

Einführung in R

Stephan Goerigk

2023-05-18

Contents

Über dieses Skript	5
1 Warum ist R so gut?	7
1.1 Open Source	7
1.2 Vielseitigkeit	7
1.3 R Markdown	7
1.4 Transparenz	8
2 R Materialien	9
2.1 Cheat Sheets	9
2.2 Hilfe und Inspiration online	9
2.3 Andere Bücher	9
3 Installation	11
3.1 Installation von R	11
3.2 Installation von RStudio	12
4 Programmaufbau	13
4.1 Die vier RStudio Fenster	13
4.2 R Packages	15
5 Datenformate	19
5.1 Skalar	19
5.2 Vektor	21
5.3 Matrizen und Dataframes	23
6 Daten erstellen	27
6.1 Manuell	27
6.2 Automatisch	28
6.3 Zufällig	28
7 Daten importieren und speichern	29
7.1 Funktionen zur Organisation des Workspace	29
7.2 Arbeitsverzeichnis (Working Directory)	30

7.3	Working Environment	31
7.4	Daten importieren	31
7.5	Daten speichern	36
8	Daten auswählen	39
8.1	Vektor	41
8.2	Dataframe	43
9	Daten Explorieren	47
9.1	Informationen über R Objekte	47
9.2	Deskriptivstatistiken	48
10	Datenmanipulation	51
10.1	Daten hinzufügen	51
10.2	Daten löschen	52
10.3	Variablen umbenennen	54
10.4	Daten verändern	56
10.5	Daten sortieren	59
10.6	Datensätze Aggreggieren	62
10.7	Datensätze transformieren	65
11	Korrelation	67
11.1	Korrelationsmatrix	71

Über dieses Skript

Liebe Studierende,

dieses Skript soll Sie in die grundlegenden Analysewerkzeuge in R einführen, von der grundlegenden Kodierung und Analyse bis hin zur Datenverarbeitung, dem Plotten und der statistischen Inferenz.

Wenn R Ihre erste Programmiersprache ist, ist das völlig in Ordnung. Wir gehen alles Schritt für Schritt gemeinsam durch. Die Techniken in diesem Skript sind zwar auf die meisten Datenanalyseprobleme anwendbar, da wir jedoch aus der Psychologie kommen, werde ich den Kurs auf die Lösung von Analyseproblemen ausrichten, die in der psychologischen Forschung häufig auftreten.

Ich wünsche Ihnen Viel Erfolg und Spaß!

Chapter 1

Warum ist R empfehlenswert?

1.1 Open Source

Im Gegensatz zu SPSS, Matlab, Excel und JMP ist R kostenlos und verfügt über eine große Unterstützergemeinschaft. Das bedeutet, dass eine große Gemeinschaft von R-Programmierer:innen ständig neue R-Funktionen und -Pakete entwickelt und verbreitet. Wenn Sie jemals eine Frage dazu haben, wie man etwas in R implementiert, wird eine schnelle Suche im Internet Sie praktisch jedes Mal zur Antwort führen.

1.2 Vielseitigkeit

R ist sehr vielseitig. Sie können R für alles verwenden, von der Berechnung einfacher zusammenfassender Statistiken über die Durchführung komplexer Simulationen bis hin zur Erstellung von Diagrammen. Wenn Sie sich eine statistische Aufgabe vorstellen können, können Sie sie mit ziemlicher Sicherheit in R implementieren.

1.3 RMarkdown

Mit RStudio, einem Programm, das Sie beim Schreiben von R-Code unterstützt, können Sie mittels RMarkdown einfach und nahtlos R-Code, Analysen, Diagramme und geschriebenen Text zu eleganten Dokumenten an einem Ort kombinieren. Tatsächlich wurde dieses gesamte Skript (Text, Formatierung,

Diagramme, Code...) in RStudio mit R Markdown geschrieben. Mit RStudio müssen Sie sich nicht mehr mit zwei oder drei Programmen herumschlagen, z. B. Excel, Word und SPSS, wo Sie die Hälfte Ihrer Zeit mit dem Kopieren, Einfügen und Formatieren von Daten, Bildern und Tests verbringen, sondern können alles an einem Ort erledigen, so dass nichts mehr falsch gelesen, getippt oder vergessen wird.

1.4 Transparenz

In R durchgeführte Analysen sind transparent, leicht weiterzugeben und reproduzierbar. Wenn Sie einen SPSS-Benutzer fragen, wie er eine bestimmte Analyse durchgeführt hat, wird er sich ggf. nicht daran erinnern, was er vor Monaten oder Jahren tatsächlich getan hat. Wenn Sie eine R-Anwender:in (der/die gute Programmiertechniken verwendet) fragen, wie er/sie eine Analyse durchgeführt hat, sollte er/sie Ihnen immer den genauen Code zeigen können, den er/sie verwendet hat. Das bedeutet natürlich nicht, dass er/sie die richtige Analyse verwendet oder sie korrekt interpretiert hat, aber mit dem gesamten Originalcode sollten etwaige Probleme völlig transparent sein! Dies ist eine Grundvoraussetzung für offene, replizierbare Forschung.

Chapter 2

R Materialien

2.1 Cheat Sheets

In diesem Skript werden Sie viele neue Funktionen kennenlernen. Wäre es nicht schön, wenn jemand ein Wörterbuch mit vielen gängigen R-Funktionen erstellen würde? Im Folgenden finden Sie eine Tabelle mit einigen der Funktionen, die ich empfehle. Ich empfehle Ihnen dringend, diese auszudrucken und die Funktionen zu markieren, wenn Sie sie lernen!

[Link zum Base R Cheat Sheet](#)

[Link zu den R Studio Cheat Sheets](#)

Insbesondere die Cheat Sheets zu ggplot2 und dplyr sind zu empfehlen (RStudio Cheat Sheets S.2).

2.2 Hilfe und Inspiration online

Eine Internetsuche nach einem spezifischen R Problem bringt Sie (fast) immer an Ihr Ziel. Häufig findet man gute Antworten in den github Hilfsmaterialien einzelner Pakete, auf den Community Seiten von R Studio und in den Foren von stackoverflow.

2.3 Andere Bücher

Die Inhalte dieser Bücher sind nicht prüfungsrelevant.

Es gibt viele, viele gute Bücher über R. Hier sind zwei, die ich empfehlen kann (von denen eines sogar umsonst ist):

Discovering Statistics with R von Field, Miles and Field

R for Data Science von Garrett Grolemund and Hadley Wickham

Chapter 3

Installation

Um R benutzen zu können, müssen wir zwei Softwarepakete herunterladen:

- **R**
- **RStudio**

R ist die Programmiersprache, mit der wir arbeiten. R-Studio ist eine Benutzeroberfläche, die uns das Programmieren mit R ungemein erleichtert.

3.1 Installation von R

Um R zu installieren, klicken Sie auf den Ihrem Betriebssystem entsprechenden Link und folgen Sie die Anleitungen.

Operating System	Link
Windows	http://cran.r-project.org/bin/windows/base/
Mac	http://cran.r-project.org/bin/macosx/

Nach dieser Installation haben Sie bereits die volle Funktionalität des Programms. Sie werden jedoch feststellen, dass beinahe alle R-Nutzer RStudio zum programmieren nutzen, da dieses eine leichter nutzbare Oberfläche hat. Tatsächlich müssen Sie nach der Installation von RStudio das R Basisprogramm nie wieder öffnen.

3.2 Installation von RStudio

Bitte installieren Sie dann RStudio - das Programm, über welches wir auf R zugreifen und mit dem wir unsere Skripte schreiben.

Um RStudio zu installieren, klicken Sie auf diesen Link und befolgen Sie die Anleitungen: <http://www.rstudio.com/products/rstudio/download/>

Chapter 4

Programmaufbau

4.1 Die vier RStudio Fenster

Wenn Sie RStudio öffnen, sehen Sie vier Fenster, wie in der folgenden Abbildung dargestellt:

Wenn Sie mögen, können Sie die Reihenfolge der Fenster in den RStudio Einstellungen verändern. Sie können die Fenster auch verstecken (Minimieren/Maximieren Symbol an der oberen rechten Ecke jedes Fensters) oder ihre Größe verändern, indem die sie ihre Grenzbalken anklicken und verschieben.

Lassen Sie uns jetzt schauen, was genau die Funktion jedes der Fenster ist:

4.1.1 Source - Ihr Schreibblock für Code

Im Source Fenster erstellen und bearbeiten Sie “R-Skripte” - Ihre Codesammlungen. Keine Sorge, R-Skripte sind nur Textdateien mit der Erweiterung “.R”. Wenn Sie RStudio öffnen, wird automatisch ein neues unbenanntes Skript gestartet. Bevor Sie mit der Eingabe eines unbenannten R-Skripts beginnen, sollten Sie die Datei immer unter einem neuen Dateinamen speichern (z.B. “Mein_RScript.R”). Wenn Ihr Computer während der Arbeit abstürzt, steht Ihr Code in R zur Verfügung, wenn Sie RStudio erneut öffnen.

Sie werden feststellen, dass R beim Schreiben des Skripts den Code während der Eingabe nicht tatsächlich ausführt. Damit R Ihren Code tatsächlich ausführt, müssen Sie den Code zunächst an die Konsole “senden” (wir werden im nächsten Abschnitt darüber sprechen).

Es gibt viele Möglichkeiten, Ihren Code aus dem Skript an die Konsole zu senden. Die langsamste Methode ist das Kopieren und Einfügen. Schneller geht es, wenn

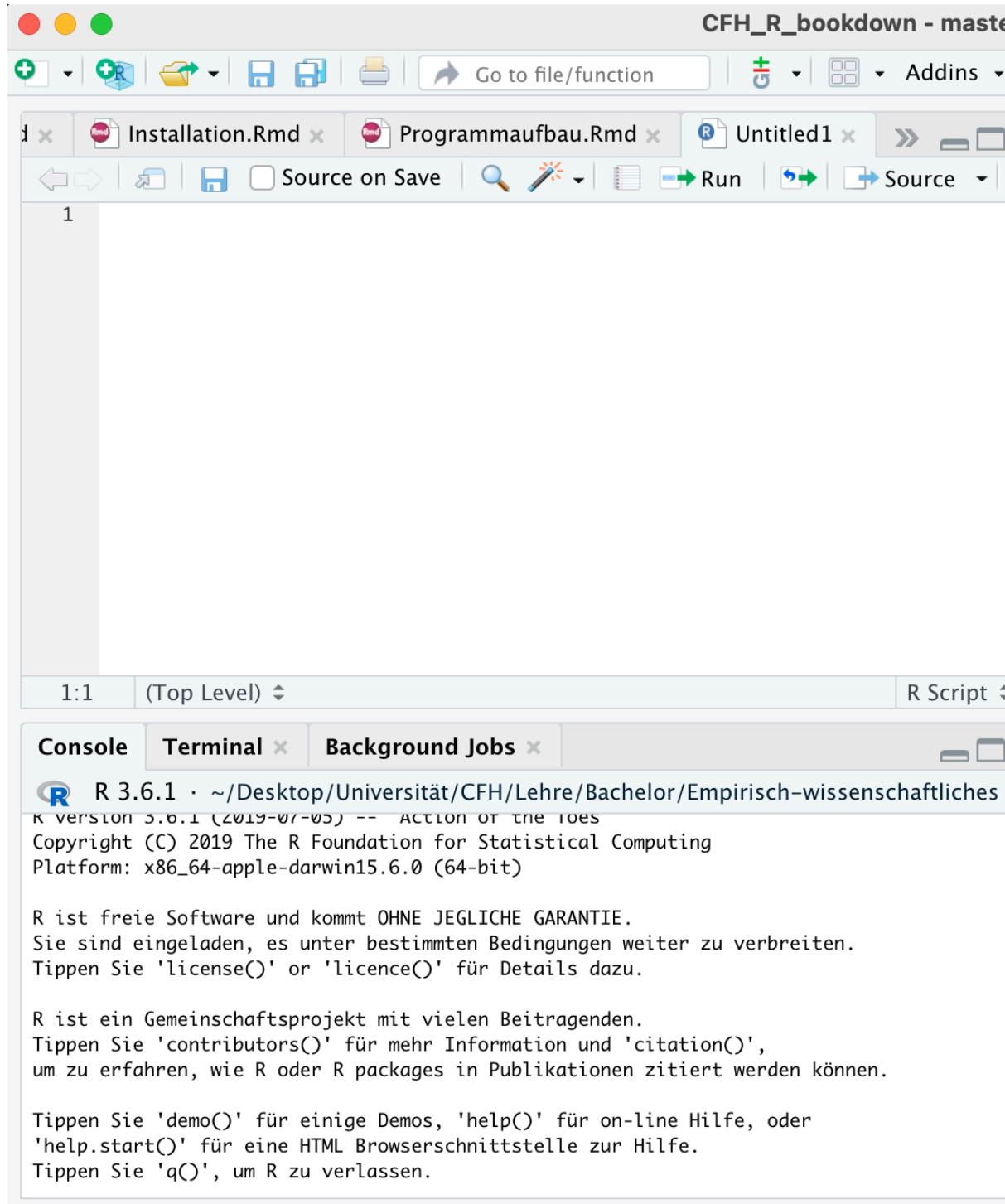


Figure 4.1: Die vier RStudio Fenster

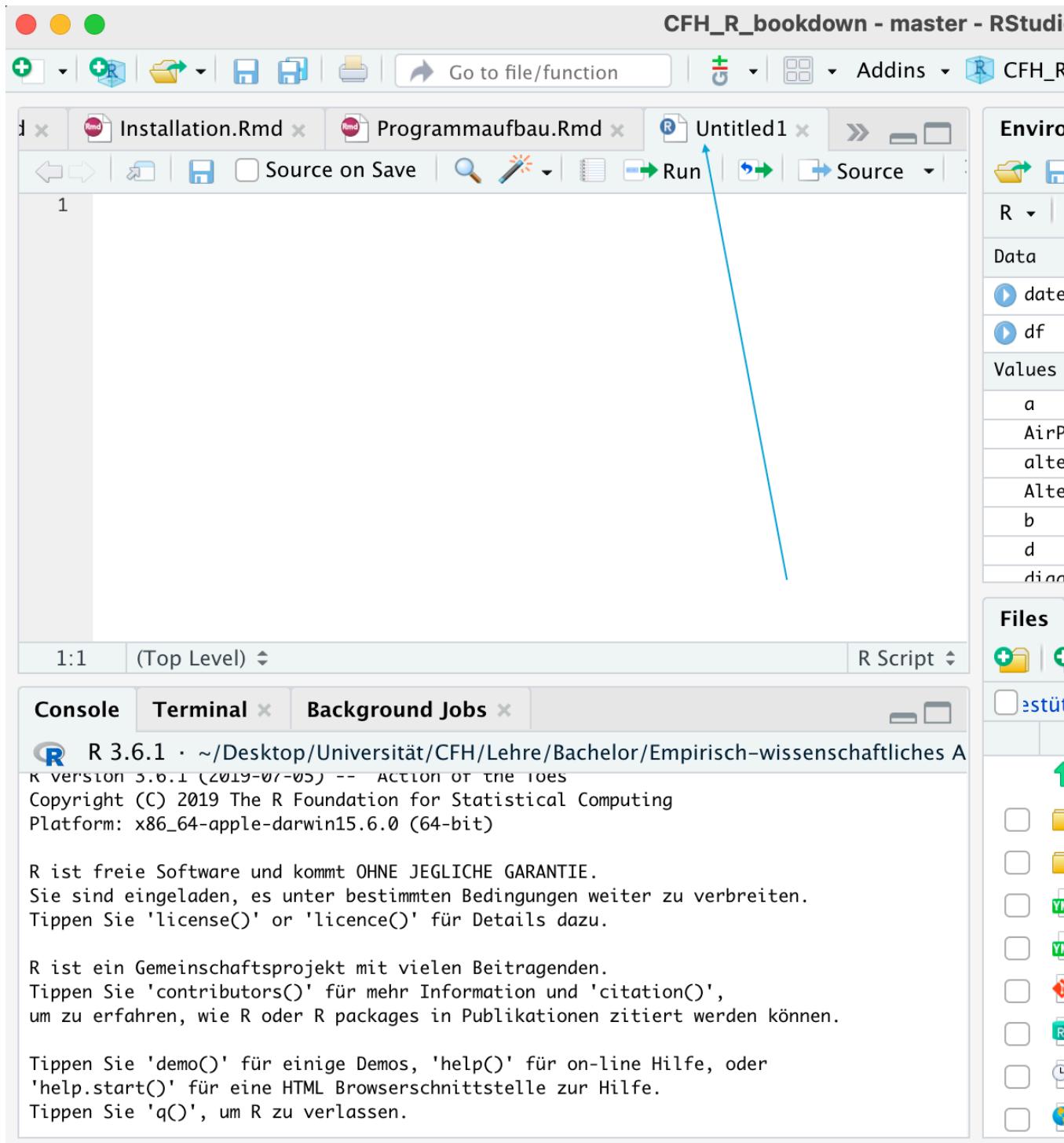


Figure 4.2: Die vier RStudio Fenster

Sie den Code, den Sie auswerten möchten, markieren und auf die Schaltfläche “Run” oben rechts in der Quelle klicken. Alternativ können Sie auch die Tastenkombination “Command + Return” auf dem Mac oder “Control + Enter” auf dem PC verwenden, um den gesamten markierten Code an die Konsole zu senden.

4.1.2 Konsole - Das Herzstück von R

Die Konsole ist das Herzstück von R. Hier führt R den Code aus. Am Anfang der Konsole sehen Sie das Zeichen “>”. Dies ist eine Eingabeaufforderung (sog. “Prompt”), die Ihnen mitteilt, dass R bereit für neuen Code ist. Sie können direkt nach dem Prompt > Code in die Konsole eingeben und erhalten sofort eine Antwort. Wenn Sie zum Beispiel 2+2 in die Konsole eingeben und die Eingabetaste drücken, werden Sie sehen, dass R sofort eine Ausgabe von 4 liefert.

2+2

[1] 4

Versuchen Sie, 2+2 zu berechnen, indem Sie den Code direkt in die Konsole eingeben - und dann Enter drücken. Sie sollten das Ergebnis [1] 4 sehen. Machen Sie sich keine Gedanken über die [1], dazu kommen wir später.

Geben Sie denselben Code in das Skript ein und senden Sie ihn an die Konsole, indem Sie den Code markieren und auf die Schaltfläche “Run” in der oberen rechten Ecke des Quelltextfensters klicken. Alternativ können Sie auch die Tastenkombination “Command + Return” auf dem Mac oder “Control + Enter” unter Windows verwenden.

Tipp: Wie Sie sehen, können Sie Code entweder über das Skript oder durch direkte Eingabe in die Konsole ausführen. In 99% der Fälle sollten Sie jedoch das Skript und nicht die Konsole verwenden. Der Grund dafür ist ganz einfach: Wenn Sie den Code in die Konsole eingeben, wird er nicht gespeichert (obwohl Sie in Ihrem Befehlsverlauf nachsehen können). Und wenn Sie beim Eingeben von Code in die Konsole einen Fehler machen, müssen Sie alles noch einmal von vorne eingeben. Stattdessen ist es besser, den gesamten Code in das Skript zu schreiben. Wenn Sie bereit sind, einen Code auszuführen, können Sie ihn mit “Run” an die Konsole senden.

4.1.3 Environment/History - Das Gedächtnis von R

In dem Tab “Environment” dieses Bereichs werden die Namen aller Datenobjekte (wie Vektoren, Matrizen und Datenrahmen) angezeigt, die Sie in Ihrer aktuellen R-Session definiert haben. Sie können auch Informationen wie die Anzahl der Spalten und Zeilen in Datensätzen sehen. Der Tab enthält auch einige

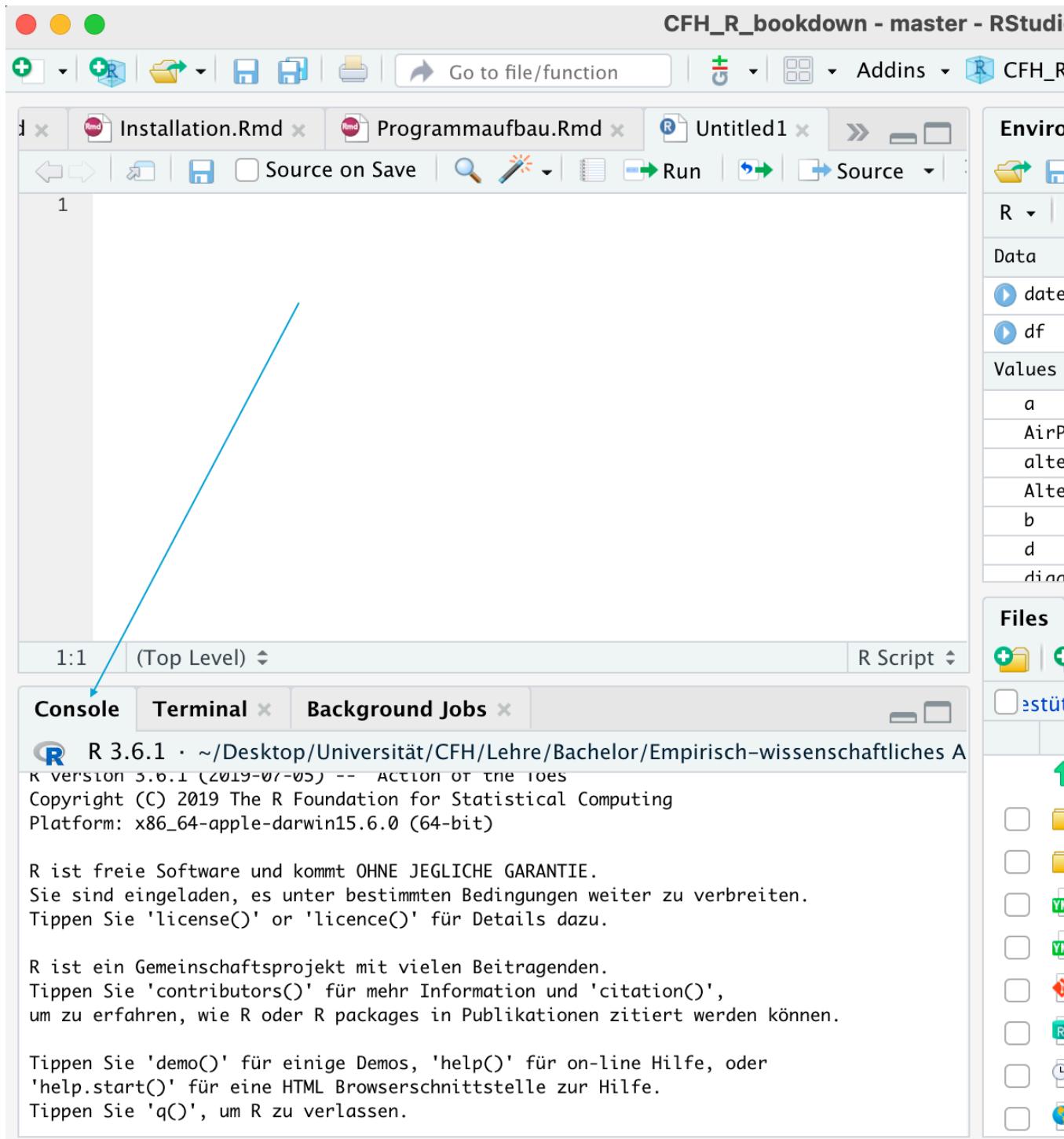


Figure 4.3: Die vier RStudio Fenster

anklickbare Aktionen wie “Datensatz importieren”, wodurch eine grafische Benutzeroberfläche (GUI) für wichtige Daten in R geöffnet wird.

Der Tab “History” dieses Bereichs zeigt Ihnen einfach eine Sammlung aller Befehle an, die Sie zuvor in der Konsole ausgewertet haben. Wenn man mit Skripten arbeitet, schaut man sich diese allerdings relativ selten an.

Wenn Sie sich mit R besser auskennen, werden Sie das Fenster Environment/History vielleicht nützlich finden. Aber für den Moment können Sie es einfach ignorieren. Wenn Sie Ihren Bildschirm entrümpeln wollen, können Sie das Fenster auch einfach minimieren, indem Sie auf die Schaltfläche Minimieren oben rechts im Fenster klicken.

4.1.4 Files/Plots/Packages/Help/Viewer - Interaktion von R mit Dateien

Die Tabs Files/Plots/Packages/Help/Viewer zeigen Ihnen viele hilfreiche Informationen. Schauen wir uns die einzelnen Registerkarten im Detail an:

1. Files - Der Tab “Files” gibt Ihnen Zugriff auf das Dateiverzeichnis Ihrer Festplatte. Dateien, die Sie in Ihrem R Projekt benutzen, liegen in der Regel in einem von Ihnen definierten Arbeitsverzeichnis. Wir werden in Kürze ausführlicher über Arbeitsverzeichnisse sprechen.
2. Plots - Das Plots-Panel zeigt (keine große Überraschung) alle Ihre Plots an.
3. Pakete - Zeigt eine Liste aller auf Ihrer Festplatte installierten R-Pakete und gibt an, ob sie derzeit geladen sind oder nicht. Pakete, die in der aktuellen Sitzung geladen sind, sind markiert, während Pakete, die installiert, aber noch nicht geladen sind, nicht markiert sind. Auf die Pakete gehen wir im nächsten Abschnitt näher ein.
4. Hilfe - Hilfemenü für R-Funktionen. Sie können entweder den Namen einer Funktion in das Suchfenster eingeben oder den Code `?function.name` in der Konsole verwenden, um nach einer Funktion mit dem Namen `function.name` zu suchen:

```
?hist # Wie funktioniert die Histogrammfunktion?  
?t.test # Wie funktioniert der t-Test?
```

4.2 R Packages

Wenn Sie R zum ersten Mal herunterladen und installieren, installieren Sie die Base R Software. Base R enthält die meisten Funktionen, die Sie täglich

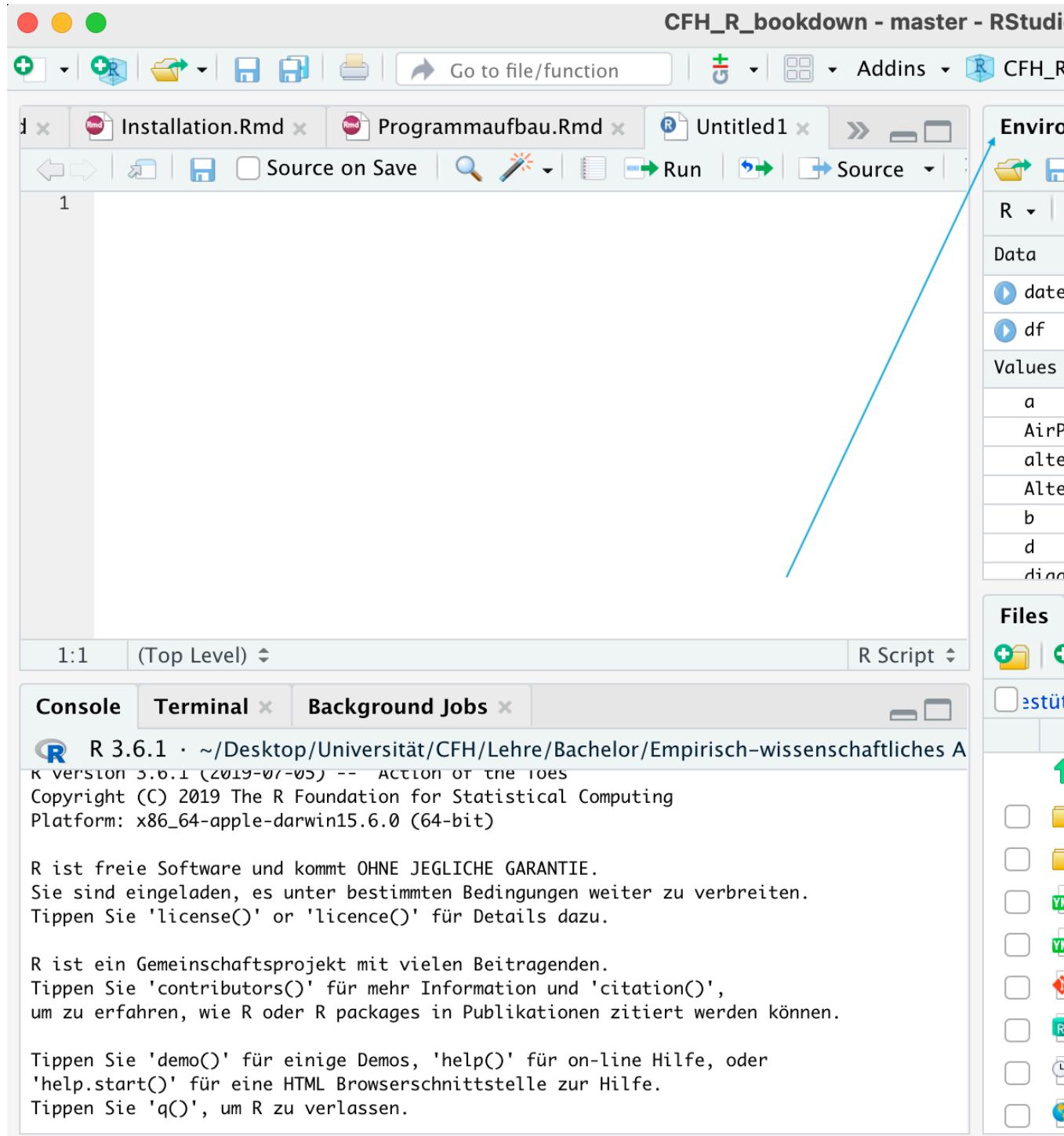


Figure 4.4: Die vier RStudio Fenster

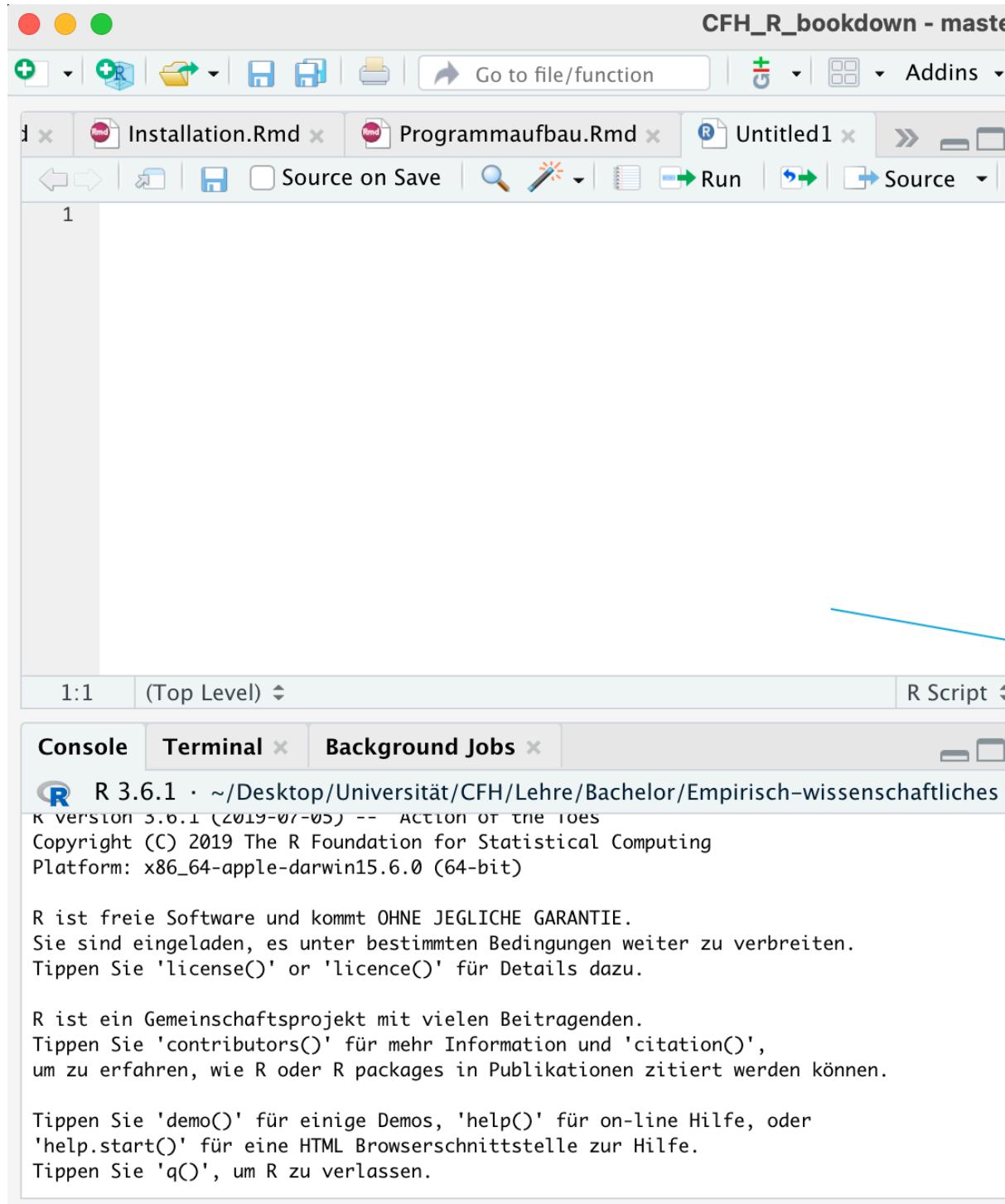


Figure 4.5: Die vier RStudio Fenster

verwenden werden, wie `mean()` und `hist()`. Allerdings werden hier nur Funktionen angezeigt, die von den ursprünglichen Autoren der Sprache R geschrieben wurden. Wenn Sie auf Daten und Code zugreifen möchten, die von anderen Personen geschrieben wurden, müssen Sie diese als “Package” installieren. Ein R-Package ist einfach ein Bündel von Funktionen (also bereits geschriebener Code), die in einem übersichtlichen Paket gespeichert sind.

Ein Paket ist wie eine Glühbirne. Um es nutzen zu können, müssen Sie es zunächst in Ihr Haus (d.h. auf Ihren Computer) bestellen, indem Sie es installieren. Wenn Sie ein Paket einmal installiert haben, brauchen Sie es nie wieder zu installieren. Jedes Mal, wenn Sie das Paket tatsächlich verwenden wollen, müssen Sie es jedoch einschalten, indem Sie es laden. Und so geht’s:

4.2.1 R Packages installieren

Ein Paket zu installieren bedeutet einfach, den Paketcode auf Ihren Computer herunterzuladen. Die gängigste Methode ist das Herunterladen aus dem Comprehensive R Archive Network (CRAN).

Um ein neues R-Paket von CRAN zu installieren, können Sie einfach den Code `install.packages("name")` ausführen, wobei “name” der Name des Pakets ist.

Um zum Beispiel das Paket `ggplot2` herunterzuladen, welches wir oft zum Erstellen von Graphen verwenden, geben Sie ein:

```
# install.packages("ggplot2")
```

4.2.2 R Packages laden

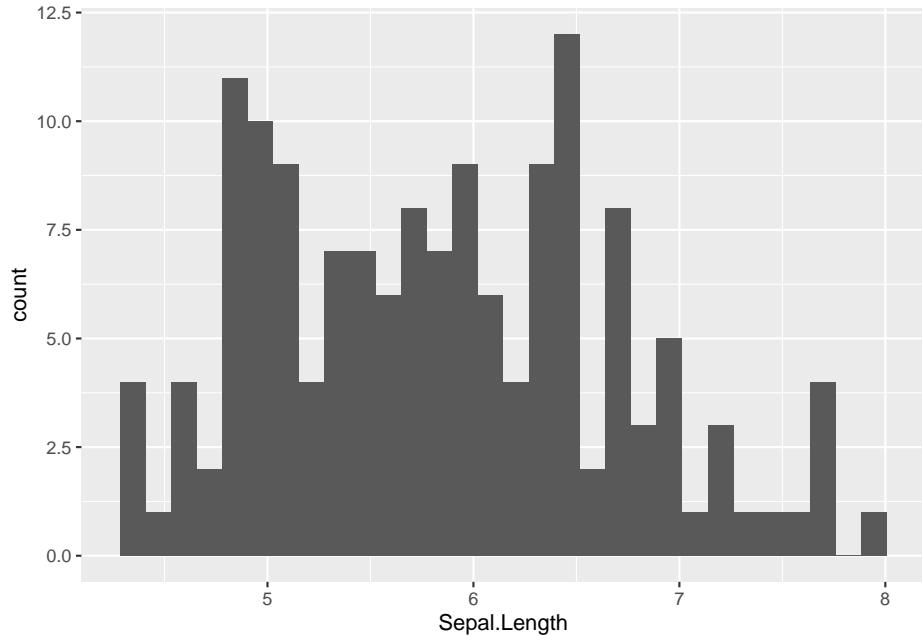
Sobald Sie ein Paket installiert haben, befindet es sich auf Ihrem Computer. Aber nur weil es auf Ihrem Computer ist, bedeutet das nicht, dass R bereit ist, es zu benutzen. Wenn Sie etwas wie eine Funktion oder einen Datensatz aus einem Paket verwenden wollen, müssen Sie *immer* zuerst das Paket in Ihrer R-Sitzung *laden*. Genau wie bei einer Glühbirne müssen Sie sie einschalten, um sie zu benutzen!

Um ein Paket zu laden, verwenden Sie die Funktion `library()`. Nachdem wir zum Beispiel das Paket `ggplot2` installiert haben, können wir es mit `library("ggplot2")` laden:

```
# Laden des "ggplot2" Paketts, damit wir es benutzen können!
# Pakete müssen zu Beginn jeder R Session neu geladen werden!
library("ggplot2")
```

Jetzt, wo Sie das Paket `ggplot2` geladen haben, können Sie jede seiner Funktionen benutzen (hier die Funktion `ggplot`, um einen Graph zu erstellen)!

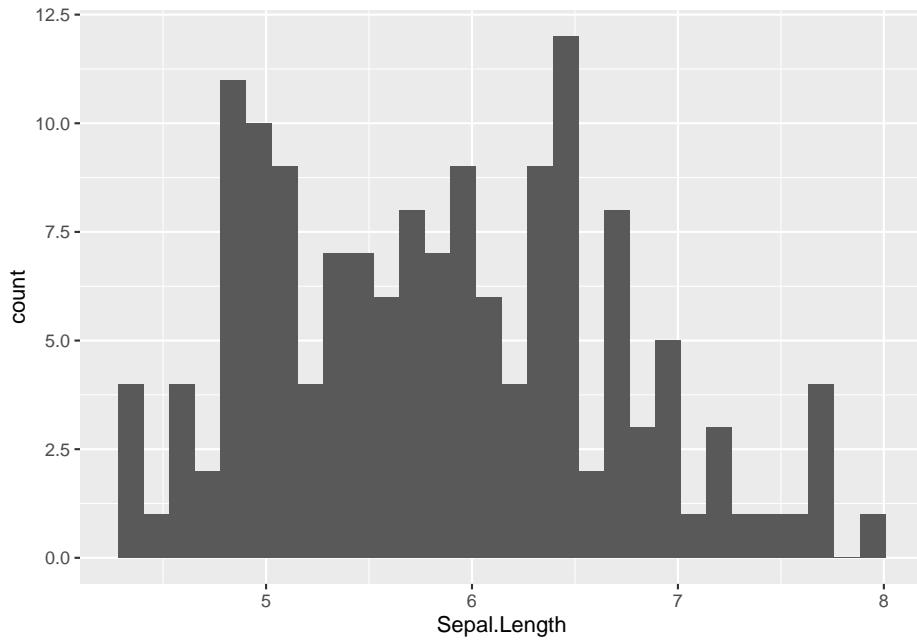
```
ggplot(data = iris, aes(x = Sepal.Length)) +
  geom_histogram()
```



Pakete müssen zu Beginn jeder R Session neu geladen werden. Deswegen schreiben wir in der Regel ganz an den Anfang unseres Skripts gleich mehrere Zeilen, mit `library()` Befehlen für alle R Pakete, die wir für unsere Analyse benötigen werden.

In R gibt es eine Möglichkeit, ein Paket vorübergehend zu laden, ohne die Funktion `library()` zu verwenden. Um dies zu tun, können Sie einfach die Notation `package::funktion` verwenden. Diese Notation sagt R einfach, dass es das Paket nur für diesen einen Codeabschnitt laden soll. Zum Beispiel könnte ich die Funktion `ggplot` aus dem Paket `ggplot2` wie folgt verwenden:

```
ggplot2::ggplot(data = iris, aes(x = Sepal.Length)) +
  geom_histogram()
```



Ein Vorteil der Notation “`package::function`” ist, dass für jeden, der den Code liest, sofort klar ist, welches Paket die Funktion enthält. Ein Nachteil ist jedoch, dass Sie, wenn Sie eine Funktion aus einem Paket häufig verwenden, gezwungen sind, den Paketnamen ständig neu einzugeben. Sie können jede Methode verwenden, die für Sie sinnvoll ist.

Chapter 5

Datenformate

5.1 Skalar

Der einfachste Objekttyp in R ist der **Skalar**. Ein Skalar Objekt ist einfach nur ein einzelner Wert, z.B. eine Zahl oder ein Wort.

Hier sind einige Beispiele für numerische Skalar Objekte:

```
# Examples of numeric scalars
a <- 100
b <- 3 / 100
c <- (a + b) / b
```

Skalare müssen nicht numerisch sein, sondern können auch Worte beinhalten. Wortobjekte heißen in R **characters** (engl. strings). In R schreibt man Worte immer in Anführungszeichen "". Hier sind einige Beispiele für character Skalare:

```
# Beispiele für character Skalare
d <- "Psychologe"
e <- "Zigarre"
f <- "Haben Psychologen wirklich alle Bärte und rauchen Zigarre?"
```

Wie Sie sich vermutlich vorstellen können, behandelt R numerische und character Skalare unterschiedlich. Zum Beispiel lassen sich mit numerischen Skalaren grundlegende arithmetische Operationen durchführen (Addition, Subtraktion, Multiplikation...) – das funktioniert mit character Skalaren nicht. Wenn Sie dennoch probieren numerische Operationen auf character Skalare anzuwenden, bekommen Sie eine Fehlermeldung, so wie diese:

```
a = "1"
b = "2"
a + b
```

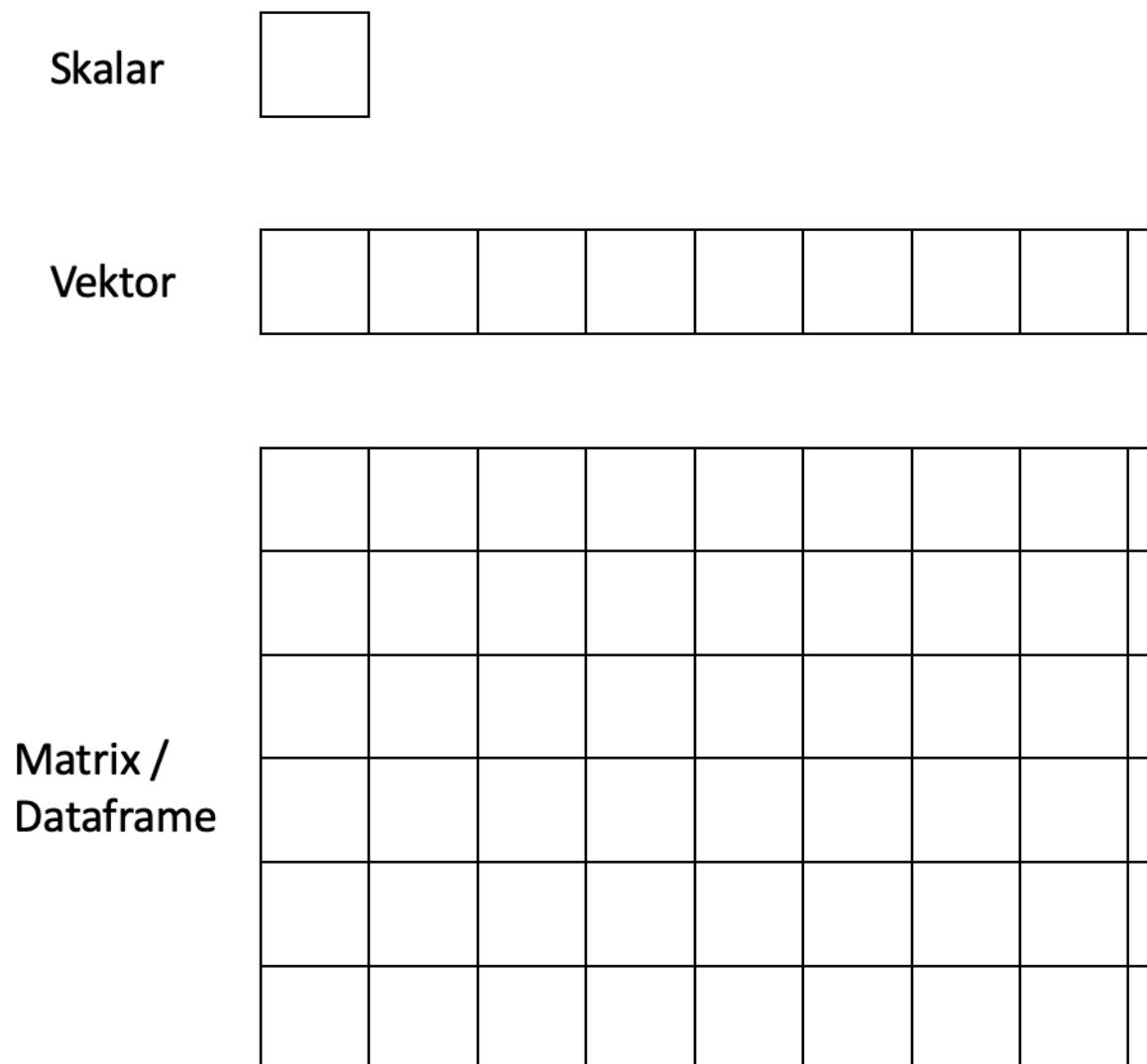


Figure 5.1: Skalar, Vektor, Matrix

“Fehler in $a + b$: nicht-numerisches Argument für binären Operator”

5.2 Vektor

Machen wir weiter mit **Vektoren**. Ein Vektor Objekt ist einfach eine Kombination mehrerer Skalare in einem einzelnen Objekt (z.B. eine Zahlen- oder Wortreihe). Zum Beispiel könnten die Zahlen von 1-10 in einen Vektor mit der Länge 10 kombiniert werden. Oder die Buchstaben des Alphabets könnten in einen Vektor mit der Länge 26 gespeichert werden. Genau wie Skalare, können Vektoren numerisch oder characters sein (Aber nicht beides auf einmal!)

Die einfachste Art einen Vektor zu erstellen ist mit der **c()** Funktion. Das **c** steht für “concatenate”, was auf Englisch so viel heißt wie “zusammenbringen”. Die **c()** Funktion nimmt mehrere Skalare als Input und erstellt einen Vektor, der diese Objekte enthält.

Wenn **mc()** benutzt, muss man immer ein **Komma** zwischen die Objekte setzen (Skalare oder Vektoren), die man kombinieren möchte.

Lassen Sie uns die **c()** Funktion nutzen um einen Vektor zu erstellen der **a** heißt und die Zahlen von 1 bis 7 enthält

```
a = c(1, 2, 3, 4, 5, 6, 7)
# Das Ergebnis ausgeben
a
```

```
## [1] 1 2 3 4 5 6 7
```

Sie können auch character Vektoren erstellen, indem Sie die **c()** auf einzelne character Skalare Funktion anwenden:

```
char.vec = c("Freud", "Wundt", "Bandura", "Watson", "Jung")
# Das Ergebnis ausgeben
char.vec
```

```
## [1] "Freud"    "Wundt"    "Bandura"   "Watson"   "Jung"
```

5.2.1 Vektor Typen

Vektoren sind ein zentrales Element von R. Ein Vektor kann Zahlen, Buchstaben oder logische Werte enthalten, aber niemals eine Kombination

Der Vektor ist die Entsprechung der **Variable** und die Skalare, aus denen der Vektor besteht, sind die **Merkmalsausprägungen** der Variable.

5.2.2 Faktor Variablen

Wir haben bereits gelernt, wie man einen Vektor aus character Objekten erstellt. Manchmal brauchen wir in R jedoch Variablen, die nicht nur Worte enthalten, sondern dem Programm mitteilen, dass es sich um feste Gruppen oder **Kategorien** handelt. Es geht also nicht nur um eine “Sammlung” von Worten (z.B. Nachnamen von Probanden), sondern um festgelegte Analyseeinheiten. Solche Variablen heißen in R **factor**.

In einer factor Variable ist jeder Kategorie eine Zahl zugeordnet (z.B. 1 = männlich, 2 = weiblich).

Um Faktor Variablen zu erstellen, machen wie einen Vorgang, den man **Kodieren** nennt und das geht so:

Wir haben einen Vektor mit Codes 1 und 2 für männlich und weiblich vorliegen:

```
geschlecht = c(1, 2, 2, 1, 2)
# Das Ergebnis ausgeben
geschlecht
```

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

In dieser Form erkennt R diesen Vektor als numerische Variable. Um Sie in einen Faktor umzuwandeln, definieren wir die Zahlen (1 und 2) als **levels** des Faktors und geben dann jedem level einen Namen (**labels**):

```
geschlecht = factor(geschlecht, levels = c(1,2), labels = c("männlich", "weiblich"))
# Das Ergebnis ausgeben
geschlecht
```

```
## [1] männlich weiblich weiblich männlich weiblich
## Levels: männlich weiblich
```

Das Ergebnis ist eine codierte Faktorvariable. Wenn wir Sie uns ausgeben lassen erhalten wir unter den Merkmalsausprägungen eine Liste mit den einzelnen Kategorien (levels) des Faktors.

R wird uns für Faktoren alle Ergebnisse nach der **Reihenfolge** der levels anzeigen. Wenn wir keine Faktorvariable haben, sondern eine character Variable funktioniert die Reihenfolge immer alphabetisch.

5.2.3 Vektor Indizierung

Manchmal möchten wir wieder einen einzelnen Skalar auswählen, der als Teil von einem Vektor gespeichert ist. Diese **Auswahl** eines Einzelements nennt man **Indizierung**. Die Auswahl eines kleineren Objekts aus einem größeren Objekt funktioniert in R immer mit `[]`.

Benötigen wir aus einem Vektor z.B. genau den 3. Skalar, schreiben wir einfach eine 3 in eckige Klammern hinter den Vektor.

```
char.vec = c("Freud", "Wundt", "Bandura", "Watson", "Jung")
# Das Ergebnis ausgeben
char.vec[5]

## [1] "Jung"
```

5.3 Matrizen und data.frames

In der Psychologie beobachten wir für unsere Studien fast immer mehr als eine Variable. Wir könnten diese alle in einzelnen Vektoren speichern und uns die Objektnamen merken. Z.B.

```
Name = c("Max", "Maja", "Mia", "Moritz", "Markus")
Alter = c(20, 31, 25, 34, 51)
Diagnose = c("Depression", "Zwangsstörung", "Depression", "Soziale Phobie", "Depression")
```

5.3.1 Erstellen von Datenmatrizen

Praktischer ist es, die einzelnen Vektoren in Tabellenform zu speichern, der **Datenmatrix**. In R heißen Datenmatrizen `data.frame`. Wir können die Vektoren folgendermaßen zu einem `data.frame` kombinieren:

```
df = data.frame(Name, Alter, Diagnose)
# Das Ergebnis ausgeben
df

##      Name Alter   Diagnose
## 1    Max    20 Depression
## 2   Maja    31 Zwangsstörung
## 3    Mia    25 Depression
## 4 Moritz   34 Soziale Phobie
## 5 Markus   51 Depression
```

Wie in jeder Datenmatrix entsprechen die **Zeilen** den einzelnen Personen (Fällen) und die **Spalten** den Variablen.

R bezeichnet Zeilen und Spalten als **rows** und **columns**. Wollen wir z.B. wissen, wie viele Zeilen der `data.frame` hat, können wir `nrow()` benutzen. Für die Anzahl der Spalten nehmen wir `ncol()`:

```
nrow(df)
```

```
## [1] 5
```

```
ncol(df)
```

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

Wenn wir die einzelnen Vektoren nicht bereits vorher definiert haben, können wir auch alles in einem Schritt machen. Das Ergebnis ist das gleiche:

```
df = data.frame("Name" = c("Max", "Maja", "Mia", "Moritz", "Markus"),
                "Alter" = c(20, 31, 25, 34, 51),
                "Diagnose" = c("Depression", "Zwangsstörung", "Depression", "Soziale Phobie", "Depression"))
# Das Ergebnis ausgeben
df
```

	Name	Alter	Diagnose
## 1	Max	20	Depression
## 2	Maja	31	Zwangsstörung
## 3	Mia	25	Depression
## 4	Moritz	34	Soziale Phobie
## 5	Markus	51	Depression

Wollen wir wieder eine einzelne Variable aus dem Datensatz benutzen, können wir diese über das \$ Zeichen anwählen:

```
df$Alter
```

```
## [1] 20 31 25 34 51
```

5.3.2 Indizierung

Wollen wir aus dem data.frame wieder einzelne Elemente verwenden, nutzen wir wieder die Indizierung. Auch hier brauchen wir die []. Da wir im data.frame Zeilen und Spalten haben, brauchen wir eine Möglichkeit, beides auszuwählen, wie ein Cursor der von links nach rechts, bzw. von oben nach unten läuft.

Wir trennen dafür unsere [] mit einem Komma [,]. Alles, was **links vom Komma** steht, bezieht sich auf Zeilen alles **rechts vom Komma** bezieht sich auf Spalten.

Lassen Sie uns einmal die Zelle in der 1. Zeile (also die 1. Person) und der 3. Variable auswählen:

```
df[1,3]
```

```
## [1] Depression
## Levels: Depression Soziale Phobie Zwangsstörung
```

Lassen wir die Zahl vor dem Komma weg, bekommen wir alle Werte aus der Spalte:

```
df[,3]
```

```
## [1] Depression      Zwangsstörung  Depression      Soziale Phobie Depression
## Levels: Depression Soziale Phobie Zwangsstörung
```

Lassen wir die Zahl nach dem Komma weg, bekommen wir alle Werte aus der Reihe:

```
df[1,]
```

```
##   Name Alter Diagnose
## 1 Max    20 Depression
```


Chapter 6

Daten erstellen

6.1 Manuell

Die manuelle Eingabe von Daten erfolgt über die `c()` Funktion. Mit ihrer Hilfe können wir Skalare zu Vektoren verbinden...

```
a = c(1, 2, 4, 6, 1)
```

...mehrere Vektoren aneinanderhängen...

```
a = c(1, 2, 4, 6, 1)
b = c(2, 3)
c = c(a, b)
c
```

```
## [1] 1 2 4 6 1 2 3
```

...und Vektoren gleicher Länge zu data.frames kombinieren:

```
daten = data.frame(aufmerksamkeit = c(58, 46, 29, 51),
                    gedaechtnis = c(22, 67, 22, 31))
daten
```

```
##   aufmerksamkeit gedaechtnis
## 1              58          22
## 2              46          67
## 3              29          22
## 4              51          31
```

6.2 Automatisch

Wir haben bereits die `c()` Funktion gelernt.

Die `c()` Funktion ist die einfachste Art einen Vektor zu erstellen, sie ist aber vermutlich auch die umständlichste. Stellen Sie sich zum Beispiel vor, Sie wollen einen Vektor erstellen, der alle Zahlen von 0 bis 100 enthält. Diese Zahlen wollen Sie definitiv nicht alle in die Klammer von `c()` eintippen.

Glücklicherweise hat R viele eingebaute Funktionen, um leicht automatisch numerische Vektoren zu erstellen.

Lassen Sie uns mit drei davon starten: `a:b`, `seq()`, and `rep()`:

Funktion	Beispiel	Ergebnis
<code>c(a, b, ...)</code>	<code>c(1, 5, 9)</code>	1, 5, 9
<code>a:b</code>	<code>1:5</code>	1, 2, 3, 4, 5
<code>seq(from, to, by, length.out)</code>	<code>seq(from = 0, to = 6, by = 2)</code>	0, 2, 4, 6
<code>rep(x, times, each, length.out)</code>	<code>rep(c(7, 8), times = 2, each = 2)</code>	7, 7, 8, 8, 7, 8, 8

6.3 Zufällig

In R kann man Daten anhand einer Wahrscheinlichkeitsverteilung simulieren.

Wollen wir z.B. eine normalverteilte Variable mit zufälligen Werten erstellen, können wir die `rnorm()` Funktion nutzen.

Dafür müssen wir lediglich angeben, wie viele Werte wir haben wollen (`n`) und welchen Mittelwert (`mean`) und welche Standardabweichung (`sd`) die Verteilung haben soll:

```
rnorm(n = 20, mean = 0, sd = 1)

## [1] -0.124675547  0.923934684  1.036799704  0.159599728  1.123417617
## [6]  0.026977181  0.003212397 -0.334274578 -1.678625666  0.658952124
## [11] -1.233806055  0.174593319  0.230391396  0.004262633 -1.310425214
## [16] -0.056179427 -0.275926178 -0.171228615 -1.495094148 -0.935589983
```

Chapter 7

Daten importieren und speichern

In diesem Kapitel werden wir die Grundlagen der R-Objektverwaltung behandeln. Es wird erläutert, wie Sie neue Objekte, z. B. externe Datensätze, in R laden, wie Sie die bereits vorhandenen Objekte verwalten und wie Sie Objekte aus R in externe Dateien exportieren, die Sie mit anderen Personen teilen oder für Ihre eigene zukünftige Verwendung speichern können.

7.1 Funktionen zur Organisation des Workspace

In diesem Kapitel werden wir einige hilfreiche Funktionen zur Verwaltung Ihres Arbeitsbereichs vorstellen:

Code	Description
<code>ls()</code>	Alle Objekte im aktuellen Arbeitsbereich anzeigen
<code>rm(x, y, ...)</code>	Entfernt die Objekte y, y... aus dem Arbeitsbereich
<code>rm(list = ls())</code>	Entfernt <i>alle</i> Objekte aus dem Arbeitsbereich
<code>getwd()</code> <code>setwd(file = "dir")</code>	Zeigt das aktuelle Arbeitsverzeichnis an Wechselt das Arbeitsverzeichnis zu einem bestimmten Dateipfad
<code>list.files()</code>	Zeigt die Namen aller Dateien im Arbeitsverzeichnis an

Code	Description
<code>write.table(x, file = "mydata.txt", sep)</code>	speichert das Objekt <code>x</code> als Textdatei <code>mydata.txt</code> . Definiere die Trennung der Spalten mittels <code>sep</code> (z.B.; <code>sep = ","</code> für eine kommagetrennte Datei (<code>csv</code>) und <code>sep = "\t"</code> für eine tab-getrennte Datei).
<code>write_rds(x, "meineDaten.rds")</code>	Speichert Objekt <code>x</code> in das R Objekt <code>meineDaten.rds</code>
<code>save.image(file = "meineSession.RData")</code>	Speichert <i>alle</i> Objekte aus dem Arbeitsbereich nach <code>meineSession.RData</code>
<code>read_rds("Daten.rds")</code> Lädt das rds Objekt <code>Daten.rds</code>	Lädt den csv Datensatz <code>Daten.csv</code>
<code> read.csv("Daten.csv")</code>	
<code>foreign::read.spss("Daten.sav")</code> Lädt den SPSS Datensatz <code>Daten.sav</code>	
<code>read.csv("Daten.csv")</code>	Lädt den csv Datensatz <code>Daten.csv</code>
<code>readxl::read_xlsx("Daten.xlsx")</code> Lädt den Excel Datensatz <code>Daten.xlsx</code>	

Ihr Computer ist ein Labyrinth aus Ordnern und Dateien. Wenn Sie außerhalb von R eine bestimmte Datei öffnen möchten, öffnen Sie wahrscheinlich ein Explorer-Fenster, mit dem Sie die Ordner auf Ihrem Computer visuell durchsuchen können. Oder Sie wählen die zuletzt geöffneten Dateien aus oder geben den Namen der Datei in ein Suchfeld ein, um den Computer die Suche für Sie übernehmen zu lassen. Während dieses Vorgehen normalerweise für nicht-programmierende Aufgaben funktioniert, ist es für R ein No-Go. Das Hauptproblem ist, dass Sie bei all diesen Methoden Ihre Ordner visuell durchsuchen und die Maus bewegen müssen, um Ordner und Dateien auszuwählen, die dem Gesuchten entsprechen. Wenn Sie in R programmieren, müssen Sie alle Schritte in Ihren Analysen so spezifizieren, dass sie von anderen und von Ihnen selbst leicht nachvollzogen werden können. Das bedeutet, dass Sie nicht einfach sagen können: "Finde diese eine Datei, die ich mir vor einer Woche gemailt habe" oder "Suche nach einer Datei, die so aussieht wie"MeinFoto.jpg". Stattdessen müssen Sie in der Lage sein, R-Code zu schreiben, der R genau sagt, wo wichtige Dateien zu finden sind - entweder auf Ihrem Computer oder im Internet.

Um diese Aufgabe zu erleichtern, verwendet R Arbeitsverzeichnisse.

7.2 Arbeitsverzeichnis (Working Directory)

Das Arbeitsverzeichnis ist lediglich ein Dateipfad auf Ihrem Computer, der den Standardspeicherort aller Dateien festlegt, die Sie in R einlesen oder aus R heraus speichern. Mit anderen Worten, ein Arbeitsverzeichnis ist wie eine kleine Kiste irgendwo auf Ihrem Computer, die an ein bestimmtes Analyseprojekt

gebunden ist. Wenn Sie R auffordern, einen Datensatz zu importieren wird davon ausgegangen, dass sich die Datei in Ihrem Arbeitsverzeichnis befindet.

Sie können zu jedem Zeitpunkt nur ein Arbeitsverzeichnis aktiv haben. Das aktive Arbeitsverzeichnis wird als Ihr aktuelles Arbeitsverzeichnis bezeichnet.

7.3 Working Environment

Der Arbeitsbereich (auch als Arbeitsumgebung bezeichnet) enthält alle Objekte und Funktionen, die Sie entweder in der aktuellen Sitzung definiert oder aus einer früheren Sitzung geladen haben. Als Sie RStudio zum ersten Mal starteten, war die Arbeitsumgebung leer, da Sie keine neuen Objekte oder Funktionen erstellt hatten. Wenn Sie jedoch neue Objekte und Funktionen mit dem Zuweisungsoperator = definiert haben, wurden diese neuen Objekte in Ihrer Arbeitsumgebung gespeichert. Wenn Sie RStudio nach der Definition neuer Objekte schlossen, erhielten Sie wahrscheinlich eine Meldung mit der Frage “Save workspace image...?”. Damit möchte RStudio Sie fragen, ob Sie alle derzeit in Ihrem Arbeitsbereich definierten Objekte als Bilddatei auf Ihrem Computer speichern möchten.

```
# getwd()
```

7.4 Daten importieren

Wenn Sie Daten in Ihrem Arbeitsverzeichnis haben, können Sie diese nun in R einlesen und dort mit ihnen rechnen. Nehmen wir an, Sie haben in Ihrem Arbeitsverzeichnis einen Ordner mit dem Namen `data`.

Je nachdem in welchem Format die Daten vorliegen, muss ein eigener Befehl genutzt werden. Teilweise braucht man hier auch eigene Pakete (z.B. Excel- oder SPSS-Format).

7.4.1 rds-Format

.rds ist das R-eigene Format. Es speichert alle Objekte die es in R gibt, also potentiell nicht nur Datensätze, sondern auch Testergebnisse, Bilder, o.ä. mittels der Funktion `read_rds()` können wir die Objekte einlesen. Dafür müssen wir das Paket `readr` installiert und mittels `library` eingelesen haben.

```
data = load(file = "data/personality.RData")
# erste Zeilen des Datensatzes ansehen
head(data)
```

```
## [1] "data"
```

7.4.2 csv-Format

Das .csv Format (comma-separated-values) ist eines der gängigsten in der Statistik. Es lässt sich mit allen gebräuchlichen Tabellenprogrammen öffnen (also neben R auch mit Excel, Numbers, o.ä.).

Um einen .csv Datensatz einzulesen, speichern wir ihn mittel `read.csv()` in ein von uns benanntes Objekt. Dieses können wir nennen wie wir wollen, z.B. `data` (schön kurz):

```
data = read.csv("data/personality.csv")
```

erste Zeilen des Datensatzes ansehen

```
head(data)
```

```
##   ID NEO_1_N NEO_2_E NEO_3_O NEO_4_V NEO_5_G NEO_6_N NEO_7_E NEO_8_O NEO_9_V
## 1  1      2      4      0      3      3      0      2      3      0
## 2  2      4      4      0      3      4      0      4      4      0
## 3  3      4      4      2      4      4      0      4      4      0
## 4  4      4      3      4      3      3      0      2      4      0
## 5  5      4      4      1      3      3      0      3      4      4
## 6  6      4      4      0      4      0      1      4      4      0
##   NEO_10_G NEO_11_N NEO_12_E NEO_13_O NEO_14_V NEO_15_G NEO_16_N NEO_17_E
## 1      3      4      1      2      0      4      4      3
## 2      4      1      0      3      0      3      4      4
## 3      4      4      2      4      0      0      4      4
## 4      3      0      0      2      0      0      4      4
## 5      3      1      4      4      0      0      4      4
## 6      2      0      0      4      0      0      4      4
##   NEO_18_O NEO_19_V NEO_20_G NEO_21_N NEO_22_E NEO_23_O NEO_24_V NEO_25_G
## 1      4      3      4      0      4      2      2      3
## 2      3      4      4      0      3      4      0      4
## 3      0      4      4      0      4      4      1      4
## 4      0      4      4      4      4      4      0      3
## 5      0      4      1      1      3      4      0      2
## 6      0      4      4      0      3      4      4      4
##   NEO_26_N NEO_27_E NEO_28_O NEO_29_V NEO_30_G NEO_31_N NEO_32_E NEO_33_O
## 1      4      4      4      2      4      3      0      0
## 2      2      0      1      0      4      4      4      4
## 3      4      0      0      0      1      4      4      3
## 4      0      0      0      0      0      4      4      0
## 5      0      4      4      2      4      4      3      4
## 6      0      0      4      2      0      4      3      2
##   NEO_34_V NEO_35_G NEO_36_N NEO_37_E NEO_38_O NEO_39_V NEO_40_G NEO_41_N
```

```

## 1      3      4      2      4      1      0      4      4
## 2      4      4      0      4      3      1      4      0
## 3      4      4      3      4      2      0      4      0
## 4      3      4      0      3      3      1      4      0
## 5      3      3      0      4      0      2      3      0
## 6      4      4      0      2      4      2      4      0
##   NEO_42_E NEO_43_O NEO_44_V NEO_45_G NEO_46_N NEO_47_E NEO_48_O NEO_49_V
## 1      2      1      4      0      4      4      3      3
## 2      2      4      0      2      4      0      4      4
## 3      4      4      4      0      4      0      4      4
## 4      2      2      0      0      4      4      3      3
## 5      0      2      0      1      4      0      2      3
## 6      0      4      0      0      4      0      2      4
##   NEO_50_G NEO_51_N NEO_52_E NEO_53_O NEO_54_V NEO_55_G NEO_56_N NEO_57_E
## 1      3      0      4      2      0      0      2      2
## 2      4      0      4      4      0      0      0      0
## 3      4      3      4      4      4      0      0      1
## 4      4      4      4      1      0      0      4      0
## 5      3      0      3      3      0      4      0      1
## 6      4      0      4      4      1      0      0      0
##   NEO_58_O NEO_59_V NEO_60_G
## 1      3      1      4
## 2      3      0      4
## 3      4      0      4
## 4      4      1      3
## 5      3      4      4
## 6      4      0      3

```

7.4.3 sav-Format

Das .sav Format ist das Format in welchem SPSS Datensätze abgespeichert werden. Dies kommt gerade in den Sozialwissenschaften relativ häufig vor, weshalb wir diese Art von Dateien auf jeden Fall einlesen können sollten.

Um einen .sav Datensatz einzulesen, speichern wir ihn mittel `read.spss()` in ein von uns benanntes Objekt. Dafür müssen wir das Paket `foreign` installiert und mittels `library` eingelesen haben. Damit R den Datensatz automatisch in einen `data.frame` speichert, geben wir als Zusatzoption `to.data.frame = TRUE` an:

```

library(foreign)
data = read.spss("data/personality.sav", to.data.frame = TRUE)

# erste Zeilen des Datensatzes ansehen
head(data)

```

```

##   ID NEO_1_N NEO_2_E NEO_3_O NEO_4_V NEO_5_G NEO_6_N NEO_7_E NEO_8_O NEO_9_V
## 1  1    2    4    0    3    3    0    2    3    0
## 2  2    4    4    0    3    4    0    4    4    0
## 3  3    4    4    2    4    4    0    4    4    0
## 4  4    4    3    4    3    3    0    2    4    0
## 5  5    4    4    1    3    3    0    3    4    4
## 6  6    4    4    0    4    0    1    4    4    0
##   NEO_10_G NEO_11_N NEO_12_E NEO_13_O NEO_14_V NEO_15_G NEO_16_N NEO_17_E
## 1  3    4    1    2    0    4    4    3
## 2  4    1    0    3    0    3    4    4
## 3  4    4    2    4    0    0    4    4
## 4  3    0    0    2    0    0    4    4
## 5  3    1    4    4    0    0    4    4
## 6  2    0    0    4    0    0    4    4
##   NEO_18_O NEO_19_V NEO_20_G NEO_21_N NEO_22_E NEO_23_O NEO_24_V NEO_25_G
## 1  4    3    4    0    4    2    2    3
## 2  3    4    4    0    3    4    0    4
## 3  0    4    4    0    4    4    1    4
## 4  0    4    4    4    4    4    0    3
## 5  0    4    1    1    3    4    0    2
## 6  0    4    4    0    3    4    4    4
##   NEO_26_N NEO_27_E NEO_28_O NEO_29_V NEO_30_G NEO_31_N NEO_32_E NEO_33_O
## 1  4    4    4    2    4    3    0    0
## 2  2    0    1    0    4    4    4    4
## 3  4    0    0    0    1    4    4    3
## 4  0    0    0    0    0    4    4    0
## 5  0    4    4    2    4    4    3    4
## 6  0    0    4    2    0    4    3    2
##   NEO_34_V NEO_35_G NEO_36_N NEO_37_E NEO_38_O NEO_39_V NEO_40_G NEO_41_N
## 1  3    4    2    4    1    0    4    4
## 2  4    4    0    4    3    1    4    0
## 3  4    4    3    4    2    0    4    0
## 4  3    4    0    3    3    1    4    0
## 5  3    3    0    4    0    2    3    0
## 6  4    4    0    2    4    2    4    0
##   NEO_42_E NEO_43_O NEO_44_V NEO_45_G NEO_46_N NEO_47_E NEO_48_O NEO_49_V
## 1  2    1    4    0    4    4    3    3
## 2  2    4    0    2    4    0    4    4
## 3  4    4    4    0    4    0    4    4
## 4  2    2    0    0    4    4    3    3
## 5  0    2    0    1    4    0    2    3
## 6  0    4    0    0    4    0    2    4
##   NEO_50_G NEO_51_N NEO_52_E NEO_53_O NEO_54_V NEO_55_G NEO_56_N NEO_57_E
## 1  3    0    4    2    0    0    2    2
## 2  4    0    4    4    0    0    0    0
## 3  4    3    4    4    4    0    0    1

```

```

## 4      4      4      4      1      0      0      4      0
## 5      3      0      3      3      0      4      0      1
## 6      4      0      4      4      1      0      0      0
##   NEO_58_O NEO_59_V NEO_60_G
## 1      3      1      4
## 2      3      0      4
## 3      4      0      4
## 4      4      1      3
## 5      3      4      4
## 6      4      0      3

```

7.4.4 xlsx-Format

Das .xlsx Format ist das Format in welchem Excel Datensätze abgespeichert werden. Auch das kommt oft vor, da es viele Forscher:innen vorziehen Daten in Excel Tabellen einzutragen.

Um einen .xlsx Datensatz einzulesen, speichern wir ihn mittels `read_xlsx()` in ein von uns benanntes Objekt. Dafür müssen wir das Paket `readxl` installiert und mittels `library` eingelesen haben. Da in Excel Tabellen manchmal mehrere Arbeitsblätter (engl. “sheets”) vorliegen, geben wir den Namen des Arbeitsblatt, welches wir brauchen, zusätzlich an:

```

library(readxl)
data = read_xlsx("data/personality.xlsx", sheet = "Tabelle1")

# erste Zeilen des Datensatzes ansehen
head(data)

## # A tibble: 6 x 61
##       ID NEO_1_N NEO_2_E NEO_3_O NEO_4_V NEO_5_G NEO_6_N NEO_7_E NEO_8_O NEO_9_V
##     <dbl>    <dbl>    <dbl>    <dbl>    <dbl>    <dbl>    <dbl> <chr>    <dbl>    <chr>
## 1     1      2      4      0      3      3      0 2      3 0
## 2     2      4      4      0      3      4      0 4      4 0
## 3     3      4      4      2      4      4      0 4      4 0
## 4     4      4      3      4      3      3      0 2      4 0
## 5     5      4      4      1      3      3      0 3      4 4
## 6     6      4      4      0      4      0      1 4      4 0
## # i 51 more variables: NEO_10_G <dbl>, NEO_11_N <dbl>, NEO_12_E <dbl>,
## #   NEO_13_O <dbl>, NEO_14_V <dbl>, NEO_15_G <dbl>, NEO_16_N <dbl>,
## #   NEO_17_E <dbl>, NEO_18_O <dbl>, NEO_19_V <dbl>, NEO_20_G <dbl>,
## #   NEO_21_N <dbl>, NEO_22_E <dbl>, NEO_23_O <dbl>, NEO_24_V <dbl>,
## #   NEO_25_G <dbl>, NEO_26_N <dbl>, NEO_27_E <dbl>, NEO_28_O <dbl>,
## #   NEO_29_V <dbl>, NEO_30_G <dbl>, NEO_31_N <dbl>, NEO_32_E <dbl>,
## #   NEO_33_O <dbl>, NEO_34_V <dbl>, NEO_35_G <dbl>, NEO_36_N <dbl>, ...

```

7.5 Daten speichern

In alle vorgestellten Formate können wir unsere Daten natürlich auch abspeichern. Erstellen wir dafür einen schnellen Test-Datensatz mittels `data.frame`:

```
newdata = data.frame(Variable1 = c(1, 2, 3, 4),
                     Variable2 = c("Person1", "Person2", "Person3", "Person4"))
newdata

##   Variable1 Variable2
## 1         1 Person1
## 2         2 Person2
## 3         3 Person3
## 4         4 Person4
```

7.5.1 rds-Format

Um in das R eigene .rds Format zu speichern, nutzen wir wieder das `readr` Paket, welches wir zunächst mittels `library` laden. Der Befehl ist nun `write_rds()`. Innerhalb des Befehls geben wir einfach das Objekt (z.B. unseren Datensatz) an, welches wir speichern wollen. Unter `file` geben wir der Datei einen Namen, so wie wir sie auf unserer Festplatte gepeichert haben wollen. Wichtig: Das Kürzel des Dateityps (.rds) nicht vergessen:

```
library(readr)

write_rds(x = newdata, file = "data.rds")
```

Die Datei erscheint nun in Ihrem Arbeitsverzeichnis.

7.5.2 csv-Format

Speichern in das .csv Format funktioniert analog:

```
write.csv(x = newdata, file = "newdata.csv")
```

Hier ein Tipp für Sie: Wie oben bereits erwähnt ist csv ein “Komma-getrenntes” Format. In Deutschland haben wir die Besonderheit, dass wir Dezimalstellen manchmal mit „,“ trennen, während man im englisch-sprachigen Raum i.d.R. „.” verwendet. Sollte die abgespeicherte Datei komisch aussehen, wenn Sie sie z.B. in Excel öffnen, liegt das vermutlich an der deutschen Einstellung Ihres Programms. Probieren Sie in diesem Fall statt der `write.csv()` Funktion einmal die `write.csv2()` Funktion aus, dies wird Ihr Problem lösen.

```
write.csv2(x = newdata, file = "newdata.csv")
```

7.5.3 sav-Format

Zum Speichern in das SPSS Format .sav nutzen wir die Funktion `write_sav()` aus dem Paket `haven`, welches wir zuvor mittel `library()` laden.

```
library(haven)

write_sav(newdata, "newdata.sav")
```

7.5.4 xlsx-Format

Zum Speichern in das Excel Format .xlsx nutzen wir die Funktion `WriteXLS` aus dem Paket `WriteXLS`, welches wir zuvor mittel `library()` laden.

```
library(WriteXLS)

WriteXLS(newdata, "newdata.xlsx")
```


Chapter 8

Daten auswählen

Oft wollen wir nur mit einem Teil der verfügbaren Daten rechnen. In diesem Fall müssen wir uns die relevanten Daten aus dem größeren Datenobjekt “herausziehen”.

Eine Möglichkeit, die **Indizierung** mit Zahlen haben wir bereits kennengelernt. Diese funktioniert mit eckigen Klammern []. Für die Auswahl eines bestimmten Objekts schreiben wir einfach dessen numerische Position mittels einer Zahl in die Klammern:

```
a = c(1,5,6,8)
```

```
# Für das 3. Objekt  
a[3]
```

```
## [1] 6
```

Dies funktioniert auch, wenn wir mehrere Objekte auswählen wollen:

```
# Für das 1. und 3. Objekt  
a[c(1, 3)]
```

```
## [1] 1 6
```

Bei data.frames, die sowohl Zeilen als auch Spalten haben, trennen wir die eckigen Klammern mit einem Komma [,]. Alles vor dem Komma bezieht sich auf die Zeilen (Fälle), alles nach dem Komma auf die Spalten (Variablen):

```
testdata = data.frame(IQ = c(101, 112, 97, 104),  
                      Variable2 = c("Person1", "Person2", "Person3", "Person4"))  
  
# Für die Zelle in der 3. Zeile der 2. Spalte  
testdata[3,2]
```

```
## [1] Person3
## Levels: Person1 Person2 Person3 Person4
```

Wir können uns auch ganze Zeilen und Spalten anzeigen lassen, wenn wir die Position vor, bzw. nach dem Komma leer lassen

```
# Für die ganze 3. Zeile
testdata[3,]
```

```
##   IQ Variable2
## 3 97  Person3
# Für die ganze 2. Spalte
testdata[,2]
```

```
## [1] Person1 Person2 Person3 Person4
## Levels: Person1 Person2 Person3 Person4
```

Die Auswahl mittels eines numerischen Index ist unkompliziert, aber es kann manchmal aufwendig sein, aus großen Datensätzen eine Vielzahl von Fällen bzw. Variablen auszuwählen. Zudem **verschieben** sich Indizes auch, wenn wir einzelne Fälle oder Variablen aus dem Datensatz löschen oder welche hinzufügen.

Wir brauchen also auch Strategien, nach welchen wir Daten mit einer **Logik** auswählen können.

Um dies auszuprobieren, werden wir als Beispiel den Datensatz “starwars” verwenden. Dieser ist in dem Paket **dplyr** gespeichert, welches wir vorher installieren und mittels **library()** laden.

Der Datensatz “starwars” umfasst alle in den Star-Wars Filmen vorkommenden Charaktere und beschreibt diese auf einer Vielzahl von Variablen. Wir wollen nur die ersten 11 Variablen nutzen (die anderen enthalten zu lange Einträge).

```
library(dplyr)

starwars = as.data.frame(starwars[,1:11])
head(starwars)

##           name height mass hair_color skin_color eye_color birth_year
## 1 Luke Skywalker    172   77     blond      fair     blue      19.0
## 2          C-3PO     167   75       <NA>      gold    yellow     112.0
## 3         R2-D2      96   32       <NA>  white, blue      red      33.0
## 4      Darth Vader    202  136       none      white    yellow     41.9
## 5      Leia Organa    150   49       brown     light    brown      19.0
## 6        Owen Lars    178  120  brown, grey     light     blue      52.0
##   sex gender homeworld species
## 1 male masculine  Tatooine   Human
## 2  none masculine  Tatooine   Droid
## 3  none masculine     Naboo   Droid
## 4 male masculine  Tatooine   Human
```

```
## 5 female feminine Alderaan Human
## 6 male masculine Tatooine Human
```

8.1 Vektor

Als Beispielvektor nutzen wir das Körpergewicht der Charaktere, im Datensatz in der Variable `mass` gespeichert. Am besten wählen wir ihn einmal direkt an:

```
gewicht = starwars$mass
gewicht
```

```
## [1] 77.0 75.0 32.0 136.0 49.0 120.0 75.0 32.0 84.0 77.0
## [11] 84.0 NA 112.0 80.0 74.0 1358.0 77.0 110.0 17.0 75.0
## [21] 78.2 140.0 113.0 79.0 79.0 83.0 NA NA 20.0 68.0
## [31] 89.0 90.0 NA 66.0 82.0 NA NA NA 40.0 NA
## [41] NA 80.0 NA 55.0 45.0 NA 65.0 84.0 82.0 87.0
## [51] NA 50.0 NA NA 80.0 NA 85.0 NA NA 80.0
## [61] 56.2 50.0 NA 80.0 NA 79.0 55.0 102.0 88.0 NA
## [71] NA 15.0 NA 48.0 NA 57.0 159.0 136.0 79.0 48.0
## [81] 80.0 NA NA NA NA NA 45.0
```

Anstatt mit Indizes zu arbeiten können wir Werte nach einer Logik auswählen. Dafür eignen sich sogenannte Bool'sche Operatoren `=`, `>`, `<`

Um zum z.B. die Gewichte aller Charaktere auszuwählen, die **genau** 79kg wiegen schreiben wir:

```
gewicht[gewicht == 79]
```

```
## [1] NA 79 79 NA 79 NA NA NA
## [26] NA 79 NA NA NA NA
```

Wie wir sehen, wählt R genau die Charaktere, die 79kg wiegen. Zusätzlich behält R jedoch auch alle Positionen, die einen fehlenden Wert aufweisen, da hier die Aussage `gewicht == 79` faktisch nicht **falsch** ist.

Wollen wir die fehlenden Werte entfernen, können wir die sehr nützliche Funktion `which()` nutzen. Diese befiehlt R alle Werte auszuwählen, auf die das Statement **explizit zutrifft**:

```
gewicht[which(gewicht == 79)]
```

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

Dasselbe funktioniert auch bei kategorischen Variablen:

```
haarfarbe = starwars$hair_color
haarfarbe
```

```

## [1] "blond"      NA          NA          "none"
## [5] "brown"       "brown, grey"  "brown"     NA
## [9] "black"       "auburn, white" "blond"    "auburn, grey"
## [13] "brown"       "brown"       NA          NA
## [17] "brown"       "brown"       "white"    "grey"
## [21] "black"       "none"        "none"     "black"
## [25] "none"        "none"        "auburn"   "brown"
## [29] "brown"       "none"        "brown"    "none"
## [33] "blond"       "none"        "none"     "none"
## [37] "brown"       "black"       "none"     "black"
## [41] "black"       "none"        "none"     "none"
## [45] "none"        "none"        "none"     "none"
## [49] "white"       "none"        "black"    "none"
## [53] "none"        "none"        "none"     "none"
## [57] "black"       "brown"       "brown"    "none"
## [61] "black"       "black"       "brown"    "white"
## [65] "black"       "black"       "blonde"   "none"
## [69] "none"        "none"        "white"    "none"
## [73] "none"        "none"        "none"     "none"
## [77] "none"        "brown"       "brown"    "none"
## [81] "none"        "black"       "brown"    "brown"
## [85] "none"        "unknown"    "brown"    
```

Zur Auswahl nur braunhaariger Charaktere schreiben wir z.B.:

```
haarfarbe[which(haarfarbe == "brown")]
```

```

## [1] "brown" "brown" "brown" "brown" "brown" "brown" "brown" "brown" "brown"
## [10] "brown" "brown" "brown" "brown" "brown" "brown" "brown" "brown" "brown"
```

Interessieren uns alle Werte, die genau nicht 79 kg (also ungleich 79 sind) sind nutzen wir `!`, was in R immer so viel wie *nicht* bedeutet:

```
gewicht[which(gewicht != 79)]
```

```

## [1]  77.0  75.0  32.0 136.0  49.0 120.0  75.0  32.0  84.0  77.0
## [11] 84.0 112.0  80.0  74.0 1358.0  77.0 110.0  17.0  75.0  78.2
## [21] 140.0 113.0  83.0  20.0  68.0  89.0  90.0  66.0  82.0  40.0
## [31]  80.0  55.0  45.0  65.0  84.0  82.0  87.0  50.0  80.0  85.0
## [41]  80.0  56.2  50.0  80.0  55.0 102.0  88.0  15.0  48.0  57.0
## [51] 159.0 136.0  48.0  80.0  45.0 
```

Für größer-kleiner Statements nutzen wir (nur bei numerischen Variablen):

```
gewicht[which(gewicht < 79)]
```

```

## [1] 77.0 75.0 32.0 49.0 75.0 32.0 77.0 74.0 77.0 17.0 75.0 78.2 20.0 68.0 66.0
## [16] 40.0 55.0 45.0 65.0 50.0 56.2 50.0 55.0 15.0 48.0 57.0 48.0 45.0 
```

```

gewicht[which(gewicht > 79)]

## [1] 136 120 84 84 112 80 1358 110 140 113 83 89 90 82 80
## [16] 84 82 87 80 85 80 80 102 88 159 136 80

# bzw.

gewicht[which(gewicht <= 79)]

## [1] 77.0 75.0 32.0 49.0 75.0 32.0 77.0 74.0 77.0 17.0 75.0 78.2 79.0 79.0 20.0
## [16] 68.0 66.0 40.0 55.0 45.0 65.0 50.0 56.2 50.0 79.0 55.0 15.0 48.0 57.0 79.0
## [31] 48.0 45.0

gewicht[which(gewicht >= 79)]

## [1] 136 120 84 84 112 80 1358 110 140 113 79 79 83 89 90
## [16] 82 80 84 82 87 80 85 80 80 79 102 88 159 136 79
## [31] 80

```

Oft wollen wir unsere Auswahl nicht nur nach einem Kriterium treffen, sondern mehrere Kriterien verbinden. Dabei helfen uns die Verknüpfungsoperatoren `&` was soviel heißt wie `und` sowie `|` was soviel heißt wie `oder`.

Wollen wir beispielsweise nur das Gewicht von Charakteren auswählen, die mehr wiegen als 50 kg und weniger wiegen als 100kg schreiben wir:

```

gewicht[which(gewicht > 50 & gewicht < 100)]

## [1] 77.0 75.0 75.0 84.0 77.0 84.0 80.0 74.0 77.0 75.0 78.2 79.0 79.0 83.0 68.0
## [16] 89.0 90.0 66.0 82.0 80.0 55.0 65.0 84.0 82.0 87.0 80.0 85.0 80.0 56.2 80.0
## [31] 79.0 55.0 88.0 57.0 79.0 80.0

```

Dürfen die Charaktere entweder leichter als 50 kg oder schwerer als 100 kg sein schreiben wir:

```

gewicht[which(gewicht < 50 | gewicht > 100)]

## [1] 32 136 49 120 32 112 1358 110 17 140 113 20 40 45 102
## [16] 15 48 159 136 48 45

```

8.2 Dataframe

Um die Ausgaben etwas übersichtlicher zu gestalten (der starwars dataframe hat 89 Zeilen), beschränken wir den Datensatz für den nächsten Abschnitt auf die ersten 5 Zeilen:

```
starwars = starwars[1:5,]
```

8.2.1 Zeilen (Fälle) auswählen

Die Logik in der Auswahl der Fälle funktioniert analog zur Auswahl bei den Vektoren. Diesmal müssen Sie jedoch, wie zuvor erwähnt, Ihre Selektion links vom Komma in die [] schreiben. Zudem müssen Sie beim Schreiben der Auswahlkriterien darauf achten, die Variablen mit dem \$ anzuhören.

Zur Auswahl aller Fälle, die braune Haare haben, schreiben wir z.B.

```
# starwars[starwars$hair_color == "brown",]
```

8.2.2 Spalten (Variablen) auswählen

Die Spalten eines Datensatzes wählt man am effizientesten über den **Variablennamen** aus.

Noch einmal zum Überblick die Variablen im **starwars** Datensatz

```
names(starwars)
```

```
## [1] "name"      "height"     "mass"       "hair_color" "skin_color"
## [6] "eye_color"  "birth_year"  "sex"        "gender"     "homeworld"
## [11] "species"
```

Bei einzelnen Variablen lässt sich dies einfach durch Nennung des Variablenamens in "" bewerkstelligen:

```
starwars[, "height"]
```

```
## [1] 172 167  96 202 150
```

Möchte man jedoch nach einer bestimmten Logik aus den **Variablennamen** auswählen (letztlich ein Vektor aus Wörtern), ist die Funktion **select()** aus dem Paket **dplyr** herausragend gut geeignet.

Folgende Hilfsfunktionen für den Befehl **select()** können wir nutzen:

##	Befehl	Funktion
## 1	<code>starts_with()</code>	Variable beginnt mit dem Präfix
## 2	<code>ends_with()</code>	Variable endet mit dem Suffix
## 3	<code>contains()</code>	Variable enthält genau diese Zeichenkette
## 4	<code>num_range()</code>	Entspricht einer Zahlenfolge

Wollen wir z.B. alle Variablen auswählen, deren Namen mit "hair" beginnen, schreiben wir:

```
library(dplyr)
```

```
select(starwars, starts_with("hair"))
```

```
##   hair_color
## 1     blond
## 2     <NA>
## 3     <NA>
## 4     none
## 5     brown
```

Wollen wir z.B. alle Variablen auswählen, deren Namen mit “color” enden, schreiben wir:

```
library(dplyr)

select(starwars, ends_with("color"))

##   hair_color skin_color eye_color
## 1     blond      fair     blue
## 2     <NA>       gold    yellow
## 3     <NA>   white, blue      red
## 4     none       white    yellow
## 5     brown      light     brown
```

Wollen wir z.B. alle Variablen auswählen, deren Namen die Zeichenkette “me” beinhalten, schreiben wir:

```
library(dplyr)

select(starwars, contains("me"))

##           name homeworld
## 1 Luke Skywalker Tatooine
## 2        C-3PO Tatooine
## 3       R2-D2   Naboo
## 4   Darth Vader Tatooine
## 5     Leia Organa Alderaan
```

Wir können die Bedingungen auch mit den & und | Operatoren verknüpfen:

```
library(dplyr)

select(starwars, starts_with("hair") & ends_with("color"))

##   hair_color
## 1     blond
## 2     <NA>
## 3     <NA>
## 4     none
## 5     brown
```


Chapter 9

Daten beschreiben

Im folgenden Abschnitt werden wir lernen, wie wir uns schnell einen Überblick über große Mengen von Daten machen können.

Hierzu gehört natürlich, dass wir einzelne Variablen mit **Deskriptivstatistiken** (z.B. Mittelwert, Median, Standardabweichung...) zusammenfassen, so wie wir es im 1. Semester gelernt haben.

Wir werden jedoch auch lernen, wie wir uns effizient Informationen über Datensätze ausgeben lassen können, z.B. über ihre Größe (z.B. Anzahl Zeilen und Spalten) Art der enthaltenen Variablen und fehlende Werte.

9.1 Informationen über R Objekte

9.1.1 str()

Um schnell Informationen über ein R Objekt (Vektor, data.frame, Bild o.ä.) zu erhalten, nutzen wir den **str()** Befehl.

So können wir einen schnellen Überblick erhalten, um was für ein Objekt es sich handelt und “was in dem Objekt drin steckt”.

Als Beispiel benutzen wir den **iris** Datensatz, der in der Basisversion von R enthalten ist. Dieser enthält Informationen über 3 Spezies der Blumenart Iris.

```
head(iris) # erste 6 Zeilen ansehen
```

```
##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
## 1         5.1       3.5        1.4       0.2   setosa
## 2         4.9       3.0        1.4       0.2   setosa
## 3         4.7       3.2        1.3       0.2   setosa
```

```

## 4      4.6      3.1      1.5      0.2  setosa
## 5      5.0      3.6      1.4      0.2  setosa
## 6      5.4      3.9      1.7      0.4  setosa
str(iris)

## 'data.frame':   150 obs. of  5 variables:
## $ Sepal.Length: num  5.1 4.9 4.7 4.6 5 5.4 4.6 5 4.4 4.9 ...
## $ Sepal.Width : num  3.5 3 3.2 3.1 3.6 3.9 3.4 3.4 2.9 3.1 ...
## $ Petal.Length: num  1.4 1.4 1.3 1.5 1.4 1.7 1.4 1.5 1.4 1.5 ...
## $ Petal.Width : num  0.2 0.2 0.2 0.2 0.2 0.4 0.3 0.2 0.2 0.1 ...
## $ Species     : Factor w/ 3 levels "setosa","versicolor",...: 1 1 1 1 1 1 1 1 1 1 ...

```

Der Befehl zeigt uns eine Übersicht, welche Variablen in dem `iris` Datensatz enthalten sind. zusätzlich sehen wir die Dimensionen (150 Zeilen, 5 Spalten) des dataframes. Für die einzelnen Variablen wird uns jeweils die Variablenart (z.B. `Petal.Length`, engl Blütenblattlänge = numerisch; `Species` = factor) sowie einzelne in dem Variablen enthaltene Werte.

9.1.2 Informationen über Vektoren

9.1.2.1 `length()`

Um herauszufinden, wie viele Stellen ein Vektor hat, nutzen wir die Funktion `length()`:

```

vektor = c("A", "B", "C", "D")
length(vektor)

## [1] 4

```

9.1.2.2 `is...` Funktionen

Wir können zudem die Variablenart des Vektors abfragen. Dies funktioniert mit dem `is.()` Befehl, ergänzt durch die Variablenart, nach der wir fragen wollen. Z.B. enthält die Variable `Petal.Length` des `iris` Datensatzes ausschließlich Zahlen:

```

is.character(iris$Petal.Length)

## [1] FALSE
is.numeric(iris$Petal.Length)

## [1] TRUE

```

9.1.2.3 `is.na()`

Wir können auch spezifisch nach fehlenden Werten Fragen. Erstellen wir uns schnell einen Beispielvektor:

```
a = c(1, 2, 3, NA, 5, NA)

is.na(a)
```

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

Auf diese Art, können wir auch leicht zählen, wie viele Elemente eines Vektors als “fehlende Werte” berücksichtigt werden müssen:

```
sum(is.na(a))

## [1] 2
```

9.1.3 Informationen über dataframes

```
#length()
#str()
#nrow()
#ncol()
#is.na()
```

9.2 Deskriptivstatistiken

9.2.1 Kategorische Variablen

9.2.1.1 Deskription kategorische Vektoren

Funktion	Beispiel	Ergebnis	Result
<code>unique(x)</code>	Zeigt Vektor aller einzigartiger Werte an	<code>unique(c(1, 1, 2, 10))</code>	1, 2, 10

Funktion	Beispiel	Ergebnis	Result
<code>table(x)</code>	Gibt Tabelle aller einzigartiger Werte und deren absolute Häufigkeit an. Um auch fehlende Werte zu zählen wählen Sie <code>exclude = NULL</code>	<code>table(c("a", "a", "b", "c"))</code>	<code>2-"a", 1-"b", 1-"c"</code>

9.2.1.2 Absolute Häufigkeiten

Leider gibt es zum Zusammenfassen kategorialer (nominalskalierten) Variablen nicht so viele Möglichkeiten wie für die numerischen Variablen (die Mittelwert, Median, Standardabweichung... ermöglichen).

Die gängigste Deskription einer kategorialen Variable ist die Erstellung einer Häufigkeitstabelle. Eine Häufigkeitstabelle mit absoluten Häufigkeiten lässt sich mit dem `table()` Befehl erstellen.

```
df = data.frame(Geschlecht = c("männlich", "männlich", "männlich", "weiblich", "weiblich",
                               Diagnose = c("Depression", "Depression", "Angst", "Depression", "Angst"))

table(df$Diagnose)

##          Angst Depression
##                2            3
```

Wollen wir die Häufigkeit einer bestimmten Kategorie (z.B. nur der Diagnose Depression) auswählen, dann geht dies so:

```
table(df$Diagnose)[["Depression"]]

## Depression
##           3
```

Wollen wir die Häufigkeit einer Kategorie innerhalb von unterschiedlichen Gruppen darstellen, geht dies ebenfalls mit `table()`.

Wir schreiben lediglich die beide relevanten Variablen in die Klammer:

```
table(df$Diagnose, df$Geschlecht)

##                         männlich weiblich
## Angst                  1            1
```

```
## Depression      2      1
```

Das Ergebnis ist eine sogenannte **4-Felder Tafel**.

9.2.1.3 Relative Häufigkeiten

9.2.1.4 unique() Funktion

Wollen wir in einem Vektor, der viele Wiederholungen der selben Merkmalsausprägungen enthält, jede Ausprägung nur einmal anzeigen lassen, nutzen wir `unique()`:

```
unique(df$Diagnose)
```

```
## [1] Depression Angst
## Levels: Angst Depression
```

9.2.2 Numerische Variablen

Funktion	Beispiel	Ergebnis
<code>sum(x)</code> , <code>product(x)</code>	<code>sum(1:10)</code>	55
<code>min(x)</code> , <code>max(x)</code>	<code>min(1:10)</code>	1
<code>mean(x)</code> , <code>median(x)</code>	<code>mean(1:10)</code>	5.5
<code>sd(x)</code> , <code>var(x)</code> ,	<code>sd(1:10)</code>	3.0276504
<code>range(x)</code>		
<code>quantile(x, probs)</code>	<code>quantile(1:10, probs = .2)</code>	2.8
<code>summary(x)</code>	<code>summary(1:10)</code>	Min = 1.00. 1st Qu. = 3.25, Median = 5.50, Mean = 5.50, 3rd Qu. = 7.75, Max = 10.0

9.2.2.1 Lagemaße

Lagemaße, auch bekannt als Maße der zentralen Tendenz, sind statistische Kennzahlen, die uns Informationen über die typische oder durchschnittliche Lage einer Verteilung geben. Sie helfen dabei, den Mittelpunkt oder die Mitte eines Datensatzes zu bestimmen.

Die gängigsten Lagemaße sind:

- Mittelwert (arithmetisches Mittel, engl. *mean*)
- Median
- Modus (Modalwert).

Lagemaße sind nützliche Werkzeuge, um einen Überblick über Daten zu gewinnen und Vergleiche zwischen verschiedenen Verteilungen anzustellen (z.B. bei Gruppenvergleichen).

Schauen wir uns die Funktionen einmal in Aktion an:

```
vektor = c(2, 4, 2, 1, 5, 2, 5, 6)

mean(vektor)

## [1] 3.375

median(vektor)

## [1] 3
```

Diese Funktionen funktionieren nur für Vektoren, nicht für `data.frames()`.

Vorsicht: Es kann vorkommen, dass wir in unseren Daten fehlende Werte (missings) haben. Dann funktionieren die Funktionen nicht mehr:

```
vektor = c(2, 4, NA, 1, 5, 2, 5, NA)

mean(vektor)

## [1] NA
```

Wir können das Problem jedoch mit einem kleinen Extra-Argument (`na.rm = T`) lösen. Die Statistik wird dann auf Basis aller verfügbaren Werte berechnet und die fehlenden Werte werden ignoriert:

```
mean(vektor, na.rm = T)

## [1] 3.166667
```

9.2.2.2 Streuungsmaße

Streuungsmaße sind statistische Kennzahlen, die uns Auskunft über die Verteilung oder Streuung von Daten um die zentrale Tendenz geben. Sie helfen dabei, die Variabilität oder die Ausdehnung eines Datensatzes zu quantifizieren.

Die gängigsten Streuungsmaße sind:

- Spannweite (engl. *range*)
- Varianz
- Standardabweichung (engl. *standard deviation*, SD)
- Quartilabstand (engl. *inter-quartile-range*, IQR)

Streuungsmaße helfen dabei, die Homogenität oder Heterogenität von Daten zu analysieren und den Grad der Variation zu verstehen.

Nun probieren wir die Funktionen zur Berechnung der Streuungsmaße aus:

```
vektor = c(2, 4, 2, 1, 5, 2, 5, 6)

var(vektor)

## [1] 3.410714

sd(vektor)

## [1] 1.846812
```

Wie wir wissen, ist die Standardabweichung die Wurzel aus der Varianz:

```
sqrt(var(vektor))
```

```
## [1] 1.846812
```

Zur Berechnung des Quartilsabstands (IQR) benutzen wir die `IQR()` Funktion aus dem `stats` Paket:

```
# install.packages("stats")

library(stats)

IQR(vektor)
```

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

9.3 Deskriptivstatistiken berichten

Berichtet man Deskriptivstatistiken im Ergebnisteil eines Papers oder in Tabellen, gibt man diese Häufig in Kombination an.

Ein Streuungsmaß ist z.B. wenig informativ, ohne zu wissen um welches Lagemaß es herum streut.

Ein Lagemaß alleine gibt der Leserschaft zwar eine Information über die Mitte der Verteilung, jedoch hat man dann keine Vorstellung über die Variabilität der Werte.

Die gängigsten Kombinationen sind wie folgt:

	Statistik 1	Statistik 2	Bericht
Absolute Häufigkeit	Relative Häufigkeit	n (%)	23 (34%)
Mittelwert	Standardabweichung	SD	42.11 (8.51)
Median	Quartilsabstand	med (IQR)	11 (3)

Man würde zum Beispiel schreiben:

Für unsere Studie wurden $N = 200$ Personen rekrutiert. Diese waren m (SD) = 31.23 (11.31) Jahre alt. Die Geschlechterverteilung umfasste 120 (60%) weibliche und 80 (40%) männliche Personen.

9.4 Alles auf einen Blick

Manchmal uns nicht die Mühe machen, die Deskriptivstatistiken alle einzeln und nacheinander zu berechnen. Schöner wäre es, mit einer Funktion alle relevanten Kennwerte auf einmal zu erhalten.

Eine solche Funktion ist `describe()` aus dem `psych` Paket:

```
# install.packages("psych")
library(psych)

## 
## Attaching package: 'psych'
## The following objects are masked from 'package:ggplot2':
## 
##     %+%, alpha
vektor = c(2, 4, 2, 1, 5, 2, 5, 6)
describe(vektor)

##      vars   n  mean    sd median trimmed   mad min max range skew kurtosis    se
## X1     1 8 3.38 1.85      3 3.38 2.22  1  6 5 0.11 -1.88 0.65
```

Diese Funktion funktioniert sogar, wenn wir sie auf einen `data.frame` anwenden. Nehmen wir hierfür noch einmal den `iris` Datensatz vom Anfang dieses Kapitels:

```
describe(iris)

##           vars   n  mean    sd median trimmed   mad min max range skew kurtosis    se
## Sepal.Length  1 150 5.84 0.83  5.80  5.81 1.04 4.3 7.9  3.6  0.31
## Sepal.Width   2 150 3.06 0.44  3.00  3.04 0.44 2.0 4.4  2.4  0.31
## Petal.Length  3 150 3.76 1.77  4.35  3.76 1.85 1.0 6.9  5.9 -0.27
## Petal.Width   4 150 1.20 0.76  1.30  1.18 1.04 0.1 2.5  2.4 -0.10
## Species*      5 150 2.00 0.82  2.00  2.00 1.48 1.0 3.0  2.0  0.00
##                 kurtosis    se
## Sepal.Length   -0.61 0.07
## Sepal.Width    0.14 0.04
## Petal.Length  -1.42 0.14
```

```
## Petal.Width      -1.36 0.06
## Species*        -1.52 0.07
```

9.5 Deskriptivstatistiken für Subgruppen

Häufig müssen Stichproben nicht als Ganzes, sondern unterteilt in Teilgruppen beschrieben werden.

Ein klassisches Beispiel ist der Aufbau einer experimentellen Studie mit Versuchs- und Kontrollgruppe (klin. Kontext: Verum vs. Placebo).

```
df = data.frame(Gruppe = c("Experiment", "Kontrolle", "Experiment", "Kontrolle", "Kontrolle", "Experiment",
                           Alter = c(22, 41, 51, 39, 71, 25),
                           Konzentration = c(3, 7, 4, 7, 5, 2),
                           Schlafdauer = c(6, 14, 7, 12, 15, 11),
                           Schlafqualität = c(5, 8, 2, 1, 8, 5))
df

##      Gruppe Alter Konzentration Schlafdauer Schlafqualität
## 1 Experiment    22            3          6           5
## 2 Kontrolle     41            7         14           8
## 3 Experiment    51            4          7           2
## 4 Kontrolle     39            7         12           1
## 5 Kontrolle     71            5         15           8
## 6 Experiment    25            2         11           5
```

Der manuelle Weg würde eine Datenselektion erfordern (z.B. für das Durchschnittsalter der Kontrollgruppe):

```
mean(df$Alter[df$Gruppe == "Kontrolle"])
```

```
## [1] 50.33333
```

Wollte man die gesamte Tabelle so zusammenfassen, z.B. für die tabellarische Stichprobenbeschreibung in einem Paper, würde einem dies jedoch einige Mühe bereiten.

Glücklicherweise können wir eine Variante der `describe()` Funktion nutzen, nämlich `describeBy()`

```
describeBy(df, group = "Gruppe")

##
## Descriptive statistics by group
## Gruppe: Experiment
##           vars n   mean      sd median trimmed  mad min max range skew
## Gruppe*       1 3  1.00  0.00      1   1.00 0.00   1   1     0  NaN
## Alter         2 3 32.67 15.95     25  32.67 4.45  22  51    29 0.37
```

```

## Konzentration      3 3  3.00  1.00      3   3.00 1.48  2   4    2  0.00
## Schlafdauer       4 3  8.00  2.65      7   8.00 1.48  6  11    5  0.32
## Schlafqualität    5 3  4.00  1.73      5   4.00 0.00  2   5    3 -0.38
##                           kurtosis   se
## Gruppe*             NaN 0.00
## Alter                -2.33 9.21
## Konzentration        -2.33 0.58
## Schlafdauer          -2.33 1.53
## Schlafqualität        -2.33 1.00
## -----
## Gruppe: Kontrolle
##           vars n  mean     sd median trimmed  mad min max range skew
## Gruppe*          1 3  2.00  0.00      2   2.00 0.00  2   2    0  NaN
## Alter            2 3 50.33 17.93     41  50.33 2.97  39  71   32  0.38
## Konzentration    3 3  6.33  1.15      7   6.33 0.00  5   7    2 -0.38
## Schlafdauer      4 3 13.67  1.53     14  13.67 1.48  12  15   3 -0.21
## Schlafqualität   5 3  5.67  4.04      8   5.67 0.00  1   8    7 -0.38
##                           kurtosis   se
## Gruppe*             NaN 0.00
## Alter                -2.33 10.35
## Konzentration        -2.33 0.67
## Schlafdauer          -2.33 0.88
## Schlafqualität        -2.33 2.33

```

Eine weitere, sogar noch etwas besser für Paper nutzbare Option, ist die `report_sample()` Funktion aus dem `report` Paket:

```

library(report)

report_sample(df, group_by = "Gruppe")

## # Descriptive Statistics
##
## Variable           | Experiment (n=3) | Kontrolle (n=3) | Total (n=6)
## -----
## Mean Alter (SD)   | 32.67 (15.95) | 50.33 (17.93) | 41.50 (18.00)
## Mean Konzentration (SD) | 3.00 (1.00) | 6.33 (1.15) | 4.67 (2.07)
## Mean Schlafdauer (SD) | 8.00 (2.65) | 13.67 (1.53) | 10.83 (3.66)
## Mean Schlafqualität (SD) | 4.00 (1.73) | 5.67 (4.04) | 4.83 (2.93)

```

So eine tabellarische Stichprobenbeschreibung könnte man direkt für ein Paper verwenden.

Chapter 10

Daten bearbeiten

10.1 Daten hinzufügen

10.1.1 Variablen hinzufügen

Mit den Operatoren `$` und Zuweisung `=` können Sie neue Spalten zu einem Datenrahmen hinzufügen. Dazu verwenden Sie einfach die Notation `df$Name` und weisen ihm einen neuen Datenvektor (Variable) zu.

Erstellen wir zum Beispiel einen `data.frame` namens `test` mit zwei Spalten: `ID` und `Alter`:

```
test = data.frame("ID" = c(1, 2, 3, 4, 5),
                  "Alter" = c(24, 25, 42, 56, 22))
```

Lassen Sie uns nun eine neue Spalte namens `Geschlecht` aus einem Vektor mit Geschlechtsdaten hinzufügen:

```
test$Geschlecht = c("männlich", "weiblich", "weiblich", "männlich", "weiblich")
```

Hier ist das Ergebnis:

```
test
```

```
##   ID Alter Geschlecht
## 1  1    24 männlich
## 2  2    25 weiblich
## 3  3    42 weiblich
## 4  4    56 männlich
## 5  5    22 weiblich
```

Dasselbe funktioniert auch mit eckigen Klammern `[]`:

```
test["Lieblingsfilm"] = c("Titanic", "Herr der Ringe", "Harry Potter", "Titanic", "Matrix")

test

##   ID Alter Geschlecht Lieblingsfilm
## 1  1    24 männlich     Titanic
## 2  2    25 weiblich Herr der Ringe
## 3  3    42 weiblich Harry Potter
## 4  4    56 männlich     Titanic
## 5  5    22 weiblich      Matrix
```

Diese Methoden hängen neue Spalten immer ganz hinten (ganz rechts) an den data.frame an. Wollen wir die neue Variable an einer bestimmten Position des data.frames einfügen, nutzen wir die `add_column()` Funktion aus dem `tibble` Paket.

Wollen wir zum Beispiel eine Variable an die 2. Position des Datensatzes einfügen nutzen wir und spezifizieren das `.after` oder `.before` Argument mit dem Index oder dem Namen der vorherigen/folgenden Variable:

```
library(tibble)

test = add_column(test, Haustier = c("Hund", "Katze", "Hund", "Hund", "Fische"), .after = 2)

# oder

# test = add_column(test, Haustier = c("Hund", "Katze", "Hund", "Hund", "Fische"), .before = 3)

test

##   ID Haustier Alter Geschlecht Lieblingsfilm
## 1  1      Hund   24 männlich     Titanic
## 2  2      Katze   25 weiblich Herr der Ringe
## 3  3      Hund   42 weiblich Harry Potter
## 4  4      Hund   56 männlich     Titanic
## 5  5      Fische  22 weiblich      Matrix
```

10.2 Daten löschen

10.2.1 Variablen löschen

Wir möchten aus unserem `test` Datensatz die Variable `Lieblingsfilm` wieder löschen.

Der einfachste Weg um eine Variable zu löschen funktioniert wieder einmal mit den `$` Operator.

```
test$Lieblingsfilm = NULL

test

##   ID Haustier Alter Geschlecht
## 1  1     Hund   24 männlich
## 2  2     Katze   25 weiblich
## 3  3     Hund   42 weiblich
## 4  4     Hund   56 männlich
## 5  5    Fische  22 weiblich
```

Wollen wir mehrere Variablen auf einmal löschen, empfiehlt sich wie bereits bei der Auswahl von Variablen die `select()` Funktion aus dem `dplyr` Paket. Um wieder etwas zum Löschen zu haben, erstellen wir zunächst 3 leere Dummy Variablen:

```
test$ZumLoeschen1 = NA
test$ZumLoeschen2 = NA
test$ZumLoeschen3 = NA

test

##   ID Haustier Alter Geschlecht ZumLoeschen1 ZumLoeschen2 ZumLoeschen3
## 1  1     Hund   24 männlich        NA        NA        NA
## 2  2     Katze   25 weiblich        NA        NA        NA
## 3  3     Hund   42 weiblich        NA        NA        NA
## 4  4     Hund   56 männlich        NA        NA        NA
## 5  5    Fische  22 weiblich        NA        NA        NA
```

Nun löschen wir diese 3 Variablen wieder

```
library(dplyr)

test = select(test, -c(ZumLoeschen1, ZumLoeschen2, ZumLoeschen3))

# oder

# test = select(test, -contains("ZumLoeschen"))

test

##   ID Haustier Alter Geschlecht
## 1  1     Hund   24 männlich
## 2  2     Katze   25 weiblich
## 3  3     Hund   42 weiblich
## 4  4     Hund   56 männlich
## 5  5    Fische  22 weiblich
```

10.2.2 Fälle löschen

Die einfachste Art Fälle zu löschen, ist das Überspeichern des data.frames ohne die zu löschen Fällen. Diese Selektion machen wir einfach mit den eckigen Klammern [] und einem -

Wir wollen die 2. Person im test data.frame löschen:

```
test = test[-2,]

test

##   ID Haustier Alter Geschlecht
## 1  1     Hund   24 männlich
## 3  3     Hund   42 weiblich
## 4  4     Hund   56 männlich
## 5  5    Fische  22 weiblich
```

Genau wie im Kapitel Daten auswählen gelernt, können wir auch die zu löschen Personen nach einer Logik auswählen:

Wir wollen alle Personen mit Geschlecht == "männlich" löschen:

```
test = test[-c(test$Geschlecht == "männlich"),]

# oder

# test = test[c(test$Geschlecht != "männlich"),]

test

##   ID Haustier Alter Geschlecht
## 3  3     Hund   42 weiblich
## 4  4     Hund   56 männlich
## 5  5    Fische  22 weiblich
```

10.3 Variablen umbenennen

10.3.1 Variablennamen ändern

Um uns die Variablennamen anzeigen zu lassen, benutzen wir `names()`:

```
names(test)

## [1] "ID"          "Haustier"      "Alter"        "Geschlecht"
```

Dieselbe Funktion können wir auch nutzen um Variablen umzubenennen.

Wir wollen die 1. Variable anstelle von ID gerne Idenfikation nennen:

```
names(test)[1] = "Idenfikation"

test

##   Idenfikation Haustier Alter Geschlecht
## 3          3     Hund    42  weiblich
## 4          4     Hund    56  männlich
## 5          5   Fische   22  weiblich
```

Dasselbe funktioniert auch mit logischen Argumenten (die Variable, die ID heisst...)

```
names(test)[names(test) == "ID"] = "Idenfikation"
```

Wollen wir mehrere Variablen auf einmal umbenennen eimpfiehlt sich die `rename()` Funktion aus `dplyr`. Wir bauen uns zunächst noch einmal 3 leere Dummy Variablen, die wir dann umbenennen:

```
test$A = NA
test$B = NA
test$C = NA

test

##   Idenfikation Haustier Alter Geschlecht  A  B  C
## 3          3     Hund    42  weiblich NA NA NA
## 4          4     Hund    56  männlich NA NA NA
## 5          5   Fische   22  weiblich NA NA NA
```

Nun zum Umbenennen:

```
test = rename(test, c(D = A,
                     E = B,
                     F = C))

test

##   Idenfikation Haustier Alter Geschlecht  D  E  F
## 3          3     Hund    42  weiblich NA NA NA
## 4          4     Hund    56  männlich NA NA NA
## 5          5   Fische   22  weiblich NA NA NA
```

10.4 Daten verändern

10.4.1 Werte ändern

Die einfachste Art zum Ändern von Werten ist wie immer das Selektieren und Überspeichern.

Schauen wir uns einmal die `Alter` Variable in unserem `test` `data.frame` an:

```
test

##   ID Alter Geschlecht
## 1  1    24 männlich
## 2  2    25 weiblich
## 3  3    42 weiblich
## 4  4    56 männlich
## 5  5    22 weiblich
```

Wenn ich beispielsweise alle 25 Jahre alten Leute 35 Jahre alt werden lassen wollte, muss ich diese nur selektieren und überspeichern:

```
test$Alter[test$Alter == 25] = 35

test

##   ID Alter Geschlecht
## 1  1    24 männlich
## 2  2    35 weiblich
## 3  3    42 weiblich
## 4  4    56 männlich
## 5  5    22 weiblich
```

10.4.2 Recodieren

Manchmal ist es nützlich, Werte nicht nacheinander zu überspeichern, sondern simultan (also alle parallel) zu ändern.

Nehmen wir einmal an, ich hätte ein Fragebogenitem zum Thema Introversion (Rating von 1-5). Die Frage lautet “Gehen Sie gerne auf Parties”?

```
test$item = c(1, 3, 5, 4, 2)

test

##   ID Alter Geschlecht Item
## 1  1    24 männlich    1
## 2  2    35 weiblich    3
## 3  3    42 weiblich    5
```

```
## 4 4      56 männlich 4
## 5 5      22 weiblich 2
```

Dieses Item gibt uns eine nützliche Information hinsichtlich der Introversion der Person, denn jemand sehr introvertiert würde vermutlich eine niedrige Zahl ankreuzen (z.B. Person 1). Um sie ggf. mit anderen Items zu verrechnen müssen wir Items jedoch häufig umdrehen (umpolen), sodass hohe Werte auch eine hohe Ausprägung des Konstrukts reflektieren.

Mein Ziel ist also folgendes:

1 → 5 2 → 4 3 → 3 4 → 2 5 → 1

Der 1. Impuls wäre folgendes:

```
# test$Item[test$Item == 1] = 5
```

Das Problem mit dieser Methode ist, dass alle ursprünglichen 1er Werte die ich zu 5er Werten mache, sich wieder zu 1er Werten verändern, sobald ich die ursprünglichen 5er in 1er verändere. Ich muss also alle Werte “auf einmal” (parallel) ändern.

Dafür nutzen wir die `recode()` Funktion aus dem Paket `dplyr`:

```
test$Item_r = recode(test$Item, "1" = 5, "2" = 4, "4" = 2, "5" = 1)

test

##   ID Alter Geschlecht Item Item_r
## 1  1     24 männlich   1      5
## 2  2     35 weiblich   3      3
## 3  3     42 weiblich   5      1
## 4  4     56 männlich   4      2
## 5  5     22 weiblich   2      4
```

10.4.3 Worte ändern

Dasselbe funktioniert natürlich für Worte. Lassen Sie uns `männlich` und `weiblich` doch einmal in `Mann` und `Frau` verändern:

```
test$Geschlecht[test$Geschlecht == "männlich"] = "Mann"
test$Geschlecht[test$Geschlecht == "weiblich"] = "Frau"

test
```

```
##   ID Alter Geschlecht Item Item_r
## 1  1     24      Mann   1      5
## 2  2     35      Frau   3      3
## 3  3     42      Frau   5      1
## 4  4     56      Mann   4      2
```

```
## 5 5 22     Frau 2 4
```

Manchmal haben wir jedoch Variablen mit unterschiedlichen Ausprägungen, die alle einen bestimmten Wortteil haben, der verändert werden soll.

Nehmen wir einmal folgende Variable:

```
test$Gruppe = c("GruppeA", "GruppeB", "GruppeC", "GruppeD", "GruppeE")
test

##   ID Alter Geschlecht Item Item_r Gruppe
## 1  1    24      Mann   1      5 GruppeA
## 2  2    35      Frau   3      3 GruppeB
## 3  3    42      Frau   5      1 GruppeC
## 4  4    56      Mann   4      2 GruppeD
## 5  5    22      Frau   2      4 GruppeE
```

Nehmen wir an, ich wollte das Wort **Gruppe** in **Klasse** verändern die Bezeichnungen A-E aber beibehalten. Dann muss ich einen bestimmten Worteil für alle Werte in der Variable ersetzen. Dafür ist die Funktion `gsub()` ideal:

```
test$Gruppe = gsub(pattern = "Gruppe", replacement = "Klasse", x = test$Gruppe)
test

##   ID Alter Geschlecht Item Item_r Gruppe
## 1  1    24      Mann   1      5 KlasseA
## 2  2    35      Frau   3      3 KlasseB
## 3  3    42      Frau   5      1 KlasseC
## 4  4    56      Mann   4      2 KlasseD
## 5  5    22      Frau   2      4 KlasseE
```

Dasselbe lässt sich natürlich auf für Variablennamen anwenden. Wir erstellen noch einmal 3 leere Dummy Variablen

```
test$Test1 = NA
test$Test2 = NA
test$Test3 = NA

test

##   ID Alter Geschlecht Item Item_r Gruppe Test1 Test2 Test3
## 1  1    24      Mann   1      5 KlasseA    NA    NA    NA
## 2  2    35      Frau   3      3 KlasseB    NA    NA    NA
## 3  3    42      Frau   5      1 KlasseC    NA    NA    NA
## 4  4    56      Mann   4      2 KlasseD    NA    NA    NA
## 5  5    22      Frau   2      4 KlasseE    NA    NA    NA
```

Nun benennen wir sie mit `gsub()` automatisch um. Wir wollen probieren, dass **Test** zu **Item** wird:

```
names(test) = gsub(pattern = "Test", replacement = "Item", x = names(test))
```

10.4.4 Faktorstufen ändern

Zunächst legen wir einmal eine Faktorvariable an. Wir wollen die Variable `Diagnose` codieren für die Stufen Depression, Angststörung und Sucht.

```
test$Diagnose = factor(c("Depression", "Angststörung", "Sucht", "Depression", "Angststörung"))

test$Diagnose

## [1] Depression  Angststörung  Sucht      Depression  Angststörung
## Levels: Angststörung Depression  Sucht
```

Wir wollen die Namen der Faktorstufen z.B. etwas abgekürzt haben. Wir gehen genauso vor wie bei der Umbenennung von Variablen, nur nutzen wir statt `names()` den Befehl `levels()`:

```
levels(test$Diagnose) = c("Angst", "Depr.", "Sucht")

test$Diagnose

## [1] Depr. Angst Sucht Depr. Angst
## Levels: Angst Depr. Sucht
```

Wie wir sehen, sortiert R Faktorstufen zunächst einmal immer alphabetisch. Vielleicht wollen wir, dass anstelle von Angst die Depression die 1. Faktorstufe ist (z.B. zur Darstellung in einer Graphik o.ä.). Dafür nutzen wir die Funktion `relevel()`

```
test$Diagnose = relevel(test$Diagnose, "Depr.")

test$Diagnose

## [1] Depr. Angst Sucht Depr. Angst
## Levels: Depr. Angst Sucht
```

10.5 Daten sortieren

Frisch gesammelte Daten können manchmal in ungeordneter Form bei uns ankommen. Eine Möglichkeit Ordnung in die Daten zu bekommen, ist das Sortieren der Daten, welches i.d.R. nach der Logik einer bestimmten Variable erfolgt.

10.5.1 Sortieren nach numerischen Variablen

Lassen Sie uns zur Demonstration einen **ungeordneten** Test dataframe erstellen:

```
test <- data.frame("ID" = c(3, 5, 1, 2, 4),
                    "Alter" = c(24, 25, 42, 56, 22))

# dataframe ansehen:

test

##   ID Alter
## 1  3    24
## 2  5    25
## 3  1    42
## 4  2    56
## 5  4    22
```

Der data.frame ist weder nach der ID Variable, noch nach dem inhaltlichen Kriterium des Alters (jung nach alt vs. alt nach jung) geordnet.

Für ein schnelles Sortieren nutzen wir die Funktion `order()`. Diese wenden wir auf die Variable an, nach der wir sortieren wollen und schreiben den Befehl links vom Komma in den eckigen Klammern (Erinnerung: links vom Komma bezieht sich immer auf Zeilen, rechts auf die Spalten). Den sortierten data.frame wollen wir `test2` nennen.

```
test2 = test[order(test$ID),]

# sortierten dataframe ansehen:

test2

##   ID Alter
## 3  1    42
## 4  2    56
## 1  3    24
## 5  4    22
## 2  5    25
```

Dasselbe funktioniert mit der Altervariable (von jung nach alt).

```
test3 = test[order(test$Alter),]

# sortierten dataframe ansehen:

test3

##   ID Alter
## 1  3    24
## 2  5    25
## 3  1    42
## 4  2    56
## 5  4    22
```

```
## 5 4    22
## 1 3    24
## 2 5    25
## 3 1    42
## 4 2    56
```

Wollten wir den dataframe anhand der Altervariable von alt nach jung sortieren, benutzen wir das Zusatzargument `decreasing = TRUE`:

```
test4 = test[order(test$Alter, decreasing = TRUE),]

# sortierten dataframe ansehen:

test4

##   ID Alter
## 4  2    56
## 3  1    42
## 2  5    25
## 1  3    24
## 5  4    22
```

10.5.2 Sortieren nach Variablen mit Werten

Wollen wir unseren dataframe nach einer Variable sortieren, die Worte enthält, führt `order()` zu einer alphabetischen Sortierung. Wir hängen zum Ausprobieren eine character Variable an unseren `test` data.frame:

```
test$name = c("Klaus", "Jan", "Anna", "John", "Cleo")

test

##   ID Alter  Name
## 1  3    24 Klaus
## 2  5    25 Jan
## 3  1    42 Anna
## 4  2    56 John
## 5  4    22 Cleo
```

Sortieren mit `order()` anhand der Variable `Name` ergibt:

```
test5 = test[order(test$name),]

test5

##   ID Alter  Name
## 3  1    42 Anna
## 5  4    22 Cleo
```

```
## 2 5 25 Jan
## 4 2 56 John
## 1 3 24 Klaus
```

10.6 Datensätze zusammensetzen

Oft kommt es vor, dass wir zwei data.frames zusammensetzen wollen. Dies könnte der Fall sein, wenn wir unsere Daten in 2 unterschiedlichen Tabellen gesammelt haben, wir aber nun mit den kombinierten Daten rechnen wollen. Damit beschäftigen wir uns in den nächsten Abschnitten.

10.6.1 cbind und rbind

Beim Zusammensetzen von dataframes sind 2 Szenarien denkbar:

1. **Spalten** des einen dataframes rechts an die Spalten des anderen dataframes (beide dataframes haben gleich viele Zeilen)
2. **Zeilen** des einen dataframes unten an die Zeilen des anderen dataframes (beide dataframes haben gleich viele Spalten)

Dafür gibt es die Funktionen `cbind()` (columns aka Spalten verbinden) und `rbind()` (rows aka Zeilen verbinden).

10.6.1.1 cbind

Zur Demonstration der `cbind()` Funktion bauen wir uns 2 kleine Test-dataframes. Wichtig ist, dass diese die selbe Anzahl von Zeilen haben:

```
test1 <- data.frame("ID" = c(1, 2, 3, 4, 5),
                     "Alter" = c(24, 25, 42, 56, 22))

test2 <- data.frame("IQ" = c(100, 102, 98, 90, 121),
                     "Neurotizismus" = c(11, 42, 31, 22, 13))
```

Hier haben wir den dataframe test1:

```
test1

##   ID Alter
## 1  1    24
## 2  2    25
## 3  3    42
## 4  4    56
## 5  5    22
```

Und hier den dataframe test2:

```
test2

##   IQ Neurotizismus
## 1 100          11
## 2 102          42
## 3  98          31
## 4  90          22
## 5 121          13
```

Zum Zusammensetzen der beiden (test2 rechts an test 1 “dranhängen”) schreiben wir beide dataframes in die `cbind()` Funktion. Der kombinierte Datensatz soll test 3 heißen:

```
test3 = cbind(test1, test2)

# Ergebnis ansehen:

test3

##   ID Alter  IQ Neurotizismus
## 1  1    24 100          11
## 2  2    25 102          42
## 3  3    42  98          31
## 4  4    56  90          22
## 5  5    22 121          13
```

10.6.1.2 rbind

Zur Demonstration der `rbind()` Funktion bauen wir uns 2 weitere kleine Test-dataframes. Wichtig ist, dass diese die selbe Anzahl von Spalten haben und dass diese die selben Variablennamen haben:

```
test1 <- data.frame("ID" = c(1, 2, 3, 4, 5),
                     "Alter" = c(24, 25, 42, 56, 22))

test2 <- data.frame("ID" = c(6, 7, 8, 9, 10),
                     "Alter" = c(27, 35, 31, 66, 51))
```

Hier haben wir den dataframe test1:

```
test1

##   ID Alter
## 1  1    24
## 2  2    25
## 3  3    42
## 4  4    56
```

```
## 5 5 22
```

Und hier den dataframe test2:

```
test2
```

```
##   ID Alter
## 1  6    27
## 2  7    35
## 3  8    31
## 4  9    66
## 5 10    51
```

Zum Zusammensetzen der beiden (test2 unten an test 1 „dranhängen“) schreiben wir beide dataframes in die `rbind()` Funktion. Der kombinierte Datensatz soll test 3 heißen:

```
test3 = rbind(test1, test2)
```

Ergebnis ansehen:

```
test3
```

```
##   ID Alter
## 1  1    24
## 2  2    25
## 3  3    42
## 4  4    56
## 5  5    22
## 6  6    27
## 7  7    35
## 8  8    31
## 9  9    66
## 10 10    51
```

10.6.2 merge

Haben wir 2 ungeordnete dataframes mit Daten derselben Personen, jedoch mit unterschiedlichen Variablen, die wir zusammensetzen wollen, haben wir das Problem, dass die Zeilenreihenfolge des einen dataframes ggf. nicht mit der Zeilenreihenfolge des anderen dataframes übereinstimmt:

```
test1 <- data.frame("ID" = c(1, 2, 3, 4, 5),
                     "Alter" = c(24, 25, 42, 56, 22))

test2 <- data.frame("ID" = c(2, 5, 3, 1, 4),
                     "IQ" = c(100, 102, 98, 90, 121))
```

Setzen wir die dataframes einfach mit `cbind()` aneinander, bekommt z.B. Person mit der `ID == 1` den IQ Wert der Person mit `ID == 2`, da diese beide in der 1. Zeile der dataframes stehen.

Wir könnten natürlich beide dataframes erst mit `order()` nach der ID Variable sortieren und sie anschließend mit `cbind()` kombinieren. Schneller geht es jedoch mit der Funktion `merge()`. Diese braucht als Information nur eine Variable, die R sagt welche Zeilen zusammengehören. Da ID eindeutige Werte aufweist und in beiden dataframes vorhanden ist, nutzen wir diese Variable für den merge:

```
test3 = merge(test1, test2, by = "ID")

test3

##   ID Alter  IQ
## 1  1    24  90
## 2  2    25 100
## 3  3    42  98
## 4  4    56 121
## 5  5    22 102
```

10.7 Datensätze transformieren

10.7.1 Von Wide- und Long-Format

Datensätze können entweder im Wide- oder Long-Format vorliegen, wobei jede Formatierung ihre eigenen Vor- und Nachteile aufweist.

10.7.1.1 Wide-Format

Im Wide-Format werden Daten normalerweise in einer breiten Tabelle dargestellt, wobei jede Variable eine eigene Spalte hat. Diese Darstellung eignet sich gut, wenn der Fokus auf einer schnellen Datenexploration und einer einfachen Zusammenfassung liegt. Es ermöglicht eine übersichtliche Sicht auf die Daten, insbesondere wenn es viele Variablen gibt.

Ein Großteil der Daten, die wir direkt nach einer Datenerhebung haben, liegen im Wide-Format vor.

Schauen wir uns ein kleines Datenbeispiel für $N = 5$ Personen an:

```
ID = c(1, 2, 3, 4, 5)
Geschlecht = c("männlich", "weiblich", "männlich", "männlich", "weiblich")
Depression_T1 = c(12, 16, 21, 8, 13)
Depression_T2 = c(7, 4, 12, 7, 11)
```

```

Depression_T3 = c(7, 4, 12, 7, 11)
df_wide = data.frame(ID, Geschlecht, Depression_T1, Depression_T2, Depression_T3)

df_wide

##   ID Geschlecht Depression_T1 Depression_T2 Depression_T3
## 1  1   männlich          12            7            7
## 2  2   weiblich          16            4            4
## 3  3   männlich          21           12           12
## 4  4   männlich           8            7            7
## 5  5   weiblich          13           11           11

```

Wichtig: Jede Person hat eine Zeile. Gibt es Messwiederholungen (hier T1, T2 und T3 des Depressionswerte; T = Time), erhält jede Messung seine eigene Spalte.

Mit jeder zusätzlichen Spalte (bzw. Messung) wird der Datensatz breiter, daher der Name “Wide-Format”.

10.7.1.2 Long-Format

Im Gegensatz dazu wird das Long-Format verwendet, um Daten in einer schmäleren Tabelle darzustellen, in der mehrere Variablen in einer Spalte zusammengefasst werden. Jede Beobachtung erstreckt sich über mehrere Zeilen, wodurch eine längere Tabelle entsteht. Das Long-Format eignet sich besonders für die Analyse von Panel- oder Zeitreihendaten, da es die Verarbeitung von wiederholten Messungen erleichtert und die Durchführung statistischer Analysen auf individueller Ebene ermöglicht.

Wichtig:

- Jede Zeile muss mittels einer ID Variable eindeutig den Personen zugeordnet werden
- Eine weitere Variable (bei Messwiederholungen z.B. **Zeit**) muss angeben, weshalb es mehrere Werte pro Fall gibt

Schauen wir uns das Datenbeispiel der von eben nun noch einmal $N = 5$ Personen von eben nun im Long-Format an:

```

ID = c(1, 2, 3, 4, 5,
      1, 2, 3, 4, 5,
      1, 2, 3, 4, 5)
Geschlecht = c("männlich", "weiblich", "männlich", "männlich", "weiblich",
             "männlich", "weiblich", "männlich", "männlich", "weiblich",
             "männlich", "weiblich", "männlich", "männlich", "weiblich")
Depression = c(12, 16, 21, 8, 13,
              7, 4, 12, 7, 11,
              7, 4, 12, 7, 11)

```

```

Time = c(1, 1, 1, 1, 1,
       2, 2, 2, 2, 2,
       3, 3, 3, 3, 3)

df_long = data.frame(ID, Geschlecht, Depression, Time)

df_long

##   ID Geschlecht Depression Time
## 1  1 männlich      12     1
## 2  2 weiblich      16     1
## 3  3 männlich      21     1
## 4  4 männlich       8     1
## 5  5 weiblich      13     1
## 6  1 männlich       7     2
## 7  2 weiblich       4     2
## 8  3 männlich      12     2
## 9  4 männlich       7     2
## 10 5 weiblich      11     2
## 11 1 männlich       7     3
## 12 2 weiblich       4     3
## 13 3 männlich      12     3
## 14 4 männlich       7     3
## 15 5 weiblich      11     3

```

Nun hat jede Person so viele Zeilen, wie sie Messwerte hat. Einträge konstanter Variablen (z.B. Geschlecht) werden wiederholt.

Mit jeder zusätzlichen Zeile (bzw. Messung) wird der Datensatz länger, daher der Name “Long-Format”.

10.7.1.3 Wide-Format nach Long-Format

Wollen wir einen Datensatz, der im Wide-Format vorliegt in das Long-Format überführen, nutzen wir die `pivot_longer()` Funktion aus dem `tidyverse` Paket:

```

library(tidyverse)

## -- Attaching core tidyverse packages ----- tidyverse 2.0.0 --
## vforcats    1.0.0     vstringr    1.5.0
## v lubridate 1.9.2     vtidy       1.3.0
## v purrr     1.0.1
## -- Conflicts ----- tidyverse_conflicts() --
## x psych::%+()%> masks ggplot2::%+()%>
## x psych::alpha()  masks ggplot2::alpha()

```

```

## x dplyr::filter() masks stats::filter()
## x dplyr::lag()    masks stats::lag()
## i Use the conflicted package (<http://conflicted.r-lib.org/>) to force all conflicts
df_wide_to_long = pivot_longer(data = df_wide,
                               cols = c("Depression_T1", "Depression_T2", "Depression_T3"),
                               names_to = "Time",
                               values_to = "Depression")

df_wide_to_long

## # A tibble: 15 x 4
##       ID Geschlecht Time      Depression
##   <dbl> <fct>     <chr>      <dbl>
## 1     1 männlich  Depression_T1    12
## 2     1 männlich  Depression_T2     7
## 3     1 männlich  Depression_T3     7
## 4     2 weiblich  Depression_T1   16
## 5     2 weiblich  Depression_T2     4
## 6     2 weiblich  Depression_T3     4
## 7     3 männlich  Depression_T1   21
## 8     3 männlich  Depression_T2   12
## 9     3 männlich  Depression_T3   12
## 10    4 männlich  Depression_T1     8
## 11    4 männlich  Depression_T2     7
## 12    4 männlich  Depression_T3     7
## 13    5 weiblich  Depression_T1   13
## 14    5 weiblich  Depression_T2   11
## 15    5 weiblich  Depression_T3   11

```

10.7.1.4 Long-Format nach Wide-Format

Wollen wir einen Datensatz, der im Long-Format vorliegt in das Wide-Format überführen, nutzen wir die `pivot_wider()` Funktion aus dem `tidyverse` Paket:

```

library(tidyverse)

df_long_to_wide = pivot_wider(data = df_long,
                               names_from = "Time",
                               names_prefix = "Depression_",
                               values_from = "Depression")

df_long_to_wide

## # A tibble: 5 x 5
##       ID Geschlecht Depression_1 Depression_2 Depression_3
##   <dbl> <fct>     <dbl>      <dbl>      <dbl>
## 1     1 männlich    12         7         7
## 2     1 männlich    12         7         7
## 3     1 männlich    12         7         7
## 4     2 weiblich   16         4         4
## 5     2 weiblich   16         4         4

```

```
##   <dbl> <fct>          <dbl>          <dbl>          <dbl>
## 1     1 männlich        12            7            7
## 2     2 weiblich        16            4            4
## 3     3 männlich        21           12           12
## 4     4 männlich         8            7            7
## 5     5 weiblich        13           11           11
```

10.7.1.5 Fazit: Wann welches Format?

Der Unterschied zwischen Wide- und Long-Format hängt also von den spezifischen Anforderungen der Datenanalyse ab. Das Wide-Format bietet eine übersichtliche und schnelle Datenexploration, während das Long-Format eine detaillierte Analyse auf individueller Ebene ermöglicht. Die Wahl des richtigen Formats hängt von der Art der Daten, den Analysezielen und den verwendeten statistischen Verfahren ab.

Chapter 11

Schleifen (Loops)

11.1 for()-Loop

Der `for()`-Loop ist eine gängige Schleifenstruktur in R. Mit dem `for()`-Loop können wiederholte Aufgaben automatisiert und effizient ausgeführt werden. Während jeder Iteration des Loops wird der Codeblock innerhalb des Loops ausgeführt. Dabei kann auf den aktuellen Iterationsschritt zugegriffen werden, um den Code entsprechend anzupassen. Der `for()`-Loop ist besonders nützlich, um über eine vordefinierte Sequenz von Elementen zu iterieren, wie zum Beispiel eine Vektor oder eine Liste. Durch die effektive Nutzung des `for()`-Loops können wiederholte Aufgaben in R effizient gelöst werden.

Die `for()` Schleife wiederholt alles was wir wollen, so oft wir wollen. Man arbeitet hier mit einem so genannten Index (häufig nutzt man die Abkürzung `i`), der z.B. die Position innerhalb des Vektors anzeigt.

Nehmen wir als Beispiel einen Vektor von Zahlen 1 bis 10

```
1:10
```

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

Der Index nimmt nacheinander die Form des 1. bis letzten Objekts des Vektors an. Probieren wir einmal mit der `print()` Funktion **nacheinander** die Zahl `i` für `1:10` anzeigen zu lassen. Der `for()` Loop hat immer folgende Form:

```
# for (i in vector) {  
#   ...  
# }
```

```
# Für unser Vorhaben:
```

```
for (i in 1:10) {
    print(i)
}

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

Alles was zwischen den {} steht, ist der auszuführende Teil des Programms.

Wörtlich macht der Loop folgendes:

1. Start
2. i ist = 1
3. Drucke i mit der Funktion print()
4. Durchlauf der Schleife fertig
5. Beginn der Schleife von Anfang solange i nicht = 10 ist
6. i ist = 2
7. Drucke i mit der Funktion print()
8. Durchlauf der Schleife fertig
9. Beginn der Schleife von Anfang solange i nicht = 10 ist
10. i ist = 3 ...
11. i ist = 10
12. Drucke i mit der Funktion print()
13. Durchlauf der Schleife fertig
14. i = 10, Ende des for() Loops

Da der Computer sehr schnell ist, werden alle Zahlen von 1 bis 10 scheinbar simultan ausgegeben. Wollen wir die einzelnen Schritte der Schleife beobachten, können wir die Funktion `Sys.sleep()` einfügen, um nach jedem Durchlauf eine kleine Pause einzulegen:

```
for (i in 1:10) {
    print(i)
    Sys.sleep(time = 1) # macht 1 Sekunde Pause
}

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

```
## [1] 5
## [1] 6
## [1] 7
## [1] 8
## [1] 9
## [1] 10
```

Das ganze funktioniert auch in Kombination mit den eckigen Klammern.

```
vector = c("A", "B", "C", "D", "E", "F", "G", "H", "I", "J")

for (i in 1:10) {
  print(vector[i])
  Sys.sleep(time = 1) # macht 1 Sekunde Pause
}

## [1] "A"
## [1] "B"
## [1] "C"
## [1] "D"
## [1] "E"
## [1] "F"
## [1] "G"
## [1] "H"
## [1] "I"
## [1] "J"
```

Nun druckt die `print()` Funktion das i-te Element aus unserem Vektor

11.1.1 for() im Datensatz

Das Prinzip des `for()`-Loops lässt sich nutzen, wenn wir mit Datensätzen arbeiten. Wir lassen einfach die Schleife über die Zeilen bzw. Spalten des Datensatzes laufen.

Erstellen wir einen kleinen Beispieldatensatz für einen Fragebogen, der 5 Depressionssymptome erfasst (Skala von 1-5):

```
df = data.frame(
  ID = 1:5,
  Stimmung = c(2, 4, 5, 1, 3),
  Antriebslosigkeit = c(3, 1, 2, 1, 4),
  Interessensverlust = c(1, 3, 5, 5, 5),
  Schlafprobleme = c(3, 1, 5, 2, 2),
  Suizidalität = c(2, 1, 5, 4, 3)
)
```

```
df
```

	ID	Stimmung	Antriebslosigkeit	Interessensverlust	Schlafprobleme	Suizidalität
## 1	1	2		3	1	3
## 2	2	4		1	3	1
## 3	3	5		2	5	5
## 4	4	1		1	5	2
## 5	5	3		4	5	2

Wir könnten uns z.B. automatisch nacheinander die Werte der Variable **Stimmung** anzeigen lassen. Dafür lassen wir die Schleife von der 1. bis zur letzten Zeile (`ncol(df)`) laufen:

```
for (i in 1:nrow(df)) {
  print(df$Stimmung[i])
  Sys.sleep(time = 1) # macht 1 Sekunde Pause
}

## [1] 2
## [1] 4
## [1] 5
## [1] 1
## [1] 3
```

Alles zwischen den geschwungenen Klammern wird dabei vom Programm ausgeführt. Hier könnten wir alles reinschreiben was wir wollen. Lassen wir uns automatisch die Mittelwerte aller Depressionssymptomvariablen anzeigen:

```
for (i in 2:ncol(df)) {
  print(mean(df[,i]))
  Sys.sleep(time = 1) # macht 1 Sekunde Pause
}

## [1] 3
## [1] 2.2
## [1] 3.8
## [1] 2.6
## [1] 3
```

Dafür lassen wir die Variable über Spalten laufen. Da die erste Spalte keine Depressionssymptomvariable ist (**ID**), beginnen wir bei der 2. loszulaufen und laufen bis zur letzten Spalte.

11.2 if()-Loop

Wie wir sehen, läuft der `for()`-Loop stur über alle Elemente hinweg, die im Index angegeben sind.

In R wird der `if()`-Loop verwendet, um bedingte Anweisungen auszuführen. Mit dem `if()`-Loop können Programmierer:innen festlegen, ob ein Codeblock ausgeführt wird oder nicht, basierend auf einer angegebenen Bedingung. Die Bedingung wird in runden Klammern angegeben und kann eine logische Aussage oder ein Vergleich sein. Wenn die Bedingung als wahr ausgewertet wird, wird der Codeblock innerhalb des `if()`-Loops (geschwungene Klammern) ausgeführt. Andernfalls wird der Code übersprungen. Bei Bedarf kann der `if()`-Loop mit zusätzlichen `else if()`- oder `else`-Statements erweitert werden, um verschiedene Bedingungen abzudecken.

Nehmen wir einmal an, wir haben in unserem Datensatz zusätzlich die Variable `Geschlecht`.

```
df = data.frame(
  ID = 1:5,
  Geschlecht = c("männlich", "weiblich", "männlich", "weiblich", "weiblich"),
  Stimmung = c(2, 4, 5, 1, 3),
  Antriebslosigkeit = c(3, 1, 2, 1, 4),
  Interessensverlust = c(1, 3, 5, 5, 5),
  Schlafprobleme = c(3, 1, 5, 2, 2),
  Suizidalität = c(2, 1, 5, 4, 3)
)

df
##   ID Geschlecht Stimmung Antriebslosigkeit Interessensverlust Schlafprobleme
## 1  1 männlich      2                  3                 1                  3
## 2  2 weiblich      4                  1                 3                  1
## 3  3 männlich      5                  2                 5                  5
## 4  4 weiblich      1                  1                 5                  2
## 5  5 weiblich      3                  4                 5                  2
##   Suizidalität
## 1              2
## 2              1
## 3              5
## 4              4
## 5              3
```

Wir wollen uns nun durch unseren `for()`-Loop nur die Stimmungswerte der Frauen anzeigen lassen:

```
for (i in 1:nrow(df)) {

  if(df$Geschlecht[i] == "weiblich"){

    print(df$Stimmung[i])
  }
}
```

```
Sys.sleep(time = 1)
}

## [1] 4
## [1] 1
## [1] 3
```

Wenn die Bedingung nicht erfüllt ist (in diesem Fall Geschlecht == “männlich”), können wir mittels `else{}` eine alternative Verhaltensweise der Schleife definieren.

Lassen Sie uns dem Nutzer eine kleine Nachricht schreiben, die erscheinen soll, wenn die Bedingung nicht zutrifft.

```
for (i in 1:nrow(df)) {

  if(df$Geschlecht[i] == "weiblich"){

    print(df$Stimmung[i])

  }else{
    print("Die Person ist ein Mann.")
  }
  Sys.sleep(time = 1)
}

## [1] "Die Person ist ein Mann."
## [1] 4
## [1] "Die Person ist ein Mann."
## [1] 1
## [1] 3
```

Chapter 12

Graphiken

Die Datenvisualisierung ist einer der wichtigsten Schritte in der Statistik. Sie ist nicht nur Teil der Datenanalyse, eine gute Darstellung komplizierter Ergebnisse kann beinahe als Kunst betrachtet werden. Die Programmiersprache R stellt uns ein leistungsstarkes Visualisierungspaket zur Verfügung, `ggplot2`.

Dieses Kapitel soll zeigen, wie man mit `ggplot2` bekannte statistische Diagramme erstellen kann und wie man sie verbessern oder anpassen kann.

12.1 Das `ggplot2` Paket

`ggplot2` ist ein Plot-System für R, das auf der “Grammatik” von Grafiken basiert. Es kümmert sich um viele der kniffligen Details, die das Plotten mühsam machen (wie z.B. das Zeichnen von Legenden) und bietet ein leistungsfähiges Grafikmodell, das es einfach macht, komplexe mehrschichtige Grafiken zu erstellen.

Warum ist `ggplot2` gut?

- Mit einem einzigen Befehl lassen sich hervorragende Themen erstellen.
- Die Farben sind schöner und ansprechender als bei den üblichen Grafiken (und frei wählbar).
- Es ist einfach, Daten mit mehreren Variablen zu visualisieren.
- Bietet eine Plattform zur Erstellung einfacher Diagramme, die eine Fülle von Informationen liefern.

12.2 ggplot()-Funktion

Die von `ggplot2` implizierte “Grammatik der Grafik”, beruht auf dem Prinzip, dass ein Plot in die folgenden grundlegenden Teile aufgeteilt werden kann:

Plot = data + aesthetics + geometry

- **data** bezieht sich auf einen Datensatz (`data.frame`).
- **aesthetics** bezeichnet die x- und y-Variablen. Sie wird auch verwendet, um R mitzuteilen, wie die Daten in einem Diagramm angezeigt werden, z. B. Farbe, Größe und Form der Punkte usw.
- **geometrie** bezieht sich auf die Art der Grafik (Balkendiagramm, Histogramm, Boxplot, Liniendiagramm, Density-Plot, Streu-/Punktdiagramm usw.)

Wir werden zum Üben den Datensatz `diamonds` aus dem `tidyverse` Paket nutzen. Er behandelt den Preis und die Charakteristika von Edelsteinen, also ein schönes Thema für schöne Graphen. Bevor wir also loslegen, installieren Sie die Pakete `ggplot2` und `tidyverse` und führen Sie aus:

```
library(ggplot2)
library(tidyverse)
```

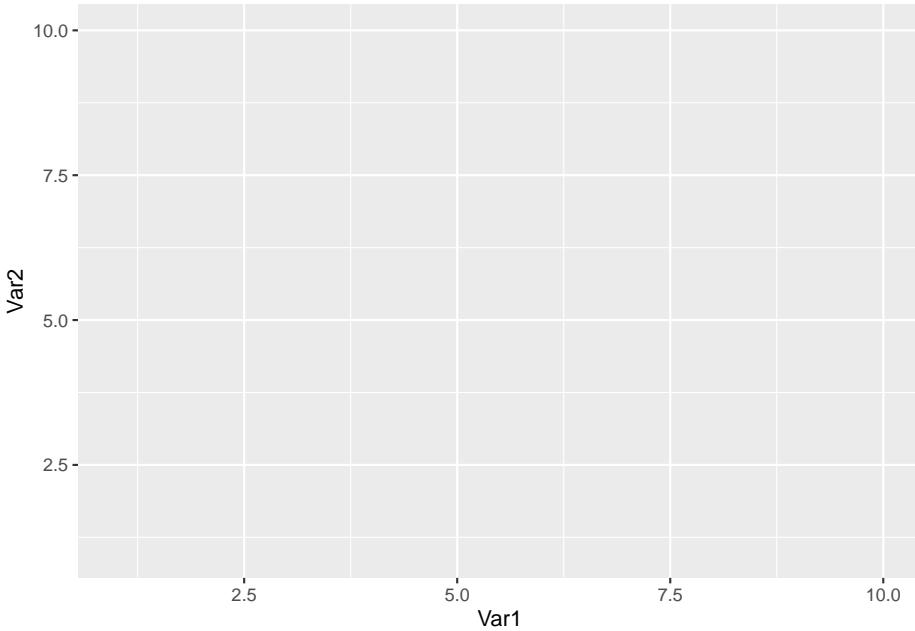
Der wichtigste (und grundlegende) Befehl zum Erstellen von Graphiken mit `ggplot2` ist `ggplot()`.

Mit `ggplot()` sagen wir R, dass jetzt eine Graphik erstellt wird. Wir können es mit dem Aufstellen einer weißen Leinwand vergleichen, die wir nun füllen wollen.

Der Befehl `ggplot()` wird mit den ersten 2 Elementen der “Grammatik der Grafik” gefüllt, `data` und `aesthetics`, kurz `aes()`. Wir geben also an, wo unsere Daten herkommen und welche Variable auf die X-Achse und welche auf die Y-Achse gelegt werden soll. Dieses Vorgehen ist für jede Art von Graphen gleich.

```
df = data.frame(Var1 = 1:10,
                Var2 = 1:10)

ggplot(data = df, aes(x = Var1, y = Var2))
```



Das Ergebnis ist ein leerer Graph, mit der Variable **Var1** auf der X-Achse und der Variable **Var2** auf der Y-Achse.

12.3 Streu-/Punktdiagramm

Streudiagramme können Ihnen helfen, die Beziehung zwischen zwei Variablen zu erkennen. Ein Streudiagramm ist eine einfache Darstellung der Kovariation einer Variablen mit einer zweiten Variable.

Besonders gut funktioniert dies, wenn man 2 numerische Variablen gegeneinander plotted.

Im Datensatz `diamonds` gibt es beispielsweise die Variable `price` (Kosten des Steins) und die Karatzahl (Masse).

```
head(diamonds)

## # A tibble: 6 x 10
##   carat cut      color clarity depth table price     x     y     z
##   <dbl> <ord>    <ord> <ord>   <dbl> <dbl> <int> <dbl> <dbl> <dbl>
## 1  0.23 Ideal    E     SI2      61.5    55   326  3.95  3.98  2.43
## 2  0.21 Premium  E     SI1      59.8    61   326  3.89  3.84  2.31
## 3  0.23 Good     E     VS1      56.9    65   327  4.05  4.07  2.31
## 4  0.29 Premium  I     VS2      62.4    58   334  4.2    4.23  2.63
## 5  0.31 Good     J     SI2      63.3    58   335  4.34  4.35  2.75
## 6  0.24 Very Good J     VVS2     62.8    57   336  3.94  3.96  2.48
```

12.3.1 Streudiagramm mit 2 Variablen

Stellen wir zunächst einmal unsere Grundfunktion auf:

```
ggplot(data = diamonds, aes(x = carat, y = price))
```

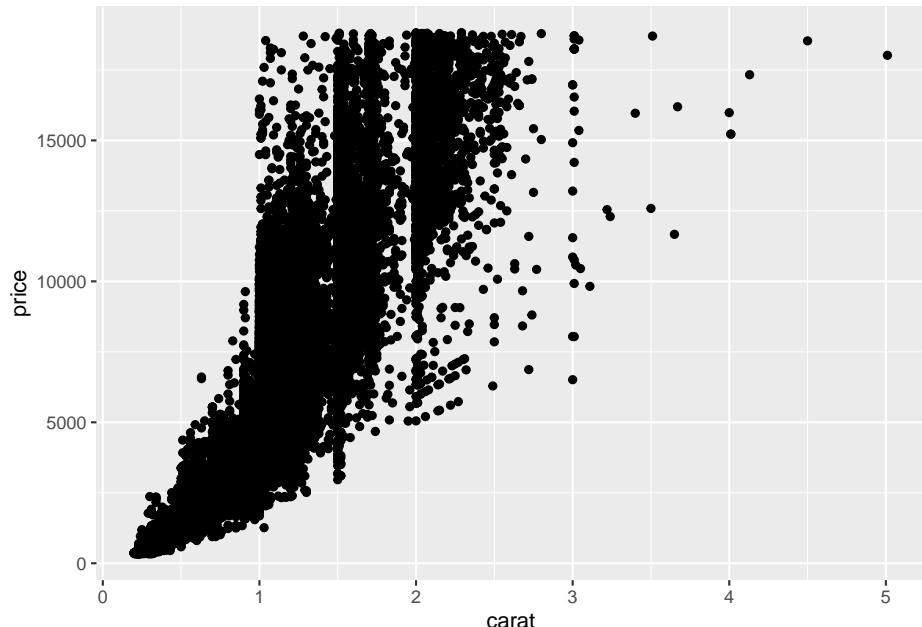
Das Ergebnis ist ein leerer Graph, mit der Variable `carat` auf der X-Achse und der Variable `price` auf der Y-Achse. Wir könnten also betrachten, ob der Preis von Diamanten steigt, wenn die Karat (Masse) höher sind.

12.3.2 geom_point()

Nun fehlen uns nun noch die charakteristischen “Punkte”, die den Graph zum Streudiagramm machen. Jeder Punkt repräsentiert einen Stein mit einer Karatzahl und einem Preis.

Wir wollen also ein neues Element in unseren Graphen zeichnen, sozusagen eine weitere Schicht auftragen. Dies funktioniert in `ggplot2` mit dem `+` Operator. Die Punkte sind eine Geometrie (das 3. Element unserer Grammatik). Geometrie wird i.d.R. mit `geom` geschrieben:

```
ggplot(data = diamonds, aes(x = carat, y = price)) +
  geom_point()
```

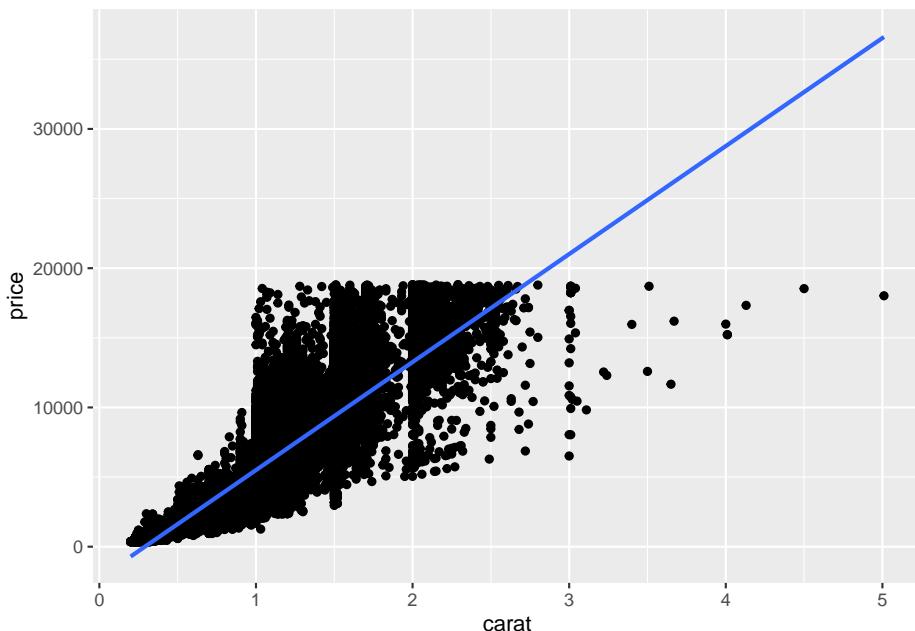


Es scheint, als bestünde ein **positiver** Zusammenhang zwischen `carat` und `price`. Mit zunehmender Karatzahl werden Steine also tendenziell teurer.

12.3.3 geom_smooth()

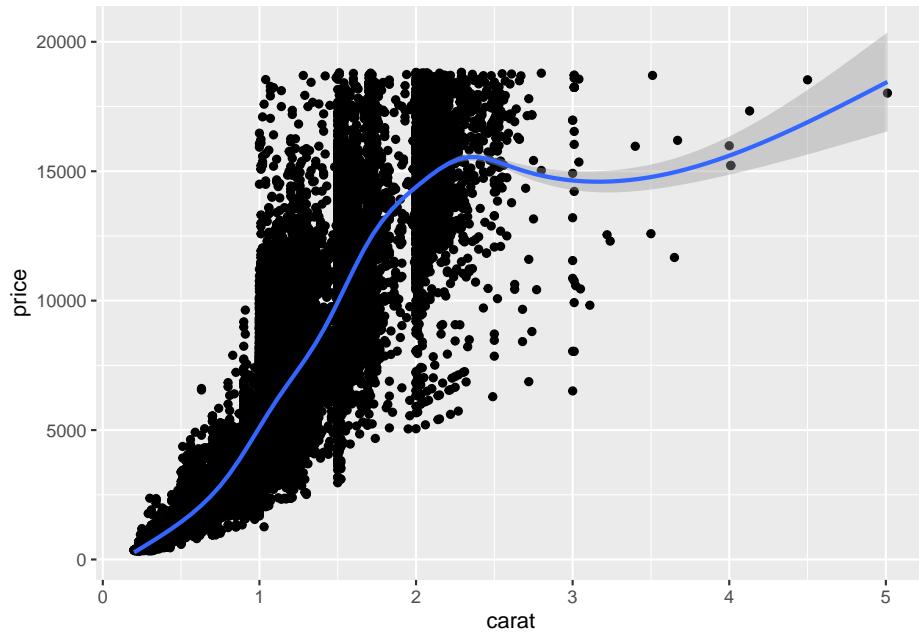
Manchmal ist es praktisch in das “Chaos” der Punkte des Streudiagramms etwas Ordnung zu bringen und sich die Zusammenhänge noch einmal mit einer linearen Funktion zu visualisieren (Regressionsgerade). Dies lässt sich mit der Funktion `geom_smooth()` machen. Wir hängen sie also einfach mit einem weiteren `+` an unseren Graphen dran:

```
ggplot(data = diamonds, aes(x = carat, y = price)) +
  geom_point() +
  geom_smooth(method = "lm")
```



Die Angabe `method = "lm"` macht die Linie zur Geraden, geben wir dieses Argument nicht an, bekommen wir eine sogenannte loess-Kurve, die jedoch schnell unübersichtlich wird:

```
ggplot(data = diamonds, aes(x = carat, y = price)) +
  geom_point() +
  geom_smooth()
```



12.3.4 Streudiagramm mit mehr als 2 Variablen

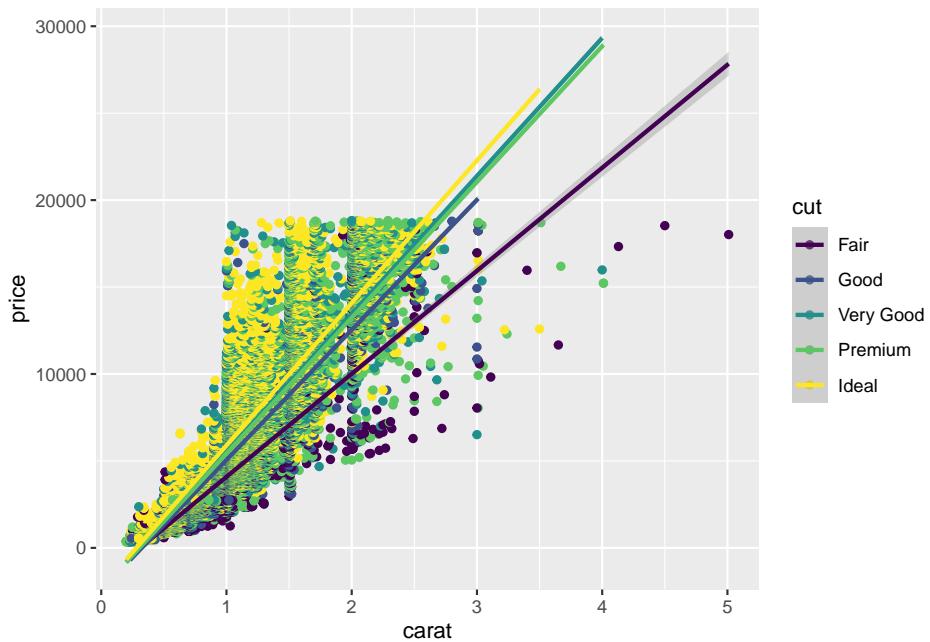
Ein weiteres wichtiges Merkmal eines Diamanten ist sein Schliff, im diamonds Datensatz heißt diese Variable `cut`. Sie wird in mehreren Kategorien gemessen:

```
table(diamonds$cut)
```

```
##          Fair       Good Very Good Premium    Ideal
##      1610      4906     12082    13791   21551
```

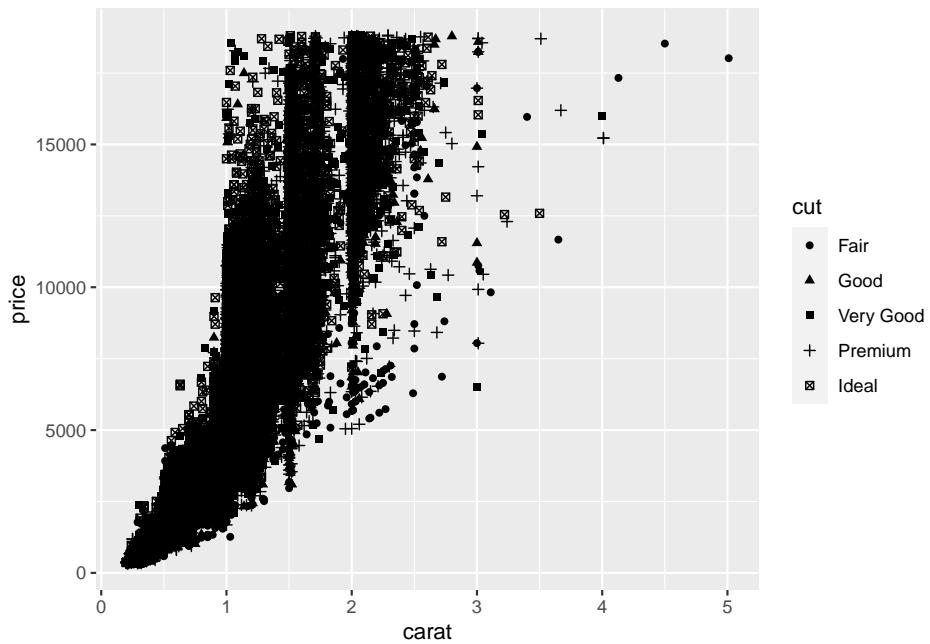
Wollen wir diese 3. Variable auch in unserem Graphen darstellen, haben wir das Problem, dass wir neben X und Y keine weitere Achse haben. Oft behilft man sich in solchen Situationen durch die Hinzunahme einer weiteren **aesthetics** Kategorie, z.B. **Farben** (`colour`). Jede Schliffkategorie erhält also ihre eigene Farbe:

```
ggplot(data = diamonds, aes(x = carat, y = price, colour = cut)) +
  geom_point() +
  geom_smooth(method = "lm")
```



Es wären auch andere **aesthetics** zur Unterscheidung denkbar, z.B. die Form der Punkte (**shape**):

```
ggplot(data = diamonds, aes(x = carat, y = price, shape = cut)) +
  geom_point()
```



12.4 Histogramm

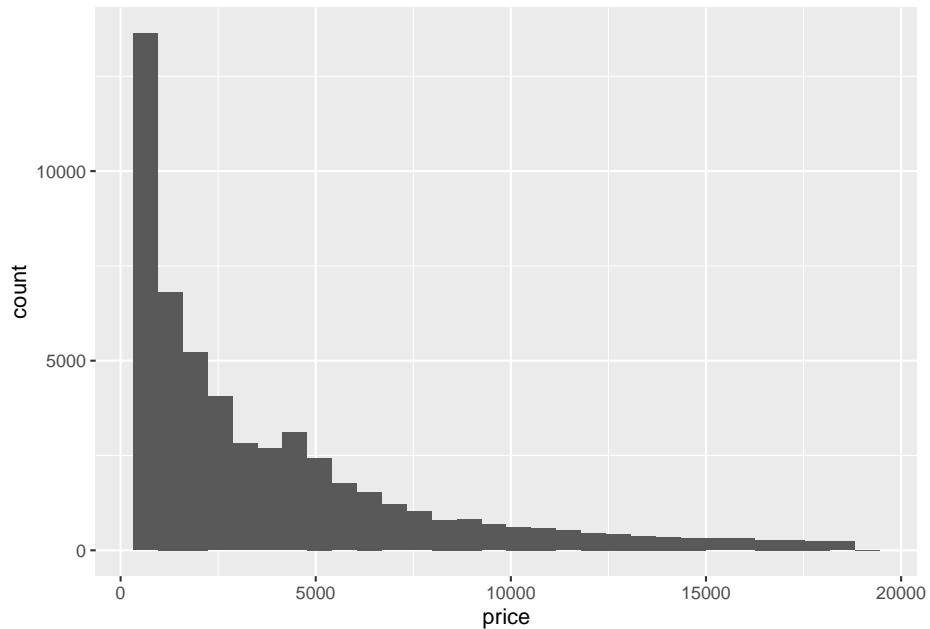
Das Histogramm ist der wichtigste Graph zur Darstellung der Verteilung einer Variable. Er ist hinsichtlich der Komplexität sogar noch etwas leichter als das Streudiagramm, da nur 1 Variable enthalten ist.

Die darzustellende Variable liegt i.d.R. auf der X-Achse, während auf der Y-Achse die absolute oder relative Häufigkeit der Merkmalsausprägungen dargestellt wird.

Lassen Sie uns anhand der Variable `price` einmal ausprobieren ein Histogramm zu erstellen:

```
ggplot(data = diamonds, aes(x = price)) +
  geom_histogram()

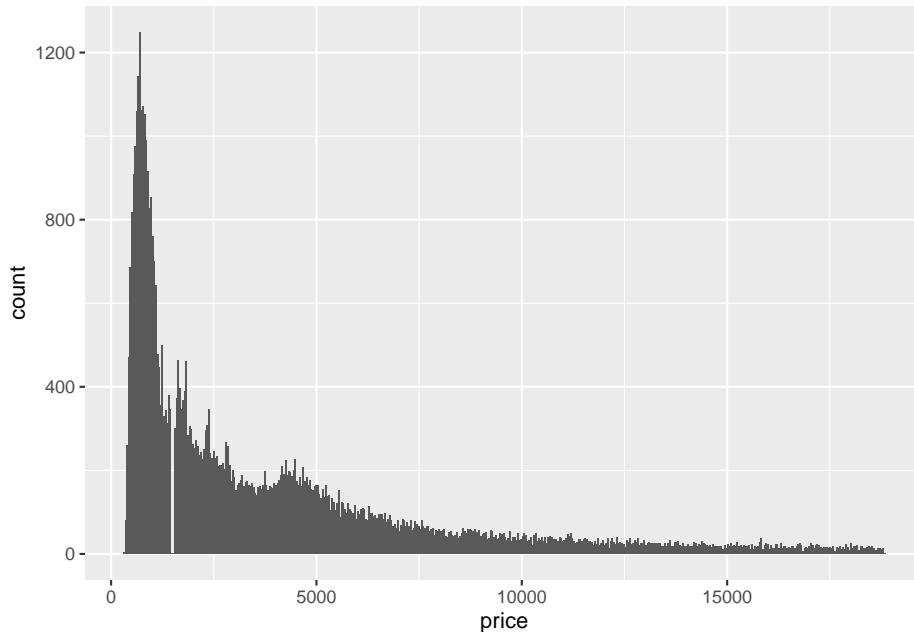
## `stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
```



Wie wir sehen, kommen günstigere Diamanten häufiger vor als teure. In diesem Fall haben wir also nicht die beliebte **normalverteilte** Form der Verteilung, sondern eine rechtsschiefe Form.

R gibt uns die Warnmeldung "Pick better value with `binwidth`". Dies ist ein Hinweis, dass die Standardanzahl von 30 Balken für die hohe Auflösung der Variable `price` nicht ausreicht. Es wird uns eine Auflösung von `binwidth = 39` vorgeschlagen:

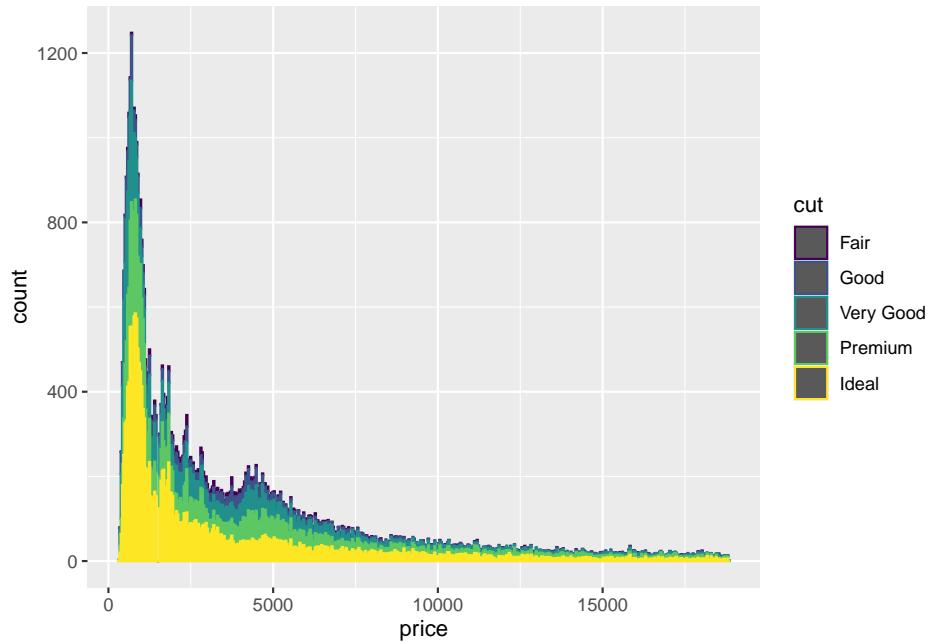
```
ggplot(data = diamonds, aes(x = price)) +  
  geom_histogram(binwidth = 39)
```



12.4.1 Histogramm für mehrere Gruppen

Natürlich lassen sich Histogramme, also die Verteilungen einer Variable auch getrennt für unterschiedliche Kategorien darstellen:

```
ggplot(data = diamonds, aes(x = price, colour = cut)) +  
  geom_histogram(binwidth = 39)
```



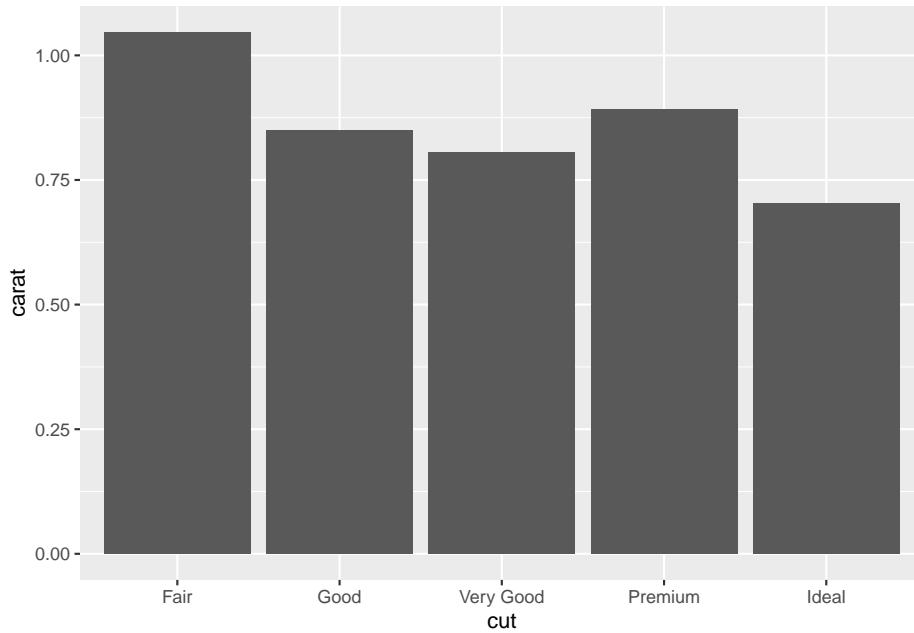
12.5 Balkendiagramm

Das Balkendiagramm ist eine Möglichkeit, die Höhe einer Deskriptivstatistik (aka summary statistics) mit der Höhe eines Balkens zu visualisieren. Es wird z.B. eingesetzt, um Gruppenunterschiede, bzw. Unterschiede zwischen den Kategorien einer Variable darzustellen.

Probieren wir es doch gleich einmal mit der Variable `cut` (Schliff eines Diamanten) aus, die wir oben bereits kennengelernt haben. Wir wollen die durchschnittlichen Karat (Gewicht) der Steine für jede Kategorie von `cut` darstellen. Die Höhe der Balken muss also jeweils den Mittelwert repräsentieren.

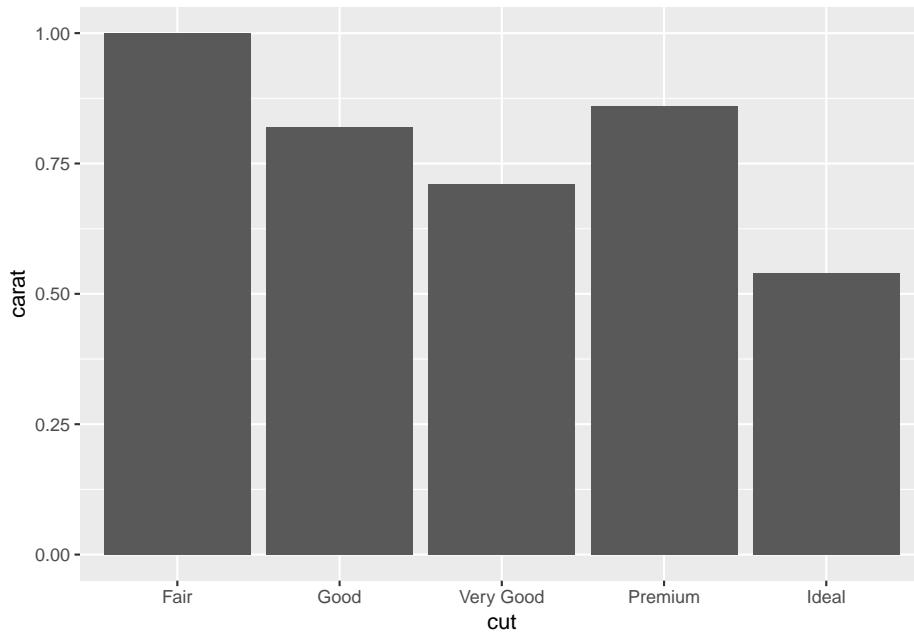
Da wir nun keinen Wert visualisieren wollen der "direkt" als Zahl im Datensatz steht, sondern für die Berechnung des Mittelwerts eigentlich ein weiterer Rechenschritt erfolgen muss, benutzen wir die `stat_summary` Funktion:

```
ggplot(data = diamonds, aes(x = cut, y = carat)) +
  stat_summary(geom = "bar", fun = mean)
```



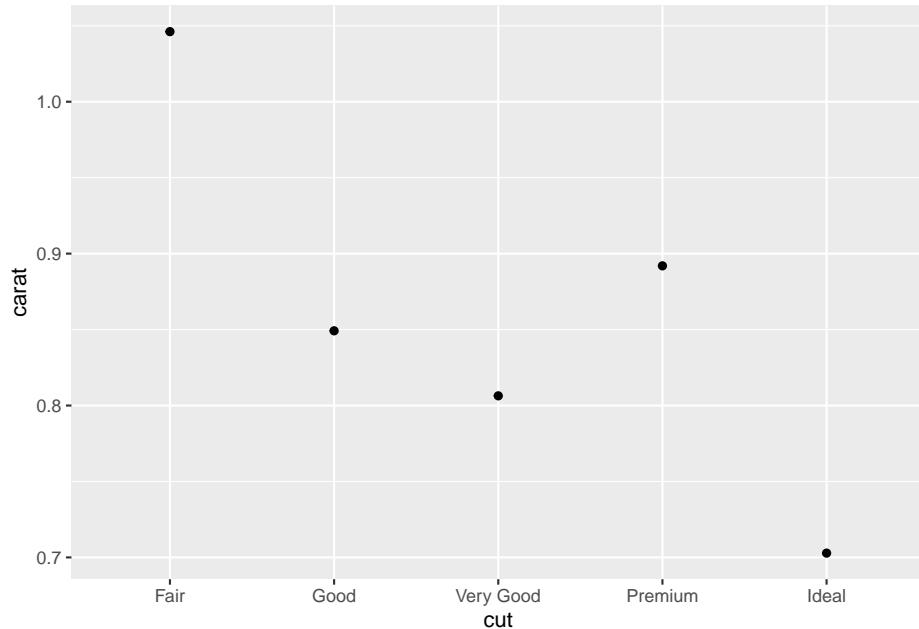
Selbiges funktioniert natürlich auch mit jeder anderen Deskriptivstatistik, z.B. dem Median

```
ggplot(data = diamonds, aes(x = cut, y = carat)) +  
  stat_summary(geom = "bar", fun = median)
```



Oder theoretisch auch mit anderen geoms

```
ggplot(data = diamonds, aes(x = cut, y = carat)) +
  stat_summary(geom = "point", fun = mean)
```

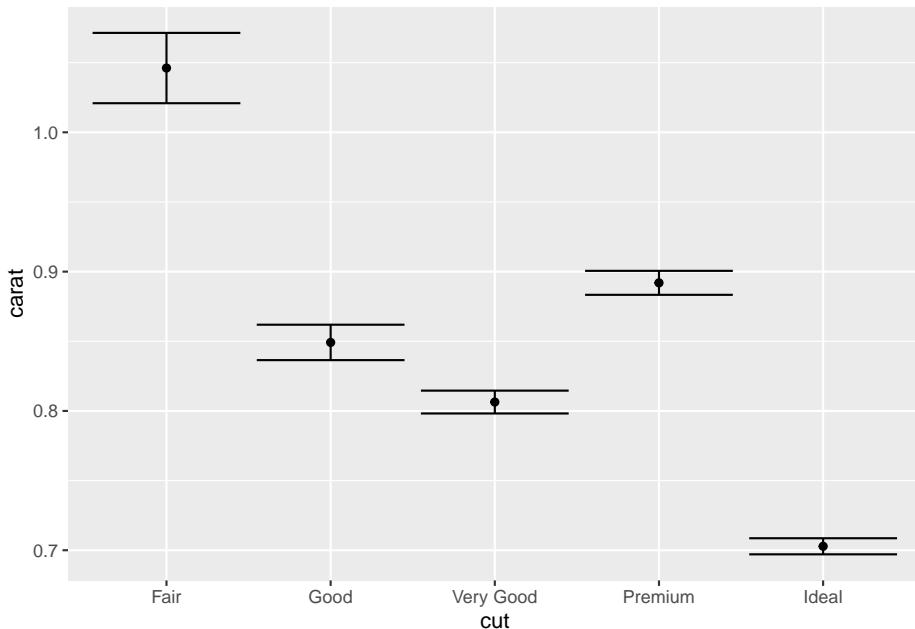


Häufig möchte man zusätzlich eine Art von Streuung/Unschärfe darstellen, wenn Mittelwerte dargestellt werden. Eine beliebte Variante ist es, das 95% Konfidenzintervall um den Mittelwert mit sogenannten **Fehlerbalken** (error bars) zu visualisieren. Vorher installieren und laden wir noch das Paket `Hmisc`, welches uns die Berechnung der Fehlerbalken ermöglicht:

```
library(Hmisc)
```

```
## Warning: package 'Hmisc' was built under R version 3.6.2
## Loading required package: lattice
## Warning: package 'lattice' was built under R version 3.6.2
## Loading required package: survival
## Warning: package 'survival' was built under R version 3.6.2
## Loading required package: Formula
## Warning: package 'Formula' was built under R version 3.6.2
##
## Attaching package: 'Hmisc'
## The following object is masked from 'package:psych':
```

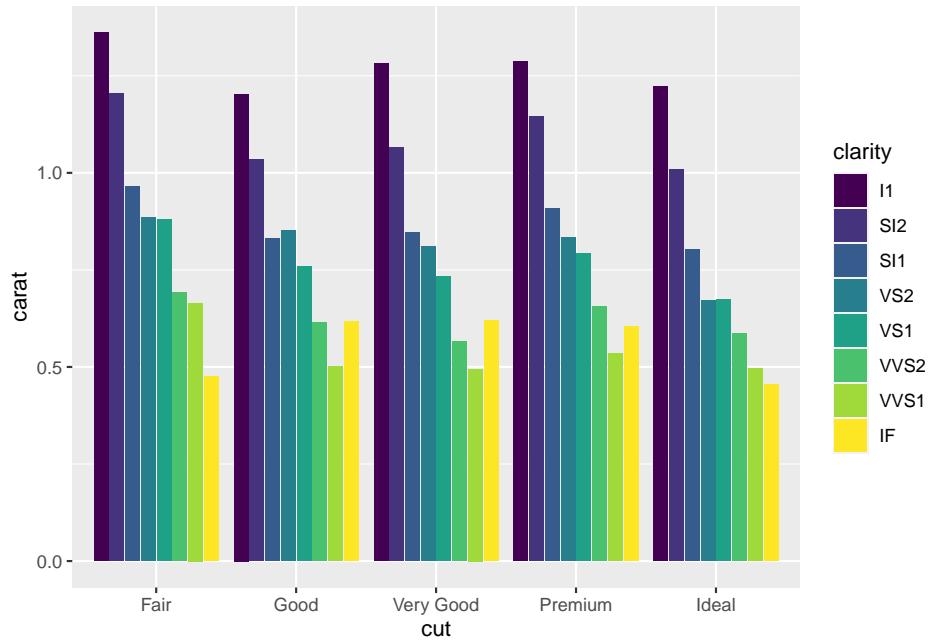
```
##  
##     describe  
  
## The following objects are masked from 'package:dplyr':  
##  
##     src, summarize  
  
## The following objects are masked from 'package:base':  
##  
##     format.pval, units  
ggplot(data = diamonds, aes(x = cut, y = carat)) +  
  stat_summary(fun.data = mean_cl_normal, geom = "errorbar") +  
  stat_summary(geom = "point", fun = mean)
```



12.6 Balkendiagramm mit mehr als 3 Variablen

Genau wie beim Streudiagramm lässt sich das Balkendiagramm auch für mehr als 2 Variablen darstellen. Statt Farben oder Punktfarben ist die einfachste Art Balken zu differenzieren mit der fill Ästhetik:

```
ggplot(data = diamonds, aes(x = cut, y = carat, fill = clarity)) +  
  stat_summary(geom = "bar", fun = mean, position = position_dodge2(.95))
```

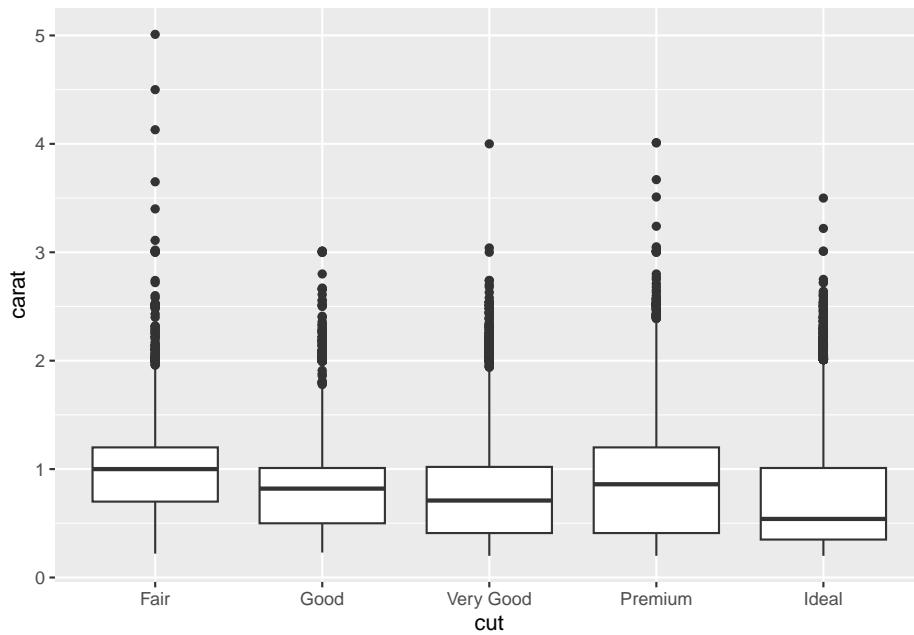


Das Argument `position_dodge2()` sorgt dafür, dass die Balken nebeneinander positioniert sind und nicht voreinander. Probieren Sie den Code gerne auch einmal ohne `position_dodge2()` aus.

12.7 Boxplot

Ähnlich wie das Balkendiagramm, lassen sich Gruppenunterschiede gut mit einem Boxplot darstellen. Dieses zeigt standardmäßig den Median (Mittelbalken), sowie die Streuung der Daten mittels der Box (Quartilabstand, IQR) und die sogenannten Whiskers ($1.5 * \text{Quartilabstand}$). Punkte außerhalb der Whiskers werden als Ausreißer mit einem Punkt gekennzeichnet.

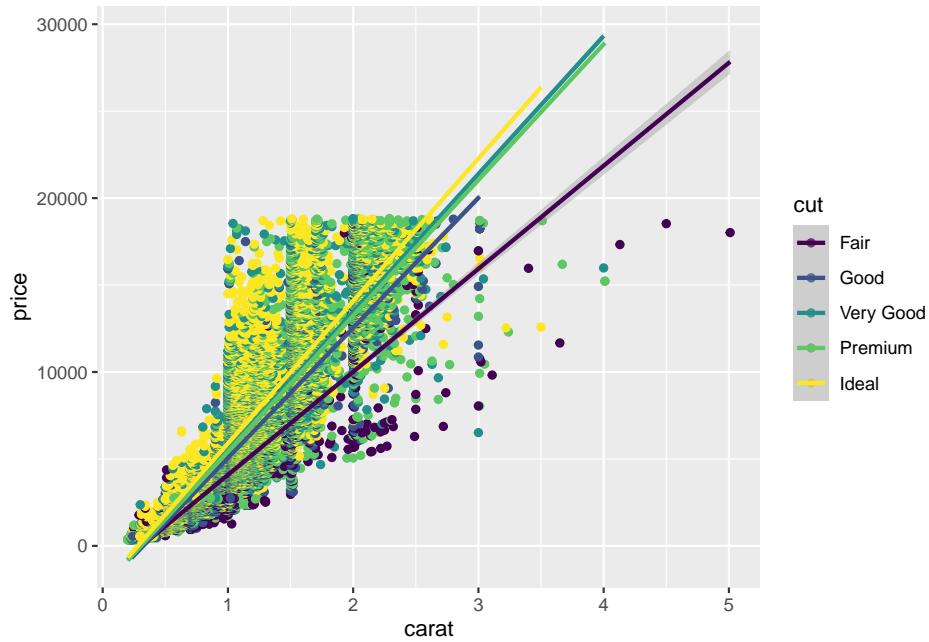
```
ggplot(data = diamonds, aes(x = cut, y = carat)) +
  geom_boxplot()
```



12.8 Facetting

Wir sehen uns noch einmal unser 3-farbiges Streudiagramm von zuvor an:

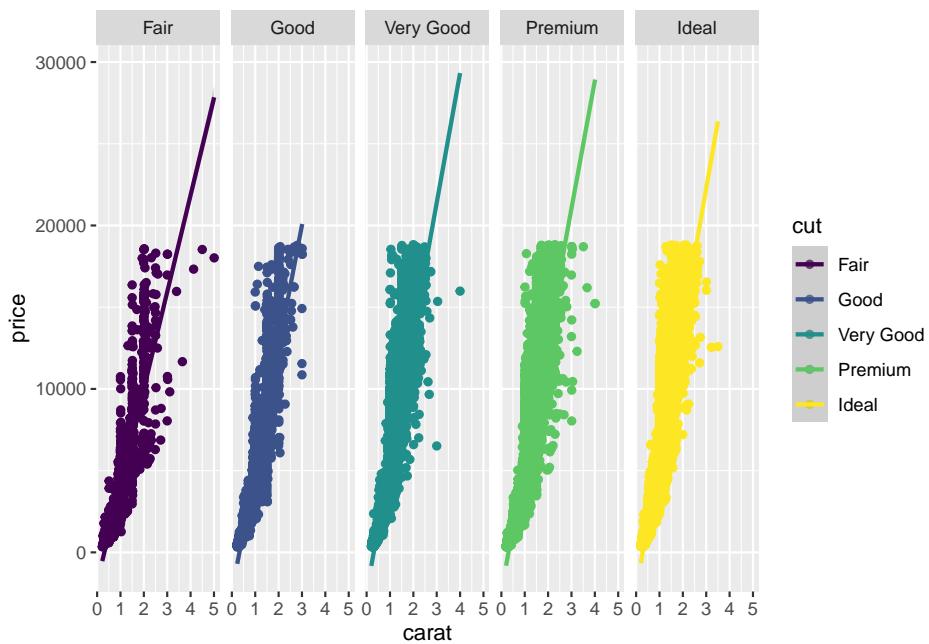
```
ggplot(data = diamonds, aes(x = carat, y = price, colour = cut)) +  
  geom_point() +  
  geom_smooth(method = "lm")
```



Es ist sichtbar, dass die Darstellung vieler Kategorien mittels weiterer aesthetics, wie Farben, Füllungen, Punktarten, etc. schnell unübersichtlich wird.

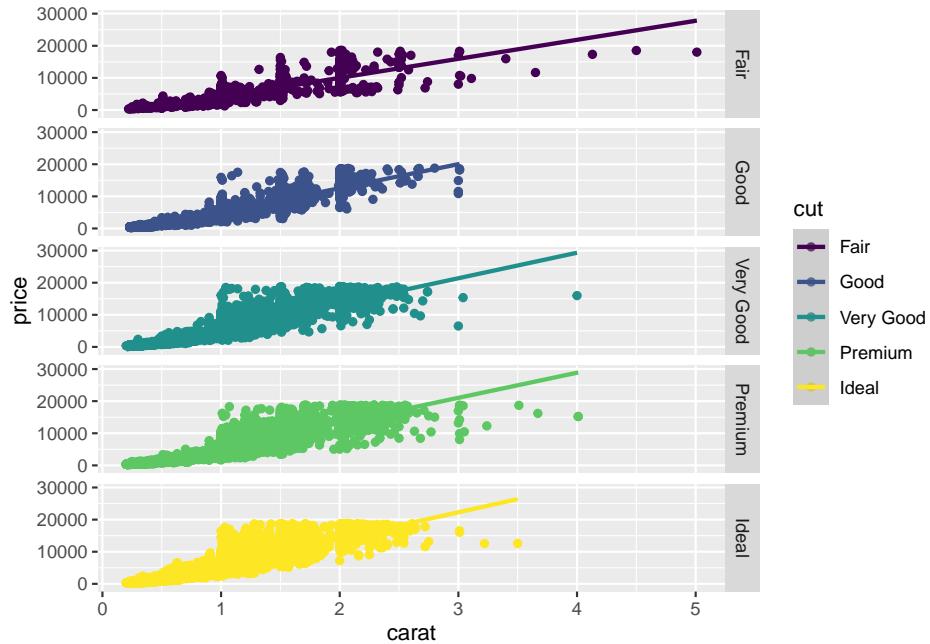
Oft ist es nützlich, durch die Darstellung einzelner Gruppen in Teilgraphen (Facetten) etwas mehr Klarheit in die Darstellung zu bringen. Der Befehl dafür ist `facet_grid()`:

```
ggplot(data = diamonds, aes(x = carat, y = price, colour = cut)) +
  geom_point() +
  geom_smooth(method = "lm") +
  facet_grid(cols = vars(cut))
```



Es ist auch eine reihenweise Darstellung möglich

```
ggplot(data = diamonds, aes(x = carat, y = price, colour = cut)) +
  geom_point() +
  geom_smooth(method = "lm") +
  facet_grid(rows = vars(cut))
```



12.9 Ästhetische Anpassungen

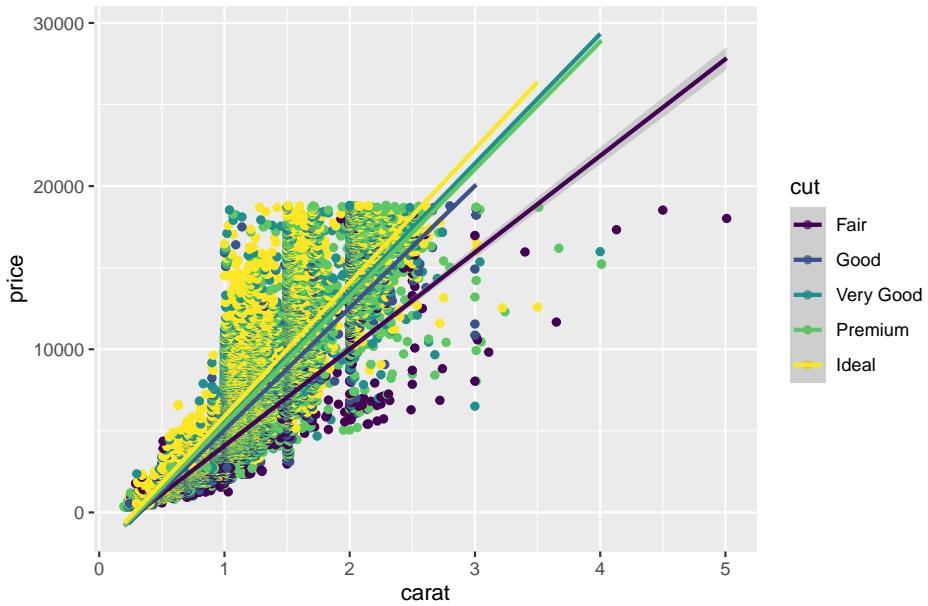
Zum Gewinnen eines schnellen Überblicks kommen wir mit den vorgestellten Darstellungsoptionen schon recht weit. Für eine Publikation oder das Verwenden der Graphik in einer Abschlussarbeit wollen wir jedoch ggf. noch einige Dinge anpassen

12.9.1 ggtitle()

Um unserer Graphik einen Titel zu geben, nutzen wir den Befehl `ggtitle()`. Vorsicht: Wie jede Zeichenkette schreiben wir auch hier den Namen in Anführungszeichen:

```
ggplot(data = diamonds, aes(x = carat, y = price, colour = cut)) +
  geom_point() +
  geom_smooth(method = "lm") +
  ggtitle("Abb 1: Streudiagramm Diamantenpreis")
```

Abb 1: Streudiagramm Diamantenpreis



12.9.2 labs()

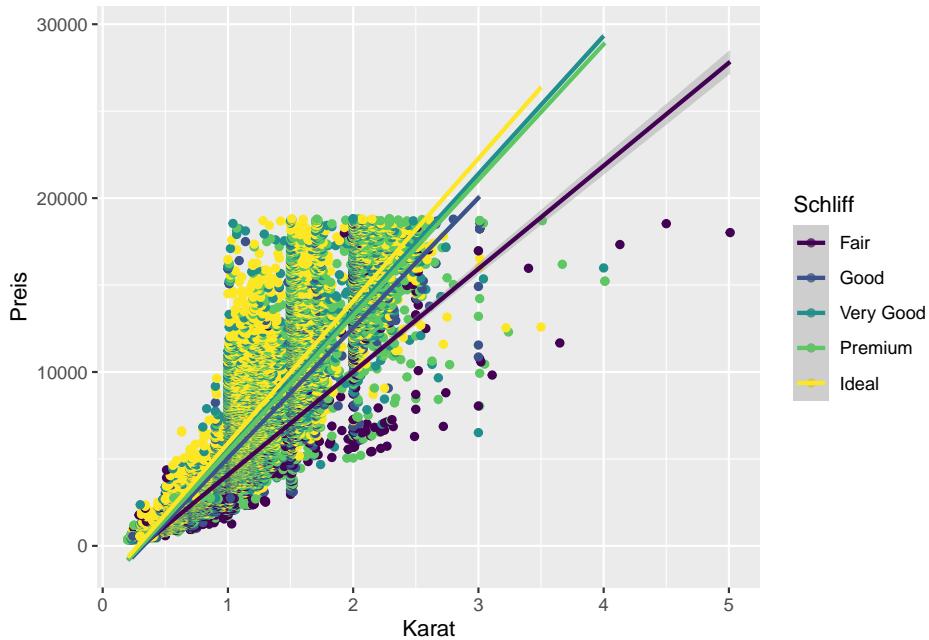
ggplot2 benutzt zur Beschriftung von X- und Y-Achse sowie als Titel der Legende automatisch die im `data.frame` enthaltenen Variablennamen

```
names(diamonds)
```

```
## [1] "carat"     "cut"       "color"      "clarity"    "depth"      "table"      "price"
## [8] "x"          "y"          "z"
```

Wollen wir dort eine schönere Beschriftung verwenden, nutzen den Befehl `labs()`. Die Y-Achse wollen wir nun auf deutsch "Preis" nennen, die X-Achse "Karat" und die Legende soll den Titel "Schliff" haben:

```
ggplot(data = diamonds, aes(x = carat, y = price, colour = cut)) +
  geom_point() +
  geom_smooth(method = "lm") +
  labs(x = "Karat", y = "Preis", colour = "Schliff")
```

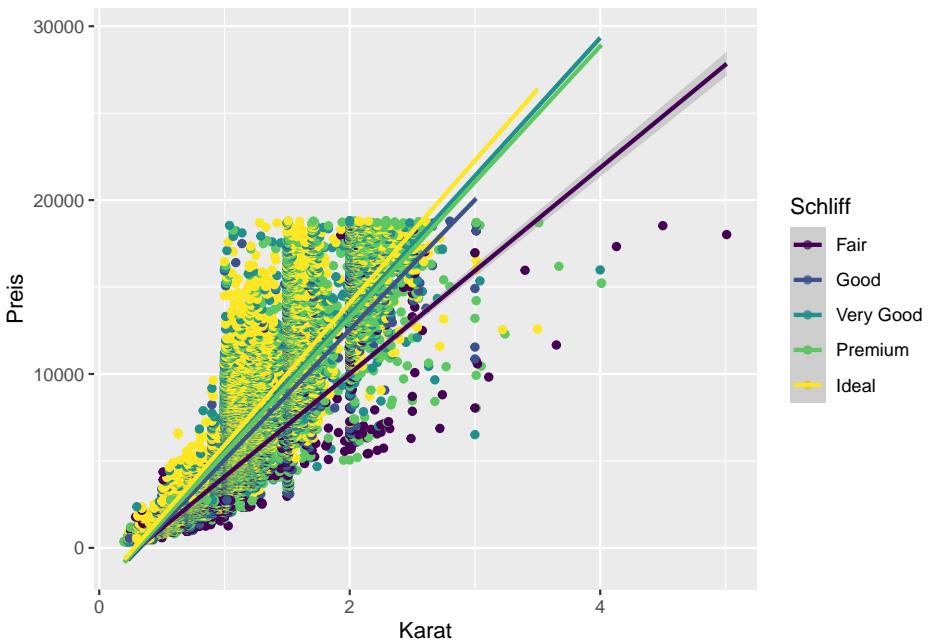


12.9.3 Achsen verändern

Auch die Einheiten, mit denen die Achsen beschriftet werden (axis ticks) werden von R automatisch und gleichmäßig gewählt. Die Beschriftung kann auch manuell gesteuert werden. Der Befehl lautet `scale_x_continuous()`, bzw. `scale_y_continuous()` für kontinuierliche (numerische) X- und Y-Achsen. Für kategoriale X- und Y-Achsen (Namen/Kategorien) lautet der Befehl `scale_x_discrete()`, bzw. `scale_y_discrete ()`.

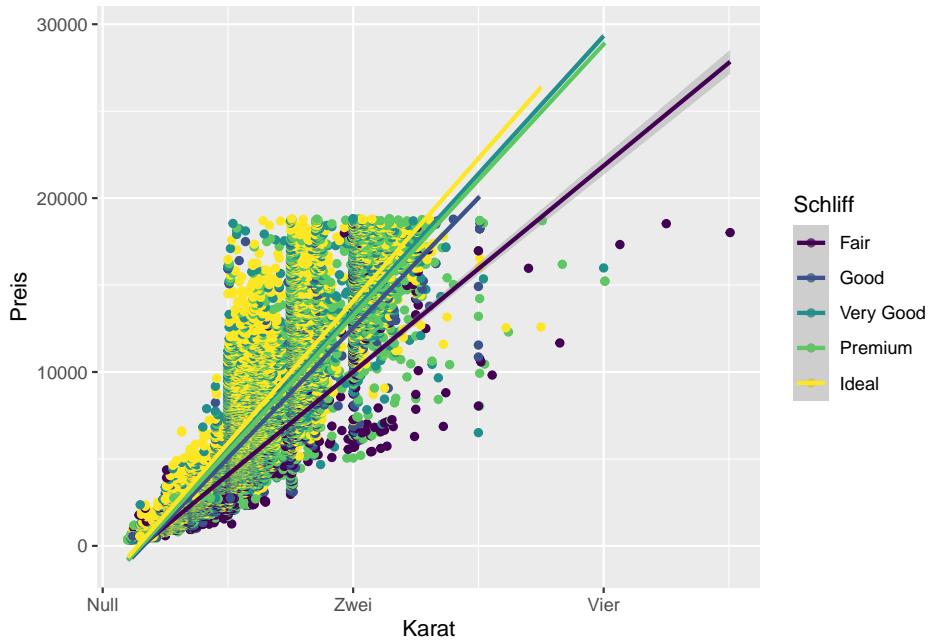
Wir wollen zum Üben in dem vorangegangenen Graphen auf der X-Achse (Karat) manuell nur die Werte 0, 2 und 4 zulassen. Dies erfolgt über `breaks = c()`:

```
ggplot(data = diamonds, aes(x = carat, y = price, colour = cut)) +
  geom_point() +
  geom_smooth(method = "lm") +
  labs(x = "Karat", y = "Preis", colour = "Schliff") +
  scale_x_continuous(breaks = c(0, 2, 4))
```



Automatisch werden an den angegebenen Positionen der kontinuierlichen X-Achse die Zahlen eingetragen, also 0, 2 und 4. Wir könnten diese aber auch mit `labels` versehen:

```
ggplot(data = diamonds, aes(x = carat, y = price, colour = cut)) +
  geom_point() +
  geom_smooth(method = "lm") +
  labs(x = "Karat", y = "Preis", colour = "Schliff") +
  scale_x_continuous(breaks = c(0, 2, 4), labels = c("Null", "Zwei", "Vier"))
```



12.10 theme()

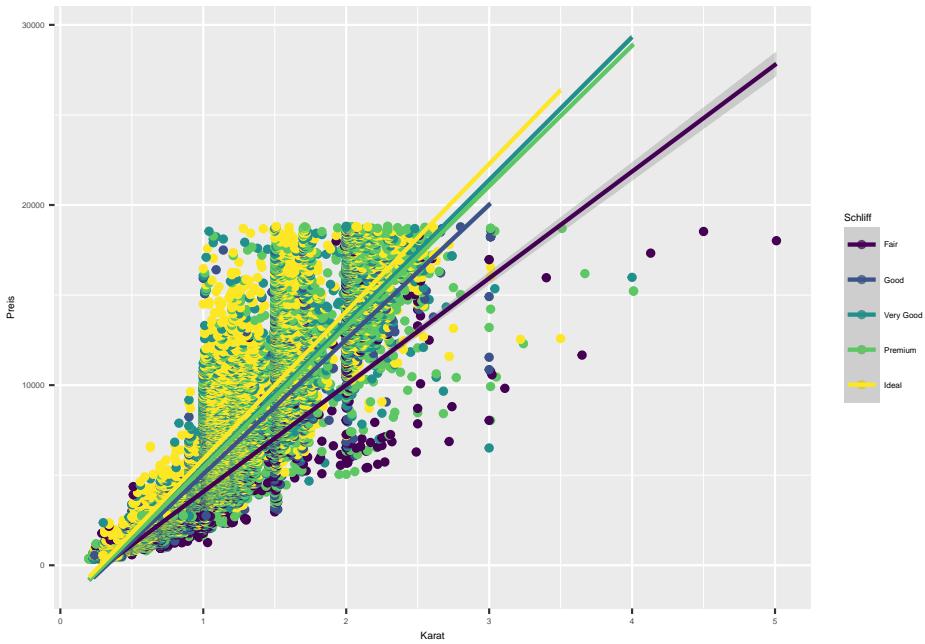
Der Befehl `theme()` ist einer der mächtigsten in `ggplot2`, denn er verändert das Aussehen des gesamten Graphen.

Er hat so viele Anpassungsmöglichkeiten, dass wir sie an dieser Stelle unmöglich auflisten könnten. Glücklicherweise ist eine Liste aller Optionen unter `?theme()` hinterlegt.

12.10.1 Elemente von `theme()`

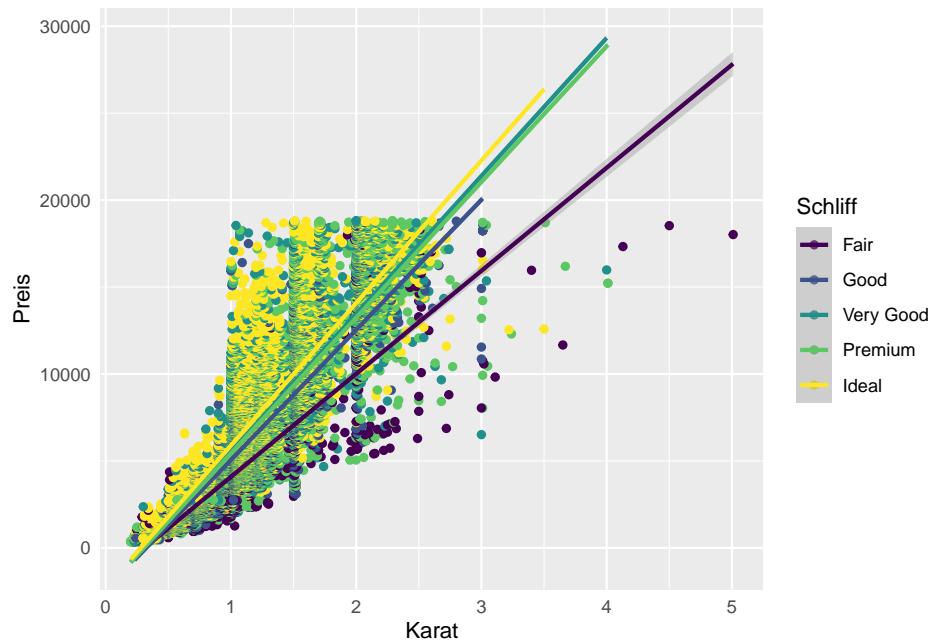
Sehen wir uns dennoch einmal ein Beispiel an. Wie wäre es, wenn wir den gesamten Text in unserem Graphen etwas kleiner machen? Das geht so:

```
ggplot(data = diamonds, aes(x = carat, y = price, colour = cut)) +
  geom_point() +
  geom_smooth(method = "lm") +
  labs(x = "Karat", y = "Preis", colour = "Schliff") +
  theme(text = element_text(size = 5))
```



Wir können auf diese Art auch Elemente (Achsen, Beschriftungen...) aus unserem Graphen löschen. Dafür wählt man die Option `element_blank()`. Lassen wir zum Beispiel einmal die kleinen Striche, die die Einheiten an den Achsen anzeigen verschwinden:

```
ggplot(data = diamonds, aes(x = carat, y = price, colour = cut)) +
  geom_point() +
  geom_smooth(method = "lm") +
  labs(x = "Karat", y = "Preis", colour = "Schliff") +
  theme(axis.ticks = element_blank())
```

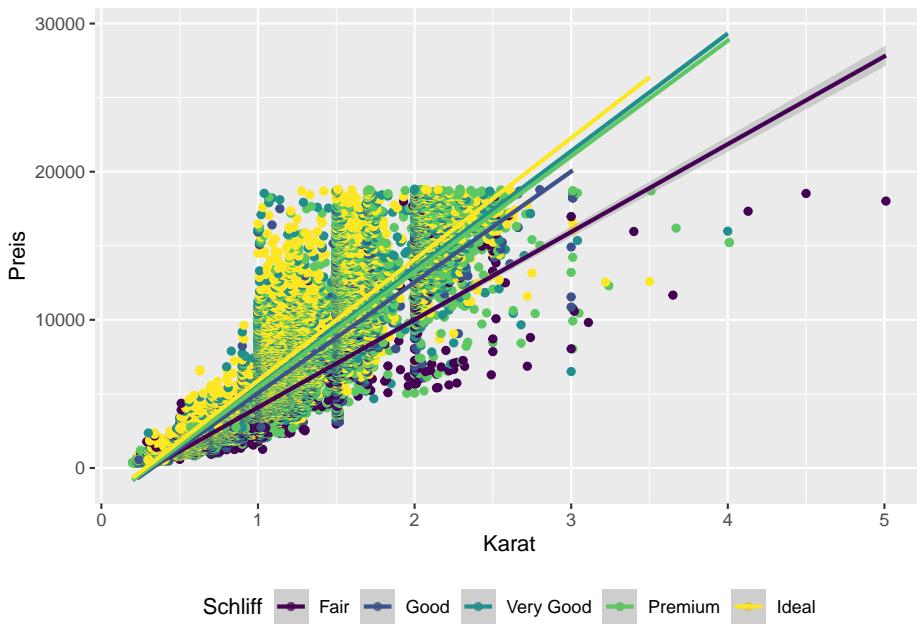


12.10.2 Legende

Auch die Legende des Graphen lässt sich über `theme()` steuern. Dazu nutzen wir das Argument `legend.position`.

Die Optionen sind “right” (das ist der Standard), “left”, “top”, “bottom”. Lassen Sie uns die Legende einmal unter den Graphen verschieben:

```
ggplot(data = diamonds, aes(x = carat, y = price, colour = cut)) +
  geom_point() +
  geom_smooth(method = "lm") +
  labs(x = "Karat", y = "Preis", colour = "Schliff") +
  theme(legend.position = "bottom")
```

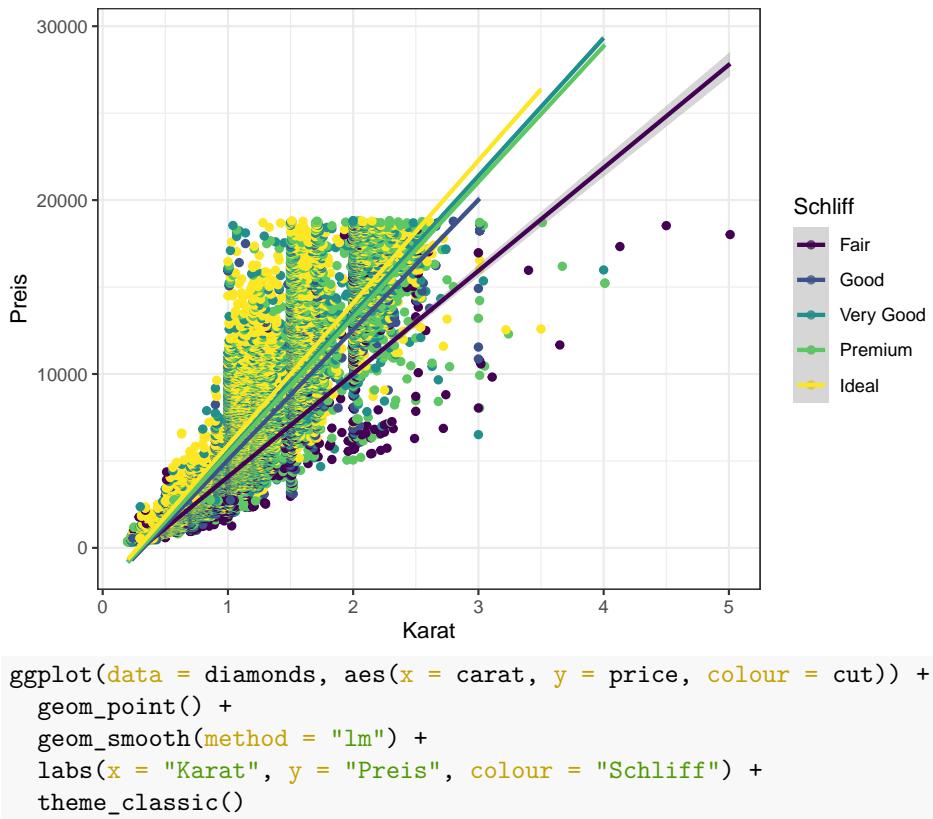


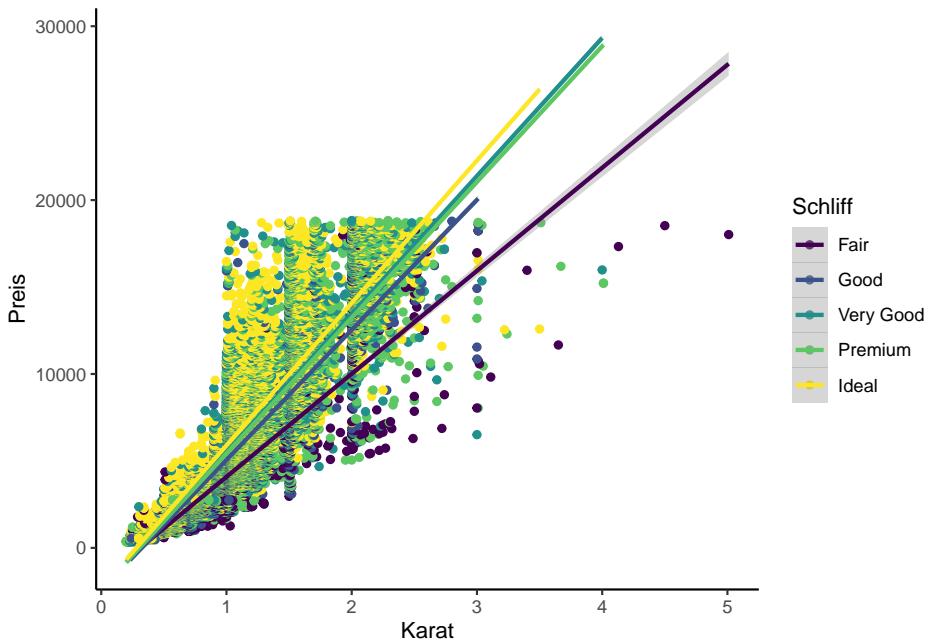
12.10.3 Vorgefertigte themes

Wenn wir nicht alle optischen Elemente des Graphen einzeln anpassen möchten, aber ihn doch etwas individueller aussehen lassen wollen, können wir eines der vorgefertigten Themen nutzen.

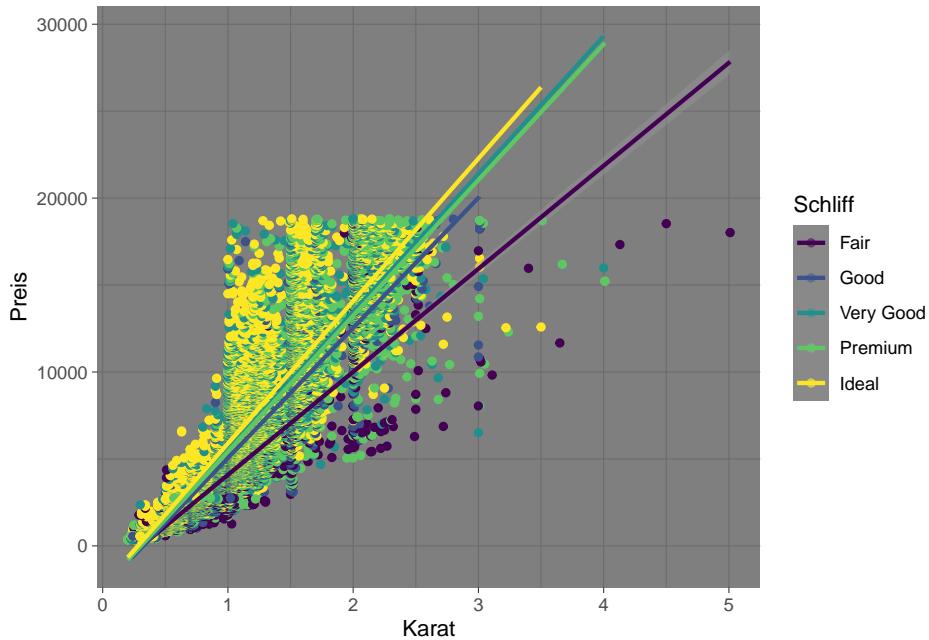
Hier einige häufig gewählte Beispiele:

```
ggplot(data = diamonds, aes(x = carat, y = price, colour = cut)) +
  geom_point() +
  geom_smooth(method = "lm") +
  labs(x = "Karat", y = "Preis", colour = "Schliff") +
  theme_bw()
```





```
ggplot(data = diamonds, aes(x = carat, y = price, colour = cut)) +  
  geom_point() +  
  geom_smooth(method = "lm") +  
  labs(x = "Karat", y = "Preis", colour = "Schliff") +  
  theme_dark()
```



Es gibt R-Pakete, die hunderte weitere Optionen bereitstellen, so zum Beispiel das Paket `ggthemes`. Installieren Sie es einfach mittels `install.packages()` und testen Sie es aus.

12.11 Abbildungen kombinieren

Gerade bei komplexeren Fragestellungen oder Studien mit mehreren abhängigen Variablen (AVs) kommt es vor, dass die Anzahl der Graphen etwas zu groß wird.

Dann kann es helfen, mehrere Graphen zu einem kombinierten Graph zusammensetzen. Die Funktion dafür heißt `ggarrange()` und kommt aus dem R-Paket `ggbpbr`.

Also installieren Sie kurz das Paket `ggbpbr`, laden es mittels `library()` und dann geht es gleich weiter:

```
# install.packages("ggbpbr")
library(ggbpbr)
```

Damit es sich auch lohnt, nehmen wir uns gleich einmal 4 Graphen vor. Nehmen wir doch einen von jeder Art. Damit es nicht zu unübersichtlich wird, benennen wir die 4 Graphen mit `a`, `b`, `c` und `d`:

```
a = ggplot(data = diamonds, aes(x = price)) +
  geom_histogram()
```

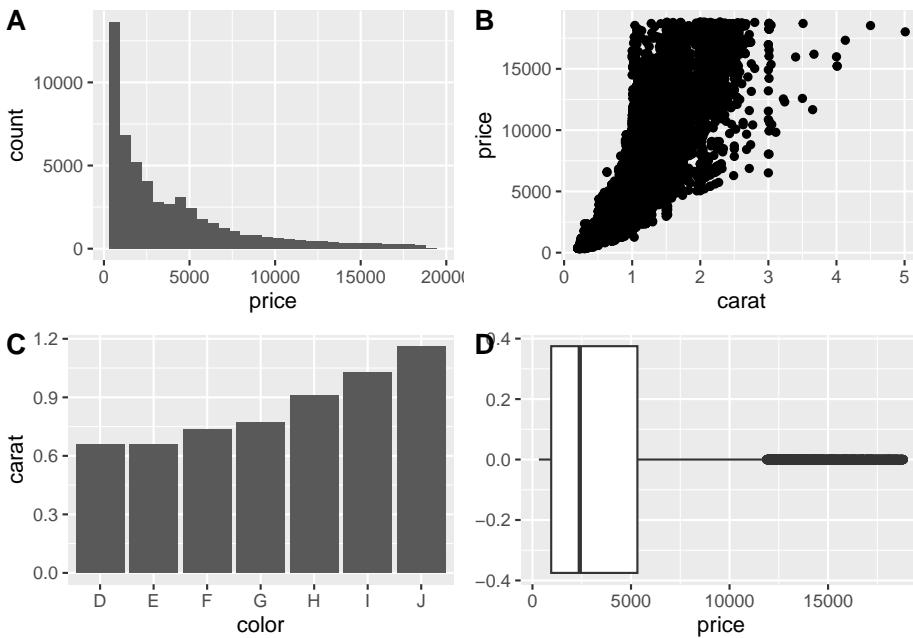
```
b = ggplot(data = diamonds, aes(x = carat, y = price)) +
  geom_point()

c = ggplot(data = diamonds, aes(x = color, y = carat)) +
  stat_summary(geom = "bar", fun = mean)

d = ggplot(data = diamonds, aes(x = price)) +
  geom_boxplot()
```

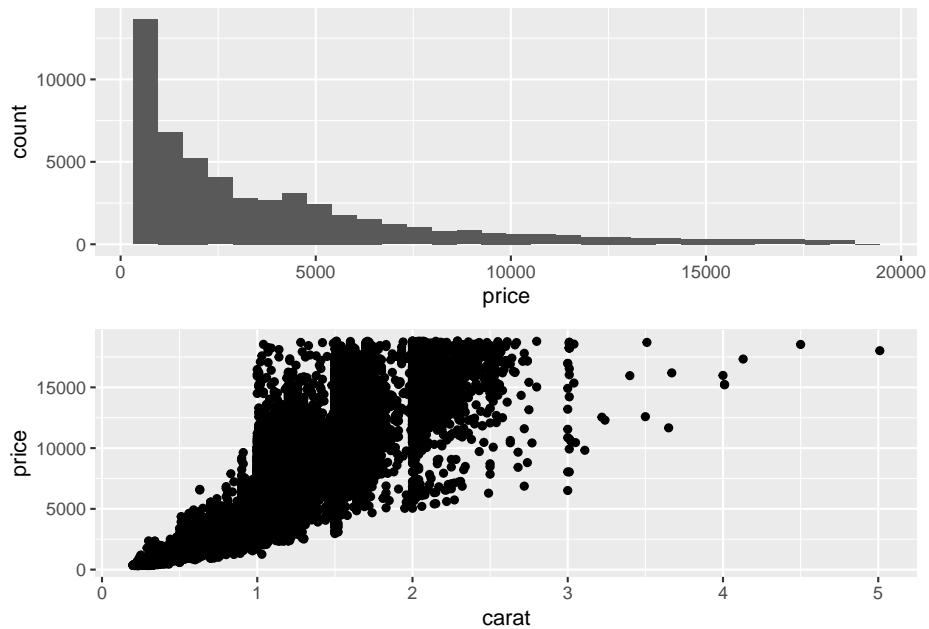
Diese 4 Objekte a, b, c und d setzen wir dann einfach in `ggarrange()` ein. Es macht Sinn den Graphen auch ein Label zu geben, dann kann man in der Fußnote der Abbildung darauf verweisen, was in welchem Graphen zu sehen ist (z.B. “In Graphik A sieht man ein Histogramm von...”):

```
ggarrange(a, b, c, d, labels = c("A", "B", "C", "D"))
```



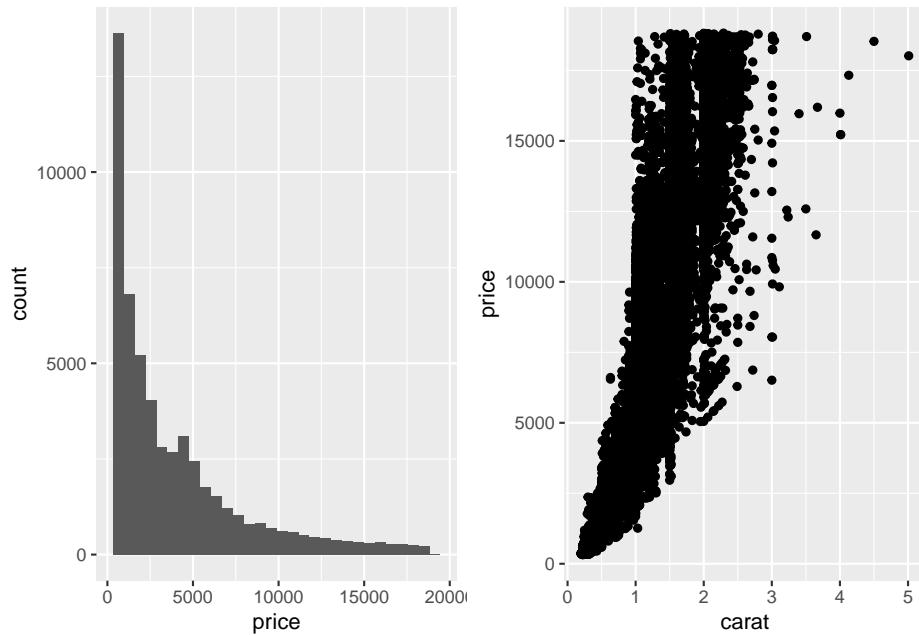
Wir können auch steuern, wie das Gitter der Graphen zusammengesetzt sein soll, zum Beispiel untereinander...

```
ggarrange(a, b, nrow = 2)
```



... oder nebeneinander:

```
ggarrange(a, b, ncol = 2)
```



12.12 Abbildungen exportieren

Der Export der Graphen meint im Prinzip folgendes: Speichere mir den Graphen in mein Arbeitsverzeichnis und das in einem gängigen Bildformat (.pdf, .png, .jpg, .tif...).

Das Vorgehen funktioniert für alle Formate gleich:

1. Wenn noch nicht geschehen, mit `setwd()` das gewünschte Arbeitsverzeichnis setzen
2. Mittels Befehl angeben, welche Art von Export gewünscht ist, z.B. `png()`
3. Den Code für den `ggplot()` Graphen ausführen
4. Mit dem Befehl `dev.off()` anzeigen, dass der Code für den Graphen fertig ist, die Datei also geschrieben werden soll.

Größe (in cm) und Auflösung (in dpi) lassen sich direkt innerhalb des Befehls steuern.

Ein Beispiel:

```
png(file = "Figure1.png", width = 5, height = 5, units = "in", res = 300)

ggplot(data = diamonds, aes(x = price, colour = cut)) +
  geom_histogram(binwidth = 39)

dev.off()

## pdf
## 2
```

Das Argument `file` gibt den Ziellnamen der .png Datei an. Die Argumente `width` und `height` die Dimensionen (hier quadratisch → für rechteckig einfach eine der beiden vergrößern/verkleinern). Die Auflösung von 300 dpi ist sehr gebräuchlich und muss nur selten verändert werden.

Ausführen des Befehls speichert Ihnen die Datei Figure1.png in das Arbeitsverzeichnis. Schauen Sie gleich einmal nach!

Chapter 13

Tabellen

Das automatische Erstellen von Tabellen in R bietet eine Reihe von praktischen Vorteilen:

1. **Effizienz:** Es spart Zeit und reduziert potenzielle Fehlerquellen. Anstatt Tabellen manuell zu erstellen und Daten händisch einzugeben, können Tabellen in R mithilfe von Funktionen und Skripten automatisch generiert werden.
2. **Transparenz:** Es ermöglicht eine hohe Reproduzierbarkeit. Durch die Verwendung von Skripten kann die Tabellenerstellung dokumentiert und leicht wiederholt werden, was die Transparenz und Nachvollziehbarkeit der Datenanalyse verbessert.
3. **Dynamische Veränderbarkeit:** Automatisch erstellte Tabellen können einfach aktualisiert werden, wenn neue Daten verfügbar sind, was die Aktualität der Ergebnisse gewährleistet.

Unser Ziel ist es, einen in R verfügbaren Inhalt (z.B. einen `data.frame`) in eine publikationsfertige Tabelle zu transformieren. Die Tabelle können wir dann z.B. in Word oder Powerpoint Dokumenten weiterverwenden, oder in einem RMarkdown-Skript nutzen, welches uns das Paper automatisch formatiert.

13.1 `data.frames`

Der einfachste Fall ist es, wenn die zu formatierende Tabelle bereits als `data.frame` vorliegt. Dieser hat bereits die Struktur einer Tabelle mit Zeilen und Spalten. Das “Verwandeln” eines `data.frame` in eine Tabelle ist z.B. mit dem Paket `flextable` möglich.

```
df = data.frame(Geschlecht = c("männlich", "männlich", "weiblich", "weiblich"),
                Konzentration = c(3, 7, 4, 7),
```

```

Schlafdauer = c(6, 14, 7, 12),
Schlafqualität = c(5, 8, 2, 1))

df

##   Geschlecht Konzentration Schlafdauer Schlafqualität
## 1 männlich            3             6              5
## 2 männlich            7            14              8
## 3 weiblich            4             7              2
## 4 weiblich            7            12              1

```

Um den `data.frame` in eine Tabelle umzubauen, nutzen wir den `flextable()` Befehl:

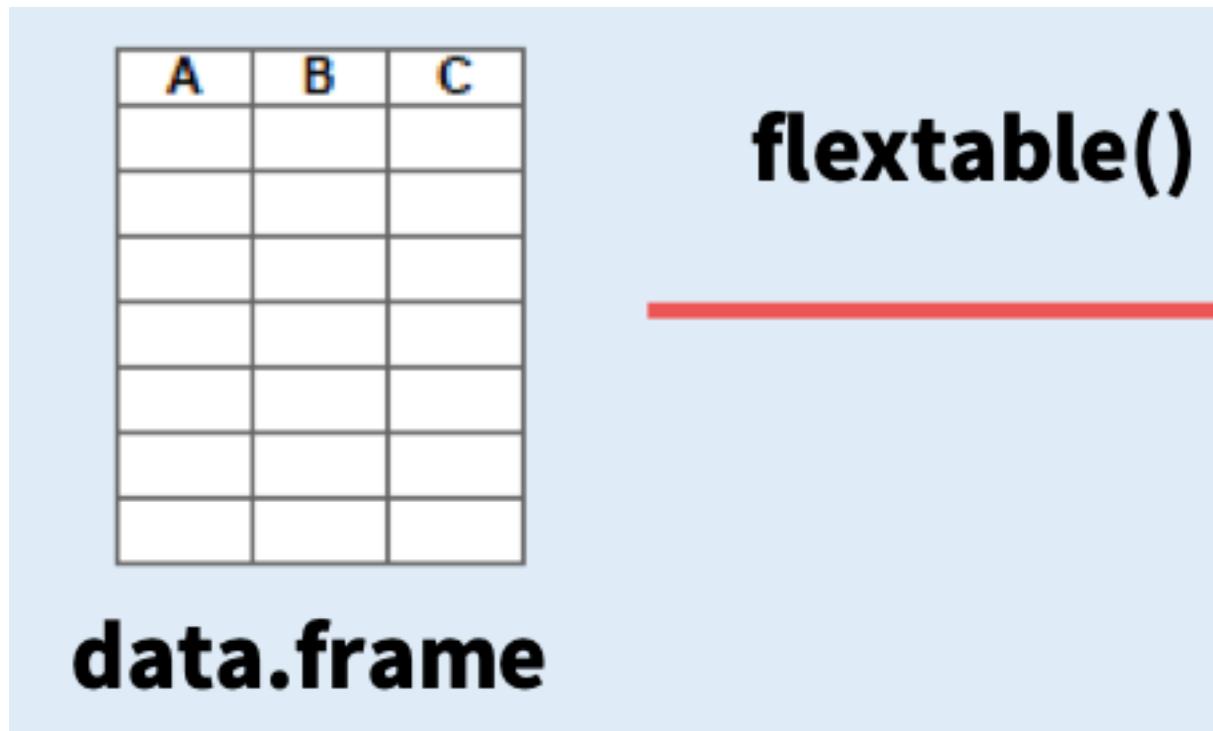


Figure 13.1: Von `data.frame` zur Tabelle

```

library(flextable)

##
## Attaching package: 'flextable'

## The following objects are masked from 'package:ggpubr':
##

```

```
##     border, font, rotate
## The following object is masked from 'package:purrr':
## 
##     compose
flextable(df)
```

Geschlecht	Konzentration	Schlafdauer	Schlafqualität
männlich	3	6	5
männlich	7	14	8
weiblich	4	7	2
weiblich	7	12	1

Die Bestandteile der Tabelle sind wie folgt:

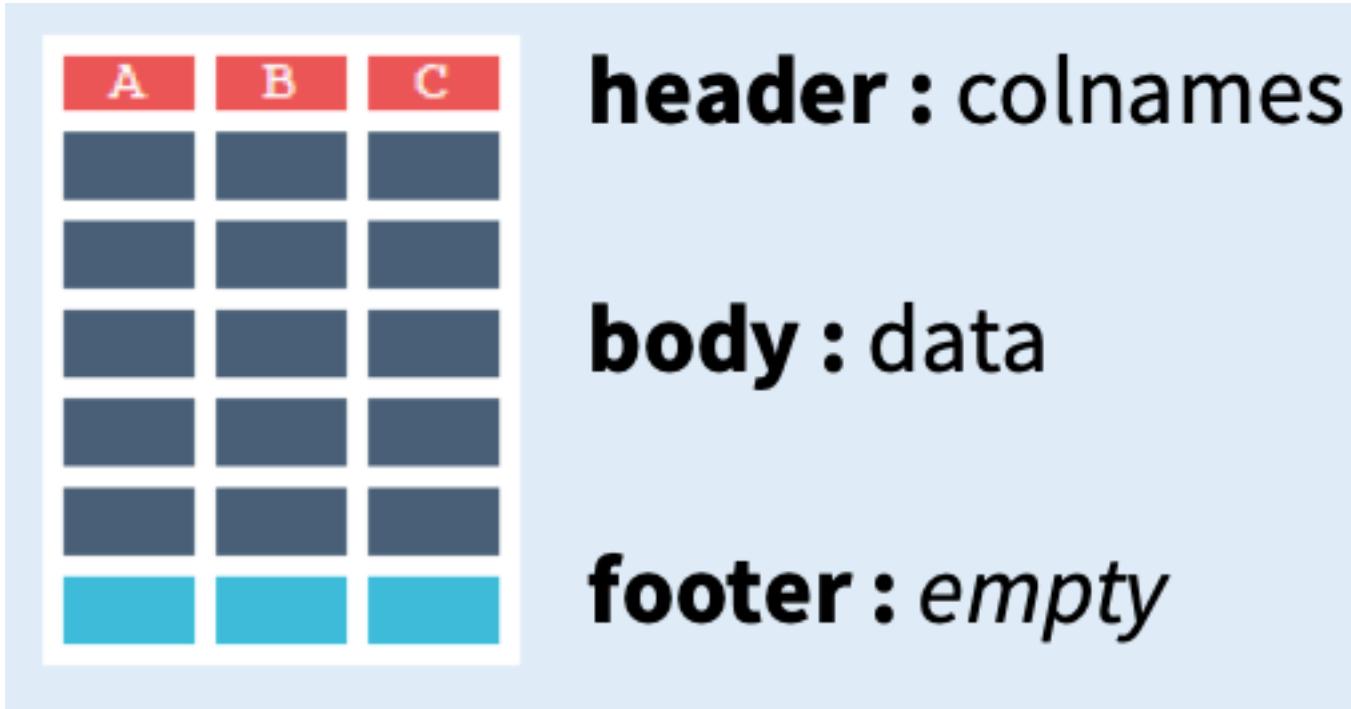


Figure 13.2: Bestandteile der Tabelle

Der allgemeine Aufbau einer `flextable` Funktion lautet:

```
function(x, i, j, part, args)
```

- x = flextable Objekt (Ergebnis von `flextable()`)
- i = Zeilenindex
- j = Spaltenindex,
- part = Tabellenteil (header, body, footer)
- args = spezielle weitere Anpassungen

13.2 Format

Es können diverse Aspekte der Tabelle angepasst werden.

13.2.1 Text

13.2.1.1 Fett schreiben

Zum Verwenden einer fetten Schriftart nutzen wir die `bold()` Funktion:

```
bold(flextable(df), j = 1, bold = T)
```

Geschlecht	Konzentration	Schlafdauer	Schlafqualität
männlich	3	6	5
männlich	7	14	8
weiblich	4	7	2
weiblich	7	12	1

13.2.1.2 Kursiv schreiben

Zum Verwenden einer kursiven Schriftart nutzen wir die `italic()` Funktion:

```
italic(flextable(df), j = 1, italic = T)
```

Geschlecht	Konzentration	Schlafdauer	Schlafqualität
<i>männlich</i>	3	6	5
<i>männlich</i>	7	14	8
<i>weiblich</i>	4	7	2
<i>weiblich</i>	7	12	1

13.2.1.3 Textgröße anpassen

Zur Veränderung der Textgröße nutzen wir die `fontsize()` Funktion:

```
fontsize(flextable(df), j = 1, size = 6)
```

Geschlecht	Konzentration	Schlafdauer	Schlafqualität
männlich	3	6	5
männlich	7	14	8
weiblich	4	7	2
weiblich	7	12	1

13.2.1.4 Textfarbe anpassen

Zur Veränderung der Textfarbe nutzen wir die `color()` Funktion:

```
color(flextable(df), j = 1, color = "red")
```

Geschlecht	Konzentration	Schlafdauer	Schlafqualität
männlich	3	6	5
männlich	7	14	8
weiblich	4	7	2
weiblich	7	12	1

13.2.1.5 Text hervorheben

Zum Hervorheben von Text nutzen wir die `highlight()` Funktion:

```
highlight(flextable(df), j = 1, color = "yellow")
```

Geschlecht	Konzentration	Schlafdauer	Schlafqualität
männlich	3	6	5
männlich	7	14	8
weiblich	4	7	2
weiblich	7	12	1

13.2.1.6 Text rotieren

Zum Rotieren von Text nutzen wir die `rotate()` Funktion:

```
rotate(flextable(df), j = 2, rotation = "tbrl")
```

Geschlecht	Konzentration	Schlafdauer	Schlafqualität
männlich	3	6	5
männlich	7	14	8
weiblich	4	7	2
weiblich	7	12	1

13.2.2 Zellen

13.2.2.1 Alignment

Um den Text in der Zelle auszurichten nutzen wir die `align()` Funktion. Die Argumente sind “left”, “right” und “center”.

```
align(flextable(df), j = 2, align = "left")
```

Geschlecht	Konzentration	Schlafdauer	Schlafqualität
männlich	3	6	5
männlich	7	14	8
weiblich	4	7	2
weiblich	7	12	1

Um den Text in der Zelle vertikal auszurichten nutzen wir die `valign()` Funktion. Die Argumente sind “top” und “bottom”.

```
valign(flextable(df), j = 2, valign = "top")
```

Geschlecht	Konzentration	Schlafdauer	Schlafqualität
männlich	3	6	5
männlich	7	14	8
weiblich	4	7	2
weiblich	7	12	1

Geschlecht	Konzentration	Schlafdauer	Schlafqualität
------------	---------------	-------------	----------------

13.2.2.2 Abstände

```
padding(flextable(df), i = 1, padding = 15)
```

Geschlecht	Konzentration	Schlafdauer	Schlafqualität
männlich	3	6	5
männlich	7	14	8
weiblich	4	7	2
weiblich	7	12	1

13.2.2.3 Hintergrundfarbe

```
bg(flextable(df), i = 1, bg = "grey")
```

Geschlecht	Konzentration	Schlafdauer	Schlafqualität
männlich	3	6	5
männlich	7	14	8
weiblich	4	7	2
weiblich	7	12	1

13.3 Layout

13.3.1 Titelzeilen einfügen

13.4 Tabelle speichern

Ein Export der Tabelle in ein weiterverwendbares Format ist für folgende Dateitypen möglich:

- .html (Web)
- .docx (Word)

- .pptx (Powerpoint)
- .png (Bild)

```
# save_as_html(df, "meine_tabelle.html")
# save_as_docx(df, "meine_tabelle.docx")
# save_as_pptx(df, "meine_tabelle.pptx")
# save_as_image(df, "meine_tabelle.png")
```

Chapter 14

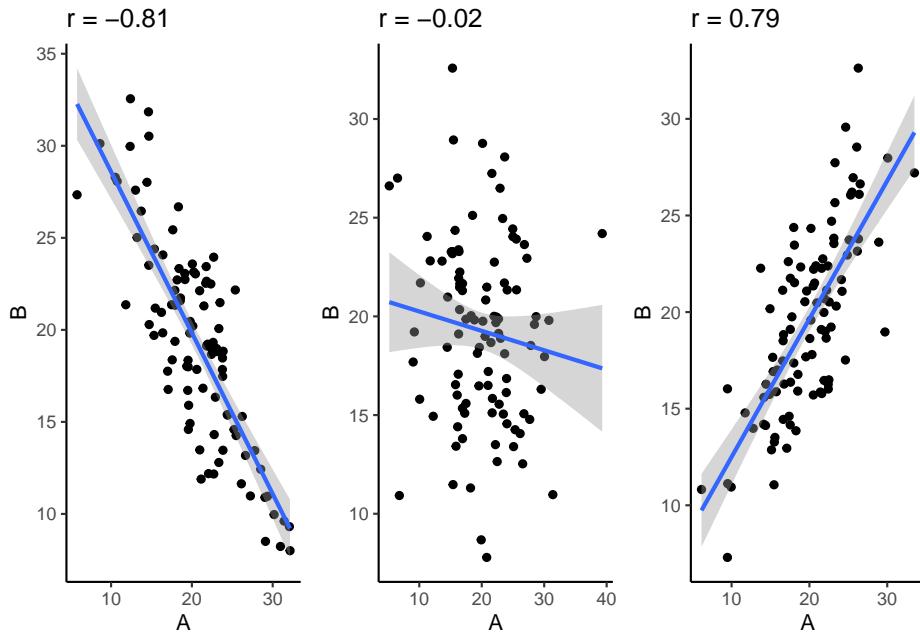
Korrelation

Als möglichen Hypothesentest schauen wir uns die Korrelationen an:

Bei einem Korrelationstest wird die Beziehung zwischen zwei Variablen auf einer Verhältnis- oder Intervallskala untersucht: z. B. Größe und Gewicht oder Einkommen und Selbstvertrauen

Die Teststatistik bei einem Korrelationstest wird als Korrelationskoeffizient bezeichnet und durch den Buchstaben r dargestellt.

Der Koeffizient kann zwischen -1 und +1 liegen, wobei -1 für eine starke negative Beziehung und +1 für eine starke positive Beziehung steht.



14.1 Hypothesen

```
data <- rnorm_multi(n = 100,
                     mu = c(100, 8),
                     sd = c(25, 2),
                     r = c(0.69),
                     varnames = c("Konzentration", "Schlaf"),
                     empirical = FALSE)
```

Die Korrelation prüft als Signifikanztest, ob ein Zusammenhang zwischen 2 Variablen besteht.

Da Korrelationskoeffizient von 0 bedeutet, dass kein Zusammenhang zwischen den Variablen besteht, wird im Signifikanztest i.d.R. r gegen 0 getestet (daher Nullhypothese).

Folgende Hypothesen sind denkbar:

Test auf Zusammenhang zwischen den beiden Variablen (ungerichtet):

- $H_0: r = 0$
- $H_1: r \neq 0$

Test auf positiven/negativen Zusammenhang zwischen den beiden Variablen (gerichtet):

- $H_0: r \leq 0$
- $H_1: r > 0$

beziehungsweise...

- $H_0: r \geq 0$
- $H_1: r < 0$

Eine typische psychologische Fragestellung für eine Zusammenhangshypothese könnte sein, ob die Anzahl der in der Nacht geschlafenen Stunden (**Schlaf**) mit der Leistung in einem Konzentrationstest zusammenhängt (**Konzentration**).

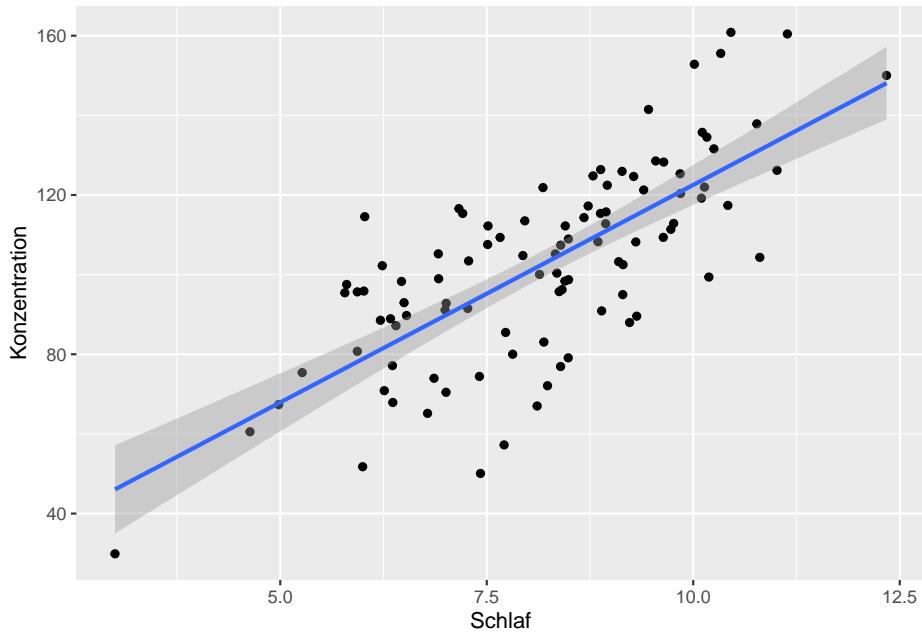
Ein entsprechender Datensatz könnte wie folgt aussehen (die ersten 6 Zeilen von $N = 100$):

```
head(data)
```

```
##   Konzentration     Schlaf
## 1      112.23087 8.450584
## 2      105.20617 8.330932
## 3      108.94331 8.488257
## 4      104.33156 10.805483
## 5      80.73751  5.933128
## 6      91.49791  7.269685
```

Um einen ersten Eindruck vom Zusammenhang zu gewinnen, können wir uns die Daten in einem Streudiagramm darstellen:

```
ggplot(data = data, aes(x = Schlaf, y = Konzentration)) +
  geom_point() +
  geom_smooth(method = "lm")
```



14.2 Berechnung der Korrelation

Der Korrelationskoeffizient lässt sich mit der R-Basisfunktion `cor()` berechnen. Dafür schreiben wir ganz einfach die beiden zu korrelierenden Variablen nebeneinander in die Funktion:

```
cor(data$Konzentration, data$Schlaf)
```

```
## [1] 0.732401
```

Wie Sie sehen, erhalten wir einen Korrelationskoeffizienten von $r = 0.73$. Also einen positiven Zusammenhang.

14.3 Unterschiedliche Korrelationsmethoden

14.3.1 Pearson-Korrelation

Nutzen wir die `cor()` Funktion ohne weitere Spezifikationen, wird der sogenannte **Pearson** Korrelationskoeffizient berechnet.

Dieser stellt jedoch gewisse Voraussetzungen an die Daten:

- Intervallskalenniveau

- keine Ausreißer
- Normalverteilung der Variablen

Sollte eine (oder mehrere) der Voraussetzungen nicht erfüllt sein, berechnen wir einen der folgenden alternativen Korrelationskoeffizienten

14.3.2 Spearman-Korrelation (aka Rangkorrelation)

Der **Spearman** Korrelationskoeffizient funktioniert im Wesentlichen wie der Pearson Korrelationskoeffizient, jedoch wird er auf Ordinalskalenniveau berechnet.

Das macht ihn unempfindlicher gegenüber Verteilungsverletzungen und Ausreißern.

Die Berechnung des **Spearman** Korrelationskoeffizienten erfolgt nach derselben Methode, mit einer kleinen Spezifikation:

```
cor(data$Konzentration, data$Schlaf, method = "spearman")
```

```
## [1] 0.7183798
```

In der Regel ist die Abweichung der beiden Korrelationskoeffizienten voneinander nicht allzu hoch.

14.3.3 Kendall-Korrelation

Die Rangkorrelationskoeffizienten von Spearman und Kendall sind beide Koefizienten, die den Zusammenhang ordinalskalierten Merkmale beschreiben können.

Der Vorteil des Kendall τ liegt darin, dass seine Verteilung bei kleineren Stichprobenumfängen bessere statistische Eigenschaften bietet und er weniger empfindlich gegen Ausreißer-Rangpaare ist.

```
cor(data$Konzentration, data$Schlaf, method = "kendall")
```

```
## [1] 0.5292929
```

14.4 Korrelation als Hypothesentest

Wie Sie bereits bemerkt haben werden, liefert Ihnen die `cor()` Funktion lediglich den Korrelationskoeffizienten, jedoch keine Informationen über statistische Signifikanz.

Ein Signifikanztest für den Korrelationskoeffizienten lässt sich jedoch einfach mit der Funktion `cor.test()` rechnen. Die Argumente der Funktion sind in der folgenden Tabelle zusammengefasst:

Argument	Description
<code>formula</code>	Argument in Formelformat <code>~ x + y</code> , x und y sind die Namen der Variablen deren bivariater Zusammenhang getestet werden soll. Diese Variablen sollten in getrennten Spalten eines dataframes stehen.
<code>data</code>	Der data.frame, der x und y enthält
<code>alternative</code>	Hier kann die Richtung der Alternativhypothese angegeben werden. Wählen Sie "two.sided" für eine ungerichtete Hypothese, oder "greater" bzw. "less" für eine gerichtete Hypothese.
<code>method</code>	Gibt die Art des zu berechnenden Korrelationskoeffizienten an. "pearson" (default) steht für die Produkt-Moment Korrelation, "kendall" und "spearman" stehen für die Rangkorrelationen nach Kendall und Spearman.
<code>subset</code>	Hier kann direkt ein Teildatensatz ausgewählt werden. Z.B.: <code>subset = sex == "female"</code>

Wenn wir zum Beispiel testen wollten, ob ein signifikanter negativer Zusammenhang zwischen Konzentration und Schlaf besteht, könnten wir das mit folgendem Signifikanztest prüfen:

```
cor.test(data$Konzentration, data$Schlaf, method = "pearson", alternative = "less")

##
## Pearson's product-moment correlation
##
## data: data$Konzentration and data$Schlaf
## t = 10.649, df = 98, p-value = 1
## alternative hypothesis: true correlation is less than 0
## 95 percent confidence interval:
## -1.0000000 0.8008208
## sample estimates:
##      cor
## 0.732401
```

Chapter 15

Tests für einfache Gruppenvergleiche

15.1 Ein-Stichproben t-Test

```
data = read.csv("data/One sample t-test.csv")
names(data) = c("gewicht", "groesse")
```

15.1.1 Hypothesen

Mit einem Ein-Stichproben t-Test vergleichen wir den Mittelwert einer Gruppe mit einem hypothetischen Mittelwert.

Der Test prüft also anhand des Mittelwerts einer Stichprobe, ob der Erwartungswert in der entsprechenden Population gleich einem vorgegebenen Wert ist (dem unter H_0 erwarteten μ_0).

Es sind folgende Hypothesen denkbar:

Test auf Unterschiedlichkeit von dem Referenzwert (ungerichtet):

- $H_0: \mu = \mu_0$
- $H_1: \mu \neq \mu_0$

Test, ob Mittelwert größer/kleiner als Referenzwert ist (gerichtet):

- $H_0: \mu \leq \mu_0; H_1: \mu > \mu_0$
- $H_0: \mu \geq \mu_0; H_1: \mu < \mu_0$

Zum Beispiel könnten wir eine Stichprobe von Menschen aus Deutschland erhoben haben und uns dafür interessieren, ob diese signifikant größer, bzw. kleiner als der Durchschnitt in Deutschland sind.

Die ersten Zeilen des Stichprobendatensatzes könnten so aussehen:

```
##   gewicht groesse
## 1     77    182
## 2     68    177
## 3     76    170
## 4     76    167
## 5     69    186
## 6     71    178
```

Zunächst brauchen wir einen hypothetischen Vergleichswert. Sucht man die geschlechterübergreifende Durchschnittsgröße in Deutschland im Internet findet man einen Wert von ca. 173 cm.

15.1.2 Deskriptive Einordnung

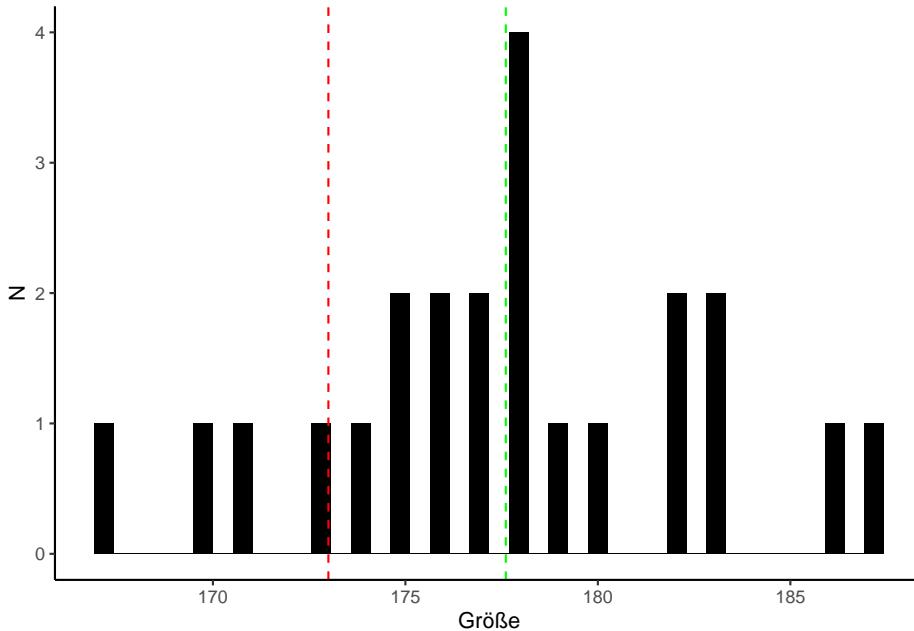
Die Berechnung unseres Mittelwerts ist einfache Deskriptivstatistik:

```
mean(data$groesse)
```

```
## [1] 177.6087
```

Der Sachverhalt lässt sich auch graphisch darstellen:

```
ggplot(data = data, aes(x = groesse)) +
  geom_histogram(bins = 40, fill = "black") +
  labs(x = "Größe", y = "N") +
  geom_vline(xintercept = 173, linetype = "dashed", colour = "red") +
  geom_vline(xintercept = mean(data$groesse), linetype = "dashed", colour = "green") +
  theme_classic()
```



Die grüne Linie im Zentrum des Histogramms stellt unseren Stichprobenmittelwert dar. Die rote Linie ist der angenommene Mittelwert in der Population von 173 cm.

15.1.3 Test durchführen

Zur Durchführung des Tests nutzen wir die in der Grundform von R vorinstallierte `t.test()` Funktion.

```
t.test(data$groesse, mu = 173, alternative = "two.sided", conf.level = 0.95)

##
## One Sample t-test
##
## data: data$groesse
## t = 4.4969, df = 22, p-value = 0.0001792
## alternative hypothesis: true mean is not equal to 173
## 95 percent confidence interval:
## 175.4833 179.7341
## sample estimates:
## mean of x
## 177.6087
```

Wir wählen die Variable `groesse` innerhalb unseres Datensatzes mit dem `$` Zeichen an. Über das `mu` Argument geben wir den hypothetischen Vergleichswert

an. Unter `alternative` können wir auswählen, ob der Test gerichtet oder ungerichtet (aka ein- oder zweiseitig) durchgeführt werden soll. Je nach Hypothese wählen wir "`two.sided`" für einen ungerichteten Test und entweder "`less`" oder "`greater`" für einen gerichteten Test. Das `conf.level` entspricht unserem Signifikanzniveau.

15.1.4 Ergebnis interpretieren

Das Ergebnis des Tests lässt sich am P-Wert ablesen ($p=0.0001792$). Ist der P-Wert kleiner als das gewählte Signifikanzniveau (i.d.R $\alpha = .05$) unterscheidet sich unser Stichprobenmittelwert (177.61 cm) signifikant von der Durchschnittsgröße in Deutschland (173 cm). Wir verwerfen also die Nullhypothese (H_0) zugunsten unserer Alternativhypothese (H_1).

15.1.5 Ergebnis berichten

Die relevanten Parameter zum Berichten eines Ein-Stichproben t-Tests sind

- M (Mittelwert)
- Grenzen des Konfidenzintervalls des Mittelwerts
- t-Wert (Teststatistik)
- df (Freiheitsgrade)
- P-Wert

Beim Berichten im Fließtext schreibt man:

Die Größe in der Stichprobe unterschied sich signifikant von der Durchschnittsgröße in Deutschland (173 cm), $M = 177.61$, 95% CI (175.48, 179.73), $t(22) = 4.5$; $p < .001$.

Diese Werte lassen sich wie folgt aus dem t-Test Objekt extrahieren:

t-Wert (Teststatistik):

```
t.test(data$groesse, mu = 173, alternative = "two.sided", conf.level = 0.95)$statistic
```

```
##          t
## 4.496872
```

df (Freiheitsgrade):

```
t.test(data$groesse, mu = 173, alternative = "two.sided", conf.level = 0.95)$parameter
```

```
## df
## 22
```

P-Wert:

```
t.test(data$groesse, mu = 173, alternative = "two.sided", conf.level = 0.95)$p.value
## [1] 0.0001792355

Grenzen des Konfidenzintervalls (unten & oben):
t.test(data$groesse, mu = 173, alternative = "two.sided", conf.level = 0.95)$conf.int[1]
## [1] 175.4833
t.test(data$groesse, mu = 173, alternative = "two.sided", conf.level = 0.95)$conf.int[2]
## [1] 179.7341
```

15.1.6 Effektstärke

15.1.6.1 Cohen's d

Die am häufigsten verwendete Effektstärke für den Ein-Stichproben t-Test ist Cohen's d ?. Cohen's d lässt sich mit dem Paket `effsize` berechnen. Dieses verwendet praktischerweise die gleiche Schreibweise, wie der t-Test:

```
effsize::cohen.d(data$groesse, f = NA, mu = 173)
##
## Cohen's d (single sample)
##
## d estimate: 0.9376626 (large)
## Reference mu: 173
## 95 percent confidence interval:
##       lower      upper
## 0.02651145 1.84881385
```

Auch die Einzelparameter von Cohen's d lassen sich extrahieren:

Cohen's d:

```
effsize::cohen.d(d = data$groesse, f = NA, mu = 173)$estimate
## [1] 0.9376626

Grenzen des Konfidenzintervalls (unten & oben):
effsize::cohen.d(d = data$groesse, f = NA, mu = 173)$conf.int[1]
##
##       lower
## 0.02651145
effsize::cohen.d(d = data$groesse, f = NA, mu = 173)$conf.int[2]
```