

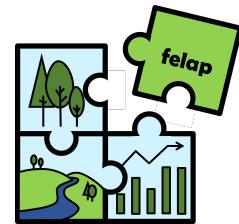
1

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:

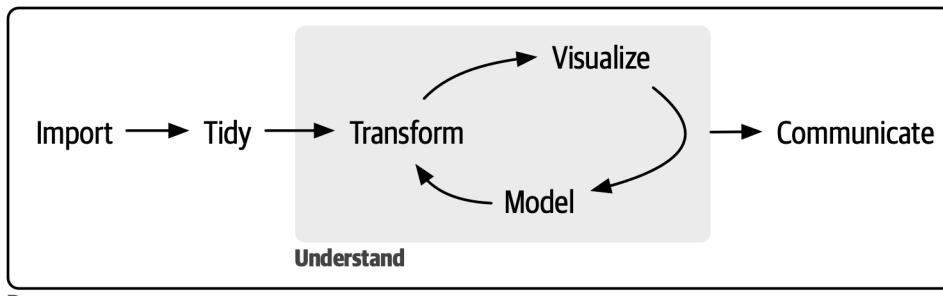
¹⁶ Signer, J. und Husmann, K. (2023) Skript zur Vorlesung Einführung in die Datenanalyse mit R, Georg-August-Universität Göttingen.

¹⁸ Letzte Aktualisierung: 5. 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)	123
115	15.1 Arbeiten mit Daten	123
116	15.1.1 Einlesen von Exceldateien	123
117	16 Zusätzliche Aufgaben	124
118	16.1 Arbeiten mit Daten	126
119	17 Literatur	128

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**).

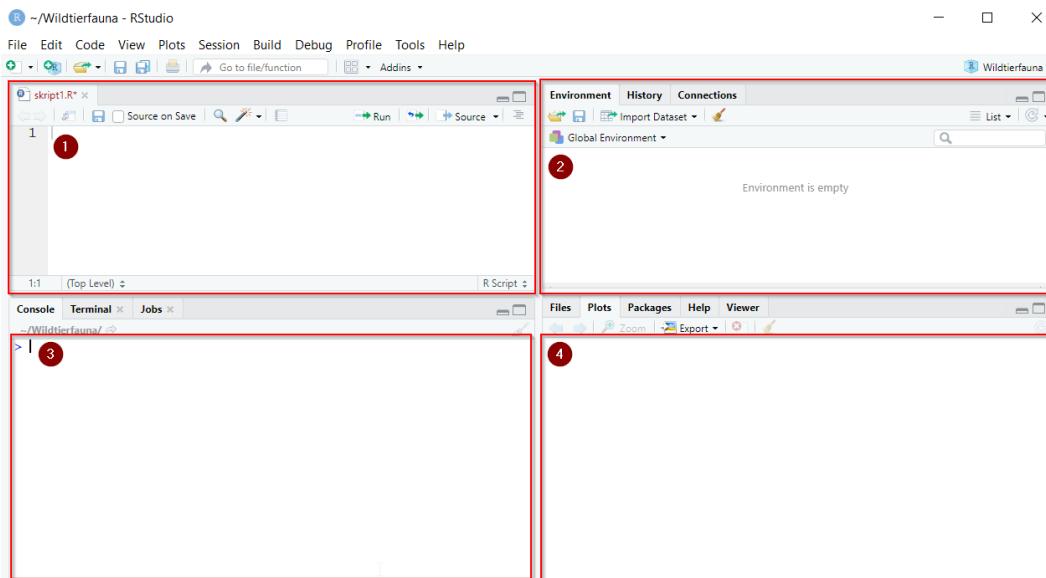


Abbildung 1: RStudio Panes.

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

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

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

10 + 5

156 `## [1] 15`

20 - 10

157 `## [1] 10`

10 * 3

158 `## [1] 30`

100 / 19

159 `## [1] 5.263158`

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

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

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

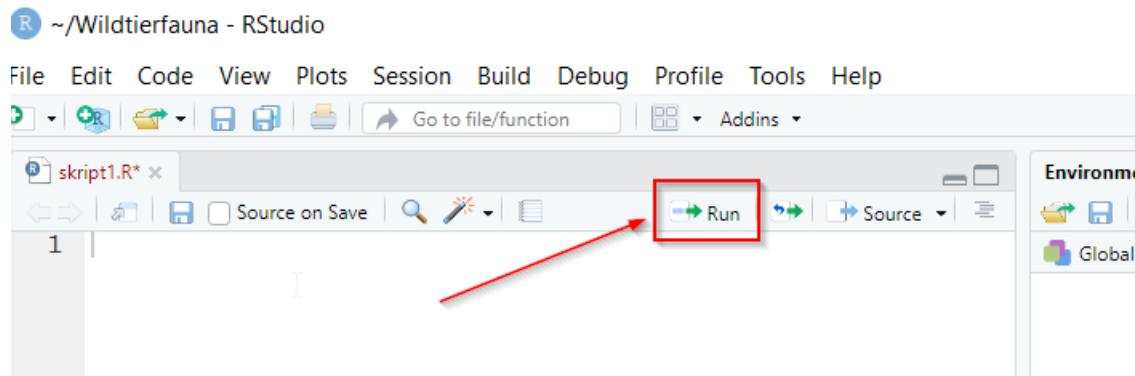


Abbildung 2: Zeilenweises Ausführen von Code in RStudio.

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

183 1.3 Gute Praxis bei der Programmierung

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

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

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

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

199 ## [1] 9

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

206

207 Aufgabe 1: Ausführen von Quellcodes

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

210 Tippen oder kopieren Sie folgenden Code in das Skript:

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

# Einfache Funktion
sqrt(20)
```

211 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

232 führt zu einem Fehler.

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

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

235 ## [1] 15

```
b / a
```

236 ## [1] 0.5

```
a^b
```

237 ## [1] 1e+05

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

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

239 ## [1] 15

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

240 ## [1] 30

241 Mit der Funktion `rm()` können Variablen, die nicht mehr benötigte Variablen, wieder gelöscht werden.
242 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
```

244 ## [1] TRUE

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

245 ## [1] FALSE

2.2 Datentypen

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

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

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

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

```
fuchs_gesehen <- TRUE
```

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

```
typeof(fuchs_gesehen)
```

265 ## [1] "logical"

266 `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

267 `## [1] 2`

FALSE + FALSE

268 `## [1] 0`

TRUE + FALSE

269 `## [1] 1`

2.3 Funktionen

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

sqrt(a)

274 `## [1] 3.162278`

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

sqrt(x = a)

287 `## [1] 3.162278`

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

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

300 2.4 Datenstrukturen

301 Verfolgen wir das Beispiel mit den Fotofallen etwas weiter. Es handelt sich um ein systematisches Monitoring.
 302 D. h., es wurde nicht nur eine Fotofalle ausgebracht, sondern insgesamt 15 Stück. Dieser Umstand erfordert
 303 komplexere Objekte. Nachfolgend sind die Anzahl Rehfotos für jede der 15 Fotofallen aufgeführt: 132, 79,
 304 129, 91, 138, 144, 55, 103, 139, 105, 96, 146, 95, 118, 107.

305 Die Frage, die sich jetzt stellt, ist: Wie kann man diese Daten sinnvoll organisieren? Zusätzlich zur Anzahl
 306 der fotografierten Rehe soll jede Fotofalle eine eindeutige ID haben (`Kamera_1`, ..., `Kamera_15`) und wir
 307 wissen, dass jeweils 5 Fotofallen in drei unterschiedlichen Revieren aufgestellt waren (Fotofalle 1 bis 5 in
 308 Revier A, Fotofalle 6 bis 10 in Revier B und Fotofalle 11 bis 15 in Revier C). Wir könnten für jede Kamera
 309 und jeden Wert ein einzelnes Objekt mit je einem Wert erstellen. Dass das zu Aufwändig wäre, leuchtet
 310 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.
```

311 Wenn wir so vorgehen würden, hätten wir 45 Objekte mit je einem Eintrag. Dieser Ansatz und führt schnell
 312 zu einem unübersichtlichen *Workspace*³. Wir werden im Verlauf sinnvollere Objekte (Vektoren oder Data
 313 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.

314

315 **Aufgabe 2: Variablen**

316

317 Verwenden Sie die folgenden Daten

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

318 und berechnen sie:

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

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

323 **2.5 Funktionen**

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

```
sqrt(a)
```

327 `## [1] 1.414214`

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

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

```
sqrt(x = a)  
337 ## [1] 1.414214
```

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

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

3 Vektoren

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

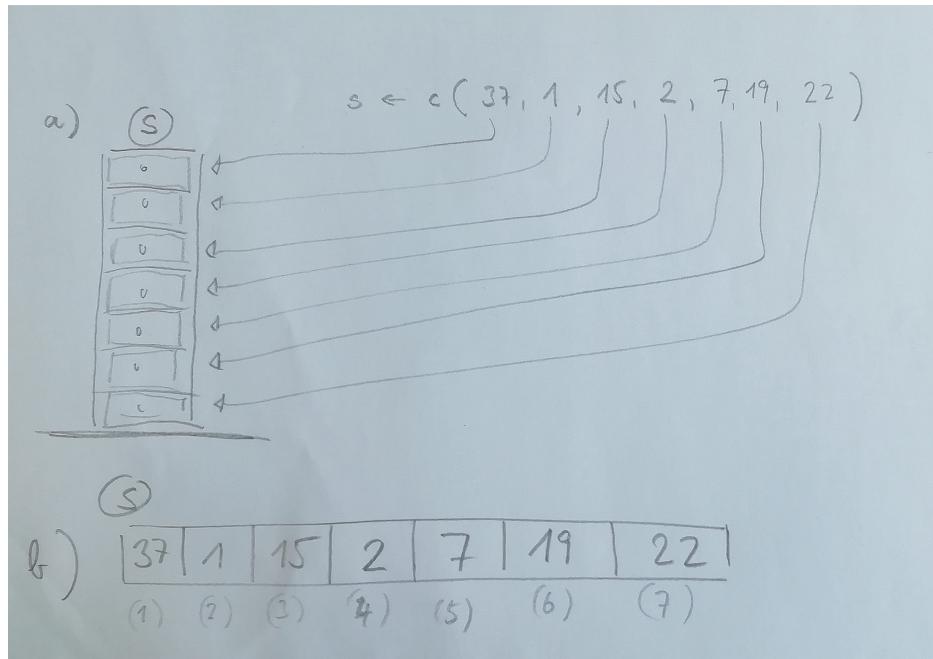


Abbildung 3: Schematische Darstellung eines Vektors in R.

356 Sie können sich Vektoren wie einen Schubladenschrank vorstellen (siehe auch Abbildung 3). Wichtig ist dabei,
357 dass man in jede Schublade immer nur ein Element verstauen kann und alle Elemente im Schublagenschrank
358 den gleichen Datentyp haben müssen. Etwas allgemeiner gesprochen heißt das, dass alle Elemente eines
359 Vektors vom gleichen Datentyp sein müssen.

360 Es gibt zahlreiche Funktionen zum Erstellen von Vektoren (einige davon werden wir im weiteren Verlauf des
361 Moduls kennenlernen). Die wohl wichtigste Funktion ist `c()`. Dabei steht `c` für *combine* oder *concatenate*.
362 Die Funktion `c()` fügt einzelne Elemente in einen Vektor zusammen (und zwar genau in der Reihenfolge wie
363 diese Elemente an `c()` übergeben werden). `c()` ist sozusagen die Funktion, die für Sie mehrere Elemente zu
364 einem Vektor zusammensetzt. `c()` erwartet folglich nur Elemente als Argumente.

365 Gehen wir nochmals zurück zu Abbildung 3, in der schematisch dargestellt wird, wie ein Vektor `s` mit 7
366 Elementen (in diesem Fall Zahlen) erstellt wird.

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

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

369 sehen:

```
s
```

370 ## [1] 37 1 15 2 7 19 22

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

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

```
s + 10
```

376 ## [1] 47 11 25 12 17 29 32

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

```
s * s
```

380 ## [1] 1369 1 225 4 49 361 484

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

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

384 ## [1] 1 2 3 4 5 6 7 8 9 10

```
(1 : 10)
```

385 ## [1] 1 2 3 4 5 6 7 8 9 10

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

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

387 ## [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.

388

Aufgabe 3: Vektoren erstellen

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

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

395 **3.1 Funktionen zum Arbeiten mit Vektoren**

396 Die Funktionen `head()` und `tail()` geben die ersten bzw. letzten `n` Elemente eines Vektors zurück. `n` hat
397 einen voreingestellten Wert von 6, dieser kann natürlich angepasst werden.

```
head(s)
```

398 ## [1] 37 1 15 2 7 19

```
head(s, n = 3)
```

399 ## [1] 37 1 15

```
tail(s, n = 2)
```

400 ## [1] 19 22

401 Die Funktion `length()` gibt die Länge eines Vektors wieder.

```
length(s)
```

402 ## [1] 7

403 Der Typ der Elemente eines Vektors kann mit der Funktion `class` abgefragt werden:

```
class(s)
```

404 ## [1] "numeric"

405 Die eindeutigen Elemente eines Vektors können mit der Funktion `unique()` abgefragt werden.

unique(s)

```
406 ## [1] 37 1 15 2 7 19 22
```

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

table(s)

```
408 ## s
409 ## 1 2 7 15 19 22 37
410 ## 1 1 1 1 1 1
```

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

rev(s)

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

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

sort(s)

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

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

rep(s, times = 2)

```
417 ## [1] 37 1 15 2 7 19 22 37 1 15 2 7 19 22
```

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

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

```
420 ## [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)

421 ## [1] 1 1 2 2 3 3 4 4
```

422

423 **Aufgabe 4: Arbeiten mit Vektoren**

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

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

426 Diese wurden immer abwechselnd mit zwei unterschiedlichen Messgeräten durchgeführt wurden.

427 Erstellen Sie einen Vektor von der Länge 8 mit den Einträgen, die immer abwechselnd G1 und G2 sind und
428 für die zwei Geräte stehen.

429 **3.2 Statistische Funktionen**

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

```
mean(s)
```

433 ## [1] 14.71429

```
median(s)
```

434 ## [1] 15

```
sd(s)
```

435 ## [1] 12.76341

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

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

439 ## [1] 1

```
sample(s, size = 3) # 2 Elemente
440 ## [1] 15 7 22
```

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

443 3.3 Beispiel Fotofallen

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

447 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)
```

448 Als zweites erstellen wir einen Vektor mit den IDs. Zur Erinnerung, diese sollten die Werte `Kamera_1` bis
 449 `Kamera_15` haben. Ein erster Ansatz könnte sein, dass wir einfach 15 Fotofallen schreiben und dann die
 450 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")
)
```

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

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

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

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

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

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

```

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

461 ## [1] "Kamera_1"  "Kamera_2"  "Kamera_3"  "Kamera_4"  "Kamera_5"  "Kamera_6"
462 ## [7] "Kamera_7"  "Kamera_8"  "Kamera_9"  "Kamera_10" "Kamera_11" "Kamera_12"
463 ## [13] "Kamera_13" "Kamera_14" "Kamera_15"

```

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

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

```

466 ## [1] "Revier A" "Revier B" "Revier C" "Revier A" "Revier B" "Revier C"
467 ## [7] "Revier A" "Revier B" "Revier C" "Revier A" "Revier B" "Revier C"
468 ## [13] "Revier A" "Revier B" "Revier C"

```

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

```

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

```

```

471 ## [1] "Revier A" "Revier A" "Revier A" "Revier A" "Revier A" "Revier B"
472 ## [7] "Revier B" "Revier B" "Revier B" "Revier B" "Revier C" "Revier C"
473 ## [13] "Revier C" "Revier C" "Revier C"

```

474

475 *Aufgabe 5: Statistische Funktionen*

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

478 2. Erstellen Sie die folgende Konsolenausgabe:

```
479 ## [1] "Die mittlere Anzahl von Rehfotos beträgt 171.8 Rehe pro Standort."
```

480 3.4 Arbeiten mit logischen Werten

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

- 483 • Gleichheit (`==`)

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

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

```
anzahl_rehe > 100
```

490 ## [1] TRUE FALSE TRUE FALSE TRUE TRUE FALSE TRUE TRUE TRUE FALSE TRUE
 491 ## [13] FALSE TRUE TRUE

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

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

494 ## [1] FALSE FALSE FALSE FALSE FALSE TRUE TRUE TRUE TRUE TRUE FALSE FALSE
 495 ## [13] FALSE FALSE FALSE

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

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

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

502 ## [1] FALSE FALSE FALSE FALSE FALSE TRUE FALSE TRUE TRUE TRUE FALSE FALSE
 503 ## [13] FALSE FALSE FALSE

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

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

507 ## [1] TRUE FALSE TRUE FALSE TRUE TRUE TRUE TRUE TRUE TRUE FALSE TRUE
 508 ## [13] FALSE TRUE TRUE

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

511

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

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

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

523 **3.5 Zugreifen auf Elemente eines Vektors (=Untermengen)**

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

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

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

```
anzahl_rehe[2]
```

537 ## [1] 79

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

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

539 ## [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.
```

540 ## [1] 132 79 129 91 138

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

543

544 Aufgabe 7: Zugreifen auf Vektorelemente

546 Erstellen Sie einen neuen Vektor `bhd`

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

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

550

551 Alternativ könnte das gleiche Ergebnis mit einem logischen Vektor erreicht werden. Für eine bessere Über-
 552 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]
```

553 ## [1] 132 79 129 91 138

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

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

557 ## [1] 132 79 129 91 138

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

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

560 ## [1] 132 79 129 91 138

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

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

563 ## [1] 113.8

564

565 Aufgabe 8: logische Werte

567 Verwenden Sie nochmals den Vektor mit den Anzahl Rehen die an unterschiedlichen Fotofallenstandorten
568 fotografiert wurden.

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

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

572 3.6 Der %in%-Operator

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

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

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

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

577 ## [1] "FI" "FI"

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

```
578 ## [1] "FI" "FI"
```

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

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

```
581 ## [1] "FI" "BU" "BU" "FI"
```

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

```
messungen_arten %in% arten
```

```
586 ## [1] TRUE TRUE TRUE FALSE FALSE FALSE TRUE FALSE FALSE
```

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

```
587 ## [1] "FI" "BU" "BU" "FI"
```

588

589 **Aufgabe 9: Auswählen von Elementen in einem Vektor (%in%)**
590

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

```
LETTERS
```

```
592 ## [1] "A" "B" "C" "D" "E" "F" "G" "H" "I" "J" "K" "L" "M" "N" "O" "P" "Q" "R" "S"
```

```
593 ## [20] "T" "U" "V" "W" "X" "Y" "Z"
```

594 Wählen Sie aus `LETTERS` nur die Vokale aus.

595 4 Faktoren (factors)

596 «««< HEAD R besitzt einen besonderen Datentyp – *Faktoren* (engl. factors) – zum speichern von kate-
 597 gorialen Kovariaten (z.B. Baumart, Augenfarbe oder Automarke). Faktoren erlauben es Daten vom Typ
 598 `character` effizienter abzuspeichern. Dabei wird jeder eindeutiger Wert (=Level) mit einer Zahl codiert
 599 und dann werden nur diese Zahlen zusammen mit einer Tabelle zum Nachschauen (engl. look-up table) der
 600 Werte gespeichert (siehe dazu auch [McNamara and Horton 2018](#)). ===== R besitzt einen besonderen
 601 Datentyp – *Faktoren* (engl. factors) – zum speichern von diskreten Kovariaten (z.B. Baumart, Augenfarbe
 602 oder Automarke). Faktoren erlauben es Daten vom Typ `character` effizienter abzuspeichern. Dabei wird
 603 jeder eindeutiger Wert (=Level) mit einer Zahl codiert und dann werden nur diese Zahlen zusammen mit
 604 einer Tabelle zum Nachschauen der Werte gespeichert (siehe dazu auch [McNamara and Horton 2018](#)). Fak-
 605 toren haben im Gegensatz zu Zeichenketten zusätzliche Eigenschaften. Man kann sie z. B. sortieren. »»»>
 606 629c903f27f5c5db526dfd1588596c4a4b00635b

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

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

609 ## [1] FI BU FI EI EI FI FI
610 ## Levels: BU EI FI

611 «««< HEAD Ohne weitere Spezifikation werden für die *Levels* die Werte alphabetisch übernommen (das
 612 kann später z.B. beim Erstellen von Abbildungen wichtig sein). Für eine benutzerdefinierte Anordnungs
 613 der Levels, kann das Argument `levels` verwendet werden. ===== Ohne weitere Spezifikation wer-
 614 den die Werte *Levels* alphabetisch angeordnet (das kann später z. B. beim Erstellen von Abbildungen
 615 wichtig sein), dies kann jedoch durch die Verwendung des Arguments `levels` gesteuert werden. »»»>
 616 629c903f27f5c5db526dfd1588596c4a4b00635b

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

617 ## [1] FI BU FI EI EI FI FI
618 ## Levels: FI BU EI

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

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

621 ## [1] Fichte Buche Fichte Eiche Eiche Fichte Fichte
622 ## Levels: Fichte Buche Eiche

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

```
levels(af)
```

625 ## [1] "Fichte" "Buche" "Eiche"

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

626 ## [1] Fi Bu Fi Ei Ei Fi Fi

627 ## Levels: Fi Bu Ei

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

```
af
```

630 ## [1] Fi Bu Fi Ei Ei Fi Fi

631 ## Levels: Fi Bu Ei

```
relevel(af, "Bu")
```

632 ## [1] Fi Bu Fi Ei Ei Fi Fi

633 ## Levels: Bu Fi Ei

634 Mit der Funktion `as.character()` kann ein Faktor wieder als Variable vom Typ `character` dargestellt
 635 werden.

```
as.character(af)
```

636 ## [1] "Fi" "Bu" "Fi" "Ei" "Ei" "Fi" "Fi"

637 Achtung mit der Funktion `as.numeric()` erhält man die interne Kodierung von Faktoren.

```
af
```

638 ## [1] Fi Bu Fi Ei Ei Fi Fi

639 ## Levels: Fi Bu Ei

```
as.numeric(af)
```

640 ## [1] 1 2 1 3 3 1 1

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

643

644 **Aufgabe 10: Faktoren**

645

- 646 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")
```

648 **4.1 Das Paket **forcats****

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

- 651 1. Die Anordnung von Levels anzupassen.
652 2. Levels zusammenzufassen oder zu entfernen.
653 3. Labels zu ändern.

654 **4.1.1 Anpassen der Anordnung von Faktoren**

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

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

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

```
factor(a)
```

- 657 ## [1] FI BU FI EI EI FI FI
658 ## Levels: BU EI FI

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

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

- 660 ## [1] FI BU FI EI EI FI FI
661 ## Levels: FI BU EI

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

```
fct_rev(f1)
```

663 ## [1] FI BU FI EI EI FI FI
664 ## Levels: EI BU FI

665 nach der Häufigkeit zu sortieren oder

```
fct_infreq(f1)
```

666 ## [1] FI BU FI EI EI FI FI
667 ## Levels: FI EI BU

668 eine zufällige Sortierung.

```
fct_shuffle(f1)
```

669 ## [1] FI BU FI EI EI FI FI
670 ## Levels: EI FI BU

671 5 Spezielle Einträge

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

- 673 • fehlenden Einträgen NA,
- 674 • leeren Einträgen NULL,
- 675 • undefinierten Einträgen NaN (Not a Number) oder
- 676 • unendlichen Zahlen (Inf).

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

678 5.1 NA

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

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

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

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

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

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

```
is.na(na1)

## [1] FALSE TRUE FALSE

na.omit(na1)

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

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

```
na2 < 3
```

696 ## [1] FALSE FALSE NA

```
1 + NA
```

697 ## [1] NA

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

```
mean(na2)
```

701 ## [1] NA

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

702 ## [1] 4.5

703 5.2 NULL

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

708 5.3 Inf

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

```
10^309
```

711 ## [1] Inf

```
2 * Inf
```

712 ## [1] Inf

1 + Inf

713 ## [1] Inf

3 / 0

714 ## [1] Inf

-3 / 0

715 ## [1] -Inf

3 / Inf

716 ## [1] 0

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

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

719 ## [1] TRUE FALSE FALSE TRUE FALSE

`is.finite(inf1)`

720 ## [1] FALSE TRUE TRUE FALSE TRUE

`inf1 < 3`

721 ## [1] FALSE TRUE FALSE TRUE FALSE

722

723 Aufgabe 11: Vektoren mit speziellen Einträgen

725 Verwenden Sie den Vektor

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

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

728 Verwenden Sie den Vektor

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

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

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

734 Berechnen Sie den arithmetischen Mittelwert von `foo`.

735 6 data.frames oder Tabellen

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

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

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

	ID	anzahl_rehe	revier
749 ## 1	Kamera_1	132	Revier A
750 ## 2	Kamera_2	79	Revier A
751 ## 3	Kamera_3	129	Revier A
752 ## 4	Kamera_4	91	Revier A
753 ## 5	Kamera_5	138	Revier A
754 ## 6	Kamera_6	144	Revier B
755 ## 7	Kamera_7	55	Revier B
756 ## 8	Kamera_8	103	Revier B
757 ## 9	Kamera_9	139	Revier B
758 ## 10	Kamera_10	105	Revier B
759 ## 11	Kamera_11	96	Revier C
760 ## 12	Kamera_12	146	Revier C
761 ## 13	Kamera_13	95	Revier C
762 ## 14	Kamera_14	118	Revier C
763 ## 15	Kamera_15	107	Revier C

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

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

770 6.1 Wichtige Funktionen zum Arbeiten mit `data.frames`

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

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

```
773 ##           ID anzahl_rehe   revier
774 ## 1 Kamera_1            132 Revier A
775 ## 2 Kamera_2            79 Revier A
```

776 Oder für die letzten 2 Beobachtungen.

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

```
777 ##           ID anzahl_rehe   revier
778 ## 14 Kamera_14          118 Revier C
779 ## 15 Kamera_15          107 Revier C
```

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

```
nrow(monitoring)
```

```
781 ## [1] 15
```

```
ncol(monitoring)
```

```
782 ## [1] 3
```

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

```
str(monitoring)
```

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

789

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

792 Stellen Sie sich vor, Sie machen eine kleine Umfrage, in der Sie fünf Menschen nach ihrem Studienfach, Semester und Alter befragen. Erstellen Sie einen `data.frame` mit dem Namen `umfrage1` für diese Informationen
793 und fragen Sie entweder fünf Mitstudierende oder erfinden Sie die Daten einfach.
794

795 **6.2 Zugreifen auf Elemente eines `data.frame`**

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

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

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

804 `## [1] 91`

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

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

808 `## [1] 91`

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

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

811 `## [1] 132 79 129 91 138`

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

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

```
814 ##      anzahl_rehe    revier
815 ## 1          132 Revier A
816 ## 2          79 Revier A
817 ## 3          129 Revier A
818 ## 4          91 Revier A
819 ## 5          138 Revier A
```

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

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

```
822 ##           ID anzahl_rehe    revier
823 ## 1 Kamera_1          132 Revier A
824 ## 2 Kamera_2          79 Revier A
825 ## 3 Kamera_3          129 Revier A
826 ## 4 Kamera_4          91 Revier A
827 ## 5 Kamera_5          138 Revier A
```

828

Aufgabe 13: Abfragen von Werten

831 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)
)
```

- 832 • Wählen Sie nur die ersten drei Zeilen aus und die erste und zweite Spalte aus.
833 • Wählen Sie alle Zeilen und die erste und dritte Spalte aus.
834 • Wählen Sie alle Spalten und die erste, dritte und vierte Zeile aus.

835

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

- 838 1. über das \$-Zeichen direkt die Spalten ansprechen.

```
monitoring$anzahl_rehe
```

```
839 ## [1] 132 79 129 91 138 144 55 103 139 105 96 146 95 118 107
```

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

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

```
841 ## [1] 132 79 129 91 138 144 55 103 139 105 96 146 95 118 107
```

842 3. Alle Zeilen und die Spalte explizit angeben.

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

```
843 ## [1] 132 79 129 91 138 144 55 103 139 105 96 146 95 118 107
```

844 Anmerkung zu 3), der Ausdruck `1:nrow(monitoring)` ergibt einen Vektor mit den Zahlen 1 bis 15, da
 845 `nrow(monitoring) = 15` ist. So eine Schreibweise ist zu empfehlen, wenn die Dimension des Vektors variabel
 846 ist.

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

```
monitoring$anzahl_rehe > 100
```

```
850 ## [1] TRUE FALSE TRUE FALSE TRUE TRUE FALSE TRUE TRUE TRUE FALSE TRUE  

  851 ## [13] FALSE TRUE TRUE
```

852 Das Ergebnis ist ein Vektor mit 15 Elementen. Hat eine Fotofalle mehr als 100 Rehfotos gemacht ist das
 853 entsprechende Element des Vektors `TRUE` ansonsten `FALSE`. In dem `data.frame monitoring` steht in jeder
 854 Zeile eine Beobachtung (also eine Fotofalle). Nun wollen wir genau diese Fotofallen haben, die mehr als 100
 855 Rehfotos gemacht haben.

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

```
856 ##          ID anzahl_rehe    revier  

  857 ## 1   Kamera_1        132 Revier A  

  858 ## 3   Kamera_3        129 Revier A  

  859 ## 5   Kamera_5        138 Revier A  

  860 ## 6   Kamera_6        144 Revier B  

  861 ## 8   Kamera_8        103 Revier B  

  862 ## 9   Kamera_9        139 Revier B  

  863 ## 10  Kamera_10       105 Revier B  

  864 ## 12  Kamera_12       146 Revier C  

  865 ## 14  Kamera_14       118 Revier C  

  866 ## 15  Kamera_15       107 Revier C
```

867

868 **Aufgabe 14: Abfragen von Werten 2**

869

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

- 871 • Alle Spalten für Studierende die Forstwissenschaften studieren.
872 • Alle Spalten für Studierende die Chemie oder Physik studieren.
873 • Die Spalte `fach` und `semester` für Studierende die 22 oder älter sind.

874 7 Schreiben und lesen von Daten

875 7.1 Textdateien

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

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

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

891 Die Datei `fotofallen.csv` finden Sie auf StudIP und kann einfach heruntergeladen werden. Sie können sich
 892 die Datei mit jedem Texteditor oder auch mit Excel oder Libre Office ansehen. Die Datei kann mit dem fol-
 893 genden Befehl in R eingelesen werden. Hier wurde die Datei in einem R Studio-Projekt in ein Unterverzeichnis
 894 `data` abgelegt.

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

```
895 ##      ID anzahl_rehe   revier  
896 ## 1 Kamera_1        132 Revier A  
897 ## 2 Kamera_2        79 Revier A  
898 ## 3 Kamera_3        129 Revier A  
899 ## 4 Kamera_4        91 Revier A  
900 ## 5 Kamera_5        138 Revier A  
901 ## 6 Kamera_6        144 Revier B
```

902 Es gibt viele Varianten der Funktion `read.table()`. Beispielsweise hat die Funktion `read.csv()` bereits die
 903 Argumente `sep = ','` und `header = TRUE` gesetzt. Die Funktion `read.csv2()` hat das Argument `sep =`
 904 `';'` gesetzt. Sie müssen natürlich darauf achten, dass sie die csv Dateien mit den gleichen Spezifikationen
 905 einlesen, mit denen sie gespeichert wurde. Siehe dazu auch die Hilfeseite von `read.table()`.

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

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.

907

908 **Aufgabe 15: Lesen und Schreiben von Datein**
909

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

913 8 Erstellen von Abbildungen

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

918 8.1 Base Plot

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

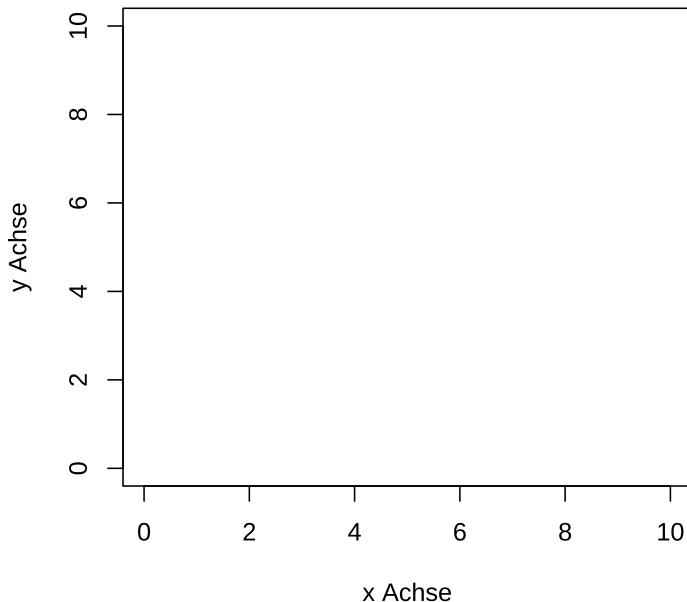
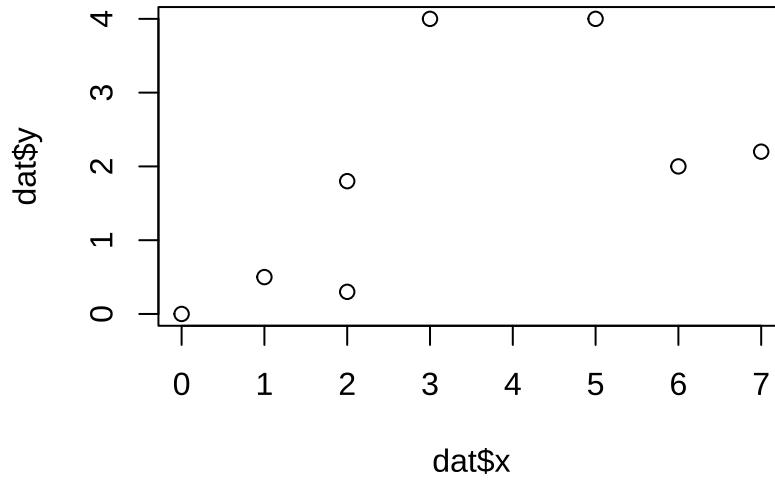


Abbildung 4: Beispiel einer leeren Grafikschnittstelle.

925 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)
```

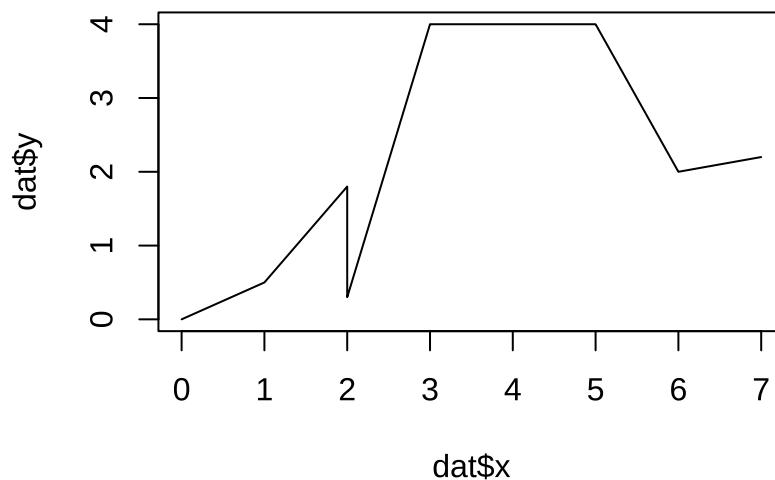
```
)  
  
plot(dat$x, dat$y, type = "p")
```



926

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

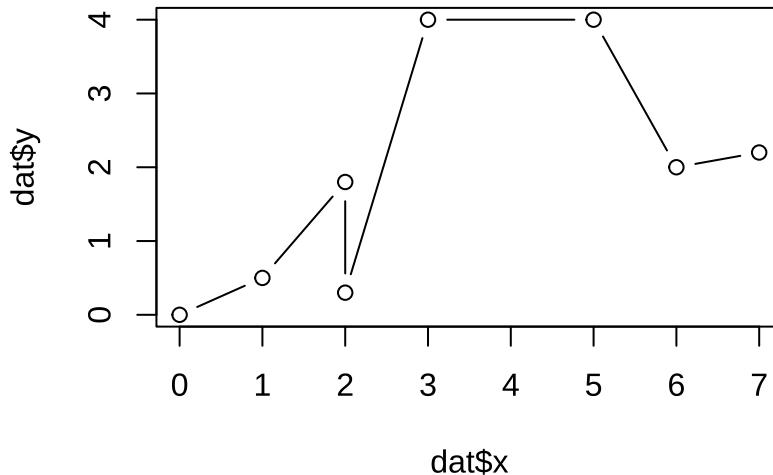
```
plot(dat$x, dat$y, type = "l")
```



929

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

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



931

932 darstellen.

933

934 **Aufgabe 16: Base Plot 1**

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

938

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

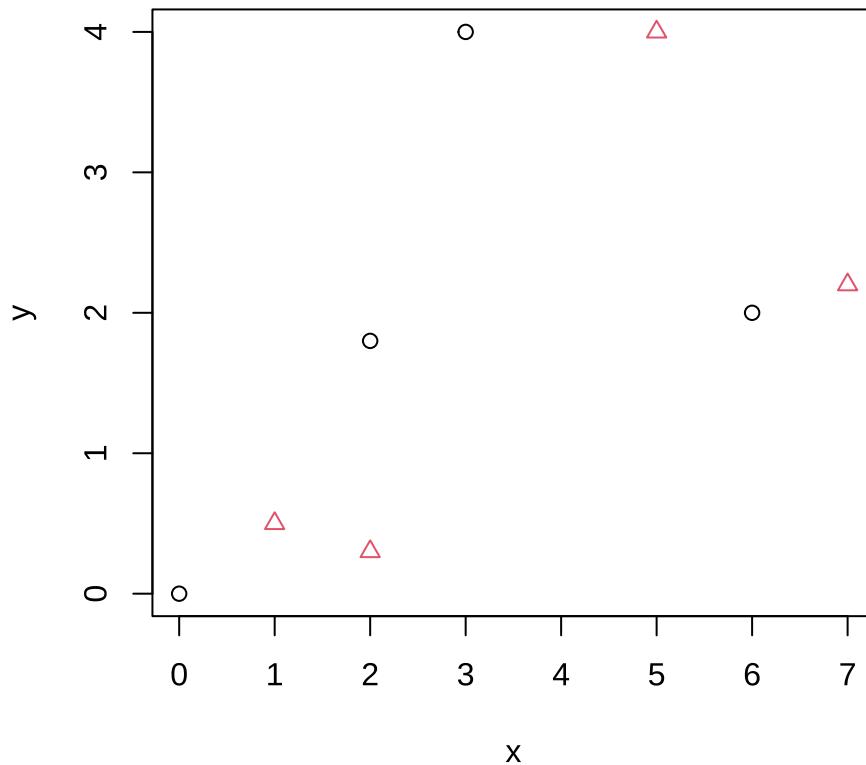
- 943 • `type` - Diagrammtyp
- 944 • `col` - Farbe
- 945 • `main` - Titel
- 946 • `sub` - Untertitel
- 947 • `pch` - Punktsymbol

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

955 Sehen Sie sich die Hilfeseiten `?plot.default()` oder `?par()` an für weitere Informationen. Dort finden Sie
 956 auch eine vollständige Liste der Befehle. Einige Argumente können als Vektor übergeben werden. Hier z. B.
 957 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")
```



958

959

960 **Aufgabe 17: Anpassen von Plots**

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

- 963 • Beschriften Sie die x- und y-Achse sinnvoll.
964 • Fügen Sie eine Überschrift hinzu.
965 • Wählen Sie ein anderes Symbol.
966 • Stellen Sie die Symbole in rot dar.

967

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

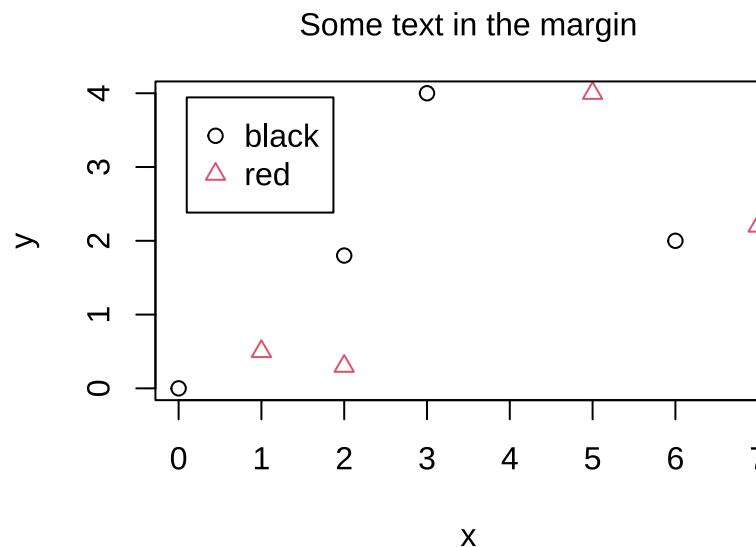
969 Die wichtigsten Funktionen sind

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

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

982 Legenden können im base plot nur als Low-Level Funktion hinzugefügt werden. Mit der Funktion `legend`
983 werden die Einträge und die entsprechenden Punktsymbole oder Linientypen dargestellt. In der folgenden
984 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")
```



985

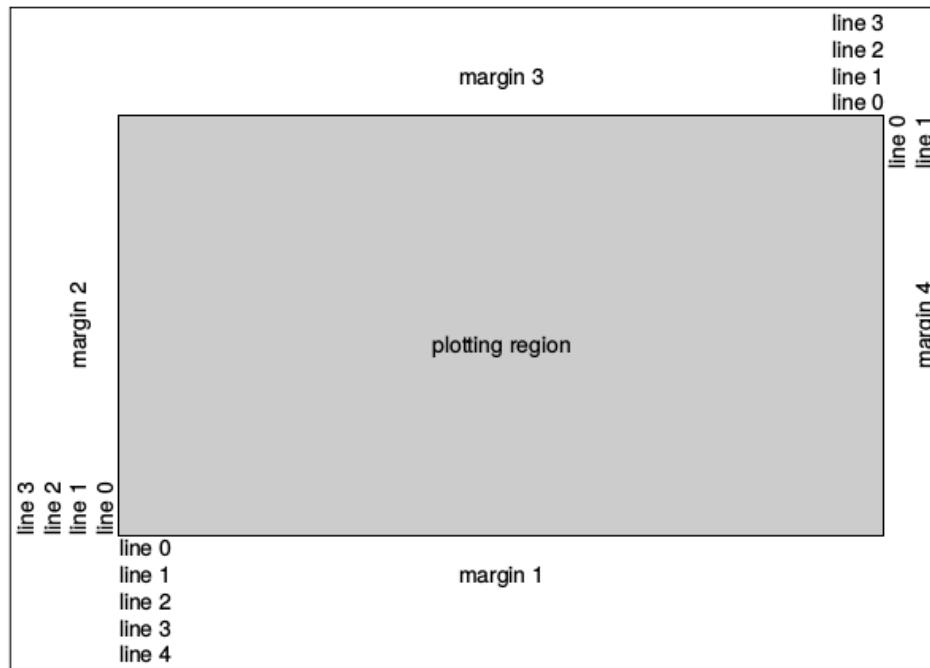


Abbildung 5: Grafikregionen eines base plots in R.

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

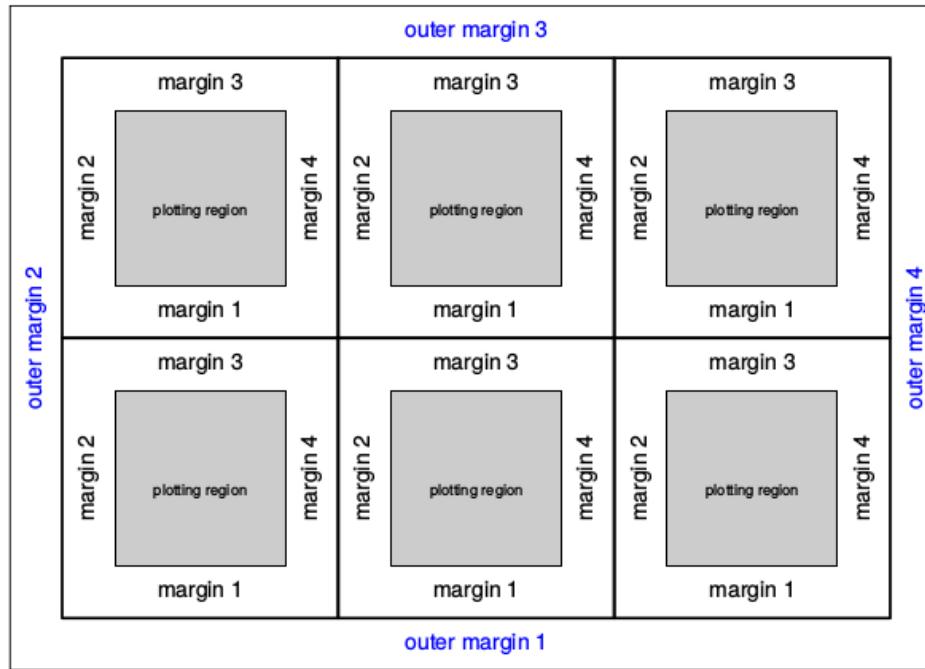


Abbildung 6: Schematischer Aufbau mehrere Diagramme in einem plot am Beispiel einer 3 x 2 Grafik.

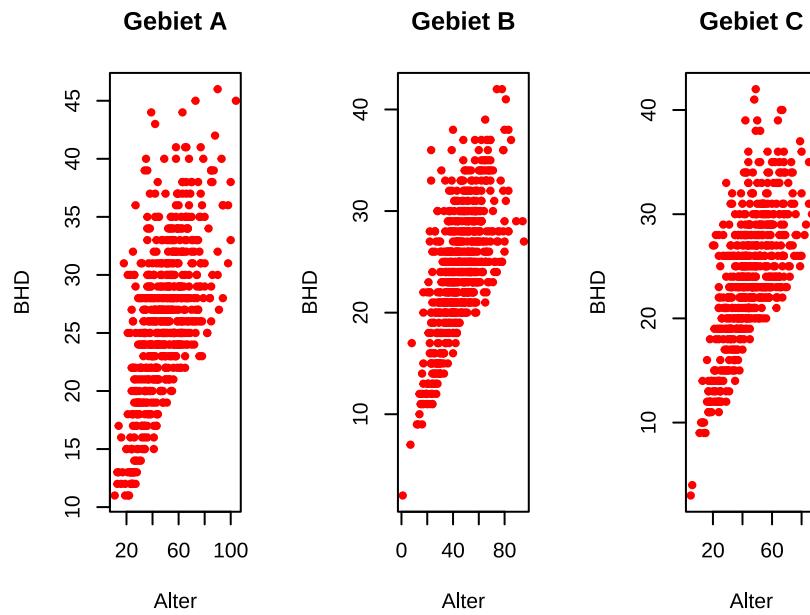
8.1.1 Mehrere Panels

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

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

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")
```



995

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

998 8.1.2 Speichern von Abbildungen

999 Wenn nicht anders angegeben, wird die Abbildung zunächst nur in der RStudio Grafik Schnittstelle abgebildet
1000 (Pane 4). Von dort aus kann die Abbildung exportiert werden. Es bietet sich jedoch an das Speichern der
1001 Abbildung direkt im Code zu programmieren. Mögliche Formate die Abbildung als Vektorgrafik zu speichern
1002 sind

- 1003 • `pdf()` oder
- 1004 • `postscript()`.

1005 Beispiele für Rastergrafiken sind

- 1006 • `png()`,
- 1007 • `bmp()` oder
- 1008 • `jpeg()`.

1009 Die Grafikschnittstelle ist dann Ihre “Leinwand”. Mit dem Befehl `dev.off()` trennen Sie die Verbindung
1010 zur Schnittstelle wieder. Ihre “Leinwand” wird also wieder geschlossen. So lange die Schnittstelle geöffnet ist
1011 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

```

1012 Achtung, wenn Sie die Funktion `dev.off()` nicht aufrufen, werden alle nachfolgenden Plots in die gleiche
 1013 Datei geschrieben. Falls Sie nach einem Versuch einen Plot zu speichern plötzlich keine weiteren Plots mehr
 1014 sehen, führen Sie einige Mal die Funktion `dev.off()` aus.

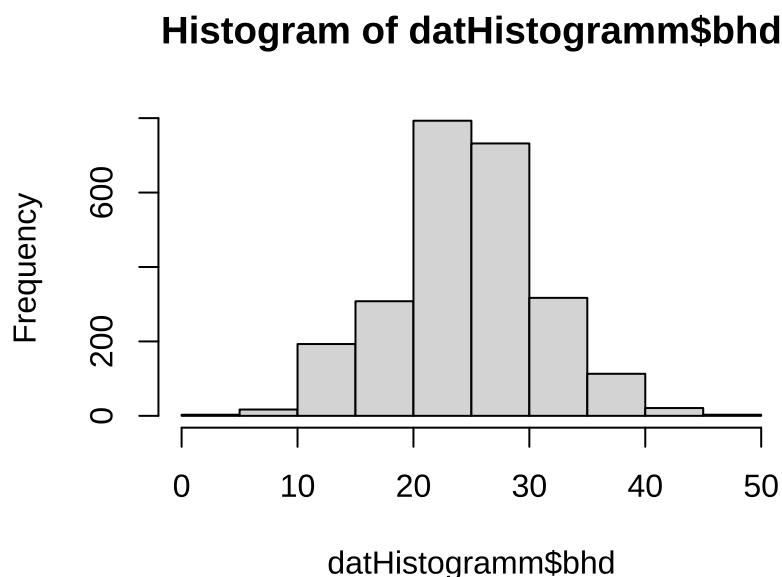
1015 8.2 Histogramme

1016 Neben den Streudiagrammen (*Scatterplots*, oder auch einfach x-y Diagramm) sind *Histogramme* in der
 1017 angewandten Datenanalyse ein weiterer wichtiger Abbildungstyp. An Histogrammen wird die Häufigkeit
 1018 von Beobachtungen nach Gruppen dargestellt. Sie sind deshalb so wichtig, weil man aus ihnen relevante
 1019 Informationen über die Verteilung der Daten ablesen kann und somit einiges über die Messungen erfährt,
 1020 das für die weitere Datenanalyse relevant sein kann. So werden auf einen Blick der Zusammenhang von
 1021 Beobachtungshäufigkeit und Streuung deutlich, sowie auch die Form der Verteilung und ihre Schiefe. Die
 1022 Interpretation werden wir bei den Boxplots noch weiter vertiefen.

```

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

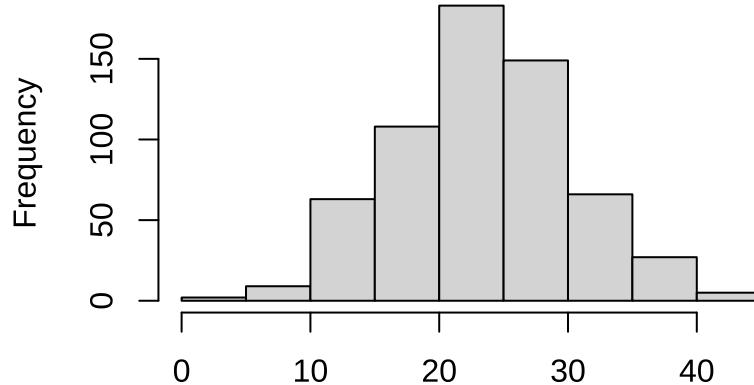
```



1023

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

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

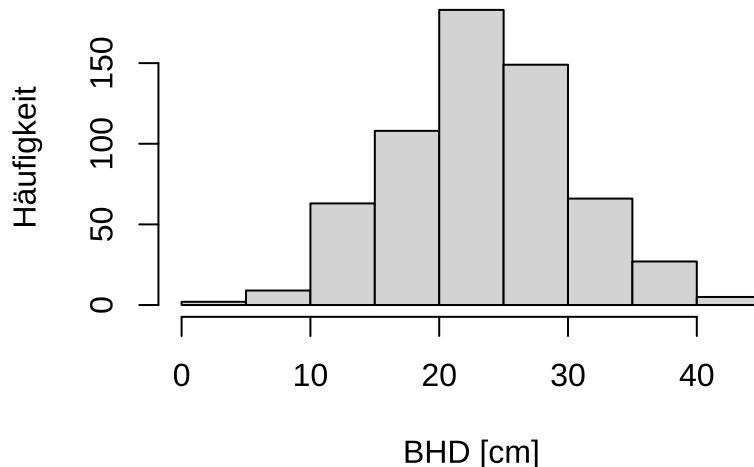


```
datHistogramm$bhd[datHistogramm$art == "EI"]
```

1024

```
# 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

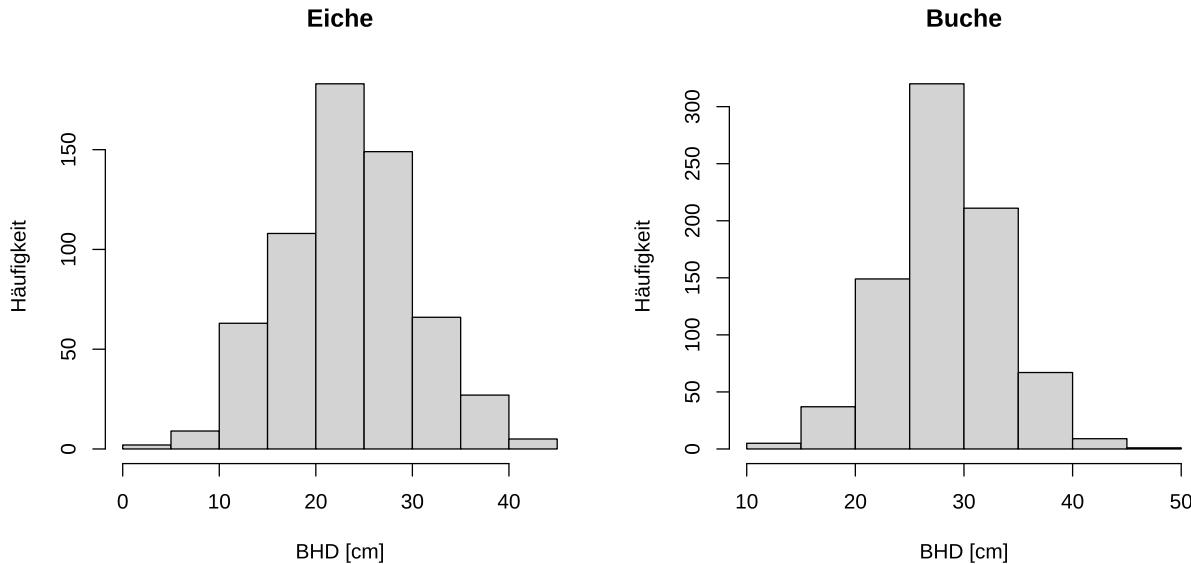


1025

1026 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")
```

1027



1028

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

1029 8.3 Boxplots

1030 Oft möchte man die Verteilung einer stetigen Variablen in Abhängigkeit einer diskreten Variable beschreiben
 1031 oder visualisieren. Ein Beispiel dafür wäre die BHD-Verteilung in Abhängigkeit der Baumarten. Eine häufige
 1032 Darstellungsform für solche Daten sind *Boxplots*. Sie stellen die durchschnittliche Ausprägung einer stetigen
 1033 Variable und ihre Schwankung kompakt dar.

1034 Boxplots bestehen aus drei Komponenten:

- 1035 1. Eine *Box*, die den Bereich zwischen 0.25 und 0.75 Percentil abdeckt, diese Distanz wird auch die IQR
 1036 (Interquartile Range), bezeichnet. Zusätzlich wird die Box durch den Median (als dicke horizontale Linie)
 1037 unterteilt.
- 1038 2. Einzelne Punkte Ausreißer. Als Ausreißer werden Punkte bezeichnet, die $> 1.5 \text{ IQR}$ vom unteren oder
 1039 oberen Ende der Box entfernt sind.

- 1040 3. Eine senkrechte Linie von jeder Seite der Box bis zum letzten “Nicht-Ausreißer-Punkt”. Also der letzte
 1041 Punkt, der $> 1.5IQR$ aber nicht > 0.75 bzw. < 0.25 Percentil ist. Diese Linie wird auch als *Whisker*
 1042 bezeichnet.

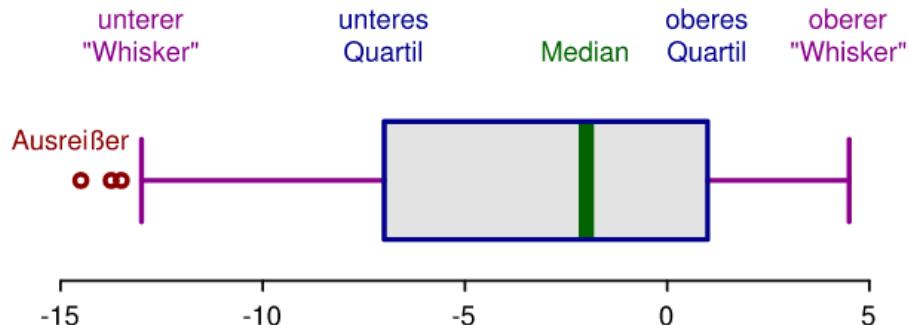
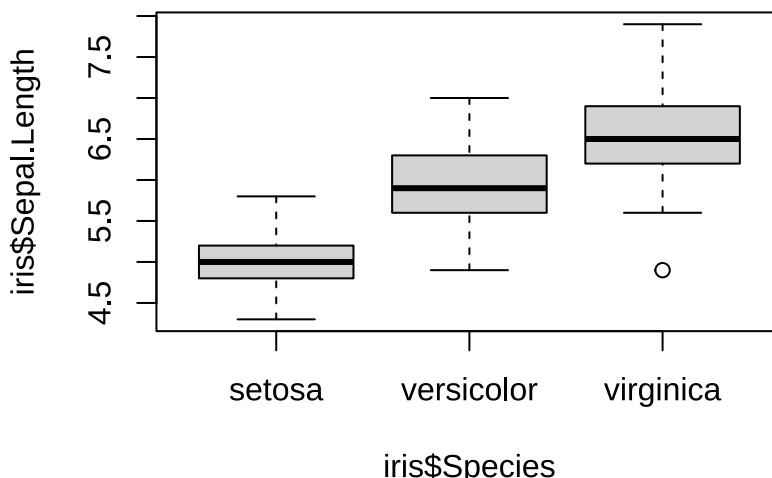


Abbildung 7: Schematische Darstellung eines Boxplots (Quelle: Von RobSeb - Eigenes Werk, CC BY-SA 3.0, <https://commons.wikimedia.org/w/index.php?curid=14697172>).

1043 Mit R kann mit der Funktion `boxplot()` ein Boxplot erstellt werden. Diese Funktion kann in zwei unter-
 1044 schiedlichen Ausprägungen verwendet werden.

- 1045 1. `boxplot(x)` erzeugt einen Boxplot für die Variable `x`.
 1046 2. `boxplot(x ~ y)` erzeugt einen oder mehrere Boxplots für `x` aber gruppiert nach `y`, dabei sollte `y` eine
 1047 kategoriale Variable sein. `x` und `y` können auch die Spaltennamen eines `data.frames` sein, dann muss
 1048 das `data.frame` mit dem Argument `data` zusätzlich übergeben werden.

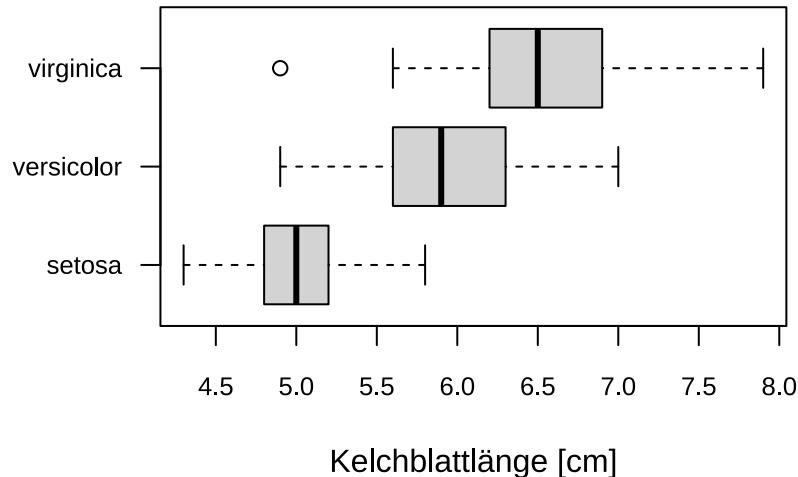
```
boxplot(iris$Sepal.Length ~ iris$Species)
```



1049

1050 Etwas eleganter ist es wenn wir das Argument `data` verwenden und den Plot etwas anpassen. Diese Schreib-
 1051 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  
)
```



1052

1053

1054 Aufgabe 18: Boxplots

1055

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

1060 Erstellen Sie einen Plot mit 3 Subplots, jeweils mit einem Boxplot für die ersten drei Studiengebiete, in dem
 1061 der BHD für jede Baumart dargestellt wird.

1062 8.4 ggplot2: Eine Alternative für Abbildungen

1063 `ggplot2` ist ein alternatives Plotting-System in *R*. Sie können mit `ggplot2` also grundsätzlich Abbildungen
 1064 mit dem selben Inhalt erstellen, wie mit Base Plots. Die Syntax und die optische Darstellung unterscheiden
 1065 sich jedoch grundsätzlich. `ggplot2` basiert auf den *grammar of graphics* von Leland Wilkinson. Die Idee ist,

1066 alle nötigen Informationen der Abbildung miteinander zu verknüpfen. *ggplot2* ist also diametral zu Base
 1067 Plots. Während Sie in base plot alle Elemente selbst definieren, ist die Idee von *ggplot2*, dass Sie nur die
 1068 Daten und den Diagrammtypen übergeben und die Funktion selbst eine schöne Abbildung erstellt. Selbstver-
 1069 ständlich können Sie aber auch in *ggplot2* viele Einstellungen vornehmen. Im base plot sehen Abbildungen
 1070 zunächst sehr karg und etwas unfertig aus. Sie müssen viele Einstellungen vornehmen, um eine publizierfä-
 1071 hige Grafik zu produzieren. In *ggplot2* sollen auch die einfachste Abbildungen schon ästhetisch sein. Mit
 1072 diesen gebündelten Informationen kann *ggplot2* die Abbildung automatisch verschönern. So werden bspw.
 1073 die Legenden automatisch erzeugt und auch die Formatierungen automatisch an die Datenlage angepasst.
 1074 *ggplot2* nimmt der*dem Entwickler*in also Arbeit ab. Dadurch sind die Abbildungen schon ohne viel Nach-
 1075 arbeit schick. Nachteil ist, dass der*dem Entwickler*in weniger Möglichkeiten zur Einstellung zur Verfügung
 1076 stehen und nuterspezifische Sonderwünsche somit schwerer umsetzbar sind. Sehen Sie sich das *Cheatsheet* zu
 1077 *ggplot2* an. Es ist in RStudio unter [Help > Cheatsheets](#) zu finden.

1078 Bei *ggplot2* sind Anweisungen zu den Daten und Anweisungen zur Darstellung voneinander getrennt. Die
 1079 Daten werden in den Ästhetikbefehl übergeben und dort klassifiziert. Dann folgen die Darstellungsanweisun-
 1080 gen. Ähnlich wie bei Base Plots, werden die Grafikelemente ebenenweise nacheinander programmiert, jedoch
 1081 mit einem `+` verbunden. Und hier liegt der wesentliche Unterschied zu Base Plots. Durch die `+` werden die
 1082 Ebenen zu einem Befehl verbunden und damit gleichzeitig erstellt.

1083 Die Erweiterung wird zunächst geladen⁷. Wir laden außerdem den Datensatz `iris`. Der Datensatz ist in R
 1084 fest integriert. Siehe `?iris` für mehr Informationen.

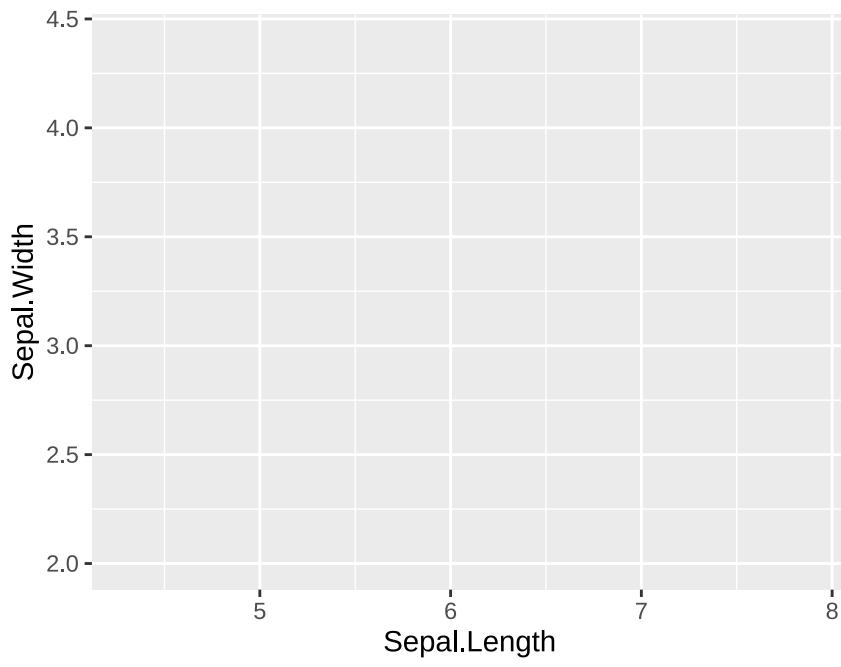
```
library(ggplot2)
head(iris)
```

```
1085 ##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
1086 ## 1          5.1       3.5      1.4       0.2  setosa
1087 ## 2          4.9       3.0      1.4       0.2  setosa
1088 ## 3          4.7       3.2      1.3       0.2  setosa
1089 ## 4          4.6       3.1      1.5       0.2  setosa
1090 ## 5          5.0       3.6      1.4       0.2  setosa
1091 ## 6          5.4       3.9      1.7       0.4  setosa
```

1092 Die Ästhetik wird bspw. folgendermaßen definiert.

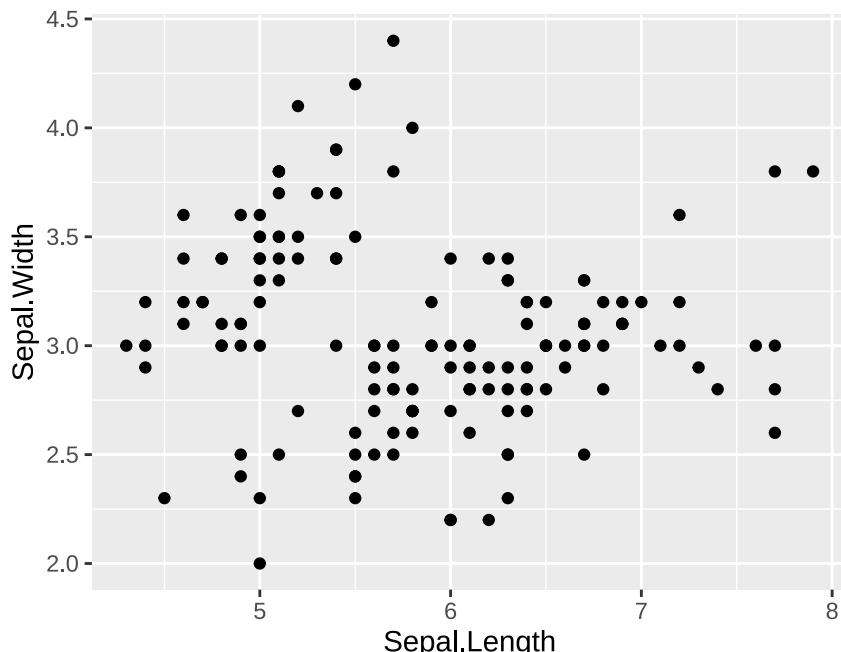
```
ggplot(iris, aes(x = Sepal.Length, y = Sepal.Width))
```

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.



1094 Dieser Befehl zeichnet noch keine Daten. Die Daten werden lediglich herangezogen, um einen leeren Plot für
 1095 die Daten zu erstellen. In dem Beispiel wird die Variable `Sepal.Length` aus dem `data.frame iris` als x und
 1096 `Sepal.Width` als y Variable definiert. Diese Informationen stehen den folgenden Layern nun zur Verfügung,
 1097 sodass nach den + nur noch x und y verwendet werden müssen. Um bspw. einen Scatterplot zu erstellen
 1098 wird ein `geom_point()` Layer hinzugefügt. x und y werden automatisch an `geom_point` übergeben. Weitere
 1099 Einstellung sind in diesem Beispiel nicht notwendig, wären jedoch möglich. Siehe `?geom_point()`.

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



1101

Aufgabe 19: Abbildungen mit *ggplot2*

1104 Verwenden Sie die Daten aus Aufgabe 16 und erstellen Sie einen Scatterplot mit *ggplot2* wie in Aufgabe 16.

1105

1106 Wir haben mit der Funktion `geom_point()` demm Plot eine Punktgeometrie hinzugefügt. Es gibt noch viele
1107 weitere Geometrien. Die wichtigsten sind:

- 1108 • `geom_line()` für eine Linie.
- 1109 • `geom_histogram()` um ein Histogramm zu erstellen.
- 1110 • `geom_boxplot()` um einen Boxplot zu erstellen.
- 1111 • `geom_bar()` um ein Säulendiagramm zu erstellen.

1112 Welche Geometrie die richtige ist, richtet nach dem Typ der darzustellenden Variablen. Beispielsweise bie-
1113 tet sich `geom_point()` an, wenn man zwei kontinuierliche Variable darstellen möchte. Wenn man hinge-
1114 gen die Verteilung von einer kontinuirlichen Variable darstellen möchte, dann bietet sich ein Histogramm
1115 (`geom_histogram()`) oder auch eine geschätzte Dichte (z.B. `geom_density()`) an.

1116

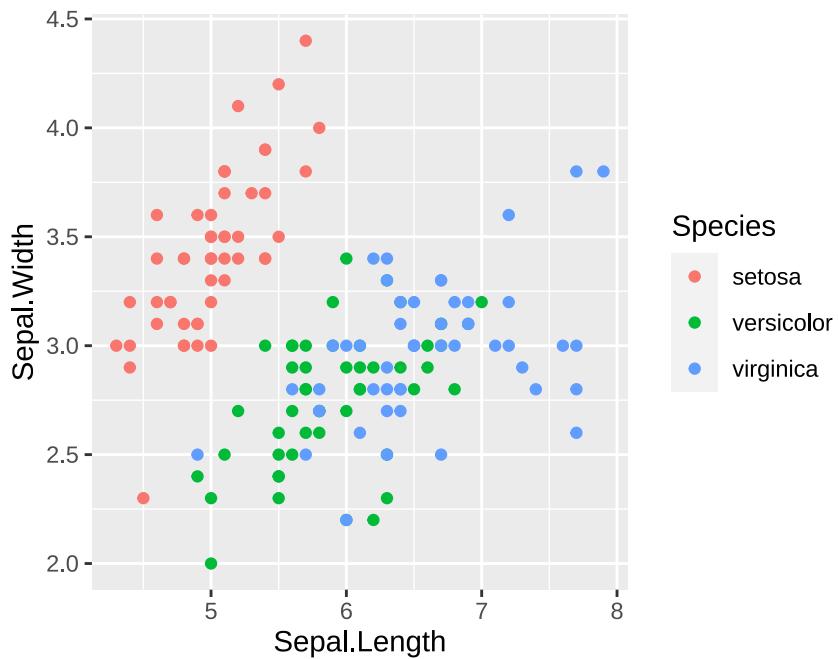
Aufgabe 20: Abbildungen mit *ggplot2*

1119 Verwenden Sie die den Iris Datensatz und erstellen Sie mit *ggplot2* einen Plot der die Verteilung der Länge
1120 der Kelchblätter zeigt (Spalte `Sepal.Length`).

1121

1122 Eine der Stärken von *ggplot2* ist, dass man den Wert unterschiedlicher Variable auf unterschiedlichen
1123 Komponenten des Plots abbilden kann. Wir haben bis jetzt ein bzw. zwei Variable auf der x- und y-Achse
1124 abgebildet. Wir können aber ein weitere Variablen verwenden um das Aussehen des Plots zu beeinflussen.
1125 Beispielsweise können wir die Farbe der Punkte (für `geom_point()`) mit dem Argument `col` beeinflussen.

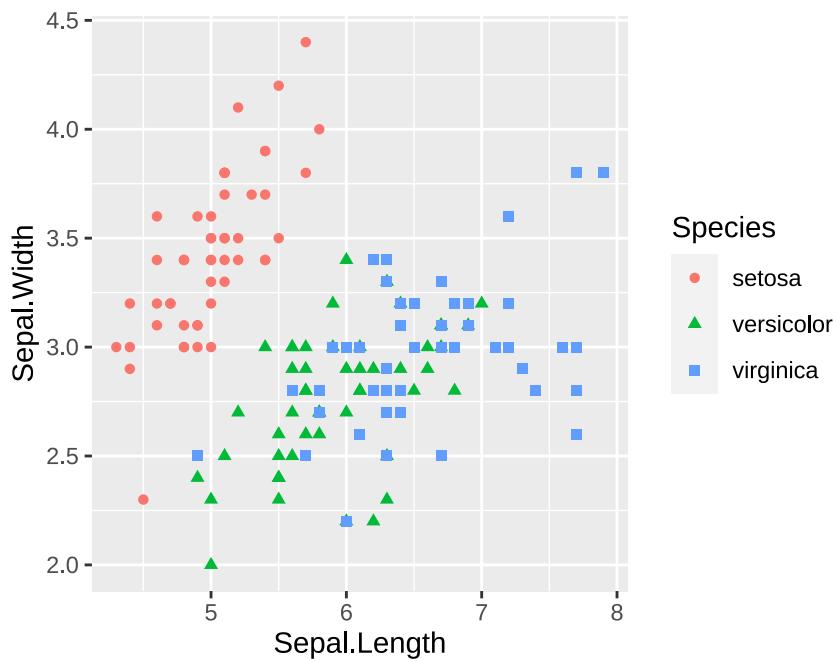
```
1126 ggplot(iris, aes(x = Sepal.Length, y = Sepal.Width, col = Species)) +  
1127   geom_point()
```



1126

- 1127 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()
```

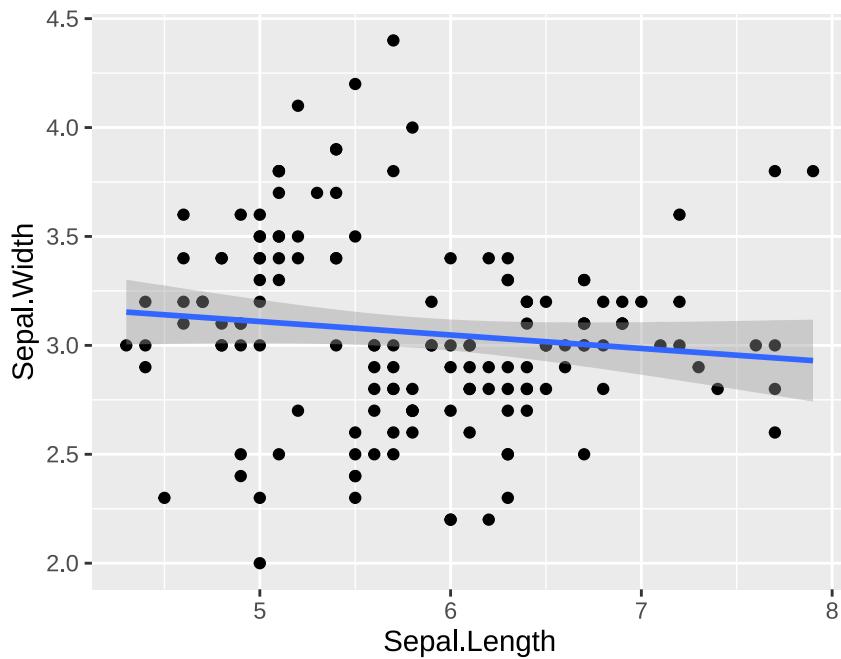


1129

⁸Natürlich könnte man auch die Farbe anpassen.

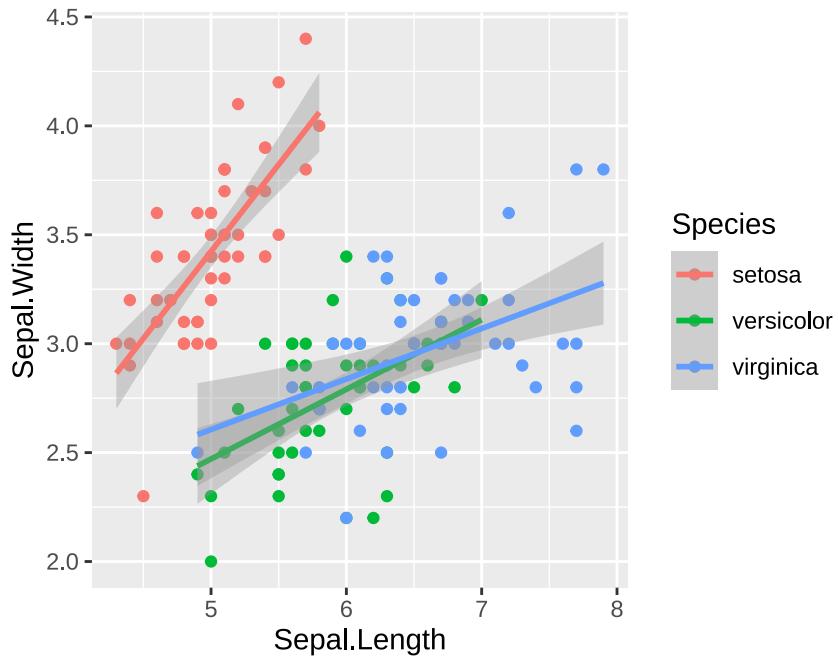
- 1130 In dem Plot ist die Information zu der Art redundant (einmal als Farbe und einmal Symbolart).
- 1131 Ein weitere sehr nützliche Geometrie ist `geom_smooth()`, die es erlaubt eine Trendlinie hinzuzufügen.

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

1132

- 1133 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 1134 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")
```



1136

1137

Aufgabe 21: Anpassen von Plots

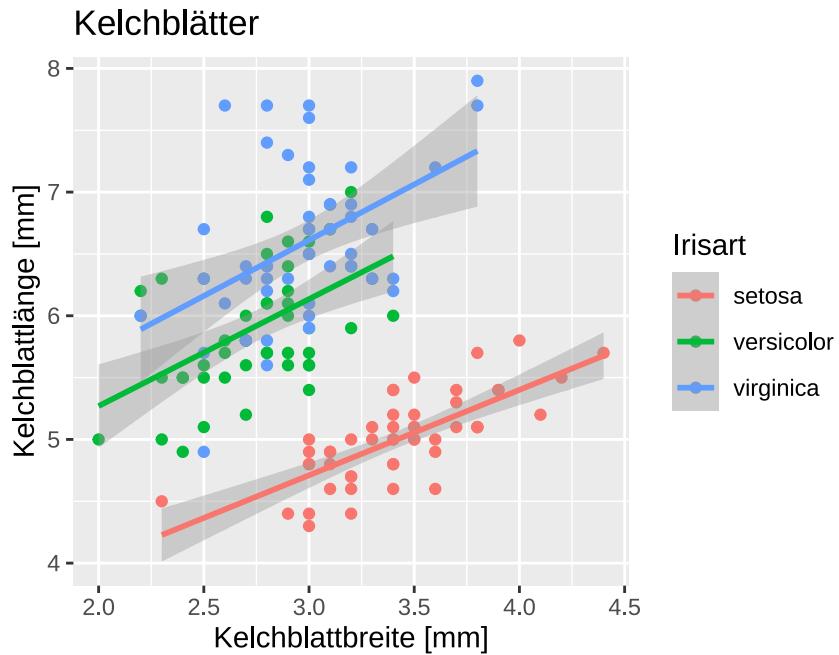
- 1140 Lesen Sie den Datensatz `data/bhd_1.txt` ein und erstellen Sie einen Boxplot für die Verteilung des BHDs
 1141 für jede Baumart. In einem zweiten Schritt verwenden Sie erst `col = gebiet` und dann `fill = gebiet`.
 1142 Welchen Unterschied stellen Sie fest?

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

1143

- 1144 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")
```



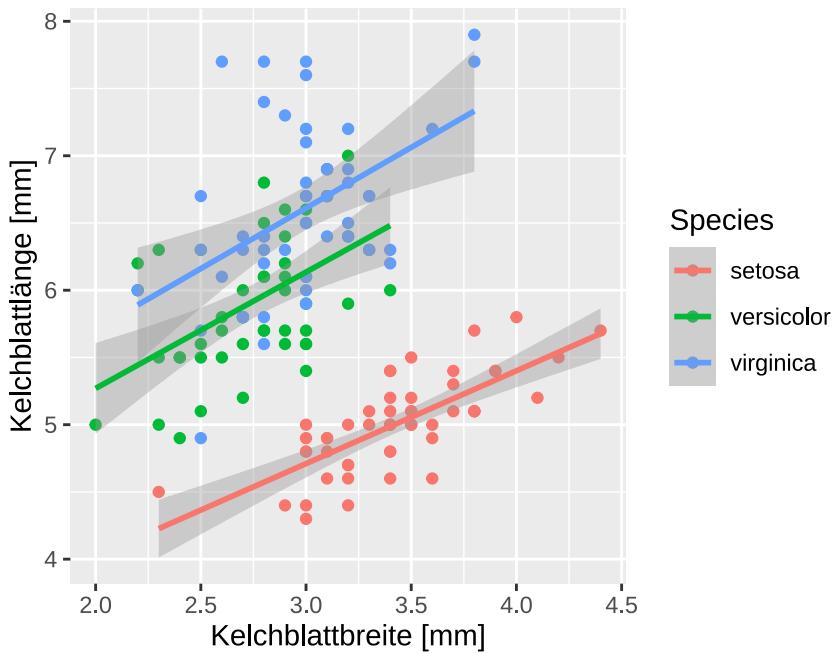
1145

- 1146 Statt einen langen Befehl zu tippen, kann ein `ggplot()` auch zwischengespeichert und wieder aufgerufen bzw.
1147 angepasst werden. Das ist vor allem sinnvoll, wenn mehrere Abbildungen auf dem selben Zwischenergebnis
1148 aufbauen sollen.

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

- 1149 Wir können jetzt mit `p1` weiter arbeiten und beispielsweise eine Beschriftung hinzufügen.

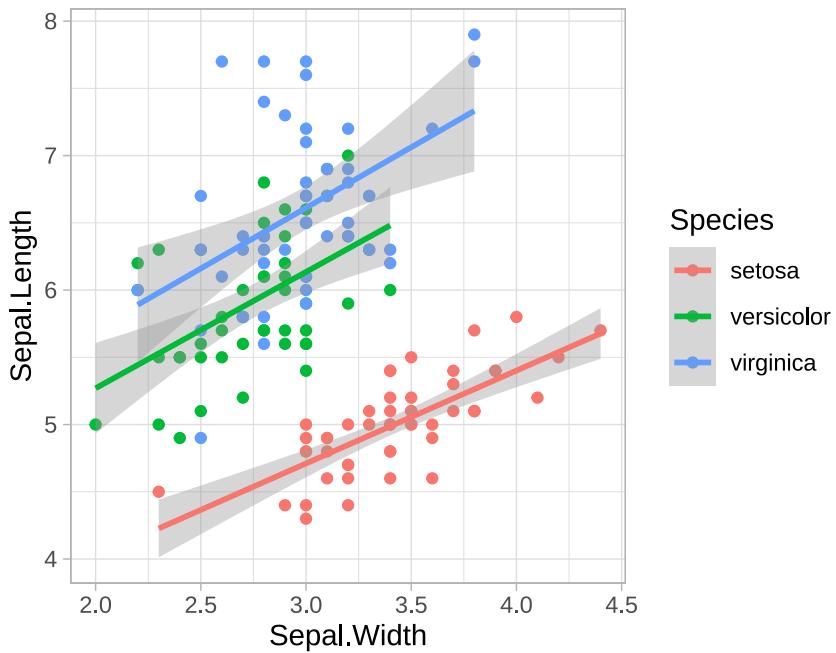
```
p1 + labs(x = "Kelchblattbreite [mm]", y = "Kelchblattlänge [mm]")
```



1150

- 1151 Oder auch den ganzen Plot anpassen. Dafür gibt es *themes*. Es gibt eine Reihe von vorgefertigten *themes*
1152 oder man kann diese auch selber erstellen (das ist aber nicht Teil dieses Kurses).

```
p1 + theme_light()
```

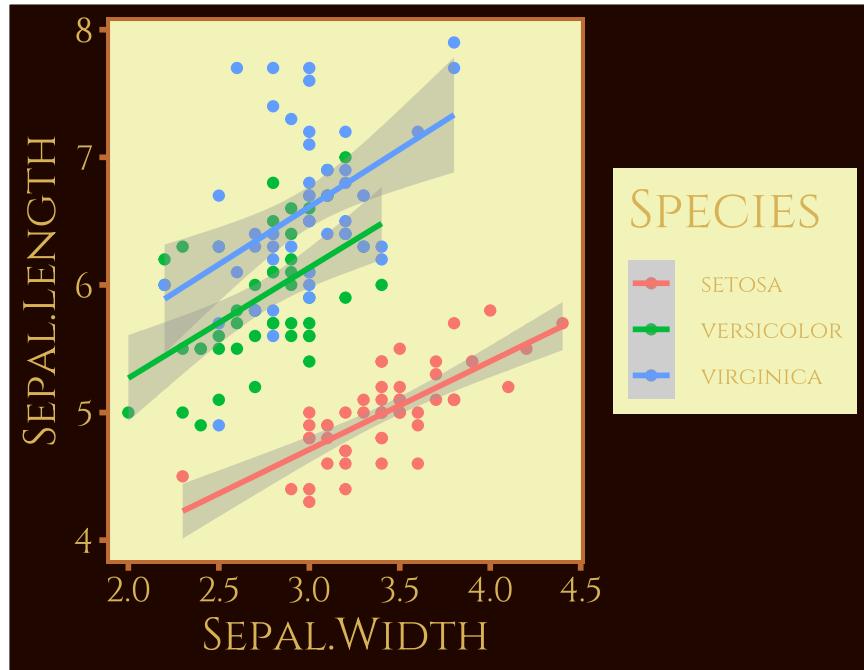


1153

- 1154 Weitere *themes* sind: `theme_bw()`, `theme_linedraw()` oder `theme_dark()`. Es gibt extra Pakete die vie-
1155 le zusätzliche weitere *themes* anbieten. Dazu gehört z.B. das Paket `ggthemes` oder `ThemePark`. Während

1156 ggthemes hauptsächlich durchdachte Grafikkonzepte liefert, sind die Themes aus ThemePark eher Popkultur
1157 und nicht 100 %ig ernst gemeint.

```
p1 + theme_gamethrones()
```

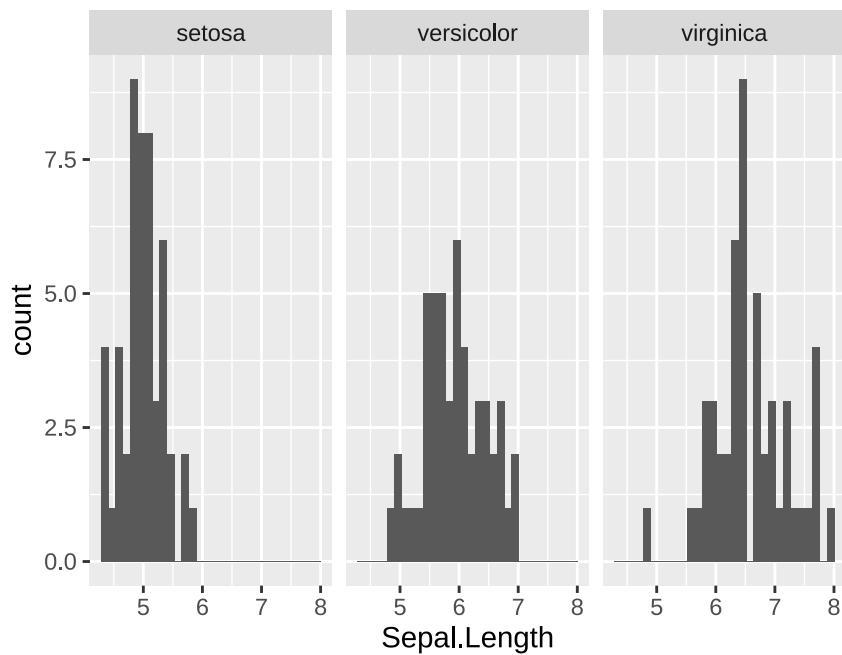


1158

1159 8.4.1 Multipanel Abbildungen

1160 Mit *ggplot2* kann man einfach Abbildungen erstellen, die mehrere Panels haben. Das bedeutet, dass eine
1161 oder mehrere weitere Variablen gibt, die einen Plot in mehrere Subplots teilt. Dafür gibt es zwei Funktionen:
1162 `facet_grid()` und `facet_wrap()`.

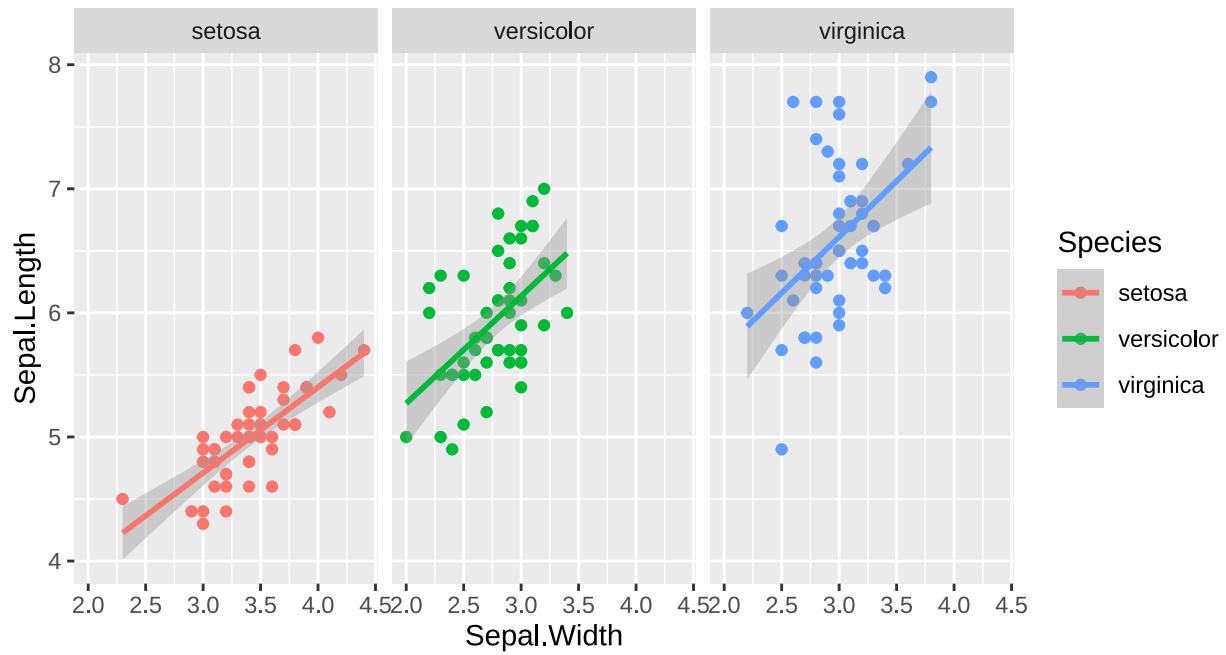
```
ggplot(iris, aes(Sepal.Length)) + geom_histogram() +  
  facet_grid(~ Species)
```



1163

- 1164 Die Funktion `facet_grid()` erzeugt einen *Grid* und beschriftet die Grid-Zeilen und -Spalten, während
 1165 `facet_wrap()` für jedes Panel (Diagramm) eine eigene Überschrift erzeugt. Beim Arrangieren der Diagram-
 1166 me wird der Vorteil von `ggplot2` gegenüber `base plot` besonders deutlich. Während es im `base plot` System
 1167 mühsam ist, die Abbildungen zu arrangieren, übernimmt `ggplot2` das Arrangement automatisch und fügt
 1168 sogar eine automatisch Legende hinzu. Die Achsenabschnitte werden so angepasst, dass die Daten vergleich-
 1169 bar sind.

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



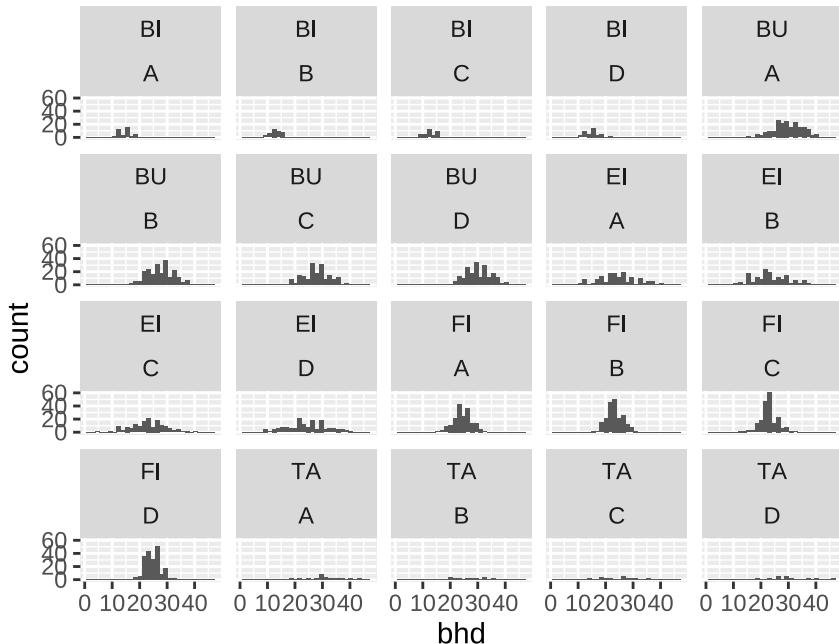
1170

1171

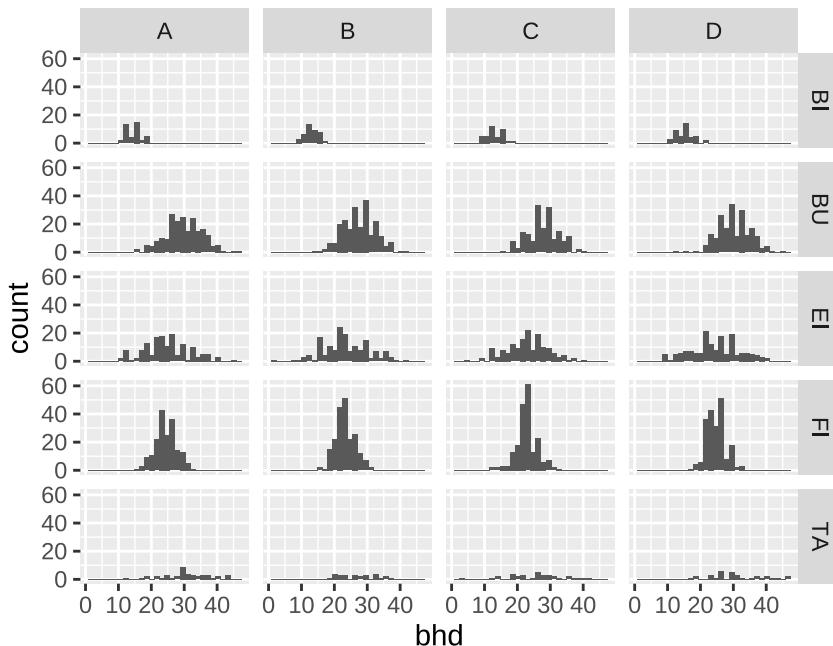
1172 Aufgabe 22: Multipanel Abbildungen

1173

- 1174 Lesen Sie erneut den Datensatz daten/bhd_1.txt ein und speichern Sie diesen in die Variable (`dat_bhd`).
- 1175 Erstellen Sie für jede Art und Gebiet ein Histogramm. Welche Unterschiede können Sie feststellen, wenn Sie `facet_grid()` oder `facet_wrap()` verwenden?



1177



1178

1179 8.4.2 Plots kombinieren

1180 Es gibt Situationen, in denen **unterschiedliche** Plots miteinander kombiniert werden müssen. Im vorherigen
 1181 Abschnitt wurde dies immer anhand einer gruppierenden Kovariate gemacht. Aber es gibt auch Situations-
 1182 en, in denen das nicht möglich ist. Beispielsweise wenn ein Histogramm und ein Scatterplot vom gleichen
 1183 Datensatz zusammengefasst werden sollen. Dafür bietet sich das Paket **patchwork** an⁹.

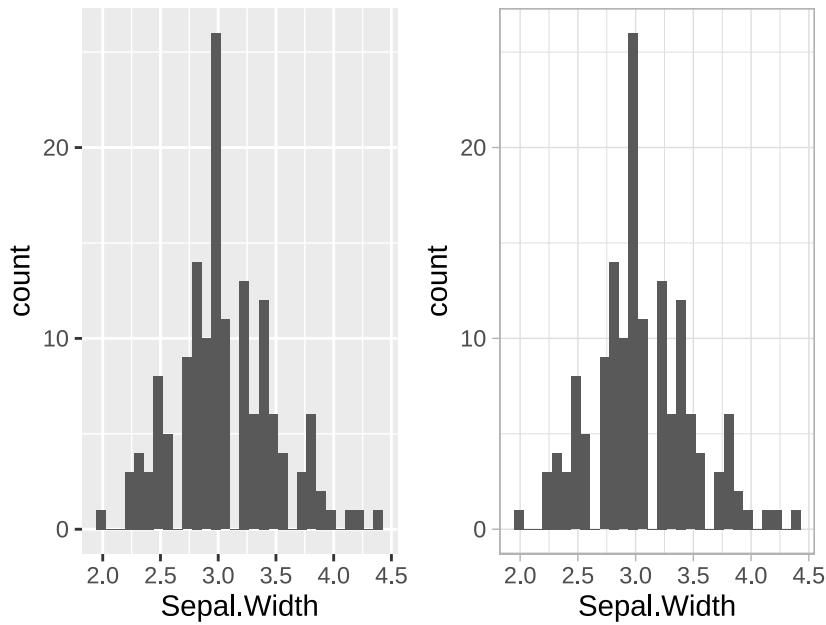
1184 Als erstes können wir zwei (oder natürlich auch mehrere Plots) erstellen. Hier unterscheiden sich die Plots
 1185 lediglich durch das Aussehen.

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

1186 Dann müssen können wir diese Plots ebenfalls mit `+` zusammenfügen.

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

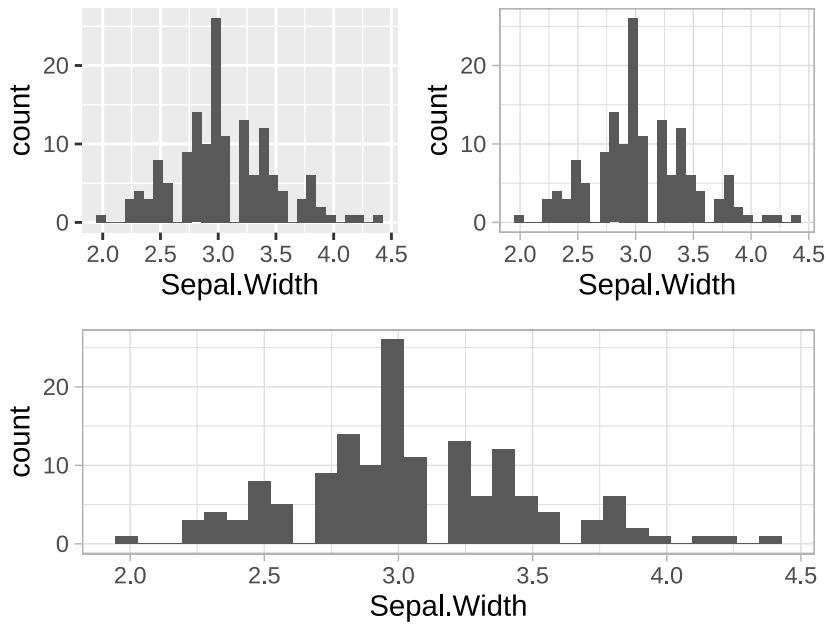
⁹Auch dieses Paket müssen Sie einmalig mit `install.packages("patchwork")` installieren.



1187

1188 Natürlich können auch weitere Plots hinzugefügt werden (auch in unterschiedlichen Dimensionen):

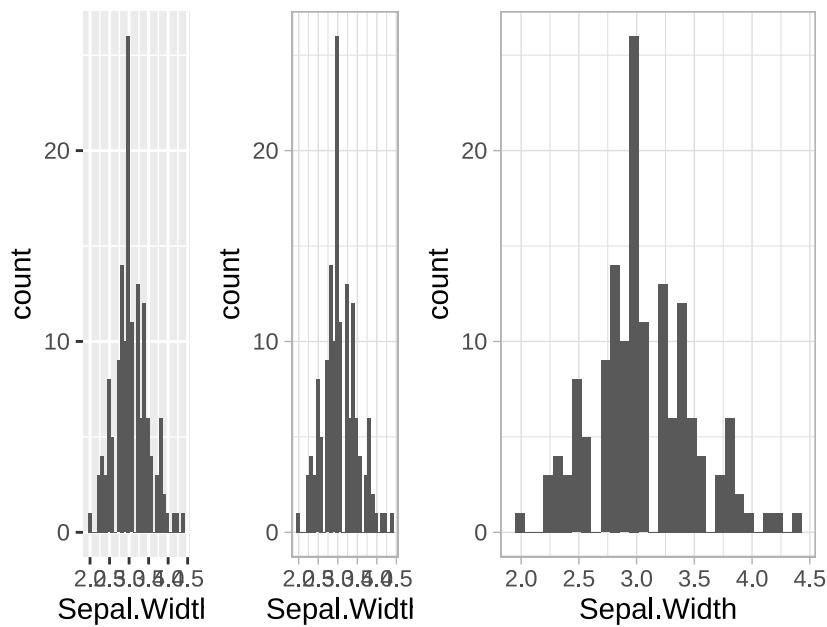
(p1 + p2) / p2



1189

1190 Des weiteren können mit | auch Plots gegenüber gestellt werden.

(p1 + p2) | p2



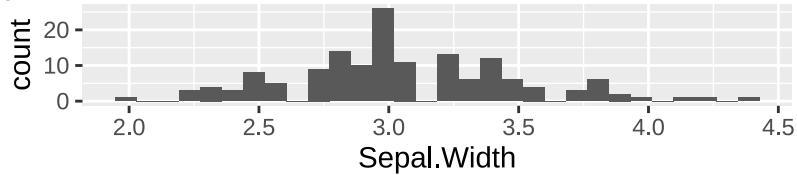
1191

1192 Weitere Optionen können mit `plot_layout()` und `plot_annotation()` angepasst werden. Mit
 1193 `plot_layout()` kann die Anordnung der Plots bestimmt werden (z.B. über die Argumente `nrow` und
 1194 `ncol`), sowie deren *relative* Größe (über die Argumente `widths` und `heights`). Mit der Funktion
 1195 `plot_annotation()` können zusätzliche Beschriftungen hinzugefügt werden, wie beispielsweise eine Titel
 1196 (Argument `title`) oder ein Buchstabe/Zahl für jedes Element (Argument `tag_levels`).

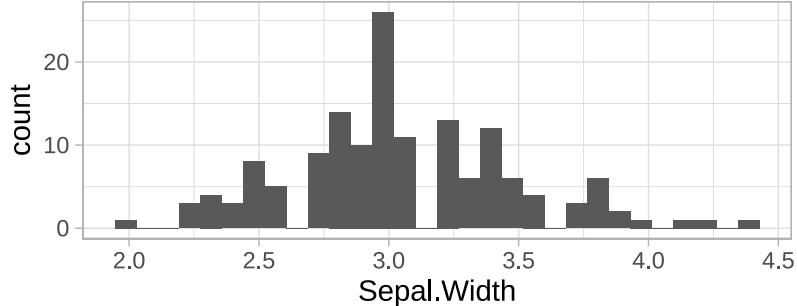
```
p1 + p2 +
  plot_layout(ncol = 1, heights = c(0.3, 0.7)) +
  plot_annotation(title = "Zwei Histogramme", tag_levels = "A")
```

Zwei Histogramme

A



B



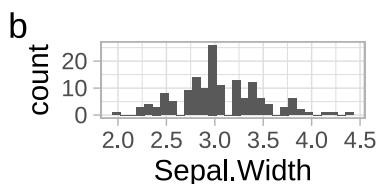
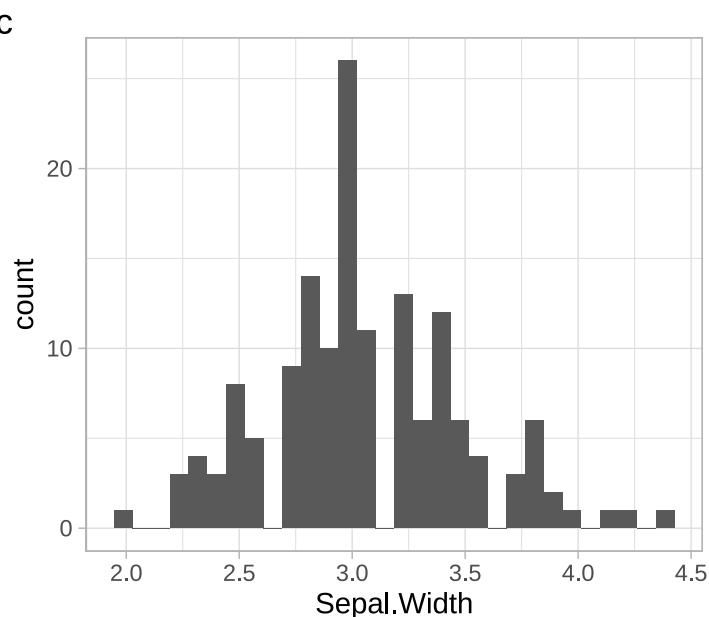
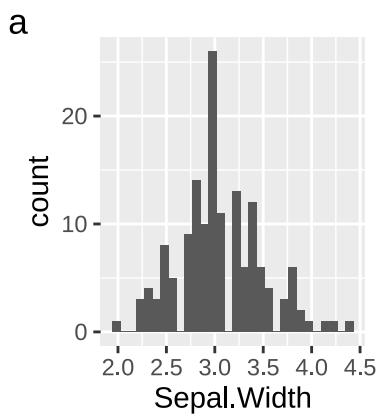
1197

1198

1199 **Aufgabe 23: Mehrere Plots zusammenfügen**

1200

- 1201 Versuchen Sie die folgende Zusammenstellung der Plots nachzumachen:



1202

1203 8.4.3 Speichern von plots

1204 Sie können mit `ggsave()` eine zwischen-gespeicherte Abbildung exportieren, indem Sie den Variablenamen
1205 übergeben. Wenn Sie keine Variable übergeben, wird automatisch die letzte Abbildung gespeichert. Das
1206 Dateiformat wird aus dem Dateinamen übernommen. Die Größe der Inhalte können Sie sehr leicht mit den
1207 Argumenten `width` und `height` oder `scale` gesteuert werden.

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

1208 9 Mit Daten arbeiten

1209 9.1 dplyr eine Einführung

1210 `dplyr` ist eine Erweiterung von R (ein Paket), die das Ziel hat, den Umgang mit Daten einfacher und
1211 schneller zu machen.

1212 `dplyr` definiert 5 Verben, um mit Daten zu arbeiten. Diese sind:

- 1213 • `filter`
- 1214 • `select`
- 1215 • `arragne`
- 1216 • `mutate`
- 1217 • `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))
```

1218 Damit die Funktionen aus `dplyr` verwendet werden können, müssen wir als erstes das Paket `dplyr` laden.
1219 The online Resource [R for Data Science](#) gibt einen sehr guten, tieferen Eindruck in die Arbeit mit `dplyr`
1220 und erklärt auch die `dplyr` Philosophie noch Mal sehr genau für Beginner.

```
library(dplyr)
```

1221 Sollte dies zu einer Fehlermeldung führen, dann müssen Sie das Paket `dplyr` erst installieren. Dafür müssen
1222 Sie `einmalig install.packages("dplyr")` installieren.

1223 `dplyr` stellt unterschiedliche Funktionen zum Arbeiten mit Daten zur Verfügung. Es gibt fünf Grundfunktionen
1224 für die am häufigsten vorkommenden Operationen. Mit der Funktion `filter()` können unterschiedliche
1225 Beobachtungen gefiltert werden:

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

```
1226 ##   id plot bhd alter
1227 ## 1   1    1  50   10
1228 ## 2   2    1  29   30
1229 ## 3   3    2  13   31
1230 ## 4   4    2  23   24
1231 ## 5   5    3  25   25
```

1232 Es können auch mehrere Spalten verwendet werden.

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

```
1233 ##   id plot bhd alter
1234 ## 1   2     1   29   30
1235 ## 2   3     2   13   31
1236 ## 3   4     2   23   24
1237 ## 4   5     3   25   25
```

1238 Natürlich kann genau das gleiche Ergebnis mit dem ‘normalen’ R erreicht werden, dies wäre dann:

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

```
1239 ##   id plot bhd alter
1240 ## 2   2     1   29   30
1241 ## 3   3     2   13   31
1242 ## 4   4     2   23   24
1243 ## 5   5     3   25   25
```

1244 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)
```

```
1246 ##   bhd
1247 ## 1   50
1248 ## 2   29
1249 ## 3   13
1250 ## 4   23
1251 ## 5   25
```

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

```
1252 ##   bhd id
1253 ## 1   50  1
1254 ## 2   29  2
1255 ## 3   13  3
1256 ## 4   23  4
1257 ## 5   25  5
```

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

```

1258 ##   BHD id
1259 ## 1 50 1
1260 ## 2 29 2
1261 ## 3 13 3
1262 ## 4 23 4
1263 ## 5 25 5

```

1264 Mit der Funktion `arrange()` können die Beobachtungen in einem `data.frame` sortiert werden.

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

```

1265 ##   id plot bhd alter
1266 ## 1 3 2 13 31
1267 ## 2 4 2 23 24
1268 ## 3 5 3 25 25
1269 ## 4 2 1 29 30
1270 ## 5 1 1 50 10

```

1271 Mit der Funktion `desc()` kann die Anordnung in absteigender Reihenfolge sortiert werden.

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

```

1272 ##   id plot bhd alter
1273 ## 1 1 1 50 10
1274 ## 2 2 1 29 30
1275 ## 3 5 3 25 25
1276 ## 4 4 2 23 24
1277 ## 5 3 2 13 31

```

1278 Mit der Funktion `mutate()` kann man eine neue Spalte hinzufügen.

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

```

1279 ##   id plot bhd alter bhd_mm          fl
1280 ## 1 1 1 50 10 500 1963.4954
1281 ## 2 2 1 29 30 290 660.5199
1282 ## 3 3 2 13 31 130 132.7323
1283 ## 4 4 2 23 24 230 415.4756
1284 ## 5 5 3 25 25 250 490.8739

```

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

```

1285 ##   id plot bhd alter mean_bhd
1286 ## 1   1     1   50    10    28
1287 ## 2   2     1   29    30    28
1288 ## 3   3     2   13    31    28
1289 ## 4   4     2   23    24    28
1290 ## 5   5     3   25    25    28

```

1291 Mit der Funktion `summarise()` können Spalten zusammengefasst werden.

```

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

1292 ##   mean_bhd  mean_sd
1293 ## 1          28 13.63818

```

1294

1295 Aufgabe 24: Datenmanipulation mit dplyr

- 1297 1. Laden Sie den Datensatz `data/bhd_1.txt`
- 1298 2. Berechnen Sie folgende Werte für alle Einträge und speichern Sie die Ergebnisse in `erg1`

```

1299 • mittlerer bhd
1300 • maximales alter
1301 • die Standardabweichung des BHDs
1302 • die Anzahl Bäume mit einem BHD > 30

```

1303 9.2 Arbeiten mit gruppierten Daten

1304 Zusätzlich können `mutate` und `summarise` auch auf gruppierte Daten angewendet werden. Dafür müssen
 1305 wir erst die Funktion `group_by()` aufrufen und als Argumente die Variablen übergeben, die die Gruppen
 1306 definieren.

```

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

```

```

1307 ##   id plot bhd alter bhd_m
1308 ## 1   1     1   50    10    28
1309 ## 2   2     1   29    30    28

```

```

1310 ## 3 3 2 13 31 28
1311 ## 4 4 2 23 24 28
1312 ## 5 5 3 25 25 28

```

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

```

1313 ## # A tibble: 5 x 5
1314 ## # Groups:   plot [3]
1315 ##       id   plot   bhd alter bhd_m
1316 ##     <int> <dbl> <dbl> <dbl> <dbl>
1317 ## 1     1     1    50    10  39.5
1318 ## 2     2     1    29    30  39.5
1319 ## 3     3     2    13    31  18
1320 ## 4     4     2    23    24  18
1321 ## 5     5     3    25    25  25

```

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

```

1322 ## bhd_m
1323 ## 1 28

```

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

```

1324 ## # A tibble: 3 x 2
1325 ##   plot bhd_m
1326 ##   <dbl> <dbl>
1327 ## 1     1  39.5
1328 ## 2     2  18
1329 ## 3     3  25

```

1330

Aufgabe 25: dplyr mit gruppierten Daten

- 1333 1. Laden Sie den Datensatz `data/bhd_1.txt`
 1334 2. Berechnen Sie für jede Baumart und Standort die folgenden Werte:

- 1335 • mittlerer `bhd`
 1336 • maximales `alter`
 1337 • die Standardabweichung des BHDS
 1338 • die Anzahl Bäume mit einem BHD > 30

- 1339 3. Ordnen Sie das Ergebnis nach Baumart und BHD.

1340 **9.3 pipes oder %>%**

1341 Mit *Pipes* (%>%) kann man das Ergebnis einer Funktion einfach an eine nachfolgende Funktion weiterreichen.

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

1342 Wir kennen bis jetzt:

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

1343 ## [1] 3.333333

1344 Mit *Pipes*, die durch das Symbol %>% dargestellt werden¹⁰, können wir das etwas vereinfachen und nacheinander schreiben:

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

1346 ## [1] 3.333333

1347 Oder sogar

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

1348 ## [1] 3.333333

1349

Aufgabe 26: Pipes %>%

1352 Wiederholen Sie die letzte Aufgabe, aber diesmal ohne Zwischenergebnisse zu speichern:

1353 1. Laden Sie den Datensatz `data/bhd_1.txt`.

1354 2. Berechnen Sie für jede Baumart und Standort die folgenden Werte:

- 1355 • mittlerer `bhd`
- 1356 • maximales `alter`
- 1357 • die Standardabweichung des BHDs
- 1358 • die Anzahl Bäume mit einem BHD > 30

1359 3. Ordnen Sie das Ergebnis nach Baumart und BHD.

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

1360 9.4 Joins

- 1361 Eine weitere häufige Aufgabe beim Daten Management ist es Daten zusammenzuführen. Nehmen Sie an,
 1362 dass wir folgende Aufnahmen gemacht haben

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

- 1363 und jeder Baum lediglich mit einer `id` versehen wurde. In einer zweiten Tabelle wurden dann weitere Daten
 1364 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")
)
```

- 1365 Ziel ist es jetzt die Bäume aus `aufnahmen` mit den Informationen aus den `metadaten` zu verbinden. Dazu
 1366 dient `id` als Bindeglied (oft auch Schlüssel genannt).

- 1367 Dazu gibt es vier Möglichkeiten.

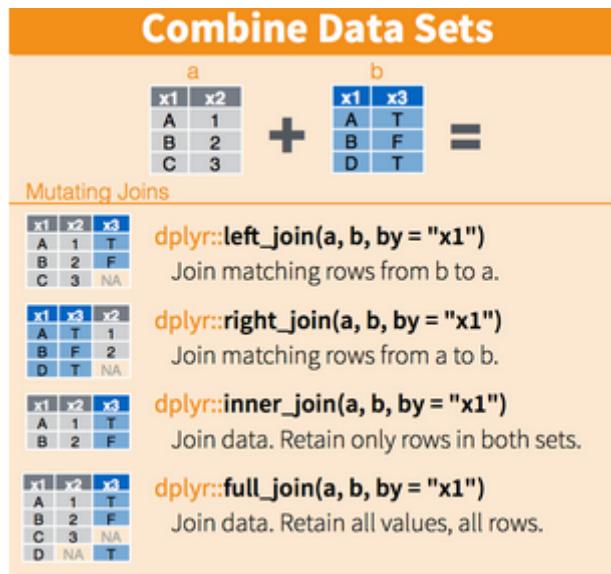


Abbildung 8: Joins (Quelle Rstudio)

- 1368 Zur Durchführung gibt es in base R die Funktion `merge()`. Wir werden aber gleich die Funktionen aus dem
 1369 Paket `dplyr` verwenden.

```

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

1370 ##   id bhd  art gebiet
1371 ## 1  1  20 <NA>    <NA>
1372 ## 2  2  31   Ta      A
1373 ## 3  3  74   Bu      B

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

1374 ##   id bhd art gebiet
1375 ## 1   2  31  Ta      A
1376 ## 2   3  74  Bu      B
1377 ## 3   4  NA  Bu      B

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

1378 ##   id bhd art gebiet
1379 ## 1   2  31  Ta      A
1380 ## 2   3  74  Bu      B

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

1381 ##   id bhd  art gebiet
1382 ## 1   1  20 <NA>    <NA>
1383 ## 2   2  31   Ta      A
1384 ## 3   3  74   Bu      B
1385 ## 4   4  NA  Bu      B

1386 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"))

1387 ##   id bhd  art gebiet
1388 ## 1   1  20 <NA>    <NA>
1389 ## 2   2  31   Ta      A
1390 ## 3   3  74   Bu      B

```

1391

1392 **Aufgabe 27: Verbinden von Daten**

1393

- 1394 • Lesen Sie die Datensätze `daten/bhd_2.txt` und `daten/bhd_2_meta.txt` ein.
1395 • Stellen Sie sicher, dass es für den `bhd` keine fehlenden Werte gibt (entfernen sie entsprechende Zeilen)
1396 • Fügen Sie zu den Metadaten (gespeichert in `bhd_2_meta`) die Anzahl Bäume und den mittleren `bhd`
1397 hinzu pro Gebiet.

1398 **9.5 ‘long’ and ‘wide’ Datenformate**

1399 Unter anderem Wickham (2014) empfiehlt das Prinzip von *tidy* Data. Nach diesem Prinzip sollten Daten wie
1400 folgt organisiert sein:

- 1401 • Jede Zeile ist ein Merkmalsträger/Subjekt/Objekt (z. B. eine Person oder ein Baum).
1402 • Jede Spalte ist eine Variable (ein Merkmal), die den Merkmalsträger beschreibt.
1403 • In jeder Zelle ist genau ein Wert (Merkmalausprägung), nämlich der Wert, der Variable für den Merk-
1404 malsträger.

1405 Zum Beispiel enthalten Spaltennamen oft Informationen, die eigentlich in einer Variable gespeichert werden
1406 sollten. Folgendes Beispiel gibt die BHD Messung von 3 Bäumen in 3 Jahren wieder. Der Vorteil von *tidy*
1407 Daten ist, dass viele Funktionen diese Form erwarten. Somit müssen sie Ihre Daten nur ein Mal organisieren
1408 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)  
)
```

1409 Der `tibble` aus dem Paket `tibble` ist ein moderner Data Frame, der sich in seinen Funktionen in das
1410 `tidy` Universum einfügt. Wir können auch einfach das Paket `tidyverse` laden, um alle Pakete, die Teil des
1411 `tidy` Universums sind gleichzeitig zu laden. Der `tibble` beinhaltet alle Funktionen, die der Data Frame
1412 auch beinhaltet und kann auch für genau die gleichen Zwecke verwendet werden. Der `tibble` hat u. A. eine
1413 moderne Darstellung im Konsolenoutput.

1414 Diese Daten sind jetzt im `wide`-Format gespeichert und folglich nicht `tidy`, weil Information über die Daten
1415 (nämlich das Jahr der Aufnahme in den Spaltennamen gespeichert sind). Besser wäre eine Struktur mit
1416 nur drei Spalten: `id`, `jahr` und `bhd`. Um die Daten in so eine Struktur zu bringen, gibt es die Funktion
1417 `pivot_longer()` aus dem Paket `tidyverse`, die ebenfalls Teil des `tidyverse` ist.

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

```
1418 ## # A tibble: 9 x 3
1419 ##       id name     value
1420 ##   <int> <chr>   <dbl>
1421 ## 1     1 bhd2015    30
1422 ## 2     1 bhd2026    31
1423 ## 3     1 bhd2017    34
1424 ## 4     2 bhd2015    31
1425 ## 5     2 bhd2026    31
1426 ## 6     2 bhd2017    32
1427 ## 7     3 bhd2015    32
1428 ## 8     3 bhd2026    33
1429 ## 9     3 bhd2017    33
```

1430 Wenn wir die Spalten für die Variable und den Wert gleich sinnvoll benennen möchten, können wir das über
 1431 die Argumente `names_to` und `values_to` machen.

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

```
1432 ## # A tibble: 9 x 3
1433 ##       id jahr     bhd
1434 ##   <int> <chr>   <dbl>
1435 ## 1     1 bhd2015    30
1436 ## 2     1 bhd2026    31
1437 ## 3     1 bhd2017    34
1438 ## 4     2 bhd2015    31
1439 ## 5     2 bhd2026    31
1440 ## 6     2 bhd2017    32
1441 ## 7     3 bhd2015    32
1442 ## 8     3 bhd2026    33
1443 ## 9     3 bhd2017    33
```

1444 Analog zu der Funktion `pivot_longer()` gibt es auch die Funktion `pivot_wider()`, um vom Daten vom
 1445 long-Format ins wide-Format zu transformieren.

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

```
1446 ## # A tibble: 3 x 4
1447 ##       id bhd2015 bhd2026 bhd2017
```

```

1448 ## <int> <dbl> <dbl> <dbl>
1449 ## 1     1     30    31    34
1450 ## 2     2     31    31    32
1451 ## 3     3     32    33    33

```

1452

Aufgabe 28: Zeitliche Verlauf von BHDs

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

1458 **9.6 Auswählen von Variablen**

1459 Sobald die Datensätze etwas umfangreicher werden (d. h., es gibt mehrere Spalten in einem `data.frame`),
 1460 können innerhalb vieler `dplyr`-Funktionen spezielle Funktionen verwendet werden, um Variablen auszuwählen.
 1461

1462 Wenn die genaue Position der Spalten bekannt ist, kann man mit dem `:`-Operator und der Position Spalten
 1463 auswählen:

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

```

1464 ## Sepal.Length Sepal.Width Petal.Length
1465 ## 1          5.1      3.5      1.4
1466 ## 2          4.9      3.0      1.4
1467 ## 3          4.7      3.2      1.3

```

1468 Diese Vorgehensweise kann gehährlich sein, da sich manchmal Spalten verschieben und sich somit die Positionen ändern. Es besser Spalten immer explizit mit Namen statt mit Nummern anzusprechen.

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

```

1470 ## Sepal.Length Sepal.Width Petal.Length
1471 ## 1          5.1      3.5      1.4
1472 ## 2          4.9      3.0      1.4
1473 ## 3          4.7      3.2      1.3

```

1474 `select()` erlaubt es, auch hier den `:`-Operator zu verwenden:

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

```
1475 ## Sepal.Length Sepal.Width Petal.Length
1476 ## 1      5.1      3.5      1.4
1477 ## 2      4.9      3.0      1.4
1478 ## 3      4.7      3.2      1.3
```

1479 Es gibt auch einige spezielle Funktionen, um Spalten innerhalb eines `select()`-Aufrufs auszuführen:

- 1480 • `starts_with()`: Hier kann man ein Muster angeben, mit dem ein Text anfangen muss.
- 1481 • `ends_with()`: Diese Funktion ist analog zu `starts_with()`, jetzt wird aber am Ende des Spaltennamens nach dem Muster gesucht.
- 1483 • `contains()`: Hier kann ein Muster übergeben werden, das irgendwo im Spaltennamen sein muss.
- 1484 • `everything()`: Mit dieser Funktion werden alle Spalten ausgewählt.
- 1485 • `last_col()`: Mit dieser Funktion wird nur die letzte Spalte ausgewählt (dass ist die Spalte, die ganz rechts ist).

1487 Sämtliche Auswahlen können mit - umgekehrt werden.

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

```
1488 ## Sepal.Length Sepal.Width
1489 ## 1      5.1      3.5
1490 ## 2      4.9      3.0
1491 ## 3      4.7      3.2
```

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

```
1492 ## Petal.Length Petal.Width Species
1493 ## 1      1.4      0.2 setosa
1494 ## 2      1.4      0.2 setosa
1495 ## 3      1.3      0.2 setosa
```

1496 `select()` bietet auch noch die Möglichkeit, Spaltennamen zu ändern.

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

```
1497 ## sep_width
1498 ## 1      3.5
1499 ## 2      3.0
1500 ## 3      3.2
```

1501

1502 **Aufgabe 29: Auswählen von Spalten**

1504 In der Datei `messungen_1.csv` sind Messungen von zwei Sensoren enthalten für die ersten vier Monate eines
1505 Jahres. Führen Sie folgende Abfragen durch:

- 1506 1. Wählen Sie alle Messungen für Januar aus.
1507 2. Wählen Sie alle Messungen für Januar und März aus.

1508 **9.7 Einzelne Beobachtungen abfragen (`slice()`)**

1509 Mit dem Befehl `slice()` kann man einzelne Beobachtungen (= Zeilen) aus einem `data.frame` abfragen.

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

1510 ## Sepal.Length Sepal.Width Petal.Length Petal.Width Species
1511 ## 1 5.1 3.5 1.4 0.2 setosa
1512 ## 2 4.4 2.9 1.4 0.2 setosa
1513 ## 3 5.1 3.5 1.4 0.3 setosa

1514 Davon gibt es drei nützliche Varianten: 1) `slice_head()` und `slice_tail()`; 2) `slice_max()` und
1515 `slice_min()`; 3) `slice_random()`.

1516 `slice_head()` und `slice_tail()` sind analog zu `head()` und `tail()`, aber mit dem entscheidenden Unter-
1517 schied, dass Gruppierungen berücksichtigt werden. Wenn keine Gruppierung in den Daten vorhanden ist,
1518 gibt es keinen Unterschied.

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

1519 ## Sepal.Length Sepal.Width Petal.Length Petal.Width Species
1520 ## 1 5.1 3.5 1.4 0.2 setosa
1521 ## 2 4.9 3.0 1.4 0.2 setosa

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

1522 ## Sepal.Length Sepal.Width Petal.Length Petal.Width Species
1523 ## 1 5.1 3.5 1.4 0.2 setosa
1524 ## 2 4.9 3.0 1.4 0.2 setosa

1525 Sobald jedoch eine gruppierende Variable eingeführt wird, gibt `slice_head()` die ersten `n` Beobachtungen
1526 für jede Gruppe zurück und `head()` für den gesamten Datensatz.

```

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

1527 ## # A tibble: 2 x 5
1528 ## # Groups:   Species [1]
1529 ##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
1530 ##     <dbl>       <dbl>       <dbl>       <dbl> <fct>
1531 ## 1     5.1         3.5         1.4        0.2  setosa
1532 ## 2     4.9         3           1.4        0.2  setosa

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

1533 ## # A tibble: 6 x 5
1534 ## # Groups:   Species [3]
1535 ##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
1536 ##     <dbl>       <dbl>       <dbl>       <dbl> <fct>
1537 ## 1     5.1         3.5         1.4        0.2  setosa
1538 ## 2     4.9         3           1.4        0.2  setosa
1539 ## 3     7           3.2         4.7        1.4  versicolor
1540 ## 4     6.4         3.2         4.5        1.5  versicolor
1541 ## 5     6.3         3.3         6          2.5  virginica
1542 ## 6     5.8         2.7         5.1        1.9  virginica

1543 slice_tail() funktioniert analog zu slice_head() mit dem einzigen Unterschied, dass nicht die ersten n
1544 Zeilen zurück gegeben werden sondern die letzten n Zeilen.

1545 slice_max() und slice_min() geben die Beobachtung mit dem maximalen bzw. minimalen Wert einer
1546 Variable zurück. Auch hier werden Gruppen berücksichtigt.

  iris %>% slice_max(Sepal.Length)

1547 ##   Sepal.Length Sepal.Width Petal.Length Petal.Width   Species
1548 ## 1           7.9         3.8         6.4          2  virginica

```

1549 Und mit Gruppen:

```

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

1550 ## # A tibble: 3 x 5

```

```

1551 ## # Groups: Species [3]
1552 ## Sepal.Length Sepal.Width Petal.Length Petal.Width Species
1553 ## <dbl>     <dbl>     <dbl>     <dbl> <fct>
1554 ## 1       5.8       4        1.2      0.2 setosa
1555 ## 2       7         3.2      4.7      1.4 versicolor
1556 ## 3       7.9      3.8      6.4      2    virginica

```

1557 `slice_min()` funktioniert genau gleich, nur dass die Beobachtung (=Zeile) mit dem kleinsten Wert einer
1558 Variable zurück gegeben wird.

1559 Die Funktion `slice_sample()` erlaubt es zufällige Beobachtungen zu ziehen. Dabei kann über das Argument
1560 `n` die Anzahl an Beobachtungen angegeben werden oder über das Argument `prop` der Anteil an Beobach-
1561 tungen.

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

```

1562 ## Sepal.Length Sepal.Width Petal.Length Petal.Width Species
1563 ## 1       6.5       2.8      4.6      1.5 versicolor
1564 ## 2       6.3       3.3      4.7      1.6 versicolor
1565 ## 3       7.2       3.2      6.0      1.8 virginica
1566 ## 4       4.9       3.6      1.4      0.1 setosa
1567 ## 5       6.0       2.7      5.1      1.6 versicolor

```

1568 Das Ergebnis ist bei jedem von Ihnen anders, da es sich um eine zufällige Ziehung handelt. Wenn Sie diese
1569 Ergebnisse wiederholen möchte, können Sie über `set.seed()` die zufällige Ziehung reproduzierbar machen.

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

```

1570 ## Sepal.Length Sepal.Width Petal.Length Petal.Width Species
1571 ## 1       4.3       3.0      1.1      0.1 setosa
1572 ## 2       5.0       3.3      1.4      0.2 setosa
1573 ## 3       7.7       3.8      6.7      2.2 virginica
1574 ## 4       4.4       3.2      1.3      0.2 setosa
1575 ## 5       5.9       3.0      5.1      1.8 virginica

```

1576 Wenn beispielsweise 5% der Beobachtungen gezogen werden sollen, kann dies so gemacht werden:

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

```

1577 ## Sepal.Length Sepal.Width Petal.Length Petal.Width Species
1578 ## 1       7.7       3.8      6.7      2.2 virginica
1579 ## 2       5.5       2.5      4.0      1.3 versicolor
1580 ## 3       5.5       2.6      4.4      1.2 versicolor

```

```

1581 ## 4      6.5      3.0      5.2      2.0  virginica
1582 ## 5      6.1      3.0      4.6      1.4  versicolor
1583 ## 6      6.3      3.4      5.6      2.4  virginica
1584 ## 7      5.1      2.5      3.0      1.1  versicolor

```

1585 `slice_sample()` berücksichtigt ebenfalls Gruppen. Mit den Argumenten `replace` und `weight_by` dann die
 1586 Zufallsziehung genauer spezifiziert werden. `replace` sagt, ob eine gezogenen Beobachtung wieder zurück ge-
 1587 legt wird oder nicht. Mit dem Argument `weight_by` können optional gewichte für jede Beobachtung vergeben
 1588 werden.

1589

1590 Aufgabe 30: Daten beschreiben

1592 Verwenden Sie den Datensatz `bhd_1.txt` und finden Sie für jedes Gebiet und Art die Beobachtung mit
 1593 kleinsten BHD.

1594 9.8 Spalten trennen

1595 Ein gut geplanter Datensatz besteht aus Beobachtungen (Zeilen), Variablen (Spalten) und in jeder Zelle ist
 1596 immer ein *genau* ein Wert gespeichert. Leider gibt es oft Datensätze, bei denen mehr als ein Wert pro Zelle
 1597 gespeichert wurde. Die Funktion `separate()` kann helfen solche Daten zu trennen.

1598 Wir verwenden einen erfunden Datensatz zu Beobachtungen von Tieren und einer geschätzten Distanz zu
 1599 diesen Tieren.

```

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

```

1600 In der Spalte `beobachtung` sind zwei Informationen gespeichert: Die Distanz zur Beobachtung und die Art.
 1601 Das ist ungünstig, weil wir so weder nach Tierart noch nach distanz filtern können (nicht *tidy*). Mit der
 1602 Funktion `separate()`, können wir Beobachtungen einer Spalte in mehrere Spalten trennen. Dafür muss der
 1603 Spaltennamen (Argument `col`), die neuen Spaltennamen (Argument `into`) und das Trennzeichen (Argument
 1604 `sep`) angegeben werden.

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

```

1605 ## # A tibble: 4 x 3
1606 ##       id Distanz Art
1607 ##     <int> <chr>   <chr>
1608 ## 1      1 10m     "Reh"

```

```
1609 ## 2      2 100m    " Reh"  
1610 ## 3      3 20m     " Fuchs"  
1611 ## 4      4 40      "Reh"
```

1612 Nach dem Aufruf von `seperate()` gibt es zwei neue Spalten (`Distanz` und `Art`), die die alte Spalte
1613 `beobachtung` ersetzen.

1614

1615 **Aufgabe 31: Aufräumen**

1617 Verwenden Sie den folgenden Datensatz und bringen Sie ihn in eine Form, die sicherstellt dass

- 1618 • jede Zelle genau einen Wert enthält.
1619 • jede Zeile eine Beobachtung ist.
1620 • 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"))  
)
```

1621 10 Arbeiten mit Text

1622 Bis jetzt haben wir fast ausschließlich mit Zahlen oder Abbildungen gearbeitet. R bietet aber auch viele
 1623 Werkzeuge, um mit Text zu arbeiten. Wir wollen hier ein paar Funktionen dafür vorstellen. Als erstes sollte
 1624 nochmals klargestellt werden, was eigentlich ein Text ist. In R ist alles, das innerhalb von doppelten ("") oder
 1625 einfachen ('') Anführungszeichen geschrieben ist, Text.

1626 Anbei einige Beispiele:

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

1627 Wichtig ist hier zu sehen, dass z nicht als Zahl sondern, als Text interpretiert wird (siehe Datentypen).

```
z + 1
```

1628 ## Error in z + 1: nicht-numerisches Argument für binären Operator

1629 Wenn man sicher ist, dass es sich bei einem Textobjekt um eine Zahl handelt, kann man dies mit der Funktion
 1630 `as.numeric()` in eine Zahl umwandeln.

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

1631 ## [1] 31

1632 Aber mit a führt dies wieder zu einem NA-Wert, da a nicht in eine Zahl umgewandelt werden kann.

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

1633 ## Warning: NAs durch Umwandlung erzeugt

1634 ## [1] NA

1635 10.1 Arbeiten mit Text

1636 Wir wollen erst einmal drei Funktionen besprechen, die es erlauben mit Text zu arbeiten. Die Funktion
 1637 `nchar()`¹¹ gibt an wie viele Zeichen ein Text hat. Also z.B.

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

1638 ## [1] 5

¹¹`char` ist kurz für *character*.

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

1639 ## [1] 2

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

1640 ## [1] 20

1641 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
```

1644 ## [1] "Eva Meier"

1645 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
```

1647 ## [1] "Eva, Meier"

1648 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)
```

1650 ## [1] "Hal"

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

1651 ## [1] "allo"

1652

1653 **Aufgabe 32: Arbeiten mit Text 1**

1655 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")
```

- 1656 1. Aus wie vielen Buchstaben besteht jedes Wort?
 1657 2. Finden Sie das längste Wort.
 1658 3. Wie viel Prozent der Wörter fangen mit einem S an?
 1659 4. Fügen Sie jedem Wort seine Position im Vektor hinzu. Beispielsweise soll aus Vogel "2. Vogel" werden
 1660 usw.

1661 10.2 Finden von Textmustern

- 1662 Mit der Funktion `grep()` können Muster in einem Text gefunden werden. Wenn wir beispielsweise folgenden
 1663 Vektor mit Textelementen haben.

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

- 1664 Und wir wollen alle Straßennamen die ein `weg` haben abfragen, dann können wir folgenden Befehl ausführen:

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

1665 ## [1] 2

- 1666 Im zweiten Element von `txt` kommen die Zeichen `Weg` vor. Beachte, in der Standardeinstellung wird zwischen
 1667 Groß- und Kleinschreibung unterschieden. Dies kann mit dem Argument `ignore.case = TRUE` angepasst
 1668 werden.

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

1669 ## [1] 1 2

- 1670 Mit der Funktion `sub` können Zeichen innerhalb einer Zeichenkette ersetzt werden.

- 1671 So ersetzt der folgende Ausdruck `ae` mit `ä`.

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

1672 ## [1] "Friedländer Weg"

- 1673 Wenn allerdings das zu ersetzende Zeichen mehr als einmal vorkommt und beide Instanzen ersetzt werden
 1674 sollen braucht man die Funktion `gsub`.

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

1675 ## [1] "Friedländer Weg und Reinhäuser Landstrasse sind zwei Strassen in Goettingen."

1676 Mit `sub()` wird nur das erste `ae` ersetzt, während `gsub()` alle `ae` mit einem `ä` ersetzt.

```
gsub("ae", "ä", txt)
```

1677 ## [1] "Friedländer Weg und Reinhäuser Landstrasse sind zwei Strassen in Goettingen."

1678 Oft ist der genaue Ausdruck den man finden möchte jedoch Variabel. Beispielsweise möchte man alle Wörter
 1679 mit einem Umlaut oder Zahlen finden möchte, kann man das oft abkürzen. Dafür gibt es reguläre Ausdrücke.
 1680 Wir werden hier nur ein paar beispielhafte Anwendungen besprechen.

1681 Sowohl in den Funktionen `grep()` als auch `(g)sub()` kann mit anstatt dem Muster (immer das erste Argument)
 1682 aus ein regulärer Ausdruck angegeben werden. Mit `[1-9]` sind alle Zahlen von 1 bis 9 gemeint.

1683 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)
```

1684 ## [1] 1 3

1685 Damit lässt sich auch das Problem mit groß- und kleingeschriebenen Wörtern lösen.

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

1686 ## [1] 1 2

1687

1688 Aufgabe 33: Arbeiten mit Text 2

1690 Verwenden Sie den folgenden Vektor:

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

- 1691 1. In wie vielen Wörtern kommt der Doppellaut **au** vor?
1692 2. Ersetzen Sie in allen Wörtern alle **au** mit **_ _**.

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

```
1693 ## [1] 5 6 13
```

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

```
1694 ## [1] "Versicherung" "Methoden"      "Fluss"        "Rudel"       "B_ _m"  
1695 ## [6] "H_ _s"          "Foto"         "Auffahrt"     "Auto"        "Handy"  
1696 ## [11] "Teller"        "Kalender"     "Aufb_ _"
```

1697 11 Arbeiten mit Zeit

1698 Für den Computer bzw. R ist ein Datum erst einmal nichts anderes als ein Text. Für uns ist es sofort klar,
 1699 dass der "13.2.2021" der 13. Februar 2021 ist, für den Computer nicht. Wir müssen R also irgendwie sagen,
 1700 dass die 13 der Tag ist, die 2 der Monat und 2021 das Jahr. Dass der Computer die einzelnen Komponenten
 1701 erkennt, und in einen Datentyp eigens für Zeitformate überführt, nennt man *parsen*¹². Das Arbeiten mit
 1702 Datum und Zeit kann anfangs sehr mühsam sein, aber sobald man einige Grundfertigkeiten erworben
 1703 hat, kann man viele Aufgaben deutlich schneller und effizienter erledigen. Starten Sie am besten gleich mit
 1704 "echten" Zeit-Datentypen und versuchen Sie nicht, sich irgendwie mit Text und numerischen Datentypen
 1705 selbst etwas zu basteln. Der erste Schritt ist immer ein Datum zu *parsen*. Wir verwenden dafür Funktionen
 1706 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)
```

1707 **lubridate** bietet eine Vielzahl von Funktionen zum *parsen* von Datum und Zeit, die sich aus:

- 1708 • y für Jahr,
- 1709 • m für Monat,
- 1710 • d für Tag,
- 1711 • h für Stunde,
- 1712 • m für Minute und
- 1713 • s für Sekunde

1714 zusammen setzten. Alle Funktionen nehmen als erstes Argument ein Textstring. Wenn wir z.B. den String
 1715 2020-01-20 parsen wollen können wir das mit der Funktion **ymd** machen.

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

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

1717 Dabei erkennt **lubridate** in der Regel die Trennzeichen:

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

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

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

1719 ## [1] "2020-01-20"

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

```
1720 ymd("2020 01 20")
```

1720 ## [1] "2020-01-20"

1721 Wenn die die Anordnung der einzelnen Komponenten anders ist, gibt es einfach eine andere Funktion.

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

1722 ## [1] "2020-01-20"

1723 Jetzt stellt sich die Frage, was der Vorteil ist, wenn R ein Datum parst.

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

1724 Wir können jetzt mit `d` arbeiten und einzelne Komponenten extrahieren.

```
day(d)
```

1725 ## [1] 20

```
month(d)
```

1726 ## [1] 1

```
year(d)
```

1727 ## [1] 2020

1728 Oder auch Zeiteinheiten hinzufügen oder abziehen.

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

1729 ## [1] "2020-01-30"

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

1730 ## [1] "2000-01-20"

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

1731 ## [1] "2020-01-21 01:00:00 UTC"

1732

Aufgabe 34: Arbeiten mit Datum und Zeit

- 1735 • Parsen Sie folgende Zeitangaben 23.1.2020, 13.2.1992 20:55:23, Mar/3/97 und 10.7.2020 19:15
 1736 und speichern Sie diese in einen Vektor d.
 1737 • Extrahieren Sie nun aus jedem Element aus d das Jahr und die Stunde.
 1738 • Fügen zu jedem Element in d 10 Tage hinzu.

11.1 Arbeiten mit Zeitintervallen

1740 Mit zwei Zeitpunkten lassen sich Zeitintervalle (Periods) erstellen, dafür können wir die Funktion
 1741 `interval()` aus dem Paket `lubridate` verwenden¹³.

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

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

1742 Wir können jetzt mit `int` arbeiten und beispielsweise das Intervall verschieben,

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

1743 ## [1] 2023-03-18 UTC--2024-03-18 UTC

1744 die Länge des Intervalls berechnen

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

1745 ## [1] 31536000

1746 oder testen, ob ein Datum innerhalb des Intervalls liegt.

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

1747 ## [1] TRUE

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

1748 ## [1] FALSE

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

1749 %within% funktioniert genauso mit Vektoren oder mit mehreren Intervallen. Wir könnten also zwei Intervalle
 1750 definieren (z.B. Ostern und Pfingsten).

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

1751 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
```

1753 ## [1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE TRUE FALSE FALSE

1754 Der Zeit-Datentyp wird auch oft benutzt, um den Zeitbedarf von längeren Berechnungen zu messen.

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

## [1] 0.4999484

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

## [1] 0.7895103
```

1757 11.2 Formatieren von Zeit

1758 Für die Ausgabe in Berichten oder Grafiken soll das Datum oft in einer speziellen Form dargestellt werden.
 1759 Die Funktion `format()` bietet Möglichkeiten ein Datumsobjekt zurück in Text umzuwandeln.

1760 Ein Beispiel wäre `ymd("2021-2-10")` als 10.2.21 auszugeben.

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

## [1] "21.02.21"
```

1762 Dabei handelt sich bei %d.%m.%y um Abkürzungen für die unterschiedlichen Komponenten eines Datumobjekts. Siehe dazu die Hilfeseite von `strptime` (`help(strptime)`).

1764

1765 **Aufgabe 35: Arbeiten mit Intervallen**

1767 Wie viele Einträge aus dem Vektor v1 befinden sich in einem Intervall, das zwischen dem 1.3.2021 und dem
1768 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"
)
```

1769 **11.3 Zeitreihen**

1770 In der Forstbranche haben wir oft mit Zeitreihen zu tun. Zeitreihen sind Variablen, die in zeitlichen
1771 Intervallen vorliegen. In der Regel monatlich, vierteljährig oder jährlich, wobei die Intervalle zwischen den
1772 Messungen immer gleich lang sind. Wiederholungsmessungen von Forstinventuren (Forsteinrichtungen, Be-
1773 triebseinventuren, die meisten forstl. Experimente, die BWI, ...) sind also in der Regel keine Zeitreihen in
1774 engeren Sinne, turnusmäßig vermessene Versuchsflächen, wie sie z. B. die Versuchsanstalten unterhalten oder
1775 jährlich gemeldete Holzpreise jedoch schon.

1776 Statistisch gesehen haben Zeitreihen Besonderheiten gegenüber anderen Variablen, da Sie in erster Linie von
1777 Ihrer eigenen Vergangenheit abhängen (autokorreliert sind) und auch die Abhängigkeit anderer Variablen in
1778 der Regel nur sinnvoll ist, wenn deren Vergangenheit mitberücksichtigt wird (cross-correlation). Konven-
1779 tionale Statistik ist oft nicht ausreichend, um Zeitreihen zu analysieren. Angefangen mit der Datendarstellung
1780 gibt es spezifische Zeitreihen-Funktionen, welche auch alle in R integriert sind. Aus diesem Grund sollten Sie
1781 Zeitreihen als solche speichern. Viele Funktionen erkennen den Datentyp und führen entsprechend spezifische
1782 Operationen durch. Laden wir z. B. die Holzpreise für Fichte 2b (das sog. Leitsortiment), das Holzaufkommen
1783 dieses Sortiments und die Preise für Nadelholz vom statistischen Bundesamt¹⁴. Wir laden die Daten
1784 zunächst als csv:

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

1785 Mit der Funktion `ts` werden die Daten in ein Zeitreihenobjekt überführt (geparst). Die Spalte mit den
1786 Jahren ist dann nicht mehr nötig, da die Informationen zur Zeit keine eigene Variable mehr sind, sondern
1787 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

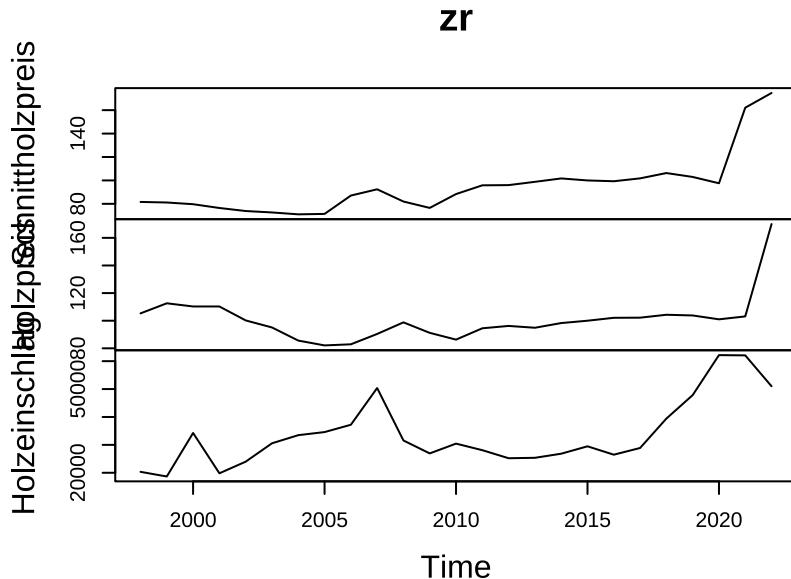
¹⁴Sie können sich die Daten auch selbst über die Website laden oder das Paket `wiesbaden` verwenden, um die Daten direkt in R zu laden. Jedoch müssen Sie sich zuerst registrieren

1788 ## [1] "double"

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

1789 Die wichtigsten Argumente sind - **data** Vektor oder Matrix, der nur die Daten enthält - **start** Startzeitpunkt -
 1790 **frequency** Anzahl der Beobachtungen pro Zeiteinheit, also z. B. 12 bei monatlichen oder 4 bei quartalsweisen
 1791 Erhebungen

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

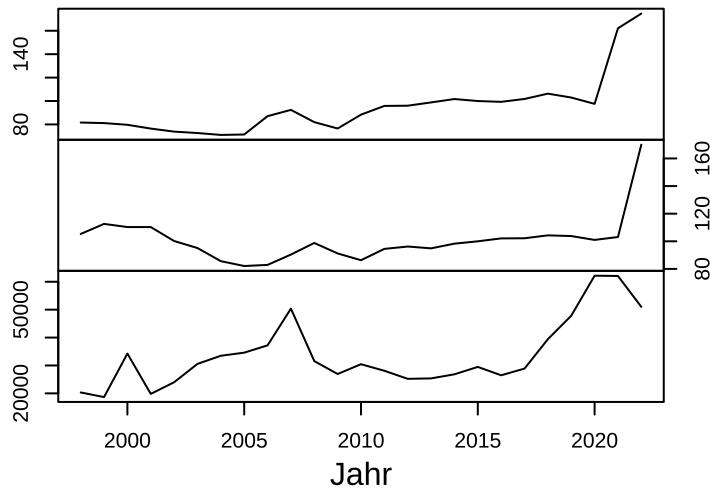


1792

1793 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")
```

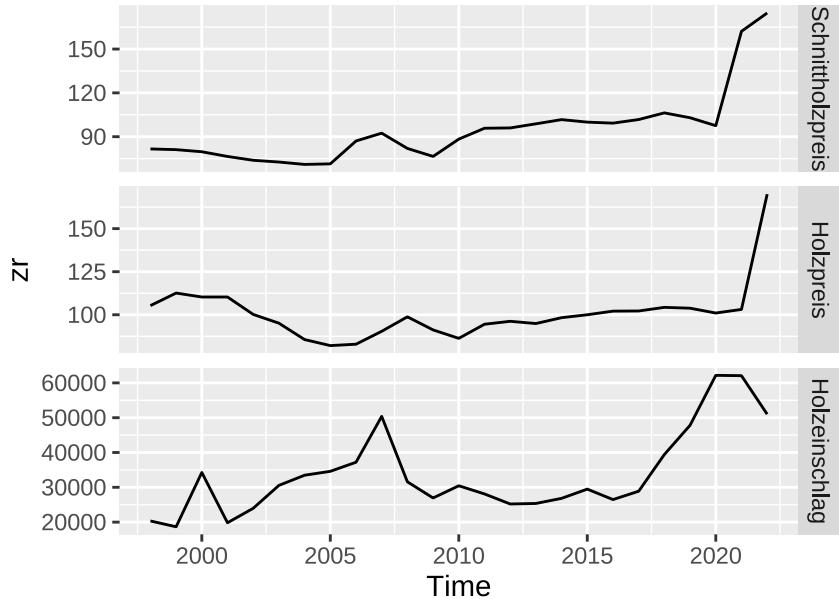
Holzmarktentwicklung seit 1998



1794

- 1795 Das Paket `forecast` ermöglicht automatisierte Zeitreihenplots im `ggplot2` Stil. Damit ist auch das Problem
1796 der y-Achsenbeschriftungen gelöst.

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

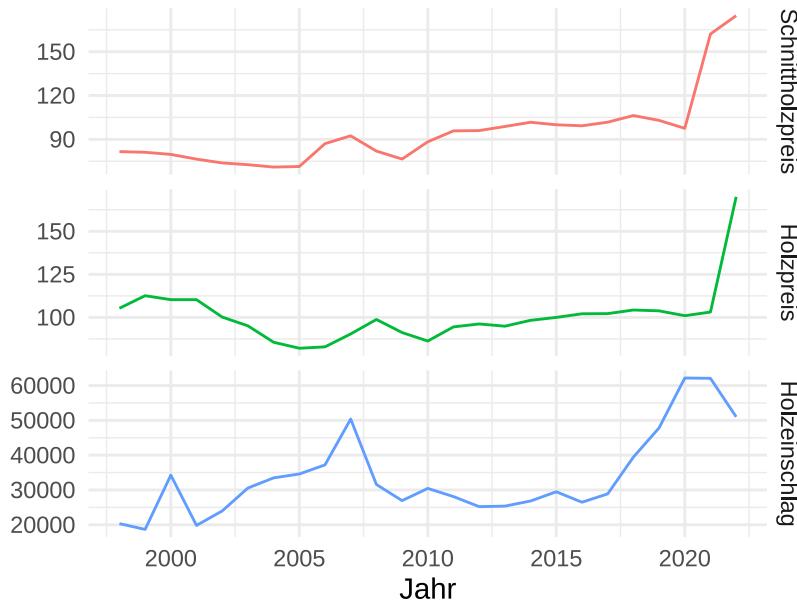


1797

- 1798 Wir können die Abbildung im `ggplot2` Stil ändern. Hier nur ein Minimalbeispiel zur Veranschaulichung.
1799 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

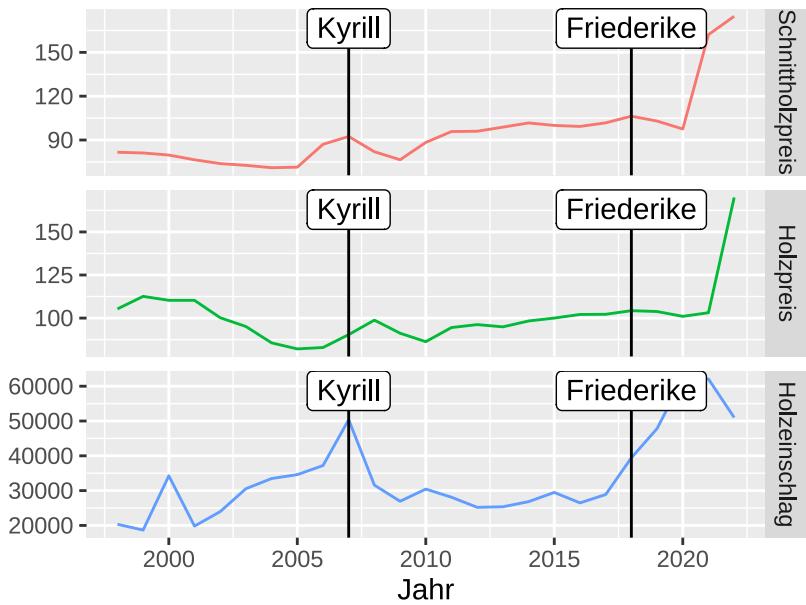
zr_autoplot + theme_minimal()
```



1800

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

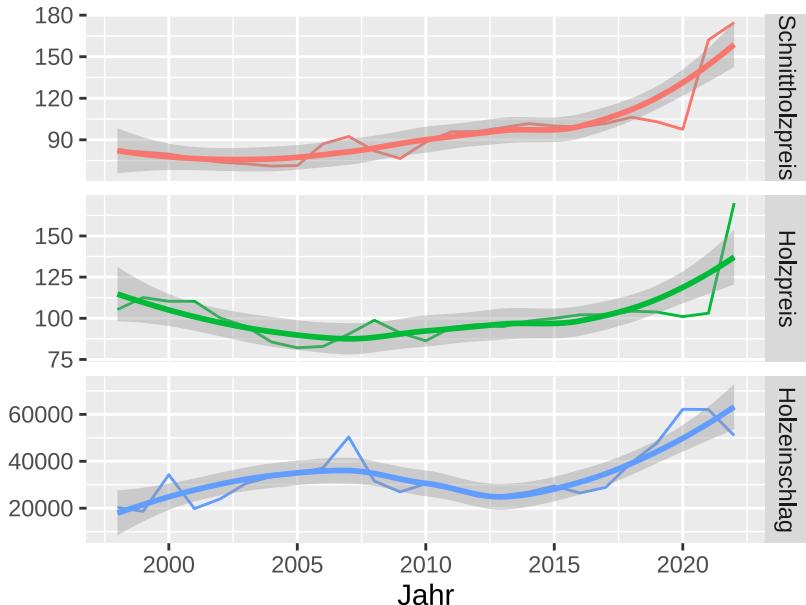
z2 + annotate(x = 2007, y = +Inf, label = "Kyrill", vjust = 1, geom = "label") +
  annotate(x = 2018, y = +Inf, label = "Friederike", vjust = 1, geom = "label")
```



1801

- 1802 Eine Trendlinie macht hier (sowie generell in Zeitreihendaten) offensichtlich keinen Sinn. Daher verwenden
 1803 wir den sog. Loess-Glättter, um einen Trend in die Daten zu legen. Der Loess-Glättter ist ein sehr flexible
 1804 Kurve.

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



1805

1806 12 Aufgaben Wiederholen (for-Schleifen)

1807 Um einfache Programme zu schreiben, müssen Sie den Ablauf der Programmcodes kontrollieren können.
 1808 Kontrollieren bedeutet in diesem Zusammenhang, dass Codeabschnitte nur unter definierten Bedingungen
 1809 ablaufen. Sie programmieren also zwei Sachen. 1) den Code selbst und 2), die Bedingungen die erfüllt sein
 1810 müssen, damit der Code ausgeführt wird. Der Code muss do generisch geschrieben sein, dass er komplett
 1811 durchläuft, auch wenn unterschiedliche Dateneingaben gemacht werden. Diese Kontrollbedingungen ermög-
 1812 lichen es Ihnen generisch zu programmieren. Sie schreiben Ihren Code also nicht speziell maßgeschneidert
 1813 für ein Problem, sondern so generell, dass er für mehrere Auswertungen funktioniert. Um dies zu gewähr-
 1814 leisten, müssen Sie bestimmte Situationen vorhersehen und abfangen. Hierbei helfen Ihnen Kontrollstruktu-
 1815 ren (**Control Flow**). Grundsätzlich gibt es **Control Flow** Funktionen zur Wiederholung von Codeblöcken
 1816 (Schleifen) und logische Bedingungen (bedingte Anweisung).

1817 12.1 Schleifen

1818 Bis jetzt wurden alle Skripte einfach der Reihe nach abgearbeitet und zwischendurch bestimmte Programm-
 1819 teile, je nach Situation, selbstständig ausgeführt oder übersprungen. Mit einer Schleife kann man erreichen,
 1820 dass eine Gruppe von Befehlen (der sog. Schleifenrumpf) mehrfach abgearbeitet wird, zum Beispiel wenn
 1821 bestimmte Auswertungsschritte auf mehrere Datensätze oder Variablen angewendet oder Funktionen mit un-
 1822 terschiedlichen Parametern oder Startwerten aufgerufen werden sollen. Weitere Anwendungsmöglichkeiten
 1823 sind iterative Algorithmen, in denen die Eingabewerte des aktuellen Iterationsschrittes von einem vorherigen
 1824 abhängig sind. Besonders in Simulationen kommen Schleifen häufig zum Einsatz, da große Anzahlen von
 1825 Wiederholungen benötigt werden.

1826 Man unterscheidet zwischen zwei Arten von Schleifen: Bei den **for()**-Schleifen steht die Anzahl der Wieder-
 1827 holungen schon beim Eintritt in die Schleife fest, während die **while()**-Schleifen so lange ausgeführt werden,
 1828 bis eine Bedingung nicht mehr wahr ist. Mit der Funktion **break** wird eine Schleife abgebrochen und die
 1829 Programmausführung wird nach der Schleife fortgesetzt.

1830 Die wesentlichen Befehle sind

- 1831 • **for (i in X) {Code}**

1832 Wiederhole den Code im “Schleifenrumpf” für jedes Element aus X.

- 1833 • **while(Bedingung) {Code}**

1834 Wiederhole den Code im “Schleifenrumpf” so lange die logische Bedingung erfüllt ist.

- 1835 • **break()**

1836 Brich die Schleife ab. **break()** muss im Schleifenrumpf an der richtigen Stelle ausgeführt werden. Gute
 1837 Praxis ist jedoch, die for oder while Bedingungen, dass kein **break()**nötig ist, da **break()** anfällig für
 1838 Programmierfehler ist.

1839 **12.1.1 Wiederholen von Befehlen mit `for()`.**

1840 Steht vor Beginn der Schleife fest wie viele Schleifendurchgänge benötigt werden, wenn zum Beispiel in
 1841 einer Simulationen 99 Realisierungen erzeugt oder alle Elemente eines Vektors verarbeitet werden sollen,
 1842 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)
}
```

1843 ## [1] 1
 1844 ## [1] 2
 1845 ## [1] 3

1846 Das `i` steht in diesem Beispiel für die Schleifen-Variable. Sie muss nicht `i` heißen, sondern kann jeden
 1847 zulässigen Namen annehmen. Das `X` steht für einen existierenden Vektor oder eine existierende Liste bzw.
 1848 einen Ausdruck, der ein solches Objekt liefert (der Objektname ist ebenfalls frei wählbar). `for` und `in` sind
 1849 Schlüsselworte, sie müssen, ebenso wie die runden Klammern, vorhanden sein.

1850 Im ersten Durchgang erhält die Schleifen-Variable `i` den ersten Wert von `X` und der Schleifenrumpf wird
 1851 mit diesem Wert ausgeführt. Die Variable `i` nimmt nacheinander so lange die Werte von `X` an, bis ihr alle
 1852 Elemente zugewiesen wurden.

1853 Das folgende Beispiel wird zwar besser durch die entsprechende Vektoroperation gelöst, zeigt aber sehr
 1854 deutlich die Arbeitsweise der `for`-Schleife.

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

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

1855 ## [1] 4
 1856 ## [1] 9
 1857 ## [1] 25

1858

1859 **Aufgabe 36: Schleifen 1**

 1860

1861 Verwenden Sie den Vektor `k <- c(1, 3, 9, 12, 15)` und schreiben Sie folgende `for`-Schleifen:

1862 1. Eine Schleife, die jedes Element aus `k` ausgibt.

- 1863 2. Eine Schleife, die zu jedem Element aus `k` 10 addiert und den neuen Wert ausgibt.
- 1864 3. Eine Schleife wie in 2), aber der neue Wert ($k + 10$) soll jetzt nicht mehr ausgegeben werden, sondern
1865 in `k10` gespeichert werden. Stellen Sie sicher, dass `k10` wieder von der Länge 5 ist.

1866

1867 Die Funktion `for()` ermöglicht es, einen Befehl beliebig oft zu wiederholen. Z.B. der folgende Ausdruck zieht
1868 10-Mal eine Stichprobe der Größe 1 aus dem Vektor `v`. Beachten Sie, dass die Schleifen-Variable `i` selbst gar
1869 nicht im Schleifenrumpf vorkommt. Das Ziel dieser Schleife ist nicht die Elemente des Vektors abzuarbeiten,
1870 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))
}
```

1871 ## [1] 3
1872 ## [1] 1
1873 ## [1] 3
1874 ## [1] 3
1875 ## [1] 2
1876 ## [1] 3
1877 ## [1] 2
1878 ## [1] 2
1879 ## [1] 1
1880 ## [1] 4

1881 Auf gleiche Weise kann man auch über die Variablen eines Dataframes iterieren¹⁵. Das folgende Beispiel
1882 hat zum Ziel die Funktionsweise von Schleifen zu verdeutlichen. Schleifen haben in R jedoch den Nachteil,
1883 dass sie sehr langsam operieren. Wenn es geht, sollte man Alternativen verwenden. Die Funktionsweise
1884 wiederholender Auswertungen wird jedoch mit `for`-Schleifen deutlicher. Aus diesem Grunde werden wir uns
1885 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.

```

1886 ## [1] "Buche"
1887 ## [1] 52
1888 ## [1] "Eiche"
1889 ## [1] 64
1890 ## [1] "Eiche"
1891 ## [1] 62
1892 ## [1] "Buche"
1893 ## [1] 85

```

1894

1895 Aufgabe 37: for-Schleife

1897 Lesen Sie den Datensatz `bhd_1.txt` ein und verwenden Sie eine `for`-Schleife.

- 1898 • Ziehen Sie 500-Mal je 35 Beobachtungen für den BHD.
- 1899 • Berechnen Sie jeweils den Mittelwert aus den 35 Werten.
- 1900 • Speichern Sie die 500 Mittelwerte in einem neuen Vektor `mittel`.
- 1901 • Wie ist die Verteilung dieser 500 Mittelwerte zu interpretieren?

1902 **12.1.2 Wiederholen von Befehlen mit `while()`**

1903 Die `while`-Schleifen finden Anwendung, wenn die Anzahl der zu durchlaufenden Wiederholungen vorher
 1904 nicht bekannt ist, wie zum Beispiel bei Iterationsverfahren, die bis zu einer gewissen Genauigkeit durchlaufen
 1905 werden sollen. Die `while`-Schleife besteht in R aus dem Schlüsselwort `while()` und einer Bedingung in runden
 1906 Klammern.

```

while (Bedingung) {
  # Schleifenrumpf
}

```

1907 Sie ist in der praktischen Programmierung nicht so relevant wie die `for`-Schleife. Sie sei deshalb hier nur
 1908 kurz erwähnt. Die Abbruchbedingung wird jedes Mal geprüft bevor der Schleifenrumpf durchlaufen wird.
 1909 Die Bedingung wird ausgewertet und wenn diese `TRUE` ist, wird der Schleifenrumpf ausgeführt und danach
 1910 erneut die Bedingung überprüft. Ist die Bedingung nicht erfüllt, wird der Schleifenrumpf nicht durchgeführt
 1911 und die Schleife beendet. Ist die Bedingung bereits vor Eintritt in die Schleife nicht erfüllt, wird die Schleife
 1912 gar nicht erst durchlaufen.

1913 Da `while`-Schleifen also so lange ausgeführt werden, bis die Bedingung nicht mehr erfüllt ist, kann eine
 1914 Endlosschleife entstehen. Dies kann passieren, wenn man nicht sauber programmiert hat. Wenn innerhalb
 1915 der Schleife nicht dafür gesorgt wird, dass die Bedingung irgendwann nicht mehr erfüllt wird, so läuft die
 1916 Schleife immer weiter. Steckt R in einer Schleife fest und reagiert nicht mehr, kann der Befehl unter Linux
 1917 mit `Strg + C` und unter Windows mit `Esc` abgebrochen werden. Alternativ können Sie auf das rote STOP
 1918 Symbol über der Konsole klicken.

1919 **12.2 Bedingte Ausführung von Codeblöcken**

1920 Innerhalb eines Skripts ist es mitunter notwendig je nach aktueller Situation unterschiedlich fortzufahren.
 1921 Die Situation wird mit einem logischen Ausdruck, einer sogenannten Bedingung, geprüft. Je nachdem, ob
 1922 die Bedingung wahr (TRUE) oder falsch (FALSE) ist, werden unterschiedliche Programmteile ausgeführt, der
 1923 jeweils andere Teil bleibt unberücksichtigt. Danach wird in jedem Fall die Programmausführung, mit den
 1924 auf die bedingte Anweisungen folgenden Anweisung, fortgesetzt. In R kann dies mit dem if-else-Konstrukt
 1925 realisiert werden, welches aus den Schlüsselwörtern `if()` und `else` sowie der in runde Klammern gefassten
 1926 Bedingung besteht.

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

1927 Im folgenden Beispiel sollen die bisher abstrakt beschriebenen Vorgänge praktische Anwendung finden. In
 1928 dem Beispiel wird zunächst durch zufälliges Ziehen einer Zahl aus dem Bereich eins bis sechs ein Würfelwurf
 1929 simuliert. Anschließend wird der Würfelwurf mit einem if-Ausdruck kommentiert. Ist die Bedingung, es wurde
 1930 eine Sechs gewürfelt, erfüllt, wird der Code innerhalb der geschweiften Klammern ausgeführt, ansonsten wird
 1931 der Klammerinhalt ignoriert.

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

1932 In den meisten Fällen ist es für R irrelevant, ob sich zwischen den verschiedenen Elementen Leerzeichen oder
 1933 Zeilenumbrüche befinden. Bei dem else-Ausdruck dagegen wird ein Fehler erzeugt, wenn vor dem `else` nicht
 1934 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.")
}
```

1935 `## [1] "Beim nächsten Wurf klappt's bestimmt."`

1936

1937 **Aufgabe 38: Bedingte Programmierung**

- 1939 • Wenn keine 6 gewürfelt wurde, lassen Sie zusätzlich ausgeben welche Zahl gewürfelt wurde.
- 1940 • Wiederholen Sie den Würfelwurf 10 Mal.

1941 13 (R)markdown

1942 13.1 Markdown Grundlagen

1943 Die Idee von Markdown ist, einfach Text strukturieren zu können und das ganze ohne umfangreiche Pro-
 1944 grammre zu erstellen. Im nächsten Abschnitt sehen wir dann, wie man Markdown und R-Code verbinden
 1945 kann. Hier soll es jedoch erst einmal darum gehen, die einfachsten Bausteine von Markdown vorzustellen.

1946 Am Anfang jedes Dokuments kommt eine Präambel. Diese fängt mit --- an und hört auch wieder mit ---
 1947 auf. Innerhalb der Präambel können dann Metainformationen über das Dokument festgelegt werden, dies
 1948 beinhaltet im einfachsten Fall: Titel, Autor und Datum. Das würde dann so aussehen:

```
1949 ---
1950 title: "Ein Titel"
1951 author: "Der, der es geschrieben hat"
1952 date: "März 2021"
1953 ---
```

1954 Danach folgt strukturierter und formatierter Text. Verschiedene Hierarchieebenen von Überschriften können
 1955 mit der Anzahl an # festgelegt werden. So ist eine Überschrift erster Ordnung # Kapitel eine Überschrift
 1956 zweiter Ordnung ## Unterkapitel usw.

1957 Listen können erstellt werden, wenn man am Anfang jeder Zeile ein – oder 1. schreibt.

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

1961 wird zu

```
1962   • Erster Eintrag
1963   • Zweiter Eintrag
1964   • Dritter Eintrag
```

1965 Eine zentrale Idee von Markdown ist es Text einfach zu formatieren. Werden eine oder mehrere Wörter mit
 1966 zwei Sternchen (**) eingefasst wird dieser Text **fett** dargestellt. Also aus **wichtig** wird **wichtig**. Das
 1967 gleiche funktioniert auch mit *kursiven* Text, jedoch muss man hier noch ein Sternchen verwenden, also aus
 1968 *kursiv* wird *kursiv*. Soll ein text fett und kursiv sein, kann man drei Sternchen verwenden. Aus ***sehr
 1969 wichtig*** wird dann **sehr wichtig**.

1970 Weitere Elemente wie Links oder Abbildungen können einfach eingebunden werden. Links werden mit [Link
 1971 text](url) in den Text integriert. Beispielsweise ist eine gute Idee bei [stackoverflow](#) bei Problemen nach
 1972 einer Lösung zu suchen. Dieser Link wurde mit [stackoverflow](www.stackoverflow.com) erstellt.

1973 Für Abbildungen gibt es einen ganz ähnlichen Syntax. Mit ![Das R Logo](abb/r_logo.png) wird die
 1974 Abbildung r_logo.png eingebunden mit der Beschriftung: "Das R Logo".



Abbildung 9: Das R Logo

1975

1976 **Aufgabe 39: Arbeiten mit markdown**

1978 Verwenden Sie das folgende Markdowndokument:

```

1979 ---
1980 title: "Dokument"
1981 author: "Ihr Name"
1982 date: "März 2021"
1983 ---
1984
1985 # Einleitung
1986
1987 # Methoden

```

- 1988 1. Kopieren Sie die Vorlage in ein Dokument, das `test.md` heißt.
- 1989 2. Fügen Sie zwei Überschriften zweiter und dritter Ordnung hinzu.
- 1990 3. Fügen Sie einen *kursiven* Text hinzu.
- 1991 4. Fügen Sie einen Link zu einer Website hinzu.
- 1992 5. Kompilieren Sie die Datei, indem Sie in Rstudio auf Preview drücken (Abbildung 10).

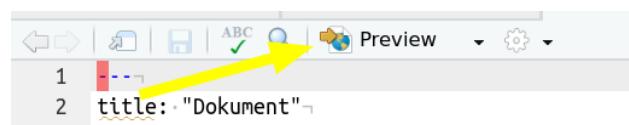


Abbildung 10: Kompilieren einer md-Datei.

1993 **13.2 R und Markdown**

1994 Markdown macht es bereits einfach Textdokumente und Dokumentationen zu verfassen, aber die wirkliche
 1995 Stärke liegt in der Möglichkeit R und Markdown zu kombinieren. Man spricht dann von Rmarkdown. Ein
 1996 weiteres Strukturelement, das wir noch nicht kennen gelernt haben, sind Code-Blöcke.

```

1997   ```
1998 a <- 1:10
1999 a[1]
2000   ```
2001 erzeugt

2002 a <- 1:10
2003 a[1]

2004 Momentan wird noch kein Code ausgeführt, sondern lediglich der Code, als Code dargestellt. Rmarkdown
2005 bietet nun die Möglichkeit Code beim kompilieren16 auszuführen. Dafür müssen wir nur einen Code-Block
2006 als R-Code-Block kennzeichnen.

```

```

2007   ``{R}
2008 a <- 1:10
2009 a[1]
2010   ```

```

2011 erzeugt

```
a <- 1:10
a[1]
```

```
2012 ## [1] 1
```

```
2013 Beachte, die Variable a wird beim kompilieren erzeugt und steht dann R zur Verfügung. R-Code-Blöcke
2014 werden auch als Code Chunks bezeichnet. Diese Chunks können sehr genau angesprochen und angepasst
2015 werden. Einige wichtige Argumente sind:
```

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

2019

2020 Aufgabe 40: Arbeiten mit Rmarkdown

```
2022 Erstellen Sie eine neue Rmarkdown Datei mit dem Namen test1.Rmd. Erstellen Sie zwei Code-Chunks.
2023 Der erste soll nicht angezeigt werden und darin werden die Daten geladen (bhd_1.txt). Im zweiten Chunk
2024 plotten Sie das Alter der Bäume gegen den BHD. Was passiert mit dem Plot, wenn Sie die Datei kompilieren
2025 (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.

2026 14 Räumliche Daten in R

2027 14.1 Was sind räumliche Daten

2028 Räumliche Daten sind Beobachtungen, wie wir sie schon oft gesehen haben, mit einem räumlichen Bezug. Der
 2029 Unterschied zu nicht räumlichen Daten liegt darin, dass räumliche Daten eindeutig im *Raum* verortet werden
 2030 können. Häufig werden sogenannte Geoinformationssysteme (GIS) zum Arbeiten mit räumlichen verwendet.
 2031 R kann in vielerlei Hinsicht wie ein GIS eingesetzt werden und hier werden einige Grundfunktionalitäten
 2032 dafür besprochen. Räumliche Daten werden in zwei unterschiedliche Datentypen unterteilt: Vektor- und Ras-
 2033 terdaten. Vektordaten modellieren einzelne Objekte (= *Features*). Rasterdaten modellieren eine Oberfläche.
 2034 Vektordaten bestehen aus zwei Komponenten: 1) einer Geometrie, die die Form und Lage der Daten definiert
 2035 und 2) Attributen, den tatsächlichen Daten. Räumliche Daten werden oft als *Features* bezeichnet. Ein Feature
 2036 ist die räumliche Einheit einer Beobachtung. Je nach Art der räumlichen Daten können Features entweder
 2037 Punkte (z.B. ein Baum), Linien (z.B. eine Straße) oder Polygone (z.B. ein See) sein. Auch können mehrere
 2038 Geometrien zu einem Feature zusammengefasst werden. Ein Beispiel wäre eine Beobachtung für ein Land,
 2039 das aber aus mehreren Polygonen bestehen kann (z.B. Festland und Inseln). Features können dann weitere
 2040 Attribute (= Attributdaten) haben, z.B. eine ID, Name oder was auch immer man gemessen hat.
 2041 Rasterdaten bestehen aus einer Oberfläche von gleichgroßen Kacheln (= *Pixel*), die ein Gebiet abdecken.
 2042 Meist sind Pixel viereckig, aber das ist keine Voraussetzung. Dabei hat jedes Pixel einen Wert (das kann
 2043 auch ein fehlender Wert sein). Typische Beispiele für Rasterdaten sind Landnutzung oder Seehöhen.
 2044 In R kann sowohl mit Vektor- als auch mit Rasterdaten gearbeitet werden. Für Vektordaten bietet sich das
 2045 Paket `sf` an und für Rasterdaten das Paket `raster`.

2046 14.2 Koordinatenbezugssystem

2047 Eine der Herausforderungen für räumliche Daten ist die eindeutige Verortung im Raum. Dazu braucht man
 2048 ein *Koordinatenbezugssystem* (KBS). Einem KBS liegt ein mathematisches Modell der Erde zugrunde. Die
 2049 Details zu KBS werden schnell relativ kompliziert und wir beschränken uns hier lediglich darauf, wie KBS
 2050 verwendet werden können. Dazu müssen wir zwei Fälle unterscheiden: 1) einem Datensatz ein KBS zuweisen
 2051 und 2) Transformation des KBS eines Datensatzes in ein anderes KBS. Die technischen Details werden in
 2052 den folgend Abschnitten besprochen. Für beide Aufgaben müssen wir auf ein KBS verweisen können, ein
 2053 einfacher Ansatz dafür sind die sogenannten *EPSG-Codes*¹⁷.

¹⁷EPSG steht für European Petrol Survey Group

2054 14.3 Vektordaten in R

- 2055 Das Paket `sf` stellt Klassen zum Abbilden von Features zur verfügen, die dann in einem `data.frame` als
 2056 Liste gespeichert werden können. In der Regel erstellen wir Features nicht individuell, sondern lesen diese aus
 2057 externen Datenquellen (z.B. ESRI Shapefile) ein. Zum besseren Verständnis, erstellen wir es einmal manuell.
 2058 Wir haben die Koordinaten für drei Städte (Göttingen, Hannover und Berlin) als geografische Koordinaten
 2059 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))
```

- 2060 Daraus können wir jetzt eine Geometriespalte für einen `data.frame` erstellen

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

- 2061 Somit haben wir die Geometrie in der Variable `geom` gespeichert, aber noch keine dazugehörigen Attribut-
 2062 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)
)
```

- 2063 In einem letzten Schritt möchten wir jetzt die Geometrie (`geom`) und die Attributdaten (`attr`) zusammen-
 2064 führen.

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

```
2065 ## Simple feature collection with 3 features and 3 fields
2066 ## Geometry type: POINT
2067 ## Dimension: XY
2068 ## Bounding box: xmin: 9.732 ymin: 51.5413 xmax: 13.405 ymax: 52.52
2069 ## Geodetic CRS: WGS 84
2070 ##           name      bundesland   einwohner          geom
2071 ## 1 Goettingen Niedersachsen    119000  POINT (9.9158 51.5413)
2072 ## 2 Hannover Niedersachsen    532000  POINT (9.732 52.3759)
2073 ## 3 Berlin      Berlin     3650000  POINT (13.405 52.52)
```

- 2074 Wir können nun mit `staedte` genau so arbeiten wie mit jedem anderen `data.frame` und die Geometrien
 2075 werden immer ‘berücksichtigt’. Zusätzlich kann man eine Reihe von geometrischen Operationen durchführen.

2076 Wenn ein `data.frame` Punkten hat, kann man dies relative einfach mit der Funktion `st_as_sf()` "räumlich"
 2077 machen. Für das vorherige Beispiel würden wir zuerst einen `data.frame` mit allen Informationen (zur
 2078 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)
)
```

2079 Dann kann man mit der Funktion `st_as_sf()` weiter arbeiten:

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

2080 14.4 Arbeiten mit Vektordaten

2081 Es gibt sehr viele Funktionen, um mit räumlichen Daten zu arbeiten, von denen wir hier einige vorstellen.

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

2082 ## Coordinate Reference System:
2083 ##   User input: EPSG:4326
2084 ##   wkt:
2085 ## GEOCRS["WGS 84",
2086 ##   ENSEMBLE["World Geodetic System 1984 ensemble",
2087 ##     MEMBER["World Geodetic System 1984 (Transit)"],
2088 ##     MEMBER["World Geodetic System 1984 (G730)"],
2089 ##     MEMBER["World Geodetic System 1984 (G873)"],
2090 ##     MEMBER["World Geodetic System 1984 (G1150)"],
2091 ##     MEMBER["World Geodetic System 1984 (G1674)"],
2092 ##     MEMBER["World Geodetic System 1984 (G1762)"],
2093 ##     MEMBER["World Geodetic System 1984 (G2139)"],
2094 ##     ELLIPSOID["WGS 84",6378137,298.257223563,
2095 ##       LENGTHUNIT["metre",1]],
2096 ##     ENSEMBLEACCURACY[2.0]],
2097 ##   PRIMEM["Greenwich",0,
2098 ##     ANGLEUNIT["degree",0.0174532925199433]],
2099 ##   CS[ellipsoidal,2],
2100 ##     AXIS["geodetic latitude (Lat)",north,
2101 ##       ORDER[1],
```

```

2102 ##           ANGLEUNIT["degree",0.0174532925199433]],  

2103 ##           AXIS["geodetic longitude (Lon)",east,  

2104 ##             ORDER[2],  

2105 ##             ANGLEUNIT["degree",0.0174532925199433]],  

2106 ##           USAGE[  

2107 ##             SCOPE["Horizontal component of 3D system."],  

2108 ##             AREA["World."],  

2109 ##             BBOX[-90,-180,90,180]],  

2110 ##             ID["EPSG",4326]]
```

2111 Wenn wir jetzt zu einem anderen KBS (z.B. EPSG:3035, ein europäisches projiziertes KBS) umrechnen
 2112 möchten, können wir das mit

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

```

2113 ## Coordinate Reference System:  

2114 ##   User input: EPSG:3035  

2115 ##   wkt:  

2116 ## PROJCRS["ETRS89-extended / LAEA Europe",  

2117 ##   BASEGEOGCRS["ETRS89",  

2118 ##     ENSEMBLE["European Terrestrial Reference System 1989 ensemble",  

2119 ##       MEMBER["European Terrestrial Reference Frame 1989"],  

2120 ##       MEMBER["European Terrestrial Reference Frame 1990"],  

2121 ##       MEMBER["European Terrestrial Reference Frame 1991"],  

2122 ##       MEMBER["European Terrestrial Reference Frame 1992"],  

2123 ##       MEMBER["European Terrestrial Reference Frame 1993"],  

2124 ##       MEMBER["European Terrestrial Reference Frame 1994"],  

2125 ##       MEMBER["European Terrestrial Reference Frame 1996"],  

2126 ##       MEMBER["European Terrestrial Reference Frame 1997"],  

2127 ##       MEMBER["European Terrestrial Reference Frame 2000"],  

2128 ##       MEMBER["European Terrestrial Reference Frame 2005"],  

2129 ##       MEMBER["European Terrestrial Reference Frame 2014"],  

2130 ##       ELLIPSOID["GRS 1980",6378137,298.257222101,  

2131 ##         LENGTHUNIT["metre",1]],  

2132 ##       ENSEMBLEACCURACY[0.1]],  

2133 ##     PRIMEM["Greenwich",0,  

2134 ##       ANGLEUNIT["degree",0.0174532925199433]],  

2135 ##     ID["EPSG",4258]],  

2136 ##     CONVERSION["Europe Equal Area 2001",  

2137 ##       METHOD["Lambert Azimuthal Equal Area",  

2138 ##         ID["EPSG",9820]],  

2139 ##       PARAMETER["Latitude of natural origin",52,  

2140 ##         ANGLEUNIT["degree",0.0174532925199433],
```

```

2141 ##           ID["EPSG",8801]],
2142 ##           PARAMETER["Longitude of natural origin",10,
2143 ##                         ANGLEUNIT["degree",0.0174532925199433],
2144 ##                         ID["EPSG",8802]],
2145 ##           PARAMETER["False easting",4321000,
2146 ##                         LENGTHUNIT["metre",1],
2147 ##                         ID["EPSG",8806]],
2148 ##           PARAMETER["False northing",3210000,
2149 ##                         LENGTHUNIT["metre",1],
2150 ##                         ID["EPSG",8807]]],
2151 ##           CS[Cartesian,2],
2152 ##             AXIS["northing (Y)",north,
2153 ##               ORDER[1],
2154 ##               LENGTHUNIT["metre",1]],
2155 ##             AXIS["easting (X)",east,
2156 ##               ORDER[2],
2157 ##               LENGTHUNIT["metre",1]],
2158 ##           USAGE[
2159 ##             SCOPE["Statistical analysis."],
2160 ##             AREA["Europe - European Union (EU) countries and candidates. Europe - onshore and offshore: BBOX[24.6,-35.58,84.73,44.83]"],
2161 ##             BBOX[24.6,-35.58,84.73,44.83]],
2162 ##             ID["EPSG",3035]]

```

2163 Die Funktion `st_buffer()` erlaubt es Features zu puffern, mit `st_distance()` kann die Distanz zwischen
2164 Features berechnet werden, mit `st_area()` kann die Fläche eines Features berechnet werden.

2165 Funktionen wie `st_intersection()`, `st_union()` und `st_difference()` erlauben es geometrische Opera-
2166 tionen zwischen unterschiedlichen Features zu berechnen. Für eine ausführliche Diskussion siehe auch hier:
2167 <https://geocompr.robinlovelace.net/geometric-operations.html>.

2168 Normalerweise lesen wir Daten von externen Datenquellen (z.B. ESRI Shapefiles). Das geht mit der Funktion
2169 `st_read()`.

2170 14.5 Rasterdaten in R

2171 Für Rasterdaten gibt es das R-Paket `raster`. Auch hier wollen wir uns wieder auf einige Grundfunktionali-
2172 täten konzentrieren. Diese umfassen das Einlesen, Zuschneiden, Rechnen und Abfragen von Rastern.
2173 Mit der Funktion `raster()` kann ein Raster in R eingelesen werden.

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

2174 `dem` steht für *Digital Elevation Model* und ist ein Raster mit den Seehöhen in Niedersachsen mit einer
2175 500-m-Auflösung. Wir können diese mit der Funktion `res()`¹⁸ abfragen.

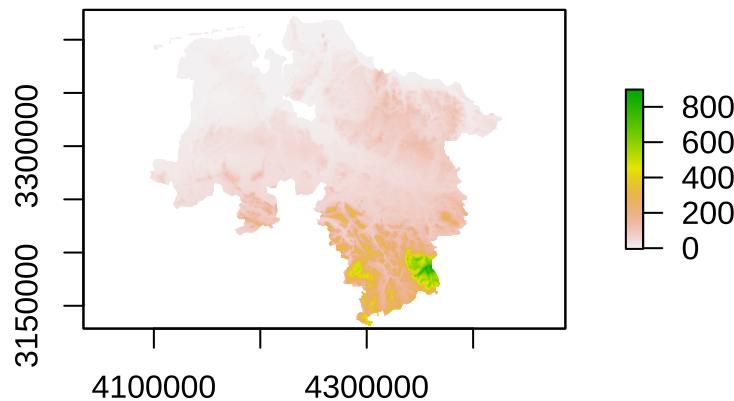
¹⁸kurz für *resolution* also Auflösung.

```
res(dem)
```

2176 ## [1] 500 500

2177 Bzw. wir können den Raster auch plotten.

```
plot(dem)
```



2178

2179 Wenn wir den Raster `dem` auf ein Gebiet zuschneiden wollen (z.B. Göttingen), müssen wir drei Schritte
2180 durchführen. Als erstes müssen wir ein Shapefile für Göttingen einlesen.

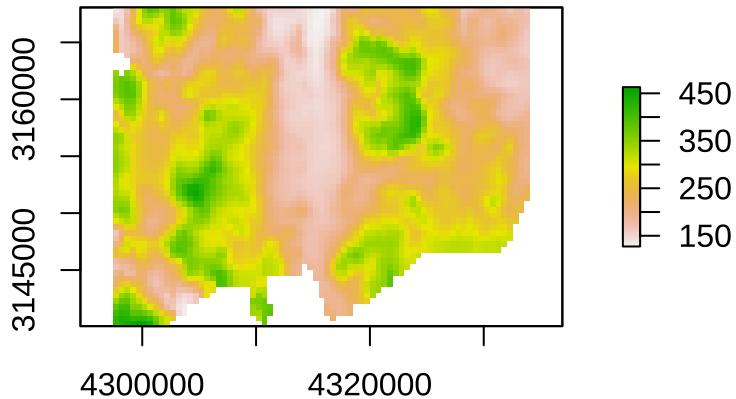
```
goe <- st_read(here::here("data/goettingen/stadt_goettingen.shp"))
```

2181 Dann müssen wir sicherstellen, dass sowohl der Raster `dem` als auch das `sf`-Objekt `goe` im selben KBS sind.
2182 Es bietet es sich in der Regel an, das KBS des Vektors zu transformieren. Mit der Funktion `projectRaster()`
2183 kann das KBS eines Raster transformiert werden.

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

2184 Mit der Funktion `crop()` kann der Raster `dem` auf Göttingen zugeschnitten werden.

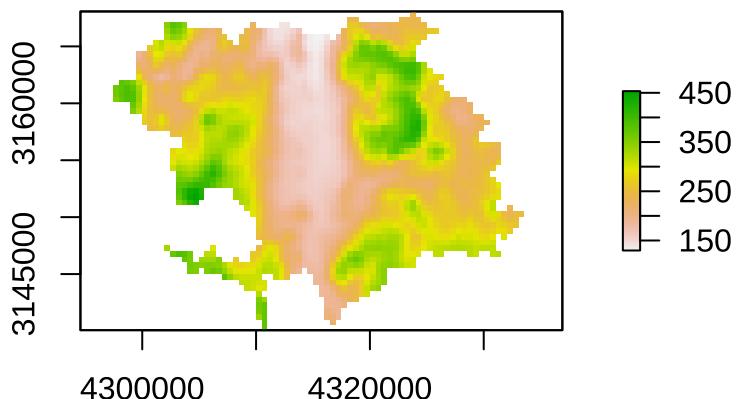
```
dem1 <- crop(dem, goe)
plot(dem1)
```



2185

2186 Der Raster hat jetzt die Größe einer Bounding-Box (BBX) von Göttingen (das ist ein Rechteck, das Göttingen
 2187 umfasst). Mit der Funktion `mask()` kann der Raster auf die genauen Grenzen des Vektors `goe` angepasst
 2188 werden.

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



2189

2190 Wenn wir an bestimmten Punkten den Wert des Rasters abfragen wollen (z.B. an `cities`) von vorhin, dann

2191 gibt es dafür die Funktion `extract`. Dann müssen wir erst sicherstellen, dass `staedte` und `dem` gleichen
 2192 KBS zu grunde liegt. Dafür transformieren wir einfach `staedte` in das KBS von `dem`. Mit der Funktion
 2193 `projection()` erhalten wir das KBS des Rasters.

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

2194 Wenn wir das KBS eines Objektes nicht kennen, können wir auch einfach das KBS übergeben. Der folgende
 2195 Code-Block macht genau das Gleiche mit dem Vorteil, dass wir keinen EPSG-Code angeben müssen.

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

2196 Dann können wir für jede Stadt die Seehöhe abfragen:

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

2197 ## [1] 149.18181 57.21486 NA

2198 Mit `raster::extract()` rufen wir *eindeutig* die Funktion `extract()` aus dem Paket `raster` auf. Wir müssen
 2199 das so machen, weil es im Paket `dplyr` auch eine Funktion `extract()` gibt, die wir hier nicht anwenden
 2200 möchten, da sie einen Fehler verursachen würde.

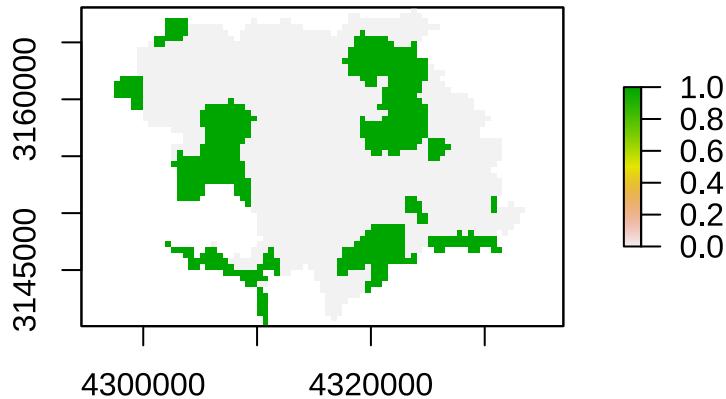
2201 Ein analoges Vorgehen ist auch für Linien und Polygone möglich.

2202 Mit Rastern kann auch einfach gerechnet werden. Wir können z.B. die Seehöhe in Kilometern anstatt Metern
 2203 berechnen:

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

2204 Auch logische Operationen sind möglich, wenn wir alle Rasterzellen mit einer Seehöhe von mehr als 300 m
 2205 in Göttingen suchen, dann geht das so:

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



2206

2207 Wenn wir jetzt auf die Werte des Rasters `dem3` zugreifen wollen, geht das mit eckigen Klammern.

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

2208 ## [1] NA NA NA NA NA NA

2209 Das sind erst einmal viele NA-Werte für die ganzen Zellen, die außerhalb von Niedersachsen liegen. Aber wir
2210 können mit so einem Vektor ganz normal arbeiten und z.B. die Fläche des Landes Niedersachsen die eine
2211 Seehöhe von mehr als 500m Seehöhe hat ausrechnen.

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

2212 ## [1] 0.265713

2213

Aufgabe 41: Arbeiten mit Rastern

2216 Verwenden Sie den Raster `wald.tif`, der auf einer 10 m Auflösung den Waldanteil jeder Rasterzelle angibt¹⁹.
2217 Der EPSG-Code für das KBS von `wald.tif` ist 3035. Nehmen Sie an, dass wenn der Waldanteil in einer Raster
2218 größer als 50 % ist, dass die Rasterzelle als Wald klassifiziert werden kann. Wie viel Prozent des Göttinger
2219 Stadtgebietes sind Wald? Wie ändert sich dieser Wert, wenn sie 70 % anstatt 50 % als Schwellenwert für
2220 Wald annehmen?

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

2221

2222 Aufgabe 42: Studiendesign

- 2224 Mit der Funktion `st_sample()` können Sie innerhalb oder entlang eines Features zufällige Punkte legen. Das
 2225 Argument `n` steuert die Anzahl Punkte und das Argument `type` wie die Punkte angeordnet werden. Für `type`
 2226 sind für uns die Werte `type = "random"` (komplett zufällig), `type = "regular"` (regelmäßiger Grid) und
 2227 `type = "hexagonal"` von Bedeutung (ein hexagonaler Grid, d.h. ein sechseckiger Raster). Unglücklicherweise
 2228 ist das Ergebnis von `st_sample()` erst eine Geometrie. Um daraus ein vollständiges `sf`-Objekt zu machen
 2229 und problemlos weiter arbeiten zu können, müssen Sie noch einmal die Funktion `st_as_sf()` ausführen.
- 2230 Stellen Sie sich vor, dass wir die tatsächliche Waldbedeckung des Göttinger Stadgebietes **nicht** kennen
 2231 und wir eine Studie durchführen, um den Anteil des Göttinger Stadgebietes, der mit Wald bedeckt ist
 2232 herauszufinden. Erstellen Sie dafür einige unterschiedliche Stichproben (diese können in der Anzahl und
 2233 Anordnung variieren).
- 2234 Berechnen Sie für jedes Stichprobendesign den Anteil an Wald und ein dazugehöriges Konfidenzintervall
 2235 (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
 2236 die Stichprobengröße). Nehmen Sie an, dass eine Rasterzelle Wald ist, sobald > 50 % der Rasterzelle mit
 2237 Wald bedeckt ist.

2238

2239 Aufgabe 43: Räumliche Daten

- 2241 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)
)
```

- 2242 1. Erstellen Sie ein `sf`-Objekt aus `df1`.
- 2243 2. Puffern Sie jeden Baum mit seinem Kronendurchmesser.
- 2244 3. Berechnen Sie die Kronenfläche jedes Baumes. *Hinweis: Die Funktion `st_area()` könnte dafür hilfreich sein.*
- 2245 4. Welcher Baum hat die größte Kronenfläche?
- 2246 5. Finden Sie für jede Art, den Baum mit der größten Kronenfläche.

2248

2249 **Aufgabe 44: Arbeiten mit räumlichen Daten**

- 2251 1. Lesen Sie das ESRI Shapefile goettingen/stadt_goettingen.shp ein.
2252 2. Wie viele Features befinden sich in dem Shapefile?
2253 3. Welches Koordinatenbezugssystem (KBS) hat das Shapefile?
2254 4. Transformieren Sie das Shapefile in das KBS 3035.
2255 5. Erstellen Sie eine neue Spalte A in der Sie die Fläche jeder Gemeinde/Stadt speichern.
2256 6. Welche Gemeinde/Stadt (Spalte GEN) ist am größten?
2257 7. Wählen Sie nun nur die Stadt Göttingen aus.

2258

2259 **Aufgabe 45: Arbeiten mit räumlichen Daten 2**

- 2261 1. Lesen Sie erneut das ESRI Shapefile goettingen/stadt_goettingen.shp ein.
2262 2. Lösen Sie die Gemeindegrenzen auf (die Funktion `st_union()` könnte hier nützlich sein).
2263 3. Wie groß ist das resultierende Feature?

2264 **15 FAQs (Oft gefragtes)**

2265 **15.1 Arbeiten mit Daten**

2266 **15.1.1 Einlesen von Exceldateien**

- 2267 Mit der Funktion `read_excel()` aus dem Paket `readxl` können Exceldateien direkt in R eingelesen werden.
2268 Ein Export als csv-Datei aus Excel ist nicht notwendig.

2269 16 Zusätzliche Aufgaben

2270

2271 **Aufgabe 46: Standardisierung**

- 2273 Unter Standardisierung (oder auch z-Transformation) versteht man die Transformation einer Variable, so
2274 dass sie den Mittelwert 0 und die Varianz 1 hat. Die Formel für die Standardisierung ist

$$x_s = \frac{x - \mu_x}{\sigma_x}$$

2275 wobei x die Variable ist, μ_x ist der Mittelwert von x und σ_x ist die Standardabweichung von x .

2276 Standardisieren Sie folgenden Vektor:

```
h <- c(0, 2, 3, 1, 0, 8, 3.4, 9, 6.8, 2.1)
```

- 2277 Und speichern Sie das Ergebnis in `h_s`. Vergewissern Sie sich, dass die Standardisierung geklappt hat und
2278 berechnen Sie den Mittelwert und Standardabweichung von `h_s`.

2279

2280 **Aufgabe 47: Arbeiten mit logischen Werten**

- 2282 Verwenden Sie nochmals den Vektor mit der Anzahl Rehe, die an unterschiedlichen Fotofallenstandorten
2283 fotografiert wurden.

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

- 2284 Für wie viele Standort trifft die Aussage zu $90 \leq x < 120$, wobei x für die Anzahl Fotos an einem Standort
2285 steht.

2286

2287 **Aufgabe 48: Auswählen von Elementen in einem Vektor**

- 2289 Lesen Sie die Datei `bhd_1.txt` ein. Und bearbeiten Sie folgende Aufgaben mit dieser Datei:

- 2290 • Finden Sie den mittleren BHD aller Eichen.
- 2291 • Wie viele Beobachtungen haben Sie für Eichen, Fichten und Buchen?
- 2292 • Finden Sie alle Bäume, die 10, 20, 21, 23, 30, 37, 78, 79, 90, 91, 92 Jahre alt sind.

2293

2294 **Aufgabe 49: Arbeiten mit Daten**

2295

2296 Wang et al. (2019) haben in einer Fotofallenstudie das Verhalten und die Habitatselektion von Ozeloten
2297 im brasilianischen Amazonas untersucht. Ziel dieser Übung ist es mit dem Datensatz etwas vertraut zu
2298 werden, wir werden noch keine ökologischen Analysen durchführen. Mehr zu dem Datensatz erfahren Sie
2299 [hier](#). Eine etwas angepasste Version des Datensatzes können Sie aus dem StudIP Ordner **daten** (die Datei
2300 heißt **ozelote.zip**) herunterladen. Speichern Sie die Datei in Ihrem RStudio Projekt und entzippen Sie sie.
2301 Der Ordner enthält zwei Dateien, für diese Übung brauchen wir lediglich die Datei **ozelote_standorte.csv**,
2302 die für jeden Fotofallen Standort einige Kovariaten angibt.

2303 Bearbeiten Sie folgende Aufgaben:

- 2304 1. Lesen Sie die Datei **ozelote_standorte.csv** in R und speichern Sie das Ergebnis in eine Variable
2305 **standorte**.
- 2306 2. Wie viele Fotofallenstandorte gab es in der Studie?
- 2307 3. Welcher Standort ist am Höchsten gelegen? Die Spalte **seehoeh**e enthält die mittlere Seehöhe.
- 2308 4. Finden Sie alle Standorte, die in unmittelbarer Nähe zu Flüssen sind. Eine Distanz von < 5 m kann
2309 als Schwellenwert angenommen werden. Die Spalte **dist_fluss** gibt die Distanz zu Flüssen an.
- 2310 5. Der Datensatz besteht aus verschiedenen Kameras, die jeweils für einen Zeitraum von 12 Tagen in einer
2311 Region aufgestellt wurden (Spalte **Region**). Erstellen Sie einen Plot, der den Zusammenhang zwischen
2312 der Region und Seehöhe darstellt.

2313

2314 **Aufgabe 50: Base Plots**

2315

2316 Erstellen Sie die folgende Beispielabbildung Schritt für Schritt selbst über Low-Level Funktionen. Die Roh-
2317 daten finden Sie in den Dateien **abbBeispiel.R** und **ertragstafeldaten.csv**.

- 2318 • Die Wachstumskurve der Region 1 (blau) lautet $41.45752(1 - \exp(-0.02168x))^{1.61787}$
- 2319 • Die Wachstumskurve der Region 2 (rot) lautet $51.11203(1 - \exp(-0.009129x))^{1.202401}$

2320 wobei x das Baumalter in Jahren angegeben ist. Die 3 schwarzen Linien sind auf der Ertragstafel abgelesen.
2321 Die Beschriftungen der 3 Ertragstafelkurven, sowie des Ausreißers, sind Zusatzaufgaben.

2322

2323 **Aufgabe 51: ggplot2 Aufgabe**

2324

- 2325 1. Laden Sie den Datensatz **daten/bhd_1.txt**

- 2326 2. Erstellen Sie ein Streudiagramm. Bilden Sie dabei den BHD gegen das Alter ab, wobei dies als Subplot
2327 für jedes Aufnahmegericht dargestellt werden sollte.
- 2328 3. Verwenden Sie für jede Baumart eine eigene Farbe.
- 2329 4. Erstellen Sie für jede Baumart einen Boxplot des BHDs.
- 2330 5. Teilen Sie die Boxplots aus 4) auf jeweils einen Subplot pro Aufnahmegericht auf.

2331

Aufgabe 52: Anwendungsbeispiel kontrollierter Programmabläufe

- 2334 • Öffnen Sie ein neues, leeres R Skript.
2335 • Laden Sie die Datei "stichprobe.csv" in eine Variable.

```
stpr <- read.csv("data/stichprobe.csv", fileEncoding = "UTF-8")
```

- 2336 • Filtern Sie den Data Frame so, dass er nur noch die Baumart "Eiche" enthält. Speichern Sie den
2337 gefilterten Data Frame in einer NEUEN Variable ab.
- 2338 • Berechnen Sie die deskriptiven Statistiken `mean()`, `sd()`, `median()`, `min()` und `max()` des Kapitels
2339 "Deskriptive Statistik" für den BHD (des gefilterten Data Frames).
- 2340 • Erstellen Sie ein Histogramm des BHD (ebenfalls mit dem gefilterten Data Frame), zeichnen Sie den
2341 arithmetischen Mittelwert als horizontale Linie in das Histogramm ein.
- 2342 • Speichern Sie den R Code und kopieren Sie ihn in ein neues R Skript.
- 2343 • Erstellen Sie nun eine Schleife, die alle Statistiken und auch die Abbildung für jede Baumart berechnet.
2344 Lassen Sie die Statistiken mit `print()` in die Konsole ausgeben.
- 2345 • ZUSATZ: Exportieren Sie die Histogramme (bspw. als PDF). TIPP: Verwenden Sie `paste()` um sinn-
2346 volle Namen für die Dateien zu erstellen. Machen Sie sich selbst mit der Funktion vertraut.
- 2347 • ZUSATZ: Sie wollen Fehlermeldungen vermeiden. Deshalb programmieren Sie eine bedingte Ausfüh-
2348 rung, um die gesamten statistischen Berechnungen und auch die Abbildung. Führen Sie Ihren gesamten
2349 Code nur unter der Bedingung aus, dass die Baumart "Ei", "Bu", "Fi", "Kie" oder "Dou" ist. TIPP:
2350 Sie können den `%in%` Operator verwenden.

16.1 Arbeiten mit Daten

- 2351 2352 Verwenden Sie erneut die Datensatz von Wang et al. (2019) zu Ozeloten in Brasilien für die nachfolgenden
2353 Übungen.

2354

2355 **Aufgabe 53: Datenzusammenfassen**

2356

- 2357 1. Laden Sie die Datei `ozelote_standorte.csv` in R und speichern Sie das Ergebnis in eine Variable
2358 `standorte`.
- 2359 2. Berechnen Sie die Anzahl an Fotofallen für jede Region. Welche Region weißt die meisten Fotofallen
2360 auf?
- 2361 3. In welcher Region ist die größte Variabilität der Seehöhe zu finden?
- 2362 4. In welchen Regionen beträgt der Anteil an Fotofallen, die < 5m vom nächsten Fluss entfernt sind,
2363 mindestens 20%?

2364

2365 **Aufgabe 54: Datenmanipulation 1**

2366

- 2367 1. Laden Sie nun zusätzlich die Datei `ozelote_fanghistorien.csv` und speichern Sie diese in die Variable
2368 (`fh`). In diesem `data.frame` gibt es für jede Session eine Spalte (V1 bis V10). Eine 1 bedeutet, dass
2369 mindestens ein Ozelot fotografiert wurde und eine 0 bedeutet, dass kein Ozelot in diesem Zeitraum
2370 fotografiert wurde. NA heißtt, dass die Kamera nicht aktiv war.
- 2371 2. Wählen Sie nur das 3. Fangereignis (das ist die Spalte V3).
- 2372 3. Wie viele Kameras waren beim 3. Fangereignis aktiv?
- 2373 4. Vergleichen Sie anhand einer Abbildung, ob sich die Distanz zum Fluss (Spalte `dist_fluss`) zwischen
2374 Standorten mit Fotos (V3 == 1) und Standorten ohne Fotos (V3 == 0) unterscheidet.

2375

2376 **Aufgabe 55: Datenmanipulation 2 (etwas knifflig)**

2377

- 2378 1. Verwenden Sie erneut die Daten zu den Fotofallenstandorten und Fanghistorien der Ozelote.
- 2379 2. Finden Sie alle Fotofallenstandorte an denen ≥ 3 Ozelote fotografiert wurden?
- 2380 3. Gibt es einen Zusammenhang zwischen der Häufigkeit an Ozelotfotos (pro Fotofallenstandort) und der
2381 Distanz zum nächsten Fluss (Spalte `dist_fluss`)? Eine Abbildung ist ausreichend.

2382 17 Literatur

- 2383 Ein guter Überblick über viele der angeschnittenen Themen gibt es in dem [R for data science](#), das online
2384 frei zugänglich ist. Das on-line Buch [Hands-On Programming with R]{[https://rstudio-education.github.io/
2385 hopr/index.html](https://rstudio-education.github.io/hopr/index.html)} ist eine nicht-Programmierer freundliche Einführung in R.
2386 McNamara, Amelia, and Nicholas J Horton. 2018. “Wrangling Categorical Data in r.” *The American Stati-
2387 stician* 72 (1): 97–104.
2388 Wang, Bingxin, Daniel G. Rocha, Mark I. Abrahams, André P. Antunes, Hugo C. M. Costa, André Luis
2389 Sousa Gonçalves, Wilson Roberto Spironello, et al. 2019. “Habitat Use of the Ocelot (*Leopardus Pardalis*)
2390 in Brazilian Amazon.” *Ecology and Evolution* 9 (9): 5049–62. <https://doi.org/10.1002/ece3.5005>.
2391 Wickham, Hadley. 2014. “Tidy Data.” *Journal of Statistical Software* 59 (10). [https://doi.org/10.18637/jss.
2392 v059.i10](https://doi.org/10.18637/jss.v059.i10).