

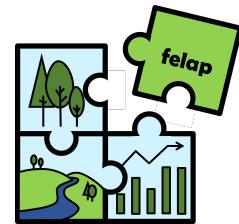
1

Einführung in die Datenanalyse mit R (700104)

4

Kursskript

5



6

Dr. Johannes SIGNER
Abteilung Wildtierwissenschaften
Büsgenweg 3
37077 Göttingen

jsigner@uni-goettingen.de

Dr. Kai HUSMANN
Abteilung Forstökonomie und nachhaltige Landnutzungsplanung
Büsgenweg 1
37077 Göttingen

kai.husmann@uni-goettingen.de

7



8

Fakultät für Forstwissenschaften und Waldökologie
Georg-August-Universität Göttingen

10



11

12

Wintersemester 2023/2024

¹³ Dieses Werk ist lizenziert unter einer [Creative Commons Namensnennung - Nicht-kommerziell - Weitergabe unter gleichen Bedingungen 4.0 International Lizenz](#).

¹⁵ Zitiervorschlag:

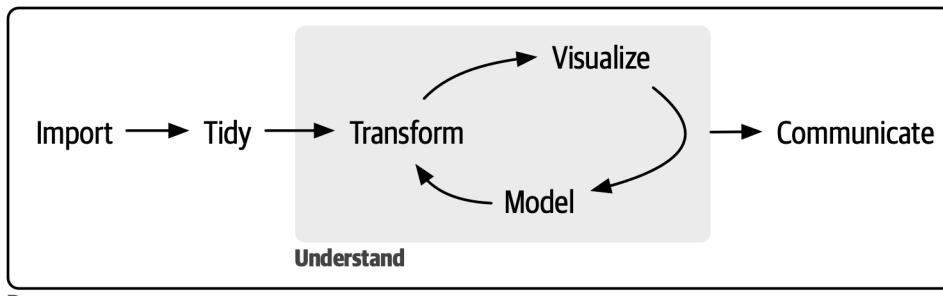
¹⁶ Signer, J. und Husmann, K. (2024) Skript zur Vorlesung Einführung in die Datenanalyse mit R, Georg-August-Universität Göttingen.

¹⁸ Letzte Aktualisierung: 6. Februar 2024

19 Vorwort und Danksagung

20

- 21 Lernziel des Kurses ist die Einführung in die Arbeit, Visualisierung und Analyse von (forstlichen) Daten-
22 sätzen mit dem Statistikprogramm *R*. Der Schwerpunkt liegt dabei auf der Datenverarbeitung. Statistische
23 Methoden werden nur an wenigen Stellen exemplarisch angewendet. Ein typisches Data Science Projekt
24 besteht laut Wickham et al. (<https://r4ds.hadley.nz/>) aus 4 Stufen.



25 **Program**

- 26 Wir werden uns in diesem Kurs insbesondere mit den ersten beiden Stufen *Import* und *Tidy* beschäftigen
27 und uns im Schritt *Understand* nur mit sehr einfachen *Models* befassen.
- 28 Weitere Materialien als dieses Kursskript und die Übungsaufgaben (StudIP) werden nicht benötigt. Die
29 gelöste Übungsaufgaben dieses Skriptes werden Ihnen über StudIP zugänglich gemacht. Dort werden auch
30 ggf. Ankündigungen bekanntgegeben. Damit Sie Credits für diesen Kurs zu erhalten, müssen Sie am Ende des
31 Kurses eine mündliche Prüfung ablegen. Für die Prüfung werden Sie zwei zufällig gezogene Prüfungsfragen
32 aus dem Dokument "Übungen: Einführung in die Datenanalyse mit R" bearbeiten und vorstellen. Nach einer
33 15-minütigen Vorbereitungszeit beträgt die Prüfungszeit weitere 15 Minuten. In der Prüfungszeit präsentieren
34 Sie zunächst Ihre Lösung und beantworten anschließend vertiefende Fragen zu Ihrer Lösung und daraufhin
35 auch zum gesamten Lehrinhalt des Kurses.
- 36 Dieses Vorlesungsskript ist ein R Markdown-Dokument, das mit R und RStudio erstellt wurde. Das Do-
37 kument besteht aus Fließtext, R Code und den entsprechenden Code-Ergebnissen. Die grau hinterlegten
38 Codepassagen sind kurze R-Skripte. Falls das Skript eine Konsoleausgabe erzeugt, ist diese direkt mit
39 "##" markiert (diese Begriffe werden in Kapitel 1.2 näher erläutert).
- 40 Dank für Anmerkungen gilt Markus Benesch, Sofie Biberacher und Josephine Trisl. Teile des Unterkapitels
41 zu Schleifen und Kontrollstrukturen sind an das R Skript des Kurses Computergestützte Datenanalyse von
42 Robert Nuske, Nikolas von Lüpke und Joachim Saborowski angelehnt. Des Weiteren wurden Beispiele aus
43 dem frei Verfügbaren Dokument *R for Data Science* (<https://r4ds.hadley.nz/intro.html>) entnommen.

44 Inhaltsverzeichnis

45	1 R und RStudio	4
46	1.1 Installation von R und RStudio	4
47	1.2 Erste Schritte in R	4
48	1.3 Gute Praxis bei der Programmierung	6
49	2 Variablen, Funktionen und Datentypen	8
50	2.1 Variablen beim Programmieren	8
51	2.2 Funktionen	10
52	2.3 Datentypen	11
53	2.4 Datenstrukturen	12
54	3 Vektoren	14
55	3.1 Funktionen zum Arbeiten mit Vektoren	16
56	3.2 Statistische Funktionen	18
57	3.3 Beispiel Fotofallen	19
58	3.4 Arbeiten mit logischen Werten	20
59	3.5 Zugreifen auf Elemente eines Vektors (=Untermengen)	22
60	3.6 Der %in%-Operator	24
61	4 Faktoren (factors)	26
62	4.1 Das Paket forcats	28
63	4.1.1 Anpassen der Anordnung von Faktoren	28
64	5 Spezielle Einträge	30
65	5.1 NA	30
66	5.2 NULL	31
67	5.3 Inf	31
68	6 data.frames oder Tabellen	34
69	6.1 Wichtige Funktionen zum Arbeiten mit data.frames	35
70	6.2 Zugreifen auf Elemente eines data.frame	36

71	7 Schreiben und lesen von Daten	40
72	7.1 Textdateien	40
73	8 Erstellen von Abbildungen	42
74	8.1 Base Plot	42
75	8.1.1 Mehrere Panels	48
76	8.1.2 Speichern von Abbildungen	49
77	8.2 Histogramme	50
78	8.3 Boxplots	52
79	8.4 ggplot2: Eine Alternative für Abbildungen	54
80	8.4.1 Multipanel Abbildungen	63
81	8.4.2 Plots kombinieren	66
82	8.4.3 Speichern von plots	70
83	9 Mit Daten arbeiten	71
84	9.1 dplyr eine Einführung	71
85	9.2 Arbeiten mit gruppierten Daten	74
86	9.3 pipes oder %>%	76
87	9.4 Joins	77
88	9.5 ‘long’ and ‘wide’ Datenformate	79
89	9.6 Auswählen von Variablen	81
90	9.7 Einzelne Beobachtungen abfragen (slice())	83
91	9.8 Spalten trennen	86
92	10 Arbeiten mit Text	88
93	10.1 Arbeiten mit Text	88
94	10.2 Finden von Textmustern	90
95	11 Arbeiten mit Zeit	93
96	11.1 Arbeiten mit Zeitintervallen	95
97	11.2 Formatieren von Zeit	96
98	11.3 Zeitreihen	97

99	12 Aufgaben Wiederholen (for-Schleifen)	104
100	12.1 Schleifen	104
101	12.1.1 Wiederholen von Befehlen mit <code>for()</code>	105
102	12.1.2 Wiederholen von Befehlen mit <code>while()</code>	107
103	12.2 Bedingte Ausführung von Codeblöcken	108
104	13 (R)markdown	110
105	13.1 Markdown Grundlagen	110
106	13.2 R und Markdown	111
107	14 Räumliche Daten in R	113
108	14.1 Was sind räumliche Daten	113
109	14.2 Koordinatenbezugssystem	113
110	14.3 Vektordaten in R	114
111	14.4 Arbeiten mit Vektordaten	115
112	14.5 Rasterdaten in R	117
113	15 FAQs (Oft gefragtes)	124
114	15.1 Arbeiten mit Daten	124
115	15.1.1 Einlesen von Exceldateien	124
116	16 Literatur	125

1 R und RStudio

1.1 Installation von R und RStudio

Als ersten Schritt müssen Sie R und RStudio installieren. Dabei ist wichtig zu unterscheiden, dass R und RStudio zwei unterschiedliche Programme sind. R ist die eigentliche Programmiersprache mit der wir arbeiten. RStudio hingegen ist eine sogenannte Entwicklungsumgebung¹, die das Arbeiten mit R vereinfachen soll.

- Sie können also mit R arbeiten ohne RStudio (was unüblich ist), aber nicht mit RStudio ohne R.
- Gehen Sie für die Installation von R, auf die Website <https://cloud.r-project.org/> und laden Sie die für ihren Computer passende R-Version herunter und folgen Sie den Installationsanweisungen. In Linux können Sie R über die Kommandozeile installieren.
- Für die Installation von RStudio gehen Sie zu der Website <https://posit.co/download/rstudio-desktop/#download> und laden die richtige Version für Ihr Betriebssystem herunter und folgen Sie den Installationsanweisungen.

1.2 Erste Schritte in R

RStudio bietet eine Vielzahl von Funktionen, die uns das Arbeiten mit R erleichtern können. Öffnen Sie RStudio. Sie erhalten eine leere Entwicklungsumgebung. Als erstes bietet es sich an, ein neues Skript zu erstellen. Gehen Sie dafür auf das Menü: [File] > [New File] > [R Script] oder klicken Sie die Tastenkombination *Strg + Umschalt + N* (**Strg** + **↑** + **N**).

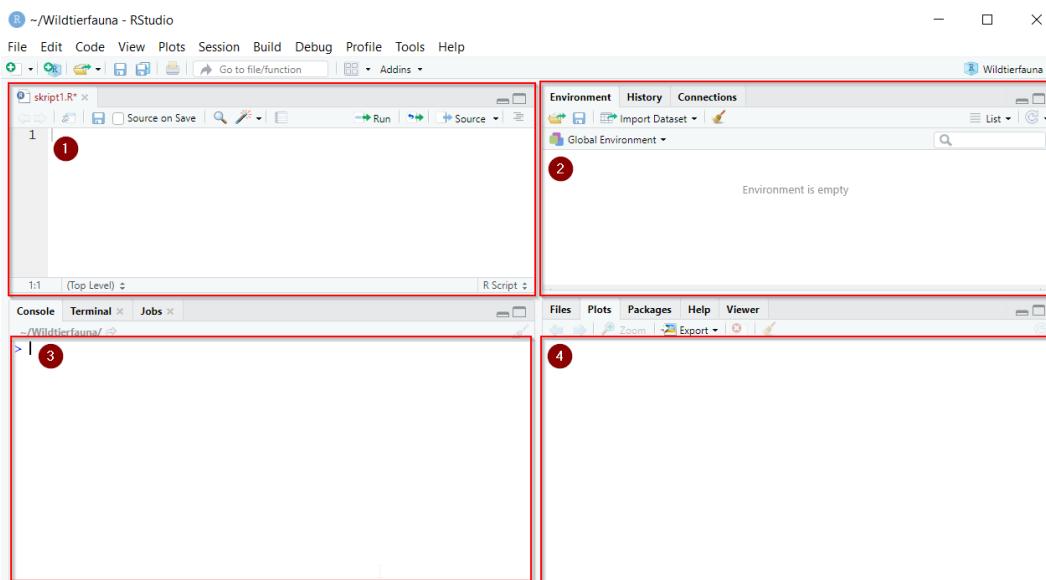


Abbildung 1: RStudio Panes.

¹Oder auch IDE (=Integrated Development Environment) genannt.

135 RStudio besteht nun aus vier sogenannten **Panes** oder Ausschnitten (siehe auch Abbildung 1). Die Aus-
136 schnitte sind wie folgt gegliedert:

- 137 1. Hier werden Skripte anzeigen, d.h., hier wird meist R Code geschrieben und dokumentiert. Der Code wird
138 beim Schreiben noch nicht ausgeführt (siehe Punkt 3), dennoch sollten Sie Code immer so schreiben,
139 dass er Zeile für Zeile abgeschickt werden kann. Sie sollten nie Code schreiben, bei dem Sie zwischen
140 den Zeilen hin und her springen müssen.
 - 141 2. Der zweite Ausschnitt erteilt Auskunft über den *Workspace*. Im Workspace werden alle verfügbaren
142 Objekte angezeigt.
 - 143 3. Die eigentliche R-Konsole wird in Ausschnitt 3 dargestellt. Hier wird in der Regel wenig Code eingege-
144 ben. Der normale Workflow ist vom Skript Code an die Konsole zu schicken. Erst durch das Abschicken
145 in die Konsole wird der Code (bzw. die Teile des Codes, die Sie abschicken) ausgeführt.
 - 146 4. Der vierte Ausschnitt enthält mehrere Reiter. Der Reiter *Files* zeigt den Verzeichnisbaum an. Im Reiter
147 *Plots* werden Plots angezeigt, wenn diese im Code aufgerufen werden. Hilfeseiten zu Funktionen werden
148 im Reiter *Help* angezeigt.
- 149 Einfache Rechenoperationen können auch direkt in der R-Konsole durchgeführt werden. Prinzipiell könnten
150 Sie alle Operationen direkt in die Konsole tippen. Der Nachteil und der Grund, warum dies keine gute Praxis
151 ist, ist, dass der Code zwar ausgeführt jedoch nicht gespeichert wird. Code der nicht als Skript gespeichert
152 wird, ist also nicht dokumentiert. Tippen Sie die folgenden Operationen in die Konsole.

10 + 5

153 `## [1] 15`

20 - 10

154 `## [1] 10`

10 * 3

155 `## [1] 30`

100 / 19

156 `## [1] 5.263158`

157 Sie sehen Ihren Code in rot und das Ergebnis dieser Operation in weiß darunter. Die Zahl in [] gibt die
158 Dimension des Ergebnisses an. Hier also [1] für eine Angabe. Dieses Skript wurde in R Markdown geschrieben
159 (siehe Vorwort). R Markdown verbindet Text und Code. Die Ergebnisse des Codes werden unter dem grau
160 hinterlegten *Codechunk* dargestellt. Darstellung und Farbe des Codes und der Ergebnisse sind jedoch nicht
161 immer exakt so wie sie es in der R Konsole wären.

162 Weitere häufig verwendete Operationen sind \wedge für eine beliebige Potenz, z.B. $2^3 = 2\wedge3 = 8$. Analog dazu
 163 gibt es die Funktion `sqrt()` zum berechnen von Wurzeln und viele weitere Funktionen. Wenn Sie einen
 164 code abschicken, der nicht funktioniertm bekommen Sie statt des Ergebnisses eine Fehlermeldung, welche
 165 bestenfalls einen Hinweis zur Korrektur enthält.

166 Meist verwenden wir jedoch **Skripte**, um den R-Code zu schreiben und ihn dann an die Konsole “zu schi-
 167 cken”. Dies hat den Vorteil, dass alle Schritte nachvollziehbar bleiben und Analysen beliebig oft wiederholt
 168 werden können. Nach der Ausführung bleibt der Code erhalten und Sie dokumentieren Ihre Berechnungen
 169 automatisch mit. Stellen Sie sich den Code im R-Skript wie ein Kochbuch vor. Wenn wir R-Code in einem
 170 R-Skript geschrieben haben gibt es mehrere Möglichkeiten diesen Code abzuschicken/ auszuführen. Wir kön-
 171 nen eine Zeile abschicken, indem wir entweder auf *Run* klicken (Abbildung 2) oder die Tastenkombination
 172 *Strg + Enter* (`Strg`+`↵`) tippen. Mehrere Zeilen abzuschicken und nacheinander ausführen zu lassen ist
 173 möglich, indem diese Zeilen markiert werden bevor Sie *Run* klicken oder die Tastenkombination tippen. Ein
 174 Klick auf *Source* bzw. die Tastenkombination *Strg + Umschalt + Enter* (`Strg`+`⇧`+`↵`).

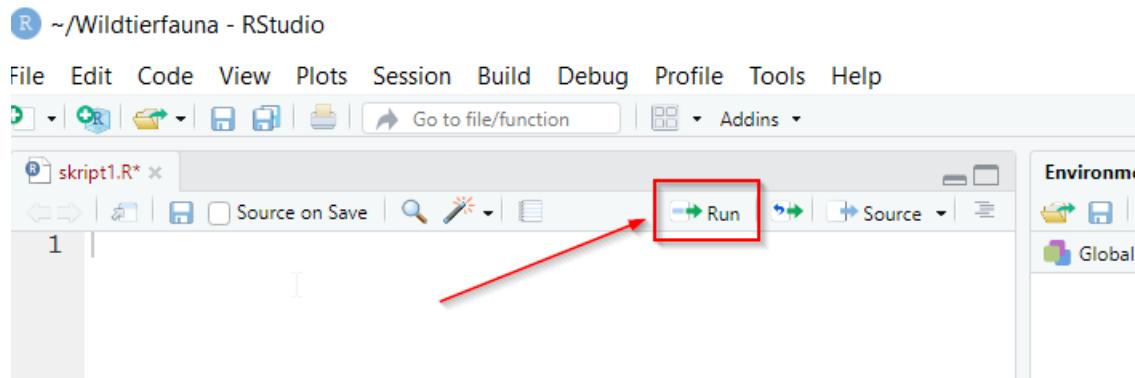


Abbildung 2: Zeilenweises Ausführen von Code in RStudio.

175 Wenn Sie die Codezeile abgeschickt haben, sehen Sie diese Zeile in der Konsole und direkt darunter das
 176 Ergebnis sowie die Dimension des Ergebnisses, also genauso, als hätten Sie den Code direkt in die Konsole
 177 getippt. Die Konsole erkennt, wenn Sie einen unvollständigen Code abschicken. In der Konsole sehen Sie in
 178 diesem Fall kein Ergebnis, sondern ein + unter der abgeschickten Zeile. Sie können nun eine weitere Zeile zur
 179 vervollständigung abschicken oder in der Konsole *Escape* (`Esc`) drücken, um abzubrechen.

180 1.3 Gute Praxis bei der Programmierung

181 Es gibt eine gute Praxis, wie der Stil der Codes sein sollte. Die sog. *Style Guides* sind eine Art generelle
 182 Vereinbarung bei der Programmierung. Style Guides sind selbstverständlich optional und wer viel program-
 183 miert, wird mit der Zeit evtl. einen eigenen Stil entwickeln. Dennoch bieten Style Guides gerade beim Einstieg
 184 in die Programmierung eine gute Orientierungshilfe und erleichtern vor allem das Arbeiten im Team. Der
 185 wichtigste und umfangreichste Style Guide für R ist <https://style.tidyverse.org/index.html>. Wir empfehlen,
 186 die Kapitel *Welcome*, *Files* und *Syntax* zu lesen, bevor Sie mit dem Programmieren beginnen. Ein weiterer
 187 berühmter Style Guide ist von Google <https://google.github.io/styleguide/>.

188 Vielleicht die wichtigste Praxis beim Programmieren ist das Kommentieren. **Kommentare** sind ein wich-
 189 tiger Bestandteil der Skripte in R und allen anderen Skript- und Programmiersprachen. Sie werden lernen,

190 dass die Kommentarfunktion ein wesentlicher Vorteil gegenüber Klickprogrammen darstellt. Ein Kommentar
191 ist Text in einem (R-)Code, der der Dokumentation dient und von der R Konsole ignoriert wird. Sämtliche
192 Zeilen, die mit dem Zeichen # beginnen, werden nicht ausgeführt, wenn Sie an die Konsole gesendet wer-
193 den. Seien Sie nicht sparsam mit Kommentaren, sondern benutzen Sie sie um Ihren Code zu strukturieren,
194 ihre Berechnungen zu für sich selbst und andere zu erläutern und, um Ergebnisse zu beschreiben oder zu
195 interpretieren.

```
# sqrt(a)
# Berechnen der Quadratwurzel
sqrt(81)
```

196 ## [1] 9

197 Sie können Kommentare auch verwenden, um Code, den sie später vielleicht wieder aktivieren wollen, aus-
198 zukommentieren. Im vorherigen Codeblock wurde der Funktionsaufruf `sqrt(a)` auskommentiert. Die Zeile #
199 Berechnen der Quadratwurzel wird bei der Ausführung ebenfalls ignoriert. Es empfiehlt sich, komplexere
200 Abläufe zu kommentieren, damit andere im Team verstehen, warum und wie etwas gemacht wurde. Ganz
201 besonders gilt das jedoch für einen selbst. Oft sind in der Zukunft Zusammenhänge nicht mehr so klar, wie
202 sie beim Schreiben des Codes waren.

203

204 Aufgabe 1: Ausführen von Quellcodes

206 Öffnen Sie RStudio, erstellen Sie ein neues Skript und speichern Sie dieses unter dem Namen `skript1.R` ab.

207 Tippen oder kopieren Sie folgenden Code in das Skript:

```
# Einfache Rechenoperationen
1 + 3
2^7

# Einfache Funktion
sqrt(20)
```

208 Führen Sie nun alle Zeilen aus.

209 2 Variablen, Funktionen und Datentypen

210 2.1 Variablen beim Programmieren

211 Ergebnisse aus Berechnungen (wie oben angeführt), aber auch z. B. aus komplexeren Operationen, werden
 212 in Variablen abgespeichert. Man kann sich eine Variable wie eine Hülle (oder bildlich gesprochen wie eine
 213 Schachtel) vorstellen, in die man etwas hinein legen kann und darauf zu einem späteren Zeitpunkt wieder
 214 zugreifen kann. Z. B. weist der folgende Ausdruck der Variable `alter` den Wert 102 zu.

```
alter <- 102
```

215 Variablen können Objekte in R speichern. Ein Objekt, im einfachsten Fall ein einzelner Wert, kann mit der
 216 Anweisung `<-` einer Variablen zugewiesen werden. Der nachfolgende Code weist der Variable `a` den Wert 10
 217 zu.

```
a <- 10
a
```

218 ## [1] 10

219 Man kann mit `=` oder `<-` einer Variable einen Wert zuweisen. Der Unterschied ist in den meisten Fällen
 220 vernachlässigbar, es wird aber allgemein empfohlen `<-` (= ist schlechter Stil) zu verwenden.
 221 Wir können beliebige Variablen erstellen, z.B.

```
abc <- 10
name <- "Johannes"
```

222 Variablennamen dürfen nicht mit einer Zahl beginnen und müssen aus einem Wort bestehen. Die Variablen
 223 erscheinen nach der Definition im *Environment* Tab in Pane 2.

- 224 • `a_123 <- 10` ist ok
- 225 • `123_a <- 10` erzeugt einen Fehler

226 Vorsicht: Groß- und Kleinschreibung muss beachtet werden.

```
name <- "Johannes"
name
```

227 ## [1] "Johannes"

228 Das Aufrufen der Variable

Name

229 führt zu einem Fehler.

230 Wir können dann mit den Werten, die in Variablen gespeichert sind, ganz normale Rechenoperationen durchführen.

```
a <- 10  
b <- 5  
  
a + b
```

232 ## [1] 15

```
b / a
```

233 ## [1] 0.5

```
a^b
```

234 ## [1] 1e+05

235 Das Ergebnis kann natürlich wieder in einer neuen Variable gespeichert werden.

```
ergebnis <- a + b  
ergebnis
```

236 ## [1] 15

```
ergebnis2 <- ergebnis * 2  
ergebnis2
```

237 ## [1] 30

238 Mit der Funktion `rm()` können Variablen, die nicht mehr benötigte Variablen, wieder gelöscht werden.
239 Alternativ können Variablen auch überschrieben werden. Es gibt keine Möglichkeit gelöschte oder überschriebene Variablen wiederherzustellen. Sie müssen ggf. neu berechnet werden.

```
var1 <- "irgendwas"  
exists("var1") # TRUE. also ja, eine Variable mit diesem Namen existiert
```

241 ## [1] TRUE

```
rm(var1)
exists("var1") # FALSE, also nein, eine Variable mit diesem Namen existiert nicht.
```

242 ## [1] FALSE

2.2 Funktionen

244 Ein zweiter wesentlicher Bestandteil beim Arbeiten mit R sind Funktionen. Während eine Variable etwas
 245 speichert, tut eine Funktion etwas. Beispielsweise zieht die Funktion `sqrt()` die Quadratwurzel aus einer
 246 Zahl.

```
sqrt(a)
```

247 ## [1] 3.162278

248 Funktionen sind in R in der Regel daran zu erkennen, dass sie immer mit dem Funktionsnamen, gefolgt
 249 von runden Klammern (), aufgerufen werden. Im vorherigen Beispiel wurde die Funktion mit dem Namen
 250 `sqrt()` aufgerufen. Das Objekt `a` haben wir bereits vorhin definiert (zur Erinnerung `a <- 10`). Die Funktion
 251 `sqrt()` arbeitet jetzt mit dem Objekt `a`, das in diesem Zusammenhang auch **Argument** genannt wird.

252 Argumente von Funktionen haben Namen, diese müssen jedoch nicht angegeben werden, wenn die Reihen-
 253 folge der Argumente berücksichtigt wird. Im vorherigen Beispiel, haben wir einfach die Funktion `sqrt(a)`
 254 aufgerufen und keinen Argumentnamen angegeben. Aus den Hilfeseiten für die Funktion `sqrt()` (siehe auch
 255 nachfolgender Abschnitt) ist zu entnehmen, dass die Funktion `sqrt()` ein Argument mit dem Namen `x` hat.
 256 Das heißt, der vollständige Aufruf der Funktion `x` wäre.

```
sqrt(x = a)
```

257 ## [1] 3.162278

258 Um mehr über eine Funktion zu erfahren (z.B. die Bedeutung von Argumenten zu verstehen oder heraus-
 259 zufinden, was eine Funktion zurück gibt), können wir die Hilfeseiten konsultieren. Es gibt unterschiedliche
 260 Wege, um zu einer Hilfeseite zu gelangen.

- 261 1. In die Konsole `?<Name der Funktion>` tippen. Also, wenn wir Hilfe für die Funktion `mean()` möchten,
 262 könnten wir einfach `?mean` in die Konsole tippen.
- 263 2. Analog zu 1), können wir mit der Funktion `help` die Hilfeseite für eine andere Funktion aufrufen (z.B.
 264 wenn wir wieder die Hilfe für die Funktion `mean()` lesen möchten, dann könnten wir auch `help(mean)`
 265 in die Konsole tippen).
- 266 3. In RStudio kann man auch auf das Help-Panel klicken und dann einfach eine Funktion suchen (siehe
 267 Abbildung 1).
- 268 4. Wenn der Cursor in RStudio auf einer Funktion ist, kann man mit der Taste F1 die dazugehörige
 269 Hilfeseite aufrufen.

2.3 Datentypen

- 270 Es wurde bereits erwähnt, dass Daten in Variablen gespeichert werden können. Die Variablen, in denen die
 271 Daten (oder auch Berechnungen oder ganze Funktionen) gespeichert werden, heißen in R Objekte. Wenn
 272 Sie beispielsweise Messwerte einer Fotofalle speichern möchten, dann hat diese Fotofalle einen Namen (z.B.
 273 Kamera1) und nach einiger Zeit im Wald wurden hoffentlich auch einige Fotos aufgenommen. Wir nehmen
 274 einmal an, dass nach drei Wochen 132 Fotos von Rehen gemacht wurden.
 275
- 276 Wir können jetzt sowohl den Namen der Fotofalle, als auch die Anzahl Fotos die aufgenommen wurden, in
 277 zwei Variablen abspeichern.

```
kamera_name <- "Kamera_1"
anzahl_rehe <- 132
```

- 278 In den zwei vorherigen Zeilen Code haben wir zwei Objekte (im Sinne von R) erstellt. Das erste Objekt heißt
 279 `kamera_name` und das zweite Objekt heißt `anzahl_rehe`. In dem Beispiel handelt es sich also um zwei sehr
 280 einfache Objekte, in denen jeweils ein Wert gespeichert ist. Auffällig ist, dass beide Objekte unterschiedliche
 281 Datentypen haben. `kamera_name` ist vom Typ `character` (also Text). Das zweite Objekt, `anzahl_rehe`, ist
 282 vom Typ `numeric` (also eine Zahl, wir unterscheiden hier nicht weiter²). Zusätzlich zu diesen zwei Typen
 283 (`character` und `numeric`), gibt es noch einen weiteren wichtigen Typ: nämlich das logische Wahr oder
 284 Falsch (in R: `TRUE` und `FALSE`) und noch weitere Datentypen auf die wir zunächst nicht eingehen (tippen Sie
 285 `?typeof` für eine Übersicht aller Datentypen). Zurückkommend auf das Beispiel mit den Fotofallen, könnte
 286 eine mögliche Fragestellung sein, ob auf einem der Fotos ein Fuchs gesehen wurde oder nicht. Dazu würden
 287 wir eine neue Variable `fuchs_gesehen` anlegen und diese auf `TRUE` setzen, da ein Fuchs gesehen wurde.

```
fuchs_gesehen <- TRUE
```

- 288 Wenn Sie sich nicht sicher, um welchen Typen es sich handelt, können sie ihn mit `str` abfragen.

```
typeof(fuchs_gesehen)
```

289 ## [1] "logical"

- 290 `TRUE` wird intern als 1 gespeichert und `FALSE` als 0. Es ist möglich mit `TRUEs` und `FALSEs` zu rechnen.

```
TRUE + TRUE
```

291 ## [1] 2

```
FALSE + FALSE
```

292 ## [1] 0

²Für Interessierte, man unterscheidet weiter zwischen Ganzzahlen (`int`) und Gleitkommazahlen (`double`) unterscheiden.

TRUE + FALSE

293 `## [1] 1`

294 2.4 Datenstrukturen

295 Verfolgen wir das Beispiel mit den Fotofallen etwas weiter. Es handelt sich um ein systematisches Monitoring.
 296 D. h., es wurde nicht nur eine Fotofalle ausgebracht, sondern insgesamt 15 Stück. Dieser Umstand erfordert
 297 komplexere Objekte. Nachfolgend sind die Anzahl Rehfotos für jede der 15 Fotofallen aufgeführt: 132, 79,
 298 129, 91, 138, 144, 55, 103, 139, 105, 96, 146, 95, 118, 107.

299 Die Frage, die sich jetzt stellt, ist: Wie kann man diese Daten sinnvoll organisieren? Zusätzlich zur Anzahl
 300 der fotografierten Rehe soll jede Fotofalle eine eindeutige ID haben (`Kamera_1`, ..., `Kamera_15`) und wir
 301 wissen, dass jeweils 5 Fotofallen in drei unterschiedlichen Revieren aufgestellt waren (Fotofalle 1 bis 5 in
 302 Revier A, Fotofalle 6 bis 10 in Revier B und Fotofalle 11 bis 15 in Revier C). Wir könnten für jede Kamera
 303 und jeden Wert ein einzelnes Objekt mit je einem Wert erstellen. Dass das zu Aufwändig wäre, leuchtet
 304 unmittelbar ein:

```
# 1. Kamera
name1 <- "Kamera_1"
anzahl_rehe1 <- 132
revier_1 <- "Revier A"

# 2. Kamera
name2 <- "Kamera_2"
anzahl_rehe2 <- 79
revier_2 <- "Revier A"

# usw.
```

305 Wenn wir so vorgehen würden, hätten wir 45 Objekte mit je einem Eintrag. Dieser Ansatz und führt schnell
 306 zu einem unübersichtlichen *Workspace*³. Wir werden im Verlauf sinnvollere Objekte (Vektoren oder Data
 307 Frames) für diesen Zweck kennenlernen.

308

309 **Aufgabe 2: Variablen**

311 Verwenden Sie die folgenden Daten

³Als *Workspace* werden alle Objekte bezeichnet, die in einer R-Session zur Verfügung stehen. Siehe Liste im *Environment* in Pane 2.

```
a <- 2
b <- "100"
p <- FALSE
```

312 und berechnen sie:

- 313 • $10 * a$
- 314 • $a / 144$ und speichern Sie das Ergebnis in einer neuen Variablen e zwischen.
- 315 • Was ist das Ergebnis von $a + b$?
- 316 • Was ist das Ergebnis von $a + p$?

```
10 * a
e <- a / 144
a + b
a + p
```

3 Vektoren

317 Die gute Nachricht zuerst. Sie haben bereits Vektoren erstellt in R (und dies wahrscheinlich nicht bewusst wahrgenommen). Wenn Sie nämlich eine Objekt mit einem Element erstellen (z.B., `a <- 10`), wird ein Vektor der Länge eins erstellt. Das heißt, der Vektor enthält genau ein Element (einen Eintrag). Vektoren sind also kein neues Objekt für Sie, sondern Sie lernen jetzt, dass die Ihnen schon bekannten Objekte Vektoren heißen und sie auch mehrere Elemente in eine mObjekt speichern können.

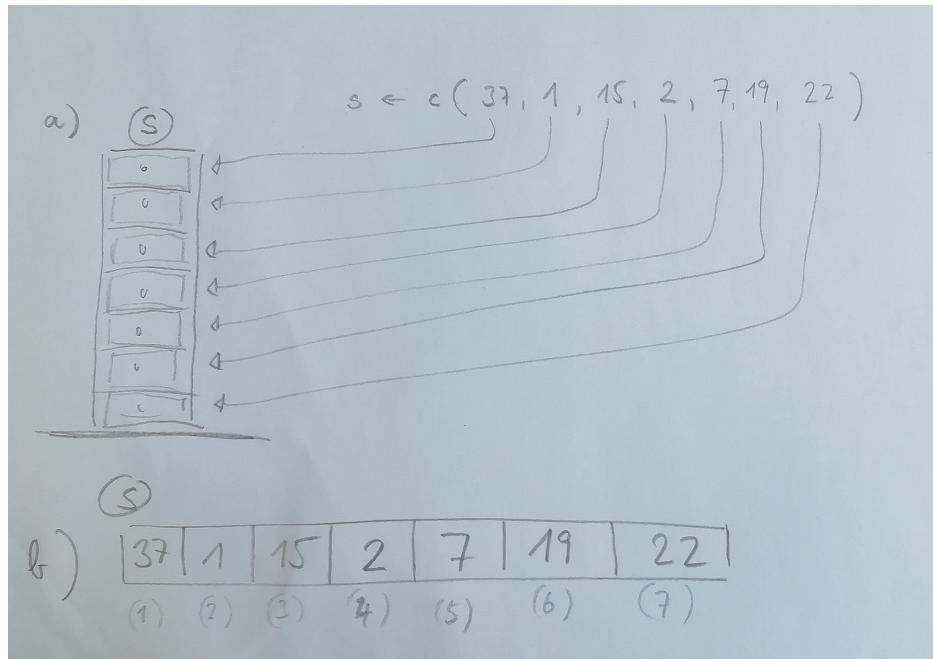


Abbildung 3: Schematische Darstellung eines Vektors in R.

323 Sie können sich Vektoren wie einen Schubladenschrank vorstellen (siehe auch Abbildung 3). Wichtig ist dabei, 324 dass man in jede Schublade immer nur ein Element verstauen kann und alle Elemente im Schublagenschrank 325 den gleichen Datentyp haben müssen. Etwas allgemeiner gesprochen heißt das, dass alle Elemente eines 326 Vektors vom gleichen Datentyp sein müssen.
327 Es gibt zahlreiche Funktionen zum Erstellen von Vektoren (einige davon werden wir im weiteren Verlauf des 328 Moduls kennenlernen). Die wohl wichtigste Funktion ist `c()`. Dabei steht `c` für *combine* oder *concatenate*. 329 Die Funktion `c()` fügt einzelne Elemente in einen Vektor zusammen (und zwar genau in der Reihenfolge wie 330 diese Elemente an `c()` übergeben werden). `c()` ist sozusagen die Funktion, die für Sie mehrere Elemente zu 331 einem Vektor zusammensetzt. `c()` erwartet folglich nur Elemente als Argumente.
332 Gehen wir nochmals zurück zu Abbildung 3, in der schematisch dargestellt wird, wie ein Vektor `s` mit 7 333 Elementen (in diesem Fall Zahlen) erstellt wird.

```
s <- c(37, 1, 15, 2, 7, 19, 22)
```

334 Die Funktion `c()` ordnet jetzt bildlich gesprochen die Zahl 37 der ersten Schublade zu, die Zahl 1 der zweiten 335 Schublade und so weiter. Wenn Sie jetzt einfach `s` in die Konsole tippen, können Sie alle Elemente von `s`

336 sehen:

```
s
```

337 ## [1] 37 1 15 2 7 19 22

338 In Abbildung 3b wird der Vektor `s` nochmals systematisch dargestellt. Dabei sieht man z. B., dass 37 an der
339 ersten Position des Vektors gespeichert wird und 22 an der letzten Position des Vektors gespeichert wird.

340 Die Grundrechenarten (+, -, /, *) und viele andere Funktionen funktionieren genau gleich mit Vektoren
341 deren Länge > 1 ist. Sie werden elementweise durchgeführt. Wir können beispielsweise zu jedem Element
342 von `s` 10 addieren

```
s + 10
```

343 ## [1] 47 11 25 12 17 29 32

344 oder `s` mit sich selbst multiplizieren. Zu beachten ist also, dass es sich bei Vektorberechnungen in R zunächst
345 nicht um Vektorrechnungen handelt, wie Sie sie aus der linearen Algebra kennen. Für die sog. Matrizenope-
346 rationen der linearen Algebra werden die Operatoren in R mit % % umschlossen, also bspw. `s %*% s`.

```
s * s
```

347 ## [1] 1369 1 225 4 49 361 484

348 Neben der Funktion `c()` gibt es zahlreiche weitere Funktionen, um Vektoren zu erstellen. Sehr häufig braucht
349 man Vektoren von Zahlenfolgen. Solche Vektoren können mit der Funktion `seq()` erstellt werden. Im ein-
350 fachsten Fall benötigt `seq()` zwei Argumente: `from` und `to`⁴.

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

351 ## [1] 1 2 3 4 5 6 7 8 9 10

```
(1 : 10)
```

352 ## [1] 1 2 3 4 5 6 7 8 9 10

353 Man kann dann auch noch die Schritte angeben, mit denen erhöht wird.

```
seq(from = 1, to = 10, by = 2)
```

354 ## [1] 1 3 5 7 9

⁴Weil solche Vektoren so häufig vorkommen gibt es hier eine Abkürzung. Man kann `seq(from, to, by = 1)` mit `from:to` abkürzen. Also `1:10` würde auch alle Zahlen von 1 bis 10 zurückgeben.

355

Aufgabe 3: Vektoren erstellen

358 Sie haben den BHD (Brusthöhendurchmesser) in cm von vier Bäumen gemessen: 13, 15.3, 23, 9

- 359 • Erstellen Sie einen Vektor mit dem Namen `bhd` in dem Sie die Werte speichern
360 • Transformieren sie die BHD-Werte in mm.
361 • Berechnen Sie die Fläche des BHD in cm^2 (nehmen Sie dafür an, dass ein Baum kreisrund ist).

362 3.1 Funktionen zum Arbeiten mit Vektoren363 Die Funktionen `head()` und `tail()` geben die ersten bzw. letzten `n` Elemente eines Vektors zurück. `n` hat
364 einen voreingestellten Wert von 6, dieser kann natürlich angepasst werden.`head(s)`365 `## [1] 37 1 15 2 7 19``head(s, n = 3)`366 `## [1] 37 1 15``tail(s, n = 2)`367 `## [1] 19 22`368 Die Funktion `length()` gibt die Länge eines Vektors wieder.`length(s)`369 `## [1] 7`370 Der Typ der Elemente eines Vektors kann mit der Funktion `class` abgefragt werden:`class(s)`371 `## [1] "numeric"`372 Die eindeutigen Elemente eines Vektors können mit der Funktion `unique()` abgefragt werden.

unique(s)

```
373 ## [1] 37 1 15 2 7 19 22
```

374 Mit der Funktion **table** kann die Häufigkeit verschiedener Elemente abgefragt werden.

table(s)

```
375 ## s
376 ## 1 2 7 15 19 22 37
377 ## 1 1 1 1 1 1
```

378 Schlussendlich kann man mit der Funktion **sort()** und **rev()** die Position von Elementen in einem Vektor
379 ändern. Die Funktion **rev** dreht die Elemente einmal um

rev(s)

```
380 ## [1] 22 19 7 2 15 1 37
```

381 während **sort()** einen Vektor nach seinen Elementen sortiert⁵.

sort(s)

```
382 ## [1] 1 2 7 15 19 22 37
```

383 Die Funktion **rep()** wiederholt einen Vektor.

rep(s, times = 2)

```
384 ## [1] 37 1 15 2 7 19 22 37 1 15 2 7 19 22
```

385 Anstelle des Arguments **times** kann auch das Argument **each** verwendet werden. Der Unterschied liegt darin,
386 dass **times** den gesamten Vektor **times**-Mal wiederholt und **each** jedes Element.

```
a <- 1:4
rep(a, times = 2)
```

```
387 ## [1] 1 2 3 4 1 2 3 4
```

⁵Auch für **sort()** gibt es ein zusätzliches Argument, das es ermöglicht die Elemente in absteigender Reihenfolge zu sortieren. Schauen Sie sich dazu, und auch für weitere Funktionen, die Hilfeseiten an.

```
rep(a, each = 2)  
  
388 ## [1] 1 1 2 2 3 3 4 4
```

389

390 **Aufgabe 4: Arbeiten mit Vektoren**
391

392 Es liegen jeweils zwei BHD-Messungen von vier Bäumen vor:

```
bhd <- c(32, 33, 23, 21, 21, 27, 18, 12)
```

393 Sie haben jeden Baum je ein Mal mit dem Messgerät G1, dann mit dem Messgerät G2 gemessen. Erstellen
394 Sie einen Vektor von der Länge 8, in dem Sie angeben, welches Messgerät Sie verwendet haben.

395 **3.2 Statistische Funktionen**

396 Zahlreiche statistische Funktionen können auf Vektoren angewendet werden, hier sind nur die wichtigsten
397 aufgeführt: `mean()` berechnet den Mittelwert, `median()` berechnet den Median und `sd()` die Standardab-
398 weichung.

```
mean(s)
```

399 ## [1] 14.71429

```
median(s)
```

400 ## [1] 15

```
sd(s)
```

401 ## [1] 12.76341

402 Eine weitere, sehr häufig verwendete Funktion ist `sample()`. Mit `sample()` werden `size` Elemente zufällig
403 aus einem Vektor, mit oder ohne Zurücklegen (mit Zurücklegen wird gezogen, wenn das Argument `replace`
404 = TRUE gesetzt wird), gezogen.

```
sample(s, size = 1) # 1 Element
```

405 ## [1] 1

```
sample(s, size = 3) # 2 Elemente
406 ## [1] 15 7 22
```

407 Wenn `size` weggelassen wird, dann bekommt man gleich viele Elemente zurück (wie der Vektor lang ist),
 408 d.h. der Vektor wird nur permutiert.

409 3.3 Beispiel Fotofallen

410 Für den weiteren Verlauf wollen wir noch einmal zu dem Beispiel mit den Fotofallen zurückkommen. Wir
 411 können jetzt 3 Vektoren erstellen, jeweils einen für die ID, die Anzahl Rehfotos und das Revier. Dabei werden
 412 zwei weitere Funktionen eingeführt (`paste` und `rep`).

413 Als erstes erstellen wir einen Vektor mit den Anzahlen Rehfotos. Das geht einfach mit `c()`:

```
anzahl_rehe <- c(132, 79, 129, 91, 138, 144, 55, 103, 139,
105, 96, 146, 95, 118, 1007)
```

414 Als zweites erstellen wir einen Vektor mit den IDs. Zur Erinnerung, diese sollten die Werte `Kamera_1` bis
 415 `Kamera_15` haben. Ein erster Ansatz könnte sein, dass wir einfach 15 Fotofallen schreiben und dann die
 416 Zahlen 1 bis 15 dahinter.

```
ids <- c("Kamera_1", "Kamera_2", "Kamera_3", "Kamera_4", "Kamera_5",
       "Kamera_6", "Kamera_7", "Kamera_8", "Kamera_9", "Kamera_10",
       "Kamera_11", "Kamera_12", "Kamera_13", "Kamera_14", "Kamera_15")
)
```

417 Dieser Ansatz ist unbefriedigend, da wir 15 mal das Wort “Kamera” tippen müssen. Wir können das Problem
 418 in zwei kleinere Probleme zerlegen: 1) 15 mal das Wort Kamera erstellen und die Zahlen 1 bis 15 erstellen,
 419 2) die zwei Vektoren aus 1) “zusammenkleben”.

420 Ein Vektor kann mit der Funktion `rep` wiederholt werden, das heißt wir können ganz einfach 15 mal das
 421 Wort “Kamera” erstellen und speichern das Zwischenergebnis in einem Vektor `v1`.

```
v1 <- rep("Kamera", 15)
```

422 Im nächsten Schritt müssen wir die Zahlen 1 bis 15 erstellen, auch dieses Zwischenergebnis speichern wir in
 423 einem neuen Vektor `v2`.

```
v2 <- 1:15
```

424 Jetzt müssen wir lediglich die Vektoren `v1` und `v2` “zusammenkleben”. Dafür gibt es die Funktion `paste`,
 425 die zwei Vektoren elementweise verbindet, dabei wird das Argument `sep` als Trennzeichen verwendet. In
 426 unserem Fall wäre das also.

```

426 ids <- paste(v1, v2, sep = "_")
427 ids
428
429 ## [1] "Kamera_1"  "Kamera_2"  "Kamera_3"  "Kamera_4"  "Kamera_5"  "Kamera_6"
430 ## [7] "Kamera_7"  "Kamera_8"  "Kamera_9"  "Kamera_10" "Kamera_11" "Kamera_12"
431 ## [13] "Kamera_13" "Kamera_14" "Kamera_15"

```

430 Mehr Funktionen zum Umgang mit Zeichenketten folgen in Kapitel “Arbeiten mit Text”. Dann fehlt jetzt
431 lediglich der Vektor mit den Revieren. Hier könnten wir erneut auf die Funktion `rep` zurückgreifen.

```
rep(c("Revier A", "Revier B", "Revier C"), 5)
```

```

432 ## [1] "Revier A" "Revier B" "Revier C" "Revier A" "Revier B" "Revier C"
433 ## [7] "Revier A" "Revier B" "Revier C" "Revier A" "Revier B" "Revier C"
434 ## [13] "Revier A" "Revier B" "Revier C"

```

435 Das Ergebnis stimmt noch nicht ganz, da wir 5 mal `Revier A` usw. brauchen. Mit dem zusätzlichen Argument
436 `each = 5` können wir genau zu diesem Ergebnis kommen.

```

reviere <- rep(c("Revier A", "Revier B", "Revier C"), each = 5)
reviere

```

```

437 ## [1] "Revier A" "Revier A" "Revier A" "Revier A" "Revier A" "Revier B"
438 ## [7] "Revier B" "Revier B" "Revier B" "Revier B" "Revier C" "Revier C"
439 ## [13] "Revier C" "Revier C" "Revier C"

```

440

441 **Aufgabe 5: Statistische Funktionen**

443 1. Berechnen Sie den Mittelwert und Median für die Anzahl Fotos.

444 2. Erstellen Sie die folgende Konsolenausgabe:

```
445 ## [1] "Die mittlere Anzahl von Rehfotos beträgt 171.8 Rehe pro Standort."
```

446 **3.4 Arbeiten mit logischen Werten**

447 Weniger bekannt sind die sogenannte booleschen Rechenregeln, also das Rechnen mit wahr (`TRUE`) und falsch
448 (`FALSE`). Dabei werden die folgenden Operationen am häufigsten verwendet.

- 449 • Gleichheit (`==`)

- 450 • Ungleichheit (\neq)
 451 • Größer ($>$) und kleiner ($<$)
 452 • Größer gleich (\geq) und kleiner gleich (\leq)

453 Das Ergebnis von logischen Operatoren ist immer TRUE oder FALSE.
 454 Bei Vektoren kommt es immer zu einer elementweisen Anwendung. Wir können beispielsweise abfragen, an
 455 welchen Fotofallenstandorten mehr als 100 Rehe fotografiert wurden:

```
anzahl_rehe > 100
```

456 ## [1] TRUE FALSE TRUE FALSE TRUE TRUE FALSE TRUE TRUE TRUE FALSE TRUE
 457 ## [13] FALSE TRUE TRUE

458 Das Ergebnis ist ein Vektor vom Datentyp `logi` in der selben Länge wie `anzahl_rehe`.
 459 Wir können z. B. abfragen, welche Fotofallenstandorte sich in Revier B befinden.

```
reviere == "Revier B"
```

460 ## [1] FALSE FALSE FALSE FALSE FALSE TRUE TRUE TRUE TRUE TRUE FALSE FALSE
 461 ## [13] FALSE FALSE FALSE

462 Des Weiteren können logische Ausdrücke miteinander verknüpft werden. Dies geschieht mit einem logischen
 463 Und (`&`) oder einem logischen Oder (`|`). Für das logische Und müssen beide Ausdrücke ein TRUE zurückgeben
 464 um ein TRUE zu erhalten. Für ein logisches Oder reicht es, wenn einer der beiden Ausdrücke TRUE zurückgibt,
 465 um ein TRUE zu erhalten.

466 Damit können wir nun z. B. die beiden vorherigen Abfragen verbinden. Die erste Abfrage ist: Hat eine
 467 Fotofalle mehr als 100 Rehe fotografiert und stand die Fotofalle in Revier B.

```
anzahl_rehe > 100 & reviere == "Revier B"
```

468 ## [1] FALSE FALSE FALSE FALSE FALSE TRUE FALSE TRUE TRUE TRUE FALSE FALSE
 469 ## [13] FALSE FALSE FALSE

470 Das war jetzt eine Abfrage mit einem logischen Und. Würden wir ein logisches Oder verwenden, dann
 471 bekommen wir für alle Elemente ein TRUE, die entweder in Gebiet B stehen oder mehr als 100 Rehfotos
 472 aufgezeichnet haben.

```
anzahl_rehe > 100 | reviere == "Revier B"
```

473 ## [1] TRUE FALSE TRUE FALSE TRUE TRUE TRUE TRUE TRUE TRUE FALSE TRUE
 474 ## [13] FALSE TRUE TRUE

475 Das Arbeiten mit logischen Werten kann für Erste etwas abstrakt erscheinen, aber wir werden im folgenden
 476 Abschnitt (Abschnitt 3.5) zahlreiche Anwendungsbeispiele dafür sehen.

477

478 **Aufgabe 6: Arbeiten mit logischen Werten**
479

480 Überlegen Sie für jede Aufgabe erst was das richtige Ergebnis ist und Überprüfen Sie dieses dann mit R.

- 481 1. TRUE | FALSE
 482 2. FALSE & TRUE
 483 3. (FALSE & TRUE) | TRUE
 484 4. (2 != 3) | FALSE
 485 5. FALSE + 10
 486 6. TRUE + 10
 487 7. TRUE + 10 == FALSE + 10
 488 8. sum(c(TRUE, TRUE, FALSE, FALSE))

489 **3.5 Zugreifen auf Elemente eines Vektors (=Untermengen)**

490 Sehr oft wollen wir auf bestimmte Werte in einer Datenstruktur zugreifen. Beispielsweise könnte es uns
 491 interessieren, wie viele Rehe im Mittel auf allen Fotofallen aus Revier A gesehen wurden. Das Zugreifen auf
 492 Elemente mit bestimmten Eigenschaften ist die häufigste Anwendung für logische Operationen in R. Aus
 493 diesem Grund ist das Unterkapitel “Arbeiten mit logischen Werten” auch Teil des Kapitels “Vektoren”.

494 Bei Vektoren kann auf die einzelnen Elemente mit eckigen Klammern ([]), diese werden auch Indizierungs-
 495 klammern genannt) zugegriffen werden. Der Ausdruck `anzahl_rehe[2]` gibt die Anzahl an fotografierten
 496 Rehen für die zweite Fotofalle zurück, also die zweite Stelle des Vektors `anzahl_rehe`. Es gibt zwei Mög-
 497 lichkeiten, was in die eckigen Klammern geschrieben werden kann: 1.) die Positionen der Elemente die man
 498 zurückhaben möchte, wie eben beschrieben. Ist es mehr als ein Element, dann muss ein Vektor mit den
 499 Positionen übergeben werden. Die 2.) Möglichkeit der Indizierung ist also ein logischer Vektor in der gleichen
 500 Länge des Vektors, den ich indizieren möchte. Es werden alle Elemente zurückgegeben, bei denen in dem
 501 logischen Vektor TRUE eingetragen ist.

502 1.) Abfragen des zweiten Elements in dem Vektor `anzahl_rehe`:`anzahl_rehe[2]`503 `## [1] 79`504 2.) Abfragen aller Elemente aus `anzahl_rehe`, die aus dem Revier A stammen.

```
ist_a <- reviere == "Revier A"
anzahl_rehe[ist_a]
```

505 `## [1] 132 79 129 91 138`

```
# oder alternativ mit Methode 1.)
anzahl_rehe[1:5] # da `1:5` einen Vektor mit allen Zahlen von 1 bis 5 erstellt.
```

506 ## [1] 132 79 129 91 138

507 Hier ist nochmals hervorzuheben, dass innerhalb der eckigen Klammer mit dem Befehl `c(1, 2, 3, 4, 5)`
 508 bzw. `1:5` ein Vektor erstellt wird, der die Position der Elemente angibt, die zurückgegeben werden sollen.

509

510 Aufgabe 7: Zugreifen auf Vektorelemente

512 Erstellen Sie einen neuen Vektor `bhd`

```
bhd <- c(12, 32, 39, 41, 12, 30)
```

- 513 • Wählen Sie aus dem Vektor `bhd` nur das 2. und 3. Element aus.
 514 • Wählen Sie aus dem Vektor `bhd` das letzte Element aus.
 515 • Wählen Sie aus dem Vektor `bhd` das letzte Element aus, ohne die Zahl 6 zu schreiben.

516

517 Alternativ könnte das gleiche Ergebnis mit einem logischen Vektor erreicht werden. Für eine bessere Über-
 518 sichtlichkeit wird erst ein Vektor `sub` erstellt, in dem die logischen Werte gespeichert werden:

```
sub <- c(TRUE, TRUE, TRUE, TRUE, TRUE, FALSE, FALSE, FALSE,
        FALSE, FALSE, FALSE, FALSE, FALSE, FALSE)
anzahl_rehe[sub]
```

519 ## [1] 132 79 129 91 138

520 Das Erstellen des `sub`-Vektors ist mühsam und wenig zielführend. Wenn wir auf die Erkenntnisse aus dem
 521 vorherigen Kapitel zurückgreifen, kann dies leicht automatisiert werden, indem wir einfach abfragen, welche
 522 Elemente in Revier zu Revier A gehören.

```
sub <- reviere == "Revier A"
anzahl_rehe[sub]
```

523 ## [1] 132 79 129 91 138

524 Das kann nochmals vereinfacht werden, indem wir den Hilfsvektor `sub` einfach weglassen und den Ausdruck
 525 direkt in die eckigen Klammern ziehen.

```
anzahl_rehe[reviere == "Revier A"]
```

526 ## [1] 132 79 129 91 138

527 Wenn wir jetzt noch den Mittelwert der Anzahl fotografierten Rehe aus Revier A bilden möchten, erweitert
528 sich der Ausdruck um einen Funktionsaufruf zur Funktion `mean`.

```
mean(anzahl_rehe[reviere == "Revier A"])
```

529 ## [1] 113.8

530

531 Aufgabe 8: logische Werte

533 Verwenden Sie nochmals den Vektor mit den Anzahl Rehen die an unterschiedlichen Fotofallenstandorten
534 fotografiert wurden.

```
anzahl_rehe <- c(132, 79, 129, 91, 138, 144, 55, 103, 139, 105, 96, 146, 95, 118, 107)
```

- 535 1. Wählen Sie alle Standorte aus für die Aussage $90 \leq x < 120$ zu trifft (wobei x für die Anzahl Fotos an
536 einem Standort steht).
- 537 2. Berechnen Sie die mittlere Anzahl Fotos für alle in 1) ausgewählten Standorte.

538 3.6 Der %in%-Operator

539 Häufig wollen wir mehrere Elemente aus einem Vektor auswählen, die in einem anderen Vektor enthalten
540 sind. Als einfaches Beispiel nehmen wir zwei Vektoren:

```
arten <- c("FI", "BU")
messungen_arten <- c("FI", "BU", "BU", "EI", "EI", "BI", "FI", "BI", "EI")
```

541 Wenn wir aus dem Vektor `messungen_arten` alle FI auswählen wollen, können wir dies mit einem logischen
542 `==` machen:

```
messungen_arten[messungen_arten == "FI"]
```

543 ## [1] "FI" "FI"

```
# oder
messungen_arten[messungen_arten == arten[1]]
```

```
544 ## [1] "FI" "FI"
```

545 Etwas komplizierter wird es, wenn wir zwei oder mehr Elemente auswählen wollen. Dies geht auch mit
546 logischen Operationen.

```
messungen_arten[messungen_arten == arten[1] | messungen_arten == arten[2]]
```

```
547 ## [1] "FI" "BU" "BU" "FI"
```

548 Diese Herangehensweise wird aber für > 2 Elemente in `arten` sehr mühsam und fehleranfällig. Eine Alternative
549 bietet der `%in%`-Operator. Dieser testet, ob Elemente eines Vektors in einem zweiten Vektors enthalten
550 sind. Der Operator ist also eine verkürzte Schreibweise für hintereinander durchgeführte logische Oder Ab-
551 fragen.

```
messungen_arten %in% arten
```

```
552 ## [1] TRUE TRUE TRUE FALSE FALSE FALSE TRUE FALSE FALSE
```

```
messungen_arten[messungen_arten %in% arten]
```

```
553 ## [1] "FI" "BU" "BU" "FI"
```

```
554
```

555 **Aufgabe 9: Auswählen von Elementen in einem Vektor (%in%)**

557 Der Vector LETTERS ist in R vorhanden und enthält die Buchstaben von A bis Z.

```
LETTERS
```

```
558 ## [1] "A" "B" "C" "D" "E" "F" "G" "H" "I" "J" "K" "L" "M" "N" "O" "P" "Q" "R" "S"  
559 ## [20] "T" "U" "V" "W" "X" "Y" "Z"
```

560 Wählen Sie aus LETTERS nur die Vokale aus.

561 4 Faktoren (factors)

562 R besitzt einen besonderen Datentyp – *Faktoren* (engl. factors) – zum speichern von diskreten Kovariaten
 563 (z.B. Baumart, Augenfarbe oder Automarke). Faktoren erlauben es Daten vom Typ `character` effizienter
 564 abzuspeichern. Dabei wird jeder eindeutiger Wert (=Level) mit einer Zahl codiert und dann werden nur diese
 565 Zahlen zusammen mit einer Tabelle zum Nachschauen der Werte gespeichert (siehe dazu auch [McNamara](#)
 566 [and Horton 2018](#)). Faktoren haben im Gegensatz zu Zeichenketten zusätzliche Eigenschaften. Man kann sie
 567 z. B. sortieren.

568 Mit der Funktion `factor()` kann ein Faktor erstellt werden. Im einfachsten Fall wird nur ein Vektor über-
 569 geben.

```
a <- c("FI", "BU", "FI", "EI", "EI", "FI", "FI")
factor(a)
```

570 ## [1] FI BU FI EI EI FI FI
 571 ## Levels: BU EI FI

572 Ohne weitere Spezifikation werden für die *Levels* die Werte alphabetisch übernommen (das kann später z.B.
 573 beim Erstellen von Abbildungen wichtig sein). Für eine benutzerdefinierte Anordnung der Levels, kann das
 574 Argument `levels` verwendet werden.

```
factor(a, levels = c("FI", "BU", "EI"))
```

575 ## [1] FI BU FI EI EI FI FI
 576 ## Levels: FI BU EI

577 Es ist auch möglich, die Beschriftung (= labels) der unterschiedlichen Levels anzugeben mit dem Argument
 578 `labels`.

```
af <- factor(a, levels = c("FI", "BU", "EI"),
             labels = c("Fichte", "Buche", "Eiche"))
af
```

579 ## [1] Fichte Buche Fichte Eiche Eiche Fichte Fichte
 580 ## Levels: Fichte Buche Eiche

581 Mit der Funktion `levels()`, können die unterschiedlichen Levels eines Faktors abgefragt und auch gesetzt
 582 werden.

```
levels(af)
```

583 ## [1] "Fichte" "Buche" "Eiche"

```

levels(af) <- c("Fi", "Bu", "Ei")
af

584 ## [1] Fi Bu Fi Ei Ei Fi Fi
585 ## Levels: Fi Bu Ei

586 Schlussendlich kann man mit der Funktion relevel() die Referenzkategorie eines Faktors (der erste Level)
587 angepasst werden. Das ist kann z. B. für lineare Modelle wichtig sein.

af

588 ## [1] Fi Bu Fi Ei Ei Fi Fi
589 ## Levels: Fi Bu Ei

relevel(af, "Bu")

590 ## [1] Fi Bu Fi Ei Ei Fi Fi
591 ## Levels: Bu Fi Ei

592 Mit der Funktion as.character() kann ein Faktor wieder als Variable vom Typ character dargestellt
593 werden.

as.character(af)

594 ## [1] "Fi" "Bu" "Fi" "Ei" "Ei" "Fi" "Fi"
595 Achtung mit der Funktion as.numeric() erhält man die interne Kodierung von Faktoren.

af

596 ## [1] Fi Bu Fi Ei Ei Fi Fi
597 ## Levels: Fi Bu Ei

as.numeric(af)

598 ## [1] 1 2 1 3 3 1 1

599 Fichte ist das erste Level des Faktors, deshalb erhalten alle Fichteneinträge den Wert 1. Bucheinträge erhalten
600 den Wert 2 und 3 für Eichen.

```

601

602 **Aufgabe 10: Faktoren**

603

- 604 Verwenden Sie den Vektor **staedte** und erstellen Sie einen Vektor mit der Anordnung der **levels** in umge-
605 kehrter alphabetischer Reihenfolge.

```
staedte <- c("Berlin", "Aachen", "Berlin", "Ulm", "Aachen",
           "Berlin", "Berlin", "Aachen", "Ulm", "Ulm")
```

606 **4.1 Das Paket **forcats****

- 607 Mit dem Paket aus **forcats** werden einige Operationen mit Faktoren erleichtert. Wir schauen uns hier
608 Funktion an, die es erleichtern:

- 609 1. Die Anordnung von Levels anzupassen.
610 2. Levels zusammenzufassen oder zu entfernen.
611 3. Labels zu ändern.

612 **4.1.1 Anpassen der Anordnung von Faktoren**

- 613 Wir verwenden nochmals den **a** Vektor mit unterschiedlichen Baumarten:

```
a <- c("FI", "BU", "FI", "EI", "EI", "FI", "FI")
```

- 614 Die Funktion **factor()** ordnet die Levels alphanumerisch, wie wir bereits gesehen haben.

```
factor(a)
```

- 615 ## [1] FI BU FI EI EI FI FI
616 ## Levels: BU EI FI

- 617 Die Funktion **fct()** aus dem **forcats**-Paket übernimmt die Reihenfolge so, wie sie in den Daten erscheinen.

```
f1 <- fct(a)  
f1
```

- 618 ## [1] FI BU FI EI EI FI FI
619 ## Levels: FI BU EI

- 620 **forcats** stellt Funktionen zur Verfügung, um die Reihenfolge umzukehren,

```
fct_rev(f1)
```

621 ## [1] FI BU FI EI EI FI FI
622 ## Levels: EI BU FI

623 nach der Häufigkeit zu sortieren oder

```
fct_infreq(f1)
```

624 ## [1] FI BU FI EI EI FI FI
625 ## Levels: FI EI BU

626 eine zufällige Sortierung.

```
fct_shuffle(f1)
```

627 ## [1] FI BU FI EI EI FI FI
628 ## Levels: EI FI BU

629 5 Spezielle Einträge

630 In vielen Fällen werden spezielle Einträge benötigt, bspw. bei

- 631 • fehlenden Einträgen NA,
- 632 • leeren Einträgen NULL,
- 633 • undefinierten Einträgen NaN (Not a Number) oder
- 634 • unendlichen Zahlen (Inf).

635 Spezielle Einträge sind reservierte Namen. Sie können nicht überspeichert werden.

636 5.1 NA

637 R verfügt über einen speziellen Wert für fehlende Einträge. Auch wenn in Vektoren eigentlich nur ein Datentyp
638 erlaubt ist, sind NA zwischen den anderen Einträgen erlaubt. Der Datentyp des Vektors wird durch NA
639 Einträge nicht verändert.

```
na1 <- c("foo", NA, "foo")
str(na1)

## chr [1:3] "foo" NA "foo"

na2 <- c(3, 6, NA)
str(na2)

## num [1:3] 3 6 NA
```

642 Der logische Operatator zum Test auf fehlende Wert ist `is.na()`. Dieser kann genauso wie die bereits be-
643 kannten logischen Operatoren bspw. zum Filtern verwendet werden. Die `na.omit()` Funktion entfernt NA
644 aus dem Datensatz.

```
is.na(na1)

## [1] FALSE TRUE FALSE

na.omit(na1)

## [1] "foo" "foo"
## attr(),"na.action")
## [1] 2
## attr(),"class")
## [1] "omit"
```

651 Die bereits bekannten logischen Operationen ergeben NA, wenn Sie auf Daten angewendet werden, die NA
652 enthalten. Berechnungen mit NA ergeben ebenfalls NA. Bei der angewandten Programmierung müssen sie also
653 darauf achten, dass Ihre Daten frei von NA sind oder sie fangen die NA vorher ab.

```
na2 < 3
```

654 ## [1] FALSE FALSE NA

```
1 + NA
```

655 ## [1] NA

656 Viele R Funktionen haben eingebaute Methoden zum Umgang mit NA. Die Funktion `mean()` bspw. ergibt
657 (wie die meisten Funktionen) standardmäßig NA wenn sie auf Vektoren mit Datenlücken angewendet wird,
658 es sei denn man stellt innerhalb der Funktion ein, dass Datenlücken entfernt werden sollen.

```
mean(na2)
```

659 ## [1] NA

```
mean(na2, na.rm = TRUE)
```

660 ## [1] 4.5

661 5.2 NULL

662 Im Gegensatz zu NA wird NULL für leere Einträge verwendet, und nicht für fehlende Einträge. Da in der
663 Mathematik leere Einträge und fehlende Einträge unterschiedliche Informationen darstellen, können diese
664 beiden Fälle unterschieden werden. Mit der Funktion `is.null()` kann man überprüfen, ob ein Element in
665 einem Vektor NULL ist oder nicht.

666 5.3 Inf

667 Die größtmögliche Zahl in R ist $1.7976931 * 10^{308}$. Größere Zahlen werden als unendlich gespeichert und
668 verarbeitet.

```
10^309
```

669 ## [1] Inf

```
2 * Inf
```

670 ## [1] Inf

1 + Inf

671 ## [1] Inf

3 / 0

672 ## [1] Inf

-3 / 0

673 ## [1] -Inf

3 / Inf

674 ## [1] 0

675 Infinity kann mit `is.infinite` und `is.finite` getestet werden. Relationäre Operatoren funktionieren erwartungsgemäß.

```
inf1 <- c(Inf, 0, 3, -Inf, 10)
is.infinite(inf1)
```

677 ## [1] TRUE FALSE FALSE TRUE FALSE

`is.finite(inf1)`

678 ## [1] FALSE TRUE TRUE FALSE TRUE

`inf1 < 3`

679 ## [1] FALSE TRUE FALSE TRUE FALSE

680

681 Aufgabe 11: Vektoren mit speziellen Einträgen

683 Verwenden Sie den Vektor

```
foo <- c(13563, -13156, -14319, 16981, 12921, 11979, 9568, 8833, -12968, 8133)
```

- 684 • Nehmen Sie jeden Eintrag hoch 75. Filtern Sie alle unendlichen Einträge aus dem Vektor.
 685 • Wie viele Einträge sind unendlich negativ?

686 Verwenden Sie den Vektor

```
foo <- c(4.3, 2.2, NULL, 2.4, NaN, 3.3, 3.1, NULL, 3.4, NA, Inf)
```

687 Sind die folgenden Einträge richtig oder falsch? Überlegen Sie zunächst selbst bevor Sie die Aussagen in R
688 testen.

- 689 • Die Länge des Vektors ist 9.
690 • `is.na()` ergibt 2 Mal TRUE.
691 • `foo[9] + 4 / Inf` ergibt NA

692 Berechnen Sie den arithmetischen Mittelwert von `foo`.

6 data.frames oder Tabellen

694 Im vorherigen Teilabschnitt haben wir gesehen, wie mehrere Elemente des gleichen Datentyps in einem Vektor
 695 zusammengefasst werden können. Abschließend wurde anhand des Fotofallenbeispiels gezeigt, wie Vektoren
 696 eingesetzt werden können, um andere Vektoren zu indizieren. Wir erstellten drei Vektoren, die jeweils die
 697 Merkmalsausprägungen eines Merkmals aller Fotofallenstandorte speichern. In statistischer Sprache, sind die
 698 Fotofallen die Beobachtungen (oder auch Merkmalsträger genannt) und die Informationen zu den Fotofallen
 699 (also ID, Anzahl Rehe und das Revier) die Merkmale. Jeder beobachtete Wert (z.B. die 132 fotografierten
 700 Rehe von Kamera 1) ist dann eine Merkmalsausprägung.

701 Sie können sich ein `data.frame` wie eine Tabelle aus einem Tabellenkalkulationsprogramm vorstellen. Es gibt
 702 Zeilen in denen die Beobachtungen gespeichert sind und Spalten, die die Merkmale speichern. In unserem
 703 Fall gäbe es 15 Zeilen (eine Zeile für jede Fotofalle) und drei Spalten (jeweils eine Spalte für ID, Anzahl Rehe
 704 und Revier). Der Befehl zum Erstellen eines `data.frames` aus Vektoren in R ist `data.frame()`. Für unser
 705 Beispiel wäre es:

```
monitoring <- data.frame(
  ID = ids,
  anzahl_rehe = anzahl_rehe,
  revier = reviere
)
monitoring
```

	ID	anzahl_rehe	revier
## 1	Kamera_1	132	Revier A
## 2	Kamera_2	79	Revier A
## 3	Kamera_3	129	Revier A
## 4	Kamera_4	91	Revier A
## 5	Kamera_5	138	Revier A
## 6	Kamera_6	144	Revier B
## 7	Kamera_7	55	Revier B
## 8	Kamera_8	103	Revier B
## 9	Kamera_9	139	Revier B
## 10	Kamera_10	105	Revier B
## 11	Kamera_11	96	Revier C
## 12	Kamera_12	146	Revier C
## 13	Kamera_13	95	Revier C
## 14	Kamera_14	118	Revier C
## 15	Kamera_15	107	Revier C

722 Auf der linken Seite der Gleichungen stehen die Spaltenname des Data Frames. Im vorhergehenden Codebei-
 723 spiel wurde ein `data.frame` erstellt und in die Variable `monitoring` gespeichert. Die Funktion `data.frame()`
 724 nimmt als Argumente beliebig viele Paare, die immer aus einem Namen und einem Vektor mit dazugehörigen
 725 Werten bestehen. D.h., dass (wie bei den Vektoren) immer eine Spalte vom selben Typ sein muss, es aber

- 726 für jede Beobachtung (=Zeile) Merkmale von unterschiedlichen Typen geben kann. Data Frames sind die
 727 Standard-Objekte zum Speichern wissenschaftlicher Daten.

728 6.1 Wichtige Funktionen zum Arbeiten mit `data.frames`

- 729 Wichtige Funktionen um das Arbeiten mit `data.frames` zu erleichtern sind wieder `head()` und `tail()`, um
 730 die ersten bzw. letzten `n` Zeilen eines `data.frames` anzuzeigen.

```
head(monitoring, n = 2)
```

```
731 ##           ID anzahl_rehe   revier
732 ## 1 Kamera_1          132 Revier A
733 ## 2 Kamera_2          79 Revier A
```

- 734 Oder für die letzten 2 Beobachtungen.

```
tail(monitoring, 2)
```

```
735 ##           ID anzahl_rehe   revier
736 ## 14 Kamera_14         118 Revier C
737 ## 15 Kamera_15         107 Revier C
```

- 738 Mit den Funktion `nrow()` und `ncol()` können die Anzahl Zeilen und die Anzahl Spalten abgefragt werden:

```
nrow(monitoring)
```

```
739 ## [1] 15
```

```
ncol(monitoring)
```

```
740 ## [1] 3
```

- 741 Mit der Funktion `str()` (kurz für *structure*) kann schnell ein Überblick über sämtliche Variablen und deren
 742 Datentypen verschafft werden.

```
str(monitoring)
```

```
743 ## 'data.frame':    15 obs. of  3 variables:
744 ##   $ ID          : chr  "Kamera_1" "Kamera_2" "Kamera_3" "Kamera_4" ...
745 ##   $ anzahl_rehe: num  132 79 129 91 138 144 55 103 139 105 ...
746 ##   $ revier       : chr  "Revier A" "Revier A" "Revier A" "Revier A" ...
```

747

748 **Aufgabe 12: `data.frame`**

750 Stellen Sie sich vor, Sie machen eine kleine Umfrage, in der Sie fünf Menschen nach ihrem Studienfach, Semes-
751 ter und Alter befragen. Erstellen Sie einen `data.frame` mit dem Namen `umfrage1` für diese Informationen
752 und fragen Sie entweder fünf Mitstudierende oder erfinden Sie die Daten einfach.

753 **6.2 Zugreifen auf Elemente eines `data.frame`**

754 Für `data.frames` gilt genau das gleiche Prinzip. Nur dass wir jetzt zwei Dimensionen berücksichtigen müs-
755 sen: nämlich die Zeilen und die Spalten. Wir können immer noch mit eckigen Klammern (`[]`) auf Elemente
756 innerhalb eines `data.frames` zugreifen, müssen aber jetzt die `Zeile(n)` und die `Spalte(n)` angeben, die wir
757 haben möchten. Die Schreibweise ist immer `[Zeile(n), Spalte(n)]`. Für Zeilen und Spalten gelten genau
758 die gleichen Regeln wie für Vektoren. Wir können entweder einen Vektor mit den Positionen für die ge-
759 wünschten Zeilen und Spalten angeben oder einen logischen Vektor, der besagt welche Zeilen und Spalten
760 wir zurückhaben möchten.

761 Wenn wir z. B. die Anzahl Rehfotos von der vierten Fotofalle abfragen möchten, könnte man das so machen.

```
monitoring[4, 2]
```

762 `## [1] 91`

763 Alternativ, kann man den Spaltennamen auch einfach Ausschreiben. Dies hat beim Programmieren den
764 Vorteil, dass der Code lauffähig bleibt, falls sich die Reihenfolge der Spalten in der Datengrundlage ändert.
765 Nachteil ist entsprechend, dass der Code nicht mehr läuft, falls die Variablennamen sich ändern.

```
monitoring[4, "anzahl_rehe"]
```

766 `## [1] 91`

767 Wenn wir die Anzahl fotografierte Rehe von den ersten fünf Fotofallen abfragen möchten, dann müssen wir
768 für die Zeilen einen Vektor mit den Zahlen 1 bis 5 übergeben, für die Spalten ändert sich nichts.

```
monitoring[1 : 5, "anzahl_rehe"]
```

769 `## [1] 132 79 129 91 138`

770 Wenn wir nun nicht nur die Anzahl fotografierte Rehe zurückhaben möchten, sondern auch noch das Revier
771 für die ersten fünf Fotofallen, dann müssen wir für die Spalten lediglich das Revier hinzufügen.

```
monitoring[1 : 5, c("anzahl_rehe", "revier")]
```

```
772 ##      anzahl_rehe    revier
773 ## 1          132 Revier A
774 ## 2          79 Revier A
775 ## 3          129 Revier A
776 ## 4          91 Revier A
777 ## 5          138 Revier A
```

778 Wenn wir alle Spalten und/oder Zeilen eines `data.frames` abfragen möchten, dann kann man diese Position
 779 einfach frei lassen. Eine Abfrage für die ersten fünf Spalten aller Fotofallen würde so aussehen.

```
monitoring[1 : 5, ]
```

```
780 ##           ID anzahl_rehe    revier
781 ## 1 Kamera_1          132 Revier A
782 ## 2 Kamera_2          79 Revier A
783 ## 3 Kamera_3          129 Revier A
784 ## 4 Kamera_4          91 Revier A
785 ## 5 Kamera_5          138 Revier A
```

786

787 Aufgabe 13: Abfragen von Werten

789 Wir nehmen folgende Werte aus Übung 12 an:

```
umfrage1 <- data.frame(
  fach = c("Forst", "Bio", "Chemie", "Physik", "Forst"),
  semester = c(2, 3, 2, 1, 5),
  alter = c(21, 22, 21, 20, 23)
)
```

- 790 • Wählen Sie nur die ersten drei Zeilen aus und die erste und zweite Spalte aus.
 791 • Wählen Sie alle Zeilen und die erste und dritte Spalte aus.
 792 • Wählen Sie alle Spalten und die erste, dritte und vierte Zeile aus.

793

794 Mit dem \$-Zeichen kann bei `data.frames` direkt auf eine Spalte zugegriffen werden. Wenn wir z. B. für alle
 795 Fotofallen die Anzahl gesehener Rehe abfragen möchten, gibt es jetzt drei Möglichkeiten:

796 1. über das \$-Zeichen direkt die Spalte ansprechen. Diese Möglichkeit hat den Vorteil, dass R Studio den
797 Spaltennamen automatisch ausfüllen kann. Beim Tippen werden mögliche Spaltennamen vorgeschla-
798 gen. Sie wählen den Vorschlag aus, in dem Sie Tabulator (drücken.

```
monitoring$anzahl_rehe
```

799 ## [1] 132 79 129 91 138 144 55 103 139 105 96 146 95 118 107

800 2. Die Positionen für die Zeilen leer lassen und die Spalte abfragen.

```
monitoring[, "anzahl_rehe"]
```

801 ## [1] 132 79 129 91 138 144 55 103 139 105 96 146 95 118 107

802 3. Alle Zeilen und die Spalte explizit angeben.

```
monitoring[1 : nrow(monitoring), "anzahl_rehe"]
```

803 ## [1] 132 79 129 91 138 144 55 103 139 105 96 146 95 118 107

804 Anmerkung zu 3), der Ausdruck `1:nrow(monitoring)` ergibt einen Vektor mit den Zahlen 1 bis 15, da
805 `nrow(monitoring) = 15` ist. Diese Schreibweise ist zu empfehlen, wenn die Dimension des Vektors variabel
806 ist. Merken Sie sich diese Kombination aus Befehlen. Auf ähnliche weise können Sie vom Ende oder von
807 Anfang variable längen indizieren. Das ist z. B. nützlich, wenn Sie n - 1 Eionträge brauchen.

808 Schlussendlich kann man einen `data.frame` genauso mit logischen Vektoren abfragen, wie einen Vektor. Ein
809 Beispiel wäre, wenn wir alle Fotofallen abfragen möchten, die mehr als 100 Rehfotos gemacht haben. Der
810 erste Schritt wäre abzufragen, ob eine Fotofalle mehr als 100 Rehfotos gemacht hat.

```
monitoring$anzahl_rehe > 100
```

811 ## [1] TRUE FALSE TRUE FALSE TRUE TRUE FALSE TRUE TRUE TRUE FALSE TRUE
812 ## [13] FALSE TRUE TRUE

813 Das Ergebnis ist ein Vektor in der Länge von `monitoring` (15 Elementen). Hat eine Fotofalle mehr als 100
814 Rehfotos gemacht, ist das entsprechende Element des Vektors `TRUE` ansonsten `FALSE`. In dem `data.frame`
815 `monitoring` steht in jeder Zeile eine Beobachtung (also eine Fotofalle). Nun wollen wir genau diese Fotofallen
816 haben, die mehr als 100 Rehfotos gemacht haben.

```
monitoring[monitoring$anzahl_rehe > 100, ]
```

```
817 ##          ID anzahl_rehe    revier
818 ## 1   Kamera_1        132 Revier A
819 ## 3   Kamera_3        129 Revier A
820 ## 5   Kamera_5        138 Revier A
821 ## 6   Kamera_6        144 Revier B
822 ## 8   Kamera_8        103 Revier B
823 ## 9   Kamera_9        139 Revier B
824 ## 10  Kamera_10       105 Revier B
825 ## 12  Kamera_12       146 Revier C
826 ## 14  Kamera_14       118 Revier C
827 ## 15  Kamera_15       107 Revier C
```

828

Aufgabe 14: Abfragen von Werten 2

831 Verwenden Sie erneut den Datensatz aus Übung 13 und führen Sie folgende Abfragen durch:

- 832 • Alle Spalten für Studierende die Forstwissenschaften studieren.
- 833 • Alle Spalten für Studierende die Chemie oder Physik studieren.
- 834 • Die Spalte `fach` und `semester` für Studierende die 22 oder älter sind.

835 7 Schreiben und lesen von Daten

836 7.1 Textdateien

837 Bis jetzt haben wir Daten immer in R erstellt, dies ist eine eher unnatürliche Situation. In den meisten Fällen
 838 bekommen Sie Daten aus Felderhebungen, Sensoren oder sonstigen Quellen. Diese Daten müssen dann in R
 839 eingelesen werden. Daten liegen meist in einer tabellarischen Form und als Textdatei vor⁶.

840 Die Funktion `read.table` erlaubt es eine Textdatei in R einzulesen. Dabei sind fürs Erste drei Argumente
 841 wichtig:

- 842 • `file`: Der Pfad zur Datei die eingelesen werden soll. Dieser kann *absolut* oder *relativ* sein. Ein absoluter
 843 Pfad gibt den Ort der Datei, die gelesen werden soll, komplett an (auf einem Windows Rechner wäre
 844 das wahrscheinlich `C:/Users/...`). Im Gegensatz dazu gibt ein relativer Pfad den Ort an, an dem die
 845 Datei, die eingelesen werden soll, relativ zum aktuellen Arbeitsverzeichnis (working directory) von R
 846 an. Man kann das Arbeitsverzeichnis von R mit der Funktion `setwd()` setzen, es hat sich jedoch als
 847 sinnvoller erwiesen mit R Studio-Projekten zu arbeiten (mehr dazu im nächsten Abschnitt). Sie müssen
 848 den Pfad dann nur ab dem Ordner eintippen, in dem das Projekt liegt.
- 849 • `header`: Dieses Argument gibt an, ob die erste Zeile eine Kopfzeile mit den Spaltenüberschriften ist.
 850 Meist haben wir eine Kopfzeile, dann wäre `header = TRUE` richtig.
- 851 • `sep`: Das Trennzeichen zwischen verschiedenen Spalten. Es ist meist ein Leerzeichen (), Komma (,)
 852 oder Strichpunkt (;).

853 Die Datei `fotofallen.csv` finden Sie auf StudIP und kann einfach heruntergeladen werden. Sie können sich
 854 die Datei mit jedem Texteditor oder auch mit Excel oder Libre Office ansehen (Libre Office ist hier sogar
 855 besser als Excel, weil die Text Importfunktion komfortabler ist und eine Autodetect Funktion enthält). Die
 856 Datei kann mit dem folgenden Befehl in R eingelesen werden. Hier wurde die Datei in einem R Studio-Projekt
 857 in ein Unterverzeichnis `data` abgelegt.

```
dat <- read.table("data/fotofallen.csv", header = TRUE, sep = ",")  
head(dat)
```

```
858 ##          ID anzahl_rehe   revier  
859 ## 1 Kamera_1        132 Revier A  
860 ## 2 Kamera_2        79 Revier A  
861 ## 3 Kamera_3        129 Revier A  
862 ## 4 Kamera_4        91 Revier A  
863 ## 5 Kamera_5        138 Revier A  
864 ## 6 Kamera_6        144 Revier B
```

865 Es gibt viele Varianten der Funktion `read.table()`. Beispielsweise hat die Funktion `read.csv()` bereits
 866 die Argumente `sep = ','` und `header = TRUE` gesetzt. Die Funktion `read.csv2()` hat die in Deutschland

⁶Natürlich gibt es viele weitere Formate wie Daten vorliegen können, diese werden aber an dieser Stelle nicht weiter behandelt. Es sei lediglich auf das Paket `readxl` verwiesen, falls Sie Daten von MS Excel direkt in R einlesen möchten.

867 üblichen Argument `sep = ';'` und `dec = ',',` gesetzt. Sie müssen natürlich darauf achten, dass sie die csv
868 Dateien mit den gleichen Spezifikationen einlesen, mit denen sie gespeichert wurde. Siehe dazu auch die
869 Hilfeseite von `read.table()`.

870 Mit der Funktion `write.table()` kann ein `data.frame` auf die Festplatte geschrieben werden.

871

872 **Aufgabe 15: Lesen und Schreiben von Datein**

874 Lesen Sie die Datei `kompliziert.txt` ein. Schauen Sie die Hilfeseite an und vergewissern Sie sich, dass Sie
875 wissen was die Argumente `header`, `sep`, `dec` und `skip` bewirken. Setzten die Argumente richtig, damit die
876 Datei `kompliziert.txt` folgendes Ergebnis liefert.

8 Erstellen von Abbildungen

877 Abbildungen sind ein elementarer Baustein statistischer Analysen und deshalb von Beginn an Teil von R.
 879 **R is a free software environment for statistical computing and graphics.** Es gibt unterschiedliche
 880 Systeme einen Plot zu erstellen. In diesem Kurs werden wir kurz *Base Plots* vorstellen und dann das
 881 Zusatzpaket **ggplot2** vorstellen.

8.1 Base Plot

882 Die wichtigsten Grafiken für die einfache Datendarstellung sind schnell verfügbar. Etwas komplexere oder
 884 spezielle Grafiken erfordern mehr Programmieraufwand (folgt teilweise noch). Für viele komplexere Dia-
 885 gramme existieren bereits Codebeispiele oder Pakete. Es lohnt sich z. B. einen Blick in die R Graph Gallery
 886 (<https://r-graph-gallery.com/index.html>) zu werfen, bevor Sie beginnen komplexe Abbildungen zu erstellen.
 887 Stellen sie sich die einfache Grafik Schnittstelle (**base plots**) als zweidimensionale Leinwand vor, auf die
 888 Sie durch Code Ebene für Ebene Grafikelemente legen:

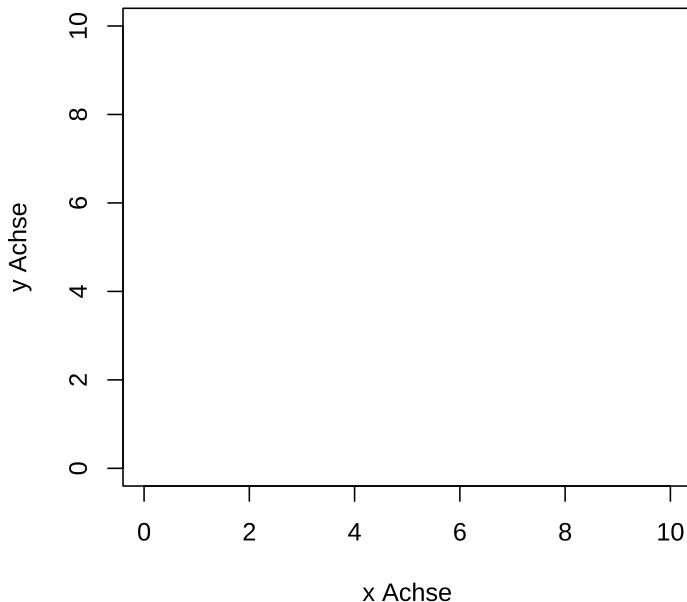
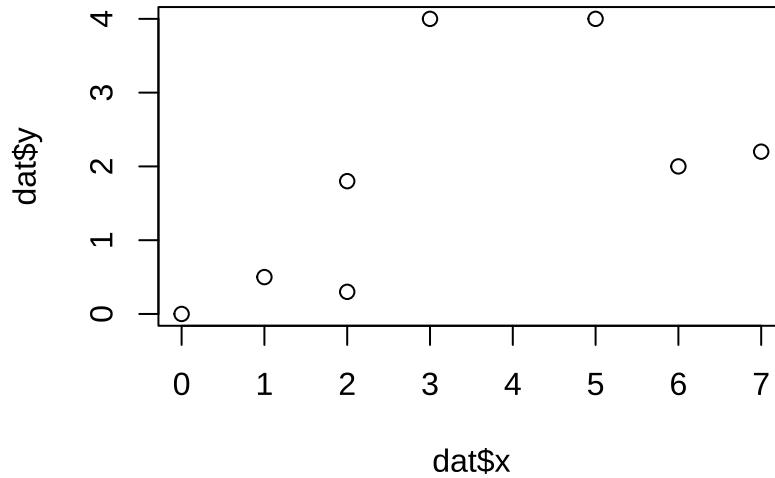


Abbildung 4: Beispiel einer leeren Grafikschnittstelle.

889 Hier drei einfache Beispiele für Abbildungen mit nur einer Ebene.

```
dat <- data.frame(
  x = c(0,    1,    2,    3,    5,    6,    7),
  y = c(0, 0.5, 1.8, 0.3, 4, 4, 2, 2.2)
```

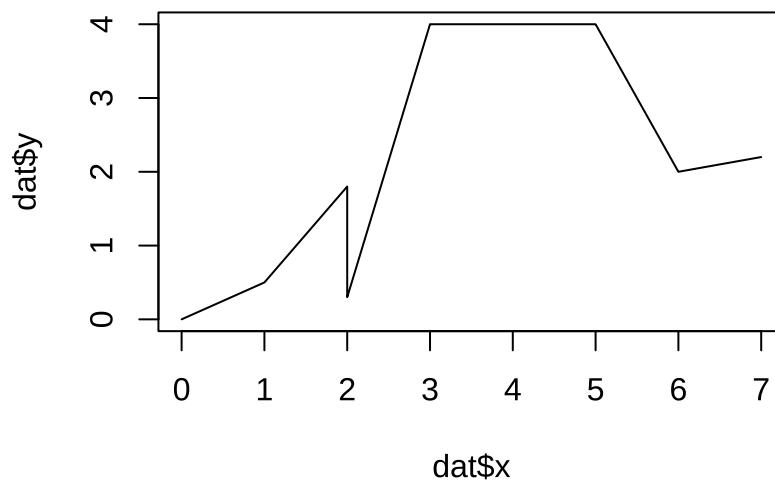
```
)  
  
plot(dat$x, dat$y, type = "p")
```



890

- 891 Mit dem Argument `type` kann die Art der Darstellung gesteuert werden. Der Standardwert ist `type = "p"`
892 (für points). Wir können den selben Plot mit Linien (`type = "l"`)

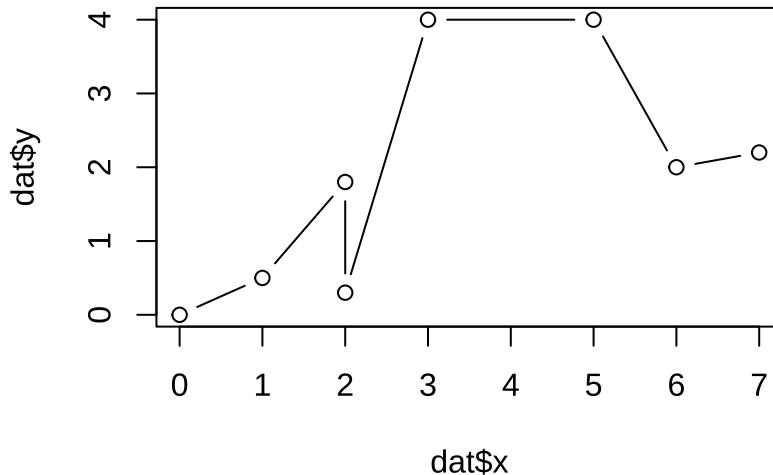
```
plot(dat$x, dat$y, type = "l")
```



893

894 oder mit Linien und Punkten (`type = "b"` für `both`)

```
plot(dat$x, dat$y, type = "b")
```



895

896 darstellen.

897

898 **Aufgabe 16: Base Plot 1**

900 Laden Sie den Datensatz `bhd_1.txt` und erstellen Sie eine Abbildung mit dem Alter jedes Baumes auf der
901 x-Achse und dem BHD auf der y-Achse.

902

903 Sie können entweder eine Grafik mit einem Befehl erzeugen (High-Level) oder die einzelnen Ebenen nach-
904 einander erzeugen (Low-Level). Sie können jeder Ebenen durch zusätzliche Befehle innerhalb des Funkti-
905 onsaufrufs Elemente hinzufügen. Mit jeder neuen Ebene können Sie die vorigen Ebenen jedoch nicht mehr
906 verändern. Die wichtigsten Argumente der `plot` Funktion sind:

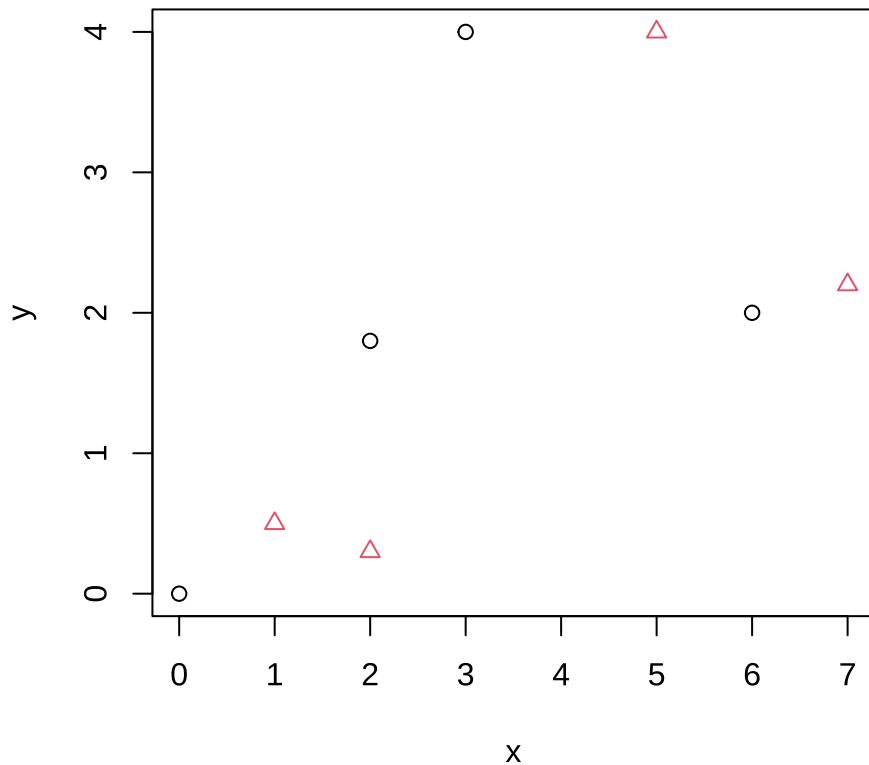
- 907 • `type` - Diagrammtyp
- 908 • `col` - Farbe
- 909 • `main` - Titel
- 910 • `sub` - Untertitel
- 911 • `pch` - Punktsymbol

- 912 • `lty` - Linientyp
 913 • `lwd` - Linienstärke
 914 • `xlab` bzw. `ylab` - Achsenbeschriftungen
 915 • `xlim`, `ylim` - Grenzen der Achsenanschnitte
 916 • `axes` - Sollen die Achsen eingezeichnet werden? Oder leer gelassen werden, um sie nachträglich als
 917 low-level Ebene einzuziehen?
 918 • `ann` - Achsenbeschriftung kann ganz weggelassen werden.

919 Sehen Sie sich die Hilfeseiten `?plot.default()` oder `?par()` an für weitere Informationen. Dort finden Sie
 920 auch eine vollständige Liste der Befehle. Einige Argumente können als Vektor übergeben werden. Hier z. B.
 921 die Farben und die Punktsymbole.

```
dat <- data.frame(
  x = c(0, 1, 2, 3, 5, 6, 7),
  y = c(0, 0.5, 1.8, 0.3, 4, 4, 2, 2.2),
  col = rep(c(1, 2), 4)
)

plot(x = dat$x, y = dat$y, col = dat$col, pch = dat$col, xlab = "x", ylab = "y")
```



922

923

924 **Aufgabe 17: Anpassen von Plots**

925

926 Verwenden Sie den Datensatz aus Übung 16 und passen Sie die Abbildung wie folgt an:

- 927 • Beschriften Sie die x- und y-Achse sinnvoll.
928 • Fügen Sie eine Überschrift hinzu.
929 • Wählen Sie ein anderes Symbol.
930 • Stellen Sie die Symbole in rot dar.

931

932 Über Low-Level Funktionen können einer Grafik Schnittstelle nacheinander Elemente hinzugefügt werden.

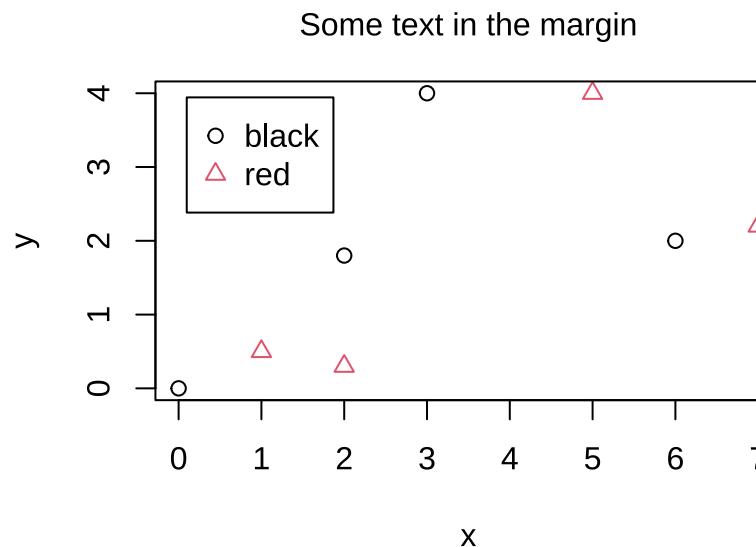
933 Die wichtigsten Funktionen sind

- 934 • `points()` - Fügt Punkte ein
935 • `lines()` - Fügt Linien ein
936 • `text()` - Fügt Text ein
937 • `mtext` - Fügt Text in den Rahmen (`margin`) ein
938 • `legend()` - Fügt eine Legende ein
939 • `abline()` - Fügt eine Gerade ein
940 • `curve()` - Fügt eine mathematische Funktion ein
941 • `arrows()` - Fügt Pfeile ein
942 • `grid()` - Fügt Hilfslinien ein

943 Dabei ist der Aufbau zunächst grundsätzlich wie in Abbildung 5 dargestellt. Der Vorteil von Low-Level
944 Funktionen ist, dass die einzelnen Level mehr Funktionen bieten als die High-Level Funktion und, dass Sie
945 sich die Reihenfolge der Ebenen definieren können.

946 Legenden können im base plot nur als Low-Level Funktion hinzugefügt werden. Mit der Funktion `legend`
947 werden die Einträge und die entsprechenden Punktsymbole oder Linientypen dargestellt. In der folgenden
948 Abbildung wird eine Legende in der linken oberen Ecke eingefügt.

```
dat <- data.frame(  
  x = c(0, 1, 2, 3, 5, 6, 7),  
  y = c(0, 0.5, 1.8, 0.3, 4, 4, 2, 2.2),  
  col = rep(c(1, 2), 4)  
)  
  
plot(x = dat$x, y = dat$y, col = dat$col, pch = dat$col, xlab = "x", ylab = "y")  
legend(x = "topleft", inset = 0.05, legend = c("black", "red"),  
       col = c(1, 2), pch = c(1, 2))  
mtext(side = 3, line = 1, "Some text in the margin")
```



949

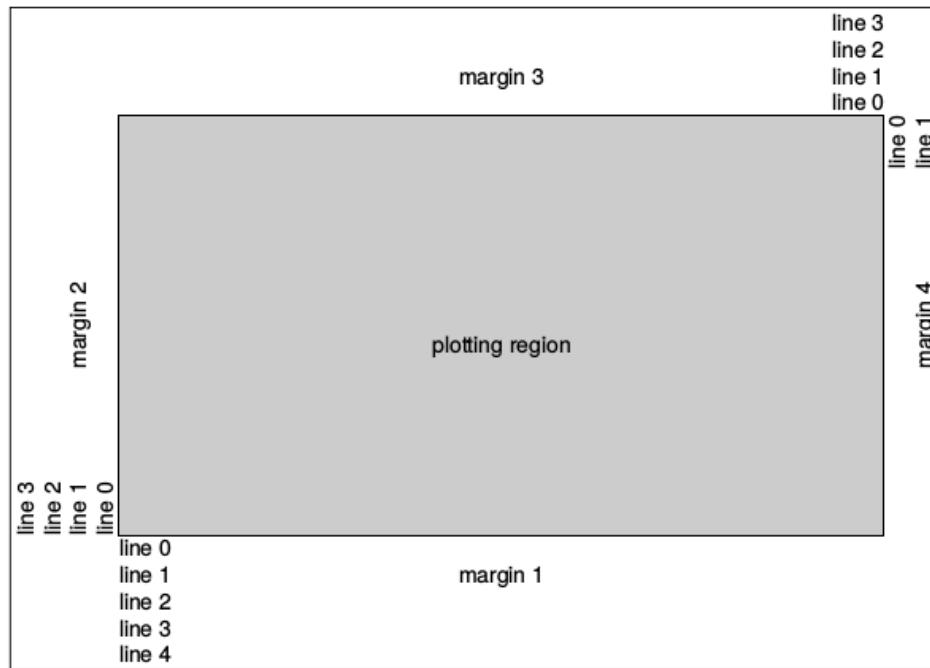


Abbildung 5: Grafikregionen eines base plots in R.

- 950 Mit diesem grundsätzlichen Aufbau sollten Sie bereits in der Lage sein auch komplexe Grafiken schnell zu
 951 gestalten. Wenn Sie mehrere Diagramme in einem Plot arrangieren möchten, können Sie mit dem `par()`
 952 Befehl ein Arrangement definieren. Sie haben dann zusätzlich zu den bereits bekannten Grafikregionen noch
 953 äußere Ränder (`outer margins`). Siehe Abbildung 6.

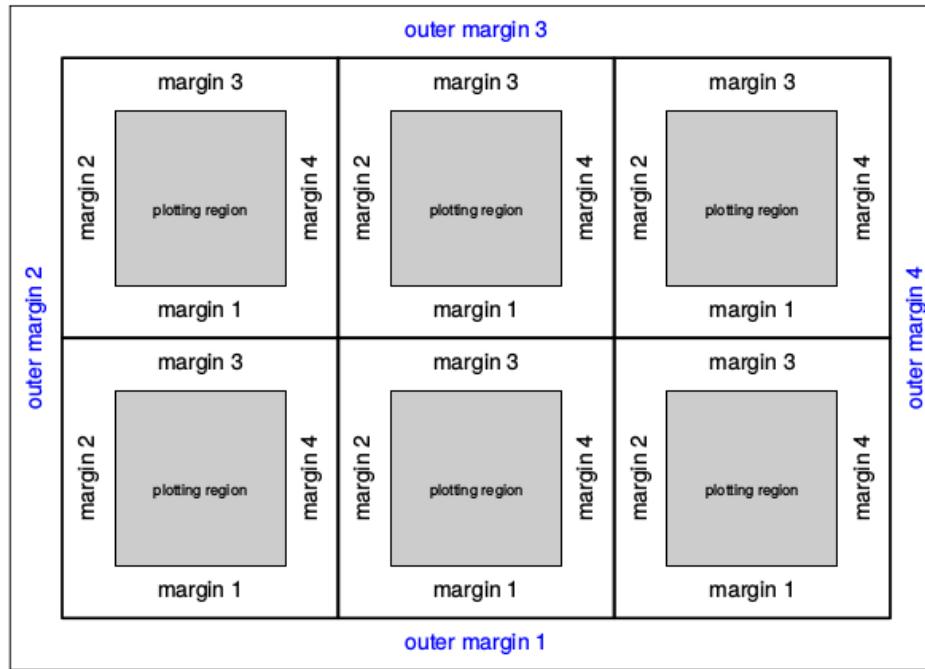


Abbildung 6: Schematischer Aufbau mehrere Diagramme in einem plot am Beispiel einer 3 x 2 Grafik.

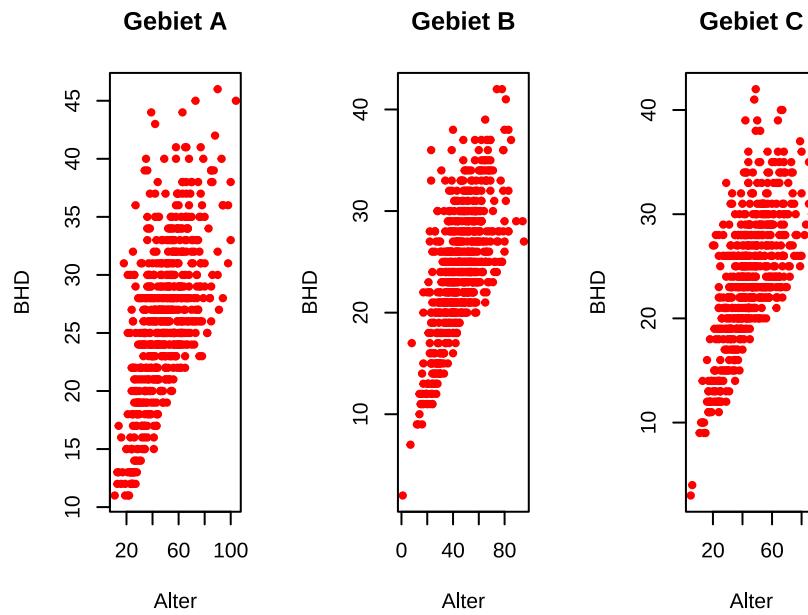
954 8.1.1 Mehrere Panels

955 Mit der Funktion `par()` kann auch eingestellt werden, dass ein Plot aus mehreren Subplots (= Panels)
 956 besteht. Die Argumente `mfrow` und `mfcol` können `par()` übergeben werden und kontrollieren die Anzahl
 957 Zeilen und Spalten für den Plot.

```
par(mfrow = c(1, 3))
```

958 Teilt den Plot in eine Zeile und drei Zeilen (= drei Plots nebeneinander).

```
par(mfrow = c(1, 3))
# Erstes Panel
plot(bhd ~ alter, xlab = "Alter", ylab = "BHD", pch = 20, col = "red",
      data = dat[dat$gebiet == "A", ], main = "Gebiet A")
# Zweites Panel
plot(bhd ~ alter, xlab = "Alter", ylab = "BHD", pch = 20, col = "red",
      data = dat[dat$gebiet == "B", ], main = "Gebiet B")
# Drittes Panel
plot(bhd ~ alter, xlab = "Alter", ylab = "BHD", pch = 20, col = "red",
      data = dat[dat$gebiet == "C", ], main = "Gebiet C")
```



959

960 Vergessen Sie nicht am Ende nochmals `par(mfrow = c(1, 1))` zu setzen, damit wieder nur ein Plot ange-
961 zeigt wird.

962 8.1.2 Speichern von Abbildungen

963 Wenn nicht anders angegeben, wird die Abbildung zunächst nur in der RStudio Grafik Schnittstelle abgebildet
964 (Pane 4). Von dort aus kann die Abbildung exportiert werden. Es bietet sich jedoch an das Speichern der
965 Abbildung direkt im Code zu programmieren. Mögliche Formate die Abbildung als Vektorgrafik zu speichern
966 sind

- 967 • `pdf()` oder
- 968 • `postscript()`.

969 Beispiele für Rastergrafiken sind

- 970 • `png()`,
- 971 • `bmp()` oder
- 972 • `jpeg()`.

973 Die Grafiksschnittstelle ist dann Ihre “Leinwand”. Mit dem Befehl `dev.off()` trennen Sie die Verbindung
974 zur Schnittstelle wieder. Ihre “Leinwand” wird also wieder geschlossen. So lange die Schnittstelle geöffnet ist
975 werden alle Low-Level Befehle an die Ausgabedatei gesendet. Hier am Beispiel einer PDF.

```

pdf("Grafik.pdf", height = 5)           # Öffnen der PDF Schnittstelle
plot(bhd ~ alter, type = "b", axes = FALSE,      # Abbildung produzieren, Ohne Achsen
     data = dat)
axis(side = 1, line = 1)                  # Achsen als High-Level Funktion hinzufügen
axis(side = 2, line = 1, las = 2)        # Sehen Sie selbst in der Hilfe was las bedeutet
dev.off()                                # Schnittstelle schließen

```

976 Achtung, wenn Sie die Funktion `dev.off()` nicht aufrufen, werden alle nachfolgenden Plots in die gleiche
 977 Datei geschrieben. Falls Sie nach einem Versuch einen Plot zu speichern plötzlich keine weiteren Plots mehr
 978 sehen, führen Sie einige Mal die Funktion `dev.off()` aus.

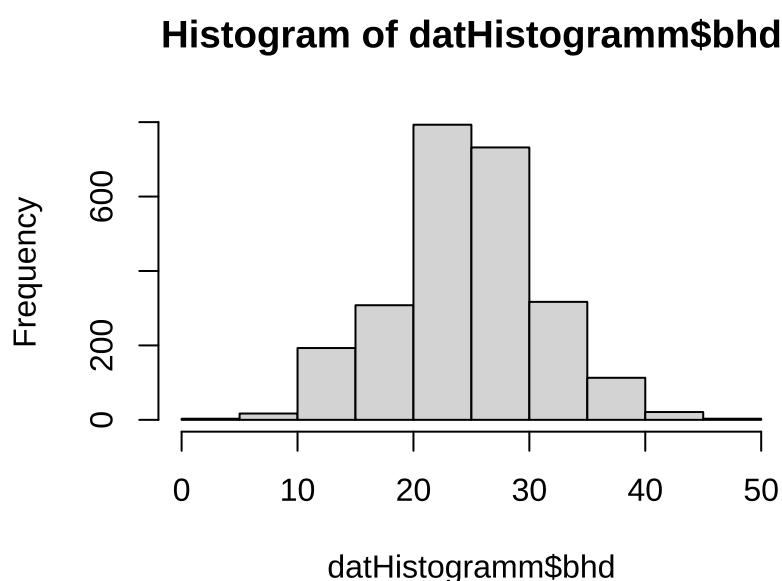
979 8.2 Histogramme

980 Neben den Streudiagrammen (*Scatterplots*, oder auch einfach x-y Diagramm) sind *Histogramme* in der
 981 angewandten Datenanalyse ein weiterer wichtiger Abbildungstyp. An Histogrammen wird die Häufigkeit
 982 von Beobachtungen nach Gruppen dargestellt. Sie sind deshalb so wichtig, weil man aus ihnen relevante
 983 Informationen über die Verteilung der Daten ablesen kann und somit einiges über die Messungen erfährt,
 984 das für die weitere Datenanalyse relevant sein kann. So werden auf einen Blick der Zusammenhang von
 985 Beobachtungshäufigkeit und Streuung deutlich, sowie auch die Form der Verteilung und ihre Schiefe. Die
 986 Interpretation werden wir bei den Boxplots noch weiter vertiefen.

```

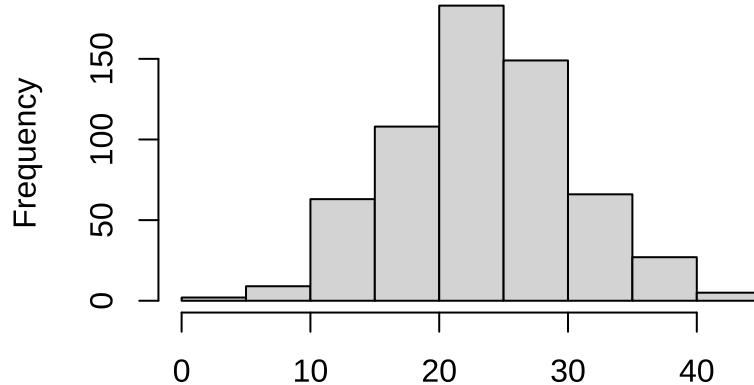
datHistogramm <- read.table("data/bhd_1.txt", header = TRUE)
# Über alle Baumarten
hist(datHistogramm$bhd)

```



```
# Nur für Eichen, Standardeinstellungen
hist(datHistogramm$bhd[datHistogramm$art == "EI"])
```

Histogramm of datHistogramm\$bhd[datHistogramm\$art == "EI"]

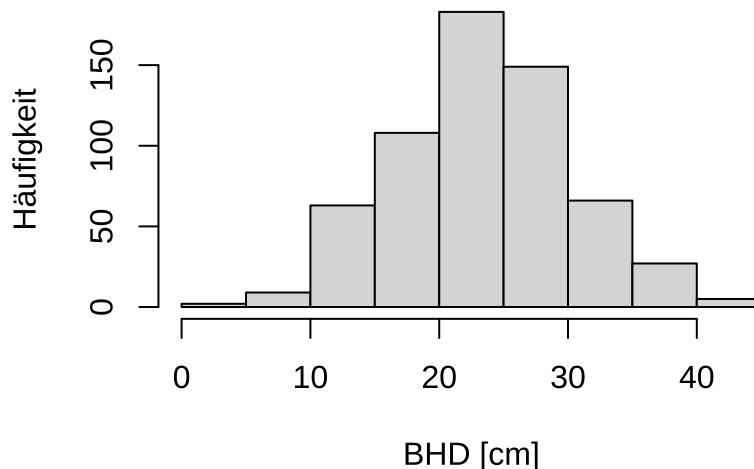


```
datHistogramm$bhd[datHistogramm$art == "EI"]
```

988

```
# Nur für Eichen, geänderte High-Level Einstellungen
hist(datHistogramm$bhd[datHistogramm$art == "EI"],
      xlab = "BHD [cm]", ylab = "Häufigkeit",
      main = "Anzahl der Eichen")
```

Anzahl der Eichen

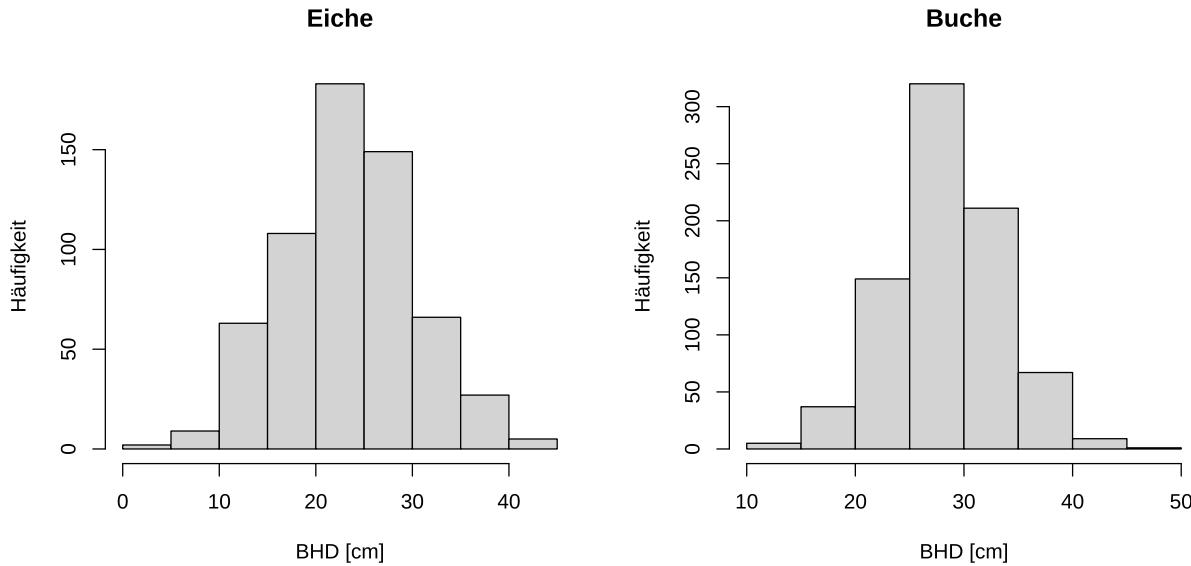


989

990 Eichen und Buchen im 2x1 Plot nebeneinander.

```
par(mfrow = c(1, 2))
hist(datHistogramm$bhd[datHistogramm$art == "EI"],
      xlab = "BHD [cm]", ylab = "Häufigkeit",
      main = "Eiche")
hist(datHistogramm$bhd[datHistogramm$art == "BU"],
      xlab = "BHD [cm]", ylab = "Häufigkeit",
      main = "Buche")
```

991



992

```
par(mfrow = c(1, 1)) # Alte Grafikeinstellungen wiederherstellen
```

993 8.3 Boxplots

994 Oft möchte man die Verteilung einer stetigen Variablen in Abhängigkeit einer diskreten Variable beschreiben
 995 oder visualisieren. Ein Beispiel dafür wäre die BHD-Verteilung in Abhängigkeit der Baumarten. Eine häufige
 996 Darstellungsform für solche Daten sind *Boxplots*. Sie stellen die durchschnittliche Ausprägung einer stetigen
 997 Variable und ihre Schwankung kompakt dar.

998 Boxplots bestehen aus drei Komponenten:

- 999 1. Eine *Box*, die den Bereich zwischen 0.25 und 0.75 Percentil abdeckt, diese Distanz wird auch die IQR
 1000 (Interquartile Range), bezeichnet. Zusätzlich wird die Box durch den Median (als dicke horizontale Linie)
 1001 unterteilt.
- 1002 2. Einzelne Punkte Ausreißer. Als Ausreißer werden Punkte bezeichnet, die $> 1.5 \text{ IQR}$ vom unteren oder
 1003 oberen Ende der Box entfernt sind.

- 1004 3. Eine senkrechte Linie von jeder Seite der Box bis zum letzten “Nicht-Ausreißer-Punkt”. Also der letzte
 1005 Punkt, der $> 1.5IQR$ aber nicht > 0.75 bzw. < 0.25 Percentil ist. Diese Linie wird auch als *Whisker*
 1006 bezeichnet.

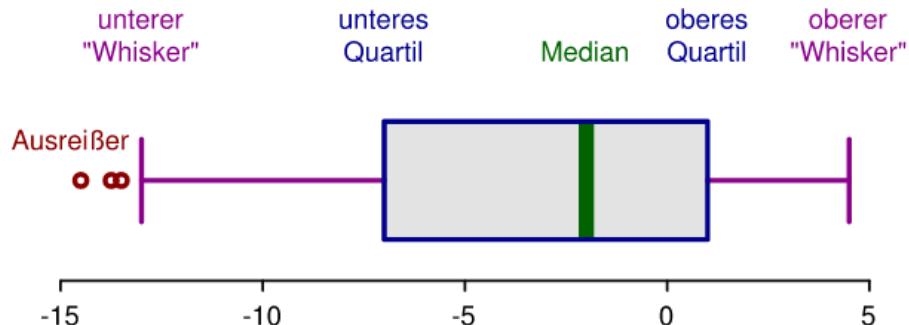
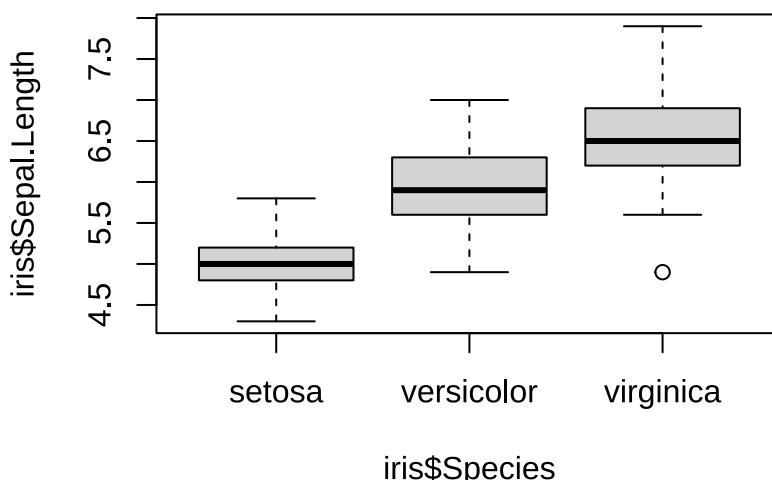


Abbildung 7: Schematische Darstellung eines Boxplots (Quelle: Von RobSeb - Eigenes Werk, CC BY-SA 3.0, <https://commons.wikimedia.org/w/index.php?curid=14697172>).

1007 Mit R kann mit der Funktion `boxplot()` ein Boxplot erstellt werden. Diese Funktion kann in zwei unter-
 1008 schiedlichen Ausprägungen verwendet werden.

- 1009 1. `boxplot(x)` erzeugt einen Boxplot für die Variable `x`.
 1010 2. `boxplot(x ~ y)` erzeugt einen oder mehrere Boxplots für `x` aber gruppiert nach `y`, dabei sollte `y` eine
 1011 kategoriale Variable sein. `x` und `y` können auch die Spaltennamen eines `data.frames` sein, dann muss
 1012 das `data.frame` mit dem Argument `data` zusätzlich übergeben werden.

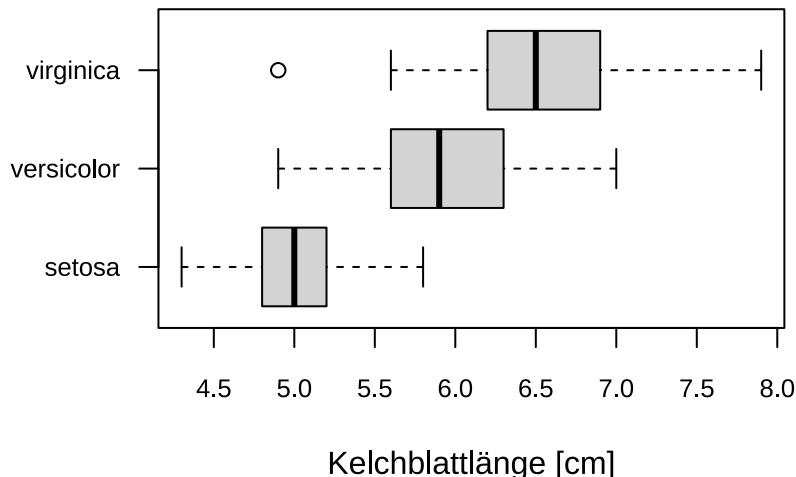
```
boxplot(iris$Sepal.Length ~ iris$Species)
```



1013

1014 Etwas eleganter ist es wenn wir das Argument `data` verwenden und den Plot etwas anpassen. Diese Schreib-
 1015 weise funktioniert für alle base plots.

```
boxplot(  
  Sepal.Length ~ Species, data = iris, ylab = NULL, xlab = "Kelchblattlänge [cm]",  
  horizontal = TRUE, las = 1, cex.axis = 0.8  
)
```



1016

1017

1018 Aufgabe 18: Boxplots

1019

- 1020 • Lesen Sie erneut den Datensatz `daten/bhd_1.txt` ein und speichern Sie diesen in die Variable
 1021 (`dat_bhd`).
 1022 • Wie viele BHD-Messungen gibt es für jedes Gebiet?
 1023 • Erstellen Sie für jedes Gebiet einen Plot

1024 Erstellen Sie Boxplots für jedes Gebiet und innerhalb der Gebiete für jede Art.

1025 8.4 ggplot2: Eine Alternative für Abbildungen

1026 `ggplot2` ist ein alternatives Plotting-System in `R`. Sie können mit `ggplot2` also grundsätzlich Abbildungen
 1027 mit dem selben Inhalt erstellen, wie mit Base Plots. Die Syntax und die optische Darstellung unterscheiden
 1028 sich jedoch grundsätzlich. `ggplot2` basiert auf den *grammar of graphics* von Leland Wilkinson. Die Idee ist,
 1029 alle nötigen Informationen der Abbildung miteinander zu verknüpfen. `ggplot2` ist also diametral zu Base

1030 Plots. Während Sie in base plot alle Elemente selbst definieren, ist die Idee von *ggplot2*, dass Sie nur die
 1031 Daten und den Diagrammtypen übergeben und die Funktion selbst eine schöne Abbildung erstellt. Selbstver-
 1032 ständlich können Sie aber auch in *ggplot2* viele Einstellungen vornehmen. Im base plot sehen Abbildungen
 1033 zunächst sehr karg und etwas unfertig aus. Sie müssen viele Einstellungen vornehmen, um eine publizierfä-
 1034 hige Grafik zu produzieren. In *ggplot2* sollen auch die einfachste Abbildungen schon ästhetisch sein. Mit
 1035 diesen gebündelten Informationen kann *ggplot2* die Abbildung automatisch verschönern. So werden bspw.
 1036 die Legenden automatisch erzeugt und auch die Formatierungen automatisch an die Datenlage angepasst.
 1037 *ggplot2* nimmt der*dem Entwickler*in also Arbeit ab. Dadurch sind die Abbildungen schon ohne viel Nach-
 1038 arbeit schick. Nachteil ist, dass der*dem Entwickler*in weniger Möglichkeiten zur Einstellung zur Verfügung
 1039 stehen und nutterspezifische Sonderwünsche somit schwerer umsetzbar sind. Sehen Sie sich das *Cheatsheet* zu
 1040 *ggplot2* an. Es ist in RStudio unter **Help > Cheatsheets** zu finden.

1041 Bei *ggplot2* sind Anweisungen zu den Daten und Anweisungen zur Darstellung voneinander getrennt. Die
 1042 Daten werden in den Ästhetikbefehl übergeben und dort klassifiziert. Dann folgen die Darstellungsanweisun-
 1043 gen. Ähnlich wie bei Base Plots, werden die Grafikelemente ebenenweise nacheinander programmiert, jedoch
 1044 mit einem `+` verbunden. Und hier liegt der wesentliche Unterschied zu Base Plots. Durch die `+` werden die
 1045 Ebenen zu einem Befehl verbunden und damit gleichzeitig erstellt.

1046 Die Erweiterung wird zunächst geladen⁷. Wir laden außerdem den Datensatz `iris`. Der Datensatz ist in R
 1047 fest integriert. Siehe `?iris` für mehr Informationen.

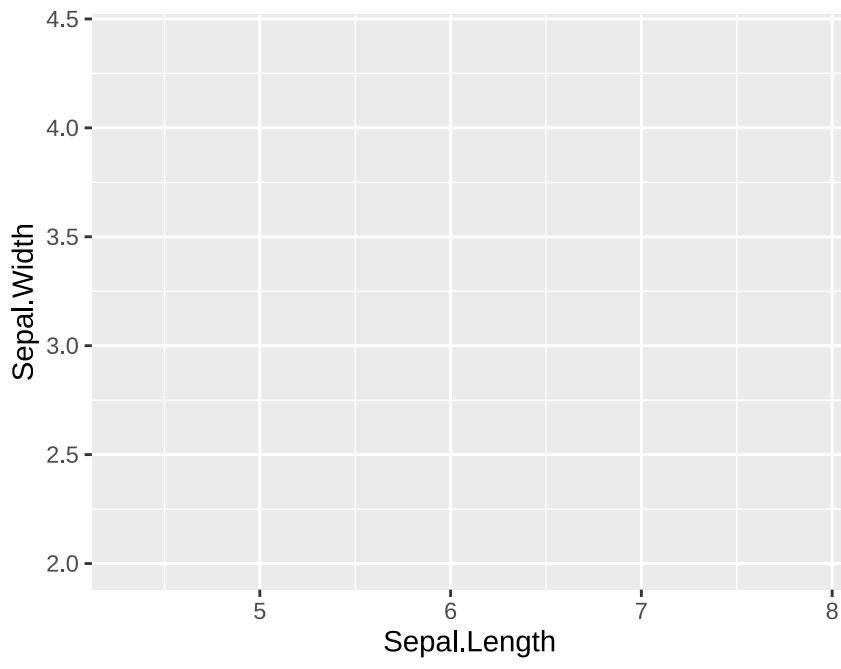
```
library(ggplot2)
head(iris)
```

```
1048 ##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
1049 ## 1          5.1       3.5      1.4       0.2  setosa
1050 ## 2          4.9       3.0      1.4       0.2  setosa
1051 ## 3          4.7       3.2      1.3       0.2  setosa
1052 ## 4          4.6       3.1      1.5       0.2  setosa
1053 ## 5          5.0       3.6      1.4       0.2  setosa
1054 ## 6          5.4       3.9      1.7       0.4  setosa
```

1055 Die Ästhetik wird bspw. folgendermaßen definiert.

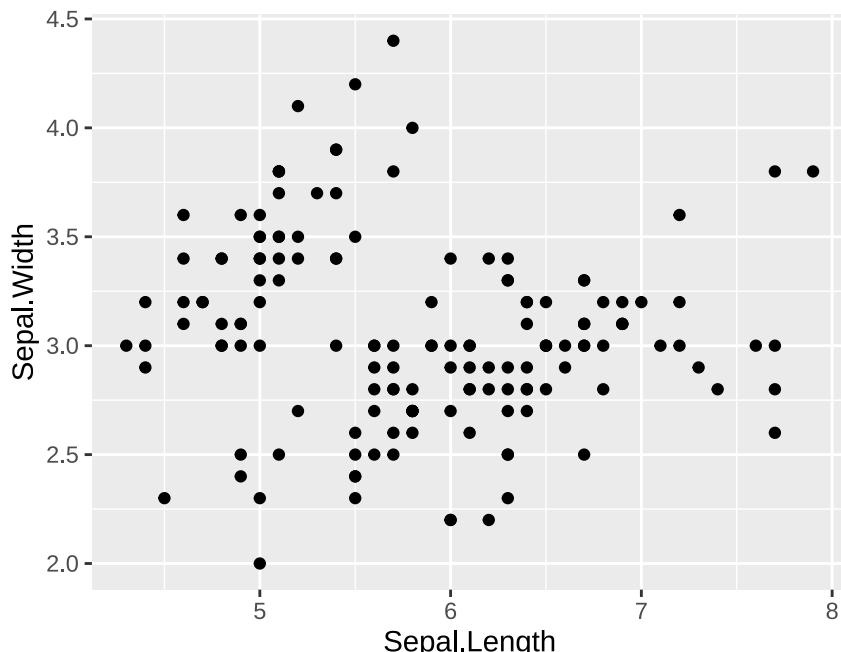
```
ggplot(iris, aes(x = Sepal.Length, y = Sepal.Width))
```

⁷Wir haben bis jetzt immer nur mit *base R* gearbeitet. D.h. wir haben nur Funktionen verwendet, die R bereits zur Verfügung stellt. Eine der großen Stärken von R sind die Erweiterungen (oder auch Pakete genannt). *ggplot2* ist so eine Erweiterung, die einmal mit `install.packages("ggplot2")` installiert werden muss. Danach muss man das Paket am Anfang jeder Session mit `library(ggplot2)` laden (am besten, Sie laden alle Bibliotheken ganz oben in Ihrem Code), damit die Funktionen aus dem Paket zur Verfügung stehen.



1057 Dieser Befehl zeichnet noch keine Daten. Die Daten werden lediglich herangezogen, um einen leeren Plot für
 1058 die Daten zu erstellen. In dem Beispiel wird die Variable `Sepal.Length` aus dem `data.frame iris` als x und
 1059 `Sepal.Width` als y Variable definiert. Diese Informationen stehen den folgenden Layern nun zur Verfügung,
 1060 sodass nach den + nur noch x und y verwendet werden müssen. Um bspw. einen Scatterplot zu erstellen
 1061 wird ein `geom_point()` Layer hinzugefügt. x und y werden automatisch an `geom_point` übergeben. Weitere
 1062 Einstellung sind in diesem Beispiel nicht notwendig, wären jedoch möglich. Siehe `?geom_point()`.

```
ggplot(iris, aes(x = Sepal.Length, y = Sepal.Width)) + geom_point()
```



1064

1065 **Aufgabe 19: Abbildungen mit ggplot2**

1066

1067 Verwenden Sie die Daten aus Aufgabe 16 und erstellen Sie einen Scatterplot mit ggplot2 wie in Aufgabe 16.

1068

1069 Wir haben mit der Funktion `geom_point()` demm Plot eine Punktgeometrie hinzugefügt. Es gibt noch viele
1070 weitere Geometrien. Die wichtigsten sind:

- 1071 • `geom_line()` für eine Linie.
1072 • `geom_histogram()` um ein Histogramm zu erstellen.
1073 • `geom_boxplot()` um einen Boxplot zu erstellen.
1074 • `geom_bar()` um ein Säulendiagramm zu erstellen.

1075 Welche Geometrie die richtige ist, richtet nach dem Typ der darzustellenden Variablen. Beispielsweise bie-
1076 tet sich `geom_point()` an, wenn man zwei kontinuierliche Variable darstellen möchte. Wenn man hinge-
1077 gen die Verteilung von einer kontinuirlichen Variable darstellen möchte, dann bietet sich ein Histogramm
1078 (`geom_histogram()`) oder auch eine geschätzte Dichte (z.B. `geom_density()`) an.

1079

1080 **Aufgabe 20: Abbildungen mit ggplot2**

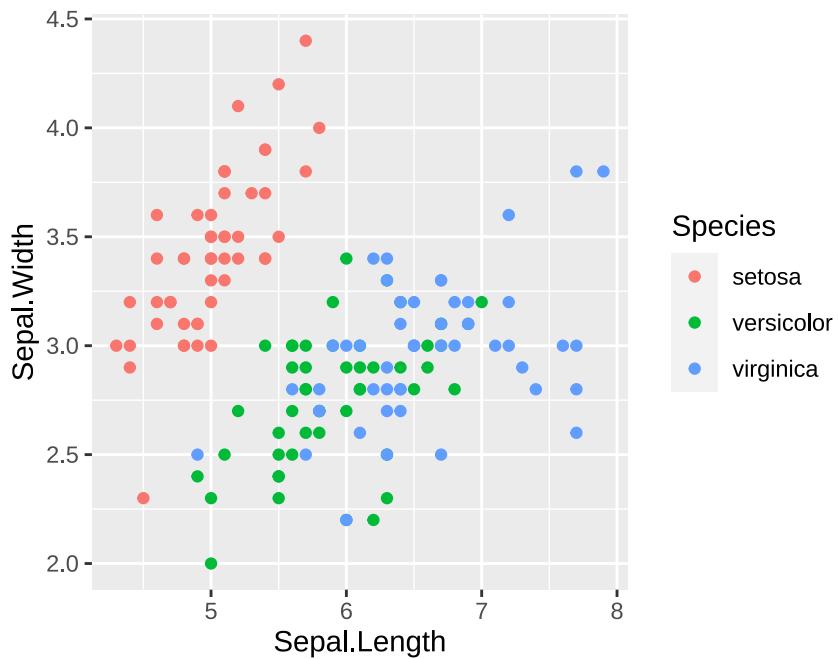
1081

1082 Verwenden Sie die den Iris Datensatz und erstellen Sie mit ggplot2 einen Plot der die Verteilung der Länge
1083 der Kelchblätter zeigt (Spalte `Sepal.Length`).

1084

1085 Eine der Stärken von ggplot2 ist, dass man den Wert unterschiedlicher Variable auf unterschiedlichen
1086 Komponenten des Plots abbilden kann. Wir haben bis jetzt ein bzw. zwei Variable auf der x- und y-Achse
1087 abgebildet. Wir können aber ein weitere Variablen verwenden um das Aussehen des Plots zu beeinflussen.
1088 Beispielsweise können wir die Farbe der Punkte (für `geom_point()`) mit dem Argument `col` beeinflussen.

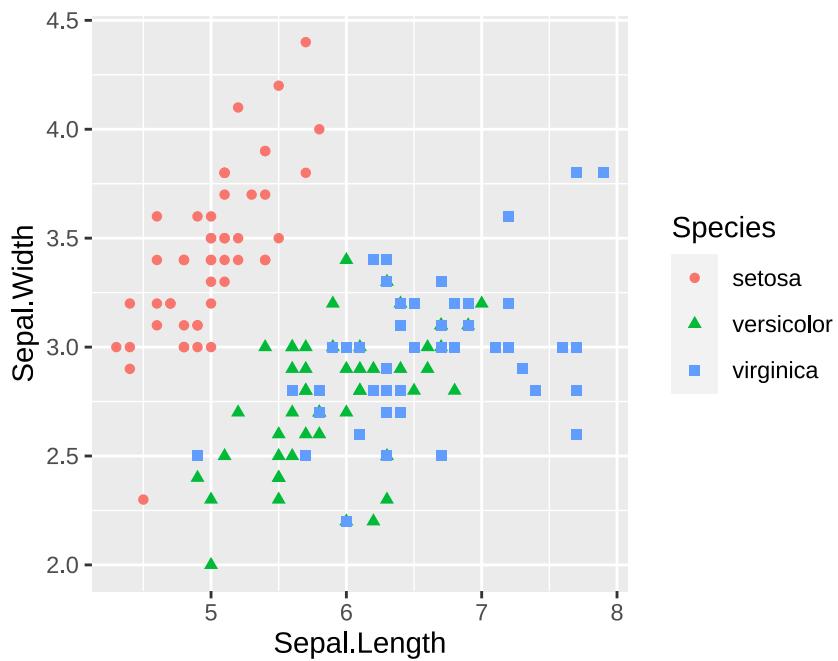
```
ggplot(iris, aes(x = Sepal.Length, y = Sepal.Width, col = Species)) +  
  geom_point()
```



1089

- 1090 Somit bekommt jede Irisart eine eigene Farbe⁸. Gleichesmaßen können wir die Punktart (`shape`), die Punktgröße (`size`) etc. anpassen. Die Legende wird automatisch an diese Farbe und Symbole angepasst.

```
ggplot(iris, aes(x = Sepal.Length, y = Sepal.Width,
                  col = Species, shape = Species)) +
  geom_point()
```

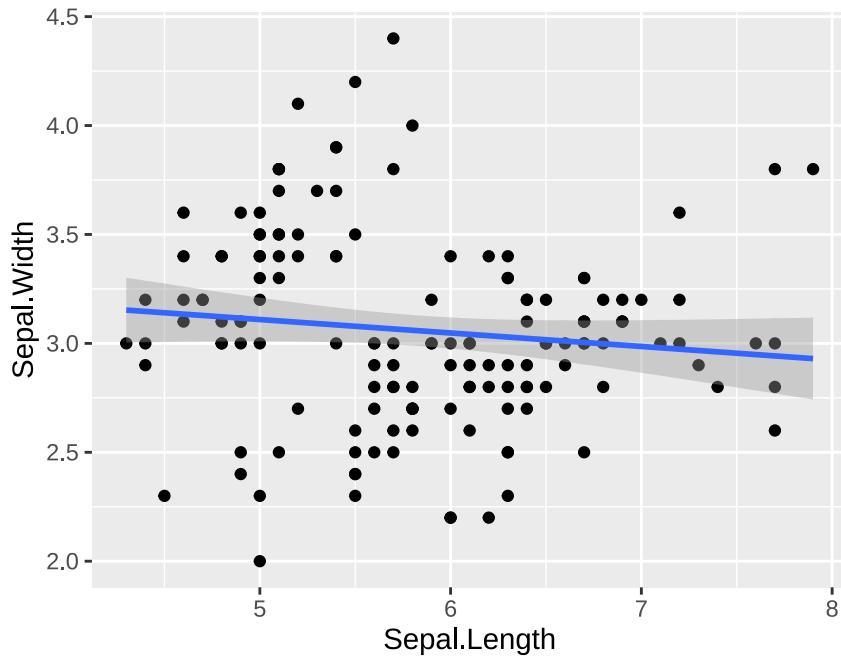


1092

⁸Natürlich könnte man auch die Farbe anpassen.

- 1093 In dem Plot ist die Information zu der Art redundant (einmal als Farbe und einmal Symbolart).
- 1094 Ein weitere sehr nützliche Geometrie ist `geom_smooth()`, die es erlaubt eine Trendlinie hinzuzufügen.

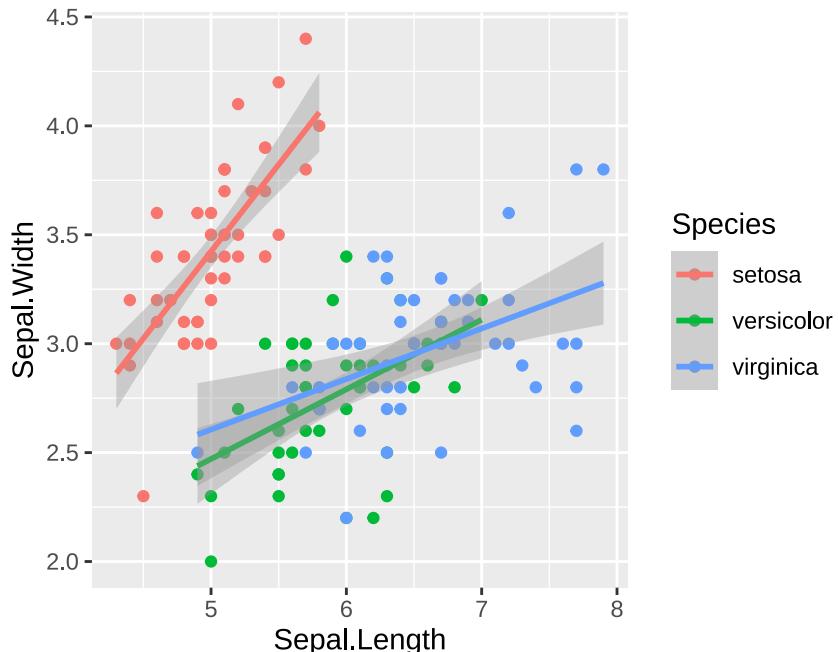
```
ggplot(iris, aes(x = Sepal.Length, y = Sepal.Width)) +
  geom_point() + geom_smooth(method = "lm")
```



1095

- 1096 Mit `method = "lm"` wird festgelegt, dass die Trendlinie gerade sein soll (es wird eine lineare Einfachregression angepasst). Wenn wird wieder eine gruppierende Variable einführen (z.B. die Beobachtungen nach Art auf die Farbe aufteilen), wir das von `geom_smooth()` berücksichtigt.

```
ggplot(iris, aes(x = Sepal.Length, y = Sepal.Width, col = Species)) +
  geom_point() + geom_smooth(method = "lm")
```



1099

1100

Aufgabe 21: Anpassen von Plots

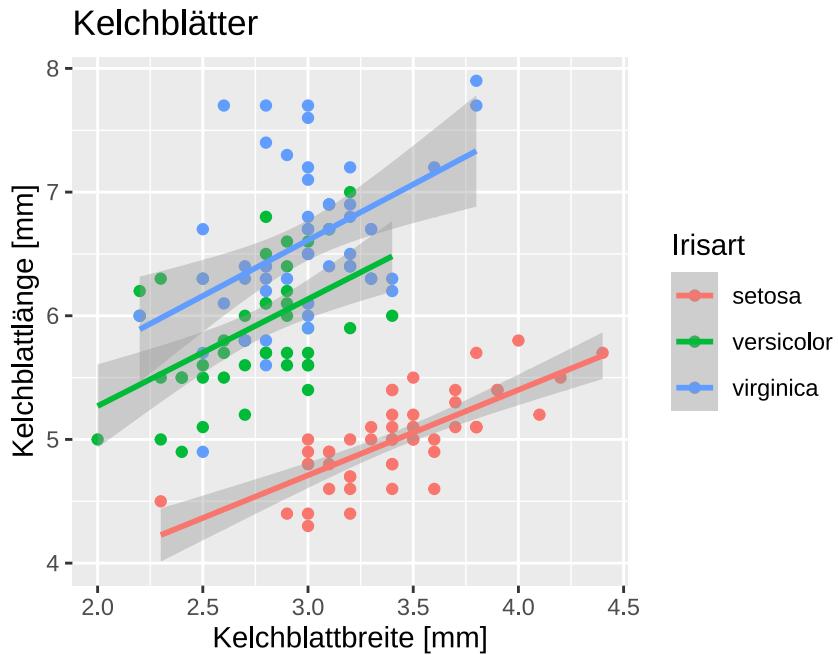
- 1103 Lesen Sie den Datensatz `data/bhd_1.txt` ein und erstellen Sie einen Boxplot für die Verteilung des BHDs
 1104 für jede Baumart. In einem zweiten Schritt verwenden Sie erst `col = gebiet` und dann `fill = gebiet`.
 1105 Welchen Unterschied stellen Sie fest?

```
dat <- read.table("data/bhd_1.txt", header = TRUE)
head(dat)
ggplot(dat, aes(art, bhd, fill = gebiet)) + geom_boxplot()
```

1106

- 1107 Mit der Funktion `labs()` werden die Beschriftungen geändert.

```
ggplot(iris, aes(x = Sepal.Width, y = Sepal.Length, color = Species)) +
  geom_point() + geom_smooth(method = "lm") +
  labs(x = "Kelchblattbreite [mm]", y = "Kelchblattlänge [mm]",
       title = "Kelchblätter", color = "Irisart")
```



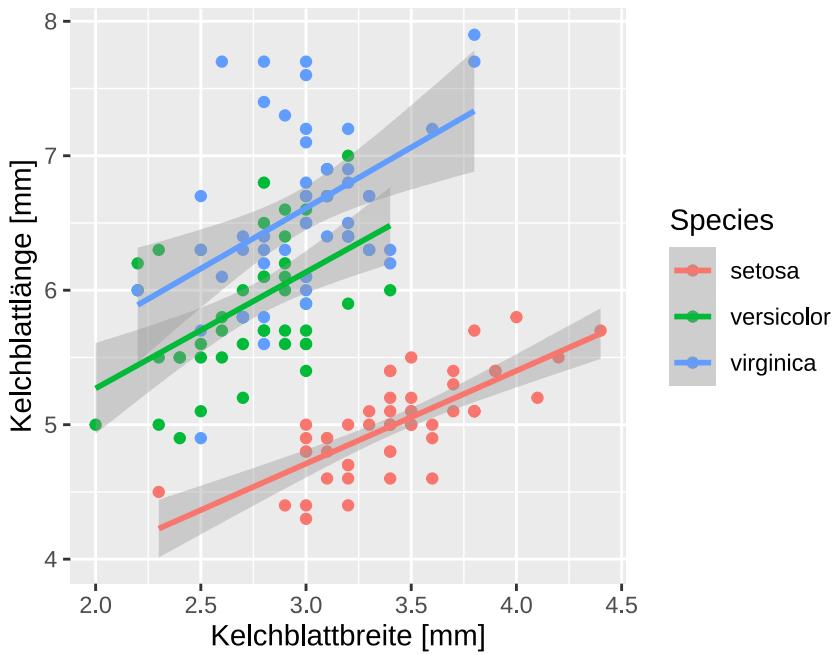
1108

- 1109 Statt einen langen Befehl zu tippen, kann ein `ggplot()` auch zwischengespeichert und wieder aufgerufen bzw.
1110 angepasst werden. Das ist vor allem sinnvoll, wenn mehrere Abbildungen auf dem selben Zwischenergebnis
1111 aufbauen sollen.

```
p1 <- ggplot(iris, aes(x = Sepal.Width, y = Sepal.Length, color = Species)) +
  geom_point() + geom_smooth(method = "lm")
```

- 1112 Wir können jetzt mit `p1` weiter arbeiten und beispielsweise eine Beschriftung hinzufügen.

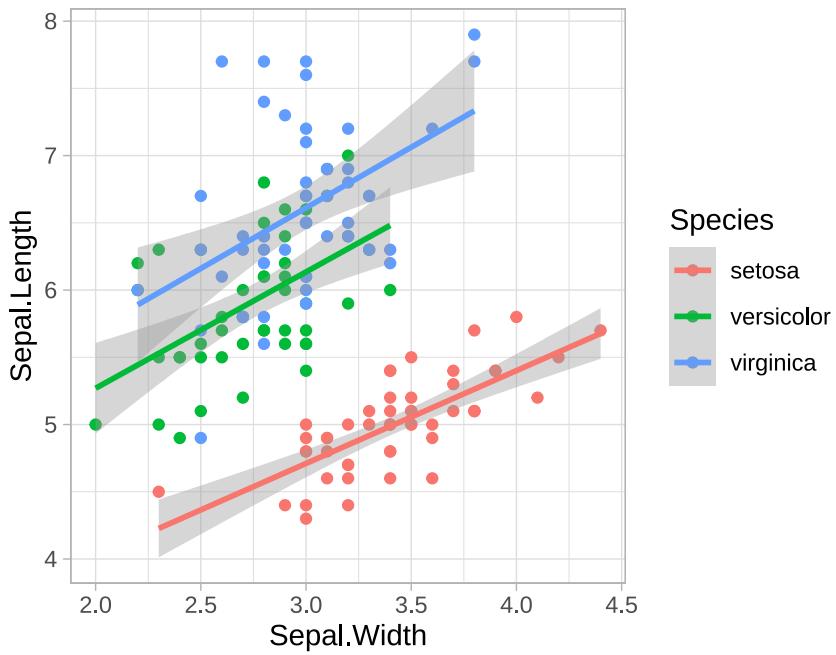
```
p1 + labs(x = "Kelchblattbreite [mm]", y = "Kelchblattlänge [mm]")
```



1113

- ¹¹¹⁴ Oder auch den ganzen Plot anpassen. Dafür gibt es *themes*. Es gibt eine Reihe von vorgefertigten *themes*
¹¹¹⁵ oder man kann diese auch selber erstellen (das ist aber nicht Teil dieses Kurses).

```
p1 + theme_light()
```

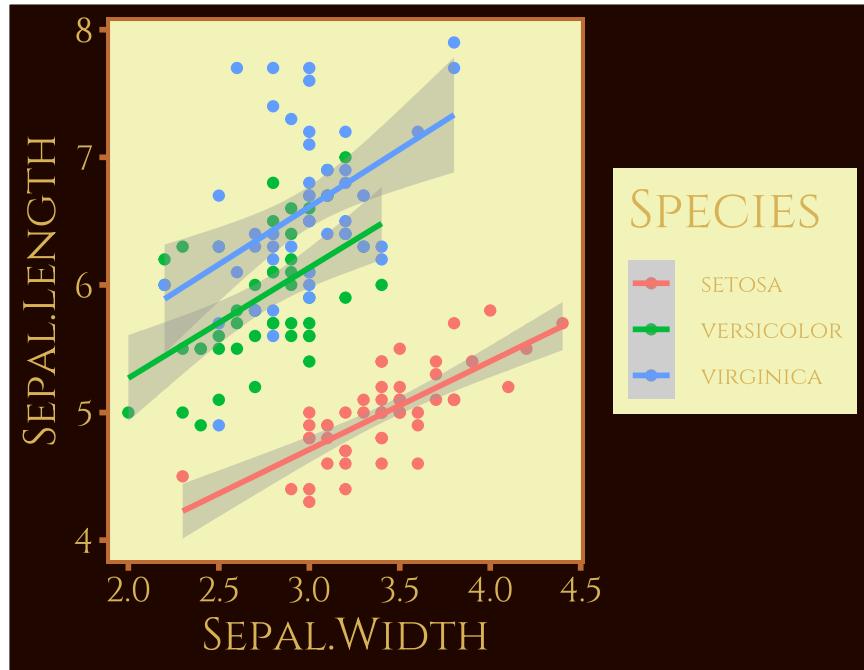


1116

- ¹¹¹⁷ Weitere *themes* sind: `theme_bw()`, `theme_linedraw()` oder `theme_dark()`. Es gibt extra Pakete die viele zusätzliche weitere *themes* anbieten. Dazu gehört z.B. das Paket `ggthemes` oder `ThemePark`. Während

1119 ggthemes hauptsächlich durchdachte Grafikkonzepte liefert, sind die Themes aus ThemePark eher Popkultur
1120 und nicht 100 %ig ernst gemeint.

```
p1 + theme_gamethrones()
```

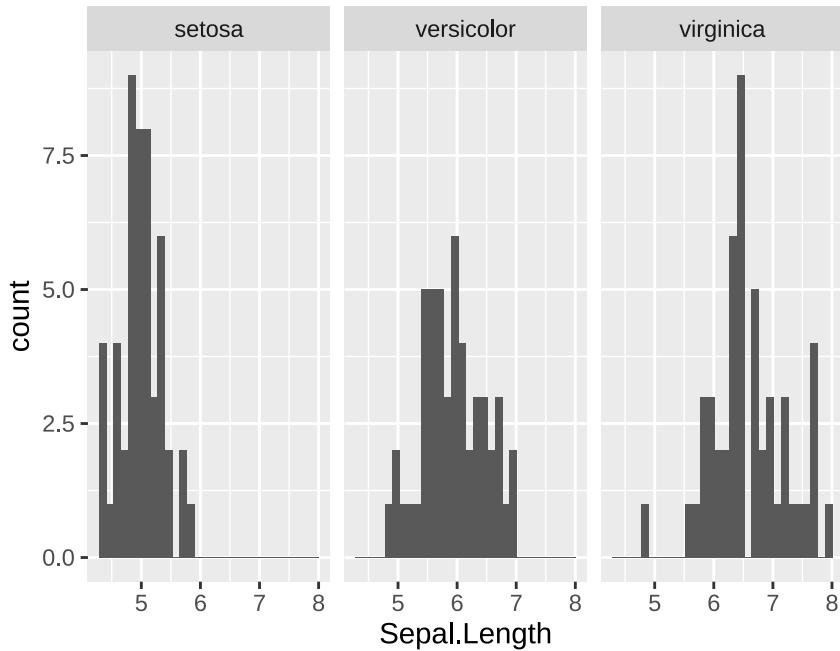


1121

1122 8.4.1 Multipanel Abbildungen

1123 Mit *ggplot2* kann man einfach Abbildungen erstellen, die mehrere Panels haben. Das bedeutet, dass eine
1124 oder mehrere weitere Variablen gibt, die einen Plot in mehrere Subplots teilt. Dafür gibt es zwei Funktion:
1125 `facet_grid()` und `facet_wrap()`.

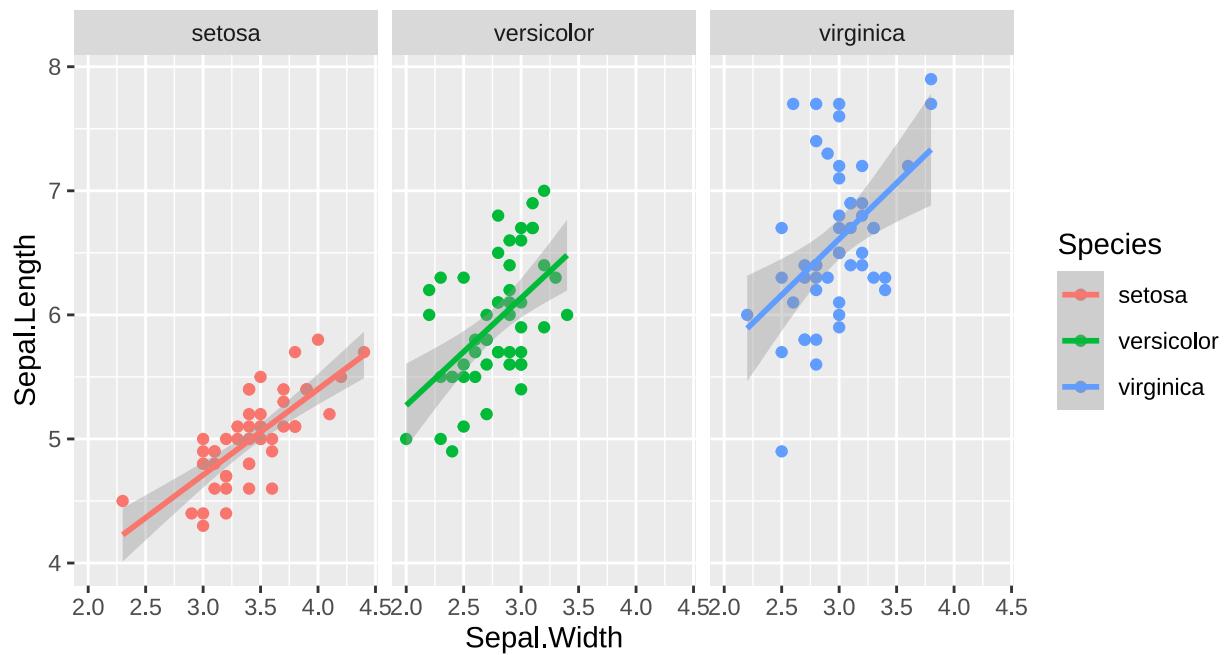
```
ggplot(iris, aes(Sepal.Length)) + geom_histogram() +  
  facet_grid(~ Species)
```



1126

1127 Die Funktion `facet_grid()` erzeugt einen *Grid* und beschriftet die Grid-Zeilen und -Spalten, während
 1128 `facet_wrap()` für jedes Panel (Diagramm) eine eigene Überschrift erzeugt. Beim Arrangieren der Diagram-
 1129 me wird der Vorteil von `ggplot2` gegenüber `base plot` besonders deutlich. Während es im `base plot` System
 1130 mühsam ist, die Abbildungen zu arrangieren, übernimmt `ggplot2` das Arrangement automatisch und fügt
 1131 sogar eine automatisch Legende hinzu. Die Achsenabschnitte werden so angepasst, dass die Daten vergleich-
 1132 bar sind.

```
ggplot(iris, aes(x = Sepal.Width, y = Sepal.Length, color = Species)) + geom_point() +
  facet_grid(~ Species) + geom_smooth(method = "lm")
```

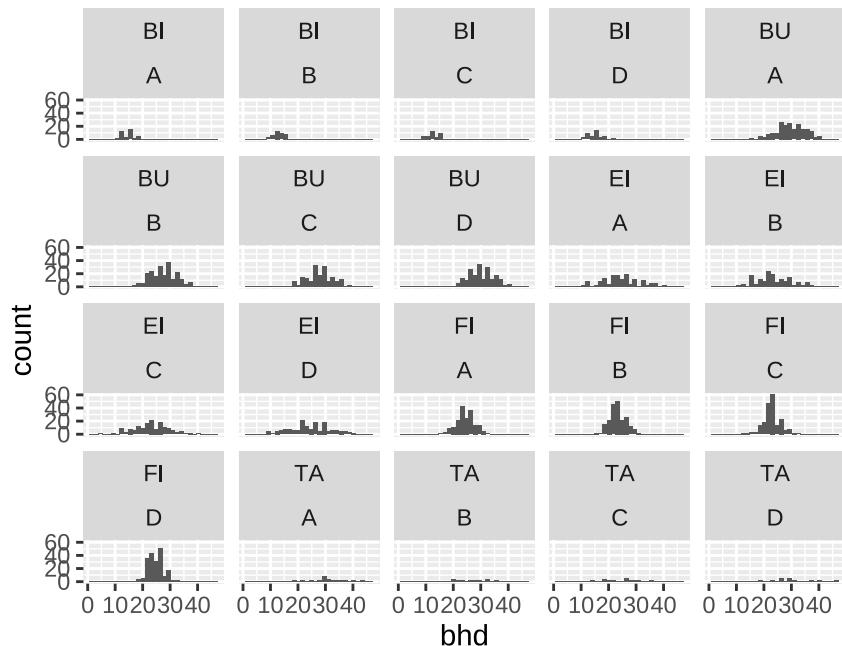


1133

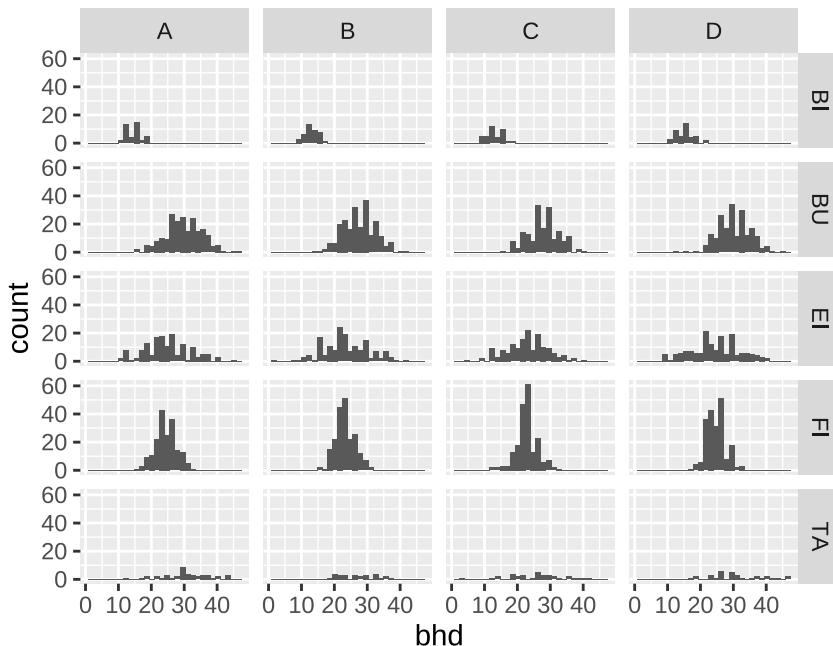
1134

1135 1136 Aufgabe 22: Multipanel Abbildungen

- 1137 Lesen Sie erneut den Datensatz daten/bhd_1.txt ein und speichern Sie diesen in die Variable (`dat_bhd`).
 1138 Erstellen Sie für jede Art und Gebiet ein Histogramm. Welche Unterschiede können Sie feststellen, wenn Sie
 1139 `facet_grid()` oder `facet_wrap()` verwenden?



1140



1141

1142 8.4.2 Plots kombinieren

1143 Es gibt Situationen, in denen **unterschiedliche** Plots miteinander kombiniert werden müssen. Im vorherigen
 1144 Abschnitt wurde dies immer anhand einer gruppierenden Kovariate gemacht. Aber es gibt auch Situations-
 1145 en, in denen das nicht möglich ist. Beispielsweise wenn ein Histogramm und ein Scatterplot vom gleichen
 1146 Datensatz zusammengefasst werden sollen. Dafür bietet sich das Paket **patchwork** an⁹.

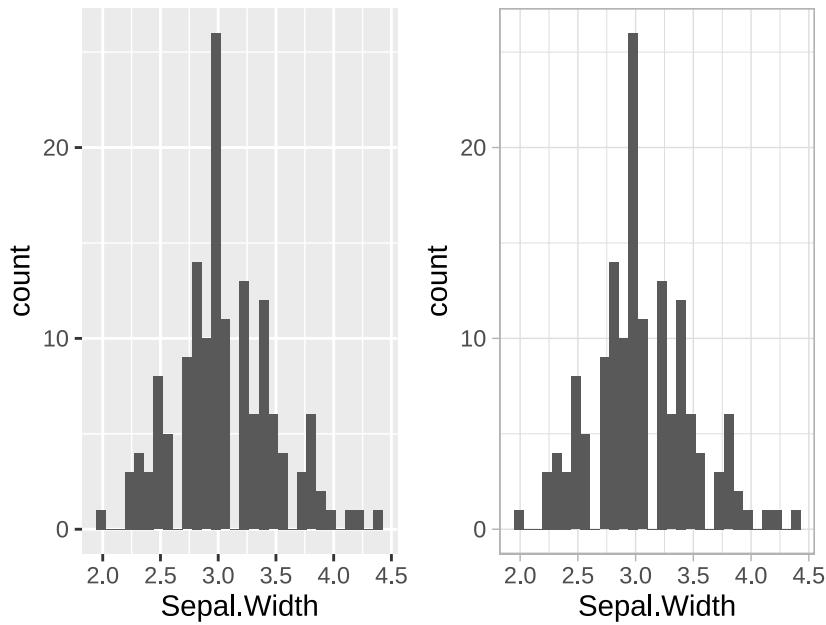
1147 Als erstes können wir zwei (oder natürlich auch mehrere Plots) erstellen. Hier unterscheiden sich die Plots
 1148 lediglich durch das Aussehen.

```
p1 <- ggplot(iris, aes(Sepal.Width)) + geom_histogram()
p2 <- p1 + theme_light()
```

1149 Dann müssen können wir diese Plots ebenfalls mit `+` zusammenfügen.

```
library(patchwork)
p1 + p2
```

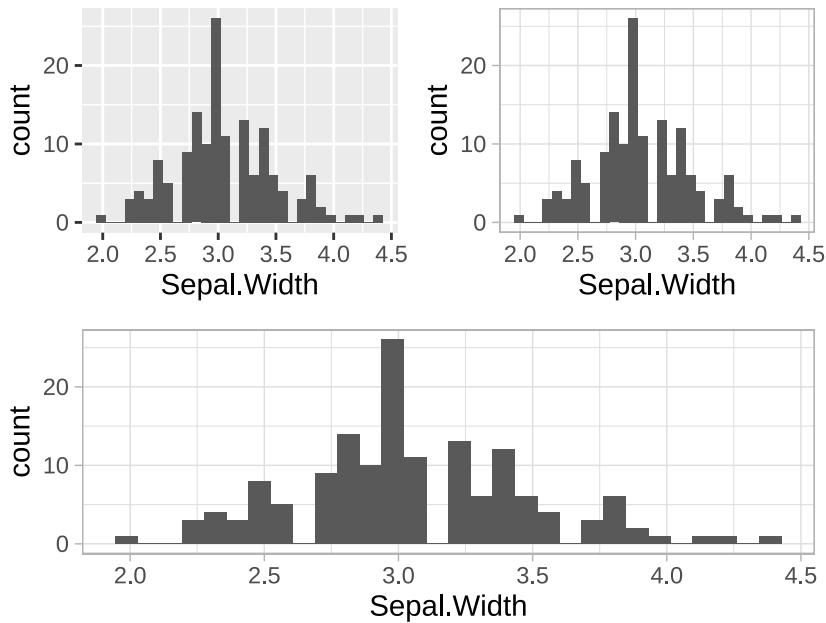
⁹Auch dieses Paket müssen Sie einmalig mit `install.packages("patchwork")` installieren.



1150

1151 Natürlich können auch weitere Plots hinzugefügt werden (auch in unterschiedlichen Dimensionen):

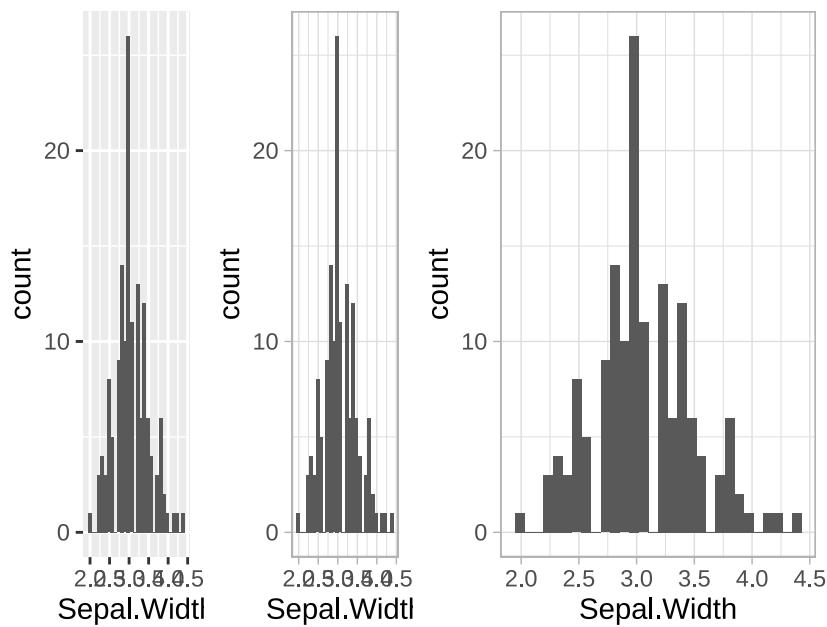
(p1 + p2) / p2



1152

1153 Des weiteren können mit | auch Plots gegenüber gestellt werden.

```
(p1 + p2) | p2
```



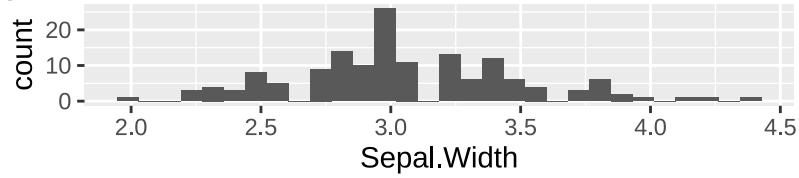
1154

1155 Weitere Optionen können mit `plot_layout()` und `plot_annotation()` angepasst werden. Mit
 1156 `plot_layout()` kann die Anordnung der Plots bestimmt werden (z.B. über die Argumente `nrow` und
 1157 `ncol`), sowie deren *relative* Größe (über die Argumente `widths` und `heights`). Mit der Funktion
 1158 `plot_annotation()` können zusätzliche Beschriftungen hinzugefügt werden, wie beispielsweise eine Titel
 1159 (Argument `title`) oder ein Buchstabe/Zahl für jedes Element (Argument `tag_levels`).

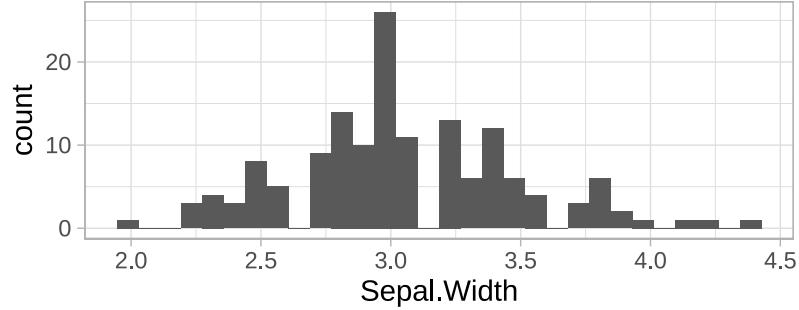
```
p1 + p2 +
  plot_layout(ncol = 1, heights = c(0.3, 0.7)) +
  plot_annotation(title = "Zwei Histogramme", tag_levels = "A")
```

Zwei Histogramme

A



B

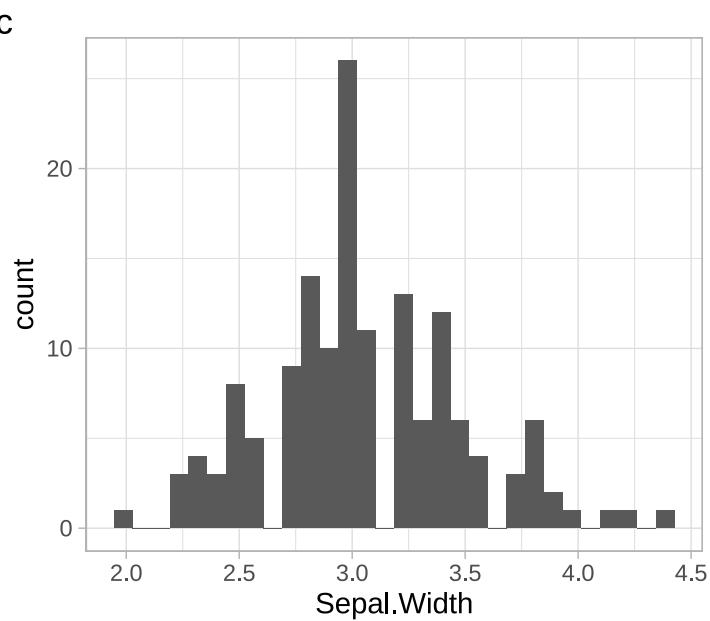
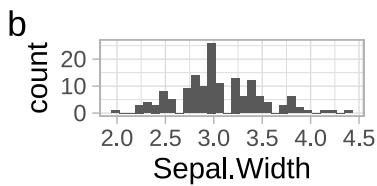
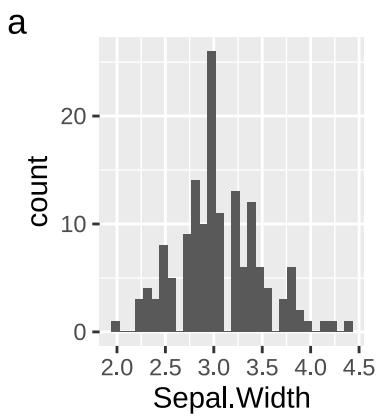


1160

1161

1162 1163 Aufgabe 23: Mehrere Plots zusammenfügen

- 1164 Versuchen Sie die folgende Zusammenstellung der Plots nachzumachen:



1165

1166 8.4.3 Speichern von plots

1167 Sie können mit `ggsave()` eine zwischen-gespeicherte Abbildung exportieren, indem Sie den Variablenamen
1168 übergeben. Wenn Sie keine Variable übergeben, wird automatisch die letzte Abbildung gespeichert. Das
1169 Dateiformat wird aus dem Dateinamen übernommen. Die Größe der Inhalte können Sie sehr leicht mit den
1170 Argumenten `width` und `height` oder `scale` gesteuert werden.

```
ggsave("letzteAbb.png")
ggsave(p1, "zwischengespeicherteAbb.png")
```

1171 9 Mit Daten arbeiten

1172 9.1 dplyr eine Einführung

1173 `dplyr` ist eine Erweiterung von R (ein Paket), die das Ziel hat, den Umgang mit Daten einfacher und
1174 schneller zu machen.

1175 `dplyr` definiert 5 Verben, um mit Daten zu arbeiten. Diese sind:

- 1176 • `filter`
- 1177 • `select`
- 1178 • `arragne`
- 1179 • `mutate`
- 1180 • `summarise`

```
dat <- data.frame(id = 1:5,
                    plot = c(1, 1, 2, 2, 3),
                    bhd = c(50, 29, 13, 23, 25),
                    alter = c(10, 30, 31, 24, 25))
```

1181 Damit die Funktionen aus `dplyr` verwendet werden können, müssen wir als erstes das Paket `dplyr` laden.
1182 The online Resource [R for Data Science](#) gibt einen sehr guten, tieferen Eindruck in die Arbeit mit `dplyr`
1183 und erklärt auch die `dplyr` Philosophie noch Mal sehr genau für Beginner.

```
library(dplyr)
```

1184 Sollte dies zu einer Fehlermeldung führen, dann müssen Sie das Paket `dplyr` erst installieren. Dafür müssen
1185 Sie `einmalig install.packages("dplyr")` installieren.

1186 `dplyr` stellt unterschiedliche Funktionen zum Arbeiten mit Daten zur Verfügung. Es gibt fünf Grundfunktionen
1187 für die am häufigsten vorkommenden Operationen. Mit der Funktion `filter()` können unterschiedliche
1188 Beobachtungen gefiltert werden:

```
filter(dat, bhd > 10)
```

```
1189 ##   id plot bhd alter
1190 ## 1   1    1  50   10
1191 ## 2   2    1  29   30
1192 ## 3   3    2  13   31
1193 ## 4   4    2  23   24
1194 ## 5   5    3  25   25
```

1195 Es können auch mehrere Spalten verwendet werden.

```
filter(dat, bhd > 10, bhd < 40)
```

```
1196 ## id plot bhd alter
1197 ## 1 2 1 29 30
1198 ## 2 3 2 13 31
1199 ## 3 4 2 23 24
1200 ## 4 5 3 25 25
```

1201 Natürlich kann genau das gleiche Ergebnis mit dem ‘normalen’ R erreicht werden, dies wäre dann:

```
dat[dat$bhd > 10 & dat$bhd < 40, ]
```

```
1202 ## id plot bhd alter
1203 ## 2 2 1 29 30
1204 ## 3 3 2 13 31
1205 ## 4 4 2 23 24
1206 ## 5 5 3 25 25
```

1207 Eine weitere Funktion aus dem Paket `dplyr` ist `select()`. Damit können Spalten aus einem `data.frame` ausgewählt werden. Dabei können auch die Spaltennamen unbenannt werden.

```
select(dat, bhd)
```

```
1209 ## bhd
1210 ## 1 50
1211 ## 2 29
1212 ## 3 13
1213 ## 4 23
1214 ## 5 25
```

```
select(dat, bhd, id)
```

```
1215 ## bhd id
1216 ## 1 50 1
1217 ## 2 29 2
1218 ## 3 13 3
1219 ## 4 23 4
1220 ## 5 25 5
```

```
select(dat, BHD = bhd, id)
```

```

1221 ##   BHD id
1222 ## 1 50 1
1223 ## 2 29 2
1224 ## 3 13 3
1225 ## 4 23 4
1226 ## 5 25 5

```

1227 Mit der Funktion `arrange()` können die Beobachtungen in einem `data.frame` sortiert werden.

```
arrange(dat, bhd)
```

```

1228 ##   id plot bhd alter
1229 ## 1 3 2 13 31
1230 ## 2 4 2 23 24
1231 ## 3 5 3 25 25
1232 ## 4 2 1 29 30
1233 ## 5 1 1 50 10

```

1234 Mit der Funktion `desc()` kann die Anordnung in absteigender Reihenfolge sortiert werden.

```
arrange(dat, desc(bhd))
```

```

1235 ##   id plot bhd alter
1236 ## 1 1 1 50 10
1237 ## 2 2 1 29 30
1238 ## 3 5 3 25 25
1239 ## 4 4 2 23 24
1240 ## 5 3 2 13 31

```

1241 Mit der Funktion `mutate()` kann man eine neue Spalte hinzufügen.

```
mutate(dat, bhd_mm = bhd * 10, fl = pi * (bhd/2)^2)
```

```

1242 ##   id plot bhd alter bhd_mm          fl
1243 ## 1 1 1 50 10 500 1963.4954
1244 ## 2 2 1 29 30 290 660.5199
1245 ## 3 3 2 13 31 130 132.7323
1246 ## 4 4 2 23 24 230 415.4756
1247 ## 5 5 3 25 25 250 490.8739

```

```
mutate(dat, mean_bhd = mean(bhd))
```

```

1248 ##   id plot bhd alter mean_bhd
1249 ## 1   1     1   50    10    28
1250 ## 2   2     1   29    30    28
1251 ## 3   3     2   13    31    28
1252 ## 4   4     2   23    24    28
1253 ## 5   5     3   25    25    28

```

1254 Mit der Funktion `summarise()` können Spalten zusammengefasst werden.

```

summarise(
  dat,
  mean_bhd = mean(bhd),
  mean_sd = sd(bhd)
)

1255 ##   mean_bhd  mean_sd
1256 ## 1          28 13.63818

```

1257

1258 Aufgabe 24: Datenmanipulation mit dplyr

- 1260 1. Laden Sie den Datensatz `data/bhd_1.txt`
- 1261 2. Berechnen Sie folgende Werte für alle Einträge und speichern Sie die Ergebnisse in `erg1`

- 1262 • mittlerer `bhd`
- 1263 • maximales `alter`
- 1264 • die Standardabweichung des BHDs
- 1265 • die Anzahl Bäume mit einem BHD > 30

1266 9.2 Arbeiten mit gruppierten Daten

1267 Zusätzlich können `mutate` und `summarise` auch auf gruppierte Daten angewendet werden. Dafür müssen
 1268 wir erst die Funktion `group_by()` aufrufen und als Argumente die Variablen übergeben, die die Gruppen
 1269 definieren.

```

dat1 <- group_by(dat, plot)
mutate(dat, bhd_m = mean(bhd)) # bhd über alle Bäume

```

```

1270 ##   id plot bhd alter bhd_m
1271 ## 1   1     1   50    10    28
1272 ## 2   2     1   29    30    28

```

```

1273 ## 3 3 2 13 31 28
1274 ## 4 4 2 23 24 28
1275 ## 5 5 3 25 25 28

```

```
mutate(dat1, bhd_m = mean(bhd)) # bhd pro Plot
```

```

1276 ## # A tibble: 5 x 5
1277 ## # Groups:   plot [3]
1278 ##       id   plot   bhd alter bhd_m
1279 ##     <int> <dbl> <dbl> <dbl> <dbl>
1280 ## 1     1     1    50    10  39.5
1281 ## 2     2     1    29    30  39.5
1282 ## 3     3     2    13    31  18
1283 ## 4     4     2    23    24  18
1284 ## 5     5     3    25    25  25

```

```
summarise(dat, bhd_m = mean(bhd))
```

```

1285 ##   bhd_m
1286 ## 1    28

```

```
summarise(dat1, bhd_m = mean(bhd))
```

```

1287 ## # A tibble: 3 x 2
1288 ##   plot bhd_m
1289 ##     <dbl> <dbl>
1290 ## 1     1  39.5
1291 ## 2     2  18
1292 ## 3     3  25

```

1293

Aufgabe 25: dplyr mit gruppierten Daten

- 1296 1. Laden Sie den Datensatz `data/bhd_1.txt`
 1297 2. Berechnen Sie für jede Baumart und Standort die folgenden Werte:

1298 • mittlerer `bhd`
 1299 • maximales `alter`
 1300 • die Standardabweichung des BHDS
 1301 • die Anzahl Bäume mit einem BHD > 30

- 1302 3. Ordnen Sie das Ergebnis nach Baumart und BHD.

1303 **9.3 pipes oder %>%**1304 Mit *Pipes* (%>%) kann man das Ergebnis einer Funktion einfach an eine nachfolgende Funktion weiterreichen.

```
a <- c(5, 3, 2, NA)
```

1305 Wir kennen bis jetzt:

```
mean(na.omit(a))
```

1306 ## [1] 3.333333

1307 Mit *Pipes*, die durch das Symbol %>% dargestellt werden¹⁰, können wir das etwas vereinfachen und nacheinander schreiben:

```
na.omit(a) %>% mean()
```

1309 ## [1] 3.333333

1310 Oder sogar

```
a %>% na.omit() %>% mean()
```

1311 ## [1] 3.333333

1312

1313 **Aufgabe 26: Pipes %>%**

1315 Wiederholen Sie die letzte Aufgabe, aber diesmal ohne Zwischenergebnisse zu speichern:

1316 1. Laden Sie den Datensatz data/bhd_1.txt.

1317 2. Berechnen Sie für jede Baumart und Standort die folgenden Werte:

- 1318 • mittlerer bhd
- 1319 • maximales alter
- 1320 • die Standardabweichung des BHDs
- 1321 • die Anzahl Bäume mit einem BHD > 30

1322 3. Ordnen Sie das Ergebnis nach Baumart und BHD.

¹⁰In RStudio kann %>% mit der Tastenkombination Strg + Umschalt + m ([Strg]+[Umschalt]+[m]) eingefügt werden.

1323 9.4 Joins

1324 Eine weitere häufige Aufgabe beim Daten Management ist es Daten zusammenzuführen. Nehmen Sie an,
 1325 dass wir folgende Aufnahmen gemacht haben

```
aufnahmen <- data.frame(
  id = 1:3,
  bhd = c(20, 31, 74)
)
```

1326 und jeder Baum lediglich mit einer `id` versehen wurde. In einer zweiten Tabelle wurden dann weitere Daten
 1327 zu Bäumen gespeichert (z.B. die Art, das Studiengebiet usw).

```
metadaten <- data.frame(
  id = 2:4,
  art = c("Ta", "Bu", "Bu"),
  gebiet = c("A", "B", "B")
)
```

1328 Ziel ist es jetzt die Bäume aus `aufnahmen` mit den Informationen aus den `metadaten` zu verbinden. Dazu
 1329 dient `id` als Bindeglied (oft auch Schlüssel genannt).

1330 Dazu gibt es vier Möglichkeiten.

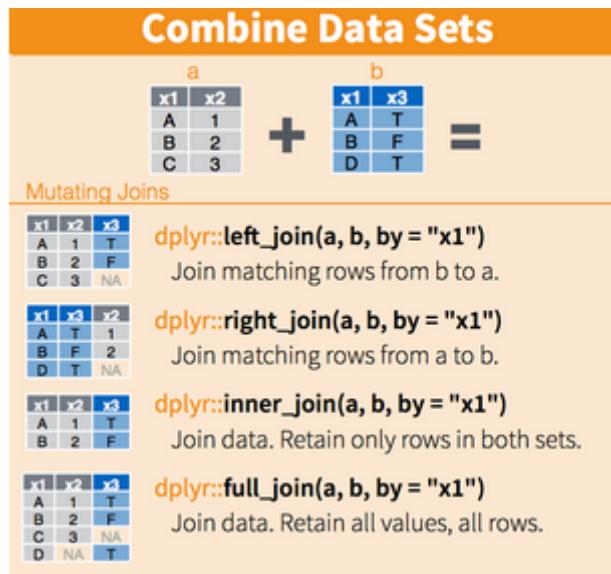


Abbildung 8: Joins (Quelle Rstudio)

1331 Zur Durchführung gibt es in base R die Funktion `merge()`. Wir werden aber gleich die Funktionen aus dem
 1332 Paket `dplyr` verwenden.

```

library(dplyr)
left_join(aufnahmen, metadaten, by = "id")

1333 ##   id bhd  art gebiet
1334 ## 1  1  20 <NA>    <NA>
1335 ## 2  2  31   Ta      A
1336 ## 3  3  74   Bu      B

right_join(aufnahmen, metadaten, by = "id")

1337 ##   id bhd art gebiet
1338 ## 1   2  31  Ta      A
1339 ## 2   3  74  Bu      B
1340 ## 3   4  NA  Bu      B

inner_join(aufnahmen, metadaten, by = "id")

1341 ##   id bhd art gebiet
1342 ## 1   2  31  Ta      A
1343 ## 2   3  74  Bu      B

full_join(aufnahmen, metadaten, by = "id")

1344 ##   id bhd  art gebiet
1345 ## 1   1  20 <NA>    <NA>
1346 ## 2   2  31   Ta      A
1347 ## 3   3  74   Bu      B
1348 ## 4   4  NA  Bu      B

1349 by kann auch unterschiedliche Spalten in den beiden data.frames ansprechen:

metadaten <- data.frame(
  baum_id = 2:4,
  art = c("Ta", "Bu", "Bu"),
  gebiet = c("A", "B", "B")
)

left_join(aufnahmen, metadaten, by = c("id" = "baum_id"))

1350 ##   id bhd  art gebiet
1351 ## 1   1  20 <NA>    <NA>
1352 ## 2   2  31   Ta      A
1353 ## 3   3  74   Bu      B

```

1354

1355 **Aufgabe 27: Verbinden von Daten**

- 1357 • Lesen Sie die Datensätze `daten/bhd_2.txt` und `daten/bhd_2_meta.txt` ein.
- 1358 • Stellen Sie sicher, dass es für den `bhd` keine fehlenden Werte gibt (entfernen sie entsprechende Zeilen)
- 1359 • Fügen Sie zu den Metadaten (gespeichert in `bhd_2_meta`) die Anzahl Bäume und den mittleren `bhd`
- 1360 hinzu pro Gebiet.

1361 **9.5 ‘long’ and ‘wide’ Datenformate**

1362 Unter anderem Wickham (2014) empfiehlt das Prinzip von *tidy* Data. Nach diesem Prinzip sollten Daten wie
1363 folgt organisiert sein:

- 1364 • Jede Zeile ist ein Merkmalsträger/Subjekt/Objekt (z. B. eine Person oder ein Baum).
- 1365 • Jede Spalte ist eine Variable (ein Merkmal), die den Merkmalsträger beschreibt.
- 1366 • In jeder Zelle ist genau ein Wert (Merkmalausprägung), nämlich der Wert, der Variable für den Merk-
- 1367 malsträger.

1368 Zum Beispiel enthalten Spaltennamen oft Informationen, die eigentlich in einer Variable gespeichert werden
1369 sollten. Folgendes Beispiel gibt die BHD Messung von 3 Bäumen in 3 Jahren wieder. Der Vorteil von *tidy*
1370 Daten ist, dass viele Funktionen diese Form erwarten. Somit müssen sie Ihre Daten nur ein Mal organisieren
1371 und können fast alle Analysen durchführen.

```
dat <- tibble(
  id = 1:3,
  bhd2015 = c(30, 31, 32),
  bhd2016 = c(31, 31, 33),
  bhd2017 = c(34, 32, 33)
)
```

1372 Der `tibble` aus dem Paket `tibble` ist ein moderner Data Frame, der sich in seinen Funktionen in das
1373 `tidy` Universum einfügt. Wir können auch einfach das Paket `tidyverse` laden, um alle Pakete, die Teil des
1374 `tidy` Universums sind gleichzeitig zu laden. Der `tibble` beinhaltet alle Funktionen, die der Data Frame
1375 auch beinhaltet und kann auch für genau die gleichen Zwecke verwendet werden. Der `tibble` hat u. A. eine
1376 modernere Darstellung im Konsolenoutput.

1377 Diese Daten sind jetzt im `wide`-Format gespeichert und folglich nicht `tidy`, weil Information über die Daten
1378 (nämlich das Jahr der Aufnahme in den Spaltennamen gespeichert sind). Besser wäre eine Struktur mit
1379 nur drei Spalten: `id`, `jahr` und `bhd`. Um die Daten in so eine Struktur zu bringen, gibt es die Funktion
1380 `pivot_longer()` aus dem Paket `tidyverse`, die ebenfalls Teil des `tidyverse` ist.

```
library(tidyr)
dat1 <- pivot_longer(dat, cols = bhd2015 : bhd2017)
dat1
```

```
1381 ## # A tibble: 9 x 3
1382 ##       id name    value
1383 ##   <int> <chr>   <dbl>
1384 ## 1     1 bhd2015     30
1385 ## 2     1 bhd2016     31
1386 ## 3     1 bhd2017     34
1387 ## 4     2 bhd2015     31
1388 ## 5     2 bhd2016     31
1389 ## 6     2 bhd2017     32
1390 ## 7     3 bhd2015     32
1391 ## 8     3 bhd2016     33
1392 ## 9     3 bhd2017     33
```

1393 Wenn wir die Spalten für die Variable und den Wert gleich sinnvoll benennen möchten, können wir das über
 1394 die Argumente `names_to` und `values_to` machen.

```
dat1 <- pivot_longer(dat, cols = bhd2015 : bhd2017, names_to = "jahr", values_to = "bhd")
dat1
```

```
1395 ## # A tibble: 9 x 3
1396 ##       id jahr    bhd
1397 ##   <int> <chr>   <dbl>
1398 ## 1     1 bhd2015     30
1399 ## 2     1 bhd2016     31
1400 ## 3     1 bhd2017     34
1401 ## 4     2 bhd2015     31
1402 ## 5     2 bhd2016     31
1403 ## 6     2 bhd2017     32
1404 ## 7     3 bhd2015     32
1405 ## 8     3 bhd2016     33
1406 ## 9     3 bhd2017     33
```

1407 Analog zu der Funktion `pivot_longer()` gibt es auch die Funktion `pivot_wider()`, um vom Daten vom
 1408 long-Format ins wide-Format zu transformieren.

```
pivot_wider(dat1, names_from = jahr, values_from = bhd)
```

```
1409 ## # A tibble: 3 x 4
1410 ##       id bhd2015 bhd2016 bhd2017
```

```

1411 ## <int> <dbl> <dbl> <dbl>
1412 ## 1     1     30    31    34
1413 ## 2     2     31    31    32
1414 ## 3     3     32    33    33

```

1415

Aufgabe 28: Zeitliche Verlauf von BHDs

1418 In der Datei `bhd_3.csv` befinden sich gemessene BHDs (in cm) von unterschiedlichen Bäumen zu unterschiedlichen Zeitpunkten. Erstellen Sie ein Liniendiagramm, das den zeitlichen (x-Achse) Verlauf der BHDs
1419 für die unterschiedlichen Bäume darstellt.
1420

9.6 Auswählen von Variablen

1422 Sobald die Datensätze etwas umfangreicher werden (d. h., es gibt mehrere Spalten in einem `data.frame`),
1423 können innerhalb vieler `dplyr`-Funktionen spezielle Funktionen verwendet werden, um Variablen auszuwählen.
1424

1425 Wenn die genaue Position der Spalten bekannt ist, kann man mit dem `:`-Operator und der Position Spalten
1426 auswählen:

```
iris %>% select(1 : 3) %>% head(3)
```

```

1427 ## Sepal.Length Sepal.Width Petal.Length
1428 ## 1     5.1     3.5     1.4
1429 ## 2     4.9     3.0     1.4
1430 ## 3     4.7     3.2     1.3

```

1431 Diese Vorgehensweise kann gehährlich sein, da sich manchmal Spalten verschieben und sich somit die Positionen ändern. Es besser Spalten immer explizit mit Namen statt mit Nummern anzusprechen.
1432

```
iris %>% select(Sepal.Length, Sepal.Width, Petal.Length) %>% head(3)
```

```

1433 ## Sepal.Length Sepal.Width Petal.Length
1434 ## 1     5.1     3.5     1.4
1435 ## 2     4.9     3.0     1.4
1436 ## 3     4.7     3.2     1.3

```

1437 `select()` erlaubt es, auch hier den `:`-Operator zu verwenden:

```
iris %>% select(Sepal.Length:Petal.Length) %>% head(3)
```

```
1438 ## Sepal.Length Sepal.Width Petal.Length
1439 ## 1      5.1      3.5      1.4
1440 ## 2      4.9      3.0      1.4
1441 ## 3      4.7      3.2      1.3
```

1442 Es gibt auch einige spezielle Funktionen, um Spalten innerhalb eines `select()`-Aufrufs auszuführen:

- 1443 • `starts_with()`: Hier kann man ein Muster angeben, mit dem ein Text anfangen muss.
- 1444 • `ends_with()`: Diese Funktion ist analog zu `starts_with()`, jetzt wird aber am Ende des Spaltennamens nach dem Muster gesucht.
- 1445 • `contains()`: Hier kann ein Muster übergeben werden, das irgendwo im Spaltennamen sein muss.
- 1446 • `everything()`: Mit dieser Funktion werden alle Spalten ausgewählt.
- 1447 • `last_col()`: Mit dieser Funktion wird nur die letzte Spalte ausgewählt (dass ist die Spalte, die ganz rechts ist).

1450 Sämtliche Auswahlen können mit - umgekehrt werden.

```
iris %>% select(starts_with("Sepal")) %>% head(3)
```

```
1451 ## Sepal.Length Sepal.Width
1452 ## 1      5.1      3.5
1453 ## 2      4.9      3.0
1454 ## 3      4.7      3.2
```

```
iris %>% select(-starts_with("Sepal")) %>% head(3)
```

```
1455 ## Petal.Length Petal.Width Species
1456 ## 1      1.4      0.2 setosa
1457 ## 2      1.4      0.2 setosa
1458 ## 3      1.3      0.2 setosa
```

1459 `select()` bietet auch noch die Möglichkeit, Spaltennamen zu ändern.

```
iris %>% select(sep_width = Sepal.Width) %>% head(3)
```

```
1460 ## sep_width
1461 ## 1      3.5
1462 ## 2      3.0
1463 ## 3      3.2
```

1464

1465 **Aufgabe 29: Auswählen von Spalten**

1467 In der Datei `messungen_1.csv` sind Messungen von zwei Sensoren enthalten für die ersten vier Monate eines
1468 Jahres. Führen Sie folgende Abfragen durch:

- 1469 1. Wählen Sie alle Messungen für Januar aus.
1470 2. Wählen Sie alle Messungen für Januar und März aus.

1471 **9.7 Einzelne Beobachtungen abfragen (`slice()`)**

1472 Mit dem Befehl `slice()` kann man einzelne Beobachtungen (= Zeilen) aus einem `data.frame` abfragen.

```
slice(iris, c(1, 9, 18))
```

1473 ## Sepal.Length Sepal.Width Petal.Length Petal.Width Species
1474 ## 1 5.1 3.5 1.4 0.2 setosa
1475 ## 2 4.4 2.9 1.4 0.2 setosa
1476 ## 3 5.1 3.5 1.4 0.3 setosa

1477 Davon gibt es drei nützliche Varianten: 1) `slice_head()` und `slice_tail()`; 2) `slice_max()` und
1478 `slice_min()`; 3) `slice_random()`.

1479 `slice_head()` und `slice_tail()` sind analog zu `head()` und `tail()`, aber mit dem entscheidenden Unter-
1480 schied, dass Gruppierungen berücksichtigt werden. Wenn keine Gruppierung in den Daten vorhanden ist,
1481 gibt es keinen Unterschied.

```
iris %>% head(n = 2)
```

1482 ## Sepal.Length Sepal.Width Petal.Length Petal.Width Species
1483 ## 1 5.1 3.5 1.4 0.2 setosa
1484 ## 2 4.9 3.0 1.4 0.2 setosa

```
iris %>% slice_head(n = 2)
```

1485 ## Sepal.Length Sepal.Width Petal.Length Petal.Width Species
1486 ## 1 5.1 3.5 1.4 0.2 setosa
1487 ## 2 4.9 3.0 1.4 0.2 setosa

1488 Sobald jedoch eine gruppierende Variable eingeführt wird, gibt `slice_head()` die ersten `n` Beobachtungen
1489 für jede Gruppe zurück und `head()` für den gesamten Datensatz.

```

# base head
iris %>% group_by(Species) %>%
  head(n = 2)

1490 ## # A tibble: 2 x 5
1491 ## # Groups:   Species [1]
1492 ##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
1493 ##     <dbl>      <dbl>      <dbl>      <dbl> <fct>
1494 ## 1       5.1        3.5       1.4       0.2  setosa
1495 ## 2       4.9        3         1.4       0.2  setosa

# dplyr slice_head
iris %>% group_by(Species) %>%
  slice_head(n = 2)

1496 ## # A tibble: 6 x 5
1497 ## # Groups:   Species [3]
1498 ##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
1499 ##     <dbl>      <dbl>      <dbl>      <dbl> <fct>
1500 ## 1       5.1        3.5       1.4       0.2  setosa
1501 ## 2       4.9        3         1.4       0.2  setosa
1502 ## 3       7          3.2       4.7       1.4  versicolor
1503 ## 4       6.4        3.2       4.5       1.5  versicolor
1504 ## 5       6.3        3.3       6         2.5  virginica
1505 ## 6       5.8        2.7       5.1       1.9  virginica

1506 slice_tail() funktioniert analog zu slice_head() mit dem einzigen Unterschied, dass nicht die ersten n
1507 Zeilen zurück gegeben werden sondern die letzten n Zeilen.
1508 slice_max() und slice_min() geben die Beobachtung mit dem maximalen bzw. minimalen Wert einer
1509 Variablen zurück. Auch hier werden Gruppen berücksichtigt.

  iris %>% slice_max(Sepal.Length)

1510 ##   Sepal.Length Sepal.Width Petal.Length Petal.Width   Species
1511 ## 1       7.9        3.8       6.4       2  virginica

1512 Und mit Gruppen:
```

```

  iris %>% group_by(Species) %>%
    slice_max(Sepal.Length)

1513 ## # A tibble: 3 x 5

```

```

1514 ## # Groups: Species [3]
1515 ## Sepal.Length Sepal.Width Petal.Length Petal.Width Species
1516 ## <dbl>     <dbl>     <dbl>     <dbl> <fct>
1517 ## 1       5.8       4        1.2      0.2 setosa
1518 ## 2       7         3.2      4.7      1.4 versicolor
1519 ## 3       7.9      3.8      6.4      2    virginica

```

1520 `slice_min()` funktioniert genau gleich, nur dass die Beobachtung (=Zeile) mit dem kleinsten Wert einer
1521 Variable zurück gegeben wird.

1522 Die Funktion `slice_sample()` erlaubt es zufällige Beobachtungen zu ziehen. Dabei kann über das Argument
1523 `n` die Anzahl an Beobachtungen angegeben werden oder über das Argument `prop` der Anteil an Beobach-
1524 tungen.

```
slice_sample(iris, n = 5)
```

```

1525 ## Sepal.Length Sepal.Width Petal.Length Petal.Width Species
1526 ## 1       6.5       2.8      4.6      1.5 versicolor
1527 ## 2       6.3       3.3      4.7      1.6 versicolor
1528 ## 3       7.2       3.2      6.0      1.8 virginica
1529 ## 4       4.9       3.6      1.4      0.1 setosa
1530 ## 5       6.0       2.7      5.1      1.6 versicolor

```

1531 Das Ergebnis ist bei jedem von Ihnen anders, da es sich um eine zufällige Ziehung handelt. Wenn Sie diese
1532 Ergebnisse wiederholen möchte, können Sie über `set.seed()` die zufällige Ziehung reproduzierbar machen.

```
set.seed(123)
slice_sample(iris, n = 5)
```

```

1533 ## Sepal.Length Sepal.Width Petal.Length Petal.Width Species
1534 ## 1       4.3       3.0      1.1      0.1 setosa
1535 ## 2       5.0       3.3      1.4      0.2 setosa
1536 ## 3       7.7       3.8      6.7      2.2 virginica
1537 ## 4       4.4       3.2      1.3      0.2 setosa
1538 ## 5       5.9       3.0      5.1      1.8 virginica

```

1539 Wenn beispielsweise 5% der Beobachtungen gezogen werden sollen, kann dies so gemacht werden:

```
slice_sample(iris, prop = 0.05)
```

```

1540 ## Sepal.Length Sepal.Width Petal.Length Petal.Width Species
1541 ## 1       7.7       3.8      6.7      2.2 virginica
1542 ## 2       5.5       2.5      4.0      1.3 versicolor
1543 ## 3       5.5       2.6      4.4      1.2 versicolor

```

```

1544 ## 4      6.5      3.0      5.2      2.0  virginica
1545 ## 5      6.1      3.0      4.6      1.4  versicolor
1546 ## 6      6.3      3.4      5.6      2.4  virginica
1547 ## 7      5.1      2.5      3.0      1.1  versicolor

```

1548 `slice_sample()` berücksichtigt ebenfalls Gruppen. Mit den Argumenten `replace` und `weight_by` dann die
 1549 Zufallsziehung genauer spezifiziert werden. `replace` sagt, ob eine gezogenen Beobachtung wieder zurück ge-
 1550 legt wird oder nicht. Mit dem Argument `weight_by` können optional gewichte für jede Beobachtung vergeben
 1551 werden.

1552

1553 Aufgabe 30: Daten beschreiben

1555 Verwenden Sie den Datensatz `bhd_1.txt` und finden Sie für jedes Gebiet und Art die Beobachtung mit
 1556 kleinsten BHD.

1557 9.8 Spalten trennen

1558 Ein gut geplanter Datensatz besteht aus Beobachtungen (Zeilen), Variablen (Spalten) und in jeder Zelle ist
 1559 immer ein *genau* ein Wert gespeichert. Leider gibt es oft Datensätze, bei denen mehr als ein Wert pro Zelle
 1560 gespeichert wurde. Die Funktion `separate()` kann helfen solche Daten zu trennen.

1561 Wir verwenden einen erfunden Datensatz zu Beobachtungen von Tieren und einer geschätzten Distanz zu
 1562 diesen Tieren.

```

dat <- tibble(
  id = 1:4,
  beobachtung = c("10m, Reh", "100m, Reh", "20m, Fuchs", "40,Reh"),
)

```

1563 In der Spalte `beobachtung` sind zwei Informationen gespeichert: Die Distanz zur Beobachtung und die Art.
 1564 Das ist ungünstig, weil wir so weder nach Tierart noch nach distanz filtern können (nicht *tidy*). Mit der
 1565 Funktion `separate()`, können wir Beobachtungen einer Spalte in mehrere Spalten trennen. Dafür muss der
 1566 Spaltennamen (Argument `col`), die neuen Spaltennamen (Argument `into`) und das Trennzeichen (Argument
 1567 `sep`) angegeben werden.

```
separate(dat, col = beobachtung, into = c("Distanz", "Art"), sep = ",")
```

```

1568 ## # A tibble: 4 x 3
1569 ##       id Distanz Art
1570 ##     <int> <chr>   <chr>
1571 ## 1      1 10m     "Reh"

```

```
1572 ## 2      2 100m    " Reh"  
1573 ## 3      3 20m     " Fuchs"  
1574 ## 4      4 40      "Reh"
```

1575 Nach dem Aufruf von `seperate()` gibt es zwei neue Spalten (`Distanz` und `Art`), die die alte Spalte
1576 `beobachtung` ersetzen.

1577

1578 **Aufgabe 31: Aufräumen**

1580 Verwenden Sie den folgenden Datensatz und bringen Sie ihn in eine Form, die sicherstellt dass

- 1581 • jede Zelle genau einen Wert enthält.
1582 • jede Zeile eine Beobachtung ist.
1583 • die Spaltennamen aus einem ausschlagkräftigen Wort bestehen.

```
dat <- data.frame(  
  standort = c("a1", "a2", "b1", "b2"),  
  j2019 = c("3 x Fuchs", "4 x Reh", "1 x Fuchs", "2 x Reh"),  
  j2020 = c("2 x Fuchs", "1 x Reh", "", "2 x Fuchs"))  
)
```

1584 10 Arbeiten mit Text

1585 Bis jetzt haben wir fast ausschließlich mit Zahlen oder Abbildungen gearbeitet. R bietet aber auch viele
 1586 Werkzeuge, um mit Text zu arbeiten. Wir wollen hier ein paar Funktionen dafür vorstellen. Als erstes sollte
 1587 nochmals klargestellt werden, was eigentlich ein Text ist. In R ist alles, das innerhalb von doppelten ("") oder
 1588 einfachen ('') Anführungszeichen geschrieben ist, Text.

1589 Anbei einige Beispiele:

```
a <- "Das ist ein kurzer Satz."
b <- "Auch das ist 'moeglich'." 
z <- "30"
```

1590 Wichtig ist hier zu sehen, dass z nicht als Zahl sondern, als Text interpretiert wird (siehe Datentypen).

```
z + 1
```

1591 ## Error in z + 1: nicht-numerisches Argument für binären Operator

1592 Wenn man sicher ist, dass es sich bei einem Textobjekt um eine Zahl handelt, kann man dies mit der Funktion
 1593 `as.numeric()` in eine Zahl umwandeln.

```
as.numeric(z) + 1
```

1594 ## [1] 31

1595 Aber mit a führt dies wieder zu einem NA-Wert, da a nicht in eine Zahl umgewandelt werden kann.

```
as.numeric(a) + 1
```

1596 ## Warning: NAs durch Umwandlung erzeugt

1597 ## [1] NA

1598 10.1 Arbeiten mit Text

1599 Wir wollen erst einmal drei Funktionen besprechen, die es erlauben mit Text zu arbeiten. Die Funktion
 1600 `nchar()`¹¹ gibt an wie viele Zeichen ein Text hat. Also z.B.

```
nchar("Hallo")
```

1601 ## [1] 5

¹¹`char` ist kurz für *character*.

```
nchar("30")
```

```
1602 ## [1] 2
```

```
nchar("Hallo und Guten Tag!")
```

```
1603 ## [1] 20
```

1604 Die Funktion `paste()` erlaubt es verschiedene Variablen mit Text zu verbinden. Wenn wir z. B. die Variablen `vorname <- "Eva"` und `name <- "Meier"` haben und wir wollen eine neue Variable `full_name <- Eva Meier"` erzeugen, dann kann das mit der Funktion `paste()` gemacht werden.

```
vorname <- "Eva"
name <- "Meier"
full_name <- paste(vorname, name)
full_name
```

```
1607 ## [1] "Eva Meier"
```

1608 Die Funktion `paste()` hat das Argument `sep`, das auf ein Leerzeichen () gesetzt ist, aber auch anders sein kann und das Trennzeichen definiert.

```
full_name <- paste(vorname, name, sep = ", ")
full_name
```

```
1610 ## [1] "Eva, Meier"
```

1611 Die Funktion `substr()` erlaubt es am Anfang oder Ende eines Wortes etwas abzuschneiden. Dabei muss immer die Anfangs- (`start`) und Endposition (`stop`) angegeben werden.

```
substr("Hallo", start = 1, stop = 3)
```

```
1613 ## [1] "Hal"
```

```
substr("Hallo", start = 2, stop = 5)
```

```
1614 ## [1] "allo"
```

```
1615
```

Aufgabe 32: Arbeiten mit Text 1

1618 Verwenden Sie den folgenden Vektor:

```
txt <- c("Versicherung", "Vogel", "Station", "Gutschein", "Statistik", "Stift",
       "Methoden", "Fluss", "Rudel", "Baum", "Haus", "Wahrscheinlich", "Foto",
       "Seife", "Garten", "Auto", "Handy", "Teller", "Kalender")
```

- 1619 1. Aus wie vielen Buchstaben besteht jedes Wort?
 1620 2. Finden Sie das längste Wort.
 1621 3. Wie viel Prozent der Wörter fangen mit einem S an?
 1622 4. Fügen Sie jedem Wort seine Position im Vektor hinzu. Beispielsweise soll aus Vogel "2. Vogel" werden
 1623 usw.

1624 10.2 Finden von Textmustern

- 1625 Mit der Funktion `grep()` können Muster in einem Text gefunden werden. Wenn wir beispielsweise folgenden
 1626 Vektor mit Textelementen haben.

```
txt <- c("Buesgenweg", "Friedlaender Weg", "Berliner Strasse")
```

- 1627 Und wir wollen alle Straßennamen die ein weg haben abfragen, dann können wir folgenden Befehl ausführen:

```
grep("Weg", txt)
```

1628 ## [1] 2

- 1629 Im zweiten Element von `txt` kommen die Zeichen Weg vor. Beachte, in der Standardeinstellung wird zwischen
 1630 Groß- und Kleinschreibung unterschieden. Dies kann mit dem Argument `ignore.case = TRUE` angepasst
 1631 werden.

```
grep("Weg", txt, ignore.case = TRUE)
```

1632 ## [1] 1 2

- 1633 Mit der Funktion `sub` können Zeichen innerhalb einer Zeichenkette ersetzt werden.

- 1634 So ersetzt der folgende Ausdruck ae mit ä.

```
sub("ae", "ä", "Friedlaender Weg")
```

1635 ## [1] "Friedländer Weg"

- 1636 Wenn allerdings das zu ersetzende Zeichen mehr als einmal vorkommt und beide Instanzen ersetzt werden
 1637 sollen braucht man die Funktion `gsub`.

```
txt <- "Friedlaender Weg und Reinhaeuser Landstrasse sind zwei Strassen in Goettingen."
sub("ae", "ä", txt)
```

1638 ## [1] "Friedländer Weg und Reinhaeuser Landstrasse sind zwei Strassen in Goettingen."

1639 Mit `sub()` wird nur das erste `ae` ersetzt, während `gsub()` alle `ae` mit einem `ä` ersetzt.

```
gsub("ae", "ä", txt)
```

1640 ## [1] "Friedländer Weg und Reinhäuser Landstrasse sind zwei Strassen in Goettingen."

1641 Oft ist der genaue Ausdruck den man finden möchte jedoch Variabel. Beispielsweise möchte man alle Wörter
 1642 mit einem Umlaut oder Zahlen finden möchte, kann man das oft abkürzen. Dafür gibt es reguläre Ausdrücke.
 1643 Wir werden hier nur ein paar beispielhafte Anwendungen besprechen.

1644 Sowohl in den Funktionen `grep()` als auch `(g)sub()` kann mit anstatt dem Muster (immer das erste Argument)
 1645 aus ein regulärer Ausdruck angegeben werden. Mit `[1-9]` sind alle Zahlen von 1 bis 9 gemeint.

1646 Das Ziel ist es jetzt alle Straßen zu finden, die auch eine Hausnummer haben:

```
txt <- c("Büsgenweg 1", "Berliner Strasse", "Kurze Strasse 13")
```

```
grep("[0-9]", txt)
```

1647 ## [1] 1 3

1648 Damit lässt sich auch das Problem mit groß- und kleingeschriebenen Wörtern lösen.

```
txt <- c("Buesgenweg", "Friedlaender Weg", "Berliner Strasse")
grep("[wW]eg", txt)
```

1649 ## [1] 1 2

1650

1651 Aufgabe 33: Arbeiten mit Text 2

1653 Verwenden Sie den folgenden Vektor:

```
txt <- c("Versicherung", "Methoden", "Fluss", "Rudel",
       "Baum", "Haus", "Foto", "Auffahrt", "Auto", "Handy", "Teller",
       "Kalender", "Aufbau")
```

- 1654 1. In wie vielen Wörtern kommt der Doppellaut **au** vor?
1655 2. Ersetzen Sie in allen Wörtern alle **au** mit **_ _**.

```
grep("au", txt)
```

```
1656 ## [1] 5 6 13
```

```
gsub("au", "_ _", txt)
```

```
1657 ## [1] "Versicherung" "Methoden"      "Fluss"        "Rudel"       "B_ _m"  
1658 ## [6] "H_ _s"          "Foto"         "Auffahrt"     "Auto"        "Handy"  
1659 ## [11] "Teller"        "Kalender"     "Aufb_ _"
```

1660 11 Arbeiten mit Zeit

1661 Für den Computer bzw. R ist ein Datum erst einmal nichts anderes als ein Text. Für uns ist es sofort
 1662 klar, dass der "13.2.2021" der 13. Februar 2021 ist, für den Computer zunächst nicht. Wir müssen R also
 1663 irgendwie sagen, dass die 13 der Tag ist, die 2 der Monat und 2021 das Jahr. Dass der Computer die einzelnen
 1664 Komponenten erkennt, und in einen Datentyp eigens für Zeitformate überführt, nennt man *parsen*¹². Durch
 1665 das *parsen* wird die Variable in den Datentyp **Date** überführt. Das Arbeiten mit Datum und Zeit kann kann
 1666 anfangs sehr mühsam sein und viele Zeit-spezifischen Datenoperationen lassen sich auch mit den Basis-
 1667 Datentypen durchführen. Sobald man einige Grundfertigkeiten erworben hat, stellt man jedoch fest, dass die
 1668 Arbeit mit dem Zeitformat-Datentyp schneller und effizienter funktioniert. Starten Sie am besten gleich mit
 1669 "echten" Zeit-Datentypen und versuchen Sie nicht, sich irgendwie mit Text und numerischen Datentypen
 1670 selbst etwas zu basteln. Der erste Schritt ist immer ein Datum zu *parsen*. Wir verwenden dafür Funktionen
 1671 aus dem Paket **lubridate**. Als erstes müssen wir wieder Paket **lubridate** laden mit:

```
library(lubridate)
# lubridate ist Teil des Tidyverse und kann auch so geladen werden:
# library(tidyverse)
```

1672 **lubridate** bietet eine Vielzahl von Funktionen zum *parsen* von Datum und Zeit, die sich aus:

- 1673 • y für Jahr,
- 1674 • m für Monat,
- 1675 • d für Tag,
- 1676 • h für Stunde,
- 1677 • m für Minute und
- 1678 • s für Sekunde

1679 zusammen setzten. Alle Funktionen nehmen als erstes Argument ein Textstring. Wenn wir z.B. den String
 1680 2020-01-20 parsen wollen können wir das mit der Funktion **ymd** machen.

```
ymd("2020-01-20")
```

1681 ## [1] "2020-01-20"

1682 Dabei erkennt **lubridate** in der Regel die Trennzeichen:

```
ymd("2020.01.20")
```

1683 ## [1] "2020-01-20"

¹²to parse heißt zergliedern bzw. grammatisch bestimmen.

```
ymd("2020/01/20")
```

```
1684 ## [1] "2020-01-20"
```

```
ymd("2020 01 20")
```

```
1685 ## [1] "2020-01-20"
```

1686 Wenn die die Anordnung der einzelnen Komponenten anders ist, gibt es einfach eine andere Funktion.

```
dmy("20.1.2020")
```

```
1687 ## [1] "2020-01-20"
```

1688 Jetzt stellt sich die Frage, was der Vorteil ist, wenn R ein Datum parst.

```
d <- dmy("20.1.2020")
```

1689 Wir können jetzt mit `d` arbeiten und einzelne Komponenten extrahieren.

```
day(d)
```

```
1690 ## [1] 20
```

```
month(d)
```

```
1691 ## [1] 1
```

```
year(d)
```

```
1692 ## [1] 2020
```

1693 Oder auch Zeiteinheiten hinzufügen oder abziehen.

```
d + days(10)
```

```
1694 ## [1] "2020-01-30"
```

```
d - years(20)
```

```
1695 ## [1] "2000-01-20"
```

```
d + hours(25)
```

1696 ## [1] "2020-01-21 01:00:00 UTC"

1697

Aufgabe 34: Arbeiten mit Datum und Zeit

- 1700 • Parsen Sie folgende Zeitangaben 23.1.2020, 13.2.1992 20:55:23, Mar/3/97 und 10.7.2020 19:15
 1701 und speichern Sie diese in einen Vektor d.
 1702 • Extrahieren Sie nun aus jedem Element aus d das Jahr und die Stunde.
 1703 • Fügen zu jedem Element in d 10 Tage hinzu.

11.1 Arbeiten mit Zeitintervallen

1705 Mit zwei Zeitpunkten lassen sich Zeitintervalle (Periods) erstellen, dafür können wir die Funktion
 1706 `interval()` aus dem Paket `lubridate` verwenden¹³.

```
anfang <- ymd("2020-03-18")
ende <- anfang + years(1)

int <- interval(anfang, ende)
```

1707 Wir können jetzt mit `int` arbeiten und beispielsweise das Intervall verschieben,

```
int_shift(int, years(3))
```

1708 ## [1] 2023-03-18 UTC--2024-03-18 UTC

1709 die Länge des Intervalls berechnen

```
int_length(int) # in Sekunden
```

1710 ## [1] 31536000

1711 oder testen, ob ein Datum innerhalb des Intervalls liegt.

```
ymd("2020-07-1") %within% int
```

1712 ## [1] TRUE

¹³Alternativ zur Funktion `interval()` kann auch der `%--%`-Operator verwendet werden. Man könnte `int` auch so erstellen `int <- anfang %--% ende`.

```
1713  ymd("2021-07-1") %within% int
```

1713 ## [1] FALSE

1714 %within% funktioniert genauso mit Vekotren oder mit mehreren Intervallen. Wir könnten also zwei Intervalle
1715 definieren (z.B. Ostern und Pfingsten).

```
ostern <- ymd("2021-04-02") --% ymd("2021-04-05")
pfingsten <- ymd("2021-05-22") --% ymd("2021-05-24")
```

1716 Und Überprüfen welche Termine in eines der zwei Intervalle fallen.

```
termine <- ymd("2021-03-29") + weeks(0:10)

# Ostern
termine %within% ostern
```

1717 ## [1] FALSE TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE

```
# Pfingsten
termine %within% pfingsten
```

1718 ## [1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE TRUE FALSE FALSE

1719 Der Zeit-Datentyp wird auch oft benutzt, um den Zeitbedarf von längeren Berechnungen zu messen.

```
t1 <- now()
mean(runif(1e7)) #Beispielhaft für eine Rechenoperation
```

1720 ## [1] 0.4999484

```
t2 <- now()
int_length(interval(t1, t2))
```

1721 ## [1] 0.6398687

1722 11.2 Formatieren von Zeit

1723 Für die Ausgabe in Berichten oder Grafiken soll das Datum oft in einer speziellen Form dargestellt werden.

1724 Die Funktion `format()` bietet Möglichkeiten ein Datumsobjekt zurück in Text umzuwandeln.

1725 Ein Beispiel wäre `ymd("2021-2-10")` als 10.2.21 auszugeben.

```

d <- ymd("2021-2-21")
format(d, "%d.%m.%y")

1726 ## [1] "21.02.21"

```

1727 Dabei handelt sich bei `%d.%m.%y` um Abkürzungen für die unterschiedlichen Komponenten eines Datumobjekts. Siehe dazu die Hilfeseite von `strptime` (`help(strptime)`).

1729

1730 1731 Aufgabe 35: Arbeiten mit Intervallen

1732 Wie viele Einträge aus dem Vektor `v1` befinden sich in einem Intervall, das zwischen dem 1.3.2021 und dem
1733 5.3.2021 definiert ist.

```

v1 <- c(
  "2021-03-05", "2021-03-03", "2021-03-09", "2021-03-09", "2021-03-09",
  "2021-03-03", "2021-03-08", "2021-03-10", "2021-03-07", "2021-03-10"
)

```

1734 11.3 Zeitreihen

1735 In der Forstbranche haben wir oft mit Zeitreihen zu tun. Zeitreihen sind Variablen, für die in zeitlichen Inter-
1736 valen Daten vorliegen. In der Regel monatlich, vierteljährig oder jährlich, wobei die Intervalle zwischen den
1737 Messungen bei Zeitreihen immer gleich lang sind. Wiederholungsmessungen von Forsteinventuren (Forstein-
1738 richtungen, Betriebsinventuren, die meisten forstl. Experimente, die BWI, ...) sind also in der Regel keine
1739 Zeitreihen in engeren Sinne. Turnusmäßig vermessene Versuchsflächen, wie sie z. B. die Versuchsanstalten
1740 unterhalten oder jährlich gemeldete Holzpreise jedoch schon.

1741 Zeitreihen unterscheiden sich nicht nur technisch, sondern auch inhaltlich fundamental von den uns schon
1742 bekannten Daten. Statistisch gesehen haben Zeitreihen Besonderheiten gegenüber anderen Variablen, da
1743 Sie von Ihrer eigenen Vergangenheit abhängen (autokorrikt sind) und auch die Abhängigkeit anderer Va-
1744 riablen in der Regel nur sinnvoll ist, wenn deren Vergangenheit mitberücksichtigt wird (cross-correlation).
1745 Konventionelle Statistik ist oft nicht möglich, um Zeitreihen zu analysieren. Selbst ein ordinärer arithme-
1746 tischer Mittelwert ist schon nicht mehr geeignet, um Zeitreihen statistisch zu beschreiben. Angefangen mit
1747 der Datendarstellung gibt es in R deshalb spezifische Zeitreihen-Funktionen. Aus diesem Grund sollten Sie
1748 Zeitreihen als solche speichern. Viele Funktionen erkennen den Datentyp und führen entsprechend spezifische
1749 Zeitreihen-Operationen durch, wenn ihnen Daten vom Typ "Zeitreihen" übergeben werden. Laden wir z. B.
1750 die Holzpreise für Fichte 2b (das sog. Leitsortiment, Fichtenholz mit einem Mittendurchmesser von 20 bis 25
1751 cm), das Holzaufkommen dieses Sortiments (Einschlagsvolumen) und die Preise für Nadelholz vom
1752 statistischen Bundesamt¹⁴. Wir laden die Daten zunächst als csv ein:

¹⁴Sie können sich die Daten auch selbst über die Website laden oder das Paket `wiesbaden` verwenden, um die Daten direkt in den R Workspace herunterzuladen zu laden. Jedoch müssen Sie sich zuerst registrieren

```
zr <- read_csv2("data/zeitreihen_halbwaren-preis_holzpreis_holzaufkommen.csv")
```

1753 Diese 3 Zeitreihen bilden zusammen ein klassisches Marktmodell mit dem Preis eines homogenen Gutes
 1754 (Leitsortimentspreis), dem Angebot (Holzeinschlag) und der Nachfrage (Schnittholzpreis). Mit der Funktion
 1755 `ts` werden die Daten in ein Zeitreihenobjekt überführt (*pasrse*). Die Spalte mit den Jahren ist dann nicht mehr
 1756 nötig, da die Informationen zur Zeit keine eigene Variable mehr sind, sondern als sog. Metainformationen in
 1757 dem Objekt gespeichert wird. Die Spalten sollten nur noch Daten enthalten.

```
zr <- read_csv2("data/zeitreihen_halbwaren-preis_holzpreis_holzaufkommen.csv")
```

```
zr <- ts(data = zr %>% select(-Jahr), start = zr$Jahr[1])
```

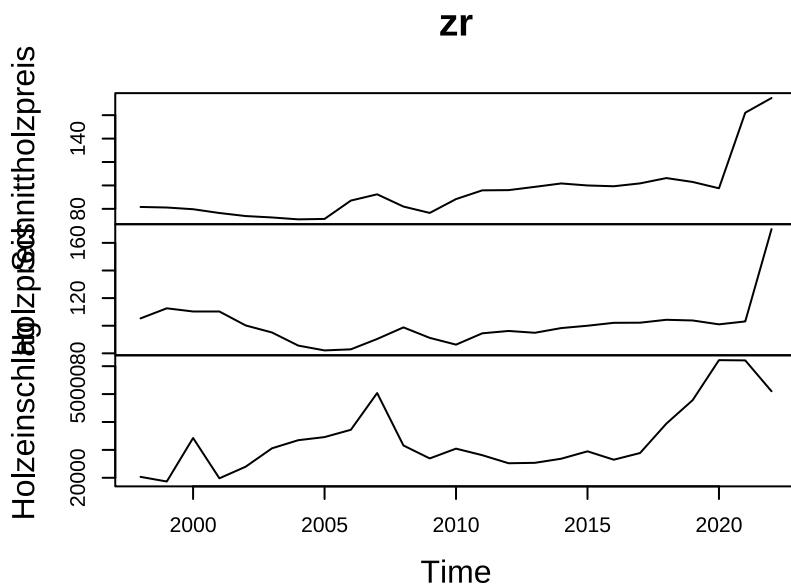
`typeof(zr)` # Zeitreihen sind sog. S3 Objekte. Das heißt, Zeitreihen gehören nicht zu den

1758 ## [1] "double"

Basis Datentypen (siehe Kapitel Variablen, Funktionen und Datentypen),
sondern sind eine Unterkategorie des Datentyps "Liste".

1759 Die wichtigsten Argumente sind - `data` Vektor oder Matrix, der nur die Daten enthält - `start` Startzeitpunkt -
 1760 `frequency` Anzahl der Beobachtungen pro Zeiteinheit, also z. B. 12 bei monatlichen oder 4 bei quartalsweisen
 1761 Erhebungen

```
plot(zr) # Die base Plot Funktion erkennt, dass es sich um Zeitreihen handelt.
```

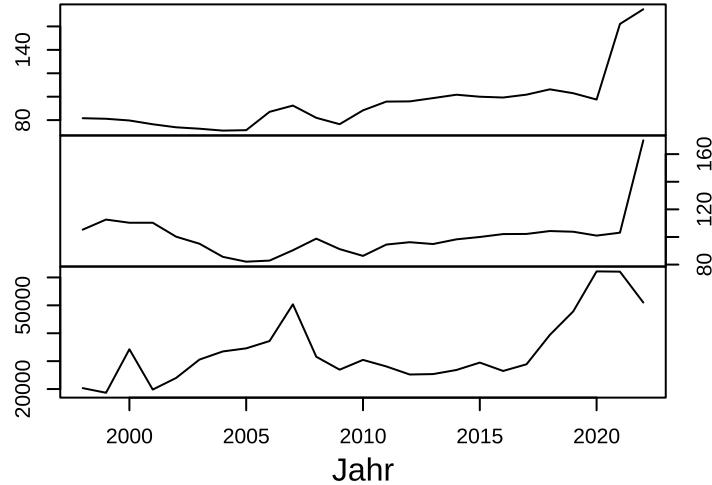


1762

- 1763 Wir können low-level Layer hinzufügen. Die meisten base plot Funktionen funktionieren auch für Zeitreihen

```
plot(zr, ann = FALSE, yax.flip = TRUE)
mtext(side = 3, cex = 1.5, font = 2, "Holzmarktentwicklung seit 1998", line = 2)
mtext(side = 1, cex = 1, line = 2.5, "Jahr")
```

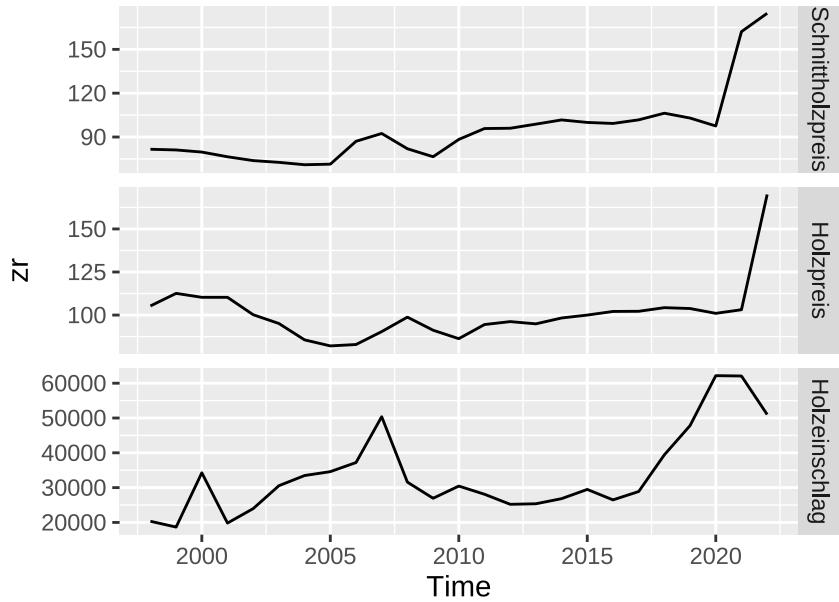
Holzmarktentwicklung seit 1998



- 1764

- 1765 Beide Plot-Philosophiebn haben eine Zeitreihen-Funktion. Das Paket `ggfortify` ermöglicht automatisierte
1766 Zeitreihenplots im `ggplot2` Stil. Damit ist auch das Problem der y-Achsenbeschriftungen gelöst.

```
autoplot(zr, facets = TRUE)
```

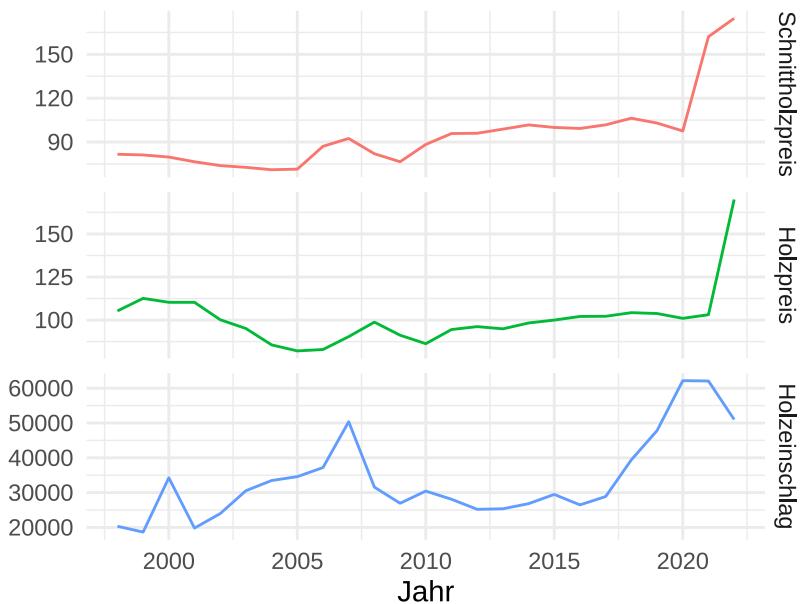


1767

- 1768 Wir können die Abbildung im ggplot2 Stil ändern. Hier nur ein Minimalbeispiel zur Veranschaulichung.
 1769 Siehe Kapitel 8.4 ggplot2: Eine Alternative für Abbildungen für mehr Möglichkeiten.

```
zr_autoplot <- autoplot(zr, facets = TRUE, colour = TRUE) +
  ylab("") # Keine y-Achsenbeschriftung
  xlab("Jahr") +
  guides(colour = "none") # Keine Legende

zr_autoplot + theme_minimal()
```



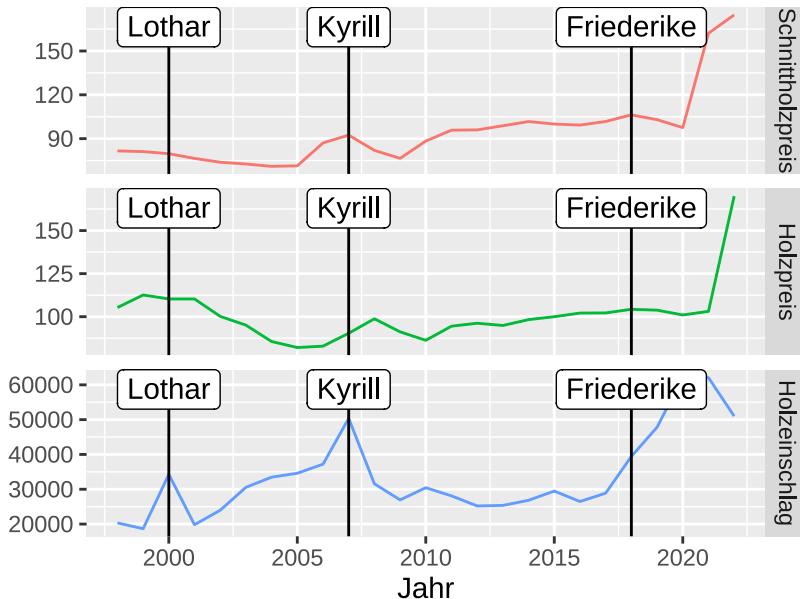
1770

```

z2 <- zr_autoplot + geom_vline(xintercept = c(2000, 2007, 2018))

z2 + annotate(x = 2000, y = +Inf, label = "Lothar", vjust = 1, geom = "label") +
  annotate(x = 2007, y = +Inf, label = "Kyrill", vjust = 1, geom = "label") +
  annotate(x = 2018, y = +Inf, label = "Friederike", vjust = 1, geom = "label")

```

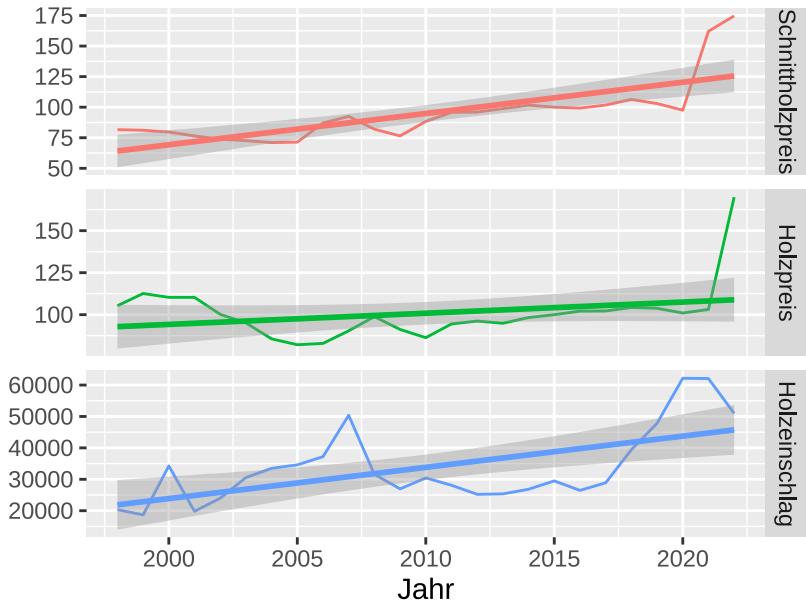


1771

¹⁷⁷² Eine Trendlinie macht hier offensichtlich keinen Sinn. Die Trendlinie ist eine lineare Regression, also eine ordinäre Statistik, die wie eingangs erwähnt für Zeitreihen ungeeignet ist. Daher verwenden wir den sog.

1774 Loess-Glättter, um einen Trend in die Daten zu legen. Der Loess-Glättter ist ein sehr flexible Kurve. Wir
1775 sehen hier beispielsweise, dass der Leitholzpreis träge oder gar nicht auf das Angebot reagiert. Die Nachfrage
1776 jedoch zumindest in der einen Periode, in der sie stark steigt, den Holzpreis jedoch mit zeitlichem Verzug
1777 stark ansteigen lässt. Dieser visuelle Eindruck lässt sich durch spezifische Zeitreihen-Regressionen schätzen.

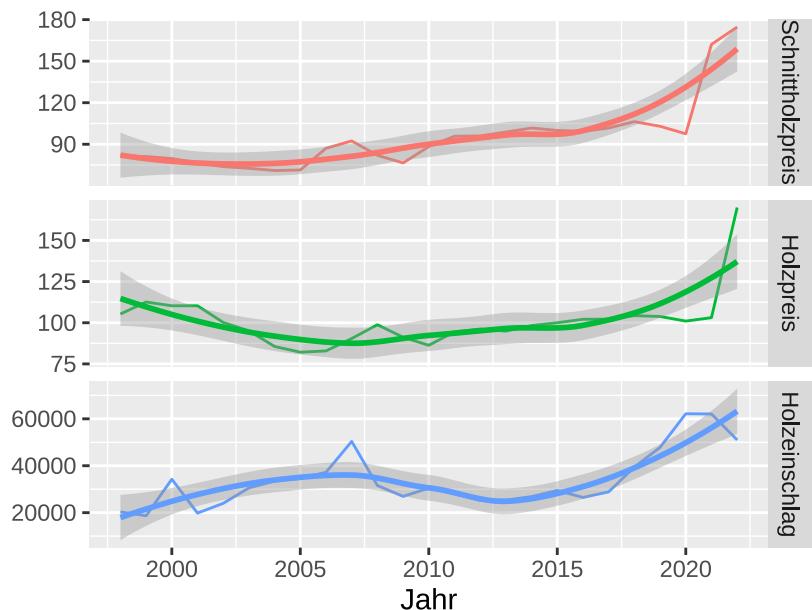
```
zr_autoplot + geom_smooth(method = "lm") +
  guides(colour = "none") # Nochmals nötig, da die geom_smooth() Funktion wieder eine
```



1778

```
# Legende erzeugt hat.

zr_autoplot + geom_smooth(method = "loess") +
  guides(colour = "none") # Nochmals nötig, da die geom_smooth() Funktion wieder eine
```



1779

Legende erzeugt hat.

1780 12 Aufgaben Wiederholen (for-Schleifen)

1781 Um einfache Programme zu schreiben, müssen Sie den Ablauf der Programmcodes kontrollieren können.
 1782 Kontrollieren bedeutet in diesem Zusammenhang, dass Codeabschnitte nur unter definierten Bedingungen
 1783 ablaufen. Sie programmieren also zwei Sachen. 1) den Code selbst und 2), die Bedingungen die erfüllt sein
 1784 müssen, damit der Code ausgeführt wird. Der Code muss do generisch geschrieben sein, dass er komplett
 1785 durchläuft, auch wenn unterschiedliche Dateneingaben gemacht werden. Diese Kontrollbedingungen ermög-
 1786 lichen es Ihnen generisch zu programmieren. Sie schreiben Ihren Code also nicht speziell maßgeschneidert
 1787 für ein Problem, sondern so generell, dass er für mehrere Auswertungen funktioniert. Um dies zu gewähr-
 1788 leisten, müssen Sie bestimmte Situationen vorhersehen und abfangen. Hierbei helfen Ihnen Kontrollstruktu-
 1789 ren (**Control Flow**). Grundsätzlich gibt es **Control Flow** Funktionen zur Wiederholung von Codeblöcken
 1790 (Schleifen) und logische Bedingungen (bedingte Anweisung).

1791 12.1 Schleifen

1792 Bis jetzt wurden alle Skripte einfach der Reihe nach abgearbeitet und zwischendurch bestimmte Programm-
 1793 teile, je nach Situation, selbstständig ausgeführt oder übersprungen. Mit einer Schleife kann man erreichen,
 1794 dass eine Gruppe von Befehlen (der sog. Schleifenrumpf) mehrfach abgearbeitet wird, zum Beispiel wenn
 1795 bestimmte Auswertungsschritte auf mehrere Datensätze oder Variablen angewendet oder Funktionen mit un-
 1796 terschiedlichen Parametern oder Startwerten aufgerufen werden sollen. Weitere Anwendungsmöglichkeiten
 1797 sind iterative Algorithmen, in denen die Eingabewerte des aktuellen Iterationsschrittes von einem vorherigen
 1798 abhängig sind. Besonders in Simulationen kommen Schleifen häufig zum Einsatz, da große Anzahlen von
 1799 Wiederholungen benötigt werden.

1800 Man unterscheidet zwischen zwei Arten von Schleifen: Bei den **for()**-Schleifen steht die Anzahl der Wieder-
 1801 holungen schon beim Eintritt in die Schleife fest, während die **while()**-Schleifen so lange ausgeführt werden,
 1802 bis eine Bedingung nicht mehr wahr ist. Mit der Funktion **break** wird eine Schleife abgebrochen und die
 1803 Programmausführung wird nach der Schleife fortgesetzt.

1804 Die wesentlichen Befehle sind

- 1805 • **for (i in X) {Code}**

1806 Wiederhole den Code im “Schleifenrumpf” für jedes Element aus X.

- 1807 • **while(Bedingung) {Code}**

1808 Wiederhole den Code im “Schleifenrumpf” so lange die logische Bedingung erfüllt ist.

- 1809 • **break()**

1810 Brich die Schleife ab. **break()** muss im Schleifenrumpf an der richtigen Stelle ausgeführt werden. Gute
 1811 Praxis ist jedoch, die for oder while Bedingungen, dass kein **break()**nötig ist, da **break()** anfällig für
 1812 Programmierfehler ist.

1813 **12.1.1 Wiederholen von Befehlen mit `for()`.**

1814 Steht vor Beginn der Schleife fest wie viele Schleifendurchgänge benötigt werden, wenn zum Beispiel in
 1815 einer Simulationen 99 Realisierungen erzeugt oder alle Elemente eines Vektors verarbeitet werden sollen,
 1816 verwendet man eine `for`-Schleife. Die allgemeine Form der `for`-Schleife ist:

```
X <- c(1 : 3) # Einträge die im Schleifenrumpf abgearbeitet werden.
for(i in X){
  # Schleifenrumpf
  print(i)
}
```

1817 ## [1] 1
 1818 ## [1] 2
 1819 ## [1] 3

1820 Das `i` steht in diesem Beispiel für die Schleifen-Variable. Sie muss nicht `i` heißen, sondern kann jeden
 1821 zulässigen Namen annehmen. Das `X` steht für einen existierenden Vektor oder eine existierende Liste bzw.
 1822 einen Ausdruck, der ein solches Objekt liefert (der Objektname ist ebenfalls frei wählbar). `for` und `in` sind
 1823 Schlüsselworte, sie müssen, ebenso wie die runden Klammern, vorhanden sein.

1824 Im ersten Durchgang erhält die Schleifen-Variable `i` den ersten Wert von `X` und der Schleifenrumpf wird
 1825 mit diesem Wert ausgeführt. Die Variable `i` nimmt nacheinander so lange die Werte von `X` an, bis ihr alle
 1826 Elemente zugewiesen wurden.

1827 Das folgende Beispiel wird zwar besser durch die entsprechende Vektoroperation gelöst, zeigt aber sehr
 1828 deutlich die Arbeitsweise der `for`-Schleife.

```
zahlen <- c(2, 3, 5)

for(element in zahlen){
  print(element^2)
}
```

1829 ## [1] 4
 1830 ## [1] 9
 1831 ## [1] 25

1832

1833 **Aufgabe 36: Schleifen 1**

1835 Verwenden Sie den Vektor `k <- c(1, 3, 9, 12, 15)` und schreiben Sie folgende `for`-Schleifen:

- 1836 1. Eine Schleife, die jedes Element aus `k` ausgibt.

- 1837 2. Eine Schleife, die zu jedem Element aus `k` 10 addiert und den neuen Wert ausgibt.
- 1838 3. Eine Schleife wie in 2), aber der neue Wert ($k + 10$) soll jetzt nicht mehr ausgegeben werden, sondern
1839 in `k10` gespeichert werden. Stellen Sie sicher, dass `k10` wieder von der Länge 5 ist.

1840

- 1841 Die Funktion `for()` ermöglicht es, einen Befehl beliebig oft zu wiederholen. Z.B. der folgende Ausdruck zieht
1842 10-Mal eine Stichprobe der Größe 1 aus dem Vektor `v`. Beachten Sie, dass die Schleifen-Variable `i` selbst gar
1843 nicht im Schleifenrumpf vorkommt. Das Ziel dieser Schleife ist nicht die Elemente des Vektors abzuarbeiten,
1844 sondern einfach nur den Ausdruck im Schleifenrumpf 10-Mal zu wiederholen.

```
v <- c(1, 4, 2, 3)
for (i in c(1 : 10)) {
  print(sample(v, 1))
}
```

1845 ## [1] 3
1846 ## [1] 1
1847 ## [1] 3
1848 ## [1] 3
1849 ## [1] 2
1850 ## [1] 3
1851 ## [1] 2
1852 ## [1] 2
1853 ## [1] 1
1854 ## [1] 4

1855 Auf gleiche Weise kann man auch über die Variablen eines Dataframes iterieren¹⁵. Das folgende Beispiel
1856 hat zum Ziel die Funktionsweise von Schleifen zu verdeutlichen. Schleifen haben in R jedoch den Nachteil,
1857 dass sie sehr langsam operieren. Wenn es geht, sollte man Alternativen verwenden. Die Funktionsweise
1858 wiederholender Auswertungen wird jedoch mit `for`-Schleifen deutlicher. Aus diesem Grunde werden wir uns
1859 in diesem Kurs auf Schleifen beschränken.

```
myLoopDf <- data.frame(a = c(2, 4, 7, 5),
                        b = c("Buche", "Eiche", "Eiche", "Buche"),
                        d = c(50, 60, 55, 80))

for (i in c(1 : 4)) {
  summeAd <- myLoopDf[i, "a"] + myLoopDf[i, "d"]
  print(myLoopDf$b[i])
  print(summeAd)
}
```

¹⁵Zur Info: Dieses Beispiel lässt sich schneller mittels der vektorwertigen Operation `apply()` lösen.

```

1860 ## [1] "Buche"
1861 ## [1] 52
1862 ## [1] "Eiche"
1863 ## [1] 64
1864 ## [1] "Eiche"
1865 ## [1] 62
1866 ## [1] "Buche"
1867 ## [1] 85

```

1868

Aufgabe 37: for-Schleife

1871 Lesen Sie den Datensatz `bhd_1.txt` ein und verwenden Sie eine `for`-Schleife.

- 1872 • Ziehen Sie 500-Mal je 35 Beobachtungen für den BHD.
- 1873 • Berechnen Sie jeweils den Mittelwert aus den 35 Werten.
- 1874 • Speichern Sie die 500 Mittelwerte in einem neuen Vektor `mittel`.
- 1875 • Wie ist die Verteilung dieser 500 Mittelwerte zu interpretieren?

12.1.2 Wiederholen von Befehlen mit `while()`

1877 Die `while`-Schleifen finden Anwendung, wenn die Anzahl der zu durchlaufenden Wiederholungen vorher nicht bekannt ist, wie zum Beispiel bei Iterationsverfahren, die bis zu einer gewissen Genauigkeit durchlaufen 1878 werden sollen. Die `while`-Schleife besteht in R aus dem Schlüsselwort `while()` und einer Bedingung in runden 1879 Klammern.

```

while (Bedingung) {
  # Schleifenrumpf
}

```

1881 Sie ist in der praktischen Programmierung nicht so relevant wie die `for`-Schleife. Sie sei deshalb hier nur 1882 kurz erwähnt. Die Abbruchbedingung wird jedes Mal geprüft bevor der Schleifenrumpf durchlaufen wird. 1883 Die Bedingung wird ausgewertet und wenn diese `TRUE` ist, wird der Schleifenrumpf ausgeführt und danach 1884 erneut die Bedingung überprüft. Ist die Bedingung nicht erfüllt, wird der Schleifenrumpf nicht durchgeführt 1885 und die Schleife beendet. Ist die Bedingung bereits vor Eintritt in die Schleife nicht erfüllt, wird die Schleife 1886 gar nicht erst durchlaufen.

1887 Da `while`-Schleifen also so lange ausgeführt werden, bis die Bedingung nicht mehr erfüllt ist, kann eine 1888 Endlosschleife entstehen. Dies kann passieren, wenn man nicht sauber programmiert hat. Wenn innerhalb 1889 der Schleife nicht dafür gesorgt wird, dass die Bedingung irgendwann nicht mehr erfüllt wird, so läuft die 1890 Schleife immer weiter. Steckt R in einer Schleife fest und reagiert nicht mehr, kann der Befehl unter Linux 1891 mit `Strg + C` und unter Windows mit `Esc` abgebrochen werden. Alternativ können Sie auf das rote STOP 1892 Symbol über der Konsole klicken.

1893 **12.2 Bedingte Ausführung von Codeblöcken**

1894 Innerhalb eines Skripts ist es mitunter notwendig je nach aktueller Situation unterschiedlich fortzufahren.
 1895 Die Situation wird mit einem logischen Ausdruck, einer sogenannten Bedingung, geprüft. Je nachdem, ob
 1896 die Bedingung wahr (TRUE) oder falsch (FALSE) ist, werden unterschiedliche Programmteile ausgeführt, der
 1897 jeweils andere Teil bleibt unberücksichtigt. Danach wird in jedem Fall die Programmausführung, mit den
 1898 auf die bedingte Anweisungen folgenden Anweisung, fortgesetzt. In R kann dies mit dem if-else-Konstrukt
 1899 realisiert werden, welches aus den Schlüsselwörtern `if()` und `else` sowie der in runde Klammern gefassten
 1900 Bedingung besteht.

```
if(Bedingung){
  # Anweisungen für Bedingung == TRUE
} else{
  # Anweisungen für Bedingung == FALSE
}
```

1901 Im folgenden Beispiel sollen die bisher abstrakt beschriebenen Vorgänge praktische Anwendung finden. In
 1902 dem Beispiel wird zunächst durch zufälliges Ziehen einer Zahl aus dem Bereich eins bis sechs ein Würfelwurf
 1903 simuliert. Anschließend wird der Würfelwurf mit einem if-Ausdruck kommentiert. Ist die Bedingung, es wurde
 1904 eine Sechs gewürfelt, erfüllt, wird der Code innerhalb der geschweiften Klammern ausgeführt, ansonsten wird
 1905 der Klammerinhalt ignoriert.

```
# Würfelwurf simulieren
x <- sample(1 : 6, 1)
# if-Konstrukt zur passgenauen Gratulation
if (x == 6) {
  print("Glückwunsch, eine Sechs!")
}
```

1906 In den meisten Fällen ist es für R irrelevant, ob sich zwischen den verschiedenen Elementen Leerzeichen oder
 1907 Zeilenumbrüche befinden. Bei dem else-Ausdruck dagegen wird ein Fehler erzeugt, wenn vor dem `else` nicht
 1908 die geschweifte Endklammer der vorausgehenden if- Bedingung steht.

```
# Würfelwurf simulieren
x <- sample(1 : 6, 1)
# if-else-Konstrukt: Gratulation oder Ermutigung
if(x == 6) {
  print("Glückwunsch, eine Sechs!")
} else {
  print("Beim nächsten Wurf klappt's bestimmt.")
}
```

1909 `## [1] "Beim nächsten Wurf klappt's bestimmt."`

1910

1911 **Aufgabe 38: Bedingte Programmierung**

- 1913
- Wenn keine 6 gewürfelt wurde, lassen Sie zusätzlich ausgeben welche Zahl gewürfelt wurde.

1914

 - Wiederholen Sie den Würfelwurf 10 Mal.

1915 13 (R)markdown

1916 13.1 Markdown Grundlagen

1917 Die Idee von Markdown ist, einfach Text strukturieren zu können und das ganze ohne umfangreiche Pro-
 1918 grammre zu erstellen. Im nächsten Abschnitt sehen wir dann, wie man Markdown und R-Code verbinden
 1919 kann. Hier soll es jedoch erst einmal darum gehen, die einfachsten Bausteine von Markdown vorzustellen.

1920 Am Anfang jedes Dokuments kommt eine Präambel. Diese fängt mit --- an und hört auch wieder mit ---
 1921 auf. Innerhalb der Präambel können dann Metainformationen über das Dokument festgelegt werden, dies
 1922 beinhaltet im einfachsten Fall: Titel, Autor und Datum. Das würde dann so aussehen:

```
1923 ---
1924 title: "Ein Titel"
1925 author: "Der, der es geschrieben hat"
1926 date: "März 2021"
1927 ---
```

1928 Danach folgt strukturierter und formatierter Text. Verschiedene Hierarchieebenen von Überschriften können
 1929 mit der Anzahl an # festgelegt werden. So ist eine Überschrift erster Ordnung # Kapitel eine Überschrift
 1930 zweiter Ordnung ## Unterkapitel usw.

1931 Listen können erstellt werden, wenn man am Anfang jeder Zeile ein – oder 1. schreibt.

```
1932 - Erster Eintrag
1933 - Zweiter Eintrag
1934 - Dritter Eintrag
```

1935 wird zu

```
1936   • Erster Eintrag
1937   • Zweiter Eintrag
1938   • Dritter Eintrag
```

1939 Eine zentrale Idee von Markdown ist es Text einfach zu formatieren. Werden eine oder mehrere Wörter mit
 1940 zwei Sternchen (**) eingefasst wird dieser Text **fett** dargestellt. Also aus **wichtig** wird **wichtig**. Das
 1941 gleiche funktioniert auch mit *kursiven* Text, jedoch muss man hier noch ein Sternchen verwenden, also aus
 1942 *kursiv* wird *kursiv*. Soll ein text fett und kursiv sein, kann man drei Sternchen verwenden. Aus ***sehr
 1943 wichtig*** wird dann **sehr wichtig**.

1944 Weitere Elemente wie Links oder Abbildungen können einfach eingebunden werden. Links werden mit [Link
 1945 text](url) in den Text integriert. Beispielsweise ist eine gute Idee bei [stackoverflow](#) bei Problemen nach
 1946 einer Lösung zu suchen. Dieser Link wurde mit [stackoverflow](www.stackoverflow.com) erstellt.

1947 Für Abbildungen gibt es einen ganz ähnlichen Syntax. Mit ![Das R Logo](abb/r_logo.png) wird die
 1948 Abbildung r_logo.png eingebunden mit der Beschriftung: "Das R Logo".



Abbildung 9: Das R Logo

1949

Aufgabe 39: Arbeiten mit markdown

1952 Verwenden Sie das folgende Markdowndokument:

```

1953 ---
1954 title: "Dokument"
1955 author: "Ihr Name"
1956 date: "März 2021"
1957 ---
1958
1959 # Einleitung
1960
1961 # Methoden

```

- 1962 1. Kopieren Sie die Vorlage in ein Dokument, das `test.md` heißt.
- 1963 2. Fügen Sie zwei Überschriften zweiter und dritter Ordnung hinzu.
- 1964 3. Fügen Sie einen *kursiven* Text hinzu.
- 1965 4. Fügen Sie einen Link zu einer Website hinzu.
- 1966 5. Kompilieren Sie die Datei, indem Sie in Rstudio auf Preview drücken (Abbildung 10).

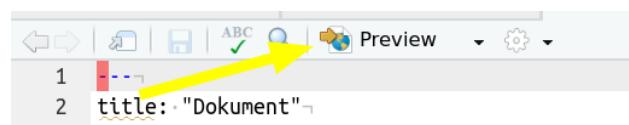


Abbildung 10: Kompilieren einer md-Datei.

1967 13.2 R und Markdown

1968 Markdown macht es bereits einfach Textdokumente und Dokumentationen zu verfassen, aber die wirkliche
 1969 Stärke liegt in der Möglichkeit R und Markdown zu kombinieren. Man spricht dann von Rmarkdown. Ein
 1970 weiteres Strukturelement, das wir noch nicht kennen gelernt haben, sind Code-Blöcke.

```

1971   ` ` `
1972   a <- 1:10
1973   a[1]
1974   ` ` `
1975   erzeugt

1976   a <- 1:10
1977   a[1]

1978 Momentan wird noch kein Code ausgeführt, sondern lediglich der Code, als Code dargestellt. Rmarkdown
1979 bietet nun die Möglichkeit Code beim kompilieren16 auszuführen. Dafür müssen wir nur einen Code-Block
1980 als R-Code-Block kennzeichnen.

```

```

1981   ` ` ` {R}
1982   a <- 1:10
1983   a[1]
1984   ` ` `

1985   erzeugt

a <- 1:10
a[1]

1986   ## [1] 1

```

1987 Beachte, die Variable `a` wird beim kompilieren erzeugt und steht dann R zur Verfügung. R-Code-Blöcke
1988 werden auch als Code Chunks bezeichnet. Diese Chunks können sehr genau angesprochen und angepasst
1989 werden. Einige wichtige Argumente sind:

- 1990 • `echo`: Gibt an, ob der Quelltext angezeigt werden soll oder nicht.
- 1991 • `result`: Gibt an, ob die Ergebnisse gezeigt werden sollen oder nicht.
- 1992 • `eval`: Diese Option gibt an ob der Chunk ausgeführt werden soll oder nicht.

1993

1994 Aufgabe 40: Arbeiten mit Rmarkdown

1996 Erstellen Sie eine neue Rmarkdown Datei mit dem Namen `test1.Rmd`. Erstellen Sie zwei Code-Chunks.
1997 Der erste soll nicht angezeigt werden und darin werden die Daten geladen (`bhd_1.txt`). Im zweiten Chunk
1998 plotten Sie das Alter der Bäume gegen den BHD. Was passiert mit dem Plot, wenn Sie die Datei kompilieren
1999 (drücken Sie dazu auf den Knit-Knopf; Abbildung 11).

¹⁶Unter kompilieren wird hier das Übersetzen eines Markdown-Dokuments in ein Ausgabeformat (z.B. pdf oder html) verstanden.

Abbildung 11: Kompilieren einer `Rmd`-Datei.

2000 14 Räumliche Daten in R

2001 14.1 Was sind räumliche Daten

2002 Räumliche Daten sind Beobachtungen, wie wir sie schon oft gesehen haben, mit einem räumlichen Bezug. Der
 2003 Unterschied zu nicht räumlichen Daten liegt darin, dass räumliche Daten eindeutig im *Raum* verortet werden
 2004 können. Häufig werden sogenannte Geoinformationssysteme (GIS) zum Arbeiten mit räumlichen verwendet.
 2005 R kann in vielerlei Hinsicht wie ein GIS eingesetzt werden und hier werden einige Grundfunktionalitäten
 2006 dafür besprochen. Räumliche Daten werden in zwei unterschiedliche Datentypen unterteilt: Vektor- und Ras-
 2007 terdaten. Vektordaten modellieren einzelne Objekte (= *Features*). Rasterdaten modellieren eine Oberfläche.
 2008 Vektordaten bestehen aus zwei Komponenten: 1) einer Geometrie, die die Form und Lage der Daten definiert
 2009 und 2) Attributen, den tatsächlichen Daten. Räumliche Daten werden oft als *Features* bezeichnet. Ein Feature
 2010 ist die räumliche Einheit einer Beobachtung. Je nach Art der räumlichen Daten können Features entweder
 2011 Punkte (z.B. ein Baum), Linien (z.B. eine Straße) oder Polygone (z.B. ein See) sein. Auch können mehrere
 2012 Geometrien zu einem Feature zusammengefasst werden. Ein Beispiel wäre eine Beobachtung für ein Land,
 2013 das aber aus mehreren Polygonen bestehen kann (z.B. Festland und Inseln). Features können dann weitere
 2014 Attribute (= Attributdaten) haben, z.B. eine ID, Name oder was auch immer man gemessen hat.
 2015 Rasterdaten bestehen aus einer Oberfläche von gleichgroßen Kacheln (= *Pixel*), die ein Gebiet abdecken.
 2016 Meist sind Pixel viereckig, aber das ist keine Voraussetzung. Dabei hat jedes Pixel einen Wert (das kann
 2017 auch ein fehlender Wert sein). Typische Beispiele für Rasterdaten sind Landnutzung oder Seehöhen.
 2018 In R kann sowohl mit Vektor- als auch mit Rasterdaten gearbeitet werden. Für Vektordaten bietet sich das
 2019 Paket `sf` an und für Rasterdaten das Paket `raster`.

2020 14.2 Koordinatenbezugssystem

2021 Eine der Herausforderungen für räumliche Daten ist die eindeutige Verortung im Raum. Dazu braucht man
 2022 ein *Koordinatenbezugssystem* (KBS). Einem KBS liegt ein mathematisches Modell der Erde zugrunde. Die
 2023 Details zu KBS werden schnell relativ kompliziert und wir beschränken uns hier lediglich darauf, wie KBS
 2024 verwendet werden können. Dazu müssen wir zwei Fälle unterscheiden: 1) einem Datensatz ein KBS zuweisen
 2025 und 2) Transformation des KBS eines Datensatzes in ein anderes KBS. Die technischen Details werden in
 2026 den folgend Abschnitten besprochen. Für beide Aufgaben müssen wir auf ein KBS verweisen können, ein
 2027 einfacher Ansatz dafür sind die sogenannten *EPSG-Codes*¹⁷.

¹⁷EPSG steht für European Petrol Survey Group

2028 14.3 Vektordaten in R

- 2029 Das Paket `sf` stellt Klassen zum Abbilden von Features zur verfügen, die dann in einem `data.frame` als
 2030 Liste gespeichert werden können. In der Regel erstellen wir Features nicht individuell, sondern lesen diese aus
 2031 externen Datenquellen (z.B. ESRI Shapefile) ein. Zum besseren Verständnis, erstellen wir es einmal manuell.
 2032 Wir haben die Koordinaten für drei Städte (Göttingen, Hannover und Berlin) als geografische Koordinaten
 2033 vorliegen (EPSG = 4326).

```
library(sf)
goe <- st_point(x = c(9.9158, 51.5413))
han <- st_point(x = c(9.7320, 52.3759))
ber <- st_point(x = c(13.405, 52.5200))
```

- 2034 Daraus können wir jetzt eine Geometriespalte für einen `data.frame` erstellen

```
geom <- st_sfc(list(goe, han, ber), crs = 4326)
```

- 2035 Somit haben wir die Geometrie in der Variable `geom` gespeichert, aber noch keine dazugehörigen Attribut-
 2036 daten. Diese können wir jetzt in einem weiteren `data.frame` abspeichern.

```
attr <- data.frame(
  name = c("Goettingen", "Hannover", "Berlin"),
  bundesland = c("Niedersachsen", "Niedersachsen", "Berlin"),
  einwohner = c(119000, 532000, 3650000)
)
```

- 2037 In einem letzten Schritt möchten wir jetzt die Geometrie (`geom`) und die Attributdaten (`attr`) zusammen-
 2038 führen.

```
staedte <- st_sf(attr, geom = geom)
staedte
```

- 2039 ## Simple feature collection with 3 features and 3 fields
2040 ## Geometry type: POINT
2041 ## Dimension: XY
2042 ## Bounding box: xmin: 9.732 ymin: 51.5413 xmax: 13.405 ymax: 52.52
2043 ## Geodetic CRS: WGS 84
2044 ## name bundesland einwohner geom
2045 ## 1 Goettingen Niedersachsen 119000 POINT (9.9158 51.5413)
2046 ## 2 Hannover Niedersachsen 532000 POINT (9.732 52.3759)
2047 ## 3 Berlin Berlin 3650000 POINT (13.405 52.52)

- 2048 Wir können nun mit `staedte` genau so arbeiten wie mit jedem anderen `data.frame` und die Geometrien
 2049 werden immer ‘berücksichtigt’. Zusätzlich kann man eine Reihe von geometrischen Operationen durchführen.

2050 Wenn ein `data.frame` Punkten hat, kann man dies relative einfach mit der Funktion `st_as_sf()` "räumlich"
 2051 machen. Für das vorherige Beispiel würden wir zuerst einen `data.frame` mit allen Informationen (zur
 2052 Geometrie und zu den Attributen erstellen).

```
dat <- data.frame(
  name = c("Goettingen", "Hannover", "Berlin"),
  bundesland = c("Niedersachsen", "Niedersachsen", "Berlin"),
  einwohner = c(119000, 532000, 3650000),
  x = c(9.9158, 9.7320, 13.405),
  y = c(51.5413, 52.3759, 52.5200)
)
```

2053 Dann kann man mit der Funktion `st_as_sf()` weiter arbeiten:

```
staedte1 <- st_as_sf(dat, coords = c("x", "y"), crs = 4326)
```

2054 14.4 Arbeiten mit Vektordaten

2055 Es gibt sehr viele Funktionen, um mit räumlichen Daten zu arbeiten, von denen wir hier einige vorstellen.

```
# Zeigt das KBS an
st_crs(staedte)
```

```
2056 ## Coordinate Reference System:
2057 ##   User input: EPSG:4326
2058 ##   wkt:
2059 ## GEOCRS["WGS 84",
2060 ##   ENSEMBLE["World Geodetic System 1984 ensemble",
2061 ##     MEMBER["World Geodetic System 1984 (Transit)"],
2062 ##     MEMBER["World Geodetic System 1984 (G730)"],
2063 ##     MEMBER["World Geodetic System 1984 (G873)"],
2064 ##     MEMBER["World Geodetic System 1984 (G1150)"],
2065 ##     MEMBER["World Geodetic System 1984 (G1674)"],
2066 ##     MEMBER["World Geodetic System 1984 (G1762)"],
2067 ##     MEMBER["World Geodetic System 1984 (G2139)"],
2068 ##     ELLIPSOID["WGS 84",6378137,298.257223563,
2069 ##       LENGTHUNIT["metre",1]],
2070 ##     ENSEMBLEACCURACY[2.0]],
2071 ##   PRIMEM["Greenwich",0,
2072 ##     ANGLEUNIT["degree",0.0174532925199433]],
2073 ##   CS[ellipsoidal,2],
2074 ##     AXIS["geodetic latitude (Lat)",north,
2075 ##       ORDER[1],
```

```

2076 ##           ANGLEUNIT["degree",0.0174532925199433]],  

2077 ##           AXIS["geodetic longitude (Lon)",east,  

2078 ##           ORDER[2],  

2079 ##           ANGLEUNIT["degree",0.0174532925199433]],  

2080 ##           USAGE[  

2081 ##           SCOPE["Horizontal component of 3D system."],  

2082 ##           AREA["World."],  

2083 ##           BBOX[-90,-180,90,180]],  

2084 ##           ID["EPSG",4326]]
```

2085 Wenn wir jetzt zu einem anderen KBS (z.B. EPSG:3035, ein europäisches projiziertes KBS) umrechnen
 2086 möchten, können wir das mit

```
s2 <- st_transform(staedte, 3035)  
st_crs(s2)
```

```

2087 ## Coordinate Reference System:  

2088 ##   User input: EPSG:3035  

2089 ##   wkt:  

2090 ##   PROJCRS["ETRS89-extended / LAEA Europe",  

2091 ##     BASEGEOGCRS["ETRS89",  

2092 ##       ENSEMBLE["European Terrestrial Reference System 1989 ensemble",  

2093 ##         MEMBER["European Terrestrial Reference Frame 1989"],  

2094 ##         MEMBER["European Terrestrial Reference Frame 1990"],  

2095 ##         MEMBER["European Terrestrial Reference Frame 1991"],  

2096 ##         MEMBER["European Terrestrial Reference Frame 1992"],  

2097 ##         MEMBER["European Terrestrial Reference Frame 1993"],  

2098 ##         MEMBER["European Terrestrial Reference Frame 1994"],  

2099 ##         MEMBER["European Terrestrial Reference Frame 1996"],  

2100 ##         MEMBER["European Terrestrial Reference Frame 1997"],  

2101 ##         MEMBER["European Terrestrial Reference Frame 2000"],  

2102 ##         MEMBER["European Terrestrial Reference Frame 2005"],  

2103 ##         MEMBER["European Terrestrial Reference Frame 2014"],  

2104 ##         ELLIPSOID["GRS 1980",6378137,298.257222101,  

2105 ##           LENGTHUNIT["metre",1]],  

2106 ##         ENSEMBLEACCURACY[0.1]],  

2107 ##     PRIMEM["Greenwich",0,  

2108 ##       ANGLEUNIT["degree",0.0174532925199433]],  

2109 ##     ID["EPSG",4258]],  

2110 ##     CONVERSION["Europe Equal Area 2001",  

2111 ##       METHOD["Lambert Azimuthal Equal Area",  

2112 ##         ID["EPSG",9820]],  

2113 ##       PARAMETER["Latitude of natural origin",52,  

2114 ##         ANGLEUNIT["degree",0.0174532925199433],
```

```

2115 ##           ID["EPSG",8801]],
2116 ##           PARAMETER["Longitude of natural origin",10,
2117 ##                         ANGLEUNIT["degree",0.0174532925199433],
2118 ##                         ID["EPSG",8802]],
2119 ##           PARAMETER["False easting",4321000,
2120 ##                         LENGTHUNIT["metre",1],
2121 ##                         ID["EPSG",8806]],
2122 ##           PARAMETER["False northing",3210000,
2123 ##                         LENGTHUNIT["metre",1],
2124 ##                         ID["EPSG",8807]]],
2125 ##           CS[Cartesian,2],
2126 ##             AXIS["northing (Y)",north,
2127 ##               ORDER[1],
2128 ##               LENGTHUNIT["metre",1]],
2129 ##             AXIS["easting (X)",east,
2130 ##               ORDER[2],
2131 ##               LENGTHUNIT["metre",1]],
2132 ##           USAGE[
2133 ##             SCOPE["Statistical analysis."],
2134 ##             AREA["Europe - European Union (EU) countries and candidates. Europe - onshore and offshore: A
2135 ##               BBOX[24.6,-35.58,84.73,44.83]],
2136 ##             ID["EPSG",3035]]

```

2137 Die Funktion `st_buffer()` erlaubt es Features zu puffern, mit `st_distance()` kann die Distanz zwischen
2138 Features berechnet werden, mit `st_area()` kann die Fläche eines Features berechnet werden.

2139 Funktionen wie `st_intersection()`, `st_union()` und `st_difference()` erlauben es geometrische Opera-
2140 tionen zwischen unterschiedlichen Features zu berechnen. Für eine ausführliche Diskussion siehe auch hier:
2141 <https://geocompr.robinlovelace.net/geometric-operations.html>.

2142 Normalerweise lesen wir Daten von externen Datenquellen (z.B. ESRI Shapefiles). Das geht mit der Funktion
2143 `st_read()`.

2144 14.5 Rasterdaten in R

2145 Für Rasterdaten gibt es das R-Paket `raster`. Auch hier wollen wir uns wieder auf einige Grundfunktionali-
2146 tätten konzentrieren. Diese umfassen das Einlesen, Zuschneiden, Rechnen und Abfragen von Rastern.

2147 Mit der Funktion `raster()` kann ein Raster in R eingelesen werden.

```

library(raster)
dem <- raster(here::here("data/dem_3035.tif"))

```

2148 `dem` steht für *Digital Elevation Model* und ist ein Raster mit den Seehöhen in Niedersachsen mit einer
2149 500-m-Auflösung. Wir können diese mit der Funktion `res()`¹⁸ abfragen.

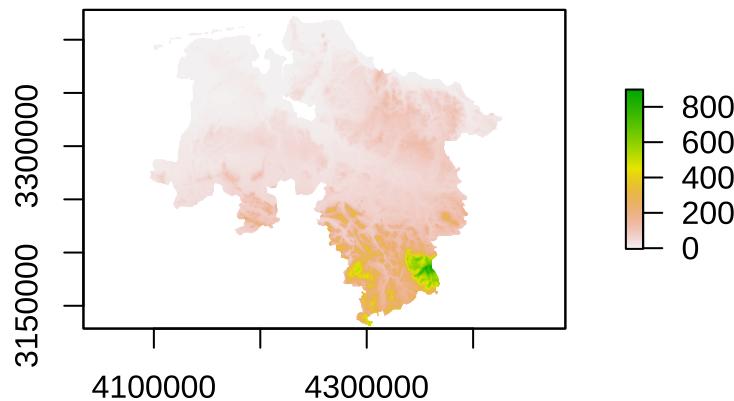
¹⁸kurz für *resolution* also Auflösung.

```
res(dem)
```

2150 ## [1] 500 500

2151 Bzw. wir können den Raster auch plotten.

```
plot(dem)
```



2152

2153 Wenn wir den Raster `dem` auf ein Gebiet zuschneiden wollen (z.B. Göttingen), müssen wir drei Schritte
2154 durchführen. Als erstes müssen wir ein Shapefile für Göttingen einlesen.

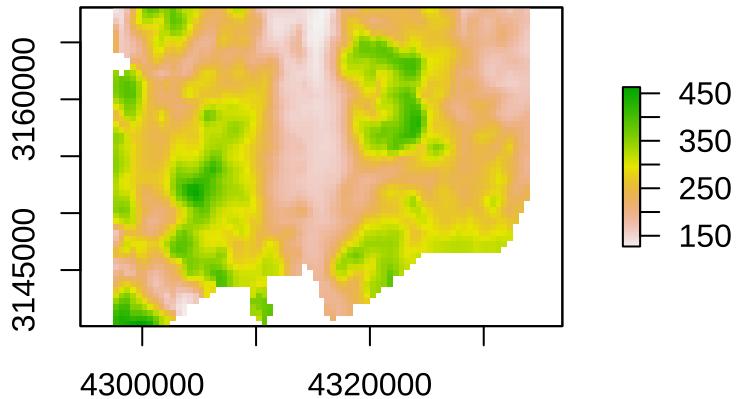
```
goe <- st_read(here::here("data/goettingen/stadt_goettingen.shp"))
```

2155 Dann müssen wir sicherstellen, dass sowohl der Raster `dem` als auch das `sf`-Objekt `goe` im selben KBS sind.
2156 Es bietet es sich in der Regel an, das KBS des Vektors zu transformieren. Mit der Funktion `projectRaster()`
2157 kann das KBS eines Raster transformiert werden.

```
goe <- st_transform(goe, 3035)
```

2158 Mit der Funktion `crop()` kann der Raster `dem` auf Göttingen zugeschnitten werden.

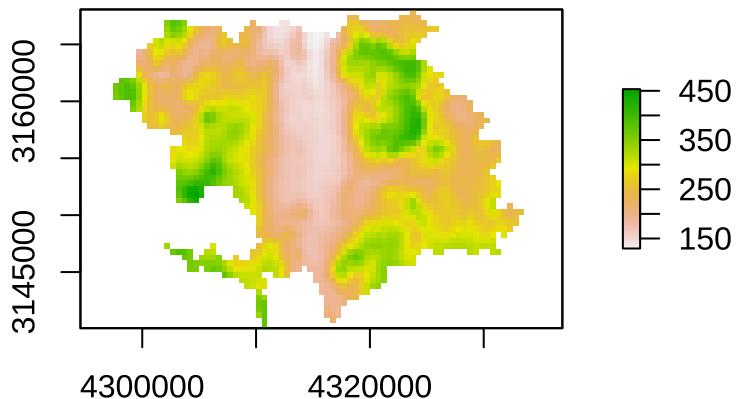
```
dem1 <- crop(dem, goe)
plot(dem1)
```



2159

2160 Der Raster hat jetzt die Größe einer Bounding-Box (BBX) von Göttingen (das ist ein Rechteck, das Göttingen
 2161 umfasst). Mit der Funktion `mask()` kann der Raster auf die genauen Grenzen des Vektors `goe` angepasst
 2162 werden.

```
dem2 <- mask(dem1, goe)
plot(dem2)
```



2163

2164 Wenn wir an bestimmten Punkten den Wert des Rasters abfragen wollen (z.B. an `cities`) von vorhin, dann

2165 gibt es dafür die Funktion `extract`. Dann müssen wir erst sicherstellen, dass `staedte` und `dem` gleichen
 2166 KBS zu grunde liegt. Dafür transformieren wir einfach `staedte` in das KBS von `dem`. Mit der Funktion
 2167 `projection()` erhalten wir das KBS des Rasters.

```
s1 <- st_transform(staedte, 3035)
```

2168 Wenn wir das KBS eines Objektes nicht kennen, können wir auch einfach das KBS übergeben. Der folgende
 2169 Code-Block macht genau das Gleiche mit dem Vorteil, dass wir keinen EPSG-Code angeben müssen.

```
s1 <- st_transform(staedte, projection(dem))
```

2170 Dann können wir für jede Stadt die Seehöhe abfragen:

```
raster::extract(dem, s1)
```

2171 ## [1] 149.18181 57.21486 NA

2172 Mit `raster::extract()` rufen wir *eindeutig* die Funktion `extract()` aus dem Paket `raster` auf. Wir müssen
 2173 das so machen, weil es im Paket `dplyr` auch eine Funktion `extract()` gibt, die wir hier nicht anwenden
 2174 möchten, da sie einen Fehler verursachen würde.

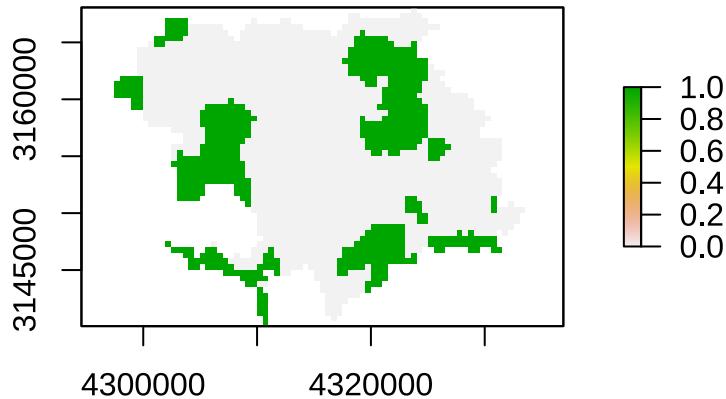
2175 Ein analoges Vorgehen ist auch für Linien und Polygone möglich.

2176 Mit Rastern kann auch einfach gerechnet werden. Wir können z.B. die Seehöhe in Kilometern anstatt Metern
 2177 berechnen:

```
dem_km <- dem / 1e3
```

2178 Auch logische Operationen sind möglich, wenn wir alle Rasterzellen mit einer Seehöhe von mehr als 300 m
 2179 in Göttingen suchen, dann geht das so:

```
dem3 <- dem2 > 300
plot(dem3)
```



2180

2181 Wenn wir jetzt auf die Werte des Rasters `dem3` zugreifen wollen, geht das mit eckigen Klammern.

```
head(dem3[])
```

2182 ## [1] NA NA NA NA NA NA

2183 Das sind erst einmal viele NA-Werte für die ganzen Zellen, die außerhalb von Niedersachsen liegen. Aber wir
2184 können mit so einem Vektor ganz normal arbeiten und z.B. die Fläche des Landes Niedersachsen die eine
2185 Seehöhe von mehr als 500m Seehöhe hat ausrechnen.

```
h <- dem3[]  
sum(h, na.rm = TRUE) / sum(!is.na(h))
```

2186 ## [1] 0.265713

2187

2188 Aufgabe 41: Arbeiten mit Rastern

2190 Verwenden Sie den Raster `wald.tif`, der auf einer 10 m Auflösung den Waldanteil jeder Rasterzelle angibt¹⁹.
2191 Der EPSG-Code für das KBS von `wald.tif` ist 3035. Nehmen Sie an, dass wenn der Waldanteil in einer Raster
2192 größer als 50 % ist, dass die Rasterzelle als Wald klassifiziert werden kann. Wie viel Prozent des Göttinger
2193 Stadtgebietes sind Wald? Wie ändert sich dieser Wert, wenn sie 70 % anstatt 50 % als Schwellenwert für
2194 Wald annehmen?

¹⁹ Die können hier <https://land.copernicus.eu/pan-european/high-resolution-layers/> für ganz Europa bezogen werden

2195

2196 **Aufgabe 42: Studiendesign**

- 2198 Mit der Funktion `st_sample()` können Sie innerhalb oder entlang eines Features zufällige Punkte legen. Das
 2199 Argument `n` steuert die Anzahl Punkte und das Argument `type` wie die Punkte angeordnet werden. Für `type`
 2200 sind für uns die Werte `type = "random"` (komplett zufällig), `type = "regular"` (regelmäßiger Grid) und
 2201 `type = "hexagonal"` von Bedeutung (ein hexagonaler Grid, d.h. ein sechseckiger Raster). Unglücklicherweise
 2202 ist das Ergebnis von `st_sample()` erst eine Geometrie. Um daraus ein vollständiges `sf`-Objekt zu machen
 2203 und problemlos weiter arbeiten zu können, müssen Sie noch einmal die Funktion `st_as_sf()` ausführen.
- 2204 Stellen Sie sich vor, dass wir die tatsächliche Waldbedeckung des Göttinger Stadgebietes **nicht** kennen
 2205 und wir eine Studie durchführen, um den Anteil des Göttinger Stadgebietes, der mit Wald bedeckt ist
 2206 herauszufinden. Erstellen Sie dafür einige unterschiedliche Stichproben (diese können in der Anzahl und
 2207 Anordnung variieren).
- 2208 Berechnen Sie für jedes Stichprobendesign den Anteil an Wald und ein dazugehöriges Konfidenzintervall
 2209 (dieses können Sie mit der Formel $\hat{p} \pm 1.96\sqrt{\frac{\hat{p}(1-\hat{p})}{n}}$ berechnen, wobei \hat{p} der geschätzte Waldanteil ist und n
 2210 die Stichprobengröße). Nehmen Sie an, dass eine Rasterzelle Wald ist, sobald > 50 % der Rasterzelle mit
 2211 Wald bedeckt ist.

2212

2213 **Aufgabe 43: Räumliche Daten**

- 2215 Verwenden Sie den folgenden Datensatz:

```
set.seed(123)
df1 <- data.frame(
  x = runif(100, 0, 100),
  y = runif(100, 0, 100),
  kronendurchmesser = runif(100, 1, 15),
  art = sample(letters[1:4], 100, TRUE)
)
```

- 2216 1. Erstellen Sie ein `sf`-Objekt aus `df1`.
- 2217 2. Puffern Sie jeden Baum mit seinem Kronendurchmesser.
- 2218 3. Berechnen Sie die Kronenfläche jedes Baumes. *Hinweis: Die Funktion `st_area()` könnte dafür hilfreich sein.*
- 2219 4. Welcher Baum hat die größte Kronenfläche?
- 2220 5. Finden Sie für jede Art, den Baum mit der größten Kronenfläche.

2222

2223 **Aufgabe 44: Arbeiten mit räumlichen Daten**

- 2225 1. Lesen Sie das ESRI Shapefile goettingen/stadt_goettingen.shp ein.
2226 2. Wie viele Features befinden sich in dem Shapefile?
2227 3. Welches Koordinatenbezugssystem (KBS) hat das Shapefile?
2228 4. Transformieren Sie das Shapefile in das KBS 3035.
2229 5. Erstellen Sie eine neue Spalte A in der Sie die Fläche jeder Gemeinde/Stadt speichern.
2230 6. Welche Gemeinde/Stadt (Spalte GEN) ist am größten?
2231 7. Wählen Sie nun nur die Stadt Göttingen aus.

2232

2233 **Aufgabe 45: Arbeiten mit räumlichen Daten 2**

- 2235 1. Lesen Sie erneut das ESRI Shapefile goettingen/stadt_goettingen.shp ein.
2236 2. Lösen Sie die Gemeindegrenzen auf (die Funktion `st_union()` könnte hier nützlich sein).
2237 3. Wie groß ist das resultierende Feature?

2238 **15 FAQs (Oft gefragtes)**

2239 **15.1 Arbeiten mit Daten**

2240 **15.1.1 Einlesen von Exceldateien**

- 2241 Mit der Funktion `read_excel()` aus dem Paket `readxl` können Exceldateien direkt in R eingelesen werden.
- 2242 Ein Export als csv-Datei aus Excel ist nicht notwendig.

2243 16 Literatur

- 2244 Ein guter Überblick über viele der angeschnittenen Themen gibt es in dem [R for data science](#), das online
2245 frei zugänglich ist. Das on-line Buch [Hands-On Programming with R]{[https://rstudio-education.github.io/
2246 hopr/index.html](https://rstudio-education.github.io/hopr/index.html)} ist eine nicht-Programmierer freundliche Einführung in R.
2247 McNamara, Amelia, and Nicholas J Horton. 2018. “Wrangling Categorical Data in r.” *The American Stati-
2248 stician* 72 (1): 97–104.
2249 Wickham, Hadley. 2014. “Tidy Data.” *Journal of Statistical Software* 59 (10). [https://doi.org/10.18637/jss.
2250 v059.i10](https://doi.org/10.18637/jss.v059.i10).