Testtheorie mit R

Martin Papenberg

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: 30. Juli 2018

Lizenz

Inhaltsverzeichnis

Chapter 1
Einstieg

Dieses Skript bietet einen Einstieg in die statistische Programmiersprache R. Es wurde als Begleitmaterial für eine ein-semestrige Lehrveranstaltung im Master-Studiengang Psychologie an der Heinrich-Heine-Universität Düsseldorf entworfen. Im Seminar wird kein Vorwissen über R vorausgesetzt. Ich habe das Skript öffentlich gemacht in der Hoffnung, dass es auch für andere R-Einsteiger nützlich sein kann. Es wird stetig aktualisiert; der aktuelle Stand ist jeweils der zweiten Seite zu entnehmen.

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.[1] Auch in der psychologischen Forschung wird R immer mehr zum Standard.[2]

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

[1] https://stackoverflow.blog/2017/10/10/impressive-growth-r/

[2] https://www.psychologicalscience.org/observer/why-you-should-become-a-user-a-brief-introduction-to-r

Über dieses Skript

Dieses Skript wurde als Begleitmaterial für eine Lehrveranstaltung konzipiert. Das Seminar selbst hat einen starken praktischen Anteil; in jeder Stunde werden Übungsaufgaben in R bearbeitet.[3] Das Skript bietet den theoretischen Unterbau zu den Übungen. Es wird empfohlen, das Skript parallel zu den Seminarstunden zu lesen.

Wenn man davor steht, R zu lernen, sollte man sich klar machen, dass die reine Aufarbeitung einer oder mehrerer schriftlicher Lektüren nicht ausreichend

[3] 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.

ist. 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.[4] Für so gut wie jede allgemeine Regel gibt es Spezialfälle, die eine Ausnahme bilden. Auf solche Spezialfälle werde ich bei der Beschreibung allgemeiner Grundsätze der Programmiersprache R nicht immer Rücksicht nehmen. Das Skript so ausgelegt, dass ein Grundstein an Kenntnissen gelegt wird, jedoch die Meisterung von R noch weitere eigenständige Einarbeitung erfordert.

[4] 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 internen "technischen" Kategorisierung von Daten in R.

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.

Credit

Zur Erstellung des Skripts wurden R (3.4.4, R Core Team, 2018) und die R-Pakete *bookdown* (0.5, Xie, 2016), *knitr* (1.18, Xie, 2015), *psychometric* (2.2, Fletcher, 2010), *rmarkdown* (1.8, Allaire et al., 2017), and *tufte* (0.2, Xie and Allaire, 2016) genutzt.

Ich danke Juliane Tkotz, Hanna Siegers, Marlene Wettstein und Frank Calio für ihre Rückmeldungen zu Fehlern in alten Versionen des Skripts.

Die Arbeitsumgebung

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.[5] 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.`

[5] Falls ihr eine andere Umgebung benutzt, ist das natürlich auch kein Problem. Ich selber benutze sogar nur selten RStudio. Alternativen sind beispielsweise rkward (`https://rkward.kde.org/`) oder emacs ESS (`https://ess.r-project.org/`).

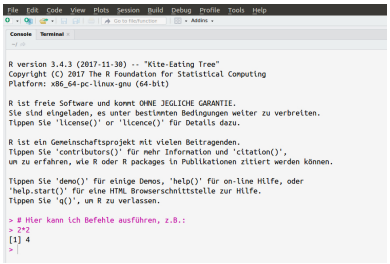
r-project.org/ gesondert heruntergeladen werden.

Hier könnt ihr beispielsweise über "Download R for Windows" → "install R for the first time" gehen. Wenn ihr RStudio und R heruntergeladen habt, startet RStudio und schreibt Folgendes in die R-Konsole und drückt `Enter`:

```
"Hallo Welt!"
```

Wenn folgende Ausgabe erscheint, hat die Installation funktioniert:

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



So sieht die R-Konsole in RStudio aus.

Die R-Konsole

Befehle können wir in R in die Konsole eingeben. Wir können das 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 uns R eine Fehlermeldung zurück. 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
```

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.[6] Um Analysen übersichtlich, nachvollziehbar und reproduzierbar zu gestalten, speichern wir unseren Code in sogenannten Quellcode-Dateien ab. Dafür gibt es in RStudio (und auch 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.

[6] 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.

Kommentare

Wenn ein #-Symbol in die Konsole oder den Skript-Editor geschrieben wird, wird der Rest dessen, was in dieser Zeile steht, nicht mehr interpretiert, (d.h.: 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,[7] den ich stets kommentiere. (Ich habe die Angewohnheit, ein doppeltes ## am Anfang einer Zeile zu benutzen, aber das hat

[7] 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).

keinerlei Bedeutung.) Gewöhnt euch ebenfalls an, **immer** euren eigenen Code zu kommentieren. Das gilt sowohl für "richtige" Projekte als auch für Übungsaufgaben. Das Kommentieren für 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.

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.

Chapter 2
Vektoren

Die einfachste und wichtigste Datenstruktur von R ist der *Vektor*. Ein Vektor ist beispielsweise eine einzelne Zahl wie in den Taschenrechner-Berechnungen oben. 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 gleichartiger Elemente, etwa Zahlen.[1] 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!

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

```
## 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:[2]

[2] 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)`[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] 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
```

Man kann 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:

[3] In den obigen Beispielen einfacher statistischer Berechnungen wird jeweils genau ein Argument übergeben, nämlich der Vektor, für den wir den Mittelwert, die Standardabweichung etc. berechnen wollten. Es ist auch möglich – und auch üblich –, dass Funktionen mehrere Argumente annehmen, die ihr Verhalten bestimmen. Die Funktion `plot` etwa verfügt über eine kaum überschaubare Menge an möglichen Argumenten, die verwendet werden können, um das Aussehen einer Abbildung zu spezifizieren.

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 viele 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 demnach jeweils ein Vektor der Länge 10. Bei gleich langen Vektoren werden solche Operationen im Allgemeinen **paarweise**[4] angewendet:

[4] Ich werde dazu oft auch *komponentenweise* sagen.

```
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".[5]

[5] 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 (gebt bei Interesse einmal die Befehle `c(1,2) * 1:4` und `c(1,2) * 1:3` in die R-Konsole ein).

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 "<-":

```
## Speichere 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:

```
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 einer Variablen gespeichert und weiter verwendet werden.

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

```
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 mit = Variablen zugewiesen.

Ausgabe versus Abspeichern

Wir haben bereits zwei verschiedene Möglichkeiten gesehen, Objekte[6] in R zu verwenden:

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

[6] Bis jetzt kennen wir nur das Vektor-Objekt. In R gibt es aber ganz verschiedene "Datencontainer", die man allgemein als Objekte bezeichnet.

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`. 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` auf der Konsole ausgeben:

```
bar
```

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

Offensichtlich hat `sort(bar)` den Vektor `bar` nicht geändert. Das ist eine fundamentale Eigenschaft von 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 will ich damit weiter rechnen.

Variablennamen

Generell bestehen Variablennamen in R aus Buchstaben und Zahlen und den Zeichen `.` und `_`. Folgende Einschränkungen sind zu beachten; wenn diese Anforderungen nicht berücksichtigt werden, wird R eine Variablenzuweisung nicht akzeptieren.

- 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
- Groß- / Kleinschreibung ist relevant (man sagt, dass Variablennamen in R "case-sensitive" sind)
  - "`bla <- 1`" ist nicht das Gleiche wie "`Bla <- 1`" oder gar "`BLA <- 1`"

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.[7]

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

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

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

Datentypen von Vektoren

In R hat jeder Vektor einen Datentyp. Bis jetzt haben wir nur mit Zahlen gearbeitet. Dieser Datentyp heißt in R "numeric". Der Datentyp eines Vektors bestimmt, was für Operationen wir damit durchführen können. Vektoren vom Typ "numeric" etwa kann man addieren, multiplizieren und so weiter. Es gibt weitere Datentypen, die wir benutzen, um unterschiedliche Informationen darzustellen.

`character`

`character` ist der Datentyp, der Text kennzeichnet. Text wird mit doppelten oder einfachen Anführungszeichen angegeben:

```
"Hallo Welt!"
```

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

```
mein_text <- 'bla bla bla'

## 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[8] und ergibt auch gar keinen Sinn, da man Text nicht mit einer Zahl multiplizieren kann:

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

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

In diesem Skript spielen Texte keine allzu große Rolle. Eine nützliche Funktion, die Vektoren vom Typ `character` generiert, sei hier jedoch kurz vorgestellt, da wir von ihr Gebrauch machen werden. Die Funktion `paste0` kann verwendet werden, um mehrere Vektoren als Text zusammenzufügen. Das wird nützlich sein, wenn wir in Datentabellen auf bestimmte Spalten zugreifen wollen. So lassen sich beispielsweise bequem 10 durchnummerierte Itemnamen als `character`-Vektor zusammenfügen:

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

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

```
length(items)
```

```
[1] 10
```

```
items
```

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

Mit der Funktion `mode` können wir überprüfen, dass der Vektor tatsächlich vom Typ `character` ist:[9]

[9] Die Ausgabe des Vektors macht uns auch schon eindeutig klar, dass es sich hier um einen Vektor vom Typ `character` handelt; die ausgegeben Elemente erscheinen nämlich in Anführungszeichen ("item_1", "item_2", ...)

```
mode(items)
```

```
[1] "character"
```

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

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

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

Wie wir sehen werden, können wir in Datentabellen mit der Funktion `paste0` Antworten auf gewünschte Items auswählen, da wir mit Textvektoren auf die Namen von Tabellenspalten zugreifen können.

`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:[10]

[10] https://de.wikipedia.org/wiki/Boolesche_Algebra#Zweielementige_boolesche_Algebra

```
## 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
```

Merke: Auch diese logischen Operationen 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
```

factor

`factor` Vektoren 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` kann genutzt werden, um numerische Werte in `factor` umzuwandeln. Dabei kann man die `levels` spezifizieren, d.h. die Werte, die der Vektor annimmt, **bevor** er in `factor` umgewandelt wird – hier `1`, `2` und `3`. `labels` wird verwendet, um anzugeben, wie die Faktorstufen angezeigt werden sollen. Das ist ähnlich den Wertelabels, die man in SPSS vergeben kann. Der Unterschied in R: Wenn ich eine Variable in `factor` umwandle, kann ich damit keine numerischen Berechnungen mehr durchführen. Für `laune_faktor` kann ich keinen Mittelwert mehr berechnen, da Vektoren vom Typ `factor` kategoriale Variablen darstellen:

```
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
```

NA

R hat einen eigenen Datentyp, um fehlende Werte zu kodieren: NA.[11] 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

[11] 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.

Daten in der psychologischen 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`[12] 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`):

[12] 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`.

```
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.

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. Diese Operationen sind fundamental für den weiteren Verlauf des Seminars. Es wird sich gegebenenfalls lohnen, bei den Themen Datenauswahl in Datentabellen noch einmal diesen Abschnitt zu konsultieren (in Kapitel 3). 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
```

```
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:[13]

[13] Für Vektoren vom Typ `factor` lässt sich auf dieselbe Art und Weise eine Abfrage auf Gleichheit umsetzen.

```
text1 <- "Hallo Welt"

text1 == "Hallo Welt"
```

```
[1] TRUE
```

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

```
[1] 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
```

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 "samplen". 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)
```

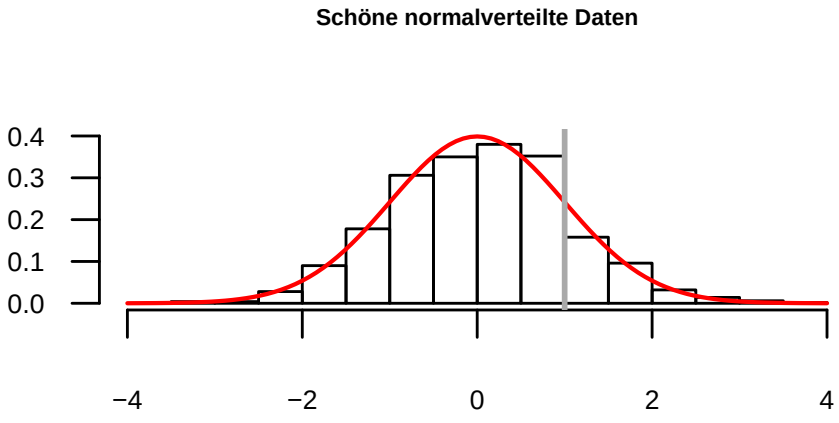
```
[1] -0.6259265 -1.7978535 -1.7894297
[4]  1.3217611 -1.2543220 -1.2287555
```

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] 154
```

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.**[14]

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

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

```
[1] 0.154
```

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:[15]

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

```
[1] 0.1586553
```

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(1e+05)  # 100000
mean(zufallsdaten > 1)
```

```
[1] 0.15807
```

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.

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. Allgemein ist die Möglichkeit Daten gezielt auszuwählen ein wichtiger Bestandteil von R, auch etwa um häufig Zeilen und Spalten aus Datentabellen auszuwählen, wie wir im nächsten Kapitel lernen werden.

[14] 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 * 2
[1] 2
```

[15] `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.)

Der `[·]`-Zugriff

Daten können mit dem `[·]`-Zugriff[16] *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 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 mehr-elementiger numerischer Vektor sein kann – hier nutzen wir die `c` Funktion:

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

```
[1] 2 4
```

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

```
[1] 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:

[16] 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 `c(4, 2, 6)[1]` wählt das erste Element aus dem Vektor `c(4, 2, 6)` aus, also 4. Der Punkt `·` in der `[·]`-Notation steht einfach als Platzhalter für einen Auswahlvektor hier.

```
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: welche Positionen des Vektors `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
 [7] FALSE FALSE FALSE FALSE FALSE FALSE
[13] FALSE FALSE FALSE FALSE FALSE  TRUE
[19]  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] 2.452844 2.211448 2.100921 2.504416
 [5] 2.735601 2.406681 2.006806 2.426910
 [9] 2.117478 2.476059
```

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.

`[·]`-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.[17] Wir wollen `R` mitteilen, diesen Wert als fehlend zu interpretieren. Hier kommt uns wiederum eine logische Abfrage zugute:

[17] 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.

```
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
 [7] 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.

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.

Zusammenfassung

- Wir haben `R`s 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

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?
4. Nutzt `paste0`, den `:`-Operator und den `[·]`-Negativ-Zugriff, um den folgenden Vektor zu erstellen:

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

5. In R haben Elemente eines Vektors nur einen Datentyp. `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?

Chapter 3
`data.frames`

Wir haben gelernt, dass R Daten in Vektoren abspeichert. Im Normalfall haben wir es aber in der psychometrischen Datenauswertung mit einer großen Menge Daten zu tun, die wir nicht einfach als einzelnen Vektor darstellen wollen. Etwa: 150 Studierende bearbeiten in einer Diagnostikklausur 42 Multiple-Choice-Klausuritems. Wir stellen solche Daten in Tabellen dar, wie wir sie auch aus Excel oder SPSS kennen. Spalten stellen Messvariablen dar, etwa die Punktzahlen in einer Klausuraufgabe. Zeilen stellen Fälle dar. Ein Fall könnte etwa eine Testteilnehmerin sein, für die in mehreren Spalten ihre Punktzahlen in allen Aufgaben abgespeichert sind.[1] In R speichert man solche Datentabellen in `data.frames` ab. Ein `data.frame` ist – vereinfacht gesagt – eine Sammlung von Vektoren; jede Spalte, d.h. jede Messvariable ist ein Vektor.

[1] Andere Formate sind auch denkbar, etwa eines in dem jede Zeile eine Aufgabe darstellt. Bei uns wird aber im Normalfall gelten: eine Zeile entspricht einem Fall – oftmals einer Person.

Die Funktion `data.frame`

Mit der Funktion `data.frame` kann ich "händisch" einen `data.frame` erstellen. In der Praxis werden wir das aber wohl nur selten machen und stattdessen Daten aus einer externen Datei einlesen.[2]

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

```
meinDataFrame <- 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"))
meinDataFrame
```

```
  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
```

Bei dieser Erstellung des `data.frames` wird deutlich, dass Spalten Vektoren enthalten, da wir für jede Spalte einen Vektor mit der Funktion `c` bzw. mit

dem :-Operator erstellen. Auch sehen wir, dass die Spalten bei der Erstellung des `data.frames` benannt werden können. **Dieser Punkt ist sehr wichtig, da wir Spalten anhand ihres Namens gezielt auswählen können**. Wenn ich die Spaltennamen, also die Namen der Messvariablen eines `data.frames` nicht weiß – etwa weil ich die Daten aus einer Datei eingelesen habe – kann ich diese mit dem Befehl `names` herausfinden:

```
names(meinDataFrame)
```

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

Diese unscheinbare Tabelle mit nur 5 Einträgen wird uns durch einen Großteil des Kapitels begleiten, um Grundlagen von `data.frame`-Operationen zu betrachten.

Zugriff auf Spalten in `data.frames`

Ich kann auf einzelne Spalten im `data.frame` mit der $-Notation zugreifen:

```
punkte <- meinDataFrame$Item1
punkte  # 'punkte' ist ein Vektor
```

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

Der $-Zugriff ist eine grundlegende Operation auf `data.frames`. Sie liest die Spalte eines `data.frames` als Vektor aus. Ich kann den $-Zugriff nicht nur verwenden, um Spalten aus einem `data.frame` auszulesen, sondern ich kann damit auch neue Spalten zum `data.frame` hinzufügen, indem in ich mit "<-" einer neuen Spalte einen Vektor zuweise:

```
meinDataFrame$Augenfarbe <- c("blau", "grau",
    "blau", "braun", "grün")
meinDataFrame
```

```
  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
  Augenfarbe
1       blau
2       grau
3       blau
4      braun
5       grün
```

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:

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

meinDataFrame$Testscore
```

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

Beachtet, dass in diesem Fall recht häufig die `$`-Notation zum Einsatz kommt, was etwas gewöhnungsbedürftig aussieht. Aber es ist wichtig darauf zu achten. Die Variablen,[3] die wir in diesem Beispiel verwenden, um den Testscore zu berechnen "wohnen" in `meinDataFrame` und können nicht ohne Verweis darauf adressiert werden. Das hier geht schief:

```
meinDataFrame$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 `meinDataFrame`.

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

```
mean(meinDataFrame$Alter)
```

```
[1] 13.4
```

```
table(meinDataFrame$Geschlecht)
```

```
m w
3 2
```

Die Funktion `mean` kennen wir bereits. Die Funktion `table` berechnet die Häufigkeiten von Werten, die in Vektoren vorkommen. Sie ist vor allem nützlich, um kategorielle Messvariablen zu beschreiben. Zur Überprüfung der Plausibilität von Daten ist `table` extrem nützlich. (Ist jeder Wert auch 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:

[3] Es ist etwas unglücklich, dass der Begriff "Variable" eine doppeldeutige Verwendung haben kann. Leider differenziere ich in diesem Skript auch nicht immer genau zwischen diesen Bedeutungen: (1) In `R` sind Variablen die Speicherorte von Objekten, die ich mit der "`<-`" Zuweisung erstelle. (2) Andererseits bezeichnet man auch Messwerte, etwa die Punktzahlen in einem Testitem, als Variable. In `R` würde man sich bei dieser Verwendung des Begriffs Variable dann auf die Spalte in einem `data.frame` beziehen. Diese Verwechslung ist unglücklich, da eine `data.frame` Spalte gar keine `R`-Variable ist. Stattdessen speichern wir den gesamten `data.frame` in **einer** Variablen ab.

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

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

Die Funktion `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(meinDataFrame$Testscore, meinDataFrame$Geschlecht,
    mean)
```

```
  m   w
1.0 1.5
```

```
tapply(meinDataFrame$Alter, meinDataFrame$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.[4] 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:

[4] 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.

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

```
  blau braun grau grün
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,[5] 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 5 – Datenpunkte hat. Das gilt gerade dann, wenn wir mehrere Gruppierungsvariablen angeben.

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

Daten auswählen: Die Funktion `subset`

Einzelne Spalten aus `data.frames` können wir mit dem `$`-Zugriff auslesen. Wir lernen nun die Funktion `subset` kennen, mit der wir bequem beliebige Spalten und Zeilen aus `data.frames` auswählen können. Anders als bei der Auswahl mit dem `$`-Operator – dessen Rückgabe ein Vektor ist –, gibt uns die Funktion `subset` immer einen ganzen `data.frame` zurück.

Mit `subset` können wir beispielsweise nur eine Teilmenge aller Fälle auswählen; etwa nur die Personen mit blauen Augen. Für diese Auswahl hilft uns unser Wissen über logische Vergleiche aus dem letzten Kapitel:[6]

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

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

Auf diese Weise haben wir mit einem logischen Vergleich aus der Tabelle nur zwei Zeilen ausgewählt. Ihr merkt: In der Funktion `subset` kann ich für die logische Auswahl nach Augenfarbe die Spalte `Augenfarbe` direkt mit ihrem Namen adressieren, ohne dass ich die `$`-Notation verwende. Das ist eine Besonderheit der Funktion `subset`; außerhalb der Funktion würde der Befehl `Augenfarbe == "blau"` einen Fehler ausgeben, da `Augenfarbe` selbst keine Variable ist – nur eine Spalte von `meinDataFrame`.[7] Innerhalb der Funktion `subset` funktioniert es nur deswegen, da das erste Argument der `data.frame` ist, aus dem ich Daten auswähle. Die Funktion `subset` weiß somit, auf welchen

[6] Beachtet, dass durch diesen Aufruf die Tabelle `meinDataFrame` nicht verändert wird. Die Funktion gibt stattdessen eine neue Tabelle zurück, die nur die Fälle enthält, bei denen `Augenfarbe == "blau"` gilt. Wir müssten das Ergebnis der Funktion in einer Variablen speichern, wenn wir damit weiter arbeiten wollen (Erinnerung: Kapitel 2).

[7] 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.

Daten sie operieren muss. Das Folgende ist also nicht nötig, obwohl es auch funktionieren würde:

```
subset(meinDataFrame, meinDataFrame$Augenfarbe == "blau")
```

Dementsprechend könnte man auch der Funktion `subset` – dieses Verhalten kennen wir von der `[·]`-Notation zur Auswahl von Elementen aus Vektoren – einen beliebigen logischen Vektor zur Auswahl der Zeilen übergeben:

```
subset(meinDataFrame, c(TRUE, FALSE, FALSE, TRUE,
    FALSE))  # wählt die erste und vierte Zeile aus
```

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

Durch die UND bzw. ODER Operationen können wir auch komplexere Anforderungen an die Auswahl stellen:

```
subset(meinDataFrame, Augenfarbe == "blau" | Augenfarbe ==
    "grün")
```

```
  Nummer Item1 Item2 Alter Geschlecht
1      1     1     1    13          w
3      3     0     0    13          m
5      5     1     1    15          m
  Augenfarbe Testscore
1       blau         2
3       blau         0
5       grün         2
```

```
subset(meinDataFrame, (Augenfarbe == "blau" |
    Augenfarbe == "grün") & Item1 == 1)
```

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

Wie schon erwähnt, können wir mit `subset` nicht nur Zeilen, sondern auch Spalten auswählen:

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

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

Hierbei habe ich mit dem dritten Argument `c('Item1', 'Augenfarbe')` eine Auswahl der Spalten durchgeführt. Dazu habe ich einen Vektor vom Typ `character` übergeben, der die auszulesenden Spalten mit Namen adressiert. Durch die Kombination der Auswahl von Zeilen und Spalten wird mir insgesamt ein `data.frame` ausgegeben, der nur die Spalten "Item1" und "Augenfarbe" enthält, und diese nur für Personen mit blauen Augen.[8]

Bei dieser Verwendung der Funktion `subset` fällt eine allgemeine Eigenschaft von Funktionen auf: `subset` erkennt anhand der Reihenfolge der Argumente, wie sie sich zu verhalten hat. Das erste Argument übergibt den `data.frame`, von dem wir Daten anfordern. Das zweite Argument wählt mit einem logischen Ausdruck **Zeilen** aus, das dritte Argument wählt durch einem `character`-Vektor **Spalten** aus. Was passiert, wenn wir diese Reihenfolge ändern?

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

Fehler in subset.data.frame(meinDataFrame,
c("Item1", "Augenfarbe"), Augenfarbe ==  : 'subset' muss
boolesch sein
```

Hier erhalten wir eine schwierig zu verstehende Fehlermeldung. Aber uns ist der Fehler klar: das zweite Argument von `subset` muss die Auswahl der Zeilen beschreiben, wir haben aber stattdessen einen `character`-Vektor übergeben, der die Spalten auswählen sollte. Was können wir machen, wenn wir **nur** eine Auswahl nach Spalte ausführen wollen? Wir können das zweite Argument ja nicht leer lassen, denn das führt zum obigen Fehler.

Dieses Problem lässt sich mit einer praktischen Eigenschaft der R-Sprache lösen: **In R haben die Argumente von Funktionen Namen.** Bislang haben wir das ignoriert bzw. nur am Rande mitbekommen (erinnern wir uns an das Argument `na.rm` der Funktion `mean`).

Die Funktion `subset` hat die folgenden drei benannten Argumente:

1. `x` – Der Datensatz, aus dem ausgewählt wird
2. `subset`: Die Auswahl der Zeilen
3. `select`: Die Auswahl der Spalten

[8] **Merke**: `subset` gibt immer einen `data.frame` zurück – selbst dann, wenn ich nur eine einzige Spalte anfordere. Mit dem `$`-Operator könnte ich hingegen eine einzelne Spalte als Vektor auslesen.

Um eine Übersicht über die verschiedenen Argumente einer Funktion zu erhalten, können wir die eingebaute Hilfe von R verwenden, die wir mit dem `?`-Operator erhalten. Wir verwenden sie wie folgt:

```
?subset
```

Die R-Hilfe informiert uns unter anderem über die Argumente, die Funktionen annehmen können. Leider ist diese 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 Hilfe-Funktion umgehen können.

Wenn wir die verschiedenen Argumente der Funktion `subset` kennen, können wir sie auch mit der folgenden Notation ausführen:

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

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

Hierbei benennen wir die Argumente, die wir nutzen, explizit. Wie wir es schon beim Argument `na.rm` der Funktion `mean` kennengelernt haben, können wir Argumente mit der Schreibweise `Funktionsargument = Wert` benannt adressieren. "`Wert`" ist dabei immer ein R-Objekt sein. Im Fall von `subset` nehmen die drei Argumente folgende Objekte an:[9]

1. `x` – ein `data.frame`
2. `subset`: Eine logischer Vektor
3. `select`: Ein Vektor vom Typ `character`

[9] Die Funktion `subset` lässt hier ein paar Ausnahmen zu, die weiter unten besprochen werden. Die erste Ausnahme kennen wir schon: Das Argument `subset` akzeptiert auch, wenn wir einen Ausdruck übergeben, der außerhalb der Funktion gar nicht als logischer Vektor erkannt würde.

Wenn ich Funktionsargumente mit Namen adressiere, kann ich die Reihenfolge, in der ich sie der Funktion übergebe, beliebig vertauschen. Dieser Aufruf etwa ist äquivalent (d.h. führt zur selben Ausgabe) wie der obige Aufruf:

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

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

In R kann man **fast immer**[10] Argumente per Position und per Name ansprechen. Oftmals wollen wir die Namen von Funktionen explizit verwenden, da

[10] Eine Ausnahme würde hier die Funktion `c` bilden, bei der wir keine Funktionsnamen angeben. Hier gilt nämlich: wir können beliebig viele Vektoren als Argumente angeben und deswegen gibt es natürlich keinen separaten Namen für jedes mögliche Argument. Feste Namen gibt es aber normalerweise, wenn es eine feste Anzahl an möglichen Argumenten gibt – wie bei der Funktion `subset`.

viele Funktionen *optionale* Argumente haben – also solche, die wir nicht immer angeben müssen. Wir können ja beispielsweise das Argument `select` weglassen, wenn wir nach Zeilen, aber nicht nach Spalten selegieren wollen. Analog muss ich nicht das Argument `subset` angeben – in dem Fall werden alle Zeilen ausgegeben, aber nur eine Teilmenge der Spalten, wie in diesem Beispiel:

```
subset(meinDataFrame, select = c("Testscore",
    "Geschlecht"))
```

```
  Testscore Geschlecht
1         2          w
2         1          m
3         0          m
4         1          w
5         2          m
```

Mit diesem Aufruf werden mir alle 5 Fälle zurückgegeben, aber für diese nur der Testscore und das Geschlecht. Wie dieser Aufruf zeigt, kann ich Auswahl nach Position und Auswahl nach Namen mischen. Für das erste Argument – `meinDataFrame` – habe ich den Argumentnamen `x` nicht angegeben. Daher wurde das Argument anhand der Position identifiziert. Das hat funktioniert, da das erste Argument der `data.frame` ist, aus dem die Datenauswahl stattfindet. Für die Auswahl der Spalten habe ich jedoch den Argumentnamen angegeben. **Das war auch nötig**, da `subset` als zweites Argument sonst 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.

Ausnahmeregeln für die Funktion `subset`

Inhalt folgt.

Fortgeschrittene Zugriffe

Achtung: Dieser Abschnitt beschäftigt sich tiefergehend mit Rs Möglichkeiten, auf Daten in `data.frames` zuzugreifen. Das ist ein delikates Thema, da umfangreich und für Einsteiger nicht unbedingt intuitiv. Ich kann mir vorstellen, dass die Inhalte dieses Abschnitts einen substantiellen Teil dessen ausmachen, was R schwierig für Einsteiger macht. Es ist absolut in Ordnung, beim Abschnitt Nützliche Funktionen zum Arbeiten mit `data.frames` fortzufah-

ren. Mutige können sich durch den Rest des Abschnitts durchkämpfen und bei akuter Verzweiflung zum nächsten Abschnitt wechseln.

Der `[[·]]`-Zugriff

Äquivalent zum `$`-Zugriff funktioniert der folgende `[[·]]` Zugriff auf eine Spalte:

```
punkte2 <- meinDataFrame[["Item1"]]
punkte2
```

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

Hierbei wird der Spaltenname als Text angesprochen. Das heißt, dass die Anführungszeichen notwendig sind, wenn man die `[[·]]`-Notation verwendet. Daraus ergibt sich, dass man statt der expliziten Angabe des Texts auch eine Variable übergeben kann, die einen ein-elementigen `character`-Vektor enthält. Das ist mit der der `$`-Notation nicht möglich.[11]

[11] Als ich zum ersten Mal diese Funktionalität – dass man eine Variable zum Zugriff mit der `[[·]]`-Notation verwenden kann – kennengelernt habe, war ich nicht sonderlich beeindruckt. Warum sollte das irgendeinen Vorteil gegenüber der `$`-Notation bringen? Tatsächlich gibt es dafür Anwendungsfälle und wir werden einen wichtigen in Kapitel 5 kennenlernen.

```
auswahl <- "Augenfarbe"
meinDataFrame[[auswahl]]
```

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

Es wäre auch möglich, eine Funktion in die `[[·]]`-Klammerung zu übergeben, die uns einen Text zurückgibt – etwa die Funktion `paste0`:

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

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

Dieser Zugriff wird für uns noch einmal interessant werden, da wir so mithilfe von *Schleifen*[12] (in Kapitel 5) in `data.frames` nacheinander auf beliebig viele Spalten zugreifen können. Dabei wird es vor allem interessant sein, nacheinander auf die Antworten auf Testitems (Item **1**, Item **2**, ...) zuzugreifen.

[12] In einer Schleife können wir dann den numerischen Wert – hier `1` – nacheinander immer wieder austauschen (1, 2, 3, 4, ...) – ohne, dass wir den Code immer wieder händisch neu schreiben müssen.

Der `[·]`-Zugriff

Nicht äquivalent zu den Zugriffen mit `$` und `[[·]]` ist folgender `[·]` Zugriff. Auch hier sind Anführungszeichen zur Identifikation der auszuwählenden Spalte nötig:

```
punkte3 <- meinDataFrame["Item1"]
punkte3
```

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

Der Unterschied von `[·]` zu `[[·]]` und `$`: `[[·]]` und `$` ergeben einen Vektor, `[·]` einen `data.frame`. Für uns bedeutet das, dass wir mit dem `[·]`-Zugriff auch gleichzeitig mehrere Spalten auswählen können, indem wir einen Vektor vom Typ `character` mit mehreren Elementen angeben. Hier hat die unscheinbare Funktion `c` noch mal einen Auftritt:

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

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

Dieser Ausdruck ist äquivalent zu

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

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

> **Merke:**
> Man kann mit `$`, `[[·]]` und `[·]` auf Spalten in `data.frames` zugreifen; dabei ergeben `$` und `[[·]]` einen Vektor, `[·]` einen `data.frame`.

Zugriff nach Name und Index

Es sei noch ein grundsätzliches Prinzip zu Datenzugriffen in `R` genannt: man kann Zugriffe in Daten – seien es Vektoren, `data.frames` oder auch andere Strukturen, die wir im Seminar gar nicht behandeln – **nach Index oder nach Name** durchführen. Wir haben bereits Beispiele für beides kennengelernt:

- In Vektoren haben wir Zugriffe mithilfe von Indexen durchgeführt, indem wir

  - die Position von Elementen explizit angegeben haben
  - oder indem wir einen logischen Vektor übergeben haben, der anhand von `TRUE` und `FALSE` Werten die Indexe auswählt, deren Elemente ausgegeben werden

- In `data.frames` haben wir Spalten nach Namen ausgewählt
  - Mit der `$`-Notation
  - Mit der `subset` Funktion
  - Mit der `[[·]]`-Notation
  - Mit der `[·]`-Notation

Es ist auch in `data.frames` möglich, Zugriffe nach Index durchzuführen. Es ist sogar so, dass wir in Vektoren Zugriffe nach Name durchführen können. Das gilt aber nur, wenn die Elemente Namen haben, was sie bei uns bislang nicht hatten (und in den meisten Fällen auch nicht nötig ist). Der Vollständigkeit halber sei hier mitgeteilt, wie man einen benannten Vektor erstellen kann und anhand der Namen auf Elemente zugreift:

```
## 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
```

Der `[·,·]`-Zugriff

Wie erwähnt, ist es auch in `data.frames` möglich, Zugriffe per Index durchzuführen. Das heißt für uns: Wir können beispielsweise die erste Spalte auswählen, ohne explizit den Namen der Spalte anzugeben. Das ist besonders nützlich,

wenn wir mit großen Datentabellen arbeiten. Um etwa Items des NPI zu "bepunkten", müssen wir Antworten aus 40 Spalten umkodieren. Dabei kann es nützlich sein nacheinander "links nach rechts" (also von der ersten zur letzten Spalte, d.h. 1, . . . , 40) alle Spalten per Index anzusprechen. Diese Funktionalität bietet uns der `[·,·]`-Operator. Dieser ist ein sehr mächtiges Werkzeug zur Bearbeitung von Daten in R. Unter anderem bietet er uns die Möglichkeit, die Funktionalität von `subset` zu reproduzieren.

Die Syntax zum Ansprechen von `data.frames` mit dem `[·,·]`-Operator ist 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, oder (b) ein logischer Vektor, der für jede Reihe/Spalte kodiert, ob diese in der Ausgabe enthalten sein soll (vgl. Kapitel 2), oder (c) ein "character" Vektor, der die Zeilen/Spalten, die ausgegeben werden sollen, nach Namen auswählt.[13]

[13] Es ist möglich, dass auch Zeilen Namen haben. Häufig sind Zeilen aber nur nummeriert und nicht explizit benannt – wie es bei uns bislang immer der Fall war.

Es ist möglich, dass entweder der Spaltenvektor oder der Reihenvektor leer ist; in dem Fall findet die Auswahl nur nach Reihe bzw. Spalte statt. Das ist analog dazu, dass wir mit der Funktion `subset` eines der Argumente `subset` oder `select` auslassen. Das führt zu einer gewöhnungsbedürftig aussehenden Syntax:

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

Tatsächlich wird man häufig nur entweder nach Spalte oder nach Zeile auswählen und nicht unbedingt beides kombinieren. Wie und ob ich an dieser Stelle vor oder nach dem Komma Leerzeichen setze, hat keine Bedeutung.

Im Folgenden finden sich Beispiele für die verschiedenen Auswahlmöglichkeiten per `[·,·]`. Wir verwenden weiterhin die Tabelle `meinDataFrame`

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

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

```
## Wähle per Index die zweite und vierte Spalte
## aus
meinDataFrame[, 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:
meinDataFrame[meinDataFrame$Testscore == 2, ]
```

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

```
## Wähle alle Personen aus, die blaue oder
## braune Augenfarbe haben:
meinDataFrame[meinDataFrame$Augenfarbe == "blau" |
    meinDataFrame$Augenfarbe == "braun", ]
```

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

```
## Wähle Fallnummer, Alter und Testscore per
## Spaltenname aus:
meinDataFrame[, 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
meinDataFrame[meinDataFrame$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
meinDataFrame[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
meinDataFrame[, paste0("Item", 1:2)]
```

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

Einige der hier genannten Auswahlen können wir auch mit der Funktion `subset` durchführen. In diesem Seminar werden wir `subset` verwenden statt den `[·,·]`-Zugriff; `subset` reicht für unsere Zwecke aus und führt oft zu besser lesbarem Code.

Merke:
Mit dem `[·,·]` Zugriff wird zuerst – vor dem Komma – die Reihe und dann – nach dem Komma – die Spalte adressiert. Man kann die Auswahl nach numerischem Index, mit einem logischen Vektor, oder mit einem `character` Vektor durchführen.

Abschließende Bemerkungen zu Zugriffen

Datenzugriffe mit der `[ ]`-Familie sind etwas, das bei R-Anfängern regelmäßig zu Kopfschmerzen führt. Diese Zugriffe sind jedoch zentral für R als Programmiersprache, weswegen man – früher oder später – nicht daran vorbeikommt. Insbesondere wenn man mit anderen Datenstrukturen – wie Matrizen oder Listen – arbeitet, wird man auf diesen Abschnitt zurückkommen müssen. Denn: In Matrizen werden Zugriffe mit der `[·,·]`-Notation durchgeführt, in Listen können auch Daten mit dem `[[·]]`- oder dem `[·]`-Zugriff ausgewählt werden.

Für uns ist der Abschnitt "Fortgeschrittene Zugriffe" aber nur als Zusatzinfo beziehungsweise Nachschlagsmöglichkeit gedacht. Wir werden zunächst nur mit dem `$` Zugriff und der Funktion `subset` arbeiten, die für unsere Zwecke ausreichend sind. In Kapitel 5 wird uns jedoch noch einmal der `[[·]]`-Zugriff zur sequentiellen Auswahl von Spalten in `data.frames` begegnen.

Nützliche Funktionen zum Arbeiten mit `data.frames`

`nrow` und `ncol`

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

```
nrow(meinDataFrame)
```

```
[1] 5
```

Analog ergibt `ncol` die Zahl der Spalten:

```
ncol(meinDataFrame)
```

```
[1] 7
```

`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 6 Zeilen ausgegeben (in R-Jargon: 6 ist der "default"-, also Standardwert des *optionalen Arguments* n). Beispiel:

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

```
  Nummer Item1 Item2 Alter Geschlecht
1      1     1     1    13         w
```

```
2      2     0     1    14          m
  Augenfarbe Testscore
1       blau         2
2       grau         1
```

```
tail(meinDataFrame)
```

```
  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
  Augenfarbe Testscore
1       blau         2
2       grau         1
3       blau         0
4      braun         1
5       grün         2
```

Im letzteren Fall werden einfach alle Zeilen zurückgegeben, da unser `data.frame` insgesamt nur fünf Zeilen hat – und somit weniger als 6.

Sortieren: `dplyr::arrange`

Oftmals wollen wir Datentabellen nach einer oder mehreren Variablen sortieren. Dies funktioniert am bequemsten, wenn wir das *Paket* `dplyr` laden:

```
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) , könnte ich es mit dem folgenden Befehl installieren:

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

Pakete stellen zusätzliche Funktionen zu 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(meinDataFrame, Testscore)  # dplyr muss geladen sein
```

```
  Nummer Item1 Item2 Alter Geschlecht
1      3     0     0    13          m
2      2     0     1    14          m
3      4     1     0    12          w
4      1     1     1    13          w
5      5     1     1    15          m
  Augenfarbe Testscore
1       blau         0
2       grau         1
3      braun         1
4       blau         2
5       grün         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(meinDataFrame, -Testscore)
```

```
  Nummer Item1 Item2 Alter Geschlecht
1      1     1     1    13          w
2      5     1     1    15          m
3      2     0     1    14          m
4      4     1     0    12          w
5      3     0     0    13          m
  Augenfarbe Testscore
1       blau         2
2       grün         2
3       grau         1
4      braun         1
5       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(meinDataFrame, Geschlecht, -Testscore)
```

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

```
4      1     1     1    13          w
5      4     1     0    12          w
  Augenfarbe Testscore
1       grün         2
2       grau         1
3       blau         0
4       blau         2
5      braun         1
```

Zusammenfassung

- Wir haben den `data.frame` als Datenstruktur zur Speicherung von psychometrischen Daten kennengelernt
- Wir haben den Zugriff auf Spalten und Zeilen in `data.frames` mit der `$`-Notation und der Funktion `subset` kennengelernt
- 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`

Abschließender Hinweis

`vector` und `data.frame` sind die einzigen Datenstrukturen, mit denen wir zunächst arbeiten werden. Früher oder später wird man sich mit jedoch mit weiteren Datenstrukturen aus 'R' auseinandersetzen müssen. Einige wichtige seien deswegen schon einmal an dieser Stelle genannt:

- `matrix`: von der Struktur her wie ein `data.frame`, aber alle Werte müssen denselben Datentyp haben
- `list`: ein „Container", der beliebige andere Daten enthalten kann; Zugriffe funktionieren zumeist wie in einem `data.frame`, da `data.frames` eigentlich selber Listen sind
- `array`: eine mehrdimensionale Matrix; etwa `tapply` kann einen Array zurückgegeben

Häufig kann man die *Klasse*, also die Art der Datenstruktur eines Objekts, mit der Funktion `class` herausfinden (probiert den Befehl `class(data.frame(foo = 1:3))`)

Fragen zum vertiefenden Verständnis

1. 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?

```
subset(meinDataFrame, select = "Item1", Augenfarbe == "blau")
```

```
subset(select = "Item1", meinDataFrame, Augenfarbe == "blau")
```

```
subset(meinDataFrame, "Item1", Augenfarbe == "blau")
```

```
subset("Item1", meinDataFrame, Augenfarbe == "blau")
```

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

```
subset(meinDataFrame, select = "Item1")
```

```
meinDataFrame["Item1"]
```

```
meinDataFrame[ ,"Item1"]
```

```
meinDataFrame[ ,"Item1", drop = FALSE]
```

```
meinDataFrame[["Item1"]]
```

```
meinDataFrame$Item1
```

Chapter 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 folgende 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

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 selber 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,[1] 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!

[1] **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.

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)  # test_data ist die Tabelle von oben.
```

```
 [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)
```

Item-Schwierigkeiten

Die Schwierigkeit eines Items ist die mittlere Punktzahl aller Personen in diesem Item.[2] 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`):

[2] 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.

```
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, kriege ich die mittlere Punktzahl der Schüler/innen in den 5 Testitems direkt mitgeliefert. Beachtet, dass ich hier eine numerische Funktion auf den ganzen `data.frame` angewendet habe. Hätte ich beispielsweise auch Spalten vom Typ `factor` oder `numeric` im `data.frame` gehabt, hätte ich Funktionen wie `rowSums` und `colMeans` nicht einfach auf den ganzen `data.frame` anwenden können.[3]

[3] In dem Fall könnte man mit `subset` nur die gewünschten Spalten auswählen.

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 auf zwei Nachkommastellen gerundet, was mit der Funktion `round` erreicht wurde.

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.[4] Eine Möglichkeit, die "part-whole" korrigierte Trennschärfe für eine Item (hier: Item 2) zu berechnen, bietet der folgende Code:

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

```
## Zunächst erstelle ich einen Vektor zur
## Auswahl der Items, die ich zur Berechnung
## des Testscores heranziehe. Dabei wird Item 2
## ausgelassen. An dieser Stelle müssen wir die
## Namen der Spalten kennen, die wir auswählen
## wollen. Diese lassen sich mit dem Befehl
## 'names' herausfinden:

names(test_data)
```

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

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

## Betrachte die Tabelle:
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
7       1       1       0       0
8       1       0       0       0
9       1       1       0       0
10      0       0       0       0
```

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

`corrected_score` ist nun der Testscore ohne Beachtung des zweiten Items. Das Vorgehen zur Berechnung der bereinigten Scores mithilfe der Funktionen `paste0`, `subset` und `rowSums` lässt sich allgemein mit beliebig vielen Items durchführen. Da wir an dieser Stelle nur eine Summe über vier Items bilden, hätte auch der folgende – simplere – Code funktioniert:

```
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 "part-whole" korrigierte Trennschärfe für Item 2 bestimmen:

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

```
[1] 0.5238095
```

Wie wir sehen, liegt die korrigierte Trennschärfe von 0.52 unter der unkorrigierten Trennschärfe von 0.65. Je weniger Items der Test hat, desto mehr Gewicht hat das einzelne Item für den Testscore, und umso stärker weichen korrigierte und unkorrigierte Trennschärfe voneinander ab. Bei nur 5 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:

Merke: Man kann `cor` statt eines `data.frames` auch zwei gleich lange Vektoren übergeben, die beispielsweise die Punktzahlen in zwei Tests enthalten. Dann berechnet `cor` die Korrelationen zwischen den Punktzahlen.

- 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 Funktion `subset`

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.

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 Punktzahl 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.[5] 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. Mit der Funktion `library` kann ich Pakete laden, die nicht zur Grundausstattung von R gehören.[6] Voraussetzung ist, dass ich das Paket auf meinem Rechner installiert habe.

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

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

```
## 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:[7]

```
test_data$score <- NULL

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

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

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
```

[7] 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.

Split-Half-Reliabilität

Cronbachs Alpha ist ein Schätzer für die Reliabilität eines Tests.[8] 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.

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

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 <- subset(test_data, select = paste0("Item_",
    1:3))
second_half <- subset(test_data, select = paste0("Item_",
    4:5))

## Berechne die Korrelation zwischen den beiden
## Testhälften
cor_halfs <- cor(rowSums(first_half), rowSums(second_half))
cor_halfs
```

```
[1] 0.5091751
```

```
## 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) ein
## Reliabilitäts-Schätzer; (b) ein
```

```
## Verlängerungsfaktor

spearman_brown <- function(reliability, factor) {
    corrected_reliability <- (reliability * factor)/(1 +
        (factor - 1) * reliability)
    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
```

```
## 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.

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

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` $\rightarrow$ `0`
`2` $\rightarrow$ `1`
`3` $\rightarrow$ `0`
`4` $\rightarrow$ `0`

In psychometrischem Jargon: Für diese Aufgabe ist der Wert `2` der *Schlüssel* (engl.: *key*). Ein Schlüssel kodiert den Eingabewert der richtigen Antwort.[9] 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 viele Items wiederholen. Das *Narcissistic Personality Inventory* etwa hat 40 Items und wir haben keine Lust, 40 Mal eine Umkodierung "händisch" neu durchzuführen.

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

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`:[10]

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

- `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 herleiten.

`ifelse` ist eine sehr nützliche Funktion, 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.

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:

Invertierter Wert = Ursprungswert * (-1) + Höchster Skalenwert + 1

Diese funktioniert, wenn unsere Punktzahlen zwischen 1 und dem höchst möglichen 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:

```
subset(big5, select = 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.

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[11]

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

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 and Terry, 1988) ein. Die Daten von mehr als 11,000 Bearbeitungen des NPI sind erfreulicherweise abrufbar über das "Open Source Psychometrics Project" unter `https://openpsychometrics.org/_rawdata`. 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
```

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

```
[1] 44
```

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

```
 [1] "score"  "Q1"     "Q2"     "Q3"
 [5] "Q4"     "Q5"     "Q6"     "Q7"
 [9] "Q8"     "Q9"     "Q10"    "Q11"
[13] "Q12"    "Q13"    "Q14"    "Q15"
[17] "Q16"    "Q17"    "Q18"    "Q19"
[21] "Q20"    "Q21"    "Q22"    "Q23"
[25] "Q24"    "Q25"    "Q26"    "Q27"
[29] "Q28"    "Q29"    "Q30"    "Q31"
[33] "Q32"    "Q33"    "Q34"    "Q35"
[37] "Q36"    "Q37"    "Q38"    "Q39"
[41] "Q40"    "elapse" "gender" "age"
```

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

```
  score Q1 Q2 Q3 Q4 Q5 Q6 Q7 Q8 Q9 Q10 Q11
1    18  2  2  2  2  1  2  1  2  2   2   1
2     6  2  2  2  1  2  2  1  2  1   1   2
3    27  1  2  2  1  2  1  2  1  2   2   2
  Q12 Q13 Q14 Q15 Q16 Q17 Q18 Q19 Q20 Q21
1   1   2   1   1   1   2   1   1   1   1
2   2   2   1   2   2   1   1   2   1   2
3   1   1   1   1   1   2   2   1   1   2
  Q22 Q23 Q24 Q25 Q26 Q27 Q28 Q29 Q30 Q31
1   1   1   2   2   2   1   2   2   2   1
2   2   1   2   2   2   2   1   2   2   2
3   2   2   2   1   2   1   1   2   1   2
  Q32 Q33 Q34 Q35 Q36 Q37 Q38 Q39 Q40 elapse
1   2   1   1   1   2   2   2   1   2    211
2   1   2   2   1   2   2   2   2   1    149
3   2   1   1   2   1   1   2   1   2    168
  gender age
1      1  50
2      1  40
3      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.[12] Für die Variable `score` erhalten wir folgende Informationen zum Narzissmus-Gesamtscore:

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

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

Identifikation von fehlenden Werten im NPI Datensatz

Das NPI besteht aus 40 Items. Aus dem *Codebuch* des NPI Datensatzes[13] 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:

[13] Dieses wird gemeinsam mit den Daten vom "Open Source Psychometrics Project" `https://openpsychometrics.org/_rawdata` runtergeladen.

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:[14]

[14] **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.

```
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.[15]

[15] 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.

Ausschluss von Fällen mit fehlenden Werten

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 <- subset(npi, select = 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:[16]

[16] 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").

```
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:[17]

[17] 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`.

```
## 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:

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` 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 <- subset(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`[18] `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 Grund funktioniert das hier:

[18] siehe Abschnitt Ausgedehntes Beispiel zum Einstieg

```
## 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:[19]

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

```
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"
 [5] "Q4"      "Q5"      "Q6"      "Q7"
 [9] "Q8"      "Q9"      "Q10"     "Q11"
[13] "Q12"     "Q13"     "Q14"     "Q15"
[17] "Q16"     "Q17"     "Q18"     "Q19"
[21] "Q20"     "Q21"     "Q22"     "Q23"
[25] "Q24"     "Q25"     "Q26"     "Q27"
[29] "Q28"     "Q29"     "Q30"     "Q31"
[33] "Q32"     "Q33"     "Q34"     "Q35"
[37] "Q36"     "Q37"     "Q38"     "Q39"
[41] "Q40"     "elapse"  "gender"  "age"
[45] "casenum"
```

```
names(item_responses)
```

```
 [1] "casenum" "Q1"      "Q2"      "Q3"
 [5] "Q4"      "Q5"      "Q6"      "Q7"
 [9] "Q8"      "Q9"      "Q10"     "Q11"
[13] "Q12"     "Q13"     "Q14"     "Q15"
[17] "Q16"     "Q17"     "Q18"     "Q19"
[21] "Q20"     "Q21"     "Q22"     "Q23"
[25] "Q24"     "Q25"     "Q26"     "Q27"
[29] "Q28"     "Q29"     "Q30"     "Q31"
[33] "Q32"     "Q33"     "Q34"     "Q35"
[37] "Q36"     "Q37"     "Q38"     "Q39"
[41] "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"
 [5] "Q5"      "Q6"      "Q7"      "Q8"
 [9] "Q9"      "Q10"     "Q11"     "Q12"
[13] "Q13"     "Q14"     "Q15"     "Q16"
[17] "Q17"     "Q18"     "Q19"     "Q20"
[21] "Q21"     "Q22"     "Q23"     "Q24"
[25] "Q25"     "Q26"     "Q27"     "Q28"
[29] "Q29"     "Q30"     "Q31"     "Q32"
[33] "Q33"     "Q34"     "Q35"     "Q36"
[37] "Q37"     "Q38"     "Q39"     "Q40"
[41] "casenum" "score"   "elapse"  "gender"
[45] "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.[20] 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:

[20] 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.

```
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.[21] Dabei ist unser nächstes Ziel für alle 40 Items eine dichotome Bepunktung 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:

[21] 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.

```
# 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 5 lernen, diese Umkodierungen effizient durchzuführen. Anschließend werden wir die psychometrischen Eigenschaften des NPI untersuchen.

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

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)?

Chapter 5
Einführung in die Programmierung mit R

Inhalt folgt.

Chapter 6
Anhang

Dieser Abschnitt arbeitet einige Schwierigkeiten auf, die sich in den praktischen Übungen des Seminars ergeben haben.

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*

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.

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 `rm()` kann man Variablen löschen, etwa:

```
foo <- 1:10
rm(foo)
```

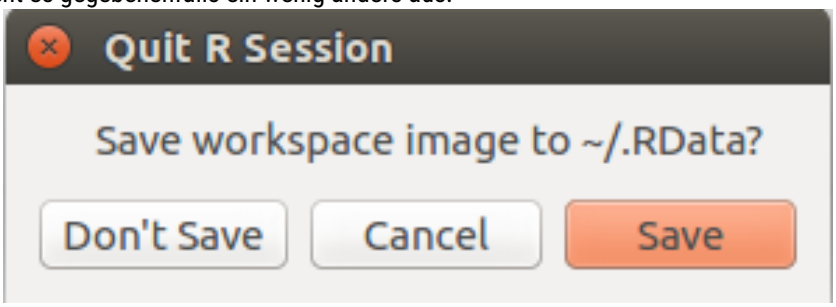
Möchte man alle Variablen aus dem Environment löschen, kann man den Befehl `rm(list = ls())` verwenden, etwa:

```
foo <- 1:10
bar <- 1:100
gaz <- mean(bar)
rm(list = ls()) # löscht alles, nur mit Vorsicht verwenden
```

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**[1] 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).

[1] 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.

Chapter 7
Literaturverzeichnis

JJ Allaire, Yihui Xie, Jonathan McPherson, Javier Luraschi, Kevin Ushey, Aron Atkins, Hadley Wickham, Joe Cheng, and Winston Chang. *rmarkdown: Dynamic Documents for R*, 2017. URL https://CRAN.R-project.org/package=rmarkdown. R package version 1.8.

Thomas D. Fletcher. *psychometric: Applied Psychometric Theory*, 2010. URL https://CRAN.R-project.org/package=psychometric. R package version 2.2.

R Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2018. URL https://www.R-project.org/.

Robert Raskin and Howard Terry. 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, 1988.

Yihui Xie. *Dynamic Documents with R and knitr*. Chapman and Hall/CRC, Boca Raton, Florida, 2nd edition, 2015. URL https://yihui.name/knitr/. ISBN 978-1498716963.

Yihui Xie. *bookdown: Authoring Books and Technical Documents with R Markdown*. Chapman and Hall/CRC, Boca Raton, Florida, 2016. URL https://github.com/rstudio/bookdown. ISBN 978-1138700109.

Yihui Xie and JJ Allaire. *tufte: Tufte's Styles for R Markdown Documents*, 2016. URL https://CRAN.R-project.org/package=tufte. R package version 0.2.