Testtheorie mit R

Martin Papenberg

Testtheorie mit R

Autor: Martin Papenberg

martin.papenberg@hhu.de, Website

„Testtheorie mit R“ wird regelmäßig erweitert. Die aktuelle Version kann unter https://osf.io/y4a6k/ abgerufen werden.

Letzte Aktualisierung: 7. Mai 2019

Lizenz

Inhaltsverzeichnis

1 Einstieg

„Testtheorie mit R“ bietet einen Einstieg in die statistische Programmiersprache R. Es wurde ursprünglich als Begleitskript für eine ein-semestrige Lehrveranstaltung im Master-Studiengang Psychologie an der Heinrich-Heine-Universität Düsseldorf entworfen. Im Seminar wird kein R-Vorwissen vorausgesetzt. Ich habe das Skript in der Hoffnung öffentlich gemacht, dass es auch für andere R-Einsteiger nützlich sein kann.

R kann – unter anderem – als eine Alternative zur kommerziellen Statistik-Software IBM-SPSS verwendet werden. Anders als SPSS ist R *frei*, d.h. wir können es gratis aus dem Internet runterladen, auf beliebig vielen Computern installieren, und unsere Analysen mit jeder anderen Person teilen, da niemand eine Lizenz zur Nutzung benötigt. Da R mithilfe von *Paketen* beliebig erweitert werden kann, stehen neue statistische Verfahren häufig schnell zur Verfügung (etwa *Bayesianische Statistik*). Die Nutzung von R ist in den letzten Jahren stark angestiegen. Auch in der psychologischen Forschung wird R immer mehr zum Standard.

Wir lernen die Nutzung von R anhand von Beispielen der psychologischen Diagnostik beziehungsweise der Testtheorie kennen. Dabei werden auch echte Datensätze verwendet, beispielsweise ein Datensatz zum *Narcissistic Personality Inventory*, der online frei über das „Open Source Psychometrics Project” (https://openpsychometrics.org/) verfügbar ist.

1.1 Über dieses Skript

Dieses Skript wurde als Begleitmaterial für eine Lehrveranstaltung konzipiert. Die Veranstaltung selbst hat einen starken praktischen Anteil; in jeder Stunde werden Übungsaufgaben in R bearbeitet. Das Skript bietet den theoretischen Unterbau. Die Übungen des Seminars aus dem Sommersemester 2018 und die zur Bearbeitung nötigen Daten – wie auch der jeweils aktuelle Stand dieses Skripts – können unter https://osf.io/y4a6k/ abgerufen werden. Unter https://m-py.github.io/TesttheorieR/ findet sich eine online lesbare Version. Das Skript wird stetig aktualisiert; diese Version wurde am 07. Mai 2019 erstellt.

Wer R lernen möchte, muss sich klar machen, dass die reine Aufarbeitung einer oder mehrerer schriftlicher Lektüren zu diesem Zweck nicht ausreicht. Die praktische Anwendung – das Ausprobieren und „Rumspielen“ – sollte einen mindestens genau so großen Anteil haben. Erst durch die Fehler, die man beim praktischen Arbeiten macht – und die macht man immer –, lassen sich die eigenen R-Fertigkeiten weiterentwickeln.

Insgesamt gilt: Das Skript und die Übungen stellen nur eine kleine Auswahl dessen vor, was R bietet. Notwendigerweise werden Inhalte ausgelassen. Bei der Darstellung wird vor allem Wert auf die inhaltliche Sinnhaftigkeit und Verständlichkeit gelegt; dafür kann es vorkommen, dass – wenn angemessen – Kompromisse bei der technischen Genauigkeit eingegangen werden. Kapitel 2 enthält beispielsweise eine Beschreibung verschiedener Datentypen in R (Zahlen, Text etc.). Diese Liste deckt zwar die für uns wichtigsten Datentypen ab, ist aber nicht vollständig. Aus inhaltlichen Gründen folgt sie außerdem nicht der R-internen „technischen“ Kategorisierung. Auch hat R für so gut wie jede allgemeine Regel mindestens eine Ausnahme. Auf solche

Spezialfälle werde ich bei der Beschreibung allgemeiner Grundsätze der Programmiersprache R nicht immer Rücksicht nehmen. Das Skript ist so ausgelegt, dass ein Grundstein an Kenntnissen gelegt wird, jedoch erfordert die Meisterung von R noch weitere eigenständige Einarbeitung.

1.1.1 Feedback und Fehlermeldungen

Für Feedback und eine Rückmeldung bei der Entdeckung von Fehlern im Skript (auch und insbesondere bei der Entdeckung einfacher Rechtschreibfehler, doppelter oder fehlender Wörter, fehlender Kommas etc.) bin ich sehr dankbar! Meldungen können mir an martin.papenberg@hhu.de gesendet werden.

1.1.2 Danksagung

Ich danke Juliane Tkotz für ihre wertvollen Beiträge und ihr nützliches Feedback zum Skript. Hanna Siegers, Marlene Wettstein, Frank Calio und Ingo Weigel danke ich für Fehlermeldungen.

Zur Erstellung des Skripts wurden R (R Core Team, 2018) und die R-Pakete *bookdown* (Xie, 2016), *knitr* (Xie, 2015), *rmarkdown* (Allaire et al., 2017) genutzt.

1.2 Erste Schritte mit R

Im Seminar nutzen wir die „integrierte Entwicklungsumgebung“ (engl: integrated developement environment; *IDE*) RStudio, um mit R zu arbeiten. Zum Nachvollziehen des Skripts und der Übungen solltet ihr deswegen RStudio auf eurem eigenen Rechner/Laptop installieren.[1] Das geht über diesen Link:

https://www.rstudio.com/products/rstudio/download/#download

Vermutlich wollt ihr eine Installationsdatei für Windows herunterladen, es gibt aber auch Optionen für Linux und Mac. Dafür schaut ihr unter „Installers for Supported Platforms“ beispielsweise unter „RStudio 1.1.442 - Windows Vista/7/8/10“.

Wichtig: RStudio ist nur die R-Umgebung, die wir nutzen, aber nicht die Programmiersprache R selbst. R muss noch einmal unter https://cran.r-project.org/ gesondert heruntergeladen werden.

Hier könnt ihr beispielsweise über „Download R for Windows“ → „install R for the first time” gehen.

[1] Falls ihr eine andere Umgebung benutzt, ist das natürlich auch kein Problem. Alternativen sind beispielsweise rkward (https://rkward.kde.org/) oder emacs ESS (https://ess.r-project.org/).

1.2.1 Die R-Konsole

Wenn wir R und RStudio installiert haben, können wir unsere ersten Schritte mit R nehmen. Dafür geben wir R-Code in die sogenannte Konsole ein. Im Normalfall finden wir in RStudio die Konsole in der Anzeige auf der linken Seite (je nachdem, wie ihr RStudio geöffnet habt, befindet sich die Konsole auch links unten). Wir erkennen die Konsole daran, dass die Zeile, in die wir unsere R-Befehle eintragen, mit einem > beginnt. Diese spitze Klammer fordert uns zum Eingeben von R-Code auf. Um unseren ersten R-Befehl auszuführen, schreiben wir Folgendes in die Konsole und drücken Enter:

```
> "Hallo Welt!"
```

Wenn folgende Ausgabe erscheint, hat die Installation funktioniert:

```
[1] "Hallo Welt!"
```

Wir können die Arbeit mit R beziehungsweise der R-Konsole als Kommunikation verstehen: Wir teilen R etwas mit, und R gibt uns dazu passend etwas zurück – **wenn unsere Anfrage ein *syntaktisch* korrekter R-Befehl war**. Andernfalls gibt R eine Fehlermeldung aus. Zum Beispiel können wir die R-Konsole als Taschenrechner benutzen:

```
1 + 3
```

```
[1] 4
```

```
3 - 17
```

```
[1] -14
```

```
3 * 2
```

```
[1] 6
```

```
3^2
```

```
[1] 9
```

```
3^2 + 4^2
```

```
[1] 25
```

```
10 / 5
```

```
[1] 2
```

```
## Auf Klammerung achten:
(3 + 5) / 2
```

```
[1] 4
```

```
3 + 5 / 2
```

```
[1] 5.5
```

1.2.2 Der Skript-Editor

Zumeist werden wir R-Code nicht nur in der Konsole schreiben und ausführen. Wenn wir einen Befehl in der Konsole geschrieben und mit Enter ausgeführt haben, ist er ja quasi verschwunden.[2] Um Analysen übersichtlich, nachvollziehbar und reproduzierbar zu gestalten, speichern wir unseren Code in sogenannten Quellcode-Dateien ab. Dafür gibt es in RStudio (und auch in anderen R-Umgebungen) einen Texteditor. Wir können eine neue Quellcode-Datei unter „Datei → Neue Datei → R Skript“ öffnen. Darin können wir unseren R-Code schreiben und permanent auf unserem Computer abspeichern (und ggf. mit anderen Personen teilen). Textdateien, die R-Code enthalten, speichern wir mit der Dateiendung „.r“ oder „.R“ ab.

Das Praktische: Wenn wir Code im Editor schreiben, können wir ihn auch direkt von dort ausführen; wir müssen ihn nicht noch einmal in die Konsole „copy-pasten“. Das funktioniert so: Wenn sich mein Cursor in einer Zeile befindet und ich STRG-Enter drücke, wird der Code in dieser Zeile ausgeführt. Wenn ich einen Code-Abschnitt markiere, kann ich ebenso mit STRG-Enter genau diesen Abschnitt ausführen. Der Code wird in diesen Fällen an die Konsole gesendet, die dann die Ausführung des Codes für uns übernimmt.

1.2.3 Kommentare

Wenn ein #-Symbol in die Konsole oder den Skript-Editor geschrieben wird, wird der Rest dieser Zeile nicht mehr interpretiert, das heißt nicht als R-Code ausgeführt. Beispiel:

```
# 5 + 5
# nichts ist passiert - `R` gibt mir nicht 10 aus
```

Man nutzt #, um Code zu „kommentieren”, das heißt um zu erklären und zu dokumentieren, was der geschriebene Code macht. Diese Kommentare fügt man in den Quelldateien ein, in denen man die eigenen Analysen abspeichert. Dieses Skript enthält viel R-Code,[3] den ich stets kommentiere. (Ich habe die Angewohnheit, ein doppeltes ## am Anfang einer Zeile zu benutzen, aber das hat keinerlei Bedeutung.) Gewöhnt euch ebenfalls an, **immer** euren eigenen Code zu kommentieren. Das gilt sowohl für „richtige“ Projekte als auch für Übungsaufgaben. Das Kommentieren von Code ist vor allem nützlich, um anderen Personen euren Code zugänglich und verständlich zu machen. Im häufigsten Fall seid ihr selbst in zwei Wochen diese „andere” Person.

[2] Praktisch: Wenn ich mich in der Konsole befinde, kann ich mit den Pfeil-Tasten (vor allem wichtig: Pfeil-nach-oben) auf meine letzten Befehle wieder zugreifen. Probiert es aus.

[3] Codeblöcke im Skript bestehen immer aus dem eigentlichen Code (dieser ist leicht grau hinterlegt) und der *Ausgabe*, die bei Eingabe des Codes auch so in der R-Konsole erscheinen würde. Den Code könnt ihr auch selbst per Copy & Paste nachvollziehen (was ich auch empfehle). Die Ausgabe des Codes erkennt ihr meistens daran, dass sie mit [1] startet; so wird in der R-Konsole das erste Element der Ausgabe eines Vektors gekennzeichnet (siehe Kapitel 2).

1.3 Ausblick

In den nächsten zwei Kapiteln beschäftigen wir uns zunächst damit, wie `R` Daten darstellt. Dabei betrachten wir zunächst die grundlegenste Datenstruktur, den Vektor (Kapitel 2). Danach lernen wir `data.frames` kennen (Kapitel 3), die in `R` Datentabellen darstellen, wie wir sie auch aus Excel oder SPSS kennen. In Kapitel 4 werden wir psychometrische Datenauswertungen durchführen und dabei das Wissen anwenden, das wir zuvor erworben haben. In den Kapiteln 5 und 6 lernen wir mit Funktionen und Schleifen wichtige Programmiersprachenelemente kennen und werden sehen, wie wir damit unsere Arbeit automatisieren können.

2 Vektoren

Die einfachste und wichtigste Datenstruktur von R ist der *Vektor*. Ein Vektor ist beispielsweise eine einzelne Zahl wie in den Taschenrechner-Berechnungen in Kapitel 1. So gilt für die Berechnung `1 + 3`:

- `1` ist ein Vektor
- `3` ein Vektor
- das Ergebnis `4` ist auch ein Vektor

Das Interessante an Vektoren ist, dass der ein-elementige Vektor nur ein Spezialfall ist. Im Normalfall können Vektoren mehrere Elemente enthalten; die „atomare" Einheit in R ist also nicht ein einzelnes Element, sondern gleich eine Aneinanderreihung beliebig vieler[4] gleichartiger Elemente, etwa Zahlen. Statistische Berechnungen – wie die Berechnung eines Mittelwerts oder einer Standardabweichung – lassen sich direkt auf einer Menge an Daten durchführen, da diese in **einem** Vektor gespeichert sind. Diese „Vektorbasiertheit" ist vermutlich die größte Stärke von R für statistische Berechnungen.

Elemente zu Vektoren zusammenfügen (sprich: **mehrere** Vektoren zu **einem** Vektor zusammenfügen) funktioniert mit der *Funktion* `c` – die vermutlich basalste Funktion in R. Sie ist so simpel und grundlegend, dass man sie gegebenenfalls vergisst, wenn man sie braucht – versucht, sie zu erinnern!

```
## Füge mehrere Zahlen zu einem Vektor zusammen:
c(0.5, 1, 1.5) # Kommazahlen mit DezimalPUNKT schreiben
```

```
[1] 0.5 1.0 1.5
```

Man kann die Funktion `c` auch auf eine einzelne Zahl anwenden. Das ist dasselbe als würde man nur die Zahl eingeben:

```
c(1)
```

```
[1] 1
```

Folgendes geht auch, da `c` mehrere Vektoren zu **einem einzelnen** Vektor „verschmilzt":

```
c(0.5, 1, 1.5, c(1, 2, 3))
```

```
[1] 0.5 1.0 1.5 1.0 2.0 3.0
```

Auf mehrelementigen Vektoren kann man statistische Berechnungen durchführen, wie etwa die Bestimmung des arithmetischen Mittels, einer Standardabweichung, der Varianz, oder des Minimums oder Maximums:[5]

[4] Interessanterweise gibt es sogar Vektoren der Länge 0 – also Vektoren, die gar kein Element beinhalten. Das soll uns aber erst einmal nicht beschäftigen.

[5] R würde oft auch bei einelementigen Vektoren ein Ergebnis ausgeben, aber das ist zum Beispiel beim Mittelwert wenig sinnvoll.

```
## Berechne einen Mittelwert
mean(c(0.5, 1, 1.5))
```

```
[1] 1
```

```
## Berechne eine Standardabweichung
sd(c(0.5, 1, 1.5))
```

```
[1] 0.5
```

```
## Berechne eine Varianz:
var(c(0.5, 1, 1.5))
```

```
[1] 0.25
```

```
## Und jetzt noch einmal die Standardabweichung:
sqrt(var(c(0.5, 1, 1.5))) # was ist `sqrt`?
```

```
[1] 0.5
```

```
## Minimum:
min(c(0.5, 1, 1.5))
```

```
[1] 0.5
```

```
## Maximum:
max(c(0.5, 1, 1.5))
```

```
[1] 1.5
```

In diesem Code-Block haben wir implizit einen wichtigen Bestandteil von `R` kennengelernt: *Funktionen*. Für den Einstieg reicht es für uns, folgende Eigenschaften von Funktionen zu verstehen:

- Funktionen haben einen Namen – etwa: `mean` oder `c`
- Hinter dem Namen einer Funktion werden in Klammern ein oder mehrere *Argumente* übergeben, etwa: ein Vektor
- Wenn einer Funktion mehrere Argumente übergeben werden, werden diese mit Kommata separiert, etwa: `c(1, 2, 3)`
- Funktionen führen eine Berechnung durch und geben uns das Ergebnis zurück

Einfach gesagt nehmen also Funktionen Daten entgegen und geben wiederum Daten zurück. Der Großteil unserer Arbeit mit `R` ist die Anwendung von Funktionen. Es ist möglich Funktionsaufrufe zu verschachteln, wie dieses Beispiel zeigte:

```
sqrt(var(c(0.5, 1, 1.5)))
```

Hier wertet die Funktion `sqrt` (die Wurzel; engl. *square root*) das Ergebnis der Funktion `var` aus, um eine Standardabweichnung zu bestimmen. Der Aufruf ist also äquivalent zu `sqrt(0.25)`, da die Varianz von 0.5, 1, und 1.5 gleich 0.25 ist. Diese Beobachtung offenbart eine weitere wichtige Eigenschaft von `R`: Wir können unseren Code immer als das verstehen,

was er ergibt, wenn er von R ausgewertet wird. Es macht keinen Unterschied, ob ich das Ergebnis einer Berechnung selber „händisch“ aufschreibe – also hier 0.25 –, oder Code schreibe, der mir dieses Ergebnis generiert – hier: `var(c(0.5, 1, 1.5))`.

Eine nützliche und oft verwendete Kurzform, um Vektoren aufsteigender, ganzer Zahlen zu erstellen ist folgende:

```
1:20
```

```
 [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20
```

So lässt sich beispielsweise sehr einfach die Summe aller Zahlen von 1 bis 1,000 berechnen:

```
sum(1:1000)
```

```
[1] 500500
```

Wir können auch absteigende Sequenzen erstellen:

```
5:-5
```

```
 [1]  5  4  3  2  1  0 -1 -2 -3 -4 -5
```

Diese Tabelle enthält einige nützliche Funktionen, die auf Vektoren anwendbar sind (in R-Jargon: sie nehmen einen Vektor als *Argument* an) und jeweils selber auch einen Vektor zurückgeben:

Name	Funktionalität
`mean`	Berechnet den Mittelwert eines Vektors
`median`	Berechnet den Median eines Vektors
`sum`	Berechnet die Summe aller Elemente eines Vektors
`max`	Gibt den größten Wert eines Vektors zurück
`min`	Gibt den kleinsten Wert eines Vektors zurück
`length`	Gibt die Zahl der Elemente eines Vektors zurück
`sd`	Berechnet die Standardabweichung eines Vektors
`var`	Berechnet die Varianz eines Vektors
`sort`	Sortiert einen Vektor aufsteigend
`rev`	Kehrt die Reihenfolge der Elemente im Vektor um
`round`	Rundet die Elemente in einem Vektor
`sqrt`	Berechnet für jedes Element im Vektor die Quadratwurzel
`unique`	Gibt alle unterschiedlichen Werte eines Vektors aus

Für die Funktionen in dieser Tabelle gilt, dass sie zwar alle einen Vektor zurückgeben, aber die Länge des Ausgabevektors unterschiedlich sein kann. Die Funktionen `mean` und `sum` ergeben etwa Vektoren der Länge 1, da sie genau einen Kennwert bestimmen. Die Funktionen `sort`, `sqrt` und `round` geben hingegen einen Vektor zurück, der aus genauso vielen Elementen besteht wie der Eingabevektor.

Auch basale mathematische Berechnungen werden gleich auf alle Elemente eines Vektors angewendet:

```
1:10 * 2
```

```
 [1]  2  4  6  8 10 12 14 16 18 20
```

```
(1:10 * 2) - 1
```

```
 [1]  1  3  5  7  9 11 13 15 17 19
```

Hierbei werden die Operationen `* 2` bzw. `-1` direkt auf alle Elemente der Vektoren `1:10` bzw. `(1:10 * 2)` angewendet; die Ausgabe ist jeweils ein Vektor der Länge 10. Bei gleich langen Vektoren werden solche Operationen im Allgemeinen **komponentenweise** angewendet:

```
2:4 * 4:6 # entspricht c(2*4, 3*5, 4*6)
```

```
[1]  8 15 24
```

Dieses Verhalten ist typisch für `R`: Viele Funktionen und Operationen in `R` arbeiten komponentenweise, wenn zwei Vektoren gleicher Länge übergeben werden. Das Element an Position 1 im einen Vektor wird dann mit dem Element an Position 1 im anderen Vektor gepaart, das Element an Position 2 im einen Vektor mit dem Element an Position 2 im anderen Vektor – und so weiter.

Werden ein ein-elementiger Vektor und ein mehr-elementiger Vektor mit einer Berechnung (etwa einer Addition) verknüpft, wird normalerweise das einzelne Element mit allen Elementen des anderen Vektors „gepaart“.[6]

2.1 Variablen

Wir wollen unsere Daten nicht nur in der Konsole ausgeben lassen, sondern auch abspeichern und damit arbeiten. Ein essentieller Bestandteil einer jeden Programmiersprache ist es, Daten in Variablen abzuspeichern. Variablen sind Namen, mit deren Hilfe wir auf gespeicherte Daten zugreifen. Wenn wir Daten in einer Variablen abgespeichert haben, können wir unter dem Namen der Variablen immer wieder darauf zugreifen. In `R` funktioniert das mit der Zuweisung `<-`.

```
## Speichert einen Vektor in einer Variablen:
meinVektor <- c(1, 2, 6, 7, 10)
```

Ich kann den Inhalt von Variablen in der `R`-Konsole ausgeben lassen, wenn ich den Namen der Variablen in die Konsole schreibe und `Enter` drücke:

[6] Wir werden nur diese Fälle betrachten: Entweder wird ein ein-elementiger Vektor mit einem längeren Vektor verknüpft oder zwei gleich lange Vektoren werden miteinander verknüpft. Es ist auch möglich, andere Kombinationen von Vektorlängen zu paaren, was wir jedoch erst einmal vernachlässigen; interessierte Leser können die Befehle `c(1,2) * 1:4` und `c(1,2) * 1:3` in die `R`-Konsole eingeben und beobachten, was passiert.

```
meinVektor
```

```
[1]  1  2  6  7 10
```

Ich kann Variablen in Berechnungen verwenden:

```
meinVektor * 2
```

```
[1]  2  4 12 14 20
```

Ich kann Funktionen auf Variablen anwenden und das Ergebnis der Funktion wiederum in einer Variablen speichern:

```
xx <- mean(meinVektor)

## "Zentrierter" numerischer Vektor:
meinVektor - xx
```

```
[1] -4.2 -3.2  0.8  1.8  4.8
```

Variablen können an jeder Stelle verwendet werden, an der man Daten sonst „händisch“ eingeben würde. Wir können jegliche Objekte – nicht nur Vektoren, sondern auch Datentabellen oder beliebig komplizierte Ergebnisse von Berechnungen – in Variablen speichern. Der Workflow in R ist so ausgelegt, dass Zwischenergebnisse weiterverwendet werden können. Hierbei unterscheidet es sich fundamental von SPSS, das einen Unterschied zwischen Daten und „Output“ macht. In R kann das Ergebnis jeglicher Berechnung als Input einer anderen Berechnung dienen.

> **Merke**: In R kann (fast) alles in Variablen gespeichert und weiterverwendet werden.

Wir können auch mit einem Gleichzeichen = Daten zu Variablen zuweisen. Das funktioniert genauso wie mit <-:

```
foo = 1:2
foo
```

```
[1] 1 2
```

In R hat sich aus historischen Gründen die Konvention durchgesetzt, <- zu verwenden, die ich in diesem Skript auch befolgen werde. In vielen anderen Programmiersprachen werden Variablen mit Gleichzeichen zugewiesen.

2.1.1 Ausgabe versus Abspeichern

Wir haben jetzt zwei verschiedene Möglichkeiten kennengelernt, Objekte[7] in R zu verwenden:

[7] Bis jetzt kennen wir nur das Vektor-Objekt. In R gibt es aber ganz verschiedene „Datencontainer“, die allgemein als Objekte bezeichnet werden.

1. Wir geben Objekte in der Konsole aus.
2. Wir speichern Objekte in einer Variable ab.

Diese beiden Verwendungen sind **fundamental** unterschiedlich. Das mag erst einmal trivial erscheinen, aber ist im Einzelfall nicht unbedingt ersichtlich. Betrachten wir das folgende Beispiel:

```
bar <- c(3, 2, 6, 3, 9, 5, 7, -3)
sort(bar)
```

```
[1] -3  2  3  3  5  6  7  9
```

Die Funktion `sort` sortiert den numerischen Vektor `bar` aufsteigend. Wie sieht der Vektor `bar` nach der Operation aus? Es gibt zwei Möglichkeiten:

1. `bar` enthält den sortierten Vektor, den ich mithilfe von `sort(bar)` erstellt habe
2. `bar` enthält den unsortierten Vektor, den ich vor der Operation `sort(bar)` erstellt habe

Wir können die Frage leicht klären, indem wir `bar` in der Konsole ausgeben:

```
bar
```

```
[1]  3  2  6  3  9  5  7 -3
```

Offensichtlich hat `sort(bar)` den Vektor, der in der Variablen `bar` gespeichert ist, nicht geändert. Das ist eine fundamentale Eigenschaft der Programmiersprache `R`: **Funktionen nehmen Daten an und sie geben Daten zurück – sie verändern aber nicht die eingegebenen Daten**. Wenn wir wollen, dass `bar` die Zahlenfolge in sortierter Reihenfolge enthält, können wir die folgende Befehlkette verwenden:

```
bar <- c(3, 2, 6, 3, 9, 5, 7, -3)
bar <- sort(bar)
```

In diesem Fall geht der Ursprungsvektor verloren und wir behalten nur den sortieren Vektor. Generell gilt: wenn wir Daten in der Konsole ausgeben lassen, verschwinden diese sozusagen im „Nirvana“. Wenn wir mit Daten weiterarbeiten wollen, müssen wir die Ausgabe einer Funktion in einer Variablen speichern. Beide Verwendungszwecke sind denkbar: Manchmal benötige ich nur die Ausgabe einer Berechnung, manchmal möchte ich das Ergebnis abspeichern.

2.1.2 Variablennamen

Generell bestehen Variablennamen aus Buchstaben und Zahlen und den Zeichen `.` und `_`. Folgende Einschränkungen sind zu beachten:

- Variablennamen dürfen keine Leerzeichen enthalten
  - `bla bla <- c(1, 2)` funktioniert nicht
  - `blabla  <- c(1, 2)` funktioniert
- Variablennamen dürfen nicht mit einer Zahl starten
  - `1bla <- c(1, 2)` funktioniert nicht
  - `bla1 <- c(1, 2)` funktioniert

- Variablennamen dürfen keine Sonderzeichen außer `_` oder `.` enthalten
  - `bla-bla <- c(1, 2)` funktioniert nicht
  - `bla%bla <- c(1, 2)` funktioniert nicht
  - `bla_bla <- c(1, 2)` funktioniert
  - `bla.bla <- c(1, 2)` funktioniert
- `bla <- 1` ist nicht das Gleiche wie `Bla <- 1` oder gar `BLA <- 1`
- Vermeidet Umlaute in Variablennamen. `R` wird diese zwar akzeptieren, aber ich würde dennoch davon abraten, sie zu nutzen.

Eine fundamentale Schwierigkeit beim Programmieren ist das Finden *guter* Variablennamen; `bla` und `blabla` sind denkbar schlechte Variablennamen. Gute Variablennamen *sprechen*, d.h. sie machen eine Aussage darüber, was für Daten sie beinhalten.

```
## Schlechter Variablenname:
foo <- mean(age)

## Ggf. etwas besser:
mean_age <- mean(age)
```

Beachtet **immer** folgende Regel: Variablennamen sollten nicht lügen, also verwendet niemals einen Namen der folgenden Art:

```
mean_age <- sd(age) # Niemals machen!
```

Man ist schnell geneigt einen unsinnigen Variablennamen zu vergeben, um keine Zeit mit der Namensfindung zu verschwenden – man hat ja schließlich wichtigen Code zu schreiben! Man sollte sich jedoch so gut wie immer kurz Zeit nehmen, einen sinnigen Namen zu finden – das zukünftige Selbst wird es einem danken. Unsinnige Variablennamen sind in Ordnung, wenn man sich zu 100% sicher ist, dass man die Variable nach einmaliger Nutzung nicht mehr verwendet. Wenn man eine Variable nicht mehr benutzen möchte, kann man sie mit der `rm` Funktion löschen:

```
foo <- 1:10 # Wegwerfvariable
rm(foo)
foo
Fehler: Objekt 'foo' nicht gefunden
```

Weiterhin ist es guter Stil *konsistent* in der Vergebung der Variablennamen zu sein. Variablennamen sollen einen semantischen Gehalt haben, das heißt sie machen eine Aussage darüber, welche Daten sie enthalten. Häufig ist diese Information nicht in einem Wort erklärbar. Um auszusagen, dass eine Variable „das mittlere Alter“ enthält, müssen mindestens die Anteile „mittel“ und „Alter“ enthalten sein. Wie soll das verknüpft werden? Verschiedene Konventionen existieren; wichtig ist, dass ihr euch konsistent für eine Variante entscheidet.[8]

```
## Mögliche Konventionen der Namensgebung von Variablen:
mean_age <- mean(age)
```

[8] Ich werde von dieser Regel in diesem Skript abweichen.

```
mean.age <- mean(age)
meanAge  <- mean(age)

## keine gute Konvention:
meanage  <- mean(age)
```

2.2 Datentypen von Vektoren

In R hat jeder Vektor genau einen Datentyp. Bis jetzt haben wir nur mit dem Datentyp Zahl gearbeitet, der in R „numeric“ heißt. Der Datentyp eines Vektors bestimmt, was für Operationen wir damit durchführen können. Vektoren vom Typ `numeric` etwa kann man addieren, multiplizieren und so weiter. Mit der Funktion `mode` können wir überprüfen, welchen Datentyp ein Vektor hat:

```
mode(1:10)
```

```
[1] "numeric"
```

In diesem Abschnitt werden weitere Datentypen behandelt, die wir nutzen, um unterschiedliche Informationen darzustellen.

2.2.1 `character`

Der Datentyp für Text heißt `character`. Text wird mit doppelten oder einfachen Anführungszeichen angegeben:

```
"Hallo Welt!" # doppelte Anführungszeichen
```

```
[1] "Hallo Welt!"
```

```
mein_text <- 'bla bla bla' # einfache Anführungszeichen

## zwei-elementiger Vektor vom Typ character:
mein_text2 <- c("Cronbachs", "Alpha")
```

Mit Texten können wir andere Operationen durchführen als mit Zahlen, etwa ergibt Folgendes eine Fehlermeldung[9] und ergibt auch gar keinen Sinn, da man Text nicht mit einer Zahl multiplizieren kann:

```
"bla" * 2
Fehler in "bla" * 2 : nicht-numerisches Argument
für binären Operator
```

In diesem Skript spielt Text keine allzu große Rolle. In erster Linie werden wir Vektoren vom Typ `character` für Datenzugriffe verwenden; im Speziellen werden wir sie einsetzen, um

[9] Leider sind Fehlermeldungen in R oftmals sehr kryptisch und gerade für Anfänger schwer verständlich.

Spalten in Datentabellen zu adressieren (siehe Kapitel 3). Zu diesem Zweck werden wir die Funktion `paste0` nutzen, die `character`-Vektoren beliebiger Länge erstellt. So lassen sich beispielsweise bequem 10 durchnummerierte Itemnamen generieren, wodurch man gleich 10 Spalten aus einer Tabelle auswählen könnte:

```
items <- paste0("item_", 1:10)
```

Hierbei wird der Text "item_" mit den Zahlen von 1 bis 10 gepaart. Das Ergebnis des Befehls ist ein 10-elementiger Vektor, was wir auch wie folgt überprüfen können:

```
length(items)
```

```
[1] 10
```

```
items
```

```
 [1] "item_1"  "item_2"  "item_3"  "item_4"  "item_5"  "item_6"  "item_7"
 [8] "item_8"  "item_9"  "item_10"
```

Wenn man mit der Funktion `paste0` mehrere ein-elementige Vektoren miteinander verknüpft, wird ein ein-elementiger Vektor vom Typ `character` ausgegeben:

```
paste0("item", "_", 1) # paste0 nimmt beliebig viele Argumente an
```

```
[1] "item_1"
```

2.2.2 logical

Es hat sich als nützlich erwiesen, einen Datentyp einzuführen, der "Wahrheit" kodiert. Dieser Datentyp wird in `R` "logical" genannt; er kennt nur die Ausprägungen `TRUE` und `FALSE`. Eine sonst gängige Bezeichnung für diesen Datentyp ist auch "boolean".

```
wahr   <- TRUE
falsch <- FALSE
```

Wir werden häufig vom Typ `logical` Gebrauch machen, wenn wir in Datentabellen Fälle auswählen (etwa alle weiblichen oder männlichen Teilnehmer in einer Umfrage).

Mit logischen Werten kann man die logischen Operationen UND (in `R`: `&`), ODER (in `R`: `|`) und NICHT (in `R`: `!`) durchführen:

```
## Logisches UND
TRUE & TRUE
```

```
[1] TRUE
```

```
TRUE & FALSE
```

```
[1] FALSE
```

```
FALSE & FALSE
```

```
[1] FALSE
```

```
## Logisches ODER
TRUE | TRUE
```

```
[1] TRUE
```

```
TRUE | FALSE
```

```
[1] TRUE
```

```
FALSE | FALSE
```

```
[1] FALSE
```

```
## Logisches NICHT
!TRUE
```

```
[1] FALSE
```

```
!FALSE
```

```
[1] TRUE
```

Die logischen Operationen UND und ODER verknüpfen zwei logische Elemente miteinander (logische Elemente = `TRUE`/`FALSE`). Die ODER-Operation ergibt `TRUE`, sobald mindestens eines der Elemente `TRUE` ist. Das logische UND ergibt nur dann `TRUE`, wenn beide Elemente `TRUE` sind. Das logische NICHT invertiert die Eingabe: Aus `TRUE` wird `FALSE` und umgekehrt.

Die logischen Operationen UND und ODER arbeiten komponentenweise auf Vektoren, die mehr als ein Element enthalten:

```
c(TRUE, FALSE, FALSE) & c(TRUE, TRUE, FALSE)
```

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

```
c(TRUE, FALSE, FALSE) | c(TRUE, TRUE, FALSE)
```

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

Auch das logische NICHT arbeitet vektorisiert. Es kann auf einen logischen Vektor angewendet werden, der beliebig viele Elemente enthält und kehrt alle Elemente darin um:

```
!c(TRUE, FALSE)
```

```
[1] FALSE  TRUE
```

Während die Operationen UND, ODER und NICHT an dieser Stelle nur abstrakt eingeführt werden, werden wir in weiteren Abschnitten (Kapitel 2 und Kapitel 3) noch lernen, wie wir logische Bedingungen verwenden, um gezielt Daten mit bestimmten Eigenschaften auszuwählen.

2.2.3 `factor`

Vektoren vom Typ `factor` stellen kategoriale Variablen dar – etwa die unabhängigen Variablen in einer ANOVA. So können wir einen Vektor vom Typ `factor` erstellen:

```
laune <- c(1, 2, 3, 1, 2, 1)

laune_faktor <- factor(laune, levels = c(1, 2, 3),
                       labels = c(":(", ":)", ":D"))

laune_faktor
```

```
[1] :( :) :D :( :) :(
Levels: :( :) :D
```

Die Funktion `factor` wird hier genutzt, um numerische Werte in `factor` umzuwandeln. Dabei wurden `levels` spezifiziert, d.h. die Werte, die der ursprüngliche Vektor angenommen hat. Das Argument `labels` ordnet den `levels` eine Bezeichnung zu; diese wird uns angezeigt, wenn wir den Vektor aufrufen. Mit einem Vektor vom Typ `factor` kann ich keine numerischen Berechnungen mehr durchführen, da er **kategoriale** Daten beinhaltet. Etwa kann ich für `laune_faktor` keinen Mittelwert berechnen:

```
mean(laune)
```

```
[1] 1.666667
```

```
mean(laune_faktor)
```

```
[1] NA
```

Da die Berechnung nicht möglich ist, gibt `R` die folgende "Warnmeldung" aus:

```
Warnmeldung:
In mean.default(laune_faktor) :
  argument is not numeric or logical: returning NA
```

2.2.4 `NA`

`R` hat einen eigenen Datentyp, um fehlende Werte zu kodieren: `NA`.[10] Da wir mit echten Datensätzen arbeiten, die oftmals „messy" sind, d.h. nicht notwendigerweise vollständig, ist diese Eigenschaft sehr nützlich. Gerade bei der Arbeit mit Daten in der psychologischen

[10] Eigentlich ist `NA` kein eigener Datentyp. In `R` hat jeder Vektor **nur genau einen** Datentyp. Es ist beispielsweise nicht möglich, dass in einem Vektor gleichzeitig Werte vom Typ `numeric`, `character` und `factor` vorkommen. `NA`-Werte können jedoch in Kombination mit jedem Datentyp vorkommen. Sie kodieren dann die Abwesenheit eines Datums; dieses Datum hätte – wenn es nicht fehlen würde – den Datentyp des Vektors.

Diagnostik ist dies wichtig: Menschen geben in Fragebögen eben nicht immer auf alle Fragen eine Antwort.

Man kann selber Vektoren erstellen, die fehlende Werte enthalten:

```
messy_data <- c(1, 3, 2, 9, 3, NA, 6, NA, 5)
```

Die Anwesenheit von fehlenden Werten hat Auswirkungen darauf, welche Berechnungen R mit dem Vektor anstellen kann. Etwa können wir nicht mehr ohne Weiteres einen Mittelwert berechnen:

```
mean(messy_data) # geht nicht wegen des fehlenden Werts
```

```
[1] NA
```

Man muss R explizit mitteilen, dass man trotz des Auftretens fehlender Werte einen Mittelwert ausrechnen möchte. Dies funktioniert mit dem *optionalen Argument* `na.rm`[11] der Funktion `mean`, welches wir auf `TRUE` setzen können. Mit dem *Argument* `na.rm` ("NA remove") teilt man `mean` mit, dass `NA` Werte bei der Berechnung des Mittelwerts nicht berücksichtigt werden sollen (andere Funktionen wie `sd` und `var` haben auch das Argument `na.rm`):

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

```
[1] 4.142857
```

Hierbei nehmen wir zur Kenntnis, dass man Argumente von Funktionen benennen kann – was wir aber nicht immer machen. Dazu später mehr.

2.3 Logische Vergleiche

Wir können in R Eigenschaften von Vektoren mithilfe von logischen Vergleichen erfragen. So kann man beispielsweise prüfen, welche Werte eines numerischen Vektors (a) gleich, (b) größer (c) kleiner, (d) größer gleich, (e) kleiner gleich oder (f) ungleich einem bestimmten Wert sind. Dieser Code-Abschnitt stellt die grundlegenden logischen Vergleiche dar:

```
vergleichswert <- 3
daten <- 1:5
daten > vergleichswert
```

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

```
daten < vergleichswert
```

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

[11] Ein Argument heißt optional, wenn wir dafür keinen Wert angeben müssen. Stattdessen hat es einen sogenannten Standardwert, der angenommen wird, wenn wir das Argument nicht selber angeben. Der Standardwert des Arguments `na.rm` in der Funktion `mean` ist `FALSE`.

```
daten >= vergleichswert
```

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

```
daten <= vergleichswert
```

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

```
daten == vergleichswert
```

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

```
daten != vergleichswert
```

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

Das Ergebnis dieser Operationen ist ein Vektor aus `TRUE` und `FALSE` Werten. Die Werte nehmen `TRUE` an, wenn die Zahlen die kleiner/größer/gleich Bedingung erfüllen – andernfalls `FALSE`. **Beachtet, dass auf Gleichheit mit dem "doppelten" == Operator getestet wird und nicht mit einem einfachen =.** Dies ist eine häufige Quelle von Fehlern, die schwierig zu entdecken sind. Betrachtet etwa folgenden Code – was geht hier schief?

```
daten = vergleichswert
```

Hierbei wird die Variable `daten` mit dem Wert in der Variablen `vergleichswert` überschrieben, da `=` als Zuweisung agiert:

```
daten
```

```
[1] 3
```

Dies ist ein Beispiel für einen Fehler (*Bug*), den man nicht anhand von einer Fehlermeldung bemerkt, da der Befehl *syntaktisch* korrekt ist. Es ist jedoch problematisch, dass ich an dieser Stelle meine Daten mit einem irrelevanten Wert überschrieben habe, und das bei einem späteren Zugriff darauf vermutlich nicht beachten werde.

Welche logischen Vergleiche möglich sind, hängt vom Datentyp eines Vektors ab. Für Vektoren vom typ `character` etwa macht eine kleiner/größer Abfrage keinen Sinn, jedoch eine Abfrage auf Gleichheit.

```
text1 <- "Hallo Welt"
text1 == "Hallo Welt"
```

```
[1] TRUE
```

```
text1 == "Hallo Welt!"
```

```
[1] FALSE
```

Für Vektoren vom Typ `factor` lässt sich genauso eine Abfrage auf Gleichheit umsetzen. Hier wird beim Test auf Gleichheit das überprüfte `factor`-Label in Anführungszeichen gesetzt:

```
geschlecht <- c(1, 2, 1, 1, 2, 1, 3)
geschlecht <- factor(geschlecht, levels = 1:3,
                     labels = c("weiblich", "maennlich", "divers"))

geschlecht == "maennlich"
```

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

Wenn zwei Vektoren gleicher Länge mit logischen Operatoren verglichen werden, werden die Elemente komponentenweise verglichen:

```
score_test1 <- c(23, 19, 44, 18, 25, 22)
score_test2 <- c(26, 23, 29, 18, 32, 19)

score_test1 > score_test2
```

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

```
score_test1 == score_test2
```

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

2.3.1 Anwendungsbeispiel: Überprüfe das Gesetz der großen Zahlen

Häufig verwendet man die Vergleichsoperatoren, um zu prüfen, wie viele Daten eine bestimmte Eigenschaft erfüllen. Dafür verknüpfen wir die Vergleichsoperatoren mit den Funktionen `sum` oder `mean`.

Dafür bietet sich ein Beispiel aus der Statistik an: Wie viele von 1,000 Zufallsdaten aus einer Standardnormalverteilung sind größer als 1? `R` hat zahlreiche Funktionen, um Zufallszahlen aus verschiedenen Verteilungen zu generieren. Mit `rnorm` lassen sich Zufallszahlen generieren, die einer Normalverteilungen folgen; wenn man keine weiteren Argumente angibt, ist die Standardnormalverteilung gemeint, die einen Mittelwert von `0` und eine Standardabweichung von `1` hat:

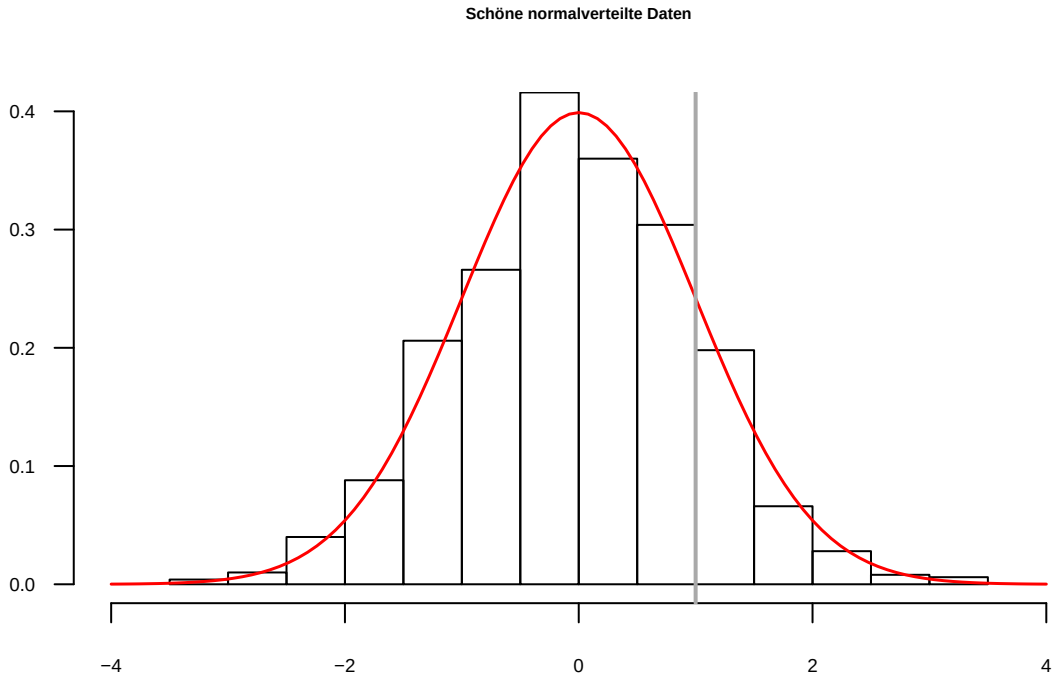
```
## Erstelle 1,000 Zufallsdaten:
zufallsdaten <- rnorm(1000)
```

Zur Verdeutlichung: Der Vektor `zufallsdaten` enthält jetzt 1,000 Elemente, wie wir mit der Funktion `length` leicht überprüfen können:

```
length(zufallsdaten)
```

```
[1] 1000
```

Die Funktion `head` zeigt uns die ersten sechs Werte des Vektors an. `head` ist sehr praktisch, um sich schnell einen Blick über Daten zu verschaffen. Das machen wir hier auch, da wir nicht alle 1,000 Werte in die Konsole schreiben wollen:




```
head(zufallsdaten)
```

```
[1]  1.3709584 -0.5646982  0.3631284  0.6328626  0.4042683 -0.1061245
```

Wir können die Daten mithilfe eines Histogramms betrachten, um uns davon zu überzeugen, dass sie tatsächlich normalverteilt sind – sich also der Großteil der Daten um die `0` tummelt und extreme Werte in beide Richtungen seltener werden (dieser Code muss nicht verstanden werden):

```
## Male Histogram
hist(zufallsdaten, freq = FALSE,
     main = "Schöne normalverteilte Daten",
     xlab = "", ylab = "", las = 1,
     xlim = c(-4, 4), ylim = c(0, 0.4),
     cex.main = 0.5, cex.axis = 0.6)

## Lege eine Normalverteilungskurve über die Daten
curve(dnorm, col = "red", add = TRUE, lwd = 1.5)

## Zeichne eine graue Linie beim x-Wert `1` ein:
abline(v = 1, lwd = 2, col = "darkgrey")
```

Nach visueller Inspektion der Verteilung der Zufallszahlen können wir mit `sum` testen, wie viele der 1,000 Zufallsdaten größer als 1 sind:

```
sum(zufallsdaten > 1)
```

```
[1] 153
```

Zur Erinnerung: Der Befehl "`zufallsdaten > 1`" ergibt einen Vektor aus `TRUE` und `FALSE` Werten, der genauso viele Elemente enthält wie der Vektor `zufallsdaten`; wann immer ein Eintrag in `zufallsdaten` größer ist als 1, erhalten wir `TRUE`, andernfalls `FALSE`. `sum` gibt die Zahl der `TRUE` Einträge aus. **Das funktioniert, da `TRUE` und `FALSE` eine numerische Interpretation haben: `TRUE` wird als `1` interpretiert und `FALSE` als `0`.**[12]

Analog können wir mit `mean` den relativen Anteil der Daten bestimmen, die größer als 1 sind:

```
mean(zufallsdaten > 1)
```

```
[1] 0.153
```

Der Aufruf `mean(zufallsdaten > 1)` ergibt also den relativen Anteil der Datenpunkte, die größer sind als 1. Es lohnt sich ein wenig darüber nachzudenken, warum wir die Funktion `mean` hier verwenden können, um einen relativen Anteil zu bestimmen. Normalerweise sind wir es eher gewohnt, dass Mittelwerte und relative Anteile etwas Unterschiedliches sind. Der Grund dafür ist folgender: Für eine Variable, bei der die Werte nur 1 oder 0 annehmen können (analog in R: `TRUE`/`FALSE`), entspricht der Mittelwert dieser Variablen dem relativen Anteil der Werte, die 1 bzw. `TRUE` sind. Als Beispiel betrachten wir den folgenden Vektor:

```
beispiel_01 <- c(1, 0, 1, 1)
mean(beispiel_01)
```

```
[1] 0.75
```

Der Mittelwert des Vektors ist `(1 + 0 + 1 + 1) / 4`, also 0.75 – und damit genau der relative Anteil der 1-Elemente. Dies ist eine praktische Eigenschaft der 1/0- bzw. `TRUE`/`FALSE`-Kodierung und macht logische Abfragen in `R` so mächtig.

Wenn wir einen relativen Anteil in eine Prozentzahl umwandeln wollen, können wir den relativen Anteil einfach mit 100 multiplizieren:

```
mean(zufallsdaten > 1) * 100
```

```
[1] 15.3
```

Der Erwartungswert, dass eine zufällige Zahl aus einer Standardnormalverteilung größer ist als `1` – also mehr als eine Standardabweichung vom Mittelwert entfernt liegt – liegt bei etwa 15.9%. Den exakten Erwartungswert könnte ich in `R` mit der Funktion `pnorm` herausfinden:[13]

```
1 - pnorm(1)
```

```
[1] 0.1586553
```

[12] Wenn logische Vektoren einer numerischen Berechnung übergeben werden, werden die `TRUE`/`FALSE` Elemente des Vektors automatisch in Zahlen, d.h. 1 und 0 umgewandelt. Deswegen funktioniert beispielsweise auch folgender Befehl: `TRUE + 1`

[13] `pnorm` ist die kummulative Verteilungsfunktion der Normalverteilung. Sie sagt aus, wie viel % der Werte in einer Normalverteilung kleiner sind als der übergebene Wert. Um heraus zu finden, wie viele Werte **größer** als 1 sind, wird hier das Komplement, also `1 - pnorm(1)`, gebildet. Das funktioniert, da die Gesamtdichte einer Wahrscheinlichkeitsverteilung immer 1 ist.

Nach dem Gesetz der großen Zahlen liegt der folgende Wert wahrscheinlich näher an 15.9% als der Schätzer, der auf 1,000 Zufallszahlen basiert:

```
## 100,000 Zufallsdaten sind für R kein Problem
zufallsdaten <- rnorm(100000) # 100000
mean(zufallsdaten > 1)
```

```
[1] 0.15838
```

Ihr könnt für das Gesetz der großen Zahlen selber ein Gefühl entwickeln, wenn ihr mehrfach `mean(rnorm(1000)>1)` und `mean(rnorm(100000)>1)` in die R-Konsole eingebt und beobachtet, welcher Wert häufiger näher an 0.159 liegt. Beachtet wie schnell R Operationen mit 100,000 Zahlen durchführen kann.

2.3.2 Der %in% Operator

Um zu testen, ob ein oder mehrere Elemente in einem Vektor enthalten sind, kann man den %in%-Operator verwenden. Der sieht zwar gewöhnungsbedürftig aus, ist aber einfach zu verwenden und hat auch eine einfache verbale Interpretation: Sind die Elemente aus Vektor A in Vektor B?

```
2 %in% 1:3
```

```
[1] TRUE
```

```
4 %in% 1:3
```

```
[1] FALSE
```

Der %in%-Operator testet für jedes der Elemente *vor* %in%, ob dieses im Vektor *nach* %in% enthalten ist:

```
c(2, 3) %in% 3:5
```

```
[1] FALSE  TRUE
```

Die Ausgabe der %in%-Operation ist also ein logischer Vektor; die Länge des Ausgabevektors entspricht dabei immer der Länge des Vektors auf der linken Seite von %in%.

2.4 Zugriff auf Vektorelemente

Der Zugriff auf Daten ist ein wichtiger Abschnitt unserer Einleitung in die Grundlagen Rs. In diesem Abschnitt lernen wir, wie wir Elemente aus einfachen Vektoren „herausgreifen“ können.

2.4.1 Der `[·]`-Zugriff

Daten können mit dem `[·]`-Zugriff[14] *indexbasiert* aus Vektoren ausgewählt werden. Jedes Element im Vektor hat einen *Index*, der seiner Position im Vektor entspricht. Im folgenden Vektor etwa hat `2` den Index `1`, `4` den Index `2` und `1` den Index `3`:

```
daten <- c(2, 4, 1)
```

Ich kann mit dem `[·]`-Zugriff durch Angabe des Index auf einzelne Elemente im Vektor zugreifen:

```
daten[1]
```

```
[1] 2
```

```
xx <- daten[3] # ein-elementiger Vektor
xx
```

```
[1] 1
```

Ebenso kann ich einen „Negativ"-Zugriff durchführen: Ich kann auswählen, welchen Index ich *nicht* in meinem Ergebnis haben will:

```
daten[-1]
```

```
[1] 4 1
```

Interessant wird diese Art des Zugriffs, da der Index in den `[·]` Klammern auch ein mehrelementiger numerischer Vektor sein kann – hier nutzen wir die `c` Funktion:

```
daten[c(1, 2)]
```

```
[1] 2 4
```

```
daten[-c(2, 3)]
```

```
[1] 2
```

2.4.2 `[·]`-Zugriff mit einem logischen Vektor

Anstatt direkt den Index eines Elements zu übergeben – den wir häufig nicht wissen, da wir bei vielen Daten nicht den Überblick über die Position aller einzelnen Datenpunkte behalten – möchten wir häufig Daten auswählen, die eine bestimmte Eigenschaft erfüllen. Hierbei machen wir uns die logischen Operationen zunutze, die wir oben kennengelernt haben:

[14] Ich nenne diese Operation `[·]`-Zugriff , da zur Datenauswahl aus Vektoren hinter den Vektor eckigen Klammern gestellt werden. Die Klammern enthalten eine Angabe darüber, welche Elemente ich aus dem Vektor auswählen will. Etwa wählt `c(4, 2, 6)[1]` das erste Element aus dem Vektor `c(4, 2, 6)` aus, also 4. Der Punkt ist bloß ein Platzhalter in der `[·]`-Notation.

```
meinVektor <- c(1, 2, 3, 7, 8, 9)

auswahl <- meinVektor > 5
auswahl
```

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

`auswahl` ist ein logischer Vektor, der kodiert, welche Elemente des Vektors `meinVektor` größer als 5 sind (spezifisch: an welchen Positionen ist in `meinVektor` ein Element enthalten, das größer ist als 5). Ich kann nun den `[·]`-Zugriff mithilfe von `auswahl` verwenden, um nur die Elemente auszuwählen, die größer sind als 5:

```
meinVektor[auswahl]
```

```
[1] 7 8 9
```

Hierbei wurden die Werte 7, 8 und 9 ausgewählt, da für diese Werte der Vektor `auswahl` auf `TRUE` steht. Genauer gesagt: `auswahl` steht für die Indexe 4, 5 und 6 auf `TRUE` und es gilt `meinVektor[4] == 7`, `meinVektor[5] == 8`, und `meinVektor[6] == 9`.

Man kann dieses Vorgehen sogar mit den UND/ODER-Operationen verknüpfen, um Daten anhand verschiedener Kriterien auszuwählen:

```
meinVektor <- 1:20

auswahl <- (meinVektor < 5) | (meinVektor > 17)
auswahl
```

```
 [1]  TRUE  TRUE  TRUE  TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[12] FALSE FALSE FALSE FALSE FALSE FALSE  TRUE  TRUE  TRUE
```

```
meinVektor[auswahl]
```

```
[1]  1  2  3  4 18 19 20
```

Hier ein weiteres Beispiel mit normalverteilten Zufallsdaten:

```
## Wähle alle Daten aus, die größer sind als 2 (das sollten im Schnitt
## etwa 2.5% der Daten sein)
daten <- rnorm(300)
daten[daten > 2]
```

```
[1] 3.196362 2.386513 2.791331 2.123950
```

An dieser Stelle sollte man sich klar machen, warum `daten` sowohl vor als auch innerhalb der `[·]` Klammern vorkommt. Das ist prinzipiell dasselbe wie im Beispiel `meinVektor[auswahl]` oben, nur das ich dort den `TRUE/FALSE` Vektor, der die Daten ausgewählt hat, in einer Variablen – `auswahl` – zwischengespeichert habe.

2.4.3 [·]-Zugriff zum Ändern von Daten

Wir sind mit dem [·]-Zugriff nicht darauf beschränkt Elemente aus Vektoren auszulesen, sondern wir können auf diese Weise auch einzelne Elemente im Vektor verändern:

```
daten <- 1:5
daten[c(2, 5)] <- 0
daten
```

```
[1] 1 0 3 4 0
```

Dies geht wiederum auch mit einem logischen Vektor in den [·]-Klammern, wie das folgende Beispiel zeigt:

```
daten <- 1:5
daten[c(TRUE, FALSE, TRUE, FALSE, FALSE)] <- 0
daten
```

```
[1] 0 2 0 4 5
```

Das würde man so "händisch" nicht machen, aber es soll zum Verständnis dessen dienen, was im folgenden – anwendungsnäheren – Beispiel passiert. Angenommen, bei einer Dateneingabe wurden fehlende Werte in einem Fragebogen mit -99 kodiert.[15] Wir wollen R mitteilen, diesen Wert als fehlend zu interpretieren. Hier kommt uns wiederum eine logische Abfrage zugute:

```
daten <- c(1, -99, 5, -99, 2, -99, 4, 1:3)
daten
```

```
 [1]   1 -99   5 -99   2 -99   4   1   2   3
```

```
missing_values <- daten == -99
missing_values
```

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

Die Variable missing_values kodiert jetzt, an welchen Positionen des Vektors daten sich eine -99 befindet. Wir können diese Werte nun wie folgt durch NA ersetzen:

```
daten[missing_values] <- NA
daten
```

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

Semantisch ist dieser Vorgang gut zu verstehen: Setze alle Werte, die einen fehlenden Wert enthalten – d.h. mit -99 kodiert wurden – auf NA, damit R für weitere Berechnungen weiß, dass diese Werte als fehlend zu verstehen sind. Technisch umgesetzt wird dies mit einem TRUE/FALSE Vektor, den wir mithilfe der Anweisung daten == -99 erstellt haben.

[15] Das macht beispielsweise Sinn, damit bei der Eingabe explizit gemacht wird, dass der Wert fehlt. Andernfalls könnte das Datum bei der Eingabe auch vergessen worden sein.

Wir werden wohl selten "händisch" per Index oder logischem `TRUE`/`FALSE` Vektor eine Auswahl/Änderung von Daten durchführen. Aber in Zusammenarbeit mit den logischen Operatoren (`>`, `<`, `==`, `&`, `|` etc.) ist die Auswahl von Elementen aus Vektoren – und auch die Auswahl von Daten aus Tabellen – eine häufige Anwendung. Diese werden wir bei der gezielten Auswahl von Zeilen aus Datentabellen (siehe Kapitel 3) wiederfinden und uns zunutze machen. Das gegebene Beispiel zum Umkodieren von fehlenden Werten werden wir in einer sehr ähnlichen Form umsetzen, da wir sonst die Daten des Narcissistic Personality Inventory nicht auswerten können. Bevor die Analyse starten kann, müssen fehlende Werte gekennzeichnet werden.

2.5 Präzedenz

Durch Klammerung können wir die *Präzedenz* von `R`-Befehlen steuern. Präzendenz bezieht sich auf die Reihenfolge, in der `R`-Befehle ausgeführt werden. Betrachten wir das folgende Beispiel:

```
TRUE | TRUE & FALSE
```

```
[1] TRUE
```

Die Ausgabe ist `TRUE`. Daraus können wir schlussfolgern, dass die Befehle ODER und UND **nicht** von links nach rechts ausgeführt wurden. In dem Fall wäre nämlich zunächst `TRUE | TRUE` ausgeführt worden, was `TRUE` ergibt. Dieses Ergebnis (also `TRUE`) wäre dann per UND mit `FALSE` verknüpft worden, was insgesamt `FALSE` ausgegeben hätte. Wir haben aber `TRUE` bekommen. Warum?

Der Grund: **Die UND-Operation hat eine höhere Präzedenz als die ODER-Operation**. Wenn UND und ODER in einem logischen Ausdruck verbunden werden, wird zunächst die UND und dann die ODER-Operation ausgeführt, unabhängig davon, in welcher Reihenfolge wir die Befehle aufschreiben. Möchten wir erzwingen, dass die ODER-Operation zuerst durchgeführt wird, können wir – ganz analog zu mathematischen Berechnungen – Klammern verwenden:

```
(TRUE | TRUE) & FALSE
```

```
[1] FALSE
```

Die Präzendenzregeln gelten ebenfalls, wenn die logischen Vektoren aus mehr als einem Element bestehen. Betrachten wir dazu die folgenden Beispiele:

```
c(FALSE, TRUE) | c(FALSE, FALSE) & c(TRUE, FALSE)
```

```
[1] FALSE  TRUE
```

```
(c(FALSE, TRUE) | c(FALSE, FALSE)) & c(TRUE, FALSE)
```

```
[1] FALSE FALSE
```

Wir werden logische Ausdrücke vor allem zur Fallauswahl in Datentabellen verwenden (siehe Kapitel 3). Dann kann es sehr wichtig sein, auf korrekte Klammerung zu achten. Andernfalls

besteht die Gefahr, dass wir nicht genau die Fälle auswählen, die wir eigentlich auswählen wollen.

Betrachten wir ein weiteres Beispiel zur Steuerung von Präzedenz: Nehmen wir an, wir benötigen eine Sequenz aller Zahlen zwischen 1 und 10 – außer der 8.[16] Ein naheliegender Befehl wäre folgender:

```
1:10[-8]
```

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

Das hat aber nicht funktioniert, die Acht ist in der Ausgabe enthalten. Woran liegt das? **Die Auswahl per eckiger Klammer hat eine höhere Präzedenz als der Doppelpunktoperator.** Die Klammerungsoperation zur Auswahl aus einem Vektor wurde also nicht auf den Vektor `1:10`, sondern auf den Vektor `10` angewendet (Erinnerung: Einzelne Zahlen sind Vektoren). Das heißt, in diesem Beispiel wurde als Erstes das achte Element aus dem Vektor `10` ausgeschlossen, das aber gar nicht existiert. Stattdessen erhalten wir einfach wieder `10`:

```
10[-8]
```

```
[1] 10
```

Durch Klammerung können wir das gewünschte Ergebnis erhalten:

```
(1:10)[-8]
```

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

> **Merke**: Im Zweifel verwenden wir Klammern lieber einmal zu viel als einmal zu wenig.

[16] Das Beispiel mag hier künstlich wirken, aber genau so etwas werden wir in Kapitel 4 machen.

2.6 Zusammenfassung

- Wir haben Rs grundlegendste Datenstruktur, den Vektor, kennengelernt
- Vektoren enthalten beliebig viele Elemente gleichartiger Daten, etwa
  - Zahlen ("numeric")
  - Texte ("character")
  - Kategorielle Daten ("factor")
  - `TRUE`/`FALSE` ("logical")
- Mit dem `[·]`-Zugriff kann man Elemente aus Vektoren auswählen
  a. indem man die Position der Elemente angibt, die man auswählen will ("Positivauswahl")
  b. indem man die Position der Elemente angibt, die man **nicht** auswählen will ("Negativauswahl")
  c. indem man einen `TRUE`/`FALSE` Vektor angibt
- Man kann mit logischen Vergleichen die Eigenschaften von Vektoren überprüfen
  - diese Operation lässt sich gut mit der `[·]`-Auswahl verbinden

2.7 Fragen zum vertiefenden Verständnis

1. Wie berechnet man den Standardfehler von `1:10`?
2. Was für Objekte nimmt die Funktion `c` entgegen, und was gibt sie zurück?
3. Was ergibt `1:6 + 1:2`? Was passiert? Warum gibt `1:4 + 1:3` eine Warnmeldung aus?
4. Nutzt `paste0`, den `:`-Operator und den `[·]`-Negativ-Zugriff, um den folgenden Vektor zu erstellen:

```
[1] "item_2"  "item_4"  "item_5"  "item_6"  "item_7"  "item_8"  "item_10"
```

5. In `R` haben Elemente eines Vektors nur einen Datentyp. Der Befehl `c(1, 'moep')` vermischt eine Zahl und einen Text miteinander, aber ergibt keinen Fehler – was ist passiert?
6. Was sind plausible Ergebnisse von `sum(rnorm(100) > 1.645)`? (Erst überlegen, dann mehrfach in der `R`-Konsole ausführen!)
7. Was sind die Ausgaben von `mode(2)` und `mode(mode(2))`. Warum?
8. Was ist der Unterschied zwischen `sum(c(TRUE, FALSE, TRUE))` und `length(c(TRUE, FALSE, TRUE))`?

3 data.frames

Wir haben gelernt, dass R Daten in Vektoren abspeichert. Im Normalfall haben wir in der psychometrischen Datenauswertung aber eine große Datenmenge vorliegen, die wir nicht sinnvoll als einzelnen Vektor darstellen können. Etwa: 150 Studierende bearbeiten in einer Diagnostikklausur 42 Multiple-Choice-Klausuritems. Wir stellen solche Daten in Tabellen dar, wie man sie auch aus Excel oder SPSS kennt. In diesen Tabellen repräsentieren Spalten Messvariablen, etwa die Punktzahlen in einer Klausuraufgabe. Zeilen stellen Fälle dar, etwa Personen, die an der Klausur teilgenommen haben. Andere Datenformate wären auch denkbar, etwa eines bei dem jede Zeile einer Aufgabe entspricht. Bei uns wird aber gelten: Jede Zeile entspricht genau einer Person.

In R speichern wir Datentabellen in `data.frames` ab. Ein `data.frame` ist, vereinfacht gesagt, eine Sammlung von Vektoren gleicher Länge. Jede Spalte – also jede Messvariable – ist ein Vektor. Mit dieser Datenstruktur werden wir uns im vorliegenden Kapitel auseinandersetzen.

3.1 Die Funktion `data.frame`

Mit der Funktion `data.frame` können wir "händisch" einen `data.frame` erstellen. In der Praxis werden wir das eher selten machen und stattdessen Daten aus einer externen Datei einlesen.[17] Die folgende unscheinbare Tabelle mit 5 Einträgen erstelle ich mit der Funktion `data.frame`; sie wird uns durch einen Großteil des Kapitels begleiten, um Grundlagen von `data.frame`-Operationen zu betrachten.

```
mdf <- data.frame(Nummer = 1:5,
                  Item1 = c(1, 0, 0, 1, 1),
                  Item2 = c(1, 1, 0, 0, 1),
                  Alter = c(13, 14, 13, 12, 15),
                  Geschlecht = c("w", "m", "m", "w", "m")
                  )
mdf
```

```
  Nummer Item1 Item2 Alter Geschlecht
1      1     1     1    13          w
2      2     0     1    14          m
3      3     0     0    13          m
4      4     1     0    12          w
5      5     1     1    15          m
```

Der `data.frame` wird in in einer Variablen mit dem Namen `mdf` – was beispielsweise für "mein `data.frame`" stehen könnte – abgespeichert. Bei der Erstellung des `data.frames` wurde jede Spalte mit der Funktion `c` oder dem Doppelpunktoperator mit genau einem Vektor befüllt. Die Spalten des `data.frames` wurden bei der Erstellung benannt. **Dieser Punkt ist sehr**

[17] Beispielsweise können die Daten in einem *Spreadsheet-Editor* wie Excel eingegeben worden sein und wir importieren diese dann in R.

wichtig, da wir Spalten anhand ihrer Namen gezielt auswählen können. Wenn ich die Spaltennamen eines `data.frames` nicht mehr weiß, kann ich sie mit der Funktion `names` abrufen:

```
names(mdf)
```

```
[1] "Nummer"     "Item1"      "Item2"      "Alter"      "Geschlecht"
```

3.2 Zugriff auf eine einzelne Spalte: die $-Notation

Der $-Zugriff ist die grundlegendste Operation auf `data.frames`. Wir nutzen ihn, um auf einzelne Spalten eines `data.frames` zuzugreifen und diese **als Vektor** auszulesen:

```
punkte <- mdf$Item1
punkte # `punkte` ist ein Vektor
```

```
[1] 1 0 0 1 1
```

Ich kann den $-Zugriff nicht nur verwenden, um eine Spalte aus einem `data.frame` auszulesen, sondern kann damit auch neue Spalten hinzufügen. Das funktioniert, indem ich der neu zu erstellenden Spalte per "<-" einen Vektor zuweise:

```
## Beachtet die Länge des Vektors
mdf$Augenfarbe <- c("blau", "grau", "blau", "braun", "gruen")
```

```
mdf
```

```
  Nummer Item1 Item2 Alter Geschlecht Augenfarbe
1      1     1     1    13          w       blau
2      2     0     1    14          m       grau
3      3     0     0    13          m       blau
4      4     1     0    12          w      braun
5      5     1     1    15          m      gruen
```

Beim Anhängen von Spalten an `data.frames` mit der $-Notation kann ich jegliche Berechnungsvorschriften für Vektoren verwenden. So kann ich etwa einen Testscore über zwei Items berechnen und direkt an den `data.frame` anhängen:

```
mdf$Testscore <- mdf$Item1 + mdf$Item2

mdf$Testscore
```

```
[1] 2 1 0 1 2
```

In diesem Beispiel kommt die $-Notation recht häufig zum Einsatz, was etwas gewöhnungsbedürftig aussieht. Aber es ist wichtig darauf zu achten. Die Variablen,[18] die wir verwenden,

[18] Es ist etwas unglücklich, dass der Begriff "Variable" doppeldeutig ist: (1) In R sind Variablen die Speicherorte von Objekten; ich erstelle sie mit der "<-"-Zuweisung. (2) Andererseits werden auch Messwerte – etwa die Punktzahlen in einem Testitem – als Variablen bezeichnet. In diesem Sinne würde der Begriff Variable in R

um den Testscore zu berechnen, "wohnen" in `mdf` und können nicht ohne Verweis darauf adressiert werden. Das hier geht schief:

```
mdf$Testscore <- Item1 + Item2
Fehler: Objekt 'Item1' nicht gefunden
```

Hier sucht `R` nach einer Variablen `Item1`, die aber nicht existiert; `Item1` ist nur eine Spalte von `mdf`. Noch schlimmer wäre es, wenn in meiner Arbeitsumgebung tatsächlich Variablen mit den Namen `Item1` und `Item2` existieren sollten. In dem Fall würden wir gegebenenfalls falsche Daten abspeichern und nicht einmal eine Fehlermeldung erhalten.

Mit der `$`-Notation werden wir häufig auf Daten zugreifen, um Berechnungen durchzuführen. Wir können beispielsweise Mittelwerte von Messvariablen berechnen oder uns Häufigkeiten von kategorialen Daten angeben lassen:

```
mean(mdf$Alter)
```

```
[1] 13.4
```

```
table(mdf$Geschlecht)
```

```
m w
3 2
```

Die Funktion `mean` kennen wir bereits. Die Funktion `table` berechnet die Häufigkeiten aller Werte, die in einem Vektor vorkommen. Wir nutzen `table` vor allem zur Beschreibung kategorialer Messvariablen. Auch zur Überprüfung der Plausibilität von Daten ist `table` nützlich. (Ist jeder Wert ein "legaler" Wert, der auch vorkommen sollte?) Ich kann die Funktion `table` auch verwenden, um die Häufigkeit der Kombination von mehreren Variablen zu erfragen, etwa wie häufig welcher Testscore nach Geschlecht auftaucht:

```
## Erstelle Kreuztabelle von Geschlecht und Augenfarbe:
table(mdf$Augenfarbe, mdf$Geschlecht)
```

```
       m w
  blau  1 1
  braun 0 1
  grau  1 0
  gruen 1 0
```

auf die Spalte in einem `data.frame` verweisen, da Spalten Messvariablen beinhalten. Diese Doppeldeutigkeit ist deswegen unglücklich, da eine Spalte in einem `data.frame` keine `R`-Variable ist. Stattdessen ist der gesamte `data.frame` in **einer** Variablen abgespeichert.

3.3 Zugriff auf Spalten und Zeilen: die `[·,·]`-Notation

Einzelne Spalten können wir mit dem `$`-Zugriff aus `data.frames` auslesen. Wir lernen nun den `[·,·]`-Zugriff kennen, mit dem wir nicht nur einzelne Spalten, sondern beliebige Spalten und Zeilen aus `data.frames` auslesen können. Wie wir sehen werden, ist der `[·,·]`-Zugriff dem `[·]`-Zugriff ähnlich, den wir zur Auswahl von Daten aus Vektoren kennengelernt haben.

Der `[·,·]`-Zugriff erlaubt es uns, eine Teilmenge aller Fälle aus `mdf` auszuwählen, etwa nur die Personen mit blauen Augen, oder alle Personen, die den maximalen Testwert erreicht haben. Für solche Auswahlen hilft uns unser Wissen über logische Vergleiche aus dem letzten Kapitel. Betrachten wir zunächst ein Beispiel:

```
mdf[mdf$Augenfarbe == "blau", ]
```

```
  Nummer Item1 Item2 Alter Geschlecht Augenfarbe Testscore
1      1     1     1    13          w       blau         2
3      3     0     0    13          m       blau         0
```

Beachtet, dass durch diesen Aufruf der `data.frame`, der in der Variablen `mdf` abgespeichert ist, nicht verändert wird. Der `[·,·]`-Zugriff gibt stattdessen einen neuen `data.frame` zurück, der nur die Fälle enthält, bei denen `mdf$Augenfarbe == "blau"` `TRUE` ergibt. Wir müssten das Ergebnis der Funktion in einer Variablen speichern, wenn wir damit weiter arbeiten wollen (Erinnerung: Kapitel 2).

Wie das folgende Beispiel zeigt, können wir mit der `[·,·]`-Notation auch gezielt Spalten aus `data.frames` auswählen:

```
mdf[, c("Augenfarbe", "Alter")]
```

```
  Augenfarbe Alter
1       blau    13
2       grau    14
3       blau    13
4      braun    12
5      gruen    15
```

Die zwei Beispiele zeigen, dass das Komma in der `[·,·]`-Notation dafür entscheidend ist, ob eine Auswahl nach Zeilen oder Spalten stattfindet. **Vor dem Komma werden Zeilen adressiert, nach dem Komma Spalten.** Es ist auch eine gleichzeitige Auswahl nach Zeilen **und** Spalten möglich. Allgemein ist die Syntax zum Ansprechen von `data.frames` mit dem `[·,·]`-Zugriff die folgende:

```
data.frame[Reihenvektor, Spaltenvektor]
```

Dabei ist *Reihenvektor*/*Spaltenvektor* entweder ein (a) numerischer Vektor, der die Indexe der Reihen/Spalten enthält, die ausgewählt werden sollen, (b) ein logischer Vektor, der für jede Reihe/Spalte kodiert, ob diese in der Ausgabe enthalten sein soll (vgl. Kapitel 2), oder (c) Vektor vom Typ `character`, der die Namen der Zeilen/Spalten enthält, die ausgegeben

werden sollen.[19]

Spalten werden am häufigsten per Namen – also durch Angabe eines einen Vektors vom Typ `character` – adressiert; Zeilen werden am häufigsten durch einen logischen Ausdruck – also durch Angabe eines einen Vektors vom Typ `logical` – adressiert ("Gib mir alle Fälle aus, die eine bestimmte Eigenschaft aufweisen."). Durch die UND- bzw. ODER-Operationen können wir auch komplexere logische Bedingungen zur Auswahl von Fällen formulieren:

```
mdf[mdf$Augenfarbe == "blau" | mdf$Augenfarbe == "braun", ]
```

```
  Nummer Item1 Item2 Alter Geschlecht Augenfarbe Testscore
1      1     1     1    13          w       blau         2
3      3     0     0    13          m       blau         0
4      4     1     0    12          w      braun         1
```

```
mdf[(mdf$Augenfarbe == "blau" | mdf$Augenfarbe == "braun") & mdf$Item1 == 1, ]
```

```
  Nummer Item1 Item2 Alter Geschlecht Augenfarbe Testscore
1      1     1     1    13          w       blau         2
4      4     1     0    12          w      braun         1
```

Beachtet, dass wir hier ohne Klammerung der ODER-Operation eine andere Ausgabe erhalten (Erinnerung: Diesen Fall kennen wir auch aus Kapitel 2):

```
mdf[mdf$Augenfarbe == "blau" | mdf$Augenfarbe == "braun" & mdf$Item1 == 1, ]
```

```
  Nummer Item1 Item2 Alter Geschlecht Augenfarbe Testscore
1      1     1     1    13          w       blau         2
3      3     0     0    13          m       blau         0
4      4     1     0    12          w      braun         1
```

Wie wir merken, wird die `[·,·]`-Notation wird recht schnell unübersichtlich, wenn sie komplexere logische Anfragen enthält. Die Verknüpfung mehrerer ODER-Bedingungen lässt sich durch den `%in%`-Operator verkürzen:

```
mdf[mdf$Augenfarbe %in% c("blau", "braun"), ]
```

```
  Nummer Item1 Item2 Alter Geschlecht Augenfarbe Testscore
1      1     1     1    13          w       blau         2
3      3     0     0    13          m       blau         0
4      4     1     0    12          w      braun         1
```

Im nächsten Abschnitt lernen wir mit der Funktion `subset` eine Möglichkeit kennen, komplexere logische Anfragen noch etwas prägnanter zu formulieren. Zum Abschluss dieses Abschnitts betrachten wir noch einige weitere Beispiele für die verschiedenen Auswahlmöglichkeiten per `[·,·]`:

[19] Auch Zeilen können benannt sein. Den Fall hatten wir bislang aber nicht und es kommt auch nicht oft vor, dass Zeilen explizit benannt sind. Häufiger ist der Fall, in dem wir Spalten nach Namen auswählen.

```
## Wähle per Index die ersten drei Zeilen aus
mdf[1:3, ]
```

```
  Nummer Item1 Item2 Alter Geschlecht Augenfarbe Testscore
1      1     1     1    13          w       blau         2
2      2     0     1    14          m       grau         1
3      3     0     0    13          m       blau         0
```

```
## Wähle per Index die zweite und vierte Spalte aus
mdf[, c(2, 4)]
```

```
  Item1 Alter
1     1    13
2     0    14
3     0    13
4     1    12
5     1    15
```

```
## Wähle per logischem Vektor alle Personen aus, die beide Aufgaben
## richtig gelöst haben:
mdf[mdf$Testscore == 2, ]
```

```
  Nummer Item1 Item2 Alter Geschlecht Augenfarbe Testscore
1      1     1     1    13          w       blau         2
5      5     1     1    15          m      gruen         2
```

```
## Wähle Fallnummer, Alter und Testscore per Spaltenname aus:
mdf[, c("Nummer", "Alter", "Testscore")]
```

```
  Nummer Alter Testscore
1      1    13         2
2      2    14         1
3      3    13         0
4      4    12         1
5      5    15         2
```

```
## Wähle Fallnummer, Alter und Testscore aus für alle Personen, die
## älter als 13 sind
mdf[mdf$Alter > 13, c("Nummer", "Alter", "Testscore")]
```

```
  Nummer Alter Testscore
2      2    14         1
5      5    15         2
```

```
## Wähle Fallnummer, Alter und Testscore aus für die ersten drei Fälle
mdf[1:3, c("Nummer", "Alter", "Testscore")]
```

```
  Nummer Alter Testscore
1      1    13         2
```

```
2      2    14         1
3      3    13         0
```

```
## Wähle die Itemscores aus - nutze dabei die Funktion paste0
mdf[, paste0("Item", 1:2)]
```

```
  Item1 Item2
1     1     1
2     0     1
3     0     0
4     1     0
5     1     1
```

> **Merke**: Mit dem `[·,·]`-Zugriff wird vor dem Komma die Zeile und nach dem Komma die Spalte adressiert. Man kann die Auswahl nach numerischem Index, mit einem logischen Vektor oder mit einem `character` Vektor durchführen.

3.4 Die Funktion `subset`

In diesem Abschnitt lernen wir die Funktion `subset` kennen, die zwei wichtige Zugriffe auf `data.frames` vereinfacht:

1. Die Auswahl von Zeilen mithilfe logischer Bedingungen
2. Die Auswahl von Spalten durch Angabe von Spaltennamen

Des Weiteren bietet der Abschnitt einen ersten allgemeinen Überblick über die Arbeitsweise von Funktionen, der in Kapitel 5 vertieft wird.

3.4.1 Vereinfachte Zeilenauswahl

Die Funktion `subset` erlaubt uns logische Bedingungen für die Zeilenauswahl zu formulieren, ohne die `$`-Notation zu verwenden:

```
subset(mdf, Augenfarbe == "blau")
```

```
  Nummer Item1 Item2 Alter Geschlecht Augenfarbe Testscore
1      1     1     1    13          w       blau         2
3      3     0     0    13          m       blau         0
```

Außerhalb der Funktion `subset` würde der Ausdruck `Augenfarbe == "blau"` einen Fehler ausgeben; schließlich ist `Augenfarbe` selbst keine `R`-Variable, sondern nur eine Spalte von `mdf`.[20] Innerhalb der Funktion `subset` kann die logische Bedingung in dieser Form jedoch verarbeitet werden. Durch Angabe von `mdf` im ersten Argument teilen wir `subset` mit, aus welchem `data.frame` Zeilen ausgewählt werden sollen.

[20] Es macht an dieser Stelle Sinn, einen Moment inne zu halten und zu überlegen, warum es eigentlich außergewöhnlich ist, dass der Befehl `Augenfarbe == "blau"` innerhalb der Funktion `subset` funktioniert.

Der äquivalente Befehl mit der `[·,·]`-Notation sähe folgendermaßen aus:

```
mdf[mdf$Augenfarbe == "blau", ]
```

Gerade bei der Verknüpfung mehrerer logischer Bedingungen ist es praktisch, nicht mehrfach die `$`-Notation verwenden zu müssen:

```
subset(mdf, Augenfarbe == "blau" & Item1 == 1)
```

```
  Nummer Item1 Item2 Alter Geschlecht Augenfarbe Testscore
1      1     1     1    13          w       blau         2
```

3.4.2 Funktionsargumente

Die Funktion `subset` nimmt optional ein drittes Argument an, das auszulesende Spalten adressiert:

```
subset(mdf, Augenfarbe == "blau", c("Item1", "Augenfarbe"))
```

```
  Item1 Augenfarbe
1     1       blau
3     0       blau
```

Durch die Kombination der Auswahl von Zeilen und Spalten gibt dieser Befehl einen `data.frame` aus, der nur die Spalten `Item1` und `Augenfarbe` enthält, und diese nur für Personen mit blauen Augen.

Was machen wir aber, wenn wir nur eine Auswahl nach Spalten durchführen wollen? Probieren wir erst einmal Folgendes:

```
subset(mdf, c("Item1", "Augenfarbe"))
```

Hier habe ich einfach das Argument für die Zeilenauswahl weggelassen und als zweites Argument einen `character`-Vektor zur Auswahl zweier Spalten angegeben – was aber nicht funktioniert hat. `R` gibt uns eine kryptische Fehlermeldung aus:

```
Fehler in subset.data.frame(mdf, c("Item1", "Augenfarbe")) :
  'subset' must be logical
```

Warum gibt uns `R` hier einen Fehler aus? An dieser Stellen machen wir uns eine wichtige Eigenschaft von Funktionen bewusst: **Funktionen identifizieren Argumente anhand der Reihenfolge, in der sie übergeben werden.** Bei der Funktion `subset` ist das erste Argument der `data.frame`, von dem wir Daten anfordern. Das zweite Argument wählt mit einem logischen Ausdruck Zeilen aus. Das dritte Argument adressiert Spalten.

Wir erhalten den obigen Fehler also, weil die Funktion `subset` an zweiter Stelle einen logischer Ausdruck zur Zeilenauswahl erwartet; die Auswahl der Spalten muss mit dem dritten Argument geschehen. Um eine Auswahl trotzdem nur nach Spalten auszuführen, können wir eine praktische Eigenschaft von `R` ausnutzen: **Funktionsargumente haben Namen.**

Anstatt Argumente anhand ihrer Position zu identifizieren, können wir sie auch benennen. Bislang haben wir das ignoriert bzw. es ist uns nur am Rande begegnet – erinnern wir uns an das Argument `na.rm` der Funktion `mean`.

Die Funktion `subset` hat drei benannte Argumente:

- `x`: der Datensatz, aus dem ausgewählt wird
- `subset`: wählt Zeilen aus
- `select`: wählt Spalten aus

Um eine Übersicht über die Argumente einer Funktion zu erhalten, können wir mit dem `?`-Operator die `R`-Hilfe anfordern:

```
?subset
```

Leider ist die `R`-Hilfe oftmals kryptisch – und das nicht nur für Anfänger. Sie ist die offizielle Dokumentation von Funktionen und legt deswegen zwar großen Wert auf technische Genauigkeit, ist aber nicht immer sonderlich ausführlich oder gar verständlich. Wir werden in Kapitel 5 bei einer ausführlicheren Besprechung von Funktionen noch einmal darauf zurückkommen, wie wir mit der `R`-Hilfe umgehen können.

Wenn wir die Namen der Argumente kennen, können wir die Funktion `subset` auch wie folgt aufrufen:

```
subset(x = mdf, subset = Augenfarbe == "blau",
       select = c("Item1", "Augenfarbe"))
```

```
  Item1 Augenfarbe
1     1       blau
3     0       blau
```

Wenn ich Funktionsargumente mit Namen adressiere, kann ich deren Reihenfolge beliebig vertauschen. Deswegen funktioniert auch der folgende Aufruf:

```
subset(select = c("Item1", "Augenfarbe"),
       subset = Augenfarbe == "blau", x = mdf)
```

```
  Item1 Augenfarbe
1     1       blau
3     0       blau
```

Eine Auswahl nur anhand von Spalten können wir wie folgt durchführen:

```
subset(mdf, select = c("Item1", "Augenfarbe"))
```

```
  Item1 Augenfarbe
1     1       blau
2     0       grau
3     0       blau
4     1      braun
5     1      gruen
```

Dieser Aufruf zeigt, dass wir im selben Aufruf manche Argumente anhand ihrer Position identifizieren können und manche anhand ihres Namens. Für das erste Argument `mdf` habe ich den Namen nicht extra angegeben – daher wurde das Argument anhand seiner Position identifiziert. Für die Auswahl der Spalten habe ich jedoch den Argumentnamen angegeben. **Das war auch nötig**, da `subset` als zweites Argument ansonsten die Auswahl der Zeilen erwartet hätte.

> **Merke**: In `R` können Funktionsargumente per Position und per Namen identifiziert werden. Die Identifikation per Name schlägt dabei die Identifikation per Position. Argumente explizit mit ihrem Namen zu benennen ist oft sicherer als auf die richtige Reihenfolge der Argumente zu vertrauen.

3.4.3 Sonderregeln zur Auswahl von Spalten

Die Funktion `subset` bietet einige Sonderregeln zur Auswahl von Spalten, die über die Angabe der Spaltennamen per `character`-Vektor hinausgehen, wie wir sie von der `[·,·]`-Notation kennen. Zunächst einmal können wir Spaltennamen ohne Anführungszeichen angeben:

```
subset(mdf, select = c(Augenfarbe, Alter))
```

```
  Augenfarbe Alter
1       blau    13
2       grau    14
3       blau    13
4      braun    12
5      gruen    15
```

Genau wie die Zeilenauswahl, die ohne die `$`-Notation auskommt, ist es eine Besonderheit der Funktion `subset`, dass wir Spaltennamen ohne Anführungszeichen adressieren können. Einfach in die Konsole eingegeben würde der Ausdruck `c(Augenfarbe, Alter)` höchstwahrscheinlich[21] einen Fehler verursachen, da `Augenfarbe` und `Alter` nicht unbedingt als Variablen definiert sind; sie sind bloß Spalten von `mdf`.

Die Auswahl von Spalten ohne Anführungszeichen ist noch keine allzu große Arbeitserleichterung im Vergleich zur `[·,·]`-Notation. Die Funktion `subset` lässt aber noch einen weiteren Sonderfall bei der Spaltenauswahl zu, der einiges an Schreibarbeit ersparen kann: Wir können den Doppelpunktoperator verwenden, um mehrere Spalten auszuwählen.

```
subset(mdf, select = Item1:Geschlecht)
```

```
  Item1 Item2 Alter Geschlecht
1     1     1    13          w
2     0     1    14          m
3     0     0    13          m
```

[21] Unter welchen Umständen würde `R` keinen Fehler ausgeben, wenn wir `c(Augenfarbe, Alter)` in die Konsole eingeben? Es ist nützlich, diesen Punkt zu verstehen.

```
4      1      0     12           w
5      1      1     15           m
```

Von „links nach rechts“ wählt der Doppelpunktperator alle Spalten zwischen einschließlich `Item1` und `Geschlecht` aus. Auch hier verzichten wir auf die Angabe von Anführungszeichen.

Es gibt noch eine weitere Vereinfachung, die uns die Funktion `subset` bietet. Wir können angeben, welche Spalten wir **nicht** ausgeben wollen:

```
subset(mdf, select = -c(Geschlecht, Alter))
```

```
  Nummer Item1 Item2 Augenfarbe Testscore
1      1     1     1       blau         2
2      2     0     1       grau         1
3      3     0     0       blau         0
4      4     1     0      braun         1
5      5     1     1      gruen         2
```

3.5 Weitere Zugriffe auf `data.frames`

Dieser Abschnitt behandelt zwei weitere Möglichkeiten, mit eckigen Klammern auf Spalten in `data.frames` zuzugreifen. Da wir diese Zugriffe danach erst einmal nicht weiter verwenden, kann der folgende Inhalt jedoch zunächst problemlos übersprungen werden. Datenzugriffe mit eckigen Klammern sind jedoch ein zentraler Bestandteil von `R`; daher lohnt es sich, diesen Abschnitt später zu konsultieren oder zum Nachschlagen zu nutzen.

3.5.1 Der `[[·]]`-Zugriff

Den `[[·]]`-Zugriff nutzen wir genau wie den `$`-Zugriff zum Auslesen einzelner Spalten aus `data.frames`:

```
mdf[["Item1"]] # dasselbe wie mdf$Item1
```

```
[1] 1 0 0 1 1
```

Hierbei wird der Spaltenname als ein-elementiger Vektor vom Typ `character` angegeben – also in Anführungszeichen. Die Anführungszeichen sind hier notwendig, bei der `$`-Notation verwenden wir sie hingegen nicht. Das hat zur Folge, dass wir statt der expliziten Angabe des Texts auch eine Variable übergeben können, die einen ein-elementigen `character`-Vektor abgespeichert hat; dies ist mit der `$`-Notation nicht möglich.

```
spalte <- "Augenfarbe"
mdf[[spalte]]
```

```
[1] "blau"  "grau"  "blau"  "braun" "gruen"
```

Ebenso ist es möglich, der `[[·]]`-Klammerung eine Funktion zu übergeben, die einen einelementigen Vektor vom Typ character ausgibt – etwa die Funktion `paste0`:

```
mdf[[paste0("Item", 1)]]
```

```
[1] 1 0 0 1 1
```

Der `[[·]]`-Zugriff wird in Zusammenspiel mit der Funktion `paste0` noch einmal interessant werden, wenn wir in Kapitel 6 mit *Schleifen* nacheinander auf beliebig viele Spalten von `data.frames` zugreifen. In einer Schleife können wir dann Spaltennamen automatisiert nacheinander austauschen (etwa: `Item_1`, `Item_2`, ...).

3.5.2 Der `[·]`-Zugriff

Nicht äquivalent zu den Zugriffen mit `$` und `[[·]]` ist folgender `[·]`-Zugriff:

```
mdf["Item1"]
```

```
  Item1
1     1
2     0
3     0
4     1
5     1
```

Auch hier sind Anführungszeichen zur Identifikation der auszuwählenden Spalte nötig. Der Unterschied von `[·]` zu `[[·]]` und `$`:

- `[[·]]` und `$` ergeben einen Vektor
- `[·]` ergibt einen `data.frame` mit einer Spalte

Außerdem können wir mit dem `[·]`-Zugriff gleichzeitig mehrere Spalten auswählen, indem wir einen mehr-elementigen Vektor vom Typ `character` übergeben. Das ist mit den Zugriffen per `[[·]]` und `$` nicht möglich, die immer nur eine Spalte ausgeben.

```
mdf[c("Item1", "Augenfarbe")]
```

```
  Item1 Augenfarbe
1     1       blau
2     0       grau
3     0       blau
4     1      braun
5     1      gruen
```

Dieser Aufruf sollte uns an die `[·,·]`-Notation zur Auswahl von Spalten erinnern; in der Tat ist der folgende Ausdruck äquivalent:

```
mdf[, c("Item1", "Augenfarbe")]
```

```
  Item1 Augenfarbe
1     1       blau
2     0       grau
3     0       blau
4     1      braun
5     1      gruen
```

Zwischen dem `[·]`-Zugriff und der `[·,·]`-Auswahl für Spalten gibt es jedoch einen Unterschied, der zutage kommt, wenn wir nur eine Spalte auswählen:

```
mdf["Item1"]
```

```
  Item1
1     1
2     0
3     0
4     1
5     1
```

```
mdf[, "Item1"]
```

```
[1] 1 0 0 1 1
```

Wenn wir nur eine Spalte auslesen, gibt die `[·,·]`-Auswahl einen Vektor aus, die `[·]`-Auswahl jedoch einen `data.frame` mit einer Spalte. Dieses Verhalten offenbart einen Sonderfall der `[·,·]`-Auswahl: Im Normalfall gibt `[·,·]` ebenfalls einen ganzen `data.frame` aus. Wenn wir aber nur eine einzige Spalte anfordern, „reduziert“ sich die Ausgabe zu einem Vektor.

Ich persönlich bevorzuge die `[·,·]`-Notation gegenüber der `[·]`-Notation zur Auswahl von Spalten, auch wenn ich hier ein zusätzliches Komma verwenden muss (ansonsten sind die beiden Notationen ja fast äquivalent zur Auswahl von Spalten). Wenn ich Code mit der `[·,·]`-Notation lese, weiß ich, dass Spalten ausgewählt werden – selbst wenn ich gar nicht weiß, was in dem Objekt steckt, auf dem die Auswahl stattfindet. Die `[·]`-Notation ist uneindeutiger: Sie könnte auch auf einem Vektor operieren, der gar keine Spalten enthält. Wir merken uns: **Code ist in erster Linie für Menschen gemacht; verständlicher Code ist gegenüber kürzerem Code zu bevorzugen.**

3.5.3 Zugriff nach Name und Index

Wir haben nun alle wichtigen Möglichkeiten kennengelernt, Zugriffe auf `data.frames` durchzuführen. An dieser Stelle lohnt es sich deswegen, ein grundsätzliches Prinzip von Datenzugriffen in `R` festzuhalten: **Datenzugriffe können nach nach Index oder nach Name stattfinden**. Dies gilt für Vektoren, `data.frames` und auch für andere Datenstrukturen, die wir noch gar nicht behandelt haben.

Wir haben bereits Beispiele für beide Arten des Datenzugriffs kennengelernt: In Vektoren haben wir Zugriffe mithilfe von Indexen durchgeführt, indem wir (a) die Position von auszuwählenden Elementen mit einem numerischen Vektor angegeben haben, oder (b) indem

wir einen logischen Vektor übergeben haben, der die Indexe auswählt, deren Elemente ausgegeben werden sollen. Der Vollständigkeit halber sei hier mitgeteilt, dass man sogar bei Vektoren Zugriffe nach Namen durchführen kann, wenn die Elemente des Vektors benannt sind. Das ist gar nicht so ungewöhnlich; wie folgt könnte man einen Vektor mit benannten Elementen erstellen und mit der bekannten `[·]`-Notation darauf zugreifen.

```
## Benannte Vektoren erstellen funktioniert wie einen data.frame zu
## erstellen:
vec <- c(foo = 1, bar = 2)
vec
```

```
foo bar
  1   2
```

```
vec["foo"]
```

```
foo
  1
```

```
vec["bar"]
```

```
bar
  2
```

```
vec[c("bar", "foo")]
```

```
bar foo
  2   1
```

In `data.frames` haben wir Spalten zumeist nach Namen ausgewählt:

- Mit der `$`-Notation
- Mit der `[·,·]`-Notation
- Mit der Funktion `subset`
- Mit der `[[·]]`-Notation
- Mit der `[·]`-Notation

Wie wir gesehen haben, können wir mit der `[·,·]`-Notation in `data.frames` zusätzlich auch Zugriffe nach numerischem oder logischem Index durchführen. Dabei kann die Auswahl sowohl nach Spalten als auch nach Zeilen – oder beidem – geschehen.

3.6 Nützliche Funktionen zum Arbeiten mit `data.frames`

3.6.1 `tapply`

Die Funktion `tapply` kann ich verwenden, um mir deskriptive Statistiken anhand von Gruppierungsvariablen ausgeben zu lassen, hier etwa die mittlere Punktzahl oder das mittlere Alter nach Geschlecht der Schüler/innen:

```
tapply(mdf$Testscore, mdf$Geschlecht, mean)
```

```
  m   w
1.0 1.5
```

```
tapply(mdf$Alter, mdf$Geschlecht, mean)
```

```
   m    w
14.0 12.5
```

Die Funktion `tapply` erhält als erstes Argument den Messwertvektor, für den Statistiken angefordert werden. Das zweite Argument ist die Gruppierungsvariable.[22] Interessanterweise ist das dritte Argument eine Funktion, in diesem Fall die Funktion `mean`. So können wir die *mittlere* Punktzahl nach Geschlecht anfordern. Entsprechend könnten wir hier andere Funktionen übergeben, um etwa die Standardabweichung des Alters zu erfragen:

```
tapply(mdf$Alter, mdf$Geschlecht, sd)
```

```
        m         w
1.0000000 0.7071068
```

Wie `table` kann auch `tapply` deskriptive Statistiken anhand mehrerer Gruppierungsvariablen anfordern. Um mehrere Gruppierungsvariablen anzufordern, klammern wir `list(...)` um die Gruppierungsvektoren im zweiten Argument:

```
tapply(mdf$Alter, list(mdf$Geschlecht, mdf$Augenfarbe), mean)
```

```
  blau braun grau gruen
m   13    NA   14    15
w   13    12   NA    NA
```

Mit nur fünf Datenpunkten macht diese Anfrage hier nur wenig Sinn, da jeder ausgegebene Mittelwert nur anhand eines einzelnen Wertes gebildet wurde,[23] was die Idee des Mittelwerts eher ad absurdum führt. Manche Kombinationen von Geschlecht und Augenfarbe kommen in unseren Daten sogar gar nicht vor; in diesen Fällen wird `NA` ausgegeben. `tapply` zeigt ihre Stärke vor allem, wenn man viele – und nicht nur fünf – Datenpunkte hat. Das gilt gerade dann, wenn wir mehrere Gruppierungsvariablen angeben.

3.6.2 `nrow` und `ncol`

Die Zahl der Zeilen eines `data.frame`s – d.h. oftmals die Zahl der *Fälle* – lässt sich mit der Funktion `nrow` bestimmen, die man sehr häufig verwendet:

[22] Beachtet, dass sowohl Messwerte als auch Gruppierungsvariable als **Vektoren** übergeben werden. Ich behandle die Funktion `tapply` jedoch im Kapitel zu `data.frames`, da es zumeist so sein wird, dass wir beide Vektoren aus **einem** `data.frame` mit der `$`-Notation auslesen werden.

[23] Wie viele Datenpunkte in die Berechnung jedes Mittelwerts eingehen, können wir in diesem Fall prüfen mit `table(mdf$Geschlecht, mdf$Augenfarbe)`.

```
nrow(mdf)
```

```
[1] 5
```

Analog ergibt `ncol` die Zahl der Spalten:

```
ncol(mdf)
```

```
[1] 7
```

3.6.3 `head` und `tail`

Um sich einen Überblick über einen `data.frame` zu verschaffen, sind die Funktionen `head` und `tail` sehr nützlich. `head` gibt die ersten Zeilen eines `data.frames` zurück, `tail` entsprechend die letzten Zeilen. Beide Funktionen haben ein zweites Argument n, welches wir nutzen können, um zu steuern, wie viele Zeilen ausgegeben werden sollen. Wenn wir n nicht angeben, werden sechs Zeilen ausgegeben (in `R`-Jargon: 6 ist der "default"-, also Standardwert des *optionalen Arguments* n). Beispiel:

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

```
  Nummer Item1 Item2 Alter Geschlecht Augenfarbe Testscore
1      1     1     1    13          w       blau         2
2      2     0     1    14          m       grau         1
```

```
tail(mdf)
```

```
  Nummer Item1 Item2 Alter Geschlecht Augenfarbe Testscore
1      1     1     1    13          w       blau         2
2      2     0     1    14          m       grau         1
3      3     0     0    13          m       blau         0
4      4     1     0    12          w      braun         1
5      5     1     1    15          m      gruen         2
```

Im letzteren Fall werden einfach alle Zeilen zurückgegeben, da unser `data.frame` insgesamt nur fünf Zeilen hat – und somit weniger als sechs.

3.6.4 Sortieren: `dplyr::arrange`

Oftmals wollen wir Datentabellen nach einer oder mehreren Variablen sortieren. Dies funktioniert am bequemsten, wenn wir das *Paket* `dplyr` laden (Wickham, François, Henry, & Müller, 2018):

```
library("dplyr")
```

Voraussetzung dafür, dass ich das Paket `dplyr` nutzen kann ist, dass ich das Paket auf meinem Rechner installiert habe. Falls das Paket noch nicht installiert ist – in dem Fall ergibt

der Befehl `library('dyplr')` einen Fehler) –. können wir es es mit dem folgenden Befehl installieren:

```
install.packages("dplyr")
```

Pakete stellen zusätzliche Funktionen zur Verfügung, die in der Basisversion von `R` nicht enthalten sind. Um ein Paket zu nutzen, müssen wir es mit der Funktion `library` in unsere `R`-Umgebung laden. Andernfalls könnten wir die Funktionen nicht nutzen, die etwa `dplyr` enthält. Die Funktion `arrange` aus `dplyr` ermöglicht es uns, einen `data.frame` zu sortieren:

```
arrange(mdf, Testscore) # dplyr muss geladen sein
```

```
  Nummer Item1 Item2 Alter Geschlecht Augenfarbe Testscore
1      3     0     0    13          m       blau         0
2      2     0     1    14          m       grau         1
3      4     1     0    12          w      braun         1
4      1     1     1    13          w       blau         2
5      5     1     1    15          m      gruen         2
```

In der Funktion `arrange` geben wir als erstes Argument den zu sortierenden `data.frame` an. Darauf folgen – mit Komma separiert – alle Spalten nach denen wir sortieren wollen (hier erst mal nur der Testscore). Standardmäßig sortiert `arrange` *aufsteigend*; wenn wir eine absteigende Sortierung wünschen, müssen wir ein Minus vor die Sortierspalte setzen:

```
arrange(mdf, -Testscore)
```

```
  Nummer Item1 Item2 Alter Geschlecht Augenfarbe Testscore
1      1     1     1    13          w       blau         2
2      5     1     1    15          m      gruen         2
3      2     0     1    14          m       grau         1
4      4     1     0    12          w      braun         1
5      3     0     0    13          m       blau         0
```

Es ist auch möglich, nach mehreren Spalten zu sortieren. In dem Fall wird bei gleichen Werten im ersten Sortierkriterium anhand des nächsten Kriteriums die Reihenfolge entschieden. Wir könnten etwa unsere Daten nach Geschlecht sortieren, und innerhalb der Personen gleichen Geschlechts nach Punktzahl:

```
arrange(mdf, Geschlecht, -Testscore)
```

```
  Nummer Item1 Item2 Alter Geschlecht Augenfarbe Testscore
1      5     1     1    15          m      gruen         2
2      2     0     1    14          m       grau         1
3      3     0     0    13          m       blau         0
4      1     1     1    13          w       blau         2
5      4     1     0    12          w      braun         1
```

3.7 Zusammenfassung

- Wir haben den `data.frame` als Datenstruktur zur Speicherung von psychometrischen Daten kennengelernt
- Wir haben die `$`-Notation für den Zugriff auf einzelne Spalten von `data.frames` kennengelernt
- Mit der `[·,·]`-Notation und der Funktion `subset` können wir Zeilen und Spalten aus `data.frames` auslesen
- Zur Anforderung von deskriptiven Statistiken können wir die Funktionen `table` und `tapply` verwenden
- Wir haben weitere Funktionen kennengelernt, die uns einen Überblick über `data.frames` verschaffen:
  - `names`
  - `nrow/ncol`
  - `head/tail`
  - `dplyr::arrange`

3.8 Fragen zum vertiefenden Verständnis

1. Worin unterscheiden sich die folgenden Aufrufe? Welche Aufrufe sind zueinander äquivalent?

```r
subset(mdf, select = "Item1")
```

```r
mdf[, "Item1"]
```

```r
mdf[, "Item1", drop = FALSE]
```

```r
mdf["Item1"]
```

```r
mdf[["Item1"]]
```

```r
mdf$Item1
```

2. Vergleicht die folgenden Aufrufe der Funktion `subset`. Warum funktionieren der erste und der zweite Aufruf, aber nicht der dritte und vierte? Wie kann es überhaupt sein, dass die ersten beiden Funktionsaufrufe funktionieren, obwohl Argumente unbenannt an der "falschen" Position stehen?

```r
subset(mdf, select = "Item1", Augenfarbe == "blau")
```

```r
subset(select = "Item1", mdf, Augenfarbe == "blau")
```

```r
subset(mdf, "Item1", Augenfarbe == "blau")
```

```r
subset("Item1", mdf, Augenfarbe == "blau")
```

4 Arbeiten mit psychometrischen Daten

Dieses Kapitel arbeitet einige Kennwerte der klassischen Testtheorie auf und bespricht wie wir diese in R berechnen können. Dabei werden die folgenden Konzepte behandelt:

- Testscores
- Item-Schwierigkeit
- Item-Trennschärfe
- Item-Interkorrelation
- Reliabilität
  - Interne Konsistenz („Cronbachs Alpha“)
  - Split-Half/Odd-Even-Reliabilität
- Spearman-Brown-Formel

Ein weiterer Teil des Kapitels beschäftigt sich mit der Aufbereitung von Rohdaten, die im Normalfall leider nicht in der Form vorliegen, die wir für unsere Analysen benötigen. Wir lernen

- Antworten umzukodieren
- Antworten zu invertieren
- Fälle mit fehlenden Werten auszuschließen

4.1 Ausgedehntes Beispiel zum Einstieg

Es folgt ein Beispiel zur Berechnung einiger grundlegender psychometrischer Kennwerte. Angenommen, uns liegt eine Datentabelle vor, die die Punktzahlen der Antworten von 10 Schulkindern auf 5 Aufgaben einer Klassenarbeit beinhaltet. Diese kann man gut in einer 10×5 (Reihe $\times$ Spalten) Datentabelle darstellen. Ein Eintrag kodiert, ob das Kind (*Reihe*) die Aufgabe (*Spalte*) korrekt gelöst hat. Korrekte Antworten werden mit 1 kodiert, falsche Antworten mit 0 – ein typisches Datenformat in der psychologischen Diagnostik.

Um das fortführende Beispiel selbst nachzuvollziehen, müsst ihr den folgenden `data.frame` erstellen:

```
test_data <- data.frame(Item_1 = c(1, 1, 1, 0, 0, 0, 1, 1, 1, 0),
                        Item_2 = c(0, 0, 1, 0, 0, 0, 0, 0, 0, 0),
                        Item_3 = c(1, 0, 1, 0, 1, 1, 1, 0, 1, 0),
                        Item_4 = c(1, 0, 1, 0, 1, 0, 0, 0, 0, 0),
                        Item_5 = c(1, 0, 1, 0, 1, 0, 0, 0, 0, 0))
```

Die Variable `test_data` enthält nun die folgende Tabelle:

	Item_1	Item_2	Item_3	Item_4	Item_5
1	1	0	1	1	1
2	1	0	0	0	0
3	1	1	1	1	1
4	0	0	0	0	0
5	0	0	1	1	1
6	0	0	1	0	0
7	1	0	1	0	0
8	1	0	0	0	0
9	1	0	1	0	0
10	0	0	0	0	0

Wenn uns Daten in diesem Format vorliegen, können wir auf viele Funktionen in `R` zurückgreifen, um grundlegende psychometrische Auswertungen durchzuführen. Dies sind etwa die Bestimmung der Schwierigkeit und der Trennschärfe von Items, sowie die Bestimmung einer Split-Half Reliabilität. Für fortgeschrittenere Auswertungen – wie etwa die Berechnung von Cronbachs Alpha oder einer Faktorenanalyse – werden wir auf Pakete zurückgreifen, die uns über die Basics in `R` hinaus weitere Funktionalitäten bieten. Aber auch für diese Analysen benötigen wir genau dieses Datenformat!

> **Merke**: Das ist das Standard-Datenformat für all unsere psychometrischen Berechnungen: (a) Zeilen sind Fälle; (b) Spalten sind Items bzw. Messvariablen; (c) Zellen enthalten Datenpunkte, etwa die Korrektheit von Antworten (kodiert mit 1/0). Datenpunkte müssen nicht unbedingt – wie es in diesem Beispiel der Fall ist – dichotom sein, sondern können beispielsweise auch die Antworten in einem Persönlichkeitsfragebogen auf einer Likert-Skala repräsentieren.

4.1.1 Testscores

Wir bestimmen zunächst die Testscores der 10 Kinder. Da jede Zeile ein Kind repräsentiert, ist der Gesamt-Testscore ist die Summe der Werte in jeder Reihe. Die Summe der Reihen eines `data.frames` (engl: *rows*) kann man mit der Funktion `rowSums` bestimmen:

```
rowSums(test_data)
```

```
 [1] 4 1 5 0 3 1 2 1 2 0
```

Es ist manchmal praktisch Berechnungen, die pro Fall einen Wert ergeben, direkt an den ursprünglichen `data.frame` anzuhängen. Wie in Kapitel 3 erklärt, ist das mit der `$`-Notation möglich:

```
test_data$score <- rowSums(test_data)
```

4.1.2 Item-Schwierigkeiten

Die Schwierigkeit eines Items ist die mittlere Punktzahl aller Personen in diesem Item.[24] Das ist somit also einfach der Mittelwert der Einträge in jeder Spalte (engl: *column*) in unserem Standardformat. Den Mittelwert pro Spalte kann ich mit der Funktion `colMeans` bestimmen (analog gibt es auch die Funktionen `colSums` und `rowMeans`):

```
colMeans(test_data)
```

```
Item_1 Item_2 Item_3 Item_4 Item_5  score
   0.6    0.1    0.6    0.3    0.3    1.9
```

Da ich gerade den Gesamtscore als Spalte an `test_data` angehängt habe, bekomme ich die mittlere Punktzahl der Schüler/innen in den fünf Testitems direkt mitgeliefert. Beachtet, dass ich hier mit `colMeans` eine numerische Funktion auf den ganzen `data.frame` angewendet habe. Würde der `data.frame` auch Spalten vom Typ `factor` oder `character` enthalten, wäre es nicht möglich, Funktionen wie `rowSums` und `colMeans` anzuwenden. In dem Fall könnte man mit der `[·,·]`-Notation oder der Funktion `subset` nur die gewünschten Spalten auswählen.

4.1.3 Item-Interkorrelationen

Als nächstes geben wir die Korrelationen zwischen allen Items als Korrelationsmatrix aus. Dies funktioniert mit der Funktion `cor`. Wenn `cor` als Argument einen `data.frame` erhält, wird eine Tabelle ausgegeben, die die Korrelation zwischen allen Spalten – d.h. Items – des `data.frames` enthält.

```
round(cor(test_data), 2)
```

	Item_1	Item_2	Item_3	Item_4	Item_5	score
Item_1	1.00	0.27	0.17	0.09	0.09	0.47
Item_2	0.27	1.00	0.27	0.51	0.51	0.65
Item_3	0.17	0.27	1.00	0.53	0.53	0.72
Item_4	0.09	0.51	0.53	1.00	1.00	0.87
Item_5	0.09	0.51	0.53	1.00	1.00	0.87
score	0.47	0.65	0.72	0.87	0.87	1.00

Die Korrelationen wurden aus Darstellungszwecken mit der Funktion `round` auf zwei Nachkommastellen gerundet.

[24] Auch bei Items, die nicht Korrektheit kodieren, kann man von Item-Schwierigkeit sprechen. Beispielsweise wäre dann die Item-Schwierigkeit die mittlere Zustimmungsrate für ein Item in einem Persönlichkeitsinventar, in dem Antworten auf einer 5-stufigen Likert-Skala gegeben werden.

4.1.4 Item-Trennschärfen

Interessant ist die letzte Spalte (bzw. genauso die letzte Zeile) der Tabelle der Item-Korrelationen. Diese gibt an, wie stark die Korrelation zwischen jedem Item und dem Testscore ausfällt. Dieser Kennwert ist die (unkorrigierte) Trennschärfe der Items; wir erhalten sie, da wir oben den Testscore als Spalte an unseren `data.frame` angehängt haben. Die Item-Trennschärfe macht eine Aussage darüber, wie stark das Abschneiden in einem Item mit dem Gesamt-Testscore zusammenhängt. Je höher die Trennschärfe, desto besser vermag das Item zwischen Schüler/innen mit viel und wenig Wissen (also einem hohen bzw. einem niedrigen Gesamt-Testscore) zu trennen. Die Trennschärfe ist ein Kennwert, der zur Beurteilung der Güte eines Items dienen kann.

Oftmals wird die „part-whole“ korrigierte Trennschärfe berechnet, bei der zur Berechnung der Trennschärfe jedes Items der Itemscore dieses Items aus der Gesamtpunktzahl ausgelassen wird. Somit wird eine „Kriterienkontamination“ vermieden, die zu einer Erhöhung der Trennschärfe führt. Diese Kriterienkontamination ergibt sich bei der unkorrigierten Trennschärfe daraus, dass der Itemscore selbst in das „Kriterium“ – also den Gesamt-Testscore – eingeht.[25] Im Folgenden berechnen wir eine part-whole korrigierte Trennschärfe für Item 2. Zunächst erstelle ich einen Vektor zur Auswahl der Items, die ich zur Berechnung des Testscores unter Ausschluss von Item 2 heranziehe:

```
## Wähle Antworten auf Items 1, 3, 4, 5 aus:
select_items   <- paste0("Item_", (1:5)[-2])
responses_no_item2 <- test_data[, select_items]

## Betrachte die Tabelle:
head(responses_no_item2)
```

```
  Item_1 Item_3 Item_4 Item_5
1      1      1      1      1
2      1      0      0      0
3      1      1      1      1
4      0      0      0      0
5      0      1      1      1
6      0      1      0      0
```

```
## Berechne den Testscore über Items 1, 3, 4 und 5:
corrected_score <- rowSums(responses_no_item2)
```

Die Variable `corrected_score` enthält nun die Testscores, die unter Ausschluss des zweiten Items gebildet wurden. Das Vorgehen zur Berechnung der bereinigten Scores lässt sich mithilfe der `[·,·]`-Notation und den Funktionen `paste0` und `rowSums` auch auf beliebig viele Items erweitern. Da wir hier den Summenwert nur über vier Items gebildet haben, hätte auch der folgende Code funktioniert:

[25] Praktisch gesehen werden unkorrigierte und korrigierte Trennschärfe dieselbe relative Rangreihe zwischen den Items hinsichtlich ihrer Diskriminationsgüte abbilden.

```
corrected_score <- test_data$Item_1 + test_data$Item_3 +
    test_data$Item_4 + test_data$Item_5
```

Wie folgt können wir nun mithilfe der Funktion `cor` die korrigierte Trennschärfe für Item 2 bestimmen:

```
cor(test_data$Item_2, corrected_score)
```

```
[1] 0.5238095
```

Die korrigierte Trennschärfe von 0.52 liegt unter der unkorrigierten Trennschärfe von `r round(item_correlations[6,2], 2)`. Je weniger Items ein Test hat, desto mehr Gewicht hat das einzelne Item für den Testscore und umso stärker weichen korrigierte und unkorrigierte Trennschärfe voneinander ab. Bei nur fünf Items kann der Effekt substantiell sein.

Es ist zu beachten, dass die Funktion `cor` an dieser Stelle anders verwendet wird als oben: Hier übergebe ich der Funktion `cor` mit dem Befehl `cor(test_data$Item_2, corrected_score)` zwei Vektoren gleicher Länge. Ein Vektor enthält die Korrektheiten der Antworten auf Item 2, der andere Vektor enthält den um Item 2 bereinigten Testscore. Oben habe ich der Funktion `cor` nur ein Argument übergeben, nämlich den `data.frame test_data`. In dem Fall wurde eine Tabelle ausgegeben – eine *Korrelationsmatrix* –, die die Korrelationen zwischen allen Spalten enthält.

Ich empfehle den Code-Block zur Berechnung der korrigierten Trennschärfe genau zu studieren. Darin finden sich viele der Grundlagen aus Kapitel 2 und 3 wieder:

- Die Erstellung von Vektoren mit der `1:n`-Notation
- Die Negativ-Auswahl von Elementen aus Vektoren mit der `[·]`-Notation
- Die Generierung eines „character“-Vektors mithilfe der Funktion `paste0`
- Die Auswahl von Spalten in einem `data.frame` mit der `[·,·]`-Notation

Wir merken, dass es mühsamer ist, die korrigierte Trennschärfe zu berechnen als die unkorrigierte. Die unkorrigierte Trennschärfe erhalte ich einfach, indem ich einen `data.frame` an die Funktion `cor` übergebe. Ich muss nur einen einzigen Funktionsaufruf—oder eine Zeile Code—investieren. Um jedoch die korrigierte Trennschärfe zu bestimmen, muss ich bei n Items n Mal einen korrigierten Gesamtscore berechnen. Für jedes Item muss ich dann jeweils die Item-Antworten mit diesem korrigierten Score korrelieren. Wenn wir das für jedes Item „händisch“ machen, wäre das sehr aufwendig (beispielsweise könnten wir den Code oben n Mal kopieren und jeweils die Itemnummern anpassen – das wäre sehr fehleranfällig). Einer der Hauptgründe, aus denen wir `R` lernen, ist dass wir uns solche Arbeit nicht machen wollen. Stattdessen wollen wir lernen, wie wir repetitive Arbeiten automatisieren können. Im nächsten Kapitel werden wir Programmierelemente von `R` kennenlernen, die uns ermöglichen, ohne wesentlich mehr Aufwand für beliebig viele Items korrigierte Trennschärfen zu bestimmen. So sparen wir gleichzeitig Aufwand und arbeiten weniger fehleranfällig.

4.1.5 Cronbachs Alpha

Als Nächstes bestimmen wir „Cronbachs Alpha“ als Maß für die interne Konsistenz der Antworten der Schüler/innen. Cronbachs Alpha ist ein Schätzer für die Reliabilität eines Tests. Im Falle eines Leistungstest mit dichotomer Bepunktung gibt es eine Antwort auf die Frage: Haben Kinder, die ein Item richtig beantworten, auch eine erhöhte Wahrscheinlichkeit, andere Items richtig zu beantworten? (Ebenso: haben Kinder, die ein Item falsch beantworten, auch eine erhöhte Wahrscheinlichkeit, andere Items falsch zu beantworten?). Je näher Cronbachs Alpha an 1 ist, desto stärker ist das der Fall – desto stärker ist die interne Konsistenz der Punktwerte. Ein Wert von 0 spricht dafür, dass gar keine Systematik in den Punktzahlen liegt – ob ich viele oder wenig Punkte bekommen habe, ist gänzlich zufällig.

`R` bietet in der Grundversion keine Möglichkeit, Cronbachs Alpha zu bestimmen. Man könnte sich eine eigene Berechnung programmieren, die Cronbachs Alpha umsetzt.[26] Wir machen uns aber zunutze, dass bereits andere `R`-Nutzer Cronbachs Alpha als Funktion umgesetzt haben, und diese in einem *Paket* zur Verfügung gestellt haben. Eine Umsetzung von Cronbachs Alpha findet sich im Paket `psychometric` (Fletcher, 2010). Mit der Funktion `library` kann ich Pakete laden, die nicht zur Grundausstattung von `R` gehören.[27] Voraussetzung ist, dass ich das Paket auf meinem Rechner installiert habe.

```
## Das Paket `psychometric` enthält eine Funktion, die Cronbachs Alpha
## berechnet
library("psychometric")
```

Falls das Paket nicht installiert ist, kann ich es mit dem folgenden Befehl installieren:

```
install.packages("psychometric")
```

Praktischerweise arbeitet die Funktion `alpha` aus dem `psychometric` Paket genau mit dem Standard-Datenformat, das uns vorliegt: Zeilen kennzeichnen Testteilnehmer, Spalten kennzeichnen Items. **Wichtig ist aber nun**: Wir haben soeben den Testscore als zusätzliche Spalte an die Testdatentabelle angehängt. Diese geht aber nicht in die Berechnung von Cronbachs Alpha ein, sondern nur die Punktzahlen für die Items. Deswegen entferne ich die Spalte `score` wie folgt wieder:[28]

```
test_data$score <- NULL

## Prüfe, dass die Spalte wirklich weg ist:
names(test_data)
```

```
[1] "Item_1" "Item_2" "Item_3" "Item_4" "Item_5"
```

[26] Das wäre sogar eine gute Übung. Die Formel findet sich unter https://de.wikipedia.org/wiki/Cronbachs_Alpha

[27] Die Erweiterbarkeit mit Paketen ist eine der großen Stärken von `R`.

[28] Wir haben gelernt, dass wir Variablen mit der Funktion `rm` löschen können. `rm` können wir aber nicht nutzen, wenn wir Spalten aus `data.frames` entfernen wollen. Das liegt daran, dass die Spalte selber keine Variable ist, sondern zu einem `data.frame` gehört. Deswegen muss man Spalten mit dem Befehl `data.frame$spalte <- NULL` entfernen. `NULL` ist in `R` ein Wert, der für „Nicht-Existenz“ steht.

Nachdem wir das Paket `psychometric` geladen haben, können wir Cronbachs Alpha mit der Funktion `alpha` bestimmen:

```
alpha(test_data) # erfordert Laden des Pakets psychometric
```

```
[1] 0.753012
```

4.1.6 Split-Half-Reliabilität

Cronbachs Alpha ist ein Schätzer für die Reliabilität eines Tests.[29] Andere mögliche Schätzer sind die Retest-Reliabilität und die Split-Half-Reliabilität. Diese basieren auf der Berechnung einer Korrelation zwischen zwei Punktwerten. Für die Bestimmung der Retest-Reliabilität lassen wir Testteilnehmer zweimal denselben Test bearbeiten und korrelieren die Punktwerte, die sich zu den zwei Testzeitpunkten ergeben.

Noch leichter ist die Bestimmung der Split-Half-Reliabilität, welche nicht das mehrmalige Bearbeiten desselben Tests erfordert. Dabei teilen wir die Items des Tests in zwei Gruppen ein und bilden Summenwerte für die beiden Testhälften, welche wir dann miteinander korrelieren. Wir müssen dabei berücksichtigen, dass wir nur die Hälfte des Tests zur Schätzung der Reliabilität verwenden. Dies kann mithilfe der *Spearman-Brown-Formel* korrigiert werden.

Die Spearman-Brown-Formel schätzt die Reliabilität eines Tests für den hypothetischen Fall, dass man diesen um einen bestimmten Faktor verlängern würde (d.h. man würde die bestehenden Items replizieren). Man kann sie verwenden, um den Reliabilitätsschätzer einer Split-Half-Korrelation zu korrigieren, da in diesen nur die Hälfte der Items eingehen. Die Spearman-Brown Formel ist diese:

$$r' = \frac{r\,n}{1 + (n-1)r}$$

Hierbei ist r' die um die Testlänge korrigierte Reliabilität. r ist der derzeitige Reliabilitätsschätzer, also beispielsweise die Korrelation von zwei Testhälften. n ist der Faktor, um den der Test hypothetisch verlängert wird.

Für die Schätzung der Split-Half-Reliabilität muss man einen Verlängerungsfaktor von 2 annehmen, da man die Reliabilität nur mit einem halbierten Test schätzt (im Vergleich dazu geht bei der Bestimmung der Retest-Reliabilität zweimal der gesamte Test in die Korrelation ein). Folgender Code berechnet eine Split-Half-Reliabilität:

```
## Wähle (a) die ersten drei und (b) die letzten zwei Items aus:

first_half  <- test_data[, paste0("Item_", 1:3)]
second_half <- test_data[, paste0("Item_", 4:5)]

## Berechne die Korrelation zwischen den beiden Testhälften
```

[29] Eigentlich sprechen wir von der Reliabilität von Testpunkten und nicht von der Reliabilität von Tests.

```r
cor_halfs <- cor(rowSums(first_half), rowSums(second_half))
cor_halfs
```

```
[1] 0.5091751
```

```r
## Führe die Spearman-Brown Korrektur durch:

## Hier ist ein erstes Beispiel einer selbst geschriebenen Funktion. Es
## reicht, das Konzept zur Kenntnis zu nehmen -- es wird in Kap. 5
## wieder aufgegriffen:

## SPEARMAN-BROWN Funktion. Nimmt zwei Argumente an: (a) einen
## Reliabilitäts-Schätzer; (b) einen Verlängerungsfaktor

spearman_brown <- function(reliability, factor) {
    numerator  <- reliability * factor
    denominator <- 1 + (factor-1) * reliability
    corrected_reliability <- numerator / denominator
    return(corrected_reliability)
}

## Rufe die selbst geschriebene SPEARMAN-BROWN Funktion auf. Unser
## initialer Schätzer der Reliabilität ist die Korrelation zwischen den
## zwei Testhälften.  Der Verlängerungsfaktor ist 2, da wir die
## Reliabilität für die doppelte Testlänge schätzen wollen:
split_half <- spearman_brown(cor_halfs, 2)
split_half
```

```
[1] 0.6747727
```

```r
## Vergleiche mit Cronbachs Alpha:
alpha(test_data)
```

```
[1] 0.753012
```

Wie wir sehen, liegt die Spearman-Brown-korrigierte Split-Half-Reliabilität näher an Cronbachs Alpha als die unkorrigierte Korrelation der zwei Testhälften. Das liegt daran, dass die Korrelation der zwei Testhälften die Reliabilität systematisch unterschätzt, da dieser Schätzer nur auf der Hälfte der Items beruht. Es ist sogar so, dass Cronbachs Alpha genau der Mittelwert aller möglichen Spearman-Brown korrigierten Split-Half-Koeffizienten ist.

Alternativ hätten wir auch die Odd-Even-Reliabilität berechnen können, die die Testitems in gerade und ungerade Items einteilt, also hier zwei Testscores einerseits für die Items 1, 3 und 5, und andererseits für die Items 2 und 4 berechnet. Diese lässt sich mit nur wenig Änderungen am Code oben umsetzen – ich schlage vor, dies als Übung zu machen.

4.2 Umgang mit echten Daten

Unser Ziel ist die Auswertung echter Daten von Persönlichkeits-Inventaren wie den BIG-5 und dem Narcissistic Personality Inventory. Leider liegen in echten Daten die Werte oftmals nicht in der Form vor, die wir brauchen. In dem vorherigen Beispiel habe ich die Daten selber generiert und konnte deswegen direkt mit der Analyse starten. Echte Daten jedoch enthalten in Rohform unter Umständen gar nicht die Informationen, die ich benötige oder haben fehlende Werte. Deswegen werden wir uns als nächstes mit den folgenden Themen beschäftigen:

1. Umkodierung von Antworten
2. Invertierung von Antworten
3. Umgang mit fehlenden Werten

4.2.1 Umkodierung von Variablen

Eine wichtige Voraussetzung für eine psychometrische Analyse war im Beispiel oben bereits gegeben: Jeder Wert kodierte genau die Information, die wir brauchten – nämlich ob Schüler/innen eine Aufgabe korrekt gelöst haben oder nicht (dargestellt durch `1` und `0`). In echten Daten muss die relevante Information jedoch häufig erst noch aus den dokumentierten Werten „abgeleitet“ werden. Die Antwort der Schüler/innen im Test könnte beispielsweise ein Kreuz in einem Multiple-Choice-Item sein:

> Aus wie vielen Bundesländern besteht die Bundesrepublik Deutschland?
>
> (1) 14
> (2) 16
> (3) 19
> (4) 21

Ob ein Kreuz bei (1), (2), (3) oder (4) gesetzt wird, ist für die Auswertung nicht von Belang. Relevant ist, ob die Frage richtig beantwortet wurde – wir benötigen also die folgende Umkodierung der Daten:

- `1` → `0`
- `2` → `1`
- `3` → `0`
- `4` → `0`

In psychometrischem Jargon: Für diese Aufgabe ist der Wert `2` der *Schlüssel* (engl.: *key*). Ein Schlüssel kodiert den Eingabewert der richtigen Antwort.[30] Wir lernen jetzt, wie wir solche Umkodierungen in `R` umsetzen. Die Stärke einer Programmiersprache wie `R`: Wenn wir einmal gelernt haben, wie wir für eine Item-Schlüssel-Kombination Daten als richtig und falsch umkodieren, können wir mit nur ein wenig mehr Aufwand diesen Prozess für beliebig

[30] Im Allgemeinen muss ein Schlüssel nicht Korrektheit anzeigen, sondern kann auch Merkmalsausprägung in einem Persönlichkeitsinventar kodieren. Wir werden das im Narcissistic Personality Inventory kennenlernen.

viele Items wiederholen. Das *Narcissistic Personality Inventory* etwa hat 40 Items und wir haben keine Lust, 40 Mal eine Umkodierung „händisch“ neu durchzuführen.

Die Funktion `ifelse`

Mit der Funktion `ifelse` lassen sich Transformationen, die anhand eines Schlüssels Korrektheit kodieren, bequem durchführen. Das folgende Beispiel basiert auf dem obigen Multiple-Choice-Item:

```
# Hypothetische Antworten auf das Bundesland Multiple-Choice-Item:
bundesland_answers <- c(1, 2, 1, 3, 2, 4, 2)
bundesland_key     <- 2

bundesland_score   <- ifelse(test = bundesland_answers == bundesland_key,
                             yes = 1, no = 0)
bundesland_score
```

```
[1] 0 1 0 0 1 0 1
```

Was ist hier passiert? Ich habe im Vektor `bundesland_answers` hypothetische Antworten generiert; die Variable `bundesland_key` enthält den Schlüssel, d.h. die korrekte Antwort. Mithilfe der Funktion `ifelse` gleiche ich die Antworten mit dem Schlüssel ab. `ifelse` nimmt drei Argumente entgegen. Diese heißen `test`, `yes`, und `no`:[31]

- `test`: Vergleicht jede Antwort mit dem Schlüssel, hier: `bundesland_answers == bundesland_key`. Ergebnis dieses Vergleichs ist der folgende logische Vektor (im Allgemeinen kann `test` einen beliebigen logischen Vektor als Argument annehmen):

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

- `yes`: Der Wert, der angenommen werden soll für Elemente, für die der `test` `TRUE` ergab (hier: `1`)
- `no`: Der Wert, der angenommen werden soll für Elemente, für die der `test` `FALSE` ergab (hier: `0`)
  - praktisch: ich muss nicht angeben, welche falschen Werte alle möglich sind; es reicht aus, den richtigen Wert anzugeben, alle anderen sind automatisch falsch

Nach der Umkodierung können wir beispielsweise die Schwierigkeit des Bundesland-Items mit der `mean` Funktion berechnen:

```
mean(bundesland_score)
```

```
[1] 0.4285714
```

In diesem Fall hätten 43% der Testteilnehmer das Bundesland-Item korrekt beantwortet. Diese Information konnten wir aus den ursprünglichen Antwortkategorien 1, 2, 3 und 4 nicht

[31] Wie wir gesehen haben, können wir Argumente in Funktionen per Name und per Position ansteuern.

herleiten.

Die Funktion `ifelse` ist sehr nützlich, mit der wir Antworten umkodieren können. Später lernen wir, wie wir mithilfe von `ifelse` ganze Tests und nicht nur einzelne Items bepunkten können. Bevor wir das jedoch effizient machen können, werden wir im nächsten Kapitel noch ein paar Grundlagen zur Programmierung mit `R` lernen.

4.2.2 Invertierung von Antworten

Eine mögliche Umkodierung von Antworten ist das Abgleichen mit einem Schlüssel, etwa zur Feststellung der Korrektheit von Antworten. Eine weitere häufig auftretende Variante ist die *Invertierung* von Antworten. Betrachten wir folgende zwei Items, die in einer Big-5 Kurzskala den Aspekt Extraversion messen:

1. Ich bin eher zurückhaltend, reserviert.
2. Ich gehe aus mir heraus, bin gesellig.

Beide Items werden auf einer Likertskala mit fünf Abstufungen gemessen, das heißt es werden Punktzahlen von 1 bis 5 vergeben. Das Problem ist, dass in Item 1 ein hoher Punktwert für wenig Extraversion steht, in Item 2 ein hoher Punktwert hingegen für eine hohe Ausprägung in Extraversion. Generell wollen wir einen *Summenwert* berechnen, also einen Wert, der die Extraversion eines jeden Testteilnehmers kodiert – und zwar über beide Items hinweg. Im vorliegenden Fall macht es aber keinen Sinn, die Punktzahlen beider Items zu addieren. Die höchste Ausprägung in Extraversion würde sich dann ergeben, wenn ein Item extravertiert beantwortet wird, aber das andere introvertiert. Damit die Punktzahlen in beiden Items „in dieselbe Richtung“ zu verstehen sind, wollen wir die Antworten auf Item 1 *invertieren*, sodass auch hier eine hohe Punktzahl für eine hohe Merkmalsausprägung in Extraversion steht. Das heißt, wir wollen die folgende Abbildung durchführen:

- `1` $\rightarrow$ `5`
- `2` $\rightarrow$ `4`
- `3` $\rightarrow$ `3`
- `4` $\rightarrow$ `2`
- `5` $\rightarrow$ `1`

Wir könnten dies mit mehrfacher Anwendung von `ifelse` hinbekommen, was jedoch mühsam wäre. Es gibt eine mathematische Umformung, welche wir auch mit nur wenig Code umsetzen können:

$$\text{Invertierter Wert} = \text{Ursprungswert} * (-1) + \text{Höchster Skalenwert} + 1$$

Diese funktioniert, wenn unsere Punktzahlen zwischen 1 und dem höchstmöglichen Skalenwert liegen. Probieren wir es mit ein paar hypothetischen Antworten aus:

```
big5 <- data.frame(item1 = c(2, 3, 2, 1, 4, 2, 1, 5),
                   item2 = c(5, 3, 3, 4, 3, 5, 3, 2))
```

```
## Betrachte den data.frame:
big5
```

```
  item1 item2
1     2     5
2     3     3
3     2     3
4     1     4
5     4     3
6     2     5
7     1     3
8     5     2
```

Wir können uns mit der `cor` Funktion die Korrelation zwischen den zwei Items ausgeben lassen:

```
round(cor(big5), 2)
```

```
      item1 item2
item1  1.00 -0.57
item2 -0.57  1.00
```

Ich habe die Antwortwerte absichtlich so gewählt, dass sich hier ein typisches Muster ergibt: Antworten auf unterschiedlich gepolte Items – die zur selbem Skala gehören – korrelieren typischerweise negativ miteinander. Das heißt: Hohe Antwortwerte im einen Item gehen tendenziell mit niedrigen Werten im anderen Item einher – wenn die unterschiedlich gepolten Items dasselbe Konstrukt erfassen. Durch die Invertierung erhalten wir Daten, die positiv miteinander korrelieren. Folgender Code führt die Invertierung durch:

```
# 5 ist der höchst-mögliche Skalenwert
big5$item1_inv <- big5$item1 * (-1) + 6
```

Schauen wir uns die Daten an, um zu prüfen, ob die Transformation funktioniert hat:

```
big5[, c("item1", "item1_inv")]
```

```
  item1 item1_inv
1     2         4
2     3         3
3     2         4
4     1         5
5     4         2
6     2         4
7     1         5
8     5         1
```

Das hat geklappt! Schauen wir uns nun auch noch einmal die Inter-Itemkorrelationen an:

```
round(cor(big5), 2)
```

```
           item1 item2 item1_inv
item1       1.00 -0.57     -1.00
item2      -0.57  1.00      0.57
item1_inv -1.00  0.57      1.00
```

Wie wir sehen, korrelieren die Spalten `item2` und `item1_inv` genau wie `item2` und `item1` – nur mit positivem Vorzeichen. Ebenfalls interessant: `item1` und `item1_inv` korrelieren perfekt negativ – und das ist genau das, was wir mit der Invertierung erreichen wollten: Einen Punktwert errechnen, der genau in die entgegengesetzte Richtung zu interpretieren ist wie der ursprüngliche Wert.

4.2.3 Umgang mit fehlenden Werten

> Real data have missing values. Missing values are an integral part of the R language. Many functions have arguments that control how missing values are to be handled. – Patrick Burns[32]

Wir lernen nun den rudimentären Umgang mit fehlenden Werten in `R` kennen. Dabei könnte man vermutlich beliebig sophistiziert vorgehen, jedoch werden wir nur einen basalen und wichtigen Spezialfall kennenlernen:

1. Wir wandeln alle Werte in `NA` um, die als fehlend zu klassifizieren sind
2. Danach schließen wir alle Fälle mit fehlenden Werten aus

Für dieses Beispiel laden wir Daten des Narcissistic Personality Inventory (NPI; Raskin & Terry, 1988) ein. Die Daten von mehr als 11,000 Bearbeitungen des NPI sind erfreulicherweise über das „Open Source Psychometrics Project“ unter https://openpsychometrics.org/_rawdata abrufbar. Wenn wir die Daten heruntergeladen haben und die Datei „data.csv“ in unserem RStudio-Projektordner liegt (siehe Anhang), können wir den Datensatz wie folgt einlesen:

```
## Lese Daten ein
npi <- read.csv("data.csv")
```

Wie folgt verschaffen wir uns einen Überblick über die Daten:

```
nrow(npi) # Wie viele Fälle
```

```
[1] 11243
```

```
ncol(npi) # Wie viele Spalten
```

```
[1] 44
```

```
names(npi) # wie heißen die Messvariablen
```

[32] http://www.burns-stat.com/documents/tutorials/why-use-the-r-language/

```
 [1] "score"  "Q1"     "Q2"     "Q3"     "Q4"     "Q5"     "Q6"
 [8] "Q7"     "Q8"     "Q9"     "Q10"    "Q11"    "Q12"    "Q13"
[15] "Q14"    "Q15"    "Q16"    "Q17"    "Q18"    "Q19"    "Q20"
[22] "Q21"    "Q22"    "Q23"    "Q24"    "Q25"    "Q26"    "Q27"
[29] "Q28"    "Q29"    "Q30"    "Q31"    "Q32"    "Q33"    "Q34"
[36] "Q35"    "Q36"    "Q37"    "Q38"    "Q39"    "Q40"    "elapse"
[43] "gender" "age"
```

```
head(npi, n = 3) # Wie sehen die Daten aus
```

```
  score Q1 Q2 Q3 Q4 Q5 Q6 Q7 Q8 Q9 Q10 Q11 Q12 Q13 Q14 Q15 Q16 Q17 Q18 Q19
1    18  2  2  2  2  1  2  1  2  2   2   1   1   2   1   1   1   2   1   1
2     6  2  2  2  1  2  2  1  2  1   1   2   2   2   1   2   2   1   1   2
3    27  1  2  2  1  2  1  2  1  2   2   2   1   1   1   1   1   2   2   1
  Q20 Q21 Q22 Q23 Q24 Q25 Q26 Q27 Q28 Q29 Q30 Q31 Q32 Q33 Q34 Q35 Q36 Q37
1   1   1   1   1   2   2   2   1   2   2   2   1   2   1   1   1   2   2
2   1   2   2   1   2   2   2   2   1   2   2   2   1   2   2   1   2   2
3   1   2   2   2   2   1   2   1   1   2   1   2   2   1   1   2   1   1
  Q38 Q39 Q40 elapse gender age
1   2   1   2    211      1  50
2   2   2   1    149      1  40
3   2   1   2    168      1  28
```

Wir bemerken, dass keine Variable als "Fallnummer" fungiert. Generell ist es **immer** wichtig, dass jeder Datensatz durch eine eindeutige Fallnummer zu identifizieren ist. Da eine solche in den eingelesenen Daten jedoch nicht enthalten ist, fügen wir selber eine Fallnummer hinzu:

```
npi$casenum <- 1:nrow(npi)
```

Eine weitere nützliche Funktion zum Betrachten von `data.frames` ist die Funktion `summary`, die uns einen schnellen Überblick über die Werte in allen Spalten des `data.frames` bietet:

```
summary(npi)
```

Die Funktion `summary` ergibt für jede Spalte eine Tabelle. Wegen der Länge des Outputs von `summary(npi)` ist nicht der gesamte Output im Skript abgebildet.[33] Für die Variable `score` erhalten wir folgende Informationen zum Narzissmus-Gesamtscore:

score
Min. : 0.0
1st Qu.: 7.0
Median :12.0
Mean :13.3
3rd Qu.:18.0
Max. :40.0

[33] Ich schlage vor, die Funktion selber auf den Datensatz aufzurufen, um die Zusammenfassung für alle Spalten zu betrachten.

Identifikation von fehlenden Werten im NPI Datensatz

Das NPI besteht aus 40 Items. Aus dem *Codebuch* des NPI Datensatzes[34] wissen wir, dass Antworten auf die NPI Items die Werte 1 und 2 annehmen können. Die Antwort auf jedes Item des NPI besteht aus einer „forced choice“ zwischen zwei Aussagen; eine davon steht für Narzissmus. Item 1 besteht beispielsweise aus den folgenden beiden Aussagen:

1. I have a natural talent for influencing people.
2. I am not good at influencing people.

Die Wahl von Aussage 1 wird mit 1 kodiert, die Wahl von Aussage 2 mit 2. Nachgeschaltet wird folgende Umkodierung vorgenommen, die die Item-Scores berechnet: Wird die „narzisstische Aussage“ ausgewählt (hier Aussage 1: „I have a natural talent for influencing people.“), wird das Item mit 1 bepunktet. Wird die Aussage gewählt, die nicht für Narzissmus steht (hier Aussage 2: „I am not good at influencing people.“), wird eine 0 vergeben. Wie wir zu Beginn des Abschnitts gelernt haben, könnten wir Item 1 deswegen wie folgt bepunkten:

```
npi$Q1_score <- ifelse(npi$Q1 == 1, 1, 0)
```

Aber Vorsicht: so würden wir einen Fehler machen! Die Spalte `npi$Q1` enthält nicht nur die Werte 1 und 2, sondern auch 0-Werte, wie wir mit dem Befehl `table(npi$Q1)` prüfen können:[35]

```
table(npi$Q1)
```

```
   0    1    2
  17 6872 4354
```

Wie wir sehen, wurden die Antwortkategorien 0, 1 und 2 vergeben. Es kommt sogar 17 Mal die Antwortkategorie 0 vor – **obwohl Antworten nur die Werte 1 und 2 annehmen dürfen**. Wie kommt das? Die Antwort ist: Bei der Bearbeitung des NPI-Fragebogens – welche im Rahmen einer Online-Studie stattfand – konnten Teilnehmer/innen Items unbeantwortet lassen. Fehlende Werte in den Antworten wurden mit einer 0 kodiert.[36]

Ausschluss von Fällen mit fehlenden Werten

[34] Dieses wird gemeinsam mit den Daten des „Open Source Psychometrics Project“ https://openpsychometrics.org/_rawdata runtergeladen.

[35] **Merke**: Es ist wichtig, sich einen Überblick über Daten zu verschaffen und unplausible und fehlende Werte zu identifizieren. Die Funktionen `summary` und `table` sind dabei hilfreich.

[36] Ich halte dies für kein gutes Vorgehen. Der Wert 0 ist nicht ausreichend unterschiedlich von anderen „legalen Werten“ in den anderen Spalten. Der Gesamt-Testscore (`npi$score`) kann beispielsweise wirklich den Wert 0 annehmen, wenn Teilnehmer/innen kein einziges Mal der narzisstischen Aussage zugestimmt haben – und dies kam tatsächlich 73 Mal vor. Der Wert `-99` wäre beispielsweise ein besserer Wert gewesen, um fehlende Werte zu kodieren.

Wir wollen als nächstes alle Fälle ausschließen, bei denen mindestens ein fehlender Wert in den Antworten auf die 40 NPI Items vorliegt, d.h. für mindestens eine der Spalten `npi$Q1`, ..., `npi$Q40` der Wert 0 ist. Erst danach können wir die Itemscores berechnen.

Zu diesem Zweck speichern wir zunächst die Antworten auf die 40 Items und die Fallnummer in einem neuen `data.frame` ab. Anhand dieses `data.frames` werden wir die Fallausschlüsse durchführen:

```
item_responses <- npi[, c("casenum", paste0("Q", 1:40))]
```

Wir können jetzt 0-Werte in `NA` umkodieren, indem wir ein Vorgehen verwenden, das wir in Kapitel 2 für Vektoren kennengelernt haben. Dieses Vorgehen funktioniert bei `data.frames` tatsächlich genauso:[37]

```
item_responses[item_responses == 0] <- NA
```

Für einzelne Spalten kann man mithilfe der Funktion `is.na` überprüfen, ob diese fehlende Werte enthalten. `is.na` ergibt einen logischen Vektor, der kodiert, ob jedes Element des übergebenen Vektors – also etwa eine Spalte, die wir mit der `$`-Notation ausgelesen haben – `NA` ist. Mit diesem Wissen können wir etwa für einzelne Items überprüfen, wie viele Personen keine Antwort angegeben haben:

```
sum(is.na(item_responses$Q1))
```

```
[1] 17
```

```
sum(is.na(item_responses$Q40))
```

```
[1] 34
```

Wichtig: Man **muss** `is.na` verwenden, um zu prüfen, ob Werte `NA` sind; Folgendes geht schief:[38]

```
## Nutze head, um nicht alle 11,000 Vergleiche auszugeben
head(item_responses$Q1 == NA)
```

```
[1] NA NA NA NA NA NA
```

Um insgesamt einen Überblick über die Verteilung der fehlenden Fälle zu erhalten, bietet sich eine erneute Anwendung der Funktion `summary` an. Diese gibt nämlich direkt für jede Spalte eines `data.frames` die Zahl der fehlenden Fälle an. Folgende Information gibt es zum ersten Item:

[37] Der Befehl sieht recht harmlos aus, aber tatsächlich steckt hier etwas mehr drin als wir bislang behandelt haben. Wir nehmen zunächst einmal einfach hin, dass man die Umkodierung von fehlenden Werten in `data.frames` genauso durchführen kann wie in Vektoren. Beachtet, dass hier ein Zugriff auf `data.frames` mit eckigen Klammern stattfindet (siehe Kapitel 3.5; tatsächlich ist dieser Zugriff aber sogar noch etwas spezieller als der in Kapitel 3.5 beschriebene – hier ist das Objekt in den eckigen Klammern eine *Matrix* vom Typ „logical“).

[38] Ein logischer Vergleich mit `NA` ergibt immer `NA`, da beim fehlenden Wert keine Aussage darüber gemacht werden kann, ob er einem anderen Wert entspricht. Man kennt ihn ja nicht. Auch der Befehl `TRUE & NA` ergibt `NA`.

Q1
Min. :1.000
1st Qu.:1.000
Median :1.000
Mean :1.388
3rd Qu.:2.000
Max. :2.000
NA's :17

Jetzt, da wir fehlende Antworten per `NA` als fehlend gekennzeichnet haben, gibt es verschiedene Möglichkeiten, die zugehörigen Fälle auszuschließen. Eine Möglichkeit wäre eine Aneinanderreihung von vielen ODER-Verknüpfungen, an die wir eine Auswahl mit `subset` oder der `[·,·]`-Notation anschließen. Dies könnte wie folgt funktionieren:

```
## Identifiziere Fälle, die in irgendeinem Item einen
## fehlenden Wert haben (hier nur exemplarisch, kein
## legaler R-Code, da Items 4 bis 39 nicht ausgeschrieben
## sind):
irgendwo_na <- is.na(item_responses$Q1) |
    is.na(item_responses$Q2) |
    is.na(item_responses$Q3) |
    ... |
    is.na(item_responses$Q40)

## Negation durchführen, um die Fälle zu erwischen, die
## *keinen* fehlenden Wert enthalten
nirgendwo_na <- !irgendwo_na

## Wähle diese Fälle aus:
item_responses <- item_responses[, nirgendwo_na]
```

Durch die Verknüpfung der ODER-Operatoren werden alle Fälle identifiziert, die mindestens eine fehlende Antwort enthalten. Diese Aneinanderreihung ist jedoch mühselig und fehleranfällig. Diese Arbeit wollen wir uns nicht machen.

Eine weitere Methode, fehlende Werte zu identifizieren nutzt aus, dass die Funktion `rowSums`[39] `NA` ausgibt, wenn mindestens ein Wert aus einer Zeile `NA` enthält – zumindest wenn wir nicht das optionale Argument `na.rm` auf `TRUE` setzen. Dies ist analog zu der Funktion `sum`, die für einen einzelnen Vektor eine Summe bestimmt. Die Funktion `sum` gibt ebenfalls `NA` aus, wenn mindestens ein Element des übergebenen Vektors `NA` ist und `na.rm` nicht auf `TRUE` gesetzt wurde. Die Funktion `rowSums` erweitert also auch im Hinblick auf den Umgang mit fehlenden Werten das Verhalten von `sum` auf alle Zeilen eines `data.frames`. Aus diesem

[39] siehe das ausgedehnte Beispiel zum Einstieg

Grund funktioniert das hier:

```
## Identifiziere Fälle, die in irgendeinem Item einen
## fehlenden Wert haben:
irgendwo_na <- is.na(rowSums(item_responses))
```

Am bequemsten ist es jedoch, wenn wir die Funktion `na.omit` nutzen, die uns einfach so alle Fälle ausschließt, die fehlende Werte enthalten:

```
item_responses <- na.omit(item_responses)
```

Die Funktion `na.omit` gibt einen `data.frame` aus, der keine der Zeilen enthält, in denen mindestens ein `NA`-Wert vorlag. So müssen wir nicht selber die Zeilen identifizieren, die fehlende Werte enthalten.

Vergleichen wir nun den ursprünglichen `data.frame npi` mit der „bereinigten“ Tabelle:[40]

```
nrow(npi)
```

```
[1] 11243
```

```
nrow(item_responses)
```

```
[1] 10440
```

Wie wir sehen, haben wir 803 Fälle wegen fehlender Werte ausgeschlossen. Etwas unschön ist, dass in unseren bereinigten Daten einige Variablen – wie das Geschlecht und das Alter – fehlen. Das liegt daran, dass wir für den Ausschluss von Fällen nur die Item-Antworten berücksichtigt haben, die wir zuvor im `data.frame item_responses` abgespeichert haben. Vergleichen wir:

```
names(npi)
```

```
 [1] "score"   "Q1"      "Q2"      "Q3"      "Q4"      "Q5"      "Q6"
 [8] "Q7"      "Q8"      "Q9"      "Q10"     "Q11"     "Q12"     "Q13"
[15] "Q14"     "Q15"     "Q16"     "Q17"     "Q18"     "Q19"     "Q20"
[22] "Q21"     "Q22"     "Q23"     "Q24"     "Q25"     "Q26"     "Q27"
[29] "Q28"     "Q29"     "Q30"     "Q31"     "Q32"     "Q33"     "Q34"
[36] "Q35"     "Q36"     "Q37"     "Q38"     "Q39"     "Q40"     "elapse"
[43] "gender"  "age"     "casenum"
```

```
names(item_responses)
```

```
 [1] "casenum" "Q1"      "Q2"      "Q3"      "Q4"      "Q5"      "Q6"
 [8] "Q7"      "Q8"      "Q9"      "Q10"     "Q11"     "Q12"     "Q13"
[15] "Q14"     "Q15"     "Q16"     "Q17"     "Q18"     "Q19"     "Q20"
[22] "Q21"     "Q22"     "Q23"     "Q24"     "Q25"     "Q26"     "Q27"
[29] "Q28"     "Q29"     "Q30"     "Q31"     "Q32"     "Q33"     "Q34"
```

[40] Es ist immer wichtig, solche Plausibilitätsüberprüfungen durchzuführen, nachdem man Daten geändert hat.

```
[36] "Q35"     "Q36"     "Q37"     "Q38"     "Q39"     "Q40"
```

Um einen `data.frame` zu erhalten, in dem alle Informationen zu den vollständigen Fällen enthalten sind, machen wir uns zunutze, dass die relevanten Informationen noch im Urspungs-`data.frame` `npi` abgespeichert sind. Wie folgt können wir mit der Funktion `merge` die ursprüngliche Tabelle `npi` mit der um fehlende Fälle bereinigten Tabelle `item_responses` zusammenführen.

```
npi_clean <- merge(npi, item_responses)
```

Wir erhalten in der Variablen `npi_clean` einen Datensatz, der nur Fälle mit vollständigen Antworten enthält – und für diese Fälle auch alle Werte abspeichert. Prüfe:

```
nrow(npi_clean)
```

```
[1] 10440
```

```
names(npi_clean)
```

```
 [1] "Q1"      "Q2"      "Q3"      "Q4"      "Q5"      "Q6"      "Q7"
 [8] "Q8"      "Q9"      "Q10"     "Q11"     "Q12"     "Q13"     "Q14"
[15] "Q15"     "Q16"     "Q17"     "Q18"     "Q19"     "Q20"     "Q21"
[22] "Q22"     "Q23"     "Q24"     "Q25"     "Q26"     "Q27"     "Q28"
[29] "Q29"     "Q30"     "Q31"     "Q32"     "Q33"     "Q34"     "Q35"
[36] "Q36"     "Q37"     "Q38"     "Q39"     "Q40"     "casenum" "score"
[43] "elapse"  "gender"  "age"
```

Die Funktionsweise der Funktion `merge` soll hier nicht tiefergehend betrachtet werden. Es reicht zu wissen, dass sie Fälle aus zwei `data.frames` anhand ihrer Werte zuordnet.[41] Dabei werden Fälle weggelassen, die keinen „Partner“ haben – also hier Fälle, zu denen nur in einer Tabelle eine Fallnummer vorliegt. Die Fälle ohne Partner sind hierbei genau die Fälle, die aus `npi_clean` wegen fehlender Werte ausgeschlossen wurden.

Die Anwendung der Funktion `merge` hat die Reihenfolge unserer Daten durcheinander gebracht. Es ist nicht so wichtig, warum das so ist, aber wir wollen diesen Nebeneffekt wieder rückgängig machen. Deswegen nutzen wir die Funktion `arrange` aus dem Paket `dplyr`, um die Daten wieder anhand der Fallnummer zu sortieren:

```
library("dplyr") # falls noch nicht geladen
npi_clean <- arrange(npi_clean, casenum)
```

Voilá – `npi_clean` ist der Datensatz, mit dem wir nun psychometrische Berechnungen durchführen können.[42] Dabei ist unser nächstes Ziel für alle 40 Items eine dichotome Bepunktung

[41] Hierfür war es wichtig, dass wir vorher eine eindeutige Fallnummer vergeben haben. Anhand dieser Fallnummer können wir nun die Fälle beider Tabellen eindeutig einander zuordnen.

[42] Es ist zu bemerken, dass wir noch nicht alle Variablen auf ihre Plausibilität überprüft haben. Die Spalte `age` enthält ebenfalls noch fehlende sowie auch gänzlich unplausible Werte (etwa 366 oder 509). Auch die Spalte `elapse`, die die Bearbeitungszeit abspeichert, enthält teilweise unplausible Werte; das Maximum der gespeicherten Bearbeitungszeit liegt bei über 40 Jahren. Doch darauf soll erst einmal nicht unser Augenmerk liegen.

durchzuführen. Wir wissen bereits, wie wir das machen könnten, nämlich indem wir mit `ifelse` die Antworten auf jedes Item mit dem Schlüssel abgleichen. Der Schlüssel für den das NPI kodiert für jedes der 40 Items den Wert, der für Narzissmus steht. Dies wäre wie folgt möglich:

```
# Hier kein legaler R-Code, nur exemplarisch:
npi_key <- c(1, 1, ..., 2) # 40 Schlüsselemente

npi$Q1_score <- ifelse(npi$Q1 == npi_key[1], 1, 0)
npi$Q2_score <- ifelse(npi$Q2 == npi_key[2], 1, 0)

...
...
...

npi$Q40_score <- ifelse(npi$Q40 == npi_key[40], 1, 0)
```

Da wir nicht denselben Code – mit leichten Abwandlungen – 40 Mal wiederholen wollen, werden wir in Kapitel 6 lernen, diese Umkodierungen effizient durchzuführen. Anschließend werden wir die psychometrischen Eigenschaften des NPI untersuchen.

4.3 Zusammenfassung

- Wir haben das Standard-Datenformat der psychometrischen Datenauswertung kennengelernt: Zeilen repräsentieren Fälle, Spalten repräsentieren Items
- Wir haben einige grundlegende psychometrische Berechnungen durchgeführt
- Wir haben gelernt, wie wir Antworten umkodieren und invertieren können
- Wir haben gelernt, wie wir Fälle mit fehlenden Werten identifizieren und aus `data.frames` ausschließen können

4.4 Fragen zum vertiefenden Verständnis

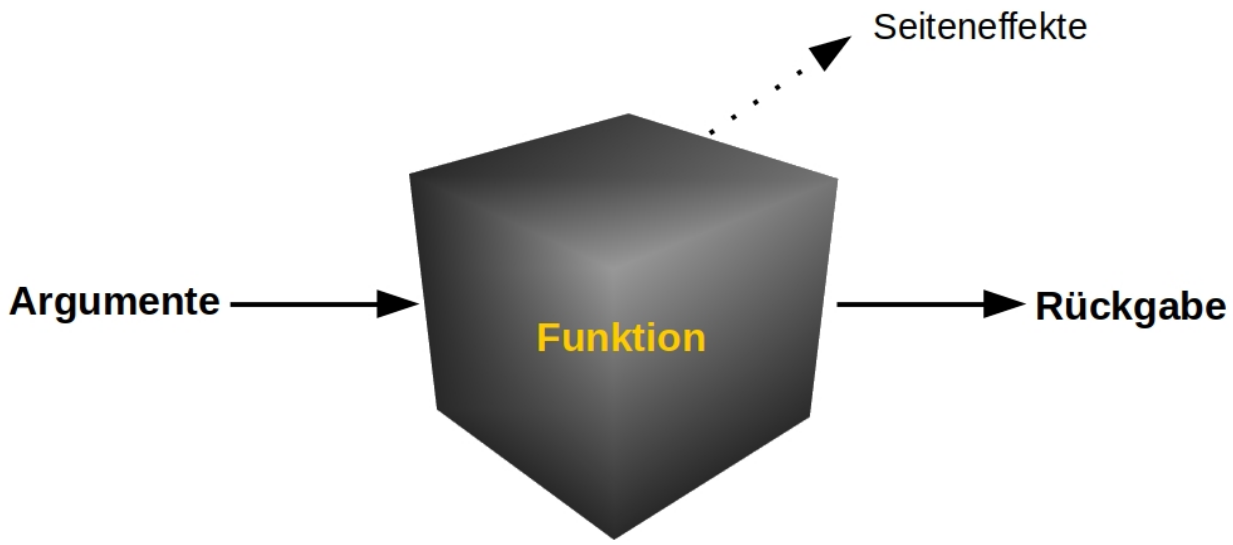
1. Gegeben ist der Antwortvektor `c(1, 2, 2, 1, 4, 5, 2, 2, 2, 3)` und der Schlüssel `2`. Wie kann ich die Item-Schwierigkeit ohne Anwendung der Funktion `ifelse()` bestimmen? Was ist die Item-Schwierigkeit?
2. Gegeben ist der Antwortvektor `c(2, 3, 2, 4, 5, 6, 2, 3)`, der die Antworten auf das Item eines Persönlichkeitsinventars enthält. Die Antworten wurden auf einer Likertskala gegeben, die zwischen 1 und 6 kodiert war. Da das Item negativ gepolt ist, müssen die Antworten vor der Analyse invertiert werden. Was ist der Mittelwert der umgepolten Antworten (d.h. die Item-Schwierigkeit)?

5 Funktionen

Funktionen sind das Herz der R-Maschinerie. Jegliche Befehle, die wir in R durchführen, sind als Funktionen umgesetzt.[43] Wir haben bereits ausgiebig von Funktionen Gebrauch gemacht. Wir haben beispielsweise mit `subset` Daten ausgewählt, mit `table` und `tapply` deskriptive Statistiken angefordert, und mit `cor`, `colMeans` und `psychometric::alpha` psychometrische Berechnungen durchgeführt. Dabei haben wir bereits einige Eigenschaften von Funktionen zur Kenntnis genommen, etwa dass sie Namen haben,[44] Argumente annehmen und ihre Rückgabewerte in Variablen gespeichert werden können. In diesem Kapitel soll nun ein tiefgründigerer Blick auf Funktionen in R gelegt werden. Am Ende des Kapitels lernen wir, unsere eigenen Funktionen zu schreiben.

5.1 Das Black-Box-Modell

Diese Grafik zeigt schematisch die Arbeitsweise von R-Funktionen:



Funktionen nehmen Argumente an, die ihr Verhalten steuern. Sie geben genau ein R-Objekt zurück, das Rückgabewert genannt wird. Wenn der Aufruf einer Funktion außer dieser Rückgabe weitere Auswirkung auf die "Umgebung" hat, nennt man das einen Seiteneffekt. Als R-Neuling ist es manchmal wichtig zu verstehen, was der Unterschied zwischen den Seiteneffekten und der Rückgabe einer Funktion ist.

Die innere Arbeitsweise von Funktionen wird als "Black Box" betrachtet. Wir wissen nicht unbedingt, wie eine Funktion intern funktioniert, was uns aber auch nicht interessiert. So lange uns `mean` den korrekten Mittelwert ausgibt, ist uns egal, wie `mean` den Mittelwert

[43] Technisch gesehen sind sogar einfache Berechnungen wie die Addition oder die Auswahl von Elementen per `[·]` Funktionen. Wir betrachten hier jedoch Funktionen, die dem Schema `Funktionsname(Argument1, Argument2, ...)` folgen.

[44] Interessanterweise können Funktionen sogar anonym sein, also keinen Namen haben. Aber dieser Spezialfall ist für uns erst einmal nicht von Interesse.

berechnet.[45] Bei Funktionen interessiert uns in erster Linie, welche Daten wir einer Funktion als Argumente übergeben müssen und was sie uns dafür in Austausch zurückgeben. Die folgenden Abschnitte befassen sich mit den einzelnen Bestandteilen des „Black-Box-Modells".

5.2 Argumente

Argumente determinieren das Verhalten von Funktionen. Im einfachsten Fall bedeutet das, dass die Elemente eines Vektors, den wir `mean` übergeben, den ausgegebenen Mittelwert determinieren. Wenn wir gar keinen Vektor übergeben, erhalten wir kein Ergebnis, sondern eine Fehlermeldung:

```
mean()
Fehler in mean.default() : Argument "x" fehlt (ohne
Standardwert)
```

Um zu verstehen, wie man mit Argumenten das Verhalten von Funktionen beeinflusst, ist ein Verständnis der folgenden Punkte wichtig:[46]

1. Manche Argumente haben Standardwerte (Englisch: „default values"), die angenommen werden, wenn wir diese Argumente beim Aufrufen der Funktion nicht explizit angeben. Diese Argumente heißen *optionale Argumente*.
2. Wenn Argumente keinen Standardwert haben, **müssen** wir dem Argument einen Wert übergeben, da die Funktion sonst eine Fehlermeldung und kein Ergebnis ausgibt.[47]
3. Argumente haben Namen, die wir verwenden können, um sie in der Form `Argumentname = Wert` zu adressieren.
4. Wenn wir für ein Argument nicht explizit den Namen angeben, wird das Argument nach seiner Position in der Liste aller angegeben Argumente identifiziert.
5. Mit der `R`-Hilfe können wir herausfinden, welche Argumente Funktionen annehmen, wie diese heißen, und welche davon optional oder verpflichtend sind.

Wir werden diese Punkte nun exemplarisch anhand der Funktion `mean` nachvollziehen. Wir wissen bereits, dass `mean` mindestens zwei Argumente annimmt. Beide Argumente haben Namen:

- `x`: der numerische oder logische Vektor, für den der Mittelwert bestimmt werden soll
- `na.rm`: ein ein-elementiger logischer Vektor, der angibt, ob `NA`-Werte von der Berechnung ausgeschlossen werden sollen

Da der Aufruf `mean` ohne Angabe eines numerischen oder logischen Vektors einen Fehler ergibt, ist uns klar, dass `x` kein optionales Argument ist. Wir **müssen** einen Vektor übergeben,

[45] Das ist natürlich zunächst einmal anders, wenn wir selbst neue Funktionen schreiben. Aber auch dann gilt: Wenn ich die Funktion geschrieben habe und der Berechnung vertraue, kann ich später auf sie zugreifen, ohne mir jedes Mal über die interne Funktionsweise Gedanken zu machen. Das kann eine enorme Arbeitserleichterung sein.

[46] Teilweise wurden diese Punkte schon in Kapitel 3 im Abschnitt zur Funktion `subset` angesprochen.

[47] Das ist streng genommen auch wieder eine Vereinfachung. Aber für uns reicht dieses Konzept: Hat ein Argument einen Standardwert, ist es optional; hat ein Argument keinen Standardwert, ist es verpflichtend.

für den wir einen Mittelwert bestimmen können. Wieso `mean` auch sonst aufrufen?

Auf der anderen Seite wissen wir auch, dass wir das Argument `na.rm` nicht unbedingt angeben müssen – das würden wir normalerweise nur dann machen, wenn wir wissen, dass unsere Daten fehlende Werte enthalten. Folgender Aufruf funktioniert nämlich, ohne dass wir dem Argument `na.rm` einen Wert übergeben:

```
mean(1:10)
```

```
[1] 5.5
```

Doch was passiert mit `na.rm`, wenn wir es nicht explizit angeben? Hierbei nehmen wir folgende Regel zur Kenntnis: **Optionale Argumente sind deswegen optional, da ihnen von der Funktion ein Wert zugewiesen wird, wenn wir das Argument nicht selbst übergeben**. Das Argument `na.rm` hat den Standardwert `FALSE`, weswegen `NA`-Werte im Normalfall nicht von der Berechnung ausgeschlossen werden. Stattdessen wird uns `NA` zurückgegeben, wenn `x` mindestens einen fehlenden Wert enthält. Folgende Aufrufe sind demnach äquivalent:

```
mean(1:10)
mean(1:10, na.rm = FALSE)
```

Da wir nicht immer alle optionalen Argumente von Funktionen angeben wollen – stattdessen "vertrauen" wir auf die Standardwerte –, ist es sehr hilfreich, dass wir Funktionsargumente per Namen ansprechen können. So können wir nur genau die optionalen Argumente auswählen, die wir anpassen wollen; für die anderen belassen wir den Standardwert. Das haben wir bei der Funktion `mean` kennengelernt: Nur wenn wir fehlende Werte von der Berechnung des Mittelwerts ausschließen wollen, weisen wir dem Argument `na.rm` den Wert `TRUE` zu:

```
mean(c(1, 2, 3, NA), na.rm = TRUE)
```

```
[1] 2
```

Interessanterweise wissen wir bislang gar nicht, ob wir an dieser Stelle `na.rm` tatsächlich mit Namen ansteuern müssen. In Funktionen können wir Argumente ja anhand ihres Namens *oder* ihrer Position ansprechen. Wenn `na.rm` das zweite Argument wäre, könnten wir auch folgenden Aufruf verwenden:

```
mean(c(1, 2, 3, NA), TRUE)
Fehler in mean.default(c(1, 2, 3, NA), TRUE) :
  'trim' must be numeric of length one
```

Das hat aber nicht funktioniert. Diese Fehlermeldung informiert uns darüber, dass `mean` an der zweiten Position ein Argument mit dem Namen "`trim`" erwartet. Offensichtlich hat `mean` mit `trim` noch ein weiteres optionales Argument, das wir bislang gar nicht kannten. Das heißt für uns: Solange wir für `trim` – was auch immer das ist – keinen Wert angeben, müssen wir `na.rm` per Namen ansprechen.

Wenn wir Argumente mit Namen ansteuern, brauchen wir uns über die Reihenfolge der

Argumente keine Gedanken machen. Das ist oftmals sehr hilfreich. Deswegen funktioniert folgender Aufruf:

```
mean(na.rm = FALSE, x = 1:10)
```

```
[1] 5.5
```

Wir erweitern nun das Beispiel von oben unter Berücksichtigung unseres Wissens über die Vergabe von Namen bei Funktionsargumenten. Folgende Aufrufe sind alle äquivalent:

```
mean(1:10)
mean(1:10, na.rm = FALSE)
mean(x = 1:10)
mean(x = 1:10, na.rm = FALSE)
mean(na.rm = FALSE, x = 1:10)
```

Nicht äquivalent zu den obigen Aufrufen sind jedoch folgende Befehle, die zu Fehlermeldungen führen, da `na.rm` nicht das zweite Argument von `mean` ist:

```
mean(1:10, FALSE)
mean(x = 1:10, FALSE)
```

Wir haben nun gelernt, wie wir Argumente an Funktionen übergeben können. Dieses Wissen hilft uns jedoch nur, wenn wir folgende Frage beantworten können:

Wie finden wir heraus, welche Argumente eine Funktion annimmt?

5.2.1 Die R-Hilfe

In R sind bereits sehr viele statistische und testtheoretische Analysen als Funktionen verfügbar. Wenn diese Funktionen noch nicht in der Basisversion von R enthalten sind, sind sie stattdessen häufig in externen Paketen verfügbar. So können wir beispielsweise ANOVAs, explorative oder konfirmatorische Faktorenanalysen und viele andere Auswertungen durchführen – wenn wir denn wollen. Wir benötigen dabei nur das folgende Wissen:

1. Welche Funktion führt die gewünschte Berechnung aus?
2. Wie können wir diese Funktionen bedienen?

Wir beschäftigen uns im Folgenden mit dem zweiten Punkt.[48] Wir lernen, wie wir uns mit der R-Hilfe einen Überblick über die Arbeitsweise von Funktionen verschaffen können. Probieren wir es exemplarisch für die Funktion `mean` aus:

```
?mean
```

Interessant ist für uns erst einmal der obere Abschnitt “Usage”:

[48] Auch wenn es häufig erst einmal der Knackpunkt ist zu wissen, ob es schon eine Funktion gibt, die das eigene Problem löst, wie diese heißt und in welchem Paket ich sie finde.

```
Usage:

    mean(x, ...)

    ## Default S3 method:
    mean(x, trim = 0, na.rm = FALSE, ...)
```

Wir ignorieren an dieser Stelle, dass uns zwei Varianten angeboten werden, die Funktion `mean` zu nutzen.[49] Wenn in der Hilfe eine „Default S3 method“ angeboten wird, interessiert uns oftmals diese; so ist es auch hier der Fall. An dieser Stelle finden wir die Informationen, die wir benötigen, um die Funktion zu nutzen. Wir sehen

- die Namen der Argumente
- die Reihenfolge der Argumente
- welche Argumente optional sind
- was die Standardwerte der optionalen Argumente sind

Die Standardwerte der optionalen Argumente lassen sich dadurch ablesen, dass in der Argumentliste in der Form `Argumentname = Wert` schon ein Wert angegeben ist. Das Argument `trim` etwa hat den Standardwert `0`. Wie wir bereits wissen, hat das Argument `na.rm` hat den Standardwert `FALSE`. Bei nicht-optionalen Argumenten ist kein Standardwert, sondern nur der Name des Arguments angegeben.

Wenn wir mehr über die Argumente erfahren wollen, müssen wir den Abschnitt “Arguments” der `R`-Hilfe konsultieren. Folgende Punkte interessieren uns bei der Beschreibung:

1. Was ist die inhaltliche Bedeutung eines Arguments?
2. Was für ein `R`-Objekt muss ich übergeben, um ein Argument anzusteuern? (Z.B. einen Vektor, einen `data.frame`, eine `matrix`, eine `list` oder sogar eine Funktion – siehe Kapitel 3: Funktion tapply)

Die Beschreibung der Argumente achtet sehr auf Prägnanz und technische Korrektheit, wie wir am Beispiel der Beschreibung der Argumente der Funktion `mean` erkennen können:

```
Arguments:

    x: An R object.  Currently there are methods for
       numeric/logical vectors and date, date-time and
       time interval objects. Complex vectors are allowed
       for ‘trim = 0’, only.

    trim: the fraction (0 to 0.5) of observations to
       be trimmed from each end of ‘x’ before the mean
       is computed.  Values of trim outside that range
       are taken as the nearest endpoint.
```

[49] Viele Funktionen können auf verschiedene Arten aufgerufen werden. Das heißt: Sie können mit unterschiedlichen `R`-Objekten als Eingabe genutzt werden.

```
     na.rm: a logical value indicating whether 'NA' values
        should be stripped before the computation proceeds.
```

Die R-Hilfe ist also hilfreich, aber nicht immer ganz leicht zu nutzen. Oftmals ist eine weitere Google-Recherche oder das Nachfragen bei einer Freundin oder einem Freund sinnvoll, wenn man die Nutzung einer Funktion meistern will. Mehr Hilfe zur Nutzung einer Funktion finden wir im Abschnitt "Examples" der R-Hilfe. Hier können wir am konkreten Beispiel betrachten, wie die Funktion angewendet werden kann. Wenn wir Glück haben, ist genau das dabei, was wir brauchen. Der Code ist dabei so gewählt, dass man ihn per Copy & Paste auch selbst in der Konsole ausführen kann. Bei `mean` finden wir etwa den folgenden Beispiel-Code:

```
Examples:
     x <- c(0:10, 50)
     xm <- mean(x)
     c(xm, mean(x, trim = 0.10))
```

5.2.2 Namenlose Argumente

In der Einführung in die Arbeitsweise von Funktionsargumenten habe ich behauptet, dass Argumente Namen haben. Wie fast jede allgemeine Regel hat auch diese Regel Ausnahmen – Argumente haben nämlich gar nicht immer einen Namen. Das ist für uns zwar nicht ganz so wichtig, aber wir können es hier zur Kenntnis nehmen. Wir haben sogar schon mit Funktionen gearbeitet, die namenlose Argumente annehmen können. Das ist beispielsweise immer dann notwendig, wenn Funktionen eine beliebige Anzahl von Argumenten annehmen. Die Funktion `c` nimmt beliebig viele Vektoren entgegen, die dann zu einem Vektor zusammengefügt werden. Auch andere Funktionen wie `table` – die beliebig viele Vektoren zur Erstellung von Kreuztabellen annimmt – und `dplyr::arrange` – die beliebig viele Kriterien zur Sortierung eines `data.frames` annimmt – haben unbenannte Argumente. In der R-Hilfe ist dies oftmals an dem Platzhalterargument "..." (lies: *ellipsis*) zu erkennen, siehe:[50]

```
?c

Usage:

     ## S3 Generic function
     c(...)

     ## Default S3 method:
     c(..., recursive = FALSE, use.names = TRUE)
```

[50] Wir nehmen interessiert zur Kenntnis, dass `c` zwei Argumente mit Standardwerten hat: `recursive` und `use.names`. Mit diesen Argumenten haben wir uns bislang nicht beschäftigt und das werden wir auch weiterhin nicht tun. Oftmals "vertraut" man den Standardwerten, wobei man damit früher oder später auch mal auf die Nase fallen wird.

```
Arguments:

     ...: objects to be concatenated.
```

5.3 Rückgabewerte

Der Rückgabewert einer Funktion ist das R-Objekt, das die Funktion ausgibt. Jede Funktion hat einen Rückgabewert. Die Funktion `mean` gibt beispielsweise einen ein-elementigen Vektor aus, der das arithmetische Mittel des Eingabevektors repräsentiert. Die Funktion `subset` gibt immer einen `data.frame` aus – wie der aussieht, bestimmen die Argumente, die wir der Funktion übergeben.

Das Verständnis von Rückgabewerten führt uns etwas tiefer in die Innereien der R-Programmiersprache. Betrachten wir im folgenden Beispiel die Funktion `t.test`, die einen *t*-Test durchführt und dabei die Mittelwerte zweier Vektoren vergleicht. Ich verwende den NPI-Datensatz aus Kapitel 4 und vergleiche den mittleren Narzissmus-Gesamtwert (Spalte `score`) zwischen weiblichen und männlichen Testnehmenden (Spalte `gender`; Kodierung: männlich = 1, weiblich = 2).[51]

```
t.test(npi$score[npi$gender == 1],
       npi$score[npi$gender == 2])
```

Wir erhalten folgende Ausgabe in der Konsole:

```
    Welch Two Sample t-test

data:  npi$score[npi$gender == 1] and npi$score[npi$gender == 2]
t = 13.249, df = 10681, p-value < 2.2e-16
alternative hypothesis: true difference in means is not equal to 0
95 percent confidence interval:
 1.801341 2.426906
sample estimates:
mean of x mean of y
 14.19595  12.08183
```

Hier werden uns alle relevanten Aspekte der *t*-Test-Berechnung ausgegeben. Ein erster interessanter Hinweis ist, dass kein klassischer *t*-Test, sondern ein "Welch Two Sample t-test" gerechnet wurde.[52] Weiterhin werden uns t-Wert, Freiheitsgrade, p-Wert ("`p-value < 2.2e-16`" bedeutet, dass der p-Wert so klein ist, dass vor dem ersten Wert hinter dem

[51] An dieser Stelle ist es sinnvoll, noch einmal zu rekapitulieren, wie hier die Narzissmuswerte der Männer und Frauen ausgewählt werden.

[52] Setzt man das Argument `var.equal` auf `TRUE`, wird ein klassischer t-Test gerechnet (Standardwert: `FALSE`). Es ist etwas ironisch, dass die Funktion `t.test` keinen klassischen t-Test rechnet. Wenn man das erwartet, könnte man an dieser Stelle auf die Nase fallen, wenn man einfach auf die Standardwerte der Funktion vertraut. Mehr Informationen zum Welch-t-Test finden sich hier: https://en.wikipedia.org/wiki/Welch%27s_t-test.

Komma, der **nicht** Null ist, mindestens 16 Nullen stehen), das 95%-Konfidenzintervall der Differenz der mittleren Werte, sowie die Mittelwerte selbst ausgegeben. Insgesamt kann man interpretieren, dass Männer im Mittel signifikant höhere Narzissmuswerte aufweisen als Frauen. Aber dieser Befund ist für uns in diesem Fall uninteressant – wir wollen uns ja mit dem Rückgabewert befassen. Was für ein R-Objekt wurde uns denn ausgegeben?

Was in der Konsole angezeigt wird, stellt nicht direkt das R-Objekt dar, das die Funktion `t.test` ausgibt. Es handelt sich hierbei um eine etwas lesefreundlichere Zusammenfassung des Tests. Um das tatsächlich ausgegebene Objekt zu inspizieren, kann ich die Funktion `str` verwenden. Dazu speichere ich die Ausgabe des t-Tests zunächst in einer Variablen ab:

```
test <- t.test(npi$score[npi$gender == 1],
               npi$score[npi$gender == 2])
str(test)
```

```
List of 9
 $ statistic  : Named num 13.2
  ..- attr(*, "names")= chr "t"
 $ parameter  : Named num 10681
  ..- attr(*, "names")= chr "df"
 $ p.value    : num 9.4e-40
 $ conf.int   : atomic [1:2] 1.8 2.43
  ..- attr(*, "conf.level")= num 0.95
 $ estimate   : Named num [1:2] 14.2 12.1
  ..- attr(*, "names")= chr [1:2] "mean of x" "mean of y"
 $ null.value : Named num 0
  ..- attr(*, "names")= chr "difference in means"
 $ alternative: chr "two.sided"
 $ method     : chr "Welch Two Sample t-test"
 $ data.name  : chr "npi$score[npi$gender == 1] and npi$score[npi$gender == 2]"
 - attr(*, "class")= chr "htest"
```

Das von `t.test` ausgegebene Objekt ist eine "List of 9", also eine Liste mit 9 Einträgen. Listen sind sehr allgemeine Datencontainer, die Elemente von beliebigem Typ und beliebiger Anzahl beinhalten können.[53] Listen stellen eine wichtige Datenstruktur in R dar – vielleicht ist es sogar die wichtigste Datenstruktur, da `data.frames` spezielle Listen sind, in denen jeder Eintrag (d.h., jede Spalte) ein Vektor gleicher Länge ist. Die Elemente der Liste können benannt sein, wie es bei der Rückgabe von `t.test` der Fall ist. In diesem Fall kann ich, wie wir es von der Spaltenauswahl in `data.frames` kennen, mit der `$`-Notation auf einzelne Elemente zugreifen:

```
test$statistic # t-Wert als ein-elementiger Vektor
```

```
       t
13.24906
```

[53] Ein Eintrag einer Liste könnte beispielsweise eine andere Liste sein.

```
test$alternative # ein- oder zweiseitiger Test?
```

```
[1] "two.sided"
```

```
test$parameter # Freiheitsgrade
```

```
     df
10681.4
```

Es ist nicht unüblich, dass komplexere statistische Berechnungen eine Liste als Ausgabeobjekt ergeben. Listen sind dann sinnvoll, wenn während der Berechnung mehrere Werte anfallen und es nützlich ist, diese an den Nutzer zurückzugeben. Im Falle des t-Tests interessieren uns etwa die Freiheitsgrade, der t-Wert und der p-Wert.

5.4 Seiteneffekte

Jede Funktion hat genau einen Rückgabewert, also genau ein R-Objekt, das von der Funktion zurückgegeben wird. Jegliche Auswirkungen, die eine Funktion darüber hinaus hat – außerhalb der inneren Arbeitsweise –, werden Seiteneffekte genannt. Beispielsweise war die Konsolen-Ausgabe der Funktion `t.test` im oberen Fall ein Seiteneffekt. Wenn ich die Funktion `t.test` aufrufe, wird mir eine Zusammenfassung des Tests in der Konsole ausgeben, die als Mensch etwas einfacher zu verarbeiten ist als die Liste, die die "tatsächliche" Rückgabe darstellt. Das Zeichnen von Abbildungen ist auch als Seiteneffekt zu verstehen. Wenn wir beispielsweise `hist(npi$score)` aufrufen, wird uns ein Histogramm der Narzissmuswerte des NPI angezeigt. Dieses Histogramm ist ein Seiteneffekt und nicht die Ausgabe der Funktion `hist`; Interessierte sind aufgefordert herausfinden, was deren tatsächliche Rückgabe ist.

Ich werde die Diskussion von Seiteneffekten bei dieser kurzen und oberflächlichen Einführung belassen. Manchmal ist die Unterscheidung in Seiteneffekte und Rückgabe sinnvoll; wenn wir etwa die Ergebnisse einer Funktion weiter verwenden möchten, ist es wichtig zu wissen, welche Datenstruktur die Funktion ausgibt und wie wir auf die ausgegebenen Werte zugreifen können. Um Power-Analysen für t-Tests zu simulieren (eine Einführung in solche Simulationen wird in Kapitel 7 folgen), ist es beispielsweise nötig, die exakten p-Werte aus der Ausgabe der Funktion `t.test` auszulesen. In dem Fall werden wir die Funktion `t.test` gegebenenfalls 10,000 Mal aufrufen und können es uns nicht leisten, den p-Wert jedes Mal aus der Konsolen-Ausgabe abzulesen. Stattdessen wollen wir den Prozess der p-Wert-Extraktion automatisieren; und genau für solche Automatisierungen lernen wir das Programmieren.

5.5 Selbst geschriebene Funktionen

Das Schreiben eigener Funktionen sollte früher oder später zum Repertoire eines R-Nutzers gehören. Mit selbst geschriebenen Funktionen können wir häufig durchgeführte Berechnungen abstrahieren und beliebig oft durchführen. Betrachten wir den folgenden Code:

```
(0.23 * 2) / (1 + (2 - 1) * 0.23)
(0.47 * 3) / (1 + (3 - 1) * 0.23) # copy-paste Fehler
(0.68 * 4) / (1 + (4 - 1) * 0.68)
```

Erkennt ihr, was berechnet wird? Falls nicht, betrachtet diesen Code:

```
spearman_brown(0.23, 2)
spearman_brown(0.37, 3)
spearman_brown(0.68, 4)
```

In der ersten Variante sind nur ein paar Zahlen und arithmetische Operationen zu sehen, die Semantik der Berechnung ist jedoch vollkommen unklar. Durch das Copy-Pasten des Codes ist mir sogar ein Fehler unterlaufen, denn ich habe in der zweiten Zeile einmal vergessen den Wert 0.23 durch 0.47 auszutauschen – solche Fehler passieren häufig und sind sehr schwierig zu entdecken (siehe Li, Lu, Myagmar, & Zhou, 2006). Durch die Nutzung der Funktion, die ich in Kapitel 4 definiert habe, ist der Code lesbarer geworden und einfach interpretierbar: Ich möchte drei Reliabilitätsschätzer um die Faktoren 2, 3 bzw. 4 korrigieren. Sofern ich bei der Definition meiner Funktion keinen Fehler gemacht habe, sind diese Aufrufe auch robuster gegenüber Copy-Paste-Fehlern.

Mit eigenen Funktionen folgen wir dem Programmierer-Credo „*do not repeat yourself*“. Wenn wir einmal Code zur Lösung eines Problems geschrieben haben, möchten wir denselben Code nicht noch einmal schreiben, um ein gleiches bzw. ähnliches Problem zu lösen. Eigene Funktionen helfen uns effizienter – man könnte sogar sagen: fauler – zu arbeiten. Außerdem führen sie zu besser lesbarem Code, denn Funktionsnamen[54] kommunizieren die Intention von Code deutlich besser als das reine Aneinanderreihen von Zahlen, Operatoren und Variablen.

5.5.1 Definition der eigenen Funktion

Erinnern wir uns an die Spearman-Brown-Funktion, die ich in Kapitel 4 definiert habe:

```
1  spearman_brown <- function(reliability, factor) {
2      numerator  <- reliability * factor
3      denominator <- 1 + (factor-1) * reliability
4      corrected_reliability <- numerator / denominator
5      return(corrected_reliability)
6  }
```

Um die Funktion zu definieren, führe ich diese sechs Zeilen Code einfach in der `R`-Konsole aus. In RStudio reicht es sogar, STRG-Enter zu drücken, wenn sich mein Cursor in Zeile 1 befindet. In dieser Zeile beginnt die Definition der Funktion: Ich erstelle eine Variable, der ich mit “`<-`” eine Funktion zuweise. Die Funktion `spearman_brown` wird also mit der Funktion `function` erstellt (kein Witz). Die Funktion `function` nimmt die Argumente entgegen, die auch meine neu definierte Funktion `spearman_brown` annehmen soll. Die Parameter, die bei

[54] Wie bei Variablen ist auch bei Funktionen eine sinnvolle Benennung unerlässlich.

der Spearman-Brown-Korrektur eine Rolle spielen, sind der Reliabilitätsschätzer und der Korrekturfaktor. Aus diesem Grund werden die zwei Argumente `reliability` und `factor` definiert.

Auf die Definition der Argumente folgt der "Körper" (engl.: *body*) der Funktion. Der Körper führt die gewünschte Berechnung durch und verwendet dabei die Funktionsargumente als Variablen. Funktionsargumente sind also nichts anderes als Variablen, die im Innern einer Funktion leben. Der Körper der Funktion wird in geschwungenen Klammern `{·}` eingeschlossen. Solche Klammern bilden einen abgeschlossenen Block von `R`-Code; sie werden uns auch im nächsten Kapitel bei der Verwendung von Schleifen wieder begegnen.

In Zeile 5 wird mit der Funktion `return`[55] angegeben, dass die Variable `corrected_reliability` von der Funktion zurückgegeben werden soll. Das heißt: das `R`-Objekt, das innerhalb der Funktion in die Variable `corrected_reliability` geschrieben wird, ist der Rückgabewert der Funktion.

5.5.2 Lokale Variablen

Werfen wir noch einmal einen Blick auf den Körper der Funktion `spearman_brown`. Wir sehen hier bereits gewohnten `R`-Code – nichts Besonderes: Variablen werden geschrieben und Berechnungen werden durchgeführt. Ein wichtiger Punkt an diesen Berechnungen ist jedoch, dass sie nur innerhalb der Funktion stattfinden und keine Wirkungen "nach außen" haben. Was heißt das konkret? Beispielsweise sind alle Variablen, die innerhalb der Funktion definiert werden, sogenannte *lokale* Variablen. Im Gegensatz zu den lokalen Variablen stehen globale Variablen. Global waren alle Variablen, die wir bislang per `<-` definiert haben. Solche globalen Variablen werden uns in RStudio im Panel oben rechts unter "Global Environment" angezeigt.

Lokale Variablen sind außerhalb der Funktion nicht sichtbar und verschwinden nach Aufruf der Funktion wieder.[56] Es ist also nicht so, dass die Variablen `numerator`, `denominator` und `corrected_reliability` in die globale `R`-Umgebung geschrieben werden, wenn ich die Funktion `spearman_brown` aufrufe. Das ist extrem nützlich: Ich habe die Variablen `numerator` und `denominator` nur definiert, damit mein Code gut lesbar ist und die einzelnen Code-Zeilen nicht zu lang werden.[57] Am Ende interessiert mich aber nur der Spearman-Brown-Schätzer als Ergebnis der Berechnung; Variablen, die ich als Zwischenergebnisse abspeichere, interessieren mich hingegen nicht. Wenn ich eine Funktion schreibe, bleiben solche Hilfsvariablen verborgen und nur der Rückgabewert dringt nach außen. Ich kann den Rückgabewert abfangen, indem ich ihn in einer Variablen abspeichere:

[55] Es ist nicht unbedingt nötig, `return` zur Definition von Rückgabewerten zu verwenden. Aber ich mache es so und verrate die Alternative auch nicht.

[56] Umgekehrt gilt hingegen: Innerhalb einer Funktion kann man auf Variablen der globalen `R`-Umgebung zugreifen. Ignoriert das jedoch bitte. Werte, mit denen eine Funktion arbeitet, sollten der Funktion per Argument übergeben werden.

[57] Es ist guter Stil, die Menge von Code pro Zeile zu begrenzen. Eine Daumenregel ist die Verwendung von nicht mehr als 80 Zeichen Code pro Zeile. Dafür kann es beispielsweise helfen, Zwischenberechnungen in Variablen abzuspeichern. Eigene Funktionen helfen ebenfall dabei, Code lesbar zu gestalten.

```
split_half_correct <- spearman_brown(0.63, 2)
```

Durch diesen Befehl wird eine korrigierte Reliabilität in eine Variable mit dem Namen `split_half_correct` geschrieben; dass der Rückgabewert der Funktion innerhalb der Funktion in einer Variablen mit dem Namen `corrected_reliability` abgespeichert ist, ist für das ausgegebene Objekt nicht von Bedeutung.

Argumente sind ebenfalls lokale Variablen in der Funktion. Wenn wir einer Funktion also ein Argument übergeben, definieren wir damit eine lokale Variable mit dem Namen des Arguments innerhalb der Funktion. Daraus können wir beispielsweise schließen, dass im Code der Funktion `mean` irgendwo eine Variable mit dem Namen `na.rm` verwendet wird.

5.5.3 Optionale Argumente

Bei der Definition einer Funktion können wir Standardwerte vergeben und somit optionale Argumente definieren. Wenn wir beispielsweise davon ausgehen, dass wir die Spearman-Brown-Korrektur meistens verwenden, um eine Split-Half-Korrelation zu korrigieren, könnten wir das Argument `factor` per default auf 2 setzen. Das funktioniert wie folgt:

```
spearman_brown <- function(reliability, factor = 2) {
    ...
}
```

Diese Schreibweise kennen wir schon vom Aufruf von Funktionen, wenn wir die Argumente mit Namen ansprechen. In diesem Fall können wir die Funktion `spearman_brown` auch wie folgt äquivalent verwenden:

```
split_half_correct <- spearman_brown(0.63)
split_half_correct <- spearman_brown(0.63, 2)
```

5.5.4 Wann schreibe ich meine eigene Funktion

Wie wir in den letzten Abschnitten gesehen haben, ist die technische Definition einer Funktion keine schwierige Sache. Schwieriger ist häufiger die Antwort auf die Frage, wann ich tatsächlich eine Funktion schreiben will. Darauf gibt es keine einzige richtige Antwort, und am Ende muss jeder für sich selbst entscheiden. Generell ist eine sinnvolle Daumenregel, dann eine Funktion zu schreiben, wenn man denselben Code mehrfach geschrieben hat. Häufiges Copy & Paste kann da ein guter Indikator sein. In dem Fall solltet ihr identifizieren, welche Details sich jeweils bei den verschiedenen Varianten des Codes geändert haben, und diese in Argumente umwandeln.

Um zu erkennen, wann Funktionen nützlich sind und welche Variablen man als Argumente umsetzen will, bedarf es sicherlich einiger Erfahrung mit `R`. Mein Tipp für Anfänger ist deswegen vor allem: Erst mal einfach "coden" – später Funktionen schreiben. Mit mehr Erfahrung kann sich diese Vorgehensweise ändern. Ich überlege oftmals schon in der Planungsphase

eines Projekts, welche Funktionen sich sinnvollerweise anbieten und wie diese zusammen arbeiten sollten. Aber das Vorgehen hängt auch stark von der Art des Projekts ab. Wenn ich bloß Daten einlese und einen t-Test oder eine ANOVA rechne, muss ich dafür keine Funktion schreiben. Wenn ich hingegen viel mit Daten an sich arbeite – also oft Daten auswähle, transformiere und aus bestehenden Werten neue Werte ableite – machen eigene Funktionen oftmals mehr Sinn.

5.6 Fragen zum vertiefenden Verständnis

1. Was ist der Rückgabewert der Funktion `str`?
2. Was ist der Rückgabewert der Funktion `hist`?
3. Kann ich mit der Funktion `spearman_brown` gleichzeitig mehrere Reliabilitätsschätzer korrigieren? (Code inspizieren → überlegen → ausprobieren)

6 Schleifen

Ein Hinweis zu Beginn: Dieses Kapitel nutzt den `[[·]]`-Zugriff auf Spalten in `data.frames`. Wer damit noch nicht vertraut ist, sollte sich vor dem Weiterlesen zunächst den kurzen Abschnitt zum `[[·]]`-Zugriff in Kapitel 3 ansehen.

Schleifen spielen in allen Programmiersprachen eine wichtige Rolle. Sie erlauben uns, eine Aufgabe mehrfach durchzuführen, ohne dass wir den Code für die Aufgabe mehrfach schreiben müssen. Beispielsweise müssen wir Umkodierungen von Items (vgl. Kapitel 4) nicht für jedes einzelne Item neu eingeben – d.h. fehleranfällig copy-pasten –, sondern können sie mithilfe einer Schleife für alle Items auf einmal durchführen. Wie auch eigene Funktionen helfen uns Schleifen bei der Automatisierung unserer Arbeit. Sie helfen uns, `R` als Programmiersprache zu nutzen.

Im Allgemeinen und in `R` im Speziellen gibt es mehrere schleifenartige Gebilde; in diesem Kapitel lernen wir die wichtige `for`-Schleife kennen.[58] Das logische Prinzip einer `for`-Schleifen ist recht simpel: Sie führt einen Code-Block mehrfach durch. In der Regel wird in den verschiedenen Durchläufen der Schleife variiert, auf welche Daten – also etwa auf welche Items eines Tests – zugegriffen wird, damit nicht in jedem Durchgang einfach dasselbe passiert. Dies ist die Syntax einer `for`-Schleife:

```
for (Schleifenvariable in vector) {
  ... # hier steht beliebiger R-Code
}
```

Das sieht erst einmal etwas beunruhigend aus. Gehen wir die einzelnen Bestandteile der Schleife einmal durch; danach schauen wir uns eine `for`-Schleife in Aktion an.

Den Anfang der Schleife definieren wir mit dem Schlagwort `for`. Die eigentliche Musik spielt in der darauf folgenden Klammer `(Schleifenvariable in vector)`. Dabei ist `vector` ein beliebiger `R`-Vektor. Von der Länge dieses Vektors hängt ab, wie oft der Code im Körper der Schleife – eingeschlossen in den geschwungenen Klammern `{·}` – ausgeführt wird. Wir könnten der `for`-Schleife beispielsweise einen der folgenden Vektoren übergeben:

```
c(83, 45, 12, -99) # Die Schleife würde 4x laufen
c("Cronbach", "Spearman", "Brown")  # 3x
1:10 # 10x
paste0("item", 1:50) # 50x
```

Wäre `vector` einer dieser vier Vektoren, würde die Schleife viermal, dreimal, zehnmal, oder 50 Mal durchgeführt werden. Um zu verstehen, warum es überhaupt Sinn macht, denselben Code-Block mehrfach durchzuführen, betrachten wir zusätzlich den Ausdruck `Schleifenvariable`, der die Magie der `for`-Schleife offenbart: **Der `Schleifenvariable` wird in jedem Schleifendurchlauf schrittweise das nächste Element von `vector` zugeordnet.** Auf die `Schleifenvariable` können wir also im Körper der Schleife zugreifen

[58] Wer nach dem Durcharbeiten des Kapitels noch nicht genug von Schleifen hat, kann sich mithilfe einer Google-Suche mit der `while`-Schleife vertraut machen.

und ihr Inhalt ändert sich in jedem Durchlauf der Schleife. Betrachten wir folgendes Spielzeug-Beispiel, das das Konzept der `for`-Schleife verdeutlicht:

```
for (name in c('Cronbach', 'Spearman', 'Brown')) {
  print(name)
}
```

```
[1] "Cronbach"
[1] "Spearman"
[1] "Brown"
```

Die Funktion `print` ist die explizite Anweisung, ein `R`-Objekt in der Konsole auszugeben. Wir sehen, dass uns die Schleife in ihren drei Durchläufen drei verschiedene Texte ausgibt, nämlich nacheinander den Inhalt des Vektors `c('Cronbach', 'Spearman', 'Brown')`. Da `print` auf die Variable `name` angewendet wurde, sehen wir, dass `name` ihren Inhalt in jedem Durchlauf der Schleife geändert hat.

Wir stellen fest, dass `for`-Schleifen folgende Eigenschaften haben:

- Sie führen einen Code-Block genauso oft aus, wie ein übergebener Vektor lang ist.
- Eine Schleifenvariable nimmt in jedem Durchlauf den nächsten Wert des übergebenen Vektors an.
- Im Körper der Schleife können wir auf die Schleifenvariable zugreifen, um in jedem Durchlauf andere Berechnungen durchzuführen.

Das ist tatsächlich schon alles! Im Folgenden lernen wir zwei konkrete Anwendungen von `for`-Schleifen kennen.

6.1 Sequentielle Bepunktung von Testitems

In Kapitel 4 haben wir gelernt, wie wir mithilfe eines Schlüssels Testfragen aus einem psychologischen Test bepunkten können. Im NPI hatten wir den Fall, dass jedes Item aus einer narzisstischen und einer nicht-narzisstischen Aussage bestand; wir haben für ein Item genau dann einen Punkt vergeben, wenn die narzisstische Aussage gewählt wurde.

Wir wollen im Folgenden diese Bepunktung mithilfe einer `for`-Schleife automatisieren, das heißt auf einmal für alle 40 Items des NPI durchführen. Dafür benötige ich für jedes Item des NPI den Schlüssel, den ich dem Codebuch entnehmen kann. Wir übertragen die 40 Schlüssel zunächst manuell in einen Vektor:

```
## Schlüssel aller 40 Items in einen Vektor
keys <- c(1, 1, 1, 2, 2, 1, 2, 1, 2, 2, 1, 1, 1, 1, 2, 1, 2, 2, 2, 2,
          1, 2, 2, 1, 1, 2, 1, 2, 1, 1, 1, 2, 1, 1, 2, 1, 1, 1, 1, 2)
```

Als Nächstes führe ich mit einer `for`-Schleife die Bepunktung aller Items durch. In diesem Code nehme ich an, dass die NPI-Antwortdaten schon eingelesen wurden und die Datenbereinigung aus Kapitel 4 durchgeführt wurde, mir also ein `data.frame` mit Namen `npi_clean` vorliegt, der alle Fälle ohne fehlende Antworten enthält.

```r
## Die Variable `npi_clean` enthält die Antworten für das NPI, siehe
## Kapitel 4

# for-Schleife für die Kodierung:
for (i in 1:40) {
    # 1. Wähle Spaltenname des i'ten Items aus:
    colname <- paste0("Q", i)
    # 2. Wähle aus Spalte die Antworten aus:
    ith_item <- npi_clean[[colname]]
    # 3. Führe Umkodierung durch:
    narcissistic_response <- ifelse(ith_item == keys[i], 1, 0)
    # 4. Erstelle Namen für neue Spalte:
    new_colname  <- paste0("coded", colname)
    # 5. Hänge kodierte Werte an data.frame an:
    npi_clean[[new_colname]] <- narcissistic_response
}
```

Der erste Befehl im Körper der Schleife generiert mit der Funktion `paste0` den Namen der Spalte, der adressiert werden soll. Im ersten Durchgang der `for`-Schleife wird also die Spalte `Q1` adressiert, da die Schleifenvariable `i` den ersten Wert des Vektors `1:40` angenommen hat. Dann folgen `Q2`, `Q3` und so weiter. Als Namen von Schleifenvariablen werden häufig kurze Namen wie `i` oder `j` verwendet, insbesondere wenn es sich um *Indexvariablen* handelt, sie also eine numerische Sequenz der Form `1:n` durchlaufen.

Der zweite Befehl im Körper der Schleife wählt die zuvor definierte Spalte von `npi_clean` als Vektor aus. Ich benutze dabei die `[[·]]`-Notation, da sie mir erlaubt, eine Variable vom Typ `character` in den Klammern zu übergeben. Der folgende Aufruf mit der `$`-Notation würde nicht funktionieren: `npi_clean$colname` – hierbei würde `R` nach einer Spalte mit dem Namen `colname` suchen, aber wir wollen hier ja stattdessen eigentlich den in der Variable `colname` enthaltenen Spaltennamen (etwa `Q23`) verwenden.

Der dritte Befehl im Körper der Schleife führt mit einem Aufruf der Funktion `ifelse` die eigentliche Umkodierung durch. Es wird kodiert, ob Probanden beim i'ten Item die narzisstische Aussage ausgewählt haben. Eine 1 wird vergeben, wenn das der Fall war, andernfalls eine Null. Ich speichere diesen numerischen Vektor aus Einsen und Nullen in der Variablen `narcissistic_response` zwischen. Beachtet, dass diese Variable in jedem Durchlauf der Schleife überschrieben wird (dasselbe gilt für die Variablen `colname`, `ith_item` und `new_colname`) .

Der vierte Befehl im Körper der Schleife generiert mit einem Aufruf der Funktion `paste0` in jedem Durchlauf einen neuen Spaltennamen. Die neuen Spalten haben Namen der Form `codedQ1`, `codedQ2` und so weiter.

Der fünfte Befehl im Körper der Schleife fügt mit der `[[·]]`-Notation die umkodierten Narzissmus-Werte als Spalte an `npi_clean` hinzu. So stelle ich sicher, dass ich auch später darauf zugreifen kann, etwa um Summenwerte über alle Items oder Item-Trennschärfen zu

bestimmen.

Beachtet, dass ich die `for`-Schleife auch mit weniger Zwischenschritten hätte umsetzen können; folgender Code würde in weniger Zeilen dasselbe Ergebnis erzielen:

```
for (i in 1:40) {
    colname <- paste0("Q", i)
    npi_clean[[paste0("coded", colname)]] <-
        ifelse(npi_clean[[colname]] == keys[i], 1, 0)
}
```

Im ersten Beispiel habe ich jedoch jeden Zwischenschritt in einer eigenen Variablen abgespeichert, um den Code besser verständlich zu machen und alle Schritte zu erklären. Aus meiner Sicht ist das Zwischenspeichern in Variablen gut geeignet, um zu kommunizieren, was Code macht – insbesondere, wenn die Variablen gut benannt sind.

6.2 Berechnung von part-whole korrigierten Trennschärfen

Nachdem ich mit der letzten `for`-Schleife die rohen Antwortdaten in die angemessene Form umkodiert habe, kann ich mit meiner Analyse starten. Ein wichtiger Teil einer Item-Analyse ist die Berechnung von Item-Trennschärfen. Dieser Abschnitt behandelt die Frage, wie wir mithilfe einer `for`-Schleife korrigierte Item-Trennschärfen für alle 40 Items des NPI berechnen können. Wir nutzen diesen Code:

```
## Wir nutzen die umkodierten NPI-Werte: speichere diese
## zunächst in einem separaten data.frame ab:
columns <- paste0("codedQ", 1:40)
items <- npi_clean[, columns]

## Berechne die Trennschärfen in Schleife
for (column in columns) {
    # 1. Summenwert unter Ausschluss eines Items
    scores <- rowSums(items[, column != colnames(items)])
    # 2. Korreliere damit den Item-Score
    part_whole <- cor(items[[column]], scores)
    # 3. Gib die Trennschärfe aus
    print(part_whole)
}
```

```
[1] 0.3558674
[1] 0.4201535
[1] 0.3321716
...
```

Mithilfe der Funktion `print` gebe ich nacheinander die 40 Trennschärfen aus, die ich in den Durchläufen der Schleife berechne; um die Seite nicht mit einer ausufernden Liste

an Trennschärfen zu fluten, habe ich hier aber nur drei davon angezeigt. Um ein Objekt während des Laufs einer Schleife in der Konsole auszugeben, muss man `print` explizit auf das auszugebende Objekt aufrufen; das Objekt ohne `print` anzusteuern würde in keiner Reaktion resultieren.

Nun aber zur Logik des Codes: Vor dem Durchlauf der Schleife wähle ich genau die Spalten aus `npi_clean` aus, die die zuvor erstellten Item-Scores enthalten und speichere sie in der Variable `items` ab. Danach startet die Schleife. In diesem Beispiel habe ich die Schleifenvariable `column` genannt. Das war recht willkürlich – ich kann der Schleifenvariable jeden Namen geben, den ich möchte. Hier habe ich mich anders als im vorherigen Beispiel nicht für den Namen `i` entschieden, da die Schleifenvariable keine Indexvariable ist und keine Sequenz von ganzen Zahlen der Form `1:n` durchläuft. Stattdessen durchläuft sie einen Vektor, der die Spaltennamen enthält, auf die ich zugreifen möchte. Deswegen erschien mir der Variablenname `column` passend.

Der erste Befehl im Schleifenkörper berechnet einen Summenwert über 39 Items. Dabei wird jeweils das Item nicht berücksichtigt, das in der Spalte `name` des von `items` abgespeichert ist. Betrachtet den Code genau: Mithilfe der `[·,·]`-Notation werden genau die 39 anderen Spalten ausgewählt. Dafür wird hinter dem Komma der `[·,·]`-Notation ein logischer Vektor der Länge 40 übergeben, der nur an einer Stelle `FALSE` enthält und sonst `TRUE`. Dieser logische Vektor wurde mit dem Befehl `column != colnames(items)` erstellt.

Der zweite Befehl im Schleifenkörper berechnet die korrigierte Trennschärfe. Hier wird der Summenscore über 39 Items mit dem verbleibenden Item korreliert und der resultierende Korrelationskoeffizient wird in der Variablen `part_whole` abgespeichert. Der dritte Befehl gibt lediglich die Trennschärfe in der Konsole aus.

6.3 Datenspeicherung in einer Schleife

Im vorherigen Beispiel habe ich Item-Trennschärfen berechnet und dann mit dem `print`-Befehl in der Konsole ausgegeben. Oftmals möchte ich die Ergebnisse von Berechnungen, die während einer Schleife anfallen, aber nicht nur ausgeben, sondern auch abspeichern. Im ersten Anwendungsbeispiel einer `for`-Schleife – Umkodierung von Items – hatten wir uns zunutze gemacht, dass man mit der `[[·]]`-Notation neue Spalten an an `data.frames` anhängen kann. Es macht jedoch keinen Sinn, die 40 Trennschärfen an den `data.frame` mit 10440 anzuhängen. Stattdessen könnte ich einen Vektor mit 40 Elementen zu erstellen, in dem ich die Trennschärfen speichere; ich berechne ja in jedem Durchlauf der Schleife genau einen Wert. Im Folgenden sehen wir uns an, wie wir das machen können. Dabei betrachten wir zwei Fälle: Einmal adressieren wir die Elemente des Vektors, der die Trennschärfen beinhaltet per Name und einmal per Index (siehe Kapitel 3.5).

6.3.1 Adressierung per Name

Zunächst erstelle ich wie folgt einen leeren[59] Vektor der Länge 40:

```
discriminations <- vector(length = 40)
```

Mithilfe der Funktion `vector`[60] erstelle ich einen Vektor; das Argument `length` bestimmt dabei die Länge des Vektors. Darin möchte ich im Verlauf der 40 Durchläufe der Schleife die Trennschärfen der 40 Items abspeichern. Um eine Adressierung per Name zu ermöglichen, gebe ich den Elementen des Vektors wie folgt Namen:

```
names(discriminations) <- paste0("codedQ", 1:40)
## Teste:
discriminations[1:3]
```

```
codedQ1 codedQ2 codedQ3
  FALSE   FALSE   FALSE
```

So haben die Elemente meines leeren Vektors dieselben Namen wie die Spalten des `data.frames items`, den ich zur Berechnung der Trennschärfen verwendet habe. Dass ich ausgerechnet diese Namen vergeben habe, hat zur Folge, dass ich recht einfach den obigen Code zur Berechnung der Trennschärfen umwandeln kann, um die Trennschärfen auch noch abzuspeichern. Dies ist der leicht angepasste Code:

```
## Wähle Items aus:
columns <- paste0("codedQ", 1:40)
items <- npi_clean[, columns]

## Erstelle leeren Vektor-Container und benenne ihn:
discriminations <- vector(length = 40)
names(discriminations) <- columns

for (column in columns) {
    # 1. Summenwert unter Ausschluss eines Items
    scores <- rowSums(items[, column != colnames(items)])
    # 2. Korreliere damit den Item-Score
    part_whole <- cor(items[[column]], scores)
    # 3. Speichere Trennschärfe ab
    discriminations[column] <- part_whole
}
## Voilá:
```

[59] Tatsächlich ist der Vektor nicht wirklich leer. Schaut ihn euch einmal nach der Erstellung an (d.h., gebt ihn in der Konsole aus).

[60] Es ist allgemein so, dass Funktionen mit dem Namen einer Datenstruktur besagte Datenstruktur erstellen. Erinnern wir uns an die Funktion `data.frame`. Ebenso gibt es die Funktion `list`, die eine Liste erstellt, oder die Funktion `matrix`, die eine Matrix erstellt. Diese Funktionen sind oft nützlich, um leere Datencontainer zu erstellen, die im Verlaufe einer `for`-Schleife gefüllt werden.

```r
head(discriminations)
```

```
  codedQ1   codedQ2   codedQ3   codedQ4   codedQ5   codedQ6
0.3558674 0.4201535 0.3321716 0.5012883 0.4414333 0.4790852
```

6.3.2 Vektorspeicherung – Adressierung per Index

Oftmals wird die Schleifenvariable als *Indexvariable* verwendet, d.h., sie durchläuft einen ganzzahligen numerischen Vektor, zumeist der Form `1:n`. So war es beispielsweise der Fall, als ich die 40 Items des NPI umkodiert habe. Diese Verwendung der Schleifenvariable ist oft dann nützlich, wenn ich in jedem Durchlauf der Schleife auf verschiedene Datenstrukturen zugreifen möchte – etwa auf einen `data.frame`, der Antworten enthält, und einen Vektor, der Schlüssel enthält. Da dieser Spezialfall wichtig ist, zeige ich auch für die Berechnung der Item-Trennschärfen, wie man die `for`-Schleife mit einer Index-Schleifenvariablen umsetzen kann:

```r
## Wähle Items aus:
columns <- paste0("codedQ", 1:40)
items <- npi_clean[, columns]

## Erstelle leeren Vektor-Container:
discriminations <- vector(length = 40)

## Erstelle Index-Vektor:
indices <- 1:40
## Berechne Trennschärfen in Schleife
for (i in indices) {
    # 1. Summenwert unter Ausschluss eines Items
    scores <- rowSums(items[, indices[-i]])
    # 2. Korreliere damit den Item-Score
    part_whole <- cor(items[[i]], scores)
    # 3. Speichere Trennschärfe ab
    discriminations[i] <- part_whole
}
## Voilá:
head(discriminations)
```

```
[1] 0.3558674 0.4201535 0.3321716 0.5012883 0.4414333 0.4790852
```

Hier wird die Index-Variable `i` gleich mehrfach verwendet: (1) zur Auswahl der Spalten, die den jeweiligen Testwert berechnen; (2) zur Auswahl der Spalte des Items, für das die Trennschärfe bestimmt wird; (3) zum Abspeichern der Trennschärfe im Vektor `discriminations`.

6.4 for-loops are evil – oder nicht?

Inhalt folgt.

7 Simulationen

Inhalt folgt.

8 Anhang

Dieser Abschnitt arbeitet einige Schwierigkeiten auf, die sich in den praktischen Übungen des Seminars ergeben haben.

8.1 Daten einlesen

Das Einlesen von Daten in `R` stellt uns vor verschiedene Probleme. Ich gehe an dieser Stelle auf ein grundlegendes Problem ein, das sich bei dem Einlesen jeglicher Daten stellt (egal ob man SPSS, Excel, csv, oder sonstige Dateien einliest): Woher weiß `R`, wo sich die Daten befinden, die ich einlesen möchte? Die Festplatte ist groß – `R` kann nur wissen, in welchem Ordner Daten liegen, wenn wir es `R` verraten.

Unsere Strategie: Wir verwenden RStudio-Projekte. Beachtet, dass dies nur eine von verschiedenen Möglichkeiten ist, mit dem "Dateisuchproblem" umzugehen. Aber es ist eben die, die wir nutzen. **Beachtet ebenfalls, dass das das Einzige ist, wofür wir RStudio Projekte nutzen: Wir legen RStudio Projekte an, um `R` mitzuteilen, wo es nach Daten suchen soll.** Bevor wir ein RStudio Projekt anlegen, müssen wir wissen, wo auf unserem Computer der Datensatz liegt. Wenn wir das wissen, legen wir in dem entsprechenden Ordner wie folgt ein Projekt an:

→ File → New project → Associate a project with an existing working directory → Browse → *Zum Ordner navigieren* → Open → Create Project

Nach dem Anlegen startet sich RStudio neu und unten rechts im Panel wird der Inhalt des Projekt-Ordners angezeigt. Wenn wir das Projekt gestartet haben, können wir Daten einlesen, die in diesem Ordner liegen. Dafür werden wir Funktionen aufrufen, die den Datensatz mit Dateinamen ansteuern. Folgender Aufruf etwa könnte eine csv-Datei einlesen und die Tabelle als `data.frame` in der Variablen `tp` speichern.

```
tp <- read.csv("technophobie.csv")
```

Wenn wir schon einmal ein Projekt im Ordner mit unseren Daten angelegt haben, können wir das Projekt beim nächsten Mal wieder aufrufen. Dafür gehen wir über

→ Open Project → *Zum Ordner navigieren* → *Projektdatei auswählen* (hat die Endung .Rproj) → *Öffnen*

8.2 Das Environment sauber halten

Wenn wir in `R` arbeiten, ist es wichtig, dass wir einen Überblick über die Variablen haben, die gerade existieren. Im Folgenden beschreibe ich ein paar grundlegende Strategien, um unsere `R`-Arbeitsumgebung einigermaßen sauber zu halten.

8.2.1 Variablen löschen

RStudio gibt uns in einem Panel oben rechts darüber Auskunft, welche Variablen sich in unserem sogenannten *Environment* befinden. Darin kommen alle Variablen vor, die wir irgendwann mit einer Zuweisung ("`<-`") erstellt haben. Um ein bisschen Ordnung zu halten, ist es nützlich zu wissen, wie man einzelne oder alle Variablen wieder entfernen kann. Es kann schnell passieren, dass man sehr viele Variablen erstellt, über die man sonst die Übersicht verliert.

Mit der Funktion `rm` kann man Variablen löschen, etwa:

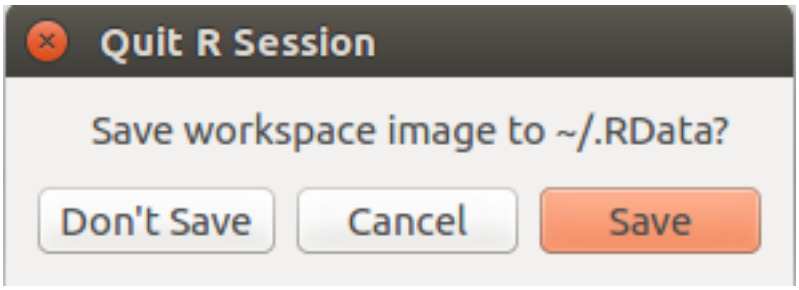
```
foo <- 1:10
rm(foo)
```

Möchte man alle Variablen aus dem Environment löschen, kann man den Befehl `rm(list = ls())` verwenden, etwa:

```
foo <- 1:10
bar <- 1:100
gaz <- mean(bar)
rm(list = ls()) # löscht alles, nur mit Vorsicht verwenden
```

8.2.2 Mit einem sauberen Environment starten

Wenn man RStudio beendet, wird einem von RStudio die Frage gestellt, ob man seinen "workspace" abspeichern will. Das kann etwa so aussehen, bei euch sieht es gegebenenfalls ein wenig anders aus:



Wenn man in diesem Fall zustimmt, wird im derzeitigen "working directory" – für uns heißt das: der Ordner unseres RStudio Projekts – eine Datei mit dem Namen ".RData" abgelegt. Diese Datei enthält alle Variablen, die sich derzeit in unserem Environment befinden. Also

alle Variablen, die uns oben rechts im Panel auch angezeigt werden. Wenn wir zustimmen und das Projekt aus dem Ordner neu laden, werden beim nächsten Mal alle Variablen unserer Session neu geladen. Ich rate stark davon ab, so zu arbeiten. Ich würde bevorzugen, **immer**[61] mit einem leeren Environment zu starten. Der einfachste Weg, um dies zu bewerkstelligen, ist immer "Don't save" auszuwählen, wenn man gefragt wird. Wenn man aus Versehen mal auf "Save" geklickt hat, kann man das Environment beim nächsten Start des Projekts mit dem Befehl `rm(list = ls())` wieder leeren. Auf Dauer hilft dann aber nur, die angelegte Datei im RStudio Projektordner zu löschen (diese wird vermutlich ".RData" heißen).

[61] Natürlich gibt es auch hier Ausnahmen. Wenn ihr selber einen Grund findet, aus dem es für euch doch gut ist, die Variablen abzuspeichern – etwa weil das Dateneinlesen sonst sehr lange dauert –, dann macht bitte das, was für euch sinnvoll ist.

9 Referenzen

Allaire, J., Xie, Y., McPherson, J., Luraschi, J., Ushey, K., Atkins, A., ... Chang, W. (2017). *Rmarkdown: Dynamic documents for r*. Retrieved from https://CRAN.R-project.org/package=rmarkdown

Fletcher, T. D. (2010). *Psychometric: Applied psychometric theory*. Retrieved from https://CRAN.R-project.org/package=psychometric

Li, Z., Lu, S., Myagmar, S., & Zhou, Y. (2006). CP-miner: Finding copy-paste and related bugs in large-scale software code. *IEEE Transactions on Software Engineering*, *32*(3), 176–192.

R Core Team. (2018). *R: A language and environment for statistical computing*. Vienna, Austria: R Foundation for Statistical Computing. Retrieved from https://www.R-project.org/

Raskin, R., & Terry, H. (1988). A principal-components analysis of the narcissistic personality inventory and further evidence of its construct validity. *Journal of Personality and Social Psychology*, *54*(5), 8–902.

Wickham, H., François, R., Henry, L., & Müller, K. (2018). *Dplyr: A grammar of data manipulation*. Retrieved from https://CRAN.R-project.org/package=dplyr

Xie, Y. (2015). *Dynamic documents with R and knitr* (2nd ed.). Boca Raton, Florida: Chapman; Hall/CRC. Retrieved from https://yihui.name/knitr/

Xie, Y. (2016). *Bookdown: Authoring books and technical documents with R markdown*. Boca Raton, Florida: Chapman; Hall/CRC. Retrieved from https://github.com/rstudio/bookdown