

1

Einführung in die Datenanalyse mit R (700104)

4

Kursskript

5



6

Dr. Johannes SIGNER
Abteilung Wildtierwissenschaften
Büsgenweg 3
37077 Göttingen

jsigner@uni-goettingen.de

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

kai.husmann@uni-goettingen.de

7



8

Fakultät für Forstwissenschaften und Waldökologie
Georg-August-Universität Göttingen

10



11

12

Wintersemester 2024/2025

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

¹⁵ Zitiervorschlag:

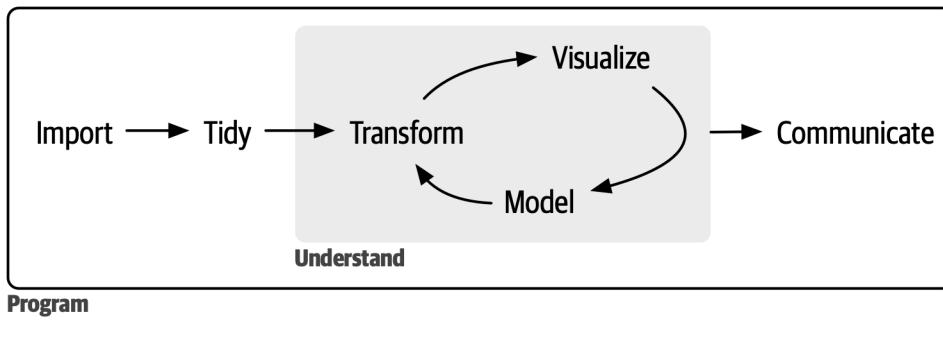
¹⁶ Signer, J. und Husmann, K. (2024) Skript zur Vorlesung Einführung in die Datenanalyse mit R, Georg-August-Universität Göttingen.

¹⁸ Letzte Aktualisierung: 28. November 2024

19 Vorwort und Danksagung

20

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



25

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

45 Inhaltsverzeichnis

46	1 R und RStudio	3
47	1.1 Installation von R und RStudio	3
48	1.2 Erste Schritte in R	3
49	1.3 Gute Praxis bei der Programmierung	5
50	1.4 RStudio Projekte	6
51	1.4.1 Erstellen eines Projektes	6
52	2 Variablen, Funktionen und Datentypen	8
53	2.1 Variablen beim Programmieren	8
54	2.2 Funktionen	10
55	2.3 Datentypen	10
56	2.4 Datenstrukturen	11
57	3 Vektoren	13
58	3.1 Funktionen zum Arbeiten mit Vektoren	15
59	3.2 Statistische Funktionen	16
60	3.3 Beispiel Fotofallen	17
61	3.4 Arbeiten mit logischen Werten	18
62	3.5 Zugreifen auf Elemente eines Vektors (=Untermengen)	19
63	3.6 Der %in%-Operator	21
64	4 Faktoren (factors)	23
65	4.1 Das Paket forcats	24
66	4.1.1 Anpassen der Anordnung von Faktoren	25
67	5 Spezielle Einträge	26
68	5.1 NA	26
69	5.2 NULL	27
70	5.3 Inf	27
71	6 data.frames oder Tabellen	29
72	6.1 Wichtige Funktionen zum Arbeiten mit data.frames	30
73	6.2 Zugreifen auf Elemente eines data.frame	31
74	7 Schreiben und lesen von Daten	34
75	7.1 Textdateien	34
76	8 Erstellen von Abbildungen	36
77	8.1 Base Plot	36
78	8.1.1 Mehrere Panels	42
79	8.1.2 Speichern von Abbildungen	42
80	8.2 Histogramme	43
81	8.3 Boxplots	46

82	8.4 ggplot2: Eine Alternative für Abbildungen	48
83	8.4.1 Multipanel Abbildungen	55
84	8.4.2 Plots kombinieren	58
85	8.4.3 Speichern von plots	60
86	9 Mit Daten arbeiten	62
87	9.1 dplyr eine Einführung	62
88	9.2 Arbeiten mit gruppierten Daten	65
89	9.3 pipes oder %>%	66
90	9.4 Joins	67
91	9.5 'long' and 'wide' Datenformate	69
92	9.6 Auswählen von Variablen	71
93	9.7 Einzelne Beobachtungen abfragen (slice())	72
94	9.8 Spalten trennen	75
95	10 Arbeiten mit Text	77
96	10.1 Arbeiten mit Text	77
97	10.2 Finden von Textmustern	78
98	11 Arbeiten mit Zeit	81
99	11.1 Arbeiten mit Zeitintervallen	82
100	11.2 Formatieren von Zeit	84
101	11.3 Zeitreihen	84
102	12 Aufgaben Wiederholen (for-Schleifen)	90
103	12.1 Schleifen	90
104	12.1.1 Wiederholen von Befehlen mit for()	90
105	12.1.2 Wiederholen von Befehlen mit while()	93
106	12.2 Bedingte Ausführung von Codeblöcken	93
107	13 (R)markdown	95
108	13.1 Markdown Grundlagen	95
109	13.2 R und Markdown	96
110	14 Räumliche Daten in R	98
111	14.1 Was sind räumliche Daten	98
112	14.2 Koordinatenbezugssystem	98
113	14.3 Vektordaten in R	98
114	14.4 Arbeiten mit Vektordaten	100
115	14.5 Rasterdaten in R	102
116	15 FAQs (Oft gefragtes)	108
117	15.1 Arbeiten mit Daten	108
118	15.1.1 Einlesen von Exceldateien	108
119	16 Literatur	109

1 R und RStudio

1.1 Installation von R und RStudio

- Als ersten Schritt müssen Sie R und RStudio installieren. Dabei ist wichtig zu unterscheiden, dass R und RStudio zwei unterschiedliche Programme sind. R ist die eigentliche Programmiersprache mit der wir arbeiten. RStudio hingegen ist eine sogenannte Entwicklungsumgebung¹, die das Arbeiten mit R vereinfacht.
- Sie können also mit R arbeiten ohne RStudio (was unüblich ist), aber nicht mit RStudio ohne R. RStudio wird unter anderem verwendet, um R Code komfortabler zu schreiben und zu verwalten. Es werden Ihnen für das Programmieren relevante Informationen bereitgestellt.
- Gehen Sie für die Installation von R, auf die Website <https://cloud.r-project.org/>, 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*).

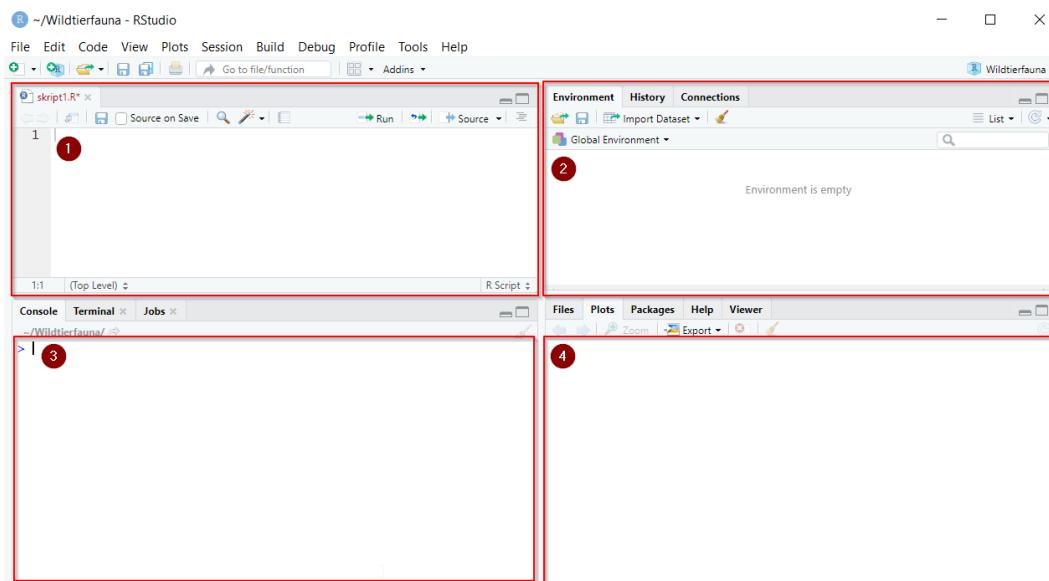


Abbildung 1: RStudio Panes.

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

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

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

10 + 5

157 **## [1] 15**

20 - 10

158 **## [1] 10**

10 * 3

159 **## [1] 30**

100 / 19

160 **## [1] 5.263158**

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

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

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

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

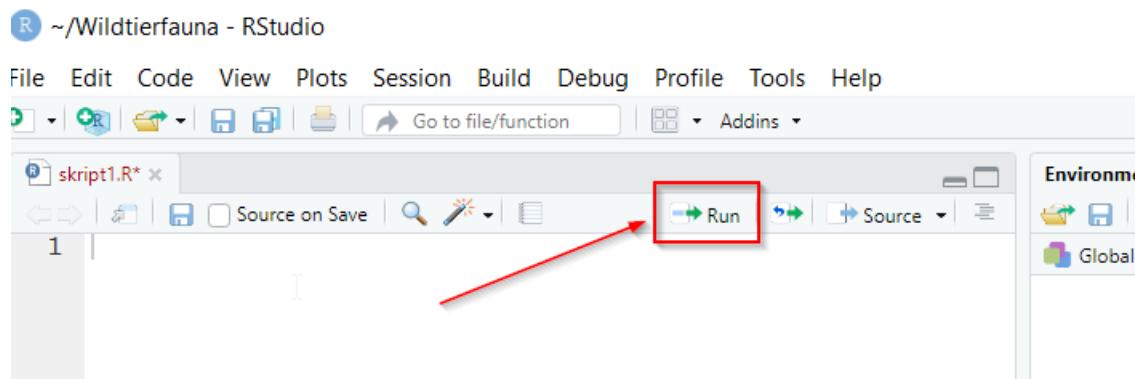


Abbildung 2: Zeilenweises Ausführen von Code in RStudio.

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

1.3 Gute Praxis bei der Programmierung

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

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

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

201 ## [1] 9

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

208

209 Aufgabe 1: Ausführen von Quellcodes

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

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

# Einfache Funktion
sqrt(20)
```

213 Führen Sie nun alle Zeilen aus.

214 1.4 RStudio Projekte

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

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

220 1.4.1 Erstellen eines Projektes

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

- 223 1. Gehen Sie zu `File > New Project ...`
- 224 2. Wählen Sie `New Project`.
- 225 3. Geben Sie einen Namen für das Projekt ein (z.B. den Namen einer Lehrveranstaltung) und ein neuer
 226 Ordner mit dem Projektnamen wird erstellt.

227 4. Drücken Sie auf **Create Project**.

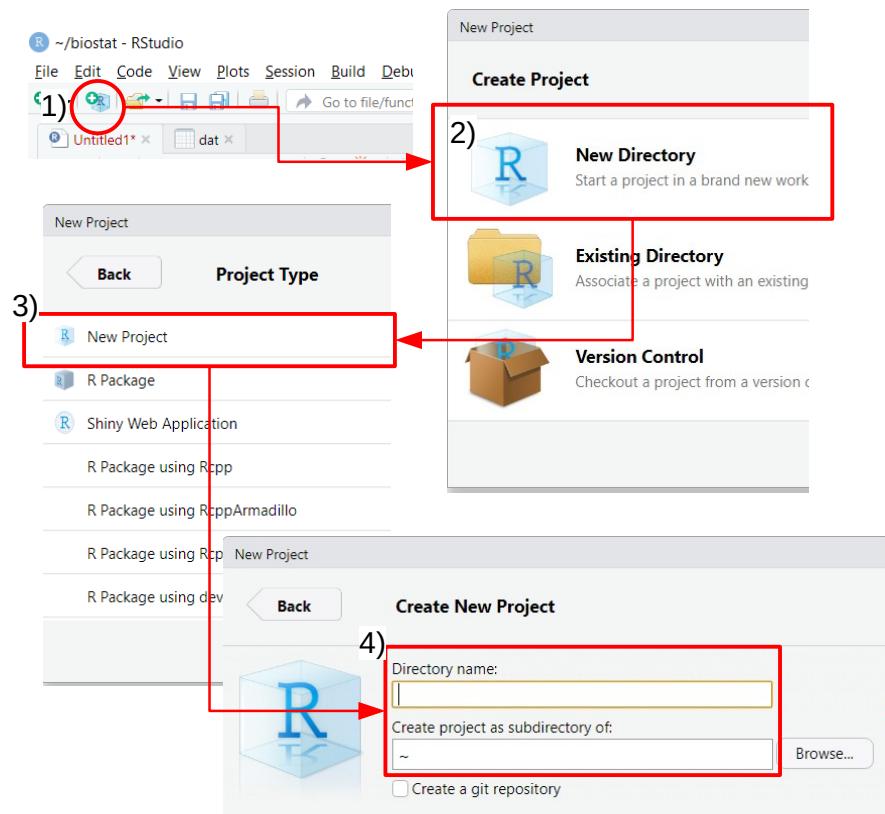


Abbildung 3: Workflow zum Erstellen eines Projekts.

- 228 Sobald ein Projekt einmal erstellt wurde, können Sie einfach wieder auf das Projekt-Icon klicken und das Projekt wieder öffnen (Abbildung 4)
- 229
- 230 Alternativ kann über **File > Open Project** in RStudio oder durch Auswählen des Projektnamens (siehe folgende Abbildung) geöffnet werden (Abbildung 5).
- 231

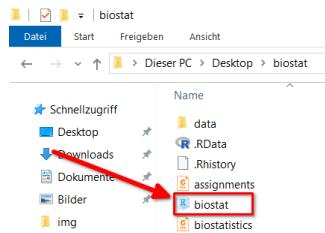


Abbildung 4: Öffnen eines RStudio Projekts.



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

250 ## [1] "Johannes"

251 Das Aufrufen der Variable
Name

252 führt zu einem Fehler.

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

a <- 10
b <- 5

a + b

255 ## [1] 15
b / a

256 ## [1] 0.5
a^b

257 ## [1] 1e+05

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

ergebnis <- a + b
ergebnis

259 ## [1] 15
ergebnis2 <- ergebnis * 2
ergebnis2

260 ## [1] 30

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

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

264 ## [1] TRUE
rm(var1)
exists("var1") # FALSE, also nein, eine Variable mit diesem Namen existiert nicht.

265 ## [1] FALSE

```

266 2.2 Funktionen

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

```
269 sqrt(a)
```

270 `## [1] 3.162278`

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

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

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

280 `## [1] 3.162278`

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

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

292 2.3 Datentypen

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

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

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

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

```
fuchs_gesehen <- TRUE
```

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

```
typeof(fuchs_gesehen)
```

311 ## [1] "logical"

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

```
TRUE + TRUE
```

313 ## [1] 2

```
FALSE + FALSE
```

314 ## [1] 0

```
TRUE + FALSE
```

315 ## [1] 1

316 2.4 Datenstrukturen

317 Verfolgen wir das Beispiel mit den Fotofallen etwas weiter. Es handelt sich um ein systematisches Monitoring.
 318 D. h., es wurde nicht nur eine Fotofalle ausgebracht, sondern insgesamt 15 Stück. Dieser Umstand erfordert
 319 komplexere Objekte. Nachfolgend sind die Anzahl Rehfotos für jede der 15 Fotofallen aufgeführt: 132, 79,
 320 129, 91, 138, 144, 55, 103, 139, 105, 96, 146, 95, 118, 107.

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

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

```
# 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.
```

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

329

330 **Aufgabe 2: Variablen**

- 332 Verwenden Sie die folgenden Daten

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

- 333 und berechnen sie:

- 334 • $10 * a$
 335 • $a / 144$ und speichern Sie das Ergebnis in einer neuen Variablen *e* zwischen.
 336 • Was ist das Ergebnis von $a + b$?
 337 • Was ist das Ergebnis von $a + p$?

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

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

3 Vektoren

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

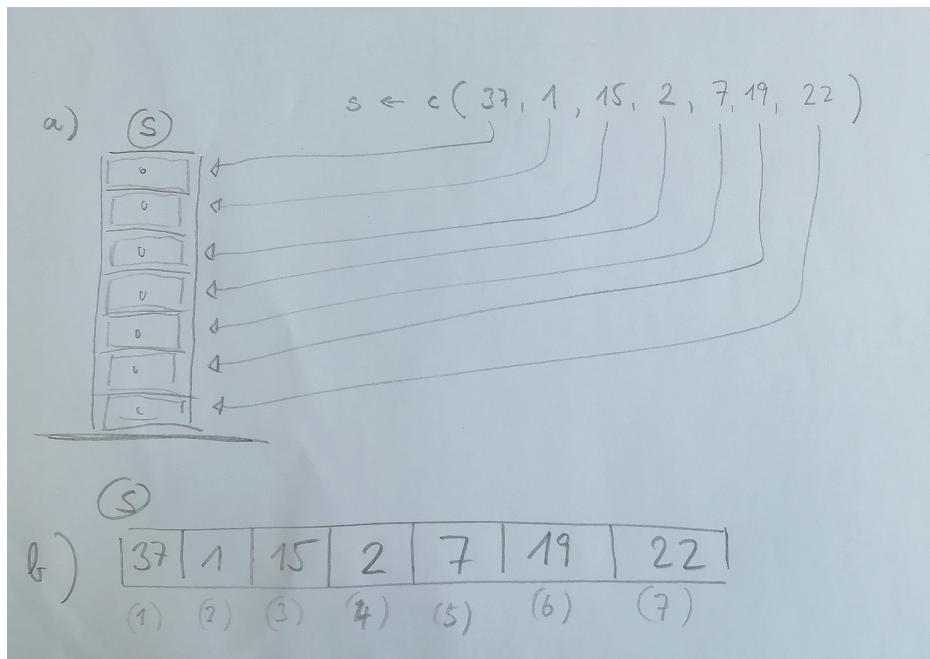


Abbildung 6: Schematische Darstellung eines Vektors in R.

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

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

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

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

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

s

358 ## [1] 37 1 15 2 7 19 22

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

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

s + 10

364 ## [1] 47 11 25 12 17 29 32

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

s * s

369 ## [1] 1369 1 225 4 49 361 484

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

seq(from = 1, to = 10)

373 ## [1] 1 2 3 4 5 6 7 8 9 10

(1 : 10)

374 ## [1] 1 2 3 4 5 6 7 8 9 10

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

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

376 ## [1] 1 3 5 7 9

377

378 Aufgabe 3: Vektoren erstellen

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

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

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

3.1 Funktionen zum Arbeiten mit Vektoren

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

```
head(s)
```

387 `## [1] 37 1 15 2 7 19`

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

388 `## [1] 37 1 15`

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

389 `## [1] 19 22`

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

```
length(s)
```

391 `## [1] 7`

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

```
class(s)
```

393 `## [1] "numeric"`

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

```
unique(s)
```

395 `## [1] 37 1 15 2 7 19 22`

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

```
table(s)
```

397 `## s`

398 `## 1 2 7 15 19 22 37`

399 `## 1 1 1 1 1 1 1`

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

```
rev(s)
```

402 `## [1] 22 19 7 2 15 1 37`

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

```
sort(s)
```

404 `## [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.

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

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

406 ## [1] 37 1 15 2 7 19 22 37 1 15 2 7 19 22

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

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

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

409 ## [1] 1 2 3 4 1 2 3 4

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

410 ## [1] 1 1 2 2 3 3 4 4

411

412 Aufgabe 4: Arbeiten mit Vektoren

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

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

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

417 3.2 Statistische Funktionen

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

```
mean(s)
```

421 ## [1] 14.71429

```
median(s)
```

422 ## [1] 15

```
sd(s)
```

423 ## [1] 12.76341

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

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

427 ## [1] 1

```

sample(s, size = 3) # 2 Elemente

428 ## [1] 15 7 22
429 Wenn size weggelassen wird, dann bekommt man gleich viele Elemente zurück (wie der Vektor lang ist), d.h.
430 der Vektor wird nur permutiert.

```

431 3.3 Beispiel Fotofallen

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

435 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)

```

436 Als zweites erstellen wir einen Vektor mit den IDs. Zur Erinnerung, diese sollten die Werte `Kamera_1` bis
437 `Kamera_15` haben. Ein erster Ansatz könnte sein, dass wir einfach 15 Fotofallen schreiben und dann die
438 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"
)

```

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

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

```

v1 <- rep("Kamera", 15)

```

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

```

v2 <- 1:15

```

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

```

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

449 ## [1] "Kamera_1"   "Kamera_2"   "Kamera_3"   "Kamera_4"   "Kamera_5"   "Kamera_6"
450 ## [7] "Kamera_7"   "Kamera_8"   "Kamera_9"   "Kamera_10"  "Kamera_11"  "Kamera_12"
451 ## [13] "Kamera_13"  "Kamera_14"  "Kamera_15"

```

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

```
454 rep(c("Revier A", "Revier B", "Revier C"), 5)
455 ## [1] "Revier A" "Revier B" "Revier C" "Revier A" "Revier B" "Revier C"
456 ## [7] "Revier A" "Revier B" "Revier C" "Revier A" "Revier B" "Revier C"
457 ## [13] "Revier A" "Revier B" "Revier C"
```

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

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

```
reviere
```

```
459 ## [1] "Revier A" "Revier A" "Revier A" "Revier A" "Revier A" "Revier B"
460 ## [7] "Revier B" "Revier B" "Revier B" "Revier B" "Revier C" "Revier C"
461 ## [13] "Revier C" "Revier C" "Revier C"
```

462

Aufgabe 5: Statistische Funktionen

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

466 2. Erstellen Sie die folgende Konsolenausgabe:

```
467 ## [1] "Die mittlere Anzahl von Rehfotos beträgt 171.8 Rehe pro Standort."
```

468 3.4 Arbeiten mit logischen Werten

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

- 471 • Gleichheit (`==`)
- 472 • Ungleichheit (`!=`)
- 473 • Größer (`>`) und kleiner (`<`)
- 474 • Größer gleich (`>=`) und kleiner gleich (`<=`)

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

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

```
anzahl_rehe > 100
```

```
478 ## [1] TRUE FALSE TRUE FALSE TRUE TRUE FALSE TRUE TRUE FALSE TRUE
479 ## [13] FALSE TRUE TRUE
```

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

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

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

```
482 ## [1] FALSE FALSE FALSE FALSE FALSE TRUE TRUE TRUE TRUE TRUE FALSE FALSE
483 ## [13] FALSE FALSE FALSE
```

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

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

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

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

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

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

```
495 ## [1] TRUE FALSE TRUE FALSE TRUE TRUE TRUE TRUE TRUE TRUE FALSE TRUE
496 ## [13] FALSE TRUE TRUE
```

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

499

500 Aufgabe 6: Arbeiten mit logischen Werten

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

```
503 1. TRUE | FALSE
504 2. FALSE & TRUE
505 3. (FALSE & TRUE) | TRUE
506 4. (2 != 3) | FALSE
507 5. FALSE + 10
508 6. TRUE + 10
509 7. TRUE + 10 == FALSE + 10
510 8. sum(c(TRUE, TRUE, FALSE, FALSE))
```

511 3.5 Zugreifen auf Elemente eines Vektors (=Untermengen)

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

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

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

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

```
anzahl_rehe[2]
```

525 ## [1] 79

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

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

527 ## [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.

528 ## [1] 132 79 129 91 138

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

531

532 Aufgabe 7: Zugreifen auf Vektorelemente

534 Erstellen Sie einen neuen Vektor `bhd`

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

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

536 • Wählen Sie aus dem Vektor `bhd` das letzte Element aus.

537 • Wählen Sie aus dem Vektor `bhd` das letzte Element aus, ohne die Zahl 6 zu schreiben.

538

539 Alternativ könnte das gleiche Ergebnis mit einem logischen Vektor erreicht werden. Für eine bessere Über-
540 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]
```

541 ## [1] 132 79 129 91 138

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

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

545 ## [1] 132 79 129 91 138

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

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

548 ## [1] 132 79 129 91 138

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

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

551 ## [1] 113.8

552

553 Aufgabe 8: logische Werte

555 Verwenden Sie nochmals den Vektor mit den Anzahl Rehen die an unterschiedlichen Fotofallenstandorten
 556 fotografiert wurden.

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

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

560 3.6 Der %in%-Operator

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

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

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

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

566 ## [1] "FI" "FI"

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

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

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

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

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

```
messungen_arten %in% arten
```

573 ## [1] TRUE TRUE TRUE FALSE FALSE FALSE TRUE FALSE FALSE

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

574 ## [1] "FI" "BU" "BU" "FI"

575

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

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

```
LETTERS
```

579 ## [1] "A" "B" "C" "D" "E" "F" "G" "H" "I" "J" "K" "L" "M" "N" "O" "P" "Q" "R" "S"

580 ## [20] "T" "U" "V" "W" "X" "Y" "Z"

581 Wählen Sie aus LETTERS nur die Vokale aus.

582 4 Faktoren (factors)

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

589 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)
```

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

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

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

595 ## [1] FI BU FI EI EI FI FI
 596 ## Levels: FI BU EI

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

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

599 ## [1] Fichte Buche Fichte Eiche Eiche Fichte Fichte
 600 ## Levels: Fichte Buche Eiche

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

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

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

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

```
af
```

```
608 ## [1] Fi Bu Fi Ei Ei Fi Fi
609 ## Levels: Fi Bu Ei
relevel(af, "Bu")
```

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

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

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

```
614 ## [1] "Fi" "Bu" "Fi" "Ei" "Ei" "Fi" "Fi"
```

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

```
af
```

```
616 ## [1] Fi Bu Fi Ei Ei Fi Fi
617 ## Levels: Fi Bu Ei
```

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

```
618 ## [1] 1 2 1 3 3 1 1
```

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

621

622 Aufgabe 10: Faktoren

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

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

626 4.1 Das Paket **forcats**

627 Mit dem Paket aus **forcats** werden einige Operationen mit Faktoren erleichtert. Wir schauen uns hier 628 Funktion an, die es erleichtern:

- 629 1. Die Anordnung von Levels anzupassen.
- 630 2. Levels zusammenzufassen oder zu entfernen.
- 631 3. Labels zu ändern.

632 **4.1.1 Anpassen der Anordnung von Faktoren**633 Wir verwenden nochmals den **a** Vektor mit unterschiedlichen Baumarten:

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

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

```
factor(a)
```

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

636 ## Levels: BU EI FI

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

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

```
f1
```

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

639 ## Levels: FI BU EI

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

```
fct_rev(f1)
```

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

642 ## Levels: EI BU FI

643 nach der Häufigkeit zu sortieren oder

```
fct_infreq(f1)
```

644 ## [1] FI BU FI EI EI FI FI

645 ## Levels: FI EI BU

646 eine zufällige Sortierung.

```
fct_shuffle(f1)
```

647 ## [1] FI BU FI EI EI FI FI

648 ## Levels: EI FI BU

5 Spezielle Einträge

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

- 651 • fehlenden Einträgen NA,
- 652 • leeren Einträgen NULL,
- 653 • undefinierten Einträgen NaN (Not a Number) oder
- 654 • unendlichen Zahlen (Inf).

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

656 5.1 NA

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

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

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

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

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

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

```
666 ## [1] "foo" "foo"
667 ## attr(,"na.action")
668 ## [1] 2
669 ## attr(,"class")
670 ## [1] "omit"
```

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

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

1 + NA

675 `## [1] NA`
 676 Viele R Funktionen haben eingebaute Methoden zum Umgang mit NA. Die Funktion `mean()` bspw. ergibt
 677 (wie die meisten Funktionen) standardmäßig NA wenn sie auf Vektoren mit Datenlücken angewendet wird, es
 678 sei denn man stellt innerhalb der Funktion ein, dass Datenlücken entfernt werden sollen.

mean(na2)

679 `## [1] NA`
`mean(na2, na.rm = TRUE)`
 680 `## [1] 4.5`

5.2 NULL

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

5.3 Inf

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

10^309

689 `## [1] Inf`
`2 * Inf`
 690 `## [1] Inf`
`1 + Inf`
 691 `## [1] Inf`
`3 / 0`
 692 `## [1] Inf`
`-3 / 0`
 693 `## [1] -Inf`
`3 / Inf`
 694 `## [1] 0`

695 Infinity kann mit `is.infinite` und `is.finite` getestet werden. Relationäre Operatoren funktionieren
 696 erwartungsgemäß.

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

697 ## [1] TRUE FALSE FALSE TRUE FALSE
is.finite(inf1)

698 ## [1] FALSE TRUE TRUE FALSE TRUE
inf1 < 3

699 ## [1] FALSE TRUE FALSE TRUE FALSE
```

700

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

703 Verwenden Sie den Vektor

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

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

706 Verwenden Sie den Vektor

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

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

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

712 Berechnen Sie den arithmetischen Mittelwert von `foo`.

713 6 data.frames oder Tabellen

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

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

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

```
726 ##           ID anzahl_rehe   revier
727 ## 1    Kamera_1      132 Revier A
728 ## 2    Kamera_2       79 Revier A
729 ## 3    Kamera_3      129 Revier A
730 ## 4    Kamera_4       91 Revier A
731 ## 5    Kamera_5      138 Revier A
732 ## 6    Kamera_6      144 Revier B
733 ## 7    Kamera_7       55 Revier B
734 ## 8    Kamera_8      103 Revier B
735 ## 9    Kamera_9      139 Revier B
736 ## 10   Kamera_10     105 Revier B
737 ## 11   Kamera_11     96 Revier C
738 ## 12   Kamera_12     146 Revier C
739 ## 13   Kamera_13     95 Revier C
740 ## 14   Kamera_14     118 Revier C
741 ## 15   Kamera_15     107 Revier C
```

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

747 Standard-Objekte zum Speichern wissenschaftlicher Daten.

748 6.1 Wichtige Funktionen zum Arbeiten mit `data.frames`

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

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

```
751 ##           ID anzahl_rehe    revier
752 ## 1 Kamera_1          132 Revier A
753 ## 2 Kamera_2          79 Revier A
```

754 Oder für die letzten 2 Beobachtungen.

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

```
755 ##           ID anzahl_rehe    revier
756 ## 14 Kamera_14         118 Revier C
757 ## 15 Kamera_15         107 Revier C
```

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

```
nrow(monitoring)
```

```
759 ## [1] 15
```

```
ncol(monitoring)
```

```
760 ## [1] 3
```

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

```
str(monitoring)
```

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

```
767
```

768 Aufgabe 12: `data.frame`

770 Stellen Sie sich vor, Sie machen eine kleine Umfrage, in der Sie fünf Menschen nach ihrem Studienfach, Semester
771 und Alter befragen. Erstellen Sie einen `data.frame` mit dem Namen `umfrage1` für diese Informationen und
772 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
```

```
804 ## 4 Kamera_4          91 Revier A
805 ## 5 Kamera_5          138 Revier A
```

806

807 Aufgabe 13: Abfragen von Werten

809 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)
)
```

- 810 • Wählen Sie nur die ersten drei Zeilen aus und die erste und zweite Spalte aus.
 811 • Wählen Sie alle Zeilen und die erste und dritte Spalte aus.
 812 • Wählen Sie alle Spalten und die erste, dritte und vierte Zeile aus.

813

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

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

```
monitoring$anzahl_rehe
```

819 ## [1] 132 79 129 91 138 144 55 103 139 105 96 146 95 118 107

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

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

821 ## [1] 132 79 129 91 138 144 55 103 139 105 96 146 95 118 107

822 3. Alle Zeilen und die Spalte explizit angeben.

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

823 ## [1] 132 79 129 91 138 144 55 103 139 105 96 146 95 118 107

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

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

```
monitoring$anzahl_rehe > 100
```

```
831 ## [1] TRUE FALSE TRUE FALSE TRUE TRUE FALSE TRUE TRUE TRUE FALSE TRUE  
832 ## [13] FALSE TRUE TRUE
```

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

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

```
837 ##          ID anzahl_rehe    revier  
838 ## 1   Kamera_1           132 Revier A  
839 ## 3   Kamera_3           129 Revier A  
840 ## 5   Kamera_5           138 Revier A  
841 ## 6   Kamera_6           144 Revier B  
842 ## 8   Kamera_8           103 Revier B  
843 ## 9   Kamera_9           139 Revier B  
844 ## 10  Kamera_10          105 Revier B  
845 ## 12  Kamera_12          146 Revier C  
846 ## 14  Kamera_14          118 Revier C  
847 ## 15  Kamera_15          107 Revier C
```

848

849 Aufgabe 14: Abfragen von Werten 2

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

- ```
851
852 • Alle Spalten für Studierende die Forstwissenschaften studieren.
853 • Alle Spalten für Studierende die Chemie oder Physik studieren.
854 • Die Spalte fach und semester für Studierende die 22 oder älter sind.
```

## 855 7 Schreiben und lesen von Daten

### 856 7.1 Textdateien

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

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

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

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

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

```
878 ## ID anzahl_rehe revier
879 ## 1 Kamera_1 132 Revier A
880 ## 2 Kamera_2 79 Revier A
881 ## 3 Kamera_3 129 Revier A
882 ## 4 Kamera_4 91 Revier A
883 ## 5 Kamera_5 138 Revier A
884 ## 6 Kamera_6 144 Revier B
```

885 Es gibt viele Varianten der Funktion `read.table()`. Beispielsweise hat die Funktion `read.csv()` bereits  
 886 die Argumente `sep = ','` und `header = TRUE` gesetzt. Die Funktion `read.csv2()` hat die in Deutschland  
 887 üblichen Argument `sep = ';'` und `dec = ','` gesetzt. Sie müssen natürlich darauf achten, dass sie die csv  
 888 Dateien mit den gleichen Spezifikationen einlesen, mit denen sie gespeichert wurde. Siehe dazu auch die  
 889 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.

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

891

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

---

893  
894 Lesen Sie die Datei `kompliziert.txt` ein. Schauen Sie die Hilfeseite an und vergewissern Sie sich, dass Sie  
895 wissen was die Argumente `header`, `sep`, `dec` und `skip` bewirken. Setzten die Argumente richtig, damit die  
896 Datei `kompliziert.txt` folgendes Ergebnis liefert.

## 8 Erstellen von Abbildungen

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

### 902 8.1 Base Plot

903 Die wichtigsten Grafiken für die einfache Datendarstellung sind schnell verfügbar. Etwas komplexere oder  
904 spezielle Grafiken erfordern mehr Programmieraufwand (folgt teilweise noch). Für viele komplexere Diagramme  
905 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  
906 sie sich die einfache Grafik Schnittstelle (**base plots**) als zweidimensionale Leinwand vor, auf die Sie durch  
907 Code Ebene für Ebene Grafikelemente legen:  
908

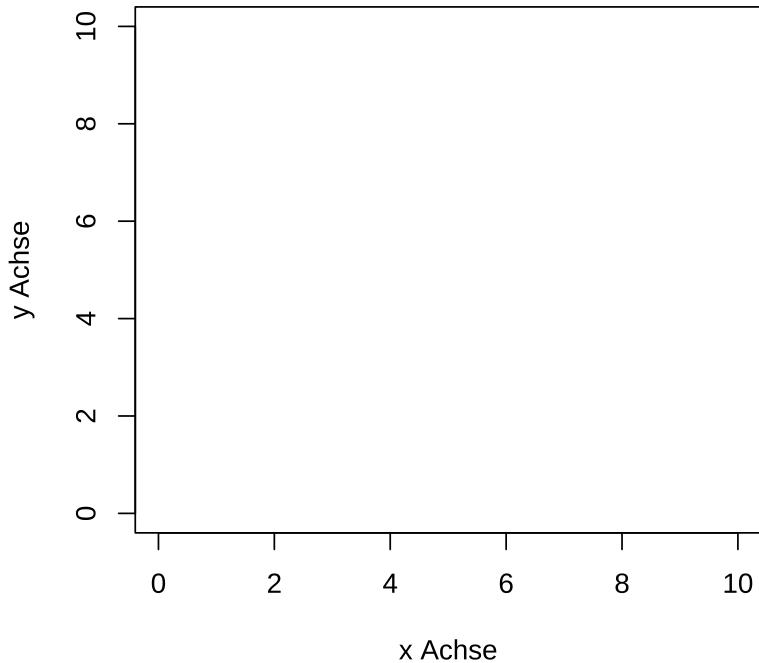
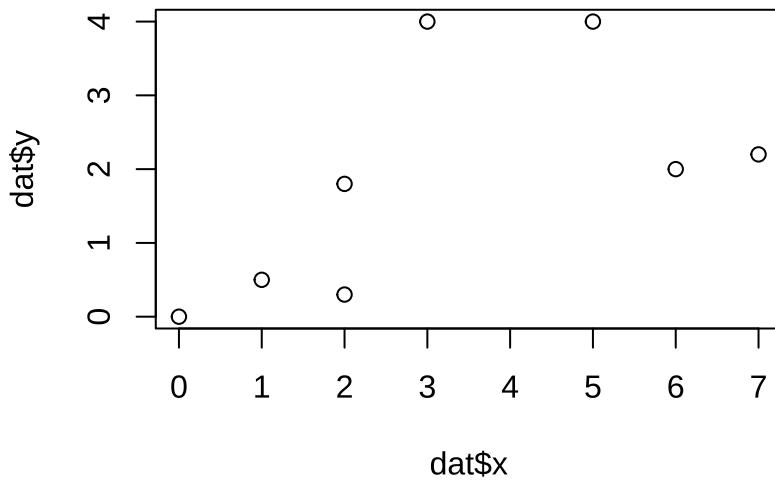


Abbildung 7: Beispiel einer leeren Grafikschnittstelle.

909 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)
)

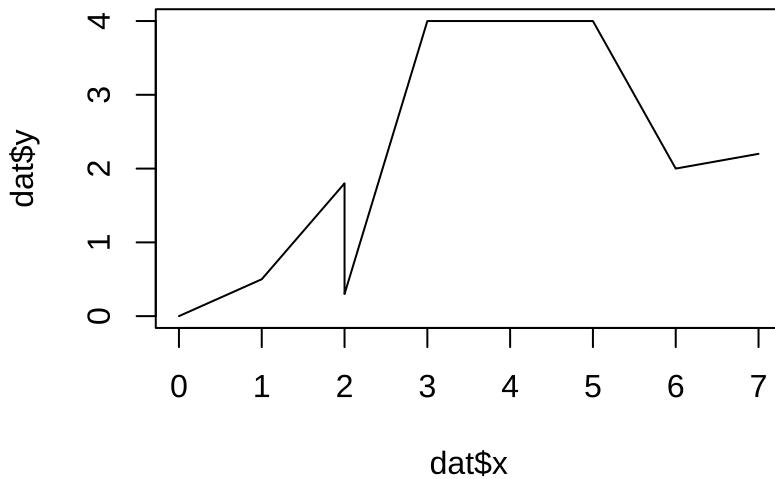
plot(datx, daty, type = "p")
```



910

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

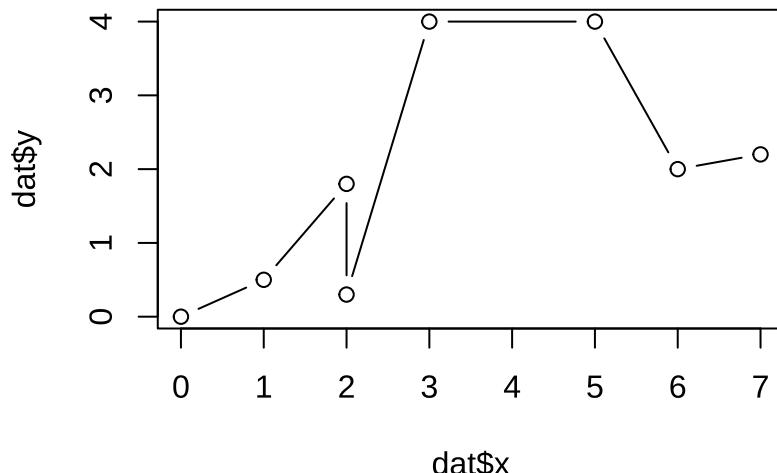
```
plot(datx, daty, type = "l")
```



913

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

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



915

916 darstellen.

917

918 **Aufgabe 16: Base Plot 1**

919

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

922

923 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  
924 Elemente hinzufügen. Mit jeder neuen Ebene können Sie die vorigen Ebenen jedoch nicht mehr verändern.  
925 Die wichtigsten Argumente der `plot` Funktion sind:

926

- `type` - Diagrammtyp
- `col` - Farbe
- `main` - Titel
- `sub` - Untertitel
- `pch` - Punktsymbol
- `lty` - Linientyp
- `lwd` - Linienstärke
- `xlab` bzw. `ylab` - Achsenbeschriftungen
- `xlim`, `ylim` - Grenzen der Achsenanschnitte
- `axes` - Sollen die Achsen eingezeichnet werden? Oder leer gelassen werden, um sie nachträglich als low-level Ebene einzuziehen?
- `ann` - Achsenbeschriftung kann ganz weggelassen werden.

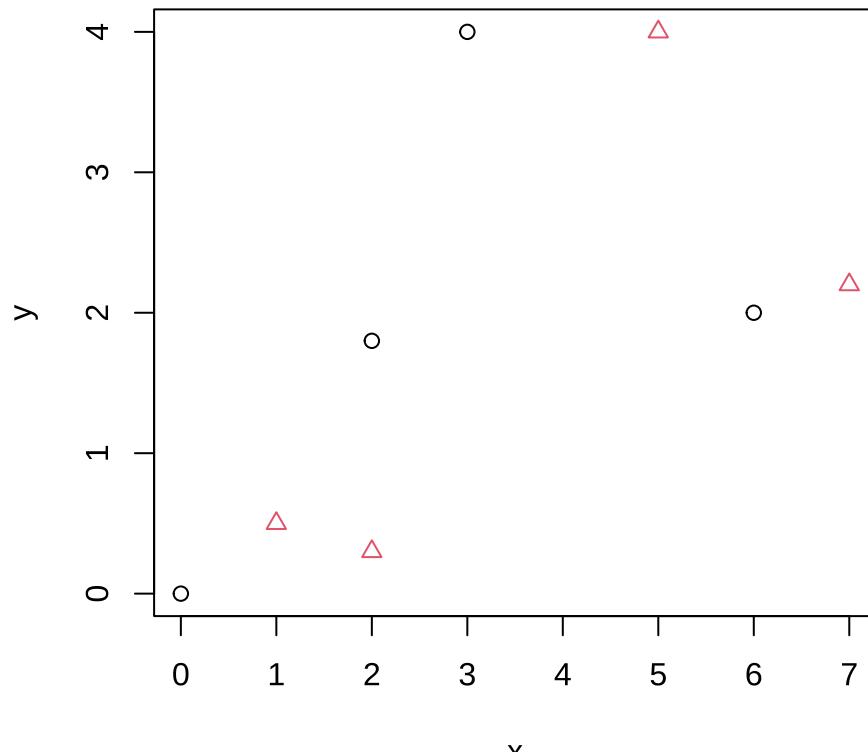
927 Sehen Sie sich die Hilfeseiten `?plot.default()` oder `?par()` an für weitere Informationen. Dort finden Sie  
928 auch eine vollständige Liste der Befehle. Einige Argumente können als Vektor übergeben werden. Hier z. B.  
929 die Farben und die Punktsymbole.

```

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")

```



942

943

---

**Aufgabe 17: Anpassen von Plots**


---

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

- 945
- 946 • Beschriften Sie die x- und y-Achse sinnvoll.
  - 947 • Fügen Sie eine Überschrift hinzu.
  - 948 • Wählen Sie ein anderes Symbol.
  - 949 • Stellen Sie die Symbole in rot dar.

951

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

953 Die wichtigsten Funktionen sind

- 954
- 955 • `points()` - Fügt Punkte ein
  - `lines()` - Fügt Linien ein

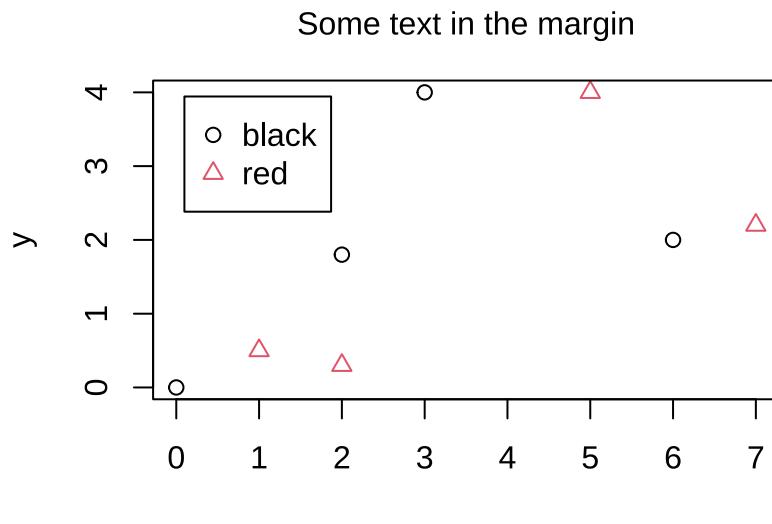
- 956 • `text()` - Fügt Text ein  
 957 • `mtext` - Fügt Text in den Rahmen (`margin`) ein  
 958 • `legend()` - Fügt eine Legende ein  
 959 • `abline()` - Fügt eine Gerade ein  
 960 • `curve()` - Fügt eine mathematische Funktion ein  
 961 • `arrows()` - Fügt Pfeile ein  
 962 • `grid()` - Fügt Hilfslinien ein

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

966 Legenden können im base plot nur als Low-Level Funktion hinzugefügt werden. Mit der Funktion `legend`  
 967 werden die Einträge und die entsprechenden Punktsymbole oder Linientypen dargestellt. In der folgenden  
 968 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")
```



969 970 Mit diesem grundsätzlichen Aufbau sollten Sie bereits in der Lage sein auch komplexe Grafiken schnell zu  
 971 gestalten. Wenn Sie mehrere Diagramme in einem Plot arrangieren möchten, können Sie mit dem `par()`  
 972 Befehl ein Arrangement definieren. Sie haben dann zusätzlich zu den bereits bekannten Grafikregionen noch  
 973 äußere Ränder (`outer margins`). Siehe Abbildung 9.

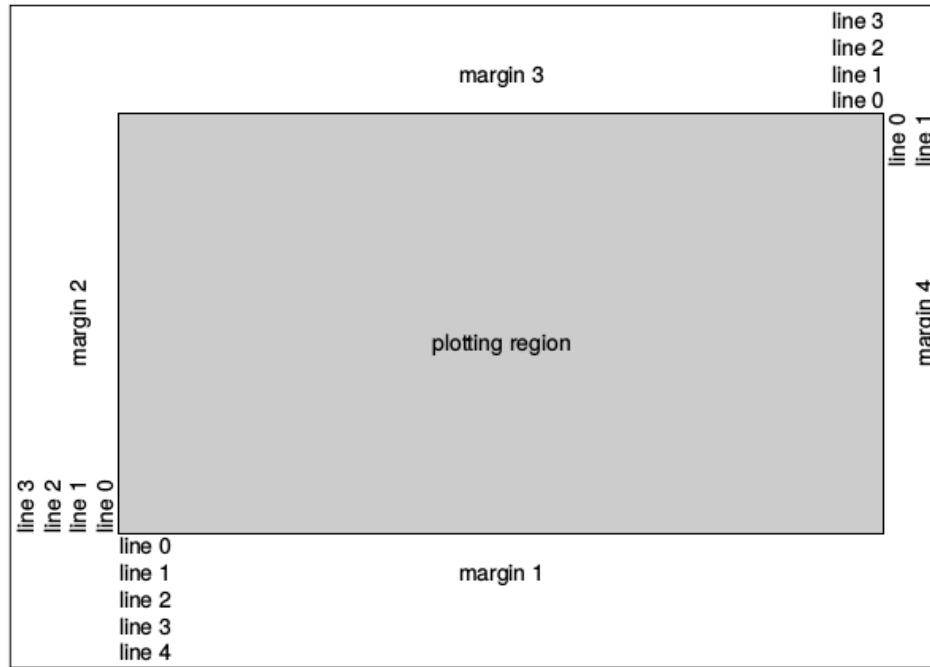
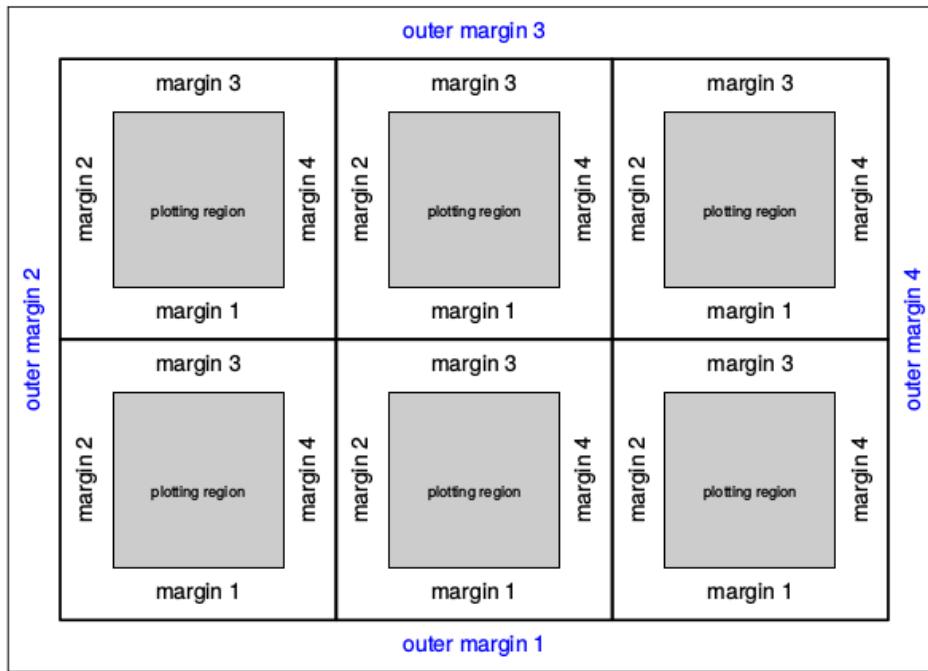


Abbildung 8: Grafikregionen eines base plots in R.

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

974 **8.1.1 Mehrere Panels**

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

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

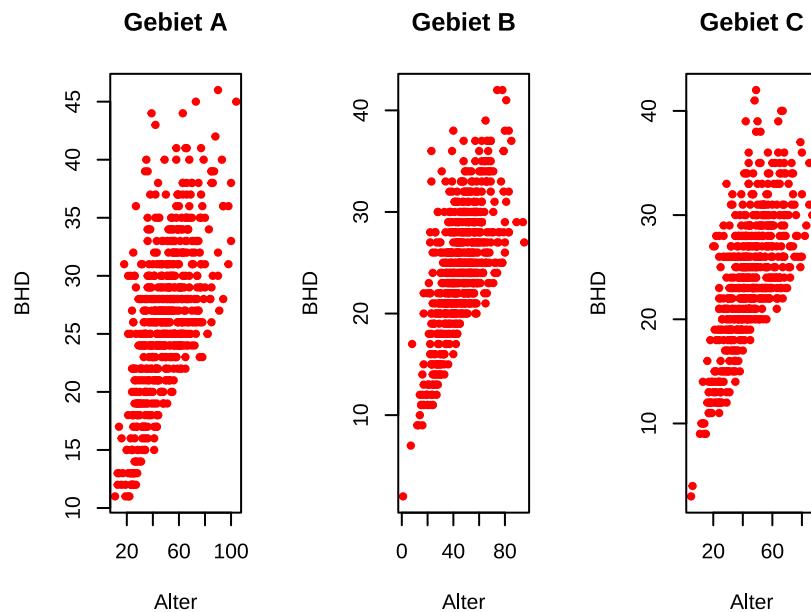
978 Teilt den Plot in eine Zeile und drei Zeilen (= drei Plots nebeneinander).

```
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")
```



979

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

982 **8.1.2 Speichern von Abbildungen**

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

- 987     • `pdf()` oder  
 988     • `postscript()`.

989 Beispiele für Rastergrafiken sind

- 990     • `png()`,  
 991     • `bmp()` oder  
 992     • `jpeg()`.

993 Die Grafikschnittstelle ist dann Ihre “Leinwand”. Mit dem Befehl `dev.off()` trennen Sie die Verbindung zur  
 994 Schnittstelle wieder. Ihre “Leinwand” wird also wieder geschlossen. So lange die Schnittstelle geöffnet ist  
 995 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
```

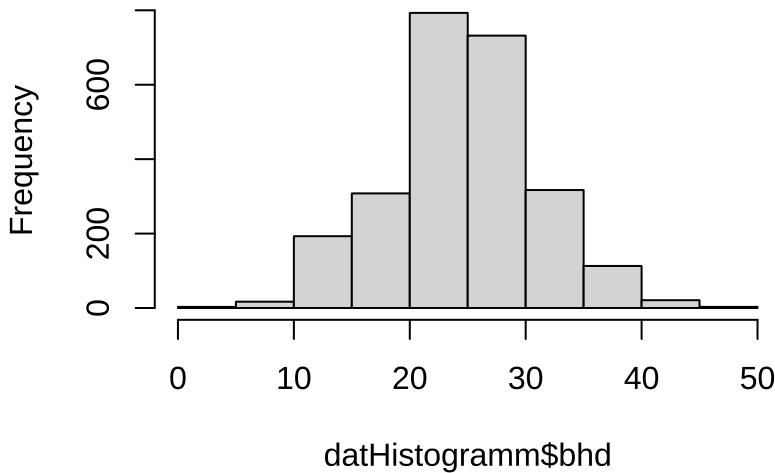
996 *Achtung*, wenn Sie die Funktion `dev.off()` nicht aufrufen, werden alle nachfolgenden Plots in die gleiche  
 997 Datei geschrieben. Falls Sie nach einem Versuch einen Plot zu speichern plötzlich keine weiteren Plots mehr  
 998 sehen, führen Sie einige Mal die Funktion `dev.off()` aus.

## 999 8.2 Histogramme

1000 Neben den Streudiagrammen (*Scatterplots*, oder auch einfach x-y Diagramm) sind *Histogramme* in der  
 1001 angewandten Datenanalyse ein weiterer wichtiger Abbildungstyp. An Histogrammen wird die Häufigkeit  
 1002 von Beobachtungen nach Gruppen dargestellt. Sie sind deshalb so wichtig, weil man aus ihnen relevante  
 1003 Informationen über die Verteilung der Daten ablesen kann und somit einiges über die Messungen erfährt,  
 1004 das für die weitere Datenanalyse relevant sein kann. So werden auf einen Blick der Zusammenhang von  
 1005 Beobachtungshäufigkeit und Streuung deutlich, sowie auch die Form der Verteilung und ihre Schiefe. Die  
 1006 Interpretation werden wir bei den Boxplots noch weiter vertiefen.

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

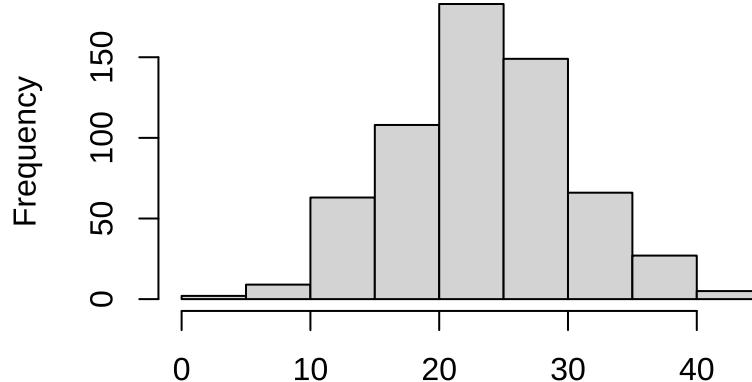
### Histogramm of datHistogramm\$bhd



1007

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

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

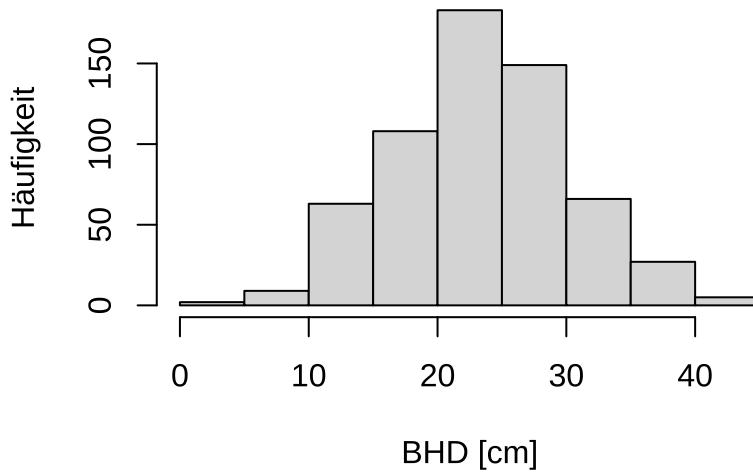


1008

```
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")
```

## Anzahl der Eichen

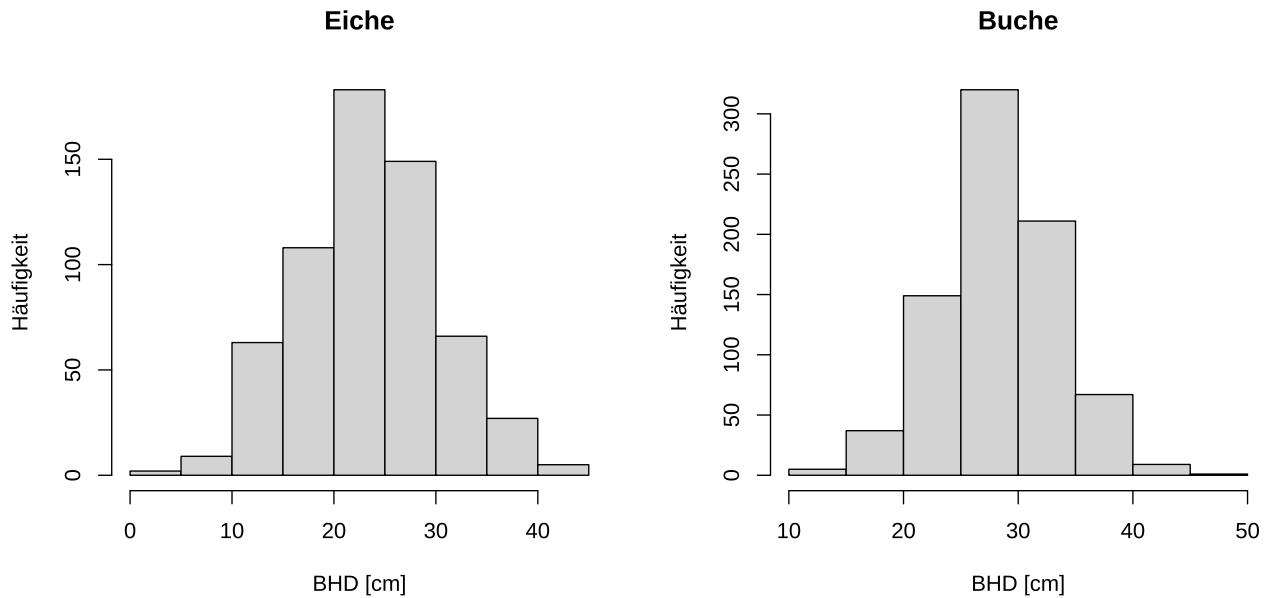


1009

1010 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")
```

1011



1012

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

### 1013 8.3 Boxplots

1014 Oft möchte man die Verteilung einer stetigen Variablen in Abhängigkeit einer diskreten Variable beschreiben  
 1015 oder visualisieren. Ein Beispiel dafür wäre die BHD-Verteilung in Abhängigkeit der Baumarten. Eine häufige  
 1016 Darstellungsform für solche Daten sind *Boxplots*. Sie stellen die durchschnittliche Ausprägung einer stetigen  
 1017 Variable und ihre Schwankung kompakt dar.

1018 Boxplots bestehen aus drei Komponenten:

- 1019 1. Eine *Box*, die den Bereich zwischen 0.25 und 0.75 Percentil abdeckt, diese Distanz wird auch die *IQR*  
 1020 (Interquartile Range), bezeichnet. Zusätzlich wird die Box durch den Median (als dicke horizontale Linie)  
 1021 unterteilt.
- 1022 2. Einzelne Punkte Ausreißer. Als Ausreißer werden Punkte bezeichnet, die  $> 1.5IQR$  vom unteren oder  
 1023 oberen Ende der Box entfernt sind.
- 1024 3. Eine senkrechte Linie von jeder Seite der Box bis zum letzten "Nicht-Ausreißer-Punkt". Also der letzte  
 1025 Punkt, der  $> 1.5IQR$  aber nicht  $> 0.75$  bzw.  $< 0.25$  Percentil ist. Diese Linie wird auch als *Whisker*  
 1026 bezeichnet.

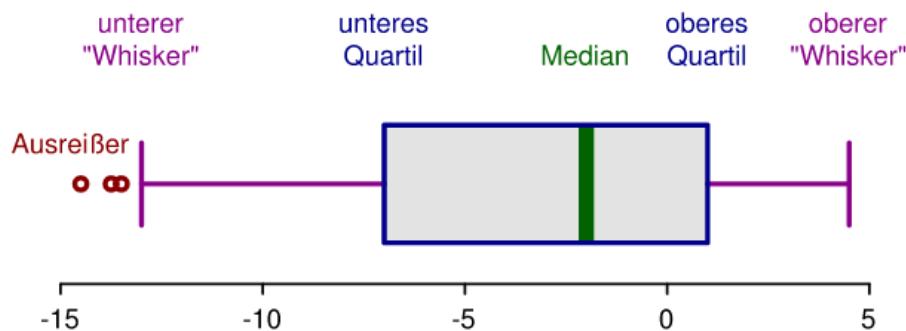
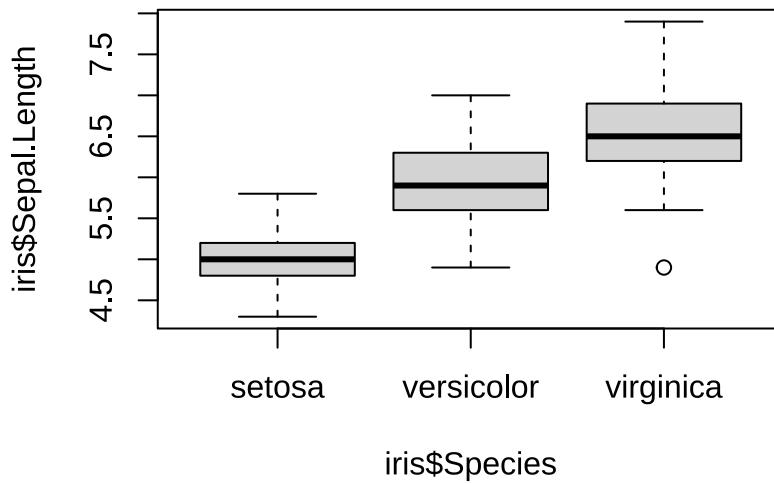


Abbildung 10: Schematische Darstellung eines Boxplots (Quelle: Von RobSeb - Eigenes Werk, CC BY-SA 3.0, <https://commons.wikimedia.org/w/index.php?curid=14697172>).

1027 Mit R kann mit der Funktion `boxplot()` ein Boxplot erstellt werden. Diese Funktion kann in zwei unter-  
 1028 schiedlichen Ausprägungen verwendet werden.

- 1029 1. `boxplot(x)` erzeugt einen Boxplot für die Variable `x`.
- 1030 2. `boxplot(x ~ y)` erzeugt einen oder mehrere Boxplots für `x` aber gruppiert nach `y`, dabei sollte `y` eine  
 1031 kategoriale Variable sein. `x` und `y` können auch die Spaltennamen eines `data.frames` sein, dann muss  
 1032 das `data.frame` mit dem Argument `data` zusätzlich übergeben werden.

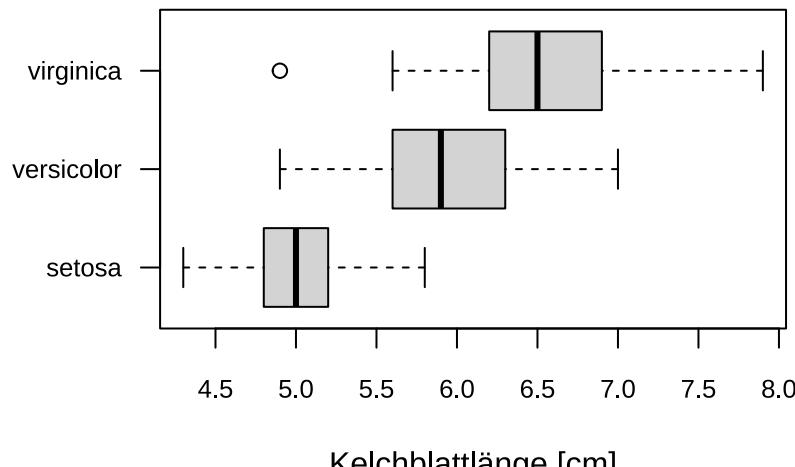
```
boxplot(iris$Sepal.Length ~ iris$Species)
```



1033

1034 Etwas eleganter ist es wenn wir das Argument `data` verwenden und den Plot etwas anpassen. Diese Schreib-  
1035 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
)
```



1036

1037

### 1038 Aufgabe 18: Boxplots

1039

- 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
- 1043 Erstellen Sie Boxplots für jedes Gebiet und innerhalb der Gebiete für jede Art.

## 1044 8.4 ggplot2: Eine Alternative für Abbildungen

1045 ggplot2 ist ein alternatives Plotting-System in R. Sie können mit ggplot2 also grundsätzlich Abbildungen  
 1046 mit dem selben Inhalt erstellen, wie mit Base Plots. Die Syntax und die optische Darstellung unterscheiden  
 1047 sich jedoch grundsätzlich. ggplot2 basiert auf den *grammar of graphics* von Leland Wilkinson. Die Idee  
 1048 ist, alle nötigen Informationen der Abbildung miteinander zu verknüpfen. ggplot2 ist also diametral zu  
 1049 Base Plots. Während Sie in base plot alle Elemente selbst definieren, ist die Idee von ggplot2, dass Sie  
 1050 nur die Daten und den Diagrammtypen übergeben und die Funktion selbst eine schöne Abbildung erstellt.  
 1051 Selbstverständlich können Sie aber auch in ggplot2 viele Einstellungen vornehmen. Im base plot sehen  
 1052 Abbildungen zunächst sehr karg und etwas unfertig aus. Sie müssen viele Einstellungen vornehmen, um eine  
 1053 publizierfähige Grafik zu produzieren. In ggplot2 sollen auch die einfachste Abbildungen schon ästhetisch  
 1054 sein. Mit diesen gebündelten Informationen kann ggplot2 die Abbildung automatisch verschönern. So  
 1055 werden bspw. die Legenden automatisch erzeugt und auch die Formatierungen automatisch an die Datenlage  
 1056 angepasst. ggplot2 nimmt der\*dem Entwickler\*in also Arbeit ab. Dadurch sind die Abbildungen schon ohne  
 1057 viel Nacharbeit schick. Nachteil ist, dass der\*dem Entwickler\*in weniger Möglichkeiten zur Einstellung zur  
 1058 Verfügung stehen und nuterspezifische Sonderwünsche somit schwerer umsetzbar sind. Sehen Sie sich das  
 1059 Cheatsheet zu ggplot2 an. Es ist in RStudio unter [Help > Cheatsheets](#) zu finden.

1060 Bei ggplot2 sind Anweisungen zu den Daten und Anweisungen zur Darstellung voneinander getrennt. Die  
 1061 Daten werden in den Ästhetikbefehl übergeben und dort klassifiziert. Dann folgen die Darstellungsanweisungen.  
 1062 Ähnlich wie bei Base Plots, werden die Grafikelemente ebenenweise nacheinander programmiert, jedoch mit  
 1063 einem + verbunden. Und hier liegt der wesentliche Unterschied zu Base Plots. Durch die + werden die Ebenen  
 1064 zu einem Befehl verbunden und damit gleichzeitig erstellt.

1065 Die Erweiterung wird zunächst geladen<sup>7</sup>. Wir laden außerdem den Datensatz **iris**. Der Datensatz ist in R  
 1066 fest integriert. Siehe `?iris` für mehr Informationen.

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

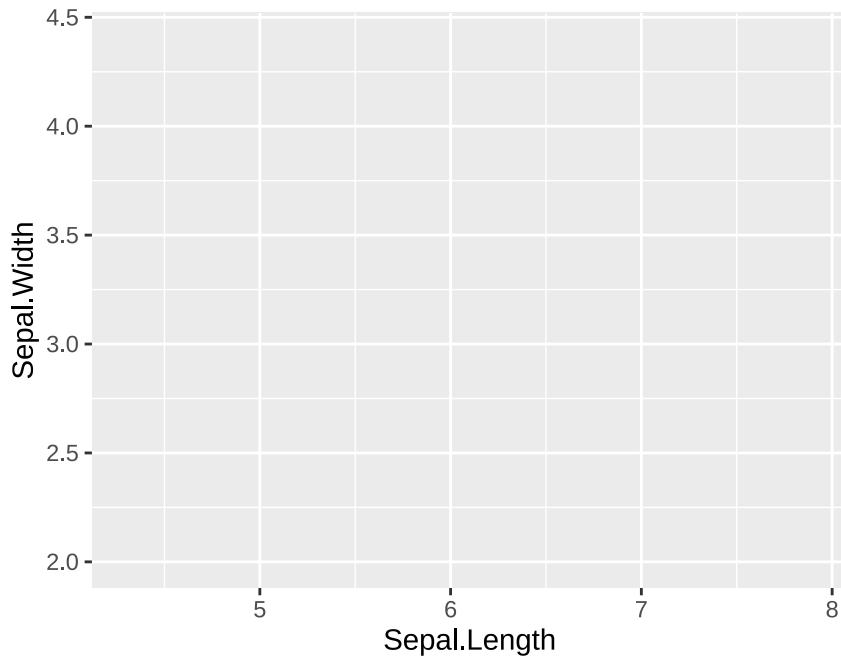
```
1067 ## Sepal.Length Sepal.Width Petal.Length Petal.Width Species
1068 ## 1 5.1 3.5 1.4 0.2 setosa
1069 ## 2 4.9 3.0 1.4 0.2 setosa
1070 ## 3 4.7 3.2 1.3 0.2 setosa
1071 ## 4 4.6 3.1 1.5 0.2 setosa
1072 ## 5 5.0 3.6 1.4 0.2 setosa
1073 ## 6 5.4 3.9 1.7 0.4 setosa
```

1074 Die Ästhetik wird bspw. folgendermaßen definiert.

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

---

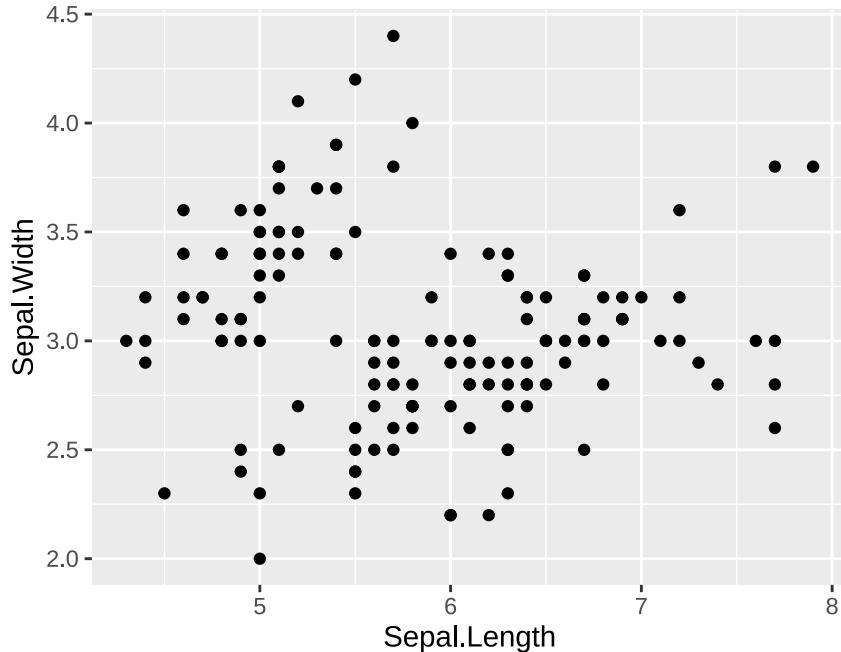
<sup>7</sup>Wir haben bis jetzt immer nur mit base R gearbeitet. D.h. wir haben nur Funktionen verwendet, die R bereits zur Verfügung stellt. Eine der großen Stärken von R sind die Erweiterungen (oder auch Pakete genannt). ggplot2 ist so eine Erweiterung, die einmal mit `install.packages("ggplot2")`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



1075

1076 Dieser Befehl zeichnet noch keine Daten. Die Daten werden lediglich herangezogen, um einen leeren Plot für  
1077 die Daten zu erstellen. In dem Beispiel wird die Variable `Sepal.Length` aus dem `data.frame iris` als x und  
1078 `Sepal.Width` als y Variable definiert. Diese Informationen stehen den folgenden Layern nun zur Verfügung,  
1079 sodass nach den + nur noch x und y verwendet werden müssen. Um bspw. einen Scatterplot zu erstellen  
1080 wird ein `geom_point()` Layer hinzugefügt. x und y werden automatisch an `geom_point` übergeben. Weitere  
1081 Einstellung sind in diesem Beispiel nicht notwendig, wären jedoch möglich. Siehe `?geom_point()`.

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



1082

1083

---

1084 **Aufgabe 19: Abbildungen mit ggplot2**

---

1085

1086 Verwenden Sie die Daten aus Aufgabe 16 und erstellen Sie einen Scatterplot mit ggplot2 wie in Aufgabe 16.

1087

1088 Wir haben mit der Funktion `geom_point()` demm Plot eine Punktgeometrie hinzugefügt. Es gibt noch viele  
1089 weitere Geometrien. Die wichtigsten sind:

- 
- 1090 •
- `geom_line()`
- für eine Linie.
- 
- 1091 •
- `geom_histogram()`
- um ein Histogramm zu erstellen.
- 
- 1092 •
- `geom_boxplot()`
- um einen Boxplot zu erstellen.
- 
- 1093 •
- `geom_bar()`
- um ein Säulendiagramm zu erstellen.

---

1094 Welche Geometrie die richtige ist, richtet nach dem Typ der darzustellenden Variablen. Beispielsweise  
1095 bietet sich `geom_point()` an, wenn man zwei kontinuierliche Variable darstellen möchte. Wenn man hin-  
1096 gegen die Verteilung von einer kontinuirlchen Variable darstellen möchte, dann bietet sich ein Histogramm  
1097 (`geom_histogram()`) oder auch eine geschätzte Dichte (z.B. `geom_density()`) an.

1098

---

1099 **Aufgabe 20: Abbildungen mit ggplot2**

---

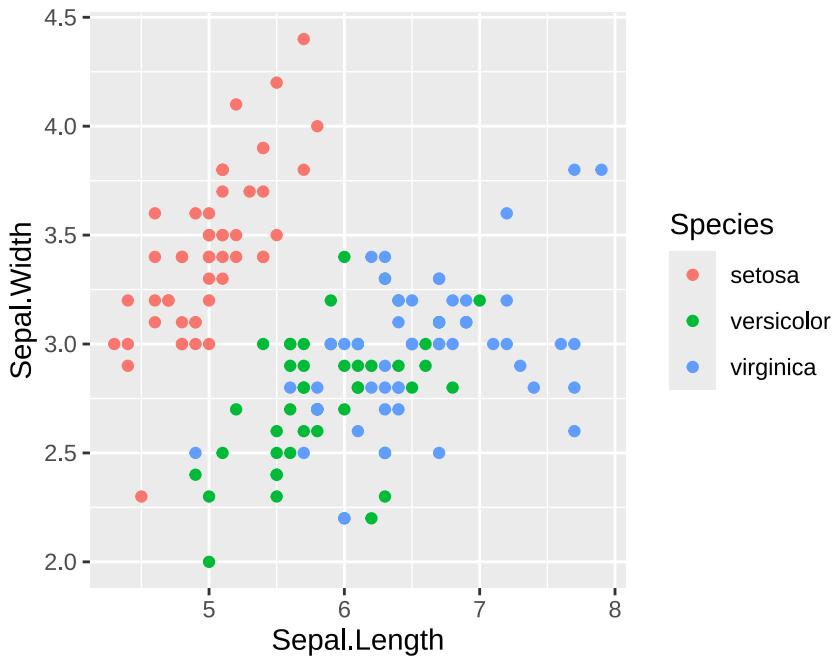
11001101 Verwenden Sie die den Iris Datensatz und erstellen Sie mit ggplot2 einen Plot der die Verteilung der Länge  
1102 der Kelchblätter zeigt (Spalte `Sepal.Length`).

1103

---

1104 Eine der Stärken von ggplot2 ist, dass man den Wert unterschiedlicher Variable auf unterschiedlichen  
1105 Komponenten des Plots abbilden kann. Wir haben bis jetzt ein bzw. zwei Variable auf der x- und y-Achse  
1106 abgebildet. Wir können aber ein weitere Variablen verwenden um das Aussehen des Plots zu beeinflussen.  
1107 Beispielsweise können wir die Farbe der Punkte (für `geom_point()`) mit dem Argument `col` beeinflussen.

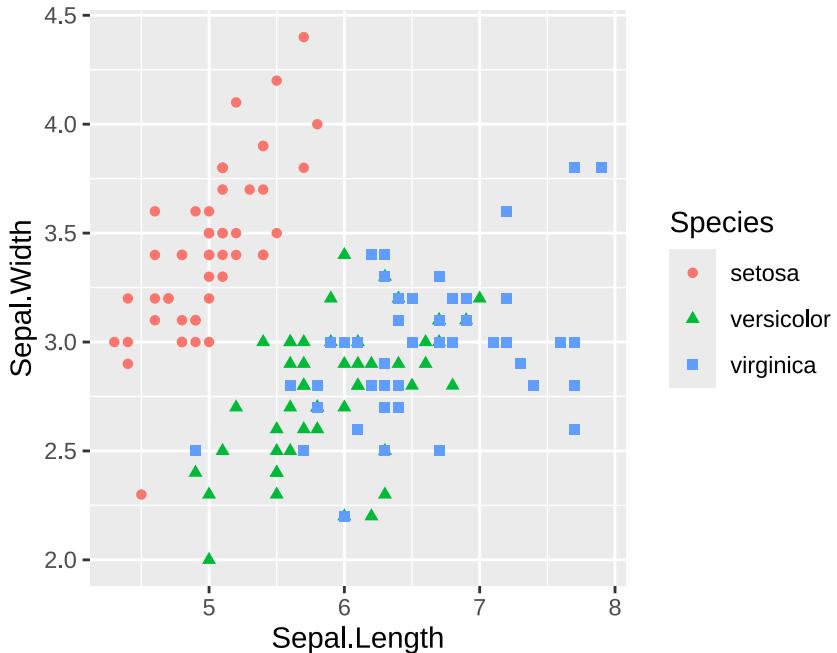
```
ggplot(iris, aes(x = Sepal.Length, y = Sepal.Width, col = Species)) +
 geom_point()
```



1108

1109 Somit bekommt jede Irisart eine eigene Farbe<sup>8</sup>. Gleichesmaßen können wir die Punktart (**shape**), die  
1110 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()
```

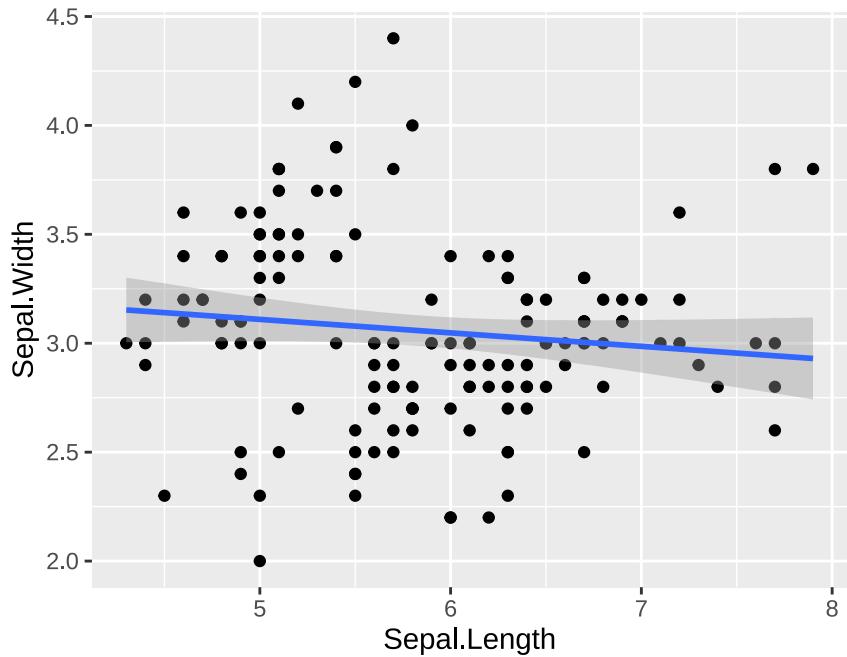


1111

1112 In dem Plot ist die Information zu der Art redundant (einmal als Farbe und einmal Symbolart).  
1113 Ein weitere sehr nützliche Geometrie ist **geom\_smooth()**, die es erlaubt eine Trendlinie hinzuzufügen.

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

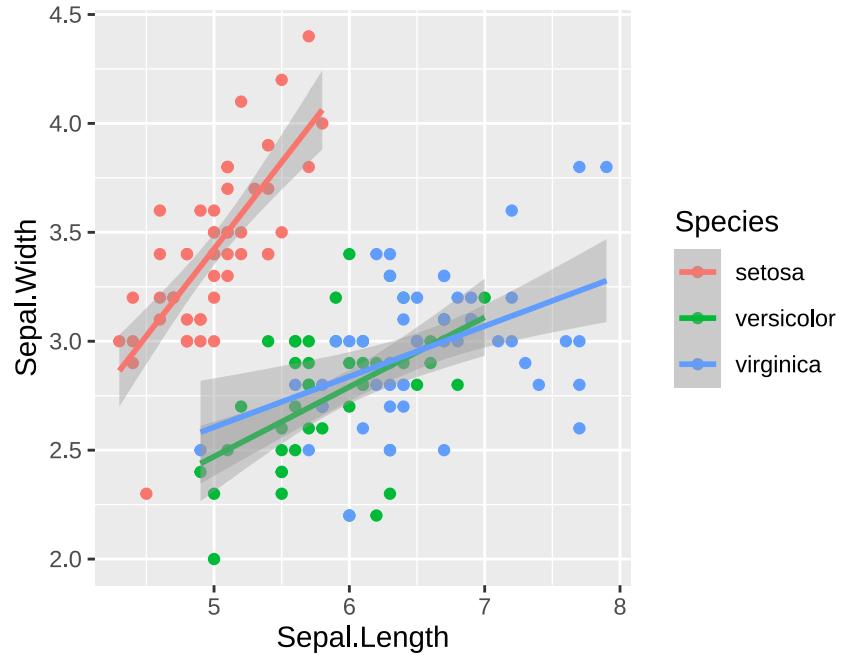
```
ggplot(iris, aes(x = Sepal.Length, y = Sepal.Width)) +
 geom_point() + geom_smooth(method = "lm")
```



1114

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")
```



1118

1119

1120 **Aufgabe 21: Anpassen von Plots**

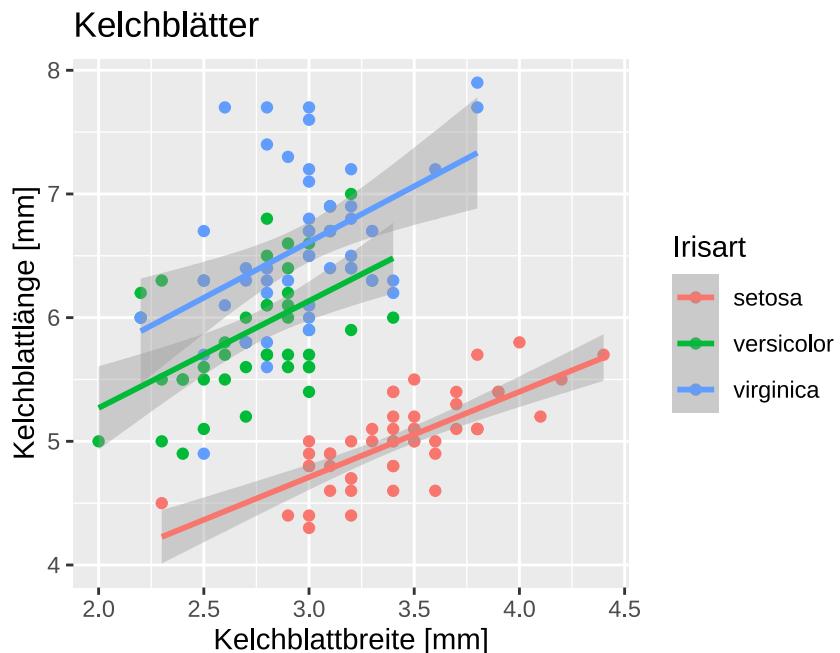
- 1122 Lesen Sie den Datensatz `data/bhd_1.txt` ein und erstellen Sie einen Boxplot für die Verteilung des BHDs  
 1123 für jede Baumart. In einem zweiten Schritt verwenden Sie erst `col = gebiet` und dann `fill = gebiet`.  
 1124 Welchen Unterschied stellen Sie fest?

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

1125

- 1126 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")
```



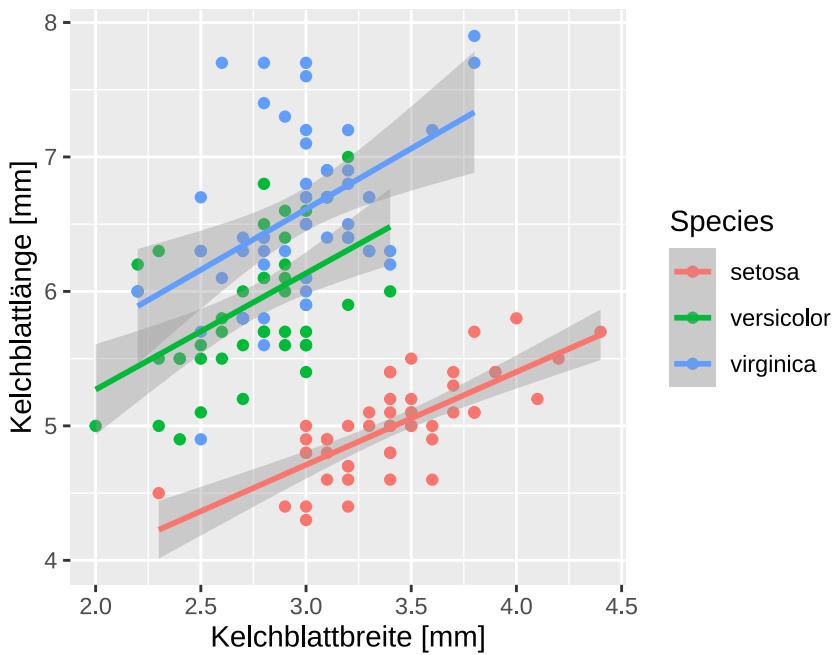
1127

- 1128 Statt einen langen Befehl zu tippen, kann ein `ggplot()` auch zwischengespeichert und wieder aufgerufen bzw.  
 1129 angepasst werden. Das ist vor allem sinnvoll, wenn mehrere Abbildungen auf dem selben Zwischenergebnis  
 1130 aufbauen sollen.

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

- 1131 Wir können jetzt mit `p1` weiter arbeiten und beispielsweise eine Beschriftung hinzufügen.

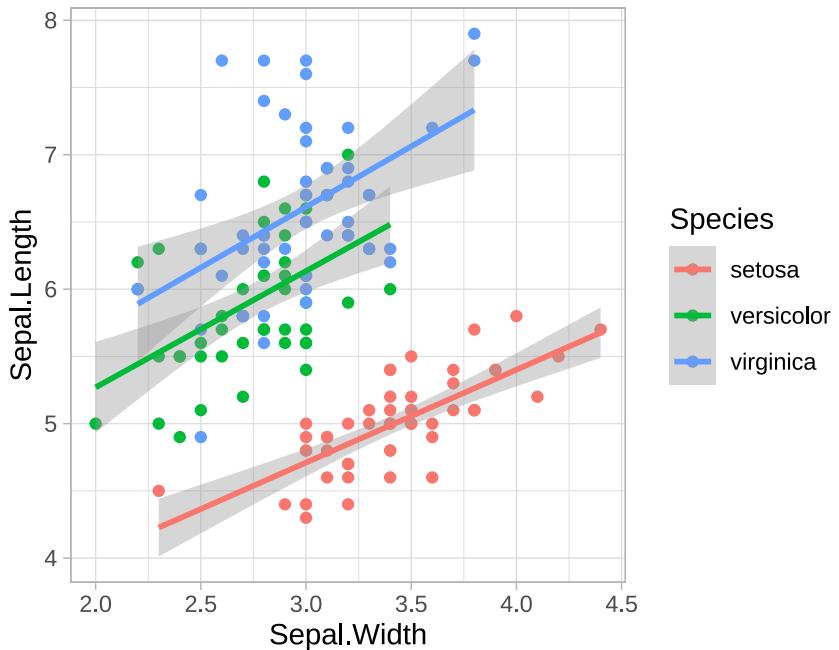
```
p1 + labs(x = "Kelchblattbreite [mm]", y = "Kelchblattlänge [mm]")
```



1132

1133 Oder auch den ganzen Plot anpassen. Dafür gibt es *themes*. Es gibt eine Reihe von vorgefertigten *themes*  
1134 oder man kann diese auch selber erstellen (das ist aber nicht Teil dieses Kurses).

```
p1 + theme_light()
```

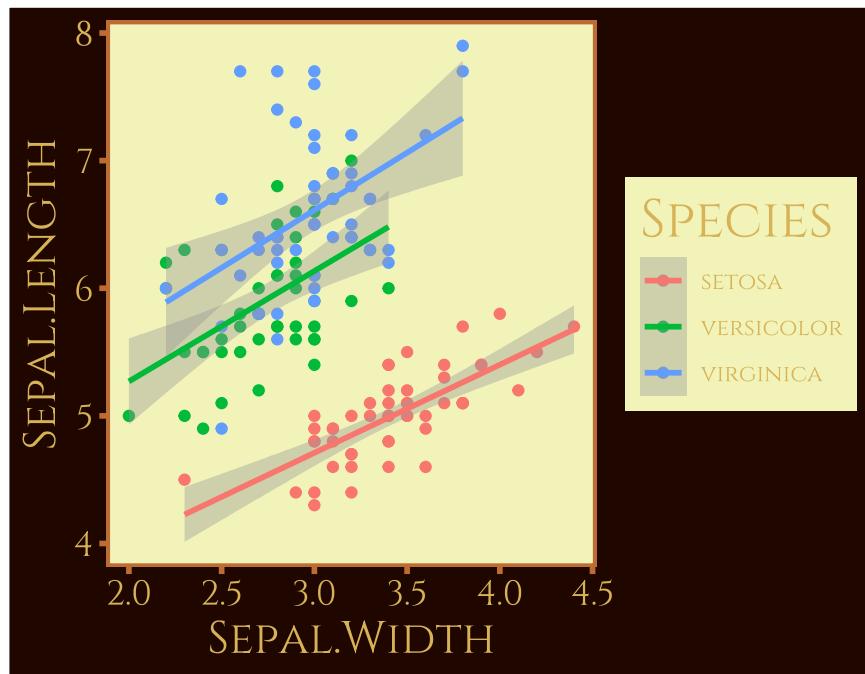


1135

1136 Weitere *themes* sind: `theme_bw()`, `theme_linedraw()` oder `theme_dark()`. Es gibt extra Pakete die viele  
1137 zusätzliche weitere *themes* anbieten. Dazu gehört z.B. das Paket `ggthemes` oder `ThemePark`. Während  
1138 `ggthemes` hauptsächlich durchdachte Grafikkonzepte liefert, sind die Themes aus `ThemePark` eher Popkultur  
1139 und nicht 100 %ig ernst gemeint. `ThemePark` muss zunächst aus GitHub installiert werden. Die Installation

1140 wird auf der GitHub erläutert.

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

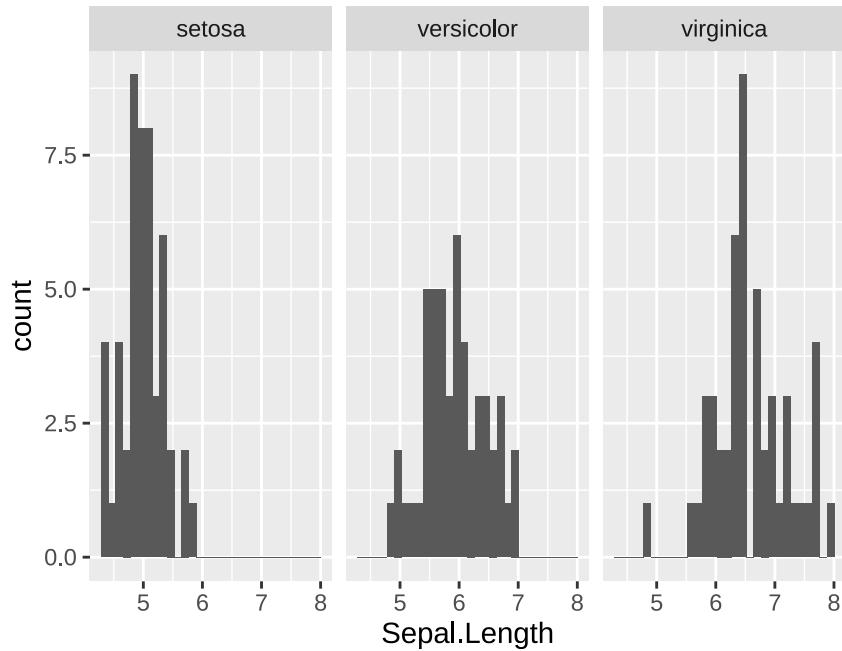


1141

#### 1142 8.4.1 Multipanel Abbildungen

1143 Mit `ggplot2` kann man einfach Abbildungen erstellen, die mehrere Panels haben. Das bedeutet, dass eine  
1144 oder mehrere weitere Variablen gibt, die einen Plot in mehrere Subplots teilt. Dafür gibt es zwei Funktionen:  
1145 `facet_grid()` und `facet_wrap()`.

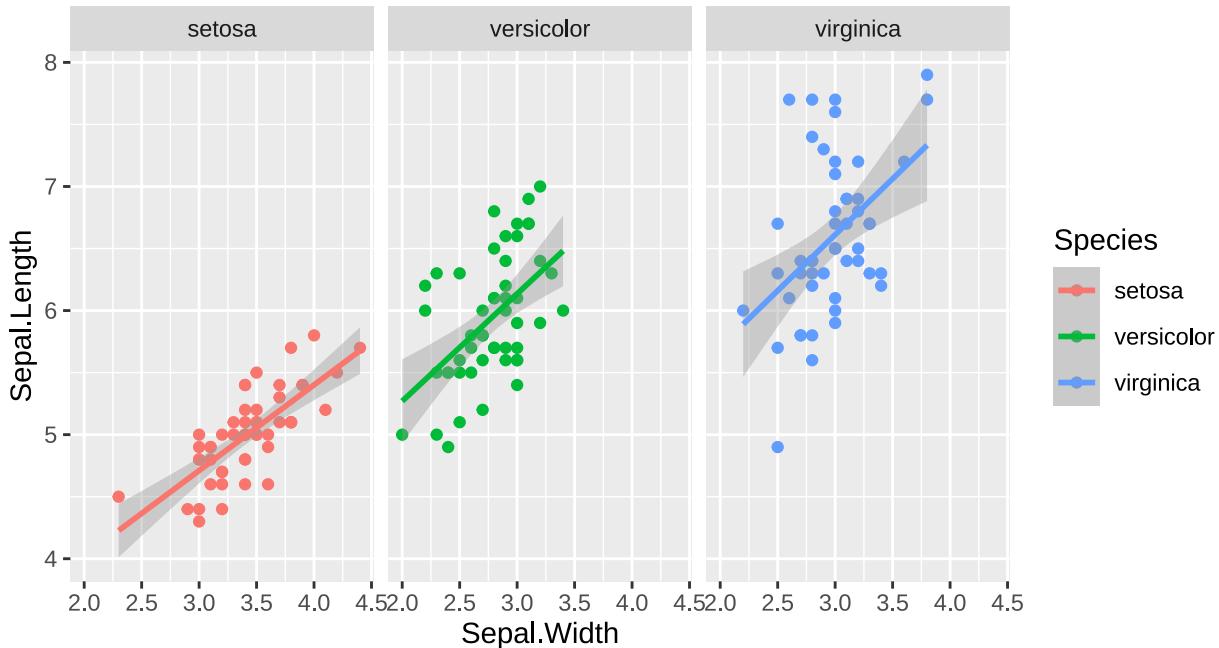
```
ggplot(iris, aes(Sepal.Length)) + geom_histogram() +
 facet_grid(~ Species)
```



1146

1147 Die Funktion `facet_grid()` erzeugt einen *Grid* und beschriftet die Grid-Zeilen und -Spalten, während  
 1148 `facet_wrap()` für jedes Panel (Diagramm) eine eigene Überschrift erzeugt. Beim Arrangieren der Diagramme  
 1149 wird der Vorteil von `ggplot2` gegenüber `base plot` besonders deutlich. Während es im `base plot` System  
 1150 mühsam ist, die Abbildungen zu arrangieren, übernimmt `ggplot2` das Arrangement automatisch und fügt  
 1151 sogar eine automatisch Legende hinzu. Die Achsenabschnitte werden so angepasst, dass die Daten vergleichbar  
 1152 sind.

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

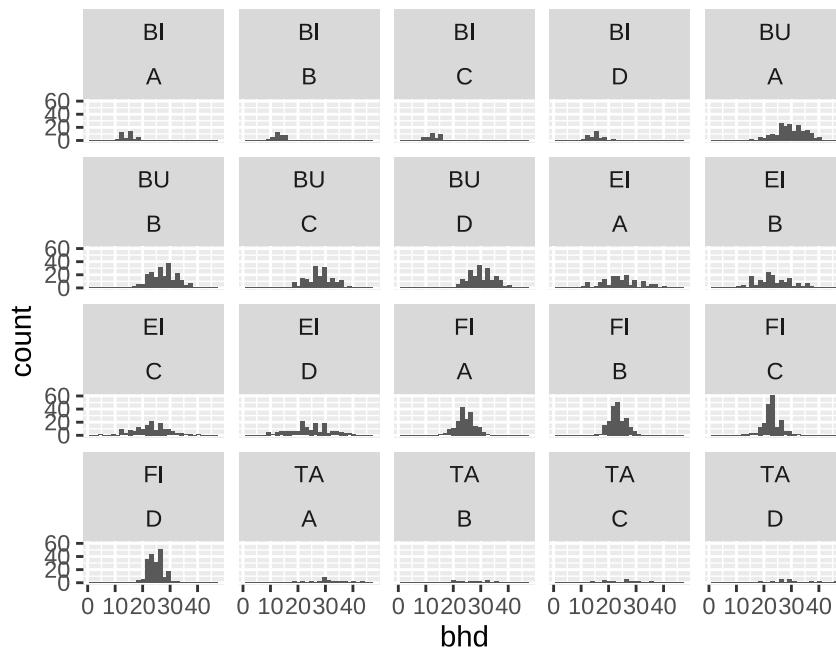


1153

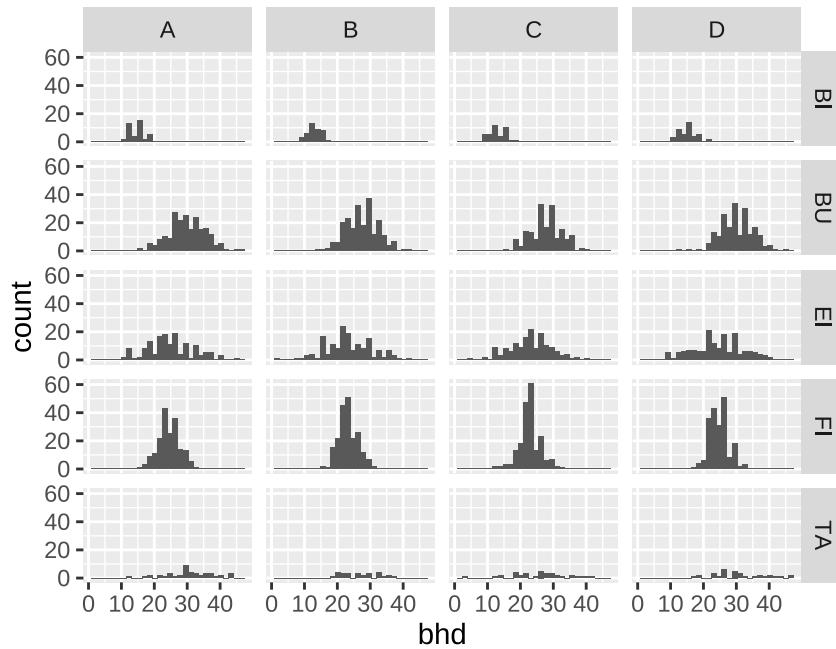
1154

1155 **Aufgabe 22: Multipanel Abbildungen**  
1156

- 1157 Lesen Sie erneut den Datensatz `daten/bhd_1.txt` ein und speichern Sie diesen in die Variable (`dat_bhd`).  
 1158 Erstellen Sie für jede Art und Gebiet ein Histogramm. Welche Unterschiede können Sie feststellen, wenn Sie  
 1159 `facet_grid()` oder `facet_wrap()` verwenden?



1160



1161

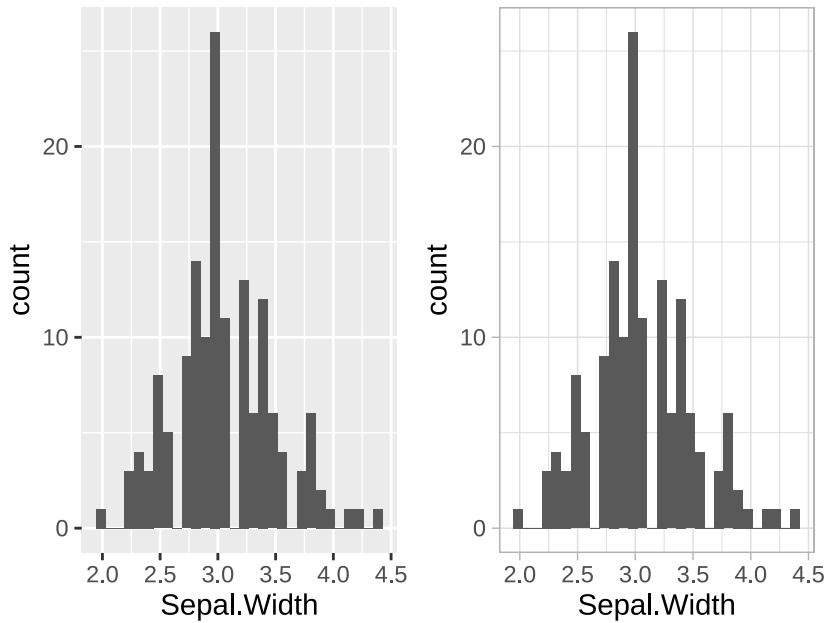
1162 **8.4.2 Plots kombinieren**

- 1163 Es gibt Situationen, in denen **unterschiedliche** Plots miteinander kombiniert werden müssen. Im vorherigen  
 1164 Abschnitt wurde dies immer anhand einer gruppierenden Kovariate gemacht. Aber es gibt auch Situationen, in  
 1165 denen das nicht möglich ist. Beispielsweise wenn ein Histogramm und ein Scatterplot vom gleichen Datensatz  
 1166 zusammengefasst werden sollen. Dafür bietet sich das Paket **patchwork** an<sup>9</sup>.  
 1167 Als erstes können wir zwei (oder natürlich auch mehrere Plots) erstellen. Hier unterscheiden sich die Plots  
 1168 lediglich durch das Aussehen.

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

- 1169 Dann müssen können wir diese Plots ebenfalls mit + zusammenfügen.

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



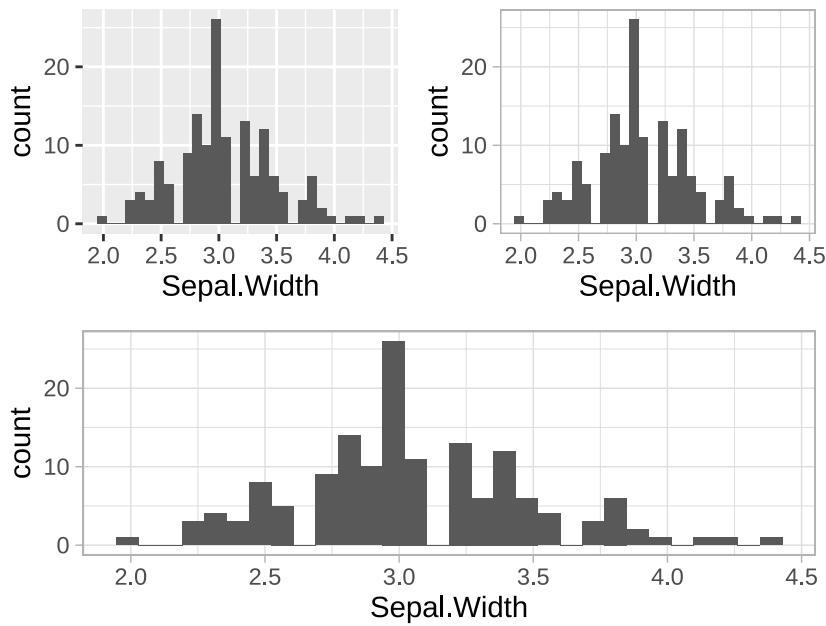
1170

- 1171 Natürlich können auch weitere Plots hinzugefügt werden (auch in unterschiedlichen Dimensionen):

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

---

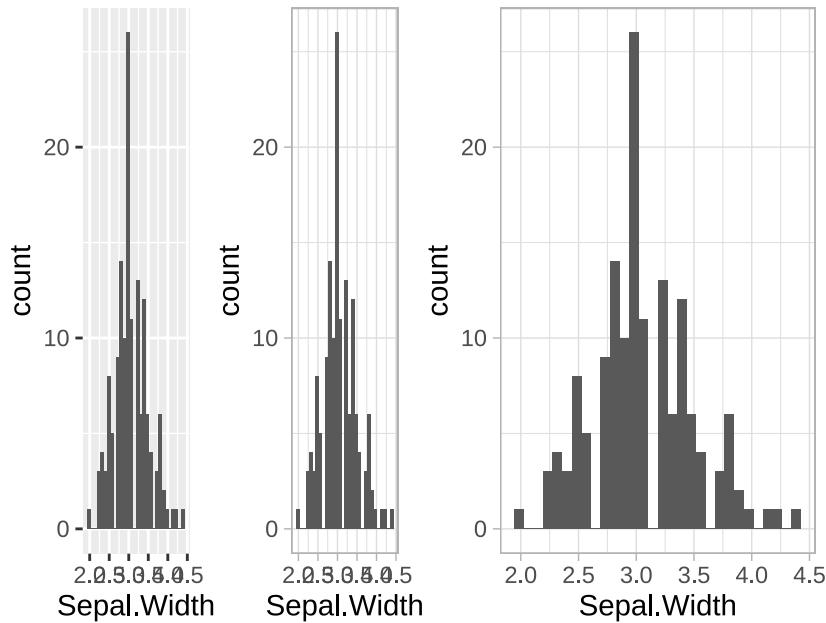
<sup>9</sup>Auch dieses Paket müssen Sie einmalig mit `install.packages("patchwork")` installieren.



1172

1173 Des weiteren können mit | auch Plots gegenüber gestellt werden.

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



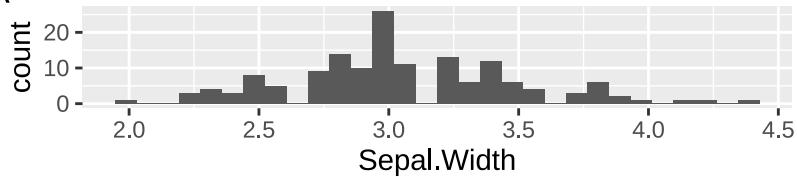
1174

1175 Weitere Optionen können mit `plot_layout()` und `plot_annotation()` angepasst werden. Mit  
1176 `plot_layout()` kann die Anordnung der Plots bestimmt werden (z.B. über die Argument `nrow`  
1177 und `ncol`), sowie deren *relative* Größe (über die Argumente `widths` und `heights`). Mit der Funktion  
1178 `plot_annotation()` können zusätzliche Beschriftungen hinzugefügt werden, wie beispielsweise eine Titel  
1179 (Argument `title`) oder ein Buchstabe/Zahl für jedes Element (Argument `tag_levels`).

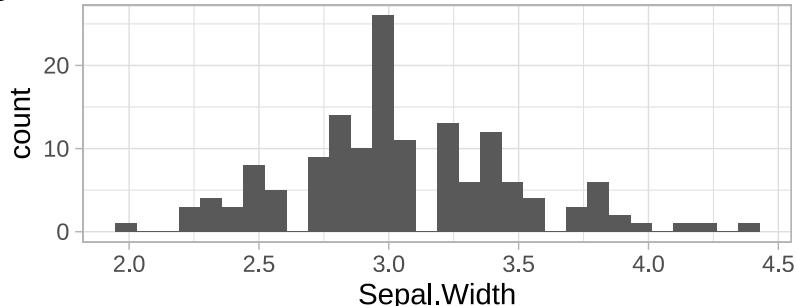
```
p1 + p2 +
 plot_layout(ncol = 1, heights = c(0.3, 0.7)) +
 plot_annotation(title = "Zwei Histogramme", tag_levels = "A")
```

## Zwei Histogramme

A



B



1180

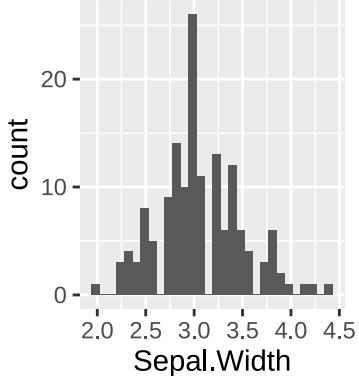
1181

### Aufgabe 23: Mehrere Plots zusammenfügen

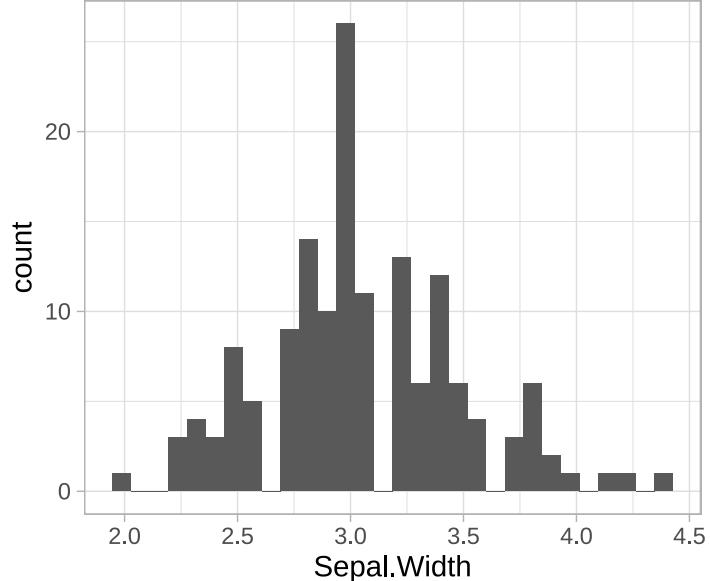
1182

1183 Versuchen Sie die folgende Zusammenstellung der Plots nachzumachen:

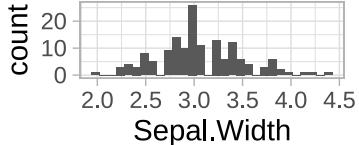
a



c



b



1185

### 8.4.3 Speichern von plots

1186 Sie können mit `ggsave()` eine zwischen-gespeicherte Abbildung exportieren, indem Sie den Variablennamen übergeben. Wenn Sie keine Variable übergeben, wird automatisch die letzte Abbildung gespeichert. Das

- 1189 Dateiformat wird aus dem Dateinamen übernommen. Die Größe der Inhalte können Sie sehr leicht mit den  
1190 Argumenten `width` und `height` oder `scale` gesteuert werden.

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

## 1191 9 Mit Daten arbeiten

### 1192 9.1 dplyr eine Einführung

1193 `dplyr` ist eine Erweiterung von R (ein Paket), die das Ziel hat, den Umgang mit Daten einfacher und  
1194 schneller zu machen.

1195 `dplyr` definiert 5 Verben, um mit Daten zu arbeiten. Diese sind:

- 1196 • `filter`
- 1197 • `select`
- 1198 • `arrange`
- 1199 • `mutate`
- 1200 • `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))
```

1201 Damit die Funktionen aus `dplyr` verwendet werden können, müssen wir als erstes das Paket `dplyr` laden.  
1202 The online Resource [R for Data Science](#) gibt einen sehr guten, tieferen Eindruck in die Arbeit mit `dplyr`  
1203 und erklärt auch die `dplyr` Philosophie noch Mal sehr genau für Beginner.

```
library(dplyr)
```

1204 Sollte dies zu einer Fehlermeldung führen, dann müssen Sie das Paket `dplyr` erst installieren. Dafür müssen  
1205 Sie `einmalig install.packages("dplyr")` installieren.

1206 `dplyr` stellt unterschiedliche Funktionen zum Arbeiten mit Daten zur Verfügung. Es gibt fünf Grundfunktionen  
1207 für die am häufigsten vorkommenden Operationen. Mit der Funktion `filter()` können unterschiedliche  
1208 Beobachtungen gefiltert werden:

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

```
1209 ## id plot bhd alter
1210 ## 1 1 1 50 10
1211 ## 2 2 1 29 30
1212 ## 3 3 2 13 31
1213 ## 4 4 2 23 24
1214 ## 5 5 3 25 25
```

1215 Es können auch mehrere Spalten verwendet werden.

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

```
1216 ## id plot bhd alter
1217 ## 1 2 1 29 30
1218 ## 2 3 2 13 31
1219 ## 3 4 2 23 24
```

```
1220 ## 4 5 3 25 25
```

1221 Natürlich kann genau das gleiche Ergebnis mit dem ‘normalen’ R erreicht werden, dies wäre dann:

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

```
1222 ## id plot bhd alter
1223 ## 2 2 1 29 30
1224 ## 3 3 2 13 31
1225 ## 4 4 2 23 24
1226 ## 5 5 3 25 25
```

1227 Eine weitere Funktion aus dem Paket **dplyr** ist **select()**. Damit können Spalten aus einem **data.frame** ausgewählt werden. Dabei können auch die Spaltennamen unbenannt werden.

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

```
1229 ## bhd
1230 ## 1 50
1231 ## 2 29
1232 ## 3 13
1233 ## 4 23
1234 ## 5 25
```

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

```
1235 ## bhd id
1236 ## 1 50 1
1237 ## 2 29 2
1238 ## 3 13 3
1239 ## 4 23 4
1240 ## 5 25 5
```

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

```
1241 ## BHD id
1242 ## 1 50 1
1243 ## 2 29 2
1244 ## 3 13 3
1245 ## 4 23 4
1246 ## 5 25 5
```

1247 Mit der Funktion **arrange()** können die Beobachtungen in einem **data.frame** sortiert werden.

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

```
1248 ## id plot bhd alter
1249 ## 1 3 2 13 31
1250 ## 2 4 2 23 24
1251 ## 3 5 3 25 25
```

```
1252 ## 4 2 1 29 30
1253 ## 5 1 1 50 10
```

1254 Mit der Funktion `desc()` kann die Anordnung in absteigender Reihenfolge sortiert werden.

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

```
1255 ## id plot bhd alter
1256 ## 1 1 1 50 10
1257 ## 2 2 1 29 30
1258 ## 3 5 3 25 25
1259 ## 4 4 2 23 24
1260 ## 5 3 2 13 31
```

1261 Mit der Funktion `mutate()` kann man eine neue Spalte hinzufügen.

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

```
1262 ## id plot bhd alter bhd_mm fl
1263 ## 1 1 1 50 10 500 1963.4954
1264 ## 2 2 1 29 30 290 660.5199
1265 ## 3 3 2 13 31 130 132.7323
1266 ## 4 4 2 23 24 230 415.4756
1267 ## 5 5 3 25 25 250 490.8739
```

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

```
1268 ## id plot bhd alter mean_bhd
1269 ## 1 1 1 50 10 28
1270 ## 2 2 1 29 30 28
1271 ## 3 3 2 13 31 28
1272 ## 4 4 2 23 24 28
1273 ## 5 5 3 25 25 28
```

1274 Mit der Funktion `summarise()` können Spalten zusammengefasst werden.

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

```
1275 ## mean_bhd mean_sd
1276 ## 1 28 13.63818
```

1277

1278 **Aufgabe 24: Datenmanipulation mit dplyr**

---

1279

- 1280 1. Laden Sie den Datensatz `data/bhd_1.txt`
- 1281 2. Berechnen Sie folgende Werte für alle Einträge und speichern Sie die Ergebnisse in `erg1`
- 1282 • mittlerer `bhd`
- 1283 • maximales `alter`
- 1284 • die Standardabweichung des BHDs
- 1285 • die Anzahl Bäume mit einem BHD > 30

1286 **9.2 Arbeiten mit gruppierten Daten**

1287 Zusätzlich können `mutate` und `summarise` auch auf gruppierte Daten angewendet werden. Dafür müssen  
1288 wir erst die Funktion `group_by()` aufrufen und als Argumente die Variablen übergeben, die die Gruppen  
1289 definieren.

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

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

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

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

summarise(dat, bhd_m = mean(bhd))

bhd_m
1 28

summarise(dat1, bhd_m = mean(bhd))

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

```
1309 ## <dbl> <dbl>
1310 ## 1 1 39.5
1311 ## 2 2 18
1312 ## 3 3 25
```

1313

**Aufgabe 25: dplyr mit gruppierten Daten**

---

- 1316 1. Laden Sie den Datensatz `data/bhd_1.txt`
- 1317 2. Berechnen Sie für jede Baumart und Standort die folgenden Werte:
  - 1318 • mittlerer `bhd`
  - 1319 • maximales `alter`
  - 1320 • die Standardabweichung des BHDs
  - 1321 • die Anzahl Bäume mit einem BHD > 30
- 1322 3. Ordnen Sie das Ergebnis nach Baumart und BHD.

**9.3 pipes oder %>%**

1324 Mit *Pipes* (`%>%`) kann man das Ergebnis einer Funktion einfach an eine nachfolgende Funktion weiterreichen.

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

1325 Wir kennen bis jetzt:

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

1326 ## [1] 3.333333

1327 Mit *Pipes*, die durch das Symbol `%>%` dargestellt werden<sup>10</sup>, können wir das etwas vereinfachen und nacheinander schreiben:

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

1329 ## [1] 3.333333

1330 Oder sogar

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

1331 ## [1] 3.333333

1332

**Aufgabe 26: Pipes %>%**

---

1335 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.

- 1336 1. Laden Sie den Datensatz `data/bhd_1.txt`.
- 1337 2. Berechnen Sie für jede Baumart und Standort die folgenden Werte:
- 1338 • mittlerer `bhd`
  - 1339 • maximales `alter`
  - 1340 • die Standardabweichung des BHDs
  - 1341 • die Anzahl Bäume mit einem BHD > 30
- 1342 3. Ordnen Sie das Ergebnis nach Baumart und BHD.

## 1343 9.4 Joins

1344 Eine weitere häufige Aufgabe beim Daten Management ist es Daten zusammenzuführen. Nehmen Sie an, dass  
1345 wir folgende Aufnahmen gemacht haben

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

1346 und jeder Baum lediglich mit einer `id` versehen wurde. In einer zweiten Tabelle wurden dann weitere Daten  
1347 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")
)
```

1348 Ziel ist es jetzt die Bäume aus `aufnahmen` mit den Informationen aus den `metadaten` zu verbinden. Dazu  
1349 dient `id` als Bindeglied (oft auch Schlüssel genannt).

1350 Dazu gibt es vier Möglichkeiten.

1351 Zur Durchführung gibt es in base R die Funktion `merge()`. Wir werden aber gleich die Funktionen aus dem  
1352 Paket `dplyr` verwenden.

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

1353 ## id bhd art gebiet
1354 ## 1 1 20 <NA> <NA>
1355 ## 2 2 31 Ta A
1356 ## 3 3 74 Bu B

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

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

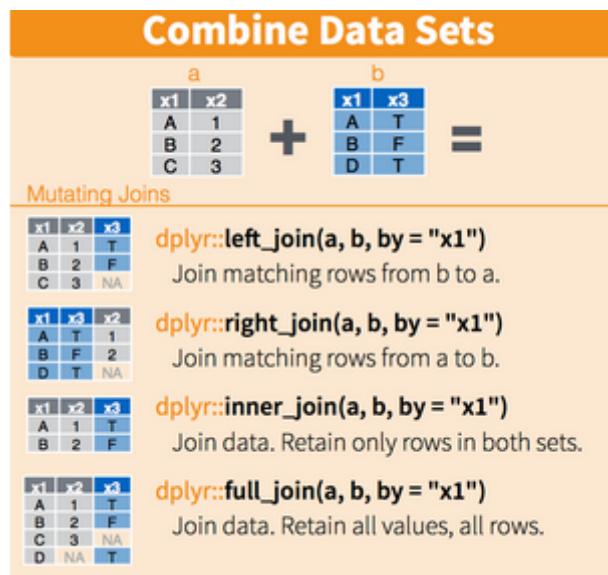


Abbildung 11: Joins (Quelle Rstudio)

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

```
1361 ## id bhd art gebiet
1362 ## 1 2 31 Ta A
1363 ## 2 3 74 Bu B
full_join(aufnahmen, metadaten, by = "id")
```

```
1364 ## id bhd art gebiet
1365 ## 1 1 20 <NA> <NA>
1366 ## 2 2 31 Ta A
1367 ## 3 3 74 Bu B
1368 ## 4 4 NA Bu B
```

1369 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"))
```

```
1370 ## id bhd art gebiet
1371 ## 1 1 20 <NA> <NA>
1372 ## 2 2 31 Ta A
1373 ## 3 3 74 Bu B
```

1374

1375 **Aufgabe 27: Verbinden von Daten**  
1376

- 1377 • Lesen Sie die Datensätze `daten/bhd_2.txt` und `daten/bhd_2_meta.txt` ein.  
 1378 • Stellen Sie sicher, dass es für den `bhd` keine fehlenden Werte gibt (entfernen sie entsprechende Zeilen)  
 1379 • Fügen Sie zu den Metadaten (gespeichert in `bhd_2_meta`) die Anzahl Bäume und den mittleren `bhd`  
 1380 hinzu pro Gebiet.

1381 **9.5 ‘long’ and ‘wide’ Datenformate**

1382 Unter anderem Wickham (2014) empfieilt das Prinzip von *tidy Data*. Nach diesem Prinzip sollten Daten wie  
 1383 folgt organisiert sein:

- 1384 • Jede Zeile ist ein Merkmalsträger/Subjekt/Objekt (z. B. eine Person oder ein Baum).  
 1385 • Jede Spalte ist eine Variable (ein Merkmal), die den Merkmalsträger beschreibt.  
 1386 • In jeder Zelle ist genau ein Wert (Merkmalausprägung), nämlich der Wert, der Variable für den Merk-  
 1387 malsträger.

1388 Zum Beispiel enthalten Spaltennamen oft Informationen, die eigentlich in einer Variable gespeichert werden  
 1389 sollten. Folgendes Beispiel gibt die BHD Messung von 3 Bäumen in 3 Jahren wieder. Der Vorteil von *tidy*  
 1390 Daten ist, dass viele Funktionen diese Form erwarten. Somit müssen sie Ihre Daten nur ein Mal organisieren  
 1391 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)
)
```

1392 Der `tibble` aus dem Paket `tibble` ist ein moderner Data Frame, der sich in seinen Funktionen in das `tidy`  
 1393 Universum einfügt. Wir können auch einfach das Paket `tidyverse` laden, um alle Pakete, die Teil des `tidy`  
 1394 Universums sind gleichzeitig zu laden. Der `tibble` beinhaltet alle Funktionen, die der Data Frame auch  
 1395 beinhaltet und kann auch für genau die gleichen Zwecke verwendet werden. Der `tibble` hat u. A. eine  
 1396 modernere Darstellung im Konsolenoutput.

1397 Diese Daten sind jetzt im `wide`-Format gespeichert und folglich nicht `tidy`, weil Information über die Daten  
 1398 (nämlich das Jahr der Aufnahme in den Spaltennamen gespeichert sind). Besser wäre eine Struktur mit  
 1399 nur drei Spalten: `id`, `jahr` und `bhd`. Um die Daten in so eine Struktur zu bringen, gibt es die Funktion  
 1400 `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
```

```

1403 ## <int> <chr> <dbl>
1404 ## 1 1 bhd2015 30
1405 ## 2 1 bhd2016 31
1406 ## 3 1 bhd2017 34
1407 ## 4 2 bhd2015 31
1408 ## 5 2 bhd2016 31
1409 ## 6 2 bhd2017 32
1410 ## 7 3 bhd2015 32
1411 ## 8 3 bhd2016 33
1412 ## 9 3 bhd2017 33

```

1413 Wenn wir die Spalten für die Variable und den Wert gleich sinnvoll benennen möchten, können wir das über  
 1414 die Argumente `names_to` und `values_to` machen.

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

```

1415 ## # A tibble: 9 x 3
1416 ## id jahr bhd
1417 ## <int> <chr> <dbl>
1418 ## 1 1 bhd2015 30
1419 ## 2 1 bhd2016 31
1420 ## 3 1 bhd2017 34
1421 ## 4 2 bhd2015 31
1422 ## 5 2 bhd2016 31
1423 ## 6 2 bhd2017 32
1424 ## 7 3 bhd2015 32
1425 ## 8 3 bhd2016 33
1426 ## 9 3 bhd2017 33

```

1427 Analog zu der Funktion `pivot_longer()` gibt es auch die Funktion `pivot_wider()`, um vom Daten vom  
 1428 `long`-Format ins `wide`-Format zu transformieren.

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

```

1429 ## # A tibble: 3 x 4
1430 ## id bhd2015 bhd2016 bhd2017
1431 ## <int> <dbl> <dbl> <dbl>
1432 ## 1 1 30 31 34
1433 ## 2 2 31 31 32
1434 ## 3 3 32 33 33

```

1435

---

1436 **Aufgabe 28: Zeitliche Verlauf von BHDs**

---

1438 In der Datei `bhd_3.csv` befinden sich gemessene BHDs (in cm) von unterschiedlichen Bäumen zu unter-  
 1439 schiedlichen Zeitpunkten. Erstellen Sie ein Liniendiagramm, das den zeitlichen (x-Achse) Verlauf der BHDs  
 1440 (y-Achse) für die unterschiedlichen Bäume darstellt.

1441 **9.6 Auswählen von Variablen**

1442 Sobald die Datensätze etwas umfangreicher werden (d. h., es gibt mehrere Spalten in einem `data.frame`),  
 1443 können innerhalb vieler `dplyr`-Funktionen spezielle Funktionen verwendet werden, um Variablen auszuwählen.  
 1444 Wenn die genaue Position der Spalten bekannt ist, kann man mit dem `:`-Operator und der Position Spalten  
 1445 auswählen:

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

1446 ## Sepal.Length Sepal.Width Petal.Length  
 1447 ## 1 5.1 3.5 1.4  
 1448 ## 2 4.9 3.0 1.4  
 1449 ## 3 4.7 3.2 1.3

1450 Diese Vorgehensweise kann gehährlich sein, da sich manchmal Spalten verschieben und sich somit die  
 1451 Positionen ändern. Es besser Spalten immer explizit mit Namen statt mit Nummern anzusprechen.

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

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

1456 `select()` erlaubt es, auch hier den `:`-Operator zu verwenden:

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

1457 ## Sepal.Length Sepal.Width Petal.Length  
 1458 ## 1 5.1 3.5 1.4  
 1459 ## 2 4.9 3.0 1.4  
 1460 ## 3 4.7 3.2 1.3

1461 Es gibt auch einige spezielle Funktionen, um Spalten innerhalb eines `select()`-Aufrufs auszuführen:

- 1462 • `starts_with()`: Hier kann man ein Muster angeben, mit dem ein Text anfangen muss.
- 1463 • `ends_with()`: Diese Funktion ist analog zu `starts_with()`, jetzt wird aber am Ende des Spaltennamens  
 1464 nach dem Muster gesucht.
- 1465 • `contains()`: Hier kann ein Muster übergeben werden, das irgendwo im Spaltennamen sein muss.
- 1466 • `everything()`: Mit dieser Funktion werden alle Spalten ausgewählt.

1467 • `last_col()`: Mit dieser Funktion wird nur die letzte Spalte ausgewählt (dass ist die Spalte, die ganz  
1468 rechts ist).

1469 Sämtliche Auswahlen können mit - umgekehrt werden.

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

1470 ## Sepal.Length Sepal.Width

1471 ## 1 5.1 3.5

1472 ## 2 4.9 3.0

1473 ## 3 4.7 3.2

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

1474 ## Petal.Length Petal.Width Species

1475 ## 1 1.4 0.2 setosa

1476 ## 2 1.4 0.2 setosa

1477 ## 3 1.3 0.2 setosa

1478 `select()` bietet auch noch die Möglichkeit, Spaltennamen zu ändern.

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

1479 ## sep\_width

1480 ## 1 3.5

1481 ## 2 3.0

1482 ## 3 3.2

1483

#### 1484 Aufgabe 29: Auswählen von Spalten

---

1486 In der Datei `messungen_1.csv` sind Messungen von zwei Sensoren enthalten für die ersten vier Monate eines  
1487 Jahres. Führen Sie folgende Abfragen durch:

1488 1. Wählen Sie alle Messungen für Januar aus.

1489 2. Wählen Sie alle Messungen für Januar und März aus.

## 1490 9.7 Einzelne Beobachtungen abfragen (`slice()`)

1491 Mit dem Befehl `slice()` kann man einzelne Beobachtungen (= Zeilen) aus einem `data.frame` abfragen.

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

1492 ## Sepal.Length Sepal.Width Petal.Length Petal.Width Species

1493 ## 1 5.1 3.5 1.4 0.2 setosa

1494 ## 2 4.4 2.9 1.4 0.2 setosa

1495 ## 3 5.1 3.5 1.4 0.3 setosa

1496 Davon gibt es drei nützliche Varianten: 1) `slice_head()` und `slice_tail()`; 2) `slice_max()` und  
 1497 `slice_min()`; 3) `slice_random()`.

1498 `slice_head()` und `slice_tail()` sind analog zu `head()` und `tail()`, aber mit dem entscheidenden Unter-  
 1499 schied, dass Gruppierungen berücksichtigt werden. Wenn keine Gruppierung in den Daten vorhanden ist, gibt  
 1500 es keinen Unterschied.

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

```
1501 ## Sepal.Length Sepal.Width Petal.Length Petal.Width Species
1502 ## 1 5.1 3.5 1.4 0.2 setosa
1503 ## 2 4.9 3.0 1.4 0.2 setosa
```

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

```
1504 ## Sepal.Length Sepal.Width Petal.Length Petal.Width Species
1505 ## 1 5.1 3.5 1.4 0.2 setosa
1506 ## 2 4.9 3.0 1.4 0.2 setosa
```

1507 Sobald jedoch eine gruppierende Variable eingeführt wird, gibt `slice_head()` die ersten n Beobachtungen  
 1508 für jede Gruppe zurück und `head()` für den gesamten Datensatz.

```
base head
```

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

```
1509 ## # A tibble: 2 x 5
1510 ## # Groups: Species [1]
1511 ## Sepal.Length Sepal.Width Petal.Length Petal.Width Species
1512 ## <dbl> <dbl> <dbl> <dbl> <fct>
1513 ## 1 5.1 3.5 1.4 0.2 setosa
1514 ## 2 4.9 3 1.4 0.2 setosa
```

```
dplyr slice_head
```

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

```
1515 ## # A tibble: 6 x 5
1516 ## # Groups: Species [3]
1517 ## Sepal.Length Sepal.Width Petal.Length Petal.Width Species
1518 ## <dbl> <dbl> <dbl> <dbl> <fct>
1519 ## 1 5.1 3.5 1.4 0.2 setosa
1520 ## 2 4.9 3 1.4 0.2 setosa
1521 ## 3 7 3.2 4.7 1.4 versicolor
1522 ## 4 6.4 3.2 4.5 1.5 versicolor
1523 ## 5 6.3 3.3 6 2.5 virginica
1524 ## 6 5.8 2.7 5.1 1.9 virginica
```

1525 `slice_tail()` funktioniert analog zu `slice_head()` mit dem einzigen Unterschied, dass nicht die ersten n

1526 Zeilen zurück gegeben werden sondern die letzten `n` Zeilen.  
 1527 `slice_max()` und `slice_min()` geben die Beobachtung mit dem maximalen bzw. minimalen Wert einer  
 1528 Variable zurück. Auch hier werden Gruppen berücksichtigt.

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

1529 ## Sepal.Length Sepal.Width Petal.Length Petal.Width Species  
 1530 ## 1 7.9 3.8 6.4 2 virginica

1531 Und mit Gruppen:

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

 slice_max(Sepal.Length)
```

1532 ## # A tibble: 3 x 5  
 1533 ## # Groups: Species [3]  
 1534 ## Sepal.Length Sepal.Width Petal.Length Petal.Width Species  
 1535 ## <dbl> <dbl> <dbl> <dbl> <fct>  
 1536 ## 1 5.8 4 1.2 0.2 setosa  
 1537 ## 2 7 3.2 4.7 1.4 versicolor  
 1538 ## 3 7.9 3.8 6.4 2 virginica

1539 `slice_min()` funktioniert genau gleich, nur dass die Beobachtung (=Zeile) mit dem kleinsten Wert einer  
 1540 Variable zurück gegeben wird.

1541 Die Funktion `slice_sample()` erlaubt es zufällige Beobachtungen zu ziehen. Dabei kann über das Argument `n`  
 1542 die Anzahl an Beobachtungen angegeben werden oder über das Argument `prop` der Anteil an Beobachtungen.

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

1543 ## Sepal.Length Sepal.Width Petal.Length Petal.Width Species  
 1544 ## 1 6.5 2.8 4.6 1.5 versicolor  
 1545 ## 2 6.3 3.3 4.7 1.6 versicolor  
 1546 ## 3 7.2 3.2 6.0 1.8 virginica  
 1547 ## 4 4.9 3.6 1.4 0.1 setosa  
 1548 ## 5 6.0 2.7 5.1 1.6 versicolor

1549 Das Ergebnis ist bei jedem von Ihnen anders, da es sich um eine zufällige Ziehung handelt. Wenn Sie diese  
 1550 Ergebnisse wiederholen möchten, können Sie über `set.seed()` die zufällige Ziehung reproduzierbar machen.

```
set.seed(123)

slice_sample(iris, n = 5)
```

1551 ## Sepal.Length Sepal.Width Petal.Length Petal.Width Species  
 1552 ## 1 4.3 3.0 1.1 0.1 setosa  
 1553 ## 2 5.0 3.3 1.4 0.2 setosa  
 1554 ## 3 7.7 3.8 6.7 2.2 virginica  
 1555 ## 4 4.4 3.2 1.3 0.2 setosa  
 1556 ## 5 5.9 3.0 5.1 1.8 virginica

1557 Wenn beispielsweise 5% der Beobachtungen gezogen werden sollen, kann dies so gemacht werden:

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

```
1558 ## Sepal.Length Sepal.Width Petal.Length Petal.Width Species
1559 ## 1 7.7 3.8 6.7 2.2 virginica
1560 ## 2 5.5 2.5 4.0 1.3 versicolor
1561 ## 3 5.5 2.6 4.4 1.2 versicolor
1562 ## 4 6.5 3.0 5.2 2.0 virginica
1563 ## 5 6.1 3.0 4.6 1.4 versicolor
1564 ## 6 6.3 3.4 5.6 2.4 virginica
1565 ## 7 5.1 2.5 3.0 1.1 versicolor

1566 slice_sample() berücksichtigt ebenfalls Gruppen. Mit den Argumenten replace und weight_by dann die
1567 Zufallsziehung genauer spezifiziert werden. replace sagt, ob eine gezogenen Beobachtung wieder zurück gelegt
1568 wird oder nicht. Mit dem Argument weight_by können optional gewichtete für jede Beobachtung vergeben
1569 werden.
```

1570

### 1571 Aufgabe 30: Daten beschreiben

---

1573 Verwenden Sie den Datensatz `bhd_1.txt` und finden Sie für jedes Gebiet und Art die Beobachtung mit
1574 kleinsten BHD.

## 1575 9.8 Spalten trennen

1576 Ein gut geplanter Datensatz besteht aus Beobachtungen (Zeilen), Variablen (Spalten) und in jeder Zelle ist
1577 immer ein **genau** ein Wert gespeichert. Leider gibt es oft Datensätze, bei denen mehr als ein Wert pro Zelle
1578 gespeichert wurde. Die Funktion `separate()` kann helfen solche Daten zu trennen.

1579 Wir verwenden einen erfunden Datensatz zu Beobachtungen von Tieren und einer geschätzten Distanz zu
1580 diesen Tieren.

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

1581 In der Spalte `beobachtung` sind zwei Informationen gespeichert: Die Distanz zur Beobachtung und die Art.
1582 Das ist ungünstig, weil wir so weder nach Tierart noch nach distanz filtern können (nicht *tidy*). Mit der
1583 Funktion `separate()`, können wir Beobachtungen einer Spalte in mehrere Spalten trennen. Dafür muss der
1584 Spaltennamen (Argument `col`), die neuen Spaltennamen (Argument `into`) und das Trennzeichen (Argument
1585 `sep`) angegeben werden.

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

1586 ## # A tibble: 4 x 3

```
1587 ## id Distanz Art
1588 ## <int> <chr> <chr>
1589 ## 1 1 10m " Reh"
1590 ## 2 2 100m " Reh"
1591 ## 3 3 20m " Fuchs"
1592 ## 4 4 40 "Reh"

1593 Nach dem Aufruf von separate() gibt es zwei neue Spalten (Distanz und Art), die die alte Spalte
1594 beobachtung ersetzen.
```

1595

---

1596 **Aufgabe 31: Aufräumen**

---

1598 Verwenden Sie den folgenden Datensatz und bringen Sie ihn in eine Form, die sicherstellt dass

- 1599 • jede Zelle genau einen Wert enthält.  
1600 • jede Zeile eine Beobachtung ist.  
1601 • 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")
)
```

## 1602 10 Arbeiten mit Text

1603 Bis jetzt haben wir fast ausschließlich mit Zahlen oder Abbildungen gearbeitet. R bietet aber auch viele  
 1604 Werkzeuge, um mit Text zu arbeiten. Wir wollen hier ein paar Funktionen dafür vorstellen. Als erstes sollte  
 1605 nochmals klar gestellt werden, was eigentlich ein Text ist. In R ist alles, das innerhalb von doppelten ("") oder  
 1606 einfachen ('') Anführungszeichen geschrieben ist, Text.

1607 Anbei einige Beispiele:

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

1608 Wichtig ist hier zu sehen, dass z nicht als Zahl sondern, als Text interpretiert wird (siehe Datentypen).

```
z + 1
```

1609 ## Error in z + 1: nicht-numerisches Argument für binären Operator

1610 Wenn man sicher ist, dass es sich bei einem Textobjekt um eine Zahl handelt, kann man dies mit der Funktion  
 1611 `as.numeric()` in eine Zahl umwandeln.

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

1612 ## [1] 31

1613 Aber mit a führt dies wieder zu einem NA-Wert, da a nicht in eine Zahl umgewandelt werden kann.

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

1614 ## Warning: NAs durch Umwandlung erzeugt

1615 ## [1] NA

### 1616 10.1 Arbeiten mit Text

1617 Wir wollen erst einmal drei Funktionen besprechen, die es erlauben mit Text zu arbeiten. Die Funktion  
 1618 `nchar()`<sup>11</sup> gibt an wie viele Zeichen ein Text hat. Also z.B.

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

1619 ## [1] 5

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

1620 ## [1] 2

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

1621 ## [1] 20

1622 Die Funktion `paste()` erlaubt es verschiedene Variablen mit Text zu verbinden. Wenn wir z. B. die Variablen  
 1623 `vorname <- "Eva"` und `name <- "Meier"` haben und wir wollen eine neue Variable `full_name <- Eva`

---

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

1624 "Meier" erzeugen, dann kann das mit der Funktion `paste()` gemacht werden.

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

1625 ## [1] "Eva Meier"

1626 Die Funktion `paste()` hat das Argument `sep`, das auf ein Leerzeichen ( ) gesetzt ist, aber auch anders sein  
1627 kann und das Trennzeichen definiert.

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

1628 ## [1] "Eva, Meier"

1629 Die Funktion `substr()` erlaubt es am Anfang oder Ende eines Wortes etwas abzuschneiden. Dabei muss  
1630 immer die Anfangs- (`start`) und Endposition (`stop`) angegeben werden.

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

1631 ## [1] "Hal"

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

1632 ## [1] "allo"

1633

---

### 1634 Aufgabe 32: Arbeiten mit Text 1

---

1636 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")
```

1637 1. Aus wie vielen Buchstaben besteht jedes Wort?

1638 2. Finden Sie das längste Wort.

1639 3. Wie viel Prozent der Wörter fangen mit einem S an?

1640 4. Fügen Sie jedem Wort seine Position im Vektor hinzu. Beispielsweise soll aus `Vogel` "2. Vogel" werden  
1641 usw.

## 1642 10.2 Finden von Textmustern

1643 Mit der Funktion `grep()` können Muster in einem Text gefunden werden. Wenn wir beispielsweise folgenden  
1644 Vektor mit Textelementen haben.

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

1645 Und wir wollen alle Straßennamen die ein `weg` haben abfragen, dann können wir folgenden Befehl ausführen:

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

1646 ## [1] 2

1647 Im zweiten Element von `txt` kommen die Zeichen `Weg` vor. Beachte, in der Standardeinstellung wird zwischen  
1648 Groß- und Kleinschreibung unterschieden. Dies kann mit dem Argument `ignore.case = TRUE` angepasst  
1649 werden.

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

1650 ## [1] 1 2

1651 Mit der Funktion `sub` können Zeichen innerhalb einer Zeichenkette ersetzt werden.

1652 So ersetzt der folgende Ausdruck `ae` mit `ä`.

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

1653 ## [1] "Friedländer Weg"

1654 Wenn allerdings das zu ersetzende Zeichen mehr als einmal vorkommt und beide Instanzen ersetzt werden  
1655 sollen braucht man die Funktion `gsub`.

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

1656 ## [1] "Friedländer Weg und Reinhäuser Landstrasse sind zwei Strassen in Goettingen."

1657 Mit `sub()` wird nur das erste `ae` ersetzt, während `gsub()` alle `ae` mit einem `ä` ersetzt.

```
gsub("ae", "ä", txt)
```

1658 ## [1] "Friedländer Weg und Reinhäuser Landstrasse sind zwei Strassen in Goettingen."

1659 Oft ist der genaue Ausdruck den man finden möchte jedoch Variabel. Beispielsweise möchte man alle Wörter  
1660 mit einem Umlaut oder Zahlen finden möchte, kann man das oft abkürzen. Dafür gibt es reguläre Ausdrücke.  
1661 Wir werden hier nur ein paar beispielhafte Anwendungen besprechen.

1662 Sowohl in den Funktionen `grep()` als auch `(g)sub()` kann mit anstatt dem Muster (immer das erste  
1663 Argument) aus ein regulärer Ausdruck angegeben werden. Mit `[1-9]` sind alle Zahlen von 1 bis 9 gemeint.

1664 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)
```

1665 ## [1] 1 3

1666 Damit lässt sich auch das Problem mit groß- und kleingeschriebenen Wörtern lösen.

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

```
1667 ## [1] 1 2
```

1668

1669 **Aufgabe 33: Arbeiten mit Text 2**

---

1670 1671 Verwenden Sie den folgenden Vektor:

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

1672 1. In wie vielen Wörtern kommt der Doppellaut au vor?

1673 2. Ersetzen Sie in allen Wörtern alle au mit \_ \_.

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

```
1674 ## [1] 5 6 13
```

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

```
1675 ## [1] "Versicherung" "Methoden" "Fluss" "Rudel" "B_ _m"
1676 ## [6] "H_ _s" "Foto" "Auffahrt" "Auto" "Handy"
1677 ## [11] "Teller" "Kalender" "Aufb_ _"
```

## 1678 11 Arbeiten mit Zeit

1679 Für den Computer bzw. R ist ein Datum erst einmal nichts anderes als ein Text. Für uns ist es sofort  
 1680 klar, dass der "13.2.2021" der 13. Februar 2021 ist, für den Computer zunächst nicht. Wir müssen R also  
 1681 irgendwie sagen, dass die 13 der Tag ist, die 2 der Monat und 2021 das Jahr. Dass der Computer die einzelnen  
 1682 Komponenten erkennt, und in einen Datentyp eigens für Zeitformate überführt, nennt man *parsen*<sup>12</sup>. Durch  
 1683 das *parsen* wird die Variable in den Datentyp **Date** überführt. Das Arbeiten mit Datum und Zeit kann  
 1684 kann anfangs sehr mühsam sein und viele Zeit-spezifischen Datenoperationen lassen sich auch mit den  
 1685 Basis-Datentypen durchführen. Sobald man einige Grundfertigkeiten erworben hat, stellt man jedoch fest,  
 1686 dass die Arbeit mit dem Zeitformat-Datentyp schneller und effizienter funktioniert. Starten Sie am besten  
 1687 gleich mit "echten" Zeit-Datentypen und versuchen Sie nicht, sich irgendwie mit Text und numerischen  
 1688 Datentypen selbst etwas zu basteln. Der erste Schritt ist immer ein Datum zu *parsen*. Wir verwenden dafür  
 1689 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)
```

1690 **lubridate** bietet eine Vielzahl von Funktionen zum *parsen* von Datum und Zeit, die sich aus:

- 1691 • y für Jahr,
- 1692 • m für Monat,
- 1693 • d für Tag,
- 1694 • h für Stunde,
- 1695 • m für Minute und
- 1696 • s für Sekunde

1697 zusammen setzten. Alle Funktionen nehmen als erstes Argument ein Textstring. Wenn wir z.B. den String  
 1698 2020-01-20 parsen wollen können wir das mit der Funktion **ymd** machen.

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

1699 ## [1] "2020-01-20"

1700 Dabei erkennt **lubridate** in der Regel die Trennzeichen:

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

1701 ## [1] "2020-01-20"

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

1702 ## [1] "2020-01-20"

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

1703 ## [1] "2020-01-20"

1704 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")

1705 ## [1] "2020-01-20"
1706 Jetzt stellt sich die Frage, was der Vorteil ist, wenn R ein Datum parst.
d <- dmy("20.1.2020")

1707 Wir können jetzt mit d arbeiten und einzelne Komponenten extrahieren.
day(d)

1708 ## [1] 20
month(d)

1709 ## [1] 1
year(d)

1710 ## [1] 2020
1711 Oder auch Zeiteinheiten hinzufügen oder abziehen.
d + days(10)

1712 ## [1] "2020-01-30"
d - years(20)

1713 ## [1] "2000-01-20"
d + hours(25)

1714 ## [1] "2020-01-21 01:00:00 UTC"

```

1715

---

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

---

- 1716 • Parsen Sie folgende Zeitangaben 23.1.2020, 13.2.1992 20:55:23, Mar/3/97 und 10.7.2020 19:15  
1717 und speichern Sie diese in einen Vektor d.
- 1718 • Extrahieren Sie nun aus jedem Element aus d das Jahr und die Stunde.  
1719 • Fügen zu jedem Element in d 10 Tage hinzu.

**11.1 Arbeiten mit Zeitintervallen**

- 1720 Mit zwei Zeitpunkten lassen sich Zeitintervalle (**Periods**) erstellen, dafür können wir die Funktion **interval()**  
1721 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)
```

1725 Wir können jetzt mit `int` arbeiten und beispielsweise das Intervall verschieben,

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

1726 ## [1] 2023-03-18 UTC--2024-03-18 UTC

1727 die Länge des Intervalls berechnen

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

1728 ## [1] 31536000

1729 oder testen, ob ein Datum innerhalb des Intervalls liegt.

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

1730 ## [1] TRUE

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

1731 ## [1] FALSE

1732 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)
```

1733 ## # A tibble: 1 x 1

1734 ## a

1735 ## <date>

1736 ## 1 2020-07-01

1737 `%within%` funktioniert genauso mit Vekotren oder mit mehreren Intervallen. Wir könnten also zwei Intervalle  
1738 definieren (z.B. Ostern und Pfingsten).

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

1739 Und Überprüfen welche Termine in eines der zwei Intervalle fallen.

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

```
Ostern
```

```
termine %within% ostern
```

1740 ## [1] FALSE TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE

```
Pfingsten
termine %within% pfingsten

1741 ## [1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE TRUE FALSE FALSE
1742 Der Zeit-Datentyp wird auch oft benutzt, um den Zeitbedarf von längeren Berechnungen zu messen.

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

1743 ## [1] 0.4999484
t2 <- now()
int_length(interval(t1, t2))

1744 ## [1] 0.6111724
```

1745 **11.2 Formatieren von Zeit**

1746 Für die Ausgabe in Berichten oder Grafiken soll das Datum oft in einer speziellen Form dargestellt werden.

1747 Die Funktion `format()` bietet Möglichkeiten ein Datumsobjekt zurück in Text umzuwandeln.

1748 Ein Beispiel wäre `ymd("2021-2-10")` als 10.2.21 auszugeben.

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

1749 ## [1] "21.02.21"

1750 Dabei handelt sich bei `%d.%m.%y` um Abkürzungen für die unterschiedlichen Komponenten eines Datumobjekts.

1751 Siehe dazu die Hilfeseite von `strptime (help(strptime))`.

1752

---

1753 **Aufgabe 35: Arbeiten mit Intervallen**

1755 Wie viele Einträge aus dem Vektor `v1` befinden sich in einem Intervall, das zwischen dem 1.3.2021 und dem  
1756 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"
)
```

1757 **11.3 Zeitreihen**

1758 In der Forstbranche haben wir oft mit Zeitreihen zu tun. Zeitreihen sind Variablen, für die in zeitlichen  
1759 Intervallen Daten vorliegen. In der Regel monatlich, vierteljährig oder jährlich, wobei die Intervalle zwischen  
1760 den Messungen bei Zeitreihen immer gleich lang sind. Wiederholungsmessungen von Forsteinventuren (Forstein-  
1761 richtungen, Betriebsinventuren, die meisten forstl. Experimente, die BWI, ...) sind also in der Regel keine

1762 Zeitreihen in engeren Sinne. Turnusmäßig vermessene Versuchsflächen, wie sie z. B. die Versuchsanstalten  
 1763 unterhalten oder jährlich gemeldete Holzpreise jedoch schon.

1764 Zeitreihen unterscheiden sich nicht nur technisch, sondern auch inhaltlich fundamental von den uns schon  
 1765 bekannten Daten. Statistisch gesehen haben Zeitreihen Besonderheiten gegenüber anderen Variablen, da  
 1766 Sie von Ihrer eigenen Vergangenheit abhängen (autokorriert sind) und auch die Abhängigkeit anderer  
 1767 Variablen in der Regel nur sinnvoll ist, wenn deren Vergangenheit mitberücksichtigt wird (cross-correlation).

1768 Konventionelle Statistik ist oft nicht möglich, um Zeitreihen zu analysieren. Selbst ein ordinärer arithmetischer  
 1769 Mittelwert ist schon nicht mehr geeignet, um Zeitreihen statistisch zu beschreiben. Angefangen mit der  
 1770 Datendarstellung gibt es in R deshalb spezifische Zeitreihen-Funktionen. Aus diesem Grund sollten Sie  
 1771 Zeitreihen als solche speichern. Viele Funktionen erkennen den Datentyp und führen entsprechend spezifische  
 1772 Zeitreihen-Operationen durch, wenn ihnen Daten vom Typ "Zeitreihe" übergeben werden. Laden wir z. B.  
 1773 die Holzpreise für Fichte 2b (das sog. Leitsortiment, Fichenholz mit einem Mittendurchmesser von 20 bis 25  
 1774 cm), das Holzaufkommen dieses Sortiments (Einschlagsvolumen) und die Preise für Nadelholz vom  
 1775 statistischen Bundesamt<sup>14</sup>. Wir laden die Daten zunächst als csv ein:

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

1776 Diese 3 Zeitreihen bilden zusammen ein klassisches Marktmodell mit dem Preis eines homogenen Gutes  
 1777 (Leitsortimentspreis), dem Angebot (Holzeinschlag) und der Nachfrage (Schnittholzpreis). Mit der Funktion  
 1778 **ts** werden die Daten in ein Zeitreihenobjekt überführt (*pasrse*). Die Spalte mit den Jahren ist dann nicht mehr  
 1779 nötig, da die Informationen zur Zeit keine eigene Variable mehr sind, sondern als sog. Metainformationen in  
 1780 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])
```

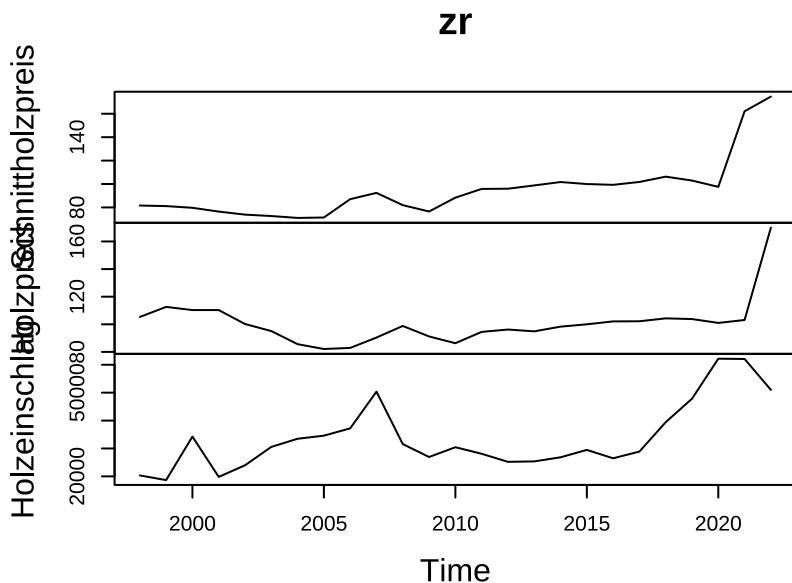
**typeof(zr)** # Zeitreihen sind sog. S3 Objekte. Das heißt, Zeitreihen gehören nicht zu den

1781 ## [1] "double"  
 # Basis Datentypen (siehe Kapitel Variablen, Funktionen und Datentypen),  
 # sondern sind eine Unterkategorie des Datentyps "Liste".

1782 Die wichtigsten Argumente sind - **data** Vektor oder Matrix, der nur die Daten enthält - **start** Startzeitpunkt -  
 1783 **frequency** Anzahl der Beobachtungen pro Zeiteinheit, also z. B. 12 bei monatlichen oder 4 bei quartalsweisen  
 1784 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

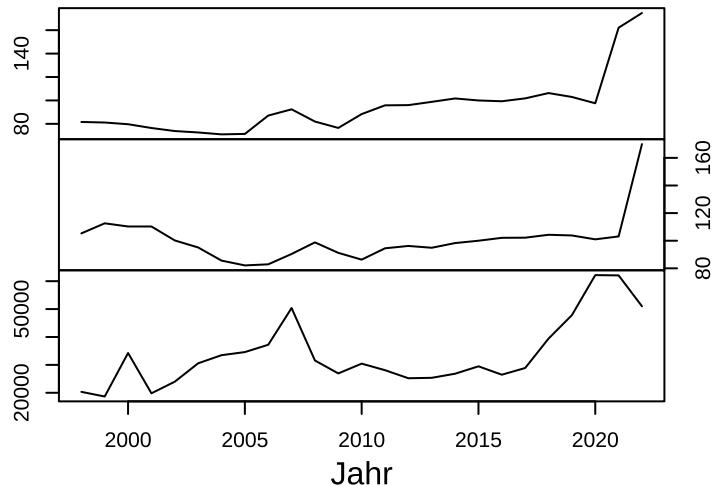


1785

1786 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")
```

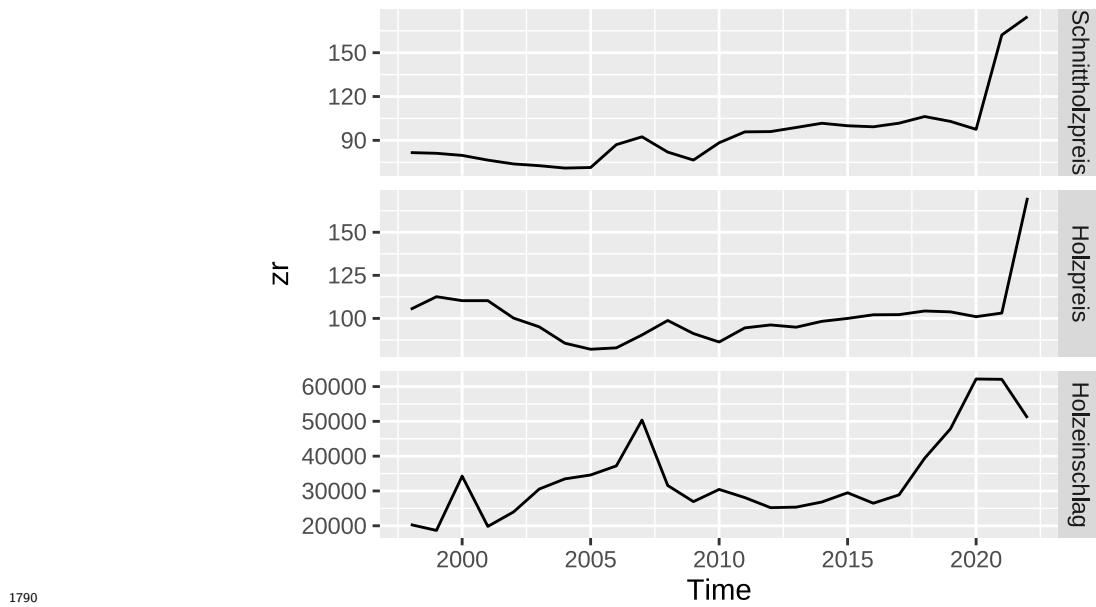
## Holzmarktentwicklung seit 1998



1787

1788 Beide Plot-Philosophiebn haben eine Zeitreihen-Funktion. Das Paket `ggfortify` ermöglicht automatisierte  
1789 Zeitreihenplots im `ggplot2` Stil. Damit ist auch das Problem der y-Achsenbeschriftungen gelöst.

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

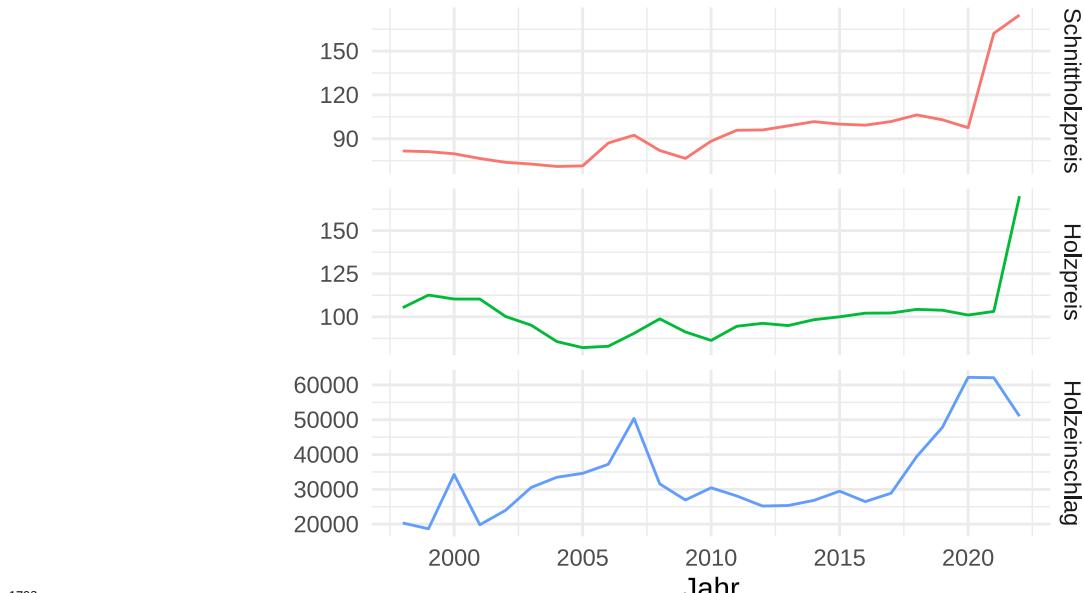


1790 1791 Wir können die Abbildung im `ggplot2` Stil ändern. Hier nur ein Minimalbeispiel zur Veranschaulichung.

1792 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()
```

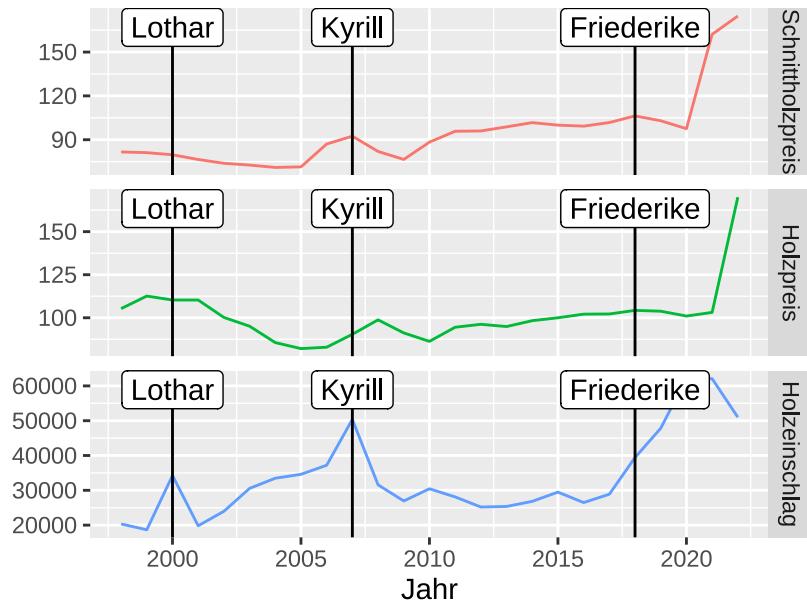


1793

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

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

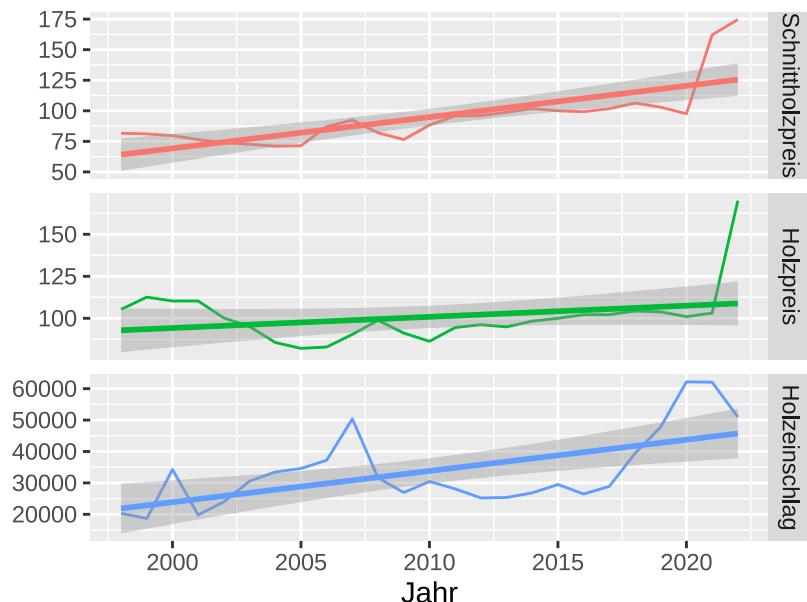
```
annotate(x = 2007, y = +Inf, label = "Kyrill", vjust = 1, geom = "label") +
 annotate(x = 2018, y = +Inf, label = "Friederike", vjust = 1, geom = "label")
```



1794

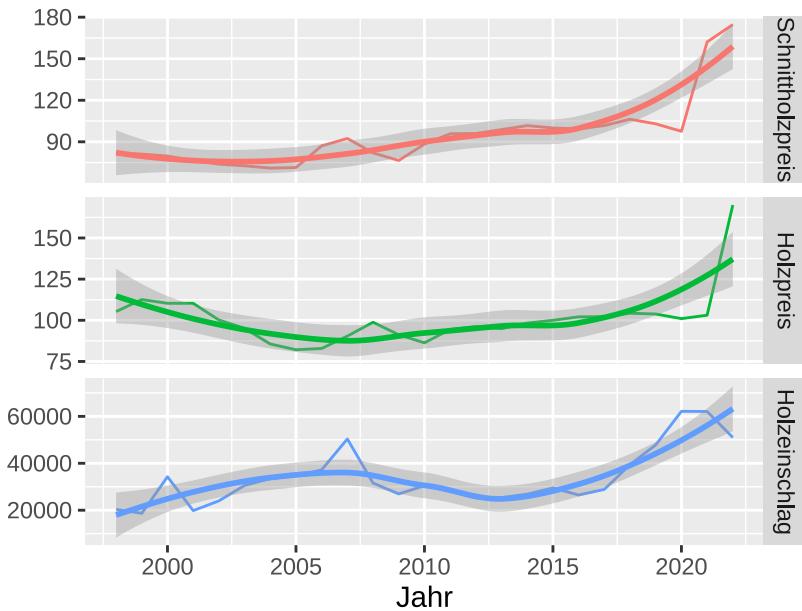
1795 Eine Trendlinie macht hier offensichtlich keinen Sinn. Die Trendlinie ist eine lineare Regression, also eine  
 1796 ordinäre Statistik, die wie eingangs erwähnt für Zeitreihen ungeeignet ist. Daher verwenden wir den sog.  
 1797 Loess-Glättter, um einen Trend in die Daten zu legen. Der Loess-Glättter ist ein sehr flexible Kurve. Wir sehen  
 1798 hier beispielsweise, dass der Leitholzpreis träge oder gar nicht auf das Angebot reagiert. Die Nachfrage jedoch  
 1799 zumindest in der einen Periode, in der sie stark steigt, den Holzpreis jedoch mit zeitlichem Verzug stark  
 1800 ansteigen lässt. Dieser visuelle Eindruck lässt sich durch spezifische Zeitreihen-Regressionen schätzen.

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



1801

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



1802

## 1803 12 Aufgaben Wiederholen (for-Schleifen)

1804 Um einfache Programme zu schreiben, müssen Sie den Ablauf der Programmcodes kontrollieren können.  
 1805 Kontrollieren bedeutet in diesem Zusammenhang, dass Codeabschnitte nur unter definierten Bedingungen ab-  
 1806 laufen. Sie programmieren also zwei Sachen. 1) den Code selbst und 2), die Bedingungen die erfüllt sein müssen,  
 1807 damit der Code ausgeführt wird. Der Code muss so generisch geschrieben sein, dass er komplett durchläuft,  
 1808 auch wenn unterschiedliche Dateneingaben gemacht werden. Diese Kontrollbedingungen ermöglichen es Ihnen  
 1809 generisch zu programmieren. Sie schreiben Ihren Code also nicht speziell maßgeschneidert für ein Problem,  
 1810 sondern so generell, dass er für mehrere Auswertungen funktioniert. Um dies zu gewährleisten, müssen Sie  
 1811 bestimmte Situationen vorhersehen und abfangen. Hierbei helfen Ihnen Kontrollstrukturen (**Control Flow**).  
 1812 Grundsätzlich gibt es **Control Flow** Funktionen zur Wiederholung von Codeblöcken (Schleifen) und logische  
 1813 Bedingungen (bedingte Anweisung).

### 1814 12.1 Schleifen

1815 Bis jetzt wurden alle Skripte einfach der Reihe nach abgearbeitet und zwischendurch bestimmte Programmteile,  
 1816 je nach Situation, selbstständig ausgeführt oder übersprungen. Mit einer Schleife kann man erreichen, dass  
 1817 eine Gruppe von Befehlen (der sog. Schleifenrumpf) mehrfach abgearbeitet wird, zum Beispiel wenn bestimmte  
 1818 Auswertungsschritte auf mehrere Datensätze oder Variablen angewendet oder Funktionen mit unterschiedlichen  
 1819 Parametern oder Startwerten aufgerufen werden sollen. Weitere Anwendungsmöglichkeiten sind iterative  
 1820 Algorithmen, in denen die Eingabewerte des aktuellen Iterationsschrittes von einem vorherigen abhängig sind.  
 1821 Besonders in Simulationen kommen Schleifen häufig zum Einsatz, da große Anzahlen von Wiederholungen  
 1822 benötigt werden.

1823 Man unterscheidet zwischen zwei Arten von Schleifen: Bei den **for()**-Schleifen steht die Anzahl der Wieder-  
 1824 holungen schon beim Eintritt in die Schleife fest, während die **while()**-Schleifen so lange ausgeführt werden,  
 1825 bis eine Bedingung nicht mehr wahr ist. Mit der Funktion **break** wird eine Schleife abgebrochen und die  
 1826 Programmausführung wird nach der Schleife fortgesetzt.

1827 Die wesentlichen Befehle sind

1828 • **for (i in X) {Code}**

1829 Wiederhole den Code im “Schleifenrumpf” für jedes Element aus X.

1830 • **while(Bedingung) {Code}**

1831 Wiederhole den Code im “Schleifenrumpf” so lange die logische Bedingung erfüllt ist.

1832 • **break()**

1833 Brich die Schleife ab. **break()** muss im Schleifenrumpf an der richtigen Stelle ausgeführt werden. Gute  
 1834 Praxis ist jedoch, die for oder while Bedingungen, dass kein **break()**nötig ist, da **break()** anfällig für  
 1835 Programmierfehler ist.

#### 1836 12.1.1 Wiederholen von Befehlen mit **for()**.

1837 Steht vor Beginn der Schleife fest wie viele Schleifendurchgänge benötigt werden, wenn zum Beispiel in einer  
 1838 Simulationen 99 Realisierungen erzeugt oder alle Elemente eines Vektors verarbeitet werden sollen, verwendet

1839 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)
}
```

1840 ## [1] 1

1841 ## [1] 2

1842 ## [1] 3

1843 Das **i** steht in diesem Beispiel für die Schleifen-Variable. Sie muss nicht **i** heißen, sondern kann jeden  
1844 zulässigen Namen annehmen. Das **X** steht für einen existierenden Vektor oder eine existierende Liste bzw.  
1845 einen Ausdruck, der ein solches Objekt liefert (der Objektname ist ebenfalls frei wählbar). **for** und **in** sind  
1846 Schlüsselworte, sie müssen, ebenso wie die runden Klammern, vorhanden sein.

1847 Im ersten Durchgang erhält die Schleifen-Variable **i** den ersten Wert von **X** und der Schleifenrumpf wird  
1848 mit diesem Wert ausgeführt. Die Variable **i** nimmt nacheinander so lange die Werte von **X** an, bis ihr alle  
1849 Elemente zugewiesen wurden.

1850 Das folgende Beispiel wird zwar besser durch die entsprechende Vektoroperation gelöst, zeigt aber sehr  
1851 deutlich die Arbeitsweise der **for**-Schleife.

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

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

1852 ## [1] 4

1853 ## [1] 9

1854 ## [1] 25

1855

### 1856 Aufgabe 36: Schleifen 1

---

1858 Verwenden Sie den Vektor **k** <- c(1, 3, 9, 12, 15) und schreiben Sie folgende **for**-Schleifen:

- 1859 1. Eine Schleife, die jedes Element aus **k** ausgibt.
- 1860 2. Eine Schleife, die zu jedem Element aus **k** 10 addiert und den neuen Wert ausgibt.
- 1861 3. Eine Schleife wie in 2), aber der neue Wert (**k** + 10) soll jetzt nicht mehr ausgegeben werden, sondern  
1862 in **k10** gespeichert werden. Stellen Sie sicher, dass **k10** wieder von der Länge 5 ist.

1863

---

1864 Die Funktion **for()** ermöglicht es, einen Befehl beliebig oft zu wiederholen. Z.B. der folgende Ausdruck zieht  
1865 10-Mal eine Stichprobe der Größe 1 aus dem Vektor **v**. Beachten Sie, dass die Schleifen-Variable **i** selbst gar

1866 nicht im Schleifenrumpf vorkommt. Das Ziel dieser Schleife ist nicht die Elemente des Vektors abzuarbeiten,  
 1867 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))
}
```

```
1868 ## [1] 3
1869 ## [1] 1
1870 ## [1] 3
1871 ## [1] 3
1872 ## [1] 2
1873 ## [1] 3
1874 ## [1] 2
1875 ## [1] 2
1876 ## [1] 1
1877 ## [1] 4
```

1878 Auf gleiche Weise kann man auch über die Variablen eines Dataframes iterieren<sup>15</sup>. Das folgende Beispiel hat  
 1879 zum Ziel die Funktionsweise von Schleifen zu verdeutlichen. Schleifen haben in R jedoch den Nachteil, dass sie  
 1880 sehr langsam operieren. Wenn es geht, sollte man Alternativen verwenden. Die Funktionsweise wiederholender  
 1881 Auswertungen wird jedoch mit `for`-Schleifen deutlicher. Aus diesem Grunde werden wir uns in diesem Kurs  
 1882 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.

1891

---

1892 **Aufgabe 37: for-Schleife**

---

18931894 Lesen Sie den Datensatz `bhd_1.txt` ein und verwenden Sie eine `for`-Schleife.

- 1895 • Ziehen Sie 500-Mal je 35 Beobachtungen für den BHD.
- 1896 • Berechnen Sie jeweils den Mittelwert aus den 35 Werten.
- 1897 • Speichern Sie die 500 Mittelwerte in einem neuen Vektor `mittel`.
- 1898 • Wie ist die Verteilung dieser 500 Mittelwerte zu interpretieren?

1899 **12.1.2 Wiederholen von Befehlen mit `while()`**

1900 Die `while`-Schleifen finden Anwendung, wenn die Anzahl der zu durchlaufenden Wiederholungen vorher  
1901 nicht bekannt ist, wie zum Beispiel bei Iterationsverfahren, die bis zu einer gewissen Genauigkeit durchlaufen  
1902 werden sollen. Die `while`-Schleife besteht in R aus dem Schlüsselwort `while()` und einer Bedingung in runden  
1903 Klammern.

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

1904 Sie ist in der praktischen Programmierung nicht so relevant wie die `for`-Schleife. Sie sei deshalb hier nur  
1905 kurz erwähnt. Die Abbruchbedingung wird jedes Mal geprüft bevor der Schleifenrumpf durchlaufen wird. Die  
1906 Bedingung wird ausgewertet und wenn diese `TRUE` ist, wird der Schleifenrumpf ausgeführt und danach erneut  
1907 die Bedingung überprüft. Ist die Bedingung nicht erfüllt, wird der Schleifenrumpf nicht durchgeführt und die  
1908 Schleife beendet. Ist die Bedingung bereits vor Eintritt in die Schleife nicht erfüllt, wird die Schleife gar nicht  
1909 erst durchlaufen.

1910 Da `while`-Schleifen also so lange ausgeführt werden, bis die Bedingung nicht mehr erfüllt ist, kann eine  
1911 Endlosschleife entstehen. Dies kann passieren, wenn man nicht sauber programmiert hat. Wenn innerhalb der  
1912 Schleife nicht dafür gesorgt wird, dass die Bedingung irgendwann nicht mehr erfüllt wird, so läuft die Schleife  
1913 immer weiter. Steckt R in einer Schleife fest und reagiert nicht mehr, kann der Befehl unter Linux mit `Strg`+  
1914 `C` und unter Windows mit `Esc` abgebrochen werden. Alternativ können Sie auf das rote STOP Symbol  
1915 über der Konsole klicken.

1916 **12.2 Bedingte Ausführung von Codeblöcken**

1917 Innerhalb eines Skripts ist es mitunter notwendig je nach aktueller Situation unterschiedlich fortzufahren.  
1918 Die Situation wird mit einem logischen Ausdruck, einer sogenannten Bedingung, geprüft. Je nachdem, ob  
1919 die Bedingung wahr (`TRUE`) oder falsch (`FALSE`) ist, werden unterschiedliche Programmteile ausgeführt, der  
1920 jeweils andere Teil bleibt unberücksichtigt. Danach wird in jedem Fall die Programmausführung, mit den  
1921 auf die bedingte Anweisungen folgenden Anweisung, fortgesetzt. In R kann dies mit dem if-else-Konstrukt  
1922 realisiert werden, welches aus den Schlüsselwörtern `if()` und `else` sowie der in runde Klammern gefassten  
1923 Bedingung besteht.

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

1924 Im folgenden Beispiel sollen die bisher abstrakt beschriebenen Vorgänge praktische Anwendung finden. In  
 1925 dem Beispiel wird zunächst durch zufälliges Ziehen einer Zahl aus dem Bereich eins bis sechs ein Würfelwurf  
 1926 simuliert. Anschließend wird der Würfelwurf mit einem if-Ausdruck kommentiert. Ist die Bedingung, es wurde  
 1927 eine Sechs gewürfelt, erfüllt, wird der Code innerhalb der geschweiften Klammern ausgeführt, ansonsten wird  
 1928 der Klammerinhalt ignoriert.

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

1929 In den meisten Fällen ist es für R irrelevant, ob sich zwischen den verschiedenen Elementen Leerzeichen oder  
 1930 Zeilenumbrüche befinden. Bei dem else-Ausdruck dagegen wird ein Fehler erzeugt, wenn vor dem `else` nicht  
 1931 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.")
}
```

1932 `## [1] "Beim nächsten Wurf klappt's bestimmt."`

1933

### 1934 Aufgabe 38: Bedingte Programmierung

---

- 1936 • Wenn keine 6 gewürfelt wurde, lassen Sie zusätzlich ausgeben welche Zahl gewürfelt wurde.  
 1937 • Wiederholen Sie den Würfelwurf 10 Mal.

## 1938 13 (R)markdown

### 1939 13.1 Markdown Grundlagen

1940 Die Idee von Markdown ist, einfach Text strukturieren zu können und das ganze ohne umfangreiche Programme  
 1941 zu erstellen. Im nächsten Abschnitt sehen wir dann, wie man Markdown und R-Code verbinden kann. Hier  
 1942 soll es jedoch erst einmal darum gehen, die einfachsten Bausteine von Markdown vorzustellen.

1943 Am Anfang jedes Dokuments kommt eine Präambel. Diese fängt mit --- an und hört auch wieder mit ---  
 1944 auf. Innerhalb der Präambel können dann Metainformationen über das Dokument festgelegt werden, dies  
 1945 beinhaltet im einfachsten Fall: Titel, Autor und Datum. Das würde dann so aussehen:

```
1946 ---
1947 title: "Ein Titel"
1948 author: "Der, der es geschrieben hat"
1949 date: "März 2021"
1950 ---
```

1951 Danach folgt strukturierter und formatierter Text. Verschiedene Hierarchieebenen von Überschriften können  
 1952 mit der Anzahl an # festgelegt werden. So ist eine Überschrift erster Ordnung # Kapitel eine Überschrift  
 1953 zweiter Ordnung ## Unterkapitel usw.

1954 Listen können erstellt werden, wenn man am Anfang jeder Zeile ein - oder 1. schreibt.

```
1955 - Erster Eintrag
1956 - Zweiter Eintrag
1957 - Dritter Eintrag
```

1958 wird zu

```
1959 • Erster Eintrag
1960 • Zweiter Eintrag
1961 • Dritter Eintrag
```

1962 Eine zentrale Idee von Markdown ist es Text einfach zu formatieren. Werden eine oder mehrere Wörter mit  
 1963 zwei Sternchen (\*\*) eingefasst wird dieser Text **fett** dargestellt. Also aus \*\*wichtig\*\* wird **wichtig**. Das  
 1964 gleiche funktioniert auch mit *kursiven* Text, jedoch muss man hier noch ein Sternchen verwenden, also aus  
 1965 \*kursiv\* wird *kursiv*. Soll ein text fett und kursiv sein, kann man drei Sternchen verwenden. Aus \*\*\*sehr  
 1966 wichtig\*\*\* wird dann **sehr wichtig**.

1967 Weitere Elemente wie Links oder Abbildungen können einfach eingebunden werden. Links werden mit [Link  
 1968 text](url) in den Text integriert. Beispielsweise ist eine gute Idee bei [stackoverflow](#) bei Problemen nach  
 1969 einer Lösung zu suchen. Dieser Link wurde mit [stackoverflow](www.stackoverflow.com) erstellt.

1970 Für Abbildungen gibt es einen ganz ähnlichen Syntax. Mit ! [Das R Logo](abb/r\_logo.png) wird die  
 1971 Abbildung r\_logo.png eingebunden mit der Beschriftung: "Das R Logo".



Abbildung 12: Das R Logo

1972

---

**Aufgabe 39: Arbeiten mit markdown**


---

1975 Verwenden Sie das folgende Markdowndokument:

```

1976 ---
1977 title: "Dokument"
1978 author: "Ihr Name"
1979 date: "März 2021"
1980 ---
1981
1982 # Einleitung
1983
1984 # Methoden
1985 1. Kopieren Sie die Vorlage in ein Dokument, das test.md heißt.
1986 2. Fügen Sie zwei Überschriften zweiter und dritter Ordnung hinzu.
1987 3. Fügen Sie einen kursiven Text hinzu.
1988 4. Fügen Sie einen Link zu einer Website hinzu.
1989 5. Kompilieren Sie die Datei, indem Sie in Rstudio auf Preview drücken (Abbildung 13).

```

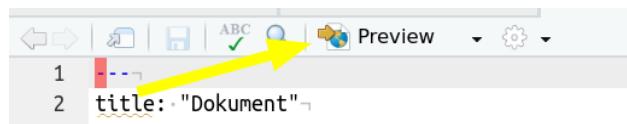


Abbildung 13: Kompilieren einer md-Datei.

**13.2 R und Markdown**

1991 Markdown macht es bereits einfach Textdokumente und Dokumentationen zu verfassen, aber die wirkliche  
 1992 Stärke liegt in der Möglichkeit R und Markdown zu kombinieren. Man spricht dann von Rmarkdown. Ein  
 1993 weiteres Strukturelement, das wir noch nicht kennen gelernt haben, sind Code-Blöcke.

1994 ~~~

1995 a &lt;- 1:10

```

1996 a[1]
1997 ``
1998 erzeugt
1999 a <- 1:10
2000 a[1]

2001 Momentan wird noch kein Code ausgeführt, sondern lediglich der Code, als Code dargestellt. Rmarkdown
2002 bietet nun die Möglichkeit Code beim kompilieren16 auszuführen. Dafür müssen wir nur einen Code-Block als
2003 R-Code-Block kennzeichnen.

2004 ``{R}
2005 a <- 1:10
2006 a[1]
2007 ```

2008 erzeugt
2009 a <- 1:10
2010 a[1]

2009 ## [1] 1

2010 Beachte, die Variable a wird beim kompilieren erzeugt und steht dann R zur Verfügung. R-Code-Blöcke
2011 werden auch als Code Chunks bezeichnet. Diese Chunks können sehr genau angesprochen und angepasst
2012 werden. Einige wichtige Argumente sind:
2013 • echo: Gibt an, ob der Quelltext angezeigt werden soll oder nicht.
2014 • result: Gibt an, ob die Ergebnisse gezeigt werden sollen oder nicht.
2015 • eval: Diese Option gibt an ob der Chunk ausgeführt werden soll oder nicht.

```

2016

---

**Aufgabe 40: Arbeiten mit Rmarkdown**


---

2019 Erstellen Sie eine neue Rmarkdown Datei mit dem Namen `test1.Rmd`. Erstellen Sie zwei Code-Chunks. Der  
2020 erste soll nicht angezeigt werden und darin werden die Daten geladen (`bhd_1.txt`). Im zweiten Chunk plotten  
2021 Sie das Alter der Bäume gegen den BHD. Was passiert mit dem Plot, wenn Sie die Datei kompilieren (drücken  
2022 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.

## 2023 14 Räumliche Daten in R

### 2024 14.1 Was sind räumliche Daten

2025 Räumliche Daten sind Beobachtungen, wie wir sie schon oft gesehen haben, mit einem räumlichen Bezug. Der  
 2026 Unterschied zu nicht räumlichen Daten liegt darin, dass räumliche Daten eindeutig im *Raum* verortet werden  
 2027 können. Häufig werden sogenannte Geoinformationssysteme (GIS) zum Arbeiten mit räumlichen verwendet.  
 2028 R kann in vielerlei Hinsicht wie ein GIS eingesetzt werden und hier werden einige Grundfunktionalitäten  
 2029 dafür besprochen. Räumliche Daten werden in zwei unterschiedliche Datentypen unterteilt: Vektor- und  
 2030 Rasterdaten. Vektordaten modellieren einzelne Objekte (= *Features*). Rasterdaten modellieren eine Oberfläche.  
 2031 Vektordaten bestehen aus zwei Komponenten: 1) einer Geometrie, die die Form und Lage der Daten definiert  
 2032 und 2) Attributen, den tatsächlichen Daten. Räumliche Daten werden oft als *Features* bezeichnet. Ein Feature  
 2033 ist die räumliche Einheit einer Beobachtung. Je nach Art der räumlichen Daten können Features entweder  
 2034 Punkte (z.B. ein Baum), Linien (z.B. eine Straße) oder Polygone (z.B. ein See) sein. Auch können mehrere  
 2035 Geometrien zu einem Feature zusammengefasst werden. Ein Beispiel wäre eine Beobachtung für ein Land,  
 2036 das aber aus mehreren Polygonen bestehen kann (z.B. Festland und Inseln). Features können dann weitere  
 2037 Attribute (= Attributdaten) haben, z.B. eine ID, Name oder was auch immer man gemessen hat.  
 2038 Rasterdaten bestehen aus einer Oberfläche von gleichgroßen Kacheln (= *Pixel*), die ein Gebiet abdecken.  
 2039 Meist sind Pixel viereckig, aber das ist keine Voraussetzung. Dabei hat jedes Pixel einen Wert (das kann  
 2040 auch ein fehlender Wert sein). Typische Beispiele für Rasterdaten sind Landnutzung oder Seehöhen.  
 2041 In R kann sowohl mit Vektor- als auch mit Rasterdaten gearbeitet werden. Für Vektordaten bietet sich das  
 2042 Paket **sf** an und für Rasterdaten das Paket **raster**.

### 2043 14.2 Koordinatenbezugssystem

2044 Eine der Herausforderungen für räumliche Daten ist die eindeutige Verortung im Raum. Dazu braucht man  
 2045 ein *Koordinatenbezugssystem (KBS)*. Einem KBS liegt ein mathematisches Modell der Erde zugrunde. Die  
 2046 Details zu KBS werden schnell relativ kompliziert und wir beschränken uns hier lediglich darauf, wie KBS  
 2047 verwendet werden können. Dazu müssen wir zwei Fälle unterscheiden: 1) einem Datensatz ein KBS zuweisen  
 2048 und 2) Transformation des KBS eines Datensatzes in ein anderes KBS. Die technischen Details werden in  
 2049 den folgend Abschnitten besprochen. Für beide Aufgaben müssen wir auf ein KBS verweisen können, ein  
 2050 einfacher Ansatz dafür sind die sogenannten *EPSG-Codes*<sup>17</sup>.

### 2051 14.3 Vektordaten in R

2052 Das Paket **sf** stellt Klassen zum Abbilden von Features zur Verfügung, die dann in einem **data.frame** als  
 2053 Liste gespeichert werden können. In der Regel erstellen wir Features nicht individuell, sondern lesen diese aus  
 2054 externen Datenquellen (z.B. ESRI Shapefile) ein. Zum besseren Verständnis, erstellen wir es einmal manuell.  
 2055 Wir haben die Koordinaten für drei Städte (Göttingen, Hannover und Berlin) als geografische Koordinaten  
 2056 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))
```

2057 Daraus können wir jetzt eine Geometriespalte für einen `data.frame` erstellen

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

2058 Somit haben wir die Geometrie in der Variable `geom` gespeichert, aber noch keine dazugehörigen Attributdaten.

2059 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)
)
```

2060 In einem letzten Schritt möchten wir jetzt die Geometrie (`geom`) und die Attributdaten (`attr`) zusammenführen.

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

```
2062 ## Simple feature collection with 3 features and 3 fields
2063 ## Geometry type: POINT
2064 ## Dimension: XY
2065 ## Bounding box: xmin: 9.732 ymin: 51.5413 xmax: 13.405 ymax: 52.52
2066 ## Geodetic CRS: WGS 84
2067 ## name bundesland einwohner geom
2068 ## 1 Goettingen Niedersachsen 119000 POINT (9.9158 51.5413)
2069 ## 2 Hannover Niedersachsen 532000 POINT (9.732 52.3759)
2070 ## 3 Berlin Berlin 3650000 POINT (13.405 52.52)
```

2071 Wir können nun mit `staedte` genau so arbeiten wie mit jedem anderen `data.frame` und die Geometrien

2072 werden immer ‘berücksichtigt’. Zusätzlich kann man eine Reihe von geometrischen Operationen durchführen.

2073 Wenn ein `data.frame` Punkten hat, kann man dies relative einfach mit der Funktion `st_as_sf()` “räumlich”

2074 machen. Für das vorherige Beispiel würden wir zuerst einen `data.frame` mit allen Informationen (zur

2075 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)
)
```

2076 Dann kann man mit der Funktion `st_as_sf()` weiter arbeiten:

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

## 2077 14.4 Arbeiten mit Vektordaten

2078 Es gibt sehr viele Funktionen, um mit räumlichen Daten zu arbeiten, von denen wir hier einige vorstellen.

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

```
2079 ## Coordinate Reference System:
2080 ## User input: EPSG:4326
2081 ## wkt:
2082 ## GEOGCRS["WGS 84",
2083 ## ENSEMBLE["World Geodetic System 1984 ensemble",
2084 ## MEMBER["World Geodetic System 1984 (Transit)"],
2085 ## MEMBER["World Geodetic System 1984 (G730)"],
2086 ## MEMBER["World Geodetic System 1984 (G873)"],
2087 ## MEMBER["World Geodetic System 1984 (G1150)"],
2088 ## MEMBER["World Geodetic System 1984 (G1674)"],
2089 ## MEMBER["World Geodetic System 1984 (G1762)"],
2090 ## MEMBER["World Geodetic System 1984 (G2139)"],
2091 ## ELLIPSOID["WGS 84",6378137,298.257223563,
2092 ## LENGTHUNIT["metre",1]],
2093 ## ENSEMBLEACCURACY[2.0]],
2094 ## PRIMEM["Greenwich",0,
2095 ## ANGLEUNIT["degree",0.0174532925199433]],
2096 ## CS[ellipsoidal,2],
2097 ## AXIS["geodetic latitude (Lat)",north,
2098 ## ORDER[1],
2099 ## ANGLEUNIT["degree",0.0174532925199433]],
2100 ## AXIS["geodetic longitude (Lon)",east,
2101 ## ORDER[2],
2102 ## ANGLEUNIT["degree",0.0174532925199433]],
2103 ## USAGE[
2104 ## SCOPE["Horizontal component of 3D system."],
2105 ## AREA["World."],
2106 ## BBOX[-90,-180,90,180]],
2107 ## ID["EPSG",4326]]
```

2108 Wenn wir jetzt zu einem anderen KBS (z.B. EPSG:3035, ein europäisches projiziertes KBS) umrechnen  
 2109 möchten, können wir das mit

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

```

2110 ## Coordinate Reference System:
2111 ## User input: EPSG:3035
2112 ## wkt:
2113 ## PROJCRS["ETRS89-extended / LAEA Europe",
2114 ## BASEGEOGCRS["ETRS89",
2115 ## ENSEMBLE["European Terrestrial Reference System 1989 ensemble",
2116 ## MEMBER["European Terrestrial Reference Frame 1989"],
2117 ## MEMBER["European Terrestrial Reference Frame 1990"],
2118 ## MEMBER["European Terrestrial Reference Frame 1991"],
2119 ## MEMBER["European Terrestrial Reference Frame 1992"],
2120 ## MEMBER["European Terrestrial Reference Frame 1993"],
2121 ## MEMBER["European Terrestrial Reference Frame 1994"],
2122 ## MEMBER["European Terrestrial Reference Frame 1996"],
2123 ## MEMBER["European Terrestrial Reference Frame 1997"],
2124 ## MEMBER["European Terrestrial Reference Frame 2000"],
2125 ## MEMBER["European Terrestrial Reference Frame 2005"],
2126 ## MEMBER["European Terrestrial Reference Frame 2014"],
2127 ## ELLIPSOID["GRS 1980",6378137,298.257222101,
2128 ## LENGTHUNIT["metre",1]],
2129 ## ENSEMBLEACCURACY[0.1]],
2130 ## PRIMEM["Greenwich",0,
2131 ## ANGLEUNIT["degree",0.0174532925199433]],
2132 ## ID["EPSG",4258],
2133 ## CONVERSION["Europe Equal Area 2001",
2134 ## METHOD["Lambert Azimuthal Equal Area",
2135 ## ID["EPSG",9820]],
2136 ## PARAMETER["Latitude of natural origin",52,
2137 ## ANGLEUNIT["degree",0.0174532925199433],
2138 ## ID["EPSG",8801]],
2139 ## PARAMETER["Longitude of natural origin",10,
2140 ## ANGLEUNIT["degree",0.0174532925199433],
2141 ## ID["EPSG",8802]],
2142 ## PARAMETER["False easting",4321000,
2143 ## LENGTHUNIT["metre",1],
2144 ## ID["EPSG",8806]],
2145 ## PARAMETER["False northing",3210000,
2146 ## LENGTHUNIT["metre",1],
2147 ## ID["EPSG",8807]]],
2148 ## CS[Cartesian,2],
2149 ## AXIS["northing (Y)",north,
2150 ## ORDER[1],
2151 ## LENGTHUNIT["metre",1]],
2152 ## AXIS["easting (X)",east,

```

```

2153 ## ORDER[2] ,
2154 ## LENGTHUNIT["metre",1]],
2155 ## USAGE [
2156 ## SCOPE["Statistical analysis."],
2157 ## AREA["Europe - European Union (EU) countries and candidates. Europe - onshore and offshore: "],
2158 ## BBOX[24.6,-35.58,84.73,44.83]],
2159 ## ID["EPSG",3035]

```

2160 Die Funktion `st_buffer()` erlaubt es Features zu puffern, mit `st_distance()` kann die Distanz zwischen  
2161 Features berechnet werden, mit `st_area()` kann die Fläche eines Features berechnet werden.

2162 Funktionen wie `st_intersection()`, `st_union()` und `st_difference()` erlauben es geometrische Opera-  
2163 tionen zwischen unterschiedlichen Features zu berechnen. Für eine ausführliche Diskussion siehe auch hier:  
2164 <https://geocompr.robinlovelace.net/geometric-operations.html>.

2165 Normalerweise lesen wir Daten von externen Datenquellen (z.B. ESRI Shapefiles). Das geht mit der Funktion  
2166 `st_read()`.

## 2167 14.5 Rasterdaten in R

2168 Für Rasterdaten gibt es das R-Paket `terra`. Auch hier wollen wir uns wieder auf einige Grundfunktionalitäten  
2169 konzentrieren. Diese umfassen das Einlesen, Zuschneiden, Rechnen und Abfragen von Rastern.

2170 Mit der Funktion `rast()` kann ein Raster in R eingelesen werden.

```

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

```

2171 `dem` steht für *Digital Elevation Model* und ist ein Raster mit den Seehöhen in Niedersachsen mit einer  
2172 500-m-Auflösung. Wir können diese mit der Funktion `res()`<sup>18</sup> abfragen.

```

res(dem)

```

2173 ## [1] 500 500

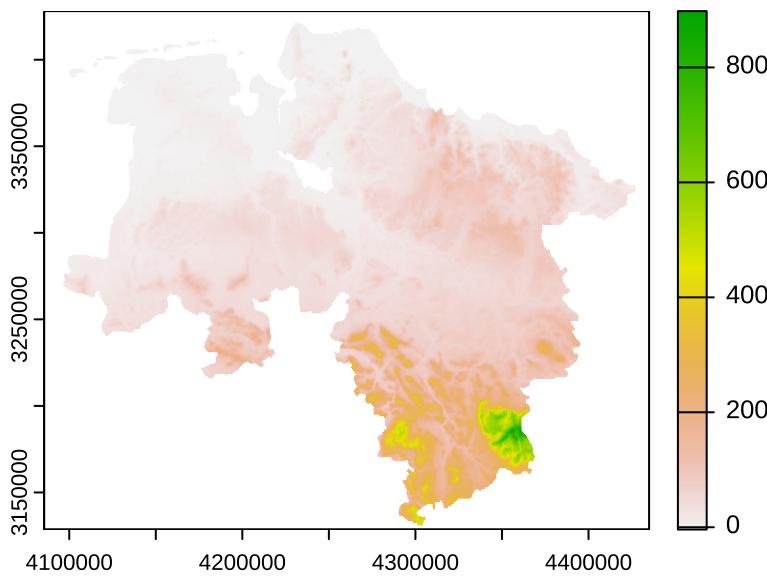
2174 Bzw. wir können den Raster auch plotten.

```

plot(dem)

```

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



2176 Wenn wir den Raster `dem` auf ein Gebiet zuschneiden wollen (z.B. Göttingen), müssen wir drei Schritte  
 2177 durchführen. Als erstes müssen wir ein Shapefile für Göttingen einlesen.

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

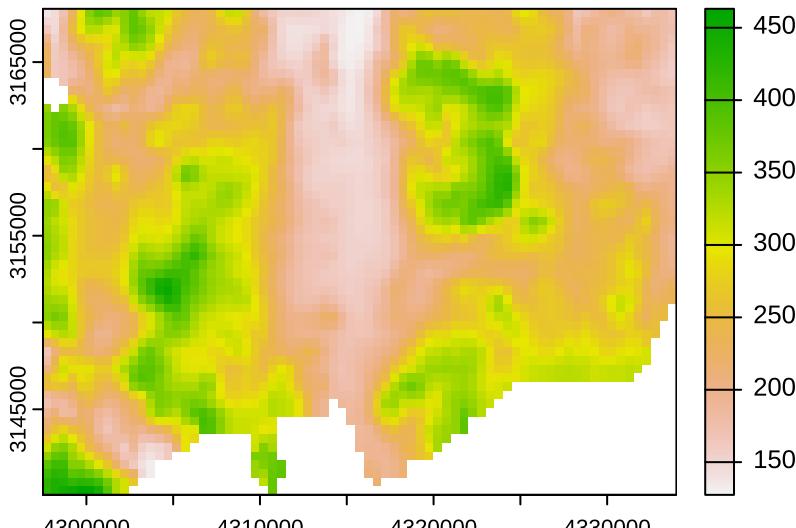
2178 Dann müssen wir sicherstellen, dass sowohl der Raster `dem` als auch das `sf`-Objekt `goe` im selben KBS sind.  
 2179 Es bietet es sich in der Regel an, das KBS des Vektors zu transformieren. Mit der Funktion `projectRaster()`  
 2180 kann das KBS eines Raster transformiert werden.

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

2181 Mit der Funktion `crop()` kann der Raster `dem` auf Göttingen zugeschnitten werden.

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

```
plot(dem1)
```



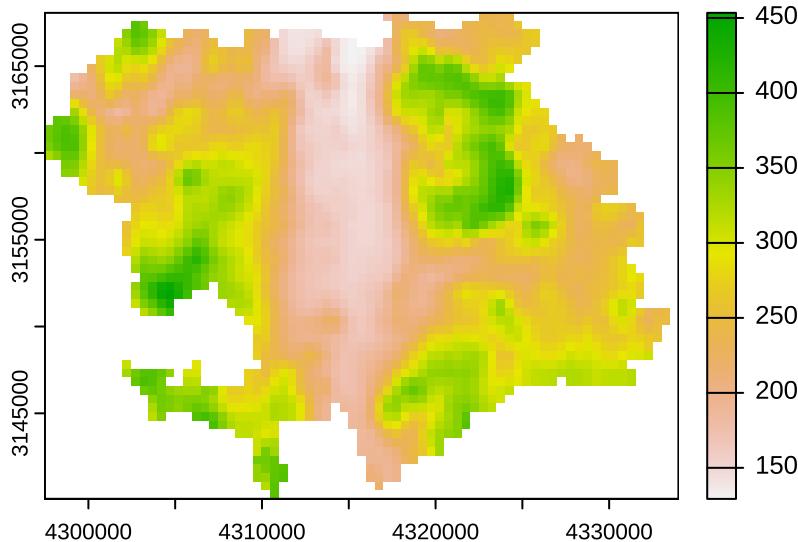
2182 Der Raster hat jetzt die Größe einer Bounding-Box (BBX) von Göttingen (das ist ein Rechteck, das Göttingen  
 2183 umfasst). Mit der Funktion `mask()` kann der Raster auf die genauen Grenzen des Vektors `goe` angepasst

2185 werden.

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

2186 ## Warning: [mask] CRS do not match

```
plot(dem2)
```



2187

2188 Wenn wir an bestimmten Punkten den Wert des Rasters abfragen wollen (z.B. an `cities`) von vorhin, dann  
2189 gibt es dafür die Funktion `extract`. Dann müssen wir erst sicherstellen, dass `staedte` und `dem` gleichen KBS  
2190 zu grunde liegt. Dafür transformieren wir einfach `staedte` in das KBS von `dem`. Mit der Funktion `crs()`  
2191 erhalten wir das KBS des Rasters.

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

2192 Wenn wir das KBS eines Objektes nicht kennen, können wir auch einfach das KBS übergeben. Der folgende  
2193 Code-Block macht genau das Gleiche mit dem Vorteil, dass wir keinen EPSG-Code angeben müssen.

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

2194 Dann können wir für jede Stadt die Seehöhe abfragen:

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

2195 ## ID dem\_3035

2196 ## 1 1 149.18181

2197 ## 2 2 57.21486

2198 ## 3 3 NA

2199 Mit `terra::extract()` rufen wir *eindeutig* die Funktion `extract()` aus dem Paket `terra` auf. Wir müssen  
2200 das so machen, weil es im Paket `dplyr` auch eine Funktion `extract()` gibt, die wir hier nicht anwenden  
2201 möchten, da sie einen Fehler verursachen würde.

2202 Ein analoges Vorgehen ist auch für Linien und Polygone möglich.

2203 Mit Rastern kann auch einfach gerechnet werden. Wir können z.B. die Seehöhe in Kilometern anstatt Metern

2204 berechnen:

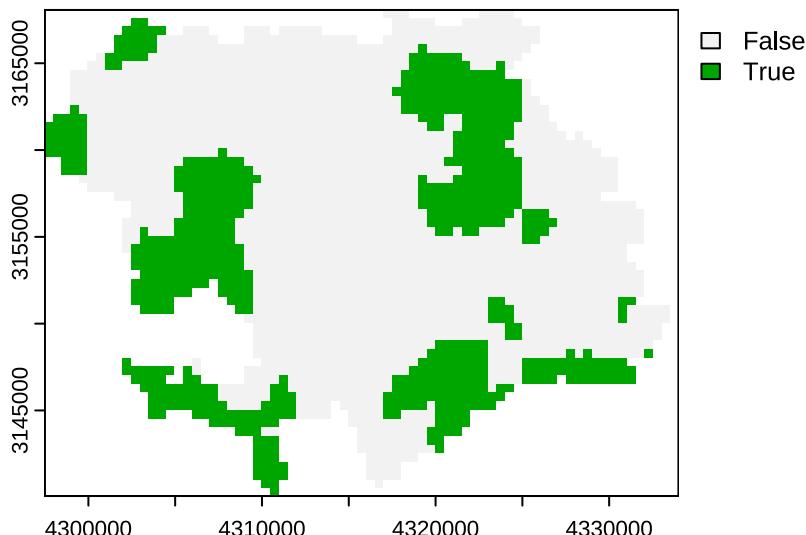
```
dem_km <- dem / 1e3
```

2205 Auch logische Operationen sind möglich, wenn wir alle Rasterzellen mit einer Seehöhe von mehr als 300 m in

2206 Göttingen suchen, dann geht das so:

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

```
plot(dem3)
```



2207

2208 Wenn wir jetzt auf die Werte des Rasters `dem3` zugreifen wollen, geht das mit eckigen Klammern.

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

2209 ## dem\_3035

2210 ## [1,] NA

2211 ## [2,] NA

2212 ## [3,] NA

2213 ## [4,] NA

2214 ## [5,] NA

2215 ## [6,] NA

2216 Das sind erst einmal viele NA-Werte für die ganzen Zellen, die außerhalb von Niedersachsen liegen. Aber wir  
2217 können mit so einem Vektor ganz normal arbeiten und z.B. die Fläche des Landes Niedersachsen die eine  
2218 Seehöhe von mehr als 500m Seehöhe hat ausrechnen.

```
h <- dem3[]
```

```
sum(h, na.rm = TRUE) / sum(!is.na(h))
```

2219 ## [1] 0.2786229

2220

---

**Aufgabe 41: Arbeiten mit Rastern**

---

2223 Verwenden Sie den Raster `wald.tif`, der auf einer 10 m Auflösung den Waldanteil jeder Rasterzelle angibt<sup>19</sup>.  
 2224 Der EPSG-Code für das KBS von `wald.tif` ist 3035. Nehmen Sie an, dass wenn der Waldanteil in einer  
 2225 Raster größer als 50 % ist, dass die Rasterzelle als Wald klassifiziert werden kann. Wie viel Prozent des  
 2226 Göttinger Stadtgebietes sind Wald? Wie ändert sich dieser Wert, wenn sie 70 % anstatt 50 % als Schwellenwert  
 2227 für Wald annehmen?

2228

---

**Aufgabe 42: Studiendesign**

---

2231 Mit der Funktion `st_sample()` können Sie innerhalb oder entlang eines Features zufällige Punkte legen. Das  
 2232 Argument `n` steuert die Anzahl Punkte und das Argument `type` wie die Punkte angeordnet werden. Für `type`  
 2233 sind für uns die Werte `type = "random"` (komplett zufällig), `type = "regular"` (regelmäßiger Grid) und  
 2234 `type = "hexagonal"` von Bedeutung (ein hexagonaler Grid, d.h. ein sechseckiger Raster). Unglücklicherweise  
 2235 ist das Ergebnis von `st_sample()` erst eine Geometrie. Um daraus ein vollständiges `sf`-Objekt zu machen  
 2236 und problemlos weiter arbeiten zu können, müssen Sie noch einmal die Funktion `st_as_sf()` ausführen.  
 2237 Stellen Sie sich vor, dass wir die tatsächliche Waldbedeckung des Göttinger Stadtgebietes **nicht** kennen und wir  
 2238 eine Studie durchführen, um den Anteil des Göttinger Stadtgebietes, der mit Wald bedeckt ist herauszufinden.  
 2239 Erstellen Sie dafür einige unterschiedliche Stichproben (diese können in der Anzahl und Anordnung variieren).  
 2240 Berechnen Sie für jedes Stichprobendesign den Anteil an Wald und ein dazugehöriges Konfidenzintervall  
 2241 (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  $n$   
 2242 die Stichprobengröße). Nehmen Sie an, dass eine Rasterzelle Wald ist, sobald > 50 % der Rasterzelle mit  
 2243 Wald bedeckt ist.

2244

---

**Aufgabe 43: Räumliche Daten**

---

2247 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)
)
```

2248 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

- 2249 2. Puffern Sie jeden Baum mit seinem Kronendurchmesser.
- 2250 3. Berechnen Sie die Kronenfläche jedes Baumes. *Hinweis: Die Funktion `st_area()` könnte dafür hilfreich sein.*
- 2251 4. Welcher Baum hat die größte Kronenfläche?
- 2252 5. Finden Sie für jede Art, den Baum mit der größten Kronenfläche.

2254

2255 **Aufgabe 44: Arbeiten mit räumlichen Daten**

---

- 2257 1. Lesen Sie das ESRI Shapefile `goettingen/stadt_goettingen.shp` ein.
- 2258 2. Wie viele Features befinden sich in dem Shapefile?
- 2259 3. Welches Koordinatenbezugssystem (KBS) hat das Shapefile?
- 2260 4. Transformieren Sie das Shapefile in das KBS 3035.
- 2261 5. Erstellen Sie eine neue Spalte `A` in der Sie die Fläche jeder Gemeinde/Stadt speichern.
- 2262 6. Welche Gemeinde/Stadt (Spalte `GEN`) ist am größten?
- 2263 7. Wählen Sie nun nur die Stadt Göttingen aus.

2264

2265 **Aufgabe 45: Arbeiten mit räumlichen Daten 2**

---

- 2267 1. Lesen Sie erneut das ESRI Shapefile `goettingen/stadt_goettingen.shp` ein.
- 2268 2. Lösen sie die Gemeindegrenzen auf (die Funktion `st_union()` könnte hier nützlich sein).
- 2269 3. Wie groß ist das resultierende Feature?

2270 **15 FAQs (Oft gefragtes)**

2271 **15.1 Arbeiten mit Daten**

2272 **15.1.1 Einlesen von Exceldateien**

2273 Mit der Funktion `read_excel()` aus dem Paket `readxl` können Exceldateien direkt in R eingelesen werden.

2274 Ein Export als csv-Datei aus Excel ist nicht notwendig.

## 2275 16 Literatur

- 2276 Ein guter Überblick über viele der angeschnittenen Themen gibt es in dem [R for data science](#), das online frei  
2277 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.  
2278
- 2279 McNamara, Amelia, and Nicholas J Horton. 2018. “Wrangling Categorical Data in r.” *The American Statistician*  
2280 72 (1): 97–104.
- 2281 Wickham, Hadley. 2014. “Tidy Data.” *Journal of Statistical Software* 59 (10). <https://doi.org/10.18637/jss.v059.i10>.  
2282