

1

Einführung in die Datenanalyse mit R (700104)

4

Kursskript

5



6

Dr. Johannes SIGNER
Abteilung Wildtierwissenschaften
Büsgenweg 3
37077 Göttingen

jsigner@uni-goettingen.de

Dr. Kai HUSMANN
Abteilung Forstökonomie und nachhaltige Landnutzungsplanung
Büsgenweg 3
37077 Göttingen

kai.husmann@uni-goettingen.de

7



8

Fakultät für Forstwissenschaften und Waldökologie
Georg-August-Universität Göttingen

10



11

12

Wintersemester 2024/2025

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

¹⁵ Zitiervorschlag:

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

¹⁸ Letzte Aktualisierung: 10. Dezember 2024

19 Vorwort und Danksagung

20

- 21 Lernziel des Kurses ist die Einführung in die Arbeit, Visualisierung und Analyse von (forstlichen) Daten
22 mit dem Statistikprogramm *R*. Der Schwerpunkt liegt dabei auf der Datenverarbeitung (Data Science).
23 Statistische Methoden werden nur exemplarisch angewendet. Sie werden lernen, wie Sie Daten einlesen, in
24 eine sinnvolle Struktur überführen, vereinfachen und visuell darstellen.
- 25 Weitere Materialien als dieses Kursskript und die Übungsaufgaben (StudIP) werden nicht benötigt. Die
26 gelöste Übungsaufgaben dieses Skriptes werden Ihnen über StudIP zugänglich gemacht. Dort werden auch
27 Ankündigungen bekanntgegeben. Um die Credits für diesen Kurs zu erhalten, müssen Sie am Ende des Kurses
28 eine mündliche Prüfung ablegen. Für die Prüfung werden Sie zwei zufällig gezogene Prüfungsfragen aus
29 dem Dokument "Übungen: Einführung in die Datenanalyse mit R"(StudIP) bearbeiten und vorstellen. Die
30 Übungsaufgaben sollen sie während des Kurses parallel bereits bearbeiten. Wir werden Ihnen jedoch keine
31 Hilfestellung dazu anbieten. Nach einer 15-minütigen Vorbereitungszeit beträgt die Prüfungszeit weitere 15
32 Minuten. In der Prüfungszeit präsentieren Sie zunächst Ihre Lösung und beantworten anschließend vertiefende
33 Fragen zu Ihrer Lösung und daraufhin auch zum gesamten Lehrinhalt des Kurses.
- 34 Dieses Vorlesungsskript ist ein R Markdown-Dokument, das mit R und RStudio erstellt wurde. Das Dokument
35 besteht aus Fließtext, R Code und den entsprechenden Code-Ergebnissen. Die grau hinterlegten Codepassagen
36 sind kurze R-Skripte. Falls das Skript eine Konsoleausgabe erzeugt, ist diese direkt mit "##" markiert (diese
37 Begriffe werden in Kapitel 1.2 näher erläutert).
- 38 Dank für Anmerkungen gilt Markus Benesch, Sofie Biberacher und Josephine Trisl. Teile des Unterkapitels
39 zu Schleifen und Kontrollstrukturen sind an das R Skript des Kurses Computergestützte Datenanalyse von
40 Robert Nuske, Nikolas von Lüpke und Joachim Saborowski angelehnt. Des Weiteren wurden Beispiele aus
41 dem frei Verfügbaren Dokument *R for Data Science* (<https://r4ds.hadley.nz/intro.html>) entnommen.

42 Inhaltsverzeichnis

| | | |
|----|-----------------------------------------------------------------------|----|
| 43 | 1 Einleitung | 4 |
| 44 | 1.1 Data Science mit R | 4 |
| 45 | 2 R und RStudio | 5 |
| 46 | 2.1 Installation von R und RStudio | 5 |
| 47 | 2.2 Erste Schritte in R | 5 |
| 48 | 2.3 Gute Praxis bei der Programmierung | 7 |
| 49 | 2.4 RStudio Projekte | 8 |
| 50 | 2.4.1 Erstellen eines Projektes | 8 |
| 51 | 3 Variablen, Funktionen und Datentypen | 10 |
| 52 | 3.1 Variablen beim Programmieren | 10 |
| 53 | 3.2 Funktionen | 12 |
| 54 | 3.3 Datentypen | 12 |
| 55 | 3.4 Datenstrukturen | 13 |
| 56 | 4 Vektoren | 15 |
| 57 | 4.1 Funktionen zum Arbeiten mit Vektoren | 17 |
| 58 | 4.2 Statistische Funktionen | 18 |
| 59 | 4.3 Beispiel Fotofallen | 19 |
| 60 | 4.4 Arbeiten mit logischen Werten | 20 |
| 61 | 4.5 Zugreifen auf Elemente eines Vektors (=Untermengen) | 22 |
| 62 | 4.6 Der %in%-Operator | 24 |
| 63 | 5 Faktoren (factors) | 25 |
| 64 | 5.1 Das Paket forcats | 27 |
| 65 | 5.1.1 Anpassen der Anordnung von Faktoren | 27 |
| 66 | 6 Spezielle Einträge | 29 |
| 67 | 6.1 NA | 29 |
| 68 | 6.2 NULL | 30 |
| 69 | 6.3 Inf | 30 |
| 70 | 7 data.frames oder Tabellen | 32 |
| 71 | 7.1 Wichtige Funktionen zum Arbeiten mit data.frames | 33 |
| 72 | 7.2 Zugreifen auf Elemente eines data.frame | 34 |
| 73 | 8 Schreiben und lesen von Daten | 37 |
| 74 | 8.1 Textdateien | 37 |
| 75 | 9 Erstellen von Abbildungen | 39 |
| 76 | 9.1 Base Plot | 39 |
| 77 | 9.1.1 Mehrere Panels | 45 |
| 78 | 9.1.2 Speichern von Abbildungen | 45 |

| | | |
|-----|----------------------------------------------------------|------------|
| 79 | 9.2 Histogramme | 46 |
| 80 | 9.3 Boxplots | 49 |
| 81 | 9.4 ggplot2: Eine Alternative für Abbildungen | 51 |
| 82 | 9.4.1 Multipanel Abbildungen | 58 |
| 83 | 9.4.2 Plots kombinieren | 61 |
| 84 | 9.4.3 Speichern von plots | 63 |
| 85 | 10 Mit Daten arbeiten | 65 |
| 86 | 10.1 dplyr eine Einführung | 65 |
| 87 | 10.2 Arbeiten mit gruppierten Daten | 68 |
| 88 | 10.3 pipes oder %>% | 69 |
| 89 | 10.4 Joins | 70 |
| 90 | 10.5 ‘long’ and ‘wide’ Datenformate | 72 |
| 91 | 10.6 Auswählen von Variablen | 74 |
| 92 | 10.7 Einzelne Beobachtungen abfragen (slice()) | 75 |
| 93 | 10.8 Spalten trennen | 78 |
| 94 | 11 Arbeiten mit Text | 80 |
| 95 | 11.1 Arbeiten mit Text | 80 |
| 96 | 11.2 Finden von Textmustern | 81 |
| 97 | 12 Arbeiten mit Zeit | 84 |
| 98 | 12.1 Arbeiten mit Zeitintervallen | 85 |
| 99 | 12.2 Formatieren von Zeit | 87 |
| 100 | 12.3 Zeitreihen | 87 |
| 101 | 13 Aufgaben Wiederholen (for-Schleifen) | 93 |
| 102 | 13.1 Schleifen | 93 |
| 103 | 13.1.1 Wiederholen von Befehlen mit for(). | 93 |
| 104 | 13.1.2 Wiederholen von Befehlen mit while() | 96 |
| 105 | 13.2 Bedingte Ausführung von Codeblöcken | 96 |
| 106 | 14 (R)markdown | 98 |
| 107 | 14.1 Markdown Grundlagen | 98 |
| 108 | 14.2 R und Markdown | 99 |
| 109 | 15 Räumliche Daten in R | 101 |
| 110 | 15.1 Was sind räumliche Daten | 101 |
| 111 | 15.2 Koordinatenbezugssystem | 101 |
| 112 | 15.3 Vektordaten in R | 101 |
| 113 | 15.4 Arbeiten mit Vektordaten | 103 |
| 114 | 15.5 Rasterdaten in R | 105 |
| 115 | 16 FAQs (Oft gefragtes) | 111 |
| 116 | 16.1 Arbeiten mit Daten | 111 |
| 117 | 16.1.1 Einlesen von Exceldateien | 111 |

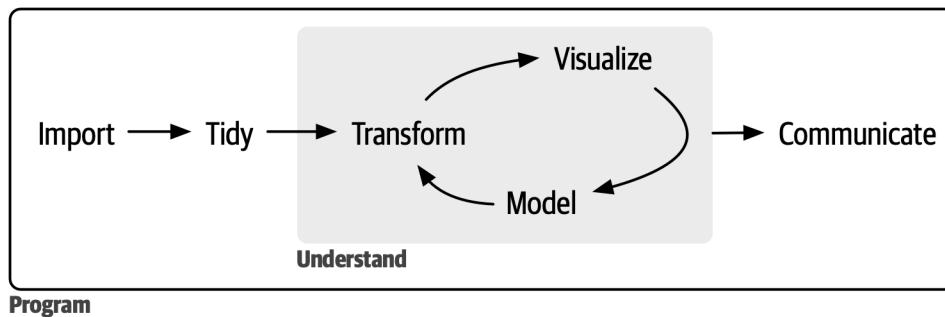
118 **17 Literatur**

112

¹¹⁹ **1 Einleitung**

¹²⁰ **1.1 Data Science mit R**

¹²¹ Ein Data Science Projekt besteht laut Wickham et al. (<https://r4ds.hadley.nz/>) aus 4 Stufen.



¹²² Wir werden in diesem Kurs zwar alle Stufen besprechen, jedoch in unterschiedlicher Intensität. Jedes Data Science Projekt beginnt mit dem Datenimport. Dies sind in der Regel Tabellen mit Erhebungsdaten, die je nach Experiment sehr unterschiedlich aufgebaut sein können und auch sehr unterschiedlich relevante Informationen enthalten können. Aus diesem Grund liegt ein Fokus dieses Kurses auf der Datenvorbereitung (*Tidy Data*). *Tidying* bedeutet, die Daten so herzustellen, dass sie konsistent und passend zu der Datenphilosophie von R sind. Bei *Tidy Data* enthalten die Zeilen die Beobachtungen und die Spalten die Variablen. Alle weiteren Informationen, wie z. B. Metadaten, Zusammenfassungen oder Erläuterungen, werden in separaten Tabellen gespeichert. Überführen Sie Ihre Daten in das *Tidy* Format, um den Überblick zu behalten und bei den folgenden Arbeiten Zeit zu sparen. *Transform* ist der Arbeitsschritt, in dem die relevanten Daten aus den *Tidy* Daten für spezifische Modelle oder Abbildungen erzeugt werden. Dies kann z. B. das filtern von Teilmengen, falls nur bestimmte Beobachtungen analysiert werden sollen, oder das Erzeugen von zusätzlichen Variablen (z. B. H/D-Wert aus Höhe und BHD) sein. Abbildungen (*Visualization*) sind eine effiziente Möglichkeit Ihre Daten zu beschreiben und zu kommunizieren. Abbildungen können auch über die reine Wiedergabe hinausgehende Informationen enthalten und lassen sich mit Modellierungsergebnissen verbinden. Gute Abbildungen zu erzeugen ist eine wichtige Aufgabe bei den meisten Data Science Anwendungen. Wir werden uns im Kurs intensiv mit Abbildungen beschäftigen. (Statistische) Modellierung *Model* ist zwar für das wissenschaftliche Arbeiten ebenfalls relevant und R ist hier sehr mächtig (R versteht sich als statistische Programmiersprache, obwohl R für Data Science Anwendungen aller Art geeignet ist), wird in diesem Kurs jedoch nur exemplarisch am Rande behandelt. Die große Zahl statistischer Modelle, welche je nach Anwendung sehr spezifisch sind, können im Rahmen des Kurses nicht abgedeckt werden. Diese können Sie im Masterstudium in einer großen Zahl von Modulen vertiefen (z. B. Statistical Data Analysis with R (M.FES.115) oder Advanced Data Analysis with R(M.FES.121)). Kommunikation *Communicate* funktioniert nur, wenn Modelle und Abbildungen auch nachvollziehbar sind. Wir werden im Kurs zeigen, wie Sie ein PDF Dokument (oder Word, Power Point, ...) aus Ihren Ergebnissen und Abbildungen erstellen (dieses Dokument ist eine aus R erstellte PDF Datei).

147 2 R und RStudio

148 2.1 Installation von R und RStudio

- 149 Als ersten Schritt müssen Sie R und RStudio installieren. Dabei ist wichtig zu unterscheiden, dass R und
150 RStudio zwei unterschiedliche Programme sind. R ist die eigentliche Programmiersprache mit der wir arbeiten.
151 RStudio hingegen ist eine sogenannte Entwicklungsumgebung¹, die das Arbeiten mit R vereinfacht.
- 152 Sie können also mit R arbeiten ohne RStudio (was unüblich ist), aber nicht mit RStudio ohne R. RStudio
153 wird unter anderem verwendet, um R Code komfortabler zu schreiben und zu verwalten. Es werden Ihnen für
154 das Programmieren relevante Informationen bereitgestellt.
- 155 Gehen Sie für die Installation von R, auf die Website <https://cloud.r-project.org/>, laden Sie die für ihren
156 Computer passende R-Version herunter und folgen Sie den Installationsanweisungen. In Linux können Sie R
157 über die Kommandozeile installieren.
- 158 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.

160 2.2 Erste Schritte in R

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

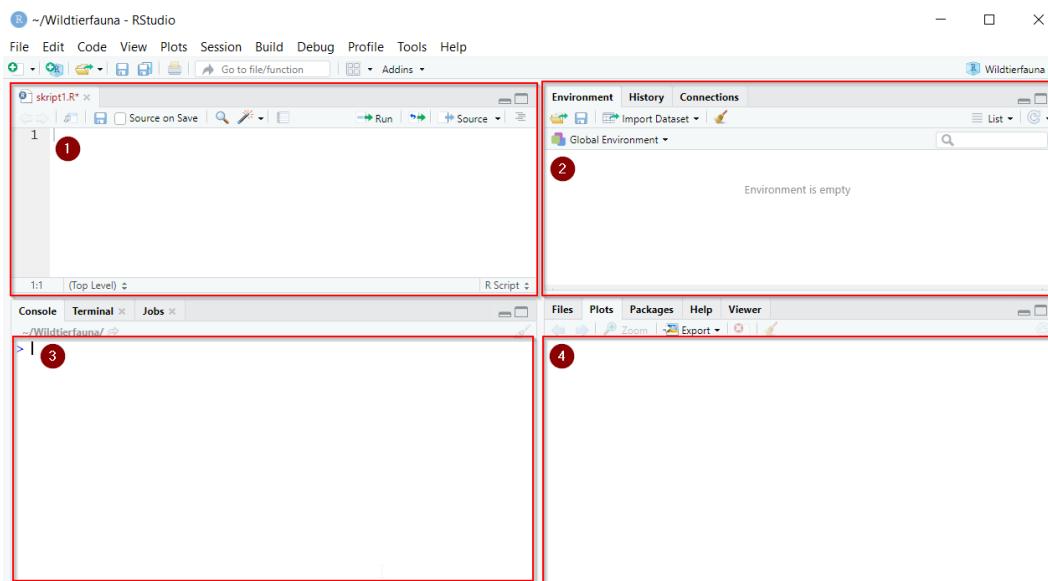


Abbildung 1: RStudio Panes.

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

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

- 167 1. Hier werden Skripte anzeigt, d.h., hier wird meist R Code geschrieben und dokumentiert. Der Code wird
 168 beim Schreiben noch nicht ausgeführt (siehe Punkt 3), dennoch sollten Sie Code immer so schreiben,
 169 dass er Zeile für Zeile abgeschickt werden kann. Sie sollten nie Code schreiben, bei dem Sie zwischen
 170 den Zeilen hin und her springen müssen.
- 171 2. Der zweite Ausschnitt enthält Informationen über den *Workspace*. Im Workspace werden alle verfügbaren
 172 Objekte angezeigt.
- 173 3. Die eigentliche R-Konsole ist in Ausschnitt 3. Hier wird in der Regel wenig Code eingegeben. Der
 174 normale Workflow ist vom Skript Code an die Konsole zu schicken. Erst durch das Abschicken in die
 175 Konsole wird der Code (bzw. die Teile des Codes, die Sie abschicken) ausgeführt. Das Ergebnis des
 176 Codes wird in der Konsole angezeigt, falls ihr Code ein Ergebnis erzeugt.
- 177 4. Der vierte Ausschnitt enthält mehrere Reiter. Der Reiter *Files* zeigt den Verzeichnisbaum an, in dem
 178 sie arbeiten. Im Reiter *Plots* werden Abbildungen angezeigt, die Sie in der Konsole erzeugt haben.
 179 Hilfeseiten zu Funktionen werden im Reiter *Help* angezeigt.
- 180 Einfache Rechenoperationen können auch direkt in der R-Konsole durchgeführt werden. Prinzipiell könnten
 181 Sie alle Operationen direkt in die Konsole tippen. Der Nachteil und der Grund, warum dies keine gute Praxis
 182 ist, ist, dass der Code zwar ausgeführt jedoch nicht gespeichert wird. Code der nicht als Skript gespeichert
 183 wird, ist also nicht reproduzierbar. Tippen Sie die folgenden Operationen in die Konsole.

10 + 5184 **## [1] 15****20 - 10**185 **## [1] 10****10 * 3**186 **## [1] 30****100 / 19**187 **## [1] 5.263158**

188 Sie sehen Ihren Code in rot und das Ergebnis dieser Operation in weiß darunter. Die Zahl in [] gibt die
 189 Position der Einträge wider. Hier also [1], da es sich nur um einen Wert handelt, der demnach an erster
 190 Position steht. Dieses Skript wurde in R Markdown geschrieben (siehe Vorwort). R Markdown verbindet Text
 191 und Code. Die Ergebnisse des Codes werden unter dem grau hinterlegten *Codechunk* dargestellt. Darstellung
 192 und Farbe des Codes und der Ergebnisse sind jedoch nicht immer exakt so wie sie es in der R Konsole wären.

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

197 Meist verwenden wir jedoch **Skripte**, um den R-Code zu schreiben und ihn dann an die Konsole zu schicken.
 198 Dies hat den Vorteil, dass alle Schritte nachvollziehbar bleiben und Analysen beliebig oft wiederholt werden
 199 können. Nach der Ausführung bleibt der Code erhalten und Sie dokumentieren Ihre Berechnungen automatisch
 200 mit. Stellen Sie sich den Code im R-Skript wie ein Kochbuch vor. Wenn wir R-Code in einem R-Skript

201 geschrieben haben gibt es mehrere Möglichkeiten diesen Code abzuschicken/ auszuführen. Wir können eine
 202 Zeile abschicken, indem wir entweder auf *Run* klicken (Abbildung 2) oder die Tastenkombination *Strg +*
 203 *Enter* (*Strg*+*↵*) tippen. Mehrere Zeilen abzuschicken und nacheinander ausführen zu lassen ist möglich,
 204 indem diese Zeilen markiert werden bevor Sie *Run* klicken oder die Tastenkombination tippen. Ein Klick auf
 205 *Source* bzw. die Tastenkombination *Strg + Umschalt + Enter* (*Strg*+*↑*+*↵*).

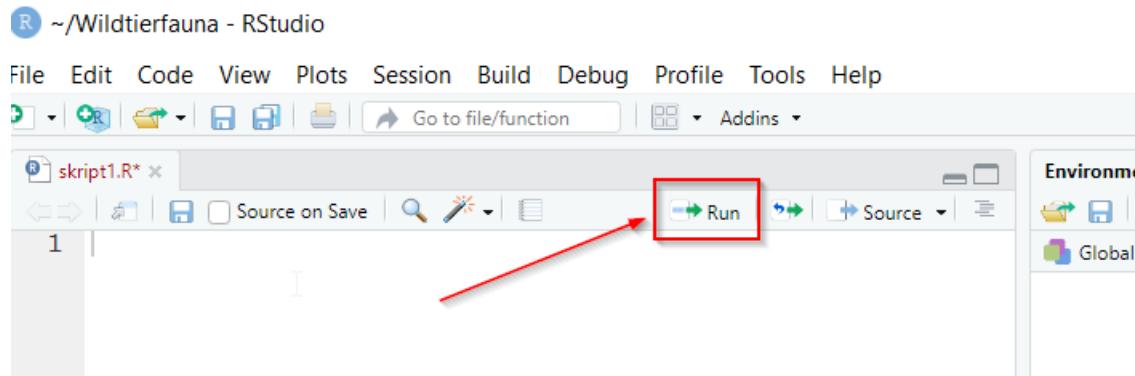


Abbildung 2: Zeilenweises Ausführen von Code in RStudio.

206 Wenn Sie die Codezeile abgeschickt haben, sehen Sie diese Zeile in der Konsole und direkt darunter das
 207 Ergebnis sowie die Dimension des Ergebnisses, also genauso, als hätten Sie den Code direkt in die Konsole
 208 getippt. Die Konsole erkennt, wenn Sie einen unvollständigen Code abschicken. In der Konsole sehen Sie in
 209 diesem Fall kein Ergebnis, sondern ein + unter der abgeschickten Zeile. Sie können nun eine weitere Zeile zur
 210 vervollständigung abschicken oder in der Konsole *Escape* (*Esc*) drücken, um abzubrechen. Sehr lange Befehle
 211 können Sie im Skript somit über mehrere Zeilen aufteilen. Nutzen Sie diese Eigenschaft, um übersichtliche
 212 Codes zu schreiben.

213 2.3 Gute Praxis bei der Programmierung

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

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

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

229 ## [1] 9

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

236

237 Aufgabe 1: Ausführen von Quellcodes

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

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

# Einfache Funktion
sqrt(20)
```

241 Führen Sie nun alle Zeilen aus.

242 2.4 RStudio Projekte

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

- 246 1. Sie können Ihre R-Session direkt in dem Projekt starten.
- 247 2. R-Projekte können zwischen unterschiedlichen Rechnern geöffnet werden, ohne dass der Pfad angepasst
 248 werden muss.

249 2.4.1 Erstellen eines Projektes

250 Zum Erstellen eines Projektes müssen folgende Schritte durchlaufen werden, diese sind in Abbildung 3
 251 zusammengefasst.

- 252 1. Gehen Sie zu `File > New Project ...`
- 253 2. Wählen Sie `New Project`.

- 254 3. Geben Sie einen Namen für das Projekt ein (z.B. den Namen einer Lehrveranstaltung) und ein neuer
 255 Ordner mit dem Projektnamen wird erstellt.
- 256 4. Drücken Sie auf **Create Project**.

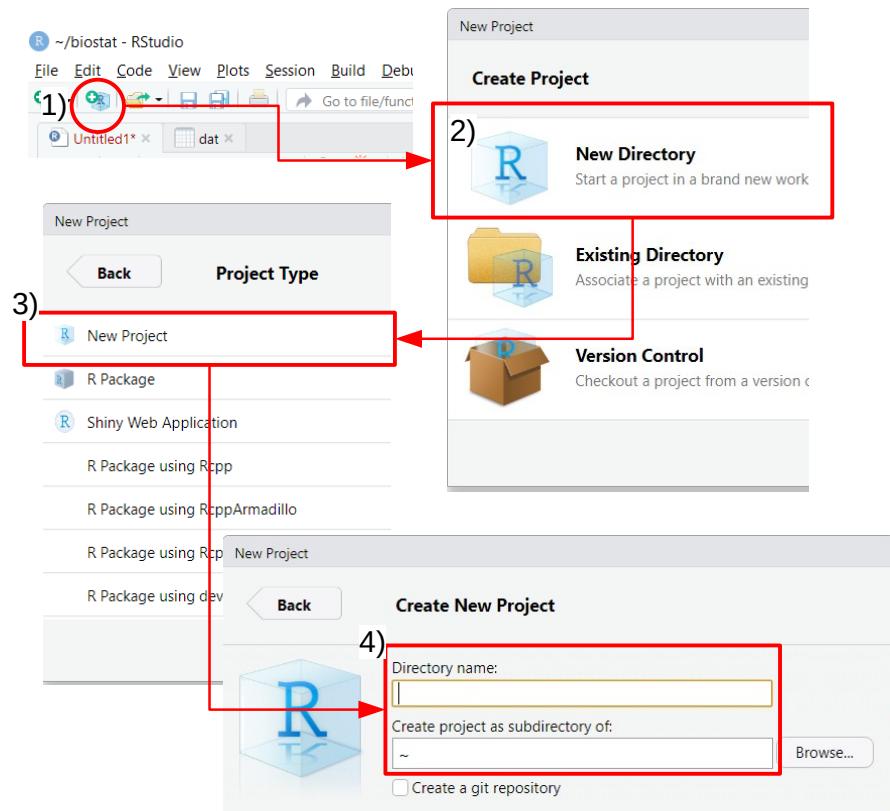


Abbildung 3: Workflow zum Erstellen eines Projekts.

- 257 Sobald ein Projekt einmal erstellt wurde, können Sie einfach wieder auf das Projekt-Icon klicken und das
 258 Projekt wieder öffnen (Abbildung 4)
- 259 Alternativ kann über **File > Open Project** in RStudio oder durch Auswählen des Projektnamens (siehe folgende
 260 Abbildung) geöffnet werden (Abbildung 5).

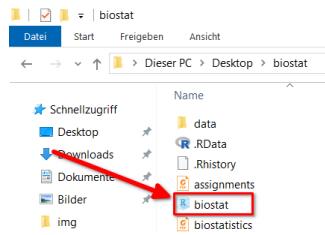


Abbildung 4: Öffnen eines RStudio Projekts.



Abbildung 5: Öffnen von Projekten.

261 3 Variablen, Funktionen und Datentypen

262 3.1 Variablen beim Programmieren

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

```
alter <- 102
```

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

```
a <- 10
a
```

270 `## [1] 10`

271 Man kann mit `=` oder `<-` einer Variable einen Wert zuweisen. In dem meisten Fällen (alle, die wir abdecken)
272 funktioniert beides gleich, es wird aber empfohlen `<-` (`=` ist schlechter Stil) zu verwenden.

273 Wir können beliebige Variablen erstellen, z.B.

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

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

- 276 • `a_123 <- 10` ist ok
- 277 • `123_a <- 10` ist nicht ok

278 Vorsicht: Groß- und Kleinschreibung muss beachtet werden.

```

name <- "Johannes"
name

279 ## [1] "Johannes"

280 Das Aufrufen der Variablen

Name

281 führt zu einem Fehler.

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

a <- 10
b <- 5

a + b

284 ## [1] 15

b / a

285 ## [1] 0.5

a^b

286 ## [1] 1e+05

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

ergebnis <- a + b
ergebnis

288 ## [1] 15

ergebnis2 <- ergebnis * 2
ergebnis2

289 ## [1] 30

290 Mit der Funktion rm() können Variablen, die nicht mehr benötigte Variablen, wieder gelöscht werden. Al-
291 ternativ können Variablen auch überschrieben werden. Es gibt keine Möglichkeit gelöschte oder überschriebene
292 Variablen wiederherzustellen. Sie müssten ggf. neu berechnet werden.

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

293 ## [1] TRUE

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

294 ## [1] FALSE

```

295 3.2 Funktionen

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

```
298 sqrt(a)
```

298 `## [1] 3.162278`

299 Funktionen sind in R daran zu erkennen, dass dem Funktionsnamen runde Klammern () folgen. Im
300 vorherigen Beispiel wurde die Funktion mit dem Namen `sqrt()` aufgerufen. Das Objekt `a` haben wir bereits
301 vorhin definiert (zur Erinnerung `a <- 10`). Die Funktion `sqrt()` arbeitet jetzt mit dem Objekt `a`, das in
302 diesem Zusammenhang auch **Argument** genannt wird.

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

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

308 `## [1] 3.162278`

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

- 312 1. In die Konsole `?<Name der Funktion>` tippen. Also, wenn wir Hilfe für die Funktion `mean()` möchten,
313 könnten wir einfach `?mean` in die Konsole tippen.
 - 314 2. Analog zu 1), können wir mit der Funktion `help` die Hilfeseite für eine andere Funktion aufrufen (z.B.
315 wenn wir wieder die Hilfe für die Funktion `mean()` lesen möchten, dann könnten wir auch `help(mean)`
316 in die Konsole tippen).
 - 317 3. In RStudio kann man auch auf das Help-Panel klicken und dann einfach eine Funktion suchen (siehe
318 Abbildung 1).
 - 319 4. Wenn der Cursor in RStudio auf einer Funktion ist, kann man mit der Taste F1 die dazugehörige
320 Hilfeseite aufrufen.
- 321 Alle R Funktionen haben eine Hilfeseite. Diese Seite ist immer gleich aufgebaut und enthält die Kapitel
322 `Description`, `Usage`, `Arguments`, `Details`, `Value`, `Authors`, und `Examples`.

323 3.3 Datentypen

324 Es wurde bereits erwähnt, dass Daten in Variablen gespeichert werden können. Diese Variablen, in denen die
325 Daten (oder auch Berechnungen oder ganze Funktionen) gespeichert werden, heißen in R Objekte. Wenn Sie
326 beispielsweise Messwerte einer Fotofalle speichern möchten, dann hätte diese Fotofalle einen Namen (z.B.
327 `Kamera1`) und hoffentlich auch einige Fotos. Wir nehmen einmal an, dass nach drei Wochen 132 Fotos von
328 Rehen gemacht wurden. Wir können jetzt sowohl den Namen der Fotofalle, als auch die Anzahl Fotos die
329 aufgenommen wurden, in zwei Variablen abspeichern.

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

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

```
fuchs_gesehen <- TRUE
```

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

```
typeof(fuchs_gesehen)
```

341 ## [1] "logical"

342 `TRUE` wird im PC als 1 gespeichert und `FALSE` als 0. Es ist möglich mit `TRUE` und `FALSE` zu rechnen.

```
TRUE + TRUE
```

343 ## [1] 2

```
FALSE + FALSE
```

344 ## [1] 0

```
TRUE + FALSE
```

345 ## [1] 1

346 3.4 Datenstrukturen

347 Verfolgen wir das Beispiel mit den Fotofallen etwas weiter. Es handelt sich um ein systematisches Monitoring.
 348 D. h., es wurde nicht nur eine Fotofalle ausgebracht, sondern insgesamt 15 Fotofallen. Diese Erweiterung
 349 erfordert komplexere Objekte. Nachfolgend sind die Anzahl Rehfotos für jede der 15 Fotofallen aufgeführt:
 350 132, 79, 129, 91, 138, 144, 55, 103, 139, 105, 96, 146, 95, 118, 107.

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

²Für Interessierte, man unterscheidet weiter zwischen Ganzzahlen (`int`) und Gleitkommazahlen (`double`). Dieser Unterschied ist in R jedoch selten wichtig.

```
# 1. Kamera
name1 <- "Kamera_1"
anzahl_rehe1 <- 132
revier1 <- "Revier A"

# 2. Kamera
name2 <- "Kamera_2"
anzahl_rehe2 <- 79
revier2 <- "Revier A"

# usw.
```

- 356 Wenn wir so vorgehen würden, hätten wir 45 Objekte mit je einem Eintrag. Dieser Ansatz und führt schnell
 357 zu einem unübersichtlichen *Workspace*³. Wir werden im Verlauf sinnvollere Objekte (Vektoren oder Data
 358 Frames) für diesen Zweck kennenlernen.

359

360 **Aufgabe 2: Variablen**

- 362 Verwenden Sie die folgenden Daten

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

- 363 und berechnen sie:

- 364 • $10 * a$
 365 • $a / 144$ und speichern Sie das Ergebnis in einer neuen Variablen *e* zwischen.
 366 • Was ist das Ergebnis von $a + b$?
 367 • Was ist das Ergebnis von $a + p$?

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

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

368 4 Vektoren

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

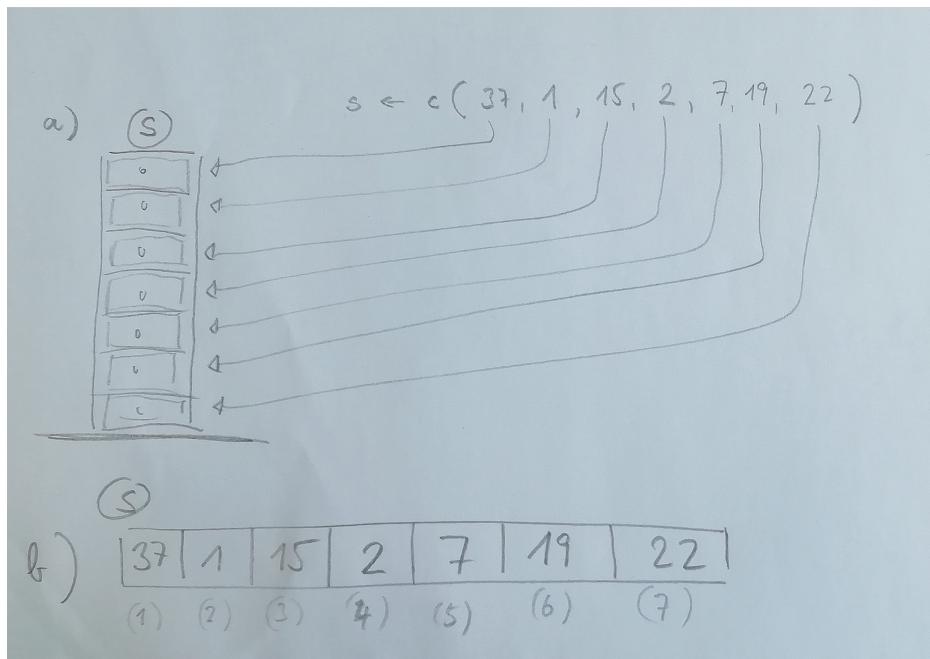


Abbildung 6: Schematische Darstellung eines Vektors in R.

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

378 Es gibt zahlreiche Funktionen zum Erstellen von Vektoren (einige davon werden wir im weiteren Verlauf des
 379 Moduls kennenlernen). Die wichtigste Funktion ist `combine c()` oder `concatenate` genannt. Die Funktion
 380 `c()` fügt einzelne Elemente in einen Vektor zusammen (und zwar in der Reihenfolge wie diese Elemente
 381 an `c()` übergeben werden). `c()` ist sozusagen die Funktion, die für Sie mehrere Elemente zu einem Vektor
 382 zusammensetzt. `c()` erwartet folglich nur Elemente als Argumente.

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

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

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

s388 `## [1] 37 1 15 2 7 19 22`

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

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

s + 10394 `## [1] 47 11 25 12 17 29 32`

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

s * s398 `## [1] 1369 1 225 4 49 361 484`

399 Neben der Funktion **c()** gibt es zahlreiche weitere Funktionen, um Vektoren zu erstellen. Wenn Sie einen
400 Vektor mit einer systematischen Zahlenfolge erstellen wollen, können Sie zum Beispiel die Funktion *sequence*
401 **seq()** verwenden. Im einfachsten Fall benötigt **seq()** zwei Argumente: **from** und **to**⁴.

seq(from = 1, to = 10)402 `## [1] 1 2 3 4 5 6 7 8 9 10`**(1 : 10)**403 `## [1] 1 2 3 4 5 6 7 8 9 10`

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

seq(from = 1, to = 10, by = 2)405 `## [1] 1 3 5 7 9`

406

Aufgabe 3: Vektoren erstellen

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

- 410 • Erstellen Sie einen Vektor mit dem Namen **bhd** in dem Sie die Werte speichern
- 411 • Transformieren Sie die BHD-Werte in mm.
- 412 • Berechnen Sie die Grundflächen der BHD in cm^2 (nehmen Sie dafür an, dass ein Baum kreisrund ist).

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

4.1 Funktionen zum Arbeiten mit Vektoren

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

```
head(s)
```

416 ## [1] 37 1 15 2 7 19

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

417 ## [1] 37 1 15

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

418 ## [1] 19 22

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

```
length(s)
```

420 ## [1] 7

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

```
class(s)
```

422 ## [1] "numeric"

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

```
unique(s)
```

424 ## [1] 37 1 15 2 7 19 22

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

```
table(s)
```

426 ## s

427 ## 1 2 7 15 19 22 37

428 ## 1 1 1 1 1 1 1

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

```
rev(s)
```

431 ## [1] 22 19 7 2 15 1 37

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

```
sort(s)
```

433 ## [1] 1 2 7 15 19 22 37

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

⁴³⁴ Die Funktion `rep()` wiederholt einen Vektor.

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

```
435 ## [1] 37 1 15 2 7 19 22 37 1 15 2 7 19 22
```

⁴³⁶ Anstelle des Arguments `times` kann auch das Argument `each` verwendet werden. Der Unterschied liegt darin,
⁴³⁷ dass `times` den gesamten Vektor `times`-Mal wiederholt und `each` jedes Element.

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

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

```
438 ## [1] 1 2 3 4 1 2 3 4
```

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

```
439 ## [1] 1 1 2 2 3 3 4 4
```

440

⁴⁴¹ Aufgabe 4: Arbeiten mit Vektoren

⁴⁴³ Es liegen jeweils zwei BHD-Messungen von vier Bäumen vor:

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

⁴⁴⁴ Sie haben jeden Baum je ein Mal mit dem Messgerät G1, dann mit dem Messgerät G2 gemessen. Erstellen Sie
⁴⁴⁵ einen Vektor von der Länge 8, in dem Sie angeben, welches Messgerät Sie verwendet haben.

⁴⁴⁶ 4.2 Statistische Funktionen

⁴⁴⁷ Zahlreiche statistische Funktionen können auf Vektoren angewendet werden, hier sind nur Beispiele aufgeführt.

```
sum(s)
```

```
448 ## [1] 103
```

```
mean(s)
```

```
449 ## [1] 14.71429
```

```
median(s)
```

```
450 ## [1] 15
```

```
sd(s)
```

```
451 ## [1] 12.76341
```

⁴⁵² Eine weitere, sehr häufig verwendete Funktion ist `sample()`. Mit `sample()` werden `size` Elemente zufällig
⁴⁵³ aus einem Vektor, mit oder ohne Zurücklegen (mit Zurücklegen wird gezogen, wenn das Argument `replace`
⁴⁵⁴ = TRUE gesetzt wird), gezogen.

```

sample(s, size = 1) # 1 Element
455 ## [1] 1
sample(s, size = 3) # 2 Elemente
456 ## [1] 15 7 22

```

457 Wenn `size` weggelassen wird, dann bekommt man gleich viele Elemente zurück (wie der Vektor lang ist), d.h.
 458 der Vektor wird nur zufällig neu arrangiert (permutiert).

459 4.3 Beispiel Fotofallen

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

463 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,
464           105, 96, 146, 95, 118, 1007)

```

465 Als zweites erstellen wir einen Vektor mit den IDs. Zur Erinnerung, diese sollten die Werte `Kamera_1` bis
 466 `Kamera_15` haben. Ein erster Ansatz könnte sein, dass wir einfach 15 Fotofallen schreiben und dann die
 467 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"
)

```

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

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

```

v1 <- rep("Kamera", 15)

```

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

```

v2 <- 1 : 15

```

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

```

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

```

```

477 ## [1] "Kamera_1"  "Kamera_2"  "Kamera_3"  "Kamera_4"  "Kamera_5"  "Kamera_6"
478 ## [7] "Kamera_7"  "Kamera_8"  "Kamera_9"  "Kamera_10" "Kamera_11" "Kamera_12"
479 ## [13] "Kamera_13" "Kamera_14" "Kamera_15"

```

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

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

```

482 ## [1] "Revier A" "Revier B" "Revier C" "Revier A" "Revier B" "Revier C"
483 ## [7] "Revier A" "Revier B" "Revier C" "Revier A" "Revier B" "Revier C"
484 ## [13] "Revier A" "Revier B" "Revier C"

```

485 Das Ergebnis stimmt noch nicht ganz, da wir 5 mal `Revier A` u.s.w. brauchen. Mit dem zusätzlichen Argument
486 `each = 5` können wir genau zu diesem Ergebnis kommen.

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

```

487 ## [1] "Revier A" "Revier A" "Revier A" "Revier A" "Revier A" "Revier B"
488 ## [7] "Revier B" "Revier B" "Revier B" "Revier B" "Revier C" "Revier C"
489 ## [13] "Revier C" "Revier C" "Revier C"

```

490

491 Aufgabe 5: Statistische Funktionen

- 493 1. Berechnen Sie den arithmetischen Mittelwert und Median für die Anzahl Fotos.
- 494 a) Verwenden Sie für die Berechnung des arithmetischen Mittelwerts die in R verfügbare Standardfunktion.
- 495 b) Schreiben Sie die arithmetische Mittelwertformel $\mu = \frac{1}{n} \sum_{i=1}^N x_i$ selbst als R Code und berechnen Sie
496 damit den arithmetischen Mittelwert der Anzahl Rehe.

497 2. Erstellen Sie die folgende Konsolenausgabe:

```
498 ## [1] "Die mittlere Anzahl von Rehfotos beträgt 171.8 Rehe pro Standort."
```

499 4.4 Arbeiten mit logischen Werten

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

- 502 • Gleichheit (`==`)
- 503 • Ungleichheit (`!=`)
- 504 • Größer (`>`) und kleiner (`<`)
- 505 • Größer gleich (`>=`) und kleiner gleich (`<=`)

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

507 Bei Vektoren kommt es immer (unabhängig von den Datentypen) zu einer elementweisen Anwendung. Wir
508 können beispielsweise abfragen, an welchen Fotofallenstandorten mehr als 100 Rehe fotografiert wurden:

```
anzahl_rehe > 100
```

```
509 ## [1] TRUE FALSE TRUE FALSE TRUE TRUE FALSE TRUE TRUE TRUE FALSE TRUE
510 ## [13] FALSE TRUE TRUE
```

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

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

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

```
513 ## [1] FALSE FALSE FALSE FALSE FALSE TRUE TRUE TRUE TRUE TRUE FALSE FALSE
514 ## [13] FALSE FALSE FALSE
```

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

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

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

```
521 ## [1] FALSE FALSE FALSE FALSE FALSE TRUE FALSE TRUE TRUE TRUE FALSE FALSE
522 ## [13] FALSE FALSE FALSE
```

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

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

```
526 ## [1] TRUE FALSE TRUE FALSE TRUE TRUE TRUE TRUE TRUE TRUE FALSE TRUE
527 ## [13] FALSE TRUE TRUE
```

528 Das Arbeiten mit logischen Werten kann fürs Erste etwas abstrakt erscheinen, aber wir werden im folgenden
529 Abschnitt (Abschnitt 4.5) zahlreiche Anwendungsbeispiele dafür sehen.

530

531 Aufgabe 6: Arbeiten mit logischen Werten

533 Überlegen Sie zunächst selbst, was das richtige Ergebnis ist und Überprüfen Sie dieses dann mit R.

- 534 1. TRUE | FALSE
- 535 2. FALSE & TRUE
- 536 3. (FALSE & TRUE) | TRUE
- 537 4. (2 != 3) | FALSE
- 538 5. FALSE + 10
- 539 6. TRUE + 10

```
540 7. TRUE + 10 == FALSE + 10
541 8. sum(c(TRUE, TRUE, FALSE, FALSE))
```

542 4.5 Zugreifen auf Elemente eines Vektors (=Untermengen)

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

547 Bei Vektoren kann auf die einzelnen Elemente mit eckigen Klammern ([]), diese werden auch Indizierungs-
 548 klammern genannt) zugegriffen werden. Der Ausdruck `anzahl_rehe[2]` gibt die Anzahl an fotografierten
 549 Rehen für die zweite Fotofalle zurück, also die zweite Stelle des Vektors `anzahl_rehe`. Es gibt zwei Möglich-
 550 keiten, was in die eckigen Klammern geschrieben werden kann: 1.) die Positionen der Elemente die indiziert
 551 werden sollen. Ist es mehr als ein Element, dann muss ein Vektor mit den Positionen übergeben werden, also z.
 552 B. `anzahl_rehe[c(1, 2)]`. Die 2.) Möglichkeit der Indizierung ist ein logischer Vektor in der gleichen Länge
 553 des Vektors. Es werden alle Elemente zurückgegeben, bei denen in dem logischen Vektor TRUE eingetragen ist.

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

```
anzahl_rehe[2]
```

```
555 ## [1] 79
```

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

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

```
557 ## [1] 132 79 129 91 138
```

```
anzahl_rehe[reviere == "Revier A"] # Auch als Einzeiler möglich.
```

```
558 ## [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.
```

```
559 ## [1] 132 79 129 91 138
```

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

562

563 Aufgabe 7: Zugreifen auf Vektorelemente

565 Erstellen Sie einen neuen Vektor `bhd`

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

- 566 • Wählen Sie aus dem Vektor `bhd` nur das 2. und 3. Element aus.

- 567 • Wählen Sie aus dem Vektor `bhd` das letzte Element aus.
 568 • Wählen Sie aus dem Vektor `bhd` das letzte Element aus, ohne die Zahl 6 zu schreiben.
 569 • Wie viele BHDs aus dem Vektor `bhd` sind größer oder gleich 30?.

570

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

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

573 ## [1] 132 79 129 91 138

574 Das Erstellen des `sub`-Vektors ist mühsam und wenig zielführend. Wenn wir auf die Erkenntnisse aus dem
 575 vorherigen Kapitel zurückgreifen, kann dies leicht automatisiert werden, indem wir einfach abfragen, welche
 576 Elemente in Revier zu `Revier A` gehören. Diese Verbindung von logischen Operatoren und Indizierung ist
 577 sehr relevant. Es ist eine der wichtigsten Tätigkeiten beim Programmieren mit R.

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

578 ## [1] 132 79 129 91 138

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

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

581 ## [1] 132 79 129 91 138

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

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

584 ## [1] 113.8

585

586 Aufgabe 8: logische Werte

588 Verwenden Sie nochmals den Vektor mit den Anzahl Rehen die an unterschiedlichen Fotofallenstandorten
 589 fotografiert wurden.

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

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

593 4.6 Der %in%-Operator

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

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

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

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

```
598 ## [1] "FI" "FI"
# oder
messungen_arten[messungen_arten == arten[1]]
```

```
599 ## [1] "FI" "FI"
```

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

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

```
602 ## [1] "FI" "BU" "BU" "FI"
```

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

```
messungen_arten %in% arten
```

```
606 ## [1] TRUE TRUE TRUE FALSE FALSE FALSE TRUE FALSE FALSE
```

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

```
607 ## [1] "FI" "BU" "BU" "FI"
```

608

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

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

```
LETTERS
```

```
612 ## [1] "A" "B" "C" "D" "E" "F" "G" "H" "I" "J" "K" "L" "M" "N" "O" "P" "Q" "R" "S"
```

```
613 ## [20] "T" "U" "V" "W" "X" "Y" "Z"
```

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

615 5 Faktoren (factors)

616 R besitzt einen besonderen Datentyp – *Faktoren* (engl. factors) – zum speichern von diskreten Kovariaten
 617 (z.B. Baumart, Augenfarbe oder Automarke). Faktoren erlauben es, Daten vom Typ **character** effizienter
 618 abzuspeichern. Denn im Gegensatz zu **character** enthalten Faktoren mehr Funktionalitäten, die bei der
 619 Datenverarbeitung aber auch bei der statistischen Datenanalyse hilfreich sind. Die Unterscheidung von
 620 Faktoren und reinem Text ist auf die historische Entwicklung von R als statistischer Programmiersprache
 621 zurückzuführen. Faktoren sind Text und statistische Information in einem. Fließtext, der z. B. ein Experiment
 622 beschreibt oder Kommentare aus dem Aufnahmebogen enthält, sollte in R besser als **character** gespeichert
 623 werden, wohingegen relevante statistisch Informationen, wie z. B. *Kontrollgruppe* oder auch unsere *Fotofalle*
 624 besser als **factor** vorliegen sollten. In Faktoren wird jeder eindeutige Wert (=Level) mit einer Zahl codiert
 625 und dann werden nur diese Zahlen zusammen mit einer Tabelle zum Nachschauen der Werte gespeichert
 626 (siehe dazu auch [McNamara and Horton 2018](#)). Faktoren haben im Gegensatz zu Zeichenketten zusätzliche
 627 Eigenschaften. Man kann sie z. B. sortieren.

628 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")
```

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

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

632 ## [1] FI BU FI EI EI FI FI

633 ## Levels: FI BU EI

```
a
```

634 ## [1] "FI" "BU" "FI" "EI" "EI" "FI" "FI"

635 Es ist auch möglich, die Beschriftung (= labels) der unterschiedlichen Levels anzugeben mit dem Argument
 636 **labels**. Das macht z. B. Sinn, wenn sie lange Beschriftungen haben, die Sie beim Programmieren nicht
 637 ständig vollständig abtippen möchten.

```
af <- factor(a, levels = c("FI", "BU", "EI"),
             labels = c("Picea abies", "Fagus sylvatica", "Quercus spp."))
af
```

638 ## [1] Picea abies Fagus sylvatica Picea abies Quercus spp.

639 ## [5] Quercus spp. Picea abies Picea abies

640 ## Levels: Picea abies Fagus sylvatica Quercus spp.

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

```
levels(af)
```

643 ## [1] "Fichte" "Buche" "Eiche"

```

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

644 ## [1] Fi Bu Fi Ei Ei Fi Fi
645 ## Levels: Fi Bu Ei

646 Schlussendlich kann man mit der Funktion relevel() die Referenzkategorie eines Faktors (der erste Level)
647 angepasst werden. Dies ist z. B. für lineare Regressionsmodelle relevant.

af

648 ## [1] Fi Bu Fi Ei Ei Fi Fi
649 ## Levels: Fi Bu Ei

relevel(af, "Bu")

650 ## [1] Fi Bu Fi Ei Ei Fi Fi
651 ## Levels: Bu Fi Ei

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

as.character(af)

654 ## [1] "Fi" "Bu" "Fi" "Ei" "Ei" "Fi" "Fi"
655 Ebenso funktioniert as.factor(). Mit der Funktion as.numeric() erhält man die interne Kodierung von
656 Faktoren.

af

657 ## [1] Fi Bu Fi Ei Ei Fi Fi
658 ## Levels: Fi Bu Ei

as.numeric(af)

659 ## [1] 1 2 1 3 3 1 1
660 Fichte ist das erste Level des Faktors, deshalb erhalten alle Fichteneinträge den Wert 1. Bucheinträge erhalten
661 den Wert 2 und Eichen 3.

662

663 Aufgabe 10: Faktoren


---


664

665 Verwenden Sie den Vektor staedte und erstellen Sie einen Vektor mit der Anordnung der levels in
666 umgekehrter alphabetischer Reihenfolge.

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

```

667 5.1 Das Paket `forcats`

668 Das Paket `forcats` enthält erweiterte Funktionalitäten zu Faktoren. Sie müssen das Paket installieren und
669 laden (aktivieren). Wir kommen später noch auf Pakete zurück.

670 Mit dem Paket aus `forcats` werden einige Operationen mit Faktoren erleichtert. Wir schauen uns hier
671 Funktion an, die es erleichtern:

- 672 1. Die Anordnung von Levels anzupassen.
- 673 2. Levels zusammenzufassen oder zu entfernen.
- 674 3. Labels zu ändern.

675 5.1.1 Anpassen der Anordnung von Faktoren

676 Wir erstellen nochmals den `a` Vektor mit unterschiedlichen Baumarten:

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

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

```
factor(a)
```

```
678 ## [1] FI BU FI EI EI FI FI  
679 ## Levels: BU EI FI
```

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

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

```
681 ## [1] FI BU FI EI EI FI FI  
682 ## Levels: FI BU EI
```

683 `forcats` stellt u. a. Funktionen zur Verfügung, um die Reihenfolge umzukehren,

```
fct_rev(f1)
```

```
684 ## [1] FI BU FI EI EI FI FI  
685 ## Levels: EI BU FI
```

686 nach der Häufigkeit zu sortieren oder

```
fct_infreq(f1)
```

```
687 ## [1] FI BU FI EI EI FI FI  
688 ## Levels: FI EI BU
```

689 zufällig zu sortieren.

```
fct_shuffle(f1)
```

```
690 ## [1] FI BU FI EI EI FI FI  
691 ## Levels: EI FI BU
```

692

693 **Aufgabe 11: *forcats***

695 Sehen Sie sich das *Cheatsheet* zum Paket **forcats** an. <https://raw.githubusercontent.com/rstudio/cheatsheets/main/factors.pdf> Cheatsheets sind 1- bis 2-seitige PDFs, die einen Schnellüberblick über Funktionen
696 eines Pakets liefern. Sie folgen keinem strengen Aufbau, wie die Hilfeseiten, sondern können je Paket sehr
697 individuell sein. Verwenden Sie nochmals den Baumartenvektor **a**.

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

- 699 1. Wandeln Sie **a** in einen Faktorvektor um.
700 2. Fassen Sie die Faktorlevels **BU** und **EI** zu **Laub** zusammen und benennen Sie **FI** in **Nadel** um.

701 6 Spezielle Einträge

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

- 703 • fehlenden Einträgen NA,
- 704 • leeren Einträgen NULL,
- 705 • undefinierten Einträgen NaN (Not a Number) oder
- 706 • unendlichen Zahlen (Inf).

707 Spezielle Einträge sind reservierte Namen. Sie können nicht überspeichert werden. Z. B. NA <- 1 geht also
708 nicht.

709 6.1 NA

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

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

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

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

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

```
is.na(na1)

## [1] FALSE TRUE FALSE
```

```
na.omit(na1)

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

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

```
na2 <- 3
```

727 ## [1] FALSE FALSE NA

1 + NA

728 `## [1] NA`
 729 Viele R Funktionen haben eingebaute Methoden zum Umgang mit NA. Die Funktion `mean()` bspw. ergibt
 730 (wie die meisten Funktionen) standardmäßig NA wenn sie auf Vektoren mit Datenlücken angewendet wird, es
 731 sei denn man stellt innerhalb der Funktion ein, dass Datenlücken entfernt werden sollen.

mean(na2)

732 `## [1] NA`
`mean(na2, na.rm = TRUE)`
 733 `## [1] 4.5`

6.2 NULL

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

6.3 Inf

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

10^309

742 `## [1] Inf`
`2 * Inf`
 743 `## [1] Inf`
`1 + Inf`
 744 `## [1] Inf`
`3 / 0`
 745 `## [1] Inf`
`-3 / 0`
 746 `## [1] -Inf`
`3 / Inf`
 747 `## [1] 0`

748 Infinity kann mit `is.infinite` und `is.finite` getestet werden. Relationäre Operatoren (`<`, `>`) funktionieren
 749 erwartungsgemäß.

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

750 ## [1] TRUE FALSE FALSE TRUE FALSE
is.finite(inf1)

751 ## [1] FALSE TRUE TRUE FALSE TRUE
inf1 < 3

752 ## [1] FALSE TRUE FALSE TRUE FALSE
```

753

754 **Aufgabe 12: Vektoren mit speziellen Einträgen**

756 Verwenden Sie den Vektor

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

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

759 Verwenden Sie den Vektor

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

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

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

765 Berechnen Sie den arithmetischen Mittelwert von `foo`.

7 data.frames oder Tabellen

766 Im vorherigen Teilabschnitt haben wir gesehen, wie mehrere Elemente des gleichen Datentyps in einem Vektor
 767 zusammengefasst werden können. Abschließend wurde anhand des Fotofallenbeispiels gezeigt, wie Vektoren
 768 eingesetzt werden können, um auch andere Vektoren zu indizieren. Wir erstellten drei Vektoren, die jeweils die
 769 Merkmalsausprägungen eines Merkmals aller Fotofallenstandorte speichern und dementsprechend gleich lang
 770 sind. In statistischer Sprache, sind die Fotofallen die Beobachtungen (oder auch Merkmalsträger genannt) und
 771 die Informationen zu den Fotofallen (also ID, Anzahl Rehe und das Revier) die Merkmale. Jeder beobachtete
 772 Wert (z.B. die 132 fotografierten Rehe von Kamera 1) ist dann eine Merkmalsausprägung. Dieser Datenaufbau
 773 ist für R typisch (man sagt auch *Tidy Data* zu diesem Datenformat). Ihre Daten sollten genau so vorliegen.
 774 Sie haben sich vermutlich schon gedacht, dass die Daten jedoch nicht als einzelne Vektoren, sondern als eine
 775 Tabelle vorliegen. In R ist diese Tabelle der **Data Frame**.
 776

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

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

```
782 ##           ID anzahl_rehe   revier
783 ## 1    Kamera_1        132 Revier A
784 ## 2    Kamera_2         79 Revier A
785 ## 3    Kamera_3        129 Revier A
786 ## 4    Kamera_4         91 Revier A
787 ## 5    Kamera_5        138 Revier A
788 ## 6    Kamera_6        144 Revier B
789 ## 7    Kamera_7         55 Revier B
790 ## 8    Kamera_8        103 Revier B
791 ## 9    Kamera_9        139 Revier B
792 ## 10  Kamera_10       105 Revier B
793 ## 11  Kamera_11       96 Revier C
794 ## 12  Kamera_12       146 Revier C
795 ## 13  Kamera_13       95 Revier C
796 ## 14  Kamera_14      118 Revier C
797 ## 15  Kamera_15      107 Revier C
```

798 Auf der linken Seite der Gleichungen stehen die Spaltenname des Data Frames. Im vorhergehenden Codebeispiel
 799 wurde ein **data.frame** erstellt und als Variable **monitoring** gespeichert. Die Funktion **data.frame()** nimmt

800 als Argumente beliebig viele Paare, die immer aus einem Namen und einem Vektor mit dazugehörigen
 801 Werten bestehen. D. h., dass (wie bei den Vektoren) immer eine Spalte vom selben Typ sein muss, es aber
 802 für jede Beobachtung (=Zeile) Merkmale von unterschiedlichen Typen geben kann. Data Frames sind die
 803 Standard-Objekte zum Speichern (wissenschaftlicher) Daten.

804 7.1 Wichtige Funktionen zum Arbeiten mit `data.frames`

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

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

```
807 ##           ID anzahl_rehe   revier
808 ## 1 Kamera_1          132 Revier A
809 ## 2 Kamera_2          79 Revier A
```

810 Oder für die letzten 2 Beobachtungen.

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

```
811 ##           ID anzahl_rehe   revier
812 ## 14 Kamera_14         118 Revier C
813 ## 15 Kamera_15         107 Revier C
```

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

```
nrow(monitoring)
```

```
815 ## [1] 15
```

```
ncol(monitoring)
```

```
816 ## [1] 3
```

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

```
str(monitoring)
```

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

```
823
```

824 Aufgabe 13: `data.frame`

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

829 7.2 Zugreifen auf Elemente eines `data.frame`

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

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

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

```
838 ## [1] 91
```

839 Alternativ, kann man den Spaltennamen auch Ausschreiben. Dies hat beim Programmieren den Vorteil, dass
840 der Code lauffähig bleibt, falls sich die Reihenfolge der Spalten in der Datengrundlage ändern sollte. Nachteil
841 ist entsprechend, dass der Code nicht mehr laufen würde, falls die Variablennamen sich ändern.

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

```
842 ## [1] 91
```

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

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

```
845 ## [1] 132 79 129 91 138
```

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

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

```
848 ##   anzahl_rehe  revier
849 ## 1      132 Revier A
850 ## 2      79 Revier A
851 ## 3      129 Revier A
852 ## 4      91 Revier A
853 ## 5      138 Revier A
```

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

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

```
856 ##           ID anzahl_rehe  revier
857 ## 1 Kamera_1      132 Revier A
858 ## 2 Kamera_2      79 Revier A
859 ## 3 Kamera_3      129 Revier A
```

```
860 ## 4 Kamera_4          91 Revier A
861 ## 5 Kamera_5          138 Revier A
```

862

863 Aufgabe 14: Abfragen von Werten

865 Wir nehmen folgende Werte aus Übung 13 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)
)
```

- 866 • Wählen Sie nur die ersten drei Zeilen aus und die erste und zweite Spalte aus.
 867 • Wählen Sie alle Zeilen und die erste und dritte Spalte aus.
 868 • Wählen Sie alle Spalten und die erste, dritte und vierte Zeile aus.

869

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

- 872 über das \$-Zeichen direkt die Spalte ansprechen. Diese Möglichkeit hat den Vorteil, dass R Studio den
 873 Spaltennamen automatisch ausfüllen kann. Beim Tippen werden mögliche Spaltennamen vorgeschlagen.
 874 Sie wählen den Vorschlag aus, in dem Sie Tabulator (\rightarrow) drücken. Danach können Sie diesen
 875 resultierenden Vektor wie einen einfachen Vektor behandeln und verarbeiten.

```
monitoring$anzahl_rehe
```

```
876 ## [1] 132 79 129 91 138 144 55 103 139 105 96 146 95 118 107
```

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

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

```
878 ## [1] 132 79 129 91 138 144 55 103 139 105 96 146 95 118 107
```

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

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

```
880 ## [1] 132 79 129 91 138 144 55 103 139 105 96 146 95 118 107
```

881 Anmerkung zu 3), der Ausdruck `1:nrow(monitoring)` ergibt einen Vektor mit den Zahlen 1 bis 15, da
 882 `nrow(monitoring) = 15` ist. Diese Schreibweise ist zu empfehlen, wenn die Dimension des Vektors variabel
 883 ist. Merken Sie sich diese Kombination aus Befehlen. Auf ähnliche weise können Sie vom Ende oder von
 884 Anfang variable Längen indizieren. Das ist z. B. nützlich, wenn Sie $n - 1$ Einträge indizieren möchten.

885 Schlussendlich kann man einen `data.frame` genauso mit logischen Vektoren abfragen, wie einen Vektor. Ein
 886 Beispiel wäre, wenn wir alle Fotofallen abfragen möchten, die mehr als 100 Rehfotos gemacht haben. Der

887 erste Schritt wäre abzufragen, ob eine Fotofalle mehr als 100 Rehfotos gemacht hat.

```
monitoring$anzahl_rehe > 100
```

```
888 ## [1] TRUE FALSE TRUE FALSE TRUE TRUE FALSE TRUE TRUE TRUE FALSE TRUE  
889 ## [13] FALSE TRUE TRUE
```

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

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

```
894 ##          ID anzahl_rehe    revier  
895 ## 1   Kamera_1           132 Revier A  
896 ## 3   Kamera_3           129 Revier A  
897 ## 5   Kamera_5           138 Revier A  
898 ## 6   Kamera_6           144 Revier B  
899 ## 8   Kamera_8           103 Revier B  
900 ## 9   Kamera_9           139 Revier B  
901 ## 10  Kamera_10          105 Revier B  
902 ## 12  Kamera_12          146 Revier C  
903 ## 14  Kamera_14          118 Revier C  
904 ## 15  Kamera_15          107 Revier C
```

905

906 Aufgabe 15: Abfragen von Werten 2

908 Verwenden Sie erneut den Datensatz aus Übung 14 und führen Sie folgende Abfragen durch:

- ```
909 • Alle Spalten für Studierende die Forstwissenschaften studieren.
910 • Alle Spalten für Studierende die Chemie oder Physik studieren.
911 • Die Spalte fach und semester für Studierende die 22 oder älter sind.
```

## 8 Schreiben und lesen von Daten

### 8.1 Textdateien

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

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

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

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

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

```
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
```

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

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

948

949 **Aufgabe 16: Lesen und Schreiben von Datein**

---

950

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

## 9 Erstellen von Abbildungen

954 Abbildungen sind ein elementarer Baustein statistischer Analysen und deshalb von Beginn an Teil von R. **R is  
955 a free software environment for statistical computing and graphics.** Es gibt unterschiedliche Systeme  
956 einen Plot zu erstellen. In diesem Kurs werden wir kurz *Base Plots* vorstellen und dann das Zusatzpaket  
957 **958 ggpplot2** vorstellen.

### 959 9.1 Base Plot

960 Die wichtigsten Grafiken für die einfache Datendarstellung sind schnell verfügbar. Etwas komplexere oder  
961 spezielle Grafiken erfordern mehr Programmieraufwand (folgt teilweise noch). Für viele komplexere Diagramme  
962 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  
963 sie sich die einfache Grafik Schnittstelle (**base plots**) als zweidimensionale Leinwand vor, auf die Sie durch  
964 Code Ebene für Ebene Grafikelemente legen:  
965

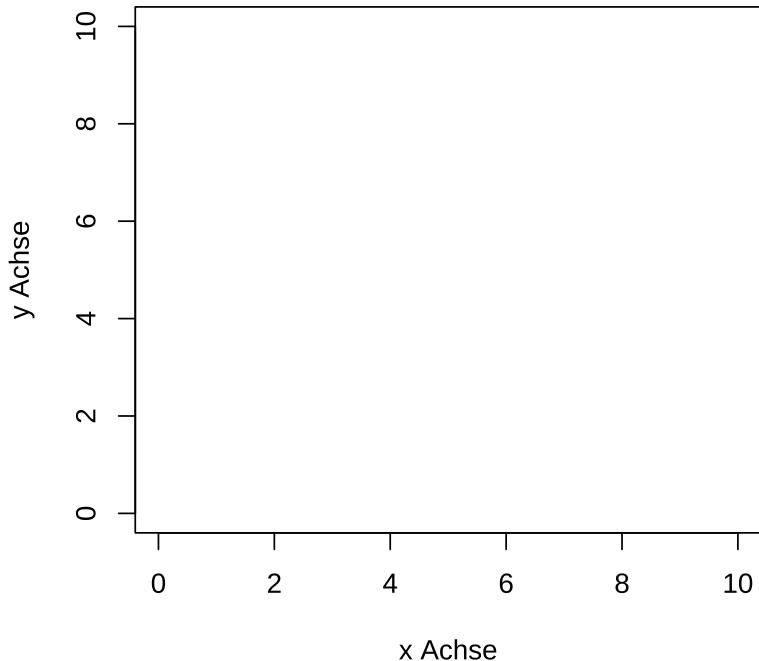
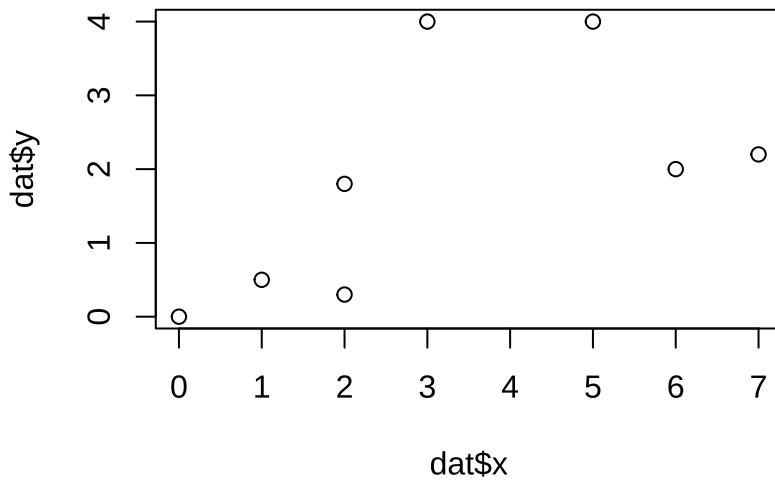


Abbildung 7: Beispiel einer leeren Grafikschnittstelle.

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

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

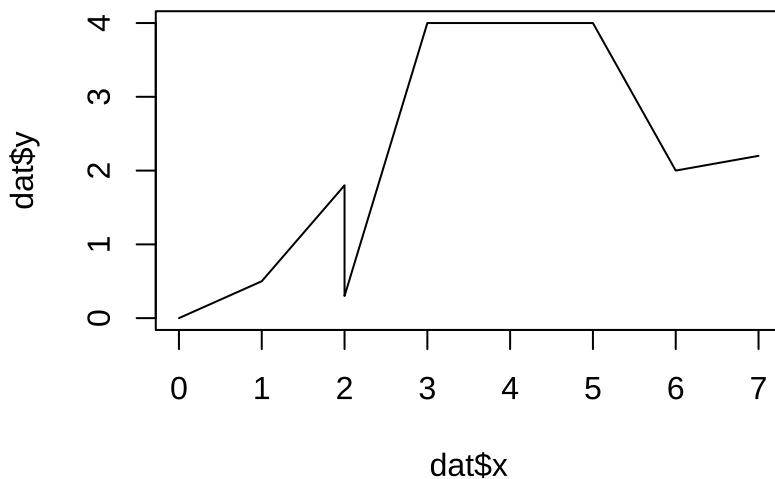
plot(datx, daty, type ="p")
```



967

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

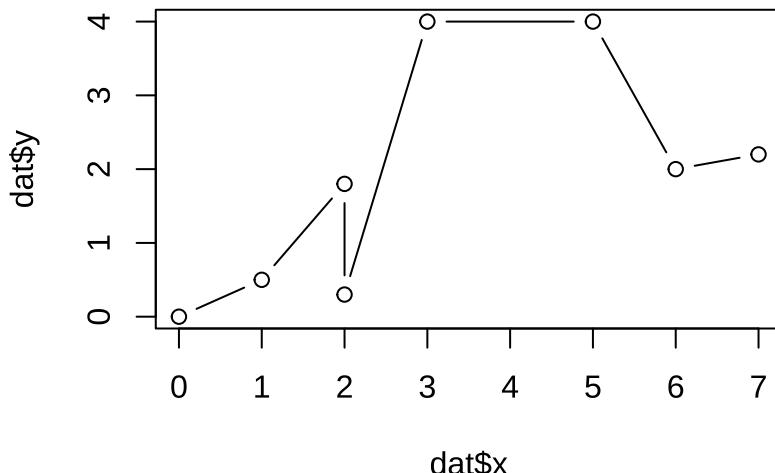
```
plot(datx, daty, type = "l")
```



970

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

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



972

973 darstellen.

974

975 **Aufgabe 17: Base Plot 1**

976

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

979

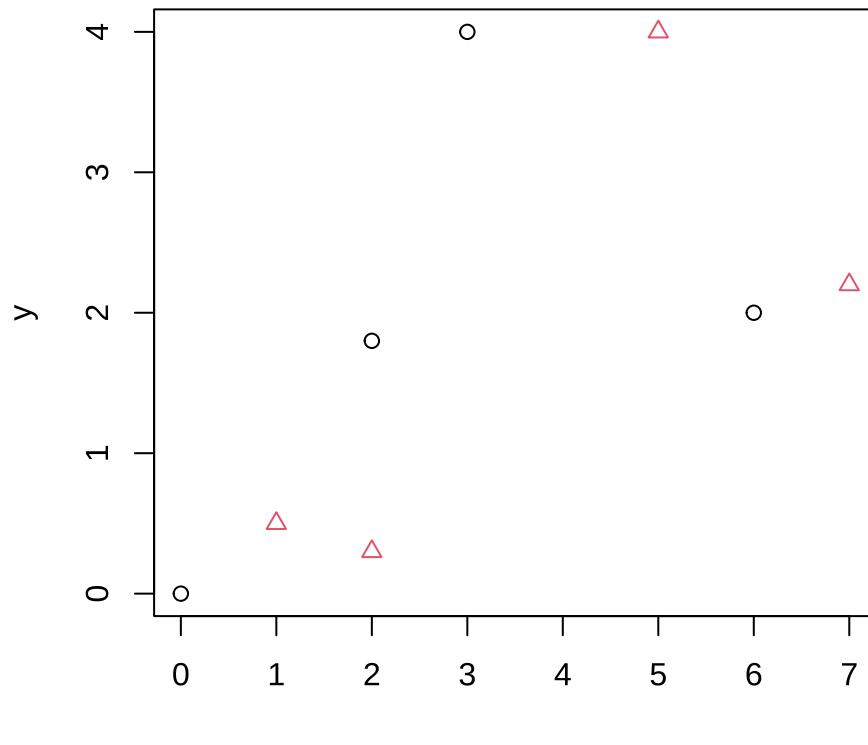
980 Sie können entweder eine Grafik mit einem Befehl erzeugen (High-Level) oder die einzelnen Ebenen nacheinander erzeugen (Low-Level). Sie können jeder Ebene durch zusätzliche Befehle innerhalb des Funktionsaufrufs 981 Elemente hinzufügen. Mit jeder neuen Ebene können Sie die vorigen Ebenen jedoch nicht mehr verändern.  
 982 Elemente hinzufügen. Mit jeder neuen Ebene können Sie die vorigen Ebenen jedoch nicht mehr verändern.  
 983 Die wichtigsten Argumente der `plot` Funktion sind:

- 984 • `type` - Diagrammtyp
- 985 • `col` - Farbe
- 986 • `main` - Titel
- 987 • `sub` - Untertitel
- 988 • `pch` - Punktsymbol
- 989 • `lty` - Linientyp
- 990 • `lwd` - Linienstärke
- 991 • `xlab` bzw. `ylab` - Achsenbeschriftungen
- 992 • `xlim`, `ylim` - Grenzen der Achsenanschnitte
- 993 • `axes` - Sollen die Achsen eingezeichnet werden? Oder leer gelassen werden, um sie nachträglich als  
 994 low-level Ebene einzuziehen?
- 995 • `ann` - Achsenbeschriftung kann ganz weggelassen werden.

996 Sehen Sie sich die Hilfeseiten `?plot.default()` oder `?par()` an für weitere Informationen. Dort finden Sie  
 997 auch eine vollständige Liste der Befehle. Einige Argumente können als Vektor übergeben werden. Hier z. B.  
 998 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")
```



999

1000

---

**Aufgabe 18: Anpassen von Plots**


---

1003 Verwenden Sie den Datensatz aus Übung 17 und passen Sie die Abbildung wie folgt an:

- 1004 • Beschriften Sie die x- und y-Achse sinnvoll.
- 1005 • Fügen Sie eine Überschrift hinzu.
- 1006 • Wählen Sie ein anderes Symbol.
- 1007 • Stellen Sie die Symbole in rot dar.

1008

1009 Über Low-Level Funktionen können einer Grafik Schnittstelle nacheinander Elemente hinzugefügt werden.

1010 Die wichtigsten Funktionen sind

- 1011 • `points()` - Fügt Punkte ein
- 1012 • `lines()` - Fügt Linien ein

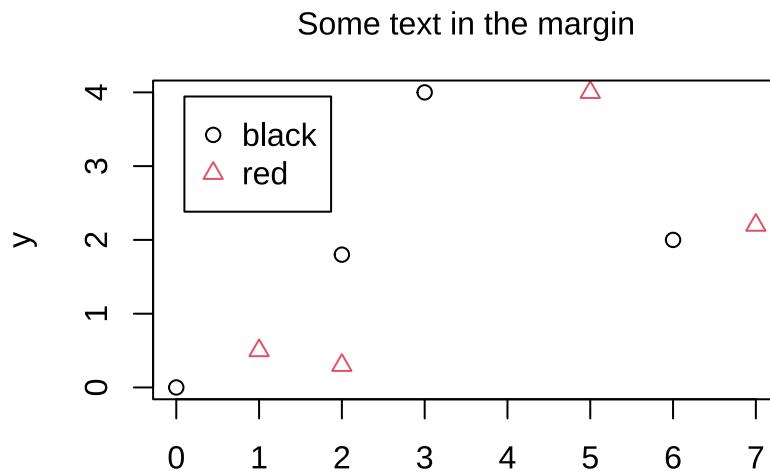
- `text()` - Fügt Text ein
- `mtext` - Fügt Text in den Rahmen (`margin`) ein
- `legend()` - Fügt eine Legende ein
- `abline()` - Fügt eine Gerade ein
- `curve()` - Fügt eine mathematische Funktion ein
- `arrows()` - Fügt Pfeile ein
- `grid()` - Fügt Hilfslinien ein

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

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



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

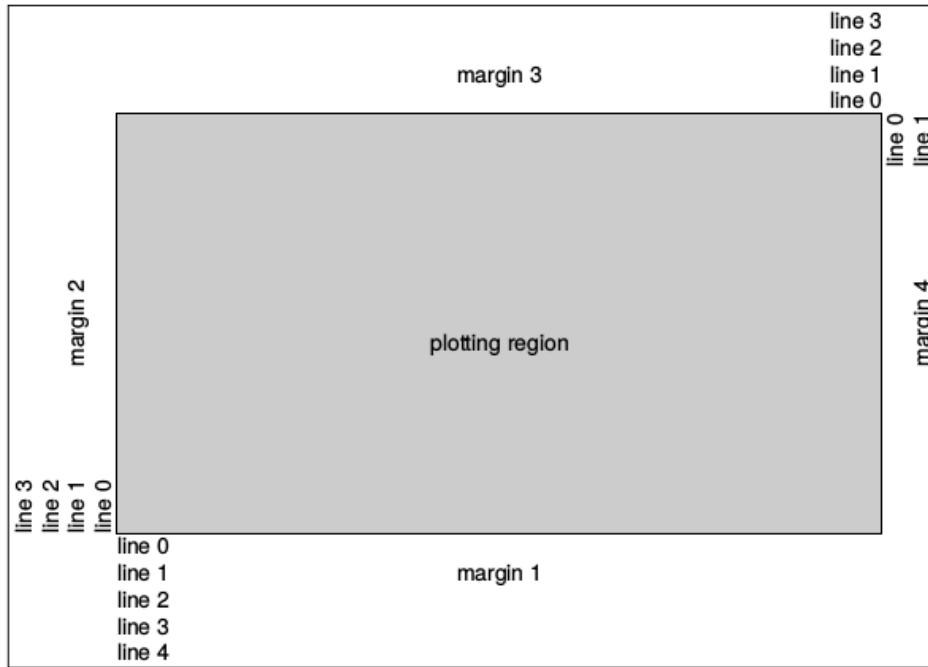


Abbildung 8: Grafikregionen eines base plots in R.

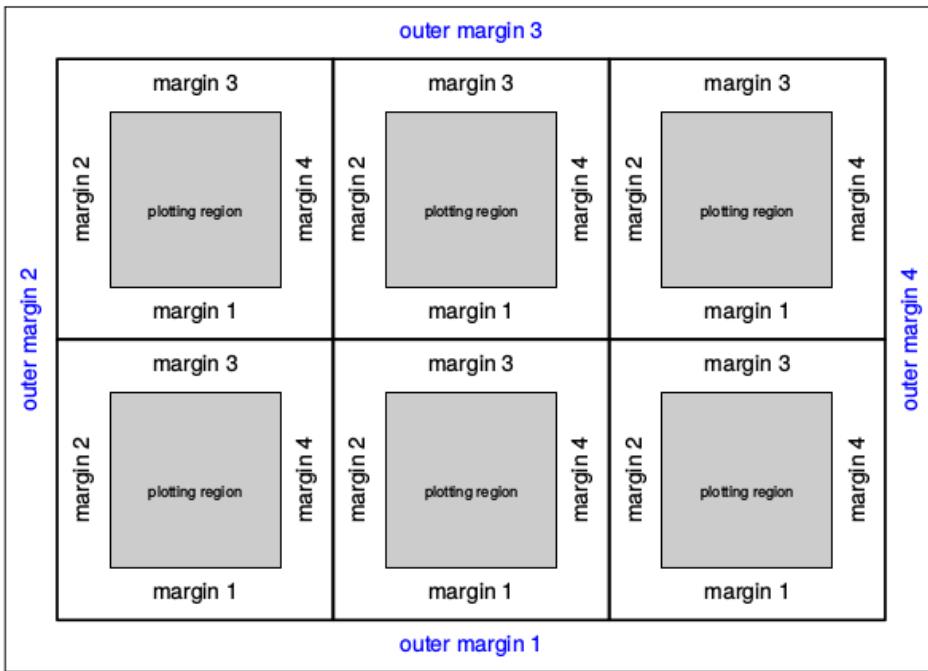


Abbildung 9: Schematischer Aufbau mehrerer Diagramme in einem plot am Beispiel einer 3 x 2 Grafik.

1031 **9.1.1 Mehrere Panels**

1032 Mit der Funktion `par()` kann auch eingestellt werden, dass ein Plot aus mehreren Subplots (= Panels)  
 1033 besteht. Die Argumente `mfrow` und `mfcol` können `par()` übergeben werden und kontrollieren die Anzahl  
 1034 Zeilen und Spalten für den Plot.

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

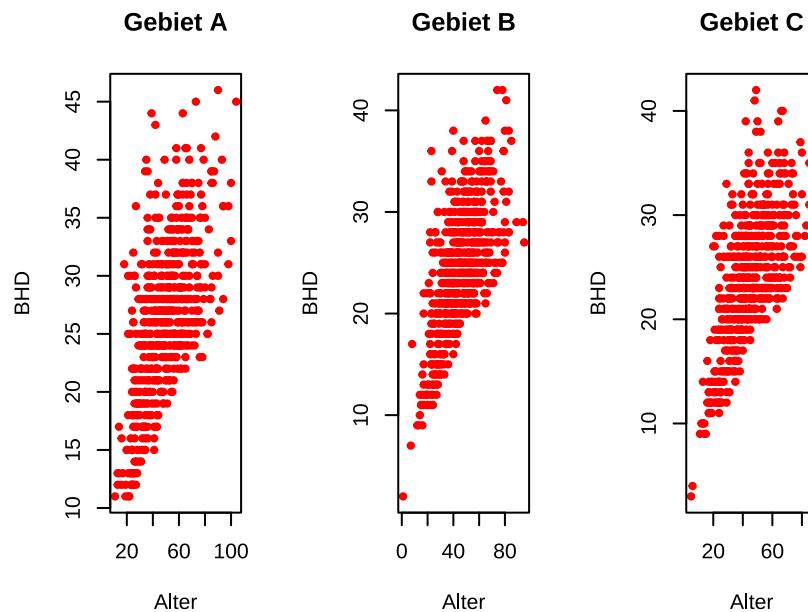
1035 Teilt den Plot in eine Zeile und drei Zeilen (= drei Plots nebeneinander).

```
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")
```



1036

1037 Vergessen Sie nicht am Ende nochmals `par(mfrow = c(1, 1))` zu setzen, damit wieder nur ein Plot angezeigt  
 1038 wird.

1039 **9.1.2 Speichern von Abbildungen**

1040 Wenn nicht anders angegeben, wird die Abbildung zunächst nur in der RStudio Grafik Schnittstelle abgebildet  
 1041 (Pane 4). Von dort aus kann die Abbildung exportiert werden. Es bietet sich jedoch an das Speichern der  
 1042 Abbildung direkt im Code zu programmieren. Mögliche Formate die Abbildung als Vektorgrafik zu speichern  
 1043 sind

- 1044     • `pdf()` oder  
 1045     • `postscript()`.

1046 Beispiele für Rastergrafiken sind

- 1047     • `png()`,  
 1048     • `bmp()` oder  
 1049     • `jpeg()`.

1050 Die Grafikschnittstelle ist dann Ihre “Leinwand”. Mit dem Befehl `dev.off()` trennen Sie die Verbindung zur  
 1051 Schnittstelle wieder. Ihre “Leinwand” wird also wieder geschlossen. So lange die Schnittstelle geöffnet ist  
 1052 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
```

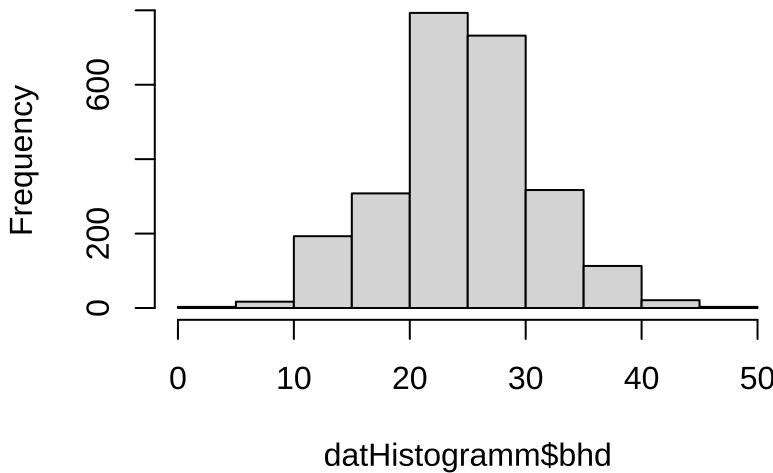
1053 *Achtung*, wenn Sie die Funktion `dev.off()` nicht aufrufen, werden alle nachfolgenden Plots in die gleiche  
 1054 Datei geschrieben. Falls Sie nach einem Versuch einen Plot zu speichern plötzlich keine weiteren Plots mehr  
 1055 sehen, führen Sie einige Mal die Funktion `dev.off()` aus.

## 1056 9.2 Histogramme

1057 Neben den Streudiagrammen (*Scatterplots*, oder auch einfach x-y Diagramm) sind *Histogramme* in der  
 1058 angewandten Datenanalyse ein weiterer wichtiger Abbildungstyp. An Histogrammen wird die Häufigkeit  
 1059 von Beobachtungen nach Gruppen dargestellt. Sie sind deshalb so wichtig, weil man aus ihnen relevante  
 1060 Informationen über die Verteilung der Daten ablesen kann und somit einiges über die Messungen erfährt,  
 1061 das für die weitere Datenanalyse relevant sein kann. So werden auf einen Blick der Zusammenhang von  
 1062 Beobachtungshäufigkeit und Streuung deutlich, sowie auch die Form der Verteilung und ihre Schiefe. Die  
 1063 Interpretation werden wir bei den Boxplots noch weiter vertiefen.

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

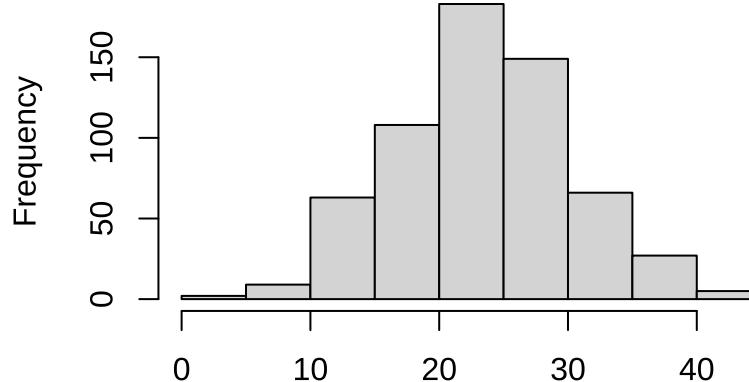
### Histogramm of datHistogramm\$bhd



1064

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

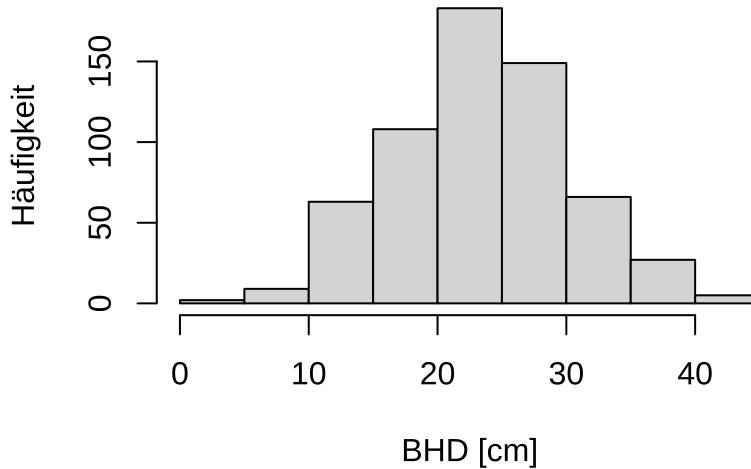
### Histogramm of datHistogramm\$bhd[datHistogramm\$art == "EI"]



1065

```
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

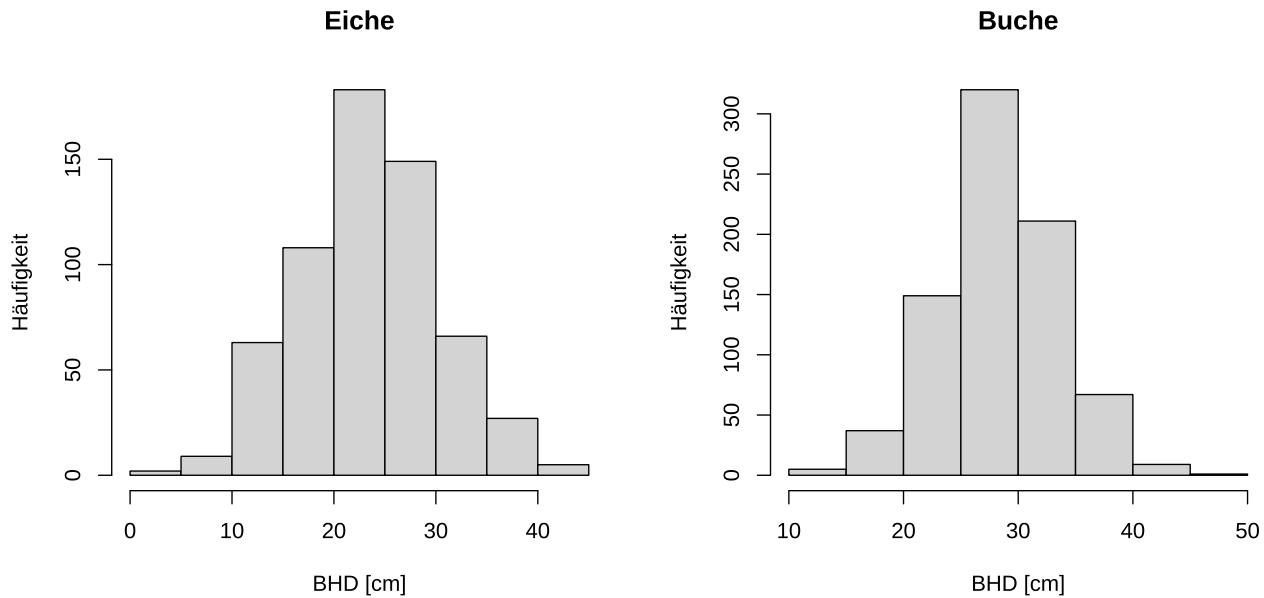


1066

1067 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")
```

1068



1069

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

### 9.3 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.

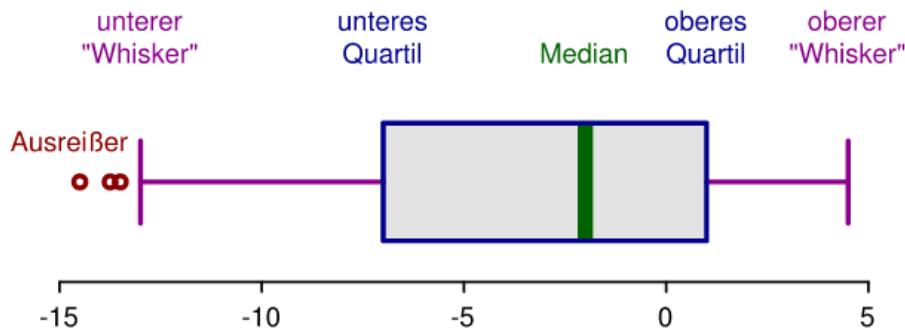
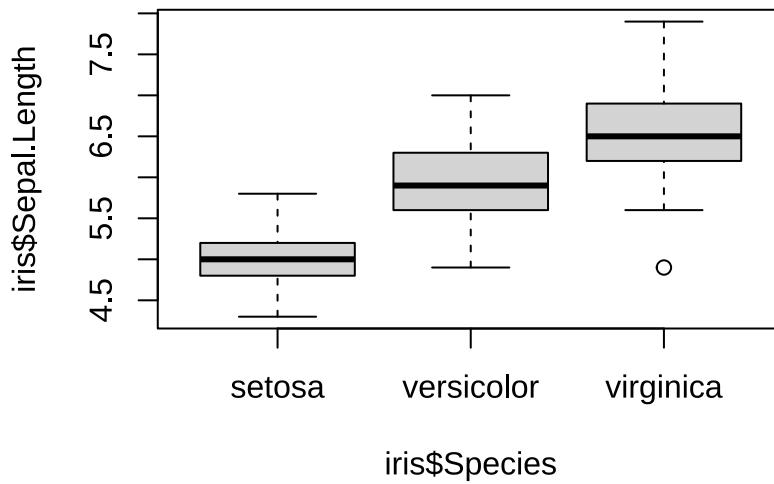


Abbildung 10: Schematische Darstellung eines Boxplots (Quelle: Von RobSeb - Eigenes Werk, CC BY-SA 3.0, <https://commons.wikimedia.org/w/index.php?curid=14697172>).

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.

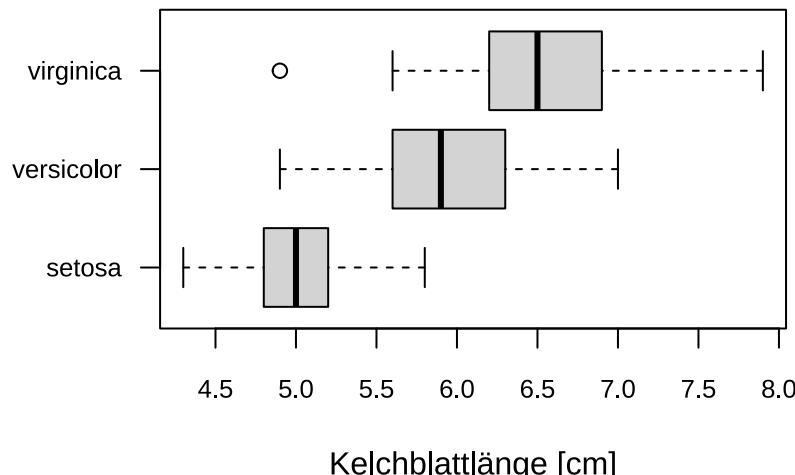
```
boxplot(iris$Sepal.Length ~ iris$Species)
```



1090

1091 Etwas eleganter ist es wenn wir das Argument `data` verwenden und den Plot etwas anpassen. Diese Schreib-  
1092 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
)
```



1093

1094

### 1095 Aufgabe 19: Boxplots

1096

- 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
- 1100 Erstellen Sie Boxplots für jedes Gebiet und innerhalb der Gebiete für jede Art.

## 9.4 ggplot2: Eine Alternative für Abbildungen

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

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

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

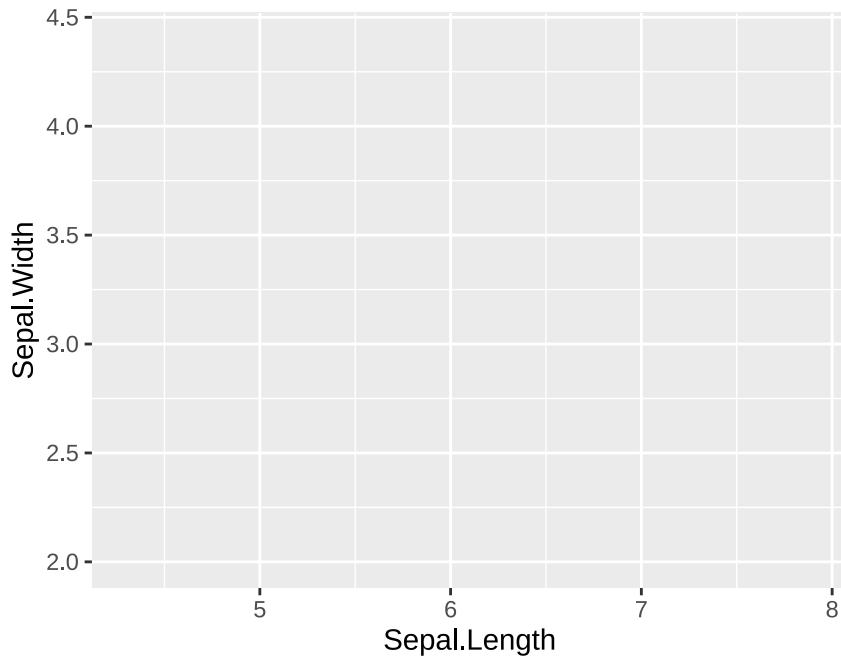
```
library(ggplot2)
head(iris)

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

Die Ästhetik wird bspw. folgendermaßen definiert.

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

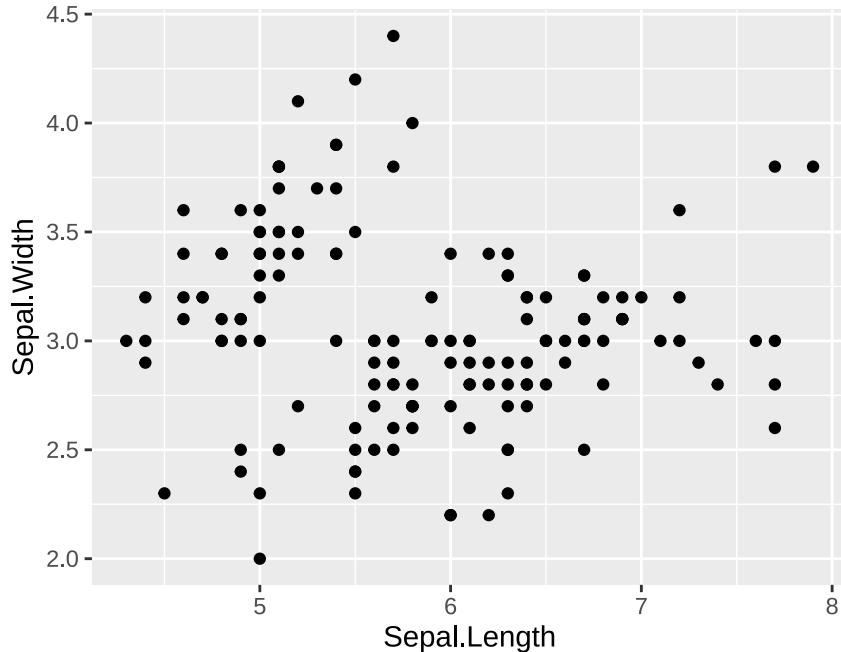
<sup>7</sup>Wir haben bis jetzt immer nur mit base R gearbeitet. D.h. wir haben nur Funktionen verwendet, die R bereits zur Verfügung stellt. Eine der großen Stärken von R sind die Erweiterungen (oder auch Pakete genannt). ggplot2 ist so eine Erweiterung, die einmal mit `install.packages("ggplot2")`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



1132

1133 Dieser Befehl zeichnet noch keine Daten. Die Daten werden lediglich herangezogen, um einen leeren Plot für  
 1134 die Daten zu erstellen. In dem Beispiel wird die Variable `Sepal.Length` aus dem `data.frame iris` als x und  
 1135 `Sepal.Width` als y Variable definiert. Diese Informationen stehen den folgenden Layern nun zur Verfügung,  
 1136 sodass nach den + nur noch x und y verwendet werden müssen. Um bspw. einen Scatterplot zu erstellen  
 1137 wird ein `geom_point()` Layer hinzugefügt. x und y werden automatisch an `geom_point` übergeben. Weitere  
 1138 Einstellung sind in diesem Beispiel nicht notwendig, wären jedoch möglich. Siehe `?geom_point()`.

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



1139

1140

---

1141 **Aufgabe 20: Abbildungen mit ggplot2**

---

1142

1143 Verwenden Sie die Daten aus Aufgabe 17 und erstellen Sie einen Scatterplot mit ggplot2 wie in Aufgabe 17.

1144

1145 Wir haben mit der Funktion `geom_point()` demm Plot eine Punktgeometrie hinzugefügt. Es gibt noch viele  
1146 weitere Geometrien. Die wichtigsten sind:

- 1147 • `geom_line()` für eine Linie.  
1148 • `geom_histogram()` um ein Histogramm zu erstellen.  
1149 • `geom_boxplot()` um einen Boxplot zu erstellen.  
1150 • `geom_bar()` um ein Säulendiagramm zu erstellen.

1151 Welche Geometrie die richtige ist, richtet nach dem Typ der darzustellenden Variablen. Beispielsweise  
1152 bietet sich `geom_point()` an, wenn man zwei kontinuierliche Variable darstellen möchte. Wenn man hin-  
1153 gegen die Verteilung von einer kontinuirlchen Variable darstellen möchte, dann bietet sich ein Histogramm  
1154 (`geom_histogram()`) oder auch eine geschätzte Dichte (z.B. `geom_density()`) an.

1155

---

1156 **Aufgabe 21: Abbildungen mit ggplot2**

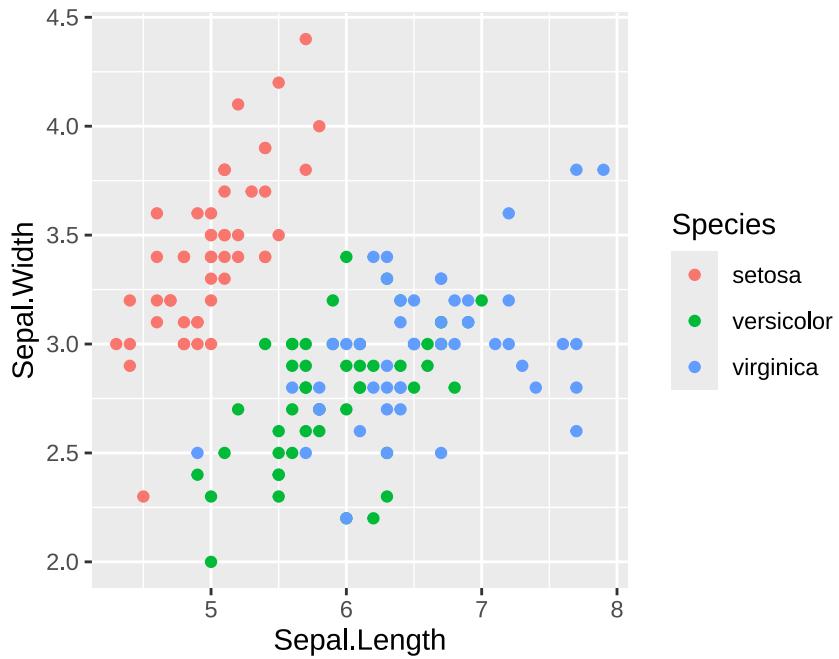
---

11571158 Verwenden Sie die den Iris Datensatz und erstellen Sie mit ggplot2 einen Plot der die Verteilung der Länge  
1159 der Kelchblätter zeigt (Spalte `Sepal.Length`).

1160

1161 Eine der Stärken von ggplot2 ist, dass man den Wert unterschiedlicher Variable auf unterschiedlichen  
1162 Komponenten des Plots abbilden kann. Wir haben bis jetzt ein bzw. zwei Variable auf der x- und y-Achse  
1163 abgebildet. Wir können aber ein weitere Variablen verwenden um das Aussehen des Plots zu beeinflussen.  
1164 Beispielsweise können wir die Farbe der Punkte (für `geom_point()`) mit dem Argument `col` beeinflussen.

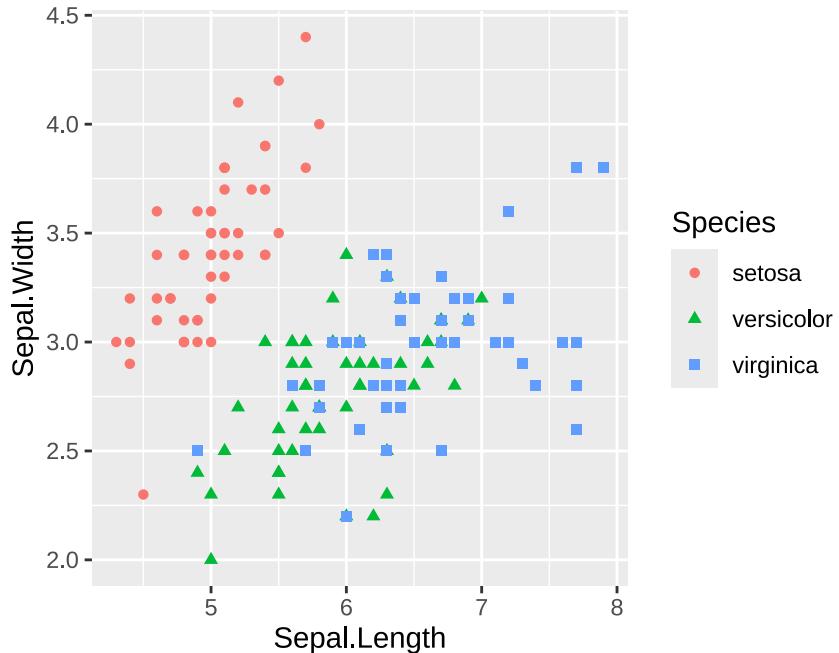
```
ggplot(iris, aes(x = Sepal.Length, y = Sepal.Width, col = Species)) +
 geom_point()
```



1165

1166 Somit bekommt jede Irisart eine eigene Farbe<sup>8</sup>. Gleichesmaßen können wir die Punktart (**shape**), die  
1167 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()
```

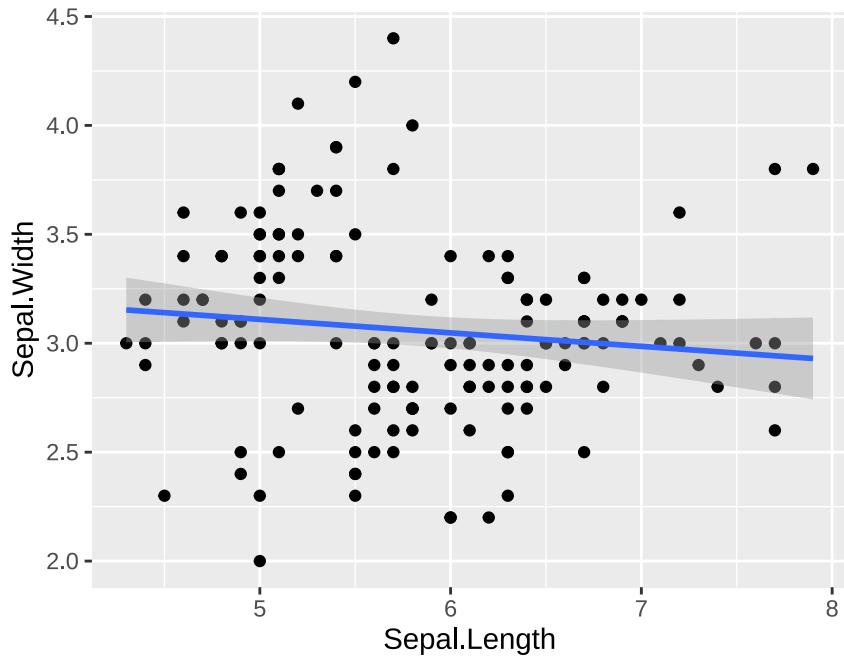


1168

1169 In dem Plot ist die Information zu der Art redundant (einmal als Farbe und einmal Symbolart).  
1170 Ein weitere sehr nützliche Geometrie ist **geom\_smooth()**, die es erlaubt eine Trendlinie hinzuzufügen.

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

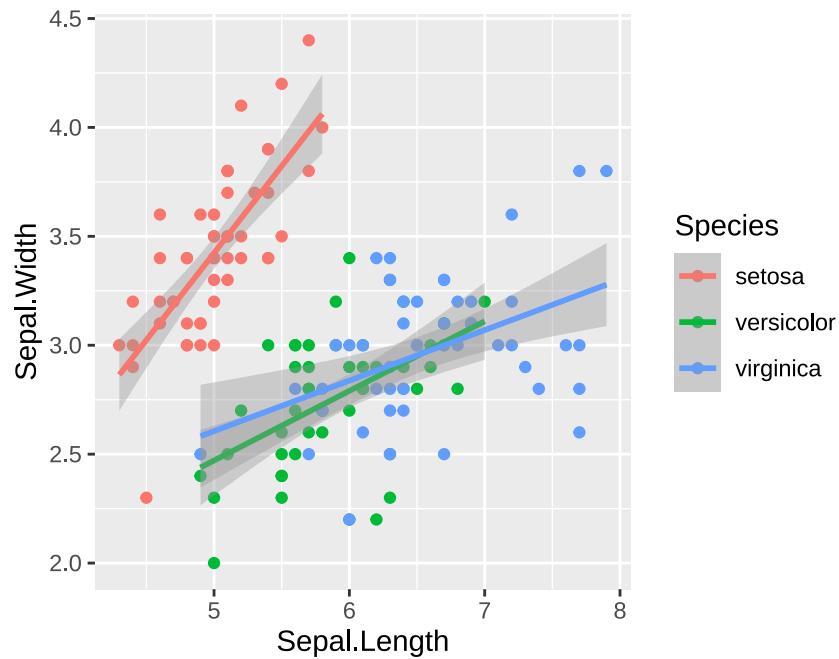
```
ggplot(iris, aes(x = Sepal.Length, y = Sepal.Width)) +
 geom_point() + geom_smooth(method = "lm")
```



1171

1172 Mit `method = "lm"` wird festgelegt, dass die Trendlinie gerade sein soll (es wird eine lineare Einfachregression  
 1173 angepasst). Wenn wird wieder eine gruppierende Variable einführen (z.B. die Beobachtungen nach Art auf  
 1174 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")
```



1175

1176

1177 **Aufgabe 22: Anpassen von Plots**  
1178

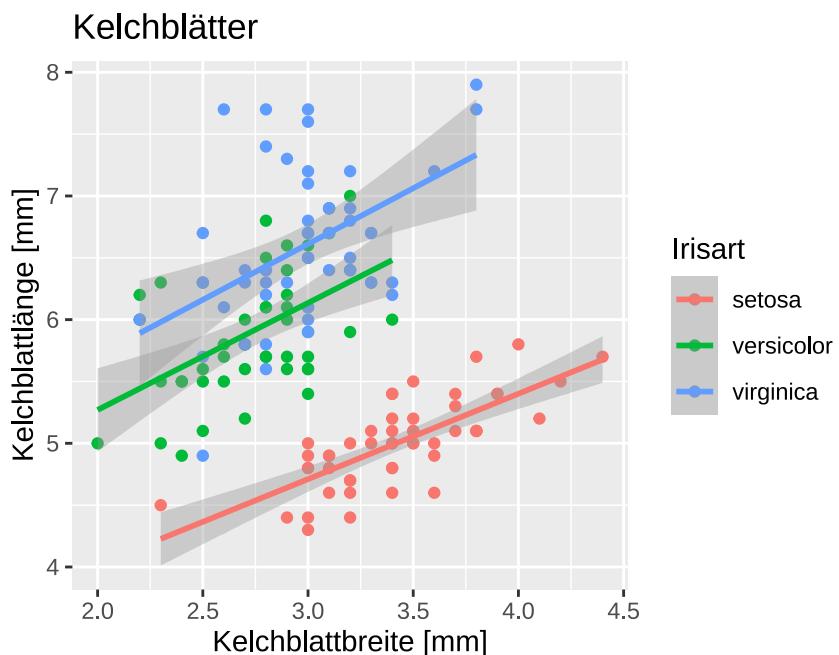
- 1179 Lesen Sie den Datensatz `data/bhd_1.txt` ein und erstellen Sie einen Boxplot für die Verteilung des BHDs  
 1180 für jede Baumart. In einem zweiten Schritt verwenden Sie erst `col = gebiet` und dann `fill = gebiet`.  
 1181 Welchen Unterschied stellen Sie fest?

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

1182

- 1183 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")
```



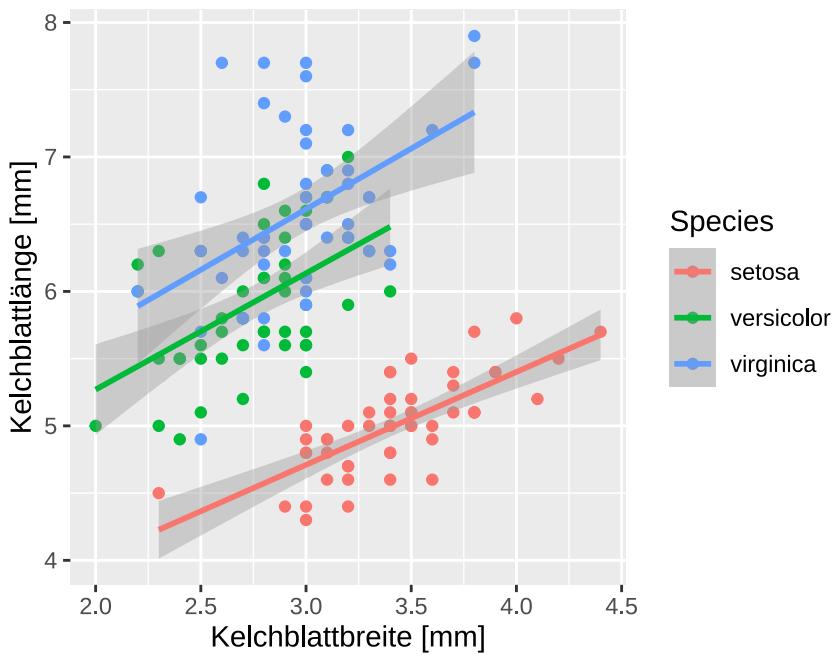
1184

- 1185 Statt einen langen Befehl zu tippen, kann ein `ggplot()` auch zwischengespeichert und wieder aufgerufen bzw.  
 1186 angepasst werden. Das ist vor allem sinnvoll, wenn mehrere Abbildungen auf dem selben Zwischenergebnis  
 1187 aufbauen sollen.

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

- 1188 Wir können jetzt mit `p1` weiter arbeiten und beispielsweise eine Beschriftung hinzufügen.

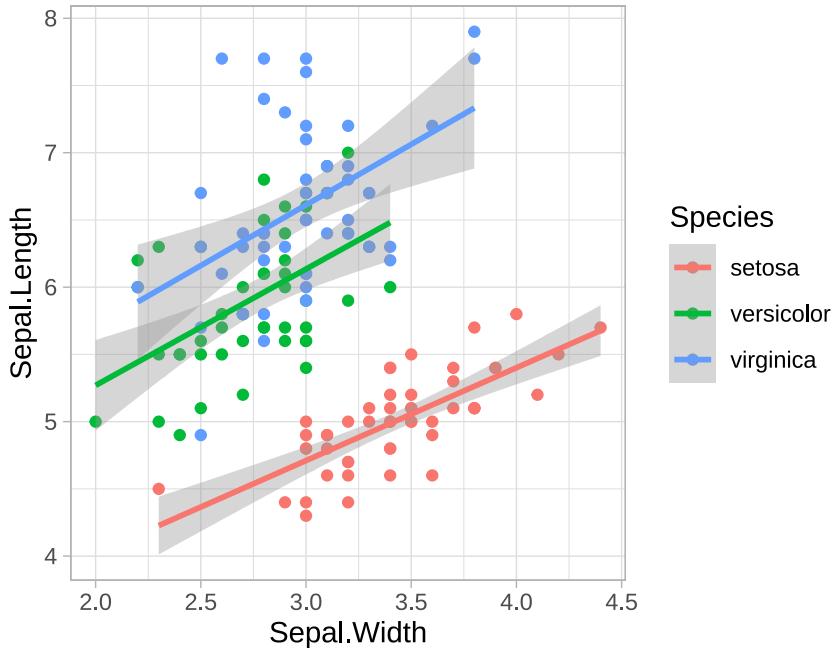
```
p1 + labs(x = "Kelchblattbreite [mm]", y = "Kelchblattlänge [mm]")
```



1189

1190 Oder auch den ganzen Plot anpassen. Dafür gibt es *themes*. Es gibt eine Reihe von vorgefertigten *themes*  
1191 oder man kann diese auch selber erstellen (das ist aber nicht Teil dieses Kurses).

```
p1 + theme_light()
```

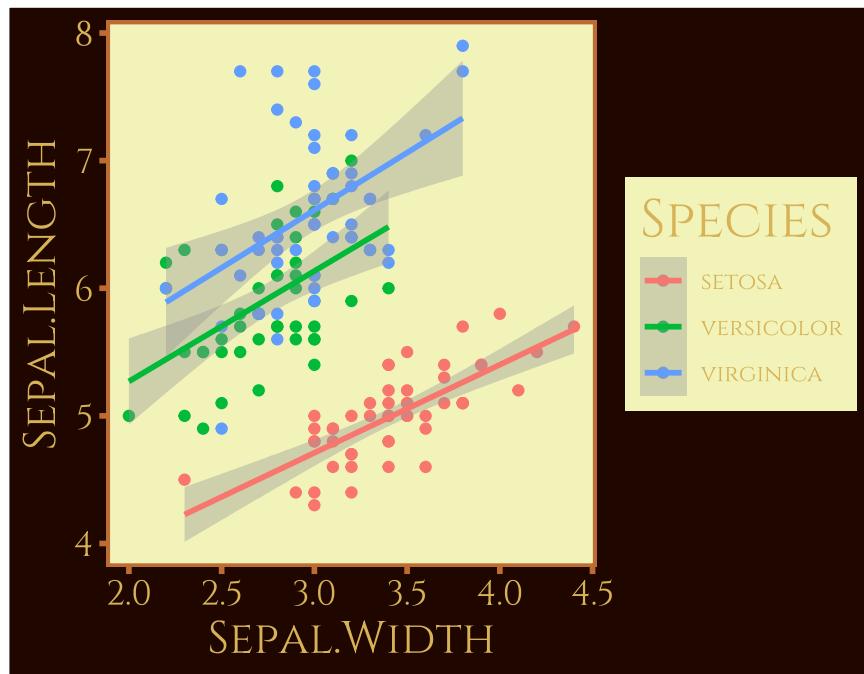


1192

1193 Weitere *themes* sind: `theme_bw()`, `theme_linedraw()` oder `theme_dark()`. Es gibt extra Pakete die viele  
1194 zusätzliche weitere *themes* anbieten. Dazu gehört z.B. das Paket `ggthemes` oder `ThemePark`. Während  
1195 `ggthemes` hauptsächlich durchdachte Grafikkonzepte liefert, sind die Themes aus `ThemePark` eher Popkultur  
1196 und nicht 100 %ig ernst gemeint. `ThemePark` muss zunächst aus GitHub installiert werden. Die Installation

1197 wird auf der GitHub erläutert.

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

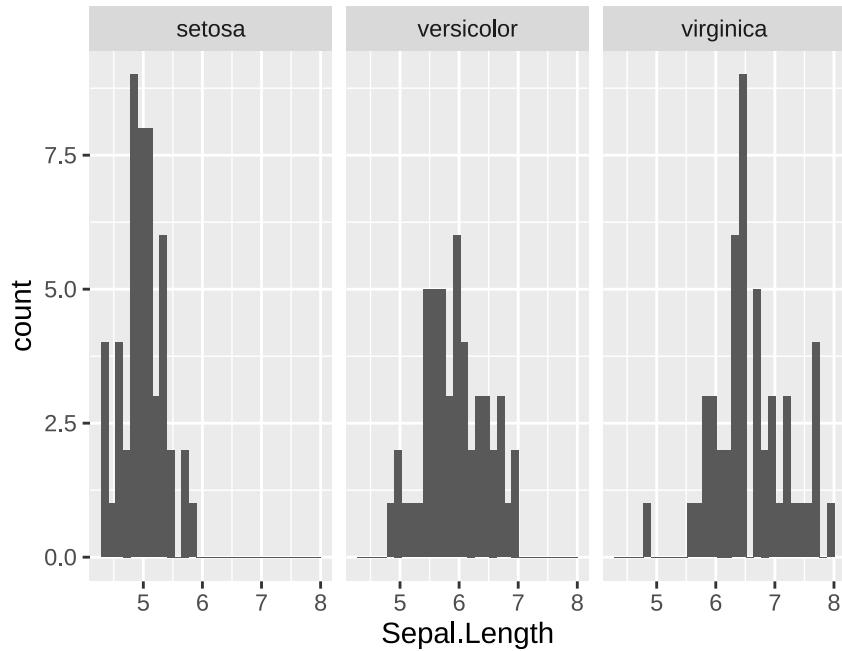


1198

#### 1199 9.4.1 Multipanel Abbildungen

1200 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:  
1201 `facet_grid()` und `facet_wrap()`.

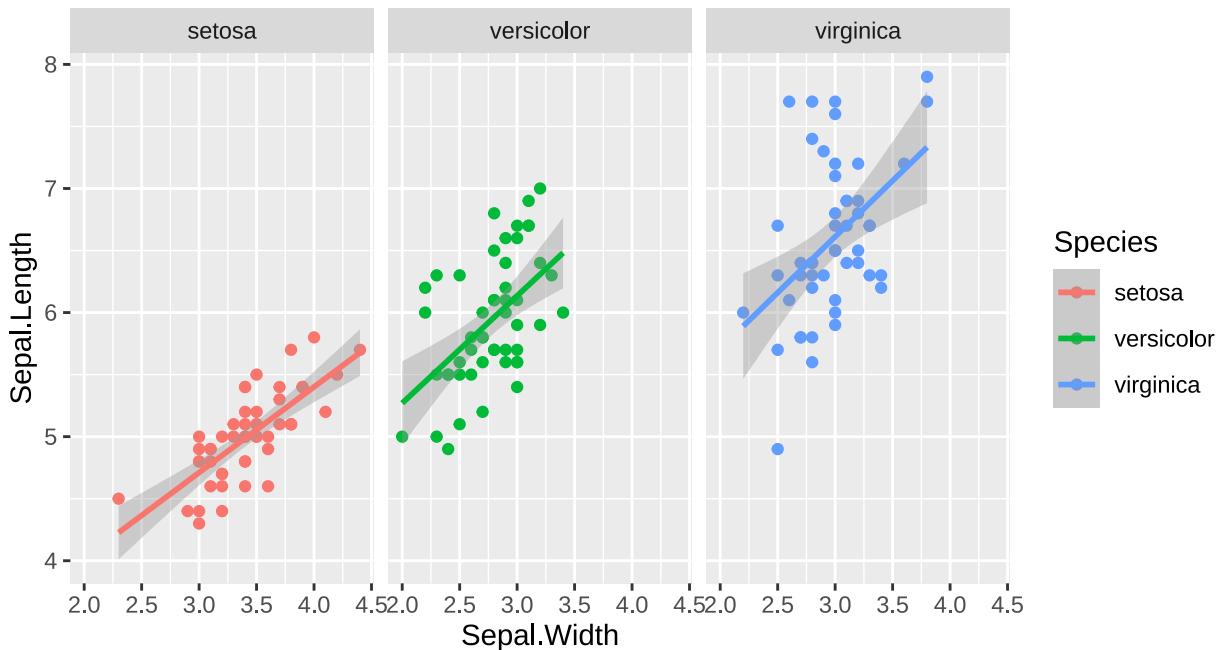
```
ggplot(iris, aes(Sepal.Length)) + geom_histogram() +
 facet_grid(~ Species)
```



1203

1204 Die Funktion `facet_grid()` erzeugt einen *Grid* und beschriftet die Grid-Zeilen und -Spalten, während  
 1205 `facet_wrap()` für jedes Panel (Diagramm) eine eigene Überschrift erzeugt. Beim Arrangieren der Diagramme  
 1206 wird der Vorteil von `ggplot2` gegenüber `base plot` besonders deutlich. Während es im `base plot` System  
 1207 mühsam ist, die Abbildungen zu arrangieren, übernimmt `ggplot2` das Arrangement automatisch und fügt  
 1208 sogar eine automatisch Legende hinzu. Die Achsenabschnitte werden so angepasst, dass die Daten vergleichbar  
 1209 sind.

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

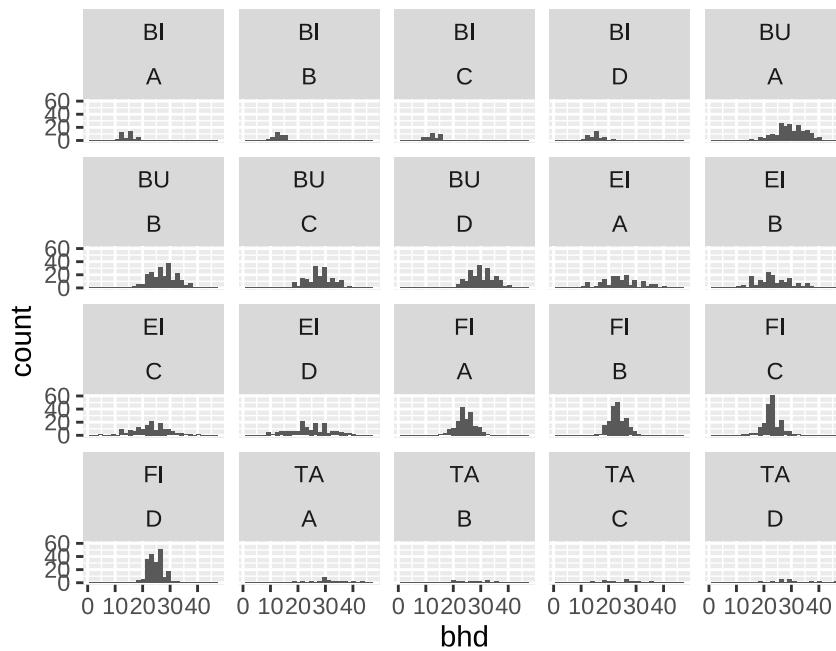


1210

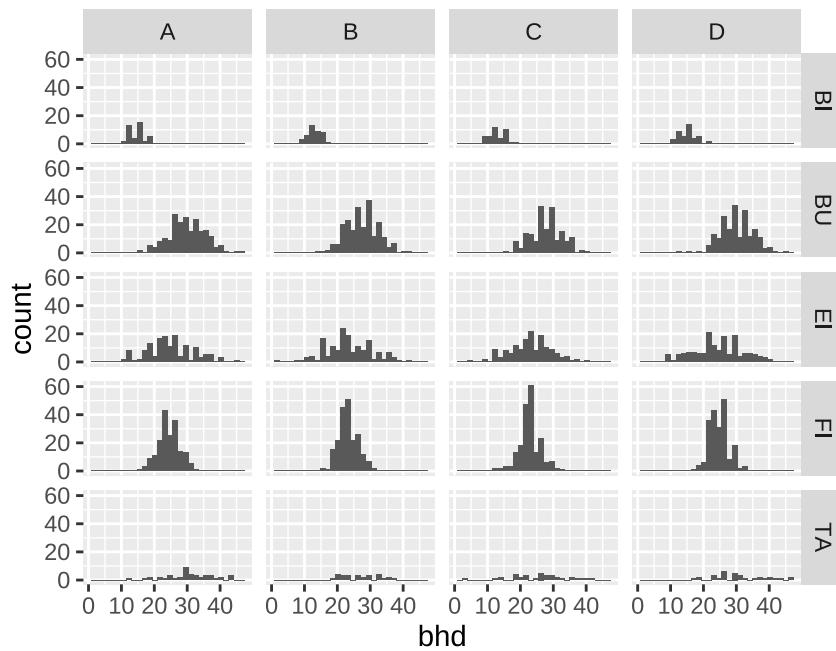
1211

1212 **Aufgabe 23: Multipanel Abbildungen**

- 1214 Lesen Sie erneut den Datensatz `daten/bhd_1.txt` ein und speichern Sie diesen in die Variable (`dat_bhd`).  
 1215 Erstellen Sie für jede Art und Gebiet ein Histogramm. Welche Unterschiede können Sie feststellen, wenn Sie  
 1216 `facet_grid()` oder `facet_wrap()` verwenden?



1217



1218

1219 **9.4.2 Plots kombinieren**

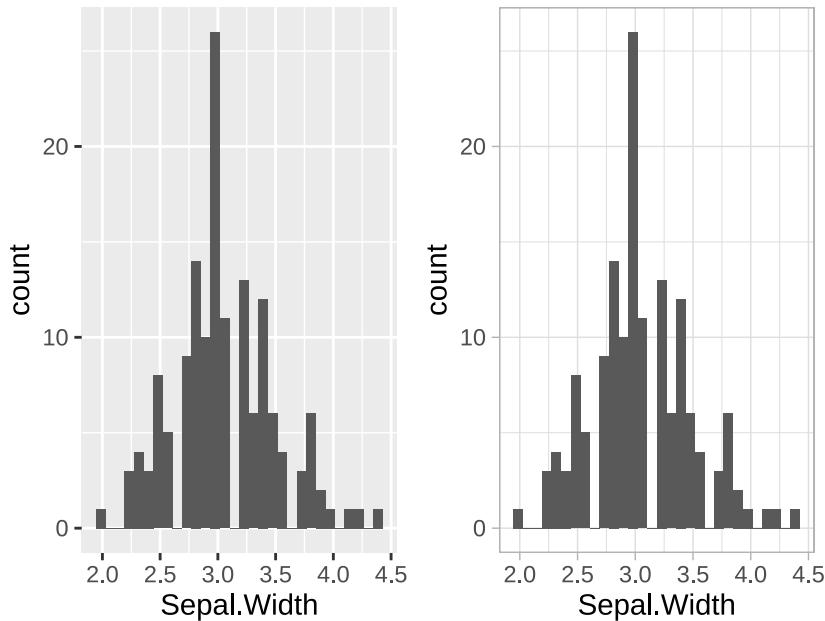
1220 Es gibt Situationen, in denen **unterschiedliche** Plots miteinander kombiniert werden müssen. Im vorherigen  
 1221 Abschnitt wurde dies immer anhand einer gruppierenden Kovariate gemacht. Aber es gibt auch Situationen, in  
 1222 denen das nicht möglich ist. Beispielsweise wenn ein Histogramm und ein Scatterplot vom gleichen Datensatz  
 1223 zusammengefasst werden sollen. Dafür bietet sich das Paket **patchwork** an<sup>9</sup>.

1224 Als erstes können wir zwei (oder natürlich auch mehrere Plots) erstellen. Hier unterscheiden sich die Plots  
 1225 lediglich durch das Aussehen.

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

1226 Dann müssen können wir diese Plots ebenfalls mit `+` zusammenfügen.

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



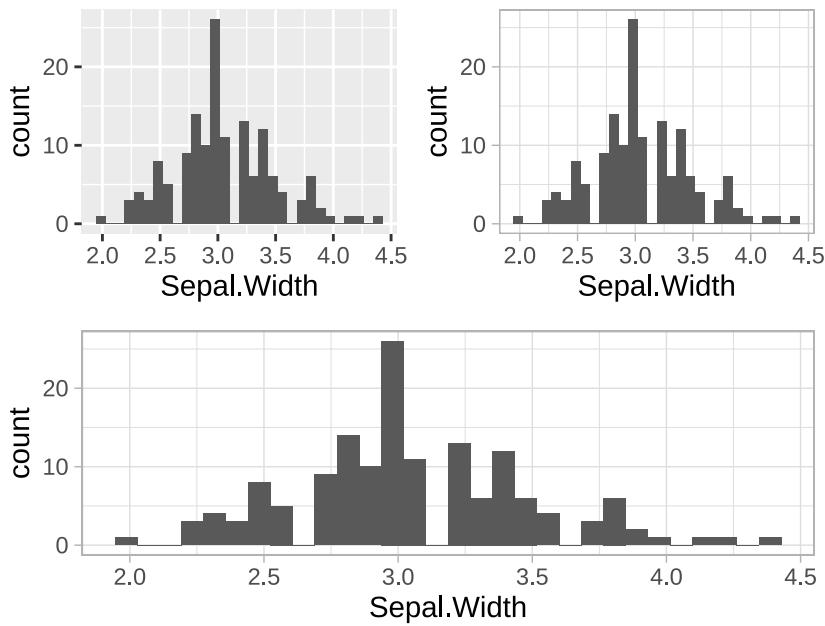
1227

1228 Natürlich können auch weitere Plots hinzugefügt werden (auch in unterschiedlichen Dimensionen):

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

---

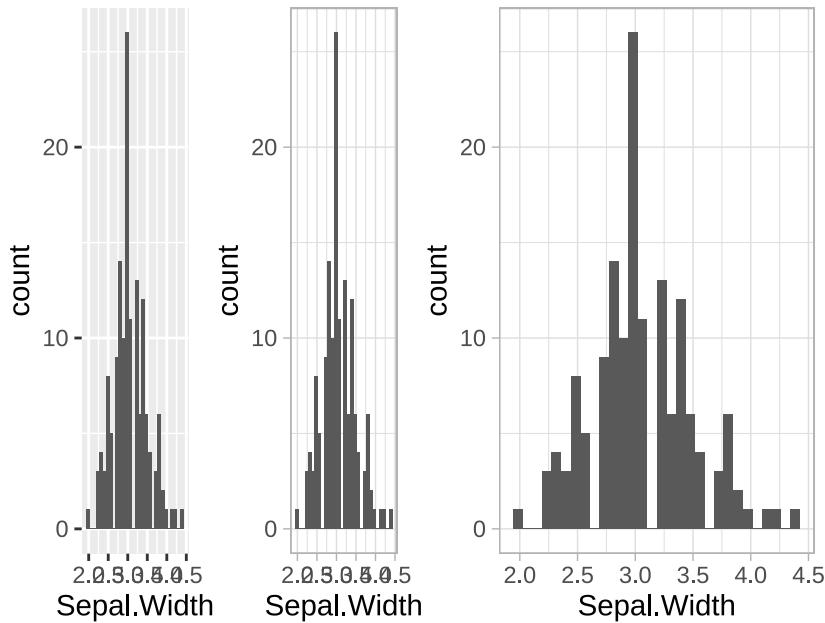
<sup>9</sup>Auch dieses Paket müssen Sie einmalig mit `install.packages("patchwork")` installieren.



1229

1230 Des weiteren können mit | auch Plots gegenüber gestellt werden.

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



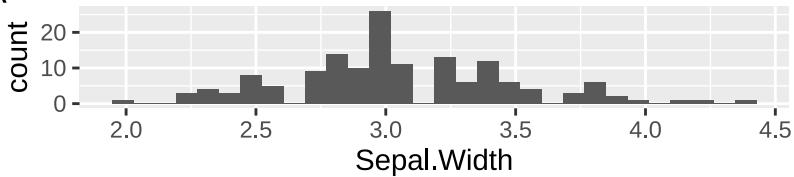
1231

1232 Weitere Optionen können mit `plot_layout()` und `plot_annotation()` angepasst werden. Mit  
1233 `plot_layout()` kann die Anordnung der Plots bestimmt werden (z.B. über die Argument `nrow`  
1234 und `ncol`), sowie deren *relative* Größe (über die Argumente `widths` und `heights`). Mit der Funktion  
1235 `plot_annotation()` können zusätzliche Beschriftungen hinzugefügt werden, wie beispielsweise eine Titel  
1236 (Argument `title`) oder ein Buchstabe/Zahl für jedes Element (Argument `tag_levels`).

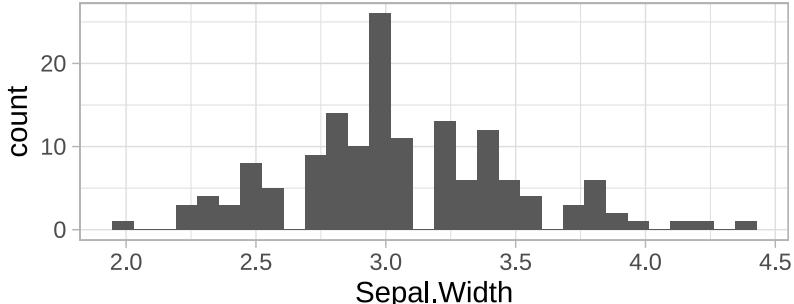
```
p1 + p2 +
 plot_layout(ncol = 1, heights = c(0.3, 0.7)) +
 plot_annotation(title = "Zwei Histogramme", tag_levels = "A")
```

## Zwei Histogramme

A



B



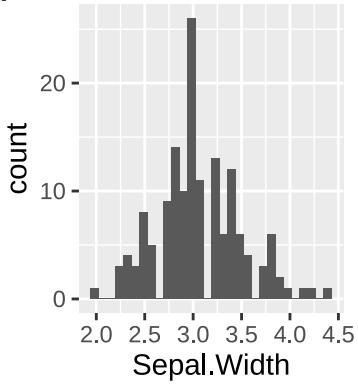
1237

1238

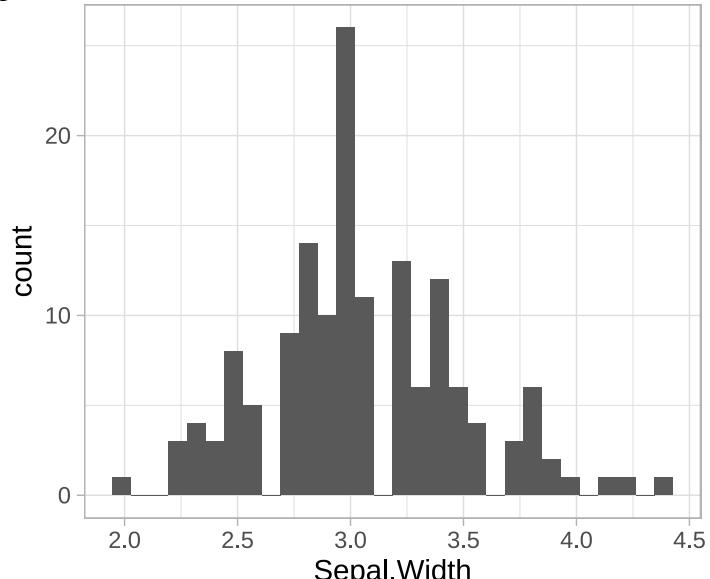
### 1239 Aufgabe 24: Mehrere Plots zusammenfügen 1240

- 1241 Versuchen Sie die folgende Zusammenstellung der Plots nachzumachen:

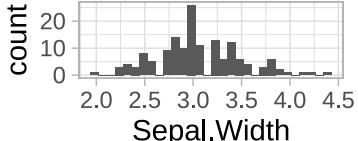
a



c



b



1242

### 1243 9.4.3 Speichern von plots

- 1244 Sie können mit `ggsave()` eine zwischen-gespeicherte Abbildung exportieren, indem Sie den Variablennamen übergeben. Wenn Sie keine Variable übergeben, wird automatisch die letzte Abbildung gespeichert. Das

- 1246 Dateiformat wird aus dem Dateinamen übernommen. Die Größe der Inhalte können Sie sehr leicht mit den  
1247 Argumenten `width` und `height` oder `scale` gesteuert werden.

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

## 1248 10 Mit Daten arbeiten

### 1249 10.1 dplyr eine Einführung

1250 `dplyr` ist eine Erweiterung von R (ein Paket), die das Ziel hat, den Umgang mit Daten einfacher und  
 1251 schneller zu machen.

1252 `dplyr` definiert 5 Verben, um mit Daten zu arbeiten. Diese sind:

- 1253 • `filter`
- 1254 • `select`
- 1255 • `arrange`
- 1256 • `mutate`
- 1257 • `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))
```

1258 Damit die Funktionen aus `dplyr` verwendet werden können, müssen wir als erstes das Paket `dplyr` laden.  
 1259 The online Resource [R for Data Science](#) gibt einen sehr guten, tieferen Eindruck in die Arbeit mit `dplyr`  
 1260 und erklärt auch die `dplyr` Philosophie noch Mal sehr genau für Beginner.

```
library(dplyr)
```

1261 Sollte dies zu einer Fehlermeldung führen, dann müssen Sie das Paket `dplyr` erst installieren. Dafür müssen  
 1262 Sie `einmalig install.packages("dplyr")` installieren.

1263 `dplyr` stellt unterschiedliche Funktionen zum Arbeiten mit Daten zur Verfügung. Es gibt fünf Grundfunktionen  
 1264 für die am häufigsten vorkommenden Operationen. Mit der Funktion `filter()` können unterschiedliche  
 1265 Beobachtungen gefiltert werden:

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

```
1266 ## id plot bhd alter
1267 ## 1 1 1 50 10
1268 ## 2 2 1 29 30
1269 ## 3 3 2 13 31
1270 ## 4 4 2 23 24
1271 ## 5 5 3 25 25
```

1272 Es können auch mehrere Spalten verwendet werden.

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

```
1273 ## id plot bhd alter
1274 ## 1 2 1 29 30
1275 ## 2 3 2 13 31
1276 ## 3 4 2 23 24
```

```
1277 ## 4 5 3 25 25
```

1278 Natürlich kann genau das gleiche Ergebnis mit dem ‘normalen’ R erreicht werden, dies wäre dann:

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

```
1279 ## id plot bhd alter
1280 ## 2 2 1 29 30
1281 ## 3 3 2 13 31
1282 ## 4 4 2 23 24
1283 ## 5 5 3 25 25
```

1284 Eine weitere Funktion aus dem Paket **dplyr** ist **select()**. Damit können Spalten aus einem **data.frame** ausgewählt werden. Dabei können auch die Spaltennamen unbenannt werden.

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

```
1286 ## bhd
1287 ## 1 50
1288 ## 2 29
1289 ## 3 13
1290 ## 4 23
1291 ## 5 25
```

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

```
1292 ## bhd id
1293 ## 1 50 1
1294 ## 2 29 2
1295 ## 3 13 3
1296 ## 4 23 4
1297 ## 5 25 5
```

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

```
1298 ## BHD id
1299 ## 1 50 1
1300 ## 2 29 2
1301 ## 3 13 3
1302 ## 4 23 4
1303 ## 5 25 5
```

1304 Mit der Funktion **arrange()** können die Beobachtungen in einem **data.frame** sortiert werden.

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

```
1305 ## id plot bhd alter
1306 ## 1 3 2 13 31
1307 ## 2 4 2 23 24
1308 ## 3 5 3 25 25
```

```
1309 ## 4 2 1 29 30
1310 ## 5 1 1 50 10
```

1311 Mit der Funktion `desc()` kann die Anordnung in absteigender Reihenfolge sortiert werden.

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

```
1312 ## id plot bhd alter
1313 ## 1 1 1 50 10
1314 ## 2 2 1 29 30
1315 ## 3 5 3 25 25
1316 ## 4 4 2 23 24
1317 ## 5 3 2 13 31
```

1318 Mit der Funktion `mutate()` kann man eine neue Spalte hinzufügen.

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

```
1319 ## id plot bhd alter bhd_mm fl
1320 ## 1 1 1 50 10 500 1963.4954
1321 ## 2 2 1 29 30 290 660.5199
1322 ## 3 3 2 13 31 130 132.7323
1323 ## 4 4 2 23 24 230 415.4756
1324 ## 5 5 3 25 25 250 490.8739
```

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

```
1325 ## id plot bhd alter mean_bhd
1326 ## 1 1 1 50 10 28
1327 ## 2 2 1 29 30 28
1328 ## 3 3 2 13 31 28
1329 ## 4 4 2 23 24 28
1330 ## 5 5 3 25 25 28
```

1331 Mit der Funktion `summarise()` können Spalten zusammengefasst werden.

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

```
1332 ## mean_bhd mean_sd
1333 ## 1 28 13.63818
```

1334

1335 **Aufgabe 25: Datenmanipulation mit dplyr**

---

1336

- 1337 1. Laden Sie den Datensatz `data/bhd_1.txt`
- 1338 2. Berechnen Sie folgende Werte für alle Einträge und speichern Sie die Ergebnisse in `erg1`
- 1339 • mittlerer `bhd`
  - 1340 • maximales `alter`
  - 1341 • die Standardabweichung des BHDs
  - 1342 • die Anzahl Bäume mit einem BHD > 30

1343 **10.2 Arbeiten mit gruppierten Daten**

1344 Zusätzlich können `mutate` und `summarise` auch auf gruppierte Daten angewendet werden. Dafür müssen  
 1345 wir erst die Funktion `group_by()` aufrufen und als Argumente die Variablen übergeben, die die Gruppen  
 1346 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>
```

```
1366 ## <dbl> <dbl>
1367 ## 1 1 39.5
1368 ## 2 2 18
1369 ## 3 3 25
```

1370

---

**Aufgabe 26: dplyr mit gruppierten Daten**

---

- 1373 1. Laden Sie den Datensatz `data/bhd_1.txt`
- 1374 2. Berechnen Sie für jede Baumart und Standort die folgenden Werte:
  - 1375 • mittlerer `bhd`
  - 1376 • maximales `alter`
  - 1377 • die Standardabweichung des BHDs
  - 1378 • die Anzahl Bäume mit einem BHD > 30
- 1379 3. Ordnen Sie das Ergebnis nach Baumart und BHD.

1380 **10.3 pipes oder %>%**

1381 Mit *Pipes* (`%>%`) kann man das Ergebnis einer Funktion einfach an eine nachfolgende Funktion weiterreichen.

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

1382 Wir kennen bis jetzt:

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

1383 `## [1] 3.333333`

1384 Mit *Pipes*, die durch das Symbol `%>%` dargestellt werden<sup>10</sup>, können wir das etwas vereinfachen und nacheinander  
1385 schreiben:

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

1386 `## [1] 3.333333`

1387 Oder sogar

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

1388 `## [1] 3.333333`

1389

---

**Aufgabe 27: Pipes %>%**

---

1392 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.

- 1393 1. Laden Sie den Datensatz `data/bhd_1.txt`.
- 1394 2. Berechnen Sie für jede Baumart und Standort die folgenden Werte:
- 1395 • mittlerer `bhd`
- 1396 • maximales `alter`
- 1397 • die Standardabweichung des BHDs
- 1398 • die Anzahl Bäume mit einem BHD > 30
- 1399 3. Ordnen Sie das Ergebnis nach Baumart und BHD.

## 1400 10.4 Joins

1401 Eine weitere häufige Aufgabe beim Daten Management ist es Daten zusammenzuführen. Nehmen Sie an, dass  
1402 wir folgende Aufnahmen gemacht haben

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

1403 und jeder Baum lediglich mit einer `id` versehen wurde. In einer zweiten Tabelle wurden dann weitere Daten  
1404 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")
)
```

1405 Ziel ist es jetzt die Bäume aus `aufnahmen` mit den Informationen aus den `metadaten` zu verbinden. Dazu  
1406 dient `id` als Bindeglied (oft auch Schlüssel genannt).

1407 Dazu gibt es vier Möglichkeiten.

1408 Zur Durchführung gibt es in base R die Funktion `merge()`. Wir werden aber gleich die Funktionen aus dem  
1409 Paket `dplyr` verwenden.

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

1410 ## id bhd art gebiet
1411 ## 1 1 20 <NA> <NA>
1412 ## 2 2 31 Ta A
1413 ## 3 3 74 Bu B

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

1414 ## id bhd art gebiet
1415 ## 1 2 31 Ta A
1416 ## 2 3 74 Bu B
1417 ## 3 4 NA Bu B
```

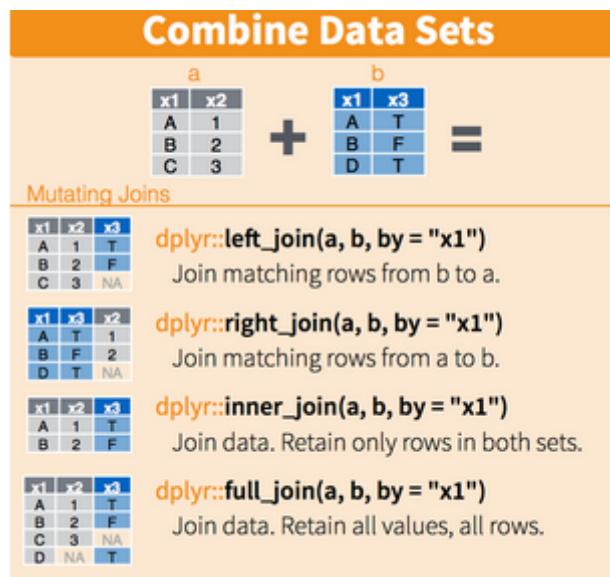


Abbildung 11: Joins (Quelle Rstudio)

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

```
1418 ## id bhd art gebiet
1419 ## 1 2 31 Ta A
1420 ## 2 3 74 Bu B
full_join(aufnahmen, metadaten, by = "id")
```

```
1421 ## id bhd art gebiet
1422 ## 1 1 20 <NA> <NA>
1423 ## 2 2 31 Ta A
1424 ## 3 3 74 Bu B
1425 ## 4 4 NA Bu B
```

1426 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"))
```

```
1427 ## id bhd art gebiet
1428 ## 1 1 20 <NA> <NA>
1429 ## 2 2 31 Ta A
1430 ## 3 3 74 Bu B
```

1431

1432 **Aufgabe 28: Verbinden von Daten**

1433

- 1434 • Lesen Sie die Datensätze `daten/bhd_2.txt` und `daten/bhd_2_meta.txt` ein.
- 1435 • Stellen Sie sicher, dass es für den `bhd` keine fehlenden Werte gibt (entfernen sie entsprechende Zeilen)
- 1436 • Fügen Sie zu den Metadaten (gespeichert in `bhd_2_meta`) die Anzahl Bäume und den mittleren `bhd`
- 1437 hinzu pro Gebiet.

1438 **10.5 ‘long’ and ‘wide’ Datenformate**

1439 Unter anderem Wickham (2014) empfieilt das Prinzip von *tidy* Data. Nach diesem Prinzip sollten Daten wie  
 1440 folgt organisiert sein:

- 1441 • Jede Zeile ist ein Merkmalsträger/Subjekt/Objekt (z. B. eine Person oder ein Baum).
- 1442 • Jede Spalte ist eine Variable (ein Merkmal), die den Merkmalsträger beschreibt.
- 1443 • In jeder Zelle ist genau ein Wert (Merkmalausprägung), nämlich der Wert, der Variable für den Merk-  
 1444 malsträger.

1445 Zum Beispiel enthalten Spaltennamen oft Informationen, die eigentlich in einer Variable gespeichert werden  
 1446 sollten. Folgendes Beispiel gibt die BHD Messung von 3 Bäumen in 3 Jahren wieder. Der Vorteil von *tidy*  
 1447 Daten ist, dass viele Funktionen diese Form erwarten. Somit müssen sie Ihre Daten nur ein Mal organisieren  
 1448 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)
)
```

1449 Der `tibble` aus dem Paket `tibble` ist ein moderner Data Frame, der sich in seinen Funktionen in das `tidy`  
 1450 Universum einfügt. Wir können auch einfach das Paket `tidyverse` laden, um alle Pakete, die Teil des `tidy`  
 1451 Universums sind gleichzeitig zu laden. Der `tibble` beinhaltet alle Funktionen, die der Data Frame auch  
 1452 beinhaltet und kann auch für genau die gleichen Zwecke verwendet werden. Der `tibble` hat u. A. eine  
 1453 modernere Darstellung im Konsolenoutput.

1454 Diese Daten sind jetzt im `wide`-Format gespeichert und folglich nicht `tidy`, weil Information über die Daten  
 1455 (nämlich das Jahr der Aufnahme in den Spaltennamen gespeichert sind). Besser wäre eine Struktur mit  
 1456 nur drei Spalten: `id`, `jahr` und `bhd`. Um die Daten in so eine Struktur zu bringen, gibt es die Funktion  
 1457 `pivot_longer()` aus dem Paket `tidyR`, die ebenfalls Teil des `tidyverse` ist.

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

A tibble: 9 x 3
id name value
```

```

1460 ## <int> <chr> <dbl>
1461 ## 1 1 bhd2015 30
1462 ## 2 1 bhd2016 31
1463 ## 3 1 bhd2017 34
1464 ## 4 2 bhd2015 31
1465 ## 5 2 bhd2016 31
1466 ## 6 2 bhd2017 32
1467 ## 7 3 bhd2015 32
1468 ## 8 3 bhd2016 33
1469 ## 9 3 bhd2017 33

```

1470 Wenn wir die Spalten für die Variable und den Wert gleich sinnvoll benennen möchten, können wir das über  
1471 die Argumente `names_to` und `value_to` machen.

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

```

1472 ## # A tibble: 9 x 3
1473 ## id jahr bhd
1474 ## <int> <chr> <dbl>
1475 ## 1 1 bhd2015 30
1476 ## 2 1 bhd2016 31
1477 ## 3 1 bhd2017 34
1478 ## 4 2 bhd2015 31
1479 ## 5 2 bhd2016 31
1480 ## 6 2 bhd2017 32
1481 ## 7 3 bhd2015 32
1482 ## 8 3 bhd2016 33
1483 ## 9 3 bhd2017 33

```

1484 Analog zu der Funktion `pivot_longer()` gibt es auch die Funktion `pivot_wider()`, um vom Daten vom  
1485 `long`-Format ins `wide`-Format zu transformieren.

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

```

1486 ## # A tibble: 3 x 4
1487 ## id bhd2015 bhd2016 bhd2017
1488 ## <int> <dbl> <dbl> <dbl>
1489 ## 1 1 30 31 34
1490 ## 2 2 31 31 32
1491 ## 3 3 32 33 33

```

1492

---

1493 **Aufgabe 29: Zeitliche Verlauf von BHDS**

---

1495 In der Datei `bhd_3.csv` befinden sich gemessene BHDS (in cm) von unterschiedlichen Bäumen zu unter-  
 1496 schiedlichen Zeitpunkten. Erstellen Sie ein Liniendiagramm, das den zeitlichen (x-Achse) Verlauf der BHDS  
 1497 (y-Achse) für die unterschiedlichen Bäume darstellt.

1498 **10.6 Auswählen von Variablen**

1499 Sobald die Datensätze etwas umfangreicher werden (d. h., es gibt mehrere Spalten in einem `data.frame`),  
 1500 können innerhalb vieler `dplyr`-Funktionen spezielle Funktionen verwendet werden, um Variablen auszuwählen.  
 1501 Wenn die genaue Position der Spalten bekannt ist, kann man mit dem `:`-Operator und der Position Spalten  
 1502 auswählen:

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

1503 ## Sepal.Length Sepal.Width Petal.Length  
 1504 ## 1 5.1 3.5 1.4  
 1505 ## 2 4.9 3.0 1.4  
 1506 ## 3 4.7 3.2 1.3

1507 Diese Vorgehensweise kann gehährlich sein, da sich manchmal Spalten verschieben und sich somit die  
 1508 Positionen ändern. Es besser Spalten immer explizit mit Namen statt mit Nummern anzusprechen.

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

1509 ## Sepal.Length Sepal.Width Petal.Length  
 1510 ## 1 5.1 3.5 1.4  
 1511 ## 2 4.9 3.0 1.4  
 1512 ## 3 4.7 3.2 1.3

1513 `select()` erlaubt es, auch hier den `:`-Operator zu verwenden:

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

1514 ## Sepal.Length Sepal.Width Petal.Length  
 1515 ## 1 5.1 3.5 1.4  
 1516 ## 2 4.9 3.0 1.4  
 1517 ## 3 4.7 3.2 1.3

1518 Es gibt auch einige spezielle Funktionen, um Spalten innerhalb eines `select()`-Aufrufs auszuführen:

- 1519 • `starts_with()`: Hier kann man ein Muster angeben, mit dem ein Text anfangen muss.
- 1520 • `ends_with()`: Diese Funktion ist analog zu `starts_with()`, jetzt wird aber am Ende des Spaltennamens  
 1521 nach dem Muster gesucht.
- 1522 • `contains()`: Hier kann ein Muster übergeben werden, das irgendwo im Spaltennamen sein muss.
- 1523 • `everything()`: Mit dieser Funktion werden alle Spalten ausgewählt.

1524 • `last_col()`: Mit dieser Funktion wird nur die letzte Spalte ausgewählt (dass ist die Spalte, die ganz  
 1525 rechts ist).

1526 Sämtliche Auswahlen können mit - umgekehrt werden.

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

1527 ## Sepal.Length Sepal.Width

1528 ## 1 5.1 3.5

1529 ## 2 4.9 3.0

1530 ## 3 4.7 3.2

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

1531 ## Petal.Length Petal.Width Species

1532 ## 1 1.4 0.2 setosa

1533 ## 2 1.4 0.2 setosa

1534 ## 3 1.3 0.2 setosa

1535 `select()` bietet auch noch die Möglichkeit, Spaltennamen zu ändern.

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

1536 ## sep\_width

1537 ## 1 3.5

1538 ## 2 3.0

1539 ## 3 3.2

1540

#### **Aufgabe 30: Auswählen von Spalten**

---

1543 In der Datei `messungen_1.csv` sind Messungen von zwei Sensoren enthalten für die ersten vier Monate eines  
 1544 Jahres. Führen Sie folgende Abfragen durch:

1545 1. Wählen Sie alle Messungen für Januar aus.

1546 2. Wählen Sie alle Messungen für Januar und März aus.

#### **10.7 Einzelne Beobachtungen abfragen (`slice()`)**

1547 Mit dem Befehl `slice()` kann man einzelne Beobachtungen (= Zeilen) aus einem `data.frame` abfragen.

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

1549 ## Sepal.Length Sepal.Width Petal.Length Petal.Width Species

1550 ## 1 5.1 3.5 1.4 0.2 setosa

1551 ## 2 4.4 2.9 1.4 0.2 setosa

1552 ## 3 5.1 3.5 1.4 0.3 setosa

1553 Davon gibt es drei nützliche Varianten: 1) `slice_head()` und `slice_tail()`; 2) `slice_max()` und  
 1554 `slice_min()`; 3) `slice_random()`.

1555 `slice_head()` und `slice_tail()` sind analog zu `head()` und `tail()`, aber mit dem entscheidenden Unter-  
 1556 schied, dass Gruppierungen berücksichtigt werden. Wenn keine Gruppierung in den Daten vorhanden ist, gibt  
 1557 es keinen Unterschied.

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

```
1558 ## Sepal.Length Sepal.Width Petal.Length Petal.Width Species
1559 ## 1 5.1 3.5 1.4 0.2 setosa
1560 ## 2 4.9 3.0 1.4 0.2 setosa
```

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

```
1561 ## Sepal.Length Sepal.Width Petal.Length Petal.Width Species
1562 ## 1 5.1 3.5 1.4 0.2 setosa
1563 ## 2 4.9 3.0 1.4 0.2 setosa
```

1564 Sobald jedoch eine gruppierende Variable eingeführt wird, gibt `slice_head()` die ersten n Beobachtungen  
 1565 für jede Gruppe zurück und `head()` für den gesamten Datensatz.

```
base head
```

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

```
1566 ## # A tibble: 2 x 5
1567 ## # Groups: Species [1]
1568 ## Sepal.Length Sepal.Width Petal.Length Petal.Width Species
1569 ## <dbl> <dbl> <dbl> <dbl> <fct>
1570 ## 1 5.1 3.5 1.4 0.2 setosa
1571 ## 2 4.9 3 1.4 0.2 setosa
```

```
dplyr slice_head
```

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

```
1572 ## # A tibble: 6 x 5
1573 ## # Groups: Species [3]
1574 ## Sepal.Length Sepal.Width Petal.Length Petal.Width Species
1575 ## <dbl> <dbl> <dbl> <dbl> <fct>
1576 ## 1 5.1 3.5 1.4 0.2 setosa
1577 ## 2 4.9 3 1.4 0.2 setosa
1578 ## 3 7 3.2 4.7 1.4 versicolor
1579 ## 4 6.4 3.2 4.5 1.5 versicolor
1580 ## 5 6.3 3.3 6 2.5 virginica
1581 ## 6 5.8 2.7 5.1 1.9 virginica
```

1582 `slice_tail()` funktioniert analog zu `slice_head()` mit dem einzigen Unterschied, dass nicht die ersten n

1583 Zeilen zurück gegeben werden sondern die letzten `n` Zeilen.  
 1584 `slice_max()` und `slice_min()` geben die Beobachtung mit dem maximalen bzw. minimalen Wert einer  
 1585 Variable zurück. Auch hier werden Gruppen berücksichtigt.

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

```
1586 ## Sepal.Length Sepal.Width Petal.Length Petal.Width Species
1587 ## 1 7.9 3.8 6.4 2 virginica
```

1588 Und mit Gruppen:

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

```
1589 ## # A tibble: 3 x 5
1590 ## # Groups: Species [3]
1591 ## Sepal.Length Sepal.Width Petal.Length Petal.Width Species
1592 ## <dbl> <dbl> <dbl> <dbl> <fct>
1593 ## 1 5.8 4 1.2 0.2 setosa
1594 ## 2 7 3.2 4.7 1.4 versicolor
1595 ## 3 7.9 3.8 6.4 2 virginica
```

1596 `slice_min()` funktioniert genau gleich, nur dass die Beobachtung (=Zeile) mit dem kleinsten Wert einer  
 1597 Variable zurück gegeben wird.

1598 Die Funktion `slice_sample()` erlaubt es zufällige Beobachtungen zu ziehen. Dabei kann über das Argument `n`  
 1599 die Anzahl an Beobachtungen angegeben werden oder über das Argument `prop` der Anteil an Beobachtungen.

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

```
1600 ## Sepal.Length Sepal.Width Petal.Length Petal.Width Species
1601 ## 1 6.5 2.8 4.6 1.5 versicolor
1602 ## 2 6.3 3.3 4.7 1.6 versicolor
1603 ## 3 7.2 3.2 6.0 1.8 virginica
1604 ## 4 4.9 3.6 1.4 0.1 setosa
1605 ## 5 6.0 2.7 5.1 1.6 versicolor
```

1606 Das Ergebnis ist bei jedem von Ihnen anders, da es sich um eine zufällige Ziehung handelt. Wenn Sie diese  
 1607 Ergebnisse wiederholen möchten, können Sie über `set.seed()` die zufällige Ziehung reproduzierbar machen.

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

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

```
1608 ## Sepal.Length Sepal.Width Petal.Length Petal.Width Species
1609 ## 1 4.3 3.0 1.1 0.1 setosa
1610 ## 2 5.0 3.3 1.4 0.2 setosa
1611 ## 3 7.7 3.8 6.7 2.2 virginica
1612 ## 4 4.4 3.2 1.3 0.2 setosa
1613 ## 5 5.9 3.0 5.1 1.8 virginica
```

1614 Wenn beispielsweise 5% der Beobachtungen gezogen werden sollen, kann dies so gemacht werden:

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

```
1615 ## Sepal.Length Sepal.Width Petal.Length Petal.Width Species
1616 ## 1 7.7 3.8 6.7 2.2 virginica
1617 ## 2 5.5 2.5 4.0 1.3 versicolor
1618 ## 3 5.5 2.6 4.4 1.2 versicolor
1619 ## 4 6.5 3.0 5.2 2.0 virginica
1620 ## 5 6.1 3.0 4.6 1.4 versicolor
1621 ## 6 6.3 3.4 5.6 2.4 virginica
1622 ## 7 5.1 2.5 3.0 1.1 versicolor

1623 slice_sample() berücksichtigt ebenfalls Gruppen. Mit den Argumenten replace und weight_by dann die
1624 Zufallsziehung genauer spezifiziert werden. replace sagt, ob eine gezogenen Beobachtung wieder zurück gelegt
1625 wird oder nicht. Mit dem Argument weight_by können optional gewichtete für jede Beobachtung vergeben
1626 werden.
```

1627

### 1628 Aufgabe 31: Daten beschreiben

---

1630 Verwenden Sie den Datensatz `bhd_1.txt` und finden Sie für jedes Gebiet und Art die Beobachtung mit
1631 kleinsten BHD.

## 1632 10.8 Spalten trennen

1633 Ein gut geplanter Datensatz besteht aus Beobachtungen (Zeilen), Variablen (Spalten) und in jeder Zelle ist
1634 immer ein **genau** ein Wert gespeichert. Leider gibt es oft Datensätze, bei denen mehr als ein Wert pro Zelle
1635 gespeichert wurde. Die Funktion `separate()` kann helfen solche Daten zu trennen.

1636 Wir verwenden einen erfunden Datensatz zu Beobachtungen von Tieren und einer geschätzten Distanz zu
1637 diesen Tieren.

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

1638 In der Spalte `beobachtung` sind zwei Informationen gespeichert: Die Distanz zur Beobachtung und die Art.
1639 Das ist ungünstig, weil wir so weder nach Tierart noch nach distanz filtern können (nicht *tidy*). Mit der
1640 Funktion `separate()`, können wir Beobachtungen einer Spalte in mehrere Spalten trennen. Dafür muss der
1641 Spaltennamen (Argument `col`), die neuen Spaltennamen (Argument `into`) und das Trennzeichen (Argument
1642 `sep`) angegeben werden.

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

1643 ## # A tibble: 4 x 3

```
1644 ## id Distanz Art
1645 ## <int> <chr> <chr>
1646 ## 1 1 10m " Reh"
1647 ## 2 2 100m " Reh"
1648 ## 3 3 20m " Fuchs"
1649 ## 4 4 40 "Reh"

1650 Nach dem Aufruf von separate() gibt es zwei neue Spalten (Distanz und Art), die die alte Spalte
1651 beobachtung ersetzen.
```

1652

---

1653 **Aufgabe 32: Aufräumen**

---

1655 Verwenden Sie den folgenden Datensatz und bringen Sie ihn in eine Form, die sicherstellt dass

- 1656 • jede Zelle genau einen Wert enthält.  
1657 • jede Zeile eine Beobachtung ist.  
1658 • 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")
)
```

## 1659 11 Arbeiten mit Text

1660 Bis jetzt haben wir fast ausschließlich mit Zahlen oder Abbildungen gearbeitet. R bietet aber auch viele  
 1661 Werkzeuge, um mit Text zu arbeiten. Wir wollen hier ein paar Funktionen dafür vorstellen. Als erstes sollte  
 1662 nochmals klar gestellt werden, was eigentlich ein Text ist. In R ist alles, das innerhalb von doppelten ("") oder  
 1663 einfachen ('') Anführungszeichen geschrieben ist, Text.

1664 Anbei einige Beispiele:

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

1665 Wichtig ist hier zu sehen, dass z nicht als Zahl sondern, als Text interpretiert wird (siehe Datentypen).

```
z + 1
```

1666 ## Error in z + 1: nicht-numerisches Argument für binären Operator

1667 Wenn man sicher ist, dass es sich bei einem Textobjekt um eine Zahl handelt, kann man dies mit der Funktion  
 1668 `as.numeric()` in eine Zahl umwandeln.

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

1669 ## [1] 31

1670 Aber mit a führt dies wieder zu einem NA-Wert, da a nicht in eine Zahl umgewandelt werden kann.

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

1671 ## Warning: NAs durch Umwandlung erzeugt

1672 ## [1] NA

### 1673 11.1 Arbeiten mit Text

1674 Wir wollen erst einmal drei Funktionen besprechen, die es erlauben mit Text zu arbeiten. Die Funktion  
 1675 `nchar()`<sup>11</sup> gibt an wie viele Zeichen ein Text hat. Also z.B.

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

1676 ## [1] 5

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

1677 ## [1] 2

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

1678 ## [1] 20

1679 Die Funktion `paste()` erlaubt es verschiedene Variablen mit Text zu verbinden. Wenn wir z. B. die Variablen  
 1680 `vorname <- "Eva"` und `name <- "Meier"` haben und wir wollen eine neue Variable `full_name <- Eva`

---

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

1681 "Meier" erzeugen, dann kann das mit der Funktion `paste()` gemacht werden.

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

1682 ## [1] "Eva Meier"

1683 Die Funktion `paste()` hat das Argument `sep`, das auf ein Leerzeichen ( ) gesetzt ist, aber auch anders sein kann und das Trennzeichen definiert.

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

1685 ## [1] "Eva, Meier"

1686 Die Funktion `substr()` erlaubt es am Anfang oder Ende eines Wortes etwas abzuschneiden. Dabei muss 1687 immer die Anfangs- (`start`) und Endposition (`stop`) angegeben werden.

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

1688 ## [1] "Hal"

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

1689 ## [1] "allo"

1690

---

### 1691 Aufgabe 33: Arbeiten mit Text 1

---

1693 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")
```

1694 1. Aus wie vielen Buchstaben besteht jedes Wort?

1695 2. Finden Sie das längste Wort.

1696 3. Wie viel Prozent der Wörter fangen mit einem S an?

1697 4. Fügen Sie jedem Wort seine Position im Vektor hinzu. Beispielsweise soll aus `Vogel` "2. Vogel" werden  
1698 usw.

## 1699 11.2 Finden von Textmustern

1700 Mit der Funktion `grep()` können Muster in einem Text gefunden werden. Wenn wir beispielsweise folgenden  
1701 Vektor mit Textelementen haben.

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

1702 Und wir wollen alle Straßennamen die ein `weg` haben abfragen, dann können wir folgenden Befehl ausführen:

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

1703 `## [1] 2`

1704 Im zweiten Element von `txt` kommen die Zeichen `Weg` vor. Beachte, in der Standardeinstellung wird zwischen  
1705 Groß- und Kleinschreibung unterschieden. Dies kann mit dem Argument `ignore.case = TRUE` angepasst  
1706 werden.

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

1707 `## [1] 1 2`

1708 Mit der Funktion `sub` können Zeichen innerhalb einer Zeichenkette ersetzt werden.

1709 So ersetzt der folgende Ausdruck `ae` mit `ä`.

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

1710 `## [1] "Friedländer Weg"`

1711 Wenn allerdings das zu ersetzende Zeichen mehr als einmal vorkommt und beide Instanzen ersetzt werden  
1712 sollen braucht man die Funktion `gsub`.

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

1713 `## [1] "Friedländer Weg und Reinhäuser Landstrasse sind zwei Strassen in Goettingen."`

1714 Mit `sub()` wird nur das erste `ae` ersetzt, während `gsub()` alle `ae` mit einem `ä` ersetzt.

```
gsub("ae", "ä", txt)
```

1715 `## [1] "Friedländer Weg und Reinhäuser Landstrasse sind zwei Strassen in Goettingen."`

1716 Oft ist der genaue Ausdruck den man finden möchte jedoch Variabel. Beispielsweise möchte man alle Wörter  
1717 mit einem Umlaut oder Zahlen finden möchte, kann man das oft abkürzen. Dafür gibt es reguläre Ausdrücke.  
1718 Wir werden hier nur ein paar beispielhafte Anwendungen besprechen.

1719 Sowohl in den Funktionen `grep()` als auch `(g)sub()` kann mit anstatt dem Muster (immer das erste  
1720 Argument) aus ein regulärer Ausdruck angegeben werden. Mit `[1-9]` sind alle Zahlen von 1 bis 9 gemeint.

1721 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)
```

1722 `## [1] 1 3`

1723 Damit lässt sich auch das Problem mit groß- und kleingeschriebenen Wörtern lösen.

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

1724 ## [1] 1 2

1725

1726 **Aufgabe 34: Arbeiten mit Text 2**

---

1728 Verwenden Sie den folgenden Vektor:

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

1729 1. In wie vielen Wörtern kommt der Doppellaut au vor?

1730 2. Ersetzen Sie in allen Wörtern alle au mit \_ \_.

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

1731 ## [1] 5 6 13

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

1732 ## [1] "Versicherung" "Methoden" "Fluss" "Rudel" "B\_ \_m"

1733 ## [6] "H\_ \_s" "Foto" "Auffahrt" "Auto" "Handy"

1734 ## [11] "Teller" "Kalender" "Aufb\_ \_"

## 1735 12 Arbeiten mit Zeit

1736 Für den Computer bzw. R ist ein Datum erst einmal nichts anderes als ein Text. Für uns ist es sofort  
 1737 klar, dass der "13.2.2021" der 13. Februar 2021 ist, für den Computer zunächst nicht. Wir müssen R also  
 1738 irgendwie sagen, dass die 13 der Tag ist, die 2 der Monat und 2021 das Jahr. Dass der Computer die einzelnen  
 1739 Komponenten erkennt, und in einen Datentyp eigens für Zeitformate überführt, nennt man *parsen*<sup>12</sup>. Durch  
 1740 das *parsen* wird die Variable in den Datentyp **Date** überführt. Das Arbeiten mit Datum und Zeit kann  
 1741 kann anfangs sehr mühsam sein und viele Zeit-spezifischen Datenoperationen lassen sich auch mit den  
 1742 Basis-Datentypen durchführen. Sobald man einige Grundfertigkeiten erworben hat, stellt man jedoch fest,  
 1743 dass die Arbeit mit dem Zeitformat-Datentyp schneller und effizienter funktioniert. Starten Sie am besten  
 1744 gleich mit "echten" Zeit-Datentypen und versuchen Sie nicht, sich irgendwie mit Text und numerischen  
 1745 Datentypen selbst etwas zu basteln. Der erste Schritt ist immer ein Datum zu *parsen*. Wir verwenden dafür  
 1746 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)
```

1747 **lubridate** bietet eine Vielzahl von Funktionen zum *parsen* von Datum und Zeit, die sich aus:

- 1748 • y für Jahr,
- 1749 • m für Monat,
- 1750 • d für Tag,
- 1751 • h für Stunde,
- 1752 • m für Minute und
- 1753 • s für Sekunde

1754 zusammen setzten. Alle Funktionen nehmen als erstes Argument ein Textstring. Wenn wir z.B. den String  
 1755 2020-01-20 parsen wollen können wir das mit der Funktion **ymd** machen.

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

1756 ## [1] "2020-01-20"

1757 Dabei erkennt **lubridate** in der Regel die Trennzeichen:

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

1758 ## [1] "2020-01-20"

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

1759 ## [1] "2020-01-20"

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

1760 ## [1] "2020-01-20"

1761 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")

1762 ## [1] "2020-01-20"

1763 Jetzt stellt sich die Frage, was der Vorteil ist, wenn R ein Datum parst.

d <- dmy("20.1.2020")

1764 Wir können jetzt mit d arbeiten und einzelne Komponenten extrahieren.

day(d)

1765 ## [1] 20

month(d)

1766 ## [1] 1

year(d)

1767 ## [1] 2020

1768 Oder auch Zeiteinheiten hinzufügen oder abziehen.

d + days(10)

1769 ## [1] "2020-01-30"

d - years(20)

1770 ## [1] "2000-01-20"

d + hours(25)

1771 ## [1] "2020-01-21 01:00:00 UTC"

```

1772

---

**Aufgabe 35: Arbeiten mit Datum und Zeit**

---

- 1775 • 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.
- 1776
- 1777 • Extrahieren Sie nun aus jedem Element aus d das Jahr und die Stunde.
- 1778 • Fügen zu jedem Element in d 10 Tage hinzu.

**12.1 Arbeiten mit Zeitintervallen**

- 1780 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)
```

1782 Wir können jetzt mit `int` arbeiten und beispielsweise das Intervall verschieben,

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

1783 ## [1] 2023-03-18 UTC--2024-03-18 UTC

1784 die Länge des Intervalls berechnen

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

1785 ## [1] 31536000

1786 oder testen, ob ein Datum innerhalb des Intervalls liegt.

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

1787 ## [1] TRUE

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

1788 ## [1] FALSE

1789 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)
```

1790 ## # A tibble: 1 x 1

1791 ## a

1792 ## <date>

1793 ## 1 2020-07-01

1794 `%within%` funktioniert genauso mit Vekotren oder mit mehreren Intervallen. Wir könnten also zwei Intervalle definieren (z.B. Ostern und Pfingsten).

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

1796 Und Überprüfen welche Termine in eines der zwei Intervalle fallen.

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

```
Ostern
```

```
termine %within% ostern
```

1797 ## [1] FALSE TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE

```

Pfingsten
termine %within% pfingsten

1798 ## [1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE TRUE FALSE FALSE
1799 Der Zeit-Datentyp wird auch oft benutzt, um den Zeitbedarf von längeren Berechnungen zu messen.

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

1800 ## [1] 0.4999484
t2 <- now()
int_length(interval(t1, t2))

1801 ## [1] 0.589988

```

1802 **12.2 Formatieren von Zeit**

1803 Für die Ausgabe in Berichten oder Grafiken soll das Datum oft in einer speziellen Form dargestellt werden.

1804 Die Funktion `format()` bietet Möglichkeiten ein Datumsobjekt zurück in Text umzuwandeln.

1805 Ein Beispiel wäre `ymd("2021-2-10")` als 10.2.21 auszugeben.

```

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

```

1806 ## [1] "21.02.21"

1807 Dabei handelt sich bei `%d.%m.%y` um Abkürzungen für die unterschiedlichen Komponenten eines Datumobjekts.

1808 Siehe dazu die Hilfeseite von `strptime (help(strptime))`.

1809

---

1810 **Aufgabe 36: Arbeiten mit Intervallen**

1812 Wie viele Einträge aus dem Vektor `v1` befinden sich in einem Intervall, das zwischen dem 1.3.2021 und dem  
1813 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"
)

```

1814 **12.3 Zeitreihen**

1815 In der Forstbranche haben wir oft mit Zeitreihen zu tun. Zeitreihen sind Variablen, für die in zeitlichen  
1816 Intervallen Daten vorliegen. In der Regel monatlich, vierteljährig oder jährlich, wobei die Intervalle zwischen  
1817 den Messungen bei Zeitreihen immer gleich lang sind. Wiederholungsmessungen von Forsteinventuren (Forstein-  
1818 richtungen, Betriebsinventuren, die meisten forstl. Experimente, die BWI, ...) sind also in der Regel keine

1819 Zeitreihen in engeren Sinne. Turnusmäßig vermessene Versuchsflächen, wie sie z. B. die Versuchsanstalten  
 1820 unterhalten oder jährlich gemeldete Holzpreise jedoch schon.

1821 Zeitreihen unterscheiden sich nicht nur technisch, sondern auch inhaltlich fundamental von den uns schon  
 1822 bekannten Daten. Statistisch gesehen haben Zeitreihen Besonderheiten gegenüber anderen Variablen, da  
 1823 Sie von Ihrer eigenen Vergangenheit abhängen (autokorriert sind) und auch die Abhängigkeit anderer  
 1824 Variablen in der Regel nur sinnvoll ist, wenn deren Vergangenheit mitberücksichtigt wird (cross-correlation).  
 1825 Konventionelle Statistik ist oft nicht möglich, um Zeitreihen zu analysieren. Selbst ein ordinärer arithmetischer  
 1826 Mittelwert ist schon nicht mehr geeignet, um Zeitreihen statistisch zu beschreiben. Angefangen mit der  
 1827 Datendarstellung gibt es in R deshalb spezifische Zeitreihen-Funktionen. Aus diesem Grund sollten Sie  
 1828 Zeitreihen als solche speichern. Viele Funktionen erkennen den Datentyp und führen entsprechend spezifische  
 1829 Zeitreihen-Operationen durch, wenn ihnen Daten vom Typ "Zeitreihe" übergeben werden. Laden wir z. B.  
 1830 die Holzpreise für Fichte 2b (das sog. Leitsortiment, Fichenholz mit einem Mittendurchmesser von 20 bis 25  
 1831 cm), das Holzaufkommen dieses Sortiments (Einschlagsvolumen) und die Preise für Nadelholz vom  
 1832 statistischen Bundesamt<sup>14</sup>. Wir laden die Daten zunächst als csv ein:

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

1833 Diese 3 Zeitreihen bilden zusammen ein klassisches Marktmodell mit dem Preis eines homogenen Gutes  
 1834 (Leitsortimentspreis), dem Angebot (Holzeinschlag) und der Nachfrage (Schnittholzpreis). Mit der Funktion  
 1835 **ts** werden die Daten in ein Zeitreihenobjekt überführt (*pasrse*). Die Spalte mit den Jahren ist dann nicht mehr  
 1836 nötig, da die Informationen zur Zeit keine eigene Variable mehr sind, sondern als sog. Metainformationen in  
 1837 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])
```

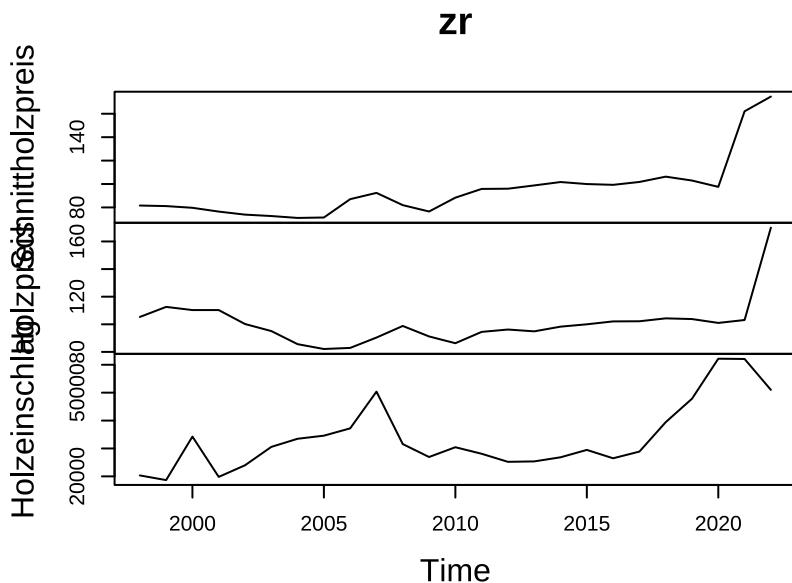
**typeof(zr)** # Zeitreihen sind sog. S3 Objekte. Das heißt, Zeitreihen gehören nicht zu den

1838 ## [1] "double"  
 # Basis Datentypen (siehe Kapitel Variablen, Funktionen und Datentypen),  
 # sondern sind eine Unterkategorie des Datentyps "Liste".

1839 Die wichtigsten Argumente sind - **data** Vektor oder Matrix, der nur die Daten enthält - **start** Startzeitpunkt -  
 1840 **frequency** Anzahl der Beobachtungen pro Zeiteinheit, also z. B. 12 bei monatlichen oder 4 bei quartalsweisen  
 1841 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

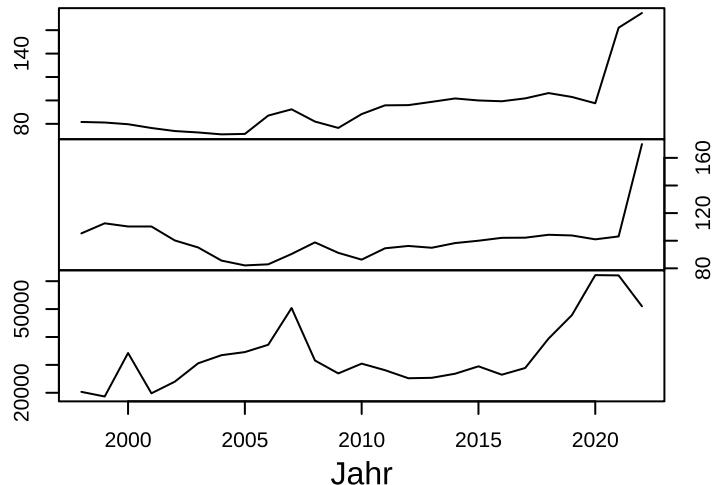


1842

1843 Wir können low-level Layer hinzufügen. Die meisten base plot Funktionen funktionieren auch für Zeitreihen

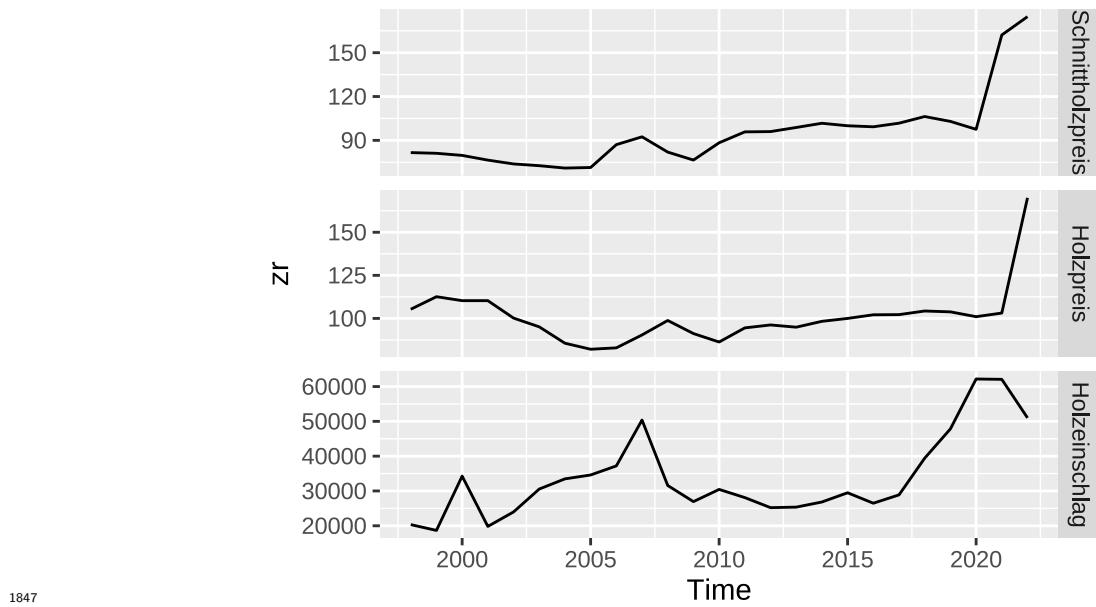
```
plot(zr, ann = FALSE, yax.flip = TRUE)
mtext(side = 3, cex = 1.5, font = 2, "Holzmarktentwicklung seit 1998", line = 2)
mtext(side = 1, cex = 1, line = 2.5, "Jahr")
```

## Holzmarktentwicklung seit 1998



1845 Beide Plot-Philosophien haben eine Zeitreihen-Funktion. Das Paket `ggfortify` ermöglicht automatisierte  
1846 Zeitreihenplots im `ggplot2` Stil. Damit ist auch das Problem der y-Achsenbeschriftungen gelöst.

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



1847

- 1848 Wir können die Abbildung im `ggplot2` Stil ändern. Hier nur ein Minimalbeispiel zur Veranschaulichung.  
1849 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

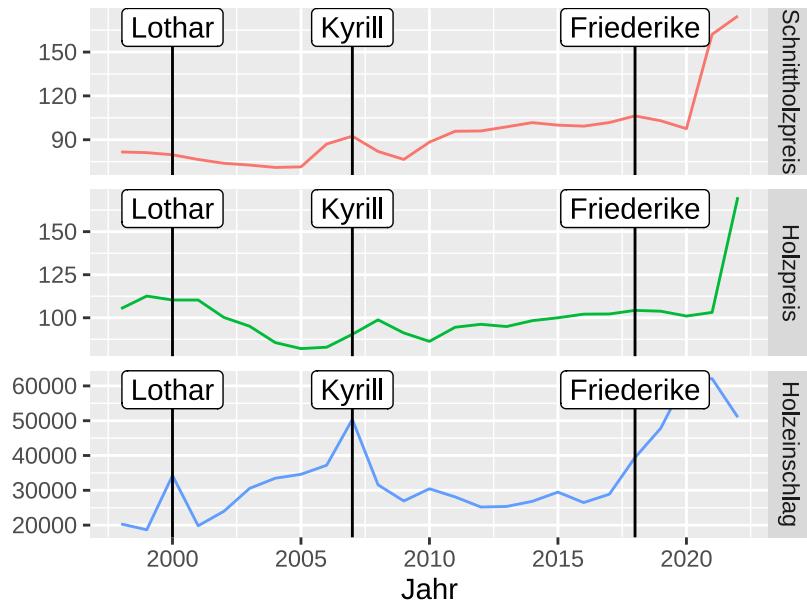
zr_autoplot + theme_minimal()
```

1850

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

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

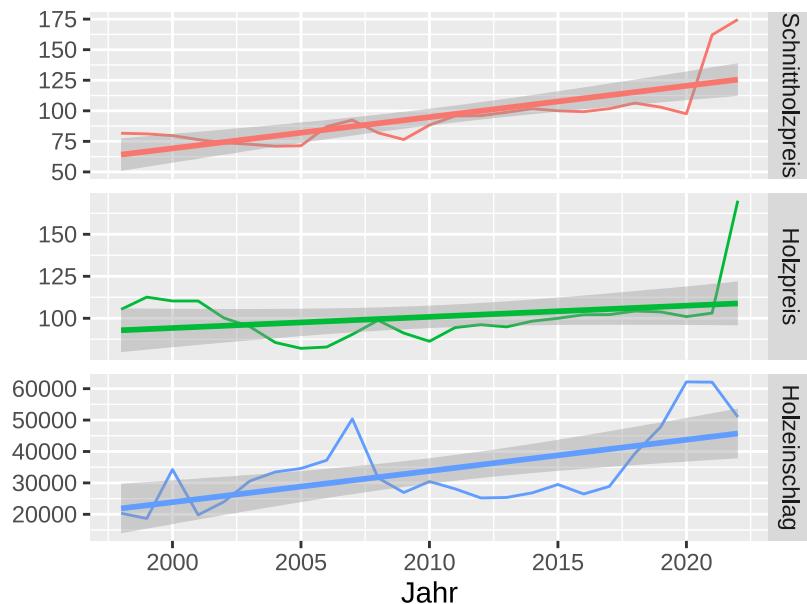
```
annotate(x = 2007, y = +Inf, label = "Kyrill", vjust = 1, geom = "label") +
 annotate(x = 2018, y = +Inf, label = "Friederike", vjust = 1, geom = "label")
```



1851

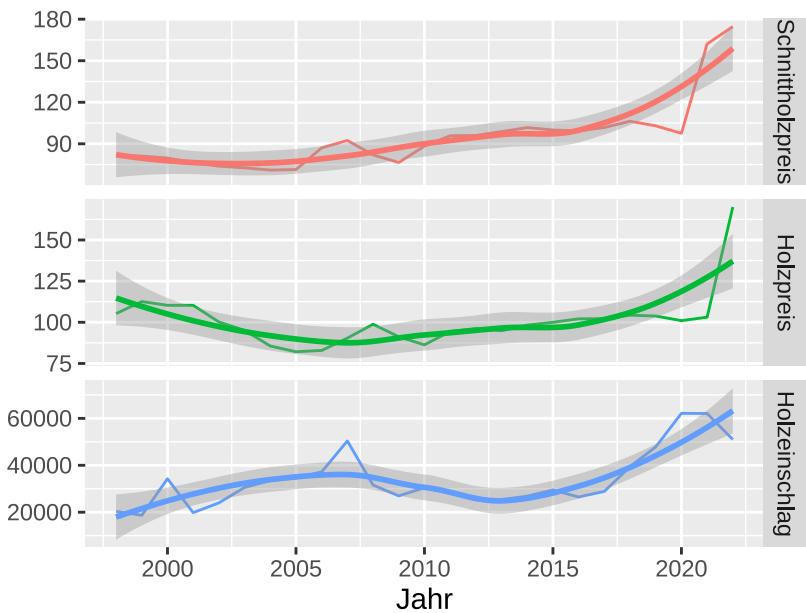
1852 Eine Trendlinie macht hier offensichtlich keinen Sinn. Die Trendlinie ist eine lineare Regression, also eine  
 1853 ordinäre Statistik, die wie eingangs erwähnt für Zeitreihen ungeeignet ist. Daher verwenden wir den sog.  
 1854 Loess-Glättter, um einen Trend in die Daten zu legen. Der Loess-Glättter ist ein sehr flexible Kurve. Wir sehen  
 1855 hier beispielsweise, dass der Leitholzpreis träge oder gar nicht auf das Angebot reagiert. Die Nachfrage jedoch  
 1856 zumindest in der einen Periode, in der sie stark steigt, den Holzpreis jedoch mit zeitlichem Verzug stark  
 1857 ansteigen lässt. Dieser visuelle Eindruck lässt sich durch spezifische Zeitreihen-Regressionen schätzen.

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



1858

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



1859

## 13 Aufgaben Wiederholen (for-Schleifen)

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

### 13.1 Schleifen

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

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

Die wesentlichen Befehle sind

- `for (i in X) {Code}`

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

- `while(Bedingung) {Code}`

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

- `break()`

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

#### 13.1.1 Wiederholen von Befehlen mit `for()`.

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

1896 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)
}
```

1897 ## [1] 1

1898 ## [1] 2

1899 ## [1] 3

1900 Das **i** steht in diesem Beispiel für die Schleifen-Variable. Sie muss nicht **i** heißen, sondern kann jeden  
 1901 zulässigen Namen annehmen. Das **X** steht für einen existierenden Vektor oder eine existierende Liste bzw.  
 1902 einen Ausdruck, der ein solches Objekt liefert (der Objektname ist ebenfalls frei wählbar). **for** und **in** sind  
 1903 Schlüsselworte, sie müssen, ebenso wie die runden Klammern, vorhanden sein.

1904 Im ersten Durchgang erhält die Schleifen-Variable **i** den ersten Wert von **X** und der Schleifenrumpf wird  
 1905 mit diesem Wert ausgeführt. Die Variable **i** nimmt nacheinander so lange die Werte von **X** an, bis ihr alle  
 1906 Elemente zugewiesen wurden.

1907 Das folgende Beispiel wird zwar besser durch die entsprechende Vektoroperation gelöst, zeigt aber sehr  
 1908 deutlich die Arbeitsweise der **for**-Schleife.

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

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

1909 ## [1] 4

1910 ## [1] 9

1911 ## [1] 25

1912

### 1913 Aufgabe 37: Schleifen 1

---

1915 Verwenden Sie den Vektor **k** <- c(1, 3, 9, 12, 15) und schreiben Sie folgende **for**-Schleifen:

- 1916 1. Eine Schleife, die jedes Element aus **k** ausgibt.
- 1917 2. Eine Schleife, die zu jedem Element aus **k** 10 addiert und den neuen Wert ausgibt.
- 1918 3. Eine Schleife wie in 2), aber der neue Wert (**k** + 10) soll jetzt nicht mehr ausgegeben werden, sondern  
 1919 in **k10** gespeichert werden. Stellen Sie sicher, dass **k10** wieder von der Länge 5 ist.

1920

---

1921 Die Funktion **for()** ermöglicht es, einen Befehl beliebig oft zu wiederholen. Z.B. der folgende Ausdruck zieht  
 1922 10-Mal eine Stichprobe der Größe 1 aus dem Vektor **v**. Beachten Sie, dass die Schleifen-Variable **i** selbst gar

1923 nicht im Schleifenrumpf vorkommt. Das Ziel dieser Schleife ist nicht die Elemente des Vektors abzuarbeiten,  
 1924 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))
}
```

```
1925 ## [1] 3
1926 ## [1] 1
1927 ## [1] 3
1928 ## [1] 3
1929 ## [1] 2
1930 ## [1] 3
1931 ## [1] 2
1932 ## [1] 2
1933 ## [1] 1
1934 ## [1] 4
```

1935 Auf gleiche Weise kann man auch über die Variablen eines Dataframes iterieren<sup>15</sup>. Das folgende Beispiel hat  
 1936 zum Ziel die Funktionsweise von Schleifen zu verdeutlichen. Schleifen haben in R jedoch den Nachteil, dass sie  
 1937 sehr langsam operieren. Wenn es geht, sollte man Alternativen verwenden. Die Funktionsweise wiederholender  
 1938 Auswertungen wird jedoch mit `for`-Schleifen deutlicher. Aus diesem Grunde werden wir uns in diesem Kurs  
 1939 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.

1948

---

**Aufgabe 38: for-Schleife**

---

1951 Lesen Sie den Datensatz `bhd_1.txt` ein und verwenden Sie eine `for`-Schleife.

- 1952 • Ziehen Sie 500-Mal je 35 Beobachtungen für den BHD.  
1953 • Berechnen Sie jeweils den Mittelwert aus den 35 Werten.  
1954 • Speichern Sie die 500 Mittelwerte in einem neuen Vektor `mittel`.  
1955 • Wie ist die Verteilung dieser 500 Mittelwerte zu interpretieren?

1956 **13.1.2 Wiederholen von Befehlen mit `while()`**

1957 Die `while`-Schleifen finden Anwendung, wenn die Anzahl der zu durchlaufenden Wiederholungen vorher  
1958 nicht bekannt ist, wie zum Beispiel bei Iterationsverfahren, die bis zu einer gewissen Genauigkeit durchlaufen  
1959 werden sollen. Die `while`-Schleife besteht in R aus dem Schlüsselwort `while()` und einer Bedingung in runden  
1960 Klammern.

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

1961 Sie ist in der praktischen Programmierung nicht so relevant wie die `for`-Schleife. Sie sei deshalb hier nur  
1962 kurz erwähnt. Die Abbruchbedingung wird jedes Mal geprüft bevor der Schleifenrumpf durchlaufen wird. Die  
1963 Bedingung wird ausgewertet und wenn diese `TRUE` ist, wird der Schleifenrumpf ausgeführt und danach erneut  
1964 die Bedingung überprüft. Ist die Bedingung nicht erfüllt, wird der Schleifenrumpf nicht durchgeführt und die  
1965 Schleife beendet. Ist die Bedingung bereits vor Eintritt in die Schleife nicht erfüllt, wird die Schleife gar nicht  
1966 erst durchlaufen.

1967 Da `while`-Schleifen also so lange ausgeführt werden, bis die Bedingung nicht mehr erfüllt ist, kann eine  
1968 Endlosschleife entstehen. Dies kann passieren, wenn man nicht sauber programmiert hat. Wenn innerhalb der  
1969 Schleife nicht dafür gesorgt wird, dass die Bedingung irgendwann nicht mehr erfüllt wird, so läuft die Schleife  
1970 immer weiter. Steckt R in einer Schleife fest und reagiert nicht mehr, kann der Befehl unter Linux mit `Strg`+  
1971 `C` und unter Windows mit `Esc` abgebrochen werden. Alternativ können Sie auf das rote STOP Symbol  
1972 über der Konsole klicken.

1973 **13.2 Bedingte Ausführung von Codeblöcken**

1974 Innerhalb eines Skripts ist es mitunter notwendig je nach aktueller Situation unterschiedlich fortzufahren.  
1975 Die Situation wird mit einem logischen Ausdruck, einer sogenannten Bedingung, geprüft. Je nachdem, ob  
1976 die Bedingung wahr (`TRUE`) oder falsch (`FALSE`) ist, werden unterschiedliche Programmteile ausgeführt, der  
1977 jeweils andere Teil bleibt unberücksichtigt. Danach wird in jedem Fall die Programmausführung, mit den  
1978 auf die bedingte Anweisungen folgenden Anweisung, fortgesetzt. In R kann dies mit dem if-else-Konstrukt  
1979 realisiert werden, welches aus den Schlüsselwörtern `if()` und `else` sowie der in runde Klammern gefassten  
1980 Bedingung besteht.

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

1981 Im folgenden Beispiel sollen die bisher abstrakt beschriebenen Vorgänge praktische Anwendung finden. In  
 1982 dem Beispiel wird zunächst durch zufälliges Ziehen einer Zahl aus dem Bereich eins bis sechs ein Würfelwurf  
 1983 simuliert. Anschließend wird der Würfelwurf mit einem if-Ausdruck kommentiert. Ist die Bedingung, es wurde  
 1984 eine Sechs gewürfelt, erfüllt, wird der Code innerhalb der geschweiften Klammern ausgeführt, ansonsten wird  
 1985 der Klammerinhalt ignoriert.

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

1986 In den meisten Fällen ist es für R irrelevant, ob sich zwischen den verschiedenen Elementen Leerzeichen oder  
 1987 Zeilenumbrüche befinden. Bei dem else-Ausdruck dagegen wird ein Fehler erzeugt, wenn vor dem `else` nicht  
 1988 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.")
}
```

1989 `## [1] "Beim nächsten Wurf klappt's bestimmt."`

1990

### 1991 Aufgabe 39: Bedingte Programmierung

---

- 1993 • Wenn keine 6 gewürfelt wurde, lassen Sie zusätzlich ausgeben welche Zahl gewürfelt wurde.  
 1994 • Wiederholen Sie den Würfelwurf 10 Mal.

## 1995 14 (R)markdown

### 1996 14.1 Markdown Grundlagen

1997 Die Idee von Markdown ist, einfach Text strukturieren zu können und das ganze ohne umfangreiche Programme  
 1998 zu erstellen. Im nächsten Abschnitt sehen wir dann, wie man Markdown und R-Code verbinden kann. Hier  
 1999 soll es jedoch erst einmal darum gehen, die einfachsten Bausteine von Markdown vorzustellen.

2000 Am Anfang jedes Dokuments kommt eine Präambel. Diese fängt mit --- an und hört auch wieder mit ---  
 2001 auf. Innerhalb der Präambel können dann Metainformationen über das Dokument festgelegt werden, dies  
 2002 beinhaltet im einfachsten Fall: Titel, Autor und Datum. Das würde dann so aussehen:

```
2003 ---
2004 title: "Ein Titel"
2005 author: "Der, der es geschrieben hat"
2006 date: "März 2021"
2007 ---
```

2008 Danach folgt strukturierter und formatierter Text. Verschiedene Hierarchieebenen von Überschriften können  
 2009 mit der Anzahl an # festgelegt werden. So ist eine Überschrift erster Ordnung # Kapitel eine Überschrift  
 2010 zweiter Ordnung ## Unterkapitel usw.

2011 Listen können erstellt werden, wenn man am Anfang jeder Zeile ein - oder 1. schreibt.

```
2012 - Erster Eintrag
2013 - Zweiter Eintrag
2014 - Dritter Eintrag
```

2015 wird zu

```
2016 • Erster Eintrag
2017 • Zweiter Eintrag
2018 • Dritter Eintrag
```

2019 Eine zentrale Idee von Markdown ist es Text einfach zu formatieren. Werden eine oder mehrere Wörter mit  
 2020 zwei Sternchen (\*\*) eingefasst wird dieser Text **fett** dargestellt. Also aus \*\*wichtig\*\* wird **wichtig**. Das  
 2021 gleiche funktioniert auch mit *kursiven* Text, jedoch muss man hier noch ein Sternchen verwenden, also aus  
 2022 \*kursiv\* wird *kursiv*. Soll ein text fett und kursiv sein, kann man drei Sternchen verwenden. Aus \*\*\*sehr  
 2023 wichtig\*\*\* wird dann **sehr wichtig**.

2024 Weitere Elemente wie Links oder Abbildungen können einfach eingebunden werden. Links werden mit [Link  
 2025 text](url) in den Text integriert. Beispielsweise ist eine gute Idee bei [stackoverflow](#) bei Problemen nach  
 2026 einer Lösung zu suchen. Dieser Link wurde mit [stackoverflow](www.stackoverflow.com) erstellt.

2027 Für Abbildungen gibt es einen ganz ähnlichen Syntax. Mit ! [Das R Logo](abb/r\_logo.png) wird die  
 2028 Abbildung r\_logo.png eingebunden mit der Beschriftung: "Das R Logo".



Abbildung 12: Das R Logo

2029

2030 **Aufgabe 40: Arbeiten mit markdown**

---

2032 Verwenden Sie das folgende Markdowndokument:

```

2033 ---
2034 title: "Dokument"
2035 author: "Ihr Name"
2036 date: "März 2021"
2037 ---
2038
2039 # Einleitung
2040
2041 # Methoden
2042 1. Kopieren Sie die Vorlage in ein Dokument, das test.md heißt.
2043 2. Fügen Sie zwei Überschriften zweiter und dritter Ordnung hinzu.
2044 3. Fügen Sie einen kursiven Text hinzu.
2045 4. Fügen Sie einen Link zu einer Website hinzu.
2046 5. Kompilieren Sie die Datei, indem Sie in Rstudio auf Preview drücken (Abbildung 13).

```

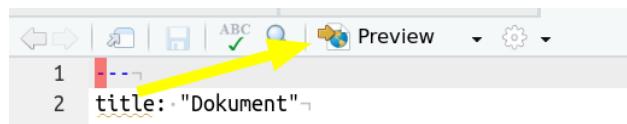


Abbildung 13: Kompilieren einer md-Datei.

2047 **14.2 R und Markdown**

2048 Markdown macht es bereits einfach Textdokumente und Dokumentationen zu verfassen, aber die wirkliche  
 2049 Stärke liegt in der Möglichkeit R und Markdown zu kombinieren. Man spricht dann von Rmarkdown. Ein  
 2050 weiteres Strukturelement, das wir noch nicht kennen gelernt haben, sind Code-Blöcke.

2051 ~~~

2052 a &lt;- 1:10

```

2053 a[1]
2054 ``
2055 erzeugt
2056 a <- 1:10
2057 a[1]

2058 Momentan wird noch kein Code ausgeführt, sondern lediglich der Code, als Code dargestellt. Rmarkdown
2059 bietet nun die Möglichkeit Code beim kompilieren16 auszuführen. Dafür müssen wir nur einen Code-Block als
2060 R-Code-Block kennzeichnen.

2061 ``{R}
2062 a <- 1:10
2063 a[1]
2064 ``
2065 erzeugt
2066 a <- 1:10
2067 a[1]

2066 ## [1] 1

2067 Beachte, die Variable a wird beim kompilieren erzeugt und steht dann R zur Verfügung. R-Code-Blöcke
2068 werden auch als Code Chunks bezeichnet. Diese Chunks können sehr genau angesprochen und angepasst
2069 werden. Einige wichtige Argumente sind:
2070 • echo: Gibt an, ob der Quelltext angezeigt werden soll oder nicht.
2071 • result: Gibt an, ob die Ergebnisse gezeigt werden sollen oder nicht.
2072 • eval: Diese Option gibt an ob der Chunk ausgeführt werden soll oder nicht.

```

2073

---

**2074 Aufgabe 41: Arbeiten mit Rmarkdown**


---

2076 Erstellen Sie eine neue Rmarkdown Datei mit dem Namen `test1.Rmd`. Erstellen Sie zwei Code-Chunks. Der  
2077 erste soll nicht angezeigt werden und darin werden die Daten geladen (`bhd_1.txt`). Im zweiten Chunk plotten  
2078 Sie das Alter der Bäume gegen den BHD. Was passiert mit dem Plot, wenn Sie die Datei kompilieren (drücken  
2079 Sie dazu auf den Knit-Knopf; Abbildung 14).



Abbildung 14: Kompilieren einer `Rmd`-Datei.

---

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

## 2080 15 Räumliche Daten in R

### 2081 15.1 Was sind räumliche Daten

2082 Räumliche Daten sind Beobachtungen, wie wir sie schon oft gesehen haben, mit einem räumlichen Bezug. Der  
 2083 Unterschied zu nicht räumlichen Daten liegt darin, dass räumliche Daten eindeutig im *Raum* verortet werden  
 2084 können. Häufig werden sogenannte Geoinformationssysteme (GIS) zum Arbeiten mit räumlichen verwendet.  
 2085 R kann in vielerlei Hinsicht wie ein GIS eingesetzt werden und hier werden einige Grundfunktionalitäten  
 2086 dafür besprochen. Räumliche Daten werden in zwei unterschiedliche Datentypen unterteilt: Vektor- und  
 2087 Rasterdaten. Vektordaten modellieren einzelne Objekte (= *Features*). Rasterdaten modellieren eine Oberfläche.  
 2088 Vektordaten bestehen aus zwei Komponenten: 1) einer Geometrie, die die Form und Lage der Daten definiert  
 2089 und 2) Attributen, den tatsächlichen Daten. Räumliche Daten werden oft als *Features* bezeichnet. Ein Feature  
 2090 ist die räumliche Einheit einer Beobachtung. Je nach Art der räumlichen Daten können Features entweder  
 2091 Punkte (z.B. ein Baum), Linien (z.B. eine Straße) oder Polygone (z.B. ein See) sein. Auch können mehrere  
 2092 Geometrien zu einem Feature zusammengefasst werden. Ein Beispiel wäre eine Beobachtung für ein Land,  
 2093 das aber aus mehreren Polygonen bestehen kann (z.B. Festland und Inseln). Features können dann weitere  
 2094 Attribute (= Attributdaten) haben, z.B. eine ID, Name oder was auch immer man gemessen hat.  
 2095 Rasterdaten bestehen aus einer Oberfläche von gleichgroßen Kacheln (= *Pixel*), die ein Gebiet abdecken.  
 2096 Meist sind Pixel viereckig, aber das ist keine Voraussetzung. Dabei hat jedes Pixel einen Wert (das kann  
 2097 auch ein fehlender Wert sein). Typische Beispiele für Rasterdaten sind Landnutzung oder Seehöhen.  
 2098 In R kann sowohl mit Vektor- als auch mit Rasterdaten gearbeitet werden. Für Vektordaten bietet sich das  
 2099 Paket **sf** an und für Rasterdaten das Paket **raster**.

### 2100 15.2 Koordinatenbezugssystem

2101 Eine der Herausforderungen für räumliche Daten ist die eindeutige Verortung im Raum. Dazu braucht man  
 2102 ein *Koordinatenbezugssystem (KBS)*. Einem KBS liegt ein mathematisches Modell der Erde zugrunde. Die  
 2103 Details zu KBS werden schnell relativ kompliziert und wir beschränken uns hier lediglich darauf, wie KBS  
 2104 verwendet werden können. Dazu müssen wir zwei Fälle unterscheiden: 1) einem Datensatz ein KBS zuweisen  
 2105 und 2) Transformation des KBS eines Datensatzes in ein anderes KBS. Die technischen Details werden in  
 2106 den folgend Abschnitten besprochen. Für beide Aufgaben müssen wir auf ein KBS verweisen können, ein  
 2107 einfacher Ansatz dafür sind die sogenannten *EPSG-Codes*<sup>17</sup>.

### 2108 15.3 Vektordaten in R

2109 Das Paket **sf** stellt Klassen zum Abbilden von Features zur Verfügung, die dann in einem **data.frame** als  
 2110 Liste gespeichert werden können. In der Regel erstellen wir Features nicht individuell, sondern lesen diese aus  
 2111 externen Datenquellen (z.B. ESRI Shapefile) ein. Zum besseren Verständnis, erstellen wir es einmal manuell.  
 2112 Wir haben die Koordinaten für drei Städte (Göttingen, Hannover und Berlin) als geografische Koordinaten  
 2113 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))
```

2114 Daraus können wir jetzt eine Geometriespalte für einen `data.frame` erstellen

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

2115 Somit haben wir die Geometrie in der Variable `geom` gespeichert, aber noch keine dazugehörigen Attributdaten.

2116 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)
)
```

2117 In einem letzten Schritt möchten wir jetzt die Geometrie (`geom`) und die Attributdaten (`attr`) zusammenführen.

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

```
2119 ## Simple feature collection with 3 features and 3 fields
2120 ## Geometry type: POINT
2121 ## Dimension: XY
2122 ## Bounding box: xmin: 9.732 ymin: 51.5413 xmax: 13.405 ymax: 52.52
2123 ## Geodetic CRS: WGS 84
2124 ## name bundesland einwohner geom
2125 ## 1 Goettingen Niedersachsen 119000 POINT (9.9158 51.5413)
2126 ## 2 Hannover Niedersachsen 532000 POINT (9.732 52.3759)
2127 ## 3 Berlin Berlin 3650000 POINT (13.405 52.52)
```

2128 Wir können nun mit `staedte` genau so arbeiten wie mit jedem anderen `data.frame` und die Geometrien  
2129 werden immer ‘berücksichtigt’. Zusätzlich kann man eine Reihe von geometrischen Operationen durchführen.

2130 Wenn ein `data.frame` Punkten hat, kann man dies relative einfach mit der Funktion `st_as_sf()` “räumlich”  
2131 machen. Für das vorherige Beispiel würden wir zuerst einen `data.frame` mit allen Informationen (zur  
2132 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)
)
```

2133 Dann kann man mit der Funktion `st_as_sf()` weiter arbeiten:

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

## 2134 15.4 Arbeiten mit Vektordaten

2135 Es gibt sehr viele Funktionen, um mit räumlichen Daten zu arbeiten, von denen wir hier einige vorstellen.

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

```
2136 ## Coordinate Reference System:
2137 ## User input: EPSG:4326
2138 ## wkt:
2139 ## GEOGCRS["WGS 84",
2140 ## ENSEMBLE["World Geodetic System 1984 ensemble",
2141 ## MEMBER["World Geodetic System 1984 (Transit)"],
2142 ## MEMBER["World Geodetic System 1984 (G730)"],
2143 ## MEMBER["World Geodetic System 1984 (G873)"],
2144 ## MEMBER["World Geodetic System 1984 (G1150)"],
2145 ## MEMBER["World Geodetic System 1984 (G1674)"],
2146 ## MEMBER["World Geodetic System 1984 (G1762)"],
2147 ## MEMBER["World Geodetic System 1984 (G2139)"],
2148 ## ELLIPSOID["WGS 84",6378137,298.257223563,
2149 ## LENGTHUNIT["metre",1]],
2150 ## ENSEMBLEACCURACY[2.0]],
2151 ## PRIMEM["Greenwich",0,
2152 ## ANGLEUNIT["degree",0.0174532925199433]],
2153 ## CS[ellipsoidal,2],
2154 ## AXIS["geodetic latitude (Lat)",north,
2155 ## ORDER[1],
2156 ## ANGLEUNIT["degree",0.0174532925199433]],
2157 ## AXIS["geodetic longitude (Lon)",east,
2158 ## ORDER[2],
2159 ## ANGLEUNIT["degree",0.0174532925199433]],
2160 ## USAGE[
2161 ## SCOPE["Horizontal component of 3D system."],
2162 ## AREA["World."],
2163 ## BBOX[-90,-180,90,180]],
2164 ## ID["EPSG",4326]]
```

2165 Wenn wir jetzt zu einem anderen KBS (z.B. EPSG:3035, ein europäisches projiziertes KBS) umrechnen  
 2166 möchten, können wir das mit

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

```

2167 ## Coordinate Reference System:
2168 ## User input: EPSG:3035
2169 ## wkt:
2170 ## PROJCRS["ETRS89-extended / LAEA Europe",
2171 ## BASEGEOGCRS["ETRS89",
2172 ## ENSEMBLE["European Terrestrial Reference System 1989 ensemble",
2173 ## MEMBER["European Terrestrial Reference Frame 1989"],
2174 ## MEMBER["European Terrestrial Reference Frame 1990"],
2175 ## MEMBER["European Terrestrial Reference Frame 1991"],
2176 ## MEMBER["European Terrestrial Reference Frame 1992"],
2177 ## MEMBER["European Terrestrial Reference Frame 1993"],
2178 ## MEMBER["European Terrestrial Reference Frame 1994"],
2179 ## MEMBER["European Terrestrial Reference Frame 1996"],
2180 ## MEMBER["European Terrestrial Reference Frame 1997"],
2181 ## MEMBER["European Terrestrial Reference Frame 2000"],
2182 ## MEMBER["European Terrestrial Reference Frame 2005"],
2183 ## MEMBER["European Terrestrial Reference Frame 2014"],
2184 ## ELLIPSOID["GRS 1980",6378137,298.257222101,
2185 ## LENGTHUNIT["metre",1]],
2186 ## ENSEMBLEACCURACY[0.1]],
2187 ## PRIMEM["Greenwich",0,
2188 ## ANGLEUNIT["degree",0.0174532925199433]],
2189 ## ID["EPSG",4258]],
2190 ## CONVERSION["Europe Equal Area 2001",
2191 ## METHOD["Lambert Azimuthal Equal Area",
2192 ## ID["EPSG",9820]],
2193 ## PARAMETER["Latitude of natural origin",52,
2194 ## ANGLEUNIT["degree",0.0174532925199433],
2195 ## ID["EPSG",8801]],
2196 ## PARAMETER["Longitude of natural origin",10,
2197 ## ANGLEUNIT["degree",0.0174532925199433],
2198 ## ID["EPSG",8802]],
2199 ## PARAMETER["False easting",4321000,
2200 ## LENGTHUNIT["metre",1],
2201 ## ID["EPSG",8806]],
2202 ## PARAMETER["False northing",3210000,
2203 ## LENGTHUNIT["metre",1],
2204 ## ID["EPSG",8807]]],
2205 ## CS[Cartesian,2],
2206 ## AXIS["northing (Y)",north,
2207 ## ORDER[1],
2208 ## LENGTHUNIT["metre",1]],
2209 ## AXIS["easting (X)",east,

```

```

2210 ## ORDER[2] ,
2211 ## LENGTHUNIT["metre",1]],
2212 ## USAGE[
2213 ## SCOPE["Statistical analysis."],
2214 ## AREA["Europe - European Union (EU) countries and candidates. Europe - onshore and offshore: "],
2215 ## BBOX[24.6,-35.58,84.73,44.83]],
2216 ## ID["EPSG",3035]

```

2217 Die Funktion `st_buffer()` erlaubt es Features zu puffern, mit `st_distance()` kann die Distanz zwischen  
2218 Features berechnet werden, mit `st_area()` kann die Fläche eines Features berechnet werden.

2219 Funktionen wie `st_intersection()`, `st_union()` und `st_difference()` erlauben es geometrische Opera-  
2220 tionen zwischen unterschiedlichen Features zu berechnen. Für eine ausführliche Diskussion siehe auch hier:  
2221 <https://geocompr.robinlovelace.net/geometric-operations.html>.

2222 Normalerweise lesen wir Daten von externen Datenquellen (z.B. ESRI Shapefiles). Das geht mit der Funktion  
2223 `st_read()`.

## 2224 15.5 Rasterdaten in R

2225 Für Rasterdaten gibt es das R-Paket `terra`. Auch hier wollen wir uns wieder auf einige Grundfunktionalitäten  
2226 konzentrieren. Diese umfassen das Einlesen, Zuschneiden, Rechnen und Abfragen von Rastern.

2227 Mit der Funktion `rast()` kann ein Raster in R eingelesen werden.

```

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

```

2228 `dem` steht für *Digital Elevation Model* und ist ein Raster mit den Seehöhen in Niedersachsen mit einer  
2229 500-m-Auflösung. Wir können diese mit der Funktion `res()`<sup>18</sup> abfragen.

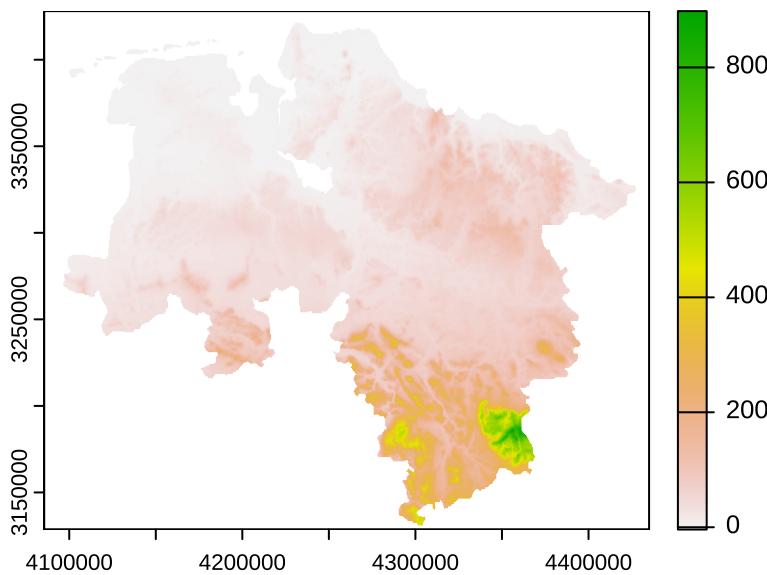
```
res(dem)
```

2230 ## [1] 500 500

2231 Bzw. wir können den Raster auch plotten.

```
plot(dem)
```

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



2232

2233 Wenn wir den Raster `dem` auf ein Gebiet zuschneiden wollen (z.B. Göttingen), müssen wir drei Schritte  
2234 durchführen. Als erstes müssen wir ein Shapefile für Göttingen einlesen.

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

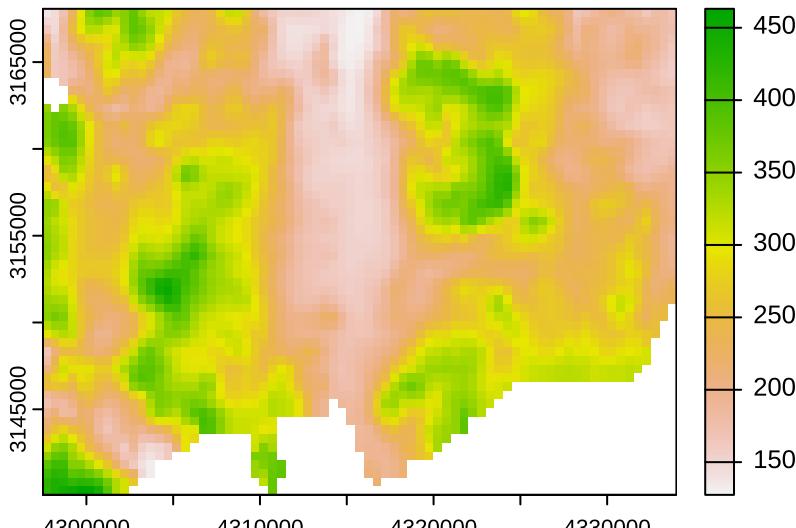
2235 Dann müssen wir sicherstellen, dass sowohl der Raster `dem` als auch das `sf`-Objekt `goe` im selben KBS sind.

2236 Es bietet es sich in der Regel an, das KBS des Vektors zu transformieren. Mit der Funktion `projectRaster()`  
2237 kann das KBS eines Rasters transformiert werden.

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

2238 Mit der Funktion `crop()` kann der Raster `dem` auf Göttingen zugeschnitten werden.

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



2239

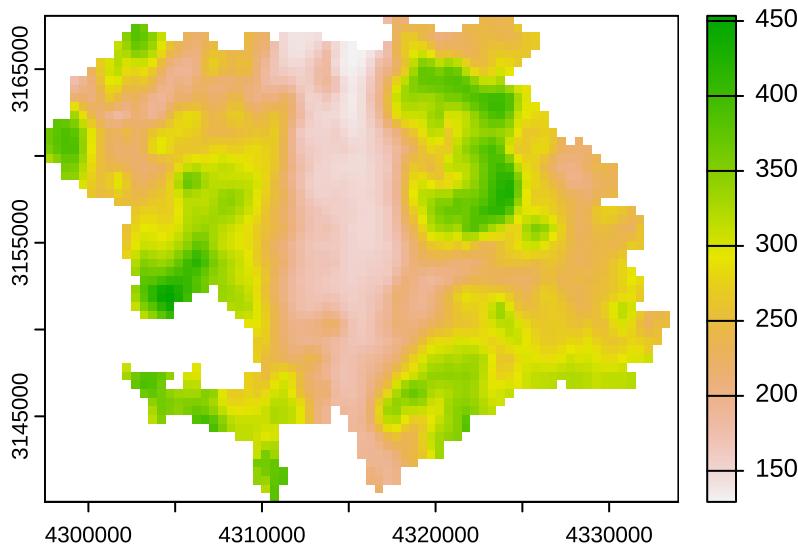
2240 Der Raster hat jetzt die Größe einer Bounding-Box (BBX) von Göttingen (das ist ein Rechteck, das Göttingen  
2241 umfasst). Mit der Funktion `mask()` kann der Raster auf die genauen Grenzen des Vektors `goe` angepasst

2242 werden.

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

2243 ## Warning: [mask] CRS do not match

```
plot(dem2)
```



2244

2245 Wenn wir an bestimmten Punkten den Wert des Rasters abfragen wollen (z.B. an `cities`) von vorhin, dann  
2246 gibt es dafür die Funktion `extract`. Dann müssen wir erst sicherstellen, dass `staedte` und `dem` gleichen KBS  
2247 zu grunde liegt. Dafür transformieren wir einfach `staedte` in das KBS von `dem`. Mit der Funktion `crs()`  
2248 erhalten wir das KBS des Rasters.

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

2249 Wenn wir das KBS eines Objektes nicht kennen, können wir auch einfach das KBS übergeben. Der folgende  
2250 Code-Block macht genau das Gleiche mit dem Vorteil, dass wir keinen EPSG-Code angeben müssen.

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

2251 Dann können wir für jede Stadt die Seehöhe abfragen:

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

2252 ## ID dem\_3035

2253 ## 1 1 149.18181

2254 ## 2 2 57.21486

2255 ## 3 3 NA

2256 Mit `terra::extract()` rufen wir *eindeutig* die Funktion `extract()` aus dem Paket `terra` auf. Wir müssen  
2257 das so machen, weil es im Paket `dplyr` auch eine Funktion `extract()` gibt, die wir hier nicht anwenden  
2258 möchten, da sie einen Fehler verursachen würde.

2259 Ein analoges Vorgehen ist auch für Linien und Polygone möglich.

2260 Mit Rastern kann auch einfach gerechnet werden. Wir können z.B. die Seehöhe in Kilometern anstatt Metern

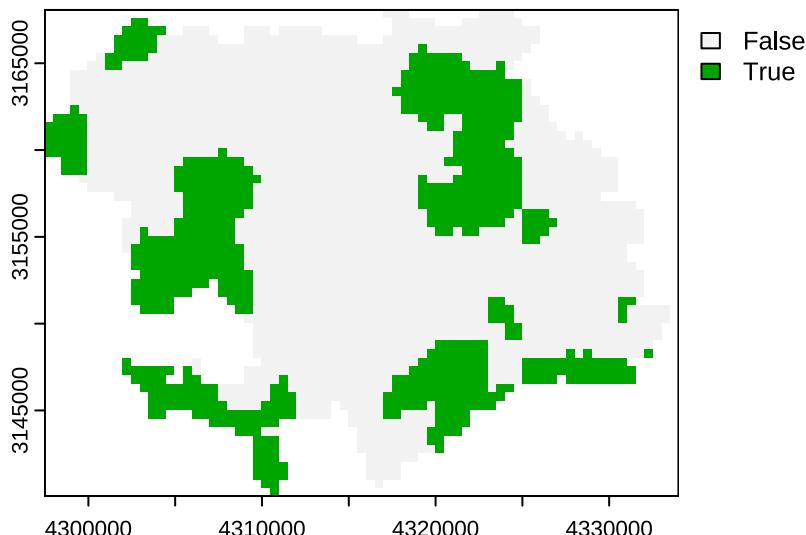
2261 berechnen:

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

2262 Auch logische Operationen sind möglich, wenn wir alle Rasterzellen mit einer Seehöhe von mehr als 300 m in  
2263 Göttingen suchen, dann geht das so:

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

```
plot(dem3)
```



2264

2265 Wenn wir jetzt auf die Werte des Rasters `dem3` zugreifen wollen, geht das mit eckigen Klammern.

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

```
2266 ## dem_3035
2267 ## [1,] NA
2268 ## [2,] NA
2269 ## [3,] NA
2270 ## [4,] NA
2271 ## [5,] NA
2272 ## [6,] NA
```

2273 Das sind erst einmal viele NA-Werte für die ganzen Zellen, die außerhalb von Niedersachsen liegen. Aber wir  
2274 können mit so einem Vektor ganz normal arbeiten und z.B. die Fläche des Landes Niedersachsen die eine  
2275 Seehöhe von mehr als 500m Seehöhe hat ausrechnen.

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

```
2276 ## [1] 0.2786229
```

2277

---

2278 **Aufgabe 42: Arbeiten mit Rastern**

---

2280 Verwenden Sie den Raster `wald.tif`, der auf einer 10 m Auflösung den Waldanteil jeder Rasterzelle angibt<sup>19</sup>.  
 2281 Der EPSG-Code für das KBS von `wald.tif` ist 3035. Nehmen Sie an, dass wenn der Waldanteil in einer  
 2282 Raster größer als 50 % ist, dass die Rasterzelle als Wald klassifiziert werden kann. Wie viel Prozent des  
 2283 Göttinger Stadtgebietes sind Wald? Wie ändert sich dieser Wert, wenn sie 70 % anstatt 50 % als Schwellenwert  
 2284 für Wald annehmen?

2285

---

2286 **Aufgabe 43: Studiendesign**

---

2288 Mit der Funktion `st_sample()` können Sie innerhalb oder entlang eines Features zufällige Punkte legen. Das  
 2289 Argument `n` steuert die Anzahl Punkte und das Argument `type` wie die Punkte angeordnet werden. Für `type`  
 2290 sind für uns die Werte `type = "random"` (komplett zufällig), `type = "regular"` (regelmäßiger Grid) und  
 2291 `type = "hexagonal"` von Bedeutung (ein hexagonaler Grid, d.h. ein sechseckiger Raster). Unglücklicherweise  
 2292 ist das Ergebnis von `st_sample()` erst eine Geometrie. Um daraus ein vollständiges `sf`-Objekt zu machen  
 2293 und problemlos weiter arbeiten zu können, müssen Sie nocheinmal die Funktion `st_as_sf()` ausführen.  
 2294 Stellen Sie sich vor, dass wir die tatsächliche Waldbedeckung des Göttinger Stadgebietes **nicht** kennen und wir  
 2295 eine Studie durchführen, um den Anteil des Göttinger Stadgebietes, der mit Wald bedeckt ist herauszufinden.  
 2296 Erstellen Sie dafür einige unterschiedliche Stichproben (diese können in der Anzahl und Anordnung variieren).  
 2297 Berechnen Sie für jedes Stichprobendesign den Anteil an Wald und ein dazugehöriges Konfidenzintervall  
 2298 (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  $n$   
 2299 die Stichprobengröße). Nehmen Sie an, dass eine Rasterzelle Wald ist, sobald > 50 % der Rasterzelle mit  
 2300 Wald bedeckt ist.

2301

---

2302 **Aufgabe 44: Räumliche Daten**

---

2304 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)
)
```

2305 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

- 2306 2. Puffern Sie jeden Baum mit seinem Kronendurchmesser.
- 2307 3. Berechnen Sie die Kronenfläche jedes Baumes. *Hinweis: Die Funktion st\_area() könnte dafür hilfreich sein.*
- 2309 4. Welcher Baum hat die größte Kronenfläche?
- 2310 5. Finden Sie für jede Art, den Baum mit der größten Kronenfläche.

2311

2312 **Aufgabe 45: Arbeiten mit räumlichen Daten**

---

- 2314 1. Lesen Sie das ESRI Shapefile goettingen/stadt\_goettingen.shp ein.
- 2315 2. Wie viele Features befinden sich in dem Shapefile?
- 2316 3. Welches Koordinatenbezugssystem (KBS) hat das Shapefile?
- 2317 4. Transformieren Sie das Shapefile in das KBS 3035.
- 2318 5. Erstellen Sie eine neue Spalte A in der Sie die Fläche jeder Gemeinde/Stadt speichern.
- 2319 6. Welche Gemeinde/Stadt (Spalte GEN) ist am größten?
- 2320 7. Wählen Sie nun nur die Stadt Göttingen aus.

2321

2322 **Aufgabe 46: Arbeiten mit räumlichen Daten 2**

---

- 2324 1. Lesen Sie erneut das ESRI Shapefile goettingen/stadt\_goettingen.shp ein.
- 2325 2. Lösen sie die Gemeindegrenzen auf (die Funktion st\_union() könnte hier nützlich sein).
- 2326 3. Wie groß ist das resultierende Feature?

2327 **16 FAQs (Oft gefragtes)**

2328 **16.1 Arbeiten mit Daten**

2329 **16.1.1 Einlesen von Exceldateien**

- 2330 Mit der Funktion `read_excel()` aus dem Paket `readxl` können Exceldateien direkt in R eingelesen werden.  
2331 Ein Export als csv-Datei aus Excel ist nicht notwendig.

## 2332 17 Literatur

- 2333 Ein guter Überblick über viele der angeschnittenen Themen gibt es in dem [R for data science](#), das online frei  
2334 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.  
2335
- 2336 McNamara, Amelia, and Nicholas J Horton. 2018. “Wrangling Categorical Data in r.” *The American Statistician*  
2337 72 (1): 97–104.
- 2338 Wickham, Hadley. 2014. “Tidy Data.” *Journal of Statistical Software* 59 (10). <https://doi.org/10.18637/jss.v059.i10>.
- 2339