

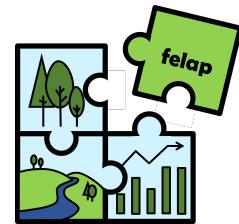
1

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:

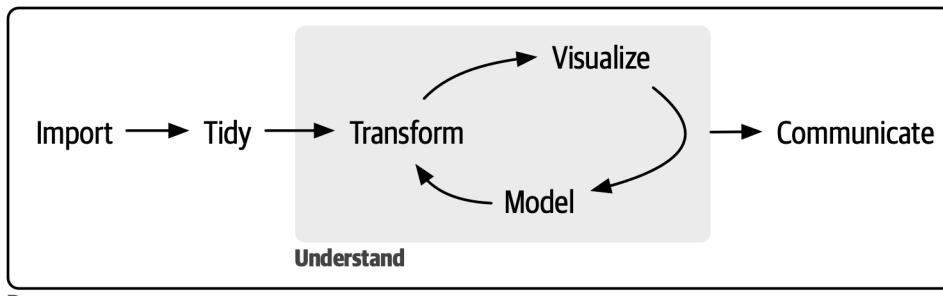
¹⁶ Signer, J. und Husmann, K. (2023) Skript zur Vorlesung Einführung in die Datenanalyse mit R, Georg-August-Universität Göttingen.

¹⁸ Letzte Aktualisierung: 19. Dezember 2023

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 Datentypen	10
52	2.3 Funktionen	11
53	2.4 Datenstrukturen	12
54	2.5 Funktionen	13
55	3 Vektoren	15
56	3.1 Funktionen zum Arbeiten mit Vektoren	17
57	3.2 Statistische Funktionen	19
58	3.3 Beispiel Fotofallen	20
59	3.4 Arbeiten mit logischen Werten	21
60	3.5 Zugreifen auf Elemente eines Vektors (=Untermengen)	23
61	3.6 Der %in%-Operator	25
62	4 Faktoren (factors)	27
63	4.1 Das Paketforcats	29
64	4.1.1 Anpassen der Anordnung von Faktoren	29
65	5 Spezielle Einträge	31
66	5.1 NA	31
67	5.2 NULL	32
68	5.3 Inf	32
69	6 data.frames oder Tabellen	35
70	6.1 Wichtige Funktionen zum Arbeiten mit data.frames	36
71	6.2 Zugreifen auf Elemente eines data.frame	37

72	7 Schreiben und lesen von Daten	41
73	7.1 Textdateien	41
74	8 Erstellen von Abbildungen	43
75	8.1 Base Plot	43
76	8.1.1 Mehrere Panels	49
77	8.1.2 Speichern von Abbildungen	50
78	8.2 Histogramme	51
79	8.3 Boxplots	53
80	8.4 <code>ggplot2</code> : Eine Alternative für Abbildungen	55
81	8.4.1 Multipanel Abbildungen	64
82	8.4.2 Plots kombinieren	67
83	8.4.3 Speichern von plots	71
84	9 Mit Daten arbeiten	72
85	9.1 <code>dplyr</code> eine Einführung	72
86	9.2 Arbeiten mit gruppierten Daten	75
87	9.3 <code>pipes</code> oder <code>%>%</code>	77
88	9.4 Joins	78
89	9.5 ‘long’ and ‘wide’ Datenformate	80
90	9.6 Auswählen von Variablen	82
91	9.7 Einzelne Beobachtungen abfragen (<code>slice()</code>)	84
92	9.8 Spalten trennen	87
93	10 Arbeiten mit Text	89
94	10.1 Arbeiten mit Text	89
95	10.2 Finden von Textmustern	91
96	11 Arbeiten mit Zeit	94
97	11.1 Arbeiten mit Zeitintervallen	96
98	11.2 Formatieren von Zeit	97
99	11.3 Zeitreihen	98

100	12 Aufgaben Wiederholen (for-Schleifen)	103
101	12.1 Schleifen	103
102	12.1.1 Wiederholen von Befehlen mit <code>for()</code>	104
103	12.1.2 Wiederholen von Befehlen mit <code>while()</code>	106
104	12.2 Bedingte Ausführung von Codeblöcken	107
105	13 (R)markdown	109
106	13.1 Markdown Grundlagen	109
107	13.2 R und Markdown	110
108	14 Räumliche Daten in R	112
109	14.1 Was sind räumliche Daten	112
110	14.2 Koordinatenbezugssystem	112
111	14.3 Vektordaten in R	113
112	14.4 Arbeiten mit Vektordaten	114
113	14.5 Rasterdaten in R	116
114	15 FAQs (Oft gefragtes)	122
115	15.1 Arbeiten mit Daten	122
116	15.1.1 Einlesen von Exceldateien	122
117	16 Literatur	123

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**).

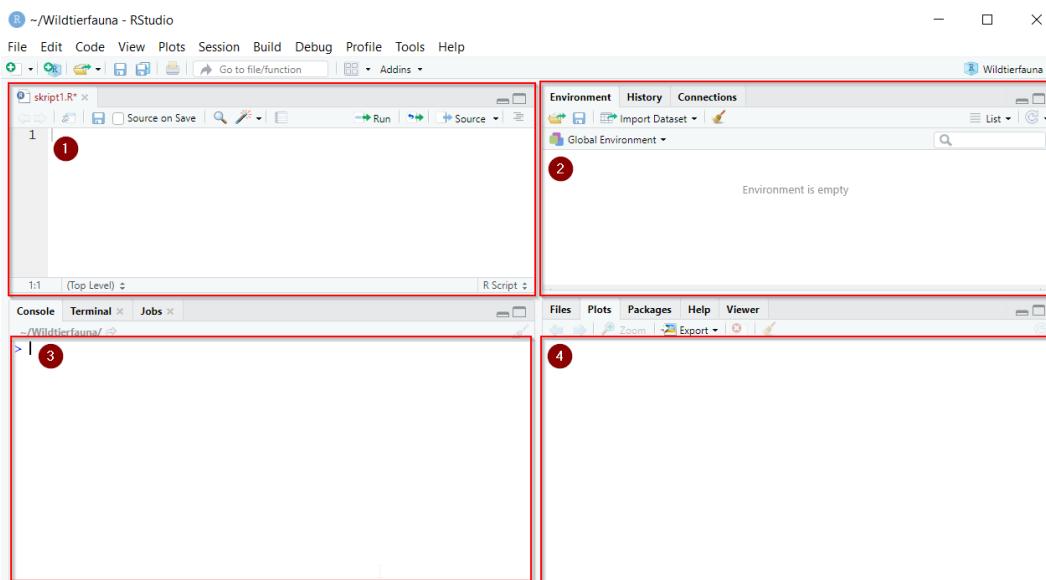


Abbildung 1: RStudio Panes.

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

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

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

10 + 5

154 `## [1] 15`

20 - 10

155 `## [1] 10`

10 * 3

156 `## [1] 30`

100 / 19

157 `## [1] 5.263158`

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

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

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

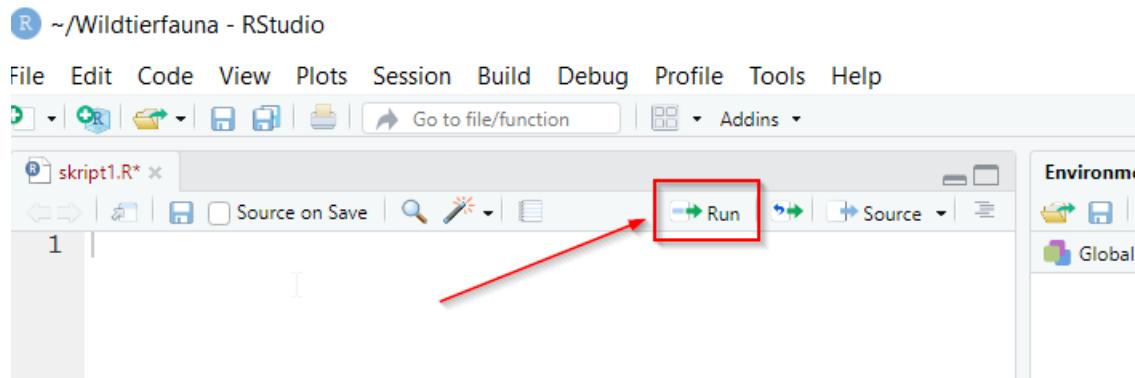


Abbildung 2: Zeilenweises Ausführen von Code in RStudio.

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

181 1.3 Gute Praxis bei der Programmierung

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

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

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

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

197 ## [1] 9

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

204

205 Aufgabe 1: Ausführen von Quellcodes

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

208 Tippen oder kopieren Sie folgenden Code in das Skript:

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

# Einfache Funktion
sqrt(20)
```

209 Führen Sie nun alle Zeilen aus.

2 Variablen, Funktionen und Datentypen

2.1 Variablen beim Programmieren

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

```
alter <- 102
```

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

```
a <- 10
a
```

`## [1] 10`

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

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

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

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

Vorsicht: Groß- und Kleinschreibung muss beachtet werden.

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

`## [1] "Johannes"`

Das Aufrufen der Variable

Name

230 führt zu einem Fehler.

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

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

233 ## [1] 15

```
b / a
```

234 ## [1] 0.5

```
a^b
```

235 ## [1] 1e+05

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

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

237 ## [1] 15

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

238 ## [1] 30

239 Mit der Funktion `rm()` können Variablen, die nicht mehr benötigte Variablen, wieder gelöscht werden.
240 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
```

242 ## [1] TRUE

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

243 ## [1] FALSE

2.2 Datentypen

245 Es wurde bereits erwähnt, dass Daten in Variablen gespeichert werden können. Die Variablen, in denen die
 246 Daten (oder auch Berechnungen oder ganze Funktionen) gespeichert werden, heißen in R Objekte. Wenn
 247 Sie beispielsweise Messwerte einer Fotofalle speichern möchten, dann hat diese Fotofalle einen Namen (z.B.
 248 Kamera1) und nach einiger Zeit im Wald wurden hoffentlich auch einige Fotos aufgenommen. Wir nehmen
 249 einmal an, dass nach drei Wochen 132 Fotos von Rehen gemacht wurden.

250 Wir können jetzt sowohl den Namen der Fotofalle, als auch die Anzahl Fotos die aufgenommen wurden, in
 251 zwei Variablen abspeichern.

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

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

```
fuchs_gesehen <- TRUE
```

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

```
typeof(fuchs_gesehen)
```

263 ## [1] "logical"

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

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

TRUE + TRUE

265 `## [1] 2`

FALSE + FALSE

266 `## [1] 0`

TRUE + FALSE

267 `## [1] 1`

268 2.3 Funktionen

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

sqrt(a)

272 `## [1] 3.162278`

273 Funktionen sind in R in der Regel daran zu erkennen, dass sie immer mit dem Funktionsnamen, gefolgt von
 274 runden Klammern (), aufgerufen werden. Der große Umfang an Funktionen für die statistische Datenana-
 275 lyse und wissenschaftliche Datenverarbeitung ist der Hauptgrund für den Erfolg von R in der Wissenschaft.
 276 Im vorherigen Beispiel wurde die Funktion mit dem Namen `sqrt()` aufgerufen. Das Objekt `a` haben wir be-
 277 reits vorhin definiert (zur Erinnerung `a <- 10`). Die Funktion `sqrt()` arbeitet jetzt mit dem Objekt `a`, das
 278 in diesem Zusammenhang auch **Argument** genannt wird. Argumente sind die Objekte, die eine Funktion
 279 als Input benötigt. Die Hilfeseite jeder Funktion enthält eine Liste aller Argumente. Argumente von Funk-
 280 tionen haben Namen, diese müssen jedoch nicht angegeben werden, wenn die Reihenfolge der Argumente,
 281 wie in der Hilfeseite angegeben, berücksichtigt wird. Im vorherigen Beispiel, haben wir die Funktion `sqrt(a)`
 282 aufgerufen und keinen Argumentnamen angegeben. Aus den Hilfeseiten für die Funktion `sqrt()` (siehe auch
 283 nachfolgender Abschnitt) ist zu entnehmen, dass die Funktion `sqrt()` nur ein Argument mit dem Namen `x`
 284 hat. Das heißt, der vollständige Aufruf der Funktion `x` wäre.

sqrt(x = a)

285 `## [1] 3.162278`

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

- 289 1. In die Konsole ?<Name der Funktion> tippen. Also, wenn wir Hilfe für die Funktion `mean()` möchten,
 290 könnten wir einfach `?mean` in die Konsole tippen.
- 291 2. Analog zu 1), können wir mit der Funktion `help` die Hilfeseite für eine Funktion aufrufen (z.B. wenn
 292 wir wieder die Hilfe für die Funktion `mean()` lesen möchten, dann könnten wir auch `help(mean)` in die
 293 Konsole tippen).
- 294 3. In R Studio kann man auch auf das Help-Tab (Pane 4) klicken und dann einfach eine Funktion suchen
 295 (siehe Abbildung 1).
- 296 4. Wenn der Cursor in RStudio auf einer Funktion ist, kann man mit der Taste F1 die dazugehörige
 297 Hilfeseite aufrufen.

298 2.4 Datenstrukturen

299 Verfolgen wir das Beispiel mit den Fotofallen etwas weiter. Es handelt sich um ein systematisches Monitoring.
 300 D. h., es wurde nicht nur eine Fotofalle ausgebracht, sondern insgesamt 15 Stück. Dieser Umstand erfordert
 301 komplexere Objekte. Nachfolgend sind die Anzahl Rehfotos für jede der 15 Fotofallen aufgeführt: 132, 79,
 302 129, 91, 138, 144, 55, 103, 139, 105, 96, 146, 95, 118, 107.

303 Die Frage, die sich jetzt stellt, ist: Wie kann man diese Daten sinnvoll organisieren? Zusätzlich zur Anzahl
 304 der fotografierten Rehe soll jede Fotofalle eine eindeutige ID haben (`Kamera_1`, ..., `Kamera_15`) und wir
 305 wissen, dass jeweils 5 Fotofallen in drei unterschiedlichen Revieren aufgestellt waren (Fotofalle 1 bis 5 in
 306 Revier A, Fotofalle 6 bis 10 in Revier B und Fotofalle 11 bis 15 in Revier C). Wir könnten für jede Kamera
 307 und jeden Wert ein einzelnes Objekt mit je einem Wert erstellen. Dass das zu Aufwändig wäre, leuchtet
 308 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.
```

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

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

312

313 **Aufgabe 2: Variablen**

314

315 Verwenden Sie die folgenden Daten

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

316 und berechnen sie:

- 317 • 10 * a
- 318 • a / 144 und speichern Sie das Ergebnis in einer neuen Variablen e zwischen.
- 319 • Was ist das Ergebnis von a + b?
- 320 • Was ist das Ergebnis von a + p?

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

321 **2.5 Funktionen**

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

```
sqrt(a)
```

325 `## [1] 1.414214`

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

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

```
sqrt(x = a)  
335 ## [1] 1.414214
```

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

- 339 1. In die Konsole `?<Name der Funktion>` tippen. Also, wenn wir Hilfe für die Funktion `mean()` möchten,
340 könnten wir einfach `?mean` in die Konsole tippen.
- 341 2. Analog zu 1), können wir mit der Funktion `help` die Hilfeseite für eine andere Funktion aufrufen (z.B.
342 wenn wir wieder die Hilfe für die Funktion `mean()` lesen möchten, dann könnten wir auch `help(mean)`
343 in die Konsole tippen).
- 344 3. In RStudio kann man auch auf das Help-Panel klicken und dann einfach eine Funktion suchen (siehe
345 Abbildung 1).
- 346 4. Wenn der Cursor in RStudio auf einer Funktion ist, kann man mit der Taste F1 die dazugehörige
347 Hilfeseite aufrufen.

3 Vektoren

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

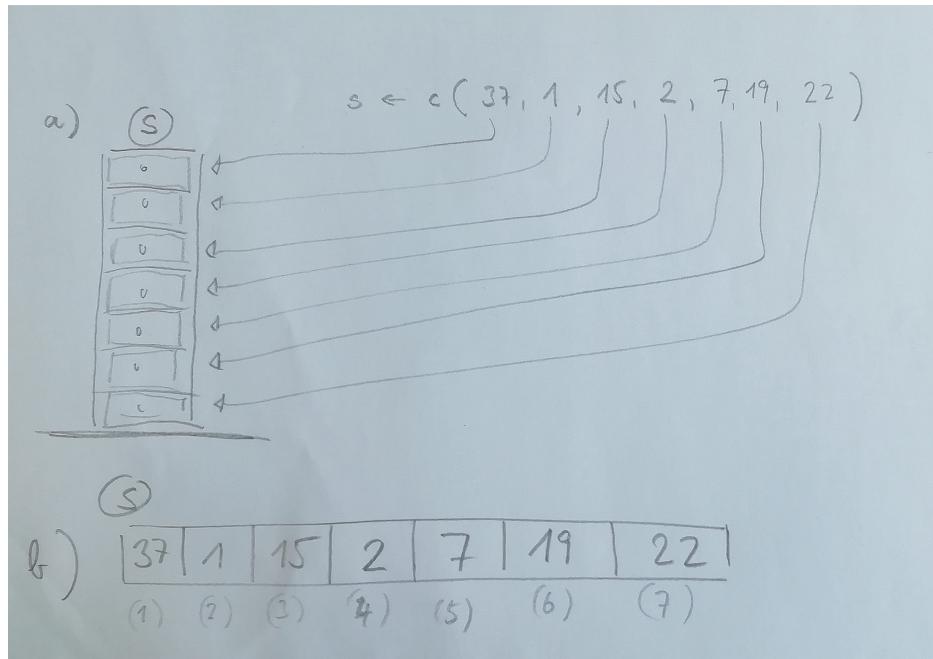


Abbildung 3: Schematische Darstellung eines Vektors in R.

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

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

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

367 sehen:

```
s
```

368 ## [1] 37 1 15 2 7 19 22

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

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

```
s + 10
```

374 ## [1] 47 11 25 12 17 29 32

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

```
s * s
```

378 ## [1] 1369 1 225 4 49 361 484

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

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

382 ## [1] 1 2 3 4 5 6 7 8 9 10

```
(1 : 10)
```

383 ## [1] 1 2 3 4 5 6 7 8 9 10

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

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

385 ## [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.

386

Aufgabe 3: Vektoren erstellen

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

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

393 **3.1 Funktionen zum Arbeiten mit Vektoren**394 Die Funktionen `head()` und `tail()` geben die ersten bzw. letzten `n` Elemente eines Vektors zurück. `n` hat
395 einen voreingestellten Wert von 6, dieser kann natürlich angepasst werden.`head(s)`396 `## [1] 37 1 15 2 7 19``head(s, n = 3)`397 `## [1] 37 1 15``tail(s, n = 2)`398 `## [1] 19 22`399 Die Funktion `length()` gibt die Länge eines Vektors wieder.`length(s)`400 `## [1] 7`401 Der Typ der Elemente eines Vektors kann mit der Funktion `class` abgefragt werden:`class(s)`402 `## [1] "numeric"`403 Die eindeutigen Elemente eines Vektors können mit der Funktion `unique()` abgefragt werden.

unique(s)

```
404 ## [1] 37 1 15 2 7 19 22
```

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

table(s)

```
406 ## s
407 ## 1 2 7 15 19 22 37
408 ## 1 1 1 1 1 1
```

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

rev(s)

```
411 ## [1] 22 19 7 2 15 1 37
```

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

sort(s)

```
413 ## [1] 1 2 7 15 19 22 37
```

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

rep(s, times = 2)

```
415 ## [1] 37 1 15 2 7 19 22 37 1 15 2 7 19 22
```

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

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

```
418 ## [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)  
  
419 ## [1] 1 1 2 2 3 3 4 4
```

420

421 **Aufgabe 4: Arbeiten mit Vektoren**
422

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

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

424 Diese wurden immer abwechselnd mit zwei unterschiedlichen Messgeräten durchgeführt wurden.
425 Erstellen Sie einen Vektor von der Länge 8 mit den Einträgen, die immer abwechselnd G1 und G2 sind und
426 für die zwei Geräte stehen.

427 **3.2 Statistische Funktionen**

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

```
mean(s)
```

431 ## [1] 14.71429

```
median(s)
```

432 ## [1] 15

```
sd(s)
```

433 ## [1] 12.76341

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

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

437 ## [1] 1

```
sample(s, size = 3) # 2 Elemente
438 ## [1] 15 7 22
```

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

441 3.3 Beispiel Fotofallen

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

445 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)
```

446 Als zweites erstellen wir einen Vektor mit den IDs. Zur Erinnerung, diese sollten die Werte `Kamera_1` bis
 447 `Kamera_15` haben. Ein erster Ansatz könnte sein, dass wir einfach 15 Fotofallen schreiben und dann die
 448 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")
)
```

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

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

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

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

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

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

```

ids <- paste(v1, v2, sep = "_")
ids

459 ## [1] "Kamera_1"  "Kamera_2"  "Kamera_3"  "Kamera_4"  "Kamera_5"  "Kamera_6"
460 ## [7] "Kamera_7"  "Kamera_8"  "Kamera_9"  "Kamera_10" "Kamera_11" "Kamera_12"
461 ## [13] "Kamera_13" "Kamera_14" "Kamera_15"

```

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

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

```

464 ## [1] "Revier A" "Revier B" "Revier C" "Revier A" "Revier B" "Revier C"
465 ## [7] "Revier A" "Revier B" "Revier C" "Revier A" "Revier B" "Revier C"
466 ## [13] "Revier A" "Revier B" "Revier C"

```

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

```

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

```

```

469 ## [1] "Revier A" "Revier A" "Revier A" "Revier A" "Revier A" "Revier B"
470 ## [7] "Revier B" "Revier B" "Revier B" "Revier B" "Revier C" "Revier C"
471 ## [13] "Revier C" "Revier C" "Revier C"

```

472

473 *Aufgabe 5: Statistische Funktionen*

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

476 2. Erstellen Sie die folgende Konsolenausgabe:

```
477 ## [1] "Die mittlere Anzahl von Rehfotos beträgt 171.8 Rehe pro Standort."
```

478 3.4 Arbeiten mit logischen Werten

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

- 481 • Gleichheit (`==`)

- 482 • Ungleichheit (\neq)
 483 • Größer ($>$) und kleiner ($<$)
 484 • Größer gleich (\geq) und kleiner gleich (\leq)

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

```
anzahl_rehe > 100
```

488 ## [1] TRUE FALSE TRUE FALSE TRUE TRUE FALSE TRUE TRUE TRUE FALSE TRUE
 489 ## [13] FALSE TRUE TRUE

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

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

492 ## [1] FALSE FALSE FALSE FALSE FALSE TRUE TRUE TRUE TRUE TRUE FALSE FALSE
 493 ## [13] FALSE FALSE FALSE

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

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

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

500 ## [1] FALSE FALSE FALSE FALSE FALSE TRUE FALSE TRUE TRUE TRUE FALSE FALSE
 501 ## [13] FALSE FALSE FALSE

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

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

505 ## [1] TRUE FALSE TRUE FALSE TRUE TRUE TRUE TRUE TRUE TRUE FALSE TRUE
 506 ## [13] FALSE TRUE TRUE

- 507 Das Arbeiten mit logischen Werten kann fürs Erste etwas abstrakt erscheinen, aber wir werden im folgenden
 508 Abschnitt (Abschnitt 3.5) zahlreiche Anwendungsbeispiele dafür sehen.

509

510 **Aufgabe 6: Arbeiten mit logischen Werten**

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

- 513 1. TRUE | FALSE
 514 2. FALSE & TRUE
 515 3. (FALSE & TRUE) | TRUE
 516 4. (2 != 3) | FALSE
 517 5. FALSE + 10
 518 6. TRUE + 10
 519 7. TRUE + 10 == FALSE + 10
 520 8. sum(c(TRUE, TRUE, FALSE, FALSE))

521 **3.5 Zugreifen auf Elemente eines Vektors (=Untermengen)**

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

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

534 1.) Abfragen des zweiten Elements in dem Vektor `anzahl_rehe`:

```
anzahl_rehe[2]
```

535 ## [1] 79

536 2.) Abfragen aller Elemente aus `anzahl_rehe`, die aus dem Revier A stammen.

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

537 ## [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.
```

538 ## [1] 132 79 129 91 138

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

541

542 Aufgabe 7: Zugreifen auf Vektorelemente

544 Erstellen Sie einen neuen Vektor `bhd`

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

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

548

549 Alternativ könnte das gleiche Ergebnis mit einem logischen Vektor erreicht werden. Für eine bessere Über-
 550 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]
```

551 ## [1] 132 79 129 91 138

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

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

555 ## [1] 132 79 129 91 138

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

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

```
558 ## [1] 132 79 129 91 138
```

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

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

```
561 ## [1] 113.8
```

```
562
```

563 Aufgabe 8: logische Werte

```
564
```

565 Verwenden Sie nochmals den Vektor mit den Anzahl Rehen die an unterschiedlichen Fotofallenstandorten
566 fotografiert wurden.

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

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

570 3.6 Der %in%-Operator

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

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

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

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

```
575 ## [1] "FI" "FI"
```

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

```
576 ## [1] "FI" "FI"
```

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

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

```
579 ## [1] "FI" "BU" "BU" "FI"
```

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

```
messungen_arten %in% arten
```

```
584 ## [1] TRUE TRUE TRUE FALSE FALSE FALSE TRUE FALSE FALSE
```

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

```
585 ## [1] "FI" "BU" "BU" "FI"
```

586

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

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

```
LETTERS
```

```
590 ## [1] "A" "B" "C" "D" "E" "F" "G" "H" "I" "J" "K" "L" "M" "N" "O" "P" "Q" "R" "S"  
591 ## [20] "T" "U" "V" "W" "X" "Y" "Z"
```

592 Wählen Sie aus LETTERS nur die Vokale aus.

593 4 Faktoren (factors)

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

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

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

602 ## [1] FI BU FI EI EI FI FI
 603 ## Levels: BU EI FI

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

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

607 ## [1] FI BU FI EI EI FI FI
 608 ## Levels: FI BU EI

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

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

611 ## [1] Fichte Buche Fichte Eiche Eiche Fichte Fichte
 612 ## Levels: Fichte Buche Eiche

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

```
levels(af)
```

615 ## [1] "Fichte" "Buche" "Eiche"

```

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

616 ## [1] Fi Bu Fi Ei Ei Fi Fi
617 ## Levels: Fi Bu Ei

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

af

620 ## [1] Fi Bu Fi Ei Ei Fi Fi
621 ## Levels: Fi Bu Ei

relevel(af, "Bu")

622 ## [1] Fi Bu Fi Ei Ei Fi Fi
623 ## Levels: Bu Fi Ei

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

as.character(af)

626 ## [1] "Fi" "Bu" "Fi" "Ei" "Ei" "Fi" "Fi"
627 Achtung mit der Funktion as.numeric() erhält man die interne Kodierung von Faktoren.

af

628 ## [1] Fi Bu Fi Ei Ei Fi Fi
629 ## Levels: Fi Bu Ei

as.numeric(af)

630 ## [1] 1 2 1 3 3 1 1

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

```

633

634 **Aufgabe 10: Faktoren**

635

- 636 Verwenden Sie den Vektor **staedte** und erstellen Sie einen Vektor mit der Anordnung der **levels** in umgekehrter alphabetischer Reihenfolge.

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

638 **4.1 Das Paket **forcats****

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

- 641 1. Die Anordnung von Levels anzupassen.
642 2. Levels zusammenzufassen oder zu entfernen.
643 3. Labels zu ändern.

644 **4.1.1 Anpassen der Anordnung von Faktoren**

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

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

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

```
factor(a)
```

- 647 ## [1] FI BU FI EI EI FI FI
648 ## Levels: BU EI FI

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

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

- 650 ## [1] FI BU FI EI EI FI FI
651 ## Levels: FI BU EI

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

```
fct_rev(f1)
```

653 ## [1] FI BU FI EI EI FI FI
654 ## Levels: EI BU FI

655 nach der Häufigkeit zu sortieren oder

```
fct_infreq(f1)
```

656 ## [1] FI BU FI EI EI FI FI
657 ## Levels: FI EI BU

658 eine zufällige Sortierung.

```
fct_shuffle(f1)
```

659 ## [1] FI BU FI EI EI FI FI
660 ## Levels: EI FI BU

661 5 Spezielle Einträge

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

- 663 • fehlenden Einträgen NA,
- 664 • leeren Einträgen NULL,
- 665 • undefinierten Einträgen NaN (Not a Number) oder
- 666 • unendlichen Zahlen (Inf).

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

668 5.1 NA

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

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

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

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

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

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

```
is.na(na1)
```

```
677 ## [1] FALSE TRUE FALSE
```

```
na.omit(na1)
```

```
678 ## [1] "foo" "foo"
679 ## attr("na.action")
680 ## [1] 2
681 ## attr("class")
682 ## [1] "omit"
```

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

```
na2 < 3
```

686 ## [1] FALSE FALSE NA

```
1 + NA
```

687 ## [1] NA

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

```
mean(na2)
```

691 ## [1] NA

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

692 ## [1] 4.5

693 5.2 NULL

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

698 5.3 Inf

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

```
10^309
```

701 ## [1] Inf

```
2 * Inf
```

702 ## [1] Inf

1 + Inf

703 ## [1] Inf

3 / 0

704 ## [1] Inf

-3 / 0

705 ## [1] -Inf

3 / Inf

706 ## [1] 0

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

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

709 ## [1] TRUE FALSE FALSE TRUE FALSE

`is.finite(inf1)`

710 ## [1] FALSE TRUE TRUE FALSE TRUE

`inf1 < 3`

711 ## [1] FALSE TRUE FALSE TRUE FALSE

712

Aufgabe 11: Vektoren mit speziellen Einträgen

715 Verwenden Sie den Vektor

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

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

718 Verwenden Sie den Vektor

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

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

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

724 Berechnen Sie den arithmetischen Mittelwert von `foo`.

725 6 data.frames oder Tabellen

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

733 Sie können sich ein `data.frame` wie eine Tabelle aus einem Tabellenkalkulationsprogramm vorstellen. Es gibt
 734 Zeilen in denen die Beobachtungen gespeichert sind und Spalten, die die Merkmale speichern. In unserem
 735 Fall gäbe es 15 Zeilen (eine Zeile für jede Fotofalle) und drei Spalten (jeweils eine Spalte für ID, Anzahl Rehe
 736 und Revier). Der Befehl zum Erstellen eines `data.frames` aus Vektoren in R ist `data.frame()`. Für unser
 737 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

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

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

760 6.1 Wichtige Funktionen zum Arbeiten mit `data.frames`

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

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

```
763 ##           ID anzahl_rehe   revier
764 ## 1 Kamera_1          132 Revier A
765 ## 2 Kamera_2          79 Revier A
```

766 Oder für die letzten 2 Beobachtungen.

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

```
767 ##           ID anzahl_rehe   revier
768 ## 14 Kamera_14         118 Revier C
769 ## 15 Kamera_15         107 Revier C
```

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

```
nrow(monitoring)
```

```
771 ## [1] 15
```

```
ncol(monitoring)
```

```
772 ## [1] 3
```

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

```
str(monitoring)
```

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

779

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

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

785 **6.2 Zugreifen auf Elemente eines `data.frame`**

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

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

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

794 `## [1] 91`

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

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

798 `## [1] 91`

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

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

801 `## [1] 132 79 129 91 138`

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

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

```
804 ##      anzahl_rehe    revier
805 ## 1          132 Revier A
806 ## 2          79 Revier A
807 ## 3          129 Revier A
808 ## 4          91 Revier A
809 ## 5          138 Revier A
```

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

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

```
812 ##           ID anzahl_rehe    revier
813 ## 1 Kamera_1          132 Revier A
814 ## 2 Kamera_2          79 Revier A
815 ## 3 Kamera_3          129 Revier A
816 ## 4 Kamera_4          91 Revier A
817 ## 5 Kamera_5          138 Revier A
```

818

819 Aufgabe 13: Abfragen von Werten

821 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)
)
```

- 822 • Wählen Sie nur die ersten drei Zeilen aus und die erste und zweite Spalte aus.
 823 • Wählen Sie alle Zeilen und die erste und dritte Spalte aus.
 824 • Wählen Sie alle Spalten und die erste, dritte und vierte Zeile aus.

825

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

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

```
monitoring$anzahl_rehe
```

831 ## [1] 132 79 129 91 138 144 55 103 139 105 96 146 95 118 107

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

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

833 ## [1] 132 79 129 91 138 144 55 103 139 105 96 146 95 118 107

834 3. Alle Zeilen und die Spalte explizit angeben.

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

835 ## [1] 132 79 129 91 138 144 55 103 139 105 96 146 95 118 107

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

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

```
monitoring$anzahl_rehe > 100
```

843 ## [1] TRUE FALSE TRUE FALSE TRUE TRUE FALSE TRUE TRUE TRUE FALSE TRUE
844 ## [13] FALSE TRUE TRUE

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

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

```
849 ##          ID anzahl_rehe    revier
850 ## 1   Kamera_1        132 Revier A
851 ## 3   Kamera_3        129 Revier A
852 ## 5   Kamera_5        138 Revier A
853 ## 6   Kamera_6        144 Revier B
854 ## 8   Kamera_8        103 Revier B
855 ## 9   Kamera_9        139 Revier B
856 ## 10  Kamera_10       105 Revier B
857 ## 12  Kamera_12       146 Revier C
858 ## 14  Kamera_14       118 Revier C
859 ## 15  Kamera_15       107 Revier C
```

860

Aufgabe 14: Abfragen von Werten 2

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

- 864 • Alle Spalten für Studierende die Forstwissenschaften studieren.
- 865 • Alle Spalten für Studierende die Chemie oder Physik studieren.
- 866 • Die Spalte `fach` und `semester` für Studierende die 22 oder älter sind.

867 7 Schreiben und lesen von Daten

868 7.1 Textdateien

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

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

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

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

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

head(dat)  
  

##          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
```

897 Es gibt viele Varianten der Funktion `read.table()`. Beispielsweise hat die Funktion `read.csv()` bereits
 898 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.

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

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

903

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

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

909 8 Erstellen von Abbildungen

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

914 8.1 Base Plot

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

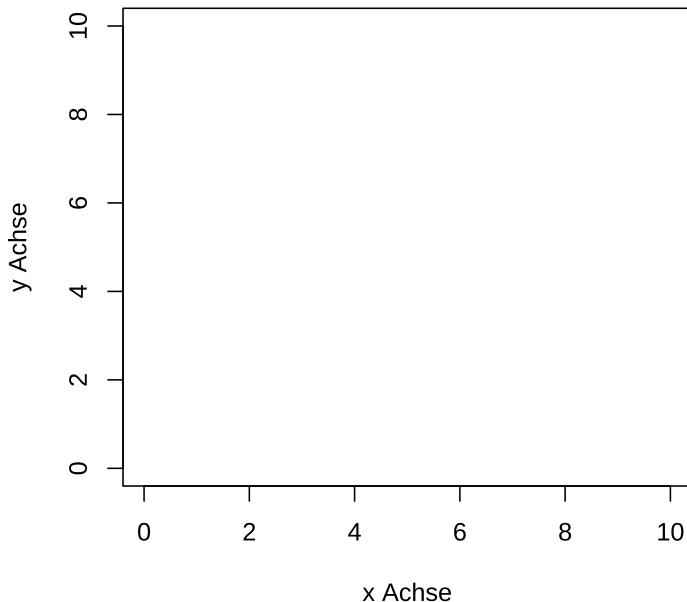
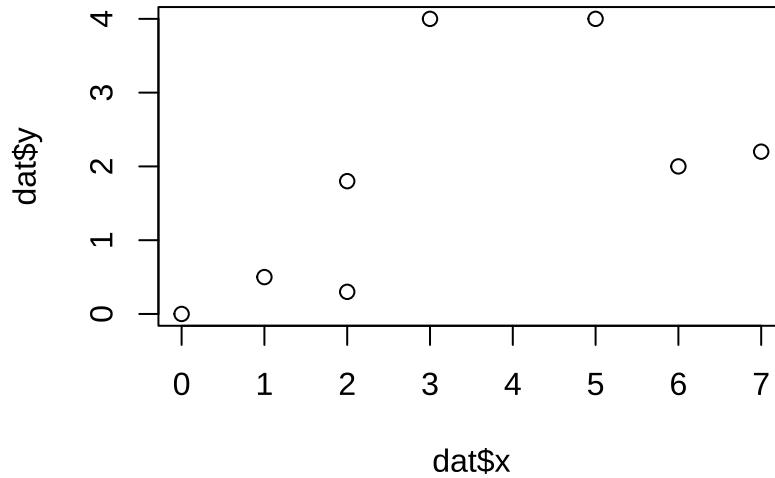


Abbildung 4: Beispiel einer leeren Grafikschnittstelle.

921 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)
```

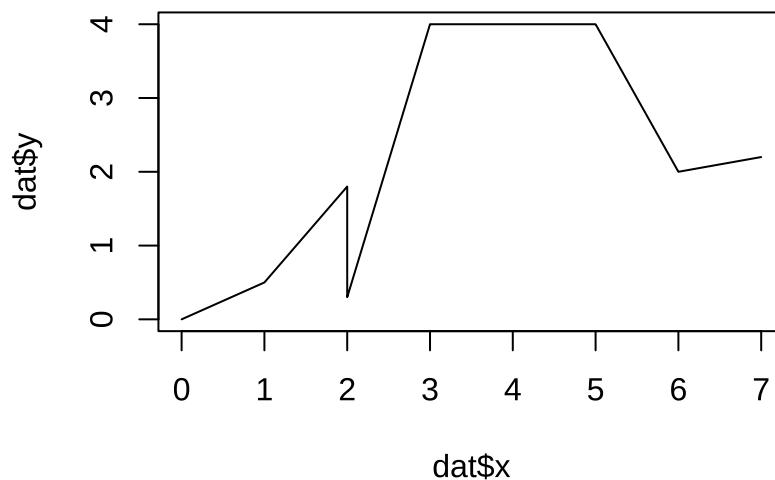
```
)  
  
plot(dat$x, dat$y, type = "p")
```



922

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

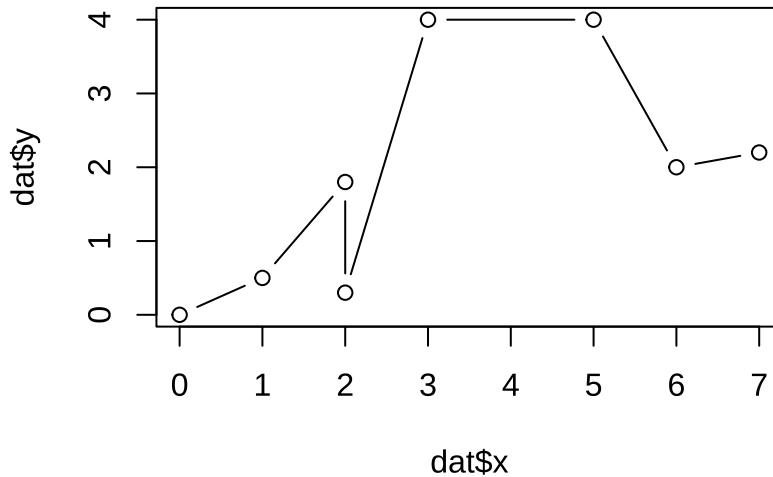
```
plot(dat$x, dat$y, type = "l")
```



925

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

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



927

928 darstellen.

929

930 **Aufgabe 16: Base Plot 1**

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

934

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

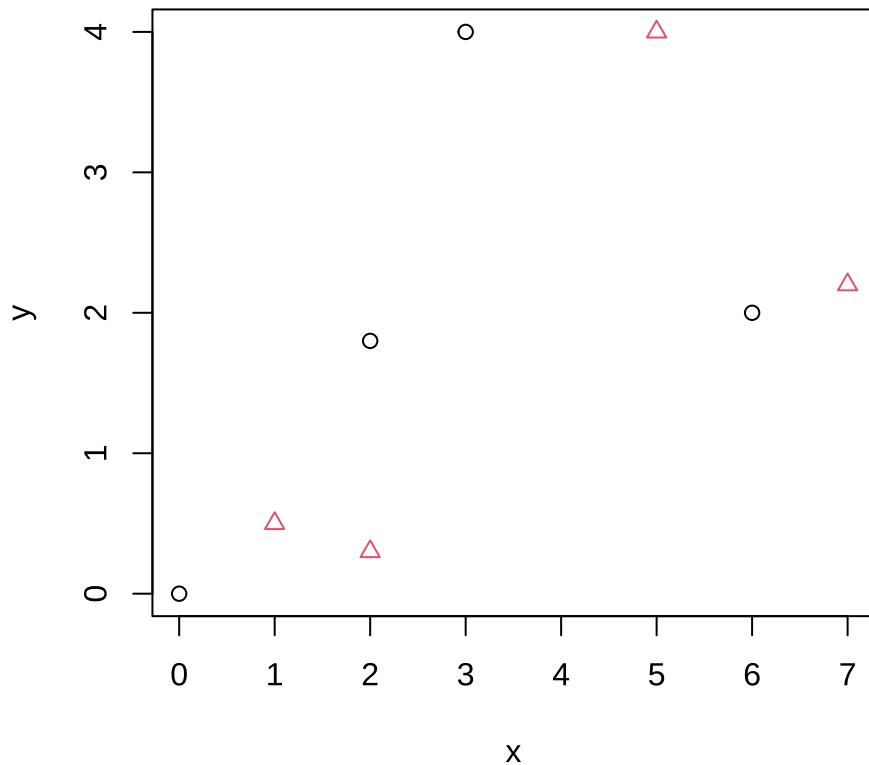
- 939 • `type` - Diagrammtyp
- 940 • `col` - Farbe
- 941 • `main` - Titel
- 942 • `sub` - Untertitel
- 943 • `pch` - Punktsymbol

- 944 • `lty` - Linientyp
 945 • `lwd` - Linienstärke
 946 • `xlab` bzw. `ylab` - Achsenbeschriftungen
 947 • `xlim`, `ylim` - Grenzen der Achsenanschnitte
 948 • `axes` - Sollen die Achsen eingezeichnet werden? Oder leer gelassen werden, um sie nachträglich als
 949 low-level Ebene einzuziehen?
 950 • `ann` - Achsenbeschriftung kann ganz weggelassen werden.

951 Sehen Sie sich die Hilfeseiten `?plot.default()` oder `?par()` an für weitere Informationen. Dort finden Sie
 952 auch eine vollständige Liste der Befehle. Einige Argumente können als Vektor übergeben werden. Hier z. B.
 953 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")
```



955

956 **Aufgabe 17: Anpassen von Plots**

957

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

- 959 • Beschriften Sie die x- und y-Achse sinnvoll.
960 • Fügen Sie eine Überschrift hinzu.
961 • Wählen Sie ein anderes Symbol.
962 • Stellen Sie die Symbole in rot dar.

963

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

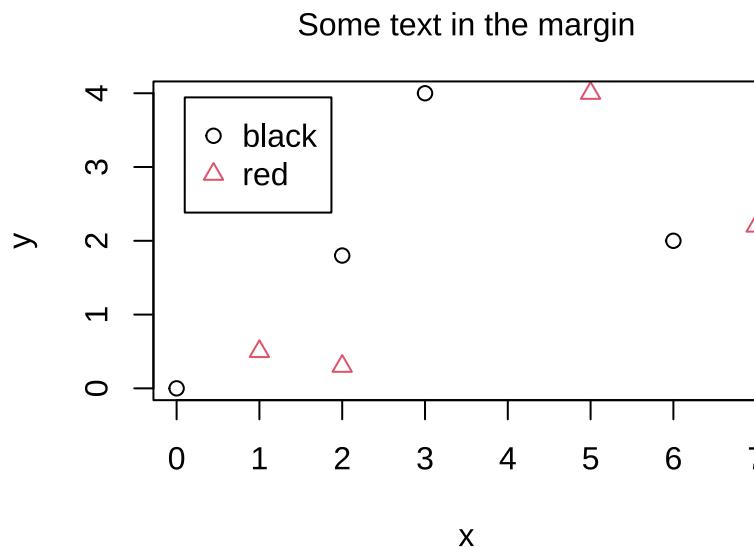
965 Die wichtigsten Funktionen sind

- 966 • `points()` - Fügt Punkte ein
967 • `lines()` - Fügt Linien ein
968 • `text()` - Fügt Text ein
969 • `mtext` - Fügt Text in den Rahmen (`margin`) ein
970 • `legend()` - Fügt eine Legende ein
971 • `abline()` - Fügt eine Gerade ein
972 • `curve()` - Fügt eine mathematische Funktion ein
973 • `arrows()` - Fügt Pfeile ein
974 • `grid()` - Fügt Hilfslinien ein

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

978 Legenden können im base plot nur als Low-Level Funktion hinzugefügt werden. Mit der Funktion `legend`
979 werden die Einträge und die entsprechenden Punktsymbole oder Linientypen dargestellt. In der folgenden
980 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")
```



981

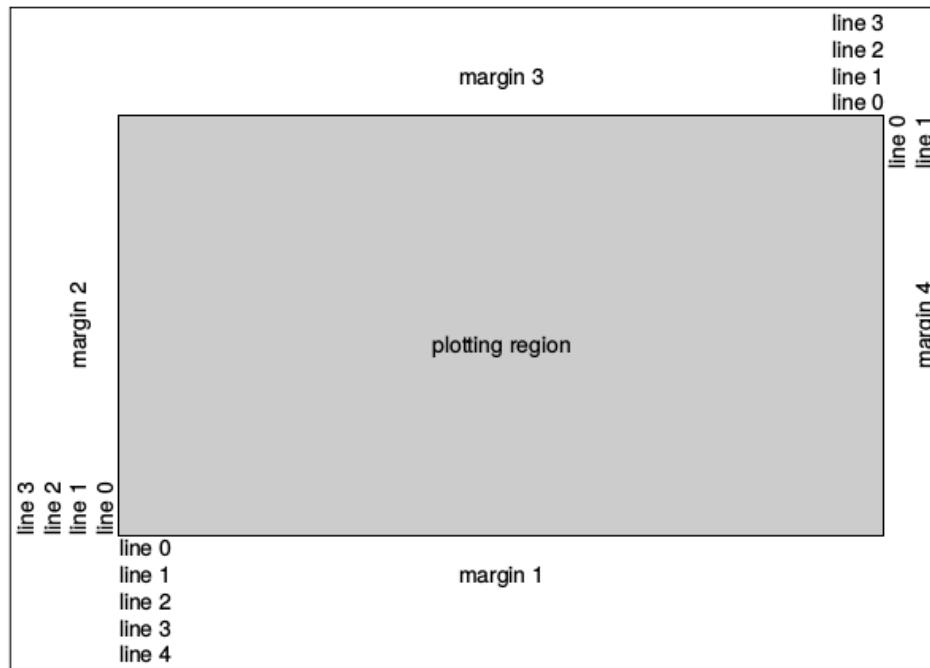


Abbildung 5: Grafikregionen eines base plots in R.

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

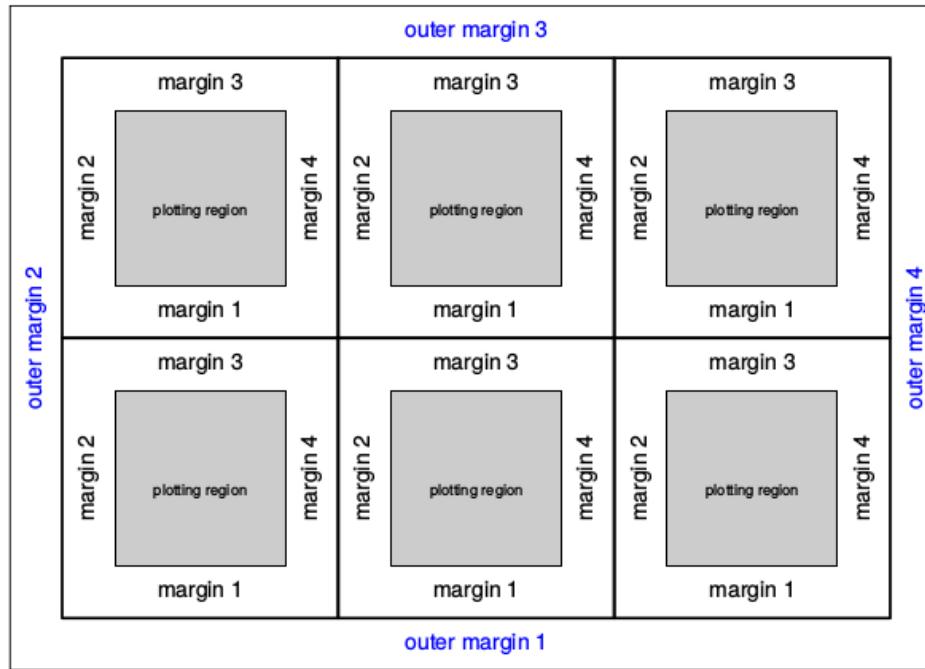


Abbildung 6: Schematischer Aufbau mehrere Diagramme in einem plot am Beispiel einer 3 x 2 Grafik.

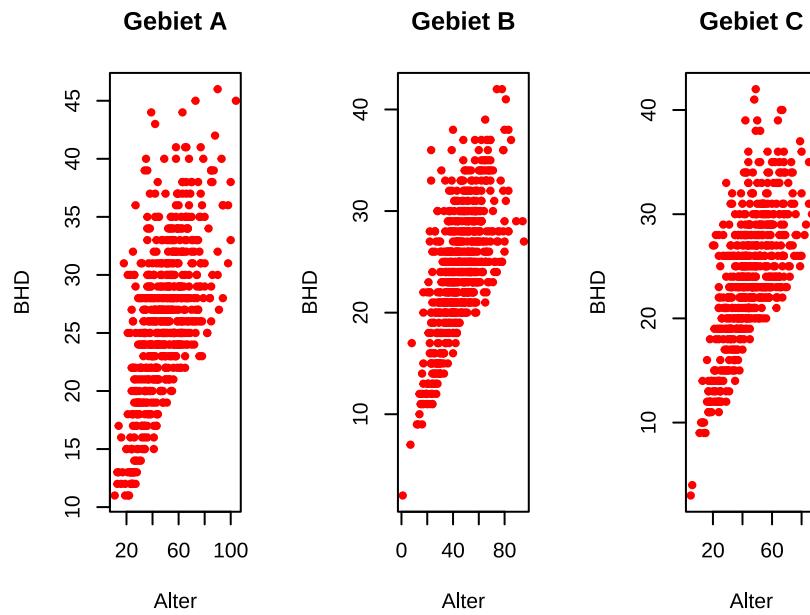
986 8.1.1 Mehrere Panels

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

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

990 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")
```



991

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

994 8.1.2 Speichern von Abbildungen

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

- 999 • `pdf()` oder
- 1000 • `postscript()`.

1001 Beispiele für Rastergrafiken sind

- 1002 • `png()`,
- 1003 • `bmp()` oder
- 1004 • `jpeg()`.

1005 Die Grafiksschnittstelle ist dann Ihre “Leinwand”. Mit dem Befehl `dev.off()` trennen Sie die Verbindung
1006 zur Schnittstelle wieder. Ihre “Leinwand” wird also wieder geschlossen. So lange die Schnittstelle geöffnet ist
1007 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

```

1008 Achtung, wenn Sie die Funktion `dev.off()` nicht aufrufen, werden alle nachfolgenden Plots in die gleiche
 1009 Datei geschrieben. Falls Sie nach einem Versuch einen Plot zu speichern plötzlich keine weiteren Plots mehr
 1010 sehen, führen Sie einige Mal die Funktion `dev.off()` aus.

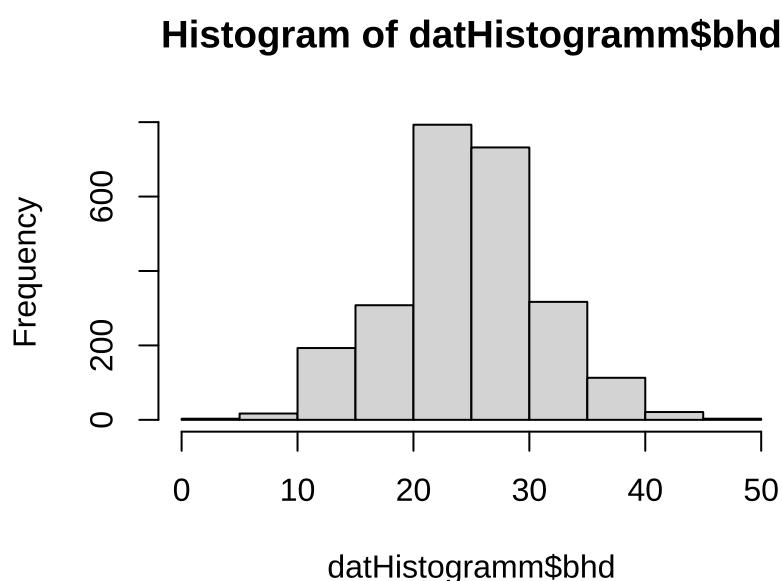
1011 8.2 Histogramme

1012 Neben den Streudiagrammen (*Scatterplots*, oder auch einfach x-y Diagramm) sind *Histogramme* in der
 1013 angewandten Datenanalyse ein weiterer wichtiger Abbildungstyp. An Histogrammen wird die Häufigkeit
 1014 von Beobachtungen nach Gruppen dargestellt. Sie sind deshalb so wichtig, weil man aus ihnen relevante
 1015 Informationen über die Verteilung der Daten ablesen kann und somit einiges über die Messungen erfährt,
 1016 das für die weitere Datenanalyse relevant sein kann. So werden auf einen Blick der Zusammenhang von
 1017 Beobachtungshäufigkeit und Streuung deutlich, sowie auch die Form der Verteilung und ihre Schiefe. Die
 1018 Interpretation werden wir bei den Boxplots noch weiter vertiefen.

```

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

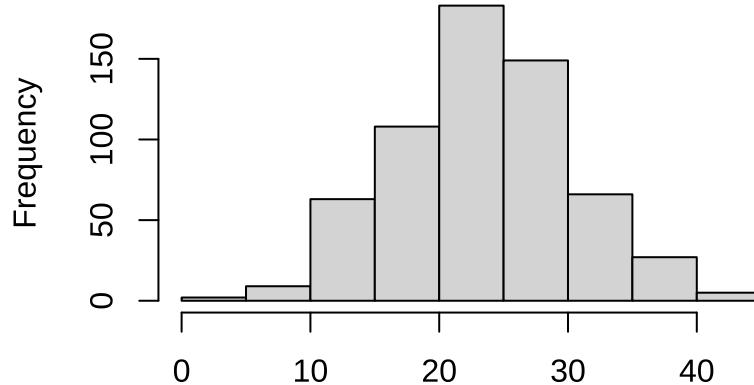
```



1019

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

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

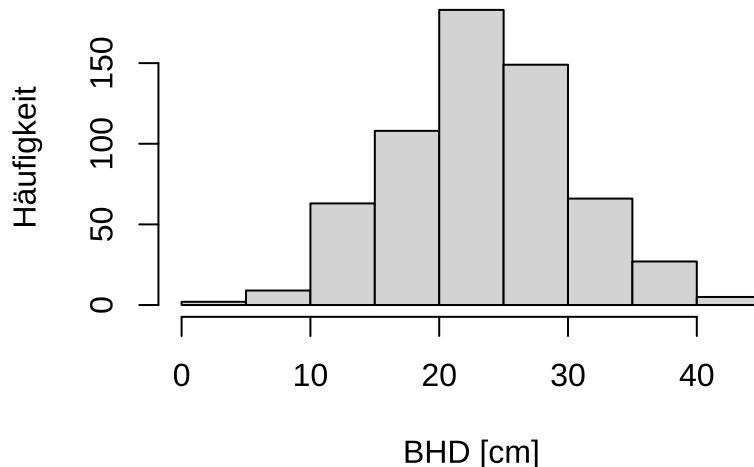


datHistogramm\$bhd[datHistogramm\$art == "EI"]

1020

```
# 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

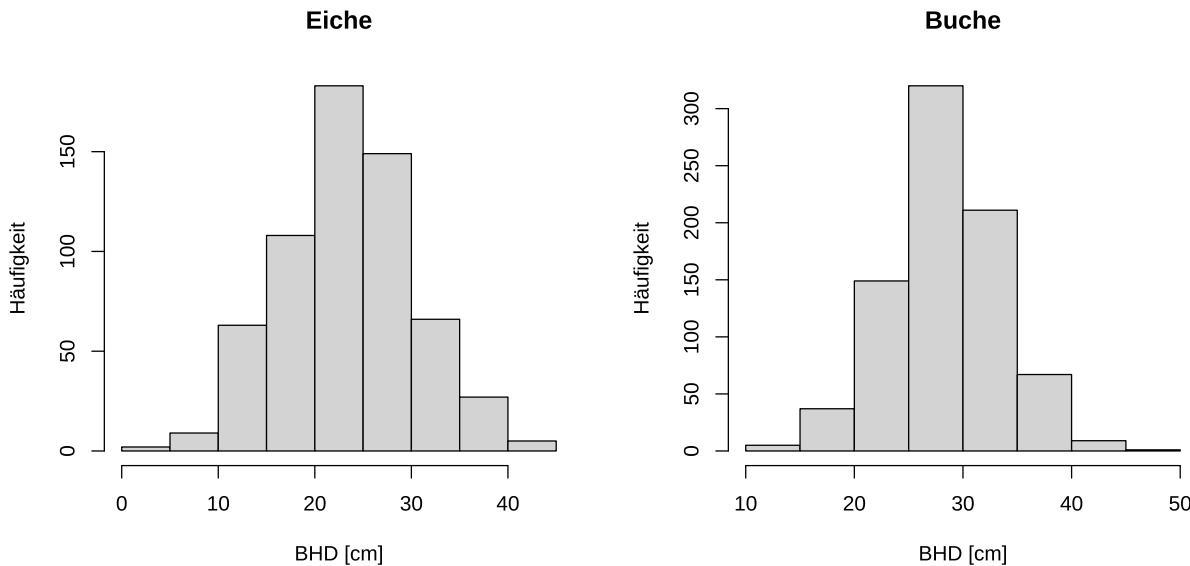


1021

1022 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")
```

1023



1024

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

1025 8.3 Boxplots

1026 Oft möchte man die Verteilung einer stetigen Variablen in Abhängigkeit einer diskreten Variable beschreiben
 1027 oder visualisieren. Ein Beispiel dafür wäre die BHD-Verteilung in Abhängigkeit der Baumarten. Eine häufige
 1028 Darstellungsform für solche Daten sind *Boxplots*. Sie stellen die durchschnittliche Ausprägung einer stetigen
 1029 Variable und ihre Schwankung kompakt dar.

1030 Boxplots bestehen aus drei Komponenten:

- 1031 1. Eine *Box*, die den Bereich zwischen 0.25 und 0.75 Percentil abdeckt, diese Distanz wird auch die IQR
 1032 (Interquartile Range), bezeichnet. Zusätzlich wird die Box durch den Median (als dicke horizontale Linie)
 1033 unterteilt.
- 1034 2. Einzelne Punkte Ausreißer. Als Ausreißer werden Punkte bezeichnet, die $> 1.5 \text{ IQR}$ vom unteren oder
 1035 oberen Ende der Box entfernt sind.

- 1036 3. Eine senkrechte Linie von jeder Seite der Box bis zum letzten “Nicht-Ausreißer-Punkt”. Also der letzte
 1037 Punkt, der $> 1.5IQR$ aber nicht > 0.75 bzw. < 0.25 Percentil ist. Diese Linie wird auch als *Whisker*
 1038 bezeichnet.

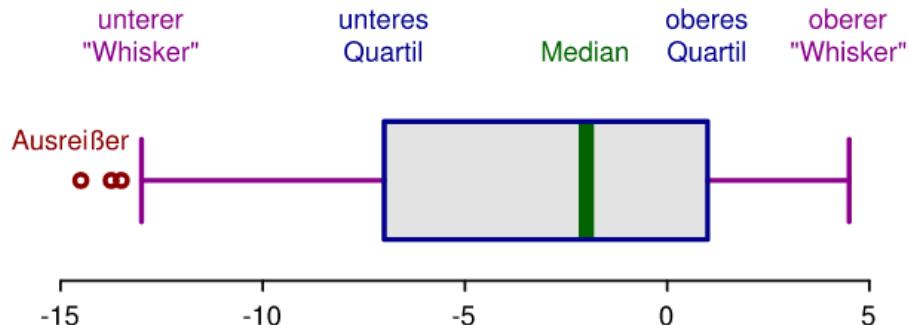
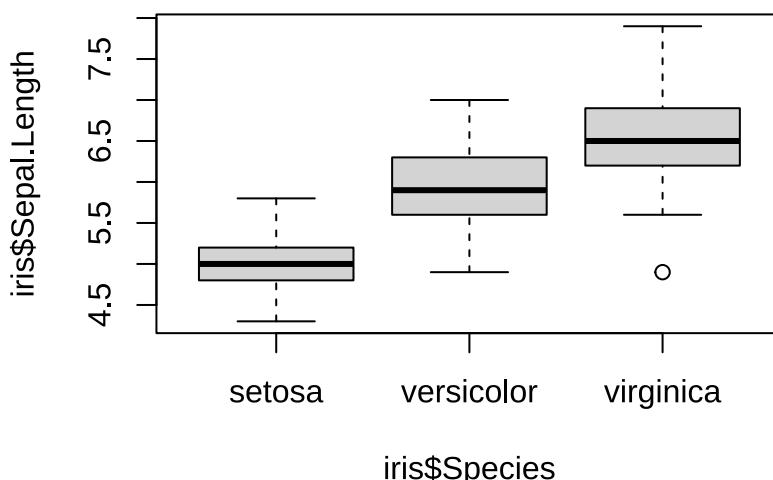


Abbildung 7: Schematische Darstellung eines Boxplots (Quelle: Von RobSeb - Eigenes Werk, CC BY-SA 3.0, <https://commons.wikimedia.org/w/index.php?curid=14697172>).

1039 Mit R kann mit der Funktion `boxplot()` ein Boxplot erstellt werden. Diese Funktion kann in zwei unter-
 1040 schiedlichen Ausprägungen verwendet werden.

- 1041 1. `boxplot(x)` erzeugt einen Boxplot für die Variable `x`.
 1042 2. `boxplot(x ~ y)` erzeugt einen oder mehrere Boxplots für `x` aber gruppiert nach `y`, dabei sollte `y` eine
 1043 kategoriale Variable sein. `x` und `y` können auch die Spaltennamen eines `data.frames` sein, dann muss
 1044 das `data.frame` mit dem Argument `data` zusätzlich übergeben werden.

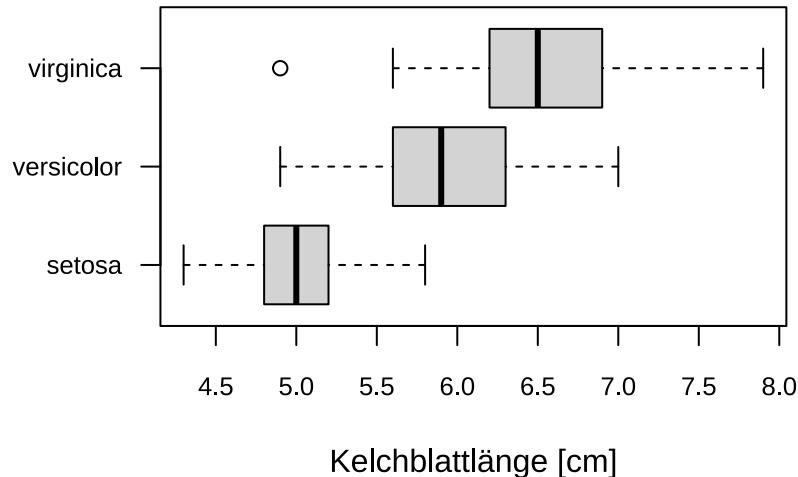
```
boxplot(iris$Sepal.Length ~ iris$Species)
```



1045

1046 Etwas eleganter ist es wenn wir das Argument `data` verwenden und den Plot etwas anpassen. Diese Schreib-
 1047 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  
)
```



1048

1049

1050 Aufgabe 18: Boxplots

1051

- 1052 • Lesen Sie erneut den Datensatz `daten/bhd_1.txt` ein und speichern Sie diesen in die Variable
 1053 `(dat_bhd)`.
- 1054 • Wie viele BHD-Messungen gibt es für jedes Gebiet?
- 1055 • Erstellen Sie für jedes Gebiet einen Plot

1056 Erstellen Sie einen Plot mit 3 Subplots, jeweils mit einem Boxplot für die ersten drei Studiengebiete, in dem
 1057 der BHD für jede Baumart dargestellt wird.

1058 8.4 ggplot2: Eine Alternative für Abbildungen

1059 `ggplot2` ist ein alternatives Plotting-System in *R*. Sie können mit `ggplot2` also grundsätzlich Abbildungen
 1060 mit dem selben Inhalt erstellen, wie mit Base Plots. Die Syntax und die optische Darstellung unterscheiden
 1061 sich jedoch grundsätzlich. `ggplot2` basiert auf den *grammar of graphics* von Leland Wilkinson. Die Idee ist,

1062 alle nötigen Informationen der Abbildung miteinander zu verknüpfen. *ggplot2* ist also diametral zu Base
 1063 Plots. Während Sie in base plot alle Elemente selbst definieren, ist die Idee von *ggplot2*, dass Sie nur die
 1064 Daten und den Diagrammtypen übergeben und die Funktion selbst eine schöne Abbildung erstellt. Selbstver-
 1065 ständlich können Sie aber auch in *ggplot2* viele Einstellungen vornehmen. Im base plot sehen Abbildungen
 1066 zunächst sehr karg und etwas unfertig aus. Sie müssen viele Einstellungen vornehmen, um eine publizierfä-
 1067 hige Grafik zu produzieren. In *ggplot2* sollen auch die einfachste Abbildungen schon ästhetisch sein. Mit
 1068 diesen gebündelten Informationen kann *ggplot2* die Abbildung automatisch verschönern. So werden bspw.
 1069 die Legenden automatisch erzeugt und auch die Formatierungen automatisch an die Datenlage angepasst.
 1070 *ggplot2* nimmt der*dem Entwickler*in also Arbeit ab. Dadurch sind die Abbildungen schon ohne viel Nach-
 1071 arbeit schick. Nachteil ist, dass der*dem Entwickler*in weniger Möglichkeiten zur Einstellung zur Verfügung
 1072 stehen und nuterspezifische Sonderwünsche somit schwerer umsetzbar sind. Sehen Sie sich das *Cheatsheet* zu
 1073 *ggplot2* an. Es ist in RStudio unter [Help > Cheatsheets](#) zu finden.

1074 Bei *ggplot2* sind Anweisungen zu den Daten und Anweisungen zur Darstellung voneinander getrennt. Die
 1075 Daten werden in den Ästhetikbefehl übergeben und dort klassifiziert. Dann folgen die Darstellungsanweisun-
 1076 gen. Ähnlich wie bei Base Plots, werden die Grafikelemente ebenenweise nacheinander programmiert, jedoch
 1077 mit einem `+` verbunden. Und hier liegt der wesentliche Unterschied zu Base Plots. Durch die `+` werden die
 1078 Ebenen zu einem Befehl verbunden und damit gleichzeitig erstellt.

1079 Die Erweiterung wird zunächst geladen⁷. Wir laden außerdem den Datensatz `iris`. Der Datensatz ist in R
 1080 fest integriert. Siehe `?iris` für mehr Informationen.

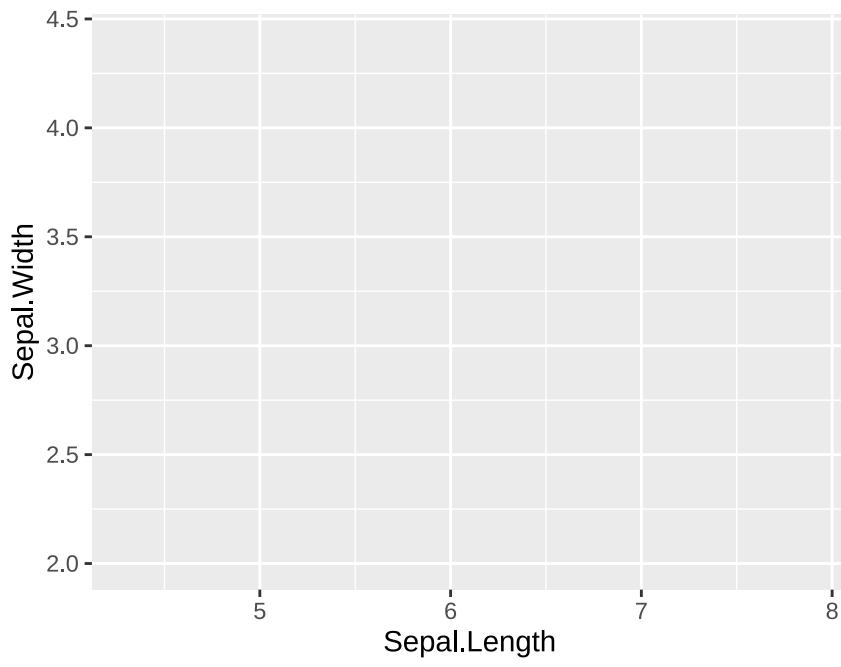
```
library(ggplot2)
head(iris)
```

```
1081 ##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
1082 ## 1          5.1       3.5      1.4       0.2  setosa
1083 ## 2          4.9       3.0      1.4       0.2  setosa
1084 ## 3          4.7       3.2      1.3       0.2  setosa
1085 ## 4          4.6       3.1      1.5       0.2  setosa
1086 ## 5          5.0       3.6      1.4       0.2  setosa
1087 ## 6          5.4       3.9      1.7       0.4  setosa
```

1088 Die Ästhetik wird bspw. folgendermaßen definiert.

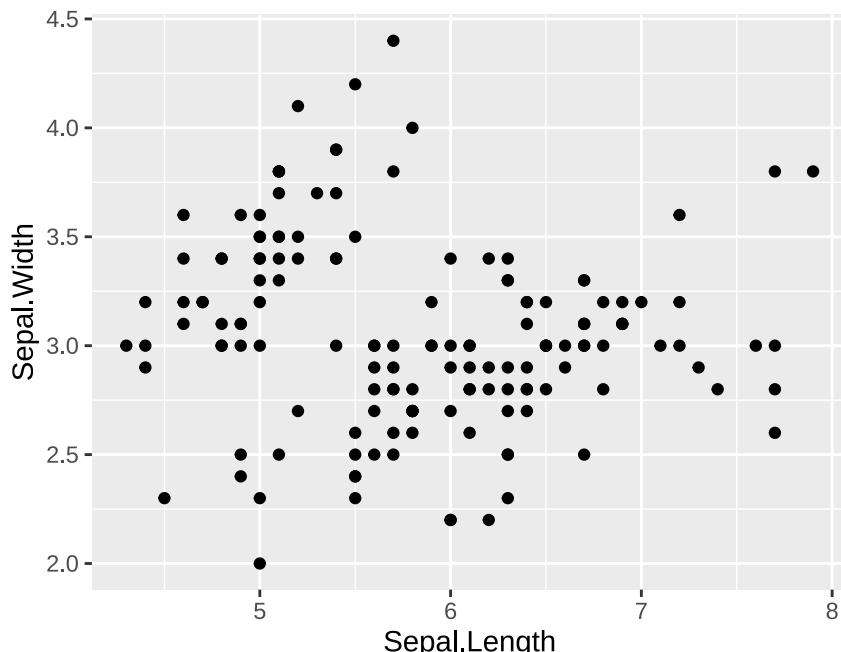
```
ggplot(iris, aes(x = Sepal.Length, y = Sepal.Width))
```

⁷Wir haben bis jetzt immer 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.



1090 Dieser Befehl zeichnet noch keine Daten. Die Daten werden lediglich herangezogen, um einen leeren Plot für
 1091 die Daten zu erstellen. In dem Beispiel wird die Variable `Sepal.Length` aus dem `data.frame iris` als x und
 1092 `Sepal.Width` als y Variable definiert. Diese Informationen stehen den folgenden Layern nun zur Verfügung,
 1093 sodass nach den + nur noch x und y verwendet werden müssen. Um bspw. einen Scatterplot zu erstellen
 1094 wird ein `geom_point()` Layer hinzugefügt. x und y werden automatisch an `geom_point` übergeben. Weitere
 1095 Einstellung sind in diesem Beispiel nicht notwendig, wären jedoch möglich. Siehe `?geom_point()`.

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



1097

1098 **Aufgabe 19: Abbildungen mit *ggplot2***

10991100 Verwenden Sie die Daten aus Aufgabe 16 und erstellen Sie einen Scatterplot mit *ggplot2* wie in Aufgabe 16.

1101

1102 Wir haben mit der Funktion `geom_point()` demm Plot eine Punktgeometrie hinzugefügt. Es gibt noch viele
1103 weitere Geometrien. Die wichtigsten sind:

- 1104 • `geom_line()` für eine Linie.
- 1105 • `geom_histogram()` um ein Histogramm zu erstellen.
- 1106 • `geom_boxplot()` um einen Boxplot zu erstellen.
- 1107 • `geom_bar()` um ein Säulendiagramm zu erstellen.

1108 Welche Geometrie die richtige ist, richtet nach dem Typ der darzustellenden Variablen. Beispielsweise bie-
1109 tet sich `geom_point()` an, wenn man zwei kontinuierliche Variable darstellen möchte. Wenn man hinge-
1110 gen die Verteilung von einer kontinuirlichen Variable darstellen möchte, dann bietet sich ein Histogramm
1111 (`geom_histogram()`) oder auch eine geschätzte Dichte (z.B. `geom_density()`) an.

1112

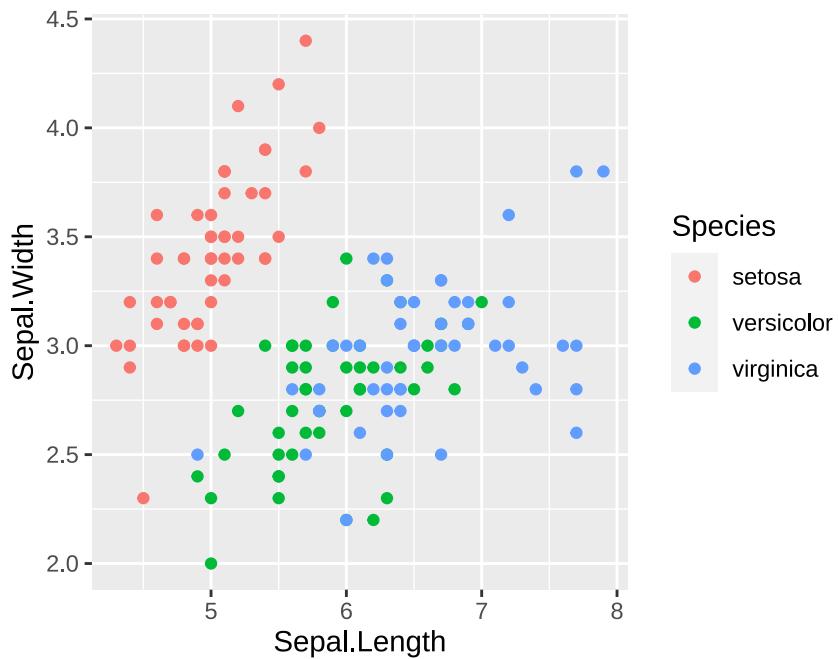
1113 **Aufgabe 20: Abbildungen mit *ggplot2***

11141115 Verwenden Sie die den Iris Datensatz und erstellen Sie mit *ggplot2* einen Plot der die Verteilung der Länge
1116 der Kelchblätter zeigt (Spalte `Sepal.Length`).

1117

1118 Eine der Stärken von *ggplot2* ist, dass man den Wert unterschiedlicher Variable auf unterschiedlichen
1119 Komponenten des Plots abbilden kann. Wir haben bis jetzt ein bzw. zwei Variable auf der x- und y-Achse
1120 abgebildet. Wir können aber ein weitere Variablen verwenden um das Aussehen des Plots zu beeinflussen.
1121 Beispielsweise können wir die Farbe der Punkte (für `geom_point()`) mit dem Argument `col` beeinflussen.

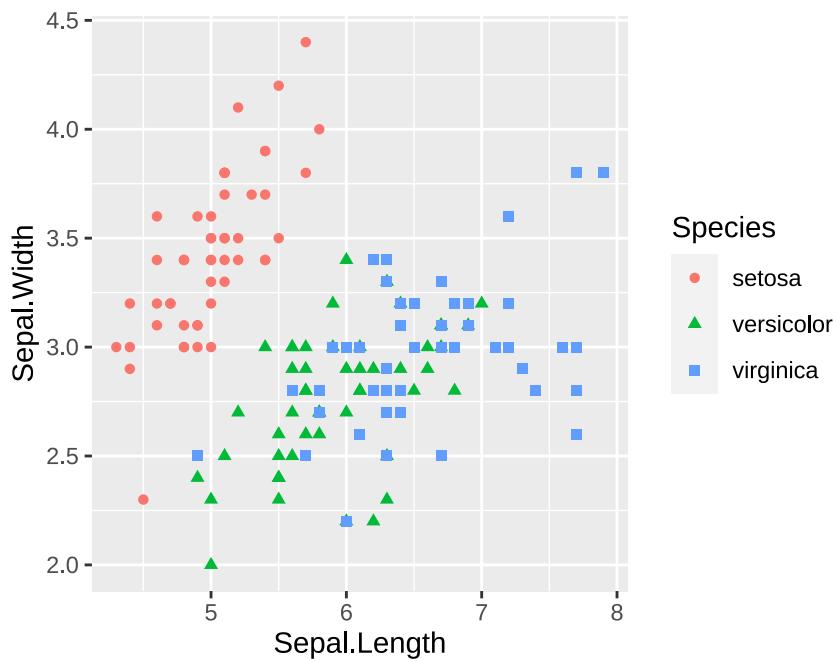
```
ggplot(iris, aes(x = Sepal.Length, y = Sepal.Width, col = Species)) +  
  geom_point()
```



1122

- 1123 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()
```

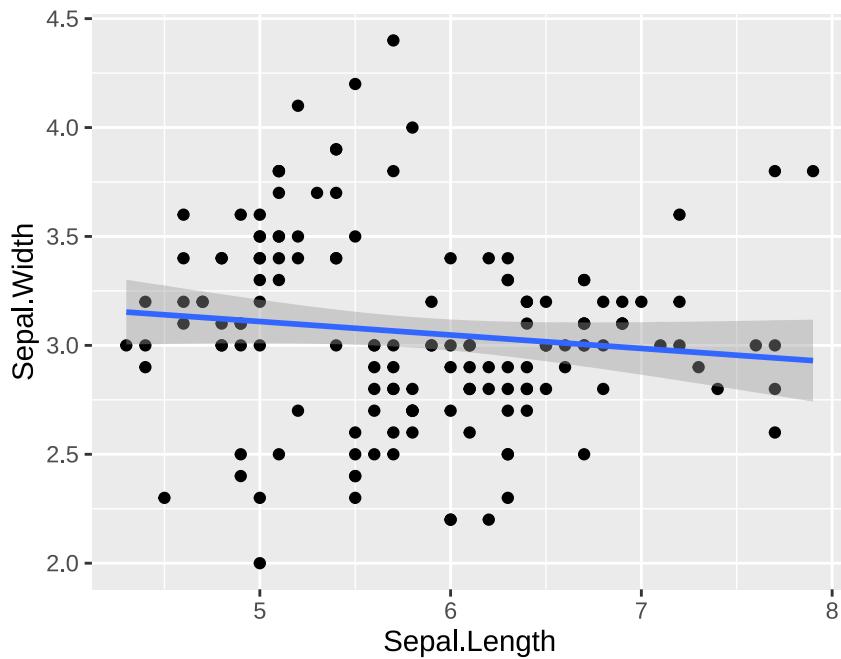


1125

⁸Natürlich könnte man auch die Farbe anpassen.

- ¹¹²⁶ In dem Plot ist die Information zu der Art redundant (einmal als Farbe und einmal Symbolart).
- ¹¹²⁷ Ein weitere sehr nützliche Geometrie ist `geom_smooth()`, die es erlaubt eine Trendlinie hinzuzufügen.

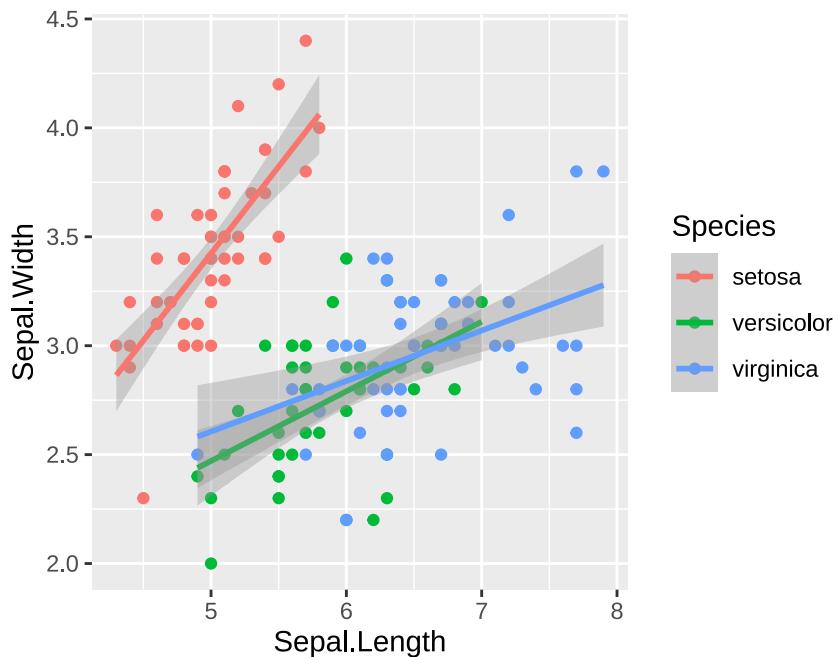
```
ggplot(iris, aes(x = Sepal.Length, y = Sepal.Width)) +
  geom_point() + geom_smooth(method = "lm")
```



¹¹²⁸

- ¹¹²⁹ 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")
```



1132

1133

Aufgabe 21: Anpassen von Plots

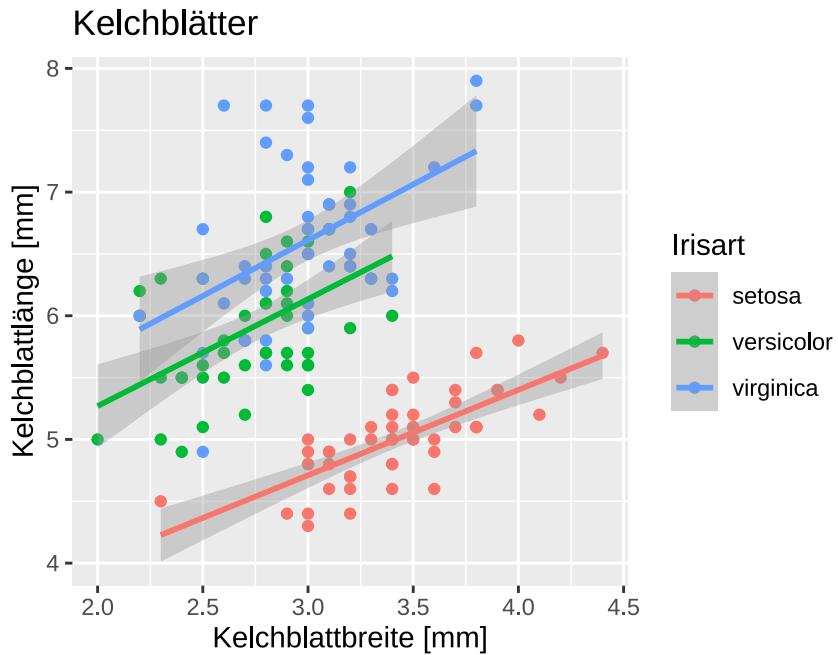
- 1134 Lesen Sie den Datensatz `data/bhd_1.txt` ein und erstellen Sie einen Boxplot für die Verteilung des BHDs
 1135 für jede Baumart. In einem zweiten Schritt verwenden Sie erst `col = gebiet` und dann `fill = gebiet`.
 1136 Welchen Unterschied stellen Sie fest?

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

1139

- 1140 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")
```



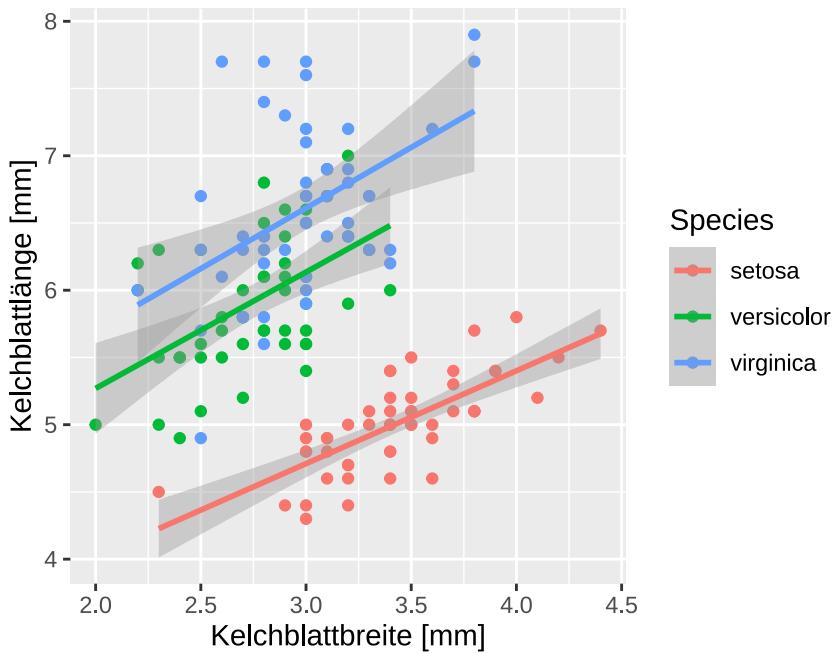
1141

- 1142 Statt einen langen Befehl zu tippen, kann ein `ggplot()` auch zwischengespeichert und wieder aufgerufen bzw.
1143 angepasst werden. Das ist vor allem sinnvoll, wenn mehrere Abbildungen auf dem selben Zwischenergebnis
1144 aufbauen sollen.

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

- 1145 Wir können jetzt mit `p1` weiter arbeiten und beispielsweise eine Beschriftung hinzufügen.

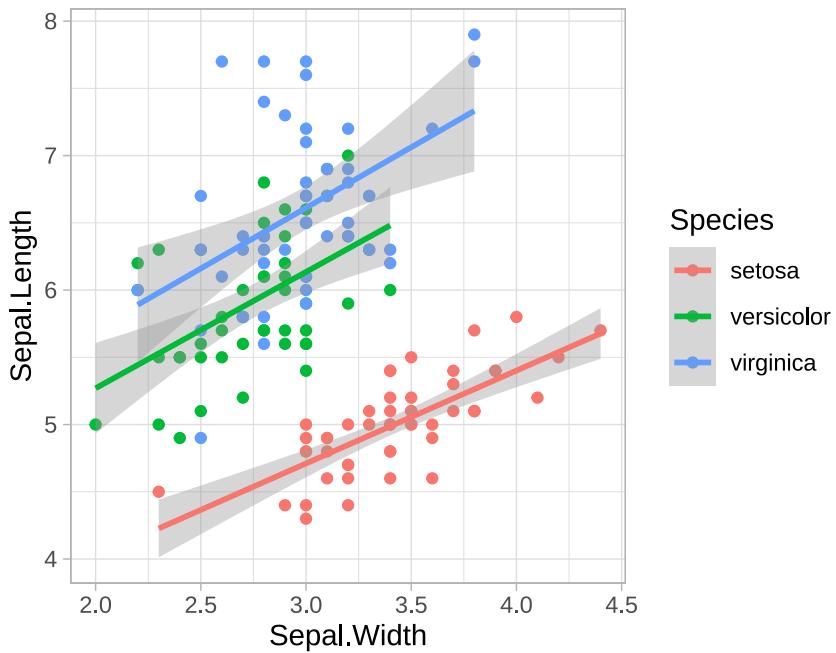
```
p1 + labs(x = "Kelchblattbreite [mm]", y = "Kelchblattlänge [mm]")
```



1146

- ¹¹⁴⁷ Oder auch den ganzen Plot anpassen. Dafür gibt es *themes*. Es gibt eine Reihe von vorgefertigten *themes* oder man kann diese auch selber erstellen (das ist aber nicht Teil dieses Kurses).

```
p1 + theme_light()
```

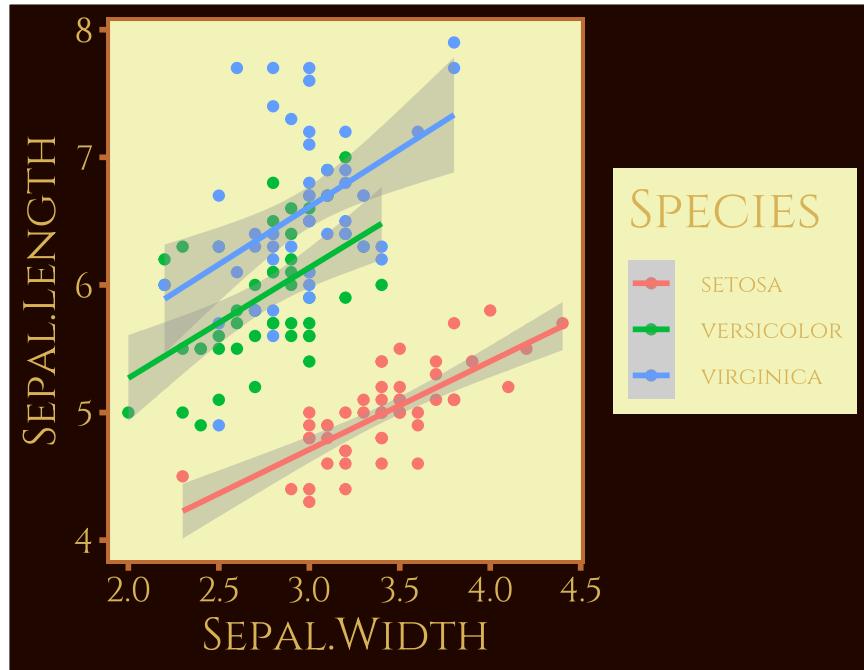


1149

- ¹¹⁵⁰ 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

1152 ggthemes hauptsächlich durchdachte Grafikkonzepte liefert, sind die Themes aus ThemePark eher Popkultur
1153 und nicht 100 %ig ernst gemeint.

```
p1 + theme_gamethrones()
```

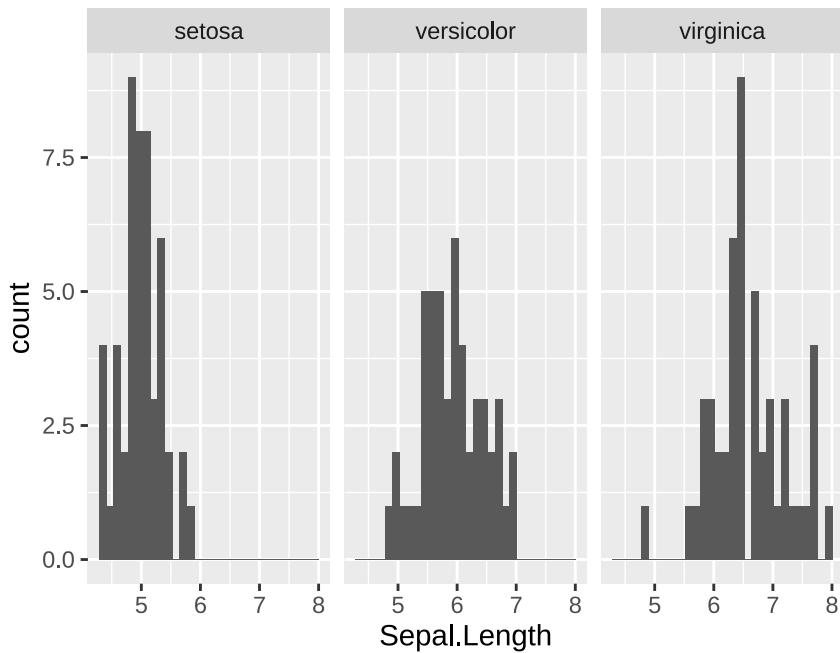


1154

1155 8.4.1 Multipanel Abbildungen

1156 Mit *ggplot2* kann man einfach Abbildungen erstellen, die mehrere Panels haben. Das bedeutet, dass eine
1157 oder mehrere weitere Variablen gibt, die einen Plot in mehrere Subplots teilt. Dafür gibt es zwei Funktion:
1158 `facet_grid()` und `facet_wrap()`.

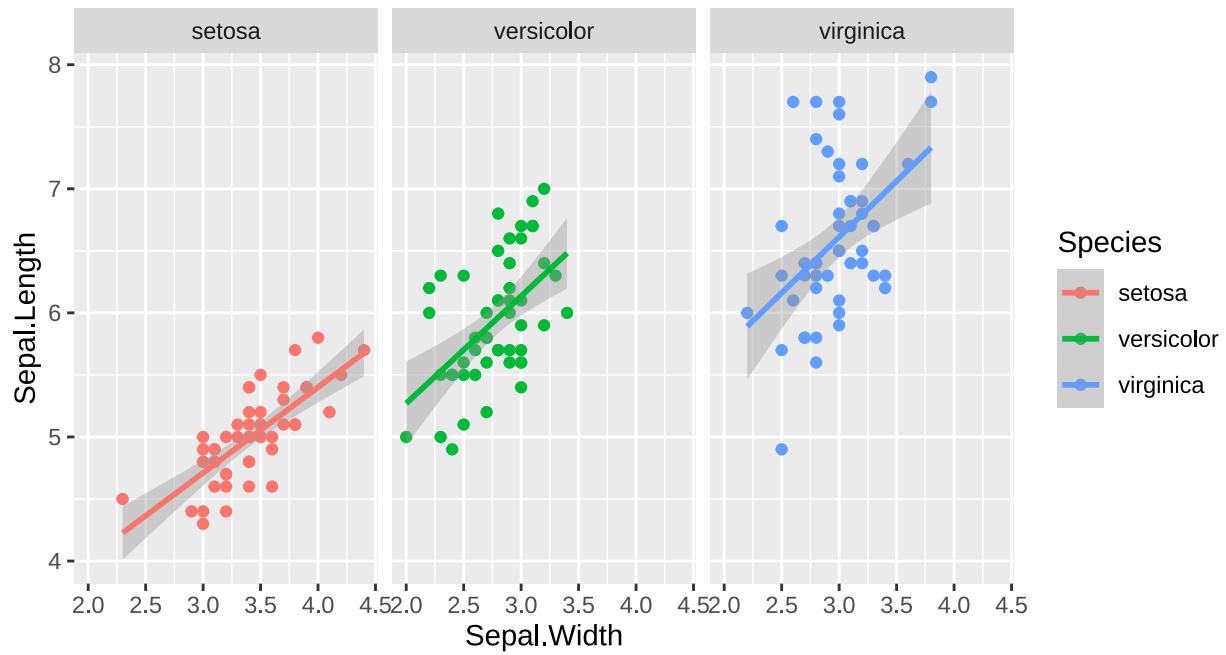
```
ggplot(iris, aes(Sepal.Length)) + geom_histogram() +  
  facet_grid(~ Species)
```



1159

1160 Die Funktion `facet_grid()` erzeugt einen *Grid* und beschriftet die Grid-Zeilen und -Spalten, während
 1161 `facet_wrap()` für jedes Panel (Diagramm) eine eigene Überschrift erzeugt. Beim Arrangieren der Diagram-
 1162 me wird der Vorteil von `ggplot2` gegenüber `base plot` besonders deutlich. Während es im `base plot` System
 1163 mühsam ist, die Abbildungen zu arrangieren, übernimmt `ggplot2` das Arrangement automatisch und fügt
 1164 sogar eine automatisch Legende hinzu. Die Achsenabschnitte werden so angepasst, dass die Daten vergleich-
 1165 bar sind.

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

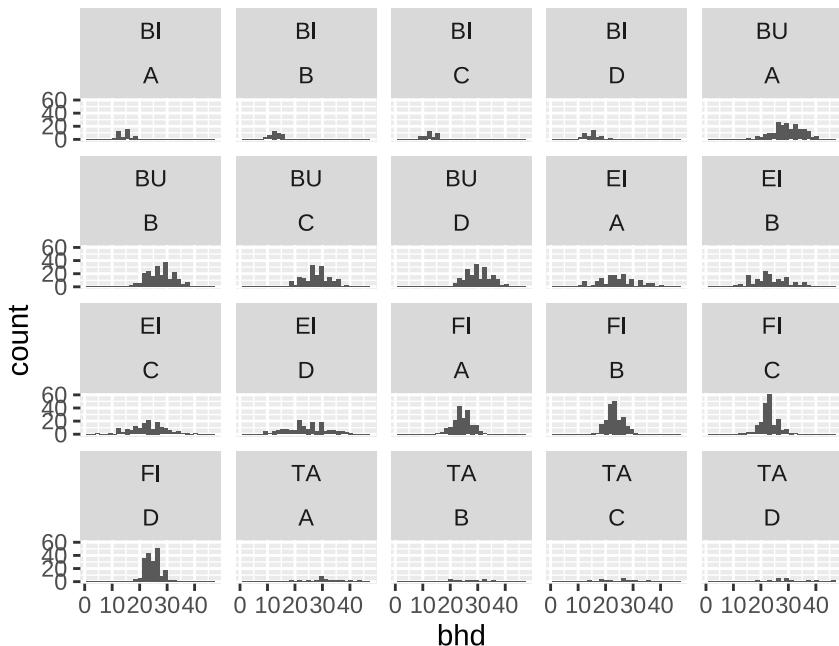


1166

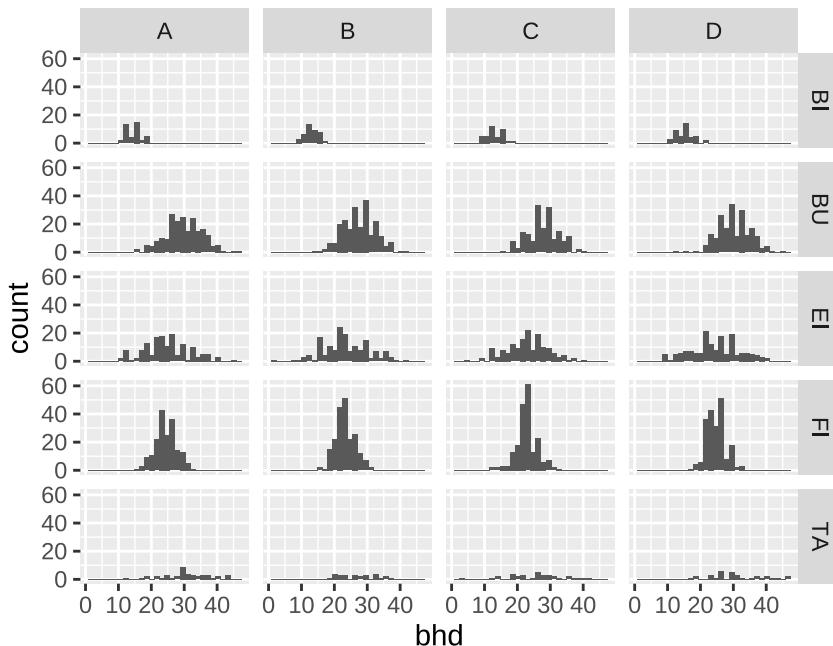
1167

1168 Aufgabe 22: Multipanel Abbildungen 1169

- 1170 Lesen Sie erneut den Datensatz daten/bhd_1.txt ein und speichern Sie diesen in die Variable (`dat_bhd`).
 1171 Erstellen Sie für jede Art und Gebiet ein Histogramm. Welche Unterschiede können Sie feststellen, wenn Sie
 1172 `facet_grid()` oder `facet_wrap()` verwenden?



1173



1174

1175 8.4.2 Plots kombinieren

1176 Es gibt Situationen, in denen **unterschiedliche** Plots miteinander kombiniert werden müssen. Im vorherigen
 1177 Abschnitt wurde dies immer anhand einer gruppierenden Kovariate gemacht. Aber es gibt auch Situations-
 1178 en, in denen das nicht möglich ist. Beispielsweise wenn ein Histogramm und ein Scatterplot vom gleichen
 1179 Datensatz zusammengefasst werden sollen. Dafür bietet sich das Paket **patchwork** an⁹.

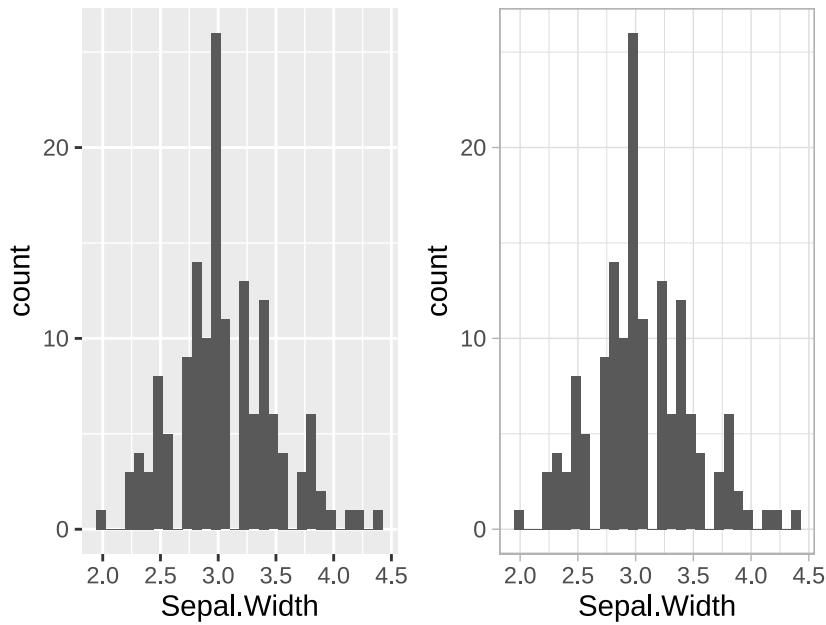
1180 Als erstes können wir zwei (oder natürlich auch mehrere Plots) erstellen. Hier unterscheiden sich die Plots
 1181 lediglich durch das Aussehen.

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

1182 Dann müssen können wir diese Plots ebenfalls mit `+` zusammenfügen.

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

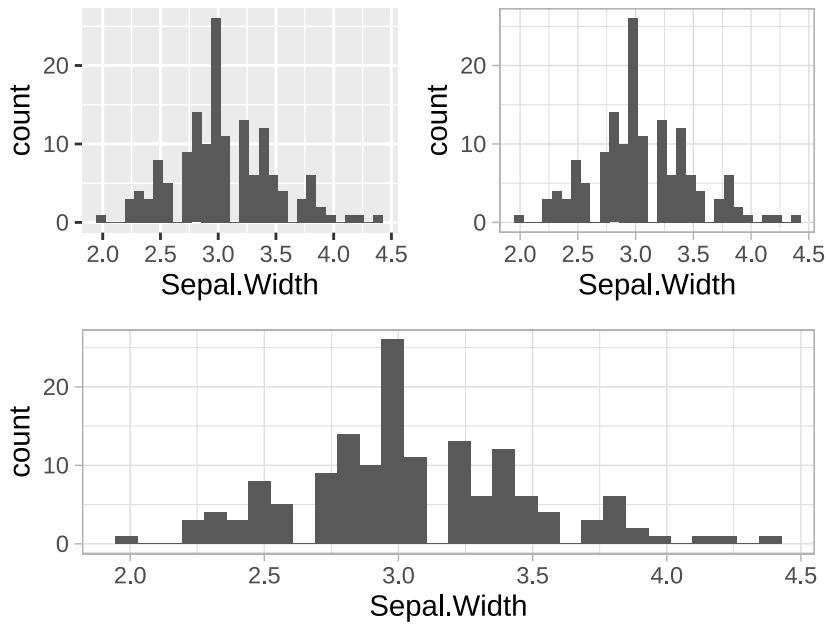
⁹Auch dieses Paket müssen Sie einmalig mit `install.packages("patchwork")` installieren.



1183

¹¹⁸⁴ Natürlich können auch weitere Plots hinzugefügt werden (auch in unterschiedlichen Dimensionen):

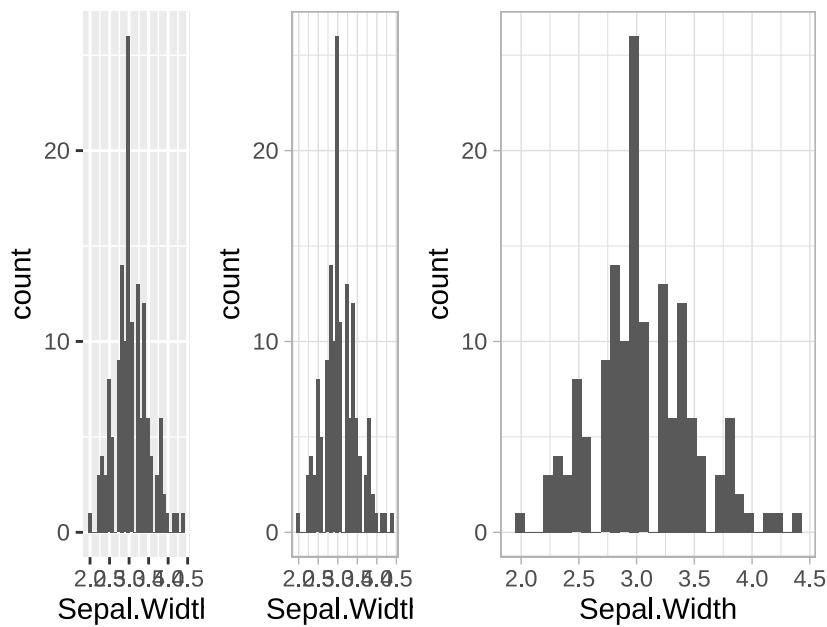
(p1 + p2) / p2



1185

¹¹⁸⁶ Des weiteren können mit | auch Plots gegenüber gestellt werden.

(p1 + p2) | p2



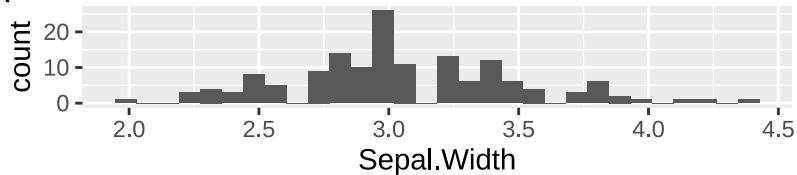
1187

1188 Weitere Optionen können mit `plot_layout()` und `plot_annotation()` angepasst werden. Mit
 1189 `plot_layout()` kann die Anordnung der Plots bestimmt werden (z.B. über die Argumente `nrow` und
 1190 `ncol`), sowie deren *relative* Größe (über die Argumente `widths` und `heights`). Mit der Funktion
 1191 `plot_annotation()` können zusätzliche Beschriftungen hinzugefügt werden, wie beispielsweise eine Titel
 1192 (Argument `title`) oder ein Buchstabe/Zahl für jedes Element (Argument `tag_levels`).

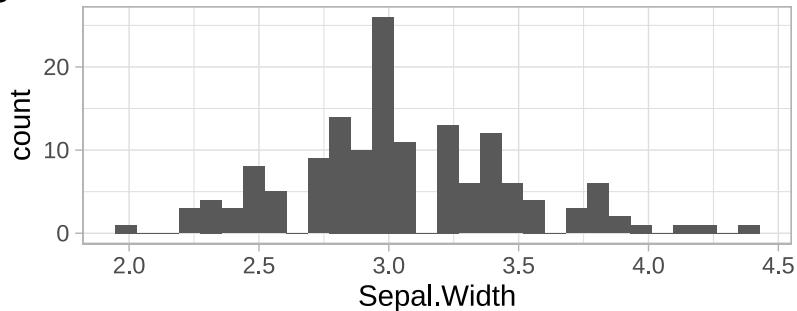
```
p1 + p2 +
  plot_layout(ncol = 1, heights = c(0.3, 0.7)) +
  plot_annotation(title = "Zwei Histogramme", tag_levels = "A")
```

Zwei Histogramme

A



B



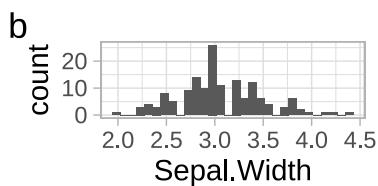
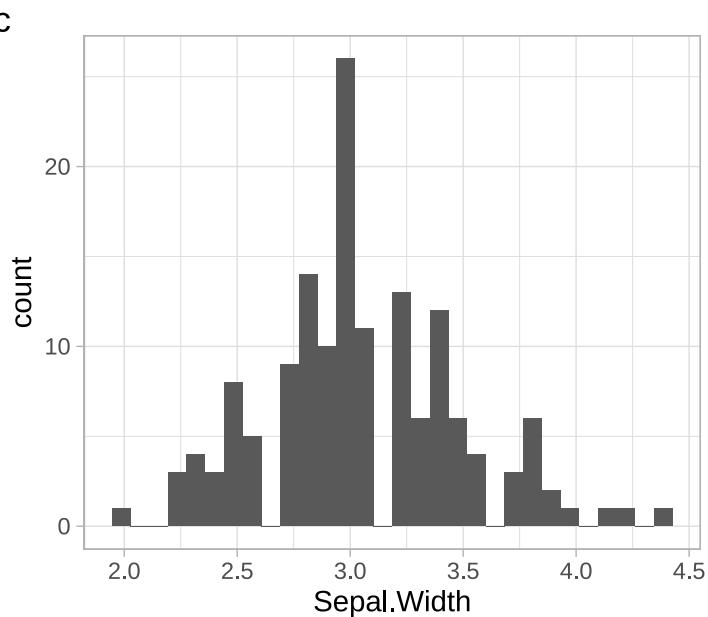
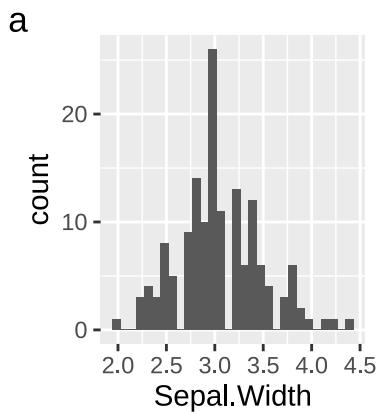
1193

1194

1195 Aufgabe 23: Mehrere Plots zusammenfügen

1196

Versuchen Sie die folgende Zusammenstellung der Plots nachzumachen:



1198

1199 8.4.3 Speichern von plots

1200 Sie können mit `ggsave()` eine zwischen-gespeicherte Abbildung exportieren, indem Sie den Variablenamen
1201 übergeben. Wenn Sie keine Variable übergeben, wird automatisch die letzte Abbildung gespeichert. Das
1202 Dateiformat wird aus dem Dateinamen übernommen. Die Größe der Inhalte können Sie sehr leicht mit den
1203 Argumenten `width` und `height` oder `scale` gesteuert werden.

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

1204 9 Mit Daten arbeiten

1205 9.1 dplyr eine Einführung

1206 `dplyr` ist eine Erweiterung von R (ein Paket), die das Ziel hat, den Umgang mit Daten einfacher und
1207 schneller zu machen.

1208 `dplyr` definiert 5 Verben, um mit Daten zu arbeiten. Diese sind:

- 1209 • `filter`
- 1210 • `select`
- 1211 • `arragne`
- 1212 • `mutate`
- 1213 • `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))
```

1214 Damit die Funktionen aus `dplyr` verwendet werden können, müssen wir als erstes das Paket `dplyr` laden.
1215 The online Resource [R for Data Science](#) gibt einen sehr guten, tieferen Eindruck in die Arbeit mit `dplyr`
1216 und erklärt auch die `dplyr` Philosophie noch Mal sehr genau für Beginner.

```
library(dplyr)
```

1217 Sollte dies zu einer Fehlermeldung führen, dann müssen Sie das Paket `dplyr` erst installieren. Dafür müssen
1218 Sie `einmalig install.packages("dplyr")` installieren.

1219 `dplyr` stellt unterschiedliche Funktionen zum Arbeiten mit Daten zur Verfügung. Es gibt fünf Grundfunktionen
1220 für die am häufigsten vorkommenden Operationen. Mit der Funktion `filter()` können unterschiedliche
1221 Beobachtungen gefiltert werden:

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

```
1222 ##   id plot bhd alter
1223 ## 1   1    1  50   10
1224 ## 2   2    1  29   30
1225 ## 3   3    2  13   31
1226 ## 4   4    2  23   24
1227 ## 5   5    3  25   25
```

1228 Es können auch mehrere Spalten verwendet werden.

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

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

1234 Natürlich kann genau das gleiche Ergebnis mit dem ‘normalen’ R erreicht werden, dies wäre dann:

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

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

1240 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)
```

```
1242 ##   bhd
1243 ## 1   50
1244 ## 2   29
1245 ## 3   13
1246 ## 4   23
1247 ## 5   25
```

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

```
1248 ##   bhd id
1249 ## 1   50  1
1250 ## 2   29  2
1251 ## 3   13  3
1252 ## 4   23  4
1253 ## 5   25  5
```

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

```

1254 ##   BHD id
1255 ## 1 50 1
1256 ## 2 29 2
1257 ## 3 13 3
1258 ## 4 23 4
1259 ## 5 25 5

```

1260 Mit der Funktion `arrange()` können die Beobachtungen in einem `data.frame` sortiert werden.

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

```

1261 ##   id plot bhd alter
1262 ## 1 3 2 13 31
1263 ## 2 4 2 23 24
1264 ## 3 5 3 25 25
1265 ## 4 2 1 29 30
1266 ## 5 1 1 50 10

```

1267 Mit der Funktion `desc()` kann die Anordnung in absteigender Reihenfolge sortiert werden.

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

```

1268 ##   id plot bhd alter
1269 ## 1 1 1 50 10
1270 ## 2 2 1 29 30
1271 ## 3 5 3 25 25
1272 ## 4 4 2 23 24
1273 ## 5 3 2 13 31

```

1274 Mit der Funktion `mutate()` kann man eine neue Spalte hinzufügen.

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

```

1275 ##   id plot bhd alter bhd_mm          fl
1276 ## 1 1 1 50 10 500 1963.4954
1277 ## 2 2 1 29 30 290 660.5199
1278 ## 3 3 2 13 31 130 132.7323
1279 ## 4 4 2 23 24 230 415.4756
1280 ## 5 5 3 25 25 250 490.8739

```

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

```

1281 ##   id plot bhd alter mean_bhd
1282 ## 1   1     1   50    10    28
1283 ## 2   2     1   29    30    28
1284 ## 3   3     2   13    31    28
1285 ## 4   4     2   23    24    28
1286 ## 5   5     3   25    25    28

```

1287 Mit der Funktion `summarise()` können Spalten zusammengefasst werden.

```

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

1288 ##   mean_bhd  mean_sd
1289 ## 1         28 13.63818

```

1290

1291 Aufgabe 24: Datenmanipulation mit dplyr

- 1293 1. Laden Sie den Datensatz `data/bhd_1.txt`
 1294 2. Berechnen Sie folgende Werte für alle Einträge und speichern Sie die Ergebnisse in `erg1`

- 1295 • mittlerer `bhd`
 1296 • maximales `alter`
 1297 • die Standardabweichung des BHDs
 1298 • die Anzahl Bäume mit einem BHD > 30

1299 9.2 Arbeiten mit gruppierten Daten

1300 Zusätzlich können `mutate` und `summarise` auch auf gruppierte Daten angewendet werden. Dafür müssen
 1301 wir erst die Funktion `group_by()` aufrufen und als Argumente die Variablen übergeben, die die Gruppen
 1302 definieren.

```

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

```

```

1303 ##   id plot bhd alter bhd_m
1304 ## 1   1     1   50    10    28
1305 ## 2   2     1   29    30    28

```

```
1306 ## 3 3 2 13 31 28
1307 ## 4 4 2 23 24 28
1308 ## 5 5 3 25 25 28
```

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

```
1309 ## # A tibble: 5 x 5
1310 ## # Groups: plot [3]
1311 ##   id   plot   bhd alter bhd_m
1312 ##   <int> <dbl> <dbl> <dbl> <dbl>
1313 ## 1     1     1    50    10  39.5
1314 ## 2     2     1    29    30  39.5
1315 ## 3     3     2    13    31  18
1316 ## 4     4     2    23    24  18
1317 ## 5     5     3    25    25  25
```

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

```
1318 ## bhd_m
1319 ## 1 28
```

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

```
1320 ## # A tibble: 3 x 2
1321 ##   plot bhd_m
1322 ##   <dbl> <dbl>
1323 ## 1     1  39.5
1324 ## 2     2  18
1325 ## 3     3  25
```

```
1326
```

Aufgabe 25: dplyr mit gruppierten Daten

- 1329 1. Laden Sie den Datensatz `data/bhd_1.txt`
 1330 2. Berechnen Sie für jede Baumart und Standort die folgenden Werte:

- 1331 • mittlerer `bhd`
 1332 • maximales `alter`
 1333 • die Standardabweichung des BHDS
 1334 • die Anzahl Bäume mit einem BHD > 30

- 1335 3. Ordnen Sie das Ergebnis nach Baumart und BHD.

1336 9.3 pipes oder %>%

1337 Mit *Pipes* (%>%) kann man das Ergebnis einer Funktion einfach an eine nachfolgende Funktion weiterreichen.

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

1338 Wir kennen bis jetzt:

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

1339 ## [1] 3.333333

1340 Mit *Pipes*, die durch das Symbol %>% dargestellt werden¹⁰, können wir das etwas vereinfachen und nacheinander schreiben:

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

1342 ## [1] 3.333333

1343 Oder sogar

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

1344 ## [1] 3.333333

1345

1346 Aufgabe 26: Pipes %>%

1348 Wiederholen Sie die letzte Aufgabe, aber diesmal ohne Zwischenergebnisse zu speichern:

1349 1. Laden Sie den Datensatz data/bhd_1.txt.

1350 2. Berechnen Sie für jede Baumart und Standort die folgenden Werte:

- 1351 • mittlerer bhd
- 1352 • maximales alter
- 1353 • die Standardabweichung des BHDs
- 1354 • die Anzahl Bäume mit einem BHD > 30

1355 3. Ordnen Sie das Ergebnis nach Baumart und BHD.

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

1356 9.4 Joins

1357 Eine weitere häufige Aufgabe beim Daten Management ist es Daten zusammenzuführen. Nehmen Sie an,
 1358 dass wir folgende Aufnahmen gemacht haben

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

1359 und jeder Baum lediglich mit einer `id` versehen wurde. In einer zweiten Tabelle wurden dann weitere Daten
 1360 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")
)
```

1361 Ziel ist es jetzt die Bäume aus `aufnahmen` mit den Informationen aus den `metadaten` zu verbinden. Dazu
 1362 dient `id` als Bindeglied (oft auch Schlüssel genannt).

1363 Dazu gibt es vier Möglichkeiten.

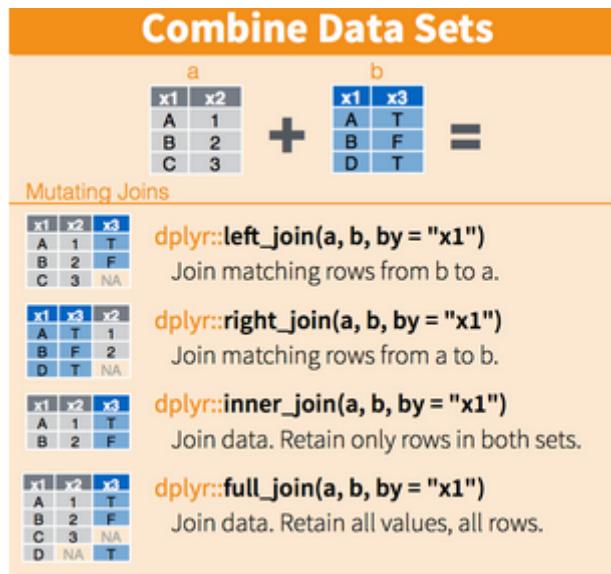


Abbildung 8: Joins (Quelle Rstudio)

1364 Zur Durchführung gibt es in base R die Funktion `merge()`. Wir werden aber gleich die Funktionen aus dem
 1365 Paket `dplyr` verwenden.

```

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

1366 ##   id bhd  art gebiet
1367 ## 1  1  20 <NA>    <NA>
1368 ## 2  2  31   Ta      A
1369 ## 3  3  74   Bu      B

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

1370 ##   id bhd art gebiet
1371 ## 1   2  31  Ta      A
1372 ## 2   3  74  Bu      B
1373 ## 3   4  NA  Bu      B

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

1374 ##   id bhd art gebiet
1375 ## 1   2  31  Ta      A
1376 ## 2   3  74  Bu      B

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

1377 ##   id bhd  art gebiet
1378 ## 1   1  20 <NA>    <NA>
1379 ## 2   2  31   Ta      A
1380 ## 3   3  74   Bu      B
1381 ## 4   4  NA  Bu      B

1382 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"))

1383 ##   id bhd  art gebiet
1384 ## 1   1  20 <NA>    <NA>
1385 ## 2   2  31   Ta      A
1386 ## 3   3  74   Bu      B

```

1387

1388 **Aufgabe 27: Verbinden von Daten**

1389

- 1390 • Lesen Sie die Datensätze `daten/bhd_2.txt` und `daten/bhd_2_meta.txt` ein.
1391 • Stellen Sie sicher, dass es für den `bhd` keine fehlenden Werte gibt (entfernen sie entsprechende Zeilen)
1392 • Fügen Sie zu den Metadaten (gespeichert in `bhd_2_meta`) die Anzahl Bäume und den mittleren `bhd`
1393 hinzu pro Gebiet.

1394 **9.5 ‘long’ and ‘wide’ Datenformate**

1395 Unter anderem Wickham (2014) empfiehlt das Prinzip von *tidy* Data. Nach diesem Prinzip sollten Daten wie
1396 folgt organisiert sein:

- 1397 • Jede Zeile ist ein Merkmalsträger/Subjekt/Objekt (z. B. eine Person oder ein Baum).
1398 • Jede Spalte ist eine Variable (ein Merkmal), die den Merkmalsträger beschreibt.
1399 • In jeder Zelle ist genau ein Wert (Merkmalausprägung), nämlich der Wert, der Variable für den Merk-
1400 malsträger.

1401 Zum Beispiel enthalten Spaltennamen oft Informationen, die eigentlich in einer Variable gespeichert werden
1402 sollten. Folgendes Beispiel gibt die BHD Messung von 3 Bäumen in 3 Jahren wieder. Der Vorteil von *tidy*
1403 Daten ist, dass viele Funktionen diese Form erwarten. Somit müssen sie Ihre Daten nur ein Mal organisieren
1404 und können fast alle Analysen durchführen.

```
dat <- tibble(  
  id = 1:3,  
  bhd2015 = c(30, 31, 32),  
  bhd2026 = c(31, 31, 33),  
  bhd2017 = c(34, 32, 33)  
)
```

1405 Der `tibble` aus dem Paket `tibble` ist ein moderner Data Frame, der sich in seinen Funktionen in das
1406 `tidy` Universum einfügt. Wir können auch einfach das Paket `tidyverse` laden, um alle Pakete, die Teil des
1407 `tidy` Universums sind gleichzeitig zu laden. Der `tibble` beinhaltet alle Funktionen, die der Data Frame
1408 auch beinhaltet und kann auch für genau die gleichen Zwecke verwendet werden. Der `tibble` hat u. A. eine
1409 modernere Darstellung im Konsolenoutput.

1410 Diese Daten sind jetzt im `wide`-Format gespeichert und folglich nicht `tidy`, weil Information über die Daten
1411 (nämlich das Jahr der Aufnahme in den Spaltennamen gespeichert sind). Besser wäre eine Struktur mit
1412 nur drei Spalten: `id`, `jahr` und `bhd`. Um die Daten in so eine Struktur zu bringen, gibt es die Funktion
1413 `pivot_longer()` aus dem Paket `tidyverse`, die ebenfalls Teil des `tidyverse` ist.

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

```
1414 ## # A tibble: 9 x 3
1415 ##      id name    value
1416 ##   <int> <chr>   <dbl>
1417 ## 1     1 bhd2015     30
1418 ## 2     1 bhd2026     31
1419 ## 3     1 bhd2017     34
1420 ## 4     2 bhd2015     31
1421 ## 5     2 bhd2026     31
1422 ## 6     2 bhd2017     32
1423 ## 7     3 bhd2015     32
1424 ## 8     3 bhd2026     33
1425 ## 9     3 bhd2017     33
```

1426 Wenn wir die Spalten für die Variable und den Wert gleich sinnvoll benennen möchten, können wir das über
 1427 die Argumente `names_to` und `values_to` machen.

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

```
1428 ## # A tibble: 9 x 3
1429 ##      id jahr    bhd
1430 ##   <int> <chr>   <dbl>
1431 ## 1     1 bhd2015     30
1432 ## 2     1 bhd2026     31
1433 ## 3     1 bhd2017     34
1434 ## 4     2 bhd2015     31
1435 ## 5     2 bhd2026     31
1436 ## 6     2 bhd2017     32
1437 ## 7     3 bhd2015     32
1438 ## 8     3 bhd2026     33
1439 ## 9     3 bhd2017     33
```

1440 Analog zu der Funktion `pivot_longer()` gibt es auch die Funktion `pivot_wider()`, um vom Daten vom
 1441 long-Format ins wide-Format zu transformieren.

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

```
1442 ## # A tibble: 3 x 4
1443 ##      id bhd2015 bhd2026 bhd2017
```

```

1444 ## <int> <dbl> <dbl> <dbl>
1445 ## 1     1     30    31    34
1446 ## 2     2     31    31    32
1447 ## 3     3     32    33    33

```

1448

Aufgabe 28: Zeitliche Verlauf von BHDs

1451 In der Datei `bhd_3.csv` befinden sich gemessene BHDs (in cm) von unterschiedlichen Bäumen zu unter-
 1452 schiedlichen Zeitpunkten. Erstellen Sie ein Liniendiagramm, das den zeitlichen (x-Achse) Verlauf der BHDs
 1453 (y-Achse) für die unterschiedlichen Bäume darstellt.

1454 **9.6 Auswählen von Variablen**

1455 Sobald die Datensätze etwas umfangreicher werden (d. h., es gibt mehrere Spalten in einem `data.frame`),
 1456 können innerhalb vieler `dplyr`-Funktionen spezielle Funktionen verwendet werden, um Variablen auszuwäh-
 1457 len.

1458 Wenn die genaue Position der Spalten bekannt ist, kann man mit dem `:-Operator` und der Position Spalten
 1459 auswählen:

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

```

1460 ## Sepal.Length Sepal.Width Petal.Length
1461 ## 1          5.1      3.5      1.4
1462 ## 2          4.9      3.0      1.4
1463 ## 3          4.7      3.2      1.3

```

1464 Diese Vorgehensweise kann gehährlich sein, da sich manchmal Spalten verschieben und sich somit die Posi-
 1465 tionen ändern. Es besser Spalten immer explizit mit Namen statt mit Nummern anzusprechen.

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

```

1466 ## Sepal.Length Sepal.Width Petal.Length
1467 ## 1          5.1      3.5      1.4
1468 ## 2          4.9      3.0      1.4
1469 ## 3          4.7      3.2      1.3

```

1470 `select()` erlaubt es, auch hier den `:-Operator` zu verwenden:

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

```
1471 ## Sepal.Length Sepal.Width Petal.Length
1472 ## 1          5.1        3.5       1.4
1473 ## 2          4.9        3.0       1.4
1474 ## 3          4.7        3.2       1.3
```

1475 Es gibt auch einige spezielle Funktionen, um Spalten innerhalb eines `select()`-Aufrufs auszuführen:

- 1476 • `starts_with()`: Hier kann man ein Muster angeben, mit dem ein Text anfangen muss.
- 1477 • `ends_with()`: Diese Funktion ist analog zu `starts_with()`, jetzt wird aber am Ende des Spaltennamens nach dem Muster gesucht.
- 1479 • `contains()`: Hier kann ein Muster übergeben werden, das irgendwo im Spaltennamen sein muss.
- 1480 • `everything()`: Mit dieser Funktion werden alle Spalten ausgewählt.
- 1481 • `last_col()`: Mit dieser Funktion wird nur die letzte Spalte ausgewählt (dass ist die Spalte, die ganz rechts ist).

1483 Sämtliche Auswahlen können mit - umgekehrt werden.

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

```
1484 ## Sepal.Length Sepal.Width
1485 ## 1          5.1        3.5
1486 ## 2          4.9        3.0
1487 ## 3          4.7        3.2
```

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

```
1488 ## Petal.Length Petal.Width Species
1489 ## 1          1.4        0.2   setosa
1490 ## 2          1.4        0.2   setosa
1491 ## 3          1.3        0.2   setosa
```

1492 `select()` bietet auch noch die Möglichkeit, Spaltennamen zu ändern.

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

```
1493 ## sep_width
1494 ## 1          3.5
1495 ## 2          3.0
1496 ## 3          3.2
```

1497

1498 **Aufgabe 29: Auswählen von Spalten**

1499

1500 In der Datei `messungen_1.csv` sind Messungen von zwei Sensoren enthalten für die ersten vier Monate eines
 1501 Jahres. Führen Sie folgende Abfragen durch:

- 1502 1. Wählen Sie alle Messungen für Januar aus.
 1503 2. Wählen Sie alle Messungen für Januar und März aus.

1504 **9.7 Einzelne Beobachtungen abfragen (`slice()`)**

1505 Mit dem Befehl `slice()` kann man einzelne Beobachtungen (= Zeilen) aus einem `data.frame` abfragen.

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

1506 ## Sepal.Length Sepal.Width Petal.Length Petal.Width Species
 1507 ## 1 5.1 3.5 1.4 0.2 setosa
 1508 ## 2 4.4 2.9 1.4 0.2 setosa
 1509 ## 3 5.1 3.5 1.4 0.3 setosa

1510 Davon gibt es drei nützliche Varianten: 1) `slice_head()` und `slice_tail()`; 2) `slice_max()` und
 1511 `slice_min()`; 3) `slice_random()`.

1512 `slice_head()` und `slice_tail()` sind analog zu `head()` und `tail()`, aber mit dem entscheidenden Unter-
 1513 schied, dass Gruppierungen berücksichtigt werden. Wenn keine Gruppierung in den Daten vorhanden ist,
 1514 gibt es keinen Unterschied.

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

1515 ## Sepal.Length Sepal.Width Petal.Length Petal.Width Species
 1516 ## 1 5.1 3.5 1.4 0.2 setosa
 1517 ## 2 4.9 3.0 1.4 0.2 setosa

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

1518 ## Sepal.Length Sepal.Width Petal.Length Petal.Width Species
 1519 ## 1 5.1 3.5 1.4 0.2 setosa
 1520 ## 2 4.9 3.0 1.4 0.2 setosa

1521 Sobald jedoch eine gruppierende Variable eingeführt wird, gibt `slice_head()` die ersten `n` Beobachtungen
 1522 für jede Gruppe zurück und `head()` für den gesamten Datensatz.

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

1523 ## # A tibble: 2 x 5
1524 ## # Groups:   Species [1]
1525 ##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
1526 ##     <dbl>       <dbl>       <dbl>       <dbl> <fct>
1527 ## 1     5.1         3.5         1.4        0.2  setosa
1528 ## 2     4.9         3           1.4        0.2  setosa
```

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

```
1529 ## # A tibble: 6 x 5
1530 ## # Groups:   Species [3]
1531 ##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
1532 ##     <dbl>       <dbl>       <dbl>       <dbl> <fct>
1533 ## 1     5.1         3.5         1.4        0.2  setosa
1534 ## 2     4.9         3           1.4        0.2  setosa
1535 ## 3     7           3.2         4.7        1.4  versicolor
1536 ## 4     6.4         3.2         4.5        1.5  versicolor
1537 ## 5     6.3         3.3         6          2.5  virginica
1538 ## 6     5.8         2.7         5.1        1.9  virginica
```

1539 `slice_tail()` funktioniert analog zu `slice_head()` mit dem einzigen Unterschied, dass nicht die ersten `n` Zeilen zurück gegeben werden sondern die letzten `n` Zeilen.

1541 `slice_max()` und `slice_min()` geben die Beobachtung mit dem maximalen bzw. minimalen Wert einer 1542 Variablen zurück. Auch hier werden Gruppen berücksichtigt.

```
iris %>% slice_max(Sepal.Length)
```

```
1543 ##   Sepal.Length Sepal.Width Petal.Length Petal.Width   Species
1544 ## 1     7.9         3.8         6.4        2  virginica
```

1545 Und mit Gruppen:

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

```
1546 ## # A tibble: 3 x 5
```

```

1547 ## # Groups: Species [3]
1548 ## Sepal.Length Sepal.Width Petal.Length Petal.Width Species
1549 ## <dbl>     <dbl>     <dbl>     <dbl> <fct>
1550 ## 1       5.8       4        1.2      0.2 setosa
1551 ## 2       7         3.2      4.7      1.4 versicolor
1552 ## 3       7.9      3.8      6.4      2    virginica

```

1553 `slice_min()` funktioniert genau gleich, nur dass die Beobachtung (=Zeile) mit dem kleinsten Wert einer
1554 Variable zurück gegeben wird.

1555 Die Funktion `slice_sample()` erlaubt es zufällige Beobachtungen zu ziehen. Dabei kann über das Argument
1556 `n` die Anzahl an Beobachtungen angegeben werden oder über das Argument `prop` der Anteil an Beobach-
1557 tungen.

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

```

1558 ## Sepal.Length Sepal.Width Petal.Length Petal.Width Species
1559 ## 1       6.5       2.8      4.6      1.5 versicolor
1560 ## 2       6.3       3.3      4.7      1.6 versicolor
1561 ## 3       7.2       3.2      6.0      1.8 virginica
1562 ## 4       4.9       3.6      1.4      0.1 setosa
1563 ## 5       6.0       2.7      5.1      1.6 versicolor

```

1564 Das Ergebnis ist bei jedem von Ihnen anders, da es sich um eine zufällige Ziehung handelt. Wenn Sie diese
1565 Ergebnisse wiederholen möchte, können Sie über `set.seed()` die zufällige Ziehung reproduzierbar machen.

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

```

1566 ## Sepal.Length Sepal.Width Petal.Length Petal.Width Species
1567 ## 1       4.3       3.0      1.1      0.1 setosa
1568 ## 2       5.0       3.3      1.4      0.2 setosa
1569 ## 3       7.7       3.8      6.7      2.2 virginica
1570 ## 4       4.4       3.2      1.3      0.2 setosa
1571 ## 5       5.9       3.0      5.1      1.8 virginica

```

1572 Wenn beispielsweise 5% der Beobachtungen gezogen werden sollen, kann dies so gemacht werden:

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

```

1573 ## Sepal.Length Sepal.Width Petal.Length Petal.Width Species
1574 ## 1       7.7       3.8      6.7      2.2 virginica
1575 ## 2       5.5       2.5      4.0      1.3 versicolor
1576 ## 3       5.5       2.6      4.4      1.2 versicolor

```

```

1577 ## 4      6.5      3.0      5.2      2.0  virginica
1578 ## 5      6.1      3.0      4.6      1.4  versicolor
1579 ## 6      6.3      3.4      5.6      2.4  virginica
1580 ## 7      5.1      2.5      3.0      1.1  versicolor

```

1581 `slice_sample()` berücksichtigt ebenfalls Gruppen. Mit den Argumenten `replace` und `weight_by` dann die
 1582 Zufallsziehung genauer spezifiziert werden. `replace` sagt, ob eine gezogenen Beobachtung wieder zurück ge-
 1583 legt wird oder nicht. Mit dem Argument `weight_by` können optional gewichte für jede Beobachtung vergeben
 1584 werden.

1585

1586 Aufgabe 30: Daten beschreiben

1588 Verwenden Sie den Datensatz `bhd_1.txt` und finden Sie für jedes Gebiet und Art die Beobachtung mit
 1589 kleinsten BHD.

1590 9.8 Spalten trennen

1591 Ein gut geplanter Datensatz besteht aus Beobachtungen (Zeilen), Variablen (Spalten) und in jeder Zelle ist
 1592 immer ein *genau* ein Wert gespeichert. Leider gibt es oft Datensätze, bei denen mehr als ein Wert pro Zelle
 1593 gespeichert wurde. Die Funktion `separate()` kann helfen solche Daten zu trennen.
 1594 Wir verwenden einen erfunden Datensatz zu Beobachtungen von Tieren und einer geschätzten Distanz zu
 1595 diesen Tieren.

```

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

```

1596 In der Spalte `beobachtung` sind zwei Informationen gespeichert: Die Distanz zur Beobachtung und die Art.
 1597 Das ist ungünstig, weil wir so weder nach Tierart noch nach distanz filtern können (nicht *tidy*). Mit der
 1598 Funktion `separate()`, können wir Beobachtungen einer Spalte in mehrere Spalten trennen. Dafür muss der
 1599 Spaltennamen (Argument `col`), die neuen Spaltennamen (Argument `into`) und das Trennzeichen (Argument
 1600 `sep`) angegeben werden.

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

```

1601 ## # A tibble: 4 x 3
1602 ##       id Distanz Art
1603 ##     <int> <chr>   <chr>
1604 ## 1      1 10m     " Reh"

```

```
1605 ## 2      2 100m    " Reh"  
1606 ## 3      3 20m     " Fuchs"  
1607 ## 4      4 40      "Reh"
```

1608 Nach dem Aufruf von `seperate()` gibt es zwei neue Spalten (`Distanz` und `Art`), die die alte Spalte
1609 `beobachtung` ersetzen.

1610

1611 **Aufgabe 31: Aufräumen**
1612

1613 Verwenden Sie den folgenden Datensatz und bringen Sie ihn in eine Form, die sicherstellt dass

- 1614 • jede Zelle genau einen Wert enthält.
1615 • jede Zeile eine Beobachtung ist.
1616 • 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"))  
)
```

1617 10 Arbeiten mit Text

1618 Bis jetzt haben wir fast ausschließlich mit Zahlen oder Abbildungen gearbeitet. R bietet aber auch viele
 1619 Werkzeuge, um mit Text zu arbeiten. Wir wollen hier ein paar Funktionen dafür vorstellen. Als erstes sollte
 1620 nochmals klargestellt werden, was eigentlich ein Text ist. In R ist alles, das innerhalb von doppelten ("") oder
 1621 einfachen ('') Anführungszeichen geschrieben ist, Text.

1622 Anbei einige Beispiele:

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

1623 Wichtig ist hier zu sehen, dass z nicht als Zahl sondern, als Text interpretiert wird (siehe Datentypen).

```
z + 1
```

1624 ## Error in z + 1: nicht-numerisches Argument für binären Operator

1625 Wenn man sicher ist, dass es sich bei einem Textobjekt um eine Zahl handelt, kann man dies mit der Funktion
 1626 `as.numeric()` in eine Zahl umwandeln.

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

1627 ## [1] 31

1628 Aber mit a führt dies wieder zu einem NA-Wert, da a nicht in eine Zahl umgewandelt werden kann.

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

1629 ## Warning: NAs durch Umwandlung erzeugt

1630 ## [1] NA

1631 10.1 Arbeiten mit Text

1632 Wir wollen erst einmal drei Funktionen besprechen, die es erlauben mit Text zu arbeiten. Die Funktion
 1633 `nchar()`¹¹ gibt an wie viele Zeichen ein Text hat. Also z.B.

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

1634 ## [1] 5

¹¹char ist kurz für character.

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

```
1635 ## [1] 2
```

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

```
1636 ## [1] 20
```

1637 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
```

```
1640 ## [1] "Eva Meier"
```

1641 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
```

```
1643 ## [1] "Eva, Meier"
```

1644 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)
```

```
1646 ## [1] "Hal"
```

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

```
1647 ## [1] "allo"
```

```
1648
```

Aufgabe 32: Arbeiten mit Text 1

1651 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")
```

- 1652 1. Aus wie vielen Buchstaben besteht jedes Wort?
 1653 2. Finden Sie das längste Wort.
 1654 3. Wie viel Prozent der Wörter fangen mit einem S an?
 1655 4. Fügen Sie jedem Wort seine Position im Vektor hinzu. Beispielsweise soll aus Vogel "2. Vogel" werden
 1656 usw.

1657 10.2 Finden von Textmustern

- 1658 Mit der Funktion `grep()` können Muster in einem Text gefunden werden. Wenn wir beispielsweise folgenden
 1659 Vektor mit Textelementen haben.

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

- 1660 Und wir wollen alle Straßennamen die ein `weg` haben abfragen, dann können wir folgenden Befehl ausführen:

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

1661 ## [1] 2

- 1662 Im zweiten Element von `txt` kommen die Zeichen `Weg` vor. Beachte, in der Standardeinstellung wird zwischen
 1663 Groß- und Kleinschreibung unterschieden. Dies kann mit dem Argument `ignore.case = TRUE` angepasst
 1664 werden.

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

1665 ## [1] 1 2

- 1666 Mit der Funktion `sub` können Zeichen innerhalb einer Zeichenkette ersetzt werden.

- 1667 So ersetzt der folgende Ausdruck `ae` mit `ä`.

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

1668 ## [1] "Friedländer Weg"

- 1669 Wenn allerdings das zu ersetzende Zeichen mehr als einmal vorkommt und beide Instanzen ersetzt werden
 1670 sollen braucht man die Funktion `gsub`.

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

1671 ## [1] "Friedländer Weg und Reinhaeuser Landstrasse sind zwei Strassen in Goettingen."

1672 Mit `sub()` wird nur das erste `ae` ersetzt, während `gsub()` alle `ae` mit einem `ä` ersetzt.

```
gsub("ae", "ä", txt)
```

1673 ## [1] "Friedländer Weg und Reinhäuser Landstrasse sind zwei Strassen in Goettingen."

1674 Oft ist der genaue Ausdruck den man finden möchte jedoch Variabel. Beispielsweise möchte man alle Wörter
 1675 mit einem Umlaut oder Zahlen finden möchte, kann man das oft abkürzen. Dafür gibt es reguläre Ausdrücke.
 1676 Wir werden hier nur ein paar beispielhafte Anwendungen besprechen.

1677 Sowohl in den Funktionen `grep()` als auch `(g)sub()` kann mit anstatt dem Muster (immer das erste Argument)
 1678 aus ein regulärer Ausdruck angegeben werden. Mit `[1-9]` sind alle Zahlen von 1 bis 9 gemeint.

1679 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)
```

1680 ## [1] 1 3

1681 Damit lässt sich auch das Problem mit groß- und kleingeschriebenen Wörtern lösen.

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

1682 ## [1] 1 2

1683

1684 1685 Aufgabe 33: Arbeiten mit Text 2

1686 Verwenden Sie den folgenden Vektor:

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

- 1687 1. In wie vielen Wörtern kommt der Doppellaut **au** vor?
1688 2. Ersetzen Sie in allen Wörtern alle **au** mit **_ _**.

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

```
1689 ## [1] 5 6 13
```

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

```
1690 ## [1] "Versicherung" "Methoden"      "Fluss"        "Rudel"       "B_ _m"  
1691 ## [6] "H_ _s"          "Foto"         "Auffahrt"     "Auto"        "Handy"  
1692 ## [11] "Teller"        "Kalender"     "Aufb_ _"
```

1693 11 Arbeiten mit Zeit

1694 Für den Computer bzw. R ist ein Datum erst einmal nichts anderes als ein Text. Für uns ist es sofort klar,
 1695 dass der "13.2.2021" der 13. Februar 2021 ist, für den Computer nicht. Wir müssen R also irgendwie sagen,
 1696 dass die 13 der Tag ist, die 2 der Monat und 2021 das Jahr. Dass der Computer die einzelnen Komponenten
 1697 erkennt, und in einen Datentyp eigens für Zeitformate überführt, nennt man *parsen*¹². Das Arbeiten mit
 1698 Datum und Zeit kann anfangs sehr mühsam sein, aber sobald man einige Grundfertigkeiten erworben
 1699 hat, kann man viele Aufgaben deutlich schneller und effizienter erledigen. Starten Sie am besten gleich mit
 1700 "echten" Zeit-Datentypen und versuchen Sie nicht, sich irgendwie mit Text und numerischen Datentypen
 1701 selbst etwas zu basteln. Der erste Schritt ist immer ein Datum zu *parsen*. Wir verwenden dafür Funktionen
 1702 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)
```

1703 **lubridate** bietet eine Vielzahl von Funktionen zum *parsen* von Datum und Zeit, die sich aus:

- 1704 • y für Jahr,
- 1705 • m für Monat,
- 1706 • d für Tag,
- 1707 • h für Stunde,
- 1708 • m für Minute und
- 1709 • s für Sekunde

1710 zusammen setzten. Alle Funktionen nehmen als erstes Argument ein Textstring. Wenn wir z.B. den String
 1711 2020-01-20 parsen wollen können wir das mit der Funktion **ymd** machen.

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

1712 ## [1] "2020-01-20"

1713 Dabei erkennt **lubridate** in der Regel die Trennzeichen:

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

1714 ## [1] "2020-01-20"

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

1715 ## [1] "2020-01-20"

¹² *to parse* heißt zergliedern bzw. grammatisch bestimmen.

```
1715 ymd("2020 01 20")
```

1716 ## [1] "2020-01-20"

1717 Wenn die die Anordnung der einzelnen Komponenten anders ist, gibt es einfach eine andere Funktion.

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

1718 ## [1] "2020-01-20"

1719 Jetzt stellt sich die Frage, was der Vorteil ist, wenn R ein Datum parst.

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

1720 Wir können jetzt mit `d` arbeiten und einzelne Komponenten extrahieren.

```
day(d)
```

1721 ## [1] 20

```
month(d)
```

1722 ## [1] 1

```
year(d)
```

1723 ## [1] 2020

1724 Oder auch Zeiteinheiten hinzufügen oder abziehen.

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

1725 ## [1] "2020-01-30"

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

1726 ## [1] "2000-01-20"

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

1727 ## [1] "2020-01-21 01:00:00 UTC"

1728

1729 **Aufgabe 34: Arbeiten mit Datum und Zeit**

- 1731 • Parsen Sie folgende Zeitangaben 23.1.2020, 13.2.1992 20:55:23, Mar/3/97 und 10.7.2020 19:15
1732 und speichern Sie diese in einen Vektor d.
1733 • Extrahieren Sie nun aus jedem Element aus d das Jahr und die Stunde.
1734 • Fügen zu jedem Element in d 10 Tage hinzu.

1735 **11.1 Arbeiten mit Zeitintervallen**

1736 Mit zwei Zeitpunkten lassen sich Zeitintervalle (Periods) erstellen, dafür können wir die Funktion
1737 `interval()` aus dem Paket `lubridate` verwenden¹³.

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

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

1738 Wir können jetzt mit `int` arbeiten und beispielsweise das Intervall verschieben,

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

1739 `## [1] 2023-03-18 UTC--2024-03-18 UTC`

1740 die Länge des Intervalls berechnen

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

1741 `## [1] 31536000`

1742 oder testen, ob ein Datum innerhalb des Intervalls liegt.

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

1743 `## [1] TRUE`

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

1744 `## [1] FALSE`

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

1745 %within% funktioniert genauso mit Vektoren oder mit mehreren Intervallen. Wir könnten also zwei Intervalle
 1746 definieren (z.B. Ostern und Pfingsten).

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

1747 Und Überprüfen welche Termine in eines der zwei Intervalle fallen.

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

# Ostern
termine %within% ostern

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

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

1749 ## [1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE TRUE FALSE FALSE

1750 Der Zeit-Datentyp wird auch oft benutzt, um den Zeitbedarf von längeren Berechnungen zu messen.

```
t1 <- now()
mean(runif(1e7))
```

1751 ## [1] 0.4999484

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

1752 ## [1] 0.7895103

1753 11.2 Formatieren von Zeit

1754 Für die Ausgabe in Berichten oder Grafiken soll das Datum oft in einer speziellen Form dargestellt werden.
 1755 Die Funktion `format()` bietet Möglichkeiten ein Datumsobjekt zurück in Text umzuwandeln.

1756 Ein Beispiel wäre `ymd("2021-2-10")` als 10.2.21 auszugeben.

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

1757 ## [1] "21.02.21"

1758 Dabei handelt sich bei `%d.%m.%y` um Abkürzungen für die unterschiedlichen Komponenten eines Datumobjekts. Siehe dazu die Hilfeseite von `strptime` (`help(strptime)`).

1760

Aufgabe 35: Arbeiten mit Intervallen

- 1763 Wie viele Einträge aus dem Vektor v1 befinden sich in einem Intervall, das zwischen dem 1.3.2021 und dem
1764 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"
)
```

1765 **11.3 Zeitreihen**

- 1766 In der Forstbranche haben wir oft mit Zeitreihen zu tun. Zeitreihen sind Variablen, die in zeitlichen
1767 Intervallen vorliegen. In der Regel monatlich, vierteljährig oder jährlich, wobei die Intervalle zwischen den
1768 Messungen immer gleich lang sind. Wiederholungsmessungen von Forstinventuren (Forsteinrichtungen, Be-
1769 triebseinventuren, die meisten forstl. Experimente, die BWI, ...) sind also in der Regel keine Zeitreihen in
1770 engeren Sinne, turnusmäßig vermessene Versuchsflächen, wie sie z. B. die Versuchsanstalten unterhalten oder
1771 jährlich gemeldete Holzpreise jedoch schon.
- 1772 Statistisch gesehen haben Zeitreihen Besonderheiten gegenüber anderen Variablen, da Sie in erster Linie von
1773 Ihrer eigenen Vergangenheit abhängen (autokorreliert sind) und auch die Abhängigkeit anderer Variablen in
1774 der Regel nur sinnvoll ist, wenn deren Vergangenheit mitberücksichtigt wird (cross-correlation). Konven-
1775 tionalelle Statistik ist oft nicht ausreichend, um Zeitreihen zu analysieren. Angefangen mit der Datendarstellung
1776 gibt es spezifische Zeitreihen-Funktionen, welche auch alle in R integriert sind. Aus diesem Grund sollten Sie
1777 Zeitreihen als solche speichern. Viele Funktionen erkennen den Datentyp und führen entsprechend spezifische
1778 Operationen durch. Laden wir z. B. die Holzpreise für Fichte 2b (das sog. Leitsortiment), das Holzaufkommen
1779 dieses Sortiments und die Preise für Nadelholz vom statistischen Bundesamt¹⁴. Wir laden die Daten
1780 zunächst als csv:

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

- 1781 Mit der Funktion `ts` werden die Daten in ein Zeitreihenobjekt überführt (geparst). Die Spalte mit den
1782 Jahren ist dann nicht mehr nötig, da die Informationen zur Zeit keine eigene Variable mehr sind, sondern
1783 zu Metainformationen werden. Die Spalten sollen 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

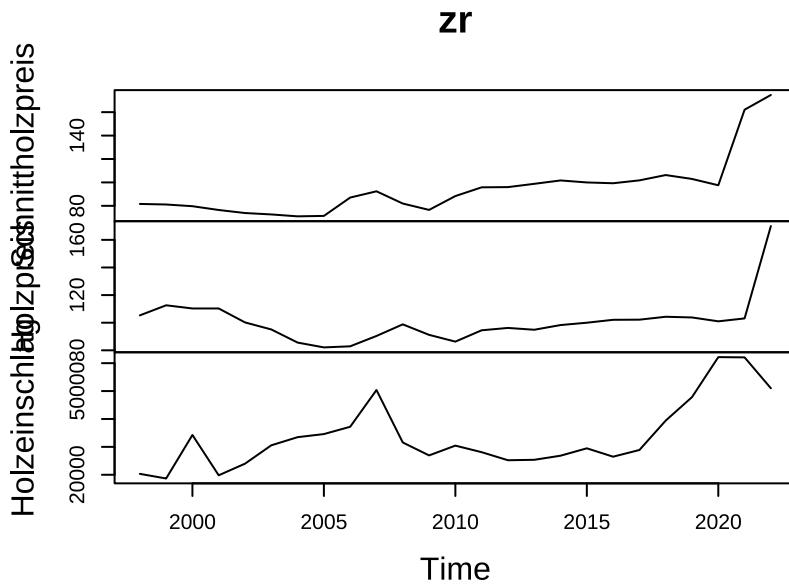
¹⁴Sie können sich die Daten auch selbst über die Website laden oder das Paket `wiesbaden` verwenden, um die Daten direkt in R zu laden. Jedoch müssen Sie sich zuerst registrieren

1784 `## [1] "double"`

```
# Basis Datentypen (siehe Kapitel Variablen, Funktionen und Datentypen),
# sondern sind eine Kategorie innerhalb des Datentyps "Liste".
```

1785 Die wichtigsten Argumente sind - `data` Vektor oder Matrix, der nur die Daten enthält - `start` Startzeitpunkt -
 1786 `frequency` Anzahl der Beobachtungen pro Zeiteinheit, also z. B. 12 bei monatlichen oder 4 bei quartalsweisen
 1787 Erhebungen

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

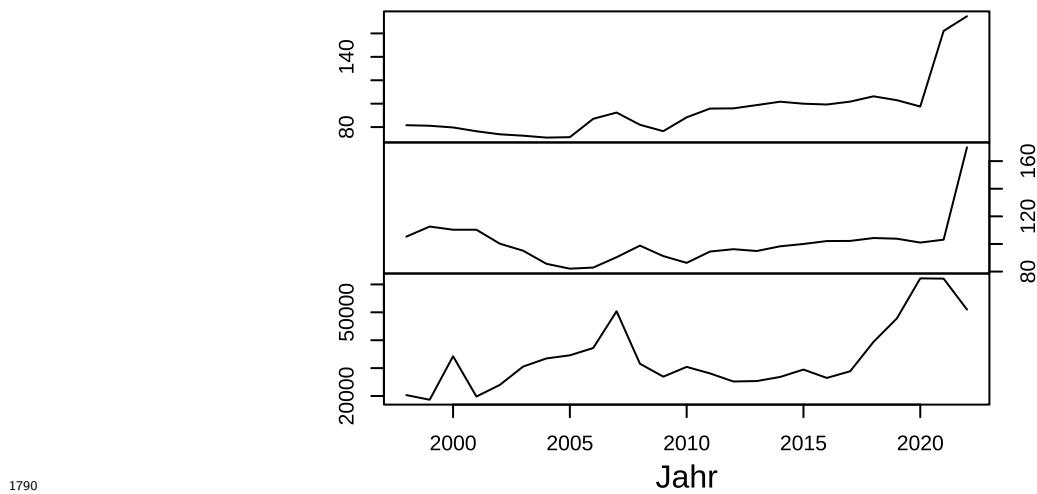


1788

1789 Wir können low-level Layer hinzufügen. Die meisten base plot Funktionen funktionieren auch für Zeitreihen

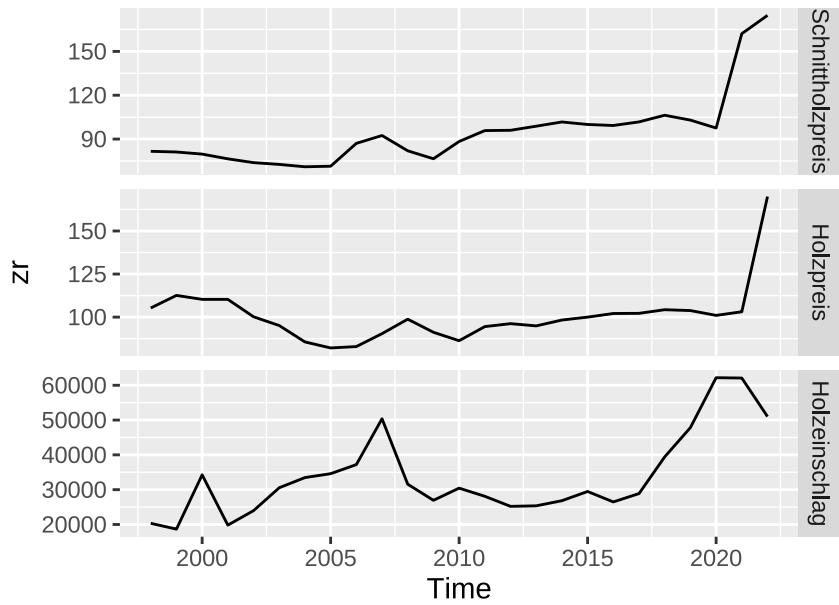
```
plot(zr, ann = FALSE, yax.flip = TRUE)
mtext(side = 3, cex = 1.5, font = 2, "Holzmarktentwicklung seit 1998", line = 2)
mtext(side = 1, cex = 1, line = 2.5, "Jahr")
```

Holzmarktentwicklung seit 1998



¹⁷⁹¹ Das Paket **forecast** ermöglicht automatisierte Zeitreihenplots im **ggplot2** Stil. Damit ist auch das Problem
¹⁷⁹² der y-Achsenbeschriftungen gelöst.

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

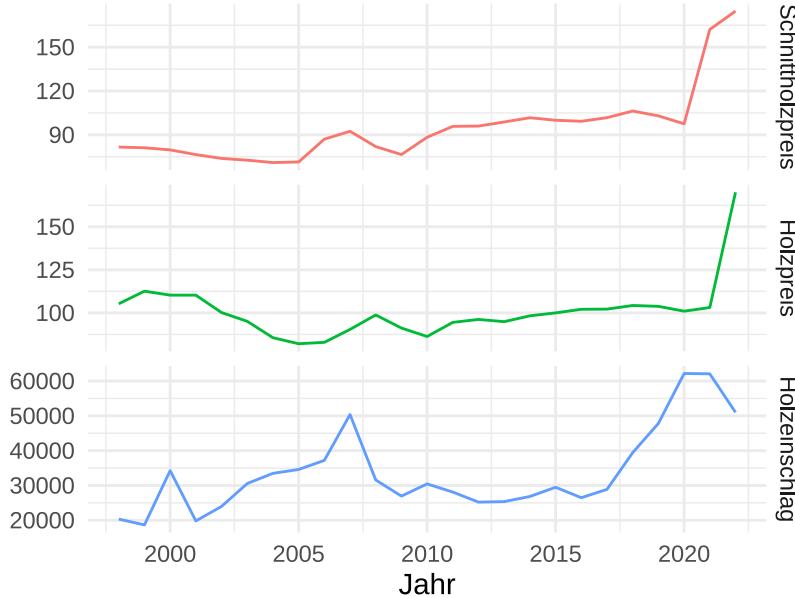


¹⁷⁹³

¹⁷⁹⁴ Wir können die Abbildung im **ggplot2** Stil ändern. Hier nur ein Minimalbeispiel zur Veranschaulichung.
¹⁷⁹⁵ Siehe Kapitel 8.4 **ggplot2**: Eine Alternative für Abbildungen für mehr Details.

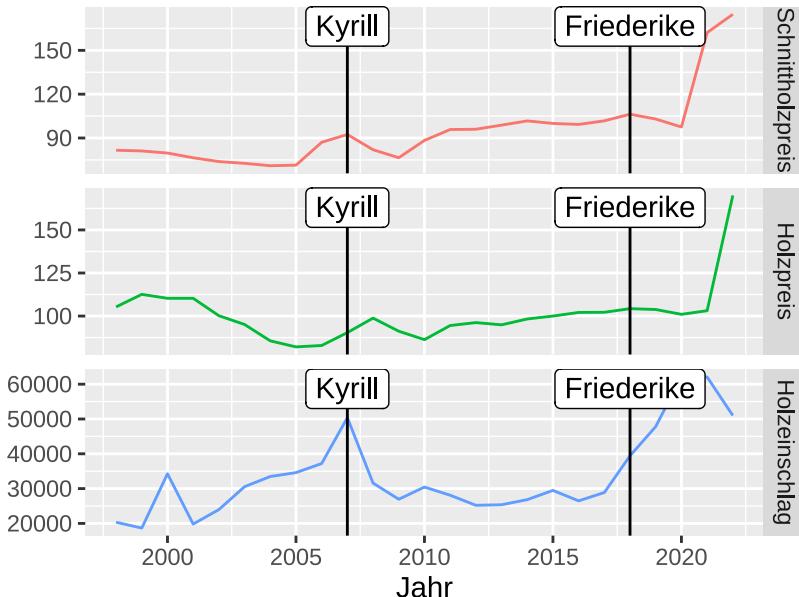
```
zr_autoplot <- autoplot(zr, facets = TRUE, colour = TRUE) +  
  ylab("") + # Keine y-Achsenbeschriftung  
  xlab("Jahr") +  
  guides(colour = "none") # Keine Legende
```

```
zr_autoplot + theme_minimal()
```



1796

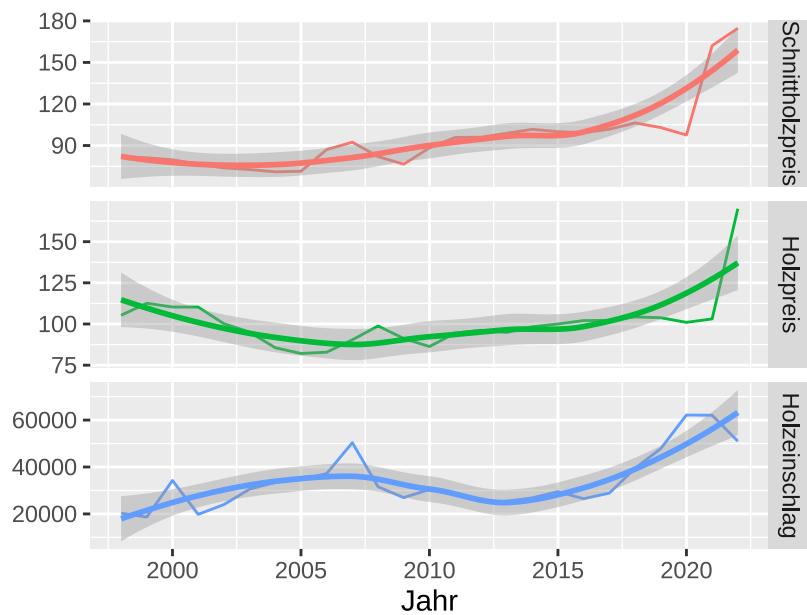
```
z2 <- zr_autoplot + geom_vline(xintercept = c(2007, 2018))
z2 + annotate(x = 2007, y = +Inf, label = "Kyrill", vjust = 1, geom = "label") +
  annotate(x = 2018, y = +Inf, label = "Friederike", vjust = 1, geom = "label")
```



1797

- 1798 Eine Trendlinie macht hier (sowie generell in Zeitreihendaten) offensichtlich keinen Sinn. Daher verwenden
 1799 wir den sog. Loess-Glättter, um einen Trend in die Daten zu legen. Der Loess-Glättter ist ein sehr flexible
 1800 Kurve.

```
zr_autoplot + geom_smooth() +  
guides(colour = "none") # Nochmals nötig, da geom_smooth() wieder eine Legende erzeugt
```



1801

1802 12 Aufgaben Wiederholen (for-Schleifen)

1803 Um einfache Programme zu schreiben, müssen Sie den Ablauf der Programmcodes kontrollieren können.
 1804 Kontrollieren bedeutet in diesem Zusammenhang, dass Codeabschnitte nur unter definierten Bedingungen
 1805 ablaufen. Sie programmieren also zwei Sachen. 1) den Code selbst und 2), die Bedingungen die erfüllt sein
 1806 müssen, damit der Code ausgeführt wird. Der Code muss do generisch geschrieben sein, dass er komplett
 1807 durchläuft, auch wenn unterschiedliche Dateneingaben gemacht werden. Diese Kontrollbedingungen ermög-
 1808 lichen es Ihnen generisch zu programmieren. Sie schreiben Ihren Code also nicht speziell maßgeschneidert
 1809 für ein Problem, sondern so generell, dass er für mehrere Auswertungen funktioniert. Um dies zu gewähr-
 1810 leisten, müssen Sie bestimmte Situationen vorhersehen und abfangen. Hierbei helfen Ihnen Kontrollstruktu-
 1811 ren (**Control Flow**). Grundsätzlich gibt es **Control Flow** Funktionen zur Wiederholung von Codeblöcken
 1812 (Schleifen) und logische Bedingungen (bedingte Anweisung).

1813 12.1 Schleifen

1814 Bis jetzt wurden alle Skripte einfach der Reihe nach abgearbeitet und zwischendurch bestimmte Programm-
 1815 teile, je nach Situation, selbstständig ausgeführt oder übersprungen. Mit einer Schleife kann man erreichen,
 1816 dass eine Gruppe von Befehlen (der sog. Schleifenrumpf) mehrfach abgearbeitet wird, zum Beispiel wenn
 1817 bestimmte Auswertungsschritte auf mehrere Datensätze oder Variablen angewendet oder Funktionen mit un-
 1818 terschiedlichen Parametern oder Startwerten aufgerufen werden sollen. Weitere Anwendungsmöglichkeiten
 1819 sind iterative Algorithmen, in denen die Eingabewerte des aktuellen Iterationsschrittes von einem vorherigen
 1820 abhängig sind. Besonders in Simulationen kommen Schleifen häufig zum Einsatz, da große Anzahlen von
 1821 Wiederholungen benötigt werden.

1822 Man unterscheidet zwischen zwei Arten von Schleifen: Bei den **for()**-Schleifen steht die Anzahl der Wieder-
 1823 holungen schon beim Eintritt in die Schleife fest, während die **while()**-Schleifen so lange ausgeführt werden,
 1824 bis eine Bedingung nicht mehr wahr ist. Mit der Funktion **break** wird eine Schleife abgebrochen und die
 1825 Programmausführung wird nach der Schleife fortgesetzt.

1826 Die wesentlichen Befehle sind

- 1827 • **for (i in X) {Code}**

1828 Wiederhole den Code im “Schleifenrumpf” für jedes Element aus X.

- 1829 • **while(Bedingung) {Code}**

1830 Wiederhole den Code im “Schleifenrumpf” so lange die logische Bedingung erfüllt ist.

- 1831 • **break()**

1832 Brich die Schleife ab. **break()** muss im Schleifenrumpf an der richtigen Stelle ausgeführt werden. Gute
 1833 Praxis ist jedoch, die for oder while Bedingungen, dass kein **break()**nötig ist, da **break()** anfällig für
 1834 Programmierfehler ist.

1835 **12.1.1 Wiederholen von Befehlen mit `for()`.**

1836 Steht vor Beginn der Schleife fest wie viele Schleifendurchgänge benötigt werden, wenn zum Beispiel in
 1837 einer Simulationen 99 Realisierungen erzeugt oder alle Elemente eines Vektors verarbeitet werden sollen,
 1838 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)
}
```

1839 ## [1] 1
 1840 ## [1] 2
 1841 ## [1] 3

1842 Das `i` steht in diesem Beispiel für die Schleifen-Variable. Sie muss nicht `i` heißen, sondern kann jeden
 1843 zulässigen Namen annehmen. Das `X` steht für einen existierenden Vektor oder eine existierende Liste bzw.
 1844 einen Ausdruck, der ein solches Objekt liefert (der Objektname ist ebenfalls frei wählbar). `for` und `in` sind
 1845 Schlüsselworte, sie müssen, ebenso wie die runden Klammern, vorhanden sein.

1846 Im ersten Durchgang erhält die Schleifen-Variable `i` den ersten Wert von `X` und der Schleifenrumpf wird
 1847 mit diesem Wert ausgeführt. Die Variable `i` nimmt nacheinander so lange die Werte von `X` an, bis ihr alle
 1848 Elemente zugewiesen wurden.

1849 Das folgende Beispiel wird zwar besser durch die entsprechende Vektoroperation gelöst, zeigt aber sehr
 1850 deutlich die Arbeitsweise der `for`-Schleife.

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

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

1851 ## [1] 4
 1852 ## [1] 9
 1853 ## [1] 25

1854

1855 **Aufgabe 36: Schleifen 1**

1857 Verwenden Sie den Vektor `k <- c(1, 3, 9, 12, 15)` und schreiben Sie folgende `for`-Schleifen:

1858 1. Eine Schleife, die jedes Element aus `k` ausgibt.

- 1859 2. Eine Schleife, die zu jedem Element aus `k` 10 addiert und den neuen Wert ausgibt.
 1860 3. Eine Schleife wie in 2), aber der neue Wert ($k + 10$) soll jetzt nicht mehr ausgegeben werden, sondern
 1861 in `k10` gespeichert werden. Stellen Sie sicher, dass `k10` wieder von der Länge 5 ist.

1862

- 1863 Die Funktion `for()` ermöglicht es, einen Befehl beliebig oft zu wiederholen. Z.B. der folgende Ausdruck zieht
 1864 10-Mal eine Stichprobe der Größe 1 aus dem Vektor `v`. Beachten Sie, dass die Schleifen-Variable `i` selbst gar
 1865 nicht im Schleifenrumpf vorkommt. Das Ziel dieser Schleife ist nicht die Elemente des Vektors abzuarbeiten,
 1866 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))
}
```

1867 ## [1] 3
1868 ## [1] 1
1869 ## [1] 3
1870 ## [1] 3
1871 ## [1] 2
1872 ## [1] 3
1873 ## [1] 2
1874 ## [1] 2
1875 ## [1] 1
1876 ## [1] 4

1877 Auf gleiche Weise kann man auch über die Variablen eines Dataframes iterieren¹⁵. Das folgende Beispiel
 1878 hat zum Ziel die Funktionsweise von Schleifen zu verdeutlichen. Schleifen haben in R jedoch den Nachteil,
 1879 dass sie sehr langsam operieren. Wenn es geht, sollte man Alternativen verwenden. Die Funktionsweise
 1880 wiederholender Auswertungen wird jedoch mit `for`-Schleifen deutlicher. Aus diesem Grunde werden wir uns
 1881 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.

```

1882 ## [1] "Buche"
1883 ## [1] 52
1884 ## [1] "Eiche"
1885 ## [1] 64
1886 ## [1] "Eiche"
1887 ## [1] 62
1888 ## [1] "Buche"
1889 ## [1] 85

```

1890

Aufgabe 37: for-Schleife

1893 Lesen Sie den Datensatz `bhd_1.txt` ein und verwenden Sie eine `for`-Schleife.

- 1894 • Ziehen Sie 500-Mal je 35 Beobachtungen für den BHD.
- 1895 • Berechnen Sie jeweils den Mittelwert aus den 35 Werten.
- 1896 • Speichern Sie die 500 Mittelwerte in einem neuen Vektor `mittel`.
- 1897 • Wie ist die Verteilung dieser 500 Mittelwerte zu interpretieren?

1898 **12.1.2 Wiederholen von Befehlen mit `while()`**

1899 Die `while`-Schleifen finden Anwendung, wenn die Anzahl der zu durchlaufenden Wiederholungen vorher
 1900 nicht bekannt ist, wie zum Beispiel bei Iterationsverfahren, die bis zu einer gewissen Genauigkeit durchlaufen
 1901 werden sollen. Die `while`-Schleife besteht in R aus dem Schlüsselwort `while()` und einer Bedingung in runden
 1902 Klammern.

```

while (Bedingung) {
  # Schleifenrumpf
}

```

1903 Sie ist in der praktischen Programmierung nicht so relevant wie die `for`-Schleife. Sie sei deshalb hier nur
 1904 kurz erwähnt. Die Abbruchbedingung wird jedes Mal geprüft bevor der Schleifenrumpf durchlaufen wird.
 1905 Die Bedingung wird ausgewertet und wenn diese `TRUE` ist, wird der Schleifenrumpf ausgeführt und danach
 1906 erneut die Bedingung überprüft. Ist die Bedingung nicht erfüllt, wird der Schleifenrumpf nicht durchgeführt
 1907 und die Schleife beendet. Ist die Bedingung bereits vor Eintritt in die Schleife nicht erfüllt, wird die Schleife
 1908 gar nicht erst durchlaufen.

1909 Da `while`-Schleifen also so lange ausgeführt werden, bis die Bedingung nicht mehr erfüllt ist, kann eine
 1910 Endlosschleife entstehen. Dies kann passieren, wenn man nicht sauber programmiert hat. Wenn innerhalb
 1911 der Schleife nicht dafür gesorgt wird, dass die Bedingung irgendwann nicht mehr erfüllt wird, so läuft die
 1912 Schleife immer weiter. Steckt R in einer Schleife fest und reagiert nicht mehr, kann der Befehl unter Linux
 1913 mit `Strg + C` und unter Windows mit `Esc` abgebrochen werden. Alternativ können Sie auf das rote STOP
 1914 Symbol über der Konsole klicken.

1915 12.2 Bedingte Ausführung von Codeblöcken

1916 Innerhalb eines Skripts ist es mitunter notwendig je nach aktueller Situation unterschiedlich fortzufahren.
 1917 Die Situation wird mit einem logischen Ausdruck, einer sogenannten Bedingung, geprüft. Je nachdem, ob
 1918 die Bedingung wahr (TRUE) oder falsch (FALSE) ist, werden unterschiedliche Programmteile ausgeführt, der
 1919 jeweils andere Teil bleibt unberücksichtigt. Danach wird in jedem Fall die Programmausführung, mit den
 1920 auf die bedingte Anweisungen folgenden Anweisung, fortgesetzt. In R kann dies mit dem if-else-Konstrukt
 1921 realisiert werden, welches aus den Schlüsselwörtern `if()` und `else` sowie der in runde Klammern gefassten
 1922 Bedingung besteht.

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

1923 Im folgenden Beispiel sollen die bisher abstrakt beschriebenen Vorgänge praktische Anwendung finden. In
 1924 dem Beispiel wird zunächst durch zufälliges Ziehen einer Zahl aus dem Bereich eins bis sechs ein Würfelwurf
 1925 simuliert. Anschließend wird der Würfelwurf mit einem if-Ausdruck kommentiert. Ist die Bedingung, es wurde
 1926 eine Sechs gewürfelt, erfüllt, wird der Code innerhalb der geschweiften Klammern ausgeführt, ansonsten wird
 1927 der Klammerinhalt ignoriert.

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

1928 In den meisten Fällen ist es für R irrelevant, ob sich zwischen den verschiedenen Elementen Leerzeichen oder
 1929 Zeilenumbrüche befinden. Bei dem else-Ausdruck dagegen wird ein Fehler erzeugt, wenn vor dem `else` nicht
 1930 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.")
}
```

1931 ## [1] "Beim nächsten Wurf klappt's bestimmt."

1932

1933 **Aufgabe 38: Bedingte Programmierung**

- 1935
- Wenn keine 6 gewürfelt wurde, lassen Sie zusätzlich ausgeben welche Zahl gewürfelt wurde.

1936

 - Wiederholen Sie den Würfelwurf 10 Mal.

1937 13 (R)markdown

1938 13.1 Markdown Grundlagen

1939 Die Idee von Markdown ist, einfach Text strukturieren zu können und das ganze ohne umfangreiche Pro-
 1940 grammre zu erstellen. Im nächsten Abschnitt sehen wir dann, wie man Markdown und R-Code verbinden
 1941 kann. Hier soll es jedoch erst einmal darum gehen, die einfachsten Bausteine von Markdown vorzustellen.

1942 Am Anfang jedes Dokuments kommt eine Präambel. Diese fängt mit --- an und hört auch wieder mit ---
 1943 auf. Innerhalb der Präambel können dann Metainformationen über das Dokument festgelegt werden, dies
 1944 beinhaltet im einfachsten Fall: Titel, Autor und Datum. Das würde dann so aussehen:

```
1945 ---
1946 title: "Ein Titel"
1947 author: "Der, der es geschrieben hat"
1948 date: "März 2021"
1949 ---
```

1950 Danach folgt strukturierter und formatierter Text. Verschiedene Hierarchieebenen von Überschriften können
 1951 mit der Anzahl an # festgelegt werden. So ist eine Überschrift erster Ordnung # Kapitel eine Überschrift
 1952 zweiter Ordnung ## Unterkapitel usw.

1953 Listen können erstellt werden, wenn man am Anfang jeder Zeile ein - oder 1. schreibt.

```
1954 - Erster Eintrag
1955 - Zweiter Eintrag
1956 - Dritter Eintrag
```

1957 wird zu

```
1958   • Erster Eintrag
1959   • Zweiter Eintrag
1960   • Dritter Eintrag
```

1961 Eine zentrale Idee von Markdown ist es Text einfach zu formatieren. Werden eine oder mehrere Wörter mit
 1962 zwei Sternchen (**) eingefasst wird dieser Text **fett** dargestellt. Also aus **wichtig** wird **wichtig**. Das
 1963 gleiche funktioniert auch mit *kursiven* Text, jedoch muss man hier noch ein Sternchen verwenden, also aus
 1964 *kursiv* wird *kursiv*. Soll ein text fett und kursiv sein, kann man drei Sternchen verwenden. Aus ***sehr
 1965 wichtig*** wird dann **sehr wichtig**.

1966 Weitere Elemente wie Links oder Abbildungen können einfach eingebunden werden. Links werden mit [Link
 1967 text](url) in den Text integriert. Beispielsweise ist eine gute Idee bei [stackoverflow](#) bei Problemen nach
 1968 einer Lösung zu suchen. Dieser Link wurde mit [stackoverflow](www.stackoverflow.com) erstellt.

1969 Für Abbildungen gibt es einen ganz ähnlichen Syntax. Mit ![Das R Logo](abb/r_logo.png) wird die
 1970 Abbildung r_logo.png eingebunden mit der Beschriftung: "Das R Logo".



Abbildung 9: Das R Logo

1971

1972 **Aufgabe 39: Arbeiten mit markdown**

1973

1974 Verwenden Sie das folgende Markdowndokument:

```
1975 ---
1976 title: "Dokument"
1977 author: "Ihr Name"
1978 date: "März 2021"
1979 ---
1980
1981 # Einleitung
1982
1983 # Methoden
```

- 1984 1. Kopieren Sie die Vorlage in ein Dokument, das `test.md` heißt.
- 1985 2. Fügen Sie zwei Überschriften zweiter und dritter Ordnung hinzu.
- 1986 3. Fügen Sie einen *kursiven* Text hinzu.
- 1987 4. Fügen Sie einen Link zu einer Website hinzu.
- 1988 5. Kompilieren Sie die Datei, indem Sie in Rstudio auf Preview drücken (Abbildung 10).

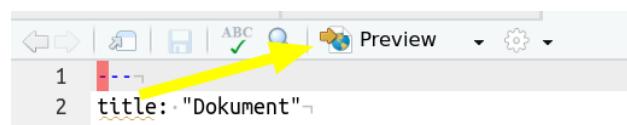


Abbildung 10: Kompilieren einer md-Datei.

1989 **13.2 R und Markdown**

1990 Markdown macht es bereits einfach Textdokumente und Dokumentationen zu verfassen, aber die wirkliche
 1991 Stärke liegt in der Möglichkeit R und Markdown zu kombinieren. Man spricht dann von Rmarkdown. Ein
 1992 weiteres Strukturelement, das wir noch nicht kennen gelernt haben, sind Code-Blöcke.

```

1993   ` ` `
1994 a <- 1:10
1995 a[1]
1996   ` ` `
1997 erzeugt

1998 a <- 1:10
1999 a[1]

2000 Momentan wird noch kein Code ausgeführt, sondern lediglich der Code, als Code dargestellt. Rmarkdown
2001 bietet nun die Möglichkeit Code beim kompilieren16 auszuführen. Dafür müssen wir nur einen Code-Block
2002 als R-Code-Block kennzeichnen.

```

```

2003   ` ` ` {R}
2004 a <- 1:10
2005 a[1]
2006   ` ` `
2007 erzeugt

a <- 1:10
a[1]

2008 ## [1] 1

```

2009 Beachte, die Variable `a` wird beim kompilieren erzeugt und steht dann R zur Verfügung. R-Code-Blöcke
2010 werden auch als Code Chunks bezeichnet. Diese Chunks können sehr genau angesprochen und angepasst
2011 werden. Einige wichtige Argumente sind:

- 2012 • `echo`: Gibt an, ob der Quelltext angezeigt werden soll oder nicht.
- 2013 • `result`: Gibt an, ob die Ergebnisse gezeigt werden sollen oder nicht.
- 2014 • `eval`: Diese Option gibt an ob der Chunk ausgeführt werden soll oder nicht.

2015

2016 Aufgabe 40: Arbeiten mit Rmarkdown

2018 Erstellen Sie eine neue Rmarkdown Datei mit dem Namen `test1.Rmd`. Erstellen Sie zwei Code-Chunks.
2019 Der erste soll nicht angezeigt werden und darin werden die Daten geladen (`bhd_1.txt`). Im zweiten Chunk
2020 plotten Sie das Alter der Bäume gegen den BHD. Was passiert mit dem Plot, wenn Sie die Datei kompilieren
2021 (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.

2022 14 Räumliche Daten in R

2023 14.1 Was sind räumliche Daten

2024 Räumliche Daten sind Beobachtungen, wie wir sie schon oft gesehen haben, mit einem räumlichen Bezug. Der
 2025 Unterschied zu nicht räumlichen Daten liegt darin, dass räumliche Daten eindeutig im *Raum* verortet werden
 2026 können. Häufig werden sogenannte Geoinformationssysteme (GIS) zum Arbeiten mit räumlichen verwendet.
 2027 R kann in vielerlei Hinsicht wie ein GIS eingesetzt werden und hier werden einige Grundfunktionalitäten
 2028 dafür besprochen. Räumliche Daten werden in zwei unterschiedliche Datentypen unterteilt: Vektor- und Ras-
 2029 terdaten. Vektordaten modellieren einzelne Objekte (= *Features*). Rasterdaten modellieren eine Oberfläche.
 2030 Vektordaten bestehen aus zwei Komponenten: 1) einer Geometrie, die die Form und Lage der Daten definiert
 2031 und 2) Attributen, den tatsächlichen Daten. Räumliche Daten werden oft als *Features* bezeichnet. Ein Feature
 2032 ist die räumliche Einheit einer Beobachtung. Je nach Art der räumlichen Daten können Features entweder
 2033 Punkte (z.B. ein Baum), Linien (z.B. eine Straße) oder Polygone (z.B. ein See) sein. Auch können mehrere
 2034 Geometrien zu einem Feature zusammengefasst werden. Ein Beispiel wäre eine Beobachtung für ein Land,
 2035 das aber aus mehreren Polygonen bestehen kann (z.B. Festland und Inseln). Features können dann weitere
 2036 Attribute (= Attributdaten) haben, z.B. eine ID, Name oder was auch immer man gemessen hat.
 2037 Rasterdaten bestehen aus einer Oberfläche von gleichgroßen Kacheln (= *Pixel*), die ein Gebiet abdecken.
 2038 Meist sind Pixel viereckig, aber das ist keine Voraussetzung. Dabei hat jedes Pixel einen Wert (das kann
 2039 auch ein fehlender Wert sein). Typische Beispiele für Rasterdaten sind Landnutzung oder Seehöhen.
 2040 In R kann sowohl mit Vektor- als auch mit Rasterdaten gearbeitet werden. Für Vektordaten bietet sich das
 2041 Paket `sf` an und für Rasterdaten das Paket `raster`.

2042 14.2 Koordinatenbezugssystem

2043 Eine der Herausforderungen für räumliche Daten ist die eindeutige Verortung im Raum. Dazu braucht man
 2044 ein *Koordinatenbezugssystem* (KBS). Einem KBS liegt ein mathematisches Modell der Erde zugrunde. Die
 2045 Details zu KBS werden schnell relativ kompliziert und wir beschränken uns hier lediglich darauf, wie KBS
 2046 verwendet werden können. Dazu müssen wir zwei Fälle unterscheiden: 1) einem Datensatz ein KBS zuweisen
 2047 und 2) Transformation des KBS eines Datensatzes in ein anderes KBS. Die technischen Details werden in
 2048 den folgend Abschnitten besprochen. Für beide Aufgaben müssen wir auf ein KBS verweisen können, ein
 2049 einfacher Ansatz dafür sind die sogenannten *EPSG-Codes*¹⁷.

¹⁷EPSG steht für European Petrol Survey Group

2050 14.3 Vektordaten in R

- 2051 Das Paket `sf` stellt Klassen zum Abbilden von Features zur verfügen, die dann in einem `data.frame` als
 2052 Liste gespeichert werden können. In der Regel erstellen wir Features nicht individuell, sondern lesen diese aus
 2053 externen Datenquellen (z.B. ESRI Shapefile) ein. Zum besseren Verständnis, erstellen wir es einmal manuell.
 2054 Wir haben die Koordinaten für drei Städte (Göttingen, Hannover und Berlin) als geografische Koordinaten
 2055 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))
```

- 2056 Daraus können wir jetzt eine Geometriespalte für einen `data.frame` erstellen

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

- 2057 Somit haben wir die Geometrie in der Variable `geom` gespeichert, aber noch keine dazugehörigen Attribut-
 2058 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)
)
```

- 2059 In einem letzten Schritt möchten wir jetzt die Geometrie (`geom`) und die Attributdaten (`attr`) zusammen-
 2060 führen.

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

- ```
2061 ## Simple feature collection with 3 features and 3 fields
2062 ## Geometry type: POINT
2063 ## Dimension: XY
2064 ## Bounding box: xmin: 9.732 ymin: 51.5413 xmax: 13.405 ymax: 52.52
2065 ## Geodetic CRS: WGS 84
2066 ## name bundesland einwohner geom
2067 ## 1 Goettingen Niedersachsen 119000 POINT (9.9158 51.5413)
2068 ## 2 Hannover Niedersachsen 532000 POINT (9.732 52.3759)
2069 ## 3 Berlin Berlin 3650000 POINT (13.405 52.52)
```

- 2070 Wir können nun mit `staedte` genau so arbeiten wie mit jedem anderen `data.frame` und die Geometrien  
 2071 werden immer ‘berücksichtigt’. Zusätzlich kann man eine Reihe von geometrischen Operationen durchführen.

2072 Wenn ein `data.frame` Punkten hat, kann man dies relative einfach mit der Funktion `st_as_sf()` "räumlich"  
 2073 machen. Für das vorherige Beispiel würden wir zuerst einen `data.frame` mit allen Informationen (zur  
 2074 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)
)
```

2075 Dann kann man mit der Funktion `st_as_sf()` weiter arbeiten:

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

## 2076 14.4 Arbeiten mit Vektordaten

2077 Es gibt sehr viele Funktionen, um mit räumlichen Daten zu arbeiten, von denen wir hier einige vorstellen.

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

```
2078 ## Coordinate Reference System:
2079 ## User input: EPSG:4326
2080 ## wkt:
2081 ## GEOCRS["WGS 84",
2082 ## ENSEMBLE["World Geodetic System 1984 ensemble",
2083 ## MEMBER["World Geodetic System 1984 (Transit)"],
2084 ## MEMBER["World Geodetic System 1984 (G730)"],
2085 ## MEMBER["World Geodetic System 1984 (G873)"],
2086 ## MEMBER["World Geodetic System 1984 (G1150)"],
2087 ## MEMBER["World Geodetic System 1984 (G1674)"],
2088 ## MEMBER["World Geodetic System 1984 (G1762)"],
2089 ## MEMBER["World Geodetic System 1984 (G2139)"],
2090 ## ELLIPSOID["WGS 84",6378137,298.257223563,
2091 ## LENGTHUNIT["metre",1]],
2092 ## ENSEMBLEACCURACY[2.0]],
2093 ## PRIMEM["Greenwich",0,
2094 ## ANGLEUNIT["degree",0.0174532925199433]],
2095 ## CS[ellipsoidal,2],
2096 ## AXIS["geodetic latitude (Lat)",north,
2097 ## ORDER[1],
```

```

2098 ## ANGLEUNIT["degree",0.0174532925199433]],

2099 ## AXIS["geodetic longitude (Lon)",east,

2100 ## ORDER[2],

2101 ## ANGLEUNIT["degree",0.0174532925199433]],

2102 ## USAGE[

2103 ## SCOPE["Horizontal component of 3D system."],

2104 ## AREA["World."],

2105 ## BBOX[-90,-180,90,180]],

2106 ## ID["EPSG",4326]]
```

2107 Wenn wir jetzt zu einem anderen KBS (z.B. EPSG:3035, ein europäisches projiziertes KBS) umrechnen  
 2108 möchten, können wir das mit

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

```

2109 ## Coordinate Reference System:

2110 ## User input: EPSG:3035

2111 ## wkt:

2112 ## PROJCRS["ETRS89-extended / LAEA Europe",

2113 ## BASEGEOGCRS["ETRS89",

2114 ## ENSEMBLE["European Terrestrial Reference System 1989 ensemble",

2115 ## MEMBER["European Terrestrial Reference Frame 1989"],

2116 ## MEMBER["European Terrestrial Reference Frame 1990"],

2117 ## MEMBER["European Terrestrial Reference Frame 1991"],

2118 ## MEMBER["European Terrestrial Reference Frame 1992"],

2119 ## MEMBER["European Terrestrial Reference Frame 1993"],

2120 ## MEMBER["European Terrestrial Reference Frame 1994"],

2121 ## MEMBER["European Terrestrial Reference Frame 1996"],

2122 ## MEMBER["European Terrestrial Reference Frame 1997"],

2123 ## MEMBER["European Terrestrial Reference Frame 2000"],

2124 ## MEMBER["European Terrestrial Reference Frame 2005"],

2125 ## MEMBER["European Terrestrial Reference Frame 2014"],

2126 ## ELLIPSOID["GRS 1980",6378137,298.257222101,

2127 ## LENGTHUNIT["metre",1]],

2128 ## ENSEMBLEACCURACY[0.1]],

2129 ## PRIMEM["Greenwich",0,

2130 ## ANGLEUNIT["degree",0.0174532925199433]],

2131 ## ID["EPSG",4258]],

2132 ## CONVERSION["Europe Equal Area 2001",

2133 ## METHOD["Lambert Azimuthal Equal Area",

2134 ## ID["EPSG",9820]],

2135 ## PARAMETER["Latitude of natural origin",52,

2136 ## ANGLEUNIT["degree",0.0174532925199433],
```

```

2137 ## ID["EPSG",8801]],
2138 ## PARAMETER["Longitude of natural origin",10,
2139 ## ANGLEUNIT["degree",0.0174532925199433],
2140 ## ID["EPSG",8802]],
2141 ## PARAMETER["False easting",4321000,
2142 ## LENGTHUNIT["metre",1],
2143 ## ID["EPSG",8806]],
2144 ## PARAMETER["False northing",3210000,
2145 ## LENGTHUNIT["metre",1],
2146 ## ID["EPSG",8807]]],
2147 ## CS[Cartesian,2],
2148 ## AXIS["northing (Y)",north,
2149 ## ORDER[1],
2150 ## LENGTHUNIT["metre",1]],
2151 ## AXIS["easting (X)",east,
2152 ## ORDER[2],
2153 ## LENGTHUNIT["metre",1]],
2154 ## USAGE[
2155 ## SCOPE["Statistical analysis."],
2156 ## AREA["Europe - European Union (EU) countries and candidates. Europe - onshore and offshore: BBOX[24.6,-35.58,84.73,44.83]"],
2157 ## ID["EPSG",3035]]
2158 ##
```

2159 Die Funktion `st_buffer()` erlaubt es Features zu puffern, mit `st_distance()` kann die Distanz zwischen  
2160 Features berechnet werden, mit `st_area()` kann die Fläche eines Features berechnet werden.

2161 Funktionen wie `st_intersection()`, `st_union()` und `st_difference()` erlauben es geometrische Opera-  
2162 tionen zwischen unterschiedlichen Features zu berechnen. Für eine ausführliche Diskussion siehe auch hier:  
2163 <https://geocompr.robinlovelace.net/geometric-operations.html>.

2164 Normalerweise lesen wir Daten von externen Datenquellen (z.B. ESRI Shapefiles). Das geht mit der Funktion  
2165 `st_read()`.

## 2166 14.5 Rasterdaten in R

2167 Für Rasterdaten gibt es das R-Paket `raster`. Auch hier wollen wir uns wieder auf einige Grundfunktionali-  
2168 täten konzentrieren. Diese umfassen das Einlesen, Zuschneiden, Rechnen und Abfragen von Rastern.

2169 Mit der Funktion `raster()` kann ein Raster in R eingelesen werden.

```
library(raster)
dem <- raster(here::here("data/dem_3035.tif"))
```

2170 `dem` steht für *Digital Elevation Model* und ist ein Raster mit den Seehöhen in Niedersachsen mit einer  
2171 500-m-Auflösung. Wir können diese mit der Funktion `res()`<sup>18</sup> abfragen.

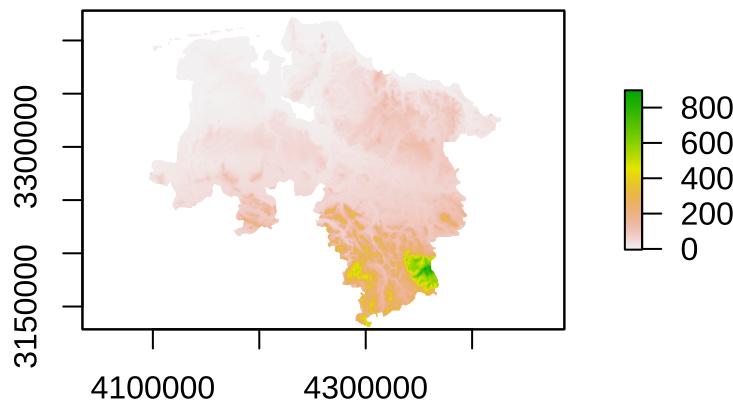
<sup>18</sup>kurz für *resolution* also Auflösung.

```
res(dem)
```

2172 ## [1] 500 500

2173 Bzw. wir können den Raster auch plotten.

```
plot(dem)
```



2174

2175 Wenn wir den Raster `dem` auf ein Gebiet zuschneiden wollen (z.B. Göttingen), müssen wir drei Schritte  
2176 durchführen. Als erstes müssen wir ein Shapefile für Göttingen einlesen.

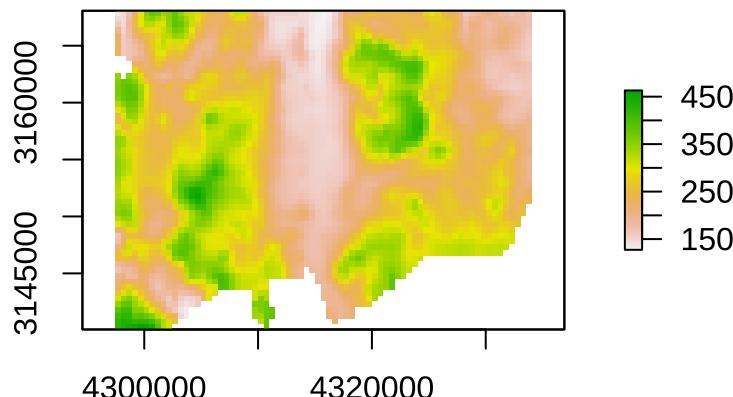
```
goe <- st_read(here::here("data/goettingen/stadt_goettingen.shp"))
```

2177 Dann müssen wir sicherstellen, dass sowohl der Raster `dem` als auch das sf-Objekt `goe` im selben KBS sind.  
2178 Es bietet es sich in der Regel an, das KBS des Vektors zu transformieren. Mit der Funktion `projectRaster()`  
2179 kann das KBS eines Raster transformiert werden.

```
goe <- st_transform(goe, 3035)
```

2180 Mit der Funktion `crop()` kann der Raster `dem` auf Göttingen zugeschnitten werden.

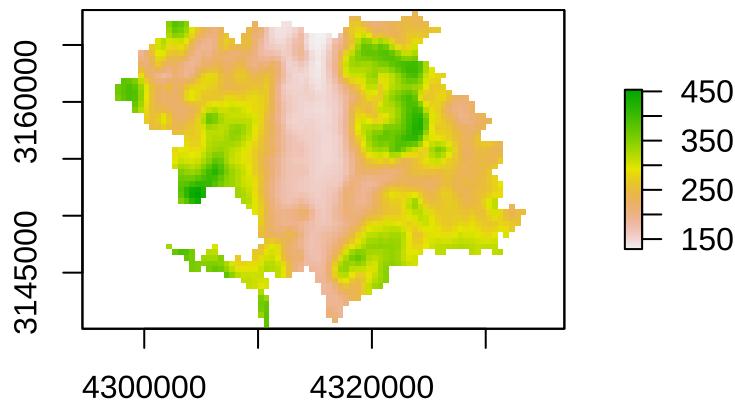
```
dem1 <- crop(dem, goe)
plot(dem1)
```



2181

2182 Der Raster hat jetzt die Größe einer Bounding-Box (BBX) von Göttingen (das ist ein Rechteck, das Göttingen  
2183 umfasst). Mit der Funktion `mask()` kann der Raster auf die genauen Grenzen des Vektors `goe` angepasst  
2184 werden.

```
dem2 <- mask(dem1, goe)
plot(dem2)
```



2185

2186 Wenn wir an bestimmten Punkten den Wert des Rasters abfragen wollen (z.B. an `cities`) von vorhin, dann  
2187 gibt es dafür die Funktion `extract`. Dann müssen wir erst sicherstellen, dass `staedte` und `dem` gleichen  
2188 KBS zu grunde liegt. Dafür transformieren wir einfach `staedte` in das KBS von `dem`. Mit der Funktion  
2189 `projection()` erhalten wir das KBS des Rasters.

```
s1 <- st_transform(staedte, 3035)
```

2190 Wenn wir das KBS eines Objektes nicht kennen, können wir auch einfach das KBS übergeben. Der folgende  
2191 Code-Block macht genau das Gleiche mit dem Vorteil, dass wir keinen EPSG-Code angeben müssen.

```
s1 <- st_transform(staedte, projection(dem))
```

2192 Dann können wir für jede Stadt die Seehöhe abfragen:

```
raster::extract(dem, s1)
```

2193 ## [1] 149.18181 57.21486 NA

2194 Mit `raster::extract()` rufen wir *eindeutig* die Funktion `extract()` aus dem Paket `raster` auf. Wir müssen  
2195 das so machen, weil es im Paket `dplyr` auch eine Funktion `extract()` gibt, die wir hier nicht anwenden  
2196 möchten, da sie einen Fehler verursachen würde.

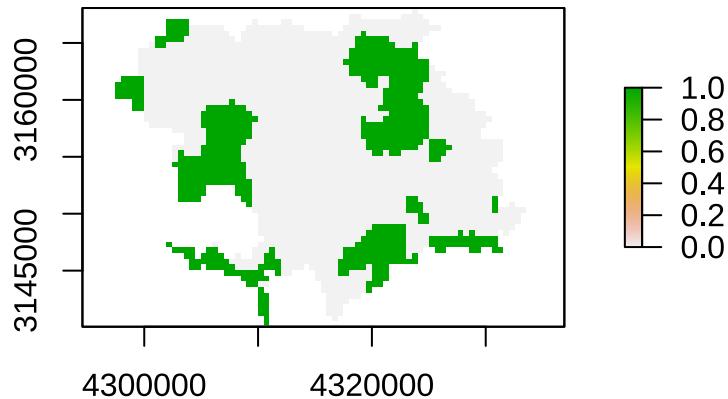
2197 Ein analoges Vorgehen ist auch für Linien und Polygone möglich.

2198 Mit Rastern kann auch einfach gerechnet werden. Wir können z.B. die Seehöhe in Kilometern anstatt Metern  
2199 berechnen:

```
dem_km <- dem / 1e3
```

- 2200 Auch logische Operationen sind möglich, wenn wir alle Rasterzellen mit einer Seehöhe von mehr als 300 m  
 2201 in Göttingen suchen, dann geht das so:

```
dem3 <- dem2 > 300
plot(dem3)
```



2202

- 2203 Wenn wir jetzt auf die Werte des Rasters `dem3` zugreifen wollen, geht das mit eckigen Klammern.

```
head(dem3[])
```

2204 ## [1] NA NA NA NA NA NA

- 2205 Das sind erst einmal viele NA-Werte für die ganzen Zellen, die außerhalb von Niedersachsen liegen. Aber wir  
 2206 können mit so einem Vektor ganz normal arbeiten und z.B. die Fläche des Landes Niedersachsen die eine  
 2207 Seehöhe von mehr als 500m Seehöhe hat ausrechnen.

```
h <- dem3[]
sum(h, na.rm = TRUE) / sum(!is.na(h))
```

2208 ## [1] 0.265713

2209

#### 2210 Aufgabe 41: Arbeiten mit Rastern

- 2212 Verwenden Sie den Raster `wald.tif`, der auf einer 10 m Auflösung den Waldanteil jeder Rasterzelle angibt<sup>19</sup>.  
 2213 Der EPSG-Code für das KBS von `wald.tif` ist 3035. Nehmen Sie an, dass wenn der Waldanteil in einer Raster  
 2214 größer als 50 % ist, dass die Rasterzelle als Wald klassifiziert werden kann. Wie viel Prozent des Göttinger  
 2215 Stadtgebietes sind Wald? Wie ändert sich dieser Wert, wenn sie 70 % anstatt 50 % als Schwellenwert für  
 2216 Wald annehmen?

<sup>19</sup>Die können hier <https://land.copernicus.eu/pan-european/high-resolution-layers/> für ganz Europa bezogen werden

2217

2218 **Aufgabe 42: Studiendesign**

---

- 2219
- 2220 Mit der Funktion `st_sample()` können Sie innerhalb oder entlang eines Features zufällige Punkte legen. Das  
 2221 Argument `n` steuert die Anzahl Punkte und das Argument `type` wie die Punkte angeordnet werden. Für `type`  
 2222 sind für uns die Werte `type = "random"` (komplett zufällig), `type = "regular"` (regelmäßiger Grid) und  
 2223 `type = "hexagonal"` von Bedeutung (ein hexagonaler Grid, d.h. ein sechseckiger Raster). Unglücklicherweise  
 2224 ist das Ergebnis von `st_sample()` erst eine Geometrie. Um daraus ein vollständiges `sf`-Objekt zu machen  
 2225 und problemlos weiter arbeiten zu können, müssen Sie noch einmal die Funktion `st_as_sf()` ausführen.
- 2226 Stellen Sie sich vor, dass wir die tatsächliche Waldbedeckung des Göttinger Stadgebietes **nicht** kennen  
 2227 und wir eine Studie durchführen, um den Anteil des Göttinger Stadgebietes, der mit Wald bedeckt ist  
 2228 herauszufinden. Erstellen Sie dafür einige unterschiedliche Stichproben (diese können in der Anzahl und  
 2229 Anordnung variieren).
- 2230 Berechnen Sie für jedes Stichprobendesign den Anteil an Wald und ein dazugehöriges Konfidenzintervall  
 2231 (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  $n$   
 2232 die Stichprobengröße). Nehmen Sie an, dass eine Rasterzelle Wald ist, sobald > 50 % der Rasterzelle mit  
 2233 Wald bedeckt ist.

2234

2235 **Aufgabe 43: Räumliche Daten**

---

- 2236
- 2237 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)
)
```

- 2238 1. Erstellen Sie ein `sf`-Objekt aus `df1`.
- 2239 2. Puffern Sie jeden Baum mit seinem Kronendurchmesser.
- 2240 3. Berechnen Sie die Kronenfläche jedes Baumes. *Hinweis: Die Funktion `st_area()` könnte dafür hilfreich  
 sein.*
- 2241 4. Welcher Baum hat die größte Kronenfläche?
- 2242 5. Finden Sie für jede Art, den Baum mit der größten Kronenfläche.

2244

---

2245 **Aufgabe 44: Arbeiten mit räumlichen Daten**

---

- 2247 1. Lesen Sie das ESRI Shapefile goettingen/stadt\_goettingen.shp ein.  
2248 2. Wie viele Features befinden sich in dem Shapefile?  
2249 3. Welches Koordinatenbezugssystem (KBS) hat das Shapefile?  
2250 4. Transformieren Sie das Shapefile in das KBS 3035.  
2251 5. Erstellen Sie eine neue Spalte A in der Sie die Fläche jeder Gemeinde/Stadt speichern.  
2252 6. Welche Gemeinde/Stadt (Spalte GEN) ist am größten?  
2253 7. Wählen Sie nun nur die Stadt Göttingen aus.

2254

---

2255 **Aufgabe 45: Arbeiten mit räumlichen Daten 2**

---

- 2257 1. Lesen Sie erneut das ESRI Shapefile goettingen/stadt\_goettingen.shp ein.  
2258 2. Lösen Sie die Gemeindegrenzen auf (die Funktion `st_union()` könnte hier nützlich sein).  
2259 3. Wie groß ist das resultierende Feature?

2260 **15 FAQs (Oft gefragtes)**

2261 **15.1 Arbeiten mit Daten**

2262 **15.1.1 Einlesen von Exceldateien**

- 2263 Mit der Funktion `read_excel()` aus dem Paket `readxl` können Exceldateien direkt in R eingelesen werden.  
2264 Ein Export als csv-Datei aus Excel ist nicht notwendig.

## 2265 16 Literatur

- 2266 Ein guter Überblick über viele der angeschnittenen Themen gibt es in dem [R for data science](#), das online  
2267 frei zugänglich ist. Das on-line Buch [Hands-On Programming with R]{[https://rstudio-education.github.io/  
2268 hopr/index.html](https://rstudio-education.github.io/hopr/index.html)} ist eine nicht-Programmierer freundliche Einführung in R.  
2269 McNamara, Amelia, and Nicholas J Horton. 2018. “Wrangling Categorical Data in r.” *The American Stati-*  
2270 *stician* 72 (1): 97–104.  
2271 Wickham, Hadley. 2014. “Tidy Data.” *Journal of Statistical Software* 59 (10). [https://doi.org/10.18637/jss.  
2272 v059.i10.](https://doi.org/10.18637/jss.v059.i10)