

1

---

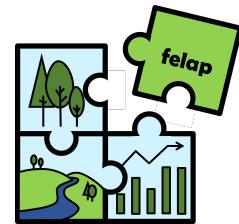
# Einführung in die Datenanalyse mit R (700104)

4

Kursskript

5

---



6

Dr. Johannes SIGNER  
Abteilung Wildtierwissenschaften  
Büsgenweg 3  
37077 Göttingen

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

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

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

7



8

Fakultät für Forstwissenschaften und Waldökologie  
Georg-August-Universität Göttingen

10



11

---

12

Wintersemester 2023/2024

---

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

<sup>15</sup> Zitiervorschlag:

<sup>16</sup> Signer, J. und Husmann, K. (2023) Skript zur Vorlesung Einführung in die Datenanalyse mit R, Georg-August-Universität Göttingen.

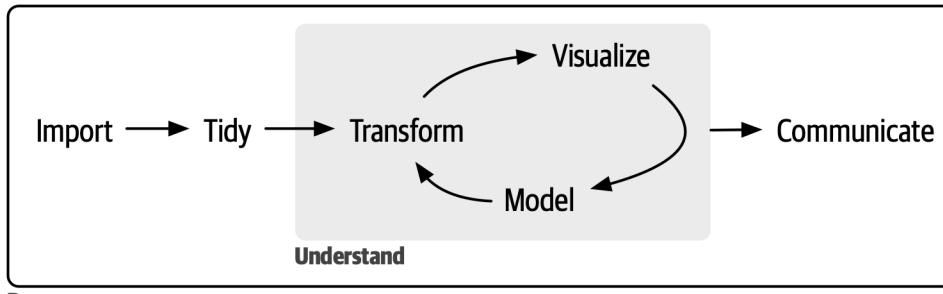
<sup>18</sup> Letzte Aktualisierung: 1. Dezember 2023

---

## 19 Vorwort und Danksagung

20

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



25 **Program**

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

## 44 Inhaltsverzeichnis

|    |   |    |
|----|---|----|
| 45 | <b>1 R und RStudio</b>  | 4  |
| 46 | 1.1 Installation von R und RStudio . . . . .                          | 4  |
| 47 | 1.2 Erste Schritte in R . . . . .                                     | 4  |
| 48 | 1.3 Gute Praxis bei der Programmierung . . . . .                      | 6  |
| 49 | <b>2 Variablen, Funktionen und Datentypen</b>                         | 8  |
| 50 | 2.1 Variablen beim Programmieren . . . . .                            | 8  |
| 51 | 2.2 Datentypen . . . . .  | 9  |
| 52 | 2.3 Funktionen . . . . .  | 10 |
| 53 | 2.4 Datenstrukturen . . . . .   | 11 |
| 54 | 2.5 Funktionen . . . . .  | 12 |
| 55 | <b>3 Vektoren</b>   | 14 |
| 56 | 3.1 Funktionen zum Arbeiten mit Vektoren . . . . .                    | 16 |
| 57 | 3.2 Statistische Funktionen . . . . .                                 | 17 |
| 58 | 3.3 Beispiel Fotofallen . . . . .                                     | 18 |
| 59 | 3.4 Arbeiten mit logischen Werten . . . . .                           | 19 |
| 60 | 3.5 Zugreifen auf Elemente eines Vektors (=Untermengen) . . . . .     | 21 |
| 61 | 3.6 Der %in%-Operator . . . . .                                       | 22 |
| 62 | <b>4 Faktoren (factors)</b>   | 24 |
| 63 | 4.1 Das Paket <b>forcats</b> . . . . .                                | 26 |
| 64 | 4.1.1 Anpassen der Anordnung von Faktoren . . . . .                   | 26 |
| 65 | <b>5 Spezielle Einträge</b>   | 27 |
| 66 | 5.1 NA . . . . .  | 27 |
| 67 | 5.2 NULL . . . . .  | 28 |
| 68 | 5.3 Inf . . . . .   | 28 |
| 69 | <b>6 data.frames oder Tabellen</b>                                    | 30 |
| 70 | 6.1 Wichtige Funktionen zum Arbeiten mit <b>data.frames</b> . . . . . | 31 |
| 71 | 6.2 Zugreifen auf Elemente eines <b>data.frame</b> . . . . .          | 32 |
| 72 | <b>7 Schreiben und lesen von Daten</b>                                | 35 |
| 73 | 7.1 Textdateien . . . . .   | 35 |
| 74 | <b>8 Erstellen von Abbildungen</b>                                    | 37 |
| 75 | 8.1 Base Plot . . . . .   | 37 |
| 76 | 8.1.1 Mehrere Panels . . . . .  | 43 |
| 77 | 8.1.2 Speichern von Abbildungen . . . . .                             | 43 |
| 78 | 8.2 Histogramme . . . . .   | 44 |
| 79 | 8.3 Boxplots . . . . .  | 47 |
| 80 | 8.4 <b>ggplot2</b> : Eine Alternative für Abbildungen . . . . .       | 49 |

|     |   |            |
|-----|---|------------|
| 81  | 8.4.1 Multipanel Abbildungen . . . . .                  | 56         |
| 82  | 8.4.2 Plots kombinieren . . . . .                       | 58         |
| 83  | 8.4.3 Speichern von plots . . . . .                     | 61         |
| 84  | <b>9 Mit Daten arbeiten</b>                             | <b>62</b>  |
| 85  | 9.1 dplyr eine Einführung . . . . .                     | 62         |
| 86  | 9.2 Arbeiten mit gruppierten Daten . . . . .            | 65         |
| 87  | 9.3 pipes oder %>% . . . . .                            | 66         |
| 88  | 9.4 Joins . . . . .                                     | 67         |
| 89  | 9.5 ‘long’ and ‘wide’ Datenformate . . . . .            | 69         |
| 90  | 9.6 Auswählen von Variablen . . . . .                   | 71         |
| 91  | 9.7 Einzelne Beobachtungen abfragen (slice()) . . . . . | 72         |
| 92  | 9.8 Spalten trennen . . . . .                           | 75         |
| 93  | <b>10 Arbeiten mit Text</b>                             | <b>77</b>  |
| 94  | 10.1 Arbeiten mit Text . . . . .                        | 77         |
| 95  | 10.2 Finden von Textmustern . . . . .                   | 78         |
| 96  | <b>11 Arbeiten mit Zeit</b>                             | <b>81</b>  |
| 97  | 11.1 Arbeiten mit Zeitintervallen . . . . .             | 82         |
| 98  | 11.2 Formatieren von Zeit . . . . .                     | 83         |
| 99  | <b>12 Aufgaben Wiederholen (for-Schleifen)</b>          | <b>85</b>  |
| 100 | 12.1 Schleifen . . . . .                                | 85         |
| 101 | 12.1.1 Wiederholen von Befehlen mit for() . . . . .     | 85         |
| 102 | 12.1.2 Wiederholen von Befehlen mit while() . . . . .   | 88         |
| 103 | 12.2 Bedingte Ausführung von Codeblöcken . . . . .      | 88         |
| 104 | <b>13 (R)markdown</b>                                   | <b>90</b>  |
| 105 | 13.1 Markdown Grundlagen . . . . .                      | 90         |
| 106 | 13.2 R und Markdown . . . . .                           | 91         |
| 107 | <b>14 Räumliche Daten in R</b>                          | <b>93</b>  |
| 108 | 14.1 Was sind räumliche Daten . . . . .                 | 93         |
| 109 | 14.2 Koordinatenbezugssystem . . . . .                  | 93         |
| 110 | 14.3 Vektordaten in R . . . . .                         | 93         |
| 111 | 14.4 Arbeiten mit Vektordaten . . . . .                 | 95         |
| 112 | 14.5 Rasterdaten in R . . . . .                         | 97         |
| 113 | <b>15 FAQs (Oft gefragtes)</b>                          | <b>102</b> |
| 114 | 15.1 Arbeiten mit Daten . . . . .                       | 102        |
| 115 | 15.1.1 Einlesen von Exceldateien . . . . .              | 102        |
| 116 | <b>16 Zusätzliche Aufgaben</b>                          | <b>103</b> |
| 117 | 16.1 Arbeiten mit Daten . . . . .                       | 105        |

118 **17 Literatur**

**107**

# 1 R und RStudio

## 1.1 Installation von R und RStudio

- Als ersten Schritt müssen Sie R und RStudio installieren. Dabei ist wichtig zu unterscheiden, dass R und RStudio zwei unterschiedliche Programme sind. R ist die eigentliche Programmiersprache mit der wir arbeiten. RStudio hingegen ist eine sogenannte Entwicklungsumgebung<sup>1</sup>, die das Arbeiten mit R vereinfachen soll.
- Sie können also mit R arbeiten ohne RStudio (was unüblich ist), aber nicht mit RStudio ohne R.
- Gehen Sie für die Installation von R, auf die Website <https://cloud.r-project.org/> und laden Sie die für ihren Computer passende R-Version herunter und folgen Sie den Installationsanweisungen. In Linux können Sie R über die Kommandozeile installieren.
- Für die Installation von RStudio gehen Sie zu der Website <https://posit.co/download/rstudio-desktop/#download> und laden die richtige Version für Ihr Betriebssystem herunter und folgen Sie den Installationsanweisungen.

## 1.2 Erste Schritte in R

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

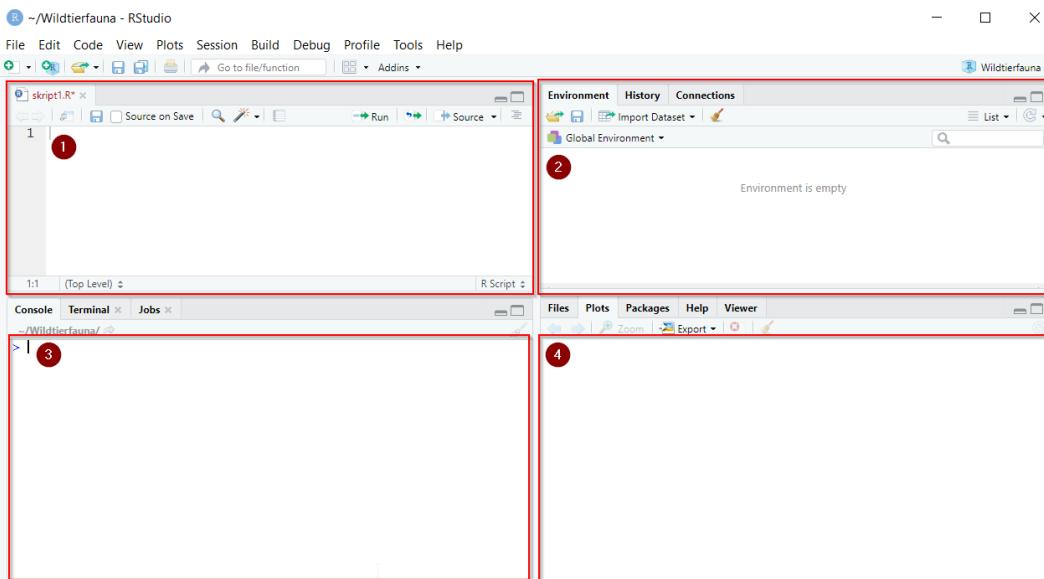


Abbildung 1: RStudio Panes.

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

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

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

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

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

**10 + 5**

153 **## [1] 15**

**20 - 10**

154 **## [1] 10**

**10 \* 3**

155 **## [1] 30**

**100 / 19**

156 **## [1] 5.263158**

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

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

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

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

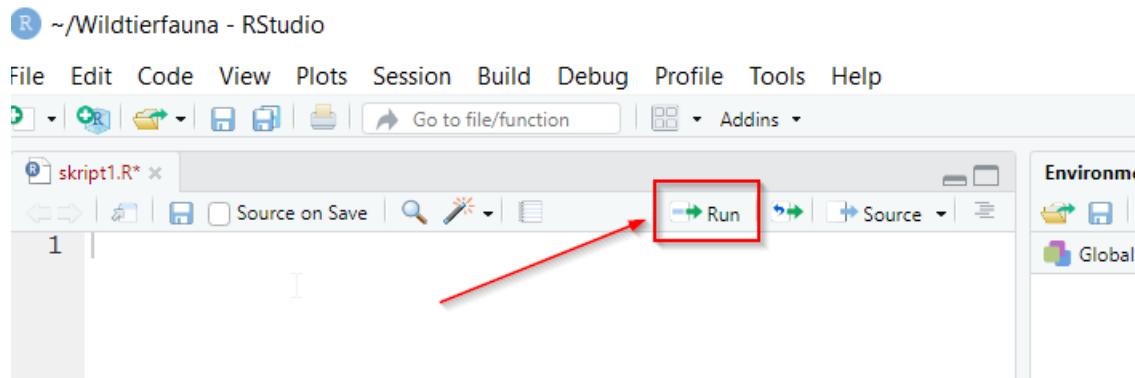


Abbildung 2: Zeilenweises Ausführen von Code in RStudio.

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

### 180 1.3 Gute Praxis bei der Programmierung

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

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

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

195 ## [1] 9

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

202

203 **Aufgabe 1: Ausführen von Quellcodes**

---

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

```
# Einfache Rechenoperationen
```

```
1 + 3
```

```
2^7
```

```
# Einfache Funktion
```

```
sqrt(20)
```

207 Führen Sie nun alle Zeilen aus.

## 208 2 Variablen, Funktionen und Datentypen

### 209 2.1 Variablen beim Programmieren

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

```
alter <- 102
```

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

```
a <- 10
a
```

217 `## [1] 10`

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

220 Wir können beliebige Variablen erstellen, z.B.

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

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

- 223 • `a_123 <- 10` ist ok
- 224 • `123_a <- 10` erzeugt einen Fehler

225 Vorsicht: Groß- und Kleinschreibung muss beachtet werden.

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

226 `## [1] "Johannes"`

227 Das Aufrufen der Variable

```
Name
```

228 führt zu einem Fehler.

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

```
a <- 10
b <- 5
a + b
```

```

231 ## [1] 15
b / a

232 ## [1] 0.5
a^b

233 ## [1] 1e+05

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

ergebnis <- a + b
ergebnis

235 ## [1] 15

ergebnis2 <- ergebnis * 2
ergebnis2

236 ## [1] 30

237 Mit der Funktion rm() können Variablen, die nicht mehr benötigte Variablen, wieder gelöscht werden. Alternativ können Variablen auch überschrieben werden. Es gibt keine Möglichkeit gelöschte oder überschriebene
238 Variablen wiederherzustellen. Sie müssen ggf. neu berechnet werden.

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

240 ## [1] TRUE

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

241 ## [1] FALSE

```

## 2.2 Datentypen

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

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

```

kamera_name <- "Kamera_1"
anzahl_rehe <- 132

```

249 In den zwei vorherigen Zeilen Code haben wir zwei Objekte (im Sinne von R) erstellt. Das erste Objekt heißt  
 250 `kamera_name` und das zweite Objekt heißt `anzahl_rehe`. In dem Beispiel handelt es sich also um zwei sehr  
 251 einfache Objekte, in denen jeweils ein Wert gespeichert ist. Auffällig ist, dass beide Objekte unterschiedliche

253 Datentypen haben. `kamera_name` ist vom Typ `character` (also Text). Das zweite Objekt, `anzahl_rehe`, ist  
 254 vom Typ `numeric` (also eine Zahl, wir unterscheiden hier nicht weiter<sup>2</sup>). Zusätzlich zu diesen zwei Typen  
 255 (`character` und `numeric`), gibt es noch einen weiteren wichtigen Typ: nämlich das logische Wahr oder Falsch  
 256 (in R: `TRUE` und `FALSE`) und noch weitere Datentypen auf die wir zunächst nicht eingehen (tippen Sie `?typeof`  
 257 für eine Übersicht aller Datentypen). Zurückkommend auf das Beispiel mit den Fotofallen, könnte eine  
 258 mögliche Fragestellung sein, ob auf einem der Fotos ein Fuchs gesehen wurde oder nicht. Dazu würden wir  
 259 eine neue Variable `fuchs_gesehen` anlegen und diese auf `TRUE` setzen, da ein Fuchs gesehen wurde.

```
fuchs_gesehen <- TRUE
```

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

```
typeof(fuchs_gesehen)
```

261 ## [1] "logical"

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

```
TRUE + TRUE
```

263 ## [1] 2

```
FALSE + FALSE
```

264 ## [1] 0

```
TRUE + FALSE
```

265 ## [1] 1

## 2.3 Funktionen

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

```
sqrt(a)
```

269 ## [1] 3.162278

270 Funktionen sind in R in der Regel daran zu erkennen, dass sie immer mit dem Funktionsnamen, gefolgt von  
 271 runden Klammern (), aufgerufen werden. Der große Umfang an Funktionen für die statistische Datenanalyse  
 272 und wissenschaftliche Datenverarbeitung ist der Hauptgrund für den Erfolg von R in der Wissenschaft. Im  
 273 vorherigen Beispiel wurde die Funktion mit dem Namen `sqrt()` aufgerufen. Das Objekt `a` haben wir bereits  
 274 vorhin definiert (zur Erinnerung `a <- 10`). Die Funktion `sqrt()` arbeitet jetzt mit dem Objekt `a`, das in  
 275 diesem Zusammenhang auch **Argument** genannt wird. Argumente sind die Objekte, die eine Funktion als  
 276 Input benötigt. Die Hilfeseite jeder Funktion enthält eine Liste aller Argumente. Argumente von Funktionen  
 277 haben Namen, diese müssen jedoch nicht angegeben werden, wenn die Reihenfolge der Argumente, wie  
 278 in der Hilfeseite angegeben, berücksichtigt wird. Im vorherigen Beispiel, haben wir die Funktion `sqrt(a)`  
 279 aufgerufen und keinen Argumentnamen angegeben. Aus den Hilfeseiten für die Funktion `sqrt()` (siehe auch

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

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

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

282 `## [1] 3.162278`

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

- 286 1. In die Konsole `?<Name der Funktion>` tippen. Also, wenn wir Hilfe für die Funktion `mean()` möchten,  
 287 könnten wir einfach `?mean` in die Konsole tippen.
- 288 2. Analog zu 1), können wir mit der Funktion `help` die Hilfeseite für eine Funktion aufrufen (z.B. wenn  
 289 wir wieder die Hilfe für die Funktion `mean()` lesen möchten, dann könnten wir auch `help(mean)` in die  
 290 Konsole tippen).
- 291 3. In R Studio kann man auch auf das Help-Tab (Pane 4) klicken und dann einfach eine Funktion suchen  
 292 (siehe Abbildung 1).
- 293 4. Wenn der Cursor in RStudio auf einer Funktion ist, kann man mit der Taste F1 die dazugehörige  
 294 Hilfeseite aufrufen.

## 295 2.4 Datenstrukturen

296 Verfolgen wir das Beispiel mit den Fotofallen etwas weiter. Es handelt sich um ein systematisches Monitoring.  
 297 D. h., es wurde nicht nur eine Fotofalle ausgebracht, sondern insgesamt 15 Stück. Dieser Umstand erfordert  
 298 komplexere Objekte. Nachfolgend sind die Anzahl Rehfotos für jede der 15 Fotofallen aufgeführt: 132, 79,  
 299 129, 91, 138, 144, 55, 103, 139, 105, 96, 146, 95, 118, 107.

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

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

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

# usw.
```

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

306 zu einem unübersichtlichen *Workspace*<sup>3</sup>. Wir werden im Verlauf sinnvollere Objekte (Vektoren oder Data  
307 Frames) für diesen Zweck kennenlernen.

308

309 **Aufgabe 2: Variablen**

---

310 Verwenden Sie die folgenden Daten

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

312 und berechnen sie:

- 313 •  $10 * a$   
314 •  $a / 144$  und speichern Sie das Ergebnis in einer neuen Variablen  $e$  zwischen.  
315 • Was ist das Ergebnis von  $a + b$ ?  
316 • Was ist das Ergebnis von  $a + p$ ?

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

317 **2.5 Funktionen**

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

```
sqrt(a)
```

320 `## [1] 1.414214`

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

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

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

---

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

330 ## [1] 1.414214

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

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

### 3 Vektoren

343 Die gute Nachricht zuerst. Sie haben bereits Vektoren erstellt in R (und dies wahrscheinlich nicht bewusst wahrgenommen). Wenn Sie nämlich eine Objekt mit einem Element erstellen (z.B., `a <- 10`), wird ein Vektor der Länge eins erstellt. Das heißt, der Vektor enthält genau ein Element (einen Eintrag). Vektoren sind also 346 kein neues Objekt für Sie, sondern Sie lernen jetzt, dass die Ihnen schon bekannten Objekte Vektoren heißen 347 und sie auch mehrere Elemente in eine mObjekt speichern können.

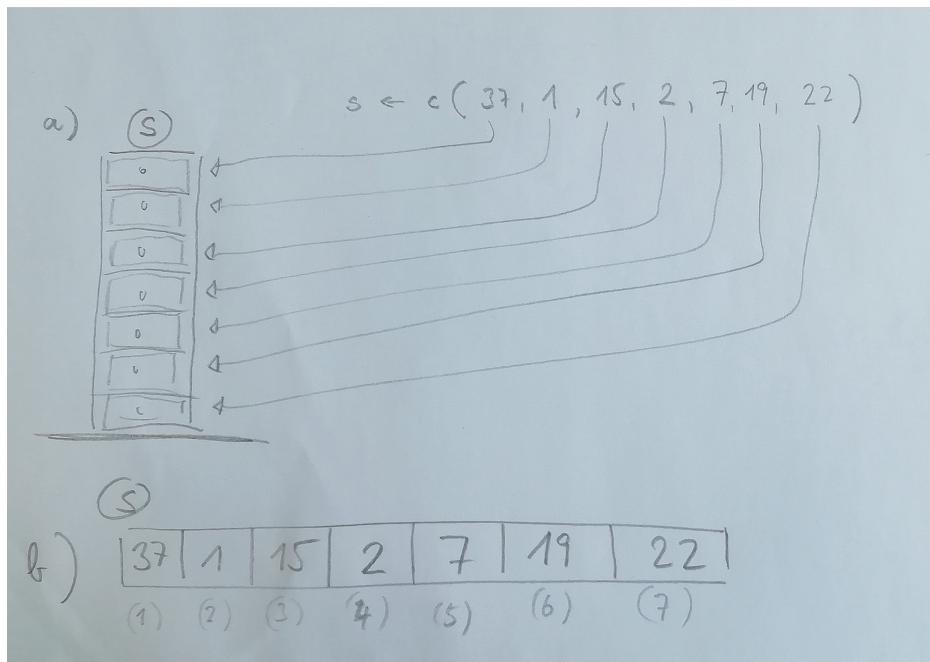


Abbildung 3: Schematische Darstellung eines Vektors in R.

344 Sie können sich Vektoren wie einen Schubladenschrank vorstellen (siehe auch Abbildung 3). Wichtig ist dabei, 345 dass man in jede Schublade immer nur ein Element verstauen kann und alle Elemente im Schublagenschrank 351 den gleichen Datentyp haben müssen. Etwas allgemeiner gesprochen heißt das, dass alle Elemente eines 352 Vektors vom gleichen Datentyp sein müssen.

353 Es gibt zahlreiche Funktionen zum Erstellen von Vektoren (einige davon werden wir im weiteren Verlauf des 354 Moduls kennenlernen). Die wohl wichtigste Funktion ist `c()`. Dabei steht `c` für *combine* oder *concatenate*. 355 Die Funktion `c()` fügt einzelne Elemente in einen Vektor zusammen (und zwar genau in der Reihenfolge wie 356 diese Elemente an `c()` übergeben werden). `c()` ist sozusagen die Funktion, die für Sie mehrere Elemente zu 357 einem Vektor zusammensetzt. `c()` erwartet folglich nur Elemente als Argumente.

358 Gehen wir nochmals zurück zu Abbildung 3, in der schematisch dargestellt wird, wie ein Vektor `s` mit 7 359 Elementen (in diesem Fall Zahlen) erstellt wird.

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

360 Die Funktion `c()` ordnet jetzt bildlich gesprochen die Zahl 37 der ersten Schublade zu, die Zahl 1 der zweiten 361 Schublade und so weiter. Wenn Sie jetzt einfach `s` in die Konsole tippen, können Sie alle Elemente von `s` 362 sehen:

s

363 ## [1] 37 1 15 2 7 19 22

364 In Abbildung 3b 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.

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

s + 10

369 ## [1] 47 11 25 12 17 29 32

370 oder `s` mit sich selbst multiplizieren. Zu beachten ist also, dass es sich bei Vektorberechnungen in R  
371 zunächst nicht um Vektorrechnungen handelt, wie Sie sie aus der linearen Algebra kennen. Für die sog.  
372 Matrizenoperationen der linearen Algebra werden die Operatoren in R mit `% %` umschlossen, also bspw. `s  
373 %*% s`.

s \* s

374 ## [1] 1369 1 225 4 49 361 484

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

seq(from = 1, to = 10)

378 ## [1] 1 2 3 4 5 6 7 8 9 10

(1 : 10)

379 ## [1] 1 2 3 4 5 6 7 8 9 10

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

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

381 ## [1] 1 3 5 7 9

382

### 383 Aufgabe 3: Vektoren erstellen

---

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

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

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

### 3.1 Funktionen zum Arbeiten mit Vektoren

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

```
head(s)
```

392 ## [1] 37 1 15 2 7 19

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

393 ## [1] 37 1 15

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

394 ## [1] 19 22

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

```
length(s)
```

396 ## [1] 7

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

```
class(s)
```

398 ## [1] "numeric"

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

```
unique(s)
```

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

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

```
table(s)
```

402 ## s

403 ## 1 2 7 15 19 22 37

404 ## 1 1 1 1 1 1 1

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

```
rev(s)
```

407 ## [1] 22 19 7 2 15 1 37

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

```
sort(s)
```

409 ## [1] 1 2 7 15 19 22 37

---

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

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

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

411 ## [1] 37 1 15 2 7 19 22 37 1 15 2 7 19 22

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

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

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

414 ## [1] 1 2 3 4 1 2 3 4

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

415 ## [1] 1 1 2 2 3 3 4 4

416

---

#### 417 Aufgabe 4: Arbeiten mit Vektoren

---

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

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

420 Diese wurden immer abwechselnd mit zwei unterschiedlichen Messgeräten durchgeführt wurden.

421 Erstellen Sie einen Vektor von der Länge 8 mit den Einträgen, die immer abwechselnd G1 und G2 sind und  
422 für die zwei Geräte stehen.

## 423 3.2 Statistische Funktionen

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

```
mean(s)
```

427 ## [1] 14.71429

```
median(s)
```

428 ## [1] 15

```
sd(s)
```

429 ## [1] 12.76341

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

```

sample(s, size = 1) # 1 Element
433 ## [1] 1
sample(s, size = 3) # 2 Elemente
434 ## [1] 15 7 22

```

435 Wenn `size` weggelassen wird, dann bekommt man gleich viele Elemente zurück (wie der Vektor lang ist), d.h.  
 436 der Vektor wird nur permutiert.

### 437 3.3 Beispiel Fotofallen

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

441 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,
442 105, 96, 146, 95, 118, 1007)

```

443 Als zweites erstellen wir einen Vektor mit den IDs. Zur Erinnerung, diese sollten die Werte `Kamera_1` bis  
 444 `Kamera_15` haben. Ein erster Ansatz könnte sein, dass wir einfach 15 Fotofallen schreiben und dann die  
 Zahlen 1 bis 15 dahinter.

```

ids <- c("Kamera_1", "Kamera_2", "Kamera_3", "Kamera_4", "Kamera_5",
445 "Kamera_6", "Kamera_7", "Kamera_8", "Kamera_9", "Kamera_10",
  "Kamera_11", "Kamera_12", "Kamera_13", "Kamera_14", "Kamera_15"
)

```

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

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

```

v1 <- rep("Kamera", 15)

```

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

```

v2 <- 1:15

```

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

```

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

```

```

455 ## [1] "Kamera_1"  "Kamera_2"  "Kamera_3"  "Kamera_4"  "Kamera_5"  "Kamera_6"
456 ## [7] "Kamera_7"  "Kamera_8"  "Kamera_9"  "Kamera_10" "Kamera_11" "Kamera_12"
457 ## [13] "Kamera_13" "Kamera_14" "Kamera_15"

```

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

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

```

460 ## [1] "Revier A" "Revier B" "Revier C" "Revier A" "Revier B" "Revier C"
461 ## [7] "Revier A" "Revier B" "Revier C" "Revier A" "Revier B" "Revier C"
462 ## [13] "Revier A" "Revier B" "Revier C"

```

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

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

```

465 ## [1] "Revier A" "Revier A" "Revier A" "Revier A" "Revier A" "Revier B"
466 ## [7] "Revier B" "Revier B" "Revier B" "Revier B" "Revier C" "Revier C"
467 ## [13] "Revier C" "Revier C" "Revier C"

```

468

#### 469 Aufgabe 5: Statistische Funktionen

---

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

472 2. Erstellen Sie die folgende Konsolenausgabe:

```
473 ## [1] "Die mittlere Anzahl von Rehfotos beträgt 171.8 Rehe pro Standort."
```

### 474 3.4 Arbeiten mit logischen Werten

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

- 477 • Gleichheit (`==`)
- 478 • Ungleichheit (`!=`)
- 479 • Größer (`>`) und kleiner (`<`)
- 480 • Größer gleich (`>=`) und kleiner gleich (`<=`)

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

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

```
anzahl_rehe > 100
```

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

```

485 ## [13] FALSE TRUE TRUE
486 Das Ergebnis ist ein Vektor vom Datentyp logi in der selben Länge wie anzahl_rehe.
487 Wir können z. B. abfragen, welche Fotofallenstandorte sich in Revier B befinden.
488 reviere == "Revier B"
489 ## [13] FALSE FALSE FALSE FALSE FALSE TRUE TRUE TRUE TRUE TRUE FALSE FALSE
490 Des Weiteren können logische Ausdrücke miteinander verknüpft werden. Dies geschieht mit einem logischen Und (&) oder einem logischen Oder (|). Für das logische Und müssen beide Ausdrücke ein TRUE zurückgeben
491 um ein TRUE zu erhalten. Für ein logisches Oder reicht es, wenn einer der beiden Ausdrücke TRUE zurückgibt,
492 um ein TRUE zu erhalten.
493
494 Damit können wir nun z. B. die beiden vorherigen Abfragen verbinden. Die erste Abfrage ist: Hat eine
495 Fotofalle mehr als 100 Rehe fotografiert und stand die Fotofalle in Revier B.
496 anzahl_rehe > 100 & reviere == "Revier B"
497 ## [13] FALSE FALSE FALSE FALSE FALSE TRUE FALSE TRUE TRUE TRUE FALSE FALSE
498 Das war jetzt eine Abfrage mit einem logischen Und. Würden wir ein logisches Oder verwenden, dann
499 bekommen wir für alle Elemente ein TRUE, die entweder in Gebiet B stehen oder mehr als 100 Rehfotos
500 aufgezeichnet haben.
501 anzahl_rehe > 100 | reviere == "Revier B"
502 ## [13] TRUE FALSE TRUE FALSE TRUE TRUE TRUE TRUE TRUE TRUE FALSE TRUE
503 Das Arbeiten mit logischen Werten kann fürs Erste etwas abstrakt erscheinen, aber wir werden im folgenden
504 Abschnitt (Abschnitt 3.5) zahlreiche Anwendungsbeispiele dafür sehen.
505

```

### 506 Aufgabe 6: Arbeiten mit logischen Werten

---

```

508 Überlegen Sie für jede Aufgabe erst was das richtige Ergebnis ist und Überprüfen Sie dieses dann mit R.
509 1. TRUE | FALSE
510 2. FALSE & TRUE
511 3. (FALSE & TRUE) | TRUE
512 4. (2 != 3) | FALSE
513 5. FALSE + 10
514 6. TRUE + 10
515 7. TRUE + 10 == FALSE + 10
516 8. sum(c(TRUE, TRUE, FALSE, FALSE))

```

### 517 3.5 Zugreifen auf Elemente eines Vektors (=Untermengen)

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

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

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

```
anzahl_rehe[2]
```

531 ## [1] 79

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

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

533 ## [1] 132 79 129 91 138

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

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

537

#### 538 Aufgabe 7: Zugreifen auf Vektorelemente

---

540 Erstellen Sie einen neuen Vektor `bhd`

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

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

544

---

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

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

547 ## [1] 132 79 129 91 138

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

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

551 ## [1] 132 79 129 91 138

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

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

554 ## [1] 132 79 129 91 138

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

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

557 ## [1] 113.8

558

### 559 Aufgabe 8: logische Werte

---

561 Verwenden Sie nochmals den Vektor mit den Anzahl Rehen die an unterschiedlichen Fotofallenstandorten  
 562 fotografiert wurden.

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

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

## 566 3.6 Der %in%-Operator

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

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

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

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

571 ## [1] "FI" "FI"

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

572 ## [1] "FI" "FI"

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

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

575 ## [1] "FI" "BU" "BU" "FI"

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

```
messungen_arten %in% arten
```

579 ## [1] TRUE TRUE TRUE FALSE FALSE FALSE TRUE FALSE FALSE

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

580 ## [1] "FI" "BU" "BU" "FI"

581

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

---

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

```
LETTERS
```

585 ## [1] "A" "B" "C" "D" "E" "F" "G" "H" "I" "J" "K" "L" "M" "N" "O" "P" "Q" "R" "S"

586 ## [20] "T" "U" "V" "W" "X" "Y" "Z"

- 587 Wählen Sie aus `LETTERS` nur die Vokale aus.

## 588 4 Faktoren (factors)

589 <<< HEAD R besitzt einen besonderen Datentyp – *Faktoren* (engl. factors) – zum speichern von kate-  
 590 gorialen Kovariaten (z.B. Baumart, Augenfarbe oder Automarke). Faktoren erlauben es Daten vom Typ  
 591 `character` effizienter abzuspeichern. Dabei wird jeder eindeutiger Wert (=Level) mit einer Zahl codiert  
 592 und dann werden nur diese Zahlen zusammen mit einer Tabelle zum Nachschauen (engl. look-up table) der  
 593 Werte gespeichert (siehe dazu auch [McNamara and Horton 2018](#)). ===== R besitzt einen besonderen  
 594 Datentyp – *Faktoren* (engl. factors) – zum speichern von diskreten Kovariaten (z.B. Baumart, Augenfarbe  
 595 oder Automarke). Faktoren erlauben es Daten vom Typ `character` effizienter abzuspeichern. Dabei wird  
 596 jeder eindeutiger Wert (=Level) mit einer Zahl codiert und dann werden nur diese Zahlen zusammen mit  
 597 einer Tabelle zum Nachschauen der Werte gespeichert (siehe dazu auch [McNamara and Horton 2018](#)). Fak-  
 598 toren haben im Gegensatz zu Zeichenketten zusätzliche Eigenschaften. Man kann sie z. B. sortieren. >>>  
 599 629c903f27f5c5db526dfd1588596c4a4b00635b

600 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)
```

```
601 ## [1] FI BU FI EI EI FI FI
602 ## Levels: BU EI FI
```

603 <<< HEAD Ohne weitere Spezifikation werden für die *Levels* die Werte alphabetisch übernommen (das  
 604 kann später z.B. beim Erstellen von Abbildungen wichtig sein). Für eine benutzerdefinierte Anordnungs  
 605 der Levels, kann das Argument `levels` verwendet werden. ===== Ohne weitere Spezifikation wer-  
 606 den die Werte *Levels* alphabetisch angeordnet (das kann später z. B. beim Erstellen von Abbildungen  
 607 wichtig sein), dies kann jedoch durch die Verwendung des Arguments `levels` gesteuert werden. >>>
 608 629c903f27f5c5db526dfd1588596c4a4b00635b

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

```
609 ## [1] FI BU FI EI EI FI FI
610 ## Levels: FI BU EI
```

611 Es ist auch möglich, die Beschriftung (= labels) der unterschiedlichen Levels anzugeben mit dem Argument  
 612 `labels`.

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

```
613 ## [1] Fichte Buche Fichte Eiche Eiche Fichte Fichte
614 ## Levels: Fichte Buche Eiche
```

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

```
levels(af)
```

```
617 ## [1] "Fichte" "Buche" "Eiche"
```

```

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

618 ## [1] Fi Bu Fi Ei Ei Fi Fi
619 ## Levels: Fi Bu Ei

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

af

622 ## [1] Fi Bu Fi Ei Ei Fi Fi
623 ## Levels: Fi Bu Ei

relevel(af, "Bu")

624 ## [1] Fi Bu Fi Ei Ei Fi Fi
625 ## Levels: Bu Fi Ei

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

as.character(af)

628 ## [1] "Fi" "Bu" "Fi" "Ei" "Ei" "Fi" "Fi"
629 Achtung mit der Funktion as.numeric() erhält man die interne Kodierung von Faktoren.

af

630 ## [1] Fi Bu Fi Ei Ei Fi Fi
631 ## Levels: Fi Bu Ei

as.numeric(af)

632 ## [1] 1 2 1 3 3 1 1
633 Fichte ist das erste Level des Faktors, deshalb erhalten alle Fichteneinträge den Wert 1. Bucheinträge erhalten
634 den Wert 2 und 3 für Eichen.

635

```

---

### 636 Aufgabe 10: Faktoren

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

```

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

```

## 640 4.1 Das Paket `forcats`

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

- 643 1. Die Anordnung von Levels anzupassen.  
644 2. Levels zusammenzufassen oder zu entfernen.  
645 3. Labels zu ändern.

### 646 4.1.1 Anpassen der Anordnung von Faktoren

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

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

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

```
factor(a)
```

649 ## [1] FI BU FI EI EI FI FI  
650 ## Levels: BU EI FI

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

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

652 ## [1] FI BU FI EI EI FI FI  
653 ## Levels: FI BU EI

654 `forcats` stellt Funktionen zur Verfügung, um die Reihenfolge umzukehren,

```
fct_rev(f1)
```

655 ## [1] FI BU FI EI EI FI FI  
656 ## Levels: EI BU FI

657 nach der Häufigkeit zu sortieren oder

```
fct_infreq(f1)
```

658 ## [1] FI BU FI EI EI FI FI  
659 ## Levels: FI EI BU

660 eine zufällige Sortierung.

```
fct_shuffle(f1)
```

661 ## [1] FI BU FI EI EI FI FI  
662 ## Levels: EI FI BU

## 5 Spezielle Einträge

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

- 665 • fehlenden Einträgen NA,
- 666 • leeren Einträgen NULL,
- 667 • undefinierten Einträgen NaN (Not a Number) oder
- 668 • unendlichen Zahlen (Inf).

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

### 670 5.1 NA

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

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

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

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

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

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

```
680 ## [1] "foo" "foo"
681 ## attr(),"na.action")
682 ## [1] 2
683 ## attr(),"class")
684 ## [1] "omit"
```

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

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

**1 + NA**

689 `## [1] NA`  
690 Viele R Funktionen haben eingebaute Methoden zum Umgang mit NA. Die Funktion `mean()` bspw. ergibt  
691 (wie die meisten Funktionen) standardmäßig NA wenn sie auf Vektoren mit Datenlücken angewendet wird, es  
692 sei denn man stellt innerhalb der Funktion ein, dass Datenlücken entfernt werden sollen.

`mean(na2)`

693 `## [1] NA`  
694 `mean(na2, na.rm = TRUE)`  
694 `## [1] 4.5`

## 695 5.2 NULL

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

## 700 5.3 Inf

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

`10^309`

703 `## [1] Inf`  
704 `2 * Inf`  
704 `## [1] Inf`  
705 `1 + Inf`  
705 `## [1] Inf`  
706 `3 / 0`  
706 `## [1] Inf`  
707 `-3 / 0`  
707 `## [1] -Inf`  
708 `3 / Inf`  
708 `## [1] 0`

709 Infinity kann mit `is.infinite` und `is.finite` getestet werden. Relationäre Operatoren funktionieren  
710 erwartungsgemäß.

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

711 ## [1] TRUE FALSE FALSE TRUE FALSE
is.finite(inf1)

712 ## [1] FALSE TRUE TRUE FALSE TRUE
inf1 < 3

713 ## [1] FALSE TRUE FALSE TRUE FALSE
```

714

---

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

---

717 Verwenden Sie den Vektor

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

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

720 Verwenden Sie den Vektor

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

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

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

726 Berechnen Sie den arithmetischen Mittelwert von `foo`.

## 727 6 data.frames oder Tabellen

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

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

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

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

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

761 Standard-Objekte zum Speichern wissenschaftlicher Daten.

## 762 6.1 Wichtige Funktionen zum Arbeiten mit `data.frames`

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

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

```
765 ##           ID anzahl_rehe    revier
766 ## 1 Kamera_1          132 Revier A
767 ## 2 Kamera_2          79 Revier A
```

768 Oder für die letzten 2 Beobachtungen.

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

```
769 ##           ID anzahl_rehe    revier
770 ## 14 Kamera_14         118 Revier C
771 ## 15 Kamera_15         107 Revier C
```

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

```
nrow(monitoring)
```

```
773 ## [1] 15
```

```
ncol(monitoring)
```

```
774 ## [1] 3
```

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

```
str(monitoring)
```

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

781

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

---

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

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

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

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

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

796 `## [1] 91`

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

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

800 `## [1] 91`

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

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

803 `## [1] 132 79 129 91 138`

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

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

806 `## anzahl_rehe revier`

807 `## 1 132 Revier A`

808 `## 2 79 Revier A`

809 `## 3 129 Revier A`

810 `## 4 91 Revier A`

811 `## 5 138 Revier A`

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

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

814 `## ID anzahl_rehe revier`

815 `## 1 Kamera_1 132 Revier A`

816 `## 2 Kamera_2 79 Revier A`

817 `## 3 Kamera_3 129 Revier A`

```
818 ## 4 Kamera_4          91 Revier A
819 ## 5 Kamera_5          138 Revier A
```

820

---

**821 Aufgabe 13: Abfragen von Werten**

---

823 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)
)
```

- 824 • Wählen Sie nur die ersten drei Zeilen aus und die erste und zweite Spalte aus.  
 825 • Wählen Sie alle Zeilen und die erste und dritte Spalte aus.  
 826 • Wählen Sie alle Spalten und die erste, dritte und vierte Zeile aus.

827

---

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

830 1. über das \$-Zeichen direkt die Spalten ansprechen.

```
monitoring$anzahl_rehe
```

```
831 ## [1] 132 79 129 91 138 144 55 103 139 105 96 146 95 118 107
```

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

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

```
833 ## [1] 132 79 129 91 138 144 55 103 139 105 96 146 95 118 107
```

834 3. Alle Zeilen und die Spalte explizit angeben.

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

```
835 ## [1] 132 79 129 91 138 144 55 103 139 105 96 146 95 118 107
```

836 Anmerkung zu 3), der Ausdruck `1:nrow(monitoring)` ergibt einen Vektor mit den Zahlen 1 bis 15, da  
 837 `nrow(monitoring) = 15` ist. So eine Schreibweise ist zu empfehlen, wenn die Dimension des Vektors variabel  
 838 ist.

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

```
monitoring$anzahl_rehe > 100
```

```
842 ## [1] TRUE FALSE TRUE FALSE TRUE TRUE FALSE TRUE TRUE TRUE FALSE TRUE
843 ## [13] FALSE TRUE TRUE
844 Das Ergebnis ist ein Vektor mit 15 Elementen. Hat eine Fotofalle mehr als 100 Rehfotos gemacht ist das
845 entsprechende Element des Vektors TRUE ansonsten FALSE. In dem data.frame monitoring steht in jeder
846 Zeile eine Beobachtung (also eine Fotofalle). Nun wollen wir genau diese Fotofallen haben, die mehr als 100
847 Rehfotos gemacht gemacht haben.
848 monitoring[monitoring$anzahl_rehe > 100, ]
849 ## #> #> ID anzahl_rehe revier
850 ## #> 1 Kamera_1 132 Revier A
851 ## #> 3 Kamera_3 129 Revier A
852 ## #> 5 Kamera_5 138 Revier A
853 ## #> 6 Kamera_6 144 Revier B
854 ## #> 8 Kamera_8 103 Revier B
855 ## #> 9 Kamera_9 139 Revier B
856 ## #> 10 Kamera_10 105 Revier B
857 ## #> 12 Kamera_12 146 Revier C
858 ## #> 14 Kamera_14 118 Revier C
859 ## #> 15 Kamera_15 107 Revier C
```

---

**Aufgabe 14: Abfragen von Werten 2**

---

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

861

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

## 866 7 Schreiben und lesen von Daten

### 867 7.1 Textdateien

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

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

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

883 Die Datei `fotofallen.csv` finden Sie auf StudIP und kann einfach heruntergeladen werden. Sie können  
 884 sich die Datei mit jedem Texteditor oder auch mit Excel oder Libre Office ansehen. Die Datei kann mit  
 885 dem folgenden Befehl in R eingelesen werden. Hier wurde die Datei in einem R Studio-Projekt in ein  
 886 Unterverzeichnis `data` abgelegt.

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

```
887 ##          ID anzahl_rehe   revier  
888 ## 1 Kamera_1        132 Revier A  
889 ## 2 Kamera_2        79 Revier A  
890 ## 3 Kamera_3        129 Revier A  
891 ## 4 Kamera_4        91 Revier A  
892 ## 5 Kamera_5        138 Revier A  
893 ## 6 Kamera_6        144 Revier B
```

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

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

---

<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.

899

900 **Aufgabe 15: Lesen und Schreiben von Datein**  
901

---

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

## 8 Erstellen von Abbildungen

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

### 8.1 Base Plot

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

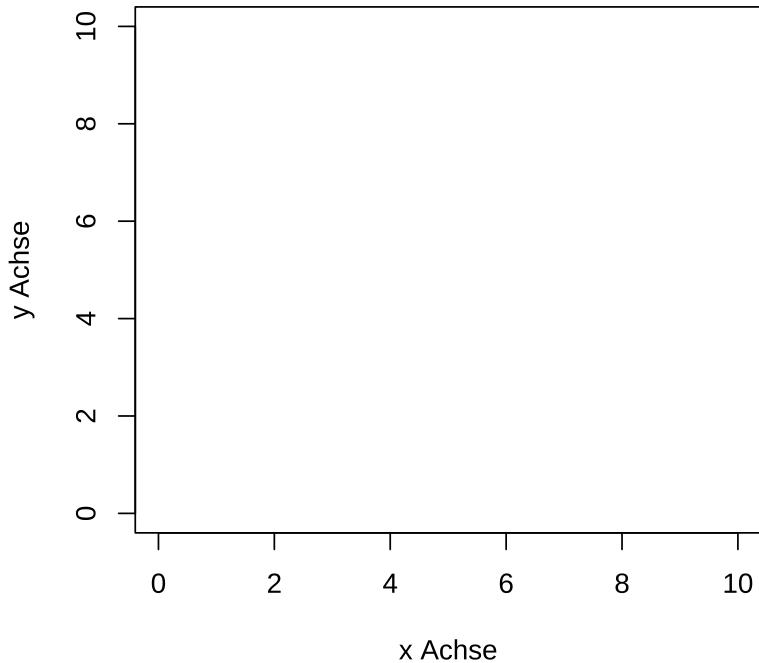
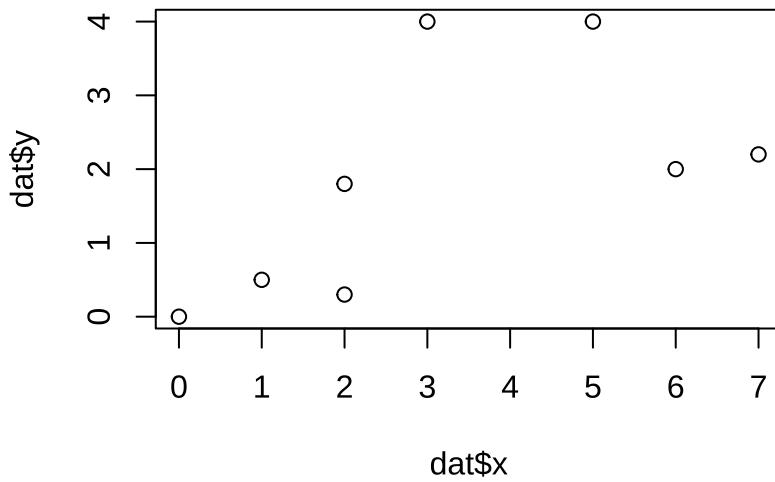


Abbildung 4: Beispiel einer leeren Grafikschnittstelle.

Hier drei einfache Beispiele für Abbildungen mit nur einer Ebene.

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

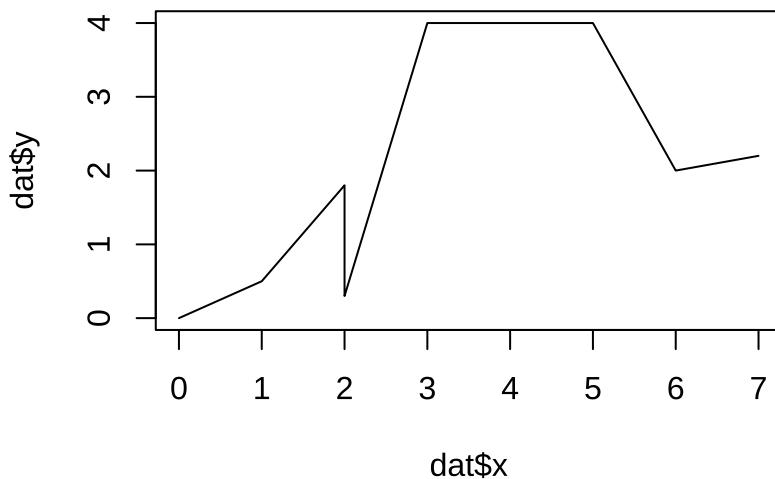
plot(dat$x, dat$y, type = "p")
```



918

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

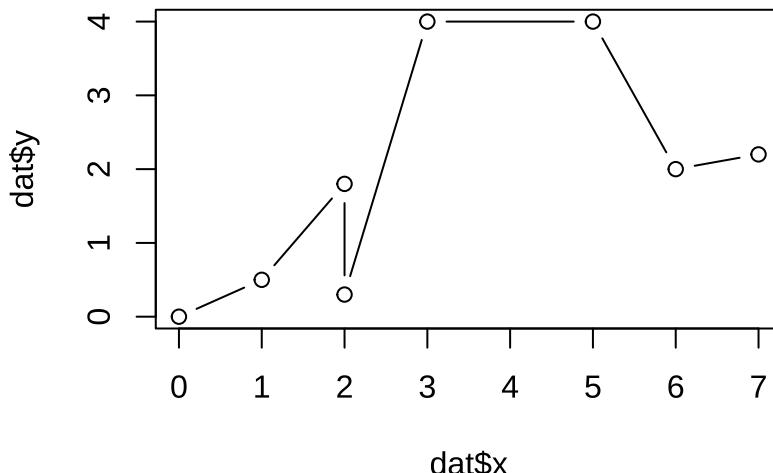
```
plot(dat$x, dat$y, type = "l")
```



921

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

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



923

924 darstellen.

925

926 **Aufgabe 16: Base Plot 1**927  
928 Laden Sie den Datensatz `bhd_1.txt` und erstellen Sie eine Abbildung mit dem Alter jedes Baumes auf der  
929 x-Achse und dem BHD auf der y-Achse.

930

931 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  
932 Elemente hinzufügen. Mit jeder neuen Ebene können Sie die vorigen Ebenen jedoch nicht mehr verändern.  
933 Die wichtigsten Argumente der `plot` Funktion sind:

- 935 • `type` - Diagrammtyp
- 936 • `col` - Farbe
- 937 • `main` - Titel
- 938 • `sub` - Untertitel
- 939 • `pch` - Punktsymbol
- 940 • `lty` - Linientyp
- 941 • `lwd` - Linienstärke
- 942 • `xlab` bzw. `ylab` - Achsenbeschriftungen
- 943 • `xlim`, `ylim` - Grenzen der Achsenanschnitte
- 944 • `axes` - Sollen die Achsen eingezeichnet werden? Oder leer gelassen werden, um sie nachträglich als  
945 low-level Ebene einzuziehen?
- 946 • `ann` - Achsenbeschriftung kann ganz weggelassen werden.

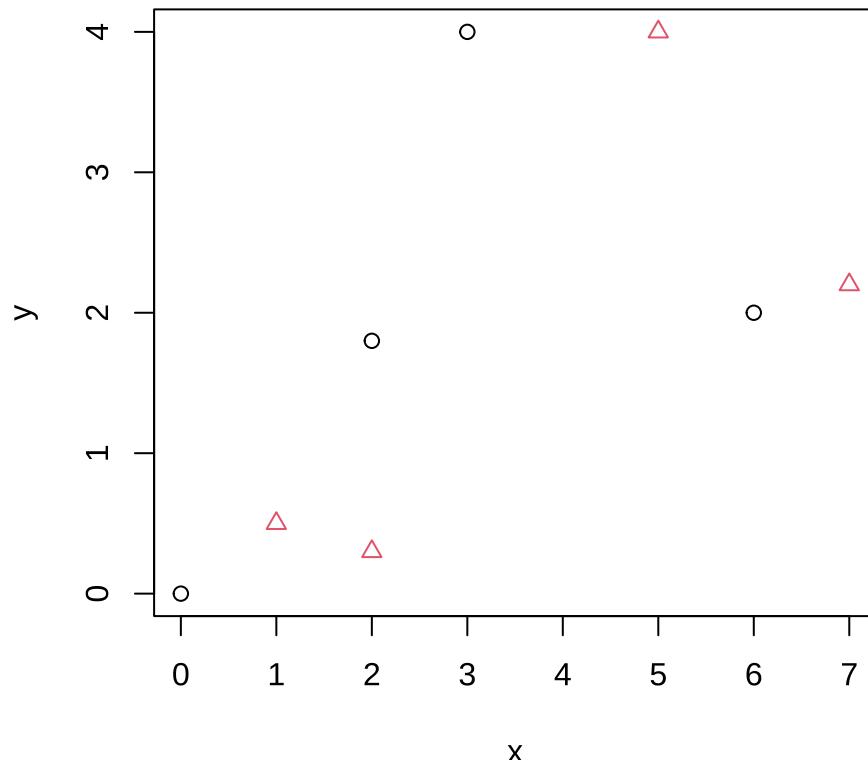
947 Sehen Sie sich die Hilfeseiten `?plot.default()` oder `?par()` an für weitere Informationen. Dort finden Sie  
948 auch eine vollständige Liste der Befehle. Einige Argumente können als Vektor übergeben werden. Hier z. B.  
949 die Farben und die Punktsymbole.

```

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")

```



950

951

---

**Aufgabe 17: Anpassen von Plots**


---

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

- 955 • Beschriften Sie die x- und y-Achse sinnvoll.
- 956 • Fügen Sie eine Überschrift hinzu.
- 957 • Wählen Sie ein anderes Symbol.
- 958 • Stellen Sie die Symbole in rot dar.

959

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

961 Die wichtigsten Funktionen sind

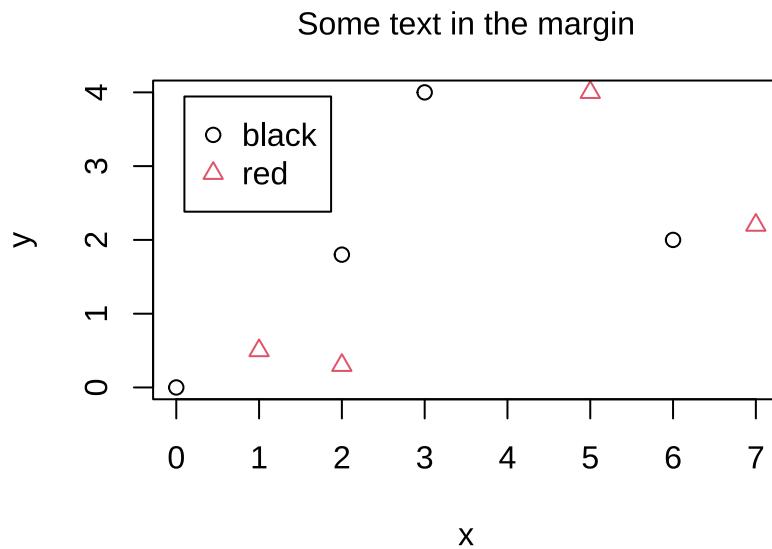
- 962 • `points()` - Fügt Punkte ein
- 963 • `lines()` - Fügt Linien ein

- 964 • `text()` - Fügt Text ein  
 965 • `mtext` - Fügt Text in den Rahmen (`margin`) ein  
 966 • `legend()` - Fügt eine Legende ein  
 967 • `abline()` - Fügt eine Gerade ein  
 968 • `curve()` - Fügt eine mathematische Funktion ein  
 969 • `arrows()` - Fügt Pfeile ein  
 970 • `grid()` - Fügt Hilfslinien ein

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

974 Legenden können im base plot nur als Low-Level Funktion hinzugefügt werden. Mit der Funktion `legend`  
 975 werden die Einträge und die entsprechenden Punktsymbole oder Linientypen dargestellt. In der folgenden  
 976 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")
```



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

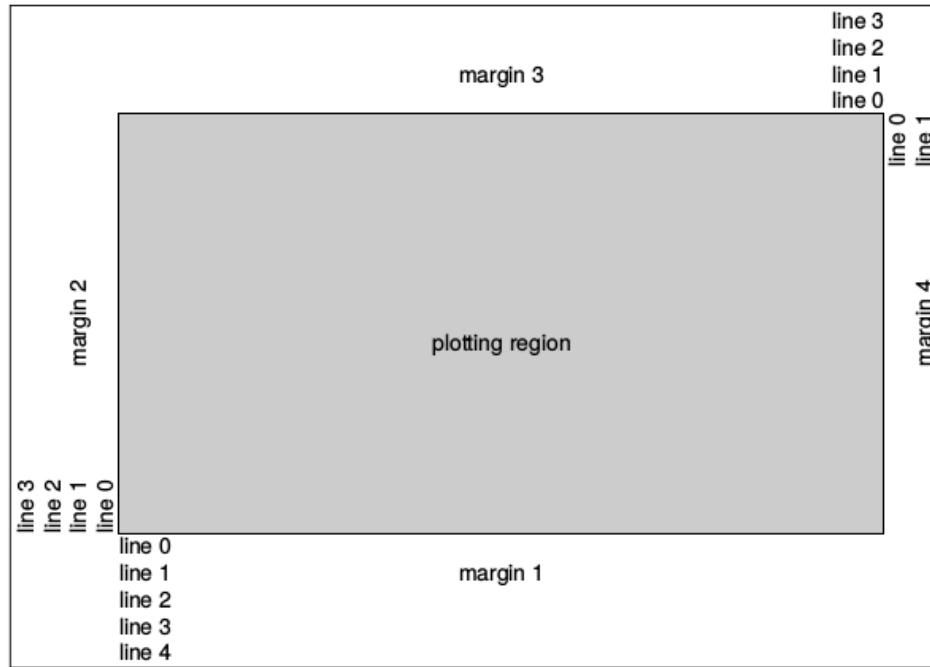
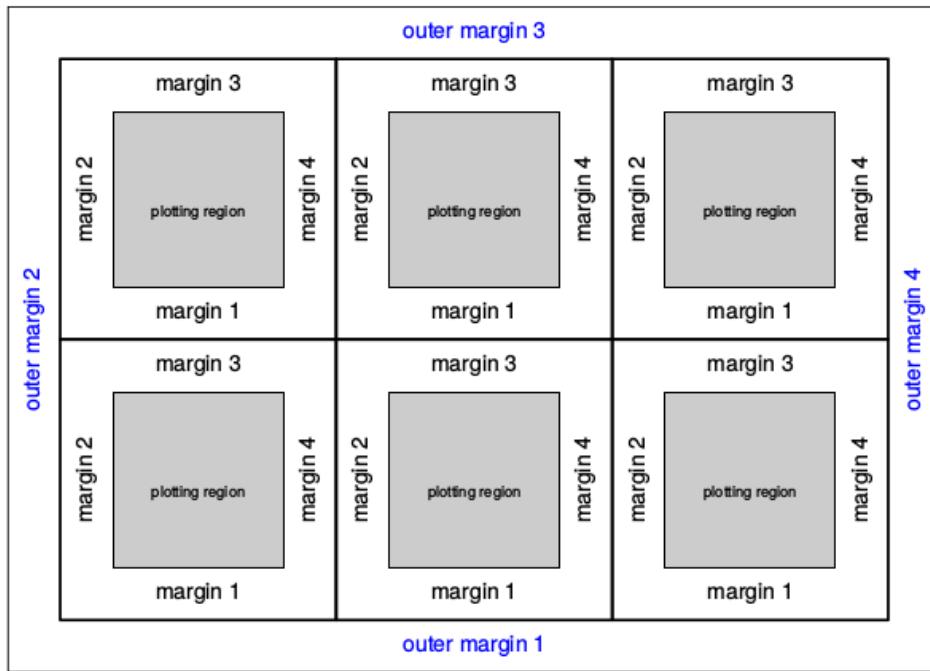


Abbildung 5: Grafikregionen eines base plots in R.

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

982 **8.1.1 Mehrere Panels**

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

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

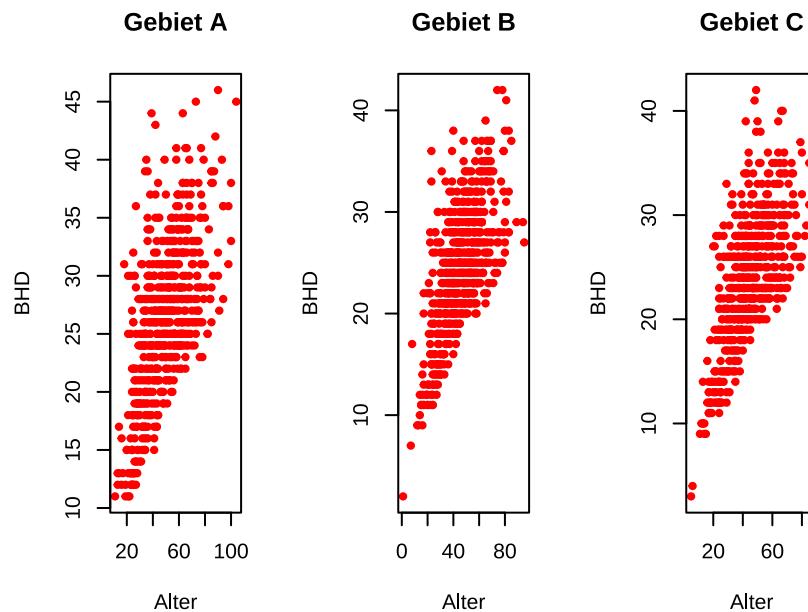
986 Teilt den Plot in eine Zeile und drei Zeilen (= drei Plots nebeneinander).

```
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")
```



987

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

990 **8.1.2 Speichern von Abbildungen**

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

- 995     • `pdf()` oder  
 996     • `postscript()`.

997 Beispiele für Rastergrafiken sind

- 998     • `png()`,  
 999     • `bmp()` oder  
 1000    • `jpeg()`.

1001 Die Grafikschnittstelle ist dann Ihre “Leinwand”. Mit dem Befehl `dev.off()` trennen Sie die Verbindung zur  
 1002 Schnittstelle wieder. Ihre “Leinwand” wird also wieder geschlossen. So lange die Schnittstelle geöffnet ist  
 1003 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
```

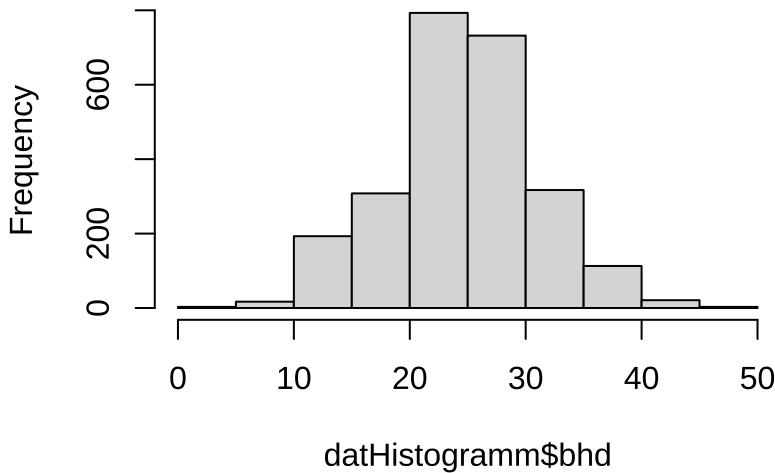
1004 *Achtung*, wenn Sie die Funktion `dev.off()` nicht aufrufen, werden alle nachfolgenden Plots in die gleiche  
 1005 Datei geschrieben. Falls Sie nach einem Versuch einen Plot zu speichern plötzlich keine weiteren Plots mehr  
 1006 sehen, führen Sie einige Mal die Funktion `dev.off()` aus.

## 1007 8.2 Histogramme

1008 Neben den Streudiagrammen (*Scatterplots*, oder auch einfach x-y Diagramm) sind *Histogramme* in der  
 1009 angewandten Datenanalyse ein weiterer wichtiger Abbildungstyp. An Histogrammen wird die Häufigkeit  
 1010 von Beobachtungen nach Gruppen dargestellt. Sie sind deshalb so wichtig, weil man aus ihnen relevante  
 1011 Informationen über die Verteilung der Daten ablesen kann und somit einiges über die Messungen erfährt,  
 1012 das für die weitere Datenanalyse relevant sein kann. So werden auf einen Blick der Zusammenhang von  
 1013 Beobachtungshäufigkeit und Streuung deutlich, sowie auch die Form der Verteilung und ihre Schiefe. Die  
 1014 Interpretation werden wir bei den Boxplots noch weiter vertiefen.

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

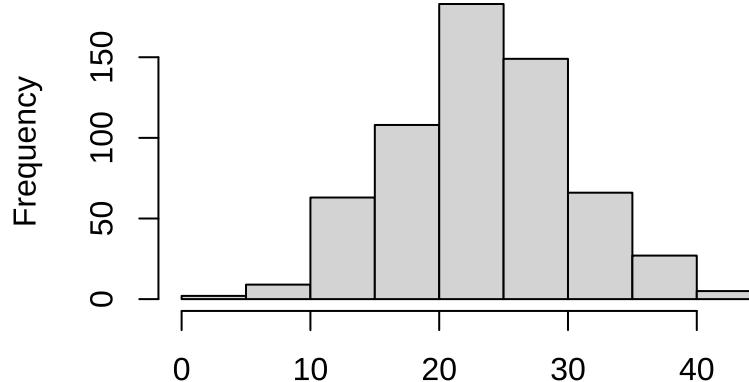
### Histogramm of datHistogramm\$bhd



1015

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

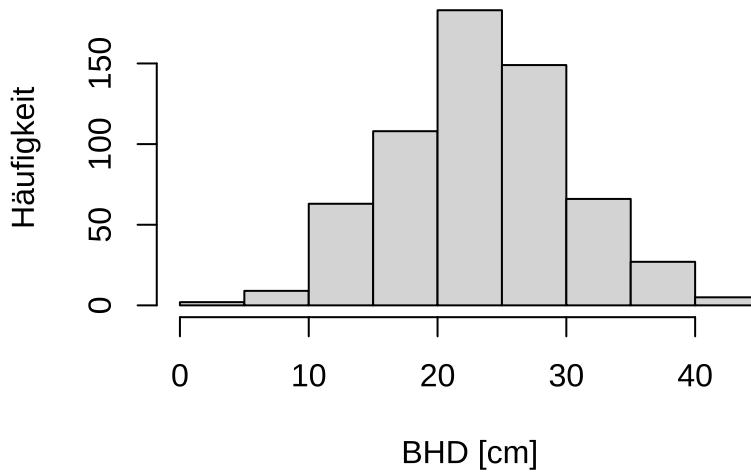
### Histogramm of datHistogramm\$bhd[datHistogramm\$art == "EI"]



1016

```
# 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

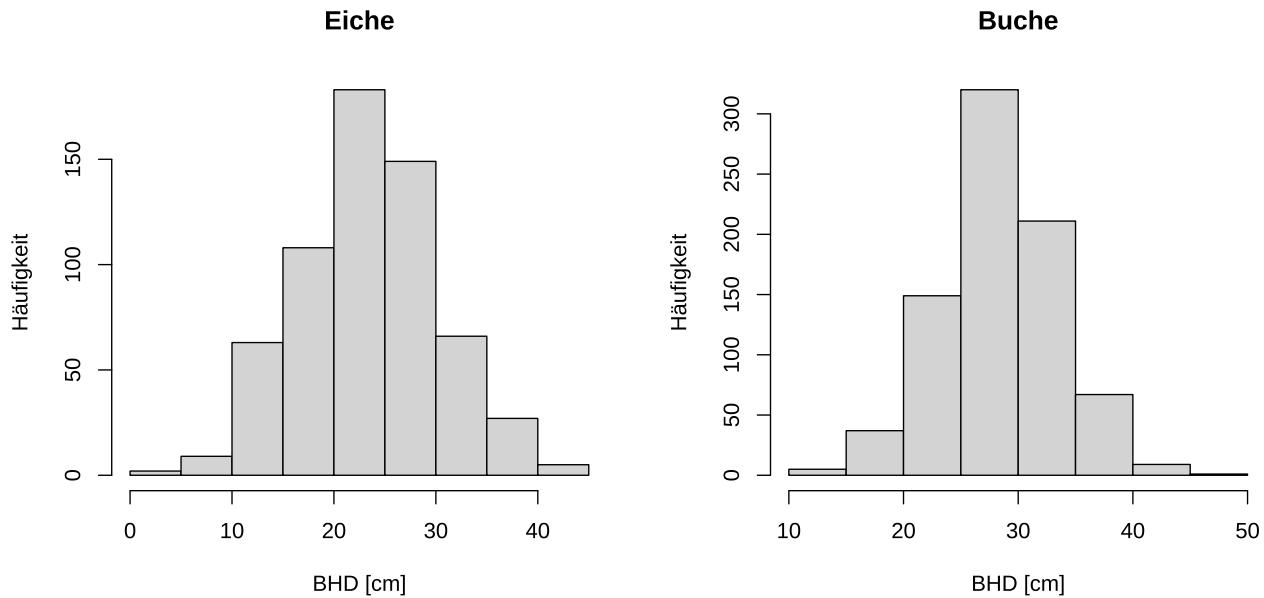


1017

1018 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")
```

1019



1020

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

### 1021 8.3 Boxplots

1022 Oft möchte man die Verteilung einer stetigen Variablen in Abhängigkeit einer diskreten Variable beschreiben  
 1023 oder visualisieren. Ein Beispiel dafür wäre die BHD-Verteilung in Abhängigkeit der Baumarten. Eine häufige  
 1024 Darstellungsform für solche Daten sind *Boxplots*. Sie stellen die durchschnittliche Ausprägung einer stetigen  
 1025 Variable und ihre Schwankung kompakt dar.

1026 Boxplots bestehen aus drei Komponenten:

- 1027 1. Eine *Box*, die den Bereich zwischen 0.25 und 0.75 Percentil abdeckt, diese Distanz wird auch die *IQR*  
 1028 (Interquartile Range), bezeichnet. Zusätzlich wird die Box durch den Median (als dicke horizontale Linie)  
 1029 unterteilt.
- 1030 2. Einzelne Punkte Ausreißer. Als Ausreißer werden Punkte bezeichnet, die  $> 1.5IQR$  vom unteren oder  
 1031 oberen Ende der Box entfernt sind.
- 1032 3. Eine senkrechte Linie von jeder Seite der Box bis zum letzten "Nicht-Ausreißer-Punkt". Also der letzte  
 1033 Punkt, der  $> 1.5IQR$  aber nicht  $> 0.75$  bzw.  $< 0.25$  Percentil ist. Diese Linie wird auch als *Whisker*  
 1034 bezeichnet.

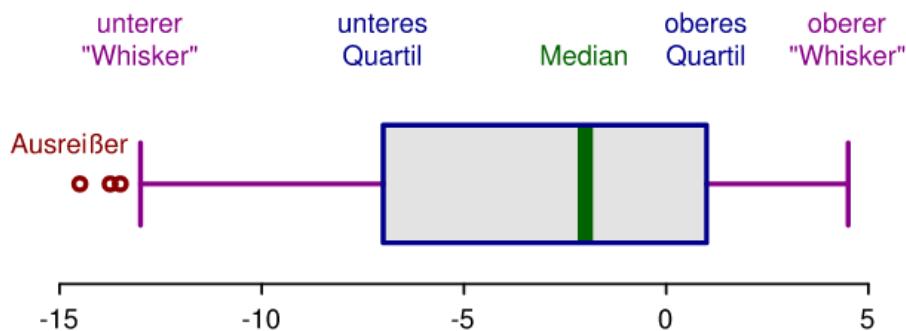
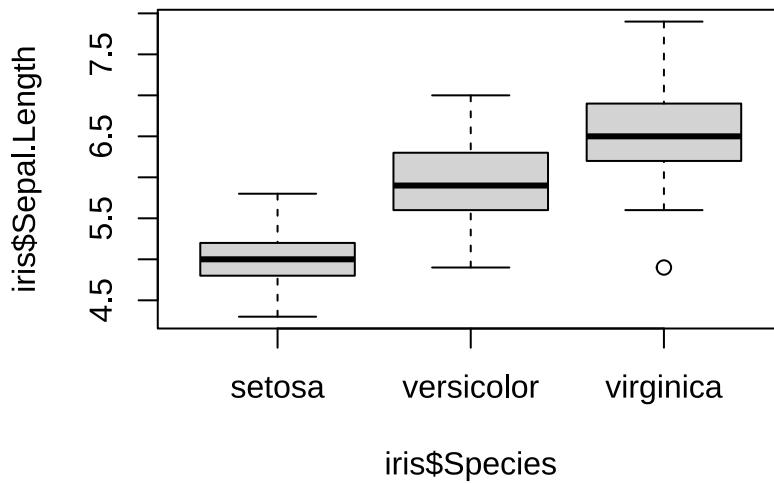


Abbildung 7: Schematische Darstellung eines Boxplots (Quelle: Von RobSeb - Eigenes Werk, CC BY-SA 3.0, <https://commons.wikimedia.org/w/index.php?curid=14697172>).

1035 Mit R kann mit der Funktion `boxplot()` ein Boxplot erstellt werden. Diese Funktion kann in zwei unter-  
 1036 schiedlichen Ausprägungen verwendet werden.

- 1037 1. `boxplot(x)` erzeugt einen Boxplot für die Variable `x`.
- 1038 2. `boxplot(x ~ y)` erzeugt einen oder mehrere Boxplots für `x` aber gruppiert nach `y`, dabei sollte `y` eine  
 1039 kategoriale Variable sein. `x` und `y` können auch die Spaltennamen eines `data.frames` sein, dann muss  
 1040 das `data.frame` mit dem Argument `data` zusätzlich übergeben werden.

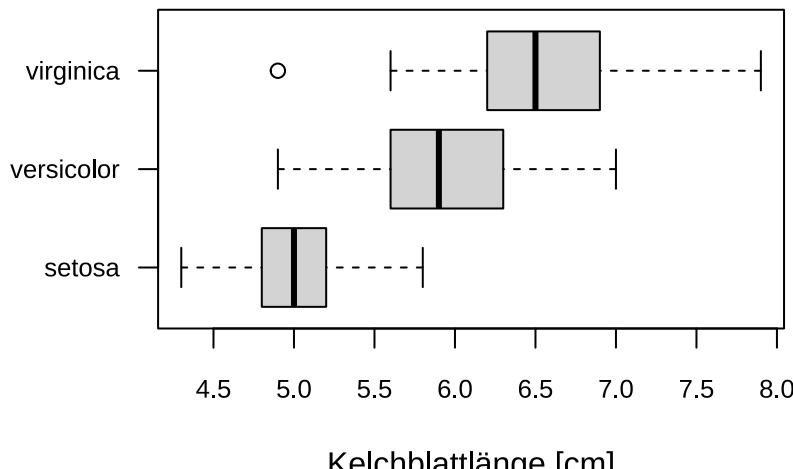
```
boxplot(iris$Sepal.Length ~ iris$Species)
```



1041

1042 Etwas eleganter ist es wenn wir das Argument `data` verwenden und den Plot etwas anpassen. Diese Schreib-  
1043 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  
)
```



1044

1045

### 1046 Aufgabe 18: Boxplots

1047

- 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
- 1051 Erstellen Sie einen Plot mit 3 Subplots, jeweils mit einem Boxplot für die ersten drei Studiengebiete, in dem  
1052 der BHD für jede Baumart dargestellt wird.

## 1053 8.4 ggplot2: Eine Alternative für Abbildungen

1054 ggplot2 ist ein alternatives Plotting-System in R. Sie können mit ggplot2 also grundsätzlich Abbildungen  
 1055 mit dem selben Inhalt erstellen, wie mit Base Plots. Die Syntax und die optische Darstellung unterscheiden  
 1056 sich jedoch grundsätzlich. ggplot2 basiert auf den *grammar of graphics* von Leland Wilkinson. Die Idee  
 1057 ist, alle nötigen Informationen der Abbildung miteinander zu verknüpfen. ggplot2 ist also diametral zu  
 1058 Base Plots. Während Sie in base plot alle Elemente selbst definieren, ist die Idee von ggplot2, dass Sie  
 1059 nur die Daten und den Diagrammtypen übergeben und die Funktion selbst eine schöne Abbildung erstellt.  
 1060 Selbstverständlich können Sie aber auch in ggplot2 viele Einstellungen vornehmen. Im base plot sehen  
 1061 Abbildungen zunächst sehr karg und etwas unfertig aus. Sie müssen viele Einstellungen vornehmen, um eine  
 1062 publizierfähige Grafik zu produzieren. In ggplot2 sollen auch die einfachste Abbildungen schon ästhetisch  
 1063 sein. Mit diesen gebündelten Informationen kann ggplot2 die Abbildung automatisch verschönern. So  
 1064 werden bspw. die Legenden automatisch erzeugt und auch die Formatierungen automatisch an die Datenlage  
 1065 angepasst. ggplot2 nimmt der\*dem Entwickler\*in also Arbeit ab. Dadurch sind die Abbildungen schon ohne  
 1066 viel Nacharbeit schick. Nachteil ist, dass der\*dem Entwickler\*in weniger Möglichkeiten zur Einstellung zur  
 1067 Verfügung stehen und nuterspezifische Sonderwünsche somit schwerer umsetzbar sind. Sehen Sie sich das  
 1068 Cheatsheet zu ggplot2 an. Es ist in RStudio unter [Help > Cheatsheets](#) zu finden.

1069 Bei ggplot2 sind Anweisungen zu den Daten und Anweisungen zur Darstellung voneinander getrennt. Die  
 1070 Daten werden in den Ästhetikbefehl übergeben und dort klassifiziert. Dann folgen die Darstellungsanweisungen.  
 1071 Ähnlich wie bei Base Plots, werden die Grafikelemente ebenenweise nacheinander programmiert, jedoch mit  
 1072 einem + verbunden. Und hier liegt der wesentliche Unterschied zu Base Plots. Durch die + werden die Ebenen  
 1073 zu einem Befehl verbunden und damit gleichzeitig erstellt.

1074 Die Erweiterung wird zunächst geladen<sup>7</sup>. Wir laden außerdem den Datensatz **iris**. Der Datensatz ist in R  
 1075 fest integriert. Siehe `?iris` für mehr Informationen.

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

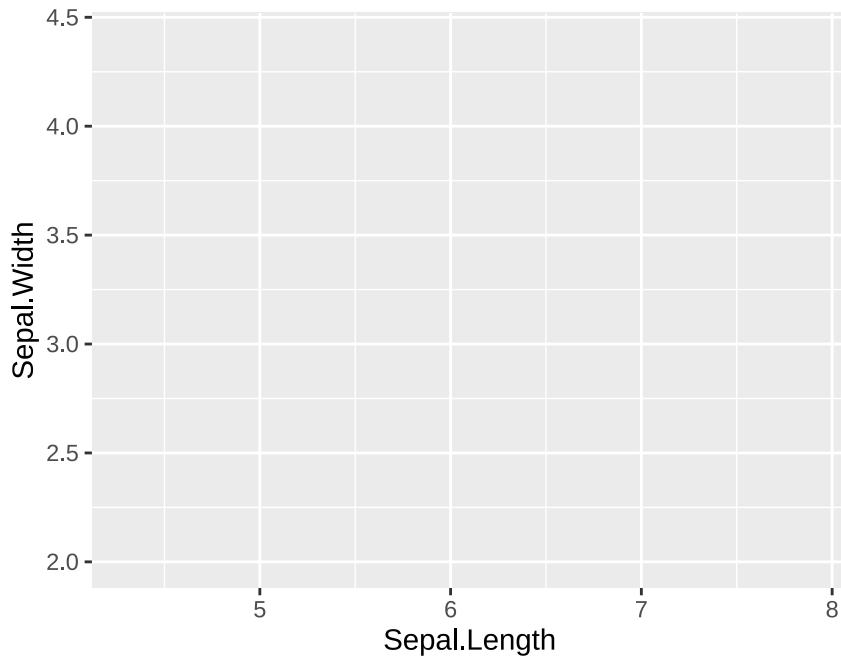
```
1076 ##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
1077 ## 1         5.1      3.5       1.4       0.2  setosa
1078 ## 2         4.9      3.0       1.4       0.2  setosa
1079 ## 3         4.7      3.2       1.3       0.2  setosa
1080 ## 4         4.6      3.1       1.5       0.2  setosa
1081 ## 5         5.0      3.6       1.4       0.2  setosa
1082 ## 6         5.4      3.9       1.7       0.4  setosa
```

1083 Die Ästhetik wird bspw. folgendermaßen definiert.

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

---

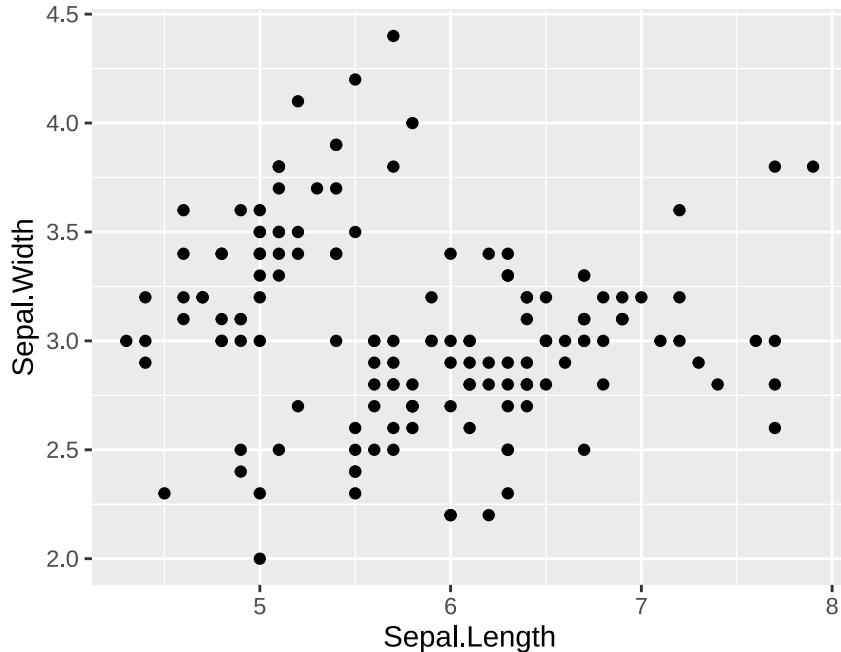
<sup>7</sup>Wir haben bis jetzt immer nur mit base R gearbeitet. D.h. wir haben nur Funktionen verwendet, die R bereits zur Verfügung stellt. Eine der großen Stärken von R sind die Erweiterungen (oder auch Pakete genannt). ggplot2 ist so eine Erweiterung, die einmal mit `install.packages("ggplot2")`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



1084

1085 Dieser Befehl zeichnet noch keine Daten. Die Daten werden lediglich herangezogen, um einen leeren Plot für  
1086 die Daten zu erstellen. In dem Beispiel wird die Variable `Sepal.Length` aus dem `data.frame iris` als x und  
1087 `Sepal.Width` als y Variable definiert. Diese Informationen stehen den folgenden Layern nun zur Verfügung,  
1088 sodass nach den + nur noch x und y verwendet werden müssen. Um bspw. einen Scatterplot zu erstellen  
1089 wird ein `geom_point()` Layer hinzugefügt. x und y werden automatisch an `geom_point` übergeben. Weitere  
1090 Einstellung sind in diesem Beispiel nicht notwendig, wären jedoch möglich. Siehe `?geom_point()`.

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



1091

1092

---

1093 **Aufgabe 19: Abbildungen mit ggplot2**

---

1094

1095 Verwenden Sie die Daten aus Aufgabe 16 und erstellen Sie einen Scatterplot mit ggplot2 wie in Aufgabe 16.

1096

1097 Wir haben mit der Funktion `geom_point()` demm Plot eine Punktgeometrie hinzugefügt. Es gibt noch viele  
1098 weitere Geometrien. Die wichtigsten sind:

- 
- 1099
- `geom_line()` für eine Linie.
  - `geom_histogram()` um ein Histogramm zu erstellen.
  - `geom_boxplot()` um einen Boxplot zu erstellen.
  - `geom_bar()` um ein Säulendiagramm zu erstellen.

---

1100 1101 1102 1103 Welche Geometrie die richtige ist, richtet nach dem Typ der darzustellenden Variablen. Beispielsweise  
1104 bietet sich `geom_point()` an, wenn man zwei kontinuierliche Variable darstellen möchte. Wenn man hin-  
1105 gegen die Verteilung von einer kontinuirlchen Variable darstellen möchte, dann bietet sich ein Histogramm  
1106 (`geom_histogram()`) oder auch eine geschätzte Dichte (z.B. `geom_density()`) an.

1107

---

1108 **Aufgabe 20: Abbildungen mit ggplot2**

---

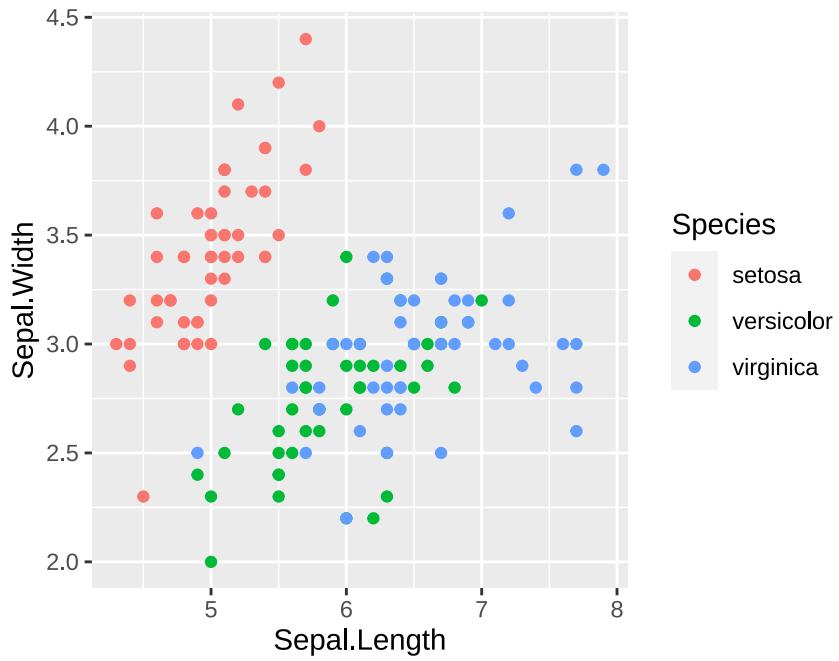
11091110 Verwenden Sie die den Iris Datensatz und erstellen Sie mit ggplot2 einen Plot der die Verteilung der Länge  
1111 der Kelchblätter zeigt (Spalte `Sepal.Length`).

1112

---

1113 1114 1115 1116 Eine der Stärken von ggplot2 ist, dass man den Wert unterschiedlicher Variable auf unterschiedlichen  
Komponenten des Plots abbilden kann. Wir haben bis jetzt ein bzw. zwei Variable auf der x- und y-Achse  
abgebildet. Wir können aber ein weitere Variablen verwenden um das Aussehen des Plots zu beeinflussen.  
Beispielsweise können wir die Farbe der Punkte (für `geom_point()`) mit dem Argument `col` beeinflussen.

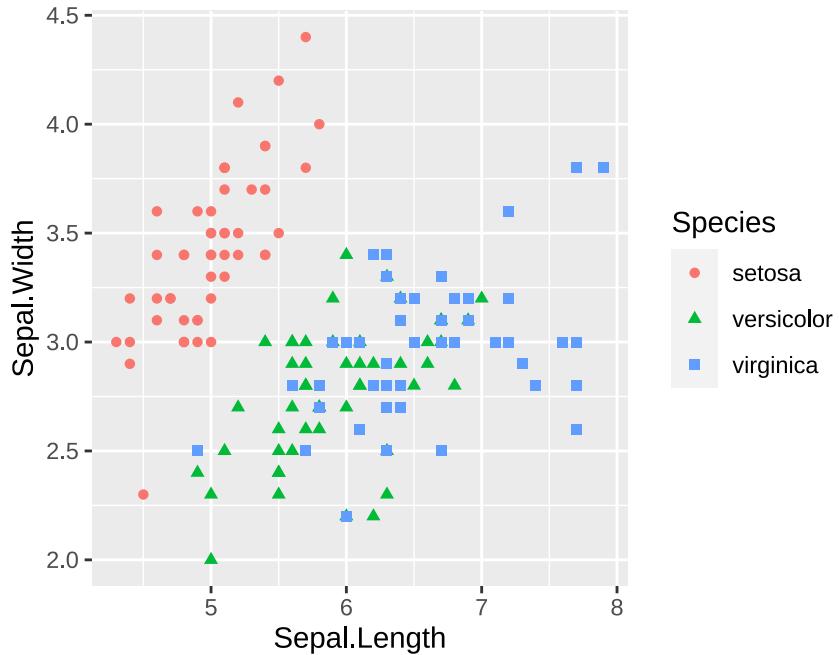
```
ggplot(iris, aes(x = Sepal.Length, y = Sepal.Width, col = Species)) +  
  geom_point()
```



1117

- 1118 Somit bekommt jede Irisart eine eigene Farbe<sup>8</sup>. Gleichesmaßen können wir die Punktart (**shape**), die  
1119 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()
```

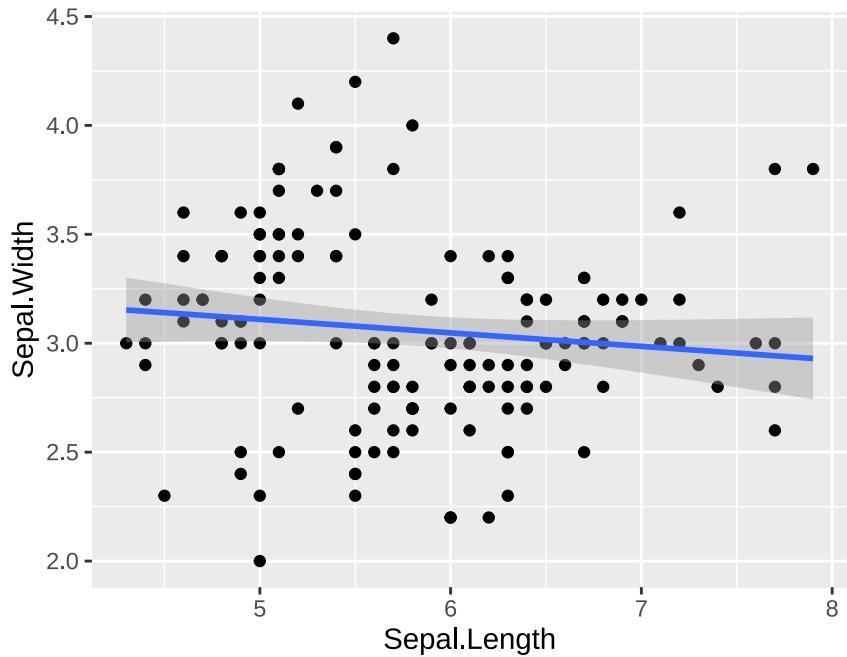


1120

- 1121 In dem Plot ist die Information zu der Art redundant (einmal als Farbe und einmal Symbolart).  
1122 Ein weitere sehr nützliche Geometrie ist **geom\_smooth()**, die es erlaubt eine Trendlinie hinzuzufügen.

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

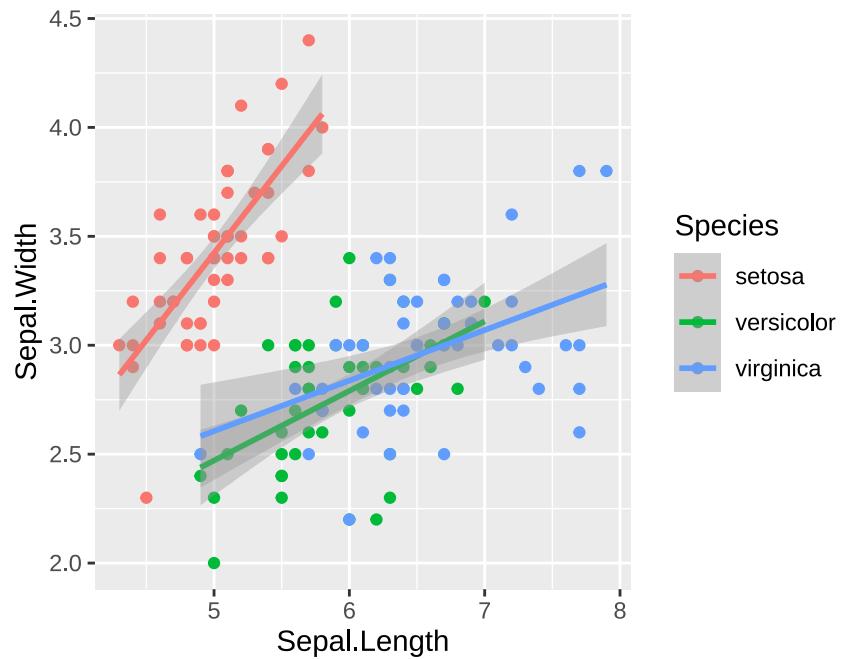
```
ggplot(iris, aes(x = Sepal.Length, y = Sepal.Width)) +
  geom_point() + geom_smooth(method = "lm")
```



1123

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")
```



1127

1128

1129 **Aufgabe 21: Anpassen von Plots**  
1130

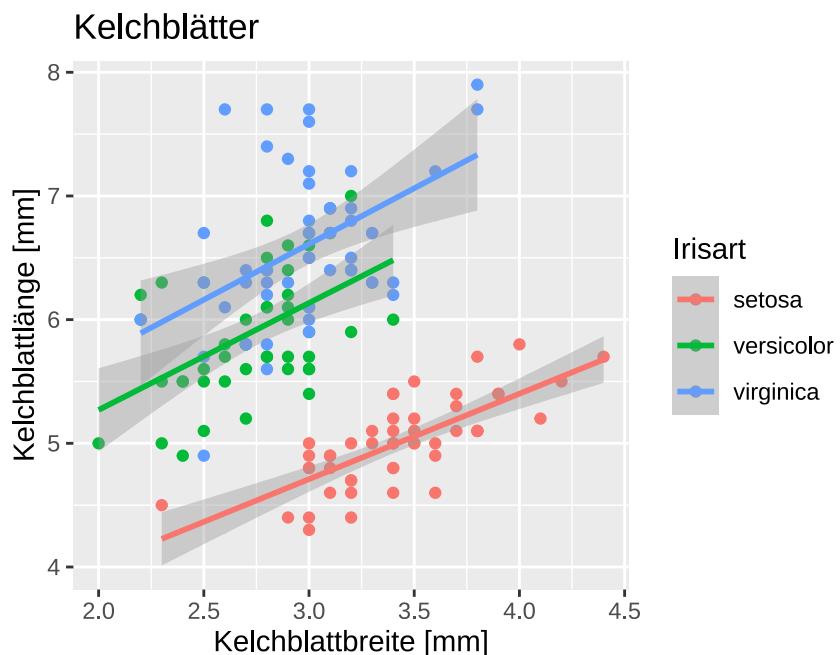
- 1131 Lesen Sie den Datensatz `data/bhd_1.txt` ein und erstellen Sie einen Boxplot für die Verteilung des BHDs  
 1132 für jede Baumart. In einem zweiten Schritt verwenden Sie erst `col = gebiet` und dann `fill = gebiet`.  
 1133 Welchen Unterschied stellen Sie fest?

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

1134

- 1135 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")
```



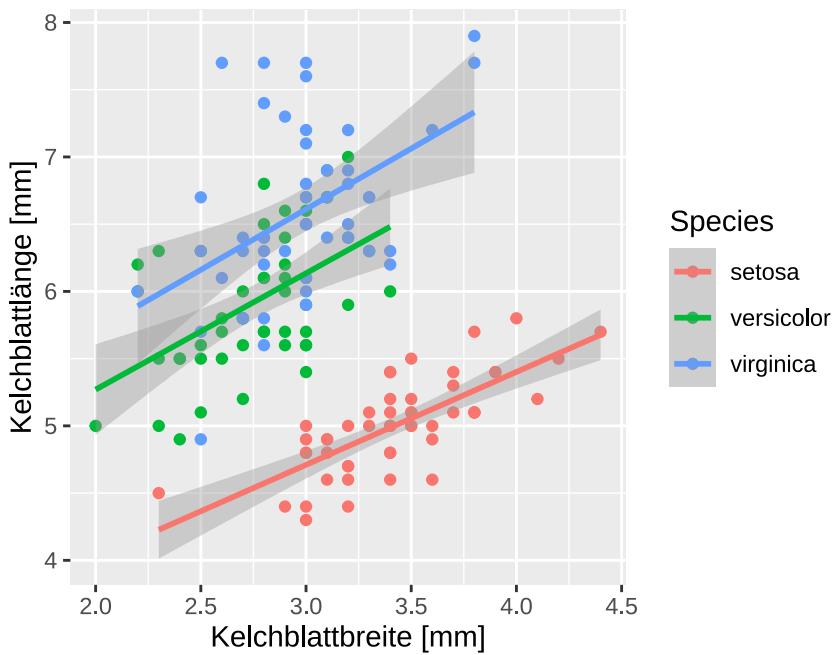
1136

- 1137 Statt einen langen Befehl zu tippen, kann ein `ggplot()` auch zwischengespeichert und wieder aufgerufen bzw.  
 1138 angepasst werden. Das ist vor allem sinnvoll, wenn mehrere Abbildungen auf dem selben Zwischenergebnis  
 1139 aufbauen sollen.

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

- 1140 Wir können jetzt mit `p1` weiter arbeiten und beispielsweise eine Beschriftung hinzufügen.

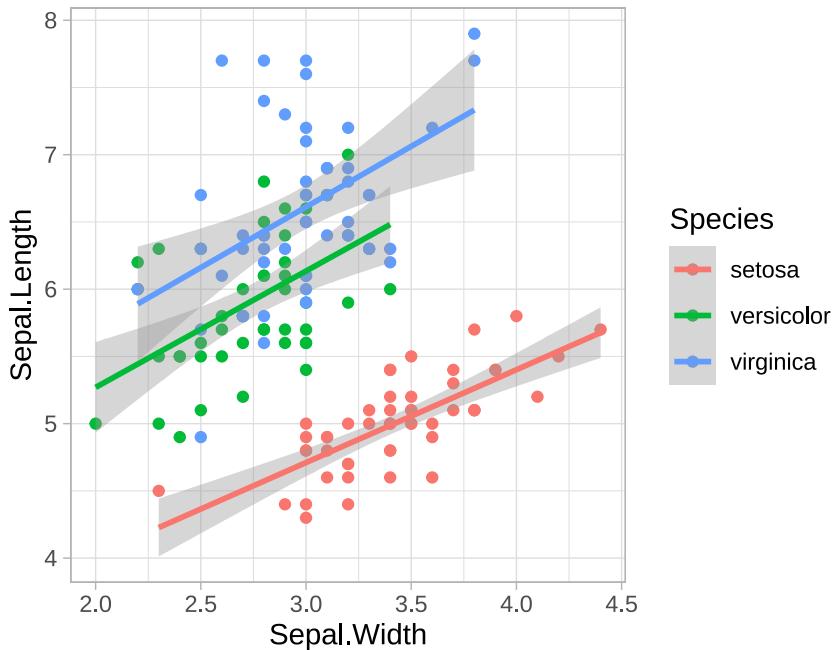
```
p1 + labs(x = "Kelchblattbreite [mm]", y = "Kelchblattlänge [mm]")
```



1141

1142 Oder auch den ganzen Plot anpassen. Dafür gibt es *themes*. Es gibt eine Reihe von vorgefertigten *themes*  
1143 oder man kann diese auch selber erstellen (das ist aber nicht Teil dieses Kurses).

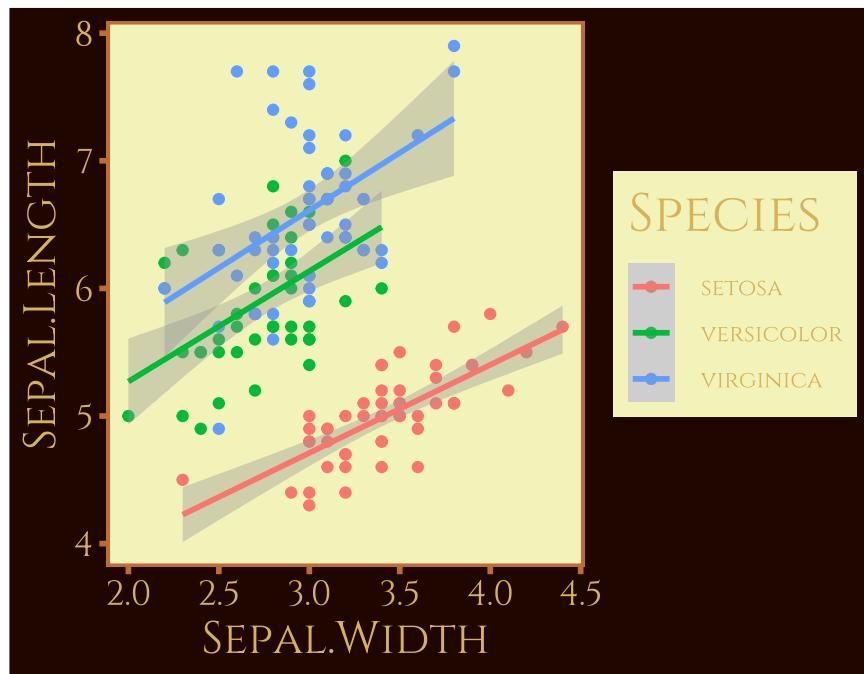
```
p1 + theme_light()
```



1144

1145 Weitere *themes* sind: `theme_bw()`, `theme_linedraw()` oder `theme_dark()`. Es gibt extra Pakete die viele  
1146 zusätzliche weitere *themes* anbieten. Dazu gehört z.B. das Paket `ggthemes` oder `ThemePark`. Während  
1147 `ggthemes` hauptsächlich durchdachte Grafikkonzepte liefert, sind die Themes aus `ThemePark` eher Popkultur  
1148 und nicht 100 %ig ernst gemeint.

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

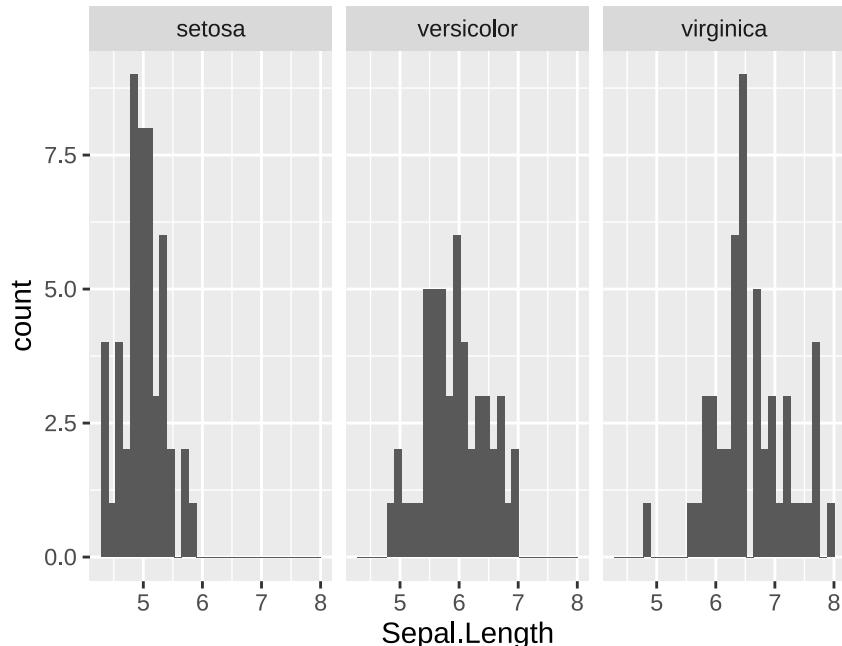


1149

#### 8.4.1 Multipanel Abbildungen

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

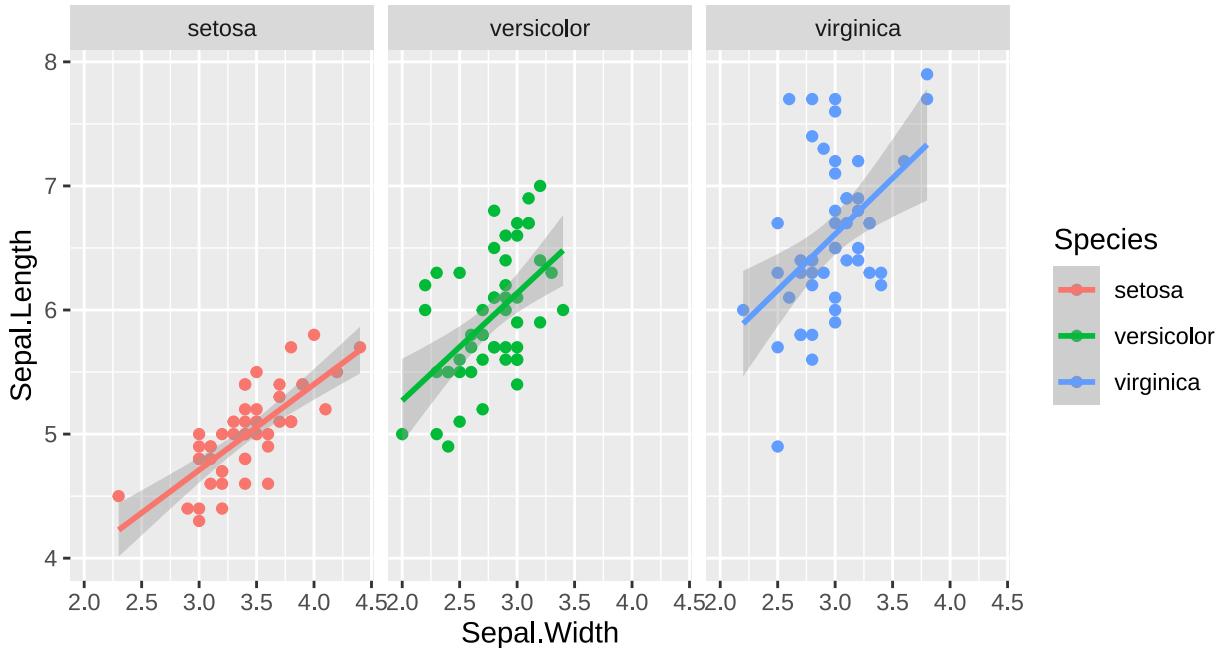
```
ggplot(iris, aes(Sepal.Length)) + geom_histogram() +
  facet_grid(~ Species)
```



1154

1155 Die Funktion `facet_grid()` erzeugt einen *Grid* und beschriftet die Grid-Zeilen und -Spalten, während  
 1156 `facet_wrap()` für jedes Panel (Diagramm) eine eigene Überschrift erzeugt. Beim Arrangieren der Diagramme  
 1157 wird der Vorteil von `ggplot2` gegenüber `base plot` besonders deutlich. Während es im `base plot` System  
 1158 mühsam ist, die Abbildungen zu arrangieren, übernimmt `ggplot2` das Arrangement automatisch und fügt  
 1159 sogar eine automatisch Legende hinzu. Die Achsenabschnitte werden so angepasst, dass die Daten vergleichbar  
 1160 sind.

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



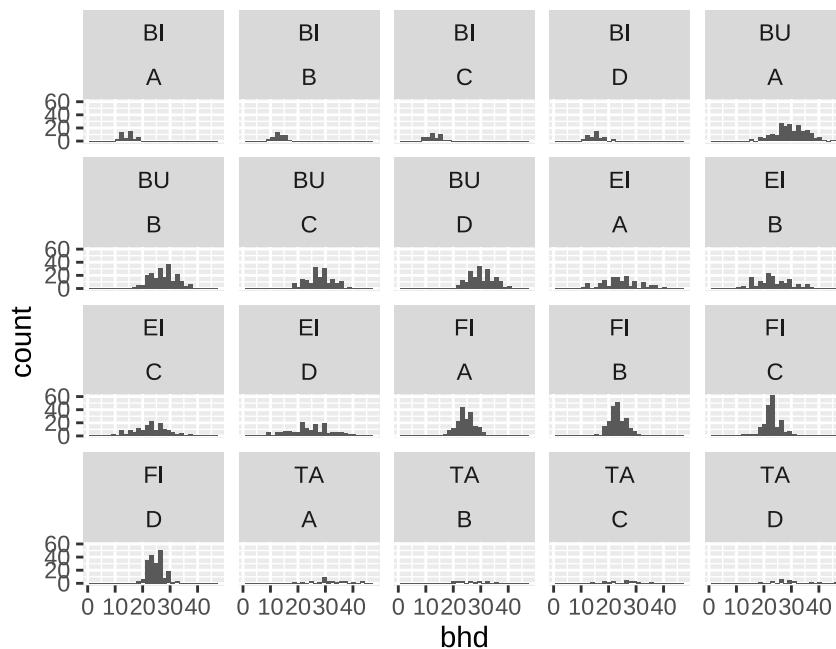
1161

1162

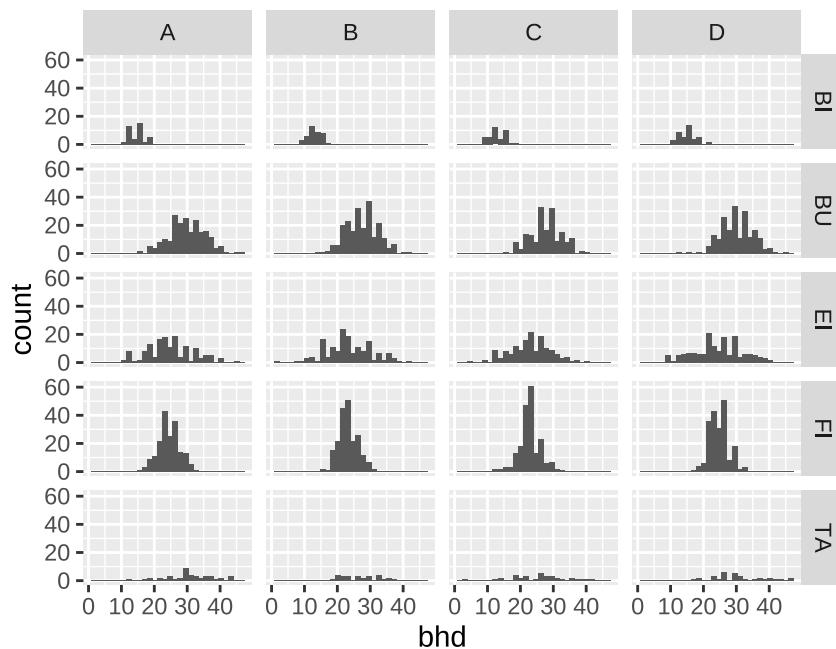
### 1163 Aufgabe 22: Multipanel Abbildungen

---

1165 Lesen Sie erneut den Datensatz `daten/bhd_1.txt` ein und speichern Sie diesen in die Variable (`dat_bhd`).  
 1166 Erstellen Sie für jede Art und Gebiet ein Histogramm. Welche Unterschiede können Sie feststellen, wenn Sie  
 1167 `facet_grid()` oder `facet_wrap()` verwenden?



1168



1169

#### 1170 8.4.2 Plots kombinieren

1171 Es gibt Situationen, in denen **unterschiedliche** Plots miteinander kombiniert werden müssen. Im vorherigen  
 1172 Abschnitt wurde dies immer anhand einer gruppierenden Kovariate gemacht. Aber es gibt auch Situationen, in  
 1173 denen das nicht möglich ist. Beispielsweise wenn ein Histogramm und ein Scatterplot vom gleichen Datensatz  
 1174 zusammengefasst werden sollen. Dafür bietet sich das Paket **patchwork** an<sup>9</sup>.

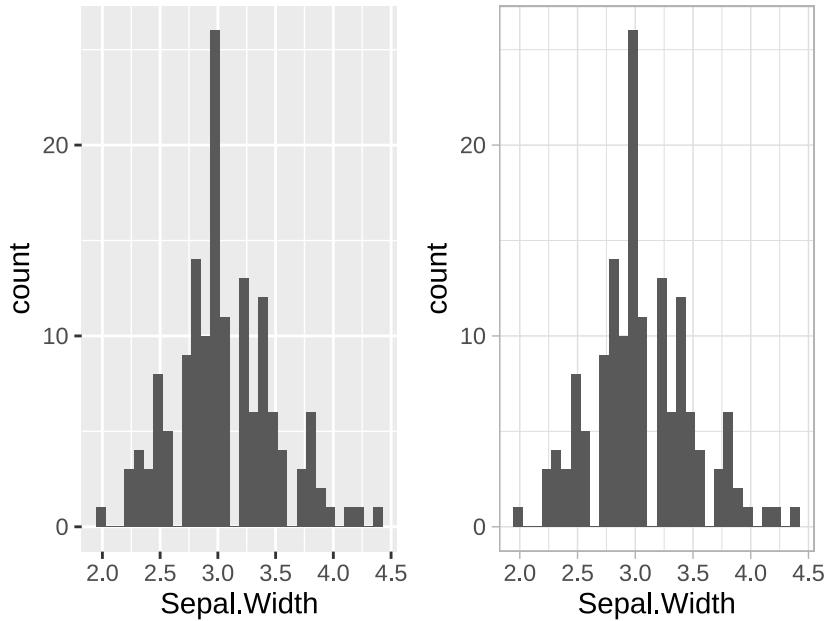
1175 Als erstes können wir zwei (oder natürlich auch mehrere Plots) erstellen. Hier unterscheiden sich die Plots  
 1176 lediglich durch das Aussehen.

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

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

1177 Dann müssen können wir diese Plots ebenfalls mit `+` zusammenfügen.

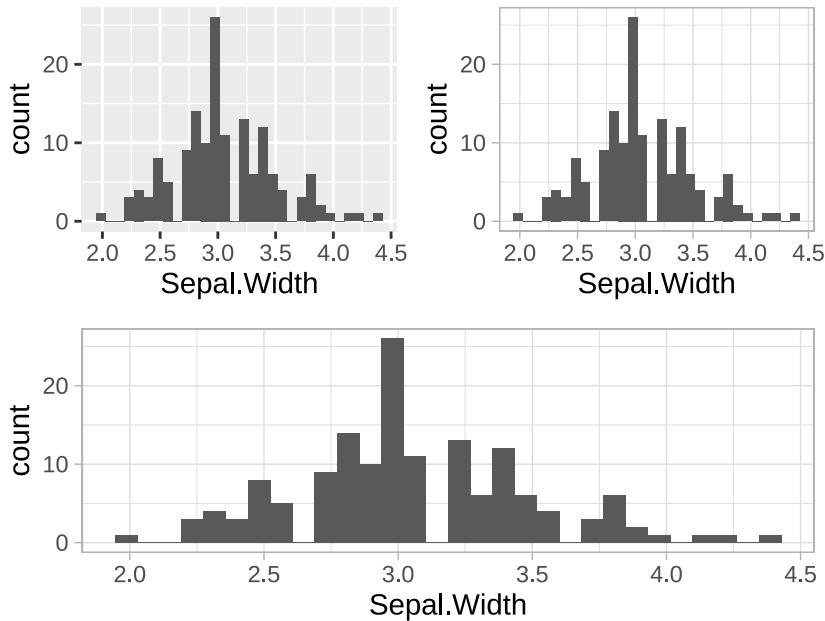
```
library(patchwork)
p1 + p2
```



1178

1179 Natürlich können auch weitere Plots hinzugefügt werden (auch in unterschiedlichen Dimensionen):

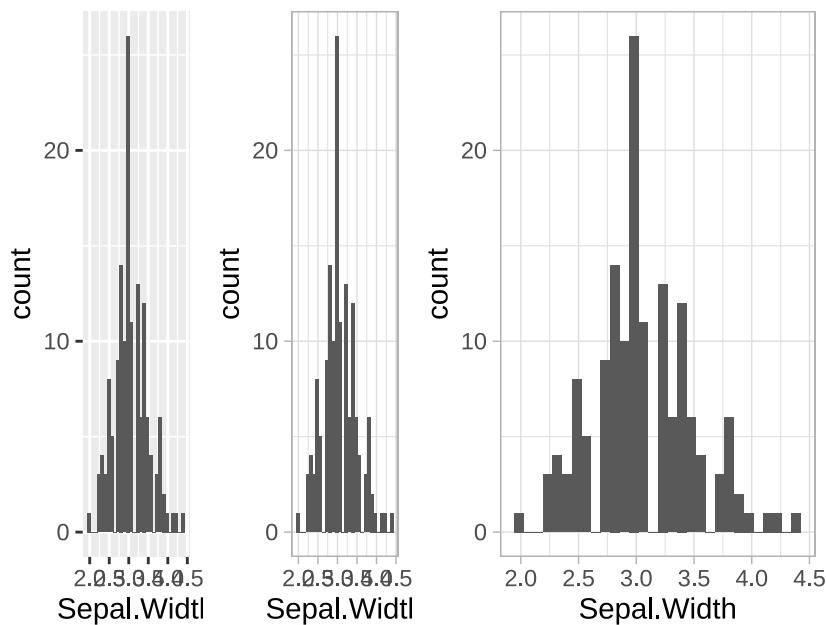
```
(p1 + p2) / p2
```



1180

1181 Des weiteren können mit `|` auch Plots gegenüber gestellt werden.

(p1 + p2) | p2

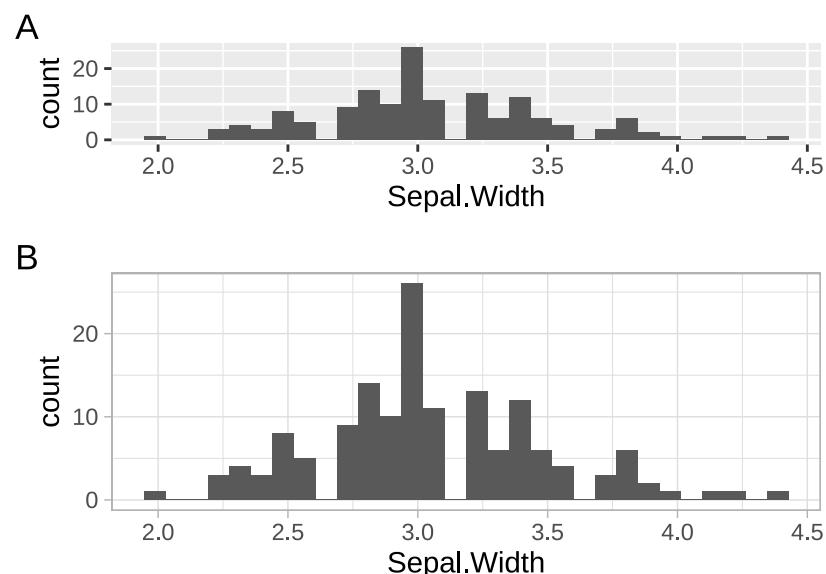


1182

1183 Weitere Optionen können mit `plot_layout()` und `plot_annotation()` angepasst werden. Mit  
1184 `plot_layout()` kann die Anordnung der Plots bestimmt werden (z.B. über die Argumente `nrow`  
1185 und `ncol`), sowie deren relative Größe (über die Argumente `widths` und `heights`). Mit der Funktion  
1186 `plot_annotation()` können zusätzliche Beschriftungen hinzugefügt werden, wie beispielsweise eine Titel  
1187 (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



1188

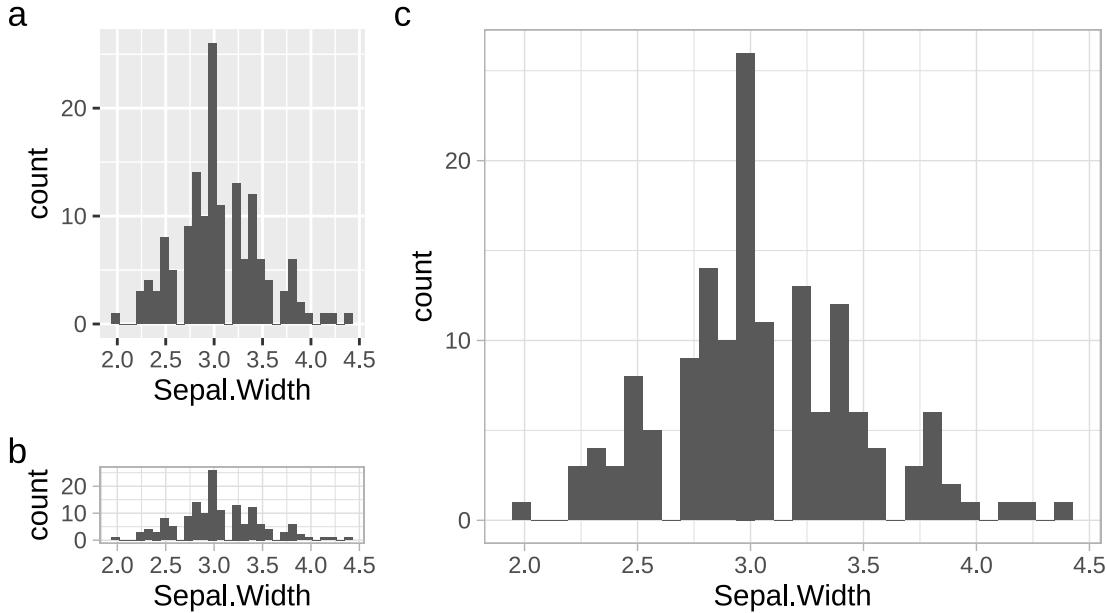
1189

---

1190 **Aufgabe 23: Mehrere Plots zusammenfügen**  
1191

---

1192 Versuchen Sie die folgende Zusammenstellung der Plots nachzumachen:



1193

1194 **8.4.3 Speichern von plots**

1195 Sie können mit `ggsave()` eine zwischen-gespeicherte Abbildung exportieren, indem Sie den Variablennamen  
1196 übergeben. Wenn Sie keine Variable übergeben, wird automatisch die letzte Abbildung gespeichert. Das  
1197 Dateiformat wird aus dem Dateinamen übernommen. Die Größe der Inhalte können Sie sehr leicht mit den  
1198 Argumenten `width` und `height` oder `scale` gesteuert werden.

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

## 1199 9 Mit Daten arbeiten

### 1200 9.1 dplyr eine Einführung

1201 `dplyr` ist eine Erweiterung von R (ein Paket), die das Ziel hat, den Umgang mit Daten einfacher und  
1202 schneller zu machen.

1203 `dplyr` definiert 5 Verben, um mit Daten zu arbeiten. Diese sind:

- 1204 • `filter`
- 1205 • `select`
- 1206 • `arrange`
- 1207 • `mutate`
- 1208 • `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))
```

1209 Damit die Funktionen aus `dplyr` verwendet werden können, müssen wir als erstes das Paket `dplyr` laden.  
1210 The online Resource [R for Data Science](#) gibt einen sehr guten, tieferen Eindruck in die Arbeit mit `dplyr`  
1211 und erklärt auch die `dplyr` Philosophie noch Mal sehr genau für Beginner.

```
library(dplyr)
```

1212 Sollte dies zu einer Fehlermeldung führen, dann müssen Sie das Paket `dplyr` erst installieren. Dafür müssen  
1213 Sie `einmalig install.packages("dplyr")` installieren.

1214 `dplyr` stellt unterschiedliche Funktionen zum Arbeiten mit Daten zur Verfügung. Es gibt fünf Grundfunktionen  
1215 für die am häufigsten vorkommenden Operationen. Mit der Funktion `filter()` können unterschiedliche  
1216 Beobachtungen gefiltert werden:

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

```
1217 ##   id plot bhd alter
1218 ## 1   1    1  50   10
1219 ## 2   2    1  29   30
1220 ## 3   3    2  13   31
1221 ## 4   4    2  23   24
1222 ## 5   5    3  25   25
```

1223 Es können auch mehrere Spalten verwendet werden.

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

```
1224 ##   id plot bhd alter
1225 ## 1   2    1  29   30
1226 ## 2   3    2  13   31
1227 ## 3   4    2  23   24
```

```
1228 ## 4 5 3 25 25
```

1229 Natürlich kann genau das gleiche Ergebnis mit dem ‘normalen’ R erreicht werden, dies wäre dann:

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

```
1230 ## id plot bhd alter
```

```
1231 ## 2 2 1 29 30
```

```
1232 ## 3 3 2 13 31
```

```
1233 ## 4 4 2 23 24
```

```
1234 ## 5 5 3 25 25
```

1235 Eine weitere Funktion aus dem Paket **dplyr** ist **select()**. Damit können Spalten aus einem **data.frame**

1236 ausgewählt werden. Dabei können auch die Spaltennamen unbenannt werden.

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

```
1237 ## bhd
```

```
1238 ## 1 50
```

```
1239 ## 2 29
```

```
1240 ## 3 13
```

```
1241 ## 4 23
```

```
1242 ## 5 25
```

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

```
1243 ## bhd id
```

```
1244 ## 1 50 1
```

```
1245 ## 2 29 2
```

```
1246 ## 3 13 3
```

```
1247 ## 4 23 4
```

```
1248 ## 5 25 5
```

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

```
1249 ## BHD id
```

```
1250 ## 1 50 1
```

```
1251 ## 2 29 2
```

```
1252 ## 3 13 3
```

```
1253 ## 4 23 4
```

```
1254 ## 5 25 5
```

1255 Mit der Funktion **arrange()** können die Beobachtungen in einem **data.frame** sortiert werden.

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

```
1256 ## id plot bhd alter
```

```
1257 ## 1 3 2 13 31
```

```
1258 ## 2 4 2 23 24
```

```
1259 ## 3 5 3 25 25
```

```
1260 ## 4 2 1 29 30
1261 ## 5 1 1 50 10
```

1262 Mit der Funktion `desc()` kann die Anordnung in absteigender Reihenfolge sortiert werden.

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

```
1263 ## id plot bhd alter
1264 ## 1 1 1 50 10
1265 ## 2 2 1 29 30
1266 ## 3 5 3 25 25
1267 ## 4 4 2 23 24
1268 ## 5 3 2 13 31
```

1269 Mit der Funktion `mutate()` kann man eine neue Spalte hinzufügen.

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

```
1270 ## id plot bhd alter bhd_mm fl
1271 ## 1 1 1 50 10 500 1963.4954
1272 ## 2 2 1 29 30 290 660.5199
1273 ## 3 3 2 13 31 130 132.7323
1274 ## 4 4 2 23 24 230 415.4756
1275 ## 5 5 3 25 25 250 490.8739
```

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

```
1276 ## id plot bhd alter mean_bhd
1277 ## 1 1 1 50 10 28
1278 ## 2 2 1 29 30 28
1279 ## 3 3 2 13 31 28
1280 ## 4 4 2 23 24 28
1281 ## 5 5 3 25 25 28
```

1282 Mit der Funktion `summarise()` können Spalten zusammengefasst werden.

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

```
1283 ## mean_bhd mean_sd
1284 ## 1 28 13.63818
```

1285

1286 **Aufgabe 24: Datenmanipulation mit dplyr**

---

- 1288 1. Laden Sie den Datensatz `data/bhd_1.txt`
- 1289 2. Berechnen Sie folgende Werte für alle Einträge und speichern Sie die Ergebnisse in `erg1`
- 1290 • mittlerer `bhd`
- 1291 • maximales `alter`
- 1292 • die Standardabweichung des BHDs
- 1293 • die Anzahl Bäume mit einem BHD > 30

1294 **9.2 Arbeiten mit gruppierten Daten**

1295 Zusätzlich können `mutate` und `summarise` auch auf gruppierte Daten angewendet werden. Dafür müssen  
 1296 wir erst die Funktion `group_by()` aufrufen und als Argumente die Variablen übergeben, die die Gruppen  
 1297 definieren.

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

1298 ##   id plot bhd alter bhd_m
1299 ## 1 1    1 50    10    28
1300 ## 2 2    1 29    30    28
1301 ## 3 3    2 13    31    28
1302 ## 4 4    2 23    24    28
1303 ## 5 5    3 25    25    28

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

1304 ## # A tibble: 5 x 5
1305 ## # Groups:   plot [3]
1306 ##       id   plot   bhd   alter   bhd_m
1307 ##     <int> <dbl> <dbl> <dbl> <dbl>
1308 ## 1     1     1    50     10    39.5
1309 ## 2     2     1    29     30    39.5
1310 ## 3     3     2    13     31    18
1311 ## 4     4     2    23     24    18
1312 ## 5     5     3    25     25    25

summarise(dat, bhd_m = mean(bhd))

1313 ##   bhd_m
1314 ## 1    28

summarise(dat1, bhd_m = mean(bhd))

1315 ## # A tibble: 3 x 2
1316 ##   plot   bhd_m
```

```
1317 ## <dbl> <dbl>
1318 ## 1      1  39.5
1319 ## 2      2  18
1320 ## 3      3  25
```

1321

1322 **Aufgabe 25: dplyr mit gruppierten Daten**

---

 1323

- 1324 1. Laden Sie den Datensatz `data/bhd_1.txt`
- 1325 2. Berechnen Sie für jede Baumart und Standort die folgenden Werte:
  - 1326 • mittlerer `bhd`
  - 1327 • maximales `alter`
  - 1328 • die Standardabweichung des BHDs
  - 1329 • die Anzahl Bäume mit einem BHD > 30
- 1330 3. Ordnen Sie das Ergebnis nach Baumart und BHD.

1331 **9.3 pipes oder %>%**

1332 Mit *Pipes* (`%>%`) kann man das Ergebnis einer Funktion einfach an eine nachfolgende Funktion weiterreichen.

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

1333 Wir kennen bis jetzt:

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

1334 ## [1] 3.333333

1335 Mit *Pipes*, die durch das Symbol `%>%` dargestellt werden<sup>10</sup>, können wir das etwas vereinfachen und nacheinander  
1336 schreiben:

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

1337 ## [1] 3.333333

1338 Oder sogar

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

1339 ## [1] 3.333333

1340

1341 **Aufgabe 26: Pipes %>%**

---

 1342

1343 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.

- 1344 1. Laden Sie den Datensatz `data/bhd_1.txt`.
- 1345 2. Berechnen Sie für jede Baumart und Standort die folgenden Werte:
- 1346 • mittlerer `bhd`
- 1347 • maximales `alter`
- 1348 • die Standardabweichung des BHDs
- 1349 • die Anzahl Bäume mit einem BHD > 30
- 1350 3. Ordnen Sie das Ergebnis nach Baumart und BHD.

## 1351 9.4 Joins

1352 Eine weitere häufige Aufgabe beim Daten Management ist es Daten zusammenzuführen. Nehmen Sie an, dass  
1353 wir folgende Aufnahmen gemacht haben

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

1354 und jeder Baum lediglich mit einer `id` versehen wurde. In einer zweiten Tabelle wurden dann weitere Daten  
1355 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")
)
```

1356 Ziel ist es jetzt die Bäume aus `aufnahmen` mit den Informationen aus den `metadaten` zu verbinden. Dazu  
1357 dient `id` als Bindeglied (oft auch Schlüssel genannt).

1358 Dazu gibt es vier Möglichkeiten.

1359 Zur Durchführung gibt es in base R die Funktion `merge()`. Wir werden aber gleich die Funktionen aus dem  
1360 Paket `dplyr` verwenden.

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

1361 ##   id bhd  art gebiet
1362 ## 1   1   20 <NA>   <NA>
1363 ## 2   2   31    Ta      A
1364 ## 3   3   74    Bu      B

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

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

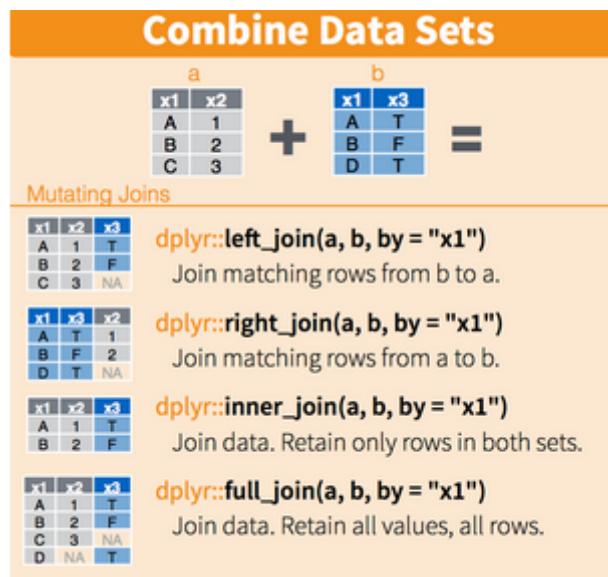


Abbildung 8: Joins (Quelle Rstudio)

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

```
1369 ##   id bhd art gebiet
1370 ## 1 2 31 Ta     A
1371 ## 2 3 74 Bu     B
full_join(aufnahmen, metadaten, by = "id")
```

```
1372 ##   id bhd art gebiet
1373 ## 1 1 20 <NA> <NA>
1374 ## 2 2 31 Ta     A
1375 ## 3 3 74 Bu     B
1376 ## 4 4 NA Bu     B
```

1377 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"))
```

```
1378 ##   id bhd art gebiet
1379 ## 1 1 20 <NA> <NA>
1380 ## 2 2 31 Ta     A
1381 ## 3 3 74 Bu     B
```

1382

1383 **Aufgabe 27: Verbinden von Daten**  
1384

- 1385 • Lesen Sie die Datensätze `daten/bhd_2.txt` und `daten/bhd_2_meta.txt` ein.  
 1386 • Stellen Sie sicher, dass es für den `bhd` keine fehlenden Werte gibt (entfernen sie entsprechende Zeilen)  
 1387 • Fügen Sie zu den Metadaten (gespeichert in `bhd_2_meta`) die Anzahl Bäume und den mittleren `bhd`  
 1388 hinzu pro Gebiet.

1389 **9.5 ‘long’ and ‘wide’ Datenformate**

1390 Unter anderem Wickham (2014) empfieilt das Prinzip von *tidy Data*. Nach diesem Prinzip sollten Daten wie  
 1391 folgt organisiert sein:

- 1392 • Jede Zeile ist ein Merkmalsträger/Subjekt/Objekt (z. B. eine Person oder ein Baum).  
 1393 • Jede Spalte ist eine Variable (ein Merkmal), die den Merkmalsträger beschreibt.  
 1394 • In jeder Zelle ist genau ein Wert (Merkmalausprägung), nämlich der Wert, der Variable für den Merk-  
 1395 malsträger.

1396 Zum Beispiel enthalten Spaltennamen oft Informationen, die eigentlich in einer Variable gespeichert werden  
 1397 sollten. Folgendes Beispiel gibt die BHD Messung von 3 Bäumen in 3 Jahren wieder. Der Vorteil von *tidy*  
 1398 Daten ist, dass viele Funktionen diese Form der Datenorganisation erwarten. Somit müssen sie Ihre Daten nur  
 1399 ein Mal organisieren und können fast alle Analysen durchführen

```
dat <- tibble(  
  id = 1:3,  
  bhd2015 = c(30, 31, 32),  
  bhd2026 = c(31, 31, 33),  
  bhd2017 = c(34, 32, 33)  
)
```

1400 Der `tibble` aus dem Paket `tibble` ist ein moderner Data Frame, der sich in seinen Funktionen in das `tidy`  
 1401 Universum einfügt. Wir können auch einfach das Paket `tidyverse` laden, um alle Pakete, die Teil des `tidy`  
 1402 Universums sind gleichzeitig zu laden. Der `tibble` beinhaltet alle Funktionen, die der Data Frame auch  
 1403 beinhaltet und kann auch für genau die gleichen Zwecke verwendet werden. Der `tibble` hat u. A. eine  
 1404 modernere Darstellung im Konsolenoutput.

1405 Diese Daten sind jetzt im `wide`-Format gespeichert und nicht `tidy`, weil Information über die Daten (nämlich  
 1406 das Jahr der Aufnahme in den Spaltennamen gespeichert sind). Besser wäre eine Struktur mit nur drei Spalten:  
 1407 `id`, `jahr` und `bhd`. Um die Daten in so eine Struktur zu bringen, gibt es die Funktion `pivot_longer()` aus  
 1408 dem Paket `tidyverse`.

```
library(tidyverse)  
dat1 <- pivot_longer(dat, cols = bhd2015 : bhd2017)  
dat1  
  
## # A tibble: 9 x 3  
##       id   name   value
```

```

1411 ##   <int> <chr>    <dbl>
1412 ## 1     1 bhd2015    30
1413 ## 2     1 bhd2026    31
1414 ## 3     1 bhd2017    34
1415 ## 4     2 bhd2015    31
1416 ## 5     2 bhd2026    31
1417 ## 6     2 bhd2017    32
1418 ## 7     3 bhd2015    32
1419 ## 8     3 bhd2026    33
1420 ## 9     3 bhd2017    33

```

1421 Wenn wir die Spalten für die Variable und den Wert gleich sinnvoll benennen möchten, können wir das über  
 1422 die Argumente `names_to` und `value_to` machen.

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

```

1423 ## # A tibble: 9 x 3
1424 ##       id jahr     bhd
1425 ##   <int> <chr>    <dbl>
1426 ## 1     1 bhd2015    30
1427 ## 2     1 bhd2026    31
1428 ## 3     1 bhd2017    34
1429 ## 4     2 bhd2015    31
1430 ## 5     2 bhd2026    31
1431 ## 6     2 bhd2017    32
1432 ## 7     3 bhd2015    32
1433 ## 8     3 bhd2026    33
1434 ## 9     3 bhd2017    33

```

1435 Analog zu der Funktion `pivot_longer()` gibt es auch die Funktion `pivot_wider()`, um vom Daten vom  
 1436 `long`-Format ins `wide`-Format zu transformieren.

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

```

1437 ## # A tibble: 3 x 4
1438 ##       id bhd2015 bhd2026 bhd2017
1439 ##   <int>    <dbl>    <dbl>    <dbl>
1440 ## 1     1        30        31        34
1441 ## 2     2        31        31        32
1442 ## 3     3        32        33        33

```

1443

---

1444 **Aufgabe 28: Zeitliche Verlauf von BHDs**

---

1446 In der Datei `bhd_3.csv` befinden sich gemessene BHDs (in cm) von unterschiedlichen Bäumen zu unter-  
 1447 schiedlichen Zeitpunkten. Erstellen Sie ein Liniendiagramm, das den zeitlichen (x-Achse) Verlauf der BHDs  
 1448 (y-Achse) für die unterschiedlichen Bäume darstellt.

1449 **9.6 Auswählen von Variablen**

1450 Sobald die Datensätze etwas umfangreicher werden (d.h. es gibt mehrere Spalten in einem `data.frame`),  
 1451 können innerhalb vieler `dplyr`-Funktionen spezielle Funktionen verwendet werden, um Variablen auszuwählen.  
 1452 Wenn die genaue Position der Spalten bekannt ist, kann man mit dem `:`-Operator und der Position Spalten  
 1453 auswählen:

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

1454 ## Sepal.Length Sepal.Width Petal.Length  
 1455 ## 1 5.1 3.5 1.4  
 1456 ## 2 4.9 3.0 1.4  
 1457 ## 3 4.7 3.2 1.3

1458 Diese Vorgehensweise kann gehährlich sein, da sich manchmal Spalten verschieben und sich somit die  
 1459 Positionen ändern. Eist besser Spalten immer explizit anzusprechen.

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

1460 ## Sepal.Length Sepal.Width Petal.Length  
 1461 ## 1 5.1 3.5 1.4  
 1462 ## 2 4.9 3.0 1.4  
 1463 ## 3 4.7 3.2 1.3

1464 `select()` erlaubt es, auch hier den `:`-Operator zu verwenden:

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

1465 ## Sepal.Length Sepal.Width Petal.Length  
 1466 ## 1 5.1 3.5 1.4  
 1467 ## 2 4.9 3.0 1.4  
 1468 ## 3 4.7 3.2 1.3

1469 Es gibt auch einige spezielle Funktionen, um Spalten innerhalb eines `select()`-Aufrufs auszuführen:

- 1470 • `starts_with()`: Hier kann man ein Muster angeben, mit dem ein Text anfangen muss.
- 1471 • `ends_with()`: Diese Funktion ist analog zu `starts_with()`, jetzt wird aber am Ende des Spaltennamens  
 1472 nach dem Muster gesucht.
- 1473 • `contains()`: Hier kann ein Muster übergeben werden, das irgendwo im Spaltennamen sein muss.
- 1474 • `everything()`: Mit dieser Funktion werden alle Spalten ausgewählt.

- 1475 • `last_col()`: Mit dieser Funktion wird nur die letzte Spalte ausgewählt (dass ist die Spalte, die ganz  
1476 rechts ist).

1477 Sämtliche Auswahlen können mit - umgekehrt werden.

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

```
1478 ## Sepal.Length Sepal.Width
1479 ## 1          5.1      3.5
1480 ## 2          4.9      3.0
1481 ## 3          4.7      3.2

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

```
1482 ## Petal.Length Petal.Width Species
1483 ## 1          1.4      0.2  setosa
1484 ## 2          1.4      0.2  setosa
1485 ## 3          1.3      0.2  setosa
```

1486 `select()` bietet auch noch die Möglichkeit, Spaltennamen zu ändern.

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

```
1487 ## sep_width
1488 ## 1      3.5
1489 ## 2      3.0
1490 ## 3      3.2
```

1491

### 1492 Aufgabe 29: Auswählen von Spalten

---

1494 In der Datei `messungen_1.csv` sind Messungen von zwei Sensoren enthalten für die ersten vier Monate eines  
1495 Jahres. Führen Sie folgende Abfragen durch:

- 1496 1. Wählen Sie alle Messungen für Januar aus.  
1497 2. Wählen Sie alle Messungen für Januar und März aus.

## 1498 9.7 Einzelne Beobachtungen abfragen (`slice()`)

1499 Mit dem Befehl `slice()` kann man einzelne Beobachtungen (= Zeilen) aus einem `data.frame` abfragen.

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

```
1500 ## Sepal.Length Sepal.Width Petal.Length Petal.Width Species
1501 ## 1          5.1      3.5      1.4      0.2  setosa
1502 ## 2          4.4      2.9      1.4      0.2  setosa
1503 ## 3          5.1      3.5      1.4      0.3  setosa
```

1504 Davon gibt es drei nützliche Varianten: 1) `slice_head()` und `slice_tail()`; 2) `slice_max()` und  
 1505 `slice_min()`; 3) `slice_random()`.

1506 `slice_head()` und `slice_tail()` sind analog zu `head()` und `tail()`, aber mit dem entscheidenden Unter-  
 1507 schied, dass Gruppierungen berücksichtigt werden. Wenn keine Gruppierung in den Daten vorhanden ist, gibt  
 1508 es keinen Unterschied.

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

```
1509 ##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
1510 ## 1       5.1      3.5      1.4      0.2 setosa
1511 ## 2       4.9      3.0      1.4      0.2 setosa
```

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

```
1512 ##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
1513 ## 1       5.1      3.5      1.4      0.2 setosa
1514 ## 2       4.9      3.0      1.4      0.2 setosa
```

1515 Sobald jedoch eine gruppierende Variable eingeführt wird, gibt `slice_head()` die ersten n Beobachtungen  
 1516 für jede Gruppe zurück und `head()` für den gesamten Datensatz.

```
# base head
```

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

```
1517 ## # A tibble: 2 x 5
1518 ## # Groups:   Species [1]
1519 ##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
1520 ##       <dbl>     <dbl>     <dbl>     <dbl> <fct>
1521 ## 1       5.1      3.5      1.4      0.2 setosa
1522 ## 2       4.9      3       1.4      0.2 setosa
```

```
# dplyr slice_head
```

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

```
1523 ## # A tibble: 6 x 5
1524 ## # Groups:   Species [3]
1525 ##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
1526 ##       <dbl>     <dbl>     <dbl>     <dbl> <fct>
1527 ## 1       5.1      3.5      1.4      0.2 setosa
1528 ## 2       4.9      3       1.4      0.2 setosa
1529 ## 3       7       3.2      4.7      1.4 versicolor
1530 ## 4       6.4      3.2      4.5      1.5 versicolor
1531 ## 5       6.3      3.3       6      2.5 virginica
1532 ## 6       5.8      2.7      5.1      1.9 virginica
```

1533 `slice_tail()` funktioniert analog zu `slice_head()` mit dem einzigen Unterschied, dass nicht die ersten n

1534 Zeilen zurück gegeben werden sondern die letzten `n` Zeilen.  
 1535 `slice_max()` und `slice_min()` geben die Beobachtung mit dem maximalen bzw. minimalen Wert einer  
 1536 Variable zurück. Auch hier werden Gruppen berücksichtigt.

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

1537 ## Sepal.Length Sepal.Width Petal.Length Petal.Width Species  
 1538 ## 1 7.9 3.8 6.4 2 virginica

1539 Und mit Gruppen:

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

1540 ## # A tibble: 3 x 5  
 1541 ## # Groups: Species [3]  
 1542 ## Sepal.Length Sepal.Width Petal.Length Petal.Width Species  
 1543 ## <dbl> <dbl> <dbl> <dbl> <fct>  
 1544 ## 1 5.8 4 1.2 0.2 setosa  
 1545 ## 2 7 3.2 4.7 1.4 versicolor  
 1546 ## 3 7.9 3.8 6.4 2 virginica

1547 `slice_min()` funktioniert genau gleich, nur dass die Beobachtung (=Zeile) mit dem kleinsten Wert einer  
 1548 Variable zurück gegeben wird.

1549 Die Funktion `slice_sample()` erlaubt es zufällige Beobachtungen zu ziehen. Dabei kann über das Argument `n`  
 1550 die Anzahl an Beobachtungen angegeben werden oder über das Argument `prop` der Anteil an Beobachtungen.

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

1551 ## Sepal.Length Sepal.Width Petal.Length Petal.Width Species  
 1552 ## 1 6.5 2.8 4.6 1.5 versicolor  
 1553 ## 2 6.3 3.3 4.7 1.6 versicolor  
 1554 ## 3 7.2 3.2 6.0 1.8 virginica  
 1555 ## 4 4.9 3.6 1.4 0.1 setosa  
 1556 ## 5 6.0 2.7 5.1 1.6 versicolor

1557 Das Ergebnis ist bei jedem von Ihnen anders, da es sich um eine zufällige Ziehung handelt. Wenn Sie diese  
 1558 Ergebnisse wiederholen möchten, können Sie über `set.seed()` die zufällige Ziehung reproduzierbar machen.

```
set.seed(123)  
slice_sample(iris, n = 5)
```

1559 ## Sepal.Length Sepal.Width Petal.Length Petal.Width Species  
 1560 ## 1 4.3 3.0 1.1 0.1 setosa  
 1561 ## 2 5.0 3.3 1.4 0.2 setosa  
 1562 ## 3 7.7 3.8 6.7 2.2 virginica  
 1563 ## 4 4.4 3.2 1.3 0.2 setosa  
 1564 ## 5 5.9 3.0 5.1 1.8 virginica

1565 Wenn beispielsweise 5% der Beobachtungen gezogen werden sollen, kann dies so gemacht werden:

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

```
1566 ##   Sepal.Length Sepal.Width Petal.Length Petal.Width   Species
1567 ## 1         7.7      3.8       6.7      2.2 virginica
1568 ## 2         5.5      2.5       4.0      1.3 versicolor
1569 ## 3         5.5      2.6       4.4      1.2 versicolor
1570 ## 4         6.5      3.0       5.2      2.0 virginica
1571 ## 5         6.1      3.0       4.6      1.4 versicolor
1572 ## 6         6.3      3.4       5.6      2.4 virginica
1573 ## 7         5.1      2.5       3.0      1.1 versicolor

1574 slice_sample() berücksichtigt ebenfalls Gruppen. Mit den Argumenten replace und weight_by dann die
1575 Zufallsziehung genauer spezifiziert werden. replace sagt, ob eine gezogenen Beobachtung wieder zurück gelegt
1576 wird oder nicht. Mit dem Argument weight_by können optional gewichtete für jede Beobachtung vergeben
1577 werden.
```

1578

---

### 1579 Aufgabe 30: Daten beschreiben 1580

1581 Verwenden Sie den Datensatz `bhd_1.txt` und finden Sie für jedes Gebiet und Art die Beobachtung mit
1582 kleinsten BHD.

## 1583 9.8 Spalten trennen

1584 Ein gut geplanter Datensatz besteht aus Beobachtungen (Zeilen), Variablen (Spalten) und in jeder Zelle ist
1585 immer ein **genau** ein Wert gespeichert. Leider gibt es oft Datensätze, bei denen mehr als ein Wert pro Zelle
1586 gespeichert wurde. Die Funktion `separate()` kann helfen solche Daten zu trennen.

1587 Wir verwenden einen erfunden Datensatz zu Beobachtungen von Tieren und einer geschätzten Distanz zu
1588 diesen Tieren.

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

1589 In der Spalte `beobachtung` sind zwei Informationen gespeichert: Die Distanz zur Beobachtung und die Art. Das
1590 ist ungünstig, weil wir so weder nach Tierart noch nach distanz filtern können. Mit der Funktion `separate()`,
1591 können wir Beobachtungen einer Spalte in mehrere Spalten trennen. Dafür muss der Spaltennamen (Argument
1592 `col`), die neuen Spaltennamen (Argument `into`) und das Trennzeichen (Argument `sep`) angegeben werden.

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

1593 ## # A tibble: 4 x 3

1594 ## id Distanz Art

```
1595 ## <int> <chr>   <chr>
1596 ## 1      1 10m     " Reh"
1597 ## 2      2 100m    " Reh"
1598 ## 3      3 20m     " Fuchs"
1599 ## 4      4 40      "Reh"
1600 Nach dem Aufruf von seperate() gibt es zwei neue Spalten (Distanz und Art), die die alte Spalte
1601 beobachtung ersetzen.
```

1602

---

**1603 Aufgabe 31: Aufräumen**

---

1605 Verwenden Sie den folgenden Datensatz und bringen Sie ihn in eine Form, die sicherstellt dass

- 1606 • jede Zelle genau einen Wert enthält.  
1607 • jede Zeile eine Beobachtung ist.  
1608 • 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")
)
```

## 1609 10 Arbeiten mit Text

1610 Bis jetzt haben wir fast ausschließlich mit Zahlen oder Abbildungen gearbeitet. R bietet aber auch viele  
 1611 Werkzeuge, um mit Text zu arbeiten. Wir wollen hier ein paar Funktionen dafür vorstellen. Als erstes sollte  
 1612 nochmals klar gestellt werden, was eigentlich ein Text ist. In R ist alles, das innerhalb von doppelten ("") oder  
 1613 einfachen ('') Anführungszeichen geschrieben ist, Text.

1614 Anbei einige Beispiele:

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

1615 Wichtig ist hier zu sehen, dass z nicht als Zahl sondern, als Text interpretiert wird.

```
z + 1
```

1616 ## Error in z + 1: non-numeric argument to binary operator

1617 Wenn man sicher ist, dass es sich bei einem Textobjekt um eine Zahl handelt, kann man dies mit der Funktion  
 1618 `as.numeric()` in eine Zahl umwandeln.

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

1619 ## [1] 31

1620 Aber mit a führt dies wieder zu einem NA-Wert, da a nicht in eine Zahl umgewandelt werden kann.

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

1621 ## Warning: NAs introduced by coercion

1622 ## [1] NA

### 1623 10.1 Arbeiten mit Text

1624 Wir wollen erst einmal drei Funktionen besprechen, die es erlauben mit Text zu arbeiten. Die Funktion  
 1625 `nchar()`<sup>11</sup> gibt an wie viele Zeichen ein Text hat. Also z.B.

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

1626 ## [1] 5

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

1627 ## [1] 2

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

1628 ## [1] 20

1629 Die Funktion `paste()` erlaubt es verschiedene Variablen mit Text zu verbinden. Wenn wir z. B. die Variablen  
 1630 `vorname <- "Eva"` und `name <- "Meier"` haben und wir wollen eine neue Variable `full_name <- Eva`

---

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

1631 "Meier" erzeugen, dann kann das mit der Funktion `paste()` gemacht werden.

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

1632 ## [1] "Eva Meier"

1633 Die Funktion `paste()` hat das Argument `sep`, das auf ein Leerzeichen ( ) gesetzt ist, aber auch anders sein kann und das Trennzeichen definiert.

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

1635 ## [1] "Eva, Meier"

1636 Die Funktion `substr()` erlaubt es am Anfang oder Ende eines Wortes etwas abzuschneiden. Dabei muss 1637 immer die Anfangs- (`start`) und Endposition (`stop`) angegeben werden.

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

1638 ## [1] "Hal"

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

1639 ## [1] "allo"

1640

#### 1641 Aufgabe 32: Arbeiten mit Text 1

---

1643 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")
```

1644 1. Aus wie vielen Buchstaben besteht jedes Wort?

1645 2. Finden Sie das längste Wort.

1646 3. Wie viel Prozent der Wörter fangen mit einem S an?

1647 4. Fügen Sie jedem Wort seine Position im Vektor hinzu. Beispielsweise soll aus `Vogel` "2. Vogel" werden

1648 usw.

## 1649 10.2 Finden von Textmustern

1650 Mit der Funktion `grep()` können Muster in einem Text gefunden werden. Wenn wir beispielsweise folgenden 1651 Vektor mit Textelementen haben.

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

1652 Und wir wollen alle Straßennamen die ein `weg` haben abfragen, dann können wir folgenden Befehl ausführen:

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

1653 ## [1] 2

1654 Im zweiten Element von `txt` kommen die Zeichen `Weg` vor. Beachte, in der Standardeinstellung wird zwischen  
1655 Groß- und Kleinschreibung unterschieden. Dies kann mit dem Argument `ignore.case = TRUE` angepasst  
1656 werden.

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

1657 ## [1] 1 2

1658 Mit der Funktion `sub` können Zeichen innerhalb einer Zeichenkette ersetzt werden.

1659 So ersetzt der folgende Ausdruck `ae` mit `ä`.

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

1660 ## [1] "Friedländer Weg"

1661 Wenn allerdings das zu ersetzende Zeichen mehr als einmal vorkommt und beide Instanzen ersetzt werden  
1662 sollen braucht man die Funktion `gsub`.

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

1663 ## [1] "Friedländer Weg und Reinhäuser Landstrasse sind zwei Strassen in Goettingen."

1664 Mit `sub()` wird nur das erste `ae` ersetzt, während `gsub()` alle `ae` mit einem `ä` ersetzt.

```
gsub("ae", "ä", txt)
```

1665 ## [1] "Friedländer Weg und Reinhäuser Landstrasse sind zwei Strassen in Goettingen."

1666 Oft ist der genaue Ausdruck den man finden möchte jedoch Variable. Beispielsweise möchte man alle Wörter  
1667 mit einem Umlaut oder Zahlen finden möchte, kann man das oft abkürzen. Dafür gibt es reguläre Ausdrücke.  
1668 Wir werden hier nur ein paar beispielhafte Anwendungen besprechen.

1669 Sowohl in den Funktionen `grep()` als auch `(g)sub()` kann mit anstatt dem Muster (immer das erste  
1670 Argument) aus ein regulärer Ausdruck angegeben werden. Mit `[1-9]` sind alle Zahlen von 1 bis 9 gemeint.

1671 Das Ziel ist es jetzt alle Straßen zu finden, die auch einen Straßennummer haben:

```
txt <- c("Büsgenweg 1", "Berliner Strasse", "Kurze Strasse 13")
```

```
grep("[0-9]", txt)
```

1672 ## [1] 1 3

1673 Damit lässt sich auch das Problem mit groß- und kleingeschriebenen Wörtern lösen.

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

```
grep("[wW]eg", txt)
```

```
1674 ## [1] 1 2
```

1675

---

**Aufgabe 33: Arbeiten mit Text 2**

---

1678 Verwenden Sie den folgenden Vektor:

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

1679 1. In wie vielen Wörtern kommt der Doppellaut au vor?

1680 2. Ersetzen Sie in allen Wörtern alle au mit \_ \_.

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

```
1681 ## [1] 5 6 13
```

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

```
1682 ## [1] "Versicherung" "Methoden"      "Fluss"        "Rudel"        "B_ _m"
```

```
1683 ## [6] "H_ _s"          "Foto"         "Auffahrt"     "Auto"         "Handy"
```

```
1684 ## [11] "Teller"        "Kalender"     "Aufb_ _"
```

## 1685 11 Arbeiten mit Zeit

1686 Für den Computer bzw. R ist ein Datum erst einmal nichts anderes als ein Text. Für uns ist es sofort klar,  
 1687 dass der "13.2.2021" der 13. Februar 2021 ist, für den Computer nicht. Wir müssen R also irgendwie sagen,  
 1688 dass die 13 der Tag ist, die 2 der Monat und 2021 das Jahr. Dass der Computer die einzelnen Komponenten  
 1689 erkennt, nennt man *parsen*<sup>12</sup>. Das Arbeiten mit Datum und Zeit kann anfangs sehr mühsam sein, aber  
 1690 sobald man einige Grundfertigkeiten erworben hat, kann man viele Aufgaben deutlich schneller und effizienter  
 1691 erledigen. Der erste Schritt ist immer ein Datum zu *parsen*. Wir verwenden dafür Funktionen aus dem Paket  
 1692 **lubridate**. Als erstes müssen wir wieder Paket **lubridate** laden mit:

```
library(lubridate)
```

1693 **lubridate** bietet eine Vielzahl von Funktionen zum *parsen* von Datum und Zeit, die sich aus:

- 1694 • y für Jahr,
- 1695 • m für Monat,
- 1696 • d für Tag,
- 1697 • h für Stunde,
- 1698 • m für Minute und
- 1699 • s für Sekunde

1700 zusammen setzten. Alle Funktionen nehmen als erstes Argument ein Textstring. Wenn wir z.B. den String  
 1701 2020-01-20 parsen wollen können wir das mit der Funktion **ymd** machen.

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

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

1703 Dabei erkennt **lubridate** in der Regel die Trennzeichen:

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

1704 ## [1] "2020-01-20"

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

1705 ## [1] "2020-01-20"

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

1706 ## [1] "2020-01-20"

1707 Wenn die die Anordnung der einzelnen Komponenten anders ist, gibt es einfach eine andere Funktion.

```
dmy("20.1.2020")
```

1708 ## [1] "2020-01-20"

1709 Jetzt stellt sich die Frage, was der Vorteil ist, wenn R ein Datum parst.

---

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

```
d <- dmy("20.1.2020")
```

1710 Wir können jetzt mit d arbeiten und einzelne Komponenten extrahieren.

```
day(d)
```

1711 ## [1] 20

```
month(d)
```

1712 ## [1] 1

```
year(d)
```

1713 ## [1] 2020

1714 Oder auch Zeiteinheiten hinzufügen oder abziehen.

```
d + days(10)
```

1715 ## [1] "2020-01-30"

```
d - years(20)
```

1716 ## [1] "2000-01-20"

```
d + hours(25)
```

1717 ## [1] "2020-01-21 01:00:00 UTC"

1718

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

---

- 1719 • Parsen Sie folgende Zeitangaben 23.1.2020, 13.2.1992 20:55:23, Mar/3/97 und 10.7.2020 19:15  
1720 und speichern Sie diese in einen Vektor d.  
1721 • Extrahieren Sie nun aus jedem Element aus d das Jahr und die Stunde.  
1722 • Fügen zu jedem Element in d 10 Tage hinzu.

## **11.1 Arbeiten mit Zeitintervallen**

1726 Mit zwei Zeitpunkten lassen sich Zeitintervalle (Periods) erstellen, dafür können wir die Funktion `interval()`  
1727 aus dem Paket lubridate verwenden<sup>13</sup>.

```
anfang <- ymd("2020-03-18")
```

```
ende <- anfang + years(1)
```

```
int <- interval(anfang, ende)
```

1728 Wir können jetzt mit int arbeiten und beispielsweise das Intervall verschieben,

<sup>13</sup>Alternativ zur Funktion `interval()` kann auch der `%--%`-Operator verwendet werden. Man könnte int auch so erstellen `int <- anfang %--% ende`.

```

int_shift(int, years(3))

1729 ## [1] 2023-03-18 UTC--2024-03-18 UTC
1730 die Länge des Intervalls berechnen
int_length(int) # in Sekunden

1731 ## [1] 31536000
1732 oder testen ob ein Datum innerhalb des Intervalls liegt.
ymd("2020-07-1") %within% int

1733 ## [1] TRUE
1734 ## [1] FALSE
1735 %within% funktioniert genauso mit Vektoren oder mit mehreren Intervallen. Wir könnten also zwei Intervalle
1736 definieren (z.B. Ostern und Pfingsten).
ostern <- ymd("2021-04-02") %--% ymd("2021-04-05")
pfingsten <- ymd("2021-05-22") %--% ymd("2021-05-24")

1737 Und Überprüfen welche Termine in eines der zwei Intervalle fallen.
termine <- ymd("2021-03-29") + weeks(0:10)

# Ostern
termine %within% ostern

1738 ## [1] FALSE TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
# Pfingsten
termine %within% pfingsten

1739 ## [1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE TRUE FALSE FALSE

1740 11.2 Formatieren von Zeit

1741 Für die Ausgabe in Berichten oder Grafiken soll das Datum oft in einer speziellen Form dargestellt werden.
1742 Die Funktion format() bietet Möglichkeiten ein Datumsobjekt zurück in Text umzuwandeln.

1743 Ein Beispiel wäre ymd("2021-2-10") als 10.2.21 auszugeben.

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

1744 ## [1] "21.02.21"

1745 Dabei handelt sich bei %d.%m.%y um Abkürzungen für die unterschiedlichen Komponenten eines Datumsobjekts.
1746 Siehe dazu die Hilfeseite von strptime (help(strptime)).
```

1747

---

**Aufgabe 35: Arbeiten mit Intervallen**

---

- 1748 1750 Wie viele Einträge aus dem Vektor v1 befinden sich in einem Intervall, das zwischen dem 1.3.2021 und dem  
1749 1751 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"  
)
```

## 1752 12 Aufgaben Wiederholen (for-Schleifen)

1753 Um einfache Programme zu schreiben, müssen Sie den Ablauf der Programmcodes kontrollieren können.  
 1754 Kontrollieren bedeutet in diesem Zusammenhang, dass Codeabschnitte nur unter definierten Bedingungen  
 1755 ablaufen. Sie programmieren also zwei Sachen. 1) den Code selbst und 2) die Bedingungen die erfüllt sein  
 1756 müssen, damit der Code ausgeführt wird. Diese Kontrollbedingungen ermöglichen es Ihnen generisch zu  
 1757 programmieren. Sie schreiben Ihren Code also nicht speziell maßgeschneidert für ein Problem, sondern so  
 1758 generell, dass er für mehrere Auswertungen funktioniert. Um dies zu gewährleisten müssen Sie bestimmte Si-  
 1759 tuationen vorhersehen und abfangen. Hierbei helfen Ihnen Kontrollstrukturen (**Control Flow**). Grundsätzlich  
 1760 gibt es **Control Flow** Funktionen zur Wiederholung von Codeblöcken (Schleifen) und logische Bedingungen  
 1761 (bedingte Anweisung).

### 1762 12.1 Schleifen

1763 Bis jetzt wurden alle Skripte einfach der Reihe nach abgearbeitet und zwischendurch bestimmte Programmteile,  
 1764 je nach Situation, selbstständig ausgeführt oder übersprungen. Mit einer Schleife kann man erreichen, dass  
 1765 eine Gruppe von Befehlen (der sog. Schleifenrumpf) mehrfach abgearbeitet wird, zum Beispiel wenn bestimmte  
 1766 Auswertungsschritte auf mehrere Datensätze oder Variablen angewendet oder Funktionen mit unterschiedlichen  
 1767 Parametern oder Startwerten aufgerufen werden sollen. Weitere Anwendungsmöglichkeiten sind iterative  
 1768 Algorithmen, in denen die Eingabewerte des aktuellen Iterationsschrittes von einem vorherigen abhängig sind.  
 1769 Besonders in Simulationen kommen Schleifen häufig zum Einsatz, da große Anzahlen von Wiederholungen  
 1770 benötigt werden.

1771 Man unterscheidet zwischen zwei Arten von Schleifen: Bei den **for()**-Schleifen steht die Anzahl der Wieder-  
 1772 holungen schon beim Eintritt in die Schleife fest, während die **while()**-Schleifen so lange ausgeführt werden,  
 1773 bis eine Bedingung nicht mehr wahr ist. Mit der Funktion **break** wird eine Schleife abgebrochen und die  
 1774 Programmausführung wird nach der Schleife fortgesetzt.

1775 Die wesentlichen Befehle sind

- 1776 • **for (i in X) {Code}**

1777 Wiederhole den Code im “Schleifenrumpf” für jedes Element aus X.

- 1778 • **while(Bedingung) {Code}**

1779 Wiederhole den Code im “Schleifenrumpf” so lange die logische Bedingung erfüllt ist.

- 1780 • **break()**

1781 Brich die Schleife ab.

#### 1782 12.1.1 Wiederholen von Befehlen mit **for()**.

1783 Steht vor Beginn der Schleife fest wie viele Schleifendurchgänge benötigt werden, wenn zum Beispiel in einer  
 1784 Simulationen 99 Realisierungen erzeugt oder alle Elemente eines Vektors verarbeitet werden sollen, verwendet  
 1785 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)
}
```

```
1786 ## [1] 1
1787 ## [1] 2
1788 ## [1] 3
```

1789 Das `i` steht in diesem Beispiel für die Schleifen-Variable. Sie muss nicht `i` heißen, sondern kann jeden  
 1790 zulässigen Namen annehmen. Das `X` steht für einen existierenden Vektor oder eine existierende Liste bzw.  
 1791 einen Ausdruck, der ein solches Objekt liefert (der Objektname ist ebenfalls frei wählbar). `for` und `in` sind  
 1792 Schlüsselworte, sie müssen, ebenso wie die runden Klammern, vorhanden sein.

1793 Im ersten Durchgang erhält die Schleifen-Variable `i` den ersten Wert von `X` und der Schleifenrumpf wird  
 1794 mit diesem Wert ausgeführt. Die Variable `i` nimmt nacheinander so lange die Werte von `X` an, bis ihr alle  
 1795 Elemente zugewiesen wurden.

1796 Das folgende Beispiel wird zwar besser durch die entsprechende Vektoroperation gelöst, zeigt aber sehr  
 1797 deutlich die Arbeitsweise der `for`-Schleife.

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

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

```
1798 ## [1] 4
1799 ## [1] 9
1800 ## [1] 25
```

1801

### 1802 Aufgabe 36: Schleifen 1

---

1804 Verwenden Sie den Vektor `k <- c(1, 3, 9, 12, 15)` und schreiben Sie folgende `for`-Schleifen:

- 1805 1. Eine Schleife, die jedes Element aus `k` ausgibt.
- 1806 2. Eine Schleife, die zu jedem Element aus `k` 10 addiert und den neuen Wert ausgibt.
- 1807 3. Eine Schleife wie in 2), aber der neue Wert (`k + 10`) soll jetzt nicht mehr ausgegeben werden, sondern  
 1808 in `k10` gespeichert werden. Stellen Sie sicher, dass `k10` wieder von der Länge 5 ist.

1809

---

1810 Die Funktion `for()` ermöglicht es, einen Befehl beliebig oft zu wiederholen. Z.B. der folgende Ausdruck zieht  
 1811 10-Mal eine Stichprobe der Größe 1 aus dem Vektor `v`. Beachten Sie, dass die Schleifen-Variable `i` selbst gar

1812 nicht im Schleifenrumpf vorkommt. Das Ziel dieser Schleife ist nicht die Elemente des Vektors abzuarbeiten,  
 1813 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))
}
```

```
1814 ## [1] 3
1815 ## [1] 4
1816 ## [1] 2
1817 ## [1] 4
1818 ## [1] 1
1819 ## [1] 4
1820 ## [1] 2
1821 ## [1] 3
1822 ## [1] 4
1823 ## [1] 1
```

1824 Auf gleiche Weise kann man auch über die Variablen eines Dataframes iterieren<sup>14</sup>. Das folgende Beispiel hat  
 1825 zum Ziel die Funktionsweise von Schleifen zu verdeutlichen. Schleifen haben in R jedoch den Nachteil, dass sie  
 1826 sehr langsam operieren. Wenn es geht, sollte man Alternativen verwenden. Die Funktionsweise wiederholender  
 1827 Auswertungen wird jedoch mit `for`-Schleifen deutlicher. Aus diesem Grunde werden wir uns in diesem Kurs  
 1828 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>14</sup>Zur Info: Dieses Beispiel lässt sich schneller mittels der vektorwertigen Operation `apply()` lösen.

1837

---

1838 **Aufgabe 37: for-Schleife**  
1839

---

1840 Lesen Sie den Datensatz `bhd_1.txt` ein und verwenden Sie eine `for`-Schleife.

- 1841 • Ziehen Sie 500-Mal je 35 Beobachtungen für den BHD.
- 1842 • Berechnen Sie jeweils den Mittelwert aus den 35 Werten.
- 1843 • Speichern Sie die 500 Mittelwerte in einem neuen Vektor `mittel`.
- 1844 • Wie ist die Verteilung dieser 500 Mittelwerte zu interpretieren?

1845 **12.1.2 Wiederholen von Befehlen mit `while()`**

1846 Die `while`-Schleifen finden Anwendung, wenn die Anzahl der zu durchlaufenden Wiederholungen vorher  
1847 nicht bekannt ist, wie zum Beispiel bei Iterationsverfahren, die bis zu einer gewissen Genauigkeit durchlaufen  
1848 werden sollen. Die `while`-Schleife besteht in R aus dem Schlüsselwort `while()` und einer Bedingung in runden  
1849 Klammern.

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

1850 Sie ist in der praktischen Programmierung nicht so relevant wie die `for`-Schleife. Sie sei deshalb hier nur  
1851 kurz erwähnt. Die Abbruchbedingung wird jedes Mal geprüft bevor der Schleifenrumpf durchlaufen wird. Die  
1852 Bedingung wird ausgewertet und wenn diese `TRUE` ist, wird der Schleifenrumpf ausgeführt und danach erneut  
1853 die Bedingung überprüft. Ist die Bedingung nicht erfüllt, wird der Schleifenrumpf nicht durchgeführt und die  
1854 Schleife beendet. Ist die Bedingung bereits vor Eintritt in die Schleife nicht erfüllt, wird die Schleife gar nicht  
1855 erst durchlaufen.

1856 Da `while`-Schleifen also so lange ausgeführt werden, bis die Bedingung nicht mehr erfüllt ist, kann eine  
1857 Endlosschleife entstehen. Dies kann passieren, wenn man nicht sauber programmiert hat. Wenn innerhalb der  
1858 Schleife nicht dafür gesorgt wird, dass die Bedingung irgendwann nicht mehr erfüllt wird, so läuft die Schleife  
1859 immer weiter. Steckt R in einer Schleife fest und reagiert nicht mehr, kann der Befehl unter Linux mit `Strg`+  
1860 `C` und unter Windows mit `Esc` abgebrochen werden. Alternativ können Sie auf das rote STOP Symbol  
1861 über der Konsole klicken.

1862 **12.2 Bedingte Ausführung von Codeblöcken**

1863 Innerhalb eines Skripts ist es mitunter notwendig je nach aktueller Situation unterschiedlich fortzufahren.  
1864 Die Situation wird mit einem logischen Ausdruck, einer sogenannten Bedingung, geprüft. Je nachdem, ob  
1865 die Bedingung wahr (`TRUE`) oder falsch (`FALSE`) ist, werden unterschiedliche Programmteile ausgeführt, der  
1866 jeweils andere Teil bleibt unberücksichtigt. Danach wird in jedem Fall die Programmausführung, mit den  
1867 auf die bedingte Anweisungen folgenden Anweisung, fortgesetzt. In R kann dies mit dem if-else-Konstrukt  
1868 realisiert werden, welches aus den Schlüsselwörtern `if()` und `else` sowie der in runde Klammern gefassten  
1869 Bedingung besteht.

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

1870 Im folgenden Beispiel sollen die bisher abstrakt beschriebenen Vorgänge praktische Anwendung finden. In  
 1871 dem Beispiel wird zunächst durch zufälliges Ziehen einer Zahl aus dem Bereich eins bis sechs ein Würfelwurf  
 1872 simuliert. Anschließend wird der Würfelwurf mit einem if-Ausdruck kommentiert. Ist die Bedingung, es wurde  
 1873 eine Sechs gewürfelt, erfüllt, wird der Code innerhalb der geschweiften Klammern ausgeführt, ansonsten wird  
 1874 der Klammerinhalt ignoriert.

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

1875 In den meisten Fällen ist es für R irrelevant, ob sich zwischen den verschiedenen Elementen Leerzeichen oder  
 1876 Zeilenumbrüche befinden. Bei dem else-Ausdruck dagegen wird ein Fehler erzeugt, wenn vor dem `else` nicht  
 1877 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.")
}
```

1878 `## [1] "Beim nächsten Wurf klappt's bestimmt."`

1879

### 1880 Aufgabe 38: Bedingte Programmierung

---

- 1882 • Wenn keine 6 gewürfelt wurde, lassen Sie zusätzlich ausgeben welche Zahl gewürfelt wurde.  
 1883 • Wiederholen Sie den Würfelwurf 10 Mal.

## 1884 13 (R)markdown

### 1885 13.1 Markdown Grundlagen

1886 Die Idee von Markdown ist, einfach Text strukturieren zu können und das ganze ohne umfangreiche Programme  
 1887 zu erstellen. Im nächsten Abschnitt sehen wir dann, wie man Markdown und R-Code verbinden kann. Hier  
 1888 soll es jedoch erst einmal darum gehen, die einfachsten Bausteine von Markdown vorzustellen.

1889 Am Anfang jedes Dokuments kommt eine Präambel. Diese fängt mit --- an und hört auch wieder mit ---  
 1890 auf. Innerhalb der Präambel können dann Metainformationen über das Dokument festgelegt werden, dies  
 1891 beinhaltet im einfachsten Fall: Titel, Autor und Datum. Das würde dann so aussehen:

```
1892 ---
1893 title: "Ein Titel"
1894 author: "Der, der es geschrieben hat"
1895 date: "März 2021"
1896 ---
```

1897 Danach folgt strukturierter und formatierter Text. Verschiedene Hierarchieebenen von Überschriften können  
 1898 mit der Anzahl an # festgelegt werden. So ist eine Überschrift erster Ordnung # Kapitel eine Überschrift  
 1899 zweiter Ordnung ## Unterkapitel usw.

1900 Listen können erstellt werden, wenn man am Anfang jeder Zeile ein - oder 1. schreibt.

```
1901 - Erster Eintrag
1902 - Zweiter Eintrag
1903 - Dritter Eintrag
```

1904 wird zu

```
1905   • Erster Eintrag
1906   • Zweiter Eintrag
1907   • Dritter Eintrag
```

1908 Eine zentrale Idee von Markdown ist es Text einfach zu formatieren. Werden eine oder mehrere Wörter mit  
 1909 zwei Sternchen (\*\*) eingefasst wird dieser Text **fett** dargestellt. Also aus \*\*wichtig\*\* wird **wichtig**. Das  
 1910 gleiche funktioniert auch mit *kursiven* Text, jedoch muss man hier noch ein Sternchen verwenden, also aus  
 1911 \*kursiv\* wird *kursiv*. Soll ein text fett und kursiv sein, kann man drei Sternchen verwenden. Aus \*\*\*sehr  
 1912 wichtig\*\*\* wird dann **sehr wichtig**.

1913 Weitere Elemente wie Links oder Abbildungen können einfach eingebunden werden. Links werden mit [Link  
 1914 text](url) in den Text integriert. Beispielsweise ist eine gute Idee bei [stackoverflow](#) bei Problemen nach  
 1915 einer Lösung zu suchen. Dieser Link wurde mit [stackoverflow](www.stackoverflow.com) erstellt.

1916 Für Abbildungen gibt es einen ganz ähnlichen Syntax. Mit ! [Das R Logo](abb/r\_logo.png) wird die  
 1917 Abbildung r\_logo.png eingebunden mit der Beschriftung: "Das R Logo".



Abbildung 9: Das R Logo

1918

1919 **Aufgabe 39: Arbeiten mit markdown**  
1920

1921 Verwenden Sie das folgende Markdowndokument:

```

1922 ---
1923 title: "Dokument"
1924 author: "Ihr Name"
1925 date: "März 2021"
1926 ---
1927
1928 # Einleitung
1929
1930 # Methoden
1931 1. Kopieren Sie die Vorlage in ein Dokument, das test.md heißt.
1932 2. Fügen Sie zwei Überschriften zweiter und dritter Ordnung hinzu.
1933 3. Fügen Sie einen kursiven Text hinzu.
1934 4. Fügen Sie einen Link zu einer Website hinzu.
1935 5. Kompilieren Sie die Datei, indem Sie in Rstudio auf Preview drücken (Abbildung 10).

```

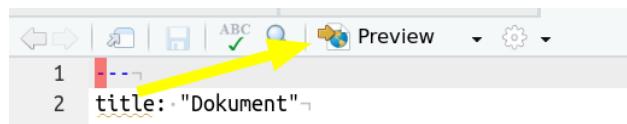


Abbildung 10: Kompilieren einer md-Datei.

1936 **13.2 R und Markdown**

1937 Markdown macht es bereits einfach Textdokumente und Dokumentationen zu verfassen, aber die wirkliche  
1938 Stärke liegt in der Möglichkeit R und Markdown zu kombinieren. Man spricht dann von Rmarkdown. Ein  
1939 weiteres Strukturelement, das wir noch nicht kennen gelernt haben, sind Code-Blöcke.

1940 ~~~

1941 a &lt;- 1:10

```

1942 a[1]
1943 ``
1944 erzeugt
1945 a <- 1:10
1946 a[1]

1947 Momentan wird noch kein Code ausgeführt, sondern lediglich der Code, als Code dargestellt. Rmarkdown
1948 bietet nun die Möglichkeit Code beim kompilieren15 auszuführen. Dafür müssen wir nur einen Code-Block als
1949 R-Code-Block kennzeichnen.

1950 ``{R}
1951 a <- 1:10
1952 a[1]
1953 ```

1954 erzeugt
1955 a <- 1:10
1956 a[1]

1957 ## [1] 1
1958 Beachte, die Variable a wird beim kompilieren erzeugt und steht dann R zur Verfügung. R-Code-Blöcke
1959 werden auch als Code Chunks bezeichnet. Diese Chunks können sehr genau angesprochen und angepasst
1960 werden. Einige wichtige Argumente sind:
1961


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

```

1962

---

**Aufgabe 40: Arbeiten mit Rmarkdown**


---

1963 Erstellen Sie eine neue Rmarkdown Datei mit dem Namen **test1.Rmd**. Erstellen Sie zwei Code-Chunks. Der  
1964 erste soll nicht angezeigt werden und darin werden die Daten geladen (**bhd\_1.txt**). Im zweiten Chunk plotten  
1965 Sie das Alter der Bäume gegen den BHD. Was passiert mit dem Plot, wenn Sie die Datei kompilieren (drücken  
1966 Sie dazu auf den Knit-Knopf; Abbildung 11).



Abbildung 11: Kompilieren einer Rmd-Datei.

---

<sup>15</sup>Unter kompilieren wird hier das Übersetzen eines Markdowndokuments in ein Ausgabeformat (z.B. pdf oder html) verstanden.

## 1969 14 Räumliche Daten in R

### 1970 14.1 Was sind räumliche Daten

1971 Räumliche Daten sind Beobachtungen, wie wir sie schon oft gesehen haben, mit einem räumlichen Bezug. Der  
 1972 Unterschied zu nicht räumlichen Daten liegt darin, dass räumliche Daten eindeutig im *Raum* verortet werden  
 1973 können. Häufig werden sogenannte Geoinformationssysteme (GIS) zum Arbeiten mit räumlichen verwendet.  
 1974 R kann in vielerlei Hinsicht wie ein GIS eingesetzt werden und hier werden einige Grundfunktionalitäten  
 1975 dafür besprochen. Räumliche Daten werden in zwei unterschiedliche Datentypen unterteilt: Vektor- und  
 1976 Rasterdaten. Vektordaten modellieren einzelne Objekte (= *Features*). Rasterdaten modellieren eine Oberfläche.  
 1977 Vektordaten bestehen aus zwei Komponenten: 1) einer Geometrie, die die Form und Lage der Daten definiert  
 1978 und 2) Attributen, den tatsächlichen Daten. Räumliche Daten werden oft als *Features* bezeichnet. Ein Feature  
 1979 ist die räumliche Einheit einer Beobachtung. Je nach Art der räumlichen Daten können Features entweder  
 1980 Punkte (z.B. ein Baum), Linien (z.B. eine Straße) oder Polygone (z.B. ein See) sein. Auch können mehrere  
 1981 Geometrien zu einem Feature zusammengefasst werden. Ein Beispiel wäre eine Beobachtung für ein Land,  
 1982 das aber aus mehreren Polygonen bestehen kann (z.B. Festland und Inseln). Features können dann weitere  
 1983 Attribute (= Attributdaten) haben, z.B. eine ID, Name oder was auch immer man gemessen hat.  
 1984 Rasterdaten bestehen aus einer Oberfläche von gleichgroßen Kacheln (= *Pixel*), die ein Gebiet abdecken.  
 1985 Meist sind Pixel viereckig, aber das ist keine Voraussetzung. Dabei hat jedes Pixel einen Wert (das kann  
 1986 auch ein fehlender Wert sein). Typische Beispiele für Rasterdaten sind Landnutzung oder Seehöhen.  
 1987 In R kann sowohl mit Vektor- als auch mit Rasterdaten gearbeitet werden. Für Vektordaten bietet sich das  
 1988 Paket **sf** an und für Rasterdaten das Paket **raster**.

### 1989 14.2 Koordinatenbezugssystem

1990 Eine der Herausforderungen für räumliche Daten ist die eindeutige Verortung im Raum. Dazu braucht man  
 1991 ein *Koordinatenbezugssystem (KBS)*. Einem KBS liegt ein mathematisches Modell der Erde zugrunde. Die  
 1992 Details zu KBS werden schnell relativ kompliziert und wir beschränken uns hier lediglich darauf, wie KBS  
 1993 verwendet werden können. Dazu müssen wir zwei Fälle unterscheiden: 1) einem Datensatz ein KBS zuweisen  
 1994 und 2) Transformation des KBSs eines Datensatzes in ein anderes KBS. Die technischen Details werden in  
 1995 den folgend Abschnitten besprochen. Für beide Aufgaben müssen wir auf ein KBS verweisen können, ein  
 1996 einfacher Ansatz dafür sind die sogenannten *EPSG-Codes*<sup>16</sup>.

### 1997 14.3 Vektordaten in R

1998 Das Paket **sf** stellt Klassen zum Abbilden von Features zur Verfügung, die dann in einem **data.frame** als  
 1999 Liste gespeichert werden können. In der Regel erstellen wir Features nicht individuell, sondern lesen diese aus  
 2000 externen Datenquellen (z.B. ESRI Shapefile) ein. Zum besseren Verständnis, erstellen wir es einmal manuell.  
 2001 Wir haben die Koordinaten für drei Städte (Göttingen, Hannover und Berlin) als geografische Koordinaten  
 2002 vorliegen (EPSG = 4326).

<sup>16</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))
```

2003 Daraus könne wir jetzt eine Geometriespalte für einen `data.frame` erstellen

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

2004 Somit haben wir die Geometrie in der Variable `geom` gespeichert, aber noch keine dazugehörigen Attributdaten.

2005 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)
)
```

2006 In einem letzten Schritt möchten wir jetzt die Geometrie (`geom`) und die Attributdaten (`attr`) zusammenführen.

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

2008 ## Simple feature collection with 3 features and 3 fields

2009 ## Geometry type: POINT

2010 ## Dimension: XY

2011 ## Bounding box: xmin: 9.732 ymin: 51.5413 xmax: 13.405 ymax: 52.52

2012 ## Geodetic CRS: WGS 84

2013 ## name bundesland einwohner geom

2014 ## 1 Goettingen Niedersachsen 119000 POINT (9.9158 51.5413)

2015 ## 2 Hannover Niedersachsen 532000 POINT (9.732 52.3759)

2016 ## 3 Berlin Berlin 3650000 POINT (13.405 52.52)

2017 Wir können nun mit `staedte` genau so arbeiten wie mit jedem anderen `data.frame` und die Geometrien

2018 werden immer ‘berücksichtigt’. Zusätzlich kann man eine Reihe von geometrischen Operationen durchführen.

2019 Wenn ein `data.frame` Punkten hat, kann man dies relative einfach mit der Funktion `st_as_sf()` “räumlich”  
2020 machen. Für das vorherige Beispiel würden wir zuerst einen `data.frame` mit allen Informationen (zur  
2021 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)
)
```

2022 Dann kann man mit der Funktion `st_as_sf()` weiter arbeiten:

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

## 2023 14.4 Arbeiten mit Vektordaten

2024 Es gibt sehr viele Funktionen, um mit räumlichen Daten zu arbeiten, von denen wir hier einige vorstellen.

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

```
2025 ## Coordinate Reference System:
2026 ##   User input: EPSG:4326
2027 ##   wkt:
2028 ## GEOGCRS["WGS 84",
2029 ##   ENSEMBLE["World Geodetic System 1984 ensemble",
2030 ##     MEMBER["World Geodetic System 1984 (Transit)"],
2031 ##     MEMBER["World Geodetic System 1984 (G730)"],
2032 ##     MEMBER["World Geodetic System 1984 (G873)"],
2033 ##     MEMBER["World Geodetic System 1984 (G1150)"],
2034 ##     MEMBER["World Geodetic System 1984 (G1674)"],
2035 ##     MEMBER["World Geodetic System 1984 (G1762)"],
2036 ##     MEMBER["World Geodetic System 1984 (G2139)"],
2037 ##     ELLIPSOID["WGS 84",6378137,298.257223563,
2038 ##       LENGTHUNIT["metre",1]],
2039 ##     ENSEMBLEACCURACY[2.0]],
2040 ##   PRIMEM["Greenwich",0,
2041 ##     ANGLEUNIT["degree",0.0174532925199433]],
2042 ##   CS[ellipsoidal,2],
2043 ##     AXIS["geodetic latitude (Lat)",north,
2044 ##       ORDER[1],
2045 ##       ANGLEUNIT["degree",0.0174532925199433]],
2046 ##     AXIS["geodetic longitude (Lon)",east,
2047 ##       ORDER[2],
2048 ##       ANGLEUNIT["degree",0.0174532925199433]],
2049 ##   USAGE[
2050 ##     SCOPE["Horizontal component of 3D system."],
2051 ##     AREA["World."],
2052 ##     BBOX[-90,-180,90,180]],
2053 ##     ID["EPSG",4326]]
```

2054 Wenn wir jetzt zu einem anderen KBS (z.B. EPSG:3035, ein europäisches projiziertes KBS) umrechnen  
 2055 möchten, können wir das mit

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

```

2056 ## Coordinate Reference System:
2057 ##   User input: EPSG:3035
2058 ##   wkt:
2059 ## PROJCRS["ETRS89-extended / LAEA Europe",
2060 ##   BASEGEOGCRS["ETRS89",
2061 ##     ENSEMBLE["European Terrestrial Reference System 1989 ensemble",
2062 ##       MEMBER["European Terrestrial Reference Frame 1989"],
2063 ##       MEMBER["European Terrestrial Reference Frame 1990"],
2064 ##       MEMBER["European Terrestrial Reference Frame 1991"],
2065 ##       MEMBER["European Terrestrial Reference Frame 1992"],
2066 ##       MEMBER["European Terrestrial Reference Frame 1993"],
2067 ##       MEMBER["European Terrestrial Reference Frame 1994"],
2068 ##       MEMBER["European Terrestrial Reference Frame 1996"],
2069 ##       MEMBER["European Terrestrial Reference Frame 1997"],
2070 ##       MEMBER["European Terrestrial Reference Frame 2000"],
2071 ##       MEMBER["European Terrestrial Reference Frame 2005"],
2072 ##       MEMBER["European Terrestrial Reference Frame 2014"],
2073 ##       ELLIPSOID["GRS 1980",6378137,298.257222101,
2074 ##         LENGTHUNIT["metre",1]],
2075 ##       ENSEMBLEACCURACY[0.1]],
2076 ##       PRIMEM["Greenwich",0,
2077 ##         ANGLEUNIT["degree",0.0174532925199433]],
2078 ##       ID["EPSG",4258]],
2079 ##     CONVERSION["Europe Equal Area 2001",
2080 ##       METHOD["Lambert Azimuthal Equal Area",
2081 ##         ID["EPSG",9820]],
2082 ##       PARAMETER["Latitude of natural origin",52,
2083 ##         ANGLEUNIT["degree",0.0174532925199433],
2084 ##         ID["EPSG",8801]],
2085 ##       PARAMETER["Longitude of natural origin",10,
2086 ##         ANGLEUNIT["degree",0.0174532925199433],
2087 ##         ID["EPSG",8802]],
2088 ##       PARAMETER["False easting",4321000,
2089 ##         LENGTHUNIT["metre",1],
2090 ##         ID["EPSG",8806]],
2091 ##       PARAMETER["False northing",3210000,
2092 ##         LENGTHUNIT["metre",1],
2093 ##         ID["EPSG",8807]]],
2094 ##     CS[Cartesian,2],
2095 ##       AXIS["northing (Y)",north,
2096 ##         ORDER[1],
2097 ##         LENGTHUNIT["metre",1]],
2098 ##       AXIS["easting (X)",east,

```

```

2099 ##           ORDER[2],
2100 ##           LENGTHUNIT["metre",1]],
2101 ##           USAGE[
2102 ##           SCOPE["Statistical analysis."],
2103 ##           AREA["Europe - European Union (EU) countries and candidates. Europe - onshore and offshore: "],
2104 ##           BBOX[24.6,-35.58,84.73,44.83]],
2105 ##           ID["EPSG",3035]

```

2106 Die Funktion `st_buffer()` erlaubt es Features zu puffern, mit `st_distance()` kann die Distanz zwischen  
2107 Features berechnet werden, mit `st_area()` kann die Fläche eines Features berechnet werden.

2108 Funktionen wie `st_intersection()`, `st_union()` und `st_difference()` erlauben es geometrische Opera-  
2109 tionen zwischen unterschiedlichen Features zu berechnen. Für eine ausführliche Diskussion siehe auch hier:  
2110 <https://geocompr.robinlovelace.net/geometric-operations.html>.

2111 Normalerweise lesen wir Daten von externen Datenquellen (z.B. ESRI Shapefiles). Das geht mit der Funktion  
2112 `st_read()`.

## 2113 14.5 Rasterdaten in R

2114 Für Rasterdaten gibt es das R-Paket `raster`. Auch hier wollen wir uns wieder auf einige Grundfunktionalitäten  
2115 konzentrieren. Diese umfassen das Einlesen, Zuschneiden, Rechnen und Abfragen von Rastern.

2116 Mit der Funktion `raster()` kann ein Raster in R eingelesen werden.

```

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

```

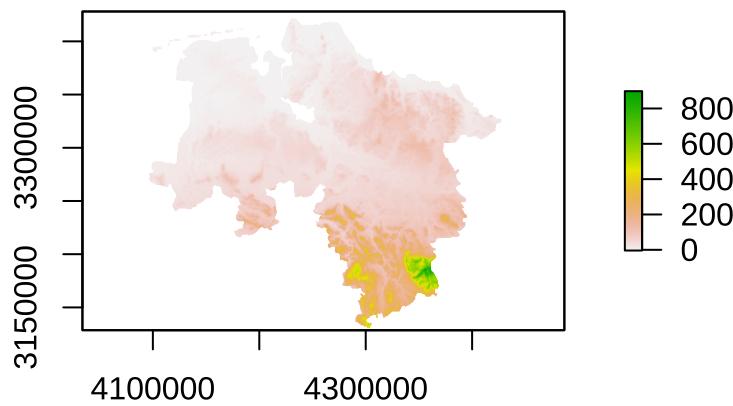
2117 `dem` steht für *Digital Elevation Model* und ist ein Raster mit den Seehöhen in Niedersachsen mit einer  
2118 500-m-Auflösung. Wir können diese mit der Funktion `res()`<sup>17</sup> abfragen.

```
res(dem)
```

2119 ## [1] 500 500

2120 Bzw. wir können den Raster auch plotten.

```
plot(dem)
```



2121

<sup>17</sup>kurz für *resolution* also Auflösung.

2122 Wenn wir den Raster `dem` auf ein Gebiet zuschneiden wollen (z.B. Göttingen), müssen wir drei Schritte  
2123 durchführen. Als erstes müssen wir ein Shapefile für Göttingen einlesen.

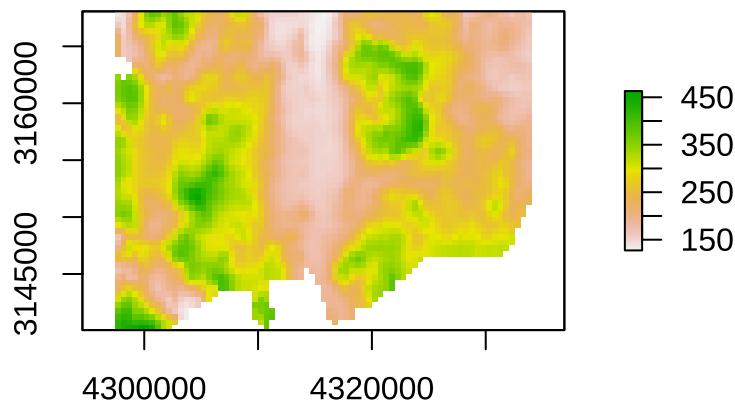
```
goe <- st_read(here::here("data/goettingen/stadt_goettingen.shp"))
```

2124 Dann müssen wir sicherstellen, dass sowohl der Raster `dem` als auch das `sf`-Objekt `goe` im selben KBS sind.  
2125 Es bietet es sich in der Regel an, das KBS des Vektors zu transformieren. Mit der Funktion `projectRaster()`  
2126 kann das KBS eines Rasters transformiert werden.

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

2127 Mit der Funktion `crop()` kann der Raster `dem` auf Göttingen zugeschnitten werden.

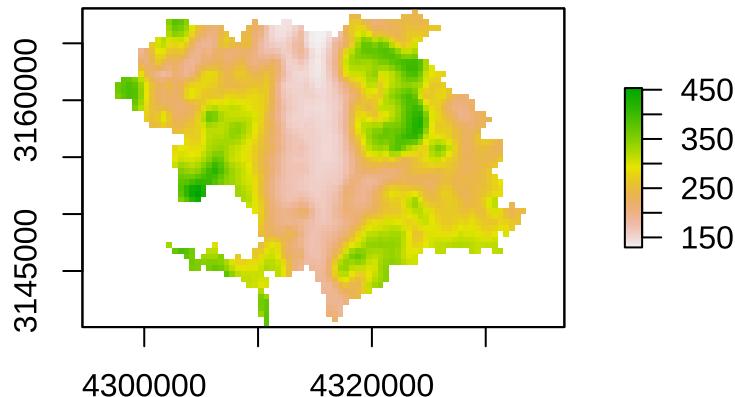
```
dem1 <- crop(dem, goe)
plot(dem1)
```



2128

2129 Der Raster hat jetzt die Größe einer Bounding-Box (BBX) von Göttingen (das ist ein Rechteck, das Göttingen  
2130 umfasst). Mit der Funktion `mask()` kann der Raster auf die genauen Grenzen des Vektors `goe` angepasst  
2131 werden.

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



2132

2133 Wenn wir an bestimmten Punkten den Wert des Rasters abfragen wollen (z.B. an `cities`) von vorhin, dann  
2134 gibt es dafür die Funktion `extract`. Dann müssen wir erst sicherstellen, dass `staedte` und `dem` gleichen  
2135 KBS zu grunde liegen. Dafür transformieren wir einfach `staedte` in das KBS von `dem`. Mit der Funktion  
2136 `projection()` erhalten wir das KBS des Rasters.

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

2137 Wenn wir das KBS eines Objektes nicht kennen, können wir auch einfach das KBS übergeben. Der folgende  
 2138 Code-Block macht genau das Gleiche mit dem Vorteil, dass wir keinen EPSG-Code angeben müssen.

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

2139 Dann können wir für jede Stadt die Seehöhe abfragen:

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

2140 ## [1] 149.18181 57.21486 NA

2141 Mit `raster::extract()` rufen wir *eindeutig* die Funktion `extract()` aus dem Paket `raster` auf. Wir müssen  
 2142 das so machen, weil es im Paket `dplyr` auch eine Funktion `extract()` gibt, die wir hier nicht anwenden  
 2143 möchten, da sie einen Fehler verursachen würde.

2144 Ein analoges Vorgehen ist auch für Linien und Polygone möglich.

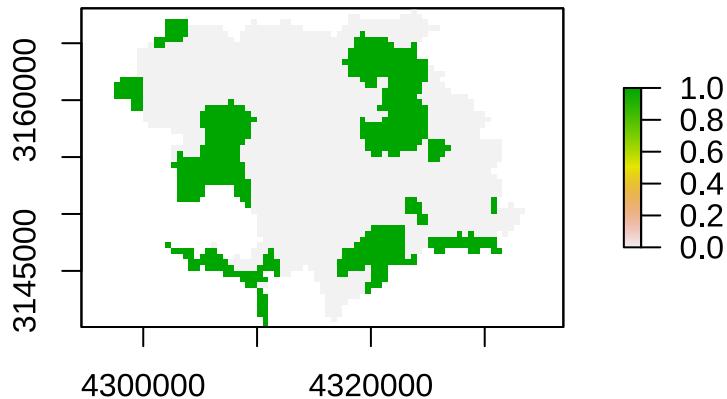
2145 Mit Rastern kann auch einfach gerechnet werden. Wir können z.B. die Seehöhe in Kilometern anstatt Metern  
 2146 berechnen:

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

2147 Auch logische Operationen sind möglich, wenn wir alle Rasterzellen mit einer Seehöhe von mehr als 300 m in  
 2148 Göttingen suchen, dann geht das so:

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

```
plot(dem3)
```



2149

2150 Wenn wir jetzt auf die Werte des Rasters `dem3` zugreifen wollen, geht das mit eckigen Klammern.

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

2151 ## [1] NA NA NA NA NA NA

2152 Das sind erst einmal viele `NA`-Werte für die ganzen Zellen, die außerhalb von Niedersachsen liegen. Aber wir  
 2153 können mit so einem Vektor ganz normal arbeiten und z.B. die Fläche des Landes Niedersachsen die eine  
 2154 Seehöhe von mehr als 500m Seehöhe hat ausrechnen.

```

2154 h <- dem3[]
2155 sum(h, na.rm = TRUE) / sum(!is.na(h))
2156 ## [1] 0.265713

```

2156

#### **Aufgabe 41: Arbeiten mit Rastern**

---

2159 Verwenden Sie den Raster `wald.tif`, der auf einer 10 m Auflösung den Waldanteil jeder Rasterzelle angibt<sup>18</sup>.  
 2160 Der EPSG-Code für das KBS von `wald.tif` ist 3035. Nehmen Sie an, dass wenn der Waldanteil in einer  
 2161 Raster größer als 50 % ist, dass die Rasterzelle als Wald klassifiziert werden kann. Wie viel Prozent des  
 2162 Göttinger Stadtgebietes sind Wald? Wie ändert sich dieser Wert, wenn sie 70 % anstatt 50 % als Schwellenwert  
 2163 für Wald annehmen?

2164

#### **Aufgabe 42: Studiendesign**

---

2167 Mit der Funktion `st_sample()` können Sie innerhalb oder entlang eines Features zufällige Punkte legen. Das  
 2168 Argument `n` steuert die Anzahl Punkte und das Argument `type` wie die Punkte angeordnet werden. Für `type`  
 2169 sind für uns die Werte `type = "random"` (komplett zufällig), `type = "regular"` (regelmäßiger Grid) und  
 2170 `type = "hexagonal"` von Bedeutung (ein hexagonaler Grid, d.h. ein sechseckiger Raster). Unglücklicherweise  
 2171 ist das Ergebnis von `st_sample()` erst eine Geometrie. Um daraus ein vollständiges `sf`-Objekt zu machen  
 2172 und problemlos weiter arbeiten zu können, müssen Sie nocheinmal die Funktion `st_as_sf()` ausführen.  
 2173 Stellen Sie sich vor, dass wir die tatsächliche Waldbedeckung des Göttinger Stadtgebietes **nicht** kennen und wir  
 2174 eine Studie durchführen, um den Anteil des Göttinger Stadtgebietes, der mit Wald bedeckt ist herauszufinden.  
 2175 Erstellen Sie dafür einige unterschiedliche Stichproben (diese können in der Anzahl und Anordnung variieren).  
 2176 Berechnen Sie für jedes Stichprobendesign den Anteil an Wald und ein dazugehöriges Konfidenzintervall  
 2177 (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  $n$   
 2178 die Stichprobengröße). Nehmen Sie an, dass eine Rasterzelle Wald ist, sobald  $> 50\%$  der Rasterzelle mit  
 2179 Wald bedeckt ist.

2180

#### **Aufgabe 43: Räumliche Daten**

---

2183 Verwenden Sie den folgenden Datensatz:

```

set.seed(123)
df1 <- data.frame(
  x = runif(100, 0, 100),

```

<sup>18</sup>Die können hier <https://land.copernicus.eu/pan-european/high-resolution-layers/> für ganz Europa bezogen werden

```

y = runif(100, 0, 100),
kronendurchmesser = runif(100, 1, 15),
art = sample(letters[1:4], 100, TRUE)
)

```

- 2184 1. Erstellen Sie ein `sf`-Objekt aus `df1`.  
 2185 2. Puffern Sie jeden Baum mit seinem Kronendurchmesser.  
 2186 3. Berechnen Sie die Kronenfläche jedes Baumes. *Hinweis: Die Funktion `st_area()` könnte dafür hilfreich sein.*  
 2187 4. Welcher Baum hat die größte Kronenfläche?  
 2188 5. Finden Sie für jede Art, den Baum mit der größten Kronenfläche.

2190

**Aufgabe 44: Arbeiten mit räumlichen Daten**

- 2193 1. Lesen Sie das ESRI Shapefile `goettingen/stadt_goettingen.shp` ein.  
 2194 2. Wie viele Features befinden sind in dem Shapefile?  
 2195 3. Welches Koordinatenbezugssystem (KBS) hat das Shapefile?  
 2196 4. Transformieren Sie das Shapefile in das KBS 3035.  
 2197 5. Erstellen Sie eine neue Spalte `A` in der Sie die Fläche jeder Gemeinde/Stadt speichern.  
 2198 6. Welche Gemeinde/Stadt (Spalte `GEN`) ist am größten?  
 2199 7. Wählen Sie nun nur die Stadt Göttingen aus.

2200

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

- 2203 1. Lesen Sie erneut das ESRI Shapefile `goettingen/stadt_goettingen.shp` ein.  
 2204 2. Lösen sie die Gemeindegrenzen auf (die Funktion `st_union()` könnte hier nützlich sein).  
 2205 3. Wie groß ist das resultierende Feature?

2206 **15 FAQs (Oft gefragtes)**

2207 **15.1 Arbeiten mit Daten**

2208 **15.1.1 Einlesen von Exceldateien**

2209 Mit der Funktion `read_excel()` aus dem Paket `readxl` können Exceldateien direkt in R eingelesen werden.

2210 Ein Export als csv-Datei aus Excel ist nicht notwendig.

## 2211 16 Zusätzliche Aufgaben

2212

### 2213 **Aufgabe 46: Standardisierung**

---

2215 Unter Standardisierung (oder auch z-Transformation) versteht man die Transformation einer Variable, so  
2216 dass sie den Mittelwert 0 und die Varianz 1 hat. Die Formel für die Standardisierung ist

$$x_s = \frac{x - \mu_x}{\sigma_x}$$

2217 wobei  $x$  die Variable ist,  $\mu_x$  ist der Mittelwert von  $x$  und  $\sigma_x$  ist die Standardabweichung von  $x$ .

2218 Standardisieren Sie folgenden Vektor:

```
h <- c(0, 2, 3, 1, 0, 8, 3.4, 9, 6.8, 2.1)
```

2219 Und speichern Sie das Ergebnis in `h_s`. Vergewissern Sie sich, dass die Standardisierung geklappt hat und  
2220 berechnen Sie den Mittelwert und Standardabweichung von `h_s`.

2221

### 2222 **Aufgabe 47: Arbeiten mit logischen Werten**

---

2224 Verwenden Sie nochmals den Vektor mit der Anzahl Rehe, die an unterschiedlichen Fotofallenstandorten  
2225 fotografiert wurden.

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

2226 Für wie viele Standort trifft die Aussage zu  $90 \leq x < 120$ , wobei  $x$  für die Anzahl Fotos an einem Standort  
2227 steht.

2228

### 2229 **Aufgabe 48: Auswählen von Elementen in einem Vektor**

---

2231 Lesen Sie die Datei `bhd_1.txt` ein. Und bearbeiten Sie folgende Aufgaben mit dieser Datei:

- 2232 • Finden Sie den mittleren BHD aller Eichen.
- 2233 • Wie viele Beobachtungen haben Sie für Eichen, Fichten und Buchen?
- 2234 • Finden Sie alle Bäume, die 10, 20, 21, 23, 30, 37, 78, 79, 90, 91, 92 Jahre alt sind.

2235

---

**Aufgabe 49: Arbeiten mit Daten**

---

2238 Wang et al. (2019) haben in einer Fotofallenstudie das Verhalten und die Habitatselektion von Ozeloten  
2239 im brasilianischen Amazonas untersucht. Ziel dieser Übung ist es mit dem Datensatz etwas vertraut zu  
2240 werden, wir werden noch keine ökologischen Analysen durchführen. Mehr zu dem Datensatz erfahren Sie [hier](#).  
2241 Eine etwas angepasste Version des Datensatzes können Sie aus dem StudIP Ordner **daten** (die Datei heißt  
2242 **ozelote.zip**) herunterladen. Speichern Sie die Datei in Ihrem RStudio Projekt und entzippen Sie sie. Der  
2243 Ordner enthält zwei Dateien, für diese Übung brauchen wir lediglich die Datei **ozelote\_standorte.csv**, die  
2244 für jeden Fotofallen Standort einige Kovariaten angibt.

2245 Bearbeiten Sie folgende Aufgaben:

- 2246 1. Lesen Sie die Datei **ozelote\_standorte.csv** in R und speichern Sie das Ergebnis in eine Variable  
2247 **standorte**.
- 2248 2. Wie viele Fotofallenstandorte gab es in der Studie?
- 2249 3. Welcher Standort ist am Höchsten gelegen? Die Spalte **seehoehe** enthält die mittlere Seehöhe.
- 2250 4. Finden Sie alle Standorte, die in unmittelbarer Nähe zu Flüssen sind. Eine Distanz von < 5 m kann als  
2251 Schwellenwert angenommen werden. Die Spalte **dist\_fluss** gibt die Distanz zu Flüssen an.
- 2252 5. Der Datensatz besteht aus verschiedenen Kameras, die jeweils für einen Zeitraum von 12 Tagen in einer  
2253 Region aufgestellt wurden (Spalte **Region**). Erstellen Sie einen Plot, der den Zusammenhang zwischen  
2254 der Region und Seehöhe darstellt.

2255

---

**Aufgabe 50: Base Plots**

---

2258 Erstellen Sie die folgende Beispiellabbildung Schritt für Schritt selbst über Low-Level Funktionen. Die Rohdaten  
2259 finden Sie in den Dateien **abbBeispiel.R** und **ertragstafeldaten.csv**.

- 2260 • Die Wachstumskurve der Region 1 (blau) lautet  $41.45752(1 - \exp(-0.02168x))^{1.61787}$
  - 2261 • Die Wachstumskurve der Region 2 (rot) lautet  $51.11203(1 - \exp(-0.009129x))^{1.202401}$
- 2262 wobei  $x$  das Baumalter in Jahren angegeben ist. Die 3 schwarzen Linien sind auf der Ertragstafel abgelesen.  
2263 Die Beschriftungen der 3 Ertragstafelkurven, sowie des Ausreißers, sind Zusatzaufgaben.

2264

---

**Aufgabe 51: ggplot2 Aufgabe**

---

- 2267 1. Laden Sie den Datensatz **daten/bhd\_1.txt**
- 2268 2. Erstellen Sie ein Streudiagramm. Bilden Sie dabei den BHD gegen das Alter ab, wobei dies als Subplot  
2269 für jedes Affnahmgebiet dargestellt werden sollte.
- 2270 3. Verwenden Sie für jede Baumart eine eigene Farbe.
- 2271 4. Erstellen Sie für jede Baumart einen Boxplot des BHDs.

2272 5. Teilen Sie die Boxplots aus 4) auf jeweils einen Subplot pro Aufnahmegerät auf.

2273

2274 **Aufgabe 52: Anwendungsbeispiel kontrollierter Programmabläufe**

---

- 2276 • Öffnen Sie ein neues, leeres R Skript.  
2277 • Laden Sie die Datei "stichprobe.csv" in eine Variable.

```
str <- read.csv("data/stichprobe.csv", fileEncoding = "UTF-8")
```

- 2278 • Filtern Sie den Data Frame so, dass er nur noch die Baumart "Eiche" enthält. Speichern Sie den  
2279 gefilterten Data Frame in einer NEUEN Variable ab.
- 2280 • Berechnen Sie die deskriptiven Statistiken `mean()`, `sd()`, `median()`, `min()` und `max()` des Kapitels  
2281 "Deskriptive Statistik" für den BHD (des gefilterten Data Frames).
- 2282 • Erstellen Sie ein Histogramm des BHD (ebenfalls mit dem gefilterten Data Frame), zeichnen Sie den  
2283 arithmetischen Mittelwert als horizontale Linie in das Histogramm ein.
- 2284 • Speichern Sie den R Code und kopieren Sie ihn in ein neues R Skript.
- 2285 • Erstellen Sie nun eine Schleife, die alle Statistiken und auch die Abbildung für jede Baumart berechnet.  
2286 Lassen Sie die Statistiken mit `print()` in die Konsole ausgeben.
- 2287 • ZUSATZ: Exportieren Sie die Histogramme (bspw. als PDF). TIPP: Verwenden Sie `paste()` um  
2288 sinnvolle Namen für die Dateien zu erstellen. Machen Sie sich selbst mit der Funktion vertraut.
- 2289 • ZUSATZ: Sie wollen Fehlermeldungen vermeiden. Deshalb programmieren Sie eine bedingte Ausführung,  
2290 um die gesamten statistischen Berechnungen und auch die Abbildung. Führen Sie Ihren gesamten  
2291 Code nur unter der Bedingung aus, dass die Baumart "Ei", "Bu", "Fi", "Kie" oder "Dou" ist. TIPP: Sie  
2292 können den `%in%` Operator verwenden.

2293 **16.1 Arbeiten mit Daten**

2294 Verwenden Sie erneut die Datensätze von Wang et al. (2019) zu Ozeloten in Brasilien für die nachfolgenden  
2295 Übungen.

2296

2297 **Aufgabe 53: Datenzusammenfassen**

---

- 2299 1. Laden Sie die Datei `ozelote_standorte.csv` in R und speichern Sie das Ergebnis in eine Variable  
2300 `standorte`.
- 2301 2. Berechnen Sie die Anzahl an Fotofallen für jede Region. Welche Region weist die meisten Fotofallen  
2302 auf?
- 2303 3. In welcher Region ist die größte Variabilität der Seehöhe zu finden?
- 2304 4. In welchen Regionen beträgt der Anteil an Fotofallen, die < 5m vom nächsten Fluss entfernt sind,  
2305 mindestens 20%?

2306

---

2307 **Aufgabe 54: Datenmanipulation 1**

---

2308

- 2309 1. Laden Sie nun zusätzlich die Datei `ozelote_fanghistorien.csv` und speichern Sie diese in die Variable  
2310 (`fh`). In diesem `data.frame` gibt es für jede Session eine Spalte (V1 bis V10). Eine 1 bedeutet, dass  
2311 mindestens ein Ozelot fotografiert wurde und eine 0 bedeutet, dass kein Ozelot in diesem Zeitraum  
2312 fotografiert wurde. `NA` heißt, dass die Kamera nicht aktiv war.  
2313 2. Wählen Sie nur das 3. Fangereignis (das ist die Spalte V3).  
2314 3. Wie viele Kameras waren beim 3. Fangereignis aktiv?  
2315 4. Vergleichen Sie anhand einer Abbildung, ob sich die Distanz zum Fluss (Spalte `dist_fluss`) zwischen  
2316 Standorten mit Fotos (V3 == 1) und Standorten ohne Fotos (V3 == 0) unterscheidet.

2317

---

2318 **Aufgabe 55: Datenmanipulation 2 (etwas knifflig)**

---

2319

- 2320 1. Verwenden Sie erneut die Daten zu den Fotofallenstandorten und Fanghistorien der Ozelote.  
2321 2. Finden Sie alle Fotofallenstandorte an denen  $\geq 3$  Ozelote fotografiert wurden?  
2322 3. Gibt es einen Zusammenhang zwischen der Häufigkeit an Ozelotfotos (pro Fotofallenstandort) und der  
2323 Distanz zum nächsten Fluss (Spalte `dist_fluss`)? Eine Abbildung ist ausreichend.

## 2324 17 Literatur

- 2325 Ein guter Überblick über viele der angeschnittenen Themen gibt es in dem [R for data science](#), das online frei  
2326 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.  
2327
- 2328 McNamara, Amelia, and Nicholas J Horton. 2018. “Wrangling Categorical Data in r.” *The American Statistician*  
2329 72 (1): 97–104.
- 2330 Wang, Bingxin, Daniel G. Rocha, Mark I. Abrahams, André P. Antunes, Hugo C. M. Costa, André Luis  
2331 Sousa Gonçalves, Wilson Roberto Spironello, et al. 2019. “Habitat Use of the Ocelot (*Leopardus pardalis*)  
2332 in Brazilian Amazon.” *Ecology and Evolution* 9 (9): 5049–62. <https://doi.org/10.1002/ece3.5005>.
- 2333 Wickham, Hadley. 2014. “Tidy Data.” *Journal of Statistical Software* 59 (10). <https://doi.org/10.18637/jss.v059.i10>.
- 2334