

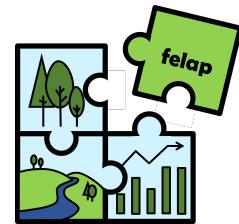
1

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:

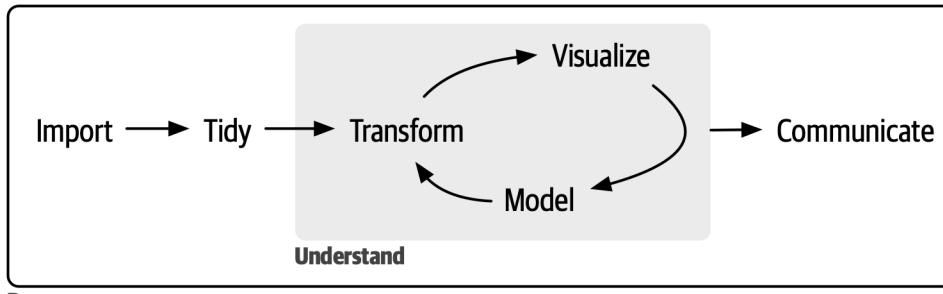
¹⁶ Signer, J. und Husmann, K. (2023) Skript zur Vorlesung Einführung in die Datenanalyse mit R, Georg-August-Universität Göttingen.

¹⁸ Letzte Aktualisierung: 20. 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 Funktionen	8
52	2.3 Datentypen	9
53	2.4 Datenstrukturen	10
54	3 Vektoren	12
55	3.1 Funktionen zum Arbeiten mit Vektoren	14
56	3.2 Statistische Funktionen	15
57	3.3 Beispiel Fotofallen	16
58	3.4 Arbeiten mit logischen Werten	17
59	3.5 Zugreifen auf Elemente eines Vektors (=Untermengen)	19
60	3.6 Der %in%-Operator	20
61	4 Faktoren (factors)	22
62	4.1 Das Paket forcats	23
63	4.1.1 Anpassen der Anordnung von Faktoren	24
64	5 Spezielle Einträge	25
65	5.1 NA	25
66	5.2 NULL	26
67	5.3 Inf	26
68	6 data.frames oder Tabellen	28
69	6.1 Wichtige Funktionen zum Arbeiten mit data.frames	29
70	6.2 Zugreifen auf Elemente eines data.frame	30
71	7 Schreiben und lesen von Daten	33
72	7.1 Textdateien	33
73	8 Erstellen von Abbildungen	35
74	8.1 Base Plot	35
75	8.1.1 Mehrere Panels	41
76	8.1.2 Speichern von Abbildungen	41
77	8.2 Histogramme	42
78	8.3 Boxplots	45
79	8.4 ggplot2 : Eine Alternative für Abbildungen	47
80	8.4.1 Multipanel Abbildungen	54

81	8.4.2 Plots kombinieren	56
82	8.4.3 Speichern von plots	59
83	9 Mit Daten arbeiten	60
84	9.1 dplyr eine Einführung	60
85	9.2 Arbeiten mit gruppierten Daten	63
86	9.3 pipes oder %>%	64
87	9.4 Joins	65
88	9.5 'long' and 'wide' Datenformate	67
89	9.6 Auswählen von Variablen	69
90	9.7 Einzelne Beobachtungen abfragen (slice())	70
91	9.8 Spalten trennen	73
92	10 Arbeiten mit Text	75
93	10.1 Arbeiten mit Text	75
94	10.2 Finden von Textmustern	76
95	11 Arbeiten mit Zeit	79
96	11.1 Arbeiten mit Zeitintervallen	80
97	11.2 Formatieren von Zeit	82
98	11.3 Zeitreihen	82
99	12 Aufgaben Wiederholen (for-Schleifen)	87
100	12.1 Schleifen	87
101	12.1.1 Wiederholen von Befehlen mit for()	87
102	12.1.2 Wiederholen von Befehlen mit while()	90
103	12.2 Bedingte Ausführung von Codeblöcken	90
104	13 (R)markdown	92
105	13.1 Markdown Grundlagen	92
106	13.2 R und Markdown	93
107	14 Räumliche Daten in R	95
108	14.1 Was sind räumliche Daten	95
109	14.2 Koordinatenbezugssystem	95
110	14.3 Vektordaten in R	95
111	14.4 Arbeiten mit Vektordaten	97
112	14.5 Rasterdaten in R	99
113	15 FAQs (Oft gefragtes)	104
114	15.1 Arbeiten mit Daten	104
115	15.1.1 Einlesen von Exceldateien	104
116	16 Literatur	105

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]**).

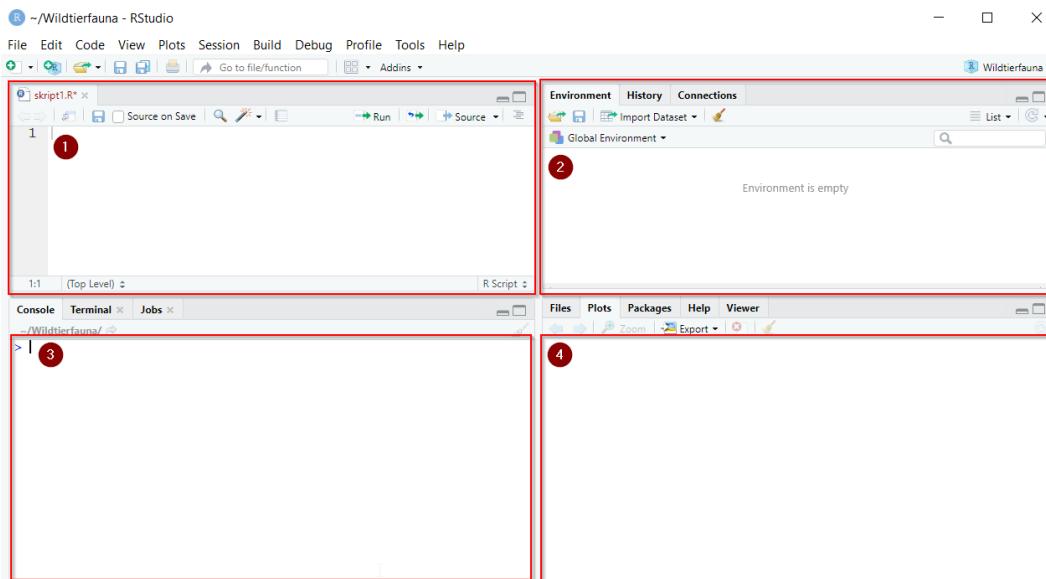


Abbildung 1: RStudio Panes.

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.

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

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

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

10 + 5

151 **## [1] 15**

20 - 10

152 **## [1] 10**

10 * 3

153 **## [1] 30**

100 / 19

154 **## [1] 5.263158**

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

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

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

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

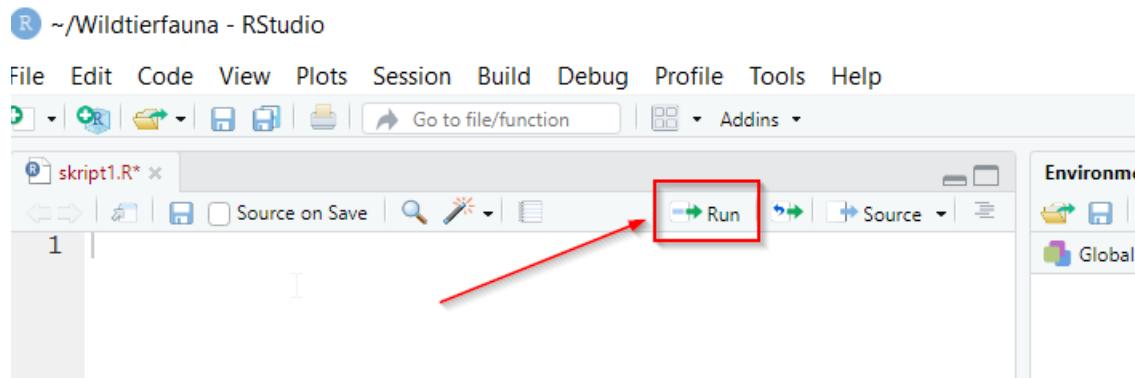


Abbildung 2: Zeilenweises Ausführen von Code in RStudio.

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

1.3 Gute Praxis bei der Programmierung

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

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

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

193 ## [1] 9

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

200

201 **Aufgabe 1: Ausführen von Quellcodes**

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

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

# Einfache Funktion
sqrt(20)
```

205 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
```

```

229 ## [1] 15
b / a

230 ## [1] 0.5
a^b

231 ## [1] 1e+05

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

ergebnis <- a + b
ergebnis

233 ## [1] 15

ergebnis2 <- ergebnis * 2
ergebnis2

234 ## [1] 30

235 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
236 Variablen wiederherzustellen. Sie müssen ggf. neu berechnet werden.

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

238 ## [1] TRUE

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

239 ## [1] FALSE

240 2.2 Funktionen

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

sqrt(a)

243 ## [1] 3.162278

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

248 Argumente von Funktionen haben Namen, diese müssen jedoch nicht angegeben werden, wenn die Reihenfolge
249 der Argumente berücksichtigt wird. Im vorherigen Beispiel, haben wir einfach die Funktion sqrt(a) aufgerufen
250 und keinen Argumentnamen angegeben. Aus den Hilfeseiten für die Funktion sqrt() (siehe auch nachfolgender

```

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

```
253 sqrt(x = a)
254 ## [1] 3.162278
255 Um mehr über eine Funktion zu erfahren (z.B. die Bedeutung von Argumenten zu verstehen oder heraus-
256 zufinden, was eine Funktion zurück gibt), können wir die Hilfeseiten konsultieren. Es gibt unterschiedliche
257 Wege, um zu einer Hilfeseite zu gelangen.
258 1. In die Konsole ?<Name der Funktion> tippen. Also, wenn wir Hilfe für die Funktion mean() möchten,
259 könnten wir einfach ?mean in die Konsole tippen.
260 2. Analog zu 1), können wir mit der Funktion help die Hilfeseite für eine andere Funktion aufrufen (z.B.
261 wenn wir wieder die Hilfe für die Funktion mean() lesen möchten, dann könnten wir auch help(mean)
262 in die Konsole tippen).
263 3. In RStudio kann man auch auf das Help-Panel klicken und dann einfach eine Funktion suchen (siehe
264 Abbildung 1).
265 4. Wenn der Cursor in RStudio auf einer Funktion ist, kann man mit der Taste F1 die dazugehörige
Hilfeseite aufrufen.
```

266 2.3 Datentypen

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

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

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

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

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

```
fuchs_gesehen <- TRUE
```

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

```
typeof(fuchs_gesehen)
```

285 ## [1] "logical"

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

```
TRUE + TRUE
```

287 ## [1] 2

```
FALSE + FALSE
```

288 ## [1] 0

```
TRUE + FALSE
```

289 ## [1] 1

2.4 Datenstrukturen

290 Verfolgen wir das Beispiel mit den Fotofallen etwas weiter. Es handelt sich um ein systematisches Monitoring.

291 D. h., es wurde nicht nur eine Fotofalle ausgebracht, sondern insgesamt 15 Stück. Dieser Umstand erfordert komplexere Objekte. Nachfolgend sind die Anzahl Rehfotos für jede der 15 Fotofallen aufgeführt: 132, 79, 129, 91, 138, 144, 55, 103, 139, 105, 96, 146, 95, 118, 107.

292 Die Frage, die sich jetzt stellt, ist: Wie kann man diese Daten sinnvoll organisieren? Zusätzlich zur Anzahl der fotografierten Rehe soll jede Fotofalle eine eindeutige ID haben (`Kamera_1`, ..., `Kamera_15`) und wir wissen, dass jeweils 5 Fotofallen in drei unterschiedlichen Revieren aufgestellt waren (Fotofalle 1 bis 5 in Revier A, Fotofalle 6 bis 10 in Revier B und Fotofalle 11 bis 15 in Revier C). Wir könnten für jede Kamera und jeden 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.
```

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

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

302 Frames) für diesen Zweck kennenlernen.

303

304 **Aufgabe 2: Variablen**

305 306 Verwenden Sie die folgenden Daten

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

307 und berechnen sie:

- 308 • $10 * a$
309 • $a / 144$ und speichern Sie das Ergebnis in einer neuen Variablen e zwischen.
310 • Was ist das Ergebnis von $a + b$?
311 • Was ist das Ergebnis von $a + p$?

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

3 Vektoren

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

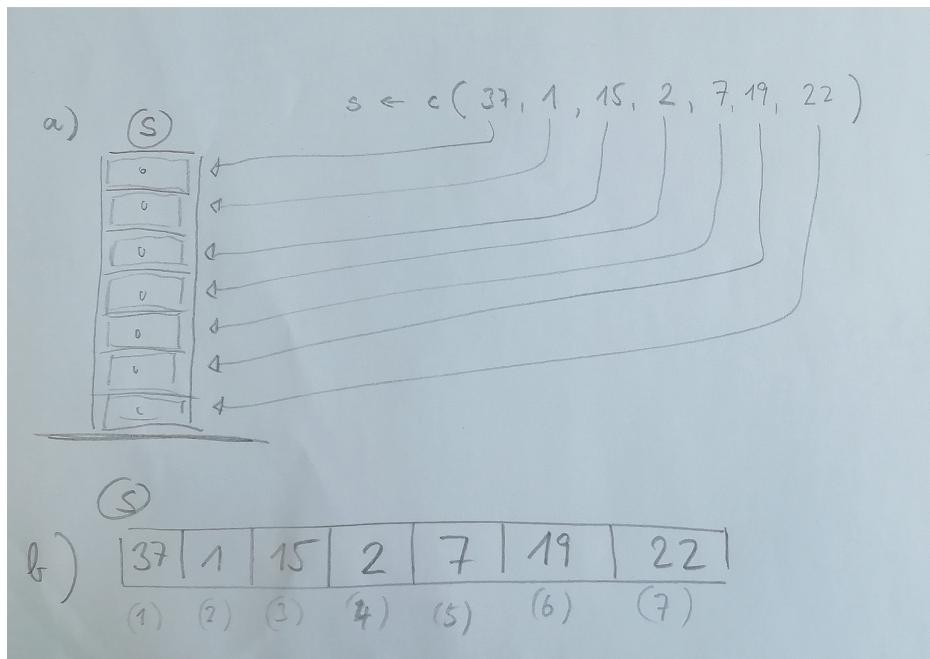


Abbildung 3: Schematische Darstellung eines Vektors in R.

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

322 Es gibt zahlreiche Funktionen zum Erstellen von Vektoren (einige davon werden wir im weiteren Verlauf des Moduls kennenlernen). Die wohl wichtigste Funktion ist `c()`. Dabei steht `c` für *combine* oder *concatenate*. 323 Die Funktion `c()` fügt einzelne Elemente in einen Vektor zusammen (und zwar genau in der Reihenfolge wie 324 diese Elemente an `c()` übergeben werden). `c()` ist sozusagen die Funktion, die für Sie mehrere Elemente zu 325 einem Vektor zusammensetzt. `c()` erwartet folglich nur Elemente als Argumente. 326

327 Gehen wir nochmals zurück zu Abbildung 3, in der schematisch dargestellt wird, wie ein Vektor `s` mit 7 328 Elementen (in diesem Fall Zahlen) erstellt wird.

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

329 Die Funktion `c()` ordnet jetzt bildlich gesprochen die Zahl 37 der ersten Schublade zu, die Zahl 1 der zweiten 330 Schublade und so weiter. Wenn Sie jetzt einfach `s` in die Konsole tippen, können Sie alle Elemente von `s` 331 sehen:

s

332 ## [1] 37 1 15 2 7 19 22

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

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

s + 10

338 ## [1] 47 11 25 12 17 29 32

339 oder `s` mit sich selbst multiplizieren. Zu beachten ist also, dass es sich bei Vektorberechnungen in R
340 zunächst nicht um Vektorrechnungen handelt, wie Sie sie aus der linearen Algebra kennen. Für die sog.
341 Matrizenoperationen der linearen Algebra werden die Operatoren in R mit % % umschlossen, also bspw. `s`
342 %*% `s`.

s * s

343 ## [1] 1369 1 225 4 49 361 484

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

seq(from = 1, to = 10)

347 ## [1] 1 2 3 4 5 6 7 8 9 10

(1 : 10)

348 ## [1] 1 2 3 4 5 6 7 8 9 10

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

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

350 ## [1] 1 3 5 7 9

351

352 Aufgabe 3: Vektoren erstellen

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

- 355 • Erstellen Sie einen Vektor mit dem Namen `bhd` in dem Sie die Werte speichern
- 356 • Transformieren Sie die BHD-Werte in mm.
- 357 • 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.

358 3.1 Funktionen zum Arbeiten mit Vektoren

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

```
head(s)
```

361 `## [1] 37 1 15 2 7 19`

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

362 `## [1] 37 1 15`

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

363 `## [1] 19 22`

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

```
length(s)
```

365 `## [1] 7`

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

```
class(s)
```

367 `## [1] "numeric"`

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

```
unique(s)
```

369 `## [1] 37 1 15 2 7 19 22`

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

```
table(s)
```

371 `## s`

372 `## 1 2 7 15 19 22 37`

373 `## 1 1 1 1 1 1 1`

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

```
rev(s)
```

376 `## [1] 22 19 7 2 15 1 37`

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

```
sort(s)
```

378 `## [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.

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

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

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

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

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

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

383 ## [1] 1 2 3 4 1 2 3 4

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

384 ## [1] 1 1 2 2 3 3 4 4

385

386 Aufgabe 4: Arbeiten mit Vektoren

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

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

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

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

3.2 Statistische Funktionen

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

```
mean(s)
```

396 ## [1] 14.71429

```
median(s)
```

397 ## [1] 15

```
sd(s)
```

398 ## [1] 12.76341

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

```

sample(s, size = 1) # 1 Element
402 ## [1] 1
sample(s, size = 3) # 2 Elemente
403 ## [1] 15 7 22

```

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

406 3.3 Beispiel Fotofallen

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

410 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,
411 105, 96, 146, 95, 118, 1007)

```

412 Als zweites erstellen wir einen Vektor mit den IDs. Zur Erinnerung, diese sollten die Werte `Kamera_1` bis
413 `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",
414 "Kamera_6", "Kamera_7", "Kamera_8", "Kamera_9", "Kamera_10",
"Kamera_11", "Kamera_12", "Kamera_13", "Kamera_14", "Kamera_15"
)

```

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

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

```

v1 <- rep("Kamera", 15)

```

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

```

v2 <- 1:15

```

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

```

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

```

```

424 ## [1] "Kamera_1"  "Kamera_2"  "Kamera_3"  "Kamera_4"  "Kamera_5"  "Kamera_6"
425 ## [7] "Kamera_7"  "Kamera_8"  "Kamera_9"  "Kamera_10" "Kamera_11" "Kamera_12"
426 ## [13] "Kamera_13" "Kamera_14" "Kamera_15"

```

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

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

```

429 ## [1] "Revier A" "Revier B" "Revier C" "Revier A" "Revier B" "Revier C"
430 ## [7] "Revier A" "Revier B" "Revier C" "Revier A" "Revier B" "Revier C"
431 ## [13] "Revier A" "Revier B" "Revier C"

```

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

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

```

434 ## [1] "Revier A" "Revier A" "Revier A" "Revier A" "Revier A" "Revier B"
435 ## [7] "Revier B" "Revier B" "Revier B" "Revier B" "Revier C" "Revier C"
436 ## [13] "Revier C" "Revier C" "Revier C"

```

437

438 Aufgabe 5: Statistische Funktionen

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

441 2. Erstellen Sie die folgende Konsolenausgabe:

```
442 ## [1] "Die mittlere Anzahl von Rehfotos beträgt 171.8 Rehe pro Standort."
```

443 3.4 Arbeiten mit logischen Werten

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

- 446 • Gleichheit (`==`)
- 447 • Ungleichheit (`!=`)
- 448 • Größer (`>`) und kleiner (`<`)
- 449 • Größer gleich (`>=`) und kleiner gleich (`<=`)

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

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

```
anzahl_rehe > 100
```

```
453 ## [1] TRUE FALSE TRUE FALSE TRUE TRUE FALSE TRUE TRUE TRUE FALSE TRUE
```

```

454 ## [13] FALSE TRUE TRUE
455 Das Ergebnis ist ein Vektor vom Datentyp logi in der selben Länge wie anzahl_rehe.
456 Wir können z. B. abfragen, welche Fotofallenstandorte sich in Revier B befinden.
457 reviere == "Revier B"
458 ## [13] FALSE FALSE FALSE FALSE FALSE TRUE TRUE TRUE TRUE TRUE FALSE FALSE
459 Des Weiteren können logische Ausdrücke miteinander verknüpft werden. Dies geschieht mit einem logischen
460 Und (&) oder einem logischen Oder (|). Für das logische Und müssen beide Ausdrücke ein TRUE zurückgeben
461 um ein TRUE zu erhalten. Für ein logisches Oder reicht es, wenn einer der beiden Ausdrücke TRUE zurückgibt,
462 um ein TRUE zu erhalten.
463 Damit können wir nun z. B. die beiden vorherigen Abfragen verbinden. Die erste Abfrage ist: Hat eine
464 Fotofalle mehr als 100 Rehe fotografiert und stand die Fotofalle in Revier B.
465 anzahl_rehe > 100 & reviere == "Revier B"
466 ## [13] FALSE FALSE FALSE FALSE FALSE TRUE FALSE TRUE TRUE TRUE FALSE FALSE
467 Das war jetzt eine Abfrage mit einem logischen Und. Würden wir ein logisches Oder verwenden, dann
468 bekommen wir für alle Elemente ein TRUE, die entweder in Gebiet B stehen oder mehr als 100 Rehfotos
469 aufgezeichnet haben.
470 anzahl_rehe > 100 | reviere == "Revier B"
471 ## [13] TRUE FALSE TRUE FALSE TRUE TRUE TRUE TRUE TRUE TRUE FALSE TRUE
472 Das Arbeiten mit logischen Werten kann fürs Erste etwas abstrakt erscheinen, aber wir werden im folgenden
473 Abschnitt (Abschnitt 3.5) zahlreiche Anwendungsbeispiele dafür sehen.
474

```

475 Aufgabe 6: Arbeiten mit logischen Werten

- 477 Überlegen Sie für jede Aufgabe erst was das richtige Ergebnis ist und Überprüfen Sie dieses dann mit R.
- 478 1. `TRUE | FALSE`
 - 479 2. `FALSE & TRUE`
 - 480 3. `(FALSE & TRUE) | TRUE`
 - 481 4. `(2 != 3) | FALSE`
 - 482 5. `FALSE + 10`
 - 483 6. `TRUE + 10`
 - 484 7. `TRUE + 10 == FALSE + 10`
 - 485 8. `sum(c(TRUE, TRUE, FALSE, FALSE))`

486 3.5 Zugreifen auf Elemente eines Vektors (=Untermengen)

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

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

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

```
anzahl_rehe[2]
```

500 ## [1] 79

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

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

502 ## [1] 132 79 129 91 138

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

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

506

507 Aufgabe 7: Zugreifen auf Vektorelemente

509 Erstellen Sie einen neuen Vektor `bhd`

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

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

513

514 Alternativ könnte das gleiche Ergebnis mit einem logischen Vektor erreicht werden. Für eine bessere Über-
 515 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]
```

516 ## [1] 132 79 129 91 138

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

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

520 ## [1] 132 79 129 91 138

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

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

523 ## [1] 132 79 129 91 138

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

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

526 ## [1] 113.8

527

528 Aufgabe 8: logische Werte

530 Verwenden Sie nochmals den Vektor mit den Anzahl Rehen die an unterschiedlichen Fotofallenstandorten
 531 fotografiert wurden.

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

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

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

535 3.6 Der %in%-Operator

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

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

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

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

## [1] "FI" "FI"

# oder

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

- 541 ## [1] "FI" "FI"
542 Etwas komplizierter wird es, wenn wir zwei oder mehr Elemente auswählen wollen. Dies geht auch mit
543 logischen Operationen.

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

- 544 ## [1] "FI" "BU" "BU" "FI"
545 Diese Herangehensweise wird aber für > 2 Elemente in `arten` sehr mühsam und fehleranfällig. Eine Alternative
546 bietet der `%in%`-Operator. Dieser testet, ob Elemente eines Vektors in einem zweiten Vektors enthalten sind.
547 Der Operator ist also eine verkürzte Schreibweise für hintereinander durchgeführte logische Oder Abfragen.

```
messungen_arten %in% arten
```

- 548 ## [1] TRUE TRUE TRUE FALSE FALSE FALSE TRUE FALSE FALSE
messungen_arten[messungen_arten %in% arten]

- 549 ## [1] "FI" "BU" "BU" "FI"

550

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

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

```
LETTERS
```

- 554 ## [1] "A" "B" "C" "D" "E" "F" "G" "H" "I" "J" "K" "L" "M" "N" "O" "P" "Q" "R" "S"
555 ## [20] "T" "U" "V" "W" "X" "Y" "Z"

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

557 4 Faktoren (factors)

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

564 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)
```

565 ## [1] FI BU FI EI EI FI FI
 566 ## Levels: BU EI FI

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

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

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

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

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

574 ## [1] Fichte Buche Fichte Eiche Eiche Fichte Fichte
 575 ## Levels: Fichte Buche Eiche

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

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

579 ## [1] Fi Bu Fi Ei Ei Fi Fi
 580 ## Levels: Fi Bu Ei

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

```
af
```

```
583 ## [1] Fi Bu Fi Ei Ei Fi Fi
584 ## Levels: Fi Bu Ei
relevel(af, "Bu")
```

```
585 ## [1] Fi Bu Fi Ei Ei Fi Fi
586 ## Levels: Bu Fi Ei
```

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

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

```
589 ## [1] "Fi" "Bu" "Fi" "Ei" "Ei" "Fi" "Fi"
```

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

```
af
```

```
591 ## [1] Fi Bu Fi Ei Ei Fi Fi
592 ## Levels: Fi Bu Ei
```

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

```
593 ## [1] 1 2 1 3 3 1 1
```

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

596

597 Aufgabe 10: Faktoren

599 Verwenden Sie den Vektor `staedte` und erstellen Sie einen Vektor mit der Anordnung der `levels` in umgekehrter 600 alphabetischer Reihenfolge.

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

601 4.1 Das Paket **forcats**

602 Mit dem Paket aus **forcats** werden einige Operationen mit Faktoren erleichtert. Wir schauen uns hier 603 Funktion an, die es erleichtern:

- 604 1. Die Anordnung von Levels anzupassen.
- 605 2. Levels zusammenzufassen oder zu entfernen.
- 606 3. Labels zu ändern.

607 **4.1.1 Anpassen der Anordnung von Faktoren**608 Wir verwenden nochmals den **a** Vektor mit unterschiedlichen Baumarten:

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

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

```
factor(a)
```

610 ## [1] FI BU FI EI EI FI FI

611 ## Levels: BU EI FI

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

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

```
f1
```

613 ## [1] FI BU FI EI EI FI FI

614 ## Levels: FI BU EI

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

```
fct_rev(f1)
```

616 ## [1] FI BU FI EI EI FI FI

617 ## Levels: EI BU FI

618 nach der Häufigkeit zu sortieren oder

```
fct_infreq(f1)
```

619 ## [1] FI BU FI EI EI FI FI

620 ## Levels: FI EI BU

621 eine zufällige Sortierung.

```
fct_shuffle(f1)
```

622 ## [1] FI BU FI EI EI FI FI

623 ## Levels: EI FI BU

5 Spezielle Einträge

- 624 In vielen Fällen werden spezielle Einträge benötigt, bspw. bei
- 626 • fehlenden Einträgen NA,
 - 627 • leeren Einträgen NULL,
 - 628 • undefinierten Einträgen NaN (Not a Number) oder
 - 629 • unendlichen Zahlen (Inf).
- 630 Spezielle Einträge sind reservierte Namen. Sie können nicht überspeichert werden.

631 5.1 NA

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

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

```
635 ## chr [1:3] "foo" NA "foo"
na2 <- c(3, 6, NA)
str(na2)
```

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

637 Der logische Operatator zum Test auf fehlende Wert ist `is.na()`. Dieser kann genauso wie die bereits bekannten 638 logischen Operatoren bspw. zum Filtern verwendet werden. Die `na.omit()` Funktion entfernt NA aus dem 639 Datensatz.

```
is.na(na1)
## [1] FALSE TRUE FALSE
na.omit(na1)
```

```
641 ## [1] "foo" "foo"
642 ## attr(,"na.action")
643 ## [1] 2
644 ## attr(,"class")
645 ## [1] "omit"
```

646 Die bereits bekannten logischen Operationen ergeben NA, wenn Sie auf Daten angewendet werden, die NA 647 enthalten. Berechnungen mit NA ergeben ebenfalls NA. Bei der angewandten Programmierung müssen sie also 648 darauf achten, dass Ihre Daten frei von NA sind oder sie fangen die NA vorher ab.

```
na2 < 3
## [1] FALSE FALSE      NA
```

1 + NA

650 ## [1] NA
 651 Viele R Funktionen haben eingebaute Methoden zum Umgang mit NA. Die Funktion `mean()` bspw. ergibt
 652 (wie die meisten Funktionen) standardmäßig NA wenn sie auf Vektoren mit Datenlücken angewendet wird, es
 653 sei denn man stellt innerhalb der Funktion ein, dass Datenlücken entfernt werden sollen.

`mean(na2)`

654 ## [1] NA
 655
 656 mean(na2, na.rm = TRUE)
 655 ## [1] 4.5

656 5.2 NULL

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

661 5.3 Inf

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

10^309

664 ## [1] Inf
 665 2 * Inf
 665 ## [1] Inf
 666 1 + Inf
 666 ## [1] Inf
 667 3 / 0
 667 ## [1] Inf
 668 -3 / 0
 668 ## [1] -Inf
 669 3 / Inf
 669 ## [1] 0

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

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

672 ## [1] TRUE FALSE FALSE TRUE FALSE
is.finite(inf1)

673 ## [1] FALSE TRUE TRUE FALSE TRUE
inf1 < 3

674 ## [1] FALSE TRUE FALSE TRUE FALSE
```

675

Aufgabe 11: Vektoren mit speziellen Einträgen

678 Verwenden Sie den Vektor

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

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

681 Verwenden Sie den Vektor

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

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

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

687 Berechnen Sie den arithmetischen Mittelwert von `foo`.

6 data.frames oder Tabellen

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

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

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

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

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

722 Standard-Objekte zum Speichern wissenschaftlicher Daten.

723 6.1 Wichtige Funktionen zum Arbeiten mit `data.frames`

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

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

```
726 ##           ID anzahl_rehe    revier
727 ## 1 Kamera_1          132 Revier A
728 ## 2 Kamera_2          79 Revier A
```

729 Oder für die letzten 2 Beobachtungen.

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

```
730 ##           ID anzahl_rehe    revier
731 ## 14 Kamera_14         118 Revier C
732 ## 15 Kamera_15         107 Revier C
```

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

```
nrow(monitoring)
```

```
734 ## [1] 15
```

```
ncol(monitoring)
```

```
735 ## [1] 3
```

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

```
str(monitoring)
```

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

742

743 Aufgabe 12: `data.frame`

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

748 6.2 Zugreifen auf Elemente eines `data.frame`

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

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

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

```
757 ## [1] 91
```

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

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

```
761 ## [1] 91
```

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

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

```
764 ## [1] 132 79 129 91 138
```

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

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

```
767 ##   anzahl_rehe  revier
768 ## 1      132 Revier A
769 ## 2      79 Revier A
770 ## 3      129 Revier A
771 ## 4      91 Revier A
772 ## 5      138 Revier A
```

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

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

```
775 ##          ID anzahl_rehe  revier
776 ## 1 Kamera_1      132 Revier A
777 ## 2 Kamera_2      79 Revier A
778 ## 3 Kamera_3      129 Revier A
```

```
779 ## 4 Kamera_4          91 Revier A
780 ## 5 Kamera_5          138 Revier A
```

781

782 Aufgabe 13: Abfragen von Werten

784 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)
)
```

- 785 • Wählen Sie nur die ersten drei Zeilen aus und die erste und zweite Spalte aus.
 786 • Wählen Sie alle Zeilen und die erste und dritte Spalte aus.
 787 • Wählen Sie alle Spalten und die erste, dritte und vierte Zeile aus.

788

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

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

```
monitoring$anzahl_rehe
```

794 ## [1] 132 79 129 91 138 144 55 103 139 105 96 146 95 118 107

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

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

796 ## [1] 132 79 129 91 138 144 55 103 139 105 96 146 95 118 107

797 3. Alle Zeilen und die Spalte explizit angeben.

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

798 ## [1] 132 79 129 91 138 144 55 103 139 105 96 146 95 118 107

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

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

```
monitoring$anzahl_rehe > 100
```

```
806 ## [1] TRUE FALSE TRUE FALSE TRUE TRUE FALSE TRUE TRUE TRUE FALSE TRUE  
807 ## [13] FALSE TRUE TRUE
```

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

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

```
812 ##          ID anzahl_rehe    revier  
813 ## 1   Kamera_1           132 Revier A  
814 ## 3   Kamera_3           129 Revier A  
815 ## 5   Kamera_5           138 Revier A  
816 ## 6   Kamera_6           144 Revier B  
817 ## 8   Kamera_8           103 Revier B  
818 ## 9   Kamera_9           139 Revier B  
819 ## 10  Kamera_10          105 Revier B  
820 ## 12  Kamera_12          146 Revier C  
821 ## 14  Kamera_14          118 Revier C  
822 ## 15  Kamera_15          107 Revier C
```

823

824 Aufgabe 14: Abfragen von Werten 2

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

- ```
827 • Alle Spalten für Studierende die Forstwissenschaften studieren.
828 • Alle Spalten für Studierende die Chemie oder Physik studieren.
829 • Die Spalte fach und semester für Studierende die 22 oder älter sind.
```

## 830 7 Schreiben und lesen von Daten

### 831 7.1 Textdateien

832 Bis jetzt haben wir Daten immer in R erstellt, dies ist eine eher unnatürliche Situation. In den meisten Fällen  
 833 bekommen Sie Daten aus Felderhebungen, Sensoren oder sonstigen Quellen. Diese Daten müssen dann in R  
 834 eingelesen werden. Daten liegen meist in einer tabellarischen Form und als Textdatei vor<sup>6</sup>.

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

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

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

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

```
853 ## ID anzahl_rehe revier
854 ## 1 Kamera_1 132 Revier A
855 ## 2 Kamera_2 79 Revier A
856 ## 3 Kamera_3 129 Revier A
857 ## 4 Kamera_4 91 Revier A
858 ## 5 Kamera_5 138 Revier A
859 ## 6 Kamera_6 144 Revier B
```

860 Es gibt viele Varianten der Funktion `read.table()`. Beispielsweise hat die Funktion `read.csv()` bereits  
 861 die Argumente `sep = ','` und `header = TRUE` gesetzt. Die Funktion `read.csv2()` hat die in Deutschland  
 862 üblichen Argument `sep = ';'` und `dec = ','` gesetzt. Sie müssen natürlich darauf achten, dass sie die csv  
 863 Dateien mit den gleichen Spezifikationen einlesen, mit denen sie gespeichert wurde. Siehe dazu auch die  
 864 Hilfeseite von `read.table()`.

<sup>6</sup>Natürlich 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.

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

866

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

---

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

## 8 Erstellen von Abbildungen

872 Abbildungen sind ein elementarer Baustein statistischer Analysen und deshalb von Beginn an Teil von R. **R is  
873 a free software environment for statistical computing and graphics.** Es gibt unterschiedliche Systeme  
874 einen Plot zu erstellen. In diesem Kurs werden wir kurz *Base Plots* vorstellen und dann das Zusatzpaket  
875  
876 `ggplot2` vorstellen.

### 877 8.1 Base Plot

878 Die wichtigsten Grafiken für die einfache Datendarstellung sind schnell verfügbar. Etwas komplexere oder  
879 spezielle Grafiken erfordern mehr Programmieraufwand (folgt teilweise noch). Für viele komplexere Diagramme  
880 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  
881 sie sich die einfache Grafik Schnittstelle (`base plots`) als zweidimensionale Leinwand vor, auf die Sie durch  
882  
883 Code Ebene für Ebene Grafikelemente legen:

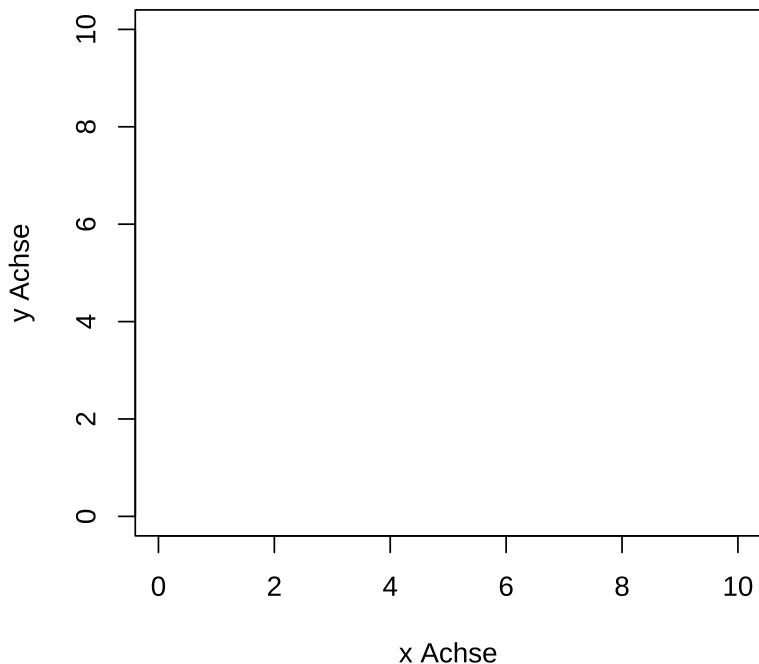
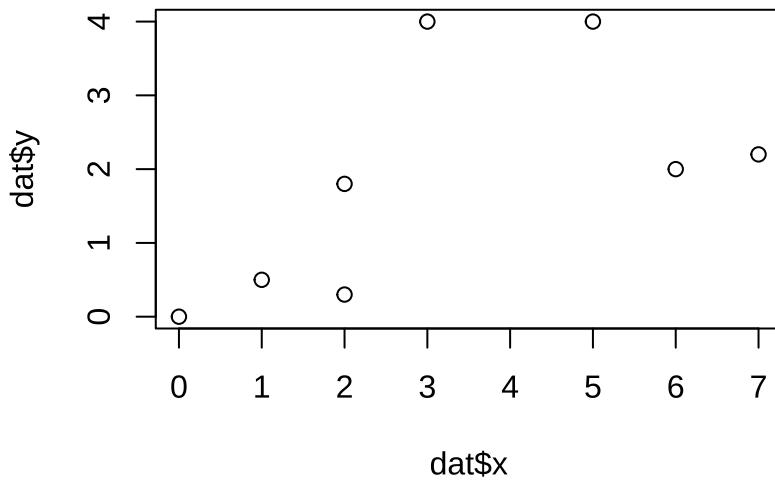


Abbildung 4: Beispiel einer leeren Grafikschnittstelle.

884 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)
)

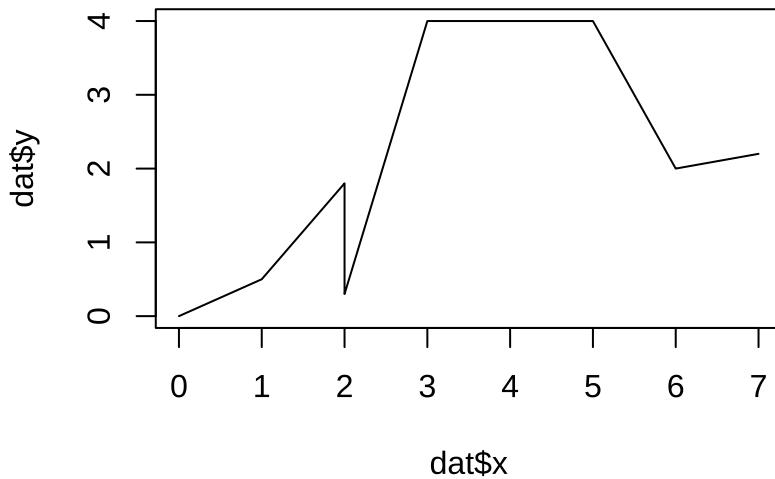
plot(datx, daty, type ="p")
```



885

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

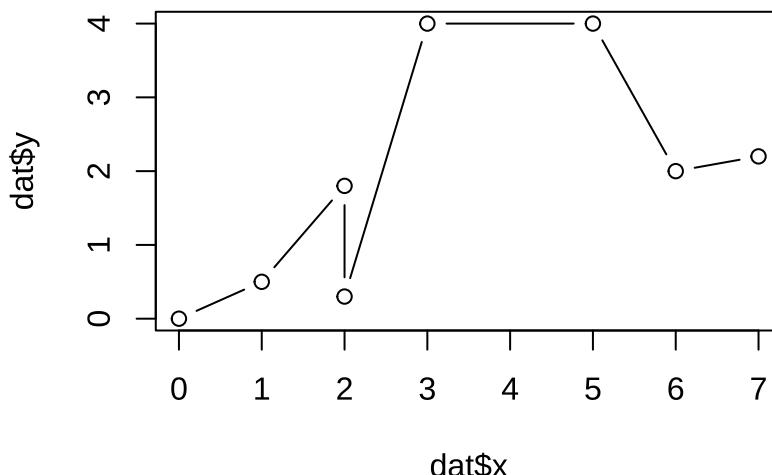
```
plot(datx, daty, type = "l")
```



888

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

```
plot(datx, daty, type = "b")
```



890

891 darstellen.

892

893 **Aufgabe 16: Base Plot 1**

894

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

897

898 Sie können entweder eine Grafik mit einem Befehl erzeugen (High-Level) oder die einzelnen Ebenen nacheinander erzeugen (Low-Level). Sie können jeder Ebene durch zusätzliche Befehle innerhalb des Funktionsaufrufs  
900 Elemente hinzufügen. Mit jeder neuen Ebene können Sie die vorigen Ebenen jedoch nicht mehr verändern.  
901 Die wichtigsten Argumente der `plot` Funktion sind:

902

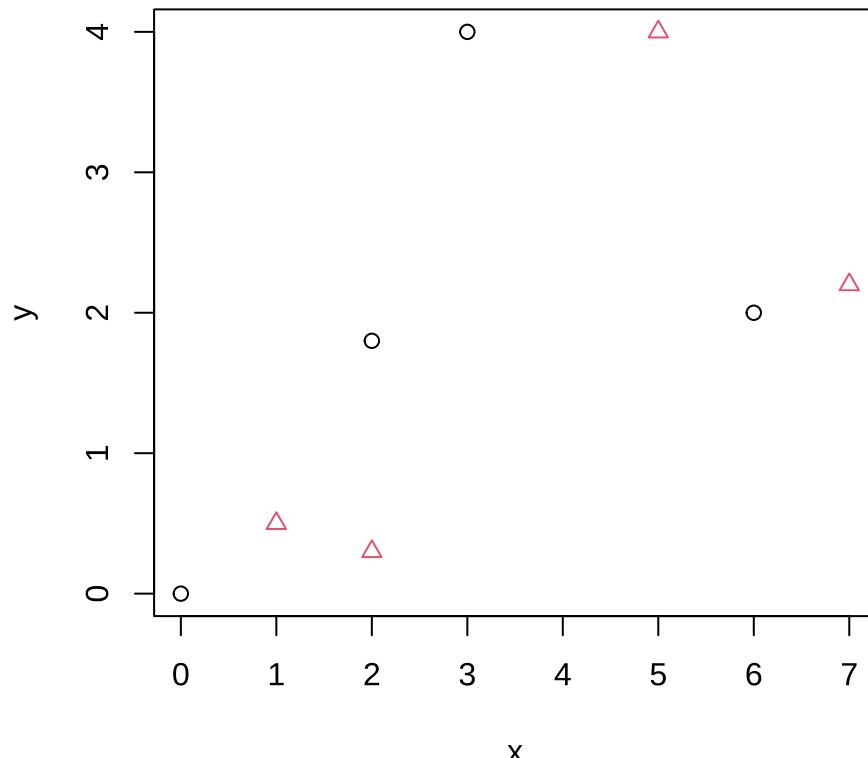
- `type` - Diagrammtyp
- `col` - Farbe
- `main` - Titel
- `sub` - Untertitel
- `pch` - Punktsymbol
- `lty` - Linientyp
- `lwd` - Linienstärke
- `xlab` bzw. `ylab` - Achsenbeschriftungen
- `xlim`, `ylim` - Grenzen der Achsenanschnitte
- `axes` - Sollen die Achsen eingezeichnet werden? Oder leer gelassen werden, um sie nachträglich als low-level Ebene einzuziehen?
- `ann` - Achsenbeschriftung kann ganz weggelassen werden.

914

Sehen Sie sich die Hilfeseiten `?plot.default()` oder `?par()` an für weitere Informationen. Dort finden Sie  
915 auch eine vollständige Liste der Befehle. Einige Argumente können als Vektor übergeben werden. Hier z. B.  
916 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")
```



917

918

---

**Aufgabe 17: Anpassen von Plots**


---

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

- 920
- 921 • Beschriften Sie die x- und y-Achse sinnvoll.
  - 922 • Fügen Sie eine Überschrift hinzu.
  - 923 • Wählen Sie ein anderes Symbol.
  - 924 • Stellen Sie die Symbole in rot dar.

925

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

927 Die wichtigsten Funktionen sind

- 928
- 929 • `points()` - Fügt Punkte ein
  - 930 • `lines()` - Fügt Linien ein

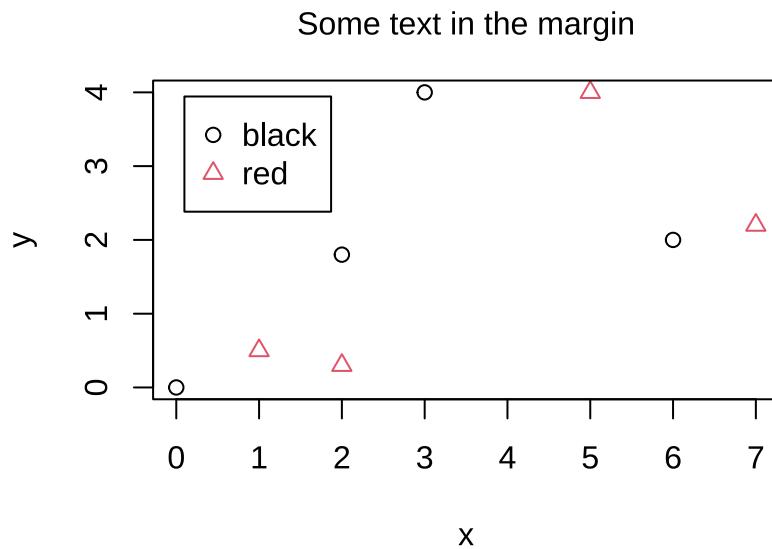
- 931 • `text()` - Fügt Text ein  
 932 • `mtext` - Fügt Text in den Rahmen (`margin`) ein  
 933 • `legend()` - Fügt eine Legende ein  
 934 • `abline()` - Fügt eine Gerade ein  
 935 • `curve()` - Fügt eine mathematische Funktion ein  
 936 • `arrows()` - Fügt Pfeile ein  
 937 • `grid()` - Fügt Hilfslinien ein

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

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

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

plot(x = dat$x, y = dat$y, col = dat$col, pch = dat$col, xlab = "x", ylab = "y")
legend(x = "topleft", inset = 0.05, legend = c("black", "red"),
 col = c(1, 2), pch = c(1, 2))
mtext(side = 3, line = 1, "Some text in the margin")
```



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

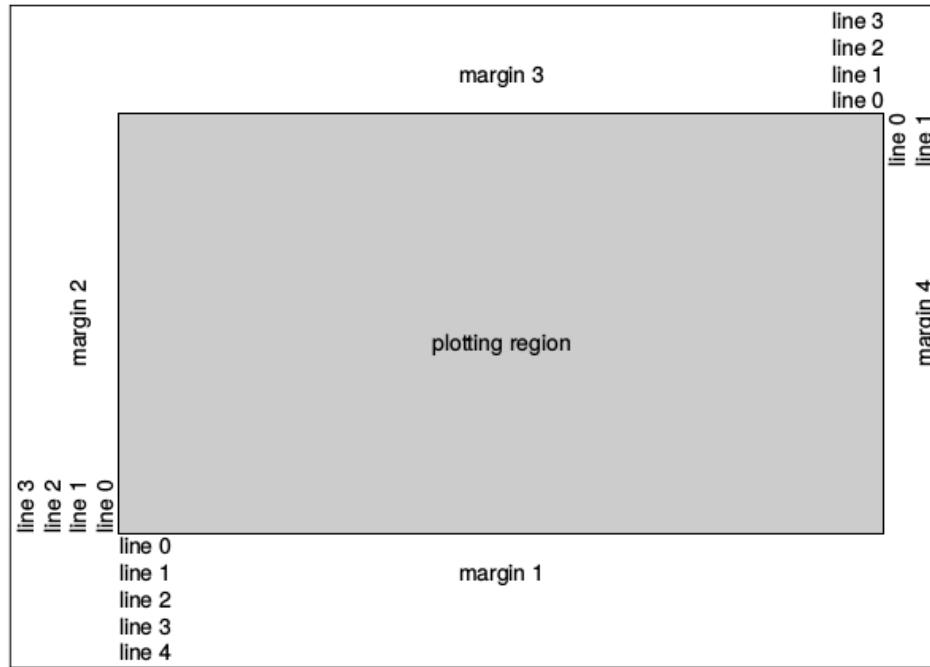
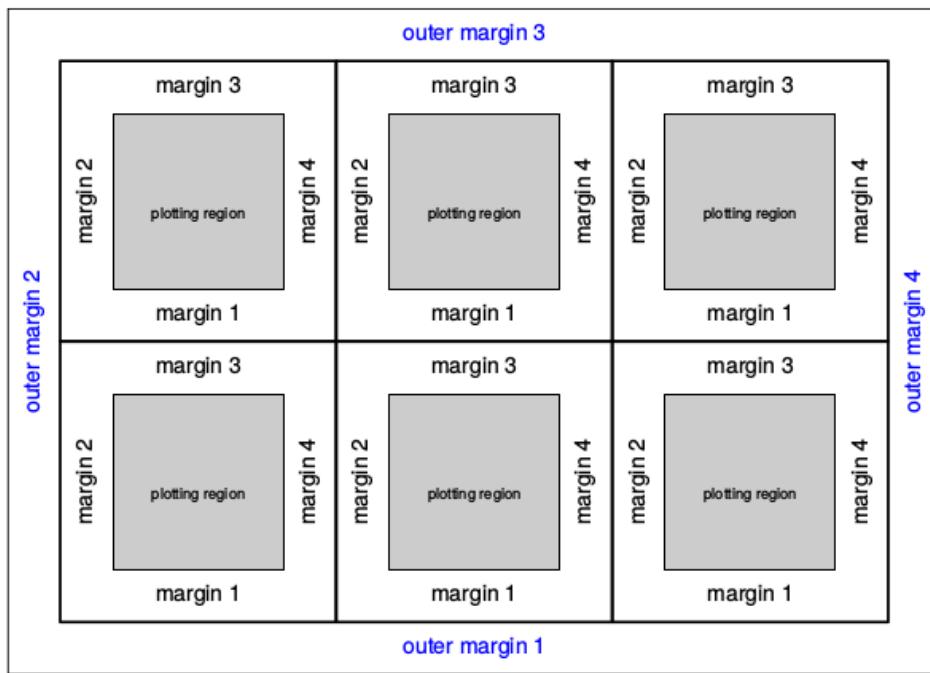


Abbildung 5: Grafikregionen eines base plots in R.

Abbildung 6: Schematischer Aufbau mehrerer Diagramme in einem plot am Beispiel einer  $3 \times 2$  Grafik.

949 **8.1.1 Mehrere Panels**

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

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

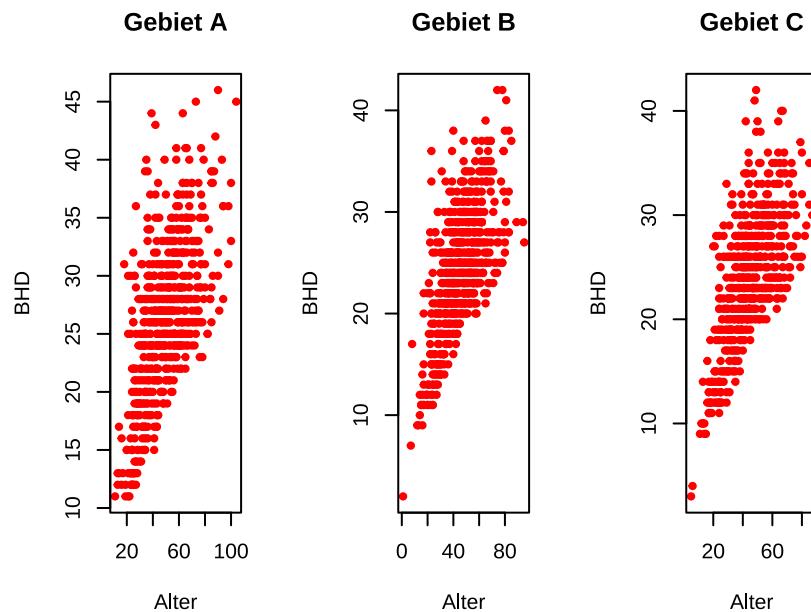
953 Teilt den Plot in eine Zeile und drei Zeilen (= drei Plots nebeneinander).

```
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")
```



954

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

957 **8.1.2 Speichern von Abbildungen**

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

- 962     • `pdf()` oder  
 963     • `postscript()`.

964 Beispiele für Rastergrafiken sind

- 965     • `png()`,  
 966     • `bmp()` oder  
 967     • `jpeg()`.

968 Die Grafikschnittstelle ist dann Ihre “Leinwand”. Mit dem Befehl `dev.off()` trennen Sie die Verbindung zur  
 969 Schnittstelle wieder. Ihre “Leinwand” wird also wieder geschlossen. So lange die Schnittstelle geöffnet ist  
 970 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
```

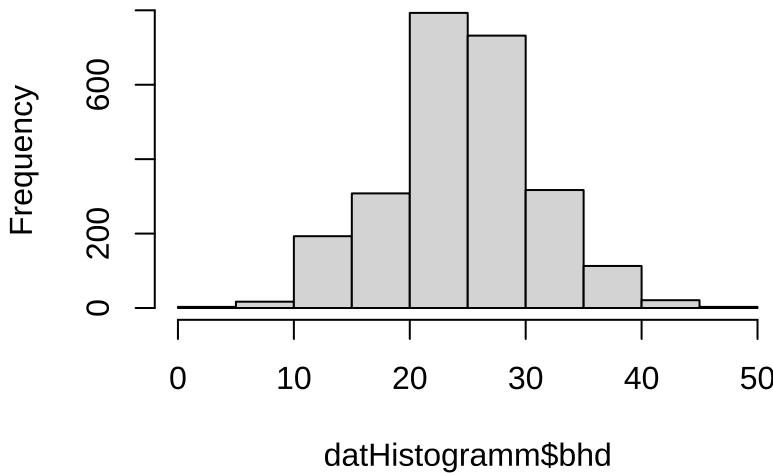
971 *Achtung*, wenn Sie die Funktion `dev.off()` nicht aufrufen, werden alle nachfolgenden Plots in die gleiche  
 972 Datei geschrieben. Falls Sie nach einem Versuch einen Plot zu speichern plötzlich keine weiteren Plots mehr  
 973 sehen, führen Sie einige Mal die Funktion `dev.off()` aus.

## 974 8.2 Histogramme

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

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

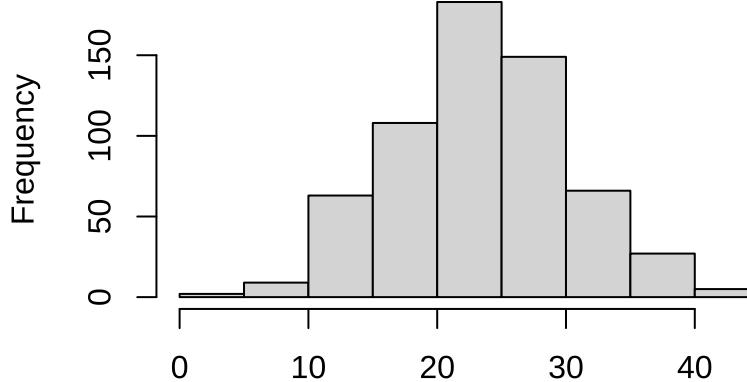
### Histogram of datHistogramm\$bhd



982

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

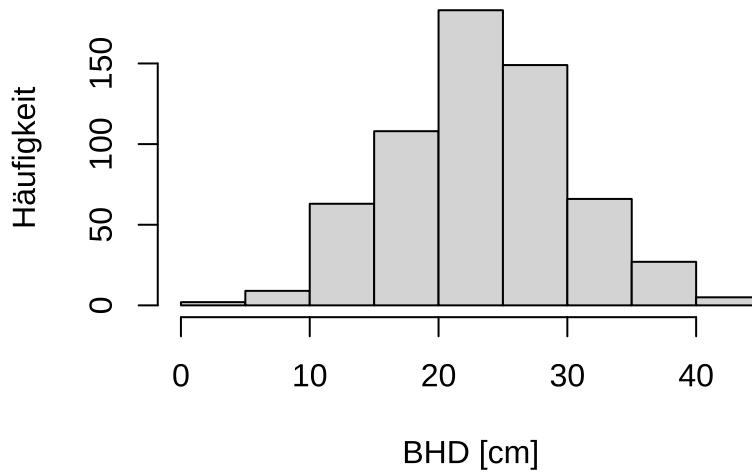
### Histogram of datHistogramm\$bhd[datHistogramm\$art == "EI"]



983

```
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

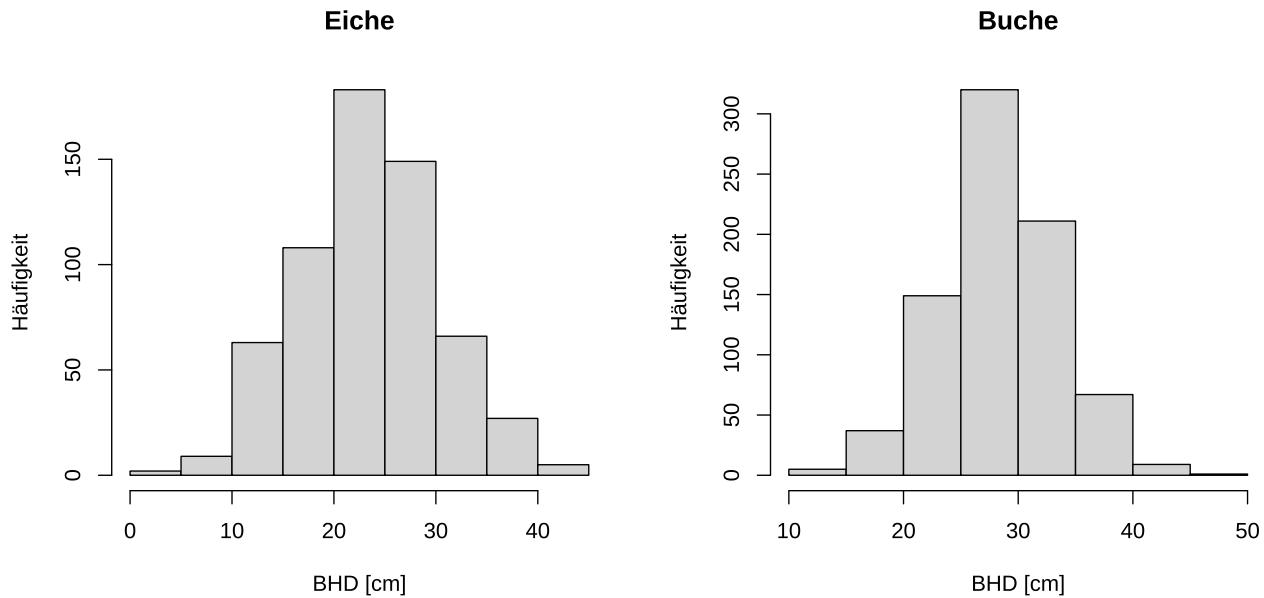


984

985 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")
```

986



987

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

### 8.3 Boxplots

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

Boxplots bestehen aus drei Komponenten:

1. Eine *Box*, die den Bereich zwischen 0.25 und 0.75 Percentil abdeckt, diese Distanz wird auch die IQR (Interquartile Range), bezeichnet. Zusätzlich wird die Box durch den Median (als dicke horizontale Linie) unterteilt.
2. Einzelne Punkte Ausreißer. Als Ausreißer werden Punkte bezeichnet, die  $> 1.5 \text{IQR}$  vom unteren oder oberen Ende der Box entfernt sind.
3. Eine senkrechte Linie von jeder Seite der Box bis zum letzten „Nicht-Ausreißer-Punkt“. Also der letzte Punkt, der  $> 1.5 \text{IQR}$  aber nicht  $> 0.75$  bzw.  $< 0.25$  Percentil ist. Diese Linie wird auch als *Whisker* bezeichnet.

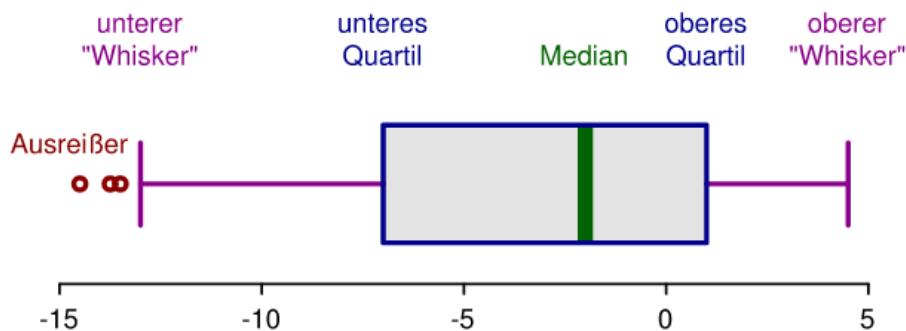
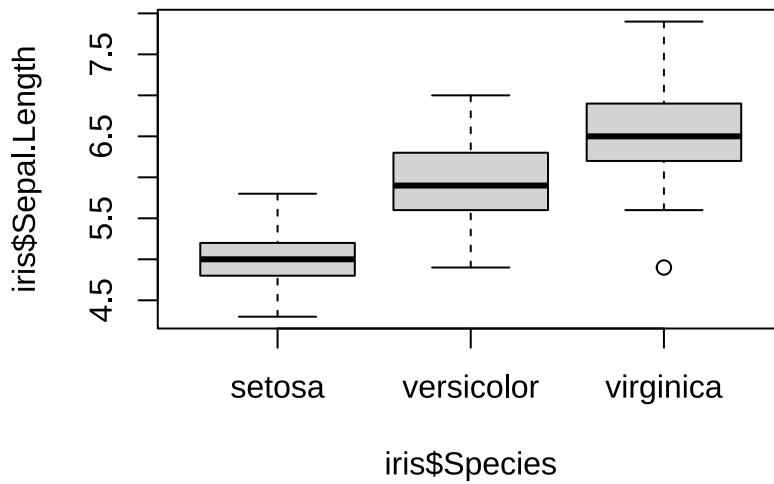


Abbildung 7: Schematische Darstellung eines Boxplots (Quelle: Von RobSeb - Eigenes Werk, CC BY-SA 3.0, <https://commons.wikimedia.org/w/index.php?curid=14697172>).

Mit R kann mit der Funktion `boxplot()` ein Boxplot erstellt werden. Diese Funktion kann in zwei unterschiedlichen Ausprägungen verwendet werden.

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

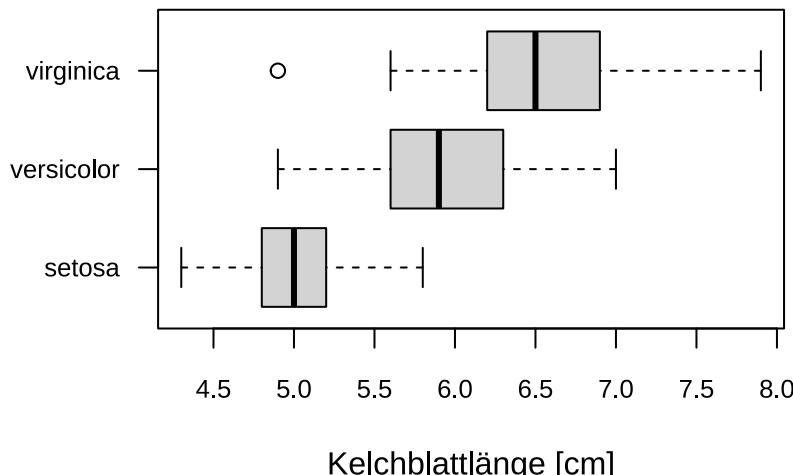
```
boxplot(iris$Sepal.Length ~ iris$Species)
```



1008

1009 Etwas eleganter ist es wenn wir das Argument `data` verwenden und den Plot etwas anpassen. Diese Schreib-  
1010 weise funktioniert für alle base plots.

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



1011

1012

### 1013 Aufgabe 18: Boxplots

1014

- Lesen Sie erneut den Datensatz `daten/bhd_1.txt` ein und speichern Sie diesen in die Variable (`dat_bhd`).
  - Wie viele BHD-Messungen gibt es für jedes Gebiet?
  - Erstellen Sie für jedes Gebiet einen Plot
- 1018 Erstellen Sie einen Plot mit 3 Subplots, jeweils mit einem Boxplot für die ersten drei Studiengebiete, in dem  
1019 der BHD für jede Baumart dargestellt wird.

## 1020 8.4 ggplot2: Eine Alternative für Abbildungen

1021 ggplot2 ist ein alternatives Plotting-System in R. Sie können mit ggplot2 also grundsätzlich Abbildungen  
 1022 mit dem selben Inhalt erstellen, wie mit Base Plots. Die Syntax und die optische Darstellung unterscheiden  
 1023 sich jedoch grundsätzlich. ggplot2 basiert auf den *grammar of graphics* von Leland Wilkinson. Die Idee  
 1024 ist, alle nötigen Informationen der Abbildung miteinander zu verknüpfen. ggplot2 ist also diametral zu  
 1025 Base Plots. Während Sie in base plot alle Elemente selbst definieren, ist die Idee von ggplot2, dass Sie  
 1026 nur die Daten und den Diagrammtypen übergeben und die Funktion selbst eine schöne Abbildung erstellt.  
 1027 Selbstverständlich können Sie aber auch in ggplot2 viele Einstellungen vornehmen. Im base plot sehen  
 1028 Abbildungen zunächst sehr karg und etwas unfertig aus. Sie müssen viele Einstellungen vornehmen, um eine  
 1029 publizierfähige Grafik zu produzieren. In ggplot2 sollen auch die einfachste Abbildungen schon ästhetisch  
 1030 sein. Mit diesen gebündelten Informationen kann ggplot2 die Abbildung automatisch verschönern. So  
 1031 werden bspw. die Legenden automatisch erzeugt und auch die Formatierungen automatisch an die Datenlage  
 1032 angepasst. ggplot2 nimmt der\*dem Entwickler\*in also Arbeit ab. Dadurch sind die Abbildungen schon ohne  
 1033 viel Nacharbeit schick. Nachteil ist, dass der\*dem Entwickler\*in weniger Möglichkeiten zur Einstellung zur  
 1034 Verfügung stehen und nuterspezifische Sonderwünsche somit schwerer umsetzbar sind. Sehen Sie sich das  
 1035 Cheatsheet zu ggplot2 an. Es ist in RStudio unter [Help > Cheatsheets](#) zu finden.

1036 Bei ggplot2 sind Anweisungen zu den Daten und Anweisungen zur Darstellung voneinander getrennt. Die  
 1037 Daten werden in den Ästhetikbefehl übergeben und dort klassifiziert. Dann folgen die Darstellungsanweisungen.  
 1038 Ähnlich wie bei Base Plots, werden die Grafikelemente ebenenweise nacheinander programmiert, jedoch mit  
 1039 einem + verbunden. Und hier liegt der wesentliche Unterschied zu Base Plots. Durch die + werden die Ebenen  
 1040 zu einem Befehl verbunden und damit gleichzeitig erstellt.

1041 Die Erweiterung wird zunächst geladen<sup>7</sup>. Wir laden außerdem den Datensatz **iris**. Der Datensatz ist in R  
 1042 fest integriert. Siehe `?iris` für mehr Informationen.

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

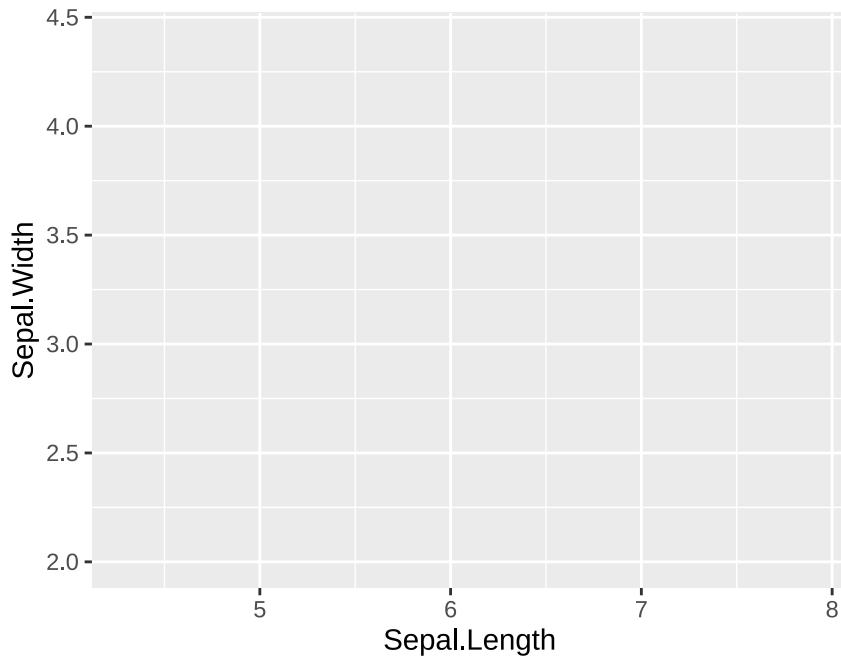
```
1043 ## Sepal.Length Sepal.Width Petal.Length Petal.Width Species
1044 ## 1 5.1 3.5 1.4 0.2 setosa
1045 ## 2 4.9 3.0 1.4 0.2 setosa
1046 ## 3 4.7 3.2 1.3 0.2 setosa
1047 ## 4 4.6 3.1 1.5 0.2 setosa
1048 ## 5 5.0 3.6 1.4 0.2 setosa
1049 ## 6 5.4 3.9 1.7 0.4 setosa
```

1050 Die Ästhetik wird bspw. folgendermaßen definiert.

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

---

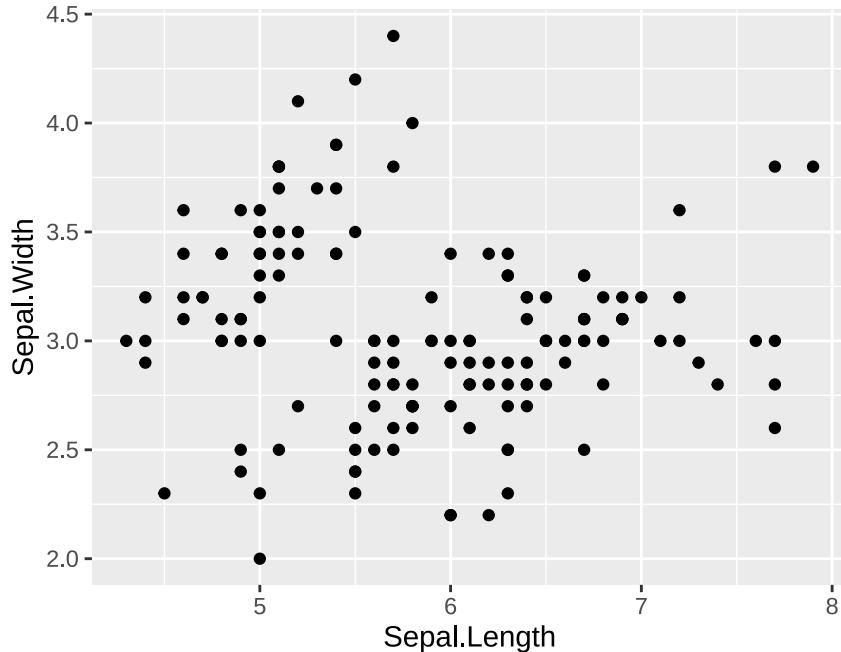
<sup>7</sup>Wir haben bis jetzt immer nur mit base R gearbeitet. D.h. wir haben nur Funktionen verwendet, die R bereits zur Verfügung stellt. Eine der großen Stärken von R sind die Erweiterungen (oder auch Pakete genannt). ggplot2 ist so eine Erweiterung, die einmal mit `install.packages("ggplot2")` 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.



1051

1052 Dieser Befehl zeichnet noch keine Daten. Die Daten werden lediglich herangezogen, um einen leeren Plot für  
1053 die Daten zu erstellen. In dem Beispiel wird die Variable `Sepal.Length` aus dem `data.frame iris` als x und  
1054 `Sepal.Width` als y Variable definiert. Diese Informationen stehen den folgenden Layern nun zur Verfügung,  
1055 sodass nach den + nur noch x und y verwendet werden müssen. Um bspw. einen Scatterplot zu erstellen  
1056 wird ein `geom_point()` Layer hinzugefügt. x und y werden automatisch an `geom_point` übergeben. Weitere  
1057 Einstellung sind in diesem Beispiel nicht notwendig, wären jedoch möglich. Siehe `?geom_point()`.

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



1058

1059

---

1060 **Aufgabe 19: Abbildungen mit ggplot2**

---

1061

1062 Verwenden Sie die Daten aus Aufgabe 16 und erstellen Sie einen Scatterplot mit ggplot2 wie in Aufgabe 16.

1063

1064 Wir haben mit der Funktion `geom_point()` demm Plot eine Punktgeometrie hinzugefügt. Es gibt noch viele  
1065 weitere Geometrien. Die wichtigsten sind:

- 
- 1066 •
- `geom_line()`
- für eine Linie.
- 
- 1067 •
- `geom_histogram()`
- um ein Histogramm zu erstellen.
- 
- 1068 •
- `geom_boxplot()`
- um einen Boxplot zu erstellen.
- 
- 1069 •
- `geom_bar()`
- um ein Säulendiagramm zu erstellen.

1070 Welche Geometrie die richtige ist, richtet nach dem Typ der darzustellenden Variablen. Beispielsweise  
1071 bietet sich `geom_point()` an, wenn man zwei kontinuierliche Variable darstellen möchte. Wenn man hin-  
1072 gegen die Verteilung von einer kontinuirlchen Variable darstellen möchte, dann bietet sich ein Histogramm  
1073 (`geom_histogram()`) oder auch eine geschätzte Dichte (z.B. `geom_density()`) an.

1074

---

1075 **Aufgabe 20: Abbildungen mit ggplot2**

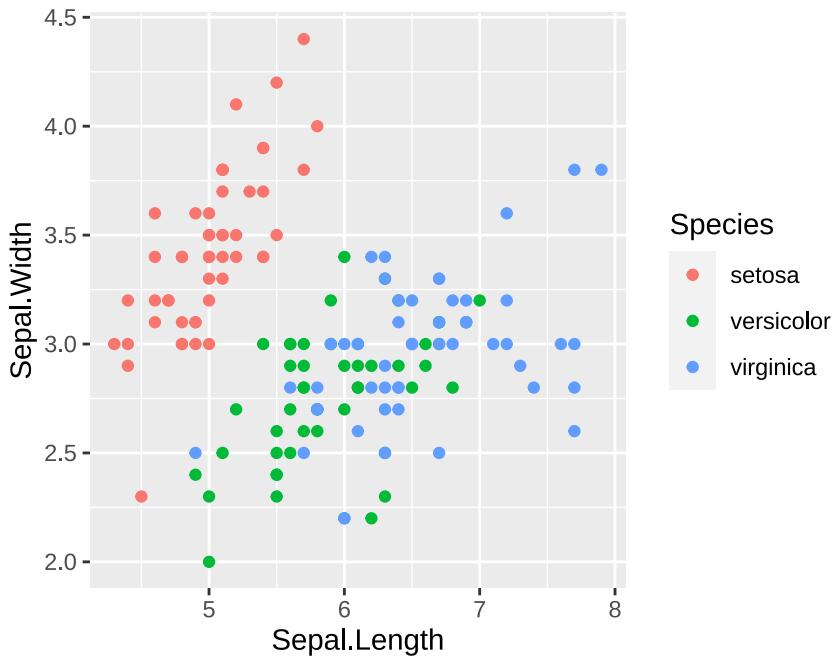
---

10761077 Verwenden Sie die den Iris Datensatz und erstellen Sie mit ggplot2 einen Plot der die Verteilung der Länge  
1078 der Kelchblätter zeigt (Spalte `Sepal.Length`).

1079

1080 Eine der Stärken von ggplot2 ist, dass man den Wert unterschiedlicher Variable auf unterschiedlichen  
1081 Komponenten des Plots abbilden kann. Wir haben bis jetzt ein bzw. zwei Variable auf der x- und y-Achse  
1082 abgebildet. Wir können aber ein weitere Variablen verwenden um das Aussehen des Plots zu beeinflussen.  
1083 Beispielsweise können wir die Farbe der Punkte (für `geom_point()`) mit dem Argument `col` beeinflussen.

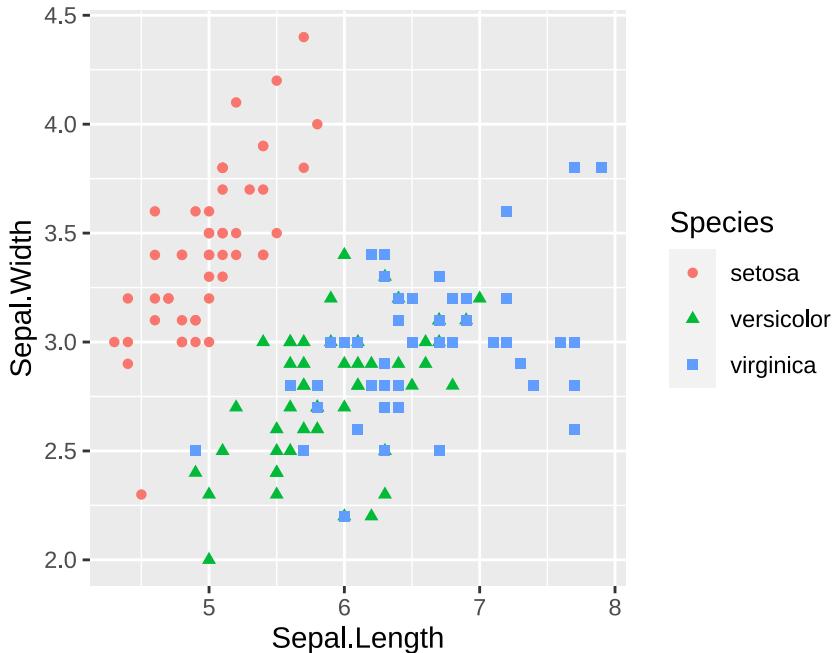
```
ggplot(iris, aes(x = Sepal.Length, y = Sepal.Width, col = Species)) +
 geom_point()
```



1084

1085 Somit bekommt jede Irisart eine eigene Farbe<sup>8</sup>. Gleichesmaßen können wir die Punktart (**shape**), die  
1086 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()
```

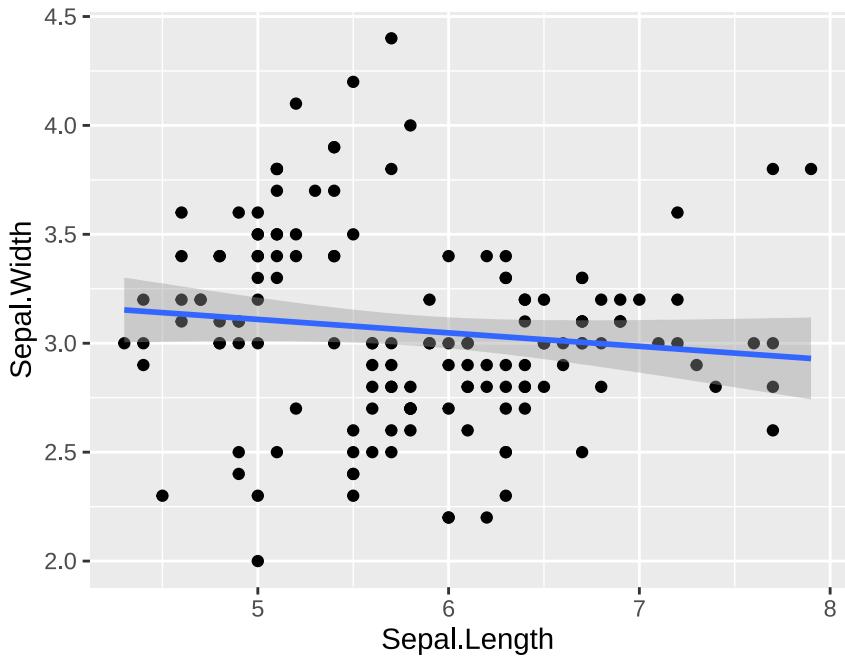


1087

1088 In dem Plot ist die Information zu der Art redundant (einmal als Farbe und einmal Symbolart).  
1089 Ein weitere sehr nützliche Geometrie ist **geom\_smooth()**, die es erlaubt eine Trendlinie hinzuzufügen.

<sup>8</sup>Natürlich könnte man auch die Farbe anpassen.

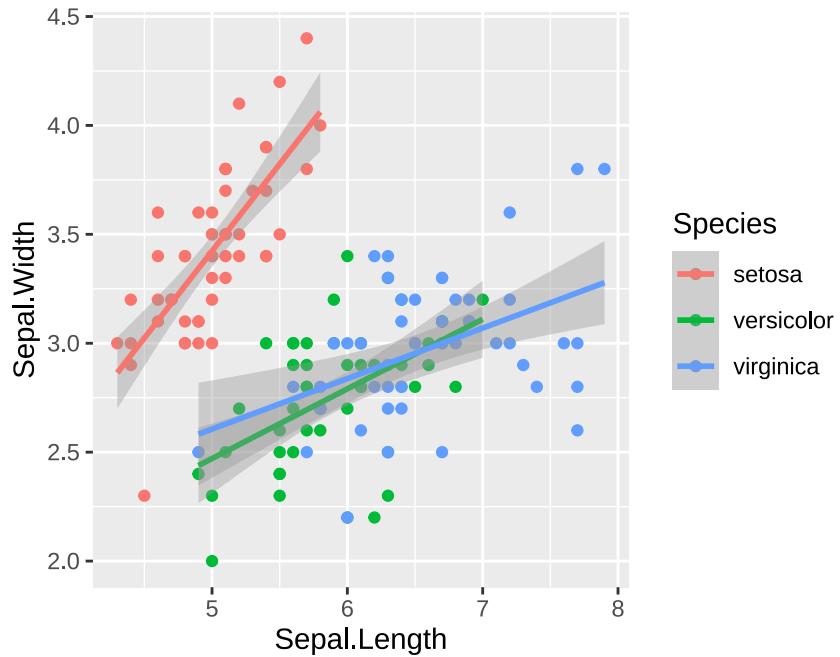
```
ggplot(iris, aes(x = Sepal.Length, y = Sepal.Width)) +
 geom_point() + geom_smooth(method = "lm")
```



1090

1091 Mit `method = "lm"` wird festgelegt, dass die Trendlinie gerade sein soll (es wird eine lineare Einfachregression  
 1092 angepasst). Wenn wird wieder eine gruppierende Variable einführen (z.B. die Beobachtungen nach Art auf  
 1093 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")
```



1094

1095

1096 **Aufgabe 21: Anpassen von Plots**

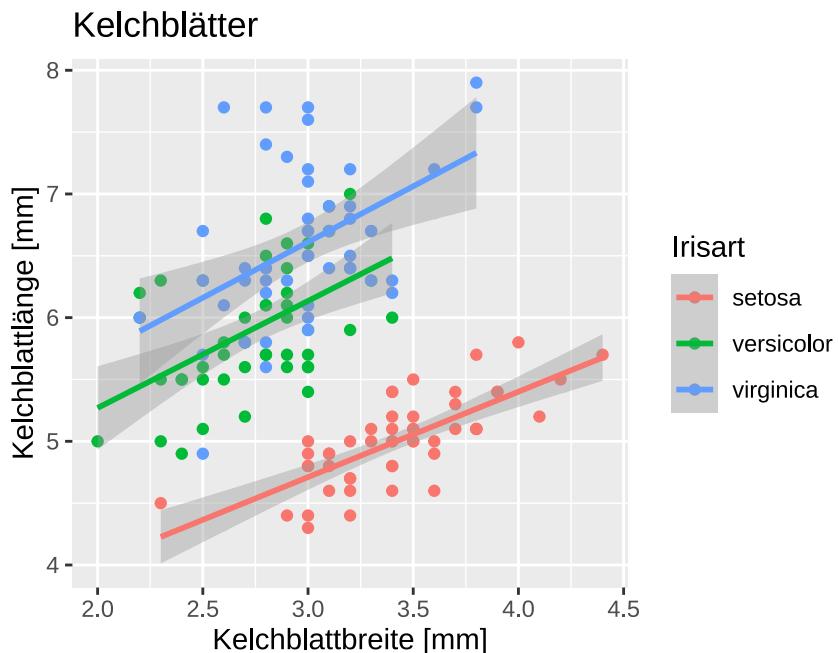
- 1098 Lesen Sie den Datensatz `data/bhd_1.txt` ein und erstellen Sie einen Boxplot für die Verteilung des BHDs  
 1099 für jede Baumart. In einem zweiten Schritt verwenden Sie erst `col = gebiet` und dann `fill = gebiet`.  
 1100 Welchen Unterschied stellen Sie fest?

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

1101

- 1102 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")
```



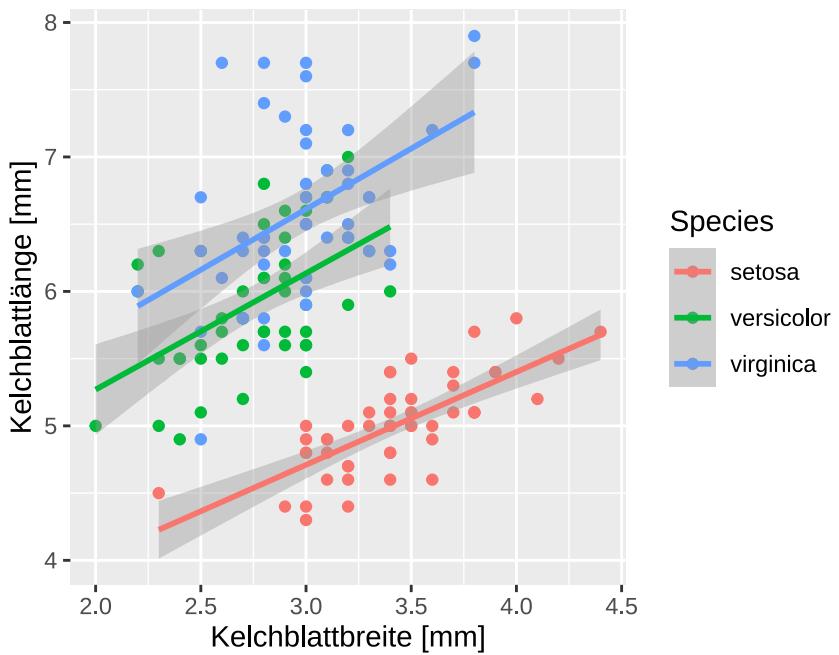
1103

- 1104 Statt einen langen Befehl zu tippen, kann ein `ggplot()` auch zwischengespeichert und wieder aufgerufen bzw.  
 1105 angepasst werden. Das ist vor allem sinnvoll, wenn mehrere Abbildungen auf dem selben Zwischenergebnis  
 1106 aufbauen sollen.

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

- 1107 Wir können jetzt mit `p1` weiter arbeiten und beispielsweise eine Beschriftung hinzufügen.

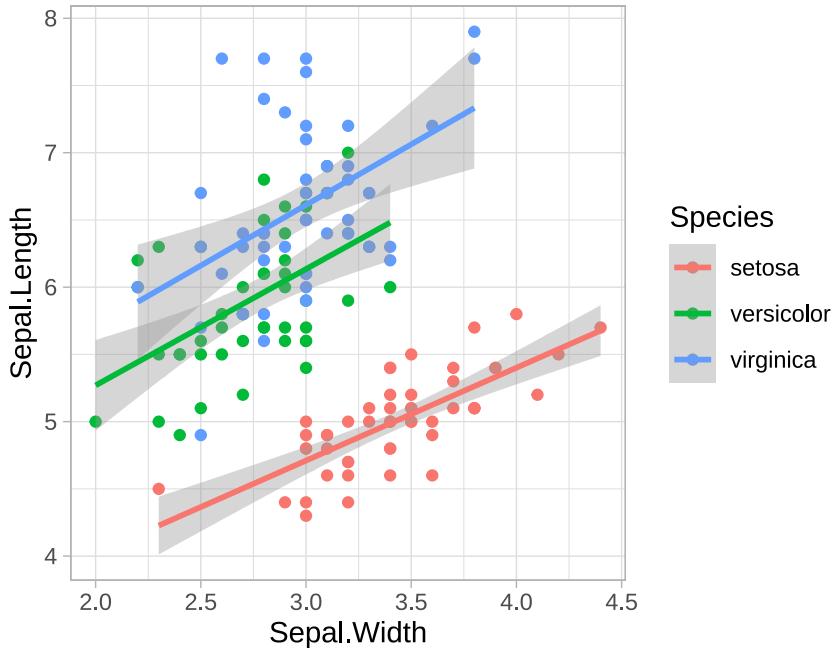
```
p1 + labs(x = "Kelchblattbreite [mm]", y = "Kelchblattlänge [mm]")
```



1108

1109 Oder auch den ganzen Plot anpassen. Dafür gibt es *themes*. Es gibt eine Reihe von vorgefertigten *themes*  
1110 oder man kann diese auch selber erstellen (das ist aber nicht Teil dieses Kurses).

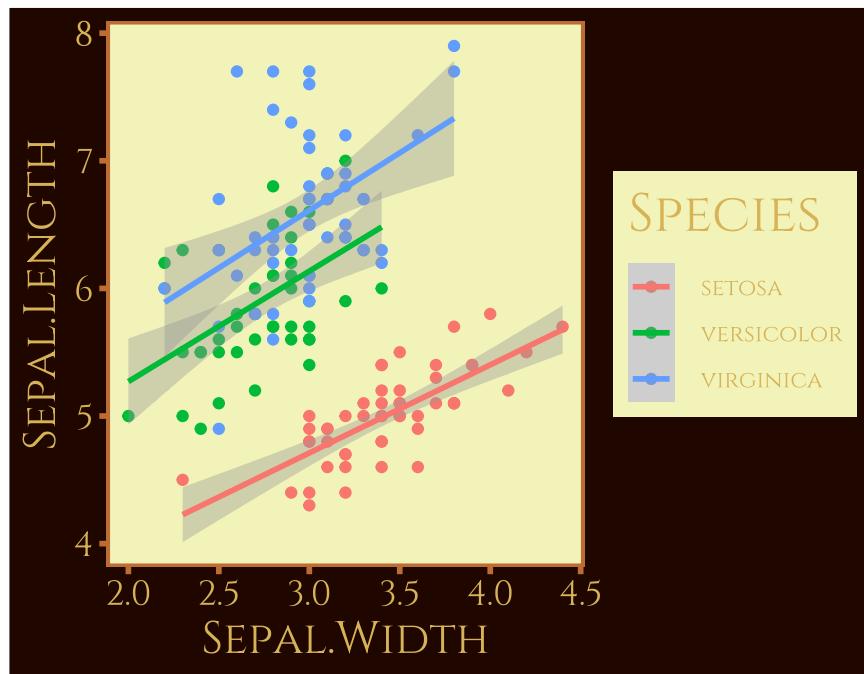
```
p1 + theme_light()
```



1111

1112 Weitere *themes* sind: `theme_bw()`, `theme_linedraw()` oder `theme_dark()`. Es gibt extra Pakete die viele  
1113 zusätzliche weitere *themes* anbieten. Dazu gehört z.B. das Paket `ggthemes` oder `ThemePark`. Während  
1114 `ggthemes` hauptsächlich durchdachte Grafikkonzepte liefert, sind die Themes aus `ThemePark` eher Popkultur  
1115 und nicht 100 %ig ernst gemeint.

```
p1 + theme_gamethrones()
```

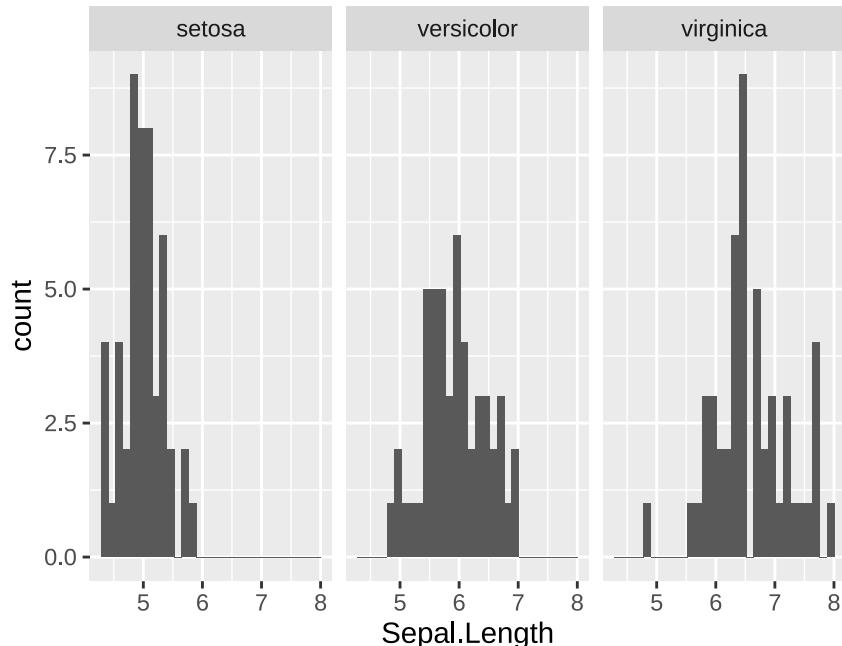


1116

#### 8.4.1 Multipanel Abbildungen

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

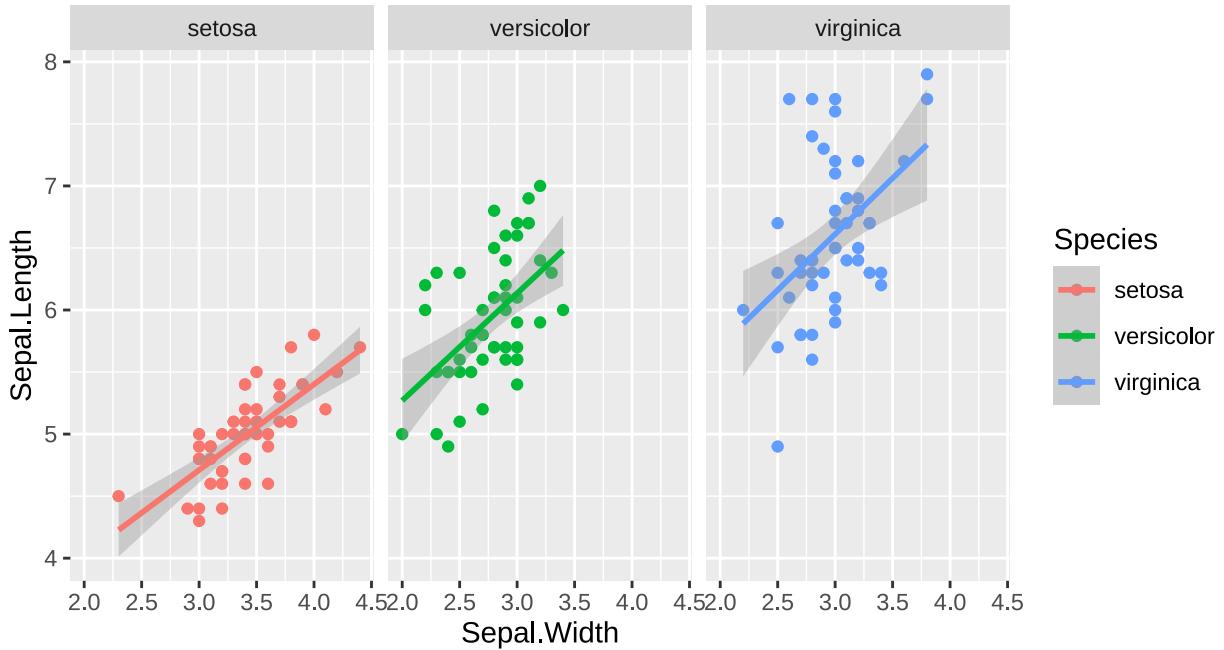
```
ggplot(iris, aes(Sepal.Length)) + geom_histogram() +
 facet_grid(~ Species)
```



1121

1122 Die Funktion `facet_grid()` erzeugt einen *Grid* und beschriftet die Grid-Zeilen und -Spalten, während  
 1123 `facet_wrap()` für jedes Panel (Diagramm) eine eigene Überschrift erzeugt. Beim Arrangieren der Diagramme  
 1124 wird der Vorteil von `ggplot2` gegenüber `base plot` besonders deutlich. Während es im `base plot` System  
 1125 mühsam ist, die Abbildungen zu arrangieren, übernimmt `ggplot2` das Arrangement automatisch und fügt  
 1126 sogar eine automatisch Legende hinzu. Die Achsenabschnitte werden so angepasst, dass die Daten vergleichbar  
 1127 sind.

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



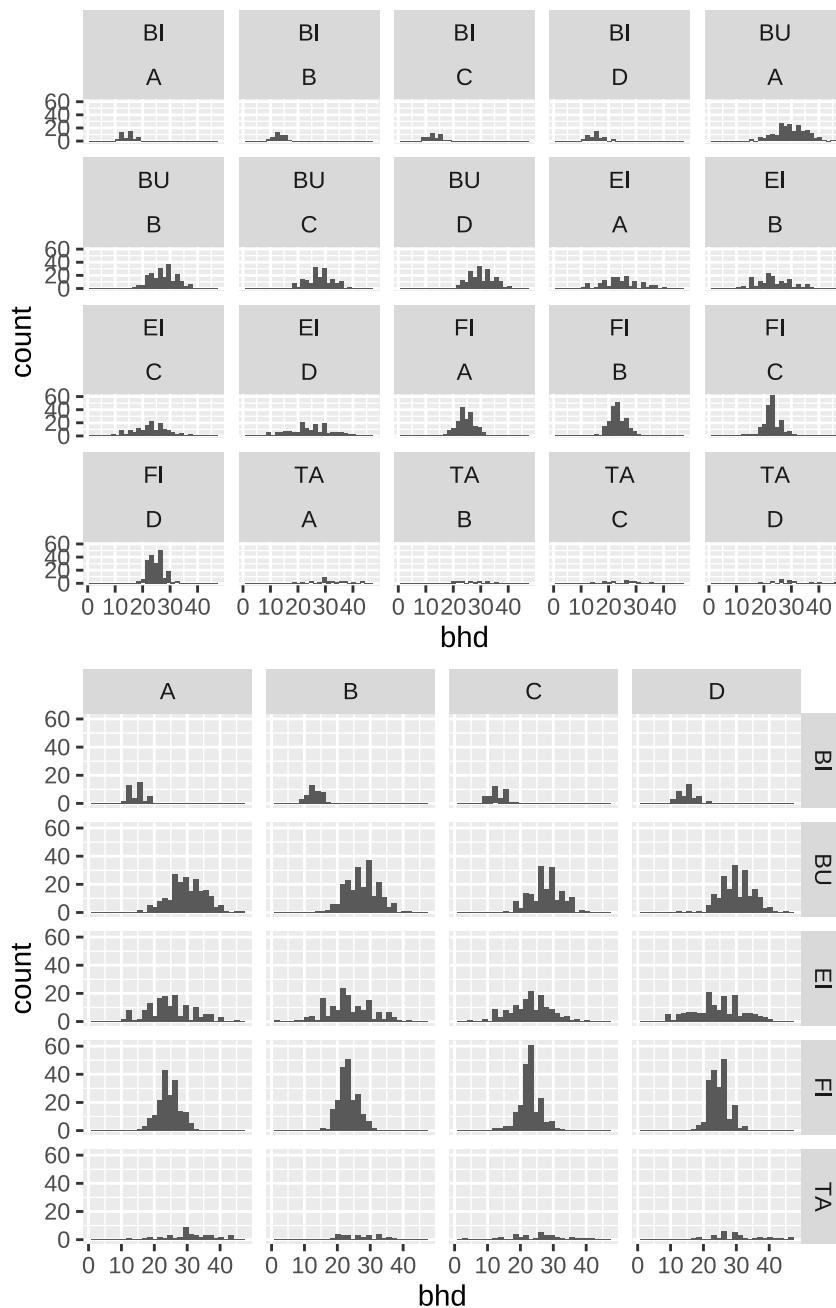
1128

1129

### 1130 Aufgabe 22: Multipanel Abbildungen

---

1132 Lesen Sie erneut den Datensatz `daten/bhd_1.txt` ein und speichern Sie diesen in die Variable (`dat_bhd`).  
 1133 Erstellen Sie für jede Art und Gebiet ein Histogramm. Welche Unterschiede können Sie feststellen, wenn Sie  
 1134 `facet_grid()` oder `facet_wrap()` verwenden?



#### 1137 8.4.2 Plots kombinieren

1138 Es gibt Situationen, in denen **unterschiedliche** Plots miteinander kombiniert werden müssen. Im vorherigen  
 1139 Abschnitt wurde dies immer anhand einer gruppierenden Kovariate gemacht. Aber es gibt auch Situationen, in  
 1140 denen das nicht möglich ist. Beispielsweise wenn ein Histogramm und ein Scatterplot vom gleichen Datensatz  
 1141 zusammengefasst werden sollen. Dafür bietet sich das Paket **patchwork** an<sup>9</sup>.

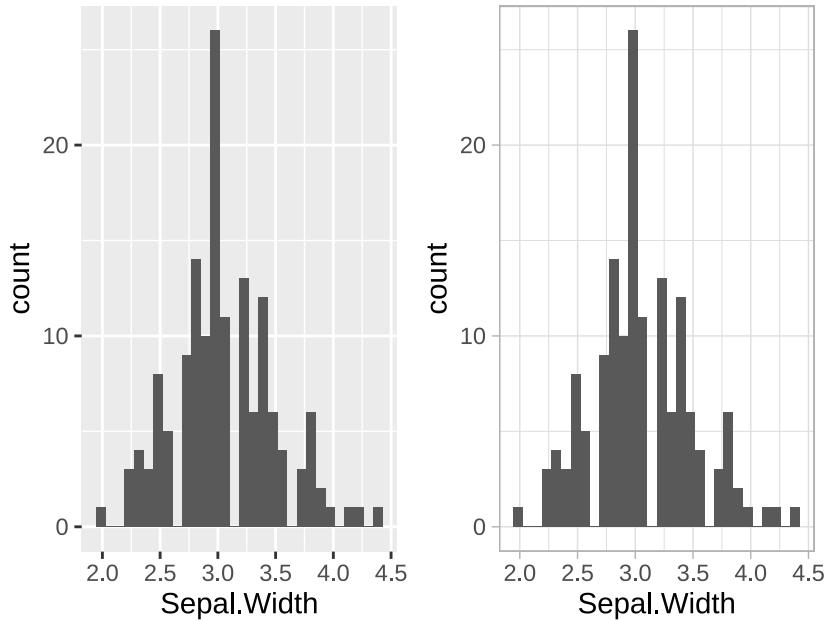
1142 Als erstes können wir zwei (oder natürlich auch mehrere Plots) erstellen. Hier unterscheiden sich die Plots  
 1143 lediglich durch das Aussehen.

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

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

- 1144 Dann müssen können wir diese Plots ebenfalls mit `+` zusammenfügen.

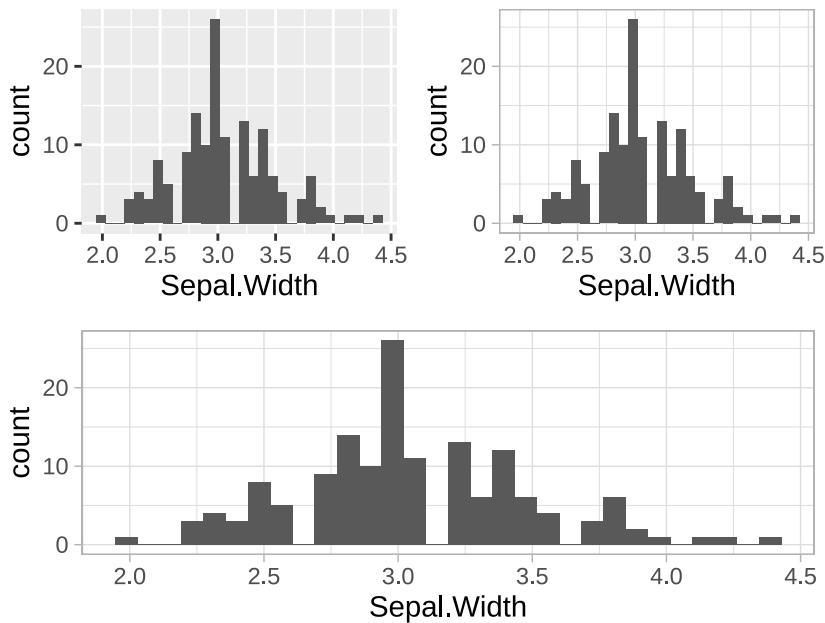
```
library(patchwork)
p1 + p2
```



1145

- 1146 Natürlich können auch weitere Plots hinzugefügt werden (auch in unterschiedlichen Dimensionen):

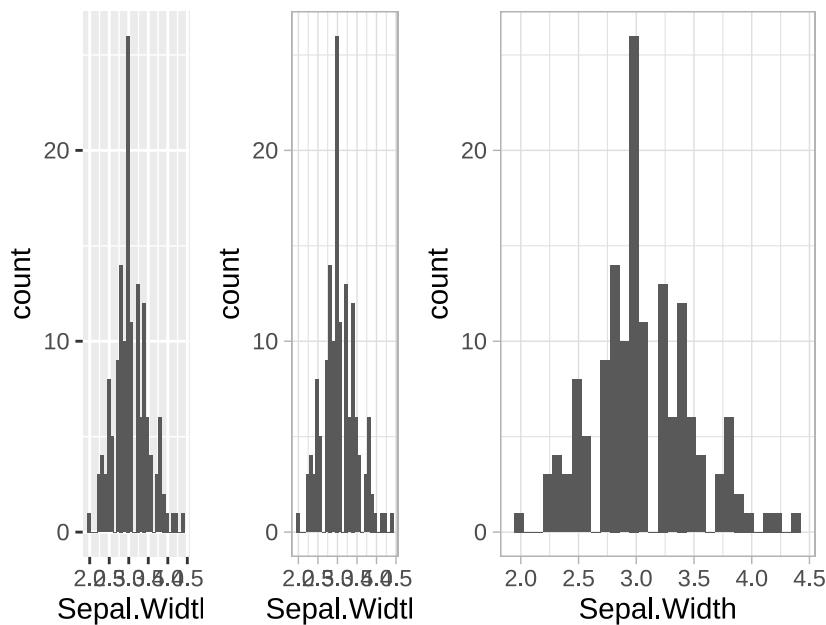
```
(p1 + p2) / p2
```



1147

- 1148 Des weiteren können mit `|` auch Plots gegenüber gestellt werden.

(p1 + p2) | p2

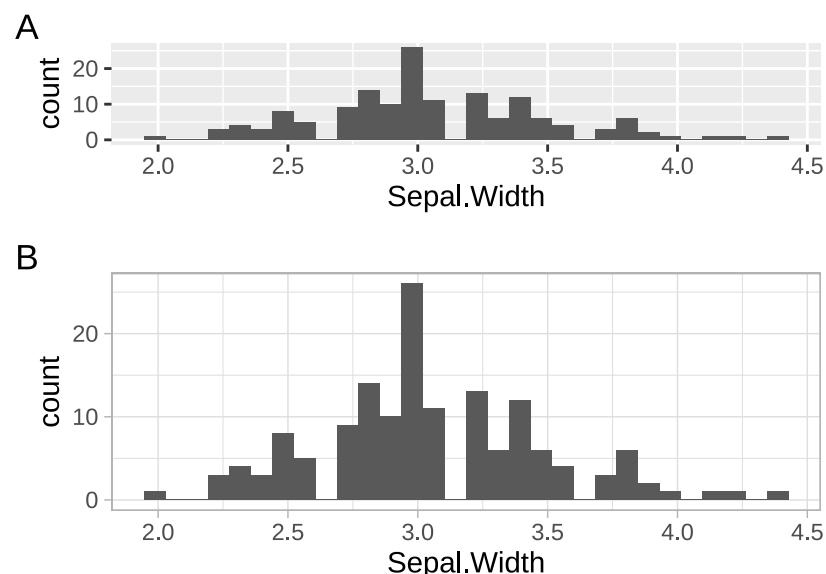


1149

1150 Weitere Optionen können mit `plot_layout()` und `plot_annotation()` angepasst werden. Mit  
 1151 `plot_layout()` kann die Anordnung der Plots bestimmt werden (z.B. über die Argument `nrow`  
 1152 und `ncol`), sowie deren *relative* Größe (über die Argumente `widths` und `heights`). Mit der Funktion  
 1153 `plot_annotation()` können zusätzliche Beschriftungen hinzugefügt werden, wie beispielsweise eine Titel  
 1154 (Argument `title`) oder ein Buchstabe/Zahl für jedes Element (Argument `tag_levels`).

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

Zwei Histogramme



1155

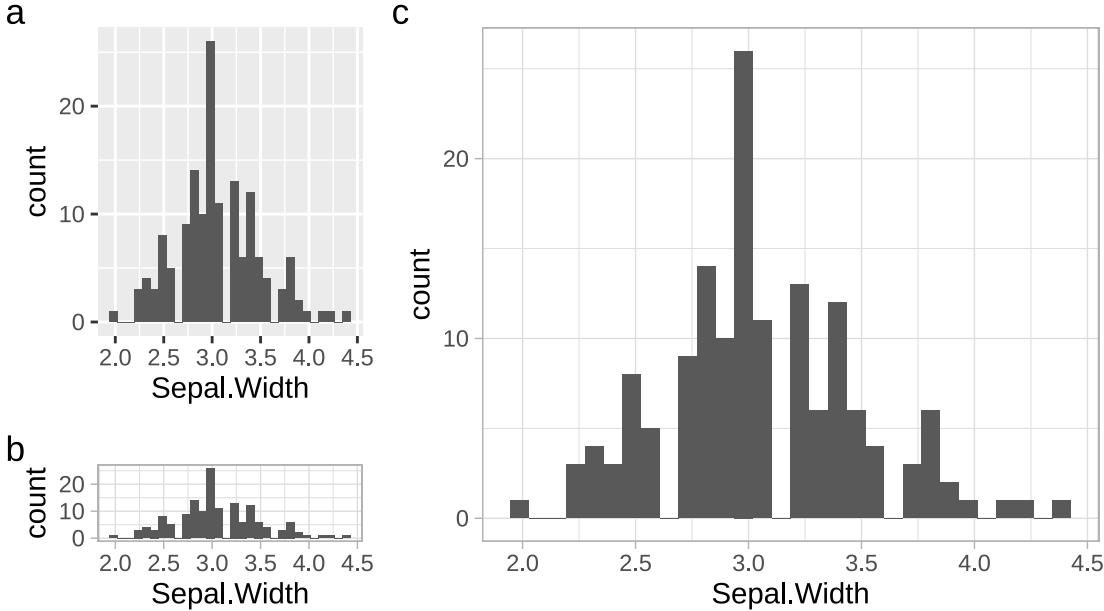
1156

---

**Aufgabe 23: Mehrere Plots zusammenfügen**

---

1159 Versuchen Sie die folgende Zusammenstellung der Plots nachzumachen:



1160

**8.4.3 Speichern von plots**

1162 Sie können mit `ggsave()` eine zwischen-gespeicherte Abbildung exportieren, indem Sie den Variablenamen  
1163 übergeben. Wenn Sie keine Variable übergeben, wird automatisch die letzte Abbildung gespeichert. Das  
1164 Dateiformat wird aus dem Dateinamen übernommen. Die Größe der Inhalte können Sie sehr leicht mit den  
1165 Argumenten `width` und `height` oder `scale` gesteuert werden.

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

## 1166 9 Mit Daten arbeiten

### 1167 9.1 dplyr eine Einführung

1168 `dplyr` ist eine Erweiterung von R (ein Paket), die das Ziel hat, den Umgang mit Daten einfacher und  
1169 schneller zu machen.

1170 `dplyr` definiert 5 Verben, um mit Daten zu arbeiten. Diese sind:

- 1171 • `filter`
- 1172 • `select`
- 1173 • `arrange`
- 1174 • `mutate`
- 1175 • `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))
```

1176 Damit die Funktionen aus `dplyr` verwendet werden können, müssen wir als erstes das Paket `dplyr` laden.  
1177 The online Resource [R for Data Science](#) gibt einen sehr guten, tieferen Eindruck in die Arbeit mit `dplyr`  
1178 und erklärt auch die `dplyr` Philosophie noch Mal sehr genau für Beginner.

```
library(dplyr)
```

1179 Sollte dies zu einer Fehlermeldung führen, dann müssen Sie das Paket `dplyr` erst installieren. Dafür müssen  
1180 Sie `einmalig install.packages("dplyr")` installieren.

1181 `dplyr` stellt unterschiedliche Funktionen zum Arbeiten mit Daten zur Verfügung. Es gibt fünf Grundfunktionen  
1182 für die am häufigsten vorkommenden Operationen. Mit der Funktion `filter()` können unterschiedliche  
1183 Beobachtungen gefiltert werden:

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

```
1184 ## id plot bhd alter
1185 ## 1 1 1 50 10
1186 ## 2 2 1 29 30
1187 ## 3 3 2 13 31
1188 ## 4 4 2 23 24
1189 ## 5 5 3 25 25
```

1190 Es können auch mehrere Spalten verwendet werden.

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

```
1191 ## id plot bhd alter
1192 ## 1 2 1 29 30
1193 ## 2 3 2 13 31
1194 ## 3 4 2 23 24
```

```
1195 ## 4 5 3 25 25
```

1196 Natürlich kann genau das gleiche Ergebnis mit dem ‘normalen’ R erreicht werden, dies wäre dann:

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

```
1197 ## id plot bhd alter
```

```
1198 ## 2 2 1 29 30
```

```
1199 ## 3 3 2 13 31
```

```
1200 ## 4 4 2 23 24
```

```
1201 ## 5 5 3 25 25
```

1202 Eine weitere Funktion aus dem Paket **dplyr** ist **select()**. Damit können Spalten aus einem **data.frame**

1203 ausgewählt werden. Dabei können auch die Spaltennamen unbenannt werden.

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

```
1204 ## bhd
```

```
1205 ## 1 50
```

```
1206 ## 2 29
```

```
1207 ## 3 13
```

```
1208 ## 4 23
```

```
1209 ## 5 25
```

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

```
1210 ## bhd id
```

```
1211 ## 1 50 1
```

```
1212 ## 2 29 2
```

```
1213 ## 3 13 3
```

```
1214 ## 4 23 4
```

```
1215 ## 5 25 5
```

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

```
1216 ## BHD id
```

```
1217 ## 1 50 1
```

```
1218 ## 2 29 2
```

```
1219 ## 3 13 3
```

```
1220 ## 4 23 4
```

```
1221 ## 5 25 5
```

1222 Mit der Funktion **arrange()** können die Beobachtungen in einem **data.frame** sortiert werden.

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

```
1223 ## id plot bhd alter
```

```
1224 ## 1 3 2 13 31
```

```
1225 ## 2 4 2 23 24
```

```
1226 ## 3 5 3 25 25
```

```
1227 ## 4 2 1 29 30
1228 ## 5 1 1 50 10
```

1229 Mit der Funktion `desc()` kann die Anordnung in absteigender Reihenfolge sortiert werden.

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

```
1230 ## id plot bhd alter
1231 ## 1 1 1 50 10
1232 ## 2 2 1 29 30
1233 ## 3 5 3 25 25
1234 ## 4 4 2 23 24
1235 ## 5 3 2 13 31
```

1236 Mit der Funktion `mutate()` kann man eine neue Spalte hinzufügen.

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

```
1237 ## id plot bhd alter bhd_mm fl
1238 ## 1 1 1 50 10 500 1963.4954
1239 ## 2 2 1 29 30 290 660.5199
1240 ## 3 3 2 13 31 130 132.7323
1241 ## 4 4 2 23 24 230 415.4756
1242 ## 5 5 3 25 25 250 490.8739
```

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

```
1243 ## id plot bhd alter mean_bhd
1244 ## 1 1 1 50 10 28
1245 ## 2 2 1 29 30 28
1246 ## 3 3 2 13 31 28
1247 ## 4 4 2 23 24 28
1248 ## 5 5 3 25 25 28
```

1249 Mit der Funktion `summarise()` können Spalten zusammengefasst werden.

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

```
1250 ## mean_bhd mean_sd
1251 ## 1 28 13.63818
```

1252

---

1253 **Aufgabe 24: Datenmanipulation mit dplyr**


---

1254

- 1255     1. Laden Sie den Datensatz
- `data/bhd_1.txt`
- 
- 1256     2. Berechnen Sie folgende Werte für alle Einträge und speichern Sie die Ergebnisse in
- `erg1`
- 
- 1257         • mittlerer
- `bhd`
- 
- 1258         • maximales
- `alter`
- 
- 1259         • die Standardabweichung des BHDs
- 
- 1260         • die Anzahl Bäume mit einem BHD > 30

1261 **9.2 Arbeiten mit gruppierten Daten**
1262 Zusätzlich können `mutate` und `summarise` auch auf gruppierte Daten angewendet werden. Dafür müssen  
1263 wir erst die Funktion `group_by()` aufrufen und als Argumente die Variablen übergeben, die die Gruppen  
1264 definieren.

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

id plot bhd alter bhd_m
1 1 1 50 10 28
2 2 2 29 30 28
3 3 2 13 31 28
4 4 2 23 24 28
5 5 3 25 25 28

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

A tibble: 5 x 5
Groups: plot [3]
id plot bhd alter bhd_m
<int> <dbl> <dbl> <dbl> <dbl>
1 1 1 50 10 39.5
2 2 2 29 30 39.5
3 3 3 13 31 18
4 4 4 23 24 18
5 5 5 25 25 25

summarise(dat, bhd_m = mean(bhd))

bhd_m
1 28

summarise(dat1, bhd_m = mean(bhd))

A tibble: 3 x 2
plot bhd_m
<dbl> <dbl>
```

```
1284 ## <dbl> <dbl>
1285 ## 1 1 39.5
1286 ## 2 2 18
1287 ## 3 3 25
```

1288

---

**Aufgabe 25: dplyr mit gruppierten Daten**

---

- 1291 1. Laden Sie den Datensatz `data/bhd_1.txt`
- 1292 2. Berechnen Sie für jede Baumart und Standort die folgenden Werte:
  - 1293 • mittlerer `bhd`
  - 1294 • maximales `alter`
  - 1295 • die Standardabweichung des BHDs
  - 1296 • die Anzahl Bäume mit einem BHD > 30
- 1297 3. Ordnen Sie das Ergebnis nach Baumart und BHD.

1298 **9.3 pipes oder %>%**

1299 Mit *Pipes* (`%>%`) kann man das Ergebnis einer Funktion einfach an eine nachfolgende Funktion weiterreichen.

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

1300 Wir kennen bis jetzt:

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

1301 ## [1] 3.333333

1302 Mit *Pipes*, die durch das Symbol `%>%` dargestellt werden<sup>10</sup>, können wir das etwas vereinfachen und nacheinander schreiben:

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

1304 ## [1] 3.333333

1305 Oder sogar

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

1306 ## [1] 3.333333

1307

---

**Aufgabe 26: Pipes %>%**

---

1310 Wiederholen Sie die letzte Aufgabe, aber diesmal ohne Zwischenergebnisse zu speichern:

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

- 1311 1. Laden Sie den Datensatz `data/bhd_1.txt`.
- 1312 2. Berechnen Sie für jede Baumart und Standort die folgenden Werte:
- 1313 • mittlerer `bhd`
  - 1314 • maximales `alter`
  - 1315 • die Standardabweichung des BHDs
  - 1316 • die Anzahl Bäume mit einem BHD > 30
- 1317 3. Ordnen Sie das Ergebnis nach Baumart und BHD.

1318 **9.4 Joins**

1319 Eine weitere häufige Aufgabe beim Daten Management ist es Daten zusammenzuführen. Nehmen Sie an, dass  
1320 wir folgende Aufnahmen gemacht haben

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

1321 und jeder Baum lediglich mit einer `id` versehen wurde. In einer zweiten Tabelle wurden dann weitere Daten  
1322 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")
)
```

1323 Ziel ist es jetzt die Bäume aus `aufnahmen` mit den Informationen aus den `metadaten` zu verbinden. Dazu  
1324 dient `id` als Bindeglied (oft auch Schlüssel genannt).

1325 Dazu gibt es vier Möglichkeiten.

1326 Zur Durchführung gibt es in base R die Funktion `merge()`. Wir werden aber gleich die Funktionen aus dem  
1327 Paket `dplyr` verwenden.

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

1328 ## id bhd art gebiet
1329 ## 1 1 20 <NA> <NA>
1330 ## 2 2 31 Ta A
1331 ## 3 3 74 Bu B

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

1332 ## id bhd art gebiet
1333 ## 1 2 31 Ta A
1334 ## 2 3 74 Bu B
1335 ## 3 4 NA Bu B
```

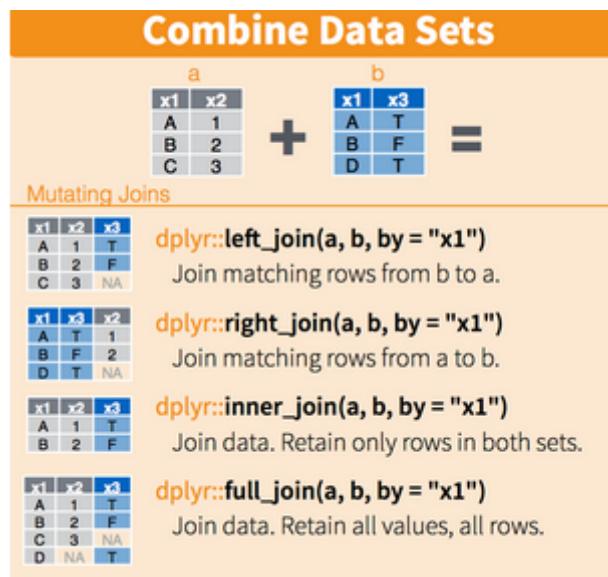


Abbildung 8: Joins (Quelle Rstudio)

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

```
1336 ## id bhd art gebiet
1337 ## 1 2 31 Ta A
1338 ## 2 3 74 Bu B
full_join(aufnahmen, metadaten, by = "id")
```

```
1339 ## id bhd art gebiet
1340 ## 1 1 20 <NA> <NA>
1341 ## 2 2 31 Ta A
1342 ## 3 3 74 Bu B
1343 ## 4 4 NA Bu B
```

1344 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"))
```

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

1349

1350 **Aufgabe 27: Verbinden von Daten**

- 
- 1351
- 1352 • Lesen Sie die Datensätze `daten/bhd_2.txt` und `daten/bhd_2_meta.txt` ein.
  - 1353 • Stellen Sie sicher, dass es für den `bhd` keine fehlenden Werte gibt (entfernen sie entsprechende Zeilen)
  - 1354 • Fügen Sie zu den Metadaten (gespeichert in `bhd_2_meta`) die Anzahl Bäume und den mittleren `bhd`
  - 1355 hinzu pro Gebiet.

1356 **9.5 ‘long’ and ‘wide’ Datenformate**

1357 Unter anderem Wickham (2014) empfieilt das Prinzip von *tidy Data*. Nach diesem Prinzip sollten Daten wie  
1358 folgt organisiert sein:

- 1359
- Jede Zeile ist ein Merkmalsträger/Subjekt/Objekt (z. B. eine Person oder ein Baum).
  - 1360 • Jede Spalte ist eine Variable (ein Merkmal), die den Merkmalsträger beschreibt.
  - 1361 • In jeder Zelle ist genau ein Wert (Merkmalausprägung), nämlich der Wert, der Variable für den Merk-  
1362 malsträger.

1363 Zum Beispiel enthalten Spaltennamen oft Informationen, die eigentlich in einer Variable gespeichert werden  
1364 sollten. Folgendes Beispiel gibt die BHD Messung von 3 Bäumen in 3 Jahren wieder. Der Vorteil von *tidy*  
1365 Daten ist, dass viele Funktionen diese Form erwarten. Somit müssen sie Ihre Daten nur ein Mal organisieren  
1366 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)
)
```

1367 Der `tibble` aus dem Paket `tibble` ist ein moderner Data Frame, der sich in seinen Funktionen in das `tidy`  
1368 Universum einfügt. Wir können auch einfach das Paket `tidyverse` laden, um alle Pakete, die Teil des `tidy`  
1369 Universums sind gleichzeitig zu laden. Der `tibble` beinhaltet alle Funktionen, die der Data Frame auch  
1370 beinhaltet und kann auch für genau die gleichen Zwecke verwendet werden. Der `tibble` hat u. A. eine  
1371 modernere Darstellung im Konsolenoutput.

1372 Diese Daten sind jetzt im `wide`-Format gespeichert und folglich nicht `tidy`, weil Information über die Daten  
1373 (nämlich das Jahr der Aufnahme in den Spaltennamen gespeichert sind). Besser wäre eine Struktur mit  
1374 nur drei Spalten: `id`, `jahr` und `bhd`. Um die Daten in so eine Struktur zu bringen, gibt es die Funktion  
1375 `pivot_longer()` aus dem Paket `tidyR`, die ebenfalls Teil des `tidyverse` ist.

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

1376 ## # A tibble: 9 x 3
1377 ## id name value
```

```

1378 ## <int> <chr> <dbl>
1379 ## 1 1 bhd2015 30
1380 ## 2 1 bhd2026 31
1381 ## 3 1 bhd2017 34
1382 ## 4 2 bhd2015 31
1383 ## 5 2 bhd2026 31
1384 ## 6 2 bhd2017 32
1385 ## 7 3 bhd2015 32
1386 ## 8 3 bhd2026 33
1387 ## 9 3 bhd2017 33

```

1388 Wenn wir die Spalten für die Variable und den Wert gleich sinnvoll benennen möchten, können wir das über  
1389 die Argumente `names_to` und `value_to` machen.

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

```

1390 ## # A tibble: 9 x 3
1391 ## id jahr bhd
1392 ## <int> <chr> <dbl>
1393 ## 1 1 bhd2015 30
1394 ## 2 1 bhd2026 31
1395 ## 3 1 bhd2017 34
1396 ## 4 2 bhd2015 31
1397 ## 5 2 bhd2026 31
1398 ## 6 2 bhd2017 32
1399 ## 7 3 bhd2015 32
1400 ## 8 3 bhd2026 33
1401 ## 9 3 bhd2017 33

```

1402 Analog zu der Funktion `pivot_longer()` gibt es auch die Funktion `pivot_wider()`, um vom Daten vom  
1403 `long`-Format ins `wide`-Format zu transformieren.

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

```

1404 ## # A tibble: 3 x 4
1405 ## id bhd2015 bhd2026 bhd2017
1406 ## <int> <dbl> <dbl> <dbl>
1407 ## 1 1 30 31 34
1408 ## 2 2 31 31 32
1409 ## 3 3 32 33 33

```

1410

1411 **Aufgabe 28: Zeitliche Verlauf von BHDs**

---

1412

1413 In der Datei `bhd_3.csv` befinden sich gemessene BHDs (in cm) von unterschiedlichen Bäumen zu unter-  
 1414 schiedlichen Zeitpunkten. Erstellen Sie ein Liniendiagramm, das den zeitlichen (x-Achse) Verlauf der BHDs  
 1415 (y-Achse) für die unterschiedlichen Bäume darstellt.

1416 **9.6 Auswählen von Variablen**

1417 Sobald die Datensätze etwas umfangreicher werden (d. h., es gibt mehrere Spalten in einem `data.frame`),  
 1418 können innerhalb vieler `dplyr`-Funktionen spezielle Funktionen verwendet werden, um Variablen auszuwählen.  
 1419 Wenn die genaue Position der Spalten bekannt ist, kann man mit dem `:`-Operator und der Position Spalten  
 1420 auswählen:

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

```
1421 ## Sepal.Length Sepal.Width Petal.Length
1422 ## 1 5.1 3.5 1.4
1423 ## 2 4.9 3.0 1.4
1424 ## 3 4.7 3.2 1.3
```

1425 Diese Vorgehensweise kann gehährlich sein, da sich manchmal Spalten verschieben und sich somit die  
 1426 Positionen ändern. Es besser Spalten immer explizit mit Namen statt mit Nummern anzusprechen.

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

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

1431 `select()` erlaubt es, auch hier den `:`-Operator zu verwenden:

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

```
1432 ## Sepal.Length Sepal.Width Petal.Length
1433 ## 1 5.1 3.5 1.4
1434 ## 2 4.9 3.0 1.4
1435 ## 3 4.7 3.2 1.3
```

1436 Es gibt auch einige spezielle Funktionen, um Spalten innerhalb eines `select()`-Aufrufs auszuführen:

- 1437 • `starts_with()`: Hier kann man ein Muster angeben, mit dem ein Text anfangen muss.
- 1438 • `ends_with()`: Diese Funktion ist analog zu `starts_with()`, jetzt wird aber am Ende des Spaltennamens  
 1439 nach dem Muster gesucht.
- 1440 • `contains()`: Hier kann ein Muster übergeben werden, das irgendwo im Spaltennamen sein muss.
- 1441 • `everything()`: Mit dieser Funktion werden alle Spalten ausgewählt.

1442 • `last_col()`: Mit dieser Funktion wird nur die letzte Spalte ausgewählt (dass ist die Spalte, die ganz  
 1443 rechts ist).

1444 Sämtliche Auswahlen können mit - umgekehrt werden.

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

1445 ## Sepal.Length Sepal.Width

1446 ## 1 5.1 3.5

1447 ## 2 4.9 3.0

1448 ## 3 4.7 3.2

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

1449 ## Petal.Length Petal.Width Species

1450 ## 1 1.4 0.2 setosa

1451 ## 2 1.4 0.2 setosa

1452 ## 3 1.3 0.2 setosa

1453 `select()` bietet auch noch die Möglichkeit, Spaltennamen zu ändern.

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

1454 ## sep\_width

1455 ## 1 3.5

1456 ## 2 3.0

1457 ## 3 3.2

1458

### 1459 Aufgabe 29: Auswählen von Spalten

---

1461 In der Datei `messungen_1.csv` sind Messungen von zwei Sensoren enthalten für die ersten vier Monate eines  
 1462 Jahres. Führen Sie folgende Abfragen durch:

1463 1. Wählen Sie alle Messungen für Januar aus.

1464 2. Wählen Sie alle Messungen für Januar und März aus.

## 1465 9.7 Einzelne Beobachtungen abfragen (`slice()`)

1466 Mit dem Befehl `slice()` kann man einzelne Beobachtungen (= Zeilen) aus einem `data.frame` abfragen.

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

1467 ## Sepal.Length Sepal.Width Petal.Length Petal.Width Species

1468 ## 1 5.1 3.5 1.4 0.2 setosa

1469 ## 2 4.4 2.9 1.4 0.2 setosa

1470 ## 3 5.1 3.5 1.4 0.3 setosa

1471 Davon gibt es drei nützliche Varianten: 1) `slice_head()` und `slice_tail()`; 2) `slice_max()` und  
 1472 `slice_min()`; 3) `slice_random()`.

1473 `slice_head()` und `slice_tail()` sind analog zu `head()` und `tail()`, aber mit dem entscheidenden Unter-  
 1474 schied, dass Gruppierungen berücksichtigt werden. Wenn keine Gruppierung in den Daten vorhanden ist, gibt  
 1475 es keinen Unterschied.

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

```
1476 ## Sepal.Length Sepal.Width Petal.Length Petal.Width Species
1477 ## 1 5.1 3.5 1.4 0.2 setosa
1478 ## 2 4.9 3.0 1.4 0.2 setosa
```

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

```
1479 ## Sepal.Length Sepal.Width Petal.Length Petal.Width Species
1480 ## 1 5.1 3.5 1.4 0.2 setosa
1481 ## 2 4.9 3.0 1.4 0.2 setosa
```

1482 Sobald jedoch eine gruppierende Variable eingeführt wird, gibt `slice_head()` die ersten n Beobachtungen  
 1483 für jede Gruppe zurück und `head()` für den gesamten Datensatz.

```
base head
```

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

```
1484 ## # A tibble: 2 x 5
1485 ## # Groups: Species [1]
1486 ## Sepal.Length Sepal.Width Petal.Length Petal.Width Species
1487 ## <dbl> <dbl> <dbl> <dbl> <fct>
1488 ## 1 5.1 3.5 1.4 0.2 setosa
1489 ## 2 4.9 3 1.4 0.2 setosa
```

```
dplyr slice_head
```

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

```
1490 ## # A tibble: 6 x 5
1491 ## # Groups: Species [3]
1492 ## Sepal.Length Sepal.Width Petal.Length Petal.Width Species
1493 ## <dbl> <dbl> <dbl> <dbl> <fct>
1494 ## 1 5.1 3.5 1.4 0.2 setosa
1495 ## 2 4.9 3 1.4 0.2 setosa
1496 ## 3 7 3.2 4.7 1.4 versicolor
1497 ## 4 6.4 3.2 4.5 1.5 versicolor
1498 ## 5 6.3 3.3 6 2.5 virginica
1499 ## 6 5.8 2.7 5.1 1.9 virginica
```

1500 `slice_tail()` funktioniert analog zu `slice_head()` mit dem einzigen Unterschied, dass nicht die ersten n

1501 Zeilen zurück gegeben werden sondern die letzten **n** Zeilen.  
 1502 `slice_max()` und `slice_min()` geben die Beobachtung mit dem maximalen bzw. minimalen Wert einer  
 1503 Variable zurück. Auch hier werden Gruppen berücksichtigt.

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

1504 ## Sepal.Length Sepal.Width Petal.Length Petal.Width Species  
 1505 ## 1 7.9 3.8 6.4 2 virginica

1506 Und mit Gruppen:

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

1507 ## # A tibble: 3 x 5  
 1508 ## # Groups: Species [3]  
 1509 ## Sepal.Length Sepal.Width Petal.Length Petal.Width Species  
 1510 ## <dbl> <dbl> <dbl> <dbl> <fct>  
 1511 ## 1 5.8 4 1.2 0.2 setosa  
 1512 ## 2 7 3.2 4.7 1.4 versicolor  
 1513 ## 3 7.9 3.8 6.4 2 virginica

1514 `slice_min()` funktioniert genau gleich, nur dass die Beobachtung (=Zeile) mit dem kleinsten Wert einer  
 1515 Variable zurück gegeben wird.

1516 Die Funktion `slice_sample()` erlaubt es zufällige Beobachtungen zu ziehen. Dabei kann über das Argument **n**  
 1517 die Anzahl an Beobachtungen angegeben werden oder über das Argument **prop** der Anteil an Beobachtungen.

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

1518 ## Sepal.Length Sepal.Width Petal.Length Petal.Width Species  
 1519 ## 1 6.5 2.8 4.6 1.5 versicolor  
 1520 ## 2 6.3 3.3 4.7 1.6 versicolor  
 1521 ## 3 7.2 3.2 6.0 1.8 virginica  
 1522 ## 4 4.9 3.6 1.4 0.1 setosa  
 1523 ## 5 6.0 2.7 5.1 1.6 versicolor

1524 Das Ergebnis ist bei jedem von Ihnen anders, da es sich um eine zufällige Ziehung handelt. Wenn Sie diese  
 1525 Ergebnisse wiederholen möchten, können Sie über `set.seed()` die zufällige Ziehung reproduzierbar machen.

```
set.seed(123)
```

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

1526 ## Sepal.Length Sepal.Width Petal.Length Petal.Width Species  
 1527 ## 1 4.3 3.0 1.1 0.1 setosa  
 1528 ## 2 5.0 3.3 1.4 0.2 setosa  
 1529 ## 3 7.7 3.8 6.7 2.2 virginica  
 1530 ## 4 4.4 3.2 1.3 0.2 setosa  
 1531 ## 5 5.9 3.0 5.1 1.8 virginica

1532 Wenn beispielsweise 5% der Beobachtungen gezogen werden sollen, kann dies so gemacht werden:

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

```
1533 ## Sepal.Length Sepal.Width Petal.Length Petal.Width Species
1534 ## 1 7.7 3.8 6.7 2.2 virginica
1535 ## 2 5.5 2.5 4.0 1.3 versicolor
1536 ## 3 5.5 2.6 4.4 1.2 versicolor
1537 ## 4 6.5 3.0 5.2 2.0 virginica
1538 ## 5 6.1 3.0 4.6 1.4 versicolor
1539 ## 6 6.3 3.4 5.6 2.4 virginica
1540 ## 7 5.1 2.5 3.0 1.1 versicolor

1541 slice_sample() berücksichtigt ebenfalls Gruppen. Mit den Argumenten replace und weight_by dann die
1542 Zufallsziehung genauer spezifiziert werden. replace sagt, ob eine gezogenen Beobachtung wieder zurück gelegt
1543 wird oder nicht. Mit dem Argument weight_by können optional gewichtete für jede Beobachtung vergeben
1544 werden.
```

1545

### 1546 Aufgabe 30: Daten beschreiben

---

1548 Verwenden Sie den Datensatz `bhd_1.txt` und finden Sie für jedes Gebiet und Art die Beobachtung mit
1549 kleinsten BHD.

## 1550 9.8 Spalten trennen

1551 Ein gut geplanter Datensatz besteht aus Beobachtungen (Zeilen), Variablen (Spalten) und in jeder Zelle ist
1552 immer ein **genau** ein Wert gespeichert. Leider gibt es oft Datensätze, bei denen mehr als ein Wert pro Zelle
1553 gespeichert wurde. Die Funktion `separate()` kann helfen solche Daten zu trennen.

1554 Wir verwenden einen erfunden Datensatz zu Beobachtungen von Tieren und einer geschätzten Distanz zu
1555 diesen Tieren.

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

1556 In der Spalte `beobachtung` sind zwei Informationen gespeichert: Die Distanz zur Beobachtung und die Art.
1557 Das ist ungünstig, weil wir so weder nach Tierart noch nach distanz filtern können (nicht *tidy*). Mit der
1558 Funktion `separate()`, können wir Beobachtungen einer Spalte in mehrere Spalten trennen. Dafür muss der
1559 Spaltennamen (Argument `col`), die neuen Spaltennamen (Argument `into`) und das Trennzeichen (Argument
1560 `sep`) angegeben werden.

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

1561 ## # A tibble: 4 x 3

```
1562 ## id Distanz Art
1563 ## <int> <chr> <chr>
1564 ## 1 1 10m " Reh"
1565 ## 2 2 100m " Reh"
1566 ## 3 3 20m " Fuchs"
1567 ## 4 4 40 "Reh"
1568 Nach dem Aufruf von separate() gibt es zwei neue Spalten (Distanz und Art), die die alte Spalte
1569 beobachtung ersetzen.
```

1570

---

### 1571 Aufgabe 31: Aufräumen

---

1573 Verwenden Sie den folgenden Datensatz und bringen Sie ihn in eine Form, die sicherstellt dass

- 1574 • jede Zelle genau einen Wert enthält.  
1575 • jede Zeile eine Beobachtung ist.  
1576 • 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")
)
```

## 1577 10 Arbeiten mit Text

1578 Bis jetzt haben wir fast ausschließlich mit Zahlen oder Abbildungen gearbeitet. R bietet aber auch viele  
 1579 Werkzeuge, um mit Text zu arbeiten. Wir wollen hier ein paar Funktionen dafür vorstellen. Als erstes sollte  
 1580 nochmals klar gestellt werden, was eigentlich ein Text ist. In R ist alles, das innerhalb von doppelten ("") oder  
 1581 einfachen ('') Anführungszeichen geschrieben ist, Text.

1582 Anbei einige Beispiele:

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

1583 Wichtig ist hier zu sehen, dass z nicht als Zahl sondern, als Text interpretiert wird (siehe Datentypen).

```
z + 1
```

1584 ## Error in z + 1: nicht-numerisches Argument für binären Operator

1585 Wenn man sicher ist, dass es sich bei einem Textobjekt um eine Zahl handelt, kann man dies mit der Funktion  
 1586 `as.numeric()` in eine Zahl umwandeln.

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

1587 ## [1] 31

1588 Aber mit a führt dies wieder zu einem NA-Wert, da a nicht in eine Zahl umgewandelt werden kann.

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

1589 ## Warning: NAs durch Umwandlung erzeugt

1590 ## [1] NA

### 1591 10.1 Arbeiten mit Text

1592 Wir wollen erst einmal drei Funktionen besprechen, die es erlauben mit Text zu arbeiten. Die Funktion  
 1593 `nchar()`<sup>11</sup> gibt an wie viele Zeichen ein Text hat. Also z.B.

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

1594 ## [1] 5

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

1595 ## [1] 2

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

1596 ## [1] 20

1597 Die Funktion `paste()` erlaubt es verschiedene Variablen mit Text zu verbinden. Wenn wir z. B. die Variablen  
 1598 `vorname <- "Eva"` und `name <- "Meier"` haben und wir wollen eine neue Variable `full_name <- Eva`

---

<sup>11</sup>char ist kurz für character.

1599 "Meier" erzeugen, dann kann das mit der Funktion `paste()` gemacht werden.

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

1600 ## [1] "Eva Meier"

1601 Die Funktion `paste()` hat das Argument `sep`, das auf ein Leerzeichen ( ) gesetzt ist, aber auch anders sein  
1602 kann und das Trennzeichen definiert.

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

1603 ## [1] "Eva, Meier"

1604 Die Funktion `substr()` erlaubt es am Anfang oder Ende eines Wortes etwas abzuschneiden. Dabei muss  
1605 immer die Anfangs- (`start`) und Endposition (`stop`) angegeben werden.

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

1606 ## [1] "Hal"

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

1607 ## [1] "allo"

1608

### 1609 Aufgabe 32: Arbeiten mit Text 1

---

1610 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")
```

1612 1. Aus wie vielen Buchstaben besteht jedes Wort?

1613 2. Finden Sie das längste Wort.

1614 3. Wie viel Prozent der Wörter fangen mit einem S an?

1615 4. Fügen Sie jedem Wort seine Position im Vektor hinzu. Beispielsweise soll aus `Vogel` "2. Vogel" werden

1616 usw.

## 1617 10.2 Finden von Textmustern

1618 Mit der Funktion `grep()` können Muster in einem Text gefunden werden. Wenn wir beispielsweise folgenden  
1619 Vektor mit Textelementen haben.

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

1620 Und wir wollen alle Straßennamen die ein `weg` haben abfragen, dann können wir folgenden Befehl ausführen:

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

1621 ## [1] 2

1622 Im zweiten Element von `txt` kommen die Zeichen `Weg` vor. Beachte, in der Standardeinstellung wird zwischen  
1623 Groß- und Kleinschreibung unterschieden. Dies kann mit dem Argument `ignore.case = TRUE` angepasst  
1624 werden.

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

1625 ## [1] 1 2

1626 Mit der Funktion `sub` können Zeichen innerhalb einer Zeichenkette ersetzt werden.

1627 So ersetzt der folgende Ausdruck `ae` mit `ä`.

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

1628 ## [1] "Friedländer Weg"

1629 Wenn allerdings das zu ersetzende Zeichen mehr als einmal vorkommt und beide Instanzen ersetzt werden  
1630 sollen braucht man die Funktion `gsub`.

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

1631 ## [1] "Friedländer Weg und Reinhäuser Landstrasse sind zwei Strassen in Goettingen."

1632 Mit `sub()` wird nur das erste `ae` ersetzt, während `gsub()` alle `ae` mit einem `ä` ersetzt.

```
gsub("ae", "ä", txt)
```

1633 ## [1] "Friedländer Weg und Reinhäuser Landstrasse sind zwei Strassen in Goettingen."

1634 Oft ist der genaue Ausdruck den man finden möchte jedoch Variabel. Beispielsweise möchte man alle Wörter  
1635 mit einem Umlaut oder Zahlen finden möchte, kann man das oft abkürzen. Dafür gibt es reguläre Ausdrücke.  
1636 Wir werden hier nur ein paar beispielhafte Anwendungen besprechen.

1637 Sowohl in den Funktionen `grep()` als auch `(g)sub()` kann mit anstatt dem Muster (immer das erste  
1638 Argument) aus ein regulärer Ausdruck angegeben werden. Mit `[1-9]` sind alle Zahlen von 1 bis 9 gemeint.

1639 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)
```

1640 ## [1] 1 3

1641 Damit lässt sich auch das Problem mit groß- und kleingeschriebenen Wörtern lösen.

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

```
grep("[wW]eg", txt)
```

```
1642 ## [1] 1 2
```

1643

1644 **Aufgabe 33: Arbeiten mit Text 2**

---

1645 Verwenden Sie den folgenden Vektor:

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

1647 1. In wie vielen Wörtern kommt der Doppellaut au vor?

1648 2. Ersetzen Sie in allen Wörtern alle au mit \_ \_.

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

```
1649 ## [1] 5 6 13
```

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

```
1650 ## [1] "Versicherung" "Methoden" "Fluss" "Rudel" "B_ _m"
```

```
1651 ## [6] "H_ _s" "Foto" "Auffahrt" "Auto" "Handy"
```

```
1652 ## [11] "Teller" "Kalender" "Aufb_ _"
```

## 1653 11 Arbeiten mit Zeit

1654 Für den Computer bzw. R ist ein Datum erst einmal nichts anderes als ein Text. Für uns ist es sofort klar,  
 1655 dass der "13.2.2021" der 13. Februar 2021 ist, für den Computer nicht. Wir müssen R also irgendwie sagen,  
 1656 dass die 13 der Tag ist, die 2 der Monat und 2021 das Jahr. Dass der Computer die einzelnen Komponenten  
 1657 erkennt, und in einen Datentyp eigens für Zeitformate überführt, nennt man *parsen*<sup>12</sup>. Das Arbeiten mit  
 1658 Datum und Zeit kann anfangs sehr mühsam sein, aber sobald man einige Grundfertigkeiten erworben  
 1659 hat, kann man viele Aufgaben deutlich schneller und effizienter erledigen. Starten Sie am besten gleich mit  
 1660 "echten" Zeit-Datentypen und versuchen Sie nicht, sich irgendwie mit Text und numerischen Datentypen  
 1661 selbst etwas zu basteln. Der erste Schritt ist immer ein Datum zu *parsen*. Wir verwenden dafür Funktionen  
 1662 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)
```

1663 **lubridate** bietet eine Vielzahl von Funktionen zum *parsen* von Datum und Zeit, die sich aus:

- 1664 • y für Jahr,
- 1665 • m für Monat,
- 1666 • d für Tag,
- 1667 • h für Stunde,
- 1668 • m für Minute und
- 1669 • s für Sekunde

1670 zusammen setzten. Alle Funktionen nehmen als erstes Argument ein Textstring. Wenn wir z.B. den String  
 1671 2020-01-20 parsen wollen können wir das mit der Funktion **ymd** machen.

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

1672 ## [1] "2020-01-20"

1673 Dabei erkennt **lubridate** in der Regel die Trennzeichen:

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

1674 ## [1] "2020-01-20"

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

1675 ## [1] "2020-01-20"

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

1676 ## [1] "2020-01-20"

1677 Wenn die die Anordnung der einzelnen Komponenten anders ist, gibt es einfach eine andere Funktion.

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

---

<sup>12</sup>to parse heißt zergliedern bzw. grammatisch bestimmen.

1678 ## [1] "2020-01-20"  
 1679 Jetzt stellt sich die Frage, was der Vorteil ist, wenn R ein Datum parst.

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

1680 Wir können jetzt mit d arbeiten und einzelne Komponenten extrahieren.  

```
day(d)
```

1681 ## [1] 20  

```
month(d)
```

1682 ## [1] 1  

```
year(d)
```

1683 ## [1] 2020  
 1684 Oder auch Zeiteinheiten hinzufügen oder abziehen.

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

1685 ## [1] "2020-01-30"  

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

1686 ## [1] "2000-01-20"  

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

1687 ## [1] "2020-01-21 01:00:00 UTC"

1688

---

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

---

- 1691 • Parsen Sie folgende Zeitangaben 23.1.2020, 13.2.1992 20:55:23, Mar/3/97 und 10.7.2020 19:15  
 1692 und speichern Sie diese in einen Vektor d.  
 1693 • Extrahieren Sie nun aus jedem Element aus d das Jahr und die Stunde.  
 1694 • Fügen zu jedem Element in d 10 Tage hinzu.

1695 **11.1 Arbeiten mit Zeitintervallen**

1696 Mit zwei Zeitpunkten lassen sich Zeitintervalle (**Periods**) erstellen, dafür können wir die Funktion **interval()**  
 1697 aus dem Paket **lubridate** verwenden<sup>13</sup>.

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

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

---

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

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

1698 Wir können jetzt mit `int` arbeiten und beispielsweise das Intervall verschieben,

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

1699 ## [1] 2023-03-18 UTC--2024-03-18 UTC

1700 die Länge des Intervalls berechnen

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

1701 ## [1] 31536000

1702 oder testen, ob ein Datum innerhalb des Intervalls liegt.

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

1703 ## [1] TRUE

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

1704 ## [1] FALSE

1705 `%within%` funktioniert genauso mit Vektoren oder mit mehreren Intervallen. Wir könnten also zwei Intervalle  
1706 definieren (z.B. Ostern und Pfingsten).

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

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

1707 Und Überprüfen welche Termine in eines der zwei Intervalle fallen.

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

```
Ostern
```

```
termine %within% ostern
```

1708 ## [1] FALSE TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE

```
Pfingsten
```

```
termine %within% pfingsten
```

1709 ## [1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE TRUE FALSE FALSE

1710 Der Zeit-Datentyp wird auch oft benutzt, um den Zeitbedarf von längeren Berechnungen zu messen.

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

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

1711 ## [1] 0.4999484

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

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

1712 `## [1] 0.6720877`

## 1713 11.2 Formatieren von Zeit

1714 Für die Ausgabe in Berichten oder Grafiken soll das Datum oft in einer speziellen Form dargestellt werden.  
1715 Die Funktion `format()` bietet Möglichkeiten ein Datumsobjekt zurück in Text umzuwandeln.

1716 Ein Beispiel wäre `ymd("2021-2-10")` als 10.2.21 auszugeben.

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

1717 `## [1] "21.02.21"`

1718 Dabei handelt sich bei `%d.%m.%y` um Abkürzungen für die unterschiedlichen Komponenten eines Datumobjekts.  
1719 Siehe dazu die Hilfeseite von `strptime` (`help(strptime)`).

1720

---

## 1721 Aufgabe 35: Arbeiten mit Intervallen

---

1723 Wie viele Einträge aus dem Vektor `v1` befinden sich in einem Intervall, das zwischen dem 1.3.2021 und dem  
1724 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"
)
```

## 1725 11.3 Zeitreihen

1726 In der Forstbranche haben wir oft mit Zeitreihen zu tun. Zeitreihen sind Variablen, die in zeitlichen Intervallen  
1727 vorliegen. In der Regel monatlich, vierteljährig oder jährlich, wobei die Intervalle zwischen den Messungen  
1728 immer gleich lang sind. Wiederholungsmessungen von Forstinventuren (Forsteinrichtungen, Betriebsinventuren,  
1729 die meisten forstl. Experimente, die BWI, ...) sind also in der Regel keine Zeitreihen im engeren Sinne,  
1730 turnusmäßig vermessene Versuchsflächen, wie sie z. B. die Versuchsanstalten unterhalten oder jährlich  
1731 gemeldete Holzpreise jedoch schon.

1732 Statistisch gesehen haben Zeitreihen Besonderheiten gegenüber anderen Variablen, da Sie in erster Linie von  
1733 Ihrer eigenen Vergangenheit abhängen (autokorriert sind) und auch die Abhängigkeit anderer Variablen in  
1734 der Regel nur sinnvoll ist, wenn deren Vergangenheit mitberücksichtigt wird (cross-correlation). Konventionelle  
1735 Statistik ist oft nicht ausreichend, um Zeitreihen zu analysieren. Angefangen mit der Datendarstellung gibt  
1736 es spezifische Zeitreihen-Funktionen, welche auch alle in R integriert sind. Aus diesem Grund sollten Sie  
1737 Zeitreihen als solche speichern. Viele Funktionen erkennen den Datentyp und führen entsprechend spezifische  
1738 Operationen durch. Laden wir z. B. die Holzpreise für Fichte 2b (das sog. Leitsortiment), das Holzaufkommen  
1739 dieses Sortiments und die Preise für Nadelholz vom statistischen Bundesamt<sup>14</sup>. Wir laden die Daten

<sup>14</sup>Sie 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

1740 zunächst als csv:

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

1741 Mit der Funktion `ts` werden die Daten in ein Zeitreihenobjekt überführt (geparst). Die Spalte mit den  
1742 Jahren ist dann nicht mehr nötig, da die Informationen zur Zeit keine eigene Variable mehr sind, sondern zu  
1743 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

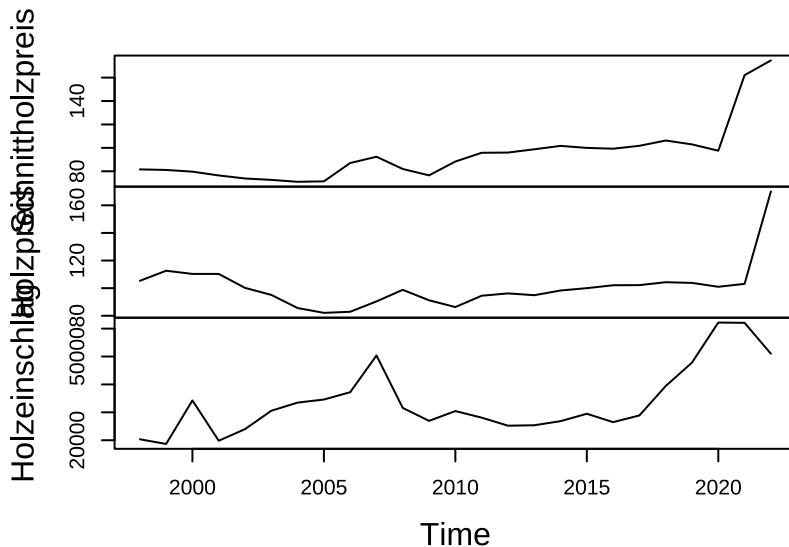
1744 ## [1] "double"

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

1745 Die wichtigsten Argumente sind - `data` Vektor oder Matrix, der nur die Daten enthält - `start` Startzeitpunkt -  
1746 `frequency` Anzahl der Beobachtungen pro Zeiteinheit, also z. B. 12 bei monatlichen oder 4 bei quartalsweisen  
1747 Erhebungen

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

**zr**

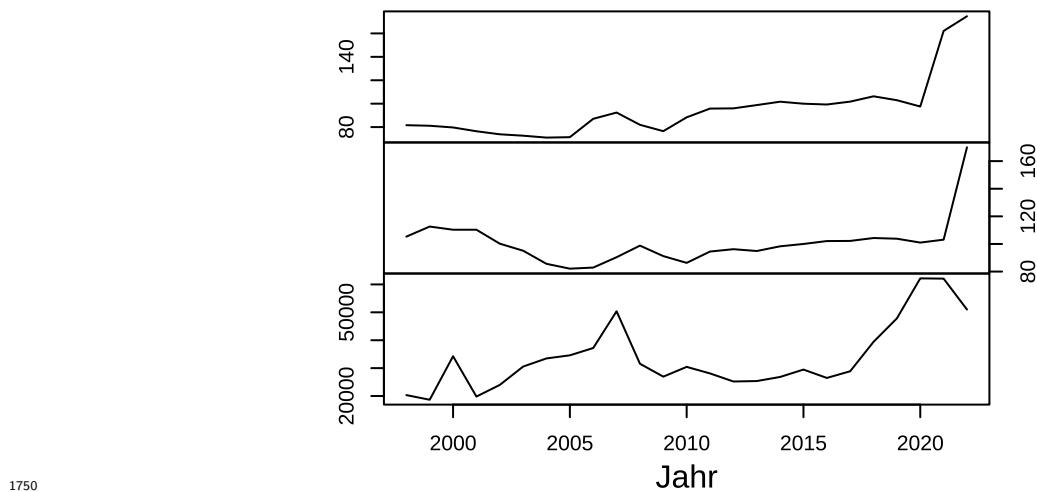


1748

1749 Wir können low-level Layer hinzufügen. Die meisten base plot Funktionen funktionieren auch für Zeitreihen

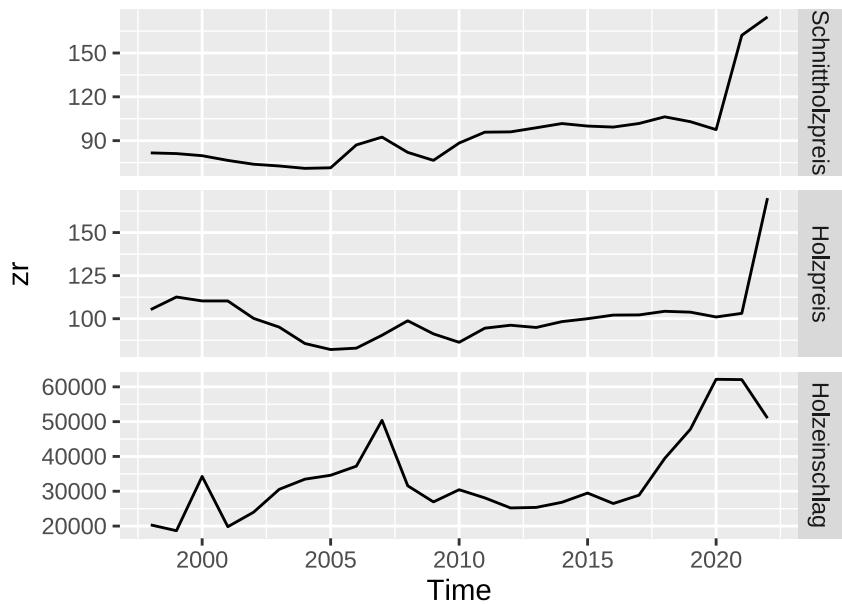
```
plot(zr, ann = FALSE, yax.flip = TRUE)
mtext(side = 3, cex = 1.5, font = 2, "Holzmarktentwicklung seit 1998", line = 2)
mtext(side = 1, cex = 1, line = 2.5, "Jahr")
```

## Holzmarktentwicklung seit 1998



1751 Das Paket `forecast` ermöglicht automatisierte Zeitreihenplots im `ggplot2` Stil. Damit ist auch das Problem  
1752 der y-Achsenbeschriftungen gelöst.

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

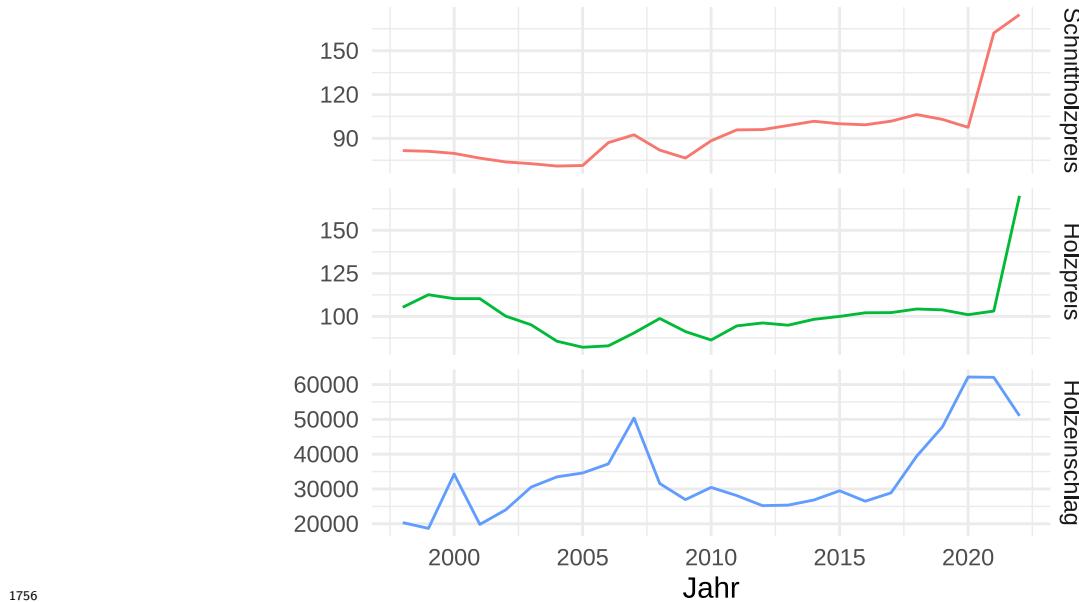


1753

1754 Wir können die Abbildung im `ggplot2` Stil ändern. Hier nur ein Minimalbeispiel zur Veranschaulichung.  
1755 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

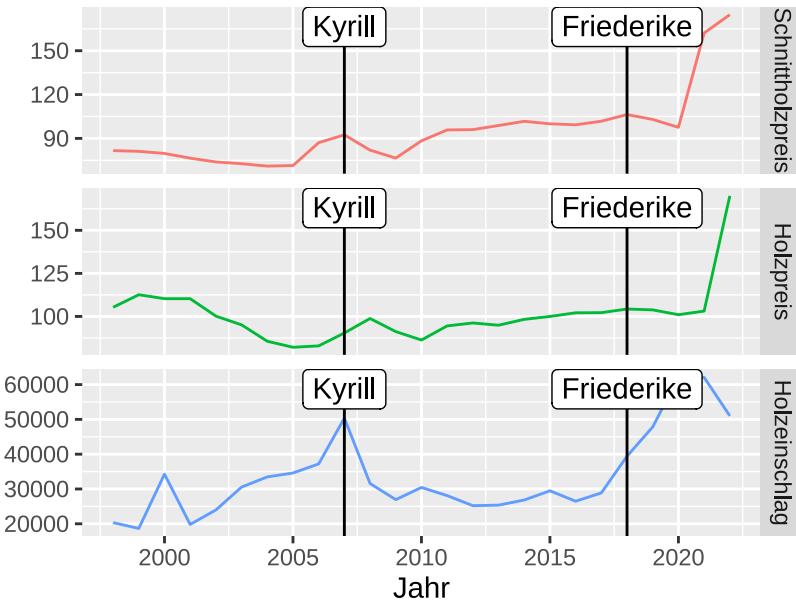
zr_autoplot + theme_minimal()
```



1756

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

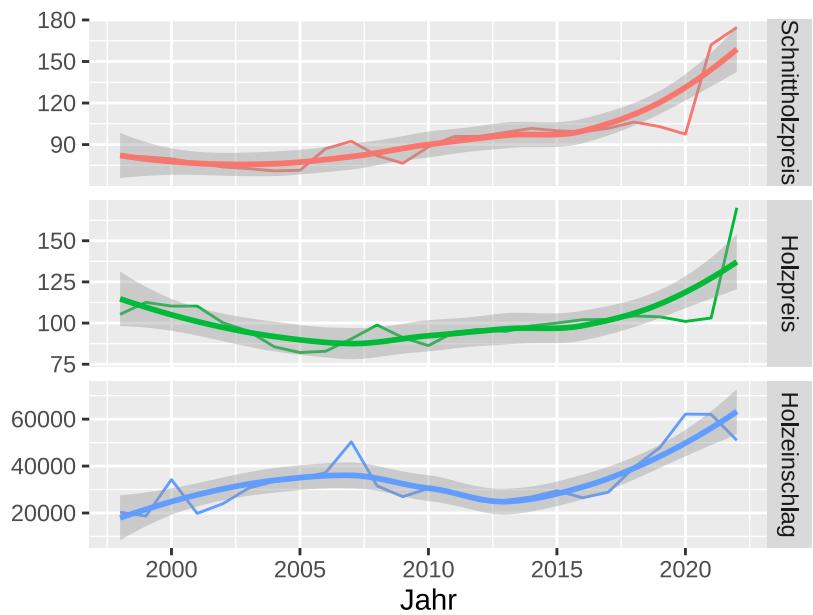
z2 + annotate(x = 2007, y = +Inf, label = "Kyrill", vjust = 1, geom = "label") +
 annotate(x = 2018, y = +Inf, label = "Friederike", vjust = 1, geom = "label")
```



1757

- 1758 Eine Trendlinie macht hier (sowie generell in Zeitreihendaten) offensichtlich keinen Sinn. Daher verwenden wir  
1759 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
```



1760

## 1761 12 Aufgaben Wiederholen (for-Schleifen)

1762 Um einfache Programme zu schreiben, müssen Sie den Ablauf der Programmcodes kontrollieren können.  
 1763 Kontrollieren bedeutet in diesem Zusammenhang, dass Codeabschnitte nur unter definierten Bedingungen ab-  
 1764 laufen. Sie programmieren also zwei Sachen. 1) den Code selbst und 2), die Bedingungen die erfüllt sein müssen,  
 1765 damit der Code ausgeführt wird. Der Code muss so generisch geschrieben sein, dass er komplett durchläuft,  
 1766 auch wenn unterschiedliche Dateneingaben gemacht werden. Diese Kontrollbedingungen ermöglichen es Ihnen  
 1767 generisch zu programmieren. Sie schreiben Ihren Code also nicht speziell maßgeschneidert für ein Problem,  
 1768 sondern so generell, dass er für mehrere Auswertungen funktioniert. Um dies zu gewährleisten, müssen Sie  
 1769 bestimmte Situationen vorhersehen und abfangen. Hierbei helfen Ihnen Kontrollstrukturen (**Control Flow**).  
 1770 Grundsätzlich gibt es **Control Flow** Funktionen zur Wiederholung von Codeblöcken (Schleifen) und logische  
 1771 Bedingungen (bedingte Anweisung).

### 1772 12.1 Schleifen

1773 Bis jetzt wurden alle Skripte einfach der Reihe nach abgearbeitet und zwischendurch bestimmte Programmteile,  
 1774 je nach Situation, selbstständig ausgeführt oder übersprungen. Mit einer Schleife kann man erreichen, dass  
 1775 eine Gruppe von Befehlen (der sog. Schleifenrumpf) mehrfach abgearbeitet wird, zum Beispiel wenn bestimmte  
 1776 Auswertungsschritte auf mehrere Datensätze oder Variablen angewendet oder Funktionen mit unterschiedlichen  
 1777 Parametern oder Startwerten aufgerufen werden sollen. Weitere Anwendungsmöglichkeiten sind iterative  
 1778 Algorithmen, in denen die Eingabewerte des aktuellen Iterationsschrittes von einem vorherigen abhängig sind.  
 1779 Besonders in Simulationen kommen Schleifen häufig zum Einsatz, da große Anzahlen von Wiederholungen  
 1780 benötigt werden.

1781 Man unterscheidet zwischen zwei Arten von Schleifen: Bei den **for()**-Schleifen steht die Anzahl der Wieder-  
 1782 holungen schon beim Eintritt in die Schleife fest, während die **while()**-Schleifen so lange ausgeführt werden,  
 1783 bis eine Bedingung nicht mehr wahr ist. Mit der Funktion **break** wird eine Schleife abgebrochen und die  
 1784 Programmausführung wird nach der Schleife fortgesetzt.

1785 Die wesentlichen Befehle sind

1786 • **for (i in X) {Code}**

1787 Wiederhole den Code im “Schleifenrumpf” für jedes Element aus X.

1788 • **while(Bedingung) {Code}**

1789 Wiederhole den Code im “Schleifenrumpf” so lange die logische Bedingung erfüllt ist.

1790 • **break()**

1791 Brich die Schleife ab. **break()** muss im Schleifenrumpf an der richtigen Stelle ausgeführt werden. Gute  
 1792 Praxis ist jedoch, die for oder while Bedingungen, dass kein **break()**nötig ist, da **break()** anfällig für  
 1793 Programmierfehler ist.

#### 1794 12.1.1 Wiederholen von Befehlen mit **for()**.

1795 Steht vor Beginn der Schleife fest wie viele Schleifendurchgänge benötigt werden, wenn zum Beispiel in einer  
 1796 Simulationen 99 Realisierungen erzeugt oder alle Elemente eines Vektors verarbeitet werden sollen, verwendet

1797 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)
}
```

1798 ## [1] 1

1799 ## [1] 2

1800 ## [1] 3

1801 Das **i** steht in diesem Beispiel für die Schleifen-Variable. Sie muss nicht **i** heißen, sondern kann jeden  
 1802 zulässigen Namen annehmen. Das **X** steht für einen existierenden Vektor oder eine existierende Liste bzw.  
 1803 einen Ausdruck, der ein solches Objekt liefert (der Objektname ist ebenfalls frei wählbar). **for** und **in** sind  
 1804 Schlüsselworte, sie müssen, ebenso wie die runden Klammern, vorhanden sein.

1805 Im ersten Durchgang erhält die Schleifen-Variable **i** den ersten Wert von **X** und der Schleifenrumpf wird  
 1806 mit diesem Wert ausgeführt. Die Variable **i** nimmt nacheinander so lange die Werte von **X** an, bis ihr alle  
 1807 Elemente zugewiesen wurden.

1808 Das folgende Beispiel wird zwar besser durch die entsprechende Vektoroperation gelöst, zeigt aber sehr  
 1809 deutlich die Arbeitsweise der **for**-Schleife.

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

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

1810 ## [1] 4

1811 ## [1] 9

1812 ## [1] 25

1813

### 1814 Aufgabe 36: Schleifen 1

---

1816 Verwenden Sie den Vektor **k** <- c(1, 3, 9, 12, 15) und schreiben Sie folgende **for**-Schleifen:

- 1817 1. Eine Schleife, die jedes Element aus **k** ausgibt.
- 1818 2. Eine Schleife, die zu jedem Element aus **k** 10 addiert und den neuen Wert ausgibt.
- 1819 3. Eine Schleife wie in 2), aber der neue Wert (**k** + 10) soll jetzt nicht mehr ausgegeben werden, sondern  
 1820 in **k10** gespeichert werden. Stellen Sie sicher, dass **k10** wieder von der Länge 5 ist.

1821

---

1822 Die Funktion **for()** ermöglicht es, einen Befehl beliebig oft zu wiederholen. Z.B. der folgende Ausdruck zieht  
 1823 10-Mal eine Stichprobe der Größe 1 aus dem Vektor **v**. Beachten Sie, dass die Schleifen-Variable **i** selbst gar

1824 nicht im Schleifenrumpf vorkommt. Das Ziel dieser Schleife ist nicht die Elemente des Vektors abzuarbeiten,  
 1825 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))
}
```

```
1826 ## [1] 3
1827 ## [1] 1
1828 ## [1] 3
1829 ## [1] 3
1830 ## [1] 2
1831 ## [1] 3
1832 ## [1] 2
1833 ## [1] 2
1834 ## [1] 1
1835 ## [1] 4
```

1836 Auf gleiche Weise kann man auch über die Variablen eines Dataframes iterieren<sup>15</sup>. Das folgende Beispiel hat  
 1837 zum Ziel die Funktionsweise von Schleifen zu verdeutlichen. Schleifen haben in R jedoch den Nachteil, dass sie  
 1838 sehr langsam operieren. Wenn es geht, sollte man Alternativen verwenden. Die Funktionsweise wiederholender  
 1839 Auswertungen wird jedoch mit `for`-Schleifen deutlicher. Aus diesem Grunde werden wir uns in diesem Kurs  
 1840 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)
}
```

```
1841 ## [1] "Buche"
1842 ## [1] 52
1843 ## [1] "Eiche"
1844 ## [1] 64
1845 ## [1] "Eiche"
1846 ## [1] 62
1847 ## [1] "Buche"
1848 ## [1] 85
```

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

1849

---

1850 **Aufgabe 37: for-Schleife**

---

1852 Lesen Sie den Datensatz `bhd_1.txt` ein und verwenden Sie eine `for`-Schleife.

- 1853 • Ziehen Sie 500-Mal je 35 Beobachtungen für den BHD.
- 1854 • Berechnen Sie jeweils den Mittelwert aus den 35 Werten.
- 1855 • Speichern Sie die 500 Mittelwerte in einem neuen Vektor `mittel`.
- 1856 • Wie ist die Verteilung dieser 500 Mittelwerte zu interpretieren?

1857 **12.1.2 Wiederholen von Befehlen mit while()**

1858 Die `while`-Schleifen finden Anwendung, wenn die Anzahl der zu durchlaufenden Wiederholungen vorher  
1859 nicht bekannt ist, wie zum Beispiel bei Iterationsverfahren, die bis zu einer gewissen Genauigkeit durchlaufen  
1860 werden sollen. Die `while`-Schleife besteht in R aus dem Schlüsselwort `while()` und einer Bedingung in runden  
1861 Klammern.

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

1862 Sie ist in der praktischen Programmierung nicht so relevant wie die `for`-Schleife. Sie sei deshalb hier nur  
1863 kurz erwähnt. Die Abbruchbedingung wird jedes Mal geprüft bevor der Schleifenrumpf durchlaufen wird. Die  
1864 Bedingung wird ausgewertet und wenn diese `TRUE` ist, wird der Schleifenrumpf ausgeführt und danach erneut  
1865 die Bedingung überprüft. Ist die Bedingung nicht erfüllt, wird der Schleifenrumpf nicht durchgeführt und die  
1866 Schleife beendet. Ist die Bedingung bereits vor Eintritt in die Schleife nicht erfüllt, wird die Schleife gar nicht  
1867 erst durchlaufen.

1868 Da `while`-Schleifen also so lange ausgeführt werden, bis die Bedingung nicht mehr erfüllt ist, kann eine  
1869 Endlosschleife entstehen. Dies kann passieren, wenn man nicht sauber programmiert hat. Wenn innerhalb der  
1870 Schleife nicht dafür gesorgt wird, dass die Bedingung irgendwann nicht mehr erfüllt wird, so läuft die Schleife  
1871 immer weiter. Steckt R in einer Schleife fest und reagiert nicht mehr, kann der Befehl unter Linux mit `Strg`+  
1872 `C` und unter Windows mit `Esc` abgebrochen werden. Alternativ können Sie auf das rote STOP Symbol  
1873 über der Konsole klicken.

1874 **12.2 Bedingte Ausführung von Codeblöcken**

1875 Innerhalb eines Skripts ist es mitunter notwendig je nach aktueller Situation unterschiedlich fortzufahren.  
1876 Die Situation wird mit einem logischen Ausdruck, einer sogenannten Bedingung, geprüft. Je nachdem, ob  
1877 die Bedingung wahr (`TRUE`) oder falsch (`FALSE`) ist, werden unterschiedliche Programmteile ausgeführt, der  
1878 jeweils andere Teil bleibt unberücksichtigt. Danach wird in jedem Fall die Programmausführung, mit den  
1879 auf die bedingte Anweisungen folgenden Anweisung, fortgesetzt. In R kann dies mit dem if-else-Konstrukt  
1880 realisiert werden, welches aus den Schlüsselwörtern `if()` und `else` sowie der in runde Klammern gefassten  
1881 Bedingung besteht.

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

1882 Im folgenden Beispiel sollen die bisher abstrakt beschriebenen Vorgänge praktische Anwendung finden. In  
 1883 dem Beispiel wird zunächst durch zufälliges Ziehen einer Zahl aus dem Bereich eins bis sechs ein Würfelwurf  
 1884 simuliert. Anschließend wird der Würfelwurf mit einem if-Ausdruck kommentiert. Ist die Bedingung, es wurde  
 1885 eine Sechs gewürfelt, erfüllt, wird der Code innerhalb der geschweiften Klammern ausgeführt, ansonsten wird  
 1886 der Klammerinhalt ignoriert.

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

1887 In den meisten Fällen ist es für R irrelevant, ob sich zwischen den verschiedenen Elementen Leerzeichen oder  
 1888 Zeilenumbrüche befinden. Bei dem else-Ausdruck dagegen wird ein Fehler erzeugt, wenn vor dem `else` nicht  
 1889 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.")
}
```

1890 `## [1] "Beim nächsten Wurf klappt's bestimmt."`

1891

### 1892 Aufgabe 38: Bedingte Programmierung

---

- 1894 • Wenn keine 6 gewürfelt wurde, lassen Sie zusätzlich ausgeben welche Zahl gewürfelt wurde.  
 1895 • Wiederholen Sie den Würfelwurf 10 Mal.

## 1896 13 (R)markdown

### 1897 13.1 Markdown Grundlagen

1898 Die Idee von Markdown ist, einfach Text strukturieren zu können und das ganze ohne umfangreiche Programme  
 1899 zu erstellen. Im nächsten Abschnitt sehen wir dann, wie man Markdown und R-Code verbinden kann. Hier  
 1900 soll es jedoch erst einmal darum gehen, die einfachsten Bausteine von Markdown vorzustellen.

1901 Am Anfang jedes Dokuments kommt eine Präambel. Diese fängt mit --- an und hört auch wieder mit ---  
 1902 auf. Innerhalb der Präambel können dann Metainformationen über das Dokument festgelegt werden, dies  
 1903 beinhaltet im einfachsten Fall: Titel, Autor und Datum. Das würde dann so aussehen:

```
1904 ---
1905 title: "Ein Titel"
1906 author: "Der, der es geschrieben hat"
1907 date: "März 2021"
1908 ---
```

1909 Danach folgt strukturierter und formatierter Text. Verschiedene Hierarchieebenen von Überschriften können  
 1910 mit der Anzahl an # festgelegt werden. So ist eine Überschrift erster Ordnung # Kapitel eine Überschrift  
 1911 zweiter Ordnung ## Unterkapitel usw.

1912 Listen können erstellt werden, wenn man am Anfang jeder Zeile ein - oder 1. schreibt.

```
1913 - Erster Eintrag
1914 - Zweiter Eintrag
1915 - Dritter Eintrag
```

1916 wird zu

```
1917 • Erster Eintrag
1918 • Zweiter Eintrag
1919 • Dritter Eintrag
```

1920 Eine zentrale Idee von Markdown ist es Text einfach zu formatieren. Werden eine oder mehrere Wörter mit  
 1921 zwei Sternchen (\*\*) eingefasst wird dieser Text **fett** dargestellt. Also aus \*\*wichtig\*\* wird **wichtig**. Das  
 1922 gleiche funktioniert auch mit *kursiven* Text, jedoch muss man hier noch ein Sternchen verwenden, also aus  
 1923 \*kursiv\* wird *kursiv*. Soll ein text fett und kursiv sein, kann man drei Sternchen verwenden. Aus \*\*\*sehr  
 1924 wichtig\*\*\* wird dann **sehr wichtig**.

1925 Weitere Elemente wie Links oder Abbildungen können einfach eingebunden werden. Links werden mit [Link  
 1926 text](url) in den Text integriert. Beispielsweise ist eine gute Idee bei [stackoverflow](#) bei Problemen nach  
 1927 einer Lösung zu suchen. Dieser Link wurde mit [stackoverflow](www.stackoverflow.com) erstellt.

1928 Für Abbildungen gibt es einen ganz ähnlichen Syntax. Mit ! [Das R Logo](abb/r\_logo.png) wird die  
 1929 Abbildung r\_logo.png eingebunden mit der Beschriftung: "Das R Logo".



Abbildung 9: Das R Logo

1930

1931 **Aufgabe 39: Arbeiten mit markdown**

1932

1933 Verwenden Sie das folgende Markdowndokument:

1934 ---

1935 title: "Dokument"  
1936 author: "Ihr Name"  
1937 date: "März 2021"

1938 ---

1939

1940 # Einleitung  
1941  
1942 # Methoden

- 1943 1. Kopieren Sie die Vorlage in ein Dokument, das `test.md` heißt.
- 1944 2. Fügen Sie zwei Überschriften zweiter und dritter Ordnung hinzu.
- 1945 3. Fügen Sie einen *kursiven* Text hinzu.
- 1946 4. Fügen Sie einen Link zu einer Website hinzu.
- 1947 5. Kompilieren Sie die Datei, indem Sie in Rstudio auf **Preview** drücken (Abbildung 10).

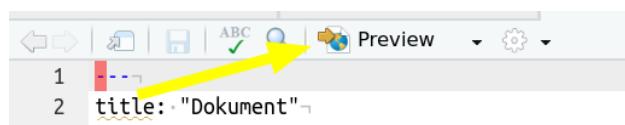


Abbildung 10: Kompilieren einer md-Datei.

1948 **13.2 R und Markdown**

1949 Markdown macht es bereits einfach Textdokumente und Dokumentationen zu verfassen, aber die wirkliche  
 1950 Stärke liegt in der Möglichkeit R und Markdown zu kombinieren. Man spricht dann von Rmarkdown. Ein  
 1951 weiteres Strukturelement, das wir noch nicht kennen gelernt haben, sind Code-Blöcke.

1952 ~~~

1953 a &lt;- 1:10

```

1954 a[1]
1955 ``
1956 erzeugt
1957 a <- 1:10
1958 a[1]

1959 Momentan wird noch kein Code ausgeführt, sondern lediglich der Code, als Code dargestellt. Rmarkdown
1960 bietet nun die Möglichkeit Code beim kompilieren16 auszuführen. Dafür müssen wir nur einen Code-Block als
1961 R-Code-Block kennzeichnen.

1962 ``{R}
1963 a <- 1:10
1964 a[1]
1965 ```

1966 erzeugt
1967 a <- 1:10
1968 a[1]

1967 ## [1] 1

1968 Beachte, die Variable a wird beim kompilieren erzeugt und steht dann R zur Verfügung. R-Code-Blöcke
1969 werden auch als Code Chunks bezeichnet. Diese Chunks können sehr genau angesprochen und angepasst
1970 werden. Einige wichtige Argumente sind:

1971 • echo: Gibt an, ob der Quelltext angezeigt werden soll oder nicht.
1972 • result: Gibt an, ob die Ergebnisse gezeigt werden sollen oder nicht.
1973 • eval: Diese Option gibt an ob der Chunk ausgeführt werden soll oder nicht.

```

1974

---

**Aufgabe 40: Arbeiten mit Rmarkdown**


---

1975 Erstellen Sie eine neue Rmarkdown Datei mit dem Namen `test1.Rmd`. Erstellen Sie zwei Code-Chunks. Der  
1976 erste soll nicht angezeigt werden und darin werden die Daten geladen (`bhd_1.txt`). Im zweiten Chunk plotten  
1977 Sie das Alter der Bäume gegen den BHD. Was passiert mit dem Plot, wenn Sie die Datei kompilieren (drücken  
1978 Sie dazu auf den Knit-Knopf; Abbildung 11).



Abbildung 11: Kompilieren einer `Rmd`-Datei.

---

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

## 1981 14 Räumliche Daten in R

### 1982 14.1 Was sind räumliche Daten

1983 Räumliche Daten sind Beobachtungen, wie wir sie schon oft gesehen haben, mit einem räumlichen Bezug. Der  
 1984 Unterschied zu nicht räumlichen Daten liegt darin, dass räumliche Daten eindeutig im *Raum* verortet werden  
 1985 können. Häufig werden sogenannte Geoinformationssysteme (GIS) zum Arbeiten mit räumlichen verwendet.  
 1986 R kann in vielerlei Hinsicht wie ein GIS eingesetzt werden und hier werden einige Grundfunktionalitäten  
 1987 dafür besprochen. Räumliche Daten werden in zwei unterschiedliche Datentypen unterteilt: Vektor- und  
 1988 Rasterdaten. Vektordaten modellieren einzelne Objekte (= *Features*). Rasterdaten modellieren eine Oberfläche.  
 1989 Vektordaten bestehen aus zwei Komponenten: 1) einer Geometrie, die die Form und Lage der Daten definiert  
 1990 und 2) Attributen, den tatsächlichen Daten. Räumliche Daten werden oft als *Features* bezeichnet. Ein Feature  
 1991 ist die räumliche Einheit einer Beobachtung. Je nach Art der räumlichen Daten können Features entweder  
 1992 Punkte (z.B. ein Baum), Linien (z.B. eine Straße) oder Polygone (z.B. ein See) sein. Auch können mehrere  
 1993 Geometrien zu einem Feature zusammengefasst werden. Ein Beispiel wäre eine Beobachtung für ein Land,  
 1994 das aber aus mehreren Polygonen bestehen kann (z.B. Festland und Inseln). Features können dann weitere  
 1995 Attribute (= Attributdaten) haben, z.B. eine ID, Name oder was auch immer man gemessen hat.  
 1996 Rasterdaten bestehen aus einer Oberfläche von gleichgroßen Kacheln (= *Pixel*), die ein Gebiet abdecken.  
 1997 Meist sind Pixel viereckig, aber das ist keine Voraussetzung. Dabei hat jedes Pixel einen Wert (das kann  
 1998 auch ein fehlender Wert sein). Typische Beispiele für Rasterdaten sind Landnutzung oder Seehöhen.  
 1999 In R kann sowohl mit Vektor- als auch mit Rasterdaten gearbeitet werden. Für Vektordaten bietet sich das  
 2000 Paket **sf** an und für Rasterdaten das Paket **raster**.

### 2001 14.2 Koordinatenbezugssystem

2002 Eine der Herausforderungen für räumliche Daten ist die eindeutige Verortung im Raum. Dazu braucht man  
 2003 ein *Koordinatenbezugssystem (KBS)*. Einem KBS liegt ein mathematisches Modell der Erde zugrunde. Die  
 2004 Details zu KBS werden schnell relativ kompliziert und wir beschränken uns hier lediglich darauf, wie KBS  
 2005 verwendet werden können. Dazu müssen wir zwei Fälle unterscheiden: 1) einem Datensatz ein KBS zuweisen  
 2006 und 2) Transformation des KBS eines Datensatzes in ein anderes KBS. Die technischen Details werden in  
 2007 den folgend Abschnitten besprochen. Für beide Aufgaben müssen wir auf ein KBS verweisen können, ein  
 2008 einfacher Ansatz dafür sind die sogenannten *EPSG-Codes*<sup>17</sup>.

### 2009 14.3 Vektordaten in R

2010 Das Paket **sf** stellt Klassen zum Abbilden von Features zur Verfügung, die dann in einem **data.frame** als  
 2011 Liste gespeichert werden können. In der Regel erstellen wir Features nicht individuell, sondern lesen diese aus  
 2012 externen Datenquellen (z.B. ESRI Shapefile) ein. Zum besseren Verständnis, erstellen wir es einmal manuell.  
 2013 Wir haben die Koordinaten für drei Städte (Göttingen, Hannover und Berlin) als geografische Koordinaten  
 2014 vorliegen (EPSG = 4326).

---

<sup>17</sup>EPSG 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))
```

2015 Daraus können wir jetzt eine Geometriespalte für einen `data.frame` erstellen

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

2016 Somit haben wir die Geometrie in der Variable `geom` gespeichert, aber noch keine dazugehörigen Attributdaten.

2017 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)
)
```

2018 In einem letzten Schritt möchten wir jetzt die Geometrie (`geom`) und die Attributdaten (`attr`) zusammenführen.

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

```
2020 ## Simple feature collection with 3 features and 3 fields
2021 ## Geometry type: POINT
2022 ## Dimension: XY
2023 ## Bounding box: xmin: 9.732 ymin: 51.5413 xmax: 13.405 ymax: 52.52
2024 ## Geodetic CRS: WGS 84
2025 ## name bundesland einwohner geom
2026 ## 1 Goettingen Niedersachsen 119000 POINT (9.9158 51.5413)
2027 ## 2 Hannover Niedersachsen 532000 POINT (9.732 52.3759)
2028 ## 3 Berlin Berlin 3650000 POINT (13.405 52.52)
```

2029 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.

2031 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)
)
```

2034 Dann kann man mit der Funktion `st_as_sf()` weiter arbeiten:

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

## 2035 14.4 Arbeiten mit Vektordaten

2036 Es gibt sehr viele Funktionen, um mit räumlichen Daten zu arbeiten, von denen wir hier einige vorstellen.

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

```
2037 ## Coordinate Reference System:
2038 ## User input: EPSG:4326
2039 ## wkt:
2040 ## GEOGCRS["WGS 84",
2041 ## ENSEMBLE["World Geodetic System 1984 ensemble",
2042 ## MEMBER["World Geodetic System 1984 (Transit)"],
2043 ## MEMBER["World Geodetic System 1984 (G730)"],
2044 ## MEMBER["World Geodetic System 1984 (G873)"],
2045 ## MEMBER["World Geodetic System 1984 (G1150)"],
2046 ## MEMBER["World Geodetic System 1984 (G1674)"],
2047 ## MEMBER["World Geodetic System 1984 (G1762)"],
2048 ## MEMBER["World Geodetic System 1984 (G2139)"],
2049 ## ELLIPSOID["WGS 84",6378137,298.257223563,
2050 ## LENGTHUNIT["metre",1]],
2051 ## ENSEMBLEACCURACY[2.0]],
2052 ## PRIMEM["Greenwich",0,
2053 ## ANGLEUNIT["degree",0.0174532925199433]],
2054 ## CS[ellipsoidal,2],
2055 ## AXIS["geodetic latitude (Lat)",north,
2056 ## ORDER[1],
2057 ## ANGLEUNIT["degree",0.0174532925199433]],
2058 ## AXIS["geodetic longitude (Lon)",east,
2059 ## ORDER[2],
2060 ## ANGLEUNIT["degree",0.0174532925199433]],
2061 ## USAGE[
2062 ## SCOPE["Horizontal component of 3D system."],
2063 ## AREA["World."],
2064 ## BBOX[-90,-180,90,180]],
2065 ## ID["EPSG",4326]]
```

2066 Wenn wir jetzt zu einem anderen KBS (z.B. EPSG:3035, ein europäisches projiziertes KBS) umrechnen  
 2067 möchten, können wir das mit

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

```

2068 ## Coordinate Reference System:
2069 ## User input: EPSG:3035
2070 ## wkt:
2071 ## PROJCRS["ETRS89-extended / LAEA Europe",
2072 ## BASEGEOGCRS["ETRS89",
2073 ## ENSEMBLE["European Terrestrial Reference System 1989 ensemble",
2074 ## MEMBER["European Terrestrial Reference Frame 1989"],
2075 ## MEMBER["European Terrestrial Reference Frame 1990"],
2076 ## MEMBER["European Terrestrial Reference Frame 1991"],
2077 ## MEMBER["European Terrestrial Reference Frame 1992"],
2078 ## MEMBER["European Terrestrial Reference Frame 1993"],
2079 ## MEMBER["European Terrestrial Reference Frame 1994"],
2080 ## MEMBER["European Terrestrial Reference Frame 1996"],
2081 ## MEMBER["European Terrestrial Reference Frame 1997"],
2082 ## MEMBER["European Terrestrial Reference Frame 2000"],
2083 ## MEMBER["European Terrestrial Reference Frame 2005"],
2084 ## MEMBER["European Terrestrial Reference Frame 2014"],
2085 ## ELLIPSOID["GRS 1980",6378137,298.257222101,
2086 ## LENGTHUNIT["metre",1]],
2087 ## ENSEMBLEACCURACY[0.1]],
2088 ## PRIMEM["Greenwich",0,
2089 ## ANGLEUNIT["degree",0.0174532925199433]],
2090 ## ID["EPSG",4258]],
2091 ## CONVERSION["Europe Equal Area 2001",
2092 ## METHOD["Lambert Azimuthal Equal Area",
2093 ## ID["EPSG",9820]],
2094 ## PARAMETER["Latitude of natural origin",52,
2095 ## ANGLEUNIT["degree",0.0174532925199433],
2096 ## ID["EPSG",8801]],
2097 ## PARAMETER["Longitude of natural origin",10,
2098 ## ANGLEUNIT["degree",0.0174532925199433],
2099 ## ID["EPSG",8802]],
2100 ## PARAMETER["False easting",4321000,
2101 ## LENGTHUNIT["metre",1],
2102 ## ID["EPSG",8806]],
2103 ## PARAMETER["False northing",3210000,
2104 ## LENGTHUNIT["metre",1],
2105 ## ID["EPSG",8807]]],
2106 ## CS[Cartesian,2],
2107 ## AXIS["northing (Y)",north,
2108 ## ORDER[1],
2109 ## LENGTHUNIT["metre",1]],
2110 ## AXIS["easting (X)",east,

```

```

2111 ## ORDER[2],
2112 ## LENGTHUNIT["metre",1]],
2113 ## USAGE[
2114 ## SCOPE["Statistical analysis."],
2115 ## AREA["Europe - European Union (EU) countries and candidates. Europe - onshore and offshore: A
2116 ## BBOX[24.6,-35.58,84.73,44.83]],
2117 ## ID["EPSG",3035]

```

2118 Die Funktion `st_buffer()` erlaubt es Features zu puffern, mit `st_distance()` kann die Distanz zwischen  
2119 Features berechnet werden, mit `st_area()` kann die Fläche eines Features berechnet werden.

2120 Funktionen wie `st_intersection()`, `st_union()` und `st_difference()` erlauben es geometrische Opera-  
2121 tionen zwischen unterschiedlichen Features zu berechnen. Für eine ausführliche Diskussion siehe auch hier:  
2122 <https://geocompr.robinlovelace.net/geometric-operations.html>.

2123 Normalerweise lesen wir Daten von externen Datenquellen (z.B. ESRI Shapefiles). Das geht mit der Funktion  
2124 `st_read()`.

## 2125 14.5 Rasterdaten in R

2126 Für Rasterdaten gibt es das R-Paket `raster`. Auch hier wollen wir uns wieder auf einige Grundfunktionalitäten  
2127 konzentrieren. Diese umfassen das Einlesen, Zuschneiden, Rechnen und Abfragen von Rastern.

2128 Mit der Funktion `raster()` kann ein Raster in R eingelesen werden.

```

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

```

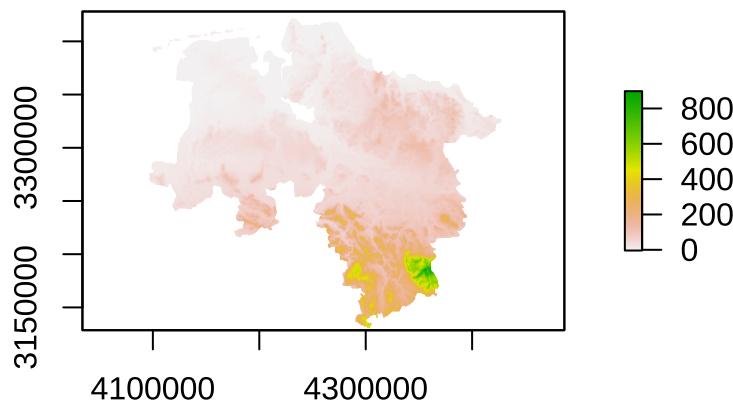
2129 `dem` steht für *Digital Elevation Model* und ist ein Raster mit den Seehöhen in Niedersachsen mit einer  
2130 500-m-Auflösung. Wir können diese mit der Funktion `res()`<sup>18</sup> abfragen.

```
res(dem)
```

2131 ## [1] 500 500

2132 Bzw. wir können den Raster auch plotten.

```
plot(dem)
```



2133

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

2134 Wenn wir den Raster `dem` auf ein Gebiet zuschneiden wollen (z.B. Göttingen), müssen wir drei Schritte  
2135 durchführen. Als erstes müssen wir ein Shapefile für Göttingen einlesen.

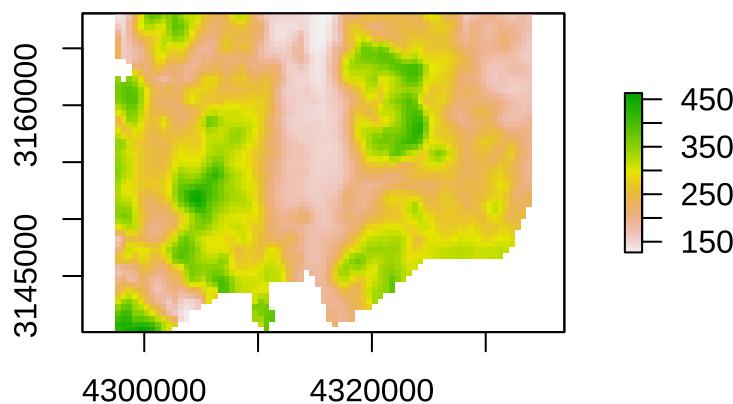
```
goe <- st_read(here::here("data/goettingen/stadt_goettingen.shp"))
```

2136 Dann müssen wir sicherstellen, dass sowohl der Raster `dem` als auch das `sf`-Objekt `goe` im selben KBS sind.  
2137 Es bietet es sich in der Regel an, das KBS des Vektors zu transformieren. Mit der Funktion `projectRaster()`  
2138 kann das KBS eines Rasters transformiert werden.

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

2139 Mit der Funktion `crop()` kann der Raster `dem` auf Göttingen zugeschnitten werden.

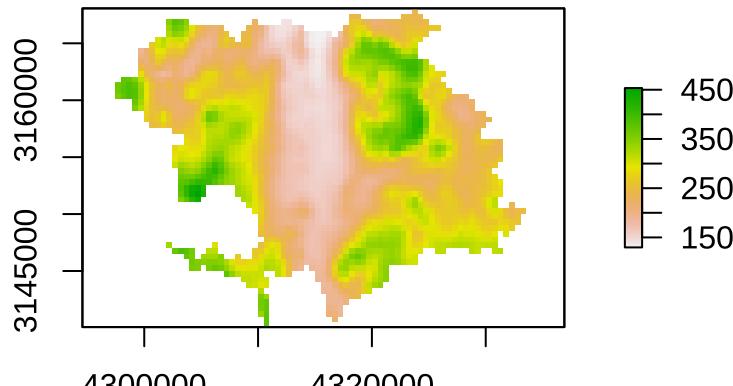
```
dem1 <- crop(dem, goe)
plot(dem1)
```



2140

2141 Der Raster hat jetzt die Größe einer Bounding-Box (BBX) von Göttingen (das ist ein Rechteck, das Göttingen  
2142 umfasst). Mit der Funktion `mask()` kann der Raster auf die genauen Grenzen des Vektors `goe` angepasst  
2143 werden.

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



2144

2145 Wenn wir an bestimmten Punkten den Wert des Rasters abfragen wollen (z.B. an `cities`) von vorhin, dann  
2146 gibt es dafür die Funktion `extract`. Dann müssen wir erst sicherstellen, dass `staedte` und `dem` gleichen  
2147 KBS zu grunde liegt. Dafür transformieren wir einfach `staedte` in das KBS von `dem`. Mit der Funktion  
2148 `projection()` erhalten wir das KBS des Rasters.

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

2149 Wenn wir das KBS eines Objektes nicht kennen, können wir auch einfach das KBS übergeben. Der folgende  
 2150 Code-Block macht genau das Gleiche mit dem Vorteil, dass wir keinen EPSG-Code angeben müssen.

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

2151 Dann können wir für jede Stadt die Seehöhe abfragen:

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

2152 ## [1] 149.18181 57.21486 NA

2153 Mit `raster::extract()` rufen wir *eindeutig* die Funktion `extract()` aus dem Paket `raster` auf. Wir müssen  
 2154 das so machen, weil es im Paket `dplyr` auch eine Funktion `extract()` gibt, die wir hier nicht anwenden  
 2155 möchten, da sie einen Fehler verursachen würde.

2156 Ein analoges Vorgehen ist auch für Linien und Polygone möglich.

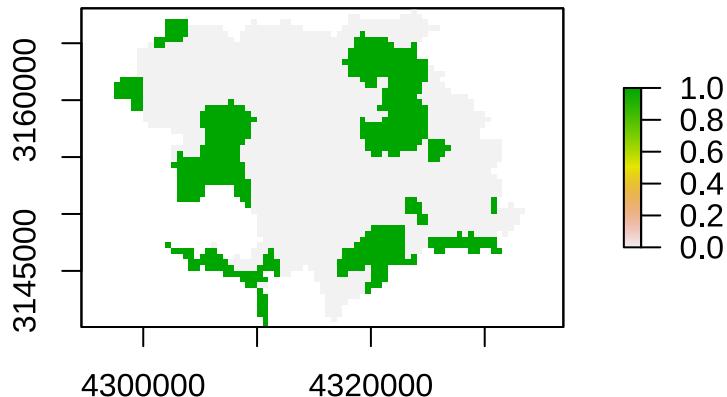
2157 Mit Rastern kann auch einfach gerechnet werden. Wir können z.B. die Seehöhe in Kilometern anstatt Metern  
 2158 berechnen:

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

2159 Auch logische Operationen sind möglich, wenn wir alle Rasterzellen mit einer Seehöhe von mehr als 300 m in  
 2160 Göttingen suchen, dann geht das so:

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

```
plot(dem3)
```



2161

2162 Wenn wir jetzt auf die Werte des Rasters `dem3` zugreifen wollen, geht das mit eckigen Klammern.

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

2163 ## [1] NA NA NA NA NA NA

2164 Das sind erst einmal viele `NA`-Werte für die ganzen Zellen, die außerhalb von Niedersachsen liegen. Aber wir  
 2165 können mit so einem Vektor ganz normal arbeiten und z.B. die Fläche des Landes Niedersachsen die eine  
 2166 Seehöhe von mehr als 500m Seehöhe hat ausrechnen.

```

2165 h <- dem3[]
2166 sum(h, na.rm = TRUE) / sum(!is.na(h))
2167 ## [1] 0.265713
2168

```

---

**Aufgabe 41: Arbeiten mit Rastern**

---

2171 Verwenden Sie den Raster `wald.tif`, der auf einer 10 m Auflösung den Waldanteil jeder Rasterzelle angibt<sup>19</sup>.  
 2172 Der EPSG-Code für das KBS von `wald.tif` ist 3035. Nehmen Sie an, dass wenn der Waldanteil in einer  
 2173 Raster größer als 50 % ist, dass die Rasterzelle als Wald klassifiziert werden kann. Wie viel Prozent des  
 2174 Göttinger Stadtgebietes sind Wald? Wie ändert sich dieser Wert, wenn sie 70 % anstatt 50 % als Schwellenwert  
 2175 für Wald annehmen?

2176

---

**Aufgabe 42: Studiendesign**

---

2177 Mit der Funktion `st_sample()` können Sie innerhalb oder entlang eines Features zufällige Punkte legen. Das  
 2178 Argument `n` steuert die Anzahl Punkte und das Argument `type` wie die Punkte angeordnet werden. Für `type`  
 2179 sind für uns die Werte `type = "random"` (komplett zufällig), `type = "regular"` (regelmäßiger Grid) und  
 2180 `type = "hexagonal"` von Bedeutung (ein hexagonaler Grid, d.h. ein sechseckiger Raster). Unglücklicherweise  
 2181 ist das Ergebnis von `st_sample()` erst eine Geometrie. Um daraus ein vollständiges `sf`-Objekt zu machen  
 2182 und problemlos weiter arbeiten zu können, müssen Sie nocheinmal die Funktion `st_as_sf()` ausführen.  
 2183  
 2184 Stellen Sie sich vor, dass wir die tatsächliche Waldbedeckung des Göttinger Stadtgebietes **nicht** kennen und wir  
 2185 eine Studie durchführen, um den Anteil des Göttinger Stadtgebietes, der mit Wald bedeckt ist herauszufinden.  
 2186 Erstellen Sie dafür einige unterschiedliche Stichproben (diese können in der Anzahl und Anordnung variieren).  
 2187  
 2188 Berechnen Sie für jedes Stichprobendesign den Anteil an Wald und ein dazugehöriges Konfidenzintervall  
 2189 (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  $n$   
 2190 die Stichprobengröße). Nehmen Sie an, dass eine Rasterzelle Wald ist, sobald  $> 50\%$  der Rasterzelle mit  
 2191 Wald bedeckt ist.

2192

---

**Aufgabe 43: Räumliche Daten**

---

2193 Verwenden Sie den folgenden Datensatz:

```

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

```

<sup>19</sup>Die 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)
)

```

- 2196 1. Erstellen Sie ein `sf`-Objekt aus `df1`.  
 2197 2. Puffern Sie jeden Baum mit seinem Kronendurchmesser.  
 2198 3. Berechnen Sie die Kronenfläche jedes Baumes. *Hinweis: Die Funktion `st_area()` könnte dafür hilfreich sein.*  
 2199 4. Welcher Baum hat die größte Kronenfläche?  
 2200 5. Finden Sie für jede Art, den Baum mit der größten Kronenfläche.

2202

**Aufgabe 44: Arbeiten mit räumlichen Daten**

- 2203 1. Lesen Sie das ESRI Shapefile `goettingen/stadt_goettingen.shp` ein.  
 2204 2. Wie viele Features befinden sind in dem Shapefile?  
 2205 3. Welches Koordinatenbezugssystem (KBS) hat das Shapefile?  
 2206 4. Transformieren Sie das Shapefile in das KBS 3035.  
 2207 5. Erstellen Sie eine neue Spalte `A` in der Sie die Fläche jeder Gemeinde/Stadt speichern.  
 2208 6. Welche Gemeinde/Stadt (Spalte `GEN`) ist am größten?  
 2209 7. Wählen Sie nun nur die Stadt Göttingen aus.

2212

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

- 2213 1. Lesen Sie erneut das ESRI Shapefile `goettingen/stadt_goettingen.shp` ein.  
 2214 2. Lösen sie die Gemeindegrenzen auf (die Funktion `st_union()` könnte hier nützlich sein).  
 2215 3. Wie groß ist das resultierende Feature?

2218 **15 FAQs (Oft gefragtes)**

2219 **15.1 Arbeiten mit Daten**

2220 **15.1.1 Einlesen von Exceldateien**

2221 Mit der Funktion `read_excel()` aus dem Paket `readxl` können Exceldateien direkt in R eingelesen werden.

2222 Ein Export als csv-Datei aus Excel ist nicht notwendig.

## 2223 16 Literatur

- 2224 Ein guter Überblick über viele der angeschnittenen Themen gibt es in dem [R for data science](#), das online frei  
2225 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.  
2226
- 2227 McNamara, Amelia, and Nicholas J Horton. 2018. “Wrangling Categorical Data in r.” *The American Statistician*  
2228 72 (1): 97–104.
- 2229 Wickham, Hadley. 2014. “Tidy Data.” *Journal of Statistical Software* 59 (10). <https://doi.org/10.18637/jss.v059.i10>.
- 2230