

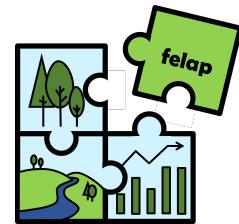
1

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:

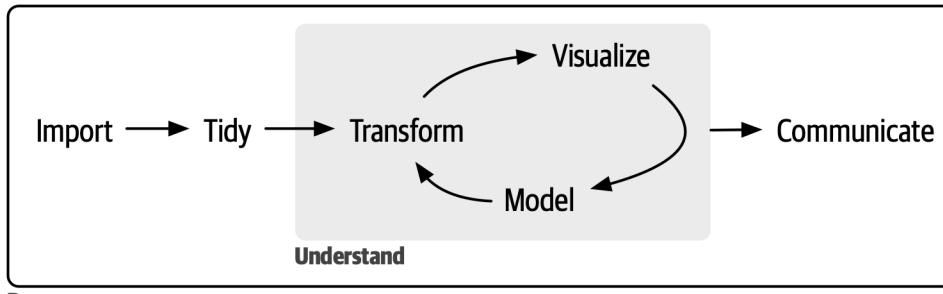
¹⁶ Signer, J. und Husmann, K. (2023) Skript zur Vorlesung Einführung in die Datenanalyse mit R, Georg-August-Universität Göttingen.

¹⁸ Letzte Aktualisierung: 13. Dezember 2023

19 Vorwort und Danksagung

20

- 21 Lernziel des Kurses ist die Einführung in die Arbeit, Visualisierung und Analyse von (forstlichen) Datensätzen
22 mit dem Statistikprogramm *R*. Der Schwerpunkt liegt dabei auf der Datenverarbeitung. Statistische Methoden
23 werden nur an wenigen Stellen exemplarisch angewendet. Ein typisches Data Science Projekt besteht laut
24 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 und
27 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 Dokument
37 besteht aus Fließtext, R Code und den entsprechenden Code-Ergebnissen. Die grau hinterlegten Codepassagen
38 sind kurze R-Skripte. Falls das Skript eine Konsoleausgabe erzeugt, ist diese direkt mit "##" markiert (diese
39 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	3
46	1.1 Installation von R und RStudio	3
47	1.2 Erste Schritte in R	3
48	1.3 Gute Praxis bei der Programmierung	5
49	2 Variablen, Funktionen und Datentypen	7
50	2.1 Variablen beim Programmieren	7
51	2.2 Datentypen	8
52	2.3 Funktionen	9
53	2.4 Datenstrukturen	10
54	2.5 Funktionen	11
55	3 Vektoren	13
56	3.1 Funktionen zum Arbeiten mit Vektoren	15
57	3.2 Statistische Funktionen	16
58	3.3 Beispiel Fotofallen	17
59	3.4 Arbeiten mit logischen Werten	18
60	3.5 Zugreifen auf Elemente eines Vektors (=Untermengen)	20
61	3.6 Der %in%-Operator	21
62	4 Faktoren (factors)	23
63	4.1 Das Paket forcats	24
64	4.1.1 Anpassen der Anordnung von Faktoren	25
65	5 Spezielle Einträge	26
66	5.1 NA	26
67	5.2 NULL	27
68	5.3 Inf	27
69	6 data.frames oder Tabellen	29
70	6.1 Wichtige Funktionen zum Arbeiten mit data.frames	30
71	6.2 Zugreifen auf Elemente eines data.frame	31
72	7 Schreiben und lesen von Daten	34
73	7.1 Textdateien	34
74	8 Erstellen von Abbildungen	36
75	8.1 Base Plot	36
76	8.1.1 Mehrere Panels	42
77	8.1.2 Speichern von Abbildungen	43
78	8.2 Histogramme	44
79	8.3 Boxplots	46
80	8.4 ggplot2 : Eine Alternative für Abbildungen	48

81	8.4.1 Multipanel Abbildungen	57
82	8.4.2 Plots kombinieren	59
83	8.4.3 Speichern von plots	63
84	9 Mit Daten arbeiten	64
85	9.1 <code>dplyr</code> eine Einführung	64
86	9.2 Arbeiten mit gruppierten Daten	67
87	9.3 <code>pipes</code> oder <code>%>%</code>	68
88	9.4 Joins	69
89	9.5 ‘long’ and ‘wide’ Datenformate	71
90	9.6 Auswählen von Variablen	73
91	9.7 Einzelne Beobachtungen abfragen (<code>slice()</code>)	74
92	9.8 Spalten trennen	77
93	10 Arbeiten mit Text	79
94	10.1 Arbeiten mit Text	79
95	10.2 Finden von Textmustern	80
96	11 Arbeiten mit Zeit	83
97	11.1 Arbeiten mit Zeitintervallen	84
98	11.2 Formatieren von Zeit	86
99	11.3 Zeitreihen	86
100	12 Aufgaben Wiederholen (for-Schleifen)	91
101	12.1 Schleifen	91
102	12.1.1 Wiederholen von Befehlen mit <code>for()</code>	91
103	12.1.2 Wiederholen von Befehlen mit <code>while()</code>	94
104	12.2 Bedingte Ausführung von Codeblöcken	94
105	13 (R)markdown	96
106	13.1 Markdown Grundlagen	96
107	13.2 R und Markdown	97
108	14 Räumliche Daten in R	99
109	14.1 Was sind räumliche Daten	99
110	14.2 Koordinatenbezugssystem	99
111	14.3 Vektordaten in R	99
112	14.4 Arbeiten mit Vektordaten	101
113	14.5 Rasterdaten in R	103
114	15 FAQs (Oft gefragtes)	108
115	15.1 Arbeiten mit Daten	108
116	15.1.1 Einlesen von Exceldateien	108
117	16 Literatur	109

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] + [Umschalt] + [N]**).

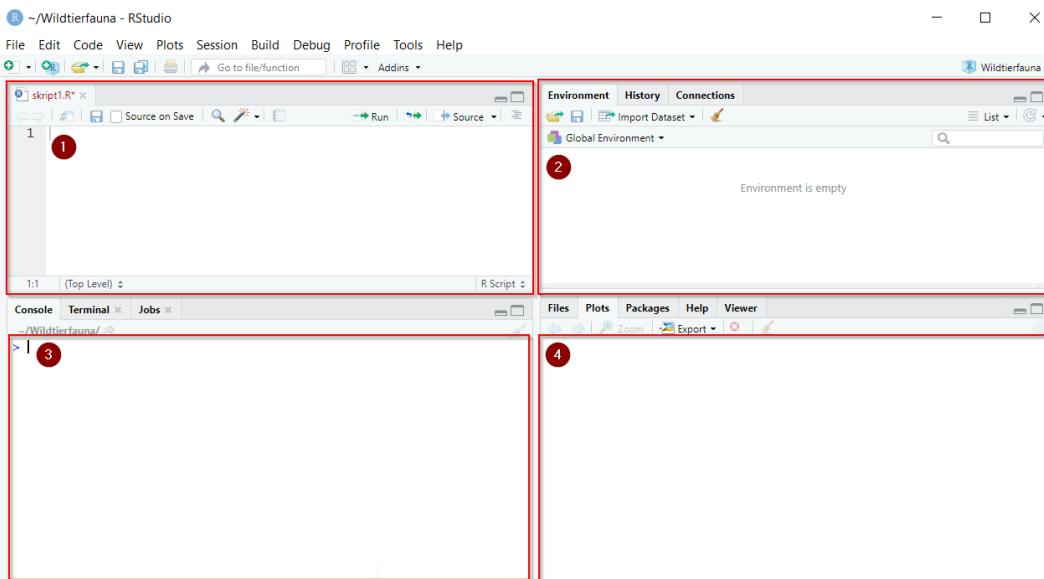


Abbildung 1: RStudio Panes.

- RStudio besteht nun aus vier sogenannten **Panes** oder Ausschnitten (siehe auch Abbildung 1). Die Ausschnitte sind wie folgt gegliedert:
1. Hier werden Skripte angezeigt, d.h., hier wird meist R Code geschrieben und dokumentiert. Der Code wird

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

137 beim Schreiben noch nicht ausgeführt (siehe Punkt 3), dennoch sollten Sie Code immer so schreiben,
 138 dass er Zeile für Zeile abgeschickt werden kann. Sie sollten nie Code schreiben, bei dem Sie zwischen
 139 den Zeilen hin und her springen müssen.

- 140 2. Der zweite Ausschnitt erteilt Auskunft über den *Workspace*. Im Workspace werden alle verfügbaren
 141 Objekte angezeigt.
- 142 3. Die eigentliche R-Konsole wird in Ausschnitt 3 dargestellt. Hier wird in der Regel wenig Code eingegeben.
 143 Der normale Workflow ist vom Skript Code an die Konsole zu schicken. Erst durch das Abschicken in
 144 die Konsole wird der Code (bzw. die Teile des Codes, die Sie abschicken) ausgeführt.
- 145 4. Der vierte Ausschnitt enthält mehrere Reiter. Der Reiter *Files* zeigt den Verzeichnisbaum an. Im Reiter
 146 *Plots* werden Plots angezeigt, wenn diese im Code aufgerufen werden. Hilfeseiten zu Funktionen werden
 147 im Reiter *Help* angezeigt.

148 Einfache Rechenoperationen können auch direkt in der R-Konsole durchgeführt werden. Prinzipiell könnten
 149 Sie alle Operationen direkt in die Konsole tippen. Der Nachteil und der Grund, warum dies keine gute Praxis
 150 ist, ist, dass der Code zwar ausgeführt jedoch nicht gespeichert wird. Code der nicht als Skript gespeichert
 151 wird, ist also nicht dokumentiert. Tippen Sie die folgenden Operationen in die Konsole.

10 + 5

152 **## [1] 15**

20 - 10

153 **## [1] 10**

10 * 3

154 **## [1] 30**

100 / 19

155 **## [1] 5.263158**

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

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

165 Meist verwenden wir jedoch **Skripte**, um den R-Code zu schreiben und ihn dann an die Konsole “zu schicken”.
 166 Dies hat den Vorteil, dass alle Schritte nachvollziehbar bleiben und Analysen beliebig oft wiederholt werden
 167 können. Nach der Ausführung bleibt der Code erhalten und Sie dokumentieren Ihre Berechnungen automatisch
 168 mit. Stellen Sie sich den Code im R-Skript wie ein Kochbuch vor. Wenn wir R-Code in einem R-Skript
 169 geschrieben haben gibt es mehrere Möglichkeiten diesen Code abzuschicken/ auszuführen. Wir können eine
 170 Zeile abschicken, indem wir entweder auf *Run* klicken (Abbildung 2) oder die Tastenkombination *Strg +*

171 Enter (**Strg**+**↵**) tippen. Mehrere Zeilen abzuschicken und nacheinander ausführen zu lassen ist möglich,
 172 indem diese Zeilen markiert werden bevor Sie *Run* klicken oder die Tastenkombination tippen. Ein Klick auf
 173 *Source* bzw. die Tastenkombination *Strg + Umschalt + Enter* (**Strg**+**↑**+**↵**).

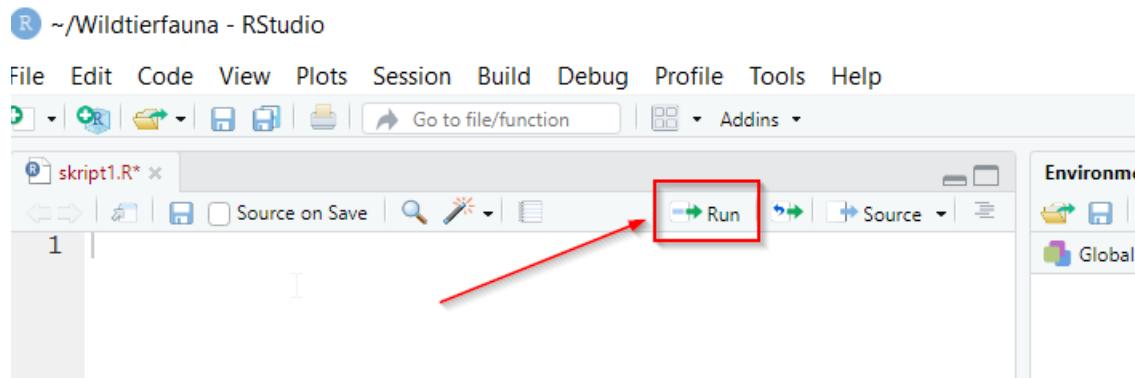


Abbildung 2: Zeilenweises Ausführen von Code in RStudio.

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

179 1.3 Gute Praxis bei der Programmierung

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

187 Vielleicht die wichtigste Praxis beim Programmieren ist das Kommentieren. **Kommentare** sind ein wichtiger
 188 Bestandteil der Skripte in R und allen anderen Skript- und Programmiersprachen. Sie werden lernen, dass die
 189 Kommentarfunktion ein wesentlicher Vorteil gegenüber Klickprogrammen darstellt. Ein Kommentar ist Text
 190 in einem (R-)Code, der der Dokumentation dient und von der R Konsole ignoriert wird. Sämtliche Zeilen, die
 191 mit dem Zeichen # beginnen, werden nicht ausgeführt, wenn Sie an die Konsole gesendet werden. Seien Sie
 192 nicht sparsam mit Kommentaren, sondern benutzen Sie sie um Ihren Code zu strukturieren, ihre Berechnungen
 193 zu für sich selbst und andere zu erläutern und, um Ergebnisse zu beschreiben oder zu interpretieren.

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

194 ## [1] 9

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

201

202 **Aufgabe 1: Ausführen von Quellcodes**

204 Öffnen Sie RStudio, erstellen Sie ein neues Skript und speichern Sie dieses unter dem Namen `skript1.R` ab.
205 Tippen oder kopieren Sie folgenden Code in das Skript:

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

# Einfache Funktion
sqrt(20)
```

206 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
```

führt zu einem Fehler.

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

```
a <- 10
b <- 5

a + b
```

```

230 ## [1] 15
b / a

231 ## [1] 0.5
a^b

232 ## [1] 1e+05

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

ergebnis <- a + b
ergebnis

234 ## [1] 15

ergebnis2 <- ergebnis * 2
ergebnis2

235 ## [1] 30

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

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

239 ## [1] TRUE

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

240 ## [1] FALSE

```

2.2 Datentypen

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

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

```

kamera_name <- "Kamera_1"
anzahl_rehe <- 132

```

249 In den zwei vorherigen Zeilen Code haben wir zwei Objekte (im Sinne von R) erstellt. Das erste Objekt heißt
250 `kamera_name` und das zweite Objekt heißt `anzahl_rehe`. In dem Beispiel handelt es sich also um zwei sehr
251 einfache Objekte, in denen jeweils ein Wert gespeichert ist. Auffällig ist, dass beide Objekte unterschiedliche

252 Datentypen haben. `kamera_name` ist vom Typ `character` (also Text). Das zweite Objekt, `anzahl_rehe`, ist
 253 vom Typ `numeric` (also eine Zahl, wir unterscheiden hier nicht weiter²). Zusätzlich zu diesen zwei Typen
 254 (`character` und `numeric`), gibt es noch einen weiteren wichtigen Typ: nämlich das logische Wahr oder Falsch
 255 (in R: `TRUE` und `FALSE`) und noch weitere Datentypen auf die wir zunächst nicht eingehen (tippen Sie `?typeof`
 256 für eine Übersicht aller Datentypen). Zurückkommend auf das Beispiel mit den Fotofallen, könnte eine
 257 mögliche Fragestellung sein, ob auf einem der Fotos ein Fuchs gesehen wurde oder nicht. Dazu würden wir
 258 eine neue Variable `fuchs_gesehen` anlegen und diese auf `TRUE` setzen, da ein Fuchs gesehen wurde.

```
fuchs_gesehen <- TRUE
```

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

```
typeof(fuchs_gesehen)
```

260 `## [1] "logical"`

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

```
TRUE + TRUE
```

262 `## [1] 2`

```
FALSE + FALSE
```

263 `## [1] 0`

```
TRUE + FALSE
```

264 `## [1] 1`

2.3 Funktionen

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

```
sqrt(a)
```

268 `## [1] 3.162278`

269 Funktionen sind in R in der Regel daran zu erkennen, dass sie immer mit dem Funktionsnamen, gefolgt von
 270 runden Klammern (), aufgerufen werden. Der große Umfang an Funktionen für die statistische Datenanalyse
 271 und wissenschaftliche Datenverarbeitung ist der Hauptgrund für den Erfolg von R in der Wissenschaft. Im
 272 vorherigen Beispiel wurde die Funktion mit dem Namen `sqrt()` aufgerufen. Das Objekt `a` haben wir bereits
 273 vorhin definiert (zur Erinnerung `a <- 10`). Die Funktion `sqrt()` arbeitet jetzt mit dem Objekt `a`, das in
 274 diesem Zusammenhang auch **Argument** genannt wird. Argumente sind die Objekte, die eine Funktion als
 275 Input benötigt. Die Hilfeseite jeder Funktion enthält eine Liste aller Argumente. Argumente von Funktionen
 276 haben Namen, diese müssen jedoch nicht angegeben werden, wenn die Reihenfolge der Argumente, wie
 277 in der Hilfeseite angegeben, berücksichtigt wird. Im vorherigen Beispiel, haben wir die Funktion `sqrt(a)`
 278 aufgerufen und keinen Argumentnamen angegeben. Aus den Hilfeseiten für die Funktion `sqrt()` (siehe auch

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

279 nachfolgender Abschnitt) ist zu entnehmen, dass die Funktion `sqrt()` nur ein Argument mit dem Namen `x`
280 hat. Das heißt, der vollständige Aufruf der Funktion `x` wäre.

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

281 `## [1] 3.162278`

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

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

294 2.4 Datenstrukturen

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

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

304 Wenn wir so vorgehen würden, hätten wir 45 Objekte mit je einem Eintrag. Dieser Ansatz und führt schnell

zu einem unübersichtlichen *Workspace*³. Wir werden im Verlauf sinnvollere Objekte (Vektoren oder Data Frames) für diesen Zweck kennenlernen.

307

308 **Aufgabe 2: Variablen**

310 Verwenden Sie die folgenden Daten

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

311 und berechnen sie:

- 312 • $10 * a$
 313 • $a / 144$ und speichern Sie das Ergebnis in einer neuen Variablen **e** zwischen.
 314 • Was ist das Ergebnis von $a + b$?
 315 • Was ist das Ergebnis von $a + p$?

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

316 **2.5 Funktionen**

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

```
sqrt(a)
```

319 ## [1] 1.414214

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

324 Argumente von Funktionen haben Namen, diese müssen jedoch nicht angegeben werden, wenn die Reihenfolge
 325 der Argumente berücksichtigt wird. Im vorherigen Beispiel, haben wir einfach die Funktion **sqrt(a)** aufgerufen
 326 und keinen Argumentnamen angegeben. Aus den Hilfeseiten für die Funktion **sqrt()** (siehe auch nachfolgender
 327 Abschnitt) ist zu entnehmen, dass die Funktion **sqrt()** ein Argument mit dem Namen **x** hat. Das heißt, der
 328 vollständige Aufruf der Funktion **x** wäre.

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

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

329 ## [1] 1.414214

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

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

3 Vektoren

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

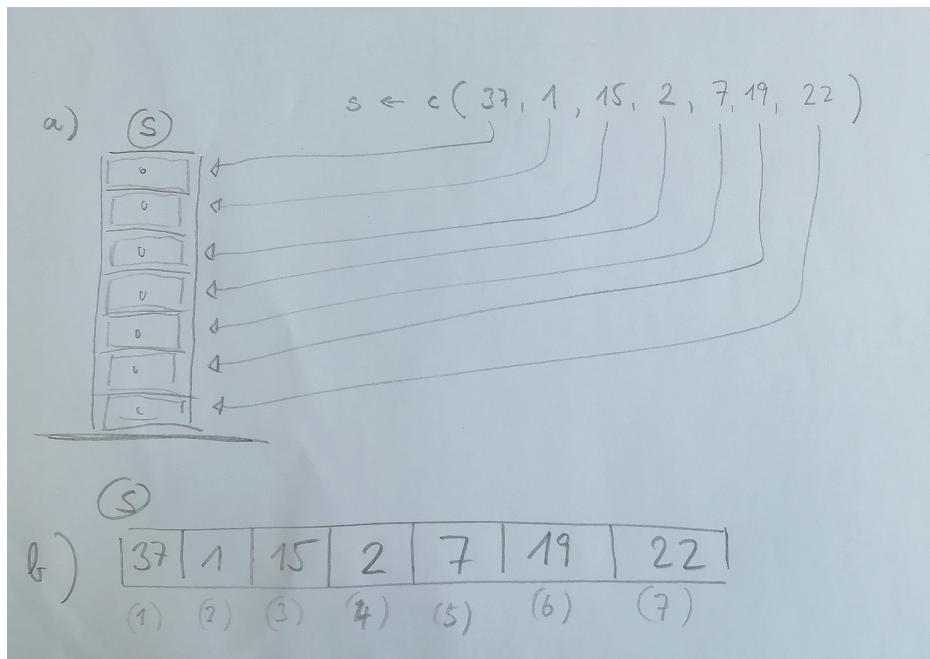


Abbildung 3: Schematische Darstellung eines Vektors in R.

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

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

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

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

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

s

362 ## [1] 37 1 15 2 7 19 22

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

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

s + 10

368 ## [1] 47 11 25 12 17 29 32

369 oder s mit sich selbst multiplizieren. Zu beachten ist also, dass es sich bei Vektorberechnungen in R
370 zunächst nicht um Vektorrechnungen handelt, wie Sie sie aus der linearen Algebra kennen. Für die sog.
371 Matrizenoperationen der linearen Algebra werden die Operatoren in R mit % % umschlossen, also bspw. s
372 %*% s.

s * s

373 ## [1] 1369 1 225 4 49 361 484

374 Neben der Funktion c() gibt es zahlreiche weitere Funktionen, um Vektoren zu erstellen. Sehr häufig
375 braucht man Vektoren von Zahlenfolgen. Solche Vektoren können mit der Funktion seq() erstellt werden. Im
376 einfachsten Fall benötigt seq() zwei Argumente: from und to⁴.

seq(from = 1, to = 10)

377 ## [1] 1 2 3 4 5 6 7 8 9 10

(1 : 10)

378 ## [1] 1 2 3 4 5 6 7 8 9 10

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

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

380 ## [1] 1 3 5 7 9

381

Aufgabe 3: Vektoren erstellen

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

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

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.

3.1 Funktionen zum Arbeiten mit Vektoren

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

```
head(s)
```

391 ## [1] 37 1 15 2 7 19

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

392 ## [1] 37 1 15

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

393 ## [1] 19 22

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

```
length(s)
```

395 ## [1] 7

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

```
class(s)
```

397 ## [1] "numeric"

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

```
unique(s)
```

399 ## [1] 37 1 15 2 7 19 22

400 Mit der Funktion `table` kann die Häufigkeit verschiedener Elemente abgefragt werden.

```
table(s)
```

401 ## s

402 ## 1 2 7 15 19 22 37

403 ## 1 1 1 1 1 1 1

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

```
rev(s)
```

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

407 während `sort()` einen Vektor nach seinen Elementen sortiert⁵.

```
sort(s)
```

408 ## [1] 1 2 7 15 19 22 37

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.

409 Die Funktion `rep()` wiederholt einen Vektor.

```
rep(s, times = 2)
```

410 ## [1] 37 1 15 2 7 19 22 37 1 15 2 7 19 22

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

```
a <- 1:4
```

```
rep(a, times = 2)
```

413 ## [1] 1 2 3 4 1 2 3 4

```
rep(a, each = 2)
```

414 ## [1] 1 1 2 2 3 3 4 4

415

416 Aufgabe 4: Arbeiten mit Vektoren

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

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

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

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

422 3.2 Statistische Funktionen

423 Zahlreiche statistische Funktionen können auf Vektoren angewendet werden, hier sind nur die wichtigsten
424 aufgeführt: `mean()` berechnet den Mittelwert, `median()` berechnet den Median und `sd()` die Standardabwei-
425 chung.

```
mean(s)
```

426 ## [1] 14.71429

```
median(s)
```

427 ## [1] 15

```
sd(s)
```

428 ## [1] 12.76341

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

```

sample(s, size = 1) # 1 Element
432 ## [1] 1
sample(s, size = 3) # 2 Elemente
433 ## [1] 15 7 22

```

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

436 3.3 Beispiel Fotofallen

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

440 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,
441 105, 96, 146, 95, 118, 1007)

```

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

```

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

```

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

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

```

v1 <- rep("Kamera", 15)

```

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

```

v2 <- 1:15

```

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

```

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

```

```

454 ## [1] "Kamera_1"  "Kamera_2"  "Kamera_3"  "Kamera_4"  "Kamera_5"  "Kamera_6"
455 ## [7] "Kamera_7"  "Kamera_8"  "Kamera_9"  "Kamera_10" "Kamera_11" "Kamera_12"
456 ## [13] "Kamera_13" "Kamera_14" "Kamera_15"

```

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

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

```

459 ## [1] "Revier A" "Revier B" "Revier C" "Revier A" "Revier B" "Revier C"
460 ## [7] "Revier A" "Revier B" "Revier C" "Revier A" "Revier B" "Revier C"
461 ## [13] "Revier A" "Revier B" "Revier C"

```

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

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

```

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

```

467

468 Aufgabe 5: Statistische Funktionen

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

471 2. Erstellen Sie die folgende Konsolenausgabe:

```
472 ## [1] "Die mittlere Anzahl von Rehfotos beträgt 171.8 Rehe pro Standort."
```

473 3.4 Arbeiten mit logischen Werten

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

- 476 • Gleichheit (`==`)
- 477 • Ungleichheit (`!=`)
- 478 • Größer (`>`) und kleiner (`<`)
- 479 • Größer gleich (`>=`) und kleiner gleich (`<=`)

480 Das Ergebnis von logischen Operatoren ist immer `TRUE` oder `FALSE`.

481 Bei Vektoren kommt es immer zu einer elementweisen Anwendung. Wir können beispielsweise abfragen, an
482 welchen Fotofallenstandorten mehr als 100 Rehe fotografiert wurden:

```
anzahl_rehe > 100
```

```
483 ## [1] TRUE FALSE TRUE FALSE TRUE TRUE FALSE TRUE TRUE TRUE FALSE TRUE
```

```

484 ## [13] FALSE TRUE TRUE
485 Das Ergebnis ist ein Vektor vom Datentyp logi in der selben Länge wie anzahl_rehe.
486 Wir können z. B. abfragen, welche Fotofallenstandorte sich in Revier B befinden.
487 reviere == "Revier B"
488 ## [13] FALSE FALSE FALSE FALSE FALSE TRUE TRUE TRUE TRUE TRUE FALSE FALSE
489 Des Weiteren können logische Ausdrücke miteinander verknüpft werden. Dies geschieht mit einem logischen Und (&) oder einem logischen Oder (|). Für das logische Und müssen beide Ausdrücke ein TRUE zurückgeben
490 um ein TRUE zu erhalten. Für ein logisches Oder reicht es, wenn einer der beiden Ausdrücke TRUE zurückgibt,
491 um ein TRUE zu erhalten.
492
493 Damit können wir nun z. B. die beiden vorherigen Abfragen verbinden. Die erste Abfrage ist: Hat eine
494 Fotofalle mehr als 100 Rehe fotografiert und stand die Fotofalle in Revier B.
495 anzahl_rehe > 100 & reviere == "Revier B"
496 ## [13] FALSE FALSE FALSE FALSE FALSE TRUE FALSE TRUE TRUE TRUE FALSE FALSE
497 Das war jetzt eine Abfrage mit einem logischen Und. Würden wir ein logisches Oder verwenden, dann
498 bekommen wir für alle Elemente ein TRUE, die entweder in Gebiet B stehen oder mehr als 100 Rehfotos
499 aufgezeichnet haben.
500 anzahl_rehe > 100 | reviere == "Revier B"
501 ## [13] TRUE FALSE TRUE FALSE TRUE TRUE TRUE TRUE TRUE TRUE FALSE TRUE
502 Das Arbeiten mit logischen Werten kann fürs Erste etwas abstrakt erscheinen, aber wir werden im folgenden
503 Abschnitt (Abschnitt 3.5) zahlreiche Anwendungsbeispiele dafür sehen.

```

504

505 Aufgabe 6: Arbeiten mit logischen Werten

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

508 1. `TRUE | FALSE`
 509 2. `FALSE & TRUE`
 510 3. `(FALSE & TRUE) | TRUE`
 511 4. `(2 != 3) | FALSE`
 512 5. `FALSE + 10`
 513 6. `TRUE + 10`
 514 7. `TRUE + 10 == FALSE + 10`
 515 8. `sum(c(TRUE, TRUE, FALSE, FALSE))`

516 3.5 Zugreifen auf Elemente eines Vektors (=Untermengen)

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

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

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

```
anzahl_rehe[2]
```

530 ## [1] 79

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

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

532 ## [1] 132 79 129 91 138

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

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

536

537 Aufgabe 7: Zugreifen auf Vektorelemente

539 Erstellen Sie einen neuen Vektor `bhd`

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

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

543

544 Alternativ könnte das gleiche Ergebnis mit einem logischen Vektor erreicht werden. Für eine bessere Über-
 545 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]
```

546 ## [1] 132 79 129 91 138

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

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

550 ## [1] 132 79 129 91 138

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

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

553 ## [1] 132 79 129 91 138

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

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

556 ## [1] 113.8

557

558 Aufgabe 8: logische Werte

560 Verwenden Sie nochmals den Vektor mit den Anzahl Rehen die an unterschiedlichen Fotofallenstandorten
 561 fotografiert wurden.

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

562 1. Wählen Sie alle Standorte aus für die Aussage $90 \leq x < 120$ zu trifft (wobei x für die Anzahl Fotos an
 563 einem Standort steht).

564 2. Berechnen Sie die mittlere Anzahl Fotos für alle in 1) ausgewählten Standorte.

565 3.6 Der %in%-Operator

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

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

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

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

## [1] "FI" "FI"

# oder

messungen_arten[messungen_arten == arten[1]]
```

- 571 ## [1] "FI" "FI"
572 Etwas komplizierter wird es, wenn wir zwei oder mehr Elemente auswählen wollen. Dies geht auch mit
573 logischen Operationen.

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

- 574 ## [1] "FI" "BU" "BU" "FI"
575 Diese Herangehensweise wird aber für > 2 Elemente in `arten` sehr mühsam und fehleranfällig. Eine Alternative
576 bietet der `%in%`-Operator. Dieser testet, ob Elemente eines Vektors in einem zweiten Vektors enthalten sind.
577 Der Operator ist also eine verkürzte Schreibweise für hintereinander durchgeführte logische Oder Abfragen.

```
messungen_arten %in% arten
```

- 578 ## [1] TRUE TRUE TRUE FALSE FALSE FALSE TRUE FALSE FALSE
messungen_arten[messungen_arten %in% arten]

[1] "FI" "BU" "BU" "FI"

580

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

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

```
LETTERS
```

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

- 586 Wählen Sie aus `LETTERS` nur die Vokale aus.

587 4 Faktoren (factors)

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

594 Mit der Funktion **factor()** kann ein Faktor erstellt werden. Im einfachsten Fall wird nur ein Vektor übergeben.

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

595 ## [1] FI BU FI EI EI FI FI
 596 ## Levels: BU EI FI

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

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

600 ## [1] FI BU FI EI EI FI FI
 601 ## Levels: FI BU EI

602 Es ist auch möglich, die Beschriftung (= labels) der unterschiedlichen Levels anzugeben mit dem Argument
 603 **labels**.

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

604 ## [1] Fichte Buche Fichte Eiche Eiche Fichte Fichte
 605 ## Levels: Fichte Buche Eiche

606 Mit der Funktion **levels()**, können die unterschiedlichen Levels eines Faktors abgefragt und auch gesetzt
 607 werden.

```
levels(af)

## [1] "Fichte" "Buche"   "Eiche"
levels(af) <- c("Fi", "Bu", "Ei")
af
```

609 ## [1] Fi Bu Fi Ei Ei Fi Fi
 610 ## Levels: Fi Bu Ei

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

```
af
```

```
613 ## [1] Fi Bu Fi Ei Ei Fi Fi
614 ## Levels: Fi Bu Ei
  relevel(af, "Bu")
```

```
615 ## [1] Fi Bu Fi Ei Ei Fi Fi
616 ## Levels: Bu Fi Ei
```

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

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

```
619 ## [1] "Fi" "Bu" "Fi" "Ei" "Ei" "Fi" "Fi"
```

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

```
af
```

```
621 ## [1] Fi Bu Fi Ei Ei Fi Fi
622 ## Levels: Fi Bu Ei
```

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

```
623 ## [1] 1 2 1 3 3 1 1
```

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

626

Aufgabe 10: Faktoren

629 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")
```

631 4.1 Das Paket **forcats**

632 Mit dem Paket aus **forcats** werden einige Operationen mit Faktoren erleichtert. Wir schauen uns hier 633 Funktion an, die es erleichtern:

- 634 1. Die Anordnung von Levels anzupassen.
- 635 2. Levels zusammenzufassen oder zu entfernen.
- 636 3. Labels zu ändern.

637 **4.1.1 Anpassen der Anordnung von Faktoren**638 Wir verwenden nochmals den **a** Vektor mit unterschiedlichen Baumarten:

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

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

```
factor(a)
```

640 ## [1] FI BU FI EI EI FI FI

641 ## Levels: BU EI FI

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

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

```
f1
```

643 ## [1] FI BU FI EI EI FI FI

644 ## Levels: FI BU EI

645 **forcats** stellt Funktionen zur Verfügung, um die Reihenfolge umzukehren,

```
fct_rev(f1)
```

646 ## [1] FI BU FI EI EI FI FI

647 ## Levels: EI BU FI

648 nach der Häufigkeit zu sortieren oder

```
fct_infreq(f1)
```

649 ## [1] FI BU FI EI EI FI FI

650 ## Levels: FI EI BU

651 eine zufällige Sortierung.

```
fct_shuffle(f1)
```

652 ## [1] FI BU FI EI EI FI FI

653 ## Levels: EI FI BU

5 Spezielle Einträge

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

- 656 • fehlenden Einträgen NA,
- 657 • leeren Einträgen NULL,
- 658 • undefinierten Einträgen NaN (Not a Number) oder
- 659 • unendlichen Zahlen (Inf).

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

661 5.1 NA

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

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

```
str(na1)
```

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

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

```
str(na2)
```

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

667 Der logische Operatator zum Test auf fehlende Wert ist `is.na()`. Dieser kann genauso wie die bereits bekannten
668 logischen Operatoren bspw. zum Filtern verwendet werden. Die `na.omit()` Funktion entfernt NA aus dem
669 Datensatz.

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

```
670 ## [1] FALSE TRUE FALSE
```

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

```
671 ## [1] "foo" "foo"
```

```
672 ## attr(,"na.action")
```

```
673 ## [1] 2
```

```
674 ## attr(,"class")
```

```
675 ## [1] "omit"
```

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

```
na2 < 3
```

```
679 ## [1] FALSE FALSE NA
```

1 + NA

680 `## [1] NA`
681 Viele R Funktionen haben eingebaute Methoden zum Umgang mit NA. Die Funktion `mean()` bspw. ergibt
682 (wie die meisten Funktionen) standardmäßig NA wenn sie auf Vektoren mit Datenlücken angewendet wird, es
683 sei denn man stellt innerhalb der Funktion ein, dass Datenlücken entfernt werden sollen.

`mean(na2)`

684 `## [1] NA`
685 `mean(na2, na.rm = TRUE)`
685 `## [1] 4.5`

686 5.2 NULL

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

691 5.3 Inf

692 Die größtmögliche Zahl in R ist `1.7976931 * 10^308`. Größere Zahlen werden als unendlich gespeichert und
693 verarbeitet.

`10^309`

694 `## [1] Inf`
695 `2 * Inf`
695 `## [1] Inf`
696 `1 + Inf`
696 `## [1] Inf`
697 `3 / 0`
697 `## [1] Inf`
698 `-3 / 0`
698 `## [1] -Inf`
699 `3 / Inf`
699 `## [1] 0`

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

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

702 ## [1] TRUE FALSE FALSE TRUE FALSE
is.finite(inf1)

703 ## [1] FALSE TRUE TRUE FALSE TRUE
inf1 < 3

704 ## [1] FALSE TRUE FALSE TRUE FALSE
```

705

706 **Aufgabe 11: Vektoren mit speziellen Einträgen**
707

708 Verwenden Sie den Vektor

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

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

711 Verwenden Sie den Vektor

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

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

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

717 Berechnen Sie den arithmetischen Mittelwert von `foo`.

718 6 data.frames oder Tabellen

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

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

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

```
731 ##           ID anzahl_rehe   revier
732 ## 1    Kamera_1      132 Revier A
733 ## 2    Kamera_2       79 Revier A
734 ## 3    Kamera_3      129 Revier A
735 ## 4    Kamera_4       91 Revier A
736 ## 5    Kamera_5      138 Revier A
737 ## 6    Kamera_6      144 Revier B
738 ## 7    Kamera_7       55 Revier B
739 ## 8    Kamera_8      103 Revier B
740 ## 9    Kamera_9      139 Revier B
741 ## 10  Kamera_10     105 Revier B
742 ## 11  Kamera_11      96 Revier C
743 ## 12  Kamera_12     146 Revier C
744 ## 13  Kamera_13      95 Revier C
745 ## 14  Kamera_14     118 Revier C
746 ## 15  Kamera_15      107 Revier C
```

747 Auf der linken Seite der Gleichungen stehen die Spaltenname des Data Frames. Im vorhergehenden Codebeispiel
 748 wurde ein `data.frame` erstellt und in die Variable `monitoring` gespeichert. Die Funktion `data.frame()`
 749 nimmt als Argumente beliebig viele Paare, die immer aus einem Namen und einem Vektor mit dazugehörigen
 750 Werten bestehen. D.h., dass (wie bei den Vektoren) immer eine Spalte vom selben Typ sein muss, es aber
 751 für jede Beobachtung (=Zeile) Merkmale von unterschiedlichen Typen geben kann. Data Frames sind die

752 Standard-Objekte zum Speichern wissenschaftlicher Daten.

753 6.1 Wichtige Funktionen zum Arbeiten mit `data.frames`

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

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

```
756 ##           ID anzahl_rehe    revier
757 ## 1 Kamera_1          132 Revier A
758 ## 2 Kamera_2          79 Revier A
```

759 Oder für die letzten 2 Beobachtungen.

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

```
760 ##           ID anzahl_rehe    revier
761 ## 14 Kamera_14         118 Revier C
762 ## 15 Kamera_15         107 Revier C
```

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

```
nrow(monitoring)
```

```
764 ## [1] 15
```

```
ncol(monitoring)
```

```
765 ## [1] 3
```

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

```
str(monitoring)
```

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

```
772
```

773 Aufgabe 12: `data.frame`

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

6.2 Zugreifen auf Elemente eines `data.frame`

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

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

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

```
## [1] 91
```

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

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

```
## [1] 91
```

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

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

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

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

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

```
##   anzahl_rehe  revier
## 1          132 Revier A
## 2          79  Revier A
## 3         129 Revier A
## 4          91  Revier A
## 5         138 Revier A
```

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

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

```
##      ID anzahl_rehe  revier
## 1 Kamera_1          132 Revier A
## 2 Kamera_2           79  Revier A
## 3 Kamera_3          129 Revier A
```

```
809 ## 4 Kamera_4          91 Revier A
810 ## 5 Kamera_5          138 Revier A
```

811

812 Aufgabe 13: Abfragen von Werten

814 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)
)
```

- 815 • Wählen Sie nur die ersten drei Zeilen aus und die erste und zweite Spalte aus.
 816 • Wählen Sie alle Zeilen und die erste und dritte Spalte aus.
 817 • Wählen Sie alle Spalten und die erste, dritte und vierte Zeile aus.

818

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

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

```
monitoring$anzahl_rehe
```

822 ## [1] 132 79 129 91 138 144 55 103 139 105 96 146 95 118 107

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

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

824 ## [1] 132 79 129 91 138 144 55 103 139 105 96 146 95 118 107

825 3. Alle Zeilen und die Spalte explizit angeben.

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

826 ## [1] 132 79 129 91 138 144 55 103 139 105 96 146 95 118 107

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

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

```
monitoring$anzahl_rehe > 100
```

```
833 ## [1] TRUE FALSE TRUE FALSE TRUE TRUE FALSE TRUE TRUE TRUE FALSE TRUE  
834 ## [13] FALSE TRUE TRUE
```

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

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

```
839 ##          ID anzahl_rehe  revier  
840 ## 1   Kamera_1      132 Revier A  
841 ## 3   Kamera_3      129 Revier A  
842 ## 5   Kamera_5      138 Revier A  
843 ## 6   Kamera_6      144 Revier B  
844 ## 8   Kamera_8      103 Revier B  
845 ## 9   Kamera_9      139 Revier B  
846 ## 10 Kamera_10     105 Revier B  
847 ## 12 Kamera_12     146 Revier C  
848 ## 14 Kamera_14     118 Revier C  
849 ## 15 Kamera_15     107 Revier C
```

850

851 Aufgabe 14: Abfragen von Werten 2

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

- 854 • Alle Spalten für Studierende die Forstwissenschaften studieren.
- 855 • Alle Spalten für Studierende die Chemie oder Physik studieren.
- 856 • Die Spalte `fach` und `semester` für Studierende die 22 oder älter sind.

857 7 Schreiben und lesen von Daten

858 7.1 Textdateien

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

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

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

874 Die Datei `fotofallen.csv` finden Sie auf StudIP und kann einfach heruntergeladen werden. Sie können
 875 sich die Datei mit jedem Texteditor oder auch mit Excel oder Libre Office ansehen. Die Datei kann mit
 876 dem folgenden Befehl in R eingelesen werden. Hier wurde die Datei in einem R Studio-Projekt in ein
 877 Unterverzeichnis `data` abgelegt.

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

```
878 ##          ID anzahl_rehe   revier  
879 ## 1 Kamera_1        132 Revier A  
880 ## 2 Kamera_2        79 Revier A  
881 ## 3 Kamera_3        129 Revier A  
882 ## 4 Kamera_4        91 Revier A  
883 ## 5 Kamera_5        138 Revier A  
884 ## 6 Kamera_6        144 Revier B
```

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

889 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.

890

891 **Aufgabe 15: Lesen und Schreiben von Datein**
892

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

8 Erstellen von Abbildungen

896 Abbildungen sind ein elementarer Baustein statistischer Analysen und deshalb von Beginn an Teil von R. **R is
897 a free software environment for statistical computing and graphics.** Es gibt unterschiedliche Systeme
898 einen Plot zu erstellen. In diesem Kurs werden wir kurz *Base Plots* vorstellen und dann das Zusatzpaket
899 **ggplot2** vorstellen.
900

901 8.1 Base Plot

902 Die wichtigsten Grafiken für die einfache Datendarstellung sind schnell verfügbar. Etwas komplexere oder
903 spezielle Grafiken erfordern mehr Programmieraufwand (folgt teilweise noch). Für viele komplexere Diagramme
904 existieren bereits Codebeispiele oder Pakete. Es lohnt sich z. B. einen Blick in die R Graph Gallery (<https://r-graph-gallery.com/index.html>) zu werfen, bevor Sie beginnen komplexe Abbildungen zu erstellen. Stellen
905 sie sich die einfache Grafik Schnittstelle (**base plots**) als zweidimensionale Leinwand vor, auf die Sie durch
906 Code Ebene für Ebene Grafikelemente legen:
907

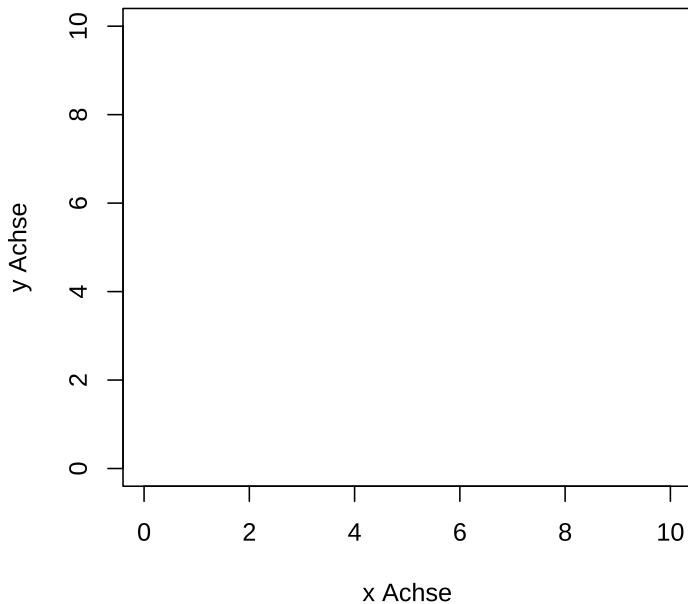
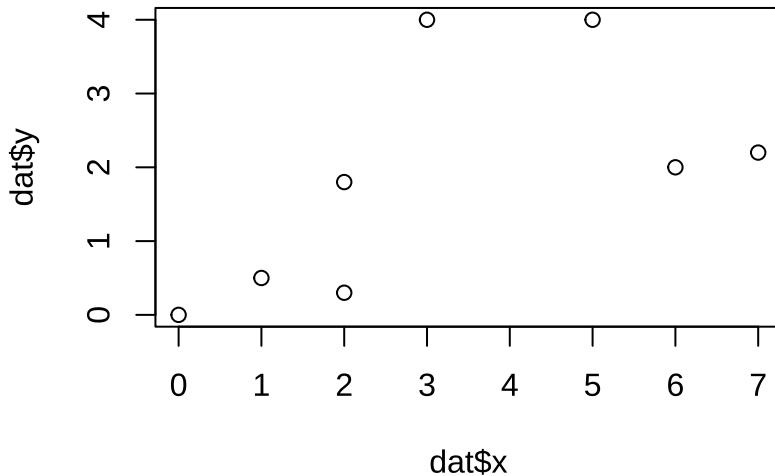


Abbildung 4: Beispiel einer leeren Grafikschnittstelle.

908 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)  
)
```

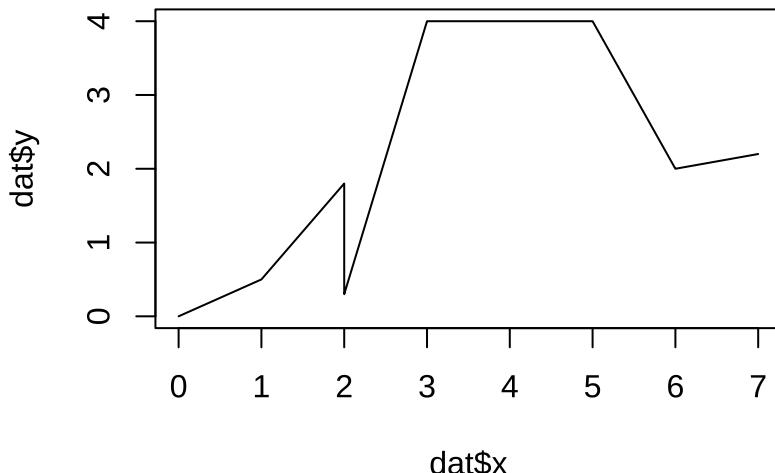
```
plot(dat$x, dat$y, type = "p")
```



909

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

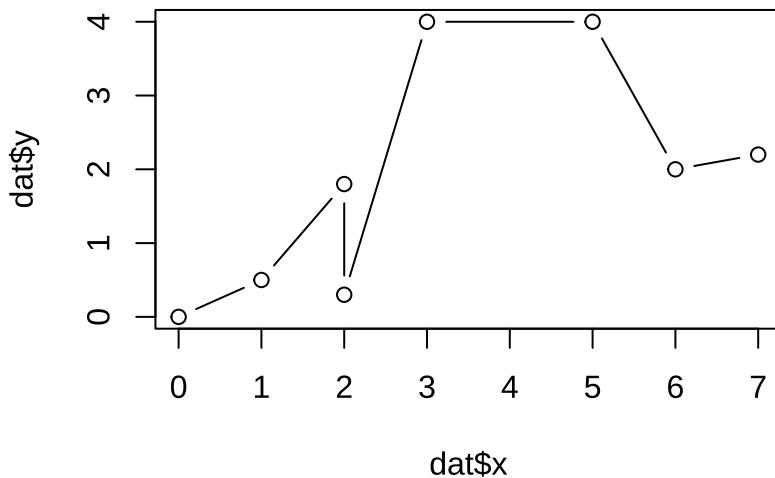
```
plot(dat$x, dat$y, type = "l")
```



912

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

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



914

915 darstellen.

916

917 **Aufgabe 16: Base Plot 1**

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

921

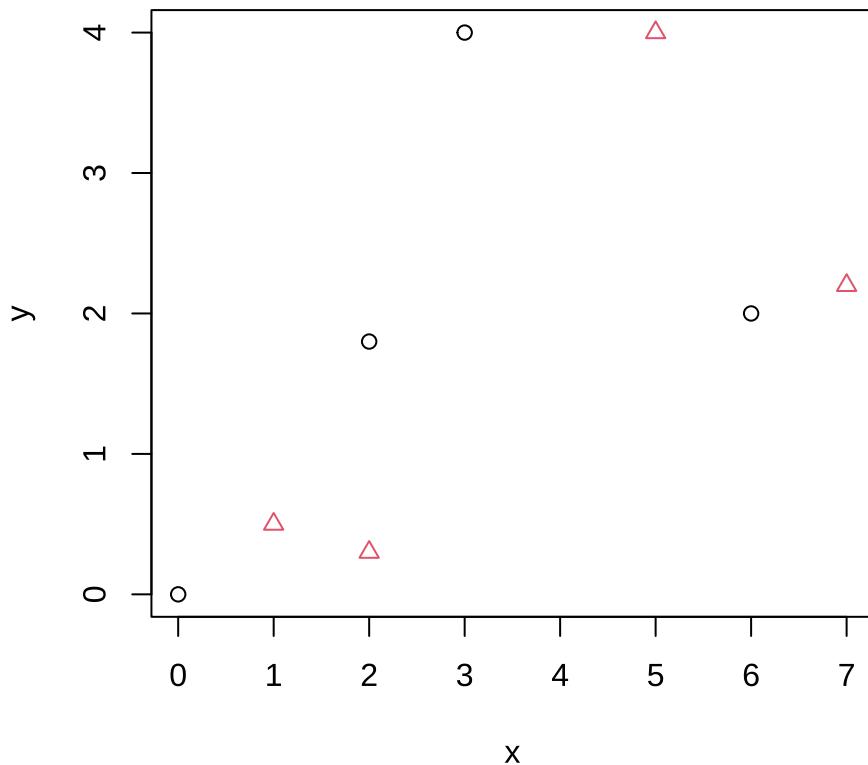
922 Sie können entweder eine Grafik mit einem Befehl erzeugen (High-Level) oder die einzelnen Ebenen nacheinan-
923 der erzeugen (Low-Level). Sie können jeder Ebenen durch zusätzliche Befehle innerhalb des Funktionsaufrufs
924 Elemente hinzufügen. Mit jeder neuen Ebene können Sie die vorigen Ebenen jedoch nicht mehr verändern.

925 Die wichtigsten Argumente der `plot` Funktion sind:

- 926 • `type` - Diagrammtyp
927 • `col` - Farbe
928 • `main` - Titel
929 • `sub` - Untertitel
930 • `pch` - Punktsymbol
931 • `lty` - Linientyp
932 • `lwd` - Linienstärke
933 • `xlab` bzw. `ylab` - Achsenbeschriftungen
934 • `xlim`, `ylim` - Grenzen der Achsenanschnitte

- 935 • **axes** - Sollen die Achsen eingezeichnet werden? Oder leer gelassen werden, um sie nachträglich als
936 low-level Ebene einzuzeichnen?
937 • **ann** - Achsenbeschriftung kann ganz weggelassen werden.
938 Sehen Sie sich die Hilfeseiten `?plot.default()` oder `?par()` an für weiter Informationen. Dort finden Sie
939 auch eine vollständige Liste der Befehle. Einige Argumente können als Vektor übergeben werden. Hier z. B.
940 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")
```



941

942

943 **Aufgabe 17: Anpassen von Plots**

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

- 946 • Beschriften Sie die x- und y-Achse sinnvoll.
947 • Fügen Sie eine Überschrift hinzu.
948 • Wählen Sie ein anderes Symbol.
949 • Stellen Sie die Symbole in rot dar.

950

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

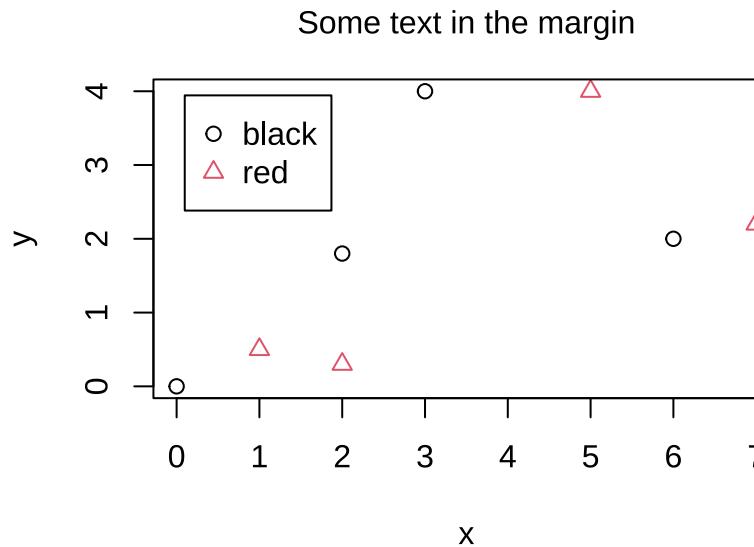
952 Die wichtigsten Funktionen sind

- 953 • `points()` - Fügt Punkte ein
954 • `lines()` - Fügt Linien ein
955 • `text()` - Fügt Text ein
956 • `mtext` - Fügt Text in den Rahmen (`margin`) ein
957 • `legend()` - Fügt eine Legende ein
958 • `abline()` - Fügt eine Gerade ein
959 • `curve()` - Fügt eine mathematische Funktion ein
960 • `arrows()` - Fügt Pfeile ein
961 • `grid()` - Fügt Hilfslinien ein

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

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

```
dat <- data.frame(  
  x = c(0, 1, 2, 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")
```



968

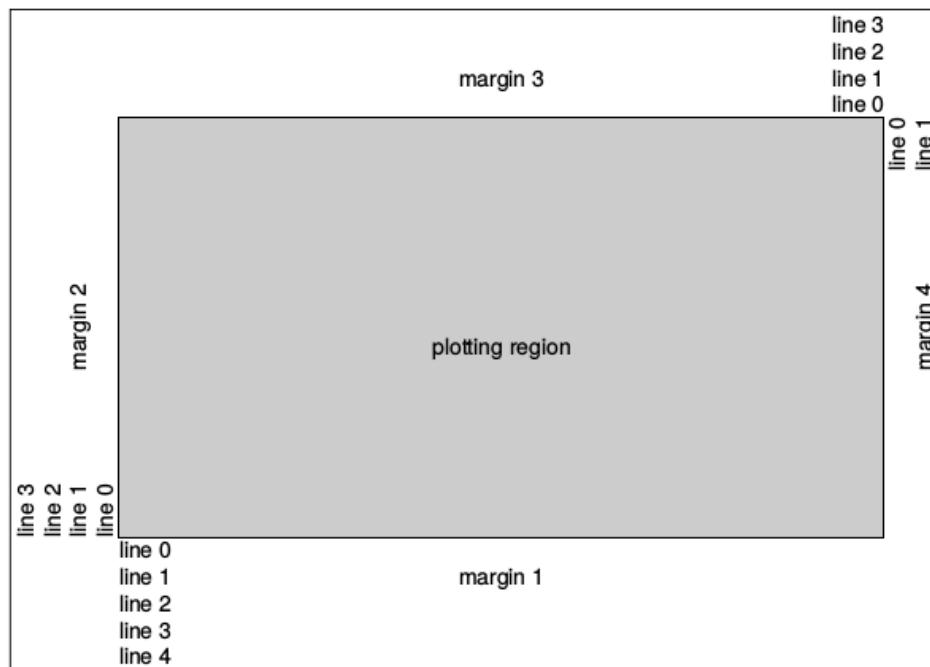


Abbildung 5: Grafikregionen eines base plots in R.

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

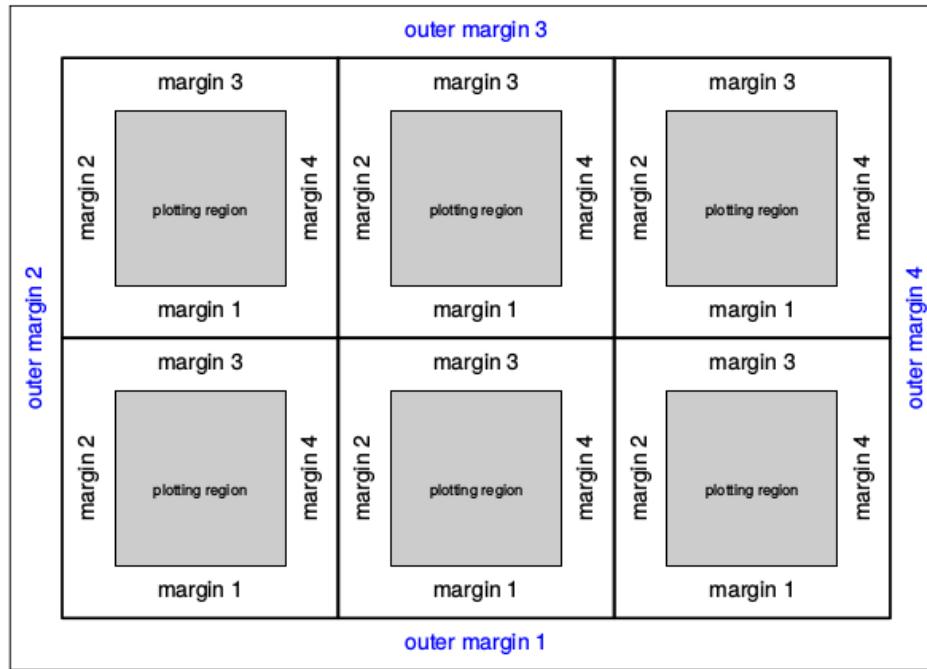


Abbildung 6: Schematischer Aufbau mehrere Diagramme in einem plot am Beispiel einer 3×2 Grafik.

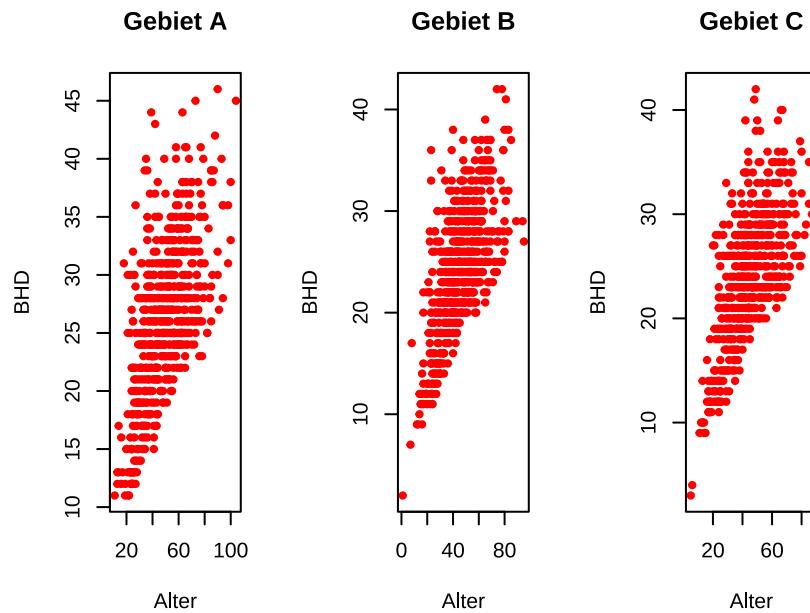
973 8.1.1 Mehrere Panels

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

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

977 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")
```



978

979 Vergessen Sie nicht am Ende nochmals `par(mfrow = c(1, 1))` zu setzen, damit wieder nur ein Plot angezeigt
 980 wird.

981 8.1.2 Speichern von Abbildungen

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

- 986 • `pdf()` oder
- 987 • `postscript()`.

988 Beispiele für Rastergrafiken sind

- 989 • `png()`,
- 990 • `bmp()` oder
- 991 • `jpeg()`.

992 Die Grafikschnittstelle ist dann Ihre "Leinwand". Mit dem Befehl `dev.off()` trennen Sie die Verbindung zur
 993 Schnittstelle wieder. Ihre "Leinwand" wird also wieder geschlossen. So lange die Schnittstelle geöffnet ist
 994 werden alle Low-Level Befehle an die Ausgabedatei gesendet. Hier am Beispiel einer PDF.

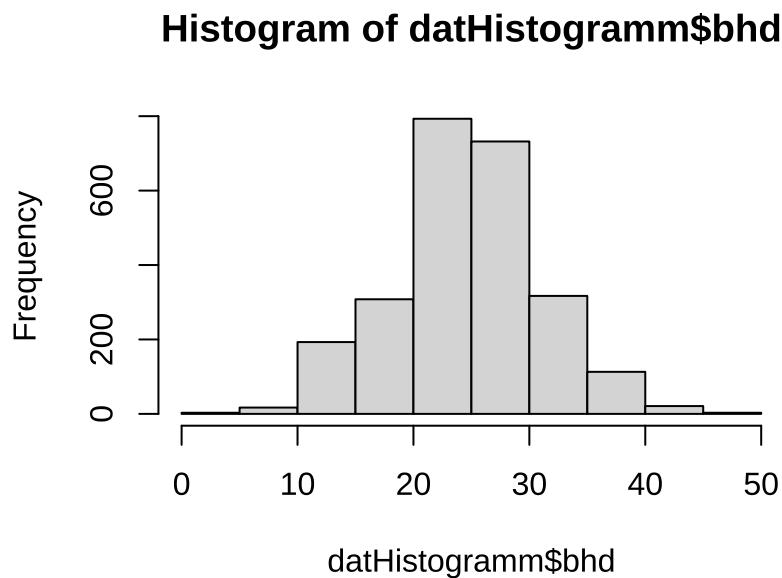
```
pdf("Grafik.pdf", height = 5)           # Öffnen der PDF Schnittstelle
plot(bhd ~ alter, type = "b", axes = FALSE,   # Abbildung produzieren, Ohne Achsen
     data = dat)
axis(side = 1, line = 1)                  # Achsen als High-Level Funktion hinzufügen
axis(side = 2, line = 1, las = 2)      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

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

998 8.2 Histogramme

999 Neben den Streudiagrammen (*Scatterplots*, oder auch einfach x-y Diagramm) sind *Histogramme* in der
 1000 angewandten Datenanalyse ein weiterer wichtiger Abbildungstyp. An Histogrammen wird die Häufigkeit
 1001 von Beobachtungen nach Gruppen dargestellt. Sie sind deshalb so wichtig, weil man aus ihnen relevante
 1002 Informationen über die Verteilung der Daten ablesen kann und somit einiges über die Messungen erfährt,
 1003 das für die weitere Datenanalyse relevant sein kann. So werden auf einen Blick der Zusammenhang von
 1004 Beobachtungshäufigkeit und Streuung deutlich, sowie auch die Form der Verteilung und ihre Schiefe. Die
 1005 Interpretation werden wir bei den Boxplots noch weiter vertiefen.

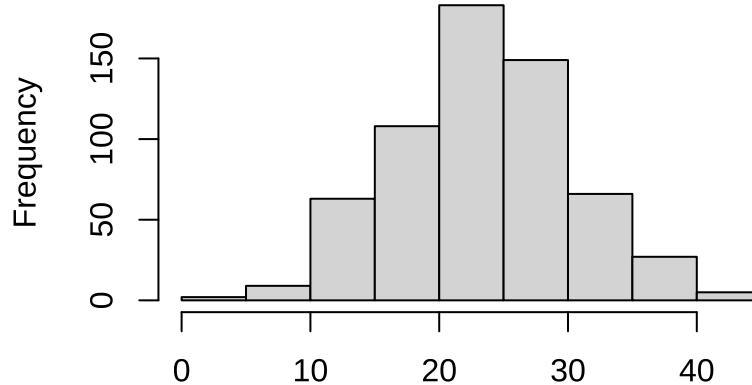
```
datHistogramm <- read.table("data/bhd_1.txt", header = TRUE)
# Über alle Baumarten
hist(datHistogramm$bhd)
```



1006

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

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

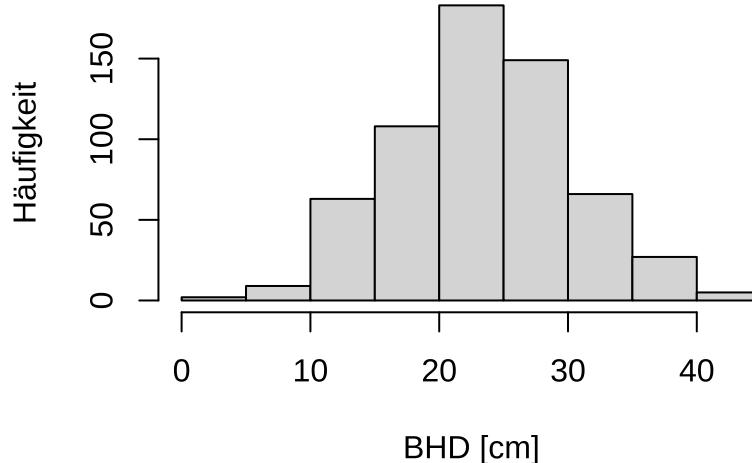


```
datHistogramm$bhd[datHistogramm$art == "EI"]
```

1007

```
# 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



1008

1009 Eichen und Buchen im 2x1 Plot nebeneinander.

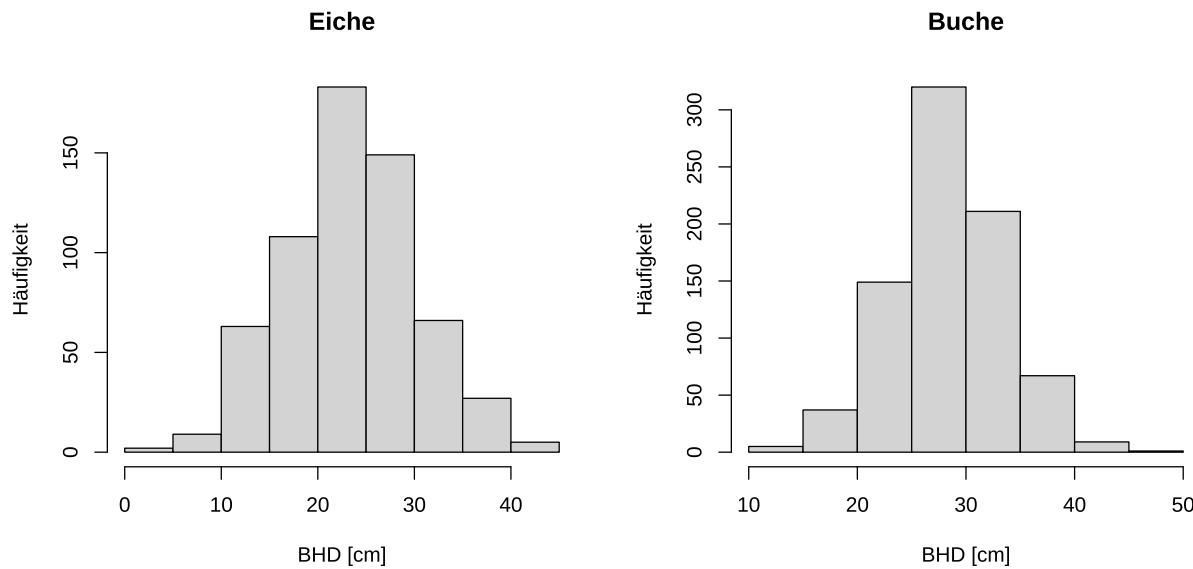
```
par(mfrow = c(1, 2))
hist(datHistogramm$bhd[datHistogramm$art == "EI"],
```

```

xlab = "BHD [cm]", ylab = "Häufigkeit",
main = "Eiche")
hist(datHistogramm$bhd[datHistogramm$art == "BU"],
xlab = "BHD [cm]", ylab = "Häufigkeit",
main = "Buche")

```

1010



1011

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

1012 8.3 Boxplots

1013 Oft möchte man die Verteilung einer stetigen Variablen in Abhängigkeit einer diskreten Variable beschreiben
 1014 oder visualisieren. Ein Beispiel dafür wäre die BHD-Verteilung in Abhängigkeit der Baumarten. Eine häufige
 1015 Darstellungsform für solche Daten sind *Boxplots*. Sie stellen die durchschnittliche Ausprägung einer stetigen
 1016 Variable und ihre Schwankung kompakt dar.

1017 Boxplots bestehen aus drei Komponenten:

- 1018 1. Eine *Box*, die den Bereich zwischen 0.25 und 0.75 Percentil abdeckt, diese Distanz wird auch die *IQR*
 1019 (Interquartile Range), bezeichnet. Zusätzlich wird die Box durch den Median (als dicke horizontale Linie)
 1020 unterteilt.
 - 1021 2. Einzelne Punkte Ausreißer. Als Ausreißer werden Punkte bezeichnet, die $> 1.5 \text{IQR}$ vom unteren oder
 1022 oberen Ende der Box entfernt sind.
 - 1023 3. Eine senkrechte Linie von jeder Seite der Box bis zum letzten „Nicht-Ausreißer-Punkt“. Also der letzte
 1024 Punkt, der $> 1.5 \text{IQR}$ aber nicht > 0.75 bzw. < 0.25 Percentil ist. Diese Linie wird auch als *Whisker*
 1025 bezeichnet.
- 1026 Mit R kann mit der Funktion `boxplot()` ein Boxplot erstellt werden. Diese Funktion kann in zwei unter-
 1027 schiedlichen Ausprägungen verwendet werden.

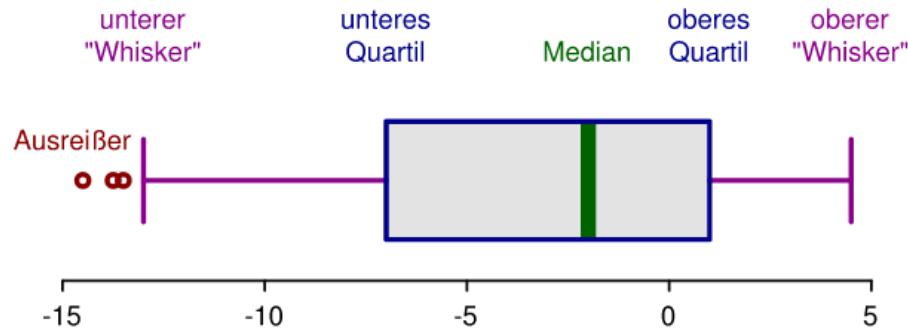
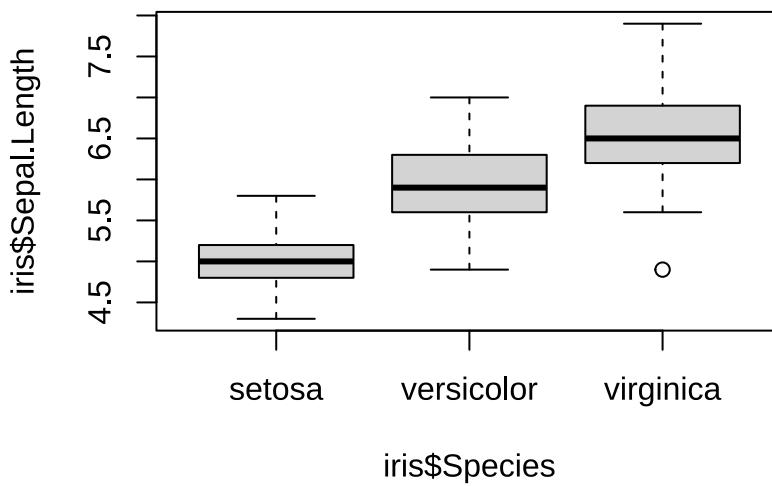


Abbildung 7: Schematische Darstellung eines Boxplots (Quelle: Von RobSeb - Eigenes Werk, CC BY-SA 3.0, <https://commons.wikimedia.org/w/index.php?curid=14697172>).

- 1028 1. `boxplot(x)` erzeugt einen Boxplot für die Variable `x`.
- 1029 2. `boxplot(x ~ y)` erzeugt einen oder mehrere Boxplots für `x` aber gruppiert nach `y`, dabei sollte `y` eine
- 1030 kategorische Variable sein. `x` und `y` können auch die Spaltennamen eines `data.frames` sein, dann muss
- 1031 das `data.frame` mit dem Argument `data` zusätzlich übergeben werden.

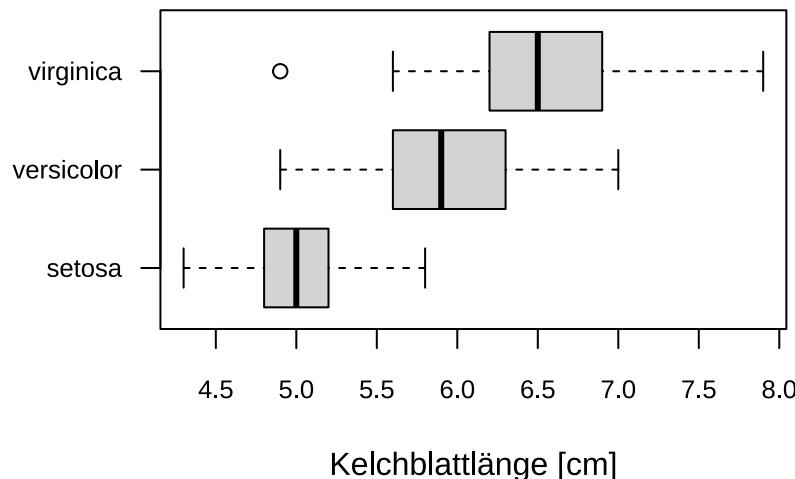
```
boxplot(iris$Sepal.Length ~ iris$Species)
```



- 1032 Etwas eleganter ist es wenn wir das Argument `data` verwenden und den Plot etwas anpassen. Diese Schreibweise funktioniert für alle base plots.

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

)



1035

1036

1037 Aufgabe 18: Boxplots

1038

- 1039 • Lesen Sie erneut den Datensatz `daten/bhd_1.txt` ein und speichern Sie diesen in die Variable (`dat_bhd`).
 - 1040 • Wie viele BHD-Messungen gibt es für jedes Gebiet?
 - 1041 • Erstellen Sie für jedes Gebiet einen Plot
- 1042 Erstellen Sie einen Plot mit 3 Subplots, jeweils mit einem Boxplot für die ersten drei Studiengebiete, in dem
1043 der BHD für jede Baumart dargestellt wird.

1044 8.4 ggplot2: Eine Alternative für Abbildungen

1045 `ggplot2` ist ein alternatives Plotting-System in *R*. Sie können mit `ggplot2` also grundsätzlich Abbildungen
1046 mit dem selben Inhalt erstellen, wie mit Base Plots. Die Syntax und die optische Darstellung unterscheiden
1047 sich jedoch grundsätzlich. `ggplot2` basiert auf den *grammar of graphics* von Leland Wilkinson. Die Idee
1048 ist, alle nötigen Informationen der Abbildung miteinander zu verknüpfen. `ggplot2` ist also diametral zu
1049 Base Plots. Während Sie in base plot alle Elemente selbst definieren, ist die Idee von `ggplot2`, dass Sie
1050 nur die Daten und den Diagrammtypen übergeben und die Funktion selbst eine schöne Abbildung erstellt.
1051 Selbstverständlich können Sie aber auch in `ggplot2` viele Einstellungen vornehmen. Im base plot sehen
1052 Abbildungen zunächst sehr karg und etwas unfertig aus. Sie müssen viele Einstellungen vornehmen, um eine
1053 publizierfähige Grafik zu produzieren. In `ggplot2` sollen auch die einfachste Abbildungen schon ästhetisch
1054 sein. Mit diesen gebündelten Informationen kann `ggplot2` die Abbildung automatisch verschönern. So
1055 werden bspw. die Legenden automatisch erzeugt und auch die Formatierungen automatisch an die Datenlage
1056 angepasst. `ggplot2` nimmt der*dem Entwickler*in also Arbeit ab. Dadurch sind die Abbildungen schon ohne

1057 viel Nacharbeit schick. Nachteil ist, dass der*dem Entwickler*in weniger Möglichkeiten zur Einstellung zur
 1058 Verfügung stehen und nuterspezifische Sonderwünsche somit schwerer umsetzbar sind. Sehen Sie sich das
 1059 *Cheatsheet* zu ggplot2 an. Es ist in RStudio unter [Help > Cheatsheets](#) zu finden.

1060 Bei ggplot2 sind Anweisungen zu den Daten und Anweisungen zur Darstellung voneinander getrennt. Die
 1061 Daten werden in den Ästhetikbefehl übergeben und dort klassifiziert. Dann folgen die Darstellungsanweisungen.
 1062 Ähnlich wie bei Base Plots, werden die Grafikelemente ebenenweise nacheinander programmiert, jedoch mit
 1063 einem + verbunden. Und hier liegt der wesentliche Unterschied zu Base Plots. Durch die + werden die Ebenen
 1064 zu einem Befehl verbunden und damit gleichzeitig erstellt.

1065 Die Erweiterung wird zunächst geladen⁷. Wir laden außerdem den Datensatz `iris`. Der Datensatz ist in R
 1066 fest integriert. Siehe `?iris` für mehr Informationen.

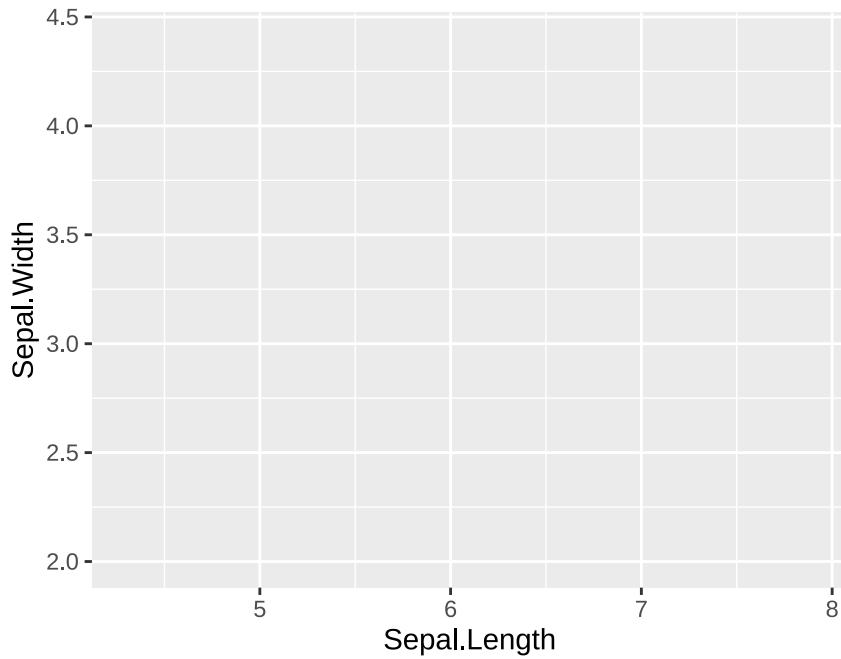
```
library(ggplot2)
head(iris)
```

1067 ## Sepal.Length Sepal.Width Petal.Length Petal.Width Species
 1068 ## 1 5.1 3.5 1.4 0.2 setosa
 1069 ## 2 4.9 3.0 1.4 0.2 setosa
 1070 ## 3 4.7 3.2 1.3 0.2 setosa
 1071 ## 4 4.6 3.1 1.5 0.2 setosa
 1072 ## 5 5.0 3.6 1.4 0.2 setosa
 1073 ## 6 5.4 3.9 1.7 0.4 setosa

1074 Die Ästhetik wird bspw. folgendermaßen definiert.

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

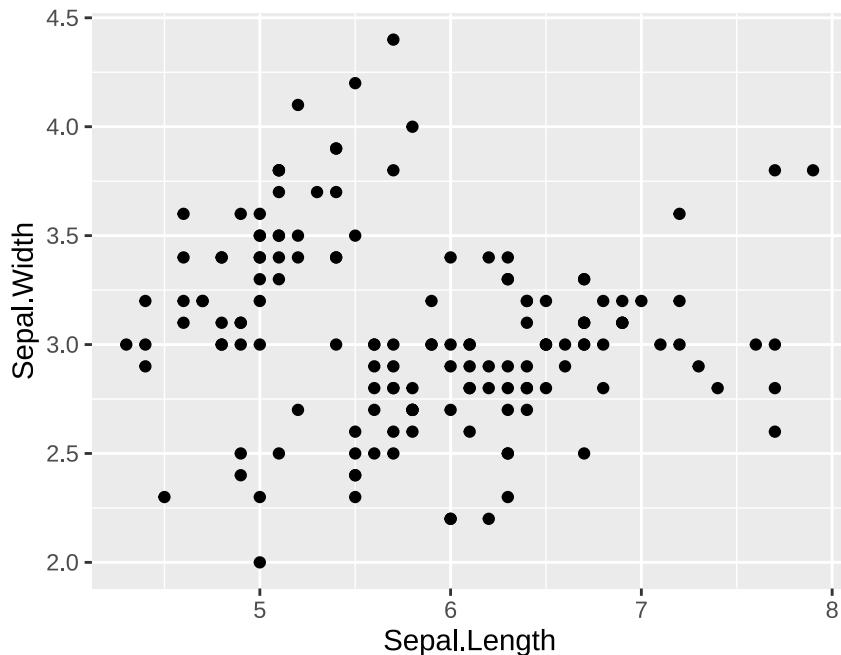
⁷Wir haben bis jetzt immer nur mit *base* R gearbeitet. D.h. wir haben nur Funktionen verwendet, die R bereits zur Verfügung stellt. Eine der großen Stärken von R sind die Erweiterungen (oder auch Pakete genannt). `ggplot2` ist so eine Erweiterung, die einmal mit `install.packages("ggplot2")` installiert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



1075

1076 Dieser Befehl zeichnet noch keine Daten. Die Daten werden lediglich herangezogen, um einen leeren Plot für
 1077 die Daten zu erstellen. In dem Beispiel wird die Variable `Sepal.Length` aus dem `data.frame iris` als x und
 1078 `Sepal.Width` als y Variable definiert. Diese Informationen stehen den folgenden Layern nun zur Verfügung,
 1079 sodass nach den + nur noch x und y verwendet werden müssen. Um bspw. einen Scatterplot zu erstellen
 1080 wird ein `geom_point()` Layer hinzugefügt. x und y werden automatisch an `geom_point` übergeben. Weitere
 1081 Einstellung sind in diesem Beispiel nicht notwendig, wären jedoch möglich. Siehe `?geom_point()`.

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



1082

1083

1084 **Aufgabe 19: Abbildungen mit ggplot2**

1085

1086 Verwenden Sie die Daten aus Aufgabe 16 und erstellen Sie einen Scatterplot mit ggplot2 wie in Aufgabe 16.

1087

1088 Wir haben mit der Funktion `geom_point()` demm Plot eine Punktgeometrie hinzugefügt. Es gibt noch viele
1089 weitere Geometrien. Die wichtigsten sind:

-
- 1090 •
- `geom_line()`
- für eine Linie.
-
- 1091 •
- `geom_histogram()`
- um ein Histogramm zu erstellen.
-
- 1092 •
- `geom_boxplot()`
- um einen Boxplot zu erstellen.
-
- 1093 •
- `geom_bar()`
- um ein Säulendiagramm zu erstellen.

1094 Welche Geometrie die richtige ist, richtet nach dem Typ der darzustellenden Variablen. Beispielsweise
1095 bietet sich `geom_point()` an, wenn man zwei kontinuierliche Variable darstellen möchte. Wenn man hin-
1096 gegen die Verteilung von einer kontinuirlchen Variable darstellen möchte, dann bietet sich ein Histogramm
1097 (`geom_histogram()`) oder auch eine geschätzte Dichte (z.B. `geom_density()`) an.

1098

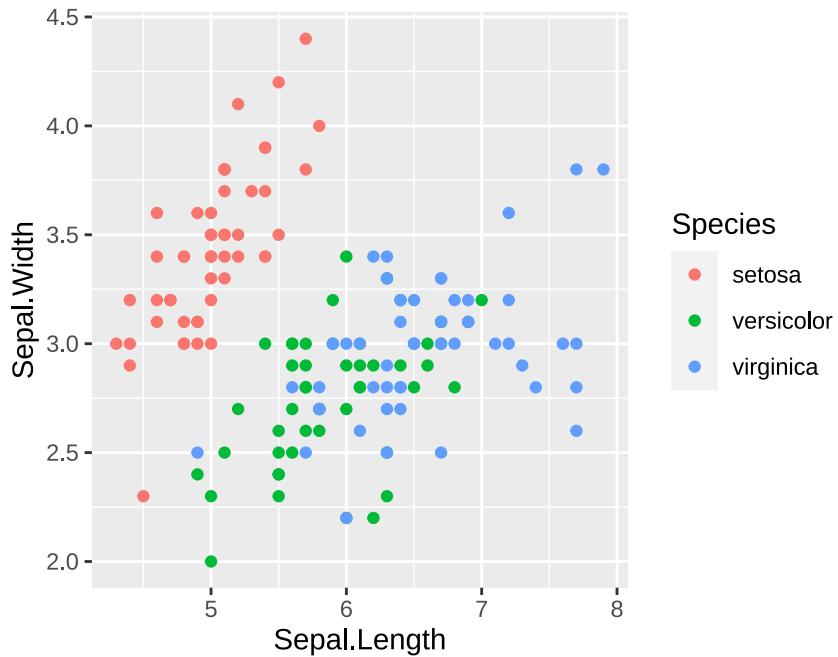
1099 **Aufgabe 20: Abbildungen mit ggplot2**

11001101 Verwenden Sie die den Iris Datensatz und erstellen Sie mit ggplot2 einen Plot der die Verteilung der Länge
1102 der Kelchblätter zeigt (Spalte `Sepal.Length`).

1103

1104 Eine der Stärken von ggplot2 ist, dass man den Wert unterschiedlicher Variable auf unterschiedlichen
1105 Komponenten des Plots abbilden kann. Wir haben bis jetzt ein bzw. zwei Variable auf der x- und y-Achse
1106 abgebildet. Wir können aber ein weitere Variablen verwenden um das Aussehen des Plots zu beeinflussen.
1107 Beispielsweise können wir die Farbe der Punkte (für `geom_point()`) mit dem Argument `col` beeinflussen.

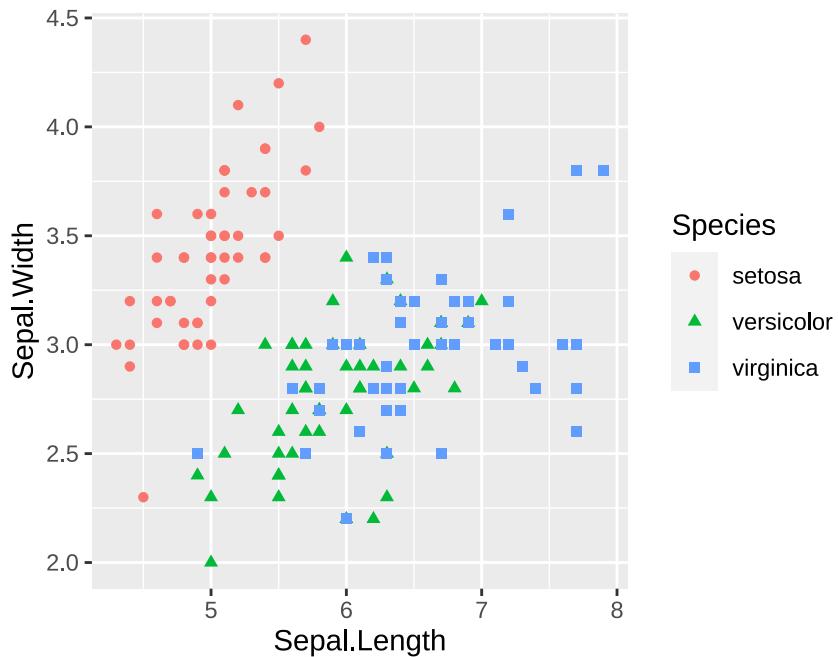
```
ggplot(iris, aes(x = Sepal.Length, y = Sepal.Width, col = Species)) +  
  geom_point()
```



1108

- 1109 Somit bekommt jede Irisart eine eigene Farbe⁸. Gleichermaßen können wir die Punktart (`shape`), die
1110 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()
```



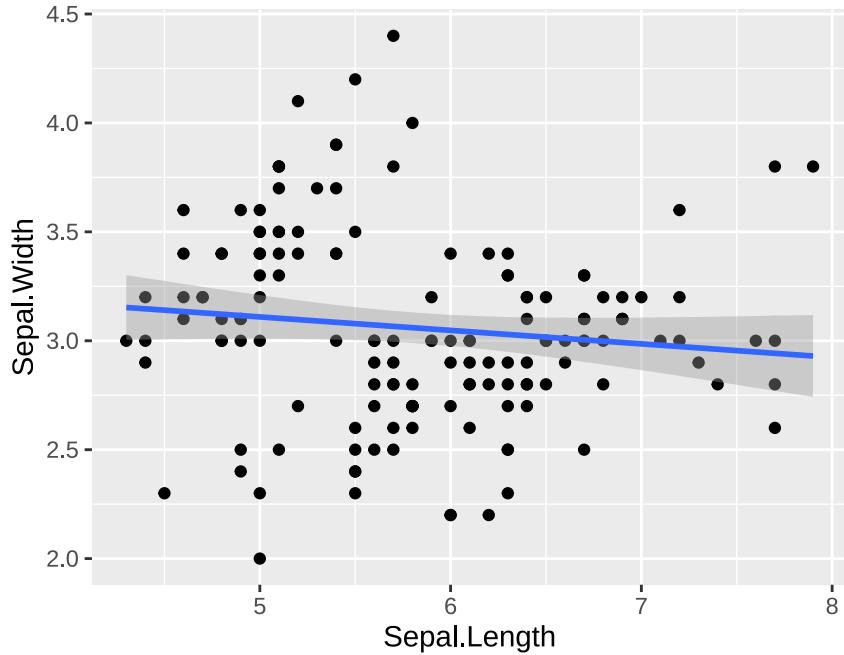
1111

- 1112 In dem Plot ist die Information zu der Art redundant (einmal als Farbe und einmal Symbolart).

⁸Natürlich könnte man auch die Farbe anpassen.

1113 Ein weitere sehr nützliche Geometrie ist `geom_smooth()`, die es erlaubt eine Trendlinie hinzuzufügen.

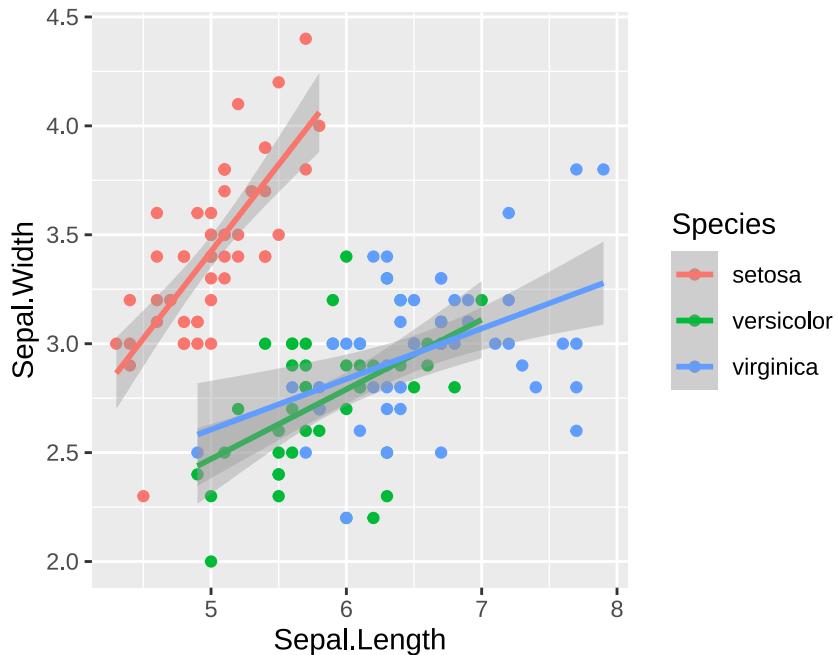
```
ggplot(iris, aes(x = Sepal.Length, y = Sepal.Width)) +
  geom_point() + geom_smooth(method = "lm")
```



1114

1115 Mit `method = "lm"` wird festgelegt, dass die Trendlinie gerade sein soll (es wird eine lineare Einfachregression angepasst). Wenn wird wieder eine gruppierende Variable einführen (z.B. die Beobachtungen nach Art auf die Farbe aufteilen), wir das von `geom_smooth()` berücksichtigt.

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



1118

1119

Aufgabe 21: Anpassen von Plots

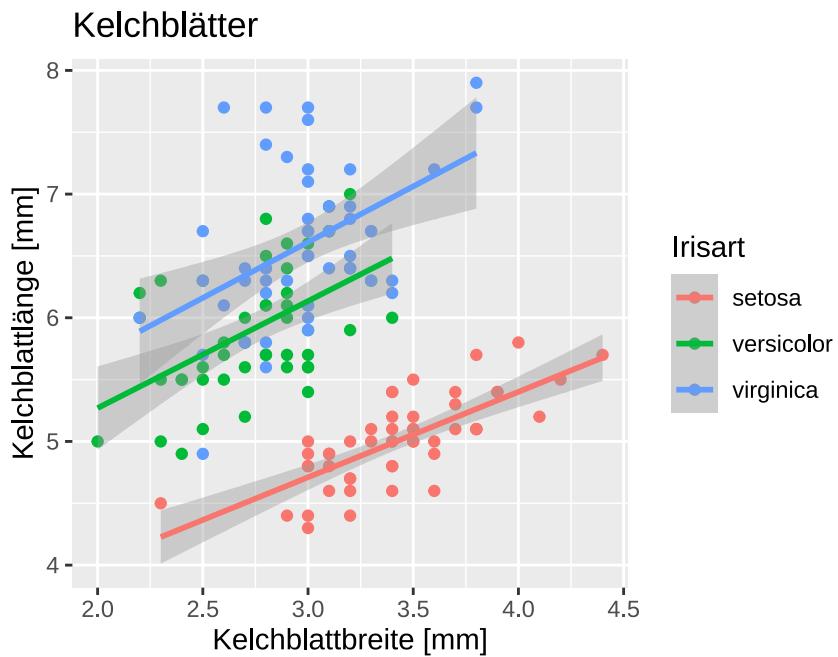
- 1122 Lesen Sie den Datensatz `data/bhd_1.txt` ein und erstellen Sie einen Boxplot für die Verteilung des BHDs
 1123 für jede Baumart. In einem zweiten Schritt verwenden Sie erst `col = gebiet` und dann `fill = gebiet`.
 1124 Welchen Unterschied stellen Sie fest?

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

1125

- 1126 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")
```



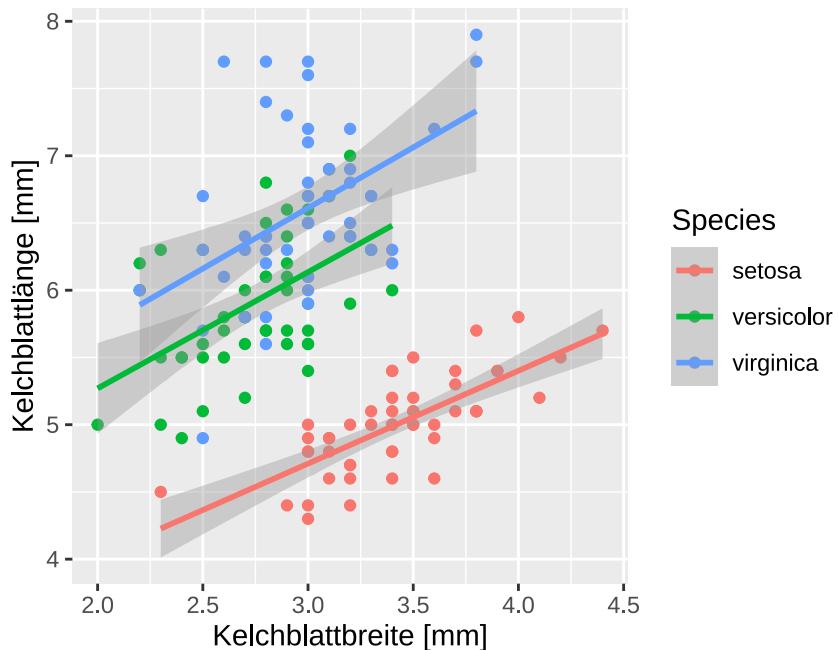
1127

1128 Statt einen langen Befehl zu tippen, kann ein `ggplot()` auch zwischengespeichert und wieder aufgerufen bzw.
 1129 angepasst werden. Das ist vor allem sinnvoll, wenn mehrere Abbildungen auf dem selben Zwischenergebnis
 1130 aufbauen sollen.

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

1131 Wir können jetzt mit `p1` weiter arbeiten und beispielsweise eine Beschriftung hinzufügen.

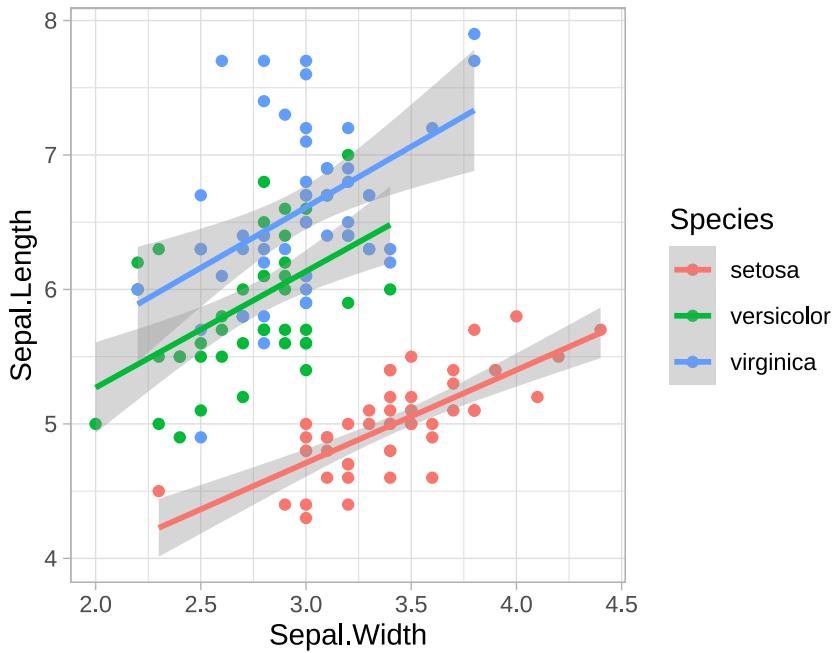
```
p1 + labs(x = "Kelchblattbreite [mm]", y = "Kelchblattlänge [mm]")
```



1132

1133 Oder auch den ganzen Plot anpassen. Dafür gibt es *themes*. Es gibt eine Reihe von vorgefertigten *themes*
1134 oder man kann diese auch selber erstellen (das ist aber nicht Teil dieses Kurses).

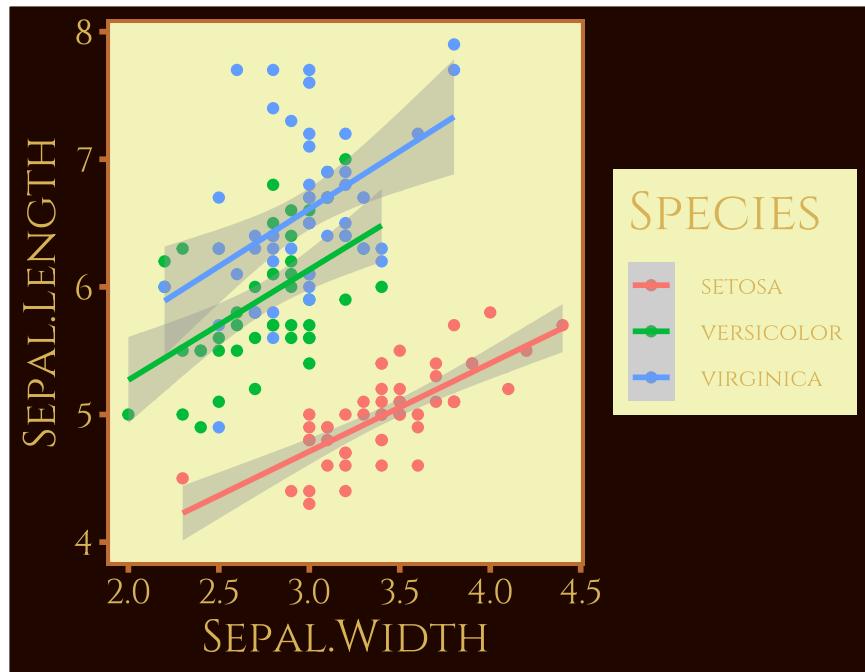
```
p1 + theme_light()
```



1135

1136 Weitere *themes* sind: `theme_bw()`, `theme_linedraw()` oder `theme_dark()`. Es gibt extra Pakete die viele
1137 zusätzliche weitere *themes* anbieten. Dazu gehört z.B. das Paket `ggthemes` oder `ThemePark`. Während
1138 `ggthemes` hauptsächlich durchdachte Grafikkonzepte liefert, sind die Themes aus `ThemePark` eher Popkultur
1139 und nicht 100 %ig ernst gemeint.

```
p1 + ThemePark::theme_gameofthrones()
```

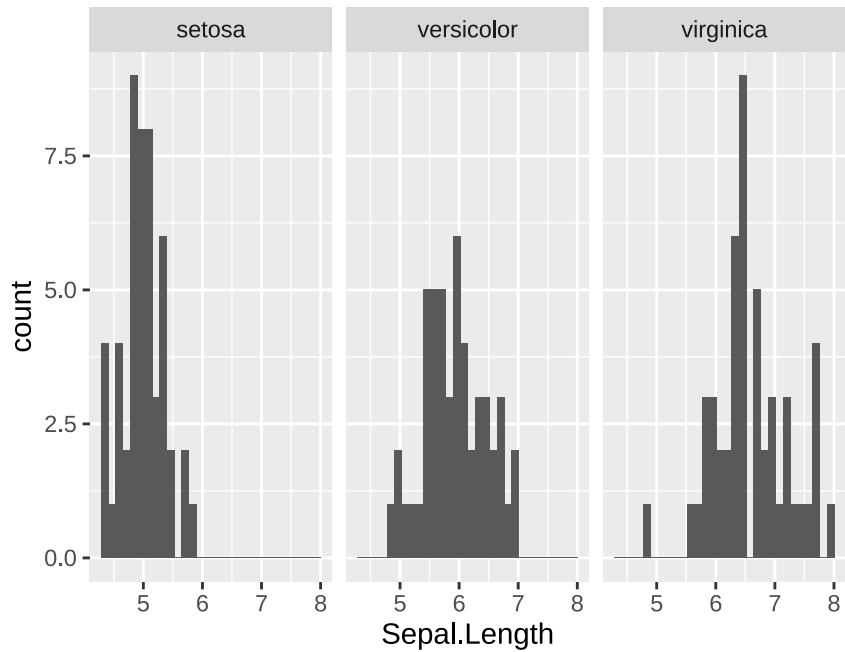


1140

1141 8.4.1 Multipanel Abbildungen

1142 Mit `ggplot2` kann man einfach Abbildungen erstellen, die mehrere Panels haben. Das bedeutet, dass eine
 1143 oder mehrere weitere Variablen gibt, die einen Plot in mehrere Subplots teilt. Dafür gibt es zwei Funktion:
 1144 `facet_grid()` und `facet_wrap()`.

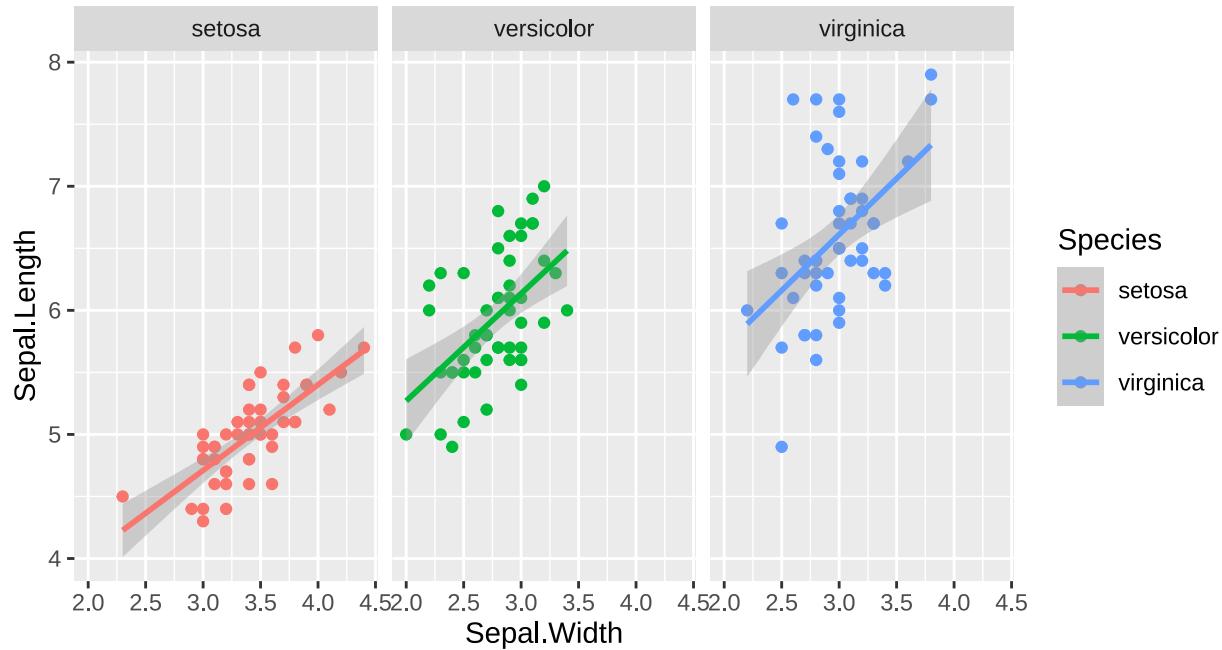
```
ggplot(iris, aes(Sepal.Length)) + geom_histogram() +
  facet_grid(~ Species)
```



1145

1146 Die Funktion `facet_grid()` erzeugt einen *Grid* und beschriftet die Grid-Zeilen und -Spalten, während
 1147 `facet_wrap()` für jedes Panel (Diagramm) eine eigene Überschrift erzeugt. Beim Arrangieren der Diagramme
 1148 wird der Vorteil von `ggplot2` gegenüber `base plot` besonders deutlich. Während es im `base plot` System
 1149 mühsam ist, die Abbildungen zu arrangieren, übernimmt `ggplot2` das Arrangement automatisch und fügt
 1150 sogar eine automatisch Legende hinzu. Die Achsenabschnitte werden so angepasst, dass die Daten vergleichbar
 1151 sind.

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

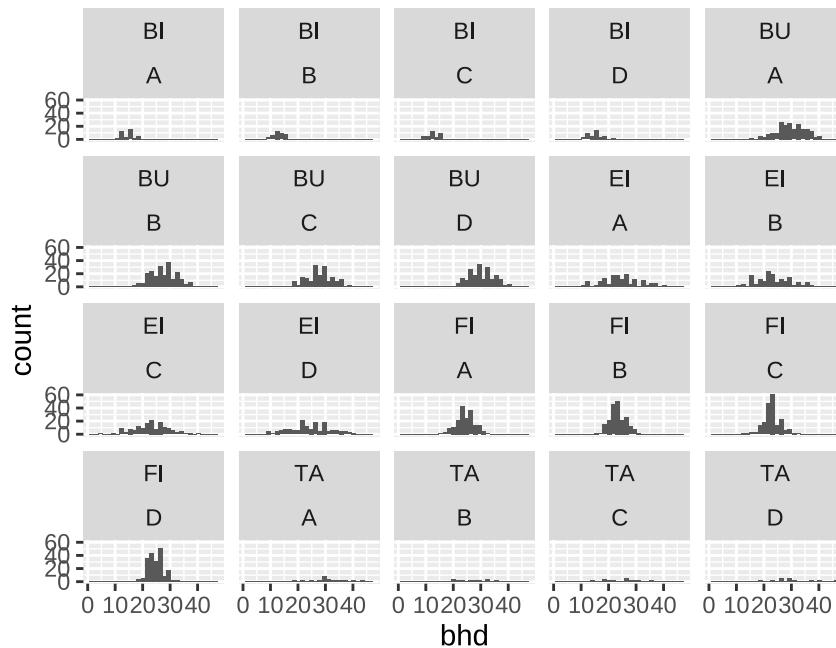


1152

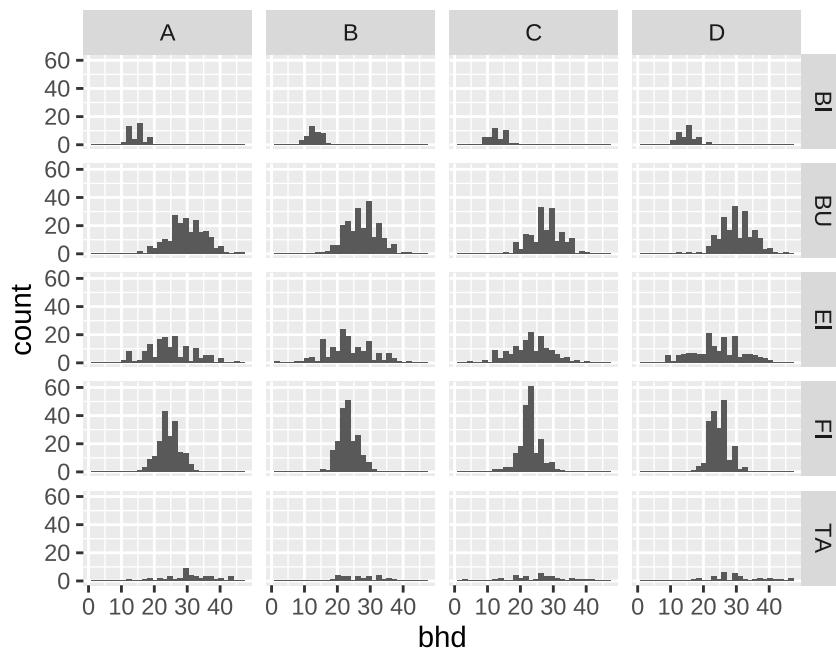
1153

1154 Aufgabe 22: Multipanel Abbildungen

1156 Lesen Sie erneut den Datensatz `daten/bhd_1.txt` ein und speichern Sie diesen in die Variable (`dat_bhd`).
 1157 Erstellen Sie für jede Art und Gebiet ein Histogramm. Welche Unterschiede können Sie feststellen, wenn Sie
 1158 `facet_grid()` oder `facet_wrap()` verwenden?



1159



1160

1161 8.4.2 Plots kombinieren

1162 Es gibt Situationen, in denen **unterschiedliche** Plots miteinander kombiniert werden müssen. Im vorherigen
 1163 Abschnitt wurde dies immer anhand einer gruppierenden Kovariate gemacht. Aber es gibt auch Situationen, in
 1164 denen das nicht möglich ist. Beispielsweise wenn ein Histogramm und ein Scatterplot vom gleichen Datensatz
 1165 zusammengefasst werden sollen. Dafür bietet sich das Paket **patchwork** an⁹.

1166 Als erstes können wir zwei (oder natürlich auch mehrere Plots) erstellen. Hier unterscheiden sich die Plots

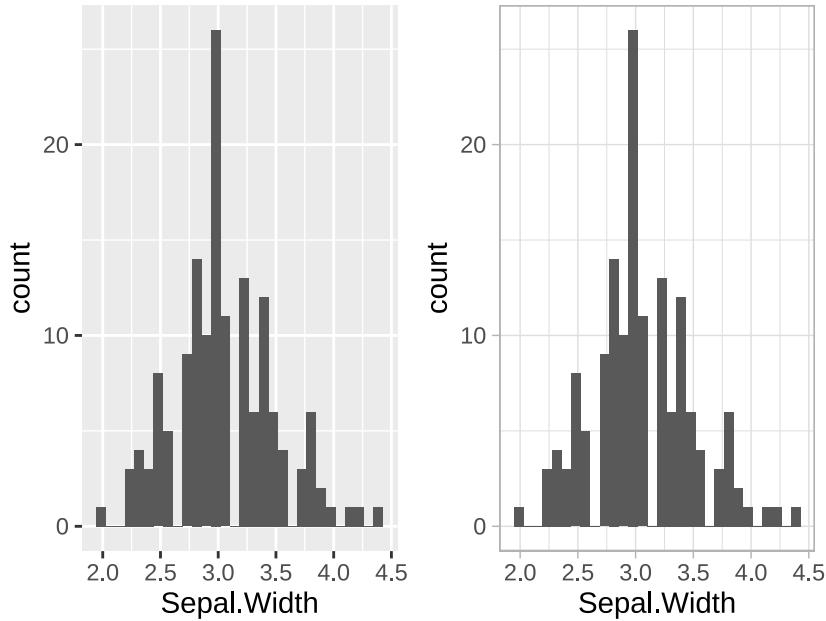
⁹Auch dieses Paket müssen Sie einmalig mit `install.packages("patchwork")` installieren.

1167 lediglich durch das Aussehen.

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

1168 Dann müssen können wir diese Plots ebenfalls mit `+` zusammenfügen.

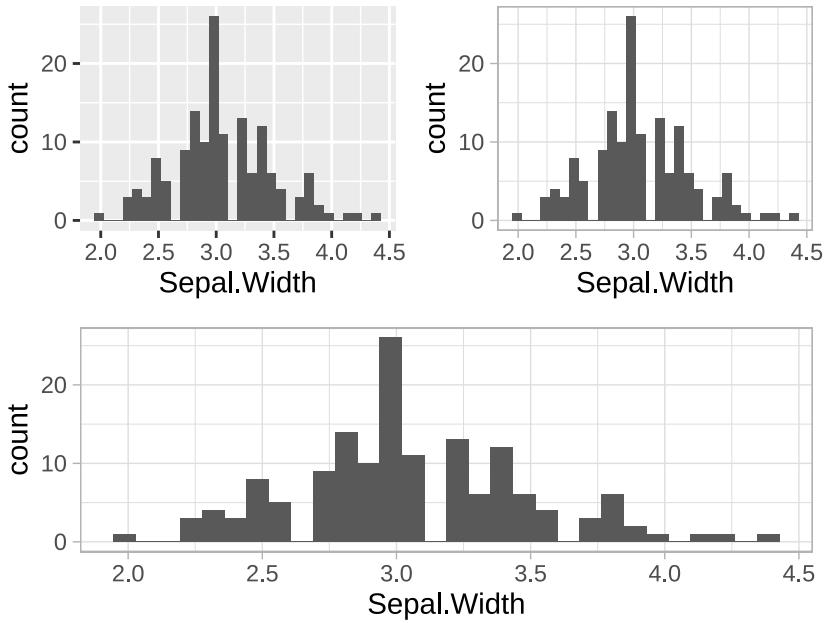
```
library(patchwork)
p1 + p2
```



1169

1170 Natürlich können auch weitere Plots hinzugefügt werden (auch in unterschiedlichen Dimensionen):

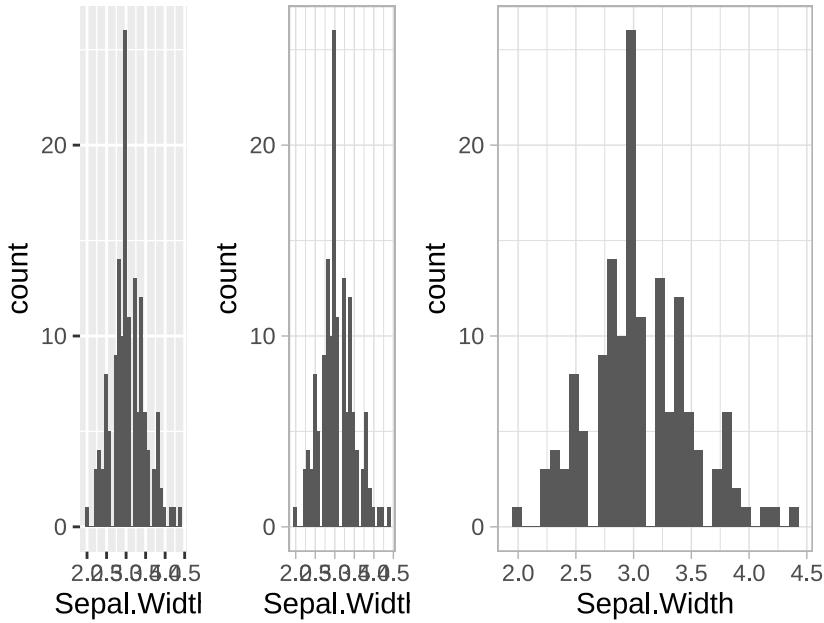
```
(p1 + p2) / p2
```



1171

¹¹⁷² Des weiteren können mit | auch Plots gegenüber gestellt werden.

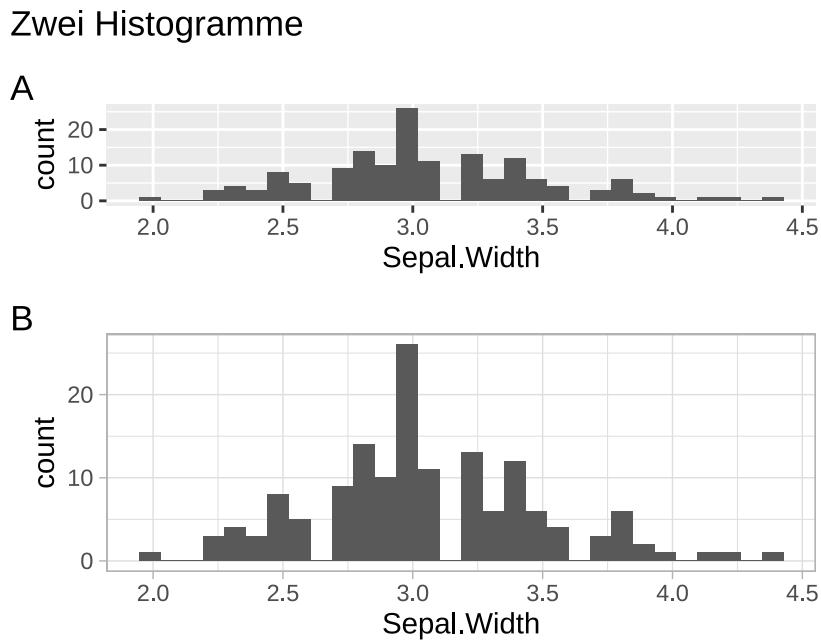
```
(p1 + p2) | p2
```



1173

¹¹⁷⁴ Weitere Optionen können mit `plot_layout()` und `plot_annotation()` angepasst werden. Mit ¹¹⁷⁵ `plot_layout()` kann die Anordnung der Plots bestimmt werden (z.B. über die Argument `nrow` und `ncol`), sowie deren *relative* Größe (über die Argumente `widths` und `heights`). Mit der Funktion ¹¹⁷⁶ `plot_annotation()` können zusätzliche Beschriftungen hinzugefügt werden, wie beispielsweise eine Titel ¹¹⁷⁷ (Argument `title`) oder ein Buchstabe/Zahl für jedes Element (Argument `tag_levels`). ¹¹⁷⁸

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

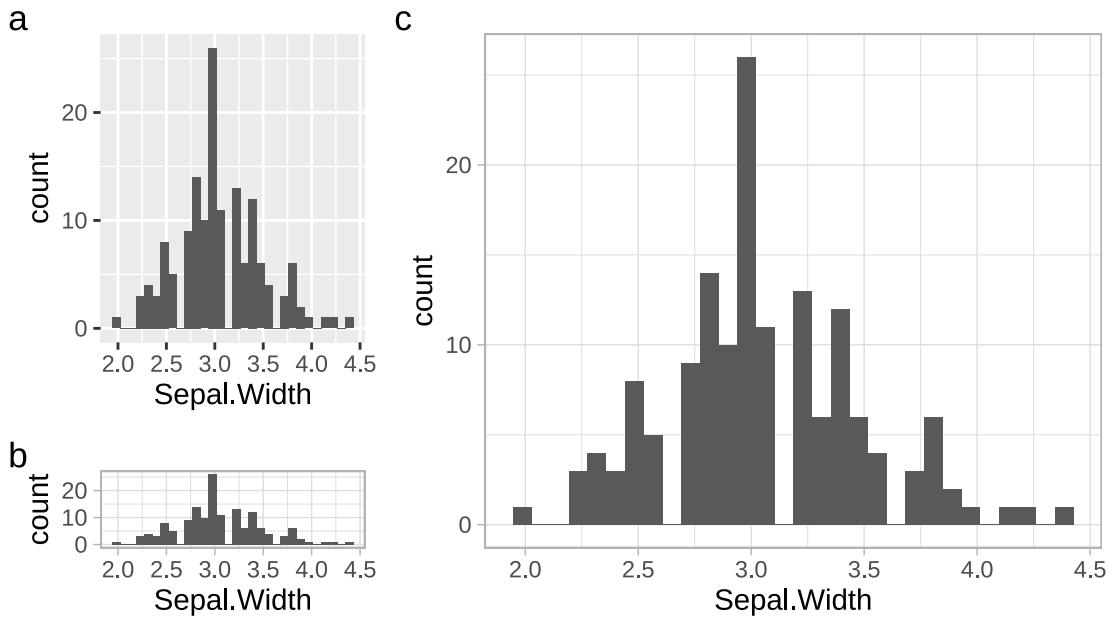


1179

1180

Aufgabe 23: Mehrere Plots zusammenfügen

- 1181
1182 Versuchen Sie die folgende Zusammenstellung der Plots nachzumachen:



1184

1185 **8.4.3 Speichern von plots**

1186 Sie können mit `ggsave()` eine zwischen-gespeicherte Abbildung exportieren, indem Sie den Variablenamen
1187 übergeben. Wenn Sie keine Variable übergeben, wird automatisch die letzte Abbildung gespeichert. Das
1188 Dateiformat wird aus dem Dateinamen übernommen. Die Größe der Inhalte können Sie sehr leicht mit den
1189 Argumenten `width` und `height` oder `scale` gesteuert werden.

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

1190 9 Mit Daten arbeiten

1191 9.1 dplyr eine Einführung

1192 `dplyr` ist eine Erweiterung von R (ein Paket), die das Ziel hat, den Umgang mit Daten einfacher und
1193 schneller zu machen.

1194 `dplyr` definiert 5 Verben, um mit Daten zu arbeiten. Diese sind:

- 1195 • `filter`
- 1196 • `select`
- 1197 • `arrange`
- 1198 • `mutate`
- 1199 • `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))
```

1200 Damit die Funktionen aus `dplyr` verwendet werden können, müssen wir als erstes das Paket `dplyr` laden.
1201 The online Resource [R for Data Science](#) gibt einen sehr guten, tieferen Eindruck in die Arbeit mit `dplyr`
1202 und erklärt auch die `dplyr` Philosophie noch Mal sehr genau für Beginner.

```
library(dplyr)
```

1203 Sollte dies zu einer Fehlermeldung führen, dann müssen Sie das Paket `dplyr` erst installieren. Dafür müssen
1204 Sie `einmalig install.packages("dplyr")` installieren.

1205 `dplyr` stellt unterschiedliche Funktionen zum Arbeiten mit Daten zur Verfügung. Es gibt fünf Grundfunktionen
1206 für die am häufigsten vorkommenden Operationen. Mit der Funktion `filter()` können unterschiedliche
1207 Beobachtungen gefiltert werden:

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

```
1208 ##   id plot bhd alter
1209 ## 1   1    1  50   10
1210 ## 2   2    1  29   30
1211 ## 3   3    2  13   31
1212 ## 4   4    2  23   24
1213 ## 5   5    3  25   25
```

1214 Es können auch mehrere Spalten verwendet werden.

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

```
1215 ##   id plot bhd alter
1216 ## 1   2    1  29   30
1217 ## 2   3    2  13   31
1218 ## 3   4    2  23   24
```

```
1219 ## 4 5 3 25 25
```

1220 Natürlich kann genau das gleiche Ergebnis mit dem ‘normalen’ R erreicht werden, dies wäre dann:

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

```
1221 ## id plot bhd alter
```

```
1222 ## 2 2 1 29 30
```

```
1223 ## 3 3 2 13 31
```

```
1224 ## 4 4 2 23 24
```

```
1225 ## 5 5 3 25 25
```

1226 Eine weitere Funktion aus dem Paket **dplyr** ist **select()**. Damit können Spalten aus einem **data.frame**

1227 ausgewählt werden. Dabei können auch die Spaltennamen unbenannt werden.

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

```
1228 ## bhd
```

```
1229 ## 1 50
```

```
1230 ## 2 29
```

```
1231 ## 3 13
```

```
1232 ## 4 23
```

```
1233 ## 5 25
```

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

```
1234 ## bhd id
```

```
1235 ## 1 50 1
```

```
1236 ## 2 29 2
```

```
1237 ## 3 13 3
```

```
1238 ## 4 23 4
```

```
1239 ## 5 25 5
```

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

```
1240 ## BHD id
```

```
1241 ## 1 50 1
```

```
1242 ## 2 29 2
```

```
1243 ## 3 13 3
```

```
1244 ## 4 23 4
```

```
1245 ## 5 25 5
```

1246 Mit der Funktion **arrange()** können die Beobachtungen in einem **data.frame** sortiert werden.

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

```
1247 ## id plot bhd alter
```

```
1248 ## 1 3 2 13 31
```

```
1249 ## 2 4 2 23 24
```

```
1250 ## 3 5 3 25 25
```

```
1251 ## 4 2 1 29 30
1252 ## 5 1 1 50 10
```

1253 Mit der Funktion `desc()` kann die Anordnung in absteigender Reihenfolge sortiert werden.

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

```
1254 ## id plot bhd alter
1255 ## 1 1 1 50 10
1256 ## 2 2 1 29 30
1257 ## 3 5 3 25 25
1258 ## 4 4 2 23 24
1259 ## 5 3 2 13 31
```

1260 Mit der Funktion `mutate()` kann man eine neue Spalte hinzufügen.

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

```
1261 ## id plot bhd alter bhd_mm fl
1262 ## 1 1 1 50 10 500 1963.4954
1263 ## 2 2 1 29 30 290 660.5199
1264 ## 3 3 2 13 31 130 132.7323
1265 ## 4 4 2 23 24 230 415.4756
1266 ## 5 5 3 25 25 250 490.8739
```

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

```
1267 ## id plot bhd alter mean_bhd
1268 ## 1 1 1 50 10 28
1269 ## 2 2 1 29 30 28
1270 ## 3 3 2 13 31 28
1271 ## 4 4 2 23 24 28
1272 ## 5 5 3 25 25 28
```

1273 Mit der Funktion `summarise()` können Spalten zusammengefasst werden.

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

```
1274 ## mean_bhd mean_sd
1275 ## 1 28 13.63818
```

1276

1277 **Aufgabe 24: Datenmanipulation mit dplyr**

- 1279 1. Laden Sie den Datensatz `data/bhd_1.txt`
- 1280 2. Berechnen Sie folgende Werte für alle Einträge und speichern Sie die Ergebnisse in `erg1`
- 1281 • mittlerer `bhd`
- 1282 • maximales `alter`
- 1283 • die Standardabweichung des BHDs
- 1284 • die Anzahl Bäume mit einem BHD > 30

1285 **9.2 Arbeiten mit gruppierten Daten**

1286 Zusätzlich können `mutate` und `summarise` auch auf gruppierte Daten angewendet werden. Dafür müssen
 1287 wir erst die Funktion `group_by()` aufrufen und als Argumente die Variablen übergeben, die die Gruppen
 1288 definieren.

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

1289 ##   id plot bhd alter bhd_m
1290 ## 1 1    1 50    10    28
1291 ## 2 2    1 29    30    28
1292 ## 3 3    2 13    31    28
1293 ## 4 4    2 23    24    28
1294 ## 5 5    3 25    25    28

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

1295 ## # A tibble: 5 x 5
1296 ## # Groups:   plot [3]
1297 ##       id   plot   bhd   alter   bhd_m
1298 ##     <int> <dbl> <dbl> <dbl> <dbl>
1299 ## 1     1     1    50     10    39.5
1300 ## 2     2     1    29     30    39.5
1301 ## 3     3     2    13     31    18
1302 ## 4     4     2    23     24    18
1303 ## 5     5     3    25     25    25

summarise(dat, bhd_m = mean(bhd))

1304 ##   bhd_m
1305 ## 1    28

summarise(dat1, bhd_m = mean(bhd))

1306 ## # A tibble: 3 x 2
1307 ##   plot   bhd_m
```

```
1308 ## <dbl> <dbl>
1309 ## 1      1  39.5
1310 ## 2      2  18
1311 ## 3      3  25
```

1312

1313 **Aufgabe 25: dplyr mit gruppierten Daten**

- 1315 1. Laden Sie den Datensatz `data/bhd_1.txt`
- 1316 2. Berechnen Sie für jede Baumart und Standort die folgenden Werte:
- 1317 • mittlerer `bhd`
 - 1318 • maximales `alter`
 - 1319 • die Standardabweichung des BHDs
 - 1320 • die Anzahl Bäume mit einem BHD > 30
- 1321 3. Ordnen Sie das Ergebnis nach Baumart und BHD.

1322 **9.3 pipes oder %>%**

1323 Mit *Pipes* (`%>%`) kann man das Ergebnis einer Funktion einfach an eine nachfolgende Funktion weiterreichen.

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

1324 Wir kennen bis jetzt:

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

```
## [1] 3.333333
```

1326 Mit *Pipes*, die durch das Symbol `%>%` dargestellt werden¹⁰, können wir das etwas vereinfachen und nacheinander schreiben:

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

```
## [1] 3.333333
```

1329 Oder sogar

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

```
## [1] 3.333333
```

1331

1332 **Aufgabe 26: Pipes %>%**

1334 Wiederholen Sie die letzte Aufgabe, aber diesmal ohne Zwischenergebnisse zu speichern:

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

- 1335 1. Laden Sie den Datensatz `data/bhd_1.txt`.
- 1336 2. Berechnen Sie für jede Baumart und Standort die folgenden Werte:
- 1337 • mittlerer `bhd`
- 1338 • maximales `alter`
- 1339 • die Standardabweichung des BHDs
- 1340 • die Anzahl Bäume mit einem BHD > 30
- 1341 3. Ordnen Sie das Ergebnis nach Baumart und BHD.

1342 9.4 Joins

1343 Eine weitere häufige Aufgabe beim Daten Management ist es Daten zusammenzuführen. Nehmen Sie an, dass
1344 wir folgende Aufnahmen gemacht haben

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

1345 und jeder Baum lediglich mit einer `id` versehen wurde. In einer zweiten Tabelle wurden dann weitere Daten
1346 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")
)
```

1347 Ziel ist es jetzt die Bäume aus `aufnahmen` mit den Informationen aus den `metadaten` zu verbinden. Dazu
1348 dient `id` als Bindeglied (oft auch Schlüssel genannt).

1349 Dazu gibt es vier Möglichkeiten.

1350 Zur Durchführung gibt es in base R die Funktion `merge()`. Wir werden aber gleich die Funktionen aus dem
1351 Paket `dplyr` verwenden.

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

1352 ##   id bhd  art gebiet
1353 ## 1   1   20 <NA>   <NA>
1354 ## 2   2   31    Ta      A
1355 ## 3   3   74    Bu      B

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

1356 ##   id bhd art gebiet
1357 ## 1   2   31   Ta      A
1358 ## 2   3   74   Bu      B
1359 ## 3   4   NA   Bu      B
```

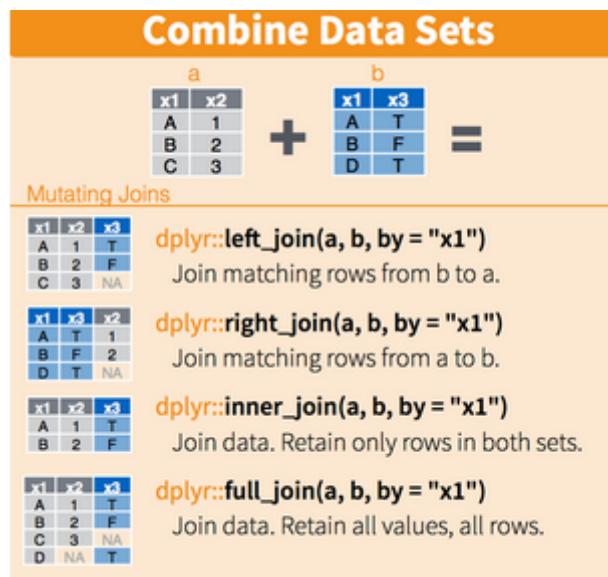


Abbildung 8: Joins (Quelle Rstudio)

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

```
1360 ##   id bhd art gebiet
1361 ## 1 2 31 Ta     A
1362 ## 2 3 74 Bu     B
full_join(aufnahmen, metadaten, by = "id")
```

```
1363 ##   id bhd art gebiet
1364 ## 1 1 20 <NA> <NA>
1365 ## 2 2 31 Ta     A
1366 ## 3 3 74 Bu     B
1367 ## 4 4 NA Bu     B
```

1368 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"))
```

```
1369 ##   id bhd art gebiet
1370 ## 1 1 20 <NA> <NA>
1371 ## 2 2 31 Ta     A
1372 ## 3 3 74 Bu     B
```

1373

1374 **Aufgabe 27: Verbinden von Daten**

- 1375
- Lesen Sie die Datensätze `daten/bhd_2.txt` und `daten/bhd_2_meta.txt` ein.
 - Stellen Sie sicher, dass es für den `bhd` keine fehlenden Werte gibt (entfernen sie entsprechende Zeilen)
 - Fügen Sie zu den Metadaten (gespeichert in `bhd_2_meta`) die Anzahl Bäume und den mittleren `bhd` hinzu pro Gebiet.

1380 **9.5 ‘long’ and ‘wide’ Datenformate**
1381 Unter anderem Wickham (2014) empfieilt das Prinzip von *tidy Data*. Nach diesem Prinzip sollten Daten wie
1382 folgt organisiert sein:

- 1383
- Jede Zeile ist ein Merkmalsträger/Subjekt/Objekt (z. B. eine Person oder ein Baum).
 - Jede Spalte ist eine Variable (ein Merkmal), die den Merkmalsträger beschreibt.
 - In jeder Zelle ist genau ein Wert (Merkmalausprägung), nämlich der Wert, der Variable für den Merkmalsträger.

1387 Zum Beispiel enthalten Spaltennamen oft Informationen, die eigentlich in einer Variable gespeichert werden
1388 sollten. Folgendes Beispiel gibt die BHD Messung von 3 Bäumen in 3 Jahren wieder. Der Vorteil von *tidy*
1389 Daten ist, dass viele Funktionen diese Form erwarten. Somit müssen sie Ihre Daten nur ein Mal organisieren
1390 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)
)
```

1391 Der `tibble` aus dem Paket `tibble` ist ein moderner Data Frame, der sich in seinen Funktionen in das `tidy`
1392 Universum einfügt. Wir können auch einfach das Paket `tidyverse` laden, um alle Pakete, die Teil des `tidy`
1393 Universums sind gleichzeitig zu laden. Der `tibble` beinhaltet alle Funktionen, die der Data Frame auch
1394 beinhaltet und kann auch für genau die gleichen Zwecke verwendet werden. Der `tibble` hat u. A. eine
1395 modernere Darstellung im Konsolenoutput.

1396 Diese Daten sind jetzt im `wide`-Format gespeichert und folglich nicht `tidy`, weil Information über die Daten
1397 (nämlich das Jahr der Aufnahme in den Spaltennamen gespeichert sind). Besser wäre eine Struktur mit
1398 nur drei Spalten: `id`, `jahr` und `bhd`. Um die Daten in so eine Struktur zu bringen, gibt es die Funktion
1399 `pivot_longer()` aus dem Paket `tidyR`, die ebenfalls Teil des `tidyverse` ist.

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

## # A tibble: 9 x 3
##       id   name   value
##   <dbl> <chr>  <dbl>
## 1     1   bhd2015 30
## 2     1   bhd2026 31
## 3     1   bhd2017 34
## 4     2   bhd2015 31
## 5     2   bhd2026 31
## 6     2   bhd2017 32
## 7     3   bhd2015 32
## 8     3   bhd2026 33
## 9     3   bhd2017 33
```

```

1402 ##   <int> <chr>   <dbl>
1403 ## 1     1 bhd2015    30
1404 ## 2     1 bhd2026    31
1405 ## 3     1 bhd2017    34
1406 ## 4     2 bhd2015    31
1407 ## 5     2 bhd2026    31
1408 ## 6     2 bhd2017    32
1409 ## 7     3 bhd2015    32
1410 ## 8     3 bhd2026    33
1411 ## 9     3 bhd2017    33

```

1412 Wenn wir die Spalten für die Variable und den Wert gleich sinnvoll benennen möchten, können wir das über
1413 die Argumente `names_to` und `values_to` machen.

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

```

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

```

1426 Analog zu der Funktion `pivot_longer()` gibt es auch die Funktion `pivot_wider()`, um vom Daten vom
1427 `long`-Format ins `wide`-Format zu transformieren.

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

```

1428 ## # A tibble: 3 x 4
1429 ##       id bhd2015 bhd2026 bhd2017
1430 ##   <int>   <dbl>   <dbl>   <dbl>
1431 ## 1     1      30      31      34
1432 ## 2     2      31      31      32
1433 ## 3     3      32      33      33

```

1434

1435 **Aufgabe 28: Zeitliche Verlauf von BHDs**

1437 In der Datei `bhd_3.csv` befinden sich gemessene BHDs (in cm) von unterschiedlichen Bäumen zu unter-
 1438 schiedlichen Zeitpunkten. Erstellen Sie ein Liniendiagramm, das den zeitlichen (x-Achse) Verlauf der BHDs
 1439 (y-Achse) für die unterschiedlichen Bäume darstellt.

1440 **9.6 Auswählen von Variablen**

1441 Sobald die Datensätze etwas umfangreicher werden (d. h., es gibt mehrere Spalten in einem `data.frame`),
 1442 können innerhalb vieler `dplyr`-Funktionen spezielle Funktionen verwendet werden, um Variablen auszuwählen.
 1443 Wenn die genaue Position der Spalten bekannt ist, kann man mit dem `:`-Operator und der Position Spalten
 1444 auswählen:

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

1445 ## Sepal.Length Sepal.Width Petal.Length
1446 ## 1          5.1      3.5      1.4
1447 ## 2          4.9      3.0      1.4
1448 ## 3          4.7      3.2      1.3
```

1449 Diese Vorgehensweise kann gehährlich sein, da sich manchmal Spalten verschieben und sich somit die
 1450 Positionen ändern. Es besser Spalten immer explizit mit Namen statt mit Nummern anzusprechen.

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

1451 ## Sepal.Length Sepal.Width Petal.Length
1452 ## 1          5.1      3.5      1.4
1453 ## 2          4.9      3.0      1.4
1454 ## 3          4.7      3.2      1.3
```

1455 `select()` erlaubt es, auch hier den `:`-Operator zu verwenden:

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

1456 ## Sepal.Length Sepal.Width Petal.Length
1457 ## 1          5.1      3.5      1.4
1458 ## 2          4.9      3.0      1.4
1459 ## 3          4.7      3.2      1.3
```

1460 Es gibt auch einige spezielle Funktionen, um Spalten innerhalb eines `select()`-Aufrufs auszuführen:

- 1461 • `starts_with()`: Hier kann man ein Muster angeben, mit dem ein Text anfangen muss.
- 1462 • `ends_with()`: Diese Funktion ist analog zu `starts_with()`, jetzt wird aber am Ende des Spaltennamens
 1463 nach dem Muster gesucht.
- 1464 • `contains()`: Hier kann ein Muster übergeben werden, das irgendwo im Spaltennamen sein muss.
- 1465 • `everything()`: Mit dieser Funktion werden alle Spalten ausgewählt.

1466 • `last_col()`: Mit dieser Funktion wird nur die letzte Spalte ausgewählt (dass ist die Spalte, die ganz
1467 rechts ist).

1468 Sämtliche Auswahlen können mit - umgekehrt werden.

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

1469 ## Sepal.Length Sepal.Width

1470 ## 1 5.1 3.5

1471 ## 2 4.9 3.0

1472 ## 3 4.7 3.2

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

1473 ## Petal.Length Petal.Width Species

1474 ## 1 1.4 0.2 setosa

1475 ## 2 1.4 0.2 setosa

1476 ## 3 1.3 0.2 setosa

1477 `select()` bietet auch noch die Möglichkeit, Spaltennamen zu ändern.

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

1478 ## sep_width

1479 ## 1 3.5

1480 ## 2 3.0

1481 ## 3 3.2

1482

1483 Aufgabe 29: Auswählen von Spalten

1485 In der Datei `messungen_1.csv` sind Messungen von zwei Sensoren enthalten für die ersten vier Monate eines
1486 Jahres. Führen Sie folgende Abfragen durch:

1487 1. Wählen Sie alle Messungen für Januar aus.

1488 2. Wählen Sie alle Messungen für Januar und März aus.

1489 9.7 Einzelne Beobachtungen abfragen (`slice()`)

1490 Mit dem Befehl `slice()` kann man einzelne Beobachtungen (= Zeilen) aus einem `data.frame` abfragen.

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

1491 ## Sepal.Length Sepal.Width Petal.Length Petal.Width Species

1492 ## 1 5.1 3.5 1.4 0.2 setosa

1493 ## 2 4.4 2.9 1.4 0.2 setosa

1494 ## 3 5.1 3.5 1.4 0.3 setosa

1495 Davon gibt es drei nützliche Varianten: 1) `slice_head()` und `slice_tail()`; 2) `slice_max()` und
 1496 `slice_min()`; 3) `slice_random()`.

1497 `slice_head()` und `slice_tail()` sind analog zu `head()` und `tail()`, aber mit dem entscheidenden Unter-
 1498 schied, dass Gruppierungen berücksichtigt werden. Wenn keine Gruppierung in den Daten vorhanden ist, gibt
 1499 es keinen Unterschied.

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

```
1500 ##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
1501 ## 1       5.1      3.5      1.4      0.2 setosa
1502 ## 2       4.9      3.0      1.4      0.2 setosa
```

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

```
1503 ##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
1504 ## 1       5.1      3.5      1.4      0.2 setosa
1505 ## 2       4.9      3.0      1.4      0.2 setosa
```

1506 Sobald jedoch eine gruppierende Variable eingeführt wird, gibt `slice_head()` die ersten n Beobachtungen
 1507 für jede Gruppe zurück und `head()` für den gesamten Datensatz.

```
# base head
```

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

```
1508 ## # A tibble: 2 x 5
1509 ## # Groups:   Species [1]
1510 ##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
1511 ##       <dbl>     <dbl>     <dbl>     <dbl> <fct>
1512 ## 1       5.1      3.5      1.4      0.2 setosa
1513 ## 2       4.9      3       1.4      0.2 setosa
```

```
# dplyr slice_head
```

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

```
1514 ## # A tibble: 6 x 5
1515 ## # Groups:   Species [3]
1516 ##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
1517 ##       <dbl>     <dbl>     <dbl>     <dbl> <fct>
1518 ## 1       5.1      3.5      1.4      0.2 setosa
1519 ## 2       4.9      3       1.4      0.2 setosa
1520 ## 3       7       3.2      4.7      1.4 versicolor
1521 ## 4       6.4      3.2      4.5      1.5 versicolor
1522 ## 5       6.3      3.3       6      2.5 virginica
1523 ## 6       5.8      2.7      5.1      1.9 virginica
```

1524 `slice_tail()` funktioniert analog zu `slice_head()` mit dem einzigen Unterschied, dass nicht die ersten n

1525 Zeilen zurück gegeben werden sondern die letzten `n` Zeilen.
 1526 `slice_max()` und `slice_min()` geben die Beobachtung mit dem maximalen bzw. minimalen Wert einer
 1527 Variable zurück. Auch hier werden Gruppen berücksichtigt.

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

1528 ## Sepal.Length Sepal.Width Petal.Length Petal.Width Species
 1529 ## 1 7.9 3.8 6.4 2 virginica

1530 Und mit Gruppen:

```
iris %>% group_by(Species) %>%  

  slice_max(Sepal.Length)
```

1531 ## # A tibble: 3 x 5
 1532 ## # Groups: Species [3]
 1533 ## Sepal.Length Sepal.Width Petal.Length Petal.Width Species
 1534 ## <dbl> <dbl> <dbl> <dbl> <fct>
 1535 ## 1 5.8 4 1.2 0.2 setosa
 1536 ## 2 7 3.2 4.7 1.4 versicolor
 1537 ## 3 7.9 3.8 6.4 2 virginica

1538 `slice_min()` funktioniert genau gleich, nur dass die Beobachtung (=Zeile) mit dem kleinsten Wert einer
 1539 Variable zurück gegeben wird.

1540 Die Funktion `slice_sample()` erlaubt es zufällige Beobachtungen zu ziehen. Dabei kann über das Argument `n`
 1541 die Anzahl an Beobachtungen angegeben werden oder über das Argument `prop` der Anteil an Beobachtungen.

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

1542 ## Sepal.Length Sepal.Width Petal.Length Petal.Width Species
 1543 ## 1 6.5 2.8 4.6 1.5 versicolor
 1544 ## 2 6.3 3.3 4.7 1.6 versicolor
 1545 ## 3 7.2 3.2 6.0 1.8 virginica
 1546 ## 4 4.9 3.6 1.4 0.1 setosa
 1547 ## 5 6.0 2.7 5.1 1.6 versicolor

1548 Das Ergebnis ist bei jedem von Ihnen anders, da es sich um eine zufällige Ziehung handelt. Wenn Sie diese
 1549 Ergebnisse wiederholen möchten, können Sie über `set.seed()` die zufällige Ziehung reproduzierbar machen.

```
set.seed(123)  

slice_sample(iris, n = 5)
```

1550 ## Sepal.Length Sepal.Width Petal.Length Petal.Width Species
 1551 ## 1 4.3 3.0 1.1 0.1 setosa
 1552 ## 2 5.0 3.3 1.4 0.2 setosa
 1553 ## 3 7.7 3.8 6.7 2.2 virginica
 1554 ## 4 4.4 3.2 1.3 0.2 setosa
 1555 ## 5 5.9 3.0 5.1 1.8 virginica

1556 Wenn beispielsweise 5% der Beobachtungen gezogen werden sollen, kann dies so gemacht werden:

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

```
1557 ##   Sepal.Length Sepal.Width Petal.Length Petal.Width   Species
1558 ## 1         7.7       3.8        6.7       2.2  virginica
1559 ## 2         5.5       2.5        4.0       1.3 versicolor
1560 ## 3         5.5       2.6        4.4       1.2 versicolor
1561 ## 4         6.5       3.0        5.2       2.0  virginica
1562 ## 5         6.1       3.0        4.6       1.4 versicolor
1563 ## 6         6.3       3.4        5.6       2.4  virginica
1564 ## 7         5.1       2.5        3.0       1.1 versicolor

1565 slice_sample() berücksichtigt ebenfalls Gruppen. Mit den Argumenten replace und weight_by dann die
1566 Zufallsziehung genauer spezifiziert werden. replace sagt, ob eine gezogenen Beobachtung wieder zurück gelegt
1567 wird oder nicht. Mit dem Argument weight_by können optional gewichtete für jede Beobachtung vergeben
1568 werden.
```

1569

1570 Aufgabe 30: Daten beschreiben

1572 Verwenden Sie den Datensatz `bhd_1.txt` und finden Sie für jedes Gebiet und Art die Beobachtung mit
1573 kleinsten BHD.

1574 9.8 Spalten trennen

1575 Ein gut geplanter Datensatz besteht aus Beobachtungen (Zeilen), Variablen (Spalten) und in jeder Zelle ist
1576 immer ein **genau** ein Wert gespeichert. Leider gibt es oft Datensätze, bei denen mehr als ein Wert pro Zelle
1577 gespeichert wurde. Die Funktion `separate()` kann helfen solche Daten zu trennen.

1578 Wir verwenden einen erfunden Datensatz zu Beobachtungen von Tieren und einer geschätzten Distanz zu
1579 diesen Tieren.

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

1580 In der Spalte `beobachtung` sind zwei Informationen gespeichert: Die Distanz zur Beobachtung und die Art.
1581 Das ist ungünstig, weil wir so weder nach Tierart noch nach distanz filtern können (nicht *tidy*). Mit der
1582 Funktion `separate()`, können wir Beobachtungen einer Spalte in mehrere Spalten trennen. Dafür muss der
1583 Spaltennamen (Argument `col`), die neuen Spaltennamen (Argument `into`) und das Trennzeichen (Argument
1584 `sep`) angegeben werden.

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

1585 ## # A tibble: 4 x 3

```
1586 ##      id Distanz Art
1587 ##    <int> <chr>   <chr>
1588 ## 1     1 10m     " Reh"
1589 ## 2     2 100m    " Reh"
1590 ## 3     3 20m     " Fuchs"
1591 ## 4     4 40      "Reh"

1592 Nach dem Aufruf von separate() gibt es zwei neue Spalten (Distanz und Art), die die alte Spalte
1593 beobachtung ersetzen.
```

1594

1595 Aufgabe 31: Aufräumen 1596

1597 Verwenden Sie den folgenden Datensatz und bringen Sie ihn in eine Form, die sicherstellt dass

1598 • jede Zelle genau einen Wert enthält.
1599 • jede Zeile eine Beobachtung ist.
1600 • 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")
)
```

1601 10 Arbeiten mit Text

1602 Bis jetzt haben wir fast ausschließlich mit Zahlen oder Abbildungen gearbeitet. R bietet aber auch viele
 1603 Werkzeuge, um mit Text zu arbeiten. Wir wollen hier ein paar Funktionen dafür vorstellen. Als erstes sollte
 1604 nochmals klar gestellt werden, was eigentlich ein Text ist. In R ist alles, das innerhalb von doppelten ("") oder
 1605 einfachen ('') Anführungszeichen geschrieben ist, Text.

1606 Anbei einige Beispiele:

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

1607 Wichtig ist hier zu sehen, dass z nicht als Zahl sondern, als Text interpretiert wird (siehe Datentypen).

```
z + 1
```

1608 ## Error in z + 1: non-numeric argument to binary operator

1609 Wenn man sicher ist, dass es sich bei einem Textobjekt um eine Zahl handelt, kann man dies mit der Funktion
 1610 `as.numeric()` in eine Zahl umwandeln.

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

1611 ## [1] 31

1612 Aber mit a führt dies wieder zu einem NA-Wert, da a nicht in eine Zahl umgewandelt werden kann.

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

1613 ## Warning: NAs introduced by coercion

1614 ## [1] NA

1615 10.1 Arbeiten mit Text

1616 Wir wollen erst einmal drei Funktionen besprechen, die es erlauben mit Text zu arbeiten. Die Funktion
 1617 `nchar()`¹¹ gibt an wie viele Zeichen ein Text hat. Also z.B.

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

1618 ## [1] 5

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

1619 ## [1] 2

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

1620 ## [1] 20

1621 Die Funktion `paste()` erlaubt es verschiedene Variablen mit Text zu verbinden. Wenn wir z. B. die Variablen
 1622 `vorname <- "Eva"` und `name <- "Meier"` haben und wir wollen eine neue Variable `full_name <- Eva`

¹¹char ist kurz für character.

1623 "Meier" erzeugen, dann kann das mit der Funktion `paste()` gemacht werden.

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

1624 ## [1] "Eva Meier"

1625 Die Funktion `paste()` hat das Argument `sep`, das auf ein Leerzeichen () gesetzt ist, aber auch anders sein
1626 kann und das Trennzeichen definiert.

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

1627 ## [1] "Eva, Meier"

1628 Die Funktion `substr()` erlaubt es am Anfang oder Ende eines Wortes etwas abzuschneiden. Dabei muss
1629 immer die Anfangs- (`start`) und Endposition (`stop`) angegeben werden.

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

1630 ## [1] "Hal"

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

1631 ## [1] "allo"

1632

1633 Aufgabe 32: Arbeiten mit Text 1

1635 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")
```

1636 1. Aus wie vielen Buchstaben besteht jedes Wort?

1637 2. Finden Sie das längste Wort.

1638 3. Wie viel Prozent der Wörter fangen mit einem S an?

1639 4. Fügen Sie jedem Wort seine Position im Vektor hinzu. Beispielsweise soll aus `Vogel` "2. Vogel" werden
1640 usw.

1641 10.2 Finden von Textmustern

1642 Mit der Funktion `grep()` können Muster in einem Text gefunden werden. Wenn wir beispielsweise folgenden
1643 Vektor mit Textelementen haben.

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

1644 Und wir wollen alle Straßennamen die ein `weg` haben abfragen, dann können wir folgenden Befehl ausführen:

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

1645 ## [1] 2

1646 Im zweiten Element von `txt` kommen die Zeichen `Weg` vor. Beachte, in der Standardeinstellung wird zwischen
1647 Groß- und Kleinschreibung unterschieden. Dies kann mit dem Argument `ignore.case = TRUE` angepasst
1648 werden.

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

1649 ## [1] 1 2

1650 Mit der Funktion `sub` können Zeichen innerhalb einer Zeichenkette ersetzt werden.

1651 So ersetzt der folgende Ausdruck `ae` mit `ä`.

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

1652 ## [1] "Friedländer Weg"

1653 Wenn allerdings das zu ersetzende Zeichen mehr als einmal vorkommt und beide Instanzen ersetzt werden
1654 sollen braucht man die Funktion `gsub`.

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

1655 ## [1] "Friedländer Weg und Reinhäuser Landstrasse sind zwei Strassen in Goettingen."

1656 Mit `sub()` wird nur das erste `ae` ersetzt, während `gsub()` alle `ae` mit einem `ä` ersetzt.

```
gsub("ae", "ä", txt)
```

1657 ## [1] "Friedländer Weg und Reinhäuser Landstrasse sind zwei Strassen in Goettingen."

1658 Oft ist der genaue Ausdruck den man finden möchte jedoch Variabel. Beispielsweise möchte man alle Wörter
1659 mit einem Umlaut oder Zahlen finden möchte, kann man das oft abkürzen. Dafür gibt es reguläre Ausdrücke.
1660 Wir werden hier nur ein paar beispielhafte Anwendungen besprechen.

1661 Sowohl in den Funktionen `grep()` als auch `(g)sub()` kann mit anstatt dem Muster (immer das erste
1662 Argument) aus ein regulärer Ausdruck angegeben werden. Mit `[1-9]` sind alle Zahlen von 1 bis 9 gemeint.

1663 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)
```

1664 ## [1] 1 3

1665 Damit lässt sich auch das Problem mit groß- und kleingeschriebenen Wörtern lösen.

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

```
1666 ## [1] 1 2
```

1667

Aufgabe 33: Arbeiten mit Text 2

1670 Verwenden Sie den folgenden Vektor:

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

1671 1. In wie vielen Wörtern kommt der Doppellaut au vor?

1672 2. Ersetzen Sie in allen Wörtern alle au mit _ _.

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

```
1673 ## [1] 5 6 13
```

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

```
1674 ## [1] "Versicherung" "Methoden"      "Fluss"        "Rudel"        "B_ _m"
```

```
1675 ## [6] "H_ _s"          "Foto"         "Auffahrt"     "Auto"        "Handy"
```

```
1676 ## [11] "Teller"        "Kalender"     "Aufb_ _"
```

1677 11 Arbeiten mit Zeit

1678 Für den Computer bzw. R ist ein Datum erst einmal nichts anderes als ein Text. Für uns ist es sofort klar,
 1679 dass der "13.2.2021" der 13. Februar 2021 ist, für den Computer nicht. Wir müssen R also irgendwie sagen,
 1680 dass die 13 der Tag ist, die 2 der Monat und 2021 das Jahr. Dass der Computer die einzelnen Komponenten
 1681 erkennt, und in einen Datentyp eigens für Zeitformate überführt, nennt man *parsen*¹². Das Arbeiten mit
 1682 Datum und Zeit kann anfangs sehr mühsam sein, aber sobald man einige Grundfertigkeiten erworben
 1683 hat, kann man viele Aufgaben deutlich schneller und effizienter erledigen. Starten Sie am besten gleich mit
 1684 "echten" Zeit-Datentypen und versuchen Sie nicht, sich irgendwie mit Text und numerischen Datentypen
 1685 selbst etwas zu basteln. Der erste Schritt ist immer ein Datum zu *parsen*. Wir verwenden dafür Funktionen
 1686 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)
```

1687 **lubridate** bietet eine Vielzahl von Funktionen zum *parsen* von Datum und Zeit, die sich aus:

- 1688 • y für Jahr,
- 1689 • m für Monat,
- 1690 • d für Tag,
- 1691 • h für Stunde,
- 1692 • m für Minute und
- 1693 • s für Sekunde

1694 zusammen setzten. Alle Funktionen nehmen als erstes Argument ein Textstring. Wenn wir z.B. den String
 1695 2020-01-20 parsen wollen können wir das mit der Funktion **ymd** machen.

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

1696 ## [1] "2020-01-20"

1697 Dabei erkennt **lubridate** in der Regel die Trennzeichen:

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

1698 ## [1] "2020-01-20"

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

1699 ## [1] "2020-01-20"

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

1700 ## [1] "2020-01-20"

1701 Wenn die die Anordnung der einzelnen Komponenten anders ist, gibt es einfach eine andere Funktion.

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

¹²to parse heißt zergliedern bzw. grammatisch bestimmen.

1702 ## [1] "2020-01-20"
 1703 Jetzt stellt sich die Frage, was der Vorteil ist, wenn R ein Datum parst.

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

1704 Wir können jetzt mit d arbeiten und einzelne Komponenten extrahieren.
`day(d)`

1705 ## [1] 20
`month(d)`

1706 ## [1] 1
`year(d)`

1707 ## [1] 2020
 1708 Oder auch Zeiteinheiten hinzufügen oder abziehen.

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

1709 ## [1] "2020-01-30"
`d - years(20)`

1710 ## [1] "2000-01-20"
`d + hours(25)`

1711 ## [1] "2020-01-21 01:00:00 UTC"

1712

Aufgabe 34: Arbeiten mit Datum und Zeit

- 1713 • Parsen Sie folgende Zeitangaben 23.1.2020, 13.2.1992 20:55:23, Mar/3/97 und 10.7.2020 19:15
 1714 und speichern Sie diese in einen Vektor d.
 1715 • Extrahieren Sie nun aus jedem Element aus d das Jahr und die Stunde.
 1716 • Fügen zu jedem Element in d 10 Tage hinzu.

1719 **11.1 Arbeiten mit Zeitintervallen**

1720 Mit zwei Zeitpunkten lassen sich Zeitintervalle (`Periods`) erstellen, dafür können wir die Funktion `interval()`
 1721 aus dem Paket `lubridate` verwenden¹³.

```
anfang <- ymd("2020-03-18")  

ende <- anfang + years(1)
```

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

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

1722 Wir können jetzt mit `int` arbeiten und beispielsweise das Intervall verschieben,

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

1723 ## [1] 2023-03-18 UTC--2024-03-18 UTC

1724 die Länge des Intervalls berechnen

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

1725 ## [1] 31536000

1726 oder testen, ob ein Datum innerhalb des Intervalls liegt.

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

1727 ## [1] TRUE

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

1728 ## [1] FALSE

1729 `%within%` funktioniert genauso mit Vektoren oder mit mehreren Intervallen. Wir könnten also zwei Intervalle
1730 definieren (z.B. Ostern und Pfingsten).

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

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

1731 Und Überprüfen welche Termine in eines der zwei Intervalle fallen.

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

```
# Ostern
```

```
termine %within% ostern
```

1732 ## [1] FALSE TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE

```
# Pfingsten
```

```
termine %within% pfingsten
```

1733 ## [1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE TRUE FALSE FALSE

1734 Der Zeit-Datentyp wird auch oft benutzt, um den Zeitbedarf von längeren Berechnungen zu messen.

```
t1 <- now()
```

```
mean(runif(1e7))
```

1735 ## [1] 0.4999484

```
t2 <- now()
```

```
int_length(interval(t1, t2))
```

1736 `## [1] 0.6105618`

1737 11.2 Formatieren von Zeit

1738 Für die Ausgabe in Berichten oder Grafiken soll das Datum oft in einer speziellen Form dargestellt werden.

1739 Die Funktion `format()` bietet Möglichkeiten ein Datumsobjekt zurück in Text umzuwandeln.

1740 Ein Beispiel wäre `ymd("2021-2-10")` als 10.2.21 auszugeben.

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

1741 `## [1] "21.02.21"`

1742 Dabei handelt sich bei `%d.%m.%y` um Abkürzungen für die unterschiedlichen Komponenten eines Datumobjekts.

1743 Siehe dazu die Hilfeseite von `strptime` (`help(strptime)`).

1744

1745 Aufgabe 35: Arbeiten mit Intervallen

1746

1747 Wie viele Einträge aus dem Vektor `v1` befinden sich in einem Intervall, das zwischen dem 1.3.2021 und dem 1748 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"
)
```

1749 11.3 Zeitreihen

1750 In der Forstbranche haben wir oft mit Zeitreihen zu tun. Zeitreihen sind Variablen, die in zeitlichen Intervallen 1751 vorliegen. In der Regel monatlich, vierteljährig oder jährlich, wobei die Intervalle zwischen den Messungen 1752 immer gleich lang sind. Wiederholungsmessungen von Forstinventuren (Forsteinrichtungen, Betriebsinventuren, 1753 die meisten forstl. Experimente, die BWI, ...) sind also in der Regel keine Zeitreihen in engeren Sinne, 1754 turnusmäßig vermessene Versuchsflächen, wie sie z. B. die Versuchsanstalten unterhalten oder jährlich 1755 gemeldete Holzpreise jedoch schon.

1756 Statistisch gesehen haben Zeitreihen Besonderheiten gegenüber anderen Variablen, da Sie in erster Linie von 1757 Ihrer eigenen Vergangenheit abhängen (autokorriert sind) und auch die Abhängigkeit anderer Variablen in 1758 der Regel nur sinnvoll ist, wenn deren Vergangenheit mitberücksichtigt wird (cross-correlation). Konventionelle 1759 Statistik ist oft nicht ausreichend, um Zeitreihen zu analysieren. Angefangen mit der Datendarstellung gibt 1760 es spezifische Zeitreihen-Funktionen, welche auch alle in R integriert sind. Aus diesem Grund sollten Sie 1761 Zeitreihen als solche speichern. Viele Funktionen erkennen den Datentyp und führen entsprechend spezifische 1762 Operationen durch. Laden wir z. B. die Holzpreise für Fichte 2b (das sog. Leitsortiment), das Holzaufkommen 1763 dieses Sortiments und die Preise für Nadelholz vom statistischen Bundesamt¹⁴. Wir laden die Daten

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

1764 zunächst als csv:

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

1765 Mit der Funktion `ts` werden die Daten in ein Zeitreihenobjekt überführt (geparst). Die Spalte mit den
1766 Jahren ist dann nicht mehr nötig, da die Informationen zur Zeit keine eigene Variable mehr sind, sondern zu
1767 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

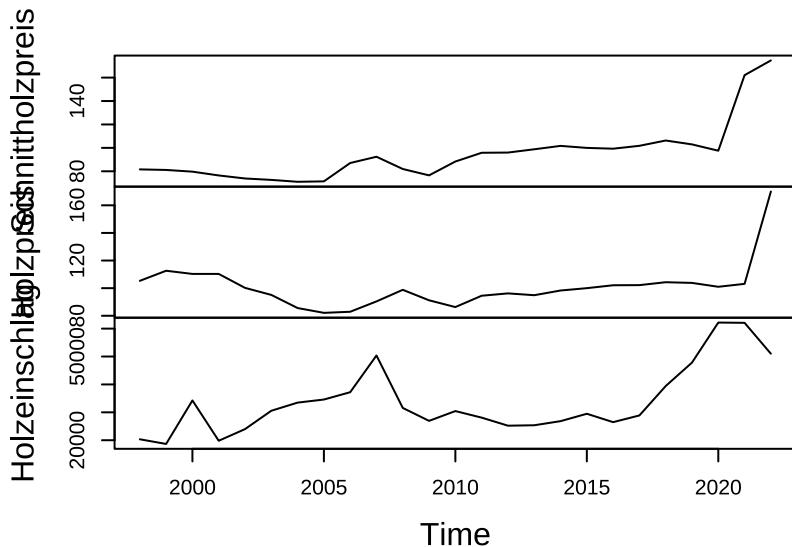
1768 ## [1] "double"

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

1769 Die wichtigsten Argumente sind - `data` Vektor oder Matrix, der nur die Daten enthält - `start` Startzeitpunkt -
1770 `frequency` Anzahl der Beobachtungen pro Zeiteinheit, also z. B. 12 bei monatlichen oder 4 bei quartalsweisen
1771 Erhebungen

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

zr

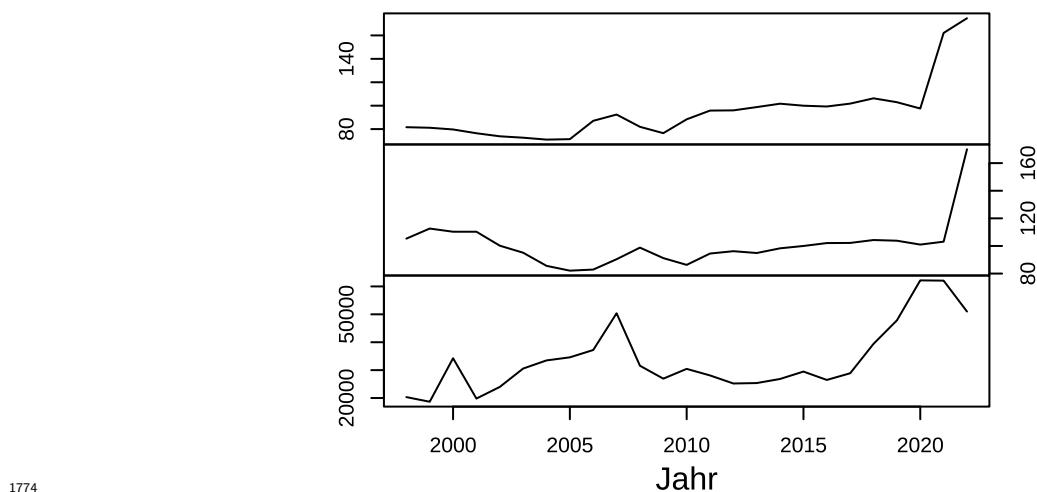


1772

1773 Wir können low-level Layer hinzufügen. Die meisten base plot Funktionen funktionieren auch für Zeitreihen

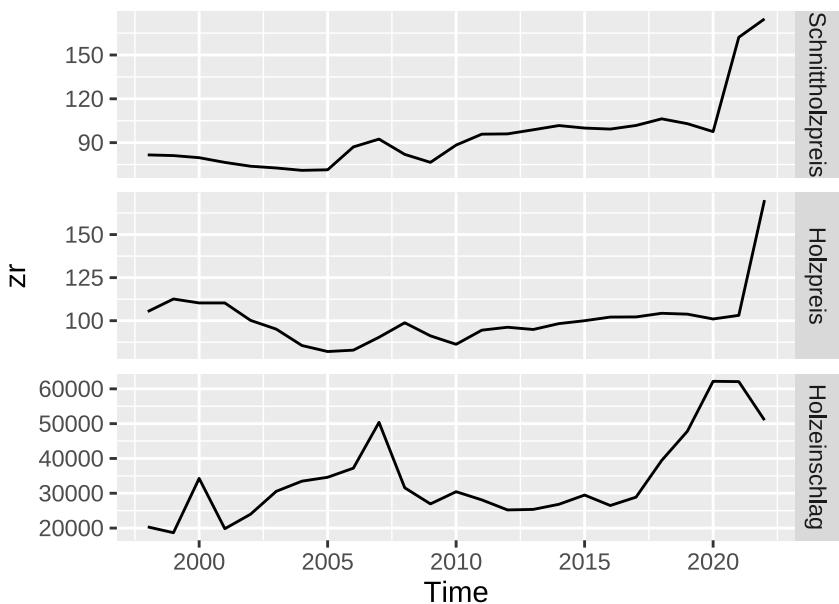
```
plot(zr, ann = FALSE, yax.flip = TRUE)
mtext(side = 3, cex = 1.5, font = 2, "Holzmarktentwicklung seit 1998", line = 2)
mtext(side = 1, cex = 1, line = 2.5, "Jahr")
```

Holzmarktentwicklung seit 1998



1775 Das Paket `forecast` ermöglicht automatisierte Zeitreihenplots im `ggplot2` Stil. Damit ist auch das Problem
1776 der y-Achsenbeschriftungen gelöst.

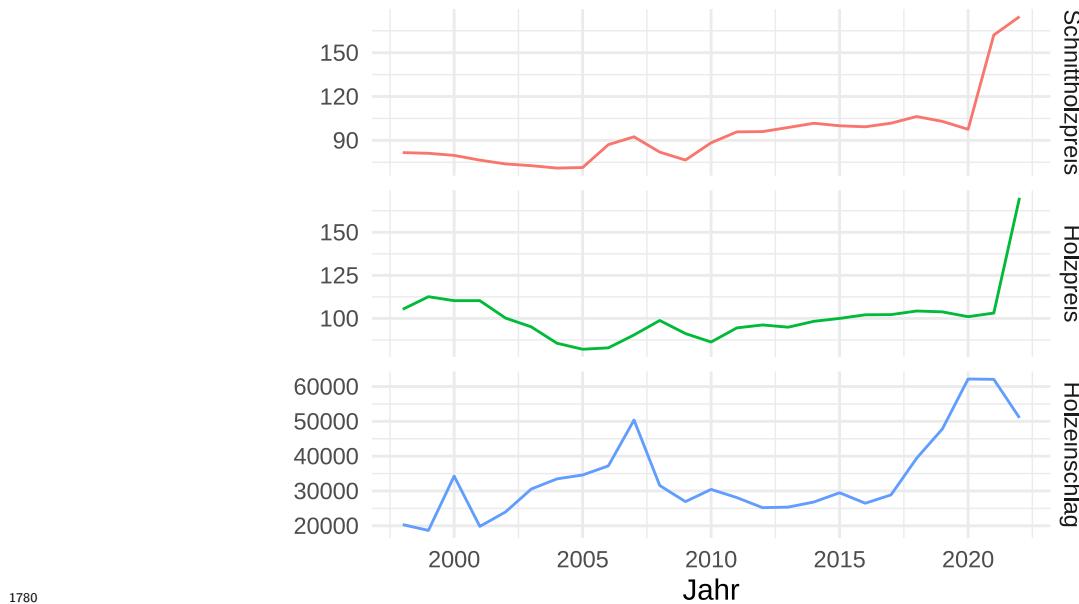
```
autoplot(zr, facets = TRUE)
```



1778 Wir können die Abbildung im `ggplot2` Stil ändern. Hier nur ein Minimalbeispiel zur Veranschaulichung.
1779 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

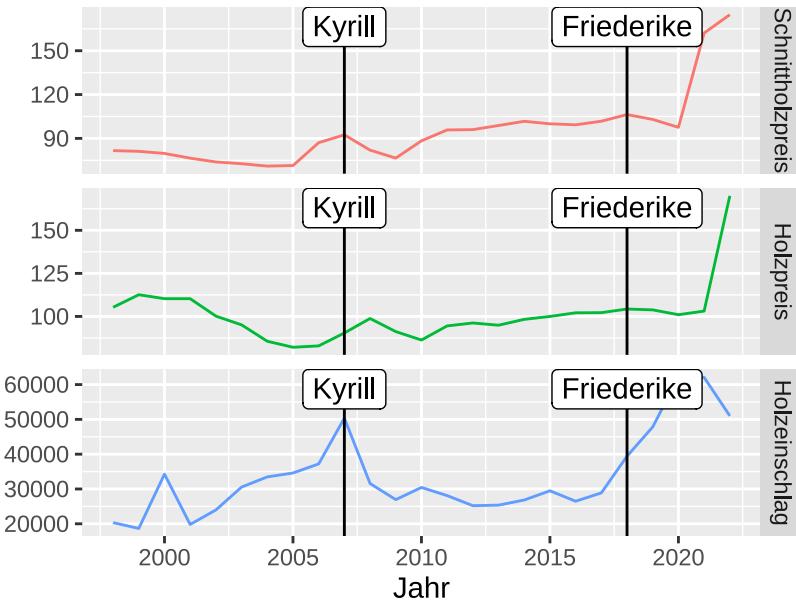
zr_autoplot + theme_minimal()
```



1780

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

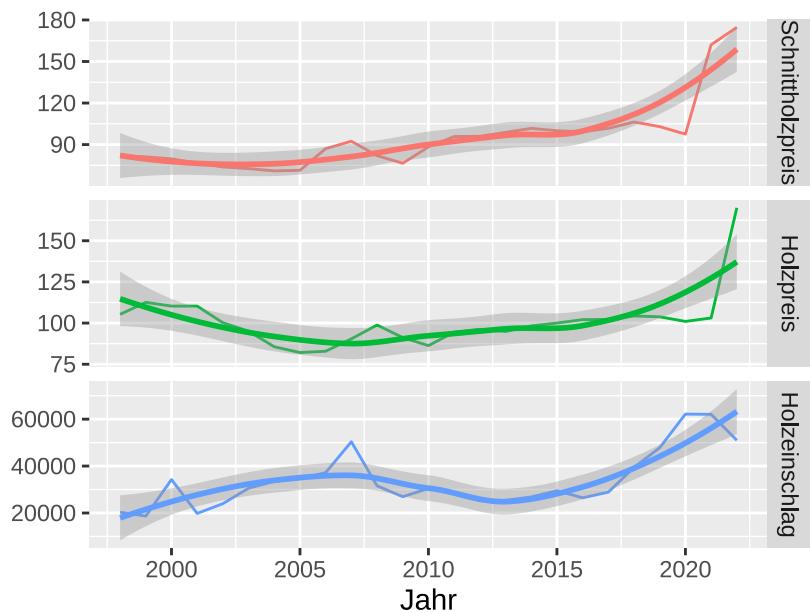
z2 + annotate(x = 2007, y = +Inf, label = "Kyrill", vjust = 1, geom = "label") +
  annotate(x = 2018, y = +Inf, label = "Friederike", vjust = 1, geom = "label")
```



1781

- 1782 Eine Trendlinie macht hier (sowie generell in Zeitreihendaten) offensichtlich keinen Sinn. Daher verwenden wir
 1783 den sog. Loess-Glättter, um einen Trend in die Daten zu legen. Der Loess-Glättter ist ein sehr flexible Kurve.

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



1784

1785 12 Aufgaben Wiederholen (for-Schleifen)

1786 Um einfache Programme zu schreiben, müssen Sie den Ablauf der Programmcodes kontrollieren können.
 1787 Kontrollieren bedeutet in diesem Zusammenhang, dass Codeabschnitte nur unter definierten Bedingungen ab-
 1788 laufen. Sie programmieren also zwei Sachen. 1) den Code selbst und 2), die Bedingungen die erfüllt sein müssen,
 1789 damit der Code ausgeführt wird. Der Code muss so generisch geschrieben sein, dass er komplett durchläuft,
 1790 auch wenn unterschiedliche Dateneingaben gemacht werden. Diese Kontrollbedingungen ermöglichen es Ihnen
 1791 generisch zu programmieren. Sie schreiben Ihren Code also nicht speziell maßgeschneidert für ein Problem,
 1792 sondern so generell, dass er für mehrere Auswertungen funktioniert. Um dies zu gewährleisten, müssen Sie
 1793 bestimmte Situationen vorhersehen und abfangen. Hierbei helfen Ihnen Kontrollstrukturen (**Control Flow**).
 1794 Grundsätzlich gibt es **Control Flow** Funktionen zur Wiederholung von Codeblöcken (Schleifen) und logische
 1795 Bedingungen (bedingte Anweisung).

1796 12.1 Schleifen

1797 Bis jetzt wurden alle Skripte einfach der Reihe nach abgearbeitet und zwischendurch bestimmte Programmteile,
 1798 je nach Situation, selbstständig ausgeführt oder übersprungen. Mit einer Schleife kann man erreichen, dass
 1799 eine Gruppe von Befehlen (der sog. Schleifenrumpf) mehrfach abgearbeitet wird, zum Beispiel wenn bestimmte
 1800 Auswertungsschritte auf mehrere Datensätze oder Variablen angewendet oder Funktionen mit unterschiedlichen
 1801 Parametern oder Startwerten aufgerufen werden sollen. Weitere Anwendungsmöglichkeiten sind iterative
 1802 Algorithmen, in denen die Eingabewerte des aktuellen Iterationsschrittes von einem vorherigen abhängig sind.
 1803 Besonders in Simulationen kommen Schleifen häufig zum Einsatz, da große Anzahlen von Wiederholungen
 1804 benötigt werden.

1805 Man unterscheidet zwischen zwei Arten von Schleifen: Bei den **for()**-Schleifen steht die Anzahl der Wieder-
 1806 holungen schon beim Eintritt in die Schleife fest, während die **while()**-Schleifen so lange ausgeführt werden,
 1807 bis eine Bedingung nicht mehr wahr ist. Mit der Funktion **break** wird eine Schleife abgebrochen und die
 1808 Programmausführung wird nach der Schleife fortgesetzt.

1809 Die wesentlichen Befehle sind

1810 • **for (i in X) {Code}**

1811 Wiederhole den Code im “Schleifenrumpf” für jedes Element aus X.

1812 • **while(Bedingung) {Code}**

1813 Wiederhole den Code im “Schleifenrumpf” so lange die logische Bedingung erfüllt ist.

1814 • **break()**

1815 Brich die Schleife ab. **break()** muss im Schleifenrumpf an der richtigen Stelle ausgeführt werden. Gute
 1816 Praxis ist jedoch, die for oder while Bedingungen, dass kein **break()**nötig ist, da **break()** anfällig für
 1817 Programmierfehler ist.

1818 12.1.1 Wiederholen von Befehlen mit **for()**.

1819 Steht vor Beginn der Schleife fest wie viele Schleifendurchgänge benötigt werden, wenn zum Beispiel in einer
 1820 Simulationen 99 Realisierungen erzeugt oder alle Elemente eines Vektors verarbeitet werden sollen, verwendet

1821 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)
}
```

1822 ## [1] 1

1823 ## [1] 2

1824 ## [1] 3

1825 Das **i** steht in diesem Beispiel für die Schleifen-Variable. Sie muss nicht **i** heißen, sondern kann jeden
 1826 zulässigen Namen annehmen. Das **X** steht für einen existierenden Vektor oder eine existierende Liste bzw.
 1827 einen Ausdruck, der ein solches Objekt liefert (der Objektname ist ebenfalls frei wählbar). **for** und **in** sind
 1828 Schlüsselworte, sie müssen, ebenso wie die runden Klammern, vorhanden sein.

1829 Im ersten Durchgang erhält die Schleifen-Variable **i** den ersten Wert von **X** und der Schleifenrumpf wird
 1830 mit diesem Wert ausgeführt. Die Variable **i** nimmt nacheinander so lange die Werte von **X** an, bis ihr alle
 1831 Elemente zugewiesen wurden.

1832 Das folgende Beispiel wird zwar besser durch die entsprechende Vektoroperation gelöst, zeigt aber sehr
 1833 deutlich die Arbeitsweise der **for**-Schleife.

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

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

1834 ## [1] 4

1835 ## [1] 9

1836 ## [1] 25

1837

1838 Aufgabe 36: Schleifen 1

1840 Verwenden Sie den Vektor **k** <- c(1, 3, 9, 12, 15) und schreiben Sie folgende **for**-Schleifen:

- 1841 1. Eine Schleife, die jedes Element aus **k** ausgibt.
- 1842 2. Eine Schleife, die zu jedem Element aus **k** 10 addiert und den neuen Wert ausgibt.
- 1843 3. Eine Schleife wie in 2), aber der neue Wert (**k** + 10) soll jetzt nicht mehr ausgegeben werden, sondern
 1844 in **k10** gespeichert werden. Stellen Sie sicher, dass **k10** wieder von der Länge 5 ist.

1845

1846 Die Funktion **for()** ermöglicht es, einen Befehl beliebig oft zu wiederholen. Z.B. der folgende Ausdruck zieht
 1847 10-Mal eine Stichprobe der Größe 1 aus dem Vektor **v**. Beachten Sie, dass die Schleifen-Variable **i** selbst gar

1848 nicht im Schleifenrumpf vorkommt. Das Ziel dieser Schleife ist nicht die Elemente des Vektors abzuarbeiten,
 1849 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))
}
```

```
1850 ## [1] 3
1851 ## [1] 1
1852 ## [1] 3
1853 ## [1] 3
1854 ## [1] 2
1855 ## [1] 3
1856 ## [1] 2
1857 ## [1] 2
1858 ## [1] 1
1859 ## [1] 4
```

1860 Auf gleiche Weise kann man auch über die Variablen eines Dataframes iterieren¹⁵. Das folgende Beispiel hat
 1861 zum Ziel die Funktionsweise von Schleifen zu verdeutlichen. Schleifen haben in R jedoch den Nachteil, dass sie
 1862 sehr langsam operieren. Wenn es geht, sollte man Alternativen verwenden. Die Funktionsweise wiederholender
 1863 Auswertungen wird jedoch mit `for`-Schleifen deutlicher. Aus diesem Grunde werden wir uns in diesem Kurs
 1864 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)
}
```

```
1865 ## [1] "Buche"
1866 ## [1] 52
1867 ## [1] "Eiche"
1868 ## [1] 64
1869 ## [1] "Eiche"
1870 ## [1] 62
1871 ## [1] "Buche"
1872 ## [1] 85
```

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

1873

1874 **Aufgabe 37: for-Schleife**

18751876 Lesen Sie den Datensatz `bhd_1.txt` ein und verwenden Sie eine `for`-Schleife.

- 1877 • Ziehen Sie 500-Mal je 35 Beobachtungen für den BHD.
1878 • Berechnen Sie jeweils den Mittelwert aus den 35 Werten.
1879 • Speichern Sie die 500 Mittelwerte in einem neuen Vektor `mittel`.
1880 • Wie ist die Verteilung dieser 500 Mittelwerte zu interpretieren?

1881 **12.1.2 Wiederholen von Befehlen mit `while()`**

1882 Die `while`-Schleifen finden Anwendung, wenn die Anzahl der zu durchlaufenden Wiederholungen vorher
1883 nicht bekannt ist, wie zum Beispiel bei Iterationsverfahren, die bis zu einer gewissen Genauigkeit durchlaufen
1884 werden sollen. Die `while`-Schleife besteht in R aus dem Schlüsselwort `while()` und einer Bedingung in runden
1885 Klammern.

```
while (Bedingung) {  
  # Schleifenrumpf  
}
```

1886 Sie ist in der praktischen Programmierung nicht so relevant wie die `for`-Schleife. Sie sei deshalb hier nur
1887 kurz erwähnt. Die Abbruchbedingung wird jedes Mal geprüft bevor der Schleifenrumpf durchlaufen wird. Die
1888 Bedingung wird ausgewertet und wenn diese `TRUE` ist, wird der Schleifenrumpf ausgeführt und danach erneut
1889 die Bedingung überprüft. Ist die Bedingung nicht erfüllt, wird der Schleifenrumpf nicht durchgeführt und die
1890 Schleife beendet. Ist die Bedingung bereits vor Eintritt in die Schleife nicht erfüllt, wird die Schleife gar nicht
1891 erst durchlaufen.

1892 Da `while`-Schleifen also so lange ausgeführt werden, bis die Bedingung nicht mehr erfüllt ist, kann eine
1893 Endlosschleife entstehen. Dies kann passieren, wenn man nicht sauber programmiert hat. Wenn innerhalb der
1894 Schleife nicht dafür gesorgt wird, dass die Bedingung irgendwann nicht mehr erfüllt wird, so läuft die Schleife
1895 immer weiter. Steckt R in einer Schleife fest und reagiert nicht mehr, kann der Befehl unter Linux mit `Strg`+
1896 `C` und unter Windows mit `Esc` abgebrochen werden. Alternativ können Sie auf das rote STOP Symbol
1897 über der Konsole klicken.

1898 **12.2 Bedingte Ausführung von Codeblöcken**

1899 Innerhalb eines Skripts ist es mitunter notwendig je nach aktueller Situation unterschiedlich fortzufahren.
1900 Die Situation wird mit einem logischen Ausdruck, einer sogenannten Bedingung, geprüft. Je nachdem, ob
1901 die Bedingung wahr (`TRUE`) oder falsch (`FALSE`) ist, werden unterschiedliche Programmteile ausgeführt, der
1902 jeweils andere Teil bleibt unberücksichtigt. Danach wird in jedem Fall die Programmausführung, mit den
1903 auf die bedingte Anweisungen folgenden Anweisung, fortgesetzt. In R kann dies mit dem if-else-Konstrukt
1904 realisiert werden, welches aus den Schlüsselwörtern `if()` und `else` sowie der in runde Klammern gefassten
1905 Bedingung besteht.

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

1906 Im folgenden Beispiel sollen die bisher abstrakt beschriebenen Vorgänge praktische Anwendung finden. In
 1907 dem Beispiel wird zunächst durch zufälliges Ziehen einer Zahl aus dem Bereich eins bis sechs ein Würfelwurf
 1908 simuliert. Anschließend wird der Würfelwurf mit einem if-Ausdruck kommentiert. Ist die Bedingung, es wurde
 1909 eine Sechs gewürfelt, erfüllt, wird der Code innerhalb der geschweiften Klammern ausgeführt, ansonsten wird
 1910 der Klammerinhalt ignoriert.

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

1911 In den meisten Fällen ist es für R irrelevant, ob sich zwischen den verschiedenen Elementen Leerzeichen oder
 1912 Zeilenumbrüche befinden. Bei dem else-Ausdruck dagegen wird ein Fehler erzeugt, wenn vor dem `else` nicht
 1913 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.")
}
```

1914 ## [1] "Beim nächsten Wurf klappt's bestimmt."

1915

1916 Aufgabe 38: Bedingte Programmierung

- 1918 • Wenn keine 6 gewürfelt wurde, lassen Sie zusätzlich ausgeben welche Zahl gewürfelt wurde.
 1919 • Wiederholen Sie den Würfelwurf 10 Mal.

1920 13 (R)markdown

1921 13.1 Markdown Grundlagen

1922 Die Idee von Markdown ist, einfach Text strukturieren zu können und das ganze ohne umfangreiche Programme
 1923 zu erstellen. Im nächsten Abschnitt sehen wir dann, wie man Markdown und R-Code verbinden kann. Hier
 1924 soll es jedoch erst einmal darum gehen, die einfachsten Bausteine von Markdown vorzustellen.

1925 Am Anfang jedes Dokuments kommt eine Präambel. Diese fängt mit --- an und hört auch wieder mit ---
 1926 auf. Innerhalb der Präambel können dann Metainformationen über das Dokument festgelegt werden, dies
 1927 beinhaltet im einfachsten Fall: Titel, Autor und Datum. Das würde dann so aussehen:

```
1928 ---
1929 title: "Ein Titel"
1930 author: "Der, der es geschrieben hat"
1931 date: "März 2021"
1932 ---
```

1933 Danach folgt strukturierter und formatierter Text. Verschiedene Hierarchieebenen von Überschriften können
 1934 mit der Anzahl an # festgelegt werden. So ist eine Überschrift erster Ordnung # Kapitel eine Überschrift
 1935 zweiter Ordnung ## Unterkapitel usw.

1936 Listen können erstellt werden, wenn man am Anfang jeder Zeile ein - oder 1. schreibt.

```
1937 - Erster Eintrag
1938 - Zweiter Eintrag
1939 - Dritter Eintrag
```

1940 wird zu

```
1941   • Erster Eintrag
1942   • Zweiter Eintrag
1943   • Dritter Eintrag
```

1944 Eine zentrale Idee von Markdown ist es Text einfach zu formatieren. Werden eine oder mehrere Wörter mit
 1945 zwei Sternchen (**) eingefasst wird dieser Text **fett** dargestellt. Also aus **wichtig** wird **wichtig**. Das
 1946 gleiche funktioniert auch mit *kursiven* Text, jedoch muss man hier noch ein Sternchen verwenden, also aus
 1947 *kursiv* wird *kursiv*. Soll ein text fett und kursiv sein, kann man drei Sternchen verwenden. Aus ***sehr
 1948 wichtig*** wird dann **sehr wichtig**.

1949 Weitere Elemente wie Links oder Abbildungen können einfach eingebunden werden. Links werden mit [Link
 1950 text](url) in den Text integriert. Beispielsweise ist eine gute Idee bei [stackoverflow](#) bei Problemen nach
 1951 einer Lösung zu suchen. Dieser Link wurde mit [stackoverflow](www.stackoverflow.com) erstellt.

1952 Für Abbildungen gibt es einen ganz ähnlichen Syntax. Mit ! [Das R Logo](abb/r_logo.png) wird die
 1953 Abbildung r_logo.png eingebunden mit der Beschriftung: "Das R Logo".



Abbildung 9: Das R Logo

1954

Aufgabe 39: Arbeiten mit markdown

1957 Verwenden Sie das folgende Markdowndokument:

```

1958 ---
1959 title: "Dokument"
1960 author: "Ihr Name"
1961 date: "März 2021"
1962 ---
1963
1964 # Einleitung
1965
1966 # Methoden
1967 1. Kopieren Sie die Vorlage in ein Dokument, das test.md heißt.
1968 2. Fügen Sie zwei Überschriften zweiter und dritter Ordnung hinzu.
1969 3. Fügen Sie einen kursiven Text hinzu.
1970 4. Fügen Sie einen Link zu einer Website hinzu.
1971 5. Kompilieren Sie die Datei, indem Sie in Rstudio auf Preview drücken (Abbildung 10).

```

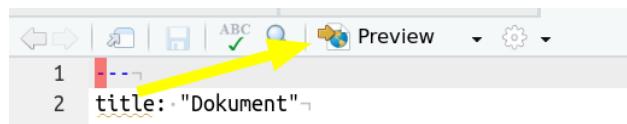


Abbildung 10: Kompilieren einer md-Datei.

13.2 R und Markdown

1972 Markdown macht es bereits einfach Textdokumente und Dokumentationen zu verfassen, aber die wirkliche Stärke liegt in der Möglichkeit R und Markdown zu kombinieren. Man spricht dann von Rmarkdown. Ein weiteres Strukturelement, das wir noch nicht kennen gelernt haben, sind Code-Blöcke.

1976 ^ ^ ^

1977 a <- 1:10

```

1978 a[1]
1979 ` ` `
1980 erzeugt
1981 a <- 1:10
1982 a[1]

1983 Momentan wird noch kein Code ausgeführt, sondern lediglich der Code, als Code dargestellt. Rmarkdown
1984 bietet nun die Möglichkeit Code beim kompilieren16 auszuführen. Dafür müssen wir nur einen Code-Block als
1985 R-Code-Block kennzeichnen.

1986 ` ` ` {R}
1987 a <- 1:10
1988 a[1]
1989 ` ` `

1990 erzeugt
1991 a <- 1:10
1992 a[1]

1993 ## [1] 1

1994 Beachte, die Variable a wird beim kompilieren erzeugt und steht dann R zur Verfügung. R-Code-Blöcke
1995 werden auch als Code Chunks bezeichnet. Diese Chunks können sehr genau angesprochen und angepasst
1996 werden. Einige wichtige Argumente sind:
1997
    • 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.

```

1998

Aufgabe 40: Arbeiten mit Rmarkdown

2001 Erstellen Sie eine neue Rmarkdown Datei mit dem Namen `test1.Rmd`. Erstellen Sie zwei Code-Chunks. Der
2002 erste soll nicht angezeigt werden und darin werden die Daten geladen (`bhd_1.txt`). Im zweiten Chunk plotten
2003 Sie das Alter der Bäume gegen den BHD. Was passiert mit dem Plot, wenn Sie die Datei kompilieren (drücken
2004 Sie dazu auf den Knit-Knopf; Abbildung 11).



Abbildung 11: Kompilieren einer `Rmd`-Datei.

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

2005 14 Räumliche Daten in R

2006 14.1 Was sind räumliche Daten

2007 Räumliche Daten sind Beobachtungen, wie wir sie schon oft gesehen haben, mit einem räumlichen Bezug. Der
 2008 Unterschied zu nicht räumlichen Daten liegt darin, dass räumliche Daten eindeutig im *Raum* verortet werden
 2009 können. Häufig werden sogenannte Geoinformationssysteme (GIS) zum Arbeiten mit räumlichen verwendet.
 2010 R kann in vielerlei Hinsicht wie ein GIS eingesetzt werden und hier werden einige Grundfunktionalitäten
 2011 dafür besprochen. Räumliche Daten werden in zwei unterschiedliche Datentypen unterteilt: Vektor- und
 2012 Rasterdaten. Vektordaten modellieren einzelne Objekte (= *Features*). Rasterdaten modellieren eine Oberfläche.
 2013 Vektordaten bestehen aus zwei Komponenten: 1) einer Geometrie, die die Form und Lage der Daten definiert
 2014 und 2) Attributen, den tatsächlichen Daten. Räumliche Daten werden oft als *Features* bezeichnet. Ein Feature
 2015 ist die räumliche Einheit einer Beobachtung. Je nach Art der räumlichen Daten können Features entweder
 2016 Punkte (z.B. ein Baum), Linien (z.B. eine Straße) oder Polygone (z.B. ein See) sein. Auch können mehrere
 2017 Geometrien zu einem Feature zusammengefasst werden. Ein Beispiel wäre eine Beobachtung für ein Land,
 2018 das aber aus mehreren Polygonen bestehen kann (z.B. Festland und Inseln). Features können dann weitere
 2019 Attribute (= Attributdaten) haben, z.B. eine ID, Name oder was auch immer man gemessen hat.
 2020 Rasterdaten bestehen aus einer Oberfläche von gleichgroßen Kacheln (= *Pixel*), die ein Gebiet abdecken.
 2021 Meist sind Pixel viereckig, aber das ist keine Voraussetzung. Dabei hat jedes Pixel einen Wert (das kann
 2022 auch ein fehlender Wert sein). Typische Beispiele für Rasterdaten sind Landnutzung oder Seehöhen.
 2023 In R kann sowohl mit Vektor- als auch mit Rasterdaten gearbeitet werden. Für Vektordaten bietet sich das
 2024 Paket **sf** an und für Rasterdaten das Paket **raster**.

2025 14.2 Koordinatenbezugssystem

2026 Eine der Herausforderungen für räumliche Daten ist die eindeutige Verortung im Raum. Dazu braucht man
 2027 ein *Koordinatenbezugssystem (KBS)*. Einem KBS liegt ein mathematisches Modell der Erde zugrunde. Die
 2028 Details zu KBS werden schnell relativ kompliziert und wir beschränken uns hier lediglich darauf, wie KBS
 2029 verwendet werden können. Dazu müssen wir zwei Fälle unterscheiden: 1) einem Datensatz ein KBS zuweisen
 2030 und 2) Transformation des KBSs eines Datensatzes in ein anderes KBS. Die technischen Details werden in
 2031 den folgend Abschnitten besprochen. Für beide Aufgaben müssen wir auf ein KBS verweisen können, ein
 2032 einfacher Ansatz dafür sind die sogenannten *EPSG-Codes*¹⁷.

2033 14.3 Vektordaten in R

2034 Das Paket **sf** stellt Klassen zum Abbilden von Features zur Verfügung, die dann in einem **data.frame** als
 2035 Liste gespeichert werden können. In der Regel erstellen wir Features nicht individuell, sondern lesen diese aus
 2036 externen Datenquellen (z.B. ESRI Shapefile) ein. Zum besseren Verständnis, erstellen wir es einmal manuell.
 2037 Wir haben die Koordinaten für drei Städte (Göttingen, Hannover und Berlin) als geografische Koordinaten
 2038 vorliegen (EPSG = 4326).

¹⁷EPSG steht für European Petrol Survey Group

```
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))
```

2039 Daraus können wir jetzt eine Geometriespalte für einen `data.frame` erstellen

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

2040 Somit haben wir die Geometrie in der Variable `geom` gespeichert, aber noch keine dazugehörigen Attributdaten.

2041 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)
)
```

2042 In einem letzten Schritt möchten wir jetzt die Geometrie (`geom`) und die Attributdaten (`attr`) zusammenführen.

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

```
2044 ## Simple feature collection with 3 features and 3 fields
2045 ## Geometry type: POINT
2046 ## Dimension: XY
2047 ## Bounding box: xmin: 9.732 ymin: 51.5413 xmax: 13.405 ymax: 52.52
2048 ## Geodetic CRS: WGS 84
2049 ##           name   bundesland   einwohner          geom
2050 ## 1 Goettingen Niedersachsen    119000 POINT (9.9158 51.5413)
2051 ## 2 Hannover Niedersachsen    532000 POINT (9.732 52.3759)
2052 ## 3 Berlin      Berlin    3650000 POINT (13.405 52.52)
```

2053 Wir können nun mit `staedte` genau so arbeiten wie mit jedem anderen `data.frame` und die Geometrien werden immer ‘berücksichtigt’. Zusätzlich kann man eine Reihe von geometrischen Operationen durchführen.

2055 Wenn ein `data.frame` Punkten hat, kann man dies relative einfach mit der Funktion `st_as_sf()` “räumlich” machen. Für das vorherige Beispiel würden wir zuerst einen `data.frame` mit allen Informationen (zur 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)
)
```

2058 Dann kann man mit der Funktion `st_as_sf()` weiter arbeiten:

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

2059 14.4 Arbeiten mit Vektordaten

2060 Es gibt sehr viele Funktionen, um mit räumlichen Daten zu arbeiten, von denen wir hier einige vorstellen.

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

```
2061 ## Coordinate Reference System:
2062 ##   User input: EPSG:4326
2063 ##   wkt:
2064 ## GEOGCRS["WGS 84",
2065 ##           ENSEMBLE["World Geodetic System 1984 ensemble",
2066 ##                     MEMBER["World Geodetic System 1984 (Transit)"],
2067 ##                     MEMBER["World Geodetic System 1984 (G730)"],
2068 ##                     MEMBER["World Geodetic System 1984 (G873)"],
2069 ##                     MEMBER["World Geodetic System 1984 (G1150)"],
2070 ##                     MEMBER["World Geodetic System 1984 (G1674)"],
2071 ##                     MEMBER["World Geodetic System 1984 (G1762)"],
2072 ##                     MEMBER["World Geodetic System 1984 (G2139)"],
2073 ##                     ELLIPSOID["WGS 84",6378137,298.257223563,
2074 ##                               LENGTHUNIT["metre",1]],
2075 ##                     ENSEMBLEACCURACY[2.0]],
2076 ##           PRIMEM["Greenwich",0,
2077 ##                  ANGLEUNIT["degree",0.0174532925199433]],
2078 ##           CS[ellipsoidal,2],
2079 ##             AXIS["geodetic latitude (Lat)",north,
2080 ##                   ORDER[1],
2081 ##                   ANGLEUNIT["degree",0.0174532925199433]],
2082 ##             AXIS["geodetic longitude (Lon)",east,
2083 ##                   ORDER[2],
2084 ##                   ANGLEUNIT["degree",0.0174532925199433]],
2085 ##           USAGE[
2086 ##             SCOPE["Horizontal component of 3D system."],
2087 ##             AREA["World."],
2088 ##             BBOX[-90,-180,90,180]],
2089 ##             ID["EPSG",4326]]
```

2090 Wenn wir jetzt zu einem anderen KBS (z.B. EPSG:3035, ein europäisches projiziertes KBS) umrechnen
 2091 möchten, können wir das mit

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

```

2092 ## Coordinate Reference System:
2093 ##   User input: EPSG:3035
2094 ##   wkt:
2095 ## PROJCRS["ETRS89-extended / LAEA Europe",
2096 ##   BASEGEOGCRS["ETRS89",
2097 ##     ENSEMBLE["European Terrestrial Reference System 1989 ensemble",
2098 ##       MEMBER["European Terrestrial Reference Frame 1989"],
2099 ##       MEMBER["European Terrestrial Reference Frame 1990"],
2100 ##       MEMBER["European Terrestrial Reference Frame 1991"],
2101 ##       MEMBER["European Terrestrial Reference Frame 1992"],
2102 ##       MEMBER["European Terrestrial Reference Frame 1993"],
2103 ##       MEMBER["European Terrestrial Reference Frame 1994"],
2104 ##       MEMBER["European Terrestrial Reference Frame 1996"],
2105 ##       MEMBER["European Terrestrial Reference Frame 1997"],
2106 ##       MEMBER["European Terrestrial Reference Frame 2000"],
2107 ##       MEMBER["European Terrestrial Reference Frame 2005"],
2108 ##       MEMBER["European Terrestrial Reference Frame 2014"],
2109 ##       ELLIPSOID["GRS 1980",6378137,298.257222101,
2110 ##         LENGTHUNIT["metre",1]],
2111 ##       ENSEMBLEACCURACY[0.1]],
2112 ##       PRIMEM["Greenwich",0,
2113 ##         ANGLEUNIT["degree",0.0174532925199433]],
2114 ##       ID["EPSG",4258],
2115 ##       CONVERSION["Europe Equal Area 2001",
2116 ##         METHOD["Lambert Azimuthal Equal Area",
2117 ##           ID["EPSG",9820]],
2118 ##         PARAMETER["Latitude of natural origin",52,
2119 ##           ANGLEUNIT["degree",0.0174532925199433],
2120 ##           ID["EPSG",8801]],
2121 ##         PARAMETER["Longitude of natural origin",10,
2122 ##           ANGLEUNIT["degree",0.0174532925199433],
2123 ##           ID["EPSG",8802]],
2124 ##         PARAMETER["False easting",4321000,
2125 ##           LENGTHUNIT["metre",1],
2126 ##           ID["EPSG",8806]],
2127 ##         PARAMETER["False northing",3210000,
2128 ##           LENGTHUNIT["metre",1],
2129 ##           ID["EPSG",8807]]],
2130 ##       CS[Cartesian,2],
2131 ##         AXIS["northing (Y)",north,
2132 ##           ORDER[1],
2133 ##           LENGTHUNIT["metre",1]],
2134 ##         AXIS["easting (X)",east,

```

```

2135 ##           ORDER[2],
2136 ##           LENGTHUNIT["metre",1]],
2137 ##           USAGE[
2138 ##           SCOPE["Statistical analysis."],
2139 ##           AREA["Europe - European Union (EU) countries and candidates. Europe - onshore and offshore: "],
2140 ##           BBOX[24.6,-35.58,84.73,44.83]],
2141 ##           ID["EPSG",3035]

```

2142 Die Funktion `st_buffer()` erlaubt es Features zu puffern, mit `st_distance()` kann die Distanz zwischen
2143 Features berechnet werden, mit `st_area()` kann die Fläche eines Features berechnet werden.

2144 Funktionen wie `st_intersection()`, `st_union()` und `st_difference()` erlauben es geometrische Opera-
2145 tionen zwischen unterschiedlichen Features zu berechnen. Für eine ausführliche Diskussion siehe auch hier:
2146 <https://geocompr.robinlovelace.net/geometric-operations.html>.

2147 Normalerweise lesen wir Daten von externen Datenquellen (z.B. ESRI Shapefiles). Das geht mit der Funktion
2148 `st_read()`.

2149 14.5 Rasterdaten in R

2150 Für Rasterdaten gibt es das R-Paket `raster`. Auch hier wollen wir uns wieder auf einige Grundfunktionalitäten
2151 konzentrieren. Diese umfassen das Einlesen, Zuschneiden, Rechnen und Abfragen von Rastern.

2152 Mit der Funktion `raster()` kann ein Raster in R eingelesen werden.

```

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

```

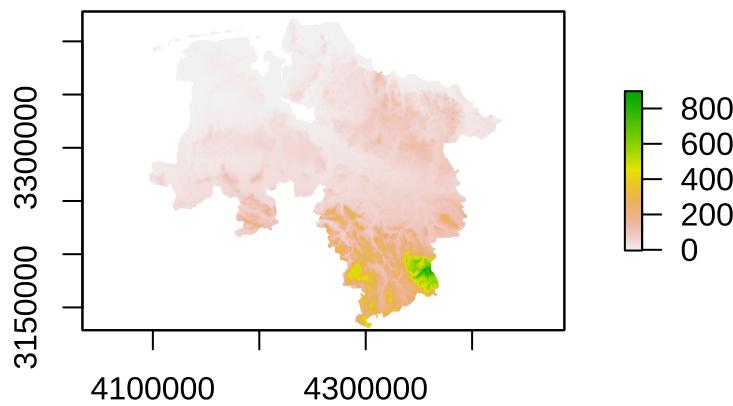
2153 `dem` steht für *Digital Elevation Model* und ist ein Raster mit den Seehöhen in Niedersachsen mit einer
2154 500-m-Auflösung. Wir können diese mit der Funktion `res()`¹⁸ abfragen.

```
res(dem)
```

2155 ## [1] 500 500

2156 Bzw. wir können den Raster auch plotten.

```
plot(dem)
```



2157

¹⁸kurz für *resolution* also Auflösung.

2158 Wenn wir den Raster `dem` auf ein Gebiet zuschneiden wollen (z.B. Göttingen), müssen wir drei Schritte
 2159 durchführen. Als erstes müssen wir ein Shapefile für Göttingen einlesen.

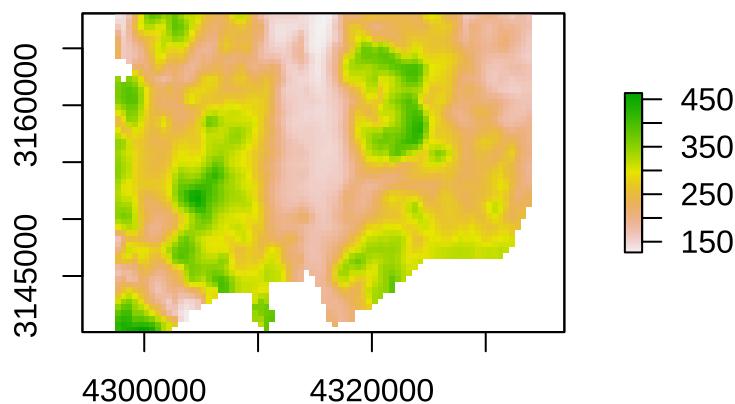
```
goe <- st_read(here::here("data/goettingen/stadt_goettingen.shp"))
```

2160 Dann müssen wir sicherstellen, dass sowohl der Raster `dem` als auch das `sf`-Objekt `goe` im selben KBS sind.
 2161 Es bietet es sich in der Regel an, das KBS des Vektors zu transformieren. Mit der Funktion `projectRaster()`
 2162 kann das KBS eines Rasters transformiert werden.

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

2163 Mit der Funktion `crop()` kann der Raster `dem` auf Göttingen zugeschnitten werden.

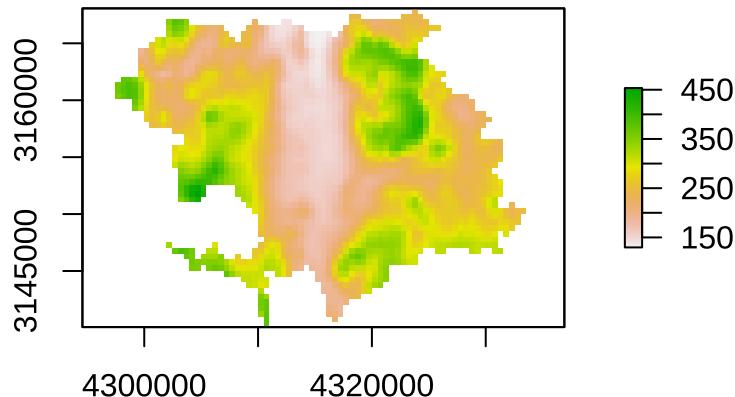
```
dem1 <- crop(dem, goe)
plot(dem1)
```



2164

2165 Der Raster hat jetzt die Größe einer Bounding-Box (BBX) von Göttingen (das ist ein Rechteck, das Göttingen
 2166 umfasst). Mit der Funktion `mask()` kann der Raster auf die genauen Grenzen des Vektors `goe` angepasst
 2167 werden.

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



2168

2169 Wenn wir an bestimmten Punkten den Wert des Rasters abfragen wollen (z.B. an `cities`) von vorhin, dann
 2170 gibt es dafür die Funktion `extract`. Dann müssen wir erst sicherstellen, dass `staedte` und `dem` gleichen
 2171 KBS zu grunde liegt. Dafür transformieren wir einfach `staedte` in das KBS von `dem`. Mit der Funktion
 2172 `projection()` erhalten wir das KBS des Rasters.

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

2173 Wenn wir das KBS eines Objektes nicht kennen, können wir auch einfach das KBS übergeben. Der folgende
 2174 Code-Block macht genau das Gleiche mit dem Vorteil, dass wir keinen EPSG-Code angeben müssen.

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

2175 Dann können wir für jede Stadt die Seehöhe abfragen:

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

2176 ## [1] 149.18181 57.21486 NA

2177 Mit `raster::extract()` rufen wir *eindeutig* die Funktion `extract()` aus dem Paket `raster` auf. Wir müssen
 2178 das so machen, weil es im Paket `dplyr` auch eine Funktion `extract()` gibt, die wir hier nicht anwenden
 2179 möchten, da sie einen Fehler verursachen würde.

2180 Ein analoges Vorgehen ist auch für Linien und Polygone möglich.

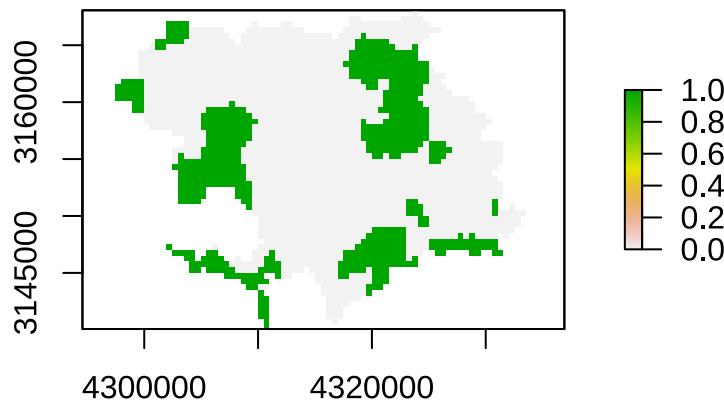
2181 Mit Rastern kann auch einfach gerechnet werden. Wir können z.B. die Seehöhe in Kilometern anstatt Metern
 2182 berechnen:

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

2183 Auch logische Operationen sind möglich, wenn wir alle Rasterzellen mit einer Seehöhe von mehr als 300 m in
 2184 Göttingen suchen, dann geht das so:

```
dem3 <- dem2 > 300
```

```
plot(dem3)
```



2185

2186 Wenn wir jetzt auf die Werte des Rasters `dem3` zugreifen wollen, geht das mit eckigen Klammern.

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

2187 ## [1] NA NA NA NA NA NA

2188 Das sind erst einmal viele `NA`-Werte für die ganzen Zellen, die außerhalb von Niedersachsen liegen. Aber wir
 2189 können mit so einem Vektor ganz normal arbeiten und z.B. die Fläche des Landes Niedersachsen die eine
 2190 Seehöhe von mehr als 500m Seehöhe hat ausrechnen.

```

2190 h <- dem3[]
2191 sum(h, na.rm = TRUE) / sum(!is.na(h))
2192 ## [1] 0.265713
2193
2194

```

Aufgabe 41: Arbeiten mit Rastern

2195 Verwenden Sie den Raster `wald.tif`, der auf einer 10 m Auflösung den Waldanteil jeder Rasterzelle angibt¹⁹.
 2196 Der EPSG-Code für das KBS von `wald.tif` ist 3035. Nehmen Sie an, dass wenn der Waldanteil in einer
 2197 Raster größer als 50 % ist, dass die Rasterzelle als Wald klassifiziert werden kann. Wie viel Prozent des
 2198 Göttinger Stadtgebietes sind Wald? Wie ändert sich dieser Wert, wenn sie 70 % anstatt 50 % als Schwellenwert
 2199 für Wald annehmen?

2200

Aufgabe 42: Studiendesign

2201 Mit der Funktion `st_sample()` können Sie innerhalb oder entlang eines Features zufällige Punkte legen. Das
 2202 Argument `n` steuert die Anzahl Punkte und das Argument `type` wie die Punkte angeordnet werden. Für `type`
 2203 sind für uns die Werte `type = "random"` (komplett zufällig), `type = "regular"` (regelmäßiger Grid) und
 2204 `type = "hexagonal"` von Bedeutung (ein hexagonaler Grid, d.h. ein sechseckiger Raster). Unglücklicherweise
 2205 ist das Ergebnis von `st_sample()` erst eine Geometrie. Um daraus ein vollständiges `sf`-Objekt zu machen
 2206 und problemlos weiter arbeiten zu können, müssen Sie nocheinmal die Funktion `st_as_sf()` ausführen.
 2207
 2208 Stellen Sie sich vor, dass wir die tatsächliche Waldbedeckung des Göttinger Stadtgebietes **nicht** kennen und wir
 2209 eine Studie durchführen, um den Anteil des Göttinger Stadtgebietes, der mit Wald bedeckt ist herauszufinden.
 2210 Erstellen Sie dafür einige unterschiedliche Stichproben (diese können in der Anzahl und Anordnung variieren).
 2211
 2212 Berechnen Sie für jedes Stichprobendesign den Anteil an Wald und ein dazugehöriges Konfidenzintervall
 2213 (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
 2214 die Stichprobengröße). Nehmen Sie an, dass eine Rasterzelle Wald ist, sobald $> 50\%$ der Rasterzelle mit
 2215 Wald bedeckt ist.

2216

Aufgabe 43: Räumliche Daten

2217 Verwenden Sie den folgenden Datensatz:

```

2218 set.seed(123)
2219 df1 <- data.frame(
2220   x = runif(100, 0, 100),
2221

```

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

```

y = runif(100, 0, 100),
kronendurchmesser = runif(100, 1, 15),
art = sample(letters[1:4], 100, TRUE)
)

```

- 2220 1. Erstellen Sie ein `sf`-Objekt aus `df1`.
 2221 2. Puffern Sie jeden Baum mit seinem Kronendurchmesser.
 2222 3. Berechnen Sie die Kronenfläche jedes Baumes. *Hinweis: Die Funktion `st_area()` könnte dafür hilfreich sein.*
 2223 4. Welcher Baum hat die größte Kronenfläche?
 2224 5. Finden Sie für jede Art, den Baum mit der größten Kronenfläche.

2226

Aufgabe 44: Arbeiten mit räumlichen Daten

- 2229 1. Lesen Sie das ESRI Shapefile `goettingen/stadt_goettingen.shp` ein.
 2230 2. Wie viele Features befinden sind in dem Shapefile?
 2231 3. Welches Koordinatenbezugssystem (KBS) hat das Shapefile?
 2232 4. Transformieren Sie das Shapefile in das KBS 3035.
 2233 5. Erstellen Sie eine neue Spalte `A` in der Sie die Fläche jeder Gemeinde/Stadt speichern.
 2234 6. Welche Gemeinde/Stadt (Spalte `GEN`) ist am größten?
 2235 7. Wählen Sie nun nur die Stadt Göttingen aus.

2236

Aufgabe 45: Arbeiten mit räumlichen Daten 2

- 2239 1. Lesen Sie erneut das ESRI Shapefile `goettingen/stadt_goettingen.shp` ein.
 2240 2. Lösen sie die Gemeindegrenzen auf (die Funktion `st_union()` könnte hier nützlich sein).
 2241 3. Wie groß ist das resultierende Feature?

2242 15 FAQs (Oft gefragtes)

2243 15.1 Arbeiten mit Daten

2244 15.1.1 Einlesen von Exceldateien

2245 Mit der Funktion `read_excel()` aus dem Paket `readxl` können Exceldateien direkt in R eingelesen werden.

2246 Ein Export als csv-Datei aus Excel ist nicht notwendig.

2247 16 Literatur

- 2248 Ein guter Überblick über viele der angeschnittenen Themen gibt es in dem [R for data science](#), das online frei
2249 zugänglich ist. Das on-line Buch [Hands-On Programming with R]{<https://rstudio-education.github.io/hopr/index.html>} ist eine nicht-Programmierer freundliche Einführung in R.
2250
- 2251 McNamara, Amelia, and Nicholas J Horton. 2018. “Wrangling Categorical Data in r.” *The American Statistician*
2252 72 (1): 97–104.
- 2253 Wickham, Hadley. 2014. “Tidy Data.” *Journal of Statistical Software* 59 (10). <https://doi.org/10.18637/jss.v059.i10>.
- 2254