeben damit eine Liste aller Methoden und Feldern, die einem Nutzer der erzeugten Klasse zur Verfügung stehen sollen. Namen von Methoden und Feldern werden nach Konvention in `snake_case` (alles klein, Wörter durch `_` getrennt) angegeben.

Methoden können auf andere Methoden und Slots mittels `self$ELEMENT` zugreifen. `self` entspricht dem aktuellen Objekt.

```
Accumulator <- R6Class("Accumulator", list(  
  sum = 0, # a field
```

```

  add = function(x = 1) { # a method
    self$sum <- self$sum + x # method accesses field "sum" via "self$"
  })
)

```

Der Rückgabewert von `R6::R6Class()` ist ein Objektgenerator der erzeugten Klasse.

```

Accumulator
## <Accumulator> object generator
##   Public:
##     sum: 0
##     add: function (x = 1)
##     clone: function (deep = FALSE)
##   Parent env: <environment: R_GlobalEnv>
##   Locked objects: TRUE
##   Locked class: FALSE
##   Portable: TRUE

```

Mit der Methode `new()` eines Objektgenerators können Objekte erzeugt werden. Sie wird mittels `OBJECT_GENERATOR$new()` aufgerufen.

```

x <- Accumulator$new()
x
## <Accumulator>
##   Public:
##     add: function (x = 1)
##     clone: function (deep = FALSE)
##     sum: 0

```

Auf Felder und Methoden eines R6-Objektes wird mit `$` zugegriffen.

```

x$sum
## [1] 0
x$add(4)
x$sum
## [1] 4

```

Für R6-Objekte ist das `class` Attribut gesetzt. Alle R6-Objekte erben von der Klasse `R6`.

```

attributes(x)
## $class
## [1] "Accumulator" "R6"

```

## 4.2 Referenzsemantik

R6-Objekte und Objektgeneratoren sind Umgebungen. Dies erklärt den Zugriff auf Elemente mittels `$`.

```

typeof(Accumulator)
## [1] "environment"
typeof(x)
## [1] "environment"

```

Da R6-Objekte Umgebungen sind, folgen sie nicht der Copy-on-Modify-Semantik sondern der Referenzsemantik.

```

y <- x
y$sum <- 0
c(x$sum, y$sum)

```

```
## [1] 0 0
y$add(100)
c(x$sum, y$sum)
## [1] 100 100
```

Jedes R6-Objekt hat eine Methode namens `clone()`. Wegen der Referenzsemantik ist sie nötig, um Kopien eines Objektes zu erstellen.

```
z <- x$clone()
z$add(10)
c(x$sum, z$sum)
## [1] 100 110
```

Achtung: Enthalten R6-Objekte andere R6-Objekte als Felder, muss `clone(deep=TRUE)` aufgerufen werden, um diese ebenfalls zu kopieren (*shallow copy* vs *deep copy*).

Referenzsemantik angewendet auf Argumente von Funktionen wird mit *Call-by-Reference* (im Gegensatz zu *Call-by-Value*) bezeichnet.

```
x$sum
## [1] 100
nil_sum <- function(acc) {
  acc$sum <- 0
}
nil_sum(x)
x$sum
## [1] 0
```

Aus der Perspektive der funktionalen Programmierung erzeugen wir damit starke Nebeneffekte von Funktionen, die dazu führen, dass ihre Wirkung schwer vorhersagbar werden.

Für weitere Fähigkeiten von R6 (Konstruktor, Vererbung, private Elemente, ...) siehe <https://adv-r.hadley.nz/r6.html>.

### 4.3 Interna

Wird eine Methode namens `print()` definiert, wird die Ausgabe auf der Konsole durch diese Methode vollzogen.

```
Accumulator <- R6Class("Accumulator", list(
  sum = 0,
  add = function(x = 1) self$sum <- self$sum + x,
  print = function() cat("Sum is", self$sum, "\n"))
)
x <- Accumulator$new()
x
## Sum is 0
```

Bemerkung: R6 nutzt hierbei S3 Funktionalität: `print(x)` ruft `print.R6(x)` auf. Dabei ist `print.R6` zwar nicht im aktuellen Suchpfad vorhanden, jedoch suchen generische S3-Funktionen nach Methoden auch in der Umgebung `__S3MethodsTable__`, die zum Base-Paket gehört.

```
library(rlang)
class(x)
## [1] "Accumulator" "R6"
x # ruft print(x) auf
## Sum is 0
# print(x) ruft print.R6(x) auf
```

```

print.R6 # not in search path
## Error in eval(expr, envir, enclos): object 'print.R6' not found
rlang:::base_env()$.__S3MethodsTable__.$print.R6
## function (x, ...)
## {
##   if (is.function(.subset2(x, "print"))) {
##     .subset2(x, "print")(...)
##   }
##   else {
##     cat(format(x, ...), sep = "\n")
##   }
##   invisible(x)
## }
## <bytecode: 0x000000001bd2f3a0>
## <environment: namespace:R6>
# .subset2(x,i) ist im Wesentlichen x[[i]]
# print.R6(x) ruft x[["print"]]() auf

```

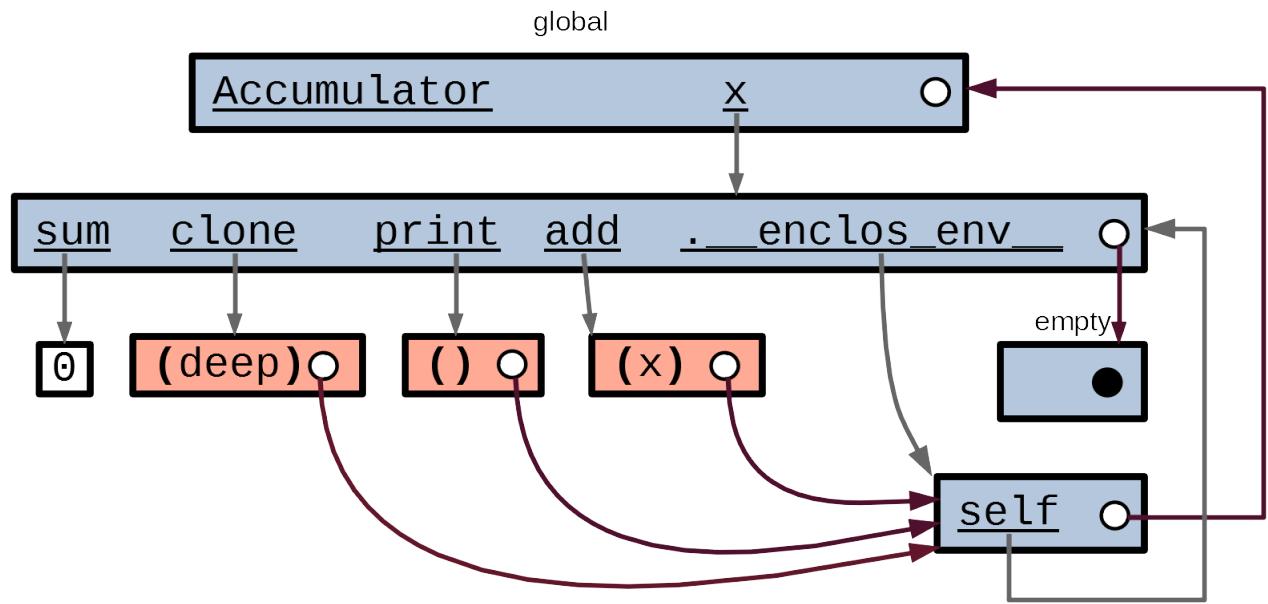
Wir untersuchen den inneren Aufbau eines R6-Objektes.

```

typeof(x)
## [1] "environment"
env_print(x)
## <environment: 000000001BBA67E0> [L]
## parent: <environment: empty>
## class: Accumulator, R6
## bindings:
##   * .__enclos_env__: <env>
##   * sum: <dbl>
##   * clone: <fn> [L]
##   * print: <fn> [L]
##   * add: <fn> [L]
env_print(x$.__enclos_env__)
## <environment: 000000001BBA6498>
## parent: <environment: global>
## bindings:
##   * self: <Accumltr>
identical(x$.__enclos_env__$self, x)
## [1] TRUE
identical(fn_env(x$add), x$.__enclos_env__)
## [1] TRUE

```

Der Zugriff auf Elemente des Objektes mittels `self` im Inneren von Methoden, lässt sich also durch eine geschickt gewählte Funktionsumgebung erklären.



# V12 - Meta-Programmierung

31. Mai 2021

## Contents

<b>1</b>	<b>Intro</b>	<b>2</b>
<b>2</b>	<b>Ausdrücke</b>	<b>2</b>
2.1	Bestandteile von Ausdrücken . . . . .	3
2.2	Syntaxbaum . . . . .	4
2.3	Ausdrücke und Strings . . . . .	5
2.4	Base-R . . . . .	6
<b>3</b>	<b>Ausdrücke verwenden</b>	<b>6</b>
3.1	Argumente . . . . .	7
3.2	Auswertung und Umgebungen . . . . .	9
3.3	Quosures . . . . .	9
<b>4</b>	<b>Anwendung</b>	<b>11</b>
4.1	Zugriff auf Argumenttext . . . . .	11
4.2	filter() . . . . .	11
4.3	source() . . . . .	12
<b>5</b>	<b>Tidy-Eval-Framework</b>	<b>12</b>
5.1	Forcing-Operatoren . . . . .	12
5.2	Pronomen . . . . .	14
5.3	Geschachtelte Quosures . . . . .	14
5.4	Anwendung im Tidyverse . . . . .	17

```
library(tidyverse)
## -- Attaching packages ----- tidyverse 1.3.1 --
## v ggplot2 3.3.3     v purrrr  0.3.4
## v tibble  3.1.2     v dplyr    1.0.6
## v tidyverse 1.1.3    v stringr  1.4.0
## v readr   1.4.0     vforcats  0.5.1
## -- Conflicts ----- tidyverse_conflicts() --
## x dplyr::filter() masks stats::filter()
## x dplyr::lag()   masks stats::lag()
library(rlang)
##
## Attaching package: 'rlang'
## The following objects are masked from 'package:purrr':
##
##     %%, as_function, flatten, flatten_chr, flatten_dbl, flatten_int,
##     flatten_lgl, flatten_raw, invoke, list_along, modify, prepend,
##     splice
library(lobstr)
```

## 1 Intro

Einige Funktionen haben ein sonderbares Verhalten:

Wie kann `library()` ohne " funktionieren?

```
rlang # keine Variable
## Error in eval(expr, envir, enclos): object 'rlang' not found
library(rlang) # funktioniert (ohne "")
```

Woher kennt `tibble()` den Variablenamen der Argumente?

```
numbers <- 1:26
tibble(numbers, letters)
## # A tibble: 26 x 2
##   numbers letters
##   <int> <chr>
## 1 1       a
## 2 2       b
## 3 3       c
## 4 4       d
## 5 5       e
## 6 6       f
## 7 7       g
## 8 8       h
## 9 9       i
## 10 10    j
## # ... with 16 more rows
```

Wie können `dplyr`-Funktionen Spalten wie Variablen verwenden?

```
tb <- tibble(x = 1:5, y = 5:1)
filter(tb, x > y)
## # A tibble: 2 x 2
##   x     y
##   <int> <int>
## 1 4     2
## 2 5     1
```

Antwort: **Meta-Programmierung!**

## 2 Ausdrücke

In der Meta-Programmierung wird R-Code selbst zu einem Objekt, das wir manipulieren und auswerten können.

Ein **Ausdruck** ist ein syntaktisch korrekter R-Funktionsaufruf, zB `3+5*4`, `x <- mean(1:100)`, `f(g(h(x)),y),z`. Dies ist unabhängig von Namensbindungen.

Um einen Ausdruck in einem R-Objekt zu speichern, verwenden wir `rlang::expr()`. `rlang::expr()` verhindert die sofortige Auswertung ihres Argumentes. Dieser Vorgang wird **Defusing** genannt.

```
x <- expr(3+5*4)
x
## 3 + 5 * 4
y <- expr(x <- mean(1:100))
y
## x <- mean(1:100)
```

```

z <- expr(f(g(h(x),y),z))
z
## f(g(h(x), y), z)

expr(f(x-)) # ERROR: nicht syntaktisch korrekt

```

Mit `rlang::exprs()` werden mehrere Ausdrücke in eine Liste von Ausdrücken aufgenommen. `c()` fügt mehrere Ausdrücke zu einer Liste zusammen.

```

str(exprs(1 + 2, f(x))) # zwei Ausdrücke
## List of 2
## $ : language 1 + 2
## $ : language f(x)
exprs(1 + 2; f(x)) # ein Ausdruck
## {
##   1 + 2
##   f(x)
## }

str(exprs(name1 = 1 + 2, name2 = f(x))) # auch als benannte Liste
## List of 2
## $ name1: language 1 + 2
## $ name2: language f(x)
c(expr(1 + 2), expr(f(x)))
## [[1]]
## 1 + 2
##
## [[2]]
## f(x)

```

`eval()` wertet Ausdruck-Objekte aus.

```

x <- expr(3+5*4)
y <- expr(x <- mean(1:100))
z <- expr(f(g(h(x),y),z))
eval(x)
## [1] 23
eval(y)
x
## [1] 50.5
eval(z)
## Error in f(g(h(x), y), z): could not find function "f"

```

## 2.1 Bestandteile von Ausdrücken

Ausdrücke setzen sich zusammen aus **Funktionsaufrufen**, **Konstanten** und **Symbolen**.

**Konstanten** sind Werte, für die keinerlei Namensauflösung vollzogen werden muss, zB `NULL`, `NA`, `FALSE`, `42L`, `5.5`, `"asdf"`, nicht jedoch `F` oder `pi`.

Für Konstanten ist `rlang::expr()` die Identität. Es wird also nicht unterschieden zwischen dem Ausdruck `1` und dem Wert `1`.

```

typeof(expr(1))
## [1] "double"
c(identical(FALSE, expr(FALSE)), identical(42L, expr(42L)), identical(F, expr(F)))
## [1] TRUE TRUE FALSE

```

Der Begriff **Symbol** bezeichnet einen Variablenamen. Da beim Erstellen eines Ausdrucks nur auf syntaktische Korrektheit geprüft wird, nicht jedoch auf Auswertbarkeit, können Symbole auch Namen sein, zu denen kein Wert gebunden ist.

Symbole haben Typ `symbol` und Klasse `name`.

```
x <- expr(a_variable_name)
c(typeof(x), class(x))
## [1] "symbol" "name"
```

Durch **Funktionsaufrufe** können Symbole, Konstanten und andere Funktionsaufrufe zu komplexeren Ausdrücken zusammengestellt werden.

Funktionsaufrufe haben Typ `language` und Klasse `call`.

```
a <- expr(f(g(h(x),y),z))
c(typeof(a), class(a))
## [1] "language" "call"
b <- expr(1 + 2)
c(typeof(b), class(b))
## [1] "language" "call"
```

## 2.2 Syntaxbaum

Jedem Ausdruck liegt eine Baumstruktur zugrunde, die angibt, welche Funktion mit welchen Argumenten ausgeführt werden soll. Sie wird als **Syntaxbaum** bezeichnet.

Der Syntaxbaum eines Ausdrucks ist ein gewurzelter Baum.

Die Wurzel ist die äußerste Funktion. In `f(g(x, h(), 1), x)` ist die Wurzel `f()`.

Die Kind-Knoten eines Knotens stehen für die Argumente der Funktion. Die Kinder des Knotens mit der Funktion `f()` sind `g(x, h(), 1)` und `x`.

Die Blätter eines Syntaxbaumes sind Symbole, Konstanten oder Funktionen ohne Argumente. Im Ausdruck `f(g(x, h(), 1), x)` sind die Blätter `x, h(), 1` und `x`.

Achtung: Die Reihenfolge der Kinder spielt eine Rolle ( $f(x,y) \neq f(y,x)$ ).

Die Funktion `lobstr::ast()` zeigt den Syntaxbaum eines Ausdrucks auf der Konsole an.

```
ast(f(g(x, h(), 1), x))
## o-f
## +-o-g
## / +-x
## / +-o-h
## / \-1
## \-x
```

`language`-Objekte sind im Wesentlichen verschachtelte Listen:

- Umwandlung zwischen `language` und `list` mit `as.call()` und `as.list()`
- Zugriff auf Elemente mit `[`, `[<-`, `[[`, `[[<-`
- Element an Position 1 ist Funktionsname, weitere Elemente sind Argumente

```
x <- expr(f(3+5,4))
x
## f(3 + 5, 4)
str(as.list(x))
## List of 3
## $ : symbol f
```

```

##  $ : language 3 + 5
##  $ : num 4
x[[1]] <- expr(g)
x
## g(3 + 5, 4)
x[c(1,3)]
## g(4)

a <- expr(f(g(x, h(), 1), x))
a[[1]]
## f
a[[2]]
## g(x, h(), 1)
a[[3]]
## x
a[[2]][[1]]
## g
a[[2]][[2]]
## x
a[[2]][[3]]
## h()
a[[2]][[4]]
## [1] 1
a[[2]][[3]][[1]]
## h
typeof(a[[2]][[3]]) # h()
## [1] "language"
typeof(a[[2]][[3]][[1]]) # h
## [1] "symbol"
# siehe auch View(a) in RStudio

```

Hier können wir Indexvektoren für [[ sinnvoll einsetzen. Erinnerung `x[[c(1,2,3)]]` entspricht `x[[1]][[2]][[3]]`.

```

a[[c(2, 3, 1)]]
## h

```

## 2.3 Ausdrücke und Strings

Um einen Ausdruck in seine Repräsentation als String zu wandeln, nutze `rlang::expr_text()`.

```

z <- expr(f(g(h(x),y),z))
z_str <- expr_text(z)
typeof(z_str)
## [1] "character"
z_str
## [1] "f(g(h(x), y), z)"

```

Um einen String, der syntaktisch korrekten R-Code enthält, in einen Ausdruck-Objekt zu konvertieren, nutze `rlang::parse_expr()`.

Enthält der String mehrere Befehle (durch ; oder Zeilenumbruch getrennt) kann mit der Funktion `rlang::parse_exprs()` eine Liste von Ausdrücken erzeugt werden.

```

y <- parse_expr("x <- 1+2")
typeof(y)

```

```

## [1] "language"
eval(y)
x
## [1] 3
z <- parse_exprs("x <- 4\nprint(x+5)")
str(z)
## List of 2
## $ : language x <- 4
## $ : language print(x + 5)

```

## 2.4 Base-R

In Base-R gibt es einen eigenen Datentyp für Listen von Ausdrücken. Dieser hat verwirrenderweise den Namen "expression".

`parse()` ist die Base-R-Variante von `rlang::parse_exprs()`.

```

typeof(parse(text="1+2;3-4"))
## [1] "expression"

```

`rlang` verwendet den Datentyp `expression` nicht. Stattdessen wird eine Liste (`list`) von Ausdrücken (`language`, `symbol` oder Konstante) verwendet.

```

# in rlang
b <- parse_exprs("1+2;3;a")
typeof(b)
## [1] "list"
str(b)
## List of 3
## $ : language 1 + 2
## $ : num 3
## $ : symbol a
b[[1]]
## 1 + 2

# in Base-R:
a <- parse(text="1+2;3;a")
typeof(a)
## [1] "expression"
str(a)
## expression(1 + 2, 3, a)
a[[1]]
## 1 + 2
typeof(a[[1]])
## [1] "language"

# Vergleich:
identical(a, b)
## [1] FALSE
mapply(identical, a, b)
## [1] TRUE TRUE TRUE

```

## 3 Ausdrücke verwenden

### 3.1 Argumente

Wir möchten nun aufnehmen, mit welchem Ausdruck ein Nutzer einer Funktion ein Argument übergibt, zB beim Aufruf von `f(1+2)` den Ausdruck `1+2` anstatt den Wert 3. Mit den bisher gelernten Funktionen schaffen wir es nicht.

```
f <- function(x) expr(x)
f(1 + 2)
## x
```

`rlang:::enexpr()` gibt den Ausdruck des übergebenen Arguments zurück.

```
h <- function(x) enexpr(x)
h(1 + 2)
## 1 + 2
```

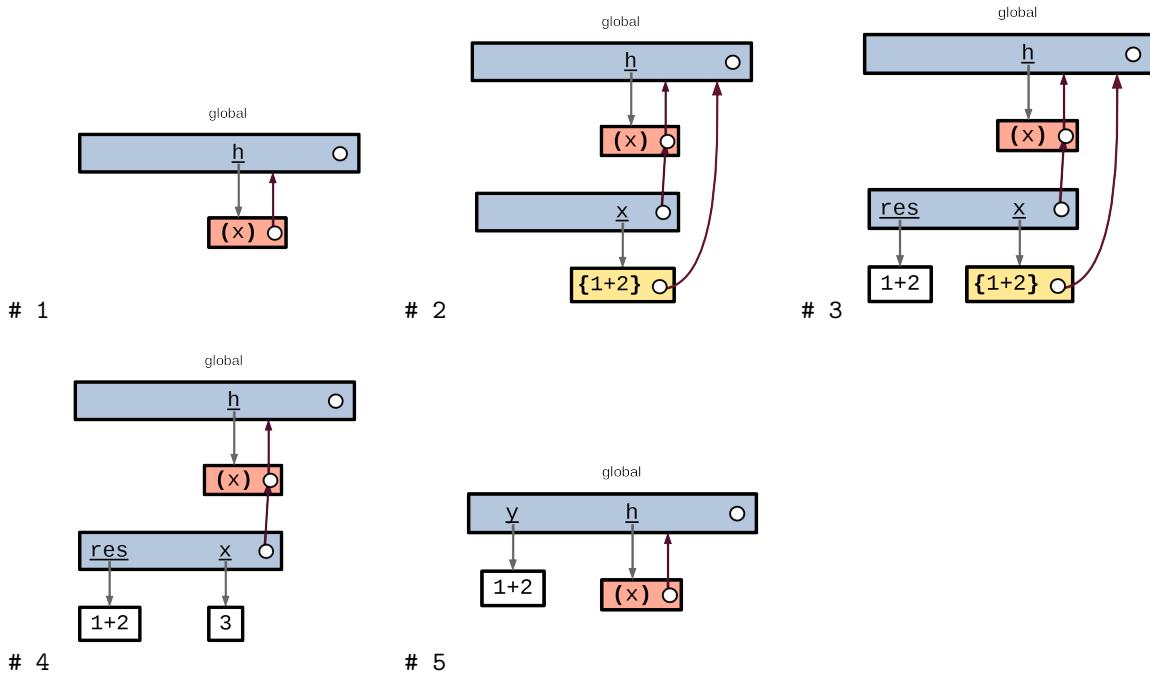
Erinnerung: Im Inneren einer Funktion sind die Argumente Versprechen, solange sie noch nicht ausgewertet worden sind. Ein Versprechen besteht aus einem Ausdruck und einer Umgebung, in der der Ausdruck ausgewertet werden soll. Mit `rlang:::enexpr()` erhalten wir den Ausdruck ohne das Versprechen einzulösen.

```
h <- function(x) {
  env_print(current_env()) # <lazy> zeigt Versprechen an
  res <- enexpr(x)
  env_print(current_env())
  force(x)
  env_print(current_env())
  res
}
y <- h(1 + 2)
## <environment: 0000000020C3F050>
## parent: <environment: global>
## bindings:
##   * x: <lazy>
##     <environment: 0000000020C3F050>
##       parent: <environment: global>
##         bindings:
##           * res: <language>
##             * x: <lazy>
##               <environment: 0000000020C3F050>
##                 parent: <environment: global>
##                   bindings:
##                     * res: <language>
##                       * x: <dbl>
y
## 1 + 2
```

Der gleiche Code mit Umgebungsdiagrammen:

```
h <- function(x) {
  # 2
  res <- enexpr(x)
  # 3
  force(x)
  # 4
  res
}
# 1
y <- h(1 + 2)
```

```
# 5
y
## 1 + 2
```



Wir müssen `rlang::enexpr()` aufrufen, bevor das Versprechen eingelöst wird. `rlang::enexpr()` selbst löst das Versprechen nicht ein.

```
h <- function(x) {
  print(enexpr(x))
  print(enexpr(x))
  force(x)
  print(enexpr(x))
}
h(1+2)
## 1 + 2
## 1 + 2
## [1] 3
```

Um die Elemente des `...`-Arguments als Liste von Ausdrücken aufzunehmen, kann `rlang::enexprs()` genutzt werden.

```
f <- function(...) enexprs(...)
str(f(x="asdf", a, z=5+3))
## List of 3
## $ x: chr "asdf"
## $ a: symbol a
## $ z: language 5 + 3
```

Als Anwendungsbeispiel geben wir die Ausdrücke der Argumente als Text aus.

```
direct_paste <- function(...) {
  dots <- enexprs(...)
  str_c(sapply(dots, expr_text), collapse = " ")
}
```

```
direct_paste(direkte, Ausgabe)
## [1] "direkte Ausgabe"
```

## 3.2 Auswertung und Umgebungen

Die Auswertung eines Ausdrucks mit `eval()` geschieht in der aktuellen Umgebung, außer wir übergeben als zweites Argument eine andere Umgebung.

```
x <- 5
x_expr <- expr(x * 2)
eval(x_expr)
## [1] 10
eval(x_expr, env(x = 4))
## [1] 8

y <- 3
eval(
  expr(x + y),
  env(`+` = `+`))
)
## [1] 2
```

## 3.3 Quosures

Da zu der Auswertung eines Ausdrucks also immer auch eine Umgebung gehört, in der der Ausdruck ausgeführt wird, stellt das Paket `rlang` die Datenstruktur *Quosure* bereit, die beides zusammenhält.

Ein *Quosure* ist ein S3-Objekt basierend auf einem Ausdruck-Objekt mit einem Attribut `.Environment`, welches die Umgebung speichert.

`quo()` und `quos()` sind analog zu `expr()` und `exprs()` und nehmen die aktuelle Umgebung mit auf.

```
q <- quo(1+x)
q
## <quosure>
## expr: ~1 + x
## env: global
typeof(q)
## [1] "language"
attributes(q)
## $class
## [1] "quosure" "formula"
##
## $.Environment
## <environment: R_GlobalEnv>
```

Mit `quo_get_expr()` und `quo_get_env()` wird auf den Ausdruck bzw die Umgebung eines Quosures zugegriffen.

`eval_tidy(q)` für eine Quosure `q` entspricht `eval(quo_get_expr(q), quo_get_env(q))`.

```
f <- function() {
  x <- 10
  quo(1+x)
}
x <- 100
q <- f()
```

```
eval(quo_get_expr(q))
## [1] 101
eval_tidy(q)
## [1] 11
```

`enquo()` und `enquos()` sind analog zu `enexpr()` und `enexprs()` und nehmen die Umgebung auf, in der die Argumente ausgewertet werden. Sie wandeln also ein Versprechen (mit Ausdruck und Umgebung) in ein Quosure.

```
foo <- function(x) {
  y <- 1
  x <- enexpr(x)
  eval(x)
}
z <- 10
foo(z * 2)
## [1] 20
y <- 100
foo(y * 2)
## [1] 2

foo2 <- function(x) {
  y <- 1
  x <- enexpr(x)
  eval(x, caller_env())
}
y <- 100
foo2(y * 2)
## [1] 200

foo3 <- function(x) {
  y <- 1
  x <- enquo(x)
  eval_tidy(x)
}
y <- 100
foo3(y * 2)
## [1] 200
```

`eval_tidy()` erlaubt es sogenannte **Data-Masks** zu verwenden: Wird `eval_tidy()` als zweites Argument eine Liste benannter Elemente übergeben, wird bei der Auswertung des Ausdrucks zuerst dort nach Namen gesucht und erst danach in der Umgebung des Quosures.

```
x <- 1
y <- 10
eval_tidy(quo(x+y), list(y=100))
## [1] 101
```

Auf dieser Mechanik basieren viele Funktionen im Tidyverse, bei denen sich Spalten in Tibbles wie Variablen verwenden lassen.

```
tb <- tibble(x=1:3, y=11:13)
z <- 100
x <- 1e4
eval_tidy(quo(x+y+z), tb)
## [1] 112 114 116
```

## 4 Anwendung

### 4.1 Zugriff auf Argumenttext

Eine Fragestellung zu Beginn des Kapitels war, wie `tibble()` Spaltennamen automatisch aus dem Text Argumente erstellen kann.

```
numbers <- 1:26
tibble(numbers, letters)
## # A tibble: 26 x 2
##   numbers letters
##   <int> <chr>
## 1 1     a
## 2 2     b
## 3 3     c
## 4 4     d
## 5 5     e
## 6 6     f
## 7 7     g
## 8 8     h
## 9 9     i
## 10 10   j
## # ... with 16 more rows
```

Wir können nun selbst eine Funktion mit dieser Fähigkeit schreiben.

```
capture_named <- function(...) {
  arg_list <- enquo(...)
  values <- lapply(arg_list, eval_tidy)
  arg_text <- sapply(arg_list, quo_text)
  names(values) <- make.names(arg_text) # make syntactically valid names
  return(values)
}
str(capture_named(numbers, letters, 1:3))
## List of 3
## $ numbers: int [1:26] 1 2 3 4 5 6 7 8 9 10 ...
## $ letters: chr [1:26] "a" "b" "c" "d" ...
## $ X1.3  : int [1:3] 1 2 3
```

Dies klärt auch, wie `library(rlang)` ohne " funktionieren kann.

### 4.2 filter()

In `dplyr::filter()` können Spalten wie Variablen gebraucht werden.

```
tb <- tibble(x = 1:5, y = 5:1)
filter(tb, x > y)
## # A tibble: 2 x 2
##   x     y
##   <int> <int>
## 1 4     2
## 2 5     1
```

Wir erstellen eine vereinfachte Implementierung dieser Mechanik.

```
filter2 <- function(tb, fltr) {
  fltr <- enquo(fltr)
  rows <- eval_tidy(fltr, tb)
```

```

    tb[rows, ]
}
filter2(tb, x > y)
## # A tibble: 2 x 2
##       x     y
##   <int> <int>
## 1     4     2
## 2     5     1

```

### 4.3 source()

Mit der Funktion `source()` lassen sich andere Skript-Dateien in der aktuellen Umgebung ausführen.

Mit `rlang::parse_exprs()` können wir eine ähnliche Funktion schreiben.

```

source2 <- function(path) {
  file <- str_c(readLines(path, warn = FALSE), collapse = "\n")
  exprs <- parse_exprs(file)
  res <- NULL
  for (i in seq_along(exprs)) {
    res <- eval(exprs[[i]], caller_env())
  }
  res
}

```

## 5 Tidy-Eval-Framework

Einige Tidyverse-Funktionen nutzen **Non-Standard-Evaluation** (NSE), zB Tibble-Spalten wie Variablen benutzen.

Die NSE im Tidyverse basiert auf dem **Tidy-Eval-Framework**, welches von `rlang` bereitgestellt wird. Elemente dieses Frameworks sind Quosures, `eval_tidy()`, Forcing-Operatoren und Pronomen.

### 5.1 Forcing-Operatoren

Die Forcing-Operatoren sind:

- `!!` Bang-Bang
- `!!!` Big-Bang
- `{()}` Curly-Curly (hier nicht besprochen)
- `:=` Walross (*walrus*)

Sie können nur in Argumenten von Funktionen genutzt werden, die das Tidy-Eval-Framework unterstützen. Dazu gehören einige Funktionen aus `rlang`, `lobstr` und dem Tidyverse, nicht jedoch Base-R-Funktionen.

```

a <- expr(f(x))
!!a # nicht ok, nicht in Argument
## Error in !a: invalid argument type
!!TRUE # doppelte Negation
## [1] TRUE
expr(1+!!!a) # hier ok
## 1 + f(x)

```

Der **Bang-Bang**-Operator `!!` wertet den nachfolgenden Teilausdruck aus.

```
x <- 4
expr(1 + !(2+3) + x + !x)
## 1 + 5 + x + 4
```

`lobstr::ast()` interpretiert ihr Argument als Ausdruck. Mit `!!` zeigen wir den Syntaxbaum einer Ausdruck-Variablen an.

```
a <- expr(f(y))
ast(a)
## a
ast(!!a)
## o-f
## \-y
ast(g(z, !!a))
## o-g
## +-z
## \-o-f
##   \-y
```

Der Operator `!!!` (**Big-Bang**) wertet zuerst den nachfolgenden Ausdruck aus; ist dieser ein Vektor, werden die Elemente als Liste aufgenommen.

```
x <- 1:3
str(exprs(!!x))
## List of 1
## $ : int [1:3] 1 2 3
str(exprs(!!!x))
## List of 3
## $ : int 1
## $ : int 2
## $ : int 3
```

Um einen Namen aus einer Variable als Argumentnamen zu verwenden, ersetzen wir `=` durch `:=` (**Walross**) und wenden `!!` auf die Variable an. Dabei verhält sich `:=` wie `=`, aber nur `:=` erlaubt `!!` auf der linken Seite.

```
name <- "asdf"
tibble(name = 1+2)
## # A tibble: 1 x 1
##   name
##   <dbl>
## 1     3
# tibble(!name = 1+2) # ERROR
tibble(name := 1+2)
## # A tibble: 1 x 1
##   name
##   <dbl>
## 1     3
tibble(!!name := 1+2)
## # A tibble: 1 x 1
##   asdf
##   <dbl>
## 1     3
```

## 5.2 Pronomen

**Pronomen** (*pronouns*) sind die Namen `.data` und `.env`. Sie haben im Tidy-Eval-Framework eine besondere Bedeutung.

Sie können in Ausdrücken wie Tibbles behandelt werden. `.data` enthält die Namen der Data-Mask, `.env` die Umgebungsvariablen.

```
y <- 4
tb <- tibble(x = 1:3, y = 3:1)
eval_tidy(quo(y + y*10), tb)
## [1] 33 22 11
eval_tidy(quo(.data$y + .env$y*10), tb)
## [1] 43 42 41
```

## 5.3 Geschachtelte Quosures

`!!` wirkt auf Teil-Ausdrücke, die zu Quosures evaluieren, in besonderer Weise.

```
x <- 1
qua <- function(y) 10*y
q <- function() quo(x+2)
v <- function() qua(x+2)
identical(eval_tidy(quo(!!v())), v())
## [1] TRUE
identical(eval_tidy(quo(!!q())), q())
## [1] FALSE
identical(eval_tidy(quo(!!q())), x+2)
## [1] TRUE
```

`!!` gliedert den Inhalt eines Quosures in das äußere Quosure ein, wodurch ein viel tieferer Syntaxbaum entstehen kann. Bei der Eingliederung eines anderen Objektes mit `!!` entsteht nur ein Blatt im Syntaxbaum.

```
ast(7+!!v())
## o-`+` 
## +-7
## \-30
ast(7+!!q())
## o-`+` 
## +-7
## \-o-`+` 
##   +-x
##   \-2
```

Quosures können Teil-Quosures mit unterschiedlichen Umgebungen haben. `eval_tidy()` wertet jedes Quosure in seiner Umgebung aus.

```
y <- 1
create_q <- function() {
  y <- 10
  quo(y)
}
create_qu <- function(x) {
  y <- 100
  quo(y + !!x)
}
qu <- create_qu(create_q())
qu # y with different environments
```

```

## <quosure>
## expr: ^y + (^y)
## env: 0000000020A89000
eval_tidy(qu)
## [1] 110
eval_tidy(qu, list(y = 1e3))
## [1] 2000

# vergleiche mit:
quo(y + y) # y from same environment
## <quosure>
## expr: ^y + y
## env: global
eval_tidy(quo(y + y))
## [1] 2
eval_tidy(quo(y + y), list(y = 1e3))
## [1] 2000

```

Beachte die Ausgabe der Quosures als  $\hat{y} + (\hat{y})$  bzw  $\hat{y} + y$ . Das Symbol  $\hat{\cdot}$  markiert den Beginn eines neuen Teil-Quosures, welches eine eigene Umgebung hat.

Im folgenden Beispiel wollen wir `tidy_eval(XXX == 1, tb)` auswerten, wobei wir `XXX` durch einen Namen einer Spalte von `tb` ersetzen. Dieser Name soll jedoch nicht fest vorgegeben, sondern in einer Variable gespeichert sein.

```

y <- 4
tb <- tibble(x = 1:5, y = 5:1)

# verschiedene Ansätze Namen zu speichern
var <- y # normale Variable
q <- quo(y) # Quosure

# Versuch 1
qu <- quo(var == 1)
qu
## <quosure>
## expr: ^var == 1
## env: global
eval_tidy(qu, tb) # nein
## [1] FALSE

# Versuch 2
qu <- quo(q == 1)
qu
## <quosure>
## expr: ^q == 1
## env: global
eval_tidy(qu, tb) # nein
## Error: Base operators are not defined for quosures.
## Do you need to unquote the quosure?
##
##     # Bad:
##     myquosure == rhs
##
##     # Good:

```

```
##    ! !myquosure == rhs

# Versuch 3
qu <- quo(!!q == 1)
qu
## <quosure>
## expr: ^(^y) == 1
## env: global
eval_tidy(qu, tb) # ja :
## [1] FALSE FALSE FALSE FALSE  TRUE
```

Bemerkung: Ist die Variable als String gespeichert, können wir sie in ein Quosure umwandeln und dann wie oben verfahren. Einfacher geht es jedoch mit dem Pronomen `.data`.

```
str <- "y" # String

qu <- quo(!!parse_quo(str, current_env()) == 1)
qu
## <quosure>
## expr: ^(^y) == 1
## env: global
eval_tidy(qu, tb)
## [1] FALSE FALSE FALSE FALSE  TRUE

qu <- quo(.data[[str]] == 1)
qu
## <quosure>
## expr: ^.data[["y"]] == 1
## env: global
eval_tidy(qu, tb)
## [1] FALSE FALSE FALSE FALSE  TRUE
```

Der große Vorteil der Nutzung von Quosures anstatt Strings ist die Möglichkeit beliebige, syntaktisch korrekte Ausdrücke als Argument zu geben.

```
q <- quo(y %% 2)
qu <- quo(!!q == 1)
qu
## <quosure>
## expr: ^(^y %% 2) == 1
## env: global
eval_tidy(qu, tb)
## [1]  TRUE FALSE  TRUE FALSE  TRUE
```

Jetzt setzen wir obiges Vorgehen analog mit Funktionsargumenten um. Dabei ersetzen wir `quo()` durch `enquo()`. Mit dieser Technik schreiben wir eine Funktion, die `filter2(tb, XXX == 1)` aufruft, wobei `XXX` ein Ausdruck ist, der Spalten aus `tb` wie Variablen benutzt.

```
y <- 4
tb <- tibble(x = 1:5, y = 5:1)
filter2 <- function(tb, fltr) {
  qu <- enquo(fltr)
  rows <- eval_tidy(qu, tb)
  tb[rows, ]}
```

```

# Versuch 1
filter2x1 <- function(var) {
  filter2(tb, var == 1)
}
filter2x1(y) # nein
## # A tibble: 0 x 2
## # ... with 2 variables: x <int>, y <int>

# Versuch 2
filter2x2 <- function(var) {
  q <- enquo(var)
  filter2(tb, q == 1)
}
filter2x2(y) # nein
## Error: Base operators are not defined for quoSures.
## Do you need to unquote the quoSure?
##
## # Bad:
## myquoSure == rhs
##
## # Good:
## !!myquoSure == rhs

# Versuch 3
filter2x3 <- function(var) {
  q <- enquo(var)
  filter2(tb, !!q == 1)
}
filter2x3(y) # ja :)
## # A tibble: 1 x 2
##       x     y
##   <int> <int>
## 1     5     1
filter2x3(y %% 2)
## # A tibble: 3 x 2
##       x     y
##   <int> <int>
## 1     1     5
## 2     3     3
## 3     5     1

```

## 5.4 Anwendung im Tidyverse

Argumente von Funktionen im Tidyverse, bei denen Spalten wie Variablen benutzt werden können, unterstützen das Tidy-Eval-Framework.

Im Folgenden zeigen wir ein paar Anwendungsbeispiele

### Grouping-Variable als Argument

```

df <- tibble(
  g1 = c(1, 1, 2, 2, 2),
  g2 = c(1, 2, 1, 2, 1),
  a = sample(5),
  b = sample(5)

```

```

)

# Versuch 1
my_summarise <- function(df, group_var) {
  df %>%
    group_by(group_var) %>%
    summarise(a = mean(a))
}
my_summarise(df, g1) # nein
## Error: Must group by variables found in `data`.
## * Column `group_var` is not found.

# Versuch 2
my_summarise <- function(df, group_var) {
  q <- enquo(group_var)
  df %>%
    group_by (!!q) %>%
    summarise(a = mean(a))
}
my_summarise(df, g1) # ja :)
## # A tibble: 2 x 2
##       g1     a
##   <dbl> <dbl>
## 1     1     4
## 2     2  2.33

```

## Zusammenzfassende Variable als Argument

```

my_summarise2 <- function(df, expr) {
  q <- enquo(expr)
  summarise(df,
    mean = mean (!!q),
    sum = sum (!!q),
    n = n()
  )
}
my_summarise2(df, a)
## # A tibble: 1 x 3
##       mean     sum     n
##   <dbl> <int> <int>
## 1     3     15     5
my_summarise2(df, a * b)
## # A tibble: 1 x 3
##       mean     sum     n
##   <dbl> <int> <int>
## 1    8.6     43     5

```

## Name neuer Spalten aus Variable

quo\_name() wandelt ein Quosure in den Text ihres Ausdruck um.

```

my_mutate <- function(df, expr) {
  q <- enquo(expr)
  mean_name <- str_c("mean_", quo_name(q))
  sum_name <- str_c("sum_", quo_name(q))

```

```

  mutate(df,
    !!mean_name := mean(!!q),
    !!sum_name := sum(!!q)
  )
}

my_mutate(df, a)
## # A tibble: 5 x 6
##       g1     g2     a     b mean_a sum_a
##   <dbl> <dbl> <int> <int>  <dbl> <int>
## 1     1     1     5     4     3     15
## 2     1     2     3     2     3     15
## 3     2     1     4     1     3     15
## 4     2     2     1     3     3     15
## 5     2     1     2     5     3     15

```

### Beliebige Anzahl an Grouping-Variablen

```

my_summarise <- function(df, ...) {
  qs <- enquos(...)
  df %>%
    group_by(!!!qs) %>%
    summarise(a = mean(a))
}

my_summarise(df, g1, g2)
## `summarise()` has grouped output by 'g1'. You can override using the `groups` argument.
## # A tibble: 4 x 3
## # Groups:   g1 [2]
##       g1     g2     a
##   <dbl> <dbl> <dbl>
## 1     1     1     5
## 2     1     2     3
## 3     2     1     3
## 4     2     2     1

```