

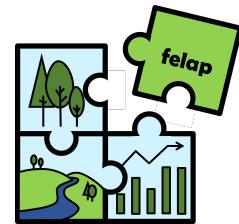
1

Einführung in die Datenanalyse mit R (700104)

4

Kursskript

5



6

Dr. Johannes SIGNER
Abteilung Wildtierwissenschaften
Büsgenweg 3
37077 Göttingen

jsigner@uni-goettingen.de

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

kai.husmann@uni-goettingen.de

7



8

Fakultät für Forstwissenschaften und Waldökologie
Georg-August-Universität Göttingen

10



11

12

Wintersemester 2023/2024

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

¹⁵ Zitiervorschlag:

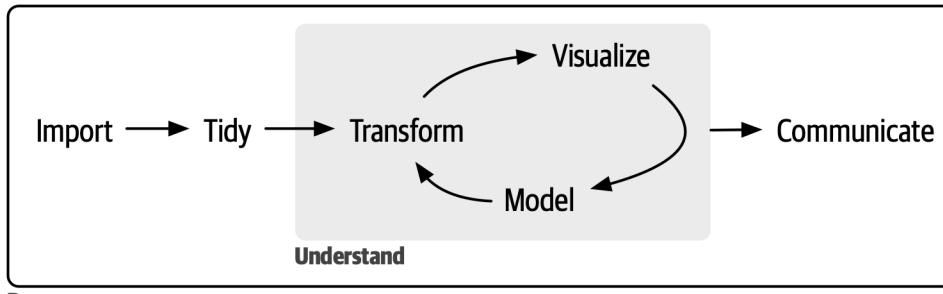
¹⁶ Signer, J. und Husmann, K. (2024) Skript zur Vorlesung Einführung in die Datenanalyse mit R, Georg-August-Universität Göttingen.

¹⁸ Letzte Aktualisierung: 8. April 2024

19 Vorwort und Danksagung

20

- 21 Lernziel des Kurses ist die Einführung in die Arbeit, Visualisierung und Analyse von (forstlichen) Datensätzen
22 mit dem Statistikprogramm *R*. Der Schwerpunkt liegt dabei auf der Datenverarbeitung. Statistische Methoden
23 werden nur an wenigen Stellen exemplarisch angewendet. Ein typisches Data Science Projekt besteht laut
24 Wickham et al. (<https://r4ds.hadley.nz/>) aus 4 Stufen.



25 **Program**

- 26 Wir werden uns in diesem Kurs insbesondere mit den ersten beiden Stufen *Import* und *Tidy* beschäftigen und
27 uns im Schritt *Understand* nur mit sehr einfachen *Models* befassen.
- 28 Weitere Materialien als dieses Kursskript und die Übungsaufgaben (StudIP) werden nicht benötigt. Die
29 gelöste Übungsaufgaben dieses Skriptes werden Ihnen über StudIP zugänglich gemacht. Dort werden auch
30 ggf. Ankündigungen bekanntgegeben. Damit Sie Credits für diesen Kurs zu erhalten, müssen Sie am Ende des
31 Kurses eine mündliche Prüfung ablegen. Für die Prüfung werden Sie zwei zufällig gezogene Prüfungsfragen
32 aus dem Dokument "Übungen: Einführung in die Datenanalyse mit R" bearbeiten und vorstellen. Nach einer
33 15-minütigen Vorbereitungszeit beträgt die Prüfungszeit weitere 15 Minuten. In der Prüfungszeit präsentieren
34 Sie zunächst Ihre Lösung und beantworten anschließend vertiefende Fragen zu Ihrer Lösung und daraufhin
35 auch zum gesamten Lehrinhalt des Kurses.
- 36 Dieses Vorlesungsskript ist ein R Markdown-Dokument, das mit R und RStudio erstellt wurde. Das Dokument
37 besteht aus Fließtext, R Code und den entsprechenden Code-Ergebnissen. Die grau hinterlegten Codepassagen
38 sind kurze R-Skripte. Falls das Skript eine Konsoleausgabe erzeugt, ist diese direkt mit "##" markiert (diese
39 Begriffe werden in Kapitel 1.2 näher erläutert).
- 40 Dank für Anmerkungen gilt Markus Benesch, Sofie Biberacher und Josephine Trisl. Teile des Unterkapitels
41 zu Schleifen und Kontrollstrukturen sind an das R Skript des Kurses Computergestützte Datenanalyse von
42 Robert Nuske, Nikolas von Lüpke und Joachim Saborowski angelehnt. Des Weiteren wurden Beispiele aus
43 dem frei Verfügbaren Dokument *R for Data Science* (<https://r4ds.hadley.nz/intro.html>) entnommen.

44 Inhaltsverzeichnis

45	1 R und RStudio	3
46	1.1 Installation von R und RStudio	3
47	1.2 Erste Schritte in R	3
48	1.3 Gute Praxis bei der Programmierung	5
49	1.4 RStudio Projekte	6
50	1.4.1 Erstellen eines Projektes	6
51	2 Variablen, Funktionen und Datentypen	8
52	2.1 Variablen beim Programmieren	8
53	2.2 Funktionen	9
54	2.3 Datentypen	10
55	2.4 Datenstrukturen	11
56	3 Vektoren	13
57	3.1 Funktionen zum Arbeiten mit Vektoren	15
58	3.2 Statistische Funktionen	16
59	3.3 Beispiel Fotofallen	17
60	3.4 Arbeiten mit logischen Werten	18
61	3.5 Zugreifen auf Elemente eines Vektors (=Untermengen)	19
62	3.6 Der %in%-Operator	21
63	4 Faktoren (factors)	23
64	4.1 Das Paket forcats	24
65	4.1.1 Anpassen der Anordnung von Faktoren	25
66	5 Spezielle Einträge	26
67	5.1 NA	26
68	5.2 NULL	27
69	5.3 Inf	27
70	6 data.frames oder Tabellen	29
71	6.1 Wichtige Funktionen zum Arbeiten mit data.frames	30
72	6.2 Zugreifen auf Elemente eines data.frame	31
73	7 Schreiben und lesen von Daten	34
74	7.1 Textdateien	34
75	8 Erstellen von Abbildungen	36
76	8.1 Base Plot	36
77	8.1.1 Mehrere Panels	42
78	8.1.2 Speichern von Abbildungen	42
79	8.2 Histogramme	43
80	8.3 Boxplots	46

81	8.4 ggplot2: Eine Alternative für Abbildungen	48
82	8.4.1 Multipanel Abbildungen	55
83	8.4.2 Plots kombinieren	57
84	8.4.3 Speichern von plots	60
85	9 Mit Daten arbeiten	61
86	9.1 dplyr eine Einführung	61
87	9.2 Arbeiten mit gruppierten Daten	64
88	9.3 pipes oder %>%	65
89	9.4 Joins	66
90	9.5 'long' and 'wide' Datenformate	68
91	9.6 Auswählen von Variablen	70
92	9.7 Einzelne Beobachtungen abfragen (slice())	71
93	9.8 Spalten trennen	74
94	10 Arbeiten mit Text	76
95	10.1 Arbeiten mit Text	76
96	10.2 Finden von Textmustern	77
97	11 Arbeiten mit Zeit	80
98	11.1 Arbeiten mit Zeitintervallen	81
99	11.2 Formatieren von Zeit	83
100	11.3 Zeitreihen	83
101	12 Aufgaben Wiederholen (for-Schleifen)	89
102	12.1 Schleifen	89
103	12.1.1 Wiederholen von Befehlen mit for()	89
104	12.1.2 Wiederholen von Befehlen mit while()	92
105	12.2 Bedingte Ausführung von Codeblöcken	92
106	13 (R)markdown	94
107	13.1 Markdown Grundlagen	94
108	13.2 R und Markdown	95
109	14 Räumliche Daten in R	97
110	14.1 Was sind räumliche Daten	97
111	14.2 Koordinatenbezugssystem	97
112	14.3 Vektordaten in R	97
113	14.4 Arbeiten mit Vektordaten	99
114	14.5 Rasterdaten in R	101
115	15 FAQs (Oft gefragtes)	107
116	15.1 Arbeiten mit Daten	107
117	15.1.1 Einlesen von Exceldateien	107
118	16 Literatur	108

1 R und RStudio

1.1 Installation von R und RStudio

- Als ersten Schritt müssen Sie R und RStudio installieren. Dabei ist wichtig zu unterscheiden, dass R und RStudio zwei unterschiedliche Programme sind. R ist die eigentliche Programmiersprache mit der wir arbeiten. RStudio hingegen ist eine sogenannte Entwicklungsumgebung¹, die das Arbeiten mit R vereinfachen soll.
- Sie können also mit R arbeiten ohne RStudio (was unüblich ist), aber nicht mit RStudio ohne R.
- Gehen Sie für die Installation von R, auf die Website <https://cloud.r-project.org/> und laden Sie die für ihren Computer passende R-Version herunter und folgen Sie den Installationsanweisungen. In Linux können Sie R über die Kommandozeile installieren.
- Für die Installation von RStudio gehen Sie zu der Website <https://posit.co/download/rstudio-desktop/#download> und laden die richtige Version für Ihr Betriebssystem herunter und folgen Sie den Installationsanweisungen.

1.2 Erste Schritte in R

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

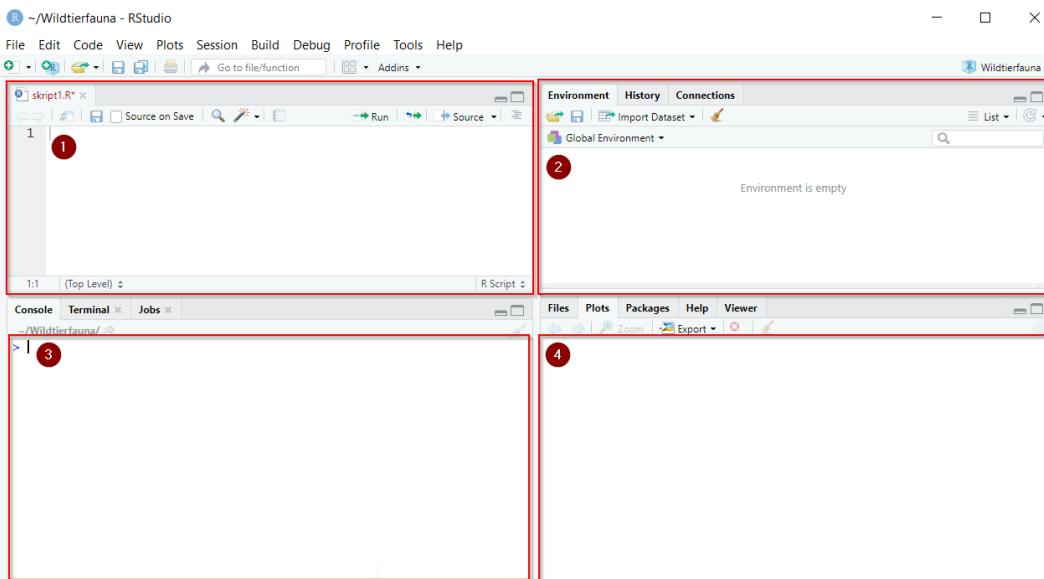


Abbildung 1: RStudio Panes.

- RStudio besteht nun aus vier sogenannten **Panes** oder Ausschnitten (siehe auch Abbildung 1). Die Ausschnitte sind wie folgt gegliedert:
1. Hier werden Skripte angezeigt, d.h., hier wird meist R Code geschrieben und dokumentiert. Der Code wird

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

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

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

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

10 + 5

153 **## [1] 15**

20 - 10

154 **## [1] 10**

10 * 3

155 **## [1] 30**

100 / 19

156 **## [1] 5.263158**

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

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

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

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

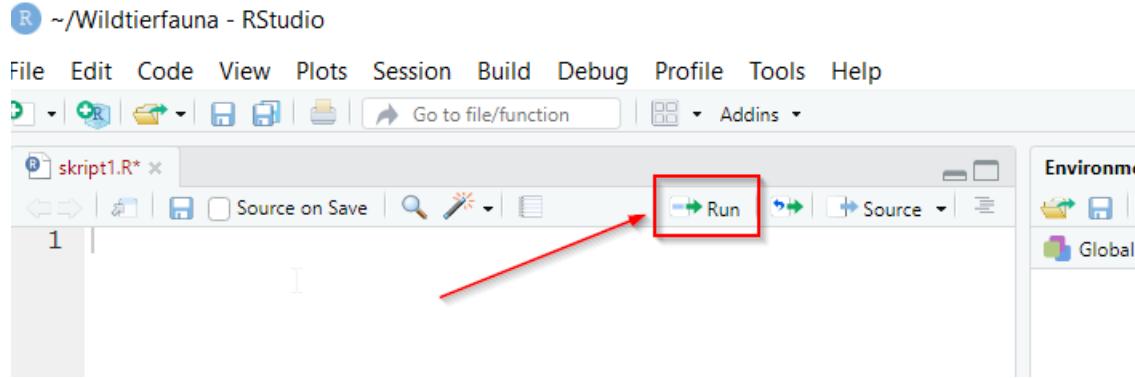


Abbildung 2: Zeilenweises Ausführen von Code in RStudio.

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

180 1.3 Gute Praxis bei der Programmierung

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

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

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

## [1] 9
```

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

202

203 **Aufgabe 1: Ausführen von Quellcodes**

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

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

# Einfache Funktion
sqrt(20)
```

207 Führen Sie nun alle Zeilen aus.

208 **1.4 RStudio Projekte**

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

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

214 **1.4.1 Erstellen eines Projektes**

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

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

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

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

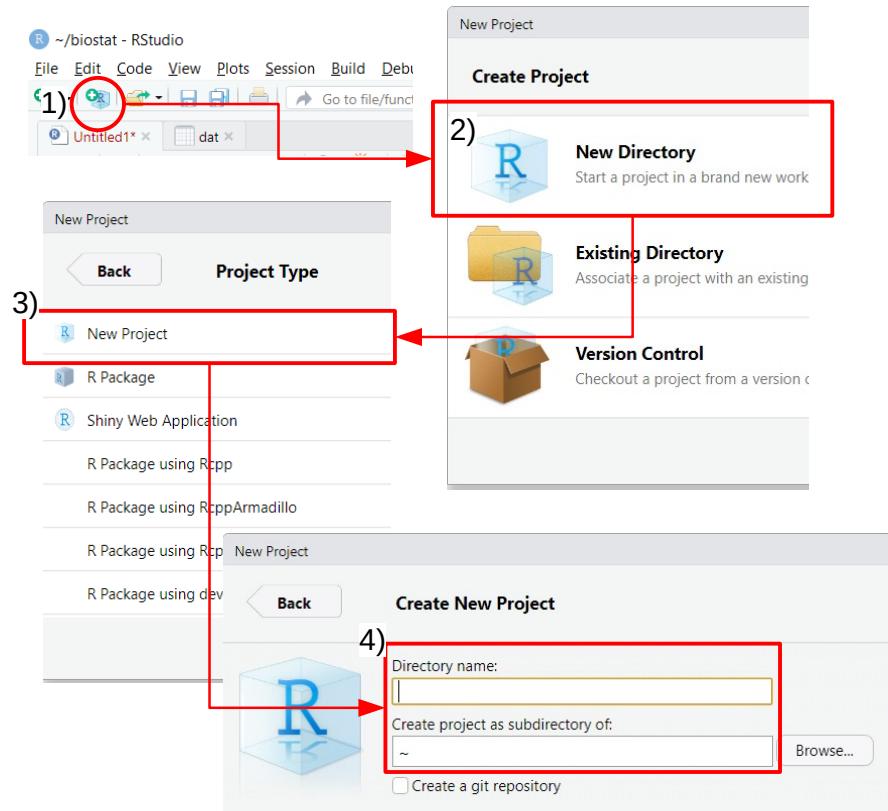


Abbildung 3: Workflow zum Erstellen eines Projekts.

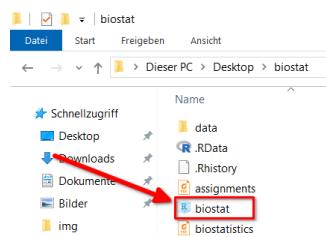


Abbildung 4: Öffnen eines RStudio Projekts.



Abbildung 5: Öffnen von Projekten.

2 Variablen, Funktionen und Datentypen

2.1 Variablen beim Programmieren

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

```
alter <- 102
```

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

```
a <- 10
a
```

```
## [1] 10
```

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

Wir können beliebige Variablen erstellen, z.B.

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

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

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

Vorsicht: Groß- und Kleinschreibung muss beachtet werden.

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

```
## [1] "Johannes"
```

Das Aufrufen der Variable

```
Name
```

führt zu einem Fehler.

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

```
a <- 10
b <- 5

a + b
```

```

249 ## [1] 15
b / a

250 ## [1] 0.5
a^b

251 ## [1] 1e+05

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

253 ## [1] 15
ergebnis2 <- ergebnis * 2
ergebnis2

254 ## [1] 30

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

258 ## [1] TRUE
rm(var1)
exists("var1") # FALSE, also nein, eine Variable mit diesem Namen existiert nicht.

259 ## [1] FALSE

260 2.2 Funktionen

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

263 ## [1] 3.162278

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

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

```

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

```
273 sqrt(x = a)
274 ## [1] 3.162278
275
276 Um mehr über eine Funktion zu erfahren (z.B. die Bedeutung von Argumenten zu verstehen oder heraus-
277 zufinden, was eine Funktion zurück gibt), können wir die Hilfeseiten konsultieren. Es gibt unterschiedliche
278 Wege, um zu einer Hilfeseite zu gelangen.
279
280 1. In die Konsole ?<Name der Funktion> tippen. Also, wenn wir Hilfe für die Funktion mean() möchten,
281 könnten wir einfach ?mean in die Konsole tippen.
282 2. Analog zu 1), können wir mit der Funktion help die Hilfeseite für eine andere Funktion aufrufen (z.B.
283 wenn wir wieder die Hilfe für die Funktion mean() lesen möchten, dann könnten wir auch help(mean)
284 in die Konsole tippen).
285 3. In RStudio kann man auch auf das Help-Panel klicken und dann einfach eine Funktion suchen (siehe
286 Abbildung 1).
287 4. Wenn der Cursor in RStudio auf einer Funktion ist, kann man mit der Taste F1 die dazugehörige
288 Hilfeseite aufrufen.
```

286 2.3 Datentypen

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

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

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

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

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

```
fuchs_gesehen <- TRUE
```

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

```
typeof(fuchs_gesehen)
```

305 ## [1] "logical"

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

```
TRUE + TRUE
```

307 ## [1] 2

```
FALSE + FALSE
```

308 ## [1] 0

```
TRUE + FALSE
```

309 ## [1] 1

310 2.4 Datenstrukturen

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

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

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

314 129, 91, 138, 144, 55, 103, 139, 105, 96, 146, 95, 118, 107.

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

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

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

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

319 Wert ein einzelnes Objekt mit je einem Wert erstellen. Dass das zu Aufwändig wäre, leuchtet unmittelbar ein:

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

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

# usw.
```

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

321 zu einem unübersichtlichen *Workspace*³. Wir werden im Verlauf sinnvollere Objekte (Vektoren oder Data

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

322 Frames) für diesen Zweck kennenlernen.

323

324 **Aufgabe 2: Variablen**

325 326 Verwenden Sie die folgenden Daten

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

327 und berechnen sie:

- 328 • $10 * a$
329 • $a / 144$ und speichern Sie das Ergebnis in einer neuen Variablen e zwischen.
330 • Was ist das Ergebnis von $a + b$?
331 • Was ist das Ergebnis von $a + p$?

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

3 Vektoren

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

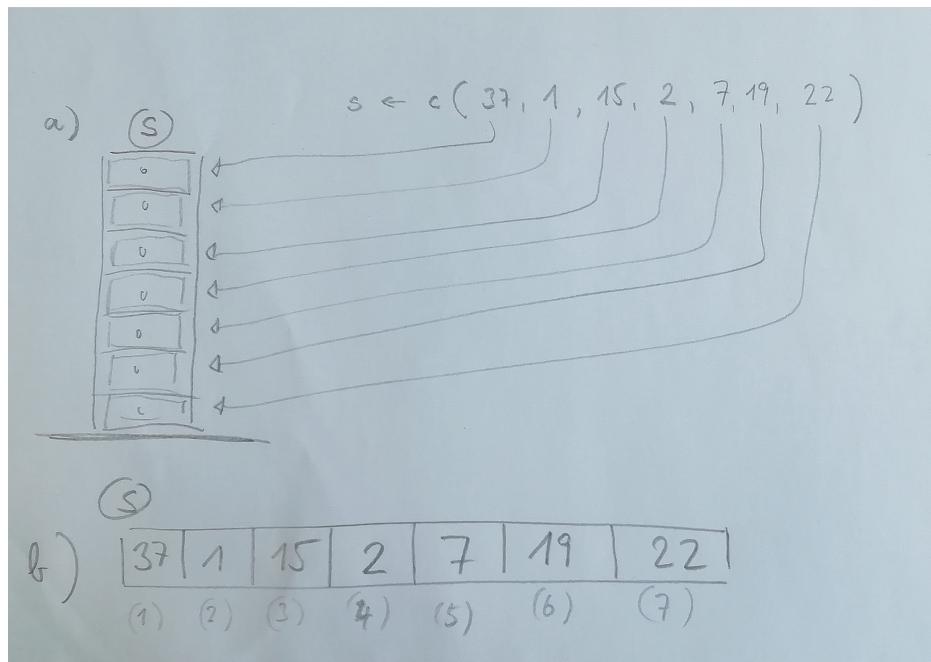


Abbildung 6: Schematische Darstellung eines Vektors in R.

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

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

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

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

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

s

352 ## [1] 37 1 15 2 7 19 22

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

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

s + 10

358 ## [1] 47 11 25 12 17 29 32

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

s * s

363 ## [1] 1369 1 225 4 49 361 484

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

seq(from = 1, to = 10)

367 ## [1] 1 2 3 4 5 6 7 8 9 10

(1 : 10)

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

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

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

370 ## [1] 1 3 5 7 9

371

372 Aufgabe 3: Vektoren erstellen

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

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

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

3.1 Funktionen zum Arbeiten mit Vektoren

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

```
head(s)
```

381 ## [1] 37 1 15 2 7 19

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

382 ## [1] 37 1 15

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

383 ## [1] 19 22

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

```
length(s)
```

385 ## [1] 7

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

```
class(s)
```

387 ## [1] "numeric"

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

```
unique(s)
```

389 ## [1] 37 1 15 2 7 19 22

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

```
table(s)
```

391 ## s

392 ## 1 2 7 15 19 22 37

393 ## 1 1 1 1 1 1 1

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

```
rev(s)
```

396 ## [1] 22 19 7 2 15 1 37

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

```
sort(s)
```

398 ## [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.

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

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

400 `## [1] 37 1 15 2 7 19 22 37 1 15 2 7 19 22`

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

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

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

403 `## [1] 1 2 3 4 1 2 3 4`

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

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

405

406 *Aufgabe 4: Arbeiten mit Vektoren*

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

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

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

411 3.2 Statistische Funktionen

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

```
mean(s)
```

415 `## [1] 14.71429`

```
median(s)
```

416 `## [1] 15`

```
sd(s)
```

417 `## [1] 12.76341`

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

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

421 `## [1] 1`

```

sample(s, size = 3) # 2 Elemente

422 ## [1] 15 7 22
423 Wenn size weggelassen wird, dann bekommt man gleich viele Elemente zurück (wie der Vektor lang ist), d.h.
424 der Vektor wird nur permutiert.

```

425 3.3 Beispiel Fotofallen

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

429 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)

```

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

```

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

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

```

v1 <- rep("Kamera", 15)

```

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

```

v2 <- 1:15

```

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

```

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

443 ## [1] "Kamera_1"   "Kamera_2"   "Kamera_3"   "Kamera_4"   "Kamera_5"   "Kamera_6"
444 ## [7] "Kamera_7"   "Kamera_8"   "Kamera_9"   "Kamera_10"  "Kamera_11"  "Kamera_12"
445 ## [13] "Kamera_13"  "Kamera_14"  "Kamera_15"

```

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

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

```
448 ## [1] "Revier A" "Revier B" "Revier C" "Revier A" "Revier B" "Revier C"  

449 ## [7] "Revier A" "Revier B" "Revier C" "Revier A" "Revier B" "Revier C"  

450 ## [13] "Revier A" "Revier B" "Revier C"
```

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

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

reviere
```

```
453 ## [1] "Revier A" "Revier A" "Revier A" "Revier A" "Revier A" "Revier B"  

454 ## [7] "Revier B" "Revier B" "Revier B" "Revier B" "Revier C" "Revier C"  

455 ## [13] "Revier C" "Revier C" "Revier C"
```

456

457 Aufgabe 5: Statistische Funktionen

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

460 2. Erstellen Sie die folgende Konsolenausgabe:

```
461 ## [1] "Die mittlere Anzahl von Rehfotos beträgt 171.8 Rehe pro Standort."
```

462 3.4 Arbeiten mit logischen Werten

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

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

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

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

```
anzahl_rehe > 100
```

```
472 ## [1] TRUE FALSE TRUE FALSE TRUE TRUE FALSE TRUE TRUE FALSE TRUE  

473 ## [13] FALSE TRUE TRUE
```

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

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

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

```
476 ## [1] FALSE FALSE FALSE FALSE FALSE TRUE TRUE TRUE TRUE TRUE FALSE FALSE
477 ## [13] FALSE FALSE FALSE
```

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

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

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

```
484 ## [1] FALSE FALSE FALSE FALSE FALSE TRUE FALSE TRUE TRUE TRUE FALSE FALSE
485 ## [13] FALSE FALSE FALSE
```

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

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

```
489 ## [1] TRUE FALSE TRUE FALSE TRUE TRUE TRUE TRUE TRUE TRUE FALSE TRUE
490 ## [13] FALSE TRUE TRUE
```

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

493

Aufgabe 6: Arbeiten mit logischen Werten

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

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

3.5 Zugreifen auf Elemente eines Vektors (=Untermengen)

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

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

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

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

```
anzahl_rehe[2]
```

519 ## [1] 79

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

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

521 ## [1] 132 79 129 91 138

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

522 ## [1] 132 79 129 91 138

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

525

526 Aufgabe 7: Zugreifen auf Vektorelemente

528 Erstellen Sie einen neuen Vektor `bhd`

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

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

532

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

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

535 ## [1] 132 79 129 91 138

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

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

539 ## [1] 132 79 129 91 138

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

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

542 ## [1] 132 79 129 91 138

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

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

545 ## [1] 113.8

546

547 Aufgabe 8: logische Werte

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

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

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

554 3.6 Der %in%-Operator

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

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

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

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

560 `## [1] "FI" "FI"`

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

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

563 `## [1] "FI" "BU" "BU" "FI"`

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

```
messungen_arten %in% arten
```

567 `## [1] TRUE TRUE TRUE FALSE FALSE FALSE TRUE FALSE FALSE`

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

568 `## [1] "FI" "BU" "BU" "FI"`

569

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

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

```
LETTERS
```

573 `## [1] "A" "B" "C" "D" "E" "F" "G" "H" "I" "J" "K" "L" "M" "N" "O" "P" "Q" "R" "S"`

574 `## [20] "T" "U" "V" "W" "X" "Y" "Z"`

575 Wählen Sie aus LETTERS nur die Vokale aus.

576 4 Faktoren (factors)

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

583 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)
```

584 ## [1] FI BU FI EI EI FI FI
 585 ## Levels: BU EI FI

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

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

589 ## [1] FI BU FI EI EI FI FI
 590 ## Levels: FI BU EI

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

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

593 ## [1] Fichte Buche Fichte Eiche Eiche Fichte Fichte
 594 ## Levels: Fichte Buche Eiche

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

```
levels(af)

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

598 ## [1] Fi Bu Fi Ei Ei Fi Fi
 599 ## Levels: Fi Bu Ei

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

```
af
```

```
602 ## [1] Fi Bu Fi Ei Ei Fi Fi
603 ## Levels: Fi Bu Ei
  relevel(af, "Bu")
```

```
604 ## [1] Fi Bu Fi Ei Ei Fi Fi
605 ## Levels: Bu Fi Ei
```

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

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

```
608 ## [1] "Fi" "Bu" "Fi" "Ei" "Ei" "Fi" "Fi"
```

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

```
af
```

```
610 ## [1] Fi Bu Fi Ei Ei Fi Fi
611 ## Levels: Fi Bu Ei
```

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

```
612 ## [1] 1 2 1 3 3 1 1
```

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

615

616 **Aufgabe 10: Faktoren**

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

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

620 4.1 Das Paket **forcats**

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

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

626 4.1.1 Anpassen der Anordnung von Faktoren

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

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

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

```
factor(a)
```

629 ## [1] FI BU FI EI EI FI FI

630 ## Levels: BU EI FI

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

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

```
f1
```

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

633 ## Levels: FI BU EI

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

```
fct_rev(f1)
```

635 ## [1] FI BU FI EI EI FI FI

636 ## Levels: EI BU FI

637 nach der Häufigkeit zu sortieren oder

```
fct_infreq(f1)
```

638 ## [1] FI BU FI EI EI FI FI

639 ## Levels: FI EI BU

640 eine zufällige Sortierung.

```
fct_shuffle(f1)
```

641 ## [1] FI BU FI EI EI FI FI

642 ## Levels: EI FI BU

5 Spezielle Einträge

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

- 645 • fehlenden Einträgen NA,
- 646 • leeren Einträgen NULL,
- 647 • undefinierten Einträgen NaN (Not a Number) oder
- 648 • unendlichen Zahlen (Inf).

649 Spezielle Einträge sind reservierte Namen. Sie können nicht überspeichert werden.

650 5.1 NA

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

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

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

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

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

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

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

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

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

1 + NA

669 ## [1] NA
670 Viele R Funktionen haben eingebaute Methoden zum Umgang mit NA. Die Funktion `mean()` bspw. ergibt
671 (wie die meisten Funktionen) standardmäßig NA wenn sie auf Vektoren mit Datenlücken angewendet wird, es
672 sei denn man stellt innerhalb der Funktion ein, dass Datenlücken entfernt werden sollen.

`mean(na2)`

673 ## [1] NA
674
675 mean(na2, na.rm = TRUE)
674 ## [1] 4.5

675 5.2 NULL

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

680 5.3 Inf

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

`10^309`

683 ## [1] Inf
684 ## [1] Inf
685 ## [1] Inf
686 ## [1] Inf
687 ## [1] -Inf

`3 / 0`

688 ## [1] 0

689 Infinity kann mit `is.infinite` und `is.finite` getestet werden. Relationäre Operatoren funktionieren
690 erwartungsgemäß.

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

691 ## [1] TRUE FALSE FALSE TRUE FALSE
is.finite(inf1)

692 ## [1] FALSE TRUE TRUE FALSE TRUE
inf1 < 3

693 ## [1] FALSE TRUE FALSE TRUE FALSE
```

694

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

697 Verwenden Sie den Vektor

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

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

700 Verwenden Sie den Vektor

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

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

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

706 Berechnen Sie den arithmetischen Mittelwert von `foo`.

707 6 data.frames oder Tabellen

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

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

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

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

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

741 Standard-Objekte zum Speichern wissenschaftlicher Daten.

742 6.1 Wichtige Funktionen zum Arbeiten mit `data.frames`

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

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

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

748 Oder für die letzten 2 Beobachtungen.

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

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

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

```
nrow(monitoring)
```

```
753 ## [1] 15
```

```
ncol(monitoring)
```

```
754 ## [1] 3
```

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

```
str(monitoring)
```

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

761

762 Aufgabe 12: `data.frame`

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

6.2 Zugreifen auf Elemente eines `data.frame`

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```
798 ## 4 Kamera_4          91 Revier A
799 ## 5 Kamera_5          138 Revier A
```

800

801 Aufgabe 13: Abfragen von Werten

803 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)
)
```

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

807

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

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

```
monitoring$anzahl_rehe
```

813 ## [1] 132 79 129 91 138 144 55 103 139 105 96 146 95 118 107

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

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

815 ## [1] 132 79 129 91 138 144 55 103 139 105 96 146 95 118 107

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

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

817 ## [1] 132 79 129 91 138 144 55 103 139 105 96 146 95 118 107

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

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

```
monitoring$anzahl_rehe > 100
```

```
825 ## [1] TRUE FALSE TRUE FALSE TRUE TRUE FALSE TRUE TRUE TRUE FALSE TRUE  
826 ## [13] FALSE TRUE TRUE
```

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

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

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

842

843 Aufgabe 14: Abfragen von Werten 2

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

- ```
845
846 • Alle Spalten für Studierende die Forstwissenschaften studieren.
847 • Alle Spalten für Studierende die Chemie oder Physik studieren.
848 • Die Spalte fach und semester für Studierende die 22 oder älter sind.
```

## 849 7 Schreiben und lesen von Daten

### 850 7.1 Textdateien

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

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

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

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

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

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

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

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

885

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

---

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

## 8 Erstellen von Abbildungen

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

### 8.1 Base Plot

894 Die wichtigsten Grafiken für die einfache Datendarstellung sind schnell verfügbar. Etwas komplexere oder  
 895 spezielle Grafiken erfordern mehr Programmieraufwand (folgt teilweise noch). Für viele komplexere Diagramme  
 896 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  
 897 sie sich die einfache Grafik Schnittstelle (**base plots**) als zweidimensionale Leinwand vor, auf die Sie durch  
 898 Code Ebene für Ebene Grafikelemente legen:

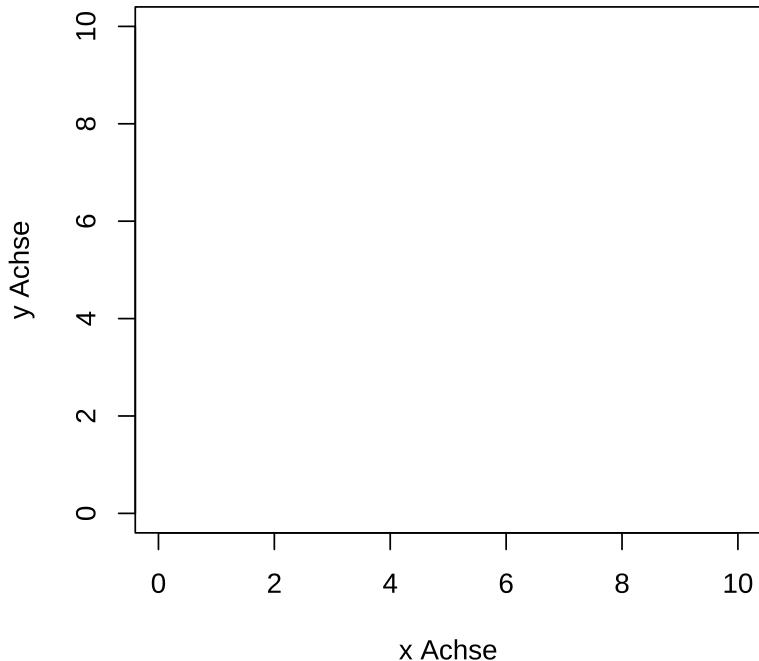
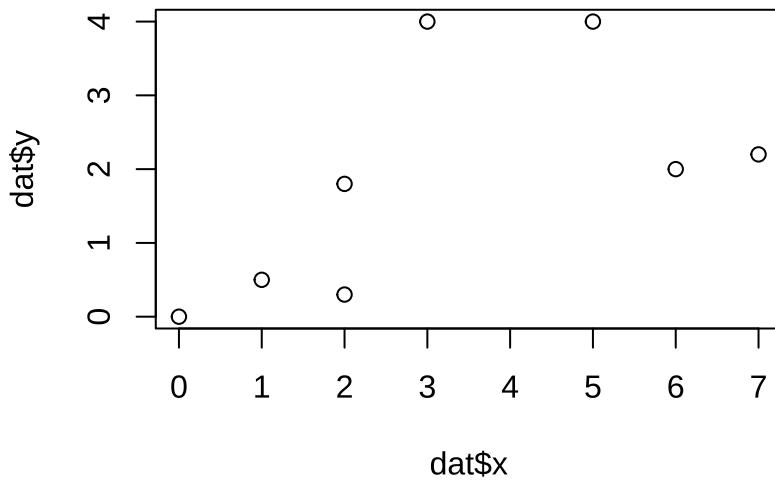


Abbildung 7: Beispiel einer leeren Grafikschnittstelle.

903 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)
)

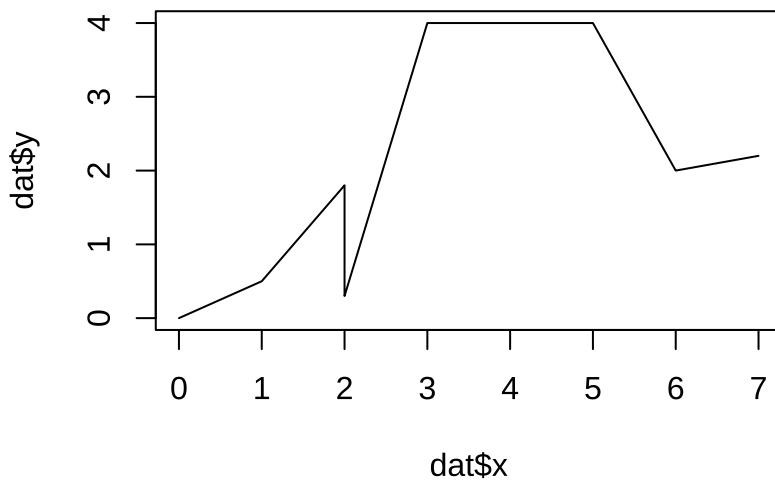
plot(datx, daty, type ="p")
```



904

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

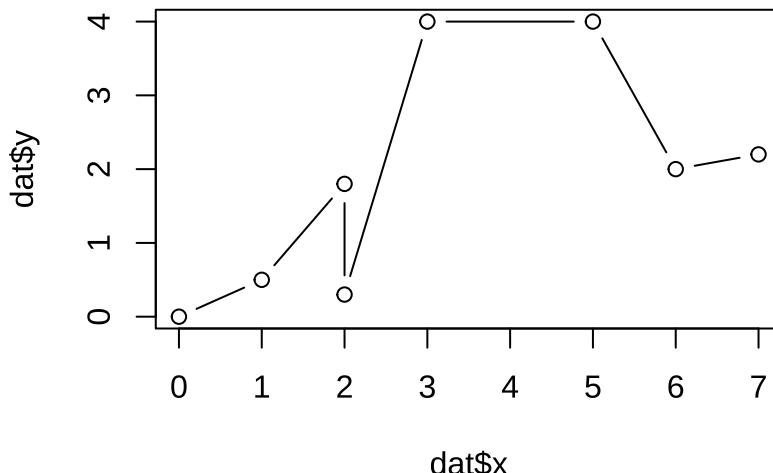
```
plot(datx, daty, type = "l")
```



907

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

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



909

910 darstellen.

911

912 **Aufgabe 16: Base Plot 1**

913

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

916

917 Sie können entweder eine Grafik mit einem Befehl erzeugen (High-Level) oder die einzelnen Ebenen nacheinander erzeugen (Low-Level). Sie können jeder Ebene durch zusätzliche Befehle innerhalb des Funktionsaufrufs  
918 Elemente hinzufügen. Mit jeder neuen Ebene können Sie die vorigen Ebenen jedoch nicht mehr verändern.  
919  
920 Die wichtigsten Argumente der `plot` Funktion sind:

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

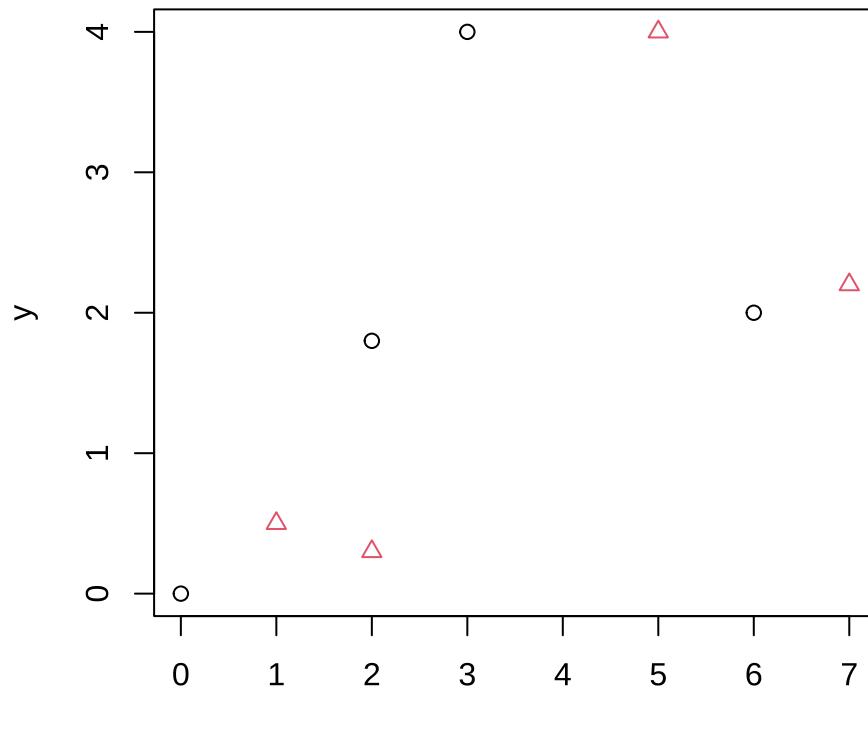
933 Sehen Sie sich die Hilfeseiten `?plot.default()` oder `?par()` an für weitere Informationen. Dort finden Sie  
934 auch eine vollständige Liste der Befehle. Einige Argumente können als Vektor übergeben werden. Hier z. B.  
935 die Farben und die Punktsymbole.

```

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")

```



936

x

937

---

**Aufgabe 17: Anpassen von Plots**


---

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

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

945

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

947 Die wichtigsten Funktionen sind

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

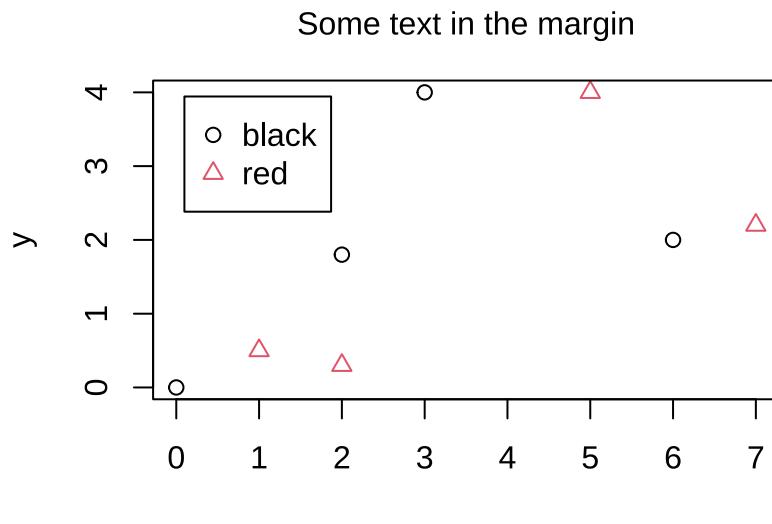
- 950 • `text()` - Fügt Text ein  
 951 • `mtext` - Fügt Text in den Rahmen (`margin`) ein  
 952 • `legend()` - Fügt eine Legende ein  
 953 • `abline()` - Fügt eine Gerade ein  
 954 • `curve()` - Fügt eine mathematische Funktion ein  
 955 • `arrows()` - Fügt Pfeile ein  
 956 • `grid()` - Fügt Hilfslinien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

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

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



963  
 964 Mit diesem grundsätzlichen Aufbau sollten Sie bereits in der Lage sein auch komplexe Grafiken schnell zu  
 965 gestalten. Wenn Sie mehrere Diagramme in einem Plot arrangieren möchten, können Sie mit dem `par()`  
 966 Befehl ein Arrangement definieren. Sie haben dann zusätzlich zu den bereits bekannten Grafikregionen noch  
 967 äußere Ränder (`outer margins`). Siehe Abbildung 9.

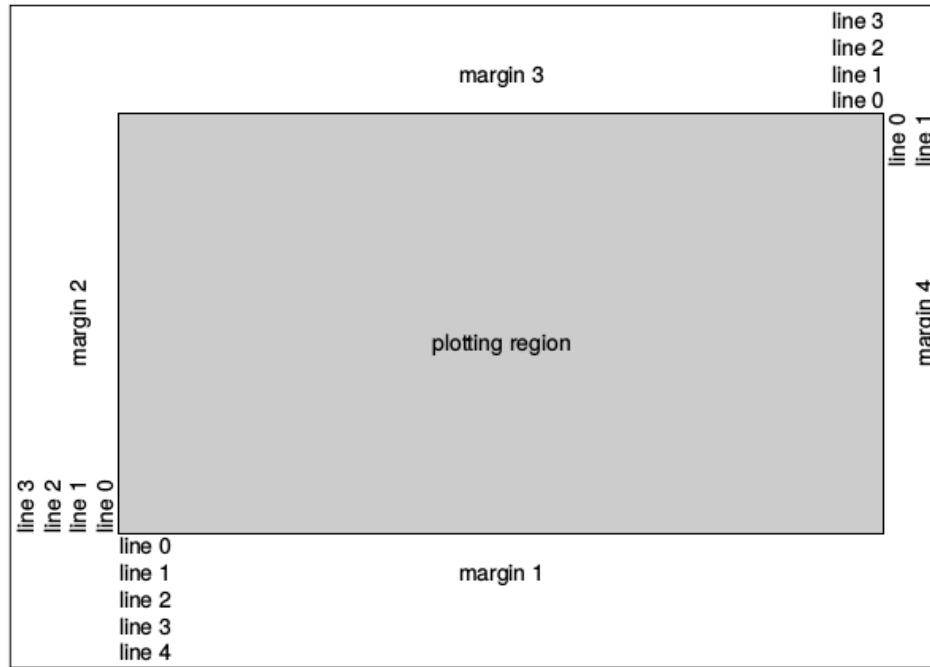
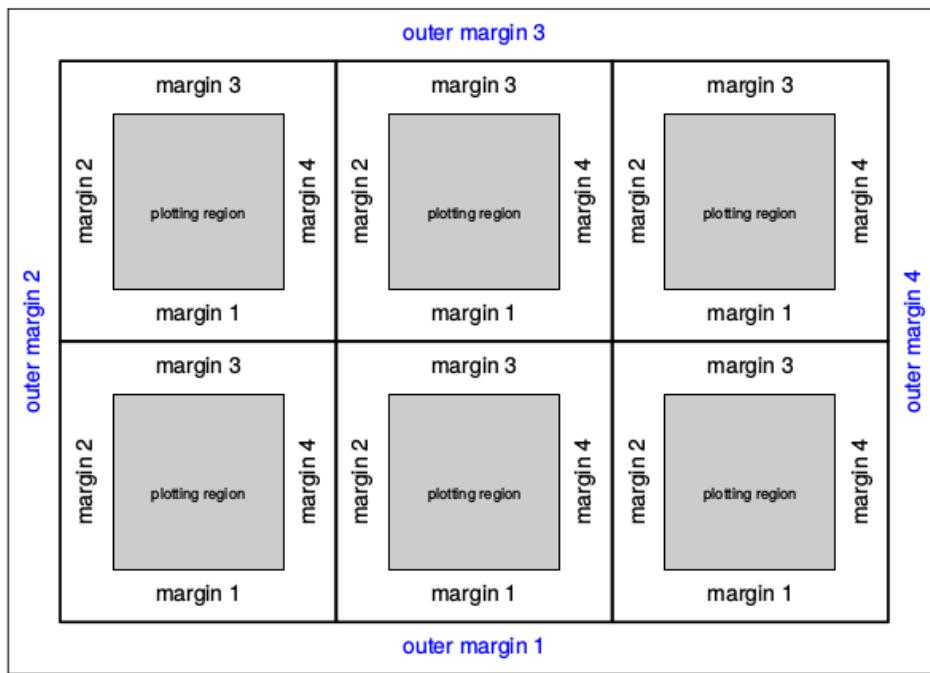


Abbildung 8: Grafikregionen eines base plots in R.

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

968 **8.1.1 Mehrere Panels**

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

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

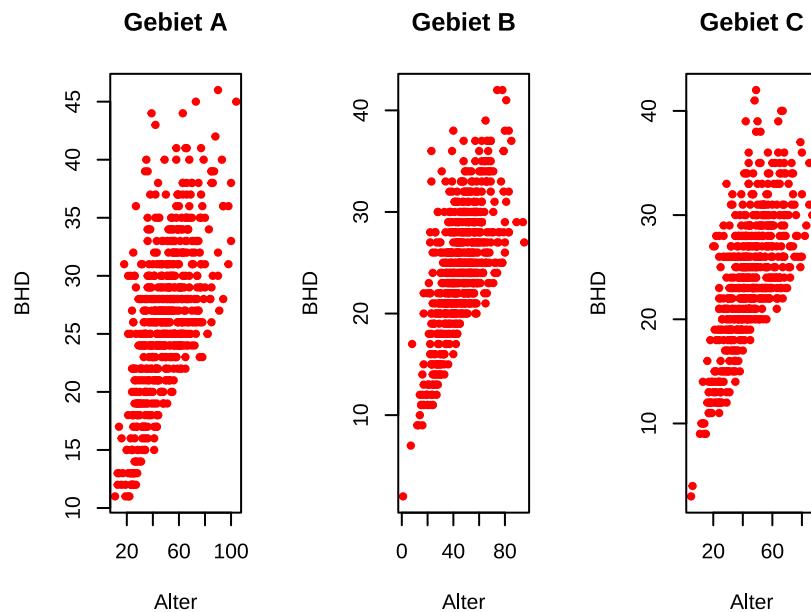
972 Teilt den Plot in eine Zeile und drei Zeilen (= drei Plots nebeneinander).

```
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")
```



973

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

976 **8.1.2 Speichern von Abbildungen**

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

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

983 Beispiele für Rastergrafiken sind

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

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

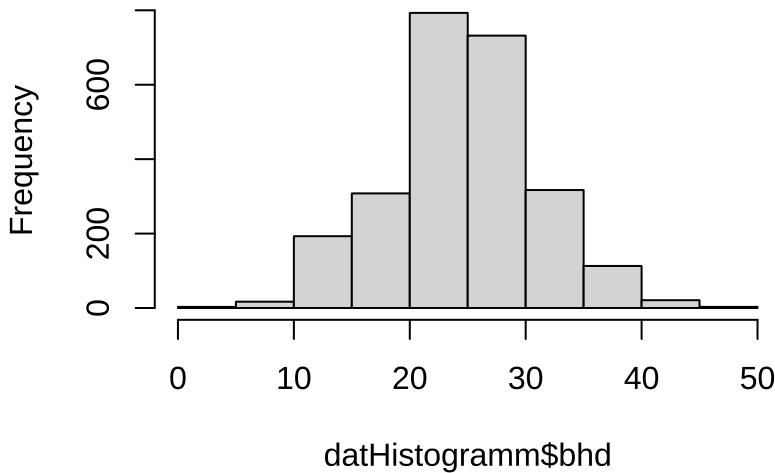
990 *Achtung*, wenn Sie die Funktion `dev.off()` nicht aufrufen, werden alle nachfolgenden Plots in die gleiche  
 991 Datei geschrieben. Falls Sie nach einem Versuch einen Plot zu speichern plötzlich keine weiteren Plots mehr  
 992 sehen, führen Sie einige Mal die Funktion `dev.off()` aus.

## 993 8.2 Histogramme

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

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

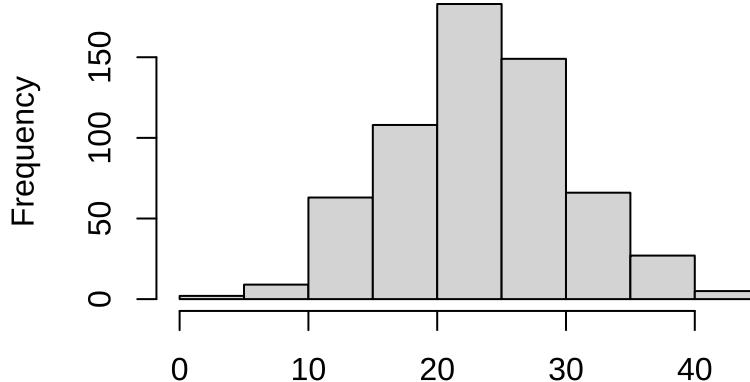
### Histogramm of datHistogramm\$bhd



1001

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

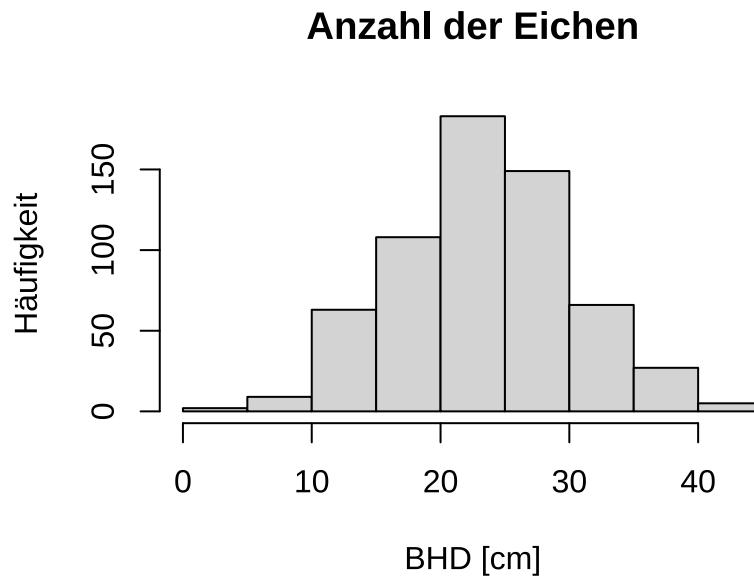
### Histogramm of datHistogramm\$bhd[datHistogramm\$art == "EI"]



1002

```
datHistogramm$bhd[datHistogramm$art == "EI"]
```

```
Nur für Eichen, geänderte High-Level Einstellungen
hist(datHistogramm$bhd[datHistogramm$art == "EI"],
 xlab = "BHD [cm]", ylab = "Häufigkeit",
 main = "Anzahl der Eichen")
```

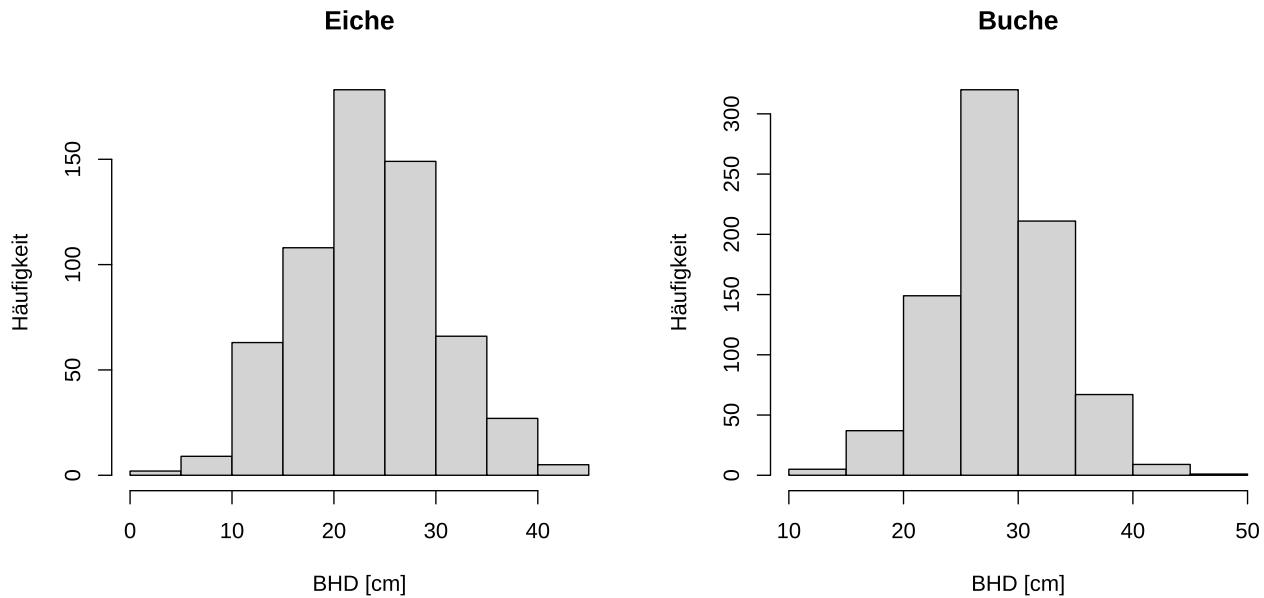


1003

1004 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")
```

1005



1006

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

### 1007 8.3 Boxplots

1008 Oft möchte man die Verteilung einer stetigen Variablen in Abhängigkeit einer diskreten Variable beschreiben  
 1009 oder visualisieren. Ein Beispiel dafür wäre die BHD-Verteilung in Abhängigkeit der Baumarten. Eine häufige  
 1010 Darstellungsform für solche Daten sind *Boxplots*. Sie stellen die durchschnittliche Ausprägung einer stetigen  
 1011 Variable und ihre Schwankung kompakt dar.

1012 Boxplots bestehen aus drei Komponenten:

- 1013 1. Eine *Box*, die den Bereich zwischen 0.25 und 0.75 Percentil abdeckt, diese Distanz wird auch die *IQR*  
 1014 (Interquartile Range), bezeichnet. Zusätzlich wird die Box durch den Median (als dicke horizontale Linie)  
 1015 unterteilt.
- 1016 2. Einzelne Punkte Ausreißer. Als Ausreißer werden Punkte bezeichnet, die  $> 1.5IQR$  vom unteren oder  
 1017 oberen Ende der Box entfernt sind.
- 1018 3. Eine senkrechte Linie von jeder Seite der Box bis zum letzten "Nicht-Ausreißer-Punkt". Also der letzte  
 1019 Punkt, der  $> 1.5IQR$  aber nicht  $> 0.75$  bzw.  $< 0.25$  Percentil ist. Diese Linie wird auch als *Whisker*  
 1020 bezeichnet.

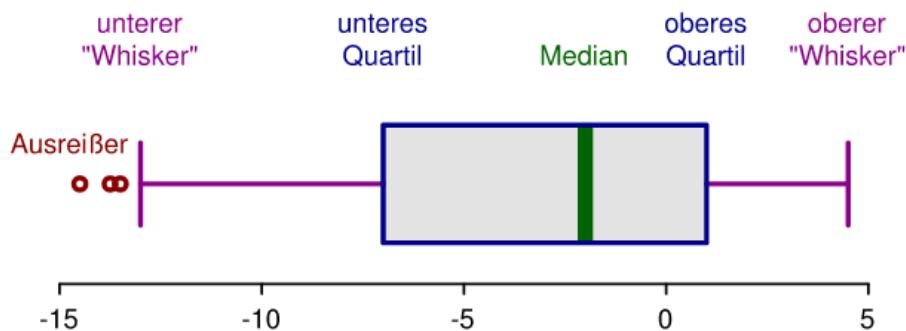
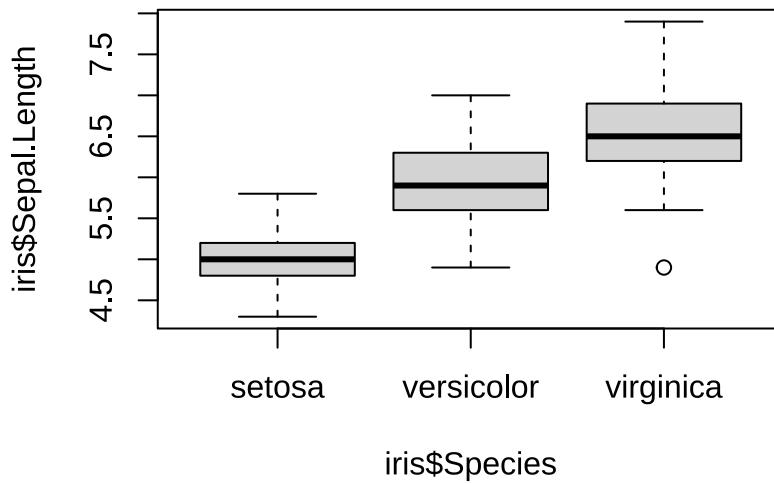


Abbildung 10: Schematische Darstellung eines Boxplots (Quelle: Von RobSeb - Eigenes Werk, CC BY-SA 3.0, <https://commons.wikimedia.org/w/index.php?curid=14697172>).

1021 Mit R kann mit der Funktion `boxplot()` ein Boxplot erstellt werden. Diese Funktion kann in zwei unter-  
 1022 schiedlichen Ausprägungen verwendet werden.

- 1023 1. `boxplot(x)` erzeugt einen Boxplot für die Variable `x`.
- 1024 2. `boxplot(x ~ y)` erzeugt einen oder mehrere Boxplots für `x` aber gruppiert nach `y`, dabei sollte `y` eine  
 1025 kategoriale Variable sein. `x` und `y` können auch die Spaltennamen eines `data.frames` sein, dann muss  
 1026 das `data.frame` mit dem Argument `data` zusätzlich übergeben werden.

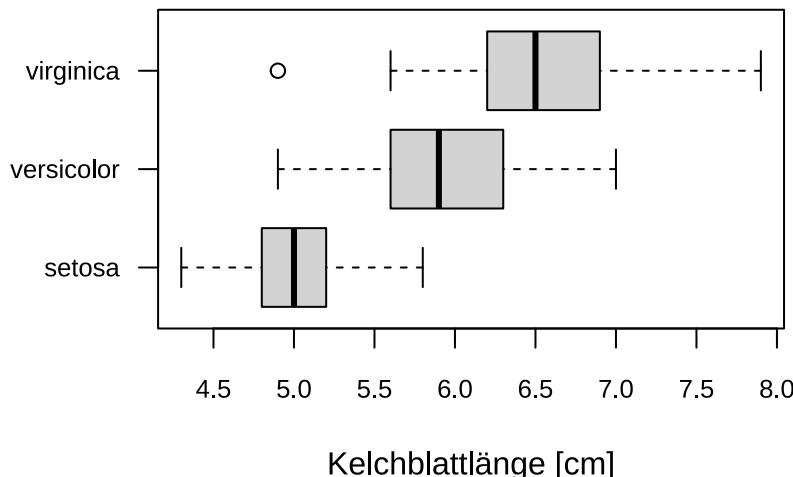
```
boxplot(iris$Sepal.Length ~ iris$Species)
```



1027

1028 Etwas eleganter ist es wenn wir das Argument `data` verwenden und den Plot etwas anpassen. Diese Schreib-  
1029 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
)
```



1030

1031

### 1032 Aufgabe 18: Boxplots

1033

- Lesen Sie erneut den Datensatz `daten/bhd_1.txt` ein und speichern Sie diesen in die Variable (`dat_bhd`).
  - Wie viele BHD-Messungen gibt es für jedes Gebiet?
  - Erstellen Sie für jedes Gebiet einen Plot
- 1037 Erstellen Sie Boxplots für jedes Gebiet und innerhalb der Gebiete für jede Art.

## 1038 8.4 ggplot2: Eine Alternative für Abbildungen

1039 ggplot2 ist ein alternatives Plotting-System in R. Sie können mit ggplot2 also grundsätzlich Abbildungen  
 1040 mit dem selben Inhalt erstellen, wie mit Base Plots. Die Syntax und die optische Darstellung unterscheiden  
 1041 sich jedoch grundsätzlich. ggplot2 basiert auf den *grammar of graphics* von Leland Wilkinson. Die Idee  
 1042 ist, alle nötigen Informationen der Abbildung miteinander zu verknüpfen. ggplot2 ist also diametral zu  
 1043 Base Plots. Während Sie in base plot alle Elemente selbst definieren, ist die Idee von ggplot2, dass Sie  
 1044 nur die Daten und den Diagrammtypen übergeben und die Funktion selbst eine schöne Abbildung erstellt.  
 1045 Selbstverständlich können Sie aber auch in ggplot2 viele Einstellungen vornehmen. Im base plot sehen  
 1046 Abbildungen zunächst sehr karg und etwas unfertig aus. Sie müssen viele Einstellungen vornehmen, um eine  
 1047 publizierfähige Grafik zu produzieren. In ggplot2 sollen auch die einfachste Abbildungen schon ästhetisch  
 1048 sein. Mit diesen gebündelten Informationen kann ggplot2 die Abbildung automatisch verschönern. So  
 1049 werden bspw. die Legenden automatisch erzeugt und auch die Formatierungen automatisch an die Datenlage  
 1050 angepasst. ggplot2 nimmt der\*dem Entwickler\*in also Arbeit ab. Dadurch sind die Abbildungen schon ohne  
 1051 viel Nacharbeit schick. Nachteil ist, dass der\*dem Entwickler\*in weniger Möglichkeiten zur Einstellung zur  
 1052 Verfügung stehen und nuterspezifische Sonderwünsche somit schwerer umsetzbar sind. Sehen Sie sich das  
 1053 Cheatsheet zu ggplot2 an. Es ist in RStudio unter [Help > Cheatsheets](#) zu finden.

1054 Bei ggplot2 sind Anweisungen zu den Daten und Anweisungen zur Darstellung voneinander getrennt. Die  
 1055 Daten werden in den Ästhetikbefehl übergeben und dort klassifiziert. Dann folgen die Darstellungsanweisungen.  
 1056 Ähnlich wie bei Base Plots, werden die Grafikelemente ebenenweise nacheinander programmiert, jedoch mit  
 1057 einem + verbunden. Und hier liegt der wesentliche Unterschied zu Base Plots. Durch die + werden die Ebenen  
 1058 zu einem Befehl verbunden und damit gleichzeitig erstellt.

1059 Die Erweiterung wird zunächst geladen<sup>7</sup>. Wir laden außerdem den Datensatz **iris**. Der Datensatz ist in R  
 1060 fest integriert. Siehe `?iris` für mehr Informationen.

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

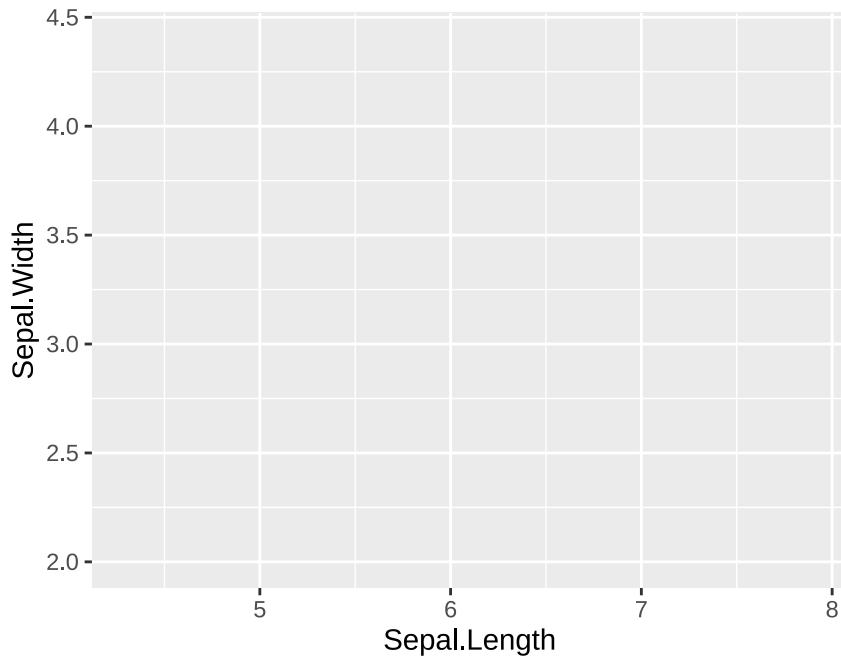
```
1061 ## Sepal.Length Sepal.Width Petal.Length Petal.Width Species
1062 ## 1 5.1 3.5 1.4 0.2 setosa
1063 ## 2 4.9 3.0 1.4 0.2 setosa
1064 ## 3 4.7 3.2 1.3 0.2 setosa
1065 ## 4 4.6 3.1 1.5 0.2 setosa
1066 ## 5 5.0 3.6 1.4 0.2 setosa
1067 ## 6 5.4 3.9 1.7 0.4 setosa
```

1068 Die Ästhetik wird bspw. folgendermaßen definiert.

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

---

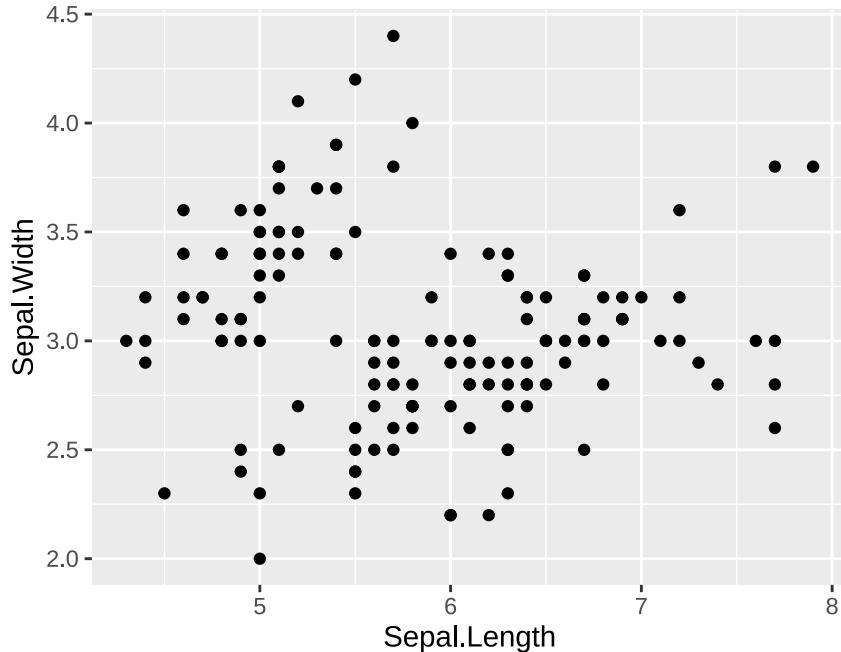
<sup>7</sup>Wir haben bis jetzt immer nur mit base R gearbeitet. D.h. wir haben nur Funktionen verwendet, die R bereits zur Verfügung stellt. Eine der großen Stärken von R sind die Erweiterungen (oder auch Pakete genannt). ggplot2 ist so eine Erweiterung, die einmal mit `install.packages("ggplot2")`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



1069

1070 Dieser Befehl zeichnet noch keine Daten. Die Daten werden lediglich herangezogen, um einen leeren Plot für  
1071 die Daten zu erstellen. In dem Beispiel wird die Variable `Sepal.Length` aus dem `data.frame iris` als x und  
1072 `Sepal.Width` als y Variable definiert. Diese Informationen stehen den folgenden Layern nun zur Verfügung,  
1073 sodass nach den + nur noch x und y verwendet werden müssen. Um bspw. einen Scatterplot zu erstellen  
1074 wird ein `geom_point()` Layer hinzugefügt. x und y werden automatisch an `geom_point` übergeben. Weitere  
1075 Einstellung sind in diesem Beispiel nicht notwendig, wären jedoch möglich. Siehe `?geom_point()`.

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



1076

1077

---

1078 **Aufgabe 19: Abbildungen mit ggplot2**

---

1079

1080 Verwenden Sie die Daten aus Aufgabe 16 und erstellen Sie einen Scatterplot mit ggplot2 wie in Aufgabe 16.

1081

1082 Wir haben mit der Funktion `geom_point()` demm Plot eine Punktgeometrie hinzugefügt. Es gibt noch viele  
1083 weitere Geometrien. Die wichtigsten sind:

- 1084 • `geom_line()` für eine Linie.  
1085 • `geom_histogram()` um ein Histogramm zu erstellen.  
1086 • `geom_boxplot()` um einen Boxplot zu erstellen.  
1087 • `geom_bar()` um ein Säulendiagramm zu erstellen.

1088 Welche Geometrie die richtige ist, richtet nach dem Typ der darzustellenden Variablen. Beispielsweise  
1089 bietet sich `geom_point()` an, wenn man zwei kontinuierliche Variable darstellen möchte. Wenn man hin-  
1090 gegen die Verteilung von einer kontinuirlchen Variable darstellen möchte, dann bietet sich ein Histogramm  
1091 (`geom_histogram()`) oder auch eine geschätzte Dichte (z.B. `geom_density()`) an.

1092

---

1093 **Aufgabe 20: Abbildungen mit ggplot2**

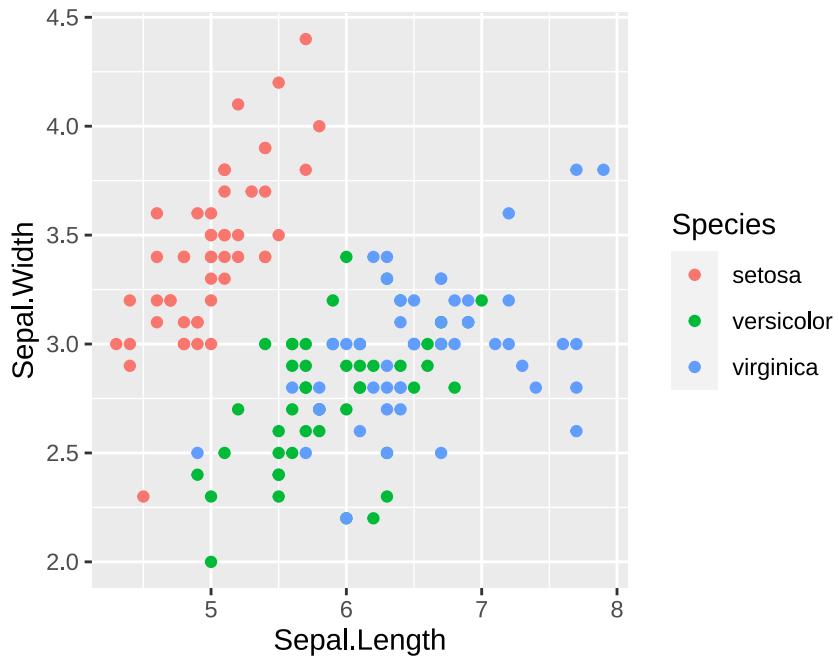
---

10941095 Verwenden Sie die den Iris Datensatz und erstellen Sie mit ggplot2 einen Plot der die Verteilung der Länge  
1096 der Kelchblätter zeigt (Spalte `Sepal.Length`).

1097

1098 Eine der Stärken von ggplot2 ist, dass man den Wert unterschiedlicher Variable auf unterschiedlichen  
1099 Komponenten des Plots abbilden kann. Wir haben bis jetzt ein bzw. zwei Variable auf der x- und y-Achse  
1100 abgebildet. Wir können aber ein weitere Variablen verwenden um das Aussehen des Plots zu beeinflussen.  
1101 Beispielsweise können wir die Farbe der Punkte (für `geom_point()`) mit dem Argument `col` beeinflussen.

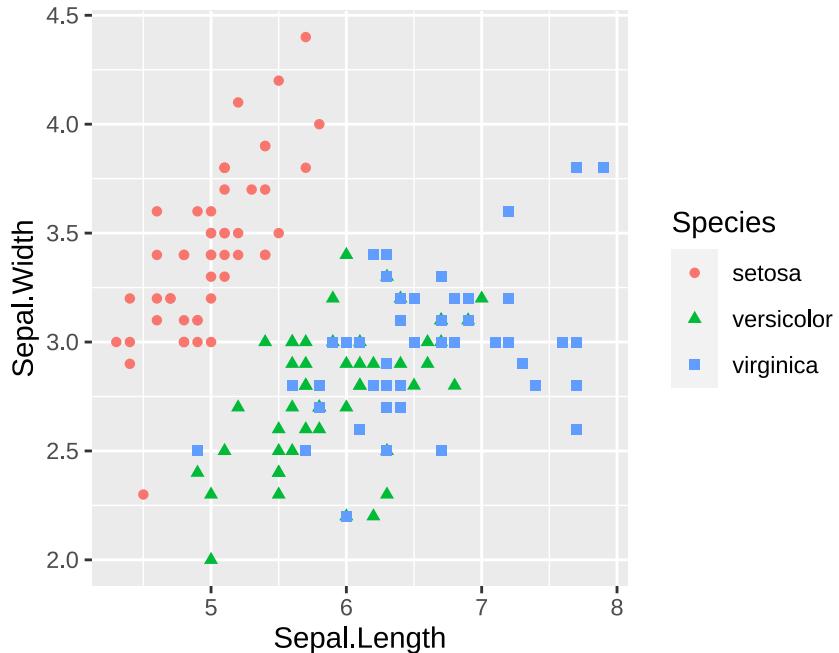
```
ggplot(iris, aes(x = Sepal.Length, y = Sepal.Width, col = Species)) +
 geom_point()
```



1102

1103 Somit bekommt jede Irisart eine eigene Farbe<sup>8</sup>. Gleichesmaßen können wir die Punktart (**shape**), die  
 1104 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()
```

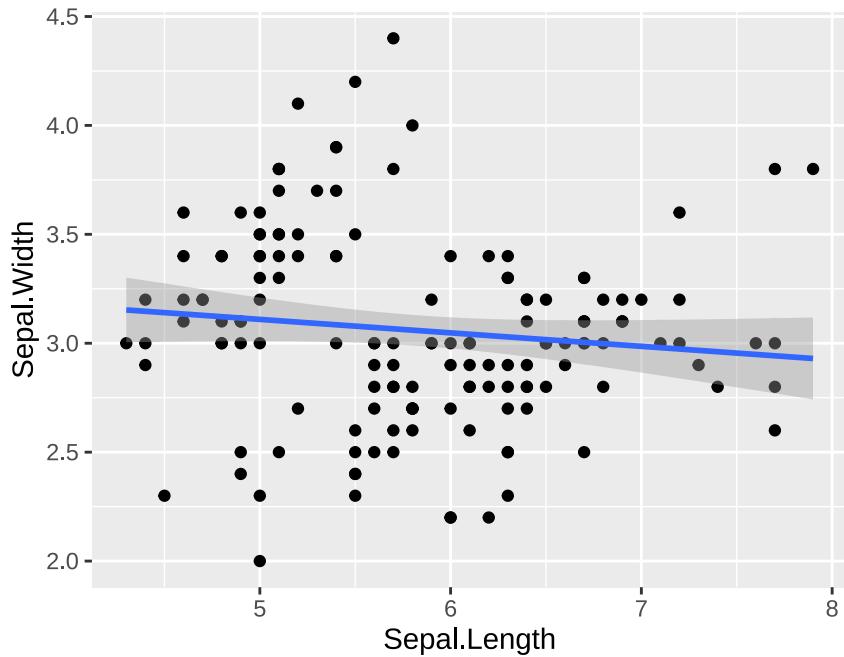


1105

1106 In dem Plot ist die Information zu der Art redundant (einmal als Farbe und einmal Symbolart).  
 1107 Ein weitere sehr nützliche Geometrie ist **geom\_smooth()**, die es erlaubt eine Trendlinie hinzuzufügen.

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

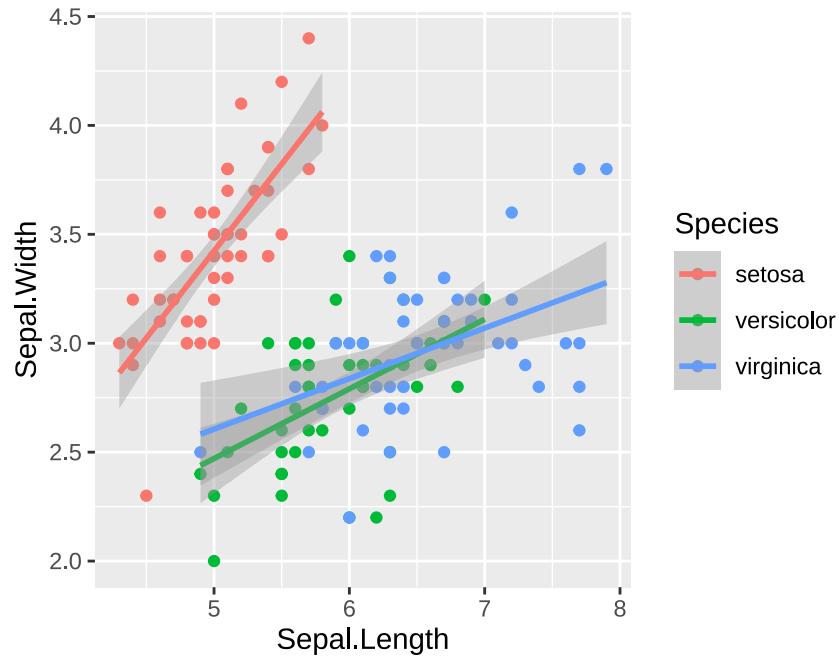
```
ggplot(iris, aes(x = Sepal.Length, y = Sepal.Width)) +
 geom_point() + geom_smooth(method = "lm")
```



1108

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")
```



1112

1113

1114 **Aufgabe 21: Anpassen von Plots**  
1115

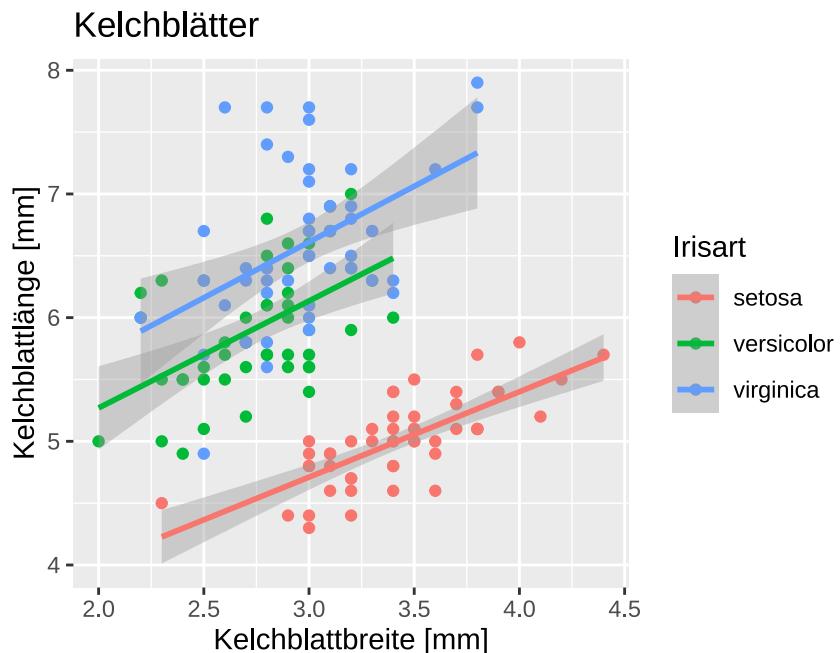
- 1116 Lesen Sie den Datensatz `data/bhd_1.txt` ein und erstellen Sie einen Boxplot für die Verteilung des BHDs  
 1117 für jede Baumart. In einem zweiten Schritt verwenden Sie erst `col = gebiet` und dann `fill = gebiet`.  
 1118 Welchen Unterschied stellen Sie fest?

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

1119

- 1120 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")
```



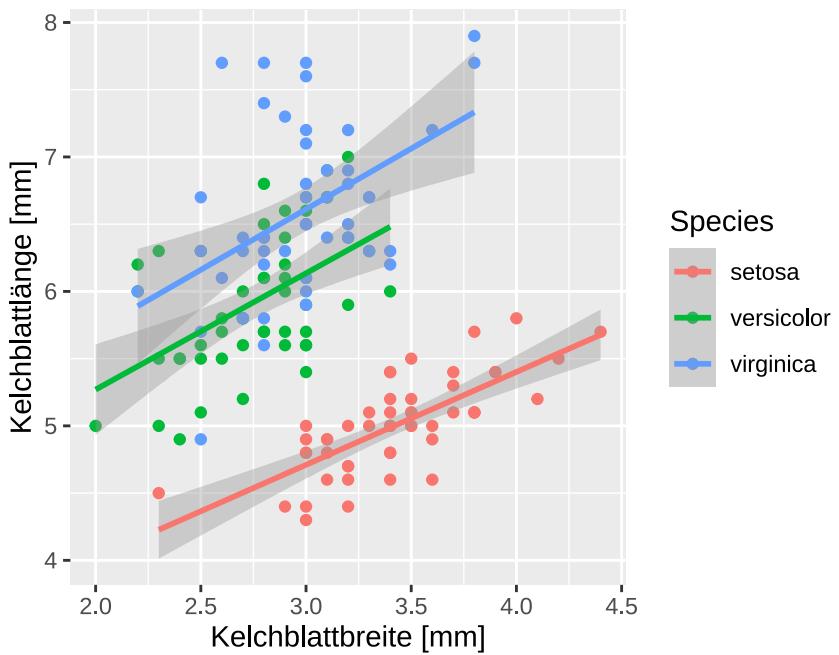
1121

- 1122 Statt einen langen Befehl zu tippen, kann ein `ggplot()` auch zwischengespeichert und wieder aufgerufen bzw.  
 1123 angepasst werden. Das ist vor allem sinnvoll, wenn mehrere Abbildungen auf dem selben Zwischenergebnis  
 1124 aufbauen sollen.

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

- 1125 Wir können jetzt mit `p1` weiter arbeiten und beispielsweise eine Beschriftung hinzufügen.

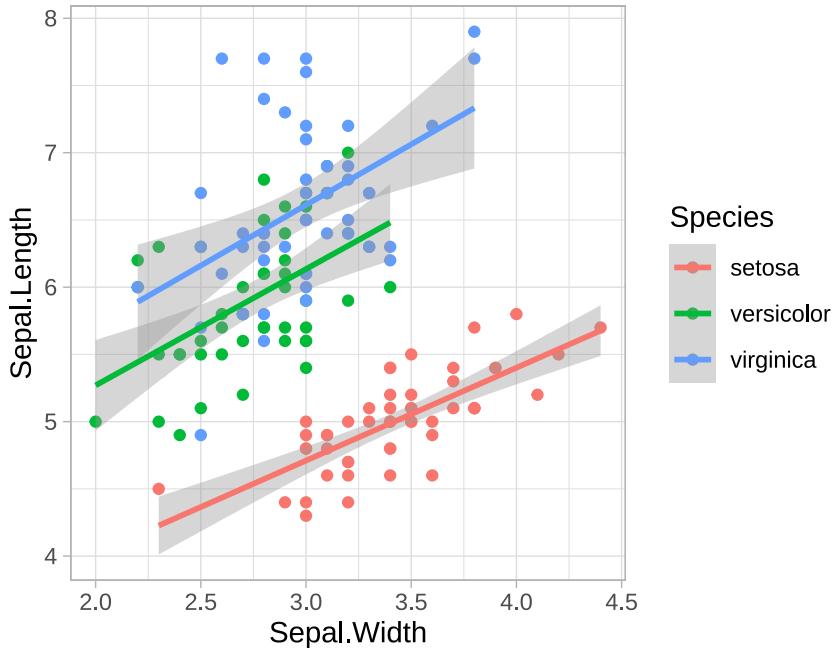
```
p1 + labs(x = "Kelchblattbreite [mm]", y = "Kelchblattlänge [mm]")
```



1126

1127 Oder auch den ganzen Plot anpassen. Dafür gibt es *themes*. Es gibt eine Reihe von vorgefertigten *themes*  
1128 oder man kann diese auch selber erstellen (das ist aber nicht Teil dieses Kurses).

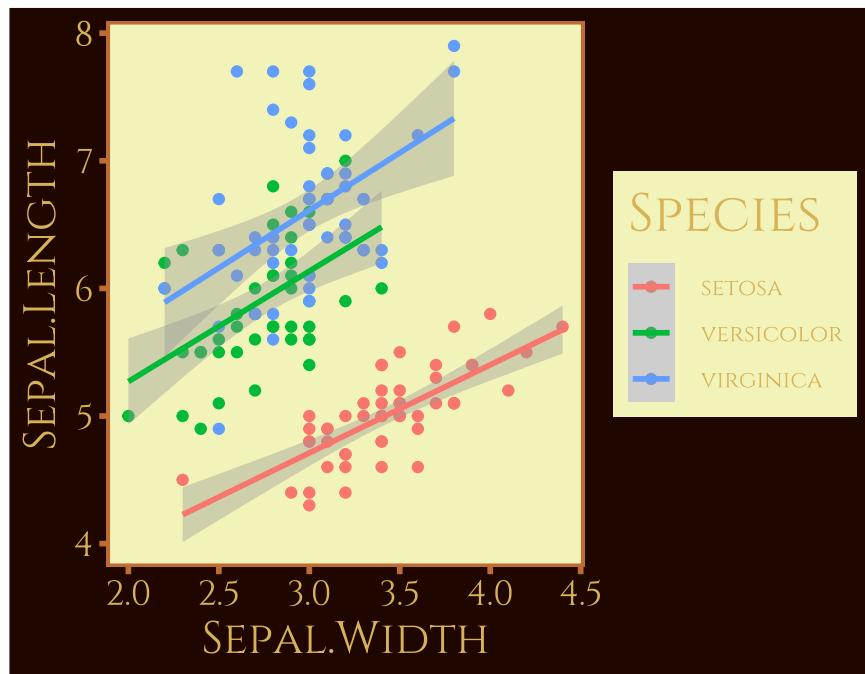
```
p1 + theme_light()
```



1129

1130 Weitere *themes* sind: `theme_bw()`, `theme_linedraw()` oder `theme_dark()`. Es gibt extra Pakete die viele  
1131 zusätzliche weitere *themes* anbieten. Dazu gehört z.B. das Paket `ggthemes` oder `ThemePark`. Während  
1132 `ggthemes` hauptsächlich durchdachte Grafikkonzepte liefert, sind die Themes aus `ThemePark` eher Popkultur  
1133 und nicht 100 %ig ernst gemeint.

```
p1 + theme_gamethrones()
```

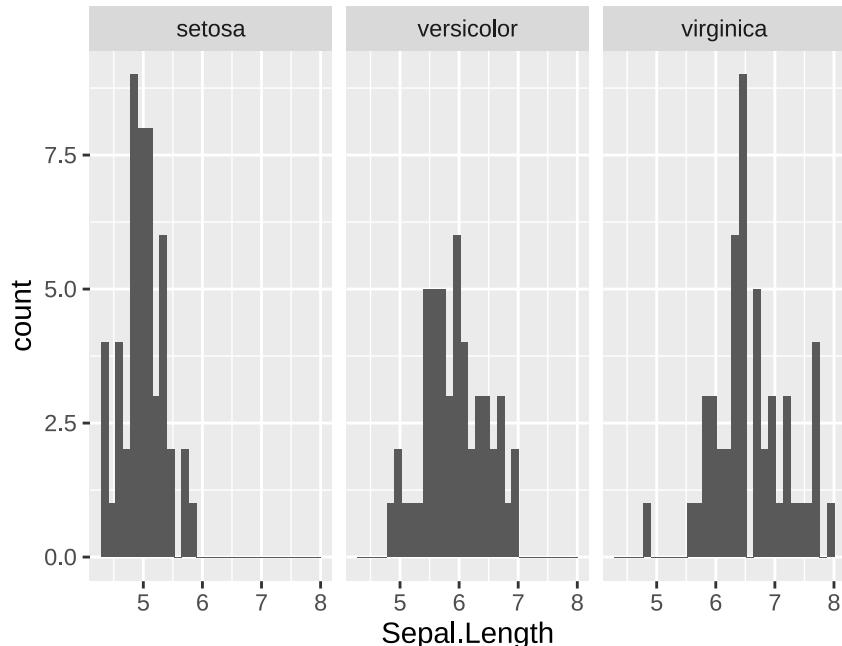


1134

#### 8.4.1 Multipanel Abbildungen

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

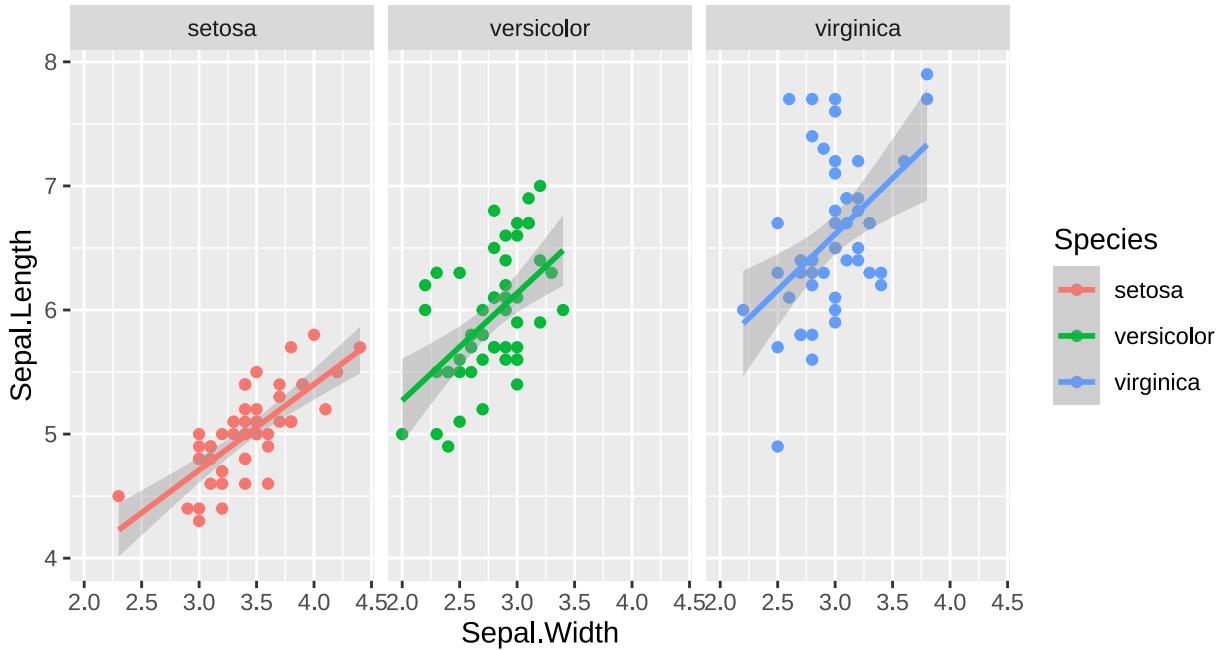
```
ggplot(iris, aes(Sepal.Length)) + geom_histogram() +
 facet_grid(~ Species)
```



1139

1140 Die Funktion `facet_grid()` erzeugt einen *Grid* und beschriftet die Grid-Zeilen und -Spalten, während  
 1141 `facet_wrap()` für jedes Panel (Diagramm) eine eigene Überschrift erzeugt. Beim Arrangieren der Diagramme  
 1142 wird der Vorteil von `ggplot2` gegenüber `base plot` besonders deutlich. Während es im `base plot` System  
 1143 mühsam ist, die Abbildungen zu arrangieren, übernimmt `ggplot2` das Arrangement automatisch und fügt  
 1144 sogar eine automatisch Legende hinzu. Die Achsenabschnitte werden so angepasst, dass die Daten vergleichbar  
 1145 sind.

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



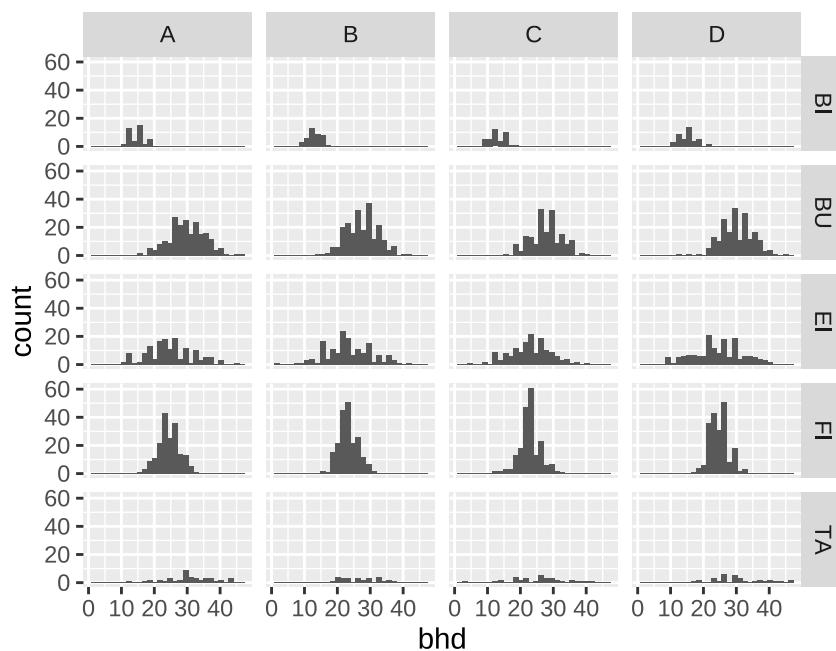
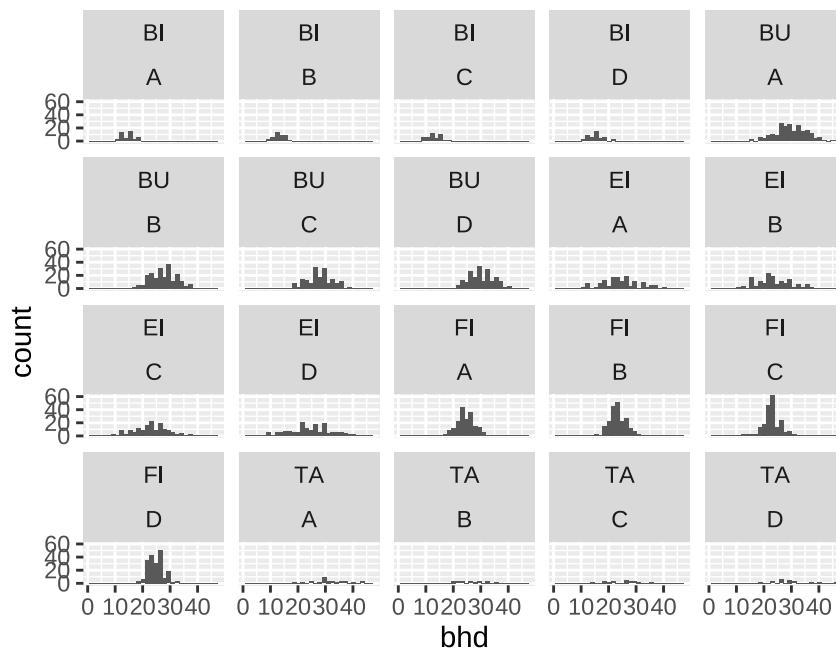
1146

1147

#### 1148 Aufgabe 22: Multipanel Abbildungen

---

1150 Lesen Sie erneut den Datensatz `daten/bhd_1.txt` ein und speichern Sie diesen in die Variable (`dat_bhd`).  
 1151 Erstellen Sie für jede Art und Gebiet ein Histogramm. Welche Unterschiede können Sie feststellen, wenn Sie  
 1152 `facet_grid()` oder `facet_wrap()` verwenden?



#### 1155 8.4.2 Plots kombinieren

1156 Es gibt Situationen, in denen **unterschiedliche** Plots miteinander kombiniert werden müssen. Im vorherigen  
 1157 Abschnitt wurde dies immer anhand einer gruppierenden Kovariate gemacht. Aber es gibt auch Situationen, in  
 1158 denen das nicht möglich ist. Beispielsweise wenn ein Histogramm und ein Scatterplot vom gleichen Datensatz  
 1159 zusammengefasst werden sollen. Dafür bietet sich das Paket **patchwork** an<sup>9</sup>.

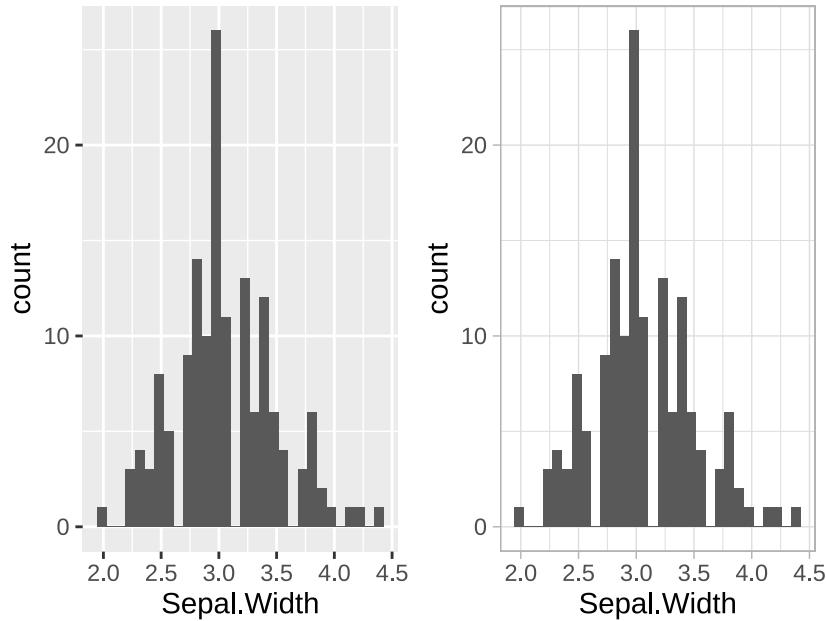
1160 Als erstes können wir zwei (oder natürlich auch mehrere Plots) erstellen. Hier unterscheiden sich die Plots  
 1161 lediglich durch das Aussehen.

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

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

- 1162 Dann müssen können wir diese Plots ebenfalls mit `+` zusammenfügen.

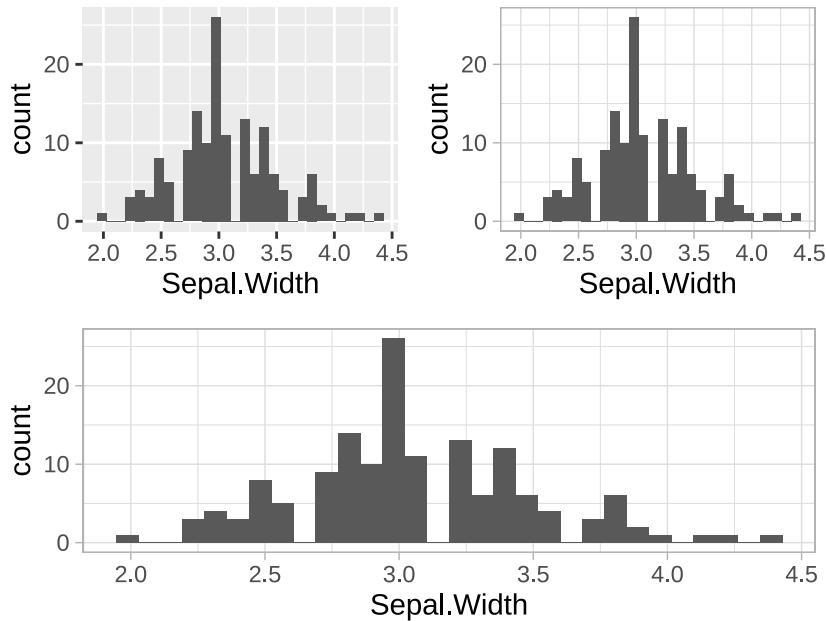
```
library(patchwork)
p1 + p2
```



1163

- 1164 Natürlich können auch weitere Plots hinzugefügt werden (auch in unterschiedlichen Dimensionen):

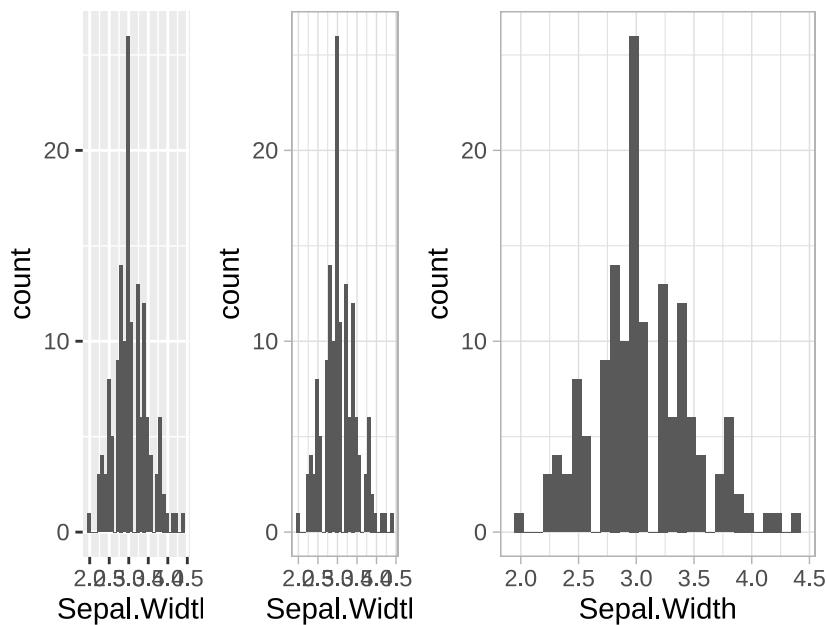
```
(p1 + p2) / p2
```



1165

- 1166 Des weiteren können mit `|` auch Plots gegenüber gestellt werden.

(p1 + p2) | p2

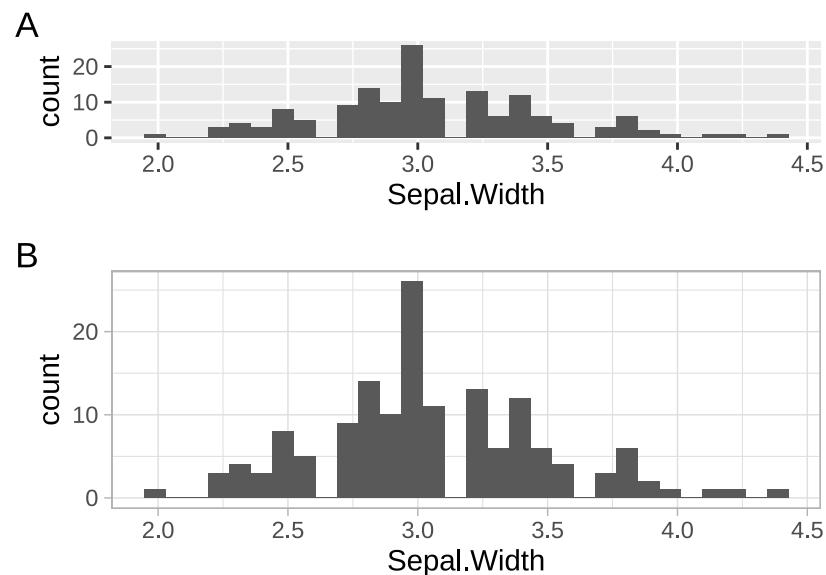


1167

1168 Weitere Optionen können mit `plot_layout()` und `plot_annotation()` angepasst werden. Mit  
1169 `plot_layout()` kann die Anordnung der Plots bestimmt werden (z.B. über die Argumente `nrow`  
1170 und `ncol`), sowie deren relative Größe (über die Argumente `widths` und `heights`). Mit der Funktion  
1171 `plot_annotation()` können zusätzliche Beschriftungen hinzugefügt werden, wie beispielsweise eine Titel  
1172 (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



1173

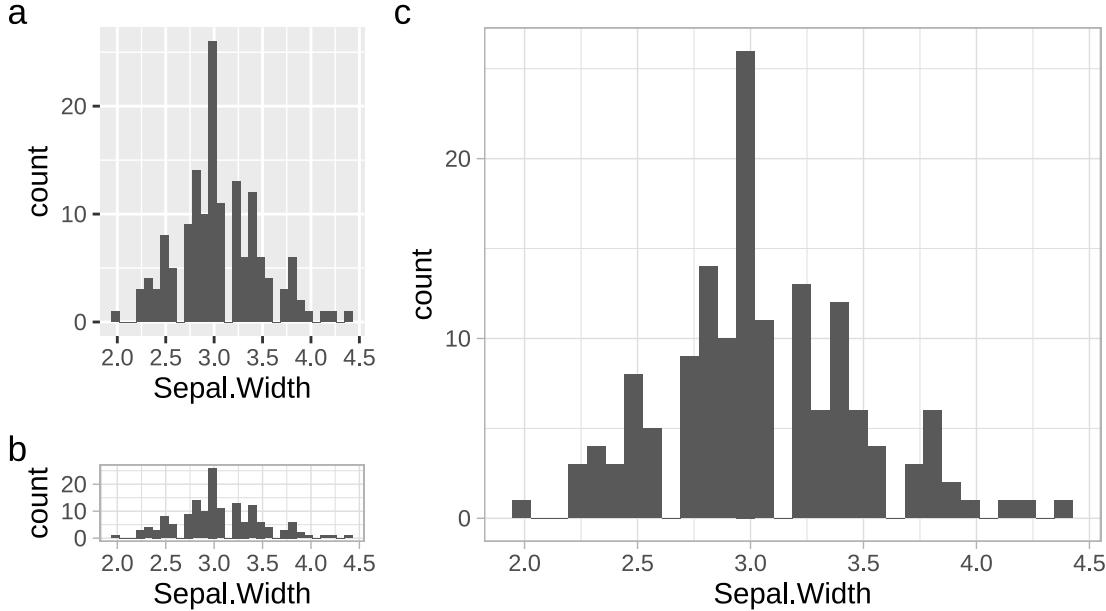
1174

---

**Aufgabe 23: Mehrere Plots zusammenfügen**

---

1177 Versuchen Sie die folgende Zusammenstellung der Plots nachzumachen:



1178

**8.4.3 Speichern von plots**

1180 Sie können mit `ggsave()` eine zwischen-gespeicherte Abbildung exportieren, indem Sie den Variablenamen  
1181 übergeben. Wenn Sie keine Variable übergeben, wird automatisch die letzte Abbildung gespeichert. Das  
1182 Dateiformat wird aus dem Dateinamen übernommen. Die Größe der Inhalte können Sie sehr leicht mit den  
1183 Argumenten `width` und `height` oder `scale` gesteuert werden.

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

## 1184 9 Mit Daten arbeiten

### 1185 9.1 dplyr eine Einführung

1186 `dplyr` ist eine Erweiterung von R (ein Paket), die das Ziel hat, den Umgang mit Daten einfacher und  
1187 schneller zu machen.

1188 `dplyr` definiert 5 Verben, um mit Daten zu arbeiten. Diese sind:

- 1189 • `filter`
- 1190 • `select`
- 1191 • `arrange`
- 1192 • `mutate`
- 1193 • `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))
```

1194 Damit die Funktionen aus `dplyr` verwendet werden können, müssen wir als erstes das Paket `dplyr` laden.  
1195 The online Resource [R for Data Science](#) gibt einen sehr guten, tieferen Eindruck in die Arbeit mit `dplyr`  
1196 und erklärt auch die `dplyr` Philosophie noch Mal sehr genau für Beginner.

```
library(dplyr)
```

1197 Sollte dies zu einer Fehlermeldung führen, dann müssen Sie das Paket `dplyr` erst installieren. Dafür müssen  
1198 Sie `einmalig install.packages("dplyr")` installieren.

1199 `dplyr` stellt unterschiedliche Funktionen zum Arbeiten mit Daten zur Verfügung. Es gibt fünf Grundfunktionen  
1200 für die am häufigsten vorkommenden Operationen. Mit der Funktion `filter()` können unterschiedliche  
1201 Beobachtungen gefiltert werden:

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

```
1202 ## id plot bhd alter
1203 ## 1 1 1 50 10
1204 ## 2 2 1 29 30
1205 ## 3 3 2 13 31
1206 ## 4 4 2 23 24
1207 ## 5 5 3 25 25
```

1208 Es können auch mehrere Spalten verwendet werden.

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

```
1209 ## id plot bhd alter
1210 ## 1 2 1 29 30
1211 ## 2 3 2 13 31
1212 ## 3 4 2 23 24
```

```
1213 ## 4 5 3 25 25
```

1214 Natürlich kann genau das gleiche Ergebnis mit dem ‘normalen’ R erreicht werden, dies wäre dann:

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

```
1215 ## id plot bhd alter
```

```
1216 ## 2 2 1 29 30
```

```
1217 ## 3 3 2 13 31
```

```
1218 ## 4 4 2 23 24
```

```
1219 ## 5 5 3 25 25
```

1220 Eine weitere Funktion aus dem Paket **dplyr** ist **select()**. Damit können Spalten aus einem **data.frame**

1221 ausgewählt werden. Dabei können auch die Spaltennamen unbenannt werden.

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

```
1222 ## bhd
```

```
1223 ## 1 50
```

```
1224 ## 2 29
```

```
1225 ## 3 13
```

```
1226 ## 4 23
```

```
1227 ## 5 25
```

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

```
1228 ## bhd id
```

```
1229 ## 1 50 1
```

```
1230 ## 2 29 2
```

```
1231 ## 3 13 3
```

```
1232 ## 4 23 4
```

```
1233 ## 5 25 5
```

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

```
1234 ## BHD id
```

```
1235 ## 1 50 1
```

```
1236 ## 2 29 2
```

```
1237 ## 3 13 3
```

```
1238 ## 4 23 4
```

```
1239 ## 5 25 5
```

1240 Mit der Funktion **arrange()** können die Beobachtungen in einem **data.frame** sortiert werden.

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

```
1241 ## id plot bhd alter
```

```
1242 ## 1 3 2 13 31
```

```
1243 ## 2 4 2 23 24
```

```
1244 ## 3 5 3 25 25
```

```
1245 ## 4 2 1 29 30
1246 ## 5 1 1 50 10
```

1247 Mit der Funktion `desc()` kann die Anordnung in absteigender Reihenfolge sortiert werden.

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

```
1248 ## id plot bhd alter
1249 ## 1 1 1 50 10
1250 ## 2 2 1 29 30
1251 ## 3 5 3 25 25
1252 ## 4 4 2 23 24
1253 ## 5 3 2 13 31
```

1254 Mit der Funktion `mutate()` kann man eine neue Spalte hinzufügen.

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

```
1255 ## id plot bhd alter bhd_mm fl
1256 ## 1 1 1 50 10 500 1963.4954
1257 ## 2 2 1 29 30 290 660.5199
1258 ## 3 3 2 13 31 130 132.7323
1259 ## 4 4 2 23 24 230 415.4756
1260 ## 5 5 3 25 25 250 490.8739
```

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

```
1261 ## id plot bhd alter mean_bhd
1262 ## 1 1 1 50 10 28
1263 ## 2 2 1 29 30 28
1264 ## 3 3 2 13 31 28
1265 ## 4 4 2 23 24 28
1266 ## 5 5 3 25 25 28
```

1267 Mit der Funktion `summarise()` können Spalten zusammengefasst werden.

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

```
1268 ## mean_bhd mean_sd
1269 ## 1 28 13.63818
```

1270

---

1271 **Aufgabe 24: Datenmanipulation mit dplyr**


---

1272

- 1273     1. Laden Sie den Datensatz
- `data/bhd_1.txt`
- 
- 1274     2. Berechnen Sie folgende Werte für alle Einträge und speichern Sie die Ergebnisse in
- `erg1`
- 
- 1275         • mittlerer
- `bhd`
- 
- 1276         • maximales
- `alter`
- 
- 1277         • die Standardabweichung des BHDs
- 
- 1278         • die Anzahl Bäume mit einem BHD > 30

1279 **9.2 Arbeiten mit gruppierten Daten**
1280 Zusätzlich können `mutate` und `summarise` auch auf gruppierte Daten angewendet werden. Dafür müssen  
1281 wir erst die Funktion `group_by()` aufrufen und als Argumente die Variablen übergeben, die die Gruppen  
1282 definieren.

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

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

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

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

summarise(dat, bhd_m = mean(bhd))

bhd_m
1 28

summarise(dat1, bhd_m = mean(bhd))

A tibble: 3 x 2
plot bhd_m
<dbl> <dbl>
1 1 28
```

```
1302 ## <dbl> <dbl>
1303 ## 1 1 39.5
1304 ## 2 2 18
1305 ## 3 3 25
```

1306

**Aufgabe 25: dplyr mit gruppierten Daten**

---

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

**1316 9.3 pipes oder %>%**

1317 Mit *Pipes* (`%>%`) kann man das Ergebnis einer Funktion einfach an eine nachfolgende Funktion weiterreichen.

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

1318 Wir kennen bis jetzt:

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

1319 ## [1] 3.333333

1320 Mit *Pipes*, die durch das Symbol `%>%` dargestellt werden<sup>10</sup>, können wir das etwas vereinfachen und nacheinander  
1321 schreiben:

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

1322 ## [1] 3.333333

1323 Oder sogar

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

1324 ## [1] 3.333333

1325

**Aufgabe 26: Pipes %>%**

---

1328 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.

- 1329 1. Laden Sie den Datensatz `data/bhd_1.txt`.
- 1330 2. Berechnen Sie für jede Baumart und Standort die folgenden Werte:
- 1331 • mittlerer `bhd`
  - 1332 • maximales `alter`
  - 1333 • die Standardabweichung des BHDs
  - 1334 • die Anzahl Bäume mit einem BHD > 30
- 1335 3. Ordnen Sie das Ergebnis nach Baumart und BHD.

## 1336 9.4 Joins

1337 Eine weitere häufige Aufgabe beim Daten Management ist es Daten zusammenzuführen. Nehmen Sie an, dass  
1338 wir folgende Aufnahmen gemacht haben

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

1339 und jeder Baum lediglich mit einer `id` versehen wurde. In einer zweiten Tabelle wurden dann weitere Daten  
1340 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")
)
```

1341 Ziel ist es jetzt die Bäume aus `aufnahmen` mit den Informationen aus den `metadaten` zu verbinden. Dazu  
1342 dient `id` als Bindeglied (oft auch Schlüssel genannt).

1343 Dazu gibt es vier Möglichkeiten.

1344 Zur Durchführung gibt es in base R die Funktion `merge()`. Wir werden aber gleich die Funktionen aus dem  
1345 Paket `dplyr` verwenden.

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

1346 ## id bhd art gebiet
1347 ## 1 1 20 <NA> <NA>
1348 ## 2 2 31 Ta A
1349 ## 3 3 74 Bu B

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

1350 ## id bhd art gebiet
1351 ## 1 2 31 Ta A
1352 ## 2 3 74 Bu B
1353 ## 3 4 NA Bu B
```

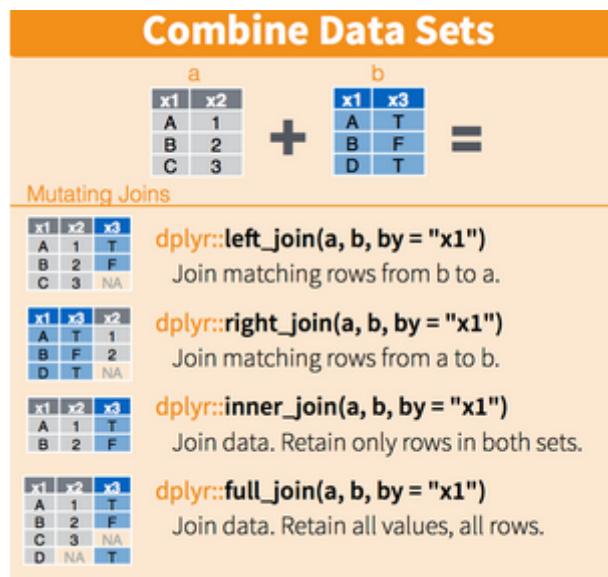


Abbildung 11: Joins (Quelle Rstudio)

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

```
1354 ## id bhd art gebiet
1355 ## 1 2 31 Ta A
1356 ## 2 3 74 Bu B
full_join(aufnahmen, metadaten, by = "id")
```

```
1357 ## id bhd art gebiet
1358 ## 1 1 20 <NA> <NA>
1359 ## 2 2 31 Ta A
1360 ## 3 3 74 Bu B
1361 ## 4 4 NA Bu B
```

1362 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"))
```

```
1363 ## id bhd art gebiet
1364 ## 1 1 20 <NA> <NA>
1365 ## 2 2 31 Ta A
1366 ## 3 3 74 Bu B
```

1367

1368 **Aufgabe 27: Verbinden von Daten**

---

- 1369
- 1370 • Lesen Sie die Datensätze `daten/bhd_2.txt` und `daten/bhd_2_meta.txt` ein.
  - 1371 • Stellen Sie sicher, dass es für den `bhd` keine fehlenden Werte gibt (entfernen sie entsprechende Zeilen)
  - 1372 • Fügen Sie zu den Metadaten (gespeichert in `bhd_2_meta`) die Anzahl Bäume und den mittleren `bhd`
  - 1373 hinz zu pro Gebiet.

1374 **9.5 ‘long’ and ‘wide’ Datenformate**

1375 Unter anderem Wickham (2014) empfiehlt das Prinzip von *tidy Data*. Nach diesem Prinzip sollten Daten wie  
 1376 folgt organisiert sein:

- 1377 • Jede Zeile ist ein Merkmalsträger/Subjekt/Objekt (z. B. eine Person oder ein Baum).
- 1378 • Jede Spalte ist eine Variable (ein Merkmal), die den Merkmalsträger beschreibt.
- 1379 • In jeder Zelle ist genau ein Wert (Merkmalausprägung), nämlich der Wert, der Variable für den Merk-  
 1380 malsträger.

1381 Zum Beispiel enthalten Spaltennamen oft Informationen, die eigentlich in einer Variable gespeichert werden  
 1382 sollten. Folgendes Beispiel gibt die BHD Messung von 3 Bäumen in 3 Jahren wieder. Der Vorteil von *tidy*  
 1383 Daten ist, dass viele Funktionen diese Form erwarten. Somit müssen sie Ihre Daten nur ein Mal organisieren  
 1384 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)
)
```

1385 Der `tibble` aus dem Paket `tibble` ist ein moderner Data Frame, der sich in seinen Funktionen in das `tidy`  
 1386 Universum einfügt. Wir können auch einfach das Paket `tidyverse` laden, um alle Pakete, die Teil des `tidy`  
 1387 Universums sind gleichzeitig zu laden. Der `tibble` beinhaltet alle Funktionen, die der Data Frame auch  
 1388 beinhaltet und kann auch für genau die gleichen Zwecke verwendet werden. Der `tibble` hat u. A. eine  
 1389 modernere Darstellung im Konsolenoutput.

1390 Diese Daten sind jetzt im `wide`-Format gespeichert und folglich nicht `tidy`, weil Information über die Daten  
 1391 (nämlich das Jahr der Aufnahme in den Spaltennamen gespeichert sind). Besser wäre eine Struktur mit  
 1392 nur drei Spalten: `id`, `jahr` und `bhd`. Um die Daten in so eine Struktur zu bringen, gibt es die Funktion  
 1393 `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
```

```

1396 ## <int> <chr> <dbl>
1397 ## 1 1 bhd2015 30
1398 ## 2 1 bhd2016 31
1399 ## 3 1 bhd2017 34
1400 ## 4 2 bhd2015 31
1401 ## 5 2 bhd2016 31
1402 ## 6 2 bhd2017 32
1403 ## 7 3 bhd2015 32
1404 ## 8 3 bhd2016 33
1405 ## 9 3 bhd2017 33

```

1406 Wenn wir die Spalten für die Variable und den Wert gleich sinnvoll benennen möchten, können wir das über  
1407 die Argumente `names_to` und `values_to` machen.

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

```

1408 ## # A tibble: 9 x 3
1409 ## id jahr bhd
1410 ## <int> <chr> <dbl>
1411 ## 1 1 bhd2015 30
1412 ## 2 1 bhd2016 31
1413 ## 3 1 bhd2017 34
1414 ## 4 2 bhd2015 31
1415 ## 5 2 bhd2016 31
1416 ## 6 2 bhd2017 32
1417 ## 7 3 bhd2015 32
1418 ## 8 3 bhd2016 33
1419 ## 9 3 bhd2017 33

```

1420 Analog zu der Funktion `pivot_longer()` gibt es auch die Funktion `pivot_wider()`, um vom Daten vom  
1421 `long`-Format ins `wide`-Format zu transformieren.

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

```

1422 ## # A tibble: 3 x 4
1423 ## id bhd2015 bhd2016 bhd2017
1424 ## <int> <dbl> <dbl> <dbl>
1425 ## 1 1 30 31 34
1426 ## 2 2 31 31 32
1427 ## 3 3 32 33 33

```

1428

---

1429 **Aufgabe 28: Zeitliche Verlauf von BHDs**

---

1431 In der Datei `bhd_3.csv` befinden sich gemessene BHDs (in cm) von unterschiedlichen Bäumen zu unter-  
 1432 schiedlichen Zeitpunkten. Erstellen Sie ein Liniendiagramm, das den zeitlichen (x-Achse) Verlauf der BHDs  
 1433 (y-Achse) für die unterschiedlichen Bäume darstellt.

1434 **9.6 Auswählen von Variablen**

1435 Sobald die Datensätze etwas umfangreicher werden (d. h., es gibt mehrere Spalten in einem `data.frame`),  
 1436 können innerhalb vieler `dplyr`-Funktionen spezielle Funktionen verwendet werden, um Variablen auszuwählen.

1437 Wenn die genaue Position der Spalten bekannt ist, kann man mit dem `:`-Operator und der Position Spalten  
 1438 auswählen:

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

```
1439 ## Sepal.Length Sepal.Width Petal.Length
1440 ## 1 5.1 3.5 1.4
1441 ## 2 4.9 3.0 1.4
1442 ## 3 4.7 3.2 1.3
```

1443 Diese Vorgehensweise kann gehährlich sein, da sich manchmal Spalten verschieben und sich somit die  
 1444 Positionen ändern. Es besser Spalten immer explizit mit Namen statt mit Nummern anzusprechen.

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

```
1445 ## Sepal.Length Sepal.Width Petal.Length
1446 ## 1 5.1 3.5 1.4
1447 ## 2 4.9 3.0 1.4
1448 ## 3 4.7 3.2 1.3
```

1449 `select()` erlaubt es, auch hier den `:`-Operator zu verwenden:

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

```
1450 ## Sepal.Length Sepal.Width Petal.Length
1451 ## 1 5.1 3.5 1.4
1452 ## 2 4.9 3.0 1.4
1453 ## 3 4.7 3.2 1.3
```

1454 Es gibt auch einige spezielle Funktionen, um Spalten innerhalb eines `select()`-Aufrufs auszuführen:

- 1455 • `starts_with()`: Hier kann man ein Muster angeben, mit dem ein Text anfangen muss.
- 1456 • `ends_with()`: Diese Funktion ist analog zu `starts_with()`, jetzt wird aber am Ende des Spaltennamens  
 1457 nach dem Muster gesucht.
- 1458 • `contains()`: Hier kann ein Muster übergeben werden, das irgendwo im Spaltennamen sein muss.
- 1459 • `everything()`: Mit dieser Funktion werden alle Spalten ausgewählt.

1460 • `last_col()`: Mit dieser Funktion wird nur die letzte Spalte ausgewählt (dass ist die Spalte, die ganz  
1461 rechts ist).

1462 Sämtliche Auswahlen können mit - umgekehrt werden.

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

1463 ## Sepal.Length Sepal.Width

1464 ## 1 5.1 3.5

1465 ## 2 4.9 3.0

1466 ## 3 4.7 3.2

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

1467 ## Petal.Length Petal.Width Species

1468 ## 1 1.4 0.2 setosa

1469 ## 2 1.4 0.2 setosa

1470 ## 3 1.3 0.2 setosa

1471 `select()` bietet auch noch die Möglichkeit, Spaltennamen zu ändern.

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

1472 ## sep\_width

1473 ## 1 3.5

1474 ## 2 3.0

1475 ## 3 3.2

1476

### 1477 Aufgabe 29: Auswählen von Spalten

---

1479 In der Datei `messungen_1.csv` sind Messungen von zwei Sensoren enthalten für die ersten vier Monate eines  
1480 Jahres. Führen Sie folgende Abfragen durch:

1481 1. Wählen Sie alle Messungen für Januar aus.

1482 2. Wählen Sie alle Messungen für Januar und März aus.

## 1483 9.7 Einzelne Beobachtungen abfragen (`slice()`)

1484 Mit dem Befehl `slice()` kann man einzelne Beobachtungen (= Zeilen) aus einem `data.frame` abfragen.

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

1485 ## Sepal.Length Sepal.Width Petal.Length Petal.Width Species

1486 ## 1 5.1 3.5 1.4 0.2 setosa

1487 ## 2 4.4 2.9 1.4 0.2 setosa

1488 ## 3 5.1 3.5 1.4 0.3 setosa

1489 Davon gibt es drei nützliche Varianten: 1) `slice_head()` und `slice_tail()`; 2) `slice_max()` und  
 1490 `slice_min()`; 3) `slice_random()`.

1491 `slice_head()` und `slice_tail()` sind analog zu `head()` und `tail()`, aber mit dem entscheidenden Unter-  
 1492 schied, dass Gruppierungen berücksichtigt werden. Wenn keine Gruppierung in den Daten vorhanden ist, gibt  
 1493 es keinen Unterschied.

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

```
1494 ## Sepal.Length Sepal.Width Petal.Length Petal.Width Species
1495 ## 1 5.1 3.5 1.4 0.2 setosa
1496 ## 2 4.9 3.0 1.4 0.2 setosa
```

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

```
1497 ## Sepal.Length Sepal.Width Petal.Length Petal.Width Species
1498 ## 1 5.1 3.5 1.4 0.2 setosa
1499 ## 2 4.9 3.0 1.4 0.2 setosa
```

1500 Sobald jedoch eine gruppierende Variable eingeführt wird, gibt `slice_head()` die ersten n Beobachtungen  
 1501 für jede Gruppe zurück und `head()` für den gesamten Datensatz.

```
base head
```

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

```
1502 ## # A tibble: 2 x 5
1503 ## # Groups: Species [1]
1504 ## Sepal.Length Sepal.Width Petal.Length Petal.Width Species
1505 ## <dbl> <dbl> <dbl> <dbl> <fct>
1506 ## 1 5.1 3.5 1.4 0.2 setosa
1507 ## 2 4.9 3 1.4 0.2 setosa
```

```
dplyr slice_head
```

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

```
1508 ## # A tibble: 6 x 5
1509 ## # Groups: Species [3]
1510 ## Sepal.Length Sepal.Width Petal.Length Petal.Width Species
1511 ## <dbl> <dbl> <dbl> <dbl> <fct>
1512 ## 1 5.1 3.5 1.4 0.2 setosa
1513 ## 2 4.9 3 1.4 0.2 setosa
1514 ## 3 7 3.2 4.7 1.4 versicolor
1515 ## 4 6.4 3.2 4.5 1.5 versicolor
1516 ## 5 6.3 3.3 6 2.5 virginica
1517 ## 6 5.8 2.7 5.1 1.9 virginica
```

1518 `slice_tail()` funktioniert analog zu `slice_head()` mit dem einzigen Unterschied, dass nicht die ersten n

1519 Zeilen zurück gegeben werden sondern die letzten `n` Zeilen.  
 1520 `slice_max()` und `slice_min()` geben die Beobachtung mit dem maximalen bzw. minimalen Wert einer  
 1521 Variable zurück. Auch hier werden Gruppen berücksichtigt.

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

1522 ## Sepal.Length Sepal.Width Petal.Length Petal.Width Species  
 1523 ## 1 7.9 3.8 6.4 2 virginica

1524 Und mit Gruppen:

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

1525 ## # A tibble: 3 x 5  
 1526 ## # Groups: Species [3]  
 1527 ## Sepal.Length Sepal.Width Petal.Length Petal.Width Species  
 1528 ## <dbl> <dbl> <dbl> <dbl> <fct>  
 1529 ## 1 5.8 4 1.2 0.2 setosa  
 1530 ## 2 7 3.2 4.7 1.4 versicolor  
 1531 ## 3 7.9 3.8 6.4 2 virginica

1532 `slice_min()` funktioniert genau gleich, nur dass die Beobachtung (=Zeile) mit dem kleinsten Wert einer  
 1533 Variable zurück gegeben wird.

1534 Die Funktion `slice_sample()` erlaubt es zufällige Beobachtungen zu ziehen. Dabei kann über das Argument `n`  
 1535 die Anzahl an Beobachtungen angegeben werden oder über das Argument `prop` der Anteil an Beobachtungen.

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

1536 ## Sepal.Length Sepal.Width Petal.Length Petal.Width Species  
 1537 ## 1 6.5 2.8 4.6 1.5 versicolor  
 1538 ## 2 6.3 3.3 4.7 1.6 versicolor  
 1539 ## 3 7.2 3.2 6.0 1.8 virginica  
 1540 ## 4 4.9 3.6 1.4 0.1 setosa  
 1541 ## 5 6.0 2.7 5.1 1.6 versicolor

1542 Das Ergebnis ist bei jedem von Ihnen anders, da es sich um eine zufällige Ziehung handelt. Wenn Sie diese  
 1543 Ergebnisse wiederholen möchten, können Sie über `set.seed()` die zufällige Ziehung reproduzierbar machen.

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

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

1544 ## Sepal.Length Sepal.Width Petal.Length Petal.Width Species  
 1545 ## 1 4.3 3.0 1.1 0.1 setosa  
 1546 ## 2 5.0 3.3 1.4 0.2 setosa  
 1547 ## 3 7.7 3.8 6.7 2.2 virginica  
 1548 ## 4 4.4 3.2 1.3 0.2 setosa  
 1549 ## 5 5.9 3.0 5.1 1.8 virginica

1550 Wenn beispielsweise 5% der Beobachtungen gezogen werden sollen, kann dies so gemacht werden:

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

```
1551 ## Sepal.Length Sepal.Width Petal.Length Petal.Width Species
1552 ## 1 7.7 3.8 6.7 2.2 virginica
1553 ## 2 5.5 2.5 4.0 1.3 versicolor
1554 ## 3 5.5 2.6 4.4 1.2 versicolor
1555 ## 4 6.5 3.0 5.2 2.0 virginica
1556 ## 5 6.1 3.0 4.6 1.4 versicolor
1557 ## 6 6.3 3.4 5.6 2.4 virginica
1558 ## 7 5.1 2.5 3.0 1.1 versicolor

1559 slice_sample() berücksichtigt ebenfalls Gruppen. Mit den Argumenten replace und weight_by dann die
1560 Zufallsziehung genauer spezifiziert werden. replace sagt, ob eine gezogenen Beobachtung wieder zurück gelegt
1561 wird oder nicht. Mit dem Argument weight_by können optional gewichtete für jede Beobachtung vergeben
1562 werden.
```

1563

#### 1564 Aufgabe 30: Daten beschreiben

---

1566 Verwenden Sie den Datensatz `bhd_1.txt` und finden Sie für jedes Gebiet und Art die Beobachtung mit
1567 kleinsten BHD.

## 1568 9.8 Spalten trennen

1569 Ein gut geplanter Datensatz besteht aus Beobachtungen (Zeilen), Variablen (Spalten) und in jeder Zelle ist
1570 immer ein **genau** ein Wert gespeichert. Leider gibt es oft Datensätze, bei denen mehr als ein Wert pro Zelle
1571 gespeichert wurde. Die Funktion `separate()` kann helfen solche Daten zu trennen.

1572 Wir verwenden einen erfunden Datensatz zu Beobachtungen von Tieren und einer geschätzten Distanz zu
1573 diesen Tieren.

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

1574 In der Spalte `beobachtung` sind zwei Informationen gespeichert: Die Distanz zur Beobachtung und die Art.
1575 Das ist ungünstig, weil wir so weder nach Tierart noch nach distanz filtern können (nicht *tidy*). Mit der
1576 Funktion `separate()`, können wir Beobachtungen einer Spalte in mehrere Spalten trennen. Dafür muss der
1577 Spaltennamen (Argument `col`), die neuen Spaltennamen (Argument `into`) und das Trennzeichen (Argument
1578 `sep`) angegeben werden.

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

1579 ## # A tibble: 4 x 3

```
1580 ## id Distanz Art
1581 ## <int> <chr> <chr>
1582 ## 1 1 10m " Reh"
1583 ## 2 2 100m " Reh"
1584 ## 3 3 20m " Fuchs"
1585 ## 4 4 40 "Reh"

1586 Nach dem Aufruf von separate() gibt es zwei neue Spalten (Distanz und Art), die die alte Spalte
1587 beobachtung ersetzen.
```

1588

---

1589 **Aufgabe 31: Aufräumen**

---

1591 Verwenden Sie den folgenden Datensatz und bringen Sie ihn in eine Form, die sicherstellt dass

- 1592 • jede Zelle genau einen Wert enthält.  
1593 • jede Zeile eine Beobachtung ist.  
1594 • 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")
)
```

## 1595 10 Arbeiten mit Text

1596 Bis jetzt haben wir fast ausschließlich mit Zahlen oder Abbildungen gearbeitet. R bietet aber auch viele  
 1597 Werkzeuge, um mit Text zu arbeiten. Wir wollen hier ein paar Funktionen dafür vorstellen. Als erstes sollte  
 1598 nochmals klar gestellt werden, was eigentlich ein Text ist. In R ist alles, das innerhalb von doppelten ("") oder  
 1599 einfachen ('') Anführungszeichen geschrieben ist, Text.

1600 Anbei einige Beispiele:

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

1601 Wichtig ist hier zu sehen, dass z nicht als Zahl sondern, als Text interpretiert wird (siehe Datentypen).

```
z + 1
```

1602 ## Error in z + 1: non-numeric argument to binary operator

1603 Wenn man sicher ist, dass es sich bei einem Textobjekt um eine Zahl handelt, kann man dies mit der Funktion  
 1604 `as.numeric()` in eine Zahl umwandeln.

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

1605 ## [1] 31

1606 Aber mit a führt dies wieder zu einem NA-Wert, da a nicht in eine Zahl umgewandelt werden kann.

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

1607 ## Warning: NAs introduced by coercion

1608 ## [1] NA

### 1609 10.1 Arbeiten mit Text

1610 Wir wollen erst einmal drei Funktionen besprechen, die es erlauben mit Text zu arbeiten. Die Funktion  
 1611 `nchar()`<sup>11</sup> gibt an wie viele Zeichen ein Text hat. Also z.B.

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

1612 ## [1] 5

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

1613 ## [1] 2

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

1614 ## [1] 20

1615 Die Funktion `paste()` erlaubt es verschiedene Variablen mit Text zu verbinden. Wenn wir z. B. die Variablen  
 1616 `vorname <- "Eva"` und `name <- "Meier"` haben und wir wollen eine neue Variable `full_name <- Eva`

---

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

1617 "Meier" erzeugen, dann kann das mit der Funktion `paste()` gemacht werden.

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

1618 ## [1] "Eva Meier"

1619 Die Funktion `paste()` hat das Argument `sep`, das auf ein Leerzeichen ( ) gesetzt ist, aber auch anders sein  
1620 kann und das Trennzeichen definiert.

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

1621 ## [1] "Eva, Meier"

1622 Die Funktion `substr()` erlaubt es am Anfang oder Ende eines Wortes etwas abzuschneiden. Dabei muss  
1623 immer die Anfangs- (`start`) und Endposition (`stop`) angegeben werden.

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

1624 ## [1] "Hal"

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

1625 ## [1] "allo"

1626

### 1627 Aufgabe 32: Arbeiten mit Text 1

---

1629 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")
```

1630 1. Aus wie vielen Buchstaben besteht jedes Wort?

1631 2. Finden Sie das längste Wort.

1632 3. Wie viel Prozent der Wörter fangen mit einem S an?

1633 4. Fügen Sie jedem Wort seine Position im Vektor hinzu. Beispielsweise soll aus `Vogel` "2. Vogel" werden  
1634 usw.

## 1635 10.2 Finden von Textmustern

1636 Mit der Funktion `grep()` können Muster in einem Text gefunden werden. Wenn wir beispielsweise folgenden  
1637 Vektor mit Textelementen haben.

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

1638 Und wir wollen alle Straßennamen die ein `weg` haben abfragen, dann können wir folgenden Befehl ausführen:

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

1639 `## [1] 2`

1640 Im zweiten Element von `txt` kommen die Zeichen `Weg` vor. Beachte, in der Standardeinstellung wird zwischen  
1641 Groß- und Kleinschreibung unterschieden. Dies kann mit dem Argument `ignore.case = TRUE` angepasst  
1642 werden.

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

1643 `## [1] 1 2`

1644 Mit der Funktion `sub` können Zeichen innerhalb einer Zeichenkette ersetzt werden.

1645 So ersetzt der folgende Ausdruck `ae` mit `ä`.

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

1646 `## [1] "Friedländer Weg"`

1647 Wenn allerdings das zu ersetzende Zeichen mehr als einmal vorkommt und beide Instanzen ersetzt werden  
1648 sollen braucht man die Funktion `gsub`.

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

1649 `## [1] "Friedländer Weg und Reinhäuser Landstrasse sind zwei Strassen in Goettingen."`

1650 Mit `sub()` wird nur das erste `ae` ersetzt, während `gsub()` alle `ae` mit einem `ä` ersetzt.

```
gsub("ae", "ä", txt)
```

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

1652 Oft ist der genaue Ausdruck den man finden möchte jedoch Variabel. Beispielsweise möchte man alle Wörter  
1653 mit einem Umlaut oder Zahlen finden möchte, kann man das oft abkürzen. Dafür gibt es reguläre Ausdrücke.  
1654 Wir werden hier nur ein paar beispielhafte Anwendungen besprechen.

1655 Sowohl in den Funktionen `grep()` als auch `(g)sub()` kann mit anstatt dem Muster (immer das erste  
1656 Argument) aus ein regulärer Ausdruck angegeben werden. Mit `[1-9]` sind alle Zahlen von 1 bis 9 gemeint.

1657 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)
```

1658 `## [1] 1 3`

1659 Damit lässt sich auch das Problem mit groß- und kleingeschriebenen Wörtern lösen.

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

```
1660 ## [1] 1 2
```

1661

---

**Aufgabe 33: Arbeiten mit Text 2**

---

1664 Verwenden Sie den folgenden Vektor:

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

1665 1. In wie vielen Wörtern kommt der Doppellaut au vor?

1666 2. Ersetzen Sie in allen Wörtern alle au mit \_ \_.

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

```
1667 ## [1] 5 6 13
```

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

```
1668 ## [1] "Versicherung" "Methoden" "Fluss" "Rudel" "B_ _m"
1669 ## [6] "H_ _s" "Foto" "Auffahrt" "Auto" "Handy"
1670 ## [11] "Teller" "Kalender" "Aufb_ _"
```

## 1671 11 Arbeiten mit Zeit

1672 Für den Computer bzw. R ist ein Datum erst einmal nichts anderes als ein Text. Für uns ist es sofort  
 1673 klar, dass der "13.2.2021" der 13. Februar 2021 ist, für den Computer zunächst nicht. Wir müssen R also  
 1674 irgendwie sagen, dass die 13 der Tag ist, die 2 der Monat und 2021 das Jahr. Dass der Computer die einzelnen  
 1675 Komponenten erkennt, und in einen Datentyp eigens für Zeitformate überführt, nennt man *parsen*<sup>12</sup>. Durch  
 1676 das *parsen* wird die Variable in den Datentyp **Date** überführt. Das Arbeiten mit Datum und Zeit kann  
 1677 kann anfangs sehr mühsam sein und viele Zeit-spezifischen Datenoperationen lassen sich auch mit den  
 1678 Basis-Datentypen durchführen. Sobald man einige Grundfertigkeiten erworben hat, stellt man jedoch fest,  
 1679 dass die Arbeit mit dem Zeitformat-Datentyp schneller und effizienter funktioniert. Starten Sie am besten  
 1680 gleich mit "echten" Zeit-Datentypen und versuchen Sie nicht, sich irgendwie mit Text und numerischen  
 1681 Datentypen selbst etwas zu basteln. Der erste Schritt ist immer ein Datum zu *parsen*. Wir verwenden dafür  
 1682 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)
```

1683 **lubridate** bietet eine Vielzahl von Funktionen zum *parsen* von Datum und Zeit, die sich aus:

- 1684 • y für Jahr,
- 1685 • m für Monat,
- 1686 • d für Tag,
- 1687 • h für Stunde,
- 1688 • m für Minute und
- 1689 • s für Sekunde

1690 zusammen setzten. Alle Funktionen nehmen als erstes Argument ein Textstring. Wenn wir z.B. den String  
 1691 2020-01-20 parsen wollen können wir das mit der Funktion **ymd** machen.

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

1692 ## [1] "2020-01-20"

1693 Dabei erkennt **lubridate** in der Regel die Trennzeichen:

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

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

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

1695 ## [1] "2020-01-20"

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

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

1697 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")

1698 ## [1] "2020-01-20"
1699 Jetzt stellt sich die Frage, was der Vorteil ist, wenn R ein Datum parst.
d <- dmy("20.1.2020")

1700 Wir können jetzt mit d arbeiten und einzelne Komponenten extrahieren.
day(d)

1701 ## [1] 20
month(d)

1702 ## [1] 1
year(d)

1703 ## [1] 2020
1704 Oder auch Zeiteinheiten hinzufügen oder abziehen.
d + days(10)

1705 ## [1] "2020-01-30"
d - years(20)

1706 ## [1] "2000-01-20"
d + hours(25)

1707 ## [1] "2020-01-21 01:00:00 UTC"

```

1708

---

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

---

- 1711 • 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.
- 1712
- 1713 • Extrahieren Sie nun aus jedem Element aus d das Jahr und die Stunde.
- 1714 • Fügen zu jedem Element in d 10 Tage hinzu.

**11.1 Arbeiten mit Zeitintervallen**

1716 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)
```

1718 Wir können jetzt mit `int` arbeiten und beispielsweise das Intervall verschieben,

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

1719 ## [1] 2023-03-18 UTC--2024-03-18 UTC

1720 die Länge des Intervalls berechnen

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

1721 ## [1] 31536000

1722 oder testen, ob ein Datum innerhalb des Intervalls liegt.

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

1723 ## [1] TRUE

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

1724 ## [1] FALSE

1725 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)
```

1726 ## # A tibble: 1 x 1

1727 ## a

1728 ## <date>

1729 ## 1 2020-07-01

1730 `%within%` funktioniert genauso mit Vekotren oder mit mehreren Intervallen. Wir könnten also zwei Intervalle  
1731 definieren (z.B. Ostern und Pfingsten).

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

1732 Und Überprüfen welche Termine in eines der zwei Intervalle fallen.

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

```
Ostern
```

```
termine %within% ostern
```

1733 ## [1] FALSE TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE

```
Pfingsten
termine %within% pfingsten

1734 ## [1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE TRUE FALSE FALSE
1735 Der Zeit-Datentyp wird auch oft benutzt, um den Zeitbedarf von längeren Berechnungen zu messen.

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

1736 ## [1] 0.4999484
t2 <- now()
int_length(interval(t1, t2))

1737 ## [1] 0.257154
```

## 1738 11.2 Formatieren von Zeit

1739 Für die Ausgabe in Berichten oder Grafiken soll das Datum oft in einer speziellen Form dargestellt werden.  
 1740 Die Funktion `format()` bietet Möglichkeiten ein Datumsobjekt zurück in Text umzuwandeln.  
 1741 Ein Beispiel wäre `ymd("2021-2-10")` als 10.2.21 auszugeben.

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

1742 ## [1] "21.02.21"

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

1745

## 1746 Aufgabe 35: Arbeiten mit Intervallen

---

1748 Wie viele Einträge aus dem Vektor `v1` befinden sich in einem Intervall, das zwischen dem 1.3.2021 und dem  
 1749 5.3.2021 definiert ist.

```
v1 <- c(
 "2021-03-05", "2021-03-03", "2021-03-09", "2021-03-09", "2021-03-09",
 "2021-03-03", "2021-03-08", "2021-03-10", "2021-03-07", "2021-03-10"
)
```

## 1750 11.3 Zeitreihen

1751 In der Forstbranche haben wir oft mit Zeitreihen zu tun. Zeitreihen sind Variablen, für die in zeitlichen  
 1752 Intervallen Daten vorliegen. In der Regel monatlich, vierteljährig oder jährlich, wobei die Intervalle zwischen  
 1753 den Messungen bei Zeitreihen immer gleich lang sind. Wiederholungsmessungen von Forsteinventuren (Forstein-  
 1754 richtungen, Betriebsinventuren, die meisten forstl. Experimente, die BWI, ...) sind also in der Regel keine

1755 Zeitreihen in engeren Sinne. Turnusmäßig vermessene Versuchsflächen, wie sie z. B. die Versuchsanstalten  
 1756 unterhalten oder jährlich gemeldete Holzpreise jedoch schon.

1757 Zeitreihen unterscheiden sich nicht nur technisch, sondern auch inhaltlich fundamental von den uns schon  
 1758 bekannten Daten. Statistisch gesehen haben Zeitreihen Besonderheiten gegenüber anderen Variablen, da  
 1759 Sie von Ihrer eigenen Vergangenheit abhängen (autokorriert sind) und auch die Abhängigkeit anderer  
 1760 Variablen in der Regel nur sinnvoll ist, wenn deren Vergangenheit mitberücksichtigt wird (cross-correlation).  
 1761 Konventionelle Statistik ist oft nicht möglich, um Zeitreihen zu analysieren. Selbst ein ordinärer arithmetischer  
 1762 Mittelwert ist schon nicht mehr geeignet, um Zeitreihen statistisch zu beschreiben. Angefangen mit der  
 1763 Datendarstellung gibt es in R deshalb spezifische Zeitreihen-Funktionen. Aus diesem Grund sollten Sie  
 1764 Zeitreihen als solche speichern. Viele Funktionen erkennen den Datentyp und führen entsprechend spezifische  
 1765 Zeitreihen-Operationen durch, wenn ihnen Daten vom Typ "Zeitreihe" übergeben werden. Laden wir z. B.  
 1766 die Holzpreise für Fichte 2b (das sog. Leitsortiment, Fichenholz mit einem Mittendurchmesser von 20 bis 25  
 1767 cm), das Holzaufkommen dieses Sortiments (Einschlagsvolumen) und die Preise für Nadelholz vom  
 1768 statistischen Bundesamt<sup>14</sup>. Wir laden die Daten zunächst als csv ein:

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

1769 Diese 3 Zeitreihen bilden zusammen ein klassisches Marktmodell mit dem Preis eines homogenen Gutes  
 1770 (Leitsortimentspreis), dem Angebot (Holzeinschlag) und der Nachfrage (Schnittholzpreis). Mit der Funktion  
 1771 **ts** werden die Daten in ein Zeitreihenobjekt überführt (*pasrse*). Die Spalte mit den Jahren ist dann nicht mehr  
 1772 nötig, da die Informationen zur Zeit keine eigene Variable mehr sind, sondern als sog. Metainformationen in  
 1773 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])
```

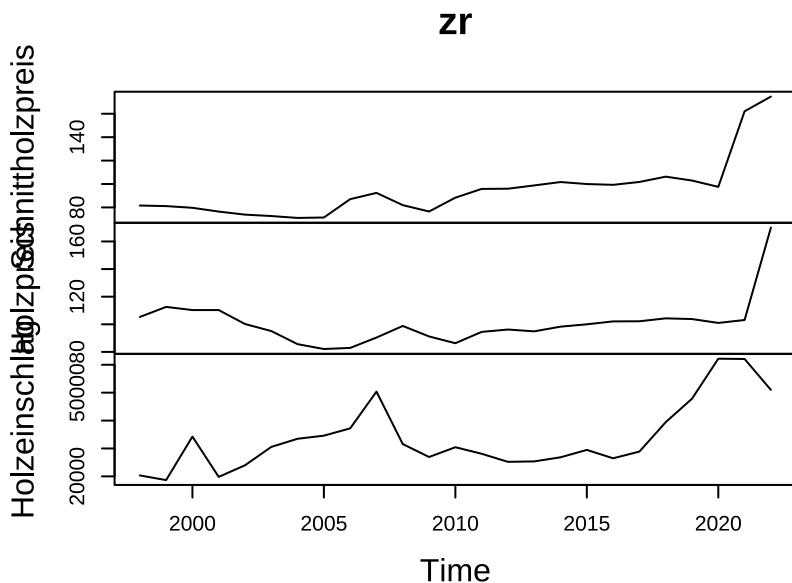
**typeof(zr)** # Zeitreihen sind sog. S3 Objekte. Das heißt, Zeitreihen gehören nicht zu den

1774 ## [1] "double"  
 # Basis Datentypen (siehe Kapitel Variablen, Funktionen und Datentypen),  
 # sondern sind eine Unterkategorie des Datentyps "Liste".

1775 Die wichtigsten Argumente sind - **data** Vektor oder Matrix, der nur die Daten enthält - **start** Startzeitpunkt -  
 1776 **frequency** Anzahl der Beobachtungen pro Zeiteinheit, also z. B. 12 bei monatlichen oder 4 bei quartalsweisen  
 1777 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

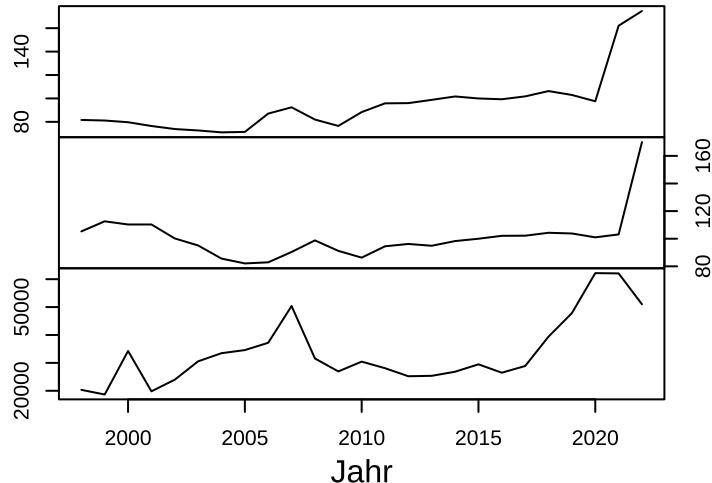


1778

1779 Wir können low-level Layer hinzufügen. Die meisten base plot Funktionen funktionieren auch für Zeitreihen

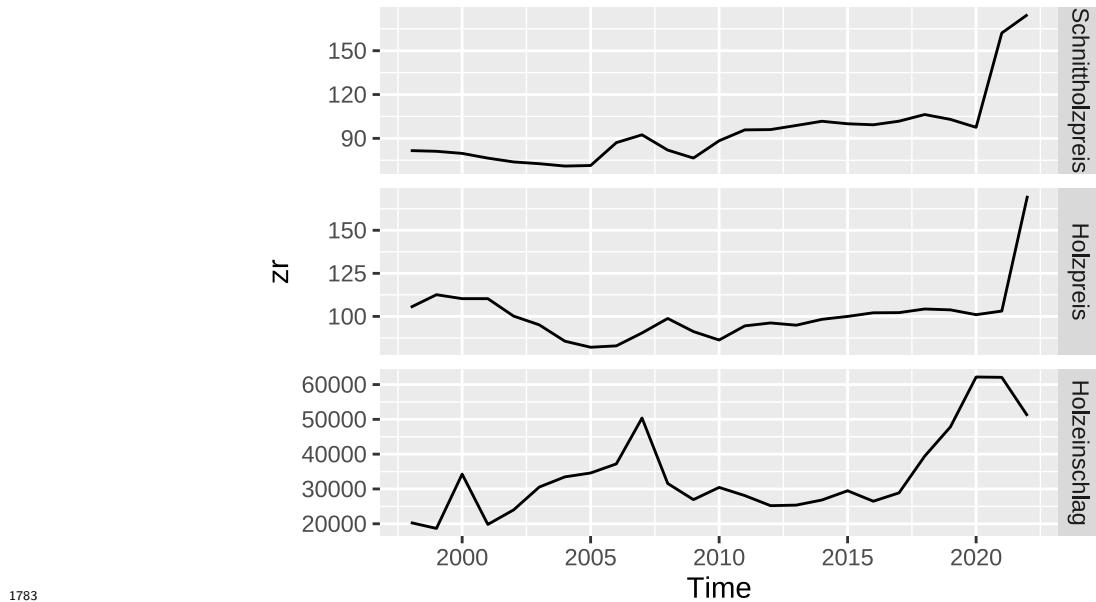
```
plot(zr, ann = FALSE, yax.flip = TRUE)
mtext(side = 3, cex = 1.5, font = 2, "Holzmarktentwicklung seit 1998", line = 2)
mtext(side = 1, cex = 1, line = 2.5, "Jahr")
```

## Holzmarktentwicklung seit 1998



1781 Beide Plot-Philosophiebn haben eine Zeitreihen-Funktion. Das Paket `ggfortify` ermöglicht automatisierte  
1782 Zeitreihenplots im `ggplot2` Stil. Damit ist auch das Problem der y-Achsenbeschriftungen gelöst.

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



1783

- 1784 Wir können die Abbildung im `ggplot2` Stil ändern. Hier nur ein Minimalbeispiel zur Veranschaulichung.  
 1785 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

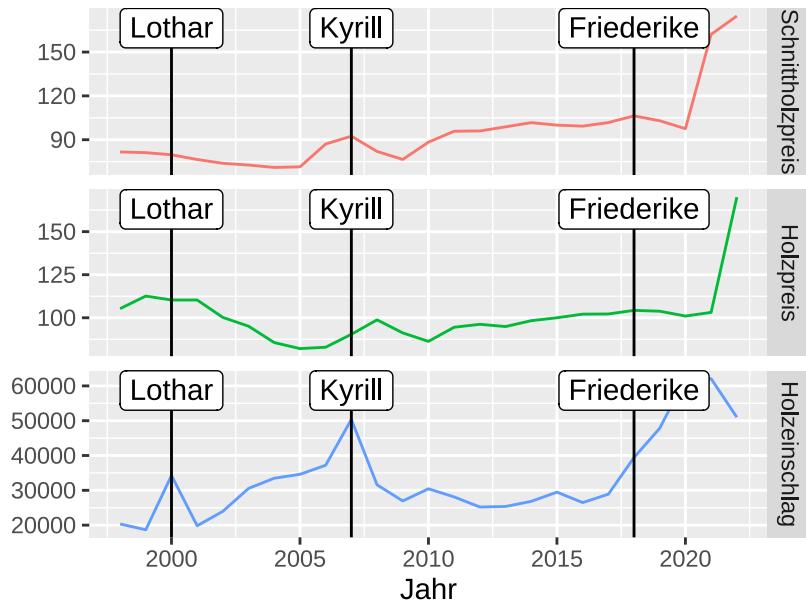
zr_autoplot + theme_minimal()
```

1786

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

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

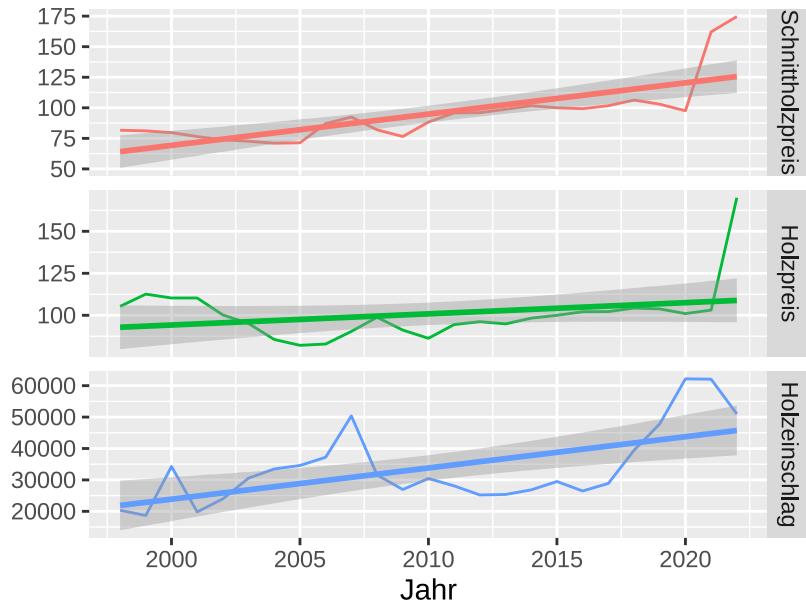
```
annotate(x = 2007, y = +Inf, label = "Kyrill", vjust = 1, geom = "label") +
 annotate(x = 2018, y = +Inf, label = "Friederike", vjust = 1, geom = "label")
```



1787

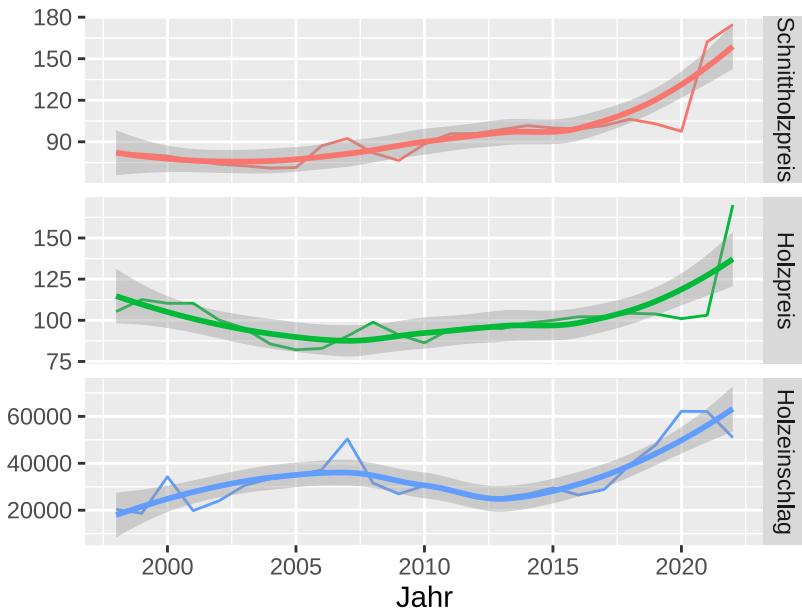
1788 Eine Trendlinie macht hier offensichtlich keinen Sinn. Die Trendlinie ist eine lineare Regression, also eine  
 1789 ordinäre Statistik, die wie eingangs erwähnt für Zeitreihen ungeeignet ist. Daher verwenden wir den sog.  
 1790 Loess-Glättter, um einen Trend in die Daten zu legen. Der Loess-Glättter ist ein sehr flexible Kurve. Wir sehen  
 1791 hier beispielsweise, dass der Leitholzpreis träge oder gar nicht auf das Angebot reagiert. Die Nachfrage jedoch  
 1792 zumindest in der einen Periode, in der sie stark steigt, den Holzpreis jedoch mit zeitlichem Verzug stark  
 1793 ansteigen lässt. Dieser visuelle Eindruck lässt sich durch spezifische Zeitreihen-Regressionen schätzen.

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



1794

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



1795

## 1796 12 Aufgaben Wiederholen (for-Schleifen)

1797 Um einfache Programme zu schreiben, müssen Sie den Ablauf der Programmcodes kontrollieren können.  
 1798 Kontrollieren bedeutet in diesem Zusammenhang, dass Codeabschnitte nur unter definierten Bedingungen ab-  
 1799 laufen. Sie programmieren also zwei Sachen. 1) den Code selbst und 2), die Bedingungen die erfüllt sein müssen,  
 1800 damit der Code ausgeführt wird. Der Code muss so generisch geschrieben sein, dass er komplett durchläuft,  
 1801 auch wenn unterschiedliche Dateneingaben gemacht werden. Diese Kontrollbedingungen ermöglichen es Ihnen  
 1802 generisch zu programmieren. Sie schreiben Ihren Code also nicht speziell maßgeschneidert für ein Problem,  
 1803 sondern so generell, dass er für mehrere Auswertungen funktioniert. Um dies zu gewährleisten, müssen Sie  
 1804 bestimmte Situationen vorhersehen und abfangen. Hierbei helfen Ihnen Kontrollstrukturen (**Control Flow**).  
 1805 Grundsätzlich gibt es **Control Flow** Funktionen zur Wiederholung von Codeblöcken (Schleifen) und logische  
 1806 Bedingungen (bedingte Anweisung).

### 1807 12.1 Schleifen

1808 Bis jetzt wurden alle Skripte einfach der Reihe nach abgearbeitet und zwischendurch bestimmte Programmteile,  
 1809 je nach Situation, selbstständig ausgeführt oder übersprungen. Mit einer Schleife kann man erreichen, dass  
 1810 eine Gruppe von Befehlen (der sog. Schleifenrumpf) mehrfach abgearbeitet wird, zum Beispiel wenn bestimmte  
 1811 Auswertungsschritte auf mehrere Datensätze oder Variablen angewendet oder Funktionen mit unterschiedlichen  
 1812 Parametern oder Startwerten aufgerufen werden sollen. Weitere Anwendungsmöglichkeiten sind iterative  
 1813 Algorithmen, in denen die Eingabewerte des aktuellen Iterationsschrittes von einem vorherigen abhängig sind.  
 1814 Besonders in Simulationen kommen Schleifen häufig zum Einsatz, da große Anzahlen von Wiederholungen  
 1815 benötigt werden.

1816 Man unterscheidet zwischen zwei Arten von Schleifen: Bei den **for()**-Schleifen steht die Anzahl der Wieder-  
 1817 holungen schon beim Eintritt in die Schleife fest, während die **while()**-Schleifen so lange ausgeführt werden,  
 1818 bis eine Bedingung nicht mehr wahr ist. Mit der Funktion **break** wird eine Schleife abgebrochen und die  
 1819 Programmausführung wird nach der Schleife fortgesetzt.

1820 Die wesentlichen Befehle sind

- 1821 • **for (i in X) {Code}**

1822 Wiederhole den Code im “Schleifenrumpf” für jedes Element aus X.

- 1823 • **while(Bedingung) {Code}**

1824 Wiederhole den Code im “Schleifenrumpf” so lange die logische Bedingung erfüllt ist.

- 1825 • **break()**

1826 Brich die Schleife ab. **break()** muss im Schleifenrumpf an der richtigen Stelle ausgeführt werden. Gute  
 1827 Praxis ist jedoch, die for oder while Bedingungen, dass kein **break()**nötig ist, da **break()** anfällig für  
 1828 Programmierfehler ist.

#### 1829 12.1.1 Wiederholen von Befehlen mit **for()**.

1830 Steht vor Beginn der Schleife fest wie viele Schleifendurchgänge benötigt werden, wenn zum Beispiel in einer  
 1831 Simulationen 99 Realisierungen erzeugt oder alle Elemente eines Vektors verarbeitet werden sollen, verwendet

1832 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)
}
```

1833 ## [1] 1

1834 ## [1] 2

1835 ## [1] 3

1836 Das **i** steht in diesem Beispiel für die Schleifen-Variable. Sie muss nicht **i** heißen, sondern kann jeden  
 1837 zulässigen Namen annehmen. Das **X** steht für einen existierenden Vektor oder eine existierende Liste bzw.  
 1838 einen Ausdruck, der ein solches Objekt liefert (der Objektname ist ebenfalls frei wählbar). **for** und **in** sind  
 1839 Schlüsselworte, sie müssen, ebenso wie die runden Klammern, vorhanden sein.

1840 Im ersten Durchgang erhält die Schleifen-Variable **i** den ersten Wert von **X** und der Schleifenrumpf wird  
 1841 mit diesem Wert ausgeführt. Die Variable **i** nimmt nacheinander so lange die Werte von **X** an, bis ihr alle  
 1842 Elemente zugewiesen wurden.

1843 Das folgende Beispiel wird zwar besser durch die entsprechende Vektoroperation gelöst, zeigt aber sehr  
 1844 deutlich die Arbeitsweise der **for**-Schleife.

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

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

1845 ## [1] 4

1846 ## [1] 9

1847 ## [1] 25

1848

#### 1849 Aufgabe 36: Schleifen 1

---

1851 Verwenden Sie den Vektor **k** <- c(1, 3, 9, 12, 15) und schreiben Sie folgende **for**-Schleifen:

- 1852 1. Eine Schleife, die jedes Element aus **k** ausgibt.
- 1853 2. Eine Schleife, die zu jedem Element aus **k** 10 addiert und den neuen Wert ausgibt.
- 1854 3. Eine Schleife wie in 2), aber der neue Wert (**k** + 10) soll jetzt nicht mehr ausgegeben werden, sondern  
 1855 in **k10** gespeichert werden. Stellen Sie sicher, dass **k10** wieder von der Länge 5 ist.

1856

---

1857 Die Funktion **for()** ermöglicht es, einen Befehl beliebig oft zu wiederholen. Z.B. der folgende Ausdruck zieht  
 1858 10-Mal eine Stichprobe der Größe 1 aus dem Vektor **v**. Beachten Sie, dass die Schleifen-Variable **i** selbst gar

1859 nicht im Schleifenrumpf vorkommt. Das Ziel dieser Schleife ist nicht die Elemente des Vektors abzuarbeiten,  
 1860 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))
}
```

```
1861 ## [1] 3
1862 ## [1] 1
1863 ## [1] 3
1864 ## [1] 3
1865 ## [1] 2
1866 ## [1] 3
1867 ## [1] 2
1868 ## [1] 2
1869 ## [1] 1
1870 ## [1] 4
```

1871 Auf gleiche Weise kann man auch über die Variablen eines Dataframes iterieren<sup>15</sup>. Das folgende Beispiel hat  
 1872 zum Ziel die Funktionsweise von Schleifen zu verdeutlichen. Schleifen haben in R jedoch den Nachteil, dass sie  
 1873 sehr langsam operieren. Wenn es geht, sollte man Alternativen verwenden. Die Funktionsweise wiederholender  
 1874 Auswertungen wird jedoch mit `for`-Schleifen deutlicher. Aus diesem Grunde werden wir uns in diesem Kurs  
 1875 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.

1884

---

1885 **Aufgabe 37: for-Schleife**

---

18861887 Lesen Sie den Datensatz `bhd_1.txt` ein und verwenden Sie eine `for`-Schleife.

- 1888 • Ziehen Sie 500-Mal je 35 Beobachtungen für den BHD.
- 1889 • Berechnen Sie jeweils den Mittelwert aus den 35 Werten.
- 1890 • Speichern Sie die 500 Mittelwerte in einem neuen Vektor `mittel`.
- 1891 • Wie ist die Verteilung dieser 500 Mittelwerte zu interpretieren?

1892 **12.1.2 Wiederholen von Befehlen mit `while()`**

1893 Die `while`-Schleifen finden Anwendung, wenn die Anzahl der zu durchlaufenden Wiederholungen vorher  
1894 nicht bekannt ist, wie zum Beispiel bei Iterationsverfahren, die bis zu einer gewissen Genauigkeit durchlaufen  
1895 werden sollen. Die `while`-Schleife besteht in R aus dem Schlüsselwort `while()` und einer Bedingung in runden  
1896 Klammern.

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

1897 Sie ist in der praktischen Programmierung nicht so relevant wie die `for`-Schleife. Sie sei deshalb hier nur  
1898 kurz erwähnt. Die Abbruchbedingung wird jedes Mal geprüft bevor der Schleifenrumpf durchlaufen wird. Die  
1899 Bedingung wird ausgewertet und wenn diese `TRUE` ist, wird der Schleifenrumpf ausgeführt und danach erneut  
1900 die Bedingung überprüft. Ist die Bedingung nicht erfüllt, wird der Schleifenrumpf nicht durchgeführt und die  
1901 Schleife beendet. Ist die Bedingung bereits vor Eintritt in die Schleife nicht erfüllt, wird die Schleife gar nicht  
1902 erst durchlaufen.

1903 Da `while`-Schleifen also so lange ausgeführt werden, bis die Bedingung nicht mehr erfüllt ist, kann eine  
1904 Endlosschleife entstehen. Dies kann passieren, wenn man nicht sauber programmiert hat. Wenn innerhalb der  
1905 Schleife nicht dafür gesorgt wird, dass die Bedingung irgendwann nicht mehr erfüllt wird, so läuft die Schleife  
1906 immer weiter. Steckt R in einer Schleife fest und reagiert nicht mehr, kann der Befehl unter Linux mit `Strg`+  
1907 `C` und unter Windows mit `Esc` abgebrochen werden. Alternativ können Sie auf das rote STOP Symbol  
1908 über der Konsole klicken.

1909 **12.2 Bedingte Ausführung von Codeblöcken**

1910 Innerhalb eines Skripts ist es mitunter notwendig je nach aktueller Situation unterschiedlich fortzufahren.  
1911 Die Situation wird mit einem logischen Ausdruck, einer sogenannten Bedingung, geprüft. Je nachdem, ob  
1912 die Bedingung wahr (`TRUE`) oder falsch (`FALSE`) ist, werden unterschiedliche Programmteile ausgeführt, der  
1913 jeweils andere Teil bleibt unberücksichtigt. Danach wird in jedem Fall die Programmausführung, mit den  
1914 auf die bedingte Anweisungen folgenden Anweisung, fortgesetzt. In R kann dies mit dem if-else-Konstrukt  
1915 realisiert werden, welches aus den Schlüsselwörtern `if()` und `else` sowie der in runde Klammern gefassten  
1916 Bedingung besteht.

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

1917 Im folgenden Beispiel sollen die bisher abstrakt beschriebenen Vorgänge praktische Anwendung finden. In  
 1918 dem Beispiel wird zunächst durch zufälliges Ziehen einer Zahl aus dem Bereich eins bis sechs ein Würfelwurf  
 1919 simuliert. Anschließend wird der Würfelwurf mit einem if-Ausdruck kommentiert. Ist die Bedingung, es wurde  
 1920 eine Sechs gewürfelt, erfüllt, wird der Code innerhalb der geschweiften Klammern ausgeführt, ansonsten wird  
 1921 der Klammerinhalt ignoriert.

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

1922 In den meisten Fällen ist es für R irrelevant, ob sich zwischen den verschiedenen Elementen Leerzeichen oder  
 1923 Zeilenumbrüche befinden. Bei dem else-Ausdruck dagegen wird ein Fehler erzeugt, wenn vor dem `else` nicht  
 1924 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.")
}
```

1925 ## [1] "Beim nächsten Wurf klappt's bestimmt."

1926

### 1927 Aufgabe 38: Bedingte Programmierung

---

- 1929 • Wenn keine 6 gewürfelt wurde, lassen Sie zusätzlich ausgeben welche Zahl gewürfelt wurde.  
 1930 • Wiederholen Sie den Würfelwurf 10 Mal.

## 1931 13 (R)markdown

### 1932 13.1 Markdown Grundlagen

1933 Die Idee von Markdown ist, einfach Text strukturieren zu können und das ganze ohne umfangreiche Programme  
 1934 zu erstellen. Im nächsten Abschnitt sehen wir dann, wie man Markdown und R-Code verbinden kann. Hier  
 1935 soll es jedoch erst einmal darum gehen, die einfachsten Bausteine von Markdown vorzustellen.

1936 Am Anfang jedes Dokuments kommt eine Präambel. Diese fängt mit --- an und hört auch wieder mit ---  
 1937 auf. Innerhalb der Präambel können dann Metainformationen über das Dokument festgelegt werden, dies  
 1938 beinhaltet im einfachsten Fall: Titel, Autor und Datum. Das würde dann so aussehen:

```
1939 ---
1940 title: "Ein Titel"
1941 author: "Der, der es geschrieben hat"
1942 date: "März 2021"
1943 ---
```

1944 Danach folgt strukturierter und formatierter Text. Verschiedene Hierarchieebenen von Überschriften können  
 1945 mit der Anzahl an # festgelegt werden. So ist eine Überschrift erster Ordnung # Kapitel eine Überschrift  
 1946 zweiter Ordnung ## Unterkapitel usw.

1947 Listen können erstellt werden, wenn man am Anfang jeder Zeile ein - oder 1. schreibt.

```
1948 - Erster Eintrag
1949 - Zweiter Eintrag
1950 - Dritter Eintrag
```

1951 wird zu

```
1952 • Erster Eintrag
1953 • Zweiter Eintrag
1954 • Dritter Eintrag
```

1955 Eine zentrale Idee von Markdown ist es Text einfach zu formatieren. Werden eine oder mehrere Wörter mit  
 1956 zwei Sternchen (\*\*) eingefasst wird dieser Text **fett** dargestellt. Also aus \*\*wichtig\*\* wird **wichtig**. Das  
 1957 gleiche funktioniert auch mit *kursiven* Text, jedoch muss man hier noch ein Sternchen verwenden, also aus  
 1958 \*kursiv\* wird *kursiv*. Soll ein text fett und kursiv sein, kann man drei Sternchen verwenden. Aus \*\*\*sehr  
 1959 wichtig\*\*\* wird dann **sehr wichtig**.

1960 Weitere Elemente wie Links oder Abbildungen können einfach eingebunden werden. Links werden mit [Link  
 1961 text](url) in den Text integriert. Beispielsweise ist eine gute Idee bei [stackoverflow](#) bei Problemen nach  
 1962 einer Lösung zu suchen. Dieser Link wurde mit [stackoverflow](www.stackoverflow.com) erstellt.

1963 Für Abbildungen gibt es einen ganz ähnlichen Syntax. Mit ! [Das R Logo](abb/r\_logo.png) wird die  
 1964 Abbildung r\_logo.png eingebunden mit der Beschriftung: "Das R Logo".



Abbildung 12: Das R Logo

1965

---

**Aufgabe 39: Arbeiten mit markdown**


---

1968 Verwenden Sie das folgende Markdowndokument:

1969 ---

1970 title: "Dokument"  
1971 author: "Ihr Name"  
1972 date: "März 2021"

1973 ---

1974

1975 # Einleitung  
1976  
1977 # Methoden

- 1978 1. Kopieren Sie die Vorlage in ein Dokument, das `test.md` heißt.
- 1979 2. Fügen Sie zwei Überschriften zweiter und dritter Ordnung hinzu.
- 1980 3. Fügen Sie einen *kursiven* Text hinzu.
- 1981 4. Fügen Sie einen Link zu einer Website hinzu.
- 1982 5. Kompilieren Sie die Datei, indem Sie in Rstudio auf **Preview** drücken (Abbildung 13).

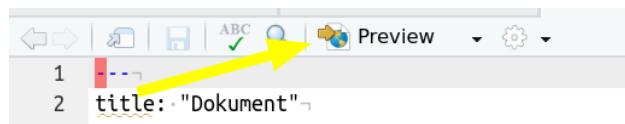


Abbildung 13: Kompilieren einer md-Datei.

**13.2 R und Markdown**

1984 Markdown macht es bereits einfach Textdokumente und Dokumentationen zu verfassen, aber die wirkliche  
 1985 Stärke liegt in der Möglichkeit R und Markdown zu kombinieren. Man spricht dann von Rmarkdown. Ein  
 1986 weiteres Strukturelement, das wir noch nicht kennen gelernt haben, sind Code-Blöcke.

1987 ~~~

1988 a &lt;- 1:10

```

1989 a[1]
1990 ``
1991 erzeugt
1992 a <- 1:10
1993 a[1]

1994 Momentan wird noch kein Code ausgeführt, sondern lediglich der Code, als Code dargestellt. Rmarkdown
1995 bietet nun die Möglichkeit Code beim kompilieren16 auszuführen. Dafür müssen wir nur einen Code-Block als
1996 R-Code-Block kennzeichnen.

1997 ``{R}
1998 a <- 1:10
1999 a[1]
2000 ```

2001 erzeugt
2002 a <- 1:10
2003 a[1]

2004 ## [1] 1

2005 Beachte, die Variable a wird beim kompilieren erzeugt und steht dann R zur Verfügung. R-Code-Blöcke
2006 werden auch als Code Chunks bezeichnet. Diese Chunks können sehr genau angesprochen und angepasst
2007 werden. Einige wichtige Argumente sind:
2008

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

```

2009

---

**Aufgabe 40: Arbeiten mit Rmarkdown**


---

2010 Erstellen Sie eine neue Rmarkdown Datei mit dem Namen `test1.Rmd`. Erstellen Sie zwei Code-Chunks. Der  
2011 erste soll nicht angezeigt werden und darin werden die Daten geladen (`bhd_1.txt`). Im zweiten Chunk plotten  
2012 Sie das Alter der Bäume gegen den BHD. Was passiert mit dem Plot, wenn Sie die Datei kompilieren (drücken  
2013 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.

## 2016 14 Räumliche Daten in R

### 2017 14.1 Was sind räumliche Daten

2018 Räumliche Daten sind Beobachtungen, wie wir sie schon oft gesehen haben, mit einem räumlichen Bezug. Der  
 2019 Unterschied zu nicht räumlichen Daten liegt darin, dass räumliche Daten eindeutig im *Raum* verortet werden  
 2020 können. Häufig werden sogenannte Geoinformationssysteme (GIS) zum Arbeiten mit räumlichen verwendet.  
 2021 R kann in vielerlei Hinsicht wie ein GIS eingesetzt werden und hier werden einige Grundfunktionalitäten  
 2022 dafür besprochen. Räumliche Daten werden in zwei unterschiedliche Datentypen unterteilt: Vektor- und  
 2023 Rasterdaten. Vektordaten modellieren einzelne Objekte (= *Features*). Rasterdaten modellieren eine Oberfläche.  
 2024 Vektordaten bestehen aus zwei Komponenten: 1) einer Geometrie, die die Form und Lage der Daten definiert  
 2025 und 2) Attributen, den tatsächlichen Daten. Räumliche Daten werden oft als *Features* bezeichnet. Ein Feature  
 2026 ist die räumliche Einheit einer Beobachtung. Je nach Art der räumlichen Daten können Features entweder  
 2027 Punkte (z.B. ein Baum), Linien (z.B. eine Straße) oder Polygone (z.B. ein See) sein. Auch können mehrere  
 2028 Geometrien zu einem Feature zusammengefasst werden. Ein Beispiel wäre eine Beobachtung für ein Land,  
 2029 das aber aus mehreren Polygonen bestehen kann (z.B. Festland und Inseln). Features können dann weitere  
 2030 Attribute (= Attributdaten) haben, z.B. eine ID, Name oder was auch immer man gemessen hat.  
 2031 Rasterdaten bestehen aus einer Oberfläche von gleichgroßen Kacheln (= *Pixel*), die ein Gebiet abdecken.  
 2032 Meist sind Pixel viereckig, aber das ist keine Voraussetzung. Dabei hat jedes Pixel einen Wert (das kann  
 2033 auch ein fehlender Wert sein). Typische Beispiele für Rasterdaten sind Landnutzung oder Seehöhen.  
 2034 In R kann sowohl mit Vektor- als auch mit Rasterdaten gearbeitet werden. Für Vektordaten bietet sich das  
 2035 Paket **sf** an und für Rasterdaten das Paket **raster**.

### 2036 14.2 Koordinatenbezugssystem

2037 Eine der Herausforderungen für räumliche Daten ist die eindeutige Verortung im Raum. Dazu braucht man  
 2038 ein *Koordinatenbezugssystem (KBS)*. Einem KBS liegt ein mathematisches Modell der Erde zugrunde. Die  
 2039 Details zu KBS werden schnell relativ kompliziert und wir beschränken uns hier lediglich darauf, wie KBS  
 2040 verwendet werden können. Dazu müssen wir zwei Fälle unterscheiden: 1) einem Datensatz ein KBS zuweisen  
 2041 und 2) Transformation des KBS eines Datensatzes in ein anderes KBS. Die technischen Details werden in  
 2042 den folgend Abschnitten besprochen. Für beide Aufgaben müssen wir auf ein KBS verweisen können, ein  
 2043 einfacher Ansatz dafür sind die sogenannten *EPSG-Codes*<sup>17</sup>.

### 2044 14.3 Vektordaten in R

2045 Das Paket **sf** stellt Klassen zum Abbilden von Features zur Verfügung, die dann in einem **data.frame** als  
 2046 Liste gespeichert werden können. In der Regel erstellen wir Features nicht individuell, sondern lesen diese aus  
 2047 externen Datenquellen (z.B. ESRI Shapefile) ein. Zum besseren Verständnis, erstellen wir es einmal manuell.  
 2048 Wir haben die Koordinaten für drei Städte (Göttingen, Hannover und Berlin) als geografische Koordinaten  
 2049 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))
```

2050 Daraus können wir jetzt eine Geometriespalte für einen `data.frame` erstellen

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

2051 Somit haben wir die Geometrie in der Variable `geom` gespeichert, aber noch keine dazugehörigen Attributdaten.

2052 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)
)
```

2053 In einem letzten Schritt möchten wir jetzt die Geometrie (`geom`) und die Attributdaten (`attr`) zusammenführen.

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

```
2055 ## Simple feature collection with 3 features and 3 fields
2056 ## Geometry type: POINT
2057 ## Dimension: XY
2058 ## Bounding box: xmin: 9.732 ymin: 51.5413 xmax: 13.405 ymax: 52.52
2059 ## Geodetic CRS: WGS 84
2060 ## name bundesland einwohner geom
2061 ## 1 Goettingen Niedersachsen 119000 POINT (9.9158 51.5413)
2062 ## 2 Hannover Niedersachsen 532000 POINT (9.732 52.3759)
2063 ## 3 Berlin Berlin 3650000 POINT (13.405 52.52)
```

2064 Wir können nun mit `staedte` genau so arbeiten wie mit jedem anderen `data.frame` und die Geometrien werden immer ‘berücksichtigt’. Zusätzlich kann man eine Reihe von geometrischen Operationen durchführen.

2066 Wenn ein `data.frame` Punkten hat, kann man dies relative einfach mit der Funktion `st_as_sf()` “räumlich” machen. Für das vorherige Beispiel würden wir zuerst einen `data.frame` mit allen Informationen (zur Geometrie und zu den Attributen erstellen).

```
dat <- data.frame(
 name = c("Goettingen", "Hannover", "Berlin"),
 bundesland = c("Niedersachsen", "Niedersachsen", "Berlin"),
 einwohner = c(119000, 532000, 3650000),
 x = c(9.9158, 9.7320, 13.405),
 y = c(51.5413, 52.3759, 52.5200)
)
```

2069 Dann kann man mit der Funktion `st_as_sf()` weiter arbeiten:

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

## 2070 14.4 Arbeiten mit Vektordaten

2071 Es gibt sehr viele Funktionen, um mit räumlichen Daten zu arbeiten, von denen wir hier einige vorstellen.

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

```
2072 ## Coordinate Reference System:
2073 ## User input: EPSG:4326
2074 ## wkt:
2075 ## GEOGCRS["WGS 84",
2076 ## ENSEMBLE["World Geodetic System 1984 ensemble",
2077 ## MEMBER["World Geodetic System 1984 (Transit)"],
2078 ## MEMBER["World Geodetic System 1984 (G730)"],
2079 ## MEMBER["World Geodetic System 1984 (G873)"],
2080 ## MEMBER["World Geodetic System 1984 (G1150)"],
2081 ## MEMBER["World Geodetic System 1984 (G1674)"],
2082 ## MEMBER["World Geodetic System 1984 (G1762)"],
2083 ## MEMBER["World Geodetic System 1984 (G2139)"],
2084 ## ELLIPSOID["WGS 84",6378137,298.257223563,
2085 ## LENGTHUNIT["metre",1]],
2086 ## ENSEMBLEACCURACY[2.0]],
2087 ## PRIMEM["Greenwich",0,
2088 ## ANGLEUNIT["degree",0.0174532925199433]],
2089 ## CS[ellipsoidal,2],
2090 ## AXIS["geodetic latitude (Lat)",north,
2091 ## ORDER[1],
2092 ## ANGLEUNIT["degree",0.0174532925199433]],
2093 ## AXIS["geodetic longitude (Lon)",east,
2094 ## ORDER[2],
2095 ## ANGLEUNIT["degree",0.0174532925199433]],
2096 ## USAGE[
2097 ## SCOPE["Horizontal component of 3D system."],
2098 ## AREA["World."],
2099 ## BBOX[-90,-180,90,180]],
2100 ## ID["EPSG",4326]]
```

2101 Wenn wir jetzt zu einem anderen KBS (z.B. EPSG:3035, ein europäisches projiziertes KBS) umrechnen  
 2102 möchten, können wir das mit

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

```

2103 ## Coordinate Reference System:
2104 ## User input: EPSG:3035
2105 ## wkt:
2106 ## PROJCRS["ETRS89-extended / LAEA Europe",
2107 ## BASEGEOGCRS["ETRS89",
2108 ## ENSEMBLE["European Terrestrial Reference System 1989 ensemble",
2109 ## MEMBER["European Terrestrial Reference Frame 1989"],
2110 ## MEMBER["European Terrestrial Reference Frame 1990"],
2111 ## MEMBER["European Terrestrial Reference Frame 1991"],
2112 ## MEMBER["European Terrestrial Reference Frame 1992"],
2113 ## MEMBER["European Terrestrial Reference Frame 1993"],
2114 ## MEMBER["European Terrestrial Reference Frame 1994"],
2115 ## MEMBER["European Terrestrial Reference Frame 1996"],
2116 ## MEMBER["European Terrestrial Reference Frame 1997"],
2117 ## MEMBER["European Terrestrial Reference Frame 2000"],
2118 ## MEMBER["European Terrestrial Reference Frame 2005"],
2119 ## MEMBER["European Terrestrial Reference Frame 2014"],
2120 ## ELLIPSOID["GRS 1980",6378137,298.257222101,
2121 ## LENGTHUNIT["metre",1]],
2122 ## ENSEMBLEACCURACY[0.1]],
2123 ## PRIMEM["Greenwich",0,
2124 ## ANGLEUNIT["degree",0.0174532925199433]],
2125 ## ID["EPSG",4258],
2126 ## CONVERSION["Europe Equal Area 2001",
2127 ## METHOD["Lambert Azimuthal Equal Area",
2128 ## ID["EPSG",9820]],
2129 ## PARAMETER["Latitude of natural origin",52,
2130 ## ANGLEUNIT["degree",0.0174532925199433],
2131 ## ID["EPSG",8801]],
2132 ## PARAMETER["Longitude of natural origin",10,
2133 ## ANGLEUNIT["degree",0.0174532925199433],
2134 ## ID["EPSG",8802]],
2135 ## PARAMETER["False easting",4321000,
2136 ## LENGTHUNIT["metre",1],
2137 ## ID["EPSG",8806]],
2138 ## PARAMETER["False northing",3210000,
2139 ## LENGTHUNIT["metre",1],
2140 ## ID["EPSG",8807]]],
2141 ## CS[Cartesian,2],
2142 ## AXIS["northing (Y)",north,
2143 ## ORDER[1],
2144 ## LENGTHUNIT["metre",1]],
2145 ## AXIS["easting (X)",east,

```

```

2146 ## ORDER[2] ,
2147 ## LENGTHUNIT["metre",1]],
2148 ## USAGE[
2149 ## SCOPE["Statistical analysis."],
2150 ## AREA["Europe - European Union (EU) countries and candidates. Europe - onshore and offshore: "],
2151 ## BBOX[24.6,-35.58,84.73,44.83]],
2152 ## ID["EPSG",3035]

```

2153 Die Funktion `st_buffer()` erlaubt es Features zu puffern, mit `st_distance()` kann die Distanz zwischen  
2154 Features berechnet werden, mit `st_area()` kann die Fläche eines Features berechnet werden.

2155 Funktionen wie `st_intersection()`, `st_union()` und `st_difference()` erlauben es geometrische Opera-  
2156 tionen zwischen unterschiedlichen Features zu berechnen. Für eine ausführliche Diskussion siehe auch hier:  
2157 <https://geocompr.robinlovelace.net/geometric-operations.html>.

2158 Normalerweise lesen wir Daten von externen Datenquellen (z.B. ESRI Shapefiles). Das geht mit der Funktion  
2159 `st_read()`.

## 2160 14.5 Rasterdaten in R

2161 Für Rasterdaten gibt es das R-Paket `terra`. Auch hier wollen wir uns wieder auf einige Grundfunktionalitäten  
2162 konzentrieren. Diese umfassen das Einlesen, Zuschneiden, Rechnen und Abfragen von Rastern.

2163 Mit der Funktion `rast()` kann ein Raster in R eingelesen werden.

```

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

```

2164 `dem` steht für *Digital Elevation Model* und ist ein Raster mit den Seehöhen in Niedersachsen mit einer  
2165 500-m-Auflösung. Wir können diese mit der Funktion `res()`<sup>18</sup> abfragen.

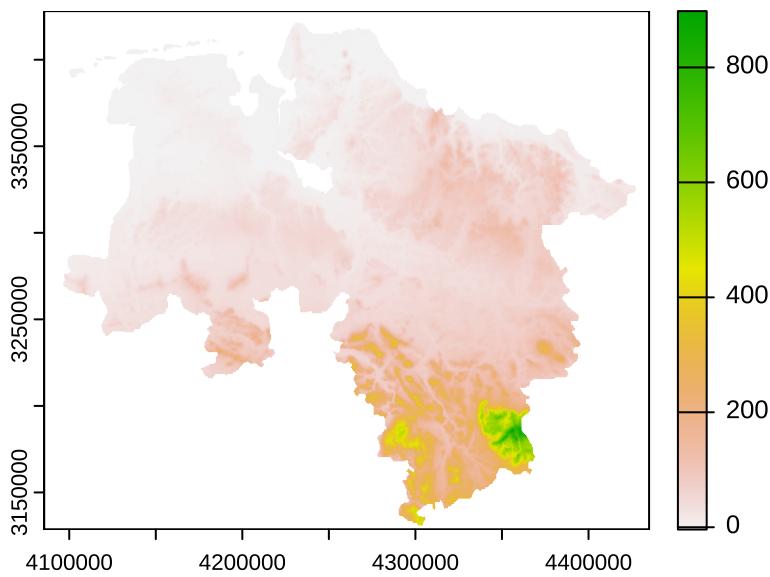
```
res(dem)
```

2166 ## [1] 500 500

2167 Bzw. wir können den Raster auch plotten.

```
plot(dem)
```

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



2168

2169 Wenn wir den Raster `dem` auf ein Gebiet zuschneiden wollen (z.B. Göttingen), müssen wir drei Schritte  
2170 durchführen. Als erstes müssen wir ein Shapefile für Göttingen einlesen.

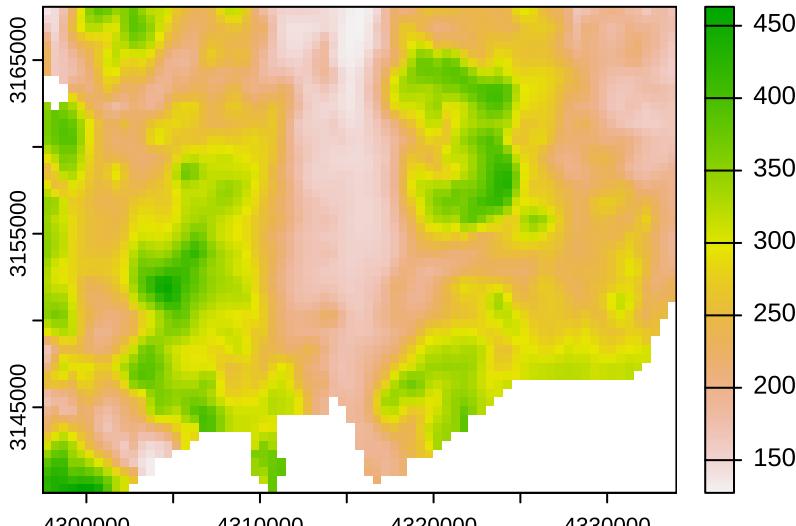
```
goe <- st_read(here:::here("data/goettingen/stadt_goettingen.shp"))
```

2171 Dann müssen wir sicherstellen, dass sowohl der Raster `dem` als auch das `sf`-Objekt `goe` im selben KBS sind.  
2172 Es bietet es sich in der Regel an, das KBS des Vektors zu transformieren. Mit der Funktion `projectRaster()`  
2173 kann das KBS eines Rasters transformiert werden.

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

2174 Mit der Funktion `crop()` kann der Raster `dem` auf Göttingen zugeschnitten werden.

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



2175

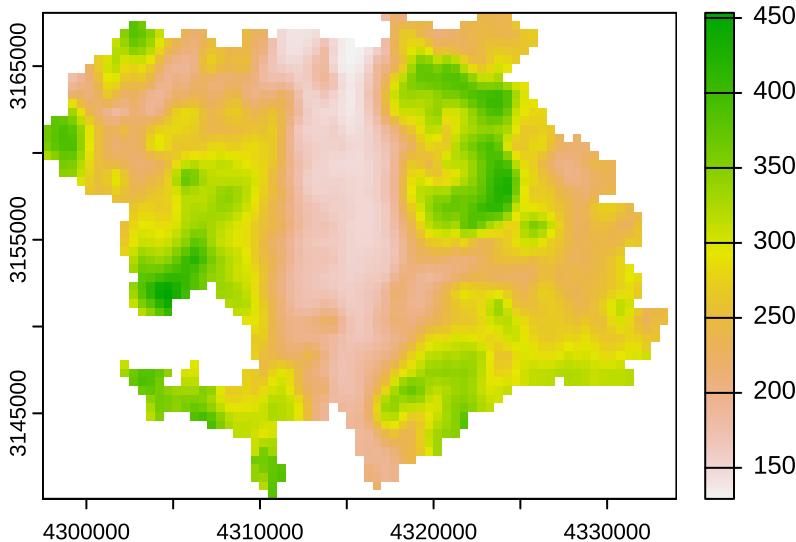
2176 Der Raster hat jetzt die Größe einer Bounding-Box (BBX) von Göttingen (das ist ein Rechteck, das Göttingen  
2177 umfasst). Mit der Funktion `mask()` kann der Raster auf die genauen Grenzen des Vektors `goe` angepasst

2178 werden.

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

2179 ## Warning: [mask] CRS do not match

```
plot(dem2)
```



2180

2181 Wenn wir an bestimmten Punkten den Wert des Rasters abfragen wollen (z.B. an `cities`) von vorhin, dann  
2182 gibt es dafür die Funktion `extract`. Dann müssen wir erst sicherstellen, dass `staedte` und `dem` gleichen KBS  
2183 zu grunde liegt. Dafür transformieren wir einfach `staedte` in das KBS von `dem`. Mit der Funktion `crs()`  
2184 erhalten wir das KBS des Rasters.

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

2185 Wenn wir das KBS eines Objektes nicht kennen, können wir auch einfach das KBS übergeben. Der folgende  
2186 Code-Block macht genau das Gleiche mit dem Vorteil, dass wir keinen EPSG-Code angeben müssen.

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

2187 Dann können wir für jede Stadt die Seehöhe abfragen:

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

2188 ## ID dem\_3035

2189 ## 1 1 149.18181

2190 ## 2 2 57.21486

2191 ## 3 3 NA

2192 Mit `terra::extract()` rufen wir *eindeutig* die Funktion `extract()` aus dem Paket `terra` auf. Wir müssen  
2193 das so machen, weil es im Paket `dplyr` auch eine Funktion `extract()` gibt, die wir hier nicht anwenden  
2194 möchten, da sie einen Fehler verursachen würde.

2195 Ein analoges Vorgehen ist auch für Linien und Polygone möglich.

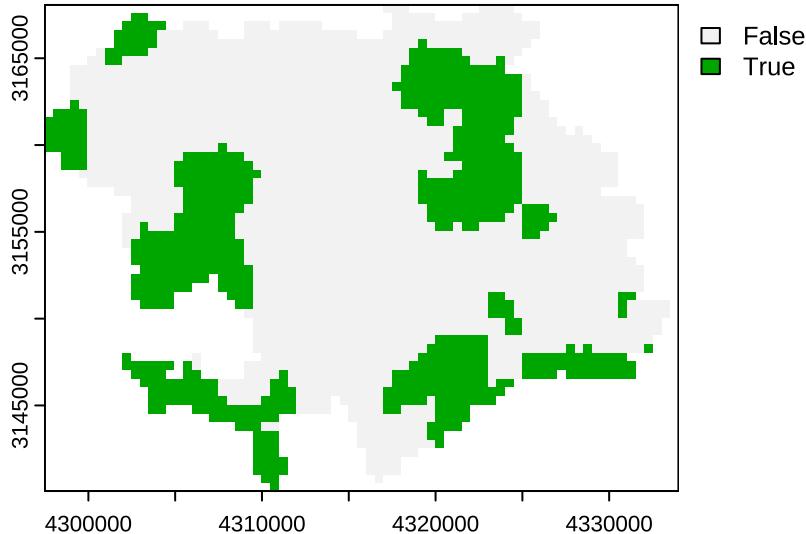
2196 Mit Rastern kann auch einfach gerechnet werden. Wir können z.B. die Seehöhe in Kilometern anstatt Metern

2197 berechnen:

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

2198 Auch logische Operationen sind möglich, wenn wir alle Rasterzellen mit einer Seehöhe von mehr als 300 m in  
2199 Göttingen suchen, dann geht das so:

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



2200

2201 Wenn wir jetzt auf die Werte des Rasters `dem3` zugreifen wollen, geht das mit eckigen Klammern.

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

```
2202 ## dem_3035
2203 ## [1,] NA
2204 ## [2,] NA
2205 ## [3,] NA
2206 ## [4,] NA
2207 ## [5,] NA
2208 ## [6,] NA
```

2209 Das sind erst einmal viele NA-Werte für die ganzen Zellen, die außerhalb von Niedersachsen liegen. Aber wir  
2210 können mit so einem Vektor ganz normal arbeiten und z.B. die Fläche des Landes Niedersachsen die eine  
2211 Seehöhe von mehr als 500m Seehöhe hat ausrechnen.

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

2212 ## [1] 0.2786229

2213

---

2214 **Aufgabe 41: Arbeiten mit Rastern**


---

2215

2216 Verwenden Sie den Raster `wald.tif`, der auf einer 10 m Auflösung den Waldanteil jeder Rasterzelle angibt<sup>19</sup>.  
2217 Der EPSG-Code für das KBS von `wald.tif` ist 3035. Nehmen Sie an, dass wenn der Waldanteil in einer  
2218 Raster größer als 50 % ist, dass die Rasterzelle als Wald klassifiziert werden kann. Wie viel Prozent des  
2219 Göttinger Stadtgebietes sind Wald? Wie ändert sich dieser Wert, wenn sie 70 % anstatt 50 % als Schwellenwert  
2220 für Wald annehmen?

2221

---

2222 **Aufgabe 42: Studiendesign**


---

2223

2224 Mit der Funktion `st_sample()` können Sie innerhalb oder entlang eines Features zufällige Punkte legen. Das  
2225 Argument `n` steuert die Anzahl Punkte und das Argument `type` wie die Punkte angeordnet werden. Für `type`  
2226 sind für uns die Werte `type = "random"` (komplett zufällig), `type = "regular"` (regelmäßiger Grid) und  
2227 `type = "hexagonal"` von Bedeutung (ein hexagonaler Grid, d.h. ein sechseckiger Raster). Unglücklicherweise  
2228 ist das Ergebnis von `st_sample()` erst eine Geometrie. Um daraus ein vollständiges `sf`-Objekt zu machen  
2229 und problemlos weiter arbeiten zu können, müssen Sie noch einmal die Funktion `st_as_sf()` ausführen.  
2230 Stellen Sie sich vor, dass wir die tatsächliche Waldbedeckung des Göttinger Stadtgebietes **nicht** kennen und wir  
2231 eine Studie durchführen, um den Anteil des Göttinger Stadtgebietes, der mit Wald bedeckt ist herauszufinden.  
2232 Erstellen Sie dafür einige unterschiedliche Stichproben (diese können in der Anzahl und Anordnung variieren).  
2233 Berechnen Sie für jedes Stichprobendesign den Anteil an Wald und ein dazugehöriges Konfidenzintervall  
2234 (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  $n$   
2235 die Stichprobengröße). Nehmen Sie an, dass eine Rasterzelle Wald ist, sobald > 50 % der Rasterzelle mit  
2236 Wald bedeckt ist.

2237

---

2238 **Aufgabe 43: Räumliche Daten**


---

2239

2240 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)
)
```

2241 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

- 2242 2. Puffern Sie jeden Baum mit seinem Kronendurchmesser.
- 2243 3. Berechnen Sie die Kronenfläche jedes Baumes. *Hinweis: Die Funktion st\_area() könnte dafür hilfreich sein.*
- 2244 4. Welcher Baum hat die größte Kronenfläche?
- 2245 5. Finden Sie für jede Art, den Baum mit der größten Kronenfläche.

2247

2248 **Aufgabe 44: Arbeiten mit räumlichen Daten**

---

- 2250 1. Lesen Sie das ESRI Shapefile goettingen/stadt\_goettingen.shp ein.
- 2251 2. Wie viele Features befinden sich in dem Shapefile?
- 2252 3. Welches Koordinatenbezugssystem (KBS) hat das Shapefile?
- 2253 4. Transformieren Sie das Shapefile in das KBS 3035.
- 2254 5. Erstellen Sie eine neue Spalte A in der Sie die Fläche jeder Gemeinde/Stadt speichern.
- 2255 6. Welche Gemeinde/Stadt (Spalte GEN) ist am größten?
- 2256 7. Wählen Sie nun nur die Stadt Göttingen aus.

2257

2258 **Aufgabe 45: Arbeiten mit räumlichen Daten 2**

---

- 2260 1. Lesen Sie erneut das ESRI Shapefile goettingen/stadt\_goettingen.shp ein.
- 2261 2. Lösen sie die Gemeindegrenzen auf (die Funktion st\_union() könnte hier nützlich sein).
- 2262 3. Wie groß ist das resultierende Feature?

2263 **15 FAQs (Oft gefragtes)**

2264 **15.1 Arbeiten mit Daten**

2265 **15.1.1 Einlesen von Exceldateien**

2266 Mit der Funktion `read_excel()` aus dem Paket `readxl` können Exceldateien direkt in R eingelesen werden.

2267 Ein Export als csv-Datei aus Excel ist nicht notwendig.

## 2268 16 Literatur

- 2269 Ein guter Überblick über viele der angeschnittenen Themen gibt es in dem [R for data science](#), das online frei  
2270 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.  
2271
- 2272 McNamara, Amelia, and Nicholas J Horton. 2018. “Wrangling Categorical Data in r.” *The American Statistician*  
2273 72 (1): 97–104.  
2274 Wickham, Hadley. 2014. “Tidy Data.” *Journal of Statistical Software* 59 (10). <https://doi.org/10.18637/jss.v059.i10>.  
2275