

1

---

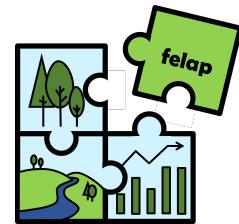
# Einführung in die Datenanalyse mit R (700104)

4

Kursskript

5

---



6

Dr. Johannes SIGNER  
Abteilung Wildtierwissenschaften  
Büsgenweg 3  
37077 Göttingen

[jsigner@uni-goettingen.de](mailto:jsigner@uni-goettingen.de)

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

[kai.husmann@uni-goettingen.de](mailto:kai.husmann@uni-goettingen.de)

7



8

Fakultät für Forstwissenschaften und Waldökologie  
Georg-August-Universität Göttingen

10



11

---

12

Wintersemester 2023/2024

---

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

<sup>15</sup> Zitiervorschlag:

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

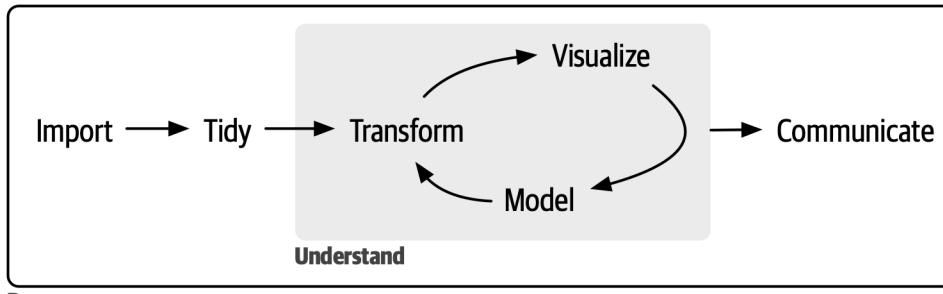
<sup>18</sup> Letzte Aktualisierung: 7. Februar 2024

---

## 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	<b>1 R und RStudio</b>	<b>3</b>
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	<b>2 Variablen, Funktionen und Datentypen</b>	<b>7</b>
50	2.1 Variablen beim Programmieren . . . . .	7
51	2.2 Funktionen . . . . .	8
52	2.3 Datentypen . . . . .	9
53	2.4 Datenstrukturen . . . . .	10
54	<b>3 Vektoren</b>	<b>12</b>
55	3.1 Funktionen zum Arbeiten mit Vektoren . . . . .	14
56	3.2 Statistische Funktionen . . . . .	15
57	3.3 Beispiel Fotofallen . . . . .	16
58	3.4 Arbeiten mit logischen Werten . . . . .	17
59	3.5 Zugreifen auf Elemente eines Vektors (=Untermengen) . . . . .	18
60	3.6 Der %in%-Operator . . . . .	20
61	<b>4 Faktoren (factors)</b>	<b>22</b>
62	4.1 Das Paket <b>forcats</b> . . . . .	23
63	4.1.1 Anpassen der Anordnung von Faktoren . . . . .	24
64	<b>5 Spezielle Einträge</b>	<b>25</b>
65	5.1 NA . . . . .	25
66	5.2 NULL . . . . .	26
67	5.3 Inf . . . . .	26
68	<b>6 data.frames oder Tabellen</b>	<b>28</b>
69	6.1 Wichtige Funktionen zum Arbeiten mit <b>data.frames</b> . . . . .	29
70	6.2 Zugreifen auf Elemente eines <b>data.frame</b> . . . . .	30
71	<b>7 Schreiben und lesen von Daten</b>	<b>33</b>
72	7.1 Textdateien . . . . .	33
73	<b>8 Erstellen von Abbildungen</b>	<b>35</b>
74	8.1 Base Plot . . . . .	35
75	8.1.1 Mehrere Panels . . . . .	41
76	8.1.2 Speichern von Abbildungen . . . . .	41
77	8.2 Histogramme . . . . .	42
78	8.3 Boxplots . . . . .	45
79	8.4 <b>ggplot2</b> : Eine Alternative für Abbildungen . . . . .	47
80	8.4.1 Multipanel Abbildungen . . . . .	54

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

# 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 Entwicklungsumgebung<sup>1</sup>, die das Arbeiten mit R vereinfachen soll. Sie können also mit R arbeiten ohne RStudio (was unüblich ist), aber nicht mit RStudio ohne R. Gehen Sie für die Installation von R, auf die Website <https://cloud.r-project.org/> und laden Sie die für ihren Computer passende R-Version herunter und folgen Sie den Installationsanweisungen. In Linux können Sie R über die Kommandozeile installieren. Für die Installation von RStudio gehen Sie zu der Website <https://posit.co/download/rstudio-desktop/#download> und laden die richtige Version für Ihr Betriebssystem herunter und folgen Sie den Installationsanweisungen.

## 1.2 Erste Schritte in R

RStudio bietet eine Vielzahl von Funktionen, die uns das Arbeiten mit R erleichtern können. Öffnen Sie RStudio. Sie erhalten eine leere Entwicklungsumgebung. Als erstes bietet es sich an, ein neues Skript zu erstellen. Gehen Sie dafür auf das Menü: **[File] > [New File] > R Script** oder klicken Sie die Tastenkombination *Strg + Umschalt + N* (**[Strg] + [Umschalt] + [N]**).

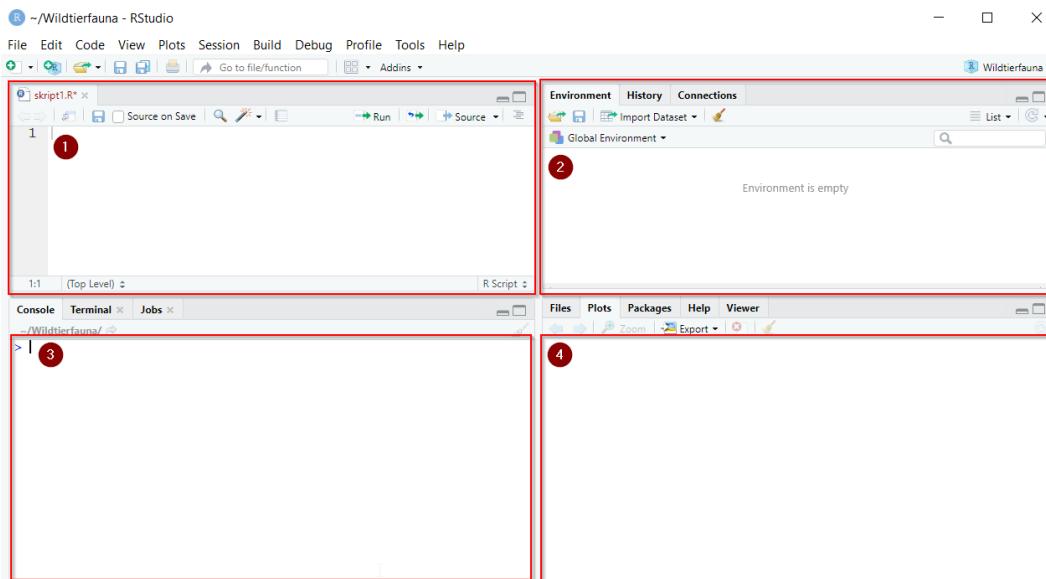


Abbildung 1: RStudio Panes.

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

1. Hier werden Skripte angezeigt, d.h., hier wird meist R Code geschrieben und dokumentiert. Der Code wird

<sup>1</sup>Oder auch IDE (=Integrated Development Environment) genannt.

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

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

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

**10 + 5**

151 **## [1] 15**

**20 - 10**

152 **## [1] 10**

**10 \* 3**

153 **## [1] 30**

**100 / 19**

154 **## [1] 5.263158**

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

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

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

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

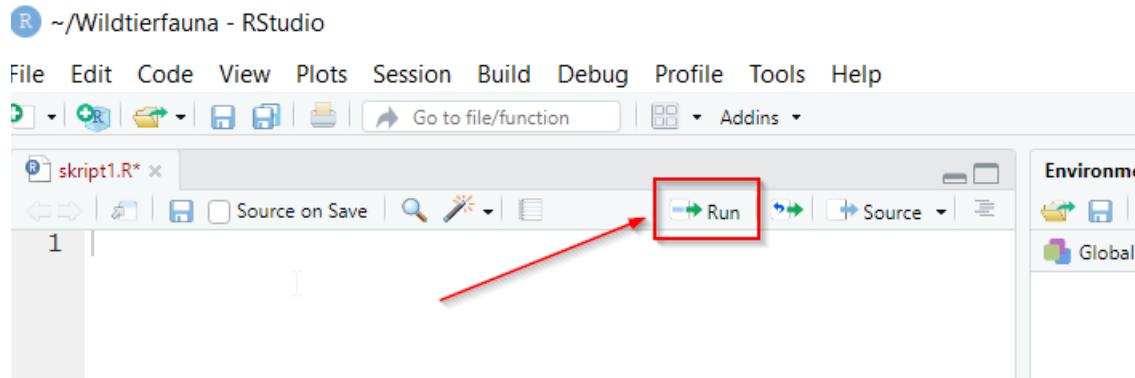


Abbildung 2: Zeilenweises Ausführen von Code in RStudio.

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

### 1.3 Gute Praxis bei der Programmierung

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

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

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

193 ## [1] 9

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

200

201 **Aufgabe 1: Ausführen von Quellcodes**

---

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

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

# Einfache Funktion
sqrt(20)
```

205 Führen Sie nun alle Zeilen aus.

## 2 Variablen, Funktionen und Datentypen

### 2.1 Variablen beim Programmieren

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

```
alter <- 102
```

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

```
a <- 10
a
```

`## [1] 10`

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

Wir können beliebige Variablen erstellen, z.B.

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

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

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

Vorsicht: Groß- und Kleinschreibung muss beachtet werden.

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

`## [1] "Johannes"`

Das Aufrufen der Variable

```
Name
```

führt zu einem Fehler.

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

```
a <- 10
b <- 5

a + b
```

```

229 ## [1] 15
b / a

230 ## [1] 0.5
a^b

231 ## [1] 1e+05

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

233 ## [1] 15
ergebnis2 <- ergebnis * 2
ergebnis2

234 ## [1] 30

235 Mit der Funktion rm() können Variablen, die nicht mehr benötigte Variablen, wieder gelöscht werden. Alternativ können Variablen auch überschrieben werden. Es gibt keine Möglichkeit gelöschte oder überschriebene
236 Variablen wiederherzustellen. Sie müssen ggf. neu berechnet werden.
var1 <- "irgendwas"
exists("var1") # TRUE. also ja, eine Variable mit diesem Namen existiert

238 ## [1] TRUE
rm(var1)
exists("var1") # FALSE, also nein, eine Variable mit diesem Namen existiert nicht.

239 ## [1] FALSE

240 2.2 Funktionen

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

243 ## [1] 3.162278

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

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

```

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

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

### 266 2.3 Datentypen

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

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

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

274 In den zwei vorherigen Zeilen Code haben wir zwei Objekte (im Sinne von R) erstellt. Das erste Objekt heißt  
 275 `kamera_name` und das zweite Objekt heißt `anzahl_rehe`. In dem Beispiel handelt es sich also um zwei sehr  
 276 einfache Objekte, in denen jeweils ein Wert gespeichert ist. Auffällig ist, dass beide Objekte unterschiedliche  
 277 Datentypen haben. `kamera_name` ist vom Typ `character` (also Text). Das zweite Objekt, `anzahl_rehe`, ist  
 278 vom Typ `numeric` (also eine Zahl, wir unterscheiden hier nicht weiter<sup>2</sup>). Zusätzlich zu diesen zwei Typen  
 279 (`character` und `numeric`), gibt es noch einen weiteren wichtigen Typ: nämlich das logische Wahr oder Falsch  
 280 (in R: `TRUE` und `FALSE`) und noch weitere Datentypen auf die wir zunächst nicht eingehen (tippen Sie `?typeof`  
 281 für eine Übersicht aller Datentypen). Zurückkommend auf das Beispiel mit den Fotofallen, könnte eine  
 282 mögliche Fragestellung sein, ob auf einem der Fotos ein Fuchs gesehen wurde oder nicht. Dazu würden wir  
 283 eine neue Variable `fuchs_gesehen` anlegen und diese auf `TRUE` setzen, da ein Fuchs gesehen wurde.

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

```
fuchs_gesehen <- TRUE
```

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

```
typeof(fuchs_gesehen)
```

285 ## [1] "logical"

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

```
TRUE + TRUE
```

287 ## [1] 2

```
FALSE + FALSE
```

288 ## [1] 0

```
TRUE + FALSE
```

289 ## [1] 1

## 2.4 Datenstrukturen

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

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

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

```
# 1. Kamera
name1 <- "Kamera_1"
anzahl_rehe1 <- 132
revier_1 <- "Revier A"

# 2. Kamera
name2 <- "Kamera_2"
anzahl_rehe2 <- 79
revier_2 <- "Revier A"

# usw.
```

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

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

302 Frames) für diesen Zweck kennenlernen.

303

304 **Aufgabe 2: Variablen**

---

305 306 Verwenden Sie die folgenden Daten

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

307 und berechnen sie:

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

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

### 3 Vektoren

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

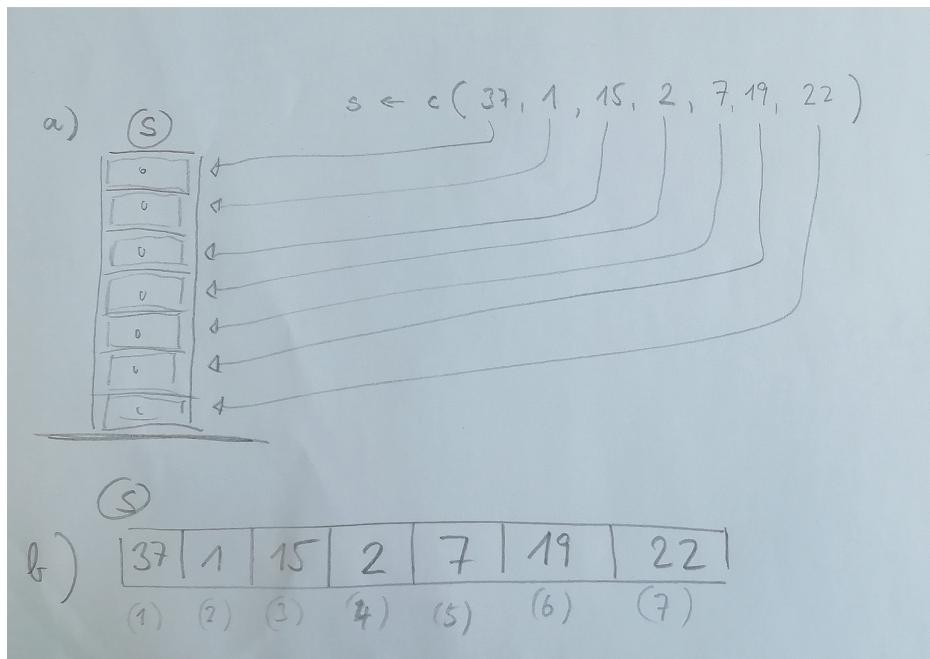


Abbildung 3: Schematische Darstellung eines Vektors in R.

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

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

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

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

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

s

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

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

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

s + 10

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

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

s \* s

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

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

seq(from = 1, to = 10)

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

(1 : 10)

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

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

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

350 ## [1] 1 3 5 7 9

351

### 352 Aufgabe 3: Vektoren erstellen

---

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

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

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

### 358 3.1 Funktionen zum Arbeiten mit Vektoren

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

```
head(s)
```

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

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

362 `## [1] 37 1 15`

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

363 `## [1] 19 22`

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

```
length(s)
```

365 `## [1] 7`

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

```
class(s)
```

367 `## [1] "numeric"`

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

```
unique(s)
```

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

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

```
table(s)
```

371 `## s`

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

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

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

```
rev(s)
```

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

377 während `sort()` einen Vektor nach seinen Elementen sortiert<sup>5</sup>.

```
sort(s)
```

378 `## [1] 1 2 7 15 19 22 37`

---

<sup>5</sup>Auch für `sort()` gibt es ein zusätzliches Argument, das es ermöglicht die Elemente in absteigender Reihenfolge zu sortieren. Schauen Sie sich dazu, und auch für weitere Funktionen, die Hilfeseiten an.

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

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

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

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

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

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

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

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

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

385

---

#### 386 Aufgabe 4: Arbeiten mit Vektoren

---

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

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

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

## 3.2 Statistische Funktionen

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

```
mean(s)
```

395 ## [1] 14.71429

```
median(s)
```

396 ## [1] 15

```
sd(s)
```

397 ## [1] 12.76341

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

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

401 ## [1] 1

```

sample(s, size = 3) # 2 Elemente

402 ## [1] 15 7 22
403 Wenn size weggelassen wird, dann bekommt man gleich viele Elemente zurück (wie der Vektor lang ist), d.h.
404 der Vektor wird nur permutiert.

```

### 405 3.3 Beispiel Fotofallen

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

409 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)

```

410 Als zweites erstellen wir einen Vektor mit den IDs. Zur Erinnerung, diese sollten die Werte `Kamera_1` bis  
411 `Kamera_15` haben. Ein erster Ansatz könnte sein, dass wir einfach 15 Fotofallen schreiben und dann die  
412 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"
)

```

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

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

```

v1 <- rep("Kamera", 15)

```

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

```

v2 <- 1:15

```

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

```

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

423 ## [1] "Kamera_1"   "Kamera_2"   "Kamera_3"   "Kamera_4"   "Kamera_5"   "Kamera_6"
424 ## [7] "Kamera_7"   "Kamera_8"   "Kamera_9"   "Kamera_10"  "Kamera_11"  "Kamera_12"
425 ## [13] "Kamera_13"  "Kamera_14"  "Kamera_15"

```

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

```
428 rep(c("Revier A", "Revier B", "Revier C"), 5)
429 ## [1] "Revier A" "Revier B" "Revier C" "Revier A" "Revier B" "Revier C"
430 ## [7] "Revier A" "Revier B" "Revier C" "Revier A" "Revier B" "Revier C"
431 ## [13] "Revier A" "Revier B" "Revier C"
```

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

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

```
reviere
```

```
433 ## [1] "Revier A" "Revier A" "Revier A" "Revier A" "Revier A" "Revier B"
434 ## [7] "Revier B" "Revier B" "Revier B" "Revier B" "Revier C" "Revier C"
435 ## [13] "Revier C" "Revier C" "Revier C"
```

436

### Aufgabe 5: Statistische Funktionen

---

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

440 2. Erstellen Sie die folgende Konsolenausgabe:

```
441 ## [1] "Die mittlere Anzahl von Rehfotos beträgt 171.8 Rehe pro Standort."
```

## 442 3.4 Arbeiten mit logischen Werten

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

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

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

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

```
anzahl_rehe > 100
```

```
452 ## [1] TRUE FALSE TRUE FALSE TRUE TRUE FALSE TRUE TRUE FALSE TRUE
453 ## [13] FALSE TRUE TRUE
```

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

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

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

```
456 ## [1] FALSE FALSE FALSE FALSE FALSE TRUE TRUE TRUE TRUE TRUE FALSE FALSE
457 ## [13] FALSE FALSE FALSE
```

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

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

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

```
464 ## [1] FALSE FALSE FALSE FALSE FALSE TRUE FALSE TRUE TRUE TRUE FALSE FALSE
465 ## [13] FALSE FALSE FALSE
```

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

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

```
469 ## [1] TRUE FALSE TRUE FALSE TRUE TRUE TRUE TRUE TRUE TRUE FALSE TRUE
470 ## [13] FALSE TRUE TRUE
```

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

473

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

---

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

```
477 1. TRUE | FALSE
478 2. FALSE & TRUE
479 3. (FALSE & TRUE) | TRUE
480 4. (2 != 3) | FALSE
481 5. FALSE + 10
482 6. TRUE + 10
483 7. TRUE + 10 == FALSE + 10
484 8. sum(c(TRUE, TRUE, FALSE, FALSE))
```

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

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

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

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

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

```
anzahl_rehe[2]
```

499 ## [1] 79

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

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

501 ## [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.

502 ## [1] 132 79 129 91 138

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

505

### 506 Aufgabe 7: Zugreifen auf Vektorelemente

---

508 Erstellen Sie einen neuen Vektor `bhd`

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

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

512

---

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

515 ## [1] 132 79 129 91 138

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

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

519 ## [1] 132 79 129 91 138

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

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

522 ## [1] 132 79 129 91 138

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

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

525 ## [1] 113.8

526

### 527 Aufgabe 8: logische Werte

---

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

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

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

## 534 3.6 Der %in%-Operator

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

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

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

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

540 `## [1] "FI" "FI"`

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

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

543 `## [1] "FI" "BU" "BU" "FI"`

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

```
messungen_arten %in% arten
```

547 `## [1] TRUE TRUE TRUE FALSE FALSE FALSE TRUE FALSE FALSE`

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

548 `## [1] "FI" "BU" "BU" "FI"`

549

---

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

---

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

```
LETTERS
```

553 `## [1] "A" "B" "C" "D" "E" "F" "G" "H" "I" "J" "K" "L" "M" "N" "O" "P" "Q" "R" "S"`

554 `## [20] "T" "U" "V" "W" "X" "Y" "Z"`

555 Wählen Sie aus LETTERS nur die Vokale aus.

## 556 4 Faktoren (factors)

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

563 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)
```

564 ## [1] FI BU FI EI EI FI FI
 565 ## Levels: BU EI FI

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

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

569 ## [1] FI BU FI EI EI FI FI
 570 ## Levels: FI BU EI

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

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

573 ## [1] Fichte Buche Fichte Eiche Eiche Fichte Fichte
 574 ## Levels: Fichte Buche Eiche

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

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

578 ## [1] Fi Bu Fi Ei Ei Fi Fi
 579 ## Levels: Fi Bu Ei

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

```
af
```

```
582 ## [1] Fi Bu Fi Ei Ei Fi Fi
583 ## Levels: Fi Bu Ei
relevel(af, "Bu")
```

```
584 ## [1] Fi Bu Fi Ei Ei Fi Fi
585 ## Levels: Bu Fi Ei
```

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

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

```
588 ## [1] "Fi" "Bu" "Fi" "Ei" "Ei" "Fi" "Fi"
```

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

```
af
```

```
590 ## [1] Fi Bu Fi Ei Ei Fi Fi
591 ## Levels: Fi Bu Ei
```

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

```
592 ## [1] 1 2 1 3 3 1 1
```

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

595

### 596 Aufgabe 10: Faktoren

---

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

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

## 600 4.1 Das Paket **forcats**

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

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

## 606 4.1.1 Anpassen der Anordnung von Faktoren

607 Wir verwenden nochmals den `a` Vektor mit unterschiedlichen Baumarten:

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

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

```
factor(a)
```

609 ## [1] FI BU FI EI EI FI FI

610 ## Levels: BU EI FI

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

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

```
f1
```

612 ## [1] FI BU FI EI EI FI FI

613 ## Levels: FI BU EI

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

```
fct_rev(f1)
```

615 ## [1] FI BU FI EI EI FI FI

616 ## Levels: EI BU FI

617 nach der Häufigkeit zu sortieren oder

```
fct_infreq(f1)
```

618 ## [1] FI BU FI EI EI FI FI

619 ## Levels: FI EI BU

620 eine zufällige Sortierung.

```
fct_shuffle(f1)
```

621 ## [1] FI BU FI EI EI FI FI

622 ## Levels: EI FI BU

## 5 Spezielle Einträge

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

### 630 5.1 NA

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

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

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

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

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

```
is.na(na1)

## [1] FALSE TRUE FALSE
```

```
na.omit(na1)

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

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

```
na2 < 3

## [1] FALSE FALSE     NA
```

**1 + NA**

649 ## [1] NA  
650 Viele R Funktionen haben eingebaute Methoden zum Umgang mit NA. Die Funktion `mean()` bspw. ergibt  
651 (wie die meisten Funktionen) standardmäßig NA wenn sie auf Vektoren mit Datenlücken angewendet wird, es  
652 sei denn man stellt innerhalb der Funktion ein, dass Datenlücken entfernt werden sollen.

`mean(na2)`

653 ## [1] NA  
654 `mean(na2, na.rm = TRUE)`  
654 ## [1] 4.5

**655 5.2 NULL**

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

**660 5.3 Inf**

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

`10^309`

663 ## [1] Inf  
664 ## [1] Inf  
665 ## [1] Inf  
666 ## [1] Inf  
667 ## [1] -Inf

668 ## [1] 0  
669 Infinity kann mit `is.infinite` und `is.finite` getestet werden. Relationäre Operatoren funktionieren  
670 erwartungsgemäß.

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

671 ## [1] TRUE FALSE FALSE TRUE FALSE
is.finite(inf1)

672 ## [1] FALSE TRUE TRUE FALSE TRUE
inf1 < 3

673 ## [1] FALSE TRUE FALSE TRUE FALSE
```

674

---

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

---

677 Verwenden Sie den Vektor

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

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

680 Verwenden Sie den Vektor

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

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

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

686 Berechnen Sie den arithmetischen Mittelwert von `foo`.

## 6 data.frames oder Tabellen

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

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

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

721 Standard-Objekte zum Speichern wissenschaftlicher Daten.

## 722 6.1 Wichtige Funktionen zum Arbeiten mit `data.frames`

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

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

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

728 Oder für die letzten 2 Beobachtungen.

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

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

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

```
nrow(monitoring)
```

```
733 ## [1] 15
```

```
ncol(monitoring)
```

```
734 ## [1] 3
```

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

```
str(monitoring)
```

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

741

### 742 Aufgabe 12: `data.frame`

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

## 6.2 Zugreifen auf Elemente eines `data.frame`

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

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

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

756 `## [1] 91`

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

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

760 `## [1] 91`

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

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

763 `## [1] 132 79 129 91 138`

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

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

766 `## anzahl_rehe revier`

767 `## 1 132 Revier A`

768 `## 2 79 Revier A`

769 `## 3 129 Revier A`

770 `## 4 91 Revier A`

771 `## 5 138 Revier A`

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

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

774 `## ID anzahl_rehe revier`

775 `## 1 Kamera_1 132 Revier A`

776 `## 2 Kamera_2 79 Revier A`

777 `## 3 Kamera_3 129 Revier A`

```
778 ## 4 Kamera_4          91 Revier A
779 ## 5 Kamera_5          138 Revier A
```

780

---

**781 Aufgabe 13: Abfragen von Werten**

---

783 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)
)
```

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

787

---

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

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

```
monitoring$anzahl_rehe
```

793 ## [1] 132 79 129 91 138 144 55 103 139 105 96 146 95 118 107

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

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

795 ## [1] 132 79 129 91 138 144 55 103 139 105 96 146 95 118 107

- 796 3. Alle Zeilen und die Spalte explizit angeben.

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

797 ## [1] 132 79 129 91 138 144 55 103 139 105 96 146 95 118 107

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

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

```
monitoring$anzahl_rehe > 100
```

```
805 ## [1] TRUE FALSE TRUE FALSE TRUE TRUE FALSE TRUE TRUE TRUE FALSE TRUE  
806 ## [13] FALSE TRUE TRUE
```

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

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

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

822

---

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

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

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

## 829 7 Schreiben und lesen von Daten

### 830 7.1 Textdateien

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

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

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

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

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

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

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

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

865

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

---

- 867  
868 Lesen Sie die Datei `kompliziert.txt` ein. Schauen Sie die Hilfeseite an und vergewissern Sie sich, dass Sie  
869 wissen was die Argumente `header`, `sep`, `dec` und `skip` bewirken. Setzten die Argumente richtig, damit die  
870 Datei `kompliziert.txt` folgendes Ergebnis liefert.

## 8 Erstellen von Abbildungen

871 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  
872 einen Plot zu erstellen. In diesem Kurs werden wir kurz *Base Plots* vorstellen und dann das Zusatzpaket  
873 **ggplot2** vorstellen.

### 8.1 Base Plot

874 Die wichtigsten Grafiken für die einfache Datendarstellung sind schnell verfügbar. Etwas komplexere oder  
875 spezielle Grafiken erfordern mehr Programmieraufwand (folgt teilweise noch). Für viele komplexere Diagramme  
876 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  
877 sie sich die einfache Grafik Schnittstelle (**base plots**) als zweidimensionale Leinwand vor, auf die Sie durch  
878 Code Ebene für Ebene Grafikelemente legen:

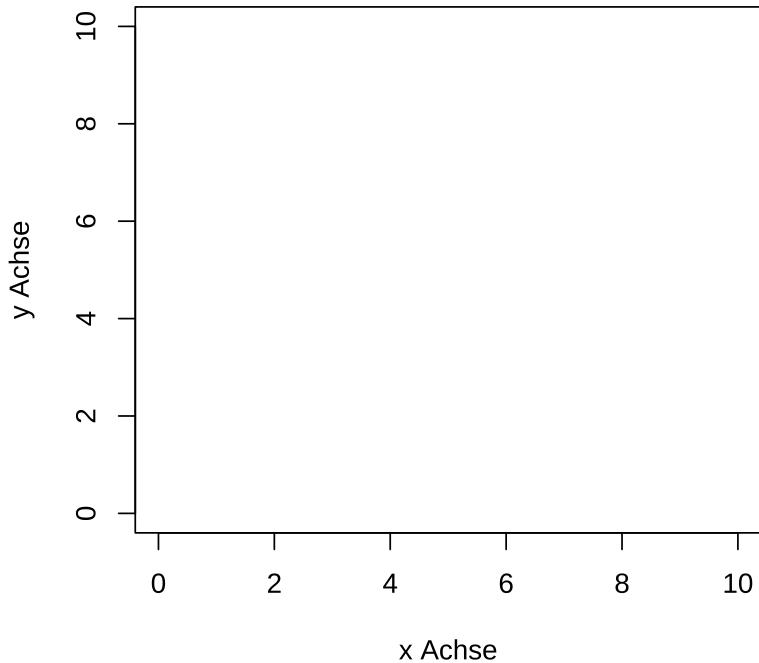
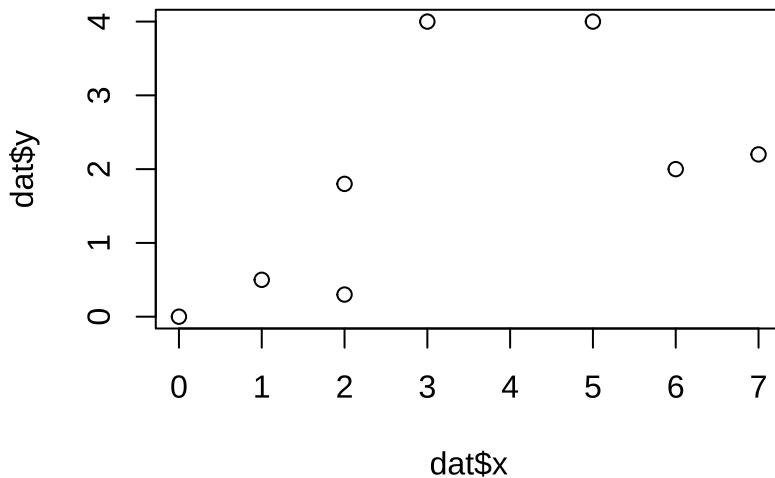


Abbildung 4: Beispiel einer leeren Grafikschnittstelle.

883 Hier drei einfache Beispiele für Abbildungen mit nur einer Ebene.

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

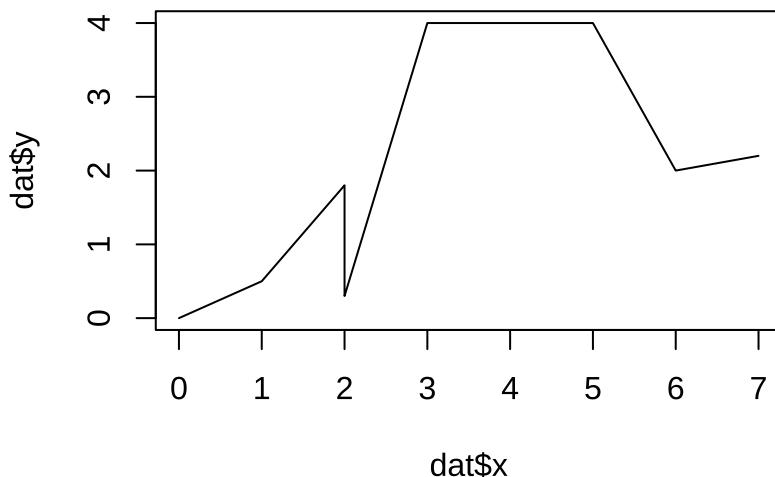
plot(dat$x, dat$y, type ="p")
```



884

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

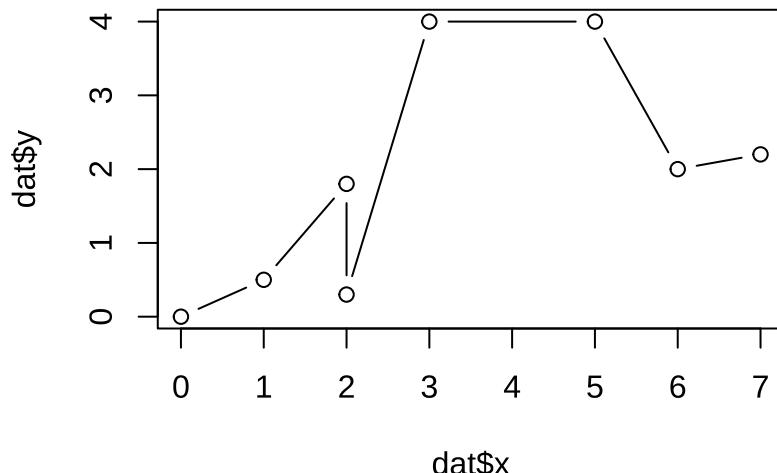
```
plot(dat$x, dat$y, type = "l")
```



887

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

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



889

890 darstellen.

891

892 **Aufgabe 16: Base Plot 1**

893

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

896

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

- 901 • `type` - Diagrammtyp
- 902 • `col` - Farbe
- 903 • `main` - Titel
- 904 • `sub` - Untertitel
- 905 • `pch` - Punktsymbol
- 906 • `lty` - Linientyp
- 907 • `lwd` - Linienstärke
- 908 • `xlab` bzw. `ylab` - Achsenbeschriftungen
- 909 • `xlim`, `ylim` - Grenzen der Achsenanschnitte
- 910 • `axes` - Sollen die Achsen eingezeichnet werden? Oder leer gelassen werden, um sie nachträglich als  
911 low-level Ebene einzuziehen?
- 912 • `ann` - Achsenbeschriftung kann ganz weggelassen werden.

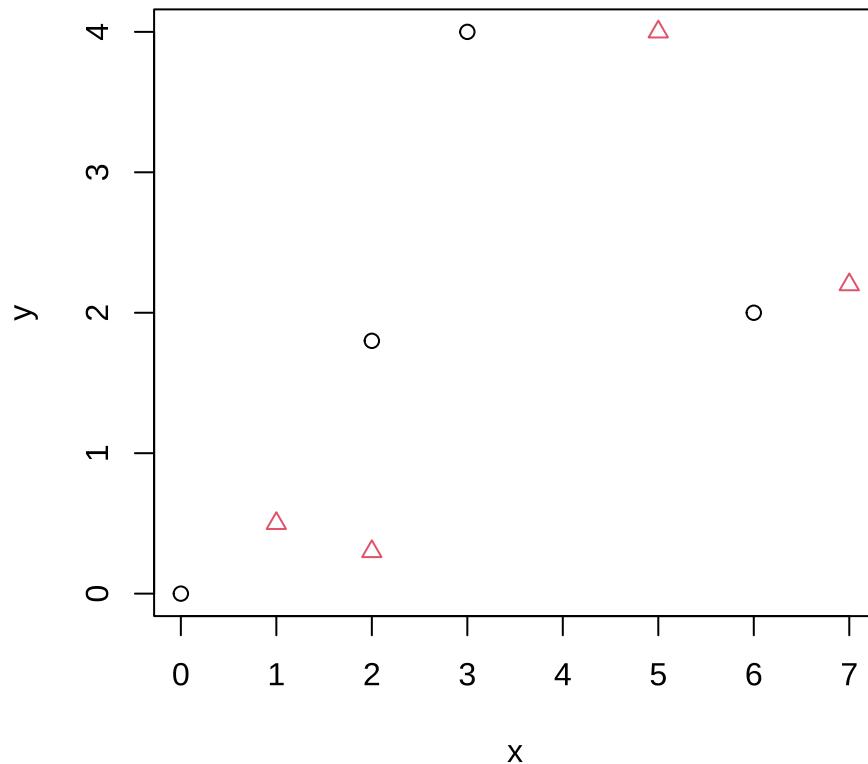
913 Sehen Sie sich die Hilfeseiten `?plot.default()` oder `?par()` an für weitere Informationen. Dort finden Sie  
914 auch eine vollständige Liste der Befehle. Einige Argumente können als Vektor übergeben werden. Hier z. B.  
915 die Farben und die Punktsymbole.

```

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")

```



916

917

---

**Aufgabe 17: Anpassen von Plots**


---

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

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

925

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

927 Die wichtigsten Funktionen sind

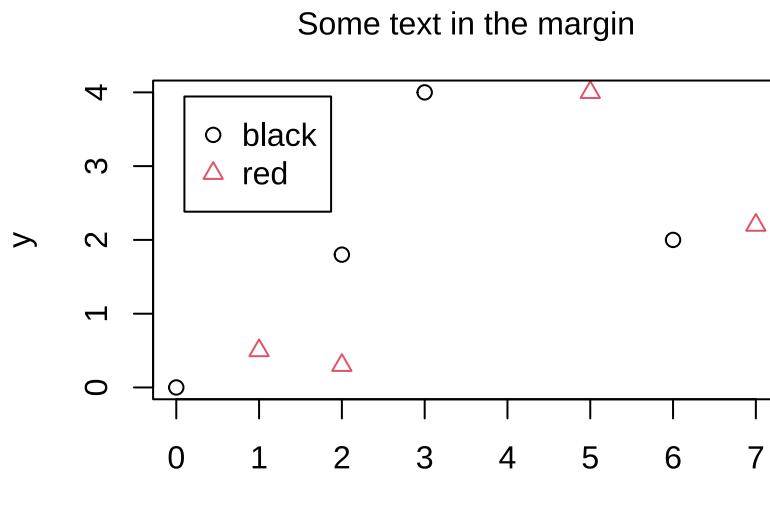
- 928 • `points()` - Fügt Punkte ein
- 929 • `lines()` - Fügt Linien ein

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

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

940 Legenden können im base plot nur als Low-Level Funktion hinzugefügt werden. Mit der Funktion `legend`  
 941 werden die Einträge und die entsprechenden Punktsymbole oder Linientypen dargestellt. In der folgenden  
 942 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")
```



943  
 944 Mit diesem grundsätzlichen Aufbau sollten Sie bereits in der Lage sein auch komplexe Grafiken schnell zu  
 945 gestalten. Wenn Sie mehrere Diagramme in einem Plot arrangieren möchten, können Sie mit dem `par()`  
 946 Befehl ein Arrangement definieren. Sie haben dann zusätzlich zu den bereits bekannten Grafikregionen noch  
 947 äußere Ränder (`outer margins`). Siehe Abbildung 6.

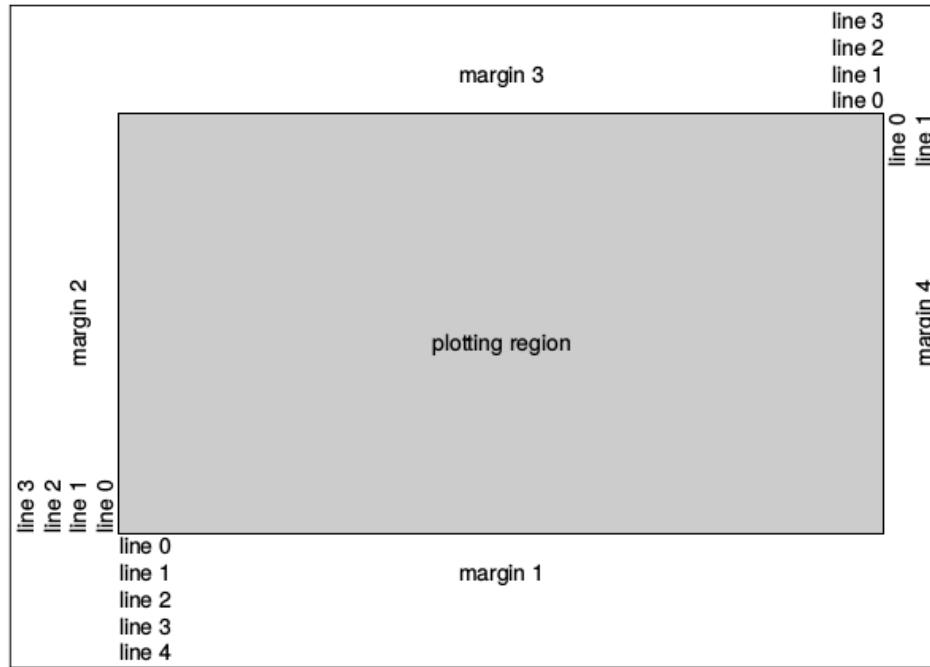
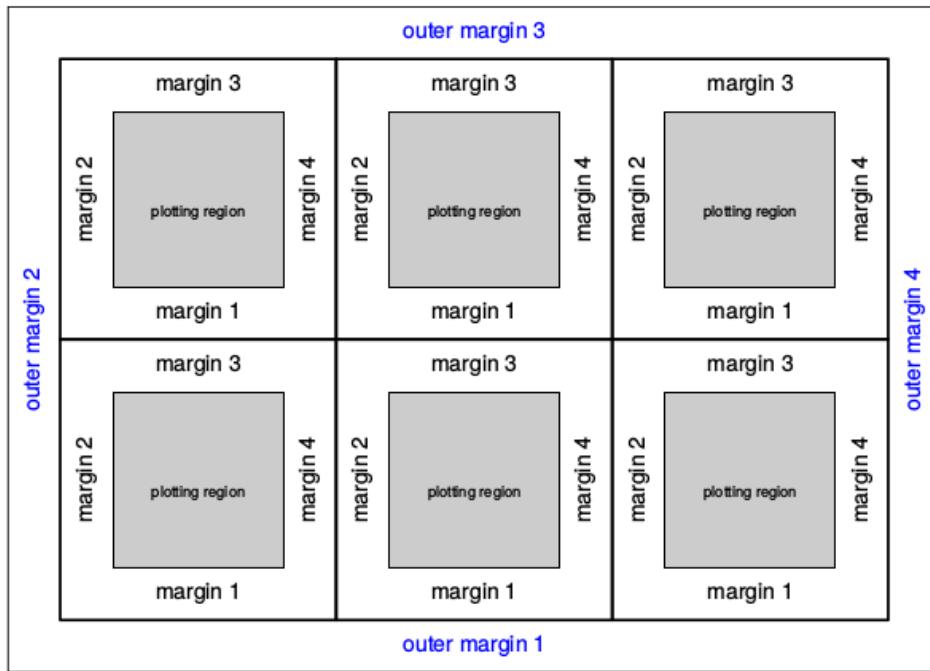


Abbildung 5: Grafikregionen eines base plots in R.

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

948 **8.1.1 Mehrere Panels**

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

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

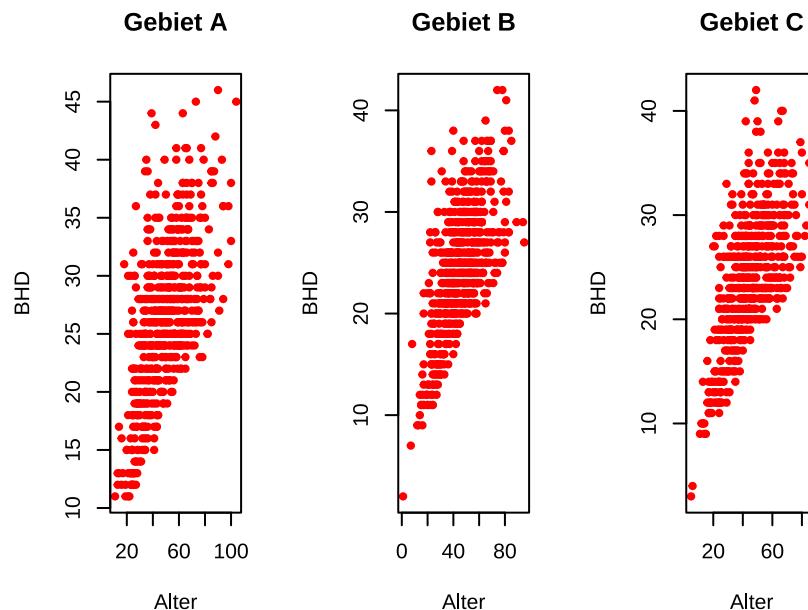
952 Teilt den Plot in eine Zeile und drei Zeilen (= drei Plots nebeneinander).

```
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")
```



953

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

956 **8.1.2 Speichern von Abbildungen**

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

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

963 Beispiele für Rastergrafiken sind

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

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

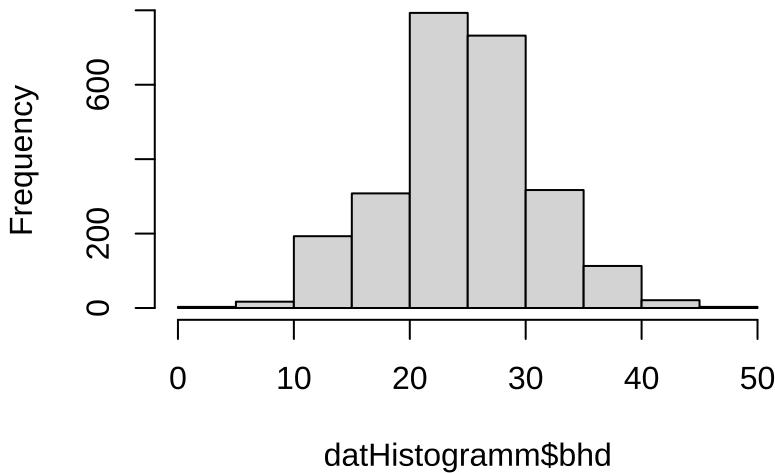
970 *Achtung*, wenn Sie die Funktion `dev.off()` nicht aufrufen, werden alle nachfolgenden Plots in die gleiche  
 971 Datei geschrieben. Falls Sie nach einem Versuch einen Plot zu speichern plötzlich keine weiteren Plots mehr  
 972 sehen, führen Sie einige Mal die Funktion `dev.off()` aus.

## 973 8.2 Histogramme

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

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

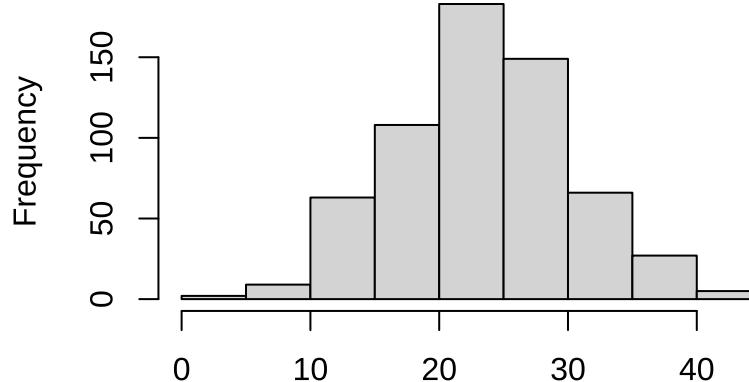
### Histogramm of datHistogramm\$bhd



981

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

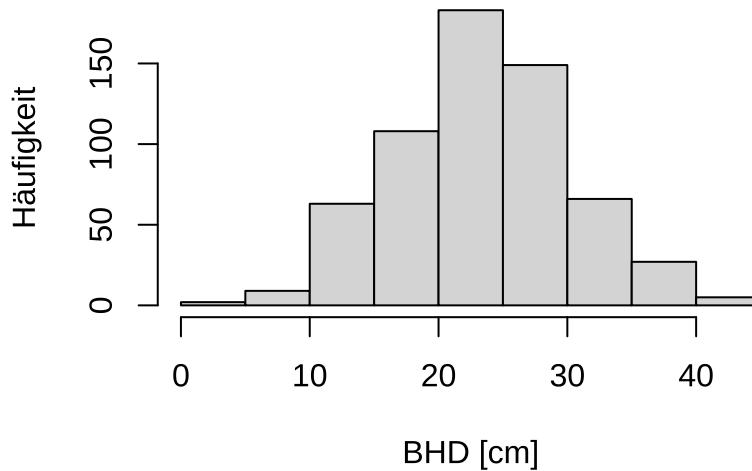
### Histogramm of datHistogramm\$bhd[datHistogramm\$art == "EI"]



982

```
# 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

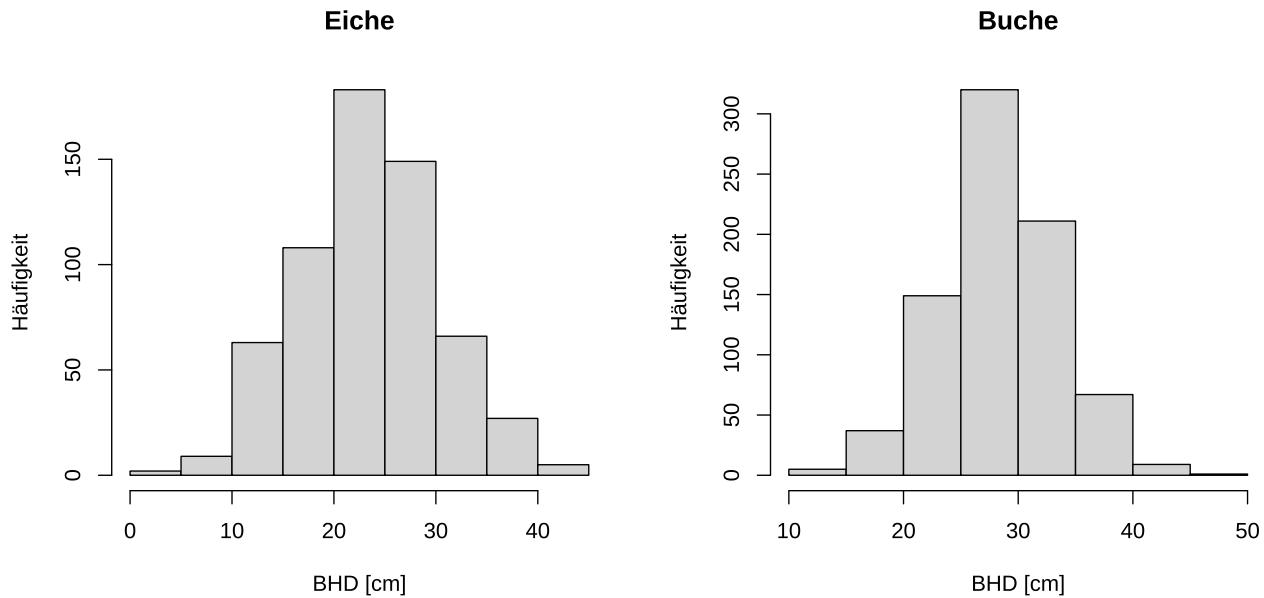


983

984 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")
```

985



986

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

### 8.3 Boxplots

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

Boxplots bestehen aus drei Komponenten:

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

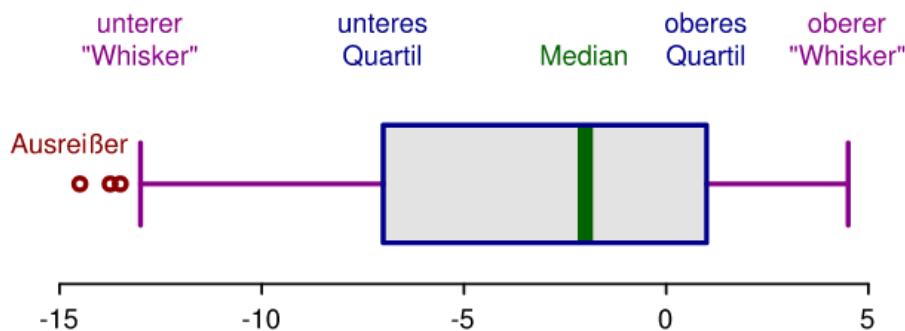
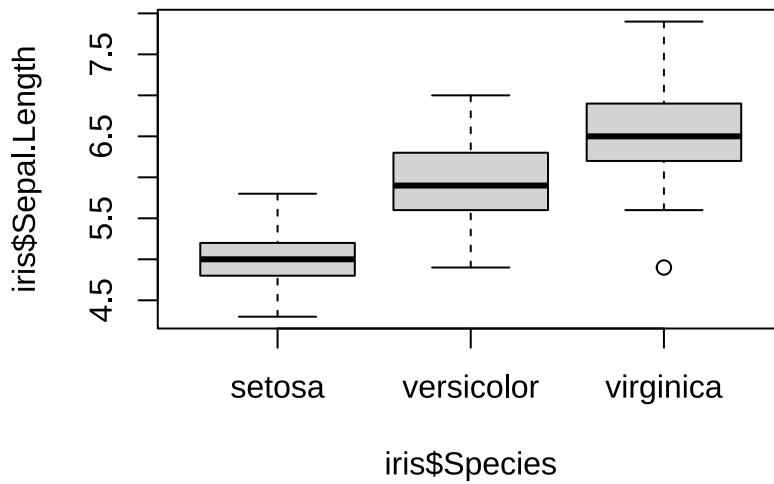


Abbildung 7: Schematische Darstellung eines Boxplots (Quelle: Von RobSeb - Eigenes Werk, CC BY-SA 3.0, <https://commons.wikimedia.org/w/index.php?curid=14697172>).

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

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

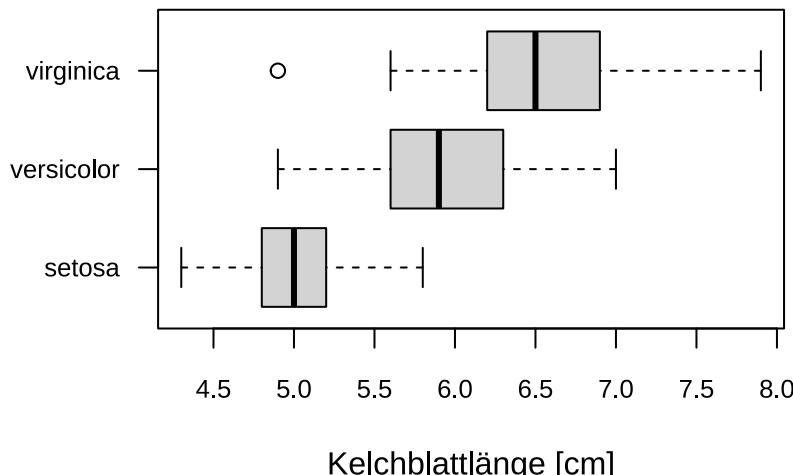
```
boxplot(iris$Sepal.Length ~ iris$Species)
```



1007

1008 Etwas eleganter ist es wenn wir das Argument `data` verwenden und den Plot etwas anpassen. Diese Schreib-  
1009 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  
)
```



1010

1011

### 1012 Aufgabe 18: Boxplots

1013

- 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
- 1017 Erstellen Sie Boxplots für jedes Gebiet und innerhalb der Gebiete für jede Art.

## 8.4 ggplot2: Eine Alternative für Abbildungen

1018 ggplot2 ist ein alternatives Plotting-System in R. Sie können mit ggplot2 also grundsätzlich Abbildungen  
 1019 mit dem selben Inhalt erstellen, wie mit Base Plots. Die Syntax und die optische Darstellung unterscheiden  
 1020 sich jedoch grundsätzlich. ggplot2 basiert auf den *grammar of graphics* von Leland Wilkinson. Die Idee  
 1021 ist, alle nötigen Informationen der Abbildung miteinander zu verknüpfen. ggplot2 ist also diametral zu  
 1022 Base Plots. Während Sie in base plot alle Elemente selbst definieren, ist die Idee von ggplot2, dass Sie  
 1023 nur die Daten und den Diagrammtypen übergeben und die Funktion selbst eine schöne Abbildung erstellt.  
 1024 Selbstverständlich können Sie aber auch in ggplot2 viele Einstellungen vornehmen. Im base plot sehen  
 1025 Abbildungen zunächst sehr karg und etwas unfertig aus. Sie müssen viele Einstellungen vornehmen, um eine  
 1026 publizierfähige Grafik zu produzieren. In ggplot2 sollen auch die einfachste Abbildungen schon ästhetisch  
 1027 sein. Mit diesen gebündelten Informationen kann ggplot2 die Abbildung automatisch verschönern. So  
 1028 werden bspw. die Legenden automatisch erzeugt und auch die Formatierungen automatisch an die Datenlage  
 1029 angepasst. ggplot2 nimmt der\*dem Entwickler\*in also Arbeit ab. Dadurch sind die Abbildungen schon ohne  
 1030 viel Nacharbeit schick. Nachteil ist, dass der\*dem Entwickler\*in weniger Möglichkeiten zur Einstellung zur  
 1031 Verfügung stehen und nuterspezifische Sonderwünsche somit schwerer umsetzbar sind. Sehen Sie sich das  
 1032 Cheatsheet zu ggplot2 an. Es ist in RStudio unter [Help > Cheatsheets](#) zu finden.  
 1033

1034 Bei ggplot2 sind Anweisungen zu den Daten und Anweisungen zur Darstellung voneinander getrennt. Die  
 1035 Daten werden in den Ästhetikbefehl übergeben und dort klassifiziert. Dann folgen die Darstellungsanweisungen.  
 1036 Ähnlich wie bei Base Plots, werden die Grafikelemente ebenenweise nacheinander programmiert, jedoch mit  
 1037 einem + verbunden. Und hier liegt der wesentliche Unterschied zu Base Plots. Durch die + werden die Ebenen  
 1038 zu einem Befehl verbunden und damit gleichzeitig erstellt.  
 1039 Die Erweiterung wird zunächst geladen<sup>7</sup>. Wir laden außerdem den Datensatz **iris**. Der Datensatz ist in R  
 1040 fest integriert. Siehe `?iris` für mehr Informationen.

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

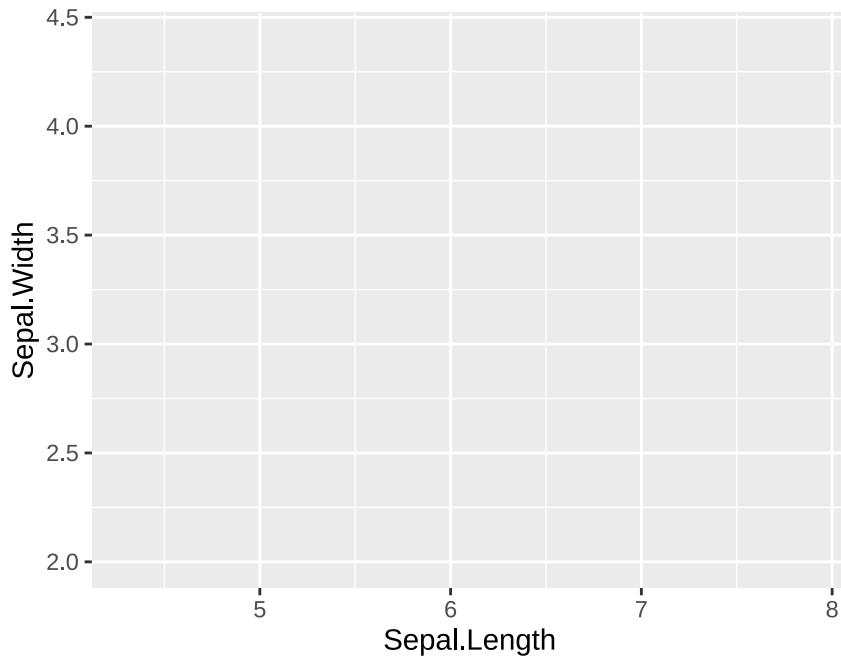
```
1041 ##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
1042 ## 1          5.1       3.5      1.4       0.2  setosa
1043 ## 2          4.9       3.0      1.4       0.2  setosa
1044 ## 3          4.7       3.2      1.3       0.2  setosa
1045 ## 4          4.6       3.1      1.5       0.2  setosa
1046 ## 5          5.0       3.6      1.4       0.2  setosa
1047 ## 6          5.4       3.9      1.7       0.4  setosa
```

1048 Die Ästhetik wird bspw. folgendermaßen definiert.

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

---

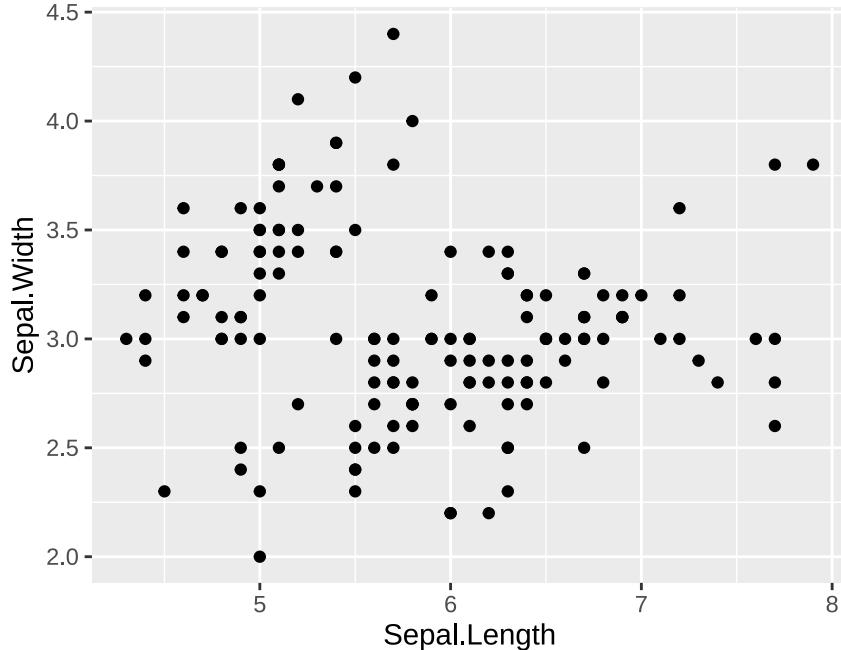
<sup>7</sup>Wir haben bis jetzt immer nur mit base R gearbeitet. D.h. wir haben nur Funktionen verwendet, die R bereits zur Verfügung stellt. Eine der großen Stärken von R sind die Erweiterungen (oder auch Pakete genannt). ggplot2 ist so eine Erweiterung, die einmal mit `install.packages("ggplot2")`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



1049

1050 Dieser Befehl zeichnet noch keine Daten. Die Daten werden lediglich herangezogen, um einen leeren Plot für  
1051 die Daten zu erstellen. In dem Beispiel wird die Variable `Sepal.Length` aus dem `data.frame iris` als x und  
1052 `Sepal.Width` als y Variable definiert. Diese Informationen stehen den folgenden Layern nun zur Verfügung,  
1053 sodass nach den + nur noch x und y verwendet werden müssen. Um bspw. einen Scatterplot zu erstellen  
1054 wird ein `geom_point()` Layer hinzugefügt. x und y werden automatisch an `geom_point` übergeben. Weitere  
1055 Einstellung sind in diesem Beispiel nicht notwendig, wären jedoch möglich. Siehe `?geom_point()`.

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



1056

1057

---

1058 **Aufgabe 19: Abbildungen mit ggplot2**

---

1059

1060 Verwenden Sie die Daten aus Aufgabe 16 und erstellen Sie einen Scatterplot mit ggplot2 wie in Aufgabe 16.

1061

1062 Wir haben mit der Funktion `geom_point()` demm Plot eine Punktgeometrie hinzugefügt. Es gibt noch viele  
1063 weitere Geometrien. Die wichtigsten sind:

- 
- 1064 •
- `geom_line()`
- für eine Linie.
- 
- 1065 •
- `geom_histogram()`
- um ein Histogramm zu erstellen.
- 
- 1066 •
- `geom_boxplot()`
- um einen Boxplot zu erstellen.
- 
- 1067 •
- `geom_bar()`
- um ein Säulendiagramm zu erstellen.

1068 Welche Geometrie die richtige ist, richtet nach dem Typ der darzustellenden Variablen. Beispielsweise  
1069 bietet sich `geom_point()` an, wenn man zwei kontinuierliche Variable darstellen möchte. Wenn man hin-  
1070 gegen die Verteilung von einer kontinuirlchen Variable darstellen möchte, dann bietet sich ein Histogramm  
1071 (`geom_histogram()`) oder auch eine geschätzte Dichte (z.B. `geom_density()`) an.

1072

---

1073 **Aufgabe 20: Abbildungen mit ggplot2**

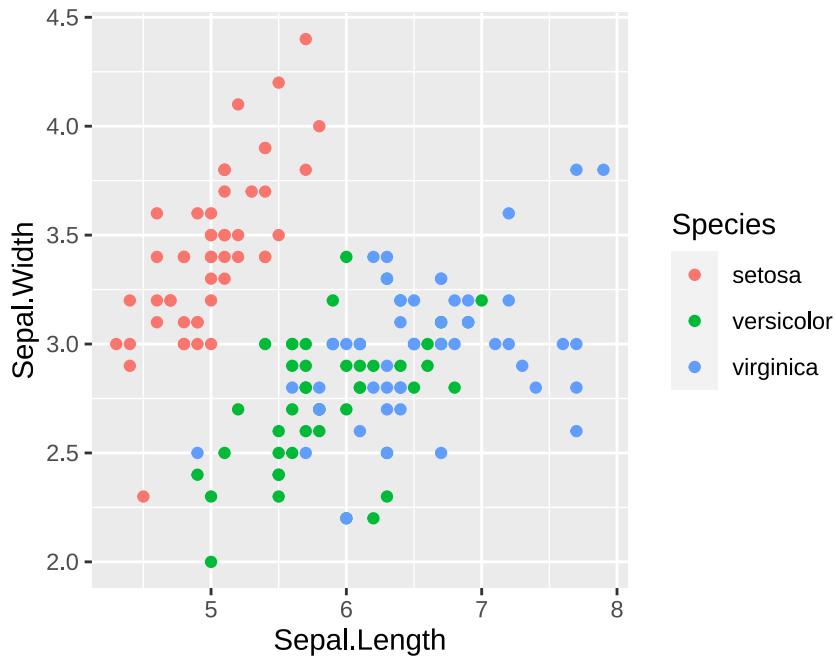
---

10741075 Verwenden Sie die den Iris Datensatz und erstellen Sie mit ggplot2 einen Plot der die Verteilung der Länge  
1076 der Kelchblätter zeigt (Spalte `Sepal.Length`).

1077

1078 Eine der Stärken von ggplot2 ist, dass man den Wert unterschiedlicher Variable auf unterschiedlichen  
1079 Komponenten des Plots abbilden kann. Wir haben bis jetzt ein bzw. zwei Variable auf der x- und y-Achse  
1080 abgebildet. Wir können aber ein weitere Variablen verwenden um das Aussehen des Plots zu beeinflussen.  
1081 Beispielsweise können wir die Farbe der Punkte (für `geom_point()`) mit dem Argument `col` beeinflussen.

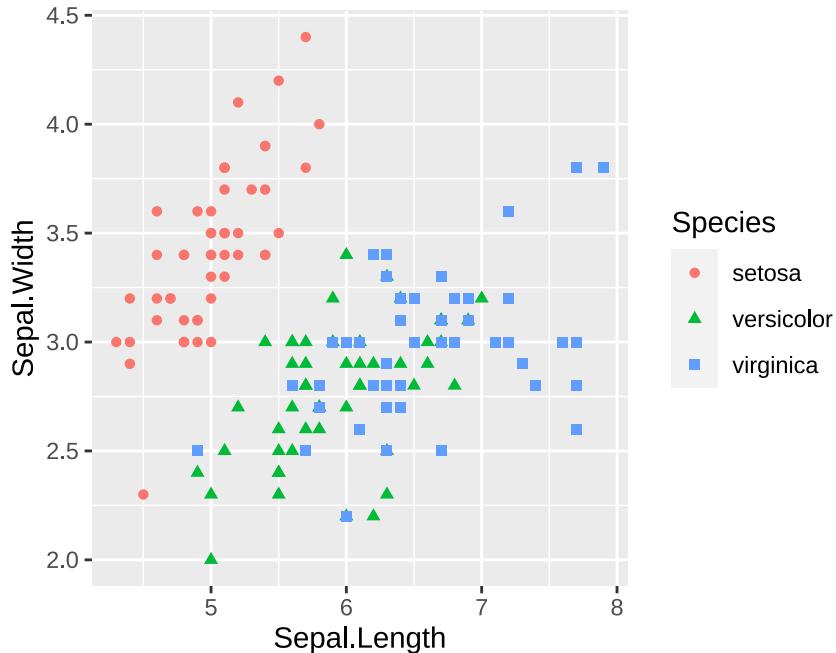
```
ggplot(iris, aes(x = Sepal.Length, y = Sepal.Width, col = Species)) +  
  geom_point()
```



1082

1083 Somit bekommt jede Irisart eine eigene Farbe<sup>8</sup>. Gleichesmaßen können wir die Punktart (**shape**), die  
1084 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()
```

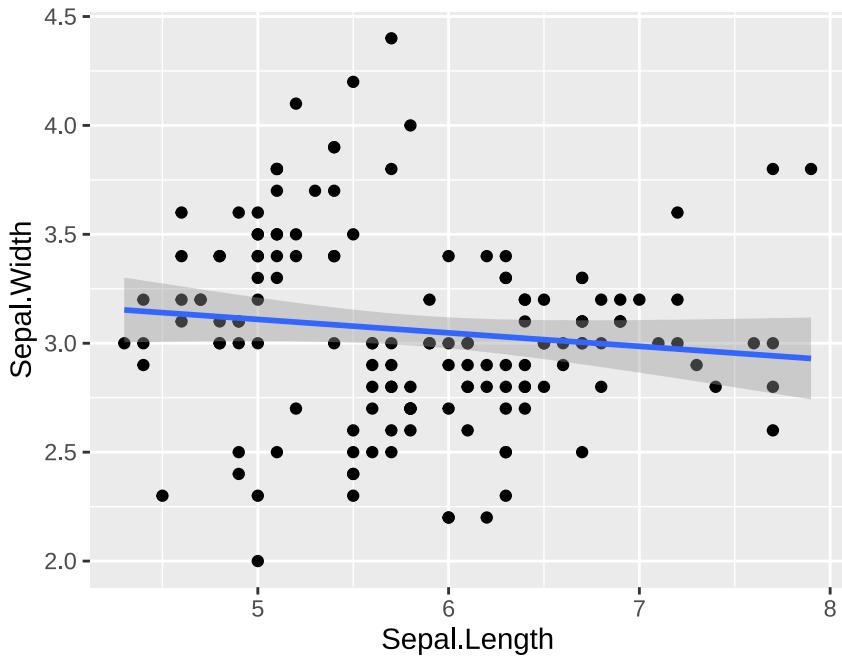


1085

1086 In dem Plot ist die Information zu der Art redundant (einmal als Farbe und einmal Symbolart).  
1087 Ein weitere sehr nützliche Geometrie ist **geom\_smooth()**, die es erlaubt eine Trendlinie hinzuzufügen.

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

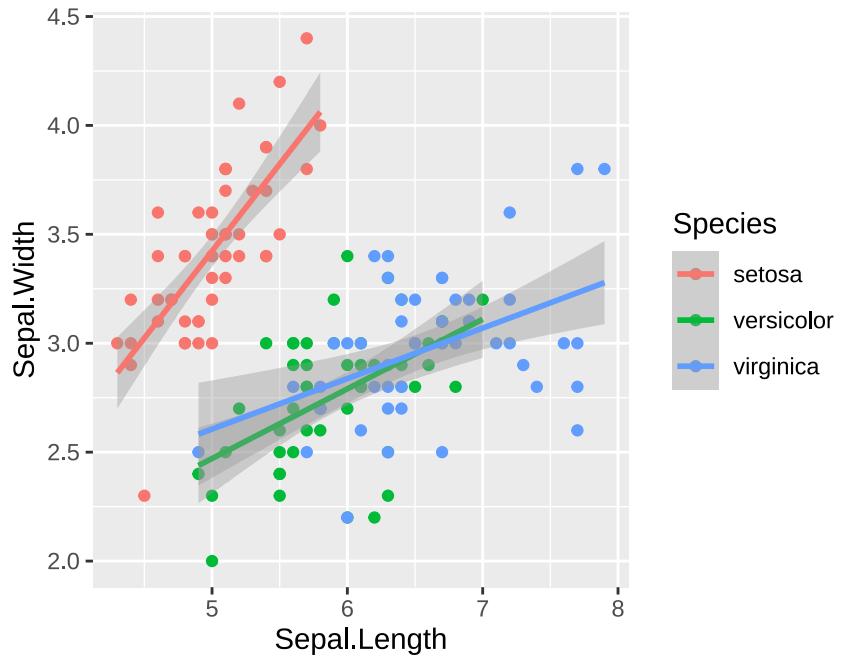
```
ggplot(iris, aes(x = Sepal.Length, y = Sepal.Width)) +
  geom_point() + geom_smooth(method = "lm")
```



1088

1089 Mit `method = "lm"` wird festgelegt, dass die Trendlinie gerade sein soll (es wird eine lineare Einfachregression  
1090 angepasst). Wenn wird wieder eine gruppierende Variable einführen (z.B. die Beobachtungen nach Art auf  
1091 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")
```



1092

1093

1094 **Aufgabe 21: Anpassen von Plots**  
1095

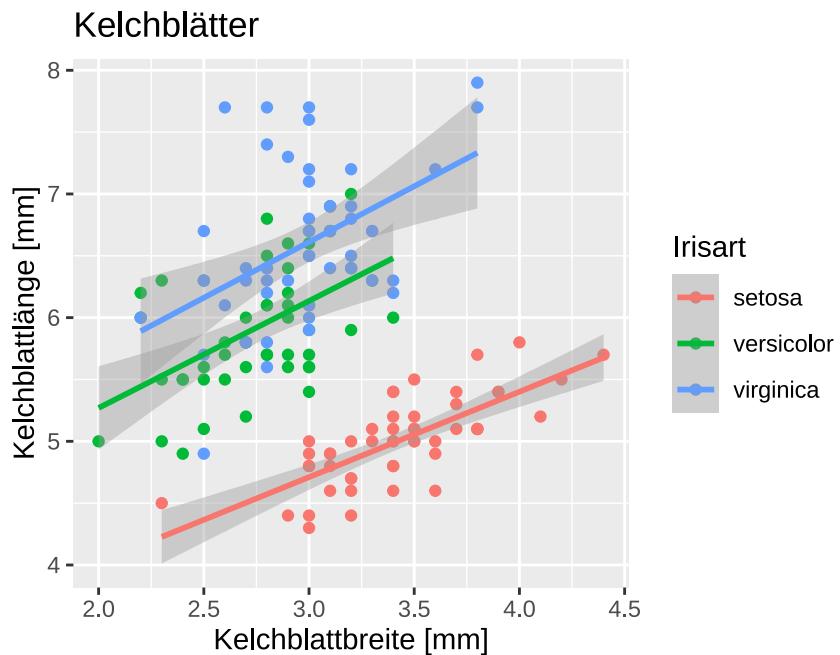
- 1096 Lesen Sie den Datensatz `data/bhd_1.txt` ein und erstellen Sie einen Boxplot für die Verteilung des BHDs  
 1097 für jede Baumart. In einem zweiten Schritt verwenden Sie erst `col = gebiet` und dann `fill = gebiet`.  
 1098 Welchen Unterschied stellen Sie fest?

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

1099

- 1100 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")
```



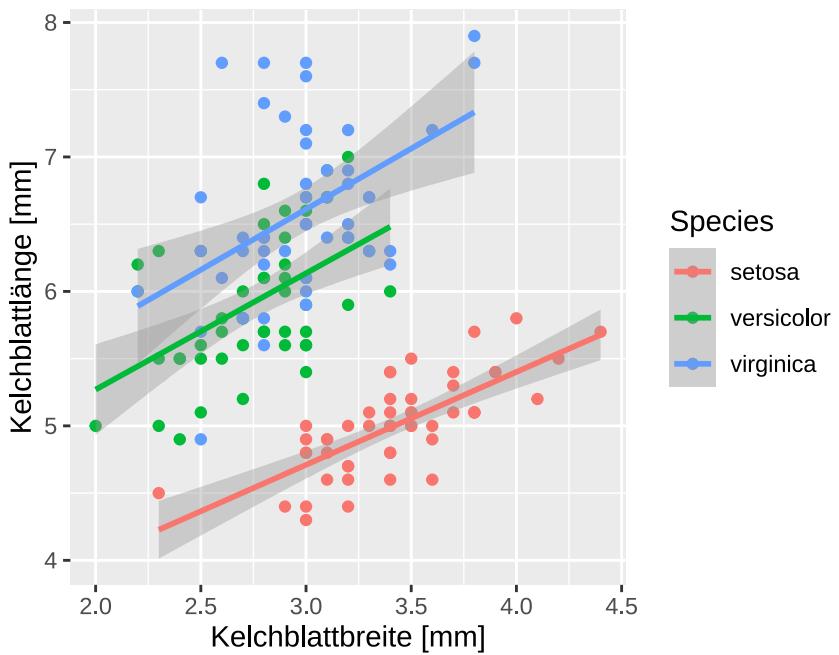
1101

- 1102 Statt einen langen Befehl zu tippen, kann ein `ggplot()` auch zwischengespeichert und wieder aufgerufen bzw.  
 1103 angepasst werden. Das ist vor allem sinnvoll, wenn mehrere Abbildungen auf dem selben Zwischenergebnis  
 1104 aufbauen sollen.

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

- 1105 Wir können jetzt mit `p1` weiter arbeiten und beispielsweise eine Beschriftung hinzufügen.

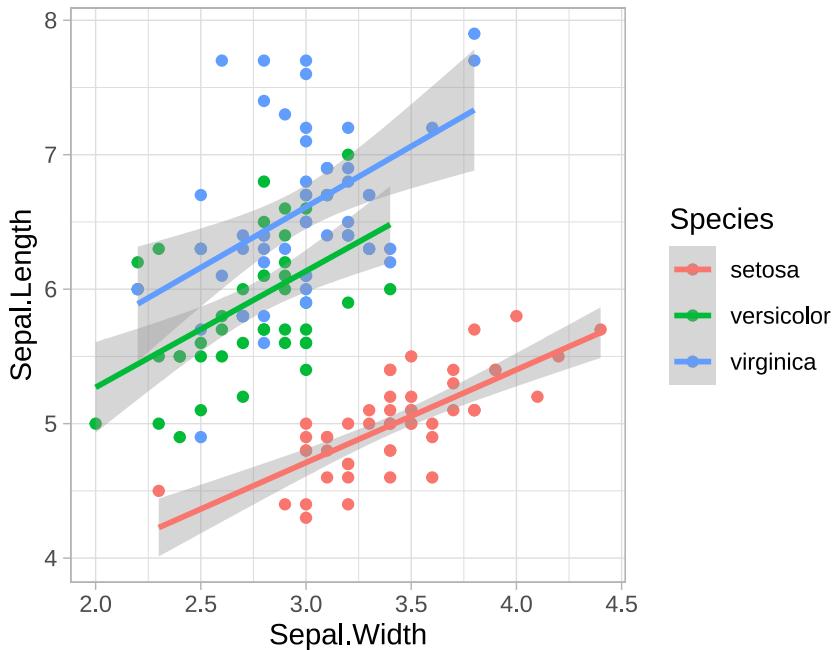
```
p1 + labs(x = "Kelchblattbreite [mm]", y = "Kelchblattlänge [mm]")
```



1106

1107 Oder auch den ganzen Plot anpassen. Dafür gibt es *themes*. Es gibt eine Reihe von vorgefertigten *themes*  
1108 oder man kann diese auch selber erstellen (das ist aber nicht Teil dieses Kurses).

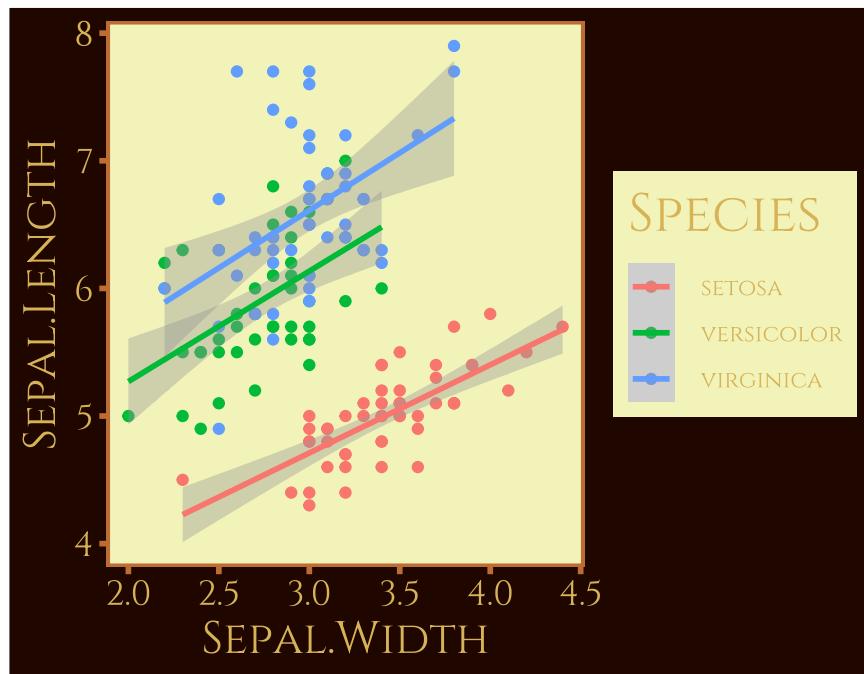
```
p1 + theme_light()
```



1109

1110 Weitere *themes* sind: `theme_bw()`, `theme_linedraw()` oder `theme_dark()`. Es gibt extra Pakete die viele  
1111 zusätzliche weitere *themes* anbieten. Dazu gehört z.B. das Paket `ggthemes` oder `ThemePark`. Während  
1112 `ggthemes` hauptsächlich durchdachte Grafikkonzepte liefert, sind die Themes aus `ThemePark` eher Popkultur  
1113 und nicht 100 %ig ernst gemeint.

```
p1 + theme_gamethrones()
```

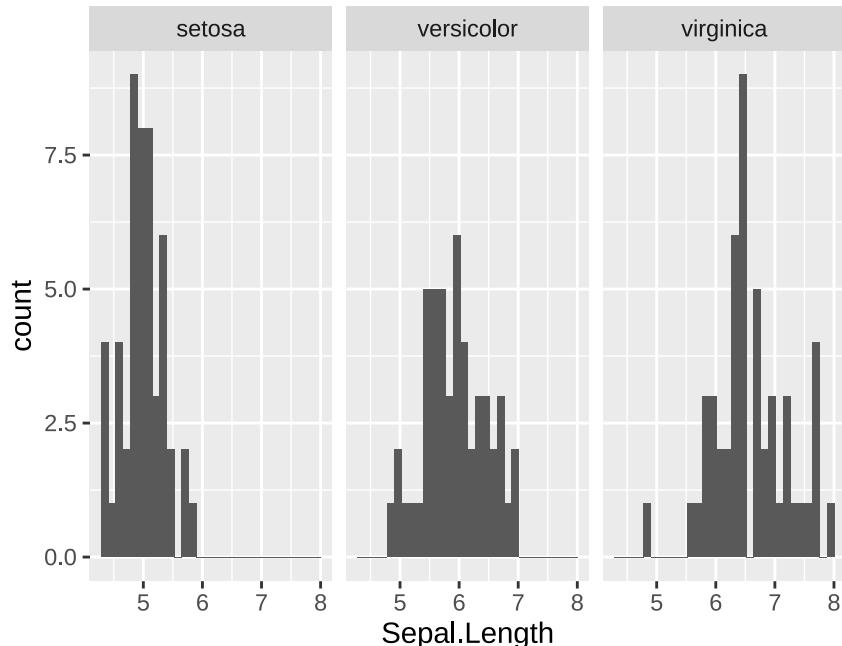


1114

#### 8.4.1 Multipanel Abbildungen

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

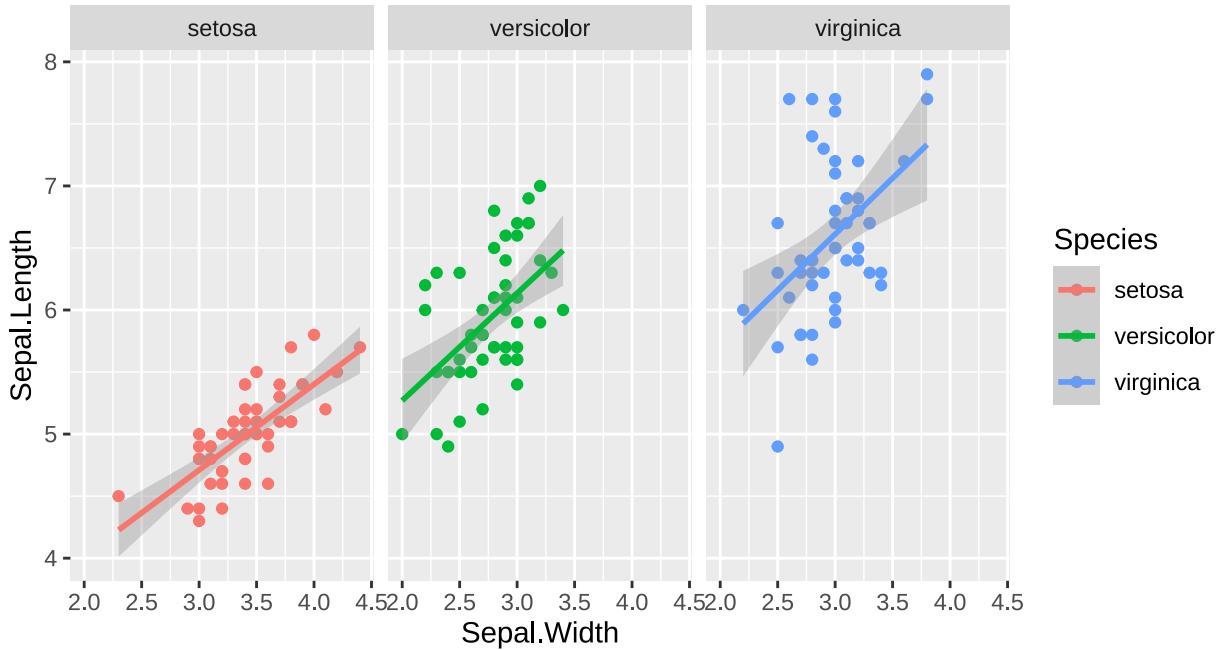
```
ggplot(iris, aes(Sepal.Length)) + geom_histogram() +
  facet_grid(~ Species)
```



1119

1120 Die Funktion `facet_grid()` erzeugt einen *Grid* und beschriftet die Grid-Zeilen und -Spalten, während  
 1121 `facet_wrap()` für jedes Panel (Diagramm) eine eigene Überschrift erzeugt. Beim Arrangieren der Diagramme  
 1122 wird der Vorteil von `ggplot2` gegenüber `base plot` besonders deutlich. Während es im `base plot` System  
 1123 mühsam ist, die Abbildungen zu arrangieren, übernimmt `ggplot2` das Arrangement automatisch und fügt  
 1124 sogar eine automatisch Legende hinzu. Die Achsenabschnitte werden so angepasst, dass die Daten vergleichbar  
 1125 sind.

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



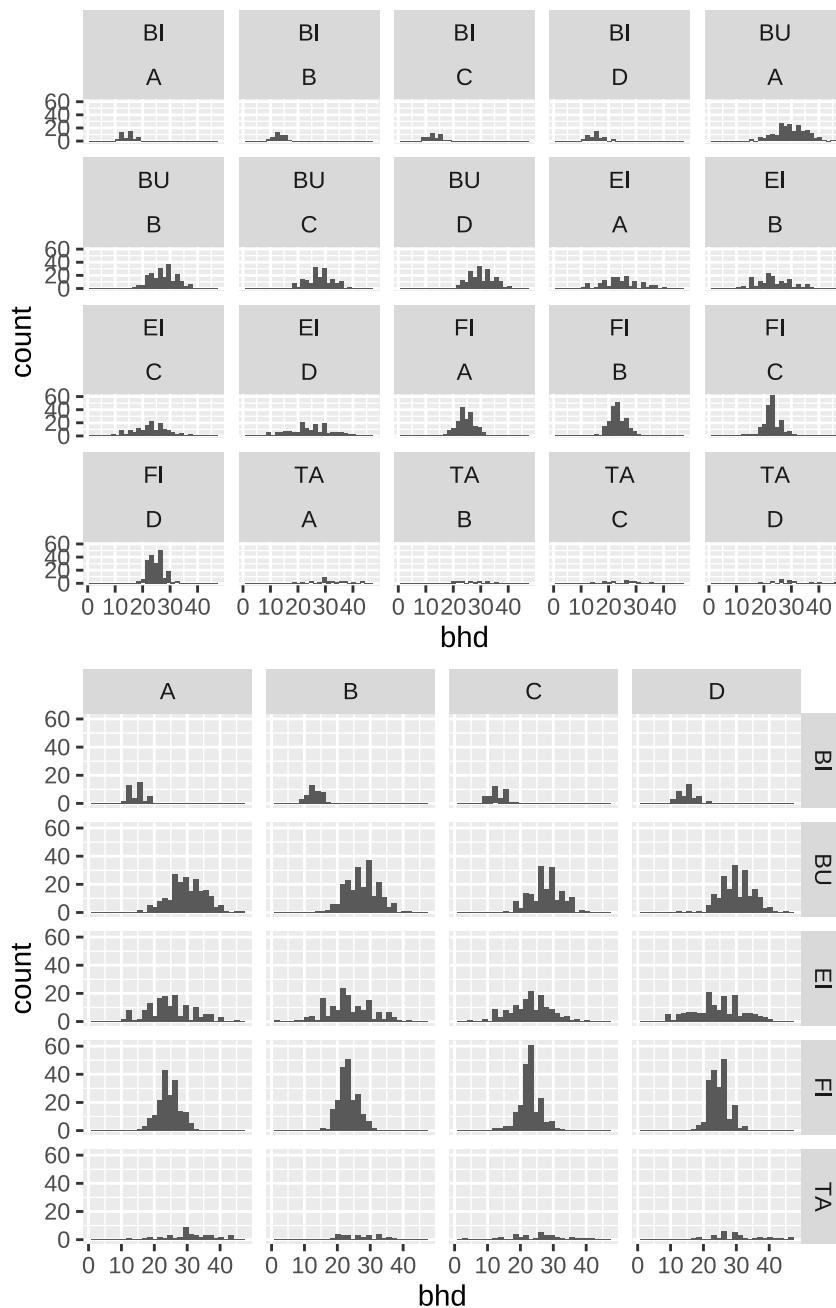
1126

1127

### 1128 Aufgabe 22: Multipanel Abbildungen

---

1130 Lesen Sie erneut den Datensatz `daten/bhd_1.txt` ein und speichern Sie diesen in die Variable (`dat_bhd`).  
 1131 Erstellen Sie für jede Art und Gebiet ein Histogramm. Welche Unterschiede können Sie feststellen, wenn Sie  
 1132 `facet_grid()` oder `facet_wrap()` verwenden?



#### 1135 8.4.2 Plots kombinieren

1136 Es gibt Situationen, in denen **unterschiedliche** Plots miteinander kombiniert werden müssen. Im vorherigen  
 1137 Abschnitt wurde dies immer anhand einer gruppierenden Kovariate gemacht. Aber es gibt auch Situationen, in  
 1138 denen das nicht möglich ist. Beispielsweise wenn ein Histogramm und ein Scatterplot vom gleichen Datensatz  
 1139 zusammengefasst werden sollen. Dafür bietet sich das Paket **patchwork** an<sup>9</sup>.

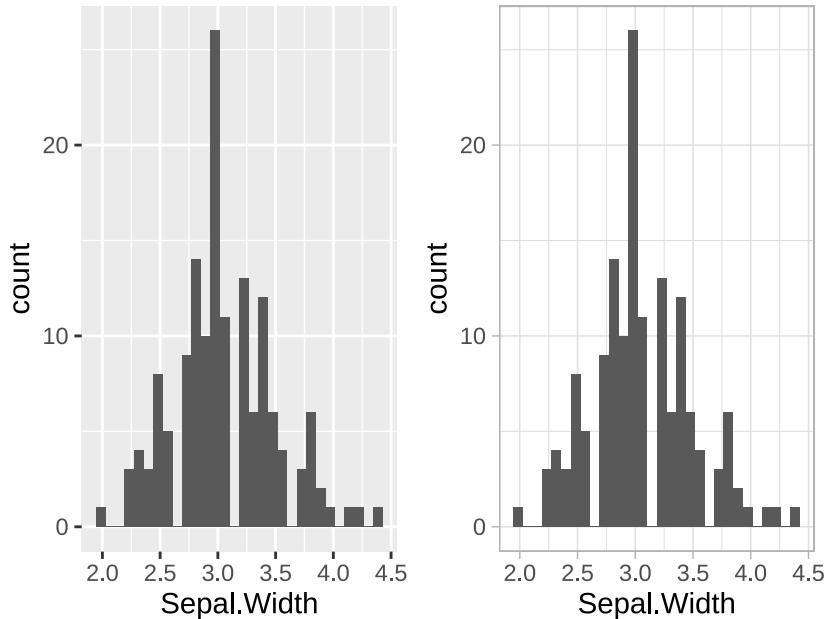
1140 Als erstes können wir zwei (oder natürlich auch mehrere Plots) erstellen. Hier unterscheiden sich die Plots  
 1141 lediglich durch das Aussehen.

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

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

- 1142 Dann müssen können wir diese Plots ebenfalls mit `+` zusammenfügen.

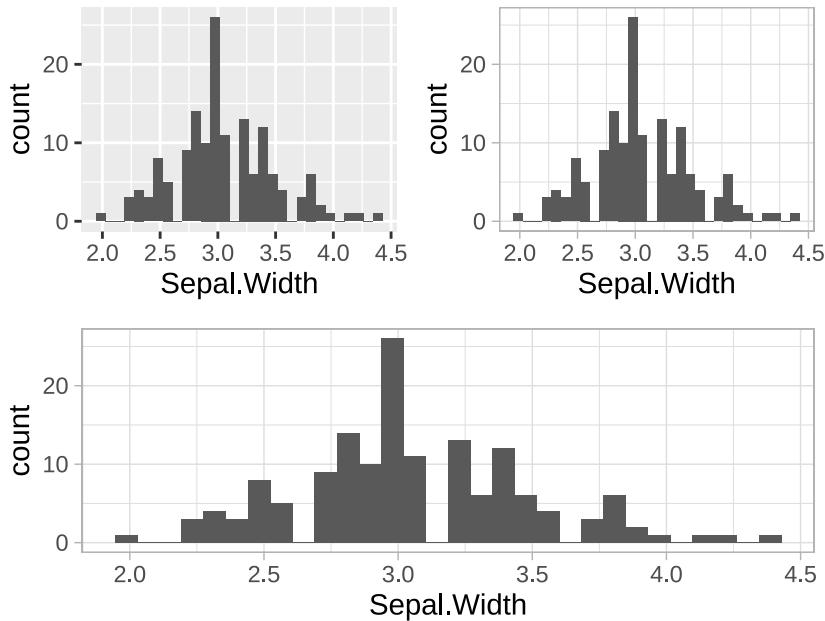
```
library(patchwork)
p1 + p2
```



1143

- 1144 Natürlich können auch weitere Plots hinzugefügt werden (auch in unterschiedlichen Dimensionen):

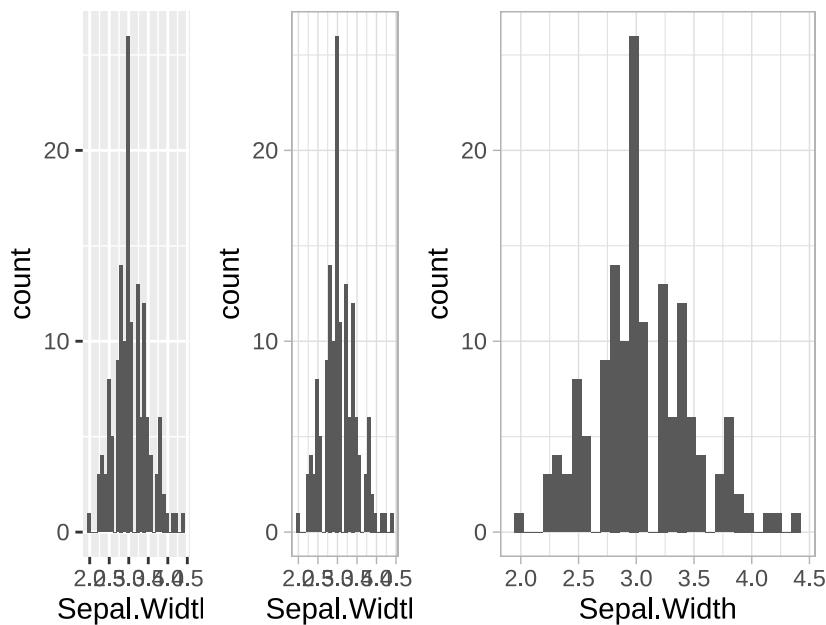
```
(p1 + p2) / p2
```



1145

- 1146 Des weiteren können mit `|` auch Plots gegenüber gestellt werden.

(p1 + p2) | p2

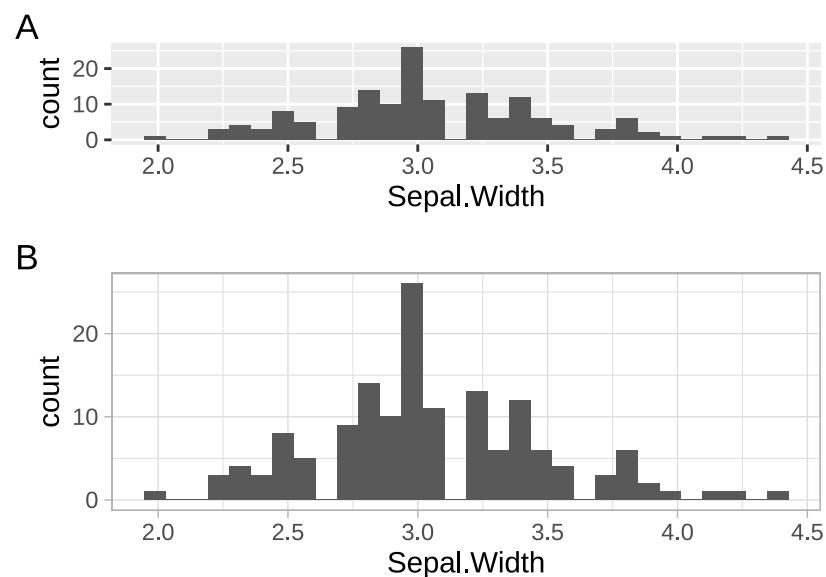


1147

1148 Weitere Optionen können mit `plot_layout()` und `plot_annotation()` angepasst werden. Mit  
 1149 `plot_layout()` kann die Anordnung der Plots bestimmt werden (z.B. über die Argument `nrow`  
 1150 und `ncol`), sowie deren *relative* Größe (über die Argumente `widths` und `heights`). Mit der Funktion  
 1151 `plot_annotation()` können zusätzliche Beschriftungen hinzugefügt werden, wie beispielsweise eine Titel  
 1152 (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



1153

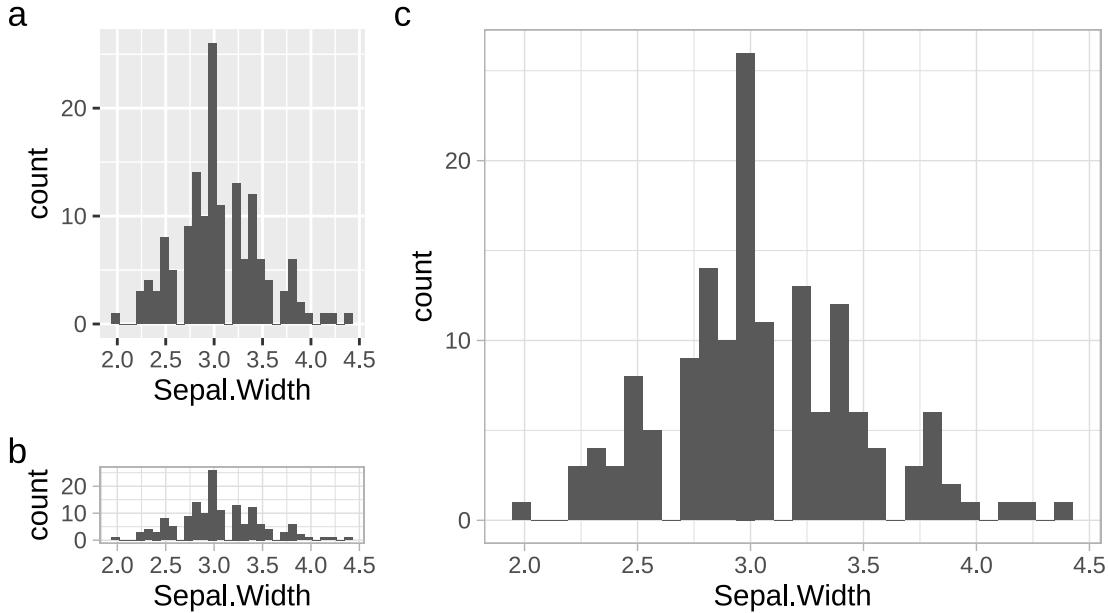
1154

---

1155 **Aufgabe 23: Mehrere Plots zusammenfügen**  
1156

---

1157 Versuchen Sie die folgende Zusammenstellung der Plots nachzumachen:



1158

1159 **8.4.3 Speichern von plots**

1160 Sie können mit `ggsave()` eine zwischen-gespeicherte Abbildung exportieren, indem Sie den Variablennamen  
1161 übergeben. Wenn Sie keine Variable übergeben, wird automatisch die letzte Abbildung gespeichert. Das  
1162 Dateiformat wird aus dem Dateinamen übernommen. Die Größe der Inhalte können Sie sehr leicht mit den  
1163 Argumenten `width` und `height` oder `scale` gesteuert werden.

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

## 1164 9 Mit Daten arbeiten

### 1165 9.1 dplyr eine Einführung

1166 `dplyr` ist eine Erweiterung von R (ein Paket), die das Ziel hat, den Umgang mit Daten einfacher und  
1167 schneller zu machen.

1168 `dplyr` definiert 5 Verben, um mit Daten zu arbeiten. Diese sind:

- 1169 • `filter`
- 1170 • `select`
- 1171 • `arrange`
- 1172 • `mutate`
- 1173 • `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))
```

1174 Damit die Funktionen aus `dplyr` verwendet werden können, müssen wir als erstes das Paket `dplyr` laden.  
1175 The online Resource [R for Data Science](#) gibt einen sehr guten, tieferen Eindruck in die Arbeit mit `dplyr`  
1176 und erklärt auch die `dplyr` Philosophie noch Mal sehr genau für Beginner.

```
library(dplyr)
```

1177 Sollte dies zu einer Fehlermeldung führen, dann müssen Sie das Paket `dplyr` erst installieren. Dafür müssen  
1178 Sie `einmalig install.packages("dplyr")` installieren.

1179 `dplyr` stellt unterschiedliche Funktionen zum Arbeiten mit Daten zur Verfügung. Es gibt fünf Grundfunktionen  
1180 für die am häufigsten vorkommenden Operationen. Mit der Funktion `filter()` können unterschiedliche  
1181 Beobachtungen gefiltert werden:

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

```
1182 ##   id plot bhd alter
1183 ## 1   1    1  50   10
1184 ## 2   2    1  29   30
1185 ## 3   3    2  13   31
1186 ## 4   4    2  23   24
1187 ## 5   5    3  25   25
```

1188 Es können auch mehrere Spalten verwendet werden.

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

```
1189 ##   id plot bhd alter
1190 ## 1   2    1  29   30
1191 ## 2   3    2  13   31
1192 ## 3   4    2  23   24
```

```
1193 ## 4 5 3 25 25
```

1194 Natürlich kann genau das gleiche Ergebnis mit dem ‘normalen’ R erreicht werden, dies wäre dann:

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

```
1195 ## id plot bhd alter
```

```
1196 ## 2 2 1 29 30
```

```
1197 ## 3 3 2 13 31
```

```
1198 ## 4 4 2 23 24
```

```
1199 ## 5 5 3 25 25
```

1200 Eine weitere Funktion aus dem Paket **dplyr** ist **select()**. Damit können Spalten aus einem **data.frame**

1201 ausgewählt werden. Dabei können auch die Spaltennamen unbenannt werden.

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

```
1202 ## bhd
```

```
1203 ## 1 50
```

```
1204 ## 2 29
```

```
1205 ## 3 13
```

```
1206 ## 4 23
```

```
1207 ## 5 25
```

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

```
1208 ## bhd id
```

```
1209 ## 1 50 1
```

```
1210 ## 2 29 2
```

```
1211 ## 3 13 3
```

```
1212 ## 4 23 4
```

```
1213 ## 5 25 5
```

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

```
1214 ## BHD id
```

```
1215 ## 1 50 1
```

```
1216 ## 2 29 2
```

```
1217 ## 3 13 3
```

```
1218 ## 4 23 4
```

```
1219 ## 5 25 5
```

1220 Mit der Funktion **arrange()** können die Beobachtungen in einem **data.frame** sortiert werden.

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

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

```
1222 ## 1 3 2 13 31
```

```
1223 ## 2 4 2 23 24
```

```
1224 ## 3 5 3 25 25
```

```
1225 ## 4 2 1 29 30
1226 ## 5 1 1 50 10
```

1227 Mit der Funktion `desc()` kann die Anordnung in absteigender Reihenfolge sortiert werden.

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

```
1228 ## id plot bhd alter
1229 ## 1 1 1 50 10
1230 ## 2 2 1 29 30
1231 ## 3 5 3 25 25
1232 ## 4 4 2 23 24
1233 ## 5 3 2 13 31
```

1234 Mit der Funktion `mutate()` kann man eine neue Spalte hinzufügen.

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

```
1235 ## id plot bhd alter bhd_mm fl
1236 ## 1 1 1 50 10 500 1963.4954
1237 ## 2 2 1 29 30 290 660.5199
1238 ## 3 3 2 13 31 130 132.7323
1239 ## 4 4 2 23 24 230 415.4756
1240 ## 5 5 3 25 25 250 490.8739
```

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

```
1241 ## id plot bhd alter mean_bhd
1242 ## 1 1 1 50 10 28
1243 ## 2 2 1 29 30 28
1244 ## 3 3 2 13 31 28
1245 ## 4 4 2 23 24 28
1246 ## 5 5 3 25 25 28
```

1247 Mit der Funktion `summarise()` können Spalten zusammengefasst werden.

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

```
1248 ## mean_bhd mean_sd
1249 ## 1 28 13.63818
```

1250

---

1251 **Aufgabe 24: Datenmanipulation mit dplyr**

---

- 1253 1. Laden Sie den Datensatz `data/bhd_1.txt`  
 1254 2. Berechnen Sie folgende Werte für alle Einträge und speichern Sie die Ergebnisse in `erg1`
- 1255 • mittlerer `bhd`
  - 1256 • maximales `alter`
  - 1257 • die Standardabweichung des BHDs
  - 1258 • die Anzahl Bäume mit einem BHD > 30

1259 **9.2 Arbeiten mit gruppierten Daten**

1260 Zusätzlich können `mutate` und `summarise` auch auf gruppierte Daten angewendet werden. Dafür müssen  
 1261 wir erst die Funktion `group_by()` aufrufen und als Argumente die Variablen übergeben, die die Gruppen  
 1262 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    1   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     1    29     30    39.5
## 3     3     2    13     31    18
## 4     4     2    23     24    18
## 5     5     3    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>
```

```
1282 ## <dbl> <dbl>
1283 ## 1      1  39.5
1284 ## 2      2  18
1285 ## 3      3  25
```

1286

---

**Aufgabe 25: dplyr mit gruppierten Daten**

---

- 1289 1. Laden Sie den Datensatz `data/bhd_1.txt`
- 1290 2. Berechnen Sie für jede Baumart und Standort die folgenden Werte:
- 1291 • mittlerer `bhd`
  - 1292 • maximales `alter`
  - 1293 • die Standardabweichung des BHDs
  - 1294 • die Anzahl Bäume mit einem BHD > 30
- 1295 3. Ordnen Sie das Ergebnis nach Baumart und BHD.

1296 **9.3 pipes oder %>%**

1297 Mit *Pipes* (`%>%`) kann man das Ergebnis einer Funktion einfach an eine nachfolgende Funktion weiterreichen.

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

1298 Wir kennen bis jetzt:

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

1299 ## [1] 3.333333

1300 Mit *Pipes*, die durch das Symbol `%>%` dargestellt werden<sup>10</sup>, können wir das etwas vereinfachen und nacheinander schreiben:

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

1302 ## [1] 3.333333

1303 Oder sogar

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

1304 ## [1] 3.333333

1305

---

**Aufgabe 26: Pipes %>%**

---

1308 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.

- 1309 1. Laden Sie den Datensatz `data/bhd_1.txt`.
- 1310 2. Berechnen Sie für jede Baumart und Standort die folgenden Werte:
- 1311 • mittlerer `bhd`
  - 1312 • maximales `alter`
  - 1313 • die Standardabweichung des BHDs
  - 1314 • die Anzahl Bäume mit einem BHD > 30
- 1315 3. Ordnen Sie das Ergebnis nach Baumart und BHD.

1316 **9.4 Joins**

1317 Eine weitere häufige Aufgabe beim Daten Management ist es Daten zusammenzuführen. Nehmen Sie an, dass  
1318 wir folgende Aufnahmen gemacht haben

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

1319 und jeder Baum lediglich mit einer `id` versehen wurde. In einer zweiten Tabelle wurden dann weitere Daten  
1320 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")
)
```

1321 Ziel ist es jetzt die Bäume aus `aufnahmen` mit den Informationen aus den `metadaten` zu verbinden. Dazu  
1322 dient `id` als Bindeglied (oft auch Schlüssel genannt).

1323 Dazu gibt es vier Möglichkeiten.

1324 Zur Durchführung gibt es in base R die Funktion `merge()`. Wir werden aber gleich die Funktionen aus dem  
1325 Paket `dplyr` verwenden.

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

1326 ##   id bhd  art gebiet
1327 ## 1   1   20 <NA>   <NA>
1328 ## 2   2   31    Ta      A
1329 ## 3   3   74    Bu      B

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

1330 ##   id bhd art gebiet
1331 ## 1   2   31   Ta      A
1332 ## 2   3   74   Bu      B
1333 ## 3   4   NA   Bu      B
```

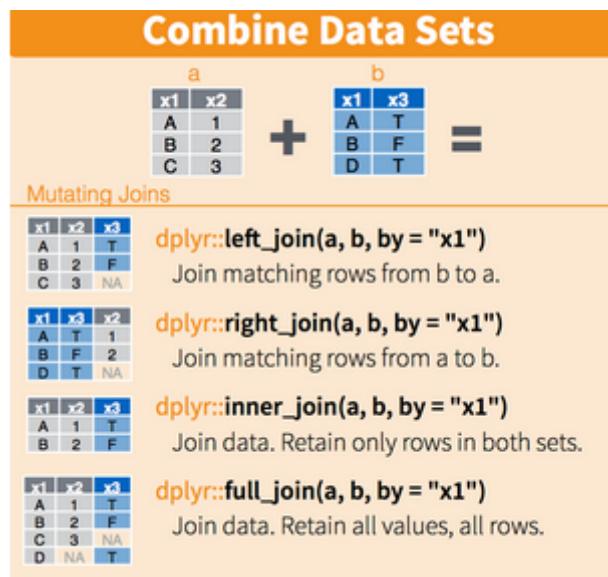


Abbildung 8: Joins (Quelle Rstudio)

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

```
1334 ##   id bhd art gebiet
1335 ## 1  2  31  Ta      A
1336 ## 2  3  74  Bu      B
full_join(aufnahmen, metadaten, by = "id")
```

```
1337 ##   id bhd  art gebiet
1338 ## 1  1  20 <NA>  <NA>
1339 ## 2  2  31  Ta      A
1340 ## 3  3  74  Bu      B
1341 ## 4  4  NA  Bu      B
```

1342 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"))
```

```
1343 ##   id bhd  art gebiet
1344 ## 1  1  20 <NA>  <NA>
1345 ## 2  2  31  Ta      A
1346 ## 3  3  74  Bu      B
```

1347

1348 **Aufgabe 27: Verbinden von Daten**

---

- 1349
- Lesen Sie die Datensätze `daten/bhd_2.txt` und `daten/bhd_2_meta.txt` ein.
  - Stellen Sie sicher, dass es für den `bhd` keine fehlenden Werte gibt (entfernen sie entsprechende Zeilen)
  - Fügen Sie zu den Metadaten (gespeichert in `bhd_2_meta`) die Anzahl Bäume und den mittleren `bhd` hinzu pro Gebiet.

1354 **9.5 ‘long’ and ‘wide’ Datenformate**
1355 Unter anderem Wickham (2014) empfieilt das Prinzip von *tidy Data*. Nach diesem Prinzip sollten Daten wie  
1356 folgt organisiert sein:

- 1357
- Jede Zeile ist ein Merkmalsträger/Subjekt/Objekt (z. B. eine Person oder ein Baum).
  - Jede Spalte ist eine Variable (ein Merkmal), die den Merkmalsträger beschreibt.
  - In jeder Zelle ist genau ein Wert (Merkmalausprägung), nämlich der Wert, der Variable für den Merkmalsträger.

1361 Zum Beispiel enthalten Spaltennamen oft Informationen, die eigentlich in einer Variable gespeichert werden  
1362 sollten. Folgendes Beispiel gibt die BHD Messung von 3 Bäumen in 3 Jahren wieder. Der Vorteil von *tidy*  
1363 Daten ist, dass viele Funktionen diese Form erwarten. Somit müssen sie Ihre Daten nur ein Mal organisieren  
1364 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)
)
```

1365 Der `tibble` aus dem Paket `tibble` ist ein moderner Data Frame, der sich in seinen Funktionen in das `tidy`  
1366 Universum einfügt. Wir können auch einfach das Paket `tidyverse` laden, um alle Pakete, die Teil des `tidy`  
1367 Universums sind gleichzeitig zu laden. Der `tibble` beinhaltet alle Funktionen, die der Data Frame auch  
1368 beinhaltet und kann auch für genau die gleichen Zwecke verwendet werden. Der `tibble` hat u. A. eine  
1369 modernere Darstellung im Konsolenoutput.

1370 Diese Daten sind jetzt im `wide`-Format gespeichert und folglich nicht `tidy`, weil Information über die Daten  
1371 (nämlich das Jahr der Aufnahme in den Spaltennamen gespeichert sind). Besser wäre eine Struktur mit  
1372 nur drei Spalten: `id`, `jahr` und `bhd`. Um die Daten in so eine Struktur zu bringen, gibt es die Funktion  
1373 `pivot_longer()` aus dem Paket `tidyR`, die ebenfalls Teil des `tidyverse` ist.

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

1374 ## # A tibble: 9 x 3
1375 ##       id name   value
```

```

1376 ##   <int> <chr>    <dbl>
1377 ## 1     1 bhd2015    30
1378 ## 2     1 bhd2026    31
1379 ## 3     1 bhd2017    34
1380 ## 4     2 bhd2015    31
1381 ## 5     2 bhd2026    31
1382 ## 6     2 bhd2017    32
1383 ## 7     3 bhd2015    32
1384 ## 8     3 bhd2026    33
1385 ## 9     3 bhd2017    33

```

1386 Wenn wir die Spalten für die Variable und den Wert gleich sinnvoll benennen möchten, können wir das über  
1387 die Argumente `names_to` und `values_to` machen.

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

```

1388 ## # A tibble: 9 x 3
1389 ##       id jahr     bhd
1390 ##   <int> <chr>    <dbl>
1391 ## 1     1 bhd2015    30
1392 ## 2     1 bhd2026    31
1393 ## 3     1 bhd2017    34
1394 ## 4     2 bhd2015    31
1395 ## 5     2 bhd2026    31
1396 ## 6     2 bhd2017    32
1397 ## 7     3 bhd2015    32
1398 ## 8     3 bhd2026    33
1399 ## 9     3 bhd2017    33

```

1400 Analog zu der Funktion `pivot_longer()` gibt es auch die Funktion `pivot_wider()`, um vom Daten vom  
1401 `long`-Format ins `wide`-Format zu transformieren.

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

```

1402 ## # A tibble: 3 x 4
1403 ##       id bhd2015 bhd2026 bhd2017
1404 ##   <int>    <dbl>    <dbl>    <dbl>
1405 ## 1     1        30        31        34
1406 ## 2     2        31        31        32
1407 ## 3     3        32        33        33

```

1408

---

1409 **Aufgabe 28: Zeitliche Verlauf von BHDS**

---

1410

1411 In der Datei `bhd_3.csv` befinden sich gemessene BHDS (in cm) von unterschiedlichen Bäumen zu unter-  
 1412 schiedlichen Zeitpunkten. Erstellen Sie ein Liniendiagramm, das den zeitlichen (x-Achse) Verlauf der BHDS  
 1413 (y-Achse) für die unterschiedlichen Bäume darstellt.

1414 **9.6 Auswählen von Variablen**

1415 Sobald die Datensätze etwas umfangreicher werden (d. h., es gibt mehrere Spalten in einem `data.frame`),  
 1416 können innerhalb vieler `dplyr`-Funktionen spezielle Funktionen verwendet werden, um Variablen auszuwählen.  
 1417 Wenn die genaue Position der Spalten bekannt ist, kann man mit dem `:`-Operator und der Position Spalten  
 1418 auswählen:

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

1419 ## Sepal.Length Sepal.Width Petal.Length  
 1420 ## 1 5.1 3.5 1.4  
 1421 ## 2 4.9 3.0 1.4  
 1422 ## 3 4.7 3.2 1.3

1423 Diese Vorgehensweise kann gehährlich sein, da sich manchmal Spalten verschieben und sich somit die  
 1424 Positionen ändern. Es besser Spalten immer explizit mit Namen statt mit Nummern anzusprechen.

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

1425 ## Sepal.Length Sepal.Width Petal.Length  
 1426 ## 1 5.1 3.5 1.4  
 1427 ## 2 4.9 3.0 1.4  
 1428 ## 3 4.7 3.2 1.3

1429 `select()` erlaubt es, auch hier den `:`-Operator zu verwenden:

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

1430 ## Sepal.Length Sepal.Width Petal.Length  
 1431 ## 1 5.1 3.5 1.4  
 1432 ## 2 4.9 3.0 1.4  
 1433 ## 3 4.7 3.2 1.3

1434 Es gibt auch einige spezielle Funktionen, um Spalten innerhalb eines `select()`-Aufrufs auszuführen:

- 1435 • `starts_with()`: Hier kann man ein Muster angeben, mit dem ein Text anfangen muss.
- 1436 • `ends_with()`: Diese Funktion ist analog zu `starts_with()`, jetzt wird aber am Ende des Spaltennamens  
 1437 nach dem Muster gesucht.
- 1438 • `contains()`: Hier kann ein Muster übergeben werden, das irgendwo im Spaltennamen sein muss.
- 1439 • `everything()`: Mit dieser Funktion werden alle Spalten ausgewählt.

1440 • `last_col()`: Mit dieser Funktion wird nur die letzte Spalte ausgewählt (dass ist die Spalte, die ganz  
1441 rechts ist).

1442 Sämtliche Auswahlen können mit - umgekehrt werden.

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

1443 ## Sepal.Length Sepal.Width

1444 ## 1 5.1 3.5

1445 ## 2 4.9 3.0

1446 ## 3 4.7 3.2

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

1447 ## Petal.Length Petal.Width Species

1448 ## 1 1.4 0.2 setosa

1449 ## 2 1.4 0.2 setosa

1450 ## 3 1.3 0.2 setosa

1451 `select()` bietet auch noch die Möglichkeit, Spaltennamen zu ändern.

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

1452 ## sep\_width

1453 ## 1 3.5

1454 ## 2 3.0

1455 ## 3 3.2

1456

### 1457 Aufgabe 29: Auswählen von Spalten

---

1459 In der Datei `messungen_1.csv` sind Messungen von zwei Sensoren enthalten für die ersten vier Monate eines  
1460 Jahres. Führen Sie folgende Abfragen durch:

1461 1. Wählen Sie alle Messungen für Januar aus.

1462 2. Wählen Sie alle Messungen für Januar und März aus.

## 1463 9.7 Einzelne Beobachtungen abfragen (`slice()`)

1464 Mit dem Befehl `slice()` kann man einzelne Beobachtungen (= Zeilen) aus einem `data.frame` abfragen.

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

1465 ## Sepal.Length Sepal.Width Petal.Length Petal.Width Species

1466 ## 1 5.1 3.5 1.4 0.2 setosa

1467 ## 2 4.4 2.9 1.4 0.2 setosa

1468 ## 3 5.1 3.5 1.4 0.3 setosa

1469 Davon gibt es drei nützliche Varianten: 1) `slice_head()` und `slice_tail()`; 2) `slice_max()` und  
 1470 `slice_min()`; 3) `slice_random()`.

1471 `slice_head()` und `slice_tail()` sind analog zu `head()` und `tail()`, aber mit dem entscheidenden Unter-  
 1472 schied, dass Gruppierungen berücksichtigt werden. Wenn keine Gruppierung in den Daten vorhanden ist, gibt  
 1473 es keinen Unterschied.

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

```
1474 ##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
1475 ## 1       5.1      3.5      1.4      0.2 setosa
1476 ## 2       4.9      3.0      1.4      0.2 setosa
```

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

```
1477 ##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
1478 ## 1       5.1      3.5      1.4      0.2 setosa
1479 ## 2       4.9      3.0      1.4      0.2 setosa
```

1480 Sobald jedoch eine gruppierende Variable eingeführt wird, gibt `slice_head()` die ersten n Beobachtungen  
 1481 für jede Gruppe zurück und `head()` für den gesamten Datensatz.

```
# base head
```

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

```
1482 ## # A tibble: 2 x 5
1483 ## # Groups:   Species [1]
1484 ##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
1485 ##       <dbl>     <dbl>     <dbl>     <dbl> <fct>
1486 ## 1       5.1      3.5      1.4      0.2 setosa
1487 ## 2       4.9      3       1.4      0.2 setosa
```

```
# dplyr slice_head
```

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

```
1488 ## # A tibble: 6 x 5
1489 ## # Groups:   Species [3]
1490 ##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
1491 ##       <dbl>     <dbl>     <dbl>     <dbl> <fct>
1492 ## 1       5.1      3.5      1.4      0.2 setosa
1493 ## 2       4.9      3       1.4      0.2 setosa
1494 ## 3       7       3.2      4.7      1.4 versicolor
1495 ## 4       6.4      3.2      4.5      1.5 versicolor
1496 ## 5       6.3      3.3       6      2.5 virginica
1497 ## 6       5.8      2.7      5.1      1.9 virginica
```

1498 `slice_tail()` funktioniert analog zu `slice_head()` mit dem einzigen Unterschied, dass nicht die ersten n

1499 Zeilen zurück gegeben werden sondern die letzten **n** Zeilen.  
 1500 `slice_max()` und `slice_min()` geben die Beobachtung mit dem maximalen bzw. minimalen Wert einer  
 1501 Variable zurück. Auch hier werden Gruppen berücksichtigt.

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

1502 ## Sepal.Length Sepal.Width Petal.Length Petal.Width Species  
 1503 ## 1 7.9 3.8 6.4 2 virginica

1504 Und mit Gruppen:

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

  slice_max(Sepal.Length)
```

1505 ## # A tibble: 3 x 5  
 1506 ## # Groups: Species [3]  
 1507 ## Sepal.Length Sepal.Width Petal.Length Petal.Width Species  
 1508 ## <dbl> <dbl> <dbl> <dbl> <fct>  
 1509 ## 1 5.8 4 1.2 0.2 setosa  
 1510 ## 2 7 3.2 4.7 1.4 versicolor  
 1511 ## 3 7.9 3.8 6.4 2 virginica

1512 `slice_min()` funktioniert genau gleich, nur dass die Beobachtung (=Zeile) mit dem kleinsten Wert einer  
 1513 Variable zurück gegeben wird.

1514 Die Funktion `slice_sample()` erlaubt es zufällige Beobachtungen zu ziehen. Dabei kann über das Argument **n**  
 1515 die Anzahl an Beobachtungen angegeben werden oder über das Argument **prop** der Anteil an Beobachtungen.

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

1516 ## Sepal.Length Sepal.Width Petal.Length Petal.Width Species  
 1517 ## 1 6.5 2.8 4.6 1.5 versicolor  
 1518 ## 2 6.3 3.3 4.7 1.6 versicolor  
 1519 ## 3 7.2 3.2 6.0 1.8 virginica  
 1520 ## 4 4.9 3.6 1.4 0.1 setosa  
 1521 ## 5 6.0 2.7 5.1 1.6 versicolor

1522 Das Ergebnis ist bei jedem von Ihnen anders, da es sich um eine zufällige Ziehung handelt. Wenn Sie diese  
 1523 Ergebnisse wiederholen möchten, können Sie über `set.seed()` die zufällige Ziehung reproduzierbar machen.

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

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

1524 ## Sepal.Length Sepal.Width Petal.Length Petal.Width Species  
 1525 ## 1 4.3 3.0 1.1 0.1 setosa  
 1526 ## 2 5.0 3.3 1.4 0.2 setosa  
 1527 ## 3 7.7 3.8 6.7 2.2 virginica  
 1528 ## 4 4.4 3.2 1.3 0.2 setosa  
 1529 ## 5 5.9 3.0 5.1 1.8 virginica

1530 Wenn beispielsweise 5% der Beobachtungen gezogen werden sollen, kann dies so gemacht werden:

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

```
1531 ##   Sepal.Length Sepal.Width Petal.Length Petal.Width   Species
1532 ## 1         7.7       3.8        6.7       2.2 virginica
1533 ## 2         5.5       2.5        4.0       1.3 versicolor
1534 ## 3         5.5       2.6        4.4       1.2 versicolor
1535 ## 4         6.5       3.0        5.2       2.0 virginica
1536 ## 5         6.1       3.0        4.6       1.4 versicolor
1537 ## 6         6.3       3.4        5.6       2.4 virginica
1538 ## 7         5.1       2.5        3.0       1.1 versicolor

1539 slice_sample() berücksichtigt ebenfalls Gruppen. Mit den Argumenten replace und weight_by dann die
1540 Zufallsziehung genauer spezifiziert werden. replace sagt, ob eine gezogenen Beobachtung wieder zurück gelegt
1541 wird oder nicht. Mit dem Argument weight_by können optional gewichtete für jede Beobachtung vergeben
1542 werden.
```

1543

#### 1544 Aufgabe 30: Daten beschreiben

---

1546 Verwenden Sie den Datensatz `bhd_1.txt` und finden Sie für jedes Gebiet und Art die Beobachtung mit
1547 kleinsten BHD.

## 1548 9.8 Spalten trennen

1549 Ein gut geplanter Datensatz besteht aus Beobachtungen (Zeilen), Variablen (Spalten) und in jeder Zelle ist
1550 immer ein **genau** ein Wert gespeichert. Leider gibt es oft Datensätze, bei denen mehr als ein Wert pro Zelle
1551 gespeichert wurde. Die Funktion `separate()` kann helfen solche Daten zu trennen.

1552 Wir verwenden einen erfunden Datensatz zu Beobachtungen von Tieren und einer geschätzten Distanz zu
1553 diesen Tieren.

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

1554 In der Spalte `beobachtung` sind zwei Informationen gespeichert: Die Distanz zur Beobachtung und die Art.
1555 Das ist ungünstig, weil wir so weder nach Tierart noch nach distanz filtern können (nicht *tidy*). Mit der
1556 Funktion `separate()`, können wir Beobachtungen einer Spalte in mehrere Spalten trennen. Dafür muss der
1557 Spaltennamen (Argument `col`), die neuen Spaltennamen (Argument `into`) und das Trennzeichen (Argument
1558 `sep`) angegeben werden.

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

1559 ## # A tibble: 4 x 3

```
1560 ##      id Distanz Art
1561 ## <int> <chr>  <chr>
1562 ## 1      1 10m    " Reh"
1563 ## 2      2 100m   " Reh"
1564 ## 3      3 20m    " Fuchs"
1565 ## 4      4 40     "Reh"
1566 Nach dem Aufruf von separate() gibt es zwei neue Spalten (Distanz und Art), die die alte Spalte
1567 beobachtung ersetzen.
```

1568

---

1569 **Aufgabe 31: Aufräumen**

---

1571 Verwenden Sie den folgenden Datensatz und bringen Sie ihn in eine Form, die sicherstellt dass

1572 • jede Zelle genau einen Wert enthält.  
1573 • jede Zeile eine Beobachtung ist.  
1574 • 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")
)
```

## 1575 10 Arbeiten mit Text

1576 Bis jetzt haben wir fast ausschließlich mit Zahlen oder Abbildungen gearbeitet. R bietet aber auch viele  
 1577 Werkzeuge, um mit Text zu arbeiten. Wir wollen hier ein paar Funktionen dafür vorstellen. Als erstes sollte  
 1578 nochmals klar gestellt werden, was eigentlich ein Text ist. In R ist alles, das innerhalb von doppelten ("") oder  
 1579 einfachen ('') Anführungszeichen geschrieben ist, Text.

1580 Anbei einige Beispiele:

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

1581 Wichtig ist hier zu sehen, dass z nicht als Zahl sondern, als Text interpretiert wird (siehe Datentypen).

```
z + 1
```

1582 ## Error in z + 1: nicht-numerisches Argument für binären Operator

1583 Wenn man sicher ist, dass es sich bei einem Textobjekt um eine Zahl handelt, kann man dies mit der Funktion  
 1584 `as.numeric()` in eine Zahl umwandeln.

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

1585 ## [1] 31

1586 Aber mit a führt dies wieder zu einem NA-Wert, da a nicht in eine Zahl umgewandelt werden kann.

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

1587 ## Warning: NAs durch Umwandlung erzeugt

1588 ## [1] NA

### 1589 10.1 Arbeiten mit Text

1590 Wir wollen erst einmal drei Funktionen besprechen, die es erlauben mit Text zu arbeiten. Die Funktion  
 1591 `nchar()`<sup>11</sup> gibt an wie viele Zeichen ein Text hat. Also z.B.

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

1592 ## [1] 5

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

1593 ## [1] 2

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

1594 ## [1] 20

1595 Die Funktion `paste()` erlaubt es verschiedene Variablen mit Text zu verbinden. Wenn wir z. B. die Variablen  
 1596 `vorname <- "Eva"` und `name <- "Meier"` haben und wir wollen eine neue Variable `full_name <- Eva`

---

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

1597 Meier" erzeugen, dann kann das mit der Funktion `paste()` gemacht werden.

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

1598 ## [1] "Eva Meier"

1599 Die Funktion `paste()` hat das Argument `sep`, das auf ein Leerzeichen ( ) gesetzt ist, aber auch anders sein  
1600 kann und das Trennzeichen definiert.

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

1601 ## [1] "Eva, Meier"

1602 Die Funktion `substr()` erlaubt es am Anfang oder Ende eines Wortes etwas abzuschneiden. Dabei muss  
1603 immer die Anfangs- (`start`) und Endposition (`stop`) angegeben werden.

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

1604 ## [1] "Hal"

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

1605 ## [1] "allo"

1606

### 1607 Aufgabe 32: Arbeiten mit Text 1

---

1609 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")
```

1610 1. Aus wie vielen Buchstaben besteht jedes Wort?

1611 2. Finden Sie das längste Wort.

1612 3. Wie viel Prozent der Wörter fangen mit einem S an?

1613 4. Fügen Sie jedem Wort seine Position im Vektor hinzu. Beispielsweise soll aus `Vogel` "2. Vogel" werden

1614 usw.

## 1615 10.2 Finden von Textmustern

1616 Mit der Funktion `grep()` können Muster in einem Text gefunden werden. Wenn wir beispielsweise folgenden  
1617 Vektor mit Textelementen haben.

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

1618 Und wir wollen alle Straßennamen die ein `weg` haben abfragen, dann können wir folgenden Befehl ausführen:

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

1619 `## [1] 2`

1620 Im zweiten Element von `txt` kommen die Zeichen `Weg` vor. Beachte, in der Standardeinstellung wird zwischen  
1621 Groß- und Kleinschreibung unterschieden. Dies kann mit dem Argument `ignore.case = TRUE` angepasst  
1622 werden.

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

1623 `## [1] 1 2`

1624 Mit der Funktion `sub` können Zeichen innerhalb einer Zeichenkette ersetzt werden.

1625 So ersetzt der folgende Ausdruck `ae` mit `ä`.

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

1626 `## [1] "Friedländer Weg"`

1627 Wenn allerdings das zu ersetzende Zeichen mehr als einmal vorkommt und beide Instanzen ersetzt werden  
1628 sollen braucht man die Funktion `gsub`.

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

1629 `## [1] "Friedländer Weg und Reinhäuser Landstrasse sind zwei Strassen in Goettingen."`

1630 Mit `sub()` wird nur das erste `ae` ersetzt, während `gsub()` alle `ae` mit einem `ä` ersetzt.

```
gsub("ae", "ä", txt)
```

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

1632 Oft ist der genaue Ausdruck den man finden möchte jedoch Variabel. Beispielsweise möchte man alle Wörter  
1633 mit einem Umlaut oder Zahlen finden möchte, kann man das oft abkürzen. Dafür gibt es reguläre Ausdrücke.  
1634 Wir werden hier nur ein paar beispielhafte Anwendungen besprechen.

1635 Sowohl in den Funktionen `grep()` als auch `(g)sub()` kann mit anstatt dem Muster (immer das erste  
1636 Argument) aus ein regulärer Ausdruck angegeben werden. Mit `[1-9]` sind alle Zahlen von 1 bis 9 gemeint.

1637 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)
```

1638 `## [1] 1 3`

1639 Damit lässt sich auch das Problem mit groß- und kleingeschriebenen Wörtern lösen.

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

```
1640 ## [1] 1 2
```

1641

---

**Aufgabe 33: Arbeiten mit Text 2**

---

1644 Verwenden Sie den folgenden Vektor:

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

1645 1. In wie vielen Wörtern kommt der Doppellaut au vor?

1646 2. Ersetzen Sie in allen Wörtern alle au mit \_ \_.

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

```
1647 ## [1] 5 6 13
```

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

```
1648 ## [1] "Versicherung" "Methoden"      "Fluss"        "Rudel"        "B_ _m"
```

```
1649 ## [6] "H_ _s"          "Foto"         "Auffahrt"     "Auto"         "Handy"
```

```
1650 ## [11] "Teller"        "Kalender"     "Aufb_ _"
```

## 1651 11 Arbeiten mit Zeit

1652 Für den Computer bzw. R ist ein Datum erst einmal nichts anderes als ein Text. Für uns ist es sofort  
 1653 klar, dass der "13.2.2021" der 13. Februar 2021 ist, für den Computer zunächst nicht. Wir müssen R also  
 1654 irgendwie sagen, dass die 13 der Tag ist, die 2 der Monat und 2021 das Jahr. Dass der Computer die einzelnen  
 1655 Komponenten erkennt, und in einen Datentyp eigens für Zeitformate überführt, nennt man *parsen*<sup>12</sup>. Durch  
 1656 das *parsen* wird die Variable in den Datentyp **Date** überführt. Das Arbeiten mit Datum und Zeit kann  
 1657 kann anfangs sehr mühsam sein und viele Zeit-spezifischen Datenoperationen lassen sich auch mit den  
 1658 Basis-Datentypen durchführen. Sobald man einige Grundfertigkeiten erworben hat, stellt man jedoch fest,  
 1659 dass die Arbeit mit dem Zeitformat-Datentyp schneller und effizienter funktioniert. Starten Sie am besten  
 1660 gleich mit "echten" Zeit-Datentypen und versuchen Sie nicht, sich irgendwie mit Text und numerischen  
 1661 Datentypen selbst etwas zu basteln. Der erste Schritt ist immer ein Datum zu *parsen*. Wir verwenden dafür  
 1662 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)
```

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```

dmy("20.1.2020")

1678 ## [1] "2020-01-20"
1679 Jetzt stellt sich die Frage, was der Vorteil ist, wenn R ein Datum parst.
d <- dmy("20.1.2020")

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

1681 ## [1] 20
month(d)

1682 ## [1] 1
year(d)

1683 ## [1] 2020
1684 Oder auch Zeiteinheiten hinzufügen oder abziehen.
d + days(10)

1685 ## [1] "2020-01-30"
d - years(20)

1686 ## [1] "2000-01-20"
d + hours(25)

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

```

1688

---

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

---

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

**11.1 Arbeiten mit Zeitintervallen**

1696 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)
```

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

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

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

1700 die Länge des Intervalls berechnen

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

1701 ## [1] 31536000

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

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

1703 ## [1] TRUE

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

1704 ## [1] FALSE

1705 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)
```

1706 ## # A tibble: 1 x 1

1707 ## a

1708 ## <date>

1709 ## 1 2020-07-01

1710 `%within%` funktioniert genauso mit Vekotren oder mit mehreren Intervallen. Wir könnten also zwei Intervalle  
1711 definieren (z.B. Ostern und Pfingsten).

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

1712 Und Überprüfen welche Termine in eines der zwei Intervalle fallen.

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

```
# Ostern
```

```
termine %within% ostern
```

1713 ## [1] FALSE TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE

```
# Pfingsten
termine %within% pfingsten

1714 ## [1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE TRUE FALSE FALSE
1715 Der Zeit-Datentyp wird auch oft benutzt, um den Zeitbedarf von längeren Berechnungen zu messen.

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

1716 ## [1] 0.4999484
t2 <- now()
int_length(interval(t1, t2))

1717 ## [1] 0.594928
```

## 1718 11.2 Formatieren von Zeit

1719 Für die Ausgabe in Berichten oder Grafiken soll das Datum oft in einer speziellen Form dargestellt werden.  
 1720 Die Funktion `format()` bietet Möglichkeiten ein Datumsobjekt zurück in Text umzuwandeln.

1721 Ein Beispiel wäre `ymd("2021-2-10")` als 10.2.21 auszugeben.

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

1722 ## [1] "21.02.21"
1723 Dabei handelt sich bei %d.%m.%y um Abkürzungen für die unterschiedlichen Komponenten eines Datumobjekts.
1724 Siehe dazu die Hilfeseite von strptime (help(strptime)).
```

1725

## 1726 Aufgabe 35: Arbeiten mit Intervallen

---

1727 Wie viele Einträge aus dem Vektor `v1` befinden sich in einem Intervall, das zwischen dem 1.3.2021 und dem 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"
)
```

## 1730 11.3 Zeitreihen

1731 In der Forstbranche haben wir oft mit Zeitreihen zu tun. Zeitreihen sind Variablen, für die in zeitlichen  
 1732 Intervallen Daten vorliegen. In der Regel monatlich, vierteljährig oder jährlich, wobei die Intervalle zwischen  
 1733 den Messungen bei Zeitreihen immer gleich lang sind. Wiederholungsmessungen von Forsteinventuren (Forstein-  
 1734 richtungen, Betriebsinventuren, die meisten forstl. Experimente, die BWI, ...) sind also in der Regel keine

1735 Zeitreihen in engeren Sinne. Turnusmäßig vermessene Versuchsflächen, wie sie z. B. die Versuchsanstalten  
 1736 unterhalten oder jährlich gemeldete Holzpreise jedoch schon.

1737 Zeitreihen unterscheiden sich nicht nur technisch, sondern auch inhaltlich fundamental von den uns schon  
 1738 bekannten Daten. Statistisch gesehen haben Zeitreihen Besonderheiten gegenüber anderen Variablen, da  
 1739 Sie von Ihrer eigenen Vergangenheit abhängen (autokorriert sind) und auch die Abhängigkeit anderer  
 1740 Variablen in der Regel nur sinnvoll ist, wenn deren Vergangenheit mitberücksichtigt wird (cross-correlation).  
 1741 Konventionelle Statistik ist oft nicht möglich, um Zeitreihen zu analysieren. Selbst ein ordinärer arithmetischer  
 1742 Mittelwert ist schon nicht mehr geeignet, um Zeitreihen statistisch zu beschreiben. Angefangen mit der  
 1743 Datendarstellung gibt es in R deshalb spezifische Zeitreihen-Funktionen. Aus diesem Grund sollten Sie  
 1744 Zeitreihen als solche speichern. Viele Funktionen erkennen den Datentyp und führen entsprechend spezifische  
 1745 Zeitreihen-Operationen durch, wenn ihnen Daten vom Typ "Zeitreihe" übergeben werden. Laden wir z. B.  
 1746 die Holzpreise für Fichte 2b (das sog. Leitsortiment, Fichenholz mit einem Mittendurchmesser von 20 bis 25  
 1747 cm), das Holzaufkommen dieses Sortiments (Einschlagsvolumen) und die Preise für Nadelholz vom  
 1748 statistischen Bundesamt<sup>14</sup>. Wir laden die Daten zunächst als csv ein:

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

1749 Diese 3 Zeitreihen bilden zusammen ein klassisches Marktmodell mit dem Preis eines homogenen Gutes  
 1750 (Leitsortimentspreis), dem Angebot (Holzeinschlag) und der Nachfrage (Schnittholzpreis). Mit der Funktion  
 1751 **ts** werden die Daten in ein Zeitreihenobjekt überführt (*pasrse*). Die Spalte mit den Jahren ist dann nicht mehr  
 1752 nötig, da die Informationen zur Zeit keine eigene Variable mehr sind, sondern als sog. Metainformationen in  
 1753 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])
```

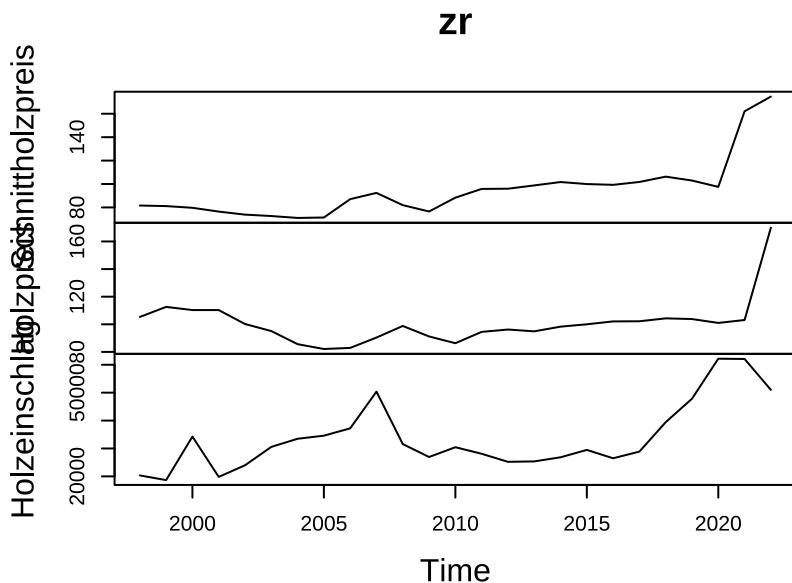
**typeof(zr)** # Zeitreihen sind sog. S3 Objekte. Das heißt, Zeitreihen gehören nicht zu den

1754 ## [1] "double"  
 # Basis Datentypen (siehe Kapitel Variablen, Funktionen und Datentypen),  
 # sondern sind eine Unterkategorie des Datentyps "Liste".

1755 Die wichtigsten Argumente sind - **data** Vektor oder Matrix, der nur die Daten enthält - **start** Startzeitpunkt -  
 1756 **frequency** Anzahl der Beobachtungen pro Zeiteinheit, also z. B. 12 bei monatlichen oder 4 bei quartalsweisen  
 1757 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

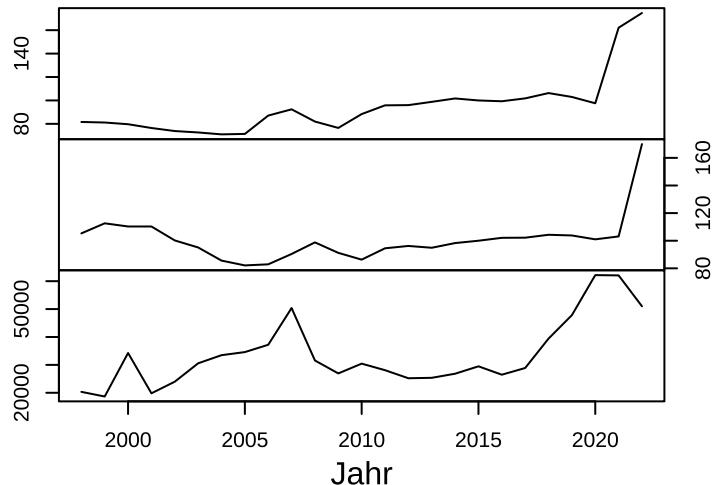


1758

1759 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")
```

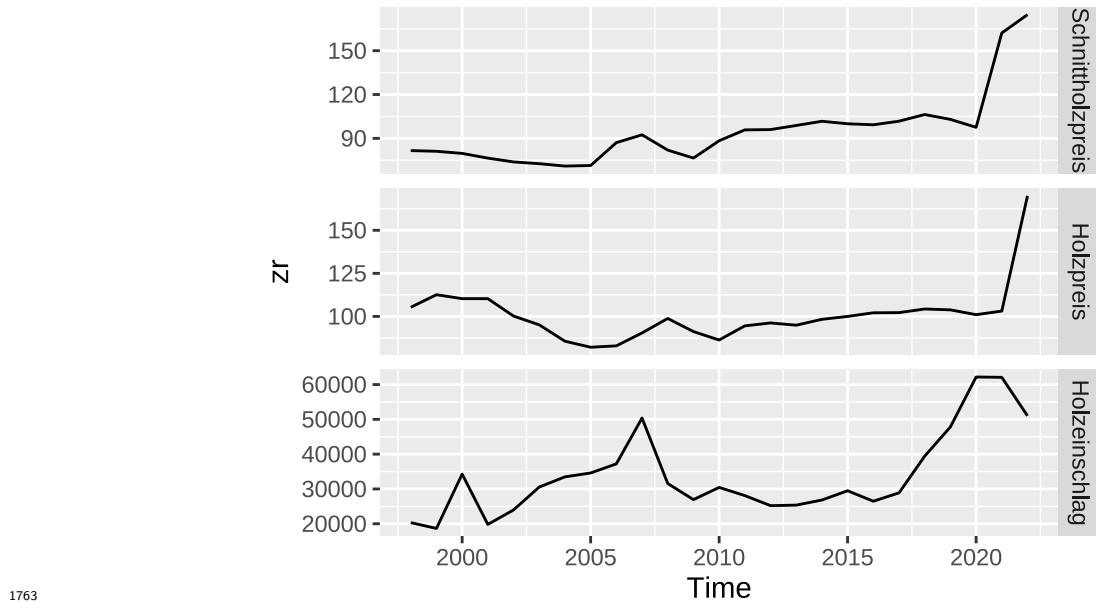
## Holzmarktentwicklung seit 1998



1760

1761 Beide Plot-Philosophiebn haben eine Zeitreihen-Funktion. Das Paket `ggfortify` ermöglicht automatisierte  
1762 Zeitreihenplots im `ggplot2` Stil. Damit ist auch das Problem der y-Achsenbeschriftungen gelöst.

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

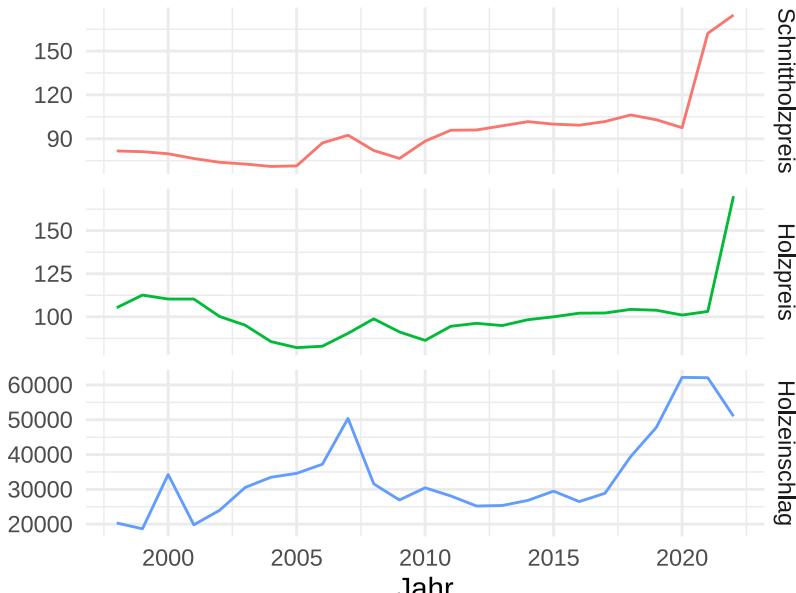


1763

1764 Wir können die Abbildung im `ggplot2` Stil ändern. Hier nur ein Minimalbeispiel zur Veranschaulichung.  
 1765 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()
```

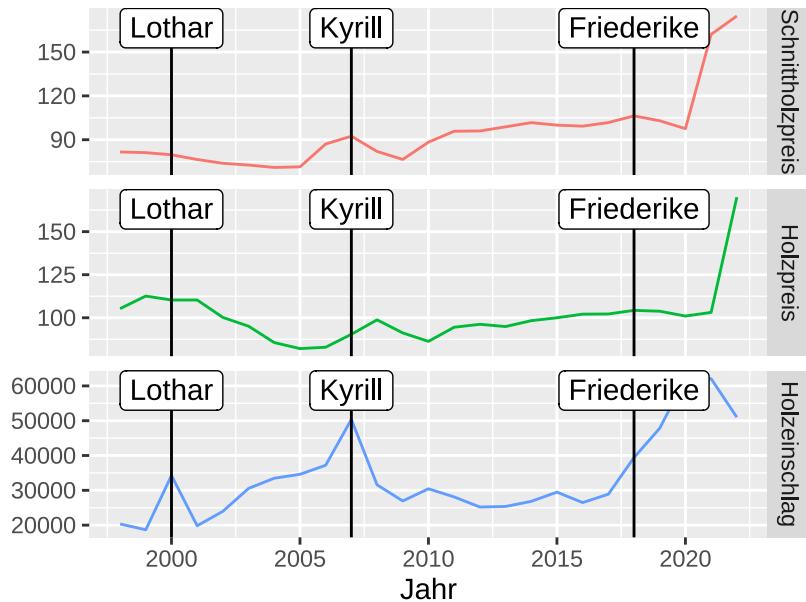


1766

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

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

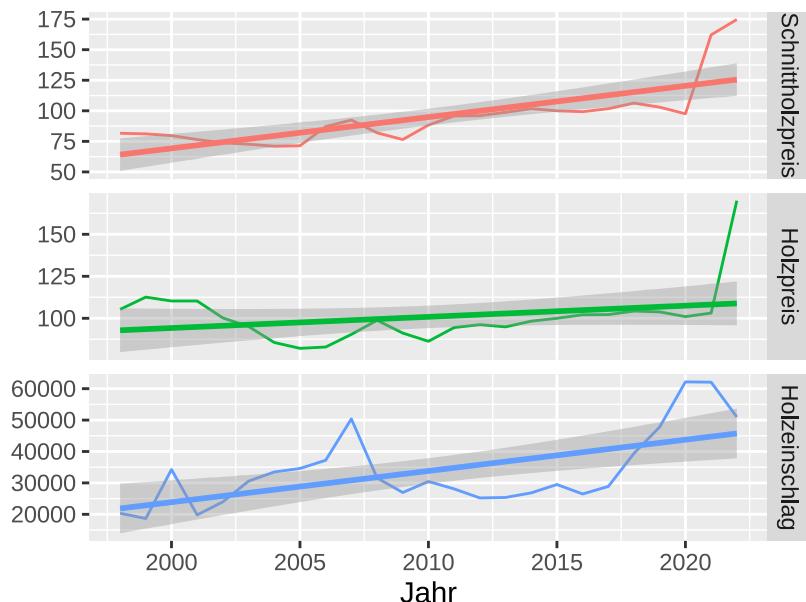
```
annotate(x = 2007, y = +Inf, label = "Kyrill", vjust = 1, geom = "label") +
  annotate(x = 2018, y = +Inf, label = "Friederike", vjust = 1, geom = "label")
```



1767

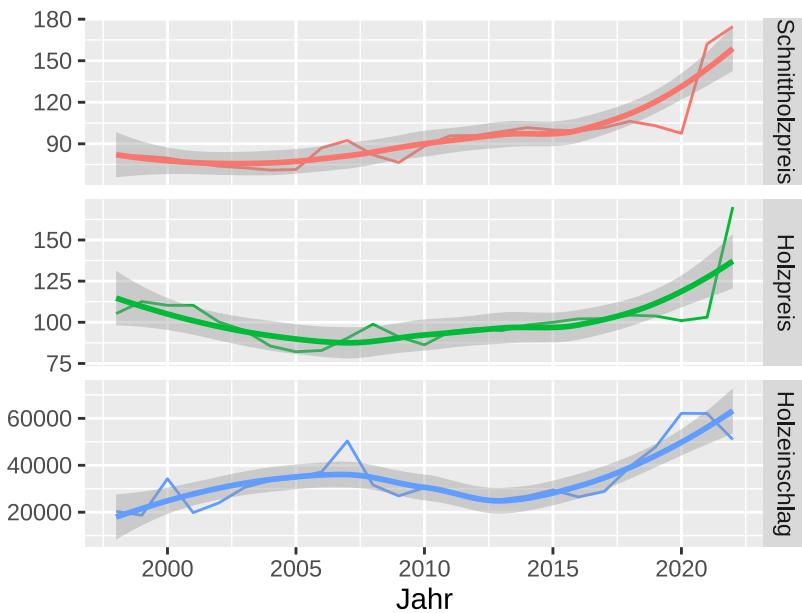
1768 Eine Trendlinie macht hier offensichtlich keinen Sinn. Die Trendlinie ist eine lineare Regression, also eine  
 1769 ordinäre Statistik, die wie eingangs erwähnt für Zeitreihen ungeeignet ist. Daher verwenden wir den sog.  
 1770 Loess-Glättter, um einen Trend in die Daten zu legen. Der Loess-Glättter ist ein sehr flexible Kurve. Wir sehen  
 1771 hier beispielsweise, dass der Leitholzpreis träge oder gar nicht auf das Angebot reagiert. Die Nachfrage jedoch  
 1772 zumindest in der einen Periode, in der sie stark steigt, den Holzpreis jedoch mit zeitlichem Verzug stark  
 1773 ansteigen lässt. Dieser visuelle Eindruck lässt sich durch spezifische Zeitreihen-Regressionen schätzen.

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



1774

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



1775

## 1776 12 Aufgaben Wiederholen (for-Schleifen)

1777 Um einfache Programme zu schreiben, müssen Sie den Ablauf der Programmcodes kontrollieren können.  
 1778 Kontrollieren bedeutet in diesem Zusammenhang, dass Codeabschnitte nur unter definierten Bedingungen ab-  
 1779 laufen. Sie programmieren also zwei Sachen. 1) den Code selbst und 2), die Bedingungen die erfüllt sein müssen,  
 1780 damit der Code ausgeführt wird. Der Code muss do generisch geschrieben sein, dass er komplett durchläuft,  
 1781 auch wenn unterschiedliche Dateneingaben gemacht werden. Diese Kontrollbedingungen ermöglichen es Ihnen  
 1782 generisch zu programmieren. Sie schreiben Ihren Code also nicht speziell maßgeschneidert für ein Problem,  
 1783 sondern so generell, dass er für mehrere Auswertungen funktioniert. Um dies zu gewährleisten, müssen Sie  
 1784 bestimmte Situationen vorhersehen und abfangen. Hierbei helfen Ihnen Kontrollstrukturen (**Control Flow**).  
 1785 Grundsätzlich gibt es **Control Flow** Funktionen zur Wiederholung von Codeblöcken (Schleifen) und logische  
 1786 Bedingungen (bedingte Anweisung).

### 1787 12.1 Schleifen

1788 Bis jetzt wurden alle Skripte einfach der Reihe nach abgearbeitet und zwischendurch bestimmte Programmteile,  
 1789 je nach Situation, selbstständig ausgeführt oder übersprungen. Mit einer Schleife kann man erreichen, dass  
 1790 eine Gruppe von Befehlen (der sog. Schleifenrumpf) mehrfach abgearbeitet wird, zum Beispiel wenn bestimmte  
 1791 Auswertungsschritte auf mehrere Datensätze oder Variablen angewendet oder Funktionen mit unterschiedlichen  
 1792 Parametern oder Startwerten aufgerufen werden sollen. Weitere Anwendungsmöglichkeiten sind iterative  
 1793 Algorithmen, in denen die Eingabewerte des aktuellen Iterationsschrittes von einem vorherigen abhängig sind.  
 1794 Besonders in Simulationen kommen Schleifen häufig zum Einsatz, da große Anzahlen von Wiederholungen  
 1795 benötigt werden.

1796 Man unterscheidet zwischen zwei Arten von Schleifen: Bei den `for()`-Schleifen steht die Anzahl der Wieder-  
 1797 holungen schon beim Eintritt in die Schleife fest, während die `while()`-Schleifen so lange ausgeführt werden,  
 1798 bis eine Bedingung nicht mehr wahr ist. Mit der Funktion `break` wird eine Schleife abgebrochen und die  
 1799 Programmausführung wird nach der Schleife fortgesetzt.

1800 Die wesentlichen Befehle sind

- 1801 • `for (i in X) {Code}`

1802 Wiederhole den Code im “Schleifenrumpf” für jedes Element aus X.

- 1803 • `while(Bedingung) {Code}`

1804 Wiederhole den Code im “Schleifenrumpf” so lange die logische Bedingung erfüllt ist.

- 1805 • `break()`

1806 Brich die Schleife ab. `break()` muss im Schleifenrumpf an der richtigen Stelle ausgeführt werden. Gute  
 1807 Praxis ist jedoch, die for oder while Bedingungen, dass kein `break()`nötig ist, da `break()` anfällig für  
 1808 Programmierfehler ist.

#### 1809 12.1.1 Wiederholen von Befehlen mit `for()`.

1810 Steht vor Beginn der Schleife fest wie viele Schleifendurchgänge benötigt werden, wenn zum Beispiel in einer  
 1811 Simulationen 99 Realisierungen erzeugt oder alle Elemente eines Vektors verarbeitet werden sollen, verwendet

1812 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)
}
```

1813 ## [1] 1

1814 ## [1] 2

1815 ## [1] 3

1816 Das **i** steht in diesem Beispiel für die Schleifen-Variable. Sie muss nicht **i** heißen, sondern kann jeden  
 1817 zulässigen Namen annehmen. Das **X** steht für einen existierenden Vektor oder eine existierende Liste bzw.  
 1818 einen Ausdruck, der ein solches Objekt liefert (der Objektname ist ebenfalls frei wählbar). **for** und **in** sind  
 1819 Schlüsselworte, sie müssen, ebenso wie die runden Klammern, vorhanden sein.

1820 Im ersten Durchgang erhält die Schleifen-Variable **i** den ersten Wert von **X** und der Schleifenrumpf wird  
 1821 mit diesem Wert ausgeführt. Die Variable **i** nimmt nacheinander so lange die Werte von **X** an, bis ihr alle  
 1822 Elemente zugewiesen wurden.

1823 Das folgende Beispiel wird zwar besser durch die entsprechende Vektoroperation gelöst, zeigt aber sehr  
 1824 deutlich die Arbeitsweise der **for**-Schleife.

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

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

1825 ## [1] 4

1826 ## [1] 9

1827 ## [1] 25

1828

### 1829 Aufgabe 36: Schleifen 1

---

1831 Verwenden Sie den Vektor **k** <- c(1, 3, 9, 12, 15) und schreiben Sie folgende **for**-Schleifen:

- 1832 1. Eine Schleife, die jedes Element aus **k** ausgibt.
- 1833 2. Eine Schleife, die zu jedem Element aus **k** 10 addiert und den neuen Wert ausgibt.
- 1834 3. Eine Schleife wie in 2), aber der neue Wert (**k** + 10) soll jetzt nicht mehr ausgegeben werden, sondern  
 1835 in **k10** gespeichert werden. Stellen Sie sicher, dass **k10** wieder von der Länge 5 ist.

1836

---

1837 Die Funktion **for()** ermöglicht es, einen Befehl beliebig oft zu wiederholen. Z.B. der folgende Ausdruck zieht  
 1838 10-Mal eine Stichprobe der Größe 1 aus dem Vektor **v**. Beachten Sie, dass die Schleifen-Variable **i** selbst gar

1839 nicht im Schleifenrumpf vorkommt. Das Ziel dieser Schleife ist nicht die Elemente des Vektors abzuarbeiten,  
 1840 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))
}
```

```
1841 ## [1] 3
1842 ## [1] 1
1843 ## [1] 3
1844 ## [1] 3
1845 ## [1] 2
1846 ## [1] 3
1847 ## [1] 2
1848 ## [1] 2
1849 ## [1] 1
1850 ## [1] 4
```

1851 Auf gleiche Weise kann man auch über die Variablen eines Dataframes iterieren<sup>15</sup>. Das folgende Beispiel hat  
 1852 zum Ziel die Funktionsweise von Schleifen zu verdeutlichen. Schleifen haben in R jedoch den Nachteil, dass sie  
 1853 sehr langsam operieren. Wenn es geht, sollte man Alternativen verwenden. Die Funktionsweise wiederholender  
 1854 Auswertungen wird jedoch mit `for`-Schleifen deutlicher. Aus diesem Grunde werden wir uns in diesem Kurs  
 1855 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.

1864

---

1865 **Aufgabe 37: for-Schleife**

---

18661867 Lesen Sie den Datensatz `bhd_1.txt` ein und verwenden Sie eine `for`-Schleife.

- 1868 • Ziehen Sie 500-Mal je 35 Beobachtungen für den BHD.  
1869 • Berechnen Sie jeweils den Mittelwert aus den 35 Werten.  
1870 • Speichern Sie die 500 Mittelwerte in einem neuen Vektor `mittel`.  
1871 • Wie ist die Verteilung dieser 500 Mittelwerte zu interpretieren?

1872 **12.1.2 Wiederholen von Befehlen mit `while()`**

1873 Die `while`-Schleifen finden Anwendung, wenn die Anzahl der zu durchlaufenden Wiederholungen vorher  
1874 nicht bekannt ist, wie zum Beispiel bei Iterationsverfahren, die bis zu einer gewissen Genauigkeit durchlaufen  
1875 werden sollen. Die `while`-Schleife besteht in R aus dem Schlüsselwort `while()` und einer Bedingung in runden  
1876 Klammern.

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

1877 Sie ist in der praktischen Programmierung nicht so relevant wie die `for`-Schleife. Sie sei deshalb hier nur  
1878 kurz erwähnt. Die Abbruchbedingung wird jedes Mal geprüft bevor der Schleifenrumpf durchlaufen wird. Die  
1879 Bedingung wird ausgewertet und wenn diese `TRUE` ist, wird der Schleifenrumpf ausgeführt und danach erneut  
1880 die Bedingung überprüft. Ist die Bedingung nicht erfüllt, wird der Schleifenrumpf nicht durchgeführt und die  
1881 Schleife beendet. Ist die Bedingung bereits vor Eintritt in die Schleife nicht erfüllt, wird die Schleife gar nicht  
1882 erst durchlaufen.

1883 Da `while`-Schleifen also so lange ausgeführt werden, bis die Bedingung nicht mehr erfüllt ist, kann eine  
1884 Endlosschleife entstehen. Dies kann passieren, wenn man nicht sauber programmiert hat. Wenn innerhalb der  
1885 Schleife nicht dafür gesorgt wird, dass die Bedingung irgendwann nicht mehr erfüllt wird, so läuft die Schleife  
1886 immer weiter. Steckt R in einer Schleife fest und reagiert nicht mehr, kann der Befehl unter Linux mit `Strg`+  
1887 `C` und unter Windows mit `Esc` abgebrochen werden. Alternativ können Sie auf das rote STOP Symbol  
1888 über der Konsole klicken.

1889 **12.2 Bedingte Ausführung von Codeblöcken**

1890 Innerhalb eines Skripts ist es mitunter notwendig je nach aktueller Situation unterschiedlich fortzufahren.  
1891 Die Situation wird mit einem logischen Ausdruck, einer sogenannten Bedingung, geprüft. Je nachdem, ob  
1892 die Bedingung wahr (`TRUE`) oder falsch (`FALSE`) ist, werden unterschiedliche Programmteile ausgeführt, der  
1893 jeweils andere Teil bleibt unberücksichtigt. Danach wird in jedem Fall die Programmausführung, mit den  
1894 auf die bedingte Anweisungen folgenden Anweisung, fortgesetzt. In R kann dies mit dem if-else-Konstrukt  
1895 realisiert werden, welches aus den Schlüsselwörtern `if()` und `else` sowie der in runde Klammern gefassten  
1896 Bedingung besteht.

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

1897 Im folgenden Beispiel sollen die bisher abstrakt beschriebenen Vorgänge praktische Anwendung finden. In  
 1898 dem Beispiel wird zunächst durch zufälliges Ziehen einer Zahl aus dem Bereich eins bis sechs ein Würfelwurf  
 1899 simuliert. Anschließend wird der Würfelwurf mit einem if-Ausdruck kommentiert. Ist die Bedingung, es wurde  
 1900 eine Sechs gewürfelt, erfüllt, wird der Code innerhalb der geschweiften Klammern ausgeführt, ansonsten wird  
 1901 der Klammerinhalt ignoriert.

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

1902 In den meisten Fällen ist es für R irrelevant, ob sich zwischen den verschiedenen Elementen Leerzeichen oder  
 1903 Zeilenumbrüche befinden. Bei dem else-Ausdruck dagegen wird ein Fehler erzeugt, wenn vor dem `else` nicht  
 1904 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.")
}
```

1905 ## [1] "Beim nächsten Wurf klappt's bestimmt."

1906

### 1907 Aufgabe 38: Bedingte Programmierung

---

- 1909 • Wenn keine 6 gewürfelt wurde, lassen Sie zusätzlich ausgeben welche Zahl gewürfelt wurde.  
 1910 • Wiederholen Sie den Würfelwurf 10 Mal.

## 1911 13 (R)markdown

### 1912 13.1 Markdown Grundlagen

1913 Die Idee von Markdown ist, einfach Text strukturieren zu können und das ganze ohne umfangreiche Programme  
 1914 zu erstellen. Im nächsten Abschnitt sehen wir dann, wie man Markdown und R-Code verbinden kann. Hier  
 1915 soll es jedoch erst einmal darum gehen, die einfachsten Bausteine von Markdown vorzustellen.

1916 Am Anfang jedes Dokuments kommt eine Präambel. Diese fängt mit --- an und hört auch wieder mit ---  
 1917 auf. Innerhalb der Präambel können dann Metainformationen über das Dokument festgelegt werden, dies  
 1918 beinhaltet im einfachsten Fall: Titel, Autor und Datum. Das würde dann so aussehen:

```
1919 ---
1920 title: "Ein Titel"
1921 author: "Der, der es geschrieben hat"
1922 date: "März 2021"
1923 ---
```

1924 Danach folgt strukturierter und formatierter Text. Verschiedene Hierarchieebenen von Überschriften können  
 1925 mit der Anzahl an # festgelegt werden. So ist eine Überschrift erster Ordnung # Kapitel eine Überschrift  
 1926 zweiter Ordnung ## Unterkapitel usw.

1927 Listen können erstellt werden, wenn man am Anfang jeder Zeile ein - oder 1. schreibt.

```
1928 - Erster Eintrag
1929 - Zweiter Eintrag
1930 - Dritter Eintrag
```

1931 wird zu

```
1932   • Erster Eintrag
1933   • Zweiter Eintrag
1934   • Dritter Eintrag
```

1935 Eine zentrale Idee von Markdown ist es Text einfach zu formatieren. Werden eine oder mehrere Wörter mit  
 1936 zwei Sternchen (\*\*) eingefasst wird dieser Text **fett** dargestellt. Also aus \*\*wichtig\*\* wird **wichtig**. Das  
 1937 gleiche funktioniert auch mit *kursiven* Text, jedoch muss man hier noch ein Sternchen verwenden, also aus  
 1938 \*kursiv\* wird *kursiv*. Soll ein text fett und kursiv sein, kann man drei Sternchen verwenden. Aus \*\*\*sehr  
 1939 wichtig\*\*\* wird dann **sehr wichtig**.

1940 Weitere Elemente wie Links oder Abbildungen können einfach eingebunden werden. Links werden mit [Link  
 1941 text](url) in den Text integriert. Beispielsweise ist eine gute Idee bei [stackoverflow](#) bei Problemen nach  
 1942 einer Lösung zu suchen. Dieser Link wurde mit [stackoverflow](www.stackoverflow.com) erstellt.

1943 Für Abbildungen gibt es einen ganz ähnlichen Syntax. Mit ! [Das R Logo](abb/r\_logo.png) wird die  
 1944 Abbildung r\_logo.png eingebunden mit der Beschriftung: "Das R Logo".



Abbildung 9: Das R Logo

1945

1946 **Aufgabe 39: Arbeiten mit markdown**

1948 Verwenden Sie das folgende Markdowndokument:

```

1949 ---
1950 title: "Dokument"
1951 author: "Ihr Name"
1952 date: "März 2021"
1953 ---
1954
1955 # Einleitung
1956
1957 # Methoden
1958 1. Kopieren Sie die Vorlage in ein Dokument, das test.md heißt.
1959 2. Fügen Sie zwei Überschriften zweiter und dritter Ordnung hinzu.
1960 3. Fügen Sie einen kursiven Text hinzu.
1961 4. Fügen Sie einen Link zu einer Website hinzu.
1962 5. Kompilieren Sie die Datei, indem Sie in Rstudio auf Preview drücken (Abbildung 10).

```

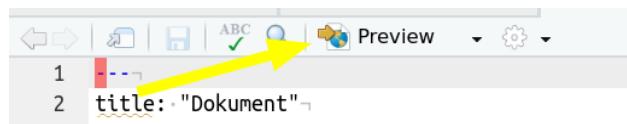


Abbildung 10: Kompilieren einer md-Datei.

1963 **13.2 R und Markdown**

1964 Markdown macht es bereits einfach Textdokumente und Dokumentationen zu verfassen, aber die wirkliche  
 1965 Stärke liegt in der Möglichkeit R und Markdown zu kombinieren. Man spricht dann von Rmarkdown. Ein  
 1966 weiteres Strukturelement, das wir noch nicht kennen gelernt haben, sind Code-Blöcke.

1967 ~~~

1968 a &lt;- 1:10

```

1969 a[1]
1970 ` ` `
1971 erzeugt
1972 a <- 1:10
1973 a[1]

1974 Momentan wird noch kein Code ausgeführt, sondern lediglich der Code, als Code dargestellt. Rmarkdown
1975 bietet nun die Möglichkeit Code beim kompilieren16 auszuführen. Dafür müssen wir nur einen Code-Block als
1976 R-Code-Block kennzeichnen.

1977 ` ` ` {R}
1978 a <- 1:10
1979 a[1]
1980 ` ` `

1981 erzeugt
1982 a <- 1:10
1983 a[1]

1982 ## [1] 1

1983 Beachte, die Variable a wird beim kompilieren erzeugt und steht dann R zur Verfügung. R-Code-Blöcke
1984 werden auch als Code Chunks bezeichnet. Diese Chunks können sehr genau angesprochen und angepasst
1985 werden. Einige wichtige Argumente sind:
1986 • echo: Gibt an, ob der Quelltext angezeigt werden soll oder nicht.
1987 • result: Gibt an, ob die Ergebnisse gezeigt werden sollen oder nicht.
1988 • eval: Diese Option gibt an ob der Chunk ausgeführt werden soll oder nicht.

```

1989

---

**Aufgabe 40: Arbeiten mit Rmarkdown**


---

1992 Erstellen Sie eine neue Rmarkdown Datei mit dem Namen `test1.Rmd`. Erstellen Sie zwei Code-Chunks. Der  
1993 erste soll nicht angezeigt werden und darin werden die Daten geladen (`bhd_1.txt`). Im zweiten Chunk plotten  
1994 Sie das Alter der Bäume gegen den BHD. Was passiert mit dem Plot, wenn Sie die Datei kompilieren (drücken  
1995 Sie dazu auf den Knit-Knopf; Abbildung 11).



Abbildung 11: Kompilieren einer `Rmd`-Datei.

---

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

## 1996 14 Räumliche Daten in R

### 1997 14.1 Was sind räumliche Daten

1998 Räumliche Daten sind Beobachtungen, wie wir sie schon oft gesehen haben, mit einem räumlichen Bezug. Der  
 1999 Unterschied zu nicht räumlichen Daten liegt darin, dass räumliche Daten eindeutig im *Raum* verortet werden  
 2000 können. Häufig werden sogenannte Geoinformationssysteme (GIS) zum Arbeiten mit räumlichen verwendet.  
 2001 R kann in vielerlei Hinsicht wie ein GIS eingesetzt werden und hier werden einige Grundfunktionalitäten  
 2002 dafür besprochen. Räumliche Daten werden in zwei unterschiedliche Datentypen unterteilt: Vektor- und  
 2003 Rasterdaten. Vektordaten modellieren einzelne Objekte (= *Features*). Rasterdaten modellieren eine Oberfläche.  
 2004 Vektordaten bestehen aus zwei Komponenten: 1) einer Geometrie, die die Form und Lage der Daten definiert  
 2005 und 2) Attributen, den tatsächlichen Daten. Räumliche Daten werden oft als *Features* bezeichnet. Ein Feature  
 2006 ist die räumliche Einheit einer Beobachtung. Je nach Art der räumlichen Daten können Features entweder  
 2007 Punkte (z.B. ein Baum), Linien (z.B. eine Straße) oder Polygone (z.B. ein See) sein. Auch können mehrere  
 2008 Geometrien zu einem Feature zusammengefasst werden. Ein Beispiel wäre eine Beobachtung für ein Land,  
 2009 das aber aus mehreren Polygonen bestehen kann (z.B. Festland und Inseln). Features können dann weitere  
 2010 Attribute (= Attributdaten) haben, z.B. eine ID, Name oder was auch immer man gemessen hat.  
 2011 Rasterdaten bestehen aus einer Oberfläche von gleichgroßen Kacheln (= *Pixel*), die ein Gebiet abdecken.  
 2012 Meist sind Pixel viereckig, aber das ist keine Voraussetzung. Dabei hat jedes Pixel einen Wert (das kann  
 2013 auch ein fehlender Wert sein). Typische Beispiele für Rasterdaten sind Landnutzung oder Seehöhen.  
 2014 In R kann sowohl mit Vektor- als auch mit Rasterdaten gearbeitet werden. Für Vektordaten bietet sich das  
 2015 Paket **sf** an und für Rasterdaten das Paket **raster**.

### 2016 14.2 Koordinatenbezugssystem

2017 Eine der Herausforderungen für räumliche Daten ist die eindeutige Verortung im Raum. Dazu braucht man  
 2018 ein *Koordinatenbezugssystem (KBS)*. Einem KBS liegt ein mathematisches Modell der Erde zugrunde. Die  
 2019 Details zu KBS werden schnell relativ kompliziert und wir beschränken uns hier lediglich darauf, wie KBS  
 2020 verwendet werden können. Dazu müssen wir zwei Fälle unterscheiden: 1) einem Datensatz ein KBS zuweisen  
 2021 und 2) Transformation des KBS eines Datensatzes in ein anderes KBS. Die technischen Details werden in  
 2022 den folgend Abschnitten besprochen. Für beide Aufgaben müssen wir auf ein KBS verweisen können, ein  
 2023 einfacher Ansatz dafür sind die sogenannten *EPSG-Codes*<sup>17</sup>.

### 2024 14.3 Vektordaten in R

2025 Das Paket **sf** stellt Klassen zum Abbilden von Features zur Verfügung, die dann in einem **data.frame** als  
 2026 Liste gespeichert werden können. In der Regel erstellen wir Features nicht individuell, sondern lesen diese aus  
 2027 externen Datenquellen (z.B. ESRI Shapefile) ein. Zum besseren Verständnis, erstellen wir es einmal manuell.  
 2028 Wir haben die Koordinaten für drei Städte (Göttingen, Hannover und Berlin) als geografische Koordinaten  
 2029 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))
```

2030 Daraus können wir jetzt eine Geometriespalte für einen `data.frame` erstellen

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

2031 Somit haben wir die Geometrie in der Variable `geom` gespeichert, aber noch keine dazugehörigen Attributdaten.

2032 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)
)
```

2033 In einem letzten Schritt möchten wir jetzt die Geometrie (`geom`) und die Attributdaten (`attr`) zusammenführen.

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

```
2035 ## Simple feature collection with 3 features and 3 fields
2036 ## Geometry type: POINT
2037 ## Dimension: XY
2038 ## Bounding box: xmin: 9.732 ymin: 51.5413 xmax: 13.405 ymax: 52.52
2039 ## Geodetic CRS: WGS 84
2040 ##           name   bundesland   einwohner          geom
2041 ## 1 Goettingen Niedersachsen    119000 POINT (9.9158 51.5413)
2042 ## 2 Hannover Niedersachsen    532000 POINT (9.732 52.3759)
2043 ## 3 Berlin      Berlin    3650000 POINT (13.405 52.52)
```

2044 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.

2046 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)
)
```

2049 Dann kann man mit der Funktion `st_as_sf()` weiter arbeiten:

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

## 2050 14.4 Arbeiten mit Vektordaten

2051 Es gibt sehr viele Funktionen, um mit räumlichen Daten zu arbeiten, von denen wir hier einige vorstellen.

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

```
2052 ## Coordinate Reference System:
2053 ##   User input: EPSG:4326
2054 ##   wkt:
2055 ## GEOGCRS["WGS 84",
2056 ##   ENSEMBLE["World Geodetic System 1984 ensemble",
2057 ##     MEMBER["World Geodetic System 1984 (Transit)"],
2058 ##     MEMBER["World Geodetic System 1984 (G730)"],
2059 ##     MEMBER["World Geodetic System 1984 (G873)"],
2060 ##     MEMBER["World Geodetic System 1984 (G1150)"],
2061 ##     MEMBER["World Geodetic System 1984 (G1674)"],
2062 ##     MEMBER["World Geodetic System 1984 (G1762)"],
2063 ##     MEMBER["World Geodetic System 1984 (G2139)"],
2064 ##     ELLIPSOID["WGS 84",6378137,298.257223563,
2065 ##       LENGTHUNIT["metre",1]],
2066 ##     ENSEMBLEACCURACY[2.0]],
2067 ##     PRIMEM["Greenwich",0,
2068 ##       ANGLEUNIT["degree",0.0174532925199433]],
2069 ##     CS[ellipsoidal,2],
2070 ##       AXIS["geodetic latitude (Lat)",north,
2071 ##         ORDER[1],
2072 ##         ANGLEUNIT["degree",0.0174532925199433]],
2073 ##       AXIS["geodetic longitude (Lon)",east,
2074 ##         ORDER[2],
2075 ##         ANGLEUNIT["degree",0.0174532925199433]],
2076 ##     USAGE[
2077 ##       SCOPE["Horizontal component of 3D system."],
2078 ##       AREA["World."],
2079 ##       BBOX[-90,-180,90,180]],
2080 ##     ID["EPSG",4326]]
```

2081 Wenn wir jetzt zu einem anderen KBS (z.B. EPSG:3035, ein europäisches projiziertes KBS) umrechnen  
 2082 möchten, können wir das mit

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

```

2083 ## Coordinate Reference System:
2084 ##   User input: EPSG:3035
2085 ##   wkt:
2086 ## PROJCRS["ETRS89-extended / LAEA Europe",
2087 ##   BASEGEOGCRS["ETRS89",
2088 ##     ENSEMBLE["European Terrestrial Reference System 1989 ensemble",
2089 ##       MEMBER["European Terrestrial Reference Frame 1989"],
2090 ##       MEMBER["European Terrestrial Reference Frame 1990"],
2091 ##       MEMBER["European Terrestrial Reference Frame 1991"],
2092 ##       MEMBER["European Terrestrial Reference Frame 1992"],
2093 ##       MEMBER["European Terrestrial Reference Frame 1993"],
2094 ##       MEMBER["European Terrestrial Reference Frame 1994"],
2095 ##       MEMBER["European Terrestrial Reference Frame 1996"],
2096 ##       MEMBER["European Terrestrial Reference Frame 1997"],
2097 ##       MEMBER["European Terrestrial Reference Frame 2000"],
2098 ##       MEMBER["European Terrestrial Reference Frame 2005"],
2099 ##       MEMBER["European Terrestrial Reference Frame 2014"],
2100 ##       ELLIPSOID["GRS 1980",6378137,298.257222101,
2101 ##         LENGTHUNIT["metre",1]],
2102 ##       ENSEMBLEACCURACY[0.1]],
2103 ##       PRIMEM["Greenwich",0,
2104 ##         ANGLEUNIT["degree",0.0174532925199433]],
2105 ##       ID["EPSG",4258],
2106 ##       CONVERSION["Europe Equal Area 2001",
2107 ##         METHOD["Lambert Azimuthal Equal Area",
2108 ##           ID["EPSG",9820]],
2109 ##         PARAMETER["Latitude of natural origin",52,
2110 ##           ANGLEUNIT["degree",0.0174532925199433],
2111 ##           ID["EPSG",8801]],
2112 ##         PARAMETER["Longitude of natural origin",10,
2113 ##           ANGLEUNIT["degree",0.0174532925199433],
2114 ##           ID["EPSG",8802]],
2115 ##         PARAMETER["False easting",4321000,
2116 ##           LENGTHUNIT["metre",1],
2117 ##           ID["EPSG",8806]],
2118 ##         PARAMETER["False northing",3210000,
2119 ##           LENGTHUNIT["metre",1],
2120 ##           ID["EPSG",8807]]],
2121 ##       CS[Cartesian,2],
2122 ##         AXIS["northing (Y)",north,
2123 ##           ORDER[1],
2124 ##           LENGTHUNIT["metre",1]],
2125 ##         AXIS["easting (X)",east,

```

```

2126 ##           ORDER[2] ,
2127 ##           LENGTHUNIT["metre",1]],
2128 ##           USAGE[
2129 ##           SCOPE["Statistical analysis."],
2130 ##           AREA["Europe - European Union (EU) countries and candidates. Europe - onshore and offshore: "],
2131 ##           BBOX[24.6,-35.58,84.73,44.83]],
2132 ##           ID["EPSG",3035]

```

2133 Die Funktion `st_buffer()` erlaubt es Features zu puffern, mit `st_distance()` kann die Distanz zwischen  
2134 Features berechnet werden, mit `st_area()` kann die Fläche eines Features berechnet werden.

2135 Funktionen wie `st_intersection()`, `st_union()` und `st_difference()` erlauben es geometrische Opera-  
2136 tionen zwischen unterschiedlichen Features zu berechnen. Für eine ausführliche Diskussion siehe auch hier:  
2137 <https://geocompr.robinlovelace.net/geometric-operations.html>.

2138 Normalerweise lesen wir Daten von externen Datenquellen (z.B. ESRI Shapefiles). Das geht mit der Funktion  
2139 `st_read()`.

## 2140 14.5 Rasterdaten in R

2141 Für Rasterdaten gibt es das R-Paket `terra`. Auch hier wollen wir uns wieder auf einige Grundfunktionalitäten  
2142 konzentrieren. Diese umfassen das Einlesen, Zuschneiden, Rechnen und Abfragen von Rastern.

2143 Mit der Funktion `rast()` kann ein Raster in R eingelesen werden.

```

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

```

2144 `dem` steht für *Digital Elevation Model* und ist ein Raster mit den Seehöhen in Niedersachsen mit einer  
2145 500-m-Auflösung. Wir können diese mit der Funktion `res()`<sup>18</sup> abfragen.

```

res(dem)

```

2146 ## [1] 500 500

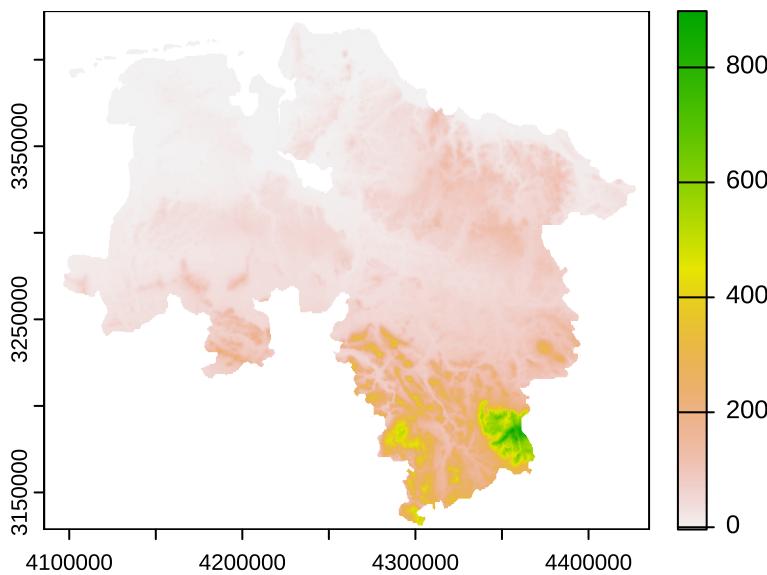
2147 Bzw. wir können den Raster auch plotten.

```

plot(dem)

```

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



2148

2149 Wenn wir den Raster `dem` auf ein Gebiet zuschneiden wollen (z.B. Göttingen), müssen wir drei Schritte  
 2150 durchführen. Als erstes müssen wir ein Shapefile für Göttingen einlesen.

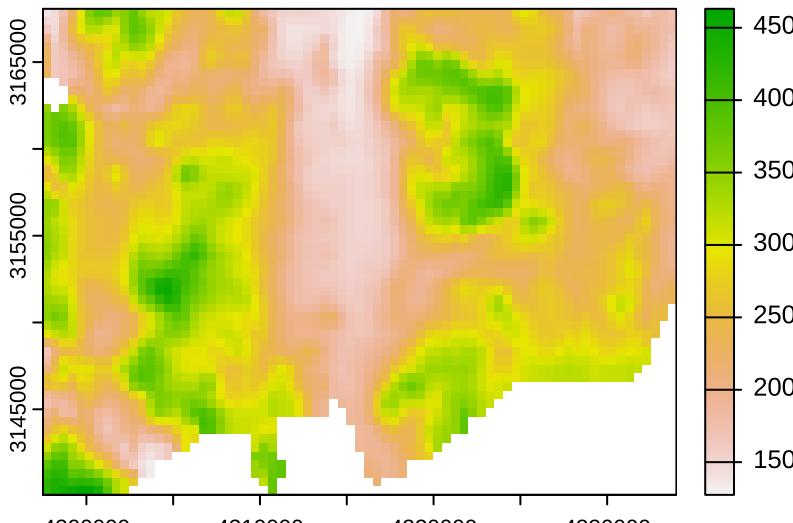
```
goe <- st_read(here::here("data/goettingen/stadt_goettingen.shp"))
```

2151 Dann müssen wir sicherstellen, dass sowohl der Raster `dem` als auch das `sf`-Objekt `goe` im selben KBS sind.  
 2152 Es bietet es sich in der Regel an, das KBS des Vektors zu transformieren. Mit der Funktion `projectRaster()`  
 2153 kann das KBS eines Raster transformiert werden.

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

2154 Mit der Funktion `crop()` kann der Raster `dem` auf Göttingen zugeschnitten werden.

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



2155

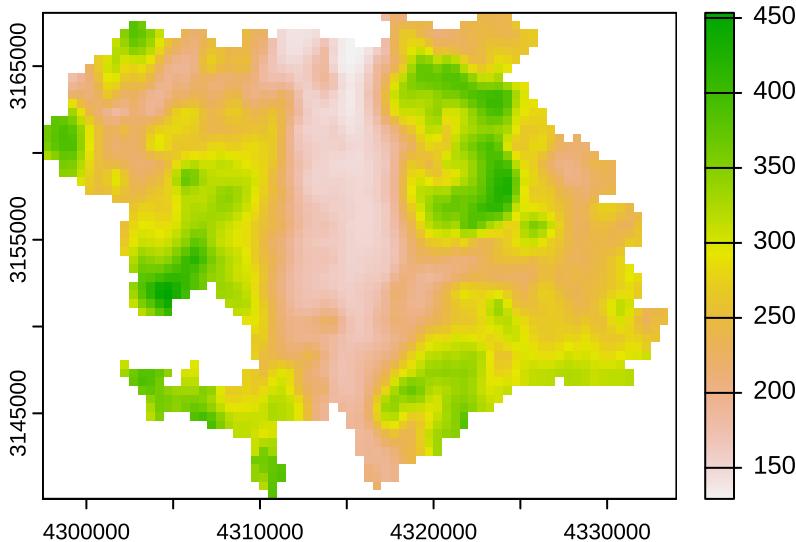
2156 Der Raster hat jetzt die Größe einer Bounding-Box (BBX) von Göttingen (das ist ein Rechteck, das Göttingen  
 2157 umfasst). Mit der Funktion `mask()` kann der Raster auf die genauen Grenzen des Vektors `goe` angepasst

2158 werden.

```
2158 dem2 <- mask(dem1, goe)
```

2159 ## Warning: [mask] CRS do not match

```
2159 plot(dem2)
```



2160

2161 Wenn wir an bestimmten Punkten den Wert des Rasters abfragen wollen (z.B. an `cities`) von vorhin, dann  
2162 gibt es dafür die Funktion `extract`. Dann müssen wir erst sicherstellen, dass `staedte` und `dem` gleichen KBS  
2163 zu grunde liegt. Dafür transformieren wir einfach `staedte` in das KBS von `dem`. Mit der Funktion `crs()`  
2164 erhalten wir das KBS des Rasters.

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

2165 Wenn wir das KBS eines Objektes nicht kennen, können wir auch einfach das KBS übergeben. Der folgende  
2166 Code-Block macht genau das Gleiche mit dem Vorteil, dass wir keinen EPSG-Code angeben müssen.

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

2167 Dann können wir für jede Stadt die Seehöhe abfragen:

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

2168 ## ID dem\_3035

2169 ## 1 1 149.18181

2170 ## 2 2 57.21486

2171 ## 3 3 NA

2172 Mit `terra::extract()` rufen wir *eindeutig* die Funktion `extract()` aus dem Paket `terra` auf. Wir müssen  
2173 das so machen, weil es im Paket `dplyr` auch eine Funktion `extract()` gibt, die wir hier nicht anwenden  
2174 möchten, da sie einen Fehler verursachen würde.

2175 Ein analoges Vorgehen ist auch für Linien und Polygone möglich.

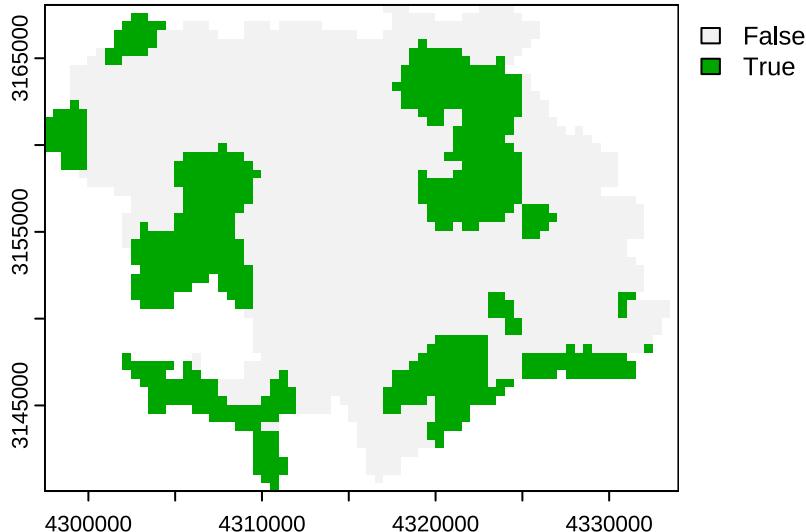
2176 Mit Rastern kann auch einfach gerechnet werden. Wir können z.B. die Seehöhe in Kilometern anstatt Metern

2177 berechnen:

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

2178 Auch logische Operationen sind möglich, wenn wir alle Rasterzellen mit einer Seehöhe von mehr als 300 m in  
2179 Göttingen suchen, dann geht das so:

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



2180

2181 Wenn wir jetzt auf die Werte des Rasters `dem3` zugreifen wollen, geht das mit eckigen Klammern.

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

```
2182 ##      dem_3035
2183 ## [1,]     NA
2184 ## [2,]     NA
2185 ## [3,]     NA
2186 ## [4,]     NA
2187 ## [5,]     NA
2188 ## [6,]     NA
```

2189 Das sind erst einmal viele NA-Werte für die ganzen Zellen, die außerhalb von Niedersachsen liegen. Aber wir  
2190 können mit so einem Vektor ganz normal arbeiten und z.B. die Fläche des Landes Niedersachsen die eine  
2191 Seehöhe von mehr als 500m Seehöhe hat ausrechnen.

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

2192 ## [1] 0.2786229

2193

---

2194 **Aufgabe 41: Arbeiten mit Rastern**

---

2196 Verwenden Sie den Raster `wald.tif`, der auf einer 10 m Auflösung den Waldanteil jeder Rasterzelle angibt<sup>19</sup>.  
 2197 Der EPSG-Code für das KBS von `wald.tif` ist 3035. Nehmen Sie an, dass wenn der Waldanteil in einer  
 2198 Raster größer als 50 % ist, dass die Rasterzelle als Wald klassifiziert werden kann. Wie viel Prozent des  
 2199 Göttinger Stadtgebietes sind Wald? Wie ändert sich dieser Wert, wenn sie 70 % anstatt 50 % als Schwellenwert  
 2200 für Wald annehmen?

2201

---

2202 **Aufgabe 42: Studiendesign**

---

2204 Mit der Funktion `st_sample()` können Sie innerhalb oder entlang eines Features zufällige Punkte legen. Das  
 2205 Argument `n` steuert die Anzahl Punkte und das Argument `type` wie die Punkte angeordnet werden. Für `type`  
 2206 sind für uns die Werte `type = "random"` (komplett zufällig), `type = "regular"` (regelmäßiger Grid) und  
 2207 `type = "hexagonal"` von Bedeutung (ein hexagonaler Grid, d.h. ein sechseckiger Raster). Unglücklicherweise  
 2208 ist das Ergebnis von `st_sample()` erst eine Geometrie. Um daraus ein vollständiges `sf`-Objekt zu machen  
 2209 und problemlos weiter arbeiten zu können, müssen Sie nocheinmal die Funktion `st_as_sf()` ausführen.  
 2210 Stellen Sie sich vor, dass wir die tatsächliche Waldbedeckung des Göttinger Stadgebietes **nicht** kennen und wir  
 2211 eine Studie durchführen, um den Anteil des Göttinger Stadgebietes, der mit Wald bedeckt ist herauszufinden.  
 2212 Erstellen Sie dafür einige unterschiedliche Stichproben (diese können in der Anzahl und Anordnung variieren).  
 2213 Berechnen Sie für jedes Stichprobendesign den Anteil an Wald und ein dazugehöriges Konfidenzintervall  
 2214 (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  $n$   
 2215 die Stichprobengröße). Nehmen Sie an, dass eine Rasterzelle Wald ist, sobald > 50 % der Rasterzelle mit  
 2216 Wald bedeckt ist.

2217

---

2218 **Aufgabe 43: Räumliche Daten**

---

2220 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)
)
```

2221 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

- 2222 2. Puffern Sie jeden Baum mit seinem Kronendurchmesser.
- 2223 3. Berechnen Sie die Kronenfläche jedes Baumes. *Hinweis: Die Funktion `st_area()` könnte dafür hilfreich sein.*
- 2224 4. Welcher Baum hat die größte Kronenfläche?
- 2225 5. Finden Sie für jede Art, den Baum mit der größten Kronenfläche.

2227

2228 **Aufgabe 44: Arbeiten mit räumlichen Daten**

---

- 2230 1. Lesen Sie das ESRI Shapefile `goettingen/stadt_goettingen.shp` ein.
- 2231 2. Wie viele Features befinden sich in dem Shapefile?
- 2232 3. Welches Koordinatenbezugssystem (KBS) hat das Shapefile?
- 2233 4. Transformieren Sie das Shapefile in das KBS 3035.
- 2234 5. Erstellen Sie eine neue Spalte `A` in der Sie die Fläche jeder Gemeinde/Stadt speichern.
- 2235 6. Welche Gemeinde/Stadt (Spalte `GEN`) ist am größten?
- 2236 7. Wählen Sie nun nur die Stadt Göttingen aus.

2237

2238 **Aufgabe 45: Arbeiten mit räumlichen Daten 2**

---

- 2240 1. Lesen Sie erneut das ESRI Shapefile `goettingen/stadt_goettingen.shp` ein.
- 2241 2. Lösen sie die Gemeindegrenzen auf (die Funktion `st_union()` könnte hier nützlich sein).
- 2242 3. Wie groß ist das resultierende Feature?

2243 **15 FAQs (Oft gefragtes)**

2244 **15.1 Arbeiten mit Daten**

2245 **15.1.1 Einlesen von Exceldateien**

- 2246 Mit der Funktion `read_excel()` aus dem Paket `readxl` können Exceldateien direkt in R eingelesen werden.  
2247 Ein Export als csv-Datei aus Excel ist nicht notwendig.

## 2248 16 Literatur

- 2249 Ein guter Überblick über viele der angeschnittenen Themen gibt es in dem [R for data science](#), das online frei  
2250 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.  
2251
- 2252 McNamara, Amelia, and Nicholas J Horton. 2018. “Wrangling Categorical Data in r.” *The American Statistician*  
2253 72 (1): 97–104.
- 2254 Wickham, Hadley. 2014. “Tidy Data.” *Journal of Statistical Software* 59 (10). <https://doi.org/10.18637/jss.v059.i10>.
- 2255