

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: 3. 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)	21
62	4.6 Der %in%-Operator	23
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	28
67	6.1 NA	28
68	6.2 NULL	29
69	6.3 Inf	29
70	7 data.frames oder Tabellen	31
71	7.1 Wichtige Funktionen zum Arbeiten mit data.frames	32
72	7.2 Zugreifen auf Elemente eines data.frame	33
73	8 Schreiben und lesen von Daten	36
74	8.1 Textdateien	36
75	9 Erstellen von Abbildungen	38
76	9.1 Base Plot	38
77	9.1.1 Mehrere Panels	44
78	9.1.2 Speichern von Abbildungen	44

79	9.2 Histogramme	45
80	9.3 Boxplots	48
81	9.4 ggplot2: Eine Alternative für Abbildungen	50
82	9.4.1 Multipanel Abbildungen	57
83	9.4.2 Plots kombinieren	60
84	9.4.3 Speichern von plots	62
85	10 Mit Daten arbeiten	64
86	10.1 dplyr eine Einführung	64
87	10.2 Arbeiten mit gruppierten Daten	67
88	10.3 pipes oder %>%	68
89	10.4 Joins	69
90	10.5 ‘long’ and ‘wide’ Datenformate	71
91	10.6 Auswählen von Variablen	73
92	10.7 Einzelne Beobachtungen abfragen (slice())	74
93	10.8 Spalten trennen	77
94	11 Arbeiten mit Text	79
95	11.1 Arbeiten mit Text	79
96	11.2 Finden von Textmustern	80
97	12 Arbeiten mit Zeit	83
98	12.1 Arbeiten mit Zeitintervallen	84
99	12.2 Formatieren von Zeit	86
100	12.3 Zeitreihen	86
101	13 Aufgaben Wiederholen (for-Schleifen)	92
102	13.1 Schleifen	92
103	13.1.1 Wiederholen von Befehlen mit for().	92
104	13.1.2 Wiederholen von Befehlen mit while()	95
105	13.2 Bedingte Ausführung von Codeblöcken	95
106	14 (R)markdown	97
107	14.1 Markdown Grundlagen	97
108	14.2 R und Markdown	98
109	15 Räumliche Daten in R	100
110	15.1 Was sind räumliche Daten	100
111	15.2 Koordinatenbezugssystem	100
112	15.3 Vektordaten in R	100
113	15.4 Arbeiten mit Vektordaten	102
114	15.5 Rasterdaten in R	104
115	16 FAQs (Oft gefragtes)	110
116	16.1 Arbeiten mit Daten	110
117	16.1.1 Einlesen von Exceldateien	110

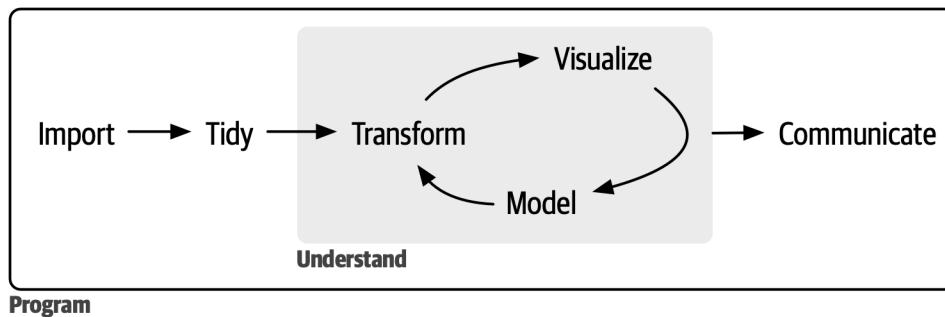
118 **17 Literatur**

111

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]**).

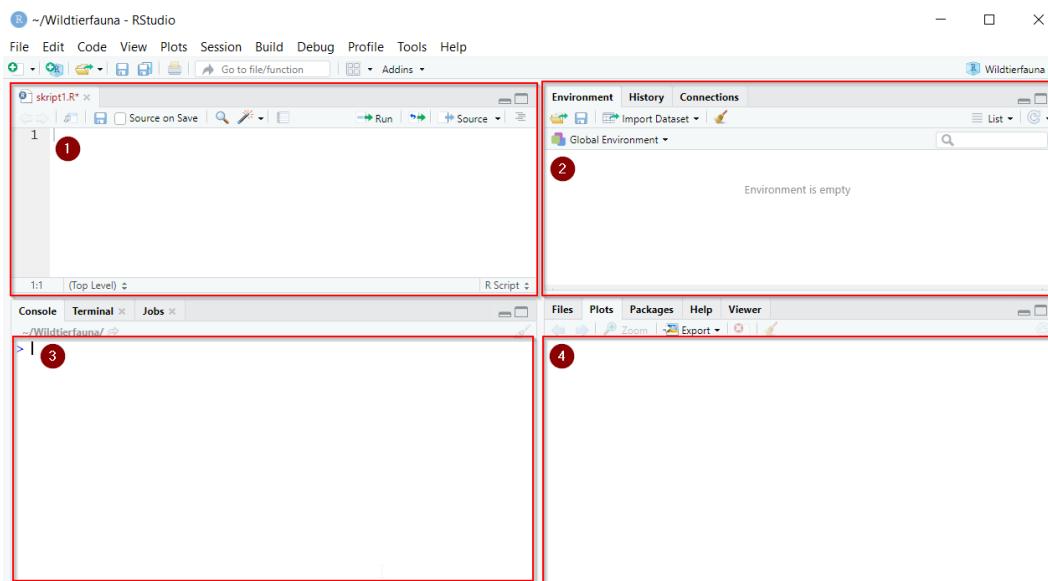


Abbildung 1: RStudio Panes.

- 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*+*↑*+*↵*).

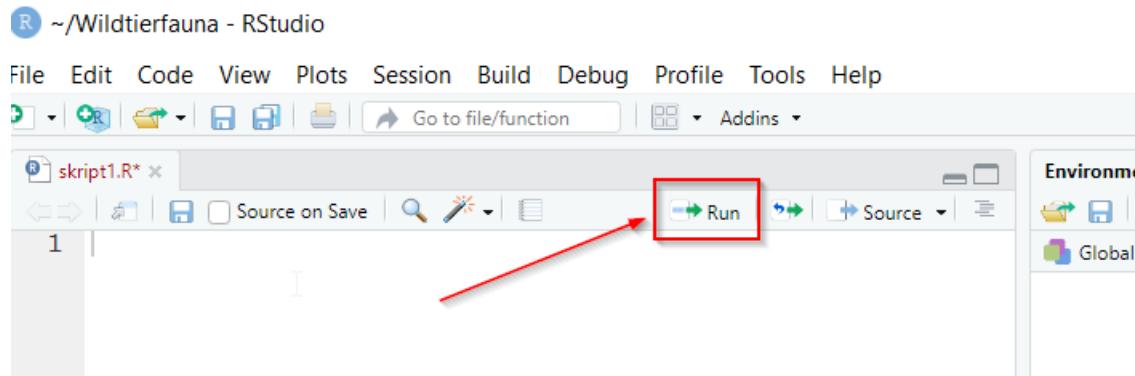


Abbildung 2: Zeilenweises Ausführen von Code in RStudio.

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**.

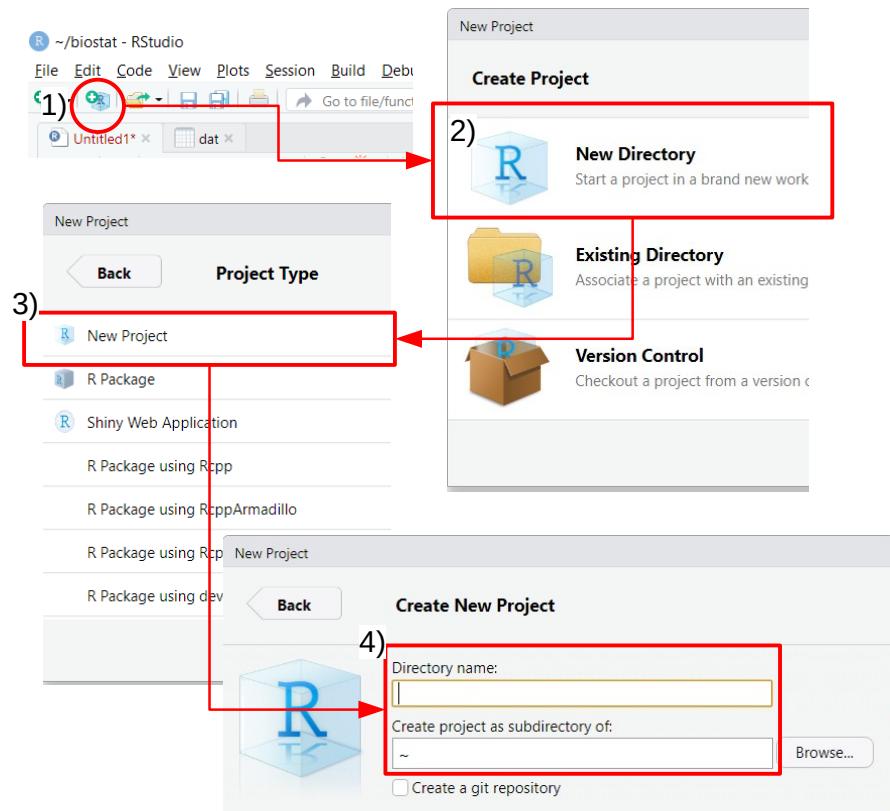


Abbildung 3: Workflow zum Erstellen eines Projekts.

- 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).

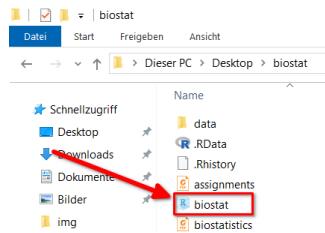


Abbildung 4: Öffnen eines RStudio Projekts.



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

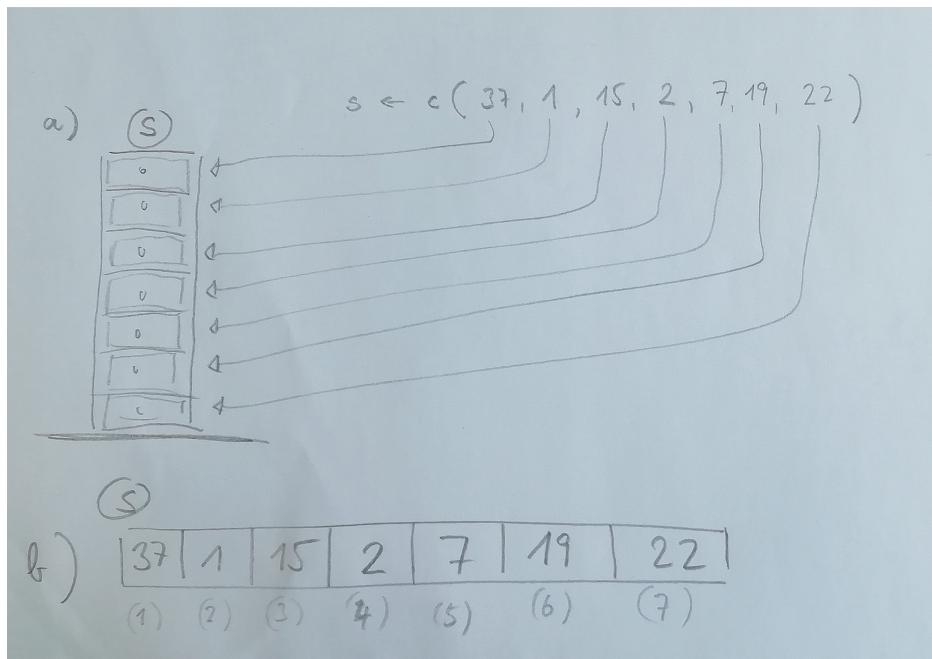


Abbildung 6: Schematische Darstellung eines Vektors in R.

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
 379 des Moduls kennenlernen). Die wichtigste Funktion ist `c()`. Dabei steht `c` für *combine* oder *concatenate*.
 380 Die Funktion `c()` fügt einzelne Elemente in einen Vektor zusammen (und zwar in der Reihenfolge wie diese
 381 Elemente an `c()` übergeben werden). `c()` ist sozusagen die Funktion, die für Sie mehrere Elemente zu einem
 382 Vektor 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:

s

388 ## [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 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 + 10

394 ## [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
396 zunächst nicht um Vektorrechnungen handelt, wie Sie sie aus der linearen Algebra kennen. Für die sog.
397 Matrizenoperationen der linearen Algebra werden die Operatoren in R mit % % umschlossen, also bspw. `s`
398 %*% `s`.

s * s

399 ## [1] 1369 1 225 4 49 361 484

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

seq(from = 1, to = 10)

403 ## [1] 1 2 3 4 5 6 7 8 9 10

(1 : 10)

404 ## [1] 1 2 3 4 5 6 7 8 9 10

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

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

406 ## [1] 1 3 5 7 9

407

Aufgabe 3: Vektoren erstellen

- 410 Sie haben den BHD (Brusthöhendurchmesser) in cm von vier Bäumen gemessen: 13, 15.3, 23, 9
- 411 • Erstellen Sie einen Vektor mit dem Namen `bhd` in dem Sie die Werte speichern
- 412 • Transformieren Sie die BHD-Werte in mm.
- 413 • Berechnen Sie die Fläche des BHD in cm^2 (nehmen Sie dafür an, dass ein Baum kreisrund ist).

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

4.1 Funktionen zum Arbeiten mit Vektoren

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

```
head(s)
```

417 ## [1] 37 1 15 2 7 19

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

418 ## [1] 37 1 15

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

419 ## [1] 19 22

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

```
length(s)
```

421 ## [1] 7

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

```
class(s)
```

423 ## [1] "numeric"

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

```
unique(s)
```

425 ## [1] 37 1 15 2 7 19 22

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

```
table(s)
```

427 ## s

428 ## 1 2 7 15 19 22 37

429 ## 1 1 1 1 1 1 1

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

```
rev(s)
```

432 ## [1] 22 19 7 2 15 1 37

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

```
sort(s)
```

434 ## [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.

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

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

436 ## [1] 37 1 15 2 7 19 22 37 1 15 2 7 19 22

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

```
a <- 1:4
rep(a, times = 2)
```

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

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

440 ## [1] 1 1 2 2 3 3 4 4

441

442 Aufgabe 4: Arbeiten mit Vektoren

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

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

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

447 4.2 Statistische Funktionen

448 Zahlreiche statistische Funktionen können auf Vektoren angewendet werden, hier sind nur drei Beispiele aufge-
449 füht: `mean()` berechnet den Mittelwert, `median()` berechnet den Median und `sd()` die Standardabweichung.

```
mean(s)
```

450 ## [1] 14.71429

```
median(s)
```

451 ## [1] 15

```
sd(s)
```

452 ## [1] 12.76341

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

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

456 ## [1] 1

```

sample(s, size = 3) # 2 Elemente

457 ## [1] 15 7 22
458 Wenn size weggelassen wird, dann bekommt man gleich viele Elemente zurück (wie der Vektor lang ist), d.h.
459 der Vektor wird nur permutiert.

```

460 4.3 Beispiel Fotofallen

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

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

```

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

```

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
477 Fall wäre das also.

```

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

478 ## [1] "Kamera_1"   "Kamera_2"   "Kamera_3"   "Kamera_4"   "Kamera_5"   "Kamera_6"
479 ## [7] "Kamera_7"   "Kamera_8"   "Kamera_9"   "Kamera_10"  "Kamera_11"  "Kamera_12"
480 ## [13] "Kamera_13"  "Kamera_14"  "Kamera_15"

```

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

```
483 rep(c("Revier A", "Revier B", "Revier C"), 5)
484 ## [1] "Revier A" "Revier B" "Revier C" "Revier A" "Revier B" "Revier C"
485 ## [7] "Revier A" "Revier B" "Revier C" "Revier A" "Revier B" "Revier C"
486 ## [13] "Revier A" "Revier B" "Revier C"
```

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

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

```
reviere
```

```
488 ## [1] "Revier A" "Revier A" "Revier A" "Revier A" "Revier A" "Revier B"
489 ## [7] "Revier B" "Revier B" "Revier B" "Revier B" "Revier C" "Revier C"
490 ## [13] "Revier C" "Revier C" "Revier C"
```

491

492 Aufgabe 5: Statistische Funktionen

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

495 2. Erstellen Sie die folgende Konsolenausgabe:

```
496 ## [1] "Die mittlere Anzahl von Rehfotos beträgt 171.8 Rehe pro Standort."
```

497 4.4 Arbeiten mit logischen Werten

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

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

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

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

```
anzahl_rehe > 100
```

```
507 ## [1] TRUE FALSE TRUE FALSE TRUE TRUE FALSE TRUE TRUE FALSE TRUE
508 ## [13] FALSE TRUE TRUE
```

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

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

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

```
511 ## [1] FALSE FALSE FALSE FALSE FALSE TRUE TRUE TRUE TRUE TRUE FALSE FALSE
512 ## [13] FALSE FALSE FALSE
```

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

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

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

```
519 ## [1] FALSE FALSE FALSE FALSE FALSE TRUE FALSE TRUE TRUE TRUE FALSE FALSE
520 ## [13] FALSE FALSE FALSE
```

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

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

```
524 ## [1] TRUE FALSE TRUE FALSE TRUE TRUE TRUE TRUE TRUE TRUE FALSE TRUE
525 ## [13] FALSE TRUE TRUE
```

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

528

Aufgabe 6: Arbeiten mit logischen Werten

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

- 532 1. TRUE | FALSE
- 533 2. FALSE & TRUE
- 534 3. (FALSE & TRUE) | TRUE
- 535 4. (2 != 3) | FALSE
- 536 5. FALSE + 10
- 537 6. TRUE + 10
- 538 7. TRUE + 10 == FALSE + 10
- 539 8. sum(c(TRUE, TRUE, FALSE, FALSE))

4.5 Zugreifen auf Elemente eines Vektors (=Untermengen)

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

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

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

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

```
anzahl_rehe[2]
```

553 ## [1] 79

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

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

555 ## [1] 132 79 129 91 138

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

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

559

560 Aufgabe 7: Zugreifen auf Vektorelemente

562 Erstellen Sie einen neuen Vektor `bhd`

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

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

566

567 Alternativ könnte das gleiche Ergebnis mit einem logischen Vektor erreicht werden. Für eine bessere Über-
568 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)  
anzahl_rehe[sub]
```

569 ## [1] 132 79 129 91 138

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

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

574 ## [1] 132 79 129 91 138

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

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

577 ## [1] 132 79 129 91 138

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

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

580 ## [1] 113.8

581

582 Aufgabe 8: logische Werte

584 Verwenden Sie nochmals den Vektor mit den Anzahl Rehen die an unterschiedlichen Fotofallenstandorten
 585 fotografiert wurden.

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

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

589 4.6 Der %in%-Operator

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

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

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

```

messungen_arten[messungen_arten == "FI"]

594 ## [1] "FI" "FI"
# oder
messungen_arten[messungen_arten == arten[1]]

595 ## [1] "FI" "FI"

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

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

598 ## [1] "FI" "BU" "BU" "FI"

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

messungen_arten %in% arten

602 ## [1] TRUE TRUE TRUE FALSE FALSE FALSE TRUE FALSE FALSE
messungen_arten[messungen_arten %in% arten]

603 ## [1] "FI" "BU" "BU" "FI"

604

605 Aufgabe 9: Auswählen von Elementen in einem Vektor (%in%)
606

```

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

LETTERS

```

608 ## [1] "A" "B" "C" "D" "E" "F" "G" "H" "I" "J" "K" "L" "M" "N" "O" "P" "Q" "R" "S"
609 ## [20] "T" "U" "V" "W" "X" "Y" "Z"

```

610 Wählen Sie aus LETTERS nur die Vokale aus.

5 Faktoren (factors)

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

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

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

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

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

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

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

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

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

```
## [1] Fichte Buche Fichte Eiche Eiche Fichte Fichte
## Levels: Fichte Buche Eiche
```

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

```
levels(af)
## [1] "Fichte" "Buche"  "Eiche"
```

```

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

640 ## [1] Fi Bu Fi Ei Ei Fi Fi
641 ## Levels: Fi Bu Ei

642 Schlussendlich kann man mit der Funktion relevel() die Referenzkategorie eines Faktors (der erste Level)
643 angepasst werden. Das ist kann z. B. für lineare Regressionsmodelle wichtig sein.

af

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

relevel(af, "Bu")

646 ## [1] Fi Bu Fi Ei Ei Fi Fi
647 ## Levels: Bu Fi Ei

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

as.character(af)

650 ## [1] "Fi" "Bu" "Fi" "Ei" "Ei" "Fi" "Fi"
651 Ebenso funktioniert as.factor(). Mit der Funktion as.numeric() erhält man die interne Kodierung von
652 Faktoren.

af

653 ## [1] Fi Bu Fi Ei Ei Fi Fi
654 ## Levels: Fi Bu Ei

as.numeric(af)

655 ## [1] 1 2 1 3 3 1 1
656 Fichte ist das erste Level des Faktors, deshalb erhalten alle Fichteneinträge den Wert 1. Bucheinträge erhalten
657 den Wert 2 und Eichen 3.

658

```

Aufgabe 10: Faktoren

661 Verwenden Sie den Vektor **staedte** und erstellen Sie einen Vektor mit der Anordnung der **levels** in
662 umgekehrter alphabetischer Reihenfolge.

```

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

```

663 5.1 Das Paket `forcats`

664 Sie müssen das Paket `forcats` installieren und laden. Wir kommen später noch auf Pakete zurück.
665 Mit dem Paket aus `forcats` werden einige Operationen mit Faktoren erleichtert. Wir schauen uns hier
666 Funktion an, die es erleichtern:

- 667 1. Die Anordnung von Levels anzupassen.
- 668 2. Levels zusammenzufassen oder zu entfernen.
- 669 3. Labels zu ändern.

670 5.1.1 Anpassen der Anordnung von Faktoren

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

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

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

```
factor(a)
```

```
673 ## [1] FI BU FI EI EI FI FI  
674 ## Levels: BU EI FI
```

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

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

```
676 ## [1] FI BU FI EI EI FI FI  
677 ## Levels: FI BU EI
```

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

```
fct_rev(f1)
```

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

681 nach der Häufigkeit zu sortieren oder

```
fct_infreq(f1)
```

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

684 zufällig zu sortieren.

```
fct_shuffle(f1)
```

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

6 Spezielle Einträge

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

- 689 • fehlenden Einträgen NA,
- 690 • leeren Einträgen NULL,
- 691 • undefinierten Einträgen NaN (Not a Number) oder
- 692 • unendlichen Zahlen (Inf).

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

695 6.1 NA

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

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

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

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

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

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

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

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

```
na2 <- 3
```

713 ## [1] FALSE FALSE NA

1 + NA

714 ## [1] NA

715 Viele R Funktionen haben eingebaute Methoden zum Umgang mit NA. Die Funktion `mean()` bspw. ergibt
716 (wie die meisten Funktionen) standardmäßig NA wenn sie auf Vektoren mit Datenlücken angewendet wird, es
717 sei denn man stellt innerhalb der Funktion ein, dass Datenlücken entfernt werden sollen.

`mean(na2)`

718 ## [1] NA

`mean(na2, na.rm = TRUE)`

719 ## [1] 4.5

720 6.2 NULL

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

725 6.3 Inf

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

`10^309`

728 ## [1] Inf

`2 * Inf`

729 ## [1] Inf

`1 + Inf`

730 ## [1] Inf

`3 / 0`

731 ## [1] Inf

`-3 / 0`

732 ## [1] -Inf

`3 / Inf`

733 ## [1] 0

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

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

736 ## [1] TRUE FALSE FALSE TRUE FALSE
is.finite(inf1)

737 ## [1] FALSE TRUE TRUE FALSE TRUE
inf1 < 3

738 ## [1] FALSE TRUE FALSE TRUE FALSE
```

739

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

742 Verwenden Sie den Vektor

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

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

745 Verwenden Sie den Vektor

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

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

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

751 Berechnen Sie den arithmetischen Mittelwert von `foo`.

7 data.frames oder Tabellen

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

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

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

```
768 ##          ID anzahl_rehe   revier
769 ## 1    Kamera_1        132 Revier A
770 ## 2    Kamera_2         79 Revier A
771 ## 3    Kamera_3        129 Revier A
772 ## 4    Kamera_4         91 Revier A
773 ## 5    Kamera_5        138 Revier A
774 ## 6    Kamera_6        144 Revier B
775 ## 7    Kamera_7         55 Revier B
776 ## 8    Kamera_8        103 Revier B
777 ## 9    Kamera_9        139 Revier B
778 ## 10  Kamera_10       105 Revier B
779 ## 11  Kamera_11       96 Revier C
780 ## 12  Kamera_12       146 Revier C
781 ## 13  Kamera_13       95 Revier C
782 ## 14  Kamera_14      118 Revier C
783 ## 15  Kamera_15      107 Revier C
```

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

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

790 **7.1 Wichtige Funktionen zum Arbeiten mit `data.frames`**

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

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

```
793 ##           ID anzahl_rehe   revier
794 ## 1 Kamera_1          132 Revier A
795 ## 2 Kamera_2          79 Revier A
```

796 Oder für die letzten 2 Beobachtungen.

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

```
797 ##           ID anzahl_rehe   revier
798 ## 14 Kamera_14         118 Revier C
799 ## 15 Kamera_15         107 Revier C
```

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

```
nrow(monitoring)
```

```
801 ## [1] 15
```

```
ncol(monitoring)
```

```
802 ## [1] 3
```

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

```
str(monitoring)
```

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

```
809
```

810 **Aufgabe 12: `data.frame`**

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

815 7.2 Zugreifen auf Elemente eines `data.frame`

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

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

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

```
824 ## [1] 91
```

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

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

```
828 ## [1] 91
```

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

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

```
831 ## [1] 132 79 129 91 138
```

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

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

```
834 ##   anzahl_rehe  revier
```

```
835 ## 1       132 Revier A
```

```
836 ## 2       79 Revier A
```

```
837 ## 3       129 Revier A
```

```
838 ## 4       91 Revier A
```

```
839 ## 5       138 Revier A
```

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

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

```
842 ##      ID anzahl_rehe  revier
```

```
843 ## 1 Kamera_1       132 Revier A
```

```
844 ## 2 Kamera_2       79 Revier A
```

```
845 ## 3 Kamera_3       129 Revier A
```

```
846 ## 4 Kamera_4          91 Revier A
847 ## 5 Kamera_5          138 Revier A
```

848

849 Aufgabe 13: Abfragen von Werten

851 Wir nehmen folgende Werte aus Übung 12 an:

```
umfrage1 <- data.frame(
  fach = c("Forst", "Bio", "Chemie", "Physik", "Forst"),
  semester = c(2, 3, 2, 1, 5),
  alter = c(21, 22, 21, 20, 23)
)
```

- 852 • Wählen Sie nur die ersten drei Zeilen aus und die erste und zweite Spalte aus.
 853 • Wählen Sie alle Zeilen und die erste und dritte Spalte aus.
 854 • Wählen Sie alle Spalten und die erste, dritte und vierte Zeile aus.

855

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

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

```
monitoring$anzahl_rehe
```

862 ## [1] 132 79 129 91 138 144 55 103 139 105 96 146 95 118 107

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

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

864 ## [1] 132 79 129 91 138 144 55 103 139 105 96 146 95 118 107

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

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

866 ## [1] 132 79 129 91 138 144 55 103 139 105 96 146 95 118 107

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

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

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

```
monitoring$anzahl_rehe > 100
```

```
874 ## [1] TRUE FALSE TRUE FALSE TRUE TRUE FALSE TRUE TRUE TRUE FALSE TRUE  
875 ## [13] FALSE TRUE TRUE
```

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

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

```
880 ##          ID anzahl_rehe    revier  
881 ## 1   Kamera_1           132 Revier A  
882 ## 3   Kamera_3           129 Revier A  
883 ## 5   Kamera_5           138 Revier A  
884 ## 6   Kamera_6           144 Revier B  
885 ## 8   Kamera_8           103 Revier B  
886 ## 9   Kamera_9           139 Revier B  
887 ## 10  Kamera_10          105 Revier B  
888 ## 12  Kamera_12          146 Revier C  
889 ## 14  Kamera_14          118 Revier C  
890 ## 15  Kamera_15          107 Revier C
```

891

892 Aufgabe 14: Abfragen von Werten 2

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

- ```
895 • Alle Spalten für Studierende die Forstwissenschaften studieren.
896 • Alle Spalten für Studierende die Chemie oder Physik studieren.
897 • 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.

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

934

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

---

- 936 937 Lesen Sie die Datei `kompliziert.txt` ein. Schauen Sie die Hilfeseite an und vergewissern Sie sich, dass Sie  
wissen was die Argumente `header`, `sep`, `dec` und `skip` bewirken. Setzten die Argumente richtig, damit die  
Datei `kompliziert.txt` folgendes Ergebnis liefert.

## 9 Erstellen von Abbildungen

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

### 945 9.1 Base Plot

946 Die wichtigsten Grafiken für die einfache Datendarstellung sind schnell verfügbar. Etwas komplexere oder  
947 spezielle Grafiken erfordern mehr Programmieraufwand (folgt teilweise noch). Für viele komplexere Diagramme  
948 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  
949 sie sich die einfache Grafik Schnittstelle (`base plots`) als zweidimensionale Leinwand vor, auf die Sie durch  
950 Code Ebene für Ebene Grafikelemente legen:  
951

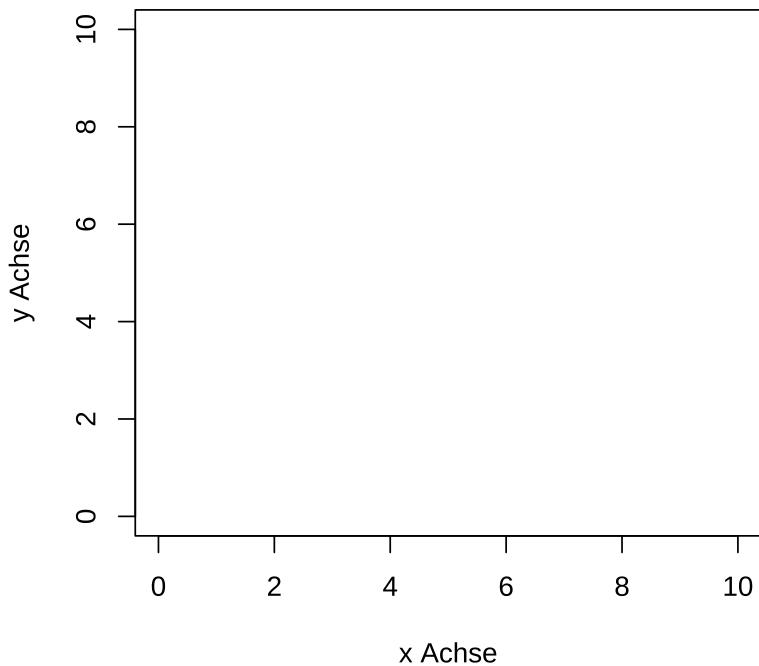
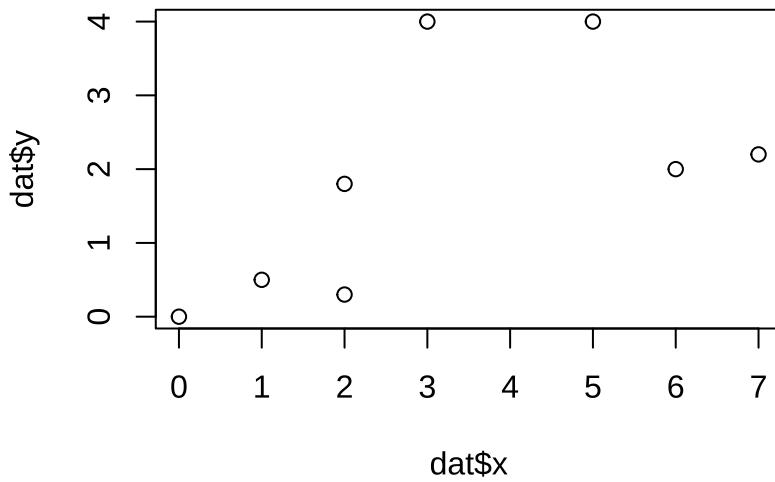


Abbildung 7: Beispiel einer leeren Grafikschnittstelle.

952 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)
)

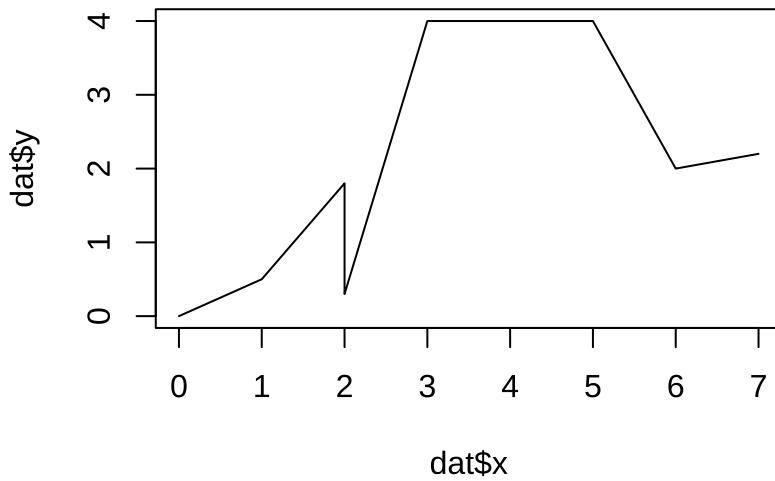
plot(datx, daty, type = "p")
```



953

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

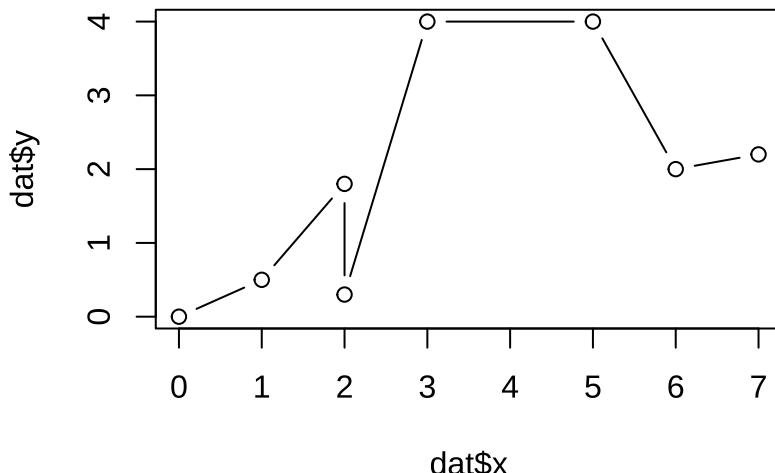
```
plot(datx, daty, type = "l")
```



956

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

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



958

959 darstellen.

960

961 **Aufgabe 16: Base Plot 1**

---

962963 Laden Sie den Datensatz `bhd_1.txt` und erstellen Sie eine Abbildung mit dem Alter jedes Baumes auf der  
964 x-Achse und dem BHD auf der y-Achse.

965

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

- 970 • `type` - Diagrammtyp
- 971 • `col` - Farbe
- 972 • `main` - Titel
- 973 • `sub` - Untertitel
- 974 • `pch` - Punktsymbol
- 975 • `lty` - Linientyp
- 976 • `lwd` - Linienstärke
- 977 • `xlab` bzw. `ylab` - Achsenbeschriftungen
- 978 • `xlim`, `ylim` - Grenzen der Achsenanschnitte
- 979 • `axes` - Sollen die Achsen eingezeichnet werden? Oder leer gelassen werden, um sie nachträglich als  
980 low-level Ebene einzuziehen?
- 981 • `ann` - Achsenbeschriftung kann ganz weggelassen werden.

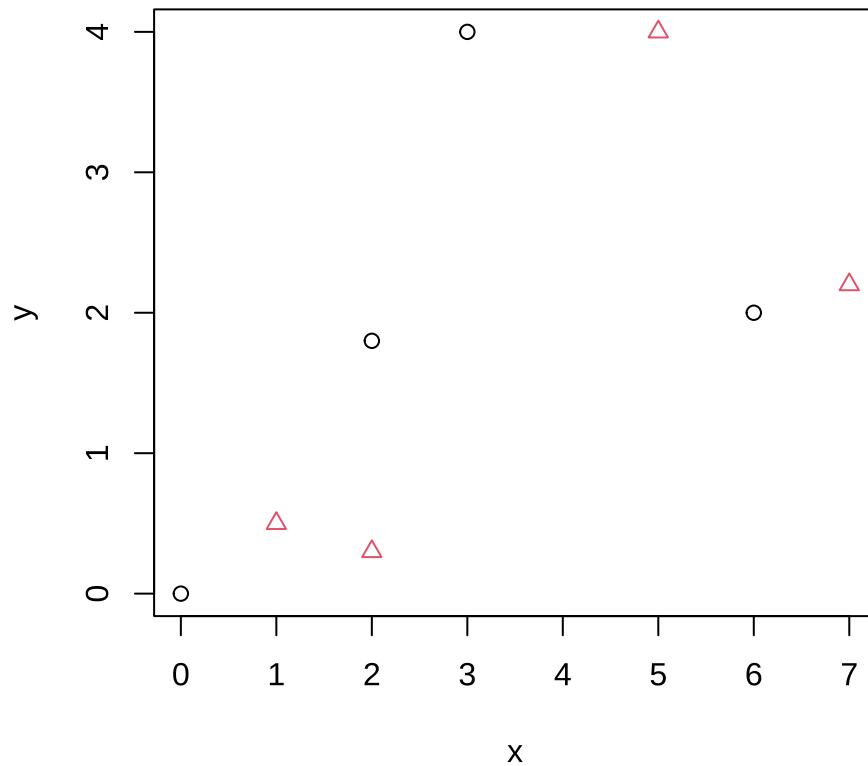
982 Sehen Sie sich die Hilfeseiten `?plot.default()` oder `?par()` an für weitere Informationen. Dort finden Sie  
983 auch eine vollständige Liste der Befehle. Einige Argumente können als Vektor übergeben werden. Hier z. B.  
984 die Farben und die Punktsymbole.

```

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")

```



985

986

---

**Aufgabe 17: Anpassen von Plots**


---

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

- 990     • Beschriften Sie die x- und y-Achse sinnvoll.  
 991     • Fügen Sie eine Überschrift hinzu.  
 992     • Wählen Sie ein anderes Symbol.  
 993     • Stellen Sie die Symbole in rot dar.

994

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

996 Die wichtigsten Funktionen sind

- 997     • `points()` - Fügt Punkte ein  
 998     • `lines()` - Fügt Linien ein

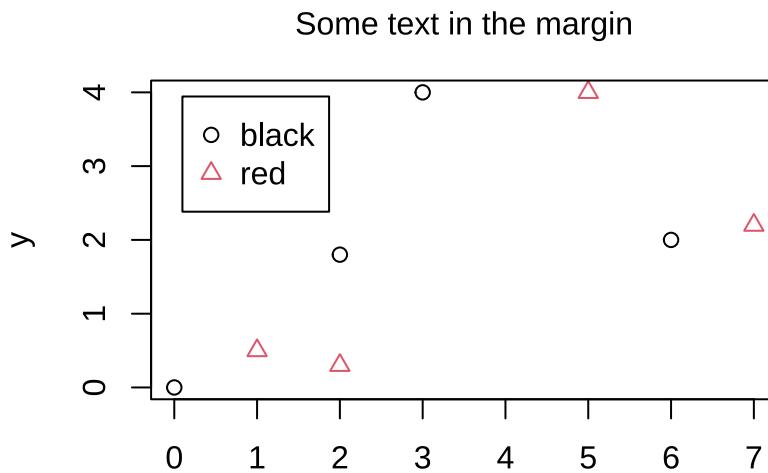
- `text()` - Fügt Text ein
- `mtext` - Fügt Text in den Rahmen (`margin`) ein
- `legend()` - Fügt eine Legende ein
- `abline()` - Fügt eine Gerade ein
- `curve()` - Fügt eine mathematische Funktion ein
- `arrows()` - Fügt Pfeile ein
- `grid()` - Fügt Hilfslinien ein

1006 Dabei ist der Aufbau zunächst grundsätzlich wie in Abbildung 8 dargestellt. Der Vorteil von Low-Level  
 1007 Funktionen ist, dass die einzelnen Level mehr Funktionen bieten als die High-Level Funktion und, dass Sie  
 1008 sich die Reihenfolge der Ebenen definieren können.

1009 Legenden können im base plot nur als Low-Level Funktion hinzugefügt werden. Mit der Funktion `legend`  
 1010 werden die Einträge und die entsprechenden Punktsymbole oder Linientypen dargestellt. In der folgenden  
 1011 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")
```



1012 Mit diesem grundsätzlichen Aufbau sollten Sie bereits in der Lage sein auch komplexe Grafiken schnell zu  
 1013 gestalten. Wenn Sie mehrere Diagramme in einem Plot arrangieren möchten, können Sie mit dem `par()`  
 1014 Befehl ein Arrangement definieren. Sie haben dann zusätzlich zu den bereits bekannten Grafikregionen noch  
 1015 äußere Ränder (`outer margins`). Siehe Abbildung 9.

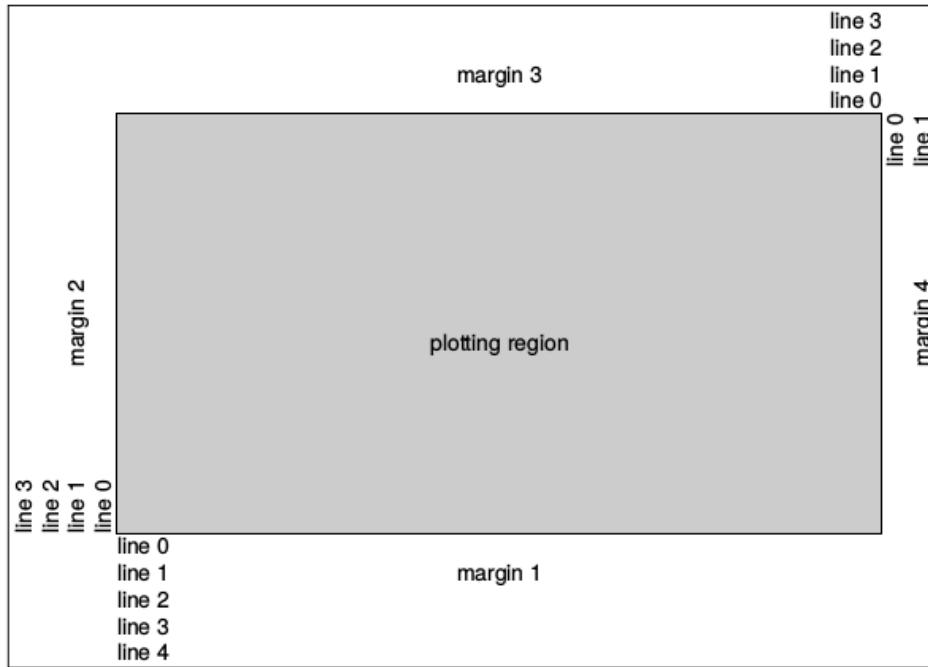


Abbildung 8: Grafikregionen eines base plots in R.

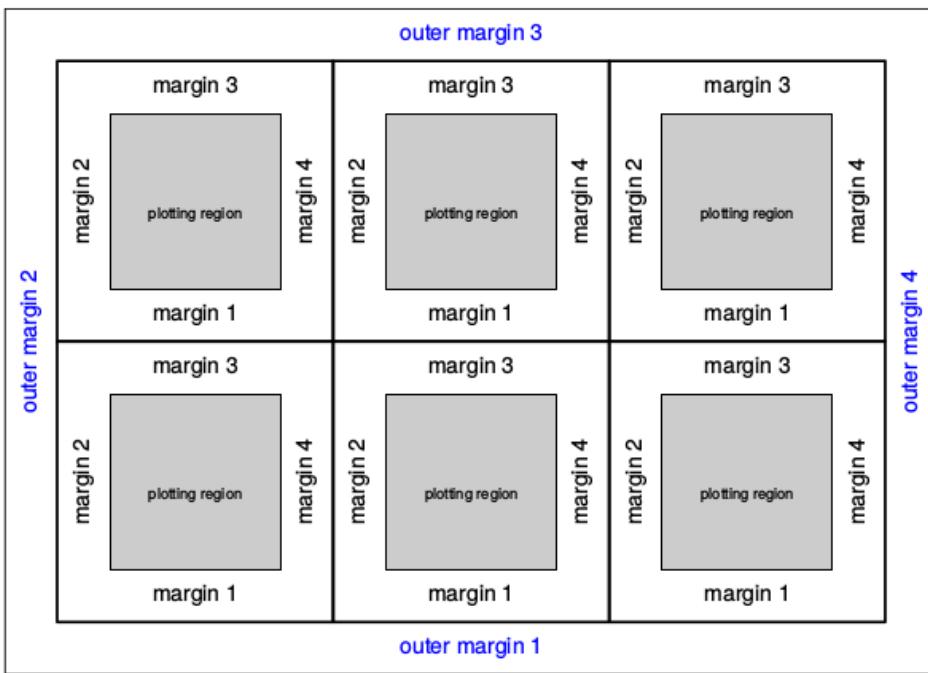


Abbildung 9: Schematischer Aufbau mehrerer Diagramme in einem plot am Beispiel einer 3 x 2 Grafik.

1017 **9.1.1 Mehrere Panels**

1018 Mit der Funktion `par()` kann auch eingestellt werden, dass ein Plot aus mehreren Subplots (= Panels)  
 1019 besteht. Die Argumente `mfrow` und `mfcol` können `par()` übergeben werden und kontrollieren die Anzahl  
 1020 Zeilen und Spalten für den Plot.

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

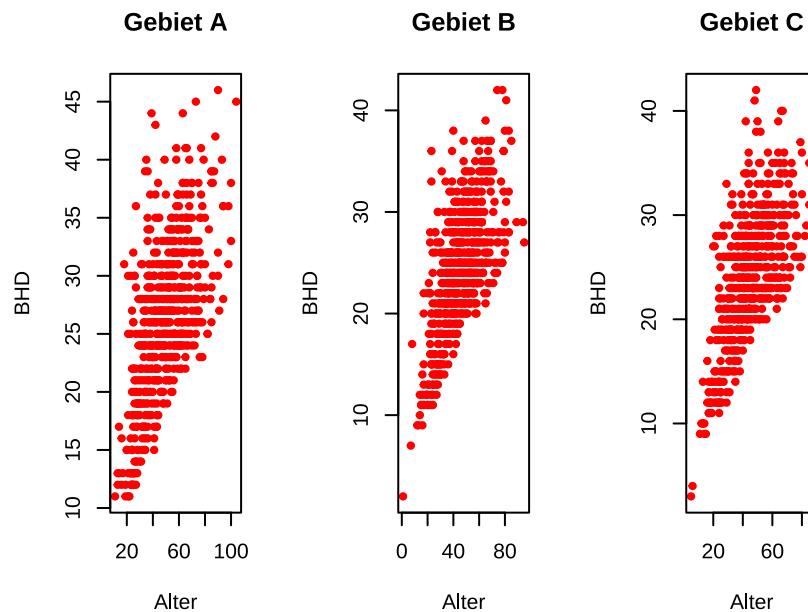
1021 Teilt den Plot in eine Zeile und drei Zeilen (= drei Plots nebeneinander).

```
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")
```



1022  
 1023 Vergessen Sie nicht am Ende nochmals `par(mfrow = c(1, 1))` zu setzen, damit wieder nur ein Plot angezeigt  
 1024 wird.

1025 **9.1.2 Speichern von Abbildungen**

1026 Wenn nicht anders angegeben, wird die Abbildung zunächst nur in der RStudio Grafik Schnittstelle abgebildet  
 1027 (Pane 4). Von dort aus kann die Abbildung exportiert werden. Es bietet sich jedoch an das Speichern der  
 1028 Abbildung direkt im Code zu programmieren. Mögliche Formate die Abbildung als Vektorgrafik zu speichern  
 1029 sind

- 1030 • `pdf()` oder
- 1031 • `postscript()`.

1032 Beispiele für Rastergrafiken sind

- 1033 • `png()`,
- 1034 • `bmp()` oder
- 1035 • `jpeg()`.

1036 Die Grafikschnittstelle ist dann Ihre “Leinwand”. Mit dem Befehl `dev.off()` trennen Sie die Verbindung zur  
1037 Schnittstelle wieder. Ihre “Leinwand” wird also wieder geschlossen. So lange die Schnittstelle geöffnet ist  
1038 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
```

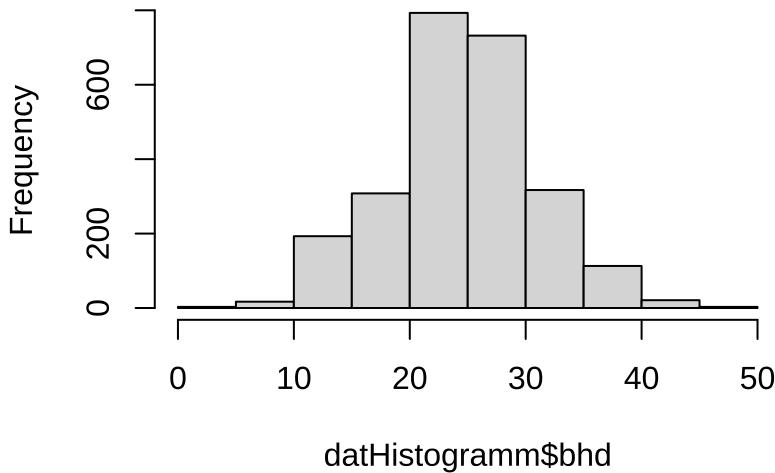
1039 *Achtung*, wenn Sie die Funktion `dev.off()` nicht aufrufen, werden alle nachfolgenden Plots in die gleiche  
1040 Datei geschrieben. Falls Sie nach einem Versuch einen Plot zu speichern plötzlich keine weiteren Plots mehr  
1041 sehen, führen Sie einige Mal die Funktion `dev.off()` aus.

## 1042 9.2 Histogramme

1043 Neben den Streudiagrammen (*Scatterplots*, oder auch einfach x-y Diagramm) sind *Histogramme* in der  
1044 angewandten Datenanalyse ein weiterer wichtiger Abbildungstyp. An Histogrammen wird die Häufigkeit  
1045 von Beobachtungen nach Gruppen dargestellt. Sie sind deshalb so wichtig, weil man aus ihnen relevante  
1046 Informationen über die Verteilung der Daten ablesen kann und somit einiges über die Messungen erfährt,  
1047 das für die weitere Datenanalyse relevant sein kann. So werden auf einen Blick der Zusammenhang von  
1048 Beobachtungshäufigkeit und Streuung deutlich, sowie auch die Form der Verteilung und ihre Schiefe. Die  
1049 Interpretation werden wir bei den Boxplots noch weiter vertiefen.

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

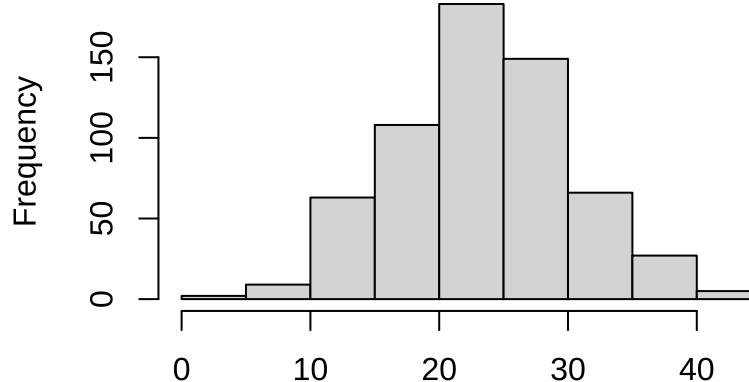
### Histogramm of datHistogramm\$bhd



1050

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

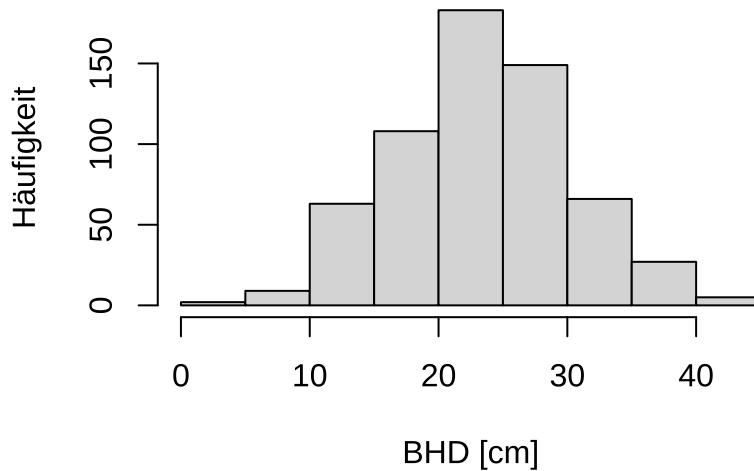
### Histogramm of datHistogramm\$bhd[datHistogramm\$art == "EI"]



1051

```
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

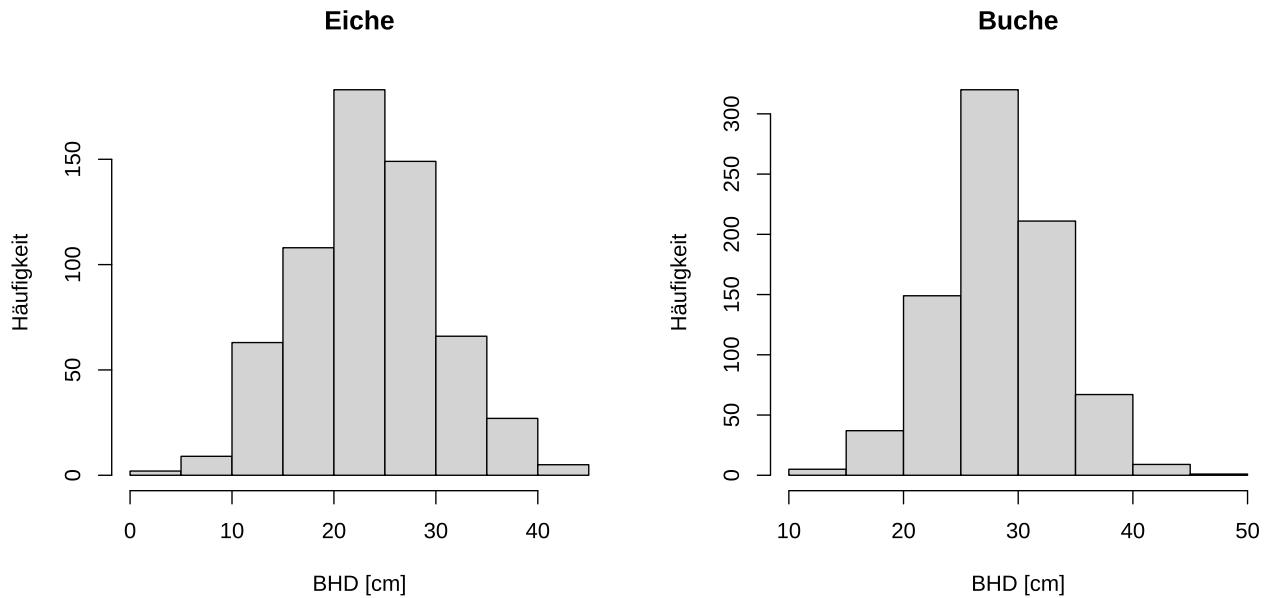


1052

1053 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")
```

1054



1055

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

### 1056 9.3 Boxplots

1057 Oft möchte man die Verteilung einer stetigen Variablen in Abhängigkeit einer diskreten Variable beschreiben  
 1058 oder visualisieren. Ein Beispiel dafür wäre die BHD-Verteilung in Abhängigkeit der Baumarten. Eine häufige  
 1059 Darstellungsform für solche Daten sind *Boxplots*. Sie stellen die durchschnittliche Ausprägung einer stetigen  
 1060 Variable und ihre Schwankung kompakt dar.

1061 Boxplots bestehen aus drei Komponenten:

- 1062 1. Eine *Box*, die den Bereich zwischen 0.25 und 0.75 Percentil abdeckt, diese Distanz wird auch die *IQR*  
 1063 (Interquartile Range), bezeichnet. Zusätzlich wird die Box durch den Median (als dicke horizontale Linie)  
 1064 unterteilt.
- 1065 2. Einzelne Punkte Ausreißer. Als Ausreißer werden Punkte bezeichnet, die  $> 1.5IQR$  vom unteren oder  
 1066 oberen Ende der Box entfernt sind.
- 1067 3. Eine senkrechte Linie von jeder Seite der Box bis zum letzten "Nicht-Ausreißer-Punkt". Also der letzte  
 1068 Punkt, der  $> 1.5IQR$  aber nicht  $> 0.75$  bzw.  $< 0.25$  Percentil ist. Diese Linie wird auch als *Whisker*  
 1069 bezeichnet.

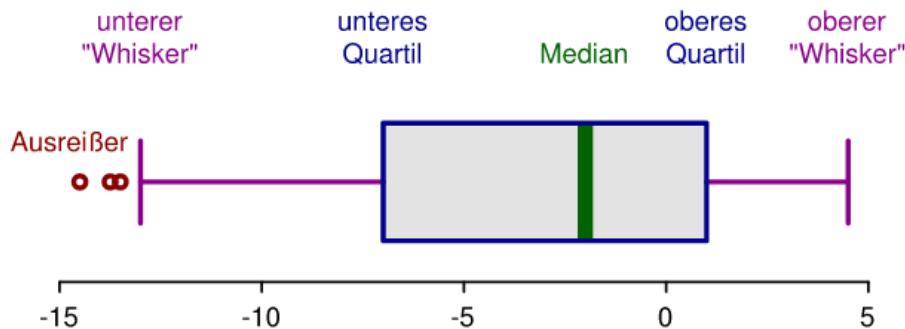
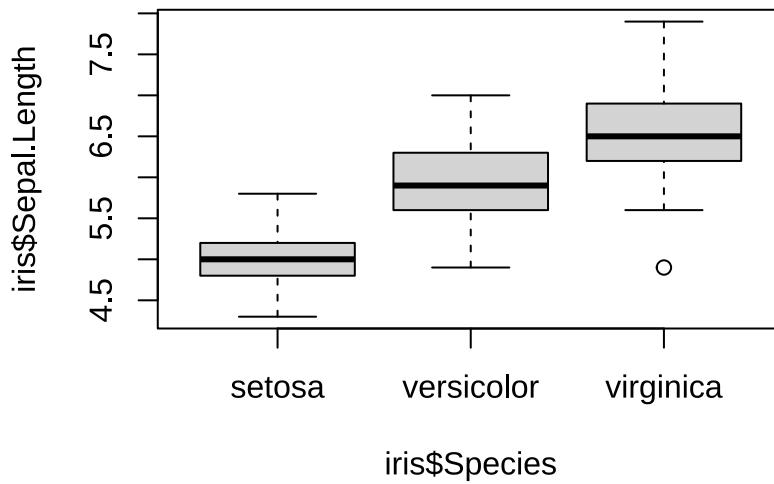


Abbildung 10: Schematische Darstellung eines Boxplots (Quelle: Von RobSeb - Eigenes Werk, CC BY-SA 3.0, <https://commons.wikimedia.org/w/index.php?curid=14697172>).

1070 Mit R kann mit der Funktion `boxplot()` ein Boxplot erstellt werden. Diese Funktion kann in zwei unter-  
 1071 schiedlichen Ausprägungen verwendet werden.

- 1072 1. `boxplot(x)` erzeugt einen Boxplot für die Variable `x`.
- 1073 2. `boxplot(x ~ y)` erzeugt einen oder mehrere Boxplots für `x` aber gruppiert nach `y`, dabei sollte `y` eine  
 1074 kategoriale Variable sein. `x` und `y` können auch die Spaltennamen eines `data.frames` sein, dann muss  
 1075 das `data.frame` mit dem Argument `data` zusätzlich übergeben werden.

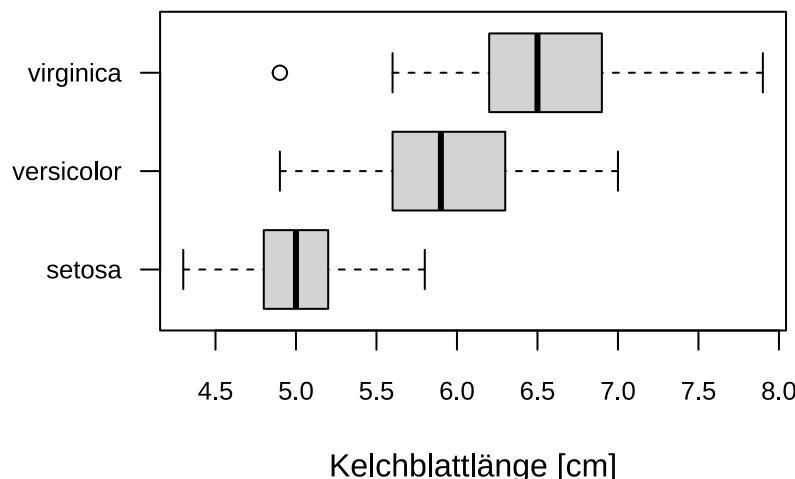
```
boxplot(iris$Sepal.Length ~ iris$Species)
```



1076

1077 Etwas eleganter ist es wenn wir das Argument `data` verwenden und den Plot etwas anpassen. Diese Schreib-  
1078 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
)
```



1079

1080

### 1081 Aufgabe 18: Boxplots

- 1083 • Lesen Sie erneut den Datensatz `daten/bhd_1.txt` ein und speichern Sie diesen in die Variable (`dat_bhd`).  
 1084 • Wie viele BHD-Messungen gibt es für jedes Gebiet?  
 1085 • Erstellen Sie für jedes Gebiet einen Plot
- 1086 Erstellen Sie Boxplots für jedes Gebiet und innerhalb der Gebiete für jede Art.

## 9.4 ggplot2: Eine Alternative für Abbildungen

1087 ggplot2 ist ein alternatives Plotting-System in R. Sie können mit ggplot2 also grundsätzlich Abbildungen  
 1088 mit dem selben Inhalt erstellen, wie mit Base Plots. Die Syntax und die optische Darstellung unterscheiden  
 1089 sich jedoch grundsätzlich. ggplot2 basiert auf den *grammar of graphics* von Leland Wilkinson. Die Idee  
 1090 ist, alle nötigen Informationen der Abbildung miteinander zu verknüpfen. ggplot2 ist also diametral zu  
 1091 Base Plots. Während Sie in base plot alle Elemente selbst definieren, ist die Idee von ggplot2, dass Sie  
 1092 nur die Daten und den Diagrammtypen übergeben und die Funktion selbst eine schöne Abbildung erstellt.  
 1093 Selbstverständlich können Sie aber auch in ggplot2 viele Einstellungen vornehmen. Im base plot sehen  
 1094 Abbildungen zunächst sehr karg und etwas unfertig aus. Sie müssen viele Einstellungen vornehmen, um eine  
 1095 publizierfähige Grafik zu produzieren. In ggplot2 sollen auch die einfachste Abbildungen schon ästhetisch  
 1096 sein. Mit diesen gebündelten Informationen kann ggplot2 die Abbildung automatisch verschönern. So  
 1097 werden bspw. die Legenden automatisch erzeugt und auch die Formatierungen automatisch an die Datenlage  
 1098 angepasst. ggplot2 nimmt der\*dem Entwickler\*in also Arbeit ab. Dadurch sind die Abbildungen schon ohne  
 1099 viel Nacharbeit schick. Nachteil ist, dass der\*dem Entwickler\*in weniger Möglichkeiten zur Einstellung zur  
 1100 Verfügung stehen und nuterspezifische Sonderwünsche somit schwerer umsetzbar sind. Sehen Sie sich das  
 1101 Cheatsheet zu ggplot2 an. Es ist in RStudio unter [Help > Cheatsheets](#) zu finden.

1102 Bei ggplot2 sind Anweisungen zu den Daten und Anweisungen zur Darstellung voneinander getrennt. Die  
 1103 Daten werden in den Ästhetikbefehl übergeben und dort klassifiziert. Dann folgen die Darstellungsanweisungen.  
 1104 Ähnlich wie bei Base Plots, werden die Grafikelemente ebenenweise nacheinander programmiert, jedoch mit  
 1105 einem + verbunden. Und hier liegt der wesentliche Unterschied zu Base Plots. Durch die + werden die Ebenen  
 1106 zu einem Befehl verbunden und damit gleichzeitig erstellt.

1107 Die Erweiterung wird zunächst geladen<sup>7</sup>. Wir laden außerdem den Datensatz **iris**. Der Datensatz ist in R  
 1108 fest integriert. Siehe `?iris` für mehr Informationen.

```
library(ggplot2)
head(iris)

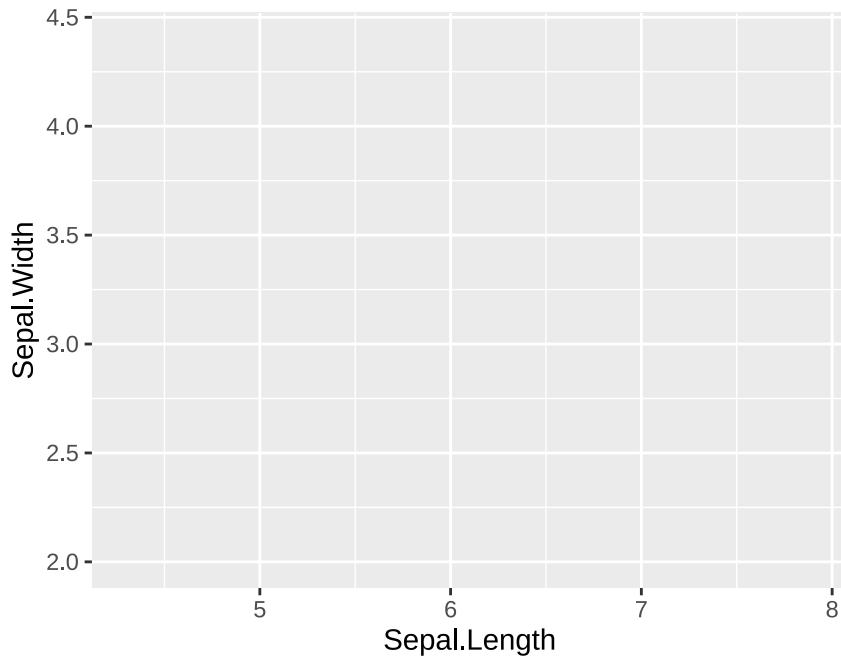
1110 ## Sepal.Length Sepal.Width Petal.Length Petal.Width Species
1111 ## 1 5.1 3.5 1.4 0.2 setosa
1112 ## 2 4.9 3.0 1.4 0.2 setosa
1113 ## 3 4.7 3.2 1.3 0.2 setosa
1114 ## 4 4.6 3.1 1.5 0.2 setosa
1115 ## 5 5.0 3.6 1.4 0.2 setosa
1116 ## 6 5.4 3.9 1.7 0.4 setosa
```

1117 Die Ästhetik wird bspw. folgendermaßen definiert.

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

---

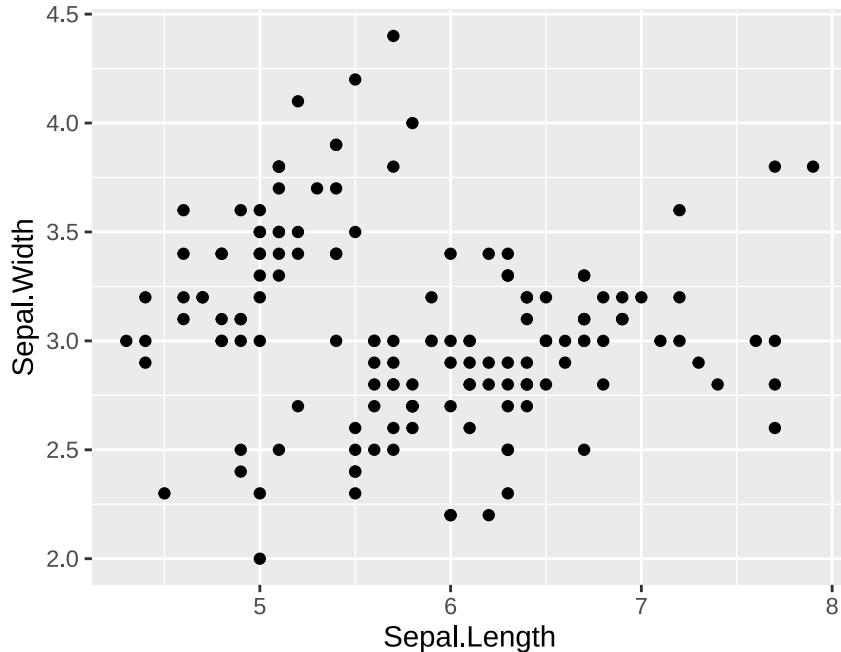
<sup>7</sup>Wir haben bis jetzt immer nur mit base R gearbeitet. D.h. wir haben nur Funktionen verwendet, die R bereits zur Verfügung stellt. Eine der großen Stärken von R sind die Erweiterungen (oder auch Pakete genannt). ggplot2 ist so eine Erweiterung, die einmal mit `install.packages("ggplot2")`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



1118

1119 Dieser Befehl zeichnet noch keine Daten. Die Daten werden lediglich herangezogen, um einen leeren Plot für  
1120 die Daten zu erstellen. In dem Beispiel wird die Variable `Sepal.Length` aus dem `data.frame iris` als x und  
1121 `Sepal.Width` als y Variable definiert. Diese Informationen stehen den folgenden Layern nun zur Verfügung,  
1122 sodass nach den + nur noch x und y verwendet werden müssen. Um bspw. einen Scatterplot zu erstellen  
1123 wird ein `geom_point()` Layer hinzugefügt. x und y werden automatisch an `geom_point` übergeben. Weitere  
1124 Einstellung sind in diesem Beispiel nicht notwendig, wären jedoch möglich. Siehe `?geom_point()`.

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



1125

1126

---

1127 **Aufgabe 19: Abbildungen mit ggplot2**

---

1128

1129 Verwenden Sie die Daten aus Aufgabe 16 und erstellen Sie einen Scatterplot mit ggplot2 wie in Aufgabe 16.

1130

1131 Wir haben mit der Funktion `geom_point()` demm Plot eine Punktgeometrie hinzugefügt. Es gibt noch viele  
1132 weitere Geometrien. Die wichtigsten sind:

- 
- 1133 •
- `geom_line()`
- für eine Linie.
- 
- 1134 •
- `geom_histogram()`
- um ein Histogramm zu erstellen.
- 
- 1135 •
- `geom_boxplot()`
- um einen Boxplot zu erstellen.
- 
- 1136 •
- `geom_bar()`
- um ein Säulendiagramm zu erstellen.

1137 Welche Geometrie die richtige ist, richtet nach dem Typ der darzustellenden Variablen. Beispielsweise  
1138 bietet sich `geom_point()` an, wenn man zwei kontinuierliche Variable darstellen möchte. Wenn man hin-  
1139 gegen die Verteilung von einer kontinuirlchen Variable darstellen möchte, dann bietet sich ein Histogramm  
1140 (`geom_histogram()`) oder auch eine geschätzte Dichte (z.B. `geom_density()`) an.

1141

---

1142 **Aufgabe 20: Abbildungen mit ggplot2**

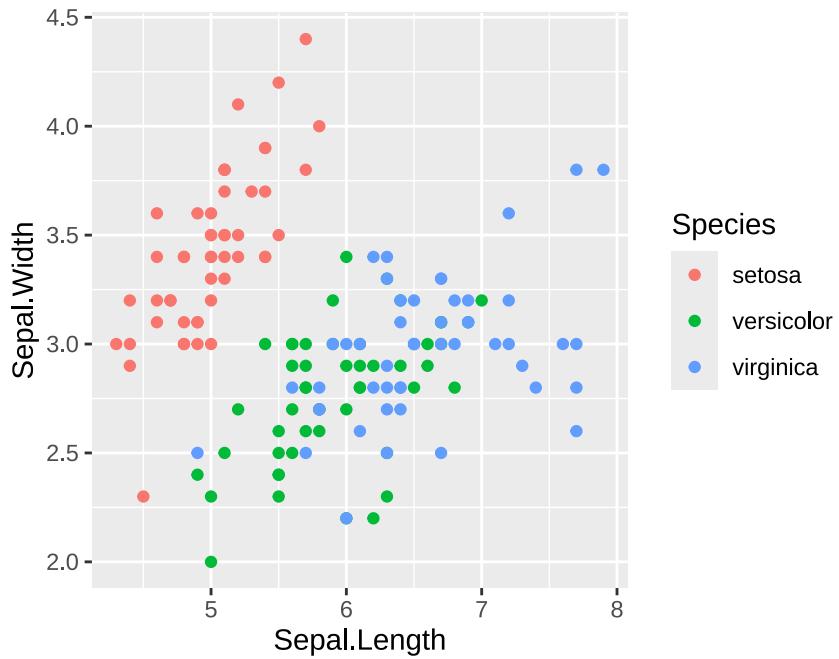
---

11431144 Verwenden Sie die den Iris Datensatz und erstellen Sie mit ggplot2 einen Plot der die Verteilung der Länge  
1145 der Kelchblätter zeigt (Spalte `Sepal.Length`).

1146

1147 Eine der Stärken von ggplot2 ist, dass man den Wert unterschiedlicher Variable auf unterschiedlichen  
1148 Komponenten des Plots abbilden kann. Wir haben bis jetzt ein bzw. zwei Variable auf der x- und y-Achse  
1149 abgebildet. Wir können aber ein weitere Variablen verwenden um das Aussehen des Plots zu beeinflussen.  
1150 Beispielsweise können wir die Farbe der Punkte (für `geom_point()`) mit dem Argument `col` beeinflussen.

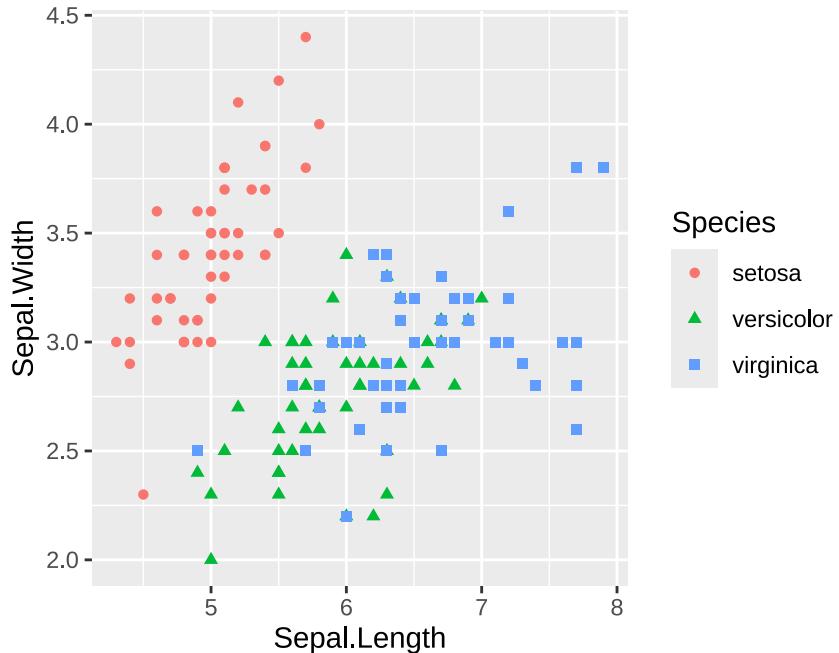
```
ggplot(iris, aes(x = Sepal.Length, y = Sepal.Width, col = Species)) +
 geom_point()
```



1151

- 1152 Somit bekommt jede Irisart eine eigene Farbe<sup>8</sup>. Gleichesmaßen können wir die Punktart (**shape**), die  
1153 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()
```

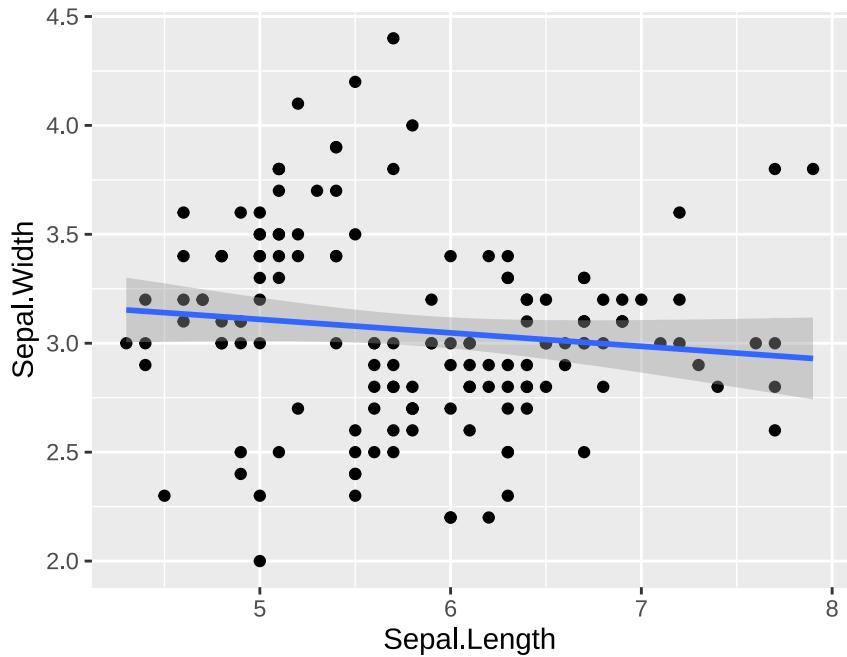


1154

- 1155 In dem Plot ist die Information zu der Art redundant (einmal als Farbe und einmal Symbolart).  
1156 Ein weitere sehr nützliche Geometrie ist **geom\_smooth()**, die es erlaubt eine Trendlinie hinzuzufügen.

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

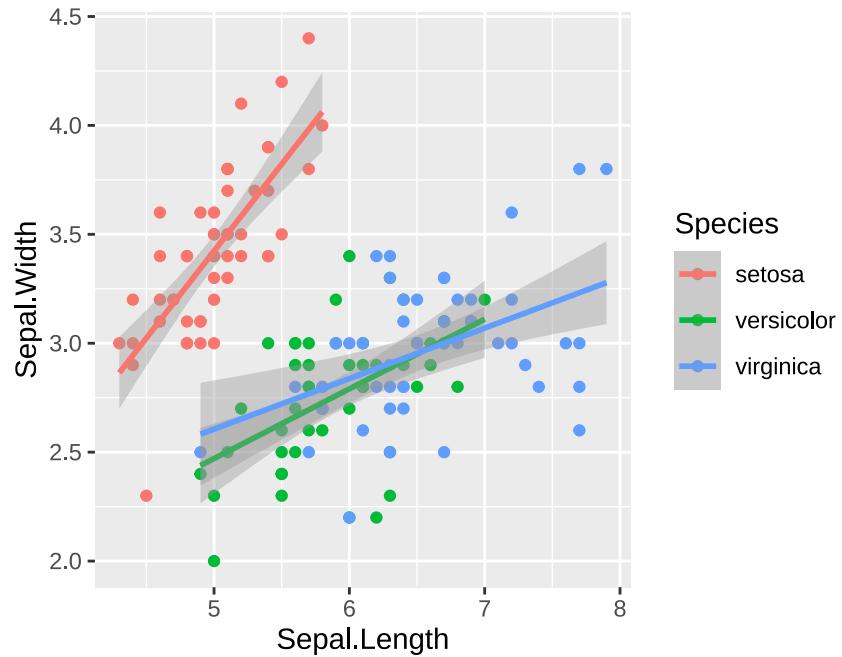
```
ggplot(iris, aes(x = Sepal.Length, y = Sepal.Width)) +
 geom_point() + geom_smooth(method = "lm")
```



1157

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

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



1161

1162

1163 **Aufgabe 21: Anpassen von Plots**  
1164

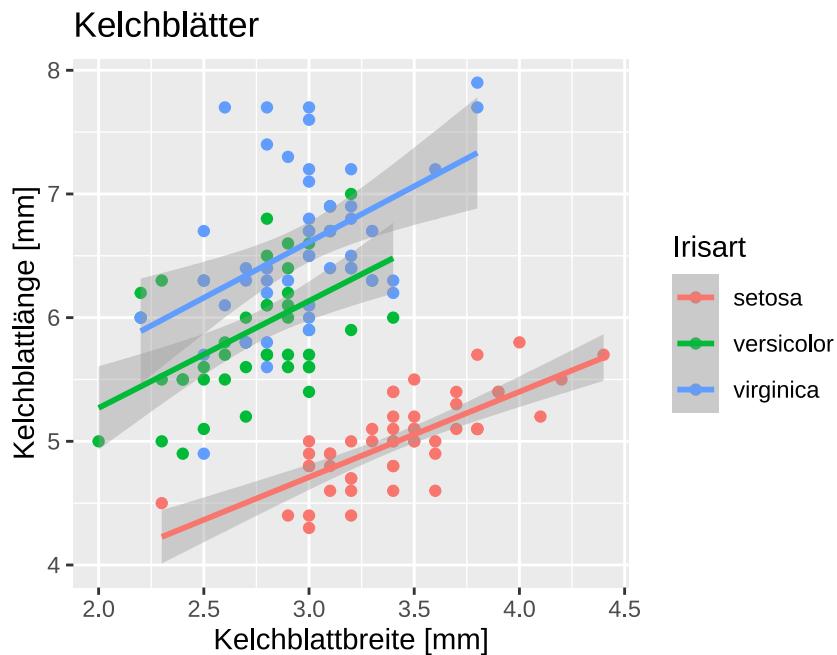
- 1165 Lesen Sie den Datensatz `data/bhd_1.txt` ein und erstellen Sie einen Boxplot für die Verteilung des BHDs  
 1166 für jede Baumart. In einem zweiten Schritt verwenden Sie erst `col = gebiet` und dann `fill = gebiet`.  
 1167 Welchen Unterschied stellen Sie fest?

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

1168

- 1169 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")
```



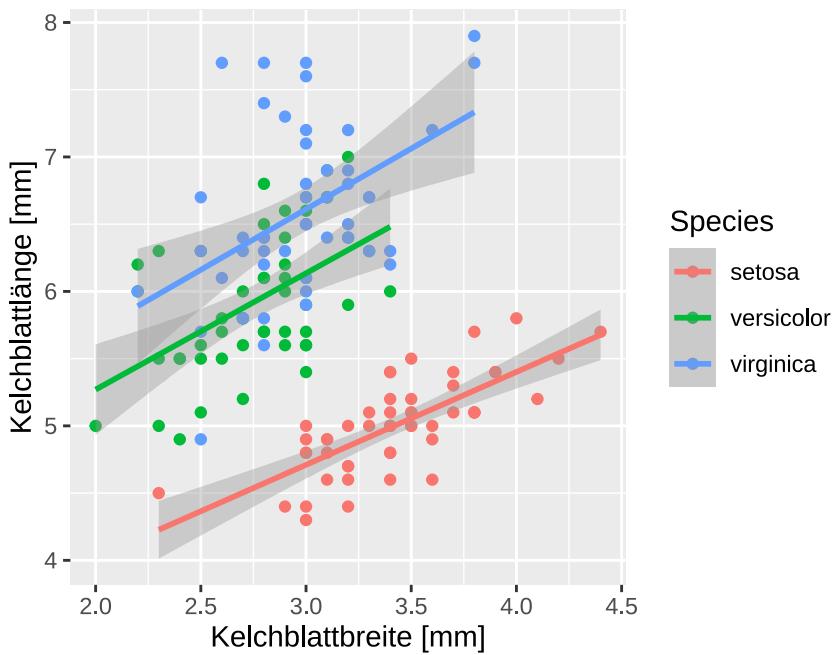
1170

- 1171 Statt einen langen Befehl zu tippen, kann ein `ggplot()` auch zwischengespeichert und wieder aufgerufen bzw.  
 1172 angepasst werden. Das ist vor allem sinnvoll, wenn mehrere Abbildungen auf dem selben Zwischenergebnis  
 1173 aufbauen sollen.

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

- 1174 Wir können jetzt mit `p1` weiter arbeiten und beispielsweise eine Beschriftung hinzufügen.

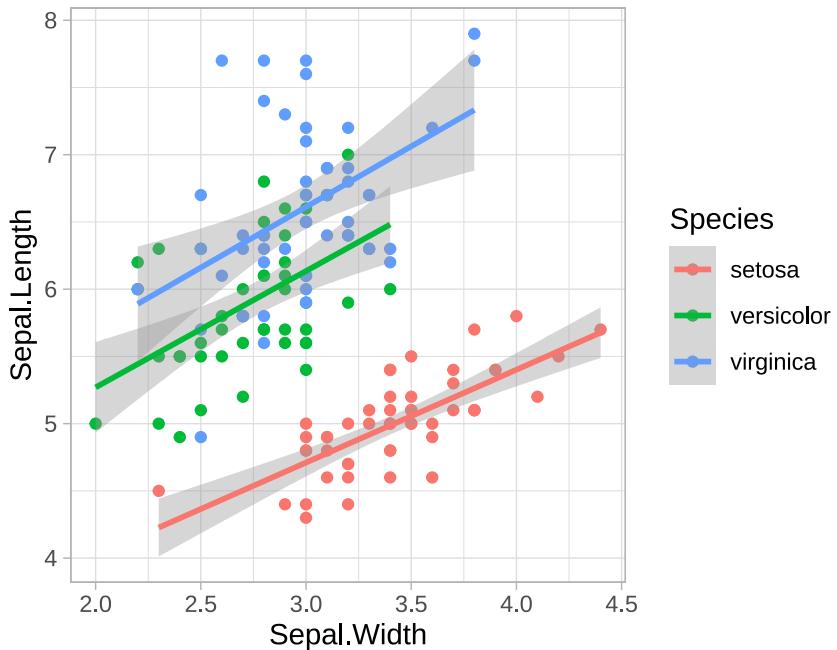
```
p1 + labs(x = "Kelchblattbreite [mm]", y = "Kelchblattlänge [mm]")
```



1175

1176 Oder auch den ganzen Plot anpassen. Dafür gibt es *themes*. Es gibt eine Reihe von vorgefertigten *themes*  
1177 oder man kann diese auch selber erstellen (das ist aber nicht Teil dieses Kurses).

```
p1 + theme_light()
```

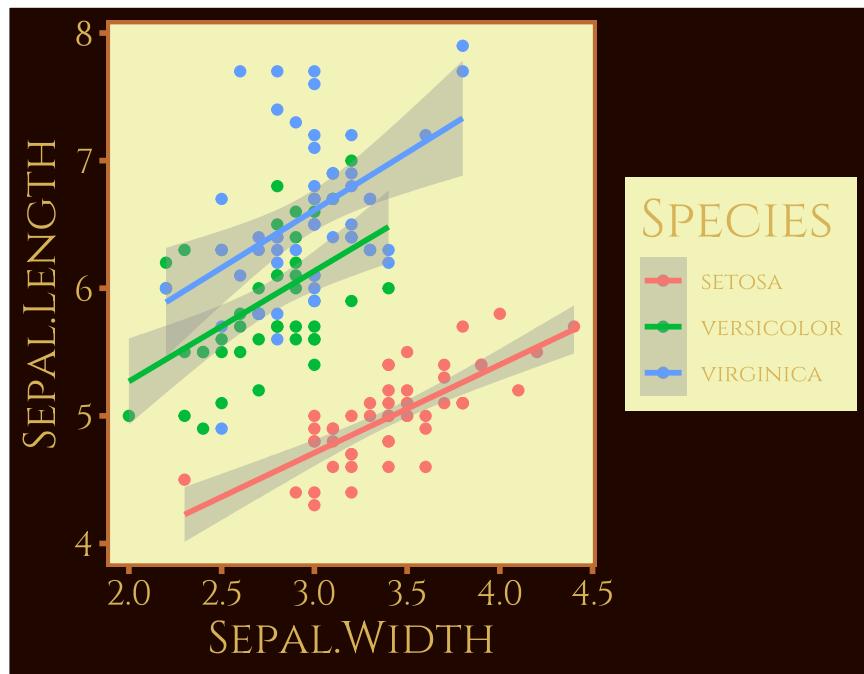


1178

1179 Weitere *themes* sind: `theme_bw()`, `theme_linedraw()` oder `theme_dark()`. Es gibt extra Pakete die viele  
1180 zusätzliche weitere *themes* anbieten. Dazu gehört z.B. das Paket `ggthemes` oder `ThemePark`. Während  
1181 `ggthemes` hauptsächlich durchdachte Grafikkonzepte liefert, sind die Themes aus `ThemePark` eher Popkultur  
1182 und nicht 100 %ig ernst gemeint. `ThemePark` muss zunächst aus GitHub installiert werden. Die Installation

1183 wird auf der GitHub erläutert.

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

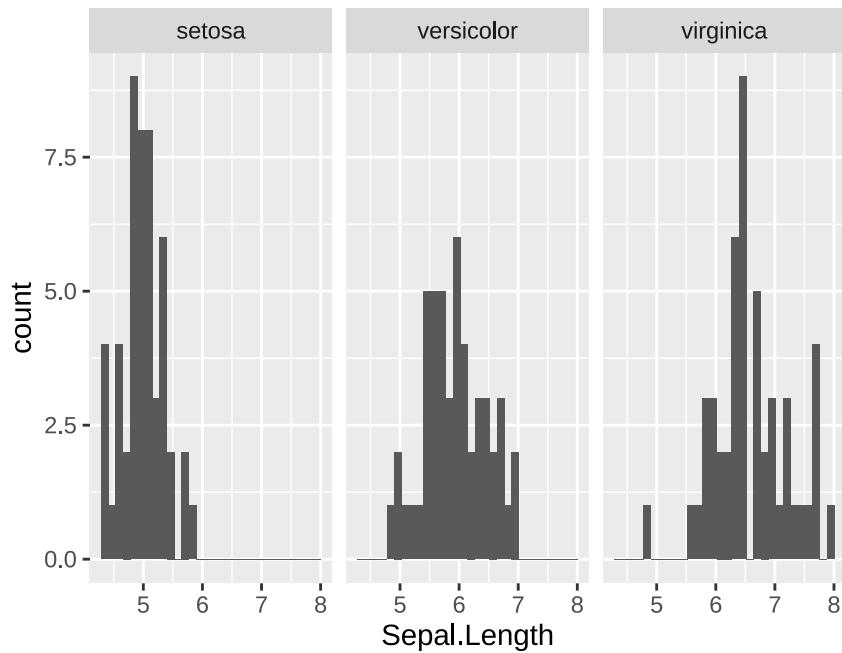


1184

#### 1185 9.4.1 Multipanel Abbildungen

1186 Mit `ggplot2` kann man einfach Abbildungen erstellen, die mehrere Panels haben. Das bedeutet, dass eine  
1187 oder mehrere weitere Variablen gibt, die einen Plot in mehrere Subplots teilt. Dafür gibt es zwei Funktionen:  
1188 `facet_grid()` und `facet_wrap()`.

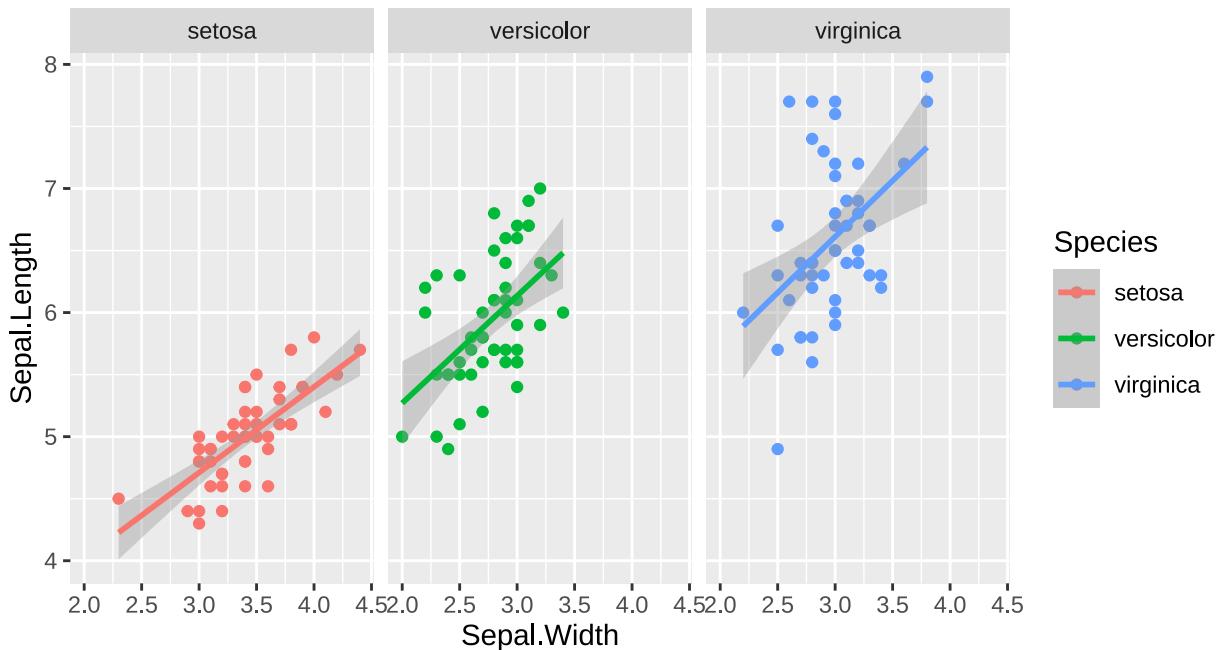
```
ggplot(iris, aes(Sepal.Length)) + geom_histogram() +
 facet_grid(~ Species)
```



1189

1190 Die Funktion `facet_grid()` erzeugt einen *Grid* und beschriftet die Grid-Zeilen und -Spalten, während  
 1191 `facet_wrap()` für jedes Panel (Diagramm) eine eigene Überschrift erzeugt. Beim Arrangieren der Diagramme  
 1192 wird der Vorteil von `ggplot2` gegenüber `base plot` besonders deutlich. Während es im `base plot` System  
 1193 mühsam ist, die Abbildungen zu arrangieren, übernimmt `ggplot2` das Arrangement automatisch und fügt  
 1194 sogar eine automatisch Legende hinzu. Die Achsenabschnitte werden so angepasst, dass die Daten vergleichbar  
 1195 sind.

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

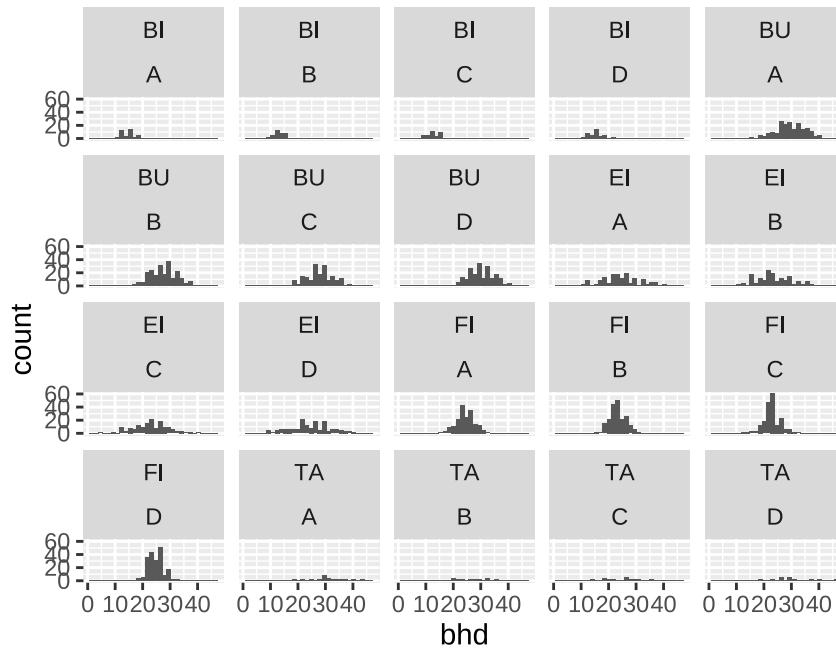


1196

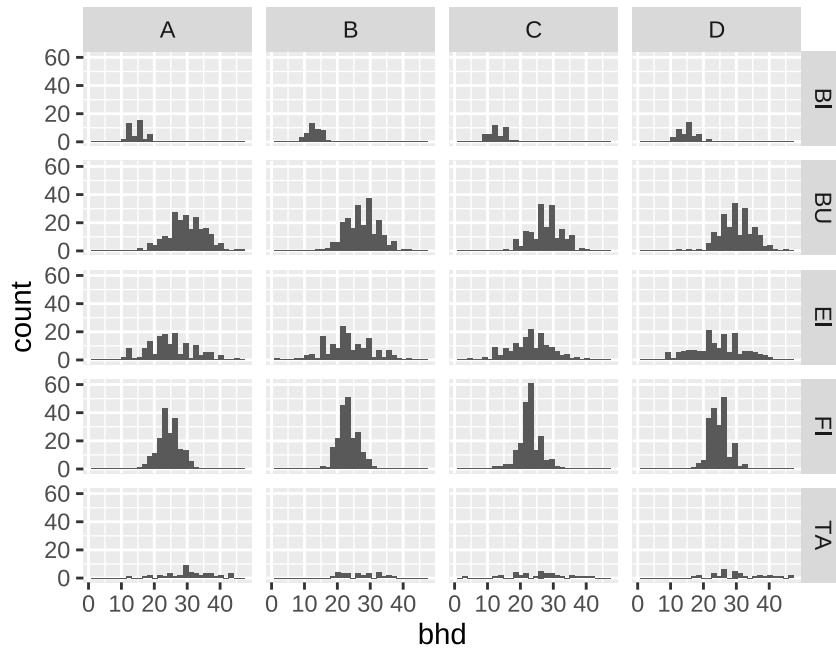
1197

1198 **Aufgabe 22: Multipanel Abbildungen**  
1199

- 1200 Lesen Sie erneut den Datensatz `daten/bhd_1.txt` ein und speichern Sie diesen in die Variable (`dat_bhd`).  
 1201 Erstellen Sie für jede Art und Gebiet ein Histogramm. Welche Unterschiede können Sie feststellen, wenn Sie  
 1202 `facet_grid()` oder `facet_wrap()` verwenden?



1203



1204

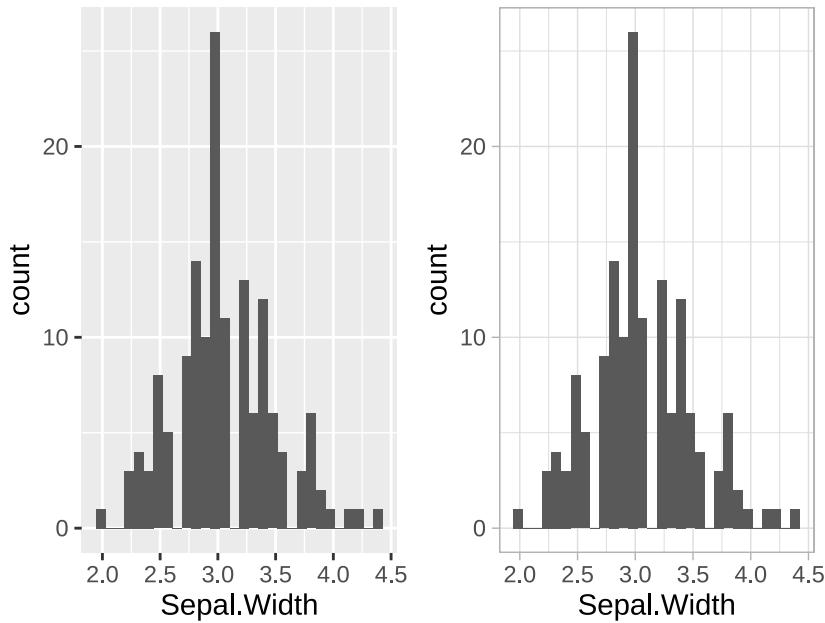
1205 **9.4.2 Plots kombinieren**

- 1206 Es gibt Situationen, in denen **unterschiedliche** Plots miteinander kombiniert werden müssen. Im vorherigen  
 1207 Abschnitt wurde dies immer anhand einer gruppierenden Kovariate gemacht. Aber es gibt auch Situationen, in  
 1208 denen das nicht möglich ist. Beispielsweise wenn ein Histogramm und ein Scatterplot vom gleichen Datensatz  
 1209 zusammengefasst werden sollen. Dafür bietet sich das Paket **patchwork** an<sup>9</sup>.  
 1210 Als erstes können wir zwei (oder natürlich auch mehrere Plots) erstellen. Hier unterscheiden sich die Plots  
 1211 lediglich durch das Aussehen.

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

- 1212 Dann müssen können wir diese Plots ebenfalls mit `+` zusammenfügen.

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



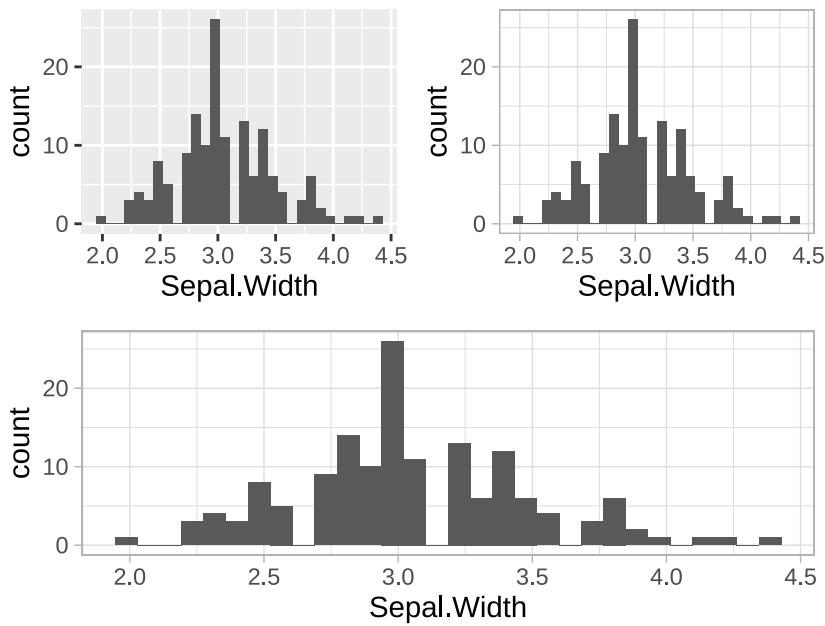
1213

- 1214 Natürlich können auch weitere Plots hinzugefügt werden (auch in unterschiedlichen Dimensionen):

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

---

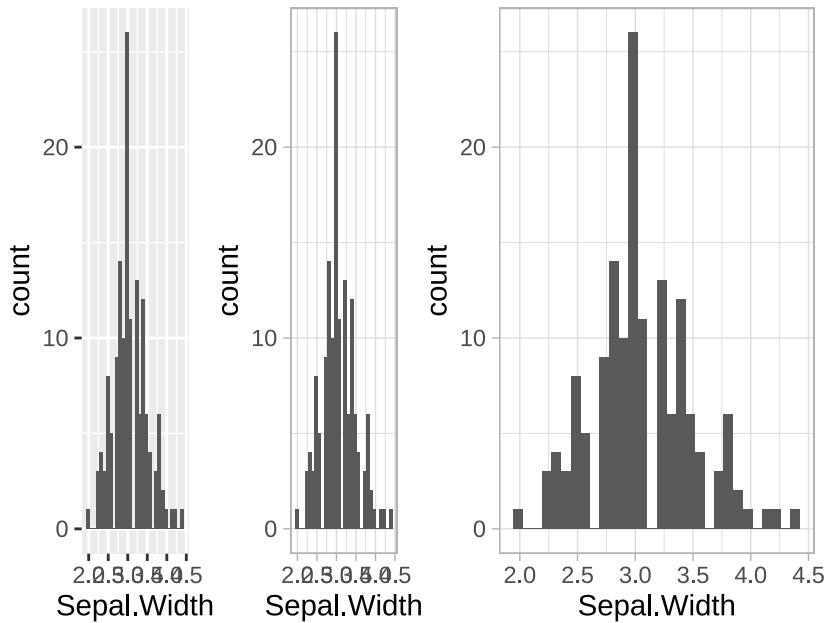
<sup>9</sup>Auch dieses Paket müssen Sie einmalig mit `install.packages("patchwork")` installieren.



1215

1216 Des weiteren können mit | auch Plots gegenüber gestellt werden.

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



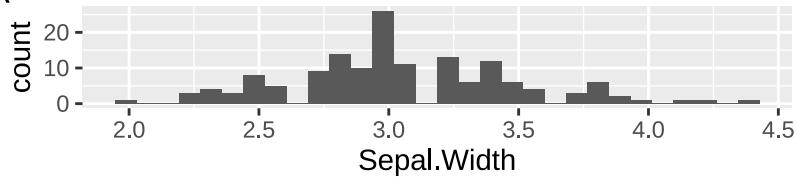
1217

1218 Weitere Optionen können mit `plot_layout()` und `plot_annotation()` angepasst werden. Mit  
1219 `plot_layout()` kann die Anordnung der Plots bestimmt werden (z.B. über die Argument `nrow`  
1220 und `ncol`), sowie deren *relative* Größe (über die Argumente `widths` und `heights`). Mit der Funktion  
1221 `plot_annotation()` können zusätzliche Beschriftungen hinzugefügt werden, wie beispielsweise eine Titel  
1222 (Argument `title`) oder ein Buchstabe/Zahl für jedes Element (Argument `tag_levels`).

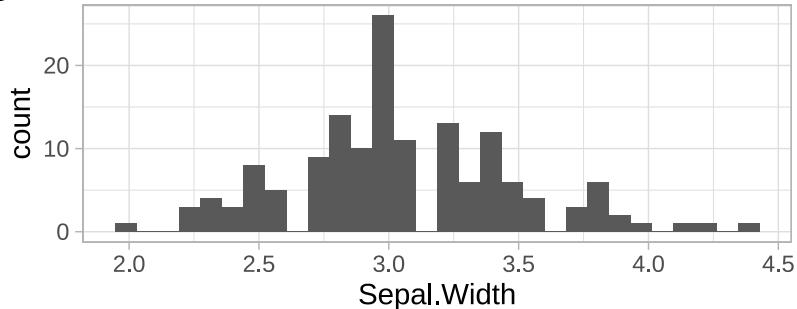
```
p1 + p2 +
 plot_layout(ncol = 1, heights = c(0.3, 0.7)) +
 plot_annotation(title = "Zwei Histogramme", tag_levels = "A")
```

## Zwei Histogramme

A



B



1223

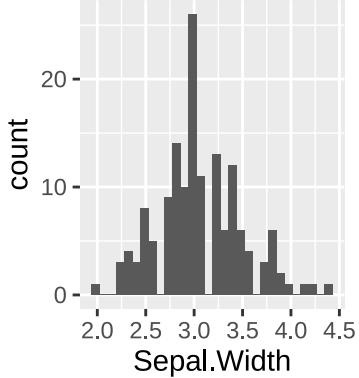
1224

### Aufgabe 23: Mehrere Plots zusammenfügen

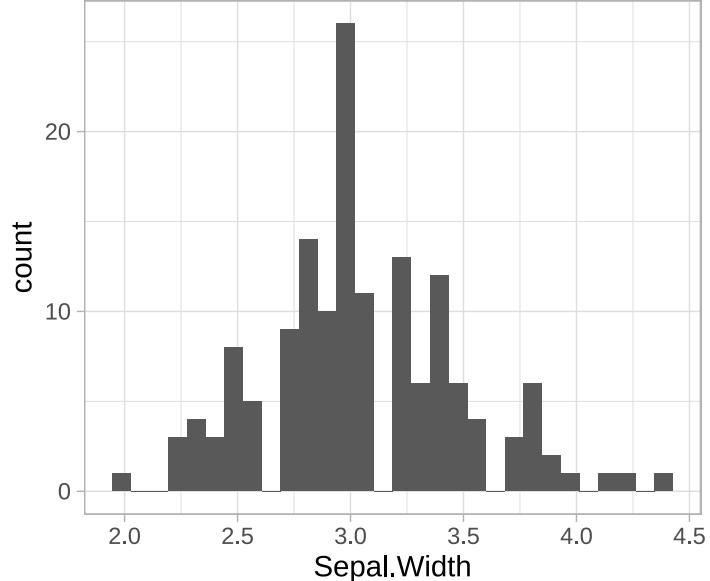
1225

1226 Versuchen Sie die folgende Zusammenstellung der Plots nachzumachen:

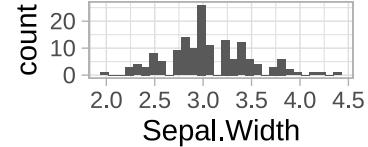
a



c



b



1228

### 9.4.3 Speichern von plots

1229 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

- 1232 Dateiformat wird aus dem Dateinamen übernommen. Die Größe der Inhalte können Sie sehr leicht mit den  
1233 Argumenten `width` und `height` oder `scale` gesteuert werden.

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

## 10 Mit Daten arbeiten

### 10.1 dplyr eine Einführung

1236 `dplyr` ist eine Erweiterung von R (ein Paket), die das Ziel hat, den Umgang mit Daten einfacher und  
1237 schneller zu machen.

1238 `dplyr` definiert 5 Verben, um mit Daten zu arbeiten. Diese sind:

- 1239 • `filter`
- 1240 • `select`
- 1241 • `arrange`
- 1242 • `mutate`
- 1243 • `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))
```

1244 Damit die Funktionen aus `dplyr` verwendet werden können, müssen wir als erstes das Paket `dplyr` laden.  
1245 The online Resource [R for Data Science](#) gibt einen sehr guten, tieferen Eindruck in die Arbeit mit `dplyr`  
1246 und erklärt auch die `dplyr` Philosophie noch Mal sehr genau für Beginner.

```
library(dplyr)
```

1247 Sollte dies zu einer Fehlermeldung führen, dann müssen Sie das Paket `dplyr` erst installieren. Dafür müssen  
1248 Sie `einmalig install.packages("dplyr")` installieren.

1249 `dplyr` stellt unterschiedliche Funktionen zum Arbeiten mit Daten zur Verfügung. Es gibt fünf Grundfunktionen  
1250 für die am häufigsten vorkommenden Operationen. Mit der Funktion `filter()` können unterschiedliche  
1251 Beobachtungen gefiltert werden:

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

```
1252 ## id plot bhd alter
1253 ## 1 1 1 50 10
1254 ## 2 2 1 29 30
1255 ## 3 3 2 13 31
1256 ## 4 4 2 23 24
1257 ## 5 5 3 25 25
```

1258 Es können auch mehrere Spalten verwendet werden.

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

```
1259 ## id plot bhd alter
1260 ## 1 2 1 29 30
1261 ## 2 3 2 13 31
1262 ## 3 4 2 23 24
```

```
1263 ## 4 5 3 25 25
```

1264 Natürlich kann genau das gleiche Ergebnis mit dem ‘normalen’ R erreicht werden, dies wäre dann:

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

```
1265 ## id plot bhd alter
```

```
1266 ## 2 2 1 29 30
```

```
1267 ## 3 3 2 13 31
```

```
1268 ## 4 4 2 23 24
```

```
1269 ## 5 5 3 25 25
```

1270 Eine weitere Funktion aus dem Paket **dplyr** ist **select()**. Damit können Spalten aus einem **data.frame**

1271 ausgewählt werden. Dabei können auch die Spaltennamen unbenannt werden.

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

```
1272 ## bhd
```

```
1273 ## 1 50
```

```
1274 ## 2 29
```

```
1275 ## 3 13
```

```
1276 ## 4 23
```

```
1277 ## 5 25
```

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

```
1278 ## bhd id
```

```
1279 ## 1 50 1
```

```
1280 ## 2 29 2
```

```
1281 ## 3 13 3
```

```
1282 ## 4 23 4
```

```
1283 ## 5 25 5
```

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

```
1284 ## BHD id
```

```
1285 ## 1 50 1
```

```
1286 ## 2 29 2
```

```
1287 ## 3 13 3
```

```
1288 ## 4 23 4
```

```
1289 ## 5 25 5
```

1290 Mit der Funktion **arrange()** können die Beobachtungen in einem **data.frame** sortiert werden.

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

```
1291 ## id plot bhd alter
```

```
1292 ## 1 3 2 13 31
```

```
1293 ## 2 4 2 23 24
```

```
1294 ## 3 5 3 25 25
```

```
1295 ## 4 2 1 29 30
1296 ## 5 1 1 50 10
```

1297 Mit der Funktion `desc()` kann die Anordnung in absteigender Reihenfolge sortiert werden.

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

```
1298 ## id plot bhd alter
1299 ## 1 1 1 50 10
1300 ## 2 2 1 29 30
1301 ## 3 5 3 25 25
1302 ## 4 4 2 23 24
1303 ## 5 3 2 13 31
```

1304 Mit der Funktion `mutate()` kann man eine neue Spalte hinzufügen.

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

```
1305 ## id plot bhd alter bhd_mm fl
1306 ## 1 1 1 50 10 500 1963.4954
1307 ## 2 2 1 29 30 290 660.5199
1308 ## 3 3 2 13 31 130 132.7323
1309 ## 4 4 2 23 24 230 415.4756
1310 ## 5 5 3 25 25 250 490.8739
```

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

```
1311 ## id plot bhd alter mean_bhd
1312 ## 1 1 1 50 10 28
1313 ## 2 2 1 29 30 28
1314 ## 3 3 2 13 31 28
1315 ## 4 4 2 23 24 28
1316 ## 5 5 3 25 25 28
```

1317 Mit der Funktion `summarise()` können Spalten zusammengefasst werden.

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

```
1318 ## mean_bhd mean_sd
1319 ## 1 28 13.63818
```

1320

1321 **Aufgabe 24: Datenmanipulation mit dplyr**

---

- 1323 1. Laden Sie den Datensatz `data/bhd_1.txt`
- 1324 2. Berechnen Sie folgende Werte für alle Einträge und speichern Sie die Ergebnisse in `erg1`
- 1325 • mittlerer `bhd`
- 1326 • maximales `alter`
- 1327 • die Standardabweichung des BHDs
- 1328 • die Anzahl Bäume mit einem BHD > 30

1329 **10.2 Arbeiten mit gruppierten Daten**

1330 Zusätzlich können `mutate` und `summarise` auch auf gruppierte Daten angewendet werden. Dafür müssen  
 1331 wir erst die Funktion `group_by()` aufrufen und als Argumente die Variablen übergeben, die die Gruppen  
 1332 definieren.

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

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

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

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

summarise(dat, bhd_m = mean(bhd))

bhd_m
1 28

summarise(dat1, bhd_m = mean(bhd))

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

```
1352 ## <dbl> <dbl>
1353 ## 1 1 39.5
1354 ## 2 2 18
1355 ## 3 3 25
```

1356

---

**Aufgabe 25: dplyr mit gruppierten Daten**

---

- 1359 1. Laden Sie den Datensatz `data/bhd_1.txt`
- 1360 2. Berechnen Sie für jede Baumart und Standort die folgenden Werte:
  - 1361 • mittlerer `bhd`
  - 1362 • maximales `alter`
  - 1363 • die Standardabweichung des BHDs
  - 1364 • die Anzahl Bäume mit einem BHD > 30
- 1365 3. Ordnen Sie das Ergebnis nach Baumart und BHD.

1366 **10.3 pipes oder %>%**

1367 Mit *Pipes* (`%>%`) kann man das Ergebnis einer Funktion einfach an eine nachfolgende Funktion weiterreichen.

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

1368 Wir kennen bis jetzt:

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

1369 ## [1] 3.333333

1370 Mit *Pipes*, die durch das Symbol `%>%` dargestellt werden<sup>10</sup>, können wir das etwas vereinfachen und nacheinander  
1371 schreiben:

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

1372 ## [1] 3.333333

1373 Oder sogar

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

1374 ## [1] 3.333333

1375

---

**Aufgabe 26: Pipes %>%**

---

1378 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.

- 1379 1. Laden Sie den Datensatz `data/bhd_1.txt`.
- 1380 2. Berechnen Sie für jede Baumart und Standort die folgenden Werte:
- 1381 • mittlerer `bhd`
- 1382 • maximales `alter`
- 1383 • die Standardabweichung des BHDs
- 1384 • die Anzahl Bäume mit einem BHD > 30
- 1385 3. Ordnen Sie das Ergebnis nach Baumart und BHD.

## 1386 10.4 Joins

1387 Eine weitere häufige Aufgabe beim Daten Management ist es Daten zusammenzuführen. Nehmen Sie an, dass  
1388 wir folgende Aufnahmen gemacht haben

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

1389 und jeder Baum lediglich mit einer `id` versehen wurde. In einer zweiten Tabelle wurden dann weitere Daten  
1390 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")
)
```

1391 Ziel ist es jetzt die Bäume aus `aufnahmen` mit den Informationen aus den `metadaten` zu verbinden. Dazu  
1392 dient `id` als Bindeglied (oft auch Schlüssel genannt).

1393 Dazu gibt es vier Möglichkeiten.

1394 Zur Durchführung gibt es in base R die Funktion `merge()`. Wir werden aber gleich die Funktionen aus dem  
1395 Paket `dplyr` verwenden.

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

1396 ## id bhd art gebiet
1397 ## 1 1 20 <NA> <NA>
1398 ## 2 2 31 Ta A
1399 ## 3 3 74 Bu B

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

1400 ## id bhd art gebiet
1401 ## 1 2 31 Ta A
1402 ## 2 3 74 Bu B
1403 ## 3 4 NA Bu B
```

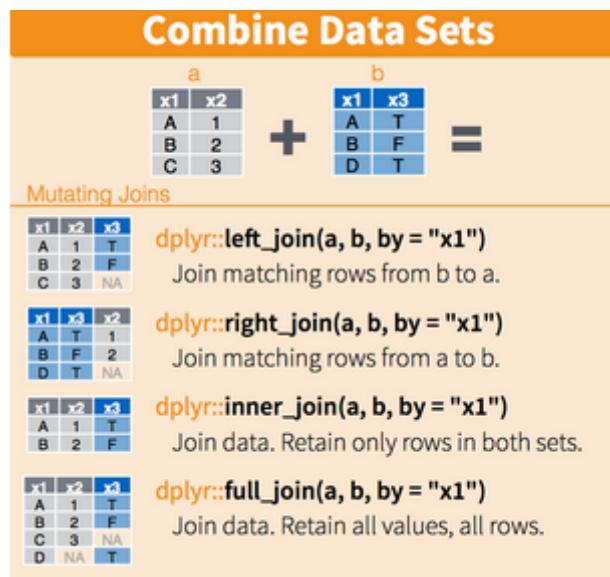


Abbildung 11: Joins (Quelle Rstudio)

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

```
1404 ## id bhd art gebiet
1405 ## 1 2 31 Ta A
1406 ## 2 3 74 Bu B
full_join(aufnahmen, metadaten, by = "id")
```

```
1407 ## id bhd art gebiet
1408 ## 1 1 20 <NA> <NA>
1409 ## 2 2 31 Ta A
1410 ## 3 3 74 Bu B
1411 ## 4 4 NA Bu B
```

1412 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"))
```

```
1413 ## id bhd art gebiet
1414 ## 1 1 20 <NA> <NA>
1415 ## 2 2 31 Ta A
1416 ## 3 3 74 Bu B
```

1417

1418 **Aufgabe 27: Verbinden von Daten**

- 
- 1419
- 1420 • Lesen Sie die Datensätze `daten/bhd_2.txt` und `daten/bhd_2_meta.txt` ein.
  - 1421 • Stellen Sie sicher, dass es für den `bhd` keine fehlenden Werte gibt (entfernen sie entsprechende Zeilen)
  - 1422 • Fügen Sie zu den Metadaten (gespeichert in `bhd_2_meta`) die Anzahl Bäume und den mittleren `bhd` hinzu pro Gebiet.

1424 **10.5 ‘long’ and ‘wide’ Datenformate**

1425 Unter anderem Wickham (2014) empfieilt das Prinzip von *tidy* Data. Nach diesem Prinzip sollten Daten wie  
1426 folgt organisiert sein:

- 1427
- 1428 • Jede Zeile ist ein Merkmalsträger/Subjekt/Objekt (z. B. eine Person oder ein Baum).
  - 1429 • Jede Spalte ist eine Variable (ein Merkmal), die den Merkmalsträger beschreibt.
  - 1430 • In jeder Zelle ist genau ein Wert (Merkmalausprägung), nämlich der Wert, der Variable für den Merkmalsträger.

1431 Zum Beispiel enthalten Spaltennamen oft Informationen, die eigentlich in einer Variable gespeichert werden  
1432 sollten. Folgendes Beispiel gibt die BHD Messung von 3 Bäumen in 3 Jahren wieder. Der Vorteil von *tidy*  
1433 Daten ist, dass viele Funktionen diese Form erwarten. Somit müssen sie Ihre Daten nur ein Mal organisieren  
1434 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)
)
```

1435 Der `tibble` aus dem Paket `tibble` ist ein moderner Data Frame, der sich in seinen Funktionen in das `tidy`  
1436 Universum einfügt. Wir können auch einfach das Paket `tidyverse` laden, um alle Pakete, die Teil des `tidy`  
1437 Universums sind gleichzeitig zu laden. Der `tibble` beinhaltet alle Funktionen, die der Data Frame auch  
1438 beinhaltet und kann auch für genau die gleichen Zwecke verwendet werden. Der `tibble` hat u. A. eine  
1439 modernere Darstellung im Konsolenoutput.

1440 Diese Daten sind jetzt im `wide`-Format gespeichert und folglich nicht `tidy`, weil Information über die Daten  
1441 (nämlich das Jahr der Aufnahme in den Spaltennamen gespeichert sind). Besser wäre eine Struktur mit  
1442 nur drei Spalten: `id`, `jahr` und `bhd`. Um die Daten in so eine Struktur zu bringen, gibt es die Funktion  
1443 `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
```

```

1446 ## <int> <chr> <dbl>
1447 ## 1 1 bhd2015 30
1448 ## 2 1 bhd2016 31
1449 ## 3 1 bhd2017 34
1450 ## 4 2 bhd2015 31
1451 ## 5 2 bhd2016 31
1452 ## 6 2 bhd2017 32
1453 ## 7 3 bhd2015 32
1454 ## 8 3 bhd2016 33
1455 ## 9 3 bhd2017 33

```

1456 Wenn wir die Spalten für die Variable und den Wert gleich sinnvoll benennen möchten, können wir das über  
1457 die Argumente `names_to` und `values_to` machen.

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

```

1458 ## # A tibble: 9 x 3
1459 ## id jahr bhd
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 Analog zu der Funktion `pivot_longer()` gibt es auch die Funktion `pivot_wider()`, um vom Daten vom  
1471 `long`-Format ins `wide`-Format zu transformieren.

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

```

1472 ## # A tibble: 3 x 4
1473 ## id bhd2015 bhd2016 bhd2017
1474 ## <int> <dbl> <dbl> <dbl>
1475 ## 1 1 30 31 34
1476 ## 2 2 31 31 32
1477 ## 3 3 32 33 33

```

1478

---

1479 **Aufgabe 28: Zeitliche Verlauf von BHDS**

---

1481 In der Datei `bhd_3.csv` befinden sich gemessene BHDS (in cm) von unterschiedlichen Bäumen zu unter-  
 1482 schiedlichen Zeitpunkten. Erstellen Sie ein Liniendiagramm, das den zeitlichen (x-Achse) Verlauf der BHDS  
 1483 (y-Achse) für die unterschiedlichen Bäume darstellt.

1484 **10.6 Auswählen von Variablen**

1485 Sobald die Datensätze etwas umfangreicher werden (d. h., es gibt mehrere Spalten in einem `data.frame`),  
 1486 können innerhalb vieler `dplyr`-Funktionen spezielle Funktionen verwendet werden, um Variablen auszuwählen.

1487 Wenn die genaue Position der Spalten bekannt ist, kann man mit dem `:`-Operator und der Position Spalten  
 1488 auswählen:

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

1489 ## Sepal.Length Sepal.Width Petal.Length  
 1490 ## 1 5.1 3.5 1.4  
 1491 ## 2 4.9 3.0 1.4  
 1492 ## 3 4.7 3.2 1.3

1493 Diese Vorgehensweise kann gehährlich sein, da sich manchmal Spalten verschieben und sich somit die  
 1494 Positionen ändern. Es besser Spalten immer explizit mit Namen statt mit Nummern anzusprechen.

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

1495 ## Sepal.Length Sepal.Width Petal.Length  
 1496 ## 1 5.1 3.5 1.4  
 1497 ## 2 4.9 3.0 1.4  
 1498 ## 3 4.7 3.2 1.3

1499 `select()` erlaubt es, auch hier den `:`-Operator zu verwenden:

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

1500 ## Sepal.Length Sepal.Width Petal.Length  
 1501 ## 1 5.1 3.5 1.4  
 1502 ## 2 4.9 3.0 1.4  
 1503 ## 3 4.7 3.2 1.3

1504 Es gibt auch einige spezielle Funktionen, um Spalten innerhalb eines `select()`-Aufrufs auszuführen:

- 1505 • `starts_with()`: Hier kann man ein Muster angeben, mit dem ein Text anfangen muss.
- 1506 • `ends_with()`: Diese Funktion ist analog zu `starts_with()`, jetzt wird aber am Ende des Spaltennamens  
 1507 nach dem Muster gesucht.
- 1508 • `contains()`: Hier kann ein Muster übergeben werden, das irgendwo im Spaltennamen sein muss.
- 1509 • `everything()`: Mit dieser Funktion werden alle Spalten ausgewählt.

1510 • `last_col()`: Mit dieser Funktion wird nur die letzte Spalte ausgewählt (dass ist die Spalte, die ganz  
1511 rechts ist).

1512 Sämtliche Auswahlen können mit - umgekehrt werden.

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

1513 ## Sepal.Length Sepal.Width

1514 ## 1 5.1 3.5

1515 ## 2 4.9 3.0

1516 ## 3 4.7 3.2

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

1517 ## Petal.Length Petal.Width Species

1518 ## 1 1.4 0.2 setosa

1519 ## 2 1.4 0.2 setosa

1520 ## 3 1.3 0.2 setosa

1521 `select()` bietet auch noch die Möglichkeit, Spaltennamen zu ändern.

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

1522 ## sep\_width

1523 ## 1 3.5

1524 ## 2 3.0

1525 ## 3 3.2

1526

---

### 1527 Aufgabe 29: Auswählen von Spalten

---

1529 In der Datei `messungen_1.csv` sind Messungen von zwei Sensoren enthalten für die ersten vier Monate eines  
1530 Jahres. Führen Sie folgende Abfragen durch:

1531 1. Wählen Sie alle Messungen für Januar aus.

1532 2. Wählen Sie alle Messungen für Januar und März aus.

## 1533 10.7 Einzelne Beobachtungen abfragen (`slice()`)

1534 Mit dem Befehl `slice()` kann man einzelne Beobachtungen (= Zeilen) aus einem `data.frame` abfragen.

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

1535 ## Sepal.Length Sepal.Width Petal.Length Petal.Width Species

1536 ## 1 5.1 3.5 1.4 0.2 setosa

1537 ## 2 4.4 2.9 1.4 0.2 setosa

1538 ## 3 5.1 3.5 1.4 0.3 setosa

1539 Davon gibt es drei nützliche Varianten: 1) `slice_head()` und `slice_tail()`; 2) `slice_max()` und  
 1540 `slice_min()`; 3) `slice_random()`.

1541 `slice_head()` und `slice_tail()` sind analog zu `head()` und `tail()`, aber mit dem entscheidenden Unter-  
 1542 schied, dass Gruppierungen berücksichtigt werden. Wenn keine Gruppierung in den Daten vorhanden ist, gibt  
 1543 es keinen Unterschied.

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

```
1544 ## Sepal.Length Sepal.Width Petal.Length Petal.Width Species
1545 ## 1 5.1 3.5 1.4 0.2 setosa
1546 ## 2 4.9 3.0 1.4 0.2 setosa
```

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

```
1547 ## Sepal.Length Sepal.Width Petal.Length Petal.Width Species
1548 ## 1 5.1 3.5 1.4 0.2 setosa
1549 ## 2 4.9 3.0 1.4 0.2 setosa
```

1550 Sobald jedoch eine gruppierende Variable eingeführt wird, gibt `slice_head()` die ersten n Beobachtungen  
 1551 für jede Gruppe zurück und `head()` für den gesamten Datensatz.

```
base head
```

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

```
1552 ## # A tibble: 2 x 5
1553 ## # Groups: Species [1]
1554 ## Sepal.Length Sepal.Width Petal.Length Petal.Width Species
1555 ## <dbl> <dbl> <dbl> <dbl> <fct>
1556 ## 1 5.1 3.5 1.4 0.2 setosa
1557 ## 2 4.9 3 1.4 0.2 setosa
```

```
dplyr slice_head
```

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

```
1558 ## # A tibble: 6 x 5
1559 ## # Groups: Species [3]
1560 ## Sepal.Length Sepal.Width Petal.Length Petal.Width Species
1561 ## <dbl> <dbl> <dbl> <dbl> <fct>
1562 ## 1 5.1 3.5 1.4 0.2 setosa
1563 ## 2 4.9 3 1.4 0.2 setosa
1564 ## 3 7 3.2 4.7 1.4 versicolor
1565 ## 4 6.4 3.2 4.5 1.5 versicolor
1566 ## 5 6.3 3.3 6 2.5 virginica
1567 ## 6 5.8 2.7 5.1 1.9 virginica
```

1568 `slice_tail()` funktioniert analog zu `slice_head()` mit dem einzigen Unterschied, dass nicht die ersten n

1569 Zeilen zurück gegeben werden sondern die letzten `n` Zeilen.  
 1570 `slice_max()` und `slice_min()` geben die Beobachtung mit dem maximalen bzw. minimalen Wert einer  
 1571 Variable zurück. Auch hier werden Gruppen berücksichtigt.

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

1572 ## Sepal.Length Sepal.Width Petal.Length Petal.Width Species  
 1573 ## 1 7.9 3.8 6.4 2 virginica

1574 Und mit Gruppen:

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

 slice_max(Sepal.Length)
```

1575 ## # A tibble: 3 x 5  
 1576 ## # Groups: Species [3]  
 1577 ## Sepal.Length Sepal.Width Petal.Length Petal.Width Species  
 1578 ## <dbl> <dbl> <dbl> <dbl> <fct>  
 1579 ## 1 5.8 4 1.2 0.2 setosa  
 1580 ## 2 7 3.2 4.7 1.4 versicolor  
 1581 ## 3 7.9 3.8 6.4 2 virginica

1582 `slice_min()` funktioniert genau gleich, nur dass die Beobachtung (=Zeile) mit dem kleinsten Wert einer  
 1583 Variable zurück gegeben wird.

1584 Die Funktion `slice_sample()` erlaubt es zufällige Beobachtungen zu ziehen. Dabei kann über das Argument `n`  
 1585 die Anzahl an Beobachtungen angegeben werden oder über das Argument `prop` der Anteil an Beobachtungen.

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

1586 ## Sepal.Length Sepal.Width Petal.Length Petal.Width Species  
 1587 ## 1 6.5 2.8 4.6 1.5 versicolor  
 1588 ## 2 6.3 3.3 4.7 1.6 versicolor  
 1589 ## 3 7.2 3.2 6.0 1.8 virginica  
 1590 ## 4 4.9 3.6 1.4 0.1 setosa  
 1591 ## 5 6.0 2.7 5.1 1.6 versicolor

1592 Das Ergebnis ist bei jedem von Ihnen anders, da es sich um eine zufällige Ziehung handelt. Wenn Sie diese  
 1593 Ergebnisse wiederholen möchten, können Sie über `set.seed()` die zufällige Ziehung reproduzierbar machen.

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

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

1594 ## Sepal.Length Sepal.Width Petal.Length Petal.Width Species  
 1595 ## 1 4.3 3.0 1.1 0.1 setosa  
 1596 ## 2 5.0 3.3 1.4 0.2 setosa  
 1597 ## 3 7.7 3.8 6.7 2.2 virginica  
 1598 ## 4 4.4 3.2 1.3 0.2 setosa  
 1599 ## 5 5.9 3.0 5.1 1.8 virginica

1600 Wenn beispielsweise 5% der Beobachtungen gezogen werden sollen, kann dies so gemacht werden:

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

```
1601 ## Sepal.Length Sepal.Width Petal.Length Petal.Width Species
1602 ## 1 7.7 3.8 6.7 2.2 virginica
1603 ## 2 5.5 2.5 4.0 1.3 versicolor
1604 ## 3 5.5 2.6 4.4 1.2 versicolor
1605 ## 4 6.5 3.0 5.2 2.0 virginica
1606 ## 5 6.1 3.0 4.6 1.4 versicolor
1607 ## 6 6.3 3.4 5.6 2.4 virginica
1608 ## 7 5.1 2.5 3.0 1.1 versicolor

1609 slice_sample() berücksichtigt ebenfalls Gruppen. Mit den Argumenten replace und weight_by dann die
1610 Zufallsziehung genauer spezifiziert werden. replace sagt, ob eine gezogenen Beobachtung wieder zurück gelegt
1611 wird oder nicht. Mit dem Argument weight_by können optional gewichtete für jede Beobachtung vergeben
1612 werden.
```

1613

### 1614 Aufgabe 30: Daten beschreiben

---

1616 Verwenden Sie den Datensatz `bhd_1.txt` und finden Sie für jedes Gebiet und Art die Beobachtung mit
1617 kleinsten BHD.

## 1618 10.8 Spalten trennen

1619 Ein gut geplanter Datensatz besteht aus Beobachtungen (Zeilen), Variablen (Spalten) und in jeder Zelle ist
1620 immer ein **genau** ein Wert gespeichert. Leider gibt es oft Datensätze, bei denen mehr als ein Wert pro Zelle
1621 gespeichert wurde. Die Funktion `separate()` kann helfen solche Daten zu trennen.

1622 Wir verwenden einen erfunden Datensatz zu Beobachtungen von Tieren und einer geschätzten Distanz zu
1623 diesen Tieren.

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

1624 In der Spalte `beobachtung` sind zwei Informationen gespeichert: Die Distanz zur Beobachtung und die Art.
1625 Das ist ungünstig, weil wir so weder nach Tierart noch nach distanz filtern können (nicht *tidy*). Mit der
1626 Funktion `separate()`, können wir Beobachtungen einer Spalte in mehrere Spalten trennen. Dafür muss der
1627 Spaltennamen (Argument `col`), die neuen Spaltennamen (Argument `into`) und das Trennzeichen (Argument
1628 `sep`) angegeben werden.

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

1629 ## # A tibble: 4 x 3

```
1630 ## id Distanz Art
1631 ## <int> <chr> <chr>
1632 ## 1 1 10m " Reh"
1633 ## 2 2 100m " Reh"
1634 ## 3 3 20m " Fuchs"
1635 ## 4 4 40 "Reh"
1636 Nach dem Aufruf von separate() gibt es zwei neue Spalten (Distanz und Art), die die alte Spalte
1637 beobachtung ersetzen.
```

1638

---

1639 **Aufgabe 31: Aufräumen**

---

1641 Verwenden Sie den folgenden Datensatz und bringen Sie ihn in eine Form, die sicherstellt dass

1642 • jede Zelle genau einen Wert enthält.  
1643 • jede Zeile eine Beobachtung ist.  
1644 • 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")
)
```

## 1645 11 Arbeiten mit Text

1646 Bis jetzt haben wir fast ausschließlich mit Zahlen oder Abbildungen gearbeitet. R bietet aber auch viele  
 1647 Werkzeuge, um mit Text zu arbeiten. Wir wollen hier ein paar Funktionen dafür vorstellen. Als erstes sollte  
 1648 nochmals klar gestellt werden, was eigentlich ein Text ist. In R ist alles, das innerhalb von doppelten ("") oder  
 1649 einfachen ('') Anführungszeichen geschrieben ist, Text.

1650 Anbei einige Beispiele:

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

1651 Wichtig ist hier zu sehen, dass z nicht als Zahl sondern, als Text interpretiert wird (siehe Datentypen).

```
z + 1
```

1652 ## Error in z + 1: nicht-numerisches Argument für binären Operator

1653 Wenn man sicher ist, dass es sich bei einem Textobjekt um eine Zahl handelt, kann man dies mit der Funktion  
 1654 `as.numeric()` in eine Zahl umwandeln.

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

1655 ## [1] 31

1656 Aber mit a führt dies wieder zu einem NA-Wert, da a nicht in eine Zahl umgewandelt werden kann.

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

1657 ## Warning: NAs durch Umwandlung erzeugt

1658 ## [1] NA

### 1659 11.1 Arbeiten mit Text

1660 Wir wollen erst einmal drei Funktionen besprechen, die es erlauben mit Text zu arbeiten. Die Funktion  
 1661 `nchar()`<sup>11</sup> gibt an wie viele Zeichen ein Text hat. Also z.B.

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

1662 ## [1] 5

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

1663 ## [1] 2

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

1664 ## [1] 20

1665 Die Funktion `paste()` erlaubt es verschiedene Variablen mit Text zu verbinden. Wenn wir z. B. die Variablen  
 1666 `vorname <- "Eva"` und `name <- "Meier"` haben und wir wollen eine neue Variable `full_name <- Eva`

---

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

1667 "Meier" erzeugen, dann kann das mit der Funktion `paste()` gemacht werden.

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

1668 ## [1] "Eva Meier"

1669 Die Funktion `paste()` hat das Argument `sep`, das auf ein Leerzeichen ( ) gesetzt ist, aber auch anders sein  
1670 kann und das Trennzeichen definiert.

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

1671 ## [1] "Eva, Meier"

1672 Die Funktion `substr()` erlaubt es am Anfang oder Ende eines Wortes etwas abzuschneiden. Dabei muss  
1673 immer die Anfangs- (`start`) und Endposition (`stop`) angegeben werden.

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

1674 ## [1] "Hal"

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

1675 ## [1] "allo"

1676

---

### 1677 Aufgabe 32: Arbeiten mit Text 1

---

1679 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")
```

1680 1. Aus wie vielen Buchstaben besteht jedes Wort?

1681 2. Finden Sie das längste Wort.

1682 3. Wie viel Prozent der Wörter fangen mit einem S an?

1683 4. Fügen Sie jedem Wort seine Position im Vektor hinzu. Beispielsweise soll aus `Vogel` "2. Vogel" werden  
1684 usw.

## 1685 11.2 Finden von Textmustern

1686 Mit der Funktion `grep()` können Muster in einem Text gefunden werden. Wenn wir beispielsweise folgenden  
1687 Vektor mit Textelementen haben.

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

1688 Und wir wollen alle Straßennamen die ein `weg` haben abfragen, dann können wir folgenden Befehl ausführen:

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

1689 `## [1] 2`

1690 Im zweiten Element von `txt` kommen die Zeichen `Weg` vor. Beachte, in der Standardeinstellung wird zwischen  
1691 Groß- und Kleinschreibung unterschieden. Dies kann mit dem Argument `ignore.case = TRUE` angepasst  
1692 werden.

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

1693 `## [1] 1 2`

1694 Mit der Funktion `sub` können Zeichen innerhalb einer Zeichenkette ersetzt werden.

1695 So ersetzt der folgende Ausdruck `ae` mit `ä`.

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

1696 `## [1] "Friedländer Weg"`

1697 Wenn allerdings das zu ersetzende Zeichen mehr als einmal vorkommt und beide Instanzen ersetzt werden  
1698 sollen braucht man die Funktion `gsub`.

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

1699 `## [1] "Friedländer Weg und Reinhäuser Landstrasse sind zwei Strassen in Goettingen."`

1700 Mit `sub()` wird nur das erste `ae` ersetzt, während `gsub()` alle `ae` mit einem `ä` ersetzt.

```
gsub("ae", "ä", txt)
```

1701 `## [1] "Friedländer Weg und Reinhäuser Landstrasse sind zwei Strassen in Goettingen."`

1702 Oft ist der genaue Ausdruck den man finden möchte jedoch Variabel. Beispielsweise möchte man alle Wörter  
1703 mit einem Umlaut oder Zahlen finden möchte, kann man das oft abkürzen. Dafür gibt es reguläre Ausdrücke.  
1704 Wir werden hier nur ein paar beispielhafte Anwendungen besprechen.

1705 Sowohl in den Funktionen `grep()` als auch `(g)sub()` kann mit anstatt dem Muster (immer das erste  
1706 Argument) aus ein regulärer Ausdruck angegeben werden. Mit `[1-9]` sind alle Zahlen von 1 bis 9 gemeint.

1707 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)
```

1708 `## [1] 1 3`

1709 Damit lässt sich auch das Problem mit groß- und kleingeschriebenen Wörtern lösen.

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

```
1710 ## [1] 1 2
```

1711

---

**Aufgabe 33: Arbeiten mit Text 2**

---

1714 Verwenden Sie den folgenden Vektor:

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

1715 1. In wie vielen Wörtern kommt der Doppellaut au vor?

1716 2. Ersetzen Sie in allen Wörtern alle au mit \_ \_.

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

```
1717 ## [1] 5 6 13
```

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

```
1718 ## [1] "Versicherung" "Methoden" "Fluss" "Rudel" "B_ _m"
1719 ## [6] "H_ _s" "Foto" "Auffahrt" "Auto" "Handy"
1720 ## [11] "Teller" "Kalender" "Aufb_ _"
```

## 1721 12 Arbeiten mit Zeit

1722 Für den Computer bzw. R ist ein Datum erst einmal nichts anderes als ein Text. Für uns ist es sofort  
 1723 klar, dass der "13.2.2021" der 13. Februar 2021 ist, für den Computer zunächst nicht. Wir müssen R also  
 1724 irgendwie sagen, dass die 13 der Tag ist, die 2 der Monat und 2021 das Jahr. Dass der Computer die einzelnen  
 1725 Komponenten erkennt, und in einen Datentyp eigens für Zeitformate überführt, nennt man *parsen*<sup>12</sup>. Durch  
 1726 das *parsen* wird die Variable in den Datentyp **Date** überführt. Das Arbeiten mit Datum und Zeit kann  
 1727 kann anfangs sehr mühsam sein und viele Zeit-spezifischen Datenoperationen lassen sich auch mit den  
 1728 Basis-Datentypen durchführen. Sobald man einige Grundfertigkeiten erworben hat, stellt man jedoch fest,  
 1729 dass die Arbeit mit dem Zeitformat-Datentyp schneller und effizienter funktioniert. Starten Sie am besten  
 1730 gleich mit "echten" Zeit-Datentypen und versuchen Sie nicht, sich irgendwie mit Text und numerischen  
 1731 Datentypen selbst etwas zu basteln. Der erste Schritt ist immer ein Datum zu *parsen*. Wir verwenden dafür  
 1732 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)
```

1733 **lubridate** bietet eine Vielzahl von Funktionen zum *parsen* von Datum und Zeit, die sich aus:

- 1734 • y für Jahr,
- 1735 • m für Monat,
- 1736 • d für Tag,
- 1737 • h für Stunde,
- 1738 • m für Minute und
- 1739 • s für Sekunde

1740 zusammen setzten. Alle Funktionen nehmen als erstes Argument ein Textstring. Wenn wir z.B. den String  
 1741 2020-01-20 parsen wollen können wir das mit der Funktion **ymd** machen.

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

1742 ## [1] "2020-01-20"

1743 Dabei erkennt **lubridate** in der Regel die Trennzeichen:

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

1744 ## [1] "2020-01-20"

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

1745 ## [1] "2020-01-20"

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

1746 ## [1] "2020-01-20"

1747 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")

1748 ## [1] "2020-01-20"

1749 Jetzt stellt sich die Frage, was der Vorteil ist, wenn R ein Datum parst.

d <- dmy("20.1.2020")

1750 Wir können jetzt mit d arbeiten und einzelne Komponenten extrahieren.

day(d)

1751 ## [1] 20

month(d)

1752 ## [1] 1

year(d)

1753 ## [1] 2020

1754 Oder auch Zeiteinheiten hinzufügen oder abziehen.

d + days(10)

1755 ## [1] "2020-01-30"

d - years(20)

1756 ## [1] "2000-01-20"

d + hours(25)

1757 ## [1] "2020-01-21 01:00:00 UTC"

```

1758

---

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

---

- 1759
- 1760
- 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.
  - Extrahieren Sie nun aus jedem Element aus d das Jahr und die Stunde.
  - Fügen zu jedem Element in d 10 Tage hinzu.

1765 **12.1 Arbeiten mit Zeitintervallen**

1766 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)
```

1768 Wir können jetzt mit `int` arbeiten und beispielsweise das Intervall verschieben,

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

1769 ## [1] 2023-03-18 UTC--2024-03-18 UTC

1770 die Länge des Intervalls berechnen

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

1771 ## [1] 31536000

1772 oder testen, ob ein Datum innerhalb des Intervalls liegt.

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

1773 ## [1] TRUE

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

1774 ## [1] FALSE

1775 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)
```

1776 ## # A tibble: 1 x 1

1777 ## a

1778 ## <date>

1779 ## 1 2020-07-01

1780 `%within%` funktioniert genauso mit Vekotren oder mit mehreren Intervallen. Wir könnten also zwei Intervalle  
1781 definieren (z.B. Ostern und Pfingsten).

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

1782 Und Überprüfen welche Termine in eines der zwei Intervalle fallen.

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

```
Ostern
```

```
termine %within% ostern
```

1783 ## [1] FALSE TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE

```
Pfingsten
termine %within% pfingsten

1784 ## [1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE TRUE FALSE FALSE
1785 Der Zeit-Datentyp wird auch oft benutzt, um den Zeitbedarf von längeren Berechnungen zu messen.

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

1786 ## [1] 0.4999484
t2 <- now()
int_length(interval(t1, t2))

1787 ## [1] 0.6111724
```

## 12.2 Formatieren von Zeit

1788 Für die Ausgabe in Berichten oder Grafiken soll das Datum oft in einer speziellen Form dargestellt werden.  
 1789 Die Funktion `format()` bietet Möglichkeiten ein Datumsobjekt zurück in Text umzuwandeln.  
 1790 Ein Beispiel wäre `ymd("2021-2-10")` als 10.2.21 auszugeben.

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

1791 Dabei handelt sich bei `%d.%m.%y` um Abkürzungen für die unterschiedlichen Komponenten eines Datumobjekts.  
 1792 Siehe dazu die Hilfeseite von `strptime (help(strptime))`.

1795

## Aufgabe 35: Arbeiten mit Intervallen

1796 Wie viele Einträge aus dem Vektor `v1` befinden sich in einem Intervall, das zwischen dem 1.3.2021 und dem  
 1797 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"
)
```

## 12.3 Zeitreihen

1800 In der Forstbranche haben wir oft mit Zeitreihen zu tun. Zeitreihen sind Variablen, für die in zeitlichen  
 1801 Intervallen Daten vorliegen. In der Regel monatlich, vierteljährig oder jährlich, wobei die Intervalle zwischen  
 1802 den Messungen bei Zeitreihen immer gleich lang sind. Wiederholungsmessungen von Forsteinventuren (Forstein-  
 1803 richtungen, Betriebsinventuren, die meisten forstl. Experimente, die BWI, ...) sind also in der Regel keine  
 1804

1805 Zeitreihen in engeren Sinne. Turnusmäßig vermessene Versuchsflächen, wie sie z. B. die Versuchsanstalten  
 1806 unterhalten oder jährlich gemeldete Holzpreise jedoch schon.

1807 Zeitreihen unterscheiden sich nicht nur technisch, sondern auch inhaltlich fundamental von den uns schon  
 1808 bekannten Daten. Statistisch gesehen haben Zeitreihen Besonderheiten gegenüber anderen Variablen, da  
 1809 Sie von Ihrer eigenen Vergangenheit abhängen (autokorriert sind) und auch die Abhängigkeit anderer  
 1810 Variablen in der Regel nur sinnvoll ist, wenn deren Vergangenheit mitberücksichtigt wird (cross-correlation).  
 1811 Konventionelle Statistik ist oft nicht möglich, um Zeitreihen zu analysieren. Selbst ein ordinärer arithmetischer  
 1812 Mittelwert ist schon nicht mehr geeignet, um Zeitreihen statistisch zu beschreiben. Angefangen mit der  
 1813 Datendarstellung gibt es in R deshalb spezifische Zeitreihen-Funktionen. Aus diesem Grund sollten Sie  
 1814 Zeitreihen als solche speichern. Viele Funktionen erkennen den Datentyp und führen entsprechend spezifische  
 1815 Zeitreihen-Operationen durch, wenn ihnen Daten vom Typ "Zeitreihe" übergeben werden. Laden wir z. B.  
 1816 die Holzpreise für Fichte 2b (das sog. Leitsortiment, Fichenholz mit einem Mittendurchmesser von 20 bis 25  
 1817 cm), das Holzaufkommen dieses Sortiments (Einschlagsvolumen) und die Preise für Nadelholz vom  
 1818 statistischen Bundesamt<sup>14</sup>. Wir laden die Daten zunächst als csv ein:

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

1819 Diese 3 Zeitreihen bilden zusammen ein klassisches Marktmodell mit dem Preis eines homogenen Gutes  
 1820 (Leitsortimentspreis), dem Angebot (Holzeinschlag) und der Nachfrage (Schnittholzpreis). Mit der Funktion  
 1821 **ts** werden die Daten in ein Zeitreihenobjekt überführt (*pasrse*). Die Spalte mit den Jahren ist dann nicht mehr  
 1822 nötig, da die Informationen zur Zeit keine eigene Variable mehr sind, sondern als sog. Metainformationen in  
 1823 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])
```

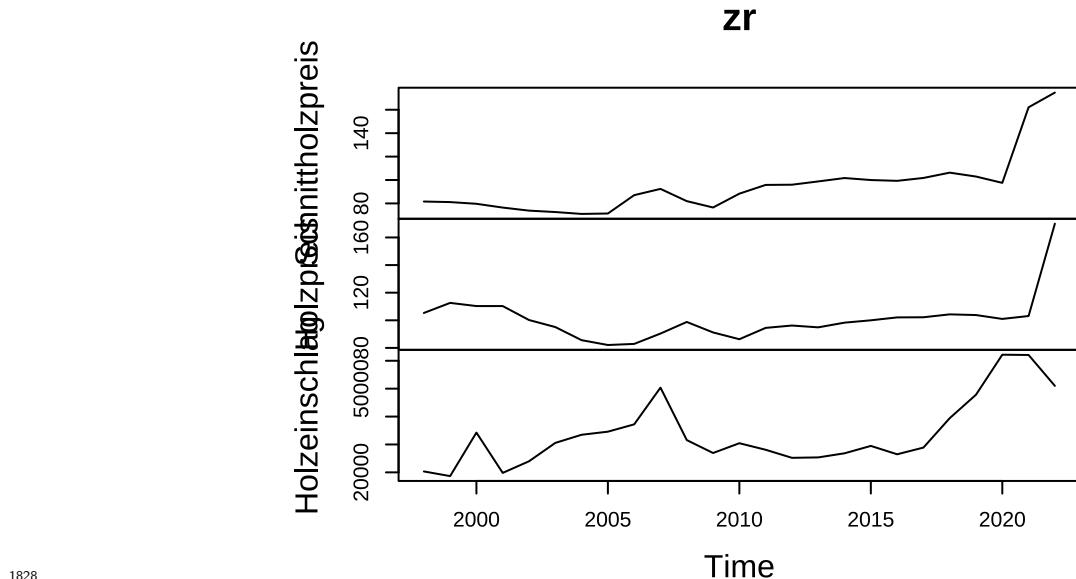
**typeof(zr)** # Zeitreihen sind sog. S3 Objekte. Das heißt, Zeitreihen gehören nicht zu den

1824 ## [1] "double"  
 # Basis Datentypen (siehe Kapitel Variablen, Funktionen und Datentypen),  
 # sondern sind eine Unterkategorie des Datentyps "Liste".

1825 Die wichtigsten Argumente sind - **data** Vektor oder Matrix, der nur die Daten enthält - **start** Startzeitpunkt -  
 1826 **frequency** Anzahl der Beobachtungen pro Zeiteinheit, also z. B. 12 bei monatlichen oder 4 bei quartalsweisen  
 1827 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



1828

1829 Wir können low-level Layer hinzufügen. Die meisten base plot Funktionen funktionieren auch für Zeitreihen

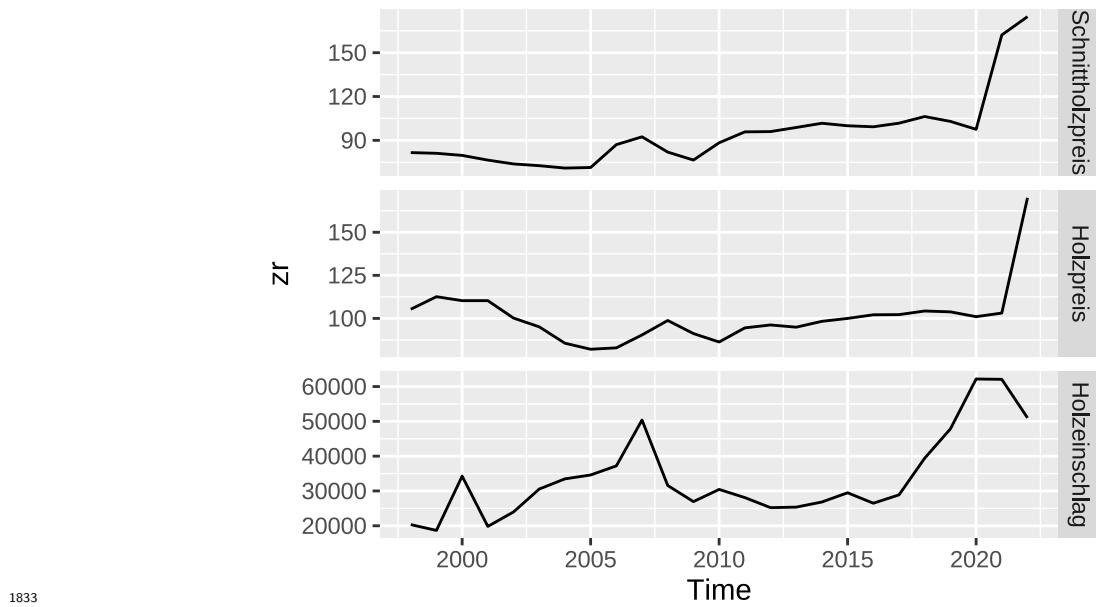
```
plot(zr, ann = FALSE, yax.flip = TRUE)
mtext(side = 3, cex = 1.5, font = 2, "Holzmarktentwicklung seit 1998", line = 2)
mtext(side = 1, cex = 1, line = 2.5, "Jahr")
```

## Holzmarktentwicklung seit 1998

1830

1831 Beide Plot-Philosophiebn haben eine Zeitreihen-Funktion. Das Paket `ggfortify` ermöglicht automatisierte  
1832 Zeitreihenplots im `ggplot2` Stil. Damit ist auch das Problem der y-Achsenbeschriftungen gelöst.

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



1833

1834 Wir können die Abbildung im `ggplot2` Stil ändern. Hier nur ein Minimalbeispiel zur Veranschaulichung.  
 1835 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

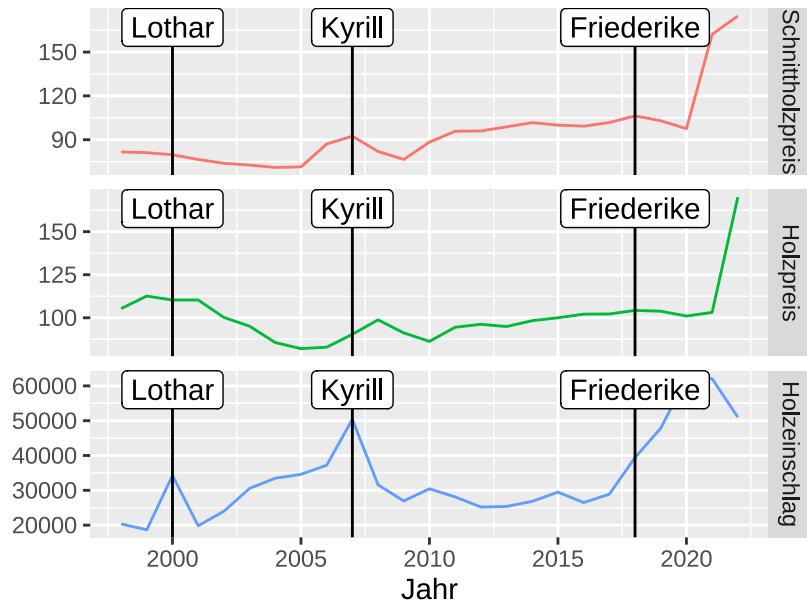
zr_autoplot + theme_minimal()
```

1836

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

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

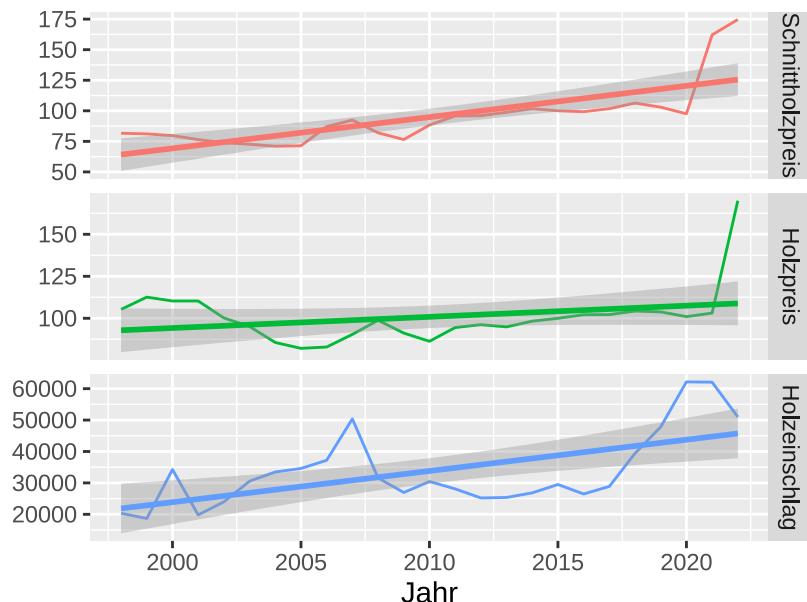
```
annotate(x = 2007, y = +Inf, label = "Kyrill", vjust = 1, geom = "label") +
 annotate(x = 2018, y = +Inf, label = "Friederike", vjust = 1, geom = "label")
```



1837

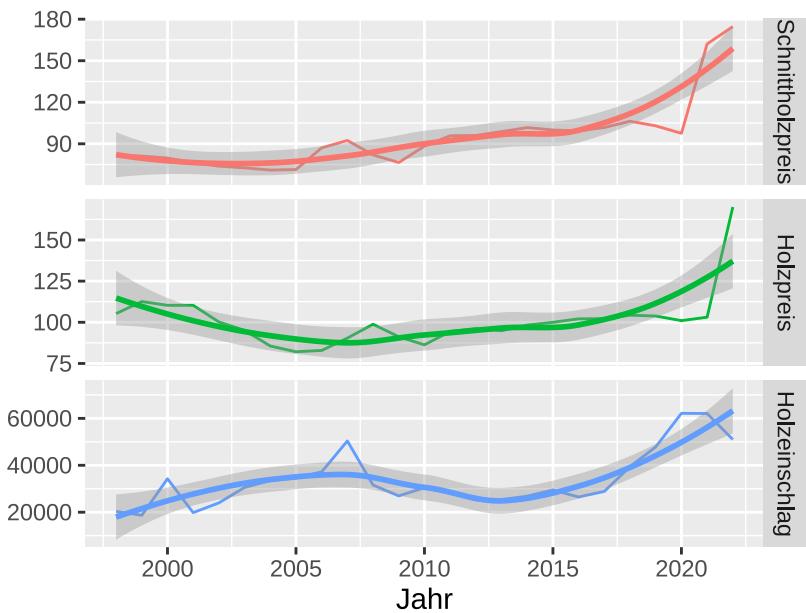
1838 Eine Trendlinie macht hier offensichtlich keinen Sinn. Die Trendlinie ist eine lineare Regression, also eine  
 1839 ordinäre Statistik, die wie eingangs erwähnt für Zeitreihen ungeeignet ist. Daher verwenden wir den sog.  
 1840 Loess-Glättter, um einen Trend in die Daten zu legen. Der Loess-Glättter ist ein sehr flexible Kurve. Wir sehen  
 1841 hier beispielsweise, dass der Leitholzpreis träge oder gar nicht auf das Angebot reagiert. Die Nachfrage jedoch  
 1842 zumindest in der einen Periode, in der sie stark steigt, den Holzpreis jedoch mit zeitlichem Verzug stark  
 1843 ansteigen lässt. Dieser visuelle Eindruck lässt sich durch spezifische Zeitreihen-Regressionen schätzen.

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



1844

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



1845

## 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

1882 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)
}
```

1883 ## [1] 1

1884 ## [1] 2

1885 ## [1] 3

1886 Das **i** steht in diesem Beispiel für die Schleifen-Variable. Sie muss nicht **i** heißen, sondern kann jeden  
 1887 zulässigen Namen annehmen. Das **X** steht für einen existierenden Vektor oder eine existierende Liste bzw.  
 1888 einen Ausdruck, der ein solches Objekt liefert (der Objektname ist ebenfalls frei wählbar). **for** und **in** sind  
 1889 Schlüsselworte, sie müssen, ebenso wie die runden Klammern, vorhanden sein.

1890 Im ersten Durchgang erhält die Schleifen-Variable **i** den ersten Wert von **X** und der Schleifenrumpf wird  
 1891 mit diesem Wert ausgeführt. Die Variable **i** nimmt nacheinander so lange die Werte von **X** an, bis ihr alle  
 1892 Elemente zugewiesen wurden.

1893 Das folgende Beispiel wird zwar besser durch die entsprechende Vektoroperation gelöst, zeigt aber sehr  
 1894 deutlich die Arbeitsweise der **for**-Schleife.

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

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

1895 ## [1] 4

1896 ## [1] 9

1897 ## [1] 25

1898

### 1899 Aufgabe 36: Schleifen 1

---

1901 Verwenden Sie den Vektor **k** <- c(1, 3, 9, 12, 15) und schreiben Sie folgende **for**-Schleifen:

- 1902 1. Eine Schleife, die jedes Element aus **k** ausgibt.
- 1903 2. Eine Schleife, die zu jedem Element aus **k** 10 addiert und den neuen Wert ausgibt.
- 1904 3. Eine Schleife wie in 2), aber der neue Wert (**k** + 10) soll jetzt nicht mehr ausgegeben werden, sondern  
 1905 in **k10** gespeichert werden. Stellen Sie sicher, dass **k10** wieder von der Länge 5 ist.

1906

---

1907 Die Funktion **for()** ermöglicht es, einen Befehl beliebig oft zu wiederholen. Z.B. der folgende Ausdruck zieht  
 1908 10-Mal eine Stichprobe der Größe 1 aus dem Vektor **v**. Beachten Sie, dass die Schleifen-Variable **i** selbst gar

1909 nicht im Schleifenrumpf vorkommt. Das Ziel dieser Schleife ist nicht die Elemente des Vektors abzuarbeiten,  
 1910 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))
}
```

```
1911 ## [1] 3
1912 ## [1] 1
1913 ## [1] 3
1914 ## [1] 3
1915 ## [1] 2
1916 ## [1] 3
1917 ## [1] 2
1918 ## [1] 2
1919 ## [1] 1
1920 ## [1] 4
```

1921 Auf gleiche Weise kann man auch über die Variablen eines Dataframes iterieren<sup>15</sup>. Das folgende Beispiel hat  
 1922 zum Ziel die Funktionsweise von Schleifen zu verdeutlichen. Schleifen haben in R jedoch den Nachteil, dass sie  
 1923 sehr langsam operieren. Wenn es geht, sollte man Alternativen verwenden. Die Funktionsweise wiederholender  
 1924 Auswertungen wird jedoch mit `for`-Schleifen deutlicher. Aus diesem Grunde werden wir uns in diesem Kurs  
 1925 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.

1934

---

**Aufgabe 37: for-Schleife**

---

1937 Lesen Sie den Datensatz `bhd_1.txt` ein und verwenden Sie eine `for`-Schleife.

- 1938 • Ziehen Sie 500-Mal je 35 Beobachtungen für den BHD.  
1939 • Berechnen Sie jeweils den Mittelwert aus den 35 Werten.  
1940 • Speichern Sie die 500 Mittelwerte in einem neuen Vektor `mittel`.  
1941 • Wie ist die Verteilung dieser 500 Mittelwerte zu interpretieren?

1942 **13.1.2 Wiederholen von Befehlen mit `while()`**

1943 Die `while`-Schleifen finden Anwendung, wenn die Anzahl der zu durchlaufenden Wiederholungen vorher  
1944 nicht bekannt ist, wie zum Beispiel bei Iterationsverfahren, die bis zu einer gewissen Genauigkeit durchlaufen  
1945 werden sollen. Die `while`-Schleife besteht in R aus dem Schlüsselwort `while()` und einer Bedingung in runden  
1946 Klammern.

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

1947 Sie ist in der praktischen Programmierung nicht so relevant wie die `for`-Schleife. Sie sei deshalb hier nur  
1948 kurz erwähnt. Die Abbruchbedingung wird jedes Mal geprüft bevor der Schleifenrumpf durchlaufen wird. Die  
1949 Bedingung wird ausgewertet und wenn diese `TRUE` ist, wird der Schleifenrumpf ausgeführt und danach erneut  
1950 die Bedingung überprüft. Ist die Bedingung nicht erfüllt, wird der Schleifenrumpf nicht durchgeführt und die  
1951 Schleife beendet. Ist die Bedingung bereits vor Eintritt in die Schleife nicht erfüllt, wird die Schleife gar nicht  
1952 erst durchlaufen.

1953 Da `while`-Schleifen also so lange ausgeführt werden, bis die Bedingung nicht mehr erfüllt ist, kann eine  
1954 Endlosschleife entstehen. Dies kann passieren, wenn man nicht sauber programmiert hat. Wenn innerhalb der  
1955 Schleife nicht dafür gesorgt wird, dass die Bedingung irgendwann nicht mehr erfüllt wird, so läuft die Schleife  
1956 immer weiter. Steckt R in einer Schleife fest und reagiert nicht mehr, kann der Befehl unter Linux mit `Strg`+  
1957 `C` und unter Windows mit `Esc` abgebrochen werden. Alternativ können Sie auf das rote STOP Symbol  
1958 über der Konsole klicken.

1959 **13.2 Bedingte Ausführung von Codeblöcken**

1960 Innerhalb eines Skripts ist es mitunter notwendig je nach aktueller Situation unterschiedlich fortzufahren.  
1961 Die Situation wird mit einem logischen Ausdruck, einer sogenannten Bedingung, geprüft. Je nachdem, ob  
1962 die Bedingung wahr (`TRUE`) oder falsch (`FALSE`) ist, werden unterschiedliche Programmteile ausgeführt, der  
1963 jeweils andere Teil bleibt unberücksichtigt. Danach wird in jedem Fall die Programmausführung, mit den  
1964 auf die bedingte Anweisungen folgenden Anweisung, fortgesetzt. In R kann dies mit dem if-else-Konstrukt  
1965 realisiert werden, welches aus den Schlüsselwörtern `if()` und `else` sowie der in runde Klammern gefassten  
1966 Bedingung besteht.

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

1967 Im folgenden Beispiel sollen die bisher abstrakt beschriebenen Vorgänge praktische Anwendung finden. In  
 1968 dem Beispiel wird zunächst durch zufälliges Ziehen einer Zahl aus dem Bereich eins bis sechs ein Würfelwurf  
 1969 simuliert. Anschließend wird der Würfelwurf mit einem if-Ausdruck kommentiert. Ist die Bedingung, es wurde  
 1970 eine Sechs gewürfelt, erfüllt, wird der Code innerhalb der geschweiften Klammern ausgeführt, ansonsten wird  
 1971 der Klammerinhalt ignoriert.

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

1972 In den meisten Fällen ist es für R irrelevant, ob sich zwischen den verschiedenen Elementen Leerzeichen oder  
 1973 Zeilenumbrüche befinden. Bei dem else-Ausdruck dagegen wird ein Fehler erzeugt, wenn vor dem `else` nicht  
 1974 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.")
}
```

1975 ## [1] "Beim nächsten Wurf klappt's bestimmt."

1976

### 1977 Aufgabe 38: Bedingte Programmierung

---

- 1979 • Wenn keine 6 gewürfelt wurde, lassen Sie zusätzlich ausgeben welche Zahl gewürfelt wurde.  
 1980 • Wiederholen Sie den Würfelwurf 10 Mal.

## 1981 14 (R)markdown

### 1982 14.1 Markdown Grundlagen

1983 Die Idee von Markdown ist, einfach Text strukturieren zu können und das ganze ohne umfangreiche Programme  
 1984 zu erstellen. Im nächsten Abschnitt sehen wir dann, wie man Markdown und R-Code verbinden kann. Hier  
 1985 soll es jedoch erst einmal darum gehen, die einfachsten Bausteine von Markdown vorzustellen.

1986 Am Anfang jedes Dokuments kommt eine Präambel. Diese fängt mit --- an und hört auch wieder mit ---  
 1987 auf. Innerhalb der Präambel können dann Metainformationen über das Dokument festgelegt werden, dies  
 1988 beinhaltet im einfachsten Fall: Titel, Autor und Datum. Das würde dann so aussehen:

```
1989 ---
1990 title: "Ein Titel"
1991 author: "Der, der es geschrieben hat"
1992 date: "März 2021"
1993 ---
```

1994 Danach folgt strukturierter und formatierter Text. Verschiedene Hierarchieebenen von Überschriften können  
 1995 mit der Anzahl an # festgelegt werden. So ist eine Überschrift erster Ordnung # Kapitel eine Überschrift  
 1996 zweiter Ordnung ## Unterkapitel usw.

1997 Listen können erstellt werden, wenn man am Anfang jeder Zeile ein - oder 1. schreibt.

```
1998 - Erster Eintrag
1999 - Zweiter Eintrag
2000 - Dritter Eintrag
2001 wird zu
```

```
2002 • Erster Eintrag
2003 • Zweiter Eintrag
2004 • Dritter Eintrag
```

2005 Eine zentrale Idee von Markdown ist es Text einfach zu formatieren. Werden eine oder mehrere Wörter mit  
 2006 zwei Sternchen (\*\*) eingefasst wird dieser Text **fett** dargestellt. Also aus \*\*wichtig\*\* wird **wichtig**. Das  
 2007 gleiche funktioniert auch mit *kursiven* Text, jedoch muss man hier noch ein Sternchen verwenden, also aus  
 2008 \*kursiv\* wird *kursiv*. Soll ein text fett und kursiv sein, kann man drei Sternchen verwenden. Aus \*\*\*sehr  
 2009 wichtig\*\*\* wird dann **sehr wichtig**.

2010 Weitere Elemente wie Links oder Abbildungen können einfach eingebunden werden. Links werden mit [Link  
 2011 text](url) in den Text integriert. Beispielsweise ist eine gute Idee bei [stackoverflow](#) bei Problemen nach  
 2012 einer Lösung zu suchen. Dieser Link wurde mit [stackoverflow](www.stackoverflow.com) erstellt.

2013 Für Abbildungen gibt es einen ganz ähnlichen Syntax. Mit ! [Das R Logo](abb/r\_logo.png) wird die  
 2014 Abbildung r\_logo.png eingebunden mit der Beschriftung: "Das R Logo".



Abbildung 12: Das R Logo

2015

2016 Aufgabe 39: Arbeiten mit markdown  
2017

2018 Verwenden Sie das folgende Markdowndokument:

```

2019 ---
2020 title: "Dokument"
2021 author: "Ihr Name"
2022 date: "März 2021"
2023 ---
2024
2025 # Einleitung
2026
2027 # Methoden
2028 1. Kopieren Sie die Vorlage in ein Dokument, das test.md heißt.
2029 2. Fügen Sie zwei Überschriften zweiter und dritter Ordnung hinzu.
2030 3. Fügen Sie einen kursiven Text hinzu.
2031 4. Fügen Sie einen Link zu einer Website hinzu.
2032 5. Kompilieren Sie die Datei, indem Sie in Rstudio auf Preview drücken (Abbildung 13).

```

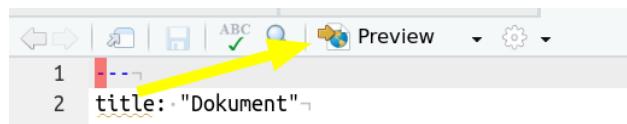


Abbildung 13: Kompilieren einer md-Datei.

## 2033 14.2 R und Markdown

2034 Markdown macht es bereits einfach Textdokumente und Dokumentationen zu verfassen, aber die wirkliche  
2035 Stärke liegt in der Möglichkeit R und Markdown zu kombinieren. Man spricht dann von Rmarkdown. Ein  
2036 weiteres Strukturelement, das wir noch nicht kennen gelernt haben, sind Code-Blöcke.

2037 ~~~

2038 a &lt;- 1:10

```

2039 a[1]
2040 ``
2041 erzeugt
2042 a <- 1:10
2043 a[1]

2044 Momentan wird noch kein Code ausgeführt, sondern lediglich der Code, als Code dargestellt. Rmarkdown
2045 bietet nun die Möglichkeit Code beim kompilieren16 auszuführen. Dafür müssen wir nur einen Code-Block als
2046 R-Code-Block kennzeichnen.

2047 ``{R}
2048 a <- 1:10
2049 a[1]
2050 ```

2051 erzeugt
2052 a <- 1:10
2053 a[1]

2052 ## [1] 1

2053 Beachte, die Variable a wird beim kompilieren erzeugt und steht dann R zur Verfügung. R-Code-Blöcke
2054 werden auch als Code Chunks bezeichnet. Diese Chunks können sehr genau angesprochen und angepasst
2055 werden. Einige wichtige Argumente sind:
2056 • echo: Gibt an, ob der Quelltext angezeigt werden soll oder nicht.
2057 • result: Gibt an, ob die Ergebnisse gezeigt werden sollen oder nicht.
2058 • eval: Diese Option gibt an ob der Chunk ausgeführt werden soll oder nicht.

```

2059

---

**Aufgabe 40: Arbeiten mit Rmarkdown**


---

2062 Erstellen Sie eine neue Rmarkdown Datei mit dem Namen `test1.Rmd`. Erstellen Sie zwei Code-Chunks. Der  
2063 erste soll nicht angezeigt werden und darin werden die Daten geladen (`bhd_1.txt`). Im zweiten Chunk plotten  
2064 Sie das Alter der Bäume gegen den BHD. Was passiert mit dem Plot, wenn Sie die Datei kompilieren (drücken  
2065 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.

## 2066 15 Räumliche Daten in R

### 2067 15.1 Was sind räumliche Daten

2068 Räumliche Daten sind Beobachtungen, wie wir sie schon oft gesehen haben, mit einem räumlichen Bezug. Der  
 2069 Unterschied zu nicht räumlichen Daten liegt darin, dass räumliche Daten eindeutig im *Raum* verortet werden  
 2070 können. Häufig werden sogenannte Geoinformationssysteme (GIS) zum Arbeiten mit räumlichen verwendet.  
 2071 R kann in vielerlei Hinsicht wie ein GIS eingesetzt werden und hier werden einige Grundfunktionalitäten  
 2072 dafür besprochen. Räumliche Daten werden in zwei unterschiedliche Datentypen unterteilt: Vektor- und  
 2073 Rasterdaten. Vektordaten modellieren einzelne Objekte (= *Features*). Rasterdaten modellieren eine Oberfläche.  
 2074 Vektordaten bestehen aus zwei Komponenten: 1) einer Geometrie, die die Form und Lage der Daten definiert  
 2075 und 2) Attributen, den tatsächlichen Daten. Räumliche Daten werden oft als *Features* bezeichnet. Ein Feature  
 2076 ist die räumliche Einheit einer Beobachtung. Je nach Art der räumlichen Daten können Features entweder  
 2077 Punkte (z.B. ein Baum), Linien (z.B. eine Straße) oder Polygone (z.B. ein See) sein. Auch können mehrere  
 2078 Geometrien zu einem Feature zusammengefasst werden. Ein Beispiel wäre eine Beobachtung für ein Land,  
 2079 das aber aus mehreren Polygonen bestehen kann (z.B. Festland und Inseln). Features können dann weitere  
 2080 Attribute (= Attributdaten) haben, z.B. eine ID, Name oder was auch immer man gemessen hat.  
 2081 Rasterdaten bestehen aus einer Oberfläche von gleichgroßen Kacheln (= *Pixel*), die ein Gebiet abdecken.  
 2082 Meist sind Pixel viereckig, aber das ist keine Voraussetzung. Dabei hat jedes Pixel einen Wert (das kann  
 2083 auch ein fehlender Wert sein). Typische Beispiele für Rasterdaten sind Landnutzung oder Seehöhen.  
 2084 In R kann sowohl mit Vektor- als auch mit Rasterdaten gearbeitet werden. Für Vektordaten bietet sich das  
 2085 Paket **sf** an und für Rasterdaten das Paket **raster**.

### 2086 15.2 Koordinatenbezugssystem

2087 Eine der Herausforderungen für räumliche Daten ist die eindeutige Verortung im Raum. Dazu braucht man  
 2088 ein *Koordinatenbezugssystem (KBS)*. Einem KBS liegt ein mathematisches Modell der Erde zugrunde. Die  
 2089 Details zu KBS werden schnell relativ kompliziert und wir beschränken uns hier lediglich darauf, wie KBS  
 2090 verwendet werden können. Dazu müssen wir zwei Fälle unterscheiden: 1) einem Datensatz ein KBS zuweisen  
 2091 und 2) Transformation des KBSs eines Datensatzes in ein anderes KBS. Die technischen Details werden in  
 2092 den folgend Abschnitten besprochen. Für beide Aufgaben müssen wir auf ein KBS verweisen können, ein  
 2093 einfacher Ansatz dafür sind die sogenannten *EPSG-Codes*<sup>17</sup>.

### 2094 15.3 Vektordaten in R

2095 Das Paket **sf** stellt Klassen zum Abbilden von Features zur Verfügung, die dann in einem **data.frame** als  
 2096 Liste gespeichert werden können. In der Regel erstellen wir Features nicht individuell, sondern lesen diese aus  
 2097 externen Datenquellen (z.B. ESRI Shapefile) ein. Zum besseren Verständnis, erstellen wir es einmal manuell.  
 2098 Wir haben die Koordinaten für drei Städte (Göttingen, Hannover und Berlin) als geografische Koordinaten  
 2099 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))
```

2100 Daraus können wir jetzt eine Geometriespalte für einen `data.frame` erstellen

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

2101 Somit haben wir die Geometrie in der Variable `geom` gespeichert, aber noch keine dazugehörigen Attributdaten.

2102 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)
)
```

2103 In einem letzten Schritt möchten wir jetzt die Geometrie (`geom`) und die Attributdaten (`attr`) zusammenführen.

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

```
2105 ## Simple feature collection with 3 features and 3 fields
2106 ## Geometry type: POINT
2107 ## Dimension: XY
2108 ## Bounding box: xmin: 9.732 ymin: 51.5413 xmax: 13.405 ymax: 52.52
2109 ## Geodetic CRS: WGS 84
2110 ## name bundesland einwohner geom
2111 ## 1 Goettingen Niedersachsen 119000 POINT (9.9158 51.5413)
2112 ## 2 Hannover Niedersachsen 532000 POINT (9.732 52.3759)
2113 ## 3 Berlin Berlin 3650000 POINT (13.405 52.52)
```

2114 Wir können nun mit `staedte` genau so arbeiten wie mit jedem anderen `data.frame` und die Geometrien  
2115 werden immer ‘berücksichtigt’. Zusätzlich kann man eine Reihe von geometrischen Operationen durchführen.

2116 Wenn ein `data.frame` Punkten hat, kann man dies relative einfach mit der Funktion `st_as_sf()` “räumlich”  
2117 machen. Für das vorherige Beispiel würden wir zuerst einen `data.frame` mit allen Informationen (zur  
2118 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)
)
```

2119 Dann kann man mit der Funktion `st_as_sf()` weiter arbeiten:

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

## 2120 15.4 Arbeiten mit Vektordaten

2121 Es gibt sehr viele Funktionen, um mit räumlichen Daten zu arbeiten, von denen wir hier einige vorstellen.

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

```
2122 ## Coordinate Reference System:
2123 ## User input: EPSG:4326
2124 ## wkt:
2125 ## GEOGCRS["WGS 84",
2126 ## ENSEMBLE["World Geodetic System 1984 ensemble",
2127 ## MEMBER["World Geodetic System 1984 (Transit)"],
2128 ## MEMBER["World Geodetic System 1984 (G730)"],
2129 ## MEMBER["World Geodetic System 1984 (G873)"],
2130 ## MEMBER["World Geodetic System 1984 (G1150)"],
2131 ## MEMBER["World Geodetic System 1984 (G1674)"],
2132 ## MEMBER["World Geodetic System 1984 (G1762)"],
2133 ## MEMBER["World Geodetic System 1984 (G2139)"],
2134 ## ELLIPSOID["WGS 84",6378137,298.257223563,
2135 ## LENGTHUNIT["metre",1]],
2136 ## ENSEMBLEACCURACY[2.0]],
2137 ## PRIMEM["Greenwich",0,
2138 ## ANGLEUNIT["degree",0.0174532925199433]],
2139 ## CS[ellipsoidal,2],
2140 ## AXIS["geodetic latitude (Lat)",north,
2141 ## ORDER[1],
2142 ## ANGLEUNIT["degree",0.0174532925199433]],
2143 ## AXIS["geodetic longitude (Lon)",east,
2144 ## ORDER[2],
2145 ## ANGLEUNIT["degree",0.0174532925199433]],
2146 ## USAGE[
2147 ## SCOPE["Horizontal component of 3D system."],
2148 ## AREA["World."],
2149 ## BBOX[-90,-180,90,180]],
2150 ## ID["EPSG",4326]]
```

2151 Wenn wir jetzt zu einem anderen KBS (z.B. EPSG:3035, ein europäisches projiziertes KBS) umrechnen  
 2152 möchten, können wir das mit

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

```

2153 ## Coordinate Reference System:
2154 ## User input: EPSG:3035
2155 ## wkt:
2156 ## PROJCRS["ETRS89-extended / LAEA Europe",
2157 ## BASEGEOGCRS["ETRS89",
2158 ## ENSEMBLE["European Terrestrial Reference System 1989 ensemble",
2159 ## MEMBER["European Terrestrial Reference Frame 1989"],
2160 ## MEMBER["European Terrestrial Reference Frame 1990"],
2161 ## MEMBER["European Terrestrial Reference Frame 1991"],
2162 ## MEMBER["European Terrestrial Reference Frame 1992"],
2163 ## MEMBER["European Terrestrial Reference Frame 1993"],
2164 ## MEMBER["European Terrestrial Reference Frame 1994"],
2165 ## MEMBER["European Terrestrial Reference Frame 1996"],
2166 ## MEMBER["European Terrestrial Reference Frame 1997"],
2167 ## MEMBER["European Terrestrial Reference Frame 2000"],
2168 ## MEMBER["European Terrestrial Reference Frame 2005"],
2169 ## MEMBER["European Terrestrial Reference Frame 2014"],
2170 ## ELLIPSOID["GRS 1980",6378137,298.257222101,
2171 ## LENGTHUNIT["metre",1]],
2172 ## ENSEMBLEACCURACY[0.1]],
2173 ## PRIMEM["Greenwich",0,
2174 ## ANGLEUNIT["degree",0.0174532925199433]],
2175 ## ID["EPSG",4258],
2176 ## CONVERSION["Europe Equal Area 2001",
2177 ## METHOD["Lambert Azimuthal Equal Area",
2178 ## ID["EPSG",9820]],
2179 ## PARAMETER["Latitude of natural origin",52,
2180 ## ANGLEUNIT["degree",0.0174532925199433],
2181 ## ID["EPSG",8801]],
2182 ## PARAMETER["Longitude of natural origin",10,
2183 ## ANGLEUNIT["degree",0.0174532925199433],
2184 ## ID["EPSG",8802]],
2185 ## PARAMETER["False easting",4321000,
2186 ## LENGTHUNIT["metre",1],
2187 ## ID["EPSG",8806]],
2188 ## PARAMETER["False northing",3210000,
2189 ## LENGTHUNIT["metre",1],
2190 ## ID["EPSG",8807]]],
2191 ## CS[Cartesian,2],
2192 ## AXIS["northing (Y)",north,
2193 ## ORDER[1],
2194 ## LENGTHUNIT["metre",1]],
2195 ## AXIS["easting (X)",east,

```

```

2196 ## ORDER[2] ,
2197 ## LENGTHUNIT["metre",1]],
2198 ## USAGE [
2199 ## SCOPE["Statistical analysis."],
2200 ## AREA["Europe - European Union (EU) countries and candidates. Europe - onshore and offshore: "],
2201 ## BBOX[24.6,-35.58,84.73,44.83]],
2202 ## ID["EPSG",3035]

2203 Die Funktion st_buffer() erlaubt es Features zu puffern, mit st_distance() kann die Distanz zwischen
2204 Features berechnet werden, mit st_area() kann die Fläche eines Features berechnet werden.

2205 Funktionen wie st_intersection(), st_union() und st_difference() erlauben es geometrische Opera-
2206 tionen zwischen unterschiedlichen Features zu berechnen. Für eine ausführliche Diskussion siehe auch hier:
2207 https://geocompr.robinlovelace.net/geometric-operations.html.

2208 Normalerweise lesen wir Daten von externen Datenquellen (z.B. ESRI Shapefiles). Das geht mit der Funktion
2209 st_read().

```

## 2210 15.5 Rasterdaten in R

```

2211 Für Rasterdaten gibt es das R-Paket terra. Auch hier wollen wir uns wieder auf einige Grundfunktionalitäten
2212 konzentrieren. Diese umfassen das Einlesen, Zuschneiden, Rechnen und Abfragen von Rastern.

2213 Mit der Funktion rast() kann ein Raster in R eingelesen werden.

```

```

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

```

```

2214 dem steht für Digital Elevation Model und ist ein Raster mit den Seehöhen in Niedersachsen mit einer
2215 500-m-Auflösung. Wir können diese mit der Funktion res()18 abfragen.

```

```

res(dem)

```

```

2216 ## [1] 500 500

```

```

2217 Bzw. wir können den Raster auch plotten.

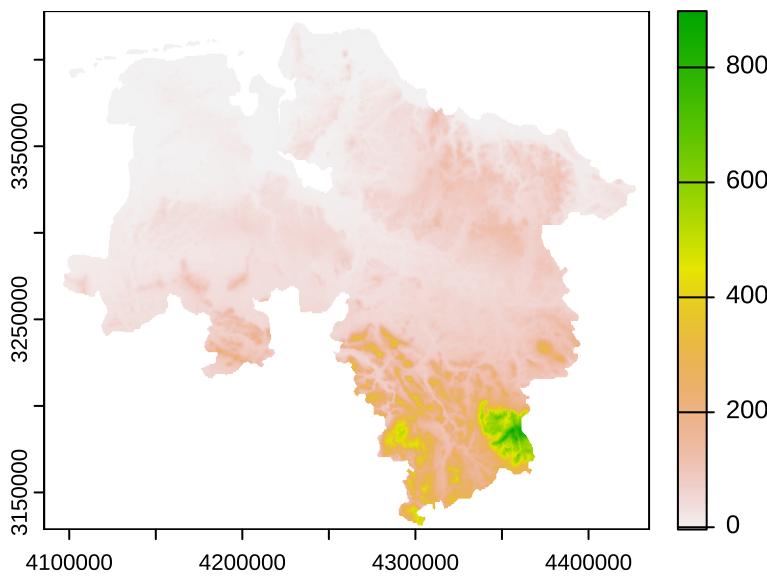
```

```

plot(dem)

```

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



2219 Wenn wir den Raster `dem` auf ein Gebiet zuschneiden wollen (z.B. Göttingen), müssen wir drei Schritte  
2220 durchführen. Als erstes müssen wir ein Shapefile für Göttingen einlesen.

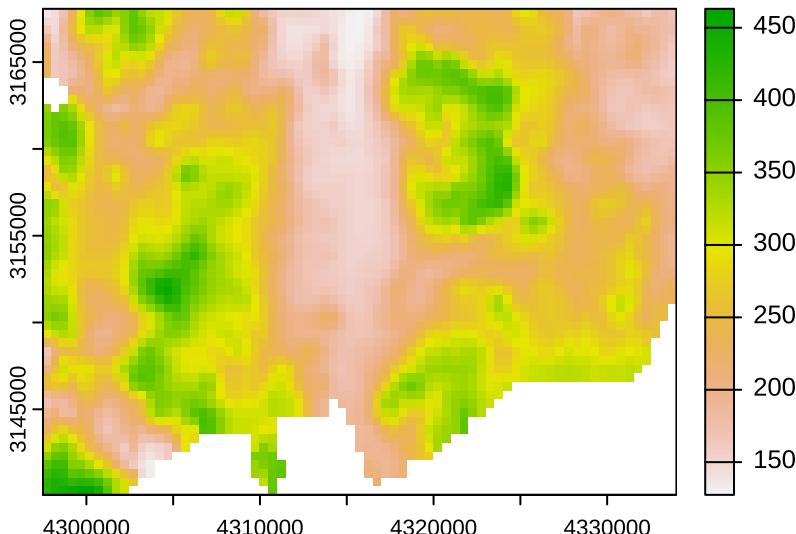
```
goe <- st_read(here::here("data/goettingen/stadt_goettingen.shp"))
```

2221 Dann müssen wir sicherstellen, dass sowohl der Raster `dem` als auch das `sf`-Objekt `goe` im selben KBS sind.  
2222 Es bietet es sich in der Regel an, das KBS des Vektors zu transformieren. Mit der Funktion `projectRaster()`  
2223 kann das KBS eines Rasters transformiert werden.

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

2224 Mit der Funktion `crop()` kann der Raster `dem` auf Göttingen zugeschnitten werden.

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



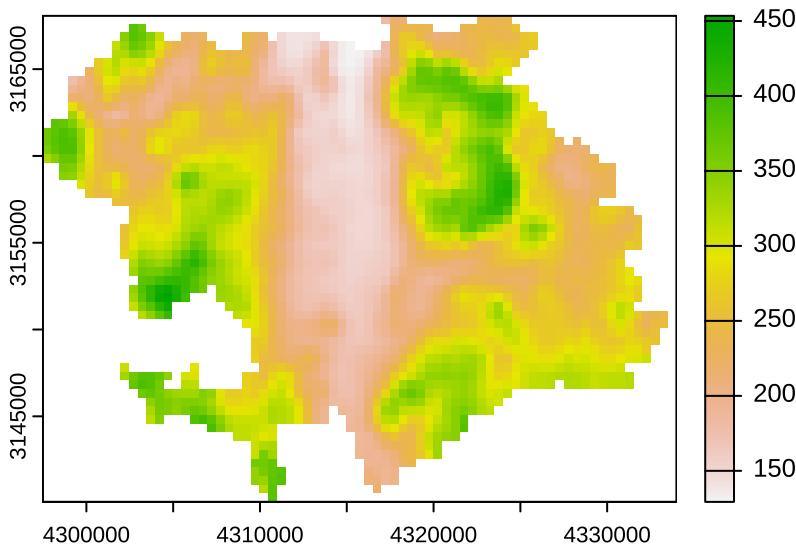
2225 Der Raster hat jetzt die Größe einer Bounding-Box (BBX) von Göttingen (das ist ein Rechteck, das Göttingen  
2226 umfasst). Mit der Funktion `mask()` kann der Raster auf die genauen Grenzen des Vektors `goe` angepasst

2228 werden.

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

2229 ## Warning: [mask] CRS do not match

```
plot(dem2)
```



2230

2231 Wenn wir an bestimmten Punkten den Wert des Rasters abfragen wollen (z.B. an `cities`) von vorhin, dann  
2232 gibt es dafür die Funktion `extract`. Dann müssen wir erst sicherstellen, dass `staedte` und `dem` gleichen KBS  
2233 zu grunde liegt. Dafür transformieren wir einfach `staedte` in das KBS von `dem`. Mit der Funktion `crs()`  
2234 erhalten wir das KBS des Rasters.

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

2235 Wenn wir das KBS eines Objektes nicht kennen, können wir auch einfach das KBS übergeben. Der folgende  
2236 Code-Block macht genau das Gleiche mit dem Vorteil, dass wir keinen EPSG-Code angeben müssen.

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

2237 Dann können wir für jede Stadt die Seehöhe abfragen:

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

2238 ## ID dem\_3035

2239 ## 1 1 149.18181

2240 ## 2 2 57.21486

2241 ## 3 3 NA

2242 Mit `terra::extract()` rufen wir *eindeutig* die Funktion `extract()` aus dem Paket `terra` auf. Wir müssen  
2243 das so machen, weil es im Paket `dplyr` auch eine Funktion `extract()` gibt, die wir hier nicht anwenden  
2244 möchten, da sie einen Fehler verursachen würde.

2245 Ein analoges Vorgehen ist auch für Linien und Polygone möglich.

2246 Mit Rastern kann auch einfach gerechnet werden. Wir können z.B. die Seehöhe in Kilometern anstatt Metern

2247 berechnen:

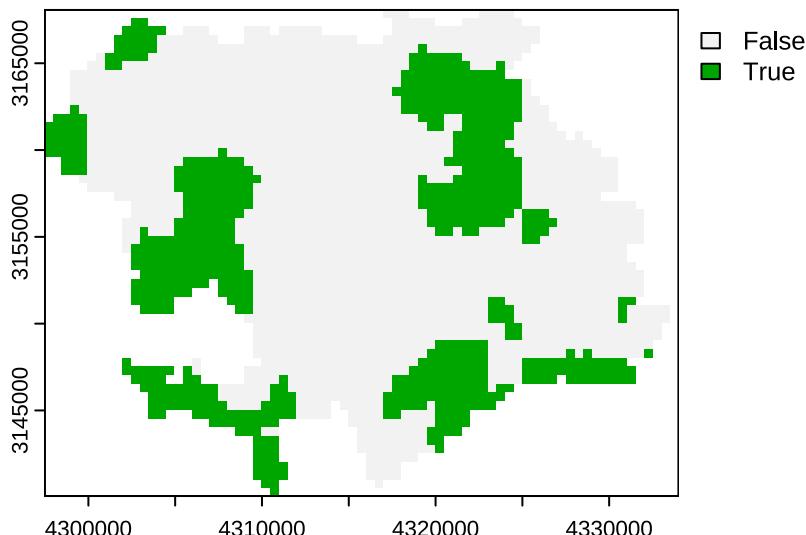
```
dem_km <- dem / 1e3
```

2248 Auch logische Operationen sind möglich, wenn wir alle Rasterzellen mit einer Seehöhe von mehr als 300 m in

2249 Göttingen suchen, dann geht das so:

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

```
plot(dem3)
```



2250

2251 Wenn wir jetzt auf die Werte des Rasters `dem3` zugreifen wollen, geht das mit eckigen Klammern.

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

```
2252 ## dem_3035
2253 ## [1,] NA
2254 ## [2,] NA
2255 ## [3,] NA
2256 ## [4,] NA
2257 ## [5,] NA
2258 ## [6,] NA
```

2259 Das sind erst einmal viele NA-Werte für die ganzen Zellen, die außerhalb von Niedersachsen liegen. Aber wir  
2260 können mit so einem Vektor ganz normal arbeiten und z.B. die Fläche des Landes Niedersachsen die eine  
2261 Seehöhe von mehr als 500m Seehöhe hat ausrechnen.

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

```
2262 ## [1] 0.2786229
```

2263

---

2264 **Aufgabe 41: Arbeiten mit Rastern**

---

2266 Verwenden Sie den Raster `wald.tif`, der auf einer 10 m Auflösung den Waldanteil jeder Rasterzelle angibt<sup>19</sup>.  
 2267 Der EPSG-Code für das KBS von `wald.tif` ist 3035. Nehmen Sie an, dass wenn der Waldanteil in einer  
 2268 Raster größer als 50 % ist, dass die Rasterzelle als Wald klassifiziert werden kann. Wie viel Prozent des  
 2269 Göttinger Stadtgebietes sind Wald? Wie ändert sich dieser Wert, wenn sie 70 % anstatt 50 % als Schwellenwert  
 2270 für Wald annehmen?

2271

---

2272 **Aufgabe 42: Studiendesign**

---

2274 Mit der Funktion `st_sample()` können Sie innerhalb oder entlang eines Features zufällige Punkte legen. Das  
 2275 Argument `n` steuert die Anzahl Punkte und das Argument `type` wie die Punkte angeordnet werden. Für `type`  
 2276 sind für uns die Werte `type = "random"` (komplett zufällig), `type = "regular"` (regelmäßiger Grid) und  
 2277 `type = "hexagonal"` von Bedeutung (ein hexagonaler Grid, d.h. ein sechseckiger Raster). Unglücklicherweise  
 2278 ist das Ergebnis von `st_sample()` erst eine Geometrie. Um daraus ein vollständiges `sf`-Objekt zu machen  
 2279 und problemlos weiter arbeiten zu können, müssen Sie nocheinmal die Funktion `st_as_sf()` ausführen.  
 2280 Stellen Sie sich vor, dass wir die tatsächliche Waldbedeckung des Göttinger Stadtgebietes **nicht** kennen und wir  
 2281 eine Studie durchführen, um den Anteil des Göttinger Stadtgebietes, der mit Wald bedeckt ist herauszufinden.  
 2282 Erstellen Sie dafür einige unterschiedliche Stichproben (diese können in der Anzahl und Anordnung variieren).  
 2283 Berechnen Sie für jedes Stichprobendesign den Anteil an Wald und ein dazugehöriges Konfidenzintervall  
 2284 (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  $n$   
 2285 die Stichprobengröße). Nehmen Sie an, dass eine Rasterzelle Wald ist, sobald > 50 % der Rasterzelle mit  
 2286 Wald bedeckt ist.

2287

---

2288 **Aufgabe 43: Räumliche Daten**

---

2290 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)
)
```

2291 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

- 2292 2. Puffern Sie jeden Baum mit seinem Kronendurchmesser.
- 2293 3. Berechnen Sie die Kronenfläche jedes Baumes. *Hinweis: Die Funktion st\_area() könnte dafür hilfreich sein.*
- 2294
- 2295 4. Welcher Baum hat die größte Kronenfläche?
- 2296 5. Finden Sie für jede Art, den Baum mit der größten Kronenfläche.

2297

2298 **Aufgabe 44: Arbeiten mit räumlichen Daten**

---

- 2300 1. Lesen Sie das ESRI Shapefile goettingen/stadt\_goettingen.shp ein.
- 2301 2. Wie viele Features befinden sich in dem Shapefile?
- 2302 3. Welches Koordinatenbezugssystem (KBS) hat das Shapefile?
- 2303 4. Transformieren Sie das Shapefile in das KBS 3035.
- 2304 5. Erstellen Sie eine neue Spalte A in der Sie die Fläche jeder Gemeinde/Stadt speichern.
- 2305 6. Welche Gemeinde/Stadt (Spalte GEN) ist am größten?
- 2306 7. Wählen Sie nun nur die Stadt Göttingen aus.

2307

2308 **Aufgabe 45: Arbeiten mit räumlichen Daten 2**

---

- 2310 1. Lesen Sie erneut das ESRI Shapefile goettingen/stadt\_goettingen.shp ein.
- 2311 2. Lösen sie die Gemeindegrenzen auf (die Funktion st\_union() könnte hier nützlich sein).
- 2312 3. Wie groß ist das resultierende Feature?

2313 **16 FAQs (Oft gefragtes)**

2314 **16.1 Arbeiten mit Daten**

2315 **16.1.1 Einlesen von Exceldateien**

2316 Mit der Funktion `read_excel()` aus dem Paket `readxl` können Exceldateien direkt in R eingelesen werden.

2317 Ein Export als csv-Datei aus Excel ist nicht notwendig.

## 2318 17 Literatur

- 2319 Ein guter Überblick über viele der angeschnittenen Themen gibt es in dem [R for data science](#), das online frei  
2320 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.  
2321
- 2322 McNamara, Amelia, and Nicholas J Horton. 2018. “Wrangling Categorical Data in r.” *The American Statistician*  
2323 72 (1): 97–104.
- 2324 Wickham, Hadley. 2014. “Tidy Data.” *Journal of Statistical Software* 59 (10). <https://doi.org/10.18637/jss.v059.i10>.  
2325