

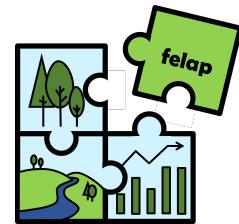
1

Einführung in die Datenanalyse mit R (700104)

4

Kursskript

5



6

Dr. Johannes SIGNER
Abteilung Wildtierwissenschaften
Büsgenweg 3
37077 Göttingen

jsigner@uni-goettingen.de

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

kai.husmann@uni-goettingen.de

7



8

Fakultät für Forstwissenschaften und Waldökologie
Georg-August-Universität Göttingen

10



11

12

Wintersemester 2023/2024

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

¹⁵ Zitiervorschlag:

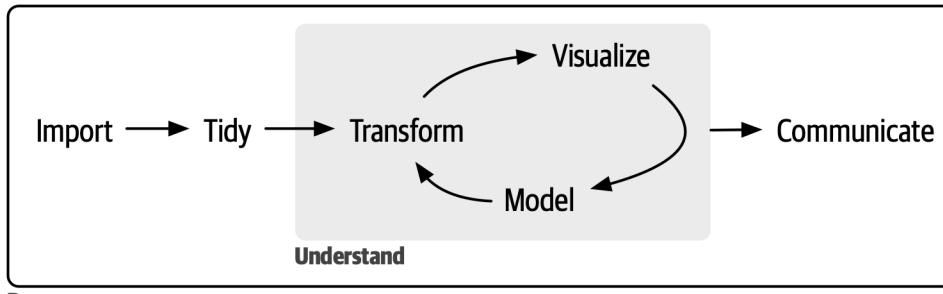
¹⁶ Signer, J. und Husmann, K. (2024) Skript zur Vorlesung Einführung in die Datenanalyse mit R, Georg-August-Universität Göttingen.

¹⁸ Letzte Aktualisierung: 9. Oktober 2024

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	1.4 RStudio Projekte	6
50	1.4.1 Erstellen eines Projektes	6
51	2 Variablen, Funktionen und Datentypen	8
52	2.1 Variablen beim Programmieren	8
53	2.2 Funktionen	9
54	2.3 Datentypen	10
55	2.4 Datenstrukturen	11
56	3 Vektoren	13
57	3.1 Funktionen zum Arbeiten mit Vektoren	15
58	3.2 Statistische Funktionen	16
59	3.3 Beispiel Fotofallen	17
60	3.4 Arbeiten mit logischen Werten	18
61	3.5 Zugreifen auf Elemente eines Vektors (=Untermengen)	19
62	3.6 Der %in%-Operator	21
63	4 Faktoren (factors)	23
64	4.1 Das Paket forcats	24
65	4.1.1 Anpassen der Anordnung von Faktoren	25
66	5 Spezielle Einträge	26
67	5.1 NA	26
68	5.2 NULL	27
69	5.3 Inf	27
70	6 data.frames oder Tabellen	29
71	6.1 Wichtige Funktionen zum Arbeiten mit data.frames	30
72	6.2 Zugreifen auf Elemente eines data.frame	31
73	7 Schreiben und lesen von Daten	34
74	7.1 Textdateien	34
75	8 Erstellen von Abbildungen	36
76	8.1 Base Plot	36
77	8.1.1 Mehrere Panels	42
78	8.1.2 Speichern von Abbildungen	42
79	8.2 Histogramme	43
80	8.3 Boxplots	46

81	8.4 ggplot2: Eine Alternative für Abbildungen	48
82	8.4.1 Multipanel Abbildungen	55
83	8.4.2 Plots kombinieren	58
84	8.4.3 Speichern von plots	60
85	9 Mit Daten arbeiten	62
86	9.1 dplyr eine Einführung	62
87	9.2 Arbeiten mit gruppierten Daten	65
88	9.3 pipes oder %>%	66
89	9.4 Joins	67
90	9.5 'long' and 'wide' Datenformate	69
91	9.6 Auswählen von Variablen	71
92	9.7 Einzelne Beobachtungen abfragen (slice())	72
93	9.8 Spalten trennen	75
94	10 Arbeiten mit Text	77
95	10.1 Arbeiten mit Text	77
96	10.2 Finden von Textmustern	78
97	11 Arbeiten mit Zeit	81
98	11.1 Arbeiten mit Zeitintervallen	82
99	11.2 Formatieren von Zeit	84
100	11.3 Zeitreihen	84
101	12 Aufgaben Wiederholen (for-Schleifen)	90
102	12.1 Schleifen	90
103	12.1.1 Wiederholen von Befehlen mit for()	90
104	12.1.2 Wiederholen von Befehlen mit while()	93
105	12.2 Bedingte Ausführung von Codeblöcken	93
106	13 (R)markdown	95
107	13.1 Markdown Grundlagen	95
108	13.2 R und Markdown	96
109	14 Räumliche Daten in R	98
110	14.1 Was sind räumliche Daten	98
111	14.2 Koordinatenbezugssystem	98
112	14.3 Vektordaten in R	98
113	14.4 Arbeiten mit Vektordaten	100
114	14.5 Rasterdaten in R	102
115	15 FAQs (Oft gefragtes)	108
116	15.1 Arbeiten mit Daten	108
117	15.1.1 Einlesen von Exceldateien	108
118	16 Literatur	109

1 R und RStudio

1.1 Installation von R und RStudio

- Als ersten Schritt müssen Sie R und RStudio installieren. Dabei ist wichtig zu unterscheiden, dass R und RStudio zwei unterschiedliche Programme sind. R ist die eigentliche Programmiersprache mit der wir arbeiten. RStudio hingegen ist eine sogenannte Entwicklungsumgebung¹, die das Arbeiten mit R vereinfachen soll.
- Sie können also mit R arbeiten ohne RStudio (was unüblich ist), aber nicht mit RStudio ohne R. RStudio wird unter anderem verwendet, um R Code komfortabler zu schreiben und zu verwalten.
- 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**).

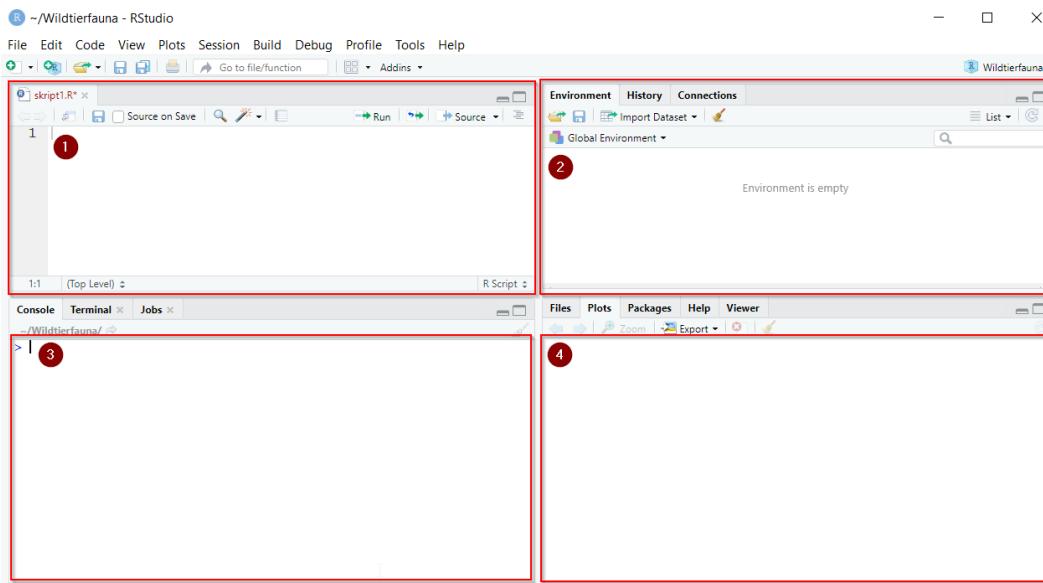


Abbildung 1: RStudio Panes.

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

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

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

10 + 5

154 **## [1] 15**

20 - 10

155 **## [1] 10**

10 * 3

156 **## [1] 30**

100 / 19

157 **## [1] 5.263158**

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

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

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

¹⁷² Zeile abschicken, indem wir entweder auf *Run* klicken (Abbildung 2) oder die Tastenkombination *Strg + Enter* (*Strg*+*Esc*) tippen. Mehrere Zeilen abzuschicken und nacheinander ausführen zu lassen ist möglich,
¹⁷³ indem diese Zeilen markiert werden bevor Sie *Run* klicken oder die Tastenkombination tippen. Ein Klick auf
¹⁷⁴ *Source* bzw. die Tastenkombination *Strg + Umschalt + Enter* (*Strg*+*Shift*+*Esc*).
¹⁷⁵

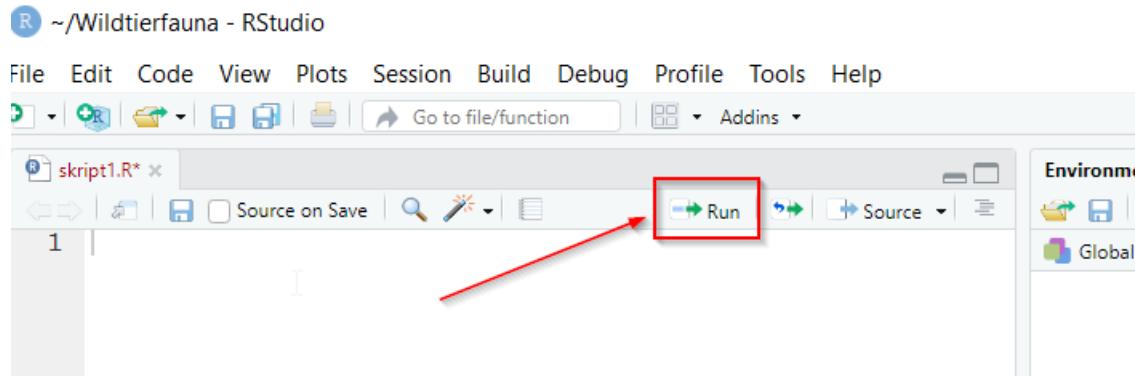


Abbildung 2: Zeilenweises Ausführen von Code in RStudio.

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

1.3 Gute Praxis bei der Programmierung

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

¹⁸⁸ Vielleicht die wichtigste Praxis beim Programmieren ist das Kommentieren. **Kommentare** sind ein wichtiger
¹⁸⁹ Bestandteil der Skripte in R und allen anderen Skript- und Programmiersprachen. Sie werden lernen, dass die
¹⁹⁰ Kommentarfunktion ein wesentlicher Vorteil gegenüber Klickprogrammen darstellt. Ein Kommentar ist Text
¹⁹¹ in einem (R-)Code, der der Dokumentation dient und von der R Konsole ignoriert wird. Sämtliche Zeilen, die
¹⁹² mit dem Zeichen # beginnen, werden nicht ausgeführt, wenn Sie an die Konsole gesendet werden. Seien Sie
¹⁹³ nicht sparsam mit Kommentaren, sondern benutzen Sie sie um Ihren Code zu strukturieren, ihre Berechnungen
¹⁹⁴ zu für sich selbst und andere zu erläutern und, um Ergebnisse zu beschreiben oder zu interpretieren.
¹⁹⁵

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

¹⁹⁶ ## [1] 9

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

203

204 **Aufgabe 1: Ausführen von Quellcodes**

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

```
# Einfache Rechenoperationen
```

```
1 + 3
```

```
2^7
```

```
# Einfache Funktion
```

```
sqrt(20)
```

208 Führen Sie nun alle Zeilen aus.

209 **1.4 RStudio Projekte**

210 Projekte in RStudio bieten eine einfache Möglichkeit Workflows zu vereinfachen. Dabei wird eine lokale
211 Umgebung erstellt und alle Pfadnamen beziehen sich auf das Verzeichnis des Projekts und sie müsse keine
212 absoluten Pfade angeben. Das hat zwei Vorteile:

213 Sie können Ihre R-Session direkt in dem Projekt starten. R-Projekte können zwischen unterschiedlichen
214 Rechnern geöffnet werden, ohne dass der Pfad angepasst werden muss.

215 **1.4.1 Erstellen eines Projektes**

216 Zum Erstellen eines Projektes müssen einige Schritte durchlaufen werden, diese sind in Abbildung 3 zusam-
217 mengefasst.

- 218 1. Gehen Sie zu `File > New Project ...`
- 219 2. Wählen Sie `New Project`.
- 220 3. Geben Sie einen Namen für das Projekt ein (z.B. den Namen einer Lehrveranstaltung) und ein neuer
221 Ordner mit dem Projektnamen wird erstellt.
- 222 4. Drücken Sie auf `Create Project`.

223 Sobald ein Projekt einmal erstellt wurde, können Sie einfach wieder auf das Projekt-Icon klicken und das
224 Projekt wieder öffnen (Abbildung 4)

225 Alternativ kann über `File > Open Project` in RStudio oder durch Auswählen des Projektnamens (siehe folgende
226 Abbildung) geöffnet werden (Abbildung 5).

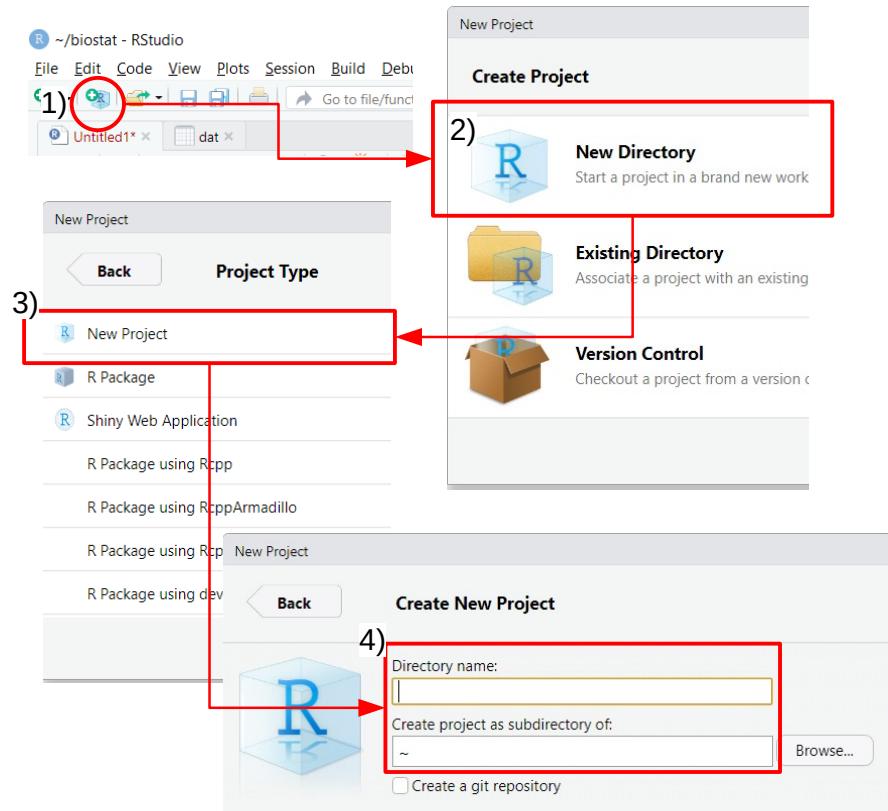


Abbildung 3: Workflow zum Erstellen eines Projekts.

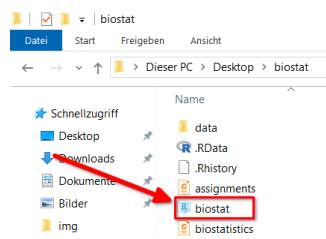


Abbildung 4: Öffnen eines RStudio Projekts.



Abbildung 5: Öffnen von Projekten.

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
```

```

250 ## [1] 15
b / a

251 ## [1] 0.5
a^b

252 ## [1] 1e+05

253 Das Ergebnis kann natürlich wieder in einer neuen Variable gespeichert werden.
ergebnis <- a + b
ergebnis

254 ## [1] 15
ergebnis2 <- ergebnis * 2
ergebnis2

255 ## [1] 30

256 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
257 Variablen wiederherzustellen. Sie müssen ggf. neu berechnet werden.
var1 <- "irgendwas"
exists("var1") # TRUE. also ja, eine Variable mit diesem Namen existiert

259 ## [1] TRUE
rm(var1)
exists("var1") # FALSE, also nein, eine Variable mit diesem Namen existiert nicht.

260 ## [1] FALSE

261 2.2 Funktionen

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

264 ## [1] 3.162278

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

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

```

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

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

274 ## [1] 3.162278

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

- 278 1. In die Konsole `?<Name der Funktion>` tippen. Also, wenn wir Hilfe für die Funktion `mean()` möchten,
279 könnten wir einfach `?mean` in die Konsole tippen.
- 280 2. Analog zu 1), können wir mit der Funktion `help` die Hilfeseite für eine andere Funktion aufrufen (z.B.
281 wenn wir wieder die Hilfe für die Funktion `mean()` lesen möchten, dann könnten wir auch `help(mean)`
282 in die Konsole tippen).
- 283 3. In RStudio kann man auch auf das Help-Panel klicken und dann einfach eine Funktion suchen (siehe
284 Abbildung 1).
- 285 4. Wenn der Cursor in RStudio auf einer Funktion ist, kann man mit der Taste F1 die dazugehörige
286 Hilfeseite aufrufen.

287 2.3 Datentypen

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

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

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

295 In den zwei vorherigen Zeilen Code haben wir zwei Objekte (im Sinne von R) erstellt. Das erste Objekt heißt
296 `kamera_name` und das zweite Objekt heißt `anzahl_rehe`. In dem Beispiel handelt es sich also um zwei sehr
297 einfache Objekte, in denen jeweils ein Wert gespeichert ist. Auffällig ist, dass beide Objekte unterschiedliche
298 Datentypen haben. `kamera_name` ist vom Typ `character` (also Text). Das zweite Objekt, `anzahl_rehe`, ist
299 vom Typ `numeric` (also eine Zahl, wir unterscheiden hier nicht weiter²). Zusätzlich zu diesen zwei Typen
300 (`character` und `numeric`), gibt es noch einen weiteren wichtigen Typ: nämlich das logische Wahr oder Falsch
301 (in R: `TRUE` und `FALSE`) und noch weitere Datentypen auf die wir zunächst nicht eingehen (tippen Sie `?typeof`
302 für eine Übersicht aller Datentypen). Zurückkommend auf das Beispiel mit den Fotofallen, könnte eine
303 mögliche Fragestellung sein, ob auf einem der Fotos ein Fuchs gesehen wurde oder nicht. Dazu würden wir
304 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
```

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

```
typeof(fuchs_gesehen)
```

306 ## [1] "logical"

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

```
TRUE + TRUE
```

308 ## [1] 2

```
FALSE + FALSE
```

309 ## [1] 0

```
TRUE + FALSE
```

310 ## [1] 1

311 2.4 Datenstrukturen

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

313 D. h., es wurde nicht nur eine Fotofalle ausgebracht, sondern insgesamt 15 Stück. Dieser Umstand erfordert

314 komplexere Objekte. Nachfolgend sind die Anzahl Rehfotos für jede der 15 Fotofallen aufgeführt: 132, 79,

315 129, 91, 138, 144, 55, 103, 139, 105, 96, 146, 95, 118, 107.

316 Die Frage, die sich jetzt stellt, ist: Wie kann man diese Daten sinnvoll organisieren? Zusätzlich zur Anzahl der

317 fotografierten Rehe soll jede Fotofalle eine eindeutige ID haben (`Kamera_1`, ..., `Kamera_15`) und wir wissen,

318 dass jeweils 5 Fotofallen in drei unterschiedlichen Revieren aufgestellt waren (Fotofalle 1 bis 5 in Revier A,

319 Fotofalle 6 bis 10 in Revier B und Fotofalle 11 bis 15 in Revier C). Wir könnten für jede Kamera und jeden

320 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.
```

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

322 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.

323 Frames) für diesen Zweck kennenlernen.

324

325 **Aufgabe 2: Variablen**

326 327 Verwenden Sie die folgenden Daten

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

328 und berechnen sie:

- 329 • 10 * a
330 • a / 144 und speichern Sie das Ergebnis in einer neuen Variablen e zwischen.
331 • Was ist das Ergebnis von a + b?
332 • Was ist das Ergebnis von a + p?

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

3 Vektoren

334 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 337 kein neues Objekt für Sie, sondern Sie lernen jetzt, dass die Ihnen schon bekannten Objekte Vektoren heißen 338 und sie auch mehrere Elemente in eine mObjekt speichern können.

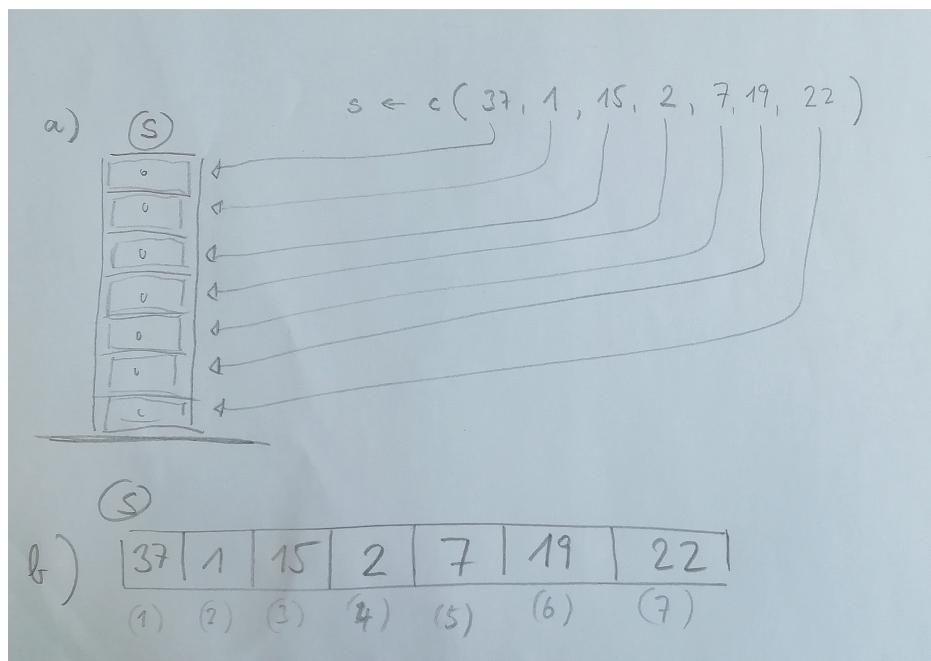


Abbildung 6: Schematische Darstellung eines Vektors in R.

339 Sie können sich Vektoren wie einen Schubladenschrank vorstellen (siehe auch Abbildung 6). Wichtig ist dabei, 340 dass man in jede Schublade immer nur ein Element verstauen kann und alle Elemente im Schublagenschrank 341 den gleichen Datentyp haben müssen. Etwas allgemeiner gesprochen heißt das, dass alle Elemente eines 342 Vektors vom gleichen Datentyp sein müssen.

343 Es gibt zahlreiche Funktionen zum Erstellen von Vektoren (einige davon werden wir im weiteren Verlauf des 344 Moduls kennenlernen). Die wohl wichtigste Funktion ist `c()`. Dabei steht `c` für *combine* oder *concatenate*. 345 Die Funktion `c()` fügt einzelne Elemente in einen Vektor zusammen (und zwar genau in der Reihenfolge wie 346 diese Elemente an `c()` übergeben werden). `c()` ist sozusagen die Funktion, die für Sie mehrere Elemente zu 347 einem Vektor zusammensetzt. `c()` erwartet folglich nur Elemente als Argumente.

348 Gehen wir nochmals zurück zu Abbildung 6, in der schematisch dargestellt wird, wie ein Vektor `s` mit 7 349 Elementen (in diesem Fall Zahlen) erstellt wird.

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

350 Die Funktion `c()` ordnet jetzt bildlich gesprochen die Zahl 37 der ersten Schublade zu, die Zahl 1 der zweiten 351 Schublade und so weiter. Wenn Sie jetzt einfach `s` in die Konsole tippen, können Sie alle Elemente von `s` 352 sehen:

s

353 `## [1] 37 1 15 2 7 19 22`

354 In Abbildung 6b wird der Vektor **s** nochmals systematisch dargestellt. Dabei sieht man z. B., dass 37 an der ersten Position des Vektors gespeichert wird und 22 an der letzten Position des Vektors gespeichert wird.

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

s + 10

359 `## [1] 47 11 25 12 17 29 32`

360 oder **s** mit sich selbst multiplizieren. Zu beachten ist also, dass es sich bei Vektorberechnungen in R
361 zunächst nicht um Vektorrechnungen handelt, wie Sie sie aus der linearen Algebra kennen. Für die sog.
362 Matrizenoperationen der linearen Algebra werden die Operatoren in R mit % % umschlossen, also bspw. **s**
363 %*% **s**.

s * s

364 `## [1] 1369 1 225 4 49 361 484`

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

seq(from = 1, to = 10)

368 `## [1] 1 2 3 4 5 6 7 8 9 10`

(1 : 10)

369 `## [1] 1 2 3 4 5 6 7 8 9 10`

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

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

371 `## [1] 1 3 5 7 9`

372

373 Aufgabe 3: Vektoren erstellen

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

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

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

3.1 Funktionen zum Arbeiten mit Vektoren

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

```
head(s)
```

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

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

383 `## [1] 37 1 15`

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

384 `## [1] 19 22`

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

```
length(s)
```

386 `## [1] 7`

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

```
class(s)
```

388 `## [1] "numeric"`

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

```
unique(s)
```

390 `## [1] 37 1 15 2 7 19 22`

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

```
table(s)
```

392 `## s`

393 `## 1 2 7 15 19 22 37`

394 `## 1 1 1 1 1 1 1`

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

```
rev(s)
```

397 `## [1] 22 19 7 2 15 1 37`

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

```
sort(s)
```

399 `## [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.

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

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

401 ## [1] 37 1 15 2 7 19 22 37 1 15 2 7 19 22

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

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

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

404 ## [1] 1 2 3 4 1 2 3 4

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

405 ## [1] 1 1 2 2 3 3 4 4

406

407 Aufgabe 4: Arbeiten mit Vektoren

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

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

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

412 3.2 Statistische Funktionen

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

```
mean(s)
```

416 ## [1] 14.71429

```
median(s)
```

417 ## [1] 15

```
sd(s)
```

418 ## [1] 12.76341

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

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

422 ## [1] 1

```

sample(s, size = 3) # 2 Elemente

423 ## [1] 15 7 22
424 Wenn size weggelassen wird, dann bekommt man gleich viele Elemente zurück (wie der Vektor lang ist), d.h.
425 der Vektor wird nur permutiert.

```

426 3.3 Beispiel Fotofallen

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

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

```

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

```

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

```

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

```

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

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

```

v1 <- rep("Kamera", 15)

```

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

```

v2 <- 1:15

```

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

```

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

444 ## [1] "Kamera_1"   "Kamera_2"   "Kamera_3"   "Kamera_4"   "Kamera_5"   "Kamera_6"
445 ## [7] "Kamera_7"   "Kamera_8"   "Kamera_9"   "Kamera_10"  "Kamera_11"  "Kamera_12"
446 ## [13] "Kamera_13"  "Kamera_14"  "Kamera_15"

```

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

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

```
450 ## [1] "Revier A" "Revier B" "Revier C" "Revier A" "Revier B" "Revier C"  

  451 ## [7] "Revier A" "Revier B" "Revier C" "Revier A" "Revier B" "Revier C"  

  452 ## [13] "Revier A" "Revier B" "Revier C"
```

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

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

reviere
```

```
454 ## [1] "Revier A" "Revier A" "Revier A" "Revier A" "Revier A" "Revier B"  

  455 ## [7] "Revier B" "Revier B" "Revier B" "Revier B" "Revier C" "Revier C"  

  456 ## [13] "Revier C" "Revier C" "Revier C"
```

457

458 Aufgabe 5: Statistische Funktionen

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

461 2. Erstellen Sie die folgende Konsolenausgabe:

```
462 ## [1] "Die mittlere Anzahl von Rehfotos beträgt 171.8 Rehe pro Standort."
```

463 3.4 Arbeiten mit logischen Werten

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

- 466 • Gleichheit (`==`)
- 467 • Ungleichheit (`!=`)
- 468 • Größer (`>`) und kleiner (`<`)
- 469 • Größer gleich (`>=`) und kleiner gleich (`<=`)

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

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

```
anzahl_rehe > 100
```

```
473 ## [1] TRUE FALSE TRUE FALSE TRUE TRUE FALSE TRUE TRUE FALSE TRUE  

  474 ## [13] FALSE TRUE TRUE
```

475 Das Ergebnis ist ein Vektor vom Datentyp `logi` in der selben Länge wie `anzahl_rehe`.

476 Wir können z. B. abfragen, welche Fotofallenstandorte sich in Revier B befinden.

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

```
477 ## [1] FALSE FALSE FALSE FALSE FALSE TRUE TRUE TRUE TRUE TRUE FALSE FALSE
478 ## [13] FALSE FALSE FALSE
```

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

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

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

```
485 ## [1] FALSE FALSE FALSE FALSE FALSE TRUE FALSE TRUE TRUE TRUE FALSE FALSE
486 ## [13] FALSE FALSE FALSE
```

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

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

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

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

494

Aufgabe 6: Arbeiten mit logischen Werten

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

```
496
497 1. TRUE | FALSE
498 2. FALSE & TRUE
499 3. (FALSE & TRUE) | TRUE
500 4. (2 != 3) | FALSE
501 5. FALSE + 10
502 6. TRUE + 10
503 7. TRUE + 10 == FALSE + 10
504 8. sum(c(TRUE, TRUE, FALSE, FALSE))
```

505 3.5 Zugreifen auf Elemente eines Vektors (=Untermengen)

506 Sehr oft wollen wir auf bestimmte Werte in einer Datenstruktur zugreifen. Beispielsweise könnte es uns
507 interessieren, wie viele Rehe im Mittel auf allen Fotofallen aus Revier A gesehen wurden. Das Zugreifen auf

509 Elemente mit bestimmten Eigenschaften ist die häufigste Anwendung für logische Operationen in R. Aus
510 diesem Grund ist das Unterkapitel “Arbeiten mit logischen Werten” auch Teil des Kapitels “Vektoren”.

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

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

```
anzahl_rehe[2]
```

520 ## [1] 79

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

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

522 ## [1] 132 79 129 91 138

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

523 ## [1] 132 79 129 91 138

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

526

527 Aufgabe 7: Zugreifen auf Vektorelemente

529 Erstellen Sie einen neuen Vektor `bhd`

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

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

533

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

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

536 ## [1] 132 79 129 91 138

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

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

540 ## [1] 132 79 129 91 138

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

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

543 ## [1] 132 79 129 91 138

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

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

546 ## [1] 113.8

547

548 Aufgabe 8: logische Werte

550 Verwenden Sie nochmals den Vektor mit den Anzahl Rehen die an unterschiedlichen Fotofallenstandorten
551 fotografiert wurden.

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

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

555 3.6 Der %in%-Operator

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

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

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

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

561 ## [1] "FI" "FI"

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

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

564 ## [1] "FI" "BU" "BU" "FI"

565 Diese Herangehensweise wird aber für > 2 Elemente in `arten` sehr mühsam und fehleranfällig. Eine Alternative
 566 bietet der `%in%`-Operator. Dieser testet, ob Elemente eines Vektors in einem zweiten Vektors enthalten sind.

567 Der Operator ist also eine verkürzte Schreibweise für hintereinander durchgeführte logische Oder Abfragen.

```
messungen_arten %in% arten
```

568 ## [1] TRUE TRUE TRUE FALSE FALSE FALSE TRUE FALSE FALSE

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

569 ## [1] "FI" "BU" "BU" "FI"

570

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

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

```
LETTERS
```

574 ## [1] "A" "B" "C" "D" "E" "F" "G" "H" "I" "J" "K" "L" "M" "N" "O" "P" "Q" "R" "S"

575 ## [20] "T" "U" "V" "W" "X" "Y" "Z"

576 Wählen Sie aus LETTERS nur die Vokale aus.

577 4 Faktoren (factors)

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

584 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)
```

585 ## [1] FI BU FI EI EI FI FI
 586 ## Levels: BU EI FI

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

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

590 ## [1] FI BU FI EI EI FI FI
 591 ## Levels: FI BU EI

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

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

594 ## [1] Fichte Buche Fichte Eiche Eiche Fichte Fichte
 595 ## Levels: Fichte Buche Eiche

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

```
levels(af)

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

599 ## [1] Fi Bu Fi Ei Ei Fi Fi
 600 ## Levels: Fi Bu Ei

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

```
af
```

```
603 ## [1] Fi Bu Fi Ei Ei Fi Fi
604 ## Levels: Fi Bu Ei
  relevel(af, "Bu")
```

```
605 ## [1] Fi Bu Fi Ei Ei Fi Fi
606 ## Levels: Bu Fi Ei
```

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

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

```
609 ## [1] "Fi" "Bu" "Fi" "Ei" "Ei" "Fi" "Fi"
```

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

```
af
```

```
611 ## [1] Fi Bu Fi Ei Ei Fi Fi
612 ## Levels: Fi Bu Ei
```

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

```
613 ## [1] 1 2 1 3 3 1 1
```

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

616

Aufgabe 10: Faktoren

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

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

621 4.1 Das Paket **forcats**

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

- 624 1. Die Anordnung von Levels anzupassen.
- 625 2. Levels zusammenzufassen oder zu entfernen.
- 626 3. Labels zu ändern.

627 **4.1.1 Anpassen der Anordnung von Faktoren**628 Wir verwenden nochmals den **a** Vektor mit unterschiedlichen Baumarten:

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

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

```
factor(a)
```

630 ## [1] FI BU FI EI EI FI FI

631 ## Levels: BU EI FI

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

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

```
f1
```

633 ## [1] FI BU FI EI EI FI FI

634 ## Levels: FI BU EI

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

```
fct_rev(f1)
```

636 ## [1] FI BU FI EI EI FI FI

637 ## Levels: EI BU FI

638 nach der Häufigkeit zu sortieren oder

```
fct_infreq(f1)
```

639 ## [1] FI BU FI EI EI FI FI

640 ## Levels: FI EI BU

641 eine zufällige Sortierung.

```
fct_shuffle(f1)
```

642 ## [1] FI BU FI EI EI FI FI

643 ## Levels: EI FI BU

5 Spezielle Einträge

- 644 In vielen Fällen werden spezielle Einträge benötigt, bspw. bei
- 646 • fehlenden Einträgen NA,
 - 647 • leeren Einträgen NULL,
 - 648 • undefinierten Einträgen NaN (Not a Number) oder
 - 649 • unendlichen Zahlen (Inf).
- 650 Spezielle Einträge sind reservierte Namen. Sie können nicht überspeichert werden.

651 5.1 NA

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

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

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

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

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

```
is.na(na1)

## [1] FALSE TRUE FALSE
```

```
na.omit(na1)

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

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

```
na2 < 3

## [1] FALSE FALSE     NA
```

1 + NA

670 `## [1] NA`
671 Viele R Funktionen haben eingebaute Methoden zum Umgang mit NA. Die Funktion `mean()` bspw. ergibt
672 (wie die meisten Funktionen) standardmäßig NA wenn sie auf Vektoren mit Datenlücken angewendet wird, es
673 sei denn man stellt innerhalb der Funktion ein, dass Datenlücken entfernt werden sollen.

`mean(na2)`

674 `## [1] NA`
675 `mean(na2, na.rm = TRUE)`
676 `## [1] 4.5`

5.2 NULL

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

5.3 Inf

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

`10^309`

684 `## [1] Inf`
685 `2 * Inf`
686 `## [1] Inf`
687 `1 + Inf`
688 `## [1] Inf`
689 `3 / 0`
690 `## [1] -Inf`
691 `3 / Inf`
692 `## [1] 0`

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

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

692 ## [1] TRUE FALSE FALSE TRUE FALSE
is.finite(inf1)

693 ## [1] FALSE TRUE TRUE FALSE TRUE
inf1 < 3

694 ## [1] FALSE TRUE FALSE TRUE FALSE

695
```

696 **Aufgabe 11: Vektoren mit speziellen Einträgen**

698 Verwenden Sie den Vektor

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

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

701 Verwenden Sie den Vektor

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

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

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

707 Berechnen Sie den arithmetischen Mittelwert von `foo`.

708 6 data.frames oder Tabellen

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

716 Sie können sich ein `data.frame` wie eine Tabelle aus einem Tabellenkalkulationsprogramm vorstellen. Es gibt
 717 Zeilen in denen die Beobachtungen gespeichert sind und Spalten, die die Merkmale speichern. In unserem
 718 Fall gäbe es 15 Zeilen (eine Zeile für jede Fotofalle) und drei Spalten (jeweils eine Spalte für ID, Anzahl Rehe
 719 und Revier). Der Befehl zum Erstellen eines `data.frames` aus Vektoren in R ist `data.frame()`. Für unser
 720 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
```

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

742 Standard-Objekte zum Speichern wissenschaftlicher Daten.

743 6.1 Wichtige Funktionen zum Arbeiten mit `data.frames`

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

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

```
746 ##           ID anzahl_rehe    revier
747 ## 1 Kamera_1          132 Revier A
748 ## 2 Kamera_2          79 Revier A
```

749 Oder für die letzten 2 Beobachtungen.

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

```
750 ##           ID anzahl_rehe    revier
751 ## 14 Kamera_14         118 Revier C
752 ## 15 Kamera_15         107 Revier C
```

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

```
nrow(monitoring)
```

```
754 ## [1] 15
```

```
ncol(monitoring)
```

```
755 ## [1] 3
```

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

```
str(monitoring)
```

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

```
762
```

763 Aufgabe 12: `data.frame`

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

6.2 Zugreifen auf Elemente eines `data.frame`

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

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

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

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

Alternativ, kann man den Spaltennamen auch einfach Ausschreiben. Dies hat beim Programmieren den Vorteil, dass der Code lauffähig bleibt, falls sich die Reihenfolge der Spalten in der Datengrundlage ändert.

Nachteil ist entsprechend, dass der Code nicht mehr läuft, falls die Variablennamen sich ändern.

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

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

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

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

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

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

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

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

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

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

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

```
799 ## 4 Kamera_4          91 Revier A
800 ## 5 Kamera_5          138 Revier A
```

801

802 **Aufgabe 13: Abfragen von Werten**

804 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)
)
```

- 805 • Wählen Sie nur die ersten drei Zeilen aus und die erste und zweite Spalte aus.
 806 • Wählen Sie alle Zeilen und die erste und dritte Spalte aus.
 807 • Wählen Sie alle Spalten und die erste, dritte und vierte Zeile aus.

808

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

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

```
monitoring$anzahl_rehe
```

814 ## [1] 132 79 129 91 138 144 55 103 139 105 96 146 95 118 107

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

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

816 ## [1] 132 79 129 91 138 144 55 103 139 105 96 146 95 118 107

817 3. Alle Zeilen und die Spalte explizit angeben.

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

818 ## [1] 132 79 129 91 138 144 55 103 139 105 96 146 95 118 107

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

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

```
monitoring$anzahl_rehe > 100
```

```
826 ## [1] TRUE FALSE TRUE FALSE TRUE TRUE FALSE TRUE TRUE TRUE FALSE TRUE  
827 ## [13] FALSE TRUE TRUE
```

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

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

```
832 ##          ID anzahl_rehe    revier  
833 ## 1   Kamera_1           132 Revier A  
834 ## 3   Kamera_3           129 Revier A  
835 ## 5   Kamera_5           138 Revier A  
836 ## 6   Kamera_6           144 Revier B  
837 ## 8   Kamera_8           103 Revier B  
838 ## 9   Kamera_9           139 Revier B  
839 ## 10  Kamera_10          105 Revier B  
840 ## 12  Kamera_12          146 Revier C  
841 ## 14  Kamera_14          118 Revier C  
842 ## 15  Kamera_15          107 Revier C
```

843

844 Aufgabe 14: Abfragen von Werten 2

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

- ```
846
847 • Alle Spalten für Studierende die Forstwissenschaften studieren.
848 • Alle Spalten für Studierende die Chemie oder Physik studieren.
849 • Die Spalte fach und semester für Studierende die 22 oder älter sind.
```

## 850 7 Schreiben und lesen von Daten

### 851 7.1 Textdateien

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

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

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

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

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

```
873 ## ID anzahl_rehe revier
874 ## 1 Kamera_1 132 Revier A
875 ## 2 Kamera_2 79 Revier A
876 ## 3 Kamera_3 129 Revier A
877 ## 4 Kamera_4 91 Revier A
878 ## 5 Kamera_5 138 Revier A
879 ## 6 Kamera_6 144 Revier B
```

880 Es gibt viele Varianten der Funktion `read.table()`. Beispielsweise hat die Funktion `read.csv()` bereits  
 881 die Argumente `sep = ','` und `header = TRUE` gesetzt. Die Funktion `read.csv2()` hat die in Deutschland  
 882 üblichen Argument `sep = ';'` und `dec = ','` gesetzt. Sie müssen natürlich darauf achten, dass sie die csv  
 883 Dateien mit den gleichen Spezifikationen einlesen, mit denen sie gespeichert wurde. Siehe dazu auch die  
 884 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.

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

886

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

---

- 888  
889 Lesen Sie die Datei `kompliziert.txt` ein. Schauen Sie die Hilfeseite an und vergewissern Sie sich, dass Sie  
890 wissen was die Argumente `header`, `sep`, `dec` und `skip` bewirken. Setzten die Argumente richtig, damit die  
891 Datei `kompliziert.txt` folgendes Ergebnis liefert.

## 8 Erstellen von Abbildungen

892 Abbildungen sind ein elementarer Baustein statistischer Analysen und deshalb von Beginn an Teil von R. **R is a free software environment for statistical computing and graphics.** Es gibt unterschiedliche Systeme  
893 einen Plot zu erstellen. In diesem Kurs werden wir kurz *Base Plots* vorstellen und dann das Zusatzpaket  
894  
895  
896 `ggplot2` vorstellen.

### 897 8.1 Base Plot

898 Die wichtigsten Grafiken für die einfache Datendarstellung sind schnell verfügbar. Etwas komplexere oder  
899 spezielle Grafiken erfordern mehr Programmieraufwand (folgt teilweise noch). Für viele komplexere Diagramme  
900 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  
901 sie sich die einfache Grafik Schnittstelle (`base plots`) als zweidimensionale Leinwand vor, auf die Sie durch  
902  
903 Code Ebene für Ebene Grafikelemente legen:

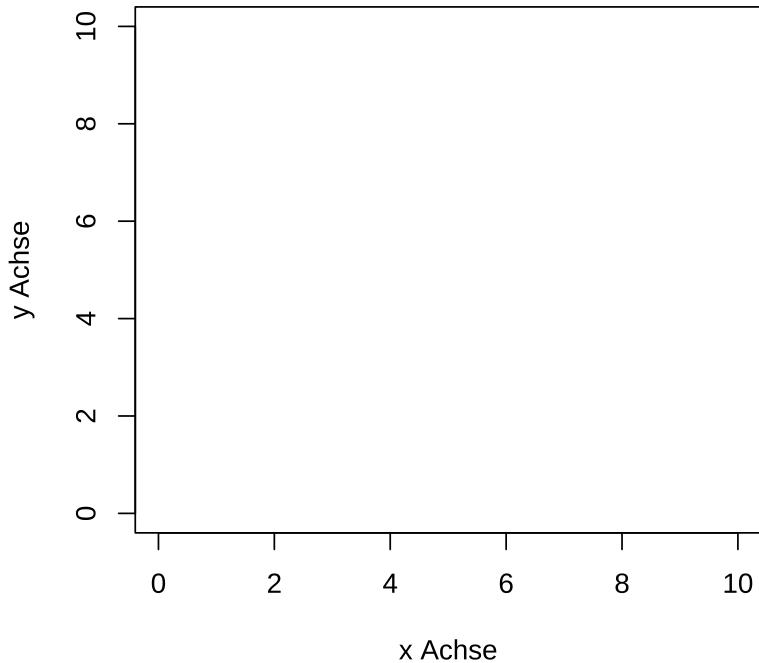
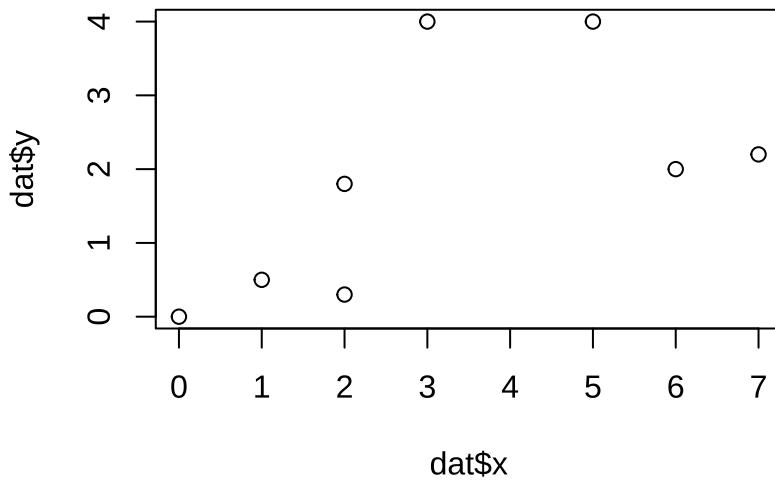


Abbildung 7: Beispiel einer leeren Grafikschnittstelle.

904 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)
)

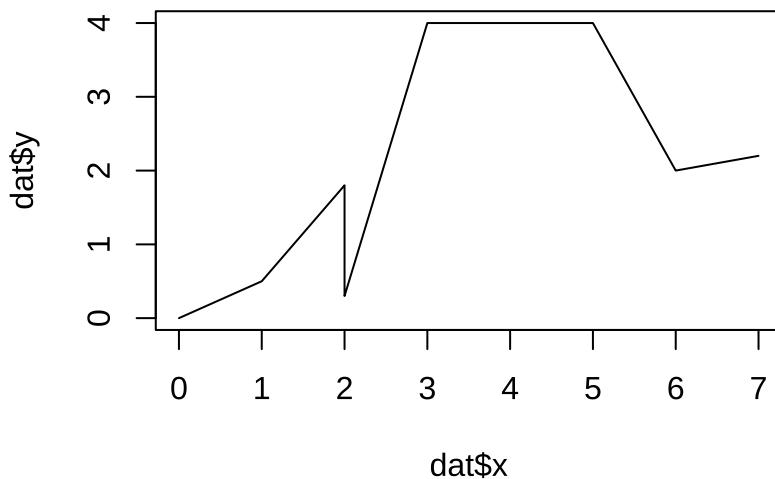
plot(datx, daty, type ="p")
```



905

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

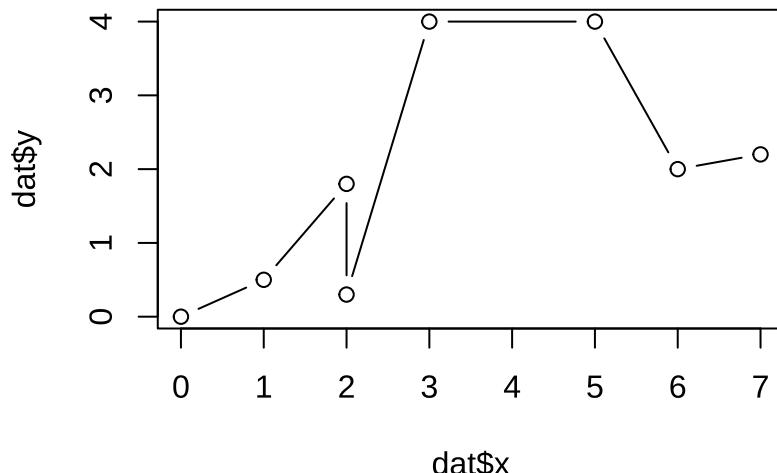
```
plot(datx, daty, type = "l")
```



908

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

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



910

911 darstellen.

912

913 **Aufgabe 16: Base Plot 1**

914

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

917

918 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  
919 Elemente hinzufügen. Mit jeder neuen Ebene können Sie die vorigen Ebenen jedoch nicht mehr verändern.  
920  
921 Die wichtigsten Argumente der `plot` Funktion sind:

- 922 • `type` - Diagrammtyp
- 923 • `col` - Farbe
- 924 • `main` - Titel
- 925 • `sub` - Untertitel
- 926 • `pch` - Punktsymbol
- 927 • `lty` - Linientyp
- 928 • `lwd` - Linienstärke
- 929 • `xlab` bzw. `ylab` - Achsenbeschriftungen
- 930 • `xlim`, `ylim` - Grenzen der Achsenanschnitte
- 931 • `axes` - Sollen die Achsen eingezeichnet werden? Oder leer gelassen werden, um sie nachträglich als  
932 low-level Ebene einzuziehen?
- 933 • `ann` - Achsenbeschriftung kann ganz weggelassen werden.

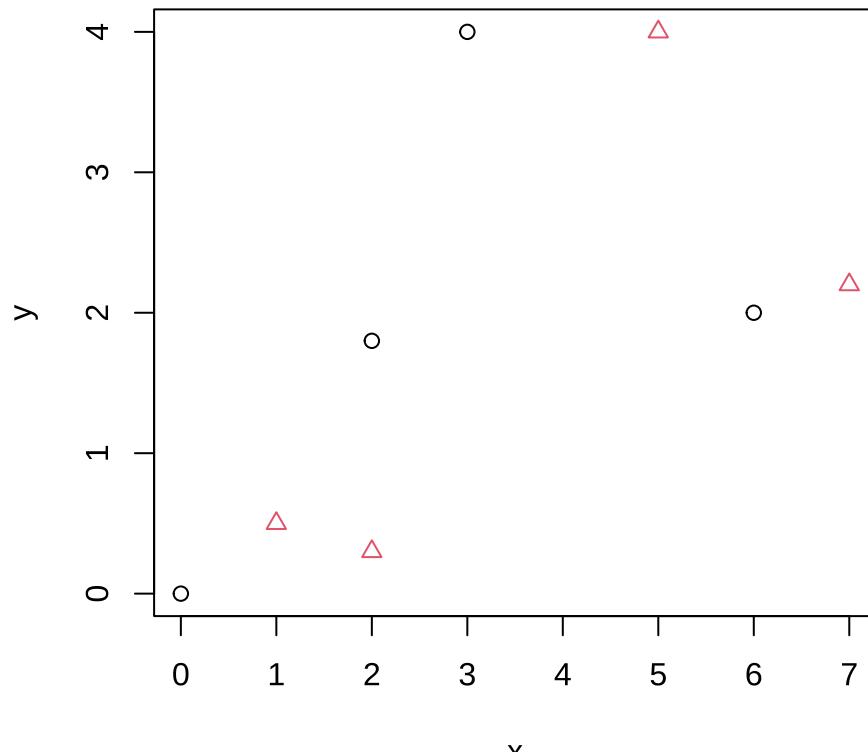
934 Sehen Sie sich die Hilfeseiten `?plot.default()` oder `?par()` an für weitere Informationen. Dort finden Sie  
935 auch eine vollständige Liste der Befehle. Einige Argumente können als Vektor übergeben werden. Hier z. B.  
936 die Farben und die Punktsymbole.

```

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")

```



937

938

---

**Aufgabe 17: Anpassen von Plots**


---

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

- 942     • Beschriften Sie die x- und y-Achse sinnvoll.  
 943     • Fügen Sie eine Überschrift hinzu.  
 944     • Wählen Sie ein anderes Symbol.  
 945     • Stellen Sie die Symbole in rot dar.

946

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

948 Die wichtigsten Funktionen sind

- 949     • `points()` - Fügt Punkte ein  
 950     • `lines()` - Fügt Linien ein

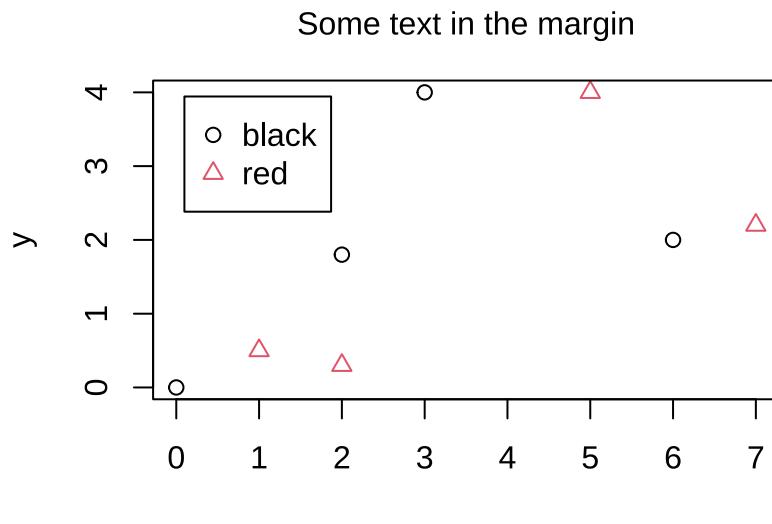
- 951 • `text()` - Fügt Text ein  
 952 • `mtext` - Fügt Text in den Rahmen (`margin`) ein  
 953 • `legend()` - Fügt eine Legende ein  
 954 • `abline()` - Fügt eine Gerade ein  
 955 • `curve()` - Fügt eine mathematische Funktion ein  
 956 • `arrows()` - Fügt Pfeile ein  
 957 • `grid()` - Fügt Hilfslinien ein

958 Dabei ist der Aufbau zunächst grundsätzlich wie in Abbildung 8 dargestellt. Der Vorteil von Low-Level  
 959 Funktionen ist, dass die einzelnen Level mehr Funktionen bieten als die High-Level Funktion und, dass Sie  
 960 sich die Reihenfolge der Ebenen definieren können.

961 Legenden können im base plot nur als Low-Level Funktion hinzugefügt werden. Mit der Funktion `legend`  
 962 werden die Einträge und die entsprechenden Punktsymbole oder Linientypen dargestellt. In der folgenden  
 963 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")
```



964  
 965 Mit diesem grundsätzlichen Aufbau sollten Sie bereits in der Lage sein auch komplexe Grafiken schnell zu  
 966 gestalten. Wenn Sie mehrere Diagramme in einem Plot arrangieren möchten, können Sie mit dem `par()`  
 967 Befehl ein Arrangement definieren. Sie haben dann zusätzlich zu den bereits bekannten Grafikregionen noch  
 968 äußere Ränder (`outer margins`). Siehe Abbildung 9.

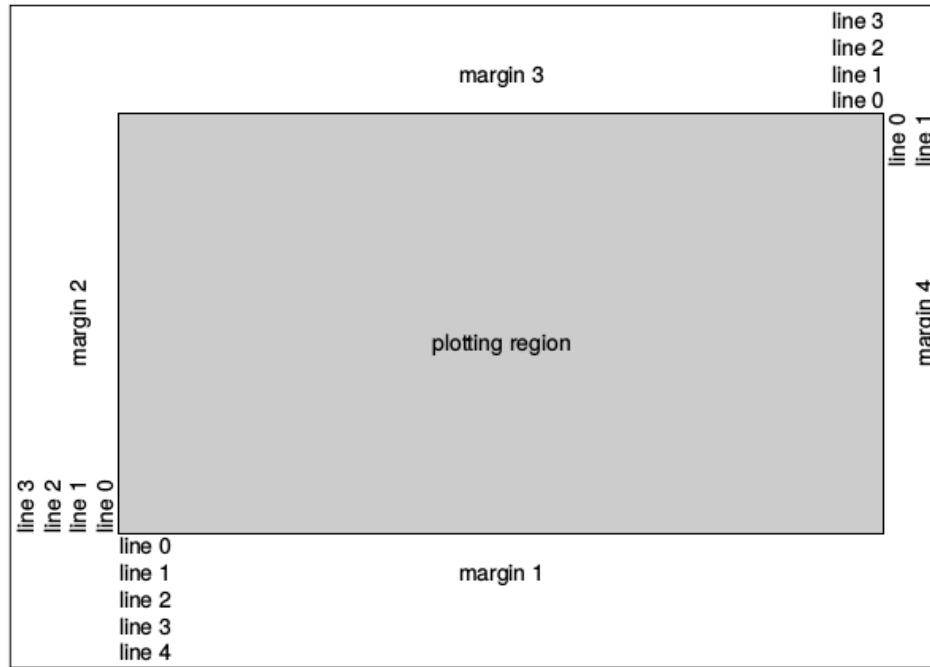
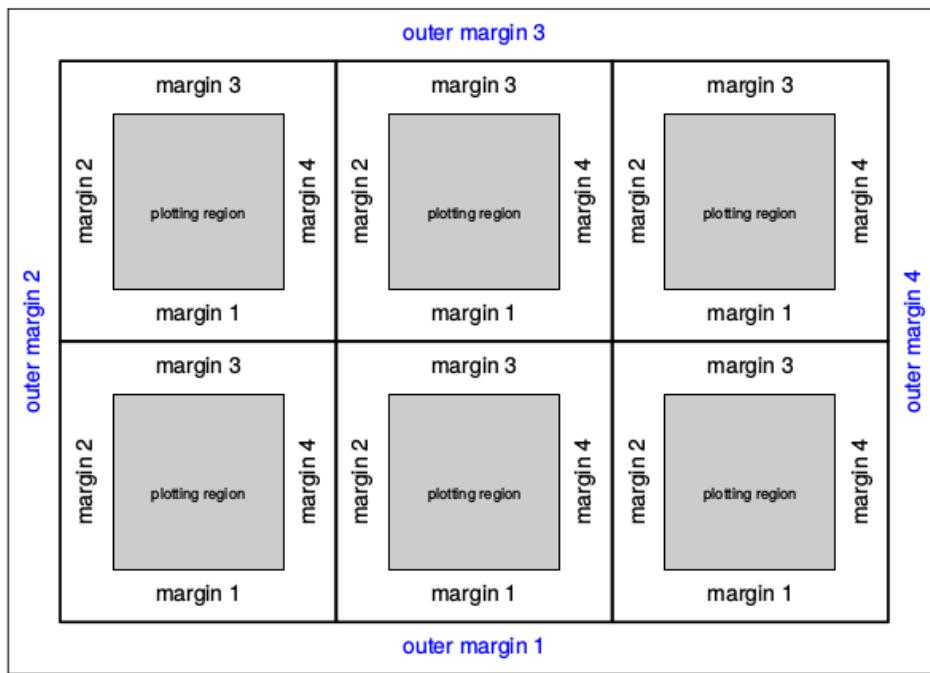


Abbildung 8: Grafikregionen eines base plots in R.

Abbildung 9: Schematischer Aufbau mehrerer Diagramme in einem plot am Beispiel einer  $3 \times 2$  Grafik.

969 **8.1.1 Mehrere Panels**

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

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

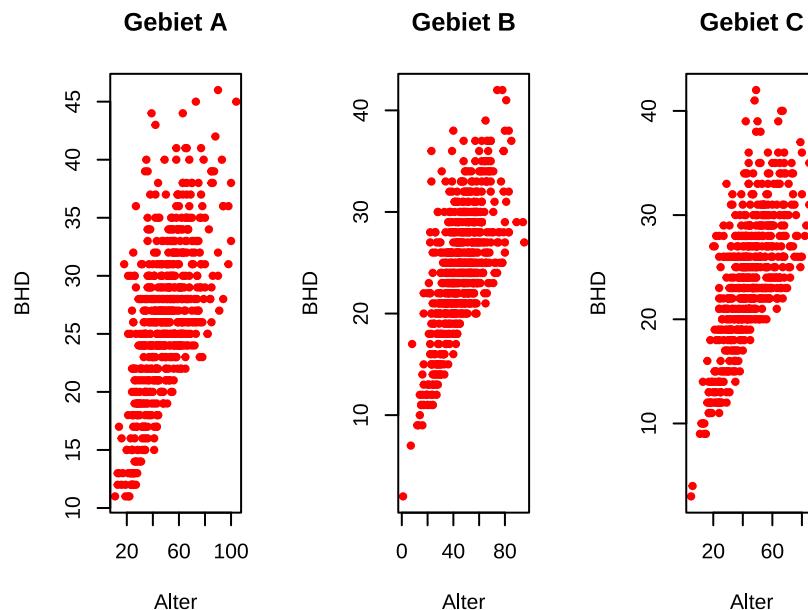
973 Teilt den Plot in eine Zeile und drei Zeilen (= drei Plots nebeneinander).

```
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")
```



974

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

977 **8.1.2 Speichern von Abbildungen**

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

- 982     • `pdf()` oder  
 983     • `postscript()`.

984 Beispiele für Rastergrafiken sind

- 985     • `png()`,  
 986     • `bmp()` oder  
 987     • `jpeg()`.

988 Die Grafikschnittstelle ist dann Ihre “Leinwand”. Mit dem Befehl `dev.off()` trennen Sie die Verbindung zur  
 989 Schnittstelle wieder. Ihre “Leinwand” wird also wieder geschlossen. So lange die Schnittstelle geöffnet ist  
 990 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
```

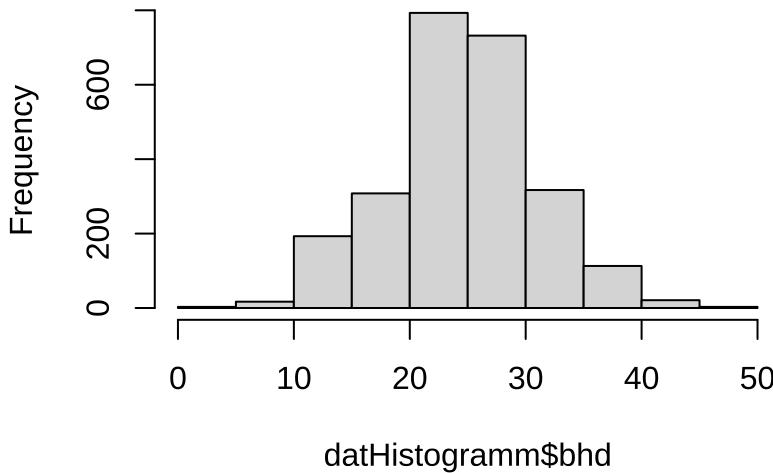
991 *Achtung*, wenn Sie die Funktion `dev.off()` nicht aufrufen, werden alle nachfolgenden Plots in die gleiche  
 992 Datei geschrieben. Falls Sie nach einem Versuch einen Plot zu speichern plötzlich keine weiteren Plots mehr  
 993 sehen, führen Sie einige Mal die Funktion `dev.off()` aus.

## 994 8.2 Histogramme

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

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

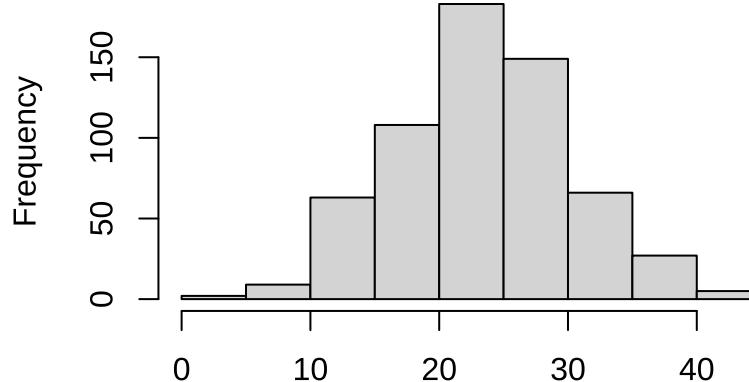
### Histogramm of datHistogramm\$bhd



1002

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

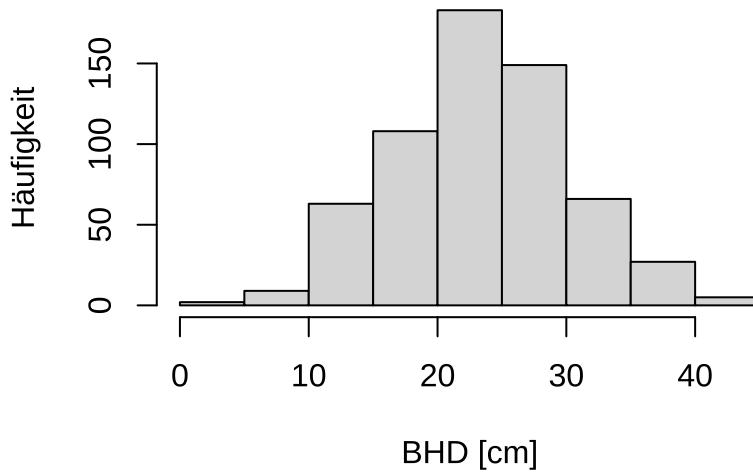
### Histogramm of datHistogramm\$bhd[datHistogramm\$art == "EI"]



1003

```
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

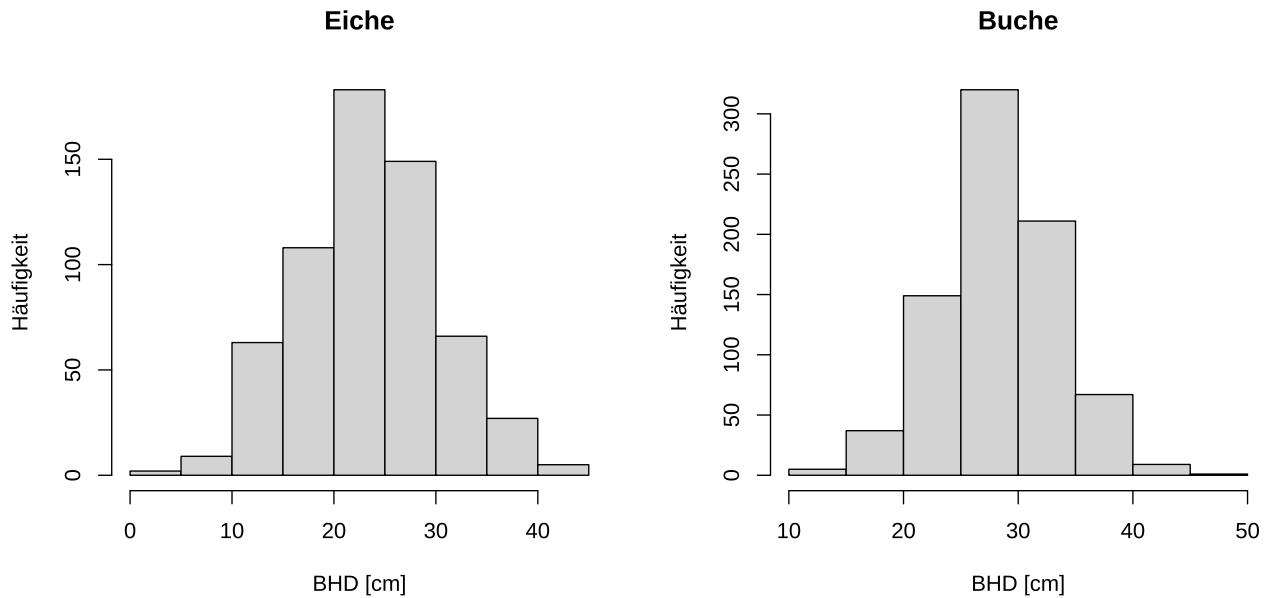


1004

1005 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")
```

1006



1007

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

### 1008 8.3 Boxplots

1009 Oft möchte man die Verteilung einer stetigen Variablen in Abhängigkeit einer diskreten Variable beschreiben  
 1010 oder visualisieren. Ein Beispiel dafür wäre die BHD-Verteilung in Abhängigkeit der Baumarten. Eine häufige  
 1011 Darstellungsform für solche Daten sind *Boxplots*. Sie stellen die durchschnittliche Ausprägung einer stetigen  
 1012 Variable und ihre Schwankung kompakt dar.

1013 Boxplots bestehen aus drei Komponenten:

- 1014 1. Eine *Box*, die den Bereich zwischen 0.25 und 0.75 Percentil abdeckt, diese Distanz wird auch die *IQR*  
 1015 (Interquartile Range), bezeichnet. Zusätzlich wird die Box durch den Median (als dicke horizontale Linie)  
 1016 unterteilt.
- 1017 2. Einzelne Punkte Ausreißer. Als Ausreißer werden Punkte bezeichnet, die  $> 1.5IQR$  vom unteren oder  
 1018 oberen Ende der Box entfernt sind.
- 1019 3. Eine senkrechte Linie von jeder Seite der Box bis zum letzten "Nicht-Ausreißer-Punkt". Also der letzte  
 1020 Punkt, der  $> 1.5IQR$  aber nicht  $> 0.75$  bzw.  $< 0.25$  Percentil ist. Diese Linie wird auch als *Whisker*  
 1021 bezeichnet.

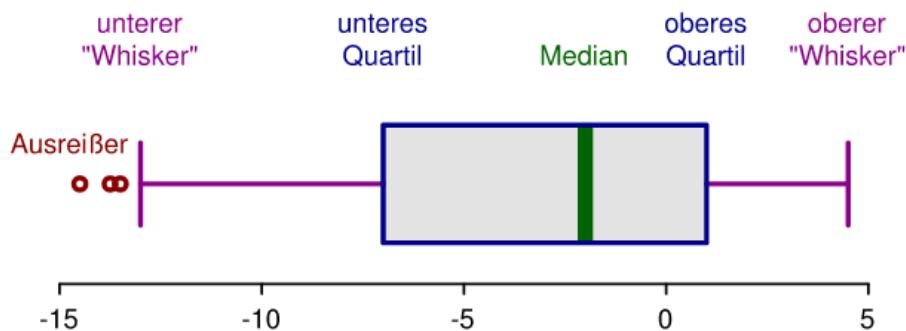
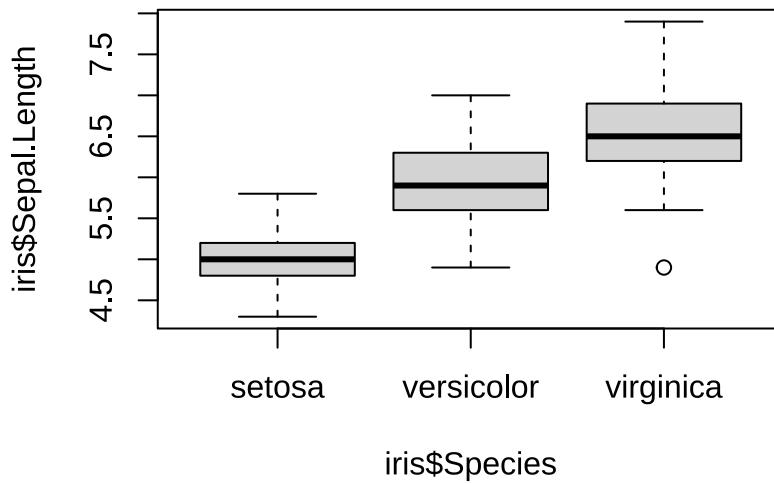


Abbildung 10: Schematische Darstellung eines Boxplots (Quelle: Von RobSeb - Eigenes Werk, CC BY-SA 3.0, <https://commons.wikimedia.org/w/index.php?curid=14697172>).

1022 Mit R kann mit der Funktion `boxplot()` ein Boxplot erstellt werden. Diese Funktion kann in zwei unter-  
 1023 schiedlichen Ausprägungen verwendet werden.

- 1024 1. `boxplot(x)` erzeugt einen Boxplot für die Variable `x`.
- 1025 2. `boxplot(x ~ y)` erzeugt einen oder mehrere Boxplots für `x` aber gruppiert nach `y`, dabei sollte `y` eine  
 1026 kategoriale Variable sein. `x` und `y` können auch die Spaltennamen eines `data.frames` sein, dann muss  
 1027 das `data.frame` mit dem Argument `data` zusätzlich übergeben werden.

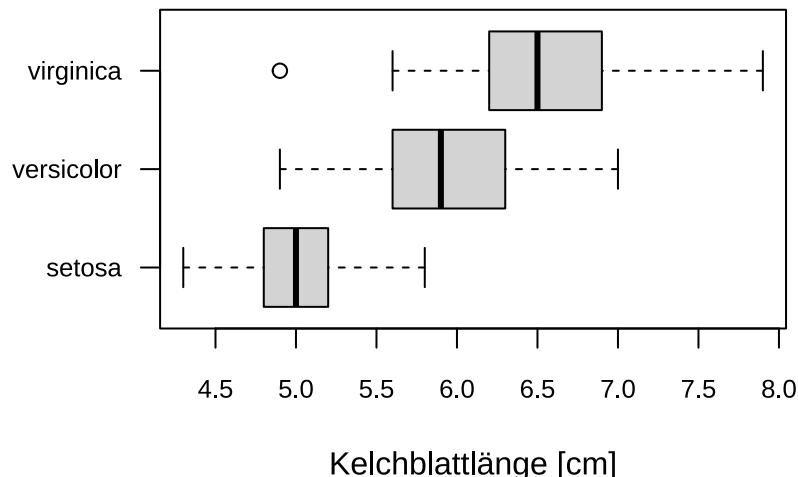
```
boxplot(iris$Sepal.Length ~ iris$Species)
```



1028

1029 Etwas eleganter ist es wenn wir das Argument `data` verwenden und den Plot etwas anpassen. Diese Schreib-  
1030 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
)
```



1031

1032

### 1033 Aufgabe 18: Boxplots

1034

- 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
- 1038 Erstellen Sie Boxplots für jedes Gebiet und innerhalb der Gebiete für jede Art.

## 1039 8.4 ggplot2: Eine Alternative für Abbildungen

1040 ggplot2 ist ein alternatives Plotting-System in R. Sie können mit ggplot2 also grundsätzlich Abbildungen  
 1041 mit dem selben Inhalt erstellen, wie mit Base Plots. Die Syntax und die optische Darstellung unterscheiden  
 1042 sich jedoch grundsätzlich. ggplot2 basiert auf den *grammar of graphics* von Leland Wilkinson. Die Idee  
 1043 ist, alle nötigen Informationen der Abbildung miteinander zu verknüpfen. ggplot2 ist also diametral zu  
 1044 Base Plots. Während Sie in base plot alle Elemente selbst definieren, ist die Idee von ggplot2, dass Sie  
 1045 nur die Daten und den Diagrammtypen übergeben und die Funktion selbst eine schöne Abbildung erstellt.  
 1046 Selbstverständlich können Sie aber auch in ggplot2 viele Einstellungen vornehmen. Im base plot sehen  
 1047 Abbildungen zunächst sehr karg und etwas unfertig aus. Sie müssen viele Einstellungen vornehmen, um eine  
 1048 publizierfähige Grafik zu produzieren. In ggplot2 sollen auch die einfachste Abbildungen schon ästhetisch  
 1049 sein. Mit diesen gebündelten Informationen kann ggplot2 die Abbildung automatisch verschönern. So  
 1050 werden bspw. die Legenden automatisch erzeugt und auch die Formatierungen automatisch an die Datenlage  
 1051 angepasst. ggplot2 nimmt der\*dem Entwickler\*in also Arbeit ab. Dadurch sind die Abbildungen schon ohne  
 1052 viel Nacharbeit schick. Nachteil ist, dass der\*dem Entwickler\*in weniger Möglichkeiten zur Einstellung zur  
 1053 Verfügung stehen und nuterspezifische Sonderwünsche somit schwerer umsetzbar sind. Sehen Sie sich das  
 1054 Cheatsheet zu ggplot2 an. Es ist in RStudio unter [Help > Cheatsheets](#) zu finden.

1055 Bei ggplot2 sind Anweisungen zu den Daten und Anweisungen zur Darstellung voneinander getrennt. Die  
 1056 Daten werden in den Ästhetikbefehl übergeben und dort klassifiziert. Dann folgen die Darstellungsanweisungen.  
 1057 Ähnlich wie bei Base Plots, werden die Grafikelemente ebenenweise nacheinander programmiert, jedoch mit  
 1058 einem + verbunden. Und hier liegt der wesentliche Unterschied zu Base Plots. Durch die + werden die Ebenen  
 1059 zu einem Befehl verbunden und damit gleichzeitig erstellt.

1060 Die Erweiterung wird zunächst geladen<sup>7</sup>. Wir laden außerdem den Datensatz **iris**. Der Datensatz ist in R  
 1061 fest integriert. Siehe `?iris` für mehr Informationen.

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

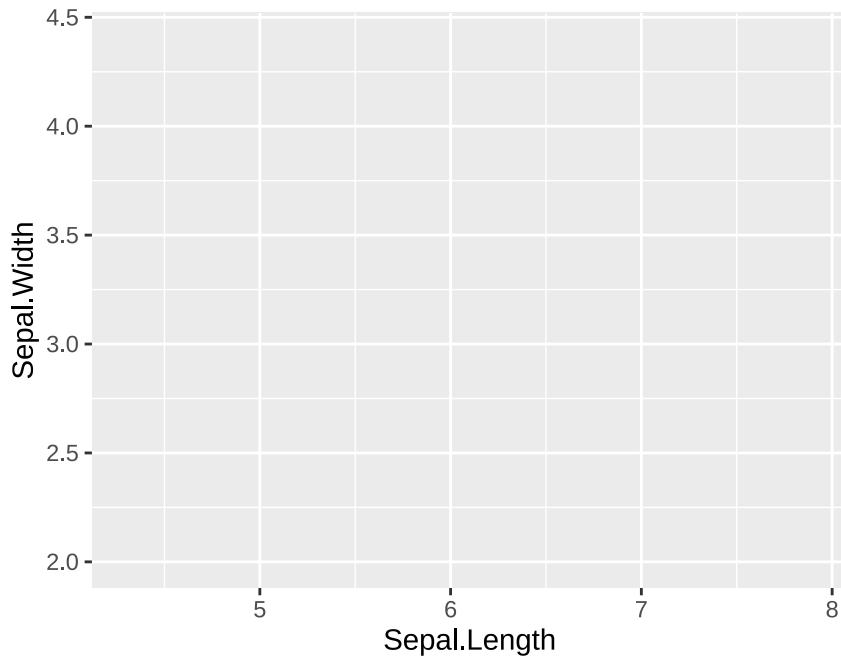
```
1062 ## Sepal.Length Sepal.Width Petal.Length Petal.Width Species
1063 ## 1 5.1 3.5 1.4 0.2 setosa
1064 ## 2 4.9 3.0 1.4 0.2 setosa
1065 ## 3 4.7 3.2 1.3 0.2 setosa
1066 ## 4 4.6 3.1 1.5 0.2 setosa
1067 ## 5 5.0 3.6 1.4 0.2 setosa
1068 ## 6 5.4 3.9 1.7 0.4 setosa
```

1069 Die Ästhetik wird bspw. folgendermaßen definiert.

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

---

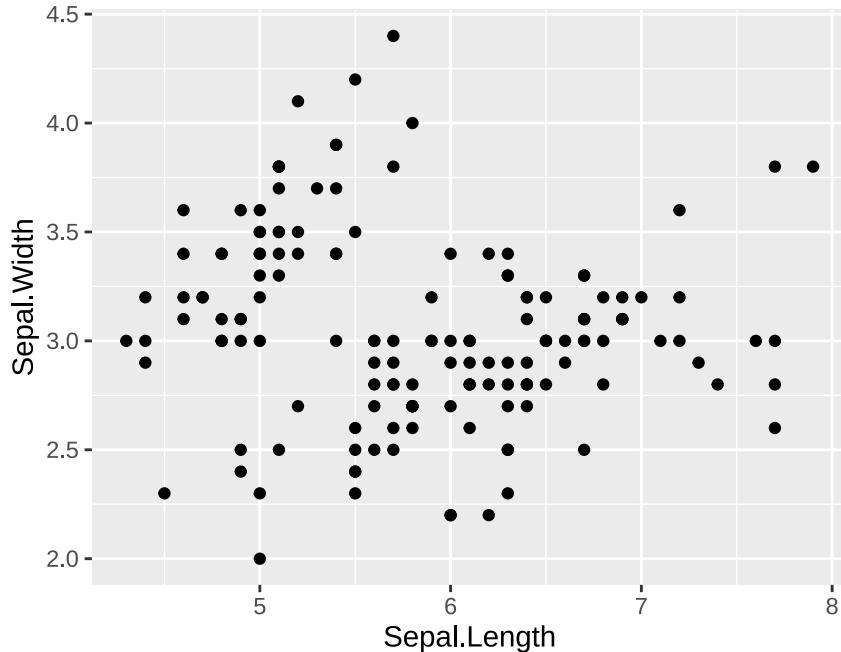
<sup>7</sup>Wir haben bis jetzt immer nur mit base R gearbeitet. D.h. wir haben nur Funktionen verwendet, die R bereits zur Verfügung stellt. Eine der großen Stärken von R sind die Erweiterungen (oder auch Pakete genannt). ggplot2 ist so eine Erweiterung, die einmal mit `install.packages("ggplot2")`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



1070

1071 Dieser Befehl zeichnet noch keine Daten. Die Daten werden lediglich herangezogen, um einen leeren Plot für  
1072 die Daten zu erstellen. In dem Beispiel wird die Variable `Sepal.Length` aus dem `data.frame iris` als x und  
1073 `Sepal.Width` als y Variable definiert. Diese Informationen stehen den folgenden Layern nun zur Verfügung,  
1074 sodass nach den + nur noch x und y verwendet werden müssen. Um bspw. einen Scatterplot zu erstellen  
1075 wird ein `geom_point()` Layer hinzugefügt. x und y werden automatisch an `geom_point` übergeben. Weitere  
1076 Einstellung sind in diesem Beispiel nicht notwendig, wären jedoch möglich. Siehe `?geom_point()`.

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



1077

1078

---

1079 **Aufgabe 19: Abbildungen mit ggplot2**

---

10801081 Verwenden Sie die Daten aus Aufgabe 16 und erstellen Sie einen Scatterplot mit ggplot2 wie in Aufgabe 16.

---

10821083 Wir haben mit der Funktion `geom_point()` demm Plot eine Punktgeometrie hinzugefügt. Es gibt noch viele  
1084 weitere Geometrien. Die wichtigsten sind:

- 
- 1085 •
- `geom_line()`
- für eine Linie.
- 
- 1086 •
- `geom_histogram()`
- um ein Histogramm zu erstellen.
- 
- 1087 •
- `geom_boxplot()`
- um einen Boxplot zu erstellen.
- 
- 1088 •
- `geom_bar()`
- um ein Säulendiagramm zu erstellen.

1089 Welche Geometrie die richtige ist, richtet nach dem Typ der darzustellenden Variablen. Beispielsweise  
1090 bietet sich `geom_point()` an, wenn man zwei kontinuierliche Variable darstellen möchte. Wenn man hin-  
1091 gegen die Verteilung von einer kontinuirlchen Variable darstellen möchte, dann bietet sich ein Histogramm  
1092 (`geom_histogram()`) oder auch eine geschätzte Dichte (z.B. `geom_density()`) an.

1093

---

1094 **Aufgabe 20: Abbildungen mit ggplot2**

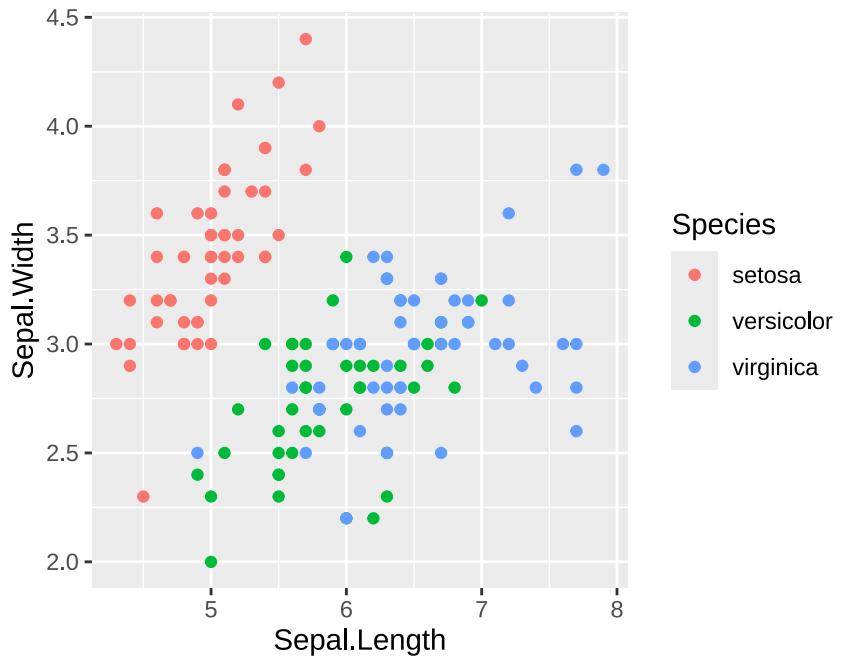
---

10951096 Verwenden Sie die den Iris Datensatz und erstellen Sie mit ggplot2 einen Plot der die Verteilung der Länge  
1097 der Kelchblätter zeigt (Spalte `Sepal.Length`).

---

10981099 Eine der Stärken von ggplot2 ist, dass man den Wert unterschiedlicher Variable auf unterschiedlichen  
1100 Komponenten des Plots abbilden kann. Wir haben bis jetzt ein bzw. zwei Variable auf der x- und y-Achse  
1101 abgebildet. Wir können aber ein weitere Variablen verwenden um das Aussehen des Plots zu beeinflussen.  
1102 Beispielsweise können wir die Farbe der Punkte (für `geom_point()`) mit dem Argument `col` beeinflussen.

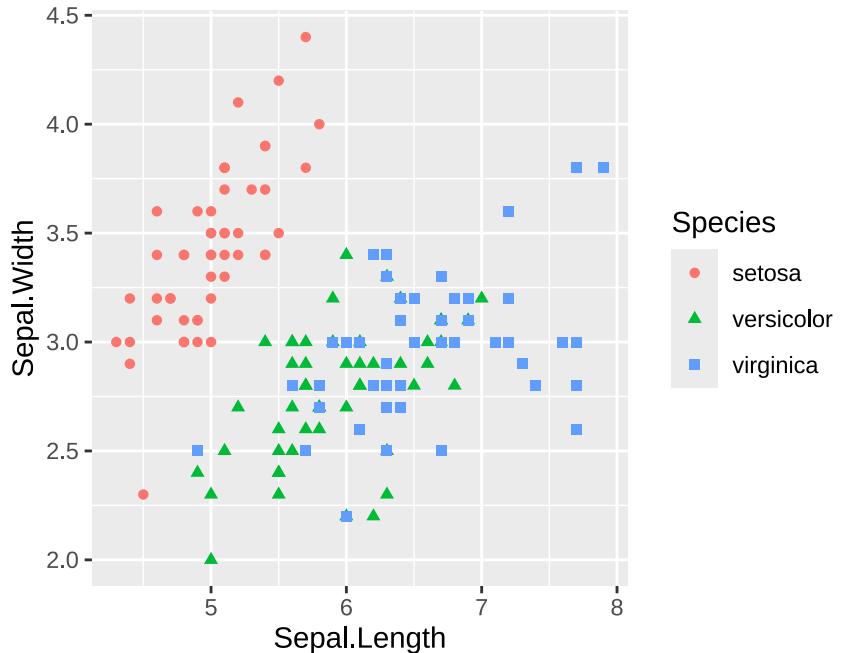
```
ggplot(iris, aes(x = Sepal.Length, y = Sepal.Width, col = Species)) +
 geom_point()
```



1103

1104 Somit bekommt jede Irisart eine eigene Farbe<sup>8</sup>. Gleichesmaßen können wir die Punktart (**shape**), die  
1105 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()
```

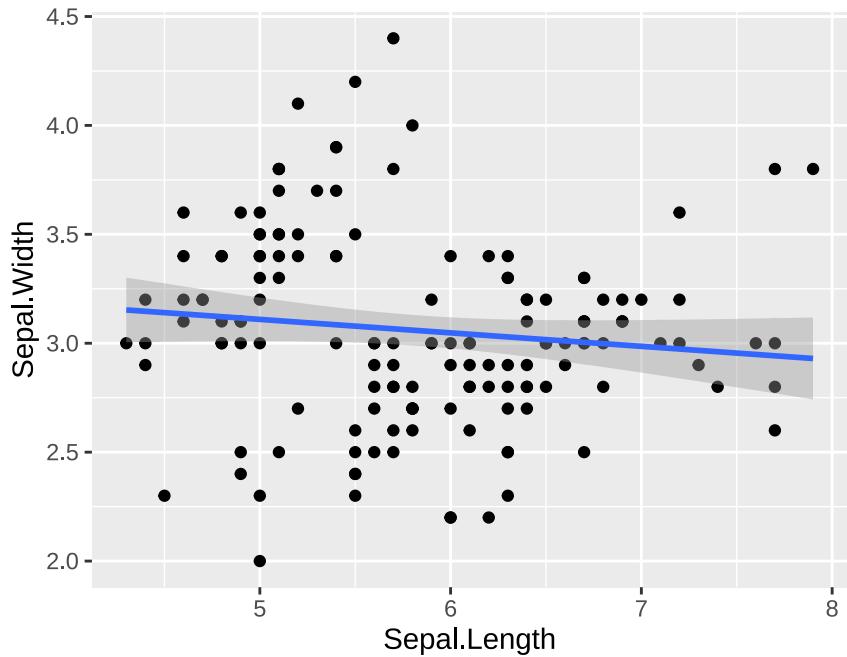


1106

1107 In dem Plot ist die Information zu der Art redundant (einmal als Farbe und einmal Symbolart).  
1108 Ein weitere sehr nützliche Geometrie ist **geom\_smooth()**, die es erlaubt eine Trendlinie hinzuzufügen.

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

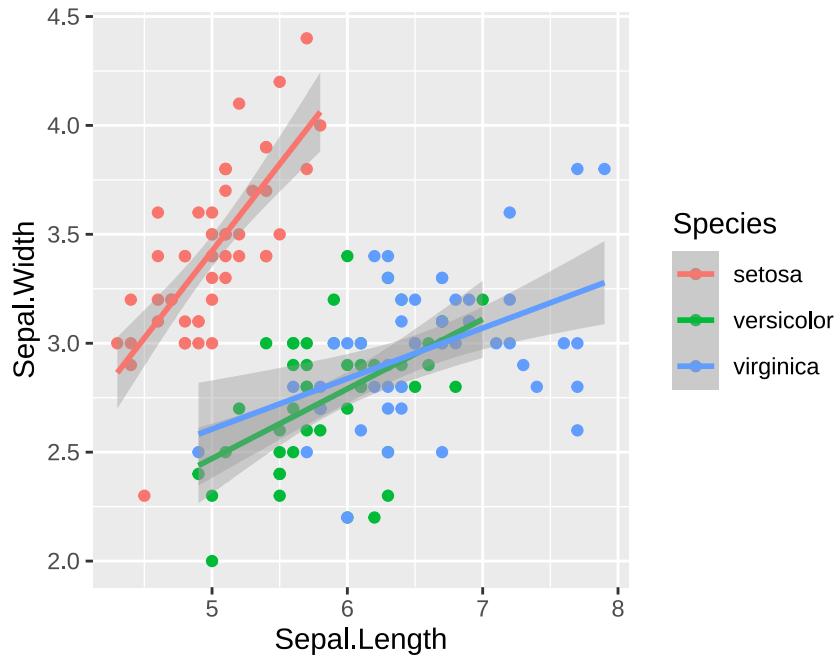
```
ggplot(iris, aes(x = Sepal.Length, y = Sepal.Width)) +
 geom_point() + geom_smooth(method = "lm")
```



1109

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

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



1113

1114

1115 **Aufgabe 21: Anpassen von Plots**  
1116

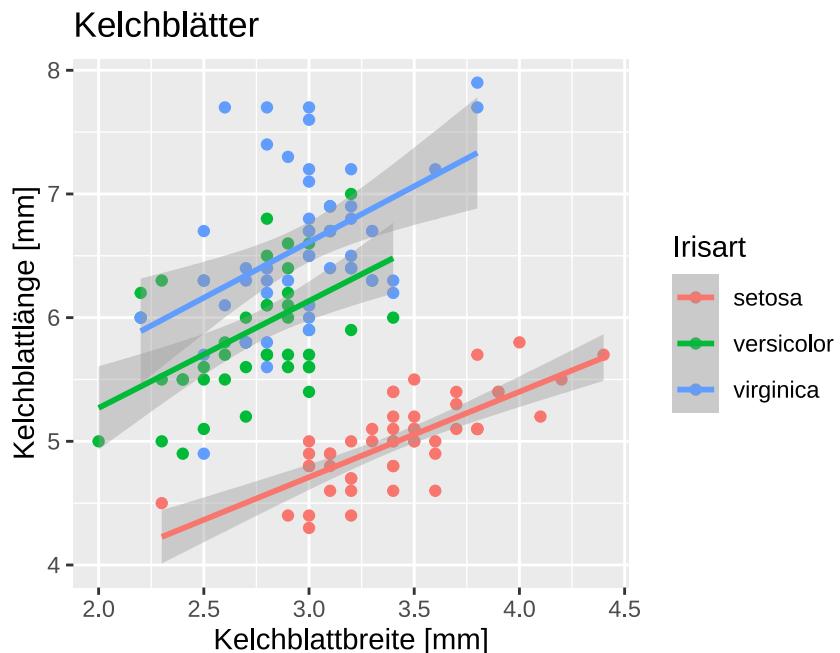
- 1117 Lesen Sie den Datensatz `data/bhd_1.txt` ein und erstellen Sie einen Boxplot für die Verteilung des BHDs  
 1118 für jede Baumart. In einem zweiten Schritt verwenden Sie erst `col = gebiet` und dann `fill = gebiet`.  
 1119 Welchen Unterschied stellen Sie fest?

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

1120

- 1121 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")
```



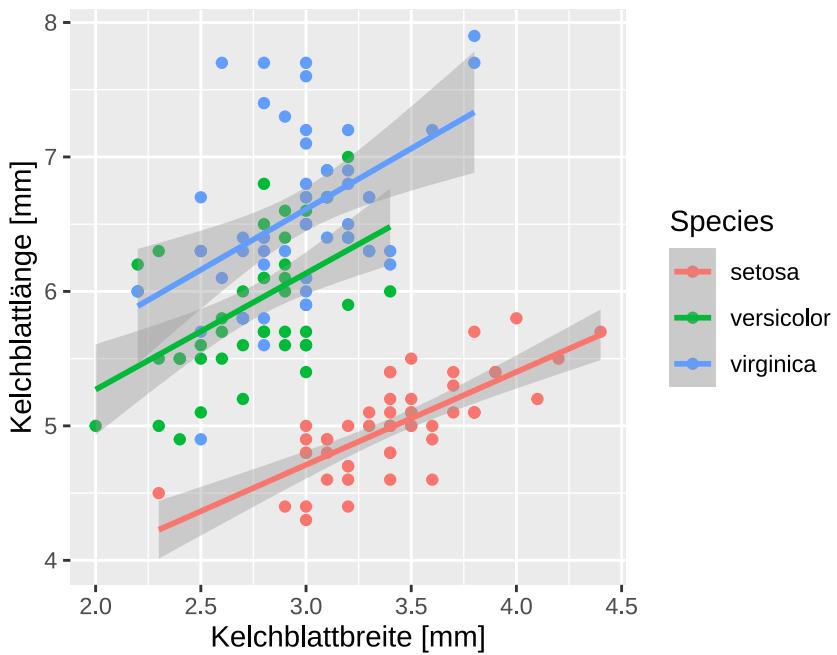
1122

- 1123 Statt einen langen Befehl zu tippen, kann ein `ggplot()` auch zwischengespeichert und wieder aufgerufen bzw.  
 1124 angepasst werden. Das ist vor allem sinnvoll, wenn mehrere Abbildungen auf dem selben Zwischenergebnis  
 1125 aufbauen sollen.

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

- 1126 Wir können jetzt mit `p1` weiter arbeiten und beispielsweise eine Beschriftung hinzufügen.

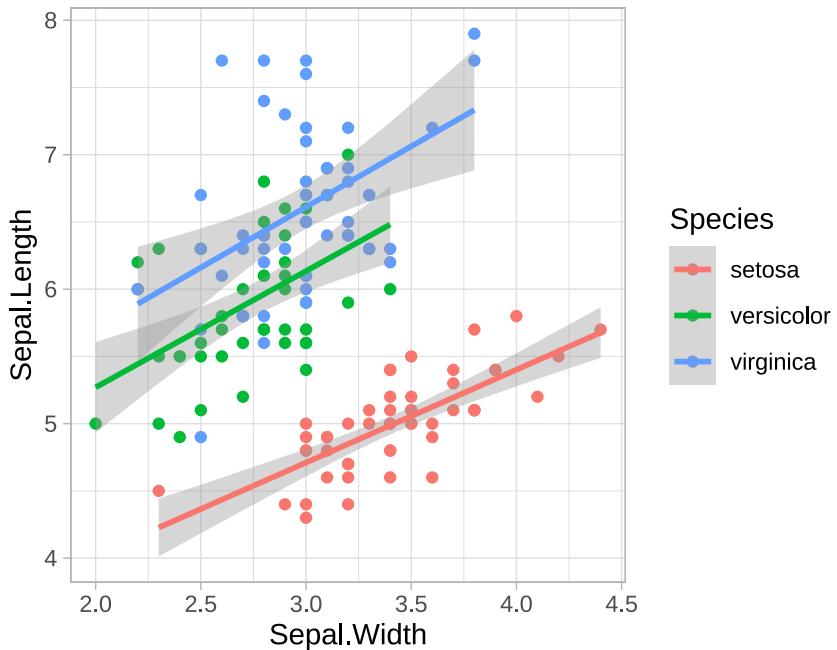
```
p1 + labs(x = "Kelchblattbreite [mm]", y = "Kelchblattlänge [mm]")
```



1127

1128 Oder auch den ganzen Plot anpassen. Dafür gibt es *themes*. Es gibt eine Reihe von vorgefertigten *themes*  
1129 oder man kann diese auch selber erstellen (das ist aber nicht Teil dieses Kurses).

```
p1 + theme_light()
```

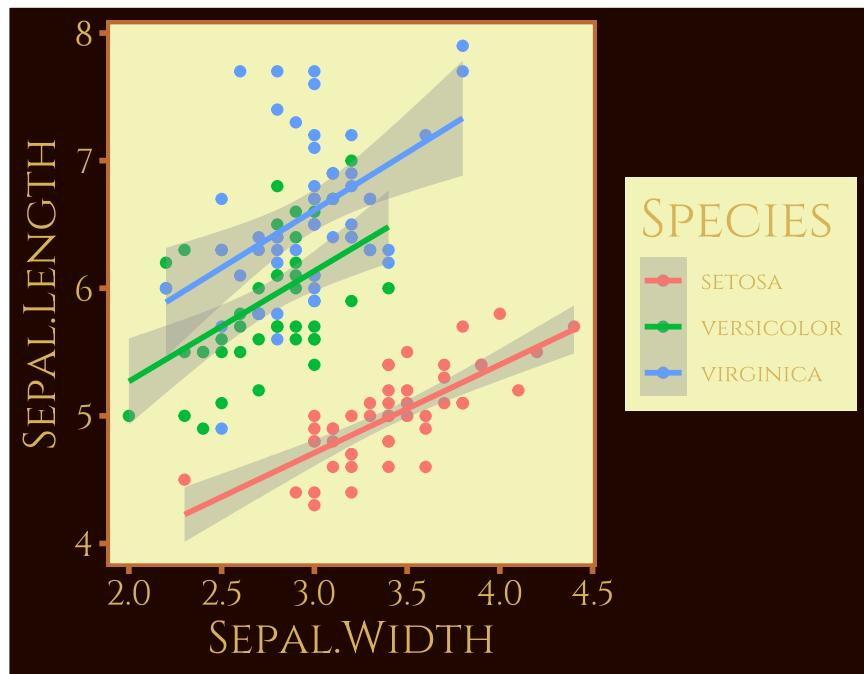


1130

1131 Weitere *themes* sind: `theme_bw()`, `theme_linedraw()` oder `theme_dark()`. Es gibt extra Pakete die viele  
1132 zusätzliche weitere *themes* anbieten. Dazu gehört z.B. das Paket `ggthemes` oder `ThemePark`. Während  
1133 `ggthemes` hauptsächlich durchdachte Grafikkonzepte liefert, sind die Themes aus `ThemePark` eher Popkultur  
1134 und nicht 100 %ig ernst gemeint. `ThemePark` muss zunächst aus GitHub installiert werden. Die Installation

1135 wird auf der GitHub erläutert.

```
p1 + themePark::theme_gameofthrones()
```

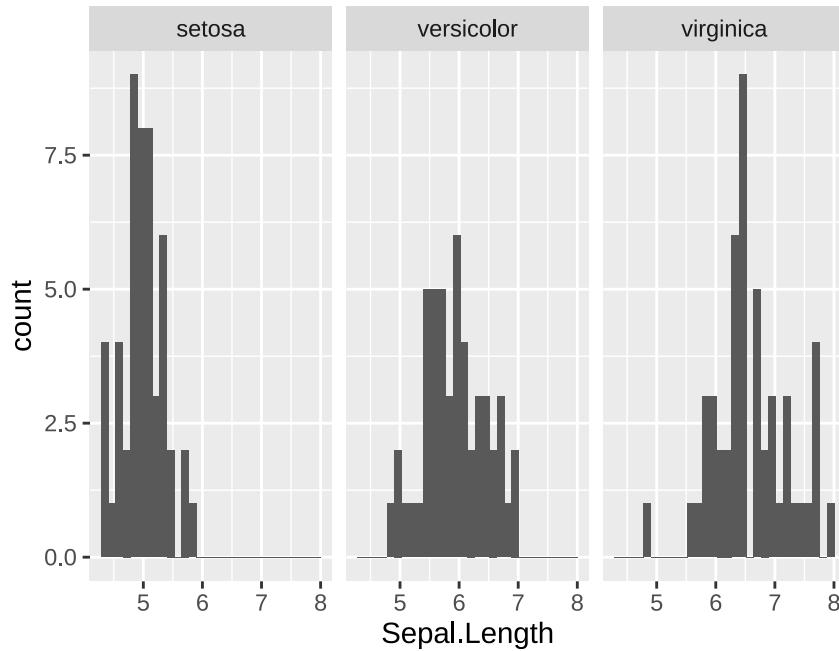


1136

#### 1137 8.4.1 Multipanel Abbildungen

1138 Mit ggplot2 kann man einfach Abbildungen erstellen, die mehrere Panels haben. Das bedeutet, dass eine  
1139 oder mehrere weitere Variablen gibt, die einen Plot in mehrere Subplots teilt. Dafür gibt es zwei Funktionen:  
1140 `facet_grid()` und `facet_wrap()`.

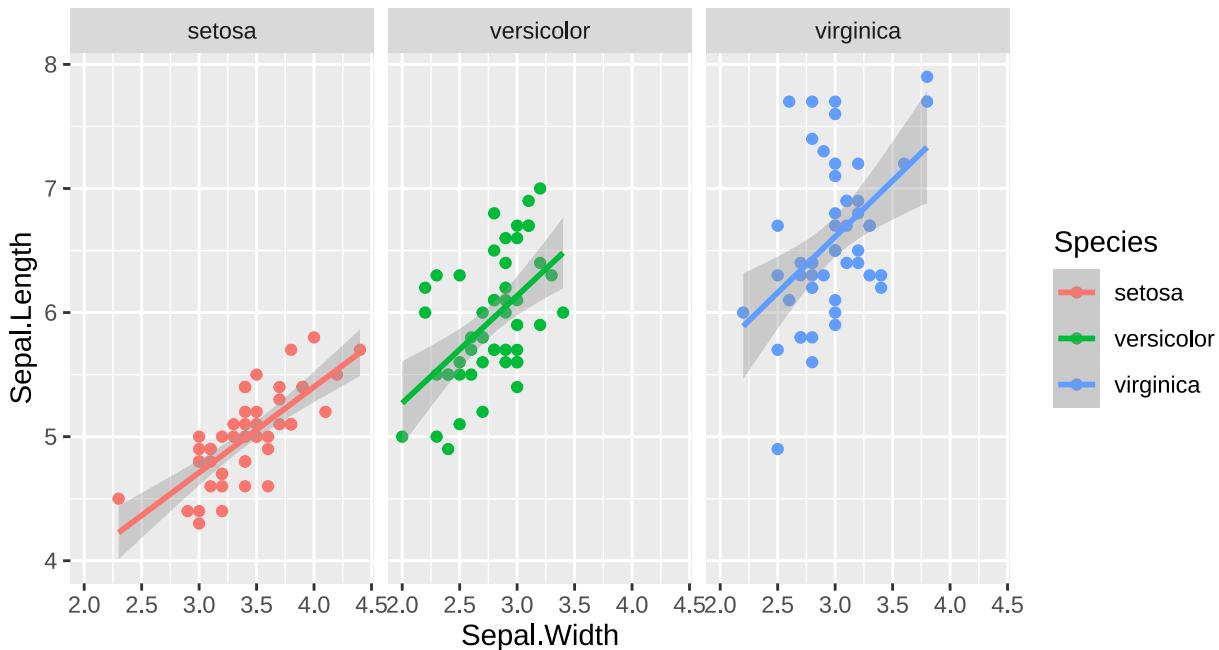
```
ggplot(iris, aes(Sepal.Length)) + geom_histogram() +
 facet_grid(~ Species)
```



1141

1142 Die Funktion `facet_grid()` erzeugt einen *Grid* und beschriftet die Grid-Zeilen und -Spalten, während  
 1143 `facet_wrap()` für jedes Panel (Diagramm) eine eigene Überschrift erzeugt. Beim Arrangieren der Diagramme  
 1144 wird der Vorteil von `ggplot2` gegenüber `base plot` besonders deutlich. Während es im `base plot` System  
 1145 mühsam ist, die Abbildungen zu arrangieren, übernimmt `ggplot2` das Arrangement automatisch und fügt  
 1146 sogar eine automatisch Legende hinzu. Die Achsenabschnitte werden so angepasst, dass die Daten vergleichbar  
 1147 sind.

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



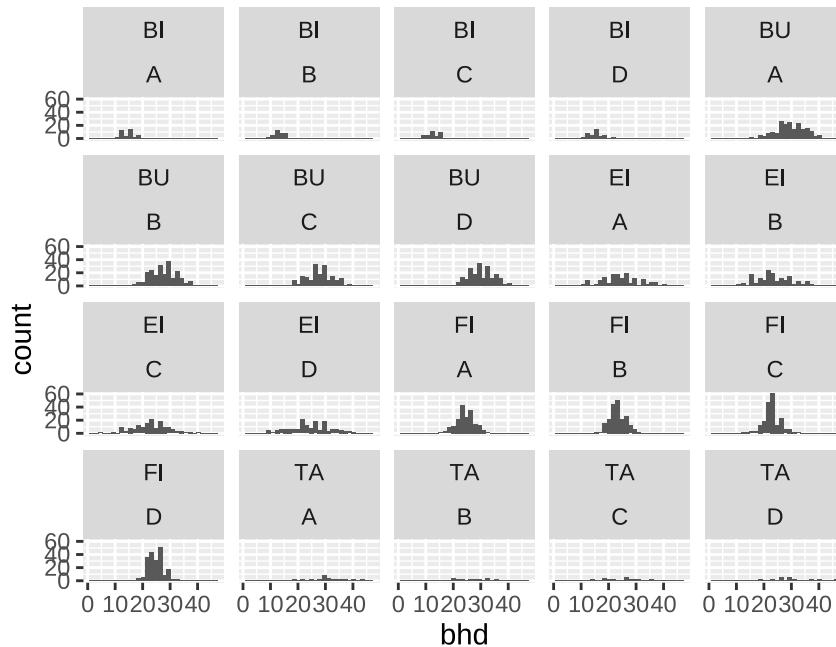
1148

1149

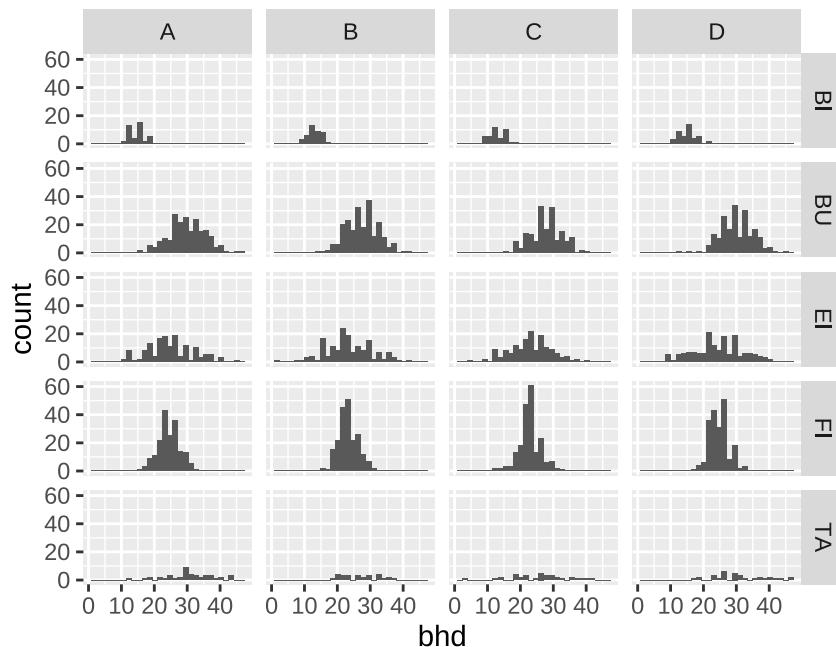
1150 **Aufgabe 22: Multipanel Abbildungen**

1151

- 1152 Lesen Sie erneut den Datensatz `daten/bhd_1.txt` ein und speichern Sie diesen in die Variable (`dat_bhd`).  
 1153 Erstellen Sie für jede Art und Gebiet ein Histogramm. Welche Unterschiede können Sie feststellen, wenn Sie  
 1154 `facet_grid()` oder `facet_wrap()` verwenden?



1155



1156

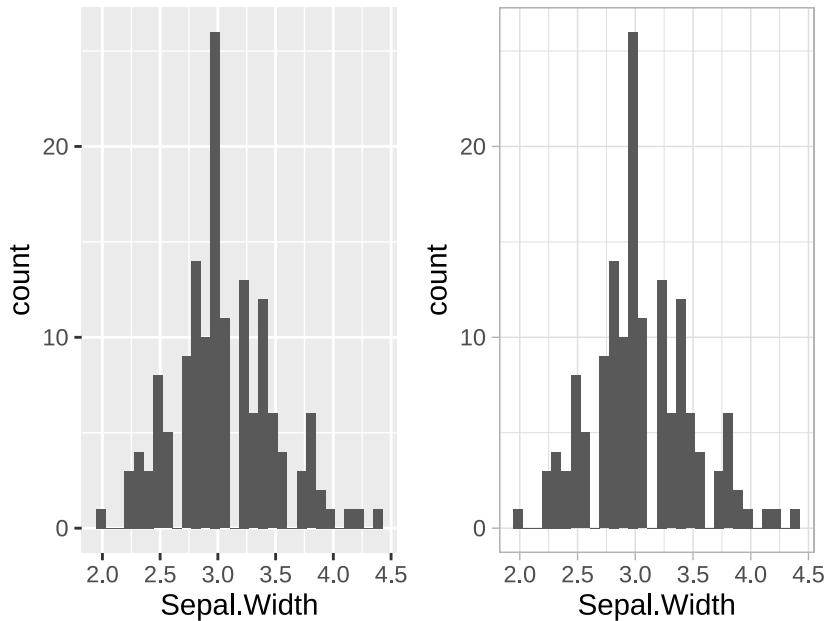
1157 **8.4.2 Plots kombinieren**

- 1158 Es gibt Situationen, in denen **unterschiedliche** Plots miteinander kombiniert werden müssen. Im vorherigen  
 1159 Abschnitt wurde dies immer anhand einer gruppierenden Kovariate gemacht. Aber es gibt auch Situationen, in  
 1160 denen das nicht möglich ist. Beispielsweise wenn ein Histogramm und ein Scatterplot vom gleichen Datensatz  
 1161 zusammengefasst werden sollen. Dafür bietet sich das Paket **patchwork** an<sup>9</sup>.  
 1162 Als erstes können wir zwei (oder natürlich auch mehrere Plots) erstellen. Hier unterscheiden sich die Plots  
 1163 lediglich durch das Aussehen.

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

- 1164 Dann müssen können wir diese Plots ebenfalls mit + zusammenfügen.

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



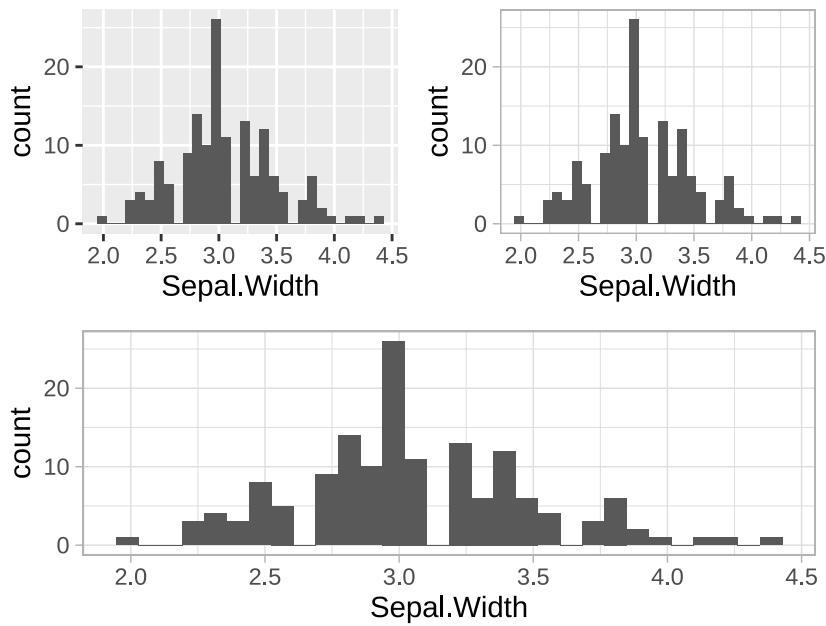
1165

- 1166 Natürlich können auch weitere Plots hinzugefügt werden (auch in unterschiedlichen Dimensionen):

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

---

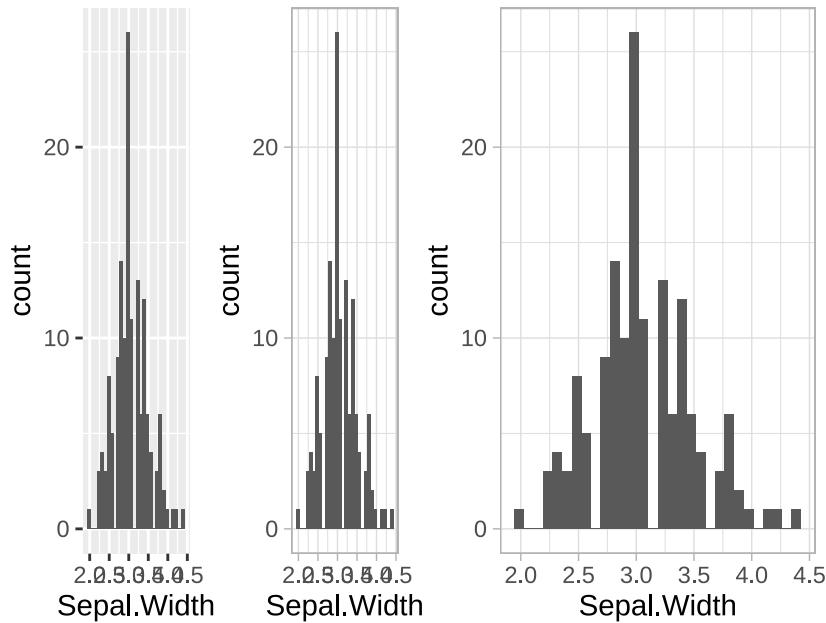
<sup>9</sup>Auch dieses Paket müssen Sie einmalig mit `install.packages("patchwork")` installieren.



1167

Des weiteren können mit | auch Plots gegenüber gestellt werden.

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



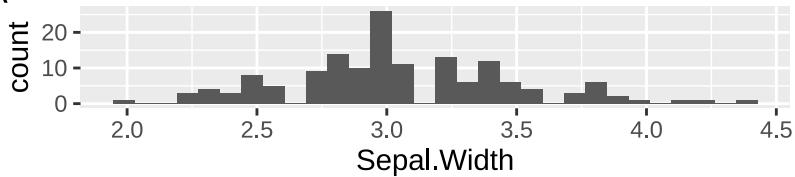
1169

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

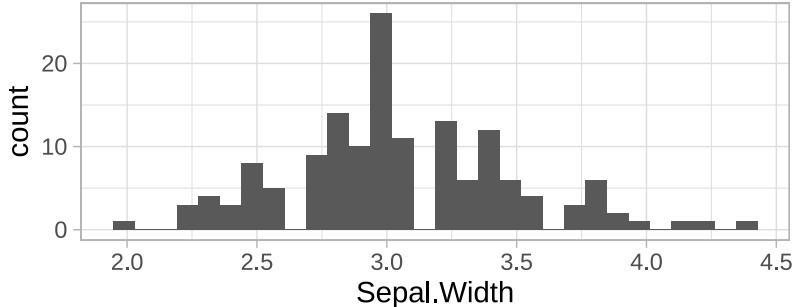
```
p1 + p2 +
 plot_layout(ncol = 1, heights = c(0.3, 0.7)) +
 plot_annotation(title = "Zwei Histogramme", tag_levels = "A")
```

## Zwei Histogramme

A



B



1175

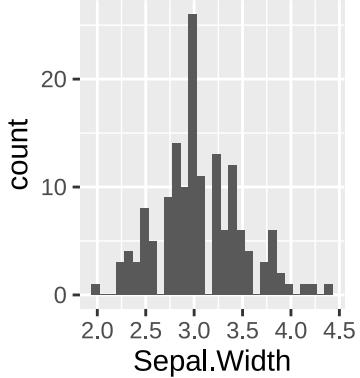
1176

### Aufgabe 23: Mehrere Plots zusammenfügen

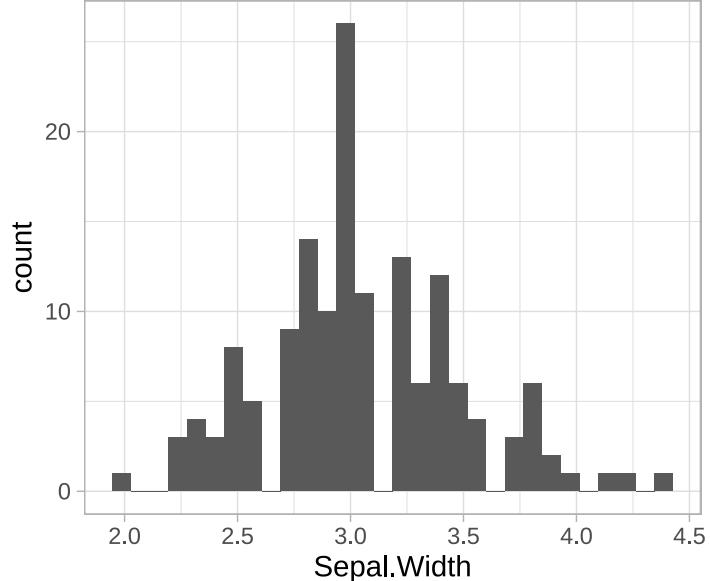
1177

1178 Versuchen Sie die folgende Zusammenstellung der Plots nachzumachen:

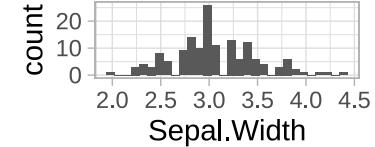
a



c



b



1179

### 8.4.3 Speichern von plots

1180 Sie können mit `ggsave()` eine zwischen-gespeicherte Abbildung exportieren, indem Sie den Variablennamen übergeben. Wenn Sie keine Variable übergeben, wird automatisch die letzte Abbildung gespeichert. Das

- 1184 Dateiformat wird aus dem Dateinamen übernommen. Die Größe der Inhalte können Sie sehr leicht mit den  
1185 Argumenten `width` und `height` oder `scale` gesteuert werden.

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

## 1186 9 Mit Daten arbeiten

### 1187 9.1 dplyr eine Einführung

1188 `dplyr` ist eine Erweiterung von R (ein Paket), die das Ziel hat, den Umgang mit Daten einfacher und  
1189 schneller zu machen.

1190 `dplyr` definiert 5 Verben, um mit Daten zu arbeiten. Diese sind:

- 1191 • `filter`
- 1192 • `select`
- 1193 • `arrange`
- 1194 • `mutate`
- 1195 • `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))
```

1196 Damit die Funktionen aus `dplyr` verwendet werden können, müssen wir als erstes das Paket `dplyr` laden.  
1197 The online Resource [R for Data Science](#) gibt einen sehr guten, tieferen Eindruck in die Arbeit mit `dplyr`  
1198 und erklärt auch die `dplyr` Philosophie noch Mal sehr genau für Beginner.

```
library(dplyr)
```

1199 Sollte dies zu einer Fehlermeldung führen, dann müssen Sie das Paket `dplyr` erst installieren. Dafür müssen  
1200 Sie `einmalig install.packages("dplyr")` installieren.

1201 `dplyr` stellt unterschiedliche Funktionen zum Arbeiten mit Daten zur Verfügung. Es gibt fünf Grundfunktionen  
1202 für die am häufigsten vorkommenden Operationen. Mit der Funktion `filter()` können unterschiedliche  
1203 Beobachtungen gefiltert werden:

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

```
1204 ## id plot bhd alter
1205 ## 1 1 1 50 10
1206 ## 2 2 1 29 30
1207 ## 3 3 2 13 31
1208 ## 4 4 2 23 24
1209 ## 5 5 3 25 25
```

1210 Es können auch mehrere Spalten verwendet werden.

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

```
1211 ## id plot bhd alter
1212 ## 1 2 1 29 30
1213 ## 2 3 2 13 31
1214 ## 3 4 2 23 24
```

```
1215 ## 4 5 3 25 25
```

1216 Natürlich kann genau das gleiche Ergebnis mit dem ‘normalen’ R erreicht werden, dies wäre dann:

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

```
1217 ## id plot bhd alter
```

```
1218 ## 2 2 1 29 30
```

```
1219 ## 3 3 2 13 31
```

```
1220 ## 4 4 2 23 24
```

```
1221 ## 5 5 3 25 25
```

1222 Eine weitere Funktion aus dem Paket **dplyr** ist **select()**. Damit können Spalten aus einem **data.frame**

1223 ausgewählt werden. Dabei können auch die Spaltennamen unbenannt werden.

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

```
1224 ## bhd
```

```
1225 ## 1 50
```

```
1226 ## 2 29
```

```
1227 ## 3 13
```

```
1228 ## 4 23
```

```
1229 ## 5 25
```

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

```
1230 ## bhd id
```

```
1231 ## 1 50 1
```

```
1232 ## 2 29 2
```

```
1233 ## 3 13 3
```

```
1234 ## 4 23 4
```

```
1235 ## 5 25 5
```

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

```
1236 ## BHD id
```

```
1237 ## 1 50 1
```

```
1238 ## 2 29 2
```

```
1239 ## 3 13 3
```

```
1240 ## 4 23 4
```

```
1241 ## 5 25 5
```

1242 Mit der Funktion **arrange()** können die Beobachtungen in einem **data.frame** sortiert werden.

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

```
1243 ## id plot bhd alter
```

```
1244 ## 1 3 2 13 31
```

```
1245 ## 2 4 2 23 24
```

```
1246 ## 3 5 3 25 25
```

```
1247 ## 4 2 1 29 30
1248 ## 5 1 1 50 10
```

1249 Mit der Funktion `desc()` kann die Anordnung in absteigender Reihenfolge sortiert werden.

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

```
1250 ## id plot bhd alter
1251 ## 1 1 1 50 10
1252 ## 2 2 1 29 30
1253 ## 3 5 3 25 25
1254 ## 4 4 2 23 24
1255 ## 5 3 2 13 31
```

1256 Mit der Funktion `mutate()` kann man eine neue Spalte hinzufügen.

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

```
1257 ## id plot bhd alter bhd_mm fl
1258 ## 1 1 1 50 10 500 1963.4954
1259 ## 2 2 1 29 30 290 660.5199
1260 ## 3 3 2 13 31 130 132.7323
1261 ## 4 4 2 23 24 230 415.4756
1262 ## 5 5 3 25 25 250 490.8739
```

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

```
1263 ## id plot bhd alter mean_bhd
1264 ## 1 1 1 50 10 28
1265 ## 2 2 1 29 30 28
1266 ## 3 3 2 13 31 28
1267 ## 4 4 2 23 24 28
1268 ## 5 5 3 25 25 28
```

1269 Mit der Funktion `summarise()` können Spalten zusammengefasst werden.

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

```
1270 ## mean_bhd mean_sd
1271 ## 1 28 13.63818
```

1272

1273 **Aufgabe 24: Datenmanipulation mit dplyr**  
1274

- 1275 1. Laden Sie den Datensatz `data/bhd_1.txt`
- 1276 2. Berechnen Sie folgende Werte für alle Einträge und speichern Sie die Ergebnisse in `erg1`
- 1277 • mittlerer `bhd`
- 1278 • maximales `alter`
- 1279 • die Standardabweichung des BHDs
- 1280 • die Anzahl Bäume mit einem BHD > 30

1281 **9.2 Arbeiten mit gruppierten Daten**

1282 Zusätzlich können `mutate` und `summarise` auch auf gruppierte Daten angewendet werden. Dafür müssen  
1283 wir erst die Funktion `group_by()` aufrufen und als Argumente die Variablen übergeben, die die Gruppen  
1284 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>
```

```
1304 ## <dbl> <dbl>
1305 ## 1 1 39.5
1306 ## 2 2 18
1307 ## 3 3 25
```

1308

1309 **Aufgabe 25: dplyr mit gruppierten Daten**

---

 1310

- 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 **9.3 pipes oder %>%**

1319 Mit *Pipes* (`%>%`) kann man das Ergebnis einer Funktion einfach an eine nachfolgende Funktion weiterreichen.

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

1320 Wir kennen bis jetzt:

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

```
1321 ## [1] 3.333333
```

1322 Mit *Pipes*, die durch das Symbol `%>%` dargestellt werden<sup>10</sup>, können wir das etwas vereinfachen und nacheinander schreiben:

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

```
1324 ## [1] 3.333333
```

1325 Oder sogar

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

```
1326 ## [1] 3.333333
```

1327

1328 **Aufgabe 26: Pipes %>%**

---

 1329

1330 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.

- 1331 1. Laden Sie den Datensatz `data/bhd_1.txt`.
- 1332 2. Berechnen Sie für jede Baumart und Standort die folgenden Werte:
- 1333 • mittlerer `bhd`
  - 1334 • maximales `alter`
  - 1335 • die Standardabweichung des BHDs
  - 1336 • die Anzahl Bäume mit einem BHD > 30
- 1337 3. Ordnen Sie das Ergebnis nach Baumart und BHD.

## 1338 9.4 Joins

1339 Eine weitere häufige Aufgabe beim Daten Management ist es Daten zusammenzuführen. Nehmen Sie an, dass  
1340 wir folgende Aufnahmen gemacht haben

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

1341 und jeder Baum lediglich mit einer `id` versehen wurde. In einer zweiten Tabelle wurden dann weitere Daten  
1342 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")
)
```

1343 Ziel ist es jetzt die Bäume aus `aufnahmen` mit den Informationen aus den `metadaten` zu verbinden. Dazu  
1344 dient `id` als Bindeglied (oft auch Schlüssel genannt).

1345 Dazu gibt es vier Möglichkeiten.

1346 Zur Durchführung gibt es in base R die Funktion `merge()`. Wir werden aber gleich die Funktionen aus dem  
1347 Paket `dplyr` verwenden.

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

1348 ## id bhd art gebiet
1349 ## 1 1 20 <NA> <NA>
1350 ## 2 2 31 Ta A
1351 ## 3 3 74 Bu B

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

1352 ## id bhd art gebiet
1353 ## 1 2 31 Ta A
1354 ## 2 3 74 Bu B
1355 ## 3 4 NA Bu B
```

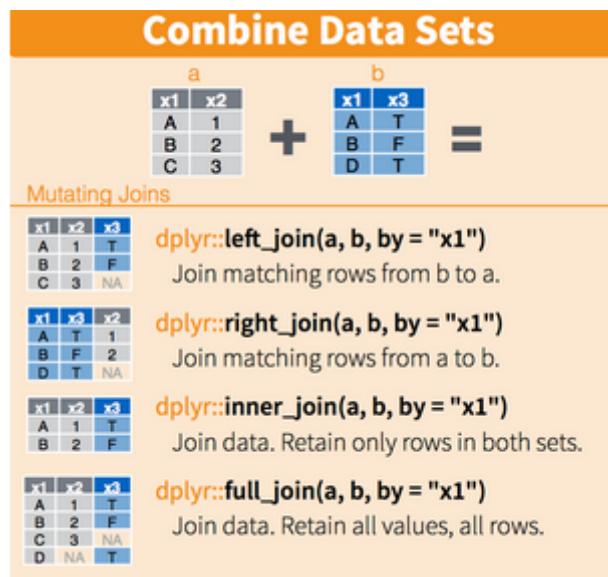


Abbildung 11: Joins (Quelle Rstudio)

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

```
1356 ## id bhd art gebiet
1357 ## 1 2 31 Ta A
1358 ## 2 3 74 Bu B
full_join(aufnahmen, metadaten, by = "id")
```

```
1359 ## id bhd art gebiet
1360 ## 1 1 20 <NA> <NA>
1361 ## 2 2 31 Ta A
1362 ## 3 3 74 Bu B
1363 ## 4 4 NA Bu B
```

1364 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"))
```

```
1365 ## id bhd art gebiet
1366 ## 1 1 20 <NA> <NA>
1367 ## 2 2 31 Ta A
1368 ## 3 3 74 Bu B
```

1369

1370 **Aufgabe 27: Verbinden von Daten**  
1371

- 1372 • Lesen Sie die Datensätze `daten/bhd_2.txt` und `daten/bhd_2_meta.txt` ein.
- 1373 • Stellen Sie sicher, dass es für den `bhd` keine fehlenden Werte gibt (entfernen sie entsprechende Zeilen)
- 1374 • Fügen Sie zu den Metadaten (gespeichert in `bhd_2_meta`) die Anzahl Bäume und den mittleren `bhd`
- 1375 hinzu pro Gebiet.

1376 **9.5 ‘long’ and ‘wide’ Datenformate**

1377 Unter anderem Wickham (2014) empfieilt das Prinzip von *tidy Data*. Nach diesem Prinzip sollten Daten wie  
1378 folgt organisiert sein:

- 1379 • Jede Zeile ist ein Merkmalsträger/Subjekt/Objekt (z. B. eine Person oder ein Baum).
- 1380 • Jede Spalte ist eine Variable (ein Merkmal), die den Merkmalsträger beschreibt.
- 1381 • In jeder Zelle ist genau ein Wert (Merkmalausprägung), nämlich der Wert, der Variable für den Merk-  
1382 malsträger.

1383 Zum Beispiel enthalten Spaltennamen oft Informationen, die eigentlich in einer Variable gespeichert werden  
1384 sollten. Folgendes Beispiel gibt die BHD Messung von 3 Bäumen in 3 Jahren wieder. Der Vorteil von *tidy*  
1385 Daten ist, dass viele Funktionen diese Form erwarten. Somit müssen sie Ihre Daten nur ein Mal organisieren  
1386 und können fast alle Analysen durchführen.

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

1387 Der `tibble` aus dem Paket `tibble` ist ein moderner Data Frame, der sich in seinen Funktionen in das `tidy`  
1388 Universum einfügt. Wir können auch einfach das Paket `tidyverse` laden, um alle Pakete, die Teil des `tidy`  
1389 Universums sind gleichzeitig zu laden. Der `tibble` beinhaltet alle Funktionen, die der Data Frame auch  
1390 beinhaltet und kann auch für genau die gleichen Zwecke verwendet werden. Der `tibble` hat u. A. eine  
1391 modernere Darstellung im Konsolenoutput.

1392 Diese Daten sind jetzt im `wide`-Format gespeichert und folglich nicht `tidy`, weil Information über die Daten  
1393 (nämlich das Jahr der Aufnahme in den Spaltennamen gespeichert sind). Besser wäre eine Struktur mit  
1394 nur drei Spalten: `id`, `jahr` und `bhd`. Um die Daten in so eine Struktur zu bringen, gibt es die Funktion  
1395 `pivot_longer()` aus dem Paket `tidyr`, die ebenfalls Teil des `tidyverse` ist.

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

1396 ## # A tibble: 9 x 3
1397 ## id name value
```

```

1398 ## <int> <chr> <dbl>
1399 ## 1 1 bhd2015 30
1400 ## 2 1 bhd2016 31
1401 ## 3 1 bhd2017 34
1402 ## 4 2 bhd2015 31
1403 ## 5 2 bhd2016 31
1404 ## 6 2 bhd2017 32
1405 ## 7 3 bhd2015 32
1406 ## 8 3 bhd2016 33
1407 ## 9 3 bhd2017 33

```

1408 Wenn wir die Spalten für die Variable und den Wert gleich sinnvoll benennen möchten, können wir das über  
1409 die Argumente `names_to` und `value_to` machen.

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

```

1410 ## # A tibble: 9 x 3
1411 ## id jahr bhd
1412 ## <int> <chr> <dbl>
1413 ## 1 1 bhd2015 30
1414 ## 2 1 bhd2016 31
1415 ## 3 1 bhd2017 34
1416 ## 4 2 bhd2015 31
1417 ## 5 2 bhd2016 31
1418 ## 6 2 bhd2017 32
1419 ## 7 3 bhd2015 32
1420 ## 8 3 bhd2016 33
1421 ## 9 3 bhd2017 33

```

1422 Analog zu der Funktion `pivot_longer()` gibt es auch die Funktion `pivot_wider()`, um vom Daten vom  
1423 `long`-Format ins `wide`-Format zu transformieren.

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

```

1424 ## # A tibble: 3 x 4
1425 ## id bhd2015 bhd2016 bhd2017
1426 ## <int> <dbl> <dbl> <dbl>
1427 ## 1 1 30 31 34
1428 ## 2 2 31 31 32
1429 ## 3 3 32 33 33

```

1430

---

1431 **Aufgabe 28: Zeitliche Verlauf von BHDs**

---

1433 In der Datei `bhd_3.csv` befinden sich gemessene BHDs (in cm) von unterschiedlichen Bäumen zu unter-  
 1434 schiedlichen Zeitpunkten. Erstellen Sie ein Liniendiagramm, das den zeitlichen (x-Achse) Verlauf der BHDs  
 1435 (y-Achse) für die unterschiedlichen Bäume darstellt.

1436 **9.6 Auswählen von Variablen**

1437 Sobald die Datensätze etwas umfangreicher werden (d. h., es gibt mehrere Spalten in einem `data.frame`),  
 1438 können innerhalb vieler `dplyr`-Funktionen spezielle Funktionen verwendet werden, um Variablen auszuwählen.  
 1439 Wenn die genaue Position der Spalten bekannt ist, kann man mit dem `:`-Operator und der Position Spalten  
 1440 auswählen:

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

```
1441 ## Sepal.Length Sepal.Width Petal.Length
1442 ## 1 5.1 3.5 1.4
1443 ## 2 4.9 3.0 1.4
1444 ## 3 4.7 3.2 1.3
```

1445 Diese Vorgehensweise kann gehährlich sein, da sich manchmal Spalten verschieben und sich somit die  
 1446 Positionen ändern. Es besser Spalten immer explizit mit Namen statt mit Nummern anzusprechen.

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

```
1447 ## Sepal.Length Sepal.Width Petal.Length
1448 ## 1 5.1 3.5 1.4
1449 ## 2 4.9 3.0 1.4
1450 ## 3 4.7 3.2 1.3
```

1451 `select()` erlaubt es, auch hier den `:`-Operator zu verwenden:

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

```
1452 ## Sepal.Length Sepal.Width Petal.Length
1453 ## 1 5.1 3.5 1.4
1454 ## 2 4.9 3.0 1.4
1455 ## 3 4.7 3.2 1.3
```

1456 Es gibt auch einige spezielle Funktionen, um Spalten innerhalb eines `select()`-Aufrufs auszuführen:

- 1457 • `starts_with()`: Hier kann man ein Muster angeben, mit dem ein Text anfangen muss.
- 1458 • `ends_with()`: Diese Funktion ist analog zu `starts_with()`, jetzt wird aber am Ende des Spaltennamens  
 1459 nach dem Muster gesucht.
- 1460 • `contains()`: Hier kann ein Muster übergeben werden, das irgendwo im Spaltennamen sein muss.
- 1461 • `everything()`: Mit dieser Funktion werden alle Spalten ausgewählt.

1462 • `last_col()`: Mit dieser Funktion wird nur die letzte Spalte ausgewählt (dass ist die Spalte, die ganz  
1463 rechts ist).

1464 Sämtliche Auswahlen können mit - umgekehrt werden.

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

1465 ## Sepal.Length Sepal.Width

1466 ## 1 5.1 3.5

1467 ## 2 4.9 3.0

1468 ## 3 4.7 3.2

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

1469 ## Petal.Length Petal.Width Species

1470 ## 1 1.4 0.2 setosa

1471 ## 2 1.4 0.2 setosa

1472 ## 3 1.3 0.2 setosa

1473 `select()` bietet auch noch die Möglichkeit, Spaltennamen zu ändern.

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

1474 ## sep\_width

1475 ## 1 3.5

1476 ## 2 3.0

1477 ## 3 3.2

1478

### 1479 Aufgabe 29: Auswählen von Spalten

---

1481 In der Datei `messungen_1.csv` sind Messungen von zwei Sensoren enthalten für die ersten vier Monate eines  
1482 Jahres. Führen Sie folgende Abfragen durch:

1483 1. Wählen Sie alle Messungen für Januar aus.

1484 2. Wählen Sie alle Messungen für Januar und März aus.

## 1485 9.7 Einzelne Beobachtungen abfragen (`slice()`)

1486 Mit dem Befehl `slice()` kann man einzelne Beobachtungen (= Zeilen) aus einem `data.frame` abfragen.

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

1487 ## Sepal.Length Sepal.Width Petal.Length Petal.Width Species

1488 ## 1 5.1 3.5 1.4 0.2 setosa

1489 ## 2 4.4 2.9 1.4 0.2 setosa

1490 ## 3 5.1 3.5 1.4 0.3 setosa

1491 Davon gibt es drei nützliche Varianten: 1) `slice_head()` und `slice_tail()`; 2) `slice_max()` und  
 1492 `slice_min()`; 3) `slice_random()`.

1493 `slice_head()` und `slice_tail()` sind analog zu `head()` und `tail()`, aber mit dem entscheidenden Unter-  
 1494 schied, dass Gruppierungen berücksichtigt werden. Wenn keine Gruppierung in den Daten vorhanden ist, gibt  
 1495 es keinen Unterschied.

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

```
1496 ## Sepal.Length Sepal.Width Petal.Length Petal.Width Species
1497 ## 1 5.1 3.5 1.4 0.2 setosa
1498 ## 2 4.9 3.0 1.4 0.2 setosa
```

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

```
1499 ## Sepal.Length Sepal.Width Petal.Length Petal.Width Species
1500 ## 1 5.1 3.5 1.4 0.2 setosa
1501 ## 2 4.9 3.0 1.4 0.2 setosa
```

1502 Sobald jedoch eine gruppierende Variable eingeführt wird, gibt `slice_head()` die ersten n Beobachtungen  
 1503 für jede Gruppe zurück und `head()` für den gesamten Datensatz.

```
base head
```

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

```
1504 ## # A tibble: 2 x 5
1505 ## # Groups: Species [1]
1506 ## Sepal.Length Sepal.Width Petal.Length Petal.Width Species
1507 ## <dbl> <dbl> <dbl> <dbl> <fct>
1508 ## 1 5.1 3.5 1.4 0.2 setosa
1509 ## 2 4.9 3 1.4 0.2 setosa
```

```
dplyr slice_head
```

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

```
1510 ## # A tibble: 6 x 5
1511 ## # Groups: Species [3]
1512 ## Sepal.Length Sepal.Width Petal.Length Petal.Width Species
1513 ## <dbl> <dbl> <dbl> <dbl> <fct>
1514 ## 1 5.1 3.5 1.4 0.2 setosa
1515 ## 2 4.9 3 1.4 0.2 setosa
1516 ## 3 7 3.2 4.7 1.4 versicolor
1517 ## 4 6.4 3.2 4.5 1.5 versicolor
1518 ## 5 6.3 3.3 6 2.5 virginica
1519 ## 6 5.8 2.7 5.1 1.9 virginica
```

1520 `slice_tail()` funktioniert analog zu `slice_head()` mit dem einzigen Unterschied, dass nicht die ersten n

1521 Zeilen zurück gegeben werden sondern die letzten **n** Zeilen.  
 1522 `slice_max()` und `slice_min()` geben die Beobachtung mit dem maximalen bzw. minimalen Wert einer  
 1523 Variable zurück. Auch hier werden Gruppen berücksichtigt.

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

1524 ## Sepal.Length Sepal.Width Petal.Length Petal.Width Species  
 1525 ## 1 7.9 3.8 6.4 2 virginica

1526 Und mit Gruppen:

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

1527 ## # A tibble: 3 x 5  
 1528 ## # Groups: Species [3]  
 1529 ## Sepal.Length Sepal.Width Petal.Length Petal.Width Species  
 1530 ## <dbl> <dbl> <dbl> <dbl> <fct>  
 1531 ## 1 5.8 4 1.2 0.2 setosa  
 1532 ## 2 7 3.2 4.7 1.4 versicolor  
 1533 ## 3 7.9 3.8 6.4 2 virginica

1534 `slice_min()` funktioniert genau gleich, nur dass die Beobachtung (=Zeile) mit dem kleinsten Wert einer  
 1535 Variable zurück gegeben wird.

1536 Die Funktion `slice_sample()` erlaubt es zufällige Beobachtungen zu ziehen. Dabei kann über das Argument **n**  
 1537 die Anzahl an Beobachtungen angegeben werden oder über das Argument **prop** der Anteil an Beobachtungen.

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

1538 ## Sepal.Length Sepal.Width Petal.Length Petal.Width Species  
 1539 ## 1 6.5 2.8 4.6 1.5 versicolor  
 1540 ## 2 6.3 3.3 4.7 1.6 versicolor  
 1541 ## 3 7.2 3.2 6.0 1.8 virginica  
 1542 ## 4 4.9 3.6 1.4 0.1 setosa  
 1543 ## 5 6.0 2.7 5.1 1.6 versicolor

1544 Das Ergebnis ist bei jedem von Ihnen anders, da es sich um eine zufällige Ziehung handelt. Wenn Sie diese  
 1545 Ergebnisse wiederholen möchten, können Sie über `set.seed()` die zufällige Ziehung reproduzierbar machen.

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

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

1546 ## Sepal.Length Sepal.Width Petal.Length Petal.Width Species  
 1547 ## 1 4.3 3.0 1.1 0.1 setosa  
 1548 ## 2 5.0 3.3 1.4 0.2 setosa  
 1549 ## 3 7.7 3.8 6.7 2.2 virginica  
 1550 ## 4 4.4 3.2 1.3 0.2 setosa  
 1551 ## 5 5.9 3.0 5.1 1.8 virginica

1552 Wenn beispielsweise 5% der Beobachtungen gezogen werden sollen, kann dies so gemacht werden:

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

```
1553 ## Sepal.Length Sepal.Width Petal.Length Petal.Width Species
1554 ## 1 7.7 3.8 6.7 2.2 virginica
1555 ## 2 5.5 2.5 4.0 1.3 versicolor
1556 ## 3 5.5 2.6 4.4 1.2 versicolor
1557 ## 4 6.5 3.0 5.2 2.0 virginica
1558 ## 5 6.1 3.0 4.6 1.4 versicolor
1559 ## 6 6.3 3.4 5.6 2.4 virginica
1560 ## 7 5.1 2.5 3.0 1.1 versicolor

1561 slice_sample() berücksichtigt ebenfalls Gruppen. Mit den Argumenten replace und weight_by dann die
1562 Zufallsziehung genauer spezifiziert werden. replace sagt, ob eine gezogenen Beobachtung wieder zurück gelegt
1563 wird oder nicht. Mit dem Argument weight_by können optional gewichtete für jede Beobachtung vergeben
1564 werden.
```

1565

### 1566 Aufgabe 30: Daten beschreiben

---

1568 Verwenden Sie den Datensatz `bhd_1.txt` und finden Sie für jedes Gebiet und Art die Beobachtung mit
1569 kleinsten BHD.

## 1570 9.8 Spalten trennen

1571 Ein gut geplanter Datensatz besteht aus Beobachtungen (Zeilen), Variablen (Spalten) und in jeder Zelle ist
1572 immer ein **genau** ein Wert gespeichert. Leider gibt es oft Datensätze, bei denen mehr als ein Wert pro Zelle
1573 gespeichert wurde. Die Funktion `separate()` kann helfen solche Daten zu trennen.

1574 Wir verwenden einen erfunden Datensatz zu Beobachtungen von Tieren und einer geschätzten Distanz zu
1575 diesen Tieren.

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

1576 In der Spalte `beobachtung` sind zwei Informationen gespeichert: Die Distanz zur Beobachtung und die Art.
1577 Das ist ungünstig, weil wir so weder nach Tierart noch nach distanz filtern können (nicht *tidy*). Mit der
1578 Funktion `separate()`, können wir Beobachtungen einer Spalte in mehrere Spalten trennen. Dafür muss der
1579 Spaltennamen (Argument `col`), die neuen Spaltennamen (Argument `into`) und das Trennzeichen (Argument
1580 `sep`) angegeben werden.

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

1581 ## # A tibble: 4 x 3

```
1582 ## id Distanz Art
1583 ## <int> <chr> <chr>
1584 ## 1 1 10m " Reh"
1585 ## 2 2 100m " Reh"
1586 ## 3 3 20m " Fuchs"
1587 ## 4 4 40 "Reh"

1588 Nach dem Aufruf von separate() gibt es zwei neue Spalten (Distanz und Art), die die alte Spalte
1589 beobachtung ersetzen.
```

1590

---

### 1591 Aufgabe 31: Aufräumen

---

1593 Verwenden Sie den folgenden Datensatz und bringen Sie ihn in eine Form, die sicherstellt dass

- 1594 • jede Zelle genau einen Wert enthält.  
1595 • jede Zeile eine Beobachtung ist.  
1596 • 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")
)
```

## 1597 10 Arbeiten mit Text

1598 Bis jetzt haben wir fast ausschließlich mit Zahlen oder Abbildungen gearbeitet. R bietet aber auch viele  
 1599 Werkzeuge, um mit Text zu arbeiten. Wir wollen hier ein paar Funktionen dafür vorstellen. Als erstes sollte  
 1600 nochmals klar gestellt werden, was eigentlich ein Text ist. In R ist alles, das innerhalb von doppelten ("") oder  
 1601 einfachen ('') Anführungszeichen geschrieben ist, Text.

1602 Anbei einige Beispiele:

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

1603 Wichtig ist hier zu sehen, dass z nicht als Zahl sondern, als Text interpretiert wird (siehe Datentypen).

```
z + 1
```

1604 ## Error in z + 1: nicht-numerisches Argument für binären Operator

1605 Wenn man sicher ist, dass es sich bei einem Textobjekt um eine Zahl handelt, kann man dies mit der Funktion  
 1606 `as.numeric()` in eine Zahl umwandeln.

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

1607 ## [1] 31

1608 Aber mit a führt dies wieder zu einem NA-Wert, da a nicht in eine Zahl umgewandelt werden kann.

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

1609 ## Warning: NAs durch Umwandlung erzeugt

1610 ## [1] NA

### 1611 10.1 Arbeiten mit Text

1612 Wir wollen erst einmal drei Funktionen besprechen, die es erlauben mit Text zu arbeiten. Die Funktion  
 1613 `nchar()`<sup>11</sup> gibt an wie viele Zeichen ein Text hat. Also z.B.

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

1614 ## [1] 5

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

1615 ## [1] 2

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

1616 ## [1] 20

1617 Die Funktion `paste()` erlaubt es verschiedene Variablen mit Text zu verbinden. Wenn wir z. B. die Variablen  
 1618 `vorname <- "Eva"` und `name <- "Meier"` haben und wir wollen eine neue Variable `full_name <- Eva`

---

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

1619 "Meier" erzeugen, dann kann das mit der Funktion `paste()` gemacht werden.

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

1620 ## [1] "Eva Meier"

1621 Die Funktion `paste()` hat das Argument `sep`, das auf ein Leerzeichen ( ) gesetzt ist, aber auch anders sein  
1622 kann und das Trennzeichen definiert.

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

1623 ## [1] "Eva, Meier"

1624 Die Funktion `substr()` erlaubt es am Anfang oder Ende eines Wortes etwas abzuschneiden. Dabei muss  
1625 immer die Anfangs- (`start`) und Endposition (`stop`) angegeben werden.

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

1626 ## [1] "Hal"

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

1627 ## [1] "allo"

1628

### 1629 Aufgabe 32: Arbeiten mit Text 1

---

1631 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")
```

1632 1. Aus wie vielen Buchstaben besteht jedes Wort?

1633 2. Finden Sie das längste Wort.

1634 3. Wie viel Prozent der Wörter fangen mit einem S an?

1635 4. Fügen Sie jedem Wort seine Position im Vektor hinzu. Beispielsweise soll aus `Vogel` "2. Vogel" werden  
1636 usw.

## 1637 10.2 Finden von Textmustern

1638 Mit der Funktion `grep()` können Muster in einem Text gefunden werden. Wenn wir beispielsweise folgenden  
1639 Vektor mit Textelementen haben.

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

1640 Und wir wollen alle Straßennamen die ein `weg` haben abfragen, dann können wir folgenden Befehl ausführen:

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

1641 ## [1] 2

1642 Im zweiten Element von `txt` kommen die Zeichen `Weg` vor. Beachte, in der Standardeinstellung wird zwischen  
1643 Groß- und Kleinschreibung unterschieden. Dies kann mit dem Argument `ignore.case = TRUE` angepasst  
1644 werden.

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

1645 ## [1] 1 2

1646 Mit der Funktion `sub` können Zeichen innerhalb einer Zeichenkette ersetzt werden.

1647 So ersetzt der folgende Ausdruck `ae` mit `ä`.

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

1648 ## [1] "Friedländer Weg"

1649 Wenn allerdings das zu ersetzende Zeichen mehr als einmal vorkommt und beide Instanzen ersetzt werden  
1650 sollen braucht man die Funktion `gsub`.

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

1651 ## [1] "Friedländer Weg und Reinhäuser Landstrasse sind zwei Strassen in Goettingen."

1652 Mit `sub()` wird nur das erste `ae` ersetzt, während `gsub()` alle `ae` mit einem `ä` ersetzt.

```
gsub("ae", "ä", txt)
```

1653 ## [1] "Friedländer Weg und Reinhäuser Landstrasse sind zwei Strassen in Goettingen."

1654 Oft ist der genaue Ausdruck den man finden möchte jedoch Variabel. Beispielsweise möchte man alle Wörter  
1655 mit einem Umlaut oder Zahlen finden möchte, kann man das oft abkürzen. Dafür gibt es reguläre Ausdrücke.  
1656 Wir werden hier nur ein paar beispielhafte Anwendungen besprechen.

1657 Sowohl in den Funktionen `grep()` als auch `(g)sub()` kann mit anstatt dem Muster (immer das erste  
1658 Argument) aus ein regulärer Ausdruck angegeben werden. Mit `[1-9]` sind alle Zahlen von 1 bis 9 gemeint.

1659 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)
```

1660 ## [1] 1 3

1661 Damit lässt sich auch das Problem mit groß- und kleingeschriebenen Wörtern lösen.

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

```
1662 ## [1] 1 2
```

1663

---

**Aufgabe 33: Arbeiten mit Text 2**

---

1664 Verwenden Sie den folgenden Vektor:

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

1667 1. In wie vielen Wörtern kommt der Doppellaut au vor?

1668 2. Ersetzen Sie in allen Wörtern alle au mit \_ \_.

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

```
1669 ## [1] 5 6 13
```

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

```
1670 ## [1] "Versicherung" "Methoden" "Fluss" "Rudel" "B_ _m"
1671 ## [6] "H_ _s" "Foto" "Auffahrt" "Auto" "Handy"
1672 ## [11] "Teller" "Kalender" "Aufb_ _"
```

## 1673 11 Arbeiten mit Zeit

1674 Für den Computer bzw. R ist ein Datum erst einmal nichts anderes als ein Text. Für uns ist es sofort  
 1675 klar, dass der "13.2.2021" der 13. Februar 2021 ist, für den Computer zunächst nicht. Wir müssen R also  
 1676 irgendwie sagen, dass die 13 der Tag ist, die 2 der Monat und 2021 das Jahr. Dass der Computer die einzelnen  
 1677 Komponenten erkennt, und in einen Datentyp eigens für Zeitformate überführt, nennt man *parsen*<sup>12</sup>. Durch  
 1678 das *parsen* wird die Variable in den Datentyp **Date** überführt. Das Arbeiten mit Datum und Zeit kann  
 1679 kann anfangs sehr mühsam sein und viele Zeit-spezifischen Datenoperationen lassen sich auch mit den  
 1680 Basis-Datentypen durchführen. Sobald man einige Grundfertigkeiten erworben hat, stellt man jedoch fest,  
 1681 dass die Arbeit mit dem Zeitformat-Datentyp schneller und effizienter funktioniert. Starten Sie am besten  
 1682 gleich mit "echten" Zeit-Datentypen und versuchen Sie nicht, sich irgendwie mit Text und numerischen  
 1683 Datentypen selbst etwas zu basteln. Der erste Schritt ist immer ein Datum zu *parsen*. Wir verwenden dafür  
 1684 Funktionen 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)
```

1685 **lubridate** bietet eine Vielzahl von Funktionen zum *parsen* von Datum und Zeit, die sich aus:

- 1686 • y für Jahr,
- 1687 • m für Monat,
- 1688 • d für Tag,
- 1689 • h für Stunde,
- 1690 • m für Minute und
- 1691 • s für Sekunde

1692 zusammen setzten. Alle Funktionen nehmen als erstes Argument ein Textstring. Wenn wir z.B. den String  
 1693 2020-01-20 parsen wollen können wir das mit der Funktion **ymd** machen.

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

1694 ## [1] "2020-01-20"

1695 Dabei erkennt **lubridate** in der Regel die Trennzeichen:

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

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

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

1697 ## [1] "2020-01-20"

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

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

1699 Wenn die die Anordnung der einzelnen Komponenten anders ist, gibt es einfach eine andere Funktion.

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

```

dmy("20.1.2020")

1700 ## [1] "2020-01-20"
1701 Jetzt stellt sich die Frage, was der Vorteil ist, wenn R ein Datum parst.
d <- dmy("20.1.2020")

1702 Wir können jetzt mit d arbeiten und einzelne Komponenten extrahieren.
day(d)

1703 ## [1] 20
month(d)

1704 ## [1] 1
year(d)

1705 ## [1] 2020
1706 Oder auch Zeiteinheiten hinzufügen oder abziehen.
d + days(10)

1707 ## [1] "2020-01-30"
d - years(20)

1708 ## [1] "2000-01-20"
d + hours(25)

1709 ## [1] "2020-01-21 01:00:00 UTC"

```

1710

---

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

---

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

**11.1 Arbeiten mit Zeitintervallen**

- 1717 Mit zwei Zeitpunkten lassen sich Zeitintervalle (**Periods**) erstellen, dafür können wir die Funktion **interval()** aus dem Paket **lubridate** verwenden<sup>13</sup>.

---

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

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

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

1720 Wir können jetzt mit `int` arbeiten und beispielsweise das Intervall verschieben,

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

1721 ## [1] 2023-03-18 UTC--2024-03-18 UTC

1722 die Länge des Intervalls berechnen

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

1723 ## [1] 31536000

1724 oder testen, ob ein Datum innerhalb des Intervalls liegt.

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

1725 ## [1] TRUE

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

1726 ## [1] FALSE

1727 Intervalle können auch zum Selektieren von Daten verwendet werden. Z. B. im `dplyr` Stil.

```
d <- tibble(a = c(ymd("2021-07-1"), ymd("2020-07-1")))
d |> filter(a %within% int)
```

1728 ## # A tibble: 1 x 1

1729 ## a

1730 ## <date>

1731 ## 1 2020-07-01

1732 `%within%` funktioniert genauso mit Vekotren oder mit mehreren Intervallen. Wir könnten also zwei Intervalle  
1733 definieren (z.B. Ostern und Pfingsten).

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

1734 Und Überprüfen welche Termine in eines der zwei Intervalle fallen.

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

```
Ostern
```

```
termine %within% ostern
```

1735 ## [1] FALSE TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE

```
Pfingsten
termine %within% pfingsten

1736 ## [1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE TRUE FALSE FALSE
1737 Der Zeit-Datentyp wird auch oft benutzt, um den Zeitbedarf von längeren Berechnungen zu messen.

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

1738 ## [1] 0.4999484
t2 <- now()
int_length(interval(t1, t2))

1739 ## [1] 0.6111724
```

## 11.2 Formatieren von Zeit

1740 Für die Ausgabe in Berichten oder Grafiken soll das Datum oft in einer speziellen Form dargestellt werden.  
 1741 Die Funktion `format()` bietet Möglichkeiten ein Datumsobjekt zurück in Text umzuwandeln.  
 1742 Ein Beispiel wäre `ymd("2021-2-10")` als 10.2.21 auszugeben.

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

1743 Dabei handelt sich bei `%d.%m.%y` um Abkürzungen für die unterschiedlichen Komponenten eines Datumobjekts.  
 1744 Siehe dazu die Hilfeseite von `strptime (help(strptime))`.

1747

### Aufgabe 35: Arbeiten mit Intervallen

1748 Wie viele Einträge aus dem Vektor `v1` befinden sich in einem Intervall, das zwischen dem 1.3.2021 und dem  
 1749 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"
)
```

## 11.3 Zeitreihen

1750 In der Forstbranche haben wir oft mit Zeitreihen zu tun. Zeitreihen sind Variablen, für die in zeitlichen  
 1751 Intervallen Daten vorliegen. In der Regel monatlich, vierteljährig oder jährlich, wobei die Intervalle zwischen  
 1752 den Messungen bei Zeitreihen immer gleich lang sind. Wiederholungsmessungen von Forsteinventuren (Forstein-  
 1753 richtungen, Betriebsinventuren, die meisten forstl. Experimente, die BWI, ...) sind also in der Regel keine

1757 Zeitreihen in engeren Sinne. Turnusmäßig vermessene Versuchsflächen, wie sie z. B. die Versuchsanstalten  
 1758 unterhalten oder jährlich gemeldete Holzpreise jedoch schon.

1759 Zeitreihen unterscheiden sich nicht nur technisch, sondern auch inhaltlich fundamental von den uns schon  
 1760 bekannten Daten. Statistisch gesehen haben Zeitreihen Besonderheiten gegenüber anderen Variablen, da  
 1761 Sie von Ihrer eigenen Vergangenheit abhängen (autokorriert sind) und auch die Abhängigkeit anderer  
 1762 Variablen in der Regel nur sinnvoll ist, wenn deren Vergangenheit mitberücksichtigt wird (cross-correlation).  
 1763 Konventionelle Statistik ist oft nicht möglich, um Zeitreihen zu analysieren. Selbst ein ordinärer arithmetischer  
 1764 Mittelwert ist schon nicht mehr geeignet, um Zeitreihen statistisch zu beschreiben. Angefangen mit der  
 1765 Datendarstellung gibt es in R deshalb spezifische Zeitreihen-Funktionen. Aus diesem Grund sollten Sie  
 1766 Zeitreihen als solche speichern. Viele Funktionen erkennen den Datentyp und führen entsprechend spezifische  
 1767 Zeitreihen-Operationen durch, wenn ihnen Daten vom Typ "Zeitreihe" übergeben werden. Laden wir z. B.  
 1768 die Holzpreise für Fichte 2b (das sog. Leitsortiment, Fichenholz mit einem Mittendurchmesser von 20 bis 25  
 1769 cm), das Holzaufkommen dieses Sortiments (Einschlagsvolumen) und die Preise für Nadelholz vom  
 1770 statistischen Bundesamt<sup>14</sup>. Wir laden die Daten zunächst als csv ein:

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

1771 Diese 3 Zeitreihen bilden zusammen ein klassisches Marktmodell mit dem Preis eines homogenen Gutes  
 1772 (Leitsortimentspreis), dem Angebot (Holzeinschlag) und der Nachfrage (Schnittholzpreis). Mit der Funktion  
 1773 **ts** werden die Daten in ein Zeitreihenobjekt überführt (*pasrse*). Die Spalte mit den Jahren ist dann nicht mehr  
 1774 nötig, da die Informationen zur Zeit keine eigene Variable mehr sind, sondern als sog. Metainformationen in  
 1775 dem Objekt gespeichert wird. Die Spalten sollten nur noch Daten enthalten.

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

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

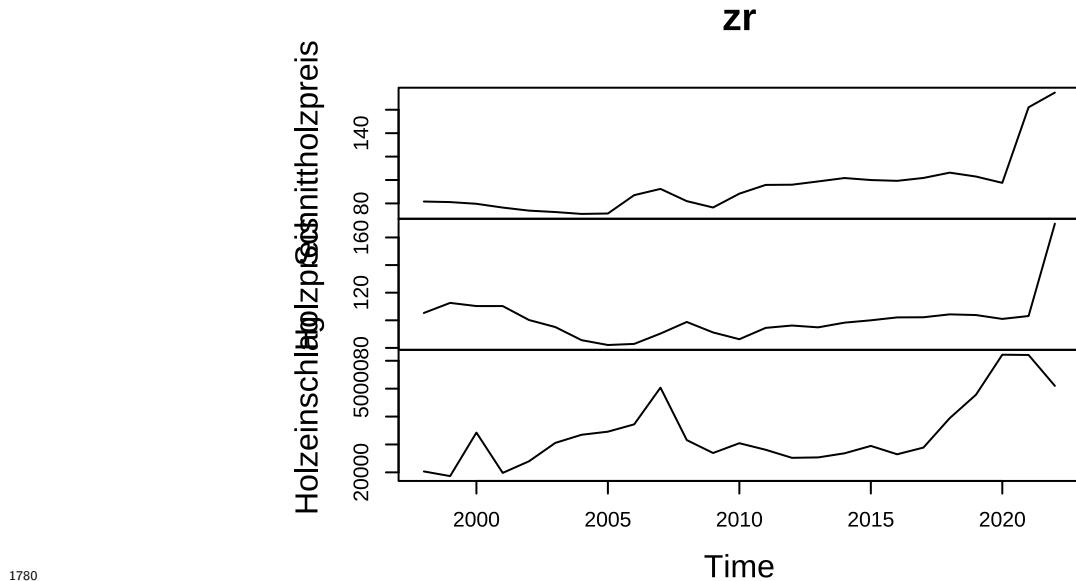
**typeof(zr)** # Zeitreihen sind sog. S3 Objekte. Das heißt, Zeitreihen gehören nicht zu den

1776 ## [1] "double"  
 # Basis Datentypen (siehe Kapitel Variablen, Funktionen und Datentypen),  
 # sondern sind eine Unterkategorie des Datentyps "Liste".

1777 Die wichtigsten Argumente sind - **data** Vektor oder Matrix, der nur die Daten enthält - **start** Startzeitpunkt -  
 1778 **frequency** Anzahl der Beobachtungen pro Zeiteinheit, also z. B. 12 bei monatlichen oder 4 bei quartalsweisen  
 1779 Erhebungen

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

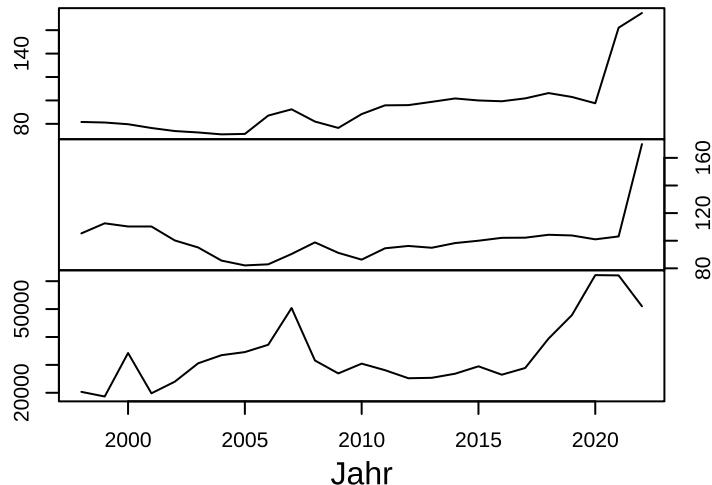
<sup>14</sup>Sie können sich die Daten auch selbst über die Website laden oder das Paket **wiesbaden** verwenden, um die Daten direkt in den R Workspace herunterzuladen zu laden. Jedoch müssen Sie sich zuerst registrieren



1781 Wir können low-level Layer hinzufügen. Die meisten base plot Funktionen funktionieren auch für Zeitreihen

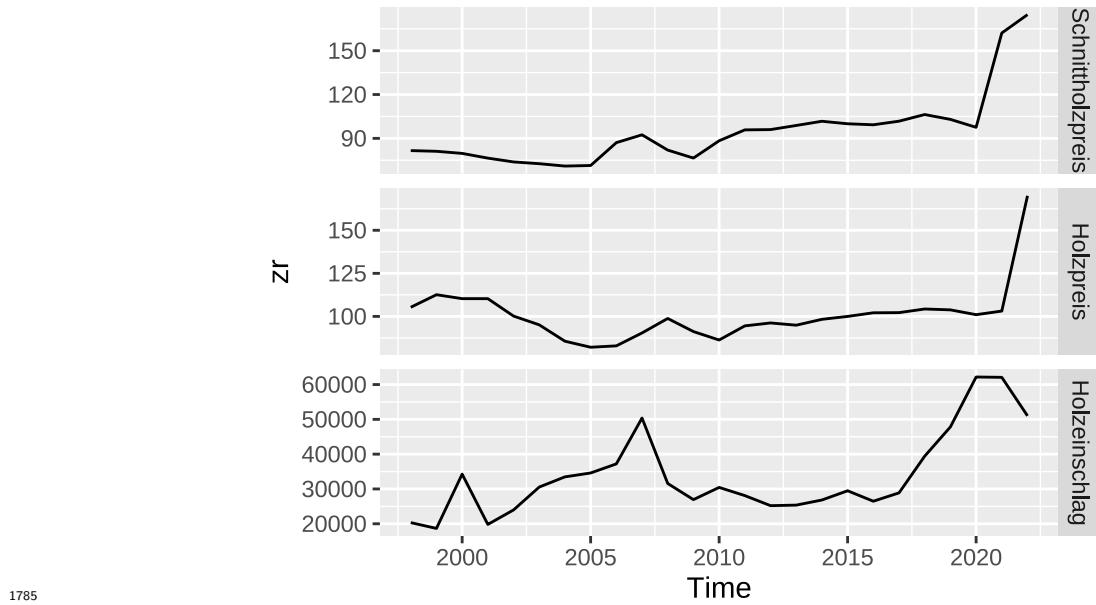
```
plot(zr, ann = FALSE, yax.flip = TRUE)
mtext(side = 3, cex = 1.5, font = 2, "Holzmarktentwicklung seit 1998", line = 2)
mtext(side = 1, cex = 1, line = 2.5, "Jahr")
```

## Holzmarktentwicklung seit 1998



1783 Beide Plot-Philosophiebn haben eine Zeitreihen-Funktion. Das Paket `ggfortify` ermöglicht automatisierte  
1784 Zeitreihenplots im `ggplot2` Stil. Damit ist auch das Problem der y-Achsenbeschriftungen gelöst.

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

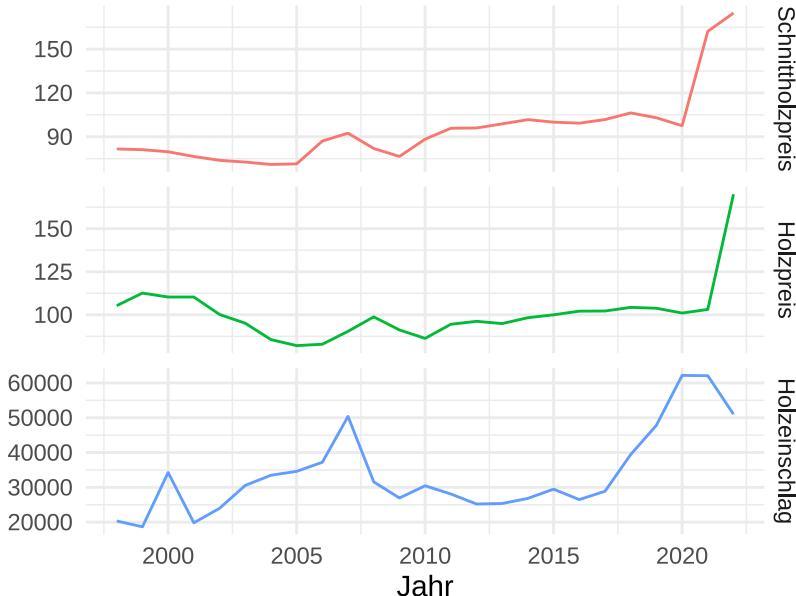


1785

1786 Wir können die Abbildung im `ggplot2` Stil ändern. Hier nur ein Minimalbeispiel zur Veranschaulichung.  
 1787 Siehe Kapitel 8.4 `ggplot2`: Eine Alternative für Abbildungen für mehr Möglichkeiten.

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

zr_autoplot + theme_minimal()
```

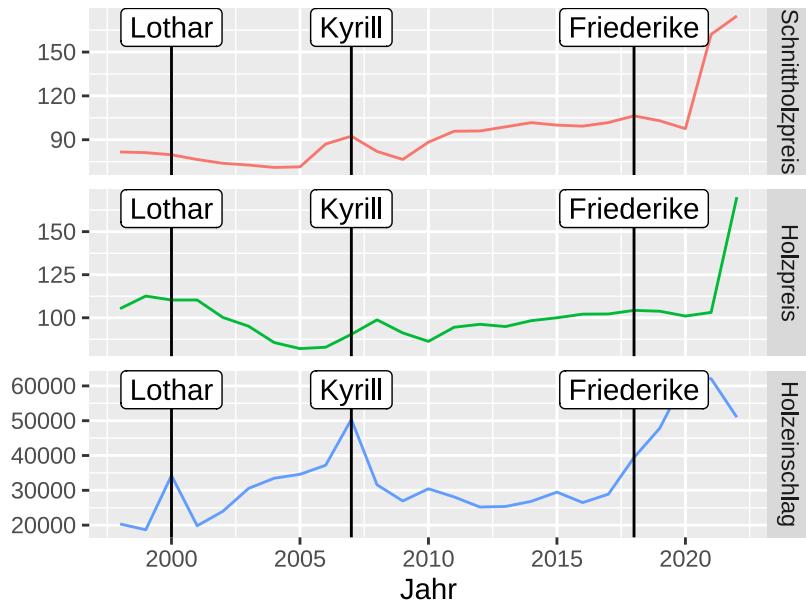


1788

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

z2 + annotate(x = 2000, y = +Inf, label = "Lothar", vjust = 1, geom = "label") +
```

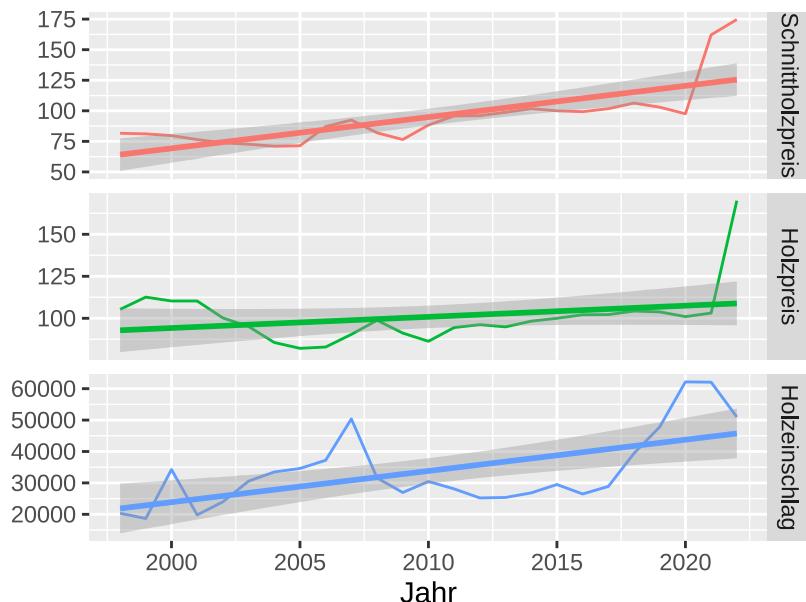
```
annotate(x = 2007, y = +Inf, label = "Kyrill", vjust = 1, geom = "label") +
 annotate(x = 2018, y = +Inf, label = "Friederike", vjust = 1, geom = "label")
```



1789

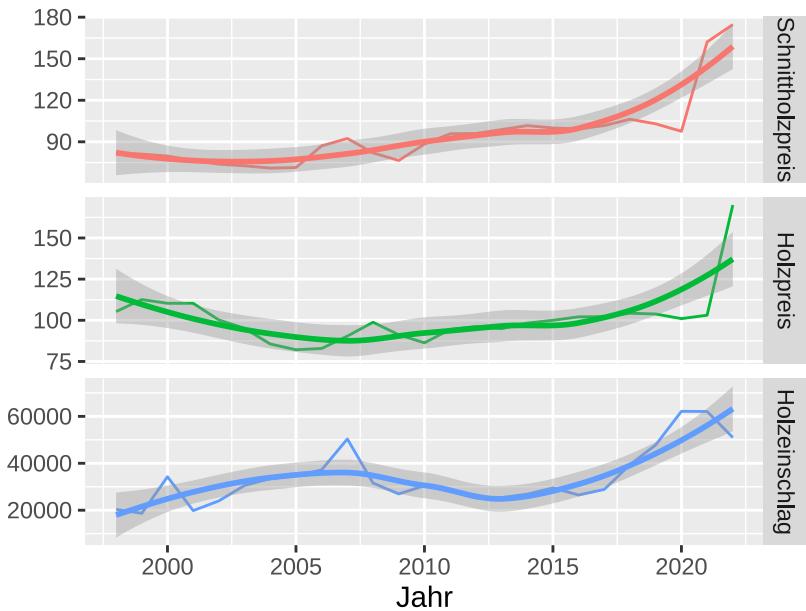
1790 Eine Trendlinie macht hier offensichtlich keinen Sinn. Die Trendlinie ist eine lineare Regression, also eine  
 1791 ordinäre Statistik, die wie eingangs erwähnt für Zeitreihen ungeeignet ist. Daher verwenden wir den sog.  
 1792 Loess-Glättter, um einen Trend in die Daten zu legen. Der Loess-Glättter ist ein sehr flexible Kurve. Wir sehen  
 1793 hier beispielsweise, dass der Leitholzpreis träge oder gar nicht auf das Angebot reagiert. Die Nachfrage jedoch  
 1794 zumindest in der einen Periode, in der sie stark steigt, den Holzpreis jedoch mit zeitlichem Verzug stark  
 1795 ansteigen lässt. Dieser visuelle Eindruck lässt sich durch spezifische Zeitreihen-Regressionen schätzen.

```
zr_autoplot + geom_smooth(method = "lm")
```



1796

```
zr_autoplot + geom_smooth(method = "loess") +
guides(colour = "none")
```



1797

## 1798 12 Aufgaben Wiederholen (for-Schleifen)

1799 Um einfache Programme zu schreiben, müssen Sie den Ablauf der Programmcodes kontrollieren können.  
 1800 Kontrollieren bedeutet in diesem Zusammenhang, dass Codeabschnitte nur unter definierten Bedingungen ab-  
 1801 laufen. Sie programmieren also zwei Sachen. 1) den Code selbst und 2), die Bedingungen die erfüllt sein müssen,  
 1802 damit der Code ausgeführt wird. Der Code muss do generisch geschrieben sein, dass er komplett durchläuft,  
 1803 auch wenn unterschiedliche Dateneingaben gemacht werden. Diese Kontrollbedingungen ermöglichen es Ihnen  
 1804 generisch zu programmieren. Sie schreiben Ihren Code also nicht speziell maßgeschneidert für ein Problem,  
 1805 sondern so generell, dass er für mehrere Auswertungen funktioniert. Um dies zu gewährleisten, müssen Sie  
 1806 bestimmte Situationen vorhersehen und abfangen. Hierbei helfen Ihnen Kontrollstrukturen (**Control Flow**).  
 1807 Grundsätzlich gibt es **Control Flow** Funktionen zur Wiederholung von Codeblöcken (Schleifen) und logische  
 1808 Bedingungen (bedingte Anweisung).

### 1809 12.1 Schleifen

1810 Bis jetzt wurden alle Skripte einfach der Reihe nach abgearbeitet und zwischendurch bestimmte Programmteile,  
 1811 je nach Situation, selbstständig ausgeführt oder übersprungen. Mit einer Schleife kann man erreichen, dass  
 1812 eine Gruppe von Befehlen (der sog. Schleifenrumpf) mehrfach abgearbeitet wird, zum Beispiel wenn bestimmte  
 1813 Auswertungsschritte auf mehrere Datensätze oder Variablen angewendet oder Funktionen mit unterschiedlichen  
 1814 Parametern oder Startwerten aufgerufen werden sollen. Weitere Anwendungsmöglichkeiten sind iterative  
 1815 Algorithmen, in denen die Eingabewerte des aktuellen Iterationsschrittes von einem vorherigen abhängig sind.  
 1816 Besonders in Simulationen kommen Schleifen häufig zum Einsatz, da große Anzahlen von Wiederholungen  
 1817 benötigt werden.

1818 Man unterscheidet zwischen zwei Arten von Schleifen: Bei den **for()**-Schleifen steht die Anzahl der Wieder-  
 1819 holungen schon beim Eintritt in die Schleife fest, während die **while()**-Schleifen so lange ausgeführt werden,  
 1820 bis eine Bedingung nicht mehr wahr ist. Mit der Funktion **break** wird eine Schleife abgebrochen und die  
 1821 Programmausführung wird nach der Schleife fortgesetzt.

1822 Die wesentlichen Befehle sind

1823 • **for (i in X) {Code}**

1824 Wiederhole den Code im “Schleifenrumpf” für jedes Element aus X.

1825 • **while(Bedingung) {Code}**

1826 Wiederhole den Code im “Schleifenrumpf” so lange die logische Bedingung erfüllt ist.

1827 • **break()**

1828 Brich die Schleife ab. **break()** muss im Schleifenrumpf an der richtigen Stelle ausgeführt werden. Gute  
 1829 Praxis ist jedoch, die for oder while Bedingungen, dass kein **break()**nötig ist, da **break()** anfällig für  
 1830 Programmierfehler ist.

#### 1831 12.1.1 Wiederholen von Befehlen mit **for()**.

1832 Steht vor Beginn der Schleife fest wie viele Schleifendurchgänge benötigt werden, wenn zum Beispiel in einer  
 1833 Simulationen 99 Realisierungen erzeugt oder alle Elemente eines Vektors verarbeitet werden sollen, verwendet

1834 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)
}
```

1835 ## [1] 1

1836 ## [1] 2

1837 ## [1] 3

1838 Das **i** steht in diesem Beispiel für die Schleifen-Variable. Sie muss nicht **i** heißen, sondern kann jeden  
 1839 zulässigen Namen annehmen. Das **X** steht für einen existierenden Vektor oder eine existierende Liste bzw.  
 1840 einen Ausdruck, der ein solches Objekt liefert (der Objektname ist ebenfalls frei wählbar). **for** und **in** sind  
 1841 Schlüsselworte, sie müssen, ebenso wie die runden Klammern, vorhanden sein.

1842 Im ersten Durchgang erhält die Schleifen-Variable **i** den ersten Wert von **X** und der Schleifenrumpf wird  
 1843 mit diesem Wert ausgeführt. Die Variable **i** nimmt nacheinander so lange die Werte von **X** an, bis ihr alle  
 1844 Elemente zugewiesen wurden.

1845 Das folgende Beispiel wird zwar besser durch die entsprechende Vektoroperation gelöst, zeigt aber sehr  
 1846 deutlich die Arbeitsweise der **for**-Schleife.

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

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

1847 ## [1] 4

1848 ## [1] 9

1849 ## [1] 25

1850

### 1851 Aufgabe 36: Schleifen 1

---

1853 Verwenden Sie den Vektor **k** <- c(1, 3, 9, 12, 15) und schreiben Sie folgende **for**-Schleifen:

- 1854 1. Eine Schleife, die jedes Element aus **k** ausgibt.
- 1855 2. Eine Schleife, die zu jedem Element aus **k** 10 addiert und den neuen Wert ausgibt.
- 1856 3. Eine Schleife wie in 2), aber der neue Wert (**k** + 10) soll jetzt nicht mehr ausgegeben werden, sondern  
 1857 in **k10** gespeichert werden. Stellen Sie sicher, dass **k10** wieder von der Länge 5 ist.

1858

---

1859 Die Funktion **for()** ermöglicht es, einen Befehl beliebig oft zu wiederholen. Z.B. der folgende Ausdruck zieht  
 1860 10-Mal eine Stichprobe der Größe 1 aus dem Vektor **v**. Beachten Sie, dass die Schleifen-Variable **i** selbst gar

1861 nicht im Schleifenrumpf vorkommt. Das Ziel dieser Schleife ist nicht die Elemente des Vektors abzuarbeiten,  
 1862 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))
}
```

```
1863 ## [1] 3
1864 ## [1] 1
1865 ## [1] 3
1866 ## [1] 3
1867 ## [1] 2
1868 ## [1] 3
1869 ## [1] 2
1870 ## [1] 2
1871 ## [1] 1
1872 ## [1] 4
```

1873 Auf gleiche Weise kann man auch über die Variablen eines Dataframes iterieren<sup>15</sup>. Das folgende Beispiel hat  
 1874 zum Ziel die Funktionsweise von Schleifen zu verdeutlichen. Schleifen haben in R jedoch den Nachteil, dass sie  
 1875 sehr langsam operieren. Wenn es geht, sollte man Alternativen verwenden. Die Funktionsweise wiederholender  
 1876 Auswertungen wird jedoch mit `for`-Schleifen deutlicher. Aus diesem Grunde werden wir uns in diesem Kurs  
 1877 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)
}

[1] "Buche"
[1] 52
[1] "Eiche"
[1] 64
[1] "Eiche"
[1] 62
[1] "Buche"
[1] 85
```

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

1886

---

1887 **Aufgabe 37: for-Schleife**

---

18881889 Lesen Sie den Datensatz `bhd_1.txt` ein und verwenden Sie eine `for`-Schleife.

- 1890 • Ziehen Sie 500-Mal je 35 Beobachtungen für den BHD.  
1891 • Berechnen Sie jeweils den Mittelwert aus den 35 Werten.  
1892 • Speichern Sie die 500 Mittelwerte in einem neuen Vektor `mittel`.  
1893 • Wie ist die Verteilung dieser 500 Mittelwerte zu interpretieren?

1894 **12.1.2 Wiederholen von Befehlen mit `while()`**

1895 Die `while`-Schleifen finden Anwendung, wenn die Anzahl der zu durchlaufenden Wiederholungen vorher  
1896 nicht bekannt ist, wie zum Beispiel bei Iterationsverfahren, die bis zu einer gewissen Genauigkeit durchlaufen  
1897 werden sollen. Die `while`-Schleife besteht in R aus dem Schlüsselwort `while()` und einer Bedingung in runden  
1898 Klammern.

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

1899 Sie ist in der praktischen Programmierung nicht so relevant wie die `for`-Schleife. Sie sei deshalb hier nur  
1900 kurz erwähnt. Die Abbruchbedingung wird jedes Mal geprüft bevor der Schleifenrumpf durchlaufen wird. Die  
1901 Bedingung wird ausgewertet und wenn diese `TRUE` ist, wird der Schleifenrumpf ausgeführt und danach erneut  
1902 die Bedingung überprüft. Ist die Bedingung nicht erfüllt, wird der Schleifenrumpf nicht durchgeführt und die  
1903 Schleife beendet. Ist die Bedingung bereits vor Eintritt in die Schleife nicht erfüllt, wird die Schleife gar nicht  
1904 erst durchlaufen.

1905 Da `while`-Schleifen also so lange ausgeführt werden, bis die Bedingung nicht mehr erfüllt ist, kann eine  
1906 Endlosschleife entstehen. Dies kann passieren, wenn man nicht sauber programmiert hat. Wenn innerhalb der  
1907 Schleife nicht dafür gesorgt wird, dass die Bedingung irgendwann nicht mehr erfüllt wird, so läuft die Schleife  
1908 immer weiter. Steckt R in einer Schleife fest und reagiert nicht mehr, kann der Befehl unter Linux mit `Strg`+  
1909 `C` und unter Windows mit `Esc` abgebrochen werden. Alternativ können Sie auf das rote STOP Symbol  
1910 über der Konsole klicken.

1911 **12.2 Bedingte Ausführung von Codeblöcken**

1912 Innerhalb eines Skripts ist es mitunter notwendig je nach aktueller Situation unterschiedlich fortzufahren.  
1913 Die Situation wird mit einem logischen Ausdruck, einer sogenannten Bedingung, geprüft. Je nachdem, ob  
1914 die Bedingung wahr (`TRUE`) oder falsch (`FALSE`) ist, werden unterschiedliche Programmteile ausgeführt, der  
1915 jeweils andere Teil bleibt unberücksichtigt. Danach wird in jedem Fall die Programmausführung, mit den  
1916 auf die bedingte Anweisungen folgenden Anweisung, fortgesetzt. In R kann dies mit dem if-else-Konstrukt  
1917 realisiert werden, welches aus den Schlüsselwörtern `if()` und `else` sowie der in runde Klammern gefassten  
1918 Bedingung besteht.

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

1919 Im folgenden Beispiel sollen die bisher abstrakt beschriebenen Vorgänge praktische Anwendung finden. In  
 1920 dem Beispiel wird zunächst durch zufälliges Ziehen einer Zahl aus dem Bereich eins bis sechs ein Würfelwurf  
 1921 simuliert. Anschließend wird der Würfelwurf mit einem if-Ausdruck kommentiert. Ist die Bedingung, es wurde  
 1922 eine Sechs gewürfelt, erfüllt, wird der Code innerhalb der geschweiften Klammern ausgeführt, ansonsten wird  
 1923 der Klammerinhalt ignoriert.

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

1924 In den meisten Fällen ist es für R irrelevant, ob sich zwischen den verschiedenen Elementen Leerzeichen oder  
 1925 Zeilenumbrüche befinden. Bei dem else-Ausdruck dagegen wird ein Fehler erzeugt, wenn vor dem `else` nicht  
 1926 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.")
}
```

1927 `## [1] "Beim nächsten Wurf klappt's bestimmt."`

1928

### 1929 Aufgabe 38: Bedingte Programmierung

---

- 1931 • Wenn keine 6 gewürfelt wurde, lassen Sie zusätzlich ausgeben welche Zahl gewürfelt wurde.  
 1932 • Wiederholen Sie den Würfelwurf 10 Mal.

## 1933 13 (R)markdown

### 1934 13.1 Markdown Grundlagen

1935 Die Idee von Markdown ist, einfach Text strukturieren zu können und das ganze ohne umfangreiche Programme  
 1936 zu erstellen. Im nächsten Abschnitt sehen wir dann, wie man Markdown und R-Code verbinden kann. Hier  
 1937 soll es jedoch erst einmal darum gehen, die einfachsten Bausteine von Markdown vorzustellen.

1938 Am Anfang jedes Dokuments kommt eine Präambel. Diese fängt mit --- an und hört auch wieder mit ---  
 1939 auf. Innerhalb der Präambel können dann Metainformationen über das Dokument festgelegt werden, dies  
 1940 beinhaltet im einfachsten Fall: Titel, Autor und Datum. Das würde dann so aussehen:

```
1941 ---
1942 title: "Ein Titel"
1943 author: "Der, der es geschrieben hat"
1944 date: "März 2021"
1945 ---
```

1946 Danach folgt strukturierter und formatierter Text. Verschiedene Hierarchieebenen von Überschriften können  
 1947 mit der Anzahl an # festgelegt werden. So ist eine Überschrift erster Ordnung # Kapitel eine Überschrift  
 1948 zweiter Ordnung ## Unterkapitel usw.

1949 Listen können erstellt werden, wenn man am Anfang jeder Zeile ein - oder 1. schreibt.

```
1950 - Erster Eintrag
1951 - Zweiter Eintrag
1952 - Dritter Eintrag
```

1953 wird zu

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

1957 Eine zentrale Idee von Markdown ist es Text einfach zu formatieren. Werden eine oder mehrere Wörter mit  
 1958 zwei Sternchen (\*\*) eingefasst wird dieser Text **fett** dargestellt. Also aus \*\*wichtig\*\* wird **wichtig**. Das  
 1959 gleiche funktioniert auch mit *kursiven* Text, jedoch muss man hier noch ein Sternchen verwenden, also aus  
 1960 \*kursiv\* wird *kursiv*. Soll ein text fett und kursiv sein, kann man drei Sternchen verwenden. Aus \*\*\*sehr  
 1961 wichtig\*\*\* wird dann **sehr wichtig**.

1962 Weitere Elemente wie Links oder Abbildungen können einfach eingebunden werden. Links werden mit [Link  
 1963 text](url) in den Text integriert. Beispielsweise ist eine gute Idee bei [stackoverflow](#) bei Problemen nach  
 1964 einer Lösung zu suchen. Dieser Link wurde mit [stackoverflow](www.stackoverflow.com) erstellt.

1965 Für Abbildungen gibt es einen ganz ähnlichen Syntax. Mit ! [Das R Logo](abb/r\_logo.png) wird die  
 1966 Abbildung r\_logo.png eingebunden mit der Beschriftung: "Das R Logo".



Abbildung 12: Das R Logo

1967

---

**Aufgabe 39: Arbeiten mit markdown**


---

1970 Verwenden Sie das folgende Markdowndokument:

1971 ---

```
1972 title: "Dokument"
1973 author: "Ihr Name"
1974 date: "März 2021"
```

1975 ---

1976

1977 # Einleitung

1978

1979 # Methoden

- 1980 1. Kopieren Sie die Vorlage in ein Dokument, das `test.md` heißt.
- 1981 2. Fügen Sie zwei Überschriften zweiter und dritter Ordnung hinzu.
- 1982 3. Fügen Sie einen *kursiven* Text hinzu.
- 1983 4. Fügen Sie einen Link zu einer Website hinzu.
- 1984 5. Kompilieren Sie die Datei, indem Sie in Rstudio auf **Preview** drücken (Abbildung 13).

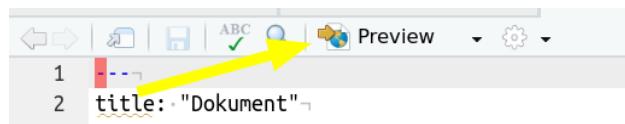


Abbildung 13: Kompilieren einer md-Datei.

**13.2 R und Markdown**

1986 Markdown macht es bereits einfach Textdokumente und Dokumentationen zu verfassen, aber die wirkliche  
 1987 Stärke liegt in der Möglichkeit R und Markdown zu kombinieren. Man spricht dann von Rmarkdown. Ein  
 1988 weiteres Strukturelement, das wir noch nicht kennen gelernt haben, sind Code-Blöcke.

1989 ~~~

1990 a &lt;- 1:10

```

1991 a[1]
1992 ``
1993 erzeugt
1994 a <- 1:10
1995 a[1]

1996 Momentan wird noch kein Code ausgeführt, sondern lediglich der Code, als Code dargestellt. Rmarkdown
1997 bietet nun die Möglichkeit Code beim kompilieren16 auszuführen. Dafür müssen wir nur einen Code-Block als
1998 R-Code-Block kennzeichnen.

1999 ``{R}
2000 a <- 1:10
2001 a[1]
2002 ```

2003 erzeugt
2004 a <- 1:10
2005 a[1]

2006 ## [1] 1

2007 Beachte, die Variable a wird beim kompilieren erzeugt und steht dann R zur Verfügung. R-Code-Blöcke
2008 werden auch als Code Chunks bezeichnet. Diese Chunks können sehr genau angesprochen und angepasst
2009 werden. Einige wichtige Argumente sind:
2010

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

```

2011

## 2012 Aufgabe 40: Arbeiten mit Rmarkdown

---

2013 Erstellen Sie eine neue Rmarkdown Datei mit dem Namen **test1.Rmd**. Erstellen Sie zwei Code-Chunks. Der  
2014 erste soll nicht angezeigt werden und darin werden die Daten geladen (**bhd\_1.txt**). Im zweiten Chunk plotten  
2015 Sie das Alter der Bäume gegen den BHD. Was passiert mit dem Plot, wenn Sie die Datei kompilieren (drücken  
2016 Sie dazu auf den Knit-Knopf; Abbildung 14).



Abbildung 14: Kompilieren einer Rmd-Datei.

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

## 2018 14 Räumliche Daten in R

### 2019 14.1 Was sind räumliche Daten

2020 Räumliche Daten sind Beobachtungen, wie wir sie schon oft gesehen haben, mit einem räumlichen Bezug. Der  
 2021 Unterschied zu nicht räumlichen Daten liegt darin, dass räumliche Daten eindeutig im *Raum* verortet werden  
 2022 können. Häufig werden sogenannte Geoinformationssysteme (GIS) zum Arbeiten mit räumlichen verwendet.  
 2023 R kann in vielerlei Hinsicht wie ein GIS eingesetzt werden und hier werden einige Grundfunktionalitäten  
 2024 dafür besprochen. Räumliche Daten werden in zwei unterschiedliche Datentypen unterteilt: Vektor- und  
 2025 Rasterdaten. Vektordaten modellieren einzelne Objekte (= *Features*). Rasterdaten modellieren eine Oberfläche.  
 2026 Vektordaten bestehen aus zwei Komponenten: 1) einer Geometrie, die die Form und Lage der Daten definiert  
 2027 und 2) Attributen, den tatsächlichen Daten. Räumliche Daten werden oft als *Features* bezeichnet. Ein Feature  
 2028 ist die räumliche Einheit einer Beobachtung. Je nach Art der räumlichen Daten können Features entweder  
 2029 Punkte (z.B. ein Baum), Linien (z.B. eine Straße) oder Polygone (z.B. ein See) sein. Auch können mehrere  
 2030 Geometrien zu einem Feature zusammengefasst werden. Ein Beispiel wäre eine Beobachtung für ein Land,  
 2031 das aber aus mehreren Polygonen bestehen kann (z.B. Festland und Inseln). Features können dann weitere  
 2032 Attribute (= Attributdaten) haben, z.B. eine ID, Name oder was auch immer man gemessen hat.  
 2033 Rasterdaten bestehen aus einer Oberfläche von gleichgroßen Kacheln (= *Pixel*), die ein Gebiet abdecken.  
 2034 Meist sind Pixel viereckig, aber das ist keine Voraussetzung. Dabei hat jedes Pixel einen Wert (das kann  
 2035 auch ein fehlender Wert sein). Typische Beispiele für Rasterdaten sind Landnutzung oder Seehöhen.  
 2036 In R kann sowohl mit Vektor- als auch mit Rasterdaten gearbeitet werden. Für Vektordaten bietet sich das  
 2037 Paket **sf** an und für Rasterdaten das Paket **raster**.

### 2038 14.2 Koordinatenbezugssystem

2039 Eine der Herausforderungen für räumliche Daten ist die eindeutige Verortung im Raum. Dazu braucht man  
 2040 ein *Koordinatenbezugssystem (KBS)*. Einem KBS liegt ein mathematisches Modell der Erde zugrunde. Die  
 2041 Details zu KBS werden schnell relativ kompliziert und wir beschränken uns hier lediglich darauf, wie KBS  
 2042 verwendet werden können. Dazu müssen wir zwei Fälle unterscheiden: 1) einem Datensatz ein KBS zuweisen  
 2043 und 2) Transformation des KBS eines Datensatzes in ein anderes KBS. Die technischen Details werden in  
 2044 den folgend Abschnitten besprochen. Für beide Aufgaben müssen wir auf ein KBS verweisen können, ein  
 2045 einfacher Ansatz dafür sind die sogenannten *EPSG-Codes*<sup>17</sup>.

### 2046 14.3 Vektordaten in R

2047 Das Paket **sf** stellt Klassen zum Abbilden von Features zur Verfügung, die dann in einem **data.frame** als  
 2048 Liste gespeichert werden können. In der Regel erstellen wir Features nicht individuell, sondern lesen diese aus  
 2049 externen Datenquellen (z.B. ESRI Shapefile) ein. Zum besseren Verständnis, erstellen wir es einmal manuell.  
 2050 Wir haben die Koordinaten für drei Städte (Göttingen, Hannover und Berlin) als geografische Koordinaten  
 2051 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))
```

2052 Daraus können wir jetzt eine Geometriespalte für einen `data.frame` erstellen

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

2053 Somit haben wir die Geometrie in der Variable `geom` gespeichert, aber noch keine dazugehörigen Attributdaten.

2054 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)
)
```

2055 In einem letzten Schritt möchten wir jetzt die Geometrie (`geom`) und die Attributdaten (`attr`) zusammenführen.

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

```
2057 ## Simple feature collection with 3 features and 3 fields
2058 ## Geometry type: POINT
2059 ## Dimension: XY
2060 ## Bounding box: xmin: 9.732 ymin: 51.5413 xmax: 13.405 ymax: 52.52
2061 ## Geodetic CRS: WGS 84
2062 ## name bundesland einwohner geom
2063 ## 1 Goettingen Niedersachsen 119000 POINT (9.9158 51.5413)
2064 ## 2 Hannover Niedersachsen 532000 POINT (9.732 52.3759)
2065 ## 3 Berlin Berlin 3650000 POINT (13.405 52.52)
```

2066 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.

2068 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)
)
```

2071 Dann kann man mit der Funktion `st_as_sf()` weiter arbeiten:

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

## 2072 14.4 Arbeiten mit Vektordaten

2073 Es gibt sehr viele Funktionen, um mit räumlichen Daten zu arbeiten, von denen wir hier einige vorstellen.

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

```
2074 ## Coordinate Reference System:
2075 ## User input: EPSG:4326
2076 ## wkt:
2077 ## GEOGCRS["WGS 84",
2078 ## ENSEMBLE["World Geodetic System 1984 ensemble",
2079 ## MEMBER["World Geodetic System 1984 (Transit)"],
2080 ## MEMBER["World Geodetic System 1984 (G730)"],
2081 ## MEMBER["World Geodetic System 1984 (G873)"],
2082 ## MEMBER["World Geodetic System 1984 (G1150)"],
2083 ## MEMBER["World Geodetic System 1984 (G1674)"],
2084 ## MEMBER["World Geodetic System 1984 (G1762)"],
2085 ## MEMBER["World Geodetic System 1984 (G2139)"],
2086 ## ELLIPSOID["WGS 84",6378137,298.257223563,
2087 ## LENGTHUNIT["metre",1]],
2088 ## ENSEMBLEACCURACY[2.0]],
2089 ## PRIMEM["Greenwich",0,
2090 ## ANGLEUNIT["degree",0.0174532925199433]],
2091 ## CS[ellipsoidal,2],
2092 ## AXIS["geodetic latitude (Lat)",north,
2093 ## ORDER[1],
2094 ## ANGLEUNIT["degree",0.0174532925199433]],
2095 ## AXIS["geodetic longitude (Lon)",east,
2096 ## ORDER[2],
2097 ## ANGLEUNIT["degree",0.0174532925199433]],
2098 ## USAGE[
2099 ## SCOPE["Horizontal component of 3D system."],
2100 ## AREA["World."],
2101 ## BBOX[-90,-180,90,180]],
2102 ## ID["EPSG",4326]]
```

2103 Wenn wir jetzt zu einem anderen KBS (z.B. EPSG:3035, ein europäisches projiziertes KBS) umrechnen  
 2104 möchten, können wir das mit

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

```

2105 ## Coordinate Reference System:
2106 ## User input: EPSG:3035
2107 ## wkt:
2108 ## PROJCRS["ETRS89-extended / LAEA Europe",
2109 ## BASEGEOGCRS["ETRS89",
2110 ## ENSEMBLE["European Terrestrial Reference System 1989 ensemble",
2111 ## MEMBER["European Terrestrial Reference Frame 1989"],
2112 ## MEMBER["European Terrestrial Reference Frame 1990"],
2113 ## MEMBER["European Terrestrial Reference Frame 1991"],
2114 ## MEMBER["European Terrestrial Reference Frame 1992"],
2115 ## MEMBER["European Terrestrial Reference Frame 1993"],
2116 ## MEMBER["European Terrestrial Reference Frame 1994"],
2117 ## MEMBER["European Terrestrial Reference Frame 1996"],
2118 ## MEMBER["European Terrestrial Reference Frame 1997"],
2119 ## MEMBER["European Terrestrial Reference Frame 2000"],
2120 ## MEMBER["European Terrestrial Reference Frame 2005"],
2121 ## MEMBER["European Terrestrial Reference Frame 2014"],
2122 ## ELLIPSOID["GRS 1980",6378137,298.257222101,
2123 ## LENGTHUNIT["metre",1]],
2124 ## ENSEMBLEACCURACY[0.1]],
2125 ## PRIMEM["Greenwich",0,
2126 ## ANGLEUNIT["degree",0.0174532925199433]],
2127 ## ID["EPSG",4258]],
2128 ## CONVERSION["Europe Equal Area 2001",
2129 ## METHOD["Lambert Azimuthal Equal Area",
2130 ## ID["EPSG",9820]],
2131 ## PARAMETER["Latitude of natural origin",52,
2132 ## ANGLEUNIT["degree",0.0174532925199433],
2133 ## ID["EPSG",8801]],
2134 ## PARAMETER["Longitude of natural origin",10,
2135 ## ANGLEUNIT["degree",0.0174532925199433],
2136 ## ID["EPSG",8802]],
2137 ## PARAMETER["False easting",4321000,
2138 ## LENGTHUNIT["metre",1],
2139 ## ID["EPSG",8806]],
2140 ## PARAMETER["False northing",3210000,
2141 ## LENGTHUNIT["metre",1],
2142 ## ID["EPSG",8807]]],
2143 ## CS[Cartesian,2],
2144 ## AXIS["northing (Y)",north,
2145 ## ORDER[1],
2146 ## LENGTHUNIT["metre",1]],
2147 ## AXIS["easting (X)",east,

```

```

2148 ## ORDER[2] ,
2149 ## LENGTHUNIT["metre",1]],
2150 ## USAGE [
2151 ## SCOPE["Statistical analysis."],
2152 ## AREA["Europe - European Union (EU) countries and candidates. Europe - onshore and offshore: "],
2153 ## BBOX[24.6,-35.58,84.73,44.83]],
2154 ## ID["EPSG",3035]

2155 Die Funktion st_buffer() erlaubt es Features zu puffern, mit st_distance() kann die Distanz zwischen
2156 Features berechnet werden, mit st_area() kann die Fläche eines Features berechnet werden.

2157 Funktionen wie st_intersection(), st_union() und st_difference() erlauben es geometrische Opera-
2158 tionen zwischen unterschiedlichen Features zu berechnen. Für eine ausführliche Diskussion siehe auch hier:
2159 https://geocompr.robinlovelace.net/geometric-operations.html.

2160 Normalerweise lesen wir Daten von externen Datenquellen (z.B. ESRI Shapefiles). Das geht mit der Funktion
2161 st_read().

```

## 2162 14.5 Rasterdaten in R

```

2163 Für Rasterdaten gibt es das R-Paket terra. Auch hier wollen wir uns wieder auf einige Grundfunktionalitäten
2164 konzentrieren. Diese umfassen das Einlesen, Zuschneiden, Rechnen und Abfragen von Rastern.

2165 Mit der Funktion rast() kann ein Raster in R eingelesen werden.

```

```

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

```

```

2166 dem steht für Digital Elevation Model und ist ein Raster mit den Seehöhen in Niedersachsen mit einer
2167 500-m-Auflösung. Wir können diese mit der Funktion res()18 abfragen.

```

```

res(dem)

```

```

2168 ## [1] 500 500

```

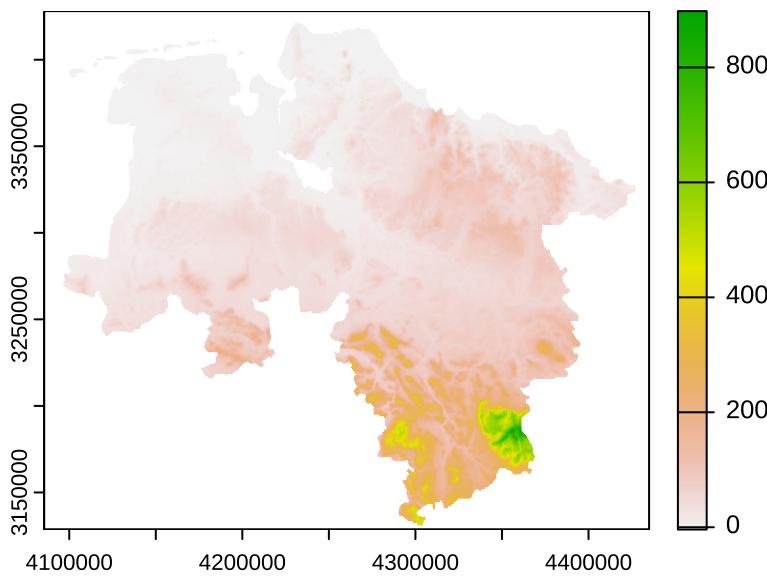
2169 Bzw. wir können den Raster auch plotten.

```

plot(dem)

```

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



2170  
2171 Wenn wir den Raster `dem` auf ein Gebiet zuschneiden wollen (z.B. Göttingen), müssen wir drei Schritte  
2172 durchführen. Als erstes müssen wir ein Shapefile für Göttingen einlesen.

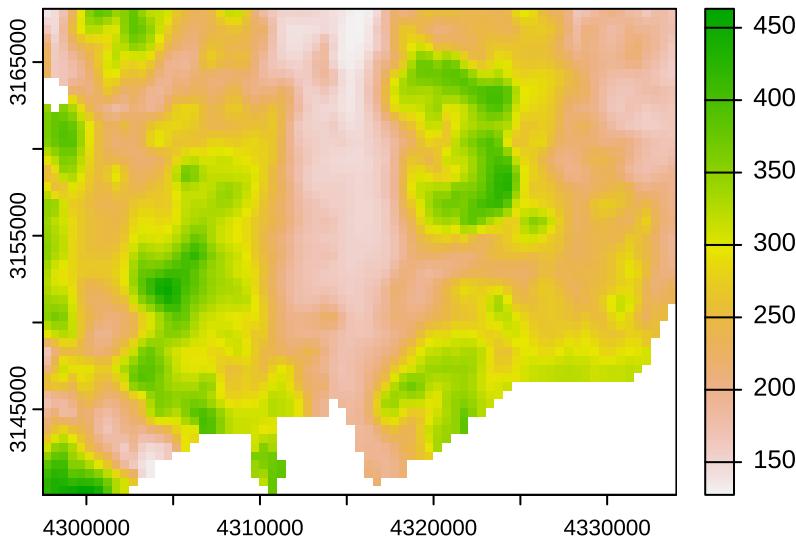
```
goe <- st_read(here::here("data/goettingen/stadt_goettingen.shp"))
```

2173 Dann müssen wir sicherstellen, dass sowohl der Raster `dem` als auch das `sf`-Objekt `goe` im selben KBS sind.  
2174 Es bietet es sich in der Regel an, das KBS des Vektors zu transformieren. Mit der Funktion `projectRaster()`  
2175 kann das KBS eines Rasters transformiert werden.

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

2176 Mit der Funktion `crop()` kann der Raster `dem` auf Göttingen zugeschnitten werden.

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



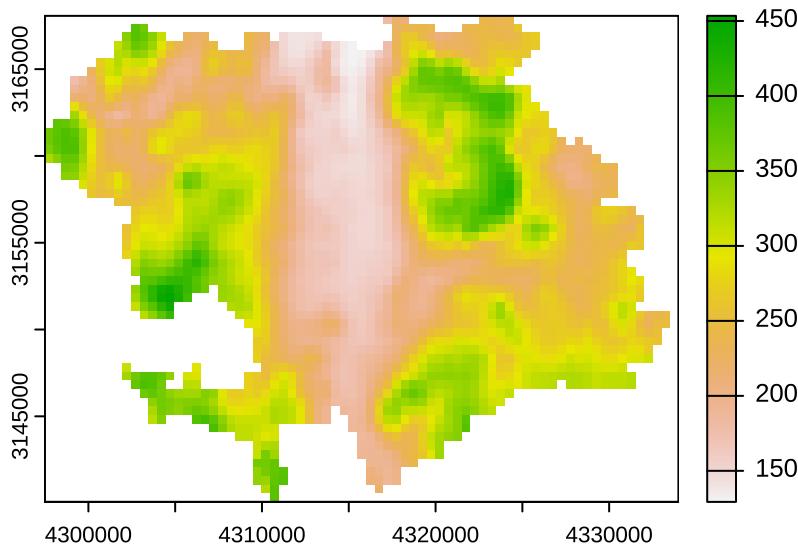
2177  
2178 Der Raster hat jetzt die Größe einer Bounding-Box (BBX) von Göttingen (das ist ein Rechteck, das Göttingen  
2179 umfasst). Mit der Funktion `mask()` kann der Raster auf die genauen Grenzen des Vektors `goe` angepasst

2180 werden.

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

2181 ## Warning: [mask] CRS do not match

```
plot(dem2)
```



2182

2183 Wenn wir an bestimmten Punkten den Wert des Rasters abfragen wollen (z.B. an `cities`) von vorhin, dann  
2184 gibt es dafür die Funktion `extract`. Dann müssen wir erst sicherstellen, dass `staedte` und `dem` gleichen KBS  
2185 zu grunde liegt. Dafür transformieren wir einfach `staedte` in das KBS von `dem`. Mit der Funktion `crs()`  
2186 erhalten wir das KBS des Rasters.

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

2187 Wenn wir das KBS eines Objektes nicht kennen, können wir auch einfach das KBS übergeben. Der folgende  
2188 Code-Block macht genau das Gleiche mit dem Vorteil, dass wir keinen EPSG-Code angeben müssen.

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

2189 Dann können wir für jede Stadt die Seehöhe abfragen:

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

2190 ## ID dem\_3035

2191 ## 1 1 149.18181

2192 ## 2 2 57.21486

2193 ## 3 3 NA

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

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

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

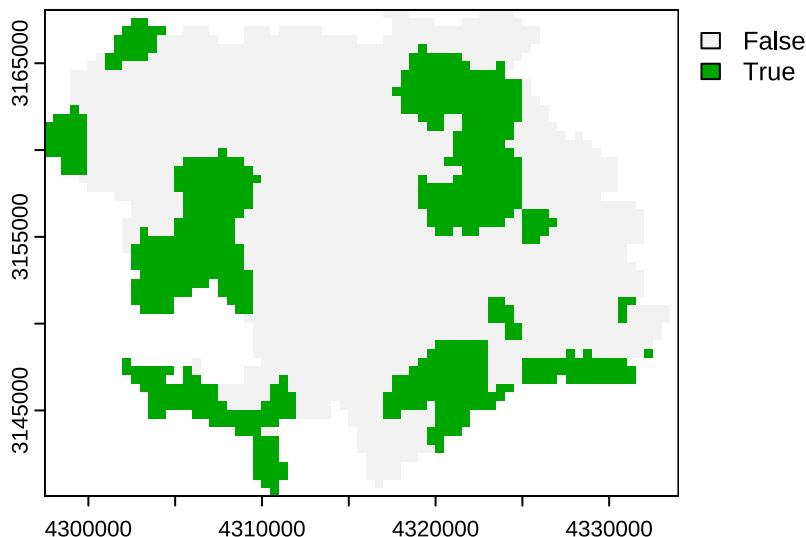
2199 berechnen:

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

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

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

```
plot(dem3)
```



2202

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

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

```
2204 ## dem_3035
2205 ## [1,] NA
2206 ## [2,] NA
2207 ## [3,] NA
2208 ## [4,] NA
2209 ## [5,] NA
2210 ## [6,] NA
```

2211 Das sind erst einmal viele NA-Werte für die ganzen Zellen, die außerhalb von Niedersachsen liegen. Aber wir  
2212 können mit so einem Vektor ganz normal arbeiten und z.B. die Fläche des Landes Niedersachsen die eine  
2213 Seehöhe von mehr als 500m Seehöhe hat ausrechnen.

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

```
2214 ## [1] 0.2786229
```

2215

---

**2216 Aufgabe 41: Arbeiten mit Rastern**

---

2218 Verwenden Sie den Raster `wald.tif`, der auf einer 10 m Auflösung den Waldanteil jeder Rasterzelle angibt<sup>19</sup>.  
 2219 Der EPSG-Code für das KBS von `wald.tif` ist 3035. Nehmen Sie an, dass wenn der Waldanteil in einer  
 2220 Raster größer als 50 % ist, dass die Rasterzelle als Wald klassifiziert werden kann. Wie viel Prozent des  
 2221 Göttinger Stadtgebietes sind Wald? Wie ändert sich dieser Wert, wenn sie 70 % anstatt 50 % als Schwellenwert  
 2222 für Wald annehmen?

2223

---

**2224 Aufgabe 42: Studiendesign**

---

2226 Mit der Funktion `st_sample()` können Sie innerhalb oder entlang eines Features zufällige Punkte legen. Das  
 2227 Argument `n` steuert die Anzahl Punkte und das Argument `type` wie die Punkte angeordnet werden. Für `type`  
 2228 sind für uns die Werte `type = "random"` (komplett zufällig), `type = "regular"` (regelmäßiger Grid) und  
 2229 `type = "hexagonal"` von Bedeutung (ein hexagonaler Grid, d.h. ein sechseckiger Raster). Unglücklicherweise  
 2230 ist das Ergebnis von `st_sample()` erst eine Geometrie. Um daraus ein vollständiges `sf`-Objekt zu machen  
 2231 und problemlos weiter arbeiten zu können, müssen Sie nocheinmal die Funktion `st_as_sf()` ausführen.  
 2232 Stellen Sie sich vor, dass wir die tatsächliche Waldbedeckung des Göttinger Stadgebietes **nicht** kennen und wir  
 2233 eine Studie durchführen, um den Anteil des Göttinger Stadgebietes, der mit Wald bedeckt ist herauszufinden.  
 2234 Erstellen Sie dafür einige unterschiedliche Stichproben (diese können in der Anzahl und Anordnung variieren).  
 2235 Berechnen Sie für jedes Stichprobendesign den Anteil an Wald und ein dazugehöriges Konfidenzintervall  
 2236 (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  $n$   
 2237 die Stichprobengröße). Nehmen Sie an, dass eine Rasterzelle Wald ist, sobald > 50 % der Rasterzelle mit  
 2238 Wald bedeckt ist.

2239

---

**2240 Aufgabe 43: Räumliche Daten**

---

2242 Verwenden Sie den folgenden Datensatz:

```
set.seed(123)
df1 <- data.frame(
 x = runif(100, 0, 100),
 y = runif(100, 0, 100),
 kronendurchmesser = runif(100, 1, 15),
 art = sample(letters[1:4], 100, TRUE)
)
```

2243 1. Erstellen Sie ein `sf`-Objekt aus `df1`.

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

- 2244 2. Puffern Sie jeden Baum mit seinem Kronendurchmesser.
- 2245 3. Berechnen Sie die Kronenfläche jedes Baumes. *Hinweis: Die Funktion st\_area() könnte dafür hilfreich sein.*
- 2246 4. Welcher Baum hat die größte Kronenfläche?
- 2247 5. Finden Sie für jede Art, den Baum mit der größten Kronenfläche.

2249

2250 **Aufgabe 44: Arbeiten mit räumlichen Daten**

---

- 2252 1. Lesen Sie das ESRI Shapefile goettingen/stadt\_goettingen.shp ein.
- 2253 2. Wie viele Features befinden sich in dem Shapefile?
- 2254 3. Welches Koordinatenbezugssystem (KBS) hat das Shapefile?
- 2255 4. Transformieren Sie das Shapefile in das KBS 3035.
- 2256 5. Erstellen Sie eine neue Spalte A in der Sie die Fläche jeder Gemeinde/Stadt speichern.
- 2257 6. Welche Gemeinde/Stadt (Spalte GEN) ist am größten?
- 2258 7. Wählen Sie nun nur die Stadt Göttingen aus.

2259

2260 **Aufgabe 45: Arbeiten mit räumlichen Daten 2**

---

- 2262 1. Lesen Sie erneut das ESRI Shapefile goettingen/stadt\_goettingen.shp ein.
- 2263 2. Lösen sie die Gemeindegrenzen auf (die Funktion st\_union() könnte hier nützlich sein).
- 2264 3. Wie groß ist das resultierende Feature?

2265 **15 FAQs (Oft gefragtes)**

2266 **15.1 Arbeiten mit Daten**

2267 **15.1.1 Einlesen von Exceldateien**

- 2268 Mit der Funktion `read_excel()` aus dem Paket `readxl` können Exceldateien direkt in R eingelesen werden.  
2269 Ein Export als csv-Datei aus Excel ist nicht notwendig.

## 2270 16 Literatur

- 2271 Ein guter Überblick über viele der angeschnittenen Themen gibt es in dem [R for data science](#), das online frei  
2272 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.  
2273
- 2274 McNamara, Amelia, and Nicholas J Horton. 2018. “Wrangling Categorical Data in r.” *The American Statistician*  
2275 72 (1): 97–104.
- 2276 Wickham, Hadley. 2014. “Tidy Data.” *Journal of Statistical Software* 59 (10). <https://doi.org/10.18637/jss.v059.i10>.  
2277